Serverless Bot Framework: Implementation Guide
Copyright © 2020 Amazon Web Services, Inc. and/or its affiliates. All rights reserved.

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.
# Table of Contents

Home ............................................................................................................................................... 1  
Overview ........................................................................................................................................... 2  
Cost .......................................................................................................................................... 2  
   Example cost tables ........................................................................................................... 2  
   Architecture ............................................................................................................................... 4  
Solution Features ............................................................................................................................... 6  
   Supported Languages ................................................................................................................. 6  
   Conversation Types .................................................................................................................... 6  
      Static conversations ........................................................................................................ 6  
      Dynamic conversations .................................................................................................... 6  
   Conversation Logs ...................................................................................................................... 6  
   Sample Web App ....................................................................................................................... 6  
Template ........................................................................................................................................... 8  
Deployment ....................................................................................................................................... 9  
   What we'll cover ........................................................................................................................ 9  
   Optional step: Create weather API keys .................................................................................. 9  
   Step 1. Launch the stack ............................................................................................................ 9  
   Step 2. Interact with the sample web application ..................................................................... 11  
Security ........................................................................................................................................... 12  
   Amazon Cognito identity pools .................................................................................................. 12  
Resources ........................................................................................................................................ 13  
Appendix A: Architectural components ................................................................................................ 14  
   Brain Amazon S3 bucket ........................................................................................................... 14  
   TrainModel ............................................................................................................................. 15  
   Core function ........................................................................................................................... 16  
      Example 1- known entity .................................................................................................... 16  
      Example 2 - unknown entity .............................................................................................. 16  
Appendix B: Collection of operational metrics ...................................................................................... 19  
Source code ..................................................................................................................................... 21  
Contributors .................................................................................................................................... 22  
Revisions ......................................................................................................................................... 23  
Notices ............................................................................................................................................ 24
Serverless Bot Framework

AWS Implementation Guide

AWS Solutions Builder Team

July 2018 (last update (p. 23): November 2020)

This implementation guide discusses architectural considerations and configuration steps for deploying the Serverless Bot Framework in the Amazon Web Services (AWS) Cloud. It includes links to an AWS CloudFormation template that launches, configures, and runs the AWS services required to deploy this solution using AWS best practices for security and availability.

The guide is intended for IT infrastructure architects and developers who have practical experience architecting in the AWS Cloud.
Overview

The Serverless Bot Framework solution allows you to quickly implement sophisticated conversational chatbots and develop engaging and lifelike experiences for your customers. The solution automatically deploys, configures, and interacts with managed AWS services that apply machine learning algorithms to interact with end-users and turn chatbot text into lifelike speech.

The solution provides the following key features:

- A sample web application
- Customizable weather forecast and pizza ordering functions
- Support for static and dynamic conversational chatbot interaction
- Language processing for English, French, German, Italian, Portuguese, Russian, and Spanish

Cost

You are responsible for the cost of the AWS services used while running the Serverless Bot Framework solution, which can vary based on the following factors:

- Number of AWS API Gateway requests per month
- Number of AWS Lambda invocations per month
- Amazon DynamoDB write/read requests per month—the Serverless Bot Framework solution logs all interactions between the user and the solution's microservices to DynamoDB. It also logs the bot's context to maintain user's session information between microservices.
- Number of characters processed by Amazon Polly per month: the solution uses Amazon Polly to turn text into lifelike speech to allow the serverless bot to talk to users.
- Number of active users per month authenticated with Amazon Cognito.

This solution is based entirely on serverless AWS services. Therefore, when the solution is not in use, you only pay for data storage.

We recommend creating a budget through AWS Cost Explorer to help manage costs. For full details, see the pricing webpage for each AWS service used in this solution.

Example cost tables

The following tables provide an example monthly cost breakdown for deploying this solution with the default parameters in the US East (N. Virginia) Region (excludes free tier).

Example 1: 10 monthly active users, with 50,000 API requests per month

<table>
<thead>
<tr>
<th>AWS Service</th>
<th>Dimensions</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon API Gateway</td>
<td>50,000 requests / month</td>
<td>$0.17</td>
</tr>
<tr>
<td>AWS Lambda</td>
<td>500,000 invocations / month (avg 300 ms duration and 128 MB memory)</td>
<td>$0.41</td>
</tr>
</tbody>
</table>
### Example cost tables

<table>
<thead>
<tr>
<th>AWS Service</th>
<th>Dimensions</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon DynamoDB</td>
<td>500,000 write requests / month x ($1.25 / million)</td>
<td>$0.625</td>
</tr>
<tr>
<td></td>
<td>100,000 read requests / month X ($0.25 / million)</td>
<td>$0.25</td>
</tr>
<tr>
<td></td>
<td>Data storage: 1 GB x $0.25</td>
<td>$0.25</td>
</tr>
<tr>
<td>Amazon S3</td>
<td>Storage (0.5 GB) and 50,000 get requests / month</td>
<td>$0.03</td>
</tr>
<tr>
<td>Amazon CloudFront</td>
<td>Regional data transfer out to internet: first 10 TB</td>
<td>$0.085</td>
</tr>
<tr>
<td></td>
<td>Regional data transfer out to origin: all data transfer</td>
<td>$0.20</td>
</tr>
<tr>
<td></td>
<td>HTTPS Requests:50,000 requests / month X ($0.01/10,000 requests)</td>
<td>$0.05</td>
</tr>
<tr>
<td>Amazon Polly</td>
<td>average 10 characters / request: 10 x 50,000 requests / month x ($4.0 / 1 million characters)</td>
<td>$2.00</td>
</tr>
<tr>
<td>Amazon Cognito</td>
<td>10 user x ($0.0055 / monthly active users (MAUs))</td>
<td>$0.055</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td>$3.85 / month</td>
</tr>
</tbody>
</table>

**Example 2: 10,000 monthly active users, with 1,000,000 API requests per month**

<table>
<thead>
<tr>
<th>AWS Service</th>
<th>Dimensions</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon API Gateway</td>
<td>1,000,000 requests / month</td>
<td>$3.50</td>
</tr>
<tr>
<td>AWS Lambda</td>
<td>10,000,000 invocations / month (avg 300 ms duration and 128 MB memory)</td>
<td>$8.25</td>
</tr>
<tr>
<td>Amazon DynamoDB</td>
<td>10,000,000 write requests / month x ($1.25 / million)</td>
<td>$12.25</td>
</tr>
<tr>
<td></td>
<td>2,000,000 read requests / month X ($0.25 / million)</td>
<td>$0.50</td>
</tr>
<tr>
<td></td>
<td>Data storage: 10 GB x $0.25</td>
<td>$2.50</td>
</tr>
<tr>
<td>Amazon S3</td>
<td>Storage (0.5 GB) and 1,000,000 get requests / month</td>
<td>$0.41</td>
</tr>
<tr>
<td>Amazon CloudFront</td>
<td>Regional data transfer out to internet: first 10 TB</td>
<td>$0.085</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$0.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$1.00</td>
</tr>
</tbody>
</table>
### AWS Service Dimensions Cost

<table>
<thead>
<tr>
<th>AWS Service</th>
<th>Dimensions</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regional data transfer out to origin: all data transfer</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HTTPS Requests: 1,000,000 requests / month X ($0.01/10,000 requests)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amazon Polly</td>
<td>average 10 characters / request: 10 x 1,000,000 requests / month x ($4.0 / 1 million characters)</td>
<td>$40.00</td>
</tr>
<tr>
<td>Amazon Cognito</td>
<td>10,000 user x ($0.0055 / monthly active users (MAUs))</td>
<td>$55.00</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>$123.70 / month</strong></td>
</tr>
</tbody>
</table>

#### Architecture overview

Deploying this solution with the **default parameters** builds the following environment in the AWS Cloud.

**Figure 1: Serverless Bot Framework reference architecture**

The AWS CloudFormation template deploys the following services:

- **An Amazon API Gateway** endpoint where customers can send requests
- **AWS Lambda** functions that apply machine learning algorithms
- **Amazon Polly**, which turns text into lifelike speech
- **Amazon DynamoDB** tables to store conversation logs, interaction context, user feedback, and other tables required by the sample microservices
- **AWS Systems Manager** to securely store API keys
• **Amazon Simple Storage Service** (Amazon S3) buckets to store configuration files

When a request is made using a natural language string, the **Core** Lambda function is triggered. This function determines which of the solution's bot microservices can answer the request using the `knowledge.json` file that is stored in the Amazon S3 (Brain) bucket. To reduce the time necessary to process the requests, a Lambda function (TrainModel) pre-processes the request, generates a response, and uploads the files back to the Amazon S3 bucket. The **Core** Lambda function invokes Amazon Polly to generate an audio version of the bot's response before it is returned to the requestor. Conversation logs and interaction context are stored in an Amazon DynamoDB table. For information on the solution's architectural components, refer to Appendix A (p. 14).

This solution also deploys a sample web application into an Amazon S3 bucket configured for static website hosting. The S3 bucket is secured and can only be accessed through an Amazon CloudFront distribution. To access the sample web application, you must provide an admin email and a username when launching this solution. Amazon Cognito generates an email providing access information to the sample web application. If access credentials are not provided, Amazon Cognito will not authorize the clients to invoke the API Gateway endpoint. For more information, refer to Security (p. 12).

**Note**
AWS CloudFormation resources are created from AWS Cloud Development Kit (AWS CDK) constructs.
Solution features

Supported languages

The Serverless Bot Framework solution supports English, French, German, Italian, Portuguese, Russian, and Spanish.

Conversation types

This solution is configured to support two types of conversations:

**Static conversations**

Static conversations do not require access to external resources or any specialized AWS Lambda functions to implement the bot’s logic. The Core Lambda function will answer all requests by looking for instructions inside the knowledge.json file that is stored in the Brain Amazon Simple Storage Service (Amazon S3) bucket.

**Dynamic conversations**

Dynamic conversations are used for more complex interactions and invoke Lambda functions to perform additional tasks, such as interacting with backend resources to validate input, retrieve data, or invoke external APIs.

For more information about how to create bots, interpret input parameters, and define external Lambda functions that implement the bot’s logic, refer to Appendix A (p. 14).

Conversation logs

This solution stores conversation logs in an Amazon DynamoDB table (ConversationLogs). To use this feature, you must update the configs.json file located inside the Brain Amazon S3 bucket, and set the persistConversation variable to true.

Sample web application

This solution includes a sample web application that you can customize to create your own application that fits your business needs. The sample web application includes the following:

- The main messages exchanged between client devices and the API endpoint using both text and audio outputs.
- Client-side components to support audio and text inputs.
Note
The sample web application leverages a Google Chrome web browser to access your computer's microphone and transform input from speech to text format before sending it to the Serverless Bot Framework API endpoint.

- Five bots:
  - A static bot to answer the bot's name.
  - A static bot to provide help instructions.
  - A dynamic bot that can be used to get weather forecasts using an API or a random simulation. This bot can be used as a reference to securely use API keys with Lambda backed functions. In the Lambda function, which makes these API requests, the API key is cached for the duration of the Lambda container’s life, so changes to the API key in Systems Manager will not be reflected until the Lambda container is taken down.
  - A dynamic bot for user feedback, which is stored in DynamoDB. This can be used as a reference to implement multi-step Lambda backed functions.
  - A dynamic chatbot that demonstrates how to integrate a chatbot with a back-end database, such as AWS DynamoDB, to build an automated order taking service (in this example, a pizza ordering service). When the customer starts their order, the chatbot retrieves the pizza menu from the back-end database, and displays it to the customer. The chatbot interacts with the customer to extract order details (for example, type and size of the pizza) and confirms the order. The order history is stored in a DynamoDB table, which helps facilitate a personalized customer experience.
The solution uses AWS CloudFormation to automate the deployment of the Serverless Bot Framework in the AWS Cloud. It includes the following AWS CloudFormation template, which you can download before deployment:

`serverless-bot-framework.template`: Use this template to launch the Serverless Bot Framework and all associated components. The default configuration deploys an Amazon API Gateway endpoint, Amazon Cognito User and Identity Pools, AWS Lambda functions, Amazon Simple Storage Service (Amazon S3) buckets, Amazon DynamoDB tables, and Amazon Polly. You can also customize the template based on your specific needs.

**Note**
AWS CloudFormation resources are created from [AWS Cloud Development Kit (AWS CDK)](https://aws.amazon.com/cdk) constructs.
Automated deployment

Before you launch the automated deployment, please review the architectural components and solution features discussed in this guide. Follow the step-by-step instructions in this section to configure and deploy the Serverless Bot Framework solution into your account.

Time to deploy: Approximately 5 minutes

What we'll cover

The procedure for deploying this architecture on AWS consists of the following steps. For detailed instructions, follow the links for each step.

Optional step: Create weather API keys

To create these API keys, see AccuWeather APIs Getting Started or the OpenWeatherMap API guide.

Step 1. Launch the stack (p. 9)

- Launch the AWS CloudFormation template into your AWS account.
- Enter values for required parameters: Bot Name, Bot Language, Bot Gender, Admin Email, Admin Name, Weather API Provider, and Weather API key.
- Review the other template parameters and adjust if necessary.

Step 2. Interact with the sample web application (p. 11)

- Launch the web application.
- Sign in using the credentials provided in the email.
- Interact with the included bots.

Step 1. Launch the stack

This automated AWS CloudFormation template deploys the Serverless Bot Framework solution in the AWS Cloud.

Note

You are responsible for the cost of the AWS services used while running this solution. Refer to the Cost (p. 2) section for more details. For full details, refer to the pricing webpage for each AWS service used in this solution.

1. Sign in to the AWS Management Console and click the button below to launch the serverless-bot-framework AWS CloudFormation template.
You can also download the template as a starting point for your own implementation.

2. The template launches in the US East (N. Virginia) Region by default. To launch this solution in a different AWS Region, use the Region selector in the console navigation bar.

   **Note**
   This solution uses Amazon Cognito, which is currently available in specific AWS Regions only. Therefore, you must launch this solution in an AWS Region where Amazon Cognito is available. For the most current AWS service availability by Region, refer to AWS service offerings by Region.

3. On the Specify stack details page, assign a name to your solution stack.

4. Under Parameters, review the parameters for the template, and modify them as necessary. This solution uses the following default values.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Bot) Name</td>
<td>Jao</td>
<td>Specify the name that the bot responds to.</td>
</tr>
<tr>
<td>(Bot) Language</td>
<td>Portuguese</td>
<td>Choose the language that the bot converses in.</td>
</tr>
<tr>
<td>(Bot) Gender</td>
<td>&lt;Requires input&gt;</td>
<td>Choose the bot voice gender, male or female.</td>
</tr>
<tr>
<td>AdminName</td>
<td>&lt;Requires input&gt;</td>
<td>Username for signing in to the sample web application.</td>
</tr>
<tr>
<td>AdminEmail</td>
<td>&lt;Requires input&gt;</td>
<td>Email address to receive the credentials to sign in to the sample web application.</td>
</tr>
<tr>
<td>Weather API provider</td>
<td>Random Weather Generator</td>
<td>Select Random Weather Generator, AccuWeather, or OpenWeather.</td>
</tr>
<tr>
<td>Weather API Key</td>
<td>&lt;Optional input&gt;</td>
<td>API key from provider. Not required for Random Weather Generator.</td>
</tr>
</tbody>
</table>

5. Choose Next.

6. On the Configure stack options page, choose Next.

7. On the Review page, review and confirm the settings. Check the boxes acknowledging that the template will create AWS Identity and Access Management (IAM) resources and capabilities required.

8. Choose Create stack to deploy the stack.

   You can view the status of the stack in the AWS CloudFormation console in the Status column. You should see a status of CREATE_COMPLETE in approximately five minutes.

   **Note**
   In addition to the primary Core, TrainModel, weather-forecast, leave-feedback, and order-pizza AWS Lambda functions, this solution includes the solution-helper
and custom-resource functions, which run only during initial configuration or when resources are updated or deleted. Do not delete these functions as they are necessary to manage associated resources.

### Step 2. Interact with the sample web application

After this solution successfully deploys in your AWS account, you can test the web application and interact with the sample bots using the following procedure.

1. Navigate to the Outputs tab.
2. Select the provided Sample WebClient URL.

**Note**

This solution performs best if you use the sample web application in a Google Chrome web browser, which supports speech to text functionality.

3. Sign in and then interact with the sample bots and web application.

![Sample web application interface](image)

**Figure 2: Sample web application interface**
Security

When you build systems on AWS infrastructure, security responsibilities are shared between you and AWS. This shared model can reduce your operational burden as AWS operates, manages, and controls the components from the host operating system and virtualization layer down to the physical security of the facilities in which the services operate. For more information about security on AWS, visit the AWS Security Center.

Amazon Cognito identity pools

This solution leverages Amazon Cognito to authenticate user’s credentials. Then, it provides a temporary token which will be used by the client to authorize API calls to Amazon API Gateway. The client’s device will request the sample web application through Amazon CloudFront then the downloaded web application on the client’s device will communicate to API Gateway.

For information on protecting your resources, refer to Controlling and Managing Access to a REST API in API Gateway in the Amazon API Gateway Developer Guide and Identity and Access Management in Amazon S3 in the Amazon Simple Storage Service Developer Guide.
Additional resources

- AWS CloudFormation
- AWS Lambda
- Amazon Simple Storage Service
- Amazon API Gateway
- AWS Systems Manager
- Amazon DynamoDB
- Amazon Cognito
- Amazon Polly
- Amazon CloudWatch
Appendix A: Architectural components

Brain Amazon S3 bucket

When you deploy this solution, a `knowledge.json` file is loaded into the Amazon Simple Storage Service (Amazon S3) Brain bucket. Use this file as a reference to create your own custom knowledge properties.

The `knowledge` property of this file contains a list of bot definitions. The screenshots in Figure 3 describe the definition of weather forecast (dynamic) and say your name (static) bots:

```
"knowledge": [{
  "id": "2",
  "source": {
    "body": "Weather forecast sample",
    "api": "AWSAMPLE_WEATHER_FORECAST_BOT_API",
    "payload": "{"city": "[city]"},
    "parameters": {
      "payloadPosition": 0,
      "name": "city",
      "regexList": [
        / in (.+)/g,
        / on (.+)/g,
        / at (.+)/g,
        / for (.+)/g
      ],
      "noMatchAsk": [{
        "speech": "Which city?",
        "text": "Which city?"
      }]
    },
    "validationSuccessMessage": [{
      "speech": "Just a minute!",
      "text": "Just a minute!"
    }
  },
  "intents": [
    "what is the weather forecast",
    "weather forecast",
    "how is the weather"
  ]
},
"knowledge": [{
  "id": "1",
  "source": {
    "body": "what is your name",
    "response": [{
      "speech": "My name is %BOT_NAME%, how can I help you?",
      "text": "My name is %BOT_NAME%, how can I help you?"
    }]
  },
  "intents": [{
    "what is your name",
    "who are you"
  }]
}]
```

Figure 3: Sample bot definitions

Each bot definition contains the following elements:

- `id` [required]: The numeric identifier of the knowledge entry. Increment the `id` value to the next integer for each new entry.
Serverless Bot Framework Implementation Guide

TrainModel

- **_source**
- **_body** [required]: The name of the knowledge entry.
- **response** [required for static bots]: Which Core lambda function should answer in both speech (representation of what the bot will generate as audio result) and text (what the bot will display on the screen).
- **arn** [required for dynamic bots]: The Amazon Resource Number (ARN) of the AWS Lambda function that the Core lambda function should invoke.
- **payload** [optional]: Dynamic knowledge entries can include a payload key, which points to additional data parameters that the bot must gather from the user before returning a response. This property describes how this variable is sent to the bot Lambda.
- **parameters** [required if payload is defined]: This property describes how to extract parameters from input sentence and how to associate to the payload that is sent to the bot Lambda.

- **_intents** [required]: The expected inputs from users looking for this piece of knowledge. The more variations of phrases you include, the better the bot can correctly identify the intent of the user request. However, recommending all possible variations will result in a biased model. The list bellow describes weather forecast intents:

```
"_intents": [
  "what is the weather",
  "what is the temperature",
  "weather forecast",
  "how is the weather",
  "will it rain",
  ...
  "is it nice outside"
]
```

The Serverless Bot Framework analyzes incoming request strings against all intent lists stored in the Brain bucket. If the request matches an intent at a minimum percentage, the Core Lambda function begins the conversation for the associated knowledge entry.

TrainModel

The Serverless Bot Framework combines two well-known models: bag of words and support vector machine. Every time a knowledge.json file is uploaded to the Amazon S3 Brain bucket, the TrainModel lambda function is invoked to execute this classification algorithm and generate an equivalent pre-processed version of it.

To confirm that your bot was successfully trained on an updated knowledge file, check the last-modified dates of the brain-amcClassificationImpl.json file in the same bucket. If it is newer than your updated knowledge.json, your bot was trained. If no classified version is generated before that, check TrainModel logs, there might be syntactical issues in your JSON file. You can use any json lint tool, such as jsonlint.com, to help troubleshoot your file.

Before you upload an updated knowledge.json file, you must temporarily set the bot to training mode. This forces the Core Lambda function to clear its cache and reload all configurations. To set your bot to training mode, use the following steps:

1. Navigate to the AWS Lambda Console.
2. Locate the Core Lambda function for this solution.
3. Set the `forceCacheUpdate` environment variable to true.

4. Save your changes.

After you complete your tests, deactivate the training mode by repeating the process above, and setting the `forceCacheUpdate` to false.

Core function

The **Core** AWS Lambda function determines which of the solution's available bot microservices can answer the received request. This process is called intent resolution engine (IRE).

During an interaction, the Serverless Bot Framework identifies possible parameters in a user request. These are known values that satisfy payload parameters, such as city names like New York or Paris. This solution automatically populates an Amazon DynamoDB table (**EntityResolver**) with the values used during conversations. Note that you can manually add acceptable values.

The following examples provide end to end executions of **Core** function:

**Example 1 - known entity**

To return an accurate weather forecast an application requires two parameters: **city** and **date**. The associated knowledge entry **weather forecast** is a dynamic conversation type and includes the following payload:

```
{"city": "$0"},
```

If a user asks the application chatbot: “What is the weather forecast in New York?” the **Core** function parses the string using some basic logic to identify possible parameters as well as the intent of the request:

- what is the weather forecast in **new york**
  - possible parameters = new, york

- what is the **weather forecast** in
  - intent = weather forecast

The Lambda function then searches for known parameter matches in the Amazon DynamoDB table (**DynamoEntityResolver**). The bigram **new york** is a known value for the parameter type **city**, so the payload is satisfied. The **Core** function then invokes the associated Lambda function using the following payload:

```
{"city": "new york"},
```

The knowledge function attempts to run its code. In this example, it runs successfully because the parameter is valid, and it returns an answer to the **Core** function.

**Example 2 - unknown entity**

If a user asks the application chatbot: “What's the temperature in Nuuk?” The **Core** function processes the string as follows:
• what is the temperature in **nuuk**
  • possible parameter = nuuk

• what is the **temperature** in
  • intent = temperature

The function then searches for a known parameter match in Amazon DynamoDB. The unigram **nuuk** is an unknown but possible value for the parameter type **city**. The **Core** function then invokes the associated Lambda function using the following payload:

```
{"city": "nuuk"},
```

The knowledge function attempts to run its code. It runs successfully because the backend application has data for the city Nuuk. The function returns an answer to the **Core**. Nuuk did not exist as a known parameter in Amazon DynamoDB, but the code was able to run because it is a valid name of a city, so the **Core** function automatically adds Nuuk to the table as a known value for the parameter type **city**. This is how the solution learns new data.

If the knowledge function was not able to run its code, the solution assumes Nuuk was an invalid parameter and returns without a key, preventing the **Core** function from storing the bad input in Amazon DynamoDB.

**Weather forecast function**

The **weather-forecast** Lambda function retrieves city weather data based on the “weather forecast” intent from the chatbot. This solution supports using an external weather API to get weather data. AccuWeather and OpenWeather are supported; however, you can add your own custom functions to support different APIs.

**Note**

This AWS Lambda function caches a call to Systems Manager to retrieve the weather service API key during the creation of the Lambda container. This is reduces the costs associated with making calls to Systems Manager. However, if the Systems Manager key is updated, the Lambda function could still be using the old key. You can prevent this by modifying the code that retrieves the Systems Manager parameter into the `lambda_handler` function so that it refreshes every time a new call is made. Note that this modification might increase cost.

**Order pizza function**

The sample pizza ordering microservice demonstrates how to integrate a chatbot with a back-end database, for example, DynamoDB, to provide a rich and personalized user experience. The **order-pizza** Lambda function handles all the logic of the pizza ordering microservice.

After an intent to order a pizza is resolved by the **Core** function, the **order-pizza** function is triggered. A customer expresses their intent to order pizza using phrases such as: “order pizza”, “I would like to order pizza”, “pizza ordering”, “pizza”, etc.

The **order-pizza** function begins by extracting the customer's email from the authentication data passed by the API gateway. The email is used as the customerId (the unique identifier) to keep a history of customers' orders and facilitate a personalized user experience. The function then runs the following flow:

1. The function uses the customerId to look up the customer's order history in the **pizza-orders** DynamoDB table. If the customer has previous order history, the function moves to step 2. Otherwise, it moves to step 3.
2. The function asks the customer if they would like to place an order similar to their previous orders. If the customer answers "yes", the function moves to step 4. If the customer answers "no", the function moves to step 3.

3. The function displays the pizza menu (retrieved from the pizza-menus DynamoDB table), and begins interacting with the customer, mimicking the way a human agent would interact with the customer to obtain order details (for example, pizza type, pizza size, number of pizzas, and type of pizza crust).

4. Finally, the order-pizza function provides a summary of the order to the customer and asks for a confirmation to place the order. If the customer answers "yes", the function generates a unique order’s number, stores the order in the pizza-orders DynamoDB table, and returns the order’s number, with the order’s total bill, to the customer with a closing message to end the interaction with the pizza-ordering microservice. If the customer answers "no", the function ends the interaction with a closing message and does not store the order.

DynamoDB tables

The sample pizza ordering microservices uses two DynamoDB tables: pizza-menus and pizza-orders.

The partition key of the pizza-menus table is menuID, which is a unique identifier used to look up different pizza menus offered by a restaurant. To initialize the pizza-menus table, the solution provides a sample menu file pizza-menus.json, which is automatically uploaded to the SampleWebClient Amazon S3 bucket. You can edit the menus file or upload a custom menus file to the Brain Amazon S3 bucket. To re-initialize the pizza-menus table with custom menus, four environment variables of the order-pizza function must be modified as follows:

- RE_INITIALIZE_MENUS_TABLE = true (default false)
- PIZZA_MENUS_INITIALIZATION_BUCKET=<s3-bucket-of-the-json-file>
- PIZZA_MENUS_INITIALIZATION_FILE=<s3-key-of-the-json-file> (default pizza-menu.json)
- PIZZA_MENU_ID=<id-of-new-menu-to-used-by-chatbot> (default main-menu-1).

Alternatively, you can directly edit the pizza-menus DynamoDB table through the AWS Console or a custom application.

The pizza-orders table uses orderId, a unique and randomly generated ID (for example, 6320-375719-2099), for the partition key. A unique and randomly generated partition key is critical for uniform partitions and avoiding hotspots. Partitioning enables the application to scale and handle a large volume of requests. The table also uses a global index with customerId as the partition key, and orderTimestamp as the sort key. The index is used by the order-pizza function to look up the last order of a customer.
Appendix B: Collection of operational metrics

This solution includes an option to send anonymous operational metrics to AWS. We use this data to better understand how customers use this solution to improve the services and products that we offer. When enabled, the following information is collected and sent to AWS:

- **Solution ID**: The AWS solution identifier
- **Unique ID (UUID)**: Randomly generated, unique identifier for each deployment
- **Timestamp**: Data-collection timestamp
- **Region**: The AWS Region where this solution is deployed
- **Version**: The AWS solution version identifier
- **Action**: The action this solution takes (create, update, or delete)
- **Bot Gender**: The bot gender (male or female)
- **Bot Language**: The language selected
- **Bot Voice**: Amazon Polly voice identifier used

When a new knowledge file is processed by the `TrainModel` Lambda function:

- **Total Knowledges**: The number of knowledges trained
- **Total Intents**: The total number of intents processed
- **Max Intents By Knowledge**: The size of the biggest list of intents among all knowledge trained
- **Min Intents By Knowledge**: The size of the smallest list of intents among all knowledge trained
- **Average Intents By Knowledge**: The average number of intents defined for every knowledge trained

Note that AWS will own the data gathered via this survey. Data collection will be subject to the AWS Privacy Policy. To opt out of this feature, complete one of the following tasks:

a) Modify the AWS CloudFormation template mapping section as follows:

```json
"Send" : {
   "SendAnonymousUsageData" : { "Data" : "Yes" }
},
```

to

```json
"Send" : {
   "SendAnonymousUsageData" : { "Data" : "No" }
},
```

OR

b) After this solution launches, go to the AWS Lambda console, search for the function whose name structure is similar to `<stack_name>-TrainModel-<unique_id>` and set the
SEND_ANONYMOUS_USAGE_DATA environment variable to No. Repeat the same process for <stack_name>-CustomResource-<unique_id> function.
Source code

Visit our GitHub repository to download the templates and scripts for this solution, and to share your customizations with others. The Serverless-bot-framework templates are generated using the AWS Cloud Development Kit (AWS CDK) and AWS Solutions Constructs. Refer to the README.md file for more information.
Contributors

The following individuals contributed to this document:

- Renato Barbosa
- Fernando Sapata
- Heitor Vital
- Mohsen Ansari
- Zain Kabani
- Tarek Abdunabi
## Document revisions

<table>
<thead>
<tr>
<th>Date</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>July 2018</td>
<td>Initial release</td>
</tr>
<tr>
<td>March 2020</td>
<td>Updated architecture to enhance security; updated solution to support Python 3.x and NodeJS 12.x</td>
</tr>
<tr>
<td>July 2020</td>
<td>Updated solution to include weather a forecast Lambda function. For information about updates and changes for v1.2.0, refer to the changelog file in the GitHub repository.</td>
</tr>
<tr>
<td>October 2020</td>
<td>Updated the solution to include a pizza ordering microservice. For information about updates and changes for v1.3.0, refer to the changelog file in the GitHub repository.</td>
</tr>
<tr>
<td>November 2020</td>
<td>AWS CDK and AWS Solutions Constructs to create AWS CloudFormation template. For information about updates and changes for v1.4.0, refer to the changelog file in the GitHub repository.</td>
</tr>
</tbody>
</table>
Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents AWS current product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. AWS responsibilities and liabilities to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Serverless Bot Framework is licensed under the terms of the Apache License Version 2.0 available at Classless Inter-Domain Routing (CIDR).