aws

Developer Guide

AWS SDK for Python

AWS SDK for Python: Developer Guide

Table of Contents

What is the AWS SDK for Python? 1
About this guide 1
Key features 1
Supported services 2
Get started with the SDK 2
Additional documentation and resources 2
Maintenance and support for SDK major versions 3
Choosing the right SDK 4
Overview 4
Key differences 4
Decision guide 5
Choose Boto3 if you: 5
Choose AWS SDK for Python (Developer Preview) if you: 6
Use both SDKs together if you: 6
Services available in the Developer Preview 6
What's coming next 7
Lifecycle and support 7
Getting started 8
Feedback 8
Getting started 9
Overview 9
Prerequisites and installation 10
Authenticating with AWS 12
Set environment variables 12
Use a profile in the shared credentials file 13
Additional authentication options 13
Example 1: Make asynchronous calls to Amazon DynamoDB 14
Before you begin 14
Write the code 14
Run the application 17
Success 17
Cleanup 17
Next steps 18
Example 2: Stream audio bidirectionally with Amazon Transcribe 18

Before you begin ... 18
Write the code ... 19
Run the application ... 22
Success ... 22
Cleanup ... 22
Next steps ... 22
Configuration ... 23
Prerequisites ... 23
Quick start ... 23
Explicit configuration ... 24
Next steps ... 24
Configuration resolution ... 24
Explicit resolution ... 25
Resolution order ... 26
Provenance tracking ... 28
Overriding values ... 29
Service-specific configuration ... 30
Configuration variables ... 30
aws_credentials_identity_resolver ... 31
endpoint_uri ... 31
interceptors ... 32
max_attempts ... 32
region ... 32
retry_mode ... 33
retry_strategy ... 34
sdk_ua_app_id ... 34
transport ... 35
user_agent_extra ... 35
Config files and profiles ... 35
Shared config files ... 35
Using named profiles ... 36
Credential providers ... 37
Default resolver chain ... 38
Configure a specific resolver ... 41
Configure static credentials ... 42
HTTP configuration ... 43

Provided HTTP clients ... 44
Override the HTTP client ... 44
Retries ... 46
Retryable errors ... 46
Standard retry strategy ... 47
Configuring retries ... 48
Using the SDK ... 51
Making requests and handling responses ... 51
Creating a service client ... 51
Constructing a typed request ... 52
Reading optional response members ... 53
Handling paginated responses ... 53
Following the modeled shape ... 54
Narrowing union members ... 55
Using the SDK asynchronously ... 55
Understanding coroutines and await ... 56
Running asynchronous code ... 56
Using asynchronous operations correctly ... 57
Running concurrent operations ... 58
Using service batch operations when available ... 58
Running a small number of operations concurrently ... 59
Bounding fan-out for a finite collection ... 60
Working with event streams ... 61
Consuming an output-only stream ... 61
Using a bidirectional event stream ... 63
Applying the pattern to other operations ... 66
Handling errors ... 66
Handling a modeled service error ... 67
Handling errors at an application boundary ... 67
Understanding retry behavior ... 68
Customizing SDK behavior with plugins and interceptors ... 68
Customizing configuration with plugins ... 69
Hooking into request processing with interceptors ... 70
Choosing lifecycle hooks carefully ... 72
Testing applications that use the SDK ... 72
Mocking an asynchronous operation ... 73

Testing through the generated client ... 74
Running application integration tests ... 75
Working with AWS services ... 77
DynamoDB ... 77
Set up the examples ... 77
Work with DynamoDB tables ... 78
Work with DynamoDB items ... 85
Query DynamoDB items ... 90
Use DynamoDB batch operations and transactions ... 93
SQS ... 98
Set up the examples ... 98
Work with Amazon SQS queues ... 99
Work with Amazon SQS messages ... 102
Configure long polling for Amazon SQS ... 107
SNS ... 109
Set up the examples ... 110
Create a topic ... 110
List topics ... 111
Subscribe an endpoint to a topic ... 112
Publish a message to a topic ... 113
Unsubscribe an endpoint from a topic ... 114
Delete a topic ... 114
More information ... 115
Bedrock Runtime ... 115
Set up the examples ... 115
Choose an inference operation ... 116
Use Converse ... 117
Use ConverseStream ... 120
Use InvokeModel ... 123
Use InvokeModelWithResponseStream ... 125
Use bidirectional streaming ... 129
Working with Boto3 ... 139
Key differences ... 139
Use both SDKs in one application ... 141
Configuration and credentials ... 142
Combining synchronous and asynchronous code ... 142

Before you begin ... 143
Scenario 1: Async-primary application with Boto3 for unsupported services ... 143
Scenario 2: Sync-primary application adding bidirectional streaming ... 148
Common pitfalls ... 154
Convert Boto3 operations to the AWS SDK for Python ... 155
Before you start ... 156
Try an operation ... 156
Example: convert a DynamoDB GetItem operation ... 157
Next steps ... 160
Security ... 162
Data protection ... 162
Transport Layer Security (TLS) ... 163
How TLS versions are selected ... 164
Check TLS version information ... 164
Enforce a minimum TLS version ... 165
AWS API endpoints and TLS 1.2 ... 165
Identity and Access Management ... 165
Audience ... 165
Authenticating with identities ... 166
Managing access using policies ... 167
How AWS services work with IAM ... 169
Troubleshooting AWS identity and access ... 169
Compliance Validation ... 171
Resilience ... 171
Infrastructure Security ... 172
Document history ... 173

What is the AWS SDK for Python?

AWS provides two SDKs for building Python applications on AWS:

- **AWS SDK for Python (Boto3)** — General Availability. The established, production-ready SDK with synchronous access to all AWS services. Use Boto3 for production workloads. Boto3 Developer Guide | SDK source on GitHub
- **AWS SDK for Python** — Developer Preview. A next-generation SDK with native asynchronous support, modular per-service packages, and bidirectional HTTP/2 streaming. Use this SDK to evaluate these capabilities and provide feedback in development and test environments only. SDK source on GitHub

You can install both SDKs in the same application. Separate Python namespaces allow them to coexist without conflicts.

> ⚠ **Developer Preview**
>
> The AWS SDK for Python is currently in Developer Preview. Use it to evaluate supported features and provide feedback in development and test environments. Do not use it for production workloads. APIs and behavior might change before general availability. Pin package versions and test your application when you upgrade. For guidance on choosing between the two SDKs, see Choosing the right AWS SDK for Python.

About this guide

This guide covers the AWS SDK for Python (Developer Preview). It provides native asynchronous service clients designed for `asyncio`, the standard Python library for asynchronous programming, including support for APIs that use bidirectional HTTP/2 streaming. Service clients are distributed as modular packages, so you can install only the clients that your application needs.

Key features

- **Native asynchronous APIs**: Service clients use Python `async` and `await` for non-blocking operations and concurrent I/O.

- **Bidirectional streaming**: Supported clients can send and receive event streams concurrently over HTTP/2.
- **Modular packages**: Each service client is available as a separate package, reducing the dependencies that you install and deploy.
- **Generated types**: Generated clients, request and response models, and type annotations provide editor completion and static-analysis support.
- **Configurable clients**: Configure Regions, endpoints, credential resolvers, retry behavior, plugins, and interceptors for your application.

Supported services

The Developer Preview supports a selected and growing set of AWS service clients. It does not yet provide clients for every AWS service or all of the features available in Boto3.

Each supported client is published as a service-specific package whose name begins with `aws-sdk-`. Before you adopt the SDK, verify that the service and operations that your application requires are available. For the current client packages and their release status, see the AWS SDK for Python repository on GitHub.

For guidance on supported AWS services, see Working with AWS services.

Get started with the SDK

To begin evaluating the AWS SDK for Python, use the following resources:

- Follow Getting started with the AWS SDK for Python to install the SDK, configure authentication, and run your first examples.
- See Configuration to configure service clients for your environment.
- Explore Using the AWS SDK for Python to make requests, use asynchronous operations, handle errors, and customize SDK behavior.

Additional documentation and resources

In addition to this guide, the following are valuable online resources for AWS SDK for Python developers:

- AWS SDKs and Tools Reference Guide: Contains settings, features, and other foundational concepts common to AWS SDKs.
- GitHub:
  - SDK source on GitHub
  - SDK issues on GitHub
- AWS SDK for Python API Reference
- Python developer blog
- The AWS Code Sample Catalog
- SDK License

Maintenance and support for SDK major versions

For information about maintenance and support for SDK major versions and their underlying dependencies, see the following in the AWS SDKs and Tools Reference Guide:

- AWS SDKs and tools maintenance policy
- AWS SDKs and tools version support matrix

Choosing the right AWS SDK for Python

AWS offers two SDKs for Python developers. This page helps you understand the differences and choose the right SDK for your needs.

Overview

Boto3 is the established, production-ready AWS SDK for Python with full coverage of all AWS services. It has been the standard for Python developers building on AWS for years and remains fully supported.

AWS SDK for Python (Developer Preview) is the next-generation SDK covered in this guide, rebuilt from the ground up with an async-first, modular architecture. It is currently in Developer Preview for evaluation and early feedback, and is not intended for production use.

Boto3 is generally available and recommended for production workloads. The AWS SDK for Python is in Developer Preview and intended for evaluation and testing only, it is not yet production-ready. Both SDKs coexist in the same project without conflicts, and you can use either or both based on your requirements.

Key differences

Capability	Boto3	AWS SDK for Python (Developer Preview)
Release status	General Availability (GA)	Developer Preview
Production use	Yes	Not recommended
Architecture	Synchronous, monolithic package	Async-first, modular per-service packages
Async support	Via community wrappers (aiobotocore/aioboto3)	Native async/await with asyncio
Packaging	Single `boto3` package for all services	Per-service packages (e.g., `aws-sdk-dynamodb`)

Capability	Boto3	AWS SDK for Python (Developer Preview)
HTTP/2 streaming	Not supported	Native bidirectional event streaming
Service coverage	All AWS services	24 services (expanding)
Paginators & waiters	Available	Coming soon
Presigned URLs	Available	Coming soon
S3 Transfer Manager	Available	Coming soon
Synchronous client	Default mode	Coming soon
Credential support	Full (SSO, Web Identity, etc.)	Default credential chain (SSO, Web Identity coming soon)
Code generation	Partial	Fully auto-generated from Smithy service models

Decision guide

Choose Boto3 if you:

- **Need production stability** — Boto3 is battle-tested, GA, and backed by complete AWS service coverage.
- **Require services not yet in the Developer Preview** — If your application depends on services beyond the current 24, Boto3 is your path.
- **Run synchronous workloads** — If your application doesn't have concurrency performance bottlenecks, Boto3's synchronous model is mature and well-suited.
- **Need full credential provider support** — If you rely on SSO via IAM Identity Center or Web Identity (EKS/GitHub Actions OIDC).
- **Depend on paginators, waiters, or presigned URLs** — These features are not yet available in the Developer Preview.

Choose AWS SDK for Python (Developer Preview) if you:

- **Are building high-throughput async applications** — Native async/await with asyncio eliminates the overhead of community wrappers and delivers true concurrency for multiple simultaneous AWS service calls.
- **Work with streaming services** — Bedrock Runtime, Transcribe Streaming, and SageMaker Runtime are designed for async, and the new SDK's HTTP/2 bidirectional streaming support provides first-class handling.
- **Need minimal deployment size** — Per-service packages (e.g., `aws-sdk-dynamodb`) with granular dependencies help you stay within AWS Lambda deployment size limits.
- **Want to influence SDK direction** — Developer Preview feedback directly shapes feature prioritization and the GA release.
- **Are evaluating for future migration** — Getting familiar with the new SDK's API patterns early helps you plan a smooth transition.

Use both SDKs together if you:

- Want to use the new SDK for async workloads (e.g., Bedrock streaming) while relying on Boto3 for everything else.
- Are incrementally evaluating the new SDK in non-production paths while maintaining Boto3 for production code.
- Need services from both SDKs in the same application.

> **Important**
>
> When using both SDKs together, be aware that mixing synchronous Boto3 calls inside an async event loop can block the loop and negate the performance benefits of async. Keep sync and async code paths separated. Refer to the Working with Boto3 page for additional details.

Services available in the Developer Preview

The Developer Preview currently supports 24 service clients, including:

- Amazon Bedrock & Bedrock Runtime
- Amazon DynamoDB
- AWS Lambda
- Amazon SQS
- Amazon SNS
- AWS STS
- Amazon Transcribe Streaming
- Amazon SageMaker Runtime
- And more — see the aws-sdk-python repository on GitHub for the full list

What's coming next

The AWS SDK for Python is being developed incrementally. Planned additions include:

- **Expanded service coverage** — Support for Amazon S3 and additional AWS services.
- **Complete credential chain** — SSO via IAM Identity Center, Web Identity, AWS Login, process credentials, and SigV4a multi-region signing.
- **Core SDK features** — Paginators, waiters, presigned URLs, S3 Transfer Manager, and synchronous client support.
- **Observability** — Adaptive retries, logging, metrics, OpenTelemetry tracing, proxy configuration, and connection pooling.

Lifecycle and support

Question	Answer
Is Boto3 being retired?	Not until the new SDK reaches feature parity with Boto3's synchronous support. AWS will provide ample notice, migration guidance, and support throughout any transition.
When will the new SDK reach GA?	The GA release will be informed by Developer Preview feedback. No date has been set.

Question	Answer
Can I use the Developer Preview in productio n?	No. It is intended for evaluation and testing in pre-production environments only.
Will both SDKs be supported simultane ously?	Yes. Both SDKs are fully supported, and you can confidently choose the one that best meets your current needs.

Getting started

- **Boto3**: Developer Guide | SDK source on GitHub
- **AWS SDK for Python (Developer Preview)**: Getting started with the AWS SDK for Python | SDK source on GitHub

Feedback

We want your feedback to guide feature prioritization and service coverage. File issues in the aws-sdk-python repository on GitHub, or reach out through AWS Support.

Getting started with the AWS SDK for Python

The topics in this chapter walk you through the essential steps to begin building asynchronous Python applications that connect to AWS services: installing the SDK, configuring credentials, and running two working examples.

Topics

- Overview
- Prerequisites and installation
- Authenticating with AWS using the AWS SDK for Python
- Example 1: Make asynchronous calls to Amazon DynamoDB
- Example 2: Stream audio bidirectionally with Amazon Transcribe

Overview

This chapter takes you from a new Python environment to two working asynchronous AWS calls. The first example creates a temporary Amazon DynamoDB table, writes and reads an item, and deletes the table. The second sends prerecorded audio to Amazon Transcribe and prints completed transcript segments as they arrive.

Both examples use the same basic workflow:

1. Install only the service client packages that your application needs.
2. Configure how the SDK authenticates with AWS.
3. Create an asynchronous client for the service.
4. Build typed input objects and await the client operation.

> **Note**
>
> The AWS SDK for Python is in Developer Preview. Use it for evaluation in development and test environments, not in production. APIs can change between preview releases, so pin and test the package versions that your application uses.

Prerequisites and installation

To start building with the AWS SDK for Python, you need an AWS account, a supported Python version, and the SDK's service client packages. The following steps help you verify your environment, create an isolated workspace, and install only the packages that your application requires.

Before you begin:

- Create an AWS account if you don't have one. For instructions, see Sign up for AWS in the *AWS Account Management User Guide*.
- Install Python 3.12 or later. The installation commands in this guide use **pip**, the package installer included with most Python installations. To install Python, see Download Python.

Confirm that Python and **pip** are available:

```
python --version
python -m pip --version
```

Create a virtual environment to keep this project's packages separate from other Python projects:

```
python -m venv .venv
```

Activate the virtual environment on macOS or Linux:

```
source .venv/bin/activate
```

On Windows, activate it with the following command instead:

```
.venv\Scripts\activate
```

Then update **pip**:

```
python -m pip install --upgrade pip
```

The preceding steps are the standard setup for most SDK projects. The packages that you install next are specific to the examples in this chapter.

The SDK publishes a separate package for each AWS service client. Install the Amazon DynamoDB and Amazon Transcribe client packages:

```
python -m pip install aws-sdk-dynamodb aws-sdk-transcribe-streaming
```

This command installs both clients as individual packages.

Alternatively, install the same clients as optional dependencies of the `aws-sdk-python` meta-package:

```
python -m pip install "aws-sdk-python[dynamodb,transcribe_streaming]"
```

The meta-package installs only the service clients that you select as extras; without extras, it installs no clients. It coordinates client versions through its MAJOR.MINOR version, so it installs compatible client versions together.

Use the meta-package when your application depends on several clients. Install individual client packages when your application uses only one or two services and you want the smallest dependency set.

The preceding commands install the latest package versions. To control which version is installed, add a version specifier to the package name. Version specifiers work for both individual client packages and the meta-package:

```
# Install version 0.11.0 specifically
python -m pip install "aws-sdk-dynamodb==0.11.0" "aws-sdk-transcribe-streaming==0.11.0"
  # individual client packages
python -m pip install "aws-sdk-python[dynamodb,transcribe_streaming]==0.11.0"
  # meta-package

# Install the latest version that is at least 0.11.0
python -m pip install "aws-sdk-dynamodb>=0.11.0" "aws-sdk-transcribe-streaming>=0.11.0"
  # individual client packages
python -m pip install "aws-sdk-python[dynamodb,transcribe_streaming]>=0.11.0"
  # meta-package

# Install the latest version that is at most 0.11.0
python -m pip install "aws-sdk-dynamodb<=0.11.0" "aws-sdk-transcribe-streaming<=0.11.0"
  # individual client packages
python -m pip install "aws-sdk-python[dynamodb,transcribe_streaming]<=0.11.0"
  # meta-package
```

```
# Install the latest version that is compatible with 0.11.0 (>=0.11.0, <0.12.0)
python -m pip install "aws-sdk-dynamodb~=0.11.0" "aws-sdk-transcribe-streaming~=0.11.0"
  # individual client packages
python -m pip install "aws-sdk-python[dynamodb,transcribe_streaming]~=0.11.0"
  # meta-package
```

Authenticating with AWS using the AWS SDK for Python

You must establish how the AWS SDK for Python authenticates with AWS when you develop with AWS services. The SDK manages credential discovery, signature creation, and credential refreshing completely behind the scenes, letting you focus on your application logic.

> **Important**
>
> During Developer Preview, the AWS SDK for Python supports fewer authentication sources than other AWS SDKs. For example, it can't use IAM Identity Center credentials or AWS Management Console sign-in credentials. For the sources this release supports, see Credential providers.

For local development, we recommend short-term credentials. Obtain them with the AWS CLI or from your identity provider, and then supply them to the SDK by using one of the following two methods.

Set environment variables

Set your credentials as environment variables on macOS or Linux:

```
export AWS_ACCESS_KEY_ID=your_access_key_id
export AWS_SECRET_ACCESS_KEY=your_secret_access_key
export AWS_SESSION_TOKEN=your_session_token
```

On Windows, use the following commands instead:

```
set AWS_ACCESS_KEY_ID=your_access_key_id
set AWS_SECRET_ACCESS_KEY=your_secret_access_key
set AWS_SESSION_TOKEN=your_session_token
```

Omit `AWS_SESSION_TOKEN` if you use long-term credentials. Short-term credentials expire after a limited period, and requests then fail. In that case, obtain new credentials and set the variables again.

Use a profile in the shared credentials file

Instead of setting environment variables in each terminal session, you can store your credentials in a named profile. The `credentials` file is shared by AWS SDKs and tools such as the AWS CLI, so it might already exist on your system. For its location, see Location of the shared files in the *AWS SDKs and Tools Reference Guide*.

Add a `[default]` profile to the `credentials` file:

```
[default]
aws_access_key_id = your_access_key_id
aws_secret_access_key = your_secret_access_key
aws_session_token = your_session_token
```

The SDK uses the `[default]` profile unless you select another one. To select a named profile, set the `AWS_PROFILE` environment variable:

```
export AWS_PROFILE=my-profile
```

Additional authentication options

The preceding sections cover local development. Code that runs within an AWS environment or assumes an IAM role uses other credential sources. For more options on authentication for the SDK, see the following:

- For every credential source the SDK supports, and how to select one in your code, see Credential providers.
- To create short-term credentials, see Temporary Security Credentials in the *IAM User Guide*.
- For best practices for protecting your AWS account and credentials, see Security best practices in IAM in the *IAM User Guide*.

Example 1: Make asynchronous calls to Amazon DynamoDB

This example uses `AsyncDynamoDBClient` to create a temporary Amazon DynamoDB table, wait until it is active, write an item, and read the item with a strongly consistent read. The `finally` block deletes the table if a later operation fails.

Before you begin

Complete Prerequisites and installation and Authenticating with AWS using the AWS SDK for Python. This example also requires the following:

- The identity selected by your authentication method must allow `dynamodb:CreateTable`, `dynamodb:DescribeTable`, `dynamodb:PutItem`, `dynamodb:GetItem`, and `dynamodb:DeleteTable`. To learn how IAM policies grant permissions like these, see Policies and permissions in IAM in the *IAM User Guide*.

> **Warning**
>
> Creating and using an Amazon DynamoDB table can incur charges. The example deletes the table when it finishes; confirm the deletion in its output.

Write the code

Create a file named `getting_started_dynamodb.py` with the following code:

```
"""Make first asynchronous DynamoDB calls with the AWS SDK for Python."""

import asyncio
import uuid

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig
from aws_sdk_dynamodb.models import (
    AttributeDefinition,
    AttributeValueN,
    AttributeValueS,
    CreateTableInput,
    DeleteTableInput,
    DescribeTableInput,
```

```
    GetItemInput,
    KeySchemaElement,
    KeyType,
    ProvisionedThroughput,
    PutItemInput,
    ResourceNotFoundException,
    ScalarAttributeType,
)

TABLE_NAME = f"python-getting-started-{uuid.uuid4().hex[:8]}"


async def main() -> None:
    config = await AsyncDynamoDBConfig.resolve(region="us-east-1")
    async with AsyncDynamoDBClient(config=config) as client:
        print(f"Creating table '{TABLE_NAME}'...")
        await client.create_table(
            input=CreateTableInput(
                table_name=TABLE_NAME,
                attribute_definitions=[
                    AttributeDefinition(
                        attribute_name="pk",
                        attribute_type=ScalarAttributeType.S,
                    )
                ],
                key_schema=[
                    KeySchemaElement(
                        attribute_name="pk", key_type=KeyType.HASH
                    )
                ],
                provisioned_throughput=ProvisionedThroughput(
                    read_capacity_units=5,
                    write_capacity_units=5,
                ),
            )
        )

        try:
            print("Waiting for the table to become active...")
            for _ in range(30):
                response = await client.describe_table(
                    input=DescribeTableInput(table_name=TABLE_NAME)
                )
                if response.table and response.table.table_status == "ACTIVE":
```

```
                break
            await asyncio.sleep(2)
        else:
            raise TimeoutError(
                f"Table '{TABLE_NAME}' did not become active in time"
            )

        await client.put_item(
            input=PutItemInput(
                table_name=TABLE_NAME,
                item={
                    "pk": AttributeValueS(value="user-001"),
                    "name": AttributeValueS(value="Ada"),
                    "visits": AttributeValueN(value="1"),
                },
            )
        )

        response = await client.get_item(
            input=GetItemInput(
                table_name=TABLE_NAME,
                key={"pk": AttributeValueS(value="user-001")},
                consistent_read=True,
            )
        )
        if response.item is None:
            raise RuntimeError("DynamoDB returned no item")

        name = response.item.get("name")
        visits = response.item.get("visits")
        if not isinstance(name, AttributeValueS) or not isinstance(
            visits, AttributeValueN
        ):
            raise RuntimeError("DynamoDB returned unexpected attribute types")

        print(f"Read item: name={name.value}, visits={visits.value}")
    finally:
        print(f"Deleting table '{TABLE_NAME}'...")
        try:
            await client.delete_table(
                input=DeleteTableInput(table_name=TABLE_NAME)
            )
        except ResourceNotFoundException:
            pass
```

```
            else:
                for _ in range(30):
                    try:
                        await client.describe_table(
                            input=DescribeTableInput(table_name=TABLE_NAME)
                        )
                    except ResourceNotFoundException:
                        break
                    await asyncio.sleep(2)
                else:
                    raise TimeoutError(
                        f"Table '{TABLE_NAME}' was not deleted"
                    )
            print("Table deleted.")


if __name__ == "__main__":
    asyncio.run(main())
```

DynamoDB creates and deletes tables asynchronously. The bounded polling loops wait for the table to become `ACTIVE` before writing and confirm that deletion completes before reporting success. A future release of the SDK will add waiters that replace this polling logic. Item attributes use generated union variants such as `AttributeValueS` and `AttributeValueN`; numbers are represented as strings to preserve precision.

Run the application

Run the program from the directory that contains `getting_started_dynamodb.py`:

```
python getting_started_dynamodb.py
```

Success

A successful run prints the generated table name, `Read item: name=Ada, visits=1`, and `Table deleted.`

Cleanup

The `finally` block requests and confirms table deletion even if writing or reading fails. If the program doesn't print `Table deleted.`, open the DynamoDB console in `us-east-1` and delete the table whose name starts with `python-getting-started-`.

Next steps

- For more Amazon DynamoDB operations and examples, see Amazon DynamoDB.
- For generated client operations and model types, see the Amazon DynamoDB API reference.
- For more information about typed inputs and outputs, see Making requests and handling responses.

Example 2: Stream audio bidirectionally with Amazon Transcribe

This example uses `AsyncTranscribeStreamingClient` to send prerecorded audio to Amazon Transcribe while receiving transcription results over the same HTTP/2 bidirectional event stream. It uses `asyncio.gather()` to publish audio and receive completed transcript segments concurrently.

Before you begin

Complete Prerequisites and installation and Authenticating with AWS using the AWS SDK for Python. This example also requires the following:

- The identity selected by your authentication method must allow `transcribe:StartStreamTranscription`. To learn how IAM policies grant permissions like these, see Policies and permissions in IAM in the *IAM User Guide*.
- Use the SDK repository's sample `test.wav` file. The sample contains 16 kHz, 16-bit, mono PCM audio. The example validates the WAV format and sends only its audio frames, without the WAV container header.

Bidirectional streaming requires the AWS Common Runtime (CRT) HTTP client, which the example selects as the client's transport. Opt in by installing the client's `awscrt` extra. For more information, see Use the AWS CRT client for bidirectional streaming.

```
python -m pip install "aws-sdk-transcribe-streaming[awscrt]"
```

Download the linked file in your browser and save it as `test.wav` in the directory where you will create the example. Alternatively, download it with Python:

```
python -c "from urllib.request import urlretrieve; urlretrieve('https://github.com/aws/
aws-sdk-python/raw/refs/heads/develop/clients/aws-sdk-transcribe-streaming/examples/
test.wav', 'test.wav')"
```

> **Warning**
>
> Streaming audio with Amazon Transcribe can incur charges. Review Amazon Transcribe pricing before repeated use.

Write the code

Create a file named `getting_started_transcribe.py` with the following code:

```
"""Transcribe a prerecorded audio stream with the AWS SDK for Python."""

import asyncio
import wave
from pathlib import Path

from smithy_http.aio.crt import AWSCRTHTTPClient

from aws_sdk_transcribe_streaming.client import AsyncTranscribeStreamingClient
from aws_sdk_transcribe_streaming.config import AsyncTranscribeStreamingConfig
from aws_sdk_transcribe_streaming.models import (
    AudioEvent,
    AudioStreamAudioEvent,
    LanguageCode,
    MediaEncoding,
    StartStreamTranscriptionInput,
    TranscriptResultStreamTranscriptEvent,
)

AUDIO_FILE = Path("test.wav")
SAMPLE_RATE = 16_000
BYTES_PER_SAMPLE = 2
CHANNELS = 1
CHUNK_FRAMES = 1_600  # 100 ms of audio


async def send_audio(stream) -> None:
    loop = asyncio.get_running_loop()
```

```
    started = loop.time()
    audio_seconds = 0.0
    chunks_sent = 0

    try:
        with wave.open(str(AUDIO_FILE), "rb") as source:
            audio_format = (
                source.getnchannels(),
                source.getsampwidth(),
                source.getframerate(),
                source.getcomptype(),
            )
            expected_format = (
                CHANNELS,
                BYTES_PER_SAMPLE,
                SAMPLE_RATE,
                "NONE",
            )
            if audio_format != expected_format:
                raise ValueError(
                    "test.wav must be uncompressed 16 kHz, "
                    "16-bit, mono PCM audio"
                )

            while chunk := await asyncio.to_thread(
                source.readframes, CHUNK_FRAMES
            ):
                await stream.input_stream.send(
                    AudioStreamAudioEvent(
                        value=AudioEvent(audio_chunk=chunk)
                    )
                )
                chunks_sent += 1
                audio_seconds += len(chunk) / (
                    BYTES_PER_SAMPLE * SAMPLE_RATE * CHANNELS
                )
                delay = started + audio_seconds - loop.time()
                if delay > 0:
                    await asyncio.sleep(delay)

        if chunks_sent == 0:
            raise RuntimeError(f"No audio read from {AUDIO_FILE}")
    finally:
        await stream.input_stream.close()
```

```
async def print_transcripts(stream) -> None:
    _, output_stream = await stream.await_output()
    if output_stream is None:
        raise RuntimeError("The service returned no output stream")

    async for event in output_stream:
        if not isinstance(
            event, TranscriptResultStreamTranscriptEvent
        ):
            raise RuntimeError(
                f"Unexpected stream event: {type(event).__name__}"
            )

        transcript = event.value.transcript
        if transcript is None:
            continue
        for result in transcript.results or []:
            if result.is_partial:
                continue
            alternatives = result.alternatives or []
            if alternatives and alternatives[0].transcript:
                print(alternatives[0].transcript)


async def main() -> None:
    # Bidirectional streaming requires the AWS CRT HTTP client.
    config = await AsyncTranscribeStreamingConfig.resolve(
        region="us-east-1",
        transport=AWSCRTHTTPClient(),
    )
    async with AsyncTranscribeStreamingClient(config=config) as client:
        stream = await client.start_stream_transcription(
            input=StartStreamTranscriptionInput(
                language_code=LanguageCode.EN_US,
                media_sample_rate_hertz=SAMPLE_RATE,
                media_encoding=MediaEncoding.PCM,
            )
        )

        async with stream:
            await asyncio.gather(
                send_audio(stream),
```

```
                print_transcripts(stream),
            )


if __name__ == "__main__":
    asyncio.run(main())
```

Amazon Transcribe returns partial and final results while audio is streamed. The example validates and paces PCM audio in `send_audio()`, wraps each chunk in the generated input event type, receives final results concurrently in `print_transcripts()`, and closes the input stream in a `finally` block. The client resolves credentials from the default credential chain and sets only the AWS Region explicitly.

Run the application

Run the program from the directory that contains `getting_started_transcribe.py` and `test.wav`:

```
python getting_started_transcribe.py
```

Success

A successful run prints one or more completed transcript segments and then exits.

Cleanup

This example creates no persistent AWS resources. It closes the input stream after sending the file, and the `async with` block closes the bidirectional stream when the application finishes.

Next steps

- For service concepts, audio requirements, and streaming best practices, see Transcribing streaming audio in the *Amazon Transcribe Developer Guide*.
- For generated request and response types, see the `start_stream_transcription()` API reference.
- For more information about publishing and consuming event streams, see Working with event streams.
- For complete SDK repository examples, see the Amazon Transcribe streaming examples. The directory includes one example for a prerecorded file and another for live microphone input.

Configuration

The AWS SDK for Python uses a configuration system that automatically discovers and resolves your AWS settings from multiple sources. When you create a client (`client = AsyncBedrockRuntimeClient()`), the SDK automatically resolves configuration values from your environment variables and AWS config files the first time an operation is called.

Prerequisites

Before using the SDK, make sure you have the following configured:

- An AWS Region, either set via the `AWS_REGION` environment variable, in your `~/.aws/config` file, or passed directly in code. Region is required and has no default.
- AWS credentials available through a supported credential source (see Credential providers).

Quick start

The following example shows the simplest way to create a client and make a request. You do not need explicit configuration. The SDK resolves all configuration values automatically from your environment when the first operation is called:

```
import asyncio
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.models import Message, ContentBlockText, ConverseInput


async def main():

    async with AsyncBedrockRuntimeClient() as client:

        response = await client.converse(
            ConverseInput(
                model_id="global.amazon.nova-2-lite-v1:0",
                messages=[Message(role="user", content=[ContentBlockText(value="What is
 2 + 2?")])],
            )
        )
```

```
        print(response.output.value.content[0].value)


asyncio.run(main())
```

This example assumes that `AWS_REGION` is set (or `region` is defined in your config file) and that credentials are available through the credential chain. When you run this, the SDK creates a client, resolves configuration on the first operation call, sends a request to Amazon Bedrock Runtime, and prints the response.

Explicit configuration

You can either let the SDK resolve configuration automatically (as shown in Quick start), or explicitly resolve it for more control. Explicit resolution lets you override specific values and validate configuration upfront before any operation is called. See Configuration resolution for details.

Next steps

- To understand how the SDK resolves configuration values, see Configuration resolution.
- To look up a specific configuration setting, see Configuration variables.
- To set up profiles for different environments, see Config files and named profiles.
- To learn how the SDK finds credentials, see Credential providers.
- To configure or replace the HTTP client, see HTTP configuration.
- To understand how retries work and how to configure them, see Retries.

Configuration resolution

This page covers how the AWS SDK for Python resolves configuration values. It includes the following topics:

- Explicit resolution shows how to resolve configuration upfront and pass it to a client.
- Resolution order describes the precedence chain the SDK uses to determine where each value comes from.
- Provenance tracking explains how to inspect the source of any resolved value.

- Overriding values shows how to set values at resolution time or mutate them after resolution.
- Service-specific configuration covers endpoint resolution for individual services.

The config resolution system guarantees the following:

- **Async-first**. All resolution happens asynchronously, either explicitly via `resolve()` or automatically on the first operation call.
- **Transparent sourcing**. Every resolved value tracks where it came from (environment, config file, override, or default), queryable at any time via `source_of()`.
- **Early validation**. Invalid values are rejected immediately during resolution, not when the first API call is made. If your configuration is invalid, you will know it right away.

Explicit resolution

If you want to resolve configuration upfront or customize values before creating a client, you can explicitly call `resolve()` and pass the resulting config object to the client. Config objects cannot be created directly. Calling `AsyncBedrockRuntimeConfig()` raises an error, and all construction must go through `resolve()`:

```
# This raises ConfigError
config = AsyncBedrockRuntimeConfig()

# This is the correct way
config = await AsyncBedrockRuntimeConfig.resolve()
```

You can also pass values directly to `resolve()` to override what would otherwise be resolved from environment variables or config files:

```
config = await AsyncBedrockRuntimeConfig.resolve(
    region="us-west-2",
    max_attempts=5,
)
```

The following example shows how to explicitly resolve configuration with an override and pass it to a client:

```
import asyncio
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
```

```
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.models import Message, ContentBlockText, ConverseInput
from smithy_aws_core.identity import EnvironmentCredentialsResolver


async def main():
    config = await AsyncBedrockRuntimeConfig.resolve(
        region="us-west-2",
        aws_credentials_identity_resolver=EnvironmentCredentialsResolver()
    )
    async with AsyncBedrockRuntimeClient(config=config) as client:

        response = await client.converse(
            ConverseInput(
                model_id="global.amazon.nova-2-lite-v1:0",
                messages=[Message(role="user", content=[ContentBlockText(value="What is
 2 + 2?")])],
            )
        )

        print(response.output.value.content[0].value)


asyncio.run(main())
```

Resolution order

Each config field is resolved using the following precedence. The first source that provides a value wins.

1. Explicit override (passed to `resolve()`)
2. Environment variable
3. Config file profile
4. Default

1. Explicit override

This is the highest-priority source. Values passed directly to `resolve()` always take precedence over all other sources. Use this when you want to hardcode a value regardless of what's in the environment or config files:

```
config = await AsyncBedrockRuntimeConfig.resolve(
    region="us-west-2",
    max_attempts=10,
)
```

2. Environment variable

If no explicit override is provided, the SDK checks for standard AWS environment variables.

```
export AWS_REGION=us-west-2
export AWS_RETRY_MODE=standard
export AWS_MAX_ATTEMPTS=5
export AWS_ENDPOINT_URL=https://custom.endpoint.com
```

```
config = await AsyncBedrockRuntimeConfig.resolve()
config.region   # "us-west-2"
```

3. Config file (profile)

If no override or environment variable is found, the SDK reads from the shared AWS configuration files. Values are read from `~/.aws/config` and `~/.aws/credentials`. The SDK reads the active profile (defaulting to `[default]` unless `AWS_PROFILE` is set or `profile=` is passed):

```
# ~/.aws/config
[default]
region = us-east-1
retry_mode = standard
max_attempts = 3
endpoint_url = https://my-endpoint.com
sdk_ua_app_id = my-application
```

```
config = await AsyncBedrockRuntimeConfig.resolve()
config.region   # "us-east-1"
```

4. Default

If no other source provides a value, the SDK falls back to built-in defaults. These are the lowest-priority source and vary per field:

```
# Empty environment, no config file
```

```
config = await AsyncBedrockRuntimeConfig.resolve(region="us-east-1")
config.retry_mode  # "standard"
```

> **Note**
>
> `region` has no default. Resolution fails with `ConfigValidationError` if no source provides it.

Provenance tracking

The SDK tracks the origin of every resolved configuration value. You can call `source_of()` on any field to find out whether its value came from an explicit override, an environment variable, a config file profile, or a built-in default. This is useful for debugging configuration issues or understanding which source is providing a given value.

```
# With AWS_REGION=us-west-2 set in the environment
# and ~/.aws/config containing:
#   [default]
#   retry_mode = standard

config = await AsyncBedrockRuntimeConfig.resolve()

config.source_of("region")       # env
config.source_of("retry_mode")   # profile
config.source_of("max_attempts") # default
```

Available sources:

Source	Meaning
`override`	Value was explicitly provided to `resolve()` or set after resolution
`env`	Value came from an environment variable
`profile`	Value came from the config/credentials file
`default`	No source provided the value; using built-in default

This is useful for debugging ("where did this region come from?") and for libraries or plugins that need to know whether a user explicitly configured a value or it was auto-resolved.

Overriding values

At resolution time

Pass keyword arguments to `resolve()` to set values that skip the resolution chain entirely:

```
config = await AsyncBedrockRuntimeConfig.resolve(
    region="eu-west-1",
    max_attempts=5,
    retry_mode="standard",
)
```

These overrides are validated immediately. Passing an invalid value raises `ConfigValidationError`:

```
config = await AsyncBedrockRuntimeConfig.resolve(region="bad!")
# raises ConfigValidationError: Invalid value for 'region': 'bad!'.
# Must be a valid AWS region identifier.
```

After resolution

Fields can be mutated after resolution. The source updates to `override` and validation runs on the new value:

```
config = await AsyncBedrockRuntimeConfig.resolve(region='us-east-1')
config.region = "us-east-2"
config.source_of("region")  # override
config.region = "bad!"      # raises ConfigValidationError
```

> **Note**
>
> Static credential fields (`aws_access_key_id`, `aws_secret_access_key`, and `aws_session_token`) cannot be modified after resolution.

Service-specific configuration

Service-specific endpoint resolution

When resolving `endpoint_uri`, service-specific configs (like `AsyncBedrockRuntimeConfig`) use an extended precedence chain. With this extended precedence chain, you can set a custom endpoint for one service without affecting other services. The SDK checks the following sources in order, and the first one that provides a value wins:

1. `AWS_ENDPOINT_URL_BEDROCK_RUNTIME` (service-specific env var)
2. `AWS_ENDPOINT_URL` (global env var)
3. Services section in config file (service-specific config)
4. `endpoint_url` in profile (global config)

Example `~/.aws/config` with a services section:

```
[default]
region = us-east-1
services = my-services

[services my-services]
bedrock_runtime =
  endpoint_url = https://bedrock-runtime.us-east-1.amazonaws.com
s3 =
  endpoint_url = https://s3.custom.local
```

```
config = await AsyncBedrockRuntimeConfig.resolve()
config.endpoint_uri   # "https://bedrock-runtime.us-east-1.amazonaws.com" (resolved
 from the services section)
```

Configuration variables

This page lists all configuration variables supported by the AWS SDK for Python, including their default values and usage examples where applicable.

aws_credentials_identity_resolver

An `IdentityResolver` that resolves AWS credentials for request authentication. Set this field only to use a specific resolver instead of the default credential resolver chain. For more information, see Credential providers.

Default	None (the client uses the default credential resolver chain)
Valid values	`IdentityResolver[AWSCredentialsIdentity, AWSIdentityProperties]`

To provide static credentials directly, see Configure static credentials.

endpoint_uri

A custom endpoint URL to use instead of the standard AWS regional endpoint.

Env var	`AWS_ENDPOINT_URL` (global), `AWS_ENDPOINT_URL_<SERVICE_ID>` (service-specific)
Profile key	`endpoint_url`
Default	None (uses standard regional endpoint)

Service-specific configs resolve this with extended precedence. See Service-specific endpoint resolution.

```
# Environment variable — global (applies to all services)
export AWS_ENDPOINT_URL=http://localhost:4566

# Environment variable — service-specific
export AWS_ENDPOINT_URL_BEDROCK_RUNTIME=http://localhost:5000
```

```
# Config file
[default]
endpoint_url = http://localhost:4566
```

interceptors

A list of interceptor hooks that are called at various points during request execution. Interceptors can inspect or modify requests and responses.

Default	`[]` (empty list)

max_attempts

An integer representing the maximum number of attempts made for a single request, including the initial attempt. For example, setting this value to 5 results in a request being retried up to 4 times. If not provided, the number of retries defaults to whatever is modeled, which is typically 3 in the `standard` retry mode.

Env var	`AWS_MAX_ATTEMPTS`
Profile key	`max_attempts`
Default	None (the retry strategy uses its own default, typically 3 for standard mode)
Valid values	Positive integer (>= 1)

```
# Environment variable
export AWS_MAX_ATTEMPTS=5
```

```
# Config file
[default]
max_attempts = 5
```

region

The AWS Region to send requests to.

Env vars	`AWS_REGION` , `AWS_DEFAULT_REGION`

Profile key	`region`
Default	None
Valid values	AWS Region identifiers like `us-east-1` , `eu-west-1` , `ap-southeast-1`

```
# Environment variable
export AWS_REGION=us-west-2
```

```
# Config file
[default]
region = us-west-2
```

```
# Explicit override
config = await AsyncBedrockRuntimeConfig.resolve(region="us-west-2")
```

retry_mode

A string that represents the type of retries the SDK performs.

Env var	`AWS_RETRY_MODE`
Profile key	`retry_mode`
Default	`"standard"`
Valid values	`"standard"` . Values `"legacy"` and `"adaptive"` from environment variables or config files are accepted but mapped to `"standard"` with a warning.

- **`standard`**: a standardized set of retry rules across the AWS SDKs. This includes a standard set of errors that are retried and support for retry quotas, which limit the number of unsuccessful retries the SDK can make. This mode will default the maximum number of attempts to 3 unless a `max_attempts` is explicitly provided.
- **`legacy`**: not supported in this SDK. If set, it maps to `"standard"` with a warning.

- **`adaptive`**: not supported at the moment. If set, it maps to `"standard"` with a warning.

```
# Environment variable
export AWS_RETRY_MODE=standard
```

```
# Config file
[default]
retry_mode = standard
```

retry_strategy

An explicit retry strategy instance that bypasses `retry_mode` and `max_attempts` entirely. Use this when you need full control over retry behavior.

Default	None (uses `retry_mode` /`max_attempts` to construct a strategy)

sdk_ua_app_id

An optional application-specific identifier that can be set. When set, it is appended to the SDK's User-Agent header, allowing you to identify your application in AWS service logs.

Env var	`AWS_SDK_UA_APP_ID`
Profile key	`sdk_ua_app_id`
Default	None
Valid values	Any string

```
# Environment variable
export AWS_SDK_UA_APP_ID=my-web-app
```

```
# Config file
[default]
sdk_ua_app_id = my-web-app
```

transport

The HTTP client that the SDK uses to send requests. For details, including the optional `AWSCRTHTTPClient` and custom HTTP clients, see the HTTP configuration page.

Default	`AIOHTTPClient`

user_agent_extra

An additional string appended to the User-Agent header.

Default	None

Config files and named profiles

This page covers how the SDK reads shared AWS configuration files and how to use named profiles to manage different environments. It includes the following topics:

- Shared config files explains which files the SDK reads and their format.
- Using named profiles shows how to switch between different configurations without changing your code.

Shared config files

The SDK searches the shared AWS configuration files when looking for configuration values. These are the same files used by the AWS CLI and other AWS SDKs, so the SDK automatically reads any settings you configured there. The SDK reads from two files:

File	Default location	Env override
Config	`~/.aws/config`	`AWS_CONFIG_FILE`
Credentials	`~/.aws/credentials`	`AWS_SHARED_CREDENTIALS_FILE`

Both files use INI-style format. The config file requires the `[profile ...]` prefix (except for `[default]`); the credentials file uses bare section names. You can create multiple profiles (logical groups of configuration) by creating sections named `[profile profile-name]`.

```
# ~/.aws/config
[default]
region = us-west-2
retry_mode = standard
max_attempts = 3

[profile production]
region = us-east-1
retry_mode = standard
max_attempts = 10
```

```
# ~/.aws/credentials
[default]
aws_access_key_id = AKIAIOSFODNN7EXAMPLE
aws_secret_access_key = wJalrXUtnFEMI/K7MDENG/bPxRfiCYEXAMPLEKEY
```

Using named profiles

A named profile is a collection of settings stored in the shared AWS config file under a specific name. With profiles, you can maintain separate configurations for different environments (such as development, staging, and production) and switch between them without changing your code.

By default, the SDK uses the `[default]` profile. To use a different profile, pass the `profile` parameter to `resolve()`:

```
config = await AsyncBedrockRuntimeConfig.resolve(profile="production")
```

Or set the `AWS_PROFILE` environment variable:

```
export AWS_PROFILE=production
```

Example config file with multiple profiles:

```
# ~/.aws/config
[default]
region = us-east-1
```

```
retry_mode = standard

[profile production]
region = us-west-2
retry_mode = standard
max_attempts = 10

[profile staging]
region = eu-west-1
endpoint_url = https://staging.internal.example.com
```

If the requested profile doesn't exist in the config file, a `ProfileNotFoundError` is raised with a message indicating where the profile name came from:

```
config = await AsyncBedrockRuntimeConfig.resolve(profile="typo")
# raises ProfileNotFoundError: Profile 'typo' from profile argument was not found in
 config file.
```

```
export AWS_PROFILE=typo
```

```
config = await AsyncBedrockRuntimeConfig.resolve()
# raises ProfileNotFoundError: Profile 'typo' from AWS_PROFILE environment variable was
 not found in config file.
```

The implicit `"default"` profile is exempt from this check. If no config file exists, the SDK simply uses defaults and environment variables without error.

Credential providers

Before the AWS SDK for Python can sign a request to an AWS service, it uses an *identity resolver* to obtain an *identity*. For these requests, the identity is a set of AWS credentials. A *credential resolver* is an identity resolver that retrieves AWS credentials from a configured source. Sources can include environment variables, an assumed IAM role, or another credential resolver. If you don't configure credentials, the SDK uses its default credential resolver chain.

> **Important**
>
> During Developer Preview, the AWS SDK for Python supports fewer credential sources than other AWS SDKs. For example, it does not support AWS IAM Identity Center or AWS

Management Console sign-in credentials. Support for additional credential sources is planned for future releases.

This page explains how the SDK resolves credentials and how to configure credential sources. It includes the following topics:

- Default credential resolver chain describes how the SDK finds credentials automatically.
- Configure a specific credential resolver shows how to use one credential source instead of the default chain.
- Configure static credentials shows how to provide credentials directly to a client configuration.

For more information about credential resolution across AWS SDKs and tools, see Standardized credential providers in the *AWS SDKs and Tools Reference Guide*.

Default credential resolver chain

If you don't specify static credentials or a credential resolver when you create a client, the SDK assembles the default credential resolver chain. During assembly, *credential providers* inspect the available configuration and add the applicable credential resolvers to the chain. When the SDK needs credentials, the chain calls each resolver in a predefined order. It stops at the first resolver that returns valid credentials.

To use the default chain, resolve the configuration without setting the `aws_credentials_identity_resolver` field or any of the static credential properties:

```
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig


async def create_client() -> AsyncBedrockRuntimeClient:
    config = await AsyncBedrockRuntimeConfig.resolve(region="us-east-1")
    return AsyncBedrockRuntimeClient(config=config)
```

Credential sources

The resolvers in the default chain use two kinds of credential sources:

- **Local sources** read credentials from your environment or from files on disk, such as environment variables or the shared AWS `config` and `credentials` files. The SDK supports these sources without additional packages.
- **Network-based sources** fetch credentials over the network, such as assuming an IAM role with the AWS Security Token Service or querying the Amazon EC2 Instance Metadata Service.

The chain uses a resolver for a network-based source only after you install its package. Each package registers a credential provider with the SDK. During assembly, the provider determines whether its source applies and adds the resolver to the chain. Without the package, the chain cannot include the resolver. This applies even when the source is configured in the environment or shared config file.

Each network-based source ships in a separate package:

Credential source	Package
Assume an IAM role with AWS STS	`aws-credentials-sts`
Amazon ECS and Amazon EKS container credentials	`aws-credentials-http`
Amazon EC2 Instance Metadata Service (IMDS)	`aws-credentials-imds`

Install the package for each source that your application uses:

```
python -m pip install aws-credentials-sts    # Assume role
python -m pip install aws-credentials-http   # Container credentials
python -m pip install aws-credentials-imds   # EC2 Instance Metadata Service
```

These packages also export resolver classes. To use a single source directly instead of the default chain, see Configure a specific credential resolver.

Credential retrieval order

Depending on your configuration, the chain can contain resolvers for the following credential sources. It calls the resolvers in this order and stops at the first one that returns valid credentials:

1. **Access key environment variables**

The SDK reads the `AWS_ACCESS_KEY_ID`, `AWS_SECRET_ACCESS_KEY`, and (if set) `AWS_SESSION_TOKEN` environment variables.

2. **Shared AWS `config` and `credentials` files**

   The SDK reads settings from the shared AWS `config` and `credentials` files. It uses the profile that you specify, or the `[default]` profile if you don't specify one.

   When a profile contains settings for more than one credential type, the SDK uses them in the following order:

   a. **Assume role** - If the profile has a `role_arn` setting with a `source_profile` or `credential_source`, the SDK obtains temporary credentials by calling AWS Security Token Service `AssumeRole`.

      Requires the `aws-credentials-sts` package.

   b. **Static access keys** - The SDK uses the `aws_access_key_id`, `aws_secret_access_key`, and `aws_session_token` settings from the profile.

   c. **Process credentials** - If the profile has a `credential_process` setting, the SDK runs the specified process and reads credentials from its output.

3. **Amazon ECS and Amazon EKS container credentials**

   The SDK obtains credentials from an HTTP endpoint, which it locates from the `AWS_CONTAINER_CREDENTIALS_RELATIVE_URI` or `AWS_CONTAINER_CREDENTIALS_FULL_URI` environment variables. Amazon Elastic Container Service (Amazon ECS) and Amazon Elastic Kubernetes Service (Amazon EKS) set these automatically when a task or pod has an IAM role.

   Requires the `aws-credentials-http` package.

4. **Amazon EC2 Instance Metadata Service**

   The SDK obtains credentials from the IAM role attached to the Amazon EC2 instance that the application runs on, through the Instance Metadata Service (IMDS).

   Requires the `aws-credentials-imds` package.

If no resolver returns valid credentials, the SDK raises an error when it signs the first request.

Configure a specific credential resolver

To use a particular credential source instead of the default chain, set the `aws_credentials_identity_resolver` configuration field to an instance of a resolver class. This gives you direct control over which source the SDK uses.

The following example uses `EnvironmentCredentialsResolver`, which reads credentials only from environment variables.

Imports

```
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from smithy_aws_core.identity import EnvironmentCredentialsResolver
```

Code

```
async def create_client() -> AsyncBedrockRuntimeClient:
    config = await AsyncBedrockRuntimeConfig.resolve(
        region="us-east-1",
        aws_credentials_identity_resolver=EnvironmentCredentialsResolver(),
    )
    return AsyncBedrockRuntimeClient(config=config)
```

The following resolvers are available:

Resolver	Import from	Credential source
`StaticCredentialsResolver`	`smithy_aws_core.identity`	A fixed set of credentials that you supply. For a simpler option, see Configure static credentials.
`EnvironmentCredentialsResolver`	`smithy_aws_core.identity`	Environment variables (`AWS_ACCESS_KEY_ID` , `AWS_SECRET_ACCESS_KEY` , and `AWS_SESSION_TOKEN`).
`ProcessCredentialsResolver`	`smithy_aws_core.identity`	An external command that returns credentials in its JSON output.

Resolver	Import from	Credential source
`AssumeRoleCredentialsResolver`	`aws_credentials_sts`	Temporary credentials for an IAM role, obtained by calling AWS STS `AssumeRole` with a role and source resolver that you specify.
`ProfileAssumeRoleCredentialsResolver`	`aws_credentials_sts`	Temporary credentials for an IAM role, using the assume-role settings (`role_arn` with `source_profile` or `credential_source`) in a shared config profile.
`ContainerCredentialsResolver`	`aws_credentials_http`	A container credential endpoint, such as the one provided by Amazon ECS and Amazon EKS.
`IMDSCredentialsResolver`	`aws_credentials_imds`	An Amazon EC2 instance's metadata, through the Instance Metadata Service (IMDS).

The resolvers in `smithy_aws_core.identity` are built-in with every service client. A resolver imported from an `aws_credentials_*` module requires installing that package first, see Credential sources.

Configure static credentials

To use a specific set of credentials, provide them using the `aws_access_key_id`, `aws_secret_access_key`, and (optionally) `aws_session_token` configuration fields. This is useful when your application already holds credentials, such as credentials that it retrieves from a secrets manager.

> **Warning**
>
> Don't hardcode credentials in source code. Static credentials are appropriate when your application obtains them at runtime from a secure source. For local development, prefer

short-term credentials supplied through environment variables or the shared AWS configuration files.

Imports

```
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
```

Code

```
async def create_client() -> AsyncBedrockRuntimeClient:
    config = await AsyncBedrockRuntimeConfig.resolve(
        region="us-east-1",
        aws_access_key_id=access_key_id,
        aws_secret_access_key=secret_access_key,
        aws_session_token=session_token,  # Optional; omit for long-term credentials.
    )
    return AsyncBedrockRuntimeClient(config=config)
```

HTTP configuration

Each service client sends every request through a *transport*, the component that sends the request and returns the response. A transport is protocol-agnostic, which lets the SDK support different wire protocols through a common interface. Because the SDK communicates with AWS over HTTP, a service client's transport is an *HTTP client*: an implementation of the `HTTPClient` protocol. See smithy-http on PyPI for the runtime library that defines this protocol.

Most applications don't need to change the transport. You might override it to use a different HTTP client, such as one that supports bidirectional streaming, or to provide your own implementation. This page includes the following topics:

- Provided HTTP clients covers the HTTP client implementations that the SDK provides and their capabilities.
- Override the HTTP client explains how to override the HTTP client, including how to implement a custom client.

Provided HTTP clients

The SDK provides two HTTP client implementations, both defined in smithy-http:

`AIOHTTPClient`

Built on the aiohttp library. This is the default transport for every service client and it's the only HTTP client installed by default. It communicates over HTTP/1.1 only. For more information, see the aiohttp website.

`AWSCRTHTTPClient`

Built on the AWS Common Runtime (CRT). It communicates over both HTTP/1.1 and HTTP/2, and it's currently the only transport that supports bidirectional (duplex) event streaming.

Each provided client accepts a configuration object for connection settings. The available settings are limited in the current release. Additional settings are planned for future releases.

Override the HTTP client

To use a transport other than the default, construct it and pass it as the `transport` field when you resolve the configuration.

Use the AWS CRT client for bidirectional streaming

Some operations use bidirectional (duplex) event streams. The default aiohttp transport doesn't support duplex streaming. To use these operations, install the client's `awscrt` extra and set `AWSCRTHTTPClient` as the transport.

The `awscrt` extra pulls in a compatible version of the awscrt library from PyPI. Install it as part of the client package:

```
python -m pip install "aws-sdk-bedrock-runtime[awscrt]"
```

Imports

```
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from smithy_http.aio.crt import AWSCRTHTTPClient
```

Code

```
async def create_client() -> AsyncBedrockRuntimeClient:
    config = await AsyncBedrockRuntimeConfig.resolve(
        region="us-east-1",
        transport=AWSCRTHTTPClient(),
    )
    return AsyncBedrockRuntimeClient(config=config)
```

You can use `AWSCRTHTTPClient` for a client even when you don't need bidirectional streaming; the `awscrt` extra is required in either case.

Provide a custom HTTP client

For full control over how requests are sent, implement the `HTTPClient` protocol and pass an instance as the `transport`. This is an advanced option; the built-in clients are sufficient for most applications. A custom client implements a `send` method that accepts an `HTTPRequest` and returns an `HTTPResponse`.

If your custom HTTP client holds network resources, such as sessions or connection pools, implement an asynchronous `close` method to release them. The generated service client's `close` method calls the transport's `close` method when present. This happens when you invoke `close` directly or exit the client's asynchronous context.

Imports

```
from smithy_http.aio.interfaces import HTTPClient, HTTPRequest, HTTPResponse
from smithy_http.interfaces import (
    HTTPClientConfiguration,
    HTTPRequestConfiguration,
)
```

Code

```
class CustomHTTPClient(HTTPClient):
    def __init__(
        self, *, client_config: HTTPClientConfiguration | None = None
    ) -> None:
        self._client_config = client_config
        # Initialize your underlying HTTP client here.
```

```
    async def send(
        self,
        request: HTTPRequest,
        *,
        request_config: HTTPRequestConfiguration | None = None,
    ) -> HTTPResponse:
        # Send the request with your HTTP client and return an HTTPResponse.
        ...

    async def close(self) -> None:
        # Release resources held by your underlying HTTP client.
        ...
```

Pass an instance of your client as the `transport` in the same way as a provided client.

Retries

This page explains how the SDK retries failed requests and how to configure retry behavior. It includes the following topics:

- Retryable errors describes which errors are retried and which are not.
- Standard retry strategy explains the backoff, jitter, and token bucket mechanics.
- Configuring retries shows how to set retry mode, max attempts, and custom strategies.

Calls to AWS services occasionally fail for reasons that have nothing to do with your request being wrong. A connection drops, a host is briefly unhealthy, or the service throttles you because too many requests arrived at the same time. Errors of this kind are often successful if the call is simply made again after a short delay, so every client in this SDK retries them automatically. This page explains how that retry behavior works and how to configure it.

Retryable errors

Only errors that are plausibly transient are retried. These include socket-level failures where no HTTP response was received (connect timeouts, read timeouts, or server-closed connections), HTTP 500, 502, 503, and 504 responses, and a set of service error codes the SDK recognizes as transient or throttling. Errors that signal a problem with the request itself like validation errors, missing or malformed parameters, authentication and authorization failures, and resource-not-found responses are never retried, because sending the same request again cannot change the outcome.

Services can also mark their own exceptions as retryable in their service model, and the SDK honors that in addition to the built-in list.

Standard retry strategy

The standard retry strategy is the default. It makes a maximum of 3 attempts for a single request, including the initial attempt. This means a request is retried up to 2 times. Each retry is delayed using exponential backoff with jitter. The strategy also maintains a retry quota that prevents additional attempts when retries keep failing.

Backoff header: x-amz-retry-after

If a response carries an `x-amz-retry-after` header, the SDK incorporates it into the backoff calculation. The value is an integer number of milliseconds. If it is shorter than the computed delay, the SDK waits the computed delay; if it is longer than the computed delay plus 5 seconds, the SDK waits the computed delay plus 5 seconds. It is not jittered, and the 20-second maximum delay does not apply to it. Invalid values (such as invalid format or exceeding integer limits) specified in `x-amz-retry-after` are ignored and the SDK falls back to exponential backoff. The standard HTTP `Retry-After` header is ignored.

Calculating exponential backoff and retry quotas

The base delay doubles with every retry and is capped at 20 seconds. That delay is then multiplied by a random value between 0 and 1. The base delay is 1 second for throttling errors and 50 milliseconds for all other retryable errors. A transient error is therefore retried after up to 50ms, then up to 100ms, while a throttling error is retried after up to 1 second, then up to 2 seconds. DynamoDB and DynamoDB Streams clients make 4 attempts and use a base delay of 25 milliseconds for non-throttling errors.

The retry quota is a token bucket held per client. Each retry must take tokens from the bucket, and if the bucket does not have enough, the SDK returns the error without retrying. The bucket holds 500 tokens. A retry costs 14 tokens after a non-throttling error and 5 after a throttling error; a successful operation returns what its retry consumed, and an operation that succeeds without retrying adds 1 token, up to the 500-token maximum. When the bucket cannot cover the next retry, the SDK raises the error immediately without retrying. The first attempt is always made, only retries are affected.

Summary: retry strategy default values

The following table shows the default values for the properties of the standard retry strategy.

Property	Default value
Maximum attempts	3
Base delay for non-throttling errors	50 ms
Base delay for throttling errors	1000 ms
Maximum delay	20 seconds
Token bucket size	500
Token cost per non-throttling retry	14
Token cost per throttling retry	5
Tokens added when an operation succeeds without retrying	1

Note

DynamoDB and DynamoDB Streams clients use a maximum of 4 attempts and a base delay of 25 ms for non-throttling errors.

Configuring retries

The standard retry mode is the only retry mode that this SDK currently supports. Setting `retry_mode` explicitly to anything other than `standard` raises a validation error during resolution. Because environment variables and shared config files are frequently shared across tools and SDKs, a `legacy` or `adaptive` value picked up from those sources is not treated as an error. It is mapped to `standard` and a warning is emitted, so a config file written for another SDK does not break your client.

The two settings that control retry behavior are the retry mode and the maximum number of attempts. If you set neither, resolving a configuration fills both in for you:

```
config = await AsyncBedrockRuntimeConfig.resolve()

print(config.retry_mode)     # "standard"
```

```
print(config.max_attempts)   # None
```

> **Note**
>
> `max_attempts` resolves to None when not explicitly set. This means the retry strategy uses its own built-in default (typically 3 for the standard strategy). It does not mean retries are disabled.

Values are resolved in order of precedence. An explicit argument to `resolve()` wins, then the environment variable, then the shared config file, and finally the built-in default. Precedence is evaluated per setting rather than per source, so `max_attempts` can come from your config file while `retry_mode` comes from the environment.

Both can be set in the environment:

```
export AWS_RETRY_MODE=standard
export AWS_MAX_ATTEMPTS=5
```

Or in the shared config file:

```
# ~/.aws/config
[default]
retry_mode = standard
max_attempts = 5
```

Or in code, when resolving a client configuration:

```
config = await AsyncBedrockRuntimeConfig.resolve(
    retry_mode="standard",
    max_attempts=5,
)
async with AsyncBedrockRuntimeClient(config=config) as client:
    response = await client.some_operation(input)
```

`max_attempts` counts total attempts, not retries, so setting it to 1 disables retries entirely while still making the original request:

```
config = await AsyncBedrockRuntimeConfig.resolve(max_attempts=1)
```

If you need behavior the built-in strategy does not provide, you can supply a retry strategy of your own. A strategy passed this way takes precedence, and any `retry_mode` or `max_attempts` set outside of it is not consulted.

```
from smithy_core.aio.retries import SimpleRetryStrategy

config = await AsyncBedrockRuntimeConfig.resolve(
    retry_strategy=SimpleRetryStrategy(max_attempts=10),
)

print(config.retry_strategy.max_attempts)  # 10
```

You can also implement your own retry strategy if the built-in options do not meet your needs. A custom strategy must satisfy the `RetryStrategy` protocol (`smithy_core.aio.interfaces.retries.RetryStrategy`), which requires a `backoff_strategy` attribute, a `max_attempts` attribute, and three async methods: `acquire_initial_retry_token`, `refresh_retry_token_for_retry`, and `record_success`. Any class that implements these is accepted by the SDK.

Using the AWS SDK for Python

This chapter describes general programming patterns for supported AWS services, beyond what is covered in Getting started with the AWS SDK for Python. For service-specific operations, see Working with AWS services.

Before using these patterns, configure credentials and an AWS Region as described in Configuration.

Topics

- Making requests and handling responses
- Using the SDK asynchronously
- Running concurrent operations
- Working with event streams
- Handling errors
- Customizing SDK behavior with plugins and interceptors
- Testing applications that use the SDK

Making requests and handling responses

Before making a request, instantiate the generated client for the service and construct the operation's input object. Each operation returns a service-specific output object. Type checkers and editors can use the generated types to identify available members before the application runs.

The operation examples on this page run inside a coroutine and use `await`. For application entry points and event loop behavior, see Using the SDK asynchronously.

Creating a service client

Create the client using the configuration object required by the service. This example uses a DynamoDB client that overrides the Region; credentials come from the default credential chain.

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig


config = await AsyncDynamoDBConfig.resolve(region="us-east-1")
```

```
client = AsyncDynamoDBClient(config=config)
```

For other client options and credential configuration, see Configuration.

Reuse a client for operations that use the same client-wide configuration. Create a separate client when you need different client-wide configuration.

The client owns network resources, such as HTTP connections. The client is an *asynchronous context manager*: an object that an `async with` statement can open and close. Leaving the `async with` block closes the client and releases those resources.

```
from aws_sdk_dynamodb.models import ListTablesInput

async with client:
    response = await client.list_tables(
        input=ListTablesInput(limit=25)
    )
```

When the client's lifetime doesn't match a single block, such as a client that the application creates at startup and shares across functions, call `await client.close()` after the last operation. The following example closes the client in a `finally` block.

```
try:
    response = await client.list_tables(
        input=ListTablesInput(limit=25)
    )
finally:
    await client.close()
```

Don't invoke operations on a closed client; create a new client instead.

Constructing a typed request

Import the operation input object from the service package and pass it as the operation's `input` parameter. All generated types for a service, including operation inputs and outputs, are located in the service package's `models` module, such as `aws_sdk_dynamodb.models`. Model member names use Python snake case.

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import ListTablesInput, ListTablesOutput
```

```
async def list_table_page(
    client: AsyncDynamoDBClient,
    start_name: str | None = None,
) -> ListTablesOutput:
    request = ListTablesInput(
        exclusive_start_table_name=start_name,
        limit=25,
    )
    return await client.list_tables(input=request)
```

Operations also accept optional plugins that customize configuration for that invocation only. For details, see Customizing SDK behavior with plugins and interceptors.

Reading optional response members

Generated response members are optional when a service can omit them. Check for None, or use an appropriate empty value, before processing the result.

```
page = await list_table_page(client)
for table_name in page.table_names or []:
    print(table_name)

if page.last_evaluated_table_name is not None:
    print(f"continue after: {page.last_evaluated_table_name}")
```

Handling paginated responses

The AWS SDK for Python doesn't currently provide paginator objects, so applications must iterate pages explicitly. When an operation returns a continuation token, pass it as the start token member of the next request. Continue until the response omits the token.

```
from collections.abc import AsyncIterator

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import ListTablesInput


async def iter_table_names(
    client: AsyncDynamoDBClient,
) -> AsyncIterator[str]:
```

```
    start_name: str | None = None

    while True:
        page = await client.list_tables(input=ListTablesInput(
            exclusive_start_table_name=start_name,
            limit=25,
        ))
        for table_name in page.table_names or []:
            yield table_name

        start_name = page.last_evaluated_table_name
        if start_name is None:
            return
```

Token member names and page-size limits are operation-specific. Process each page as it arrives, since the complete result set might be too large to keep in memory.

Following the modeled shape

Most operation responses are instances of generated Python classes rather than untyped response dictionaries. Their members correspond to fields defined by the service API, and each member's type depends on the API model. For example, a field modeled as a map is represented by a typed Python dictionary, while a field modeled as a document is represented by a `Document` object that can contain object, array, scalar, or null data.

DynamoDB items are one example of a modeled map. Each item is a dictionary whose keys are attribute names. Each value is a generated class that identifies the DynamoDB attribute type, such as `AttributeValueS` for a string. The following example checks the value's class before reading it.

```
from aws_sdk_dynamodb.models import AttributeValueS, GetItemInput

response = await client.get_item(
    input=GetItemInput(
        table_name="Books",
        key={"id": AttributeValueS(value="book-123")},
    )
)

if response.item is not None:
    title = response.item.get("title")
    if isinstance(title, AttributeValueS):
```

```
        print(title.value)
```

Narrowing union members

A union is a model member that can contain one of several possible variants. The SDK represents each variant with a different generated Python class, and only one variant is present in a union value. Use `isinstance()` or pattern matching to identify that class before accessing its `value`. The DynamoDB attribute value in the preceding section is one example of a union: each `AttributeValue` class is a variant of that union.

Some responses nest unions inside unions. In the following Amazon Bedrock Runtime example, `response.output` can be one of several output variants. The function first checks for the message variant and then checks each content block for the text variant.

```
from aws_sdk_bedrock_runtime.models import (
    ContentBlockText,
    ConverseOperationOutput,
    ConverseOutputMessage,
)


def print_converse_text(response: ConverseOperationOutput) -> None:
    if not isinstance(response.output, ConverseOutputMessage):
        raise RuntimeError(
            f"Unexpected output type: {type(response.output).__name__}"
        )

    for block in response.output.value.content:
        if isinstance(block, ContentBlockText):
            print(block.value)
```

Using the SDK asynchronously

Some AWS service requests spend significant time waiting on network I/O, such as streaming responses, batch writes, or calls to geographically distant endpoints. The SDK's asynchronous clients let your application continue running while these requests wait for responses. These asynchronous operations are currently the only form of operation that the SDK provides.

Calling an asynchronous operation creates a *coroutine*: an object that represents work that runs only when you await it. While one coroutine awaits, the event loop can run other operations.

Running independent operations concurrently reduces the impact of network latency and improves throughput. This page explains how to await SDK operations, start asynchronous code from different entry points, and avoid common pitfalls.

Understanding coroutines and await

Calling an asynchronous client method creates a coroutine but does not send the request. Awaiting the coroutine starts the operation and pauses only the current coroutine until the response is ready; it does not block the entire event loop.

```
async def table_names(
    client: AsyncDynamoDBClient,
) -> list[str]:
    operation = client.list_tables(
        input=ListTablesInput(limit=25)
    )
    # No request has been sent yet. Awaiting starts the operation.
    response = await operation
    return response.table_names or []
```

To start multiple independent operations before awaiting their results, schedule them as tasks as described in Running concurrent operations.

Running asynchronous code

Starting a coroutine depends on who owns the application's event loop. A standalone script creates the loop at its synchronous entry point, while an asynchronous framework or interactive environment provides a loop for your code.

Starting from a script

Use `asyncio.run()` once at the synchronous entry point of a script. Create the client inside the asynchronous application flow and await its operations.

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig
from aws_sdk_dynamodb.models import ListTablesInput
```

```
async def main() -> None:
    config = await AsyncDynamoDBConfig.resolve(region="us-east-1")
    async with AsyncDynamoDBClient(config=config) as client:
        response = await client.list_tables(
            input=ListTablesInput(limit=25)
        )
        print(response.table_names or [])


if __name__ == "__main__":
    asyncio.run(main())
```

Running inside an existing event loop

When an asynchronous framework invokes your handler, the framework already owns the event loop. Define an asynchronous handler and await SDK operations from it instead of calling `asyncio.run()` again.

```
async def handle_request(
    client: AsyncDynamoDBClient,
) -> dict[str, list[str]]:
    response = await client.list_tables(
        input=ListTablesInput(limit=25)
    )
    return {"table_names": response.table_names or []}
```

Interactive environments such as Jupyter notebooks also provide an event loop and support top-level `await`. In a notebook cell, call a coroutine directly, such as the `handle_request` function above: `result = await handle_request(client)`.

Using asynchronous operations correctly

Awaiting or scheduling SDK operations

Every coroutine returned by an SDK operation must be awaited directly or scheduled as a task that the application later awaits. If neither happens, the request is not sent.

```
# Incorrect: the request is not sent.
response = client.list_tables(input=ListTablesInput(limit=25))

# Correct: awaiting the coroutine sends the request.
```

```
response = await client.list_tables(
    input=ListTablesInput(limit=25)
)
```

Keeping blocking work off the event loop

A blocking library call pauses every task on the event loop. When no asynchronous alternative exists, move the blocking call to a worker thread with `asyncio.to_thread()`.

```
from pathlib import Path


def read_template(path: Path) -> str:
    return path.read_text()


template = await asyncio.to_thread(
    read_template, Path("request-template.json")
)
```

Closing event streams

A non-streaming operation returns a fully deserialized response, so application code has no response resource to close. An event-streaming operation remains open while events are received; close it with its asynchronous context manager as described in Working with event streams.

Running concurrent operations

The Using the SDK asynchronously page showed how to await a single operation without blocking the event loop. When your application has multiple independent calls to make, you can go further: schedule them as concurrent tasks so their network wait time overlaps.

This page demonstrates patterns to run multiple operations using existing client APIs or with concurrency primitives provided by Python's `asyncio` library.

Using service batch operations when available

Before scheduling one SDK call for each input item, check whether the service provides a batch operation. A batch operation is a service API, not a Python concurrency mechanism: it reduces the number of service requests by accepting multiple items in one request. When no suitable batch

operation exists, independent SDK calls can make progress concurrently while they wait for service responses.

Examples of batch operations include:

- **DynamoDB:** batch_get_item() and batch_write_item()
- **Amazon SQS:** send_message_batch()

A batch operation isn't necessarily atomic or all-or-nothing. Follow the operation-specific size limits and inspect its modeled response for failed entries or unprocessed work. Resubmit only eligible failed or unprocessed entries, with bounded attempts and a delay between attempts.

When an input exceeds an operation's batch-size limit, divide it into valid batches. If you submit multiple batches concurrently, use Bounding fan-out for a finite collection to limit the number of batch requests in flight.

Running a small number of operations concurrently

When the number of independent operations is small and known before scheduling, and it's appropriate for all of them to be in flight at once, pass their coroutines to `asyncio.gather()`. Awaiting `gather()` lets the operations make progress concurrently and returns successful results in input order.

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import DescribeTableInput


async def describe_two_tables(
    client: AsyncDynamoDBClient,
    first_table_name: str,
    second_table_name: str,
):
    responses = await asyncio.gather(
        client.describe_table(
            input=DescribeTableInput(table_name=first_table_name)
        ),
        client.describe_table(
            input=DescribeTableInput(table_name=second_table_name)
        ),
```

```
    )
    return [response.table for response in responses]
```

If one operation raises an exception, `gather()` propagates it but doesn't automatically cancel the other operations. If the input size can vary or might be large, use Bounding fan-out for a finite collection instead.

Bounding fan-out for a finite collection

Fan-out creates one request for each input item. For a finite input collection of manageable size, this pattern uses `gather()` to collect the results and a semaphore to limit how many SDK calls are active at once.

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import DescribeTableInput


async def describe_tables_bounded(
    client: AsyncDynamoDBClient,
    table_names: list[str],
    maximum_concurrency: int = 10,
):
    if maximum_concurrency < 1:
        raise ValueError("maximum_concurrency must be at least 1")
    limit = asyncio.Semaphore(maximum_concurrency)

    async def describe(table_name: str):
        async with limit:
            response = await client.describe_table(
                input=DescribeTableInput(table_name=table_name)
            )
            return response.table

    return await asyncio.gather(*(
        describe(name) for name in table_names
    ))
```

The semaphore bounds active SDK calls, but `gather()` still creates and schedules one coroutine for every item in `table_names`. This pattern also has the exception behavior described in the preceding section.

For a very large collection or input that arrives over time, don't pass the entire source to `gather()`. Use a bounded producer and worker pattern instead. For example, a producer can add items to an `asyncio.Queue` with a fixed `maxsize`, and a fixed number of workers can remove items and invoke the SDK operation. The producer waits when the queue is full and reads or generates more input as workers finish. This bounds both active SDK calls and work waiting in memory.

Choose the concurrency or worker limit based on the service quota, operation cost, expected latency, and the application's available memory. Higher concurrency doesn't bypass service quotas and can increase throttling.

Working with event streams

A standard request-response call works well when the service can return a complete result at once, but some workloads — live transcription, real-time inference, interactive conversations — produce data continuously while an operation is still running. For these cases, the SDK provides event stream operations that stay open and exchange a sequence of modeled events over time. The SDK supports multiple types of event streams:

- A unidirectional *output event stream*: The service sends events to the application. The application reads events until the service closes the output.
- A bidirectional *duplex event stream*: The application sends events through `input_stream` while it receives service events from the output side.

A service can also model an input-only event stream, in which only the application sends events. The pattern for sending events matches the input side of a bidirectional stream.

Every event stream's lifecycle is managed with an asynchronous context manager. Leaving the context closes its resources. The service model defines the generated input and output event types and any other events required to complete the operation.

This page explains how to invoke and handle output and bidirectional event streams.

Consuming an output-only stream

An output event stream operation returns an open stream object. Calling the operation starts the request, but event data arrives later as the application iterates `output_stream`. Each item is one generated variant of the operation's output event union. The union represents the several possible event types.

The following example follows the stream lifecycle: call the operation, enter its asynchronous context, and wait for events with `async for`. It collects payload chunks and reports an event type that the current client doesn't recognize.

```
from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.models import (
    InvokeModelWithResponseStreamInput,
    ResponseStreamChunk,
    ResponseStreamUnknown,
)


async def collect_response_chunks(
    client: AsyncBedrockRuntimeClient,
    request: InvokeModelWithResponseStreamInput,
) -> list[bytes]:
    response = await client.invoke_model_with_response_stream(input=request)
    chunks: list[bytes] = []

    async with response as stream:
        async for event in stream.output_stream:
            if isinstance(event, ResponseStreamChunk):
                if event.value.bytes_:
                    chunks.append(event.value.bytes_)
            elif isinstance(event, ResponseStreamUnknown):
                raise RuntimeError(
                    f"Unknown output event: {event.tag}"
                )
            else:
                raise RuntimeError(
                    f"Unexpected output event: {type(event).__name__}"
                )

    return chunks
```

Leaving the asynchronous context closes the output stream even when event processing raises an exception. A modeled stream error can arrive while the application is iterating events; the SDK raises it as a generated service exception at that point. The event payloads and the signal that completes a stream vary by operation. For details, see the invoke_model_with_response_stream() API reference.

Using a bidirectional event stream

A bidirectional application sends input and receives output at the same time. Its overall lifecycle is:

1. Call the operation to open the stream, and enter the stream's asynchronous context.
2. Run a sender and receiver concurrently so that neither side waits for the other side to finish first.
3. Send modeled input events. After the last required input or completion event, close `input_stream` to indicate that no more input will follow.
4. Continue processing output events until the service closes the output side, and then leave the context to release the stream resources.

Bidirectional operations require the AWS Common Runtime (CRT) HTTP client as the client's transport. To opt in, install the client's `awscrt` extra and set `AWSCRTHTTPClient` as the transport. For more information, see Use the AWS CRT client for bidirectional streaming.

The following example uses Amazon Transcribe types to make the bidirectional pattern concrete. It represents application-provided input as an asynchronous iterable and delegates output events to an application-provided handler. This keeps producing and interpreting service payloads separate from opening, using, and closing the stream.

The following snippet contains only the imports that the rest of this section shares.

```
import asyncio
from collections.abc import AsyncIterable, Callable

from smithy_core.aio.interfaces.eventstream import EventPublisher, EventReceiver

from aws_sdk_transcribe_streaming.client import AsyncTranscribeStreamingClient
from aws_sdk_transcribe_streaming.models import (
    AudioEvent,
    AudioStream,
    AudioStreamAudioEvent,
    StartStreamTranscriptionInput,
    TranscriptEvent,
    TranscriptResultStream,
    TranscriptResultStreamTranscriptEvent,
    TranscriptResultStreamUnknown,
)
```

Sending input events

A bidirectional stream exposes `input_stream` for events sent by the application. The operation model defines an input event union, and each generated union variant represents one event type. For Amazon Transcribe, `AudioStreamAudioEvent` is the variant that carries an `AudioEvent`.

The sender wraps each prepared payload in the generated event classes and awaits `send()`. The asynchronous source determines how the application produces those payloads.

```
async def send_audio(
    input_stream: EventPublisher[AudioStream],
    audio_chunks: AsyncIterable[bytes],
) -> None:
    """Send prepared audio chunks to the input side of the stream."""
    try:
        async for chunk in audio_chunks:
            await input_stream.send(
                AudioStreamAudioEvent(value=AudioEvent(audio_chunk=chunk))
            )
    finally:
        await input_stream.close()
```

Closing `input_stream` half-closes the operation: the application can send no more input events, but the service can continue returning output events. The `finally` block closes the input side even if producing or sending a chunk fails. Closing this side doesn't close the output side or discard output that is still arriving.

Receiving output events

A bidirectional operation can return an initial modeled response before it starts yielding output events. This response contains the non-streaming members of the operation output; a separate receiver supplies the later events. The coordinating function below calls `await_output()` to wait for both, ignores the initial response, and passes the receiver to `receive_events()`.

Each received item is one variant of the generated output event union. The receiver validates the variant and delegates the modeled event payload to an application-provided handler. The handler determines how to interpret or store the service-specific payload.

```
async def receive_events(
    output_stream: EventReceiver[TranscriptResultStream],
```

```
    handle_event: Callable[[TranscriptEvent], None],
) -> None:
    """Receive modeled events and pass them to an application handler."""
    async for event in output_stream:
        if isinstance(event, TranscriptResultStreamUnknown):
            raise RuntimeError(f"Unknown stream event: {event.tag}")
        if not isinstance(event, TranscriptResultStreamTranscriptEvent):
            raise RuntimeError(f"Unexpected stream event: {type(event).__name__}")

        handle_event(event.value)
```

A modeled service error can also arrive as the application iterates the receiver; the SDK raises it as a generated exception. A generated `TranscriptResultStreamUnknown` value means that the service sent an event variant the installed SDK doesn't recognize. Its `tag` identifies that unrecognized variant so the application can report it without treating it as a known event.

When iteration ends, the service has closed `output_stream` and no more output events remain.

Coordinating both sides

If an application waits for all input to finish before reading output, the service or client can block while output waits to be consumed. The following function accepts an application-configured request and event handler, starts the operation, enters the stream context, and uses `asyncio.gather()` to run the sender and receiver concurrently.

```
async def transcribe_audio(
    client: AsyncTranscribeStreamingClient,
    request: StartStreamTranscriptionInput,
    audio_chunks: AsyncIterable[bytes],
    handle_event: Callable[[TranscriptEvent], None],
) -> None:
    """Exchange events over a bidirectional transcription stream."""
    stream = await client.start_stream_transcription(input=request)

    async with stream:
        _, output_stream = await stream.await_output()
        if output_stream is None:
            raise RuntimeError("The service returned no output stream")
        await asyncio.gather(
            send_audio(stream.input_stream, audio_chunks),
            receive_events(output_stream, handle_event),
        )
```

On successful completion, `send_audio` closes the input side after the source has no more chunks, while `receive_events` continues until the service closes the output side. Leaving the asynchronous stream context then releases both sides.

The operation model defines how a bidirectional stream ends. Send any required completion event before closing `input_stream`.

See Example 2: Stream audio bidirectionally with Amazon Transcribe in the Getting Started chapter for a runnable application. The SDK repository also provides examples for a prerecorded file and live microphone input on GitHub.

Applying the pattern to other operations

The following generated Python methods support HTTP/2 bidirectional event streams:

- **Amazon Bedrock Runtime:** invoke_model_with_bidirectional_stream()
- **Amazon Connect Health:** start_medical_scribe_listening_session()
- **Amazon Lex Runtime V2:** start_conversation()
- **Amazon Polly:** start_speech_synthesis_stream()
- **Amazon Q Business:** chat()
- **Amazon SageMaker Runtime HTTP2:** invoke_endpoint_with_bidirectional_stream()
- **Amazon Transcribe Streaming:** start_call_analytics_stream_transcription(), start_medical_scribe_stream(), start_medical_stream_transcription(), and start_stream_transcription()

Each operation defines its own input and output event variants and the signal used to finish the stream. Read the operation API reference before reusing the Amazon Transcribe coordination pattern. For the Amazon Nova Sonic protocol used by Bedrock Runtime, see Use bidirectional streaming.

Handling errors

AWS service calls can fail for reasons your application must anticipate: a resource doesn't exist, a request is throttled, or the network is unreachable. The AWS SDK for Python surfaces these failures as typed exceptions rather than return codes, letting you use standard try/except blocks to handle each failure mode precisely. This page shows how to catch modeled service errors, handle SDK

runtime exceptions, and understand the retry behavior that occurs before an error reaches your code.

Handling a modeled service error

Modeled exceptions describe failures that a service operation can return. The following example catches `ResourceNotFoundException` because a missing DynamoDB table is an expected result for this application. It lets every other failure propagate.

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    DescribeTableInput,
    ResourceNotFoundException,
)


async def table_exists(
    client: AsyncDynamoDBClient, table_name: str
) -> bool:
    try:
        await client.describe_table(
            input=DescribeTableInput(table_name=table_name)
        )
    except ResourceNotFoundException:
        return False
    return True
```

Avoid catching a broad exception near an operation when the application doesn't know how to recover from it.

Handling errors at an application boundary

An application boundary is a place where unhandled failures receive one common response, such as a command-line entry point, job handler, or web request handler. `SmithyError` is the base class for exceptions raised by the SDK, including modeled service exceptions and SDK runtime exceptions.

The following example catches `SmithyError` after `table_exists` has handled the specific failure it understands. The boundary translates any remaining SDK failure into an application error.

```
from smithy_core.exceptions import SmithyError
```

```
async def safely_check_table(
    client: AsyncDynamoDBClient, table_name: str
) -> bool:
    try:
        return await table_exists(client, table_name)
    except SmithyError as error:
        raise RuntimeError(
            "The SDK could not complete the DynamoDB request"
        ) from error
```

Exceptions don't expose a common response dictionary. Read fields only from the concrete generated exception type that defines them.

Understanding retry behavior

The SDK retry strategy can automatically repeat a request after a failure that it classifies as retryable. It stops after a bounded number of attempts. Therefore, an exception that reaches the application can mean that the SDK has already attempted the request more than once and couldn't complete it.

An application-level retry starts a new operation invocation. Add one only for a transient failure and only when repeating the operation is safe. An operation is *idempotent* when sending the same request again doesn't cause an additional change after the first successful request. Always limit retry attempts so a persistent failure doesn't cause an unbounded retry loop or add sustained load to the service.

Some operations provide an idempotency token to identify repeated requests. Others can partially succeed, so retrying the entire request might repeat work that already completed. These behaviors are service-specific; check the operation API reference before adding an application retry. For DynamoDB examples, see Use DynamoDB batch operations and transactions.

Customizing SDK behavior with plugins and interceptors

When you invoke an operation, the SDK processes it through its request pipeline. The pipeline automatically handles stages such as serialization, signing, sending an HTTP request, and deserialization. However, some applications need to observe or adjust that behavior. Plugins and interceptors customize different parts of this process.

- A *plugin* is a setup function that receives the configuration object before the pipeline is created. Use a plugin to change configuration, such as the Region, or to register other custom configurations.
- An *interceptor* is an object whose hook methods are called by the pipeline while an operation runs. Use an interceptor to observe or modify requests and responses at defined stages, such as before serialization, before transmission, or after execution.

This page shows how to write both, when to scope them to a client versus a single operation, and which lifecycle hooks to choose.

Customizing configuration with plugins

To define a plugin, write a callable that accepts the generated service configuration and returns None. The SDK applies a client plugin directly to the client's configuration. It applies an operation plugin to a copy of that configuration, so the changes affect only that invocation.

The following example defines `use_region`, which returns a plugin that sets the Region in a DynamoDB `AsyncDynamoDBConfig`. Passing the plugin to the client establishes the Region used by that client.

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig, Plugin
from aws_sdk_dynamodb.models import ListTablesInput


def use_region(region: str) -> Plugin:
    def configure(config: AsyncDynamoDBConfig) -> None:
        config.region = region

    return configure


config = await AsyncDynamoDBConfig.resolve()
client = AsyncDynamoDBClient(
    config=config,
    plugins=[use_region("us-east-1")],
)
```

The following call passes another `use_region` plugin to `list_tables`. The SDK applies it to the operation's copy of the configuration, changing the Region only for this call. Later calls continue to use the client's configured Region.

```
async with client:
    response = await client.list_tables(
        input=ListTablesInput(limit=25),
        plugins=[use_region("us-west-2")],
    )
```

Hooking into request processing with interceptors

An interceptor implements hook methods that the request pipeline calls at defined stages. The following interceptor logs when an SDK operation starts and whether the completed execution succeeded or failed. It supports every operation because it doesn't depend on a service-specific input or output type.

```
import logging
from typing import Any

from aws_sdk_dynamodb.config import AsyncDynamoDBConfig
from smithy_core.interceptors import (
    InputContext,
    Interceptor,
    OutputContext,
)


logger = logging.getLogger(__name__)


class ExecutionLogger(Interceptor[Any, Any, Any, Any]):
    def read_before_execution(
        self, context: InputContext[Any]
    ) -> None:
        logger.info("Starting SDK operation")

    def read_after_execution(
        self, context: OutputContext[Any, Any, Any, Any]
    ) -> None:
        if isinstance(context.response, Exception):
            logger.warning("SDK operation failed")
```

```
        else:
            logger.info("SDK operation completed")
```

Registering an interceptor on a client

To run an interceptor for every operation invoked by a client, include the interceptor when you resolve the configuration. The following example doesn't define or pass a plugin.

```
config = await AsyncDynamoDBConfig.resolve(
    region="us-east-1",
    interceptors=[ExecutionLogger()],
)
async with AsyncDynamoDBClient(config=config) as logged_client:
    response = await logged_client.list_tables(
        input=ListTablesInput(limit=25)
    )
```

The client copies this configuration for each operation, so each operation pipeline includes the registered interceptor. An interceptor registered at this scope must support every operation that the client can invoke.

Registering an interceptor for one operation

An operation doesn't accept an `interceptors` argument. To register an interceptor for one invocation, define an operation plugin that adds it to the copied configuration used by that invocation.

```
def add_execution_logger(config: AsyncDynamoDBConfig) -> None:
    config.interceptors.append(ExecutionLogger())


response = await client.list_tables(
    input=ListTablesInput(limit=25),
    plugins=[add_execution_logger],
)
```

In this example, `add_execution_logger` is the plugin and `ExecutionLogger` is the interceptor. The plugin runs before the pipeline is created and registers the interceptor in the operation's configuration copy. The interceptor then runs while that pipeline processes this call. Later calls that don't receive the plugin don't use this interceptor.

Choosing lifecycle hooks carefully

The `Interceptor` base class in the `smithy_core.interceptors` module defines the full set of hook methods; see the interceptors module on GitHub. Each hook runs at one stage of the operation:

- **Execution:** `read_before_execution`, `modify_before_completion`, and `read_after_execution`
- **Serialization:** `modify_before_serialization`, `read_before_serialization`, and `read_after_serialization`
- **Retries and attempts:** `modify_before_retry_loop`, `read_before_attempt`, `modify_before_attempt_completion`, and `read_after_attempt`
- **Signing:** `modify_before_signing`, `read_before_signing`, and `read_after_signing`
- **Transmission:** `modify_before_transmit`, `read_before_transmit`, and `read_after_transmit`
- **Deserialization:** `modify_before_deserialization`, `read_before_deserialization`, and `read_after_deserialization`

Follow these rules when choosing a hook:

- `read_*` hooks observe their context and must not modify request or response objects.
- `modify_*` hooks return a replacement request, response, transport request, or transport response of the same type.
- Execution hooks run once for an operation invocation. Attempt hooks can run more than once when the retry strategy makes another attempt.
- Interceptors run in configuration order. An exception raised by an interceptor can cause the operation to fail.

Do not log credentials, authorization headers, or sensitive request and response data from an interceptor.

Testing applications that use the SDK

A well-structured test suite clearly distinguishes what the application owns versus what the SDK handles. Your tests should verify that application code constructs the right requests, interprets

responses correctly, and recovers from expected errors, without reimplementing serialization, signing, or transport logic. This page presents three testing patterns, from the fastest and most isolated (operation mocks) to the most realistic (integration tests).

Choose the narrowest boundary that verifies the behavior your application owns:

- Mock a client operation to test application request construction, response handling, and error paths.
- Inject a mock HTTP transport when application behavior must run through the generated client pipeline without making a network request.
- Use an integration test when application behavior depends on service-side state or validation.

The examples use pytest with pytest-asyncio. Mark asynchronous tests with `pytest.mark.asyncio`, or configure pytest-asyncio with `asyncio_mode = auto`. Keep application logic in functions that receive a client so each test can supply the appropriate client implementation.

Mocking an asynchronous operation

Use `unittest.mock.AsyncMock` for a fast unit test of how application code calls an SDK operation and handles its modeled output.

```
from unittest.mock import AsyncMock

import pytest

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import ListTablesInput, ListTablesOutput


async def first_table(client: AsyncDynamoDBClient) -> str | None:
    response = await client.list_tables(
        input=ListTablesInput(limit=1)
    )
    return (response.table_names or [None])[0]


@pytest.mark.asyncio
async def test_first_table() -> None:
    client = AsyncMock(spec=AsyncDynamoDBClient)
```

```
    client.list_tables.return_value = ListTablesOutput(
        table_names=["Books"]
    )

    assert await first_table(client) == "Books"
    client.list_tables.assert_awaited_once_with(
        input=ListTablesInput(limit=1)
    )
```

This test verifies the application's request and response logic. It intentionally does not test serialization, signing, transport behavior, or service behavior.

Testing through the generated client

An operation mock replaces the SDK operation and bypasses the generated client's internal processing. A mock HTTP transport keeps the generated client in the test but replaces the component that sends requests over the network. The client still converts the input model into an HTTP request, applies authentication and interceptors, and converts the HTTP response into an output model.

Use `MockHTTPClient` only when the application behavior depends on that client processing. Because this is a low-level test, supply a raw HTTP response with the status, headers, and body required by the service protocol. The following example places that response in the mock transport and supplies the transport when it resolves the configuration.

```
import pytest

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig
from aws_sdk_dynamodb.models import ListTablesInput
from smithy_http.testing import MockHTTPClient


async def first_table(client: AsyncDynamoDBClient) -> str | None:
    response = await client.list_tables(
        input=ListTablesInput(limit=1)
    )
    return (response.table_names or [None])[0]


@pytest.mark.asyncio
```

```
async def test_first_table_through_client() -> None:
    transport = MockHTTPClient()
    transport.add_response(
        status=200,
        headers=[
            ("content-type", "application/x-amz-json-1.0"),
            ("x-amzn-requestid", "test-request-id"),
        ],
        body=b'{"TableNames":["Books"]}',
    )

    config = await AsyncDynamoDBConfig.resolve(
        region="us-east-1",
        aws_access_key_id="TEST_ONLY",
        aws_secret_access_key="TEST_ONLY",
        transport=transport,
    )
    async with AsyncDynamoDBClient(config=config) as client:
        assert await first_table(client) == "Books"
        assert transport.call_count == 1
```

The static values in this isolated test are placeholders required by request authentication; the SDK attaches a static credentials resolver for them automatically. Never use placeholder values as application credentials. `MockHTTPClient` is not a modeled service stub or service emulator, and the test is coupled to the service's HTTP protocol. Prefer an operation mock unless the application deliberately customizes or depends on the generated client pipeline.

Running application integration tests

Integration tests call application code with a real SDK client and dedicated non-production resources. Use them for application workflows that depend on service-side state transitions, validation, or modeled errors.

- Pass the client for the non-production test environment into the same functions or classes that receive a mock client in unit tests. Avoid constructing a fixed client inside the application logic.
- Generate unique resource names so parallel and repeated test runs do not collide.
- Use pytest fixtures to create the resources required by the application workflow, wait until they are ready, yield them to tests, and delete them in a `finally` block.
- Assert application-visible outcomes and error-handling behavior, including how application code handles modeled errors returned by the client.

- Run integration tests separately from unit tests because they require credentials, network access, and additional execution time.

Keep resource setup and cleanup in fixtures so each test remains focused on one application behavior. Wait for asynchronous deletion to finish, and ensure cleanup still runs when setup partially succeeds or a test assertion fails.

Working with AWS services

This chapter provides short tutorials and guidance for how to work with select AWS services using the AWS SDK for Python.

Topics

- Amazon DynamoDB
- Amazon SQS
- Amazon SNS
- Amazon Bedrock Runtime

Amazon DynamoDB

This chapter provides runnable examples for common DynamoDB workflows — creating tables, reading and writing items, running queries, and coordinating batch operations and transactions. Each example uses the SDK's asynchronous DynamoDB client with typed inputs and outputs, so you can adapt them to your application without translating from raw JSON. Start here to install the client package and configure shared setup that the remaining pages build on.

Install the DynamoDB client before running the asynchronous examples on this page:

```
python -m pip install aws-sdk-dynamodb
```

Set up the examples

The `client` that is used in the following examples can be created from the following snippet by calling `client = await create_client()` from your asynchronous code. Use the client as an asynchronous context manager, or call `await client.close()` after the last operation, as described in Creating a service client. When you resolve the generated asynchronous configuration with `resolve()`, the SDK reads the AWS Region from standard AWS settings, such as environment variables and shared AWS configuration files. The SDK also reads credentials from the same settings. Configure both before running the examples: if no Region is configured, `resolve()` fails before the client is created, and if no credentials are configured, requests fail when the SDK signs them. For more information about the Region and other settings, see Configuration variables. For how the SDK finds credentials, see Credential providers.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig
```

Code

```
async def create_client() -> AsyncDynamoDBClient:
    config = await AsyncDynamoDBConfig.resolve()
    return AsyncDynamoDBClient(config=config)
```

Topics

- Work with DynamoDB tables
- Work with DynamoDB items
- Query DynamoDB items
- Use DynamoDB batch operations and transactions

Work with DynamoDB tables

Tables are the containers for all items in a DynamoDB database. Before you can add or remove data from DynamoDB, you must create a table.

For each table, you must define:

- A table *name* that is unique for your AWS account and AWS Region.
- A *primary key* for which every value must be unique. No two items in your table can have the same primary key value.

  A primary key can be *simple*, consisting of a single partition (HASH) key, or *composite*, consisting of a partition and a sort (RANGE) key.

  Each key value has an associated data type, specified in its `AttributeDefinition` as the `attribute_type` value. The key value can be binary (`"B"`), numeric (`"N"`), or a string (`"S"`). For more information, see Naming Rules and Data Types in the Amazon DynamoDB Developer Guide.
- *Provisioned throughput* values that define the number of reserved read and write capacity units for the table.

> **Note**
>
> Amazon DynamoDB pricing is based on the provisioned throughput values that you set on your tables, so reserve only as much capacity as you think you'll need for your table. Provisioned throughput for a table can be modified at any time, so you can adjust capacity if your needs change.

The `client` that is used in the following examples can be created from the snippet in Set up the examples.

Create a table

Use `create_table` to create a new DynamoDB table. You construct table attributes and a table schema, both of which identify the primary key of your table. You must also supply initial provisioned throughput values and a table name. Each example polls with a bounded loop until DynamoDB reports that the new table is active.

Create a table with a simple primary key

For request and response details, see the create_table() and describe_table() API references.

A simple primary key contains only a partition key.

Imports

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeDefinition,
    CreateTableInput,
    DescribeTableInput,
    KeySchemaElement,
    ProvisionedThroughput,
)
```

Code

```
async def create_table(
```

```
    client: AsyncDynamoDBClient,
    table_name: str,
    max_attempts: int = 30,
    poll_delay: float = 2.0,
) -> None:
    """Create a table and wait a bounded time for it to become active."""
    await client.create_table(
        input=CreateTableInput(
            table_name=table_name,
            attribute_definitions=[
                AttributeDefinition(attribute_name="id", attribute_type="S")
            ],
            key_schema=[KeySchemaElement(attribute_name="id", key_type="HASH")],
            provisioned_throughput=ProvisionedThroughput(
                read_capacity_units=5, write_capacity_units=5
            ),
        )
    )
    for _ in range(max_attempts):
        response = await client.describe_table(
            input=DescribeTableInput(table_name=table_name)
        )
        if response.table and response.table.table_status == "ACTIVE":
            return
        await asyncio.sleep(poll_delay)
    raise TimeoutError(f"Table '{table_name}' did not become active")
```

Create a table with a composite primary key

For request and response details, see the create_table() and describe_table() API references.

A composite primary key contains a partition key and a sort key. The item and query examples use this schema.

Imports

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeDefinition,
    CreateTableInput,
    DescribeTableInput,
```

```
    KeySchemaElement,
    ProvisionedThroughput,
)
```

Code

```
async def create_table_composite_key(
    client: AsyncDynamoDBClient,
    table_name: str,
    max_attempts: int = 30,
    poll_delay: float = 2.0,
) -> None:
    """Create a table with an author partition key and title sort key."""
    await client.create_table(
        input=CreateTableInput(
            table_name=table_name,
            attribute_definitions=[
                AttributeDefinition(attribute_name="author", attribute_type="S"),
                AttributeDefinition(attribute_name="title", attribute_type="S"),
            ],
            key_schema=[
                KeySchemaElement(attribute_name="author", key_type="HASH"),
                KeySchemaElement(attribute_name="title", key_type="RANGE"),
            ],
            provisioned_throughput=ProvisionedThroughput(
                read_capacity_units=5, write_capacity_units=5
            ),
        )
    )
    for _ in range(max_attempts):
        response = await client.describe_table(
            input=DescribeTableInput(table_name=table_name)
        )
        if response.table and response.table.table_status == "ACTIVE":
            return
        await asyncio.sleep(poll_delay)
    raise TimeoutError(f"Table '{table_name}' did not become active")
```

> **Note**
>
> Applications normally create durable tables through an infrastructure deployment. This table setup is included so that the data-operation examples can use a consistent schema.

List tables

For request and response details, see the list_tables() API reference.

`list_tables` can return a partial list. Continue with `last_evaluated_table_name` until DynamoDB returns no continuation name.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import ListTablesInput
```

Code

```
async def list_tables(
    client: AsyncDynamoDBClient, max_pages: int = 100
) -> list[str]:
    """Return table names from at most ``max_pages`` response pages."""
    table_names: list[str] = []
    start_name = None
    for _ in range(max_pages):
        response = await client.list_tables(
            input=ListTablesInput(
                exclusive_start_table_name=start_name,
                limit=100,
            )
        )
        table_names.extend(response.table_names or [])
        start_name = response.last_evaluated_table_name
        if not start_name:
            return table_names
    raise RuntimeError("DynamoDB table pagination exceeded the page limit")
```

Describe a table

For request and response details, see the describe_table() API reference.

Use `describe_table` to inspect table metadata such as status, key schema, item count, and provisioned throughput.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import DescribeTableInput, TableDescription
```

Code

```
async def describe_table(
    client: AsyncDynamoDBClient, table_name: str
) -> TableDescription:
    """Return the service's description of a table."""
    response = await client.describe_table(
        input=DescribeTableInput(table_name=table_name)
    )
    if response.table is None:
        raise RuntimeError("DynamoDB returned no table description")
    return response.table
```

Update a table

For request and response details, see the update_table() and describe_table() API references.

You can modify your table's provisioned throughput values at any time with `update_table`. This example changes the provisioned read and write capacity, then waits until the table is active and confirms that DynamoDB reports the requested values.

Imports

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    DescribeTableInput,
    ProvisionedThroughput,
    UpdateTableInput,
)
```

Code

```
async def update_table(
    client: AsyncDynamoDBClient,
    table_name: str,
    capacity_units: int = 6,
    max_attempts: int = 30,
```

```
    poll_delay: float = 2.0,
) -> None:
    """Update throughput and wait a bounded time for the change."""
    await client.update_table(
        input=UpdateTableInput(
            table_name=table_name,
            provisioned_throughput=ProvisionedThroughput(
                read_capacity_units=capacity_units,
                write_capacity_units=capacity_units,
            ),
        )
    )
    for _ in range(max_attempts):
        response = await client.describe_table(
            input=DescribeTableInput(table_name=table_name)
        )
        table = response.table
        throughput = table.provisioned_throughput if table else None
        if (
            table and table.table_status == "ACTIVE" and throughput
            and throughput.read_capacity_units == capacity_units
            and throughput.write_capacity_units == capacity_units
        ):
            return
        await asyncio.sleep(poll_delay)
    raise TimeoutError(f"Table '{table_name}' was not updated")
```

Delete a table

For request and response details, see the delete_table() and describe_table() API references.

Delete each temporary table that you created after running the examples. The bounded loop confirms cleanup by waiting for the modeled `ResourceNotFoundException`.

Imports

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    DeleteTableInput,
    DescribeTableInput,
    ResourceNotFoundException,
```

```
)
```

Code

```
async def delete_table(
    client: AsyncDynamoDBClient,
    table_name: str,
    max_attempts: int = 30,
    poll_delay: float = 2.0,
) -> None:
    """Delete a table if present and wait a bounded time for removal."""
    try:
        await client.delete_table(input=DeleteTableInput(table_name=table_name))
    except ResourceNotFoundException:
        return
    for _ in range(max_attempts):
        try:
            await client.describe_table(
                input=DescribeTableInput(table_name=table_name)
            )
        except ResourceNotFoundException:
            return
        await asyncio.sleep(poll_delay)
    raise TimeoutError(f"Table '{table_name}' was not deleted")
```

More information

- Working with tables and data in DynamoDB in the Amazon DynamoDB Developer Guide
- CreateTable in the Amazon DynamoDB API Reference
- ListTables in the Amazon DynamoDB API Reference
- DescribeTable in the Amazon DynamoDB API Reference
- UpdateTable in the Amazon DynamoDB API Reference
- DeleteTable in the Amazon DynamoDB API Reference

Work with DynamoDB items

In DynamoDB, an item is a collection of *attributes*, each of which has a *name* and a *value*. An attribute value can be a scalar, set, or document type. For more information, see Naming Rules and Data Types in the Amazon DynamoDB Developer Guide.

The low-level client represents an item as a map of attribute names to typed `AttributeValue` models, such as `AttributeValueS` for a string and `AttributeValueN` for a number. Numbers are strings to preserve decimal precision. Response attributes use the same models, so check the concrete model before reading its `value`, as shown in Get an item.

The examples on this page use the composite-key table created in Create a table with a composite primary key. The `client` that is used in the following examples can be created from the snippet in Set up the examples.

Put an item

For request and response details, see the put_item() API reference.

Pass a complete map of typed attributes to `put_item`. A put replaces an existing item with the same primary key unless you add a condition.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeValueN,
    AttributeValueS,
    PutItemInput,
)
```

Code

```
async def put_item(client: AsyncDynamoDBClient, table_name: str) -> None:
    """Put a book item, replacing any item with the same primary key."""
    await client.put_item(
        input=PutItemInput(
            table_name=table_name,
            item={
                "author": AttributeValueS(value="Ada Lovelace"),
                "title": AttributeValueS(value="Notes on the Analytical Engine"),
                "year": AttributeValueN(value="1843"),
                "copies": AttributeValueN(value="0"),
            },
        )
    )
```

Get an item

For request and response details, see the get_item() API reference.

Supply every primary-key attribute to `get_item`. This example requests a strongly consistent read and returns the optional item.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import AttributeValue, AttributeValueS, GetItemInput
```

Code

```
async def get_item(
    client: AsyncDynamoDBClient, table_name: str
) -> dict[str, AttributeValue] | None:
    """Get a book by its complete composite primary key."""
    response = await client.get_item(
        input=GetItemInput(
            table_name=table_name,
            key={
                "author": AttributeValueS(value="Ada Lovelace"),
                "title": AttributeValueS(value="Notes on the Analytical Engine"),
            },
            consistent_read=True,
        )
    )
    return response.item
```

Each returned attribute is a typed `AttributeValue` union. Narrow it to its concrete model before reading its `value`.

```
if item:
    title = item.get("title")
    if isinstance(title, AttributeValueS):
        print(title.value)
```

Update an item

For request and response details, see the update_item() API reference.

Use an update expression to change selected attributes. Expression-name placeholders safely represent attribute names, and `ALL_NEW` returns the item after the update.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeValue,
    AttributeValueN,
    AttributeValueS,
    UpdateItemInput,
)
```

Code

```
async def update_item(
    client: AsyncDynamoDBClient, table_name: str
) -> dict[str, AttributeValue]:
    """Increment a book's copy count and return the updated item."""
    response = await client.update_item(
        input=UpdateItemInput(
            table_name=table_name,
            key={
                "author": AttributeValueS(value="Ada Lovelace"),
                "title": AttributeValueS(value="Notes on the Analytical Engine"),
            },
            update_expression="SET #copies = #copies + :increment",
            expression_attribute_names={"#copies": "copies"},
            expression_attribute_values={
                ":increment": AttributeValueN(value="1")
            },
            return_values="ALL_NEW",
        )
    )
    return response.attributes or {}
```

Delete an item

For request and response details, see the delete_item() API reference.

Delete one item by supplying its complete primary key.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import AttributeValueS, DeleteItemInput
```

Code

```
async def delete_item(client: AsyncDynamoDBClient, table_name: str) -> None:
    """Delete a book by its complete composite primary key."""
    await client.delete_item(
        input=DeleteItemInput(
            table_name=table_name,
            key={
                "author": AttributeValueS(value="Ada Lovelace"),
                "title": AttributeValueS(value="Notes on the Analytical Engine"),
            },
        )
    )
```

Protect a write with a condition expression

For request and response details, see the put_item() API reference.

A condition expression is evaluated before DynamoDB changes the item. This conditional put prevents replacement of an item that already has the same key. A false condition raises the modeled `ConditionalCheckFailedException` and does not change the item.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeValueS,
    ConditionalCheckFailedException,
    PutItemInput,
)
```

Code

```
async def put_item_conditionally(
    client: AsyncDynamoDBClient, table_name: str
) -> bool:
    """Put a book only when its partition key does not already exist."""
```

```
    try:
        await client.put_item(
            input=PutItemInput(
                table_name=table_name,
                item={
                    "author": AttributeValueS(value="Ada Lovelace"),
                    "title": AttributeValueS(
                        value="Notes on the Analytical Engine"
                    ),
                },
                condition_expression="attribute_not_exists(#author)",
                expression_attribute_names={"#author": "author"},
            )
        )
        return True
    except ConditionalCheckFailedException:
        return False
```

More information

- Using expressions in DynamoDB in the Amazon DynamoDB Developer Guide
- PutItem in the Amazon DynamoDB API Reference
- GetItem in the Amazon DynamoDB API Reference
- UpdateItem in the Amazon DynamoDB API Reference
- DeleteItem in the Amazon DynamoDB API Reference

Query DynamoDB items

DynamoDB provides two operations for reading multiple items. A *query* finds items by their primary-key values and is the efficient access path: it reads only the items that match the key condition. A *scan* reads every item in the table and applies any filter afterward. Prefer key-based queries, and reserve scans for access patterns that a key cannot express. For more information, see Querying tables in DynamoDB in the Amazon DynamoDB Developer Guide.

The examples on this page use the composite-key table created in Create a table with a composite primary key. The `client` that is used in the following examples can be created from the snippet in Set up the examples. Attributes in the returned items are typed `AttributeValue` unions; narrow each one to its concrete model before reading its value, as shown in Get an item.

Query by partition key

For request and response details, see the query() API reference.

Use `query` when you know the partition-key value. The key condition must include equality on the partition key and can also constrain the sort key.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import AttributeValue, AttributeValueS, QueryInput
```

Code

```
async def query_items(
    client: AsyncDynamoDBClient, table_name: str, author: str
) -> list[dict[str, AttributeValue]]:
    """Query one response page by partition-key value."""
    response = await client.query(
        input=QueryInput(
            table_name=table_name,
            key_condition_expression="#author = :author",
            expression_attribute_names={"#author": "author"},
            expression_attribute_values={
                ":author": AttributeValueS(value=author)
            },
            consistent_read=True,
        )
    )
    return response.items or []
```

Paginate a query manually

For request and response details, see the query() API reference.

A query can return a partial result. Continue while `last_evaluated_key` is present, and pass that key as `exclusive_start_key` on the next request.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import AttributeValue, AttributeValueS, QueryInput
```

Code

```
async def query_items_paginated(
    client: AsyncDynamoDBClient,
    table_name: str,
    author: str,
    max_pages: int = 100,
) -> list[dict[str, AttributeValue]]:
    """Query all pages, bounded by ``max_pages``."""
    items: list[dict[str, AttributeValue]] = []
    start_key = None
    for _ in range(max_pages):
        page = await client.query(
            input=QueryInput(
                table_name=table_name,
                key_condition_expression="#author = :author",
                expression_attribute_names={"#author": "author"},
                expression_attribute_values={
                    ":author": AttributeValueS(value=author)
                },
                exclusive_start_key=start_key,
            )
        )
        items.extend(page.items or [])
        start_key = page.last_evaluated_key
        if not start_key:
            return items
    raise RuntimeError("DynamoDB query exceeded the page limit")
```

Scan with a filter

For request and response details, see the scan() API reference.

Use `scan` only when a key-based query cannot express the access pattern. A filter is applied after DynamoDB reads each page, so it does not reduce the read capacity consumed. Continue with `last_evaluated_key` even when a filtered page returns no items.

Imports

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import AttributeValue, AttributeValueN, ScanInput
```

Code

```
async def scan_items(
    client: AsyncDynamoDBClient,
    table_name: str,
    minimum_year: int,
    max_pages: int = 100,
) -> list[dict[str, AttributeValue]]:
    """Scan filtered pages, bounded by ``max_pages``."""
    items: list[dict[str, AttributeValue]] = []
    start_key = None
    for _ in range(max_pages):
        page = await client.scan(
            input=ScanInput(
                table_name=table_name,
                filter_expression="#year >= :minimum",
                expression_attribute_names={"#year": "year"},
                expression_attribute_values={
                    ":minimum": AttributeValueN(value=str(minimum_year))
                },
                exclusive_start_key=start_key,
            )
        )
        items.extend(page.items or [])
        start_key = page.last_evaluated_key
        if not start_key:
            return items
    raise RuntimeError("DynamoDB scan exceeded the page limit")
```

More information

- Querying tables in DynamoDB in the Amazon DynamoDB Developer Guide
- Scanning tables in DynamoDB in the Amazon DynamoDB Developer Guide
- Query in the Amazon DynamoDB API Reference
- Scan in the Amazon DynamoDB API Reference

Use DynamoDB batch operations and transactions

DynamoDB provides two ways to group item writes and reads. A *batch* operation sends up to 25 writes or 100 reads in one request; each item succeeds or fails independently, so your application must resubmit unprocessed entries. A *transaction* applies up to 100 write actions as a single all-

or-nothing unit. For more information, see DynamoDB transactions in the Amazon DynamoDB Developer Guide.

The examples on this page use the composite-key table created in Create a table with a composite primary key. The `client` that is used in the following examples can be created from the snippet in Set up the examples.

Write items in a batch

For request and response details, see the batch_write_item() API reference.

A successful `batch_write_item` response can still contain unprocessed writes. Resubmit only those writes, cap the number of attempts, and delay between attempts.

Imports

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeValueS,
    BatchWriteItemInput,
    PutRequest,
    WriteRequest,
)
```

Code

```
async def batch_write_items(
    client: AsyncDynamoDBClient,
    table_name: str,
    max_attempts: int = 5,
) -> None:
    """Write items with bounded retries for unprocessed writes."""
    pending = {
        table_name: [
            WriteRequest(
                put_request=PutRequest(
                    item={
                        "author": AttributeValueS(value="Grace Hopper"),
                        "title": AttributeValueS(
                            value="The Education of a Computer"
```

```
                        ),
                    }
                )
            ),
            WriteRequest(
                put_request=PutRequest(
                    item={
                        "author": AttributeValueS(value="Edsger Dijkstra"),
                        "title": AttributeValueS(
                            value="Go To Statement Considered Harmful"
                        ),
                    }
                )
            ),
        ]
    }
    for attempt in range(max_attempts):
        response = await client.batch_write_item(
            input=BatchWriteItemInput(request_items=pending)
        )
        pending = response.unprocessed_items or {}
        if not pending:
            return
        await asyncio.sleep(0.2 * (2**attempt))
    raise RuntimeError("DynamoDB did not process every batch write")
```

Get items in a batch

For request and response details, see the batch_get_item() API reference.

`batch_get_item` can also return unprocessed keys. Collect each response page, resubmit only those keys with a bounded delay, and identify returned items by key because DynamoDB does not guarantee their order.

Imports

```
import asyncio

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeValue,
    AttributeValueS,
    BatchGetItemInput,
```

```
    KeysAndAttributes,
)
```

Code

```
async def batch_get_items(
    client: AsyncDynamoDBClient,
    table_name: str,
    max_attempts: int = 5,
) -> list[dict[str, AttributeValue]]:
    """Get items with bounded retries for unprocessed keys."""
    pending = {
        table_name: KeysAndAttributes(
            keys=[
                {
                    "author": AttributeValueS(value="Grace Hopper"),
                    "title": AttributeValueS(value="The Education of a Computer"),
                },
                {
                    "author": AttributeValueS(value="Edsger Dijkstra"),
                    "title": AttributeValueS(
                        value="Go To Statement Considered Harmful"
                    ),
                },
            ],
            consistent_read=True,
        )
    }
    items: list[dict[str, AttributeValue]] = []
    for attempt in range(max_attempts):
        response = await client.batch_get_item(
            input=BatchGetItemInput(request_items=pending)
        )
        items.extend((response.responses or {}).get(table_name, []))
        pending = response.unprocessed_keys or {}
        if not pending:
            return items
        await asyncio.sleep(0.2 * (2**attempt))
    raise RuntimeError("DynamoDB did not process every batch key")
```

Write items in an idempotent transaction

For request and response details, see the transact_write_items() API reference.

`transact_write_items` applies all actions as one unit. Generate the client request token once, and reuse that same token only when retrying the identical transaction. This keeps an identical retry idempotent during DynamoDB's token window.

Imports

```
import uuid

from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.models import (
    AttributeValueS,
    Put,
    TransactWriteItem,
    TransactWriteItemsInput,
)
```

Code

```
async def transact_write_items(
    client: AsyncDynamoDBClient,
    table_name: str,
    request_token: str | None = None,
) -> None:
    """Atomically put two books, using one token for identical retries."""
    request = TransactWriteItemsInput(
        transact_items=[
            TransactWriteItem(
                put=Put(
                    table_name=table_name,
                    item={
                        "author": AttributeValueS(value="Alan Turing"),
                        "title": AttributeValueS(value="On Computable Numbers"),
                    },
                    condition_expression="attribute_not_exists(#author)",
                    expression_attribute_names={"#author": "author"},
                )
            ),
            TransactWriteItem(
                put=Put(
                    table_name=table_name,
                    item={
                        "author": AttributeValueS(value="Katherine Johnson"),
                        "title": AttributeValueS(value="Orbital Mechanics Notes"),
```

```
                    },
                    condition_expression="attribute_not_exists(#author)",
                    expression_attribute_names={"#author": "author"},
                )
            ),
        ],
        client_request_token=request_token or str(uuid.uuid4()),
    )
    await client.transact_write_items(input=request)
```

More information

- DynamoDB transactions in the Amazon DynamoDB Developer Guide
- BatchWriteItem in the Amazon DynamoDB API Reference
- BatchGetItem in the Amazon DynamoDB API Reference
- TransactWriteItems in the Amazon DynamoDB API Reference

Amazon SQS

This chapter provides runnable examples for common Amazon Simple Queue Service (Amazon SQS) workflows — creating and managing queues, sending and receiving messages, and configuring long polling to reduce empty responses and lower costs. Each example uses the SDK's asynchronous SQS client with typed inputs and outputs, so you can adapt them to your application without assembling raw request parameters. Start here to install the client package and configure shared setup that the remaining pages build on.

Install the SQS client before running the asynchronous examples on this page:

```
python -m pip install aws-sdk-sqs
```

Set up the examples

The `client` that is used in the following examples can be created from the following snippet by calling `client = await create_client()` from your asynchronous code. Use the client as an asynchronous context manager, or call `await client.close()` after the last operation, as described in Creating a service client. When you resolve the generated asynchronous configuration with `resolve()`, the SDK reads the AWS Region from standard AWS settings, such as environment variables and shared AWS configuration files. The SDK also reads credentials from the same

settings. Configure both before running the examples: if no Region is configured, `resolve()` fails before the client is created, and if no credentials are configured, requests fail when the SDK signs them. For more information about the Region and other settings, see Configuration variables. For how the SDK finds credentials, see Credential providers.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.config import AsyncSQSConfig
```

Code

```
async def create_client() -> AsyncSQSClient:
    config = await AsyncSQSConfig.resolve()
    return AsyncSQSClient(config=config)
```

Topics

- Work with Amazon SQS queues
- Work with Amazon SQS messages
- Configure long polling for Amazon SQS

Work with Amazon SQS queues

A *message queue* is the logical container used for sending messages reliably in Amazon Simple Queue Service. There are two types of queues: *standard* and *first-in, first-out* (FIFO). To learn more about queues and the differences between these types, see the Amazon Simple Queue Service Developer Guide.

The `client` that is used in the following examples can be created from the snippet in Set up the examples.

Create a queue

For request and response details, see the create_queue() API reference.

Create a queue with `create_queue`. The response returns the queue URL, which SQS data-plane operations use. To configure long polling when creating a queue, see Configure long polling for Amazon SQS.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import CreateQueueInput
```

Code

```
async def create_queue(client: AsyncSQSClient, queue_name: str) -> str:
    """Create a queue and return its URL."""
    response = await client.create_queue(
        input=CreateQueueInput(queue_name=queue_name)
    )
    if not response.queue_url:
        raise RuntimeError("SQS did not return a queue URL")
    return response.queue_url
```

List queues

For request and response details, see the list_queues() API reference.

Use `list_queues` to discover queues by name prefix. Continue with `next_token` until the service returns no token.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import ListQueuesInput
```

Code

```
async def list_queues(client: AsyncSQSClient, prefix: str) -> list[str]:
    """Return every queue URL whose queue name starts with the prefix."""
    queue_urls: list[str] = []
    next_token = None
    while True:
        page = await client.list_queues(
            input=ListQueuesInput(
                queue_name_prefix=prefix,
                next_token=next_token,
            )
        )
        queue_urls.extend(page.queue_urls or [])
```

```
        next_token = page.next_token
        if not next_token:
            return queue_urls
```

Get a queue URL

For request and response details, see the get_queue_url() API reference.

If you have a queue name instead of its URL, resolve it with `get_queue_url`.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import GetQueueUrlInput
```

Code

```
async def get_queue_url(client: AsyncSQSClient, queue_name: str) -> str:
    """Return the URL of an existing queue."""
    response = await client.get_queue_url(
        input=GetQueueUrlInput(queue_name=queue_name)
    )
    if not response.queue_url:
        raise RuntimeError("SQS did not return a queue URL")
    return response.queue_url
```

Get a queue ARN

For request and response details, see the get_queue_attributes() API reference.

SQS data-plane operations use the queue URL. Resource policies and service integrations use the same queue's ARN. Retrieve it with `get_queue_attributes`.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import GetQueueAttributesInput
```

Code

```
async def get_queue_arn(client: AsyncSQSClient, queue_url: str) -> str:
    """Return the ARN of an existing queue."""
```

```
    response = await client.get_queue_attributes(
        input=GetQueueAttributesInput(
            queue_url=queue_url,
            attribute_names=["QueueArn"],
        )
    )
    queue_arn = (response.attributes or {}).get("QueueArn")
    if not queue_arn:
        raise RuntimeError("SQS did not return the queue ARN")
    return queue_arn
```

Delete a queue

For request and response details, see the delete_queue() API reference.

Delete a temporary queue by URL after its messages and integrations are no longer needed.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import DeleteQueueInput
```

Code

```
async def delete_queue(client: AsyncSQSClient, queue_url: str) -> None:
    """Delete the queue at the specified URL."""
    await client.delete_queue(input=DeleteQueueInput(queue_url=queue_url))
```

More information

- CreateQueue in the Amazon Simple Queue Service API Reference
- ListQueues in the Amazon Simple Queue Service API Reference
- GetQueueUrl in the Amazon Simple Queue Service API Reference
- GetQueueAttributes in the Amazon Simple Queue Service API Reference
- DeleteQueue in the Amazon Simple Queue Service API Reference

Work with Amazon SQS messages

A *message* is a piece of data that can be sent and received by distributed components. Messages are always delivered using an SQS queue.

The `client` that is used in the following examples can be created from the snippet in Set up the examples.

Send a message

For request and response details, see the send_message() API reference.

Send a message to a queue URL with `send_message`. Message attributes carry structured metadata separately from the body.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import MessageAttributeValue, SendMessageInput
```

Code

```
async def send_message(
    client: AsyncSQSClient, queue_url: str, message_body: str
) -> str:
    """Send one message and return its service-assigned ID."""
    response = await client.send_message(
        input=SendMessageInput(
            queue_url=queue_url,
            message_body=message_body,
            message_attributes={
                "event_type": MessageAttributeValue(
                    data_type="String", string_value="order-created"
                ),
                "priority": MessageAttributeValue(
                    data_type="Number", string_value="7"
                ),
            },
        )
    )
    if not response.message_id:
        raise RuntimeError("SQS did not return a message ID")
    return response.message_id
```

Receive messages

For request and response details, see the receive_message() API reference.

Receive up to 10 messages with `receive_message`. An empty response is valid, so poll with a bounded attempt count or an application shutdown condition. Message attributes are returned only when the request names them; this example requests all of them. To wait for messages instead of returning immediately, see Configure long polling for Amazon SQS.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import Message, ReceiveMessageInput
```

Code

```
async def receive_messages(
    client: AsyncSQSClient, queue_url: str, attempts: int = 3
) -> list[Message]:
    """Receive up to ten messages, stopping after the first nonempty response."""
    messages: list[Message] = []
    for _ in range(attempts):
        response = await client.receive_message(
            input=ReceiveMessageInput(
                queue_url=queue_url,
                max_number_of_messages=10,
                message_attribute_names=["All"],
            )
        )
        messages = response.messages or []
        if messages:
            break
    return messages
```

Delete a message

For request and response details, see the delete_message() API reference.

Receiving a message hides it for the visibility timeout but does not remove it. Process the body first, and call this function with the delivery's current receipt handle only after processing succeeds.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import DeleteMessageInput
```

Code

```
async def delete_message(
    client: AsyncSQSClient, queue_url: str, receipt_handle: str
) -> None:
    """Delete one message delivery after successful processing."""
    await client.delete_message(
        input=DeleteMessageInput(
            queue_url=queue_url,
            receipt_handle=receipt_handle,
        )
    )
```

Send and delete messages in batches

For request and response details, see the send_message_batch() and delete_message_batch() API references.

Use `send_message_batch` to send up to 10 entries to one queue. Entry IDs must be unique within the request. A successful request can contain per-entry failures, so associate failures with their IDs and retry only appropriate failed entries.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import SendMessageBatchInput, SendMessageBatchRequestEntry
```

Code

```
async def send_message_batch(
    client: AsyncSQSClient, queue_url: str, message_bodies: list[str]
) -> None:
    """Send a message batch and raise if any entry fails."""
    if not 1 <= len(message_bodies) <= 10:
        raise ValueError("A message batch must contain between 1 and 10 entries")
    response = await client.send_message_batch(
        input=SendMessageBatchInput(
            queue_url=queue_url,
            entries=[
                SendMessageBatchRequestEntry(id=f"message-{index}", message_body=body)
                for index, body in enumerate(message_bodies)
```

```
            ],
        )
    )
    if response.failed:
        details = ", ".join(
            f"{failure.id}: {failure.code}" for failure in response.failed
        )
        raise RuntimeError(f"SQS batch send failed: {details}")
```

After each message has been processed successfully, use `delete_message_batch` with its receipt handle. Inspect per-entry failures just as you do for a batch send.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import DeleteMessageBatchInput, DeleteMessageBatchRequestEntry
```

Code

```
async def delete_message_batch(
    client: AsyncSQSClient, queue_url: str, receipt_handles: list[str]
) -> None:
    """Delete a message batch and raise if any entry fails."""
    if not 1 <= len(receipt_handles) <= 10:
        raise ValueError("A delete batch must contain between 1 and 10 entries")
    response = await client.delete_message_batch(
        input=DeleteMessageBatchInput(
            queue_url=queue_url,
            entries=[
                DeleteMessageBatchRequestEntry(
                    id=f"delete-{index}", receipt_handle=receipt_handle
                )
                for index, receipt_handle in enumerate(receipt_handles)
            ],
        )
    )
    if response.failed:
        details = ", ".join(
            f"{failure.id}: {failure.code}" for failure in response.failed
        )
        raise RuntimeError(f"SQS batch delete failed: {details}")
```

More information

- Message metadata for Amazon SQS in the Amazon Simple Queue Service Developer Guide
- SendMessage in the Amazon Simple Queue Service API Reference
- ReceiveMessage in the Amazon Simple Queue Service API Reference
- DeleteMessage in the Amazon Simple Queue Service API Reference
- SendMessageBatch in the Amazon Simple Queue Service API Reference
- DeleteMessageBatch in the Amazon Simple Queue Service API Reference

Configure long polling for Amazon SQS

Long polling waits for a message to become available instead of returning immediately. Configure it at two levels: set `ReceiveMessageWaitTimeSeconds` as the queue default, or set `wait_time_seconds` on an individual `receive_message` request. A request-level value overrides the queue default.

The `client` that is used in the following examples can be created from the snippet in Set up the examples.

Queue attribute values are strings. Valid values for `ReceiveMessageWaitTimeSeconds` are `"0"` through `"20"`. Values greater than `"0"` enable long polling, and `"0"` uses short polling. The corresponding request-level values are integers from `0` through `20`.

Enable long polling when creating a queue

For request and response details, see the create_queue() API reference.

Set `ReceiveMessageWaitTimeSeconds` in the queue attributes to establish the default when the queue is created.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import CreateQueueInput
```

Code

```
async def create_long_polling_queue(
    client: AsyncSQSClient, queue_name: str, wait_time_seconds: int = 20
```

```
) -> str:
    """Create a queue with a default long-poll wait and return its URL."""
    if not 0 <= wait_time_seconds <= 20:
        raise ValueError("The long-poll wait must be between 0 and 20 seconds")
    response = await client.create_queue(
        input=CreateQueueInput(
            queue_name=queue_name,
            attributes={
                "ReceiveMessageWaitTimeSeconds": str(wait_time_seconds)
            },
        )
    )
    if not response.queue_url:
        raise RuntimeError("SQS did not return a queue URL")
    return response.queue_url
```

Enable long polling on an existing queue

For request and response details, see the set_queue_attributes() API reference.

Use `set_queue_attributes` to change the default for an existing queue. Queue attribute changes can take up to 60 seconds to propagate.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import SetQueueAttributesInput
```

Code

```
async def set_long_polling(
    client: AsyncSQSClient, queue_url: str, wait_time_seconds: int = 20
) -> None:
    """Set the queue's default receive wait time."""
    if not 0 <= wait_time_seconds <= 20:
        raise ValueError("The long-poll wait must be between 0 and 20 seconds")
    await client.set_queue_attributes(
        input=SetQueueAttributesInput(
            queue_url=queue_url,
            attributes={"ReceiveMessageWaitTimeSeconds": str(wait_time_seconds)},
        )
    )
```

Enable long polling for a receive request

For request and response details, see the receive_message() API reference.

Set `wait_time_seconds` on `receive_message` to configure one request. Ensure that the client's response timeout is longer than the wait time. An empty response is still valid when the wait time expires.

Imports

```
from aws_sdk_sqs.client import AsyncSQSClient
from aws_sdk_sqs.models import Message, ReceiveMessageInput
```

Code

```
async def receive_with_long_polling(
    client: AsyncSQSClient, queue_url: str, wait_time_seconds: int = 20
) -> list[Message]:
    """Receive up to ten messages using a request-specific wait time."""
    if not 0 <= wait_time_seconds <= 20:
        raise ValueError("The long-poll wait must be between 0 and 20 seconds")
    response = await client.receive_message(
        input=ReceiveMessageInput(
            queue_url=queue_url,
            max_number_of_messages=10,
            wait_time_seconds=wait_time_seconds,
        )
    )
    return response.messages or []
```

More information

- Amazon SQS short and long polling in the Amazon Simple Queue Service Developer Guide
- SetQueueAttributes in the Amazon Simple Queue Service API Reference
- ReceiveMessage in the Amazon Simple Queue Service API Reference

Amazon SNS

This page provides runnable examples for common Amazon Simple Notification Service (Amazon SNS) workflows — creating topics, subscribing endpoints across protocols, publishing messages

with metadata attributes, and managing subscription and topic lifecycle. Each example uses the SDK's asynchronous SNS client with typed inputs and outputs, so you can adapt them to your application without assembling raw request parameters. Start below to install the client package and configure shared setup that the remaining examples build on.

Install the SNS client before running the asynchronous examples on this page:

```
python -m pip install aws-sdk-sns
```

Set up the examples

The `client` that is used in the following examples can be created from the following snippet by calling `client = await create_client()` from your asynchronous code. Use the client as an asynchronous context manager, or call `await client.close()` after the last operation, as described in Creating a service client. When you resolve the generated asynchronous configuration with `resolve()`, the SDK reads the AWS Region from standard AWS settings, such as environment variables and shared AWS configuration files. The SDK also reads credentials from the same settings. Configure both before running the examples: if no Region is configured, `resolve()` fails before the client is created, and if no credentials are configured, requests fail when the SDK signs them. For more information about the Region and other settings, see Configuration variables. For how the SDK finds credentials, see Credential providers.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.config import AsyncSNSConfig
```

Code

```
async def create_client() -> AsyncSNSClient:
    config = await AsyncSNSConfig.resolve()
    return AsyncSNSClient(config=config)
```

Create a topic

For request and response details, see the create_topic() API reference.

A *topic* is a logical grouping of communication channels that defines which systems to send a message to, for example, fanning out a message to AWS Lambda and an HTTP webhook. You send

messages to Amazon SNS, then they're distributed to the channels defined in the topic. This makes the messages available to subscribers.

Create a topic with `create_topic` and retain the returned ARN. Later operations identify the topic by this ARN.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.models import CreateTopicInput
```

Code

```
async def create_topic(client: AsyncSNSClient, topic_name: str) -> str:
    """Create a topic and return its ARN."""
    response = await client.create_topic(
        input=CreateTopicInput(name=topic_name)
    )
    if not response.topic_arn:
        raise RuntimeError("SNS did not return a topic ARN")
    return response.topic_arn
```

List topics

For request and response details, see the list_topics() API reference.

`list_topics` can return a partial result. Continue with `next_token` until Amazon SNS returns no continuation token.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.models import ListTopicsInput
```

Code

```
async def list_topics(client: AsyncSNSClient) -> list[str]:
    """Return all topic ARNs, following pagination tokens."""
    topic_arns: list[str] = []
    next_token: str | None = None
```

```
    while True:
        page = await client.list_topics(
            input=ListTopicsInput(next_token=next_token)
        )
        topic_arns.extend(
            topic.topic_arn
            for topic in page.topics or []
            if topic.topic_arn
        )
        next_token = page.next_token
        if not next_token:
            return topic_arns
```

Subscribe an endpoint to a topic

For request and response details, see the subscribe() API reference.

After you create a topic, you can configure which communication channels will be endpoints for that topic. Messages are distributed to these endpoints after Amazon SNS receives them.

Subscribe an endpoint by specifying its protocol and address. This example subscribes an email address that you control. Email subscriptions do not receive publications until the recipient confirms the subscription.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.models import SubscribeInput
```

Code

```
async def subscribe_email(
    client: AsyncSNSClient, topic_arn: str, email: str
) -> str:
    """Request an email subscription and return its ARN or status."""
    response = await client.subscribe(
        input=SubscribeInput(
            topic_arn=topic_arn,
            protocol="email",
            endpoint=email,
            return_subscription_arn=True,
```

```
        )
    )
    if not response.subscription_arn:
        raise RuntimeError("SNS did not return a subscription status")
    return response.subscription_arn
```

Different protocols have different endpoint formats and confirmation behavior. Do not publish sensitive data until the intended endpoint has confirmed the subscription.

Publish a message to a topic

For request and response details, see the publish() API reference.

After you have a topic and one or more endpoints configured for it, you can publish a message to it.

Publish to a topic ARN after its intended endpoints are confirmed. Message attributes carry typed metadata separately from the body.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.models import MessageAttributeValue, PublishInput
```

Code

```
async def publish_to_topic(
    client: AsyncSNSClient, topic_arn: str, message: str
) -> str:
    """Publish a message and return its message ID."""
    response = await client.publish(
        input=PublishInput(
            topic_arn=topic_arn,
            message=message,
            message_attributes={
                "event_type": MessageAttributeValue(
                    data_type="String",
                    string_value="order-created",
                )
            },
        )
```

```
    )
    if not response.message_id:
        raise RuntimeError("SNS did not return a message ID")
    return response.message_id
```

Unsubscribe an endpoint from a topic

For request and response details, see the unsubscribe() API reference.

You can remove the communication channels configured as endpoints for a topic. After doing that, the topic itself continues to exist and distributes messages to any other endpoints configured for that topic.

Use `unsubscribe` with the subscription ARN returned by `subscribe`. The subscription must be confirmed first: because the examples request `return_subscription_arn=True`, `subscribe` returns a real ARN even before the endpoint confirms, but unsubscribing a subscription that is still pending confirmation fails with the modeled `InvalidParameterException`.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.models import UnsubscribeInput
```

Code

```
async def unsubscribe(client: AsyncSNSClient, subscription_arn: str) -> None:
    """Unsubscribe an endpoint identified by its subscription ARN."""
    await client.unsubscribe(
        input=UnsubscribeInput(subscription_arn=subscription_arn)
    )
```

Delete a topic

For request and response details, see the delete_topic() API reference.

Delete a temporary topic when it is no longer needed. Deleting a topic also deletes its subscriptions.

Imports

```
from aws_sdk_sns.client import AsyncSNSClient
from aws_sdk_sns.models import DeleteTopicInput
```

Code

```
async def delete_topic(client: AsyncSNSClient, topic_arn: str) -> None:
    """Delete the topic identified by its ARN."""
    await client.delete_topic(input=DeleteTopicInput(topic_arn=topic_arn))
```

More information

- Amazon SNS Developer Guide
- CreateTopic in the Amazon Simple Notification Service API Reference
- ListTopics in the Amazon Simple Notification Service API Reference
- Subscribe in the Amazon Simple Notification Service API Reference
- Publish in the Amazon Simple Notification Service API Reference
- Unsubscribe in the Amazon Simple Notification Service API Reference
- DeleteTopic in the Amazon Simple Notification Service API Reference

Amazon Bedrock Runtime

This chapter provides runnable examples for common Amazon Bedrock Runtime inference patterns — sending conversational messages, streaming model output as it generates, invoking models with native request formats, and exchanging real-time bidirectional events. Each example uses the SDK's asynchronous Bedrock Runtime client with typed inputs and outputs, so you can adapt them to your application without constructing model-specific JSON payloads by hand. Start here to install the client package, configure shared setup, and choose the right inference operation for your use case.

Set up the examples

Install the Bedrock Runtime client package.

```
python -m pip install aws-sdk-bedrock-runtime
```

Each example in this chapter is a complete program: its Imports, helper functions, operation code, and `main` entry point together form the full example file. Each program pins the AWS Region to `us-east-1` when it resolves the client configuration, because that Region offers the models that these examples use. Model availability varies by Region, so if you switch to a different model, also set the Region to one that offers that model. The SDK reads credentials from standard AWS settings, such as environment variables and shared AWS configuration files. Configure credentials before running the examples. If none are configured, requests fail when the SDK signs them. For how the SDK finds credentials, see Credential providers.

Running the examples sends inference requests that incur charges. For details, see Amazon Bedrock pricing.

Choose an inference operation

Choose an API family first, then choose whether the application needs a complete response or incremental output.

- **Portable conversation format:** Use `converse` to get the model's complete reply in a single response. Use `converse_stream` to receive that reply as a stream of events while the model is still generating it, so your application can display text as it arrives. Both use Bedrock message and content-block models.
- **Model-native format:** Use `invoke_model` to get the model's complete native response in a single call. Use `invoke_model_with_response_stream` to receive that native output as a stream of events while the model is still generating it. The request, response, and event schemas depend on the selected model.
- **Real-time bidirectional format:** Use `invoke_model_with_bidirectional_stream` with a compatible model when the application must continue sending input while it receives output. The event protocol depends on the selected model.

Operation support varies by model. For each model's supported operations and capabilities, see Models at a glance in the Amazon Bedrock User Guide and select a model to view its details.

Topics

- Use Converse
- Use ConverseStream
- Use InvokeModel

- Use InvokeModelWithResponseStream
- Use bidirectional streaming

Use Converse

Converse is the portable conversation API for Amazon Bedrock Runtime. A conversation is a list of *messages*, each with a role and a list of typed *content blocks*. Use it when you want one request format that works across supported models, so you can switch models without rewriting request code.

The following example runs a complete two-turn conversation. For request and response details, see the converse() API reference.

Imports

```
import asyncio

from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from aws_sdk_bedrock_runtime.models import (
    ContentBlockText,
    ConverseInput,
    ConverseOutputMessage,
    InferenceConfiguration,
    Message,
)
```

The generated `output` member is a union. Narrow it to `ConverseOutputMessage` before using `.value`. Each message content block is also a union, so narrow text blocks before reading their values. The following helper functions fail when the response carries no message and when it contains no text content.

```
def narrow_message(response: object) -> Message:
    """Narrow a Converse output union and return its message."""
    output = getattr(response, "output", None)
    if not isinstance(output, ConverseOutputMessage):
        raise RuntimeError(f"Unexpected Converse output: {type(output).__name__}")
    return output.value
```

```
def message_text(message: Message) -> str:
    """Collect text only after narrowing every content-block union."""
    text = "".join(
        block.value for block in message.content if isinstance(block, ContentBlockText)
    )
    if not text:
        raise RuntimeError("The model returned no text content")
    return text
```

Create a user `Message` with a text content-block union and pass it to `converse`. Common generation settings belong in `InferenceConfiguration`. Bedrock Runtime does not retain conversation history between calls, so for the second turn the code appends the narrowed assistant message and the next user message, then sends the complete history.

The code also prints the response's `stop_reason` and `usage` fields. The `stop_reason` field reports why the model stopped generating. For example, `end_turn` means the model finished its reply normally, and `max_tokens` means the reply was cut off at the `max_tokens` limit. The `usage` field reports the input and output token counts that inference is billed on.

Code

```
async def converse(client: AsyncBedrockRuntimeClient) -> tuple[str, str]:
    """Run a complete two-turn conversation and return both assistant replies."""
    model_id = "global.amazon.nova-2-lite-v1:0"
    prompt = "Name one moon of Saturn."

    messages = [Message(role="user", content=[ContentBlockText(value=prompt)])]
    response = await client.converse(
        ConverseInput(
            model_id=model_id,
            messages=messages,
            inference_config=InferenceConfiguration(
                max_tokens=100,
                temperature=0.2,
                top_p=0.9,
            ),
        )
    )

    assistant_message = narrow_message(response)
    first_text = message_text(assistant_message)
    print(first_text)
```

```
    print(f"stop reason: {response.stop_reason}")
    print(
        f"tokens: {response.usage.input_tokens} input, "
        f"{response.usage.output_tokens} output"
    )

    # Bedrock does not retain history, so include every preceding message.
    follow_up_prompt = "Name another one."

    messages.extend(
        [
            assistant_message,
            Message(
                role="user",
                content=[ContentBlockText(value=follow_up_prompt)],
            ),
        ]
    )
    second_response = await client.converse(
        ConverseInput(model_id=model_id, messages=messages)
    )
    second_text = message_text(narrow_message(second_response))
    print(second_text)
    return first_text, second_text
```

The entry point creates the client and runs the conversation. Run the program with `python converse.py`.

```
async def main() -> None:
    """Create a client in a Region that offers the model and run the example."""
    config = await AsyncBedrockRuntimeConfig.resolve(region="us-east-1")
    async with AsyncBedrockRuntimeClient(config=config) as client:
        await converse(client)


if __name__ == "__main__":
    asyncio.run(main())
```

More information

- Use the Converse API in the Amazon Bedrock User Guide
- Converse in the Amazon Bedrock API Reference

Use ConverseStream

ConverseStream accepts the same request format as Converse but returns the response as a stream of events, so your application can process output as the model generates it instead of waiting for the complete response. Use it for interactive applications, such as chat, where users should start seeing text as soon as the model begins its reply.

The following example sends one request, prints the reply as the model generates it, and then verifies that the stream completed. For request and response details, see the converse_stream() API reference.

Imports

```
import asyncio
from typing import Any

from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from aws_sdk_bedrock_runtime.models import (
    ContentBlockDeltaText,
    ContentBlockText,
    ConverseStreamInput,
    ConverseStreamOutputContentBlockDelta,
    ConverseStreamOutputContentBlockStart,
    ConverseStreamOutputContentBlockStop,
    ConverseStreamOutputInternalServerException,
    ConverseStreamOutputMessageStart,
    ConverseStreamOutputMessageStop,
    ConverseStreamOutputMetadata,
    ConverseStreamOutputModelStreamErrorException,
    ConverseStreamOutputServiceUnavailableException,
    ConverseStreamOutputThrottlingException,
    ConverseStreamOutputUnknown,
    ConverseStreamOutputValidationException,
    Message,
    StopReason,
    TokenUsage,
)
```

Group the modeled stream-error events so that the event loop can treat every failure variant the same way.

```
STREAM_ERRORS = (
    ConverseStreamOutputInternalServerException,
    ConverseStreamOutputModelStreamErrorException,
    ConverseStreamOutputServiceUnavailableException,
    ConverseStreamOutputThrottlingException,
    ConverseStreamOutputValidationException,
)
```

`converse_stream` returns the response as a stream of events over an open network connection. The following function reads the events inside an `async with` block, which closes the connection when the stream ends or an error interrupts it. It narrows every generated event union, collects the text deltas, and raises on `STREAM_ERRORS` events and unknown variants instead of accepting partial output. The message-stop event carries the stop reason, and the metadata event carries the token usage. The function requires both events before it accepts the accumulated text as complete.

```
async def consume_stream(response: Any) -> tuple[str, StopReason, TokenUsage]:
    """Consume a Converse stream and reject errors or incomplete output."""
    text_parts: list[str] = []
    stop_reason: StopReason | None = None
    usage: TokenUsage | None = None

    async with response as stream:
        async for event in stream.output_stream:
            if isinstance(event, ConverseStreamOutputMessageStart):
                print(f"assistant role: {event.value.role}")
            elif isinstance(event, ConverseStreamOutputContentBlockStart):
                print(f"content block {event.value.content_block_index} started")
            elif isinstance(event, ConverseStreamOutputContentBlockDelta):
                delta = event.value.delta
                if isinstance(delta, ContentBlockDeltaText):
                    text_parts.append(delta.value)
                    print(delta.value, end="", flush=True)
            elif isinstance(event, ConverseStreamOutputContentBlockStop):
                print(f"\ncontent block {event.value.content_block_index} stopped")
            elif isinstance(event, ConverseStreamOutputMessageStop):
                stop_reason = event.value.stop_reason
            elif isinstance(event, ConverseStreamOutputMetadata):
                usage = event.value.usage
            elif isinstance(event, STREAM_ERRORS):
                raise RuntimeError(event.value.message or type(event).__name__)
            elif isinstance(event, ConverseStreamOutputUnknown):
                raise RuntimeError(f"Unknown stream event: {event.tag}")
```

```
            else:
                raise RuntimeError(f"Unexpected stream event: {type(event).__name__}")

    if not text_parts or stop_reason is None or usage is None:
        raise RuntimeError("The stream ended without all expected events")
    return "".join(text_parts), stop_reason, usage
```

The following function sends one complete Converse request and hands the response to `consume_stream`.

Code

```
async def converse_stream(
    client: AsyncBedrockRuntimeClient,
) -> tuple[str, StopReason, TokenUsage]:
    """Send a ConverseStream request and consume its complete response."""
    model_id = "global.amazon.nova-2-lite-v1:0"
    prompt = "Name one moon of Saturn."

    response = await client.converse_stream(
        ConverseStreamInput(
            model_id=model_id,
            messages=[
                Message(
                    role="user",
                    content=[ContentBlockText(value=prompt)],
                )
            ],
        )
    )
    text, stop_reason, usage = await consume_stream(response)
    print(f"\nstop reason: {stop_reason}")
    print(f"tokens: {usage.input_tokens} input, {usage.output_tokens} output")
    return text, stop_reason, usage
```

The entry point creates the client and runs the streaming request. Run the program with `python converse_stream.py`.

```
async def main() -> None:
    """Create a client in a Region that offers the model and run the example."""
    config = await AsyncBedrockRuntimeConfig.resolve(region="us-east-1")
    async with AsyncBedrockRuntimeClient(config=config) as client:
```

```
        await converse_stream(client)


if __name__ == "__main__":
    asyncio.run(main())
```

More information

- Use the Converse API in the Amazon Bedrock User Guide
- ConverseStream in the Amazon Bedrock API Reference

Use InvokeModel

InvokeModel sends a request in the selected model's native JSON format and returns the model's complete native response. Use it when you need model-specific request or response features that the portable Converse format doesn't expose. The request and response schemas depend on the selected model.

The following example sends one Amazon Nova 2 Messages API request and prints the reply text. For request and response details, see the invoke_model() API reference.

Imports

```
import asyncio
import json
from typing import Any

from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from aws_sdk_bedrock_runtime.models import InvokeModelInput
```

The following helper functions build the model-native request and validate the model-native response.

```
def build_native_request(prompt: str) -> dict[str, Any]:
    """Build an Amazon Nova 2 Messages API request."""
    return {
        "messages": [{"role": "user", "content": [{"text": prompt}]}],
        "inferenceConfig": {
            "maxTokens": 100,
```

```
            "temperature": 0.2,
            "topP": 0.9,
        },
    }


def parse_native_text(body: bytes) -> str:
    """Validate an Amazon Nova 2 response and collect its text blocks."""
    payload: Any = json.loads(body.decode("utf-8"))
    if not isinstance(payload, dict):
        raise RuntimeError("The model returned a malformed native response")
    output = payload.get("output")
    message = output.get("message") if isinstance(output, dict) else None
    content = message.get("content") if isinstance(message, dict) else None
    if not isinstance(content, list):
        raise RuntimeError("The model returned no native text content")
    text = "".join(
        block["text"]
        for block in content
        if isinstance(block, dict)
        and isinstance(block.get("text"), str)
        and block["text"]
    )
    if not text:
        raise RuntimeError("The model returned no native text content")
    return text
```

Serialize the native request to UTF-8 JSON bytes, set both media types, and decode the response according to the selected model's response schema.

Code

```
async def invoke_model(client: AsyncBedrockRuntimeClient) -> str:
    """Invoke a Nova model and print its validated native text response."""
    # Model-native request and response schemas vary by model provider.
    model_id = "global.amazon.nova-2-lite-v1:0"
    prompt = "Name one moon of Saturn."

    response = await client.invoke_model(
        InvokeModelInput(
            model_id=model_id,
            content_type="application/json",
            accept="application/json",
```

```
            body=json.dumps(build_native_request(prompt)).encode("utf-8"),
        )
    )
    text = parse_native_text(response.body)
    print(text)
    return text
```

The entry point creates the client and runs the request. Run the program with `python invoke_model.py`.

```
async def main() -> None:
    """Create a client in a Region that offers the model and run the example."""
    config = await AsyncBedrockRuntimeConfig.resolve(region="us-east-1")
    async with AsyncBedrockRuntimeClient(config=config) as client:
        await invoke_model(client)


if __name__ == "__main__":
    asyncio.run(main())
```

More information

- Use the Invoke API with Amazon Nova 2 in the Amazon Nova User Guide
- Inference request parameters and response fields in the Amazon Bedrock User Guide
- InvokeModel in the Amazon Bedrock API Reference

Use InvokeModelWithResponseStream

InvokeModelWithResponseStream accepts the same model-native request format as InvokeModel but returns the response as a stream of model-native events, so your application can process output as the model generates it instead of waiting for the complete response. Use it for interactive applications that work in a model's native format. The request and event schemas depend on the selected model.

The following example sends one Amazon Nova 2 Messages API request, prints the reply as the model generates it, and then verifies that the stream completed. For request and response details, see the invoke_model_with_response_stream() API reference.

Imports

```
import asyncio
import json
from typing import Any

from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from aws_sdk_bedrock_runtime.models import (
    InvokeModelWithResponseStreamInput,
    ResponseStreamChunk,
    ResponseStreamInternalServerException,
    ResponseStreamModelStreamErrorException,
    ResponseStreamModelTimeoutException,
    ResponseStreamServiceUnavailableException,
    ResponseStreamThrottlingException,
    ResponseStreamUnknown,
    ResponseStreamValidationException,
)
```

The constants name the expected Amazon Nova 2 event types and the modeled stream-error events.

```
NOVA_EVENT_NAMES = {
    "messageStart",
    "contentBlockStart",
    "contentBlockDelta",
    "contentBlockStop",
    "messageStop",
    "metadata",
}
NATIVE_STREAM_ERRORS = (
    ResponseStreamInternalServerException,
    ResponseStreamModelStreamErrorException,
    ResponseStreamModelTimeoutException,
    ResponseStreamServiceUnavailableException,
    ResponseStreamThrottlingException,
    ResponseStreamValidationException,
)
```

The following helper builds the model-native request.

```
def build_native_request(prompt: str) -> dict[str, Any]:
    """Build an Amazon Nova 2 Messages API request."""
```

```
    return {
        "messages": [{"role": "user", "content": [{"text": prompt}]}],
        "inferenceConfig": {"maxTokens": 100, "temperature": 0.2, "topP": 0.9},
    }
```

`invoke_model_with_response_stream` returns model-native JSON events over an open network connection. The following function reads the events inside an `async with` block, which closes the connection when the stream ends or an error interrupts it. It narrows every generated event union, validates every model-native JSON payload, collects the text deltas, and accepts output only if the model signaled a complete message and the stream finished without a modeled error.

```
async def consume_stream(response: Any) -> str:
    """Consume Nova JSON events and reject errors or incomplete output."""
    text_parts: list[str] = []
    saw_message_stop = False

    async with response as stream:
        async for event in stream.output_stream:
            if isinstance(event, ResponseStreamChunk):
                payload = event.value.bytes_
                if not payload:
                    continue
                chunk: Any = json.loads(payload.decode("utf-8"))
                if not isinstance(chunk, dict):
                    raise RuntimeError("The model returned a malformed native event")
                event_names = NOVA_EVENT_NAMES.intersection(chunk)
                if len(event_names) != 1:
                    raise RuntimeError("The model returned an unexpected native event")
                event_name = event_names.pop()
                if event_name == "messageStop":
                    saw_message_stop = True
                elif event_name == "contentBlockDelta":
                    block_delta = chunk[event_name]
                    delta = (
                        block_delta.get("delta")
                        if isinstance(block_delta, dict)
                        else None
                    )
                    if not isinstance(delta, dict):
                        raise RuntimeError("The model returned a malformed text delta")
                    text = delta.get("text")
```

```
                    if text is not None and not isinstance(text, str):
                        raise RuntimeError("The model returned a malformed text delta")
                    if text:
                        text_parts.append(text)
                        print(text, end="", flush=True)
            elif isinstance(event, NATIVE_STREAM_ERRORS):
                raise RuntimeError(event.value.message or type(event).__name__)
            elif isinstance(event, ResponseStreamUnknown):
                raise RuntimeError(f"Unknown native stream event: {event.tag}")
            else:
                raise RuntimeError(
                    f"Unexpected native stream event: {type(event).__name__}"
                )

    if not text_parts or not saw_message_stop:
        raise RuntimeError("The native stream ended without a complete text response")
    return "".join(text_parts)
```

Serialize the native request to UTF-8 JSON bytes, set both media types, and consume the validated stream.

Code

```
async def invoke_model_with_response_stream(client: AsyncBedrockRuntimeClient) -> str:
    """Send a native streaming request and consume its validated output."""
    # Model-native request and event schemas vary by model provider.
    model_id = "global.amazon.nova-2-lite-v1:0"
    prompt = "Name one moon of Saturn."

    response = await client.invoke_model_with_response_stream(
        InvokeModelWithResponseStreamInput(
            model_id=model_id,
            content_type="application/json",
            accept="application/json",
            body=json.dumps(build_native_request(prompt)).encode("utf-8"),
        )
    )
    text = await consume_stream(response)
    print()
    return text
```

The entry point creates the client and runs the streaming request. Run the program with `python invoke_model_with_response_stream.py`.

```
async def main() -> None:
    """Create a client in a Region that offers the model and run the example."""
    config = await AsyncBedrockRuntimeConfig.resolve(region="us-east-1")
    async with AsyncBedrockRuntimeClient(config=config) as client:
        await invoke_model_with_response_stream(client)


if __name__ == "__main__":
    asyncio.run(main())
```

More information

- Stream Amazon Nova 2 responses in the Amazon Nova User Guide
- Inference request parameters and response fields in the Amazon Bedrock User Guide
- InvokeModelWithResponseStream in the Amazon Bedrock API Reference

Use bidirectional streaming

A bidirectional event stream lets the application publish input events to the service while it receives output events over the same connection. Use it with a compatible model when the application must keep sending input, such as live audio, while it receives model output. The event protocol depends on the selected model. For an introduction to event streams, see Working with event streams.

This example uses Amazon Nova 2 Sonic. The program loads its input audio from a `test.pcm` file in the working directory. To provide it, do one of the following:

- Use the SDK repository's sample `test.pcm` file on GitHub. The sample contains headerless, 16 kHz, signed 16-bit, mono PCM audio.
- Use your own recording saved in the same format, and set `audio_file` in `main` to its path.

For request and response details, see the invoke_model_with_bidirectional_stream() API reference.

Bidirectional streaming requires the AWS Common Runtime (CRT) HTTP client, which supports HTTP/2 bidirectional event streams. The SDK's default HTTP transport does not, so opt in by installing the client's `awscrt` extra and passing `transport=AWSCRTHTTPClient()` when you resolve the configuration. For more information, see Use the AWS CRT client for bidirectional streaming.

```
python -m pip install "aws-sdk-bedrock-runtime[awscrt]"
```

Imports

```
import asyncio
import base64
import json
import uuid
from pathlib import Path
from typing import Any

from smithy_http.aio.crt import AWSCRTHTTPClient

from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from aws_sdk_bedrock_runtime.models import (
    BidirectionalInputPayloadPart,
    InvokeModelWithBidirectionalStreamInputChunk,
    InvokeModelWithBidirectionalStreamOperationInput,
    InvokeModelWithBidirectionalStreamOutputChunk,
    InvokeModelWithBidirectionalStreamOutputInternalServerException,
    InvokeModelWithBidirectionalStreamOutputModelStreamErrorException,
    InvokeModelWithBidirectionalStreamOutputModelTimeoutException,
    InvokeModelWithBidirectionalStreamOutputServiceUnavailableException,
    InvokeModelWithBidirectionalStreamOutputThrottlingException,
    InvokeModelWithBidirectionalStreamOutputUnknown,
    InvokeModelWithBidirectionalStreamOutputValidationException,
)
```

The program sends the audio in 512-byte chunks and sleeps 0.016 seconds after each chunk, simulating the real-time delay of live audio. The first four constants control that timing. `DEFAULT_SYSTEM_PROMPT` is the system instruction that the session sends before the audio, and `BIDIRECTIONAL_STREAM_ERRORS` groups the modeled stream-error events. The validation helper rejects a missing or empty audio file before any network call.

```
CHUNK_SIZE = 512
SILENCE_CHUNKS = 125
CHUNK_INTERVAL_SECONDS = 0.016
RESPONSE_WAIT_SECONDS = 3

DEFAULT_SYSTEM_PROMPT = (
    "You are a friendly assistant. Keep your responses short, "
```

```
    "generally one or two sentences."
)

BIDIRECTIONAL_STREAM_ERRORS = (
    InvokeModelWithBidirectionalStreamOutputInternalServerException,
    InvokeModelWithBidirectionalStreamOutputModelStreamErrorException,
    InvokeModelWithBidirectionalStreamOutputModelTimeoutException,
    InvokeModelWithBidirectionalStreamOutputServiceUnavailableException,
    InvokeModelWithBidirectionalStreamOutputThrottlingException,
    InvokeModelWithBidirectionalStreamOutputValidationException,
)
```

```
def validate_audio_input(audio_file: Path) -> None:
    """Require a nonempty audio file supplied by the caller."""
    if not audio_file.is_file() or audio_file.stat().st_size == 0:
        raise ValueError(f"Audio file is missing or empty: {audio_file}")
```

Define and send model-native events

Nova 2 Sonic input events are JSON documents. The session nests three scopes, and an event pair opens and closes each scope: `sessionStart` and `sessionEnd` wrap the connection, `promptStart` and `promptEnd` wrap one prompt, and `contentStart` and `contentEnd` wrap each content block inside the prompt. A content block carries either text or audio. More than one content block opens and closes on the same stream, so each event carries the `promptName` and `contentName` of the scope it belongs to. The templates leave those values as `%s` placeholders. The following templates define every event that this example sends.

```
START_SESSION_EVENT = """{
    "event": {
        "sessionStart": {
            "inferenceConfiguration": {
                "maxTokens": 1024,
                "topP": 0.9,
                "temperature": 0.7
            }
        }
    }
}"""

START_PROMPT_EVENT = """{
    "event": {
```

```
        "promptStart": {
            "promptName": "%s",
            "textOutputConfiguration": {
                "mediaType": "text/plain"
            },
            "audioOutputConfiguration": {
                "mediaType": "audio/lpcm",
                "sampleRateHertz": 24000,
                "sampleSizeBits": 16,
                "channelCount": 1,
                "voiceId": "matthew",
                "encoding": "base64",
                "audioType": "SPEECH"
            }
        }
    }
}"""

TEXT_CONTENT_START_EVENT = """{
    "event": {
        "contentStart": {
            "promptName": "%s",
            "contentName": "%s",
            "type": "TEXT",
            "interactive": true,
            "role": "%s",
            "textInputConfiguration": {
                "mediaType": "text/plain"
            }
        }
    }
}"""

TEXT_INPUT_EVENT = """{
    "event": {
        "textInput": {
            "promptName": "%s",
            "contentName": "%s",
            "content": "%s"
        }
    }
}"""

AUDIO_CONTENT_START_EVENT = """{
```

```
    "event": {
        "contentStart": {
            "promptName": "%s",
            "contentName": "%s",
            "type": "AUDIO",
            "interactive": true,
            "role": "USER",
            "audioInputConfiguration": {
                "mediaType": "audio/lpcm",
                "sampleRateHertz": 16000,
                "sampleSizeBits": 16,
                "channelCount": 1,
                "audioType": "SPEECH",
                "encoding": "base64"
            }
        }
    }
}"""

AUDIO_INPUT_EVENT = """{
    "event": {
        "audioInput": {
            "promptName": "%s",
            "contentName": "%s",
            "content": "%s"
        }
    }
}"""

CONTENT_END_EVENT = """{
    "event": {
        "contentEnd": {
            "promptName": "%s",
            "contentName": "%s"
        }
    }
}"""

PROMPT_END_EVENT = """{
    "event": {
        "promptEnd": {
            "promptName": "%s"
        }
    }
```

```
}"""

SESSION_END_EVENT = """{
    "event": {
        "sessionEnd": {}
    }
}"""
```

The following function encodes one filled-in template and wraps it in the generated input union before sending it to the stream.

```
async def send_event(stream: Any, event_json: str) -> None:
    """Send one JSON input event to the stream."""
    await stream.input_stream.send(
        InvokeModelWithBidirectionalStreamInputChunk(
            value=BidirectionalInputPayloadPart(bytes_=event_json.encode("utf-8"))
        )
    )
```

Publish audio in real time

Read 512-byte chunks, encode them as base64, and wait `CHUNK_INTERVAL_SECONDS` after each chunk to keep the real-time pacing. After the file, the function sends two seconds of silence, waits three seconds for the model to respond, then closes the audio content, prompt, and session in order.

```
async def publish_audio(
    stream: Any,
    prompt_name: str,
    content_name: str,
    audio_file: Path,
) -> None:
    """Publish caller-provided PCM audio, trailing silence, and end events."""
    try:
        chunks_sent = 0
        with audio_file.open("rb") as source:
            while chunk := source.read(CHUNK_SIZE):
                chunks_sent += 1
                encoded = base64.b64encode(chunk).decode("utf-8")
                await send_event(
                    stream, AUDIO_INPUT_EVENT % (prompt_name, content_name, encoded)
                )
```

```
                await asyncio.sleep(CHUNK_INTERVAL_SECONDS)
        if chunks_sent == 0:
            raise RuntimeError(f"No audio read from {audio_file}")

        silence = base64.b64encode(bytes(CHUNK_SIZE)).decode("utf-8")
        for _ in range(SILENCE_CHUNKS):
            await send_event(
                stream, AUDIO_INPUT_EVENT % (prompt_name, content_name, silence)
            )
            await asyncio.sleep(CHUNK_INTERVAL_SECONDS)

        await send_event(stream, CONTENT_END_EVENT % (prompt_name, content_name))
        await asyncio.sleep(RESPONSE_WAIT_SECONDS)
        await send_event(stream, PROMPT_END_EVENT % prompt_name)
        await send_event(stream, SESSION_END_EVENT)
    finally:
        await stream.input_stream.close()
```

Receive text and audio output

Await the service output before iterating it. Narrow every generated output event, decode its JSON payload, collect text, and record audio output. Stop when the model sends `completionEnd`.

```
async def receive_events(stream: Any) -> tuple[list[str], bool, bool]:
    """Receive narrowed output events until the model signals completion."""
    _, output_stream = await stream.await_output()
    if output_stream is None:
        raise RuntimeError("The service returned no output stream")

    text_output: list[str] = []
    got_audio = False
    completed = False
    async for event in output_stream:
        if isinstance(event, InvokeModelWithBidirectionalStreamOutputChunk):
            payload = event.value.bytes_
            if not payload:
                continue
            event_data = json.loads(payload.decode("utf-8")).get("event", {})
            if "textOutput" in event_data:
                text_output.append(event_data["textOutput"].get("content", ""))
            if "audioOutput" in event_data:
                got_audio = True
            if "completionEnd" in event_data:
```

```
                completed = True
                break
        elif isinstance(event, BIDIRECTIONAL_STREAM_ERRORS):
            raise RuntimeError(event.value.message or type(event).__name__)
        elif isinstance(event, InvokeModelWithBidirectionalStreamOutputUnknown):
            raise RuntimeError(f"Unknown bidirectional stream event: {event.tag}")
        else:
            raise RuntimeError(
                f"Unexpected bidirectional stream event: {type(event).__name__}"
            )
    return text_output, got_audio, completed
```

Open the stream and exchange audio

Open the stream, then send the `init_events` list in protocol order: start the session and the prompt, send the system instruction as a complete text content block, and open the user audio content block. Then run the publisher and receiver concurrently, so the program keeps sending audio while it receives output. Verify that the stream completed and produced both text and audio before accepting the result.

```
async def invoke_model_with_bidirectional_stream(
    client: AsyncBedrockRuntimeClient,
    audio_file: Path,
) -> str:
    """Open a Nova 2 Sonic stream and concurrently send and receive audio."""
    validate_audio_input(audio_file)

    stream = await client.invoke_model_with_bidirectional_stream(
        InvokeModelWithBidirectionalStreamOperationInput(
            model_id="amazon.nova-2-sonic-v1:0"
        )
    )
    prompt_name = str(uuid.uuid4())
    system_content_name = str(uuid.uuid4())
    audio_content_name = str(uuid.uuid4())

    init_events = [
        START_SESSION_EVENT,
        START_PROMPT_EVENT % prompt_name,
        TEXT_CONTENT_START_EVENT % (prompt_name, system_content_name, "SYSTEM"),
        TEXT_INPUT_EVENT % (prompt_name, system_content_name, DEFAULT_SYSTEM_PROMPT),
        CONTENT_END_EVENT % (prompt_name, system_content_name),
        AUDIO_CONTENT_START_EVENT % (prompt_name, audio_content_name),
```

```
    ]

    async with stream:
        for event in init_events:
            await send_event(stream, event)
        _, received = await asyncio.gather(
            publish_audio(stream, prompt_name, audio_content_name, audio_file),
            receive_events(stream),
        )

    text_output, got_audio, completed = received
    if not completed or not text_output or not got_audio:
        raise RuntimeError(
            "The stream ended without completion, text, and audio output"
        )
    # The first text event is the transcript of the input audio, and the
    # following events are the model's reply, one segment per event.
    text = "\n".join(part.strip() for part in text_output)
    print(text)
    return text
```

The entry point creates the client with the CRT transport and runs the streaming session. Run the program with `python invoke_model_with_bidirectional_stream.py`.

```
async def main() -> None:
    """Create a client with the CRT transport and run the example."""
    # Load the input audio (headerless, 16 kHz, signed 16-bit, mono PCM).
    audio_file = Path("test.pcm")
    # Bidirectional streaming requires the AWS CRT HTTP client.
    config = await AsyncBedrockRuntimeConfig.resolve(
        region="us-east-1",
        transport=AWSCRTHTTPClient(),
    )
    async with AsyncBedrockRuntimeClient(config=config) as client:
        await invoke_model_with_bidirectional_stream(client, audio_file)


if __name__ == "__main__":
    asyncio.run(main())
```

The example follows the same event sequence as [test_bidirectional_streaming.py] in the AWS SDK for Python repository on GitHub.

More information

- Speech-to-Speech (Amazon Nova 2 Sonic) in the Amazon Nova User Guide
- InvokeModelWithBidirectionalStream in the Amazon Bedrock API Reference

Working with Boto3

If you have existing Boto3 code and want to evaluate the AWS SDK for Python, these pages help you get started:

- Key differences — API-model and execution differences between the two SDKs.
- Use both SDKs in one application — Architectural patterns for running both SDKs together, including configuration, sync/async boundaries, and common pitfalls.
- Convert Boto3 operations to the AWS SDK for Python — Rewrite individual Boto3 calls using the generated types and async model of the new SDK.
- Choosing the right AWS SDK for Python — Decision guide for determining the best path for your workload.

Key differences

Boto3 and the AWS SDK for Python have different execution models and Python client interfaces. The following table summarizes the API-model and execution differences that affect your application code. The operation, request, response, streaming, and error rows compare generated AWS SDK for Python clients with Boto3 low-level clients.

Area	Boto3	AWS SDK for Python
Execution model	Service operations are synchronous and block the calling thread until they complete.	Service operations are asynchronous and yield to the event loop instead of blocking, so other tasks can run while a call is in flight.
Packages and service coverage	Installing `boto3` provides access to clients and features for a broad set of AWS services.	Each service client is distributed as a separate package. The Developer Preview covers a growing subset of AWS services, and supported clients do not yet implement every Boto3 feature.

Area	Boto3	AWS SDK for Python
Clients and imports	Import `boto3` and create clients dynamically, such as `boto3.client("dynamodb")`.	Install a service package and import its generated client, such as `AsyncDynamoDBClient` from `aws-sdk-dynamodb`.
Higher-level features	Provides higher-level abstractions for supported services, including Resources, paginator and waiter objects, and managed transfer utilities.	The Developer Preview focuses on generated service clients. It doesn't currently provide Boto3-equivalent Resources, paginator or waiter objects, or managed transfer utilities.
Operation and member names	Low-level client operation methods use snake case, such as `put_object`, but request parameter names preserve modeled API casing, such as `Bucket` and `Key`.	Operation methods and generated input and output model members use snake case, such as `start_stream_transcription()` and `media_sample_rate_hertz`.
Requests	Low-level clients accept modeled request parameters as keyword arguments.	Construct a generated input model and pass it as the operation's `input` parameter, either positionally or with `input=`.
Responses	Low-level client operations generally return service responses as dictionaries. Read values by key and check whether optional keys are present. Higher-level Boto3 Resources and helpers have API-specific return types, such as resource objects, iterables, or `None`.	Most non-streaming operations return generated output models, while streaming operations return typed asynchronous stream wrappers. Read model attributes and check optional members for `None`. When a member is a union, narrow its generated variant before accessing its value.

Area	Boto3	AWS SDK for Python
Streaming	Boto3 clients don't provide native asynchronous operations. Reading supported streaming response bodies or event streams uses synchronous APIs and can block the calling thread.	Supported streaming operations expose typed asynchronous event streams. Some operations provide bidirectional streams over HTTP/2.
Errors	Client-side failures use Botocore exception classes. Service errors can be caught as `ClientError`, which exposes details through its response dictionary.	Modeled service errors use concrete generated exception classes, including errors raised while consuming a stream. Other SDK runtime exceptions can occur during configuration, request preparation, transport, or response processing.
Configuration	`boto3.client()` uses the default Boto3 session. Create a `boto3.Session` or pass a Botocore `Config` to customize settings.	Generated clients use their default configuration. `AsyncAwsConfig` resolves common AWS settings, and generated service-specific configuration classes add service settings. To customize a client, call the service-specific class's inherited `resolve()` method, such as `await AsyncDynamoDBConfig.resolve(...)`, and pass the result.

Use both SDKs in one application

You can use the AWS SDK for Python and Boto3 together in the same application. The most common reasons to combine both SDKs are:

- A service or operation you need isn't yet available in the AWS SDK for Python, so you use Boto3 for that service while using the new SDK for async or streaming workloads.

- You want to evaluate the async capabilities of the AWS SDK for Python in an existing Boto3 application without rewriting your synchronous code.

Keep existing Boto3 code when it continues to meet your application's needs, and add the AWS SDK for Python for supported capabilities such as native asynchronous APIs, concurrent I/O, bidirectional streaming, modular service packages, and generated types.

Configuration and credentials

Both SDKs can resolve standard AWS settings and authenticate as the same AWS identity, but each SDK resolves and refreshes its own configuration independently. Programmatic settings aren't shared between the SDKs.

Key points:

- A profile or Region selected with a Boto3 `Session` doesn't apply to AWS SDK for Python configuration resolution.
- The generated asynchronous service configuration requires a Region. Resolution fails if no Region is available.
- Configure equivalent options for each SDK when necessary (Region, credentials, endpoint overrides).

Before running either scenario, make a Region available to each SDK through standard AWS settings (environment variables or config files) or an SDK-specific override.

Combining synchronous and asynchronous code

How you combine synchronous and asynchronous code depends on which SDK is primary:

Your application is...	Pattern	Example
Async-primary (AWS SDK for Python owns the event loop)	Use `asyncio.to_thread()` for Boto3 calls so they don't block the event loop	Scenario 1

Your application is...	Pattern	Example
Sync-primary (Boto3 owns the main thread)	Use `asyncio.run()` to enter an async workflow for the AWS SDK for Python	Scenario 2

For details on running asynchronous code, see Running asynchronous code. For keeping blocking work off the event loop, see Keeping blocking work off the event loop.

Before you begin

Both scenarios require the following:

- Complete Prerequisites and installation and Authenticating with AWS using the AWS SDK for Python.
- A Region available to both SDKs through standard AWS settings or SDK-specific configuration.
- IAM permissions as described in each scenario.

> **Warning**
>
> Both scenarios use AWS services that can incur charges. Use non-production resources and review pricing for the relevant services before repeated use.

Scenario 1: Async-primary application with Boto3 for unsupported services

This scenario represents a new asynchronous application that uses `AsyncBedrockRuntimeClient` for output streaming. The application sends a request to Amazon Bedrock Runtime and asynchronously processes response events as the model generates output. After the stream completes, the application uses Boto3 to save the prompt and completed response as a JSON object in Amazon S3, which isn't yet available in the AWS SDK for Python.

When to use this pattern: Your application is async-first and you need Boto3 only for services or features not yet available in the AWS SDK for Python.

Install

```
python -m pip install boto3 aws-sdk-bedrock-runtime
```

Additional requirements

- Access to an Amazon Bedrock model that supports `ConverseStream` in the configured AWS Region. This example processes text output only.
- An Amazon S3 bucket and key where the example can store the model response.
- The identity resolved by the AWS SDK for Python allows `bedrock:InvokeModelWithResponseStream` for the selected model.
- The identity resolved by Boto3 allows `s3:PutObject` for the output object.

Write the code

Create a file named `sdk_primary_with_boto3.py` with the following code:

```
"""Use Boto3 from an asynchronous AWS SDK for Python application."""

import argparse
import asyncio
import json

import boto3

from aws_sdk_bedrock_runtime.client import AsyncBedrockRuntimeClient
from aws_sdk_bedrock_runtime.config import AsyncBedrockRuntimeConfig
from aws_sdk_bedrock_runtime.models import (
    ContentBlockDeltaText,
    ContentBlockText,
    ConverseStreamInput,
    ConverseStreamOutputContentBlockDelta,
    ConverseStreamOutputMessageStop,
    ConverseStreamOutputMetadata,
    ConverseStreamOutputUnknown,
    InternalServerException,
    Message,
    ModelStreamErrorException,
    ServiceUnavailableException,
    ThrottlingException,
```

```
    ValidationException,
)

DEFAULT_MODEL_ID = "global.amazon.nova-2-lite-v1:0"
MODELED_STREAM_ERRORS = (
    InternalServerException,
    ModelStreamErrorException,
    ServiceUnavailableException,
    ThrottlingException,
    ValidationException,
)


async def stream_response(
    client: AsyncBedrockRuntimeClient,
    model_id: str,
    prompt: str,
) -> str:
    text_parts: list[str] = []
    stop_reason = None
    got_metadata = False
    try:
        response = await client.converse_stream(
            input=ConverseStreamInput(
                model_id=model_id,
                messages=[
                    Message(
                        role="user",
                        content=[ContentBlockText(value=prompt)],
                    )
                ],
            )
        )
        async with response as stream:
            async for event in stream.output_stream:
                if isinstance(
                    event, ConverseStreamOutputContentBlockDelta
                ):
                    delta = event.value.delta
                    if isinstance(delta, ContentBlockDeltaText):
                        text_parts.append(delta.value)
                        print(delta.value, end="", flush=True)
                elif isinstance(event, ConverseStreamOutputMessageStop):
                    stop_reason = event.value.stop_reason
```

```
                elif isinstance(event, ConverseStreamOutputMetadata):
                    got_metadata = True
                elif isinstance(event, ConverseStreamOutputUnknown):
                    raise RuntimeError(
                        f"Unknown stream event: {event.tag}"
                    )
    except MODELED_STREAM_ERRORS as error:
        raise RuntimeError(
            error.message or type(error).__name__
        ) from error

    if not text_parts or stop_reason is None or not got_metadata:
        raise RuntimeError("Amazon Bedrock returned an incomplete response")
    print()
    return "".join(text_parts)


def save_response(
    bucket: str,
    key: str,
    prompt: str,
    response: str,
) -> None:
    body = json.dumps(
        {"prompt": prompt, "response": response},
        ensure_ascii=False,
        indent=2,
    ).encode("utf-8")
    s3 = boto3.client("s3")
    s3.put_object(
        Bucket=bucket,
        Key=key,
        Body=body,
        ContentType="application/json",
    )


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument("bucket")
    parser.add_argument("key")
    parser.add_argument("prompt")
    parser.add_argument("--model-id", default=DEFAULT_MODEL_ID)
    return parser.parse_args()
```

```
async def main(args: argparse.Namespace) -> None:
    config = await AsyncBedrockRuntimeConfig.resolve()
    async with AsyncBedrockRuntimeClient(config=config) as client:
        response = await stream_response(
            client,
            args.model_id,
            args.prompt,
        )

        await asyncio.to_thread(
            save_response,
            args.bucket,
            args.key,
            args.prompt,
            response,
        )
        print(f"Saved response to s3://{args.bucket}/{args.key}")


if __name__ == "__main__":
    asyncio.run(main(parse_args()))
```

How it works

The AWS SDK for Python owns the event loop and the application's primary workflow:

1. `AsyncBedrockRuntimeConfig.resolve()` resolves Region and credentials from standard AWS settings.
2. `stream_response()` consumes response events that Amazon Bedrock Runtime streams as the model generates output.
3. After the stream completes, `asyncio.to_thread()` runs the synchronous Boto3 S3 upload without blocking the event loop.

`AsyncBedrockRuntimeConfig.resolve()` and `boto3.client("s3")` independently use the standard AWS settings configured for the environment. The application doesn't copy the Region or credentials from one SDK to the other.

Run the application

Pass the output bucket, output key, and model prompt:

```
python sdk_primary_with_boto3.py amzn-s3-demo-bucket responses/example.json "Give three
 tips for writing clear Python code."
```

Success

A successful run prints the streamed model response and the output Amazon S3 URI. The output object contains the prompt and completed response as UTF-8 JSON.

Cleanup

This example doesn't create persistent resources other than the output Amazon S3 object. Delete that object when you no longer need it.

Next steps

- For how the SDK resolves configuration values, see Configuration resolution. For how it finds credentials, see Credential providers.
- For running blocking functions from an asynchronous application, see Keeping blocking work off the event loop.
- For more Amazon Bedrock Runtime operations and streaming behavior, see Amazon Bedrock Runtime.
- For generated Amazon Bedrock Runtime request, response, and event types, see the ConverseStream API reference.
- For Boto3 S3 operations, see the SDK for Python (Boto3) S3 API reference.

Scenario 2: Sync-primary application adding bidirectional streaming

This scenario represents an existing synchronous application that uses Boto3 to download PCM audio from Amazon S3. The application adds `AsyncTranscribeStreamingClient` to send audio to Amazon Transcribe and receive transcript events concurrently over a bidirectional stream. After the stream completes, the existing Boto3 workflow uploads the completed transcript to Amazon S3.

When to use this pattern: You have an existing Boto3 application and want to add async or streaming capabilities for specific operations without rewriting your synchronous code.

Install

Install the `awscrt` extra to use the AWS Common Runtime (CRT) HTTP client that bidirectional streaming requires. For more information, see Use the AWS CRT client for bidirectional streaming.

```
python -m pip install boto3 "aws-sdk-transcribe-streaming[awscrt]"
```

Additional requirements

- An Amazon S3 object containing US English audio as headerless, signed 16-bit little-endian, mono PCM sampled at 16 kHz. WAV files include a container header and aren't valid input for this example. To transcribe another supported language, change `LanguageCode.EN_US` in the request. To try this example, you can download a sample PCM clip (`test.pcm`) from the aws-sdk-python repository on GitHub.
- An output Amazon S3 bucket and key where the example can store the UTF-8 transcript.
- The identity resolved by Boto3 allows `s3:GetObject` for the input object and `s3:PutObject` for the output object.
- The identity resolved by the AWS SDK for Python allows `transcribe:StartStreamTranscription`.

Write the code

Create a file named `boto3_with_streaming.py` with the following code:

```
"""Add bidirectional transcription to a synchronous Boto3 application."""

import argparse
import asyncio
from pathlib import Path
from tempfile import TemporaryDirectory
from typing import Any

import boto3

from smithy_http.aio.crt import AWSCRTHTTPClient
```

```
from aws_sdk_transcribe_streaming.client import (
    AsyncTranscribeStreamingClient,
)
from aws_sdk_transcribe_streaming.config import (
    AsyncTranscribeStreamingConfig,
)
from aws_sdk_transcribe_streaming.models import (
    AudioEvent,
    AudioStreamAudioEvent,
    BadRequestException,
    ConflictException,
    InternalFailureException,
    LanguageCode,
    LimitExceededException,
    MediaEncoding,
    ServiceUnavailableException,
    StartStreamTranscriptionInput,
    TranscriptResultStreamTranscriptEvent,
    TranscriptResultStreamUnknown,
)

SAMPLE_RATE = 16_000
BYTES_PER_SAMPLE = 2
CHUNK_SIZE = 3_200  # 100 ms of 16 kHz, 16-bit, mono PCM
MODELED_STREAM_ERRORS = (
    BadRequestException,
    ConflictException,
    InternalFailureException,
    LimitExceededException,
    ServiceUnavailableException,
)


async def send_audio(stream: Any, audio_path: Path) -> None:
    loop = asyncio.get_running_loop()
    started = loop.time()
    audio_seconds = 0.0

    try:
        with audio_path.open("rb") as source:
            while chunk := await asyncio.to_thread(
                source.read, CHUNK_SIZE
            ):
                await stream.input_stream.send(
```

```
                    AudioStreamAudioEvent(
                        value=AudioEvent(audio_chunk=chunk)
                    )
                )
                audio_seconds += len(chunk) / (
                    BYTES_PER_SAMPLE * SAMPLE_RATE
                )
                delay = started + audio_seconds - loop.time()
                if delay > 0:
                    await asyncio.sleep(delay)
    finally:
        await stream.input_stream.close()


async def receive_transcripts(stream: Any) -> list[str]:
    transcripts: list[str] = []
    try:
        _, output_stream = await stream.await_output()
        if output_stream is None:
            raise RuntimeError("Amazon Transcribe returned no output stream")

        async for event in output_stream:
            if isinstance(event, TranscriptResultStreamUnknown):
                raise RuntimeError(
                    f"Unknown stream event: {event.tag}"
                )
            if not isinstance(
                event, TranscriptResultStreamTranscriptEvent
            ):
                raise RuntimeError(
                    f"Unexpected stream event: {type(event).__name__}"
                )

            transcript = event.value.transcript
            if transcript is None:
                continue
            for result in transcript.results or []:
                if result.is_partial:
                    continue
                alternatives = result.alternatives or []
                if alternatives and alternatives[0].transcript:
                    transcripts.append(alternatives[0].transcript)
    except MODELED_STREAM_ERRORS as error:
        raise RuntimeError(
```

```
            error.message or type(error).__name__
        ) from error

    return transcripts


async def transcribe_audio(audio_path: Path) -> str:
    # Bidirectional streaming requires the AWS CRT HTTP client.
    config = await AsyncTranscribeStreamingConfig.resolve(
        transport=AWSCRTHTTPClient(),
    )
    async with AsyncTranscribeStreamingClient(config=config) as client:
        stream = await client.start_stream_transcription(
            input=StartStreamTranscriptionInput(
                language_code=LanguageCode.EN_US,
                media_sample_rate_hertz=SAMPLE_RATE,
                media_encoding=MediaEncoding.PCM,
            )
        )

        async with stream:
            _, transcripts = await asyncio.gather(
                send_audio(stream, audio_path),
                receive_transcripts(stream),
            )

        return "\n".join(transcripts)


def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument("input_bucket")
    parser.add_argument("input_key")
    parser.add_argument("output_bucket")
    parser.add_argument("output_key")
    return parser.parse_args()


def main() -> None:
    args = parse_args()
    s3 = boto3.client("s3")

    with TemporaryDirectory() as temp_directory:
        audio_path = Path(temp_directory) / "input.pcm"
```

```
        s3.download_file(
            args.input_bucket,
            args.input_key,
            str(audio_path),
        )
        print(
            f"Downloaded s3://{args.input_bucket}/{args.input_key}"
        )

        transcript = asyncio.run(transcribe_audio(audio_path))

    if not transcript:
        raise RuntimeError("Amazon Transcribe returned no completed text")

    s3.put_object(
        Bucket=args.output_bucket,
        Key=args.output_key,
        Body=transcript.encode("utf-8"),
        ContentType="text/plain; charset=utf-8",
    )
    print(
        f"Uploaded transcript to "
        f"s3://{args.output_bucket}/{args.output_key}"
    )


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

How it works

The synchronous `main()` function keeps the existing Boto3 workflow unchanged:

1. Boto3 downloads the PCM audio from Amazon S3.
2. `asyncio.run()` enters the async transcription workflow once — the event loop starts and stops within this call.
3. Inside the async context, `asyncio.gather()` sends audio and receives transcript events concurrently over one bidirectional HTTP/2 stream.
4. After `asyncio.run()` returns, Boto3 uploads the completed transcript to Amazon S3 synchronously.

`AsyncTranscribeStreamingConfig.resolve()` and `boto3.client("s3")` independently use the standard AWS settings configured for the environment. The application doesn't pass the Region or credentials between the SDKs.

Run the application

Pass the input bucket and PCM object key, followed by the output bucket and transcript object key:

```
python boto3_with_streaming.py amzn-s3-demo-input-bucket audio/test.pcm amzn-s3-demo-output-bucket transcripts/test.txt
```

Success

A successful run prints the input and output Amazon S3 URIs. The output object contains the completed transcript segments returned by Amazon Transcribe.

Cleanup

The temporary local audio file is removed automatically. This example doesn't delete the input or output Amazon S3 objects. Delete the output object when you no longer need it.

Next steps

- For how the SDK resolves configuration values, see Configuration resolution. For how it finds credentials, see Credential providers.
- For running an asynchronous workflow from a synchronous application, see Running asynchronous code.
- For event-stream lifecycle and completion behavior, see Working with event streams.
- For generated Amazon Transcribe Streaming request and event types, see the StartStreamTranscription API reference.
- For Boto3 S3 operations, see the SDK for Python (Boto3) S3 API reference.

Common pitfalls

Avoid these issues when using both SDKs together:

Pitfall	Impact	Resolution
Calling synchronous Boto3 methods inside an active `asyncio` event loop	Blocks the event loop, eliminating concurrency benefits and potentially causing timeouts	Use `asyncio.to_thread()` to run Boto3 calls in a thread pool (see Scenario 1)
Assuming configuration is shared between SDKs	One SDK may use an unexpected Region or credentials	Verify each SDK resolves the correct Region and credentials independently
Calling `asyncio.run()` from inside a running event loop	Raises `RuntimeError` — you cannot nest event loops	Use `await` directly if already inside an async context, or restructure to call `asyncio.run()` only from synchronous code
Creating Boto3 clients inside tight async loops	Client construction is synchronous and repeated creation adds latency	Create Boto3 clients once outside the loop and pass them in, wrapped with `asyncio.to_thread()` if called from async code

Convert Boto3 operations to the AWS SDK for Python

The AWS SDK for Python is currently in Developer Preview. You can try your existing Boto3 application code with the new SDK to evaluate its asynchronous capabilities, modular packages, and generated types in development and test environments. Do not use the AWS SDK for Python for production workloads.

The two SDKs use separate packages and namespaces, so you can add the AWS SDK for Python alongside Boto3 and try one supported operation or workflow at a time.

What you can do today: Try converting your Boto3 application code to the AWS SDK for Python for prototyping and evaluation. This lets you assess the asynchronous model, generated types, and modular architecture of the new SDK before general availability.

What this page is not: A guide for migrating production deployments. Production workloads should remain on Boto3 until the AWS SDK for Python reaches General Availability.

For guidance on choosing between the two SDKs, see Choosing the right AWS SDK for Python.

Before you start

- Check Supported services for current client availability, then confirm each required operation and generated type in the AWS SDK for Python API Reference. An available client or operation does not by itself establish support for Boto3 Resources, paginators, waiters, transfer utilities, or streaming.
- Identify the complete workflow to migrate, including its callers, configuration, error handling, tests, and any Boto3-specific features. Review Key differences before changing its API contracts.
- Keep Boto3 installed alongside the new SDK. Pin the generated service-package versions used by your application and retest when you upgrade them.

Try an operation

1. Add the generated package for the service. Import its asynchronous client, configuration, input and output models, unions, and modeled exceptions.
2. Resolve the generated asynchronous service configuration and pass it to the client. Values held in a Boto3 `Session` or Botocore `Config` object do not transfer to the new client. Both SDKs can independently resolve environment variables and shared AWS files, so verify the effective settings for each implementation.
3. Move the path that calls the new SDK into `async def` functions. A synchronous script can call `asyncio.run()` once at its entry point; code running in an existing event loop must await the migrated function. See Running asynchronous code.
4. Replace Boto3 keyword arguments and request dictionaries with the generated input model. Use snake-case member names and construct the appropriate generated union variants.
5. Replace response-dictionary access with generated output attributes. Check optional members for `None` and narrow generated unions before reading their values.
6. Replace `ClientError` parsing with generated modeled exceptions where applicable, and account for runtime exceptions raised while preparing, sending, or processing a request. Update tests for the new asynchronous and generated types. See Handling errors.

Example: convert a DynamoDB GetItem operation

This example applies the conversion sequence to one low-level DynamoDB `GetItem` call. Both versions request the same composite primary key with a strongly consistent read.

Before you run the example:

- Create a DynamoDB table with string partition and sort keys named `author` and `title`.
- Add an item with `author` set to "Ada Lovelace" and `title` set to "Notes on the Analytical Engine".

Step 1: Add the service package

Install both packages for side-by-side comparison:

```
python -m pip install boto3 aws-sdk-dynamodb
```

The configured identity needs `dynamodb:GetItem` permission for the table. Use a non-production table while comparing the implementations.

Step 2: Identify the Boto3 request and response

The original Boto3 code creates the client dynamically, runs the operation synchronously, accepts modeled request names as keyword arguments, and returns nested dictionaries.

```
from typing import Any

import boto3

AUTHOR = "Ada Lovelace"
TITLE = "Notes on the Analytical Engine"


def get_book_with_boto3(
    client: Any,
    table_name: str,
) -> dict[str, Any] | None:
    response = client.get_item(
        TableName=table_name,
        Key={
            "author": {"S": AUTHOR},
```

```
            "title": {"S": TITLE},
        },
        ConsistentRead=True,
    )
    return response.get("Item")


def run_boto3(table_name: str, region: str) -> None:
    client = boto3.client("dynamodb", region_name=region)
    item = get_book_with_boto3(client, table_name)
    if item is None:
        print("The book was not found.")
        return

    title = item.get("title", {}).get("S")
    if not isinstance(title, str):
        raise RuntimeError("DynamoDB returned no string title")
    print(title)
```

The service function and its caller separate the SDK operation from application-level result handling. The conversion changes four SDK-facing parts: client construction, the synchronous boundary, request dictionaries, and response-dictionary access. The not-found message and title validation are application behavior that the migrated version preserves.

Step 3: Replace the complete operation

The migrated version preserves the same result handling. Its SDK-specific code imports the generated client, asynchronous configuration, input model, and attribute-value union variants; resolves the configuration; awaits `get_item()`; reads the optional `item` output attribute; and narrows the returned title to `AttributeValueS`.

```
from aws_sdk_dynamodb.client import AsyncDynamoDBClient
from aws_sdk_dynamodb.config import AsyncDynamoDBConfig
from aws_sdk_dynamodb.models import (
    AttributeValue,
    AttributeValueS,
    GetItemInput,
)

AUTHOR = "Ada Lovelace"
TITLE = "Notes on the Analytical Engine"
```

```
async def get_book_with_sdk(
    client: AsyncDynamoDBClient,
    table_name: str,
) -> dict[str, AttributeValue] | None:
    response = await client.get_item(
        GetItemInput(
            table_name=table_name,
            key={
                "author": AttributeValueS(value=AUTHOR),
                "title": AttributeValueS(value=TITLE),
            },
            consistent_read=True,
        )
    )
    return response.item


async def run_sdk(table_name: str, region: str) -> None:
    config = await AsyncDynamoDBConfig.resolve(region=region)
    async with AsyncDynamoDBClient(config=config) as client:
        item = await get_book_with_sdk(client, table_name)
        if item is None:
            print("The book was not found.")
            return

        title = item.get("title")
        if not isinstance(title, AttributeValueS):
            raise RuntimeError("DynamoDB returned no string title")
        print(title.value)
```

The complete example keeps these SDK-specific functions separate from its shared command-line entry point. Code that is already running in an event loop should await `run_sdk()` instead of calling `asyncio.run()`.

Step 4: Run and compare the implementations

The complete example uses one shared command-line parser and entry point for both implementations:

```
import argparse
import asyncio
```

```
def parse_args() -> argparse.Namespace:
    parser = argparse.ArgumentParser()
    parser.add_argument("table_name")
    parser.add_argument("--region", required=True)
    parser.add_argument(
        "--implementation",
        choices=("boto3", "sdk"),
        default="sdk",
    )
    return parser.parse_args()


def main() -> None:
    args = parse_args()
    if args.implementation == "boto3":
        run_boto3(args.table_name, args.region)
    else:
        asyncio.run(run_sdk(args.table_name, args.region))


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

`--implementation boto3` calls the synchronous Boto3 branch directly. `--implementation sdk` starts one event loop for the asynchronous AWS SDK for Python branch.

Run each implementation against the same table, key, and Region:

```
python migrate_boto3_dynamodb.py Books --region us-east-1 --implementation boto3
python migrate_boto3_dynamodb.py Books --region us-east-1 --implementation sdk
```

Both commands should print the same title for the example item and should both report when that item is absent.

Next steps

- For guidance on choosing the best path for your application, see Choosing the right AWS SDK for Python.
- For current client availability, see Supported services.
- For API-model and execution differences, see Key differences.

- For workflows that continue to require both SDKs, see Use both SDKs in one application.
- For asynchronous unit and integration testing patterns, see Testing applications that use the SDK.

Security for the AWS SDK for Python

Cloud security at Amazon Web Services (AWS) is the highest priority. As an AWS customer, you benefit from a data center and network architecture that is 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 all of the services offered in the AWS Cloud and providing you with services that you can use securely. Our security responsibility is the highest priority at AWS, and the effectiveness of our security is regularly tested and verified by third-party auditors as part of the AWS Compliance Programs.

Security in the Cloud – Your responsibility is determined by the AWS service you are using, and other factors including the sensitivity of your data, your organization's requirements, and applicable laws and regulations.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Topics

- Data protection in the AWS SDK for Python
- Working with TLS in the AWS SDK for Python
- Identity and Access Management
- Compliance Validation for this AWS Product or Service
- Resilience for this AWS Product or Service
- Infrastructure Security for this AWS Product or Service

Data protection in the AWS SDK for Python

The AWS shared responsibility model applies to data protection in AWS SDK for Python. 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 Data Privacy FAQ. For information about data protection in Europe, see the General Data Protection Regulation (GDPR) Center.

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. For information about using CloudTrail trails to capture AWS activities, see Working with CloudTrail trails in the *AWS CloudTrail User Guide*.
- 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 140-3 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) 140-3.

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 AWS SDK for Python 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.

Working with TLS in the AWS SDK for Python

The AWS SDK for Python relies on the TLS implementation of its HTTP transport to secure connections to AWS services. AWS service endpoints require TLS 1.2 or later, and TLS 1.3 is recommended.

TLS 1.3 is required to enable post-quantum cryptography, which might require additional actions or configuration. To learn more, see Enabling hybrid post-quantum TLS.

How TLS versions are selected

The SDK does not select a specific TLS protocol version. The TLS behavior depends on the HTTP transport that the client's configuration uses:

- By default, clients use `AIOHTTPClient`. This transport creates an `aiohttp` client session without supplying an SSL context. By default, `aiohttp` uses a verified SSL context created by Python's `ssl` module. The enabled versions and cipher suites depend on the TLS library used by the Python runtime and its configuration.
- Clients configured with `AWSCRTHTTPClient`, which is required for HTTP/2 bidirectional streaming, use an AWS Common Runtime (CRT) TLS context with default options, and the SDK does not override its TLS-version settings. The enabled versions depend on AWS CRT and its platform TLS implementation.

When a connection is established, the transport negotiates a TLS version supported by both the client environment and the AWS service endpoint. Both built-in transports verify server certificates by default. If you provide a custom transport, consult its documentation for its TLS behavior.

Check TLS version information

For clients that use `AIOHTTPClient`, use the Python `ssl` module to inspect the TLS library and the version limits of the default SSL context:

```
import ssl

context = ssl.create_default_context()

print(f"TLS library: {ssl.OPENSSL_VERSION}")
print(f"Minimum TLS version: {context.minimum_version.name}")
print(f"Maximum TLS version: {context.maximum_version.name}")
```

The output depends on the Python runtime and platform. It shows the settings of the default Python SSL context, not the TLS version negotiated for a particular request.

This check does not apply to clients that use `AWSCRTHTTPClient` because that transport uses AWS CRT instead of Python's `ssl` module. The built-in CRT transport does not provide a public API for reporting the TLS version negotiated for a connection. If you need to confirm the negotiated version, use TLS diagnostic tooling appropriate for your environment.

Enforce a minimum TLS version

The SDK does not override the minimum TLS version of either built-in transport, and their public configuration does not provide a minimum-version setting. AWS service endpoints reject connections that use a version earlier than TLS 1.2.

If your application must explicitly enforce a minimum version on the client, provide a compatible custom HTTP transport through the generated client's `transport` configuration. Configure the transport or its TLS implementation to enforce the required minimum version, and follow the transport's documentation.

AWS API endpoints and TLS 1.2

For more information about the TLS 1.2 minimum for AWS API endpoints, see TLS 1.2 to become the minimum TLS protocol level for all AWS API endpoints.

Identity and Access Management

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 AWS resources. IAM is an AWS service that you can use with no additional charge.

Topics

- Audience
- Authenticating with identities
- Managing access using policies
- How AWS services work with IAM
- Troubleshooting AWS identity and access

Audience

How you use AWS Identity and Access Management (IAM) differs, depending on the work that you do in AWS.

Service user – If you use AWS services to do your job, then your administrator provides you with the credentials and permissions that you need. As you use more AWS 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 AWS, see Troubleshooting AWS identity and access or the user guide of the AWS service you are using.

Service administrator – If you're in charge of AWS resources at your company, you probably have full access to AWS. It's your job to determine which AWS 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 AWS, see the user guide of the AWS service you are using.

IAM administrator – If you're an IAM administrator, you might want to learn details about how you can write policies to manage access to AWS. To view example AWS identity-based policies that you can use in IAM, see the user guide of the AWS service you are using.

Authenticating with identities

Authentication is how you sign in to AWS using your identity credentials. You must be authenticated as the AWS account root user, an IAM user, or by assuming an IAM role.

You can sign in as a federated identity using credentials from an identity source like AWS IAM Identity Center (IAM Identity Center), single sign-on authentication, or Google/Facebook credentials. For more information about signing in, see How to sign in to your AWS account in the *AWS Sign-In User Guide*.

For programmatic access, AWS provides an SDK and CLI to cryptographically sign requests. For more information, see AWS Signature Version 4 for API requests in the *IAM User Guide*.

AWS account root user

When you create an AWS account, you begin with one sign-in identity called the AWS account *root user* that has complete access to all AWS services and resources. We strongly recommend that you don't use the root user for everyday tasks. For tasks that require root user credentials, see Tasks that require root user credentials in the *IAM User Guide*.

Federated identity

As a best practice, require human users to use federation with an identity provider to access AWS services using temporary credentials.

A *federated identity* is a user from your enterprise directory, web identity provider, or Directory Service that accesses AWS services using credentials from an identity source. Federated identities assume roles that provide temporary credentials.

For centralized access management, we recommend AWS IAM Identity Center. For more information, see What is IAM Identity Center? in the *AWS IAM Identity Center User Guide*.

IAM users and groups

An *IAM user* is an identity with specific permissions for a single person or application. We recommend using temporary credentials instead of IAM users with long-term credentials. For more information, see Require human users to use federation with an identity provider to access AWS using temporary credentials in the *IAM User Guide*.

An *IAM group* specifies a collection of IAM users and makes permissions easier to manage for large sets of users. For more information, see Use cases for IAM users in the *IAM User Guide*.

IAM roles

An *IAM role* is an identity with specific permissions that provides temporary credentials. You can assume a role by switching from a user to an IAM role (console) or by calling an AWS CLI or AWS API operation. For more information, see Methods to assume a role in the *IAM User Guide*.

IAM roles are useful for federated user access, temporary IAM user permissions, cross-account access, cross-service access, and applications running on Amazon EC2. For more information, see Cross account resource access in IAM 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 defines permissions when associated with an identity or resource. AWS evaluates these policies when a principal makes a request. Most policies are stored in AWS as JSON documents. For more information about JSON policy documents, see Overview of JSON policies in the *IAM User Guide*.

Using policies, administrators specify who has access to what by defining which **principal** can perform **actions** on what **resources**, and under what **conditions**.

By default, users and roles have no permissions. An IAM administrator creates IAM policies and adds them to roles, which users can then assume. IAM policies define permissions regardless of the method used to perform the operation.

Identity-based policies

Identity-based policies are JSON permissions policy documents that you attach to an identity (user, group, or role). These policies control what actions identities can perform, on which resources, and under what conditions. To learn how to create an identity-based policy, see Define custom IAM permissions with customer managed policies in the *IAM User Guide*.

Identity-based policies can be *inline policies* (embedded directly into a single identity) or *managed policies* (standalone policies attached to multiple identities). To learn how to choose between managed and inline policies, see Choose 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 include 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. You must specify a principal in a resource-based policy.

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 policy types that can set the maximum permissions granted by more common policy types:

- **Permissions boundaries** – Set the maximum permissions that an identity-based policy can grant to an IAM entity. For more information, see Permissions boundaries for IAM entities in the *IAM User Guide*.

- **Service control policies (SCPs)** – Specify the maximum permissions for an organization or organizational unit in AWS Organizations. For more information, see Service control policies in the *AWS Organizations User Guide*.
- **Resource control policies (RCPs)** – Set the maximum available permissions for resources in your accounts. For more information, see Resource control policies (RCPs) in the *AWS Organizations User Guide*.
- **Session policies** – Advanced policies passed as a parameter when creating a temporary session for a role or federated user. 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 AWS services work with IAM

To get a high-level view of how AWS services work with most IAM features, see AWS services that work with IAM in the *IAM User Guide*.

To learn how to use a specific AWS service with IAM, see the security section of the relevant service's User Guide.

Troubleshooting AWS identity and access

Use the following information to help you diagnose and fix common issues that you might encounter when working with AWS and IAM.

Topics

- I am not authorized to perform an action in AWS
- I am not authorized to perform iam:PassRole
- I want to allow people outside of my AWS account to access my AWS resources

I am not authorized to perform an action in AWS

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 `awes:GetWidget` permissions.

```
User: arn:aws:iam::123456789012:user/mateojackson is not authorized to perform:
 awes: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 `awes: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 AWS.

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 AWS. 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 AWS 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 AWS supports these features, see How AWS services work 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 Cross account resource access in IAM in the *IAM User Guide*.

Compliance Validation for this AWS Product or Service

Our new AWS sign-up experience is not designed for regulated workloads. If you're using our new AWS sign-up experience, but you want to use AWS for regulated workloads, you can sign up for AWS (advanced) or activate advanced features for your AWS environment.

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. For more information about your compliance responsibility when using AWS services, see AWS Security Documentation.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Resilience for this AWS Product or Service

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.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Infrastructure Security for this AWS Product or Service

This AWS product or service uses managed services, and therefore 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 this AWS Product or Service 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.

This AWS product or service follows the shared responsibility model through the specific Amazon Web Services (AWS) services it supports. For AWS service security information, see the AWS service security documentation page and AWS services that are in scope of AWS compliance efforts by compliance program.

Document history for the AWS SDK for Python Developer Guide

The following table describes the documentation releases for AWS SDK for Python.

Change	Description	Date
AWS SDK for Python Developer Preview release	Initial release of the AWS SDK for Python Developer Guide	August 26, 2026