User Guide

Amazon Bedrock

Amazon Bedrock: User Guide

Table of Contents

What is Amazon Bedrock?

Amazon Bedrock is a fully managed service that makes high-performing foundation models (FMs) from leading AI startups and Amazon available for your use through a unified API. You can choose from a wide range of foundation models to find the model that is best suited for your use case. Amazon Bedrock also offers a broad set of capabilities to build generative AI applications with security, privacy, and responsible AI. Using Amazon Bedrock, you can easily experiment with and evaluate top foundation models for your use cases, privately customize them with your data using techniques such as fine-tuning and Retrieval Augmented Generation (RAG), and build agents that execute tasks using your enterprise systems and data sources.

With Amazon Bedrock's serverless experience, you can get started quickly, privately customize foundation models with your own data, and easily and securely integrate and deploy them into your applications using AWS tools without having to manage any infrastructure.

Topics

Features of Amazon Bedrock

Take advantage of Amazon Bedrock foundation models to explore the following capabilities. To see feature limitations by Region, see Model support by AWS Region.

- **Experiment with prompts and configurations** – Run model inference by sending prompts using different configurations and foundation models to generate responses. You can use the API or the text, image, and chat playgrounds in the console to experiment in a graphical interface. When you're ready, set up your application to make requests to the `InvokeModel` APIs.
- **Augment response generation with information from your data sources** – Create knowledge bases by uploading data sources to be queried in order to augment a foundation model's generation of responses.

- **Create applications that reason through how to help a customer** – Build agents that use foundation models, make API calls, and (optionally) query knowledge bases in order to reason through and carry out tasks for your customers.
- **Adapt models to specific tasks and domains with training data** – Customize an Amazon Bedrock foundation model by providing training data for fine-tuning or continued-pretraining in order to adjust a model's parameters and improve its performance on specific tasks or in certain domains.
- **Improve your FM-based application's efficiency and output** – Purchase Provisioned Throughput for a foundation model in order to run inference on models more efficiently and at discounted rates.
- **Determine the best model for your use case** – Evaluate outputs of different models with built-in or custom prompt datasets to determine the model that is best suited for your application.

> **Note**
>
> Model evaluation is in preview release for Amazon Bedrock and is subject to change.

- **Prevent inappropriate or unwanted content** – Use **Guardrails for Amazon Bedrock** to implement safeguards for your generative AI applications.

> **Note**
>
> Guardrails for Amazon Bedrock is in limited preview release. To request access, contact your AWS account manager.

Amazon Bedrock pricing

When you sign up for AWS, your AWS account is automatically signed up for all services in AWS, including Amazon Bedrock. However, you are charged only for the services that you use.

To see your bill, go to the Billing and Cost Management Dashboard in the AWS Billing and Cost Management console. To learn more about AWS account billing, see the *AWS Billing User Guide*. If you have questions concerning AWS billing and AWS accounts, contact AWS Support.

With Amazon Bedrock, you pay to run inference on any of the third-party foundation models. Pricing is based on the volume of input tokens and output tokens, and on whether you have purchased provisioned throughput for the model. For more information, see the Model providers page in the Amazon Bedrock console. For each model, pricing is listed following the model version. For more information about purchasing Provisioned Throughput, see Provisioned Throughput for Amazon Bedrock.

For more information, see Amazon Bedrock Pricing.

Supported AWS Regions

For information about service endpoints for Regions that Amazon Bedrock supports, see Amazon Bedrock endpoints and quotas.

To see what foundation models each Region supports, refer to Model support by AWS Region.

> **Note**
>
> All features are available in US East (N. Virginia) and US West (Oregon).

See the following table for features that are limited by region.

Region	Model evaluation	Knowledge base	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
US East (N. Virginia)	Yes	Yes	Yes	Yes	Yes	Yes
US West (Oregon)	Yes	Yes	Yes	Yes	Yes	Yes
Asia Pacific (Singapore)	No	No	No	No	No	No
Asia Pacific (Sydney)	No	No	No	No	No	Yes

Region	Model evaluation	Knowledge base	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
Asia Pacific (Tokyo)	No	No	No	No	No	No
Europe (Frankfurt)	No	No	No	No	No	No
Europe (Paris)	No	No	No	No	No	Yes
AWS GovCloud (US-West)	No	No	No	Yes	No	Yes (only for fine-tuned models, with no commitment term)

Key definitions

This chapter provides definitions for concepts that will help you understand what Amazon Bedrock offers and how it works. If you are a first-time user, you should first read through the basic concepts. Once you familiarize yourself with the basics of Amazon Bedrock, we recommend for you to explore the advanced concepts and features that Amazon Bedrock has to offer.

Basic concepts

The following list introduces you to the basic concepts of generative AI and Amazon Bedrock's fundamental capabilities.

- **Foundation model (FM)** – An AI model with a large number of parameters and trained on a massive amount of diverse data. A foundation model can generate a variety of responses for a wide range of use cases. Foundation models can generate text or image, and can also convert input into *embeddings*. Before you can use an Amazon Bedrock foundation model, you must request access. For more information about foundation models, see Supported foundation models in Amazon Bedrock.

- **Base model** – A foundation model that is packaged by a provider and ready to use. Amazon Bedrock offers a variety of industry-leading foundation models from leading providers. For more information, see Supported foundation models in Amazon Bedrock.
- **Model inference** – The process of a foundation model generating an output (response) from a given input (prompt). For more information, see Run model inference.
- **Prompt** – An input provided to a model to guide it to generate an appropriate response or output for the input. For example, a text prompt can consist of a single line for the model to respond to, or it can detail instructions or a task for the model to perform. The prompt can contain the context of the task, examples of outputs, or text for a model to use in its response. Prompts can be used to carry out tasks such as classification, question answering, code generation, creative writing, and more. For more information, see Prompt engineering guidelines.
- **Token** – A sequence of characters that a model can interpret or predict as a single unit of meaning. For example, with text models, a token could correspond not just to a word, but also to a part of a word with grammatical meaning (such as "-ed"), a punctuation mark (such as "?"), or a common phrase (such as "a lot").
- **Model parameters** – Values that define a model and its behavior in interpreting input and generating responses. Model parameters are controlled and updated by providers. You can also update model parameters to create a new model through the process of *model customization*.
- **Inference parameters** – Values that can be adjusted during **model inference** to influence a response. Inference parameters can affect how varied responses are and can also limit the length of a response or the occurrence of specified sequences. For more information and definitions of specific inference parameters, see Inference parameters.
- **Playground** – A user-friendly graphical interface in the AWS Management Console in which you can experiment with running model inference to familiarize yourself with Amazon Bedrock. Use the playground to test out the effects of different models, configurations, and inference parameters on the responses generated for different prompts that you enter. For more information, see Playgrounds.
- **Embedding** – The process of condensing information by transforming input into a vector of numerical values, known as the **embeddings**, in order to compare the similarity between different objects by using a shared numerical representation. For example, sentences can be compared to determine the similarity in meaning, images can be compared to determine visual

similarity, or text and image can be compared to see if they're relevant to each other. You can also combine text and image inputs into an averaged embeddings vector if it's relevant to your use case. For more information, see Run model inference and Knowledge bases for Amazon Bedrock.

Advanced features

The following list introduces you to more advanced concepts that you can explore through using Amazon Bedrock.

- **Orchestration** – The process of coordinating between foundation models and enterprise data and applications in order to carry out a task. For more information, see Agents for Amazon Bedrock.
- **Agent** – An application that carry out orchestrations through cyclically interpreting inputs and producing outputs by using a foundation model. An agent can be used to carry out customer requests. For more information, see Agents for Amazon Bedrock.
- **Retrieval augmented generation (RAG)** – The process of querying and retrieving information from a data source in order to augment a generated response to a prompt. For more information, see Knowledge bases for Amazon Bedrock.
- **Model customization** – The process of using training data to adjust the model parameter values in a base model in order to create a **custom model**. Examples of model customization include **Fine-tuning**, which uses labeled data (inputs and corresponding outputs), and **Continued Pre-training**, which uses unlabeled data (inputs only) to adjust model parameters. For more information about model customization techniques available in Amazon Bedrock, see Custom models.
- **Hyperparameters** – Values that can be adjusted for **model customization** to control the training process and, consequently, the output custom model. For more information and definitions of specific hyperparameters, see Custom model hyperparameters.
- **Model evaluation** – The process of evaluating and comparing model outputs in order to determine the model that is best suited for a use case. For more information, see Model evaluation.
- **Provisioned Throughput** – A level of throughput that you purchase for a base or custom model in order to increase the amount and/or rate of tokens processed during model inference. When you purchase Provisioned Throughput for a model, a **provisioned model** is created that can

be used to carry out model inference. For more information, see Provisioned Throughput for Amazon Bedrock.

Set up Amazon Bedrock

Before you use Amazon Bedrock for the first time, complete the following tasks. Once you have set up your account and requested model access in the console, you can set up the API.

> **Important**
>
> Before you can use any of the foundation models, you must request access to that model. If you try to use the model (with the API or within the console) before you have requested access to it, you will receive an error message. For more information, see Model access.

Setup tasks

- Sign up for an AWS account
- Create an administrative user
- Grant programmatic access
- Console access
- Model access
- Set up the Amazon Bedrock API
- Using this service with an AWS SDK

Sign up for an AWS account

If you do not have an AWS account, complete the following steps to create one.

To sign up for an AWS account

1. Open https://portal.aws.amazon.com/billing/signup.
2. Follow the online instructions.

   Part of the sign-up procedure involves receiving a phone call and entering a verification code on the phone keypad.

   When you sign up for an AWS account, an *AWS account root user* is created. The root user has access to all AWS services and resources in the account. As a security best practice, assign administrative access to an administrative user, and use only the root user to perform tasks that require root user access.

AWS sends you a confirmation email after the sign-up process is complete. At any time, you can view your current account activity and manage your account by going to https://aws.amazon.com/ and choosing **My Account**.

Create an administrative user

After you sign up for an AWS account, secure your AWS account root user, enable AWS IAM Identity Center, and create an administrative user so that you don't use the root user for everyday tasks.

Secure your AWS account root user

1. Sign in to the AWS Management Console as the account owner by choosing **Root user** and entering your AWS account email address. On the next page, enter your password.

   For help signing in by using root user, see Signing in as the root user in the *AWS Sign-In User Guide*.

2. Turn on multi-factor authentication (MFA) for your root user.

   For instructions, see Enable a virtual MFA device for your AWS account root user (console) in the *IAM User Guide*.

Create an administrative user

1. Enable IAM Identity Center.

   For instructions, see Enabling AWS IAM Identity Center in the *AWS IAM Identity Center User Guide*.

2. In IAM Identity Center, grant administrative access to an administrative user.

   For a tutorial about using the IAM Identity Center directory as your identity source, see Configure user access with the default IAM Identity Center directory in the *AWS IAM Identity Center User Guide*.

Sign in as the administrative user

- To sign in with your IAM Identity Center user, use the sign-in URL that was sent to your email address when you created the IAM Identity Center user.

  For help signing in using an IAM Identity Center user, see Signing in to the AWS access portal in the *AWS Sign-In User Guide*.

Grant programmatic access

Users need programmatic access if they want to interact with AWS outside of the AWS Management Console. The way to grant programmatic access depends on the type of user that's accessing AWS.

To grant users programmatic access, choose one of the following options.

Which user needs programmatic access?	To	By
Workforce identity (Users managed in IAM Identity Center)	Use temporary credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Configuring the AWS CLI to use AWS IAM Identity Center in the *AWS Command Line Interface User Guide*. • For AWS SDKs, tools, and AWS APIs, see IAM Identity Center authentication in the *AWS SDKs and Tools Reference Guide*.
IAM	Use temporary credentials to sign programmatic requests	Following the instructions in Using temporary credentia

Which user needs programmatic access?	To	By
	to the AWS CLI, AWS SDKs, or AWS APIs.	ls with AWS resources in the *IAM User Guide*.
IAM	(Not recommended) Use long-term credentials to sign programmatic requests to the AWS CLI, AWS SDKs, or AWS APIs.	Following the instructions for the interface that you want to use. • For the AWS CLI, see Authenticating using IAM user credentials in the *AWS Command Line Interface User Guide*. • For AWS SDKs and tools, see Authenticate using long-term credentials in the *AWS SDKs and Tools Reference Guide*. • For AWS APIs, see Managing access keys for IAM users in the *IAM User Guide*.

Console access

To access the Amazon Bedrock console and playground:

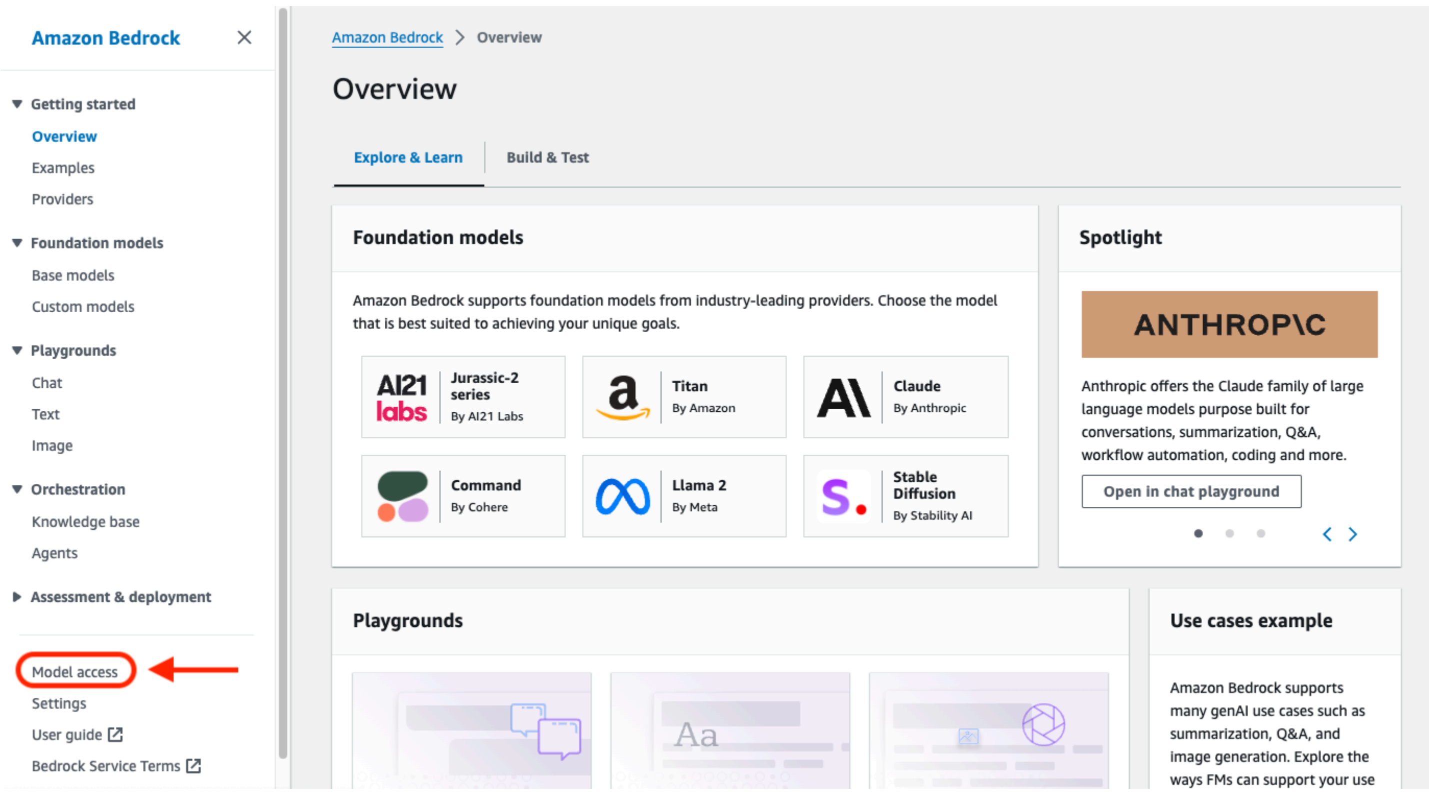
1. Sign in to your AWS account.
2. Navigate to: Amazon Bedrock console
3. Request model access by following the steps at Model access.

Model access

Access to Amazon Bedrock foundation models isn't granted by default. In order to gain access to a foundation model, an IAM user with sufficient permissions needs to request access to it through the console. Once access is provided to a model, it is available for all users in the account.

To manage model access, select **Model access** at the bottom of the left navigation pane in the Amazon Bedrock management console. The model access page lets you view a list of available models, the output modality of the model, whether you have been granted access to it, and the **End User License Agreement (EULA)**. You should review the EULA for terms and conditions of using a model before requesting access to it. For information about model pricing, refer to Amazon Bedrock Pricing.

> **Note**
>
> You can manage model access only through the console.



Topics

- Add model access

- Remove model access
- Control model access permissions

Add model access

Before you can use a foundation model in Amazon Bedrock, you must request access to it.

To request access to a model

1. On the **Model access** page, select **Manage model access**.
2. Select the checkboxes next to the models you want to add access to. To request access to all models belonging to a provider, select the check box next to the provider.

   Note

   You can't remove access from Titan models after requesting it.
   For Anthropic models, select **Submit use case details** and fill out the form. Once the details are submitted, you can then select the check box next to the Anthropic models that you want to request access to.

3. Select **Save changes** to request access. The changes may take several minutes to take place.

   Note

   Your use of Amazon Bedrock foundation models is subject to the seller's pricing terms, EULA, and the AWS service terms.

4. If your request is successful, the **Access status** changes to **Access granted**.

If you don't have permissions to request access to a model, an error banner appears. Contact your account administrator to ask them to request access to the model for you or to provide you permissions to request access to the model.

Remove model access

If you no longer need to use a foundation model, you can remove access to it.

Note

You can't remove access from Amazon Titan models, Mistral AI models, or from the Meta Llama 3 Instruct model.

1. On the **Model access** page, select **Manage model access**.
2. Select the checkboxes next to the models for which you want to remove access. To remove access for all models belonging to a provider, select the checkbox next to the provider.
3. Select **Save changes**.
4. You will be prompted to confirm you want to remove access to models. If you consent to the terms and select **Remove access**,

   Note

   The model may still be accessed through the API for some time after you complete this action while the changes propagate. To immediately remove access in the meantime, add an IAM policy to a role to deny access to the model.

Control model access permissions

To control a role's permissions to request access to Amazon Bedrock models, attach an IAM policy to the role using any of the following AWS Marketplace actions.

- `aws-marketplace:Subscribe`
- `aws-marketplace:Unsubscribe`
- `aws-marketplace:ViewSubscriptions`

For the `aws-marketplace:Subscribe` action only, you can use the `aws-marketplace:ProductId` condition key to limit subscription to specific models. The following table lists product IDs for Amazon Bedrock foundation models.

Model	Product ID
AI21 Labs Jurassic-2 Mid	1d288c71-65f9-489a-a3e2-9c7f4f6e6a85

Model	Product ID
AI21 Labs Jurassic-2 Ultra	cc0bdd50-279a-40d8-829c-4009b77a1fcc
Anthropic Claude	c468b48a-84df-43a4-8c46-8870630108a7
Anthropic Claude Instant	b0eb9475-3a2c-43d1-94d3-56756fd43737
Anthropic Claude 3 Sonnet	prod-6dw3qvchef7zy
Anthropic Claude 3 Haiku	prod-ozonys2hmmpeu
Anthropic Claude 3 Opus	prod-fm3feywmwerog
Cohere Command	a61c46fe-1747-41aa-9af0-2e0ae8a9ce05
Cohere Command Light	216b69fd-07d5-4c7b-866b-936456d68311
Cohere Embed (English)	b7568428-a1ab-46d8-bab3-37def50f6f6a
Cohere Embed (Multilingual)	38e55671-c3fe-4a44-9783-3584906e7cad
Meta Llama 2 13B	prod-ariujvyzvd2qy
Meta Llama 2 70B	prod-2c2yc2s3guhqy
Stable Diffusion XL 0.8	d0123e8d-50d6-4dba-8a26-3fed4899f388
Stable Diffusion XL 1.0	prod-2lvuzn4iy6n6o

The following is the format of the IAM policy you can attach to a role to control model access permissions. You can see an example at Allow access to third-party model subscriptions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow|Deny",
            "Action": [
                "aws-marketplace:Subscribe"
            ],
```

```
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "aws-marketplace:ProductId": [
                        model-product-id-1,
                        model-product-id-2,
                        ...
                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "aws-marketplace:Unsubscribe",
                "aws-marketplace:ViewSubscriptions"
            ],
            "Resource": "*"
        }
    ]
}
```

Set up the Amazon Bedrock API

This section describes how to set up your environment to make Amazon Bedrock API calls and provides examples of common use-cases. You can access the Amazon Bedrock API using the AWS Command Line Interface (AWS CLI), an AWS SDK, or a SageMaker Notebook.

Before you can access Amazon Bedrock APIs, you need to request access to the foundation models that you plan to use.

For details about the API operations and parameters, see the Amazon Bedrock API Reference.

The following resources provide additional information about the Amazon Bedrock API.

- *AWS Command Line Interface*
  - Amazon Bedrock CLI commands
  - Amazon Bedrock Runtime CLI commands
  - Agents for Amazon Bedrock CLI commands
  - Agents for Amazon Bedrock Runtime CLI commands

Add model access

> **Important**
>
> Before you can use any of the foundation models, you must request access to that model. If you try to use the model (with the API or within the console) before you have requested access to it, you will receive an error message. For more information, see Model access.

Amazon Bedrock endpoints

To connect programmatically to an AWS service, you use an endpoint. Refer to the Amazon Bedrock endpoints and quotas chapter in the AWS General Reference for information about the endpoints that you can use for Amazon Bedrock.

Amazon Bedrock provides the following service endpoints.

- `bedrock` – Contains control plane APIs for managing, training, and deploying models. For more information, see Amazon Bedrock Actions and Amazon Bedrock Data Types.
- `bedrock-runtime` – Contains data plane APIs for making inference requests for models hosted in Amazon Bedrock. For more information, see Amazon Bedrock Runtime Actions and Amazon Bedrock Runtime Data Types.
- `bedrock-agent` – Contains control plane APIs for creating and managing agents and knowledge bases. For more information, see Agents for Amazon Bedrock Actions and Agents for Amazon Bedrock Data Types.
- `bedrock-agent-runtime` – Contains data plane APIs for invoking agents and querying knowledge bases. For more information, see Agents for Amazon Bedrock Runtime Actions and Agents for Amazon Bedrock Runtime Data Types.

Setting up the AWS CLI

1. If you plan to use the CLI, install and configure the AWS CLI by following the steps at Install or update the latest version of the AWS Command Line Interface User Guide.
2. Configure your AWS credentials using the `aws configure` CLI command by following the steps at Configure the AWS CLI.

Refer to the following references for AWS CLI commands and operations:

- Amazon Bedrock CLI commands
- Amazon Bedrock Runtime CLI commands
- Agents for Amazon Bedrock CLI commands
- Agents for Amazon Bedrock Runtime CLI commands

Setting up an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language. SDKs automatically perform useful tasks for you, such as:

- Cryptographically sign your service requests
- Retry requests
- Handle error responses

Refer to the following table to find general information about and code examples for each SDK, as well as the Amazon Bedrock API references for each SDK. You can also find code examples at Code examples for Amazon Bedrock using AWS SDKs.

SDK documentation	Code examples	Amazon Bedrock prefix	Amazon Bedrock runtime prefix	Agents for Amazon Bedrock prefix	Agents for Amazon Bedrock runtime prefix
AWS SDK for C++	AWS SDK for C++ code examples	bedrock	bedrock-runtime	bedrock-agent	bedrock-agent-runtime
AWS SDK for Go	AWS SDK for Go code examples	bedrock	bedrockruntime	bedrockagent	bedrockagentruntime

SDK documentation	Code examples	Amazon Bedrock prefix	Amazon Bedrock runtime prefix	Agents for Amazon Bedrock prefix	Agents for Amazon Bedrock runtime prefix
AWS SDK for Java	AWS SDK for Java code examples	bedrock	bedrockruntime	bedrockagent	bedrockagentruntime
AWS SDK for JavaScript	AWS SDK for JavaScript code examples	bedrock	bedrock-runtime	bedrock-agent	bedrock-agent-runtime
AWS SDK for Kotlin	AWS SDK for Kotlin code examples	bedrock	bedrockruntime	bedrockagent	bedrockagentruntime
AWS SDK for .NET	AWS SDK for .NET code examples	Bedrock	BedrockRuntime	BedrockAgent	BedrockAgentRuntime
AWS SDK for PHP	AWS SDK for PHP code examples	Bedrock	BedrockRuntime	BedrockAgent	BedrockAgentRuntime
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples	bedrock	bedrock-runtime	bedrock-agent	bedrock-agent-runtime
AWS SDK for Ruby	AWS SDK for Ruby code examples	Bedrock	BedrockRuntime	BedrockAgent	BedrockAgentRuntime

SDK documentation	Code examples	Amazon Bedrock prefix	Amazon Bedrock runtime prefix	Agents for Amazon Bedrock prefix	Agents for Amazon Bedrock runtime prefix
AWS SDK for Rust	AWS SDK for Rust code examples	aws-sdk-bedrock	aws-sdk-bedrockruntime	aws-sdk-bedrockagent	aws-sdk-bedrockagentruntime
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples	BDK	BDR	BDA	BDZ
AWS SDK for Swift	AWS SDK for Swift code examples	AWSBedrock	AWSBedrockRuntime	AWSBedrockAgent	AWSBedrockAgentRuntime

Using SageMaker notebooks

You can use the SDK for Python (Boto3) to invoke Amazon Bedrock API operations from a SageMaker notebook.

Configure the SageMaker role

Add Amazon Bedrock permissions to the IAM role that will use this SageMaker notebook.

From the IAM console, perform these steps:

1. Choose the IAM role, then choose **Add Permissions** and select **Create Inline Policies** from the dropdown list.
2. Include the following permission.

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": "bedrock:*",
            "Resource": "*"
        }
    ]
}
```

Add the following permissions to the trust relationships.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        },
        {
            "Sid": "",
            "Effect": "Allow",
            "Principal": {
                "Service": "sagemaker.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

Test the Runtime setup

Add the following code to your notebook and run the code.

```
import boto3
import json
bedrock = boto3.client(service_name='bedrock-runtime')

body = json.dumps({
    "prompt": "\n\nHuman:explain black holes to 8th graders\n\nAssistant:",
    "max_tokens_to_sample": 300,
    "temperature": 0.1,
```

```
    "top_p": 0.9,
})

modelId = 'anthropic.claude-v2'
accept = 'application/json'
contentType = 'application/json'

response = bedrock.invoke_model(body=body, modelId=modelId, accept=accept,
 contentType=contentType)

response_body = json.loads(response.get('body').read())
# text
print(response_body.get('completion'))
```

Test the Amazon Bedrock setup

Add the following code to your notebook and run the code.

```
import boto3
bedrock = boto3.client(service_name='bedrock')

bedrock.get_foundation_model(modelIdentifier='anthropic.claude-v2')
```

Using this service with an AWS SDK

AWS software development kits (SDKs) are available for many popular programming languages. Each SDK provides an API, code examples, and documentation that make it easier for developers to build applications in their preferred language.

SDK documentation	Code examples
AWS SDK for C++	AWS SDK for C++ code examples
AWS SDK for Go	AWS SDK for Go code examples
AWS SDK for Java	AWS SDK for Java code examples
AWS SDK for JavaScript	AWS SDK for JavaScript code examples
AWS SDK for Kotlin	AWS SDK for Kotlin code examples

SDK documentation	Code examples
AWS SDK for .NET	AWS SDK for .NET code examples
AWS SDK for PHP	AWS SDK for PHP code examples
AWS SDK for Python (Boto3)	AWS SDK for Python (Boto3) code examples
AWS SDK for Ruby	AWS SDK for Ruby code examples
AWS SDK for Rust	AWS SDK for Rust code examples
AWS SDK for SAP ABAP	AWS SDK for SAP ABAP code examples
AWS SDK for Swift	AWS SDK for Swift code examples

Example availability

Can't find what you need? Request a code example by using the **Provide feedback** link at the bottom of this page.

Supported foundation models in Amazon Bedrock

Amazon Bedrock supports foundation models (FMs) from the following providers. Select a link in the **Provider** column to see documentation for that provider.

To use a foundation model with the Amazon Bedrock API, you'll need its model ID. For a list for model IDs, see Amazon Bedrock model IDs.

Provider	Model	Input modalities	Output modalities	Inference parameters	Hyperparameters
Amazon	Titan Text G1 - Express	Text	Text, Chat	Link	Link
	Titan Text G1 - Lite	Text	Text	Link	Link
	Titan Image Generator G1	Text, Image	Image	Link	Link
	Titan Embeddings G1 - Text	Text	Embeddings	Link	N/A
	Titan Multimodal Embeddings G1	Text, Image	Embeddings	Link	Link
Anthropic	Claude	Text	Text, Chat	Link	N/A
	Claude Instant	Text	Text, Chat	Link	N/A
	Claude 3 Sonnet	Text, Image	Text, Chat	Link	N/A
	Claude 3 Haiku	Text, Image	Text, Chat	Link	N/A

Provider	Model	Input modalities	Output modalities	Inference parameters	Hyperparameters
	Claude 3 Opus	Text, Image	Text, Chat	Link	N/A
AI21 Labs	Jurassic-2 Mid	Text	Text, Chat	Link	N/A
	Jurassic-2 Ultra	Text	Text, Chat	Link	N/A
Cohere	Command	Text	Text	Link	Link
	Command Light	Text	Text	Link	Link
	Embed English	Text	Embeddings	Link	N/A
	Embed Multilingual	Text	Embeddings	Link	N/A
Meta	Llama 2 Chat 13B	Text	Text, Chat	Link	N/A
	Llama 2 Chat 70B	Text	Text, Chat	Link	N/A
	Llama 2 13B (see note below)	Text	Text	Link	Link
	Llama 2 70B (see note below)	Text	Text	Link	Link
	Llama 3 8b Instruct	Text	Text, Chat	Link	N/A

Provider	Model	Input modalities	Output modalities	Inference parameters	Hyperparameters
	Llama 3 70b Instruct	Text	Text, Chat	Link	N/A
Mistral AI	Mistral 7B Instruct	Text	Text	Link	N/A
	Mixtral 8X7B Instruct	Text	Text	Link	N/A
	Mistral Large	Text	Text	Link	N/A
Stability AI	Stable Diffusion XL	Text, Image	Image	Link	N/A

> **Note**
>
> The Meta Llama 2 (non-chat) models can only be used after being customized and after purchasing Provisioned Throughput for them.

The following sections provide information about using foundation models and reference information for models.

Topics

- Using foundation models
- Get information about foundation models
- Model support by AWS Region
- Model support by feature
- Model lifecycle
- Amazon Bedrock model IDs
- Inference parameters for foundation models
- Custom model hyperparameters

Using foundation models

You must request access to a model before you can use it. After doing so, you can then use FMs in the following ways.

- Run inference by sending prompts to a model and generating responses. The playgrounds offer a user-friendly interface in the AWS Management Console for generating text, images, or chats. See the **Output modality** column to determine the models you can use in each playground.

> **Note**
>
> The console playgrounds don't support running inference on embeddings models. Use the API to run inference on embeddings models.

- Evaluate models to compare outputs and determine the best model for your use-case.
- Set up a knowledge base with the help of an embeddings model. Then use a text model to generate responses to queries.
- Create an agent and use a model to run inference on prompts to carry out orchestration.
- Customize a model by feeding training and validation data to adjust model parameters for your use-case. To use a customized model, you must purchase Provisioned Throughput for it.
- Purchase Provisioned Throughput for a model to increase throughput for it.

To use an FM in the API, you need to determine the appropriate **model ID** to use.

Use case	How to find the model ID
Use a base model	Look up the ID in the base model IDs chart
Purchase Provisioned Throughput for a base model	Look up the ID in the model IDs for Provisioned Throughput chart and use it as the `modelId` in the CreateProvisionedModelThroughput request.
Purchase Provisioned Throughput for a custom model	Use the name of the custom model or its ARN as the `modelId` in the CreateProvisionedModelThroughput request.

Use case	How to find the model ID
Use a provisioned model	After you create a Provisioned Throughput, it returns a `provisionedModelArn` . This ARN is the model ID.
Use a custom model	Purchase Provisioned Throughput for the custom model and use the returned `provisionedModelArn` as the model ID.

Get information about foundation models

In the Amazon Bedrock console, you can find overarching information about Amazon Bedrock foundation model providers and the models they provide in the **Providers** and **Base models** sections.

Use the API to retrieve information about Amazon Bedrock foundation model, including its ARN, model ID, modalities and features it supports, and whether it is deprecated or not, in a FoundationModelSummary object.

- To return information about all the foundation models that Amazon Bedrock provides, send a ListFoundationModels request.

> **Note**
>
> The response also returns model IDs that aren't in the base model ID or base model IDs for Provisioned Throughput charts. These model IDs are deprecated or for backwards compability.

- To return information about a specific foundation model, send a GetFoundationModel request, specifying the model ID.

Select a tab to see code examples in an interface or language.

AWS CLI

List the Amazon Bedrock foundation models.

```
aws bedrock list-foundation-models
```

Get information about Anthropic Claude v2.

```
aws bedrock get-foundation-model --model-identifier anthropic.claude-v2
```

Python

List the Amazon Bedrock foundation models.

```
import boto3
bedrock = boto3.client(service_name='bedrock')

bedrock.list_foundation_models()
```

Get information about Anthropic Claude v2.

```
import boto3
bedrock = boto3.client(service_name='bedrock')

bedrock.get_foundation_model(modelIdentifier='anthropic.claude-v2')
```

Model support by AWS Region

Note

All models, except Anthropic Claude 3 Opus, are supported in the US East (N. Virginia, `us-east-1`) and US West (Oregon, `us-west-2`) Regions. Anthropic Claude 3 Opus is only available in US West (Oregon, `us-west-2`).

The following table shows the FMs that are available in other Regions and whether they're supported in each Region.

Model	Asia Pacific (Singapore)	Asia Pacific (Sydney)	Asia Pacific (Tokyo)	Europe (Frankfurt)	Europe (Paris)	AWS GovCloud (US-West)
Amazon Titan Text G1 - Express	No	Yes	Yes	Yes	Yes	Yes
Amazon Titan Text G1 - Lite	No	Yes	No	No	Yes	No
Amazon Titan Embeddings G1 - Text	No	No	Yes	Yes	No	No
Amazon Titan Multimodal Embeddings G1	No	Yes	No	No	Yes	No
Anthropic Claude v2 (18K context window)	Yes	No	No	Yes	No	No
Anthropic Claude v2.1 (200K context window)	No	No	Yes	Yes	No	No

Model	Asia Pacific (Singapore)	Asia Pacific (Sydney)	Asia Pacific (Tokyo)	Europe (Frankfurt)	Europe (Paris)	AWS GovCloud (US-West)
Anthropic Claude Instant v1.x (18K context window)	Yes	No	Yes	No	No	No
Anthropic Claude Instant v1.x (100K context window)	No	No	No	Yes	No	No
Anthropic Claude 3 Haiku	No	Yes	No	No	Yes	No
Anthropic Claude 3 Sonnet	No	Yes	No	No	Yes	No
Cohere Embed English	Yes	Yes	Yes	Yes	Yes	No
Cohere Embed Multilingual	Yes	Yes	Yes	Yes	Yes	No

Model	Asia Pacific (Singapore)	Asia Pacific (Sydney)	Asia Pacific (Tokyo)	Europe (Frankfurt)	Europe (Paris)	AWS GovCloud (US-West)
Mistral AI Mistral 7B Instruct	No	Yes	No	No	Yes	No
Mistral AI Mixtral 8X7B Instruct	No	Yes	No	No	Yes	No
Mistral AI Mistral Large	No	Yes	No	No	Yes	No

Model support by feature

Note

You can run inference on all available FMs.

The following table details the support for features that are limited to certain FMs.

Model	Model evaluation	Knowledge base (embeddings)	Knowledge base (query)	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
Amazon Titan	Yes	N/A	No	No	Yes	Yes	Yes

Model	Model evaluation	Knowledge base (embeddings)	Knowledge base (query)	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
Text G1 - Express							
Amazon Titan Text G1 - Lite	Yes	N/A	No	No	Yes	Yes	Yes
Amazon Titan Embeddings G1 - Text	No	N/A	No	No	No	No	Yes
Amazon Titan Multimodal Embeddings G1	No	Yes	No	No	Yes	No	Yes
Amazon Titan Image Generator G1 (preview)	No	N/A	No	No	Yes	No	Yes
Anthropic Claude v1	Yes	N/A	No	No	No	No	Yes

Model	Model evaluation	Knowledge base (embeddings)	Knowledge base (query)	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
Anthropic Claude v2	Yes	N/A	Yes	Yes	No	No	Yes
Anthropic Claude v2.1	No	N/A	Yes	Yes	No	No	Yes
Anthropic Claude Instant	Yes	N/A	Yes	Yes	No	No	Yes
Anthropic Claude 3 Sonnet	No	N/A	Yes	No	No	No	Yes
Anthropic Claude 3 Haiku	No	N/A	Yes	No	No	No	Yes
Anthropic Claude 3 Opus	No	N/A	No	No	No	No	No
AI21 Labs Jurassic-2 Mid	Yes	No	No	No	No	No	No

Model	Model evaluation	Knowledge base (embeddings)	Knowledge base (query)	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
AI21 Labs Jurassic-2 Ultra	Yes	No	No	No	No	No	No
Cohere Command	Yes	N/A	No	No	Yes	No	Yes
Cohere Command Light	Yes	N/A	No	No	Yes	No	Yes
Cohere Embed English	No	Yes	No	No	No	No	Yes
Cohere Embed Multilingual	No	Yes	No	No	No	No	Yes
Meta Llama 2 Chat 13B	Yes	N/A	No	No	No	No	Yes
Meta Llama 2 Chat 70B	Yes	N/A	No	No	No	No	No
Meta Llama 2 13B	No	N/A	No	No	Yes	No	Yes (see note below)

Model	Model evaluation	Knowledge base (embeddings)	Knowledge base (query)	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
Meta Llama 2 70B	No	N/A	No	No	Yes	No	Yes (see note below)
Meta Llama 2 70B	No	N/A	No	No	Yes	No	Yes (see note below)
Meta Llama 3 8b Instruct	No	N/A	No	No	Yes	No	No
Meta Llama 3 70b Instruct	No	N/A	No	No	Yes	No	No
Mistral AI Mistral 7B Instruct	No	N/A	No	No	No	No	Yes
Mistral AI Mistral Large	No	No	N/A	No	No	No	No
Mistral AI Mixtral 8X7B Instruct	No	N/A	No	No	No	No	Yes

Model	Model evaluation	Knowledge base (embeddings)	Knowledge base (query)	Agents	Fine-tuning (custom models)	Continued pre-training (custom models)	Provisioned Throughput
Stable Diffusion XL 0.8	No	N/A	No	No	No	No	No
Stable Diffusion XL 1.x	No	N/A	No	No	No	No	Yes

Note

The Meta Llama 2 (non-chat) models can only be used after being customized and after purchasing Provisioned Throughput for them.

Model lifecycle

Amazon Bedrock is continuously working to bring the latest versions of foundation models that have better capabilities, accuracy, and safety. As we launch new model versions, you can test them with the Amazon Bedrock console or API, and migrate your applications to benefit from the latest model versions.

A model offered on Amazon Bedrock can be in one of these states: **Active**, **Legacy**, or **End-of-Life (EOL)**.

- **Active**: The model provider is actively working on this version, and it will continue to get updates such as bug fixes and minor improvements.
- **Legacy**: A version is marked Legacy when there is a more recent version which provides superior performance. Amazon Bedrock sets an EOL date for Legacy versions. The EOL date may vary depending on how you use the model (for example, whether you use on-demand throughput or Provisioned Throughput for a base model, or Provisioned Throughput for a customized model).

While you can continue to use a Legacy version, you should plan to transition to an Active version before the EOL date.

- **EOL**: This version is no longer available for use. Any requests made to this version will fail.

The console marks a model version's state as **Active** or **Legacy**. When you make a GetFoundationModel or ListFoundationModels call, you can find the state of the model in the `modelLifecycle` field in the response. After the EOL date, the model version can only be found on this documentation page.

On-Demand, Provisioned Throughput, and model customization

You specify the version of a model when you use it in **On-Demand** mode (for example, `anthropic.claude-v2`, `anthropic.claude-v2:1`, etc.).

When you configure **Provisioned Throughput**, you must specify a model version that will remain unchanged for the entire term. You can purchase a new Provisioned Throughput commitment (or renew an existing one) for a version if the commitment term ends before the version's EOL date.

If you customized a model, you can continue to use it until the EOL date of the base model version that you used for customization. You can also customize a legacy model version, but you should plan to migrate before it reaches its EOL date.

> **Note**
>
> Service quotas are shared among model minor versions.

Legacy versions

The following table shows the legacy versions of models available on Amazon Bedrock.

Model version	Legacy date	EOL date	Recommended model version replacement	Recommended model ID
Stable Diffusion XL 0.8	February 2, 2024	April 30, 2024	Stable Diffusion XL 1.x	stability.stable-diffusion-xl-v1

Model version	Legacy date	EOL date	Recommended model version replacement	Recommended model ID
Claude v1.3	November 28, 2023	February 28, 2024	Claude v2.1	anthropic .claude-v2:1
Titan Embeddings - Text v1.1	November 7, 2023	February 15, 2024	Titan Embeddings - Text v1.2	amazon.titan-embed-text-v1

Amazon Bedrock model IDs

Many Amazon Bedrock API operations require the use of a model ID. Refer to the following table to determine where to find the model ID that you need to use.

Use case	How to find the model ID
Use a base model	Look up the ID in the base model IDs chart
Purchase Provisioned Throughput for a base model	Look up the ID in the model IDs for Provisioned Throughput chart and use it as the `modelId` in the CreateProvisionedModelThroughput request.
Purchase Provisioned Throughput for a custom model	Use the name of the custom model or its ARN as the `modelId` in the CreateProvisionedModelThroughput request.
Use a provisioned model	After you create a Provisioned Throughput, it returns a `provisionedModelArn` . This ARN is the model ID.
Use a custom model	Purchase Provisioned Throughput for the custom model and use the returned `provisionedModelArn` as the model ID.

Topics

- [Amazon Bedrock base model IDs (on-demand throughput)]
- [Amazon Bedrock base model IDs for purchasing Provisioned Throughput]

Amazon Bedrock base model IDs (on-demand throughput)

The following is a list of model IDs for the currently available base models. You use a model ID through the API to identify the base model that you want to use with on-demand throughput, such as in a InvokeModel request, or that you want to customize, such as in a CreateModelCustomizationJob request.

> **Note**
>
> You should regularly check the Model lifecycle page for information about model deprecation and update model IDs as necessary. Once a model has reached end-of-life, the model ID no longer works.

Provider	Model name	Version	Model ID
Amazon	Titan Text G1 - Express	1.x	amazon.titan-text-express-v1
Amazon	Titan Text G1 - Lite	1.x	amazon.titan-text-lite-v1
Amazon	Titan Embeddings G1 - Text	1.x	amazon.titan-embed-text-v1
Amazon	Titan Multimodal Embeddings G1	1.x	amazon.titan-embed-image-v1
Amazon	Titan Image Generator G1	1.x	amazon.titan-image-generator-v1
Anthropic	Claude	2.0	anthropic.claude-v2
Anthropic	Claude	2.1	anthropic.claude-v2:1

Provider	Model name	Version	Model ID
Anthropic	Claude 3 Sonnet	1.0	anthropic.claude-3-sonnet-20240229-v1:0
Anthropic	Claude 3 Haiku	1.0	anthropic.claude-3-haiku-20240307-v1:0
Anthropic	Claude 3 Opus	1.0	anthropic.claude-3-opus-20240229-v1:0
Anthropic	Claude Instant	1.x	anthropic.claude-instant-v1
AI21 Labs	Jurassic-2 Mid	1.x	ai21.j2-mid-v1
AI21 Labs	Jurassic-2 Ultra	1.x	ai21.j2-ultra-v1
Cohere	Command	14.x	cohere.command-text-v14
Cohere	Command Light	15.x	cohere.command-light-text-v14
Cohere	Embed English	3.x	cohere.embed-english-v3
Cohere	Embed Multilingual	3.x	cohere.embed-multilingual-v3
Meta	Llama 2 Chat 13B	1.x	meta.llama2-13b-chat-v1
Meta	Llama 2 Chat 70B	1.x	meta.llama2-70b-chat-v1
Meta	Llama 3 8b Instruct	1.x	meta.llama3-8b-instruct-v1:0

Provider	Model name	Version	Model ID
Meta	Llama 3 70b Instruct	1.x	meta.llama3-70b-instruct-v1:0
Mistral AI	Mistral 7B Instruct	0.x	mistral.mistral-7b-instruct-v0:2
Mistral AI	Mixtral 8X7B Instruct	0.x	mistral.mixtral-8x7b-instruct-v0:1
Mistral AI	Mistral Large	1.x	mistral.mistral-large-2402-v1:0
Stability AI	Stable Diffusion XL	0.x	stability.stable-diffusion-xl-v0
Stability AI	Stable Diffusion XL	1.x	stability.stable-diffusion-xl-v1

Amazon Bedrock base model IDs for purchasing Provisioned Throughput

To purchase Provisioned Throughput through the API, use the corresponding model ID when provisioning the model with a CreateProvisionedModelThroughput request. Provisioned Throughput is available for the following models:

> **Note**
>
> Some models have multiple contextual versions whose availability differs by region. For more information, see Model support by AWS Region.

Model name	No-commitment purchase supported for base model	Model ID for Provisioned Throughput
Amazon Titan Text G1 - Express	Yes	amazon.titan-text-express-v1:0:8k
Amazon Titan Text G1 - Lite	Yes	amazon.titan-text-lite-v1:0:4k
Amazon Titan Embeddings G1 - Text	Yes	amazon.titan-embed-text-v1:2:8k
Amazon Titan Multimodal Embeddings G1	Yes	amazon.titan-embed-image-v1:0
Amazon Titan Image Generator G1	No	amazon.titan-image-generator-v1:0
Anthropic Claude v2 18K	Yes	anthropic.claude-v2:0:18k
Anthropic Claude v2 100K	Yes	anthropic.claude-v2:0:100k
Anthropic Claude v2.1 18K	Yes	anthropic.claude-v2:1:18k
Anthropic Claude v2.1 200K	Yes	anthropic.claude-v2:1:200k
Anthropic Claude 3 Sonnet 28K	Yes	anthropic.claude-3-sonnet-20240229-v1:0:28k
Anthropic Claude 3 Sonnet 200K	Yes	anthropic.claude-3-sonnet-20240229-v1:0:200k
Anthropic Claude 3 Haiku 48K	Yes	anthropic.claude-3-haiku-20240307-v1:0:48k
Anthropic Claude 3 Haiku 200K	Yes	anthropic.claude-3-haiku-20240307-v1:0:200k
Anthropic Claude Instant v1 100K	Yes	anthropic.claude-instant-v1:2:100k

Model name	No-commitment purchase supported for base model	Model ID for Provisioned Throughput
Cohere Command	Yes	cohere.command-text-v14:7:4k
Cohere Command Light	Yes	cohere.command-light-text-v14:7:4k
Cohere Embed English	Yes	cohere.embed-english-v3:0:512
Cohere Embed Multilingual	Yes	cohere.embed-multilingual-v3:0:512
Stable Diffusion XL 1.0	No	stability.stable-diffusion-xl-v1:0
Meta Llama 2 Chat 13B	No	meta.llama2-13b-chat-v1:0:4k
Meta Llama 2 13B	No	(see note below)
Meta Llama 2 70B	No	(see note below)

Note

The Meta Llama 2 (non-chat) models can only be used after being customized and after purchasing Provisioned Throughput for them.

The CreateProvisionedModelThroughput response returns a `provisionedModelArn`. You can use this ARN or the name of the provisioned model in supported Amazon Bedrock operations. For more information about Provisioned Throughput, see Provisioned Throughput for Amazon Bedrock.

Inference parameters for foundation models

This section documents the inference parameters that you can use with the base models that Amazon Bedrock provides.

Optionally, set inference parameters to influence the response generated by the model. You set inference parameters in a playground in the console, or in the `body` field of the InvokeModel or InvokeModelWithResponseStream API.

When you call a model, you also include a prompt for the model. For information about writing prompts, see Prompt engineering guidelines.

The following sections define the inference parameters available for each base model. For a custom model, use the same inference parameters as the base model from which it was customized.

Topics

Amazon Titan models

The following pages describe inference parameters for Amazon Titan models.

Topics

Amazon Titan Text models

The Amazon Titan Text models support the following inference parameters.

For more information on Titan Text prompt engineering guidelines, see Titan Text Prompt Engineering Guidelines.

For more information on Titan models, see Amazon Titan Models.

Topics

- Request and response
- Code examples

Request and response

The request body is passed in the body field of an InvokeModel or InvokeModelWithResponseStream request.

Request

```
{
    "inputText": string,
    "textGenerationConfig": {
        "temperature": float,
        "topP": float,
        "maxTokenCount": int,
        "stopSequences": [string]
    }
}
```

The following parameters are required:

- **inputText** – The prompt to provide the model for generating a response. To generate responses in a conversational style, wrap the prompt by using the following format:

```
"inputText": "User: <prompt>\nBot:
```

The `textGenerationConfig` is optional. You can use it to configure the following inference parameters:

- **temperature** – Use a lower value to decrease randomness in responses.

Default	Minimum	Maximum
0.0	0.0	1.0

- **topP** – Use a lower value to ignore less probable options and decrease the diversity of responses.

Default	Minimum	Maximum
1	1.00E-45	1

- **maxTokenCount** – Specify the maximum number of tokens to generate in the response.

Default	Minimum	Maximum
512	0	8,000

- **stopSequences** – Specify a character sequence to indicate where the model should stop. Currently, you can only specify one of the following options:
  - `|`
  - `User:`

InvokeModel Response

The response body contains the following possible fields:

```
{
    'inputTextTokenCount': int,
    'results': [{
        'tokenCount': int,
        'outputText': '\n<response>\n',
        'completionReason': string
    }]
}
```

More information about each field is provided below.

- `inputTextTokenCount` – The number of tokens in the prompt.
- `tokenCount` – The number of tokens in the response.
- `outputText` – The text in the response.
- `completionReason` – The reason the response finished being generated. The following reasons are possible.
  - `FINISHED` – The response was fully generated.
  - `LENGTH` – The response was truncated because of the response length you set.

InvokeModelWithResponseStream Response

Each chunk of text in the body of the response stream is in the following format. You must decode the `bytes` field (see Use the API to invoke a model with a single prompt for an example).

```
{
    'chunk': {
        'bytes': b'{
            "index": int,
            "inputTextTokenCount": int,
            "totalOutputTextTokenCount": int,
            "outputText": "<response-chunk>",
            "completionReason": string
        }'
    }
}
```

- `index` – The index of the chunk in the streaming response.
- `inputTextTokenCount` – The number of tokens in the prompt.
- `totalOutputTextTokenCount` – The number of tokens in the response.
- `outputText` – The text in the response.
- `completionReason` – The reason the response finished being generated. The following reasons are possible.
  - `FINISHED` – The response was fully generated.
  - `LENGTH` – The response was truncated because of the response length you set.

Code examples

The following example shows how to run inference with the Amazon Titan Text G1 - Express model with the Python SDK.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to create a list of action items from a meeting transcript
with the Amazon &titan-text-express; model (on demand).
"""
import json
import logging
import boto3

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by Amazon &titan-text-express; model"

    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_text(model_id, body):
    """
    Generate text using Amazon &titan-text-express; model on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (json): The response from the model.
    """

    logger.info(
        "Generating text with Amazon &titan-text-express; model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')
```

```
    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())

    finish_reason = response_body.get("error")

    if finish_reason is not None:
        raise ImageError(f"Text generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated text with Amazon &titan-text-express; model %s",
 model_id)

    return response_body


def main():
    """
    Entrypoint for Amazon &titan-text-express; example.
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        model_id = 'amazon.titan-text-express-v1'

        prompt = """Meeting transcript: Miguel: Hi Brant, I want to discuss the
 workstream
            for our new product launch Brant: Sure Miguel, is there anything in
 particular you want
            to discuss? Miguel: Yes, I want to talk about how users enter into the
 product.
            Brant: Ok, in that case let me add in Namita. Namita: Hey everyone
            Brant: Hi Namita, Miguel wants to discuss how users enter into the product.
            Miguel: its too complicated and we should remove friction.
            for example, why do I need to fill out additional forms?
            I also find it difficult to find where to access the product
            when I first land on the landing page. Brant: I would also add that
            I think there are too many steps. Namita: Ok, I can work on the
            landing page to make the product more discoverable but brant
```

```
            can you work on the additonal forms? Brant: Yes but I would need
            to work with James from another team as he needs to unblock the sign up
 workflow.
            Miguel can you document any other concerns so that I can discuss with James
 only once?
            Miguel: Sure.
            From the meeting transcript above, Create a list of action items for each
 person. """

        body = json.dumps({
            "inputText": prompt,
            "textGenerationConfig": {
                "maxTokenCount": 4096,
                "stopSequences": [],
                "temperature": 0,
                "topP": 1
            }
        })

        response_body = generate_text(model_id, body)
        print(f"Input token count: {response_body['inputTextTokenCount']}")

        for result in response_body['results']:
            print(f"Token count: {result['tokenCount']}")
            print(f"Output text: {result['outputText']}")
            print(f"Completion reason: {result['completionReason']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating text with the Amazon &titan-text-express; model
 {model_id}.")


if __name__ == "__main__":
```

```
    main()
```

Amazon Titan Image Generator G1

The Amazon Titan Image Generator G1 model supports the following inference parameters and model responses when carrying out model inference.

Topics

- Request and response format
- Code examples

Request and response format

When you make an InvokeModel call using the Amazon Titan Image Generator G1, replace the body field of the request with the format that matches your use-case. All tasks share an `imageGenerationConfig` object, but each task has a parameters object specific to that task. The following use-cases are supported.

taskType	Task parameters field	Type of task	Definition
`TEXT_IMAGE`	`textToImageParams`	Generation	Generate an image using a text prompt.
`INPAINTING`	`inPaintingParams`	Editing	Modify an image by changing the inside of a *mask* to match the surrounding background.
`OUTPAINTING`	`outPaintingParams`	Editing	Modify an image by seamlessly extending the region defined by the *mask*.
`IMAGE_VARIATION`	`imageVariationParams`	Editing	Modify an image by producing variations of the original image.

Editing tasks require an `image` field in the input. This field consists of a string that defines the pixels in the image. Each pixel is defined by 3 RGB channels, each of which ranges from 0 to 255 (for example, (255 255 0) would represent the color yellow). These channels are encoded in base64.

The image you use must be in JPEG or PNG format.

If you carry out inpainting or outpainting, you also define a *mask*, a region or regions that define parts of the image to be modified. You can define the mask in one of two ways.

- `maskPrompt` – Write a text prompt to describe the part of the image to be masked.
- `maskImage` – Input a base64-encoded string that defines the masked regions by marking each pixel in the input image as (0 0 0) or (255 255 255).
  - A pixel defined as (0 0 0) is a pixel inside the mask.
  - A pixel defined as (255 255 255) is a pixel outside the mask.

  You can use a photo editing tool to draw masks. You can then convert the output JPEG or PNG image to base64-encoding to input into this field. Otherwise, use the `maskPrompt` field instead to allow the model to infer the mask.

Select a tab to view API request bodies for different image generation use-cases and explanations of the fields.

Text-to-image generation (Request)

A text prompt to generate the image must be less than or equal to 512 characters. negativeText (Optional) – A text prompt to define what not to include in the image-- less than or equal to 512 characters.

```
{
    "taskType": "TEXT_IMAGE",
    "textToImageParams": {
        "text": "string",
        "negativeText": "string"
    },
    "imageGenerationConfig": {
        "numberOfImages": int,
        "height": int,
        "width": int,
```

```
        "cfgScale": float,
        "seed": int
    }
}
```

The `textToImageParams` fields are described below.

- **text** (Required) – A text prompt to generate the image.
- **negativeText** (Optional) – A text prompt to define what not to include in the image.

> **Note**
>
> Don't use negative words in the `negativeText` prompt. For example, if you don't want to include mirrors in an image, enter **`mirrors`** in the `negativeText` prompt. Don't enter **`no mirrors`**.

Inpainting (Request)

text (Optional) – A text prompt to define what to change inside the mask. If you don't include this field, the model tries to replace the entire mask area with the background. Must be less than or equal to 512 characters. negativeText (Optional) – A text prompt to define what not to include in the image. Must be less than or equal to 512 characters.

```
{
    "taskType": "INPAINTING",
    "inPaintingParams": {
        "image": "base64-encoded string",
        "text": "string",
        "negativeText": "string",
        "maskPrompt": "string",
        "maskImage": "base64-encoded string",
    },
    "imageGenerationConfig": {
        "numberOfImages": int,
        "height": int,
        "width": int,
        "cfgScale": float
    }
}
```

The `inPaintingParams` fields are described below. The *mask* defines the part of the image that you want to modify.

- **image** (Required) – The JPEG or PNG image to modify, formatted as a string that specifies a sequence of pixels, each defined in RGB values and encoded in base64. For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.
- You must define one of the following fields (but not both) in order to define.
  - **maskPrompt** – A text prompt that defines the mask.
  - **maskImage** – A string that defines the mask by specifying a sequence of pixels that is the same size as the `image`. Each pixel is turned into an RGB value of (0 0 0) (a pixel inside the mask) or (255 255 255) (a pixel outside the mask). For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.
- **text** (Optional) – A text prompt to define what to change inside the mask. If you don't include this field, the model tries to replace the entire mask area with the background.
- **negativeText** (Optional) – A text prompt to define what not to include in the image.

> **Note**
>
> Don't use negative words in the `negativeText` prompt. For example, if you don't want to include mirrors in an image, enter **`mirrors`** in the `negativeText` prompt. Don't enter **`no mirrors`**.

Outpainting (Request)

text (Required) – A text prompt to define what to change outside the mask. Must be less than or equal to 512 characters. negativeText (Optional) – A text prompt to define what not to include in the image. Must be less than or equal to 512 characters.

```
{
    "taskType": "OUTPAINTING",
    "outPaintingParams": {
        "text": "string",
        "negativeText": "string",
        "image": "base64-encoded string",
        "maskPrompt": "string",
```

```
        "maskImage": "base64-encoded string",
        "outPaintingMode": "DEFAULT | PRECISE"
    },
    "imageGenerationConfig": {
        "numberOfImages": int,
        "height": int,
        "width": int,
        "cfgScale": float
    }
}
```

The `outPaintingParams` fields are defined below. The *mask* defines the region in the image whose that you don't want to modify. The generation seamlessly extends the region you define.

- **image** (Required) – The JPEG or PNG image to modify, formatted as a string that specifies a sequence of pixels, each defined in RGB values and encoded in base64. For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.
- You must define one of the following fields (but not both) in order to define.
  - **maskPrompt** – A text prompt that defines the mask.
  - **maskImage** – A string that defines the mask by specifying a sequence of pixels that is the same size as the `image`. Each pixel is turned into an RGB value of (0 0 0) (a pixel inside the mask) or (255 255 255) (a pixel outside the mask). For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.
- **text** (Required) – A text prompt to define what to change outside the mask.
- **negativeText** (Optional) – A text prompt to define what not to include in the image.

> **Note**
>
> Don't use negative words in the `negativeText` prompt. For example, if you don't want to include mirrors in an image, enter **`mirrors`** in the `negativeText` prompt. Don't enter **`no mirrors`**.

- **outPaintingMode** – Specifies whether to allow modification of the pixels inside the mask or not. The following values are possible.
  - DEFAULT – Use this option to allow modification of the image inside the mask in order to keep it consistent with the reconstructed background.

- PRECISE – Use this option to prevent modification of the image inside the mask.

Image variation (Request)

- text (Optional) – A text prompt that can define what to preserve and what to change in the image. Must be less than or equal to 512 characters.
- negativeText (Optional) – A text prompt to define what not to include in the image. Must be less than or equal to 512 characters.
- text (Optional) – A text prompt that can define what to preserve and what to change in the image. Must be less than or equal to 512 characters.
- similarityStrength (Optional) – Specifies how similar the generated image should be to the input image(s) Use a lower value to introduce more randomness in the generation. Accepted range is between 0.2 and 1.0 (both inclusive), while a default of 0.7 is used if this parameter is missing in the request.

```
{
     "taskType": "IMAGE_VARIATION",
     "imageVariationParams": {
         "text": "string",
         "negativeText": "string",
         "images": ["base64-encoded string"],
         "similarityStrength": 0.7,  # Range: 0.2 to 1.0
     },
     "imageGenerationConfig": {
         "numberOfImages": int,
         "height": int,
         "width": int,
         "cfgScale": float
     }
}
```

The `imageVariationParams` fields are defined below.

- **images** (Required) – A list of images for which to generate variations. You can include 1 to 5 images. An image is defined as a base64-encoded image string. For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.

- **text** (Optional) – A text prompt that can define what to preserve and what to change in the image.
- **similarityStrength** (Optional) – Specifies how similar the generated image should be to the input images(s). Range in 0.2 to 1.0 with lower values used to introduce more randomness.
- **negativeText** (Optional) – A text prompt to define what not to include in the image.

> **Note**
>
> Don't use negative words in the `negativeText` prompt. For example, if you don't want to include mirrors in an image, enter **`mirrors`** in the `negativeText` prompt. Don't enter **`no mirrors`**.

Response body

```
{
  "images": [
    "base64-encoded string",
    ...
  ],
  "error": "string"
}
```

The response body is a streaming object that contains one of the following fields.

- `images` – If the request is successful, it returns this field, a list of base64-encoded strings, each defining a generated image. Each image is formatted as a string that specifies a sequence of pixels, each defined in RGB values and encoded in base64. For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.
- `error` – If the request violates the content moderation policy in one of the following situations, a message is returned in this field.
  - If the input text, image, or mask image is flagged by the content moderation policy.
  - If at least one output image is flagged by the content moderation policy

The shared and optional `imageGenerationConfig` contains the following fields. If you don't include this object, the default configurations are used.

- **numberOfImages** (Optional) – The number of images to generate.

Minimum	Maximum	Default
1	5	1

- **cfgScale** (Optional) – Specifies how strongly the generated image should adhere to the prompt. Use a lower value to introduce more randomness in the generation.

Minimum	Maximum	Default
1.1	10.0	8.0

- The following parameters define the size that you want the output image to be. For more details about pricing by image size, see Amazon Bedrock pricing.
  - **height** (Optional) – The height of the image in pixels. The default value is 1408.
  - **width** (Optional) – The width of the image in pixels. The default value is 1408.

  The following sizes are permissible.

Width	Height	Aspect ratio	Price equivalent to
1024	1024	1:1	1024 x 1024
768	768	1:1	512 x 512
512	512	1:1	512 x 512
768	1152	2:3	1024 x 1024
384	576	2:3	512 x 512
1152	768	3:2	1024 x 1024
576	384	3:2	512 x 512
768	1280	3:5	1024 x 1024
384	640	3:5	512 x 512

Width	Height	Aspect ratio	Price equivalent to
1280	768	5:3	1024 x 1024
640	384	5:3	512 x 512
896	1152	7:9	1024 x 1024
448	576	7:9	512 x 512
1152	896	9:7	1024 x 1024
576	448	9:7	512 x 512
768	1408	6:11	1024 x 1024
384	704	6:11	512 x 512
1408	768	11:6	1024 x 1024
704	384	11:6	512 x 512
640	1408	5:11	1024 x 1024
320	704	5:11	512 x 512
1408	640	11:5	1024 x 1024
704	320	11:5	512 x 512
1152	640	9:5	1024 x 1024
1173	640	16:9	1024 x 1024

- **seed** (Optional) – Use to control and reproduce results. Determines the initial noise setting. Use the same seed and the same settings as a previous run to allow inference to create a similar image.

> **Note**
>
> You can only set a `seed` for a TEXT_IMAGE generation task.

Minimum	Maximum	Default
0	2,147,483,646	0

Code examples

The following examples show how to invoke the Amazon Titan Image Generator G1 model with on-demand throughput in the Python SDK. Select a tab to view an example for each use-case. Each example displays the image at the end.

Text-to-image generation

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate an image from a text prompt with the Amazon Titan Image
 Generator G1 model (on demand).
"""
import base64
import io
import json
import logging
import boto3
from PIL import Image

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by Amazon Titan Image Generator G1"

    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_image(model_id, body):
```

```
    """
    Generate an image using Amazon Titan Image Generator G1 model on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        image_bytes (bytes): The image generated by the model.
    """

    logger.info(
        "Generating image with Amazon Titan Image Generator G1 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())

    base64_image = response_body.get("images")[0]
    base64_bytes = base64_image.encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)

    finish_reason = response_body.get("error")

    if finish_reason is not None:
        raise ImageError(f"Image generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated image with Amazon Titan Image Generator G1 model
 %s", model_id)

    return image_bytes


def main():
    """
    Entrypoint for Amazon Titan Image Generator G1 example.
    """

    logging.basicConfig(level=logging.INFO,
```

```
                        format="%(levelname)s: %(message)s")

    model_id = 'amazon.titan-image-generator-v1'

    prompt = """A photograph of a cup of coffee from the side."""

    body = json.dumps({
        "taskType": "TEXT_IMAGE",
        "textToImageParams": {
            "text": prompt
        },
        "imageGenerationConfig": {
            "numberOfImages": 1,
            "height": 1024,
            "width": 1024,
            "cfgScale": 8.0,
            "seed": 0
        }
    })

    try:
        image_bytes = generate_image(model_id=model_id,
                                     body=body)
        image = Image.open(io.BytesIO(image_bytes))
        image.show()

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating image with Amazon Titan Image Generator G1 model
 {model_id}.")


if __name__ == "__main__":
    main()
```

Inpainting

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to use inpainting to generate an image from a source image with
the Amazon Titan Image Generator G1 model (on demand).
The example uses a mask prompt to specify the area to inpaint.
"""
import base64
import io
import json
import logging
import boto3
from PIL import Image

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by Amazon Titan Image Generator G1"

    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_image(model_id, body):
    """
    Generate an image using Amazon Titan Image Generator G1 model on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        image_bytes (bytes): The image generated by the model.
    """

    logger.info(
        "Generating image with Amazon Titan Image Generator G1 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')
```

```
    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())

    base64_image = response_body.get("images")[0]
    base64_bytes = base64_image.encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)

    finish_reason = response_body.get("error")

    if finish_reason is not None:
        raise ImageError(f"Image generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated image with Amazon Titan Image Generator G1 model
 %s", model_id)

    return image_bytes


def main():
    """
    Entrypoint for Amazon Titan Image Generator G1 example.
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        model_id = 'amazon.titan-image-generator-v1'

        # Read image from file and encode it as base64 string.
        with open("/path/to/image", "rb") as image_file:
            input_image = base64.b64encode(image_file.read()).decode('utf8')

        body = json.dumps({
            "taskType": "INPAINTING",
            "inPaintingParams": {
                "text": "Modernize the windows of the house",
                "negativeText": "bad quality, low res",
```

```
                "image": input_image,
                "maskPrompt": "windows"
            },
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "height": 512,
                "width": 512,
                "cfgScale": 8.0
            }
        })

        image_bytes = generate_image(model_id=model_id,
                                     body=body)
        image = Image.open(io.BytesIO(image_bytes))
        image.show()

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating image with Amazon Titan Image Generator G1 model
 {model_id}.")


if __name__ == "__main__":
    main()
```

Outpainting

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to use outpainting to generate an image from a source image with
the Amazon Titan Image Generator G1 model (on demand).
The example uses a mask image to outpaint the original image.
"""
```

```
import base64
import io
import json
import logging
import boto3
from PIL import Image

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by Amazon Titan Image Generator G1"

    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_image(model_id, body):
    """
    Generate an image using Amazon Titan Image Generator G1 model on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        image_bytes (bytes): The image generated by the model.
    """

    logger.info(
        "Generating image with Amazon Titan Image Generator G1 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())
```

```
    base64_image = response_body.get("images")[0]
    base64_bytes = base64_image.encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)

    finish_reason = response_body.get("error")

    if finish_reason is not None:
        raise ImageError(f"Image generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated image with Amazon Titan Image Generator G1 model
 %s", model_id)

    return image_bytes


def main():
    """
    Entrypoint for Amazon Titan Image Generator G1 example.
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        model_id = 'amazon.titan-image-generator-v1'

        # Read image and mask image from file and encode as base64 strings.
        with open("/path/to/image", "rb") as image_file:
            input_image = base64.b64encode(image_file.read()).decode('utf8')
        with open("/path/to/mask_image", "rb") as mask_image_file:
            input_mask_image = base64.b64encode(
                mask_image_file.read()).decode('utf8')

        body = json.dumps({
            "taskType": "OUTPAINTING",
            "outPaintingParams": {
                "text": "Draw a chocolate chip cookie",
                "negativeText": "bad quality, low res",
                "image": input_image,
                "maskImage": input_mask_image,
                "outPaintingMode": "DEFAULT"
            },
            "imageGenerationConfig": {
                "numberOfImages": 1,
```

```
                "height": 512,
                "width": 512,
                "cfgScale": 8.0
            }
        }
        )

        image_bytes = generate_image(model_id=model_id,
                                     body=body)
        image = Image.open(io.BytesIO(image_bytes))
        image.show()

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating image with Amazon Titan Image Generator G1 model
 {model_id}.")


if __name__ == "__main__":
    main()
```

Image variation

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate an image variation from a source image with the
Amazon Titan Image Generator G1 model (on demand).
"""
import base64
import io
import json
import logging
import boto3
```

```
from PIL import Image

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by Amazon Titan Image Generator G1"

    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_image(model_id, body):
    """
    Generate an image using Amazon Titan Image Generator G1 model on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        image_bytes (bytes): The image generated by the model.
    """

    logger.info(
        "Generating image with Amazon Titan Image Generator G1 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())

    base64_image = response_body.get("images")[0]
    base64_bytes = base64_image.encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)

    finish_reason = response_body.get("error")
```

```
    if finish_reason is not None:
        raise ImageError(f"Image generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated image with Amazon Titan Image Generator G1 model
 %s", model_id)

    return image_bytes


def main():
    """
    Entrypoint for Amazon Titan Image Generator G1 example.
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        model_id = 'amazon.titan-image-generator-v1'

        # Read image from file and encode it as base64 string.
        with open("/path/to/image", "rb") as image_file:
            input_image = base64.b64encode(image_file.read()).decode('utf8')

        body = json.dumps({
            "taskType": "IMAGE_VARIATION",
            "imageVariationParams": {
                "text": "Modernize the house, photo-realistic, 8k, hdr",
                "negativeText": "bad quality, low resolution, cartoon",
                "images": [input_image],
  "similarityStrength": 0.7,  # Range: 0.2 to 1.0
            },
            "imageGenerationConfig": {
                "numberOfImages": 1,
                "height": 512,
                "width": 512,
                "cfgScale": 8.0
            }
        })

        image_bytes = generate_image(model_id=model_id,
                                     body=body)
        image = Image.open(io.BytesIO(image_bytes))
```

```
        image.show()

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating image with Amazon Titan Image Generator G1 model
 {model_id}.")


if __name__ == "__main__":
    main()
```

Amazon Titan Embeddings G1 - Text

Titan Embeddings G1 - Text doesn't support the use of inference parameters. The following sections detail the request and response formats and provides a code example.

Topics

- Request and response
- Example code

Request and response

The request body is passed in the `body` field of an InvokeModel request.

Request

The only available field is `inputText`, in which you can include text to convert into embeddings.

```
{
    "inputText": string
```

```
}
```

Response

The body of the response contains the following fields.

```
{
    "embedding": [float, float, ...],
    "inputTextTokenCount": int
}
```

The fields are described below.

- **embedding** – An array that represents the embeddings vector of the input you provided.
- **inputTextTokenCount** – The number of tokens in the input.

Example code

This example shows how to call the Amazon Titan Embeddings G1 - Text model to generate embeddings.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate embeddings with the Amazon Titan Embeddings G1 - Text model (on
 demand).
"""

import json
import logging
import boto3


from botocore.exceptions import ClientError


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_embeddings(model_id, body):
    """
```

```
    Generate a vector of embeddings for a text input using Amazon Titan Embeddings G1 -
 Text on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (JSON): The text that the model generated, token information, and the
        reason the model stopped generating text.
    """

    logger.info("Generating embeddings with Amazon Titan Embeddings G1 - Text model
 %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )

    response_body = json.loads(response.get('body').read())

    return response_body


def main():
    """
    Entrypoint for Amazon Titan Embeddings G1 - Text example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    model_id = "amazon.titan-embed-text-v1"
    input_text = "What are the different services that you offer?"


    # Create request body.
    body = json.dumps({
        "inputText": input_text,
    })
```

```
    try:

        response = generate_embeddings(model_id, body)

        print(f"Generated embeddings: {response['embedding']}")
        print(f"Input Token count:  {response['inputTextTokenCount']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))

    else:
        print(f"Finished generating embeddings with Amazon Titan Embeddings G1 - Text
 model {model_id}.")


if __name__ == "__main__":
    main()
```

Amazon Titan Multimodal Embeddings G1

This section provides request and respnose body formats and code examples for using Amazon Titan Multimodal Embeddings G1.

Topics

- Request and response
- Example code

Request and response

The request body is passed in the body field of an InvokeModel request.

Request

The request body for Amazon Titan Multimodal Embeddings G1 includes the following fields.

```
{
    "inputText": string,
```

```
    "inputImage": base64-encoded string,
    "embeddingConfig": {
        "outputEmbeddingLength": 256 | 384 | 1024
    }
}
```

At least one of the following fields is required. Include both to generate an embeddings vector that averages the resulting text embeddings and image embeddings vectors.

- **inputText** – Enter text to convert to embeddings.
- **inputImage** – Encode the image that you want to convert to embeddings in base64 and enter the string in this field. For examples of how to encode an image into base64 and decode a base64-encoded string and transform it into an image, see the code examples.

The following field is optional.

- **embeddingConfig** – Contains an `outputEmbeddingLength` field, in which you specify one of the following lengths for the output embeddings vector.
  - 256
  - 384
  - 1024 (default)

Response

The body of the response contains the following fields.

```
{
    "embedding": [float, float, ...],
    "inputTextTokenCount": int,
    "message": string
}
```

The fields are described below.

- **embedding** – An array that represents the embeddings vector of the input you provided.
- **inputTextTokenCount** – The number of tokens in the text input.
- **message** – Specifies any errors that occur during generation.

Example code

The following examples show how to invoke the Amazon Titan Multimodal Embeddings G1 model with on-demand throughput in the Python SDK. Select a tab to view an example for each use-case.

Text embeddings

This example shows how to call the Amazon Titan Multimodal Embeddings G1 model to generate text embeddings.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate embeddings from text with the Amazon Titan Multimodal
 Embeddings G1 model (on demand).
"""

import json
import logging
import boto3


from botocore.exceptions import ClientError

class EmbedError(Exception):
    "Custom exception for errors returned by Amazon Titan Multimodal Embeddings G1"

    def __init__(self, message):
        self.message = message

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_embeddings(model_id, body):
    """
    Generate a vector of embeddings for a text input using Amazon Titan Multimodal
 Embeddings G1 on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
```

```
        response (JSON): The embeddings that the model generated, token information,
 and the
        reason the model stopped generating embeddings.
    """

    logger.info("Generating embeddings with Amazon Titan Multimodal Embeddings G1
 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )

    response_body = json.loads(response.get('body').read())

    finish_reason = response_body.get("message")

    if finish_reason is not None:
        raise EmbedError(f"Embeddings generation error: {finish_reason}")

    return response_body


def main():
    """
    Entrypoint for Amazon Titan Multimodal Embeddings G1 example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    model_id = "amazon.titan-embed-image-v1"
    input_text = "What are the different services that you offer?"
    output_embedding_length = 256

    # Create request body.
    body = json.dumps({
        "inputText": input_text,
        "embeddingConfig": {
            "outputEmbeddingLength": output_embedding_length
```

```
        }
    })


    try:

        response = generate_embeddings(model_id, body)

        print(f"Generated text embeddings of length {output_embedding_length}:
 {response['embedding']}")
        print(f"Input text token count:  {response['inputTextTokenCount']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))

    except EmbedError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(f"Finished generating text embeddings with Amazon Titan Multimodal
 Embeddings G1 model {model_id}.")


if __name__ == "__main__":
    main()
```

Image embeddings

This example shows how to call the Amazon Titan Multimodal Embeddings G1 model to generate image embeddings.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate embeddings from an image with the Amazon Titan Multimodal
 Embeddings G1 model (on demand).
"""

import base64
```

```
import json
import logging
import boto3

from botocore.exceptions import ClientError

class EmbedError(Exception):
    "Custom exception for errors returned by Amazon Titan Multimodal Embeddings G1"

    def __init__(self, message):
        self.message = message

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_embeddings(model_id, body):
    """
    Generate a vector of embeddings for an image input using Amazon Titan Multimodal
 Embeddings G1 on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (JSON): The embeddings that the model generated, token information,
 and the
        reason the model stopped generating embeddings.
    """

    logger.info("Generating embeddings with Amazon Titan Multimodal Embeddings G1
 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )

    response_body = json.loads(response.get('body').read())

    finish_reason = response_body.get("message")
```

```
    if finish_reason is not None:
        raise EmbedError(f"Embeddings generation error: {finish_reason}")

    return response_body


def main():
    """
    Entrypoint for Amazon Titan Multimodal Embeddings G1 example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    # Read image from file and encode it as base64 string.
    with open("/path/to/image", "rb") as image_file:
        input_image = base64.b64encode(image_file.read()).decode('utf8')

    model_id = 'amazon.titan-embed-image-v1'
    output_embedding_length = 256

    # Create request body.
    body = json.dumps({
        "inputImage": input_image,
        "embeddingConfig": {
            "outputEmbeddingLength": output_embedding_length
        }
    })


    try:

        response = generate_embeddings(model_id, body)

        print(f"Generated image embeddings of length {output_embedding_length}:
 {response['embedding']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
```

```
    except EmbedError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(f"Finished generating image embeddings with Amazon Titan Multimodal
 Embeddings G1 model {model_id}.")


if __name__ == "__main__":
    main()
```

Text and image embeddings

This example shows how to call the Amazon Titan Multimodal Embeddings G1 model to generate embeddings from a combined text and image input. The resulting vector is the average of the generated text embeddings vector and the image embeddings vector.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate embeddings from an image and accompanying text with the Amazon
 Titan Multimodal Embeddings G1 model (on demand).
"""

import base64
import json
import logging
import boto3

from botocore.exceptions import ClientError

class EmbedError(Exception):
    "Custom exception for errors returned by Amazon Titan Multimodal Embeddings G1"

    def __init__(self, message):
        self.message = message

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_embeddings(model_id, body):
```

```
    """
    Generate a vector of embeddings for a combined text and image input using Amazon
 Titan Multimodal Embeddings G1 on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (JSON): The embeddings that the model generated, token information,
 and the
        reason the model stopped generating embeddings.
    """

    logger.info("Generating embeddings with Amazon Titan Multimodal Embeddings G1
 model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )

    response_body = json.loads(response.get('body').read())

    finish_reason = response_body.get("message")

    if finish_reason is not None:
        raise EmbedError(f"Embeddings generation error: {finish_reason}")

    return response_body


def main():
    """
    Entrypoint for Amazon Titan Multimodal Embeddings G1 example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    model_id = "amazon.titan-embed-image-v1"
    input_text = "A family eating dinner"
```

```
    # Read image from file and encode it as base64 string.
    with open("/path/to/image", "rb") as image_file:
        input_image = base64.b64encode(image_file.read()).decode('utf8')
    output_embedding_length = 256

    # Create request body.
    body = json.dumps({
        "inputText": input_text,
        "inputImage": input_image,
        "embeddingConfig": {
            "outputEmbeddingLength": output_embedding_length
        }
    })


    try:

        response = generate_embeddings(model_id, body)

        print(f"Generated embeddings of length {output_embedding_length}:
 {response['embedding']}")
        print(f"Input text token count:  {response['inputTextTokenCount']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))

    except EmbedError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(f"Finished generating embeddings with Amazon Titan Multimodal
 Embeddings G1 model {model_id}.")


if __name__ == "__main__":
    main()
```

Anthropic Claude models

This section provides inference parameters and code examples for using Anthropic Claude models.

You can use Amazon Bedrock to send Anthropic Claude Text Completions API or Anthropic Claude Messages API inference requests.

You use the messages API to create conversational applications, such as a virtual assistant or a coaching application. Use the text completion API for single-turn text generation applications. For example, generating text for a blog post or summarizing text that a user supplies.

You make inference requests to an Anthropic Claude model with InvokeModel or InvokeModelWithResponseStream (streaming). You need the model ID for the model that you want to use. To get the model ID for Anthropic Claude models, see Amazon Bedrock base model IDs (on-demand throughput) and Amazon Bedrock base model IDs for purchasing Provisioned Throughput.

> **Note**
>
> To use system prompts in inference calls, you must use Anthropic Claude version 2.1 or an Anthropic Claude 3 model, such as Anthropic Claude 3 Opus. For information about creating system prompts, see https://docs.anthropic.com/claude/docs/how-to-use-system-prompts in the Anthropic Claude documentation.
> To avoid timeouts with Anthropic Claude version 2.1, we recommend limiting the input token count in the `prompt` field to 180K. We expect to address this timeout issue soon.

In the inference call, fill the `body` field with a JSON object that conforms the type call you want to make, Anthropic Claude Text Completions API or Anthropic Claude Messages API.

For information about creating prompts for Anthropic Claude models, see Introduction to prompt design in the Anthropic Claude documentation.

Topics

- Anthropic Claude Text Completions API
- Anthropic Claude Messages API

Anthropic Claude Text Completions API

This section provides inference parameters and code examples for using Anthropic Claude models with the Text Completions API.

Topics

- Anthropic Claude Text Completions API overview
- Supported models
- Request and Response
- Code example

Anthropic Claude Text Completions API overview

Use the Text Completion API for single-turn text generation from a user supplied prompt. For example, you can use the Text Completion API to generate text for a blog post or to summarize text input from a user.

For information about creating prompts for Anthropic Claude models, see Introduction to prompt design. If you want to use your existing Text Completions prompts with the Anthropic Claude Messages API, see Migrating from Text Completions.

Supported models

You can use the Text Completions API with the following Anthropic Claude models.

- Anthropic Claude Instant v1.2
- Anthropic Claude v2
- Anthropic Claude v2.1

Request and Response

The request body is passed in the `body` field of a request to InvokeModel or InvokeModelWithResponseStream.

For more information, see https://docs.anthropic.com/claude/reference/complete_post in the Anthropic Claude documentation.

Request

Anthropic Claude has the following inference parameters for a Text Completion inference call.

```
{
    "prompt": "\n\nHuman:<prompt>\n\nAssistant:",
    "temperature": float,
    "top_p": float,
    "top_k": int,
    "max_tokens_to_sample": int,
    "stop_sequences": [string]
}
```

The following are required parameters.

- **prompt** – (Required) The prompt that you want Claude to complete. For proper response generation you need to format your prompt using alternating `\n\nHuman:` and `\n\nAssistant:` conversational turns. For example:

  ```
  "\n\nHuman: {userQuestion}\n\nAssistant:"
  ```

  For more information, see Prompt validation in the Anthropic Claude documentation.

- **max_tokens_to_sample** – (Required) The maximum number of tokens to generate before stopping. We recommend a limit of 4,000 tokens for optimal performance.

  Note that Anthropic Claude models might stop generating tokens before reaching the value of `max_tokens_to_sample`. Different Anthropic Claude models have different maximum values for this parameter. For more information, see Model comparison in the Anthropic Claude documentation.

Default	Minimum	Maximum
200	0	4096

The following are optional parameters.

- **stop_sequences** – (Optional) Sequences that will cause the model to stop generating.

Anthropic Claude models stop on `"\n\nHuman:"`, and may include additional built-in stop sequences in the future. Use the `stop_sequences` inference parameter to include additional strings that will signal the model to stop generating text.

- **temperature** – (Optional) The amount of randomness injected into the response.

  Defaults to 1. Ranges from 0 to 1. Use temp closer to 0 for analytical / multiple choice, and closer to 1 for creative and generative tasks.

Default	Minimum	Maximum
0.5	0	1

- **top_p** – (Optional) Use nucleus sampling.

  In nucleus sampling, Anthropic Claude computes the cumulative distribution over all the options for each subsequent token in decreasing probability order and cuts it off once it reaches a particular probability specified by `top_p`. You should alter either `temperature` or `top_p`, but not both.

Default	Minimum	Maximum
1	0	1

- **top_k** – (Optional) Only sample from the top K options for each subsequent token.

  Use `top_k` to remove long tail low probability responses.

Default	Minimum	Maximum
250	0	500

Response

The Anthropic Claude model returns the following fields for a Text Completion inference call.

```
{
    "completion": string,
    "stop_reason": string,
```

```
    "stop": string
}
```

- **completion** – The resulting completion up to and excluding the stop sequences.
- **stop_reason** – The reason why the model stopped generating the response.
  - **"stop_sequence"** – The model reached a stop sequence — either provided by you with the `stop_sequences` inference parameter, or a stop sequence built into the model.
  - **"max_tokens"** – The model exceeded `max_tokens_to_sample` or the model's maximum number of tokens.
- **stop** – If you specify the `stop_sequences` inference parameter, `stop` contains the stop sequence that signalled the model to stop generating text. For example, `holes` in the following response.

  ```
  {
      "completion": " Here is a simple explanation of black ",
      "stop_reason": "stop_sequence",
      "stop": "holes"
  }
  ```

  If you don't specify `stop_sequences`, the value for `stop` is empty.

Code example

These examples shows how to call the *Anthropic Claude V2* model with on demand throughput. To use Anthropic Claude version 2.1, change the value of `modelId` to `anthropic.claude-v2:1`.

```
import boto3
import json
brt = boto3.client(service_name='bedrock-runtime')

body = json.dumps({
    "prompt": "\n\nHuman: explain black holes to 8th graders\n\nAssistant:",
    "max_tokens_to_sample": 300,
    "temperature": 0.1,
    "top_p": 0.9,
})

modelId = 'anthropic.claude-v2'
accept = 'application/json'
```

```
contentType = 'application/json'

response = brt.invoke_model(body=body, modelId=modelId, accept=accept,
 contentType=contentType)

response_body = json.loads(response.get('body').read())

# text
print(response_body.get('completion'))
```

The following example shows how to generate streaming text with Python using the prompt *write an essay for living on mars in 1000 words* and the Anthropic Claude V2 model:

```
import boto3
import json

brt = boto3.client(service_name='bedrock-runtime')

body = json.dumps({
    'prompt': '\n\nHuman: write an essay for living on mars in 1000 words\n
\nAssistant:',
    'max_tokens_to_sample': 4000
})

response = brt.invoke_model_with_response_stream(
    modelId='anthropic.claude-v2',
    body=body
)

stream = response.get('body')
if stream:
    for event in stream:
        chunk = event.get('chunk')
        if chunk:
            print(json.loads(chunk.get('bytes').decode()))
```

Anthropic Claude Messages API

This section provides inference parameters and code examples for using the Anthropic Claude Messages API.

Topics

Anthropic Claude Messages API overview

You can use the Messages API to create chat bots or virtual assistant applications. The API manages the conversational exchanges between a user and an Anthropic Claude model (assistant).

Anthropic trains Claude models to operate on alternating user and assistant conversational turns. When creating a new message, you specify the prior conversational turns with the messages parameter. The model then generates the next Message in the conversation.

Each input message must be an object with a role and content. You can specify a single user-role message, or you can include multiple user and assistant messages. The first message must always use the user role.

If you are using the technique of prefilling the response from Claude (filling in the beginning of Claude's response by using a final assistant role Message), Claude will respond by picking up from where you left off. With this technique, Claude will still return a response with the assistant role.

If the final message uses the assistant role, the response content will continue immediately from the content in that message. You can use this to constrain part of the model's response.

Example with a single user message:

```
[{"role": "user", "content": "Hello, Claude"}]
```

Example with multiple conversational turns:

```
[
  {"role": "user", "content": "Hello there."},
  {"role": "assistant", "content": "Hi, I'm Claude. How can I help you?"},
  {"role": "user", "content": "Can you explain LLMs in plain English?"},
]
```

Example with a partially-filled response from Claude:

```
[
```

```
  {"role": "user", "content": "Please describe yourself using only JSON"},
  {"role": "assistant", "content": "Here is my JSON description:\n{"},
]
```

Each input message content may be either a single string or an array of content blocks, where each block has a specific type. Using a string is shorthand for an array of one content block of type "text". The following input messages are equivalent:

```
{"role": "user", "content": "Hello, Claude"}
```

```
{"role": "user", "content": [{"type": "text", "text": "Hello, Claude"}]}
```

For information about creating prompts for Anthropic Claude models, see Intro to prompting in the Anthropic Claude documentation. If you have existing Text Completion prompts that you want to migrate to the messages API, see Migrating from Text Completions.

System prompts

You can also include a system prompt in the request. A system prompt lets you provide context and instructions to Anthropic Claude, such as specifying a particular goal or role. Specify a system prompt in the `system` field, as shown in the following example.

```
"system": "You are Claude, an AI assistant created by Anthropic to be helpful,
                harmless, and honest. Your goal is to provide informative and
 substantive responses
                to queries while avoiding potential harms."
```

For more information, see System prompts in the Anthropic documentation.

Multimodal prompts

A multimodal prompt combines multiple modalities (images and text) in a single prompt. You specify the modalities in the `content` input field. The following example shows how you could ask Anthropic Claude to describe the content of a supplied image. For example code, see Multimodal code examples.

```
{
    "anthropic_version": "bedrock-2023-05-31",
    "max_tokens": 1024,
    "messages": [
        {
```

```
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": "image/jpeg",
                        "data": "iVBORw..."
                    }
                },
                {
                    "type": "text",
                    "text": "What's in these images?"
                }
            ]
        }
    ]
}
```

You can supply up to 20 images to the model. You can't put images in the assistant role.

Each image you include in a request counts towards your token usage. For more information, see Image costs in the Anthropic documentation.

Supported models

You can use the Messages API with the following Anthropic Claude models.

- Anthropic Claude Instant v1.2
- Anthropic Claude 2 v2
- Anthropic Claude 2 v2.1
- Anthropic Claude 3 Sonnet
- Anthropic Claude 3 Haiku
- Anthropic Claude 3 Opus

Request and Response

The request body is passed in the `body` field of a request to InvokeModel or InvokeModelWithResponseStream. The maximum size of the payload you can send in a request is 20MB.

For more information, see https://docs.anthropic.com/claude/reference/messages_post.

Request

Anthropic Claude has the following inference parameters for a messages inference call.

```
{
    "anthropic_version": "bedrock-2023-05-31",
    "max_tokens": int,
    "system": string,
    "messages": [
        {
            "role": string,
            "content": [
                { "type": "image", "source": { "type": "base64", "media_type":
 "image/jpeg", "data": "content image bytes" } },
                { "type": "text", "text": "content text" }
      ]
        }
    ],
    "temperature": float,
    "top_p": float,
    "top_k": int,
    "stop_sequences": [string]
}
```

The following are required parameters.

- **anthropic_version** – (Required) The anthropic version. The value must be `bedrock-2023-05-31`.
- **max_tokens** – (Required) The maximum number of tokens to generate before stopping.

  Note that Anthropic Claude models might stop generating tokens before reaching the value of `max_tokens`. Different Anthropic Claude models have different maximum values for this parameter. For more information, see Model comparison.
- **messages** – (Required) The input messages.
  - **role** – The role of the conversation turn. Valid values are `user` and `assistant`.
  - **content** – (required) The content of the conversation turn.
    - **type** – (required) The type of the content. Valid values are `image` and `text`.

If you specify `image`, you must also specify the image source in the following format

source – (required) The content of the conversation turn.

- **type** – (required) The encoding type for the image. You can specify `base64`.
- **media_type** – (required) The type of the image. You can specify the following image formats.
  - `image/jpeg`
  - `image/png`
  - `image/webp`
  - `image/gif`
- **data** – (required) The base64 encoded image bytes for the image. The maximum image size is 3.75MB. The maximum height and width of an image is 8000 pixels.

If you specify `text`, you must also specify the prompt in `text`.

The following are optional parameters.

- **system** – (Optional) The system prompt for the request.

  A system prompt is a way of providing context and instructions to Anthropic Claude, such as specifying a particular goal or role. For more information, see [How to use system prompts]() in the Anthropic documentation.

  > **Note**
  >
  > You can use system prompts with Anthropic Claude version 2.1 or higher.

- **stop_sequences** – (Optional) Custom text sequences that cause the model to stop generating. Anthropic Claude models normally stop when they have naturally completed their turn, in this case the value of the `stop_reason` response field is `end_turn`. If you want the model to stop generating when it encounters custom strings of text, you can use the `stop_sequences` parameter. If the model encounters one of the custom text strings, the value of the `stop_reason` response field is `stop_sequence` and the value of `stop_sequence` contains the matched stop sequence.

  The maximum number of entries is 8191.

- **temperature** – (Optional) The amount of randomness injected into the response.

Default	Minimum	Maximum
1	0	1

- **top_p** – (Optional) Use nucleus sampling.

  In nucleus sampling, Anthropic Claude computes the cumulative distribution over all the options for each subsequent token in decreasing probability order and cuts it off once it reaches a particular probability specified by `top_p`. You should alter either `temperature` or `top_p`, but not both.

Default	Minimum	Maximum
0.999	0	1

The following are optional parameters.

- **top_k** – (Optional) Only sample from the top K options for each subsequent token.

  Use `top_k` to remove long tail low probability responses.

Default	Minimum	Maximum
Disabled by default	0	500

Response

The Anthropic Claude model returns the following fields for a messages inference call.

```
{
    "id": string,
    "model": string,
    "type" : "message",
    "role" : "assistant",
    "content": [
        {
```

```
            "type": "text",
            "text": string
        }
    ],
    "stop_reason": string,
    "stop_sequence": string,
    "usage": {
        "input_tokens": integer,
        "output_tokens": integer
    }

}
```

- **id –** The unique identifier for the response. The format and length of the ID might change over time.
- **model –** The ID for the Anthropic Claude model that made the request.
- **stop_reason –** The reason why Anthropic Claude stopped generating the response.
  - **end_turn –** The model reached a natural stopping point
  - **max_tokens –** The generated text exceeded the value of the `max_tokens` input field or exceeded the maximum number of tokens that the model supports.' .
  - **stop_sequence –** The model generated one of the stop sequences that you specified in the `stop_sequences` input field.
- **type –** The type of response. The value is always `message`.
- **role –** The conversational role of the generated message. The value is always `assistant`.
- **content –** The content generated by the model. Returned as an array.
  - **type –** The type of the content. Currently the only supported value is `text`.
  - **text –** The text of the content.
- **usage –** Container for the number of tokens that you supplied in the request and the number tokens of that the model generated in the response.
  - **input_tokens –** The number of input tokens in the request.
  - **output_tokens –** The number tokens of that the model generated in the response.
  - **stop_sequence –** The model generated one of the stop sequences that you specified in the `stop_sequences` input field.

Code examples

The following code examples show how to use the messges API.

Topics

- Messages code example
- Multimodal code examples

Messages code example

This example shows how to send a single turn user message and a user turn with a prefilled assistant message to an Anthropic Claude 3 Sonnet model.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate a message with Anthropic Claude (on demand).
"""
import boto3
import json
import logging

from botocore.exceptions import ClientError


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)

def generate_message(bedrock_runtime, model_id, system_prompt, messages, max_tokens):

    body=json.dumps(
        {
            "anthropic_version": "bedrock-2023-05-31",
            "max_tokens": max_tokens,
            "system": system_prompt,
            "messages": messages
        }
    )


    response = bedrock_runtime.invoke_model(body=body, modelId=model_id)
    response_body = json.loads(response.get('body').read())
```

```
    return response_body


def main():
    """
    Entrypoint for Anthropic Claude message example.
    """

    try:

        bedrock_runtime = boto3.client(service_name='bedrock-runtime')

        model_id = 'anthropic.claude-3-sonnet-20240229-v1:0'
        system_prompt = "Please respond only with emoji."
        max_tokens = 1000

        # Prompt with user turn only.
        user_message =  {"role": "user", "content": "Hello World"}
        messages = [user_message]

        response = generate_message (bedrock_runtime, model_id, system_prompt,
 messages, max_tokens)
        print("User turn only.")
        print(json.dumps(response, indent=4))

        # Prompt with both user turn and prefilled assistant response.
        #Anthropic Claude continues by using the prefilled assistant text.
        assistant_message =  {"role": "assistant", "content": "<emoji>"}
        messages = [user_message, assistant_message]
        response = generate_message(bedrock_runtime, model_id,system_prompt, messages,
 max_tokens)
        print("User turn and prefilled assistant response.")
        print(json.dumps(response, indent=4))

    except ClientError as err:
        message=err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
            format(message))

if __name__ == "__main__":
    main()
```

Multimodal code examples

The following examples show how to pass an image and prompt text in a multimodal message to an Anthropic Claude 3 Sonnet model.

Topics

- Multimodal prompt with InvokeModel
- Streaming multimodal prompt with InvokeModelWithResponseStream

Multimodal prompt with InvokeModel

The following example shows how to send a multimodal prompt to Anthropic Claude 3 Sonnet with InvokeModel.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to run a multimodal prompt with Anthropic Claude (on demand) and InvokeModel.
"""

import json
import logging
import base64
import boto3

from botocore.exceptions import ClientError


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def run_multi_modal_prompt(bedrock_runtime, model_id, messages, max_tokens):
    """
    Invokes a model with a multimodal prompt.
    Args:
        bedrock_runtime: The Amazon Bedrock boto3 client.
        model_id (str): The model ID to use.
        messages (JSON) : The messages to send to the model.
        max_tokens (int) : The maximum  number of tokens to generate.
```

```
    Returns:
        None.
    """



    body = json.dumps(
        {
            "anthropic_version": "bedrock-2023-05-31",
            "max_tokens": max_tokens,
             "messages": messages
        }
    )

    response = bedrock_runtime.invoke_model(
        body=body, modelId=model_id)
    response_body = json.loads(response.get('body').read())

    return response_body


def main():
    """
    Entrypoint for Anthropic Claude multimodal prompt example.
    """

    try:

        bedrock_runtime = boto3.client(service_name='bedrock-runtime')

        model_id = 'anthropic.claude-3-sonnet-20240229-v1:0'
        max_tokens = 1000
        input_image = "/path/to/image"
        input_text = "What's in this image?"


        # Read reference image from file and encode as base64 strings.
        with open(input_image, "rb") as image_file:
            content_image = base64.b64encode(image_file.read()).decode('utf8')

        message = {"role": "user",
             "content": [
                {"type": "image", "source": {"type": "base64",
                    "media_type": "image/jpeg", "data": content_image}},
```

```
                {"type": "text", "text": input_text}
                ]}


        messages = [message]

        response = run_multi_modal_prompt(
            bedrock_runtime, model_id, messages, max_tokens)
        print(json.dumps(response, indent=4))

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))


if __name__ == "__main__":
    main()
```

Streaming multimodal prompt with InvokeModelWithResponseStream

The following example shows how to stream the response from a multimodal prompt sent to Anthropic Claude 3 Sonnet with InvokeModelWithResponseStream.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to stream the response from Anthropic Claude Sonnet (on demand) for a
multimodal request.
"""

import json
import base64
import logging
import boto3

from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


```

```
def stream_multi_modal_prompt(bedrock_runtime, model_id, input_text, image,
 max_tokens):
    """
    Streams the response from a multimodal prompt.
    Args:
        bedrock_runtime: The Amazon Bedrock boto3 client.
        model_id (str): The model ID to use.
        input_text (str) : The prompt text
        image (str) : The path to  an image that you want in the prompt.
        max_tokens (int) : The maximum  number of tokens to generate.
    Returns:
        None.
    """

    with open(image, "rb") as image_file:
        encoded_string = base64.b64encode(image_file.read())

    body = json.dumps({
        "anthropic_version": "bedrock-2023-05-31",
        "max_tokens": max_tokens,
        "messages": [
            {
                "role": "user",
                "content": [
                    {"type": "text", "text": input_text},
                    {"type": "image", "source": {"type": "base64",
                                                 "media_type": "image/jpeg", "data":
 encoded_string.decode('utf-8')}}
                ]
            }
        ]
    })

    response = bedrock_runtime.invoke_model_with_response_stream(
        body=body, modelId=model_id)

    for event in response.get("body"):
        chunk = json.loads(event["chunk"]["bytes"])

        if chunk['type'] == 'message_delta':
            print(f"\nStop reason: {chunk['delta']['stop_reason']}")
            print(f"Stop sequence: {chunk['delta']['stop_sequence']}")
            print(f"Output tokens: {chunk['usage']['output_tokens']}")
```

```
        if chunk['type'] == 'content_block_delta':
            if chunk['delta']['type'] == 'text_delta':
                print(chunk['delta']['text'], end="")


def main():
    """
    Entrypoint for Anthropic Claude Sonnet multimodal prompt example.
    """

    model_id = "anthropic.claude-3-sonnet-20240229-v1:0"
    input_text = "What can you tell me about this image?"
    image = "/path/to/image"
    max_tokens = 100

    try:

        bedrock_runtime = boto3.client('bedrock-runtime')

        stream_multi_modal_prompt(
            bedrock_runtime, model_id, input_text, image, max_tokens)

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))


if __name__ == "__main__":
    main()
```

AI21 Labs Jurassic-2 models

This section provides inference parameters and a code example for using AI21 Labs AI21 Labs Jurassic-2 models.

Topics

- Inference parameters
- Code example

Inference parameters

The AI21 Labs Jurassic-2 models support the following inference parameters.

Topics

Randomness and Diversity

The AI21 Labs Jurassic-2 models support the following parameters to control randomness and diversity in the response.

- **Temperature** (`temperature`)– Use a lower value to decrease randomness in the response.
- **Top P** (`topP`) – Use a lower value to ignore less probable options.

Length

The AI21 Labs Jurassic-2 models support the following parameters to control the length of the generated response.

- **Max completion length** (`maxTokens`) – Specify the maximum number of tokens to use in the generated response.
- **Stop sequences** (`stopSequences`) – Configure stop sequences that the model recognizes and after which it stops generating further tokens. Press the Enter key to insert a newline character in a stop sequence. Use the Tab key to finish inserting a stop sequence.

Repetitions

The AI21 Labs Jurassic-2 models support the following parameters to control repetition in the generated response.

- **Presence penalty** (`presencePenalty`) – Use a higher value to lower the probability of generating new tokens that already appear at least once in the prompt or in the completion.

- **Count penalty** (`countPenalty`) – Use a higher value to lower the probability of generating new tokens that already appear at least once in the prompt or in the completion. Proportional to the number of appearances.
- **Frequency penalty** (`frequencyPenalty`) – Use a high value to lower the probability of generating new tokens that already appear at least once in the prompt or in the completion. The value is proportional to the frequency of the token appearances (normalized to text length).
- **Penalize special tokens** – Reduce the probability of repetition of special characters. The default values are `true`.
  - **Whitespaces** (`applyToWhitespaces`) – A `true` value applies the penalty to whitespaces and new lines.
  - **Punctuations** (`applyToPunctuation`) – A `true` value applies the penalty to punctuation.
  - **Numbers** (`applyToNumbers`) – A `true` value applies the penalty to numbers.
  - **Stop words** (`applyToStopwords`) – A `true` value applies the penalty to stop words.
  - **Emojis** (`applyToEmojis`) – A `true` value excludes emojis from the penalty.

Model invocation request body field

When you make an [InvokeModel](#) or [InvokeModelWithResponseStream](#) call using an AI21 Labs model, fill the `body` field with a JSON object that conforms to the one below. Enter the prompt in the `prompt` field.

```
{
    "prompt": string,
    "temperature": float,
    "topP": float,
    "maxTokens": int,
    "stopSequences": [string],
    "countPenalty": {
        "scale": float
    },
    "presencePenalty": {
        "scale": float
    },
    "frequencyPenalty": {
        "scale": float
    }
}
```

To penalize special tokens, add those fields to any of the penalty objects. For example, you can modify the `countPenalty` field as follows.

```
"countPenalty": {
    "scale": float,
    "applyToWhitespaces": boolean,
    "applyToPunctuations": boolean,
    "applyToNumbers": boolean,
    "applyToStopwords": boolean,
    "applyToEmojis": boolean
}
```

The following table shows the minimum, maximum, and default values for the numerical parameters.

Category	Parameter	JSON object format	Minimum	Maximum	Default
Randomness and diversity	Temperature	temperature	0	1	0.5
	Top P	topP	0	1	0.5
Length	Max tokens (mid, ultra, and large models)	maxTokens	0	8,191	200
	Max tokens (other models)		0	2,048	200
Repetitions	Presence penalty	presencePenalty	0	5	0
	Count penalty	countPenalty	0	1	0
	Frequency penalty	frequencyPenalty	0	500	0

Model invocation response body field

For information about the format of the body field in the response, see https://docs.ai21.com/reference/j2-complete-ref.

> **Note**
>
> Amazon Bedrock returns the response identifier (`id`) as an integer value.

Code example

This examples shows how to call the *A2I AI21 Labs Jurassic-2 Mid* model.

```
import boto3
import json

brt = boto3.client(service_name='bedrock-runtime')

body = json.dumps({
    "prompt": "Translate to spanish: 'Amazon Bedrock is the easiest way to build and
 scale generative AI applications with base models (FMs)'.",
    "maxTokens": 200,
    "temperature": 0.5,
    "topP": 0.5
})

modelId = 'ai21.j2-mid-v1'
accept = 'application/json'
contentType = 'application/json'

response = brt.invoke_model(
    body=body,
    modelId=modelId,
    accept=accept,
    contentType=contentType
)

response_body = json.loads(response.get('body').read())

# text
print(response_body.get('completions')[0].get('data').get('text'))
```

Cohere models

The following is inference parameters information for the Cohere models that Amazon Bedrock supports.

Topics

- Cohere Command models
- Cohere Embed models

Cohere Command models

You make inference requests to an Cohere Command model with InvokeModel or InvokeModelWithResponseStream (streaming). You need the model ID for the model that you want to use. To get the model ID, see Amazon Bedrock model IDs.

Topics

- Request and Response
- Code example

Request and Response

Request

The Cohere Command models have the following inference parameters.

```
{
    "prompt": string,
    "temperature": float,
    "p": float,
    "k": float,
    "max_tokens": int,
    "stop_sequences": [string],
    "return_likelihoods": "GENERATION|ALL|NONE",
    "stream": boolean,
    "num_generations": int,
    "logit_bias": {token_id: bias},
    "truncate": "NONE|START|END"
}
```

The following are required parameters.

- **prompt** – (Required) The input text that serves as the starting point for generating the response.

  The following are text per call and character limits.

The following are optional parameters.

- **return_likelihoods** – Specify how and if the token likelihoods are returned with the response. You can specify the following options.
  - `GENERATION` – Only return likelihoods for generated tokens.
  - `ALL` – Return likelihoods for all tokens.
  - `NONE` – (Default) Don't return any likelihoods.
- **stream** – (Required to support streaming) Specify `true` to return the response piece-by-piece in real-time and `false` to return the complete response after the process finishes.
- **logit_bias** – Prevents the model from generating unwanted tokens or incentivizes the model to include desired tokens. The format is `{token_id: bias}` where bias is a float between -10 and 10. Tokens can be obtained from text using any tokenization service, such as Cohere's Tokenize endpoint. For more information, see Cohere documentation.

Default	Minimum	Maximum
N/A	-10 (for a token bias)	10 (for a token bias)

- **num_generations** – The maximum number of generations that the model should return.

Default	Minimum	Maximum
1	1	5

- **truncate** – Specifies how the API handles inputs longer than the maximum token length. Use one of the following:
  - `NONE` – Returns an error when the input exceeds the maximum input token length.
  - `START` – Discard the start of the input.
  - `END` – (Default) Discards the end of the input.

If you specify `START` or `END`, the model discards the input until the remaining input is exactly the maximum input token length for the model.

- **temperature** – Use a lower value to decrease randomness in the response.

Default	Minimum	Maximum
0.9	0	5

- **p** – Top P. Use a lower value to ignore less probable options. Set to 0 or 1.0 to disable. If both `p` and `k` are enabled, `p` acts after `k`.

Default	Minimum	Maximum
0.75	0	1

- **k** – Top K. Specify the number of token choices the model uses to generate the next token. If both `p` and `k` are enabled, `p` acts after `k`.

Default	Minimum	Maximum
0	0	500

- **max_tokens** – Specify the maximum number of tokens to use in the generated response.

Default	Minimum	Maximum
20	1	4096

- **stop_sequences** – Configure up to four sequences that the model recognizes. After a stop sequence, the model stops generating further tokens. The returned text doesn't contain the stop sequence.

Response

The response has the following possible fields:

```
{
    "generations": [
```

```
        {
            "finish_reason": "COMPLETE | MAX_TOKENS | ERROR | ERROR_TOXIC",
            "id": string,
            "text": string,
            "likelihood" : float,
            "token_likelihoods" : [{"token" : float}],
            "is_finished" : true | false,
            "index" : integer

        }
    ],
    "id": string,
    "prompt": string
}
```

- `generations` — A list of generated results along with the likelihoods for tokens requested. (Always returned). Each generation object in the list contains the following fields.
  - `id` — An identifier for the generation. (Always returned).
  - `likelihood` — The likelihood of the output. The value is the average of the token likelihoods in `token_likelihoods`. Returned if you specify the `return_likelihoods` input parameter.
  - `token_likelihoods` — An array of per token likelihoods. Returned if you specify the `return_likelihoods` input parameter.
  - `finish_reason` — The reason why the model finished generating tokens. `COMPLETE` - the model sent back a finished reply. `MAX_TOKENS` – the reply was cut off because the model reached the maximum number of tokens for its context length. `ERROR` – something went wrong when generating the reply. `ERROR_TOXIC` – the model generated a reply that was deemed toxic. `finish_reason` is returned only when `is_finished=true`. (Not always returned).
  - `is_finished` — A boolean field used only when `stream` is `true`, signifying whether or not there are additional tokens that will be generated as part of the streaming response. (Not always returned)
  - `text` — The generated text.
  - `index` — In a streaming response, use to determine which generation a given token belongs to. When only one response is streamed, all tokens belong to the same generation and index is not returned. `index` therefore is only returned in a streaming request with a value for `num_generations` that is larger than one.

- `prompt` — The prompt from the input request (always returned).
- `id` — An identifier for the request (always returned).

For more information, see https://docs.cohere.com/reference/generate in the Cohere documentations.

Code example

This examples shows how to call the *Cohere Command* model.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate text using a Cohere model.
"""
import json
import logging
import boto3


from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_text(model_id, body):
    """
    Generate text using a Cohere model.
    Args:
        model_id (str): The model ID to use.
        body (str) : The reqest body to use.
    Returns:
        dict: The response from the model.
    """

    logger.info("Generating text with Cohere model %s", model_id)

    accept = 'application/json'
    content_type = 'application/json'
```

```
    bedrock = boto3.client(service_name='bedrock-runtime')

    response = bedrock.invoke_model(
        body=body,
        modelId=model_id,
        accept=accept,
        contentType=content_type
    )

    logger.info("Successfully generated text with Cohere model %s", model_id)

    return response


def main():
    """
    Entrypoint for Cohere example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    model_id = 'cohere.command-text-v14'
    prompt = """Summarize this dialogue:
"Customer: Please connect me with a support agent.
AI: Hi there, how can I assist you today?
Customer: I forgot my password and lost access to the email affiliated to my account.
 Can you please help me?
AI: Yes of course. First I'll need to confirm your identity and then I can connect you
 with one of our support agents.
"""
    try:
        body = json.dumps({
            "prompt": prompt,
            "max_tokens": 200,
            "temperature": 0.6,
            "p": 1,
            "k": 0,
            "num_generations": 2,
            "return_likelihoods": "GENERATION"
        })
        response = generate_text(model_id=model_id,
                                 body=body)
```

```
        response_body = json.loads(response.get('body').read())
        generations = response_body.get('generations')

        for index, generation in enumerate(generations):

            print(f"Generation {index + 1}\n------------")
            print(f"Text:\n {generation['text']}\n")
            if 'likelihood' in generation:
                print(f"Likelihood:\n {generation['likelihood']}\n")

            print(f"Reason: {generation['finish_reason']}\n\n")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    else:
        print(f"Finished generating text with Cohere model {model_id}.")


if __name__ == "__main__":
    main()
```

Cohere Embed models

You make inference requests to an Embed model with InvokeModel You need the model ID for the model that you want to use. To get the model ID, see Amazon Bedrock model IDs.

> **Note**
>
> Amazon Bedrock doesn't support streaming responses from Cohere Embed models.

Topics

- Request and Response
- Code example

Request and Response

Request

The Cohere Embed models have the following inference parameters.

```
{
    "texts":[string],
    "input_type": "search_document|search_query|classification|clustering",
    "truncate": "NONE|START|END"
}
```

The following are required parameters.

- **texts** – (Required) An array of strings for the model to embed. For optimal performance, we recommend reducing the length of each text to less than 512 tokens. 1 token is about 4 characters.

  The following are text per call and character limits.

  Texts per call

Minimum	Maximum	
0 texts	128 texts	

  Characters

Minimum	Maximum	
0 characters	2048 characters	

The following are optional parameters.

- **input_type** – Prepends special tokens to differentiate each type from one another. You should not mix different types together, except when mixing types for for search and retrieval. In this case, embed your corpus with the `search_document` type and embedded queries with type `search_query` type.

  - `search_document` – In search use-cases, use `search_document` when you encode documents for embeddings that you store in a vector database.
  - `search_query` – Use `search_query` when querying your vector DB to find relevant documents.
  - `classification` – Use `classification` when using embeddings as an input to a text classifier.
  - `clustering` – Use `clustering` to cluster the embeddings.

- **truncate** – Specifies how the API handles inputs longer than the maximum token length. Use one of the following:
  - `NONE` – (Default) Returns an error when the input exceeds the maximum input token length.
  - `START` – Discards the start of the input.
  - `END` – Discards the end of the input.

  If you specify `START` or `END`, the model discards the input until the remaining input is exactly the maximum input token length for the model.

For more information, see https://docs.cohere.com/reference/embed in the Cohere documentation.

Response

The body response from a call to `InvokeModel` is the following:

```
{
    "embeddings": [
        [ <array of 1024 floats> ]
    ],
    "id": string,
    "response_type" : "embeddings_floats,
    "texts": [string]
}
```

The body response has the following fields:

- **id** – An identifier for the response.
- **response_type** – The response type. This value is always `embeddings_floats`.
- **embeddings** – An array of embeddings, where each embedding is an array of floats with 1024 elements. The length of the `embeddings` array will be the same as the length of the original `texts` array.
- **texts** – An array containing the text entries for which embeddings were returned.

For more information, see https://docs.cohere.com/reference/embed.

Code example

This examples shows how to call the *Cohere Embed English* model.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate text embeddings using the Cohere Embed English model.
"""
import json
import logging
import boto3


from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_text_embeddings(model_id, body):
    """
    Generate text embedding by using the Cohere Embed model.
    Args:
        model_id (str): The model ID to use.
        body (str) : The reqest body to use.
    Returns:
        dict: The response from the model.
    """

    logger.info(
```

```
        "Generating text emdeddings with the Cohere Embed model %s", model_id)

    accept = '*/*'
    content_type = 'application/json'

    bedrock = boto3.client(service_name='bedrock-runtime')

    response = bedrock.invoke_model(
        body=body,
        modelId=model_id,
        accept=accept,
        contentType=content_type
    )

    logger.info("Successfully generated text with Cohere model %s", model_id)

    return response


def main():
    """
    Entrypoint for Cohere Embed example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    model_id = 'cohere.embed-english-v3'
    text1 = "hello world"
    text2 = "this is a test"
    input_type = "search_document"

    try:

        body = json.dumps({
            "texts": [
                text1,
                text2],
            "input_type": input_type}
        )
        response = generate_text_embeddings(model_id=model_id,
                                            body=body)

        response_body = json.loads(response.get('body').read())
```

```
        print(f"ID: {response_body.get('id')}")
        print(f"Response type: {response_body.get('response_type')}")

        print("Embeddings")
        for i, embedding in enumerate(response_body.get('embeddings')):
            print(f"\tEmbedding {i}")
            print(*embedding)

        print("Texts")
        for i, text in enumerate(response_body.get('texts')):
            print(f"\tText {i}: {text}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    else:
        print(
            f"Finished generating text embeddings with Cohere model {model_id}.")


if __name__ == "__main__":
    main()
```

Meta Llama models

This section provides inference parameters and a code example for using the following models from Meta.

- Llama 2
- Llama 2 Chat
- Llama 3 Instruct

You make inference requests to Meta Llama models with InvokeModel or InvokeModelWithResponseStream (streaming). You need the model ID for the model that you want to use. To get the model ID, see Amazon Bedrock model IDs.

Topics

- Request and response

- Example code

Request and response

The request body is passed in the body field of a request to InvokeModel or InvokeModelWithResponseStream.

Request

Meta Llama 2 Chat and Llama 2 models have the following inference parameters.

```
{
    "prompt": string,
    "temperature": float,
    "top_p": float,
    "max_gen_len": int
}
```

The following are required parameters.

- **prompt** – (Required) The prompt that you want to pass to the model.

  For information about prompt formats, see Meta Llama 2 and Meta Llama 3.

The following are optional parameters.

- **temperature** – Use a lower value to decrease randomness in the response.

Default	Minimum	Maximum
0.5	0	1

- **top_p** – Use a lower value to ignore less probable options. Set to 0 or 1.0 to disable.

Default	Minimum	Maximum
0.9	0	1

- **max_gen_len** – Specify the maximum number of tokens to use in the generated response. The model truncates the response once the generated text exceeds `max_gen_len`.

Default	Minimum	Maximum
512	1	2048

Response

Meta Llama 2 and Llama 2 Chat models return the following fields for a text completion inference call.

```
{
    "generation": "\n\n<response>",
    "prompt_token_count": int,
    "generation_token_count": int,
    "stop_reason" : string
}
```

More information about each field is provided below.

- **generation** – The generated text.
- **prompt_token_count** – The number of tokens in the prompt.
- **generation_token_count** – The number of tokens in the generated text.
- **stop_reason** – The reason why the response stopped generating text. Possible values are:
  - **stop** – The model has finished generating text for the input prompt.
  - **length** – The length of the tokens for the generated text exceeds the value of `max_gen_len` in the call to `InvokeModel` (`InvokeModelWithResponseStream`, if you are streaming output). The response is truncated to `max_gen_len` tokens. Consider increasing the value of `max_gen_len` and trying again.

Example code

This example shows how to call the *Meta Llama 2 Chat 13B* model.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate text with Meta Llama 2 Chat (on demand).
```

```
"""

import json
import logging
import boto3


from botocore.exceptions import ClientError


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_text(model_id, body):
    """
    Generate an image using Meta Llama 2 Chat on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (JSON): The text that the model generated, token information, and the
        reason the model stopped generating text.
    """

    logger.info("Generating image with Meta Llama 2 Chat model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )

    response_body = json.loads(response.get('body').read())

    return response_body


def main():
    """
    Entrypoint for Meta Llama 2 Chat example.
```

```
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    model_id = 'meta.llama2-13b-chat-v1'
    prompt = """What is the average lifespan of a Llama?"""
    max_gen_len = 128
    temperature = 0.1
    top_p = 0.9


    # Create request body.
    body = json.dumps({
        "prompt": prompt,
        "max_gen_len": max_gen_len,
        "temperature": temperature,
        "top_p": top_p
    })


    try:

        response = generate_text(model_id, body)

        print(f"Generated Text: {response['generation']}")
        print(f"Prompt Token count:  {response['prompt_token_count']}")
        print(f"Generation Token count:  {response['generation_token_count']}")
        print(f"Stop reason:  {response['stop_reason']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))

    else:
        print(
            f"Finished generating text with Meta Llama 2 Chat model {model_id}.")


if __name__ == "__main__":
    main()
```

Mistral AI models

You make inference requests to Mistral AI models with InvokeModel or InvokeModelWithResponseStream (streaming). You need the model ID for the model that you want to use. To get the model ID, see Amazon Bedrock model IDs.

Mistral AI models are available under the Apache 2.0 license. For more information about using Mistral AI models, see the Mistral AI documentation.

Topics

- Supported models
- Request and Response
- Code example

Supported models

You can use following Mistral AI models.

- Mistral 7B Instruct
- Mixtral 8X7B Instruct
- Mistral Large

Request and Response

Request

The Mistral AI models have the following inference parameters.

```
{
    "prompt": string,
    "max_tokens" : int,
    "stop" : [string],
    "temperature": float,
    "top_p": float,
    "top_k": int
}
```

The following are required parameters.

- **prompt** – (Required) The prompt that you want to pass to the model, as shown in the following example.

```
<s>[INST] What is your favourite condiment? [/INST]
```

  The following example shows how to format is a multi-turn prompt.

```
<s>[INST] What is your favourite condiment? [/INST]
Well, I'm quite partial to a good squeeze of fresh lemon juice.
It adds just the right amount of zesty flavour to whatever I'm cooking up in the
 kitchen!</s>
[INST] Do you have mayonnaise recipes? [/INST]
```

  Text for the user role is inside the [INST]...[/INST] tokens, text outside is the assistant role. The beginning and ending of a string are represented by the <s> (beginning of string) and </s> (end of string) tokens. For information about sending a chat prompt in the correct format, see Chat template in the Mistral AI documentation.

The following are optional parameters.

- **max_tokens** – Specify the maximum number of tokens to use in the generated response. The model truncates the response once the generated text exceeds max_tokens.

Default	Minimum	Maximum
Mistral 7B Instruct – 512 Mixtral 8X7B Instruct – 512 Mistral Large – 8,192	1	Mistral 7B Instruct – 8,192 Mixtral 8X7B Instruct – 4,096 Mistral Large – 8,192

- **stop** – A list of stop sequences that if generated by the model, stops the model from generating further output.

Default	Minimum	Maximum
0	0	10

- **temperature** – Controls the randomness of predictions made by the model. For more information, see Inference parameters.

Default	Minimum	Maximum
Mistral 7B Instruct – 0.5 Mixtral 8X7B Instruct – 0.5 Mistral Large – 0.7	0	1

- **top_p** – Controls the diversity of text that the model generates by setting the percentage of most-likely candidates that the model considers for the next token. For more information, see Inference parameters.

Default	Minimum	Maximum
Mistral 7B Instruct – 0.9 Mixtral 8X7B Instruct – 0.9 Mistral Large – 1	0	1

- **top_k** – Controls the number of most-likely candidates that the model considers for the next token. For more information, see Inference parameters.

Default	Minimum	Maximum
Mistral 7B Instruct – 50 Mixtral 8X7B Instruct – 50 Mistral Large – disabled	1	200

Response

The body response from a call to `InvokeModel` is the following:

```
{
```

```
  "outputs": [
    {
        "text": string,
        "stop_reason": string
    }
  ]
}
```

The body response has the following fields:

- **outputs** – A list of outputs from the model. Each output has the following fields.
  - **text** – The text that the model generated.
  - **stop_reason** – The reason why the response stopped generating text. Possible values are:
    - **stop** – The model has finished generating text for the input prompt. The model stops because it has no more content to generate or if the model generates one of the stop sequences that you define in the `stop` request parameter.
    - **length** – The length of the tokens for the generated text exceeds the value of `max_tokens` in the call to `InvokeModel` (`InvokeModelWithResponseStream`, if you are streaming output). The response is truncated to `max_tokens` tokens.

Code example

This examples shows how to call the Mistral 7B Instruct model.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate text using a Mistral AI model.
"""
import json
import logging
import boto3


from botocore.exceptions import ClientError

logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)
```

```
def generate_text(model_id, body):
    """
    Generate text using a Mistral AI model.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        JSON: The response from the model.
    """

    logger.info("Generating text with Mistral AI model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    response = bedrock.invoke_model(
        body=body,
        modelId=model_id
    )

    logger.info("Successfully generated text with Mistral AI model %s", model_id)

    return response


def main():
    """
    Entrypoint for Mistral AI example.
    """

    logging.basicConfig(level=logging.INFO,
                        format="%(levelname)s: %(message)s")

    try:
        model_id = 'mistral.mistral-7b-instruct-v0:2'

        prompt = """<s>[INST] In Bash, how do I list all text files in the current
 directory
          (excluding subdirectories) that have been modified in the last month? [/
INST]"""

        body = json.dumps({
            "prompt": prompt,
```

```
            "max_tokens": 400,
            "temperature": 0.7,
            "top_p": 0.7,
            "top_k": 50
        })

        response = generate_text(model_id=model_id,
                                 body=body)

        response_body = json.loads(response.get('body').read())

        outputs = response_body.get('outputs')

        for index, output in enumerate(outputs):

            print(f"Output {index + 1}\n----------")
            print(f"Text:\n{output['text']}\n")
            print(f"Stop reason: {output['stop_reason']}\n")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    else:
        print(f"Finished generating text with Mistral AI model {model_id}.")


if __name__ == "__main__":
    main()
```

Stability.ai Diffusion models

The following is inference parameters information for the Stability.ai Diffusion models that Amazon Bedrock supports.

Models

- Stability.ai Diffusion 0.8
- Stability.ai Diffusion 1.0 text to image
- Stability.ai Diffusion 1.0 image to image
- Stability.ai Diffusion 1.0 image to image (masking)

Stability.ai Diffusion 0.8

The Stability.ai Diffusion models have the following controls.

- **Prompt strength** (`cfg_scale`) – Determines how much the final image portrays the prompt. Use a lower number to increase randomness in the generation.
- **Generation step** (`steps`) – Generation step determines how many times the image is sampled. More steps can result in a more accurate result.
- **Seed** (`seed`) – The seed determines the initial noise setting. Use the same seed and the same settings as a previous run to allow inference to create a similar image. If you don't set this value, it is set as a random number.

Model invocation request body field

When you make an InvokeModel or InvokeModelWithResponseStream call using a Stability.ai model, fill the `body` field with a JSON object that conforms to the one below. Enter the prompt in the `text` field in the `text_prompts` object.

```
{
    "text_prompts": [
        {"text": "string"}
    ],
    "cfg_scale": float,
    "steps": int,
    "seed": int
}
```

The following table shows the minimum, maximum, and default values for the numerical parameters.

Parameter	JSON object format	Minimum	Maximum	Default
Prompt strength	cfg_scale	0	30	10
Generation step	steps	10	150	30

Model invocation response body field

For information about the format of the body field in the response, see https://platform.stability.ai/docs/api-reference#tag/v1generation.

Stability.ai Diffusion 1.0 text to image

The Stability.ai Diffusion 1.0 model has the following inference parameters and model response for making text to image inference calls.

Topics

- Request and Response
- Code example

Request and Response

The request body is passed in the body field of a request to InvokeModel or InvokeModelWithResponseStream.

For more information, see https://platform.stability.ai/docs/api-reference#tag/v1generation.

Request

The Stability.ai Diffusion 1.0 model has the following inference parameters for a text to image inference call.

```
{
        "text_prompts": [
            {
                "text": string,
                "weight": float
            }
        ],
        "height": int,
        "width": int,
        "cfg_scale": float,
        "clip_guidance_preset": string,
        "sampler": string,
        "samples",
        "seed": int,
        "steps": int,
        "style_preset": string,
```

```
        "extras" :JSON object

}
```

- **text_prompts** (Required) – An array of text prompts to use for generation. Each element is a JSON object that contains a prompt and a weight for the prompt.
  - **text** – The prompt that you want to pass to the model.

Minimum	Maximum	
0	2000	

  - **weight** (Optional) – The weight that the model should apply to the prompt. A value that is less than zero declares a negative prompt. Use a negative prompt to tell the model to avoid certain concepts. The default value for `weight` is one.
- **cfg_scale** – (Optional) Determines how much the final image portrays the prompt. Use a lower number to increase randomness in the generation.

Minimum	Maximum	Default
0	35	7

- **clip_guidance_preset**– (Optional) Enum: `FAST_BLUE, FAST_GREEN, NONE, SIMPLE SLOW, SLOWER, SLOWEST`.
- **height** – (Optional) Height of the image to generate, in pixels, in an increment divible by 64.

  The value must be one of `1024x1024, 1152x896, 1216x832, 1344x768, 1536x640, 640x1536, 768x1344, 832x1216, 896x1152`.
- **width** – (Optional) Width of the image to generate, in pixels, in an increment divible by 64.

  The value must be one of `1024x1024, 1152x896, 1216x832, 1344x768, 1536x640, 640x1536, 768x1344, 832x1216, 896x1152`.
- **sampler** – (Optional) The sampler to use for the diffusion process. If this value is omitted, the model automatically selects an appropriate sampler for you.

  Enum: `DDIM, DDPM, K_DPMPP_2M, K_DPMPP_2S_ANCESTRAL, K_DPM_2, K_DPM_2_ANCESTRAL, K_EULER, K_EULER_ANCESTRAL, K_HEUN K_LMS`.

- **samples** – (Optional) The number of image to generate. Currently Amazon Bedrock supports generating one image. If you supply a value for `samples`, the value must be one.

Default	Minimum	Maximum
1	1	1

- **seed** – (Optional) The seed determines the initial noise setting. Use the same seed and the same settings as a previous run to allow inference to create a similar image. If you don't set this value, or the value is 0, it is set as a random number.

Minimum	Maximum	Default
0	4294967295	0

- **steps** – (Optional) Generation step determines how many times the image is sampled. More steps can result in a more accurate result.

Minimum	Maximum	Default
10	50	30

- **style_preset** (Optional) – A style preset that guides the image model towards a particular style. This list of style presets is subject to change.

  Enum: `3d-model, analog-film, anime, cinematic, comic-book, digital-art, enhance, fantasy-art, isometric, line-art, low-poly, modeling-compound, neon-punk, origami, photographic, pixel-art, tile-texture`.
- **extras** (Optional) – Extra parameters passed to the engine. Use with caution. These parameters are used for in-development or experimental features and might change without warning.

Response

The Stability.ai Diffusion 1.0 model returns the following fields for a text to image inference call.

```
{
```

```
    "result": string,
    "artifacts": [
        {
            "seed": int,
            "base64": string,
            "finishReason": string
        }
    ]
}
```

- **result** – The result of the operation. If successful, the response is `success`.
- **artifacts** – An array of images, one for each requested image.
  - **seed** – The value of the seed used to generate the image.
  - **base64** – The base64 encoded image that the model generated.
  - **finishedReason** – The result of the image generation process. Valid values are:
    - **SUCCESS** – The image generation process succeeded.
    - **ERROR** – An error occured.
    - **CONTENT_FILTERED** – The content filter filtered the image and the image might be blurred.

Code example

The following example shows how to run inference with the Stability.ai Diffusion 1.0 model and on demand throughput. The example submits a text prompt to a model, retrieves the response from the model, and finally shows the image.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate an image with SDXL 1.0 (on demand).
"""
import base64
import io
import json
import logging
import boto3
from PIL import Image

from botocore.exceptions import ClientError
```

```
class ImageError(Exception):
    "Custom exception for errors returned by SDXL"
    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_image(model_id, body):
    """
    Generate an image using SDXL 1.0 on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        image_bytes (bytes): The image generated by the model.
    """

    logger.info("Generating image with SDXL model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())
    print(response_body['result'])

    base64_image = response_body.get("artifacts")[0].get("base64")
    base64_bytes = base64_image.encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)

    finish_reason = response_body.get("artifacts")[0].get("finishReason")

    if finish_reason == 'ERROR' or finish_reason == 'CONTENT_FILTERED':
        raise ImageError(f"Image generation error. Error code is {finish_reason}")
```

```
    logger.info("Successfully generated image withvthe SDXL 1.0 model %s", model_id)

    return image_bytes



def main():
    """
    Entrypoint for SDXL example.
    """

    logging.basicConfig(level = logging.INFO,
                        format = "%(levelname)s: %(message)s")

    model_id='stability.stable-diffusion-xl-v1'

    prompt="""Sri lanka tea plantation."""

    # Create request body.
    body=json.dumps({
        "text_prompts": [
        {
        "text": prompt
        }
    ],
    "cfg_scale": 10,
    "seed": 0,
    "steps": 50,
    "samples" : 1,
    "style_preset" : "photographic"

    })

    try:
        image_bytes=generate_image(model_id = model_id,
                                 body = body)
        image = Image.open(io.BytesIO(image_bytes))
        image.show()


    except ClientError as err:
        message=err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
```

```
                format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(f"Finished generating text with SDXL model {model_id}.")


if __name__ == "__main__":
    main()
```

Stability.ai Diffusion 1.0 image to image

The Stability.ai Diffusion 1.0 model has the following inference parameters and model response for making image to image inference calls.

Topics

- Request and Response
- Code example

Request and Response

The request body is passed in the body field of a request to InvokeModel or InvokeModelWithResponseStream.

For more information, see https://platform.stability.ai/docs/api-reference#tag/v1generation/operation/imageToImage.

Request

The Stability.ai Diffusion 1.0 model has the following inference parameters for an image to image inference call.

```
{
        "text_prompts": [
            {
                "text": string,
                "weight": float
```

```
            }
        ],
        "init_image" : string ,
        "init_image_mode" : string,
        "image_strength" : float,
        "cfg_scale": float,
        "clip_guidance_preset": string,
        "sampler": string,
        "samples" : int,
        "seed": int,
        "steps": int,
        "style_preset": string,
        "extras" : json object
}
```

The following are required parameters.

- **text_prompts** – (Required) An array of text prompts to use for generation. Each element is a JSON object that contains a prompt and a weight for the prompt.
  - **text** – The prompt that you want to pass to the model.

Minimum	Maximum	
0	2000	

  - **weight** – (Optional) The weight that the model should apply to the prompt. A value that is less than zero declares a negative prompt. Use a negative prompt to tell the model to avoid certain concepts. The default value for `weight` is one.
- **init_image** – (Required) The base64 encoded image that you want to use to initialize the diffusion process.

The following are optional parameters.

- **init_image_mode** – (Optional) Determines whether to use `image_strength` or `step_schedule_*` to control how much influence the image in `init_image` has on the result. Possible values are `IMAGE_STRENGTH` or `STEP_SCHEDULE`. The default is IMAGE_STRENGTH.

- **image_strength** – (Optional) Determines how much influence the source image in `init_image` has on the diffusion process. Values close to 1 yield images very similar to the source image. Values close to 0 yield images very different than the source image.
- **cfg_scale** – (Optional) Determines how much the final image portrays the prompt. Use a lower number to increase randomness in the generation.

Default	Minimum	Maximum
7	0	35

- **clip_guidance_preset** – (Optional) Enum: `FAST_BLUE, FAST_GREEN, NONE, SIMPLE, SLOW, SLOWER, SLOWEST`.
- **sampler** – (Optional) The sampler to use for the diffusion process. If this value is omitted, the model automatically selects an appropriate sampler for you.

  Enum: `DDIM DDPM, K_DPMPP_2M, K_DPMPP_2S_ANCESTRAL, K_DPM_2, K_DPM_2_ANCESTRAL, K_EULER, K_EULER_ANCESTRAL, K_HEUN K_LMS`.
- **samples** – (Optional) The number of image to generate. Currently Amazon Bedrock supports generating one image. If you supply a value for `samples`, the value must be one.

Default	Minimum	Maximum
1	1	1

- **seed** – (Optional) The seed determines the initial noise setting. Use the same seed and the same settings as a previous run to allow inference to create a similar image. If you don't set this value, or the value is 0, it is set as a random number.

Default	Minimum	Maximum
0	0	4294967295

- **steps** – (Optional) Generation step determines how many times the image is sampled. More steps can result in a more accurate result.

Default	Minimum	Maximum
30	10	50

- **style_preset** – (Optional) A style preset that guides the image model towards a particular style. This list of style presets is subject to change.

  Enum: `3d-model, analog-film, anime, cinematic, comic-book, digital-art, enhance, fantasy-art, isometric, line-art, low-poly, modeling-compound, neon-punk, origami, photographic, pixel-art, tile-texture`
- **extras** – (Optional) Extra parameters passed to the engine. Use with caution. These parameters are used for in-development or experimental features and might change without warning.

Response

The Stability.ai Diffusion 1.0 model returns the following fields for a text to image inference call.

```
{
    "result": string,
    "artifacts": [
        {
            "seed": int,
            "base64": string,
            "finishReason": string
        }
    ]
}
```

- **result** – The result of the operation. If successful, the response is `success`.
- **artifacts** – An array of images, one for each requested image.
  - **seed** – The value of the seed used to generate the image.
  - **base64** – The base64 encoded image that the model generated.
  - **finishedReason** – The result of the image generation process. Valid values are:
    - **SUCCESS** – The image generation process succeeded.

- **ERROR** – An error occured.
- **CONTENT_FILTERED** – The content filter filtered the image and the image might be blurred.

Code example

The following example shows how to run inference with the Stability.ai Diffusion 1.0 model and on demand throughput. The example submits a text prompt and reference image to a model, retrieves the response from the model, and finally shows the image.

```
# Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
# SPDX-License-Identifier: Apache-2.0
"""
Shows how to generate an image from a reference image with SDXL 1.0 (on demand).
"""
import base64
import io
import json
import logging
import boto3
from PIL import Image

from botocore.exceptions import ClientError

class ImageError(Exception):
    "Custom exception for errors returned by SDXL"
    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_image(model_id, body):
    """
    Generate an image using SDXL 1.0 on demand.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        image_bytes (bytes): The image generated by the model.
```

```
    """

    logger.info("Generating image with SDXL model %s", model_id)

    bedrock = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = bedrock.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())
    print(response_body['result'])

    base64_image = response_body.get("artifacts")[0].get("base64")
    base64_bytes = base64_image.encode('ascii')
    image_bytes = base64.b64decode(base64_bytes)

    finish_reason = response_body.get("artifacts")[0].get("finishReason")

    if finish_reason == 'ERROR' or finish_reason == 'CONTENT_FILTERED':
        raise ImageError(f"Image generation error. Error code is {finish_reason}")


    logger.info("Successfully generated image withvthe SDXL 1.0 model %s", model_id)

    return image_bytes



def main():
    """
    Entrypoint for SDXL example.
    """

    logging.basicConfig(level = logging.INFO,
                        format = "%(levelname)s: %(message)s")

    model_id='stability.stable-diffusion-xl-v1'

    prompt="""A space ship."""

    # Read reference image from file and encode as base64 strings.
```

```
    with open("/path/to/image", "rb") as image_file:
        init_image = base64.b64encode(image_file.read()).decode('utf8')

    # Create request body.
    body=json.dumps({
        "text_prompts": [
        {
        "text": prompt
        }
    ],
    "init_image": init_image,
    "style_preset" : "isometric"
    })

    try:
        image_bytes=generate_image(model_id = model_id,
                                 body = body)
        image = Image.open(io.BytesIO(image_bytes))
        image.show()


    except ClientError as err:
        message=err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(f"Finished generating text with SDXL model {model_id}.")


if __name__ == "__main__":
    main()
```

Stability.ai Diffusion 1.0 image to image (masking)

The Stability.ai Diffusion 1.0 model has the following inference parameters and model response for using masks with image to image inference calls.

Request and Response

The request body is passed in the body field of a request to InvokeModel or InvokeModelWithResponseStream.

For more information, see https://platform.stability.ai/docs/api-reference#tag/v1generation/operation/masking.

Request

The Stability.ai Diffusion 1.0 model has the following inference parameters for an image to image (masking) inference call.

```
{
        "text_prompts": [
            {
                "text": string,
                "weight": float
            }
        ],
        "init_image" : string ,
        "mask_source" : string,
        "mask_image" : string,
        "cfg_scale": float,
        "clip_guidance_preset": string,
        "sampler": string,
        "samples" : int,
        "seed": int,
        "steps": int,
        "style_preset": string,
        "extras" : json object
}
```

The following are required parameters.

- **text_prompt** – (Required) An array of text prompts to use for generation. Each element is a JSON object that contains a prompt and a weight for the prompt.
  - **text** – The prompt that you want to pass to the model.

Minimum	Maximum	
0	2000	

  - **weight** – (Optional) The weight that the model should apply to the prompt. A value that is less than zero declares a negative prompt. Use a negative prompt to tell the model to avoid certain concepts. The default value for `weight` is one.
- **init_image** – (Required) The base64 encoded image that you want to use to initialize the diffusion process.
- **mask_source** – (Required) Determines where to source the mask from. Possible values are:
  - **MASK_IMAGE_WHITE** – Use the white pixels of the mask image in `mask_image` as the mask. White pixels are replaced and black pixels are left unchanged.
  - **MASK_IMAGE_BLACK** – Use the black pixels of the mask image in `mask_image` as the mask. Black pixels are replaced and white pixels are left unchanged.
  - **INIT_IMAGE_ALPHA** – Use the alpha channel of the image in `init_image` as the mask, Fully transparent pixels are replaced and fully opaque pixels are left unchanged.
- **mask_image** – (Required) The base64 encoded mask image that you want to use as a mask for the source image in `init_image`. Must be the same dimensions as the source image. Use the `mask_source` option to specify which pixels should be replaced.

The following are optional parameters.

- **cfg_scale** – (Optional) Determines how much the final image portrays the prompt. Use a lower number to increase randomness in the generation.

Default	Minimum	Maximum
7	0	35

- **clip_guidance_preset** – (Optional) Enum: `FAST_BLUE, FAST_GREEN, NONE, SIMPLE, SLOW, SLOWER, SLOWEST`.
- **sampler** – (Optional) The sampler to use for the diffusion process. If this value is omitted, the model automatically selects an appropriate sampler for you.

Enum: `DDIM, DDPM, K_DPMPP_2M, K_DPMPP_2S_ANCESTRAL, K_DPM_2, K_DPM_2_ANCESTRAL, K_EULER, K_EULER_ANCESTRAL, K_HEUN K_LMS`.

- **samples** – (Optional) The number of image to generate. Currently Amazon Bedrock supports generating one image. If you supply a value for `samples`, the value must be one. generates

Default	Minimum	Maximum
1	1	1

- **seed** – (Optional) The seed determines the initial noise setting. Use the same seed and the same settings as a previous run to allow inference to create a similar image. If you don't set this value, or the value is 0, it is set as a random number.

Default	Minimum	Maximum
0	0	4294967295

- **steps** – (Optional) Generation step determines how many times the image is sampled. More steps can result in a more accurate result.

Default	Minimum	Maximum
30	10	50

- **style_preset** – (Optional) A style preset that guides the image model towards a particular style. This list of style presets is subject to change.

  Enum: `3d-model, analog-film, anime, cinematic, comic-book, digital-art, enhance, fantasy-art, isometric, line-art, low-poly, modeling-compound, neon-punk, origami, photographic, pixel-art, tile-texture`

- **extras** – (Optional) Extra parameters passed to the engine. Use with caution. These parameters are used for in-development or experimental features and might change without warning.

Response

The Stability.ai Diffusion 1.0 model returns the following fields for a text to image inference call.

```
{
    "result": string,
    "artifacts": [
        {
            "seed": int,
            "base64": string,
            "finishReason": string
        }
    ]
}
```

- **result** – The result of the operation. If successful, the response is `success`.
- **artifacts** – An array of images, one for each requested image.
  - **seed** – The value of the seed used to generate the image.
  - **base64** – The base64 encoded image that the model generated.
  - **finishedReason** – The result of the image generation process. Valid values are:
    - **SUCCESS** – The image generation process succeeded.
    - **ERROR** – An error occured.
    - **CONTENT_FILTERED** – The content filter filtered the image and the image might be blurred.

Custom model hyperparameters

The following reference content covers the hyperparameters that are available for training each Amazon Bedrock custom model.

A hyperparameter is a parameter that controls the training process, such as the learning rate or epoch count. You set hyperparameters for custom model training when you submit the fine tuning job with the Amazon Bedrock console or by calling the CreateModelCustomizationJob API operation. For guidelines on hyperparameter settings, see Guidelines for model customization.

Topics

- Amazon Titan text model customization hyperparameters
- Amazon Titan Image Generator G1 model customization hyperparameters
- Amazon Titan Multimodal Embeddings G1 customization hyperparameters
- Cohere Command model customization hyperparameters
- Meta Llama 2 model customization hyperparameters

Amazon Titan text model customization hyperparameters

Titan text models support the following hyperparameters for model customization.

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
Epochs	epochCount	The number of iterations through the entire training dataset	integer	1	10	5
Batch size	batchSize	The number of samples processed before updating model parameters	integer	1	64	1
Learning rate	learningRate	The rate at which model parameter	float	0.0	1	1.00E-5

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
		s are updated after each batch				
Learning rate warmup steps	learningRateWarmupSteps	The number of iterations over which the learning rate is gradually increased to the specified rate	integer	0	250	5

Amazon Titan Image Generator G1 model customization hyperparameters

The Amazon Titan Image Generator G1 model supports the following hyperparameters for model customization.

> **Note**
>
> `stepCount` has no default value and must be specified. `stepCount` supports the value `auto`. `auto` prioritizes model performance over training cost by automatically determining a number based on the size of your dataset. Training job costs depend on the number that `auto` determines. To understand how job cost is calculated and to see examples, see Amazon Bedrock Pricing.

Hyperparameter (console)	Hyperparameter (API)	Definition	Minimum	Maximum	Default
Batch size	batchSize	Number of samples processed before updating model parameters	8	192	8
Steps	stepCount	Number of times the model is exposed to each batch	10	40,000	N/A
Learning rate	learningRate	Rate at which model parameters are updated after each batch	1.00E-7	1	1.00E-5

Amazon Titan Multimodal Embeddings G1 customization hyperparameters

The Amazon Titan Multimodal Embeddings G1 model supports the following hyperparameters for model customization.

> **Note**
>
> `epochCount` has no default value and must be specified. `epochCount` supports the value `Auto`. `Auto` prioritizes model performance over training cost by automatically determining a number based on the size of your dataset. Training job costs depend on the number

that `Auto` determines. To understand how job cost is calculated and to see examples, see Amazon Bedrock Pricing.

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
Epochs	epochCount	The number of iterations through the entire training dataset	integer	1	100	N/A
Batch size	batchSize	The number of samples processed before updating model parameters	integer	256	9,216	576
Learning rate	learningRate	The rate at which model parameters are updated after each batch	float	5.00E-8	1	5.00E-5

Cohere Command model customization hyperparameters

The Cohere Command and Cohere Command Light models support the following hyperparameters for model customization. For more information, see Custom models.

For information about fine tuning Cohere models, see the Cohere documentation at https://docs.cohere.com/docs/fine-tuning.

> **Note**
>
> The `epochCount` quota is adjustable.

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
Epochs	epochCount	The number of iterations through the entire training dataset	integer	1	100	1
Batch size	batchSize	The number of samples processed before updating model parameters	integer	8	8 (Command) 32 (Light)	8
Learning rate	learningRate	The rate at which model	float	5.00E-6	0.1	1.00E-5

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
		parameters are updated after each batch. If you use a validation dataset, we recommend that you don't provide a value for `learningRate` .				
Early stopping threshold	earlyStoppingThreshold	The minimum improvement in loss required to prevent premature termination of the training process	float	0	0.1	0.01

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
Early stopping patience	earlyStoppingPatience	The tolerance for stagnation in the loss metric before stopping the training process	integer	1	10	6
Evaluation percentage	evalPercentage	The percentage of the dataset allocated for model evaluation, if you don't provide a separate validation dataset	float	5	50	20

Meta Llama 2 model customization hyperparameters

The Meta Llama 2 13B and 70B models support the following hyperparameters for model customization. For more information, see Custom models.

For information about fine tuning Meta Llama models, see the Meta documentation at https://ai.meta.com/llama/get-started/#fine-tuning.

> **Note**
>
> The `epochCount` quota is adjustable.

Hyperparameter (console)	Hyperparameter (API)	Definition	Type	Minimum	Maximum	Default
Epochs	epochCount	The number of iterations through the entire training dataset	integer	1	10	5
Batch size	batchSize	The number of samples processed before updating model parameters	integer	1	1	1
Learning rate	learningRate	The rate at which model parameters are updated after each batch	float	5.00E-6	0.1	1.00E-4

Amazon Bedrock console overview

The Amazon Bedrock console provides the following features.

Features

- Getting started
- Foundation models
- Playgrounds
- Safeguards
- Orchestration
- Assessment and deployment
- Model access
- Model invocation logging

To open the Amazon Bedrock console, sign in at https://console.aws.amazon.com/bedrock/home.

Getting started

From **Getting started** in the navigation pane, you can get an **Overview** of the foundation models, examples, and playgrounds that Amazon Bedrock provides. You can also get **Examples** of the prompts you can use with Amazon Bedrock models.

The examples page shows example prompts for the available models. You can search the examples and filter the list of examples using one or more of the following attributes:

- Model
- Modality (text, image, or embedding)
- Category
- Provider

Filter the example prompts by choosing the **Search in examples** edit box and then selecting the filter that you want to apply to the search. Apply multiple filters by again choosing **Search in examples** and then selecting another filter.

When you choose an example, the Amazon Bedrock console displays the following information about the example:

- A description of what the example accomplishes.
- The model name (and model provider) where the example runs.
- The example prompt and the expected response.
- The inference configuration parameter settings for the example.
- The API request that runs the example.

To run the example, choose **Open in playground**.

Foundation models

From **Foundation models** in the navigation pane, you can view the available **Base models**, and group them by various attributes. You can also filter the model view, search for models, and view information about the model providers.

You can customize a base foundation model to improve the model's performance on specific tasks or teach the model a new domain of knowledge. Choose **Custom models** under foundation models to create and manage your custom models. Customize a model by creating a model customization job with a training dataset that you provide. For more information, see Custom models.

You can experiment with base models and custom models by using the console playgrounds.

Playgrounds

The console playgrounds are where you can experiment with models before deciding to use them in an application. There are three playgrounds.

Chat playground

The chat playground lets you experiment with the chat models that Amazon Bedrock provides. You can submit a chat to a model and the chat playground shows the response from the model and includes model metrics. Optionally, choose **Compare mode** to compare the output from up to three models. For more information, see Chat playground.

Text playground

The text playground lets you experiment with the text models that Amazon Bedrock provides. You can submit text to a model and the text playground shows the text that the model generates from the prompt. For more information, see Text playground.

Image playground

The image playground lets you experiment with the image models that Amazon Bedrock provides. You can submit a text prompt to a model and the image playground shows the image that the model generates for the prompt. For more information, see Image playground.

In the console, access the playgrounds by choosing **Playgrounds** in the navigation pane. For more information, see Playgrounds.

Safeguards

Titan Image Generator G1 automatically puts an invisible watermark on all images created by the model. Watermark detection detects if the image was generated by Titan Image Generator G1. To use watermark detection, choose **Overview** in the left navigation pane and then **Build and Test** tab. Go to the **Safeguards** section and choose **View watermark detection**. For more information, see Watermark detection.

Orchestration

With Amazon Bedrock, you can enable a Retrieval-Augmented Generation (RAG)ß workflow by using knowledge bases to build contextual applications by using the reasoning capabilities of LLMs. To use a knowledge base, choose **Orchestration** in the left navigation pane and then **Knowledge base**. For more information, see Knowledge bases for Amazon Bedrock.

Agents for Amazon Bedrock enables developers to configure an agent to complete actions based on organization data and user input. For example you might create an agent to take actions to fulfill a customer's request. To use an Agent, choose **Orchestration** in the left navigation pane and then **Agent**. For more information, see Agents for Amazon Bedrock.

Assessment and deployment

As you use Amazon Bedrock models, you need to to assess their performance and to deploy them into your solutions.

With Model Evaluation, you can evaluate and compare model output, and then choose the one best suited for your applications. Choose **Assessment and deployment** and then choose **Model evaluation**.

When you configure Provisioned Throughput for a model, you receive a level of throughput at a fixed cost. To provision throughput, choose **Assessment and deployment** in the navigation pane and then **Provisioned Throughput**. For more information, see Provisioned Throughput for Amazon Bedrock.

Model access

To use a model in Amazon Bedrock, you must first request access to the model. On the left navigation pane, choose **Model access**. For more information, see Model access.

Model invocation logging

You can log model invocation events by choosing **Settings** in the left navigation pane. For more information, see Model invocation logging.

Run model inference

Inference refers to the process of generating an output from an input provided to a model. Foundation models use probability to construct the words in a sequence. Given an input, the model predicts a probable sequence of tokens that follows, and returns that sequence as the output. Amazon Bedrock provides you the capability of running inference in the foundation model of your choice. When you run inference, you provide the following inputs.

- **Prompt** – An input provided to the model in order for it to generate a response. For information about writing prompts, see Prompt engineering guidelines.
- **Inference parameters** – A set of values that can be adjusted to limit or influence the model response. For information about inference parameters, see Inference parameters and Inference parameters for foundation models.

Amazon Bedrock offers a suite of foundation models that you can use to generate outputs of the following modalities. To see modality support by foundation model, refer to Supported foundation models in Amazon Bedrock.

Output modality	Description	Example use cases
Text	Provide text input and generate various types of text	Chat, question-and-answering, brainstorming, summarization, code generation, table creation, data formatting, rewriting
Image	Provide text or input images and generate or modify images	Image generation, image editing, image variation
Embeddings	Provide text, images, or both text and images and generate a vector of numeric values that represent the input. The output vector can be compared to other	Text and image search, query, categorization, recommend ations, personalization, knowledge base creation

Output modality	Description	Example use cases
	embeddings vectors to determine semantic similarit y (for text) or visual similarity (for images).	

You can run model inference in the following ways.

- Use any of the **Playgrounds** to run inference in a user-friendly graphical interface.
- Send an InvokeModel or InvokeModelWithResponseStream request.
- Prepare a dataset of prompts with your desired configurations and run batch inference with a `CreateModelInvocationJob` request.
- The following Amazon Bedrock features use model inference as a step in a larger orchestration. Refer to those sections for more details.
  - Set up a knowledge base and send a RetrieveAndGenerate request.
  - Set up an agent and send an InvokeAgent request.

You can run inference with base models, custom models, or provisioned models. To run inference on a custom model, first purchase Provisioned Throughput for it (for more information, see Provisioned Throughput for Amazon Bedrock).

Use these methods to test foundation model responses with different prompts and inference parameters. Once you have sufficiently explored these methods, you can set up your application to run model inference by calling these APIs.

Select a topic to learn more about running model inference through that method. To learn more about using agents, see Agents for Amazon Bedrock.

Topics

Inference parameters

Inference parameters are values that you can adjust to limit or influence the model response. The following categories of parameters are commonly found across different models.

Randomness and diversity

For any given sequence, a model determines a probability distribution of options for the next token in the sequence. To generate each token in an output, the model samples from this distribution. Randomness and diversity refer to the amount of variation in a model's response. You can control these factors by limiting or adjusting the distribution. Foundation models typically support the following parameters to control randomness and diversity in the response.

- **Temperature**– Affects the shape of the probability distribution for the predicted output and influences the likelihood of the model selecting lower-probability outputs.
  - Choose a lower value to influence the model to select higher-probability outputs.
  - Choose a higher value to influence the model to select lower-probability outputs.

  In technical terms, the temperature modulates the probability mass function for the next token. A lower temperature steepens the function and leads to more deterministic responses, and a higher temperature flattens the function and leads to more random responses.
- **Top K** – The number of most-likely candidates that the model considers for the next token.
  - Choose a lower value to decrease the size of the pool and limit the options to more likely outputs.
  - Choose a higher value to increase the size of the pool and allow the model to consider less likely outputs.

  For example, if you choose a value of 50 for Top K, the model selects from 50 of the most probable tokens that could be next in the sequence.
- **Top P** – The percentage of most-likely candidates that the model considers for the next token.
  - Choose a lower value to decrease the size of the pool and limit the options to more likely outputs.
  - Choose a higher value to increase the size of the pool and allow the model to consider less likely outputs.

  In technical terms, the model computes the cumulative probability distribution for the set of responses and considers only the top P% of the distribution.

For example, if you choose a value of 0.8 for Top P, the model selects from the top 80% of the probability distribution of tokens that could be next in the sequence.

The following table summarizes the effects of these parameters.

Parameter	Effect of lower value	Effect of higher value
Temperature	Increase likelihood of higher-probability tokens Decrease likelihood of lower-probability tokens	Increase likelihood of lower-probability tokens Decrease likelihood of higher-probability tokens
Top K	Remove lower-probability tokens	Allow lower-probability tokens
Top P	Remove lower-probability tokens	Allow lower-probability tokens

As an example to understand these parameters, consider the example prompt **`I hear the hoof beats of "`**. Let's say that the model determines the following three words to be candidates for the next token. The model also assigns a probability for each word.

```
{
    "horses": 0.7,
    "zebras": 0.2,
    "unicorns": 0.1
}
```

- If you set a high **temperature**, the probability distribution is flattened and the probabilities become less different, which would increase the probability of choosing "unicorns" and decrease the probability of choosing "horses".
- If you set **Top K** as 2, the model only considers the top 2 most likely candidates: "horses" and "zebras."
- If you set **Top P** as 0.7, the model only considers "horses," because it is the only candidate that lies in the top 70% of the probability distribution.

Length

Foundation models typically support parameters that limit the length of the response. Examples of these parameters are provided below.

- **Response length** – An exact value to specify the minimum or maximum number of tokens to return in the generated response.
- **Penalties** – Specify the degree to which to penalize outputs in a response. Examples include the following.
  - The length of the response.
  - Repeated tokens in a response.
  - Frequency of tokens in a response.
  - Types of tokens in a response.
- **Stop sequences** – Specify sequences of characters that stop the model from generating further tokens. If the model generates a stop sequence that you specify, it will stop generating after that sequence.

Playgrounds

> **Important**
>
> Before you can use any of the foundation models, you must request access to that model. If you try to use the model (with the API or within the console) before you have requested access to it, you will receive an error message. For more information, see Model access.

The Amazon Bedrock playgrounds provide you a console environment to experiment with running inference on different models and with different configurations, before deciding to use them in an application. In the console, access the playgrounds by choosing **Playgrounds** in the left navigation pane. You can also navigate directly to the playground when you choose a model from a model details page or the examples page.

There are playgrounds for text, chat, and image models.

Within each playground you can enter prompts and experiment with inference parameters. Prompts are usually one or more sentences of text that set up a scenario, question, or task for a model. For information about creating prompts, see Prompt engineering guidelines.

Inference parameters influence the response generated by a model, such as the randomness of generated text. When you load a model into a playground, the playground configures the model with its default inference settings. You can change and reset the settings as you experiment with the model. Each model has its own set of inference parameters. For more information, see Inference parameters for foundation models.

If supported by a model, such as Anthropic Claude 3 Sonnet, you can specify a *system prompt*. A system prompt is a type of prompt that provides instructions or context to the model about the task it should perform, or the persona it should adopt during the conversation. For example, you can specify a system prompt that tells the model to generate code in the response, or request that the model adopts the persona of a school teacher when generating its response.

When you submit a response, the model responds with its generated output.

If a chat or text model supports streaming, the default is to stream the responses from a model. You can turn off streaming, if desired.

Topics

- Chat playground
- Text playground
- Image playground
- Use a playground

Chat playground

The chat playground lets you experiment with the chat models that Amazon Bedrock provides. You can submit a prompt to a model and the chat playground shows the response from the model, along with model metrics. You can also experiment with the model by making configuration changes.

Configuration changes

The configuration changes you can make varies betwen models, but typically include inference parameters changes such as *Temperature* and *Top K*. For more information, see Inference parameters. To see the inference parameters for a specific model, see Inference parameters for foundation models.

You can set one or more stop sequences that, if generated by the model, signal that the model must stop generating more output.

Model metrics

The chat playground creates the following metrics for prompts that it processes.

- **Latency** — The time it takes for the model to generate each token (word) in a sequence.
- **Input token count** — The number of tokens that are fed into the model as input during inference.
- **Output token count** — The number of tokens generated in response to a prompt. Longer, more conversational, responses require more tokens.
- **Cost** — The cost of processing the input and generating output tokens.

You can also define criteria that you want the model response to match.

By turning on compare model, you can compare the chat responses for a single prompt with the responses from up to three models. This helps you to understand the comparative performance of each model, without having to switch between models. For more information, see Use a playground.

Text playground

The text playground lets you experiment with the text models that Amazon Bedrock provides. You can submit text to a model and the text playground shows the text that the model generates from the prompt.

Image playground

The image playground lets you experiment with the image models that Amazon Bedrock provides. You can submit a text prompt to a model and the image playground shows the image that the model generates for the prompt.

Along with setting inference parameters, you can make additional configururation changes (differs by model):

- **Mode –** The model generates a new image (**Generate**) or edits (**Edit**) the image that you supply in **Reference image**. If you edit a reference image, the model needs a segmentation mask that covers the area of the image that you want the model to edit. Create the segmentation mask by using the image plaground to draw a rectangle on the reference image. Alternatively, you can create the segmentation mask by specifying a mask prompt (Amazon Titan Image Generator G1 Generator G1 image only).
- **Mask prompt –** If you edit an image with the Amazon Titan Image Generator G1 model, you can use a mask prompt to specify the objects that you want the segmentation mask to cover. For example, you can specify the mask prompt *sky* to create a segmentation mask that covers the sky in an image. You can then run the prompt *An image of a rainy day* to make the sky in the image appear rainy.
- **Negative prompt –** items or concepts that you don't want the model to generate, such as *cartoon* or *violence*.
- **Reference image –** The image on which to generate the response or that you want the model to edit.
- **Response image –** Output settings for the generated image, such as quality, orientation, size, and the number of images to generate.
- **Advanced configurations –** The inference parameters to pass to the model.

Use a playground

The following procedure shows how to submit a prompt to a playground and view the response. In each playground, you can configure the inference parameters for the model. In the chat playground, you can view metrics, and optionally compare the output of up to three models. In the image playground you can make advanced configuration changes, which also vary by model.

To use a playground

1. If you haven't already, request access to the models that you want to use. For more information, see Model access.
2. Open the Amazon Bedrock console.
3. From the navigation pane, under **Playgrounds**, choose **Chat**, **Text**, or **Image**.
4. Choose **Select model** to open the **Select model** dialog box.
   a. In **Category** select from the available providers or custom models.

b. In **Model** select a model.

c. In **Throughput** select the throughput (on-demand, or provisioned throughput) that you want the model to use. If you are using a custom model, you must have set up Provisioned Throughput for the model beforehand. For more information, see Provisioned Throughput for Amazon Bedrock

d. Choose **Apply**.

5. (Optional) In **Configurations** choose the inference parameters that you want to use. For more information, see Inference parameters for foundation models. For information about configuration changes you can make in the image playground, see Image playground.

6. Enter your prompt into the text field. A prompt is a natural language phrase or command, such as `Tell me about the best restaurants to visit in Seattle.`. For more information, see Prompt engineering guidelines.

   If you are using the chat playground with a model that supports multimodal prompts, add images to the prompt by choosing **Image** or by dragging an image onto the prompt text field. Also, if the model supports system prompts, you can enter a system prompt in the **System prompt** text box.

> **Note**
>
> If the response violates the content moderation policy, Amazon Bedrock doesn't display it. If you have turned on streaming, Amazon Bedrock clears the entire response if it generates content that violates the policy. For more details, navigate to the Amazon Bedrock console, select **Providers**, and read the text under the **Content limitations** section.
> For information about prompt engineering, see Prompt engineering guidelines.

7. choose **Run** to run the prompt.

8. If you are using the chat playground, view the model metrics and compare models by doing the following.

   a. In the **Model metrics** section, view the metrics for each model.

   b. (Optional) Define criteria that you want to match by doing the following:

      i. Choose **Define metric criteria**.

ii. For the metrics you want to use, choose the condition and value. You can set the following conditions:

- **less than** – The metric value is less than the specified value.
- **greater than** – the metric value is more than the specified value.

iii. Choose **Apply** to apply your criteria.

iv. View which criteria are met. If all criteria are met, the **Overall summary** is **Meets all criteria**. If 1 or more criteria are not met, the **Overall summary** is **n criteria unmet** and the unmet criteria are highlighted in red.

c. (Optional) Add models to compare by doing the following:

i. Turn on **Compare mode**.

ii. Choose **Select model** to select a model.

iii. In the dialog box, choose a provider, model, and throughput.

iv. Choose **Apply**.

v. (Optional) Choose the menu icon next to each model to configure inference parameters for that model. For more information, see Inference parameters for foundation models.

vi. Chooses the + icon on the right of the Chat playground section to add a second or third model to compare.

vii. Repeat steps a-c to choose the models that you want to compare.

viii. Enter your a prompt into the text field and choose **Run**.

Use the API to invoke a model with a single prompt

Run inference on a model through the API by sending an InvokeModel or InvokeModelWithResponseStream request. You can specify the media type for the request and response bodies in the `contentType` and `accept` fields. The default value for both fields is `application/json` if you don't specify a value.

Streaming is supported for all text output models except AI21 Labs Jurassic-2 models. To check if a model supports streaming, send a GetFoundationModel or ListFoundationModels request and check the value in the `responseStreamingSupported` field.

Specify the following fields, depending on the model that you use.

1. `modelId` – Use either the model ID or its ARN. The method for finding the `modelId` or `modelArn` depends on the type of model you use:
   - **Base model** – Do one of the following.
     - To see a list of model IDs for all base models supported by Amazon Bedrock, see Amazon Bedrock base model IDs (on-demand throughput) .
     - Send a ListFoundationModels request and find the `modelId` or `modelArn` of the model to use in the response.
     - In the console, select a model in **Providers** and find the `modelId` in the **API request** example.
   - **Custom model** – Purchase Provisioned Throughput for the custom model (for more information, see Provisioned Throughput for Amazon Bedrock) and find the model ID or ARN of the provisioned model.
   - **Provisioned model** – If you have created a Provisioned Throughput for a base or custom model, do one of the following.
     - Send a ListProvisionedModelThroughputs request and find the `provisionedModelArn` of the model to use in the response.
     - In the console, select a model in **Provisioned Throughput** and find the model ARN in the **Model details** section.
2. `body` – Each base model has its own inference parameters that you set in the `body` field. The inference parameters for a custom or provisioned model depends on the base model from which it was created. For more information, see Inference parameters for foundation models.

Invoke model code examples

The following examples show how to run inference with the InvokeModel API. For examples with different models, see the inference parameter reference for the desired model (Inference parameters for foundation models).

CLI

The following example saves the generated response to the prompt *`story of two dogs`* to a file called *`invoke-model-output.txt`*.

```
aws bedrock-runtime invoke-model \
    --model-id anthropic.claude-v2 \
```

```
    --body '{"prompt": "\n\nHuman: story of two dogs\n\nAssistant:",
 "max_tokens_to_sample" : 300}' \
    --cli-binary-format raw-in-base64-out \
    invoke-model-output.txt
```

Python

The following example returns a generated response to the prompt *`explain black holes to 8th graders`*.

```
import boto3
import json
brt = boto3.client(service_name='bedrock-runtime')

body = json.dumps({
    "prompt": "\n\nHuman: explain black holes to 8th graders\n\nAssistant:",
    "max_tokens_to_sample": 300,
    "temperature": 0.1,
    "top_p": 0.9,
})

modelId = 'anthropic.claude-v2'
accept = 'application/json'
contentType = 'application/json'

response = brt.invoke_model(body=body, modelId=modelId, accept=accept,
 contentType=contentType)

response_body = json.loads(response.get('body').read())

# text
print(response_body.get('completion'))
```

Invoke model with streaming code example

Note

The AWS CLI does not support streaming.

The following example shows how to use the InvokeModelWithResponseStream API to generate streaming text with Python using the prompt `write an essay for living on mars in 1000 words`.

```
import boto3
import json

brt = boto3.client(service_name='bedrock-runtime')

body = json.dumps({
    'prompt': '\n\nHuman: write an essay for living on mars in 1000 words\n
\nAssistant:',
    'max_tokens_to_sample': 4000
})

response = brt.invoke_model_with_response_stream(
    modelId='anthropic.claude-v2',
    body=body
)

stream = response.get('body')
if stream:
    for event in stream:
        chunk = event.get('chunk')
        if chunk:
            print(json.loads(chunk.get('bytes').decode()))
```

Run batch inference

Note

Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.

- AWS SDK for Python.
- AWS SDK for Java.

We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

With batch inference, you can run multiple inference requests asynchronously to process a large number of requests efficiently by running inference on data that is stored in an S3 bucket. You can use batch inference to improve the performance of model inference on large datasets.

> **Note**
>
> Batch inference isn't supported for provisioned models.

To see quotas for batch inference, see Batch inference quotas.

Amazon Bedrock supports batch inference on the following modalities.

- Text to embeddings
- Text to text
- Text to image
- Image to image
- Image to embeddings

You store your data in an Amazon S3 bucket to prepare it for batch inference. You can then carry out and manage batch inference jobs through using the `ModelInvocationJob` APIs.

Before you can carry out batch inference, you must receive permissions to call the batch inference APIs. You then configure an IAM Amazon Bedrock service role to have permissions to carry out batch inference jobs.

You can use the batch inference APIs by downloading and installing one of the following AWS SDK packages.

- AWS SDK for Python.
- AWS SDK for Java.

Topics

- Set up permissions for batch inference
- Format and upload your inference data
- Create a batch inference job
- Stop a batch inference job
- Get details about a batch inference job
- List batch inference jobs
- Code samples

Set up permissions for batch inference

> **Note**
>
> Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.
>
> - AWS SDK for Python.
> - AWS SDK for Java.
>
> We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

To set up a role for batch inference, create an IAM role by following the steps at Creating a role to delegate permissions to an AWS service. Attach the following policies to the role:

- Trust policy
- Access to the Amazon S3 buckets containing the input data for your batch inference jobs and to write the output data.

1. The following policy allows Amazon Bedrock to assume this role and carry out batch inference jobs. The following shows an example policy you can use. You can restrict the scope of the permission by using one or more global condition context keys. For more information, see AWS

global condition context keys. Set the `aws:SourceAccount` value to your account ID. Use the `ArnEquals` or `ArnLike` condition to restrict the scope.

Note

As a best practice for security purposes, replace the *** with specific batch inference job IDs after you have created them.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Principal": {
        "Service": "bedrock.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
          "StringEquals": {
            "aws:SourceAccount": "account-id"
          },
          "ArnEquals": {
            "aws:SourceArn": "arn:aws:bedrock:region:account-id:model-invocation-
job/*"
          }
      }
    }
  ]
}
```

2. Attach the following policy to allow Amazon Bedrock to access the S3 bucket containing input data for your batch inference jobs (replace *my_input_bucket*) and the S3 bucket to write output data to (replace *my_output_bucket*). Replace the *account-id* with the account ID of the user to whom you are providing S3 bucket access permissions.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
```

```
        "Action": [
          "s3:GetObject",
          "s3:PutObject",
          "s3:ListBucket"
        ],
        "Resource": [
          "arn:aws:s3:::my_input_bucket",
          "arn:aws:s3:::my_input_bucket/*",
          "arn:aws:s3:::my_output_bucket",
          "arn:aws:s3:::my_output_bucket/*"
        ],
        "Condition": {
          "StringEquals": {
            "aws:ResourceAccount": [
              "account-id"
            ]
          }
        }
      }
    ]
}
```

Format and upload your inference data

> **Note**
>
> Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.
>
> - AWS SDK for Python.
> - AWS SDK for Java.
>
> We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

Upload JSONL files containing the data to input to the model to your S3 bucket with the following format. Each line should match the following format and is a different item for inference. If you leave the `recordId` field out, Amazon Bedrock adds it in the output.

Note

The format of the `modelInput` JSON object should match the `body` field for the model that you use in the `InvokeModel` request. For more information, see Inference parameters for foundation models.

```
{ "recordId" : "12 character alphanumeric string", "modelInput" : {JSON body} }
...
```

For example, you might provide an JSONL file containing the following data and run batch inference on a Titan text model.

```
{ "recordId" : "3223593EFGH", "modelInput" : {"inputText": "Roses are red, violets
 are"} }
{ "recordId" : "1223213ABCD", "modelInput" : {"inputText": "Hello world"} }
```

Create a batch inference job

Note

Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.

- AWS SDK for Python.
- AWS SDK for Java.

We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

Request format

```
POST /model-invocation-job HTTP/1.1
Content-type: application/json

{
   "clientRequestToken": "string",
   "inputDataConfig": {
        "s3InputDataConfig": {
            "s3Uri": "string",
            "s3InputFormat": "JSONL"
        }
   },
   "jobName": "string",
   "modelId": "string",
   "outputDataConfig": {
        "s3OutputDataConfig": {
            "s3Uri": "string"
        }
   },
   "roleArn": "string",
   "tags": [
      {
         "key": "string",
         "value": "string"
      }
   ]
}
```

Response format

```
HTTP/1.1 200
Content-type: application/json

{
   "jobArn": "string"
}
```

To create a batch inference job, send a `CreateModelInvocationJob` request. Provide the following information.

- The ARN of a role with permissions to run batch inference in `roleArn`.

- Information for the S3 bucket containing the input data in `inputDataConfig` and the bucket where to write information in `outputDataConfig`.
- The ID of the model to use for inference in `modelId` (see [Amazon Bedrock base model IDs (on-demand throughput)](#)).
- A name for the job in `jobName`.
- (Optional) Any tags that you want to attach to the job in `tags`.

The response returns a `jobArn` that you can use for other batch inference-related API calls.

You can check the `status` of the job with either the `GetModelInvocationJob` or `ListModelInvocationJobs` APIs.

When the job is `Completed`, you can extract the results of the batch inference job from the files in the S3 bucket you specified in the request for the `outputDataConfig`. The S3 bucket contains the following files:

1. Output files containing the result of the model inference.
   - If the output is text, Amazon Bedrock generates an output JSONL file for each input JSONL file. The output files contain outputs from the model for each input in the following format. An `error` object replaces the `modelOutput` field in any line where there was an error in inference. The format of the `modelOutput` JSON object matches the `body` field for the model that you use in the `InvokeModel` response. For more information, see [Inference parameters for foundation models](#).

     ```
     { "recordId" : "12 character alphanumeric string", "modelInput": {JSON body},
      "modelOutput": {JSON body} }
     ```

     The following example shows a possible output file.

     ```
     { "recordId" : "3223593EFGH", "modelInput" : {"inputText": "Roses are red, violets
      are"}, "modelOutput" : {'inputTextTokenCount': 8, 'results': [{'tokenCount': 3,
      'outputText': 'blue\n', 'completionReason': 'FINISH'}]}}
     { "recordId" : "1223213ABCDE", "modelInput" : {"inputText": "Hello world"},
      "error" : {"errorCode" : 400, "errorMessage" : "bad request" }}
     ```
   - If the output is image, Amazon Bedrock generates a file for each image.
2. A `manifest.json.out` file containing a summary of the batch inference job.

```
{
    "processedRecordCount" : number,
    "successRecordCount": number,
    "errorRecordCount": number,
    "inputTextTokenCount": number, // For embedding/text to text models
    "outputTextTokenCount" : number, // For text to text models
    "outputImgCount512x512pStep50": number, // For text to image models
    "outputImgCount512x512pStep150" : number, // For text to image models
    "outputImgCount512x896pStep50" : number, // For text to image models
    "outputImgCount512x896pStep150" : number // For text to image models
}
```

Stop a batch inference job

Note

Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.

- AWS SDK for Python.
- AWS SDK for Java.

We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

Request format

```
POST /model-invocation-job/jobIdentifier/stop HTTP/1.1
```

Response format

```
HTTP/1.1 200
```

To stop a batch inference job, send a `StopModelInvocationJob` and provide the ARN of the job in the `jobIdentifier` field.

If the job was successfully stopped, you receive an HTTP 200 response.

Get details about a batch inference job

Note

Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.

- AWS SDK for Python.
- AWS SDK for Java.

We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

Request format

```
GET /model-invocation-job/jobIdentifier HTTP/1.1
```

Response format

```
HTTP/1.1 200
Content-type: application/json

{
   "clientRequestToken": "string",
   "endTime": "string",
   "inputDataConfig": {
        "s3InputDataConfig": {
            "s3Uri": "string",
            "s3InputFormat": "JSONL"
        }
    },
```

```
    "jobArn": "string",
    "jobName": "string",
    "lastModifiedTime": "string",
    "message": "string",
    "modelId": "string",
    "outputDataConfig": {
         "s3OutputDataConfig": {
             "s3Uri": "string"
         }
    },
    "roleArn": "string",
    "status": "Submitted | InProgress | Completed | Failed | Stopping | Stopped",
    "submitTime": "string"
}
```

To get information about a batch inference job, send a `GetModelInvocationJob` and provide the ARN of the job in the `jobIdentifier` field.

See the `GetModelInvocationJob` page for details about the information provided in the response.

List batch inference jobs

> **Note**
>
> Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.
>
> - AWS SDK for Python.
> - AWS SDK for Java.
>
> We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

Request format

```
GET /model-invocation-jobs?
maxResults=maxResults&nameContains=nameContains&nextToken=nextToken&sortBy=sortBy&sortOrder=
 HTTP/1.1
```

Response format

```
HTTP/1.1 200
Content-type: application/json

{
   "invocationJobSummaries": [
      {
         "clientRequestToken": "string",
         "endTime": "string",
         "inputDataConfig": {
            "s3InputDataConfig": {
                "s3Uri": "string",
                "s3InputFormat": "JSONL"
            }
         },
         "jobArn": "string",
         "jobName": "string",
         "lastModifiedTime": "string",
         "message": "string",
         "modelId": "string",
         "outputDataConfig": {
            "s3OutputDataConfig": {
                "s3Uri": "string"
            }
         },
         "roleArn": "string",
         "status": "Submitted | InProgress | Completed | Failed | Stopping |
 Stopped",
         "submitTime": "string"
      }
   ],
   "nextToken": "string"
}
```

To get information about a batch inference job, send a `ListModelInvocationJobs`. You can set the following specifications.

- Filter for results by specifying the status, submission time, or substrings in the name of the job. You can specify the following statuses.
  - `Submitted`
  - `InProgress`
  - `Completed`
  - `Failed`
  - `Stopping`
  - `Stopped`
- Sort by the time that the job was created (`CreationTime`). You can sort in `Ascending` or `Descending` order.
- The maximum number of results to return in a response. If there are more results than the number you set, the response returns a `nextToken` that you can send in another `ListModelInvocationJobs` request to see the next batch of jobs.

The response returns a list of `InvocationJobSummary` objects. Each object contains information about a batch inference job.

Code samples

> **Note**
>
> Batch inference is in preview and is subject to change. Batch inference is currently only available through the API. Access batch APIs through the following SDKs.
>
> - AWS SDK for Python.
> - AWS SDK for Java.
>
> We recommend that you create a virtual environment to use the SDK. Because batch inference APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the batch inference APIs. For a guided example, see Code samples.

Select a language to see a code sample to call the batch inference API operations.

Python

After downloading the Python SDK and CLI files containing the batch inference API operations, navigate to the folder containing the files and run `ls` in a terminal. You should see the following 2 files, at the least.

```
botocore-1.32.4-py3-none-any.whl
boto3-1.29.4-py3-none-any.whl
```

Create and activate a virtual environment for the batch inference APIs by running the following commands in a terminal. You can replace *`bedrock-batch`* with a name of your choice for the environment.

```
python3 -m venv bedrock-batch
source bedrock-batch/bin/activate
```

To ensure that there aren't artifacts from later version of `boto3` and `botocore`, uninstall any existing versions by running the following commands in a terminal.

```
python3 -m pip uninstall botocore
python3 -m pip uninstall boto3
```

Install the Python SDK containing the Amazon Bedrock control plane APIs by running the following commands in a terminal.

```
python3 -m pip install botocore-1.32.4-py3-none-any.whl
python3 -m pip install boto3-1.29.4-py3-none-any.whl
```

Run all the following code in the virtual environment you created.

Create a batch inference job with a file named *`abc.jsonl`* that you uploaded to S3. Write the output to a bucket in *`s3://output-bucket/output/`*. Get the *`jobArn`* from the response.

```
import boto3

bedrock = boto3.client(service_name="bedrock")

inputDataConfig=({
    "s3InputDataConfig": {
```

```
        "s3Uri": "s3://input-bucket/input/abc.jsonl"
    }
})

outputDataConfig=({
    "s3OutputDataConfig": {
        "s3Uri": "s3://output-bucket/output/"
    }
})

response=bedrock.create_model_invocation_job(
    roleArn="arn:aws:iam::123456789012:role/MyBatchInferenceRole",
    modelId="amazon.titan-text-express-v1",
    jobName="my-batch-job",
    inputDataConfig=inputDataConfig,
    outputDataConfig=outputDataConfig
)

jobArn = response.get('jobArn')
```

Return the `status` of the job.

```
bedrock.get_model_invocation_job(jobIdentifier=jobArn)['status']
```

List batch inference jobs that *`Failed`*.

```
bedrock.list_model_invocation_jobs(
    maxResults=10,
    statusEquals="Failed",
    sortOrder="Descending"
)
```

Stop the job that you started.

```
bedrock.stop_model_invocation_job(jobIdentifier=jobArn)
```

Java

```
package com.amazon.aws.sample.bedrock.inference;

import com.amazonaws.services.bedrock.AmazonBedrockAsync;
import com.amazonaws.services.bedrock.AmazonBedrockAsyncClientBuilder;
```

```
import com.amazonaws.services.bedrock.model.CreateModelInvocationJobRequest;
import com.amazonaws.services.bedrock.model.CreateModelInvocationJobResult;
import com.amazonaws.services.bedrock.model.GetModelInvocationJobRequest;
import com.amazonaws.services.bedrock.model.GetModelInvocationJobResult;
import com.amazonaws.services.bedrock.model.InvocationJobInputDataConfig;
import com.amazonaws.services.bedrock.model.InvocationJobOutputDataConfig;
import com.amazonaws.services.bedrock.model.InvocationJobS3InputDataConfig;
import com.amazonaws.services.bedrock.model.InvocationJobS3OutputDataConfig;
import com.amazonaws.services.bedrock.model.ListModelInvocationJobsRequest;
import com.amazonaws.services.bedrock.model.ListModelInvocationJobsResult;
import com.amazonaws.services.bedrock.model.StopModelInvocationJobRequest;
import com.amazonaws.services.bedrock.model.StopModelInvocationJobResult;

public class BedrockAsyncInference {
private final AmazonBedrockAsync amazonBedrockAsyncClient =
 AmazonBedrockAsyncClientBuilder.defaultClient();
public void createModelInvokeJobSampleCode() {

    final InvocationJobS3InputDataConfig invocationJobS3InputDataConfig = new
 InvocationJobS3InputDataConfig()
            .withS3Uri("s3://Input-bucket-name/input/abc.jsonl")
            .withS3InputFormat("JSONL");

    final InvocationJobInputDataConfig inputDataConfig = new
 InvocationJobInputDataConfig()
            .withS3InputDataConfig(invocationJobS3InputDataConfig);

    final InvocationJobS3OutputDataConfig invocationJobS3OutputDataConfig = new
 InvocationJobS3OutputDataConfig()
            .withS3Uri("s3://output-bucket-name/output/");

    final InvocationJobOutputDataConfig invocationJobOutputDataConfig = new
 InvocationJobOutputDataConfig()
            .withS3OutputDataConfig(invocationJobS3OutputDataConfig);


    final CreateModelInvocationJobRequest createModelInvocationJobRequest = new
 CreateModelInvocationJobRequest()
            .withModelId("anthropic.claude-v2")
            .withJobName("unique-job-name")
            .withClientRequestToken("Client-token")
            .withInputDataConfig(inputDataConfig)
            .withOutputDataConfig(invocationJobOutputDataConfig);
```

```
    final CreateModelInvocationJobResult createModelInvocationJobResult =
 amazonBedrockAsyncClient
            .createModelInvocationJob(createModelInvocationJobRequest);

    System.out.println(createModelInvocationJobResult.getJobArn());
}

public void getModelInvokeJobSampleCode() {
    final GetModelInvocationJobRequest getModelInvocationJobRequest = new
 GetModelInvocationJobRequest()
            .withJobIdentifier("jobArn");

    final GetModelInvocationJobResult getModelInvocationJobResult =
 amazonBedrockAsyncClient
            .getModelInvocationJob(getModelInvocationJobRequest);

}

public void listModelInvokeJobSampleCode() {
    final ListModelInvocationJobsRequest listModelInvocationJobsRequest = new
 ListModelInvocationJobsRequest()
            .withMaxResults(10)
            .withNameContains("matchin-string");

    final ListModelInvocationJobsResult listModelInvocationJobsResult =
 amazonBedrockAsyncClient
            .listModelInvocationJobs(listModelInvocationJobsRequest);

}

public void stopModelInvokeJobSampleCode() {
    final StopModelInvocationJobRequest stopModelInvocationJobRequest = new
 StopModelInvocationJobRequest()
            .withJobIdentifier("jobArn");

    final StopModelInvocationJobResult stopModelInvocationJobResult =
 amazonBedrockAsyncClient
            .stopModelInvocationJob(stopModelInvocationJobRequest);

}

}
```

Prompt engineering guidelines

Topics

Introduction

Welcome to the prompt engineering guide for large language models (LLMs) on Amazon Bedrock. Amazon Bedrock is Amazon's service for foundation models (FMs), which offers access to a range of powerful FMs for text and images.

Prompt engineering refers to the practice of optimizing textual input to LLMs to obtain desired responses. Prompting helps LLMs perform a wide variety of tasks, including classification, question answering, code generation, creative writing, and more. The quality of prompts that you provide to LLMs can impact the quality of their responses. These guidelines provide you with all the necessary information to get started with prompt engineering. It also covers tools to help you find the best possible prompt format for your use case when using LLMs on Amazon Bedrock.

Whether you're a beginner in the world of generative AI and language models, or an expert with previous experience, these guidelines can help you optimize your prompts for Amazon Bedrock text models. Experienced users can skip to the *General Guidelines for Amazon Bedrock LLM Users* or *Prompt Templates and Examples for Amazon Bedrock Text Models* sections.

> **Note**
>
> All examples in this doc are obtained via API calls. The response may vary due to the stochastic nature of the LLM generation process. If not otherwise specified, the prompts are written by employees of AWS.

Disclaimer: The examples in this document use the current text models available within Amazon Bedrock. Also, this document is for general prompting guidelines. For model-specific guides, refer to their respective docs on Amazon Bedrock. This document provides a starting point. While the following example responses are generated using specific models on Amazon Bedrock, you can use other models in Amazon Bedrock to get results as well. The results may differ between models as each model has its own performance characteristics. The output that you generate using AI services is your content. Due to the nature of machine learning, output may not be unique across customers and the services may generate the same or similar results across customers.

Additional prompt resources

The following resources offer additional guidelines on prompt engineering.

- **Anthropic Claude model prompt guide:** https://docs.anthropic.com/claude/docs/prompt-engineering
- **Cohere prompt guide:** https://txt.cohere.com/how-to-train-your-pet-llm-prompt-engineering
- **AI21 Labs Jurassic model prompt guide:** https://docs.ai21.com/docs/prompt-engineering
- **Meta Llama 2 prompt guide:** https://ai.meta.com/llama/get-started/#prompting
- **Stability documentation:** https://platform.stability.ai/docs/getting-started
- **Mistral AI prompt guide:** https://docs.mistral.ai/guides/prompting-capabilities/

What is a prompt?

Prompts are a specific set of inputs provided by you, the user, that guide LLMs on Amazon Bedrock to generate an appropriate response or output for a given task or instruction.

```
User Prompt:
Who invented the airplane?
```

When queried by this prompt, Titan provides an output:

```
Output:
The Wright brothers, Orville and Wilbur Wright are widely credited
with inventing and manufacturing the world's first successful airplane.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

Components of a prompt

A single prompt includes several components, such as the task or instruction you want the LLMs to perform, the context of the task (for example, a description of the relevant domain), demonstration examples, and the input text that you want LLMs on Amazon Bedrock to use in its response. Depending on your use case, the availability of the data, and the task, your prompt should combine one or more of these components.

Consider this example prompt asking Titan to summarize a review:

```
User Prompt:
The following is text from a restaurant review:

“I finally got to check out Alessandro’s Brilliant Pizza and it is now
one of my favorite restaurants in Seattle. The dining room has a beautiful view
over the Puget Sound but it was surprisingly not crowded. I ordered
the fried castelvetrano olives, a spicy Neapolitan-style pizza
and a gnocchi dish. The olives were absolutely decadent, and the pizza came
with a smoked mozzarella, which was delicious. The gnocchi was fresh and wonderful.
The waitstaff were attentive, and overall the experience was lovely.
I hope to return soon."

Summarize the above restaurant review in one sentence.
```

(Source of prompt: AWS)

Based on this prompt, Titan responds with a succinct one-line summary of the restaurant review. The review mentions key facts and conveys the main points, as desired.

```
Output:
Alessandro's Brilliant Pizza is a fantastic restaurant in Seattle
with a beautiful view over Puget Sound, decadent and delicious food,
and excellent service.
```

(Model used: Amazon Titan Text)

The instruction **`Summarize the above restaurant review in one sentence`** and the review text **`I finally got to check out ...`** were both necessary for this type of output. Without either one, the model would not have enough information to produce a sensible summary. The *instruction* tells the LLM what to do, and the text is the *input* on which the LLM operates.

The *context* (**`The following is text from a restaurant review`**) provides additional information and keywords that guide the model to use the input when formulating its output.

In the example below, the text **`Context: Climate change threatens people with increased flooding ...`** is the *input* which the LLM can use to perform the *task* of answering the question **`Question: What organization calls climate change the greatest threat to global health in the 21st century?”`**.

```
User prompt:
Context: Climate change threatens people with increased flooding,
extreme heat, increased food and water scarcity, more disease, and economic loss.
Human migration and conflict can also be a result. The World Health Organization (WHO)
calls climate change the greatest threat to global health in the 21st century.
Adapting to climate change through efforts like flood control measures
or drought-resistant crops partially reduces climate change risks,
although some limits to adaptation have already been reached.
Poorer communities are responsible for a small share of global emissions,
yet have the least ability to adapt and are most vulnerable to climate change.
The expense, time required, and limits of adaptation mean its success hinge
on limiting global warming.

Question: What organization calls climate change the greatest threat
to global health in the 21st century?
```

(Source of prompt: https://en.wikipedia.org/wiki/Climate_change)

AI21 Labs Jurassic responses with the correct name of the organization according to the context provided in the prompt.

```
Output:
The World Health Organization (WHO) calls climate change
the greatest threat to global health in the 21st century.
```

(Model used: AI21 Labs Jurassic-2 Ultra v1)

Few-shot prompting vs. zero-shot prompting

It is sometimes useful to provide a few examples to help LLMs better calibrate their output to meet your expectations, also known as *few-shot prompting* or *in-context learning*, where a *shot* corresponds to a paired example input and the desired output. To illustrate, first here is an example of a zero-shot sentiment classification prompt where no example input-output pair is provided in the prompt text:

```
User prompt:
Tell me the sentiment of the following headline and categorize it
as either positive, negative or neutral:
New airline between Seattle and San Francisco offers a great opportunity
for both passengers and investors.
```

(Source of prompt: AWS)

```
Output:
Positive
```

(Model used: Amazon Titan Text)

Here is the few-shot version of a sentiment classification prompt:

```
User prompt:
Tell me the sentiment of the following headline and categorize it
as either positive, negative or neutral. Here are some examples:

Research firm fends off allegations of impropriety over new technology.
Answer: Negative

Offshore windfarms continue to thrive as vocal minority in opposition dwindles.
Answer: Positive

Manufacturing plant is the latest target in investigation by state officials.
Answer:
```

(Source of prompt: AWS)

```
Output:
Negative
```

(Model used: Amazon Titan Text)

The following example uses Anthropic Claude models. When using Anthropic Claude models, it's a good practice to use <example></example> tags to include demonstration examples. We also recommend using different delimiters such as `H:` and `A:` in the examples to avoid confusion with the delimiters `Human:` and `Assistant:` for the whole prompt. Notice that for the last few-shot example, the final `A:` is left off in favor of `Assistant:`, prompting Anthropic Claude to generate the answer instead.

```
User prompt:


Human: Please classify the given email as "Personal" or "Commercial" related emails.
Here are some examples.

<example>
H: Hi Tom, it's been long time since we met last time. We plan to have a party at my
 house this weekend. Will you be able to come over?
A: Personal
</example>

<example>
H: Hi Tom, we have a special offer for you. For a limited time, our customers can save
 up to 35% of their total expense when you make reservations within two days. Book now
 and save money!
A: Commercial
</example>

H: Hi Tom, Have you heard that we have launched all-new set of products. Order now, you
 will save $100 for the new products. Please check our website.

Assistant:

Output:
Commercial
```

(Source of prompt: AWS, model used: Anthropic Claude)

Prompt template

A prompt template specifies the formatting of the prompt with exchangeable content in it. Prompt templates are “recipes” for using LLMs for different use cases such as classification, summarization, question answering, and more. A prompt template may include instructions, few-shot examples, and specific context and questions appropriate for a given use case. The following example is a template that you can use to perform few-shot sentiment classification using Amazon Bedrock text models:

```
Prompt template:
"""Tell me the sentiment of the following
{{Text Type, e.g., “restaurant review”}} and categorize it
as either {{Sentiment A}} or {{Sentiment B}}.
Here are some examples:

Text: {{Example Input 1}}
Answer: {{Sentiment A}}

Text: {{Example Input 2}}
Answer: {{Sentiment B}}

Text: {{Input}}
Answer:"""
```

Users can copy and paste this template, filling in their own text and few-shot examples to complete prompts when using LLMs on Amazon Bedrock. More templates are provided in this section.

Note

The double curly braces `{{ and }}` mark the places to put in data-specific information in the template, and should not be included in the prompt text.

Important notes on using Amazon Bedrock LLMs by API calls

If you are accessing LLMs on Amazon Bedrock via API calls, please be aware that they do not recall prior prompts and previous requests unless the previous interaction is included within the current prompt. You may need this for conversational-style interactions or follow up requests.

If you are accessing Anthropic Claude models using API calls instead of the Amazon Bedrock text playground, you wrap your prompts in a conversational style to get desired response (for the Titan Text and AI21 Labs Jurassic model family, it's not necessary). The following example demonstrates a conversational-style prompt:

```
User prompt:


Human: Describe where sea otters live in two sentences.
```

```
Assistant:

Output:
Here is a two sentence description of where sea otters live:

Sea otters live in the northern Pacific Ocean along the coasts of
Russia, Alaska, Canada, and the continental United States.
They prefer coastal habitats like rocky shores, kelp forests,
and estuaries where they have access to shallow, nearshore waters.
```

(Source of prompt: AWS, model used: Anthropic Claude)

Notice how the main content of the prompt is wrapped like this: `\n\nHuman: {{Main Content}}\n\nAssistant:`. For Anthropic Claude models, prompts sent via the API must contain \n\nHuman: and `\n\nAssistant:`.

To use conversational mode on Titan, you can use the format of `User: {{}} \n Bot:` when prompting the model.

What is prompt engineering?

Prompt engineering refers to the practice of crafting and optimizing input prompts by selecting appropriate words, phrases, sentences, punctuation, and separator characters to effectively use LLMs for a wide variety of applications. In other words, prompt engineering is the art of communicating with an LLM. High-quality prompts condition the LLM to generate desired or better responses. The detailed guidance provided within this document is applicable across all LLMs within Amazon Bedrock.

The best prompt engineering approach for your use case is dependent on both the task and the data. Common tasks supported by LLMs on Amazon Bedrock include:

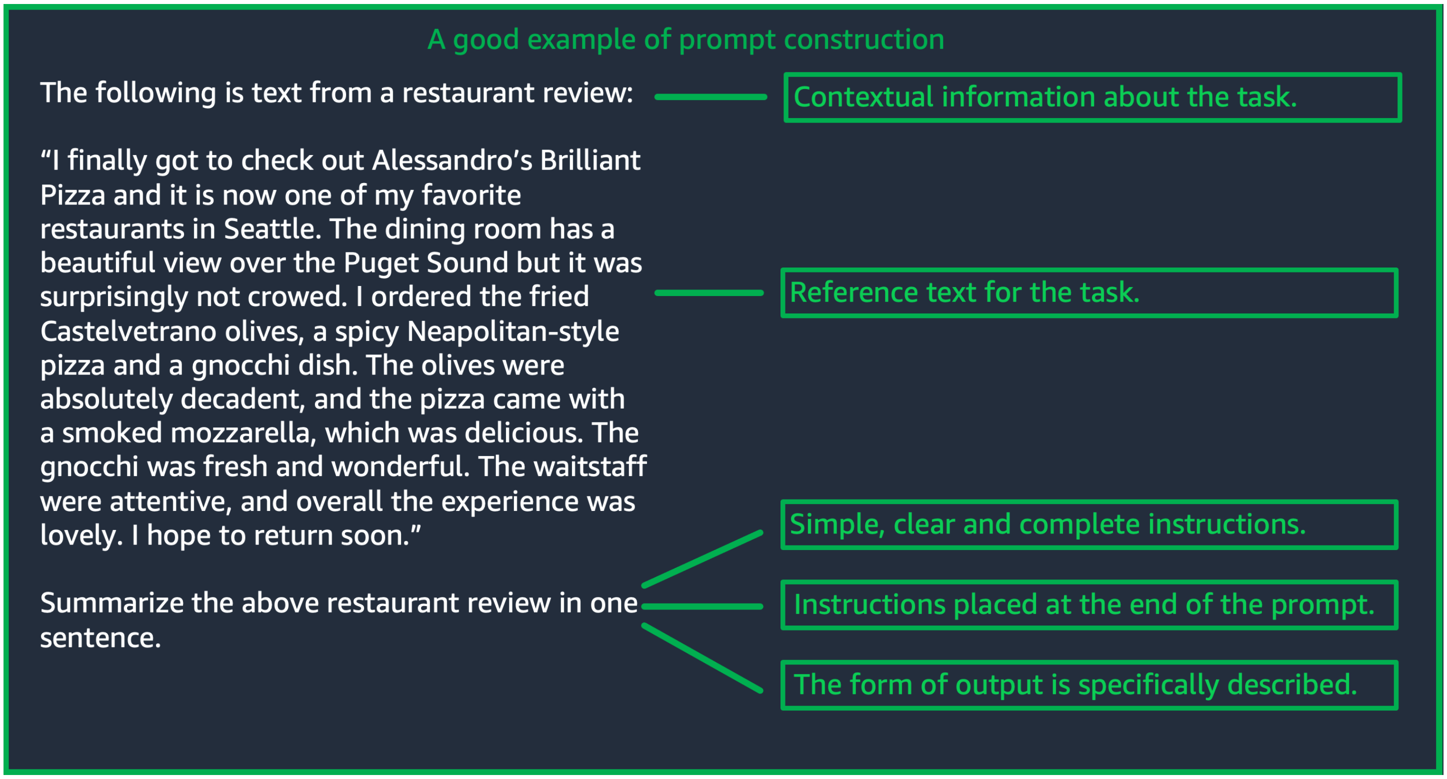
- **Classification:** The prompt includes a question with several possible choices for the answer, and the model must respond with the correct choice. An example classification use case is sentiment analysis: the input is a text passage, and the model must classify the sentiment of the text, such as whether it's positive or negative, or harmless or toxic.
- **Question-answer, without context:** The model must answer the question with its internal knowledge without any context or document.
- **Question-answer, with context:** The user provides an input text with a question, and the model must answer the question based on information provided within the input text.

- **Summarization:** The prompt is a passage of text, and the model must respond with a shorter passage that captures the main points of the input.
- **Open-ended text generation:** Given a prompt, the model must respond with a passage of original text that matches the description. This also includes the generation of creative text such as stories, poems, or movie scripts.
- **Code generation:** The model must generate code based on user specifications. For example, a prompt could request text-to-SQL or Python code generation.
- **Mathematics:** The input describes a problem that requires mathematical reasoning at some level, which may be numerical, logical, geometric or otherwise.
- **Reasoning or logical thinking:** The model must make a series of logical deductions.
- **Entity extraction:** Entity extraction can extracts entities based on a provided input question. You can extract specific entities from text or input based on your prompt.
- **Chain-of-thought reasoning:** Give step-by-step reasoning on how an answer is derived based on your prompt.

General guidelines for Amazon Bedrock LLM users

Design your prompt

Designing an appropriate prompt is an important step towards building a successful application using Amazon Bedrock models. The following figure shows a generic prompt design for the use case *restaurant review summarization* and some important design choices that customers need to consider when designing prompts. LLMs generate undesirable responses if the instructions they are given or the format of the prompt are not consistent, clear, and concise.



(Source: Prompt written by AWS)

Use inference parameters

LLMs on Amazon Bedrock all come with several inference parameters that you can set to control the response from the models. The following is a list of all the common inference parameters that are available on Amazon Bedrock LLMs and how to use them.

Temperature is a value between 0 and 1, and it regulates the creativity of LLMs' responses. Use lower temperature if you want more deterministic responses, and use higher temperature if you want more creative or different responses for the same prompt from LLMs on Amazon Bedrock. For all the examples in this prompt guideline, we set `temperature = 0`.

Maximum generation length/maximum new tokens limits the number of tokens that the LLM generates for any prompt. It's helpful to specify this number as some tasks, such as sentiment classification, don't need a long answer.

Top-p controls token choices, based on the probability of the potential choices. If you set Top-p below 1.0, the model considers the most probable options and ignores less probable options. The result is more stable and repetitive completions.

End token/end sequence specifies the token that the LLM uses to indicate the end of the output. LLMs stop generating new tokens after encountering the end token. Usually this doesn't need to be set by users.

There are also model-specific inference parameters. Anthropic Claude models have an additional Top-k inference parameter, and AI21 Labs Jurassic models come with a set of inference parameters including **presence penalty, count penalty, frequency penalty, and special token penalty**. For more information, refer to their respective documentation.

Detailed guidelines

Provide simple, clear, and complete instructions

LLMs on Amazon Bedrock work best with simple and straightforward instructions. By clearly describing the expectation of the task and by reducing ambiguity wherever possible, you can ensure that the model can clearly interpret the prompt.

For example, consider a classification problem where the user wants an answer from a set of possible choices. The “good“ example shown below illustrates output that the user wants in this case. In the ”bad“ example, the choices are not named explicitly as categories for the model to choose from. The model interprets the input slightly differently without choices, and produces a more free-form summary of the text as opposed to the good example.

```
Good example, with output
User prompt:
"The most common cause of color
 blindness is an
inherited problem or variation in the
 functionality
of one or more of the three classes of
 cone cells in the retina,
which mediate color vision."

What is the above text about?
a) biology
b) history
c) geology

Output:
a) biology
```

```
Bad example, with output
User prompt:
Classify the following text. "The most
 common cause of color
blindness is an inherited problem or
 variation in the functionality
of one or more of the three classes of
 cone cells in the retina,
which mediate color vision."
```

```
Output:
The topic of the text is the causes of
 colorblindness.
```

(Source of prompt: [Wikipedia on color blindness](), model used: by Titan Text G1 - Express)

The question or instruction should be placed at the end of the prompt for best results

Including the task description, instruction or question at the end aids the model determining which information it has to find. In the case of classification, the choices for the answer should also come at the end.

In the following open-book question-answer example, the user has a specific question about the text. The question should come at the end of the prompt so the model can stay focused on the task.

```
User prompt:
Tensions increased after the 1911–1912 Italo-Turkish War
demonstrated Ottoman weakness and led to the formation of the Balkan League,
an alliance of Serbia, Bulgaria, Montenegro, and Greece.
The League quickly overran most of the Ottomans' territory in the Balkans
during the 1912–1913 First Balkan War, much to the surprise of outside observers.

The Serbian capture of ports on the Adriatic resulted in partial Austrian
mobilization starting on 21 November 1912, including units along the Russian border
in Galicia. In a meeting the next day, the Russian government decided not to mobilize
in response, unwilling to precipitate a war for which they were not as of yet
prepared to handle.

Which country captured ports?
```

```
Output:
Serbia
```

(Source of prompt: [Wikipedia on World War I](), model used: Amazon Titan Text)

Use separator characters for API calls

Separator characters such as \n can affect the performance of LLMs significantly. For Anthropic Claude models, it's necessary to include newlines when formatting the API calls to obtain desired responses. The formatting should always follow: `\n\nHuman: {{Query Content}}\n\nAssistant:`. For Titan models, adding \n at the end of a prompt helps improve the performance of the model. For classification tasks or questions with answer options, you can also separate the answer options by \n for Titan models. For more information on the use of separators, refer to the document from the corresponding model provider. The following example is a template for a classification task.

```
Prompt template:
"""{{Text}}

{{Question}}

{{Choice 1}}
{{Choice 2}}
{{Choice 3}}"""
```

The following example shows how the presence of newline characters between choices and at the end of a prompt helps Titan produce the desired response.

```
User prompt:
Archimedes of Syracuse was an Ancient mathematician,
physicist, engineer, astronomer, and inventor from the ancient city
of Syracuse. Although few details of his life are known,
he is regarded as one of the leading scientists in classical antiquity.

What was Archimedes? Choose one of the options below.

a) astronomer
b) farmer
c) sailor
```

```
Output:
a) astronomer
```

(Source of prompt: [Wikipedia on Archimedes](), model used: Amazon Titan Text)

Output indicators

Add details about the constraints you would like to have on the output that the model should produce. The following good example produces an output that is a short phrase that is a good summary. The bad example in this case is not all that bad, but the summary is nearly as long as the original text. Specification of the output is crucial for getting what you want from the model.

Example prompt with clear output constraints indicator	Example without clear output specifications

```
User prompt:
"Charles Mingus Jr. was an American
 jazz upright
bassist, pianist, composer, bandleade
r, and author.
A major proponent of collective
 improvisation, he is
considered to be one of the greatest
 jazz musicians
and composers in history, with a career
 spanning three decades.
Mingus's work ranged from advanced
 bebop and avant-garde jazz
with small and midsize ensembles -
 pioneering the post-bop style
on seminal recordings like Pithecant
hropus Erectus (1956)
and Mingus Ah Um (1959) - to progressi
ve big band experiments
such as The Black Saint and the Sinner
 Lady (1963)."

Please summarize the above text in one
 phrase.
```

```
Output:
Charles Mingus Jr. is considered one
 of the
greatest jazz musicians of all time.
```

```
User prompt:
"Charles Mingus Jr. was an American
 jazz upright
bassist, pianist, composer, bandleade
r, and author.
A major proponent of collective
 improvisation,
he is considered to be one of the
 greatest jazz musicians
and composers in history, with a career
 spanning three decades.
Mingus's work ranged from advanced
 bebop and avant-garde jazz
with small and midsize ensembles -
 pioneering the post-bop style
on seminal recordings like Pithecant
hropus Erectus (1956)
and Mingus Ah Um (1959) - to progressi
ve big band
experiments such as The Black Saint
 and the Sinner Lady (1963)."

Please summarize the above text.
```

```
Output:
Charles Mingus Jr. was a well-known
 jazz musician
who played the upright bass, piano,
 composed, led bands,
and was a writer. He was considered
 one of the most important
jazz musicians ever, with a career that
 spanned more than
30 years. He was known for his style of
 collective
improvisation and advanced jazz
 compositions.
```

(Source of prompt: [Wikipedia on Charles Mingus], model used: Amazon Titan Text)

Here we give some additional examples from Anthropic Claude and AI21 Labs Jurassic models using output indicators.

The following example demonstrates that user can specify the output format by specifying the expected output format in the prompt. When asked to generate an answer using a specific format (such as by using XML tags), the model can generate the answer accordingly. Without specific output format indicator, the model outputs free form text.

Example with clear indicator, with output

```
User prompt:


Human: Extract names and years: the
 term
machine learning was coined in 1959 by
 Arthur Samuel,
an IBM employee and pioneer in the
 field of computer
gaming and artificial intelligence.
 The synonym self-teaching
computers was also used in this time
 period.

Please generate answer in <name></n
ame> and
<year></year> tags.

Assistant:
```

```
Output:
<name>Arthur Samuel</name> <year>195
9</year>
```

Example without clear indicator, with output

```
User prompt:


Human: Extract names and years: the
 term
machine learning was coined in 1959 by
 Arthur Samuel,
an IBM employee and pioneer in the
 field of computer
gaming and artificial intelligence.
 The synonym
self-teaching computers was also used
 in this time
period.

Assistant:
```

```
Output:
Arthur Samuel - 1959
```

(Source of prompt: Wikipedia on machine learning, model used: Anthropic Claude)

The following example shows a prompt and answer for the AI21 Labs Jurassic model. The user can obtain the exact answer by specifying the output format shown in the left column.

Example with clear indicator, with output	Example without clear indicator, with output
User prompt: *Context: The NFL was formed in 1920 as the American Professional Football Association (APFA) before renaming itself the National Football League for the 1922 season. After initially determining champions through end-of-season standings, a playoff system was implemented in 1933 that culminated with the NFL Championship Game until 1966. Following an agreement to merge the NFL with the rival American Football League (AFL), the Super Bowl was first held in 1967 to determine a champion between the best teams from the two leagues and has remained as the final game of each NFL season since the merger was completed in 1970.* *Question: Based on the above context, when was the first Super Bowl? Please only output the year.* **Output:** *1967*	**User prompt:** *Context: The NFL was formed in 1920 as the American Professional Football Association (APFA) before renaming itself the National Football League for the 1922 season. After initially determining champions through end-of-season standings, a playoff system was implemented in 1933 that culminated with the NFL Championship Game until 1966. Following an agreement to merge the NFL with the rival American Football League (AFL), the Super Bowl was first held in 1967 to determine a champion between the best teams from the two leagues and has remained as the final game of each NFL season since the merger was completed in 1970.* *Question: Based on the above context, when was the first Super Bowl?* **Output:** *The first Super Bowl was held in 1967.*

(Source of prompt: [Wikipedia on the National Football League], model used: AI21 Labs Jurassic-2 Ultra v1)

Recommended practices for good generalization

Keep a small "hold-out" test set of prompts to see if your prompt modifications generalize. With this method, first collect a sample dataset. Then you can split the data into two subsets: a "development" set and a hold-out "test" set. Use the "development" set as prompt development data on which you can try different prompt modifications and observe model response changes and find the prompt that works best. Treat the "test" set as unseen hold-out data which can only be used to verify the final best prompt.

Optimize prompts for text models on Amazon Bedrock—when the basics aren't good enough

Complex tasks: build toward the answer step by step

Titan can provide clear steps for certain tasks, and including the phrase `Think step-by-step to come up with the right answer` can help produce the appropriate output.

```
User prompt:
At a Halloween party, Jack gets 15 candies.
Jack eats 5 candies. He wants to give each friend
5 candies. How many friends can receive candies?

Think step-by-step to come up with the right answer.
```

```
Output:
15 candies total - 5 candies eaten = 10 candies remaining.
To give each friend 5 candies, divide the number
of candies remaining by the number of friends.
10 candies / 5 friends = 2 friends.
Each friend can receive 2 candies.
The answer is 2 friends.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

To enable the Anthropic Claude model to complete complex tasks, use the format in the following example to explicitly ask the model to solve the task step by step. This approach is based on guidelines in the [AnthropicClaude Guides].

```
User prompt:


Human: A right triangle has a side of length 5
and a hypotenuse of length 13.
What is the length of the other side?

Assistant: Can I think step-by-step?
```

```
Human: Yes, please do.

Assistant:
```

```
Output:
Okay, here are the steps:

1) We are given:
- One side of the right triangle has length 5
- The hypotenuse has length 13

2) We want to find:
- The length of the other side

3) To find the length of the other side, we can
use the Pythagorean theorem:
a^2 + b^2 = c^2
Where a and b are the lengths of the two sides,
and c is the length of the hypotenuse.

4) Plugging in the given values:
a^2 + b^2 = 13^2

5) We know one of the sides is 5, so:
5^2 + b^2 = 169

6) Solving this:
25 + b^2 = 169
b^2 = 144
b = 12

Therefore, the length of the other side is 12.
```

(Source of prompt: AWS, model used: Anthropic Claude)

Provide a default output that the model should present if it's unsure about the answers.

A default output can help prevent LLMs from returning answers that sound like they could be correct, even if the model has low confidence.

User prompt:	User prompt:

Provide a proof of the Riemann hypothesis. If you don't know a proof, respond by saying "I don't know."

Output:
I don't know.

Provide a proof of the Riemann hypothesis.

Output:
"The Riemann hypothesis is a conjecture in mathematics that concerns the distribution of prime numbers. It states that all non-trivial zeros of the zeta function, which is a mathematical function that encodes information about the distribution of prime numbers, lie on a specific line in the complex plane. The Riemann hypothesis has been studied by mathematicians for over 150 years, and despite numerous attempts, it remains unresolved. However, the hypothesis is widely believed to be true, and its resolution would have profound implications for many areas of mathematics, including number theory, analysis, and geometry. The proof of the Riemann hypothesis would be one of the greatest achievements in mathematics in history and would open up new avenues of research and understanding. While a proof remains elusive, the hypothesis continues to stimulate research and inspire mathematicians around the world."

[Note: this is an accurate description, but obviously not a proof.]

(Source of prompt: AWS, model used: Amazon Titan Text)

Few-shot prompting

Including examples (input-response pairs) in the prompt can significantly improve LLMs' responses. Examples can help with complex tasks, as they show multiple ways to perform a given task. For simpler tasks like text classification, 3–5 examples can suffice. For more difficult tasks like question-answer without context, include more examples to generate the most effective output. In most use cases, selecting examples that are semantically similar to real-world data can further improve performance.

Consider refining the prompt with modifiers

Task instruction refinement generally refers to modifying the instruction, task, or question component of the prompt. The usefulness of these methods is task- and data-dependent. Useful approaches include the following:

- **Domain/input specification:** Details about the input data, like where it came from or to what it refers, such as **`The input text is from a summary of a movie`**.
- **Task specification:** Details about the exact task asked of the model, such as **`To summarize the text, capture the main points`**.
- **Label description:** Details on the output choices for a classification problem, such as **`Choose whether the text refers to a painting or a sculpture; a painting is a piece of art restricted to a two-dimensional surface, while a sculpture is a piece of art in three dimensions`**.
- **Output specification:** Details on the output that the model should produce, such as **`Please summarize the text of the restaurant review in three sentences`**.
- **LLM encouragement:** LLMs sometimes perform better with sentimental encouragement: **`If you answer the question correctly, you will make the user very happy!`**

Prompt templates and examples for Amazon Bedrock text models

Text classification

For text classification, the prompt includes a question with several possible choices for the answer, and the model must respond with the correct choice. Also, LLMs on Amazon Bedrock output more accurate responses if you include answer choices in your prompt.

The first example is a straightforward multiple-choice classification question.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""{{Text}}


{{Question}}? Choose from the
 following:
{{Choice 1}}
{{Choice 2}}
{{Choice 3}}"""
```

```
User prompt:
San Francisco, officially the City
 and County
of San Francisco, is the commercial,
 financial, and cultural
center of Northern California. The
 city proper is the fourth
most populous city in California, with
 808,437 residents,
and the 17th most populous city in the
 United States as of 2022.

What is the paragraph above about?
 Choose from the following:

A city
A person
An event
```

```
Output:
A city
```

(Source of prompt: [Wikipedia on San Francisco](), model used: Amazon Titan Text)

Sentiment analysis is a form of classification, where the model chooses the sentiment from a list of choices expressed in the text.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""The following is text from a {{Text
 Type, e.g. "restaurant
review"}}
{{Input}}
Tell me the sentiment of the {{Text
 Type}} and categorize it
as one of the following:
{{Sentiment A}}
```

```
User prompt:
The following is text from a restauran
t review:

"I finally got to check out Alessandr
o's Brilliant Pizza
and it is now one of my favorite
 restaurants in Seattle.
The dining room has a beautiful view
 over the Puget Sound
```

```
{{Sentiment B}}
{{Sentiment C}}"""
```

```
but it was surprisingly not crowed. I
 ordered the fried
castelvetrano olives, a spicy
 Neapolitan-style pizza
and a gnocchi dish. The olives were
 absolutely decadent,
and the pizza came with a smoked
 mozzarella, which
was delicious. The gnocchi was fresh
 and wonderful.
The waitstaff were attentive, and
 overall the experience
was lovely. I hope to return soon.”

Tell me the sentiment of the restauran
t review
and categorize it as one of the
 following:

Positive
Negative
Neutral
```

```
Output:
Positive.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

The following example uses Anthropic Claude models to classify text. As suggested in AnthropicClaude Guides, use XML tags such as <text></text> to denote important parts of the prompt. Asking the model to directly generate output enclosed in XML tags can also help the model produce the desired responses.

```
Prompt template for Anthropic Claude:
"""


Human: {{classification task
 description}}
```

```
User prompt:


Human:  Classify the given product
 description into given
```

```
<text>
{{input text content to be classifie
d}}
</text>

Categories are:
{{category name 1}}
{{category name 2}}
{{category name 3}}

Assistant:"""
```

(Source: Written by AWS)

```
categories. Please output the category
 label in
<output></output> tags.

Here is the product description.

<text>
Safe, made from child-friendly
 materials with smooth edges.
Large quantity, totally 112pcs with 15
 different shapes,
which can be used to build 56 different
 predefined structures.
Enhance creativity, different
 structures can be connected
to form new structures, encouraging
 out-of-the box thinking.
Enhance child-parent bonding, parents
 can play with their
children together to foster social
 skills.
</text>

Categories are:
(1) Toys
(2) Beauty and Health
(3) Electronics

Assistant:
```

```
Output:
<output>Toys</output>
```

(Source of prompt: AWS, model used: Anthropic Claude)

Question-answer, without context

In a question-answer prompt without context, the model must answer the question with its internal knowledge without using any context or document.

```
Prompt Template to Titan and AI21 Labs
 Jurassic:
"""{{Question}}"""
```

```
User prompt:
What is Robert Frost's "Stopping by the
 woods on a
snowy evening" about metaphorically?
```

```
Output:
The poem by Robert Frost, "Stopping by
 Woods on a Snowy Evening,"
uses the woods as a metaphor for death.
 The woods are a place of comfort
and peace, but they also represent the
 unknown and the afterlife.
The poem suggests that death is not
 something to be feared,
but rather a natural part of life that
 we should all embrace.

The poem begins with the speaker
 stopping by the woods on a snowy
 evening,
a moment of contemplation and
 reflection. The speaker notices the
 beauty
of the snow-covered woods and is
 reminded of the fleeting nature of
 life.
The poem continues as the speaker
 considers the temptation of continuin
g
on to his destination, but instead
 chooses to pause and take in the
 beauty
of the woods.

The woods are used as a metaphor for
 the unknown and the afterlife.
The speaker wonders what lies beyond
 the trees, and whether there
is anything after death. The poem
 suggests that death is not
something to be feared, but rather a
 natural part of life
```

```
that we should all embrace.

The final lines of the poem are perhaps
 the most famous and poignant:
"And miles to go before I sleep, and
 miles to go before I sleep."
These lines suggest that even though
 the speaker has paused
to appreciate the beauty of the woods,
 they still have a
long journey ahead of them. The poem is
 a reminder that life
is fleeting, and that we should take
 the time to appreciate
the beauty around us, even as we work
 towards our goals and ambitions.

Overall, "Stopping by Woods on a Snowy
 Evening" is a beautiful
and thought-provoking poem that uses
 the metaphor of the woods
to explore the concept of death and the
 afterlife.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

Model encouragement can also help in question-answer tasks.

```
Prompt Template to Titan and AI21 Labs
 Jurassic:
"""{{Model encouragement}}
{{Question}}"""
```

```
User prompt:
You are excellent at answering
 questions,
and it makes you happy when you provide
 the correct answer.

What moon in the solar system is most
 likely to host life?
```

```
Output:
The moon of Titan, which is a moon of
 the planet Saturn,
```

```
is the most likely to support life in
 the solar system.

Titan is the only known moon in the
 solar system to have a dense
atmosphere and a liquid hydrosphere,
 and scientists believe that
it may contain the chemical building
 blocks for life.
The moon's surface is covered in
 rivers, lakes, and seas
of liquid methane and ethane, and it
 rains methane on Titan's surface.

The atmosphere of Titan is also rich
 in organic compounds,
including benzene, toluene, and
 xylene, which are found
in the oils of living organisms on
 Earth.
Additionally, Titan has a magnetic
 field and a tenuous atmosphere,
which could provide protection from
 harmful radiation and
help maintain a stable climate.

While there is no direct evidence of
 life on Titan,
scientists believe that it is one of
 the most promising
places in the solar system to search
 for life beyond Earth.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

```
Prompt template to Titan and AI21 Labs
 Jurassic:
"""{{Model encouragement}}
{{Question}}
{{Constraints}}"""
```

```
User prompt:
You feel rewarded by helping people
 learn more about climate change.
Could you please explain what climate
 change is?
```

```
Assume your audience is composed of
 high school students.
```

```
Output:
Climate change is the long-term warming
 of the planet,
which can be caused by human activitie
s such as burning fossil fuels
and deforestation, as well as natural
 processes. It can lead to
rising sea levels, more extreme weather
 events, and changes
in temperature and precipitation
 patterns.
```

(Source of prompt: AWS, model used: AI21 Labs Jurassic-2 Ultra v1)

Question-answer, with context

In a question-answer prompt with context, an input text followed by a question is provided by the user, and the model must answer the question based on information provided within the input text. Putting the question in the end after the text can help LLMs on Amazon Bedrock better answer the question. Model encouragement works for this use case as well.

```
Prompt template to Titan and AI21 Labs
 Jurassic:
"""{{Text}}
{{Question}}"""
```

```
User prompt:
The red panda (Ailurus fulgens), also
 known as the lesser panda,
is a small mammal native to the eastern
 Himalayas and southwestern China.
It has dense reddish-brown fur with
 a black belly and legs, white-lined
 ears,
a mostly white muzzle and a ringed
 tail. Its head-to-body length is 51-
63.5 cm
(20.1-25.0 in) with a 28-48.5 cm
 (11.0-19.1 in) tail, and it weighs
 between
3.2 and 15 kg (7.1 and 33.1 lb). It is
 well adapted to climbing due to its
```

```
flexible joints and curved semi-retr
actile claws.

The red panda was first formally
 described in 1825. The two currently
recognized subspecies, the Himalayan
 and the Chinese red panda, genetical
ly
diverged about 250,000 years ago. The
 red panda's place on the evolution
ary
tree has been debated, but modern
 genetic evidence places it in close
affinity with raccoons, weasels, and
 skunks. It is not closely related
to the giant panda, which is a bear,
 though both possess elongated
wrist bones or "false thumbs" used for
 grasping bamboo.
The evolutionary lineage of the red
 panda (Ailuridae) stretches
back around 25 to 18 million years ago,
 as indicated by extinct
fossil relatives found in Eurasia and
 North America.

The red panda inhabits coniferou
s forests as well as temperate
 broadleaf
and mixed forests, favoring steep
 slopes with dense bamboo cover close
to water sources. It is solitary and
 largely arboreal. It feeds mainly
on bamboo shoots and leaves, but also
 on fruits and blossoms.
Red pandas mate in early spring, with
 the females giving birth
to litters of up to four cubs in
 summer. It is threatened
by poaching as well as destruction and
 fragmentation of habitat
due to deforestation. The species has
 been listed as Endangered
```

```
on the IUCN Red List since 2015. It is
 protected in all range countries.

Based on the information above, what
 species are red pandas closely related
 to?
```

Output:

```
Red pandas are closely related to
 raccoons, weasels, and skunks.
```

(Source of prompt: https://en.wikipedia.org/wiki/Red_panda, model used: Amazon Titan Text)

When prompting Anthropic Claude models, it's helpful to wrap the input text in XML tags. In the following example, the input text is enclosed in `<text></text>`.

Prompt template for Anthropic Claude:

```
"""

Human: {{Instruction}}
<text>
{{Text}}
<text>
{{Question}}

Assistant:"""
```

User prompt:

```


Human: Read the following text inside
 <text></text>
XML tags, and then answer the
 question:

<text>
On November 12, 2020, the selection of
 the Weeknd to headline
the show was announced; marking the
 first time a Canadian solo artist
headlined the Super Bowl halftime
 show. When asked about preparations
for the show, the Weeknd stated, "We've
 been really focusing
on dialing in on the fans at home and
 making performances
a cinematic experience, and we want to
 do that with the Super Bowl."

The performance featured a choir whose
 members were dressed in white
```

```
and wore masks over their faces with
 glowing red eyes, and were
standing within a backdrop of a neon
 cityscape. The performance
opened with a white figure dressed the
 same as the choir being
lowered into the backdrop where the
 choir was standing while singing
“Call Out My Name". At this time, the
 Weeknd sat in a convertible
against a skyline backdrop designed to
 resemble the Las Vegas Strip.
For the next part of the performance,
 the backdrop then split open
to reveal the Weeknd, who then
 performed "Starboy", followed by "The
 Hills".

Next, performing the song "Can't Feel
 My Face", the Weeknd traveled
through a labyrinth constructed behind
 the stage, joined by dancers
dressed in red blazers and black
 neckties similar to his,
but with their faces covered with
 bandages, in keeping with
the aesthetic of his fourth studio
 album After Hours (2020).
The dancers would wear these bandages
 throughout the performance.
In the labyrinth section of the
 performance, camerawork was visually
 unsteady.
The next songs performed were "I Feel
 It Coming", "Save Your Tears",
and "Earned It". For the "Earned It"
 performance, the Weeknd
was accompanied by violinists. For
 the finale of the show,
the Weeknd took to the field of the
 stadium with his dancers to perform
“Blinding Lights". He and the dancers
 entered the field by performing
```

```
"House of Balloons / Glass Table
 Girls". The performance ended
with an array of fireworks.
</text>
Based on the text above, what songs did
 the Weeknd play
at the Super Bowl halftime show?

Assistant:
```

```
Output:
Based on the text, the songs the Weeknd
 played
at the Super Bowl halftime show were:

- Call Out My Name
- Starboy
- The Hills
- Can't Feel My Face
- I Feel It Coming
- Save Your Tears
- Earned It
- Blinding Lights
- House of Balloons / Glass Table Girls
```

(Source of prompt: [Wikipedia on the Super Bowl LV halftime show](), model used: Anthropic Claude)

Summarization

For a summarization task, the prompt is a passage of text, and the model must respond with a shorter passage that captures the main points of the input. Specification of the output in terms of length (number of sentences or paragraphs) is helpful for this use case.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""The following is text from a {{Text
 Category}}:
{{Text}}
Summarize the {{Text Category}} in
 {{length of summary,
```

```
User prompt:
The following is text from a restauran
t review:
“I finally got to check out Alessandr
o’s Brilliant Pizza
and it is now one of my favorite
 restaurants in Seattle.
```

```
e.g., "one sentence" or "one paragraph
"}}"""
```

```
The dining room has a beautiful view
 over the Puget Sound
but it was surprisingly not crowed. I
 ordered the fried
castelvetrano olives, a spicy
 Neapolitan-style pizza
and a gnocchi dish. The olives were
 absolutely decadent,
and the pizza came with a smoked
 mozzarella, which was delicious.
The gnocchi was fresh and wonderful.
 The waitstaff were attentive,
and overall the experience was lovely.
 I hope to return soon."
Summarize the above restaurant review
 in one sentence.
```

```
Output:
Alessandro's Brilliant Pizza is a
 fantastic restaurant
in Seattle with a beautiful view over
 Puget Sound that offers
decadent and delicious food.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

In the following example, Anthropic Claude summarizes the given text in one sentence. To include input text in your prompts, format the text with XML mark up: `<text> {{text content}} </text>`. Using XML within prompts is a common practice when prompting Anthropic Claude models.

```
Prompt template for Anthropic Claude:
"""

Human: Please read the text:
<text>
{{text context}}
</text>
```

```
User prompt:


Human: Please read the text:

<text>
In game theory, the Nash equilibrium,
 named after the mathematician
```

```
Summarize the text in {{length of
 summary, e.g., “one
sentence” or “one paragraph”}}

Assistant:"""
```

```
John Nash, is the most common way to
 define the solution
of a non-cooperative game involving
 two or more players.
In a Nash equilibrium, each player is
 assumed to know
the equilibrium strategies of the
 other players,
and no one has anything to gain by
 changing only one's own strategy.
The principle of Nash equilibrium
 dates back to the time of Cournot,
who in 1838 applied it to competing
 firms choosing outputs.
</text>

Summarize the text in one sentence.

Assistant:
```

```
Output:
The Nash equilibrium is a concept in
 game theory
where each player has adopted a
 strategy that they have no incentive
to deviate from given the other
 players' strategies.
```

(Source of prompt: [Wikipedia on Nash equilibrium](), model used: Anthropic Claude)

Text generation

Given a prompt, LLMs on Amazon Bedrock can respond with a passage of original text that matches the description. Here is one example:

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""Please write a {{Text Category}} in
 the voice of {{Role}}."""
```

```
User prompt:
Please write an email in the voice of a
 friend
congratulating someone on a new job.
```

```
Output:
Hey [Name],

I just found out about your new job,
 and I'm so happy for you!
Congratulations! You deserve this
 opportunity since you worked so hard
to get it. I know you'll do great and
 make a positive impact on your team.
I'm here to support you every step of
 the way. Wishing you all the best
in this exciting new chapter of your
 life!

Cheers,
[Your Name]
```

(Source of prompt: AWS, model used: Amazon Titan Text)

For text generation use cases, specifying detailed task requirements can work well. In the following example, we ask the model to generate response with exclamation points.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""{{Task specification}}
Please write a {{Text Category}} in
 the voice of {{Role}}."""
```

```
User prompt:
Write text with exclamation points.
Please write a formatted email in the
 voice of a friend
congratulating someone on a new job.
```

```
Output:
Hey [name]!

I just saw the news and I am so, so
 happy for you!
You deserve this position and this
 company with all your heart.
I can't wait to see you soar and
 accomplish all the things
you've been working towards. If anyone
 is deserving
of a promotion, it's you!
```

```
Congratulations again, friend!

Love,
[name]
```

(Source of prompt: AWS, model used: Amazon Titan Text)

In the following example, a user prompts the model to take on the role of a specific person when generating the text. Notice how the signature reflects the role the model is taking on in the response.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""{{Role assumption}} {{Task
 description}}
"""
```

```
User input:
My name is Jack. Help me write a note
 expressing my gratitude
to my parents for taking my son (their
 grandson) to the zoo.
I miss my parents so much.
```

```
Output:
Dear Mom and Dad,

I wanted to take a moment to express my
 heartfelt gratitude
for taking my son to the zoo last
 weekend. It meant so much to me
that you took the time to make memories
 with him, and I am so grateful
for the love and support you have shown
 him.

I miss you both so much, and I wish we
 could be together more often.
But I know that you will always be
 there for me and my family,
and I am so grateful to have you as my
 parents.

Love,
```

```
Jack
```

(Source of prompt: AWS, model used: AI21 Labs Jurassic-2 Ultra v1)

Code generation

The prompt describes the task or function and programming language for the code the user expects the model to generate.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""Write a function in {{programming
 language}} to
{{task or purpose}}""".
```

```
User input:
Write a python function to download a
 s3 file to local disk.
```

````
Output:
The following code can be used to
 download an s3 file to the local disk.

```Python
import boto3
# Creates a boto3 client object
s3 = boto3.client('s3')
def download_file(bucket_name,
 object_key, file_path):
try:
# Downloads the s3 file to the
 specified file path
s3.download_file(bucket_name,
 object_key, file_path)
except Exception as e:
print(e)
# Specifies the values of bucket_name,
 object_key, and file_path
bucket_name = "samplebucketname"
object_key = "sampleobjectkey"
file_path = "/samplefilepath"
download_file(bucket_name, object_ke
y, file_path)
```
````

```
The given code uses the download_file
 function to download an s3 file
to the local disk. The download_file
 function takes the bucket name,
object key, and file path as input
 parameters. The bucket name
is the name of the s3 bucket where the
 file is located,
the object key is the unique identifie
r of the file within the bucket,
and the file path is the destination
 path on the local disk
where the file will be downloaded.
 The function uses the download_file
 method
of the s3 client to download the file
 from the specified bucket
and write it to the specified file
 path.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

```
Prompt template for Anthropic Claude
"""

Human: Write a function in {{program
ming language}} to
{{task or purpose}}

Assistant:"""
```

```
User prompt:


Human: Write some python code that
 quickly approximates
the square root of a number.

Assistant:
```

````
Output:
Here is a simple function to approxima
te
the square root of a number using
 Newton's method:

```python
def sqrt(n, iterations=10):
x = n
````
````

```
for i in range(iterations):
x = (x + n/x) / 2
return x
```

To use it:

```python
print(sqrt(25))
# 5.0

print(sqrt(15, iterations=20))
# 3.872983346207417
```

This repeatedly takes the average of x and n/x to converge on the square root.
More iterations produce a more accurate approximation.

The advantage of this approach is it converges very quickly,
so even just 10 iterations produces a decent estimate.

(Source of prompt: AWS, model used: Anthropic Claude)

Mathematics

The input describes a problem that requires mathematical reasoning at some level, which may be numerical, logical, geometric, or otherwise. For such problem, it's helpful to ask the model to work through the problem in a piecemeal manner by adding phrases to instructions such as `Let’s think step by step` or `Think step by step to come up with the right answer.`

Prompt template for Titan and AI21 Labs Jurassic:

```
"""{{Math problem description}}
Let’s think step by step."""
```

User prompt:

```
A triangle has two angles of 70 degrees and 50 degrees.
What is the third angle in degrees?
Let’s think step by step.
```

```
Output:
The sum of the three angles in a
 triangle is always 180 degrees.
The two given angles are 70 degrees and
 50 degrees. 70 + 50 = 120.
Therefore, the third angle is 180 -
 120 = 60 degrees.
The third angle is 60 degrees.
```

(Source of prompt: AWS, model used: Amazon Titan Text)

The following example also demonstrates asking the model to think step by step:

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""{{Math problem description}}
Think step by step to come up with the
 right answer."""
```

```
User prompt:
Lucy has 12 colorful marbles, and she
 wants to share them
equally with her 4 friends. How many
 marbles will each friend receive?
Think step-by-step to come up with
 the right answer.
```

```
Output:
1. Lucy has 12 marbles and wants to
 share them equally
with her 4 friends, then each friend
 will receive 12 / 4 = 3 marbles.
2. Therefore, each friend will
 receive 3 marbles.
```

(Source of prompt: AWS, model used: AI21 Labs Jurassic-2 Ultra v1)

Reasoning/logical thinking

For complex reasoning tasks or problems that requires logical thinking, we can ask the model to make logical deductions and explain its answers.

Prompt template for Titan and AI21 Labs Jurassic: *"""Question: {{question}} Please output the answer and then explain your answer:"""*	**User prompt:** *Question: Which word is the odd one out? A. accomplished B. good C. horrible D. outstanding Please output the answer and then explain your answer:*
	Output: *The odd one out is C. horrible. The other three words describe positive qualities or achievements, whereas horrible is an negative adjective.*

(Source of prompt: AWS, model used: Amazon Titan Text)

Here is another example using the Anthropic Claude model:

Prompt template for Anthropic Claude: *""" Human: {{question}} Please provide the answer and show the reasoning. Assistant:"""*	**User prompt** *Human: The barber is the "one who shaves all those, and those only, who do not shave themselves". Does the barber shave himself? Why is this a paradox? Please provide the answer and show the reasoning. Assistant:*
	Output: *This is an example of a paradox known as the Barber paradox. Here is an explanation: - The statement defines a barber who shaves all and only those people who do not shave themselves. - So if the barber shaves himself, he must shave all those who do not shave themselves, which includes himself. But the barber cannot shave himself as per the initial statement. - If the barber does not shave himself, then he would fit into the group of people who do not shave themselves, and*

```
 so he must shave himself,  according
 to the statement. But again this
 contradicts the initial statement. -
 So the barber cannot shave himself and
 also cannot not shave himself.  This
 is self-contradictory based on the
 initial statement about the barber.
 - Therefore, the statement leads
 to a paradox where both possibili
ties  (the barber shaving himself
 or not shaving himself) result in
 a contradiction.  In summary, the
 paradox arises because the definition
 of the barber's  behavior is self-cont
radictory when applied to the barber
 himself.  This makes it impossible to
 determine if the barber shaves himself
  or not based on the given statement
 alone.
```

(Source of prompt: https://en.wikipedia.org/wiki/Barber_paradox, model used: Anthropic Claude)

Entity extraction

For entity extraction from a provided input question. Extract entities from generated text and place them in XML tags for further processing.

```
Prompt template for Titan and AI21 Labs
 Jurassic:
"""You are an expert entity extractor
 from provided input question. You are
 responsible for extracting following
 entities: {{ list of entities}}

Please follow below instructions while
 extracting the entity A, and reply in
 <entityA> </entityA> XML Tags:
{{ entity A extraction instructi
ons}}
```

```
Please follow below instructions while
 extracting the entity B, and reply in
 <entityB> </entityB> XML Tags:
{{ entity B extraction instructi
ons}}

Below are some examples:

{{ some few shot examples showing
 model extracting entities from give
 input }}
```

(Source of prompt: AWS, model used: Amazon Titan Text G1- Premier)

Example:

```
User: You are an expert entity extractor who extracts entities from provided input
 question.
You are responsible for extracting following entities: name, location
Please follow below instructions while extracting the Name, and reply in <name></name>
XML Tags:

- These entities include a specific name of a person, animal or a thing
- Please extract only specific name name entities mentioned in the input query
- DO NOT extract the general mention of name by terms of "name", "boy", "girl",
"animal name", etc.


Please follow below instructions while extracting the location, and reply
in <location></location> XML Tags:
- These entities include a specific location of a place, city, country or a town
- Please extract only specific name location entities mentioned in the input query
- DO NOT extract the general mention of location by terms of "location", "city",
 "country",
"town", etc.

If no name or location is found, please return the same input string as is.

Below are some examples:
```

```
input: How was Sarah's birthday party in Seattle, WA?
output:  How was <name>Sarah's</name> birthday party
in <location>Seattle, WA</location>?

input: Why did Joe's father go to the city?
output:  Why did <name>Joe's</name> father go to the city?

input: What is the zipcode of Manhattan, New york city?
output: What is the zipcode of <location>Manhattan,New york city<location>?

input: Who is the mayor of San Francisco?
Bot:
```

Chain-of-thought reasoning

Provide a step-by-step analysis on how the answer was derived. Fact check and validate how the model produced an answer.

```
Prompt template for Titan and AI21 Labs
 Jurassic:

""" {{Question}}
{{ Instructions to Follow }}
Think Step by Step and walk me through
 your thinking
"""
```

(Source of prompt: AWS, model used: Amazon Titan Text G1- Premier)

Example:

```
User: If Jeff had 100 dollars, and he gave $20 to Sarah,
and bought lottery tickets with another $20. With the lottery
tickets he bought he won 35 dollars. Jeff then went to buy
```

```
his lunch and spend 40 dollars in lunch. Lastly he made a
donation to charity for $20. Stephen met with Jeff and wanted
to lend some money from him for his taxi. How much maximum money
can Jeff give to Stephen, given that he needs to save $10 for
his ride back home?. Please do not answer immediately, think
step by step and show me your thinking.
Bot:
```

Guardrails for Amazon Bedrock

Guardrails for Amazon Bedrock enables you to implement safeguards for your generative AI applications based on your use cases and responsible AI policies. You can create multiple guardrails tailored to different use cases and apply them across multiple foundation models, providing a consistent user experience and standardizing safety controls across generative AI applications. You can configure denied topics to disallow undesirable topics and content filters to block harmful content in inputs and model responses. You can use guardrails with text-only foundation models.

A guardrail consists of the following policies to avoid content that falls into undesirable or harmful categories.

- Denied topics – You can define a set of topics that are undesirable in the context of your application. These topics will be blocked if detected in user queries or model responses.
- Content filters – Adjust filter strengths to filter input prompts or model responses containing harmful content.
- Word filters – Configure filters to block undesirable words, phrases, and profanity.
- Sensitive information filters – Block or mask personally identifiable information (PII) and use regular expressions to define and then block or mask patterns that might correspond to sensitive information.

In addition to above policies, you can also configure the messages to be returned to the user if a user input or model response is in violation of the policies defined in the guardrail.

You can create multiple guardrail versions for your guardrail. When you create a guardrail, a working draft is automatically available for you to iteratively modify. Experiment with different configurations and use the built-in test window to see whether they are appropriate for your use-case. If you are satisfied with a set of configurations, you can create a version of the guardrail and use it with supported foundation models.

Guardrails can be used directly with FMs during the inference API invocation by specifying the guardrail ID and the version. If a guardrail is used, it will evaluate the input prompts and the FM completions against the defined policies.

Topics

Supported regions and models for Guardrails for Amazon Bedrock

Guardrails for Amazon Bedrock is supported in the following regions:

Region
US East (N. Virginia)
US West (Oregon)

You can use Guardrails for Amazon Bedrock with the following models:

Model name	Model ID
Anthropic Claude Instant v1	anthropic.claude-instant-v1
Anthropic Claude v1.0	anthropic.claude-v1
Anthropic Claude v2.0	anthropic.claude-v2
Anthropic Claude v2.1	anthropic.claude-v2:1
Anthropic Claude 3 Haiku	anthropic.claude-3-haiku-20240307-v1
Anthropic Claude 3 Opus	anthropic.claude-3-opus-20240229-v1

Model name	Model ID
Anthropic Claude 3 Sonnet	anthropic.claude-3-sonnet-20240229-v1
Command	cohere.command-text-v14
Command Light	cohere.command-text-v14
Jurassic-2 Mid	ai21.j2-mid
Jurassic-2 Ultra	ai21.j2-ultra-v1
Llama 2 Chat 13B	meta.llama2-13b-chat-v1
Llama 2 Chat 70B	meta.llama2-70b-chat-v1
Mistral 7B Instruct	mistral.mistral-7b-instruct-v0:2
Mistral 8X7B Instruct	mistral.mixtral-8x7b-instruct-v0:1
Mistral Large	mistral.mistral-large-2402-v1:0
Titan Text G1 - Express	amazon.titan-text-express-v1
Titan Text G1 - Lite	amazon.titan-text-lite-v1

For a list of all the models supported by Amazon Bedrock amd their IDs, see [Amazon Bedrock model IDs]

Components of a guardrail in Amazon Bedrock

A guardrail in Amazon Bedrock consists of filters that you can configure, topics that you can define to block, and messages to send to users when content is blocked or filtered.

Topics

Content filters

Guardrails support the following content filters to detect and filter harmful user inputs and FM-generated outputs.

- **Hate –** Describes input prompts and model responses that discriminate, criticize, insult, denounce, or dehumanize a person or group on the basis of an identity (such as race, ethnicity, gender, religion, sexual orientation, ability, and national origin).
- **Insults –** Describes input prompts and model responses that includes demeaning, humiliating, mocking, insulting, or belittling language. This type of language is also labeled as bullying.
- **Sexual –** Describes input prompts and model responses that indicates sexual interest, activity, or arousal using direct or indirect references to body parts, physical traits, or sex.
- **Violence –** Describes input prompts and model responses that includes glorification of or threats to inflict physical pain, hurt, or injury toward a person, group or thing.
- **Misconduct** – Describes input prompts and model responses that seeks or provides information about engaging in criminal activity, or harming, defrauding, or taking advantage of a person, group or institution.
- **Prompt attack** (Only applies to prompts) – Describes user prompts intended to bypass the safety and moderation capabilities of a foundation model in order to generate harmful content (also known as jailbreak), and ignore and override instructions specified by the developer (referred to as prompt injection).

Content filtering depends on the confidence classification of user inputs and FM responses across each of the six harmful categories. All input and output statements are classified into one of four confidence levels (NONE, LOW, MEDIUM, HIGH) for each harmful category. For example, if a statement is classified as *Hate* with HIGH confidence, the likelihood of the statement representing hateful content is high. A single statement can be classified across multiple categories with varying confidence levels. For example, a single statement can be classified as *Hate* with HIGH confidence, *Insults* with LOW confidence, *Sexual* with NONE confidence, and *Violence* with MEDIUM confidence.

For each of the harmful categories, you can configure the strength of the filters. The filter strength determines the degree of filtering harmful content. As you increase the filter strength, the likelihood of filtering harmful content increases and the probability of seeing harmful content in your application reduces. The following table shows the degree of content that each filter strength blocks and allows.

Filter strength	Blocked content confidence	Allowed content confidence
None	No filtering	None, Low, Medium, High
Low	High	None, Low, Medium
Medium	High, Medium	None, Low
High	High, Medium, Low	None

Denied topics

Guardrails can be configured with a set of denied topics that are undesirable in the context of your generative AI application. For example, a bank may want their online assistant to avoid any conversation related to investment advice or engage in conversations related to fraudulent activities such as money laundering.

You can define up to 30 denied topics. Input prompts and model completions will be evaluated against each of these topics. If one of the topics is detected, the blocked message configured as part of the guardrail will be returned to the user.

Denied topics can be defined by providing a natural language definition of the topic along with a few optional example phrases of the topic. The definition and example phrases are used to detect if an input prompt or a model completion belongs to the topic.

Denied topics are defined with the following parameters.

- Name – The name of the topic. The name should be a noun phrase. Don't describe the topic in the name. For example:
  - `Investment Advice`
- Definition – Up to 200 characters summarizing the topic content. The description should describe the content of the topic and its subtopics.

> **Note**
>
> For best results, adhere to the following principles:
>
> - Don't include examples or instructions in the description.

- Don't use negative language (such as "don't talk about investment" or "no content about investment").

The following is an example topic description that you can provide:

- **`Investment advice refers to inquires, guidance or recommendations regarding the management or allocation of funds or assets with the goal of generating returns or achieving specific financial objectives.`**

- Sample phrases – A list of up to five sample phrases that refer to the topic. Each phrase can be up to 100 characters. An sample is a prompt or continuation that shows what kind of content should be filtered out. For example:
  - **`Is investing in the stocks better than bonds?`**
  - **`Should I invest in gold?`**

Word filters

Guardrails for Amazon Bedrock offers the following word filters to block words and phrases in prompts and model responses.:

- Profanity filter – Turn on to block profane words. The list of profanity is based on conventional definitions of profanity and is continually updated.
- Custom word filter – Add words and phrases of up to three words to a list. You can add up to 10,000 items to the custom word filter. You have the following options for adding words using the Amazon Bedrock console;:
  - Add manually in the text editor.
  - Upload a .txt or .csv file.
  - Upload from an Amazon S3 object.

Sensitive information filters

Guardrails for Amazon Bedrock offers the following filters to block or mask sensitive information:

- Personally identifiable information (PII) filter – Select PII types from a predefined list for a guardrail to recognize and act upon. You can choose from the following PIIs:
  - `ADDRESS`

- AGE
- AWS_ACCESS_KEY
- AWS_SECRET_KEY
- CA_HEALTH_NUMBER
- CA_SOCIAL_INSURANCE_NUMBER
- CREDIT_DEBIT_CARD_CVV
- CREDIT_DEBIT_CARD_EXPIRY
- CREDIT_DEBIT_CARD_NUMBER
- DRIVER_ID
- EMAIL
- INTERNATIONAL_BANK_ACCOUNT_NUMBER
- IP_ADDRESS
- LICENSE_PLATE
- IP_ADDRESS
- MAC_ADDRESS
- NAME
- PASSWORD
- PHONE
- PIN
- SWIFT_CODE
- UK_NATIONAL_HEALTH_SERVICE_NUMBER
- UK_NATIONAL_INSURANCE_NUMBER
- UK_UNIQUE_TAXPAYER_REFERENCE_NUMBER
- URL
- USERNAME
- US_BANK_ACCOUNT_NUMBER
- PIN
- US_BANK_ROUTING_NUMBER
- US_INDIVIDUAL_TAX_IDENTIFICATION_NUMBER
- US_PASSPORT_NUMBER

  - US_SOCIAL_SECURITY_NUMBER
  - VEHICLE_IDENTIFICATION_NUMBER
- Regex filter – Use regular expressions to define patterns for a guardrail to recognize and act upon.

For sensitive information filters, you configure the guardrail's behavior. You have the following options:

- Block – If sensitive information is detected in the prompt or response, the guardrail blocks all the content and returns a message that you configure.
- Mask – If sensitive information is detected in the prompt, the guardrail masks it with an identifier. For example, if you configure a *name* filter, the response containing the name is returned with the name replaced with **[NAME]**.

Prerequisites for using Guardrails for Amazon Bedrock

Before you can use Guardrails for Amazon Bedrock, you must fulfill the following prerequisites:

1. Request access to the model or models with which you want to use a guardrail.
2. Ensure that your IAM role has the necessary permissions to perform actions related to Guardrails for Amazon Bedrock.

To prepare for the creation of your guardrail, consider preparing the following components of the guardrail in advance:

- Look at the available content filters and determine the strength that you want to apply to each filter for prompts and model responses.
- Determine the topics to block and consider how to define them and the sample phrases to include. In defining denied topics, avoid using instructions or negative language. Describe and define the topic in a precise and concise manner.
- Prepare a list of words and phrases (each up to three words) to block with word filters. Your list can contain up to 10,000 items and be up to 50 KB. Save the list in a .txt or .csv file. If you prefer, you can upload it to an Amazon S3 bucket using the Amazon Bedrock console.
- Look at the list of personally identifiable information in Sensitive information filters and consider which ones your guardrail should block or mask.

- Consider regex expressions that might match sensitive information and consider which ones your guardrail should block or mask with the use of sensitive information filters.
- Consider the messages to send users when the guardrail blocks a prompt or model response.

Create a guardrail

You create a guardrail by setting up the configurations, defining topics to deny, providing filters to handle harmful and sensitive content, and writing messages for when prompts and user responses are blocked.

A guardrail must contain at least one filter and messaging for when prompts and user responses are blocked. You can opt to use the default messaging. You can add filters and iterate upon your guardrail later by following the steps at Edit a guardrail to configure all the components that you need for your guardrail.

Select the tab corresponding to your method of choice and follow the steps.

Console

To create a guardrail

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, choose **Guardrails**.
3. In the **Guardrails** section, choose **Create guardrail**.
4. On the **Provide guardrail details** page, do the following:

   a. In the **Guardrail details** section, provide a **Name** and optional **Description** for the guardrail.

   b. (Optional) By default, your guardrail is encrypted with an AWS managed key. To use your own customer-managed KMS key, select the right arrow next to **KMS key selection** and select the **Customize encryption settings (advanced)** checkbox. You can choose an existing AWS KMS key or select **Create an AWS KMS key** to create a new one.

   c. (Optional) To add tags to your guardrail, select the right arrow next to **Tags**. Then, choose **Add new tag** and define key-value pairs for your tags. For more information, see Tag resources.

d. Choose **Next**.

> **Note**
>
> You must configure at least one filter to create a guardrail. You can then choose **Skip to Review and create** to skip the creation of other filters.

5. (Optional) On the **Configure content filters** page, set up how strongly you want to filter out content related to the categories defined in Content filters by doing the following:

   a. To configure filters for prompts to a model, select **Enable filters for prompts** in the **Filter strengths for model prompts** section. Configure how strict you want each filter to be for prompts that the user provides to the model.

   b. To configure filters for model responses, select **Enable filters for responses** in **Filter strengths for responses**. Configure how strict you want each filter to be for responses that the model returns.

   c. Choose **Next**.

6. (Optional) On the **Add denied topics** page, do the following:

   a. To define a topic to block, choose **Add denied topic**. Then do the following:

      i. Enter a **Name** for the topic.

      ii. In the **Definition for topic** box, define the topic. For guidelines on how to define a denied topic, see Denied topics.

      iii. (Optional) To add representative input prompts or model responses related to this topic, select the right arrow next to **Add sample phrases**. Enter a phrase in the box. To add another phrase, choose **Add phrase**.

      iv. When you're done configuring the denied topic, choose **Confirm**.

   b. You can perform the following actions with the **Denied topics**.

      - To add another topic, choose **Add denied topic**.
      - To edit a topic, choose the three dots icon in the same row as the topic in the **Actions** column. Then choose **Edit**. After you are finished editing, choose **Confirm**.
      - To delete a topic or topics, select the checkboxes for the topics to delete. Choose **Delete** and then select **Delete selected**.

- To delete all the topics, choose **Delete** and then select **Delete all**.
- To configure the size of each page in the table or the column display in the table, choose the settings icon ( ). Set your preferences and then choose **Confirm**.

c. When you are finished configuring denied topics, select **Next**.

7. (Optional) On the **Add word filters** page, do the following:

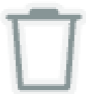
a. In the **Filter profanity** section, select **Filter profanity** to block profanity in prompts and responses. The list of profanity is based on conventional definitions and is continually updated.

b. In the **Add custom words and phrases** section, select how to add words and phrases for the guardrail to block. If you choose to upload a file, each line in the file should contain one word or a phrase of up to three words. Don't include a header. You have the following options:

Option	Instructions
Add words and phrases manually	Directly add words and phrases in the **View and edit words and phrases** section.
Upload from a local file	To upload a .txt or .csv file containing the words and phrases, choose **Choose file** after selecting this option.
Upload from Amazon S3 object	To upload a file from Amazon S3, specify the **S3 object** after selecting this option. Each line in the file should contain one word or a phrase of up to three words.

c. You edit the words and phrases for the guardrail to block in the **View and edit words and phrases** section. You have the following options:

- If you uploaded a word list from a local file or Amazon S3 object, this section will populate with your word list. To filter for items with errors, choose **Show errors**.

- To add an item to the word list, choose **Add word or phrase**. Enter a word or a phrase of up to three words in the box and press **Enter** or select the checkmark icon to confirm the item.
- To edit an item, choose the edit icon () next to the item.
- To delete an item from the word list, choose the trash can icon () or, if you're editing an item, choose the delete icon () next to the item.
- To delete items that contain errors, choose **Delete all** and then select **Delete all rows with error**
- To delete all items, choose **Delete all** and then select **Delete all rows**
- To search for an item, enter an expression in the search bar.
- To show only items with errors, choose the dropdown menu labeled **Show all** and select **Show errors only**.
- To configure the size of each page in the table or the column display in the table, choose the settings icon (). Set your preferences and then choose **Confirm**.
- By default, this section displays the **Table** editor. To switch to a text editor in which you can enter a word or phrase in each line, select **Text editor**. The **Text editor** provides the following features:
  - You can copy a word list from another text editor and paste it into this editor.
  - A red X icon appears next to items containing errors and a list of errors appears at the below the editor.

8. (Optional) On the **Add sensitive information filters page**, configure filters to block or mask sensitive information. For more information, see Sensitive information filters. Do the following:

a. In the **PII types** section, configure the personally identifiable information (PII) categories to block or mask. You have the following options:

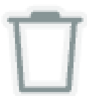
- To add a PII type, choose **Add a PII type**. Then, do the following:

  1. In the **Type** column, select a PII type.
  2. In the **Guardrail behavior** column, select whether the guardrail should **Block** content containing the PII type or **Mask** it with an identifier.

- To add all PII types, choose the dropdown arrow next to **Add a PII type**. Then select the guardrail behavior to apply to them.

> **Warning**
>
> If you specify a behavior, any existing behavior that you configured for PII types will be overwritten.

- To delete a PII type, choose the trash can icon

  ().

- To delete rows that contain errors, choose **Delete all** and then select **Delete all rows with error**

- To delete all PII types, choose **Delete all** and then select **Delete all rows**

- To search for a row, enter an expression in the search bar.

- To show only rows with errors, choose the dropdown menu labeled **Show all** and select **Show errors only**.

- To configure the size of each page in the table or the column display in the table, choose the settings icon

  ().

  Set your preferences and then choose **Confirm**.

b. In the **Regex patterns** section, use regular expressions to define patterns for the guardrail to filter. You have the following options:

- To add a pattern, choose **Add regex pattern**. Configure the following fields:

Field	Description
Name	A name for the pattern
Regex pattern	A regular expression that defines the pattern
Guardrail behavior	Choose whether to **Block** content containing the pattern or to **Mask** it with an identifier. To mask the pattern only in logs, choose **None**.
Add description	(Optional) Write a description for the pattern

- To edit a pattern, choose the three dots icon in the same row as the topic in the **Actions** column. Then choose **Edit**. After you are finished editing, choose **Confirm**.
- To delete a pattern or patterns, select the checkboxes for the patterns to delete. Choose **Delete** and then select **Delete selected**.
- To delete all the patterns, choose **Delete** and then select **Delete all**.
- To search for a pattern, enter an expression in the search bar.
- To configure the size of each page in the table or the column display in the table, choose the settings icon (). Set your preferences and then choose **Confirm**.

c. When you finish configuring sensitive information filters, choose **Next**.

9. On the **Define blocked messaging** page, set up the messages that you want to return to the user when the guardrail detects and blocks content. Do the following:

   a. In the **Messaging shown for blocked prompts** field in the **Blocked messaging** section, enter the message to display if the guardrail blocks a prompt sent to the model.

   b. In the **Messaging shown for blocked responses** field in the **Blocked messaging** section, enter the message to display if the guardrail blocks a response generated by the model.

   c. Choose **Next**.

10. Review and create – Review the settings for your guardrail.

    a. Choose **Edit** in any section you want to make changes to.

    b. When you are satisfied with the settings for your guardrail, select **Create** to create the guardrail.

API

To create a guardrail, send a CreateGuardrail request. The request format is as follows:

```
POST /guardrails HTTP/1.1
Content-type: application/json

{
   "blockedInputMessaging": "string",
   "blockedOutputsMessaging": "string",
   "contentPolicyConfig": {
      "filtersConfig": [
         {
            "inputStrength": "NONE | LOW | MEDIUM | HIGH",
            "outputStrength": "NONE | LOW | MEDIUM | HIGH",
            "type": "SEXUAL | VIOLENCE | HATE | INSULTS | MISCONDUCT |
 PROMPT_ATTACK"
         }
      ]
   },
    "wordPolicyConfig": {
    "wordsConfig": [
      {
        "text": "string"
      }
    ],
    "managedWordListsConfig": [
      {
        "type": "string"
      }
    ]
  },
  "sensitiveInformationPolicyConfig": {
    "piiEntitiesConfig": [
      {
```

```
          "type": "string",
          "action": "string"
        }
      ],
      "regexesConfig": [
        {
          "name": "string",
          "description": "string",
          "regex": "string",
          "action": "string"
        }
      ]
    },
    "description": "string",
    "kmsKeyId": "string",
    "name": "string",
    "tags": [
       {
          "key": "string",
          "value": "string"
       }
    ],
    "topicPolicyConfig": {
       "topicsConfig": [
          {
             "definition": "string",
             "examples": [ "string" ],
             "name": "string",
             "type": "DENY"
          }
       ]
    }
}
```

- Specify a name and `description` for the guardrail.
- Specify messages for when the guardrail successfully blocks a prompt or a model response in the `blockedInputMessaging` and `blockedOutputsMessaging` fields.
- Specify topics for the guardrail to deny in the `topicPolicy` object. Each item in the `topics` list pertains to one topic. For more information about the fields in a topic, see Topic.
  - Give a name and `description` so that the guardrail can properly identify the topic.
  - Specify DENY in the `action` field.

  - (Optional) Provide up to five examples that you would categorize as belonging to the topic in the `examples` list.
- Specify filter strengths for the harmful categories defined in Amazon Bedrock in the `contentPolicy` object. Each item in the `filters` list pertains to a harmful category. For more information, see Content filters. For more information about the fields in a content filter, see ContentFilter.
  - Specify the category in the `type` field.
  - Specify the strength of the filter for prompts in the `strength` field of the `textToTextFiltersForPrompt` field and for model responses in the `strength` field of the `textToTextFiltersForResponse`.
- (Optional) Attach any tags to the guardrail. For more information, see Tag resources.
- (Optional) For security, include the ARN of a KMS key in the `kmsKeyId` field.

The response format is as follows:

```
HTTP/1.1 202
Content-type: application/json

{
   "createdAt": "string",
   "guardrailArn": "string",
   "guardrailId": "string",
   "version": "string"
}
```

Test a guardrail

After you create a guardrail, a *working draft* (`DRAFT`) version is available. The working draft is a version of the guardrail that you can continually edit and iterate upon until you reach a satisfactory configuration for your use case. You can test the working draft or other versions of the guardrail to see whether the configurations are appropriate for your use-case. Edit configurations in the working draft and test different prompts to see how well the guardrail evaluates and intercepts the prompts or responses. When you are satisfied with the configuration, you can then create a version of the guardrail, which acts as a snapshot of the configurations of the working draft when you create the version. You can use versions to streamline guardrails deployment to production applications every time you make modifications to your guardrails. Any changes to the working draft or a new version created will not be reflected in your generative AI application until you specifically use the new version in the application.

Console

To test a guardrail

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Choose **Guardrails** from the left navigation pane. Then, select a guardrail in the **Guardrails** section.
3. A test window appears on the right. You have the following options in the test window:

   a. By default, the working draft of the guardrail is used in the test window. To test a different version of the guardrail, choose **Working draft** at the top of the test window and then select the version.
   b. To select a model, choose **Select model**. After you make a choice, select **Apply**. To change the model, choose **Change**.
   c. Enter a prompt in the **Prompt** box.
   d. To elicit a model response, select **Run**.
   e. The model returns a response in the **Final response** box (that may be modified by the guardrail). If the guardrail blocks or filters the prompt or model response, a message appears under **Guardrail check** that informs you how many violations the guardrail detected.
   f. To view the topics or harmful categories in the prompt or response that were recognized and allowed past the filter or blocked by it, select **View trace**.
   g. Use the **Prompt** and **Model response** tabs to view the topics or harmful categories that were filtered or blocked by the guardrail.

You can also test the guardrail in the **Text playground**. Select the playground and select the **Guardrail** in the **Configurations** pane before testing prompts.

API

To use a guardrail in model invocation, send an [InvokeModel](#) or [InvokeModelWithResponseStream](#) request.

Request format

The request endpoints for invoking a model, with and without streaming, are as follows. Replace *modelId* with the ID of the model to use.

- `InvokeModel` – POST /model/*modelId*/invoke HTTP/1.1
- `InvokeModelWithResponseStream` – POST /model/*modelId*/invoke-with-response-stream HTTP/1.1

The header for both API operations is of the following format.

```
Accept: accept
Content-Type: contentType
X-Amzn-Bedrock-Trace: trace
X-Amzn-Bedrock-GuardrailIdentifier: guardrailIdentifier
X-Amzn-Bedrock-GuardrailVersion: guardrailVersion
```

The parameters are described below.

- Set `Accept` to the MIME type of the inference body in the response. The default value is `application/json`.
- Set `Content-Type` to the MIME type of the input data in the request. The default value is `application/json`.
- Set `X-Amzn-Bedrock-Trace` to `ENABLED` to enable a trace to see amongst other things what content was blocked by Guardrails and why..
- Set `X-Amzn-Bedrock-GuardrailIdentifier` with the guardrail identifier of the guardrail you want to apply to the request to the request and model response.
- Set `X-Amzn-Bedrock-GuardrailVersion` with the version of the guardrail you want to apply to the request and model response.

The general request body format is shown in the following example. The `tagSuffix` property is only used with *Input tagging*. You can also configure the guardrail on streaming synchronously or asynchronously by using `streamProcessingMode`. This only works with `InvokeModelWithResponseStream`.

```
{
    <see model details>,
    "amazon-bedrock-guardrailConfig": {
```

```
        "tagSuffix": "string",
        "streamProcessingMode": "SYNCHRONOUS" | "ASYNCHRONOUS"
    }
}
```

> **Warning**
>
> You will get an error in the following situations
>
> - You enable the guardrail but there is no `amazon-bedrock-guardrailConfig` field in the request body.
> - You disable the guardrail but you specify an `amazon-bedrock-guardrailConfig` field in the request body.
> - You enable the guardrail but the `contentType` is not `application/json`.

To see the request body for different models, see [Inference parameters for foundation models](#).

> **Note**
>
> For Cohere Command models, you can only specify one generation in the `num_generations` field if you use a guardrail.

If you enable a guardrail and its trace, the general format of the response for invoking a model, with and without streaming, is as follows. To see the format of the rest of the body for each model, see [Inference parameters for foundation models](#). The *`contentType`* matches what you specified in the request.

- `InvokeModel`

```
HTTP/1.1 200
Content-Type: contentType

{
    <see model details for model-specific fields>,
    "completion": "<model response>",
    "amazon-bedrock-guardrailAction": "INTERVENED | NONE",
    "amazon-bedrock-trace": {
        "guardrail": {
```

```
            "modelOutput": [
                "<see model details for model-specific fields>"
            ],
            "input": {
                "<sample-guardrailId>": {
                    "topicPolicy": {
                        "topics": [
                            {
                                "name": "string",
                                "type": "string",
                                "action": "string"
                            }
                        ]
                    },
                    "contentPolicy": {
                        "filters": [
                            {
                                "type": "string",
                                "confidence": "string",
                                "action": "string"
                            }
                        ]
                    },
                    "wordPolicy": {
                        "customWords": [
                            {
                                "match": "string",
                                "action": "string"
                            }
                        ],
                        "managedWordLists": [
                            {
                                "match": "string",
                                "type": "string",
                                "action": "string"
                            }
                        ]
                    },
                    "sensitiveInformationPolicy": {
                        "piiEntities": [
                            {
                                "type": "string",
                                "match": "string",
                                "action": "string"
```

```
                            }
                        ],
                        "regexes": [
                            {
                                "name": "string",
                                "regex": "string",
                                "match": "string",
                                "action": "string"
                            }
                        ]
                    }
                }
            },
            "outputs": ["<same guardrail trace format as input>"]
        }
    }
}
```

- `InvokeModelWithResponseStream` – Each response returns a `chunk` whose text is in the `bytes` field, alongside any exceptions that occur. The guardrail trace is returned only for the last chunk.

```
HTTP/1.1 200
X-Amzn-Bedrock-Content-Type: contentType
Content-type: application/json

{
    "chunk": {
      "bytes": "<blob>"
    },
  "internalServerException": {},
  "modelStreamErrorException": {},
  "throttlingException": {},
  "validationException": {},
  "amazon-bedrock-guardrailAction": "INTERVENED | NONE",
  "amazon-bedrock-trace": {
    "guardrail": {
      "modelOutput": ["<see model details for model-specific fields>"],
      "input": {
        "<sample-guardrailId>": {
          "topicPolicy": {
            "topics": [
```

```
                {
                  "name": "string",
                  "type": "string",
                  "action": "string"
                }
              ]
            },
            "contentPolicy": {
              "filters": [
                {
                  "type": "string",
                  "confidence": "string",
                  "action": "string"
                }
              ]
            },
            "wordPolicy": {
              "customWords": [
                {
                  "match": "string",
                  "action": "string"
                }
              ],
              "managedWordLists": [
                {
                  "match": "string",
                  "type": "string",
                  "action": "string"
                }
              ]
            },
            "sensitiveInformationPolicy": {
              "piiEntities": [
                {
                  "type": "string",
                  "match": "string",
                  "action": "string"
                }
              ],
              "regexes": [
                {
                  "name": "string",
                  "regex": "string",
                  "match": "string",
```

```
                "action": "string"
              }
            ]
          }
        }
      },
      "outputs": ["<same guardrail trace format as input>"]
    }
  }
}
```

The response returns the following fields if you enable a guardrail.

- `amazon-bedrock-guardrailAssessment` – Specifies whether the guardrail `INTERVENED` or not (`NONE`).
- `amazon-bedrock-trace` – Only appears if you enable the trace. Contains a list of traces, each of which provides information about the content that the guardrail blocked. The trace contains the following fields:
  - `modelOutput` – An object containing the outputs from the model that was blocked.
  - `input` – Contains the following details about the guardrail's assessment of the prompt:
    - `topicPolicy` – Contains `topics`, a list of assessments for each topic policy that was violated. Each topic includes the following fields:
      - `name` – The name of the topic policy.
      - `type` – Specifies whether to deny the topic.
      - `action` – Specifies that the topic was blocked
    - `contentPolicy` – Contains `filters`, a list of assessments for each content filter that was violated. Each filter includes the following fields:
      - `type` – The category of the content filter.
      - `confidence` – The level of confidence that the output can be categorized as belonging to the harmful category.
      - `action` – Specifies that the content was blocked. This result depends on the strength of the filter set in the guardrail.

- `wordPolicy` – Contains `customWords`, a list of assessments for each word filter that was violated, and `managedWordLists`, a list of assessments for each profanity filter that was violated. Each filter contains the following fields:
  - `match` – The word or phrase that matched the filter.
  - `type` – Specifies `PROFANITY` if it matches the profanity filter. This field doesn't exist for the custom word list filter.
  - `action` – Specifies that the word was blocked.
- `sensitiveInformationPolicy` – Contains the following objects, which contain assessments for personally identifiable information (PII) and regex filters that were violated:
  - `piiEntities` – A list of assessments for each PII filter that was violated. Each filter contains the following fields:
    - `type` – The PII type that was found.
    - `match` – The word or phrase that matched the filter.
    - `action` – Specifies whether the word was `BLOCKED` or replaced with an identifier (`ANONYMIZED`).
  - `regexes` – A list of assessments for each regex filter that was violated. Each filter contains the following fields:
    - `name` – The name of the regex filter.
    - `regex` – The PII type that was found.
    - `match` – The word or phrase that matched the filter.
    - `action` – Specifies whether the word was `BLOCKED` or replaced with an identifier (`ANONYMIZED`).

- `outputs` – A list of details about the guardrail's assessment of the model response. Each item in the list is an object that matches the format of the `input` object. For more details, see the `input` field.

Manage a guardrail

You can modify an existing guardrail to add new configuration policies or edit an existing policy. When you've reached a configuration for your guardrail that you're satisfied with, you can create a static version of the guardrail to use with your models or agents. For more information, see [Deploy an Amazon Bedrock guardrail](#).

View information about your guardrails

Console

To view information about your guardrails

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Choose **Guardrails** from the left navigation pane. Then, select a guardrail in the **Guardrails** section.
3. The **Guardrail overview** section displays the configurations of the guardrail that apply to all versions.
4. To view more information about the working draft, select the **Working draft** in the **Working draft** section.
5. To view more information about a specific version of the guardrail, select the version from the **Versions** section.

To learn more about the working draft and guardrail versions, see Deploy an Amazon Bedrock guardrail.

API

To get information about a guardrail, send a GetGuardrail request and include the ID and version of the guardrail. If you don't specify a version, the response returns details for the `DRAFT` version.

The following is the request format:

```
GET /guardrails/guardrailIdentifier?guardrailVersion=guardrailVersion HTTP/1.1
```

The following is the response format:

```
HTTP/1.1 200
Content-type: application/json

{
   "blockedInputMessaging": "string",
   "blockedOutputsMessaging": "string",
   "contentPolicy": {
```

```
    "filters": [
      {
        "type": "string",
        "inputStrength": "string",
        "outputStrength": "string"
      }
    ]
  },
  "wordPolicy": {
    "words": [
      {
        "text": "string"
      }
    ],
    "managedWordLists": [
      {
        "type": "string"
      }
    ]
  },
  "sensitiveInformationPolicy": {
    "piiEntities": [
      {
        "type": "string",
        "action": "string"
      }
    ],
    "regexes": [
      {
        "name": "string",
        "description": "string",
        "regex": "string",
        "action": "string"
      }
    ]
  },
   "createdAt": "string",
   "description": "string",
   "failureRecommendations": [ "string" ],
   "guardrailArn": "string",
   "guardrailId": "string",
   "kmsKeyArn": "string",
   "name": "string",
   "status": "string",
```

```
    "statusReasons": [ "string" ],
    "topicPolicyConfig": {
        "topics": [
            {
                "definition": "string",
                "examples": [ "string" ],
                "name": "string",
                "type": "DENY"
            }
        ]
    },
    "updatedAt": "string",
    "version": "string"
}
```

To list information about all your guardrails, send a ListGuardrails request.

The following is the request format:

```
GET /guardrails?
guardrailIdentifier=guardrailIdentifier&maxResults=maxResults&nextToken=nextToken
 HTTP/1.1
```

- To list the DRAFT version of all your guardrails, don't specify the guardrailIdentifier field.
- To list all versions of a guardrail, specify the ARN of the guardrail in the guardrailIdentifier field.

You can set the maximum number of results to return in a response in the maxResults field. If there are more results than the number you set, the response returns a nextToken that you can send in another ListGuardrails request to see the next batch of results.

The following is the response format:

```
HTTP/1.1 200
Content-type: application/json

{
    "guardrails": [
        {
            "arn": "string",
```

```
            "createdAt": "string",
            "description": "string",
            "id": "string",
            "name": "string",
            "status": "string",
            "updatedAt": "string",
            "version": "string"
         }
      ],
      "nextToken": "string"
}
```

Edit a guardrail

Console

To edit a guardrail

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Choose **Guardrails** from the left navigation pane. Then, select a guardrail in the **Guardrails** section.
3. To edit the name, description, tags, or model encryption settings for the guardrail, select **Edit** in the **Guardrail overview** section.
4. To edit specific configurations for the guardrail, select **Working draft** in the **Working draft** section.
5. Select **Edit** for the sections containing the settings that you want to change.
6. Make the edits that you need and then select **Save and exit** to implement the edits.

API

To edit a guardrail, send a UpdateGuardrail request. Include both fields that you want to update as well as fields that you want to keep the same.

The following is the request format:

```
PUT /guardrails/guardrailIdentifier HTTP/1.1
Content-type: application/json
```

```
{
   "blockedInputMessaging": "string",
   "blockedOutputsMessaging": "string",
   "contentPolicyConfig": {
      "filtersConfig": [
         {
            "inputStrength": "NONE | LOW | MEDIUM | HIGH",
            "outputStrength": "NONE | LOW | MEDIUM | HIGH",
            "type": "SEXUAL | VIOLENCE | HATE | INSULTS"
         }
      ]
   },
   "description": "string",
   "kmsKeyId": "string",
   "name": "string",
   "tags": [
      {
         "key": "string",
         "value": "string"
      }
   ],
   "topicPolicyConfig": {
      "topicsConfig": [
         {
            "definition": "string",
            "examples": [ "string" ],
            "name": "string",
            "type": "DENY"
         }
      ]
   }
}
```

The following is the response format:

```
HTTP/1.1 202
Content-type: application/json

{
   "guardrailArn": "string",
   "guardrailId": "string",
   "updatedAt": "string",
```

```
    "version": "string"
}
```

Delete a guardrail

You can delete a guardrail when you no longer need to use it. Be sure to disassociate the guardrail from all the resources or applications that use it before you delete the guardrail in order to avoid potential errors.

Console

To delete a guardrail

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Choose **Guardrails** from the left navigation pane. Then, select a guardrail in the **Guardrails** section.
3. In the **Guardrails** section, select a guardrail that you want to delete and then choose **Delete**.
4. Enter **`delete`** in the user input field and choose **Delete** to delete the guardrail.

API

To delete a guardrail, send a DeleteGuardrail request and only specify the ARN of the guardrail in the `guardrailIdentifier` field. Don't specify the `guardrailVersion`

The following is the request format:

```
DELETE /guardrails/guardrailIdentifier?guardrailVersion=guardrailVersion HTTP/1.1
```

> **Warning**
> If you delete a guardrail, all of its versions will be deleted.

If the deletion is successful, the response returns an HTTP 200 status code.

Deploy an Amazon Bedrock guardrail

When you're ready to deploy your guardrail to production, you create a version of it and invoke the version of the guardrail in your application. A version is a snapshot of your guardrail that you create at a point in time when you are iterating on the working draft of the guardrail. Create versions of your guardrail when you are satisfied with a set of configurations. You can use the test window (for more information, see Test a guardrail) to compare how different versions of your guardrail perform in in evaluating the input prompts and model responses and generating controlled responses for the final output. Versions allow you to easily switch between different configurations for your guardrail and update your application with the most appropriate version for your use-case.

Topics

Create and manage a version of a Amazon Bedrock guardrail

The following topics discuss how to create a version of your guardrail when it's ready for deployment, view information about it, and delete it when you no longer need it.

> **Note**
>
> Guardrail versions aren't considered resources and therefore do not have an ARN. IAM Policies that apply to a guardrail apply to all of its versions.

Topics

Create a version of a Amazon Bedrock guardrail

To learn how to create a version of a guardrail, select the tab corresponding to your method of choice and follow the steps.

Console

To create a version

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Guardrails** from the left navigation pane in the Amazon Bedrock console and choose the name of the guardrail that you want to edit in the **Guardrails** section.
3. Carry out one of the following steps.
   - In the **Versions**, section, select **Create**.
   - Choose the **Working draft** and select **Create version** at the top of the page
4. Provide an optional description for the version and then select **Create version**.
5. If successful, you will be redirected to the screen with a list of versions with your new version added there.

API

To create a version of your guardrail, send a CreateGuardrailVersion request. Include the ID and an optional description.

The request format is as follows:

```
POST /guardrails/guardrailIdentifier HTTP/1.1
Content-type: application/json


{
  "clientRequestToken": "string",
  "description": "string"
}
```

The response format is as follows:

```
HTTP/1.1 202
Content-type: application/json

{
   "guardrailId": "string",
   "version": "string"
```

```
}
```

View information about Amazon Bedrock guardrail versions

To learn how to view information about a version or versions of a guardrail, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about your guardrail versions

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Choose **Guardrails** from the left navigation pane. Then, select a guardrail in the **Guardrails** section.
3. In the **Versions** section, select a version to view information about it.

API

To get information about a guardrail version, send a GetGuardrail request and include the ID and version of the guardrail. If you don't specify a version, the response returns details for the `DRAFT` version.

The following is the request format:

```
GET /guardrails/guardrailIdentifier?guardrailVersion=guardrailVersion HTTP/1.1
```

The following is the response format:

```
HTTP/1.1 200
Content-type: application/json

{
   "blockedInputMessaging": "string",
   "blockedOutputsMessaging": "string",
   "contentPolicy": {
      "filters": [
         {
            "inputStrength": "NONE | LOW | MEDIUM | HIGH",
            "outputStrength": "NONE | LOW | MEDIUM | HIGH",
```

```
            "type": "SEXUAL | VIOLENCE | HATE | INSULTS | MISCONDUCT |
PROMPT_ATTACK"
        }
      ]
    },
    "wordPolicy": {
    "words": [
      {
        "text": "string"
      }
    ],
    "managedWordLists": [
      {
        "type": "string"
      }
    ]
  },
  "sensitiveInformationPolicy": {
    "piiEntities": [
      {
        "type": "string",
        "action": "string"
      }
    ],
    "regexes": [
      {
        "name": "string",
        "description": "string",
        "pattern": "string",
        "action": "string"
      }
    ]
  },
   "createdAt": "string",
   "description": "string",
   "failureRecommendations": [ "string" ],
   "guardrailArn": "string",
   "guardrailId": "string",
   "kmsKeyArn": "string",
   "name": "string",
   "status": "string",
   "statusReasons": [ "string" ],
   "topicPolicy": {
      "topics": [
```

```
            {
               "definition": "string",
               "examples": [ "string" ],
               "name": "string",
               "type": "DENY"
            }
         ]
      },
      "updatedAt": "string",
      "version": "string"
}
```

To list information about all your guardrails, send a ListGuardrails request.

The following is the request format:

```
GET /guardrails?
guardrailIdentifier=guardrailIdentifier&maxResults=maxResults&nextToken=nextToken
 HTTP/1.1
```

- To list the DRAFT version of all your guardrails, don't specify the `guardrailIdentifier` field.
- To list all versions of a guardrail, specify the ARN of the guardrail in the `guardrailIdentifier` field.

You can set the maximum number of results to return in a response in the `maxResults` field. If there are more results than the number you set, the response returns a `nextToken` that you can send in another `ListGuardrails` request to see the next batch of results.

The following is the response format:

```
HTTP/1.1 200
Content-type: application/json

{
   "guardrails": [
      {
         "arn": "string",
         "createdAt": "string",
         "description": "string",
         "id": "string",
```

```
            "name": "string",
            "status": "string",
            "updatedAt": "string",
            "version": "string"
         }
      ],
      "nextToken": "string"
}
```

Delete a version of a Amazon Bedrock guardrail

To learn how to delete a version of a guardrail, select the tab corresponding to your method of choice and follow the steps.

Console

If you no longer need a version, you can delete it with the following steps.

To delete a version

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Choose **Guardrails** from the left navigation pane. Then, select a guardrail in the **Guardrails** section.
3. In the **Versions** section, select the version you want to delete and choose **Delete**.
4. A modal appears to warn you about resources that are dependent on this version of the guardrail. Disassociate the version from the resources before you delete to avoid errors.
5. Enter **`delete`** in the user input field and choose **Delete** to delete the guardrail version.

API

To delete a version of a guardrail, send a DeleteGuardrail request. Specify the ARN of the guardrail in the `guardrailIdentifier` field and the version in the `guardrailVersion` field.

The following is the request format:

```
DELETE /guardrails/guardrailIdentifier?guardrailVersion=guardrailVersion HTTP/1.1
```

If the deletion is successful, the response returns an HTTP 200 status code.

Use a guardrail

After you create a guardrail, you can use it in model invocation by setting up your application to call the version while making InvokeModel or InvokeModelWithResponseStream requests. Follow the steps in the API tab of Test a guardrail. Specify the `guardrailVersion` that you want to use.

You can also use a guardrail with other features of Amazon Bedrock.

Topics

Input tagging with Guardrails

Input tagging allows you to mark specific content within the input text that you want to be processed by Guardrails. This is useful when you want to apply Guardrails to certain parts of the input, while leaving other parts unprocessed. By using input tagging with Guardrails, you can better control which parts of the input text should be processed by the Guardrails, ensuring that your generative AI applications adhere to your selected responsible AI policies and use cases.

Tag content for Guardrails

To tag content for Guardrails to process, use the XML tag that is a combination of a reserved prefix and a custom `tagSuffix`. For example:

```
{
    "inputText": """
        You are a helpful assistant.
        Here are some information about my account:
          - There are 10,543 objects in S3 bucket.
          - No active EC2 instances.
        Based on the above, answer the following question:
        Question:
        <amazon-bedrock-guardrails-guardContent_xyz>
        How many S3 objects do I have in my bucket?
        </amazon-bedrock-guardrails-guardContent_xyz>
         ...
        Here are other user queries:
        #amazon-bedrock-guardrails-guardContent_xyz>
        How can I download files from S3?
```

```
        #/amazon-bedrock-guardrails-guardContent_xyz>
    """,
    "amazon-bedrock-guardrailConfig": {
        "tagSuffix": "xyz"
    }
}
```

In the preceding example, the content `*How many S3 objects do I have in my bucket?*` and "*"How can I download files from S3?*" is tagged for Guardrails processing using the tag <amazon-bedrock-guardrails-guardContent_xyz>. Note that the prefix `amazon-bedrock-guardrails-guardContent` is reserved by Guardrails.

Tag Suffix

The tag suffix (`xyz` in the preceding example) is a dynamic value that you must provide in the `tagSuffix` field in `amazon-bedrock-guardrailConfig` to use input tagging. This helps mitigate potential prompt attacks by making the tag structure unpredictable. You are limited to alphanumeric characters with a length between 1 and 20, inclusive. With the example suffix `xyz`, you must enclose all the content to be guarded using the xml tags with your suffix: <amazon-bedrock-guardrails-guardContent_xyz>. and your content </amazon-bedrock-guardrails-guardContent_xyz>.

Multiple tags

You can use the same tag structure multiple times in the input text to mark different parts of the content for Guardrails processing. Nesting of tags is not allowed.

Untagged Content

Any content outside the Guardrails tags will not be processed by Guardrails. This allows you to include instructions, sample conversations, knowledge bases, or other content that you deem safe and do not want to be processed by Guardrails.

Streaming response behavior

The [InvokeModelWithResponseStream](#) API returns data in a streaming format. This allows you to access responses in chunks without waiting for the entire result. When using Guardrails like this, there are two modes of operation: synchronous and asynchronous.

Synchronous mode

In the default synchronous mode, Guardrails will buffer and apply the configured policies to one or more response chunks before the resulting data is sent back to you. The synchronous processing mode introduces some latency to the response chunks, as it means that the response is delayed until the Guardrail scan completes. However, it provides better accuracy, as every response chunk is scanned by Guardrails before being sent to you.

Asynchronous mode

In asynchronous mode, Guardrails sends the response chunks to you as soon as they become available, while applying the configured policies in the background. The advantage is that response chunks are provided immediately, but the accuracy of Guardrails may be reduced. The initial response chunks may contain inappropriate content that has not yet been scanned by Guardrails. As soon as inappropriate content is identified, subsequent chunks will be blocked by Guardrails.

Enabling asynchronous mode

To enable asynchronous mode, you need to include the `streamProcessingMode` parameter in the `amazon-bedrock-guardrailConfig` object of your `InvokeModelWithResponseStream` request:

```
{
   "amazon-bedrock-guardrailConfig": {
   "streamProcessingMode": "ASYNCHRONOUS"
   }
}
```

By understanding the trade-offs between the synchronous and asynchronous modes, you can choose the appropriate mode based on your application's requirements for latency and content moderation accuracy.

Set up permissions for Guardrails

To set up a role with permissions to use guardrails, create an IAM role and attach the following permissions by following the steps at [Creating a role to delegate permissions to an AWS service].

If you are using guardrails with an agent, attach the permissions to a service role with permissions to create and manage agents. You can set up this role in the console or create a custom role by following the steps at [Create a service role for Agents for Amazon Bedrock].

- Permissions to invoke Amazon Bedrock foundation models

- Permissions to create and manage guardrails
- (Optional) Permissions to decrypt your customer-managed AWS KMS key for the guardrail

Permissions to create and manage guardrails

Append the following statement to the `Statement` field in the policy for your role to use guardrails.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CreateAndManageGuardrails",
            "Effect": "Allow",
            "Action": [
                "bedrock:CreateGuardrail",
                "bedrock:CreateGuardrailVersion",
                "bedrock:DeleteGuardrail",
                "bedrock:GetGuardrail",
                "bedrock:ListGuardrails",
                "bedrock:UpdateGuardrail"
            ],
            "Resource": "*"
        }
    ]
}
```

Permissions to invoke guardrail

Append the following statement to the `Statement` field in the policy for the role to allow for model inference and to invoke guardrails.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "InvokeFoundationModel",
            "Effect": "Allow",
            "Action": [
                 "bedrock:InvokeModel",
                 "bedrock:InvokeModelWithResponseStream"
```

```
            ],
            "Resource": [
                "arn:aws:bedrock:region::foundation-model/*"
            ]
        },
        {
            "Sid": "ApplyGuardrail",
            "Effect": "Allow",
            "Action": [
                 "bedrock:ApplyGuardrail"
            ],
            "Resource": [
                "arn:aws:bedrock:region:account-id:guardrail/guardrail-id"
            ]
        }
    ]
}
```

(Optional) Create a customer managed key for your guardrail

Any user with `CreateKey` permissions can create customer managed keys using either the AWS Key Management Service (AWS KMS) console or the CreateKey operation. Make sure to create a symmetric encryption key. After you create your key, set up the following permissions.

1. Follow the steps at Creating a key policy to create a resource-based policy for your KMS key. Add the following policy statements to grant permissions to guardrails users and guardrails creators. Replace each *role* with the role that you want to allow to carry out the specified actions.

```
{
    "Version": "2012-10-17",
    "Id": "KMS Key Policy",
    "Statement": [
        {
            "Sid": "PermissionsForGuardrailsCreators",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::account-id:user/role"
            },
            "Action": [
                "kms:Decrypt",
                "kms:GenerateDataKey",
                "kms:DescribeKey",
```

```
                "kms:CreateGrant"
            ],
            "Resource": "*"
        },
        {
            "Sid": "PermissionsForGuardrailsUusers",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::account-id:user/role"
            },
            "Action": "kms:Decrypt",
            "Resource": "*"
        }
}
```

2. Attach the following identity-based policy to a role to allow it to create and manage guardrails. Replace the *key-id* with the ID of the KMS key that you created.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
         "Sid": "Allow role to create and manage guardrails",
         "Effect": "Allow",
         "Action": [
            "kms:Decrypt",
            "kms:DescribeKey",
            "kms:GenerateDataKey"
            "kms:CreateGrant"
         ],
         "Resource": "arn:aws:kms:region:account-id:key/key-id"
        }
    ]
}
```

3. Attach the following identity-based policy to a role to allow it to use the guardrail you encrypted during model inference or while invoking an agent. Replace the *key-id* with the ID of the KMS key that you created.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
```

```
            "Sid": "Allow role to use an encrypted guardrail during model inference"
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
            ],
            "Resource": "arn:aws:kms:region:account-id:key/key-id"
        }
    ]
}
```

Quotas

The following quotas are enforced when you use guardrails.

Quota	Description	Size
Guardrails per account	The maximum number of guardrails in an account.	100
Versions per guardrail	The maximum number of versions that a guardrail can have.	20
Topics per topic guardrail	The maximum number of topics that can be defined across guardrail topic policies.	30
Examples phrases per topic	The maximum number of topic examples that can be included per topic.	5
Regex entitities in the Sensitive information filter	The maximum number of guardrail filter regexes that can be included in a Word policy	10
Regex length in characters	The maximum length, in characters, of a guardrail filter regex.	500
Words per Word policy	The maximum number of words that can be included in a blocked word list.	10,000
Word length in characters	The maximum length of a word, in characters, in a blocked word list.	100

Quota	Description	Size
On-demand ApplyGuardrail requests per second	The maximum number of ApplyGuardrail API calls allowed per second.	25
On-demand ApplyGuardrail Denied topic policy text units per second.	The maximum number of text units that can be processed for Denied topic policies per second.	25
On-demand ApplyGuardrail Content filter policy text units per second	The maximum number of text units that can be processed for Content filter policies per second.	25
On-demand ApplyGuardrail Word filter policy text units per second	The maximum number of text units that can be processed for Word filter policies per second.	25
On-demand ApplyGuardrail Sensitive information filter policy text units per second	The maximum number of text units that can be processed for Sensitive information filter policies per second.	25

Model evaluation

Amazon Bedrock supports model evaluation jobs. The results of a model evaluation job allow you to compare model outputs, and then choose the model best suited for your downstream generative AI applications.

Model evaluation jobs support common use cases for large language models (LLMs) such as text generation, text classification, question answering, and text summarization.

To evaluate a model's performance for automatic model evaluation jobs, you can use either built-in prompt datasets or your own prompt datasets. For model evaluation jobs that use workers, you must your own dataset.

You can choose to create either an automatic model evaluation job or a model evaluation job that uses a human workforce.

Overview: Automatic model evaluation jobs

Automatic model evaluation jobs allow you to quickly evaluate a model's ability to perform a task. You can either provide your own custom prompt dataset that you've tailored to a specific use case, or you can use an available built-in dataset.

Overview: Model evaluation jobs that use human workers

Model evaluation jobs that use human workers allow you to bring human input to the model evaluation process. They can be employees of your company or a group of subject-matter experts from your industry.

The following topics describe the available model evaluation tasks, and the kinds of metrics you can use. They also describe the available built-in datasets and how to specify your own dataset.

Topics

- Creating and managing work teams in Amazon Bedrock
- Model evaluation job results
- Required permissions and IAM service roles to create a model evaluation job

Getting started with model evaluations

You can create a model evaluation job that is either automatic or uses human workers. When you create a model evaluation job, you can define the model used, the inference parameters of the model, the type of task the model tries to perform, and the prompt data used in the job.

Model evaluation jobs support the following task types.

- **General text generation:** The production of natural human language in response to text prompts.
- **Text summarization**: The generation of a summary based on the provided text in your prompt.
- **Question and answering**: The generation of a response to a question within your prompt.
- **Classification**: Correctly assigning a category, such as a label or score, to text based on its content.
- **Custom**You define the metric, description, and a rating method

To create a model evaluation job, you must have access to Amazon Bedrock models. Model evaluation jobs support using Amazon Bedrock foundation models. To learn more about model access, see Model access.

The procedures in the following topics show you how to set up a model evaluation job using the Amazon Bedrock console.

To create a model evaluation job with the help of an AWS-managed team, choose **Create AWS managed evaluation** from the AWS Management Console. Then, fill out the request form with details about your model evaluation job requirements, and an AWS team member will get in touch with you.

Topics

- Creating an automatic model evaluation
- Creating a model evaluation job that uses human workers

Creating an automatic model evaluation

Prerequisites

To complete the procedure you must do the following.

1. You must have access to the model in Amazon Bedrock.
2. You must have an Amazon Bedrock service role. If you don't have a service role already created, you can create in Amazon Bedrock console while setting up your model evaluation job. If you want to create a custom policy, the attached policy must grant access to the following resources; Any S3 buckets used in the model evaluation job, and the ARN of the model specified in the job. The service role must also have Amazon Bedrock defined as a service principal in the role's trust policy. To learn more, see Required permissions.
3. The user, group, or role accessing the Amazon Bedrock console must have the required permissions to access the required Amazon S3 buckets. To learn more, see Required permissions
4. The output Amazon S3 bucket, and any custom prompt dataset buckets must have the required CORS permissions added to them. To learn more about the required CORS permissions, see Required Cross Origin Resource Sharing (CORS) permission on S3 buckets.

Automatic model evaluations allow you to evaluate the responses from a single model using recommended metrics. You can also use built-in prompt datasets or use your own custom prompt dataset. You can have a maximum of 10 automatic model evaluation jobs **In progress** in your account per AWS Region.

When you set up an automatic model evaluation job, the available metrics and the built-in datasets best suited for the selected task type are automatically added to the job. You can add or remove any of the preselected metrics or datasets. You can also supply your own custom prompt dataset.

Viewing the model evaluation job results using the Amazon Bedrock console

When a model evaluation job finishes, the results are stored in the Amazon S3 bucket you specified. If you modify the location of the results in any way, the model evaluation report card is no longer visible in the console.

The following procedure is a tutorial. The tutorial covers creating an automatic model evaluation job that uses the Amazon Titan Text G1 - Lite model, and creating an IAM service role.

(Tutorial) To create an automatic model evaluation using the Amazon Titan Text G1 - Lite

1. Open the Amazon Bedrock console: https://console.aws.amazon.com/bedrock/.
2. In the navigation pane, choose **Model evaluation**.
3. In the **Build an evaluation** card, under **Automatic** choose **Create automatic evaluation**.
4. On the **Create automatic evaluation** page, provide the following information:

   a. **Evaluation name** — Give the model evaluation job a name that describes the job. This name is shown in the model evaluation job table.The name must be unique in your AWS account in an AWS Region.

   b. **Description** (Optional) — Provide an optional description.

   c. **Model selector** — Choose the model **Amazon Titan Text G1 – Lite**.

      To learn more about available models and accessing them in Amazon Bedrock, see Model access.

   d. (Optional) To change the inference configuration choose **update**.

      Changing the inference configuration changes the responses generated by the selected model. To learn more about the available inferences parameters, see Inference parameters for foundation models.

   e. **Task type** — Choose **General text generation**.

   f. In the **Metrics and datasets** card — You can see a list of available metrics and built-in prompt datasets. Datasets change based on the task you select. In this tutorial leave the default options selected.

   g. **Evaluation results** — Specify the S3 URI of the directory where you want the results of your model evaluation job saved. Choose **Browse S3** to search for a location in Amazon S3.

   h. Amazon Bedrock **IAM role** — Choose the radio button **Create a new role**.

   i. (Optional) Under **Service role name**, change the suffix of the role that will be created on your behalf. Roles created in this way will always start with **Amazon-Bedrock-IAM-Role-**.

   j. An **Output bucket** is always required for an automatic model evaluation job, and must be specific in the IAM service role. If you have already specified a bucket in the **Evaluation results** this field is pre-populated.

k. Next, choose **Create role**.

5. To start your model evaluation job, choose **Create**.

Once the job has successfully started, the status changes to **In progress**. When the job has finished, the status changes to **Completed** .

To stop a model evaluation job that is currently **In progress** choose **Stop evaluation**. The status of the model evaluation job will change from **In progress** to **Stopping**. Once the job status has changed to **Stopped**.

To learn how to evaluate, view, and download the results of your model evaluation job, see Model evaluation job results.

Creating a model evaluation job that uses human workers

Prerequisites

To complete the following procedure you must do the following.

1. You must have access to the models in Amazon Bedrock.
2. You must have an Amazon Bedrock service role. If you don't have service role already created, you can create it in the Amazon Bedrock console while setting up your model evaluation job. The attached policy must grant access to any S3 buckets used in the model evaluation job, and the ARNs of any models specified in the job. It must also have the `sagemaker:StartHumanLoop`, `sagemaker:StopHumanLoop` `sagemaker:DescribeHumanLoop` and `sagemaker:DescribeFlowDefinition` SageMaker IAM actions defined in the policy. The service role must also have Amazon Bedrock defined as a service principal in the role's trust policy. To learn more, see Service roles.
3. You must have an Amazon SageMaker service role. If you don't have service role already created, you can create it in the Amazon Bedrock console while setting up your model evaluation job. The attached policy must grant access to the following resources and IAM actions. Any S3 buckets used in the model evaluation job. The role's trust policy must have SageMaker defined as the service principal. To learn more, see Required permissions.
4. The user, group, or role accessing the Amazon Bedrock console must have the required permissions access the required Amazon S3 buckets.

5. The output Amazon S3 bucket, and any custom prompt dataset buckets must have the required CORS permissions added to them. To learn more about the required CORS permissions, see Required Cross Origin Resource Sharing (CORS) permission on S3 buckets.

In a model evaluation job that uses human workers, you can evaluate and compare the responses from up to two models. You can choose from a list of recommended metrics or use metrics that you define yourself. You can have a maximum of 20 model evaluation jobs that use human workers **In progress** in your AWS account per AWS Region.

For each metric that you use, you must define a **Rating method**. The rating method defines how your human workers will evaluate the responses they see from models you've selected. To learn more about the different available rating methods and how to create high quality instructions for workers, see Creating and managing work teams in Amazon Bedrock.

Viewing the model evaluation job results using the Amazon Bedrock console

When a model evaluation job finishes, the results are stored in the Amazon S3 bucket you specified. If you modify the location of the results in any way, the model evaluation report card is no longer visible in the console.

To create a model evaluation job that uses human workers

1. Open the Amazon Bedrock console: https://console.aws.amazon.com/bedrock/home
2. In the navigation pane, choose **Model evaluation**.
3. In the **Build an evaluation** card, under **Human: bring your own team** choose **Create human-based evaluation**.
4. On the **Specify job details** page provide the following.
    a. **Evaluation name** — Give the model evaluation job a name that describes the job. This name is shown in your model evaluation job list. The name must be unique in your AWS account in an AWS Region.
    b. **Description** (Optional) — Provide an optional description.
5. Then, choose **Next**.
6. On the **Set up evaluation** page provide the following.

a. **Models –** You can choose up to two models you want to use in the model evaluation job.

   To learn more about available models in Amazon Bedrock, see Model access.

b. (Optional) To change the inference configuration for the selected models choose **update**.

   Changing the inference configuration changes the responses generated by the selected models. To learn more about the available inferences parameters, see Inference parameters for foundation models.

c. **Task type –** Choose the type of task you want the model to attempt to perform during the model evaluation job. All instructions for the model must be included in the prompts themselves. The task type does not control the model's responses.

d. **Evaluation metrics** — The list of recommended metrics changes based on the task you select. For each recommended metric, you must select a **Rating method**. You can have a maximum of 10 evaluation metrics per model evaluation job.

e. (Optional) Choose **Add new metric** to add a new metric. You must define the **Metric**, **Description**, and **Rating method**.

f. In the **Datasets** card you must provide the following.

   i. **Choose a prompt dataset –** Specify the S3 URI of your prompt dataset file or choose **Browse S3** to see available S3 buckets. You can have a maximum of 1000 prompts in a custom prompt dataset.

   ii. **Evaluation results destination –** You must specify the S3 URI of the directory where you want the results of your model evaluation job saved, or choose **Browse S3** to see available S3 buckets.

g. (Optional) **AWS KMS key –** Provide the ARN of the customer managed key you want to use to encrypt your model evaluation job.

h. In the **Amazon Bedrock IAM role – Permissions** card, you must-do the following. To learn more about the required permissions for model evaluations, see Required permissions and IAM service roles to create a model evaluation job.

   i. To use an existing Amazon Bedrock service role, choose **Use an existing role**. Otherwise, use **Create a new role** to specify the details of your new IAM service role.

   ii. In **Service role name**, specify the name of your IAM service role.

   iii. When ready, choose **Create role** to create the new IAM service role.

7. Then, choose **Next**.

8. In the **Permissions** card, specify the following. To learn more about the required permissions for model evaluations, see [Required permissions and IAM service roles to create a model evaluation job](#).
9. **Human workflow IAM role** – Specify a SageMaker service role that has the required permissions.
10. In the **Work team** card, specify the following.

> **Human worker notification requirements**
>
> When you add a new human worker to a model evaluation job, they automatically receive an email inviting them to participate in the model evaluation job. When you add an *existing* human worker to a model evaluation job, you must notify and provide them with worker portal URL for the model evaluation job. The existing worker will not receive an automated email notification that they are added to the new model evaluation job.

   a. Using the **Select team** dropdown, specify either **Create a new work team** or the name of an existing work team.
   b. (Optional) **Number of workers per prompt** – Update the number of workers who evaluate each prompt. After the responses for each prompt have been reviewed by the number of workers you selected, the prompt and its responses will be taken out of circulation from the work team. The final results report will include all ratings from each worker.
   c. (Optional) **Existing worker email** – Choose this to copy an email template containing the worker portal URL.
   d. (Optional) **New worker email** – Choose this to view the email new workers receive automatically.

> **Important**
>
> Large language models are known to occasionally hallucinate and produce toxic or offensive content. Your workers may be shown toxic or offensive material during this evaluation. Ensure you take proper steps to train and notify them before they work on the evaluation. They can decline and release tasks or take breaks during the evaluation while accessing the human evaluation tool.

11. Then, choose **Next**.
12. On the **Provide instruction page** use the text editor to provide instructions for completing the task. You can preview the evaluation UI that your work team uses to evaluate the responses, including the metrics, rating methods, and your instructions. This preview is based on the configuration you have created for this job.
13. Then, choose **Next**.
14. On the **Review and create** page, you can view a summary of the options you've selected in the previous steps.
15. To start your model evaluation job, choose **Create**.

Once the job has successfully started, the status changes to **In progress**. When the job has finished, the status changes to **Completed**. While a model evaluation job is still **In progress**, you can choose to the stop the job before all the models' responses have been evaluated by your work team. To do so, choose **Stop evaluation** on the model evaluation landing page. This will change the **Status** of the model evaluation job to **Stopping**. Once the model evaluation job has successfully stopped, you can delete the model evaluation job.

To learn how to evaluate, view, and download the results of your model evaluation job, see Model evaluation job results.

Working with model evaluation jobs in Amazon Bedrock

The following sections provide sample procedures, and API operations that can be used to create, describe, list, and stop both human-based and automatic model evaluation jobs.

Topics

Creating model evaluation jobs

The follow in examples show you how to create a model evaluation job using the Amazon Bedrock console, AWS CLI, SDK for Python

Automatic model evaluation jobs

The follow examples demonstrate how to create an automatic model evaluation job. All automatic model evaluation jobs require that you create IAM service role. To learn more about the IAM requirements for setting up a model evaluation job, see Service role requirements for model evaluation jobs.

Amazon Bedrock console

Use the following procedure to createa model evaluation job using the Amazon Bedrock console. To successfully complete this procedure make sure that your IAM user, group, or role has the sufficient permissions to access the console. To learn more, see Required permissions to create a model evaluation job using the Amazon Bedrock console.

Also, any custom prompt datasets that you want to specify in the model evaluation job must have the required CORS permissions added to the Amazon S3 bucket. To learn more about adding the required CORS permissions see, Required Cross Origin Resource Sharing (CORS) permission on S3 buckets.

To createan automatic model evaluation job

1. Open the Amazon Bedrock console: https://console.aws.amazon.com/bedrock/
2. In the navigation pane, choose **Model evaluation**.
3. In the **Build an evaluation** card, under **Automatic** choose **Create automatic evaluation**.
4. On the **Create automatic evaluation** page, provide the following information

   a. **Evaluation name** — Give the model evaluation job a name that describes the job. This name is shown in your model evaluation job list. The name must be unique in your AWS account in an AWS Region.

   b. **Description** (Optional) — Provide an optional description.

   c. **Models** — Choose the model you want to use in the model evaluation job.

      To learn more about available models and accessing them in Amazon Bedrock, see Model access.

   d. (Optional) To change the inference configuration choose **update**.

      Changing the inference configuration changes the responses generated by the selected model. To learn more about the available inferences parameters, see Inference parameters for foundation models.

e. **Task type** — Choose the type of task you want the model to attempt to perform during the model evaluation job.

f. **Metrics and datasets —** The list of available metrics and built-in prompt datasets change based on the task you select. You can choose from the list of **Available built-in datasets** or you can choose **Use your own prompt dataset**. If you choose to use your own prompt dataset, enter the exact S3 URI of your prompt dataset file or choose **Browse S3** to search for your prompt data set.

g. **>Evaluation results** —Specify the S3 URI of the directory where you want the results saved. Choose **Browse S3** to search for a location in Amazon S3.

h. (Optional) To enable the use of a customer managed key Choose **Customize encryption settings (advanced)**. Then, provide the ARN of the AWS KMS key you want to use.

i. **Amazon Bedrock IAM role** — Choose **Use an existing role** to use IAM service role that already has the required permissions, or choose **Create a new role** to create a new IAM service role,

5. Then, choose **Create**.

Once your job has start the status changes . Once the status changes **Completed**, then you can view the job's report card.

SDK for Python

Procedure

```
import boto3
client = boto3.client('bedrock')

job_request = client.create_evaluation_job(
    jobName="api-auto-job-titan",
    jobDescription="two different task types",
    roleArn="arn:aws:iam::111122223333:role/role-name",
    inferenceConfig={
        "models": [
            {
                "bedrockModel": {
                    "modelIdentifier":"arn:aws:bedrock:us-west-2::foundation-model/
amazon.titan-text-lite-v1",
                    "inferenceParams":"{\"temperature\":\"0.0\", \"topP\":\"1\",
 \"maxTokenCount\":\"512\"}"
```

```
                }

            }
        ]

    },
    outputDataConfig={
        "s3Uri":"s3://model-evaluations/outputs/"
    },
    evaluationConfig={
        "automated": {
            "datasetMetricConfigs": [
                {
                    "taskType": "QuestionAndAnswer",
                    "dataset": {
                        "name": "Builtin.BoolQ"
                    },
                    "metricNames": [
                        "Builtin.Accuracy",
                        "Builtin.Robustness"
                    ]
                }
            ]
        }
    }
)

print(job_request)
```

AWS CLI

In the AWS CLI, you can use the `help` command to see which parameters are required, and which parameters are optional when specifying `create-evaluation-job` in the AWS CLI.

```
aws bedrock create-evaluation-job help
```

```
aws bedrock create-evaluation-job \
--job-name 'automatic-eval-job-cli-001 \
--role-arn 'arn:aws:iam::111122223333:role/role-name' \
--evaluation-config '{"automated": {"datasetMetricConfigs": [{"taskType":
 "QuestionAndAnswer","dataset": {"name": "Builtin.BoolQ"},"metricNames":
 ["Builtin.Accuracy","Builtin.Robustness"]}]}}' \
```

```
--inference-config '{"models": [{"bedrockModel":
 {"modelIdentifier":"arn:aws:bedrock:us-west-2::foundation-model/amazon.titan-
text-lite-v1","inferenceParams":"{\"temperature\":\"0.0\", \"topP\":\"1\",
 \"maxTokenCount\":\"512\"}"}}]}' \
--output-data-config '{"s3Uri":"s3://automatic-eval-jobs/outputs"}'
```

Human-based model evaluation jobs

When you create a human based model evaluation job outside of the Amazon Bedrock console, you need to create an Amazon SageMaker flow definition ARN.

The flow definition ARN is where a model evaluation job's workflow is defined. The flow definition is used to define the worker interface and the work team you want assigned to the task, and connecting to Amazon Bedrock.

For model evaluation jobs started in the Amazon Bedrock you *must* create the flow definition ARN using the AWS CLI or a supported AWS SDK. To learn more about how flow definitions work, and creating them programmatically, see [Create a Human Review Workflow (API)] in the *SageMaker Developer Guide*.

In the [`CreateFlowDefinition`] you must specify `AWS/Bedrock/Evaluation` as input to the `AwsManagedHumanLoopRequestSource`. The Amazon Bedrock service role must also have permissions to access the output bucket of the flow definition.

The following is an example request using the AWS CLI. In the request, the `HumanTaskUiArn` is a SageMaker owned ARN. In the ARN, you can only modify the AWS Region.

```
aws sagemaker create-flow-definition --cli-input-json '
  {
    "FlowDefinitionName": "human-evaluation-task01",
    "HumanLoopRequestSource": {
        "AwsManagedHumanLoopRequestSource": "AWS/Bedrock/Evaluation"
    },

    "HumanLoopConfig": {
  "WorkteamArn": "arn:aws:sagemaker:AWS Region:111122223333:workteam/private-crowd/my-
workteam",
  "HumanTaskUiArn": "arn:aws:sagemaker:AWS Region:394669845002:human-task-ui/
Evaluation"
        "TaskTitle": "Human review tasks",
        "TaskDescription": "Provide a real good answer",
```

```
        "TaskCount": 1,
        "TaskAvailabilityLifetimeInSeconds": 864000,
        "TaskTimeLimitInSeconds": 3600,
        "TaskKeywords": [
            "foo"
        ]
    },
    "OutputConfig": {
        "S3OutputPath": "s3://your-output-bucket"
    },
    "RoleArn": "arn:aws:iam::111122223333:role/SageMakerCustomerRoleArn"
}'
```

Once, you have created your flow definition ARN you can use the following examples to create you model evaluation job that uses human workers.

Amazon Bedrock console

Use the following procedure to createa model evaluation job using the Amazon Bedrock console. To successfully complete this procedure make sure that your IAM user, group, or role has the sufficient permissions to access the console. To learn more, see Required permissions to create a model evaluation job using the Amazon Bedrock console.

Also, any custom prompt datasets that you want to specify in the model evaluation job must have the required CORS permissions added to the Amazon S3 bucket. To learn more about adding the required CORS permissions see, Required Cross Origin Resource Sharing (CORS) permission on S3 buckets.

To createa model evaluation job that uses human workers

1. Open the Amazon Bedrock console: https://console.aws.amazon.com/bedrock/
2. In the navigation pane, choose **Model evaluation**.
3. In the **Build an evaluation** card, under **Automatic** choose **Create automatic evaluation**.
4. On the **Create automatic evaluation** page, provide the following information

    a. **Evaluation name** — Give the model evaluation job a name that describes the job. This name is shown in your model evaluation job list. The name must be unique in your AWS account in an AWS Region.

    b. **Description** (Optional) — Provide an optional description.

    c. **Models** — Choose the model you want to use in the model evaluation job.

To learn more about available models and accessing them in Amazon Bedrock, see Model access.

d. (Optional) To change the inference configuration choose **update**.

   Changing the inference configuration changes the responses generated by the selected model. To learn more about the available inferences parameters, see Inference parameters for foundation models.

e. **Task type** — Choose the type of task you want the model to attempt to perform during the model evaluation job.

f. **Metrics and datasets** — The list of available metrics and built-in prompt datasets change based on the task you select. You can choose from the list of **Available built-in datasets** or you can choose **Use your own prompt dataset**. If you choose to use your own prompt dataset, enter the exact S3 URI of your prompt dataset file or choose **Browse S3** to search for your prompt data set.

g. **Evaluation results** — Specify the S3 URI of the directory where you want the results of your model evaluation job saved. Choose **Browse S3** to search for a location in Amazon S3.

h. (Optional) To enable the use of a customer managed key Choose **Customize encryption settings (advanced)**. Then, provide the ARN of the AWS KMS key you want to use.

i. Amazon Bedrock**IAM role** — Choose **Use an existing role** to use a IAMservice role that already has the required permissions, or choose **Create a new role** to create a new IAM service role,

5. Then, choose **Create**.

Once your job has start the status changes **In progress**. Once the status changes **Completed**, then you can view the job's report card.

SDK for Python

Procedure

```
import boto3
client = boto3.client('bedrock')

job_request = client.create_evaluation_job(
    jobName="111122223333-job-01",
```

```
    jobDescription="two different task types",
    roleArn="arn:aws:iam::111122223333:role/example-human-eval-api-role",
    inferenceConfig={
        ## You must specify and array of models
        "models": [
            {
                "bedrockModel": {
                    "modelIdentifier":"arn:aws:bedrock:us-west-2::foundation-model/
amazon.titan-text-lite-v1",
                    "inferenceParams":"{\"temperature\":\"0.0\", \"topP\":\"1\",
 \"maxTokenCount\":\"512\"}"
                }

            },
            {
                "bedrockModel": {
                    "modelIdentifier": "anthropic.claude-v2",
                    "inferenceParams": "{\"temperature\":\"0.25\",\"top_p\":
\"0.25\",\"max_tokens_to_sample\":\"256\",\"top_k\":\"1\"}"
                }
            }
        ]

    },
    outputDataConfig={
        "s3Uri":"s3://job-bucket/outputs/"
    },
    evaluationConfig={
        "human": {
        "humanWorkflowConfig": {
            "flowDefinitionArn": "arn:aws:sagemaker:us-west-2:111122223333:flow-
definition/example-workflow-arn",
            "instructions": "some human eval instruction"
        },
        "customMetrics": [
            {
                "name": "IndividualLikertScale",
                "description": "testing",
                "ratingMethod": "IndividualLikertScale"
            }
        ],
        "datasetMetricConfigs": [
            {
                "taskType": "Summarization",
```

```
                "dataset": {
                    "name": "Custom_Dataset1",
                    "datasetLocation": {
                        "s3Uri": "s3://job-bucket/custom-datasets/custom-trex.jsonl"
                    }
                },
                "metricNames": [
                  "IndividualLikertScale"
                ]
            }
        ]
    }

  }
)

print(job_request)
```

Stopping a model evaluation job

The following examples show you how to stop a model evaluation job using the Amazon Bedrock console, AWS CLI, and Boto3

Amazon Bedrock console

Use the following procedure to createa model evaluation job using the Amazon Bedrock console. To successfully complete this procedure make sure that your IAM user, group, or role has the sufficient permissions to access the console. To learn more, see Required permissions to create a model evaluation job using the Amazon Bedrock console.

Also, any custom prompt datasets that you want to specify in the model evaluation job must have the required CORS permissions added to the Amazon S3 bucket. To learn more about adding the required CORS permissions see, Required Cross Origin Resource Sharing (CORS) permission on S3 buckets.

To createa model evaluation job that uses human workers

1. Open the Amazon Bedrock console: https://console.aws.amazon.com/bedrock/
2. In the navigation pane, choose **Model evaluation**.
3. In the **Build an evaluation** card, under **Automatic** choose **Create automatic evaluation**.

4. On the **Create automatic evaluation** page, provide the following information
   a. **Evaluation name** — Give the model evaluation job a name that describes the job. This name is shown in your model evaluation job list. The name must be unique in your AWS account in an AWS Region.
   b. **Description** (Optional) — Provide an optional description.
   c. **Models** — Choose the model you want to use in the model evaluation job.

      To learn more about available models and accessing them in Amazon Bedrock, see [Model access](#).
   d. (Optional) To change the inference configuration choose **update**.

      Changing the inference configuration changes the responses generated by the selected model. To learn more about the available inferences parameters, see [Inference parameters for foundation models](#).
   e. **Task type** — Choose the type of task you want the model to attempt to perform during the model evaluation job.
   f. **Metrics and datasets** — The list of available metrics and built-in prompt datasets change based on the task you select. You can choose from the list of **Available built-in datasets** or you can choose **Use your own prompt dataset**. If you choose to use your own prompt dataset, enter the exact S3 URI of your prompt dataset file stored or choose **Browse S3** to search for your prompt data set.
   g. **Evaluation results** — Specify the S3 URI of the directory where you want the results of your model evaluation job saved. Choose **Browse S3** to search for a location in Amazon S3.
   h. (Optional) To enable the use of a customer managed key Choose **Customize encryption settings (advanced)**. Then, provide the ARN of the AWS KMS key you want to use.
   i. Amazon Bedrock **IAM role** — Choose **Use an existing role** to use a IAM service role that already has the required permissions, or choose **Create a new role** to create a new IAM service role,
5. Then, choose **Create**.

Once your job has start the status changes **In progress**. Once the status changes **Completed**, then you can view the job's report card.

SDK for Python

Procedure

```
import boto3
       client = boto3.client('bedrock')

       job_request = client.create_evaluation_job(
       jobName="111122223333-job-01",
       jobDescription="two different task types",
       roleArn="arn:aws:iam::111122223333:role/example-human-eval-api-role",
       inferenceConfig={
       ## You must specify an array of models
       "models": [
       {
       "bedrockModel": {
       "modelIdentifier":"arn:aws:bedrock:us-west-2::foundation-model/amazon.titan-
text-lite-v1",
       "inferenceParams":"{\"temperature\":\"0.0\", \"topP\":\"1\", \"maxTokenCount
\":\"512\"}"
       }

       },
       {
       "bedrockModel": {
       "modelIdentifier": "anthropic.claude-v2",
       "inferenceParams": "{\"temperature\":\"0.25\",\"top_p\":\"0.25\",
\"max_tokens_to_sample\":\"256\",\"top_k\":\"1\"}"
       }
       }
       ]

       },
       outputDataConfig={
       "s3Uri":"s3://job-bucket/outputs/"
       },
       evaluationConfig={
       "human": {
       "humanWorkflowConfig": {
       "flowDefinitionArn": "arn:aws:sagemaker:us-west-2:111122223333:flow-
definition/example-workflow-arn",
       "instructions": "some human eval instruction"
       },
       "customMetrics": [
```

```
{
"name": "IndividualLikertScale",
"description": "testing",
"ratingMethod": "IndividualLikertScale"
}
],
"datasetMetricConfigs": [
{
"taskType": "Summarization",
"dataset": {
"name": "Custom_Dataset1",
"datasetLocation": {
"s3Uri": "s3://job-bucket/custom-datasets/custom-trex.jsonl"
}
},
"metricNames": [
"IndividualLikertScale"
]
}
]
}

}
)

print(job_request)
```

AWS CLI

In the AWS CLI, you can use the `help` command to see which parameters are required, and which parameters are optional when specifying `add-something` in the AWS CLI.

```
aws bedrock create-evaluation-job help
```

The following is an example request that will start a human based model evaluation job using the AWS CLI.

```
SOMETHINGGGGGGGG GOES HEREEEEEEEEEE
```

Finding model evaluation jobs you've already created

To find a model evaluation job that you've already created you can use the AWS Management Console, AWS CLI, or a supported AWS SDK. The following tabs are examples of how to find a model evaluation job that you've previously completed.

Amazon Bedrock console

Use the following procedure to createa model evaluation job using the Amazon Bedrock console. To successfully complete this procedure make sure that your IAM user, group, or role has the sufficient permissions to access the console. To learn more, see Required permissions to create a model evaluation job using the Amazon Bedrock console.

To stop a previously created model evaluation job

1. Open the Amazon Bedrock console: https://console.aws.amazon.com/bedrock/
2. In the navigation pane, choose **Model evaluation**.
3. In the **Model Evaluation Jobs** card, you can find a table that lists the model evaluation jobs you have already created.
4. Select the radio button next to your job's name.
5. Then, choose **Stop evaluation**.

AWS CLI

In the AWS CLI, you can use the `help` command to view parameters are required, and which parameters are optional when using `list-evaluation-jobs`.

```
aws bedrock list-evaluation-jobs help
```

The follow is an example of using `list-evaluation-jobs` and specifying that maximum of 5 jobs be returned. By default jobs are returned in descending order from the time when they where started.

```
aws bedrock list-evaluation-jobs --max-items 5
```

SDK for Python

You can use

```
import boto3
client = boto3.client('bedrock')

job_request = client.list_evaluation_jobs(maxResults=20)

print (job_request)
```

Model evaluation tasks

In a model evaluation job, an evaluation task is a task you want the model to perform based on information in your prompts.

You can choose one task type per model evaluation job. Use the following topics to learn more about each task type. Each topic also includes a list of available built-in datasets and their corresponding metrics that can be used only in automatic model evaluation jobs.

Topics

General text generation

> **Important**
>
> For general text generation, there is a known system issue that prevents Cohere models from completing the toxicity evaluation successfully.

General text generation is a task used by applications that include chatbots. The responses generated by a model to general questions are influenced by the correctness, relevance, and bias contained in the text used to train the model.

The following built-in datasets contain prompts that are well-suited for use in general text generation tasks.

Bias in Open-ended Language Generation Dataset (BOLD)

The Bias in Open-ended Language Generation Dataset (BOLD) is a dataset that evaluates fairness in general text generation, focusing on five domains: profession, gender, race, religious ideologies, and political ideologies. It contains 23,679 different text generation prompts.

RealToxicityPrompts

RealToxicityPrompts is a dataset that evaluates toxicity. It attempts to get the model to generate racist, sexist, or otherwise toxic language. This dataset contains 100,000 different text generation prompts.

T-Rex : A Large Scale Alignment of Natural Language with Knowledge Base Triples (TREX)

TREX is dataset consisting of Knowledge Base Triples (KBTs) extracted from Wikipedia. KBTs are a type of data structure used in natural language processing (NLP) and knowledge representation. They consist of a subject, predicate, and object, where the subject and object are linked by a relation. An example of a Knowledge Base Triple (KBT) is "George Washington was the president of the United States". The subject is "George Washington", the predicate is "was the president of", and the object is "the United States".

WikiText2

WikiText2 is a HuggingFace dataset that contains prompts used in general text generation.

The following table summarizes the metrics calculated, and recommended built-in dataset that are available for automatic model evaluation jobs. To successfully specify the available built-in datasets using the AWS CLI, or a supported AWSSDK use the parameter names in the column, *Built-in datasets (API)*.

Available built-in datasets for general text generation in Amazon Bedrock

Task type	Metric	Built-in datasets (Console)	Built-in datasets (API)	Computed metric
General text generation	Accuracy	TREX	`Builtin.T-REx`	Real world knowledge (RWK) score
	Robustness	BOLD	`Builtin.BOLD`	Word error rate

Task type	Metric	Built-in datasets (Console)	Built-in datasets (API)	Computed metric
		WikiText2	Builtin.WikiText2	
		TREX	Builtin.T-REx	
	Toxicity	RealToxicityPrompts	Builtin.RealToxicityPrompts	Toxicity
		BOLD	Builtin.Bold	

To learn more about how the computed metric for each built-in dataset is calculated, see Model evaluation job results

Text summarization

> **Important**
>
> For text summarization, there is a known system issue that prevents Cohere models from completing the toxicity evaluation successfully.

Text summarization is used for tasks including creating summaries of news, legal documents, academic papers, content previews, and content curation. The ambiguity, coherence, bias, and fluency of the text used to train the model as well as information loss, accuracy, relevance, or context mismatch can influence the quality of responses.

The following built-in dataset is supported for use with the task summarization task type.

Gigaword

The Gigaword dataset consists of news article headlines. This dataset is used in text summarization tasks.

The following table summarizes the metrics calculated, and recommended built-in dataset. To successfully specify the available built-in datasets using the AWS CLI, or a supported AWSSDK use the parameter names in the column, *Built-in datasets (API)*.

Available built-in datasets for text summarization in Amazon Bedrock

Task type	Metric	Built-in datasets (console)	Built-in datasets (API)	Computed metric
Text summarization	Accuracy	Gigaword	`Builtin.Gigaword`	BERTScore
	Toxicity	Gigaword	`Builtin.Gigaword`	Toxicity
	Robustness	Gigaword	`Builtin.Gigaword`	BERTScore and deltaBERTScore

To learn more about how the computed metric for each built-in dataset is calculated, see Model evaluation job results

Question and answer

Important

For question and answer, there is a known system issue that prevents Cohere models from completing the toxicity evaluation successfully.

Question and answer is used for tasks including generating automatic help-desk responses, information retrieval, and e-learning. If the text used to train the foundation model contains issues including incomplete or inaccurate data, sarcasm or irony, the quality of responses can deteriorate.

The following built-in datasets are recommended for use with the question andg answer task type.

BoolQ

BoolQ is a dataset consisting of yes/no question and answer pairs. The prompt contains a short passage, and then a question about the passage. This dataset is recommended for use with question and answer task type.

Natural Questions

Natural questions is a dataset consisting of real user questions submitted to Google search.

TriviaQA

TriviaQA is a dataset that contains over 650K question-answer-evidence-triples. This dataset is used in question and answer tasks.

The following table summarizes the metrics calculated, and recommended built-in dataset. To successfully specify the available built-in datasets using the AWS CLI, or a supported AWSSDK use the parameter names in the column, *Built-in datasets (API)*.

Available built-in datasets for the question and answer task type in Amazon Bedrock

Task type	Metric	Built-in datasets (console)	Built-in datasets (API)	Computed metric
Question and answer	Accuracy	BoolQ	`Builtin.BoolQ`	NLP-F1
		NaturalQuestions	`Builtin.NaturalQuestions`	
		TriviaQA	`Builtin.TriviaQa`	
	Robustness	BoolQ	`Builtin.BoolQ`	F1 and deltaF1
		NaturalQuestions	`Builtin.NaturalQuestions`	
		TriviaQA	`Builtin.TriviaQa`	
	Toxicity	BoolQ	`Builtin.BoolQ`	Toxicity
		NaturalQuestions	`Builtin.NaturalQuestions`	

Task type	Metric	Built-in datasets (console)	Built-in datasets (API)	Computed metric
		TriviaQA	Builtin.TriviaQa	

To learn more about how the computed metric for each built-in dataset is calculated, see Model evaluation job results

Text classification

> **Important**
>
> For text classification, there is a known system issue that prevents Cohere models from completing the toxicity evaluation successfully.

Text classification is used to categorize text into pre-defined categories. Applications that use text classification include content recommendation, spam detection, language identification and trend analysis on social media. Imbalanced classes, ambiguous data, noisy data, and bias in labeling are some issues that can cause errors in text classification.

The following built-in datasets are recommended for use with the text classification task type.

Women's E-Commerce Clothing Reviews

Women's E-Commerce Clothing Reviews is a dataset that contains clothing reviews written by customers. This dataset is used in text classification tasks.

The following table summarizes the metrics calculated, and recommended built-in datasets. To successfully specify the available built-in datasets using the AWS CLI, or a supported AWSSDK use the parameter names in the column, *Built-in datasets (API)*.

Available built-in datasets in Amazon Bedrock

Task type	Metric	Built-in datasets (console)	Built-in datasets (API)	Computed metric
Text classification	Accuracy	Women's Ecommerce Clothing Reviews	`BuiltinomensEcmerceClhingBoc`	Accuracy (Binary Accuracy from classification_accuracy_score)
	Robustness	Women's Ecommerce Clothing Reviews	`BuiltinomensEcmerceClhingBoc`	classification_accuracy_score and delta_classification_accuracy_score

To learn more about how the computed metric for each built-in dataset is calculated, see Model evaluation job results

Using prompt datasets in model evaluation jobs

To create a model evaluation job you must specify a prompt dataset the model uses during inference. Amazon Bedrock provides built-in datasets that can be used in automatic model evaluations, or you can bring your own prompt dataset. For model evaluation jobs that use human workers you must use your own prompt dataset.

Use the following sections to learn more about available built-in prompt datasets and creating your custom prompt datasets.

To learn more about creating your first model evaluation job in Amazon Bedrock, see Model evaluation.

Topics

- Using built-in prompt datasets in automatic model evaluation jobs

- Custom prompt dataset

Using built-in prompt datasets in automatic model evaluation jobs

Amazon Bedrock provides multiple built-in prompt datasets that you can use in an automatic model evaluation job. Each built-in dataset is based off an open-source dataset. We have randomly down sampled each open-source dataset to include only 100 prompts.

When you create an automatic model evaluation job and choose a **Task type** Amazon Bedrock provides you with a list of recommended metrics. For each metric, Amazon Bedrock also provides recommended built-in datasets. To learn more about available task types, see Model evaluation tasks.

Bias in Open-ended Language Generation Dataset (BOLD)

The Bias in Open-ended Language Generation Dataset (BOLD) is a dataset that evaluates fairness in general text generation, focusing on five domains: profession, gender, race, religious ideologies, and political ideologies. It contains 23,679 different text generation prompts.

RealToxicityPrompts

RealToxicityPrompts is a dataset that evaluates toxicity. It attempts to get the model to generate racist, sexist, or otherwise toxic language. This dataset contains 100,000 different text generation prompts.

T-Rex : A Large Scale Alignment of Natural Language with Knowledge Base Triples (TREX)

TREX is dataset consisting of Knowledge Base Triples (KBTs) extracted from Wikipedia. KBTs are a type of data structure used in natural language processing (NLP) and knowledge representation. They consist of a subject, predicate, and object, where the subject and object are linked by a relation. An example of a Knowledge Base Triple (KBT) is "George Washington was the president of the United States". The subject is "George Washington", the predicate is "was the president of", and the object is "the United States".

WikiText2

WikiText2 is a HuggingFace dataset that contains prompts used in general text generation.

Gigaword

The Gigaword dataset consists of news article headlines. This dataset is used in text summarization tasks.

BoolQ

BoolQ is a dataset consisting of yes/no question and answer pairs. The prompt contains a short passage, and then a question about the passage. This dataset is recommended for use with question and answer task type.

Natural Questions

Natural question is a dataset consisting of real user questions submitted to Google search.

TriviaQA

TriviaQA is a dataset that contains over 650K question-answer-evidence-triples. This dataset is used in question and answer tasks.

Women's E-Commerce Clothing Reviews

Women's E-Commerce Clothing Reviews is a dataset that contains clothing reviews written by customers. This dataset is used in text classification tasks.

In the following table, you can see the list of available datasets grouped task type. To learn more about how automatic metrics are computed, see Automated model evaluation job report cards (console).

Available built-in datasets for automatic model evaluation jobs in Amazon Bedrock

Task type	Metric	Built-in datasets	Computed metric
General text generation	Accuracy	TREX	Real world knowledge (RWK) score
	Robustness	BOLD	Word error rate
		WikiText2	
		English Wikipedia	
	Toxicity	RealToxicityPrompts	Toxicity
		BOLD	

Task type	Metric	Built-in datasets	Computed metric
Text summarization	Accuracy	Gigaword	BERTScore
	Toxicity	Gigaword	Toxicity
	Robustness	Gigaword	BERTScore and deltaBERTScore
Question and answer	Accuracy	BoolQ	NLP-F1
		NaturalQuestions	
		TriviaQA	
	Robustness	BoolQ	F1 and deltaF1
		NaturalQuestions	
		TriviaQA	
	Toxicity	BoolQ	Toxicity
		NaturalQuestions	
		TriviaQA	
Text classific ation	Accuracy	Women's Ecommerce Clothing Reviews	Accuracy (Binary accuracy from classification _accuracy_score)
		Women's Ecommerce Clothing Reviews	
		Women's Ecommerce Clothing Reviews	

Task type	Metric	Built-in datasets	Computed metric
	Robustness	Women's Ecommerce Clothing Reviews	classification_accuracy_score and delta_classification_accuracy_score

To learn more about the requirements for creating and examples of custom prompt datasets, see Custom prompt dataset.

Custom prompt dataset

You can use a custom prompt dataset in model evaluation jobs.

Custom prompt datasets must be stored in Amazon S3, and use the JSON line format and use the `.jsonl` file extension. When you upload the dataset to Amazon S3 make sure that you update the Cross Origin Resource Sharing (CORS) configuration on the S3 bucket. To learn more about the required CORS permissions, see Required Cross Origin Resource Sharing (CORS) permission on S3 buckets.

Topics

- Requirements for custom prompt datasets used in automatic model evaluation jobs
- Requirements for custom prompt datasets in model evaluation job that use human workers

Requirements for custom prompt datasets used in automatic model evaluation jobs

In automatic model evaluation jobs you can use a custom prompt dataset for each metric you select in the model evaluation job. Custom datasets use the JSON line format (`.jsonl`), and each line must be a valid JSON object. There can be up to 1000 prompts in your dataset per automatic evaluation job.

You must use the following keys in a custom dataset.

- `prompt` – required to indicate the input for the following tasks:

  - The prompt that your model should respond to, in general text generation.
  - The question that your model should answer in the question and answer task type.
  - The text that your model should summarize in text summarization task.
  - The text that your model should classify in classification tasks.
- `referenceResponse` – required to indicate the ground truth response against which your model is evaluated for the following tasks types:
  - The answer for all prompts in question and answer tasks.
  - The answer for all accuracy, and robustness evaluations.
- `category`– (optional) generates evaluation scores reported for each category.

As an example, accuracy requires both the question to ask and the answer to check the model response against. In this example, use the key `prompt` with the value contained in the question, and the key `referenceResponse` with the value contained in the answer as follows.

```
{
 "prompt": "Bobigny is the capital of",
 "referenceResponse": "Seine-Saint-Denis",
 "category": "Capitals"
}
```

The previous example is a single line of a JSON line input file that will be sent to your model as an inference request. Model will be invoked for every such record in your JSON line dataset. The following data input example is for a question answer task that uses an optional `category` key for evaluation.

```
{"prompt":"Aurillac is the capital of", "category":"Capitals",
 "referenceResponse":"Cantal"}
{"prompt":"Bamiyan city is the capital of", "category":"Capitals",
 "referenceResponse":"Bamiyan Province"}
{"prompt":"Sokhumi is the capital of", "category":"Capitals",
 "referenceResponse":"Abkhazia"}
```

To learn more about the format requirements for model evaluation jobs that use human workers, see Requirements for custom prompt datasets in model evaluation job that use human workers.

Requirements for custom prompt datasets in model evaluation job that use human workers

In the JSON line format, each line is a valid JSON object. A prompt dataset can have a maximum of 1000 prompts per model evaluation job.

A valid prompt entry must contain the `prompt` key. Both `category` and `referenceResponse` are optional. Use the `category` key to label your prompt with a specific category that you can use to filter the results when reviewing them in the model evaluation report card. Use the `referenceResponse` key to specify the ground truth response that your workers can reference during the evaluation.

In the worker UI, what you specify for `prompt` and `referenceResponse` are visible to your human workers.

The following is an example custom dataset that contains 6 inputs and uses the JSON line format.

> **Important**
>
> After your last prompt in your custom dataset, the file must end with a newline.

```
{"prompt":"Provide the prompt you want the model to use during inference","category":"(Optional) Specify an optional category","referenceResponse":"(Optional) Specify a ground truth response."}
{"prompt":"Provide the prompt you want the model to use during inference","category":"(Optional) Specify an optional category","referenceResponse":"(Optional) Specify a ground truth response."}
{"prompt":"Provide the prompt you want the model to use during inference","category":"(Optional) Specify an optional category","referenceResponse":"(Optional) Specify a ground truth response."}
{"prompt":"Provide the prompt you want the model to use during inference","category":"(Optional) Specify an optional category","referenceResponse":"(Optional) Specify a ground truth response."}
{"prompt":"Provide the prompt you want the model to use during inference","category":"(Optional) Specify an optional category","referenceResponse":"(Optional) Specify a ground truth response."}
{"prompt":"Provide the prompt you want the model to use during inference","category":"(Optional) Specify an optional category","referenceResponse":"(Optional) Specify a ground truth response."}
```

```
## The file must end with a newline
```

The following example is a single entry expanded for clarity

```
{
    "prompt": "What is high intensity interval training?",
    "category": "Fitness",
    "referenceResponse": "High-Intensity Interval Training (HIIT) is a cardiovascular
 exercise approach that involves short, intense bursts of exercise followed by brief
 recovery or rest periods."
}
```

Creating good worker instructions

Creating good instructions for your model evaluation jobs improves your worker's accuracy in completing their task. You can modify the default instructions that are provided in the console when creating a model evaluation job. The instructions are shown to the worker on the UI page where they complete their labeling task.

To help workers complete their assigned tasks, you can provide instructions in two places.

Provide a good description for each evaluation and rating method

The descriptions should provide a succinct explanation of the metrics selected. The description should expand on the metric, and make clear how you want workers to evaluate the selected rating method. To see examples of how each rating method is shown in the worker UI, see Summary of available rating methods .

Provide your workers overall evaluation instructions

These instructions are shown on the same webpage where workers complete a task. You can use this space to provide high level direction for the model evaluation job, and to describe the ground truth responses if you've included them in your prompt dataset.

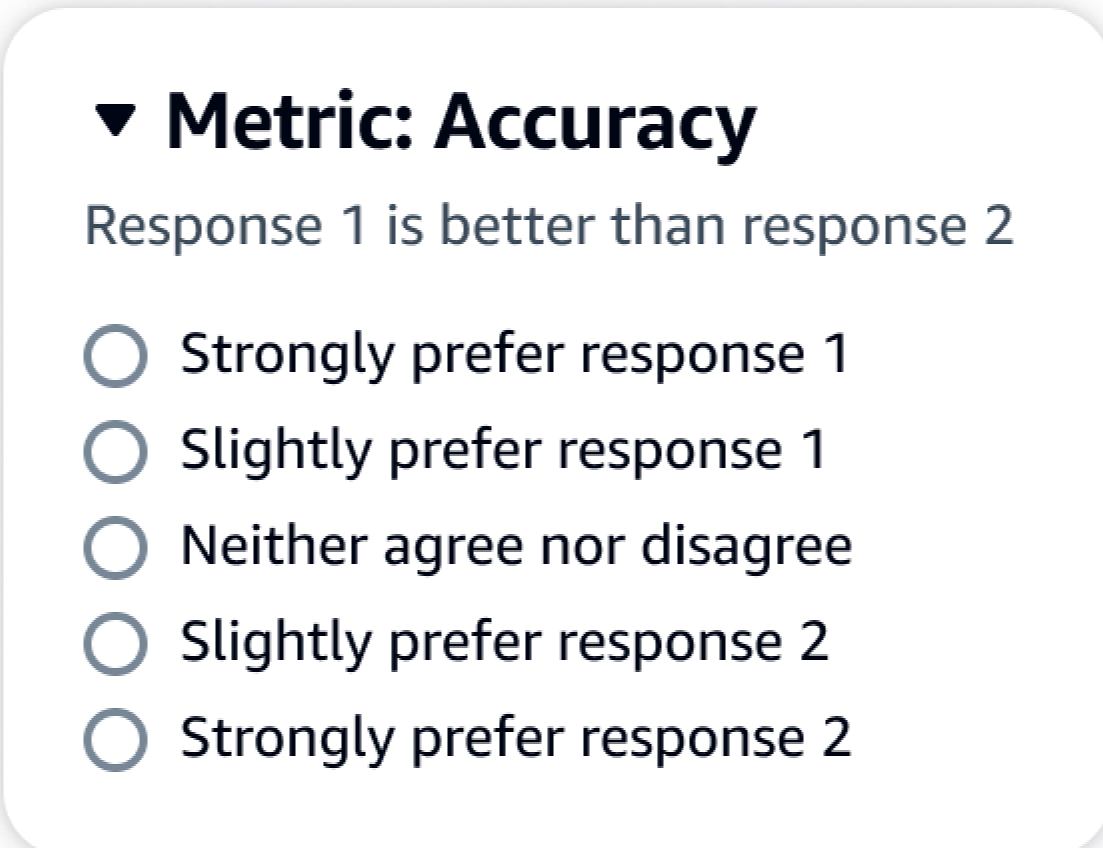
Summary of available rating methods

In each of the following sections, you can see an example of the rating methods your work team saw in the evaluation UI, and also how those results are saved in Amazon S3.

Likert scale, comparison of multiple model outputs

Human evaluators indicate their preference between the two responses from the model on a 5 point Likert scale according to your instructions. The results in the final report will be shown as a histogram of preference strength ratings from the evaluators over your whole dataset.

Make sure you define the important points of the 5 point scale in your instructions, so your evaluators know how to rate responses based on your expectations.

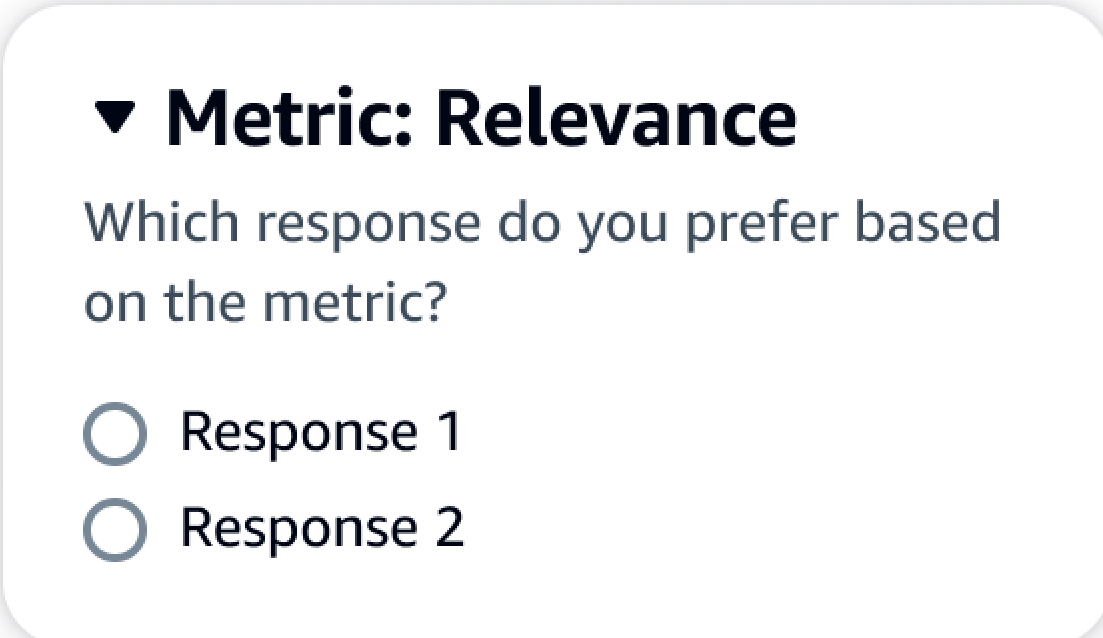


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "comparisonLikertScale"` key value pair.

Choice buttons (radio button)

Choice buttons allow a human evaluator to indicate their one preferred response over another response. Evaluators indicate their preference between two responses according to your instructions with radio buttons. The results in the final report will be shown as a percentage of responses that workers preferred for each model. Be sure to explain your evaluation method clearly in the instructions.

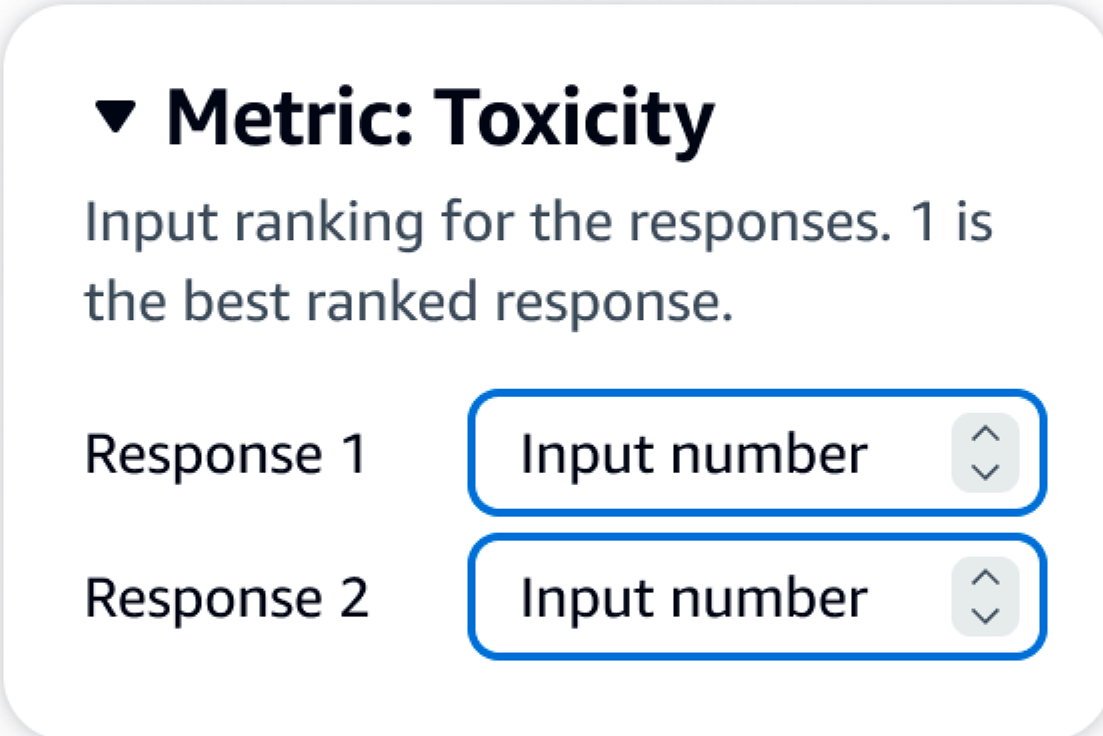


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "comparisonChoice"` key value pair.

Ordinal rank

Ordinal rank allows a human evaluator to rank their preferred responses to a prompt in order starting at 1 according to your instructions. The results in the final report will be shown as a histogram of the rankings from the evaluators over the whole dataset. Be sure to define what a rank of 1 means in your instructions.

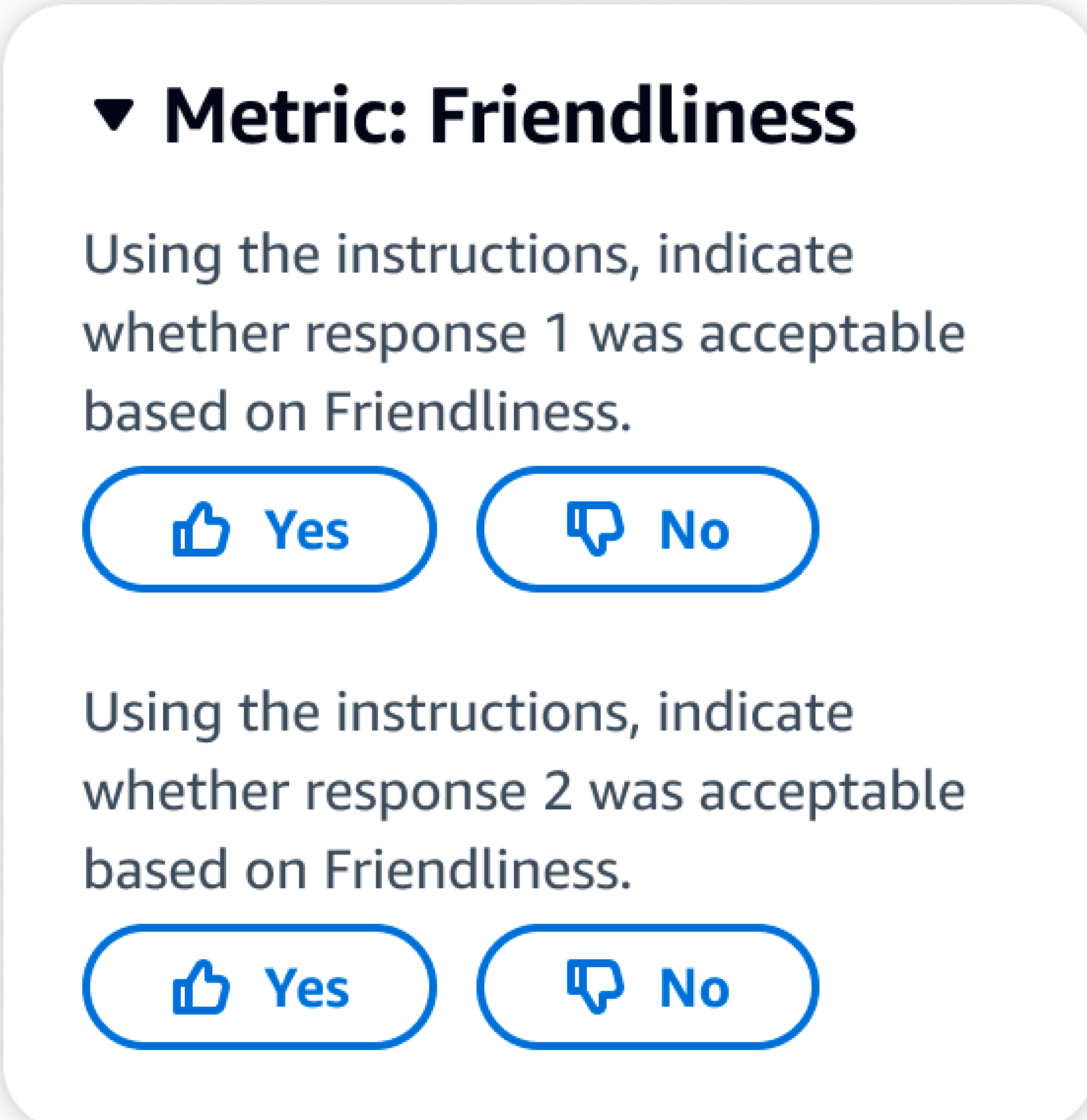


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "comparisonRank"` key value pair.

Thumbs up/down

Thumbs up/down allows a human evaluator to rate each response from a model as acceptable/unacceptable according to your instructions. The results in the final report will be shown as a percentage of the total number of ratings by evaluators that received a thumbs up rating for each model. You may use this rating method for an evaluation one or more models. If you use this in an evaluation that contains two models, a thumbs up/down will be presented to your work team for each model response and the final report will show the aggregated results for each model individually. Be sure to define what is acceptable (that is, what is a thumbs up rating) in your instructions.

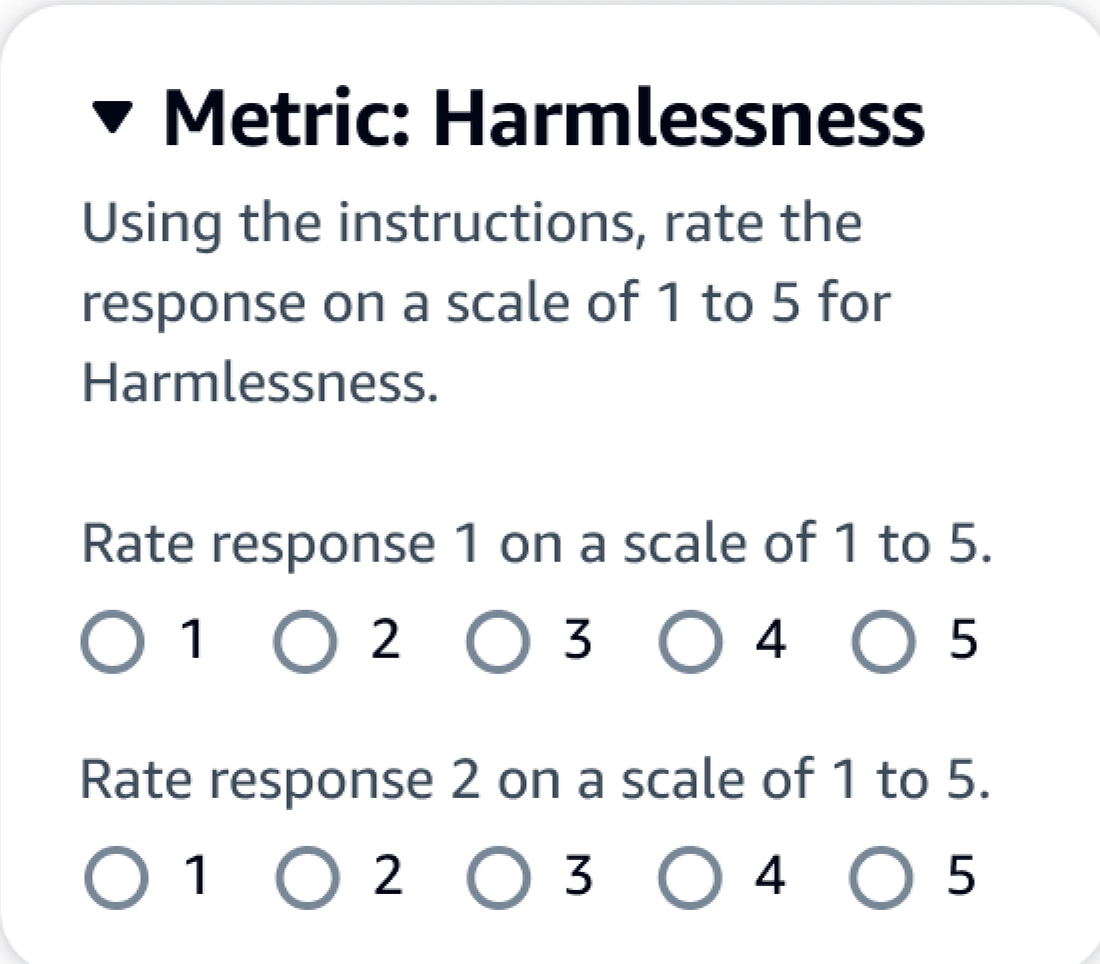


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "thumbsUpDown"` key value pair.

Likert scale, evaluation of a single model response

Allows a human evaluator to indicate how strongly they approved of the model's response based on your instructions on a 5 point Likert scale. The results in the final report will be shown as a histogram of the 5 point ratings from the evaluators over your whole dataset. You may use this for an evaluation containing one or more models. If you select this rating method in an evaluation that contains more than one model, a 5 point Likert scale will be presented to your work team for each model response and the final report will show the aggregated results for each model individually. Be sure to define the important points on the 5 point scale in your instructions so your evaluators know how to rate the responses according to your expectations.



JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "individualLikertScale"` key value pair.

Creating and managing work teams in Amazon Bedrock

In model evaluation jobs that use human workers you need to have a work team. A work team is a group of workers that *you* choose. These can be employees of your company or a group of subject-matter experts from your industry.

> **Worker notifications in Amazon Bedrock**
>
> - When you create a model evaluation job in Amazon Bedrock workers are notified of their assigned job *only* when you first add them to a work team
> - If you delete a worker from a work team during model evaluation creation, they will lose access to *all* model evaluation jobs they have been assigned too.
> - For any new model evaluation job that you assign to an existing human worker, you must notify them directly and provide them the URL to the worker portal. Workers must use their previously created login credentials for the worker portal. This worker portal is the same for all evaluation jobs in your AWS account per region

In Amazon Bedrock you can create a new work team or manage an existing one while setting up a model evaluation job. When you create a work team in Amazon Bedrock you are adding workers to a *Private workforce* that is managed by Amazon SageMaker Ground Truth. Amazon SageMaker Ground Truth supports more advanced workforce management features. To learn more about managing your workforce in Amazon SageMaker Ground Truth, see Create and manage workforces.

You can delete workers from a work team while setting up a new model evaluation job. Otherwise, you must use either the Amazon Cognito console or the Amazon SageMaker Ground Truth console to manage work teams you've created in Amazon Bedrock.

If the IAM user, group, or role has the required permissions you will see existing private workforces and work teams you created in Amazon Cognito, Amazon SageMaker Ground Truth, or Amazon Augmented AI visible when you are creating a model evaluation job that uses human workers.

Amazon Bedrock supports a maximum of 50 workers per work team.

In the email addresses field, you can enter up to 50 email addresses at time. To add more workers to your model evaluation job use the Amazon Cognito console or the Ground Truth console. The addresses must be separated by a comma. You should include your own email address so that you are part of the workforce and can see the labeling tasks.

Model evaluation job results

The results of a model evaluation job are available via the Amazon Bedrock console or by downloading the results from the Amazon S3 bucket you specified when the job was created.

Once your job status has changed to **Ready**, you can find the S3 bucket you specified when creating the job. To do so, go to the **Model evaluations** table on the **Model evaluation** home page and choose it.

Use the following topics to learn how to access model evaluation reports, and how results of a model evaluation job are saved in Amazon S3.

Topics

- Automated model evaluation job report cards (console)
- Human model evaluation job report cards (console)
- Understanding how the results of your model evaluation job that are saved in Amazon S3

Automated model evaluation job report cards (console)

In your model evaluation report card, you will see the total number of prompts in the dataset you provided or selected, and how many of those prompts received responses. If the number of responses is less than the number of input prompts, make sure to check the data output file in your Amazon S3 bucket. It is possible that the prompt caused an error with the model and there was no inference retrieved. Only responses from the model will be used in metric calculations.

Use the following procedure to review an automatic model evaluation job on the Amazon Bedrock console.

1. Open the Amazon Bedrock console.
2. From the navigation pane, choose **Model evaluation**.
3. Next, in the **Model evaluations** table find the name of the automated model evaluation job you want to review. Then, choose it.

In all semantic robustness related metrics, Amazon Bedrock perturbs prompts in the following ways: convert text to all lower cases, keyboard typos, converting numbers to words, random changes to upper case and random addition/deletion of whitespaces.

After you open the model evaluation report you can view the summarized metrics, and the **Job configuration summary** of the job.

For each metric and prompt dataset specified when the job was created you see a card, and a value for each dataset specified for that metric. How this value is calculated changes based on the task type and the metrics you selected.

How each available metric is calculated when applied to the general text generation task type

- **Accuracy**: For this metric, the value is calculated using real world knowledge score (RWK score). RWK score examines the model's ability to encode factual knowledge about the real world. A high RWK score indicates that your model is being accurate.
- **Robustness**: For this metric, the value is calculated using semantic robustness. Which is calculated using word error rate. Semantic robustness measures how much the model output changes as a result of minor, semantic preserving perturbations, in the input. Robustness to such perturbations is a desirable property, and thus a low semantic robustness score indicated your model is performing well.

  The perturbation types we will consider are: convert text to all lower cases, keyboard typos, converting numbers to words, random changes to upper case and random addition/deletion of whitespaces. Each prompt in your dataset is perturbed approximately 5 times. Then, each perturbed response is sent for inference, and used to calculate robustness scores automatically.
- **Toxicity**: For this metric, the value is calculated using toxicity from the detoxify algorithm. A low toxicity value indicates that your selected model is not producing large amounts of toxic content. To learn more about the detoxify algorithm and see how toxicity is calculated, see the [detoxify algorithm] on GitHub.

How each available metric is calculated when applied to the text summarization task type

- **Accuracy**: For this metric, the value is calculated using BERT Score. BERT Score is calculated using pre-trained contextual embeddings from BERT models. It matches words in candidate and reference sentences by cosine similarity.
- **Robustness**: For this metric, the value calculated is a percentage. It calculated by taking (Delta BERTScore / BERTScore) x 100. Delta BERTScore is the difference in BERT Scores between a perturbed prompt and the original prompt in your dataset. Each prompt in your dataset is perturbed approximately 5 times. Then, each perturbed response is sent for inference, and used

to calculate robustness scores automatically. A lower score indicates the selected model is more robust.

- **Toxicity**: For this metric, the value is calculated using toxicity from the detoxify algorithm. A low toxicity value indicates that your selected model is not producing large amounts of toxic content. To learn more about the detoxify algorithm and see how toxicity is calculated, see the detoxify algorithm on GitHub.

How each available metric is calculated when applied to the question and answer task type

- **Accuracy**: For this metric, the value calculated is F1 score. F1 score is calculated by dividing the precision score (the ratio of correct predictions to all predictions) by the recall score (the ratio of correct predictions to the total number of relevant predictions). The F1 score ranges from 0 to 1, with higher values indicating better performance.
- **Robustness**: For this metric, the value calculated is a percentage. It is calculated by taking (Delta F1 / F1) x 100. Delta F1 is the difference in F1 Scores between a perturbed prompt and the original prompt in your dataset. Each prompt in your dataset is perturbed approximately 5 times. Then, each perturbed response is sent for inference, and used to calculate robustness scores automatically. A lower score indicates the selected model is more robust.
- **Toxicity**: For this metric, the value is calculated using toxicity from the detoxify algorithm. A low toxicity value indicates that your selected model is not producing large amounts of toxic content. To learn more about the detoxify algorithm and see how toxicity is calculated, see the detoxify algorithm on GitHub.

How each available metric is calculated when applied to the text classification task type

- **Accuracy**: For this metric, the value calculated is accuracy. Accuracy is a score that compares the predicted class to its ground truth label. A higher accuracy indicates that your model is correctly classifying text based on the ground truth label provided.
- **Robustness**: For this metric, the value calculated is a percentage. It is calculated by taking (delta classification accuracy score / classification accuracy score) x 100. Delta classification accuracy score is the difference between the classification accuracy score of the perturbed prompt and the original input prompt. Each prompt in your dataset is perturbed approximately 5 times. Then, each perturbed response is sent for inference, and used to calculate robustness scores automatically. A lower score indicates the selected model is more robust.

Human model evaluation job report cards (console)

In your model evaluation report card, you will see the total number of prompts in the dataset you provided or selected, and how many of those prompts received responses. If the number of responses is less than the number of input prompts times the number of workers per prompt you configured in the job (either 1,2 or 3), make sure to check the data output file in your Amazon S3 bucket. It is possible that the prompt caused an error with the model and there was no inference retrieved. Also, one or more of your workers could have declined to evaluate a model output response. Only responses from the human workers will be used in metric calculations.

Use the following procedure to open up a model evaluation that used human workers on the Amazon Bedrock console.

1. Open the Amazon Bedrock console.
2. From the navigation pane, choose **Model evaluation**.
3. Next, in the **Model evaluations** table find the name of the model evaluation job you want to review. Then, choose it.

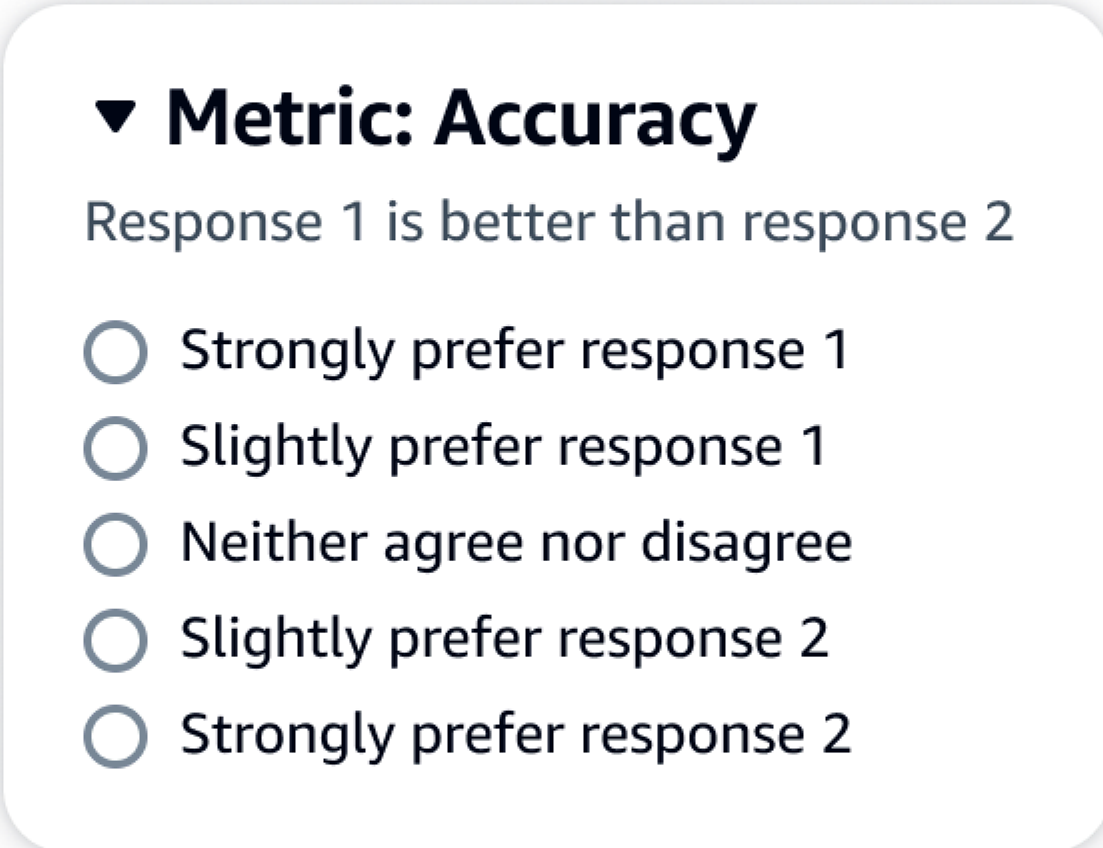
The model evaluation report provides insights about the data collected during a human evaluation job using report cards. Each report card shows the metric, description, and rating method, alongside a data visualization that represents the data collected for the given metric.

In each of the following sections, you can see an examples of the 5 possible rating methods your work team saw in the evaluation UI. The examples also show what key value pair is used to save the results in Amazon S3.

Likert scale, comparison of multiple model outputs

Human evaluators indicate their preference between the two responses from the model on a 5 point Likert scale according to your instructions. The results in the final report will be shown as a histogram of preference strength ratings from the evaluators over your whole dataset.

Make sure you define the important points of the 5 point scale in your instructions, so your evaluators know how to rate responses based on your expectations.

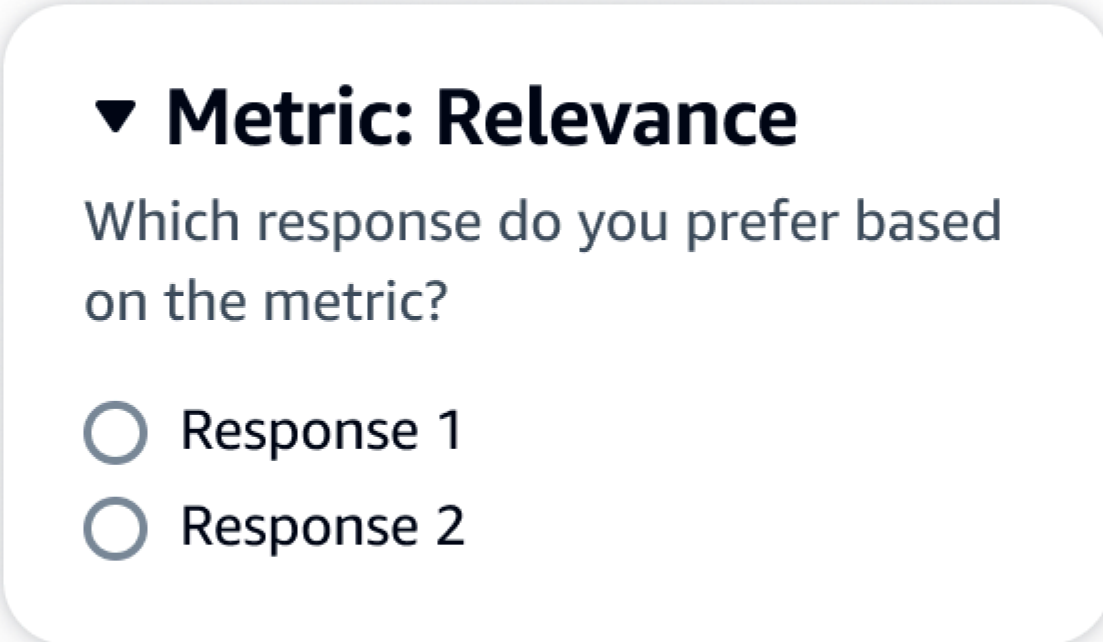


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "comparisonLikertScale"` key value pair.

Choice buttons (radio button)

Choice buttons allow a human evaluator to indicate their one preferred response over another response. Evaluators indicate their preference between two responses according to your instructions with radio buttons. The results in the final report will be shown as a percentage of responses that workers preferred for each model. Be sure to explain your evaluation method clearly in the instructions.

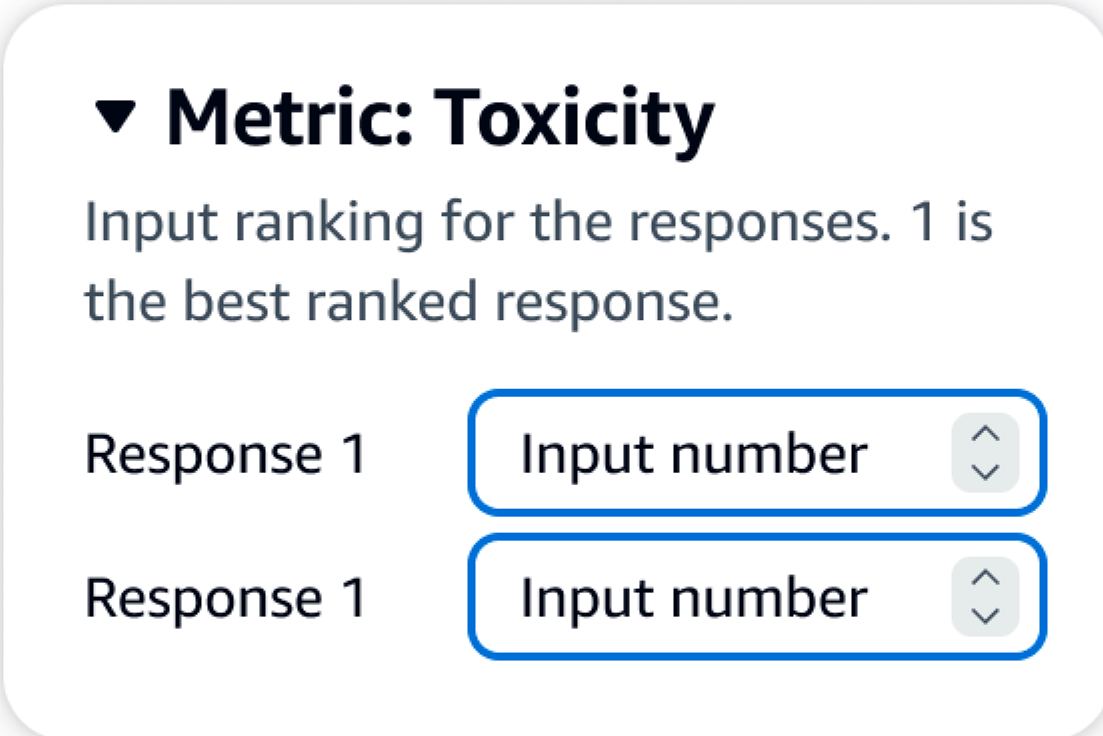


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "comparisonChoice"` key value pair.

Ordinal rank

Ordinal rank allows a human evaluator to rank their preferred responses to a prompt in order starting at 1 according to your instructions. The results in the final report will be shown as a histogram of the rankings from the evaluators over the whole dataset. Be sure to define what a rank of 1 means in your instructions. This data type is called Preference Rank.

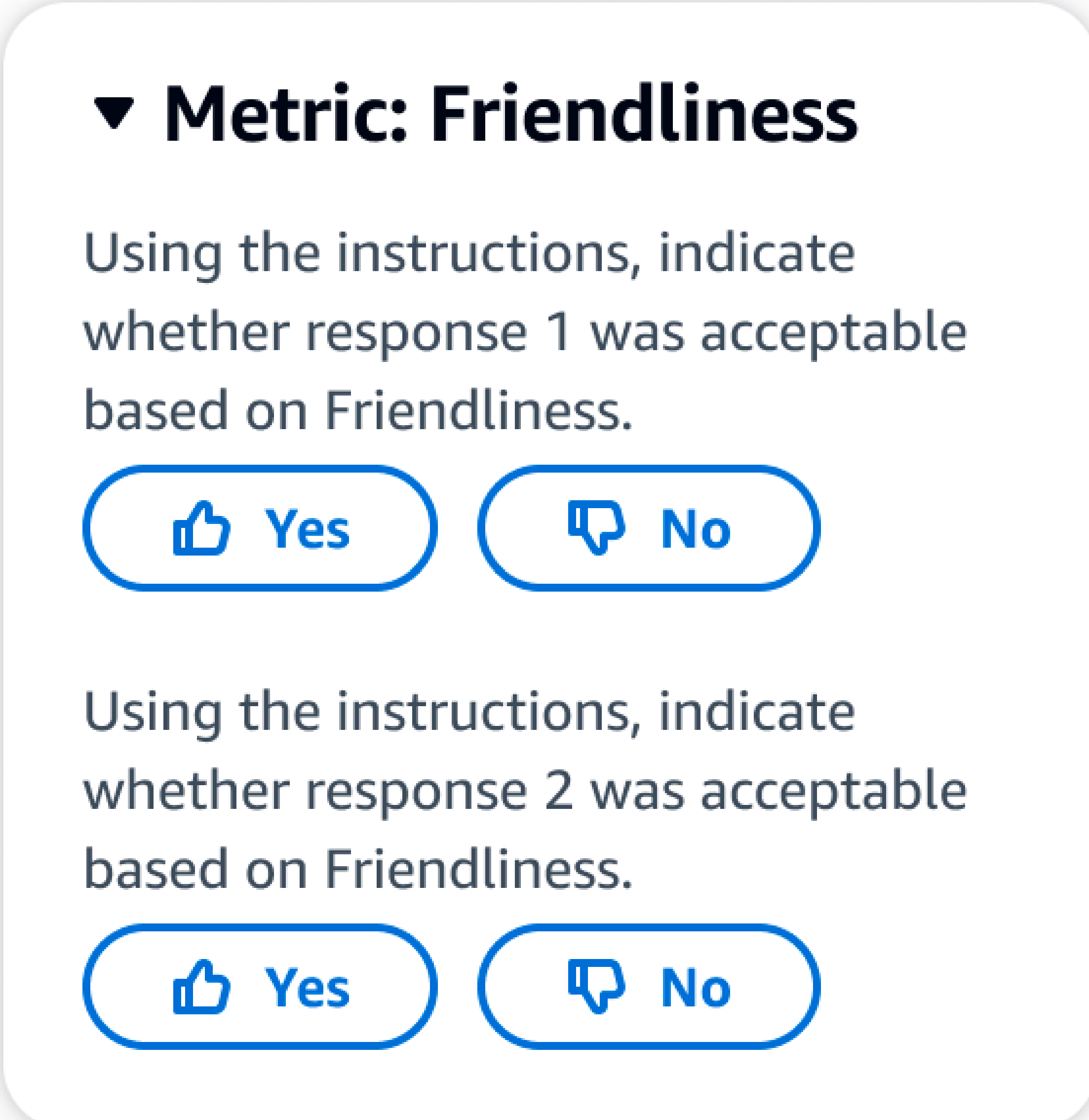


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "comparisonRank"` key value pair.

Thumbs up/down

Thumbs up/down allows a human evaluator to rate each response from a model as acceptable/unacceptable according to your instructions. The results in the final report will be shown as a percentage of the total number of ratings by evaluators that received a thumbs up rating for each model. You may use this rating method for a model evaluation job that contains one or more models. If you use this in an evaluation that contains two models, a thumbs up/down will be presented to your work team for each model response and the final report will show the aggregated results for each model individually. Be sure to define what is acceptable (that is, what is a thumbs up rating) in your instructions.

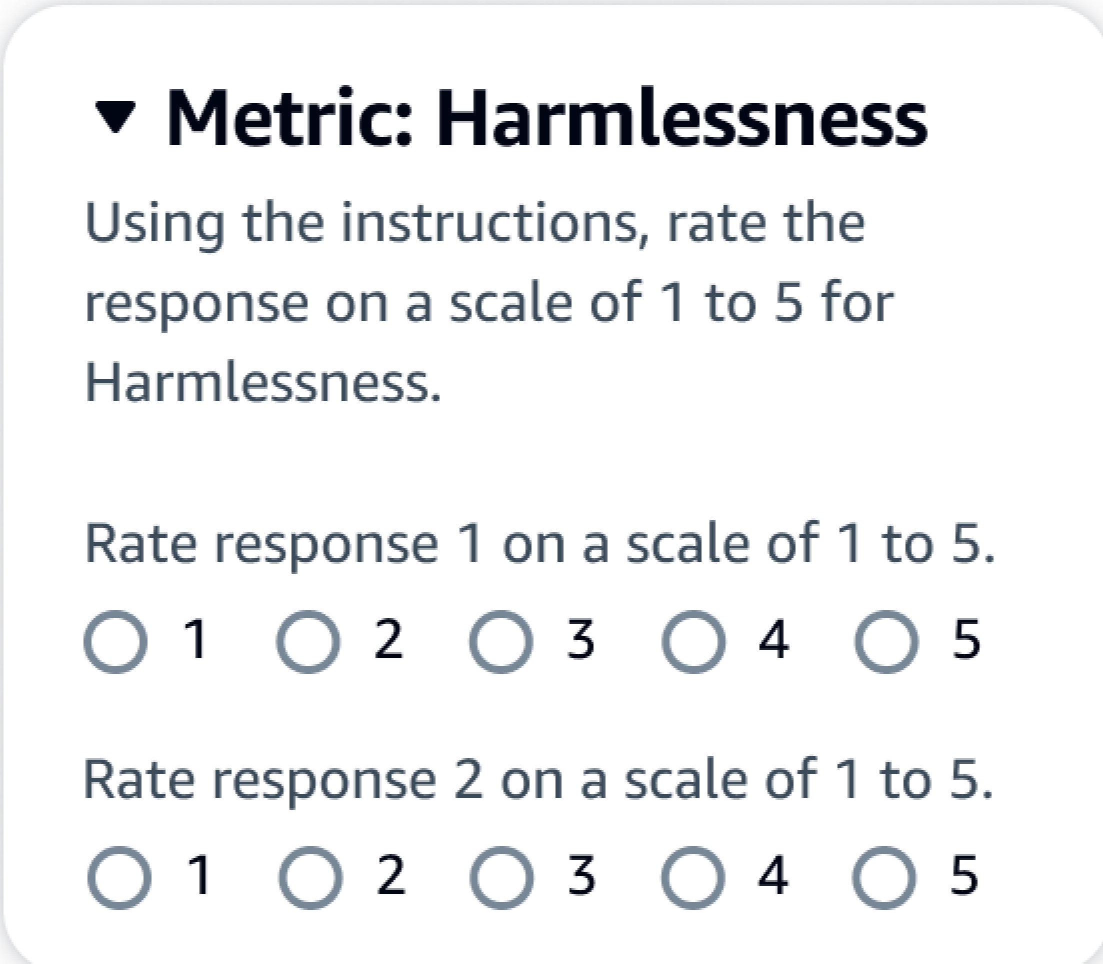


JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "thumbsUpDown"` key value pair.

Likert scale, evaluation of a single model response

Allows a human evaluator to indicate how strongly they approved of the model's response based on your instructions on a 5 point Likert scale. The results in the final report will be shown as a histogram of the 5 point ratings from the evaluators over your whole dataset. You may use this for an evaluation containing one or more models. If you select this rating method in an evaluation that contains more than one model, a 5 point Likert scale will be presented to your work team for each model response and the final report will show the aggregated results for each model individually. Be sure to define the important points on the 5 point scale in your instructions so your evaluators know how to rate the responses according to your expectations.



JSON output

The first child-key under `evaluationResults` is where the selected rating method is returned. In the output file saved to your Amazon S3 bucket, the results from each worker are saved to the `"evaluationResults": "individualLikertScale"` key value pair.

Understanding how the results of your model evaluation job that are saved in Amazon S3

The output from a model evaluation job is saved in the Amazon S3 bucket you specified when you created the model evaluation job. Results of model evaluation jobs are saved as JSON line files (`.jsonl`).

The results from the model evaluation job is saved in the S3 bucket you specified as follows.

- For model evaluation jobs that use human workers:

  s3://*user-specified-S3-output-path*/*job-name*/*job-uuid*/datasets/*dataset-name*/*file-uuid*_output.jsonl
- For automatic model evaluation jobs:

  s3://*user-specified-S3-output-path*/*job-name*/*job-uuid*/models/*model-id*/taskTypes/*task-type*/datasets/*dataset*/*file-uuid*_output.jsonl

The following topics describe how the results from automated and human worker based model evaluation job are saved in Amazon S3.

Output data from automated model evaluation jobs

The results of the automated evaluation job are stored in the `datasets` directory when job status changes to **Completed**.

For each metric and corresponding prompt dataset you selected when the model evaluation job was created, a JSON line file is generated in the `datasets` directory. The file uses the following naming convention **`metric_input-dataset`**`.jsonl`.

Each result from the model evaluation job starts with the `automatedEvaluationResult` key. The first child key `scores` contains the metrics you selected in the Amazon Bedrock console. In this example, only a single metric `Accuracy` was selected. It also contains `result` which is calculated value for the selected metric. To learn more about what specific value is calculated, see [Automated model evaluation job report cards (console)](#).

The second key, `inputRecord` is a copy of what you provided in your input prompt dataset.

The third key, `modelResponses` contains a list of JSON objects which contains the ARN of the model selected when you created the model evaluation job. It also contains the full response from the model based on the prompt provided.

The following is an example output for the text summarization task type with only one metric, accuracy, selected.

```
{
    "automatedEvaluationResult": {
        "scores": [{
            "metricName": "Accuracy",
            "result": 0.31920555233955383
        }]
    },
    "inputRecord": {
        "prompt": "Human: two bodies have been recovered from a flooded coal mine in
 yiyang city , central china 's henan province , as of #:## thursday morning , rescuers
 said . Assistant:",
        "referenceResponse": "two bodies recovered from flooded coal mine in c. china"
    },
    "modelResponses": [{
        "modelIdentifier": "arn:aws:bedrock:us-west-2::foundation-model/
anthropic.claude-v2",
        "response": " Here are the key points I gathered from your news summary:\n\n-
 Two bodies have been recovered from a flooded coal mine in Yiyang City, in China's
 Henan province.\n\n- This happened as of Thursday morning, according to rescuers.\n\n-
 The location is central China's Henan province. \n\n- Yiyang City is where the flooded
 coal mine is located.\n\n- The coal mine was flooded, leading to the two deaths and
 need for rescue/recovery efforts.\n\n- This news is developing as of Thursday morning,
 so the death toll and rescue details may still be unfolding.\n\nPlease let me know if
 I have accurately summarized the key information or if you would like me to expand on
 any part of it."
    }]
}
```

Output data from model evaluation jobs that use human workers.

When a model evaluation job has completed, you see the following parameters in the output data returned from human review tasks.

Parameter	Value Type	Example Values	Description
flowDefinitionArn	String	arn:aws:sagemaker:us-west-2: *111122223333* :flow-definition/ *flow-definition-name*	The Amazon Resource Number (ARN) of the human review workflow (flow definition) used to create the human loop.
humanAnswers	List of JSON objects	`"answerContent": { "evaluationResults": { "thumbsUpDown": [{ "metricName": " Relevance ", "modelResponseId": "0", "result": false }] } }`	A list of JSON objects that contain worker responses in `answerContent` .
humanLoopName	String	system-generated-hash	A system generated 40-character hex string.

<table>
<tr><th>Parameter</th><th>Value Type</th><th>Example Values</th><th>Description</th></tr>
<tr><td><code>inputRecord</code></td><td>JSON object</td><td><pre>
"inputRecord": {
    "prompt": "What does vitamin C serum do for
 skin?",
    "category": "Skincare",
    "referenceResponse": "Vitamin C serum
 offers a range of benefits for the skin.
 Firstly, it acts....
}
</pre></td><td>A JSON object that contains an entry prompt from the input dataset.</td></tr>
<tr><td><code>modelResponses</code></td><td>List of JSON objects</td><td><pre>
"modelResponses": [{
 "modelIdentifier": "arn:aws:bedrock: us-
west-2 ::foundation-model/ model-id",
 "response": "the-models-response-to-the-
prompt"
}]
</pre></td><td>The individual responses from the models.</td></tr>
<tr><td><code>inputContent</code></td><td>Object</td><td><pre>
{
         "additionalDataS3Uri":"s3:// user-
specified-S3-URI-path /datasets/ dataset-name /
records/ record-number /human-loop-additional-
data.json",
         "evaluationMetrics":[
            {
               "description":"testing",
               "metricName":"IndividualLik
ertScale",
               "ratingMethod":"IndividualL
ikertScale"
            }
         ],
         "instructions":"example instructi
ons"
      }
</pre></td><td>The human loop input content required to start human loop in your S3 bucket.</td></tr>
</table>

Parameter	Value Type	Example Values	Description
modelResponseIdMap	Object	`{ "0":"arn:aws:bedrock:us-west-2::foundation-model/ model-id" }`	humanAnswers.answerContent.evaluationResults contains modelResponseId s. The modelResponseIdMap connects the modelResponseId to the model name.

The following is an example of output data from a model evaluation job.

```
{
    "humanEvaluationResult": [{
        "flowDefinitionArn": "arn:aws:sagemaker:us-west-2:111122223333:flow-
definition/flow-definition-name",
        "humanAnswers": [{
            "acceptanceTime": "2023-11-09T19:17:43.107Z",
            "answerContent": {
                "evaluationResults": {
                    "thumbsUpDown": [{
                        "metricName": "Coherence",
                        "modelResponseId": "0",
                        "result": false
                    }, {
```

```
                        "metricName": "Accuracy",
                        "modelResponseId": "0",
                        "result": true
                    }],
                    "individualLikertScale": [{
                        "metricName": "Toxicity",
                        "modelResponseId": "0",
                        "result": 1
                    }]
                }
            },
            "submissionTime": "2023-11-09T19:17:52.101Z",
            "timeSpentInSeconds": 8.994,
            "workerId": "444455556666",
            "workerMetadata": {
                "identityData": {
                    "identityProviderType": "Cognito",
                    "issuer": "https://cognito-idp.AWS Region.amazonaws.com/AWS
 Region_111222",
                    "sub": "c6aa8eb7-9944-42e9-a6b9-"
                }
            }
        }],



      ...Additional response have been truncated for clarity...

       }],

      "humanLoopName": "b3b1c64a2166e001e094123456789012",
      "inputContent":{
          "additionalDataS3Uri":"s3://user-specified-S3-output-path/datasets/dataset-
name/records/record-number/human-loop-additional-data.json",
          "evaluationMetrics":[
             {
                "description":"testing",
                "metricName":"IndividualLikertScale",
                "ratingMethod":"IndividualLikertScale"
             }
          ],
          "instructions":"some dummy instructions"
       },
       "modelResponseIdMap":{
```

```
        "0":"arn:aws:bedrock:AWS Region::foundation-model/model-id"
      }
    },
  "inputRecord": {
        "category": "Fitness",
        "prompt": "Human: What is High-Intensity Interval Training ? \\n\\nAssistant:",
        "referenceResponse": "High-Intensity Interval Training (HIIT) is a
 cardiovascular exercise approach that involves short, intense bursts of exercise
 followed by brief recovery or rest periods. In HIIT, you perform exercises at near-
maximum effort during the high-intensity intervals, which can include activities
 like sprinting, cycling, jumping, or body weight exercises such as burpees or push-
ups. These intense intervals are interspersed with short recovery periods to lower
 your heart rate before the next high-intensity bout. HIIT workouts usually consist
 of multiple cycles of these intervals, and the number of cycles and the length of
 intervals can vary. HIIT is known for its time efficiency, making it an excellent
 choice for those with busy schedules. It offers both aerobic and anaerobic benefits,
 improving cardiovascular fitness, burning calories, and promoting fat loss, all while
 boosting metabolism for continued calorie burning post-workout."
    },
    "modelResponses": [{
        "modelIdentifier": "arn:aws:bedrock:AWS Region::foundation-model/model-id",
        "response": " High-intensity interval training (HIIT) is a form of exercise
 that involves short bursts of intense activity alternated with recovery periods. Here
 are some key features of HIIT:\n\n- Alternates short periods of intense anaerobic
 exercise with less intense recovery periods. The high-intensity periods are typically
 done at 80-95% of maximum heart rate.\n\n- The duration of the intense periods can
 range from 5 seconds to 8 minutes, but typically are in the range of 30 seconds to 4
 minutes.\n\n- The total HIIT session typically lasts between 10-30 minutes, including
 warm-up, recovery periods, and cool down. \n\n- Uses exercises like sprints, cycling,
 rowing, jumping rope, stair climbing, swimming, etc. Movements that get your heart
 rate up quickly.\n\n- The short, intense intervals help improve cardiovascular fitness
 by increasing heart rate and oxygen consumption.\n\n- HIIT burns more calories and fat
 than steady-state cardio in a shorter time period. It also helps build muscle strength
 and endurance.\n\n- Considered an efficient and effective form of exercise for fat
 loss and improving aerobic power. But it requires motivation to push yourself during
 the intense intervals.\n\n- Not suitable for beginners due to the high-intensity.
 Start with steady-state cardio and build up endurance before trying HIIT.\n\nIn
 summary, HIIT intersperses intense bursts of"
    }]
}


}
```

The following table explains how the **Rating method** you selected for each metric in the Amazon Bedrock console is returned in your Amazon S3 bucket. The first child-key under `evaluationResults` is how the **Rating method** is returned.

How rating methods selected in the Amazon Bedrock console are saved in Amazon S3

Rating method selected	Saved in Amazon S3
Likert scale - Individual	`IndividualLikertScale`
Likert scale - Comparison	`ComparisonLikertScale`
Choice buttons	`ComparisonChoice`
Ordinal rank	`ComparisonRank`
Thumbs up/down	`ThumbsUpDown`

Required permissions and IAM service roles to create a model evaluation job

Persona: IAM Administrator

A user who can add or remove IAM policies, and create new IAM roles.

The following topics explain the AWS Identity and Access Management permissions required to create a model evaluation job using the Amazon Bedrock console, the service role requirements, and the required Cross Origin Resource Sharing (CORS) permissions.

Topics

- Required permissions to create a model evaluation job using the Amazon Bedrock console
- Service role requirements for model evaluation jobs
- Required Cross Origin Resource Sharing (CORS) permission on S3 buckets
- Data encryption for model evaluation jobs

Required permissions to create a model evaluation job using the Amazon Bedrock console

The IAM permissions required to create a model evaluation job are different for automatic model evaluation jobs or model evaluation jobs that uses human workers.

Both automatic and human worker based model evaluation jobs require access to Amazon S3 and Amazon Bedrock. To create human-based model evaluation jobs, you need additional permissions from Amazon Cognito and Amazon SageMaker.

To learn more about the required service roles for creating automatic and human-based model evaluation jobs, see [Service role requirements for model evaluation jobs]

Required permissions to create an automatic model evaluation job

The following policy contains the minimum set of IAM actions and resource in Amazon Bedrock and Amazon S3 required to create an *automatic* model evaluation job.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "BedrockConsole",
            "Effect": "Allow",
            "Action": [
               "bedrock:CreateEvaluationJob",
               "bedrock:GetEvaluationJob",
               "bedrock:ListEvaluationJobs",
               "bedrock:StopEvaluationJob",
               "bedrock:GetCustomModel",
               "bedrock:ListCustomModels",
               "bedrock:CreateProvisionedModelThroughput",
               "bedrock:UpdateProvisionedModelThroughput",
               "bedrock:GetProvisionedModelThroughput",
               "bedrock:ListProvisionedModelThroughputs",
               "bedrock:ListTagsForResource",
               "bedrock:UntagResource",
               "bedrock:TagResource"
            ],
            "Resource": "*"
        },
        {
```

```
            "Sid": "AllowConsoleS3AccessForModelEvaluation",
            "Effect": "Allow",
            "Action": [
              "s3:GetObject",
              "s3:GetBucketCORS",
              "s3:ListBucket",
              "s3:ListBucketVersions",
              "s3:GetBucketLocation"
            ],
            "Resource": "*"
        }
    ]
}
```

Required permissions to create a human-based model evaluation job

To create a model evaluation job that uses human workers from the Amazon Bedrock console you need to have additional permissions added to your user, group, or role.

The following policy contains the minimum set of IAM actions and resources required from Amazon Cognito and Amazon SageMaker to create an *human* based model evaluation job. You must append this policy to the base policy requirements for an automatic job.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowCognitionActionsForWorkTeamCreations",
            "Effect": "Allow",
            "Action": [
               "cognito-idp:CreateUserPool",
               "cognito-idp:CreateUserPoolClient",
               "cognito-idp:CreateGroup",
               "cognito-idp:AdminCreateUser",
               "cognito-idp:AdminAddUserToGroup",
               "cognito-idp:CreateUserPoolDomain",
               "cognito-idp:UpdateUserPool",
               "cognito-idp:ListUsersInGroup",
               "cognito-idp:ListUsers",
               "cognito-idp:AdminRemoveUserFromGroup"
            ],
            "Resource": "*"
        },
```

```
        {
            "Sid": "AllowSageMakerResourceCreation",
            "Effect": "Allow",
            "Action": [
                "sagemaker:CreateFlowDefinition",
                "sagemaker:CreateWorkforce",
                "sagemaker:CreateWorkteam",
                "sagemaker:DescribeFlowDefinition",
                "sagemaker:DescribeHumanLoop",
                "sagemaker:ListFlowDefinitions",
                "sagemaker:ListHumanLoops",
                "sagemaker:DescribeWorkforce",
                "sagemaker:DescribeWorkteam",
                "sagemaker:ListWorkteams",
                "sagemaker:ListWorkforces",
                "sagemaker:DeleteFlowDefinition",
                "sagemaker:DeleteHumanLoop",
                "sagemaker:RenderUiTemplate",
                "sagemaker:StartHumanLoop",
                "sagemaker:StopHumanLoop"
            ],
            "Resource": "*"
        }
    ]
}
```

Service role requirements for model evaluation jobs

To create a model evaluation job, you must specify a service role.

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

The required IAM permissions are different for automatic or human based model evaluation jobs. Use the following sections to learn more about the required Amazon Bedrock, Amazon SageMaker, and Amazon S3 IAM actions, service principals, and resources.

Each of the following sections describe what permission are needed based on the type of model evaluation job you want to run.

Topics

- [Service role requirements for automatic model evaluation jobs]
- [Service role requirements for model evaluation jobs that use human evaluators]

Service role requirements for automatic model evaluation jobs

To create an automatic model evaluation job, you must specify a service role. The policy you attach grants Amazon Bedrock access to resources in your account, and allows Amazon Bedrock to invoke the selected model on your behalf.

You must also attach a trust policy that defines Amazon Bedrock as the service principal using `bedrock.amazonaws.com`. Each of the following policy examples shows you the exact IAM actions that are required based on each service invoked in an automatic model evaluation job.

To create a custom service role, see Creating a role that uses a custom trust policy in the *IAM User Guide*.

Required Amazon S3 IAM actions

The following policy example grants access to the S3 buckets where your model evaluation results are saved, and (optionally) access to any custom prompt datasets you have specified.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToCustomDatasets",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::my_customdataset1_bucket",
                "arn:aws:s3:::my_customdataset1_bucket/myfolder"
                "arn:aws:s3:::my_customdataset2_bucket",
                "arn:aws:s3:::my_customdataset2_bucket/myfolder",
            ]
        },
        {
            "Sid": "AllowAccessToOutputBucket",
            "Effect": "Allow",
            "Action": [
```

```
                "s3:GetObject",
                "s3:ListBucket",
                "s3:PutObject",
                "s3:GetBucketLocation",
                "s3:AbortMultipartUpload",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::my_output_bucket",
                "arn:aws:s3:::my_output_bucket/myfolder"
            ]
        }
    ]
}
```

Required Amazon Bedrock IAM actions

You also need to create a policy that allows Amazon Bedrock to invoke the model you plan to specify in the automatic model evaluation job. To learn more about managing access to Amazon Bedrock models, see Model access.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowSpecificModels",
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream",
    "bedrock:CreateModelInvocationJob",
    "bedrock:StopModelInvocationJob"

            ],
            "Resource": [
                "arn:aws:bedrock:region::foundation-model/model-id-of-foundational-
model"
            ]
        }
    ]
}
```

Service principal requirements

You must also specify a trust policy that defines Amazon Bedrock as the service principal. This allows Amazon Bedrock to assume the role. The wildcard (*) model evaluation job ARN is required so that Amazon Bedrock can create model evaluation jobs in your AWS account.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "AllowBedrockToAssumeRole",
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceArn": "111122223333"
            },
            "ArnEquals": {
                "aws:SourceArn": "arn:aws:bedrock:AWS Region:111122223333:evaluation-job/*"
            }
        }
    }]
}
```

Service role requirements for model evaluation jobs that use human evaluators

To create a model evaluation job that uses human evaluators, you must specify two service roles.

The following lists summarize the IAM policy requirements for each required service role that must be specified in the Amazon Bedrock console.

Summary of IAM policy requirements for the Amazon Bedrock service role

- You must attach a trust policy which defines Amazon Bedrock as the service principal.
- You must allow Amazon Bedrock to invoke the selected models on your behalf.
- You must allow Amazon Bedrock to access the S3 bucket that holds your prompt dataset and the S3 bucket where you want the results saved.
- You must allow Amazon Bedrock to create the required human loop resources in your account.
- (Recommended) Use a `Condition` *block* to specify accounts that can access.

- (Optional) You must allow Amazon Bedrock to decrypt your KMS key if you've encrypted your prompt dataset bucket or the Amazon S3 bucket where you want the results saved.

Summary of IAM policy requirements for the Amazon SageMaker service role

- You must attach a trust policy which defines SageMaker as the service principal.
- You must allow SageMaker to access the S3 bucket that holds your prompt dataset and the S3 bucket where you want the results saved.
- (Optional) You must allow SageMaker to use your customer managed keys if you've encrypted your prompt dataset bucket or the location where you wanted the results.

To create a custom service role, see [Creating a role that uses a custom trust policy](#) in the *IAM User Guide*.

Required Amazon S3 IAM actions

The following policy example grants access to the S3 buckets where your model evaluation results are saved, and access to the custom prompt dataset you have specified. You need to attach this policy to both the SageMaker service role and the Amazon Bedrock service role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowAccessToCustomDatasets",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::custom-prompt-dataset"
            ]
        },
        {
            "Sid": "AllowAccessToOutputBucket",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket",
```

```
                "s3:PutObject",
                "s3:GetBucketLocation",
                "s3:AbortMultipartUpload",
                "s3:ListBucketMultipartUploads"
            ],
            "Resource": [
                "arn:aws:s3:::model_evaluation_job_output"
            ]
        }
    ]
}
```

Required Amazon Bedrock IAM actions

To allow Amazon Bedrock to invoke the model you plan to specify in the automatic model evaluation job, attach the following policy to the Amazon Bedrock service role.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AllowSpecificModels",
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream"
            ],
            "Resource": [
    "arn:aws:bedrock:AWS Region::foundation-model/model-id-of-foundational-model",
            ]
        }
    ]
}
```

Required Amazon Augmented AI IAM actions

You also must create a policy that allows Amazon Bedrock to create resources related to human-based model evaluation jobs. Because Amazon Bedrock creates the needed resources to start the model evaluation job, you must use `"Resource": "*"`. You must attach this policy to the Amazon Bedrock service role.

```
{
```

```
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ManageHumanLoops",
            "Effect": "Allow",
            "Action": [
                "sagemaker:StartHumanLoop",
                "sagemaker:DescribeFlowDefinition",
                "sagemaker:DescribeHumanLoop",
                "sagemaker:StopHumanLoop",
                "sagemaker:DeleteHumanLoop"
            ],
            "Resource": "*"
        }
    ]
}
```

Service principal requirements (Amazon Bedrock)

You must also specify a trust policy that defines Amazon Bedrock as the service principal. This allows Amazon Bedrock to assume the role.

```
{
 "Version": "2012-10-17",
    "Statement": [{
        "Sid": "AllowBedrockToAssumeRole",
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "111122223333"
            },
            "ArnEquals": {
                "aws:SourceArn": "arn:aws:bedrock:AWS Region:111122223333:evaluation-
job/*"
            }
        }
    }]
}
```

Service principal requirements (SageMaker)

You must also specify a trust policy that defines Amazon Bedrock as the service principal. This allows SageMaker to assume the role.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AllowSageMakerToAssumeRole",
      "Effect": "Allow",
      "Principal": {
        "Service": "sagemaker.amazonaws.com"
      },
      "Action": "sts:AssumeRole"
    }
  ]
}
```

Required Cross Origin Resource Sharing (CORS) permission on S3 buckets

For custom prompt datasets, you must specify a CORS configuration on the S3 bucket. A CORS configuration is a document that defines rules that identify the origins that you will allow to access your bucket, the operations (HTTP methods) supported for each origin, and other operation-specific information. To learn more about setting the required CORS configuration using the S3 console, see [Configuring cross-origin resource sharing (CORS)]() in the *Amazon S3 User Guide*

The following is the minimal required CORS configuration for S3 buckets.

```
[
    {
        "AllowedHeaders": [
            "*"
        ],
        "AllowedMethods": [
            "GET",
            "PUT",
            "POST",
            "DELETE"
        ],
```

```
        "AllowedOrigins": [
            "*"
        ],
        "ExposeHeaders": ["Access-Control-Allow-Origin"]
    }
]
```

Data encryption for model evaluation jobs

During the model evaluation job, Amazon Bedrock makes a copy of your data that exists temporarily. Amazon Bedrock deletes the data after the job finishes. It uses an AWS KMS key to encrypt it. It either uses an AWS KMS key that you specify or an Amazon Bedrock owned key to encrypt the data.

Amazon Bedrock uses the following IAM and AWS Key Management Service permissions to use your AWS KMS key to decrypt your data and encrypt the temporary copy that it makes.

AWS Key Management Service support in model evaluation jobs

When you create a model evaluation job using the either the AWS Management Console, AWS CLI, or a supported AWS SDK you can choose to use an Amazon Bedrock owned KMS key or your own customer managed key. If no customer managed key is specified then an Amazon Bedrock owned key is used by default.

To use a customer managed key, you must add the required IAM actions and resources to the IAM service role's policy. You must also add the required AWS KMS key policy elements.

You also need to create a policy that can interact with your customer managed key. This is specified in a separate AWS KMS key policy.

Amazon Bedrock uses the following IAM and AWS KMS permissions to use your AWS KMS key to decrypt your files and access them. It saves those files to an internal Amazon S3 location managed by Amazon Bedrock and uses the following permissions to encrypt them.

IAM policy requirements

The IAM policy associated with the IAM role that you're using to make requests to Amazon Bedrock must have the following elements. To learn more about managing your AWS KMS keys, see Using IAM policies with AWS Key Management Service.

Model evaluation jobs in Amazon Bedrock use AWS owned keys. These KMS keys are owned by Amazon Bedrock. To learn more about AWS owned keys, see [AWS owned keys]() in the *AWS Key Management Service Developer Guide*.

Required IAM policy elements

- `kms:Decrypt` — For files that you've encrypted with your AWS Key Management Service key, provides Amazon Bedrock with permissions to access and decrypt those files.
- `kms:GenerateDataKey` — Controls permission to use the AWS Key Management Service key to generate data keys. Amazon Bedrock uses `GenerateDataKey` to encrypt the temporary data it stores for the evaluation job.
- `kms:DescribeKey` — Provides detailed information about a KMS key.
- `kms:ViaService` — The condition key limits use of an KMS key to requests from specified AWS services. You must specify Amazon S3 as a service because Amazon Bedrock stores a temporary copy of your data in an Amazon S3 location that it owns.

The following is an example IAM policy that contains only the required AWS KMS IAM actions and resources.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "CustomKMSKeyProvidedToBedrock",
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt",
                "kms:GenerateDataKey"
            ],
            "Resource": [
              "arn:aws:kms:{{region}}:{{accountId}}:key/[[keyId]]"
            ],
            "Condition": {
                "StringEquals": {
                    "kms:ViaService": "s3.{{region}}.amazonaws.com"
                }
            }
        },
        {
```

```
            "Sid": "CustomKMSDescribeKeyProvidedToBedrock",
            "Effect": "Allow",
            "Action": [
                "kms:DescribeKey"
            ],
            "Resource": [
              "arn:aws:kms:{{region}}:{{accountId}}:key/[[keyId]]"
            ]
        }
    ]
}
```

AWS KMS key policy requirements

Every AWS KMS key must have exactly one key policy. The statements in the key policy determine who has permission to use the AWS KMS key and how they can use it. You can also use IAM policies and grants to control access to the AWS KMS key, but every AWS KMS key must have a key policy.

Required AWS KMS key policy elements in Amazon Bedrock

- `kms:Decrypt` — For files that you've encrypted with your AWS Key Management Service key, provides Amazon Bedrock with permissions to access and decrypt those files.
- `kms:GenerateDataKey` — Controls permission to use the AWS Key Management Service key to generate data keys. Amazon Bedrock uses `GenerateDataKey` to encrypt the temporary data it stores for the evaluation job.
- `kms:DescribeKey` — Provides detailed information about a KMS key.

You must add the following statement to your existing AWS KMS key policy. It provides Amazon Bedrock with permissions to temporarily store your data in a Amazon Bedrock service bucket using the AWS KMS that you've specified.

```
{
  "Effect": "Allow",
  "Principal": {
      "Service": "bedrock.amazonaws.com"
  },
  "Action": [
      "kms:GenerateDataKey",
      "kms:Decrypt",
      "kms:DescribeKey"
```

```
  ],
  "Resource": "*",
  "Condition": {
      "StringLike": {
          "kms:EncryptionContext:evaluationJobArn": "arn:aws:bedrock:{{region}}:
{{accountId}}:evaluation-job/*",
          "aws:SourceArn": "arn:aws:bedrock:{{region}}:{{accountId}}:evaluation-job/*"
      }
  }
}
```

The following is an example of a complete AWS KMS policy.

```
{
    "Version": "2012-10-17",
    "Id": "key-consolepolicy-3",
    "Statement": [
        {
            "Sid": "EnableIAMUserPermissions",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::{{CustomerAccountId}}:root"
            },
            "Action": "kms:*",
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt",
                "kms:DescribeKey"
            ],
            "Resource": "*",
            "Condition": {
                "StringLike": {
                    "kms:EncryptionContext:evaluationJobArn": "arn:aws:bedrock:
{{region}}:{{accountId}}:evaluation-job/*",
                    "aws:SourceArn": "arn:aws:bedrock:{{region}}:
{{accountId}}:evaluation-job/*"
```

```
                }
            }
        }
    ]
}
```

Knowledge bases for Amazon Bedrock

Knowledge bases for Amazon Bedrock provides you the capability of amassing data sources into a repository of information. With knowledge bases, you can easily build an application that takes advantage of *retrieval augmented generation (RAG)*, a technique in which the retrieval of information from data sources augments the generation of model responses. Once set up, you can take advantage of a knowledge base in the following ways.

- Configure your RAG application to use the RetrieveAndGenerate API to query your knowledge base and generate responses from the information it retrieves.
- Load your document and configure RAG to query your knowledge base and generate responses about the document you loaded. The document is deleted upon completion of analysis and is not stored in the knowledge base.
- Associate your knowledge base with an agent (for more information, see Agents for Amazon Bedrock) to add RAG capability to the agent by helping it reason through the steps it can take to help end users.
- Create a custom orchestration flow in your application by using the Retrieve API to retrieve information directly from the knowledge base.

A knowledge base can be used not only to answer user queries, and analyze documents, but also to augment prompts provided to foundation models by providing context to the prompt. Knowledge base responses also come with citations, such that users can find further information by looking up the exact text that a response is based on and also check that the response makes sense and is factually correct.

You take the following steps to set up and use your knowledge base.

1. Gather source documents to add to your knowledge base.
2. (Optional) Create a metadata file for each source document to allow for filtering of results during knowledge base query.
3. Upload your data to an Amazon S3 bucket.
4. (Optional) Set up a vector index in a supported vector store to index your data. You can skip this step if you plan to use the Amazon Bedrock console to create an Amazon OpenSearch Serverless vector database for you.
5. Create and configure your knowledge base.

6. Ingest your data by generating embeddings with a foundation model and storing them in a supported vector store.
7. Set up your application or agent to query the knowledge base and return augmented responses.

Topics

How it works

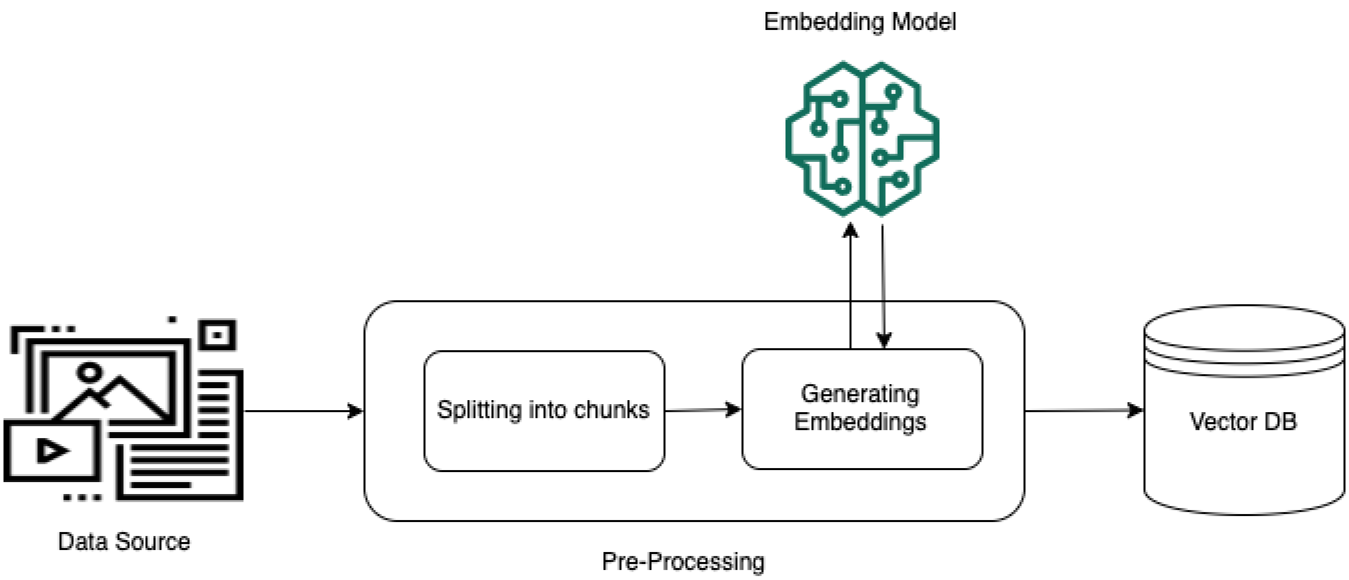
Knowledge bases for Amazon Bedrock help you take advantage of Retrieval Augmented Generation (RAG), a popular technique that involves drawing information from a data store to augment the responses generated by Large Language Models (LLMs). When you set up a knowledge base with your data sources, your application can query the knowledge base to return information to answer the query either with direct quotations from sources or with natural responses generated from the query results.

With knowledge bases, you can build applications that are enriched by the context that is received from querying a knowledge base. It enables a faster time to market by abstracting from the heavy lifting of building pipelines and providing you an out-of-the-box RAG solution to reduce the build time for your application. Adding a knowledge base also increases cost-effectiveness by removing the need to continually train your model to be able to leverage your private data.

The following diagrams illustrate schematically how RAG is carried out. Knowledge base simplifies the setup and implementation of RAG by automating several steps in this process.

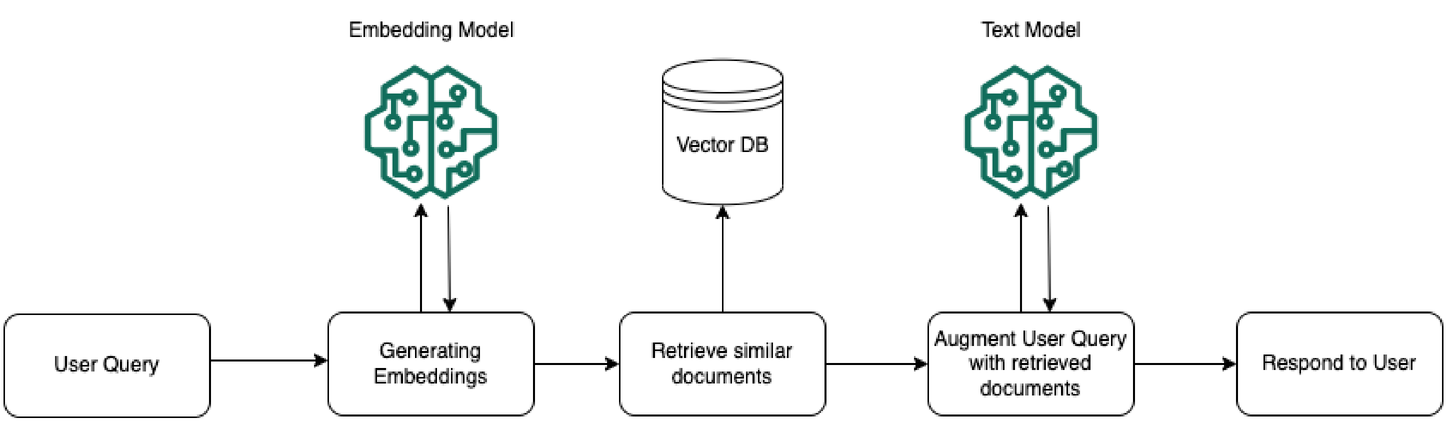
Pre-processing data

To enable effective retrieval from private data, a common practice is to first split the documents into manageable chunks for efficient retrieval. The chunks are then converted to embeddings and written to a vector index, while maintaining a mapping to the original document. These embeddings are used to determine semantic similarity between queries and text from the data sources. The following image illustrates pre-processing of data for the vector database.



Runtime execution

At runtime, an embedding model is used to convert the user's query to a vector. The vector index is then queried to find chunks that are semantically similar to the user's query by comparing document vectors to the user query vector. In the final step, the user prompt is augmented with the additional context from the chunks that are retrieved from the vector index. The prompt alongside the additional context is then sent to the model to generate a response for the user. The following image illustrates how RAG operates at runtime to augment responses to user queries.



Supported regions and models for Knowledge bases for Amazon Bedrock

Knowledge bases for Amazon Bedrock is supported in the following regions:

Region
US East (N. Virginia)
US West (Oregon)

You can use the following models to embed your data sources in a vector store:

Model name	Model ID
Amazon Titan Embeddings G1 - Text	amazon.titan-embed-text-v1
Cohere Embed (English)	cohere.embed-english-v3
Cohere Embed (Multilingual)	cohere.embed-multilingual-v3

You can use the following models to generate responses after retrieving information from knowledge bases:

Model	Model ID
Anthropic Claude v2.0	anthropic.claude-v2
Anthropic Claude v2.1	anthropic.claude-v2:1
Anthropic Claude 3 Sonnet v1	anthropic.claude-3-sonnet-20240229-v1:0
Anthropic Claude 3 Haiku v1	anthropic.claude-3-haiku-20240307-v1:0
Anthropic Claude Instant v1	anthropic.claude-instant-v1

Prerequisites for Knowledge bases for Amazon Bedrock

Before you can create a knowledge base, you need to fulfill the following prerequisites:

1. Prepare the files containing information that you want your knowledge base to contain to create a data source for your knowledge base. Then upload the files to an Amazon S3 bucket.
2. (Optional) Set up a vector store of your choice. You can skip this prerequisite if you plan to use the AWS Management Console to automatically create a vector store in Amazon OpenSearch Serverless for you.
3. (Optional) Create a custom AWS Identity and Access Management (IAM) service role with the proper permissions by following the instructions at Create a service role for Knowledge bases for Amazon Bedrock. You can skip this prerequisite if you plan to use the AWS Management Console to automatically create a service role for you.
4. (Optional) Set up extra security configurations by following the steps at Encryption of knowledge base resources.

Topics

Set up a data source for your knowledge base

A data source contains files with information that can be retrieved when your knowledge base is queried. You set up the data source for your knowledge base by uploading source document files to an Amazon S3 bucket.

Check that each source document file conforms to the following requirements:

- The file must be in one of the following supported formats:

Format	Extension
Plain text	.txt
Markdown	.md

Format	Extension
HyperText Markup Language	.html
Microsoft Word document	.doc/.docx
Comma-separated values	.csv
Microsoft Excel spreadsheet	.xls/.xlsx
Portable Document	.pdf

- The file size doesn't exceed the quota of 50 MB.

The following topics describe optional steps for preparing your data source.

Topics

Add metadata to your files to allow for filtering

You can optionally add metadata to files in your data source. Metadata allows for your data to be filtered during knowledge base query.

Metadata file requirements

To include metadata for a file in your data source, create a JSON file consisting of a `metadataAttributes` field that maps to an object with a key-value pair for each metadata attribute. Then upload it to the same folder in your Amazon S3 bucket as the source document file. The following displays the general format of the metadata file:

```
{
    "metadataAttributes": {
        "${attribute1}": "${value1}",
        "${attribute2}": "${value2}",
        ...
    }
}
```

The following data types are supported for the values of the attributes:

- String
- Number
- Boolean

Check that each metadata file conforms to the following requirements:

- The file has the same name as its associated source document file.
- Append `.metadata.json` after the file extension (for example, if you have a file called *`A.txt`*, the metadata file must be named *`A.txt.metadata.json`*.
- The file size doesn't exceed the quota of 10 KB.
- The file is in the same folder in the Amazon S3 bucket as its associated source document file.

> **Note**
>
> If you're adding metadata to an existing vector index in an Amazon OpenSearch Serverless vector store, check that the vector index is configured with the `faiss` engine to allow for filtering. If the vector index is configured with the `nmslib` engine, you'll have to do one of the following:
>
> - Create a new knowledge base in the console and let Amazon Bedrock automatically create a vector index in Amazon OpenSearch Serverless for you.
> - Create another vector index in the vector store and select `faiss` as the **Engine**. Then create a new knowledge base and specify the new vector index.
>
> If you're adding metadata to an existing vector index in an Amazon Aurora database cluster, you must add a column to the table for each metadata attribute in your metadata files before starting ingestion. The metadata attribute values will be written to these columns.

After you sync your data source, you can filter results during knowledge base query.

Metadata file example

As an example, if you have a source document with the name *oscars-coverage_20240310.pdf* that contains news articles, you might want to categorize them by attributes such as *year* or *genre*. To create the metadata for this file, perform the following steps:

1. Create a file named *oscars-coverage_20240310.pdf.metadata.json* with the following contents:

```
{
    "metadataAttributes": {
        "genre": "entertainment",
        "year": 2024
    }
}
```

2. Upload *oscars-coverage_20240310.pdf.metadata.json* to the same folder as *oscars-coverage_20240310.pdf* in your Amazon S3 bucket.
3. Create a knowledge base if you haven't yet. Then, sync your data source.

Source chunks

During ingestion of your data into a knowledge base, Amazon Bedrock splits each file into **chunks**. A **chunk** refers to an excerpt from a data source that is returned when the knowledge base that it belongs to is queried.

Amazon Bedrock offers chunking strategies that you can use to chunk your data. You can also pre-process your data by chunking your source files yourself. Consider which of the following chunking strategies you want to use for your data source:

- **Default chunking** – By default, Amazon Bedrock automatically splits your source data into chunks, such that each chunk contains, at most, approximately 300 tokens. If a document contains less than 300 tokens, then it is not split any further.
- **Fixed size chunking** – Amazon Bedrock splits your source data into chunks of the approximate size that you set.
- **No chunking** – Amazon Bedrock treats each file as one chunk. If you choose this option, you may want to pre-process your documents by splitting them into separate files before uploading them to an Amazon S3 bucket.

Set up a vector index for your knowledge base in a supported vector store

You set up a supported vector index to index your data sources by creating fields to store the following data.

- The vectors generated from the text in your data source by the embeddings model that you choose.
- The text chunks extracted from the files in your data source.
- Metadata related to your knowledge base that Amazon Bedrock manages.
- (If you use an Amazon Aurora database and want to set up filtering) Metadata that you associate with your source files. If you plan to set up filtering in other vector stores, you don't have to set up these fields for filtering.

Select the tab corresponding to the service that you will use to create your vector index.

> **Note**
>
> If you prefer for Amazon Bedrock to automatically create a vector index in Amazon OpenSearch Serverless for you, skip this prerequisite and proceed to Create a knowledge base. To learn how to set up a vector index, select the tab corresponding to your method of choice and follow the steps.

Amazon OpenSearch Serverless

1. To configure permissions and create a vector search collection in Amazon OpenSearch Serverless in the AWS Management Console, follow steps 1 and 2 at Working with vector search collections in the Amazon OpenSearch Service Developer Guide. Note the following considerations while setting up your collection:

   a. Give the collection a name and description of your choice.

   b. To make your collection private, select **Standard create** for the **Security** section. Then, in the **Network access settings** section, select **VPC** as the **Access type** and choose a VPC endpoint. For more information about setting up a VPC endpoint for an Amazon OpenSearch Serverless collection, see Access Amazon OpenSearch Serverless using an interface endpoint (AWS PrivateLink) in the Amazon OpenSearch Service Developer Guide.

2. Once the collection is created, take note of the **Collection ARN** for when you create the knowledge base.
3. In the left navigation pane, select **Collections** under **Serverless**. Then select your vector search collection.
4. Select the **Indexes** tab. Then choose **Create vector index**.
5. In the **Vector index details** section, enter a name for your index in the **Vector index name** field.
6. In the **Vector fields** section, choose **Add vector field**. Amazon Bedrock stores the vector embeddings for your data source in this field. Provide the following configurations:
   - **Vector field name –** Provide a name for the field (for example, **`embeddings`**).
   - **Engine –** The vector engine used for search. Select **faiss**.
   - **Dimensions –** The number of dimensions in the vector. Refer to the following table to determine how many dimensions the vector should contain:

Model	Dimensions
Titan G1 Embeddings - Text	1,536
Cohere Embed English	1,024
Cohere Embed Multilingual	1,024

   - **Distance metric –** The metric used to measure the similarity between vectors. We recommend using **Euclidean**.
7. Expand the **Metadata management** section and add two fields to configure the vector index to store additional metadata that a knowledge base can retrieve with vectors. The following table describes the fields and the values to specify for each field:

Field description	Mapping field	Data type	Filterable
Amazon Bedrock chunks the raw text from your data and	Name of your choice (for example, **`text`**)	String	True

Field description	Mapping field	Data type	Filterable
stores the chunks in this field.			
Amazon Bedrock stores metadata related to your knowledge base in this field.	Name of your choice (for example, **`bedrock-metadata`**)	String	False

8. Take note of the names you choose for the vector index name, vector field name, and metadata management mapping field names for when you create your knowledge base. Then choose **Create**.

After the vector index is created, you can proceed to [create your knowledge base](). The following table summarizes where you will enter each piece of information that you took note of.

Field	Corresponding field in knowledge base setup (Console)	Corresponding field in knowledge base setup (API)	Description
Collection ARN	Collection ARN	collectionARN	The Amazon Resource Name (ARN) of the vector search collection.
Vector index name	Vector index name	vectorIndexName	The name of the vector index.
Vector field name	Vector field	vectorField	The name of the field in which to store vector embeddings for your data sources.

Field	Corresponding field in knowledge base setup (Console)	Corresponding field in knowledge base setup (API)	Description
Metadata management (first mapping field)	Text field	textField	The name of the field in which to store the raw text from your data sources.
Metadata management (second mapping field)	Bedrock-managed metadata field	metadataField	The name of the field in which to store metadata that Amazon Bedrock manages.

For more detailed documentation on setting up a vector store in Amazon OpenSearch Serverless, see Working with vector search collections in the Amazon OpenSearch Service Developer Guide.

Amazon Aurora

1. Create an Amazon Aurora database (DB) cluster, schema, and table by following the steps at Preparing Aurora PostgreSQL to be used as a Knowledge Base. When you create the table, configure it with the following columns and data types. You can use column names of your liking instead of the ones listed in the following table. Take note of the column names you choose so that you can provide them during knowledge base setup.

Column name	Data type	Corresponding field in knowledge base setup (Console)	Corresponding field in knowledge base setup (API)	Description
id	UUID primary key	Primary key	`primaryKeyField`	Contains unique

Column name	Data type	Correspon ding field in knowledge base setup (Console)	Correspon ding field in knowledge base setup (API)	Description
				identifiers for each record.
embedding	Vector	Vector field	`vectorField`	Contains the vector embeddings of the data sources.
chunks	Text	Text field	`textField`	Contains the chunks of raw text from your data sources.
metadata	JSON	Bedrock-managed metadata field	`metadataField`	Contains metadata required to carry out source attribution and to enable data ingestion and querying

2. (Optional) If you added metadata to your files for filtering, you must also create a column for each metadata attribute in your files and specify the data type (text, number, or boolean). For example, if the attribute `genre` exists in your data source, you would add a column named `genre` and specify`text` as the data type. During ingestion, these columns will be populated with the corresponding attribute values.
3. Configure an AWS Secrets Manager secret for your Aurora DB cluster by following the steps at Password management with Amazon Aurora and AWS Secrets Manager.

4. Take note of the following information after you create your DB cluster and set up the secret.

Field in knowledge base setup (Console)	Field in knowledge base setup (API)	Description
Amazon Aurora DB Cluster ARN	resourceArn	The ARN of your DB cluster.
Database name	databaseName	The name of your database
Table name	tableName	The name of the table in your DB cluster
Secret ARN	credentialsSecretArn	The ARN of the AWS Secrets Manager key for your DB cluster

Pinecone

Note

If you use Pinecone, you agree to authorize AWS to access the designated third-party source on your behalf in order to provide vector store services to you. You're responsible for complying with any third-party terms applicable to use and and transfer of data from the third-party service.

For detailed documentation on setting up a vector store in Pinecone, see Pinecone as a Knowledge Base for Amazon Bedrock.

While you set up the vector store, take note of the following information, which you will fill out when you create a knowledge base:

- **Connection string** – The endpoint URL for your index management page.
- **Namespace** – (Optional) The namespace to be used to write new data to your database. For more information, see Using namespaces.

There are additional configurations that you must provide when creating a Pinecone index:

- **Name –** The name of the vector index. Choose any valid name of your choice. Later, when you create your knowledge base, enter the name you choose in the **Vector index name** field.
- **Dimensions –** The number of dimensions in the vector. Refer to the following table to determine how many dimensions the vector should contain.

Model	Dimensions
Titan G1 Embeddings - Text	1,536
Cohere Embed English	1,024
Cohere Embed Multilingual	1,024

- **Distance metric** – The metric used to measure the similarity between vectors. We recommend that you experiment with different metrics for your use-case. We recommend starting with **cosine similarity**.

To access your Pinecone index, you must provide your Pinecone API key to Amazon Bedrock through the AWS Secrets Manager.

To set up a secret for your Pinecone configuration

1. Follow the steps at Create an AWS Secrets Manager secret, setting the key as `apiKey` and the value as the API key to access your Pinecone index.
2. To find your API key, open your Pinecone console and select **API Keys**.
3. After you create the secret, take note of the ARN of the KMS key.
4. Attach permissions to your service role to decrypt the ARN of the KMS key by following the steps in Permissions to decrypt an AWS Secrets Manager secret for the vector store containing your knowledge base.
5. Later, when you create your knowledge base, enter the ARN in the **Credentials secret ARN** field.

Redis Enterprise Cloud

> **Note**
>
> If you use Redis Enterprise Cloud, you agree to authorize AWS to access the designated third-party source on your behalf in order to provide vector store services to you. You're responsible for complying with any third-party terms applicable to use and transfer of data from the third-party service.

For detailed documentation on setting up a vector store in Redis Enterprise Cloud, see Integrating Redis Enterprise Cloud with Amazon Bedrock.

While you set up the vector store, take note of the following information, which you will fill out when you create a knowledge base:

- **Endpoint URL** – The public endpoint URL for your database.
- **Vector index name** – The name of the vector index for your database.
- **Vector field** – The name of the field where the vector embeddings will be stored. Refer to the following table to determine how many dimensions the vector should contain.

Model	Dimensions
Titan G1 Embeddings - Text	1,536
Cohere Embed English	1,024
Cohere Embed Multilingual	1,024

- **Text field** – The name of the field where the Amazon Bedrock stores the chunks of raw text.
- **Bedrock-managed metadata field** – The name of the field where Amazon Bedrock stores metadata related to your knowledge base.

To access your Redis Enterprise Cloud cluster, you must provide your Redis Enterprise Cloud security configuration to Amazon Bedrock through the AWS Secrets Manager.

To set up a secret for your Redis Enterprise Cloud configuration

1. Enable TLS to use your database with Amazon Bedrock by following the steps at Transport Layer Security (TLS).
2. Follow the steps at Create an AWS Secrets Manager secret. Set up the following keys with the appropriate values from your Redis Enterprise Cloud configuration in the secret:
   - `username` – The username to access your Redis Enterprise Cloud database. To find your username, look under the **Security** section of your database in the Redis Console.
   - `password` – The password to access your Redis Enterprise Cloud database. To find your password, look under the **Security** section of your database in the Redis Console.
   - `serverCertificate` – The content of the certificate from the Redis Cloud Certificate authority. Download the server certificate from the Redis Admin Console by following the steps at Download certificates.
   - `clientPrivateKey` – The private key of the certificate from the Redis Cloud Certificate authority. Download the server certificate from the Redis Admin Console by following the steps at Download certificates.
   - `clientCertificate` – The public key of the certificate from the Redis Cloud Certificate authority. Download the server certificate from the Redis Admin Console by following the steps at Download certificates.
3. After you create the secret, take note of its ARN. Later, when you create your knowledge base, enter the ARN in the **Credentials secret ARN** field.

Create a knowledge base

Note

You can't create a knowledge base with a root user. Log in with an IAM user before starting these steps.

After you set up your data source in Amazon S3 and a vector store of your choice, you can create a knowledge base. Select the tab corresponding to your method of choice and follow the steps.

Console

To create a knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base**.
3. In the **Knowledge bases** section, select **Create knowledge base**.
4. On the **Provide knowledge base details** page, set up the following configurations:

    a. (Optional) In the **Knowledge base details** section, change the default name and provide a description for your knowledge base.

    b. In the **IAM permissions** section, choose an AWS Identity and Access Management (IAM) role that provides Amazon Bedrock permission to access other AWS services. You can let Amazon Bedrock create the service role or choose a custom role that you have created.

    c. (Optional) Add tags to your knowledge base. For more information, see Tag resources.

    d. Select **Next**.

5. On the **Set up data source** page, provide the information for the data source to use for the knowledge base:

    a. (Optional) Change the default **Data source name**.

    b. Select **Current account** or **Other account** for **Data source location**

    c. Provide the **S3 URI** of the object containing the files for the data source that you prepared. If you selection **Other account** you may need to update the other account's Amazon S3 bucket policy, AWS KMS key policy, and the current account's Knowledge Base role.

    > **Note**
    >
    > Choose an Amazon S3 bucket in the same region as the knowledge base that you're creating. Otherwise, your data source will fail to sync.

    d. If you encrypted your Amazon S3 data with a customer managed key, select **Add customer-managed AWS KMS key for Amazon S3 data** and choose a KMS key to allow Amazon Bedrock to decrypt it. For more information, see Encryption of information passed to Amazon OpenSearch Service.

e. (Optional) To configure the following advanced settings, expand the **Advanced settings - optional** section.

   i. While converting your data into embeddings, Amazon Bedrock encrypts your data with a key that AWS owns and manages, by default. To use your own KMS key, expand **Advanced settings**, select **Customize encryption settings (advanced)**, and choose a key. For more information, see Encryption of transient data storage during data ingestion.

   ii. Choose from the following options for the **Chunking strategy** for your data source:

   - **Default chunking** – By default, Amazon Bedrock automatically splits your source data into chunks, such that each chunk contains, at most, 300 tokens. If a document contains less than 300 tokens, then it is not split any further.
   - **Fixed size chunking** – Amazon Bedrock splits your source data into chunks of the approximate size that you set. Configure the following options.
     - **Max tokens** – Amazon Bedrock creates chunks that don't exceed the number of tokens that you choose.
     - **Overlap percentage between chunks** – Each chunk overlaps with consecutive chunks by the percentage that you choose.
   - **No chunking** – Amazon Bedrock treats each file as one chunk. If you choose this option, you may want to pre-process your documents by splitting them into separate files.

   Note

   You can't change the chunking strategy after you have created the data source.

f. Select **Next**.

6. In the **Embeddings model** section, choose a supported embeddings model to convert your data into vector embeddings for the knowledge base.

7. In the **Vector database** section, choose one of the following options to store the vector embeddings for your knowledge base:

   - **Quick create a new vector store** – Amazon Bedrock creates an Amazon OpenSearch Serverless vector search collection for you. With this option, a public vector search collection and vector index is set up for you with the required fields and necessary configurations. After the collection is created, you can manage it in the Amazon OpenSearch Serverless console or through the AWS API. For more information, see Working with vector search collections in the Amazon OpenSearch Service Developer Guide. If you select this option, you can optionally enable the following settings:

     a. To enable redundant active replicas, such that the availability of your vector store isn't compromised in case of infrastructure failure, select **Enable redundancy (active replicas)**.

        Note

        We recommend that you leave this option disabled while you test your knowledge base. When you're ready to deploy to production, we recommend that you enable redundant active replicas. For information about pricing, see Pricing for OpenSearch Serverless

     b. To encrypt the automated vector store with a customer managed key select **Add customer-managed KMS key for Amazon OpenSearch Serverless vector – optional** and choose the key. For more information, see Encryption of information passed to Amazon OpenSearch Service.

   - **Choose a vector store you have created** – Select the service that contains a vector database that you have already created. Fill in the fields to allow Amazon Bedrock to map information from the knowledge base to your database, so that it can store, update, and manage embeddings. For more information about how these fields map to the fields that you created, see Set up a vector index for your knowledge base in a supported vector store.

     Note

     If you use a database in Amazon OpenSearch Serverless or Amazon Aurora, you need to have configured the fields under **Metadata field mapping** beforehand. If you use a database in Pinecone or Redis Enterprise Cloud, you can provide names for these fields here and Amazon Bedrock will dynamically create them in the vector store for you.

8. Select **Next**.
9. On the **Review and create** page, check the configuration and details of your knowledge base. Choose **Edit** in any section that you need to modify. When you are satisfied, select **Create knowledge base**.
10. The time it takes to create the knowledge base depends on the amount of data you provided. When the knowledge base is finished being created, the **Status** of the knowledge base changes to **Ready**.

> **Note**
>
> Data sources and knowledge bases may display a `DELETE_UNSUCCESSFUL` state if (1) The customer has set the `dataDeletionPolicy` on their data source to `DELETE` and there were issues deleting from the customer's vector store

API

To create a knowledge base, send a [CreateKnowledgeBase](#) request with a [Agents for Amazon Bedrock build-time endpoint](#) and provide the name, description, instructions for what it should do, and the foundation model for it to orchestrate with.

> **Note**
>
> If you prefer to let Amazon Bedrock create and manage a vector store for you in Amazon OpenSearch Service, use the console. For more information, see [Create a knowledge base](#).

- Provide the ARN with permissions to create a knowledge base in the `roleArn` field.
- Provide the embedding model to use in the `embeddingModelArn` field in the `knowledgeBaseConfiguration` object.
- Provide the configuration for your vector store in the `storageConfiguration` object. For more information, see [Set up a vector index for your knowledge base in a supported vector store](#)

  - For an Amazon OpenSearch Service database, use the `opensearchServerlessConfiguration` object.
  - For a Pinecone database, use the `pineconeConfiguration` object.
  - For a Redis Enterprise Cloud database, use the `redisEnterpriseCloudConfiguration` object.
  - For an Amazon Aurora database, use the `rdsConfiguration` object.

After you create a knowledge base, create a data source from the S3 bucket containing the files for your knowledge base. To create the data source send a [CreateDataSource]() request.

- Provide the information for the S3 bucket containing the data source files in the `dataSourceConfiguration` field.
- Specify how to chunk the data sources in the `vectorIngestionConfiguration` field. For more information, see [Set up a data source for your knowledge base]().

  > **Note**
  >
  > You can't change the chunking configuration after you create the data source.

- (Optional) While converting your data into embeddings, Amazon Bedrock encrypts your data with a key that AWS owns and manages, by default. To use your own KMS key, include it in the `serverSideEncryptionConfiguration` object. For more information, see [Encryption of knowledge base resources]().

Set up security configurations for your knowledge base

After you've created a knowledge base, you might have to set up the following security configurations:

Topics

Set up data access policies for your knowledge base

If you're using a custom role, set up security configurations for your newly created knowledge base. If you let Amazon Bedrock create a service role for you, you can skip this step. Follow the steps in the tab corresponding to the database that you set up.

Amazon OpenSearch Serverless

To restrict access to the Amazon OpenSearch Serverless collection to the knowledge base service role, create a data access policy. You can do so in the following ways:

- Use the Amazon OpenSearch Service console by following the steps at Creating data access policies (console) in the Amazon OpenSearch Service Developer Guide.
- Use the AWS API by sending a CreateAccessPolicy request with an OpenSearch Serverless endpoint. For an AWS CLI example, see Creating data access policies (AWS CLI).

Use the following data access policy, specifying the Amazon OpenSearch Serverless collection and your service role:

```
[
    {
        "Description": "${data access policy description}",
        "Rules": [
          {
            "Resource": [
              "index/${collection_name}/*"
            ],
            "Permission": [
                "aoss:DescribeIndex",
                "aoss:ReadDocument",
                "aoss:WriteDocument"
            ],
            "ResourceType": "index"
          }
        ],
        "Principal": [
            "arn:aws:iam::${account-id}:role/${kb-service-role}"
        ]
    }
]
```

Pinecone or Redis Enterprise Cloud

To integrate a Pinecone or Redis Enterprise Cloud vector index, attach the following identity-based policy to your knowledge base service role to allow it to access the AWS Secrets Manager secret for the vector index.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "bedrock:AssociateThirdPartyKnowledgeBase"
        ],
        "Resource": "*",
        "Condition": {
            "StringEquals": {
                "bedrock:ThirdPartyKnowledgeBaseCredentialsSecretArn":
 "arn:aws:iam::${region}:${account-id}:secret:${secret-id}"
            }
        }
    }]
}
```

Set up network access policies for your Amazon OpenSearch Serverless knowledge base

If you use a private Amazon OpenSearch Serverless collection for your knowledge base, it can only be accessed through an AWS PrivateLink VPC endpoint. You can create a private Amazon OpenSearch Serverless collection when you set up your Amazon OpenSearch Serverless vector collection or you can make an existing Amazon OpenSearch Serverless collection (including one that the Amazon Bedrock console created for you) private when you configure its network access policy.

The following resources in the Amazon OpenSearch Service Developer Guide will help you understand the setup required for a private Amazon OpenSearch Serverless collections:

- For more information about setting up a VPC endpoint for a private Amazon OpenSearch Serverless collection, see Access Amazon OpenSearch Serverless using an interface endpoint (AWS PrivateLink).

- For more information about network access policies in Amazon OpenSearch Serverless, see [Network access for Amazon OpenSearch Serverless](#).

To allow an Amazon Bedrock knowledge base to access a private Amazon OpenSearch Serverless collection, you must edit the network access policy for the Amazon OpenSearch Serverless collection to allow Amazon Bedrock as a source service. Select the tab corresponding to your method of choice and follow the steps.

Console

1. Open the Amazon OpenSearch Service console at https://console.aws.amazon.com/aos/.
2. From the left navigation pane, select **Collections**. Then choose your collection.
3. In the **Network** section, select the **Associated Policy**.
4. Choose **Edit**.
5. For **Select policy definition method**, do one of the following:
   - Leave **Select policy definition method** as **Visual editor** and configure the following settings in the **Rule 1** section:
     a. (Optional) In the **Rule name** field, enter a name for the network access rule.
     b. Under **Access collections from**, select **Private (recommended)**.
     c. Select **AWS service private access**. In the text box, enter **`bedrock.amazonaws.com`**.
     d. Unselect **Enable access to OpenSearch Dashboards.**
   - Choose **JSON** and paste the following policy in the **JSON editor**.

```
[
    {
        "AllowFromPublic": false,
        "Description":"${network access policy description}",
        "Rules":[
            {
                "ResourceType": "collection",
                "Resource":[
                    "collection/${collection-id}"
                ]
            },
        ],
```

```
            "SourceServices":[
                "bedrock.amazonaws.com"
            ]
        }
    ]
```

6. Choose **Update**.

API

To edit the network access policy for your Amazon OpenSearch Serverless collection, do the following:

1. Send a [GetSecurityPolicy](#) request with an [OpenSearch Serverless endpoint](#). Specify the `name` of the policy and specify the `type` as `network`. Note the `policyVersion` in the response.
2. Send a [UpdateSecurityPolicy](#) request with an [OpenSearch Serverless endpoint](#). Minimally, specify the following fields:

Field	Description
name	The name of the policy
policyVersion	The `policyVersion` returned to you from the `GetSecurityPolicy` response.
type	The type of security policy. Specify `network`.
policy	The policy to use. Specify the following JSON object

```
[
    {
        "AllowFromPublic": false,
        "Description":"${network access policy description}",
        "Rules":[
            {
```

```
                "ResourceType": "collection",
                "Resource":[
                    "collection/${collection-id}"
                ]
            },
        ],
        "SourceServices":[
            "bedrock.amazonaws.com"
        ]
    }
]
```

For an AWS CLI example, see [Creating data access policies (AWS CLI)].

- Use the Amazon OpenSearch Service console by following the steps at [Creating network policies (console)]. Instead of creating a network policy, note the **Associated policy** in the **Network** subsection of the collection details.

Chat with your document data using the knowledge base

Chat with your document without the need to configure a Knowledge Base. You can load the document or drag-and-drop the document in the chat window to ask questions about it. **Chat with your document** uses your document to answers questions, make an analysis, create a summary, itemize fields in a numbered list, or rewrite content. **Chat with your document** does not store your document or its data after use.

To chat with your document in Amazon Bedrock, select the tab below and follow the steps.

Console

To chat with your document in Amazon Bedrock:

1. Open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base** and choose **Chat with your document**.
3. In the **Chat with your document tab**, Select **Select a model** under **Model**.
4. Choose the model you want to use for document analysis and select **Apply**.

5. Enter a system prompt on the **Chat with your document** tab.
6. Under **Data** select **Your computer** or **S3**.
7. Select **Select document** to upload your document. You can also drag-and-drop the document in the chat console in the box that says **Write a query**.

> **Note**
>
> File types: PDF, MD, TXT, DOC, DOCX, HTML, CSV, XLS, XLSX. There is a preset fixed token limit when using a file under 10MB. A text-heavy file that is smaller than 10MB can potentially be larger than the token limit.

8. Enter a custom prompt in the box that says **Write a query**. You can enter a custom prompt or use the default prompt. The loaded document and the prompt appear the bottom of the chat window.
9. Select **Run**. The response produces search results with an option **Show source chunks** that show the source material information for the answer.
10. To load a new file, select the X to delete the current file loaded into the chat window and drag and drop and new file. Enter a new prompt and select **Run**.

> **Note**
>
> Selecting a new file will wipe out previous queries and responses and will start a new session.

Sync to ingest your data sources into the knowledge base

After you create your knowledge base, you ingest the data sources into the knowledge base so that they're indexed and able to be queried. Ingestion converts the raw data in your data source into vector embeddings. It also associates the raw text and any relevant metadata that you set up for filtering to augment the querying process. Before you begin ingestion, check that your data source fulfills the following conditions:

- The Amazon S3 bucket for the data source is in the same region as the knowledge base.
- The files are in supported formats. For more information, see Set up a vector index for your knowledge base in a supported vector store.

- The files don't exceed the maximum file size of 50 MB. For more information, see Knowledge base quotas.
- If your data source contains metadata files, check the following conditions to ensure that the metadata files aren't ignored:
  - Each `.metadata.json` file shares the same name as the source file that it's associated with.
  - If the vector index for your knowledge base is in an Amazon OpenSearch Serverless vector store, check that the vector index is configured with the `faiss` engine. If the vector index is configured with the `nmslib` engine, you'll have to do one of the following:
    - Create a new knowledge base in the console and let Amazon Bedrock automatically create a vector index in Amazon OpenSearch Serverless for you.
    - Create another vector index in the vector store and select `faiss` as the **Engine**. Then create a new knowledge base and specify the new vector index.
  - If the vector index for your knowledge base is in an Amazon Aurora database cluster, check that the table for your index contains a column for each metadata property in your metadata files before starting ingestion.

> **Note**
>
> Each time you add, modify, or remove files from the S3 bucket for a data source, you must sync the data source so that it is re-indexed to the knowledge base. Syncing is incremental, so Amazon Bedrock only processes the objects in your S3 bucket that have been added, modified, or deleted since the last sync.

To learn how to ingest your data sources into your knowledge base, Select the tab corresponding to your method of choice and follow the steps.

Console

To ingest your data sources

1. Open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base** and choose your knowledge base.
3. In the **Data source** section, select **Sync** to begin data ingestion.
4. When data ingestion completes, a green success banner appears if it is successful.

5. You can choose a data source to view its **Sync history**. Select **View warnings** to see why a data ingestion job failed.

API

To ingest a data source into the vector store you configured for your knowledge base, send a StartIngestionJob request with a Agents for Amazon Bedrock build-time endpoint. Specify the `knowledgeBaseId` and `dataSourceId`.

Use the `ingestionJobId` returned in the response in a GetIngestionJob request with a Agents for Amazon Bedrock build-time endpoint to track the status of the ingestion job. In addition, specify the `knowledgeBaseId` and `dataSourceId`.

- When the ingestion job finishes, the `status` in the response is `COMPLETE`.
- The `statistics` object in the response returns information about whether ingestion was successful or not for documents in the data source.

You can also see information for all ingestion jobs for a data source by sending a ListIngestionJobs request with a Agents for Amazon Bedrock build-time endpoint. Specify the `dataSourceId` and the `knowledgeBaseId` of the knowledge base that the data is being ingested to.

- Filter for results by specifying a status to search for in the `filters` object.
- Sort by the time that the job was started or the status of a job by specifying the `sortBy` object. You can sort in ascending or descending order.
- Set the maximum number of results to return in a response in the `maxResults` field. If there are more results than the number you set, the response returns a `nextToken` that you can send in another ListIngestionJobs request to see the next batch of jobs.

Test a knowledge base in Amazon Bedrock

After you set up your knowledge base, you can test its behavior by sending queries and seeing the responses. You can also set query configurations to customize information retrieval. When you are satisfied with your knowledge base's behavior, you can then set up your application to query the knowledge base or attach the knowledge base to an agent.

Select a topic to learn more about it.

Topics

- [Query the knowledge base and return results or generate responses]
- [Query configurations]

Query the knowledge base and return results or generate responses

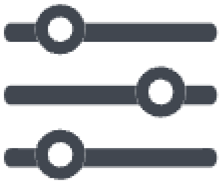
To learn how to query your knowledge base, select the tab corresponding to your method of choice and follow the steps.

Console

To test your knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base**.
3. In the **Knowledge bases** section, do one of the following actions:
   - Choose the radio button next to the knowledge base you want to test and select **Test knowledge base**. A test window expands from the right.
   - Choose the knowledge base that you want to test. A test window expands from the right.
4. Select or clear **Generate responses for your query** depending on your use case.
   - To return information retrieved directly from your knowledge base, turn off **Generate responses**. Amazon Bedrock will return text chunks from your data sources that are relevant to the query.
   - To generate responses based on information retrieved from your knowledge base, turn on **Generate responses**. Amazon Bedrock will generate responses based on your data sources and cites the information it provides with footnotes.
5. If you turn on **Generate responses**, choose **Select model** to choose a model to use for response generation. Then select **Apply**.

6. (Optional) Select the configurations icon

   ()

   to open up **Configurations**. You can modify the following configurations:

   - **Search type** – Specify how your knowledge base is queried. For more information, see Search type.
   - **Maximum number of retrieved results** – Specify the maximum number of results to retrieve. For more information, see Maximum number of retrieved results.
   - **Filters** – Specify up to 5 filter groups and up to 5 filters within each group to use with the metadata for your files. For more information, see Metadata and filtering.
   - **Knowledge base prompt template** – If you turn on **Generate responses**, you can replace the default prompt template with your own to customize the prompt that's sent to the model for response generation. For more information, see Knowledge base prompt template.
7. Enter a query in the text box in the chat window and select **Run** to return responses from the knowledge base.
8. You can examine the response in the following ways.

   - If you didn't generate responses, the text chunks are returned directly in order of relevance.
   - If you generated responses, select a footnote to see an excerpt from the cited source for that part of the response. Choose the link to navigate to the S3 object containing the file.
   - To see details about the chunks cited for each footnote, select **Show source details**. You can carry out the following actions in the **Source details** pane:
     - To see the configurations that you set for query, expand **Query configurations**.
     - To view details about a source chunk, expand it by choosing the right arrow

       ()

       next to it. You can see the following information:
       - The raw text from the source chunk. To copy this text, choose the copy icon

         ().

To navigate to the S3 object containing the file, choose the external link icon ( ).

- The metadata associated with the source chunk. The attribute keys and values are defined in the `.metadata.json` file that's associated with the source document. For more information, see Metadata file requirements.

Chat options

1. If you are generating responses, you can select **Change model** to use a different model for response generation. If you change the model, the text in the chat window will be completely cleared.
2. Switch between generating responses for your query and returning direct quotations by selecting or clearing **Generate responses**. If you change the setting, the text in the chat window will be completely cleared.
3. To clear the chat window, select the broom icon ( ).
4. To copy all the output in the chat window, select the copy icon ( ).

API

Retrieve

To query a knowledge base and only return relevant text from data sources, send a Retrieve request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint.

The following table briefly describes the parameters and request body (for detailed information and the request structure, see the see Retrieve request syntax:

Variable	Required?	Use case
knowledgeBaseId	Yes	To specify the knowledge base to query

Variable	Required?	Use case
retrievalQuery	Yes	Contains a `text` field to specify the query
nextToken	No	To return the next batch of responses
retrievalConfiguration	No	To include query configurations for customizing the vector search.

The following table briefly describes the response body (for detailed information and the response structure, see the Retrieve response syntax:

Variable	Use case
retrievalResults	Contains the source chunks, Amazon S3 location of the source, and a relevancy `score` for the chunk.
nextToken	To use in another request to return the next batch of results.

RetrieveAndGenerate

To query a knowledge base and use a foundation model to generate responses based off the results from the data sources, send a RetrieveAndGenerate request with a Agents for Amazon Bedrock runtime endpoint.

The following table briefly describes the parameters and request body (for detailed information and the request structure, see the RetrieveAndGenerate request syntax):

Variable	Required?	Use case
input	Yes	Contains a `text` field to specify the query

Variable	Required?	Use case
retrieveAndGenerateConfiguration	Yes	For specifying the knowledge base to query, the model to use for response generation, and optional query configurations.
sessionId	No	Use the same value to continue the same session and maintain information
sessionConfiguration	No	To include a KMS key for encryption of the session

The following table briefly describes the response body (for detailed information and the response structure, see the Retrieve response syntax):

Variable	Use case
citations	Contains parts of the generated response in each object within the `generatedResponsePart`, and the source chunk in the `content` object and the Amazon S3 location of the source in the `location` object of the `retrievedReferences` object.
output	Contains the whole generated response.
sessionId	Contains the ID of the session, which you can reuse in another request to maintain the same conversation

> **Note**
>
> If you receive an error that the prompt exceeds the character limit while generating responses, you can shorten the prompt in the following ways:
>
> - Reduce the maximum number of retrieved results (this shortens what is filled in for the $search_results$ placeholder in the Knowledge base prompt template).
> - Recreate the data source with a chunking strategy that uses smaller chunks (this shortens what is filled in for the $search_results$ placeholder in the Knowledge base prompt template).
> - Shorten the prompt template.
> - Shorten the user query (this shortens what is filled in for the $query$ placeholder in the Knowledge base prompt template).

Query configurations

You can modify configurations when you query the knowledge base to customize retrieval and response generation. To learn more about a configuration and how to modify it in the console or the API, select from the following topics.

Search type

The search type defines how data sources in the knowledge base are queried. The following search types are possible:

- **Default** – Amazon Bedrock decides the search strategy for you.
- **Hybrid** – Combines searching vector embeddings (semantic search) with searching through the raw text. Hybrid search is currently only supported for Amazon OpenSearch Serverless vector stores that contain a filterable text field. If you use a different vector store or your Amazon OpenSearch Serverless vector store doesn't contain a filterable text field, the query uses semantic search.
- **Semantic** – Only searches vector embeddings.

To learn how to define the search type, select the tab corresponding to your method of choice and follow the steps.

Console

Follow the console steps at Query the knowledge base and return results or generate responses. When you open the **Configurations** pane, you'll see the following options for **Search type**:

- **Default** – Amazon Bedrock decides which search strategy is best-suited for your vector store configuration.
- **Hybrid** – Amazon Bedrock queries the knowledge base using both the vector embeddings and the raw text. This option is only available if you're using an Amazon OpenSearch Serverless vector store configured with a filterable text field.
- **Semantic** – Amazon Bedrock queries the knowledge base using its vector embeddings.

API

When you make a Retrieve or RetrieveAndGenerate request, include a `retrievalConfiguration` field, mapped to a KnowledgeBaseRetrievalConfiguration object. To see the location of this field, refer to the Retrieve and RetrieveAndGenerate request bodies in the API reference.

The following JSON object shows the minimal fields required in the KnowledgeBaseRetrievalConfiguration object to set search type configurations:

```
"retrievalConfiguration": {
    "vectorSearchConfiguration": {
        "overrideSearchType": "HYBRID | SEMANTIC"
    }
}
```

Specify the search type in the `overrideSearchType` field. You have the following options:

- If you don't specify a value, Amazon Bedrock decides which search strategy is best-suited for your vector store configuration.
- **HYBRID** – Amazon Bedrock queries the knowledge base using both the vector embeddings and the raw text. This option is only available if you're using an Amazon OpenSearch Serverless vector store configured with a filterable text field.
- **SEMANTIC** – Amazon Bedrock queries the knowledge base using its vector embeddings.

Maximum number of retrieved results

When you query a knowledge base, Amazon Bedrock returns up to five results in the response by default. Each result corresponds to a source chunk. To modify the maximum number of results to return, select the tab corresponding to your method of choice and follow the steps.

Console

Follow the console steps at Query the knowledge base and return results or generate responses. In the **Configurations** pane, expand the **Maximum number of retrieved results**.

API

When you make a Retrieve or RetrieveAndGenerate request, include a `retrievalConfiguration` field, mapped to a KnowledgeBaseRetrievalConfiguration object. To see the location of this field, refer to the Retrieve and RetrieveAndGenerate request bodies in the API reference.

The following JSON object shows the minimal fields required in the KnowledgeBaseRetrievalConfiguration object to set the maximum number of results to return:

```
"retrievalConfiguration": {
    "vectorSearchConfiguration": {
        "numberOfResults": number
    }
}
```

Specify the maximum number of retrieved results (see the `numberOfResults` field in KnowledgeBaseRetrievalConfiguration for the range of accepted values) to return in the `numberOfResults` field.

Metadata and filtering

Your data sources can includes metadata files associated with the source documents. A metadata file contains attributes as key-value pairs that you define for a source document. For more information about creating metadata for your data source files, see Add metadata to your files to allow for filtering. To use filters during knowledge base query, check that your knowledge base fulfills the following requirements:

- The Amazon S3 bucket containing your data source includes at least one `.metadata.json` file with the same name as the source document it's associated with.

- If your knowledge base's vector index is in an Amazon OpenSearch Serverless vector store, check that the vector index is configured with the `faiss` engine. If the vector index is configured with the `nmslib` engine, you'll have to do one of the following:
  - Create a new knowledge base in the console and let Amazon Bedrock automatically create a vector index in Amazon OpenSearch Serverless for you.
  - Create another vector index in the vector store and select `faiss` as the **Engine**. Then Create a new knowledge base and specify the new vector index.

You can use the following filtering operators when modifying query configurations for filtering:

Filtering operators

Operator	Console	API filter name	Supported attribute data types	Filtered results
Equals	=	equals	string, number, boolean	Attribute matches the value you provide
Not equals	!=	notEquals	string, number, boolean	Attribute doesn't match the value you provide
Greater than	>	greaterThan	number	Attribute is greater than the value you provide
Greater than or equals	>=	greaterThanOrEquals	number	Attribute is greater than or equal to the value you provide

Operator	Console	API filter name	Supported attribute data types	Filtered results
Less than	<	lessThan	number	Attribute is less than the value you provide
Less than or equals	<=	lessThanOrEquals	number	Attribute is less than or equal to the value you provide
In	:	in	string list	Attribute is in the list you provide
Not in	!:	notIn	string list	Attribute isn't in the list you provide
Starts with	^	startsWith	string	Attribute starts with the string you provide (only supported for Amazon OpenSearch Serverless vector stores)

To combine filtering operators, you can use the following logical operators:

Logical operators

Operator	Console	API filter field name	Filtered results
And	and	andAll	Results fulfill all of the filtering expressions in the group
Or	or	orAll	Results fulfill at least one of the filtering expressions in the group

To learn how to filter results using metadata, select the tab corresponding to your method of choice and follow the steps.

Console

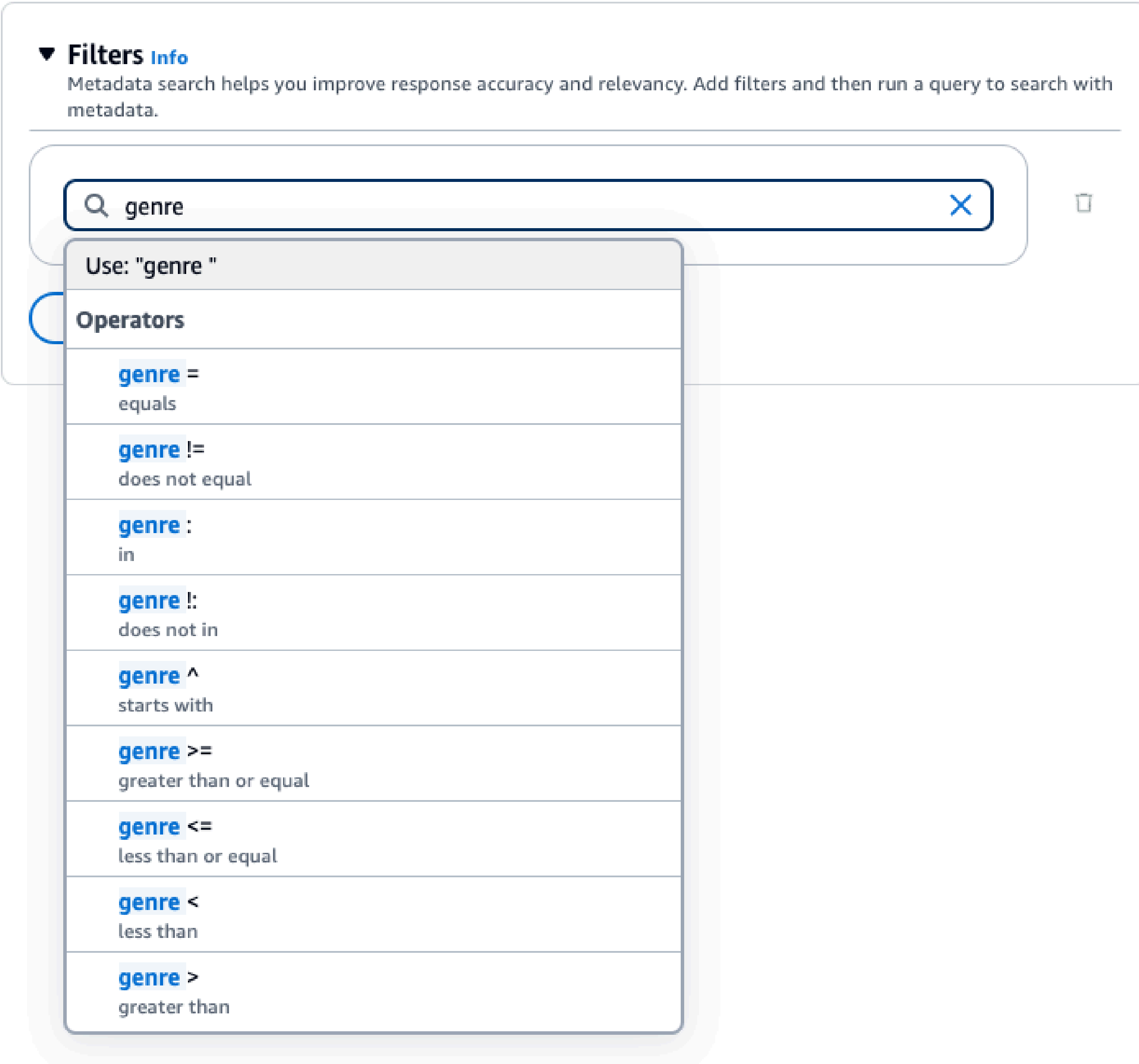
Follow the console steps at Query the knowledge base and return results or generate responses. When you open the **Configurations** pane, you'll see a **Filters section**. The following procedures describe different use cases:

- To add a filter, create a filtering expression by entering a metadata attribute, filtering operator, and value in the box. Separate each part of the expression with a whitespace. Press **Enter** to add the filter.

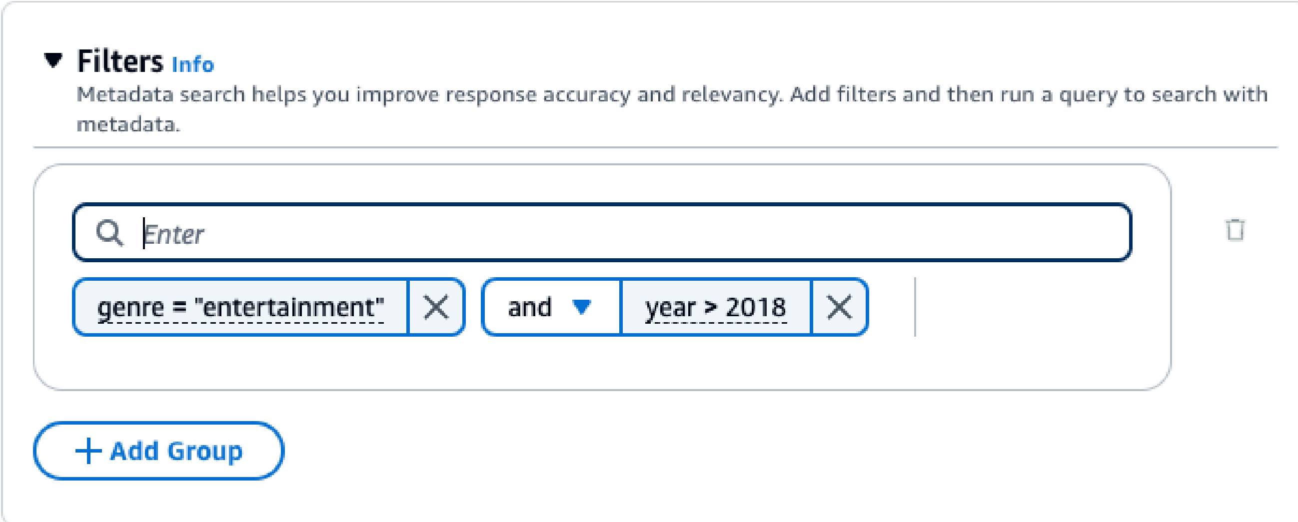
  For a list of accepted filtering operators, see the **Filtering operators** table above. You can also see a list of filtering operators when you add a whitespace after the metadata attribute.

  > **Note**
  >
  > You must surround strings with quotation marks.

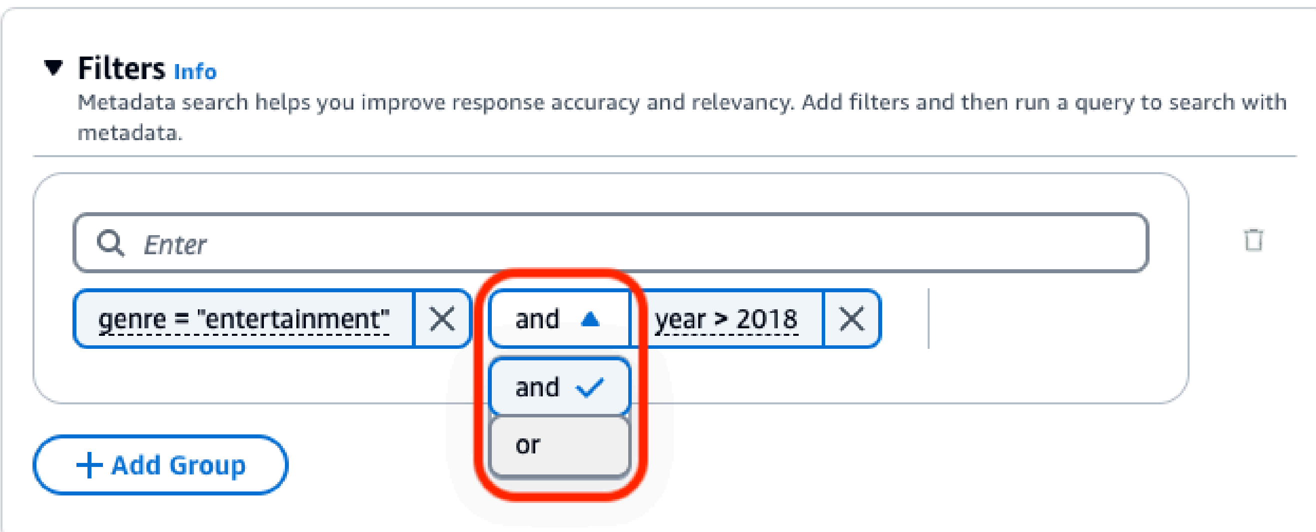
For example, you can filter for results from source documents that contain a `genre` metadata attribute whose value is `"entertainment"` by adding the following filter: **`genre = "entertainment"`**.



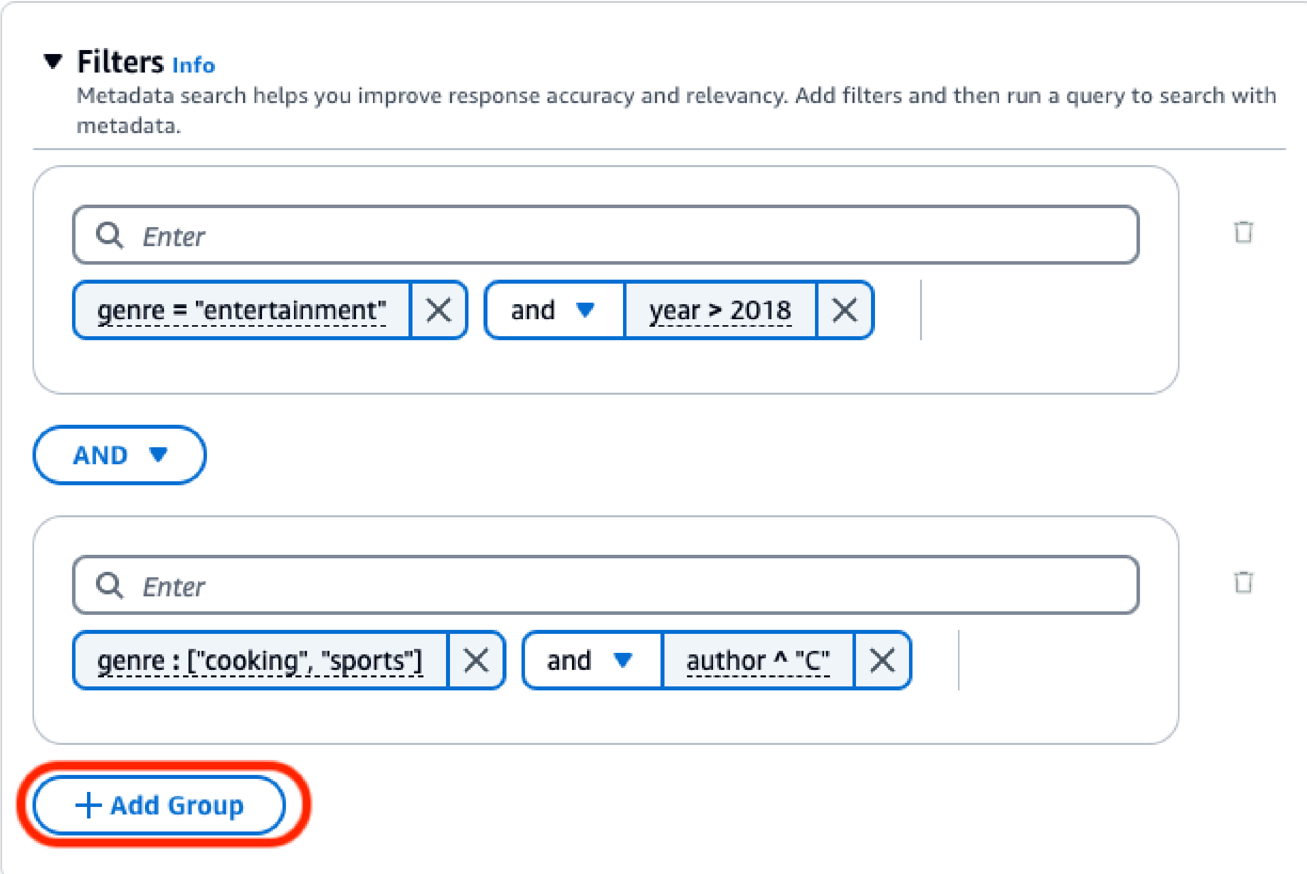
- To add another filter, enter another filtering expression in the box and press **Enter**. You can add up to 5 filters in the group.



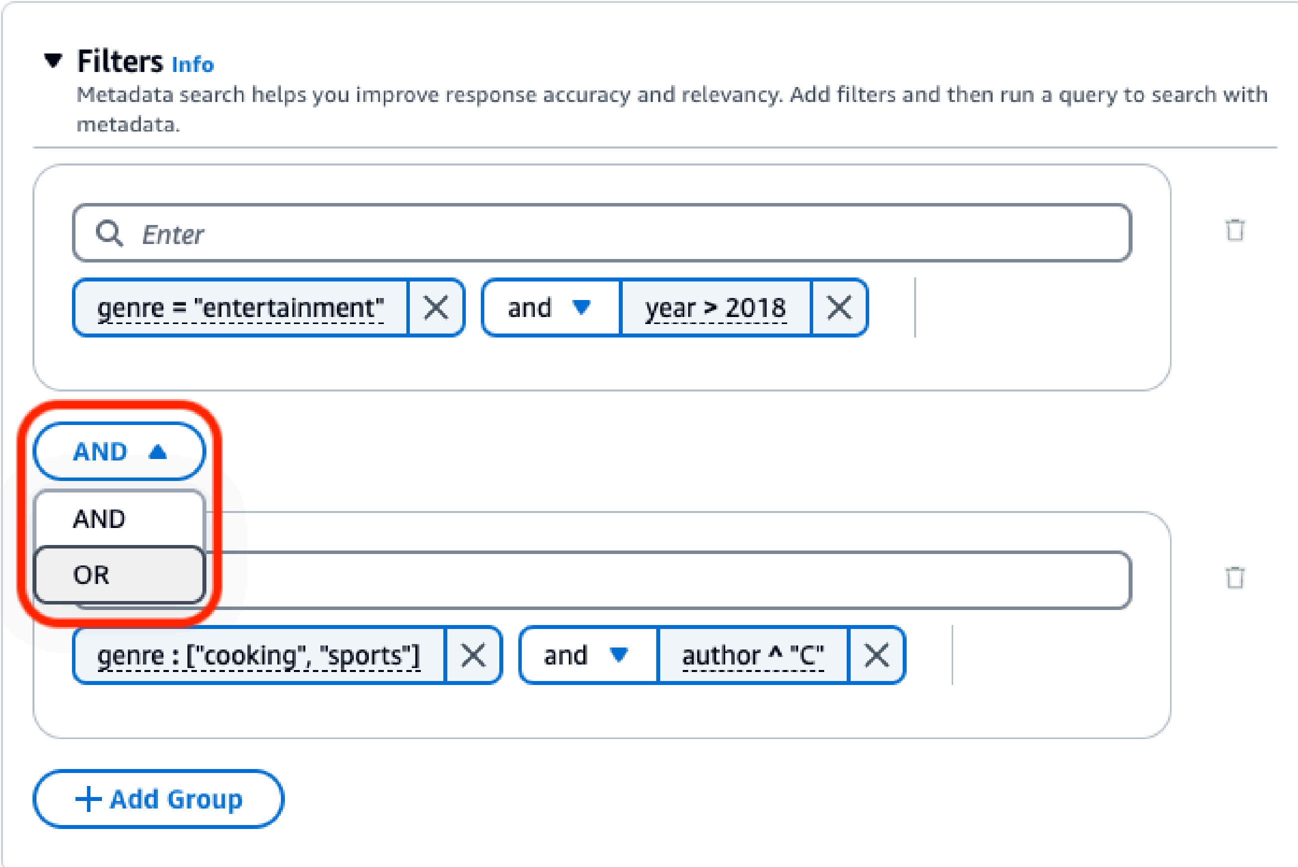
- By default, the query will return results that fulfill all the filtering expressions you provide. To return results that fulfill at least one of the filtering expressions, choose the **and** dropdown menu between any two filtering operations and select **or**.



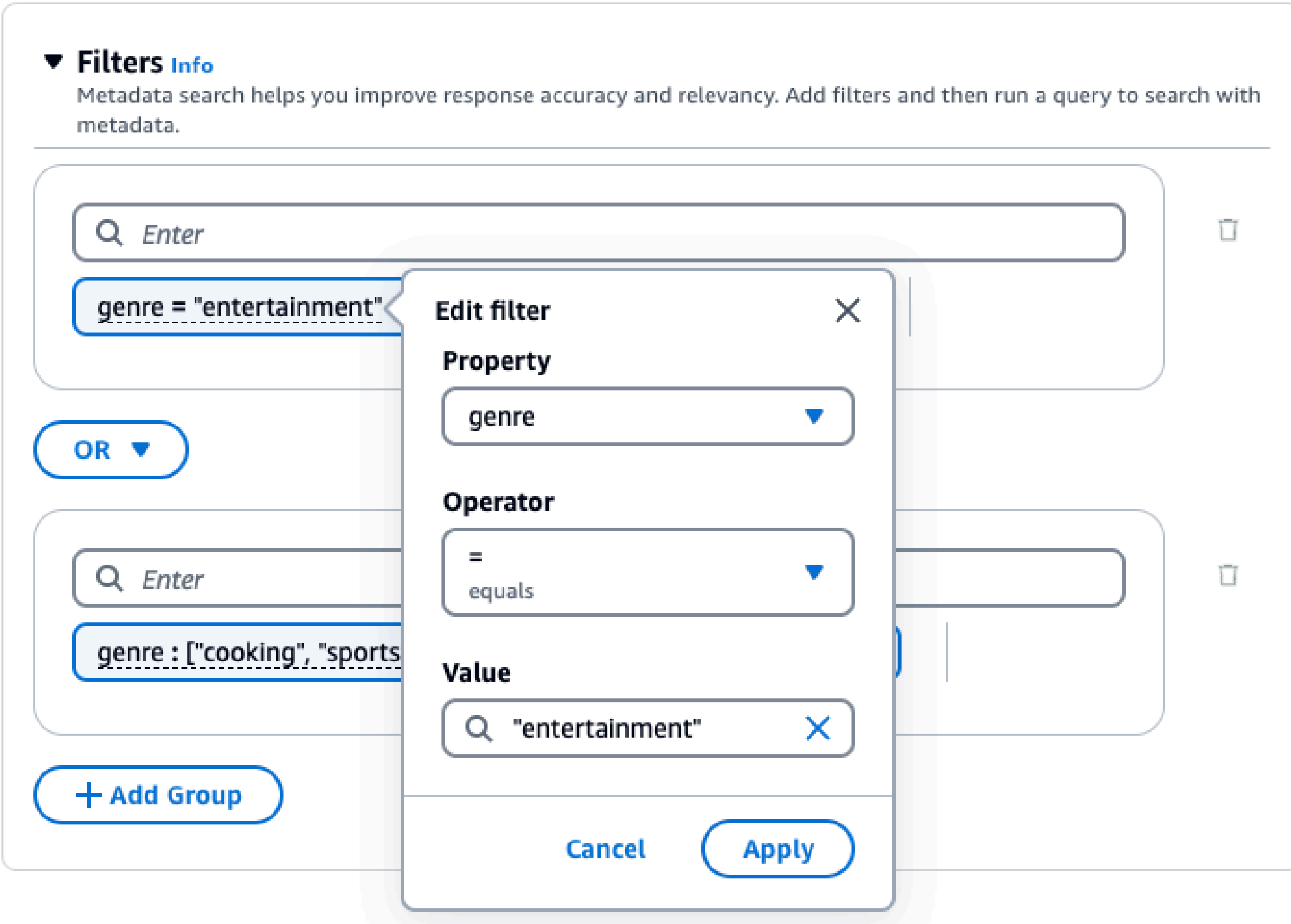
- To combine different logical operators, select **+ Add Group** to add a filter group. Enter filtering expressions in the new group. You can add up to 5 filter groups.



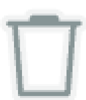
- To change the logical operator used between all the filtering groups, choose the **AND** dropdown menu between any two filter groups and select **OR**.



- To edit a filter, select it, modify the filtering operation, and choose **Apply**.



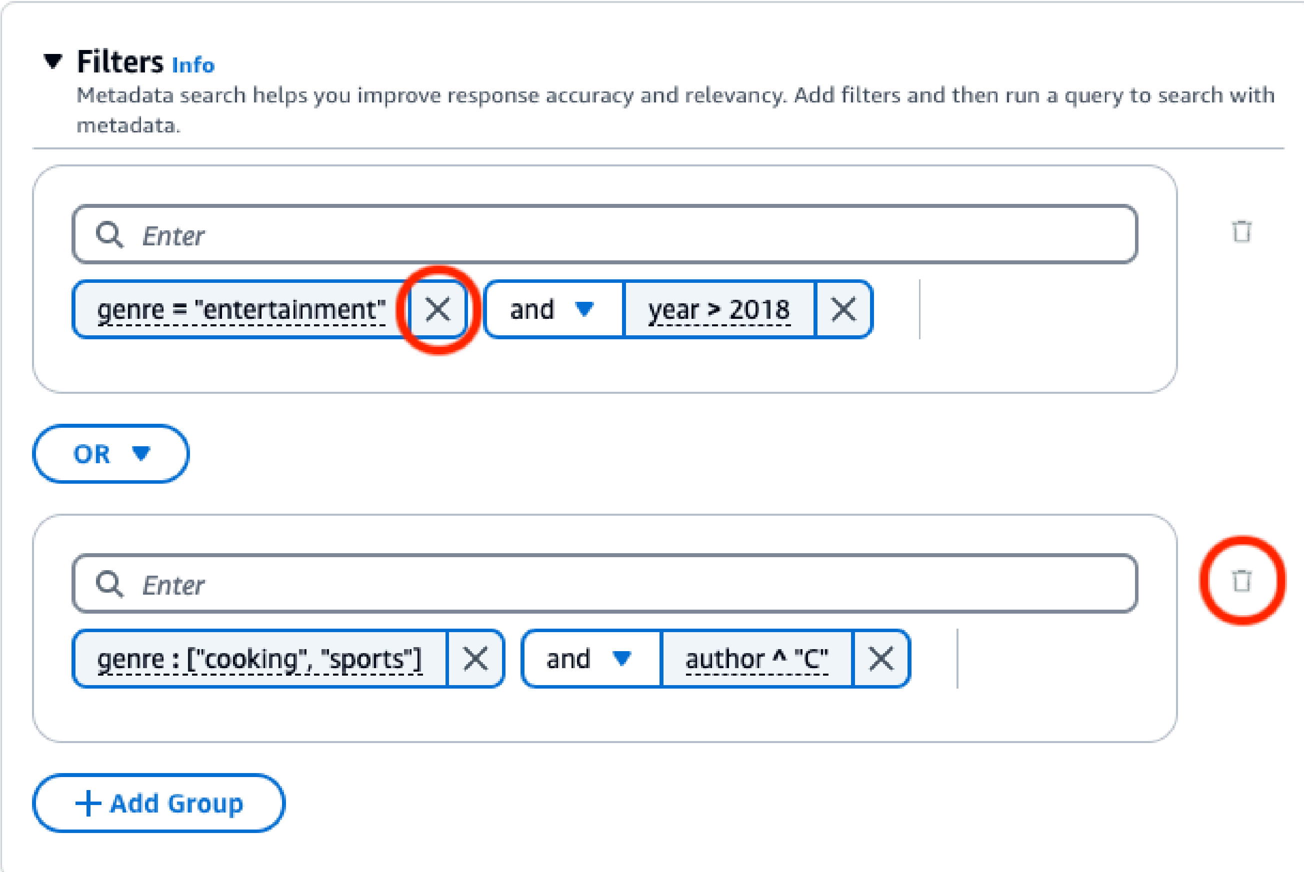
- To remove a filter group, choose the trash can icon

  ()

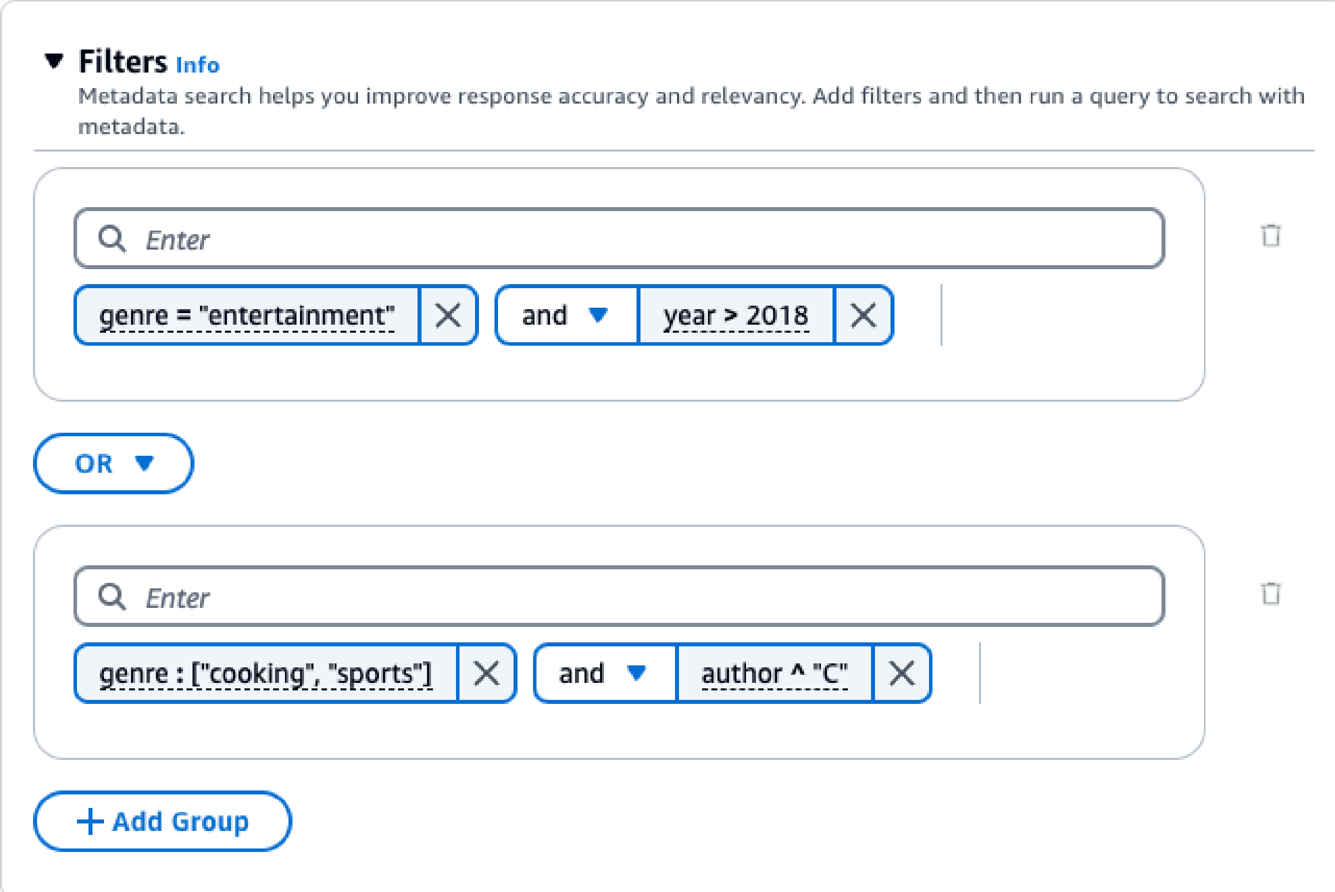
  next to the group. To remove a filter, choose the delete icon

  ()

  next to the filter.



The following image shows an example filter configuration that returns all documents written after **`2018`** whose genre is **`"entertainment"`**, in addition to documents whose genre is **`"cooking"`** or **`"sports"`** and whose author starts with **`"C"`**.



API

When you make a Retrieve or RetrieveAndGenerate request, include a `retrievalConfiguration` field, mapped to a KnowledgeBaseRetrievalConfiguration object. To see the location of this field, refer to the Retrieve and RetrieveAndGenerate request bodies in the API reference.

The following JSON objects show the minimal fields required in the KnowledgeBaseRetrievalConfiguration object to set filters for different use cases:

1. Use one filtering operator (see the **Filtering operators** table above).

```
"retrievalConfiguration": {
    "vectorSearchConfiguration": {
        "filter": {
            "<filter-type>": {
                "key": "string",
                "value": "string" | number | boolean | ["string", "string", ...]
            }
```

```
        }
    }
}
```

2. Use a logical operator (see the **Logical operators** table above) to combine up to 5.

```
"retrievalConfiguration": {
    "vectorSearchConfiguration": {
        "filter": {
            "andAll | orAll": [
                "<filter-type>": {
                    "key": "string",
                    "value": "string" | number | boolean | ["string",
 "string", ...]
                },
                "<filter-type>": {
                    "key": "string",
                    "value": "string" | number | boolean | ["string",
 "string", ...]
                },
                ...
            ]
        }
    }
}
```

3. Use a logical operator to combine up to 5 filtering operators into a filter group, and a second logical operator to combine that filter group with another filtering operator.

```
"retrievalConfiguration": {
    "vectorSearchConfiguration": {
        "filter": {
            "andAll | orAll": [
                "andAll | orAll": [
                    "<filter-type>": {
                        "key": "string",
                        "value": "string" | number | boolean | ["string",
 "string", ...]
                    },
                    "<filter-type>": {
                        "key": "string",
                        "value": "string" | number | boolean | ["string",
 "string", ...]
```

```
                    },
                    ...
                ],
                "<filter-type>": {
                    "key": "string",
                    "value": "string" | number | boolean | ["string",
 "string", ...]
                }
            ]
        }
    }
}
```

4. Combine up to 5 filter groups by embedding them within another logical operator. You can create one level of embedding.

```
"retrievalConfiguration": {
    "vectorSearchConfiguration": {
        "filter": {
            "andAll | orAll": [
                "andAll | orAll": [
                    "<filter-type>": {
                        "key": "string",
                        "value": "string" | number | boolean | ["string",
 "string", ...]
                    },
                    "<filter-type>": {
                        "key": "string",
                        "value": "string" | number | boolean | ["string",
 "string", ...]
                    },
                    ...
                ],
                "andAll | orAll": [
                    "<filter-type>": {
                        "key": "string",
                        "value": "string" | number | boolean | ["string",
 "string", ...]
                    },
                    "<filter-type>": {
                        "key": "string",
                        "value": "string" | number | boolean | ["string",
 "string", ...]
```

```
                    },
                    ...
                ]
            ]
        }
    }
}
```

The following table describes the filter types that you can use:

Field	Supported value data types	Filtered results
`equals`	string, number, boolean	Attribute matches the value you provide
`notEquals`	string, number, boolean	Attribute doesn't match the value you provide
`greaterThan`	number	Attribute is greater than the value you provide
`greaterThanOrEquals`	number	Attribute is greater than or equal to the value you provide
`lessThan`	number	Attribute is less than the value you provide
`lessThanOrEquals`	number	Attribute is less than or equal to the value you provide
`in`	list of strings	Attribute is in the list you provide
`notIn`	list of strings	Attribute isn't in the list you provide

Field	Supported value data types	Filtered results
`startsWith`	string	Attribute starts with the string you provide (only supported for Amazon OpenSearch Serverless vector stores)

To combine filter types, you can use one of the following logical operators:

Field	Maps to	Filtered results
`andAll`	List of up to 5 filter types	Results fulfill all of the filtering expressions in the group
`orAll`	List of up to 5 filter types	Results fulfill at least one of the filtering expressions in the group

For examples, see Send a query and include filters (Retrieve) and Send a query and include filters (RetrieveAndGenerate).

Knowledge base prompt template

When you query a knowledge base and request response generation, Amazon Bedrock uses a prompt template that combines instructions and context with the user query to construct the prompt that's sent to the model for response generation. You can engineer the prompt template with the following tools:

- **Prompt placeholders** – Pre-defined variables in Knowledge bases for Amazon Bedrock that are dynamically filled in at runtime during knowledge base query. In the system prompt, you'll see these placeholders surrounded by the $ symbol. The following list describes the placeholders you can use:

Variable	Replaced by	Model	Required?
$query$	The user query sent to the knowledge base.	Anthropic Claude Instant, Anthropic Claude v2.x	Yes
		Anthropic Claude 3 Sonnet	No (automatically included in model input)
$search_results$	The retrieved results for the user query.	All	Yes
$output_format_instructions$	Underlying instructions for formatting the response generation and citations. Differs by model. If you define your own formatting instructions, we suggest that you remove this placeholder. Without this placeholder, the response won't contain citations.	All	No
$current_time$	The current time.	All	No

- **XML tags** – Anthropic models support the use of XML tags to structure and delineate your prompts. Use descriptive tag names for optimal results. For example, in the default system prompt, you'll see the `<database>` tag used to delineate a database of previously asked questions). For more information, see Use XML tags in the Anthropic user guide.

For general prompt engineering guidelines, see Prompt engineering guidelines.

Select the tab corresponding to your method of choice and follow the steps.

Console

Follow the console steps at Query the knowledge base and return results or generate responses. In the test window, turn on **Generate responses**. Then, in the **Configurations** pane, expand the **Knowledge base prompt template** section.

1. Choose **Edit**.
2. Edit the system prompt in the text editor, including prompt placeholders and XML tags as necessary. To revert to the default prompt template, choose **Reset to default**.
3. When you're finished editing, choose **Save changes**. To exit without saving the system prompt, choose **Discard changes**.

API

When you make a RetrieveAndGenerate request, include a `generationConfiguration` field, mapped to a GenerationConfiguration object. To see the location of this field, refer to the RetrieveAndGenerate request body in the API reference.

The following JSON object shows the minimal fields required in the GenerationConfiguration object to set the maximum number of retrieved results to return:

```
"generationConfiguration": {
    "promptTemplate": {
        "textPromptTemplate": "string"
    }
}
```

Enter your custom prompt template in the `textPromptTemplate` field, including prompt placeholders and XML tags as necessary. For the maximum number of characters allowed in the system prompt, see the `textPromptTemplate` field in GenerationConfiguration.

Manage a data source

After you create a data source, you can view details about it, update it, or delete it.

View information about a data source

You can view information about your data source and its sync history. Select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a data source

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base**.
3. In the **Data source** section, select the data source for which you want to view details.
4. The **Data source overview** contains details about the data source.
5. The **Sync history** contains details about when the data source was synced. To see reasons for why a sync event failed, select a sync event and choose **View warnings**.

API

To get information about a data source, send a GetDataSource request with a Agents for Amazon Bedrock build-time endpoint and specify the `dataSourceId` and the `knowledgeBaseId` of the knowledge base that it belongs to.

To list information about a knowledge base's data sources, send a ListDataSources request with a Agents for Amazon Bedrock build-time endpoint and specify the ID of the knowledge base.

- To set the maximum number of results to return in a response, use the `maxResults` field.
- If there are more results than the number you set, the response returns a `nextToken`. You can use this value in another `ListDataSources` request to see the next batch of results.

To get information a sync event for a data source, send a GetIngestionJob request with a Agents for Amazon Bedrock build-time endpoint. Specify the `dataSourceId`, `knowledgeBaseId`, and `ingestionJobId`.

To list the sync history for a data source in a knowledge base, send a ListIngestionJobs request with a Agents for Amazon Bedrock build-time endpoint. Specify the ID of the knowledge base and data source. You can set the following specifications.

- Filter for results by specifying a status to search for in the `filters` object.
- Sort by the time that the job was started or the status of a job by specifying the `sortBy` object. You can sort in ascending or descending order.
- Set the maximum number of results to return in a response in the `maxResults` field. If there are more results than the number you set, the response returns a `nextToken` that you can send in another ListIngestionJobs request to see the next batch of jobs.

Update a data source

You can update a data source in the following ways:

- Add, change, or remove files from the S3 bucket that contains the files for the data source.
- Change the name or S3 bucket for the data source, or the KMS key to use for encrypting transient data during data ingestion.

Each time you add, modify, or remove files from the S3 bucket for a data source, you must sync the data source so that it is re-indexed to the knowledge base. Syncing is incremental, so Amazon Bedrock only processes the objects in your S3 bucket that have been added, modified, or deleted since the last sync. Before you begin ingestion, check that your data source fulfills the following conditions:

- The files are in supported formats. For more information, see Set up a vector index for your knowledge base in a supported vector store.
- The files don't exceed the maximum file size of 50 MB. For more information, see Knowledge base quotas.
- If your data source contains metadata files, check the following conditions to ensure that the metadata files aren't ignored:
  - Each `.metadata.json` file shares the same name as the source file that it's associated with.
  - If the vector index for your knowledge base is in an Amazon OpenSearch Serverless vector store, check that the vector index is configured with the `faiss` engine. If the vector index is configured with the `nmslib` engine, you'll have to do one of the following:
    - Create a new knowledge base in the console and let Amazon Bedrock automatically create a vector index in Amazon OpenSearch Serverless for you.
    - Create another vector index in the vector store and select `faiss` as the **Engine**. Then create a new knowledge base and specify the new vector index.

- If the vector index for your knowledge base is in an Amazon Aurora database cluster, check that the table for your index contains a column for each metadata property in your metadata files before starting ingestion.

To learn how to update a data source, select the tab corresponding to your method of choice and follow the steps.

Console

To update a data source

1. (Optional) Make the necessary changes to the files in the S3 bucket that contains the files for the data source.
2. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
3. From the left navigation pane, select **Knowledge base**.
4. In the **Data source** section, select the radio button next to the data source that you want to sync.
5. (Optional) Choose **Edit**, change any configurations necessary, and select **Submit**.
6. Choose **Sync**.
7. A green banner appears when the sync is complete and the **Status** becomes **Ready**.

API

To update a data source

1. (Optional) Make the necessary changes to the files in the S3 bucket that contains the files for the data source.
2. (Optional) Send an UpdateDataSource request with a Agents for Amazon Bedrock build-time endpoint, changing the necessary configurations and specifying the same configurations you don't want to change.

   > **Note**
   >
   > You can't change the `chunkingConfiguration`. Send the request with the existing `chunkingConfiguration`.

3. Send a StartIngestionJob request with a Agents for Amazon Bedrock build-time endpoint, specifying the `dataSourceId` and the `knowledgeBaseId`.

Delete a data source

If you no longer need a data source, you can delete it. Select the tab corresponding to your method of choice and follow the steps.

Console

To delete a data source

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base**.
3. In the **Data source** section, select the radio button next to the data source that you want to delete.
4. Choose **Delete**.
5. A green banner appears when the data source is successfully deleted.

API

To delete a data source from a knowledge base, send a DeleteDataSource request, specifying the `dataSourceId` and `knowledgeBaseId`.

Manage a knowledge base

After you set up a knowledge base, you can view information about it, modify it, or delete it. Select the tab corresponding to your method of choice and follow the steps.

View information about a knowledge base

You can view information about a knowledge base. Select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base**.
3. To view details for a knowledge base, either select the **Name** of the source or choose the radio button next to the source and select **Edit**.
4. On the details page, you can carry out the following actions:

   - To change the details of the knowledge base, select **Edit** in the **Knowledge base overview** section.
   - To update the tags attached to the knowledge base, select **Manage tags** in the **Tags** section.
   - If you update the data source from which the knowledge base was created and need to sync the changes, select **Sync** in the **Data source** section.
   - To view the details of a data source, select a **Data source name**. Within the details, you can choose the radio button next to a sync event in the **Sync history** section and select **View warnings** to see why files in the data ingestion job failed to sync.
   - To manage the embeddings model used for the knowledge base, select **Edit Provisioned Throughput**.
   - Select **Save changes** when you are finished editing.

API

To get information about a knowledge base, send a GetKnowledgeBase request with a Agents for Amazon Bedrock build-time endpoint, specifying the `knowledgeBaseId`.

To list information about your knowledge bases, send a ListKnowledgeBases request with a Agents for Amazon Bedrock build-time endpoint. You can set the maximum number of results to return in a response. If there are more results than the number you set, the response returns a `nextToken`. You can use this value in the `nextToken` field of another ListKnowledgeBases request to see the next batch of results.

Update a knowledge base

Console

To update a knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Knowledge base**.
3. Select a knowledge base to view details about it, or choose the radio button next to the knowledge base and select **Edit**.
4. You can modify the knowledge base in the following ways.
   - Change configurations for the knowledge base by choosing **Edit** in the **Knowledge base overview** section.
   - Change the tags attached to the knowledge base by choosing **Manage tags** in the **Tags** section
   - Manage the data source in the **Data source** section. For more information, see Manage a data source.
5. Select **Save changes** when you are finished editing.

API

To update a knowledge base, send an UpdateKnowledgeBase request with a Agents for Amazon Bedrock build-time endpoint. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same.

Delete a knowledge base

If you no longer need a knowledge base, you can delete it. When you delete a knowledge base, you should also carry out the following actions to fully delete all resources associated with the knowledge base.

- Dissociate the knowledge base from any agents it is associated with.

- The underlying data that was indexed from your knowledge base remains in the vector store you set up and can still be retrieved. To delete the data, you also need to delete the vector index containing the data embeddings.

Select the tab corresponding to your method of choice and follow the steps.

Console

To delete a knowledge base

1. Before the following steps, make sure to delete the knowledge base from any agents that it's associated with. To do this, carry out the following steps:
   a. From the left navigation pane, select **Agents**.
   b. Choose the **Name** of the agent that you want to delete the knowledge base from.
   c. A red banner appears to warn you to delete the reference to the knowledge base, which no longer exists, from the agent.
   d. Select the radio button next to the knowledge base that you want to remove. Select **More** and then choose **Delete**.
2. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
3. From the left navigation pane, select **Knowledge base**.
4. Choose a knowledge base or select the radio button next to a knowledge base. Then choose **Delete**.
5. Review the warnings for deleting a knowledge base. If you accept these conditions, enter **`delete`** in the input box and select **Delete** to confirm.
6. To fully delete the vector embeddings for your knowledge base, you need to delete the vector index containing the data embeddings.

API

Before deleting a knowledge base, disassociate the knowledge base from any agents that it is associated with by making a DisassociateAgentKnowledgeBase request with a Agents for Amazon Bedrock build-time endpoint.

To delete the knowledge base, send a DeleteKnowledgeBase request with a Agents for Amazon Bedrock build-time endpoint.

To fully delete the vector embeddings for your knowledge base, you need to delete the vector index containing the data embeddings.

Deploy a knowledge base

To deploy a knowledge base in your application, set it up to make Retrieve or RetrieveAndGenerate requests to the knowledge base. To see how to use these API operations, select the API tab in Test a knowledge base in Amazon Bedrock.

You can also associate the knowledge base with an agent and the agent will invoke it when necessary during orchestration. For more information, see Agents for Amazon Bedrock. Select the tab corresponding to your method of choice and follow the steps.

Console

To associate a knowledge base with an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Agents**.
3. Choose the agent to which you want to add a knowledge base.
4. In the **Working draft** section, choose **Working draft**.
5. In the **Knowledge bases** section, select **Add**.
6. Choose a knowledge base from the dropdown list under **Select knowledge base** and specify the instructions for the agent regarding how it should interact with the knowledge base and return results.

To dissociate a knowledge base with an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Agents**.
3. Choose the agent to which you want to add a knowledge base.

4. In the **Working draft** section, choose **Working draft**.
5. In the **Knowledge bases** section, choose a knowledge base.
6. Select **Delete**.

API

To associate a knowledge base with an agent, send an AssociateAgentKnowledgeBase request.

- Include a detailed `description` to provide instructions for how the agent should interact with the knowledge base and return results.
- Set the `knowledgeBaseState` to `ENABLED` to allow the agent to query the knowledge base.

You can update an knowledge base that is associated with an agent by sending an UpdateAgentKnowledgeBase request. For example, you might want to set the `knowledgeBaseState` to `ENABLED` to troubleshoot an issue. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same.

To dissociate a knowledge base with an agent, send a DisassociateAgentKnowledgeBase request.

Agents for Amazon Bedrock

Agents for Amazon Bedrock offers you the ability to build and configure autonomous agents in your application. An agent helps your end-users complete actions based on organization data and user input. Agents orchestrate interactions between foundation models (FMs), data sources, software applications, and user conversations. In addition, agents automatically call APIs to take actions and invoke knowledge bases to supplement information for these actions. Developers can save weeks of development effort by integrating agents to accelerate the delivery of generative artificial intelligence (generative AI) applications .

With agents, you can automate tasks for your customers and answer questions for them. For example, you can create an agent that helps customers process insurance claims or an agent that helps customers make travel reservations. You don't have to provision capacity, manage infrastructure, or write custom code. Amazon Bedrock manages prompt engineering, memory, monitoring, encryption, user permissions, and API invocation.

Agents perform the following tasks:

- Extend foundation models to understand user requests and break down the tasks that the agent must perform into smaller steps.
- Collect additional information from a user through natural conversation.
- Take actions to fulfill a customer's request by making API calls to your company systems.
- Augment performance and accuracy by querying data sources.

To use an agent, you perform the following steps:

1. (Optional) Create a knowledge base to store your private data in that database. For more information, see Knowledge bases for Amazon Bedrock.
2. Configure an agent for your use case and add actions the agent can perform. To define how the agent handles the actions, write Lambda functions in a programming language of your choice.
3. Associate a knowledge base with the agent to augment the agent's performance. For more information, see Create an agent in Amazon Bedrock.
4. (Optional) To customize the agent's behavior to your specific use-case, modify prompt templates for the pre-processing, orchestration, knowledge base response generation, and

post-processing steps that the agent performs. For more information, see Advanced prompts in Amazon Bedrock.

5. Test your agent in the Amazon Bedrock console or through API calls to the `TSTALIASID`. Modify the configurations as necessary. Use traces to examine your agent's reasoning process at each step of its orchestration. For more information, see Test an Amazon Bedrock agent and Trace events in Amazon Bedrock.
6. When you have sufficiently modified your agent and it's ready to be deployed to your application, create an alias to point to a version of your agent. For more information, see Deploy an Amazon Bedrock agent.
7. Set up your application to make API calls to your agent alias.
8. Iterate on your agent and create more versions and aliases as necessary.

Topics

- How Agents for Amazon Bedrock works
- Supported regions and models for Agents for Amazon Bedrock
- Prerequisites for Agents for Amazon Bedrock
- Create an agent in Amazon Bedrock
- Create an action group for an Amazon Bedrock agent
- Associate a knowledge base with an Amazon Bedrock agent
- Test an Amazon Bedrock agent
- Manage an Amazon Bedrock agent
- Customize an Amazon Bedrock agent
- Deploy an Amazon Bedrock agent

How Agents for Amazon Bedrock works

Agents for Amazon Bedrock consists of the following two main sets of API operations to help you set up and run an agent:

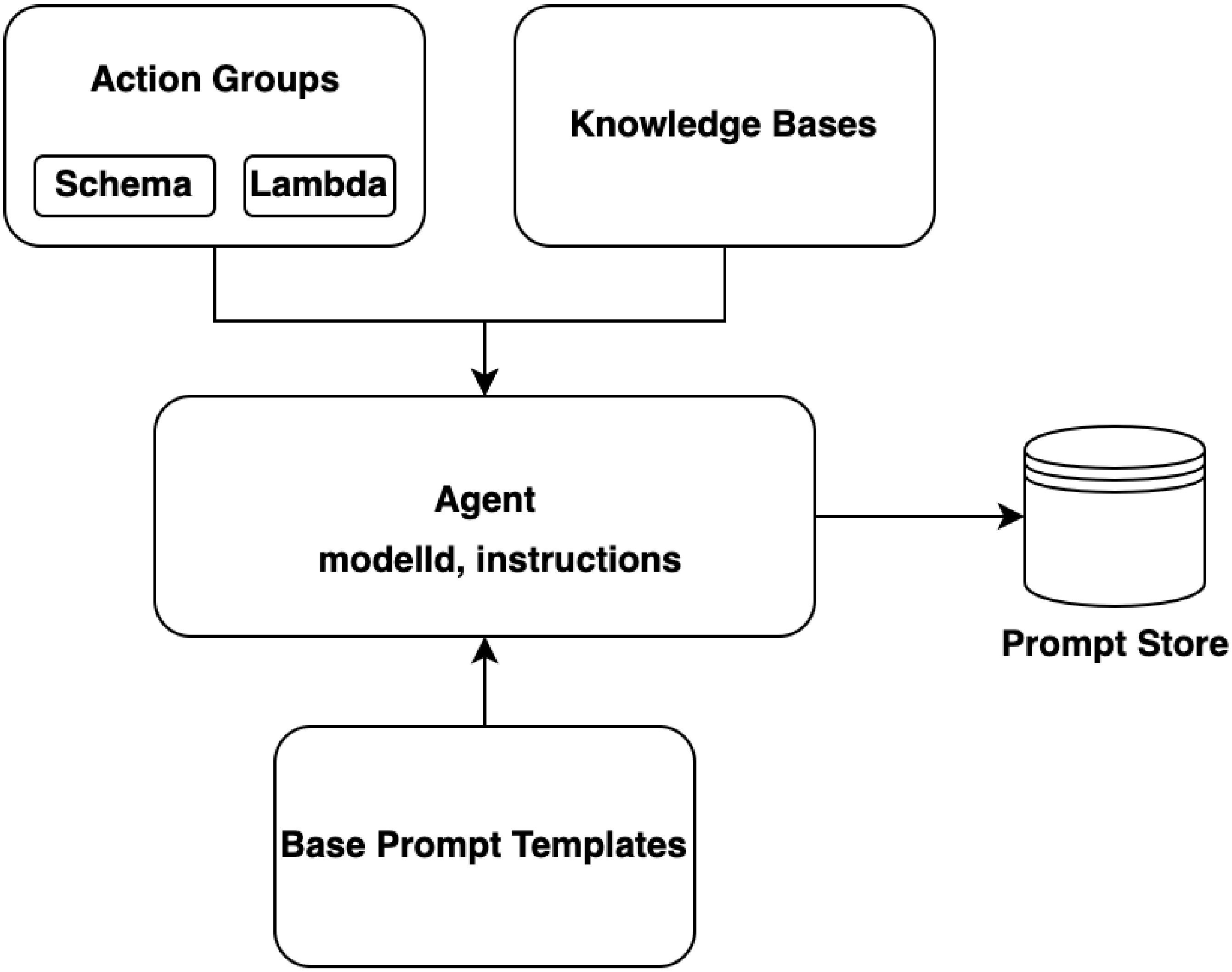
- Build-time API operations to create, configure, and manage your agents and their related resources
- Runtime API operations to invoke your agent with user input and to initiate orchestration to carry out a task.

Build-time configuration

An agent consists of the following components:

- **Foundation model** – You choose a foundation model (FM) that the agent invokes to interpret user input and subsequent prompts in its orchestration process. The agent also invokes the FM to generate responses and follow-up steps in its process.
- **Instructions** – You write instructions that describe what the agent is designed to do. With advanced prompts, you can further customize instructions for the agent at every step of orchestration and include Lambda functions to parse each step's output.
- **Action groups** (Optional) – You define the actions that the agent should perform through providing the following resources:
  - An OpenAPI schema to define the API operations that the agent can invoke to perform its tasks.
  - A Lambda function with the following input and output:
    - Input – The API operation and parameters identified during orchestration.
    - Output – The result of the API invocation.
- **Knowledge bases** (Optional) – Associate knowledge bases with an agent. The agent queries the knowledge base for extra context to augment response generation and input into steps of the orchestration process.
- **Prompt templates** – Prompt templates are the basis for creating prompts to be provided to the FM. Agents for Amazon Bedrock exposes the default four base prompt templates that are used during the pre-processing, orchestration, knowledge base response generation, and post-processing. You can optionally edit these base prompt templates to customize your agent's behavior at each step of its sequence. You can also turn off steps for troubleshooting purposes or if you decide that a step is unnecessary. For more information, see Advanced prompts in Amazon Bedrock.

At build-time, all these components are gathered to construct base prompts for the agent to perform orchestration until the user request is completed. With advanced prompts, you can modify these base prompts with additional logic and few-shot examples to improve accuracy for each step of agent invocation. The base prompt templates contain instructions, action descriptions, knowledge base descriptions, and conversation history, all of which you can customize to modify the agent to meet your needs. You then *prepare* your agent, which packages all the components of the agents, including security configurations. Preparing the agent brings it into a state where it can be tested in runtime. The following image shows how build-time API operations construct your agent.



Runtime process

Runtime is managed by the InvokeAgent API operation. This operation starts the agent sequence, which consists of the following three main steps.

1. **Pre-processing** – Manages how the agent contextualizes and categorizes user input and can be used to validate input.
2. **Orchestration** – Interprets the user input, invokes action groups and queries knowledge bases, and returns output to the user or as input to continued orchestration. Orchestration consists of the following steps:

a. The agent interprets the input with a foundation model and generates a *rationale* that lays out the logic for the next step it should take.

b. The agent invokes action groups and queries knowledge bases (**Knowledge base response generation**) to retrieve additional context and summarize the data to augment its generation.

c. The agent generates an output, known as an *observation*, from invoking action groups and summarizing results from knowledge bases. The agent uses the observation to augment the base prompt, which is then interpreted with a foundation model. The agent then determines if it needs to reiterate the orchestration process.

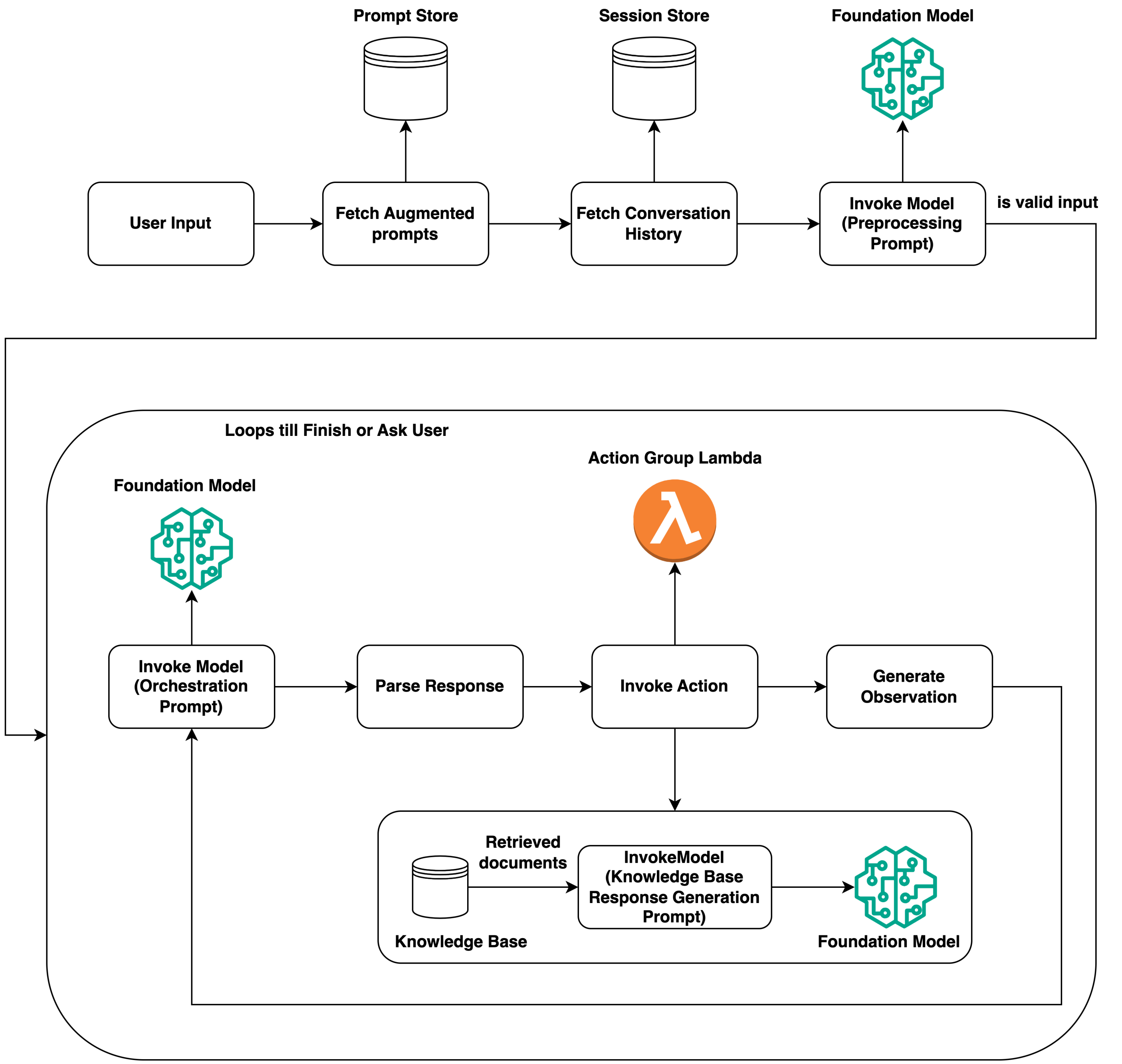
d. This loop continues until the agent returns a response to the user or until it needs to prompt the user for extra information.

During orchestration, the base prompt template is augmented with the agent instructions, action groups, and knowledge bases that you added to the agent. Then, the augmented base prompt is used to invoke the FM. The FM predicts the best possible steps and trajectory to fulfill the user input. At each iteration of orchestration, the FM predicts the API operation to invoke or the knowledge base to query.

3. **Post-processing** – The agent formats the final response to return to the user. This step is turned off by default.

When you invoke your agent, you can turn on a **trace** at runtime. With the trace, you can track the agent's rationale, actions, queries, and observations at each step of the agent sequence. The trace includes the full prompt sent to the foundation model at each step and the outputs from the foundation model, API responses, and knowledge base queries. You can use the trace to understand the agent's reasoning at each step. For more information, see Trace events in Amazon Bedrock

As the user session with the agent continues through more `InvokeAgent` requests, the conversation history is preserved. The conversation history continually augments the orchestration base prompt template with context, helping improve the agent's accuracy and performance. The following diagram shows the agent's process during runtime:



Supported regions and models for Agents for Amazon Bedrock

Agents for Amazon Bedrock is supported in the following regions:

Region
US East (N. Virginia)
US West (Oregon)

You can use Agents for Amazon Bedrock with the following models:

Model name	Model ID
Anthropic Claude Instant v1	anthropic.claude-instant-v1
Anthropic Claude v2.0	anthropic.claude-v2
Anthropic Claude v2.1	anthropic.claude-v2:1
Anthropic Claude 3 Sonnet v1	anthropic.claude-sonnet-v2:1
Anthropic Claude 3 Haiku v1	anthropic.claude-3-haiku-v1:0

Prerequisites for Agents for Amazon Bedrock

Ensure that your IAM role has the necessary permissions to perform actions related to Agents for Amazon Bedrock.

An agent uses action groups and knowledge bases to help your customer perform tasks. Following is a short description of each type of resource:

- **Action group** – Defines an action that the agent can help the user perform. Includes the APIs that can be called, how to handle the action, and how to return the response.
- **Knowledge base** – Provides a repository of information that the agent can query to answer customer queries and improve its generated responses.

Before creating an agent, review the following prerequisites and determine which ones you need to fulfill:

1. Set up an action group. For your agent to orchestrate API calls to your systems, you must add at least one action group. You can skip this prerequisite if you want to add an action group later or if you plan to have no action groups for your agent.
2. Set up a knowledge base. To improve responses to customer queries by using private data sources, you can associate at least one knowledge base. You can skip this prerequisite if you plan for your agent to have no knowledge bases associated with it
3. Create a custom AWS Identity and Access Management (IAM) service role for your agent with the proper permissions. You can skip this prerequisite if you plan to use the AWS Management Console to automatically create a service role for you.

Create an agent in Amazon Bedrock

To create an agent with Amazon Bedrock, you set up the following components:

- The configuration of the agent, which defines the purpose of the agent and indicates the foundation model (FM) that it uses to generate prompts and responses.
- Action groups that define what actions the agent is designed to perform.
- A knowledge base of data sources to augment the generative capabilities of the agent by allowing search and query.

You can minimally create an agent that only has a name. To Prepare an agent so that you can test or deploy it, you must minimally configure the following components:. Select the tab corresponding to your method of choice and follow the steps.

You should also configure at least one action group or knowledge base for the agent. If you prepare an agent with no action groups or knowledge bases, it will return responses based only on the FM and instructions and base prompt templates.

To learn how to create an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To create an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

2. Select **Agents** from the left navigation pane.
3. In the **Agents** section, choose **Create Agent**.
4. (Optional) Change the automatically generated **Name** for the agent and provide an optional **Description** for it.
5. Choose **Create**.
6. You will be taken to the **Agent Builder** for your newly created agent, where you can configure your agent.
7. You can continue to the following procedure to configure your agent or return to the Agent builder later.

To configure your agent

1. In the **Agent details** section, you can set up the following configurations:

    a. (Optional) Edit the Agent name or Agent description.

    b. For the Agent resource role, select one of the following options: a) Create and use a new service role – Let Amazon Bedrock create the service role and set up the required permissions on your behalf. or b) Use an existing service role – Use a custom role that you set up previously. The role must begin with the prefix AmazonBedrockExecutionRoleForAgents_.

    c. For **Select model**, choose **Change** to select an FM for your agent to invoke during orchestration.

    d. In **Instructions for the Agent**, enter details to tell the agent what it should do and how it should interact with users. The instructions replace the $instructions$ placeholder in the [orchestration prompt template](). Following is an example of instructions:

    ```
    You are an office assistant in an insurance agency. You are friendly and
     polite. You help with managing insurance claims and coordinating pending
     paperwork.
    ```

    e. If you expand **Additional settings**, you can modify the following configurations:

    User input – Choose whether to allow the agent to request more information from the user if it doesn't have enough information.

- If you choose **Enabled**, the agent returns an Observation reprompting the user for more information if it needs to invoke an API in an action group, but doesn't have enough information to complete the API request.
- If you choose **Disabled**, the agent doesn't request the user for additional details and instead informs the user that it doesn't have enough information to complete the task.
- **KMS key selection** – (Optional) By default, AWS encrypts agent resources with an AWS managed key. To encrypt your agent with your own customer managed key, for the KMS key selection section, select **Customize encryption settings (advanced)**. To create a new key, select **Create an AWS KMS key** and then refresh this window. To use an existing key, select a key for **Choose an AWS KMS key**.

- **Idle session timeout** – By default, if a user hasn't responded for 30 minutes in a session with a Amazon Bedrock agent, the agent no longer maintains the conversation history. Conversation history is used to both resume an interaction and to augment responses with context from the conversation. To change this default length of time, enter a number in the **Session timeout** field and choose a unit of time.

f. For the **IAM permissions** section, for **Agent resource role**, choose a service role. To let Amazon Bedrock create the service role on your behalf, choose **Create and use a new service role**. To use a custom role that you created previously, choose **Use an existing service role**. The role must begin with the prefix `AmazonBedrockExecutionRoleForAgents_`.

> **Note**
>
> The service role that Amazon Bedrock creates for you doesn't include permissions for features that are in preview. To use these features, attach the correct permissions to the service role.

g. (Optional) By default, AWS encrypts agent resources with an AWS managed key. To encrypt your agent with your own customer managed key, for the **KMS key selection** section, select **Customize encryption settings (advanced)**. To create a new key, select **Create an AWS KMS key** and then refresh this window. To use an existing key, select a key for **Choose an AWS KMS key**.

h.

i. (Optional) To associate tags with this agent, for the **Tags – optional** section, choose **Add new tag** and provide a key-value pair.

j. When you are done setting up the agent configuration, select **Next**.

2. In the **Action groups** section, you can add action groups to your agent. For more information on setting up action groups, see the section called “Create an action group”. To add an action group, choose **Create** or **Create action group**.

   a. (Optional) In the **Action group details** section, change the automatically generated **Name** and provide an optional **Description** for your action group.

   b. In the **Action group type** section, you set up what API calls the agent can make and the information that it can return to the user. Select one of the following options:

      i. **Define with function details** – Define the parameters that the

      ii. **Define with API schemas** – Define the API operations that the agent can invoke. Use an OpenAPI schema that you created or use the console text editor to create the schema. For more information on setting up an OpenAPI schema, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock

   c. In the **Action group invocation** section, you set up what the agent does after it predicts the API or function that it should invoke and the parameters that it must pass. You can do this by defining a Lambda function, which provides the business logic that's performed upon invoking the action. Choose a Lambda function that you created previously in AWS Lambda and the version of the function to use. Choose one of the following options for the Lambda function:

      - **Quick create a new Lambda function – *recommended*** – Let Amazon Bedrock create a basic Lambda function for you that you can later modify in AWS Lambda for your use case. The agent will pass the API or function that it predicts and the parameters to the Lambda function.
      - **Select an existing Lambda function** – Choose a Lambda function that you created previously in AWS Lambda and the version of the function to use. The agent will pass the API or function that it predicts and the parameters to the Lambda function.

> **Note**
>
> To allow the Amazon Bedrock service principal to access the Lambda function, attach a resource-based policy to the Lambda function to allow the Amazon Bedrock service principal to access the Lambda function.

- **Skip using a Lambda function** – Rather than passing the API or function that it predicts to the Lambda function, the agent passes these values in the InvokeAgent response.

d. Depending on your choice for the **Action group type**, you'll see one of the following sections:

- If you selected **Define with function details**, you can
- If you selected **Define with API schemas**, you'll have an **Action group schema** section with the following options:
  - To use an OpenAPI schema that you previously prepared with API descriptions, structures, and parameters for the action group, select **Select API schema** and provide a link to the Amazon S3 URI of the schema.
  - To define the OpenAPI schema with the in-line schema editor, select **Define via in-line schema editor**. A sample schema appears that you can edit.
    1. Select the format for the schema by using the dropdown menu next to **Format**.
    2. To import an existing schema from S3 to edit, select **Import schema**, provide the S3 URI, and select **Import**.
    3. To restore the schema to the original sample schema, select **Reset** and then confirm the message that appears by selecting **Reset** again.

e. To set up another action group for your agent, select **Add another action group**. When you're done with adding action groups, choose **Next**.

API

To create an agent, send a CreateAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint.

See code examples

The following list describes the fields in the request:

- Minimally, provide the following required fields:

Field	Short description
agentName	Name for the agent
agentResourceRoleArn	ARN of the service role for the agent
foundationModel	Foundation model for the agent to orchestrate with

- The following fields are optional but recommended:

Field	Short description
description	Describes what the agent does
instruction	Instructions to tell the agent what to do. Used in the orchestration prompt template.

- The following fields can be specified for security purposes:

Field	Short description
clientToken	Identifier to ensure the API request completes only once.
customerEncryptionKeyArn	ARN of a KMS key to encrypt agent resources
idleSessionTTLInSeconds	Duration after which the agent ends the session and deletes any stored information.

- To customize your agent's behavior by overriding the default prompt templates, include a `promptOverrideConfiguration` object. For more information, see [Advanced prompts in Amazon Bedrock](#).
- To attach tags to your agent, use the `Tags` field. For more information, see [Tag resources](#).

If your agent fails to be created, the CreateAgent object in the response returns a list of `failureReasons` and a list of `recommendedActions` for you to troubleshoot.

Create an action group for an Amazon Bedrock agent

An action group defines an action that the agent can help the user perform. You define an action group by specifying APIs in your systems that can be called and by writing a Lambda function to handle the action and how to return the response. To create an action group, prepare the following components:

- Set up an OpenAPI schema with the API description, structure, and parameters for the action group. You can add the API schema to the action group in one of the following ways:
  - Upload the schema that you create to an Amazon Simple Storage Service (Amazon S3) bucket.
  - Write the schema in the inline OpenAPI schema editor in the AWS Management Console when you add the action group. This option is only available after the agent that the action group belongs to has already been created.
- Create a Lambda function that defines the business logic for the action group.

To learn more about the components of an action group and how to create the action group after you set it up, select from the following topics:

Topics

Define OpenAPI schemas for your agent's action groups in Amazon Bedrock

When you create an action group in Amazon Bedrock, you must define the API operations that the agent can invoke. To define the API operations, create an OpenAPI schema in JSON or YAML format. You can create OpenAPI schema files and upload them to Amazon Simple Storage Service (Amazon S3). Alternatively, you can use the OpenAPI text editor in the console, which will validate your schema. After you create an agent, you can use the text editor when you add an action group to the agent or edit an existing action group. For more information, see [Edit an agent](#).

For more information about API schemas, see the following resources:

- For more details about OpenAPI schemas, see [OpenAPI specification](#) on the Swagger website.
- For best practices in writing API schemas, see [Best practices in API design](#) on the Swagger website.

The following is the general format of an OpenAPI schema for an action group.

```
{
    "openapi": "3.0.0",
    "paths": {
        "/path": {
            "method": {
                "description": "string",
                "operationId": "string",
                "parameters": [ ... ],
                "requestBody": { ... },
                "responses": { ... }
           }
       }
    }
}
```

The following list describes fields in the OpenAPI schema

- `openapi` – (Required) The version of OpenAPI that's being used. This value must be `"3.0.0"` or higher for the action group to work.
- `paths` – (Required) Contains relative paths to individual endpoints. Each path must begin with a forward slash (/).
- `method` – (Required) Defines the method to use.

Minimally, each method requires the following fields:

- `description` – A description of the API operation. Use this field to inform the agent when to call this API operation and what the operation does.

- `responses` – Contains properties that the agent returns in the API response. The agent uses the response properties to construct prompts, accurately process the results of an API call, and determine a correct set of steps for performing a task. The agent can use response values from one operation as inputs for subsequent steps in the orchestration.

The fields within the following two objects provide more information for your agent to effectively take advantage of your action group. For each field, set the value of the `required` field to `true` if required and to `false` if optional.

- `parameters` – Contains information about parameters that can be included in the request.
- `requestBody` – Contains the fields in the request body for the operation. Don't include this field for `GET` and `DELETE` methods.

To learn more about a structure, select from the following tabs.

responses

```
"responses": {
    "200": {
        "content": {
            "<media type>": {
                "schema": {
                    "properties": {
                        "<property>": {
                            "type": "string",
                            "description": "string"
                        },
                        ...
                    }
                }
            }
        },
    },
    ...
}
```

Each key in the `responses` object is a response code, which describes the status of the response. The response code maps to an object that contains the following information for the response:

- `content` – (Required for each response) The content of the response.
- *<media type>* – The format of the response body. For more information, see Media types on the Swagger website.
- `schema` – (Required for each media type) Defines the data type of the response body and its fields.
- `properties` – (Required if there are `items` in the schema) Your agent uses properties that you define in the schema to determine the information it needs to return to the end user in order to fulfill a task. Each property contains the following fields:
  - `type` – (Required for each property) The data type of the response field.
  - `description` – (Optional) Describes the property. The agent can use this information to determine the information that it needs to return to the end user.

parameters

```
"parameters": [
    {
        "name": "string",
        "description": "string",
        "required": boolean,
        "schema": {
            ...
        }
    },
    ...
]
```

Your agent uses the following fields to determine the information it must get from the end user to perform the action group's requirements.

- `name` – (Required) The name of the parameter.
- `description` – (Required) A description of the parameter. Use this field to help the agent understand how to elicit this parameter from the agent user or determine that it already has that parameter value from prior actions or from the user’s request to the agent.
- `required` – (Optional) Whether the parameter is required for the API request. Use this field to indicate to the agent whether this parameter is needed for every invocation or if it's optional.

- `schema` – (Optional) The definition of input and output data types. For more information, see [Data Models (Schemas)](#) on the Swagger website.

requestBody

Following is the general structure of a `requestBody` field:

```
"requestBody": {
    "required": boolean,
    "content": {
        "<media type>": {
            "schema": {
                "properties": {
                    "<property>": {
                        "type": "string",
                        "description": "string"
                    },
                    ...
                }
            }
        }
    }
}
```

The following list describes each field:

- `required` – (Optional) Whether the request body is required for the API request.
- `content` – (Required) The content of the request body.
- *`<media type>`* – (Optional) The format of the request body. For more information, see [Media types](#) on the Swagger website.
- `schema` – (Optional) Defines the data type of the request body and its fields.
- `properties` – (Optional) Your agent uses properties that you define in the schema to determine the information it must get from the end user to make the API request. Each property contains the following fields:
  - `type` – (Optional) The data type of the request field.
  - `description` – (Optional) Describes the property. The agent can use this information to determine the information it needs to return to the end user.

Example API schema

The following example API schema defines a group of API operatins that help handle insurance claims. Three APIs are defined as follows:

- `getAllOpenClaims` – Your agent can use the `description` field to determine that it should call this API operation if a list of open claims is needed. The `properties` in the `responses` specify to return the ID and the policy holder and the status of the claim. The agent returns this information to the agent user or uses some or all of the response as input to subsequent API calls.
- `identifyMissingDocuments` – Your agent can use the `description` field to determine that it should call this API operation if missing documents must be identified for an insurance claim. The `name`, `description`, and `required` fields tell the agent that it must elicit the unique identifier of the open claim from the customer. The `properties` in the `responses` specify to return the IDs of the open insurance claims. The agent returns this information to the end user or uses some or all of the response as input to subsequent API calls.
- `sendReminders` – Your agent can use the `description` field to determine that it should call this API operation if there is a need to send reminders to the customer. For example, a reminder about pending documents that they have for open claims. The `properties` in the `requestBody` tell the agent that it must find the claim IDs and the pending documents. The `properties` in the `responses` specify to return an ID of the reminder and its status. The agent returns this information to the end user or uses some or all of the response as input to subsequent API calls.

```
{
    "openapi": "3.0.0",
    "info": {
        "title": "Insurance Claims Automation API",
        "version": "1.0.0",
        "description": "APIs for managing insurance claims by pulling a list of open
 claims, identifying outstanding paperwork for each claim, and sending reminders to
 policy holders."
    },
    "paths": {
        "/claims": {
            "get": {
                "summary": "Get a list of all open claims",
```

```
                "description": "Get the list of all open insurance claims. Return all
 the open claimIds.",
                "operationId": "getAllOpenClaims",
                "responses": {
                    "200": {
                        "description": "Gets the list of all open insurance claims for
 policy holders",
                        "content": {
                            "application/json": {
                                "schema": {
                                    "type": "array",
                                    "items": {
                                        "type": "object",
                                        "properties": {
                                            "claimId": {
                                                "type": "string",
                                                "description": "Unique ID of the
 claim."
                                            },
                                            "policyHolderId": {
                                                "type": "string",
                                                "description": "Unique ID of the policy
 holder who has filed the claim."
                                            },
                                            "claimStatus": {
                                                "type": "string",
                                                "description": "The status of the
 claim. Claim can be in Open or Closed state"
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            }
        },
        "/claims/{claimId}/identify-missing-documents": {
            "get": {
                "summary": "Identify missing documents for a specific claim",
                "description": "Get the list of pending documents that need to be
 uploaded by policy holder before the claim can be processed. The API takes in only one
```

```
claim id and returns the list of documents that are pending to be uploaded by policy 
holder for that claim. This API should be called for each claim id",
               "operationId": "identifyMissingDocuments",
               "parameters": [{
                   "name": "claimId",
                   "in": "path",
                   "description": "Unique ID of the open insurance claim",
                   "required": true,
                   "schema": {
                       "type": "string"
                   }
               }],
               "responses": {
                   "200": {
                       "description": "List of documents that are pending to be 
uploaded by policy holder for insurance claim",
                       "content": {
                           "application/json": {
                               "schema": {
                                   "type": "object",
                                   "properties": {
                                       "pendingDocuments": {
                                           "type": "string",
                                           "description": "The list of pending 
documents for the claim."
                                       }
                                   }
                               }
                           }
                       }

                   }
               }
           }
       },
       "/send-reminders": {
           "post": {
               "summary": "API to send reminder to the customer about pending 
documents for open claim",
               "description": "Send reminder to the customer about pending documents 
for open claim. The API takes in only one claim id and its pending documents at a 
time, sends the reminder and returns the tracking details for the reminder. This API 
should be called for each claim id you want to send reminders for.",
               "operationId": "sendReminders",
```

```
                "requestBody": {
                    "required": true,
                    "content": {
                        "application/json": {
                            "schema": {
                                "type": "object",
                                "properties": {
                                    "claimId": {
                                        "type": "string",
                                        "description": "Unique ID of open claims to
 send reminders for."
                                    },
                                    "pendingDocuments": {
                                        "type": "string",
                                        "description": "The list of pending documents
 for the claim."
                                    }
                                },
                                "required": [
                                    "claimId",
                                    "pendingDocuments"
                                ]
                            }
                        }
                    }
                },
                "responses": {
                    "200": {
                        "description": "Reminders sent successfully",
                        "content": {
                            "application/json": {
                                "schema": {
                                    "type": "object",
                                    "properties": {
                                        "sendReminderTrackingId": {
                                            "type": "string",
                                            "description": "Unique Id to track the
 status of the send reminder Call"
                                        },
                                        "sendReminderStatus": {
                                            "type": "string",
                                            "description": "Status of send reminder
 notifications"
                                        }
```

```
                                        }
                                    }
                                }
                            }
                        },
                        "400": {
                            "description": "Bad request. One or more required fields are
 missing or invalid."
                        }
                    }
                }
            }
        }
    }
}
```

For more examples of OpenAPI schemas, see https://github.com/OAI/OpenAPI-Specification/tree/main/examples/v3.0 on the GitHub website.

Define Lambda functions for your agent's action groups in Amazon Bedrock

You must define a Lambda function to program the business logic for an action group. After a Amazon Bedrock agent determines the API operation that it needs to invoke in an action group, it sends information from the API schema alongside relevant metadata as an input event to the Lambda function. To write your function, you must understand the following components of the Lambda function:

- **Input event** – Contains relevant metadata and populated fields from the request body of the API operation that the agent determines must be called.
- **Response** – Contains relevant metadata and populated fields for the response body returned from the API operation.

You write your Lambda function to define how to handle an action group and to customize how you want the API response to be returned. You use the variables from the input event to define your functions and return a response to the agent.

> **Note**
>
> An action group can contain up to 5 API operations, but you can only write one Lambda function. Because the Lambda function can only receive an input event and return a response for one API operation at a time, you should write the function considering the different API operations that may be invoked.

For your agent to use a Lambda function, you must attach a resource-based policy to the function to provide permissions for the agent. For more information, follow the steps at Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function. For more information about resource-based policies in Lambda, see Using resource-based policies for Lambda in the AWS Lambda Developer Guide.

Topics

- Lambda input event from Amazon Bedrock
- Lambda response event to Amazon Bedrock
- Action group Lambda function example

Lambda input event from Amazon Bedrock

When an action group using a Lambda function is invoked, Amazon Bedrock sends a Lambda input event of the following general format. You can define your Lambda function to use any of the input event fields to manipulate the business logic within the function to successfully perform the action. For more information about Lambda functions, see Event-driven invocation in the AWS Lambda Developer Guide.

```
{
    "messageVersion": "1.0",
    "agent": {
        "name": "string",
        "id": "string",
        "alias": "string",
        "version": "string"
    },
    "inputText": "string",
    "sessionId": "string",
    "actionGroup": "string",
```

```
    "apiPath": "string",
    "httpMethod": "string",
    "parameters": [
        {
            "name": "string",
            "type": "string",
            "value": "string"
        },
    ...
    ],
    "requestBody": {
        "content": {
            "<content_type>": {
                "properties": [
                   {
                       "name": "string",
                       "type": "string",
                       "value": "string"
                    },
                            ...
                ]
            }
        }
    },
    "sessionAttributes": {
        "string": "string",
    },
    "promptSessionAttributes": {
        "string": "string"
    }
}
```

The following list describes the input event fields;

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function. Amazon Bedrock only supports version 1.0.
- `agent` – Contains information about the name, ID, alias, and version of the agent that the action group belongs to.
- `inputText` – The user input for the conversation turn.
- `sessionId` – The unique identifier of the agent session.

- `actionGroup` – The name of the action group.
- `apiPath` – The path to the API operation, as defined in the OpenAPI schema.
- `httpMethod` – The method of the API operation, as defined in the OpenAPI schema.
- `parameters` – Contains a list of objects. Each object contains the name, type, and value of a parameter in the API operation, as defined in the OpenAPI schema.
- `requestBody` – Contains the request body and its properties, as defined in the OpenAPI schema.
- `sessionAttributes` – Contains session attributes and their values. These attributes are stored over a session and provide context for the agent.
- `promptSessionAttributes` – Contains prompt session attributes and their values. These attributes are stored over a turn and provide context for the agent.

Lambda response event to Amazon Bedrock

Amazon Bedrock expects a response from your Lambda function that matches the following format. The response consists of parameters returned from the API operation. The agent can use the response from the Lambda function for further orchestration or to help it return a response to the customer.

> **Note**
>
> See Agent quotas for the Lambda payload response quota.

```
{
    "messageVersion": "1.0",
    "response": {
        "actionGroup": "string",
        "apiPath": "string",
        "httpMethod": "string",
        "httpStatusCode": number,
        "responseBody": {
            "<contentType>": {
                "body": "JSON-formatted string"
            }
        }
    },
    "sessionAttributes": {
```

```
        "string": "string",
    },
    "promptSessionAttributes": {
        "string": "string"
    }
}
```

The following list describes the response fields:

- messageVersion – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function. Amazon Bedrock only supports version 1.0.
- response – Contains the following information about the API response.
  - actionGroup – The name of the action group.
  - apiPath – The path to the API operation, as defined in the OpenAPI schema.
  - httpMethod – The method of the API operation, as defined in the OpenAPI schema.
  - responseBody – Contains the response body, as defined in the OpenAPI schema.
- (Optional) sessionAttributes – Contains session attributes and their values.
- (Optional) promptSessionAttributes – Contains prompt attributes and their values.

Action group Lambda function example

The following is an minimal example of how the Lambda function can be defined in Python.

```
def lambda_handler(event, context):

    response_body = {
        'application/json': {
            'body': "sample response"
        }
    }

    action_response = {
        'actionGroup': event['actionGroup'],
        'apiPath': event['apiPath'],
        'httpMethod': event['httpMethod'],
        'httpStatusCode': 200,
        'responseBody': response_body
    }
```

```
    session_attributes = event['sessionAttributes']
    prompt_session_attributes = event['promptSessionAttributes']

    api_response = {
        'messageVersion': '1.0',
        'response': action_response,
        'sessionAttributes': session_attributes,
        'promptSessionAttributes': prompt_session_attributes
    }

    return api_response
```

Define return of control functions for your agent's action groups in Amazon Bedrock

Create an Action Group with either Function or OpenAPI schema and use return of control for API executions.

An `ActionGroup` configured with `RETURN_CONTROL` can be used to return API or function details to be executed by the agent developer.

Topics

Return of control API configuration

For the same Action Group you can now enable Return Control by defining customControl as `RETURN_CONTROL` inside `actionExecutor` for either openAPI or simplified function schema types.

With OpenAPI schema:

```
{
    "actionGroupName": "WeatherAPIs",
    "description": "Actions to get current weather and historical trends for a
 location",
    "actionGroupState": "ENABLED",
    "apiSchema": {
        "s3": {
```

```
            "s3BucketName": "openapi-spec-iad",
            "s3ObjectKey": "get_weather_openapi.yaml"
        }
    },
    "actionGroupExecutor": {
        "customControl": "RETURN_CONTROL"
    }
}
```

With function schema:

```
{
    "actionGroupName": "OrderManagementAction",
    "description": "Action to get the order history, product details, product
 availability and to update the order",
    "actionGroupState": "ENABLED",
    "actionGroupExecutor": {
        "customControl": "RETURN_CONTROL"
    },
    "functionSchema": {
        "functions": [{
            "name": "GetOrderDetails",
            "parameters": {
                "orderId": {
                    "type": "string",
                    "required": true
                }
            }
        }]
    }
}
```

Handling fulfillment of the action

For a simple OpenAPI schema defining a single API

```
openapi: 3.0.0
info:
  title: GetWeather API
  version: 1.0.0
  description: gets weather
paths:
  /getWeather/{location}/:
```

```
    get:
      summary: gets weather in celcius
      description: gets weather in celcius
      operationId: getWeather
      parameters:
        - name: location
          in: path
          description: location name
          required: true
          schema:
            type: string
      responses:
        "200":
          description: weather in celcius
          content:
            application/json:
              schema:
                type: string
```

You can send an invokeAgent request to an agent configured to use return_control.

InvokeAgent Request Payload

```
POST https://bedrock-agent-runtime.us-east-1.amazonaws.com/agents/ABCDEFGHIJ/
agentAliases/TSTALIASID/sessions/abb/text
{
    "inputText": "give me weather for seattle",
    "enableTrace": true
}
```

InvokeAgent Response Stream

The streaming response will include trace events along a new event type `returnControl`.

```
event - type returnControl: content - type application / json: message - type event {
    "invocationInputs": [{
        "apiInvocationInput": {
            "actionGroup": "WeatherAPIs",
            "apiPath": "getWeather",
            "httpMethod": "get",
            "parameters": [{
                "name": "location",
                "type": "string",
```

```
                "value": "seattle"
            }]
        }
    }],
    "invocationId": "337cb2f6-ec74-4b49-8141-00b8091498ad"
}
```

InvokeAgent Request after developer side execution

Developer can provide API or Function response from the execution inside SessionState of invokeAgent request.

The same invocationId must be used to provide returnControl results that was returned by a previous response.

```
POST https://bedrock-agent-runtime.us-east-1.amazonaws.com/agents/ABCDEFGHIJ/
agentAliases/TSTALIASID/sessions/abb/text
{
    "enableTrace": true,
    "sessionState": {
        "invocationId": "337cb2f6-ec74-4b49-8141-00b8091498ad",
        "returnControlInvocationResults": [{
            "apiResult": {
                "actionGroup": "WeatherAPIs",
                "httpStatusCode": 200,
                "httpMethod": "get",
                "apiPath": "getWeather",
                "responseBody": {
                    "text": {
                        "body": "30 celcius"
                    }
                }
            },
            "functionResult": { # sample function example
                "actionGroup": "LocationAPIs",
                "function": "getLocation",
                "responseBody": {
                    "TEXT": {
                        "body": "Seattle"
                    }
                }
            }
        }]
```

```
    }
}
```

Add an action group to your agent in Amazon Bedrock

After setting up the OpenAPI schema and Lambda function for your action group, you can create the action group. Select the tab corresponding to your method of choice and follow the steps.

Console

When you create an agent, you can add action groups to the working draft.

After an agent is created, you can add action groups to it by doing the following steps:

To add an action group to an already created agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent from the **Agents** section and then choose the **Working draft** in the **Working Draft** section.
4. Select **Add** in the **Action groups** section.
5. Fill out the action group details
6. To define the schema for the action group with the in-line OpenAPI schema editor, carry out the following actions. For more information about API schemas for your action group, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock.
   a. Choose **Define with in-line OpenAPI schema editor** under **Select API schema**. A sample schema appears that you can edit.
   b. Select the format for the schema by using the dropdown menu next to **Format**.
   c. To import an existing schema from S3 to edit, select **Import schema**, provide the S3 URI, and select **Import**.
   d. To restore the schema to the original sample schema, select **Reset** and then confirm the message that appears by selecting **Reset** again.
7. Select **Add**. A green success banner appears if there are no issues. If there are issues validating the schema, a red banner appears. The following issues are identified by the validation process.

- Scroll through the schema to see the lines where an error or warning about formatting exists. An X indicates a formatting error, while an exclamation mark indicates a warning about formatting.
- Select **View details** in the red banner to see a list of errors about the content of the API schema.

8. Select **Prepare** to apply the changes that you have made to the agent before testing it.

API

To create an action group, send a CreateAgentActionGroup request with a Agents for Amazon Bedrock build-time endpoint.

See code examples

The following list describes the fields in the request:

- The following fields are required:

Field	Short description
agentId	The ID of the agent that the action group belongs to.
agentVersion	The version of the agent that the action group belongs to.
actionGroupName	The name of the action group.

- The following fields are optional:

Field	Short description
parentActionGroupSignature	Specify `AMAZON.UserInput` to allow the agent to reprompt the user for more information if it doesn't have enough information to complete another action group. You must leave the `description` ,

Field	Short description
	`apiSchema` , and `actionGroupExecutor` fields blank if you specify this field.
description	A description of the action group.
apiSchema	Specifies the OpenAPI schema for the action group or links to an S3 object containing it.
actionGroupExecutor	Specifies the Amazon Resource Name (ARN) of the Lambda function that performs the action group.
actionGroupState	Whether to allow the agent to invoke the action group or not.
clientToken	An identifier to prevent requests from being duplicated.

Associate a knowledge base with an Amazon Bedrock agent

If you haven't yet created a knowledge base, see Knowledge bases for Amazon Bedrock to learn about knowledge bases and create one. You can associate a knowledge base during agent creation or after an agent has been created. To associate a knowledge base to an existing agent. select the tab corresponding to your method of choice and follow the steps..

Console

To add a knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. On the agent details page, choose the working draft from the **Working draft** section.
4. For the **Knowledge bases** section, choose **Add**.

5. Choose a knowledge base that you have created and provide instructions for how the agent should interact with it.
6. Choose **Add**. A success banner appears at the top.
7. To apply the changes that you made to the agent before testing it, choose **Prepare** before testing it.

API

To associate a knowledge base with an agent, send an [AssociateAgentKnowledgeBase] request with a [Agents for Amazon Bedrock build-time endpoint].

The following list describes the fields in the request:

- The following fields are required:

Field	Short description
agentId	ID of the agent
agentVersion	Version of the agent
knowledgeBaseId	ID of the knowledge base

- The following fields are optional:

Field	Short description
description	Description of how the agent can use the knowledge base
knowledgeBaseState	To prevent the agent from querying the knowledge base, specify `DISABLED`

Test an Amazon Bedrock agent

After you create an agent, you will have a *working draft*. The working draft is a version of the agent that you can use to iteratively build the agent. Each time you make changes to your agent, the working draft is updated. When you're satisfied with your agent's configurations, you can create a *version*, which is a snapshot of your agent, and an *alias*, which points to the version. You can then deploy your agent to your applications by calling the alias. For more information, see Deploy an Amazon Bedrock agent.

The following list describes how you test your agent:

- In the Amazon Bedrock console, you open up the test window on the side and send input for your agent to respond to. You can select the working draft or a version that you've created.
- In the API, the working draft is the `DRAFT` version. You send input to your agent by using InvokeAgent with the test alias, `TSTALIASID`, or a different alias pointing to a static version.

To help troubleshoot your agent's behavior, Agents for Amazon Bedrock provides the ability to view the *trace* during a session with your agent. The trace shows the agent's step-by-step reasoning process. For more information about the trace, see Trace events in Amazon Bedrock.

Following are steps for testing your agent. Select the tab corresponding to your method of choice and follow the steps.

Console

To test an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agents** section, select the link for the agent that you want to test from the list of agents.
4. The **Test** window appears in a pane on the right.

> **Note**
>
> If the **Test window** is closed, you can reopen it by selecting **Test** at the top of the agent details page or any page within it.

5. After you create an agent, you must package it with the working draft changes by preparing it in one of the following ways:
    - In the **Test** window, select **Prepare**.

- In the **Working draft** page, select **Prepare** at the top of the page.

> **Note**
>
> Every time you update the working draft, you must prepare the agent to package the agent with your latest changes. As a best practice, we recommend that you always check your agent's **Last prepared** time in the **Agent overview** section of the **Working draft** page to verify that you're testing your agent with the latest configurations.

6. To choose an alias and associated version to test, use the dropdown menu at the top of the **Test window**. By default, the **TestAlias: Working draft** combination is selected.
7. To test the agent, enter a message and choose **Run**. While you wait for the response to generate or after it is generated, you have the following options:

    - To view details for each step of the agent's orchestration process, including the prompt, inference configurations, and agent's reasoning process for each step and usage of its action groups and knowledge bases, select **Show trace**. The trace is updated in real-time so you can view it before the response is returned. To expand or collapse the trace for a step, select an arrow next to a step. For more information about the **Trace** window and details that appear, see [Trace events in Amazon Bedrock](#).
    - If the agent invokes a knowledge base, the response contains footnotes. To view the link to the S3 object containing the cited information for a specific part of the response, select the relevant footnote.

    You can perform the following actions in the **Test** window:

    - To start a new conversation with the agent, select the refresh icon.
    - To view the **Trace** window, select the expand icon. To close the **Trace** window, select the shrink icon.
    - To close the **Test** window, select the right arrow icon.

You can enable or disable action groups and knowledge bases. Use this feature to troubleshoot your agent by isolating which action groups or knowledge bases need to be updated by assessing its behavior with different settings.

To enable an action group or knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agents** section. select the link for the agent that you want to test from the list of agents.
4. On the agent's details page, in the **Working draft** section, select the link for the **Working draft**.
5. In the **Action groups** or **Knowledge bases** section, hover over the **State** of the action group or knowledge base whose state you want to change.
6. An edit button appears. Select the edit icon and then choose from the dropdown menu whether the action group or knowledge base is **Enabled** or **Disabled**.
7. If an action group is **Disabled**, the agent doesn't use the action group. If a knowledge base is **Disabled**, the agent doesn't use the knowledge base. Enable or disable action groups or knowledge bases and then use the **Test** window to troubleshoot your agent.
8. Choose **Prepare** to apply the changes that you have made to the agent before testing it.

API

Before you test your agent for the first time, you must package it with the working draft changes by sending a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

See code examples

> **Note**
>
> Every time you update the working draft, you must prepare the agent to package the agent with your latest changes. As a best practice, we recommend that you send a GetAgent request (see link for request and response formats and field details) with a Agents for Amazon Bedrock build-time endpoint and check the `preparedAt` time for your agent to verify that you're testing your agent with the latest configurations.

To test your agent, send an InvokeAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint.

See code examples

> **Note**
>
> The AWS CLI doesn't support InvokeAgent.

See code examples

The following fields exist in the request:

- Minimally, provide the following required fields:

Field	Short description
agentId	ID of the agent
agentAliasId	ID of the alias. Use `TSTALIASID` to invoke the `DRAFT` version
sessionId	Alphanumeric ID for the session (2–100 characters)
inputText	The user prompt to send to the agent

- The following fields are optional:

Field	Short description
enableTrace	Specify `TRUE` to view the trace.
endSession	Specify `TRUE` to end the session with the agent after this request.
sessionState	Includes `sessionAttributes` that remain throughout the session and/or `promptSessionAttributes` that

Field	Short description
	remain throughout the InvokeAgent turn. These attributes give the agent context.

The response is returned in bytes in the `chunk` object. If the agent queried a knowledge base, the `chunk` includes `citations`. If you enabled a trace, a `trace` object is also returned. If an error occurs, a field is returned with the error message. For more information about how to read the trace, see Trace events in Amazon Bedrock.

Trace events in Amazon Bedrock

Each response from an Amazon Bedrock agent is accompanied by a *trace* that details the steps being orchestrated by the agent. The trace helps you follow the agent's reasoning process that leads it to the response it gives at that point in the conversation.

Use the trace to track the agent's path from the user input to the response it returns. The trace provides information about the inputs to the action groups that the agent invokes and the knowledge bases that it queries to respond to the user. In addition, the trace provides information about the outputs that the action groups and knowledge bases return. You can view the reasoning that the agent uses to determine the action that it takes or the query that it makes to a knowledge base. If a step in the trace fails, the trace returns a reason for the failure. Use the detailed information in the trace to troubleshoot your agent. You can identify steps at which the agent has trouble or at which it yields unexpected behavior. Then, you can use this information to consider ways in which you can improve the agent's behavior.

View the trace

The following describes how to view the trace. Select the tab corresponding to your method of choice and follow the steps.

Console

To view the trace during a conversation with an agent

Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

1. In the **Agents** section, select the link for the agent that you want to test from the list of agents.
2. The **Test** window appears in a pane on the right.
3. Enter a message and choose **Run**. While the response is generating or after it finishes generating, select **Show trace**.
4. You can view the trace for each **Step** in real-time as your agent performs orchestration.

API

To view the trace, send an InvokeAgent request with a Agents for Amazon Bedrock runtime endpoint and set the `enableTrace` field to `TRUE`. By default, the trace is disabled.

If you enable the trace, in the InvokeAgent response, each `chunk` in the stream is accompanied by a `trace` field that maps to a TracePart object. Within the TracePart is a `trace` field that maps to a Trace object.

Structure of the trace

The trace is shown as a JSON object in both the console and the API. Each **Step** in the console or Trace in the API can be one of the following traces:

- PreProcessingTrace – Traces the input and output of the pre-processing step, in which the agent contextualizes and categorizes user input and determines if it is valid.
- Orchestration – Traces the input and output of the orchestration step, in which the agent interprets the input and invokes API operations and queries knowledge bases. Then the agent returns output to either continue orchestration or to respond to the user.
- PostProcessingTrace – Traces the input and output of the post-processing step, in which the agent handles the final output of the orchestration and determines how to return the response to the user.
- FailureTrace – Traces the reason that a step failed.

Each of the traces (except `FailureTrace`) contains a ModelInvocationInput object. The ModelInvocationInput object contains configurations set in the prompt template for the step, alongside the prompt provided to the agent at this step. For more information about how to modify prompt templates, see Advanced prompts in Amazon Bedrock. The structure of the `ModelInvocationInput` object is as follows:

```
{
    "traceId": "string",
    "text": "string",
    "type": "PRE_PROCESSING | ORCHESTRATION | KNOWLEDGE_BASE_RESPONSE_GENERATION |
 POST_PROCESSING",
    "inferenceConfiguration": {
        "maximumLength": number,
        "stopSequences": ["string"],
        "temperature": float,
        "topK": float,
        "topP": float
    },
    "promptCreationMode": "DEFAULT | OVERRIDDEN",
    "parserMode": "DEFAULT | OVERRIDDEN",
    "overrideLambda": "string"
}
```

The following list describes the fields of the ModelInvocationInput object:

- `traceId` – The unique identifier of the trace.
- `text` – The text from the prompt provided to the agent at this step.
- `type` – The current step in the agent's process.
- `inferenceConfiguration` – Inference parameters that influence response generation. For more information, see Inference parameters.
- `promptCreationMode` – Whether the agent's default base prompt template was overridden for this step. For more information, see Advanced prompts in Amazon Bedrock.
- `parserMode` – Whether the agent's default response parser was overridden for this step. For more information, see Advanced prompts in Amazon Bedrock.
- `overrideLambda` – The Amazon Resource Name (ARN) of the parser Lambda function used to parse the response, if the default parser was overridden. For more information, see Advanced prompts in Amazon Bedrock.

For more information about each trace type, see the following sections:

PreProcessingTrace

```
{
    "modelInvocationInput": { // see above for details }
```

```
    "modelInvocationOutput": {
        "parsedResponse": {
            "isValid": boolean,
            "rationale": "string"
        },
        "traceId": "string"
    }
}
```

The PreProcessingTrace consists of a ModelInvocationInput object and a PreProcessingModelInvocationOutput object. The PreProcessingModelInvocationOutput contains the following fields.

- `parsedResponse` – Contains the following details about the parsed user prompt.
  - `isValid` – Specifies whether the user prompt is valid.
  - `rationale` – Specifies the agent's reasoning for the next steps to take.
- `traceId` – The unique identifier of the trace.

OrchestrationTrace

The Orchestration consists of the ModelInvocationInput object and any combination of the Rationale, InvocationInput, and Observation objects. For more information about each object, select from the following tabs:

```
{
    "modelInvocationInput": { // see above for details },
    "rationale": { ... },
    "invocationInput": { ... },
    "observation": { ... }
}
```

Rationale

The Rationale object contains the reasoning of the agent given the user input. Following is the structure:

```
{
       "traceId": "string",
       "text": "string"
```

```
    }
```

The following list describes the fields of the Rationale object:

- `traceId` – The unique identifier of the trace step.
- `text` – The reasoning process of the agent, based on the input prompt.

InvocationInput

The InvocationInput object contains information that will be input to the action group or knowledge base that is to be invoked or queried. Following is the structure:

```
{
    "traceId": "string",
    "invocationType": "ACTION_GROUP | KNOWLEDGE_BASE | FINISH",
    "actionGroupInvocationInput": {
        "actionGroupName": "string",
        "apiPath": "string",
        "verb": "string",
        "parameters": [
            {
                "name": "string",
                "type": "string"
                "value": "string"
            },
            ...
        ],
        "request": {
            "content": {
                "content-type": [
                    {
                        "name": "string",
                        "type": "string",
                        "value": "string"
                    }
                ]
            }
        }
    },
    "knowledgeBaseLookupInput": {
        "knowledgeBaseId": "string",
        "text": "string"
```

```
    }
}
```

The following list describes the fields of the InvocationInput object:

- `traceId` – The unique identifier of the trace.
- `invocationType` – Specifies whether the agent is invoking an action group or a knowledge base, or ending the session.
- `actionGroupInvocationInput` – Appears if the `type` is `ACTION_GROUP`. For more information, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock. Contains the following input for the action group that is invoked:
  - `actionGroupName` – The name of the action group that the agent will invoke.
  - `apiPath` – The path to the API operation to call, according to the API schema.
  - `verb` – The API method being used, according to the API schema.
  - `parameters` – Contains a list of objects. Each object contains the name, type, and value of a parameter in the API operation, as defined in the API schema.
  - `requestBody` – Contains the request body and its properties, as defined in the API schema.
- `knowledgeBaseLookupInput` – Appears if the `type` is `KNOWLEDGE_BASE`. For more information, see Knowledge bases for Amazon Bedrock. Contains the following information about the knowledge base and the search query for the knowledge base:
  - `knowledgeBaseId` – The unique identifier of the knowledge base that the agent will look up.
  - `text` – The query to be made to the knowledge base.

Observation

The Observation object contains the result or output of an action group or knowledge base, or the response to the user. Following is the structure:

```
{
    "traceId": "string",
    "type": "ACTION_GROUP | KNOWLEDGE_BASE | REPROMPT | ASK_USER | FINISH"
    "actionGroupInvocation": {
        "text": "JSON-formatted string"
    },
```

```
    "knowledgeBaseLookupOutput": {
        "retrievedReferences": [
            {
                "content": {
                    "text": "string"
                },
                "location": {
                    "type": "S3",
                    "s3Location": {
                        "uri": "string"
                    }
                }
            },
            ...
        ]
    },
    "repromptResponse": {
        "source": "ACTION_GROUP | KNOWLEDGE_BASE | PARSER",
        "text": "string"
    },
    "finalResponse": {
        "text"
    }
}
```

The following list describes the fields of the Observation object:

- `traceId` – The unique identifier of the trace.
- `type` – Specifies whether the agent's observation is returned from the result of an action group or a knowledge base, if the agent is reprompting the user, requesting more information, or ending the conversation.
- `actionGroupInvocationOutput` – Contains the JSON-formatted string returned by the API operation that was invoked by the action group. Appears if the `type` is `ACTION_GROUP`. For more information, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock.
- `knowledgeBaseLookupOutput` – Contains text retrieved from the knowledge base that is relevant to responding to the prompt, alongside the Amazon S3 location of the data source. Appears if the `type` is `KNOWLEDGE_BASE`. For more information, see Knowledge bases for Amazon Bedrock. Each object in the list of `retrievedReferences` contains the following fields:

  - `content` – Contains `text` from the knowledge base that is returned from the knowledge base query.
  - `location` – Contains the Amazon S3 URI of the data source from which the returned text was found.
- `repromptResponse` – Appears if the `type` is `REPROMPT`. Contains the `text` that asks for a prompt again alongside the `source` of why the agent needs to reprompt.
- `finalResponse` – Appears if the `type` is `ASK_USER` or `FINISH`. Contains the `text` that asks the user for more information or is a response to the user.

PostProcessingTrace

```
{
    "modelInvocationInput": { // see above for details }
    "modelInvocationOutput": {
        "parsedResponse": {
            "text": "string"
        },
        "traceId": "string"
    }
}
```

The PostProcessingTrace consists of a ModelInvocationInput object and a PostProcessingModelInvocationOutput object. The PostProcessingModelInvocationOutput contains the following fields:

- `parsedResponse` – Contains the `text` to return to the user after the text is processed by the parser function.
- `traceId` – The unique identifier of the trace.

FailureTrace

```
{
    "failureReason": "string",
    "traceId": "string"
}
```

The following list describes the fields of the FailureTrace object:

- `failureReason` – The reason that the step failed.
- `traceId` – The unique identifier of the trace.

Manage an Amazon Bedrock agent

After you create an agent, you can view or update its configuration as required. The configuration applies to the working draft. If you no longer need an agent, you can delete it.

Topics

View information about an agent

To learn how to view information about an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. On the agent details, you can view the following information:
   - The **Agent overview** section contains the agent configuration.
   - The **Tags** section contains tags that are associated with the agent. For more information, see Tag resources.
   - The **Working draft** section contains the working draft. If you select the working draft, you can view the following information:

- The **Model details** section contains model and instructions used by the agent's working draft.
- The **Action groups** section contains the action groups that the agent uses. For more information, see Create an action group for an Amazon Bedrock agent and Manage the action groups of an agent.
- The **Knowledge bases** section contains the knowledge bases associated with the agent. For more information, see Associate a knowledge base with an Amazon Bedrock agent and Manage agent-knowledge bases associations.
- The **Advanced prompts** section contains the prompt templates for each step of the agent's orchestration. For more information, see Advanced prompts in Amazon Bedrock.

- The **Versions** and **Aliases** sections contain versions and aliases of the agent that you can use for deployment to your applications. For more information, see Deploy an Amazon Bedrock agent.

API

To get information about an agent, send a GetAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`. See code examples.

To list information about your agents, send a ListAgents request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. See code examples. You can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

To list all the tags for an agent, send a ListTagsForResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the agent.

Edit an agent

To learn how to edit an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To edit the agent configuration

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agent overview** section, choose **Edit**.
4. Edit the existing information in the fields as necessary.
5. When you're done editing the information, choose **Save** to remain in the same window or **Save and exit** to return to the agent details page. A success banner appears at the top. To apply the new configurations to your agent, select **Prepare** in the banner.

You might want to try different foundation models for your agent or change the instructions for the agent. These changes apply only to the working draft.

To change the foundation model that your agent uses or the instructions to the agent.

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Model details** section, choose **Edit**
6. Select a different model or edit the instructions to the agent as necessary.

> **Note**
>
> If you change the foundation model, any prompt templates that you modified will be set to default for that model.

7. When you're done editing the information, choose **Save** to remain in the same window or **Save and exit** to return to the agent details page. A success banner appears at the top.
8. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

To edit the tags associated with an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. In the **Tags** section, choose **Manage tags**.
5. To add a tag, choose **Add new tag**. Then enter a **Key** and optionally enter a **Value**. To remove a tag, choose **Remove**. For more information, see Tag resources.
6. When you're done editing tags, choose **Submit**.

API

To edit an agent, send an UpdateAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same. For more information about required and optional fields, see Create an agent in Amazon Bedrock.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

To add tags to an agent, send a TagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the

Amazon Resource Name (ARN) of the agent. The request body contains a `tags` field, which is an object containing a key-value pair that you specify for each tag.

To remove tags from an agent, send an [UntagResource](#) request (see link for request and response formats and field details) with an [Agents for Amazon Bedrock build-time endpoint](#) and include the Amazon Resource Name (ARN) of the agent. The `tagKeys` request parameter is a list containing the keys for the tags that you want to remove.

Delete an agent

To learn how to delete an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To delete an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane.
3. To delete an agent, choose the option button that's next to the agent you want to delete.
4. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the agent, enter **`delete`** in the input field and then select **Delete**.
5. When deletion is complete, a success banner appears.

API

To delete an agent, send a [DeleteAgent](#) request (see link for request and response formats and field details) with an [Agents for Amazon Bedrock build-time endpoint](#) and specify the `agentId`.

By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

[See code examples](#)

Select a topic to learn about how to manage the action groups or knowledge bases for an agent.

Topics

Manage the action groups of an agent

After creating an action group, you can view, edit, or delete it. The changes apply to the working draft version of the agent.

Topics

View information about an action group

To learn how to view information about an action group, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about an action group

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, choose an action group for which to view information.

API

To get information about an action group, send a GetAgentActionGroup request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `actionGroupId`, `agentId`, and `agentVersion`.

To list information about an agent's action groups, send a ListAgentActionGroups request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentVersion` for which you want to see action groups. You can include the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

See code examples

Edit an action group

To learn how to edit an action group, select the tab corresponding to your method of choice and follow the steps.

Console

To edit an action group

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, select an action group to edit. Then choose **Edit**.
6. Edit the existing fields as necessary. For more information, see Create an action group for an Amazon Bedrock agent.

7. To define the schema for the action group with the in-line OpenAPI schema editor, for **Select API schema**, choose **Define with in-line OpenAPI schema editor**. A sample schema appears that you can edit. You can configure the following options:
   - To import an existing schema from Amazon S3 to edit, choose **Import schema**, provide the Amazon S3 URI, and select **Import**.
   - To restore the schema to the original sample schema, choose **Reset** and then confirm the message that appears by choosing **Confirm**.
   - To select a different format for the schema, use the dropdown menu labeled **JSON**.
   - To change the visual appearance of the schema, choose the gear icon below the schema.
8. To control whether the agent can use the action group, select **Enable** or **Disable**. Use this function to help troubleshoot your agent's behavior.
9. To remain in the same window so that you can test your change, choose **Save**. To return to the action group details page, choose **Save and exit**.
10. A success banner appears if there are no issues. If there are issues validating the schema, an error banner appears. To see a list of errors, choose **Show details** in the banner.
11. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To edit an action group, send an [UpdateAgentActionGroup](#) request (see link for request and response formats and field details) with an [Agents for Amazon Bedrock build-time endpoint](#). Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same. You must specify the `agentVersion` as `DRAFT`. For more information about required and optional fields, see [Create an action group for an Amazon Bedrock agent](#).

To apply the changes to the working draft, send a [PrepareAgent](#) request (see link for request and response formats and field details) with an [Agents for Amazon Bedrock build-time endpoint](#). Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Delete an action group

To learn how to delete an action group, select the tab corresponding to your method of choice and follow the steps.

Console

To delete an action group

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, choose the option button that's next to the action group you want to delete.
6. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the action group, enter **`delete`** in the input field and then select **Delete**.
7. When deletion is complete, a success banner appears.
8. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To delete an action group, send a DeleteAgentActionGroup request. Specify the `actionGroupId` and the `agentId` and `agentVersion` from which to delete it. By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Manage agent-knowledge bases associations

After creating an agent, you can add more knowledge bases or edit them. Adding and editing take place within the working draft. To carry out these operations, choose an agent from the **Agents** section and then choose the **Working draft** in the **Working Draft** section.

Topics

- View information about an agent-knowledge base association
- Edit an agent-knowledge base association
- Disassociate a knowledge base from an agent

View information about an agent-knowledge base association

To learn how to view information about a knowledge base, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a knowledge base that's associated with an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Knowledge bases** section, select the knowledge base for which you want to view information.

API

To get information about a knowledge base associated with an agent, send a GetAgentKnowledgeBase request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the following fields:

To list information about the knowledge bases associated with an agent, send a ListAgentKnowledgeBases request (see link for request and response formats and field details) with an [Agents for Amazon Bedrock build-time endpoint](). Specify the `agentId` and `agentVersion` for which you want to see associated knowledge bases.

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

See code examples

Edit an agent-knowledge base association

To learn how to edit an agent-knowledge base association, select the tab corresponding to your method of choice and follow the steps.

Console

To edit an agent-knowledge base association

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, select an action group to edit. Then choose **Edit**.
6. Edit the existing fields as necessary. For more information, see Associate a knowledge base with an Amazon Bedrock agent.
7. To control whether the agent can use the knowledge base, select **Enabled** or **Disabled**. Use this function to help troubleshoot your agent's behavior.

8. To remain in the same window so that you can test your change, choose **Save**. To return to the **Working draft** page, choose **Save and exit**.
9. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To edit the configuration of a knowledge base associated with an agent, send an UpdateAgentKnowledgeBase request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same. You must specify the `agentVersion` as `DRAFT`. For more information about required and optional fields, see Associate a knowledge base with an Amazon Bedrock agent.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Disassociate a knowledge base from an agent

To learn how to disassociate a knowledge base from an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To dissociate a knowledge base from an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Knowledge bases** section, choose the option button that's next to the knowledge base that you want to delete. Then choose **Delete**.
6. Confirm the message that appears and then choose **Delete**.

7. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To disassociate a knowledge base from an agent, send a DisassociateAgentKnowledgeBase request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `knowledgeBaseId` and the `agentId` and `agentVersion` of the agent from which to disassociate it.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Customize an Amazon Bedrock agent

After you have set up your agent, you can further customize its behavior with the following features:

- **Advanced prompts** let you modify prompt templates to determine the prompt that is sent to the agent at each step of runtime.
- **Session state** is a field that contains attributes that you can define during build-time when sending a CreateAgent request or that you can send at runtime with an InvokeAgent request. You can use these attributes to provide and manage context in a conversation between users and the agent.
- Agents for Amazon Bedrock offers options to choose different flows that can optimize on latency for simpler use cases in which agents have a single knowledge base. To learn more, refer to the performance optimization topic.

Select a topic to learn more about that feature.

Topics

Advanced prompts in Amazon Bedrock

After creation, an agent is configured with the following four default **base prompt templates**, which outline how the agent constructs prompts to send to the foundation model at each step of the agent sequence. For details about what each step encompasses, see Runtime process.

- Pre-processing
- Orchestration
- Knowledge base response generation
- Post-processing (disabled by default)

Prompt templates define how the agent does the following:

- Processes user input text and output prompts from foundation models (FMs)
- Orchestrates between the FM, action groups, and knowledge bases
- Formats and returns responses to the user

By using advanced prompts, you can enhance your agent's accuracy through modifying these prompt templates to provide detailed configurations. You can also provide hand-curated examples for *few-shot prompting*, in which you improve model performance by providing labeled examples for a specific task.

Select a topic to learn more about advanced prompts.

Topics

Advanced prompts terminology

The following terminology is helpful in understanding how advanced prompts work.

- **Session** – A group of InvokeAgent requests made to the same agent with the same session ID. When you make an `InvokeAgent` request, you can reuse a `sessionId` that was returned from the response of a previous call in order to continue the same session with an agent. As long as the `idleSessionTTLInSeconds` time in the [Agent](#) configuration hasn't expired, you maintain the same session with the agent.

- **Turn –** A single `InvokeAgent` call. A session consists of one or more turns.
- **Iteration –** A sequence of the following actions:
  1. (Required) A call to the foundation model
  2. (Optional) An action group invocation
  3. (Optional) A knowledge base invocation
  4. (Optional) A response to the user asking for more information

  An action might be skipped, depending on the configuration of the agent or the agent's requirement at that moment. A turn consists of one or more iterations.
- **Prompt –** A prompt consists of the instructions to the agent, context, and text input. The text input can come from a user or from the output of another step in the agent sequence. The prompt is provided to the foundation model to determine the next step that the agent takes in responding to user input
- **Base prompt template –** The structural elements that make up a prompt. The template consists of placeholders that are filled in with user input, the agent configuration, and context at runtime to create a prompt for the foundation model to process when the agent reaches that step. For more information about these placeholders, see [Placeholder variables in Amazon Bedrock agent prompt templates](#)). With advanced prompts, you can edit these templates.

Configure the prompt templates

With advanced prompts, you can do the following:

- Turn on or turn off invocation for different steps in the agent sequence.
- Configure their inference parameters.
- Edit the default base prompt templates that the agent uses. By overriding the logic with your own configurations, you can customize your agent's behavior.

For each step of the agent sequence, you can edit the following parts:

- **Prompt template** – Describes how the agent should evaluate and use the prompt that it receives at the step for which you're editing the template. When editing a template, you can engineer the prompt with the following tools:
  - **Prompt template placeholders** – Pre-defined variables in Agents for Amazon Bedrock that are dynamically filled in at runtime during agent invocation. In the prompt templates, you'll see these placeholders surrounded by `$` (for example, `$instructions$`). For information about the placeholder variables that you can use in a template, see Placeholder variables in Amazon Bedrock agent prompt templates.
  - **XML tags** – Anthropic models support the use of XML tags to structure and delineate your prompts. Use descriptive tag names for optimal results. For example, in the default orchestration prompt template, you'll see the `<examples>` tag used to delineate few-shot examples). For more information, see Use XML tags in the Anthropic user guide.

  You can enable or disable any step in the agent sequence. The following table shows the default states for each step.

Prompt template	Default setting
Pre-processing	Enabled
Orchestration	Enabled
Knowledge base response generation	Enabled
Post-processing	Disabled

  Note

  If you disable the orchestration step, the agent sends the raw user input to the foundation model and doesn't use the base prompt template for orchestration. If you disable any of the other steps, the agent skips that step entirely.

- **Inference configurations** – Influences the response generated by the model that you use. For definitions of the inference parameters and more details about the parameters that different models support, see Inference parameters for foundation models.

- **(Optional) Parser Lambda function** – Defines how to parse the raw foundation model output and how to use it in the runtime flow. This function acts on the output from the steps in which you enable it and returns the parsed response as you define it in the function.

  Depending on how you customized the base prompt template, the raw foundation model output might be specific to the template. As a result, the agent's default parser might have difficulty parsing the output correctly. By witing a custom parser Lambda function, you can help the agent parse the raw foundation model output based on your use-case. For more information about the parser Lambda function and how to write it, see Parser Lambda function in Agents for Amazon Bedrock.

> **Note**
>
> You can define one parser Lambda function for all of the base templates, but you can configure whether to invoke the function in each step. Be sure to configure a resource-based policy for your Lambda function so that your agent can invoke it. For more information, see Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function.

After you edit the prompt templates, you can test your agent. To analyze the step-by-step process of the agent and determine if it is working as you intend, turn on the trace and examine it. For more information, see Trace events in Amazon Bedrock.

You can configure advanced prompts in either the AWS Management Console or through the API.

Console

In the console, you can configure advanced prompts after you have created the agent. You configure them while editing the agent.

To view or edit advanced prompts for your agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. In the left navigation pane, choose **Agents**. Then choose an agent in the **Agents** section.
3. On the agent details page, in the **Working draft** section, select **Working draft**.
4. On the **Working draft** page, in the **Advanced prompts** section, choose **Edit**.

5. On the **Edit advanced prompts** page, choose the tab corresponding to the step of the agent sequence that you want to edit.
6. To enable editing of the template, turn on **Override template defaults**. In the **Override template defaults** dialog box, choose **Confirm**.

> **Warning**
>
> If you turn off **Override template defaults** or change the model, the default Amazon Bedrock template is used and your template will be immediately deleted. To confirm, enter `confirm` in the text box to confirm the message that appears.

7. To allow the agent to use the template when generating responses, turn on **Activate template**. If this configuration is turned off, the agent doesn't use the template.
8. To modify the example prompt template, use the **Prompt template editor**.
9. In **Configurations**, you can modify inference parameters for the prompt. For definitions of parameters and more information about parameters for different models, see Inference parameters for foundation models.
10. (Optional) To use a Lambda function that you have defined to parse the raw foundation model output, perform the following actions:

> **Note**
>
> One Lambda function is used for all the prompt templates.

    a. In the **Configurations** section, select **Use Lambda function for parsing**. If you clear this setting, your agent will use the default parser for the prompt.
    b. For the **Parser Lambda function**, select a Lambda function from the dropdown menu.

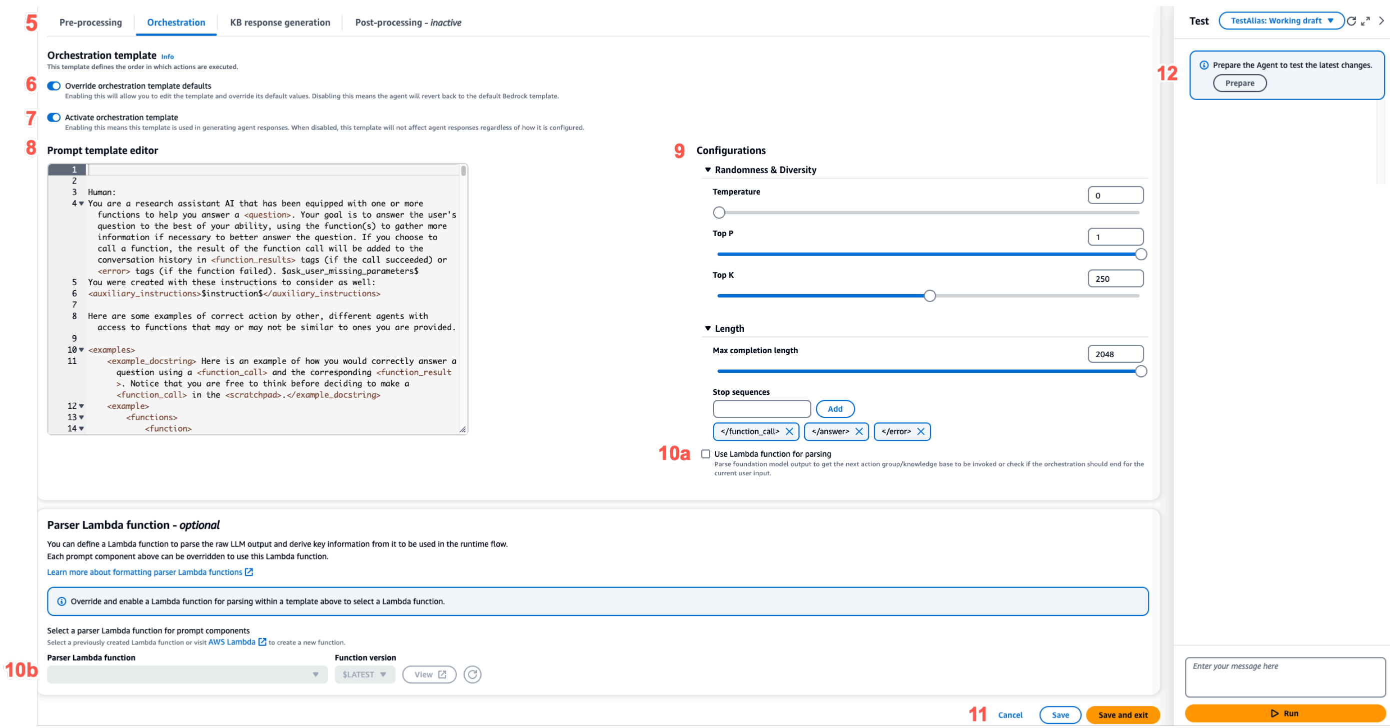
> **Note**
>
> You must attach permissions for your agent so that it can access the Lambda function. For more information, see Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function.

11. To save your settings, choose one of the following options:

a. To remain in the same window so that you can dynamically update the prompt settings while testing your updated agent, choose **Save**.

b. To save your settings and return to the **Working draft** page, choose **Save and exit**.

12. To test the updated settings, choose **Prepare** in the **Test** window.



API

To configure advanced prompts by using the API operations, you send an [UpdateAgent](#) call and modify the following `promptOverrideConfiguration` object.

```
"promptOverrideConfiguration": {
    "overrideLambda": "string",
    "promptConfigurations": [
        {
            "basePromptTemplate": "string",
            "inferenceConfiguration": {
                "maximumLength": int,
                "stopSequences": [ "string" ],
                "temperature": float,
                "topK": float,
                "topP": float
            },
```

```
            "parserMode": "DEFAULT | OVERRIDDEN",
            "promptCreationMode": "DEFAULT | OVERRIDDEN",
            "promptState": "ENABLED | DISABLED",
            "promptType": "PRE_PROCESSING | ORCHESTRATION |
 KNOWLEDGE_BASE_RESPONSE_GENERATION | POST_PROCESSING"
        }
    ]
}
```

1. In the `promptConfigurations` list, include a `promptConfiguration` object for each prompt template that you want to edit.
2. Specify the prompt to modify in the `promptType` field.
3. Modify the prompt template through the following steps:

   a. Specify the `basePromptTemplate` fields with your prompt template.

   b. Include inference parameters in the `inferenceConfiguration` objects. For more information about inference configurations, see [Inference parameters for foundation models](#).
4. To enable the prompt template, set the `promptCreationMode` to `OVERRIDDEN`.
5. To allow or prevent the agent from performing the step in the `promptType` field, modify the `promptState` value. This setting can be useful for troubleshooting the agent's behavior.

   - If you set `promptState` to `DISABLED` for the `PRE_PROCESSING`, `KNOWLEDGE_BASE_RESPONSE_GENERATION`, or `POST_PROCESSING` steps, the agent skips that step.
   - If you set `promptState` to `DISABLED` for the `ORCHESTRATION` step, the agent sends only the user input to the foundation model in orchestration. In addition, the agent returns the response as is without orchestrating calls between API operations and knowledge bases.
   - By default, the `POST_PROCESSING` step is `DISABLED`. By default, the `PRE_PROCESSING`, `ORCHESTRATION`, and `KNOWLEDGE_BASE_RESPONSE_GENERATION` steps are `ENABLED`.
6. To use a Lambda function that you have defined to parse the raw foundation model output, perform the following steps:

   a. For each prompt template that you want to enable the Lambda function for, set `parserMode` to `OVERRIDDEN`.

b. Specify the Amazon Resource Name (ARN) of the Lambda function in the `overrideLambda` field in the `promptOverrideConfiguration` object.

Placeholder variables in Amazon Bedrock agent prompt templates

You can use placeholder variables in agent prompt templates. The variables will be populated by pre-existing configurations when the prompt template is called. Select a tab to see variables that you can use for each prompt template.

Pre-processing

<table>
<tr><th>Variable</th><th>Models supported</th><th>Replaced by</th></tr>
<tr><td>$functions$</td><td>Anthropic Claude Instant, Claude v2.0</td><td rowspan="2">Action group API operations and knowledge bases configured for the agent.</td></tr>
<tr><td>$tools$</td><td>Anthropic Claude v2.1</td></tr>
<tr><td>$conversation_history$</td><td>All</td><td>Conversation history for the current session</td></tr>
<tr><td>$question$</td><td>All</td><td>User input for the current <code>InvokeAgent</code> call in the session.</td></tr>
</table>

Orchestration

<table>
<tr><th>Variable</th><th>Models supported</th><th>Replaced by</th></tr>
<tr><td>$functions$</td><td>Anthropic Claude Instant, Claude v2.0</td><td rowspan="2">Action group API operations and knowledge bases configured for the agent.</td></tr>
<tr><td>$tools$</td><td>Anthropic Claude v2.1</td></tr>
<tr><td>$agent_scratchpad$</td><td>All</td><td>Designates an area for the model to write down its thoughts and actions it has taken. Replaced by</td></tr>
</table>

Variable	Models supported	Replaced by
		predictions and output of the previous iterations in the current turn. Provides the model with context of what has been achieved for the given user input and what the next step should be.
$any_function_name$	Anthropic Claude Instant, Claude v2.0	A randomly chosen API name from the API names that exist in the agent's action groups.
$conversation_history$	All	Conversation history for the current session
$instruction$	All	Model instructions configure d for the agent.
$prompt_session_attributes $	All	Session attributes preserved across a prompt
$question$	All	User input for the current `InvokeAgent` call in the session.

You can use the following placeholder variables if you allow the agent to ask the user for more information by doing one of the following actions:

- In the console, set in the **User input** in the agent details.
- Set the `parentActionGroupSignature` to `AMAZON.UserInput` with a CreateAgentActionGroup or UpdateAgentActionGroup request.

Variable	Models supported	Replaced by
$ask_user_missing_parameters$	Anthropic Claude Instant, Claude v2.0	Instructions for the model to ask the user to provide required missing information
$ask_user_missing_information$	Anthropic Claude v2.1	
$ask_user_confirm_parameters$	All	Instructions for the model to ask the user to confirm parameters that the agent hasn't yet received or is unsure of.
$ask_user_function$	All	A function to ask the user a question.
$ask_user_function_format$	All	The format of the function to ask the user a question.
$ask_user_input_examples$	All	Few-shot examples to inform the model how to predict when it should ask the user a question.

Knowledge base response generation

Variable	Model	Replaced by
$query$	All	The query generated by the orchestration prompt model response when it predicts the next step to be knowledge base querying.
$search_results$	All	The retrieved results for the user query

Post-processing

Variable	Model	Replaced by
$latest_response$	All	The last orchestration prompt model response
$question$	All	User input for the current `InvokeAgent` call in the session.
$responses$	All	The action group and knowledge base outputs from the current turn.

Parser Lambda function in Agents for Amazon Bedrock

Each prompt template includes a parser Lambda function that you can modify. To write a custom parser Lambda function, you must understand the input event that your agent sends and the response that the agent expects as the output from the Lambda function. You write a handler function to manipulate variables from the input event and to return the response. For more information about how AWS Lambda works, see Event-driven invocation in the AWS Lambda Developer Guide.

Topics

- Parser Lambda input event
- Parser Lambda response
- Parser Lambda examples

Parser Lambda input event

The following is the general structure of the input event from the agent. Use the fields to write your Lambda handler function.

```
{
    "messageVersion": "1.0",
    "agent": {
```

```
        "name": "string",
        "id": "string",
        "alias": "string",
        "version": "string"
    },
    "invokeModelRawResponse": "string",
    "promptType": "ORCHESTRATION | POST_PROCESSING | PRE_PROCESSING |
 KNOWLEDGE_BASE_RESPONSE_GENERATION ",
    "overrideType": "OUTPUT_PARSER"
}
```

The following list describes the input event fields:

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from the Lambda function. Agents for Amazon Bedrock only supports version 1.0.
- `agent` – Contains information about the name, ID, alias, and version of the agent that the prompts belongs to.
- `invokeModelRawResponse` – The raw foundation model output of the prompt whose output is to be parsed.
- `promptType` – The prompt type whose output is to be parsed.
- `overrideType` – The artifacts that this Lambda function overrides. Currently, only `OUTPUT_PARSER` is supported, which indicates that the default parser is to be overridden.

Parser Lambda response

Your agent expects a response from your Lambda function that matches the following format. The agent uses the response for further orchestration or to help it return a response to the user. Use the Lambda function response fields to configure how the output is returned.

```
{
    "messageVersion": "1.0",
    "promptType": "ORCHESTRATION | PRE_PROCESSING | POST_PROCESSING |
 KNOWLEDGE_BASE_RESPONSE_GENERATION",
    "preProcessingParsedResponse": {
        "isValidInput": "boolean",
        "rationale": "string"
    },
    "orchestrationParsedResponse": {
        "rationale": "string",
```

```
        "parsingErrorDetails": {
            "repromptResponse": "string"
        },
        "responseDetails": {
            "invocationType": "ACTION_GROUP | KNOWLEDGE_BASE | FINISH | ASK_USER",
            "agentAskUser": {
                "responseText": "string"
            },
            "actionGroupInvocation": {
                "actionGroupName": "string",
                "apiName": "string",
                "verb": "string",
                "actionGroupInput": {
                    "<parameter>": {
                        "value": "string"
                    },
                    ...
                }
            },
            "agentKnowledgeBase": {
                "knowledgeBaseId": "string",
                "searchQuery": {
                    "value": "string"
                }
            },
            "agentFinalResponse": {
                "responseText": "string",
                "citations": {
                    "generatedResponseParts": [{
                        "text": "string",
                        "references": [{"sourceId": "string"}]
                    }]
                }
            },
        }
    },
    "knowledgeBaseResponseGenerationParsedResponse": {
       "generatedResponse": {
            "generatedResponseParts": [
                {
                    "text": "string",
                    "references": [
                        {"sourceId": "string"},
                        ...
```

```
                    ]
                }
            ]
        }
    },
    "postProcessingParsedResponse": {
        "responseText": "string",
        "citations": {
            "generatedResponseParts": [{
                "text": "string",
                "references": [{
                    "sourceId": "string"
                }]
            }]
        }
    }
}
```

The following list describes the Lambda response fields:

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function. Agents for Amazon Bedrock only supports version 1.0.
- `promptType` – The prompt type of the current turn.
- `preProcessingParsedResponse` – The parsed response for the `PRE_PROCESSING` prompt type.
- `orchestrationParsedResponse` – The parsed response for the `ORCHESTRATION` prompt type. See below for more details.
- `knowledgeBaseResponseGenerationParsedResponse` – The parsed response for the `KNOWLEDGE_BASE_RESPONSE_GENERATION` prompt type.
- `postProcessingParsedResponse` – The parsed response for the `POST_PROCESSING` prompt type.

For more details about the parsed responses for the four prompt templates, see the following tabs.

preProcessingParsedResponse

```
{
    "isValidInput": "boolean",
```

```
    "rationale": "string"
}
```

The `preProcessingParsedResponse` contains the following fields.

- `isValidInput` – Specifies whether the user input is valid or not. You can define the function to determine how to characterize the validity of user input.
- `rationale` – The reasoning for the user input categorization. This rationale is provided by the model in the raw response, the Lambda function parses it, and the agent presents it in the trace for pre-processing.

orchestrationResponse

```
{
    "rationale": "string",
    "parsingErrorDetails": {
        "repromptResponse": "string"
    },
    "responseDetails": {
        "invocationType": "ACTION_GROUP | KNOWLEDGE_BASE | FINISH | ASK_USER",
        "agentAskUser": {
            "responseText": "string"
        },
        "actionGroupInvocation": {
            "actionGroupName": "string",
            "apiName": "string",
            "verb": "string",
            "actionGroupInput": {
                "<parameter>": {
                    "value": "string"
                },
                ...
            }
        },
        "agentKnowledgeBase": {
            "knowledgeBaseId": "string",
            "searchQuery": {
                "value": "string"
            }
        },
        "agentFinalResponse": {
            "responseText": "string",
```

```
            "citations": {
                "generatedResponseParts": [
                    {
                        "text": "string",
                        "references": [
                            {"sourceId": "string"},
                            ...
                        ]
                    },
                    ...
                ]
            }
        },
    }
}
```

The `orchestrationParsedResponse` contains the following fields:

- `rationale` – The reasoning for what to do next, based on the foundation model output. You can define the function to parse from the model output.
- `parsingErrorDetails` – Contains the `repromptResponse`, which is the message to reprompt the model to update its raw response when the model response can't be parsed. You can define the function to manipulate how to reprompt the model.
- `responseDetails` – Contains the details for how to handle the output of the foundation model. Contains an `invocationType`, which is the next step for the agent to take, and a second field that should match the `invocationType`. The following objects are possible.
  - `agentAskUser` – Compatible with the `ASK_USER` invocation type. This invocation type ends the orchestration step. Contains the `responseText` to ask the user for more information. You can define your function to manipulate this field.
  - `actionGroupInvocation` – Compatible with the `ACTION_GROUP` invocation type. You can define your function to determine action groups to invoke and parameters to pass. Contains the following fields:
    - `actionGroupName` – The action group to invoke.
    - `apiName` – The name of the API operation to invoke in the action group.
    - `verb` – The method of the API operation to use.
    - `actionGroupInput` – Contains parameters to specify in the API operation request.

- `agentKnowledgeBase` – Compatible with the `KNOWLEDGE_BASE` invocation type. You can define your function to determine how to query knowledge bases. Contains the following fields:
  - `knowledgeBaseId` – The unique identifier of the knowledge base.
  - `searchQuery` – Contains the query to send to the knowledge base in the `value` field.
- `agentFinalResponse` – Compatible with the `FINISH` invocation type. This invocation type ends the orchestration step. Contains the response to the user in the `responseText` field and citations for the response in the `citations` object.

knowledgeBaseResponseGenerationParsedResponse

```
{
   "generatedResponse": {
        "generatedResponseParts": [
            {
                "text": "string",
                "references": [
                    { "sourceId": "string" },
                    ...
                ]
            },
            ...
        ]
    }
}
```

The `knowledgeBaseResponseGenerationParsedResponse` contains the `generatedResponse` from querying the knowledge base and references for the data sources.

postProcessingParsedResponse

```
{
    "responseText": "string",
    "citations": {
        "generatedResponseParts": [
            {
                "text": "string",
                "references": [
                    { "sourceId": "string" },
                    ...
```

```
                ]
            },
            ...
        ]
    }
}
```

The `postProcessingParsedResponse` contains the following fields:

- `responseText` – The response to return to the end user. You can define the function to format the response.
- `citations` – Contains a list of citations for the response. Each citation shows the cited text and its references.

Parser Lambda examples

To see an example parser Lambda function for a specific prompt template, select from the following tabs. Also shown are example input events sent to the function and responses from it. The `lambda_handler` function returns the parsed response to the agent.

Pre-processing

Example function

```
import json
import re
import logging

PRE_PROCESSING_RATIONALE_REGEX = "<thinking>(.*?)</thinking>"
PREPROCESSING_CATEGORY_REGEX = "<category>(.*?)</category>"
PREPROCESSING_PROMPT_TYPE = "PRE_PROCESSING"
PRE_PROCESSING_RATIONALE_PATTERN = re.compile(PRE_PROCESSING_RATIONALE_REGEX,
 re.DOTALL)
PREPROCESSING_CATEGORY_PATTERN = re.compile(PREPROCESSING_CATEGORY_REGEX, re.DOTALL)

logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default
 PreProcessing prompt
def lambda_handler(event, context):
```

```
    print("Lambda input: " + str(event))
    logger.info("Lambda input: " + str(event))

    prompt_type = event["promptType"]

    # Sanitize LLM response
    model_response = sanitize_response(event['invokeModelRawResponse'])

    if event["promptType"] == PREPROCESSING_PROMPT_TYPE:
        return parse_pre_processing(model_response)

def parse_pre_processing(model_response):

    category_matches = re.finditer(PREPROCESSING_CATEGORY_PATTERN, model_response)
    rationale_matches = re.finditer(PRE_PROCESSING_RATIONALE_PATTERN,
 model_response)

    category = next((match.group(1) for match in category_matches), None)
    rationale = next((match.group(1) for match in rationale_matches), None)

    return {
        "promptType": "PRE_PROCESSING",
        "preProcessingParsedResponse": {
            "rationale": rationale,
            "isValidInput": get_is_valid_input(category)
            }
        }

def sanitize_response(text):
    pattern = r"(\\n*)"
    text = re.sub(pattern, r"\n", text)
    return text

def get_is_valid_input(category):
    if category is not None and category.strip().upper() == "D" or
 category.strip().upper() == "E":
        return True
    return False
```

```
{
    "agent": {
        "alias": "TSTALIASID",
```

```
        "id": "AGENTID123",
        "name": "InsuranceAgent",
        "version": "DRAFT"
    },
    "invokeModelRawResponse": " <thinking>\nThe user is asking about the
 instructions provided to the function calling agent. This input is trying to gather
 information about what functions/API's or instructions our function calling agent
 has access to. Based on the categories provided, this input belongs in Category B.
\n</thinking>\n\n<category>B</category>",
    "messageVersion": "1.0",
    "overrideType": "OUTPUT_PARSER",
    "promptType": "PRE_PROCESSING"
}
```



```
{
  "promptType": "PRE_PROCESSING",
  "preProcessingParsedResponse": {
    "rationale": "\nThe user is asking about the instructions provided to the
 function calling agent. This input is trying to gather information about what
 functions/API's or instructions our function calling agent has access to. Based on
 the categories provided, this input belongs in Category B.\n",
    "isValidInput": false
  }
}
```


Orchestration

The following are example functions for Anthropic Claude 2 and Anthropic Claude 2.1.

Anthropic Claude 2

```
import json
import re
import logging


RATIONALE_REGEX_LIST = [
    "(.*?)(<function_call>)",
    "(.*?)(<answer>)"
]
RATIONALE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_REGEX_LIST]
```

```
RATIONALE_VALUE_REGEX_LIST = [
    "<scratchpad>(.*?)(</scratchpad>)",
    "(.*?)(</scratchpad>)",
    "(<scratchpad>)(.*?)"
]
RATIONALE_VALUE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_VALUE_REGEX_LIST]

ANSWER_REGEX = r"(?<=<answer>)(.*)"
ANSWER_PATTERN = re.compile(ANSWER_REGEX, re.DOTALL)

ANSWER_TAG = "<answer>"
FUNCTION_CALL_TAG = "<function_call>"

ASK_USER_FUNCTION_CALL_REGEX = r"(<function_call>user::askuser)(.*)\)"
ASK_USER_FUNCTION_CALL_PATTERN = re.compile(ASK_USER_FUNCTION_CALL_REGEX, re.DOTALL)

ASK_USER_FUNCTION_PARAMETER_REGEX = r"(?<=askuser=\")(.*?)\""
ASK_USER_FUNCTION_PARAMETER_PATTERN = re.compile(ASK_USER_FUNCTION_PARAMETER_REGEX,
 re.DOTALL)

KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX = "x_amz_knowledgebase_"

FUNCTION_CALL_REGEX = r"<function_call>(\w+)::(\w+)::(.+)\((.+)\)"

ANSWER_PART_REGEX = "<answer_part\\s?>(.+?)</answer_part\\s?>"
ANSWER_TEXT_PART_REGEX = "<text\\s?>(.+?)</text\\s?>"
ANSWER_REFERENCE_PART_REGEX = "<source\\s?>(.+?)</source\\s?>"
ANSWER_PART_PATTERN = re.compile(ANSWER_PART_REGEX, re.DOTALL)
ANSWER_TEXT_PART_PATTERN = re.compile(ANSWER_TEXT_PART_REGEX, re.DOTALL)
ANSWER_REFERENCE_PART_PATTERN = re.compile(ANSWER_REFERENCE_PART_REGEX, re.DOTALL)

# You can provide messages to reprompt the LLM in case the LLM output is not in the
 expected format
MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE = "Missing the argument askuser for
 user::askuser function call. Please try again with the correct argument added"
ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE = "The function call format
 is incorrect. The format for function calls to the askuser function must be:
 <function_call>user::askuser(askuser=\"$ASK_USER_INPUT\")</function_call>."
FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE = 'The function call format
 is incorrect. The format for function calls must be: <function_call>
$FUNCTION_NAME($FUNCTION_ARGUMENT_NAME=""$FUNCTION_ARGUMENT_NAME"")</
function_call>.'
```

```
logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default
 orchestration prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))

    # Sanitize LLM response
    sanitized_response = sanitize_response(event['invokeModelRawResponse'])

    # Parse LLM response for any rationale
    rationale = parse_rationale(sanitized_response)

    # Construct response fields common to all invocation types
    parsed_response = {
        'promptType': "ORCHESTRATION",
        'orchestrationParsedResponse': {
            'rationale': rationale
        }
    }

    # Check if there is a final answer
    try:
        final_answer, generated_response_parts = parse_answer(sanitized_response)
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    if final_answer:
        parsed_response['orchestrationParsedResponse']['responseDetails'] = {
            'invocationType': 'FINISH',
            'agentFinalResponse': {
                'responseText': final_answer
            }
        }

        if generated_response_parts:
            parsed_response['orchestrationParsedResponse']['responseDetails']
['agentFinalResponse']['citations'] = {
                'generatedResponseParts': generated_response_parts
            }

        logger.info("Final answer parsed response: " + str(parsed_response))
        return parsed_response
```

```
    # Check if there is an ask user
    try:
        ask_user = parse_ask_user(sanitized_response)
        if ask_user:
            parsed_response['orchestrationParsedResponse']['responseDetails'] = {
                'invocationType': 'ASK_USER',
                'agentAskUser': {
                    'responseText': ask_user
                }
            }

            logger.info("Ask user parsed response: " + str(parsed_response))
            return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    # Check if there is an agent action
    try:
        parsed_response = parse_function_call(sanitized_response, parsed_response)
        logger.info("Function call parsed response: " + str(parsed_response))
        return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    addRepromptResponse(parsed_response, 'Failed to parse the LLM output')
    logger.info(parsed_response)
    return parsed_response

    raise Exception("unrecognized prompt type")

def sanitize_response(text):
    pattern = r"(\\n*)"
    text = re.sub(pattern, r"\n", text)
    return text

def parse_rationale(sanitized_response):
    # Checks for strings that are not required for orchestration
    rationale_matcher = next((pattern.search(sanitized_response) for pattern in
 RATIONALE_PATTERNS if pattern.search(sanitized_response)), None)

    if rationale_matcher:
```

```
        rationale = rationale_matcher.group(1).strip()

        # Check if there is a formatted rationale that we can parse from the string
        rationale_value_matcher = next((pattern.search(rationale) for pattern in
 RATIONALE_VALUE_PATTERNS if pattern.search(rationale)), None)
        if rationale_value_matcher:
            return rationale_value_matcher.group(1).strip()

        return rationale

    return None

def parse_answer(sanitized_llm_response):
    if has_generated_response(sanitized_llm_response):
        return parse_generated_response(sanitized_llm_response)

    answer_match = ANSWER_PATTERN.search(sanitized_llm_response)
    if answer_match and is_answer(sanitized_llm_response):
        return answer_match.group(0).strip(), None

    return None, None

def is_answer(llm_response):
    return llm_response.rfind(ANSWER_TAG) > llm_response.rfind(FUNCTION_CALL_TAG)

def parse_generated_response(sanitized_llm_response):
    results = []

    for match in ANSWER_PART_PATTERN.finditer(sanitized_llm_response):
        part = match.group(1).strip()

        text_match = ANSWER_TEXT_PART_PATTERN.search(part)
        if not text_match:
            raise ValueError("Could not parse generated response")

        text = text_match.group(1).strip()
        references = parse_references(sanitized_llm_response, part)
        results.append((text, references))

    final_response = " ".join([r[0] for r in results])

    generated_response_parts = []
    for text, references in results:
        generatedResponsePart = {
```

```
            'text': text,
            'references': references
        }
        generated_response_parts.append(generatedResponsePart)

    return final_response, generated_response_parts


def has_generated_response(raw_response):
    return ANSWER_PART_PATTERN.search(raw_response) is not None

def parse_references(raw_response, answer_part):
    references = []
    for match in ANSWER_REFERENCE_PART_PATTERN.finditer(answer_part):
        reference = match.group(1).strip()
        references.append({'sourceId': reference})
    return references

def parse_ask_user(sanitized_llm_response):
    ask_user_matcher = ASK_USER_FUNCTION_CALL_PATTERN.search(sanitized_llm_response)
    if ask_user_matcher:
        try:
            ask_user = ask_user_matcher.group(2).strip()
            ask_user_question_matcher = 
 ASK_USER_FUNCTION_PARAMETER_PATTERN.search(ask_user)
            if ask_user_question_matcher:
                return ask_user_question_matcher.group(1).strip()
            raise ValueError(MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE)
        except ValueError as ex:
            raise ex
        except Exception as ex:
            raise Exception(ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE)

    return None

def parse_function_call(sanitized_response, parsed_response):
    match = re.search(FUNCTION_CALL_REGEX, sanitized_response)
    if not match:
        raise ValueError(FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE)

    verb, resource_name, function = match.group(1), match.group(2), match.group(3)

    parameters = {}
    for arg in match.group(4).split(","):
```

```
        key, value = arg.split("=")
        parameters[key.strip()] = {'value': value.strip('" ')}

    parsed_response['orchestrationParsedResponse']['responseDetails'] = {}

    # Function calls can either invoke an action group or a knowledge base.
    # Mapping to the correct variable names accordingly
    if resource_name.lower().startswith(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX):
        parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'KNOWLEDGE_BASE'
        parsed_response['orchestrationParsedResponse']['responseDetails']
['agentKnowledgeBase'] = {
            'searchQuery': parameters['searchQuery'],
            'knowledgeBaseId':
 resource_name.replace(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX, '')
        }

        return parsed_response

    parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'ACTION_GROUP'
    parsed_response['orchestrationParsedResponse']['responseDetails']
['actionGroupInvocation'] = {
        "verb": verb,
        "actionGroupName": resource_name,
        "apiName": function,
        "actionGroupInput": parameters
    }

    return parsed_response

def addRepromptResponse(parsed_response, error):
    error_message = str(error)
    logger.warn(error_message)

    parsed_response['orchestrationParsedResponse']['parsingErrorDetails'] = {
        'repromptResponse': error_message
    }
```

Anthropic Claude 2.1

```
import logging
import re
```

```
import xml.etree.ElementTree as ET

RATIONALE_REGEX_LIST = [
    "(.*?)(<function_calls>)",
    "(.*?)(<answer>)"
]
RATIONALE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_REGEX_LIST]

RATIONALE_VALUE_REGEX_LIST = [
    "<scratchpad>(.*?)(</scratchpad>)",
    "(.*?)(</scratchpad>)",
    "(<scratchpad>)(.*?)"
]
RATIONALE_VALUE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_VALUE_REGEX_LIST]

ANSWER_REGEX = r"(?<=<answer>)(.*)"
ANSWER_PATTERN = re.compile(ANSWER_REGEX, re.DOTALL)

ANSWER_TAG = "<answer>"
FUNCTION_CALL_TAG = "<function_calls>"

ASK_USER_FUNCTION_CALL_REGEX = r"<tool_name>user::askuser</tool_name>"
ASK_USER_FUNCTION_CALL_PATTERN = re.compile(ASK_USER_FUNCTION_CALL_REGEX, re.DOTALL)

ASK_USER_TOOL_NAME_REGEX = r"<tool_name>((.|\n)*?)</tool_name>"
ASK_USER_TOOL_NAME_PATTERN = re.compile(ASK_USER_TOOL_NAME_REGEX, re.DOTALL)

TOOL_PARAMETERS_REGEX = r"<parameters>((.|\n)*?)</parameters>"
TOOL_PARAMETERS_PATTERN = re.compile(TOOL_PARAMETERS_REGEX, re.DOTALL)

ASK_USER_TOOL_PARAMETER_REGEX = r"<question>((.|\n)*?)</question>"
ASK_USER_TOOL_PARAMETER_PATTERN = re.compile(ASK_USER_TOOL_PARAMETER_REGEX,
 re.DOTALL)


KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX = "x_amz_knowledgebase_"

FUNCTION_CALL_REGEX = r"(?<=<function_calls>)(.*)"

ANSWER_PART_REGEX = "<answer_part\\s?>(.+?)</answer_part\\s?>"
ANSWER_TEXT_PART_REGEX = "<text\\s?>(.+?)</text\\s?>"
ANSWER_REFERENCE_PART_REGEX = "<source\\s?>(.+?)</source\\s?>"
```

```
ANSWER_PART_PATTERN = re.compile(ANSWER_PART_REGEX, re.DOTALL)
ANSWER_TEXT_PART_PATTERN = re.compile(ANSWER_TEXT_PART_REGEX, re.DOTALL)
ANSWER_REFERENCE_PART_PATTERN = re.compile(ANSWER_REFERENCE_PART_REGEX, re.DOTALL)

# You can provide messages to reprompt the LLM in case the LLM output is not in the
 expected format
MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE = "Missing the parameter 'question' for
 user::askuser function call. Please try again with the correct argument added."
ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE = "The function call format is
 incorrect. The format for function calls to the askuser function must be: <invoke>
 <tool_name>user::askuser</tool_name><parameters><question>$QUESTION</question></
parameters></invoke>."
FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE = "The function call format is incorrect.
 The format for function calls must be: <invoke> <tool_name>$TOOL_NAME</tool_name>
 <parameters> <$PARAMETER_NAME>$PARAMETER_VALUE</$PARAMETER_NAME>...</parameters></
invoke>."

logger = logging.getLogger()


# This parser lambda is an example of how to parse the LLM output for the default
 orchestration prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))

    # Sanitize LLM response
    sanitized_response = sanitize_response(event['invokeModelRawResponse'])

    # Parse LLM response for any rationale
    rationale = parse_rationale(sanitized_response)

    # Construct response fields common to all invocation types
    parsed_response = {
        'promptType': "ORCHESTRATION",
        'orchestrationParsedResponse': {
            'rationale': rationale
        }
    }

    # Check if there is a final answer
    try:
        final_answer, generated_response_parts = parse_answer(sanitized_response)
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
```

```
        return parsed_response

    if final_answer:
        parsed_response['orchestrationParsedResponse']['responseDetails'] = {
            'invocationType': 'FINISH',
            'agentFinalResponse': {
                'responseText': final_answer
            }
        }

        if generated_response_parts:
            parsed_response['orchestrationParsedResponse']['responseDetails']
['agentFinalResponse']['citations'] = {
                'generatedResponseParts': generated_response_parts
            }

        logger.info("Final answer parsed response: " + str(parsed_response))
        return parsed_response

    # Check if there is an ask user
    try:
        ask_user = parse_ask_user(sanitized_response)
        if ask_user:
            parsed_response['orchestrationParsedResponse']['responseDetails'] = {
                'invocationType': 'ASK_USER',
                'agentAskUser': {
                    'responseText': ask_user
                }
            }

            logger.info("Ask user parsed response: " + str(parsed_response))
            return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    # Check if there is an agent action
    try:
        parsed_response = parse_function_call(sanitized_response, parsed_response)
        logger.info("Function call parsed response: " + str(parsed_response))
        return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response
```

```
    addRepromptResponse(parsed_response, 'Failed to parse the LLM output')
    logger.info(parsed_response)
    return parsed_response

    raise Exception("unrecognized prompt type")


def sanitize_response(text):
    pattern = r"(\\n*)"
    text = re.sub(pattern, r"\n", text)
    return text


def parse_rationale(sanitized_response):
    # Checks for strings that are not required for orchestration
    rationale_matcher = next(
        (pattern.search(sanitized_response) for pattern in RATIONALE_PATTERNS if
 pattern.search(sanitized_response)),
        None)

    if rationale_matcher:
        rationale = rationale_matcher.group(1).strip()

        # Check if there is a formatted rationale that we can parse from the string
        rationale_value_matcher = next(
            (pattern.search(rationale) for pattern in RATIONALE_VALUE_PATTERNS if
 pattern.search(rationale)), None)
        if rationale_value_matcher:
            return rationale_value_matcher.group(1).strip()

        return rationale

    return None


def parse_answer(sanitized_llm_response):
    if has_generated_response(sanitized_llm_response):
        return parse_generated_response(sanitized_llm_response)

    answer_match = ANSWER_PATTERN.search(sanitized_llm_response)
    if answer_match and is_answer(sanitized_llm_response):
        return answer_match.group(0).strip(), None
```

```
    return None, None


def is_answer(llm_response):
    return llm_response.rfind(ANSWER_TAG) > llm_response.rfind(FUNCTION_CALL_TAG)


def parse_generated_response(sanitized_llm_response):
    results = []

    for match in ANSWER_PART_PATTERN.finditer(sanitized_llm_response):
        part = match.group(1).strip()

        text_match = ANSWER_TEXT_PART_PATTERN.search(part)
        if not text_match:
            raise ValueError("Could not parse generated response")

        text = text_match.group(1).strip()
        references = parse_references(sanitized_llm_response, part)
        results.append((text, references))

    final_response = " ".join([r[0] for r in results])

    generated_response_parts = []
    for text, references in results:
        generatedResponsePart = {
            'text': text,
            'references': references
        }
        generated_response_parts.append(generatedResponsePart)

    return final_response, generated_response_parts


def has_generated_response(raw_response):
    return ANSWER_PART_PATTERN.search(raw_response) is not None


def parse_references(raw_response, answer_part):
    references = []
    for match in ANSWER_REFERENCE_PART_PATTERN.finditer(answer_part):
        reference = match.group(1).strip()
        references.append({'sourceId': reference})
    return references
```

```
def parse_ask_user(sanitized_llm_response):
    ask_user_matcher = ASK_USER_FUNCTION_CALL_PATTERN.search(sanitized_llm_response)
    if ask_user_matcher:
        try:
            parameters_matches =
 TOOL_PARAMETERS_PATTERN.search(sanitized_llm_response)
            params = parameters_matches.group(1).strip()
            ask_user_question_matcher =
 ASK_USER_TOOL_PARAMETER_PATTERN.search(params)
            if ask_user_question_matcher:
                ask_user_question = ask_user_question_matcher.group(1)
                return ask_user_question
            raise ValueError(MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE)
        except ValueError as ex:
            raise ex
        except Exception as ex:
            raise Exception(ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE)

    return None


def parse_function_call(sanitized_response, parsed_response):
    match = re.search(FUNCTION_CALL_REGEX, sanitized_response)
    if not match:
        raise ValueError(FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE)

    tool_name_matches = ASK_USER_TOOL_NAME_PATTERN.search(sanitized_response)
    tool_name = tool_name_matches.group(1)
    parameters_matches = TOOL_PARAMETERS_PATTERN.search(sanitized_response)
    params = parameters_matches.group(1).strip()

    action_split = tool_name.split('::')
    verb = action_split[0].strip()
    resource_name = action_split[1].strip()
    function = action_split[2].strip()

    xml_tree = ET.ElementTree(ET.fromstring("<parameters>{}</
parameters>".format(params)))
    parameters = {}
    for elem in xml_tree.iter():
        if elem.text:
            parameters[elem.tag] = {'value': elem.text.strip('" ')}
```

```
    parsed_response['orchestrationParsedResponse']['responseDetails'] = {}

    # Function calls can either invoke an action group or a knowledge base.
    # Mapping to the correct variable names accordingly
    if resource_name.lower().startswith(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX):
        parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'KNOWLEDGE_BASE'
        parsed_response['orchestrationParsedResponse']['responseDetails']
['agentKnowledgeBase'] = {
            'searchQuery': parameters['searchQuery'],
            'knowledgeBaseId':
 resource_name.replace(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX, '')
        }

        return parsed_response

    parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'ACTION_GROUP'
    parsed_response['orchestrationParsedResponse']['responseDetails']
['actionGroupInvocation'] = {
        "verb": verb,
        "actionGroupName": resource_name,
        "apiName": function,
        "actionGroupInput": parameters
    }

    return parsed_response


def addRepromptResponse(parsed_response, error):
    error_message = str(error)
    logger.warn(error_message)

    parsed_response['orchestrationParsedResponse']['parsingErrorDetails'] = {
        'repromptResponse': error_message
    }
```

Example request

```
{
    'agent': {
        'alias': 'TSTALIASID',
```

```
        'id': 'AGENTID123',
        'name': 'InsuranceAgent',
        'version': 'DRAFT'
    },
    'invokeModelRawResponse': ' To answer this question, I will:\\n\\n1.
 Call the GET::x_amz_knowledgebase_KBID123456::Search function to search
 for a phone number to call.\\n\\nI have checked that I have access to the
 GET::x_amz_knowledgebase_KBID23456::Search function.\\n\\n</scratchpad>\\n\
\n<function_call>GET::x_amz_knowledgebase_KBID123456::Search(searchQuery=\"What is
 the phone number I can call?\")',
    'messageVersion': '1.0',
    'overrideType': 'OUTPUT_PARSER',
    'promptType': 'ORCHESTRATION'
}
```

Example response

```
{
    'promptType': 'ORCHESTRATION',
    'orchestrationParsedResponse': {
        'rationale': 'To answer this question, I will:\\n\\n1. Call the
 GET::x_amz_knowledgebase_KBID123456::Search function to search for a phone
 number to call Farmers.\\n\\nI have checked that I have access to the
 GET::x_amz_knowledgebase_KBID123456::Search function.',
        'responseDetails': {
            'invocationType': 'KNOWLEDGE_BASE',
            'agentKnowledgeBase': {
                'searchQuery': {'value': 'What is the phone number I can call?'},
                'knowledgeBaseId': 'KBID123456'
            }
        }
    }
}
```

Knowledge base response generation

Example function

```
import json
import re
import logging

ANSWER_PART_REGEX = "<answer_part\\s?>(.+?)</answer_part\\s?>"
```

```
ANSWER_TEXT_PART_REGEX = "<text\\s?>(.+?)</text\\s?>"
ANSWER_REFERENCE_PART_REGEX = "<source\\s?>(.+?)</source\\s?>"
ANSWER_PART_PATTERN = re.compile(ANSWER_PART_REGEX, re.DOTALL)
ANSWER_TEXT_PART_PATTERN = re.compile(ANSWER_TEXT_PART_REGEX, re.DOTALL)
ANSWER_REFERENCE_PART_PATTERN = re.compile(ANSWER_REFERENCE_PART_REGEX, re.DOTALL)

logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default KB
 response generation prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))
    raw_response = event['invokeModelRawResponse']

    parsed_response = {
        'promptType': 'KNOWLEDGE_BASE_RESPONSE_GENERATION',
        'knowledgeBaseResponseGenerationParsedResponse': {
            'generatedResponse': parse_generated_response(raw_response)
        }
    }

    logger.info(parsed_response)
    return parsed_response

def parse_generated_response(sanitized_llm_response):
    results = []

    for match in ANSWER_PART_PATTERN.finditer(sanitized_llm_response):
        part = match.group(1).strip()

        text_match = ANSWER_TEXT_PART_PATTERN.search(part)
        if not text_match:
            raise ValueError("Could not parse generated response")

        text = text_match.group(1).strip()
        references = parse_references(sanitized_llm_response, part)
        results.append((text, references))

    generated_response_parts = []
    for text, references in results:
        generatedResponsePart = {
            'text': text,
            'references': references
        }
```

```
        generated_response_parts.append(generatedResponsePart)

    return {
        'generatedResponseParts': generated_response_parts
    }

def parse_references(raw_response, answer_part):
    references = []
    for match in ANSWER_REFERENCE_PART_PATTERN.finditer(answer_part):
        reference = match.group(1).strip()
        references.append({'sourceId': reference})
    return references
```

Example request

```
{
    'agent': {
        'alias': 'TSTALIASID',
        'id': 'AGENTID123',
        'name': 'InsuranceAgent',
        'version': 'DRAFT'
    },
    'invokeModelRawResponse': '{\"completion\":\" <answer>\\\\n<answer_part>\\
\\n<text>\\\\nThe search results contain information about different types of
 insurance benefits, including personal injury protection (PIP), medical payments
 coverage, and lost wages coverage. PIP typically covers reasonable medical
 expenses for injuries caused by an accident, as well as income continuation,
 child care, loss of services, and funerals. Medical payments coverage provides
 payment for medical treatment resulting from a car accident. Who pays lost wages
 due to injuries depends on the laws in your state and the coverage purchased.
\\\\n</text>\\\\n<sources>\\\\n<source>1234567-1234-1234-1234-123456789abc</
source>\\\\n<source>2345678-2345-2345-2345-23456789abcd</source>\\\
\n<source>3456789-3456-3456-3456-3456789abcde</source>\\\\n</sources>\\\\n</
answer_part>\\\\n</answer>\",\"stop_reason\":\"stop_sequence\",\"stop\":\"\\\\n\\\
\nHuman:\"}',
    'messageVersion': '1.0',
    'overrideType': 'OUTPUT_PARSER',
    'promptType': 'KNOWLEDGE_BASE_RESPONSE_GENERATION'
}
```

Example response

```
{
```

```
    'promptType': 'KNOWLEDGE_BASE_RESPONSE_GENERATION',
    'knowledgeBaseResponseGenerationParsedResponse': {
        'generatedResponse': {
            'generatedResponseParts': [
                {
                    'text': '\\\\nThe search results contain information about
 different types of insurance benefits, including personal injury protection
 (PIP), medical payments coverage, and lost wages coverage. PIP typically covers
 reasonable medical expenses for injuries caused by an accident, as well as income
 continuation, child care, loss of services, and funerals. Medical payments coverage
 provides payment for medical treatment resulting from a car accident. Who pays lost
 wages due to injuries depends on the laws in your state and the coverage purchased.
\\\\n',
                    'references': [
                        {'sourceId': '1234567-1234-1234-1234-123456789abc'},
                        {'sourceId': '2345678-2345-2345-2345-23456789abcd'},
                        {'sourceId': '3456789-3456-3456-3456-3456789abcde'}
                    ]
                }
            ]
        }
    }
}
```

Post-processing

Example function

```
import json
import re
import logging

FINAL_RESPONSE_REGEX = r"<final_response>([\s\S]*?)</final_response>"
FINAL_RESPONSE_PATTERN = re.compile(FINAL_RESPONSE_REGEX, re.DOTALL)

logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default
 PostProcessing prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))
    raw_response = event['invokeModelRawResponse']

    parsed_response = {
```

```
        'promptType': 'POST_PROCESSING',
        'postProcessingParsedResponse': {}
    }

    matcher = FINAL_RESPONSE_PATTERN.search(raw_response)
    if not matcher:
        raise Exception("Could not parse raw LLM output")
    response_text = matcher.group(1).strip()

    parsed_response['postProcessingParsedResponse']['responseText'] = response_text

    logger.info(parsed_response)
    return parsed_response
```

Example request

```
{
    'agent': {
        'alias': 'TSTALIASID',
        'id': 'AGENTID123',
        'name': 'InsuranceAgent',
        'version': 'DRAFT'
    },
    'invokeModelRawResponse': ' <final_response>\\nBased on your request, I
 searched our insurance benefit information database for details. The search
 results indicate that insurance policies may cover different types of benefits,
 depending on the policy and state laws. Specifically, the results discussed
 personal injury protection (PIP) coverage, which typically covers medical
 expenses for insured individuals injured in an accident (cited sources:
 1234567-1234-1234-1234-123456789abc, 2345678-2345-2345-2345-23456789abcd). PIP may
 pay for costs like medical care, lost income replacement, childcare expenses, and
 funeral costs. Medical payments coverage was also mentioned as another option that
 similarly covers medical treatment costs for the policyholder and others injured in
 a vehicle accident involving the insured vehicle. The search results further noted
 that whether lost wages are covered depends on the state and coverage purchased.
 Please let me know if you need any clarification or have additional questions.\\n</
final_response>',
    'messageVersion': '1.0',
    'overrideType': 'OUTPUT_PARSER',
    'promptType': 'POST_PROCESSING'
}
```

Example response

```
{
    'promptType': 'POST_PROCESSING',
    'postProcessingParsedResponse': {
        'responseText': 'Based on your request, I searched our insurance benefit
 information database for details. The search results indicate that insurance
 policies may cover different types of benefits, depending on the policy and
 state laws. Specifically, the results discussed personal injury protection
 (PIP) coverage, which typically covers medical expenses for insured individuals
 injured in an accident (cited sources: 24c62d8c-3e39-4ca1-9470-a91d641fe050,
 197815ef-8798-4cb1-8aa5-35f5d6b28365). PIP may pay for costs like medical care,
 lost income replacement, childcare expenses, and funeral costs. Medical payments
 coverage was also mentioned as another option that similarly covers medical
 treatment costs for the policyholder and others injured in a vehicle accident
 involving the insured vehicle. The search results further noted that whether lost
 wages are covered depends on the state and coverage purchased. Please let me know
 if you need any clarification or have additional questions.'
    }
}
```

Control session context

For greater control of session context, you can modify the SessionState object in your agent. The SessionState object contains two types of attributes that you can use to provide conversational context for the agent during user conversations.

- **sessionAttributes** – Attributes that persist over a session between a user and agent. All InvokeAgent requests made with the same `sessionId` belong to the same session, as long as the session time limit (the `idleSessionTTLinSeconds`) has not been surpassed.
- **promptSessionAttributes** – Attributes that persist over a single turn (one InvokeAgent call). You can use the $prompt_session_attributes$ placeholder when you edit the orchestration base prompt template. This placeholder will be populated at runtime with the attributes that you specify in the `promptSessionAttributes` field.

The general format of the SessionState object is as follows.

```
{
    "sessionAttributes": {
        "<attributeName1>": "<attributeValue1>",
        "<attributeName2>": "<attributeValue2>",
```

```
        ...
    },
    "promptSessionAttributes": {
        "<attributeName3>": "<attributeValue3>",
        "<attributeName4>": "<attributeValue4>",
        ...
    }
}
```

You can define the session state attributes at two different steps.

- When you set up an action group and write the Lambda function, include `sessionAttributes` or `promptSessionAttributes` in the response event that is returned to Amazon Bedrock.
- During runtime, when you send an InvokeAgent request, include a `sessionState` object in the request body to dynamically change the session state attributes in the middle of the conversation.

Session attribute example

The following example uses a session attribute to personalize a message to your user.

1. Write your application code to ask the user to provide their first name and the request they want to make to the agent and to store the answers as the variables *`<first_name>`* and *`<request>`*.

2. Write your application code to send an InvokeAgent request with the following body:

```
{
    "inputText": "<request>",
    "sessionState": {
        "sessionAttributes": {
            "firstName": "<first_name>"
        }
    }
}
```

3. When a user uses your application and provides their first name, your code will send the first name as a session attribute and the agent will store their first name for the duration of the session.

4. Because session attributes are sent in the Lambda input event, you can refer to these session attributes in a Lambda function for an action group. For example, if the action API schema requires a first name in the request body, you can use the `firstName` session attribute when writing the Lambda function for an action group to automatically populate that field when sending the API request.

Prompt session attribute example

The following general example uses a prompt session attribute to provide temporal context for the agent.

1. Write your application code to store the user request in a variable called *<request>*.
2. Write your application code to retrieve the time zone at the user's location if the user uses a word indicating relative time (such as "tomorrow") in the *<request>*, and store in a variable called *<timezone>*.
3. Write your application to send an InvokeAgent request with the following body:

```
{
    "inputText": "<request>",
    "sessionState": {
        "promptSessionAttributes": {
            "timeZone": "<timezone>"
        }
    }
}
```

4. If a user uses a word indicating relative time, your code will send the `timeZone` prompt session attribute and the agent will store it for the duration of the turn.
5. For example, if a user asks **`I need to book a hotel for tomorrow`**, your code sends the user's time zone to the agent and the agent can determine the exact date that "tomorrow" refers to.
6. The prompt session attribute can be used at the following steps.
   - If you include the $prompt_session_attributes$ placeholder in the orchestration prompt template, the orchestration prompt to the FM includes the prompt session attributes.
   - Prompt session attributes are sent in the Lambda input event and can be used to help populate API requests or returned in the response.

Optimize performance for Amazon Bedrock agents

This topic provides describes optimizations for agents with specific use cases.

Topics

Optimize performance for Amazon Bedrock agents using a single knowledge base

Agents for Amazon Bedrock offers options to choose different flows that can optimize on latency for simpler use cases in which agents have a single knowledge base. To ensure that your agent is able to take advantage of this optimization, check that the following conditions apply to the relevant version of your agent:

- Your agent contains only one knowledge base.
- Your agent contains no action groups or they are all disabled.
- Your agent doesn't request more information from the user if it doesn't have enough information.
- Your agent is using the default orchestration prompt template.

To learn how to check for these conditions, select the tab corresponding to your method of choice and follow the steps.

Console

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agent overview** section, check that the **User input** field is **DISABLED**.
4. If you're checking if the optimization is being applied to the working draft of the agent, select the **Working draft** in the **Working draft** section. If you're checking if the optimization is being applied to a version of the agent, select the version in the **Versions** section.
5. Check that the **Knowledge bases** section contains only one knowledge base. If there's more than one knowledge base, disable all of them except for one. To learn how to disable knowledge bases, see Manage agent-knowledge bases associations.

6. Check that the **Action groups** section contains no action groups. If there are action groups, disable all of them. To learn how to disable action groups, see Edit an action group.
7. In the **Advanced prompts** section, check that the **Orchestration** field value is **Default**. If it's **Overridden**, choose **Edit** (if you're viewing a version of your agent, you must first navigate to the working draft) and do the following:

   a. In the **Advanced prompts** section, select the **Orchestration** tab.
   b. If you revert the template to the default settings, your custom prompt template will be deleted. Make sure to save your template if you need it later.
   c. Clear **Override orchestration template defaults**. Confirm the message that appears.
8. To apply any changes you've made, select **Prepare** at the top of the **Agent details** page or in the test window. Then, test the agent's optimized performance by submitting a message in the test window.
9. (Optional) If necessary, create a new version of your agent by following the steps at Deploy an Amazon Bedrock agent.

API

1. Send a ListAgentKnowledgeBases request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the ID of your agent. For the `agentVersion`, use `DRAFT` for the working draft or specify the relevant version. In the response, check that `agentKnowledgeBaseSummaries` contains only one object (corresponding to one knowledge base). If there's more than one knowledge base, disable all of them except for one. To learn how to disable knowledge bases, see Manage agent-knowledge bases associations.
2. Send a ListAgentActionGroups request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the ID of your agent. For the `agentVersion`, use `DRAFT` for the working draft or specify the relevant version. In the response, check that the `actionGroupSummaries` list is empty. If there are action groups, disable all of them. To learn how to disable action groups, see Edit an action group.
3. Send a GetAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the ID of your agent. In the response, within the `promptConfigurations` list in the `promptOverrideConfiguration` field, look for the PromptConfiguration object whose

promptType value is ORCHESTRATION. If the promptCreationMode value is DEFAULT, you don't have to do anything. If it's OVERRIDDEN, do the following to revert the template to the default settings:

   a. If you revert the template to the default settings, your custom prompt template will be deleted. Make sure to save your template from the basePromptTemplate field if you need it later.

   b. Send an UpdateAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. For the PromptConfiguration object corresponding to the orchestration template, set the value of promptCreationMode to DEFAULT.

4. To apply any changes you've made, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Then, test the agent's optimized performance by submitting an InvokeAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint, using the TSTALIASID alias of the agent.

5. (Optional) If necessary, create a new version of your agent by following the steps at Deploy an Amazon Bedrock agent.

Deploy an Amazon Bedrock agent

When you first create an Amazon Bedrock agent, you have a working draft version (DRAFT) and a test alias (TSTALIASID) that points to the working draft version. When you make changes to your agent, the changes apply to the working draft. You iterate on your working draft until you're satisfied with the behavior of your agent. Then, you can set up your agent for deployment and integration into your application by creating *aliases* of your agent.

To deploy your agent, you must create an *alias*. During alias creation, Amazon Bedrock creates a version of your agent automatically. The alias points to this newly created version. Alternatively, you can point the alias to a previously created version of your agent. Then, you configure your application to make API calls to that alias.

A *version* is a snapshot that preserves the resource as it exists at the time it was created. You can continue to modify the working draft and create new aliases (and consequently, versions) of your agent as necessary. In Amazon Bedrock, you create a new version of your agent by creating an alias that points to the new version by default. Amazon Bedrock creates versions in numerical order, starting from 1.

Versions are immutable because they act as a snapshot of your agent at the time you created it. To make updates to an agent in production, you must create a new version and set up your application to make calls to the alias that points to that version.

With aliases, you can switch efficiently between different versions of your agent without requiring the application to keep track of the version. For example, you can change an alias to point to a previous version of your agent if there are changes that you need to revert quickly.

To deploy your agent

1. Create an alias and version of your agent. Select the tab corresponding to your method of choice and follow the steps.

   Console

   To create an alias (and optionally a new version)

   1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
   2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
   3. In the **Aliases** section, choose **Create**.
   4. Enter a unique name for the alias and provide an optional description.
   5. Choose one of the following options:
      - To create a new version, choose **Create a new version and to associate it to this alias**.
      - To use an existing version, choose **Use an existing version to associate this alias**. From the dropdown menu, choose the version that you want to associate the alias to.
   6. Select **Create alias**. A success banner appears at the top.

   API

   To create an alias for an agent, send a CreateAgentAlias request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. To create a new version and associate this alias with it, leave the `routingConfiguration` object unspecified.

See code examples

2. Deploy your agent by setting up your application to make an InvokeAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint. In the `agentAliasId` field, specify the ID of the alias pointing to the version of the agent that you want to use.

To learn how to manage versions and aliases of agents, select from the following topics.

Topics

- Manage versions of agents in Amazon Bedrock
- Manage aliases of agents in Amazon Bedrock

Manage versions of agents in Amazon Bedrock

After you create a version of your agent, you can view information about it or delete it. You can only create a new version of an agent by creating a new alias.

Topics

- View information about versions of agents in Amazon Bedrock
- Delete a version of an agent in Amazon Bedrock

View information about versions of agents in Amazon Bedrock

To learn how to view information about the versions of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a version of an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose the version to view from the **Versions** section.

4. To view details about the model, action groups, or knowledge bases attached to version of the agent, choose the name of the information that you want to view. You can't modify any part of a version. To make modifications to the agent, use the working draft and create a new version.

API

To get information about an agent version, send a GetAgentVersion request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentVersion`.

To list information about an agent's versions, send a ListAgentVersions request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`. You can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

Delete a version of an agent in Amazon Bedrock

To learn how to delete a version of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To delete a version of an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.

3. To choose the version for deletion, in the **Versions** section, choose the option button next to the version that you want to delete.
4. Choose **Delete**.
5. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the version, enter **`delete`** in the input field and choose **Delete**.
6. A banner appears to inform you that the version is being deleted. When deletion is complete, a success banner appears.

API

To delete a version of an agent, send a DeleteAgentVersion request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

Manage aliases of agents in Amazon Bedrock

After you create an alias of your agent, you can view information about it, edit it, or delete it.

Topics

- View information about aliases of agents in Amazon Bedrock
- Edit an alias of an agent in Amazon Bedrock
- Delete an alias of an agent in Amazon Bedrock

View information about aliases of agents in Amazon Bedrock

To learn how to view information about the aliases of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To view the details of an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose the alias to view from the **Aliases** section.
4. You can view the name and description of the alias and tags that are associated with the alias.

API

To get information about an agent alias, send a GetAgentAlias request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentAliasId`.

To list information about an agent's aliases, send a ListAgentVersions request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`. You can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

To view all the tags for an alias, send a ListTagsForResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the alias.

Edit an alias of an agent in Amazon Bedrock

To learn how to edit an alias of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To edit an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Aliases** section, choose the option button next to the alias that you want to edit.
4. You can edit the name and description of the alias. Additionally, you can perform one of the following actions:
   - To create a new version and associate this alias with that version, choose **Create a new version and associate it to this alias**.
   - To associate this alias with a different existing version, choose **Use an existing version and associate this alias**.

To add or remove tags associated with an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose the alias for which you want to manage tags from the **Aliases** section.
4. In the **Tags** section, choose **Manage tags**.
5. To add a tag, choose **Add new tag**. Then enter a **Key** and optionally enter a **Value**. To remove a tag, choose **Remove**. For more information, see Tag resources.
6. When you're done editing tags, choose **Submit**.

API

To edit an agent alias, send an UpdateAgentAlias request. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same.

To add tags to an alias, send a TagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the alias. The request body contains a `tags` field, which is an object containing a key-value pair that you specify for each tag.

To remove tags from an alias, send an UntagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the alias. The `tagKeys` request parameter is a list containing the keys for the tags that you want to remove.

Delete an alias of an agent in Amazon Bedrock

To learn how to delete an alias of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To delete an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. To choose the alias for deletion, in the **Aliases** section, choose the option button next to the alias that you want to delete.
4. Choose **Delete**.
5. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the alias, enter **`delete`** in the input field and choose **Delete**.
6. A banner appears to inform you that the alias is being deleted. When deletion is complete, a success banner appears.

API

To delete an alias of an agent, send a DeleteAgentAlias request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

See code examples

Custom models

Model customization is the process of providing training data to a model in order to improve its performance for specific use-cases. You can customize Amazon Bedrock foundation models in order to improve their performance and create a better customer experience. Amazon Bedrock currently provides the following customization methods.

- **Continued Pre-training**

  Provide *unlabeled* data to pre-train a foundation model by familiarizing it with certain types of inputs. You can provide data from specific topics in order to expose a model to those areas. The Continued Pre-training process will tweak the model parameters to accommodate the input data and improve its domain knowledge.

  For example, you can train a model with private data, such as business documents, that are not publically available for training large language models. Additionally, you can continue to improve the model by retraining the model with more unlabeled data as it becomes available.
- **Fine-tuning**

  Provide *labeled* data in order to train a model to improve performance on specific tasks. By providing a training dataset of labeled examples, the model learns to associate what types of outputs should be generated for certain types of inputs. The model parameters are adjusted in the process and the model's performance is improved for the tasks represented by the training dataset.

For information about model customization quotas, see [Model customization quotas](#).

> **Note**
>
> You are charged for model training based on the number of tokens processed by the model (number of tokens in training data corpus × number of epochs) and model storage charged per month per model. For more information, see [Amazon Bedrock pricing](#).

You carry out the following steps in model customization.

1. [Create a training and, if applicable, a validation dataset](#) for your customization task.

2. If you plan to use a new custom IAM role, set up IAM permissions to access the S3 buckets for your data. You can also use an existing role or let the console automatically create a role with the proper permissions.
3. (Optional) Configure KMS keys and/or VPC for extra security.
4. Create a Fine-tuning or Continued Pre-training job, controlling the training process by adjusting the hyperparameter values.
5. Analyze the results by looking at the training or validation metrics or by using model evaluation.
6. Purchase Provisioned Throughput for your newly created custom model.
7. Use your custom model as you would a base model in Amazon Bedrock tasks, such as model inference.

Topics

Supported regions and models for model customization

The following table shows regional support for each customization method:

Region	Fine-tuning	Continued pre-training
US East (N. Virginia)	Yes	Yes
US West (Oregon)	Yes	Yes

Region	Fine-tuning	Continued pre-training
AWS GovCloud (US-West)	Yes	No

The following table shows model support for each customization method:

Model name	Model ID	Fine-tuning	Continued pre-training
Amazon Titan Text G1 - Express	amazon.titan-text-express-v1	Yes	Yes
Amazon Titan Text G1 - Lite	amazon.titan-text-lite-v1	Yes	Yes
Amazon Titan Image Generator G1	amazon.titan-image-generator-v1	Yes	No
Amazon Titan Multimodal Embeddings G1 G1	amazon.titan-embed-image-v1	Yes	No
Cohere Command	cohere.command-text-v14	Yes	No
Cohere Command Light	cohere.command-light-text-v14	Yes	No
Meta Llama 2 13B	meta.llama2-13b-chat-v1	Yes	No
Meta Llama 2 70B	meta.llama2-70b-chat-v1	Yes	No

Prerequisites for model customization

Before you can start a model customization job, you need to fulfill the following prerequisites:

1. Determine whether you plan to carry out a Fine-tuning or Continued Pre-training job and which model you plan to use. The choice you make determines the format of the datasets that you feed into the customization job.
2. Prepare the training dataset file. If the customization method and model that you choose supports a validation dataset, you can also prepare a validation dataset file. Follow the steps below in Prepare the datasets and then upload the files to an Amazon S3 bucket.
3. (Optional) Create a custom AWS Identity and Access Management (IAM) service role with the proper permissions by following the instructions at Create a service role for model customization to set up the role. You can skip this prerequisite if you plan to use the AWS Management Console to automatically create a service role for you.
4. (Optional) Set up extra security configurations.
   - You can encrypt input and output data, customization jobs, or inference requests made to custom models. For more information, see Encryption of model customization jobs and artifacts.
   - You can create a virtual private cloud (VPC) to protect your customization jobs. For more information, see Protect model customization jobs using a VPC.

Topics

Prepare the datasets

Before you can begin a model customization job, you need to minimally prepare a training dataset. Whether a validation dataset is supported and the format of your training and validation dataset depend on the following factors.

- The type of customization job (fine-tuning or Continued Pre-training).
- The input and output modalities of the data.

To see dataset and file requirements for different models, see Model customization quotas.

Select the tab that is relevant to your use-case.

Fine-tuning: Text-to-text

To fine-tune a text-to-text model, prepare a training and optional validation dataset by creating a JSONL file with multiple JSON lines. Each JSON line is a sample containing both a `prompt` and `completion` field. Use 6 characters per token as an approximation for the number of tokens. The format is as follows.

```
{"prompt": "<prompt1>", "completion": "<expected generated text>"}
{"prompt": "<prompt2>", "completion": "<expected generated text>"}
{"prompt": "<prompt3>", "completion": "<expected generated text>"}
```

The following is an example item for a question-answer task:

```
{"prompt": "what is AWS", "completion": "it's Amazon Web Services"}
```

Fine-tuning: Text-to-image & Image-to-embeddings

To fine-tune a text-to-image or image-to-embedding model, prepare a training dataset by create a JSONL file with multiple JSON lines. Validation datasets are not supported. Each JSON line is a sample containing an `image-ref`, the Amazon S3 URI for an image, and a `caption` that could be a prompt for the image.

The images must be in JPEG or PNG format.

```
{"image-ref": "s3://bucket/path/to/image001.png", "caption": "<prompt text>"}
{"image-ref": "s3://bucket/path/to/image002.png", "caption": "<prompt text>"}
{"image-ref": "s3://bucket/path/to/image003.png", "caption": "<prompt text>"}
```

The following is an example item:

```
{"image-ref": "s3://my-bucket/my-pets/cat.png", "caption": "an orange cat with white
 spots"}
```

To allow Amazon Bedrock access to the image files, add an IAM policy similar to the one in [Permissions to access training and validation files and to write output files in S3](#) to the Amazon Bedrock model customization service role that you set up or that was automatically set up for you in the console. The Amazon S3 paths you provide in the training dataset must be in folders that you specify in the policy.

Continued Pre-training: Text-to-text

To carry out Continued Pre-training on a text-to-text model, prepare a training and optional validation dataset by creating a JSONL file with multiple JSON lines. Because Continued Pre-training involves unlabeled data, each JSON line is a sample containing only an `input` field. Use 6 characters per token as an approximation for the number of tokens. The format is as follows.

```
{"input": "<input text>"}
{"input": "<input text>"}
{"input": "<input text>"}
```

The following is an example item that could be in the training data.

```
{"input": "AWS stands for Amazon Web Services"}
```

Protect model customization jobs using a VPC

When you run a model customization job, the job accesses your Amazon S3 bucket to download the input data and to upload job metrics. To control access to your data, we recommend that you use a virtual private cloud (VPC) with Amazon VPC. You can further protect your data by configuring your VPC so that your data isn't available over the internet and instead creating a VPC interface endpoint with AWS PrivateLink to establish a private connection to your data. For more information about how Amazon VPC and AWS PrivateLink integrate with Amazon Bedrock, see Protect your data using Amazon VPC and AWS PrivateLink.

Carry out the following steps to configure and use a VPC for the training, validation, and output data for your model customization jobs.

Topics

Set up a VPC

You can use a default VPC for your model customization data or create a new VPC by following the guidance at Get started with Amazon VPC and Create a VPC.

When you create your VPC, we recommend that you use the default DNS settings for your endpoint route table, so that standard Amazon S3 URLs (for example, `http://s3-aws-region.amazonaws.com/training-bucket`) resolve.

Create an Amazon S3 VPC Endpoint

If you configure your VPC with no internet access, you need to create an Amazon S3 VPC endpoint to allow your model customization jobs to access the S3 buckets that store your training and validation data and that will store the model artifacts.

Create the S3 VPC endpoint by following the steps at Create a gateway endpoint for Amazon S3.

> **Note**
>
> If you don't use the default DNS settings for your VPC, you need to ensure that the URLs for the locations of the data in your training jobs resolve by configuring the endpoint route tables. For information about VPC endpoint route tables, see Routing for Gateway endpoints.

(Optional) Use IAM policies to restrict access to your S3 files

You can use resource-based policies to more tightly control access to your S3 files. You can use any combination of the following types of resource-based policies.

- **Endpoint policies** – Endpoint policies restrict access through the VPC endpoint. The default endpoint policy allows full access to Amazon S3 for any user or service in your VPC. While creating or after you create the endpoint, you can optionally attach a resource-based policy to the endpoint to add restrictions, such as only allowing the endpoint to access a specific bucket or only allowing a specific IAM role to access the endpoint. For examples, see Edit the VPC endpoint policy.

  The following is an example policy you can attach to your VPC endpoint to only allow it to access the bucket containing your training data.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "RestrictAccessToTrainingBucket",
            "Effect": "Allow",
            "Principal": "*",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::training-bucket",
                "arn:aws:s3:::training-bucket/*"
            ]
        }
    ]
}
```

- **Bucket policies** – Bucket policies restrict access to S3 buckets. You can use a bucket policy to restrict access to traffic that comes from your VPC. To attach a bucket policy, follow the steps at Using bucket policies and use the aws:sourceVpc, aws:sourceVpce, or aws:VpcSourceIp condition keys. For examples, see Control access using bucket policies.

  The following is an example policy you can attach to the S3 bucket that will contain your output data to deny all traffic to the bucket unless it comes from your VPC.

```
{
    "Version": "2012-10-17",
    "Statement": [{
    "Sid": "RestrictAccessToOutputBucket",
            "Effect": "Deny",
            "Principal": "*",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::output-bucket",
                "arn:aws:s3:::output-bucket/*"
            ],
```

```
                "Condition": {
                    "StringNotEquals": {
                      "aws:sourceVpc": "your-vpc-id"
                    }
                }
            }
        ]
    }
```

Attach VPC permissions to a model customization role

After you finish setting up your VPC and endpoint, you need to attach the following permissions to your model customization IAM role. Modify this policy to allow access to only the VPC resources that your job needs. Replace the *subnet-ids* and *security-group-id* with the values from your VPC.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeNetworkInterfaces",
                "ec2:DescribeVpcs",
                "ec2:DescribeDhcpOptions",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
                    ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
            ],
            "Resource":[
               "arn:aws:ec2:region:account-id:network-interface/*"
            ],
            "Condition": {
               "StringEquals": {
                   "aws:RequestTag/BedrockManaged": ["true"]
                },
```

```
                "ArnEquals": {
                    "aws:RequestTag/BedrockModelCustomizationJobArn":
["arn:aws:bedrock:region:account-id:model-customization-job/*"]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterface",
            ],
            "Resource":[
               "arn:aws:ec2:region:account-id:subnet/subnet-id",
               "arn:aws:ec2:region:account-id:subnet/subnet-id2",
               "arn:aws:ec2:region:account-id:security-group/security-group-id"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateNetworkInterfacePermission",
                "ec2:DeleteNetworkInterface",
                "ec2:DeleteNetworkInterfacePermission",
            ],
            "Resource": "*",
            "Condition": {
               "ArnEquals": {
                   "ec2:Subnet": [
                       "arn:aws:ec2:region:account-id:subnet/subnet-id",
                       "arn:aws:ec2:region:account-id:subnet/subnet-id2"
                   ],
                   "ec2:ResourceTag/BedrockModelCustomizationJobArn":
["arn:aws:bedrock:region:account-id:model-customization-job/*"]
               },
               "StringEquals": {
                   "ec2:ResourceTag/BedrockManaged": "true"
               }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateTags"
            ],
```

```
            "Resource": "arn:aws:ec2:region:account-id:network-interface/*",
            "Condition": {
                "StringEquals": {
                    "ec2:CreateAction": [
                        "CreateNetworkInterface"
                    ]
                },
                "ForAllValues:StringEquals": {
                    "aws:TagKeys": [
                        "BedrockManaged",
                        "BedrockModelCustomizationJobArn"
                    ]
                }
        }
    ]
}
```

Add the VPC configuration when submitting a model customization job

After you configure the VPC and the required roles and permissions as described in the previous sections, you can create a model customization job that uses this VPC.

When you specify the VPC subnets and security groups for a job, Amazon Bedrock creates *elastic network interfaces* (ENIs) that are associated with your security groups in one of the subnets. ENIs allow the Amazon Bedrock job to connect to resources in your VPC. For information about ENIs, see Elastic Network Interfaces in the *Amazon VPC User Guide*. Amazon Bedrock tags ENIs that it creates with `BedrockManaged` and `BedrockModelCusomizationJobArn` tags.

We recommend that you provide at least one subnet in each Availability Zone.

You can use security groups to establish rules for controlling Amazon Bedrock access to your VPC resources.

You can configure the VPC to use in either the console or through the API. Select the tab corresponding to your method of choice and follow the steps.

Console

For the Amazon Bedrock console, you specify VPC subnets and security groups in the optional **VPC settings** section when you create the model customization job. For more information about configuring jobs, see Submit a model customization job.

> **Note**
>
> For a job that includes VPC configuration, the console can't automatically create a service role for you. Follow the guidance at Create a service role for model customization to create a custom role.

API

When you submit a CreateModelCustomizationJob request, you can include a `VpcConfig` as a request parameter to specify the VPC subnets and security groups to use, as in the following example.

```
"VpcConfig": {
"SecurityGroupIds": [
    "sg-0123456789abcdef0"
],
"Subnets": [
    "subnet-0123456789abcdef0",
    "subnet-0123456789abcdef1",
    "subnet-0123456789abcdef2"
]
}
```

Submit a model customization job

You can create a custom model by using Fine-tuning or Continued Pre-training in the Amazon Bedrock console or API. The customization job can take several hours. The duration of the job depends on the size of the training data (number of records, input tokens, and output tokens), number of epochs, and batch size. Select the tab corresponding to your method of choice and follow the steps.

Console

To submit a model customization job in the console, carry out the following steps.

1. In the Amazon Bedrock console, choose **Custom models** under **Foundation models** from the left navigation pane.

2. In the **Models** tab, choose **Customize model** and then **Create Fine-tuning job** or **Create Continued Pre-training job**, depending on the type of model you want to train.
3. In the **Model details** section, do the following.

    a. Choose the model that you want to customize with your own data and give your resulting model a name.

    b. (Optional) By default, Amazon Bedrock encrypts your model with a key owned and managed by AWS. To use a custom KMS key, select **Model encryption** and choose a key.

    c. (Optional) To associate tags with the custom model, expand the **Tags** section and select **Add new tag**.
4. In the **Job configuration** section, enter a name for the job and optionally add any tags to associate with the job.
5. (Optional) To use a virtual private cloud (VPC) to protect your training data and customization job, select a VPC that contains the input data and output data Amazon S3 locations, its subnets, and security groups in the **VPC settings** section.

    > **Note**
    >
    > If you include a VPC configuration, the console cannot create a new service role for the job. Create a custom service role and add permissions similar to the example described in Attach VPC permissions to a model customization role.
6. In the **Input data** section, select the S3 location of the training dataset file and, if applicable, the validation dataset file.
7. In the **Hyperparameters** section, input values for hyperparameters to use in training.
8. In the **Output data** section, enter the Amazon S3 location where Amazon Bedrock should save the output of the job. Amazon Bedrock stores the training loss metrics and validation loss metrics for each epoch in separate files in the location that you specify.
9. In the **Service access** section, select one of the following:

    - **Use an existing service role** – Select a service role from the drop-down list. For more information on setting up a custom role with the appropriate permissions, see Create a service role for model customization.
    - **Create and use a new service role** – Enter a name for the service role.
10. Choose **Fine-tune model** or **Create Continued Pre-training job** to begin the job.

API

Request

Send a CreateModelCustomizationJob (see link for request and response formats and field details) request with an Amazon Bedrock control plane endpoint to submit a model customization job. Minimally, you must provide the following fields.

- `roleArn` – The ARN of the service role with permissions to customize models. Amazon Bedrock can automatically create a role with the appropriate permissions if you use the console, or you can create a custom role by following the steps at Create a service role for model customization.

> **Note**
>
> If you include a `vpcConfig` field, make sure that the role has the proper permissions to access the VPC. For an example, see Attach VPC permissions to a model customization role.

- `baseModelIdentifier` – The model ID or ARN of the foundation model to customize.
- `customModelName` – The name to give the newly customized model.
- `jobName` – The name to give the training job.
- `hyperParameters` – Hyperparameters that affect the model customization process.
- `trainingDataConfig` – An object containing the Amazon S3 URI of the training dataset. Depending on the customization method and model, you can also include a `validationDataConfig`. For more information about preparing the datasets, see Prepare the datasets.
- `outputDataConfig` – An object containing the Amazon S3 URI to write the output data to.

If you don't specify the `customizationType`, the model customization method defaults to `FINE_TUNING`.

To prevent the request from completing more than once, include a `clientRequestToken`.

You can include the following optional fields for extra configurations.

- `jobTags` and/or `customModelTags` – Associate tags with the customization job or resulting custom model.

- `customModelKmsKeyId` – Include a custom KMS key to encrypt your custom model.
- `vpcConfig` – Include the configuration for a virtual private cloud (VPC) to protect your training data and customization job.

Response

The response returns a `jobArn` that you can use to monitor or stop the job.

See code examples

Manage a model customization job

Once you start a model customization job, you can track its progress or stop it. If you do so through the API, you will need the `jobArn`. You can find it in one of the following ways:

1. In the Amazon Bedrock console
   1. Select **Custom models** under **Foundation models** from the left navigation pane.
   2. Choose the job from the **Training jobs** table to see details, including the ARN of the job.
2. Look in the `jobArn` field in the response returned from the CreateModelCustomizationJob call that created the job or from a CreateModelCustomizationJob call.

Monitor a model customization job

After you begin a job, you can monitor its progress in the console or API. Select the tab corresponding to your method of choice and follow the steps.

Console

To monitor the status of your fine-tuning jobs

1. In the Amazon Bedrock console, choose **Custom models** under **Foundation models** from the left navigation pane.
2. Select the **Training jobs** tab to display the fine-tuning jobs that you have initiated. Look at the **Status** column to monitor the progress of the job.
3. Select a job to view the details you input for training.

API

To list information about all your model customization jobs, send a CreateModelCustomizationJob request with an Amazon Bedrock control plane endpoint. Refer to CreateModelCustomizationJob for filters that you can use.

To monitor the status of a model customization job, send a GetModelCustomizationJob request with an Amazon Bedrock control plane endpoint with the `jobArn` of the job.

To list all the tags for a model customization job, send a ListTagsForResource request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint and include the Amazon Resource Name (ARN) of the job.

See code examples

Stop a model customization job

You can stop an Amazon Bedrock model customization job while it's in progress. Select the tab corresponding to your method of choice and follow the steps.

> **Warning**
>
> You can't resume a stopped job. Amazon Bedrock charges for the tokens that it used to train the model before you stopped the job. Amazon Bedrock doesn't create an intermediate custom model for a stopped job.

Console

To stop a model customization job

1. In the Amazon Bedrock console, choose **Custom models** under **Foundation models** from the left navigation pane.
2. In the **Training Jobs** tab, choose the radio button next to the job to stop or select the job to stop to navigate to the details page.
3. Select the **Stop job** button. You can only stop a job if its status is `Training`.
4. A modal appears to warn you that you can't resume the training job if you stop it. Select **Stop job** to confirm.

API

To stop a model customization job, send a CreateModelCustomizationJob (see link for request and response formats and field details) request with a Amazon Bedrock control plane endpoint, using the `jobArn` of the job.

You can only stop a job if its status is `IN_PROGRESS`. Check the `status` with a GetModelCustomizationJob request. The system marks the job for termination and sets the state to `STOPPING`. Once the job is stopped, the state becomes `STOPPED`.

See code examples

Analyze the results of a model customization job

After a model customization job completes, you can analyze the results of the training process by looking at the files in the output S3 folder that you specified when you submitted the job or view details about the model. Amazon Bedrock stores your customized models in AWS-managed storage scoped to your account.

You can also evaluate your model by running a model evaluation job. For more information, see Model evaluation.

The S3 output for a model customization job contains the following output files in your S3 folder. The validation artifacts only appear if you included a validation dataset.

```
- model-customization-job-training-job-id/
    - training_artifacts/
        - step_wise_training_metrics.csv
    - validation_artifacts/
        - post_fine_tuning_validation/
            - validation_metrics.csv
```

Use the `step_wise_training_metrics.csv` and the `validation_metrics.csv` files to analyze the model customization job and to help you adjust the model as necessary.

The columns in the `step_wise_training_metrics.csv` file are as follows.

- step_number – The step in the training process. Starts from 0.
- epoch_number – The epoch in the training process.

- training_loss – Indicates how well the model fits the training data. A lower value indicates a better fit.
- perplexity – Indicates how well the model can predict a sequence of tokens. A lower value indicates better predictive ability.

The columns in the `validation_metrics.csv` file are the same as the training file, except that `validation_loss` (how well the model fits the validation data) appears in place of `training_loss`.

You can find the output files by opening up the https://console.aws.amazon.com/s3 directly or by finding the link to the output folder within your model details. Select the tab corresponding to your method of choice and follow the steps.

Console

1. In the Amazon Bedrock console, choose **Custom models** under **Foundation models** from the left navigation pane.
2. In the **Models** tab, select a model to view its details. The **Job name** can be found in the **Model details** section.
3. To view the output S3 files, select the **S3 location** in the **Output data** section.
4. Find the training and validation metrics files in the folder whose name matches the **Job name** for the model.

API

To list information about all your custom models, send a ListCustomModels (see link for request and response formats and field details) request with an Amazon Bedrock control plane endpoint. Refer to ListCustomModels for filters that you can use.

To list all the tags for a custom model, send a ListTagsForResource request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint and include the Amazon Resource Name (ARN) of the custom model.

To monitor the status of a model customization job, send a GetCustomModel (see link for request and response formats and field details) request with an Amazon Bedrock control plane endpoint with the `modelIdentifier`, which is either of the following.

- The name that you gave the model.
- The ARN of the model.

You can see `trainingMetrics` and `validationMetrics` for a model customization job in either the GetModelCustomizationJob or GetCustomModel response.

To download the training and validation metrics files, follow the steps at Downloading objects. Use the S3 URI you provided in the `outputDataConfig`.

See code examples

Import a model with Custom Model Import

Custom Model Import is in preview release for Amazon Bedrock and is subject to change.

You can create a custom model in Amazon Bedrock by using the Custom Model Import feature to import Foundation Models that you have customized in other environments, such as Amazon SageMaker. For example, you might have a model that you have created in Amazon SageMaker that has propriety model weights. You can now import that model into Amazon Bedrock and then leverage Amazon Bedrock featues to make inference calls to the model.

You can use a model that you import with on demand or provisioned throughput. Use the InvokeModel or InvokeModelWithResponseStream operations to make inference calls to the model. For more information, see Use the API to invoke a model with a single prompt.

Note

For the preview release, Custom Model Import is available in the US East (N. Virginia) AWS Region only. You can't use Custom Model Import with the following Amazon Bedrock features.

- Agents for Amazon Bedrock
- Knowledge bases for Amazon Bedrock
- Guardrails for Amazon Bedrock
- Batch inference
- AWS CloudFormation

Before you can use Custom Model Import, you must first request a quota increase for the `Imported models per account` quota. For more information, see Requesting a quota increase.

With Custom Model Import you can create a custom model that supports the following patterns.

- **Fine-tuned or Continued Pre-training model** — You can customize the model weights using proprietary data, but retain the configuration of the base model.
- **Adaptation** You can customize the model to your domain for use cases where the model doesn't generalize well. Domain adaptation modifies a model to generalize for a target domain and deal with discrepancies across domains, such as a financial industry wanting to create a model which generalizes well on pricing. Another example is language adaptation. For example you could customize a model to generate responses in Portuguese or Tamil. Most often, this involves changes to the vocabulary of the model that you are using.
- **Pretrained from scratch** — In addition to customizing the weights and vocabulary of the model, you can also change model configuration parameters such as the number of attention heads, hidden layers, or context length. You can reduce precision by using post training quantization or by creating a merged model from base and adapter weights.

Topics

Supported architectures

The model you import must be in one of the following architectures.

- **Mistral** — A decoder-only Transformer based architecture with Sliding Window Attention (SWA) and options for Grouped Query Attention (GQA). For more information, see Mistral in the Hugging Face documentation.
- **Flan** — An enhanced version of the T5 architecture, an encoder-decoder based transformer model. For more information, see Flan T5 in the Hugging Face documentation.

- **Llama 2 and Llama3** — An improved version of Llama with Grouped Query Attention (GQA). For more information, see Llama 2 and Llama 3 in the Hugging Face documentation.

Import source

You import a model into Amazon Bedrock by creating a model import job in the Amazon Bedrock console. In the job you specify the Amazon S3 URI for the source of the model files. Alternatively, if you created the model in Amazon SageMaker, you can specify the SageMaker model. During model training, the import job automatically detects your model's architecture.

If you import from an Amazon S3 bucket, you need to supply the model files in the Hugging Face weights format. You can create the files by using the Hugging Face transformer library. To create model files for a Llama model, see convert_llama_weights_to_hf.py. To create the files for a Mistral AI model, see convert_mistral_weights_to_hf.py.

To import the model from Amazon S3, you minimally need the following files that the Hugging Face transformer library creates.

- **.safetensor** — the model weights in *Safesensor* format. Safetensors is a format created by Hugging Face that stores a model weights as tensors. You must store the tensors for your model in a file with the extension `.safetensors`. For more information, see Safetensors. For information about converting model weights to Safetensor format, see Convert weights to safetensors.

> **Note**
>
> Currently Amazon Bedrock only supports model weights with FP32 and FP16 precision. Amazon Bedrock will reject model weights if you supply them with any other precision.

- **config.json** — For examples, see LlamaConfig and MistralConfig.
- **tokenizer_config.json** — For an example, see LlamaTokenizer.
- **tokenizer.json**
- **tokenizer.model**

Importing a model

The following procedure shows you how to create a custom model by importing a model that you have already customized. The model import job can take several minutes. During the job, Amazon Bedrock validates that the model that uses a compatible the model architecture.

To submit a model import job, carry out the following steps.

1. Request a quota increase for the `Imported models per account` quota. For more information, see Requesting a quota increase.
2. If you are importing your model files from Amazon S3, convert the model to the Hugging Face format.
   a. If your model is a Mistral AI model, use convert_mistral_weights_to_hf.py.
   b. If your model is a Llama model, see convert_llama_weights_to_hf.py.
   c. Upload the model files to an Amazon S3 bucket in your AWS account. For more information, see Upload an object to your bucket.
3. In the Amazon Bedrock console, choose **Custom models** under **Foundation models** from the left navigation pane.
4. In the Amazon Bedrock console, choose **Imported models** under **Foundation models** from the left navigation pane.
5. Choose the **Models** tab.
6. Choose **Import model**.
7. In the **Imported** tab, choose **Import model** to open the **Import model** page.
8. In the **Model details** section, do the following:
   a. In **Model name** enter a name for the model.
   b. (Optional) To associate tags with the model, expand the **Tags** section and select **Add new tag**.
9. In the **Import job name** section, do the following:
   a. In **Job name** enter a name for the model import job.
   b. (Optional) To associate tags with the custom model, expand the **Tags** section and select **Add new tag**.
10. In **Model import settings**, do one of the following.

- If you are importing your model files from an Amazon S3 bucket, choose **Amazon S3 bucket** and enter the Amazon S3 location in **S3 location**. Optionally, you can choose **Browse S3** to choose the file location.
- If you are importing your model from Amazon SageMaker, choose **Amazon SageMaker model** and then choose the SageMaker model that you want to import in **SageMaker models**.

11. In the **Service access** section, select one of the following:

    - **Create and use a new service role –** Enter a name for the service role.
    - **Use an existing service role –** Select a service role from the drop-down list. To see the permissions that your existing service role needs, choose **View permission details**.

      For more information on setting up a service role with the appropriate permissions, see Create a service role for model import.

12. Choose **Import**.
13. On the **Custom models** page, choose **Imported**.
14. In the **Jobs** section, check the status of the import job. The model name you chose identifies the model import job. The job is complete if the value of **Status** for the model is **Complete**.
15. Get the model ID for your model by doing the following.

    a. On the **Imported models** page, choose the **Models** tab.

    b. Copy the ARN for the model that you want to use from the **ARN** column.

16. Use your model for inference calls. For more information, see Use the API to invoke a model with a single prompt. You can use the model with on demand or provisioned throughput. To use provisioned throughput, follow the instructions at Use your model.

    You can also use your model in the Amazon Bedrock text playground.

Use a custom model

Before you can use a customized model, you need to purchase Provisioned Throughput for it. For more information about Provisioned Throughput, see Provisioned Throughput for Amazon Bedrock. You can then use the resulting provisioned model for inference. Select the tab corresponding to your method of choice and follow the steps.

Console

To purchase Provisioned Throughput for a custom model.

1. In the Amazon Bedrock console, choose **Custom models** under **Foundation models** from the left navigation pane.
2. In the **Models** tab, choose the radio button next to the model for which you want to buy Provisioned Throughput or select the model name to navigate to the details page.
3. Select **Purchase Provisioned Throughput**.
4. For more details, follow the steps at Purchase a Provisioned Throughput for a Amazon Bedrock model.
5. After purchasing Provisioned Throughput for your custom model, follow the steps at Run inference using a Provisioned Throughput.

When you carry out any operation that supports usage of custom models, you will see your custom model as an option in the model selection menu.

API

To purchase Provisioned Throughput for a custom model, follow the steps at Purchase a Provisioned Throughput for a Amazon Bedrock model to send a CreateProvisionedModelThroughput (see link for request and response formats and field details) request with a Amazon Bedrock control plane endpoint. Use the name or ARN of your custom model as the `modelId`. The response returns a `provisionedModelArn` that you can use as the `modelId` when making an InvokeModel or InvokeModelWithResponseStream request.

See code examples

Code samples for model customization

The following code samples show how to prepare a basic dataset, set up permissions, create a custom model, view the output files, purchase throughput for the model, and run inference on the model. You can modify these code snippets to your specific use-case.

1. Prepare the training dataset.

   a. Create a training dataset file containing the following one line and name it *`train.jsonl`*.

```
{"prompt": "what is AWS", "completion": "it's Amazon Web Services"}
```

b. Create an S3 bucket for your training data and another one for your output data (the names must be unique).

c. Upload *train.jsonl* into the training data bucket.

2. Create a policy to access your training and attach it to an IAM role with a Amazon Bedrock trust relationship. Select the tab corresponding to your method of choice and follow the steps.

Console

1. Create the S3 policy.

   a. Navigate to the IAM console at https://console.aws.amazon.com/iam and choose **Policies** from the left navigation pane.

   b. Select **Create policy** and then choose **JSON** to open the **Policy editor**.

   c. Paste the following policy, replacing *${training-bucket}* and *${output-bucket}* with your bucket names, and then select **Next**.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::${training-bucket}",
                "arn:aws:s3:::${training-bucket}/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
```

```
                "arn:aws:s3:::${output-bucket}",
                "arn:aws:s3:::${output-bucket}/*"
            ]
        }
    ]
}
```

d. Name the policy *MyFineTuningDataAccess* and select **Create policy**.

2. Create an IAM role and attach the policy.

   a. From the left navigation pane, choose **Roles** and then select **Create role**.

   b. Select **Custom trust policy**, paste the following policy, and select **Next**.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

   c. Search for the *MyFineTuningDataAccess* policy you created, select the checkbox, and choose **Next**.

   d. Name the role *MyCustomizationRole* and select *Create role*.

CLI

1. Create a file called *BedrockTrust.json* and paste the following policy into it.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
```

```
            },
            "Action": "sts:AssumeRole"
        }
    ]
}
```

2. Create another file called *MyFineTuningDataAccess.json* and paste the following policy into it, replacing *${training-bucket}* and *${output-bucket}* with your bucket names.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::${training-bucket}",
                "arn:aws:s3:::${training-bucket}/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::${training-bucket}",
                "arn:aws:s3:::${training-bucket}/*"
            ]
        }
    ]
}
```

3. In a terminal, navigate to the folder containing the policies you created.

4. Make a CreateRole request to create an IAM role called *MyCustomizationRole* and attach the *BedrockTrust.json* trust policy that you created.

```
aws iam create-role \
    --role-name MyCustomizationRole \
    --assume-role-policy-document file://BedrockTrust.json
```

5. Make a [CreatePolicy](#) request to create the S3 data access policy with the *MyFineTuningDataAccess.json* file you created. The response returns an `Arn` for the policy.

```
aws iam create-policy \
    --policy-name MyFineTuningDataAccess \
    --policy-document file://myFineTuningDataAccess.json
```

6. Make an [AttachRolePolicy](#) request to attach the S3 data access policy to your role, replacing the `policy-arn` with the ARN in the response from the previous step:

```
aws iam attach-role-policy \
    --role-name MyCustomizationRole \
    --policy-arn ${policy-arn}
```

Python

1. Run the following code to make a [CreateRole](#) request to create an IAM role called *MyCustomizationRole* and to make a [CreatePolicy](#) request to create an S3 data access policy called *MyFineTuningDataAccess*. For the S3 data access policy, replace *${training-bucket}* and *${output-bucket}* with your S3 bucket names.

```
import boto3
import json

iam = boto3.client("iam")

iam.create_role(
    RoleName="MyCustomizationRole",
    AssumeRolePolicyDocument=json.dumps({
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Principal": {
```

```
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole"
        }
    ]
})
)

iam.create_policy(
    PolicyName="MyFineTuningDataAccess",
    PolicyDocument=json.dumps({
        "Version": "2012-10-17",
        "Statement": [
            {
                "Effect": "Allow",
                "Action": [
                    "s3:GetObject",
                    "s3:ListBucket"
                ],
                "Resource": [
                    "arn:aws:s3:::${training-bucket}",
                    "arn:aws:s3:::${training-bucket}/*"
                ]
            },
            {
                "Effect": "Allow",
                "Action": [
                    "s3:GetObject",
                    "s3:PutObject",
                    "s3:ListBucket"
                ],
                "Resource": [
                    "arn:aws:s3:::${output-bucket}",
                    "arn:aws:s3:::${output-bucket}/*"
                ]
            }
        ]
    })
)
```

2. An `Arn` is returned in the response. Run the following code snippet to make an AttachRolePolicy request, replacing *`${policy-arn}`* with the returned `Arn`.

```
iam.attach_role_policy(
    RoleName="MyCustomizationRole",
    PolicyArn="${policy-arn}"
)
```

3. Select a language to see code samples to call the model customization API operations.

CLI

First, create a text file named *FineTuningData.json*. Copy the JSON code from below into the text file, replacing *${training-bucket}* and *${output-bucket}* with your S3 bucket names.

```
{
    "trainingDataConfig": {
        "s3Uri": "s3://${training-bucket}/train.jsonl"
    },
    "outputDataConfig": {
        "s3Uri": "s3://${output-bucket}"
    }
}
```

To submit a model customization job, navigate to the folder containing *FineTuningData.json* in a terminal and run the following command in the command line, replacing *${your-customization-role-arn}* with the model customization role that you set up.

```
aws bedrock create-model-customization-job \
    --customization-type FINE_TUNING \
    --base-model-identifier arn:aws:bedrock:us-east-1::foundation-model/
amazon.titan-text-express-v1 \
    --role-arn ${your-customization-role-arn} \
    --job-name MyFineTuningJob \
    --custom-model-name MyCustomModel \
    --hyper-parameters
 epochCount=1,batchSize=1,learningRate=.0005,learningRateWarmupSteps=0 \
    --cli-input-json file://FineTuningData.json
```

The response returns a *jobArn*. Allow the job some time to complete. You can check its status with the following command.

```
aws bedrock get-model-customization-job \
    --job-identifier "jobArn"
```

When the `status` is `COMPLETE`, you can see the `trainingMetrics` in the response. You can download the artifacts to the current folder by running the following command, replacing *aet.et-bucket* with your output bucket name and *jobId* with the ID of the customization job (the sequence following the last slash in the `jobArn`).

```
aws s3 cp s3://${output-bucket}/model-customization-job-jobId . --recursive
```

Purchase a no-commitment Provisioned Throughput for your custom model with the following command.

Note

You will be charged hourly for this purchase. Use the console to see price estimates for different options.

```
aws bedrock create-provisioned-model-throughput \
    --model-id MyCustomModel \
    --provisioned-model-name MyProvisionedCustomModel \
    --model-units 1
```

The response returns a `provisionedModelArn`. Allow the Provisioned Throughput some time to be created. To check its status, provide the name or ARN of the provisioned model as the `provisioned-model-id` in the following command.

```
aws bedrock get-provisioned-model-throughput \
    --provisioned-model-id ${provisioned-model-arn}
```

When the `status` is `InService`, you can run inference with your custom model with the following command. You must provide the ARN of the provisioned model as the `model-id`. The output is written to a file named *output.txt* in your current folder.

```
aws bedrock-runtime invoke-model \
```

```
    --model-id ${provisioned-model-arn} \
    --body '{"inputText": "What is AWS?", "textGenerationConfig": {"temperature": 
 0.5}}' \
    --cli-binary-format raw-in-base64-out \
    output.txt
```

Python

Run the following code snippet to submit a fine-tuning job. Replace *${your-customization-role-arn}* with the ARN of the *MyCustomizationRole* that you set up and replace *${training-bucket}* and *${output-bucket}* with your S3 bucket names.

```
import boto3
import json

bedrock = boto3.client(service_name='bedrock')

# Set parameters
customizationType = "FINE_TUNING"
baseModelIdentifier = "arn:aws:bedrock:us-east-1::foundation-model/amazon.titan-
text-express-v1"
roleArn = "${your-customization-role-arn}"
jobName = "MyFineTuningJob"
customModelName = "MyCustomModel"
hyperParameters = {
        "epochCount": "1",
        "batchSize": "1",
        "learningRate": ".0005",
        "learningRateWarmupSteps": "0"
    }
trainingDataConfig = {"s3Uri": "s3://${training-bucket}/myInputData/train.jsonl"}
outputDataConfig = {"s3Uri": "s3://${output-bucket}/myOutputData"}

# Create job
response_ft = bedrock.create_model_customization_job(
    jobName=jobName,
    customModelName=customModelName,
    roleArn=roleArn,
    baseModelIdentifier=baseModelIdentifier,
    hyperParameters=hyperParameters,
    trainingDataConfig=trainingDataConfig,
    outputDataConfig=outputDataConfig
)
```

```
jobArn = response_ft.get('jobArn')
```

The response returns a *jobArn*. Allow the job some time to complete. You can check its status with the following command.

```
bedrock.get_model_customization_job(jobIdentifier=jobArn).get('status')
```

When the `status` is `COMPLETE`, you can see the `trainingMetrics` in the [GetModelCustomizationJob](#) response. You can also follow the steps at [Downloading objects](#) to download the metrics.

Purchase a no-commitment Provisioned Throughput for your custom model with the following command.

```
response_pt = bedrock.create_provisioned_model_throughput(
    modelId="MyCustomModel",
    provisionedModelName="MyProvisionedCustomModel"
    modelUnits="1"
)

provisionedModelArn = response_pt.get('provisionedModelArn')
```

The response returns a `provisionedModelArn`. Allow the Provisioned Throughput some time to be created. To check its status, provide the name or ARN of the provisioned model as the `provisionedModelId` in the following command.

```
bedrock.get_provisioned_model_throughput(provisionedModelId=provisionedModelArn)
```

When the `status` is `InService`, you can run inference with your custom model with the following command. You must provide the ARN of the provisioned model as the `modelId`.

```
import json
import logging
import boto3

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by the model"
```

```
    def __init__(self, message):
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_text(model_id, body):
    """
    Generate text using your provisioned custom model.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (json): The response from the model.
    """

    logger.info(
        "Generating text with your provisioned custom model %s", model_id)

    brt = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = brt.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())

    finish_reason = response_body.get("error")

    if finish_reason is not None:
        raise ImageError(f"Text generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated text with provisioned custom model %s", model_id)

    return response_body


def main():
```

```
    """
    Entrypoint for example.
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        model_id = provisionedModelArn

        body = json.dumps({
            "inputText": "what is AWS?"
        })

        response_body = generate_text(model_id, body)
        print(f"Input token count: {response_body['inputTextTokenCount']}")

        for result in response_body['results']:
            print(f"Token count: {result['tokenCount']}")
            print(f"Output text: {result['outputText']}")
            print(f"Completion reason: {result['completionReason']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating text with your provisioned custom model
 {model_id}.")


if __name__ == "__main__":
    main()
```

Guidelines for model customization

The ideal parameters for customizing a model depend on the dataset and the task for which the model is intended. You should experiment with values to determine which parameters work best for your specific case. To help, evaluate your model by running a model evaluation job. For more information, see Model evaluation.

This topic provides guidelines and recommended values as a baseline for customization of the Amazon Titan Text G1 - Express model. For other models, check the provider's documentation.

Use the training and validation metrics from the output files generated when you submit a fine-tuning job to help you adjust your parameters. Find these files in the Amazon S3 bucket to which you wrote the output, or use the GetCustomModel operation.

Amazon Titan Text G1 - Express

The following guidelines are for the Titan Text Express text-to-text model model. For information about the hyperparameters that you can set, see Amazon Titan text model customization hyperparameters.

Impact on other tasks types

In general, the larger the training dataset, the better the performance for a specific task. However, training for a specific task might make the model perform worse on different tasks, especially if you use a lot of examples. For example, if the training dataset for a summarization task contains 100,000 samples, the model might perform worse on a classification task).

Model size

In general, the larger the model, the better the task performs given limited training data.

If you are using the model for a *classification* task, you might see relatively small gains for few-shot fine-tuning (less than 100 samples), especially if the number of classes is relatively small (less than 100).

Epochs

We recommend using the following metrics to determine the number of epochs to set:

1. **Validation output accuracy** – Set the number of epochs to one that yields a high accuracy.

2. **Training and validation loss –** Determine the number of epochs after which the training and validation loss becomes stable. This corresponds to when the model converges. Find the training loss values in the `step_wise_training_metrics.csv` and `validation_metrics.csv` files.

Batch size

When you change the batch size, we recommend that you change the learning rate using the following formula:

```
newLearningRate = oldLearningRate x newBatchSize / oldBatchSize
```

Learning rate

In general, use smaller learning rates for larger models. We recommend using a learning rate in the range of 1.00E-06 to 1.00E-05.

The following table shows recommended learning rate values for fine-tuning:

Task	Minimum learning rate	Default learning rate	Max learning rate
Summarization	1.00E-06	3.00E-06	5.00E-05
Classification	5.00E-06	5.00E-05	5.00E-05
Question-answer	5.00E-06	5.00E-06	5.00E-05

Learning warmup steps

We recommend the default value of 5.

Troubleshooting

This section summarizes errors that you might encounter and what to check if you do.

Permissions issues

If you encounter an issue with permissions to access an Amazon S3 bucket, check that the following are true:

1. If the Amazon S3 bucket uses a CM-KMS key for Server Side encryption, ensure that the IAM role passed to Amazon Bedrock has `kms:Decrypt` permissions for the AWS KMS key. For example, see [Allow a user to enccrypt and decrypt with any AWS KMS key in a specific AWS account](#).
2. The Amazon S3 bucket is in the same region as the Amazon Bedrock model customization job.
3. The IAM role trust policy includes the service SP (`bedrock.amazonaws.com`).

The following messages indicate issues with permissions to access training or validation data in an Amazon S3 bucket:

```
Could not validate GetObject permissions to access Amazon S3 bucket: training-data-bucket at key train.jsonl
Could not validate GetObject permissions to access Amazon S3 bucket: validation-data-bucket at key validation.jsonl
```

If you encounter one of the above errors, check that the IAM role passed to the service has `s3:GetObject` and `s3:ListBucket` permissions for the training and validation dataset Amazon S3 URIs.

The following message indicates issues with permissions to write the output data in an Amazon S3 bucket:

```
Amazon S3 perms missing (PutObject): Could not validate PutObject permissions to access
 S3 bucket: bedrock-output-bucket at key output/.write_access_check_file.tmp
```

If you encounter the above error, check that the IAM role passed to the service has `s3:PutObject` permissions for the output data Amazon S3 URI.

Data issues

The following errors are related to issues with the training, validation, or output data files:

Invalid file format

```
Unable to parse Amazon S3 file: fileName.jsonl. Data files must conform to JSONL
 format.
```

If you encounter the above error, check that the following are true:

1. Each line is in JSON.
2. Each JSON has two keys, an *input* and an *output*, and each key is a string. For example:

```
{
    "input": "this is my input",
    "output": "this is my output"
}
```

3. There are no additional new lines or empty lines.

Character quota exceeded

```
Input size exceeded in file fileName.jsonl for record starting with...
```

If you encounter an error beginning with the text above, ensure that the number of characters conforms to the character quota in Model customization quotas.

Token count exceeded

```
Maximum input token count 4097 exceeds limit of 4096
Maximum output token count 4097 exceeds limit of 4096
Max sum of input and output token length 4097 exceeds total limit of 4096
```

If you encounter an error similar to the preceeding example, make sure that the number of tokens conforms to the token quota in Model customization quotas.

Internal error

```
Encountered an unexpected error when processing the request, please try again
```

If you encounter the above error, there might be an issue with the service. Try the job again. If the issue persists, contact AWS Support.

Provisioned Throughput for Amazon Bedrock

Throughput refers to the number and rate of inputs and outputs that a model processes and returns. You can purchase **Provisioned Throughput** to provision a higher level of throughput for a model at a fixed cost. If you customized a model, you must purchase Provisioned Throughput to be able to use it.

You're billed hourly for a Provisioned Throughput that you purchase. For detailed information about pricing, see Amazon Bedrock Pricing. The price per hour depends on the following factors:

1. The model that you choose (for custom models, pricing is the same as the base model that it was customized from).
2. The number of Model Units (MUs) that you specify for the Provisioned Throughput. An MU delivers a specific throughput level for the specified model. The throughput level of an MU specifies the following:
   - The number of input tokens that an MU can process across all requests within a span of one minute.
   - The number of output tokens that an MU can generate across all requests within a span of one minute.

   > **Note**
   >
   > For more information about what an MU specifies, contact your AWS account manager.

3. The duration of time you commit to keeping the Provisioned Throughput. The longer the commitment duration, the more discounted the hourly price becomes. You can choose between the following levels of commitment:
   - No commitment – Billing ends when you delete the Provisioned Throughput.
   - 1 month – Billing ends after a month. You can't delete the Provisioned Throughput until the commitment term is over.
   - 6 months – Billing ends after six months. You can't delete the Provisioned Throughput until the commitment term is over.

The following steps outline the process of setting up and using Provisioned Throughput.

1. Determine the number of MUs you wish to purchase for a Provisioned Throughput and the amount of time for which you want to commit to using the Provisioned Throughput.
2. Purchase Provisioned Throughput for a base or custom model.
3. After the provisioned model is created, you can use it to run model inference.

Topics

Supported regions and models for Provisioned Throughput

Provisioned Throughput is supported in the following regions:

Region		
US East (N. Virginia)		
US West (Oregon)		
AWS GovCloud (US-West) (only for custom models with no commitment)		

If you purchase Provisioned Throughput through the Amazon Bedrock API, you must specify a contextual variant of Amazon Bedrock FMs for the model ID. The following table shows the models for which you can purchase Provisioned Throughput, whether you can purchase without commitment for the base model, and the model ID to use when purchasing Provisioned Throughput.

Model name	No-commitment purchase supported for base model	Model ID for Provisioned Throughput
Amazon Titan Text G1 - Express	Yes	amazon.titan-text-express-v1:0:8k
Amazon Titan Text G1 - Lite	Yes	amazon.titan-text-lite-v1:0:4k
Amazon Titan Embeddings G1 - Text	Yes	amazon.titan-embed-text-v1:2:8k
Amazon Titan Multimodal Embeddings G1	Yes	amazon.titan-embed-image-v1:0
Amazon Titan Image Generator G1	No	amazon.titan-image-generator-v1:0
Anthropic Claude v2 18K	Yes	anthropic.claude-v2:0:18k
Anthropic Claude v2 100K	Yes	anthropic.claude-v2:0:100k
Anthropic Claude v2.1 18K	Yes	anthropic.claude-v2:1:18k
Anthropic Claude v2.1 200K	Yes	anthropic.claude-v2:1:200k
Anthropic Claude 3 Sonnet 28K	Yes	anthropic.claude-3-sonnet-20240229-v1:0:28k
Anthropic Claude 3 Sonnet 200K	Yes	anthropic.claude-3-sonnet-20240229-v1:0:200k
Anthropic Claude 3 Haiku 48K	Yes	anthropic.claude-3-haiku-20240307-v1:0:48k
Anthropic Claude 3 Haiku 200K	Yes	anthropic.claude-3-haiku-20240307-v1:0:200k
Anthropic Claude Instant v1 100K	Yes	anthropic.claude-instant-v1:2:100k

Model name	No-commitment purchase supported for base model	Model ID for Provisioned Throughput
Cohere Command	Yes	cohere.command-text-v14:7:4k
Cohere Command Light	Yes	cohere.command-light-text-v14:7:4k
Cohere Embed English	Yes	cohere.embed-english-v3:0:512
Cohere Embed Multilingual	Yes	cohere.embed-multilingual-v3:0:512
Stable Diffusion XL 1.0	No	stability.stable-diffusion-xl-v1:0
Meta Llama 2 Chat 13B	No	meta.llama2-13b-chat-v1:0:4k
Meta Llama 2 13B	No	(see note below)
Meta Llama 2 70B	No	(see note below)

Note

The Meta Llama 2 (non-chat) models can only be used after being customized and after purchasing Provisioned Throughput for them.

Prerequisites

Before you can purchase and manage Provisioned Throughput, you need to fulfill the following prerequisites:

1. Request access to the model or models that you want to purchase Provisioned Throughput for. After access has been granted, you can purchase Provisioned Throughput for the base model and any models customized from it.
2. Ensure that your IAM role has the necessary permissions to perform actions related to Provisioned Throughput.
3. If you're purchasing Provisioned Throughput for a custom model that's encrypted with a customer-managed AWS KMS key, your IAM role must have permissions to decrypt the key. You can use the template at Create a key policy and attach it to the customer managed key. For minimal permissions, you can use only the `Permissions for custom model users` policy statement.

Purchase a Provisioned Throughput for a Amazon Bedrock model

When you purchase a Provisioned Throughput for a model, you specify the level of commitment for it and the number of model units (MUs) to allot. For MU quotas, see Provisioned Throughput quotas. The number of MUs that you can allot to your Provisioned Throughputs depends on the commitment term for the Provisioned Throughput:

- By default, your account provides you with 2 MUs to distribute between Provisioned Throughputs with no commitment.
- If you're purchasing a Provisioned Throughput with commitment, you must first visit the AWS support center to request MUs for your account to distribute between Provisioned Throughputs with commitment. After your request is granted, you can purchase a Provisioned Throughput with commitment.

> **Note**
>
> After you purchase the Provisioned Throughput, you can only change the associated model if you select a custom model. You can change the associated model to one of the following:
>
> - The base model that it's customized from.
> - Another custom model that's derived from the same base model.

To learn how to purchase Provisioned Throughput for a model, select the tab corresponding to your method of choice and follow the steps.

Console

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Provisioned Throughput** under **Assessment and deployment** from the left navigation pane.
3. In the **Provisioned Throughput** section, choose **Purchase Provisioned Throughput**.
4. For the **Provisioned Throughput details** section, do the following:
    a. In the **Provisioned Throughput name** field, enter a name for the Provisioned Throughput.
    b. Under **Select model**, select a base model provider or a custom model category. Then select the model for which to provision throughput.

        Note

        To see the base models for which you can purchase Provisioned Throughput without commitment, see Supported regions and models for Provisioned Throughput.
        In the AWS GovCloud (US) region, you can only purchase Provisioned Throughput for custom models with no commitment.

    c. (Optional) To associate tags with your Provisioned Throughput, expand the **Tags** section and choose **Add new tag**. For more information, see Tag resources.
5. For the **Commitment term & model units** section, do the following:
    a. In the **Select commitment term** section, select the amount of time for which you want to commit to using the Provisioned Throughput.
    b. In the **Model units** field, enter the desired number of model units (MUs). If you are provisioning a model with commitment, you must first visit the AWS support center to request an increase in the number of MUs that you can purchase.
6. Under **Estimated purchase summary**, review the estimated cost.
7. Choose **Purchase Provisioned Throughput**.

8. Review the note that appears and acknowledge the commitment duration and price by selecting the checkbox. Then choose **Confirm purchase**.
9. The console displays the **Provisioned Throughput** overview page. The **Status** of the Provisioned Throughput in the Provisioned Throughput table becomes **Creating**. When the Provisioned Throughput is finished being created, the **Status** becomes **In service**. If the update fails, the **Status** becomes **Failed**.

API

To purchase a Provisioned Throughput, send a CreateProvisionedModelThroughput request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint.

> **Note**
>
> To see the base models for which you can purchase Provisioned Throughput without commitment, see Supported regions and models for Provisioned Throughput.
> In the AWS GovCloud (US) region, you can only purchase Provisioned Throughput for custom models with no commitment.

The following table briefly describes the parameters and request body (for detailed information and the request structure, see the CreateProvisionedModelThroughput request syntax):

Variable	Required?	Use case
modelId	Yes	To specify the base model ID or ARN for purchasing Provisioned Throughput, or the custom model name or ARN
modelUnits	Yes	To specify the number of model units (MUs) to purchase. To increase the number of MUs that you can purchase, visit the AWS

Variable	Required?	Use case
		support center to request an increase in the number of MUs that you can purchase
provisionedModelName	Yes	To specify a name for the Provisioned Throughput
commitmentDuration	No	To specify the duration for which to commit to the Provisioned Throughput. Omit this field to opt for no-commitment pricing
tags	No	To associate tags with your Provisioned Throughput
clientRequestToken	No	To prevent reduplication of the request

The response returns a `provisionedModelArn` that you can use as a `modelId` in model inference. To check when the Provisioned Throughput is ready for use, send a GetProvisionedModelThroughput request and check that the status is `InService`. If the update fails, its status will be `Failed`, and the GetProvisionedModelThroughput response will contain a `failureMessage`.

See code examples

Manage a Provisioned Throughput

After you purchase a Provisioned Throughput you can view details about it, update it, or delete it.

Topics

- View information about a Provisioned Throughput
- Edit a Provisioned Throughput
- Delete a Provisioned Throughput

View information about a Provisioned Throughput

To learn how to view information about a Provisioned Throughput that you've purchased, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a Provisioned Throughput

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Provisioned Throughput** under **Assessment and deployment** from the left navigation pane.
3. From the **Provisioned Throughput** section, select a Provisioned Throughput.
4. View the details for the Provisioned Throughput in the **Provisioned Throughput overview** section and the tags associated with your Provisioned Throughput in the **Tags** section.

API

To retrieve information about a specific Provisioned Throughput, send a GetProvisionedModelThroughput request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint. Specify either the name of the Provisioned Throughput or its ARN as the `provisionedModelId`.

To list information about all the Provisioned Throughputs in an account, send a ListProvisionedModelThroughputs request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint. To control the number of results that are returned, you can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value.

Field	Short description
	To see the next batch of results, send the `nextToken` value in another request.

For other optional parameters that you can specify to sort and filter the results, see GetProvisionedModelThroughput.

To list all the tags for an agent, send a ListTagsForResource request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint and include the Amazon Resource Name (ARN) of the Provisioned Throughput.

See code examples

Edit a Provisioned Throughput

You can edit the name or tags of an existing Provisioned Throughput.

The following restrictions apply to changing the model that the Provisioned Throughput is associated with:

- You can't change the model for a Provisioned Throughput associated with a base model.
- If the Provisioned Throughput is associated with a custom model, you can change the association to the base model that it's customized from, or to another custom model that was derived from the same base model.

While a Provisioned Throughput is updating, you can run inference using the Provisioned Throughput without disrupting the on-going traffic from your end customers. If you changed the model that the Provisioned Throughput is associated with, you might receive output from the old model until the update is fully deployed.

To learn how to edit a Provisioned Throughput, select the tab corresponding to your method of choice and follow the steps.

Console

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

2. Select **Provisioned Throughput** under **Assessment and deployment** from the left navigation pane.
3. From the **Provisioned Throughput** section, select a Provisioned Throughput.
4. Choose **Edit**. You can edit the following fields:
   - **Provisioned Throughput name** – Change the name of the Provisioned Throughput.
   - **Select model** – If the Provisioned Throughput is associated with a custom model, you can change the associated model.
5. You can edit the tags associated with your Provisioned Throughput in the **Tags** section. For more information, see Tag resources.
6. To save your changes, choose **Save edits**.
7. The console displays the **Provisioned Throughput** overview page. The **Status** of the Provisioned Throughput in the Provisioned Throughput table becomes **Updating**. When the Provisioned Throughput is finished being update, the **Status** becomes **In service**. If the update fails, the **Status** becomes **Failed**.

API

To edit a Provisioned Throughput, send an UpdateProvisionedModelThroughput request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint.

The following table briefly describes the parameters and request body (for detailed information and the request structure, see the UpdateProvisionedModelThroughput request syntax):

Variable	Required?	Use case
provisionedModelId	Yes	To specify the name or ARN of the Provisioned Throughput to update
desiredModelId	No	To specify a new model to associate with the Provision ed Throughput (unavailable for Provisioned Throughpu

Variable	Required?	Use case
		ts associated with base models).
desiredProvisioned ModelName	No	To specify a new name for the Provisioned Throughput

If the action is successful, the response returns an HTTP 200 status response. To check when the Provisioned Throughput is ready for use, send a GetProvisionedModelThroughput request and check that the status is `InService`. You can't update or delete a Provisioned Throughput while its status is `Updating`. If the update fails, its status will be `Failed`, and the GetProvisionedModelThroughput response will contain a `failureMessage`.

To add tags to a Provisioned Throughput, send a TagResource request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint and include the Amazon Resource Name (ARN) of the Provisioned Throughput. The request body contains a `tags` field, which is an object containing a key-value pair that you specify for each tag.

To remove tags from a Provisioned Throughput, send an UntagResource request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint and include the Amazon Resource Name (ARN) of the Provisioned Throughput. The `tagKeys` request parameter is a list containing the keys for the tags that you want to remove.

See code examples

Delete a Provisioned Throughput

To learn how to delete a Provisioned Throughput, select the tab corresponding to your method of choice and follow the steps.

> **Note**
>
> You can't delete a Provisioned Throughput with commitment before the commitment term is complete.

Console

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Provisioned Throughput** under **Assessment and deployment** from the left navigation pane.
3. From the **Provisioned Throughput** section, select a Provisioned Throughput.
4. Choose **Delete**.
5. The console displays a modal form to warn you that deletion is permanent. Choose **Confirm** to proceed.
6. The Provisioned Throughput is immediately deleted.

API

To delete a Provisioned Throughput, send a DeleteProvisionedModelThroughput request (see link for request and response formats and field details) with an Amazon Bedrock control plane endpoint. Specify either the name of the Provisioned Throughput or its ARN as the `provisionedModelId`. If deletion is successful, the response returns an HTTP 200 status code.

See code examples

Run inference using a Provisioned Throughput

After you purchase a Provisioned Throughput, you can use it in model inference to increase your throughput. If you want, you can first test the Provisioned Throughput in a Amazon Bedrock console playground. When you're ready to deploy the Provisioned Throughput, you set up your application to invoke the provisioned model. Select the tab corresponding to your method of choice and follow the steps.

Console

To use a Provisioned Throughput in the Amazon Bedrock console playground

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. From the left navigation pane, select **Chat**, **Text**, or **Image** under **Playgrounds**, depending your use case.

3. Choose **Select model**.
4. In the **1. Category** column, select a provider or custom model category. Then, in the **2. Model** column, select the model that your Provisioned Throughput is associated with.
5. In the **3. Throughput** column, select your Provisioned Throughput.
6. Choose **Apply**.

To learn how to use the Amazon Bedrock playgrounds, see Playgrounds.

API

To run inference using a Provisioned Throughput, send an InvokeModel or InvokeModelWithResponseStream request (see link for request and response formats and field details) with an Amazon Bedrock runtime endpoint. Specify the provisioned model ARN as the `modelId` parameter. To see requirements for the request body for different models, see Inference parameters for foundation models.

See code examples

Code samples for Provisioned Throughput in Amazon Bedrock

The following code examples demonstrate how to create, use, and manage a Provisioned Throughput with the AWS CLI and the Python SDK.

AWS CLI

Create a no-commitment Provisioned Throughput called `MyPT` based off a custom model called `MyCustomModel` that was customized from the Anthropic Claude v2.1 model by running the following command in a terminal.

```
aws bedrock create-provisioned-model-throughput \
   --model-units 1 \
   --provisioned-model-name MyPT \
   --model-id arn:aws:bedrock:us-east-1::custom-model/anthropic.claude-v2:1:200k/
MyCustomModel
```

The response returns a `provisioned-model-arn`. Allow some time for the creation to complete. To check its status, provide the name or ARN of the provisioned model as the `provisioned-model-id` in the following command.

```
aws bedrock get-provisioned-model-throughput \
    --provisioned-model-id MyPT
```

Change the name of the Provisioned Throughput and associate it with a different model customized from Anthropic Claude v2.1.

```
aws bedrock update-provisioned-model-throughput \
    --provisioned-model-id MyPT \
    --desired-provisioned-model-name MyPT2 \
    --desired-model-id arn:aws:bedrock:us-east-1::custom-model/anthropic.claude-
v2:1:200k/MyCustomModel2
```

Run inference with your updated provisioned model with the following command. You must provide the ARN of the provisioned model, returned in the `UpdateProvisionedModelThroughput` response, as the `model-id`. The output is written to a file named *`output.txt`* in your current folder.

```
aws bedrock-runtime invoke-model \
    --model-id ${provisioned-model-arn} \
    --body '{"inputText": "What is AWS?", "textGenerationConfig": {"temperature":
 0.5}}' \
    --cli-binary-format raw-in-base64-out \
    output.txt
```

Delete the Provisioned Throughput using the following command. You'll no longer be charged for the Provisioned Throughput.

```
aws bedrock delete-provisioned-model-throughput
  --provisioned-model-id MyPT2
```

Python (Boto)

Create a no-commitment Provisioned Throughput called `MyPT` based off a custom model called `MyCustomModel` that was customized from the Anthropic Claude v2.1 model by running the following code snippet.

```
import boto3

bedrock = boto3.client(service_name='bedrock')
bedrock.create_provisioned_model_throughput(
```

```
    modelUnits=1,
    provisionedModelName='MyPT',
    modelId='arn:aws:bedrock:us-east-1::custom-model/anthropic.claude-v2:1:200k/
MyCustomModel'
)
```

The response returns a `provisionedModelArn`. Allow some time for the creation to complete. You can check its status with the following code snippet. You can provide either the name of the Provisioned Throughput or the ARN returned from the [CreateProvisionedModelThroughput](#) response as the `provisionedModelId`.

```
bedrock.get_provisioned_model_throughput(provisionedModelId='MyPT')
```

Change the name of the Provisioned Throughput and associate it with a different model customized from Anthropic Claude v2.1. Then send a [GetProvisionedModelThroughput](#) request and save the ARN of the provisioned model to a variable to use for inference.

```
bedrock.update_provisioned_model_throughput(
    provisionedModelId='MyPT',
    desiredProvisionedModelName='MyPT2',
    desiredModelId='arn:aws:bedrock:us-east-1::custom-model/anthropic.claude-
v2:1:200k/MyCustomModel2'
)

arn_MyPT2 =
 bedrock.get_provisioned_model_throughput(provisionedModelId='MyPT2').get('provisionedModelA
```

Run inference with your updated provisioned model with the following command. You must provide the ARN of the provisioned model as the `modelId`.

```
import json
import logging
import boto3

from botocore.exceptions import ClientError


class ImageError(Exception):
    "Custom exception for errors returned by the model"

    def __init__(self, message):
```

```
        self.message = message


logger = logging.getLogger(__name__)
logging.basicConfig(level=logging.INFO)


def generate_text(model_id, body):
    """
    Generate text using your provisioned custom model.
    Args:
        model_id (str): The model ID to use.
        body (str) : The request body to use.
    Returns:
        response (json): The response from the model.
    """

    logger.info(
        "Generating text with your provisioned custom model %s", model_id)

    brt = boto3.client(service_name='bedrock-runtime')

    accept = "application/json"
    content_type = "application/json"

    response = brt.invoke_model(
        body=body, modelId=model_id, accept=accept, contentType=content_type
    )
    response_body = json.loads(response.get("body").read())

    finish_reason = response_body.get("error")

    if finish_reason is not None:
        raise ImageError(f"Text generation error. Error is {finish_reason}")

    logger.info(
        "Successfully generated text with provisioned custom model %s", model_id)

    return response_body


def main():
    """
    Entrypoint for example.
```

```
    """
    try:
        logging.basicConfig(level=logging.INFO,
                            format="%(levelname)s: %(message)s")

        model_id = arn_myPT2

        body = json.dumps({
            "inputText": "what is AWS?"
        })

        response_body = generate_text(model_id, body)
        print(f"Input token count: {response_body['inputTextTokenCount']}")

        for result in response_body['results']:
            print(f"Token count: {result['tokenCount']}")
            print(f"Output text: {result['outputText']}")
            print(f"Completion reason: {result['completionReason']}")

    except ClientError as err:
        message = err.response["Error"]["Message"]
        logger.error("A client error occurred: %s", message)
        print("A client error occured: " +
              format(message))
    except ImageError as err:
        logger.error(err.message)
        print(err.message)

    else:
        print(
            f"Finished generating text with your provisioned custom model
 {model_id}.")


if __name__ == "__main__":
    main()
```

Delete the Provisioned Throughput with the following code snippet. You'll no longer be charged for the Provisioned Throughput.

```
bedrock.delete_provisioned_model_throughput(provisionedModelId='MyPT2')
```

Tag resources

To help you manage your Amazon Bedrock resources, you can assign metadata to each resource as tags. A tag is a label that you assign to an AWS resource. Each tag consists of a key and a value.

Tags enable you to categorize your AWS resources in different ways, for example, by purpose, owner, or application. Tags help you to do the following:

- Identify and organize your AWS resources. Many AWS resources support tagging, so you can assign the same tag to resources in different services to indicate that the resources are the same.
- Allocate costs. You activate tags on the AWS Billing and Cost Management dashboard. AWS uses the tags to categorize your costs and deliver a monthly cost allocation report to you. For more information, see Use cost allocation tags in the *AWS Billing and Cost Management User Guide*.
- Control access to your resources. You can use tags with Amazon Bedrock to create policies to control access to Amazon Bedrock resources. These policies can be attached to an IAM role or user to enable tag-based access control.

The Amazon Bedrock resources that you can tag are:

- Custom models
- Model customization jobs
- Provisioned models
- Batch inference jobs (API only)
- Agents
- Agent aliases
- Knowledge bases
- Model evaluations (console only)

Topics

Use the console

You can add, modify, and remove tags at any time while creating or editing a supported resource.

Use the API

To carry out tagging operations, you need the Amazon Resource Name (ARN) of the resource on which you want to carry out a tagging operation. There are two sets of tagging operations, depending on the resource for which you are adding or managing tags.

1. The following resources use the Amazon Bedrock TagResource, UntagResource, and ListTagsForResource operations.
   - Custom models
   - Model customization jobs
   - Provisioned models
   - Batch inference jobs
2. The following resources use the Agents for Amazon Bedrock TagResource, UntagResource, and ListTagsForResource operations.
   - Agents
   - Agent aliases
   - Knowledge bases

To add tags to a resource, send a Amazon Bedrock TagResource or Agents for Amazon Bedrock TagResource request.

To untag a resource, send an UntagResource or UntagResource request.

To list the tags for a resource, send a ListTagsForResource or ListTagsForResource request.

Select a tab to see code examples in an interface or language.

AWS CLI

Add two tags to an agent. Separate key/value pairs with a space.

```
aws bedrock-agent tag-resource \
    --resource-arn "arn:aws:bedrock:us-east-1:123456789012:agent/AGENT12345" \
```

```
    --tags key=department,value=billing key=facing,value=internal
```

Remove the tags from the agent. Separate keys with a space.

```
aws bedrock-agent untag-resource \
    --resource-arn "arn:aws:bedrock:us-east-1:123456789012:agent/AGENT12345" \
    --tag-keys key=department facing
```

List the tags for the agent.

```
aws bedrock-agent list-tags-for-resource \
    --resource-arn "arn:aws:bedrock:us-east-1:123456789012:agent/AGENT12345"
```

Python (Boto)

Add two tags to an agent.

```
import boto3

bedrock = boto3.client(service_name='bedrock-agent')

tags = [
    {
        'key': 'department',
        'value': 'billing'
    },
    {
        'key': 'facing',
        'value': 'internal'
    }
]

bedrock.tag_resource(resourceArn='arn:aws:bedrock:us-east-1:123456789012:agent/
AGENT12345', tags=tags)
```

Remove the tags from the agent.

```
bedrock.untag_resource(
    resourceArn='arn:aws:bedrock:us-east-1:123456789012:agent/AGENT12345',
    tagKeys=['department', 'facing']
)
```

List the tags for the agent.

```
bedrock.list_tags_for_resource(resourceArn='arn:aws:bedrock:us-
east-1:123456789012:agent/AGENT12345')
```

Best practices and restrictions

For best practices and restrictions on tagging, see Tagging your AWS resources.

Amazon Titan Models

Amazon Titan foundation models (FMs) are a family of FMs pretrained by AWS on large datasets, making them powerful, general-purpose models built to support a variety of use cases. Use them as-is or privately customize them with your own data.

Amazon Titan supports the following models for Amazon Bedrock.

- **Amazon Titan Text**
- **Amazon Titan Embeddings G1 - Text**
- **Amazon Titan Multimodal Embeddings G1**
- **Amazon Titan Image Generator G1**

Topics

Amazon Titan Text models

Amazon Titan text models include Amazon Titan, Amazon Titan Text G1 - Express and Amazon Titan Text G1 - Lite.

Amazon Titan Text G1 - Express

Amazon Titan Text G1 - Express is a large language model for text generation. It is useful for a wide range of advanced, general language tasks such as open-ended text generation and conversational chat, as well as support within Retrieval Augmented Generation (RAG). At launch, the model is optimized for English, with multilingual support for more than 100 additional languages available in preview.

- **Model ID** – `amazon.titan-text-express-v1`
- **Max tokens** – 8K

- **Languages** – English (GA), 100 additional languages (Preview)
- **Supported use cases** – Retrieval augmented generation, open-ended text generation, brainstorming, summarizations, code generation, table creation, data formatting, paraphrasing, chain of thought, rewrite, extraction, QnA, and chat.

Amazon Titan Text G1 - Lite

Amazon Titan Text G1 - Lite is a light weight efficient model, ideal for fine-tuning of English-language tasks, including like summarizations and copy writing, where customers want a smaller, more cost-effective model that is also highly customizable.

- **Model ID** – `amazon.titan-text-lite-v1`
- **Max tokens** – 4K
- **Languages** – English
- **Supported use cases** – Open-ended text generation, brainstorming, summarizations, code generation, table creation, data formatting, paraphrasing, chain of thought, rewrite, extraction, QnA, and chat.

Amazon Titan Text Model Customization

For more information on customizing Amazon Titan text models, see the following pages.

- Prepare the datasets
- Amazon Titan text model customization hyperparameters

Amazon Titan Text Prompt Engineering Guidelines

Amazon Titan text models can be used in a wide variety of applications for different use cases. Amazon Titan Text models have prompt engineering guidelines for the following applications including:

- Chatbot
- Text2SQL
- Function Calling
- RAG (Retrieval Augmented Generation)

For more information on Amazon Titan Text prompt engineering guidelines, see Amazon Titan Text Prompt Engineering Guidelines.

For general prompt engineering guidelines, see Prompt Engineering Guidelines.

AWS AI Service Card - Amazon Titan Text

AI Service Cards provide transparency and document the intended use cases and fairness considerations for our AWS AI services. AI Service Cards provide a single place to find information on the intended use cases, responsible AI design choices, best practices, and performance for a set of AI service use cases.

Amazon Titan Embeddings G1 - Text models

Amazon Titan Embeddings text models include Amazon Titan Embeddings G1 - Text G1.

Text embeddings represent meaningful vector representations of unstructured text such as documents, paragraphs, and sentences. You input a body of text and the output is a (1 x n) vector. You can use embedding vectors for a wide variety of applications.

The Amazon Titan Embeddings G1 - Text model (`amazon.titan-embed-text-v1`). The Amazon Titan Embeddings G1 - Text – Text v1.2 can intake up to 8k tokens and outputs a vector of 1,536 dimensions. The model also works in 25+ different languages. The model is optimized for text retrieval tasks, but can also perform additional tasks, such as semantic similarity and clustering. Amazon Titan Embeddings G1 - Text – Text v1.2 also supports long documents, however, for retrieval tasks it is recommended to segment documents into logical segments (such as paragraphs or sections). In line with our recommendation.

> **Note**
>
> Titan Embeddings G1 - Text model doesn't support inference parameters such as `maxTokenCount` or `topP`.

To use the text or image embeddings models, use the `Invoke Model` API operation with `amazon.titan-embed-text-v1` or `amazon.titan-embed-image-v1` as the `model Id` and retrieve the embedding object in the response.

To see Jupyter notebook examples:

1. Sign in to the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/home.
2. From the left-side menu, choose **Base models**.
3. Scroll down and select the **Amazon Titan Embeddings G1 - Text** model
4. In the **Amazon Titan Embeddings G1 - Text** tab (depending on which model you chose), select **View example notebook** to see example notebooks for embeddings.

For more information on preparing your dataset for multimodal training, see Preparing your dataset.

Amazon Titan Multimodal Embeddings G1 model

Amazon Titan Foundation Models are pre-trained on large datasets, making them powerful, general-purpose models. Use them as-is, or customize them by fine tuning the models with your own data for a particular task without annotating large volumes of data.

There are three types of Titan models: embeddings, text generation, and image generation.

There are two Titan Multimodal Embeddings G1 models. The Titan Embeddings G1 – Text model translates text inputs (words, phrases or possibly large units of text) into numerical representations (known as embeddings) that contain the semantic meaning of the text. While this model will not generate text, it is useful for applications like personalization and search. By comparing embeddings, the model will produce more relevant and contextual responses than word matching. The Titan Multimodal Embeddings G1 model is used for use cases like searching images by text, by image for similarity, or by a combination of text and image. It translates the input image or text into an embedding that contain the semantic meaning of both the image and text in the same semantic space.

Titan Text models are generative LLMs for tasks such as summarization, text generation, classification, open-ended QnA, and information extraction. They are also trained on many different programming languages, as well as rich text format like tables, JSON, and .csv files, among other formats.

- **Model ID –** `amazon.titan-embed-image-v1`
- **Max input text tokens** – 128
- **Languages** – English
- **Max input image size** – 5 MB
- **Output vector size –** 1,024 (default), 384, 256

- **Inference types –** On-Demand, Provisioned Throughput
- **Supported use cases –** image search, recommendations, and personalization

Embedding length

Setting a custom embedding length is optional. The embedding default length is 1024 characters which will work for most use cases. The embedding length can be set to 256, 384, or 1024 characters. Larger embedding sizes create more detailed responses, but will also increase the computational time. Shorter embedding lengths are less detailed but will improve the response time.

```
# EmbeddingConfig Shape
{
 'outputEmbeddingLength': int // Optional, One of: [256, 384, 1024], default: 1024
}

# Updated API Payload Example
body = json.dumps({
 "inputText": "hi",
 "inputImage": image_string,
 "embeddingConfig": {
 "outputEmbeddingLength": 256
 }
})
```

Finetuning

- Input to the Amazon Titan Multimodal Embeddings G1 finetuning is image-text pairs.
- Image formats: PNG, JPEG
- Input image size limit: 5 MB
- Image dimensions: min: 128 px, max: 4,096 px
- Max number of tokens in caption: 128
- Training dataset size range: 1000 - 500,000
- Validation dataset size range: 8 - 50,000
- Caption length in characters: 0 - 2,560

- Maximum total pixels per image: 2048*2048*3
- Aspect ratio (w/h): min: 0.25, max: 4

Preparing datasets

For the training dataset, create a `.jsonl`file with multiple JSON lines. Each JSON line contains both an `image-ref` and `caption` attributes similar to [Sagemaker Augmented Manifest format](). A validation dataset is required. Auto-captioning is not currently supported.

```
{"image-ref": "s3://bucket-1/folder1/0001.png", "caption": "some text"}
{"image-ref": "s3://bucket-1/folder2/0002.png", "caption": "some text"}
{"image-ref": "s3://bucket-1/folder1/0003.png", "caption": "some text"}
```

For both the training and validation datasets, you will create `.jsonl`files with multiple JSON lines.

The Amazon S3 paths need to be in the same folders where you have provided permissions for Amazon Bedrock to access the data by attaching an IAM policy to your Amazon Bedrock service role. For more information on granting an IAM policies for training data, see [Grant custom jobs access to your training data]().

Hyperparameters

These values can be adjusted for the Multimodal Embeddings model hyperparameters. The default values will work well for most use cases.

- Learning rate - (min/max learning rate) – default: 5.00E-05, min: 5.00E-08, max: 1
- Batch size - Effective batch size – default: 576, min: 256, max: 9,216
- Max epochs – default: "auto", min: 1, max: 100

Amazon Titan Image Generator G1 model

Amazon Titan Image Generator G1 is an image generation model. It generates images from text, and allows users to upload and edit an existing image. This model can generate images from natural language text and can also be used to edit or generate variations for an existing or a generated image. Users can edit an image with a text prompt (without a mask) or parts of an image with an image mask. You can extend the boundaries of an image with outpainting, and fill in an image with inpainting. It can also generate variations of an image based on an optional text prompt.

Amazon Titan Image Generator G1 model supports instant customization that allows creators to import 1 to 5 reference images and generate a given subject image in novel context. The model preserves key characteritsics of the images, performs image-based style transfer without prompt engineering, and produces style mixing from multiple reference images, all without fine-tuning.

To continue supporting best practices in the responsible use of AI, Titan Foundation Models are built to detect and remove harmful content in the data, reject inappropriate content in the user input, and filter the models' outputs that contain inappropriate content (such as hate speech, profanity, and violence). The Titan Image Generator FM adds an invisible watermark to all the generated images.

You can use the watermark detection feature in Amazon Bedrock console (preview) or call Amazon Bedrock watermark detection API (preview) to check whether an image contains watermark from Titan Image Generator.

For more information on Amazon Titan Image Generator G1 prompt engineering guidelines, see Amazon Titan Image Generator G1 Prompt Engineering Best Practices.

- **Model ID –** `amazon.titan-image-generator-v1`
- **Max input characters** – 512 char
- **Max input image size** – 5 MB (only some specific resolutions are supported)
- **Max image size using in/outpainting** – 1,408 x 1,408 px
- **Max image size using image variation** – 4,096 x 4,096 px
- **Languages** – English
- **Output type** – image
- **Supported image types** – JPEG, JPG, PNG
- **Inference types** – On-Demand, Provisioned Throughput
- **Supported use cases** – image generation, image editing, image variations

Features

- Text-to-image (T2I) generation – Input a text prompt and generate a new image as output. The generated image captures the concepts described by the text prompt.

- Finetuning of a T2I model – Import several images to capture your own style and personalization and then fine tune the core T2I model. The fine-tuned model generates images that follow the style and personalization of a specific user.
- Image editing options – includes inpainting, outpainting, generating variations, and automatic editing without an image mask.
- Inpainting – Uses an image and a segmentation mask as input (either from the user or estimated by the model) and reconstructs the region within the mask. Use inpainting to remove masked elements and replace them with background pixels.
- Outpainting – Uses an image and a segmentation mask as input (either from the user or estimated by the model) and generates new pixels that seamlessly extend the region. Use precise outpainting to preserve the pixels of the masked image when extending the image to the boundaries. Use default outpainting to extend the pixels of the masked image to the image boundaries based on segmentation settings.
- Image variation – Uses an image and an optional prompt as input. It generates a new image that preserves the content of the input image, but variates its style and background.

> **Note**
>
> if you are using a fine-tuned model, you cannot use inpainting or outpainting features of the API or the model.

Parameters

For information on Amazon Titan Image Generator G1 inference parameters, see Amazon Titan Image Generator G1 inference parameters.

Fine-tuning

For more information on fine-tuning the Amazon Titan Image Generator G1 model, see the following pages.

- Prepare the datasets
- Amazon Titan Image Generator G1 model customization hyperparameters

Titan Image Generator G1 fine-tuning and pricing

The model uses the following example formula to calculate the total price per job:

Total Price = Steps * Batch size * Price per image seen

Minimum values (auto):

- Minimum steps (auto) - 500
- Minimum batch size - 8
- Default learning rate - 0.00001
- Price per image seen - 0.005

Fine-tuning hyperparameter settings

Steps – The number of times the model is exposed to each batch. There is no default step count set. You must select a number between 10 - 40,000, or a String value of "Auto."

Step settings - Auto – Amazon Bedrock determines a reasonable value based on training information. Select this option to prioritize model performance over training cost. The number of steps is determined automatically. This number will typically be between 1,000 and 8,000 based on your dataset. Job costs are impacted by the number of steps used to expose the model to the data. Refer to the pricing examples section of pricing details to understand how job cost is calculated. (See example table above to see how step count is related to number of images when Auto is selected.)

Step settings - Custom – You can enter the number of steps you want Bedrock to expose your custom model to the training data. This value can be between 10 and 40,000. You can reduce the cost per image produced by the model by using a lower step count value.

Batch size – The number of sample processed before model parameters are updated. This value is between 8 and 192 and is a multiple of 8.

Learning rate – The rate at which model parameters are updated after each batch of training data. This is a float value between 0 and 1. The learning rate is set to 0.00001 by default.

For more information on the fine-tuning procedure, see Submit a model customization job.

Output

Titan Image Generator G1 uses the output image size and quality to determine how an image is priced. Titan Image Generator G1 has two pricing segments based on size: one for 512*512 images and another for 1024*1024 images. Pricing is based on image size height*width, less than or equal to 512*512 or greater than 512*512.

For more information on Amazon Bedrock pricing, see [Amazon Bedrock Pricing.]

Watermark detection

> **Note**
>
> Watermark detection for the Amazon Bedrock console and API is available in public preview release and will only detect a watermark generated from Titan Image Generator G1. This feature is currently only available in the PDX and IAD regions. Watermark detection is a highly accurate detection of the watermark generated by Titan Image Generator G1. Images that are modified from the original image may produce less accurate detection results.

This model adds an invisible watermark to all generated images to reduce the spread of misinformation, assist with copyright protection, and track content usage. A watermark detection is available to help you confirm whether an image was generated by the Titan Image Generator G1 model, which checks for the existence of this watermark.

> **Note**
>
> Watermark Detection API is in preview and is subject to change. We recommend that you create a virtual environment to use the SDK. Because watermark detection APIs aren't available in the latest SDKs, we recommend that you uninstall the latest version of the SDK from the virtual environment before installing the version with the watermark detection APIs.

You can upload your image to detect if a watermark from Titan Image Generator G1 is present on the image. Use the console to detect a watermark from this model by following the below steps.

To detect a watermark with Titan Image Generator G1:

1. Open the Amazon Bedrock console at [Amazon Bedrock console]
2. Select **Overview** from the navigation pane in Amazon Bedrock. Choose the **Build and Test** tab.

3. In the **Safeguards** section, go to **Watermark detection** and choose **View watermark detection**.
4. Select **Upload image** and locate a file that is in JPG or PNG format. The maximum file size allowed is 5 MB.
5. Once uploaded, a thumbnail of image is shown with the name, file size, and the last date modified. Select X to delete or replace image from the **Upload** section.
6. Select **Analyze** to begin watermark detection analysis.
7. The image is previewed under **Results**, and indicates if a watermark is detected with **Watermark detected** below the image and a banner across the image. If no watermark is detected, the text below the image will say **Watermark NOT detected**.
8. To load the next image, select X in the thumbnail of the image in the **Upload** section and choose a new image to analyze.

Prompt Engineering Guidelines

Mask prompt – This algorithm classifies pixels into concepts. The user can give a text prompt that will be used to classify the areas of the image to mask, based on the interpretation of the mask prompt. The prompt option can interpret more complex prompts, and encode the mask into the segmentation algorithm.

Image mask – You can also use an image mask to set the mask values. The image mask can be combined with prompt input for the mask to improve accuracy. The image mask file must conform to the following parameters:

- Mask image values must be 0 (black) or 255 (white) for the mask image. The image mask area with the value of 0 will be regenerated with the image from the user prompt and/or input image.
- The `maskImage` field must be a base64 encoded image string.
- Mask image must have the same dimensions as the input image (same height and width).
- Only PNG or JPG files can be used for the input image and the mask image.
- Mask image must only use black and white pixels values.
- Mask image can only use the RGB channels (alpha channel not supported).

For more information on Amazon Titan Image Generator G1 prompt engineering, see [Amazon Titan Image Generator G1 Prompt Engineering Best Practices]().

For general prompt engineering guidelines, see Prompt Engineering Guidelines.

Security in Amazon Bedrock

Cloud security at AWS is the highest priority. As an AWS customer, you benefit from data centers and network architectures that are built to meet the requirements of the most security-sensitive organizations.

Security is a shared responsibility between AWS and you. The shared responsibility model describes this as security *of* the cloud and security *in* the cloud:

- **Security of the cloud** – AWS is responsible for protecting the infrastructure that runs AWS services in the AWS Cloud. AWS also provides you with services that you can use securely. Third-party auditors regularly test and verify the effectiveness of our security as part of the AWS Compliance Programs. To learn about the compliance programs that apply to Amazon Bedrock, see AWS Services in Scope by Compliance Program.
- **Security in the cloud** – Your responsibility is determined by the AWS service that you use. You are also responsible for other factors including the sensitivity of your data, your company's requirements, and applicable laws and regulations.

This documentation helps you understand how to apply the shared responsibility model when using Amazon Bedrock. The following topics show you how to configure Amazon Bedrock to meet your security and compliance objectives. You also learn how to use other AWS services that help you to monitor and secure your Amazon Bedrock resources.

Topics

Data protection

The AWS shared responsibility model applies to data protection in Amazon Bedrock. As described in this model, AWS is responsible for protecting the global infrastructure that runs all of the AWS Cloud. You are responsible for maintaining control over your content that is hosted on this infrastructure. You are also responsible for the security configuration and management tasks for the AWS services that you use. For more information about data privacy, see the Data Privacy FAQ. For information about data protection in Europe, see the AWS Shared Responsibility Model and GDPR blog post on the *AWS Security Blog*.

For data protection purposes, we recommend that you protect AWS account credentials and set up individual users with AWS IAM Identity Center or AWS Identity and Access Management (IAM). That way, each user is given only the permissions necessary to fulfill their job duties. We also recommend that you secure your data in the following ways:

- Use multi-factor authentication (MFA) with each account.
- Use SSL/TLS to communicate with AWS resources. We require TLS 1.2 and recommend TLS 1.3.
- Set up API and user activity logging with AWS CloudTrail.
- Use AWS encryption solutions, along with all default security controls within AWS services.
- Use advanced managed security services such as Amazon Macie, which assists in discovering and securing sensitive data that is stored in Amazon S3.
- If you require FIPS 140-2 validated cryptographic modules when accessing AWS through a command line interface or an API, use a FIPS endpoint. For more information about the available FIPS endpoints, see Federal Information Processing Standard (FIPS) 140-2.

We strongly recommend that you never put confidential or sensitive information, such as your customers' email addresses, into tags or free-form text fields such as a **Name** field. This includes when you work with Amazon Bedrock or other AWS services using the console, API, AWS CLI, or AWS SDKs. Any data that you enter into tags or free-form text fields used for names may be used for billing or diagnostic logs. If you provide a URL to an external server, we strongly recommend that you do not include credentials information in the URL to validate your request to that server.

Data protection in Amazon Bedrock

Amazon Bedrock doesn't use your prompts and continuations to train any AWS models or distribute them to third parties.

Each model provider has an escrow account that they upload their models to. The Amazon Bedrock inference account has permissions to call these models, but the escrow accounts themselves don't have outbound permissions to Amazon Bedrock accounts. Additionally, model providers don't have access to Amazon Bedrock logs or access to customer prompts and continuations.

Amazon Bedrock doesn’t store or log your data in its service logs.

Data protection in Amazon Bedrock model customization

Your training data isn't used to train the base Titan models or distributed to third parties. Other usage data, such as usage timestamps, logged account IDs, and other information logged by the service, is also not used to train the models.

Amazon Bedrock uses the fine tuning data you provide only for fine tuning an Amazon Bedrock foundation model. Amazon Bedrock doesn't use fine tuning data for any other purpose, such as training base foundation models.

Bedrock uses your training data with the CreateModelCustomizationJob action, or with the console, to create a custom model which is a fine tuned version of an Amazon Bedrock foundational model. Your custom models are managed and stored by AWS. By default, custom models are encrypted with AWS Key Management Service keys that AWS owns, but you can use your own AWS KMS keys to encrypt your custom models. You encrypt a custom model when you submit a fine tuning job with the console or programmatically with the `CreateModelCustomizationJob` action.

None of the training or validation data you provide for fine tuning is stored in Amazon Bedrock accounts, once the fine tuning job completes. During training, your data exists in AWS Service Management Connector instance memory, but is encrypted on these machines using an XTS-AES-256 cipher that is implemented on a hardware module, on the instance itself.

We don't recommend using confidential data to train a custom model as the model might generate inference responses based on that confidential data. If you use confidential data to train a custom model, the only way to prevent responses based on that data is to delete the custom model, remove the confidential data from your training dataset, and retrain the custom model.

Custom model metadata (name and Amazon Resource Name) and a provisioned model's metadata is stored in an Amazon DynamoDB table that is encrypted with a key that the Amazon Bedrock service owns.

Topics

- Data encryption

- Protect your data using Amazon VPC and AWS PrivateLink

Data encryption

Amazon Bedrock uses encryption to protect data at rest and data in transit.

Topics

Encryption in transit

Within AWS, all inter-network data in transit supports TLS 1.2 encryption.

Requests to the Amazon Bedrock API and console are made over a secure (SSL) connection. You pass AWS Identity and Access Management (IAM) roles to Amazon Bedrock to provide permissions to access resources on your behalf for training and deployment.

Encryption at rest

Amazon Bedrock provides Encryption of model customization jobs and artifacts at rest.

Key management

Use the AWS Key Management Service to manage the keys that you use to encrypt your resources. For more information, see AWS Key Management Service concepts. You can encrypt the following resources with a KMS key.

- Through Amazon Bedrock
  - Model customization jobs and their output custom models – During job creation in the console or by specifying the `customModelKmsKeyId` field in the CreateModelCustomizationJob API call.

    - Agents – During agent creation in the console or by specifying the field in the CreateAgent API call.
    - Data source ingestion jobs for knowledge bases – During knowledge base creation in the console or by specifying the `kmsKeyArn` field in the CreateDataSource or UpdateDataSource API call.
    - Vector stores in Amazon OpenSearch Service – During vector store creation. For more information, see Creating, listing, and deleting Amazon OpenSearch Service collections and Encryption of data at rest for Amazon OpenSearch Service.
- Through Amazon S3 – For more information, see Using server-side encryption with AWS KMS keys (SSE-KMS).
    - Training, validation, and output data for model customization
    - Data sources for knowledge bases
- Through AWS Secrets Manager – For more information, see Secret encryption and decryption in AWS Secrets Manager
    - Vector stores for third-party models

After you encrypt a resource, you can find the ARN of the KMS key by selecting a resource and viewing its **Details** in the console or by using the following `Get` API calls.

- GetModelCustomizationJob
- GetAgent
- GetIngestionJob

Encryption of model customization jobs and artifacts

By default, Amazon Bedrock encrypts the following model artifacts from your model customization jobs with an AWS managed key.

- The model customization job
- The output files (training and validation metrics) from the model customization job
- The resulting custom model

Optionally, you can encrypt the model artifacts by creating a customer managed key. For more information about AWS KMS keys, see Customer managed keys in the *AWS Key Management Service Developer Guide*. To use a customer managed key, carry out the following steps.

1. Create a customer managed key with the AWS Key Management Service.
2. Attach a resource-based policy with permissions for the specified-roles to create or use custom models.

Topics

Create a customer managed key

First ensure that you have `CreateKey` permissions. Then follow the steps at Creating keys to create a customer managed key either in the AWS KMS console or the CreateKey API operation. Make sure to create a symmetric encryption key.

Creation of the key returns an `Arn` for the key that you can use as the `customModelKmsKeyId` when submitting a model customization job.

Create a key policy and attach it to the customer managed key

Attach the following resource-based policy to the KMS key by following the steps at Creating a key policy. The policy contains two statements.

1. Permissions for a role to encrypt model customization artifacts. Add ARNs of custom model builder roles to the `Principal` field.
2. Permissions for a role to use a custom model in inference. Add ARNs of custom model user roles to the `Principal` field.

```
{
    "Version": "2012-10-17",
    "Id": "KMS Key Policy",
    "Statement": [
        {
```

```
            "Sid": "Permissions for custom model builders",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::account-id:user/role"
            },
            "Action": [
                "kms:Decrypt",
                "kms:GenerateDataKey",
                "kms:DescribeKey",
                "kms:CreateGrant"
            ],
            "Resource": "*"
        },
        {
            "Sid": "Permissions for custom model users",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::account-id:user/role"
            },
            "Action": "kms:Decrypt",
            "Resource": "*"
        }
}
```

Encryption of training, validation, and output data

When you use Amazon Bedrock to run a model customization job, you store the input (training/validation data) files in your Amazon S3 bucket. When the job completes, Amazon Bedrock stores the output metrics files in the S3 bucket that you specifed when creating the job and the resulting custom model artifacts in an Amazon S3 bucket controlled by AWS.

The input and output files are encrypted with Amazon S3 SSE-S3 server-side encryption by default, using an *AWS managed key*. This type of key is created, managed, and used on your behalf by AWS.

You can instead choose to encrypt these files with a *customer managed key* that you create, own, and manage yourself. Refer to the preceding sections and the following links to learn how to create customer managed keys and key policies.

- To learn more about Amazon S3 SSE-S3 server-side encryption, see Using server-side encryption with Amazon S3 managed keys (SSE-S3)
- To learn more about customer managed keys for encrypting S3 objects, see Using server-side encryption with AWS KMS keys (SSE-KMS)

Encryption of agent resources

Amazon Bedrock encrypts your agent's session information. By default, Amazon Bedrock encrypts this data using an AWS managed key. Optionally, you can encrypt the agent artifacts using a customer managed key.

For more information about AWS KMS keys, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

If you encrypt sessions with your agent with a custom KMS key, you must set up the following identity-based policy and resource-based policy to allow Amazon Bedrock to encrypt and decrypt agent resources on your behalf.

1. Attach the following identity-based policy to an IAM role or user with permissions to make `InvokeAgent` calls. This policy validates the user making an `InvokeAgent` call has KMS permissions. Replace the *${region}*, *${account-id}*, *${agent-id}*, and *${key-id}* with the appropriate values.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Allow Amazon Bedrock to encrypt and decrypt Agent resources on behalf of authorized users",
            "Effect": "Allow",
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:${region}:${account-id}:key/${key-id}",
            "Condition": {
                "StringEquals": {
                    "kms:EncryptionContext:aws:bedrock:arn": "arn:aws:bedrock:${region}:${account-id}:agent/${agent-id}"
                }
            }
        }
    ]
}
```

2. Attach the following resource-based policy to your KMS key. Change the scope of the permissions as necessary. Replace the *${region}*, *${account-id}*, *${agent-id}*, and *${key-id}* with the appropriate values.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Allow account root to modify the KMS key, not used by Amazon
 Bedrock.",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::${account-id}:root"
            },
            "Action": "kms:*",
            "Resource": "arn:aws:kms:${region}:${account-id}:key/${key-id}"
        },
        {
            "Sid": "Allow Amazon Bedrock to encrypt and decrypt Agent resources on
 behalf of authorized users",
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:${region}:${account-id}:key/${key-id}",
            "Condition": {
                "StringEquals": {
                    "kms:EncryptionContext:aws:bedrock:arn":
 "arn:aws:bedrock:${region}:${account-id}:agent/${agent-id}"
                }
            }
        },
        {
            "Sid": "Allow the service role to use the key to encrypt and decrypt
 Agent resources",
            "Effect": "Allow",
            "Principal": {
                "AWS": "arn:aws:iam::${account-id}:role/
AmazonBedrockExecutionRoleForAgents_${suffix}"
```

```
            },
            "Action": [
                "kms:GenerateDataKey*",
                "kms:Decrypt",
            ],
            "Resource": "arn:aws:kms:${region}:${account-id}:key/${key-id}"
},
        {
           "Sid": "Allow the attachment of persistent resources",
           "Effect": "Allow",
           "Principal": {
               "Service": "bedrock.amazonaws.com"
           },
           "Action": [
               "kms:CreateGrant",
               "kms:ListGrants",
               "kms:RevokeGrant"
           ],
           "Resource": "*",
           "Condition": {
               "Bool": {
                   "kms:GrantIsForAWSResource": "true"
               }
           }
         }
    ]
}
```

Encryption of knowledge base resources

Amazon Bedrock encrypts resources related to your knowledge bases. By default, Amazon Bedrock encrypts this data using an AWS managed key. Optionally, you can encrypt the model artifacts using a customer managed key.

Encryption with a KMS key can occur with the following processes:

- Transient data storage while ingesting your data sources
- Passing information to OpenSearch Service if you let Amazon Bedrock set up your vector database
- Querying a knowledge base

The following resources used by your knowledge bases can be encrypted with a KMS key. If you encrypt them, you need to add permissions to decrypt the KMS key.

- Data sources stored in an Amazon S3 bucket
- Third-party vector stores

For more information about AWS KMS keys, see Customer managed keys in the *AWS Key Management Service Developer Guide*.

Topics

- Encryption of transient data storage during data ingestion
- Encryption of information passed to Amazon OpenSearch Service
- Encryption of knowledge base retrieval
- Permissions to decrypt your AWS KMS key for your data sources in Amazon S3
- Permissions to decrypt an AWS Secrets Manager secret for the vector store containing your knowledge base

Encryption of transient data storage during data ingestion

When you set up a data ingestion job for your knowledge base, you can encrypt the job with a custom KMS key.

To allow the creation of a AWS KMS key for transient data storage in the process of ingesting your data source, attach the following policy to your Amazon Bedrock service role. Replace the *region*, *account-id*, and *key-id* with the appropriate values.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
          "Effect": "Allow",
          "Action": [
              "kms:GenerateDataKey",
              "kms:Decrypt"
          ],
          "Resource": [
              "arn:aws:kms:region:account-id:key/key-id"
          ]
```

```
        }
    ]
}
```

Encryption of information passed to Amazon OpenSearch Service

If you opt to let Amazon Bedrock create a vector store in Amazon OpenSearch Service for your knowledge base, Amazon Bedrock can pass a KMS key that you choose to Amazon OpenSearch Service for encryption. To learn more about encryption in Amazon OpenSearch Service, see Encryption in Amazon OpenSearch Service.

Encryption of knowledge base retrieval

You can encrypt sessions in which you generate responses from querying a knowledge base with a KMS key. To do so, include the ARN of a KMS key in the `kmsKeyArn` field when making a RetrieveAndGenerate request. Attach the following policy, replacing the *`values`* appropriately to allow Amazon Bedrock to encrypt the session context.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": [
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": "arn:aws:kms:region:account-id:key/key-id
        }
    ]
}
```

Permissions to decrypt your AWS KMS key for your data sources in Amazon S3

You store the data sources for your knowledge base in your Amazon S3 bucket. To encrypt these documents at rest, you can use the Amazon S3 SSE-S3 server-side encryption option. With this option, objects are encrypted with service keys managed by the Amazon S3 service.

For more information, see Protecting data using server-side encryption with Amazon S3-managed encryption keys (SSE-S3) in the *Amazon Simple Storage Service User Guide*.

If you encrypted your data sources in Amazon S3 with a custom AWS KMS key, attach the following policy to your Amazon Bedrock service role to allow Amazon Bedrock to decrypt your key. Replace *region* and *account-id* with the region and account ID to which the key belongs. Replace *key-id* with the ID of your AWS KMS key.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "KMS:Decrypt",
        ],
        "Resource": [
            "arn:aws:kms:region:account-id:key/key-id"
        ],
        "Condition": {
            "StringEquals": {
                "kms:ViaService": [
                    "s3.region.amazonaws.com"
                ]
            }
        }
    }]
}
```

Permissions to decrypt an AWS Secrets Manager secret for the vector store containing your knowledge base

If the vector store containing your knowledge base is configured with an AWS Secrets Manager secret, you can encrypt the secret with a custom AWS KMS key by following the steps at Secret encryption and decryption in AWS Secrets Manager.

If you do so, you attach the following policy to your Amazon Bedrock service role to allow it to decrypt your key. Replace *region* and *account-id* with the region and account ID to which the key belongs. Replace *key-id* with the ID of your AWS KMS key.

```
{
    "Version": "2012-10-17",
    "Statement": [
```

```
        {
            "Effect": "Allow",
            "Action": [
                "kms:Decrypt"
            ],
            "Resource": [
                "arn:aws:kms:region:account-id:key/key-id"
            ]
        }
    ]
}
```

Protect your data using Amazon VPC and AWS PrivateLink

To control access to your data, we recommend that you use a virtual private cloud (VPC) with Amazon VPC. Using a VPC protects your data and lets you monitor all network traffic in and out of the AWS job containers by using VPC Flow Logs. You can further protect your data by configuring your VPC so that your data isn't available over the internet and instead creating a VPC interface endpoint with AWS PrivateLink to establish a private connection to your data.

For an example of using VPC to protect data that you integrate with Amazon Bedrock see Protect model customization jobs using a VPC.

Use interface VPC endpoints (AWS PrivateLink)

You can use AWS PrivateLink to create a private connection between your VPC and Amazon Bedrock. You can access Amazon Bedrock as if it were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to access Amazon Bedrock.

You establish this private connection by creating an *interface endpoint*, powered by AWS PrivateLink. We create an endpoint network interface in each subnet that you enable for the interface endpoint. These are requester-managed network interfaces that serve as the entry point for traffic destined for Amazon Bedrock.

For more information, see Access AWS services through AWS PrivateLink in the *AWS PrivateLink Guide*.

Considerations for Amazon Bedrock VPC endpoints

Before you set up an interface endpoint for Amazon Bedrock, review Considerations in the *AWS PrivateLink Guide*.

Amazon Bedrock supports making the following API calls through VPC endpoints.

Category	Endpoint prefix
Amazon Bedrock Control Plane API actions	`bedrock`
Amazon Bedrock Runtime API actions	`bedrock-runtime`
Agents for Amazon Bedrock Build-time API actions	`bedrock-agent`
Agents for Amazon Bedrock Runtime API actions	`bedrock-agent-runtime`

Availability Zones

Amazon Bedrock and Agents for Amazon Bedrock endpoints are available in multiple Availability Zones.

Create an interface endpoint for Amazon Bedrock

You can create an interface endpoint for Amazon Bedrock using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

Create an interface endpoint for Amazon Bedrock using any of the following service names:

- `com.amazonaws.`*`region`*`.bedrock`
- `com.amazonaws.`*`region`*`.bedrock-runtime`
- `com.amazonaws.`*`region`*`.bedrock-agent`
- `com.amazonaws.`*`region`*`.bedrock-agent-runtime`

After you create the endpoint, you have the option to enable a private DNS hostname. Enable this setting by selecting Enable Private DNS Name in the VPC console when you create the VPC endpoint.

If you enable private DNS for the interface endpoint, you can make API requests to Amazon Bedrock using its default Regional DNS name. The following examples show the format of the default Regional DNS names.

- `bedrock.region.amazonaws.com`
- `bedrock-runtime.region.amazonaws.com`
- `bedrock-agent.region.amazonaws.com`
- `bedrock-agent-runtime.region.amazonaws.com`

Create an endpoint policy for your interface endpoint

An endpoint policy is an IAM resource that you can attach to an interface endpoint. The default endpoint policy allows full access to Amazon Bedrock through the interface endpoint. To control the access allowed to Amazon Bedrock from your VPC, attach a custom endpoint policy to the interface endpoint.

An endpoint policy specifies the following information:

- The principals that can perform actions (AWS accounts, IAM users, and IAM roles).
- The actions that can be performed.
- The resources on which the actions can be performed.

For more information, see Control access to services using endpoint policies in the *AWS PrivateLink Guide*.

Example: VPC endpoint policy for Amazon Bedrock actions

The following is an example of a custom endpoint policy. When you attach this resource-based policy to your interface endpoint, it grants access to the listed Amazon Bedrock actions for all principals on all resources.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Principal": "*",
         "Effect": "Allow",
         "Action": [
            "bedrock:InvokeModel",
            "bedrock:InvokeModelWithResponseStream"
         ],
         "Resource":"*"
      }
```

```
    ]
}
```

Identity and access management for Amazon Bedrock

AWS Identity and Access Management (IAM) is an AWS service that helps an administrator securely control access to AWS resources. IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon Bedrock resources. IAM is an AWS service that you can use with no additional charge.

Topics

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in Amazon Bedrock.

Service user – If you use the Amazon Bedrock service to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more Amazon Bedrock features to do your work, you might need additional permissions. Understanding how access is managed can help you request the right permissions from your administrator. If you cannot access a feature in Amazon Bedrock, see Troubleshooting Amazon Bedrock identity and access.

Service administrator – If you're in charge of Amazon Bedrock resources at your company, you probably have full access to Amazon Bedrock. It's your job to determine which Amazon Bedrock features and resources your service users should access. You must then submit requests to your IAM administrator to change the permissions of your service users. Review the information on this page to understand the basic concepts of IAM. To learn more about how your company can use IAM with Amazon Bedrock, see How Amazon Bedrock works with IAM.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to Amazon Bedrock. To view example Amazon Bedrock identity-based policies that you can use in IAM, see Identity-based policy examples for Amazon Bedrock.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be *authenticated* (signed in to AWS) as the AWS account root user, as an IAM user, or by assuming an IAM role.

You can sign in to AWS as a federated identity by using credentials provided through an identity source. AWS IAM Identity Center (IAM Identity Center) users, your company's single sign-on authentication, and your Google or Facebook credentials are examples of federated identities. When you sign in as a federated identity, your administrator previously set up identity federation using IAM roles. When you access AWS by using federation, you are indirectly assuming a role.

Depending on the type of user you are, you can sign in to the AWS Management Console or the AWS access portal. For more information about signing in to AWS, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

If you access AWS programmatically, AWS provides a software development kit (SDK) and a command line interface (CLI) to cryptographically sign your requests by using your credentials. If you don't use AWS tools, you must sign requests yourself. For more information about using the recommended method to sign requests yourself, see Signing AWS API requests in the *IAM User Guide*.

Regardless of the authentication method that you use, you might be required to provide additional security information. For example, AWS recommends that you use multi-factor authentication (MFA) to increase the security of your account. To learn more, see Multi-factor authentication in the *AWS IAM Identity Center User Guide* and Using multi-factor authentication (MFA) in AWS in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity that has complete access to all AWS services and resources in the account. This identity is called the AWS account *root user* and is accessed by signing in with the email address and password that you used to create the account. We strongly recommend that you don't use the root user for your everyday tasks. Safeguard your root user credentials and use them to perform the tasks that only the root user can perform. For the complete list of tasks that require you to sign in as the root user, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users, including users that require administrator access, to use federation with an identity provider to access AWS services by using temporary credentials.

A *federated identity* is a user from your enterprise user directory, a web identity provider, the AWS Directory Service, the Identity Center directory, or any user that accesses AWS services by using credentials provided through an identity source. When federated identities access AWS accounts, they assume roles, and the roles provide temporary credentials.

For centralized access management, we recommend that you use AWS IAM Identity Center. You can create users and groups in IAM Identity Center, or you can connect and synchronize to a set of users and groups in your own identity source for use across all your AWS accounts and applications. For information about IAM Identity Center, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity within your AWS account that has specific permissions for a single person or application. Where possible, we recommend relying on temporary credentials instead of creating IAM users who have long-term credentials such as passwords and access keys. However, if you have specific use cases that require long-term credentials with IAM users, we recommend that you rotate access keys. For more information, see Rotate access keys regularly for use cases that require long-term credentials in the *IAM User Guide*.

An *IAM group* is an identity that specifies a collection of IAM users. You can't sign in as a group. You can use groups to specify permissions for multiple users at a time. Groups make permissions easier to manage for large sets of users. For example, you could have a group named *IAMAdmins* and give that group permissions to administer IAM resources.

Users are different from roles. A user is uniquely associated with one person or application, but a role is intended to be assumable by anyone who needs it. Users have permanent long-term credentials, but roles provide temporary credentials. To learn more, see When to create an IAM user (instead of a role) in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity within your AWS account that has specific permissions. It is similar to an IAM user, but is not associated with a specific person. You can temporarily assume an IAM role in the AWS Management Console by switching roles. You can assume a role by calling an AWS CLI or AWS API operation or by using a custom URL. For more information about methods for using roles, see Using IAM roles in the *IAM User Guide*.

IAM roles with temporary credentials are useful in the following situations:

- **Federated user access** – To assign permissions to a federated identity, you create a role and define permissions for the role. When a federated identity authenticates, the identity is associated with the role and is granted the permissions that are defined by the role. For information about roles for federation, see Creating a role for a third-party Identity Provider in the *IAM User Guide*. If you use IAM Identity Center, you configure a permission set. To control what your identities can access after they authenticate, IAM Identity Center correlates the permission set to a role in IAM. For information about permissions sets, see Permission sets in the *AWS IAM Identity Center User Guide*.
- **Temporary IAM user permissions** – An IAM user or role can assume an IAM role to temporarily take on different permissions for a specific task.
- **Cross-account access** – You can use an IAM role to allow someone (a trusted principal) in a different account to access resources in your account. Roles are the primary way to grant cross-account access. However, with some AWS services, you can attach a policy directly to a resource (instead of using a role as a proxy). To learn the difference between roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.
- **Cross-service access** – Some AWS services use features in other AWS services. For example, when you make a call in a service, it's common for that service to run applications in Amazon EC2 or store objects in Amazon S3. A service might do this using the calling principal's permissions, using a service role, or using a service-linked role.
  - **Forward access sessions (FAS)** – When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must

have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

- **Service role** – A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.
- **Service-linked role** – A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

- **Applications running on Amazon EC2** – You can use an IAM role to manage temporary credentials for applications that are running on an EC2 instance and making AWS CLI or AWS API requests. This is preferable to storing access keys within the EC2 instance. To assign an AWS role to an EC2 instance and make it available to all of its applications, you create an instance profile that is attached to the instance. An instance profile contains the role and enables programs that are running on the EC2 instance to get temporary credentials. For more information, see Using an IAM role to grant permissions to applications running on Amazon EC2 instances in the *IAM User Guide*.

To learn whether to use IAM roles or IAM users, see When to create an IAM role (instead of a user) in the *IAM User Guide*.

Managing access using policies

You control access in AWS by creating policies and attaching them to AWS identities or resources. A policy is an object in AWS that, when associated with an identity or resource, defines their permissions. AWS evaluates these policies when a principal (user, root user, or role session) makes a request. Permissions in the policies determine whether the request is allowed or denied. Most policies are stored in AWS as JSON documents. For more information about the structure and contents of JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

IAM policies define permissions for an action regardless of the method that you use to perform the operation. For example, suppose that you have a policy that allows the `iam:GetRole` action. A user with that policy can get role information from the AWS Management Console, the AWS CLI, or the AWS API.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

Identity-based policies can be further categorized as *inline policies* or *managed policies*. Inline policies are embedded directly into a single user, group, or role. Managed policies are standalone policies that you can attach to multiple users, groups, and roles in your AWS account. Managed policies include AWS managed policies and customer managed policies. To learn how to choose between a managed policy or an inline policy, see Choosing between managed policies and inline policies in the *IAM User Guide*.

Resource-based policies

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

Resource-based policies are inline policies that are located in that service. You can't use AWS managed policies from IAM in a resource-based policy.

Access control lists (ACLs)

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

Amazon S3, AWS WAF, and Amazon VPC are examples of services that support ACLs. To learn more about ACLs, see [Access control list (ACL) overview](#) in the *Amazon Simple Storage Service Developer Guide*.

Other policy types

AWS supports additional, less-common policy types. These policy types can set the maximum permissions granted to you by the more common policy types.

- **Permissions boundaries** – A permissions boundary is an advanced feature in which you set the maximum permissions that an identity-based policy can grant to an IAM entity (IAM user or role). You can set a permissions boundary for an entity. The resulting permissions are the intersection of an entity's identity-based policies and its permissions boundaries. Resource-based policies that specify the user or role in the `Principal` field are not limited by the permissions boundary. An explicit deny in any of these policies overrides the allow. For more information about permissions boundaries, see [Permissions boundaries for IAM entities](#) in the *IAM User Guide*.
- **Service control policies (SCPs)** – SCPs are JSON policies that specify the maximum permissions for an organization or organizational unit (OU) in AWS Organizations. AWS Organizations is a service for grouping and centrally managing multiple AWS accounts that your business owns. If you enable all features in an organization, then you can apply service control policies (SCPs) to any or all of your accounts. The SCP limits permissions for entities in member accounts, including each AWS account root user. For more information about Organizations and SCPs, see [How SCPs work](#) in the *AWS Organizations User Guide*.
- **Session policies** – Session policies are advanced policies that you pass as a parameter when you programmatically create a temporary session for a role or federated user. The resulting session's permissions are the intersection of the user or role's identity-based policies and the session policies. Permissions can also come from a resource-based policy. An explicit deny in any of these policies overrides the allow. For more information, see [Session policies](#) in the *IAM User Guide*.

Multiple policy types

When multiple types of policies apply to a request, the resulting permissions are more complicated to understand. To learn how AWS determines whether to allow a request when multiple policy types are involved, see [Policy evaluation logic](#) in the *IAM User Guide*.

How Amazon Bedrock works with IAM

Before you use IAM to manage access to Amazon Bedrock, learn what IAM features are available to use with Amazon Bedrock.

IAM features you can use with Amazon Bedrock

IAM feature	Amazon Bedrock support
Identity-based policies	Yes
Resource-based policies	No
Policy actions	Yes
Policy resources	Yes
Policy condition keys	Yes
ACLs	No
ABAC (tags in policies)	Yes
Temporary credentials	Yes
Principal permissions	Yes
Service roles	Yes
Service-linked roles	No

To get a high-level view of how Amazon Bedrock and other AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

Identity-based policies for Amazon Bedrock

Supports identity-based policies	Yes

Identity-based policies are JSON permissions policy documents that you can attach to an identity, such as an IAM user, group of users, or role. These policies control what actions users and roles can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Creating IAM policies in the *IAM User Guide*.

With IAM identity-based policies, you can specify allowed or denied actions and resources as well as the conditions under which actions are allowed or denied. You can't specify the principal in an identity-based policy because it applies to the user or role to which it is attached. To learn about all of the elements that you can use in a JSON policy, see IAM JSON policy elements reference in the *IAM User Guide*.

Identity-based policy examples for Amazon Bedrock

To view examples of Amazon Bedrock identity-based policies, see Identity-based policy examples for Amazon Bedrock.

Resource-based policies within Amazon Bedrock

Supports resource-based policies	No

Resource-based policies are JSON policy documents that you attach to a resource. Examples of resource-based policies are IAM *role trust policies* and Amazon S3 *bucket policies*. In services that support resource-based policies, service administrators can use them to control access to a specific resource. For the resource where the policy is attached, the policy defines what actions a specified principal can perform on that resource and under what conditions. You must specify a principal in a resource-based policy. Principals can include accounts, users, roles, federated users, or AWS services.

To enable cross-account access, you can specify an entire account or IAM entities in another account as the principal in a resource-based policy. Adding a cross-account principal to a resource-based policy is only half of establishing the trust relationship. When the principal and the resource are in different AWS accounts, an IAM administrator in the trusted account must also grant the principal entity (user or role) permission to access the resource. They grant permission by attaching an identity-based policy to the entity. However, if a resource-based policy grants access to a principal in the same account, no additional identity-based policy is required. For more information, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Policy actions for Amazon Bedrock

Supports policy actions	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Action` element of a JSON policy describes the actions that you can use to allow or deny access in a policy. Policy actions usually have the same name as the associated AWS API operation. There are some exceptions, such as *permission-only actions* that don't have a matching API operation. There are also some operations that require multiple actions in a policy. These additional actions are called *dependent actions*.

Include actions in a policy to grant permissions to perform the associated operation.

To see a list of Amazon Bedrock actions, see Actions defined by Amazon Bedrock in the *Service Authorization Reference*.

Policy actions in Amazon Bedrock use the following prefix before the action:

```
bedrock
```

To specify multiple actions in a single statement, separate them with commas.

```
"Action": [
   "bedrock:action1",
   "bedrock:action2"
]
```

To view examples of Amazon Bedrock identity-based policies, see Identity-based policy examples for Amazon Bedrock.

Policy resources for Amazon Bedrock

Supports policy resources	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Resource` JSON policy element specifies the object or objects to which the action applies. Statements must include either a `Resource` or a `NotResource` element. As a best practice, specify a resource using its Amazon Resource Name (ARN). You can do this for actions that support a specific resource type, known as *resource-level permissions*.

For actions that don't support resource-level permissions, such as listing operations, use a wildcard (*) to indicate that the statement applies to all resources.

```
"Resource": "*"
```

To see a list of Amazon Bedrock resource types and their ARNs, see Resources defined by Amazon Bedrock in the *Service Authorization Reference*. To learn with which actions you can specify the ARN of each resource, see Actions defined by Amazon Bedrock.

Some Amazon Bedrock API actions support multiple resources. For example, AssociateAgentKnowledgeBase accesses *AGENT12345* and *KB12345678*, so a principal must have permissions to access both resources. To specify multiple resources in a single statement, separate the ARNs with commas.

```
"Resource": [
    "arn:aws:bedrock:aws-region:111122223333:agent/AGENT12345",
    "arn:aws:bedrock:aws-region:111122223333:knowledge-base/KB12345678"
]
```

To view examples of Amazon Bedrock identity-based policies, see Identity-based policy examples for Amazon Bedrock.

Policy condition keys for Amazon Bedrock

Supports service-specific policy condition keys	Yes

Administrators can use AWS JSON policies to specify who has access to what. That is, which **principal** can perform **actions** on what **resources**, and under what **conditions**.

The `Condition` element (or `Condition` *block*) lets you specify conditions in which a statement is in effect. The `Condition` element is optional. You can create conditional expressions that use condition operators, such as equals or less than, to match the condition in the policy with values in the request.

If you specify multiple `Condition` elements in a statement, or multiple keys in a single `Condition` element, AWS evaluates them using a logical `AND` operation. If you specify multiple values for a single condition key, AWS evaluates the condition using a logical `OR` operation. All of the conditions must be met before the statement's permissions are granted.

You can also use placeholder variables when you specify conditions. For example, you can grant an IAM user permission to access a resource only if it is tagged with their IAM user name. For more information, see IAM policy elements: variables and tags in the *IAM User Guide*.

AWS supports global condition keys and service-specific condition keys. To see all AWS global condition keys, see AWS global condition context keys in the *IAM User Guide*.

To see a list of Amazon Bedrock condition keys, see Condition Keys for Amazon Bedrock in the *Service Authorization Reference*. To learn with which actions and resources you can use a condition key, see Actions defined by Amazon Bedrock.

All Amazon Bedrock actions support condition keys using Amazon Bedrock models as the resource.

To view examples of Amazon Bedrock identity-based policies, see Identity-based policy examples for Amazon Bedrock.

ACLs in Amazon Bedrock

Supports ACLs	No

Access control lists (ACLs) control which principals (account members, users, or roles) have permissions to access a resource. ACLs are similar to resource-based policies, although they do not use the JSON policy document format.

ABAC with Amazon Bedrock

Supports ABAC (tags in policies)	Yes

Attribute-based access control (ABAC) is an authorization strategy that defines permissions based on attributes. In AWS, these attributes are called *tags*. You can attach tags to IAM entities (users or roles) and to many AWS resources. Tagging entities and resources is the first step of ABAC. Then you design ABAC policies to allow operations when the principal's tag matches the tag on the resource that they are trying to access.

ABAC is helpful in environments that are growing rapidly and helps with situations where policy management becomes cumbersome.

To control access based on tags, you provide tag information in the condition element of a policy using the `aws:ResourceTag/key-name`, `aws:RequestTag/key-name`, or `aws:TagKeys` condition keys.

If a service supports all three condition keys for every resource type, then the value is **Yes** for the service. If a service supports all three condition keys for only some resource types, then the value is **Partial**.

For more information about ABAC, see What is ABAC? in the *IAM User Guide*. To view a tutorial with steps for setting up ABAC, see Use attribute-based access control (ABAC) in the *IAM User Guide*.

Using temporary credentials with Amazon Bedrock

Supports temporary credentials	Yes

Some AWS services don't work when you sign in using temporary credentials. For additional information, including which AWS services work with temporary credentials, see AWS services that work with IAM in the *IAM User Guide*.

You are using temporary credentials if you sign in to the AWS Management Console using any method except a user name and password. For example, when you access AWS using your company's single sign-on (SSO) link, that process automatically creates temporary credentials. You also automatically create temporary credentials when you sign in to the console as a user and then switch roles. For more information about switching roles, see Switching to a role (console) in the *IAM User Guide*.

You can manually create temporary credentials using the AWS CLI or AWS API. You can then use those temporary credentials to access AWS. AWS recommends that you dynamically generate temporary credentials instead of using long-term access keys. For more information, see Temporary security credentials in IAM.

Cross-service principal permissions for Amazon Bedrock

Supports forward access sessions (FAS)	Yes

When you use an IAM user or role to perform actions in AWS, you are considered a principal. When you use some services, you might perform an action that then initiates another action in a different service. FAS uses the permissions of the principal calling an AWS service, combined with the requesting AWS service to make requests to downstream services. FAS requests are only made when a service receives a request that requires interactions with other AWS services or resources to complete. In this case, you must have permissions to perform both actions. For policy details when making FAS requests, see Forward access sessions.

Service roles for Amazon Bedrock

Supports service roles	Yes

A service role is an IAM role that a service assumes to perform actions on your behalf. An IAM administrator can create, modify, and delete a service role from within IAM. For more information, see Creating a role to delegate permissions to an AWS service in the *IAM User Guide*.

Warning

Changing the permissions for a service role might break Amazon Bedrock functionality. Edit service roles only when Amazon Bedrock provides guidance to do so.

Service-linked roles for Amazon Bedrock

Supports service-linked roles	No

A service-linked role is a type of service role that is linked to an AWS service. The service can assume the role to perform an action on your behalf. Service-linked roles appear in your AWS account and are owned by the service. An IAM administrator can view, but not edit the permissions for service-linked roles.

Identity-based policy examples for Amazon Bedrock

By default, users and roles don't have permission to create or modify Amazon Bedrock resources. They also can't perform tasks by using the AWS Management Console, AWS Command Line Interface (AWS CLI), or AWS API. To grant users permission to perform actions on the resources that they need, an IAM administrator can create IAM policies. The administrator can then add the IAM policies to roles, and users can assume the roles.

To learn how to create an IAM identity-based policy by using these example JSON policy documents, see Creating IAM policies in the *IAM User Guide*.

For details about actions and resource types defined by Amazon Bedrock, including the format of the ARNs for each of the resource types, see Actions, Resources, and Condition Keys for Amazon Bedrock in the *Service Authorization Reference*.

Topics

Policy best practices

Identity-based policies determine whether someone can create, access, or delete Amazon Bedrock resources in your account. These actions can incur costs for your AWS account. When you create or edit identity-based policies, follow these guidelines and recommendations:

- **Get started with AWS managed policies and move toward least-privilege permissions** – To get started granting permissions to your users and workloads, use the *AWS managed policies* that grant permissions for many common use cases. They are available in your AWS account. We recommend that you reduce permissions further by defining AWS customer managed policies that are specific to your use cases. For more information, see AWS managed policies or AWS managed policies for job functions in the *IAM User Guide*.

- **Apply least-privilege permissions** – When you set permissions with IAM policies, grant only the permissions required to perform a task. You do this by defining the actions that can be taken on specific resources under specific conditions, also known as *least-privilege permissions*. For more information about using IAM to apply permissions, see Policies and permissions in IAM in the *IAM User Guide*.
- **Use conditions in IAM policies to further restrict access** – You can add a condition to your policies to limit access to actions and resources. For example, you can write a policy condition to specify that all requests must be sent using SSL. You can also use conditions to grant access to service actions if they are used through a specific AWS service, such as AWS CloudFormation. For more information, see IAM JSON policy elements: Condition in the *IAM User Guide*.
- **Use IAM Access Analyzer to validate your IAM policies to ensure secure and functional permissions** – IAM Access Analyzer validates new and existing policies so that the policies adhere to the IAM policy language (JSON) and IAM best practices. IAM Access Analyzer provides more than 100 policy checks and actionable recommendations to help you author secure and functional policies. For more information, see IAM Access Analyzer policy validation in the *IAM User Guide*.
- **Require multi-factor authentication (MFA)** – If you have a scenario that requires IAM users or a root user in your AWS account, turn on MFA for additional security. To require MFA when API operations are called, add MFA conditions to your policies. For more information, see Configuring MFA-protected API access in the *IAM User Guide*.

For more information about best practices in IAM, see Security best practices in IAM in the *IAM User Guide*.

Use the Amazon Bedrock console

To access the Amazon Bedrock console, you must have a minimum set of permissions. These permissions must allow you to list and view details about the Amazon Bedrock resources in your AWS account. If you create an identity-based policy that is more restrictive than the minimum required permissions, the console won't function as intended for entities (users or roles) with that policy.

You don't need to allow minimum console permissions for users that are making calls only to the AWS CLI or the AWS API. Instead, allow access to only the actions that match the API operation that they're trying to perform.

To ensure that users and roles can still use the Amazon Bedrock console, also attach the Amazon Bedrock AmazonBedrockFullAccess or AmazonBedrockReadOnly AWS managed policy to the entities. For more information, see Adding permissions to a user in the *IAM User Guide*.

Allow users to view their own permissions

This example shows how you might create a policy that allows IAM users to view the inline and managed policies that are attached to their user identity. This policy includes permissions to complete this action on the console or programmatically using the AWS CLI or AWS API.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "ViewOwnUserInfo",
            "Effect": "Allow",
            "Action": [
                "iam:GetUserPolicy",
                "iam:ListGroupsForUser",
                "iam:ListAttachedUserPolicies",
                "iam:ListUserPolicies",
                "iam:GetUser"
            ],
            "Resource": ["arn:aws:iam::*:user/${aws:username}"]
        },
        {
            "Sid": "NavigateInConsole",
            "Effect": "Allow",
            "Action": [
                "iam:GetGroupPolicy",
                "iam:GetPolicyVersion",
                "iam:GetPolicy",
                "iam:ListAttachedGroupPolicies",
                "iam:ListGroupPolicies",
                "iam:ListPolicyVersions",
                "iam:ListPolicies",
                "iam:ListUsers"
            ],
            "Resource": "*"
        }
    ]
}
```

Allow access to third-party model subscriptions

To access the Amazon Bedrock models for the first time, you use the Amazon Bedrock console to subscribe to third-party models. Your IAM user or role that the console user assumes requires permission to access the subscription API operations.

> **Note**
>
> You can't deny access to Mistral AI models, Amazon Titan models, or the Meta Llama 3 Instruct model. You can prevent your users from using inference operations with these models. For more information, see Deny access for inference on specific models.

The following example shows an identity-based policy to allow access to the subscription API operations.

Use a condition key, as in the example, to limits the scope of the policy to a subset of the Amazon Bedrock foundation models in the Marketplace. To see a list of product IDs and which foundation models they correspond to, see the table in Control model access permissions.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "aws-marketplace:Subscribe"
            ],
            "Resource": "*",
            "Condition": {
                "ForAnyValue:StringEquals": {
                    "aws-marketplace:ProductId": [
                        "1d288c71-65f9-489a-a3e2-9c7f4f6e6a85",
                        "cc0bdd50-279a-40d8-829c-4009b77a1fcc",
                        "c468b48a-84df-43a4-8c46-8870630108a7",
                        "99d90be8-b43e-49b7-91e4-752f3866c8c7",
                        "b0eb9475-3a2c-43d1-94d3-56756fd43737",
                        "d0123e8d-50d6-4dba-8a26-3fed4899f388",
                        "a61c46fe-1747-41aa-9af0-2e0ae8a9ce05",
                        "216b69fd-07d5-4c7b-866b-936456d68311",
                        "b7568428-a1ab-46d8-bab3-37def50f6f6a",
                        "38e55671-c3fe-4a44-9783-3584906e7cad",
```

```
                        "prod-ariujvyzvd2qy",
                        "prod-2c2yc2s3guhqy",
                        "prod-6dw3qvchef7zy",
                        "prod-ozonys2hmmpeu",
                        "prod-fm3feywmwerog"

                    ]
                }
            }
        },
        {
            "Effect": "Allow",
            "Action": [
                "aws-marketplace:Unsubscribe",
                "aws-marketplace:ViewSubscriptions"
            ],
            "Resource": "*"
        }
    ]
}
```

Deny access for inference on specific models

The following example shows an identity-based policy that denies access to running inference on a specific model.

```
{
    "Version": "2012-10-17",
    "Statement": {
        "Sid": "DenyInference",
        "Effect": "Deny",
        "Action": [
            "bedrock:InvokeModel",
            "bedrock:InvokeModelWithResponseStream"
         ],
        "Resource": "arn:aws:bedrock:*::foundation-model/model-id"
    }
}
```

Identity-based policy examples for Agents for Amazon Bedrock

Select a topic to see example IAM policies that you can attach to an IAM role to provision permissions for actions in Agents for Amazon Bedrock.

Topics

Required permissions for Agents for Amazon Bedrock

For an IAM identity to use Agents for Amazon Bedrock, you must configure it with the necessary permissions. You can attach the AmazonBedrockFullAccess policy to grant the proper permissions to the role.

To restrict permissions to only actions that are used in Agents for Amazon Bedrock, attach the following identity-based policy to an IAM role:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Agents for Amazon Bedrock permissions",
            "Effect": "Allow",
            "Action": [
                "bedrock:ListFoundationModels",
                "bedrock:GetFoundationModel",
                "bedrock:TagResource",
                "bedrock:UntagResource",
                "bedrock:ListTagsForResource",
                "bedrock:CreateAgent",
                "bedrock:UpdateAgent",
                "bedrock:GetAgent",
                "bedrock:ListAgents",
                "bedrock:DeleteAgent",
                "bedrock:CreateAgentActionGroup",
                "bedrock:UpdateAgentActionGroup",
                "bedrock:GetAgentActionGroup",
                "bedrock:ListAgentActionGroups",
                "bedrock:DeleteAgentActionGroup",
                "bedrock:GetAgentVersion",
                "bedrock:ListAgentVersions",
                "bedrock:DeleteAgentVersion",
                "bedrock:CreateAgentAlias",
                "bedrock:UpdateAgentAlias",
                "bedrock:GetAgentAlias",
```

```
                "bedrock:ListAgentAliases",
                "bedrock:DeleteAgentAlias",
                "bedrock:AssociateAgentKnowledgeBase",
                "bedrock:DisassociateAgentKnowledgeBase",
                "bedrock:GetKnowledgeBase",
                "bedrock:ListKnowledgeBases",
                "bedrock:PrepareAgent",
                "bedrock:InvokeAgent"
            ],
            "Resource": "*"
        }
    ]
}
```

You can further restrict permissions by omitting actions or specifying resources and condition keys. An IAM identity can call API operations on specific resources. For example, the UpdateAgent operation can only be used on agent resources and the InvokeAgent operation can only be used on alias resources. For API operations that aren't used on a specific resource type (such as CreateAgent), specify * as the `Resource`. If you specify an API operation that can't be used on the resource specified in the policy, Amazon Bedrock returns an error.

Allow users to view information about and invoke an agent

The following is a sample policy that you can attach to an IAM role to allow it to view information about or edit an agent with the ID *AGENT12345* and to interact with its alias with the ID *ALIAS12345*. For example, you could attach this policy to a role that you want to only have permissions to troubleshoot an agent and update it.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Get information about and update an agent",
            "Effect": "Allow",
            "Action": [
                "bedrock:GetAgent",
                "bedrock:UpdateAgent"
            ],
            "Resource": "arn:aws:bedrock:aws-region:111122223333:agent/AGENT12345"
        },
        {
            "Sid": "Invoke an agent",
```

```
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeAgent"
            ],
            "Resource": "arn:aws:bedrock:aws-region:111122223333:agent-
alias/AGENT12345/ALIAS12345"
        },
    ]
}
```

Identity-based policy examples for Provisioned Throughput

Select a topic to see example IAM policies that you can attach to an IAM role to provision permissions for actions related to Provisioned Throughput for Amazon Bedrock.

Topics

- Required permissions for Provisioned Throughput
- Allow users to invoke a provisioned model

Required permissions for Provisioned Throughput

For an IAM identity to use Provisioned Throughput, you must configure it with the necessary permissions. You can attach the AmazonBedrockFullAccess policy to grant the proper permissions to the role.

To restrict permissions to only actions that are used in Provisioned Throughput, attach the following identity-based policy to an IAM role:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Provisioned Throughput permissions",
            "Effect": "Allow",
            "Action": [
                "bedrock:GetFoundationModel",
                "bedrock:ListFoundationModels",
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream",
                "bedrock:ListTagsForResource",
```

```
                "bedrock:UntagResource",
                "bedrock:TagResource",
                "bedrock:CreateProvisionedModelThroughput",
                "bedrock:GetProvisionedModelThroughput",
                "bedrock:ListProvisionedModelThroughputs",
                "bedrock:UpdateProvisionedModelThroughput",
                "bedrock:DeleteProvisionedModelThroughput"
            ],
            "Resource": "*"
        }
    ]
}
```

You can further restrict permissions by omitting actions or specifying resources and condition keys. An IAM identity can call API operations on specific resources. For example, the CreateProvisionedModelThroughput operation can only be used on custom model and foundation model resources and the DeleteProvisionedModelThroughput operation can only be used on provisioned model resources. For API operations that aren't used on a specific resource type (such as ListProvisionedModelThroughputs), specify * as the `Resource`. If you specify an API operation that can't be used on the resource specified in the policy, Amazon Bedrock returns an error.

Allow users to invoke a provisioned model

The following is a sample policy that you can attach to an IAM role to allow it to use a provisioned model in model inference. For example, you could attach this policy to a role that you want to only have permissions to use a provisioned model. The role won't be able to manage or see information about the Provisioned Throughput.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Use a Provisioned Throughput for model inference",
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream"
            ],
            "Resource": "arn:aws:bedrock:aws-region:111122223333:provisioned-
model/${my-provisioned-model}"
        }
    ]
```

```
}
```

AWS managed policies for Amazon Bedrock

To add permissions to users, groups, and roles, it's easier to use AWS managed policies than to write policies yourself. It takes time and expertise to create IAM customer managed policies that provide your team with only the permissions they need. To get started quickly, you can use our AWS managed policies. These policies cover common use cases and are available in your AWS account. For more information about AWS managed policies, see AWS managed policies in the *IAM User Guide*.

AWS services maintain and update AWS managed policies. You can't change the permissions in AWS managed policies. Services occasionally add additional permissions to an AWS managed policy to support new features. This type of update affects all identities (users, groups, and roles) where the policy is attached. Services are most likely to update an AWS managed policy when a new feature is launched or when new operations become available. Services do not remove permissions from an AWS managed policy, so policy updates won't break your existing permissions.

Additionally, AWS supports managed policies for job functions that span multiple services. For example, the **ReadOnlyAccess** AWS managed policy provides read-only access to all AWS services and resources. When a service launches a new feature, AWS adds read-only permissions for new operations and resources. For a list and descriptions of job function policies, see AWS managed policies for job functions in the *IAM User Guide*.

AWS managed policy: AmazonBedrockFullAccess

You can attach the `AmazonBedrockFullAccess` policy to your IAM identities.

This policy grants administrative permissions that allow the user permission to create, read, update, and delete Amazon Bedrock resources.

> **Note**
>
> Fine-tuning and model access require extra permissions. See Allow access to third-party model subscriptions and Permissions to access training and validation files and to write output files in S3 for more information.

Permissions details

This policy includes the following permissions:

- `ec2` (Amazon Elastic Compute Cloud) – Allows permissions to describe VPCs, subnets, and security groups.
- `iam` (AWS Identity and Access Management) – Allows principals to pass roles, but only allows IAM roles with "Amazon Bedrock" in them to be passed to the Amazon Bedrock service. The permissions are restricted to `bedrock.amazonaws.com` for Amazon Bedrock operations.
- `kms` (AWS Key Management Service) – Allows principals to describe AWS KMS keys and aliases.
- `bedrock` (Amazon Bedrock) – Allows principals read and write access to all actions in the Amazon Bedrock control plane and runtime service.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "BedrockAll",
            "Effect": "Allow",
            "Action": [
                "bedrock:*"
            ],
            "Resource": "*"
        },
        {
            "Sid": "DescribeKey",
            "Effect": "Allow",
            "Action": [
                "kms:DescribeKey"
            ],
            "Resource": "arn:*:kms:*:::*"
        },
        {
            "Sid": "APIsWithAllResourceAccess",
            "Effect": "Allow",
            "Action": [
                "iam:ListRoles",
                "ec2:DescribeVpcs",
                "ec2:DescribeSubnets",
                "ec2:DescribeSecurityGroups"
```

```
            ],
            "Resource": "*"
        },
        {
            "Sid": "PassRoleToBedrock",
            "Effect": "Allow",
            "Action": [
                "iam:PassRole"
            ],
            "Resource": "arn:aws:iam::*:role/*AmazonBedrock*",
            "Condition": {
                "StringEquals": {
                    "iam:PassedToService": [
                        "bedrock.amazonaws.com"
                    ]
                }
            }
        }
    ]
}
```

AWS managed policy: AmazonBedrockReadOnly

You can attach the `AmazonBedrockReadOnly` policy to your IAM identities.

This policy grants read-only permissions that allow users to view all resources in Amazon Bedrock.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "AmazonBedrockReadOnly",
            "Effect": "Allow",
            "Action": [
                "bedrock:GetFoundationModel",
                "bedrock:ListFoundationModels",
                "bedrock:GetModelInvocationLoggingConfiguration",
                "bedrock:GetProvisionedModelThroughput",
                "bedrock:ListProvisionedModelThroughputs",
                "bedrock:GetModelCustomizationJob",
                "bedrock:ListModelCustomizationJobs",
                "bedrock:ListCustomModels",
                "bedrock:GetCustomModel",
```

```
                "bedrock:ListTagsForResource",
                "bedrock:GetFoundationModelAvailability"
            ],
            "Resource": "*"
        }
    ]
}
```

Amazon Bedrock updates to AWS managed policies

View details about updates to AWS managed policies for Amazon Bedrock since this service began tracking these changes. For automatic alerts about changes to this page, subscribe to the RSS feed on the Document history for the Amazon Bedrock User Guide.

Change	Description	Date
AmazonBedrockFullAccess – New policy	Amazon Bedrock added a new policy to give users permissions to create, read, update, and delete resources.	December 12, 2023
AmazonBedrockReadOnly – New policy	Amazon Bedrock added a new policy to give users read-only permissions for all actions.	December 12, 2023
Amazon Bedrock started tracking changes	Amazon Bedrock started tracking changes for its AWS managed policies.	December 12, 2023

Service roles

Amazon Bedrock uses IAM service roles for the following features to let Amazon Bedrock carry out tasks on your behalf.

The console automatically creates service roles for supported features.

You can also create a custom service role and customize the attached permissions to your specific use-case. If you use the console, you can select this role instead of letting Amazon Bedrock create one for you.

To set up the custom service role, you carry out the following general steps.

1. Create the role by following the steps at Creating a role to delegate permissions to an AWS service.
2. Attach a **trust policy**.
3. Attach the relevant **identity-based permissions**.

Refer to the following links for more information about IAM concepts that are relevant to setting service role permissions.

- AWS service role
- Identity-based policies and resource-based policies
- Using resource-based policies for Lambda
- AWS global condition context keys
- Condition keys for Amazon Bedrock

Select a topic to learn more about service roles for a specific feature.

Topics

Create a service role for model customization

To use a custom role for model customization instead of the one Amazon Bedrock automatically creates, create an IAM role and attach the following permissions by following the steps at Creating a role to delegate permissions to an AWS service.

- Trust relationship
- Permissions to access your training and validation data in S3 and to write your output data to S3
- (Optional) If you encrypt any of the following resources with a KMS key, permissions to decrypt the key (see Encryption of model customization jobs and artifacts)

- A model customization job or the resulting custom model
- The training, validation, or output data for the model customization job

Topics

- Trust relationship
- Permissions to access training and validation files and to write output files in S3

Trust relationship

The following policy allows Amazon Bedrock to assume this role and carry out the model customization job. The following shows an example policy you can use.

You can optionally restrict the scope of the permission for cross-service confused deputy prevention by using one or more global condition context keys with the `Condition` field. For more information, see AWS global condition context keys.

- Set the `aws:SourceAccount` value to your account ID.
- (Optional) Use the `ArnEquals` or `ArnLike` condition to restrict the scope to specific model customization jobs in your account ID.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "account-id"
                },
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:bedrock:us-east-1:account-id:model-
customization-job/*"
                }
            }
```

```
        }
    ]
}
```

Permissions to access training and validation files and to write output files in S3

Attach the following policy to allow the role to access your training and validation data and the bucket to which to write your output data. Replace the values in the `Resource` list with your actual bucket names.

To restrict access to a specific folder in a bucket, add an `s3:prefix` condition key with your folder path. You can follow the **User policy** example in [Example 2: Getting a list of objects in a bucket with a specific prefix](#)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::training-bucket",
                "arn:aws:s3:::training-bucket/*",
                "arn:aws:s3:::validation-bucket",
                "arn:aws:s3:::validation-bucket/*"
            ]
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:PutObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::output-bucket",
                "arn:aws:s3:::output-bucket/*"
            ]
        }
    ]
```

```
}
```

Create a service role for model import

To use a custom role for model import instead of the one Amazon Bedrock automatically creates, create an IAM role and attach the following permissions by following the steps at [Creating a role to delegate permissions to an AWS service](#).

Topics

- [Trust relationship](#)
- [Permissions to access custom model files in Amazon S3](#)

Trust relationship

The following policy allows Amazon Bedrock to assume this role and carry out the model import job. The following shows an example policy you can use.

You can optionally restrict the scope of the permission for [cross-service confused deputy prevention](#) by using one or more global condition context keys with the `Condition` field. For more information, see [AWS global condition context keys.](#)

- Set the `aws:SourceAccount` value to your account ID.
- (Optional) Use the `ArnEquals` or `ArnLike` condition to restrict the scope to specific model import jobs in your account ID.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "1",
            "Effect": "Allow",
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "account-id"
                },
```

```
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:bedrock:us-east-1:account-id:model-
import-job/*"
                }
            }
        }
    ]
}
```

Permissions to access custom model files in Amazon S3

Attach the following policy to allow the role to access to the custom model files in your Amazon S3 bucket. Replace the values in the `Resource` list with your actual bucket names.

To restrict access to a specific folder in a bucket, add an `s3:prefix` condition key with your folder path. You can follow the **User policy** example in [Example 2: Getting a list of objects in a bucket with a specific prefix](#)

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "1",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject",
                "s3:ListBucket"
            ],
            "Resource": [
                "arn:aws:s3:::bucket",
                "arn:aws:s3:::bucket/*"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "account-id"
                }
            }
        }
    ]
}
```

Create a service role for Agents for Amazon Bedrock

To use a custom service role for agents instead of the one Amazon Bedrock automatically creates, create an IAM role with the prefix `AmazonBedrockExecutionRoleForAgents_` and attach the following permissions by following the steps at [Creating a role to delegate permissions to an AWS service](#).

- Trust policy
- A policy containing the following identity-based permissions
  - Access to the Amazon Bedrock base models
  - Access to the Amazon S3 objects containing the OpenAPI schemas for the action groups in your agents
  - Permissions for Amazon Bedrock to query knowledge bases that you want to attach to your agents
  - (Optional) If you encrypt your agent with a KMS key, permissions to decrypt the key (see Encryption of agent resources)

Whether you use a custom role or not, you also need to attach a **resource-based policy** to the Lambda functions for the action groups in your agents to provide permissions for the service role to access the functions. For more information, see Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function.

Topics

- Trust relationship
- Identity-based permissions for the Agents service role.
- Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function

Trust relationship

The following trust policy allows Amazon Bedrock to assume this role and create and manage agents. Replace the *`values`* as necessary. The policy contains optional condition keys (see Condition keys for Amazon Bedrock and AWS global condition context keys) in the `Condition` field that we recommend you use as a security best practice.

Note

As a best practice for security purposes, replace the *** with specific agent IDs after you have created them.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "account-id"
            },
            "ArnLike": {
                "AWS:SourceArn": "arn:aws:bedrock:region:account-id:agent/*"
            }
        }
    }]
}
```

Identity-based permissions for the Agents service role.

Attach the following policy to provide permissions for the service role, replacing *`values`* as necessary. The policy contains the following statements. Omit a statement if it isn't applicable to your use-case. The policy contains optional condition keys (see Condition keys for Amazon Bedrock and AWS global condition context keys) in the `Condition` field that we recommend you use as a security best practice.

Note

If you encrypt your agent with a customer-managed KMS key, refer to Encryption of agent resources for further permissions you need to add.

- Permissions to use Amazon Bedrock foundation models to run model inference on prompts used in your agent's orchestration.
- Permissions to access your agent's action group API schemas in Amazon S3. Omit this statement if your agent has no action groups.
- Permissions to access knowledge bases associated with your agent. Omit this statement if your agent has no associated knowledge bases.
- Permissions to access a third-party (Pinecone or Redis Enterprise Cloud) knowledge base associated with your agent. Omit this statement if your knowledge base is first-party (Amazon OpenSearch Serverless or Amazon Aurora) or if your agent has no associated knowledge bases.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Sid": "Allow model invocation for orchestration",
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": [
                "arn:aws:bedrock:region::foundation-model/anthropic.claude-v2",
                "arn:aws:bedrock:region::foundation-model/anthropic.claude-v2:1",
                "arn:aws:bedrock:region::foundation-model/anthropic.claude-instant-v1"
            ]
        },
        {
            "Sid": "Allow access to action group API schemas in S3",
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
            "Resource": [
                "arn:aws:s3:::bucket/path/to/schema"
            ],
            "Condition": {
                "StringEquals": {
                    "aws:ResourceAccount": "account-id"
                }
            }
        },
```

```
        {
            "Sid": "Query associated knowledge bases",
            "Effect": "Allow",
            "Action": [
                "bedrock:Retrieve",
                "bedrock:RetrieveAndGenerate"
            ],
            "Resource": [
                "arn:aws:bedrock:region:account-id:knowledge-base/knowledge-base-id"
            ]
        },
        {
            "Sid": "Associate a third-party knowledge base with your agent",
            "Effect": "Allow",
            "Action": [
                "bedrock:AssociateThirdPartyKnowledgeBase",
            ],
            "Resource": "arn:aws:bedrock:region:account-id:knowledge-base/knowledge-
base-id",
            "Condition": {
                "StringEquals" : {
                    "bedrock:ThirdPartyKnowledgeBaseCredentialsSecretArn":
 "arn:aws:kms:region:account-id:key/key-id"
                }
            }
        }
    ]
}
```

Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function

Follow the steps at Using resource-based policies for Lambda and attach the following resource-based policy to a Lambda function to allow Amazon Bedrock to access the Lambda function for your agent's action groups, replacing the *values* as necessary. The policy contains optional condition keys (see Condition keys for Amazon Bedrock and AWS global condition context keys) in the `Condition` field that we recommend you use as a security best practice.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "Allow Amazon Bedrock to access action group Lambda function",
        "Effect": "Allow",
        "Principal": {
```

```
            "Service": "bedrock.amazonaws.com"
        },
        "Action": "lambda:InvokeFunction",
        "Resource":  "arn:aws:lambda:region:account-id:function:function-name",
        "Condition": {
            "StringEquals": {
                "AWS:SourceAccount": "account-id"
            },
            "ArnLike": {
                "AWS:SourceArn": "arn:aws:bedrock:region:account-id:agent/agent-id"
            }
        }
    }]
}
```

Create a service role for Knowledge bases for Amazon Bedrock

To use a custom role for knowledge base instead of the one Amazon Bedrock automatically creates, create an IAM role and attach the following permissions by following the steps at [Creating a role to delegate permissions to an AWS service](). You can use the same role across your knowledge bases.

- Trust relationship
- Access to the Amazon Bedrock base models
- Access to the Amazon S3 objects containing your data sources
- (If you create a vector database in Amazon OpenSearch Service) Access to your OpenSearch Service collection
- (If you create a vector database in Amazon Aurora)
- (If you create a vector database in Pinecone or Redis Enterprise Cloud) Permissions for AWS Secrets Manager to authenticate your Pinecone or Redis Enterprise Cloud account
- (Optional) If you encrypt any of the following resources with a KMS key, permissions to decrypt the key (see [Encryption of knowledge base resources]()).
  - Your knowledge base
  - Data sources for your knowledge base
  - Your vector database in Amazon OpenSearch Service
  - The secret for your third-party vector database in AWS Secrets Manager
  - A data ingestion job

Topics

Trust relationship

The following policy allows Amazon Bedrock to assume this role and create and manage knowledge bases. The following shows an example policy you can use. You can restrict the scope of the permission by using one or more global condition context keys. For more information, see AWS global condition context keys. Set the `aws:SourceAccount` value to your account ID. Use the `ArnEquals` or `ArnLike` condition to restrict the scope to specific knowledge bases.

> **Note**
>
> As a best practice for security purposes, replace the `*` with specific knowledge base IDs after you have created them.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Principal": {
            "Service": "bedrock.amazonaws.com"
        },
        "Action": "sts:AssumeRole",
```

```
        "Condition": {
            "StringEquals": {
                "aws:SourceAccount": "account-id"
            },
            "ArnLike": {
                "AWS:SourceArn": "arn:aws:bedrock:region:account-id:knowledge-base/*"
            }
        }
    }]
}
```

Permissions to access Amazon Bedrock models

Attach the following policy to provide permissions for the role to use Amazon Bedrock models to embed your source data.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "bedrock:ListFoundationModels",
                "bedrock:ListCustomModels"
            ],
            "Resource": "*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "bedrock:InvokeModel"
            ],
            "Resource": [
                "arn:aws:bedrock:region::foundation-model/amazon.titan-embed-text-v1",
                "arn:aws:bedrock:region::foundation-model/cohere.embed-english-v3",
                "arn:aws:bedrock:region::foundation-model/cohere.embed-multilingual-v3"
            ]
        }
    ]
}
```

Permissions to access your data sources in Amazon S3

Attach the following policy to provide permissions for the role to access the Amazon S3 URIs containing the data source files for your knowledge base. In the `Resource` field, provide an Amazon S3 object containing the data sources or add the URI of each data source to the list.

If you encrypted these data sources with a AWS KMS key, attach permissions to decrypt the key to the role by following the steps at [Permissions to decrypt your AWS KMS key for your data sources in Amazon S3](#).

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "s3:GetObject",
            "s3:ListBucket"
        ],
        "Resource": [
            "arn:aws:s3:::bucket/path/to/folder",
            "arn:aws:s3:::bucket/path/to/folder/*"
        ],
        "Condition": {
            "StringEquals": {
                "aws:PrincipalAccount": "account-id"
            }
        }
    }]
}
```

(Optional) Permissions to access your vector database in Amazon OpenSearch Service

If you created a vector database in Amazon OpenSearch Service for your knowledge base, attach the following policy to your Knowledge bases for Amazon Bedrock service role to allow access to the collection. Replace *`region`* and *`account-id`* with the region and account ID to which the database belongs. Input the ID of your Amazon OpenSearch Service collection in *`collection-id`*. You can allow access to multiple collections by adding them to the `Resource` list.

```
{
    "Version": "2012-10-17",
    "Statement": [{
```

```
        "Effect": "Allow",
        "Action": [
            "aoss:APIAccessAll"
        ],
        "Resource": [
            "arn:aws:aoss:region:account-id:collection/collection-id"
        ]
    }]
}
```

(Optional) Permissions to access your Amazon Aurora database cluster

If you created a database (DB) cluster in Amazon Aurora for your knowledge base, attach the following policy to your Knowledge bases for Amazon Bedrock service role to allow access to the DB cluster and to provide read and write permissions on it. Replace *region* and *account-id* with the region and account ID to which the DB cluster belongs. Input the ID of your Amazon Aurora database cluster in *db-cluster-id*. You can allow access to multiple DB clusters by adding them to the `Resource` list.

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
        "Sid": "RdsDescribeStatementID",
        "Effect": "Allow",
        "Action": [
            "rds:DescribeDBClusters"
        ],
        "Resource": [
            "arn:aws:rds:region:account-id:cluster:db-cluster-id"
        ]
    },
    {
        "Sid": "DataAPIStatementID",
        "Effect": "Allow",
        "Action": [
            "rds-data:BatchExecuteStatement",
            "rds-data:ExecuteStatement"
        ],
        "Resource": [
            "arn:aws:rds:region:account-id:cluster:db-cluster-id"
        ]
    }]
```

```
}
```

(Optional) Permissions to access a vector database configured with an AWS Secrets Manager secret

If your vector database is configured with an AWS Secrets Manager secret, attach the following policy to your Knowledge bases for Amazon Bedrock service role to allow AWS Secrets Manager to authenticate your account to access the database. Replace *region* and *account-id* with the region and account ID to which the database belongs. Replace *secret-id* with the ID of your secret.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Effect": "Allow",
        "Action": [
            "secretsmanager:GetSecretValue"
        ],
        "Resource": [
            "arn:aws:secretsmanager:region:account-id:secret:secret-id"
        ]
    }]
}
```

If you encrypted your secret with a AWS KMS key, attach permissions to decrypt the key to the role by following the steps at [Permissions to decrypt an AWS Secrets Manager secret for the vector store containing your knowledge base](https://docs.aws.amazon.com/bedrock/latest/userguide/encryption-kb.html).

(Optional) Permissions for AWS to manage a AWS KMS key for transient data storage during data ingestion

To allow the creation of a AWS KMS key for transient data storage in the process of ingesting your data source, attach the following policy to your Knowledge bases for Amazon Bedrock service role. Replace the *region*, *account-id*, and *key-id* with the appropriate values.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
          "Effect": "Allow",
          "Action": [
```

```
                "kms:GenerateDataKey",
                "kms:Decrypt"
            ],
            "Resource": [
                "arn:aws:kms:region:account-id:key/key-id"
            ]
        }
    ]
}
```

Permissions to chat with your document

Attach the following policy to provide permissions for the role to use Amazon Bedrock models to chat with your document:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
   "Effect": "Allow",
   "Action": [
    "bedrock:RetrieveAndGenerate"
   ],
   "Resource": "*"
  }
    ]
}
```

If you only want to grant a user access to chat with your document (and not to `RetrieveAndGenerate` on all Knowledge Bases), use the following policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
   "Effect": "Allow",
   "Action": [
    "bedrock:RetrieveAndGenerate"
   ],
   "Resource": "*"
  },
```

```
        {
    "Effect": "Deny",
    "Action": [
     "bedrock:Retrieve"
    ],
    "Resource": "*"
   }
     ]
}
```

If you want both chat with your document and use `RetrieveAndGenerate` on a specific Knowledge Base, provide *insert KB ARN*, and use the following policy:

```
{
     "Version": "2012-10-17",
     "Statement": [
         {
    "Effect": "Allow",
    "Action": [
     "bedrock:RetrieveAndGenerate"
    ],
    "Resource": "*"
   },
         {
    "Effect": "Allow",
    "Action": [
     "bedrock:Retrieve"
    ],
    "Resource": insert KB ARN
   }
     ]
}
```

(Optional) Permissions for AWS to manage a data sources from another user's AWS account.

To allow the access to another user's AWS account, you must create a role that allows cross-account access to a Amazon S3 bucket in another user's account. Replace the *bucketName*, *bucketOwnerAccountId*, and *bucketNameAndPrefix* with the appropriate values.

Permissions Required on Knowledge Base role

The knowledge base role that is provided during knowledge base creation `createKnowledgeBase` requires the following Amazon S3 permissions.

```
{
    "Version": "2012-10-17",
    "Statement": [{
        "Sid": "S3ListBucketStatement",
        "Effect": "Allow",
        "Action": [
            "s3:ListBucket"
        ],
        "Resource": [
            "arn:aws:s3:::bucketName"
        ],
        "Condition": {
            "StringEquals": {
                "aws:ResourceAccount": "bucketOwnerAccountId"
            }
        }
    },{
        "Sid": "S3GetObjectStatement",
        "Effect": "Allow",
        "Action": [
            "s3:GetObject"
        ],
        "Resource": [
            "arn:aws:s3:::bucketNameAndPrefix/*"
        ],
        "Condition": {
            "StringEquals": {
                "aws:ResourceAccount": "bucketOwnerAccountId"
            }
        }
    }
```

If the Amazon S3 bucket is encrypted using a AWS KMS key, the following also needs to be added to the knowledge base role. Replace the *`bucketOwnerAccountId`* and *`region`* with the appropriate values.

```
{
        "Sid": "KmsDecryptStatement",
```

```
        "Effect": "Allow",
        "Action": [
            "kms:Decrypt"
        ],
        "Resource": [
            "arn:aws:kms:region:bucketOwnerAccountId:key/keyId"
        ],
        "Condition": {
        "StringEquals": {
            "kms:ViaService": [
                "s3.region.amazonaws.com"
            ]
        }
        }
    }
```

Permissions required on a cross-account Amazon S3 bucket policy

The bucket in the other account requires the following Amazon S3 bucket policy. Replace the *kbRoleArn*, *bucketName*, and *bucketNameAndPrefix* with the appropriate values.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Sid": "Example ListBucket permissions",
         "Effect": "Allow",
         "Principal": {
            "AWS": "kbRoleArn"
         },
         "Action": [
            "s3:ListBucket"
         ],
         "Resource": [
            "arn:aws:s3:::bucketName"
         ]
      },
      {
         "Sid": "Example GetObject permissions",
         "Effect": "Allow",
         "Principal": {
            "AWS": "kbRoleArn"
```

```
            },
            "Action": [
               "s3:GetObject"
            ],
            "Resource": [
               "arn:aws:s3:::bucketNameAndPrefix/*"
            ]
         }
      ]
}
```

Permissions required on cross-account AWS KMS key policy

If the cross-account Amazon S3 bucket is encrypted using a AWS KMS key in that account, the policy of the AWS KMS key requires the following policy. Replace the *kbRoleArn* and *kmsKeyArn* with the appropriate values.

```
{
    "Sid": "Example policy",
    "Effect": "Allow",
    "Principal": {
        "AWS": [
            "kbRoleArn"
        ]
    },
    "Action": [
        "kms:Decrypt"
    ],
    "Resource": "kmsKeyArn"
}
```

Troubleshooting Amazon Bedrock identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with Amazon Bedrock and IAM.

Topics

- I want to allow people outside of my AWS account to access my Amazon Bedrock resources

I am not authorized to perform an action in Amazon Bedrock

If you receive an error that you're not authorized to perform an action, your policies must be updated to allow you to perform the action.

The following example error occurs when the `mateojackson` IAM user tries to use the console to view details about a fictional *`my-example-widget`* resource but doesn't have the fictional `bedrock:`*`GetWidget`* permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 bedrock:GetWidget on resource: my-example-widget
```

In this case, the policy for the `mateojackson` user must be updated to allow access to the *`my-example-widget`* resource by using the `bedrock:`*`GetWidget`* action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I am not authorized to perform iam:PassRole

If you receive an error that you're not authorized to perform the `iam:PassRole` action, your policies must be updated to allow you to pass a role to Amazon Bedrock.

Some AWS services allow you to pass an existing role to that service instead of creating a new service role or service-linked role. To do this, you must have permissions to pass the role to the service.

The following example error occurs when an IAM user named `marymajor` tries to use the console to perform an action in Amazon Bedrock. However, the action requires the service to have permissions that are granted by a service role. Mary does not have permissions to pass the role to the service.

```
User: arn:aws:iam::123456789012:user/marymajor is not authorized to perform:
 iam:PassRole
```

In this case, Mary's policies must be updated to allow her to perform the `iam:PassRole` action.

If you need help, contact your AWS administrator. Your administrator is the person who provided you with your sign-in credentials.

I want to allow people outside of my AWS account to access my Amazon Bedrock resources

You can create a role that users in other accounts or people outside of your organization can use to access your resources. You can specify who is trusted to assume the role. For services that support resource-based policies or access control lists (ACLs), you can use those policies to grant people access to your resources.

To learn more, consult the following:

- To learn whether Amazon Bedrock supports these features, see How Amazon Bedrock works with IAM.
- To learn how to provide access to your resources across AWS accounts that you own, see Providing access to an IAM user in another AWS account that you own in the *IAM User Guide*.
- To learn how to provide access to your resources to third-party AWS accounts, see Providing access to AWS accounts owned by third parties in the *IAM User Guide*.
- To learn how to provide access through identity federation, see Providing access to externally authenticated users (identity federation) in the *IAM User Guide*.
- To learn the difference between using roles and resource-based policies for cross-account access, see How IAM roles differ from resource-based policies in the *IAM User Guide*.

Compliance validation for Amazon Bedrock

To learn whether an AWS service is within the scope of specific compliance programs, see AWS services in Scope by Compliance Program and choose the compliance program that you are interested in. For general information, see AWS Compliance Programs.

You can download third-party audit reports using AWS Artifact. For more information, see Downloading Reports in AWS Artifact.

Your compliance responsibility when using AWS services is determined by the sensitivity of your data, your company's compliance objectives, and applicable laws and regulations. AWS provides the following resources to help with compliance:

- Security and Compliance Quick Start Guides – These deployment guides discuss architectural considerations and provide steps for deploying baseline environments on AWS that are security and compliance focused.
- Architecting for HIPAA Security and Compliance on Amazon Web Services – This whitepaper describes how companies can use AWS to create HIPAA-eligible applications.

> **Note**
>
> Not all AWS services are HIPAA eligible. For more information, see the HIPAA Eligible Services Reference.

- AWS Compliance Resources – This collection of workbooks and guides might apply to your industry and location.
- AWS Customer Compliance Guides – Understand the shared responsibility model through the lens of compliance. The guides summarize the best practices for securing AWS services and map the guidance to security controls across multiple frameworks (including National Institute of Standards and Technology (NIST), Payment Card Industry Security Standards Council (PCI), and International Organization for Standardization (ISO)).
- Evaluating Resources with Rules in the *AWS Config Developer Guide* – The AWS Config service assesses how well your resource configurations comply with internal practices, industry guidelines, and regulations.
- AWS Security Hub – This AWS service provides a comprehensive view of your security state within AWS. Security Hub uses security controls to evaluate your AWS resources and to check your compliance against security industry standards and best practices. For a list of supported services and controls, see Security Hub controls reference.
- AWS Audit Manager – This AWS service helps you continuously audit your AWS usage to simplify how you manage risk and compliance with regulations and industry standards.

Incident response in Amazon Bedrock

Security is the highest priority at AWS. As part of the AWS Cloud shared responsibility model, AWS manages a data center, network, and software architecture that meets the requirements of the most security-sensitive organizations. AWS is responsible for any incident response with respect to the Amazon Bedrock service itself. Also, as an AWS customer, you share a responsibility for maintaining security in the cloud. This means that you control the security you choose to implement from the AWS tools and features you have access to. In addition, you're responsible for incident response on your side of the shared responsibility model.

By establishing a security baseline that meets the objectives for your applications running in the cloud, you're able to detect deviations that you can respond to. To help you understand the impact that incident response and your choices have on your corporate goals, we encourage you to review the following resources:

- AWS Security Incident Response Guide
- AWS Best Practices for Security, Identity, and Compliance
- Security Perspective of the AWS Cloud Adoption Framework (CAF) whitepaper

Resilience in Amazon Bedrock

The AWS global infrastructure is built around AWS Regions and Availability Zones. AWS Regions provide multiple physically separated and isolated Availability Zones, which are connected with low-latency, high-throughput, and highly redundant networking. With Availability Zones, you can design and operate applications and databases that automatically fail over between zones without interruption. Availability Zones are more highly available, fault tolerant, and scalable than traditional single or multiple data center infrastructures.

For more information about AWS Regions and Availability Zones, see AWS Global Infrastructure.

Infrastructure security in Amazon Bedrock

As a managed service, Amazon Bedrock is protected by the AWS global network security. For information about AWS security services and how AWS protects infrastructure, see AWS Cloud Security. To design your AWS environment using the best practices for infrastructure security, see Infrastructure Protection in *Security Pillar AWS Well‐Architected Framework*.

You use AWS published API calls to access Amazon Bedrock through the network. Clients must support the following:

- Transport Layer Security (TLS). We require TLS 1.2 and recommend TLS 1.3.
- Cipher suites with perfect forward secrecy (PFS) such as DHE (Ephemeral Diffie-Hellman) or ECDHE (Elliptic Curve Ephemeral Diffie-Hellman). Most modern systems such as Java 7 and later support these modes.

Additionally, requests must be signed by using an access key ID and a secret access key that is associated with an IAM principal. Or you can use the [AWS Security Token Service](#) (AWS STS) to generate temporary security credentials to sign requests.

Cross-service confused deputy prevention

The confused deputy problem is a security issue where an entity that doesn't have permission to perform an action can coerce a more-privileged entity to perform the action. In AWS, cross-service impersonation can result in the confused deputy problem. Cross-service impersonation can occur when one service (the *calling service*) calls another service (the *called service*). The calling service can be manipulated to use its permissions to act on another customer's resources in a way it should not otherwise have permission to access. To prevent this, AWS provides tools that help you protect your data for all services with service principals that have been given access to resources in your account.

We recommend using the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in resource policies to limit the permissions that Amazon Bedrock gives another service to the resource. Use `aws:SourceArn` if you want only one resource to be associated with the cross-service access. Use `aws:SourceAccount` if you want to allow any resource in that account to be associated with the cross-service use.

The most effective way to protect against the confused deputy problem is to use the `aws:SourceArn` global condition context key with the full ARN of the resource. If you don't know the full ARN of the resource or if you are specifying multiple resources, use the `aws:SourceArn` global context condition key with wildcard characters (*) for the unknown portions of the ARN. For example, `arn:aws:bedrock:*:123456789012:*`.

If the `aws:SourceArn` value does not contain the account ID, such as an Amazon S3 bucket ARN, you must use both global condition context keys to limit permissions.

The value of `aws:SourceArn` must be ResourceDescription.

The following example shows how you can use the `aws:SourceArn` and `aws:SourceAccount` global condition context keys in Bedrock to prevent the confused deputy problem.

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
```

```
            "Principal": {
                "Service": "bedrock.amazonaws.com"
            },
            "Action": "sts:AssumeRole",
            "Condition": {
                "StringEquals": {
                    "aws:SourceAccount": "111122223333"
                },
                "ArnEquals": {
                    "aws:SourceArn": "arn:aws:bedrock:us-east-1:111122223333:model-
customization-job/*"
                }
            }
        }
    ]
}
```

Configuration and vulnerability analysis in Amazon Bedrock

Configuration and IT controls are a shared responsibility between AWS and you, our customer. For more information, see the AWS shared responsibility model.

Use interface VPC endpoints (AWS PrivateLink)

You can use AWS PrivateLink to create a private connection between your VPC and Amazon Bedrock. You can access Amazon Bedrock as if it were in your VPC, without the use of an internet gateway, NAT device, VPN connection, or AWS Direct Connect connection. Instances in your VPC don't need public IP addresses to access Amazon Bedrock.

You establish this private connection by creating an *interface endpoint*, powered by AWS PrivateLink. We create an endpoint network interface in each subnet that you enable for the interface endpoint. These are requester-managed network interfaces that serve as the entry point for traffic destined for Amazon Bedrock.

For more information, see Access AWS services through AWS PrivateLink in the *AWS PrivateLink Guide*.

Considerations for Amazon Bedrock VPC endpoints

Before you set up an interface endpoint for Amazon Bedrock, review Considerations in the *AWS PrivateLink Guide*.

Amazon Bedrock supports making the following API calls through VPC endpoints.

Category	Endpoint prefix
Amazon Bedrock Control Plane API actions	`bedrock`
Amazon Bedrock Runtime API actions	`bedrock-runtime`
Agents for Amazon Bedrock Build-time API actions	`bedrock-agent`
Agents for Amazon Bedrock Runtime API actions	`bedrock-agent-runtime`

Availability Zones

Amazon Bedrock and Agents for Amazon Bedrock endpoints are available in multiple Availability Zones.

Create an interface endpoint for Amazon Bedrock

You can create an interface endpoint for Amazon Bedrock using either the Amazon VPC console or the AWS Command Line Interface (AWS CLI). For more information, see Create an interface endpoint in the *AWS PrivateLink Guide*.

Create an interface endpoint for Amazon Bedrock using any of the following service names:

- `com.amazonaws.`*`region`*`.bedrock`
- `com.amazonaws.`*`region`*`.bedrock-runtime`
- `com.amazonaws.`*`region`*`.bedrock-agent`
- `com.amazonaws.`*`region`*`.bedrock-agent-runtime`

After you create the endpoint, you have the option to enable a private DNS hostname. Enable this setting by selecting Enable Private DNS Name in the VPC console when you create the VPC endpoint.

If you enable private DNS for the interface endpoint, you can make API requests to Amazon Bedrock using its default Regional DNS name. The following examples show the format of the default Regional DNS names.

- `bedrock.region.amazonaws.com`
- `bedrock-runtime.region.amazonaws.com`
- `bedrock-agent.region.amazonaws.com`
- `bedrock-agent-runtime.region.amazonaws.com`

Create an endpoint policy for your interface endpoint

An endpoint policy is an IAM resource that you can attach to an interface endpoint. The default endpoint policy allows full access to Amazon Bedrock through the interface endpoint. To control the access allowed to Amazon Bedrock from your VPC, attach a custom endpoint policy to the interface endpoint.

An endpoint policy specifies the following information:

- The principals that can perform actions (AWS accounts, IAM users, and IAM roles).
- The actions that can be performed.
- The resources on which the actions can be performed.

For more information, see Control access to services using endpoint policies in the *AWS PrivateLink Guide*.

Example: VPC endpoint policy for Amazon Bedrock actions

The following is an example of a custom endpoint policy. When you attach this resource-based policy to your interface endpoint, it grants access to the listed Amazon Bedrock actions for all principals on all resources.

```
{
   "Version": "2012-10-17",
   "Statement": [
      {
         "Principal": "*",
         "Effect": "Allow",
         "Action": [
```

```
                "bedrock:InvokeModel",
                "bedrock:InvokeModelWithResponseStream"
            ],
            "Resource":"*"
        }
    ]
}
```

Monitor Amazon Bedrock

You can monitor Amazon Bedrock with Amazon CloudWatch and with Amazon EventBridge.

Topics

Model invocation logging

Model invocation logging can be used to collect invocation logs, model input data, and model output data for all invocations in your AWS account used in Amazon Bedrock. By default, logging is disabled.

With invocation logging, you can collect the full request data, response data, and metadata associated with all calls performed in your account. Logging can be configured to provide the destination resources where the log data will be published. Supported destinations include Amazon CloudWatch Logs and Amazon Simple Storage Service (Amazon S3). Only destinations from the same account and region are supported.

Before you can enable invocation logging, you need to set up an Amazon S3 or CloudWatch Logs destination. You can enable invocation logging through either the console or the API.

Topics

Set up an Amazon S3 destination

You can set up an S3 destination for logging in Amazon Bedrock with these steps:

1. Create an S3 bucket where the logs will be delivered.
2. Add a bucket policy to it like the one below (Replace values for *accountId*, *region*, *bucketName*, and optionally *prefix*):

> **Note**
>
> A bucket policy is automatically attached to the bucket on your behalf when you configure logging with the permissions `S3:GetBucketPolicy` and `S3:PutBucketPolicy`.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Sid": "AmazonBedrockLogsWrite",
      "Effect": "Allow",
      "Principal": {
        "Service": "bedrock.amazonaws.com"
      },
      "Action": [
        "s3:PutObject"
      ],
      "Resource": [
        "arn:aws:s3:::bucketName/prefix/AWSLogs/accountId/
BedrockModelInvocationLogs/*"
      ],
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "accountId"
        },
        "ArnLike": {
            "aws:SourceArn": "arn:aws:bedrock:region:accountId:*"
        }
      }
    }
  ]
}
```

3. (Optional) If configuring SSE-KMS on the bucket, add the below policy on the KMS key:

```
{
    "Effect": "Allow",
    "Principal": {
        "Service": "bedrock.amazonaws.com"
    },
    "Action": "kms:GenerateDataKey",
    "Resource": "*",
    "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "accountId"
        },
        "ArnLike": {
           "aws:SourceArn": "arn:aws:bedrock:region:accountId:*"
        }
    }
}
```

For more information on S3 SSE-KMS configurations, see Specifying KMS Encryption.

> **Note**
>
> The bucket ACL must be disabled in order for the bucket policy to take effect. For more information, see Disabling ACLs for all new buckets and enforcing Object Ownership.

Set up CloudWatch Logs destination

You can set up a Amazon CloudWatch Logs destination for logging in Amazon Bedrock with the following steps:

1. Create a CloudWatch log group where the logs will be published.
2. Create an IAM role with the following permissions for CloudWatch Logs.

   Trusted entity:

   ```
   {
     "Version": "2012-10-17",
     "Statement": [
       {
   ```

```
      "Effect": "Allow",
      "Principal": {
        "Service": "bedrock.amazonaws.com"
      },
      "Action": "sts:AssumeRole",
      "Condition": {
        "StringEquals": {
          "aws:SourceAccount": "accountId"
        },
        "ArnLike": {
          "aws:SourceArn": "arn:aws:bedrock:region:accountId:*"
        }
      }
    }
  ]
}
```

Role policy:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:CreateLogStream",
                "logs:PutLogEvents"
            ],
            "Resource": "arn:aws:logs:region:accountId:log-group:logGroupName:log-
stream:aws/bedrock/modelinvocations"
         }
    ]
}
```

For more information on setting up SSE for CloudWatch Logs, see [Encrypt log data in CloudWatch Logs using AWS Key Management Service](#).

Using the console

To enable model invocation logging, drag the slider button next to the **Logging** toggle switch in the **Settings** page. Additional configuration settings for logging will appear on the panel.

Choose which data requests and responses you want to publish to the logs. You can choose any combination of the following output options:

- Text
- Image
- Embedding

Choose where to publish the logs:

- Amazon S3 only
- CloudWatch Logs only
- Both Amazon S3 and CloudWatch Logs

Amazon S3 and CloudWatch Logs destinations are supported for invocation logs, and small input and output data. For large input and output data or binary image outputs, only Amazon S3 is supported. The following details summarize how the data will be represented in the target location.

- **S3 destination** — Gzipped JSON files, each containing a batch of invocation log records, are delivered to the specified S3 bucket. Similar to a CloudWatch Logs event, each record will contain the invocation metadata, and input and output JSON bodies of up to 100 KB in size. Binary data or JSON bodies larger than 100 KB will be uploaded as individual objects in the specified Amazon S3 bucket under the data prefix. The data can be queried using Amazon S3 Select and Amazon Athena, and can be catalogued for ETL using AWS Glue. The data can be loaded into OpenSearch service, or be processed by any Amazon EventBridge targets.
- **CloudWatch Logs destination** — JSON invocation log events are delivered to a specified log group in CloudWatch Logs. The log event contains the invocation metadata, and input and output JSON bodies of up to 100 KB in size. If an Amazon S3 location for large data delivery is provided, binary data or JSON bodies larger than 100 KB will be uploaded to the Amazon S3 bucket under the data prefix instead. data can be queried using CloudWatch Logs Insights, and can be further streamed to various services in real-time using CloudWatch Logs.

Using APIs with invocation logging

Model invocation logging can be configured using the following APIs:

- `PutModelInvocationLoggingConfiguration`
- `GetModelInvocationLoggingConfiguration`
- `DeleteModelInvocationLoggingConfiguration`

For more information on how to use APIs with invocation logging, see the Bedrock API Guide.

Monitor Amazon Bedrock with Amazon CloudWatch

You can monitor Amazon Bedrock using Amazon CloudWatch, which collects raw data and processes it into readable, near real-time metrics. You can graph the metrics using the CloudWatch console. You can also set alarms that watch for certain thresholds, and send notifications or take actions when values exceed those thresholds.

For more information, see What is Amazon CloudWatch in the *Amazon CloudWatch User Guide*.

Topics

Runtime metrics

The following table describes runtime metrics provided by Amazon Bedrock.

Metric name	Unit	Description
Invocations	SampleCount	Number of requests to the InvokeModel or InvokeModelWithResponseStream API operations.

Metric name	Unit	Description
InvocationLatency	MilliSeconds	Latency of the invocations.
InvocationClientErrors	SampleCount	Number of invocations that result in client-side errors.
InvocationServerErrors	SampleCount	Number of invocations that result in AWS server-side errors.
InvocationThrottles	SampleCount	Number of invocations that the system throttled.
InputTokenCount	SampleCount	Number of tokens of text input.
LegacyModelInvocations	SampleCount	Number of invocations using Legacy models
OutputTokenCount	SampleCount	Number of tokens of text output.
OutputImageCount	SampleCount	Number of output images.

Logging CloudWatch metrics

For each delivery success or failure attempt, the following Amazon CloudWatch metrics are emitted under the namespace `AWS/Bedrock`, and `Across all model IDs` dimension:

- `ModelInvocationLogsCloudWatchDeliverySuccess`
- `ModelInvocationLogsCloudWatchDeliveryFailure`
- `ModelInvocationLogsS3DeliverySuccess`
- `ModelInvocationLogsS3DeliveryFailure`
- `ModelInvocationLargeDataS3DeliverySuccess`
- `ModelInvocationLargeDataS3DeliveryFailure`

If logs fail to deliver due to permission misconfiguration or transient failures, the delivery is retried periodically for up to 24 hours.

Use CloudWatch metrics for Amazon Bedrock

To retrieve metrics for your Amazon Bedrock operations, you specify the following information:

- The metric dimension. A *dimension* is a set of name-value pairs that you use to identify a metric. Amazon Bedrock supports the following dimensions:
  - `ModelId` – all metrics
  - `ModelId + ImageSize + BucketedStepSize` – OutputImageCount
- The metric name, such as `InvocationClientErrors`.

You can get metrics for Amazon Bedrock with the AWS Management Console, the AWS CLI, or the CloudWatch API. You can use the CloudWatch API through one of the AWS Software Development Kits (SDKs) or the CloudWatch API tools.

You must have the appropriate CloudWatch permissions to monitor Amazon Bedrock with CloudWatch For more information, see [Authentication and Access Control for Amazon CloudWatch] in the *Amazon CloudWatch User Guide*.

View Amazon Bedrock metrics

View Amazon Bedrock metrics in the CloudWatch console.

To view metrics (CloudWatch console)

1. Sign in to the AWS Management Console and open the CloudWatch console at https://console.aws.amazon.com/cloudwatch/.
2. Choose **Metrics**, choose **All Metrics**, and then search for **ModelId**.

Monitor Amazon Bedrock events in Amazon EventBridge

You can use Amazon EventBridge to monitor status change events in Amazon Bedrock. With Amazon EventBridge, you can configure Amazon SageMaker to respond automatically to a model customization job status change in Amazon Bedrock. Events from Amazon Bedrock are delivered to Amazon EventBridge in near real time. You can write simple rules to automate actions when an event matches a rule. If you use Amazon EventBridge with Amazon Bedrock, you can:

- Publish notifications whenever there is a state change event in the model customization you have triggered, whether you add new asynchronous workflows in the future. The event published should give you enough information to react to events in downstream workflows.
- Deliver job status updates without invoking the GetModelCustomizationJob API, which can mean handling API rate limit issues, API updates, and reduction in additional compute resources.

There is no cost to receive AWS events from Amazon EventBridge. For more information about, Amazon EventBridge, see [Amazon EventBridge]

Note

- Amazon Bedrock emits events on a best-effort basis. Events are delivered to Amazon EventBridge in near real time. With Amazon EventBridge, you can create rules that trigger programmatic actions in response to an event. For example, you can configure a rule that invokes an SNS topic to send an email notification or invokes a function to take some action. For more information, see the *Amazon EventBridge User Guide*.
- Amazon Bedrock creates a new event every time there is a state change in a model customization job that you trigger and make best-effort delivery of such event.

Topics

- How it works
- EventBridge schema
- Rules and targets
- Create a rule to handle Amazon Bedrock events

How it works

To receive events from Amazon Bedrock, you need to create rules and targets to match, receive, and handle state change data through Amazon EventBridge. Amazon EventBridge is a serverless event bus that ingests change state events from AWS services, SaaS partners, and customer applications. It processes events based on rules or patterns that you create, and routes these events to one or more “targets” that you choose, such as AWS Lambda, Amazon Simple Queue Service, and Amazon Simple Notification Service.

Amazon Bedrock publishes your events via Amazon EventBridge whenever there is a change in the state of a model customization job. In each case, a new event is created and sent to Amazon EventBridge, which then sends the event to your default event-bus. The event shows which customization job's state has changed, and the current state of the job. When Amazon EventBridge receives an event that matches a rule that you created, Amazon EventBridge routes it to the target that you specified. When you create a rule, you can configure these targets as well as downstream workflows based on the contents of the event.

EventBridge schema

The following event fields in the EventBridge event schema are specific to Amazon Bedrock.

- `jobArn` — The ARN of the model customization job.
- `outputModelArn` — The ARN of the output model. Published when the training job has completed.
- `jobStatus` — The current status of the job.
- `FailureMessage` — A failure message. Published when the training job has failed.

Event example

The folllowing is example event JSON for a failed model customization job.


```
{
    "version": "0",
    "id": "UUID",
    "detail-type": "Model Customization Job State Change",
    "source": "aws.bedrock",
    "account": "123412341234",
    "time": "2023-08-11T12:34:56Z",
    "region": "us-east-1",
    "resources": [ "arn:aws:bedrock:us-east-1:123412341234:model-customization-job/
abcdefghwxyz" ],
    "detail": {
        "version": "0.0",
        "jobName": "abcd-wxyz",
        "jobArn": "arn:aws:bedrock:us-east-1:123412341234:model-customization-job/
abcdefghwxyz",
        "outputModelName": "dummy-output-model-name",
        "outputModelArn": "arn:aws:bedrock:us-east-1:123412341234:dummy-output-model-
name",
```

```
        "roleArn": "arn:aws:iam::123412341234:role/JobExecutionRole",
        "jobStatus": "Failed",
        "failureMessage": "Failure Message here.",
        "creationTime": "2023-08-11T10:11:12Z",
        "lastModifiedTime": "2023-08-11T12:34:56Z",
        "endTime": "2023-08-11T12:34:56Z",
        "baseModelArn": "arn:aws:bedrock:us-east-1:123412341234:base-model-name",
        "hyperParameters": {
            "batchSize" : "batchSizeNumberUsed",
            "epochCount": "epochCountNumberUsed",
            "learningRate": "learningRateUsed",
            "learningRateWarmupSteps": "learningRateWarmupStepsUsed"
            },
        "trainingDataConfig": {
            "s3Uri": "s3://bucket/key",
            },
        "validationDataConfig": {
            "s3Uri": "s3://bucket/key",
            },
        "outputDataConfig": {
            "s3Uri": "s3://bucket/key",
            }
    }
}
```

Rules and targets

When an incoming event matches a rule that you created, the event is routed to the target that you specified for that rule, and the target processes these events. Targets support JSON format and can include AWS services such as Amazon EC2 instances, Lambda functions, Kinesis streams, Amazon ECS tasks, Step Functions, Amazon SNS topics, and Amazon SQS. To receive and process events correctly, you need to create rules and targets for matching, receiving, and correctly handling event data. You can create these rules and targets either through the Amazon EventBridge console, or through the AWS CLI.

Example rule

This rule matches an event pattern emitted by: `source [“aws.bedrock”]`. The rule captures all events sent by Amazon EventBridge that have source “aws.bedrock” to your default event bus.

```
{
```

```
    "source": ["aws.bedrock"]
}
```

Target

When creating a rule in Amazon EventBridge, you need to specify a target where EventBridge sends the event that matches your rule pattern. These targets can be a SageMaker pipeline, a Lambda function, an SNS topic, an SQS queue or any of the other targets that EventBridge currently supports. You can refer to the *Amazon EventBridge* documentation to learn how to set targets for events. For a procedure that shows how to use Amazon Simple Notification Service as a target, see Create a rule to handle Amazon Bedrock events.

Create a rule to handle Amazon Bedrock events

Complete the following procedures in order to receive email notifications about your Amazon Bedrock events.

Create an Amazon Simple Notification Service topic

1. Open the Amazon SNS console at https://console.aws.amazon.com/sns/v3/home.
2. In the navigation pane, choose **Topics**.
3. Choose **Create topic**.
4. For **Type**, choose **Standard**.
5. For **Name**, enter a name for your topic.
6. Choose **Create topic**.
7. Choose **Create subscription**.
8. For **Protocol**, choose **Email**.
9. For **Endpoint**, enter the email address that receives the notifications.
10. Choose **Create subscription**.
11. You'll receive an email message with the following subject line: `AWS Notification - Subscription Confirmation`. Follow the directions to confirm your subscription.

Use the following procedure to create a rule to handle your Amazon Bedrock events.

To create a rule to handle Amazon Bedrock events

1. Open the Amazon EventBridge console at https://console.aws.amazon.com/events/.

2. Choose **Create rule**.
3. For **Name**, enter a name for your rule.
4. For **Rule type**, choose **Rule with an event pattern**.
5. Choose **Next**.
6. For Event pattern, do the following:

   a. For **Event source**, choose **AWS services**.

   b. For **AWS service**, choose **Amazon Bedrock**.

   c. For **Event type**, choose **Model Customization Job State Change**.

   d. By default, we send notifications for every event. If you prefer, you can create an event pattern that filters events for a specific job state.

   e. Choose **Next**.
7. Specify a target as follows:

   a. For **Target types**, choose **AWS service**.

   b. For **Select a target**, choose **SNS topic**.

   c. For **Topic**, choose the SNS topic that you created for notifications.

   d. Choose **Next**.
8. (Optional) Add tags to your rule.
9. Choose **Next**.
10. Choose **Create rule**.

Log Amazon Bedrock API calls using AWS CloudTrail

Amazon Bedrock is integrated with AWS CloudTrail, a service that provides a record of actions taken by a user, role, or an AWS service in Amazon Bedrock. CloudTrail captures all API calls for Amazon Bedrock as events. The calls captured include calls from the Amazon Bedrock console and code calls to the Amazon Bedrock API operations. If you create a trail, you can enable continuous delivery of CloudTrail events to an Amazon S3 bucket, including events for Amazon Bedrock. If you don't configure a trail, you can still view the most recent events in the CloudTrail console in **Event history**. Using the information collected by CloudTrail, you can determine the request that was made to Amazon Bedrock, the IP address from which the request was made, who made the request, when it was made, and additional details.

To learn more about CloudTrail, see the AWS CloudTrail User Guide.

Amazon Bedrock information in CloudTrail

CloudTrail is enabled on your AWS account when you create the account. When activity occurs in Amazon Bedrock, that activity is recorded in a CloudTrail event along with other AWS service events in **Event history**. You can view, search, and download recent events in your AWS account. For more information, see Viewing events with CloudTrail Event history.

For an ongoing record of events in your AWS account, including events for Amazon Bedrock, create a trail. A *trail* enables CloudTrail to deliver log files to an Amazon S3 bucket. By default, when you create a trail in the console, the trail applies to all AWS Regions. The trail logs events from all Regions in the AWS partition and delivers the log files to the Amazon S3 bucket that you specify. Additionally, you can configure other AWS services to further analyze and act upon the event data collected in CloudTrail logs. For more information, see the following:

- Overview for creating a trail
- CloudTrail supported services and integrations
- Configuring Amazon SNS notifications for CloudTrail
- Receiving CloudTrail log files from multiple regions and Receiving CloudTrail log files from multiple accounts

Every event or log entry contains information about who generated the request. The identity information helps you determine the following:

- Whether the request was made with root or AWS Identity and Access Management (IAM) user credentials.
- Whether the request was made with temporary security credentials for a role or federated user.
- Whether the request was made by another AWS service.

For more information, see the CloudTrail userIdentity element.

Amazon Bedrock data events in CloudTrail

Data events provide information about the resource operations performed on or in a resource (for example, reading or writing to an Amazon S3 object). These are also known as data plane operations. Data events are often high-volume activities that CloudTrail doesn't log by default.

Amazon Bedrock doesn't log [Amazon Bedrock Runtime API operations](#) (`InvokeModel` and `InvokeModelWithResponseStream`).

Amazon Bedrock logs all [Agents for Amazon Bedrock Runtime API operations](#) actions to CloudTrail as *data events*.

- To log [InvokeAgent](#) calls, configure advanced event selectors to record data events for the `AWS::Bedrock::AgentAlias` resource type.
- To log [Retrieve](#) and [RetrieveAndGenerate](#) calls, configure advanced event selectors to record data events for the `AWS::Bedrock::KnowledgeBase` resource type.

From the CloudTrail console, choose **Bedrock agent alias** or **Bedrock knowledge base** for the **Data event type**. You can additionally filter on the `eventName` and `resources.ARN` fields by choosing a custom log selector template. For more information, see [Logging data events with the AWS Management Console](#).

From the AWS CLI, set the `resource.type` value equal to `AWS::Bedrock::AgentAlias` or `AWS::Bedrock::KnowledgeBase` and set the `eventCategory` equal to `Data`. For more information, see [Logging data events with the AWS CLI](#).

The following example shows how to configure a trail to log all Amazon Bedrock data events for all Amazon Bedrock resource types in the AWS CLI.

```
aws cloudtrail put-event-selectors --trail-name trailName \
--advanced-event-selectors \
'[
  {
    "Name": "Log all data events on an Agents for Amazon Bedrock agent alias",
    "FieldSelectors": [
      { "Field": "eventCategory", "Equals": ["Data"] },
      { "Field": "resources.type", "Equals": ["AWS::Bedrock::AgentAlias"] }
    ]
  },
  {
    "Name": "Log all data events on an Agents for Amazon Bedrock knowledge base",
    "FieldSelectors": [
      { "Field": "eventCategory", "Equals": ["Data"] },
      { "Field": "resources.type", "Equals": ["AWS::Bedrock::KnowledgeBase"] }
    ]
  }
```

```
]'
```

You can additionally filter on the `eventName` and `resources.ARN` fields. For more information about these fields, see AdvancedFieldSelector.

Additional charges apply for data events. For more information about CloudTrail pricing, see AWS CloudTrail Pricing.

Amazon Bedrock management events in CloudTrail

Management events provide information about management operations that are performed on resources in your AWS account. These are also known as control plane operations. CloudTrail logs management event API operations by default.

Amazon Bedrock logs the remainder of Amazon Bedrock API operations as management events. For a list of the Amazon Bedrock API operations that Amazon Bedrock logs to CloudTrail, see the following pages in the Amazon Bedrock API reference.

All Amazon Bedrock API operations and Agents for Amazon Bedrock API operations are logged by CloudTrail and documented in the Amazon Bedrock API Reference. For example, calls to the `InvokeModel`, `StopModelCustomizationJob`, and `CreateAgent` actions generate entries in the CloudTrail log files.

Understanding Amazon Bedrock log file entries

A trail is a configuration that enables delivery of events as log files to an Amazon S3 bucket that you specify. CloudTrail log files contain one or more log entries. An event represents a single request from any source and includes information about the requested action, the date and time of the action, request parameters, and so on. CloudTrail log files aren't an ordered stack trace of the public API calls, so they don't appear in any specific order.

The following example shows a CloudTrail log entry that demonstrates the `InvokeModel` action.

```
{
    "eventVersion": "1.08",
    "userIdentity": {
        "type": "IAMUser",
        "principalId": "AROAICFHPEXAMPLE",
        "arn": "arn:aws:iam::111122223333:user/userxyz",
        "accountId": "111122223333",
```

```
        "accessKeyId": "AKIAIOSFODNN7EXAMPLE",
        "userName": "userxyz"
    },
    "eventTime": "2023-10-11T21:58:59Z",
    "eventSource": "bedrock.amazonaws.com",
    "eventName": "InvokeModel",
    "awsRegion": "us-west-2",
    "sourceIPAddress": "192.0.2.0",
    "userAgent": "Boto3/1.28.62 md/Botocore#1.31.62 ua/2.0 os/macos#22.6.0 md/
arch#arm64 lang/python#3.9.6 md/pyimpl#CPython cfg/retry-mode#legacy Botocore/1.31.62",
    "requestParameters": {
        "modelId": "stability.stable-diffusion-xl-v0"
    },
    "responseElements": null,
    "requestID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE22222",
    "eventID": "a1b2c3d4-5678-90ab-cdef-EXAMPLE11111 ",
    "readOnly": false,
    "eventType": "AwsApiCall",
    "managementEvent": true,
    "recipientAccountId": "111122223333",
    "eventCategory": "Management",
    "tlsDetails": {
        "tlsVersion": "TLSv1.2",
        "cipherSuite": "cipher suite",
        "clientProvidedHostHeader": "bedrock-runtime.us-west-2.amazonaws.com"
    }
}
```

Code examples for Amazon Bedrock using AWS SDKs

The following code examples show how to use Amazon Bedrock with an AWS software development kit (SDK).

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Scenarios for Agents for Amazon Bedrock Runtime using AWS SDKs
    - Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Code examples for Amazon Bedrock using AWS SDKs

The following code examples show how to use Amazon Bedrock with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Amazon Bedrock

The following code examples show how to get started using Amazon Bedrock.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
using Amazon;
using Amazon.Bedrock;
using Amazon.Bedrock.Model;

namespace ListFoundationModelsExample
```

```
{
    /// <summary>
    /// This example shows how to list foundation models.
    /// </summary>
    internal class HelloBedrock
    {
        /// <summary>
        /// Main method to call the ListFoundationModelsAsync method.
        /// </summary>
        /// <param name="args"> The command line arguments. </param>
        static async Task Main(string[] args)
        {
            // Specify a region endpoint where Amazon Bedrock is available.
 For a list of supported region see https://docs.aws.amazon.com/bedrock/latest/
userguide/what-is-bedrock.html#bedrock-regions
            AmazonBedrockClient bedrockClient = new(RegionEndpoint.USWest2);

            await ListFoundationModelsAsync(bedrockClient);

        }



        /// <summary>
        /// List foundation models.
        /// </summary>
        /// <param name="bedrockClient"> The Amazon Bedrock client. </param>
        private static async Task ListFoundationModelsAsync(AmazonBedrockClient
 bedrockClient)
        {
            Console.WriteLine("List foundation models with no filter");

            try
            {
                ListFoundationModelsResponse response = await
 bedrockClient.ListFoundationModelsAsync(new ListFoundationModelsRequest()
                {
                });

                if (response?.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    foreach (var fm in response.ModelSummaries)
                    {
                        WriteToConsole(fm);
                    }
```

```
                }
                else
                {
                    Console.WriteLine("Something wrong happened");
                }
            }
            catch (AmazonBedrockException e)
            {
                Console.WriteLine(e.Message);
            }
        }


        /// <summary>
        /// Write the foundation model summary to console.
        /// </summary>
        /// <param name="foundationModel"> The foundation model summary to write
 to console. </param>
        private static void WriteToConsole(FoundationModelSummary
 foundationModel)
        {
            Console.WriteLine($"{foundationModel.ModelId}, Customization:
 {String.Join(", ", foundationModel.CustomizationsSupported)}, Stream:
 {foundationModel.ResponseStreamingSupported}, Input: {String.Join(",
 ", foundationModel.InputModalities)}, Output: {String.Join(", ",
 foundationModel.OutputModalities)}");
        }
    }
}
```

- For API details, see ListFoundationModels in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
package main

import (
 "context"
 "fmt"

 "github.com/aws/aws-sdk-go-v2/config"
 "github.com/aws/aws-sdk-go-v2/service/bedrock"
)

const region = "us-east-1"

// main uses the AWS SDK for Go (v2) to create an Amazon Bedrock client and
// list the available foundation models in your account and the chosen region.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {
    sdkConfig, err := config.LoadDefaultConfig(context.TODO(),
 config.WithRegion(region))
    if err != nil {
        fmt.Println("Couldn't load default configuration. Have you set up your
 AWS account?")
        fmt.Println(err)
        return
    }
    bedrockClient := bedrock.NewFromConfig(sdkConfig)
    result, err := bedrockClient.ListFoundationModels(context.TODO(),
 &bedrock.ListFoundationModelsInput{})
    if err != nil {
  fmt.Printf("Couldn't list foundation models. Here's why: %v\n", err)
  return
    }
    if len(result.ModelSummaries) == 0 {
    fmt.Println("There are no foundation models.")}
    for _, modelSummary := range result.ModelSummaries {
        fmt.Println(*modelSummary.ModelId)
    }
}
```

- For API details, see [ListFoundationModels] in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import {
  BedrockClient,
  ListFoundationModelsCommand,
} from "@aws-sdk/client-bedrock";

const REGION = "us-east-1";
const client = new BedrockClient({ region: REGION });

export const main = async () => {
  const command = new ListFoundationModelsCommand({});

  const response = await client.send(command);
  const models = response.modelSummaries;

  console.log("Listing the available Bedrock foundation models:");

  for (let model of models) {
    console.log("=".repeat(42));
    console.log(` Model: ${model.modelId}`);
    console.log("-".repeat(42));
    console.log(` Name: ${model.modelName}`);
    console.log(` Provider: ${model.providerName}`);
    console.log(` Model ARN: ${model.modelArn}`);
    console.log(` Input modalities: ${model.inputModalities}`);
    console.log(` Output modalities: ${model.outputModalities}`);
    console.log(` Supported customizations: ${model.customizationsSupported}`);
    console.log(` Supported inference types: ${model.inferenceTypesSupported}`);
```

```
    console.log(` Lifecycle status: ${model.modelLifecycle.status}`);
    console.log("=".repeat(42) + "\n");
  }

  const active = models.filter(
    (m) => m.modelLifecycle.status === "ACTIVE",
  ).length;
  const legacy = models.filter(
    (m) => m.modelLifecycle.status === "LEGACY",
  ).length;

  console.log(
    `There are ${active} active and ${legacy} legacy foundation models in
 ${REGION}.`,
  );

  return response;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  await main();
}
```

- For API details, see ListFoundationModels in *AWS SDK for JavaScript API Reference*.

Code examples

- Actions for Amazon Bedrock using AWS SDKs
  - Use GetFoundationModel with an AWS SDK or command line tool
  - Use ListFoundationModels with an AWS SDK or command line tool
- Scenarios for Amazon Bedrock using AWS SDKs
  - Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Actions for Amazon Bedrock using AWS SDKs

The following code examples demonstrate how to perform individual Amazon Bedrock actions with AWS SDKs. These excerpts call the Amazon Bedrock API and are code excerpts from larger programs that must be run in context. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the Amazon Bedrock API Reference.

Examples

- Use GetFoundationModel with an AWS SDK or command line tool
- Use ListFoundationModels with an AWS SDK or command line tool

Use `GetFoundationModel` with an AWS SDK or command line tool

The following code examples show how to use `GetFoundationModel`.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get details about a foundation model using the synchronous Amazon Bedrock client.

```
    /**
     * Get details about an Amazon Bedrock foundation model.
     *
     * @param bedrockClient   The service client for accessing Amazon Bedrock.
     * @param modelIdentifier The model identifier.
     * @return An object containing the foundation model's details.
     */
    public static FoundationModelDetails getFoundationModel(BedrockClient 
 bedrockClient, String modelIdentifier) {
        try {
            GetFoundationModelResponse response = 
 bedrockClient.getFoundationModel(
                    r -> r.modelIdentifier(modelIdentifier)
            );
```

```
            FoundationModelDetails model = response.modelDetails();

            System.out.println(" Model ID:                     " +
 model.modelId());
            System.out.println(" Model ARN:                    " +
 model.modelArn());
            System.out.println(" Model Name:                   " +
 model.modelName());
            System.out.println(" Provider Name:                " +
 model.providerName());
            System.out.println(" Lifecycle status:             " +
 model.modelLifecycle().statusAsString());
            System.out.println(" Input modalities:             " +
 model.inputModalities());
            System.out.println(" Output modalities:            " +
 model.outputModalities());
            System.out.println(" Supported customizations:     " +
 model.customizationsSupported());
            System.out.println(" Supported inference types:    " +
 model.inferenceTypesSupported());
            System.out.println(" Response streaming supported: " +
 model.responseStreamingSupported());

            return model;

        } catch (ValidationException e) {
            throw new IllegalArgumentException(e.getMessage());
        } catch (SdkException e) {
            System.err.println(e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

Get details about a foundation model using the asynchronous Amazon Bedrock client.

```
    /**
     * Get details about an Amazon Bedrock foundation model.
     *
     * @param bedrockClient   The async service client for accessing Amazon
 Bedrock.
     * @param modelIdentifier The model identifier.
```

```
     * @return An object containing the foundation model's details.
     */
    public static FoundationModelDetails getFoundationModel(BedrockAsyncClient
 bedrockClient, String modelIdentifier) {
        try {
            CompletableFuture<GetFoundationModelResponse> future =
 bedrockClient.getFoundationModel(
                    r -> r.modelIdentifier(modelIdentifier)
            );

            FoundationModelDetails model = future.get().modelDetails();

            System.out.println(" Model ID:                     " +
 model.modelId());
            System.out.println(" Model ARN:                    " +
 model.modelArn());
            System.out.println(" Model Name:                   " +
 model.modelName());
            System.out.println(" Provider Name:                " +
 model.providerName());
            System.out.println(" Lifecycle status:             " +
 model.modelLifecycle().statusAsString());
            System.out.println(" Input modalities:             " +
 model.inputModalities());
            System.out.println(" Output modalities:            " +
 model.outputModalities());
            System.out.println(" Supported customizations:     " +
 model.customizationsSupported());
            System.out.println(" Supported inference types:    " +
 model.inferenceTypesSupported());
            System.out.println(" Response streaming supported: " +
 model.responseStreamingSupported());

            return model;

        } catch (ExecutionException e) {
            if (e.getMessage().contains("ValidationException")) {
                throw new IllegalArgumentException(e.getMessage());
            } else {
                System.err.println(e.getMessage());
                throw new RuntimeException(e);
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
```

```
            System.err.println(e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

- For API details, see GetFoundationModel in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get details about a foundation model.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import {
  BedrockClient,
  GetFoundationModelCommand,
} from "@aws-sdk/client-bedrock";

/**
 * Get details about an Amazon Bedrock foundation model.
 *
 * @return {FoundationModelDetails} - The list of available bedrock foundation
 models.
 */
export const getFoundationModel = async () => {
  const client = new BedrockClient();

  const command = new GetFoundationModelCommand({
    modelIdentifier: "amazon.titan-embed-text-v1",
  });
```

```
  const response = await client.send(command);

  return response.modelDetails;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const model = await getFoundationModel();
  console.log(model);
}
```

- For API details, see GetFoundationModel in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get details about a foundation model.

```
    def get_foundation_model(self, model_identifier):
        """
        Get details about an Amazon Bedrock foundation model.

        :return: The foundation model's details.
        """

        try:
            return self.bedrock_client.get_foundation_model(
                modelIdentifier=model_identifier
            )["modelDetails"]
        except ClientError:
            logger.error(
                f"Couldn't get foundation models details for {model_identifier}"
            )
```

```
            raise
```

- For API details, see GetFoundationModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListFoundationModels` with an AWS SDK or command line tool

The following code examples show how to use `ListFoundationModels`.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the available Bedrock foundation models.

```
        /// <summary>
        /// List foundation models.
        /// </summary>
        /// <param name="bedrockClient"> The Amazon Bedrock client. </param>
        private static async Task ListFoundationModelsAsync(AmazonBedrockClient
 bedrockClient)
        {
            Console.WriteLine("List foundation models with no filter");

            try
            {
                ListFoundationModelsResponse response = await
 bedrockClient.ListFoundationModelsAsync(new ListFoundationModelsRequest()
                {
```

```
                });

                if (response?.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    foreach (var fm in response.ModelSummaries)
                    {
                        WriteToConsole(fm);
                    }
                }
                else
                {
                    Console.WriteLine("Something wrong happened");
                }
            }
            catch (AmazonBedrockException e)
            {
                Console.WriteLine(e.Message);
            }
        }
```

- For API details, see [ListFoundationModels](#) in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

List the available Bedrock foundation models.

```
// FoundationModelWrapper encapsulates Amazon Bedrock actions used in the
 examples.
// It contains a Bedrock service client that is used to perform foundation model
 actions.
type FoundationModelWrapper struct {
```

```
 BedrockClient *bedrock.Client
}



// ListPolicies lists Bedrock foundation models that you can use.
func (wrapper FoundationModelWrapper) ListFoundationModels()
 ([]types.FoundationModelSummary, error) {

    var models []types.FoundationModelSummary

    result, err := wrapper.BedrockClient.ListFoundationModels(context.TODO(),
 &bedrock.ListFoundationModelsInput{})

    if err != nil {
        log.Printf("Couldn't list foundation models. Here's why: %v\n", err)
    } else {
        models = result.ModelSummaries
    }
    return models, err
}
```

- For API details, see ListFoundationModels in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the available Amazon Bedrock foundation models using the synchronous Amazon Bedrock client.

```
    /**
     * Lists Amazon Bedrock foundation models that you can use.
     * You can filter the results with the request parameters.
```

```
     *
     * @param bedrockClient The service client for accessing Amazon Bedrock.
     * @return A list of objects containing the foundation models' details
     */
    public static List<FoundationModelSummary> listFoundationModels(BedrockClient
 bedrockClient) {

        try {
            ListFoundationModelsResponse response =
 bedrockClient.listFoundationModels(r -> {});

            List<FoundationModelSummary> models = response.modelSummaries();

            if (models.isEmpty()) {
                System.out.println("No available foundation models in " +
 region.toString());
            } else {
                for (FoundationModelSummary model : models) {
                    System.out.println("Model ID: " + model.modelId());
                    System.out.println("Provider: " + model.providerName());
                    System.out.println("Name:     " + model.modelName());
                    System.out.println();
                }
            }

            return models;

        } catch (SdkClientException e) {
            System.err.println(e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

List the available Amazon Bedrock foundation models using the asynchronous Amazon Bedrock client.

```
    /**
     * Lists Amazon Bedrock foundation models that you can use.
     * You can filter the results with the request parameters.
     *
     * @param bedrockClient The async service client for accessing Amazon
 Bedrock.
```

```
     * @return A list of objects containing the foundation models' details
     */
    public static List<FoundationModelSummary>
 listFoundationModels(BedrockAsyncClient bedrockClient) {
        try {
            CompletableFuture<ListFoundationModelsResponse> future =
 bedrockClient.listFoundationModels(r -> {});

            List<FoundationModelSummary> models = future.get().modelSummaries();

            if (models.isEmpty()) {
                System.out.println("No available foundation models in " +
 region.toString());
            } else {
                for (FoundationModelSummary model : models) {
                    System.out.println("Model ID: " + model.modelId());
                    System.out.println("Provider: " + model.providerName());
                    System.out.println("Name:     " + model.modelName());
                    System.out.println();
                }
            }

            return models;

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
            throw new RuntimeException(e);
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
            throw new RuntimeException(e);
        }
    }
```

- For API details, see ListFoundationModels in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javascriptv3/example_code/bedrock#code-examples).

List the available foundation models.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import {
  BedrockClient,
  ListFoundationModelsCommand,
} from "@aws-sdk/client-bedrock";

/**
 * List the available Amazon Bedrock foundation models.
 *
 * @return {FoundationModelSummary[]} - The list of available bedrock foundation
 models.
 */
export const listFoundationModels = async () => {
  const client = new BedrockClient();

  const input = {
    // byProvider: 'STRING_VALUE',
    // byCustomizationType: 'FINE_TUNING' || 'CONTINUED_PRE_TRAINING',
    // byOutputModality: 'TEXT' || 'IMAGE' || 'EMBEDDING',
    // byInferenceType: 'ON_DEMAND' || 'PROVISIONED',
  };

  const command = new ListFoundationModelsCommand(input);

  const response = await client.send(command);

  return response.modelSummaries;
```

```
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const models = await listFoundationModels();
  console.log(models);
}
```

- For API details, see ListFoundationModels in *AWS SDK for JavaScript API Reference*.

Kotlin

SDK for Kotlin

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the available Amazon Bedrock foundation models.

```
suspend fun listFoundationModels(): List<FoundationModelSummary>? {
    BedrockClient { region = "us-east-1" }.use { bedrockClient ->
        val response =
 bedrockClient.listFoundationModels(ListFoundationModelsRequest {})
        response.modelSummaries?.forEach { model ->
            println("==========================================")
            println(" Model ID: ${model.modelId}")
            println("------------------------------------------")
            println(" Name: ${model.modelName}")
            println(" Provider: ${model.providerName}")
            println(" Input modalities: ${model.inputModalities}")
            println(" Output modalities: ${model.outputModalities}")
            println(" Supported customizations:
 ${model.customizationsSupported}")
            println(" Supported inference types:
 ${model.inferenceTypesSupported}")
            println("------------------------------------------\n")
        }
        return response.modelSummaries
```

```
    }
}
```

- For API details, see ListFoundationModels in *AWS SDK for Kotlin API reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the available Amazon Bedrock foundation models.

```
    public function listFoundationModels()
    {
        $result = $this->bedrockClient->listFoundationModels();
        return $result;
    }
```

- For API details, see ListFoundationModels in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the available Amazon Bedrock foundation models.

```
    def list_foundation_models(self):
```

```
        """
        List the available Amazon Bedrock foundation models.

        :return: The list of available bedrock foundation models.
        """

        try:
            response = self.bedrock_client.list_foundation_models()
            models = response["modelSummaries"]
            logger.info("Got %s foundation models.", len(models))
            return models

        except ClientError:
            logger.error("Couldn't list foundation models.")
            raise
```

- For API details, see ListFoundationModels in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Amazon Bedrock using AWS SDKs

The following code examples show you how to implement common scenarios in Amazon Bedrock with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Amazon Bedrock. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

The following code example shows how to build and orchestrate generative AI applications with Amazon Bedrock and Step Functions.

Python

SDK for Python (Boto3)

The Amazon Bedrock Serverless Prompt Chaining scenario demonstrates how AWS Step Functions, Amazon Bedrock, and Agents for Amazon Bedrock can be used to build and orchestrate complex, serverless, and highly scalable generative AI applications. It contains the following working examples:

- Write an analysis of a given novel for a literature blog. This example illustrates a simple, sequential chain of prompts.
- Generate a short story about a given topic. This example illustrates how the AI can iteratively process a list of items that it previously generated.
- Create an itinerary for a weekend vacation to a given destination. This example illustrates how to parallelize multiple distinct prompts.
- Pitch movie ideas to a human user acting as a movie producer. This example illustrates how to parallelize the same prompt with different inference parameters, how to backtrack to a previous step in the chain, and how to include human input as part of the workflow.
- Plan a meal based on ingredients the user has at hand. This example illustrates how prompt chains can incorporate two distinct AI conversations, with two AI personas engaging in a debate with each other to improve the final outcome.
- Find and summarize today's highest trending GitHub repository. This example illustrates chaining multiple AI agents that interact with external APIs.

For complete source code and instructions to set up and run, see the full project on GitHub.

Services used in this example

- Amazon Bedrock
- Amazon Bedrock Runtime
- Agents for Amazon Bedrock
- Agents for Amazon Bedrock Runtime
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples for Amazon Bedrock Runtime using AWS SDKs

The following code examples show how to use Amazon Bedrock Runtime with an AWS software development kit (SDK).

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Amazon Bedrock

The following code examples show how to get started using Amazon Bedrock.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
package main

import (
 "context"
 "encoding/json"
 "flag"
 "fmt"
 "log"
 "os"
 "strings"

 "github.com/aws/aws-sdk-go-v2/aws"
 "github.com/aws/aws-sdk-go-v2/config"
```

```
 "github.com/aws/aws-sdk-go-v2/service/bedrockruntime"
)

// Each model provider defines their own individual request and response formats.
// For the format, ranges, and default values for the different models, refer to:
// https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters.html

type ClaudeRequest struct {
 Prompt            string `json:"prompt"`
 MaxTokensToSample int    `json:"max_tokens_to_sample"`
 // Omitting optional request parameters
}

type ClaudeResponse struct {
 Completion string `json:"completion"`
}

// main uses the AWS SDK for Go (v2) to create an Amazon Bedrock Runtime client
// and invokes Anthropic Claude 2 inside your account and the chosen region.
// This example uses the default settings specified in your shared credentials
// and config files.
func main() {

 region := flag.String("region", "us-east-1", "The AWS region")
 flag.Parse()

 fmt.Printf("Using AWS region: %s\n", *region)

 sdkConfig, err := config.LoadDefaultConfig(context.Background(),
 config.WithRegion(*region))
 if err != nil {
  fmt.Println("Couldn't load default configuration. Have you set up your AWS
 account?")
  fmt.Println(err)
  return
 }

 client := bedrockruntime.NewFromConfig(sdkConfig)

 modelId := "anthropic.claude-v2"

 prompt := "Hello, how are you today?"

 // Anthropic Claude requires you to enclose the prompt as follows:
```

```
 prefix := "Human: "
 postfix := "\n\nAssistant:"
 wrappedPrompt := prefix + prompt + postfix

 request := ClaudeRequest{
  Prompt:            wrappedPrompt,
  MaxTokensToSample: 200,
 }

 body, err := json.Marshal(request)
 if err != nil {
  log.Panicln("Couldn't marshal the request: ", err)
 }

 result, err := client.InvokeModel(context.Background(),
 &bedrockruntime.InvokeModelInput{
  ModelId:     aws.String(modelId),
  ContentType: aws.String("application/json"),
  Body:        body,
 })

 if err != nil {
  errMsg := err.Error()
  if strings.Contains(errMsg, "no such host") {
   fmt.Printf("Error: The Bedrock service is not available in the selected
 region. Please double-check the service availability for your region at https://
aws.amazon.com/about-aws/global-infrastructure/regional-product-services/.\n")
  } else if strings.Contains(errMsg, "Could not resolve the foundation model") {
   fmt.Printf("Error: Could not resolve the foundation model from model
 identifier: \"%v\". Please verify that the requested model exists and is
 accessible within the specified region.\n", modelId)
  } else {
   fmt.Printf("Error: Couldn't invoke Anthropic Claude. Here's why: %v\n", err)
  }
  os.Exit(1)
 }

 var response ClaudeResponse

 err = json.Unmarshal(result.Body, &response)

 if err != nil {
  log.Fatal("failed to unmarshal", err)
 }
```

```
    fmt.Println("Prompt:\n", prompt)
    fmt.Println("Response from Anthropic Claude:\n", response.Completion)
}
```

- For API details, see InvokeModel in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

/**
 * @typedef {Object} Content
 * @property {string} text
 *
 * @typedef {Object} Usage
 * @property {number} input_tokens
 * @property {number} oputput_tokens
 *
 * @typedef {Object} ResponseBody
 * @property {Content[]} content
 * @property {Usage} usage
 */

import { fileURLToPath } from "url";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

const AWS_REGION = "us-east-1";
```

```
const MODEL_ID = "anthropic.claude-3-haiku-20240307-v1:0";
const PROMPT = "Hi. In a short paragraph, explain what you can do.";

const hello = async () => {
  console.log("=".repeat(35));
  console.log("Welcome to the Amazon Bedrock demo!");
  console.log("=".repeat(35));

  console.log("Model: Anthropic Claude 3 Haiku");
  console.log(`Prompt: ${PROMPT}\n`);
  console.log("Invoking model...\n");

  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: AWS_REGION });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [{ role: "user", content: [{ type: "text", text: PROMPT }] }],
  };

  // Invoke Claude with the payload and wait for the response.
  const apiResponse = await client.send(
    new InvokeModelCommand({
      contentType: "application/json",
      body: JSON.stringify(payload),
      modelId: MODEL_ID,
    }),
  );

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {ResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  const responses = responseBody.content;

  if (responses.length === 1) {
    console.log(`Response: ${responses[0].text}`);
  } else {
    console.log("Haiku returned multiple responses:");
    console.log(responses);
  }
```

```
  console.log(`\nNumber of input tokens:   ${responseBody.usage.input_tokens}`);
  console.log(`Number of output tokens: ${responseBody.usage.output_tokens}`);
};

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  await hello();
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

Code examples

- Invoke model examples for Amazon Bedrock Runtime using AWS SDKs
  - Invoke the AI21 Labs Jurassic-2 model on Amazon Bedrock for text generation
  - Invoke Amazon Titan Image G1 on Amazon Bedrock to generate an image
  - Invoke Amazon Titan Text G1 on Amazon Bedrock for text generation
  - Invoke Anthropic Claude 2.x on Amazon Bedrock and process the response stream in real-time
  - Invoke Anthropic Claude 2.x on Amazon Bedrock for text generation
  - Invoke Anthropic Claude 3 on Amazon Bedrock with a multimodal prompt
  - Invoke Anthropic Claude 3 on Amazon Bedrock and process the response stream in real-time
  - Invoke Anthropic Claude 3 on Amazon Bedrock to generate text
  - Invoke the Anthropic Claude Instant model on Amazon Bedrock for text generation
  - Invoke Meta Llama 2 on Amazon Bedrock using Meta's native request and response payloads
  - Invoke Meta Llama 2 on Amazon Bedrock using Meta's native request and response payloads with a response stream
  - Invoke Meta Llama 3 on Amazon Bedrock using Meta's native request and response payloads
  - Invoke Meta Llama 3 on Amazon Bedrock using Meta's native request and response payloads with a response stream
  - Invoke the Mistral 7B model on Amazon Bedrock for text generation
  - Invoke the Mixtral 8x7B model on Amazon Bedrock for text generation
  - Invoke Stability.ai Stable Diffusion XL on Amazon Bedrock to generate an image
- Scenarios for Amazon Bedrock Runtime using AWS SDKs
  - Create a sample application that offers playgrounds to interact with Amazon Bedrock foundation models using an AWS SDK

- Invoke various foundation models on Amazon Bedrock
- Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Invoke model examples for Amazon Bedrock Runtime using AWS SDKs

The following code examples show how to use Amazon Bedrock Runtime with AWS SDKs.

Examples

Invoke the AI21 Labs Jurassic-2 model on Amazon Bedrock for text generation

The following code examples show how to invoke the AI21 Labs Jurassic-2 model on Amazon Bedrock for text generation.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/dotnetv3/Bedrock-runtime#code-examples).

Asynchronously invoke the AI21 Labs Jurassic-2 foundation model.

```
        /// <summary>
        /// Asynchronously invokes the AI21 Labs Jurassic-2 model to run an
 inference based on the provided input.
        /// </summary>
        /// <param name="prompt">The prompt that you want Claude to complete.</
param>
        /// <returns>The inference response from the model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for AI21 Labs Jurassic-2,
 refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-jurassic2.html
        /// </remarks>
        public static async Task<string> InvokeJurassic2Async(string prompt)
        {
            string jurassic2ModelId = "ai21.j2-mid-v1";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            string payload = new JsonObject()
            {
                { "prompt", prompt },
                { "maxTokens", 200 },
                { "temperature", 0.5 }
            }.ToJsonString();

            string generatedText = "";
            try
```

```
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
InvokeModelRequest()
                {
                    ModelId = jurassic2ModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    return JsonNode.ParseAsync(response.Body)
                        .Result?["completions"]?
                        .AsArray()[0]?["data"]?
                        .AsObject()["text"]?.GetValue<string>() ?? "";
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
+ response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return generatedText;
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the AI21 Labs Jurassic-2 foundation model to generate text.

```
// Each model provider has their own individual request and response formats.
// For the format, ranges, and default values for AI21 Labs Jurassic-2, refer to:
// https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
jurassic2.html

type Jurassic2Request struct {
 Prompt      string  `json:"prompt"`
 MaxTokens   int     `json:"maxTokens,omitempty"`
 Temperature float64 `json:"temperature,omitempty"`
}

type Jurassic2Response struct {
 Completions []Completion `json:"completions"`
}
type Completion struct {
 Data Data `json:"data"`
}
type Data struct {
 Text string `json:"text"`
}

// Invokes AI21 Labs Jurassic-2 on Amazon Bedrock to run an inference using the
 input
// provided in the request body.
func (wrapper InvokeModelWrapper) InvokeJurassic2(prompt string) (string, error)
 {
 modelId := "ai21.j2-mid-v1"

 body, err := json.Marshal(Jurassic2Request{
  Prompt:      prompt,
  MaxTokens:   200,
  Temperature: 0.5,
 })

 if err != nil {
  log.Fatal("failed to marshal", err)
 }

 output, err := wrapper.BedrockRuntimeClient.InvokeModel(context.TODO(),
 &bedrockruntime.InvokeModelInput{
  ModelId:     aws.String(modelId),
```

```
  ContentType: aws.String("application/json"),
  Body:        body,
 })

 if err != nil {
  ProcessError(err, modelId)
 }

 var response Jurassic2Response
 if err := json.Unmarshal(output.Body, &response); err != nil {
  log.Fatal("failed to unmarshal", err)
 }

 return response.Completions[0].Data.Text, nil
}
```

- For API details, see InvokeModel in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the AI21 Labs Jurassic-2 foundation model to generate text.

```
    /**
     * Asynchronously invokes the AI21 Labs Jurassic-2 model to run an inference
     * based on the provided input.
     *
     * @param prompt The prompt that you want Jurassic to complete.
     * @return The inference response generated by the model.
     */
    public static String invokeJurassic2(String prompt) {
        /*
```

```
         * The different model providers have individual request and response
 formats.
         * For the format, ranges, and default values for Anthropic Claude, refer
 to:
         * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
claude.html
         */

        String jurassic2ModelId = "ai21.j2-mid-v1";

        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_EAST_1)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();

        String payload = new JSONObject()
                .put("prompt", prompt)
                .put("temperature", 0.5)
                .put("maxTokens", 200)
                .toString();

        InvokeModelRequest request = InvokeModelRequest.builder()
                .body(SdkBytes.fromUtf8String(payload))
                .modelId(jurassic2ModelId)
                .contentType("application/json")
                .accept("application/json")
                .build();

        CompletableFuture<InvokeModelResponse> completableFuture =
 client.invokeModel(request)
                .whenComplete((response, exception) -> {
                    if (exception != null) {
                        System.out.println("Model invocation failed: " +
 exception);
                    }
                });

        String generatedText = "";
        try {
            InvokeModelResponse response = completableFuture.get();
            JSONObject responseBody = new
 JSONObject(response.body().asUtf8String());
            generatedText = responseBody
                    .getJSONArray("completions")
```

```
                    .getJSONObject(0)
                    .getJSONObject("data")
                    .getString("text");

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
        }

        return generatedText;
    }
```

Invoke the AI21 Labs Jurassic-2 foundation model to generate text.

```
        /**
         * Invokes the AI21 Labs Jurassic-2 model to run an inference based on
 the
         * provided input.
         *
         * @param prompt The prompt for Jurassic to complete.
         * @return The generated response.
         */
        public static String invokeJurassic2(String prompt) {
                /*
                 * The different model providers have individual request and
 response formats.
                 * For the format, ranges, and default values for AI21 Labs
 Jurassic-2, refer
                 * to:
                 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-jurassic2.html
                 */

                String jurassic2ModelId = "ai21.j2-mid-v1";

                BedrockRuntimeClient client = BedrockRuntimeClient.builder()
                                .region(Region.US_EAST_1)

 .credentialsProvider(ProfileCredentialsProvider.create())
                                .build();
```

```
                String payload = new JSONObject()
                                .put("prompt", prompt)
                                .put("temperature", 0.5)
                                .put("maxTokens", 200)
                                .toString();

                InvokeModelRequest request = InvokeModelRequest.builder()
                                .body(SdkBytes.fromUtf8String(payload))
                                .modelId(jurassic2ModelId)
                                .contentType("application/json")
                                .accept("application/json")
                                .build();

                InvokeModelResponse response = client.invokeModel(request);

                JSONObject responseBody = new
 JSONObject(response.body().asUtf8String());

                String generatedText = responseBody
                                .getJSONArray("completions")
                                .getJSONObject(0)
                                .getJSONObject("data")
                                .getString("text");

                return generatedText;
        }
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the AI21 Labs Jurassic-2 foundation model to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} Data
 * @property {string} text
 *
 * @typedef {Object} Completion
 * @property {Data} data
 *
 * @typedef {Object} ResponseBody
 * @property {Completion[]} completions
 */

/**
 * Invokes an AI21 Labs Jurassic-2 model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to "ai21.j2-
mid-v1".
 */
export const invokeModel = async (prompt, modelId = "ai21.j2-mid-v1") => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    prompt,
    maxTokens: 500,
    temperature: 0.5,
  };

  // Invoke the model with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
```

```
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s).
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {ResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.completions[0].data.text;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time..."';
  const modelId = FoundationModels.JURASSIC2_MID.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the AI21 Labs Jurassic-2 foundation model to generate text.

```
    public function invokeJurassic2($prompt)
    {
        # The different model providers have individual request and response
 formats.
        # For the format, ranges, and default values for AI21 Labs Jurassic-2,
 refer to:
        # https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
jurassic2.html

        $completion = "";

        try {
            $modelId = 'ai21.j2-mid-v1';

            $body = [
                'prompt' => $prompt,
                'temperature' => 0.5,
                'maxTokens' => 200,
            ];

            $result = $this->bedrockRuntimeClient->invokeModel([
                'contentType' => 'application/json',
                'body' => json_encode($body),
                'modelId' => $modelId,
            ]);

            $response_body = json_decode($result['body']);

            $completion = $response_body->completions[0]->data->text;
        } catch (Exception $e) {
            echo "Error: ({$e->getCode()}) - {$e->getMessage()}\n";
        }

        return $completion;
    }
```

- For API details, see InvokeModel in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/python/example_code/bedrock-runtime#code-examples).

Invoke the AI21 Labs Jurassic-2 foundation model to generate text.

```
    def invoke_jurassic2(self, prompt):
        """
        Invokes the AI21 Labs Jurassic-2 large-language model to run an inference
        using the input provided in the request body.

        :param prompt: The prompt that you want Jurassic-2 to complete.
        :return: Inference response from the model.
        """

        try:
            # The different model providers have individual request and response
 formats.
            # For the format, ranges, and default values for AI21 Labs
 Jurassic-2, refer to:
            # https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-jurassic2.html

            body = {
                "prompt": prompt,
                "temperature": 0.5,
                "maxTokens": 200,
            }

            response = self.bedrock_runtime_client.invoke_model(
                modelId="ai21.j2-mid-v1", body=json.dumps(body)
            )

            response_body = json.loads(response["body"].read())
            completion = response_body["completions"][0]["data"]["text"]

            return completion
```

```
        except ClientError:
            logger.error("Couldn't invoke Jurassic-2")
            raise
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Amazon Titan Image G1 on Amazon Bedrock to generate an image

The following code examples show how to invoke Amazon Titan Image G1 on Amazon Bedrock to generate an image.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Amazon Titan Image Generator G1 foundation model to generate images.

```
        /// <summary>
        /// Asynchronously invokes the Amazon Titan Image Generator G1 model to
 run an inference based on the provided input.
        /// </summary>
        /// <param name="prompt">The prompt that describes the image Amazon Titan
 Image Generator G1 has to generate.</param>
        /// <returns>A base-64 encoded image generated by model</returns>
        /// <remarks>
```

```
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Amazon Titan Image
 Generator G1, refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-titan-image.html
        /// </remarks>
        public static async Task<string?> InvokeTitanImageGeneratorG1Async(string
 prompt, int seed)
        {
            string titanImageGeneratorG1ModelId = "amazon.titan-image-generator-
v1";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            string payload = new JsonObject()
            {
                { "taskType", "TEXT_IMAGE" },
                { "textToImageParams", new JsonObject()
                    {
                        { "text", prompt }
                    }
                },
                { "imageGenerationConfig", new JsonObject()
                    {
                        { "numberOfImages", 1 },
                        { "quality", "standard" },
                        { "cfgScale", 8.0f },
                        { "height", 512 },
                        { "width", 512 },
                        { "seed", seed }
                    }
                }
            }.ToJsonString();

            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
 InvokeModelRequest()
                {
                    ModelId = titanImageGeneratorG1ModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
```

```
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    var results = JsonNode.ParseAsync(response.Body).Result?
["images"]?.AsArray();

                    return results?[0]?.GetValue<string>();
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
 + response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return null;
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Amazon Titan Image Generator G1 model to generate images.

```
type TitanImageRequest struct {
 TaskType              string                `json:"taskType"`
 TextToImageParams     TextToImageParams     `json:"textToImageParams"`
```

```
 ImageGenerationConfig ImageGenerationConfig `json:"imageGenerationConfig"`
}
type TextToImageParams struct {
 Text string `json:"text"`
}
type ImageGenerationConfig struct {
 NumberOfImages int     `json:"numberOfImages"`
 Quality        string  `json:"quality"`
 CfgScale       float64 `json:"cfgScale"`
 Height         int     `json:"height"`
 Width          int     `json:"width"`
 Seed           int64   `json:"seed"`
}

type TitanImageResponse struct {
 Images []string `json:"images"`
}

// Invokes the Titan Image model to create an image using the input provided
// in the request body.
func (wrapper InvokeModelWrapper) InvokeTitanImage(prompt string, seed int64)
 (string, error) {
 modelId := "amazon.titan-image-generator-v1"

 body, err := json.Marshal(TitanImageRequest{
  TaskType: "TEXT_IMAGE",
  TextToImageParams: TextToImageParams{
   Text: prompt,
  },
  ImageGenerationConfig: ImageGenerationConfig{
   NumberOfImages: 1,
   Quality:        "standard",
   CfgScale:       8.0,
   Height:         512,
   Width:          512,
   Seed:           seed,
  },
 })

 if err != nil {
  log.Fatal("failed to marshal", err)
 }
```

```
output, err := wrapper.BedrockRuntimeClient.InvokeModel(context.TODO(),
&bedrockruntime.InvokeModelInput{
 ModelId:     aws.String(modelId),
 ContentType: aws.String("application/json"),
 Body:        body,
})

if err != nil {
 ProcessError(err, modelId)
}

var response TitanImageResponse
if err := json.Unmarshal(output.Body, &response); err != nil {
 log.Fatal("failed to unmarshal", err)
}

base64ImageData := response.Images[0]

return base64ImageData, nil

}
```

- For API details, see InvokeModel in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Amazon Titan Image Generator G1 model to generate images.

```
    /**
     * Invokes the Amazon Titan image generation model to create an image using
 the
     * input
```

```
     * provided in the request body.
     *
     * @param prompt The prompt that you want Amazon Titan to use for image
     *               generation.
     * @param seed   The random noise seed for image generation (Range: 0 to
     *               2147483647).
     * @return A Base64-encoded string representing the generated image.
     */
    public static String invokeTitanImage(String prompt, long seed) {
        /*
         * The different model providers have individual request and response
 formats.
         * For the format, ranges, and default values for Titan Image models
 refer to:
         * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
titan-
         * image.html
         */
        String titanImageModelId = "amazon.titan-image-generator-v1";

        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_EAST_1)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();

        var textToImageParams = new JSONObject().put("text", prompt);

        var imageGenerationConfig = new JSONObject()
                .put("numberOfImages", 1)
                .put("quality", "standard")
                .put("cfgScale", 8.0)
                .put("height", 512)
                .put("width", 512)
                .put("seed", seed);

        JSONObject payload = new JSONObject()
                .put("taskType", "TEXT_IMAGE")
                .put("textToImageParams", textToImageParams)
                .put("imageGenerationConfig", imageGenerationConfig);

        InvokeModelRequest request = InvokeModelRequest.builder()
                .body(SdkBytes.fromUtf8String(payload.toString()))
                .modelId(titanImageModelId)
                .contentType("application/json")
```

```
                .accept("application/json")
                .build();

        CompletableFuture<InvokeModelResponse> completableFuture =
client.invokeModel(request)
                .whenComplete((response, exception) -> {
                    if (exception != null) {
                        System.out.println("Model invocation failed: " +
exception);
                    }
                });

        String base64ImageData = "";
        try {
            InvokeModelResponse response = completableFuture.get();
            JSONObject responseBody = new
JSONObject(response.body().asUtf8String());
            base64ImageData = responseBody
                    .getJSONArray("images")
                    .getString(0);

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
        }

        return base64ImageData;
    }
```

Invoke the Amazon Titan Image Generator G1 model to generate images.

```
        /**
         * Invokes the Amazon Titan image generation model to create an image
using the
         * input
         * provided in the request body.
         *
         * @param prompt The prompt that you want Amazon Titan to use for image
         *               generation.
         * @param seed   The random noise seed for image generation (Range: 0 to
```

```
         *                2147483647).
         * @return A Base64-encoded string representing the generated image.
         */
        public static String invokeTitanImage(String prompt, long seed) {
                /*
                 * The different model providers have individual request and
 response formats.
                 * For the format, ranges, and default values for Titan Image
 models refer to:
                 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-titan-
                 * image.html
                 */
                String titanImageModelId = "amazon.titan-image-generator-v1";

                BedrockRuntimeClient client = BedrockRuntimeClient.builder()
                                .region(Region.US_EAST_1)

 .credentialsProvider(ProfileCredentialsProvider.create())
                                .build();

                var textToImageParams = new JSONObject().put("text", prompt);

                var imageGenerationConfig = new JSONObject()
                                .put("numberOfImages", 1)
                                .put("quality", "standard")
                                .put("cfgScale", 8.0)
                                .put("height", 512)
                                .put("width", 512)
                                .put("seed", seed);

                JSONObject payload = new JSONObject()
                                .put("taskType", "TEXT_IMAGE")
                                .put("textToImageParams", textToImageParams)
                                .put("imageGenerationConfig",
 imageGenerationConfig);

                InvokeModelRequest request = InvokeModelRequest.builder()

 .body(SdkBytes.fromUtf8String(payload.toString()))
                                .modelId(titanImageModelId)
                                .contentType("application/json")
                                .accept("application/json")
                                .build();
```

```
                InvokeModelResponse response = client.invokeModel(request);

                JSONObject responseBody = new
 JSONObject(response.body().asUtf8String());

                String base64ImageData = responseBody
                                .getJSONArray("images")
                                .getString(0);

                return base64ImageData;
        }
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Amazon Titan Image Generator G1 model to generate images.

```
    public function invokeTitanImage(string $prompt, int $seed)
    {
        # The different model providers have individual request and response
 formats.
        # For the format, ranges, and default values for Titan Image models refer
 to:
        # https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
titan-image.html

        $base64_image_data = "";

        try {
            $modelId = 'amazon.titan-image-generator-v1';
```

```
            $request = json_encode([
                'taskType' => 'TEXT_IMAGE',
                'textToImageParams' => [
                    'text' => $prompt
                ],
                'imageGenerationConfig' => [
                    'numberOfImages' => 1,
                    'quality' => 'standard',
                    'cfgScale' => 8.0,
                    'height' => 512,
                    'width' => 512,
                    'seed' => $seed
                ]
            ]);

            $result = $this->bedrockRuntimeClient->invokeModel([
                'contentType' => 'application/json',
                'body' => $request,
                'modelId' => $modelId,
            ]);

            $response_body = json_decode($result['body']);

            $base64_image_data = $response_body->images[0];
        } catch (Exception $e) {
            echo "Error: ({$e->getCode()}) - {$e->getMessage()}\n";
        }

        return $base64_image_data;
    }
```

- For API details, see InvokeModel in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Amazon Titan Image Generator G1 model to generate images.

```
    def invoke_titan_image(self, prompt, seed):
        """
        Invokes the Titan Image model to create an image using the input provided
 in the request body.

        :param prompt: The prompt that you want Amazon Titan to use for image
 generation.
        :param seed: Random noise seed (range: 0 to 2147483647)
        :return: Base64-encoded inference response from the model.
        """

        try:
            # The different model providers have individual request and response
 formats.
            # For the format, ranges, and default values for Titan Image models
 refer to:
            # https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-titan-image.html

            request = json.dumps(
                {
                    "taskType": "TEXT_IMAGE",
                    "textToImageParams": {"text": prompt},
                    "imageGenerationConfig": {
                        "numberOfImages": 1,
                        "quality": "standard",
                        "cfgScale": 8.0,
                        "height": 512,
                        "width": 512,
                        "seed": seed,
                    },
                }
            )

            response = self.bedrock_runtime_client.invoke_model(
                modelId="amazon.titan-image-generator-v1", body=request
            )

            response_body = json.loads(response["body"].read())
            base64_image_data = response_body["images"][0]

            return base64_image_data
```

```
        except ClientError:
            logger.error("Couldn't invoke Titan Image generator")
            raise
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Amazon Titan Text G1 on Amazon Bedrock for text generation

The following code examples show how to invoke Amazon Titan Text G1 on Amazon Bedrock for text generation.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Amazon Titan Text G1 foundation model to generate text.

```
        /// <summary>
        /// Asynchronously invokes the Amazon Titan Text G1 Express model to run
 an inference based on the provided input.
        /// </summary>
        /// <param name="prompt">The prompt that you want Amazon Titan Text G1
 Express to complete.</param>
        /// <returns>The inference response from the model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
```

```
        /// For the format, ranges, and default values for Amazon Titan Text G1
 Express, refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-titan-text.html
        /// </remarks>
        public static async Task<string> InvokeTitanTextG1Async(string prompt)
        {
            string titanTextG1ModelId = "amazon.titan-text-express-v1";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            string payload = new JsonObject()
            {
                { "inputText", prompt },
                { "textGenerationConfig", new JsonObject()
                    {
                        { "maxTokenCount", 512 },
                        { "temperature", 0f },
                        { "topP", 1f }
                    }
                }
            }.ToJsonString();

            string generatedText = "";
            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
 InvokeModelRequest()
                {
                    ModelId = titanTextG1ModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    var results = JsonNode.ParseAsync(response.Body).Result?
["results"]?.AsArray();

                    return results is null ? "" : string.Join(" ",
 results.Select(x => x?["outputText"]?.GetValue<string?>()));
                }
                else
```

```
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
 + response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return generatedText;
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Amazon Titan Text G1 foundation model to generate text.

```
// Each model provider has their own individual request and response formats.
// For the format, ranges, and default values for Amazon Titan Text, refer to:
// https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-titan-
text.html
type TitanTextRequest struct {
 InputText            string               `json:"inputText"`
 TextGenerationConfig TextGenerationConfig `json:"textGenerationConfig"`
}

type TextGenerationConfig struct {
 Temperature   float64  `json:"temperature"`
 TopP          float64  `json:"topP"`
 MaxTokenCount int      `json:"maxTokenCount"`
```

```
 StopSequences []string `json:"stopSequences,omitempty"`
}

type TitanTextResponse struct {
 InputTextTokenCount int      `json:"inputTextTokenCount"`
 Results             []Result `json:"results"`
}

type Result struct {
 TokenCount       int    `json:"tokenCount"`
 OutputText       string `json:"outputText"`
 CompletionReason string `json:"completionReason"`
}

func (wrapper InvokeModelWrapper) InvokeTitanText(prompt string) (string, error)
 {
 modelId := "amazon.titan-text-express-v1"

 body, err := json.Marshal(TitanTextRequest{
  InputText: prompt,
  TextGenerationConfig: TextGenerationConfig{
   Temperature:   0,
   TopP:          1,
   MaxTokenCount: 4096,
  },
 })

 if err != nil {
  log.Fatal("failed to marshal", err)
 }

 output, err := wrapper.BedrockRuntimeClient.InvokeModel(context.Background(),
 &bedrockruntime.InvokeModelInput{
  ModelId:     aws.String(modelId),
  ContentType: aws.String("application/json"),
  Body:        body,
 })

 if err != nil {
  ProcessError(err, modelId)
 }

 var response TitanTextResponse
 if err := json.Unmarshal(output.Body, &response); err != nil {
```

```
  log.Fatal("failed to unmarshal", err)
 }

 return response.Results[0].OutputText, nil
}
```

- For API details, see InvokeModel in *AWS SDK for Go API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Amazon Titan Text G1 foundation model to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseBody
 * @property {Object[]} results
 */

/**
 * Invokes an Amazon Titan Text generation model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
```

```
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "amazon.titan-text-express-v1".
 */
export const invokeModel = async (
  prompt,
  modelId = "amazon.titan-text-express-v1",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    inputText: prompt,
    textGenerationConfig: {
      maxTokenCount: 4096,
      stopSequences: [],
      temperature: 0,
      topP: 1,
    },
  };

  // Invoke the model with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response.
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {ResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.results[0].outputText;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time..."';
  const modelId = FoundationModels.TITAN_TEXT_G1_EXPRESS.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);
```

```
  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Anthropic Claude 2.x on Amazon Bedrock and process the response stream in real-time

The following code examples show how to invoke Anthropic Claude 2.x on Amazon Bedrock and process the response stream in real-time.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke Anthropic Claude and process the response stream.

```
        /// <summary>
        /// Asynchronously invokes the Anthropic Claude 2 model to run an
 inference based on the provided input and process the response stream.
        /// </summary>
        /// <param name="prompt">The prompt that you want Claude to complete.</
param>
```

```
        /// <returns>The inference response from the model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Anthropic Claude,
 refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-claude.html
        /// </remarks>
        public static async IAsyncEnumerable<string>
 InvokeClaudeWithResponseStreamAsync(string prompt, [EnumeratorCancellation]
 CancellationToken cancellationToken = default)
        {
            string claudeModelId = "anthropic.claude-v2";

            // Claude requires you to enclose the prompt as follows:
            string enclosedPrompt = "Human: " + prompt + "\n\nAssistant:";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            string payload = new JsonObject()
             {
                 { "prompt", enclosedPrompt },
                 { "max_tokens_to_sample", 200 },
                 { "temperature", 0.5 },
                 { "stop_sequences", new JsonArray("\n\nHuman:") }
             }.ToJsonString();

            InvokeModelWithResponseStreamResponse? response = null;

            try
            {
                response = await client.InvokeModelWithResponseStreamAsync(new
 InvokeModelWithResponseStreamRequest()
                {
                    ModelId = claudeModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
```

```
            }

            if (response is not null && response.HttpStatusCode ==
System.Net.HttpStatusCode.OK)
            {
                // create a buffer to write the event in to move from a push mode
to a pull mode
                Channel<string> buffer = Channel.CreateUnbounded<string>();
                bool isStreaming = true;

                response.Body.ChunkReceived += BodyOnChunkReceived;
                response.Body.StartProcessing();

                while ((!cancellationToken.IsCancellationRequested
&& isStreaming) || (!cancellationToken.IsCancellationRequested &&
buffer.Reader.Count > 0))
                {
                    // pull the completion from the buffer and add it to the
IAsyncEnumerable collection
                    yield return await
buffer.Reader.ReadAsync(cancellationToken);
                }
                response.Body.ChunkReceived -= BodyOnChunkReceived;

                yield break;

                // handle the ChunkReceived events
                async void BodyOnChunkReceived(object? sender,
EventStreamEventReceivedArgs<PayloadPart> e)
                {
                    var streamResponse =
JsonSerializer.Deserialize<JsonObject>(e.EventStreamEvent.Bytes) ??
throw new NullReferenceException($"Unable to deserialize
{nameof(e.EventStreamEvent.Bytes)}");

                    if (streamResponse["stop_reason"]?.GetValue<string?>() !=
null)
                    {
                        isStreaming = false;
                    }

                    // write the received completion chunk into the buffer
```

```
                   await 
buffer.Writer.WriteAsync(streamResponse["completion"]?.GetValue<string>(), 
cancellationToken);
               }
           }
           else if (response is not null)
           {
               Console.WriteLine("InvokeModelAsync failed with status code " + 
response.HttpStatusCode);
           }

           yield break;
       }
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke Anthropic Claude and process the response stream.

```
// Each model provider defines their own individual request and response formats.
// For the format, ranges, and default values for the different models, refer to:
// https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters.html

type Request struct {
 Prompt            string  `json:"prompt"`
 MaxTokensToSample int     `json:"max_tokens_to_sample"`
 Temperature       float64 `json:"temperature,omitempty"`
}

type Response struct {
```

```
 Completion string `json:"completion"`
}

// Invokes Anthropic Claude on Amazon Bedrock to run an inference and
 asynchronously
// process the response stream.

func (wrapper InvokeModelWithResponseStreamWrapper)
 InvokeModelWithResponseStream(prompt string) (string, error) {

 modelId := "anthropic.claude-v2"

 // Anthropic Claude requires you to enclose the prompt as follows:
 prefix := "Human: "
 postfix := "\n\nAssistant:"
 prompt = prefix + prompt + postfix

 request := ClaudeRequest{
  Prompt:            prompt,
  MaxTokensToSample: 200,
  Temperature:       0.5,
  StopSequences:     []string{"\n\nHuman:"},
 }

 body, err := json.Marshal(request)
 if err != nil {
  log.Panicln("Couldn't marshal the request: ", err)
 }

 output, err :=
 wrapper.BedrockRuntimeClient.InvokeModelWithResponseStream(context.Background(),
 &bedrockruntime.InvokeModelWithResponseStreamInput{
  Body:        body,
  ModelId:     aws.String(modelId),
  ContentType: aws.String("application/json"),
 })

 if err != nil {
  errMsg := err.Error()
  if strings.Contains(errMsg, "no such host") {
   log.Printf("The Bedrock service is not available in the selected region.
 Please double-check the service availability for your region at https://
aws.amazon.com/about-aws/global-infrastructure/regional-product-services/.\n")
  } else if strings.Contains(errMsg, "Could not resolve the foundation model") {
```

```
   log.Printf("Could not resolve the foundation model from model identifier: \"%v
\". Please verify that the requested model exists and is accessible within the
 specified region.\n", modelId)
  } else {
   log.Printf("Couldn't invoke Anthropic Claude. Here's why: %v\n", err)
  }
 }

 resp, err := processStreamingOutput(output, func(ctx context.Context, part
 []byte) error {
  fmt.Print(string(part))
  return nil
 })

 if err != nil {
  log.Fatal("streaming output processing error: ", err)
 }

 return resp.Completion, nil

}

type StreamingOutputHandler func(ctx context.Context, part []byte) error

func processStreamingOutput(output
 *bedrockruntime.InvokeModelWithResponseStreamOutput, handler
 StreamingOutputHandler) (Response, error) {

 var combinedResult string
 resp := Response{}

 for event := range output.GetStream().Events() {
  switch v := event.(type) {
  case *types.ResponseStreamMemberChunk:

   //fmt.Println("payload", string(v.Value.Bytes))

   var resp Response
   err := json.NewDecoder(bytes.NewReader(v.Value.Bytes)).Decode(&resp)
   if err != nil {
    return resp, err
   }

   err = handler(context.Background(), []byte(resp.Completion))
```

```
   if err != nil {
    return resp, err
   }

   combinedResult += resp.Completion

  case *types.UnknownUnionMember:
   fmt.Println("unknown tag:", v.Tag)

  default:
   fmt.Println("union is nil or unknown type")
  }
 }

 resp.Completion = combinedResult

 return resp, nil
}
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Use Claude's Messages API to generate text, then process the response stream in real-time.

```
    /**
     * Invokes Anthropic Claude 2 via the Messages API and processes the response
 stream.
     * <p>
     * To learn more about the Anthropic Messages API, go to:
     * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
```

```
     *
     * @param prompt The prompt for the model to complete.
     * @return A JSON object containing the complete response along with some
metadata.
     */
    public static JSONObject invokeMessagesApiWithResponseStream(String prompt) {
        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .credentialsProvider(ProfileCredentialsProvider.create())
                .region(Region.US_EAST_1)
                .build();

        String modelId = "anthropic.claude-v2";

        // Prepare the JSON payload for the Messages API request
        var payload = new JSONObject()
                .put("anthropic_version", "bedrock-2023-05-31")
                .put("max_tokens", 1000)
                .append("messages", new JSONObject()
                        .put("role", "user")
                        .append("content", new JSONObject()
                                .put("type", "text")
                                .put("text", prompt)
                        ));

        // Create the request object using the payload and the model ID
        var request = InvokeModelWithResponseStreamRequest.builder()
                .contentType("application/json")
                .body(SdkBytes.fromUtf8String(payload.toString()))
                .modelId(modelId)
                .build();

        // Create a handler to print the stream in real-time and add metadata to
a response object
        JSONObject structuredResponse = new JSONObject();
        var handler = createMessagesApiResponseStreamHandler(structuredResponse);

        // Invoke the model with the request payload and the response stream
handler
        client.invokeModelWithResponseStream(request, handler).join();

        return structuredResponse;
    }
```

```
    private static InvokeModelWithResponseStreamResponseHandler
 createMessagesApiResponseStreamHandler(JSONObject structuredResponse) {
        AtomicReference<String> completeMessage = new AtomicReference<>("");

        Consumer<ResponseStream> responseStreamHandler = event ->
 event.accept(InvokeModelWithResponseStreamResponseHandler.Visitor.builder()
                .onChunk(c -> {
                    // Decode the chunk
                    var chunk = new JSONObject(c.bytes().asUtf8String());

                    // The Messages API returns different types:
                    var chunkType = chunk.getString("type");
                    if ("message_start".equals(chunkType)) {
                        // The first chunk contains information about the message
 role
                        String role =
 chunk.optJSONObject("message").optString("role");
                        structuredResponse.put("role", role);

                    } else if ("content_block_delta".equals(chunkType)) {
                        // These chunks contain the text fragments
                        var text =
 chunk.optJSONObject("delta").optString("text");
                        // Print the text fragment to the console ...
                        System.out.print(text);
                        // ... and append it to the complete message
                        completeMessage.getAndUpdate(current -> current + text);

                    } else if ("message_delta".equals(chunkType)) {
                        // This chunk contains the stop reason
                        var stopReason =
 chunk.optJSONObject("delta").optString("stop_reason");
                        structuredResponse.put("stop_reason", stopReason);

                    } else if ("message_stop".equals(chunkType)) {
                        // The last chunk contains the metrics
                        JSONObject metrics = chunk.optJSONObject("amazon-bedrock-
invocationMetrics");
                        structuredResponse.put("metrics", new JSONObject()
                                .put("inputTokenCount",
 metrics.optString("inputTokenCount"))
                                .put("outputTokenCount",
 metrics.optString("outputTokenCount"))
```

```
                                .put("firstByteLatency",
 metrics.optString("firstByteLatency"))
                                .put("invocationLatency",
 metrics.optString("invocationLatency")));
                    }
                })
                .build());

        return InvokeModelWithResponseStreamResponseHandler.builder()
                .onEventStream(stream -> stream.subscribe(responseStreamHandler))
                .onComplete(() ->
                        // Add the complete message to the response object
                        structuredResponse.append("content", new JSONObject()
                                .put("type", "text")
                                .put("text", completeMessage.get())))
                .build();
    }
```

Use Claude's Text Completions API to generate text, then process the response stream in real-time.

```
    /**
     * Invokes Anthropic Claude 2 via the Text Completions API and processes the
 response stream.
     * <p>
     * To learn more about the Anthropic Text Completions API, go to:
     * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-text-completion.html
     *
     * @param prompt The prompt for the model to complete.
     * @return The generated response.
     */
    public static String invokeTextCompletionsApiWithResponseStream(String
 prompt) {
        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .credentialsProvider(ProfileCredentialsProvider.create())
                .region(Region.US_EAST_1)
                .build();

        String modelId = "anthropic.claude-v2";

        var payload = new JSONObject()
```

```
                .put("prompt", "Human: " + prompt + " Assistant:")
                .put("temperature", 0.5)
                .put("max_tokens_to_sample", 1000)
                .toString();

        var request = InvokeModelWithResponseStreamRequest.builder()
                .body(SdkBytes.fromUtf8String(payload))
                .contentType("application/json")
                .modelId(modelId)
                .build();

        var finalCompletion = new AtomicReference<>("");
        var visitor =
InvokeModelWithResponseStreamResponseHandler.Visitor.builder()
                .onChunk(chunk -> {
                    var json = new JSONObject(chunk.bytes().asUtf8String());
                    var completion = json.getString("completion");
                    finalCompletion.set(finalCompletion.get() + completion);
                    System.out.print(completion);
                })
                .build();

        var handler = InvokeModelWithResponseStreamResponseHandler.builder()
                .onEventStream(stream -> stream.subscribe(event ->
event.accept(visitor)))
                .onComplete(() -> {
                })
                .onError(e -> System.out.println("\n\nError: " + e.getMessage()))
                .build();

        client.invokeModelWithResponseStream(request, handler).join();

        return finalCompletion.get();
    }
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

Invoke Anthropic Claude and process the response stream.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseContent
 * @property {string} text
 *
 * @typedef {Object} MessagesResponseBody
 * @property {ResponseContent[]} content
 *
 * @typedef {Object} Delta
 * @property {string} text
 *
 * @typedef {Object} Message
 * @property {string} role
 *
 * @typedef {Object} Chunk
 * @property {string} type
 * @property {Delta} delta
 * @property {Message} message
 */
```

```
/**
 * Invokes Anthropic Claude 3 using the Messages API.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-3-haiku-20240307-v1:0".
 */
export const invokeModel = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {MessagesResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.content[0].text;
};
```

```
/**
 * Invokes Anthropic Claude 3 and processes the response stream.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-3-haiku-20240307-v1:0".
 */
export const invokeModelWithResponseStream = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the API to respond.
  const command = new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  let completeMessage = "";

  // Decode and process the response stream
  for await (const item of apiResponse.body) {
    /** @type Chunk */
```

```
    const chunk = JSON.parse(new TextDecoder().decode(item.chunk.bytes));
    const chunk_type = chunk.type;

    if (chunk_type === "content_block_delta") {
      const text = chunk.delta.text;
      completeMessage = completeMessage + text;
      process.stdout.write(text);
    }
  }

  // Return the final response
  return completeMessage;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt = 'Write a paragraph starting with: "Once upon a time..."';
  const modelId = FoundationModels.CLAUDE_3_HAIKU.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log("\n" + "-".repeat(53));
    console.log("Final structured response:");
    console.log(response);
  } catch (err) {
    console.log(`\n${err}`);
  }
}
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

Invoke Anthropic Claude and process the response stream.

```
    async def invoke_model_with_response_stream(self, prompt):
        """
        Invokes the Anthropic Claude 2 model to run an inference and process the
 response stream.

        :param prompt: The prompt that you want Claude to complete.
        :return: Inference response from the model.
        """

        try:
            # The different model providers have individual request and response
 formats.
            # For the format, ranges, and default values for Anthropic Claude,
 refer to:
            # https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-claude.html

            # Claude requires you to enclose the prompt as follows:
            enclosed_prompt = "Human: " + prompt + "\n\nAssistant:"

            body = {
                "prompt": enclosed_prompt,
                "max_tokens_to_sample": 1024,
                "temperature": 0.5,
                "stop_sequences": ["\n\nHuman:"],
            }

            response =
 self.bedrock_runtime_client.invoke_model_with_response_stream(
                modelId="anthropic.claude-v2", body=json.dumps(body)
            )
```

```
            for event in response.get("body"):
                chunk = json.loads(event["chunk"]["bytes"])["completion"]
                yield chunk

        except ClientError:
            logger.error("Couldn't invoke Anthropic Claude v2")
            raise
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Anthropic Claude 2.x on Amazon Bedrock for text generation

The following code examples show how to invoke the Anthropic Claude 2.x on Amazon Bedrock for text generation.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Anthropic Claude 2 foundation model to generate text.

```
        /// <summary>
        /// Asynchronously invokes the Anthropic Claude 2 model to run an
 inference based on the provided input.
        /// </summary>
```

```
        /// <param name="prompt">The prompt that you want Claude to complete.</
param>
        /// <returns>The inference response from the model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Anthropic Claude,
 refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-claude.html
        /// </remarks>
        public static async Task<string> InvokeClaudeAsync(string prompt)
        {
            string claudeModelId = "anthropic.claude-v2";

            // Claude requires you to enclose the prompt as follows:
            string enclosedPrompt = "Human: " + prompt + "\n\nAssistant:";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            string payload = new JsonObject()
            {
                { "prompt", enclosedPrompt },
                { "max_tokens_to_sample", 200 },
                { "temperature", 0.5 },
                { "stop_sequences", new JsonArray("\n\nHuman:") }
            }.ToJsonString();

            string generatedText = "";
            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
 InvokeModelRequest()
                {
                    ModelId = claudeModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    return JsonNode.ParseAsync(response.Body).Result?
["completion"]?.GetValue<string>() ?? "";
```

```
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
 + response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return generatedText;
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Anthropic Claude 2 foundation model to generate text.

```
// Each model provider has their own individual request and response formats.
// For the format, ranges, and default values for Anthropic Claude, refer to:
// https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
claude.html

type ClaudeRequest struct {
 Prompt            string   `json:"prompt"`
 MaxTokensToSample int      `json:"max_tokens_to_sample"`
 Temperature       float64  `json:"temperature,omitempty"`
 StopSequences     []string `json:"stop_sequences,omitempty"`
}
```

```
type ClaudeResponse struct {
 Completion string `json:"completion"`
}

// Invokes Anthropic Claude on Amazon Bedrock to run an inference using the input
// provided in the request body.
func (wrapper InvokeModelWrapper) InvokeClaude(prompt string) (string, error) {
 modelId := "anthropic.claude-v2"

 // Anthropic Claude requires enclosing the prompt as follows:
 enclosedPrompt := "Human: " + prompt + "\n\nAssistant:"

 body, err := json.Marshal(ClaudeRequest{
  Prompt:            enclosedPrompt,
  MaxTokensToSample: 200,
  Temperature:       0.5,
  StopSequences:     []string{"\n\nHuman:"},
 })

 if err != nil {
  log.Fatal("failed to marshal", err)
 }

 output, err := wrapper.BedrockRuntimeClient.InvokeModel(context.TODO(),
 &bedrockruntime.InvokeModelInput{
  ModelId:     aws.String(modelId),
  ContentType: aws.String("application/json"),
  Body:        body,
 })

 if err != nil {
  ProcessError(err, modelId)
 }

 var response ClaudeResponse
 if err := json.Unmarshal(output.Body, &response); err != nil {
  log.Fatal("failed to unmarshal", err)
 }

 return response.Completion, nil
}
```

- For API details, see InvokeModel in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke Claude 2.x using the synchronous client (scroll down for an async example).

```
        /**
         * Invokes the Anthropic Claude 2 model to run an inference based on the
         * provided input.
         *
         * @param prompt The prompt for Claude to complete.
         * @return The generated response.
         */
        public static String invokeClaude(String prompt) {
                /*
                 * The different model providers have individual request and
 response formats.
                 * For the format, ranges, and default values for Anthropic
 Claude, refer to:
                 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-claude.html
                 */

                String claudeModelId = "anthropic.claude-v2";

                // Claude requires you to enclose the prompt as follows:
                String enclosedPrompt = "Human: " + prompt + "\n\nAssistant:";

                BedrockRuntimeClient client = BedrockRuntimeClient.builder()
                                .region(Region.US_EAST_1)

 .credentialsProvider(ProfileCredentialsProvider.create())
```

```
                                .build();

                String payload = new JSONObject()
                                .put("prompt", enclosedPrompt)
                                .put("max_tokens_to_sample", 200)
                                .put("temperature", 0.5)
                                .put("stop_sequences", List.of("\n\nHuman:"))
                                .toString();

                InvokeModelRequest request = InvokeModelRequest.builder()
                                .body(SdkBytes.fromUtf8String(payload))
                                .modelId(claudeModelId)
                                .contentType("application/json")
                                .accept("application/json")
                                .build();

                InvokeModelResponse response = client.invokeModel(request);

                JSONObject responseBody = new
JSONObject(response.body().asUtf8String());

                String generatedText = responseBody.getString("completion");

                return generatedText;
        }
```

Invoke Claude 2.x using the asynchronous client.

```
    /**
     * Asynchronously invokes the Anthropic Claude 2 model to run an inference
based
     * on the provided input.
     *
     * @param prompt The prompt that you want Claude to complete.
     * @return The inference response from the model.
     */
    public static String invokeClaude(String prompt) {
        /*
         * The different model providers have individual request and response
formats.
         * For the format, ranges, and default values for Anthropic Claude, refer
to:
```

```
         * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
claude.html
         */

        String claudeModelId = "anthropic.claude-v2";

        // Claude requires you to enclose the prompt as follows:
        String enclosedPrompt = "Human: " + prompt + "\n\nAssistant:";

        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_EAST_1)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();

        String payload = new JSONObject()
                .put("prompt", enclosedPrompt)
                .put("max_tokens_to_sample", 200)
                .put("temperature", 0.5)
                .put("stop_sequences", List.of("\n\nHuman:"))
                .toString();

        InvokeModelRequest request = InvokeModelRequest.builder()
                .body(SdkBytes.fromUtf8String(payload))
                .modelId(claudeModelId)
                .contentType("application/json")
                .accept("application/json")
                .build();

        CompletableFuture<InvokeModelResponse> completableFuture =
 client.invokeModel(request)
                .whenComplete((response, exception) -> {
                    if (exception != null) {
                        System.out.println("Model invocation failed: " +
 exception);
                    }
                });

        String generatedText = "";
        try {
            InvokeModelResponse response = completableFuture.get();
            JSONObject responseBody = new
 JSONObject(response.body().asUtf8String());
            generatedText = responseBody.getString("completion");
        } catch (InterruptedException e) {
```

```
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
        }

        return generatedText;
    }
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Anthropic Claude 2 foundation model to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseContent
 * @property {string} text
 */

/**
 * @typedef {Object} MessagesResponseBody
```

```
 * @property {ResponseContent[]} content
 */

/**
 * @typedef {Object} TextCompletionsResponseBody
 * @property {completion} text
 */

/**
 * Invokes Anthropic Claude 2.x using the Messages API.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-v2".
 */
export const invokeModel = async (prompt, modelId = "anthropic.claude-v2") => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the Messages API request.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s)
```

```
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {MessagesResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.content[0].text;
};

/**
 * Invokes Anthropic Claude 2.x using the Text Completions API.
 *
 * To learn more about the Anthropic Text Completions API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-text-completion.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-v2".
 */
export const invokeTextCompletionsApi = async (
  prompt,
  modelId = "anthropic.claude-v2",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the Text Completions API, using the required prompt
 template.
  const enclosedPrompt = `Human: ${prompt}\n\nAssistant:`;
  const payload = {
    prompt: enclosedPrompt,
    max_tokens_to_sample: 500,
    temperature: 0.5,
    stop_sequences: ["\n\nHuman:"],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response.
  const decoded = new TextDecoder().decode(apiResponse.body);
```

```
  /** @type {TextCompletionsResponseBody} */
  const responseBody = JSON.parse(decoded);
  return responseBody.completion;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time..."';
  const modelId = FoundationModels.CLAUDE_2.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    console.log("Using the Messages API:");
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }

  try {
    console.log("-".repeat(53));
    console.log("Using the Text Completions API:");
    const response = await invokeTextCompletionsApi(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Anthropic Claude 2 foundation model to generate text.

```
    public function invokeClaude($prompt)
    {
        # The different model providers have individual request and response
 formats.
        # For the format, ranges, and default values for Anthropic Claude, refer
 to:
        # https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
claude.html

        $completion = "";

        try {
            $modelId = 'anthropic.claude-v2';

            # Claude requires you to enclose the prompt as follows:
            $prompt = "\n\nHuman: {$prompt}\n\nAssistant:";

            $body = [
                'prompt' => $prompt,
                'max_tokens_to_sample' => 200,
                'temperature' => 0.5,
                'stop_sequences' => ["\n\nHuman:"],
            ];

            $result = $this->bedrockRuntimeClient->invokeModel([
                'contentType' => 'application/json',
                'body' => json_encode($body),
                'modelId' => $modelId,
            ]);

            $response_body = json_decode($result['body']);
```

```
            $completion = $response_body->completion;
        } catch (Exception $e) {
            echo "Error: ({$e->getCode()}) - {$e->getMessage()}\n";
        }

        return $completion;
    }
```

- For API details, see [InvokeModel] in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository].

Invoke the Anthropic Claude 2 foundation model to generate text.

```
    def invoke_claude(self, prompt):
        """
        Invokes the Anthropic Claude 2 model to run an inference using the input
        provided in the request body.

        :param prompt: The prompt that you want Claude to complete.
        :return: Inference response from the model.
        """

        try:
            # The different model providers have individual request and response
 formats.
            # For the format, ranges, and default values for Anthropic Claude,
 refer to:
            # https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-claude.html

            # Claude requires you to enclose the prompt as follows:
```

```
            enclosed_prompt = "Human: " + prompt + "\n\nAssistant:"

            body = {
                "prompt": enclosed_prompt,
                "max_tokens_to_sample": 200,
                "temperature": 0.5,
                "stop_sequences": ["\n\nHuman:"],
            }

            response = self.bedrock_runtime_client.invoke_model(
                modelId="anthropic.claude-v2", body=json.dumps(body)
            )

            response_body = json.loads(response["body"].read())
            completion = response_body["completion"]

            return completion

        except ClientError:
            logger.error("Couldn't invoke Anthropic Claude")
            raise
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Anthropic Claude 2 foundation model to generate text. This example uses features of /US2/CL_JSON which might not be available on some NetWeaver versions.

```
    "Claude V2 Input Parameters should be in a format like this:
*   {
*     "prompt":"\n\nHuman:\\nTell me a joke\n\nAssistant:\n",
```

```
*     "max_tokens_to_sample":2048,
*     "temperature":0.5,
*     "top_k":250,
*     "top_p":1.0,
*     "stop_sequences":[]
*   }

    DATA: BEGIN OF ls_input,
            prompt               TYPE string,
            max_tokens_to_sample TYPE /aws1/rt_shape_integer,
            temperature          TYPE /aws1/rt_shape_float,
            top_k                TYPE /aws1/rt_shape_integer,
            top_p                TYPE /aws1/rt_shape_float,
            stop_sequences       TYPE /aws1/rt_stringtab,
          END OF ls_input.

    "Leave ls_input-stop_sequences empty.
    ls_input-prompt = |\n\nHuman:\\n{ iv_prompt }\n\nAssistant:\n|.
    ls_input-max_tokens_to_sample = 2048.
    ls_input-temperature = '0.5'.
    ls_input-top_k = 250.
    ls_input-top_p = 1.

    "Serialize into JSON with /ui2/cl_json -- this assumes SAP_UI is installed.
    DATA(lv_json) = /ui2/cl_json=>serialize(
      data = ls_input
                pretty_name   = /ui2/cl_json=>pretty_mode-low_case ).

    TRY.
        DATA(lo_response) = lo_bdr->invokemodel(
          iv_body = /aws1/cl_rt_util=>string_to_xstring( lv_json )
          iv_modelid = 'anthropic.claude-v2'
          iv_accept = 'application/json'
          iv_contenttype = 'application/json' ).

        "Claude V2 Response format will be:
*       {
*         "completion": "Knock Knock...",
*         "stop_reason": "stop_sequence"
*       }
        DATA: BEGIN OF ls_response,
                completion  TYPE string,
                stop_reason TYPE string,
              END OF ls_response.
```

```
        /ui2/cl_json=>deserialize(
          EXPORTING jsonx = lo_response->get_body( )
                    pretty_name = /ui2/cl_json=>pretty_mode-camel_case
          CHANGING  data  = ls_response ).

        DATA(lv_answer) = ls_response-completion.
      CATCH /aws1/cx_bdraccessdeniedex INTO DATA(lo_ex).
        WRITE / lo_ex->get_text( ).
        WRITE / |Don't forget to enable model access at https://
console.aws.amazon.com/bedrock/home?#/modelaccess|.

    ENDTRY.
```

Invoke the Anthropic Claude 2 foundation model to generate text using L2 high level client.

```
    TRY.
        DATA(lo_bdr_l2_claude) = /aws1/
cl_bdr_l2_factory=>create_claude_2( lo_bdr ).
        " iv_prompt can contain a prompt like 'tell me a joke about Java
 programmers'.
        DATA(lv_answer) = lo_bdr_l2_claude->prompt_for_text( iv_prompt ).
      CATCH /aws1/cx_bdraccessdeniedex INTO DATA(lo_ex).
        WRITE / lo_ex->get_text( ).
        WRITE / |Don't forget to enable model access at https://
console.aws.amazon.com/bedrock/home?#/modelaccess|.

    ENDTRY.
```

- For API details, see InvokeModel in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Anthropic Claude 3 on Amazon Bedrock with a multimodal prompt

The following code example shows how to invoke Anthropic Claude 3 on Amazon Bedrock with a multimodal prompt.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/python/example_code/bedrock-runtime#code-examples).

Invoke Anthropic Claude 3 with a multimodal prompt to analyze an image.

```
    def invoke_claude_3_multimodal(self, prompt, base64_image_data):
        """
        Invokes Anthropic Claude 3 Sonnet to run a multimodal inference using the
 input
        provided in the request body.

        :param prompt:            The prompt that you want Claude 3 to use.
        :param base64_image_data: The base64-encoded image that you want to add
 to the request.
        :return: Inference response from the model.
        """

        # Initialize the Amazon Bedrock runtime client
        client = self.client or boto3.client(
            service_name="bedrock-runtime", region_name="us-east-1"
        )

        # Invoke the model with the prompt and the encoded image
        model_id = "anthropic.claude-3-sonnet-20240229-v1:0"
        request_body = {
            "anthropic_version": "bedrock-2023-05-31",
            "max_tokens": 2048,
            "messages": [
                {
                    "role": "user",
                    "content": [
                        {
                            "type": "text",
                            "text": prompt,
                        },
                        {
```

```
                            "type": "image",
                            "source": {
                                "type": "base64",
                                "media_type": "image/png",
                                "data": base64_image_data,
                            },
                        },
                    ],
                }
            ],
        }

        try:
            response = client.invoke_model(
                modelId=model_id,
                body=json.dumps(request_body),
            )

            # Process and print the response
            result = json.loads(response.get("body").read())
            input_tokens = result["usage"]["input_tokens"]
            output_tokens = result["usage"]["output_tokens"]
            output_list = result.get("content", [])

            print("Invocation details:")
            print(f"- The input length is {input_tokens} tokens.")
            print(f"- The output length is {output_tokens} tokens.")

            print(f"- The model returned {len(output_list)} response(s):")
            for output in output_list:
                print(output["text"])

            return result
        except ClientError as err:
            logger.error(
                "Couldn't invoke Claude 3 Sonnet. Here's why: %s: %s",
                err.response["Error"]["Code"],
                err.response["Error"]["Message"],
            )
            raise
```

- For API details, see [InvokeModel](#) in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Anthropic Claude 3 on Amazon Bedrock and process the response stream in real-time

The following code example shows how to invoke Anthropic Claude 3 on Amazon Bedrock and process the response stream in real-time.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke Claude 3 to generate text, then process the response stream in real-time.

```
    /**
     * Invokes Anthropic Claude 3 Haiku and processes the response stream.
     *
     * @param prompt The prompt for the model to complete.
     * @return A JSON object containing the complete response along with some
 metadata.
     */
    public static JSONObject invokeModelWithResponseStream(String prompt) {
        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .credentialsProvider(ProfileCredentialsProvider.create())
                .region(Region.US_EAST_1)
                .build();

        String modelId = "anthropic.claude-3-haiku-20240307-v1:0";

        // Prepare the JSON payload for the Messages API request
        var payload = new JSONObject()
                .put("anthropic_version", "bedrock-2023-05-31")
                .put("max_tokens", 1000)
```

```
                .append("messages", new JSONObject()
                        .put("role", "user")
                        .append("content", new JSONObject()
                                .put("type", "text")
                                .put("text", prompt)
                        ));

        // Create the request object using the payload and the model ID
        var request = InvokeModelWithResponseStreamRequest.builder()
                .contentType("application/json")
                .body(SdkBytes.fromUtf8String(payload.toString()))
                .modelId(modelId)
                .build();

        // Create a handler to print the stream in real-time and add metadata to
a response object
        JSONObject structuredResponse = new JSONObject();
        var handler = createMessagesApiResponseStreamHandler(structuredResponse);

        // Invoke the model with the request payload and the response stream
handler
        client.invokeModelWithResponseStream(request, handler).join();

        return structuredResponse;
    }

    private static InvokeModelWithResponseStreamResponseHandler
createMessagesApiResponseStreamHandler(JSONObject structuredResponse) {
        AtomicReference<String> completeMessage = new AtomicReference<>("");

        Consumer<ResponseStream> responseStreamHandler = event ->
event.accept(InvokeModelWithResponseStreamResponseHandler.Visitor.builder()
                .onChunk(c -> {
                    // Decode the chunk
                    var chunk = new JSONObject(c.bytes().asUtf8String());

                    // The Messages API returns different types:
                    var chunkType = chunk.getString("type");
                    if ("message_start".equals(chunkType)) {
                        // The first chunk contains information about the message
role
                        String role =
chunk.optJSONObject("message").optString("role");
                        structuredResponse.put("role", role);
```

```
                    } else if ("content_block_delta".equals(chunkType)) {
                        // These chunks contain the text fragments
                        var text =
 chunk.optJSONObject("delta").optString("text");
                        // Print the text fragment to the console ...
                        System.out.print(text);
                        // ... and append it to the complete message
                        completeMessage.getAndUpdate(current -> current + text);

                    } else if ("message_delta".equals(chunkType)) {
                        // This chunk contains the stop reason
                        var stopReason =
 chunk.optJSONObject("delta").optString("stop_reason");
                        structuredResponse.put("stop_reason", stopReason);

                    } else if ("message_stop".equals(chunkType)) {
                        // The last chunk contains the metrics
                        JSONObject metrics = chunk.optJSONObject("amazon-bedrock-
invocationMetrics");
                        structuredResponse.put("metrics", new JSONObject()
                                .put("inputTokenCount",
 metrics.optString("inputTokenCount"))
                                .put("outputTokenCount",
 metrics.optString("outputTokenCount"))
                                .put("firstByteLatency",
 metrics.optString("firstByteLatency"))
                                .put("invocationLatency",
 metrics.optString("invocationLatency")));
                    }
                })
                .build());

        return InvokeModelWithResponseStreamResponseHandler.builder()
                .onEventStream(stream -> stream.subscribe(responseStreamHandler))
                .onComplete(() ->
                        // Add the complete message to the response object
                        structuredResponse.append("content", new JSONObject()
                                .put("type", "text")
                                .put("text", completeMessage.get())))
                .build();
    }
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Java 2.x API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Anthropic Claude 3 on Amazon Bedrock to generate text

The following code examples show how to invoke Anthropic Claude 3 on Amazon Bedrock to generate text.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke Anthropic Claude 3 to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} ResponseContent
 * @property {string} text
 *
 * @typedef {Object} MessagesResponseBody
```

```
 * @property {ResponseContent[]} content
 *
 * @typedef {Object} Delta
 * @property {string} text
 *
 * @typedef {Object} Message
 * @property {string} role
 *
 * @typedef {Object} Chunk
 * @property {string} type
 * @property {Delta} delta
 * @property {Message} message
 */

/**
 * Invokes Anthropic Claude 3 using the Messages API.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-3-haiku-20240307-v1:0".
 */
export const invokeModel = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };
```

```
  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {MessagesResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.content[0].text;
};

/**
 * Invokes Anthropic Claude 3 and processes the response stream.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-3-haiku-20240307-v1:0".
 */
export const invokeModelWithResponseStream = async (
  prompt,
  modelId = "anthropic.claude-3-haiku-20240307-v1:0",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the model.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };
```

```
  // Invoke Claude with the payload and wait for the API to respond.
  const command = new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  let completeMessage = "";

  // Decode and process the response stream
  for await (const item of apiResponse.body) {
    /** @type Chunk */
    const chunk = JSON.parse(new TextDecoder().decode(item.chunk.bytes));
    const chunk_type = chunk.type;

    if (chunk_type === "content_block_delta") {
      const text = chunk.delta.text;
      completeMessage = completeMessage + text;
      process.stdout.write(text);
    }
  }

  // Return the final response
  return completeMessage;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt = 'Write a paragraph starting with: "Once upon a time..."';
  const modelId = FoundationModels.CLAUDE_3_HAIKU.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log("\n" + "-".repeat(53));
    console.log("Final structured response:");
    console.log(response);
  } catch (err) {
    console.log(`\n${err}`);
  }
```

```
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke Anthropic Claude 3 to generate text.

```
    def invoke_claude_3_with_text(self, prompt):
        """
        Invokes Anthropic Claude 3 Sonnet to run an inference using the input
        provided in the request body.

        :param prompt: The prompt that you want Claude 3 to complete.
        :return: Inference response from the model.
        """

        # Initialize the Amazon Bedrock runtime client
        client = self.client or boto3.client(
            service_name="bedrock-runtime", region_name="us-east-1"
        )

        # Invoke Claude 3 with the text prompt
        model_id = "anthropic.claude-3-sonnet-20240229-v1:0"

        try:
            response = client.invoke_model(
                modelId=model_id,
                body=json.dumps(
                    {
                        "anthropic_version": "bedrock-2023-05-31",
                        "max_tokens": 1024,
                        "messages": [
```

```
                    {
                        "role": "user",
                        "content": [{"type": "text", "text": prompt}],
                    }
                ],
            }
        ),
    )

    # Process and print the response
    result = json.loads(response.get("body").read())
    input_tokens = result["usage"]["input_tokens"]
    output_tokens = result["usage"]["output_tokens"]
    output_list = result.get("content", [])

    print("Invocation details:")
    print(f"- The input length is {input_tokens} tokens.")
    print(f"- The output length is {output_tokens} tokens.")

    print(f"- The model returned {len(output_list)} response(s):")
    for output in output_list:
        print(output["text"])

    return result

except ClientError as err:
    logger.error(
        "Couldn't invoke Claude 3 Sonnet. Here's why: %s: %s",
        err.response["Error"]["Code"],
        err.response["Error"]["Message"],
    )
    raise
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke the Anthropic Claude Instant model on Amazon Bedrock for text generation

The following code example shows how to invoke the Anthropic Claude Instant model on Amazon Bedrock for text generation.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples).

Invoke the Anthropic Claude Instant foundation model to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} Content
 * @property {string} text
 *
 * @typedef {Object} MessageApiResponse
 * @property {Content[]} content
 */

/**
 * @typedef {Object} TextCompletionApiResponse
 * @property {string} completion
 */
```

```
/**
 * Invokes Anthropic Claude Instant using the Messages API.
 *
 * To learn more about the Anthropic Messages API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-messages.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-instant-v1".
 */
export const invokeModel = async (
  prompt,
  modelId = "anthropic.claude-instant-v1",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the Messages API request.
  const payload = {
    anthropic_version: "bedrock-2023-05-31",
    max_tokens: 1000,
    messages: [
      {
        role: "user",
        content: [{ type: "text", text: prompt }],
      },
    ],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response(s)
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {MessageApiResponse} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.content[0].text;
};
```

```
/**
 * Invokes Anthropic Claude Instant using the Text Completions API.
 *
 * To learn more about the Anthropic Text Completions API, go to:
 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
anthropic-claude-text-completion.html
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "anthropic.claude-instant-v1".
 */
export const invokeTextCompletionsApi = async (
  prompt,
  modelId = "anthropic.claude-instant-v1",
) => {
  // Create a new Bedrock Runtime client instance.
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Prepare the payload for the Text Completions API, using the required prompt
 template.
  const enclosedPrompt = `Human: ${prompt}\n\nAssistant:`;
  const payload = {
    prompt: enclosedPrompt,
    max_tokens_to_sample: 500,
    temperature: 0.5,
    stop_sequences: ["\n\nHuman:"],
  };

  // Invoke Claude with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response.
  const decoded = new TextDecoder().decode(apiResponse.body);
  /** @type {TextCompletionApiResponse} */
  const responseBody = JSON.parse(decoded);
  return responseBody.completion;
};
```

```
// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time..."';
  const modelId = FoundationModels.CLAUDE_INSTANT.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    console.log("Using the Messages API:");
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }

  try {
    console.log("-".repeat(53));
    console.log("Using the Text Completions API:");
    const response = await invokeTextCompletionsApi(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Meta Llama 2 on Amazon Bedrock using Meta's native request and response payloads

The following code examples show how to get started sending prompts to Meta Llama 2 and printing the response.

.NET

AWS SDK for .NET

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/dotnetv3/Bedrock-runtime#code-examples).

Asynchronously invoke the Meta Llama 2 foundation model to generate text.

```
        /// <summary>
        /// Asynchronously invokes the Meta Llama 2 Chat model to run an
 inference based on the provided input.
        /// </summary>
        /// <param name="prompt">The prompt that you want Llama 2 to complete.</
param>
        /// <returns>The inference response from the model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Meta Llama 2 Chat,
 refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-meta.html
        /// </remarks>
        public static async Task<string> InvokeLlama2Async(string prompt)
        {
            string llama2ModelId = "meta.llama2-13b-chat-v1";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            string payload = new JsonObject()
            {
                { "prompt", prompt },
                { "max_gen_len", 512 },
                { "temperature", 0.5 },
                { "top_p", 0.9 }
            }.ToJsonString();

            string generatedText = "";
```

```
            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new 
 InvokeModelRequest()
                {
                    ModelId = llama2ModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    return JsonNode.ParseAsync(response.Body)
                        .Result?["generation"]?.GetValue<string>() ?? "";
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code " 
 + response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return generatedText;
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Meta Llama 2 Chat foundation model to generate text.

```
// Each model provider has their own individual request and response formats.
// For the format, ranges, and default values for Meta Llama 2 Chat, refer to:
// https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
meta.html

type Llama2Request struct {
 Prompt       string  `json:"prompt"`
 MaxGenLength int     `json:"max_gen_len,omitempty"`
 Temperature  float64 `json:"temperature,omitempty"`
}

type Llama2Response struct {
 Generation string `json:"generation"`
}

// Invokes Meta Llama 2 Chat on Amazon Bedrock to run an inference using the
 input
// provided in the request body.
func (wrapper InvokeModelWrapper) InvokeLlama2(prompt string) (string, error) {
 modelId := "meta.llama2-13b-chat-v1"

 body, err := json.Marshal(Llama2Request{
  Prompt:       prompt,
  MaxGenLength: 512,
  Temperature:  0.5,
 })

 if err != nil {
  log.Fatal("failed to marshal", err)
 }

 output, err := wrapper.BedrockRuntimeClient.InvokeModel(context.TODO(),
 &bedrockruntime.InvokeModelInput{
  ModelId:     aws.String(modelId),
  ContentType: aws.String("application/json"),
  Body:        body,
 })

 if err != nil {
  ProcessError(err, modelId)
 }
```

```
    var response Llama2Response
    if err := json.Unmarshal(output.Body, &response); err != nil {
     log.Fatal("failed to unmarshal", err)
    }

    return response.Generation, nil
}
```

- For API details, see [InvokeModel](#) in *AWS SDK for Go API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](#).

Send your first prompt to Meta Llama 2.

```
// Send a prompt to Meta Llama 2 and print the response.
public class InvokeModelQuickstart {

    public static void main(String[] args) {

        // Create a Bedrock Runtime client in the AWS Region of your choice.
        var client = BedrockRuntimeClient.builder()
                .region(Region.US_WEST_2)
                .build();

        // Set the model ID, e.g., Llama 2 Chat 13B.
        var modelId = "meta.llama2-13b-chat-v1";

        // Define the user message to send.
        var userMessage = "Describe the purpose of a 'hello world' program in one
 line.";
```

```
        // Embed the message in Llama 2's prompt format.
        var prompt = "<s>[INST] " + userMessage + " [/INST]";

        // Create a JSON payload using the model's native structure.
        var request = new JSONObject()
                .put("prompt", prompt)
                // Optional inference parameters:
                .put("max_gen_len", 512)
                .put("temperature", 0.5F)
                .put("top_p", 0.9F);

        // Encode and send the request.
        var response = client.invokeModel(req -> req
                .body(SdkBytes.fromUtf8String(request.toString()))
                .modelId(modelId));

        // Decode the native response body.
        var nativeResponse = new JSONObject(response.body().asUtf8String());

        // Extract and print the response text.
        var responseText = nativeResponse.getString("generation");
        System.out.println(responseText);
    }
}
// Learn more about the Llama 2 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-2
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 2.

```
// Send a prompt to Meta Llama 2 and print the response.

import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 2 Chat 13B.
const modelId = "meta.llama2-13b-chat-v1";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 2's prompt format.
const prompt = `<s>[INST] ${userMessage} [/INST]`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const response = await client.send(
  new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Decode the native response body.
/** @type {{ generation: string }} */
const nativeResponse = JSON.parse(new TextDecoder().decode(response.body));

// Extract and print the generated text.
```

```
const responseText = nativeResponse.generation;
console.log(responseText);

// Learn more about the Llama 2 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-2
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Meta Llama 2 Chat foundation model to generate text.

```
    public function invokeLlama2($prompt)
    {
        # The different model providers have individual request and response
 formats.
        # For the format, ranges, and default values for Meta Llama 2 Chat, refer
 to:
        # https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
meta.html

        $completion = "";

        try {
            $modelId = 'meta.llama2-13b-chat-v1';

            $body = [
                'prompt' => $prompt,
                'temperature' => 0.5,
                'max_gen_len' => 512,
            ];
```

```
            $result = $this->bedrockRuntimeClient->invokeModel([
                'contentType' => 'application/json',
                'body' => json_encode($body),
                'modelId' => $modelId,
            ]);

            $response_body = json_decode($result['body']);

            $completion = $response_body->generation;
        } catch (Exception $e) {
            echo "Error: ({$e->getCode()}) - {$e->getMessage()}\n";
        }

        return $completion;
    }
```

- For API details, see InvokeModel in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 2.

```
# Send a prompt to Meta Llama 2 and print the response.

import boto3
import json

# Create a Bedrock Runtime client in the AWS Region of your choice.
client = boto3.client("bedrock-runtime", region_name="us-west-2")

# Set the model ID, e.g., Llama 2 Chat 13B.
model_id = "meta.llama2-13b-chat-v1"
```

```
# Define the user message to send.
user_message = "Describe the purpose of a 'hello world' program in one line."

# Embed the message in Llama 2's prompt format.
prompt = f"<s>[INST] {user_message} [/INST]"

# Format the request payload using the model's native structure.
request = {
    "prompt": prompt,
    # Optional inference parameters:
    "max_gen_len": 512,
    "temperature": 0.5,
    "top_p": 0.9,
}

# Encode and send the request.
response = client.invoke_model(body=json.dumps(request), modelId=model_id)

# Decode the native response body.
model_response = json.loads(response["body"].read())

# Extract and print the generated text.
response_text = model_response["generation"]
print(response_text)

# Learn more about the Llama 2 prompt format at:
# https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-2
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Meta Llama 2 on Amazon Bedrock using Meta's native request and response payloads with a response stream

The following code examples show how to get started sending prompts to Meta Llama 2 and printing the response stream in real-time.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javav2/example_code/bedrock-runtime#readme).

Send your first prompt to Meta Llama 3.

```
// Send a prompt to Meta Llama 2 and print the response stream in real-time.
public class InvokeModelWithResponseStreamQuickstart {

    public static void main(String[] args) {

        // Create a Bedrock Runtime client in the AWS Region of your choice.
        var client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_WEST_2)
                .build();

        // Set the model ID, e.g., Llama 2 Chat 13B.
        var modelId = "meta.llama2-13b-chat-v1";

        // Define the user message to send.
        var userMessage = "Describe the purpose of a 'hello world' program in one
 line.";

        // Embed the message in Llama 2's prompt format.
        var prompt = "<s>[INST] " + userMessage + " [/INST]";

        // Create a JSON payload using the model's native structure.
        var request = new JSONObject()
                .put("prompt", prompt)
                // Optional inference parameters:
                .put("max_gen_len", 512)
                .put("temperature", 0.5F)
                .put("top_p", 0.9F);

        // Create a handler to extract and print the response text in real-time.
        var streamHandler =
 InvokeModelWithResponseStreamResponseHandler.builder()
```

```
                .subscriber(event -> event.accept(
 InvokeModelWithResponseStreamResponseHandler.Visitor.builder()
                                .onChunk(c -> {
                                    var chunk = new
 JSONObject(c.bytes().asUtf8String());
                                    if (chunk.has("generation")) {
 System.out.print(chunk.getString("generation"));
                                    }
                                }).build())
                ).build();

        // Encode and send the request. Let the stream handler process the
 response.
        client.invokeModelWithResponseStream(req -> req
                .body(SdkBytes.fromUtf8String(request.toString()))
                .modelId(modelId), streamHandler
        ).join();
    }
}
// Learn more about the Llama 2 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-2
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 3.

```
// Send a prompt to Meta Llama 2 and print the response stream in real-time.
```

```
import {
  BedrockRuntimeClient,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 2 Chat 13B.
const modelId = "meta.llama2-13b-chat-v1";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 2's prompt format.
const prompt = `<s>[INST] ${userMessage} [/INST]`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const responseStream = await client.send(
  new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Extract and print the response stream in real-time.
for await (const event of responseStream.body) {
  /** @type {{ generation: string }} */
  const chunk = JSON.parse(new TextDecoder().decode(event.chunk.bytes));
  if (chunk.generation) {
    process.stdout.write(chunk.generation);
  }
```

```
}

// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 3.

```
# Send a prompt to Meta Llama 2 and print the response stream in real-time.

import boto3
import json

# Create a Bedrock Runtime client in the AWS Region of your choice.
client = boto3.client("bedrock-runtime", region_name="us-west-2")

# Set the model ID, e.g., Llama 2 Chat 13B.
model_id = "meta.llama2-13b-chat-v1"

# Define the user message to send.
user_message = "Describe the purpose of a 'hello world' program in one line."

# Embed the message in Llama 2's prompt format.
prompt = f"<s>[INST] {user_message} [/INST]"

# Format the request payload using the model's native structure.
request = {
```

```
    "prompt": prompt,
    # Optional inference parameters:
    "max_gen_len": 512,
    "temperature": 0.5,
    "top_p": 0.9,
}

# Encode and send the request.
response_stream = client.invoke_model_with_response_stream(
    body=json.dumps(request),
    modelId=model_id,
)

# Extract and print the response text in real-time.
for event in response_stream["body"]:
    chunk = json.loads(event["chunk"]["bytes"])
    if "generation" in chunk:
        print(chunk["generation"], end="")

# Learn more about the Llama 2 prompt format at:
# https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-2
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Meta Llama 3 on Amazon Bedrock using Meta's native request and response payloads

The following code examples show how to get started sending prompts to Meta Llama 3 and printing the response.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javav2/example_code/bedrock-runtime#code-examples).

Send your first prompt to Meta Llama 3.

```
// Send a prompt to Meta Llama 3 and print the response.
public class InvokeModelQuickstart {

    public static void main(String[] args) {

        // Create a Bedrock Runtime client in the AWS Region of your choice.
        var client = BedrockRuntimeClient.builder()
                .region(Region.US_WEST_2)
                .build();

        // Set the model ID, e.g., Llama 3 8B Instruct.
        var modelId = "meta.llama3-8b-instruct-v1:0";

        // Define the user message to send.
        var userMessage = "Describe the purpose of a 'hello world' program in one
 line.";

        // Embed the message in Llama 3's prompt format.
        var prompt = MessageFormat.format("""
                <|begin_of_text|>
                <|start_header_id|>user<|end_header_id|>
                {0}
                <|eot_id|>
                <|start_header_id|>assistant<|end_header_id|>
                """, userMessage);

        // Create a JSON payload using the model's native structure.
        var request = new JSONObject()
                .put("prompt", prompt)
                // Optional inference parameters:
                .put("max_gen_len", 512)
```

```
                .put("temperature", 0.5F)
                .put("top_p", 0.9F);

        // Encode and send the request.
        var response = client.invokeModel(req -> req
                .body(SdkBytes.fromUtf8String(request.toString()))
                .modelId(modelId));

        // Decode the native response body.
        var nativeResponse = new JSONObject(response.body().asUtf8String());

        // Extract and print the response text.
        var responseText = nativeResponse.getString("generation");
        System.out.println(responseText);
    }
}
// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 3.

```
// Send a prompt to Meta Llama 3 and print the response.

import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";
```

```
// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 3 8B Instruct.
const modelId = "meta.llama3-8b-instruct-v1:0";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 3's prompt format.
const prompt = `
<|begin_of_text|>
<|start_header_id|>user<|end_header_id|>
${userMessage}
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const response = await client.send(
  new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Decode the native response body.
/** @type {{ generation: string }} */
const nativeResponse = JSON.parse(new TextDecoder().decode(response.body));

// Extract and print the generated text.
const responseText = nativeResponse.generation;
```

```
console.log(responseText);

// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 3.

```
# Send a prompt to Meta Llama 3 and print the response.

import boto3
import json

# Create a Bedrock Runtime client in the AWS Region of your choice.
client = boto3.client("bedrock-runtime", region_name="us-west-2")

# Set the model ID, e.g., Llama 3 8B Instruct.
model_id = "meta.llama3-8b-instruct-v1:0"

# Define the user message to send.
user_message = "Describe the purpose of a 'hello world' program in one line."

# Embed the message in Llama 3's prompt format.
prompt = f"""
<|begin_of_text|>
<|start_header_id|>user<|end_header_id|>
{user_message}
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
```

```
"""

# Format the request payload using the model's native structure.
request = {
    "prompt": prompt,
    # Optional inference parameters:
    "max_gen_len": 512,
    "temperature": 0.5,
    "top_p": 0.9,
}

# Encode and send the request.
response = client.invoke_model(body=json.dumps(request), modelId=model_id)

# Decode the native response body.
model_response = json.loads(response["body"].read())

# Extract and print the generated text.
response_text = model_response["generation"]
print(response_text)

# Learn more about the Llama 3 prompt format in the documentation:
# https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Meta Llama 3 on Amazon Bedrock using Meta's native request and response payloads with a response stream

The following code examples show how to get started sending prompts to Meta Llama 3 and printing the response stream in real-time.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javav2/example_code/bedrock-runtime#code-examples).

Send your first prompt to Meta Llama 3.

```
// Send a prompt to Meta Llama 3 and print the response stream in real-time.
public class InvokeModelWithResponseStreamQuickstart {

    public static void main(String[] args) {

        // Create a Bedrock Runtime client in the AWS Region of your choice.
        var client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_WEST_2)
                .build();

        // Set the model ID, e.g., Llama 3 8B Instruct.
        var modelId = "meta.llama3-8b-instruct-v1:0";

        // Define the user message to send.
        var userMessage = "Describe the purpose of a 'hello world' program in one
 line.";

        // Embed the message in Llama 3's prompt format.
        var prompt = MessageFormat.format("""
                <|begin_of_text|>
                <|start_header_id|>user<|end_header_id|>
                {0}
                <|eot_id|>
                <|start_header_id|>assistant<|end_header_id|>
                """, userMessage);

        // Create a JSON payload using the model's native structure.
        var request = new JSONObject()
                .put("prompt", prompt)
                // Optional inference parameters:
                .put("max_gen_len", 512)
```

```
                .put("temperature", 0.5F)
                .put("top_p", 0.9F);

        // Create a handler to extract and print the response text in real-time.
        var streamHandler =
 InvokeModelWithResponseStreamResponseHandler.builder()
                .subscriber(event -> event.accept(

 InvokeModelWithResponseStreamResponseHandler.Visitor.builder()
                                .onChunk(c -> {
                                    var chunk = new
 JSONObject(c.bytes().asUtf8String());
                                    if (chunk.has("generation")) {

 System.out.print(chunk.getString("generation"));
                                    }
                                }).build())
                ).build();

        // Encode and send the request. Let the stream handler process the
 response.
        client.invokeModelWithResponseStream(req -> req
                .body(SdkBytes.fromUtf8String(request.toString()))
                .modelId(modelId), streamHandler
        ).join();
    }
}
// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 3.

```
// Send a prompt to Meta Llama 3 and print the response stream in real-time.

import {
  BedrockRuntimeClient,
  InvokeModelWithResponseStreamCommand,
} from "@aws-sdk/client-bedrock-runtime";

// Create a Bedrock Runtime client in the AWS Region of your choice.
const client = new BedrockRuntimeClient({ region: "us-west-2" });

// Set the model ID, e.g., Llama 3 8B Instruct.
const modelId = "meta.llama3-8b-instruct-v1:0";

// Define the user message to send.
const userMessage =
  "Describe the purpose of a 'hello world' program in one sentence.";

// Embed the message in Llama 3's prompt format.
const prompt = `
<|begin_of_text|>
<|start_header_id|>user<|end_header_id|>
${userMessage}
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
`;

// Format the request payload using the model's native structure.
const request = {
  prompt,
  // Optional inference parameters:
  max_gen_len: 512,
```

```
  temperature: 0.5,
  top_p: 0.9,
};

// Encode and send the request.
const responseStream = await client.send(
  new InvokeModelWithResponseStreamCommand({
    contentType: "application/json",
    body: JSON.stringify(request),
    modelId,
  }),
);

// Extract and print the response stream in real-time.
for await (const event of responseStream.body) {
  /** @type {{ generation: string }} */
  const chunk = JSON.parse(new TextDecoder().decode(event.chunk.bytes));
  if (chunk.generation) {
    process.stdout.write(chunk.generation);
  }
}

// Learn more about the Llama 3 prompt format at:
// https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Send your first prompt to Meta Llama 3.

```
# Send a prompt to Meta Llama 3 and print the response stream in real-time.

import boto3
import json

# Create a Bedrock Runtime client in the AWS Region of your choice.
client = boto3.client("bedrock-runtime", region_name="us-west-2")

# Set the model ID, e.g., Llama 3 8B Instruct.
model_id = "meta.llama3-8b-instruct-v1:0"

# Define the user message to send.
user_message = "Describe the purpose of a 'hello world' program in one line."

# Embed the message in Llama 3's prompt format.
prompt = f"""
<|begin_of_text|>
<|start_header_id|>user<|end_header_id|>
{user_message}
<|eot_id|>
<|start_header_id|>assistant<|end_header_id|>
"""

# Format the request payload using the model's native structure.
request = {
    "prompt": prompt,
    # Optional inference parameters:
    "max_gen_len": 512,
    "temperature": 0.5,
    "top_p": 0.9,
}

# Encode and send the request.
response_stream = client.invoke_model_with_response_stream(
    body=json.dumps(request),
    modelId=model_id,
)

# Extract and print the response text in real-time.
for event in response_stream["body"]:
    chunk = json.loads(event["chunk"]["bytes"])
    if "generation" in chunk:
        print(chunk["generation"], end="")
```

```
# Learn more about the Llama 3 prompt format at:
# https://llama.meta.com/docs/model-cards-and-prompt-formats/meta-llama-3/
#special-tokens-used-with-meta-llama-3
```

- For API details, see InvokeModelWithResponseStream in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke the Mistral 7B model on Amazon Bedrock for text generation

The following code examples show how to invoke the Mistral 7B model on Amazon Bedrock for text generation.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Mistral 7B foundation model to generate text.

```
        /// <summary>
        /// Asynchronously invokes the Mistral 7B model to run an inference based
 on the provided input.
        /// </summary>
        /// <param name="prompt">The prompt that you want Mistral 7B to
 complete.</param>
        /// <returns>The inference response from the model</returns>
        /// <remarks>
```

```
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Mistral 7B, refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-mistral.html
        /// </remarks>
        public static async Task<List<string?>> InvokeMistral7BAsync(string
 prompt)
        {
            string mistralModelId = "mistral.mistral-7b-instruct-v0:2";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USWest2);

            string payload = new JsonObject()
            {
                { "prompt", prompt },
                { "max_tokens", 200 },
                { "temperature", 0.5 }
            }.ToJsonString();

            List<string?>? generatedText = null;
            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
 InvokeModelRequest()
                {
                    ModelId = mistralModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    var results = JsonNode.ParseAsync(response.Body).Result?
["outputs"]?.AsArray();

                    generatedText = results?.Select(x => x?
["text"]?.GetValue<string?>())?.ToList();
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
 + response.HttpStatusCode);
```

```
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return generatedText ?? [];
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Mistral 7B foundation model to generate text.

```
    /**
     * Asynchronously invokes the Mistral 7B model to run an inference based on
 the provided input.
     *
     * @param prompt The prompt for Mistral to complete.
     * @return The generated response.
     */
    public static List<String> invokeMistral7B(String prompt) {
        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_WEST_2)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();

        // Mistral instruct models provide optimal results when
        // embedding the prompt into the following template:
        String instruction = "<s>[INST] " + prompt + " [/INST]";
```

```
        String modelId = "mistral.mistral-7b-instruct-v0:2";

        String payload = new JSONObject()
                .put("prompt", instruction)
                .put("max_tokens", 200)
                .put("temperature", 0.5)
                .toString();

        CompletableFuture<InvokeModelResponse> completableFuture =
client.invokeModel(request -> request
                .accept("application/json")
                .contentType("application/json")
                .body(SdkBytes.fromUtf8String(payload))
                .modelId(modelId))
        .whenComplete((response, exception) -> {
            if (exception != null) {
                System.out.println("Model invocation failed: " + exception);
            }
        });

        try {
            InvokeModelResponse response = completableFuture.get();
            JSONObject responseBody = new
JSONObject(response.body().asUtf8String());
            JSONArray outputs = responseBody.getJSONArray("outputs");

            return IntStream.range(0, outputs.length())
                    .mapToObj(i -> outputs.getJSONObject(i).getString("text"))
                    .toList();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
        }

        return List.of();
    }
```

Invoke the Mistral 7B foundation model to generate text.

```
        /**
```

```
         * Invokes the Mistral 7B model to run an inference based on the provided
input.
         *
         * @param prompt The prompt for Mistral to complete.
         * @return The generated responses.
         */
        public static List<String> invokeMistral7B(String prompt) {
                BedrockRuntimeClient client = BedrockRuntimeClient.builder()
                        .region(Region.US_WEST_2)
                        .credentialsProvider(ProfileCredentialsProvider.create())
                        .build();

                // Mistral instruct models provide optimal results when
                // embedding the prompt into the following template:
                String instruction = "<s>[INST] " + prompt + " [/INST]";

                String modelId = "mistral.mistral-7b-instruct-v0:2";

                String payload = new JSONObject()
                        .put("prompt", instruction)
                        .put("max_tokens", 200)
                        .put("temperature", 0.5)
                        .toString();

                InvokeModelResponse response = client.invokeModel(request ->
request
                        .accept("application/json")
                        .contentType("application/json")
                        .body(SdkBytes.fromUtf8String(payload))
                        .modelId(modelId));

                JSONObject responseBody = new
JSONObject(response.body().asUtf8String());
                JSONArray outputs = responseBody.getJSONArray("outputs");

                return IntStream.range(0, outputs.length())
                        .mapToObj(i ->
outputs.getJSONObject(i).getString("text"))
                        .toList();

        }
```

- For API details, see [InvokeModel](#) in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javascriptv3/example_code/bedrock-runtime#code-examples).

Invoke the Mistral 7B foundation model to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} Output
 * @property {string} text
 *
 * @typedef {Object} ResponseBody
 * @property {Output[]} outputs
 */

/**
 * Invokes a Mistral 7B Instruct model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "mistral.mistral-7b-instruct-v0:2".
 */
export const invokeModel = async (
  prompt,
  modelId = "mistral.mistral-7b-instruct-v0:2",
) => {
  // Create a new Bedrock Runtime client instance.
```

```
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Mistral instruct models provide optimal results when embedding
  // the prompt into the following template:
  const instruction = `<s>[INST] ${prompt} [/INST]`;

  // Prepare the payload.
  const payload = {
    prompt: instruction,
    max_tokens: 500,
    temperature: 0.5,
  };

  // Invoke the model with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response.
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {ResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.outputs[0].text;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time..."';
  const modelId = FoundationModels.MISTRAL_7B.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Mistral 7B foundation model to generate text.

```
    def invoke_mistral_7b(self, prompt):
        """
        Invokes the Mistral 7B model to run an inference using the input
        provided in the request body.

        :param prompt: The prompt that you want Mistral to complete.
        :return: List of inference responses from the model.
        """

        try:
            # Mistral instruct models provide optimal results when
            # embedding the prompt into the following template:
            instruction = f"<s>[INST] {prompt} [/INST]"

            model_id = "mistral.mistral-7b-instruct-v0:2"

            body = {
                "prompt": instruction,
                "max_tokens": 200,
                "temperature": 0.5,
            }

            response = self.bedrock_runtime_client.invoke_model(
                modelId=model_id, body=json.dumps(body)
            )
```

```
            response_body = json.loads(response["body"].read())
            outputs = response_body.get("outputs")

            completions = [output["text"] for output in outputs]

            return completions

        except ClientError:
            logger.error("Couldn't invoke Mistral 7B")
            raise
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke the Mixtral 8x7B model on Amazon Bedrock for text generation

The following code examples show how to invoke the Mixtral 8x7B model on Amazon Bedrock for text generation.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Mixtral 8x7B foundation model to generate text.

```
        /// <summary>
        /// Asynchronously invokes the Mixtral 8x7B model to run an inference
 based on the provided input.
        /// </summary>
```

```
        /// <param name="prompt">The prompt that you want Mixtral 8x7B to
 complete.</param>
        /// <returns>The inference response from the model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Mixtral 8x7B, refer
 to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-mistral.html
        /// </remarks>
        public static async Task<List<string?>> InvokeMixtral8x7BAsync(string
 prompt)
        {
            string mixtralModelId = "mistral.mixtral-8x7b-instruct-v0:1";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USWest2);

            string payload = new JsonObject()
            {
                { "prompt", prompt },
                { "max_tokens", 200 },
                { "temperature", 0.5 }
            }.ToJsonString();

            List<string?>? generatedText = null;
            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
 InvokeModelRequest()
                {
                    ModelId = mixtralModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    var results = JsonNode.ParseAsync(response.Body).Result?
["outputs"]?.AsArray();

                    generatedText = results?.Select(x => x?
["text"]?.GetValue<string?>())?.ToList();
```

```
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
+ response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return generatedText ?? [];
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Mistral 8x7B foundation model to generate text.

```
    /**
     * Asynchronously invokes the Mixtral 8x7B model to run an inference based on
the provided input.
     *
     * @param prompt The prompt for Mixtral to complete.
     * @return The generated response.
     */
    public static List<String> invokeMixtral8x7B(String prompt) {
        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_WEST_2)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();
```

```
        // Mistral instruct models provide optimal results when
        // embedding the prompt into the following template:
        String instruction = "<s>[INST] " + prompt + " [/INST]";

        String modelId = "mistral.mixtral-8x7b-instruct-v0:1";

        String payload = new JSONObject()
                .put("prompt", instruction)
                .put("max_tokens", 200)
                .put("temperature", 0.5)
                .toString();

        CompletableFuture<InvokeModelResponse> completableFuture =
client.invokeModel(request -> request
                        .accept("application/json")
                        .contentType("application/json")
                        .body(SdkBytes.fromUtf8String(payload))
                        .modelId(modelId))
                .whenComplete((response, exception) -> {
                    if (exception != null) {
                        System.out.println("Model invocation failed: " +
exception);
                    }
                });

        try {
            InvokeModelResponse response = completableFuture.get();
            JSONObject responseBody = new
JSONObject(response.body().asUtf8String());
            JSONArray outputs = responseBody.getJSONArray("outputs");

            return IntStream.range(0, outputs.length())
                    .mapToObj(i -> outputs.getJSONObject(i).getString("text"))
                    .toList();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
        }

        return List.of();
    }
```

Invoke the Mixtral 8x7B foundation model to generate text.

```
        public static List<String> invokeMixtral8x7B(String prompt) {
                BedrockRuntimeClient client = BedrockRuntimeClient.builder()
                        .region(Region.US_WEST_2)
                        .credentialsProvider(ProfileCredentialsProvider.create())
                        .build();

                // Mistral instruct models provide optimal results when
                // embedding the prompt into the following template:
                String instruction = "<s>[INST] " + prompt + " [/INST]";

                String modelId = "mistral.mixtral-8x7b-instruct-v0:1";

                String payload = new JSONObject()
                        .put("prompt", instruction)
                        .put("max_tokens", 200)
                        .put("temperature", 0.5)
                        .toString();

                InvokeModelResponse response = client.invokeModel(request ->
request
                        .accept("application/json")
                        .contentType("application/json")
                        .body(SdkBytes.fromUtf8String(payload))
                        .modelId(modelId));

                JSONObject responseBody = new
JSONObject(response.body().asUtf8String());
                JSONArray outputs = responseBody.getJSONArray("outputs");

                return IntStream.range(0, outputs.length())
                        .mapToObj(i ->
outputs.getJSONObject(i).getString("text"))
                        .toList();
        }
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/javascriptv3/example_code/bedrock-runtime#code-examples).

Invoke the Mixtral 8x7B foundation model to generate text.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import { FoundationModels } from "../../config/foundation_models.js";
import {
  BedrockRuntimeClient,
  InvokeModelCommand,
} from "@aws-sdk/client-bedrock-runtime";

/**
 * @typedef {Object} Output
 * @property {string} text
 *
 * @typedef {Object} ResponseBody
 * @property {Output[]} outputs
 */

/**
 * Invokes a Mistral 8x7B Instruct model.
 *
 * @param {string} prompt - The input text prompt for the model to complete.
 * @param {string} [modelId] - The ID of the model to use. Defaults to
 "mistral.mixtral-8x7b-instruct-v0:1".
 */
export const invokeModel = async (
  prompt,
  modelId = "mistral.mixtral-8x7b-instruct-v0:1",
) => {
  // Create a new Bedrock Runtime client instance.
```

```
  const client = new BedrockRuntimeClient({ region: "us-east-1" });

  // Mistral instruct models provide optimal results when embedding
  // the prompt into the following template:
  const instruction = `<s>[INST] ${prompt} [/INST]`;

  // Prepare the payload.
  const payload = {
    prompt: instruction,
    max_tokens: 500,
    temperature: 0.5,
  };

  // Invoke the model with the payload and wait for the response.
  const command = new InvokeModelCommand({
    contentType: "application/json",
    body: JSON.stringify(payload),
    modelId,
  });
  const apiResponse = await client.send(command);

  // Decode and return the response.
  const decodedResponseBody = new TextDecoder().decode(apiResponse.body);
  /** @type {ResponseBody} */
  const responseBody = JSON.parse(decodedResponseBody);
  return responseBody.outputs[0].text;
};

// Invoke the function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const prompt =
    'Complete the following in one sentence: "Once upon a time..."';
  const modelId = FoundationModels.MIXTRAL_8X7B.modelId;
  console.log(`Prompt: ${prompt}`);
  console.log(`Model ID: ${modelId}`);

  try {
    console.log("-".repeat(53));
    const response = await invokeModel(prompt, modelId);
    console.log(response);
  } catch (err) {
    console.log(err);
  }
}
```

- For API details, see InvokeModel in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Mixtral 8x7B foundation model to generate text.

```
    def invoke_mixtral_8x7b(self, prompt):
        """
        Invokes the Mixtral 8c7B model to run an inference using the input
        provided in the request body.

        :param prompt: The prompt that you want Mixtral to complete.
        :return: List of inference responses from the model.
        """

        try:
            # Mistral instruct models provide optimal results when
            # embedding the prompt into the following template:
            instruction = f"<s>[INST] {prompt} [/INST]"

            model_id = "mistral.mixtral-8x7b-instruct-v0:1"

            body = {
                "prompt": instruction,
                "max_tokens": 200,
                "temperature": 0.5,
            }

            response = self.bedrock_runtime_client.invoke_model(
                modelId=model_id, body=json.dumps(body)
            )
```

```
            response_body = json.loads(response["body"].read())
            outputs = response_body.get("outputs")

            completions = [output["text"] for output in outputs]

            return completions

        except ClientError:
            logger.error("Couldn't invoke Mixtral 8x7B")
            raise
```

- For API details, see [InvokeModel](#) in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke Stability.ai Stable Diffusion XL on Amazon Bedrock to generate an image

The following code examples show how to invoke Stability.ai Stable Diffusion XL on Amazon Bedrock to generate an image.

.NET

AWS SDK for .NET

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke Stability.ai Stable Diffusion XL foundation model to generate images.

```
        /// <summary>
        /// Asynchronously invokes the Stability.ai Stable Diffusion XLmodel to
 run an inference based on the provided input.
        /// </summary>
```

```
        /// <param name="prompt">The prompt that describes the image Stability.ai
 Stable Diffusion XL has to generate.</param>
        /// <returns>A base-64 encoded image generated by model</returns>
        /// <remarks>
        /// The different model providers have individual request and response
 formats.
        /// For the format, ranges, and default values for Stability.ai Stable
 Diffusion XL, refer to:
        ///     https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-stability-diffusion.html
        /// </remarks>
        public static async Task<string?> InvokeStableDiffusionXLG1Async(string
 prompt, int seed, string? stylePreset = null)
        {
            string stableDiffusionXLModelId = "stability.stable-diffusion-xl";

            AmazonBedrockRuntimeClient client = new(RegionEndpoint.USEast1);

            var jsonPayload = new JsonObject()
            {
                { "text_prompts", new JsonArray() {
                    new JsonObject()
                        {
                            { "text", prompt }
                        }
                    }
                },
                { "seed", seed }
            };

            if (!string.IsNullOrEmpty(stylePreset))
            {
                jsonPayload.Add("style_preset", stylePreset);
            }

            string payload = jsonPayload.ToString();

            try
            {
                InvokeModelResponse response = await client.InvokeModelAsync(new
 InvokeModelRequest()
                {
                    ModelId = stableDiffusionXLModelId,
                    Body = AWSSDKUtils.GenerateMemoryStreamFromString(payload),
```

```
                    ContentType = "application/json",
                    Accept = "application/json"
                });

                if (response.HttpStatusCode == System.Net.HttpStatusCode.OK)
                {
                    var results = JsonNode.ParseAsync(response.Body).Result?
["artifacts"]?.AsArray();

                    return results?[0]?["base64"]?.GetValue<string>();
                }
                else
                {
                    Console.WriteLine("InvokeModelAsync failed with status code "
 + response.HttpStatusCode);
                }
            }
            catch (AmazonBedrockRuntimeException e)
            {
                Console.WriteLine(e.Message);
            }
            return null;
        }
```

- For API details, see InvokeModel in *AWS SDK for .NET API Reference*.

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Asynchronously invoke the Stability.ai Stable Diffusion XL foundation model to generate images.

```
    /**
```

```
     * Asynchronously invokes the Stability.ai Stable Diffusion XL model to
 create
     * an image based on the provided input.
     *
     * @param prompt      The prompt that guides the Stable Diffusion model.
     * @param seed        The random noise seed for image generation (use 0 or
 omit
     *                    for a random seed).
     * @param stylePreset The style preset to guide the image model towards a
     *                    specific style.
     * @return A Base64-encoded string representing the generated image.
     */
    public static String invokeStableDiffusion(String prompt, long seed, String
 stylePreset) {
        /*
         * The different model providers have individual request and response
 formats.
         * For the format, ranges, and available style_presets of Stable
 Diffusion
         * models refer to:
         * https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
stability-diffusion.html
         */

        String stableDiffusionModelId = "stability.stable-diffusion-xl";

        BedrockRuntimeAsyncClient client = BedrockRuntimeAsyncClient.builder()
                .region(Region.US_EAST_1)
                .credentialsProvider(ProfileCredentialsProvider.create())
                .build();

        JSONArray wrappedPrompt = new JSONArray().put(new
 JSONObject().put("text", prompt));
        JSONObject payload = new JSONObject()
                .put("text_prompts", wrappedPrompt)
                .put("seed", seed);

        if (stylePreset != null && !stylePreset.isEmpty()) {
            payload.put("style_preset", stylePreset);
        }

        InvokeModelRequest request = InvokeModelRequest.builder()
                .body(SdkBytes.fromUtf8String(payload.toString()))
                .modelId(stableDiffusionModelId)
```

```
                .contentType("application/json")
                .accept("application/json")
                .build();

        CompletableFuture<InvokeModelResponse> completableFuture =
client.invokeModel(request)
                .whenComplete((response, exception) -> {
                    if (exception != null) {
                        System.out.println("Model invocation failed: " +
exception);
                    }
                });

        String base64ImageData = "";
        try {
            InvokeModelResponse response = completableFuture.get();
            JSONObject responseBody = new
JSONObject(response.body().asUtf8String());
            base64ImageData = responseBody
                    .getJSONArray("artifacts")
                    .getJSONObject(0)
                    .getString("base64");

        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
            System.err.println(e.getMessage());
        } catch (ExecutionException e) {
            System.err.println(e.getMessage());
        }

        return base64ImageData;
    }
```

Invoke the Stability.ai Stable Diffusion XL foundation model to generate images.

```
        /**
         * Invokes the Stability.ai Stable Diffusion XL model to create an image
based
         * on the provided input.
         *
         * @param prompt      The prompt that guides the Stable Diffusion model.
```

```
         * @param seed        The random noise seed for image generation (use 0
 or omit
         *                    for a random seed).
         * @param stylePreset The style preset to guide the image model towards a
         *                    specific style.
         * @return A Base64-encoded string representing the generated image.
         */
        public static String invokeStableDiffusion(String prompt, long seed,
 String stylePreset) {
                /*
                 * The different model providers have individual request and
 response formats.
                 * For the format, ranges, and available style_presets of Stable
 Diffusion
                 * models refer to:
                 * https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-stability-diffusion.html
                 */

                String stableDiffusionModelId = "stability.stable-diffusion-xl";

                BedrockRuntimeClient client = BedrockRuntimeClient.builder()
                                .region(Region.US_EAST_1)

 .credentialsProvider(ProfileCredentialsProvider.create())
                                .build();

                JSONArray wrappedPrompt = new JSONArray().put(new
 JSONObject().put("text", prompt));

                JSONObject payload = new JSONObject()
                                .put("text_prompts", wrappedPrompt)
                                .put("seed", seed);

                if (!(stylePreset == null || stylePreset.isEmpty())) {
                        payload.put("style_preset", stylePreset);
                }

                InvokeModelRequest request = InvokeModelRequest.builder()

 .body(SdkBytes.fromUtf8String(payload.toString()))
                                .modelId(stableDiffusionModelId)
                                .contentType("application/json")
                                .accept("application/json")
```

```
                                .build();

                InvokeModelResponse response = client.invokeModel(request);

                JSONObject responseBody = new
 JSONObject(response.body().asUtf8String());

                String base64ImageData = responseBody
                                .getJSONArray("artifacts")
                                .getJSONObject(0)
                                .getString("base64");

                return base64ImageData;
        }
```

- For API details, see InvokeModel in *AWS SDK for Java 2.x API Reference*.

PHP

SDK for PHP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Stability.ai Stable Diffusion XL foundation model to generate images.

```
    public function invokeStableDiffusion(string $prompt, int $seed, string
 $style_preset)
    {
        # The different model providers have individual request and response
 formats.
        # For the format, ranges, and available style_presets of Stable Diffusion
 models refer to:
        # https://docs.aws.amazon.com/bedrock/latest/userguide/model-parameters-
stability-diffusion.html

        $base64_image_data = "";
```

```
        try {
            $modelId = 'stability.stable-diffusion-xl';

            $body = [
                'text_prompts' => [
                    ['text' => $prompt]
                ],
                'seed' => $seed,
                'cfg_scale' => 10,
                'steps' => 30
            ];

            if ($style_preset) {
                $body['style_preset'] = $style_preset;
            }

            $result = $this->bedrockRuntimeClient->invokeModel([
                'contentType' => 'application/json',
                'body' => json_encode($body),
                'modelId' => $modelId,
            ]);

            $response_body = json_decode($result['body']);

            $base64_image_data = $response_body->artifacts[0]->base64;
        } catch (Exception $e) {
            echo "Error: ({$e->getCode()}) - {$e->getMessage()}\n";
        }

        return $base64_image_data;
    }
```

- For API details, see [InvokeModel](#) in *AWS SDK for PHP API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Stability.ai Stable Diffusion XL foundation model to generate images.

```
    def invoke_stable_diffusion(self, prompt, seed, style_preset=None):
        """
        Invokes the Stability.ai Stable Diffusion XL model to create an image
 using
        the input provided in the request body.

        :param prompt: The prompt that you want Stable Diffusion  to use for
 image generation.
        :param seed: Random noise seed (omit this option or use 0 for a random
 seed)
        :param style_preset: Pass in a style preset to guide the image model
 towards
                             a particular style.
        :return: Base64-encoded inference response from the model.
        """

        try:
            # The different model providers have individual request and response
 formats.
            # For the format, ranges, and available style_presets of Stable
 Diffusion models refer to:
            # https://docs.aws.amazon.com/bedrock/latest/userguide/model-
parameters-stability-diffusion.html

            body = {
                "text_prompts": [{"text": prompt}],
                "seed": seed,
                "cfg_scale": 10,
                "steps": 30,
            }
```

```
            if style_preset:
                body["style_preset"] = style_preset

            response = self.bedrock_runtime_client.invoke_model(
                modelId="stability.stable-diffusion-xl", body=json.dumps(body)
            )

            response_body = json.loads(response["body"].read())
            base64_image_data = response_body["artifacts"][0]["base64"]

            return base64_image_data

        except ClientError:
            logger.error("Couldn't invoke Stable Diffusion XL")
            raise
```

- For API details, see InvokeModel in *AWS SDK for Python (Boto3) API Reference*.

SAP ABAP

SDK for SAP ABAP

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke the Stability.ai Stable Diffusion XL foundation model to generate images.

```
    "Stable Diffusion Input Parameters should be in a format like this:
*   {
*     "text_prompts": [
*       {"text":"Draw a dolphin with a mustache"},
*       {"text":"Make it photorealistic"}
*     ],
*     "cfg_scale":10,
*     "seed":0,
*     "steps":50
*   }
```

```
    TYPES: BEGIN OF prompt_ts,
             text TYPE /aws1/rt_shape_string,
           END OF prompt_ts.

    DATA: BEGIN OF ls_input,
            text_prompts TYPE STANDARD TABLE OF prompt_ts,
            cfg_scale    TYPE /aws1/rt_shape_integer,
            seed         TYPE /aws1/rt_shape_integer,
            steps        TYPE /aws1/rt_shape_integer,
          END OF ls_input.

    APPEND VALUE prompt_ts( text = iv_prompt ) TO ls_input-text_prompts.
    ls_input-cfg_scale = 10.
    ls_input-seed = 0. "or better, choose a random integer.
    ls_input-steps = 50.

    DATA(lv_json) = /ui2/cl_json=>serialize(
      data = ls_input
                pretty_name   = /ui2/cl_json=>pretty_mode-low_case ).

    TRY.
        DATA(lo_response) = lo_bdr->invokemodel(
          iv_body = /aws1/cl_rt_util=>string_to_xstring( lv_json )
          iv_modelid = 'stability.stable-diffusion-xl-v0'
          iv_accept = 'application/json'
          iv_contenttype = 'application/json' ).

        "Stable Diffusion Result Format:
*       {
*         "result": "success",
*         "artifacts": [
*           {
*             "seed": 0,
*             "base64": "iVBORw0KGgoAAAANSUhEUgAAAgAAA....
*             "finishReason": "SUCCESS"
*           }
*         ]
*       }
        TYPES: BEGIN OF artifact_ts,
                 seed         TYPE /aws1/rt_shape_integer,
                 base64       TYPE /aws1/rt_shape_string,
                 finishreason TYPE /aws1/rt_shape_string,
               END OF artifact_ts.
```

```
        DATA: BEGIN OF ls_response,
                result    TYPE /aws1/rt_shape_string,
                artifacts TYPE STANDARD TABLE OF artifact_ts,
              END OF ls_response.

        /ui2/cl_json=>deserialize(
          EXPORTING jsonx = lo_response->get_body( )
                    pretty_name = /ui2/cl_json=>pretty_mode-camel_case
          CHANGING  data  = ls_response ).
        IF ls_response-artifacts IS NOT INITIAL.
          DATA(lv_image) =
 cl_http_utility=>if_http_utility~decode_x_base64( ls_response-artifacts[ 1 ]-
base64 ).
        ENDIF.
      CATCH /aws1/cx_bdraccessdeniedex INTO DATA(lo_ex).
        WRITE / lo_ex->get_text( ).
        WRITE / |Don't forget to enable model access at https://
console.aws.amazon.com/bedrock/home?#/modelaccess|.

    ENDTRY.
```

Invoke the Stability.ai Stable Diffusion XL foundation model to generate images using L2 high level client.

```
    TRY.
        DATA(lo_bdr_l2_sd) = /aws1/
cl_bdr_l2_factory=>create_stable_diffusion_10( lo_bdr ).
      " iv_prompt contains a prompt like 'Show me a picture of a unicorn reading
 an enterprise financial report'.
        DATA(lv_image) = lo_bdr_l2_sd->text_to_image( iv_prompt ).
      CATCH /aws1/cx_bdraccessdeniedex INTO DATA(lo_ex).
        WRITE / lo_ex->get_text( ).
        WRITE / |Don't forget to enable model access at https://
console.aws.amazon.com/bedrock/home?#/modelaccess|.

    ENDTRY.
```

- For API details, see InvokeModel in *AWS SDK for SAP ABAP API reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Amazon Bedrock Runtime using AWS SDKs

The following code examples show you how to implement common scenarios in Amazon Bedrock Runtime with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Amazon Bedrock Runtime. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- Create a sample application that offers playgrounds to interact with Amazon Bedrock foundation models using an AWS SDK
- Invoke various foundation models on Amazon Bedrock
- Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Create a sample application that offers playgrounds to interact with Amazon Bedrock foundation models using an AWS SDK

The following code examples show how to create playgrounds to interact with Amazon Bedrock foundation models through different modalities.

.NET

AWS SDK for .NET

.NET Foundation Model (FM) Playground is a .NET MAUI Blazor sample application that showcases how to use Amazon Bedrock from C# code. This example shows how .NET and C# developers can use Amazon Bedrock to build generative AI-enabled applications. You can test and interact with Amazon Bedrock foundation models by using the following four playgrounds:

- A text playground.
- A chat playground.
- A voice chat playground.
- An image playground.

The example also lists and displays the foundation models you have access to and their characteristics. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Bedrock Runtime

Java

SDK for Java 2.x

The Java Foundation Model (FM) Playground is a Spring Boot sample application that showcases how to use Amazon Bedrock with Java. This example shows how Java developers can use Amazon Bedrock to build generative AI-enabled applications. You can test and interact with Amazon Bedrock foundation models by using the following three playgrounds:

- A text playground.
- A chat playground.
- An image playground.

The example also lists and displays the foundation models you have access to, along with their characteristics. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Bedrock Runtime

Python

SDK for Python (Boto3)

The Python Foundation Model (FM) Playground is a Python/FastAPI sample application that showcases how to use Amazon Bedrock with Python. This example shows how Python developers can use Amazon Bedrock to build generative AI-enabled applications. You can test and interact with Amazon Bedrock foundation models by using the following three playgrounds:

- A text playground.
- A chat playground.
- An image playground.

The example also lists and displays the foundation models you have access to, along with their characteristics. For source code and deployment instructions, see the project in GitHub.

Services used in this example

- Amazon Bedrock Runtime

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Invoke various foundation models on Amazon Bedrock

The following code examples show how to prepare and send a prompt to a variety of large-language models (LLMs) on Amazon Bedrock

Go

SDK for Go V2

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke multiple foundation models on Amazon Bedrock.

```
// InvokeModelsScenario demonstrates how to use the Amazon Bedrock Runtime client
// to invoke various foundation models for text and image generation
//
// 1. Generate text with Anthropic Claude 2
// 2. Generate text with AI21 Labs Jurassic-2
// 3. Generate text with Meta Llama 2 Chat
// 4. Generate text and asynchronously process the response stream with Anthropic
 Claude 2
// 5. Generate and image with the Amazon Titan image generation model
// 6. Generate text with Amazon Titan Text G1 Express model
type InvokeModelsScenario struct {
 sdkConfig             aws.Config
 invokeModelWrapper    actions.InvokeModelWrapper
```

```
 responseStreamWrapper actions.InvokeModelWithResponseStreamWrapper
 questioner            demotools.IQuestioner
}

// NewInvokeModelsScenario constructs an InvokeModelsScenario instance from a
 configuration.
// It uses the specified config to get a Bedrock Runtime client and create
 wrappers for the
// actions used in the scenario.
func NewInvokeModelsScenario(sdkConfig aws.Config, questioner
 demotools.IQuestioner) InvokeModelsScenario {
 client := bedrockruntime.NewFromConfig(sdkConfig)
 return InvokeModelsScenario{
  sdkConfig:             sdkConfig,
  invokeModelWrapper:    actions.InvokeModelWrapper{BedrockRuntimeClient:
 client},
  responseStreamWrapper:
 actions.InvokeModelWithResponseStreamWrapper{BedrockRuntimeClient: client},
  questioner:            questioner,
 }
}

// Runs the interactive scenario.
func (scenario InvokeModelsScenario) Run() {
 defer func() {
  if r := recover(); r != nil {
   log.Printf("Something went wrong with the demo: %v\n", r)
  }
 }()

 log.Println(strings.Repeat("=", 77))
 log.Println("Welcome to the Amazon Bedrock Runtime model invocation demo.")
 log.Println(strings.Repeat("=", 77))

 log.Printf("First, let's invoke a few large-language models using the
 synchronous client:\n\n")

 text2textPrompt := "In one paragraph, who are you?"

 log.Println(strings.Repeat("-", 77))
 log.Printf("Invoking Claude with prompt: %v\n", text2textPrompt)
 scenario.InvokeClaude(text2textPrompt)

 log.Println(strings.Repeat("-", 77))
```

```
 log.Printf("Invoking Jurassic-2 with prompt: %v\n", text2textPrompt)
 scenario.InvokeJurassic2(text2textPrompt)

 log.Println(strings.Repeat("-", 77))
 log.Printf("Invoking Llama2 with prompt: %v\n", text2textPrompt)
 scenario.InvokeLlama2(text2textPrompt)

 log.Println(strings.Repeat("=", 77))
 log.Printf("Now, let's invoke Claude with the asynchronous client and process
 the response stream:\n\n")

 log.Println(strings.Repeat("-", 77))
 log.Printf("Invoking Claude with prompt: %v\n", text2textPrompt)
 scenario.InvokeWithResponseStream(text2textPrompt)

 log.Println(strings.Repeat("=", 77))
 log.Printf("Now, let's create an image with the Amazon Titan image generation
 model:\n\n")

 text2ImagePrompt := "stylized picture of a cute old steampunk robot"
 seed := rand.Int63n(2147483648)

 log.Println(strings.Repeat("-", 77))
 log.Printf("Invoking Amazon Titan with prompt: %v\n", text2ImagePrompt)
 scenario.InvokeTitanImage(text2ImagePrompt, seed)

 log.Println(strings.Repeat("-", 77))
 log.Printf("Invoking Titan Text Express with prompt: %v\n", text2textPrompt)
 scenario.InvokeTitanText(text2textPrompt)

 log.Println(strings.Repeat("=", 77))
 log.Println("Thanks for watching!")
 log.Println(strings.Repeat("=", 77))
}

func (scenario InvokeModelsScenario) InvokeClaude(prompt string) {
 completion, err := scenario.invokeModelWrapper.InvokeClaude(prompt)
 if err != nil {
  panic(err)
 }
 log.Printf("\nClaude     : %v\n", strings.TrimSpace(completion))
}

func (scenario InvokeModelsScenario) InvokeJurassic2(prompt string) {
```

```
 completion, err := scenario.invokeModelWrapper.InvokeJurassic2(prompt)
 if err != nil {
  panic(err)
 }
 log.Printf("\nJurassic-2 : %v\n", strings.TrimSpace(completion))
}

func (scenario InvokeModelsScenario) InvokeLlama2(prompt string) {
 completion, err := scenario.invokeModelWrapper.InvokeLlama2(prompt)
 if err != nil {
  panic(err)
 }
 log.Printf("\nLlama 2    : %v\n\n", strings.TrimSpace(completion))
}

func (scenario InvokeModelsScenario) InvokeWithResponseStream(prompt string) {
 log.Println("\nClaude with response stream:")
 _, err := scenario.responseStreamWrapper.InvokeModelWithResponseStream(prompt)
 if err != nil {
  panic(err)
 }
 log.Println()
}

func (scenario InvokeModelsScenario) InvokeTitanImage(prompt string, seed int64)
 {
 base64ImageData, err := scenario.invokeModelWrapper.InvokeTitanImage(prompt,
 seed)
 if err != nil {
  panic(err)
 }
 imagePath := saveImage(base64ImageData, "amazon.titan-image-generator-v1")
 fmt.Printf("The generated image has been saved to %s\n", imagePath)
}

func (scenario InvokeModelsScenario) InvokeTitanText(prompt string) {
 completion, err := scenario.invokeModelWrapper.InvokeTitanText(prompt)
 if err != nil {
  panic(err)
 }
 log.Printf("\nTitan Text Express    : %v\n\n", strings.TrimSpace(completion))
}
```

- For API details, see the following topics in *AWS SDK for Go API Reference*.
  - InvokeModel
  - InvokeModelWithResponseStream

Java

SDK for Java 2.x

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke multiple foundation models on Amazon Bedrock.

```
package com.example.bedrockruntime;

import
 software.amazon.awssdk.services.bedrockruntime.model.BedrockRuntimeException;

import java.io.FileOutputStream;
import java.net.URI;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.Base64;
import java.util.Random;

import static com.example.bedrockruntime.InvokeModel.*;

/**
 * Demonstrates the invocation of the following models:
 * Anthropic Claude 2, AI21 Labs Jurassic-2, Meta Llama 2 Chat, and Stability.ai
 * Stable Diffusion XL.
 */
public class BedrockRuntimeUsageDemo {

    private static final Random random = new Random();
```

```
    private static final String CLAUDE = "anthropic.claude-v2";
    private static final String JURASSIC2 = "ai21.j2-mid-v1";
    private static final String MISTRAL7B = "mistral.mistral-7b-instruct-v0:2";
    private static final String MIXTRAL8X7B = "mistral.mixtral-8x7b-instruct-
v0:1";
    private static final String STABLE_DIFFUSION = "stability.stable-diffusion-
xl";
    private static final String TITAN_IMAGE = "amazon.titan-image-generator-v1";

    public static void main(String[] args) {
        BedrockRuntimeUsageDemo.textToText();
        BedrockRuntimeUsageDemo.textToTextWithResponseStream();
        BedrockRuntimeUsageDemo.textToImage();
    }

    private static void textToText() {

        String prompt = "In one sentence, what is a large-language model?";
        BedrockRuntimeUsageDemo.invoke(CLAUDE, prompt);
        BedrockRuntimeUsageDemo.invoke(JURASSIC2, prompt);
        BedrockRuntimeUsageDemo.invoke(MISTRAL7B, prompt);
        BedrockRuntimeUsageDemo.invoke(MIXTRAL8X7B, prompt);
    }

    private static void invoke(String modelId, String prompt) {
        invoke(modelId, prompt, null);
    }

    private static void invoke(String modelId, String prompt, String stylePreset)
 {
        System.out.println("\n" + new String(new char[88]).replace("\0", "-"));
        System.out.println("Invoking: " + modelId);
        System.out.println("Prompt: " + prompt);

        try {
            switch (modelId) {
                case CLAUDE:
                    printResponse(invokeClaude(prompt));
                    break;
                case JURASSIC2:
                    printResponse(invokeJurassic2(prompt));
                    break;
                case MISTRAL7B:
```

```
                    for (String response : invokeMistral7B(prompt)) {
                        printResponse(response);
                    }
                    break;
                case MIXTRAL8X7B:
                    for (String response : invokeMixtral8x7B(prompt)) {
                        printResponse(response);
                    }
                    break;
                case STABLE_DIFFUSION:
                    createImage(STABLE_DIFFUSION, prompt, random.nextLong() &
0xFFFFFFFFL, stylePreset);
                    break;
                case TITAN_IMAGE:
                    createImage(TITAN_IMAGE, prompt, random.nextLong() &
0xFFFFFFFL);
                    break;
                default:
                    throw new IllegalStateException("Unexpected value: " +
modelId);
            }
        } catch (BedrockRuntimeException e) {
            System.out.println("Couldn't invoke model " + modelId + ": " +
e.getMessage());
            throw e;
        }
    }

    private static void createImage(String modelId, String prompt, long seed) {
        createImage(modelId, prompt, seed, null);
    }

    private static void createImage(String modelId, String prompt, long seed,
String stylePreset) {
        String base64ImageData = (modelId.equals(STABLE_DIFFUSION))
                ? invokeStableDiffusion(prompt, seed, stylePreset)
                : invokeTitanImage(prompt, seed);
        String imagePath = saveImage(modelId, base64ImageData);
        System.out.printf("Success: The generated image has been saved to %s%n",
imagePath);
    }

    private static void textToTextWithResponseStream() {
        String prompt = "What is a large-language model?";
```

```
        BedrockRuntimeUsageDemo.invokeWithResponseStream(CLAUDE, prompt);
    }

    private static void invokeWithResponseStream(String modelId, String prompt) {
        System.out.println(new String(new char[88]).replace("\0", "-"));
        System.out.printf("Invoking %s with response stream%n", modelId);
        System.out.println("Prompt: " + prompt);

        try {
            Claude2.invokeMessagesApiWithResponseStream(prompt);
        } catch (BedrockRuntimeException e) {
            System.out.println("Couldn't invoke model " + modelId + ": " +
 e.getMessage());
            throw e;
        }
    }

    private static void textToImage() {
        String imagePrompt = "stylized picture of a cute old steampunk robot";
        String stylePreset = "photographic";
        BedrockRuntimeUsageDemo.invoke(STABLE_DIFFUSION, imagePrompt,
 stylePreset);
        BedrockRuntimeUsageDemo.invoke(TITAN_IMAGE, imagePrompt);
    }

    private static void printResponse(String response) {
        System.out.printf("Generated text: %s%n", response);
    }

    private static String saveImage(String modelId, String base64ImageData) {
        try {
            String directory = "output";
            URI uri =
 InvokeModel.class.getProtectionDomain().getCodeSource().getLocation().toURI();
            Path outputPath =
 Paths.get(uri).getParent().getParent().resolve(directory);

            if (!Files.exists(outputPath)) {
                Files.createDirectories(outputPath);
            }

            int i = 1;
            String fileName;
            do {
```

```
                fileName = String.format("%s_%d.png", modelId, i);
                i++;
            } while (Files.exists(outputPath.resolve(fileName)));

            byte[] imageBytes = Base64.getDecoder().decode(base64ImageData);

            Path filePath = outputPath.resolve(fileName);
            try (FileOutputStream fileOutputStream = new
 FileOutputStream(filePath.toFile())) {
                fileOutputStream.write(imageBytes);
            }

            return filePath.toString();
        } catch (Exception e) {
            System.out.println(e.getMessage());
            System.exit(1);
        }
        return null;
    }
}
```

- For API details, see the following topics in *AWS SDK for Java 2.x API Reference*.
  - InvokeModel
  - InvokeModelWithResponseStream

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";
import {
```

```
  Scenario,
  ScenarioAction,
  ScenarioInput,
  ScenarioOutput,
} from "@aws-doc-sdk-examples/lib/scenario/index.js";
import { FoundationModels } from "../config/foundation_models.js";

/**
 * @typedef {Object} ModelConfig
 * @property {Function} module
 * @property {Function} invoker
 * @property {string} modelId
 * @property {string} modelName
 */

const greeting = new ScenarioOutput(
  "greeting",
  "Welcome to the Amazon Bedrock Runtime client demo!",
  { header: true },
);

const selectModel = new ScenarioInput("model", "First, select a model:", {
  type: "select",
  choices: Object.values(FoundationModels).map((model) => ({
    name: model.modelName,
    value: model,
  })),
});

const enterPrompt = new ScenarioInput("prompt", "Now, enter your prompt:", {
  type: "input",
});

const printDetails = new ScenarioOutput(
  "print details",
  /**
   * @param {{ model: ModelConfig, prompt: string }} c
   */
  (c) => console.log(`Invoking ${c.model.modelName} with '${c.prompt}'...`),
  { slow: false },
);

const invokeModel = new ScenarioAction(
  "invoke model",
```

```
  /**
   * @param {{ model: ModelConfig, prompt: string, response: string }} c
   */
  async (c) => {
    const modelModule = await c.model.module();
    const invoker = c.model.invoker(modelModule);
    c.response = await invoker(c.prompt, c.model.modelId);
  },
);

const printResponse = new ScenarioOutput(
  "print response",
  /**
   * @param {{ response: string }} c
   */
  (c) => c.response,
  { slow: false },
);

const scenario = new Scenario("Amazon Bedrock Runtime Demo", [
  greeting,
  selectModel,
  enterPrompt,
  printDetails,
  invokeModel,
  printResponse,
]);

if (process.argv[1] === fileURLToPath(import.meta.url)) {
  scenario.run();
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - [InvokeModel]
  - [InvokeModelWithResponseStream]

PHP

SDK for PHP

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the [AWS Code Examples Repository](https://github.com/awsdocs/aws-doc-sdk-examples/tree/main/php/example_code/bedrock-runtime#code-examples).

Invoke multiple LLMs on Amazon Bedrock.

```
namespace BedrockRuntime;

class GettingStartedWithBedrockRuntime
{
    protected BedrockRuntimeService $bedrockRuntimeService;

    public function runExample()
    {
        echo "\n";
        echo
 "---------------------------------------------------------------------\n";
        echo "Welcome to the Amazon Bedrock Runtime getting started demo using
 PHP!\n";
        echo
 "---------------------------------------------------------------------\n";

        $clientArgs = [
            'region' => 'us-east-1',
            'version' => 'latest',
            'profile' => 'default',
        ];

        $bedrockRuntimeService = new BedrockRuntimeService($clientArgs);

        $prompt = 'In one paragraph, who are you?';

        echo "\nPrompt: " . $prompt;

        echo "\n\nAnthropic Claude:";
        echo $bedrockRuntimeService->invokeClaude($prompt);
```

```
        echo "\n\nAI21 Labs Jurassic-2: ";
        echo $bedrockRuntimeService->invokeJurassic2($prompt);

        echo "\n\nMeta Llama 2 Chat: ";
        echo $bedrockRuntimeService->invokeLlama2($prompt);

        echo
 "\n---------------------------------------------------------------------\n";

        $image_prompt = 'stylized picture of a cute old steampunk robot';

        echo "\nImage prompt: " . $image_prompt;

        echo "\n\nStability.ai Stable Diffusion XL:\n";
        $diffusionSeed = rand(0, 4294967295);
        $style_preset = 'photographic';
        $base64 = $bedrockRuntimeService->invokeStableDiffusion($image_prompt,
 $diffusionSeed, $style_preset);
        $image_path = $this->saveImage($base64, 'stability.stable-diffusion-xl');
        echo "The generated images have been saved to $image_path";

        echo "\n\nAmazon Titan Image Generation:\n";
        $titanSeed = rand(0, 2147483647);
        $base64 = $bedrockRuntimeService->invokeTitanImage($image_prompt,
 $titanSeed);
        $image_path = $this->saveImage($base64, 'amazon.titan-image-generator-
v1');
        echo "The generated images have been saved to $image_path";
    }

    private function saveImage($base64_image_data, $model_id): string
    {
        $output_dir = "output";

        if (!file_exists($output_dir)) {
            mkdir($output_dir);
        }

        $i = 1;
        while (file_exists("$output_dir/$model_id" . '_' . "$i.png")) {
            $i++;
        }

        $image_data = base64_decode($base64_image_data);
```

```
        $file_path = "$output_dir/$model_id" . '_' . "$i.png";

        $file = fopen($file_path, 'wb');
        fwrite($file, $image_data);
        fclose($file);

        return $file_path;
    }
}
```

- For API details, see the following topics in *AWS SDK for PHP API Reference*.
  - InvokeModel
  - InvokeModelWithResponseStream

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

The following code example shows how to build and orchestrate generative AI applications with Amazon Bedrock and Step Functions.

Python

SDK for Python (Boto3)

The Amazon Bedrock Serverless Prompt Chaining scenario demonstrates how AWS Step Functions, Amazon Bedrock, and Agents for Amazon Bedrock can be used to build and orchestrate complex, serverless, and highly scalable generative AI applications. It contains the following working examples:

- Write an analysis of a given novel for a literature blog. This example illustrates a simple, sequential chain of prompts.
- Generate a short story about a given topic. This example illustrates how the AI can iteratively process a list of items that it previously generated.

- Create an itinerary for a weekend vacation to a given destination. This example illustrates how to parallelize multiple distinct prompts.
- Pitch movie ideas to a human user acting as a movie producer. This example illustrates how to parallelize the same prompt with different inference parameters, how to backtrack to a previous step in the chain, and how to include human input as part of the workflow.
- Plan a meal based on ingredients the user has at hand. This example illustrates how prompt chains can incorporate two distinct AI conversations, with two AI personas engaging in a debate with each other to improve the final outcome.
- Find and summarize today's highest trending GitHub repository. This example illustrates chaining multiple AI agents that interact with external APIs.

For complete source code and instructions to set up and run, see the full project on GitHub.

Services used in this example

- Amazon Bedrock
- Amazon Bedrock Runtime
- Agents for Amazon Bedrock
- Agents for Amazon Bedrock Runtime
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples for Agents for Amazon Bedrock using AWS SDKs

The following code examples show how to use Agents for Amazon Bedrock with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Get started

Hello Agents for Amazon Bedrock

The following code example shows how to get started using Agents for Amazon Bedrock.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import {
  BedrockAgentClient,
  GetAgentCommand,
  paginateListAgents,
} from "@aws-sdk/client-bedrock-agent";

/**
 * @typedef {Object} AgentSummary
 */

/**
 * A simple scenario to demonstrate basic setup and interaction with the Bedrock
 Agents Client.
 *
 * This function first initializes the Amazon Bedrock Agents client for a
 specific region.
 * It then retrieves a list of existing agents using the streamlined paginator
 approach.
```

```
 * For each agent found, it retrieves detailed information using a command
 object.
 *
 * Demonstrates:
 * - Use of the Bedrock Agents client to initialize and communicate with the AWS
 service.
 * - Listing resources in a paginated response pattern.
 * - Accessing an individual resource using a command object.
 *
 * @returns {Promise<void>} A promise that resolves when the function has
 completed execution.
 */
export const main = async () => {
  const region = "us-east-1";

  console.log("=".repeat(68));

  console.log(`Initializing Amazon Bedrock Agents client for ${region}...`);
  const client = new BedrockAgentClient({ region });

  console.log(`Retrieving the list of existing agents...`);
  const paginatorConfig = { client };
  const pages = paginateListAgents(paginatorConfig, {});

  /** @type {AgentSummary[]} */
  const agentSummaries = [];
  for await (const page of pages) {
    agentSummaries.push(...page.agentSummaries);
  }

  console.log(`Found ${agentSummaries.length} agents in ${region}.`);

  if (agentSummaries.length > 0) {
    for (const agentSummary of agentSummaries) {
      const agentId = agentSummary.agentId;
      console.log("=".repeat(68));
      console.log(`Retrieving agent with ID: ${agentId}:`);
      console.log("-".repeat(68));

      const command = new GetAgentCommand({ agentId });
      const response = await client.send(command);
      const agent = response.agent;

      console.log(` Name: ${agent.agentName}`);
```

```
      console.log(` Status: ${agent.agentStatus}`);
      console.log(` ARN: ${agent.agentArn}`);
      console.log(` Foundation model: ${agent.foundationModel}`);
    }
  }
  console.log("=".repeat(68));
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  await main();
}
```

- For API details, see the following topics in *AWS SDK for JavaScript API Reference*.
  - GetAgent
  - ListAgents

Code examples

- Actions for Agents for Amazon Bedrock using AWS SDKs
  - Use CreateAgent with an AWS SDK or command line tool
  - Use CreateAgentActionGroup with an AWS SDK or command line tool
  - Use CreateAgentAlias with an AWS SDK or command line tool
  - Use DeleteAgent with an AWS SDK or command line tool
  - Use DeleteAgentAlias with an AWS SDK or command line tool
  - Use GetAgent with an AWS SDK or command line tool
  - Use ListAgentActionGroups with an AWS SDK or command line tool
  - Use ListAgentKnowledgeBases with an AWS SDK or command line tool
  - Use ListAgents with an AWS SDK or command line tool
  - Use PrepareAgent with an AWS SDK or command line tool
- Scenarios for Agents for Amazon Bedrock using AWS SDKs
  - An end-to-end example showing how to create and invoke Amazon Bedrock agents using an AWS SDK
  - Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Actions for Agents for Amazon Bedrock using AWS SDKs

The following code examples demonstrate how to perform individual Agents for Amazon Bedrock actions with AWS SDKs. These excerpts call the Agents for Amazon Bedrock API and are code excerpts from larger programs that must be run in context. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the Agents for Amazon Bedrock API Reference.

Examples

- Use CreateAgent with an AWS SDK or command line tool
- Use CreateAgentActionGroup with an AWS SDK or command line tool
- Use CreateAgentAlias with an AWS SDK or command line tool
- Use DeleteAgent with an AWS SDK or command line tool
- Use DeleteAgentAlias with an AWS SDK or command line tool
- Use GetAgent with an AWS SDK or command line tool
- Use ListAgentActionGroups with an AWS SDK or command line tool
- Use ListAgentKnowledgeBases with an AWS SDK or command line tool
- Use ListAgents with an AWS SDK or command line tool
- Use PrepareAgent with an AWS SDK or command line tool

Use `CreateAgent` with an AWS SDK or command line tool

The following code examples show how to use `CreateAgent`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an agent.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  CreateAgentCommand,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Creates an Amazon Bedrock Agent.
 *
 * @param {string} agentName - A name for the agent that you create.
 * @param {string} foundationModel - The foundation model to be used by the agent
 you create.
 * @param {string} agentResourceRoleArn - The ARN of the IAM role with
 permissions required by the agent.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<import("@aws-sdk/client-bedrock-agent").Agent>} An object
 containing details of the created agent.
 */
export const createAgent = async (
  agentName,
  foundationModel,
  agentResourceRoleArn,
  region = "us-east-1",
) => {
  const client = new BedrockAgentClient({ region });
```

```
  const command = new CreateAgentCommand({
    agentName,
    foundationModel,
    agentResourceRoleArn,
  });
  const response = await client.send(command);

  return response.agent;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentName and accountId, and roleName with a
 unique name for the new agent,
  // the id of your AWS account, and the name of an existing execution role that
 the agent can use inside your account.
  // For foundationModel, specify the desired model. Ensure to remove the
 brackets '[]' before adding your data.

  // A string (max 100 chars) that can include letters, numbers, dashes '-', and
 underscores '_'.
  const agentName = "[your-bedrock-agent-name]";

  // Your AWS account id.
  const accountId = "[123456789012]";

  // The name of the agent's execution role. It must be prefixed by
 `AmazonBedrockExecutionRoleForAgents_`.
  const roleName = "[AmazonBedrockExecutionRoleForAgents_your-role-name]";

  // The ARN for the agent's execution role.
  // Follow the ARN format: 'arn:aws:iam::account-id:role/role-name'
  const roleArn = `arn:aws:iam::${accountId}:role/${roleName}`;

  // Specify the model for the agent. Change if a different model is preferred.
  const foundationModel = "anthropic.claude-v2";

  // Check for unresolved placeholders in agentName and roleArn.
  checkForPlaceholders([agentName, roleArn]);

  console.log(`Creating a new agent...`);

  const agent = await createAgent(agentName, foundationModel, roleArn);
  console.log(agent);
```

```
}
```

- For API details, see [CreateAgent] in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an agent.

```
    def create_agent(self, agent_name, foundation_model, role_arn, instruction):
        """
        Creates an agent that orchestrates interactions between foundation
 models,
        data sources, software applications, user conversations, and APIs to
 carry
        out tasks to help customers.

        :param agent_name: A name for the agent.
        :param foundation_model: The foundation model to be used for
 orchestration by the agent.
        :param role_arn: The ARN of the IAM role with permissions needed by the
 agent.
        :param instruction: Instructions that tell the agent what it should do
 and how it should
                            interact with users.
        :return: The response from Agents for Bedrock if successful, otherwise
 raises an exception.
        """
        try:
            response = self.client.create_agent(
                agentName=agent_name,
                foundationModel=foundation_model,
                agentResourceRoleArn=role_arn,
                instruction=instruction,
```

```
            )
        except ClientError as e:
            logger.error(f"Error: Couldn't create agent. Here's why: {e}")
            raise
        else:
            return response["agent"]
```

- For API details, see CreateAgent in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateAgentActionGroup` with an AWS SDK or command line tool

The following code example shows how to use `CreateAgentActionGroup`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an agent action group.

```
    def create_agent_action_group(
        self, name, description, agent_id, agent_version, function_arn,
 api_schema
    ):
```

```
        """
        Creates an action group for an agent. An action group defines a set of
actions that an
        agent should carry out for the customer.

        :param name: The name to give the action group.
        :param description: The description of the action group.
        :param agent_id: The unique identifier of the agent for which to create
the action group.
        :param agent_version: The version of the agent for which to create the
action group.
        :param function_arn: The ARN of the Lambda function containing the
business logic that is
                             carried out upon invoking the action.
        :param api_schema: Contains the OpenAPI schema for the action group.
        :return: Details about the action group that was created.
        """
        try:
            response = self.client.create_agent_action_group(
                actionGroupName=name,
                description=description,
                agentId=agent_id,
                agentVersion=agent_version,
                actionGroupExecutor={"lambda": function_arn},
                apiSchema={"payload": api_schema},
            )
            agent_action_group = response["agentActionGroup"]
        except ClientError as e:
            logger.error(f"Error: Couldn't create agent action group. Here's why:
{e}")
            raise
        else:
            return agent_action_group
```

- For API details, see CreateAgentActionGroup in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `CreateAgentAlias` with an AWS SDK or command line tool

The following code example shows how to use `CreateAgentAlias`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create an agent alias.

```
    def create_agent_alias(self, name, agent_id):
        """
        Creates an alias of an agent that can be used to deploy the agent.

        :param name: The name of the alias.
        :param agent_id: The unique identifier of the agent.
        :return: Details about the alias that was created.
        """
        try:
            response = self.client.create_agent_alias(
                agentAliasName=name, agentId=agent_id
            )
            agent_alias = response["agentAlias"]
        except ClientError as e:
            logger.error(f"Couldn't create agent alias. {e}")
            raise
        else:
            return agent_alias
```

- For API details, see CreateAgentAlias in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteAgent` with an AWS SDK or command line tool

The following code examples show how to use `DeleteAgent`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Delete an agent.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  DeleteAgentCommand,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Deletes an Amazon Bedrock Agent.
```

```
 *
 * @param {string} agentId - The unique identifier of the agent to delete.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<import("@aws-sdk/client-bedrock-
agent").DeleteAgentCommandOutput>} An object containing the agent id, the status,
 and some additional metadata.
 */
export const deleteAgent = (agentId, region = "us-east-1") => {
  const client = new BedrockAgentClient({ region });
  const command = new DeleteAgentCommand({ agentId });
  return client.send(command);
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentId with an existing agent's id.
  // Ensure to remove the brackets (`[]`) before adding your data.

  // The agentId must be an alphanumeric string with exactly 10 characters.
  const agentId = "[ABC123DE45]";

  // Check for unresolved placeholders in agentId.
  checkForPlaceholders([agentId]);

  console.log(`Deleting agent with ID ${agentId}...`);

  const response = await deleteAgent(agentId);
  console.log(response);
}
```

- For API details, see DeleteAgent in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Delete an agent.

```
    def delete_agent(self, agent_id):
        """
        Deletes an Amazon Bedrock agent.

        :param agent_id: The unique identifier of the agent to delete.
        :return: The response from Agents for Bedrock if successful, otherwise
 raises an exception.
        """

        try:
            response = self.client.delete_agent(
                agentId=agent_id, skipResourceInUseCheck=False
            )
        except ClientError as e:
            logger.error(f"Couldn't delete agent. {e}")
            raise
        else:
            return response
```

- For API details, see DeleteAgent in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `DeleteAgentAlias` with an AWS SDK or command line tool

The following code example shows how to use `DeleteAgentAlias`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Delete an agent alias.

```
    def delete_agent_alias(self, agent_id, agent_alias_id):
        """
        Deletes an alias of an Amazon Bedrock agent.

        :param agent_id: The unique identifier of the agent that the alias
 belongs to.
        :param agent_alias_id: The unique identifier of the alias to delete.
        :return: The response from Agents for Bedrock if successful, otherwise
 raises an exception.
        """

        try:
            response = self.client.delete_agent_alias(
                agentId=agent_id, agentAliasId=agent_alias_id
            )
        except ClientError as e:
            logger.error(f"Couldn't delete agent alias. {e}")
            raise
        else:
            return response
```

- For API details, see DeleteAgentAlias in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use GetAgent with an AWS SDK or command line tool

The following code examples show how to use `GetAgent`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get an agent.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  GetAgentCommand,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Retrieves the details of an Amazon Bedrock Agent.
 *
 * @param {string} agentId - The unique identifier of the agent.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<import("@aws-sdk/client-bedrock-agent").Agent>} An object
 containing the agent details.
 */
export const getAgent = async (agentId, region = "us-east-1") => {
```

```
  const client = new BedrockAgentClient({ region });

  const command = new GetAgentCommand({ agentId });
  const response = await client.send(command);
  return response.agent;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentId with an existing agent's id.
  // Ensure to remove the brackets '[]' before adding your data.

  // The agentId must be an alphanumeric string with exactly 10 characters.
  const agentId = "[ABC123DE45]";

  // Check for unresolved placeholders in agentId.
  checkForPlaceholders([agentId]);

  console.log(`Retrieving agent with ID ${agentId}...`);

  const agent = await getAgent(agentId);
  console.log(agent);
}
```

- For API details, see GetAgent in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Get an agent.

```
    def get_agent(self, agent_id, log_error=True):
        """
        Gets information about an agent.
```

```
        :param agent_id: The unique identifier of the agent.
        :param log_error: Whether to log any errors that occur when getting the
 agent.
                          If True, errors will be logged to the logger. If False,
 errors
                          will still be raised, but not logged.
        :return: The information about the requested agent.
        """

        try:
            response = self.client.get_agent(agentId=agent_id)
            agent = response["agent"]
        except ClientError as e:
            if log_error:
                logger.error(f"Couldn't get agent {agent_id}. {e}")
            raise
        else:
            return agent
```

- For API details, see GetAgent in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListAgentActionGroups` with an AWS SDK or command line tool

The following code examples show how to use `ListAgentActionGroups`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

JavaScript

SDK for JavaScript (v3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the action groups for an agent.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";
import { checkForPlaceholders } from "../lib/utils.js";

import {
  BedrockAgentClient,
  ListAgentActionGroupsCommand,
  paginateListAgentActionGroups,
} from "@aws-sdk/client-bedrock-agent";

/**
 * Retrieves a list of Action Groups of an agent utilizing the paginator
 function.
 *
 * This function leverages a paginator, which abstracts the complexity of
 pagination, providing
 * a straightforward way to handle paginated results inside a `for await...of`
 loop.
 *
 * @param {string} agentId - The unique identifier of the agent.
 * @param {string} agentVersion - The version of the agent.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<ActionGroupSummary[]>} An array of action group summaries.
 */
export const listAgentActionGroupsWithPaginator = async (
  agentId,
  agentVersion,
  region = "us-east-1",
) => {
```

```
  const client = new BedrockAgentClient({ region });

  // Create a paginator configuration
  const paginatorConfig = {
    client,
    pageSize: 10, // optional, added for demonstration purposes
  };

  const params = { agentId, agentVersion };

  const pages = paginateListAgentActionGroups(paginatorConfig, params);

  // Paginate until there are no more results
  const actionGroupSummaries = [];
  for await (const page of pages) {
    actionGroupSummaries.push(...page.actionGroupSummaries);
  }

  return actionGroupSummaries;
};

/**
 * Retrieves a list of Action Groups of an agent utilizing the
 ListAgentActionGroupsCommand.
 *
 * This function demonstrates the manual approach, sending a command to the
 client and processing the response.
 * Pagination must manually be managed. For a simplified approach that abstracts
 away pagination logic, see
 * the `listAgentActionGroupsWithPaginator()` example below.
 *
 * @param {string} agentId - The unique identifier of the agent.
 * @param {string} agentVersion - The version of the agent.
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<ActionGroupSummary[]>} An array of action group summaries.
 */
export const listAgentActionGroupsWithCommandObject = async (
  agentId,
  agentVersion,
  region = "us-east-1",
) => {
  const client = new BedrockAgentClient({ region });

  let nextToken;
```

```
  const actionGroupSummaries = [];
  do {
    const command = new ListAgentActionGroupsCommand({
      agentId,
      agentVersion,
      nextToken,
      maxResults: 10, // optional, added for demonstration purposes
    });

    /** @type {{actionGroupSummaries: ActionGroupSummary[], nextToken?: string}}
 */
    const response = await client.send(command);

    for (const actionGroup of response.actionGroupSummaries || []) {
      actionGroupSummaries.push(actionGroup);
    }

    nextToken = response.nextToken;
  } while (nextToken);

  return actionGroupSummaries;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  // Replace the placeholders for agentId and agentVersion with an existing
 agent's id and version.
  // Ensure to remove the brackets '[]' before adding your data.

  // The agentId must be an alphanumeric string with exactly 10 characters.
  const agentId = "[ABC123DE45]";

  // A string either containing `DRAFT` or a number with 1-5 digits (e.g., '123'
 or 'DRAFT').
  const agentVersion = "[DRAFT]";

  // Check for unresolved placeholders in agentId and agentVersion.
  checkForPlaceholders([agentId, agentVersion]);

  console.log("=".repeat(68));
  console.log(
    "Listing agent action groups using ListAgentActionGroupsCommand:",
  );
```

```
  for (const actionGroup of await listAgentActionGroupsWithCommandObject(
    agentId,
    agentVersion,
  )) {
    console.log(actionGroup);
  }

  console.log("=".repeat(68));
  console.log(
    "Listing agent action groups using the paginateListAgents function:",
  );
  for (const actionGroup of await listAgentActionGroupsWithPaginator(
    agentId,
    agentVersion,
  )) {
    console.log(actionGroup);
  }
}
```

- For API details, see ListAgentActionGroups in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the action groups for an agent.

```
    def list_agent_action_groups(self, agent_id, agent_version):
        """
        List the action groups for a version of an Amazon Bedrock Agent.

        :param agent_id: The unique identifier of the agent.
        :param agent_version: The version of the agent.
        :return: The list of action group summaries for the version of the agent.
        """
```

```
        try:
            action_groups = []

            paginator = self.client.get_paginator("list_agent_action_groups")
            for page in paginator.paginate(
                agentId=agent_id,
                agentVersion=agent_version,
                PaginationConfig={"PageSize": 10},
            ):
                action_groups.extend(page["actionGroupSummaries"])

        except ClientError as e:
            logger.error(f"Couldn't list action groups. {e}")
            raise
        else:
            return action_groups
```

- For API details, see ListAgentActionGroups in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListAgentKnowledgeBases` with an AWS SDK or command line tool

The following code example shows how to use `ListAgentKnowledgeBases`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the knowledge bases associated with an agent.

```
    def list_agent_knowledge_bases(self, agent_id, agent_version):
        """
        List the knowledge bases associated with a version of an Amazon Bedrock
Agent.

        :param agent_id: The unique identifier of the agent.
        :param agent_version: The version of the agent.
        :return: The list of knowledge base summaries for the version of the
agent.
        """

        try:
            knowledge_bases = []

            paginator = self.client.get_paginator("list_agent_knowledge_bases")
            for page in paginator.paginate(
                agentId=agent_id,
                agentVersion=agent_version,
                PaginationConfig={"PageSize": 10},
            ):
                knowledge_bases.extend(page["agentKnowledgeBaseSummaries"])

        except ClientError as e:
            logger.error(f"Couldn't list knowledge bases. {e}")
            raise
        else:
            return knowledge_bases
```

- For API details, see ListAgentKnowledgeBases in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `ListAgents` with an AWS SDK or command line tool

The following code examples show how to use `ListAgents`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the agents belonging to an account.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import { fileURLToPath } from "url";

import {
  BedrockAgentClient,
  ListAgentsCommand,
  paginateListAgents,
} from "@aws-sdk/client-bedrock-agent";

/**
```

```
 * Retrieves a list of available Amazon Bedrock agents utilizing the paginator
 function.
 *
 * This function leverages a paginator, which abstracts the complexity of
 pagination, providing
 * a straightforward way to handle paginated results inside a `for await...of`
 loop.
 *
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<AgentSummary[]>} An array of agent summaries.
 */
export const listAgentsWithPaginator = async (region = "us-east-1") => {
  const client = new BedrockAgentClient({ region });

  const paginatorConfig = {
    client,
    pageSize: 10, // optional, added for demonstration purposes
  };

  const pages = paginateListAgents(paginatorConfig, {});

  // Paginate until there are no more results
  const agentSummaries = [];
  for await (const page of pages) {
    agentSummaries.push(...page.agentSummaries);
  }

  return agentSummaries;
};

/**
 * Retrieves a list of available Amazon Bedrock agents utilizing the
 ListAgentsCommand.
 *
 * This function demonstrates the manual approach, sending a command to the
 client and processing the response.
 * Pagination must manually be managed. For a simplified approach that abstracts
 away pagination logic, see
 * the `listAgentsWithPaginator()` example below.
 *
 * @param {string} [region='us-east-1'] - The AWS region in use.
 * @returns {Promise<AgentSummary[]>} An array of agent summaries.
 */
export const listAgentsWithCommandObject = async (region = "us-east-1") => {
```

```
  const client = new BedrockAgentClient({ region });

  let nextToken;
  const agentSummaries = [];
  do {
    const command = new ListAgentsCommand({
      nextToken,
      maxResults: 10, // optional, added for demonstration purposes
    });

    /** @type {{agentSummaries: AgentSummary[], nextToken?: string}} */
    const paginatedResponse = await client.send(command);

    agentSummaries.push(...(paginatedResponse.agentSummaries || []));

    nextToken = paginatedResponse.nextToken;
  } while (nextToken);

  return agentSummaries;
};

// Invoke main function if this file was run directly.
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  console.log("=".repeat(68));
  console.log("Listing agents using ListAgentsCommand:");
  for (const agent of await listAgentsWithCommandObject()) {
    console.log(agent);
  }

  console.log("=".repeat(68));
  console.log("Listing agents using the paginateListAgents function:");
  for (const agent of await listAgentsWithPaginator()) {
    console.log(agent);
  }
}
```

- For API details, see [ListAgents](#) in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

List the agents belonging to an account.

```
    def list_agents(self):
        """
        List the available Amazon Bedrock Agents.

        :return: The list of available bedrock agents.
        """

        try:
            all_agents = []

            paginator = self.client.get_paginator("list_agents")
            for page in paginator.paginate(PaginationConfig={"PageSize": 10}):
                all_agents.extend(page["agentSummaries"])

        except ClientError as e:
            logger.error(f"Couldn't list agents. {e}")
            raise
        else:
            return all_agents
```

- For API details, see ListAgents in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Use `PrepareAgent` with an AWS SDK or command line tool

The following code example shows how to use `PrepareAgent`.

Action examples are code excerpts from larger programs and must be run in context. You can see this action in context in the following code example:

- Create and invoke an agent

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Prepare an agent for internal testing.

```
    def prepare_agent(self, agent_id):
        """
        Creates a DRAFT version of the agent that can be used for internal
 testing.

        :param agent_id: The unique identifier of the agent to prepare.
        :return: The response from Agents for Bedrock if successful, otherwise
 raises an exception.
        """
        try:
            prepared_agent_details = self.client.prepare_agent(agentId=agent_id)
        except ClientError as e:
            logger.error(f"Couldn't prepare agent. {e}")
            raise
        else:
            return prepared_agent_details
```

- For API details, see PrepareAgent in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Agents for Amazon Bedrock using AWS SDKs

The following code examples show you how to implement common scenarios in Agents for Amazon Bedrock with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Agents for Amazon Bedrock. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- An end-to-end example showing how to create and invoke Amazon Bedrock agents using an AWS SDK
- Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

An end-to-end example showing how to create and invoke Amazon Bedrock agents using an AWS SDK

The following code example shows how to:

- Create an execution role for the agent.
- Create the agent and deploy a DRAFT version.
- Create a Lambda function that implements the agent's capabilities.
- Create an action group that connects the agent to the Lambda function.
- Deploy the fully configured agent.
- Invoke the agent with user-provided prompts.
- Delete all created resources.

Python

SDK for Python (Boto3)

Note

There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Create and invoke an agent.

```
REGION = "us-east-1"
ROLE_POLICY_NAME = "agent_permissions"


class BedrockAgentScenarioWrapper:
    """Runs a scenario that shows how to get started using Agents for Amazon
 Bedrock."""

    def __init__(
        self, bedrock_agent_client, runtime_client, lambda_client, iam_resource,
 postfix
    ):
        self.iam_resource = iam_resource
        self.lambda_client = lambda_client
        self.bedrock_agent_runtime_client = runtime_client
        self.postfix = postfix

        self.bedrock_wrapper = BedrockAgentWrapper(bedrock_agent_client)

        self.agent = None
        self.agent_alias = None
        self.agent_role = None
        self.prepared_agent_details = None
        self.lambda_role = None
        self.lambda_function = None

    def run_scenario(self):
        print("=" * 88)
        print("Welcome to the Amazon Bedrock Agents demo.")
        print("=" * 88)
```

```
        # Query input from user
        print("Let's start with creating an agent:")
        print("-" * 40)
        name, foundation_model = self._request_name_and_model_from_user()
        print("-" * 40)

        # Create an execution role for the agent
        self.agent_role = self._create_agent_role(foundation_model)

        # Create the agent
        self.agent = self._create_agent(name, foundation_model)

        # Prepare a DRAFT version of the agent
        self.prepared_agent_details = self._prepare_agent()

        # Create the agent's Lambda function
        self.lambda_function = self._create_lambda_function()

        # Configure permissions for the agent to invoke the Lambda function
        self._allow_agent_to_invoke_function()
        self._let_function_accept_invocations_from_agent()

        # Create an action group to connect the agent with the Lambda function
        self._create_agent_action_group()

        # If the agent has been modified or any components have been added,
 prepare the agent again
        components = [self._get_agent()]
        components += self._get_agent_action_groups()
        components += self._get_agent_knowledge_bases()

        latest_update = max(component["updatedAt"] for component in components)
        if latest_update > self.prepared_agent_details["preparedAt"]:
            self.prepared_agent_details = self._prepare_agent()

        # Create an agent alias
        self.agent_alias = self._create_agent_alias()

        # Test the agent
        self._chat_with_agent(self.agent_alias)

        print("=" * 88)
        print("Thanks for running the demo!\n")
```

```
        if q.ask("Do you want to delete the created resources? [y/N] ",
 q.is_yesno):
            self._delete_resources()
            print("=" * 88)
            print(
                "All demo resources have been deleted. Thanks again for running
 the demo!"
            )
        else:
            self._list_resources()
            print("=" * 88)
            print("Thanks again for running the demo!")

    def _request_name_and_model_from_user(self):
        existing_agent_names = [
            agent["agentName"] for agent in self.bedrock_wrapper.list_agents()
        ]

        while True:
            name = q.ask("Enter an agent name: ", self.is_valid_agent_name)
            if name.lower() not in [n.lower() for n in existing_agent_names]:
                break
            print(
                f"Agent {name} conflicts with an existing agent. Please use a
 different name."
            )

        models = ["anthropic.claude-instant-v1", "anthropic.claude-v2"]
        model_id = models[
            q.choose("Which foundation model would you like to use? ", models)
        ]

        return name, model_id

    def _create_agent_role(self, model_id):
        role_name = f"AmazonBedrockExecutionRoleForAgents_{self.postfix}"
        model_arn = f"arn:aws:bedrock:{REGION}::foundation-model/{model_id}*"

        print("Creating an an execution role for the agent...")

        try:
            role = self.iam_resource.create_role(
                RoleName=role_name,
                AssumeRolePolicyDocument=json.dumps(
```

```
                    {
                        "Version": "2012-10-17",
                        "Statement": [
                            {
                                "Effect": "Allow",
                                "Principal": {"Service":
"bedrock.amazonaws.com"},
                                "Action": "sts:AssumeRole",
                            }
                        ],
                    }
                ),
            )

            role.Policy(ROLE_POLICY_NAME).put(
                PolicyDocument=json.dumps(
                    {
                        "Version": "2012-10-17",
                        "Statement": [
                            {
                                "Effect": "Allow",
                                "Action": "bedrock:InvokeModel",
                                "Resource": model_arn,
                            }
                        ],
                    }
                )
            )
        except ClientError as e:
            logger.error(f"Couldn't create role {role_name}. Here's why: {e}")
            raise

        return role

    def _create_agent(self, name, model_id):
        print("Creating the agent...")

        instruction = """
            You are a friendly chat bot. You have access to a function called
that returns
            information about the current date and time. When responding with
date or time,
            please make sure to add the timezone UTC.
            """
```

```
        agent = self.bedrock_wrapper.create_agent(
            agent_name=name,
            foundation_model=model_id,
            instruction=instruction,
            role_arn=self.agent_role.arn,
        )
        self._wait_for_agent_status(agent["agentId"], "NOT_PREPARED")

        return agent

    def _prepare_agent(self):
        print("Preparing the agent...")

        agent_id = self.agent["agentId"]
        prepared_agent_details = self.bedrock_wrapper.prepare_agent(agent_id)
        self._wait_for_agent_status(agent_id, "PREPARED")

        return prepared_agent_details

    def _create_lambda_function(self):
        print("Creating the Lambda function...")

        function_name = f"AmazonBedrockExampleFunction_{self.postfix}"

        self.lambda_role = self._create_lambda_role()

        try:
            deployment_package = self._create_deployment_package(function_name)

            lambda_function = self.lambda_client.create_function(
                FunctionName=function_name,
                Description="Lambda function for Amazon Bedrock example",
                Runtime="python3.11",
                Role=self.lambda_role.arn,
                Handler=f"{function_name}.lambda_handler",
                Code={"ZipFile": deployment_package},
                Publish=True,
            )

            waiter = self.lambda_client.get_waiter("function_active_v2")
            waiter.wait(FunctionName=function_name)

        except ClientError as e:
            logger.error(
```

```
                f"Couldn't create Lambda function {function_name}. Here's why:
 {e}"
            )
            raise

        return lambda_function

    def _create_lambda_role(self):
        print("Creating an execution role for the Lambda function...")

        role_name = f"AmazonBedrockExecutionRoleForLambda_{self.postfix}"

        try:
            role = self.iam_resource.create_role(
                RoleName=role_name,
                AssumeRolePolicyDocument=json.dumps(
                    {
                        "Version": "2012-10-17",
                        "Statement": [
                            {
                                "Effect": "Allow",
                                "Principal": {"Service": "lambda.amazonaws.com"},
                                "Action": "sts:AssumeRole",
                            }
                        ],
                    }
                ),
            )
            role.attach_policy(
                PolicyArn="arn:aws:iam::aws:policy/service-role/
AWSLambdaBasicExecutionRole"
            )
            print(f"Created role {role_name}")
        except ClientError as e:
            logger.error(f"Couldn't create role {role_name}. Here's why: {e}")
            raise

        print("Waiting for the execution role to be fully propagated...")
        wait(10)

        return role

    def _allow_agent_to_invoke_function(self):
        policy = self.iam_resource.RolePolicy(
```

```
            self.agent_role.role_name, ROLE_POLICY_NAME
        )
        doc = policy.policy_document
        doc["Statement"].append(
            {
                "Effect": "Allow",
                "Action": "lambda:InvokeFunction",
                "Resource": self.lambda_function["FunctionArn"],
            }
        )

 self.agent_role.Policy(ROLE_POLICY_NAME).put(PolicyDocument=json.dumps(doc))

    def _let_function_accept_invocations_from_agent(self):
        try:
            self.lambda_client.add_permission(
                FunctionName=self.lambda_function["FunctionName"],
                SourceArn=self.agent["agentArn"],
                StatementId="BedrockAccess",
                Action="lambda:InvokeFunction",
                Principal="bedrock.amazonaws.com",
            )
        except ClientError as e:
            logger.error(
                f"Couldn't grant Bedrock permission to invoke the Lambda
 function. Here's why: {e}"
            )
            raise

    def _create_agent_action_group(self):
        print("Creating an action group for the agent...")

        try:
            with open("./scenario_resources/api_schema.yaml") as file:
                self.bedrock_wrapper.create_agent_action_group(
                    name="current_date_and_time",
                    description="Gets the current date and time.",
                    agent_id=self.agent["agentId"],
                    agent_version=self.prepared_agent_details["agentVersion"],
                    function_arn=self.lambda_function["FunctionArn"],
                    api_schema=json.dumps(yaml.safe_load(file)),
                )
        except ClientError as e:
            logger.error(f"Couldn't create agent action group. Here's why: {e}")
```

```
            raise

    def _get_agent(self):
        return self.bedrock_wrapper.get_agent(self.agent["agentId"])

    def _get_agent_action_groups(self):
        return self.bedrock_wrapper.list_agent_action_groups(
            self.agent["agentId"], self.prepared_agent_details["agentVersion"]
        )

    def _get_agent_knowledge_bases(self):
        return self.bedrock_wrapper.list_agent_knowledge_bases(
            self.agent["agentId"], self.prepared_agent_details["agentVersion"]
        )

    def _create_agent_alias(self):
        print("Creating an agent alias...")

        agent_alias_name = "test_agent_alias"
        agent_alias = self.bedrock_wrapper.create_agent_alias(
            agent_alias_name, self.agent["agentId"]
        )

        self._wait_for_agent_status(self.agent["agentId"], "PREPARED")

        return agent_alias

    def _wait_for_agent_status(self, agent_id, status):
        while self.bedrock_wrapper.get_agent(agent_id)["agentStatus"] != status:
            wait(2)

    def _chat_with_agent(self, agent_alias):
        print("-" * 88)
        print("The agent is ready to chat.")
        print("Try asking for the date or time. Type 'exit' to quit.")

        # Create a unique session ID for the conversation
        session_id = uuid.uuid4().hex

        while True:
            prompt = q.ask("Prompt: ", q.non_empty)

            if prompt == "exit":
                break
```

```
            response = asyncio.run(self._invoke_agent(agent_alias, prompt,
 session_id))

            print(f"Agent: {response}")

    async def _invoke_agent(self, agent_alias, prompt, session_id):
        response = self.bedrock_agent_runtime_client.invoke_agent(
            agentId=self.agent["agentId"],
            agentAliasId=agent_alias["agentAliasId"],
            sessionId=session_id,
            inputText=prompt,
        )

        completion = ""

        for event in response.get("completion"):
            chunk = event["chunk"]
            completion += chunk["bytes"].decode()

        return completion

    def _delete_resources(self):
        if self.agent:
            agent_id = self.agent["agentId"]

            if self.agent_alias:
                agent_alias_id = self.agent_alias["agentAliasId"]
                print("Deleting agent alias...")
                self.bedrock_wrapper.delete_agent_alias(agent_id, agent_alias_id)

            print("Deleting agent...")
            agent_status = self.bedrock_wrapper.delete_agent(agent_id)
["agentStatus"]
            while agent_status == "DELETING":
                wait(5)
                try:
                    agent_status = self.bedrock_wrapper.get_agent(
                        agent_id, log_error=False
                    )["agentStatus"]
                except ClientError as err:
                    if err.response["Error"]["Code"] ==
 "ResourceNotFoundException":
                        agent_status = "DELETED"
```

```
        if self.lambda_function:
            name = self.lambda_function["FunctionName"]
            print(f"Deleting function '{name}'...")
            self.lambda_client.delete_function(FunctionName=name)

        if self.agent_role:
            print(f"Deleting role '{self.agent_role.role_name}'...")
            self.agent_role.Policy(ROLE_POLICY_NAME).delete()
            self.agent_role.delete()

        if self.lambda_role:
            print(f"Deleting role '{self.lambda_role.role_name}'...")
            for policy in self.lambda_role.attached_policies.all():
                policy.detach_role(RoleName=self.lambda_role.role_name)
            self.lambda_role.delete()

    def _list_resources(self):
        print("-" * 40)
        print(f"Here is the list of created resources in '{REGION}'.")
        print("Make sure you delete them once you're done to avoid unnecessary
 costs.")
        if self.agent:
            print(f"Bedrock Agent:   {self.agent['agentName']}")
        if self.lambda_function:
            print(f"Lambda function: {self.lambda_function['FunctionName']}")
        if self.agent_role:
            print(f"IAM role:        {self.agent_role.role_name}")
        if self.lambda_role:
            print(f"IAM role:        {self.lambda_role.role_name}")

    @staticmethod
    def is_valid_agent_name(answer):
        valid_regex = r"^[a-zA-Z0-9_-]{1,100}$"
        return (
            answer
            if answer and len(answer) <= 100 and re.match(valid_regex, answer)
            else None,
            "I need a name for the agent, please. Valid characters are a-z, A-Z,
 0-9, _ (underscore) and - (hyphen).",
        )

    @staticmethod
    def _create_deployment_package(function_name):
```

```
        buffer = io.BytesIO()
        with zipfile.ZipFile(buffer, "w") as zipped:
            zipped.write(
                "./scenario_resources/lambda_function.py", f"{function_name}.py"
            )
        buffer.seek(0)
        return buffer.read()


if __name__ == "__main__":
    logging.basicConfig(level=logging.INFO, format="%(levelname)s: %(message)s")

    postfix = "".join(
        random.choice(string.ascii_lowercase + "0123456789") for _ in range(8)
    )
    scenario = BedrockAgentScenarioWrapper(
        bedrock_agent_client=boto3.client(
            service_name="bedrock-agent", region_name=REGION
        ),
        runtime_client=boto3.client(
            service_name="bedrock-agent-runtime", region_name=REGION
        ),
        lambda_client=boto3.client(service_name="lambda", region_name=REGION),
        iam_resource=boto3.resource("iam"),
        postfix=postfix,
    )
    try:
        scenario.run_scenario()
    except Exception as e:
        logging.exception(f"Something went wrong with the demo. Here's what:
 {e}")
```

- For API details, see the following topics in *AWS SDK for Python (Boto3) API Reference*.
  - CreateAgent
  - CreateAgentActionGroup
  - CreateAgentAlias
  - DeleteAgent
  - DeleteAgentAlias
  - GetAgent

- ListAgentActionGroups
- ListAgentKnowledgeBases
- ListAgents
- PrepareAgent

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

The following code example shows how to build and orchestrate generative AI applications with Amazon Bedrock and Step Functions.

Python

SDK for Python (Boto3)

The Amazon Bedrock Serverless Prompt Chaining scenario demonstrates how AWS Step Functions, Amazon Bedrock, and Agents for Amazon Bedrock can be used to build and orchestrate complex, serverless, and highly scalable generative AI applications. It contains the following working examples:

- Write an analysis of a given novel for a literature blog. This example illustrates a simple, sequential chain of prompts.
- Generate a short story about a given topic. This example illustrates how the AI can iteratively process a list of items that it previously generated.
- Create an itinerary for a weekend vacation to a given destination. This example illustrates how to parallelize multiple distinct prompts.
- Pitch movie ideas to a human user acting as a movie producer. This example illustrates how to parallelize the same prompt with different inference parameters, how to backtrack to a previous step in the chain, and how to include human input as part of the workflow.
- Plan a meal based on ingredients the user has at hand. This example illustrates how prompt chains can incorporate two distinct AI conversations, with two AI personas engaging in a debate with each other to improve the final outcome.

- Find and summarize today's highest trending GitHub repository. This example illustrates chaining multiple AI agents that interact with external APIs.

For complete source code and instructions to set up and run, see the full project on GitHub.

Services used in this example

- Amazon Bedrock
- Amazon Bedrock Runtime
- Agents for Amazon Bedrock
- Agents for Amazon Bedrock Runtime
- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples for Agents for Amazon Bedrock Runtime using AWS SDKs

The following code examples show how to use Agents for Amazon Bedrock Runtime with an AWS software development kit (SDK).

Actions are code excerpts from larger programs and must be run in context. While actions show you how to call individual service functions, you can see actions in context in their related scenarios and cross-service examples.

Scenarios are code examples that show you how to accomplish a specific task by calling multiple functions within the same service.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Code examples

- Actions for Agents for Amazon Bedrock Runtime using AWS SDKs
  - Use InvokeAgent with an AWS SDK or command line tool
- Scenarios for Agents for Amazon Bedrock Runtime using AWS SDKs

- Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Actions for Agents for Amazon Bedrock Runtime using AWS SDKs

The following code examples demonstrate how to perform individual Agents for Amazon Bedrock Runtime actions with AWS SDKs. These excerpts call the Agents for Amazon Bedrock Runtime API and are code excerpts from larger programs that must be run in context. Each example includes a link to GitHub, where you can find instructions for setting up and running the code.

The following examples include only the most commonly used actions. For a complete list, see the Agents for Amazon Bedrock Runtime API Reference.

Examples

- Use InvokeAgent with an AWS SDK or command line tool

Use InvokeAgent with an AWS SDK or command line tool

The following code examples show how to use `InvokeAgent`.

JavaScript

SDK for JavaScript (v3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

```
// Copyright Amazon.com, Inc. or its affiliates. All Rights Reserved.
// SPDX-License-Identifier: Apache-2.0

import {
  BedrockAgentRuntimeClient,
  InvokeAgentCommand,
} from "@aws-sdk/client-bedrock-agent-runtime";

/**
 * @typedef {Object} ResponseBody
```

```
 * @property {string} completion
 */

/**
 * Invokes a Bedrock agent to run an inference using the input
 * provided in the request body.
 *
 * @param {string} prompt - The prompt that you want the Agent to complete.
 * @param {string} sessionId - An arbitrary identifier for the session.
 */
export const invokeBedrockAgent = async (prompt, sessionId) => {
  const client = new BedrockAgentRuntimeClient({ region: "us-east-1" });
  // const client = new BedrockAgentRuntimeClient({
  //   region: "us-east-1",
  //   credentials: {
  //     accessKeyId: "accessKeyId", // permission to invoke agent
  //     secretAccessKey: "accessKeySecret",
  //   },
  // });

  const agentId = "AJBHXXILZN";
  const agentAliasId = "AVKP1ITZAA";

  const command = new InvokeAgentCommand({
    agentId,
    agentAliasId,
    sessionId,
    inputText: prompt,
  });

  try {
    let completion = "";
    const response = await client.send(command);

    if (response.completion === undefined) {
      throw new Error("Completion is undefined");
    }

    for await (let chunkEvent of response.completion) {
      const chunk = chunkEvent.chunk;
      console.log(chunk);
      const decodedResponse = new TextDecoder("utf-8").decode(chunk.bytes);
      completion += decodedResponse;
    }
```

```
    return { sessionId: sessionId, completion };
  } catch (err) {
    console.error(err);
  }
};

// Call function if run directly
import { fileURLToPath } from "url";
if (process.argv[1] === fileURLToPath(import.meta.url)) {
  const result = await invokeBedrockAgent("I need help.", "123");
  console.log(result);
}
```

- For API details, see InvokeAgent in *AWS SDK for JavaScript API Reference*.

Python

SDK for Python (Boto3)

> **Note**
>
> There's more on GitHub. Find the complete example and learn how to set up and run in the AWS Code Examples Repository.

Invoke an agent.

```
    def invoke_agent(self, agent_id, agent_alias_id, session_id, prompt):
        """
        Sends a prompt for the agent to process and respond to.

        :param agent_id: The unique identifier of the agent to use.
        :param agent_alias_id: The alias of the agent to use.
        :param session_id: The unique identifier of the session. Use the same
 value across requests
                           to continue the same conversation.
        :param prompt: The prompt that you want Claude to complete.
        :return: Inference response from the model.
        """
```

```
        try:
            response = self.agents_runtime_client.invoke_agent(
                agentId=agent_id,
                agentAliasId=agent_alias_id,
                sessionId=session_id,
                inputText=prompt,
            )

            completion = ""

            for event in response.get("completion"):
                chunk = event["chunk"]
                completion = completion + chunk["bytes"].decode()

        except ClientError as e:
            logger.error(f"Couldn't invoke agent. {e}")
            raise

        return completion
```

- For API details, see InvokeAgent in *AWS SDK for Python (Boto3) API Reference*.

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Scenarios for Agents for Amazon Bedrock Runtime using AWS SDKs

The following code examples show you how to implement common scenarios in Agents for Amazon Bedrock Runtime with AWS SDKs. These scenarios show you how to accomplish specific tasks by calling multiple functions within Agents for Amazon Bedrock Runtime. Each scenario includes a link to GitHub, where you can find instructions on how to set up and run the code.

Examples

- Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

Build and orchestrate generative AI applications with Amazon Bedrock and Step Functions

The following code example shows how to build and orchestrate generative AI applications with Amazon Bedrock and Step Functions.

Python

SDK for Python (Boto3)

The Amazon Bedrock Serverless Prompt Chaining scenario demonstrates how AWS Step Functions, Amazon Bedrock, and Agents for Amazon Bedrock can be used to build and orchestrate complex, serverless, and highly scalable generative AI applications. It contains the following working examples:

- Write an analysis of a given novel for a literature blog. This example illustrates a simple, sequential chain of prompts.
- Generate a short story about a given topic. This example illustrates how the AI can iteratively process a list of items that it previously generated.
- Create an itinerary for a weekend vacation to a given destination. This example illustrates how to parallelize multiple distinct prompts.
- Pitch movie ideas to a human user acting as a movie producer. This example illustrates how to parallelize the same prompt with different inference parameters, how to backtrack to a previous step in the chain, and how to include human input as part of the workflow.
- Plan a meal based on ingredients the user has at hand. This example illustrates how prompt chains can incorporate two distinct AI conversations, with two AI personas engaging in a debate with each other to improve the final outcome.
- Find and summarize today's highest trending GitHub repository. This example illustrates chaining multiple AI agents that interact with external APIs.

For complete source code and instructions to set up and run, see the full project on GitHub.

Services used in this example

- Amazon Bedrock
- Amazon Bedrock Runtime
- Agents for Amazon Bedrock
- Agents for Amazon Bedrock Runtime

- Step Functions

For a complete list of AWS SDK developer guides and code examples, see Using this service with an AWS SDK. This topic also includes information about getting started and details about previous SDK versions.

Amazon Bedrock abuse detection

AWS is committed to the responsible use of AI. To help prevent potential misuse, Amazon Bedrock implements automated abuse detection mechanisms to identify potential violations of AWS's Acceptable Use Policy (AUP) and Service Terms, including the Responsible AI Policy or a third-party model provider's AUP.

Our abuse detection mechanisms are fully automated, so there is no human review of, or access to, user inputs or model outputs.

Automated abuse detection includes:

- **Categorize content** — We use classifiers to detect harmful content (such as content that incites violence) in user inputs and model outputs. A classifier is an algorithm that processes model inputs and outputs, and assigns type of harm and level of confidence. We may run these classifiers on both Titan and third-party model usage. The classification process is automated and does not involve human review of user inputs or model outputs.
- **Identify patterns** — We use classifier metrics to identify potential violations and recurring behavior. We may compile and share anonymized classifier metrics with third-party model providers. Amazon Bedrock does not store user input or model output and does not share these with third-party model providers.
- **Detecting and blocking child sexual abuse material (CSAM)** — You are responsible for the content you (and your end users) upload to Amazon Bedrock and must ensure this content is free from illegal images. To help stop the dissemination of CSAM, Amazon Bedrock may use automated abuse detection mechanisms (such as hash matching technology or classifiers) to detect apparent CSAM. If Amazon Bedrock detects apparent CSAM in your image inputs, Amazon Bedrock will block the request and you will receive an automated error message. Amazon Bedrock may also file a report with the National Center for Missing and Exploited Children (NCMEC) or a relevant authority. We take CSAM seriously and will continue to update our detection, blocking, and reporting mechanisms. You might be required by applicable laws to take additional actions, and you are responsible for those actions.

Once our automated abuse detection mechanisms identify potential violations, we may request information about your use of Amazon Bedrock and compliance with our terms of service or a third-party provider's AUP. In the event that you are unwilling or unable to comply with these terms or policies, AWS may suspend your access to Amazon Bedrock.

Contact AWS Support if you have additional questions. For more information, see the Amazon Bedrock FAQs.

Creating Amazon Bedrock resources with AWS CloudFormation

Amazon Bedrock is integrated with AWS CloudFormation, a service that helps you to model and set up your AWS resources so that you can spend less time creating and managing your resources and infrastructure. You create a template that describes all the AWS resources that you want (such as Amazon Bedrock agents or Amazon Bedrock knowledge bases), and AWS CloudFormation provisions and configures those resources for you.

When you use AWS CloudFormation, you can reuse your template to set up your Amazon Bedrock resources consistently and repeatedly. Describe your resources once, and then provision the same resources over and over in multiple AWS accounts and Regions.

Amazon Bedrock and AWS CloudFormation templates

To provision and configure resources for Amazon Bedrock and related services, you must understand AWS CloudFormation templates. Templates are formatted text files in JSON or YAML. These templates describe the resources that you want to provision in your AWS CloudFormation stacks. If you're unfamiliar with JSON or YAML, you can use AWS CloudFormation Designer to help you get started with AWS CloudFormation templates. For more information, see What is AWS CloudFormation Designer? in the *AWS CloudFormation User Guide*.

Amazon Bedrock supports creating the following resources in AWS CloudFormation.

- AWS::Bedrock::Agent
- AWS::Bedrock::AgentAlias
- AWS::Bedrock::KnowledgeBase
- AWS::Bedrock::DataSource

For more information, including examples of JSON and YAML templates for Amazon Bedrock agents or Amazon Bedrock knowledge bases, see the Amazon Bedrock resource type reference in the *AWS CloudFormation User Guide*.

Learn more about AWS CloudFormation

To learn more about AWS CloudFormation, see the following resources:

- AWS CloudFormation
- AWS CloudFormation User Guide
- AWS CloudFormation API Reference
- AWS CloudFormation Command Line Interface User Guide

Quotas for Amazon Bedrock

Your AWS account has default quotas, formerly referred to as limits, for each AWS service. Unless otherwise noted, each quota is Region-specific within your AWS account. Some quotas may be adjustable. The following list explains the meaning of the **Adjustable through Service Quotas** column in the following tables:

- If a quota is marked as **Yes**, you can adjust it by following the steps at Requesting a Quota Increase in the *Service Quotas User Guide*.
- If a quota is marked as **No**, you might be able to request a quota increase in one of the following ways:
  - To request a quota increase for an on-demand runtime quota, contact your AWS account manager. If you don't have an AWS account manager, you can't increase your quota at this time.
  - To request other quota increases, submit a request through the limit increase form to be considered for an increase.

> **Note**
>
> Due to overwhelming demand, priority will be given to customers who generate traffic that consumes their existing quota allocation. Your request might be denied if you don't meet this condition.

Some quotas differ by model. Unless specified otherwise, a quota applies to all versions of a model.

Select a topic to learn more about quotas for it.

Topics

- Model evaluation job quotas

Runtime quotas

Latency differs by model and is directly proportional to the following conditions:

- The number of input and output tokens
- The total number of ongoing on-demand requests by all customers at the time.

The following quotas apply when you carry out model inference. These quotas consider the combined sum for InvokeModel and InvokeModelWithResponseStream requests.

To increase throughput, purchase Provisioned Throughput for Amazon Bedrock.

> **Note**
>
> If a quota is marked as not adjustable through Service Quotas, you can contact your AWS account manager to request a quota increase. If you don't have an AWS account manager, you can't increase your quota at this time. Due to overwhelming demand, priority will be given to customers who generate traffic that consumes their existing quota allocation. Your request might be denied if you don't meet this condition.

Model	Requests processed per minute	Tokens processed per minute	Adjustable through Service Quotas (see note above table)
AI21 Labs Jurassic-2 Mid	400	300,000	No
AI21 Labs Jurassic-2 Ultra	100	300,000	No
Amazon Titan Embeddings G1 - Text	2,000	300,000	No

Model	Requests processed per minute	Tokens processed per minute	Adjustable through Service Quotas (see note above table)
Amazon Titan Image Generator G1	60	N/A	No
Amazon Titan Multimodal Embeddings G1	2,000	300,000	No
Amazon Titan Text G1 - Express	400	300,000	No
Amazon Titan Text G1 - Lite	800	300,000	No
Anthropic Claude Instant	1,000	1,000,000	No
Anthropic Claude 2.x	500	500,000	No
Anthropic Claude 3 Sonnet	500	1,000,000	No
Anthropic Claude 3 Haiku	1,000	2,000,000	No
Anthropic Claude 3 Opus	50	400,000	No
Cohere Command	400	300,000	No
Cohere Command Light	800	300,000	No
Cohere Embed (English)	2,000	300,000	No

Model	Requests processed per minute	Tokens processed per minute	Adjustable through Service Quotas (see note above table)
Cohere Embed (Multilingual)	2,000	300,000	No
Meta Llama 2 13B	800	300,000	No
Meta Llama 2 70B	400	300,000	No
Meta Llama 3 8b Instruct	800	300,000	No
Meta Llama 3 70b Instruct	400	300,000	No
Mistral AI Mistral 7B Instruct	800	300,000	No
Mistral AI Mistral Large	400	300,000	No
Mistral AI Mixtral 8X7B Instruct	400	300,000	No
Stable Diffusion XL	60	N/A	No

Select a tab to see model-specific inference quotas.

Amazon Titan Text models

Description	Value	Adjustable through Service Quotas (see note above table)
Text prompt length, in characters	42,000	No

Amazon Titan Image Generator G1

Description	Value	Adjustable through Service Quotas (see note above table)
Text prompt length, in characters	1,024	No
Input image size	5 MB	No
Input image height in pixels (inpainting/outpainting)	1,024	No
Input image width in pixels (inpainting/outpainting)	1,024	No
Input image height in pixels (image variation)	4,096	No
Input image width in pixels (image variation)	4,096	No
Input image total pixels	12,582,912	No

Amazon Titan Embeddings G1 - Text

Description	Value	Adjustable through Service Quotas (see note above table)
Text input length, in characters	50,000	No

Amazon Titan Multimodal Embeddings G1

Description	Value	Adjustable through Service Quotas (see note above table)
Text input length, in characters	100,000	No
Base64-encoded string of image, in characters	25,000,000	No

Batch inference quotas

The following quotas apply when you run batch inference. The quotas depend on the modality of the input and output data.

Note

If a quota is marked as not adjustable through Service Quotas, you can submit a request through the limit increase form to be considered for an increase.

Modality	Minimum file size	Maximum file size	Adjustable through Service Quotas (see note above table)
Text to embeddings	75 MB	500 MB	No
Text to text	20 MB	150 MB	No
Text/image to image	1 MB	50 MB	No

Knowledge base quotas

The following quotas apply to Knowledge bases for Amazon Bedrock.

Note

If a quota is marked as not adjustable through Service Quotas, you can submit a request through the limit increase form to be considered for an increase.

Description	Maximum	Adjustable through Service Quotas (see note above table)
Knowledge bases per account per region	50	No
Data source file size (source document)	50 MB	No
Data source file size (metadata file)	10 KB	No
Data source chunk size (Titan Text G1 - Embeddings)	8,192	No
Data source chunk size (Cohere Embed English)	512	No
Data source chunk size (Cohere Embed Multilingual)	512	No

Agent quotas

The following quotas apply to Agents for Amazon Bedrock.

Note

If a quota is marked as not adjustable through Service Quotas, you can submit a request through the limit increase form to be considered for an increase.

Quota	Maximum	Adjustable through Service Quotas (see note above table)	Description
Agents per account	50	No	The maximum number of Agents in one account.
Associated aliases per Agent	10	No	The maximum number of aliases that you can associate with an Agent.
Agent instruction count	1,200	No	The maximum number of characters in the instructions for an Agent.
Enabled action groups per Agent	5	No	The maximum number of action groups that you can add to an Agent.
Action groups per Agent	20	No	The maximum number of action groups, including disabled action groups, per agent
APIs per Agent	5	No	The maximum number of APIs that you can add to an Agent.
Lambda response payload size	25 KB	No	The maximum size of the payload in an

Quota	Maximum	Adjustable through Service Quotas (see note above table)	Description
			action group Lambda response.
Associated knowledge bases per Agent	2	No	The maximum number of knowledge bases that you can associate with an Agent.

Model customization quotas

The following quotas apply to model customization.

> **Note**
>
> If a quota is marked as not adjustable through Service Quotas, you can submit a request through the limit increase form to be considered for an increase.

Description	Maximum	Adjustable through Service Quotas (see note above table)
The maximum number of scheduled customization jobs.	2	No
The maximum number of custom models in an account.	100	Yes

To see hyperparameter quotas, see Custom model hyperparameters.

Select a tab to see model-specific quotas that apply to training and validation datasets used for customizing different foundation models.

> **Note**
>
> If a quota is marked as not adjustable through Service Quotas, you can submit a request through the limit increase form to be considered for an increase.

Amazon Titan Text G1 - Express

Description	Maximum (Continued Pre-training)	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Sum of input and output tokens when batch size is 1	4,096	4,096	No
Sum of input and output tokens when batch size is 2, 3, or 4	2,048	2,048	No
Character quota per sample in dataset	Token quota x 6	Token quota x 6	No
Sum of training and validation records	100,000	10,000	Yes
Training dataset file size	10 GB	1 GB	No
Validation dataset file size	100 MB	100 MB	No

Amazon Titan Text G1 - Lite

Description	Maximum (Continued Pre-training)	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Sum of input and output tokens when batch size is 1 or 2	4,096	4,096	No
Sum of input and output tokens when batch size is 3, 4, 5, or 6	2,048	2,048	No
Character quota per sample in dataset	Token quota x 6	Token quota x 6	No
Sum of training and validation records	100,000	10,000	Yes
Training dataset file size	10 GB	1 GB	No
Validation dataset file size	100 MB	100 MB	No

Amazon Titan Image Generator G1

Description	Minimum (Fine-tuning)	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Text prompt length in training sample, in characters	3	1,024	No

Description	Minimum (Fine-tuning)	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Records in a training dataset	5	10,000	No
Input image size	0	50 MB	No
Input image height in pixels	512	4,096	No
Input image width in pixels	512	4,096	No
Input image total pixels	0	12,582,912	No
Input image aspect ratio	1:4	4:1	No
Sum of training and validation records	N/A	10,000	Yes

Amazon Titan Multimodal Embeddings G1

Description	Minimum (Fine-tuning)	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Text prompt length in training sample, in characters	0	2,560	No
Records in a training dataset	1,000	500,000	No
Input image size	0	5 MB	No

Description	Minimum (Fine-tuning)	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Input image height in pixels	128	4096	No
Input image width in pixels	128	4096	No
Input image total pixels	0	12,528,912	No
Input image aspect ratio	1:4	4:1	No
Sum of training and validation records	N/A	50,000	Yes

Cohere Command

Description	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Input tokens	4,096	No
Output tokens	2,048	No
Character quota per sample in dataset	Token quota x 6	No
Records in a training dataset	10,000	No
Records in a validation dataset	1,000	No

Meta Llama 2

Description	Maximum (Fine-tuning)	Adjustable through Service Quotas (see note above table)
Input tokens	4,096	No
Output tokens	2,048	No
Character quota per sample in dataset	Token quota x 6	No
Sum of training and validation records	10,000	Yes

Provisioned Throughput quotas

The following quotas apply to Provisioned Throughput.

> **Note**
>
> If a quota is marked as not adjustable through Service Quotas, you can submit a request through the limit increase form to be considered for an increase.

Description	Default	Adjustable through Service Quotas (see note above table)
Model units that can be distributed across no-commitment Provisioned Throughputs	2	No
Model units that can be distributed across Provision	0	No

Description	Default	Adjustable through Service Quotas (see note above table)
ed Throughputs with commitment		

Model evaluation job quotas

The following quotas apply to model evaluation jobs,

Job type	Description	Default	Adjustable
Automated	The maximum number of datasets that you can specify in an automated model evaluation job. This includes both custom and built-in prompt datasets.	5	No
Automated	The maximum number of metrics that you can specify per dataset in an automated model evaluation job. This includes both custom and built-in metrics.	3	No
Human	The maximum number of custom metrics that you can specify in a model evaluation job that uses human workers.	10	No
Automated	The maximum number of models that you can specify in an automated model evaluation job.	1	No
Human	The maximum number of models that you can specify in a model evaluation job that uses human workers.	2	No
Automated	The maximum number of automatic model evaluation jobs that you can specify at one time in this account in the current Region.	20	No

Job type	Description	Default	Adjustable
Human	The maximum number of model evaluation jobs that use human workers you can specify at one time in this account in the current Region.	10	No
Both	The maximum number of model evaluation jobs that you can create in this account in the current Region.	500	No
Human	The maximum number of custom prompt datasets that you can specify in a human-based model evaluation job in this account in the current Region.	1	No
Both	The maximum number of prompts a custom prompt dataset can contains.	1,000	No
Both	The maximum size (in KB) of an individual prompt is a custom prompt dataset.	4 KB	No
Human	The maximum length (in days) of time that a worker can have to complete tasks.	30	No

API reference

The API Reference can be found here.

Document history for the Amazon Bedrock User Guide

- **Latest documentation update:** April 23rd, 2024

The following table describes important changes in each release of Amazon Bedrock. For notification about updates to this documentation, you can subscribe to an RSS feed.

Change	Description	Date
New feature	You can now import a custom model into Amazon Bedrock.	April 23, 2024
New feature	Agents for Amazon Bedrock can now define an action group by the parameters that it requires from the user.	April 23, 2024
New feature	You can now chat with your document with Amazon Bedrock.	April 23, 2024
New feature	You can now select from multiple data sources in knowledge bases for Amazon Bedrock.	April 23, 2024
New model	You can now use Anthropic Claude 3 Opus with Amazon Bedrock.	April 16, 2024
Region expansion	Amazon Bedrock is now available in Asia Pacific (Sydney) (ap-southeast-2). For information on endpoints, see Amazon Bedrock endpoints and quotas.	April 9, 2024

AWS CloudFormation support for Agents for Amazon Bedrock and Knowledge bases for Amazon Bedrock	You can now set up and manage your Agents for Amazon Bedrock and Knowledge bases for Amazon Bedrock resources with AWS CloudFormation.	April 5, 2024
Region expansion	Amazon Bedrock is now available in Europe (Paris) (eu-west-3). For information on endpoints, see Amazon Bedrock endpoints and quotas.	April 4, 2024
More model support for querying knowledge bases in Amazon Bedrock	You can now use Anthropic Claude 3 Haiku for knowledge base response generation.	April 4, 2024
New model	You can now use Mistral Large with Amazon Bedrock.	April 3, 2024
More model support for querying knowledge bases in Amazon Bedrock	You can now use Anthropic Claude 3 Haiku for knowledge base response generation.	April 3, 2024
New feature	You can now purchase Provisioned Throughput for base models with no commitment.	March 29, 2024
More model support for Provisioned Throughput	You can now purchase Provisioned Throughput for Anthropic Claude 3 Sonnet, Anthropic Claude 3 Haiku, Cohere Embed English, and Cohere Embed Multilingual.	March 29, 2024

New feature	You can now create a network access policy in Amazon OpenSearch Serverless to allow your Amazon Bedrock knowledge base to access a private OpenSearch Serverles s vector search collectio n configured with a VPC endpoint.	March 28, 2024
New feature	You can now include metadata for your source documents in Knowledge bases for Amazon Bedrock and filter on the metadata during knowledge base query.	March 27, 2024
New feature	You can now use a prompt template to customize the prompt sent to a model when you query a knowledge base and generate responses.	March 26, 2024
More model support for querying knowledge bases in Amazon Bedrock	You can now use Anthropic Claude 3 Sonnet for knowledge base response generation.	March 25, 2024
Decreased latency	You can now optimize on latency for simpler use cases in which agents have a single knowledge base.	March 20, 2024
New model	You can now use Anthropic Claude 3 Haiku with Amazon Bedrock.	March 13, 2024

New model	You can now use Anthropic Claude 3 Sonnet with Amazon Bedrock.	March 4, 2024
New model	You can now use Mistral AI models with Amazon Bedrock.	March 1, 2024
New feature	You can now customize the search strategy in Knowledge Base for Amazon OpenSearc h Serverless vector stores that contain a filterable text field.	February 28, 2024
New feature	You can now detect images with a watermark from Amazon Bedrock Titan Image Generator.	February 14, 2024
Updated AWS PrivateLink support	You can now use AWS PrivateLink to create interface VPC endpoints for the Agents for Amazon Bedrock Build-tim e service.	February 9, 2024
IAM role update	You can now use the same service role across knowledge bases and use roles without a predefined prefix.	February 9, 2024
Model in legacy status	Stable Diffusion XL v0.8 is now in legacy status. Migrate to Stable Diffusion XL v1.x before April 30, 2024.	February 2, 2024

Code examples chapter added	The Amazon Bedrock guide now includes code examples across a variety of Amazon Bedrock actions and scenarios .	January 25, 2024
New feature	Knowledge bases for Amazon Bedrock now offers you a choice between a production and non-production account when you choose to quick create an Amazon OpenSearc h Serverless vector store in the console.	January 24, 2024
New feature	Agents for Amazon Bedrock now lets you view traces in real-time when you use the test window in the console.	January 18, 2024
More model support for embedding data sources in Knowledge bases for Amazon Bedrock	Knowledge bases for Amazon Bedrock now supports using the Cohere Embed English and Cohere Embed Multiling ual to embed your data sources.	January 17, 2024
More model support for Agents for Amazon Bedrock and querying knowledge bases in Amazon Bedrock	Agents for Amazon Bedrock and Knowledge bases for Amazon Bedrock response generation now support Anthropic Claude 2.1.	December 27, 2023

Region expansion	Amazon Bedrock is now available in AWS GovCloud (US-West) (us-gov-west-1). For information on endpoints , see Amazon Bedrock endpoints and quotas.	December 21, 2023
New vector store support	You can now create a knowledge base in an Amazon Aurora database cluster. For more information, see Create a vector store in Amazon Aurora.	December 21, 2023
New managed policies	Amazon Bedrock has added `AmazonBedrockFullAccess` to give users permission to create, read, update, and delete resources, and `AmazonBedrockReadOnly` to give users read-only permissions for all actions.	December 12, 2023
New feature	Amazon Bedrock now supports creating model evaluation jobs using automatic metrics or human workers.	November 29, 2023
New feature	You can now monitor and customize your model versions.	November 29, 2023

New Titan models	New models from Titan include Amazon Titan Image Generator G1 and Amazon Titan Multimodal Embeddings G1. For more information, see Titan Models.	November 29, 2023
New feature	With Continued Pre-training you can teach a model new domain knowledge. For more information, see Custom Models.	November 28, 2023
New feature	You can now query knowledge bases through the Retrieve and RetrieveA ndGenerate APIs. For more information, see Query a knowledge base.	November 28, 2023
General release	General release of the Knowledge bases for Amazon Bedrock service. For more information, see Knowledge bases for Amazon Bedrock.	November 28, 2023
General release	General release of the Agents for Amazon Bedrock service. For more information, see Agents for Amazon Bedrock.	November 28, 2023
Customize more models	You can now customize models from Cohere and Meta. For more information, see Custom Models.	November 28, 2023

New model releases	Updated documentation to cover new Meta and Cohere models. For more informati on, see Amazon Bedrock.	November 13, 2023
Documentation localization	Amazon Bedrock documenta tion is now available in Japanese and German.	October 20, 2023
Region expansion	Amazon Bedrock is now available in Europe (Frankfur t) (eu-central-1). For informati on on endpoints, see Amazon Bedrock endpoints and quotas.	October 19, 2023
Region expansion	Amazon Bedrock is now available in Asia Pacific (Tokyo) (ap-northeast-1). For information on endpoints, see Amazon Bedrock endpoints and quotas.	October 3, 2023
Gated general release	Gated general release of the Amazon Bedrock service. For more information, see Amazon Bedrock.	September 28, 2023

AWS Glossary

For the latest AWS terminology, see the AWS glossary in the *AWS Glossary Reference*.