Agents for Amazon Bedrock

Agents for Amazon Bedrock offers you the ability to build and configure autonomous agents in your application. An agent helps your end-users complete actions based on organization data and user input. Agents orchestrate interactions between foundation models (FMs), data sources, software applications, and user conversations. In addition, agents automatically call APIs to take actions and invoke knowledge bases to supplement information for these actions. Developers can save weeks of development effort by integrating agents to accelerate the delivery of generative artificial intelligence (generative AI) applications .

With agents, you can automate tasks for your customers and answer questions for them. For example, you can create an agent that helps customers process insurance claims or an agent that helps customers make travel reservations. You don't have to provision capacity, manage infrastructure, or write custom code. Amazon Bedrock manages prompt engineering, memory, monitoring, encryption, user permissions, and API invocation.

Agents perform the following tasks:

- Extend foundation models to understand user requests and break down the tasks that the agent must perform into smaller steps.
- Collect additional information from a user through natural conversation.
- Take actions to fulfill a customer's request by making API calls to your company systems.
- Augment performance and accuracy by querying data sources.

To use an agent, you perform the following steps:

1. (Optional) Create a knowledge base to store your private data in that database. For more information, see Knowledge bases for Amazon Bedrock.
2. Configure an agent for your use case and add actions the agent can perform. To define how the agent handles the actions, write Lambda functions in a programming language of your choice.
3. Associate a knowledge base with the agent to augment the agent's performance. For more information, see Create an agent in Amazon Bedrock.
4. (Optional) To customize the agent's behavior to your specific use-case, modify prompt templates for the pre-processing, orchestration, knowledge base response generation, and

post-processing steps that the agent performs. For more information, see Advanced prompts in Amazon Bedrock.

5. Test your agent in the Amazon Bedrock console or through API calls to the `TSTALIASID`. Modify the configurations as necessary. Use traces to examine your agent's reasoning process at each step of its orchestration. For more information, see Test an Amazon Bedrock agent and Trace events in Amazon Bedrock.
6. When you have sufficiently modified your agent and it's ready to be deployed to your application, create an alias to point to a version of your agent. For more information, see Deploy an Amazon Bedrock agent.
7. Set up your application to make API calls to your agent alias.
8. Iterate on your agent and create more versions and aliases as necessary.

Topics

How Agents for Amazon Bedrock works

Agents for Amazon Bedrock consists of the following two main sets of API operations to help you set up and run an agent:

- Build-time API operations to create, configure, and manage your agents and their related resources
- Runtime API operations to invoke your agent with user input and to initiate orchestration to carry out a task.

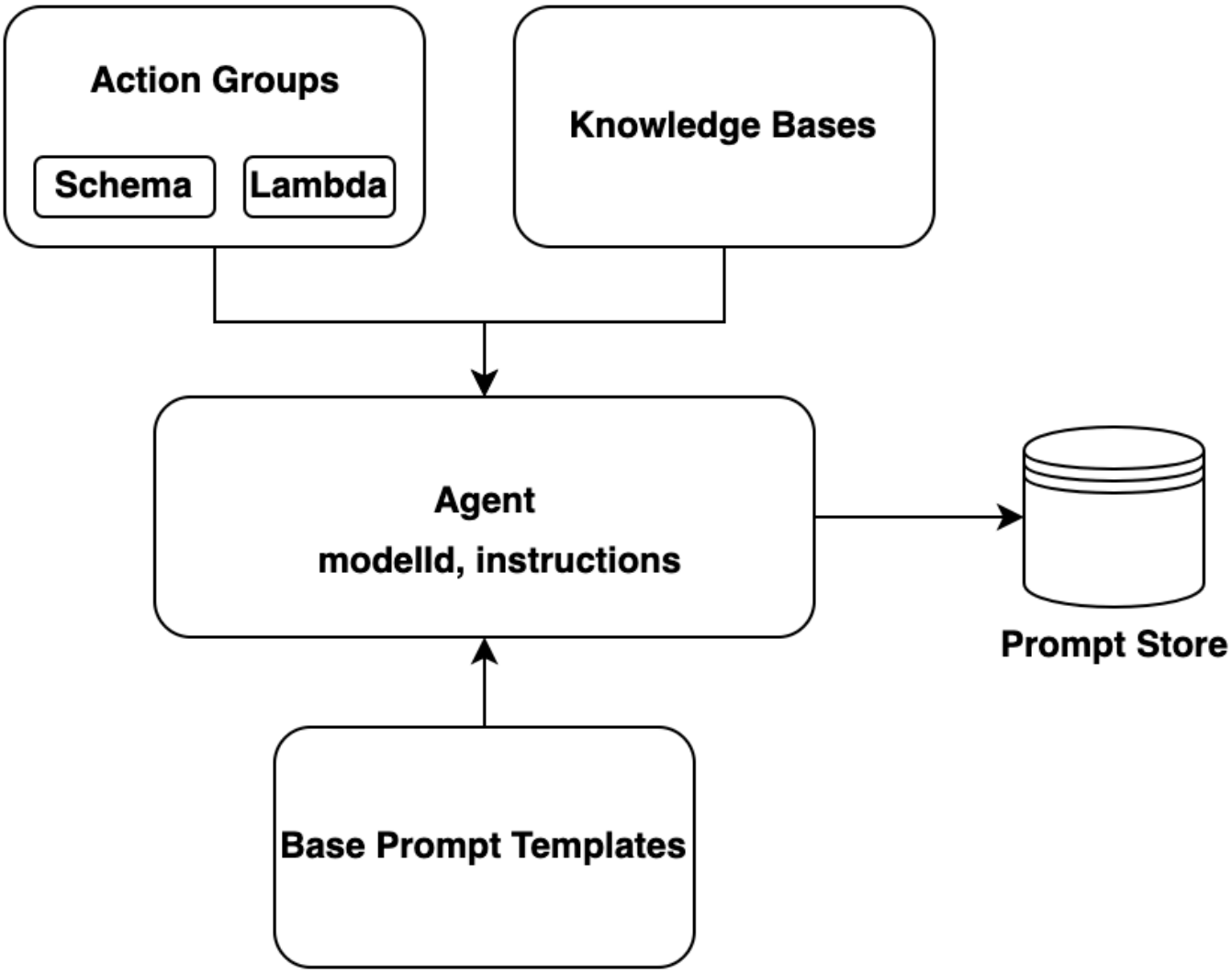
Build-time configuration

An agent consists of the following components:

- **Foundation model** – You choose a foundation model (FM) that the agent invokes to interpret user input and subsequent prompts in its orchestration process. The agent also invokes the FM to generate responses and follow-up steps in its process.
- **Instructions** – You write instructions that describe what the agent is designed to do. With advanced prompts, you can further customize instructions for the agent at every step of orchestration and include Lambda functions to parse each step's output.
- **Action groups** (Optional) – You define the actions that the agent should perform through providing the following resources:
  - An OpenAPI schema to define the API operations that the agent can invoke to perform its tasks.
  - A Lambda function with the following input and output:
    - Input – The API operation and parameters identified during orchestration.
    - Output – The result of the API invocation.
- **Knowledge bases** (Optional) – Associate knowledge bases with an agent. The agent queries the knowledge base for extra context to augment response generation and input into steps of the orchestration process.
- **Prompt templates** – Prompt templates are the basis for creating prompts to be provided to the FM. Agents for Amazon Bedrock exposes the default four base prompt templates that are used during the pre-processing, orchestration, knowledge base response generation, and post-processing. You can optionally edit these base prompt templates to customize your agent's behavior at each step of its sequence. You can also turn off steps for troubleshooting purposes or if you decide that a step is unnecessary. For more information, see Advanced prompts in Amazon Bedrock.

At build-time, all these components are gathered to construct base prompts for the agent to perform orchestration until the user request is completed. With advanced prompts, you can modify these base prompts with additional logic and few-shot examples to improve accuracy for each step of agent invocation. The base prompt templates contain instructions, action descriptions, knowledge base descriptions, and conversation history, all of which you can customize to modify the agent to meet your needs. You then *prepare* your agent, which packages all the components of the agents, including security configurations. Preparing the agent brings it into a state where it

can be tested in runtime. The following image shows how build-time API operations construct your agent.



Runtime process

Runtime is managed by the InvokeAgent API operation. This operation starts the agent sequence, which consists of the following three main steps.

1. **Pre-processing** – Manages how the agent contextualizes and categorizes user input and can be used to validate input.
2. **Orchestration** – Interprets the user input, invokes action groups and queries knowledge bases, and returns output to the user or as input to continued orchestration. Orchestration consists of the following steps:

a. The agent interprets the input with a foundation model and generates a *rationale* that lays out the logic for the next step it should take.

b. The agent invokes action groups and queries knowledge bases (**Knowledge base response generation**) to retrieve additional context and summarize the data to augment its generation.

c. The agent generates an output, known as an *observation*, from invoking action groups and summarizing results from knowledge bases. The agent uses the observation to augment the base prompt, which is then interpreted with a foundation model. The agent then determines if it needs to reiterate the orchestration process.

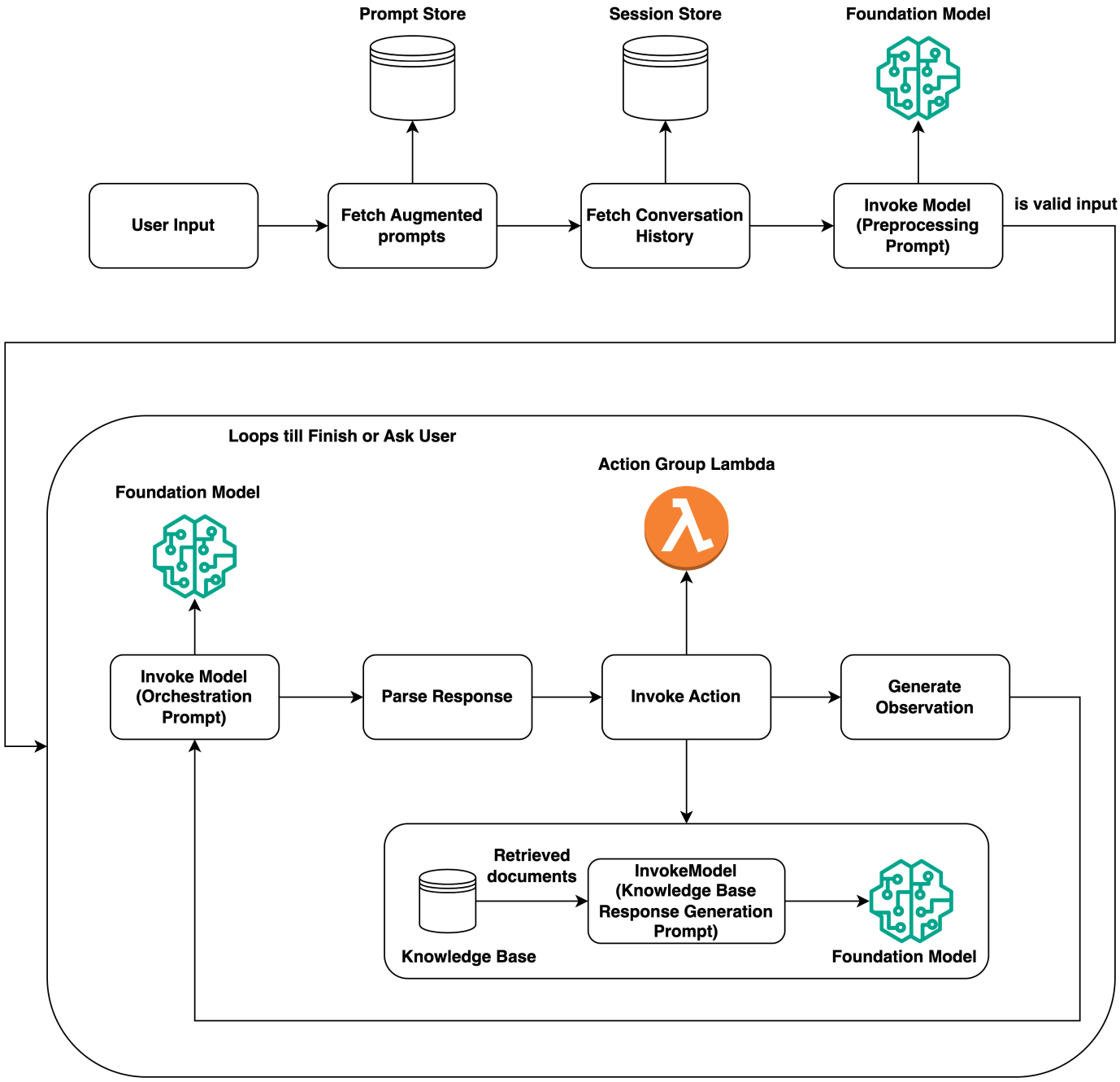
d. This loop continues until the agent returns a response to the user or until it needs to prompt the user for extra information.

During orchestration, the base prompt template is augmented with the agent instructions, action groups, and knowledge bases that you added to the agent. Then, the augmented base prompt is used to invoke the FM. The FM predicts the best possible steps and trajectory to fulfill the user input. At each iteration of orchestration, the FM predicts the API operation to invoke or the knowledge base to query.

3. **Post-processing** – The agent formats the final response to return to the user. This step is turned off by default.

When you invoke your agent, you can turn on a **trace** at runtime. With the trace, you can track the agent's rationale, actions, queries, and observations at each step of the agent sequence. The trace includes the full prompt sent to the foundation model at each step and the outputs from the foundation model, API responses, and knowledge base queries. You can use the trace to understand the agent's reasoning at each step. For more information, see Trace events in Amazon Bedrock

As the user session with the agent continues through more `InvokeAgent` requests, the conversation history is preserved. The conversation history continually augments the orchestration base prompt template with context, helping improve the agent's accuracy and performance. The following diagram shows the agent's process during runtime:



Supported regions and models for Agents for Amazon Bedrock

Agents for Amazon Bedrock is supported in the following regions:

Region
US East (N. Virginia)
US West (Oregon)

You can use Agents for Amazon Bedrock with the following models:

Model name	Model ID
Anthropic Claude Instant v1	anthropic.claude-instant-v1
Anthropic Claude v2.0	anthropic.claude-v2
Anthropic Claude v2.1	anthropic.claude-v2:1

Prerequisites for Agents for Amazon Bedrock

Ensure that your IAM role has the necessary permissions to perform actions related to Agents for Amazon Bedrock.

An agent uses action groups and knowledge bases to help your customer perform tasks. Following is a short description of each type of resource:

- **Action group** – Defines an action that the agent can help the user perform. Includes the APIs that can be called, how to handle the action, and how to return the response.
- **Knowledge base** – Provides a repository of information that the agent can query to answer customer queries and improve its generated responses.

Before creating an agent, review the following prerequisites and determine which ones you need to fulfill:

1. Set up an action group. For your agent to orchestrate API calls to your systems, you must add at least one action group. You can skip this prerequisite if you want to add an action group later or if you plan to have no action groups for your agent.

2. Set up a knowledge base. To improve responses to customer queries by using private data sources, you can associate at least one knowledge base. You can skip this prerequisite if you plan for your agent to have no knowledge bases associated with it
3. Create a custom AWS Identity and Access Management (IAM) service role for your agent with the proper permissions. You can skip this prerequisite if you plan to use the AWS Management Console to automatically create a service role for you.

Create an agent in Amazon Bedrock

To create an agent with Amazon Bedrock, you set up the following components:

- The configuration of the agent, which defines the purpose of the agent and indicates the foundation model (FM) that it uses to generate prompts and responses.
- (Optional) Action groups that define what actions the agent is designed to perform.
- (Optional) A knowledge base of data sources to augment the generative capabilities of the agent.

You can create an agent in either the console or the API. Select the tab corresponding to your method of choice and follow the steps.

Console

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane.
3. In the **Agents** section, choose **Create Agent**.
4. On the **Provide Agent details**, set up the following configurations:

    a. For the **Agent name** section, enter a name for the agent and an optional description.

    b. For the **User input** section, choose whether to allow the agent to request more information from the user if it doesn't have enough information.

    - If you choose **Yes**, the agent returns an Observation reprompting the user for more information if it needs to invoke an API in an action group, but doesn't have enough information to complete the API request.

- If you choose **No**, the agent doesn't request the user for additional details and instead informs the user that it doesn't have enough information to complete the task.

c. For the **IAM permissions** section, for **Agent resource role**, choose a service role. To let Amazon Bedrock create the service role on your behalf, choose **Create and use a new service role**. To use a custom role that you created previously, choose **Use an existing service role**. The role must begin with the prefix `AmazonBedrockExecutionRoleForAgents_`.

> **Note**
>
> The service role that Amazon Bedrock creates for you doesn't include permissions for features that are in preview. To use these features, attach the correct permissions to the service role.

d. (Optional) By default, AWS encrypts agent resources with an AWS managed key. To encrypt your agent with your own customer managed key, for the **KMS key selection** section, select **Customize encryption settings (advanced)**. To create a new key, select **Create an AWS KMS key** and then refresh this window. To use an existing key, select a key for **Choose an AWS KMS key**.

e. For the **Idle session timeout** section, choose a duration of time. If the user hasn't responded in the same session after this amount of time has elapsed, Amazon Bedrock no longer maintains the conversation history. Conversation history is used to both resume an interaction and to augment responses with context from the conversation.

f. (Optional) To associate tags with this agent, for the **Tags – optional** section, choose **Add new tag** and provide a key-value pair.

g. When you are done setting up the agent configuration, select **Next**.

5. On the **Select model** page, set up the following configurations for the model that you want your agent to use in orchestration:

a. For the **Model details** section, choose a model provider and then choose a model in the dropdown menu.

b. In **Instructions for the Agent**, enter details to tell the agent what it should do and how it should interact with users. The instructions replace the $instructions$ placeholder in the orchestration prompt template. Following is an example of instructions:

```
You are an office assistant in an insurance agency. You are friendly and
 polite. You help with managing insurance claims and coordinating pending
 paperwork.
```

c. When you're done configuring the FM for your agent, choose **Next**.

6. (Optional) On the **Add action groups** page, add action groups to your agent. If you want to add action groups later or if your agent doesn't need an action group, choose **Next** to skip this step. Otherwise, do the following steps:

   a. For **Enter action group name**, provide a name for your action group. For **Description - optional** enter a description of the action group.

   b. For **Select Lambda function**, choose a Lambda function that you created previously in AWS Lambda. The Lambda function provides the business logic that is performed upon invoking the action. Choose the version of the function to use.

   > **Note**
   >
   > To allow the Amazon Bedrock service principal to access the Lambda function, attach a resource-based policy to the Lambda function to allow the Amazon Bedrock service principal to access the Lambda function.

   c. For **Select API schema**, provide a link to the Amazon S3 URI of the OpenAPI schema with the API description, structure, and parameters for the action group.

   > **Note**
   >
   > If you prefer to use the console text editor to create the API schema, skip this step and add an action group after you have created the agent.

   d. To set up another action group for your agent, select **Add another action group**. When you're done with adding action groups, choose **Next**.

7. (Optional) On the **Add knowledge bases** page, associate knowledge bases with your agent. If you want to add knowledge bases later or if your agent doesn't need a knowledge base, choose **Next** to skip this step. Otherwise, do the following steps:

a. If you haven't yet created any knowledge bases, follow the instructions at Create a knowledge base. Otherwise, for **Select knowledge base**, choose a knowledge base from the dropdown menu.

b. For **Knowledge base instructions for the agent**, enter instructions to describe how the agent should use the knowledge base. For example, you might use the following text for a knowledge base called *`Domain name system details`*:

```
Use this knowledge base whenever you are creating a DNS record
```

c. To set up another knowledge base for your agent, choose **Add another knowledge base**. When you're done adding knowledge bases, choose **Next**.

8. On the **Review and create** page, review the configuration of your agent and choose **Edit** for any sections that you want to change. When you're ready to create the agent, choose **Create**. When the process finishes, a banner appears at the top to inform you that the agent was successfully created.

API

To create an agent, send a CreateAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint.

See code examples

The following list describes the fields in the request:

- Minimally, provide the following required fields:

Field	Short description
agentName	Name for the agent
agentResourceRoleArn	ARN of the service role for the agent
foundationModel	Foundation model for the agent to orchestrate with

- The following fields are optional but recommended:

Field	Short description
description	Describes what the agent does
instruction	Instructions to tell the agent what to do. Used in the orchestration prompt template.

- The following fields can be specified for security purposes:

Field	Short description
clientToken	Identifier to ensure the API request completes only once.
customerEncryptionKeyArn	ARN of a KMS key to encrypt agent resources
idleSessionTTLInSeconds	Duration after which the agent ends the session and deletes any stored information.

- To customize your agent's behavior by overriding the default prompt templates, include a `promptOverrideConfiguration` object. For more information, see Advanced prompts in Amazon Bedrock.
- To attach tags to your agent, use the `Tags` field. For more information, see Tag resources.

If your agent fails to be created, the CreateAgent object in the response returns a list of `failureReasons` and a list of `recommendedActions` for you to troubleshoot.

Create an action group for an Amazon Bedrock agent

An action group defines an action that the agent can help the user perform. You define an action group by specifying APIs in your systems that can be called and by writing a Lambda function to handle the action and how to return the response. To create an action group, prepare the following components:

- Set up an OpenAPI schema with the API description, structure, and parameters for the action group. You can add the API schema to the action group in one of the following ways:
  - Upload the schema that you create to an Amazon Simple Storage Service (Amazon S3) bucket.
  - Write the schema in the inline OpenAPI schema editor in the AWS Management Console when you add the action group. This option is only available after the agent that the action group belongs to has already been created.
- Create a Lambda function that defines the business logic for the action group.

To learn more about the components of an action group and how to create the action group after you set it up, select from the following topics:

Topics

Define OpenAPI schemas for your agent's action groups in Amazon Bedrock

When you create an action group in Amazon Bedrock, you must define the API operations that the agent can invoke. To define the API operations, create an OpenAPI schema in JSON or YAML format. You can create OpenAPI schema files and upload them to Amazon Simple Storage Service (Amazon S3). Alternatively, you can use the OpenAPI text editor in the console, which will validate your schema. After you create an agent, you can use the text editor when you add an action group to the agent or edit an existing action group. For more information, see Edit an agent.

For more information about API schemas, see the following resources:

- For more details about OpenAPI schemas, see OpenAPI specification on the Swagger website.
- For best practices in writing API schemas, see Best practices in API design on the Swagger website.

The following is the general format of an OpenAPI schema for an action group.

```
{
```

```
    "openapi": "3.0.0",
    "paths": {
        "/path": {
            "method": {
                "description": "string",
                "operationId": "string",
                "parameters": [ ... ],
                "requestBody": { ... },
                "responses": { ... }
            }
        }
    }
}
```

The following list describes fields in the OpenAPI schema

- `openapi` – (Required) The version of OpenAPI that's being used. This value must be `"3.0.0"` or higher for the action group to work.
- `paths` – (Required) Contains relative paths to individual endpoints. Each path must begin with a forward slash (/).
- `method` – (Required) Defines the method to use.

Minimally, each method requires the following fields:

- `description` – A description of the API operation. Use this field to inform the agent when to call this API operation and what the operation does.
- `responses` – Contains properties that the agent returns in the API response. The agent uses the response properties to construct prompts, accurately process the results of an API call, and determine a correct set of steps for performing a task. The agent can use response values from one operation as inputs for subsequent steps in the orchestration.

The fields within the following two objects provide more information for your agent to effectively take advantage of your action group. For each field, set the value of the `required` field to `true` if required and to `false` if optional.

- `parameters` – Contains information about parameters that can be included in the request.
- `requestBody` – Contains the fields in the request body for the operation. Don't include this field for `GET` and `DELETE` methods.

To learn more about a structure, select from the following tabs.

responses

```
"responses": {
    "200": {
        "content": {
            "<media type>": {
                "schema": {
                    "properties": {
                        "<property>": {
                            "type": "string",
                            "description": "string"
                        },
                        ...
                    }
                }
            }
        },
    },
    ...
}
```

Each key in the `responses` object is a response code, which describes the status of the response. The response code maps to an object that contains the following information for the response:

- `content` – (Required for each response) The content of the response.
- *`<media type>`* – The format of the response body. For more information, see Media types on the Swagger website.
- `schema` – (Required for each media type) Defines the data type of the response body and its fields.
- `properties` – (Required if there are `items` in the schema) Your agent uses properties that you define in the schema to determine the information it needs to return to the end user in order to fulfill a task. Each property contains the following fields:
  - `type` – (Required for each property) The data type of the response field.
  - `description` – (Optional) Describes the property. The agent can use this information to determine the information that it needs to return to the end user.

parameters

```
"parameters": [
    {
        "name": "string",
        "description": "string",
        "required": boolean,
        "schema": {
            ...
        }
    },
    ...
]
```

Your agent uses the following fields to determine the information it must get from the end user to perform the action group's requirements.

- `name` – (Required) The name of the parameter.
- `description` – (Required) A description of the parameter. Use this field to help the agent understand how to elicit this parameter from the agent user or determine that it already has that parameter value from prior actions or from the user's request to the agent.
- `required` – (Optional) Whether the parameter is required for the API request. Use this field to indicate to the agent whether this parameter is needed for every invocation or if it's optional.
- `schema` – (Optional) The definition of input and output data types. For more information, see Data Models (Schemas) on the Swagger website.

requestBody

Following is the general structure of a `requestBody` field:

```
"requestBody": {
    "required": boolean,
    "content": {
        "<media type>": {
            "schema": {
                "properties": {
                    "<property>": {
                        "type": "string",
```

```
                        "description": "string"
                    },
                    ...
                }
            }
        }
    }
}
```

The following list describes each field:

- `required` – (Optional) Whether the request body is required for the API request.
- `content` – (Required) The content of the request body.
- *`<media type>`* – (Optional) The format of the request body. For more information, see Media types on the Swagger website.
- `schema` – (Optional) Defines the data type of the request body and its fields.
- `properties` – (Optional) Your agent uses properties that you define in the schema to determine the information it must get from the end user to make the API request. Each property contains the following fields:
  - `type` – (Optional) The data type of the request field.
  - `description` – (Optional) Describes the property. The agent can use this information to determine the information it needs to return to the end user.

Example API schema

The following example API schema defines a group of API operatins that help handle insurance claims. Three APIs are defined as follows:

- `getAllOpenClaims` – Your agent can use the `description` field to determine that it should call this API operation if a list of open claims is needed. The `properties` in the `responses` specify to return the ID and the policy holder and the status of the claim. The agent returns this information to the agent user or uses some or all of the response as input to subsequent API calls.
- `identifyMissingDocuments` – Your agent can use the `description` field to determine that it should call this API operation if missing documents must be identified for an insurance claim. The `name`, `description`, and `required` fields tell the agent that it must elicit the unique identifier of the open claim from the customer. The `properties` in the `responses` specify to

return the IDs of the open insurance claims. The agent returns this information to the end user or uses some or all of the response as input to subsequent API calls.

- `sendReminders` – Your agent can use the `description` field to determine that it should call this API operation if there is a need to send reminders to the customer. For example, a reminder about pending documents that they have for open claims. The `properties` in the `requestBody` tell the agent that it must find the claim IDs and the pending documents. The `properties` in the `responses` specify to return an ID of the reminder and its status. The agent returns this information to the end user or uses some or all of the response as input to subsequent API calls.

```
{
    "openapi": "3.0.0",
    "info": {
        "title": "Insurance Claims Automation API",
        "version": "1.0.0",
        "description": "APIs for managing insurance claims by pulling a list of open
 claims, identifying outstanding paperwork for each claim, and sending reminders to
 policy holders."
    },
    "paths": {
        "/claims": {
            "get": {
                "summary": "Get a list of all open claims",
                "description": "Get the list of all open insurance claims. Return all
 the open claimIds.",
                "operationId": "getAllOpenClaims",
                "responses": {
                    "200": {
                        "description": "Gets the list of all open insurance claims for
 policy holders",
                        "content": {
                            "application/json": {
                                "schema": {
                                    "type": "array",
                                    "items": {
                                        "type": "object",
                                        "properties": {
                                            "claimId": {
                                                "type": "string",
                                                "description": "Unique ID of the
 claim."
```

```
                                            },
                                            "policyHolderId": {
                                                "type": "string",
                                                "description": "Unique ID of the policy
holder who has filed the claim."
                                            },
                                            "claimStatus": {
                                                "type": "string",
                                                "description": "The status of the
claim. Claim can be in Open or Closed state"
                                            }
                                        }
                                    }
                                }
                            }
                        }
                    }
                }
            },
            "/claims/{claimId}/identify-missing-documents": {
                "get": {
                    "summary": "Identify missing documents for a specific claim",
                    "description": "Get the list of pending documents that need to be
uploaded by policy holder before the claim can be processed. The API takes in only one
claim id and returns the list of documents that are pending to be uploaded by policy
holder for that claim. This API should be called for each claim id",
                    "operationId": "identifyMissingDocuments",
                    "parameters": [{
                        "name": "claimId",
                        "in": "path",
                        "description": "Unique ID of the open insurance claim",
                        "required": true,
                        "schema": {
                            "type": "string"
                        }
                    }],
                    "responses": {
                        "200": {
                            "description": "List of documents that are pending to be
uploaded by policy holder for insurance claim",
                            "content": {
                                "application/json": {
                                    "schema": {
```

```
                                    "type": "object",
                                    "properties": {
                                        "pendingDocuments": {
                                            "type": "string",
                                            "description": "The list of pending
documents for the claim."
                                        }
                                    }
                                }
                            }
                        }

                    }
                }
            }
        },
        "/send-reminders": {
            "post": {
                "summary": "API to send reminder to the customer about pending
documents for open claim",
                "description": "Send reminder to the customer about pending documents
for open claim. The API takes in only one claim id and its pending documents at a
time, sends the reminder and returns the tracking details for the reminder. This API
should be called for each claim id you want to send reminders for.",
                "operationId": "sendReminders",
                "requestBody": {
                    "required": true,
                    "content": {
                        "application/json": {
                            "schema": {
                                "type": "object",
                                "properties": {
                                    "claimId": {
                                        "type": "string",
                                        "description": "Unique ID of open claims to
send reminders for."
                                    },
                                    "pendingDocuments": {
                                        "type": "string",
                                        "description": "The list of pending documents
for the claim."
                                    }
                                },
                                "required": [
```

```
                                    "claimId",
                                    "pendingDocuments"
                                ]
                            }
                        }
                    }
                },
                "responses": {
                    "200": {
                        "description": "Reminders sent successfully",
                        "content": {
                            "application/json": {
                                "schema": {
                                    "type": "object",
                                    "properties": {
                                        "sendReminderTrackingId": {
                                            "type": "string",
                                            "description": "Unique Id to track the
 status of the send reminder Call"
                                        },
                                        "sendReminderStatus": {
                                            "type": "string",
                                            "description": "Status of send reminder
 notifications"
                                        }
                                    }
                                }
                            }
                        },
                        "400": {
                            "description": "Bad request. One or more required fields are
 missing or invalid."
                        }
                    }
                }
            }
        }
    }
}
```

For more examples of OpenAPI schemas, see https://github.com/OAI/OpenAPI-Specification/tree/main/examples/v3.0 on the GitHub website.

Define Lambda functions for your agent's action groups in Amazon Bedrock

You must define a Lambda function to program the business logic for an action group. After a Amazon Bedrock agent determines the API operation that it needs to invoke in an action group, it sends information from the API schema alongside relevant metadata as an input event to the Lambda function. To write your function, you must understand the following components of the Lambda function:

- **Input event** – Contains relevant metadata and populated fields from the request body of the API operation that the agent determines must be called.
- **Response** – Contains relevant metadata and populated fields for the response body returned from the API operation.

You write your Lambda function to define how to handle an action group and to customize how you want the API response to be returned. You use the variables from the input event to define your functions and return a response to the agent.

> **Note**
>
> An action group can contain up to 5 API operations, but you can only write one Lambda function. Because the Lambda function can only receive an input event and return a response for one API operation at a time, you should write the function considering the different API operations that may be invoked.

For your agent to use a Lambda function, you must attach a resource-based policy to the function to provide permissions for the agent. For more information, follow the steps at Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function. For more information about resource-based policies in Lambda, see Using resource-based policies for Lambda in the AWS Lambda Developer Guide.

Topics

Lambda input event from Amazon Bedrock

When an action group using a Lambda function is invoked, Amazon Bedrock sends a Lambda input event of the following general format. You can define your Lambda function to use any of the input event fields to manipulate the business logic within the function to successfully perform the action. For more information about Lambda functions, see Event-driven invocation in the AWS Lambda Developer Guide.

```
{
    "messageVersion": "1.0",
    "agent": {
        "name": "string",
        "id": "string",
        "alias": "string",
        "version": "string"
    },
    "inputText": "string",
    "sessionId": "string",
    "actionGroup": "string",
    "apiPath": "string",
    "httpMethod": "string",
    "parameters": [
        {
            "name": "string",
            "type": "string",
            "value": "string"
        },
    ...
    ],
    "requestBody": {
        "content": {
            "<content_type>": {
                "properties": [
                   {
                       "name": "string",
                       "type": "string",
                       "value": "string"
                    },
                            ...
                ]
            }
        }
    },
```

```
    "sessionAttributes": {
        "string": "string",
    },
    "promptSessionAttributes": {
        "string": "string"
    }
}
```

The following list describes the input event fields;

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function. Amazon Bedrock only supports version 1.0.
- `agent` – Contains information about the name, ID, alias, and version of the agent that the action group belongs to.
- `inputText` – The user input for the conversation turn.
- `sessionId` – The unique identifier of the agent session.
- `actionGroup` – The name of the action group.
- `apiPath` – The path to the API operation, as defined in the OpenAPI schema.
- `httpMethod` – The method of the API operation, as defined in the OpenAPI schema.
- `parameters` – Contains a list of objects. Each object contains the name, type, and value of a parameter in the API operation, as defined in the OpenAPI schema.
- `requestBody` – Contains the request body and its properties, as defined in the OpenAPI schema.
- `sessionAttributes` – Contains session attributes and their values. These attributes are stored over a session and provide context for the agent.
- `promptSessionAttributes` – Contains prompt session attributes and their values. These attributes are stored over a turn and provide context for the agent.

Lambda response event to Amazon Bedrock

Amazon Bedrock expects a response from your Lambda function that matches the following format. The response consists of parameters returned from the API operation. The agent can use the response from the Lambda function for further orchestration or to help it return a response to the customer.

> **Note**
>
> See Agent quotas for the Lambda payload response quota.

```
{
    "messageVersion": "1.0",
    "response": {
        "actionGroup": "string",
        "apiPath": "string",
        "httpMethod": "string",
        "httpStatusCode": number,
        "responseBody": {
            "<contentType>": {
                "body": "JSON-formatted string"
            }
        }
    },
    "sessionAttributes": {
        "string": "string",
    },
    "promptSessionAttributes": {
        "string": "string"
    }
}
```

The following list describes the response fields:

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function. Amazon Bedrock only supports version 1.0.
- `response` – Contains the following information about the API response.
  - `actionGroup` – The name of the action group.
  - `apiPath` – The path to the API operation, as defined in the OpenAPI schema.
  - `httpMethod` – The method of the API operation, as defined in the OpenAPI schema.
  - `responseBody` – Contains the response body, as defined in the OpenAPI schema.
- (Optional) `sessionAttributes` – Contains session attributes and their values.
- (Optional) `promptSessionAttributes` – Contains prompt attributes and their values.

Action group Lambda function example

The following is an minimal example of how the Lambda function can be defined in Python.

```
def lambda_handler(event, context):

    response_body = {
        'application/json': {
            'body': "sample response"
        }
    }

    action_response = {
        'actionGroup': event['actionGroup'],
        'apiPath': event['apiPath'],
        'httpMethod': event['httpMethod'],
        'httpStatusCode': 200,
        'responseBody': response_body
    }

    session_attributes = event['sessionAttributes']
    prompt_session_attributes = event['promptSessionAttributes']

    api_response = {
        'messageVersion': '1.0',
        'response': action_response,
        'sessionAttributes': session_attributes,
        'promptSessionAttributes': prompt_session_attributes
    }

    return api_response
```

Add an action group to your agent in Amazon Bedrock

After setting up the OpenAPI schema and Lambda function for your action group, you can create the action group. Select the tab corresponding to your method of choice and follow the steps.

Console

When you create an agent, you can add action groups to the working draft.

After an agent is created, you can add action groups to it by doing the following steps:

To add an action group to an already created agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent from the **Agents** section and then choose the **Working draft** in the **Working Draft** section.
4. Select **Add** in the **Action groups** section.
5. Fill out the action group details
6. To define the schema for the action group with the in-line OpenAPI schema editor, carry out the following actions. For more information about API schemas for your action group, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock.
   a. Choose **Define with in-line OpenAPI schema editor** under **Select API schema**. A sample schema appears that you can edit.
   b. Select the format for the schema by using the dropdown menu next to **Format**.
   c. To import an existing schema from S3 to edit, select **Import schema**, provide the S3 URI, and select **Import**.
   d. To restore the schema to the original sample schema, select **Reset** and then confirm the message that appears by selecting **Reset** again.
7. Select **Add**. A green success banner appears if there are no issues. If there are issues validating the schema, a red banner appears. The following issues are identified by the validation process.
   - Scroll through the schema to see the lines where an error or warning about formatting exists. An X indicates a formatting error, while an exclamation mark indicates a warning about formatting.
   - Select **View details** in the red banner to see a list of errors about the content of the API schema.
8. Select **Prepare** to apply the changes that you have made to the agent before testing it.

API

To create an action group, send a CreateAgentActionGroup request with a Agents for Amazon Bedrock build-time endpoint.

See code examples

The following list describes the fields in the request:

- The following fields are required:

Field	Short description
agentId	The ID of the agent that the action group belongs to.
agentVersion	The version of the agent that the action group belongs to.
actionGroupName	The name of the action group.

- The following fields are optional:

Field	Short description
parentActionGroupSignature	Specify `AMAZON.UserInput` to allow the agent to reprompt the user for more information if it doesn't have enough information to complete another action group. You must leave the `description`, `apiSchema`, and `actionGroupExecutor` fields blank if you specify this field.
description	A description of the action group.
apiSchema	Specifies the OpenAPI schema for the action group or links to an S3 object containing it.
actionGroupExecutor	Specifies the Amazon Resource Name (ARN) of the Lambda function that performs the action group.

Field	Short description
actionGroupState	Whether to allow the agent to invoke the action group or not.
clientToken	An identifier to prevent requests from being duplicated.

Associate a knowledge base with an Amazon Bedrock agent

If you haven't yet created a knowledge base, see Knowledge bases for Amazon Bedrock to learn about knowledge bases and create one. You can associate a knowledge base during agent creation or after an agent has been created. To associate a knowledge base to an existing agent. select the tab corresponding to your method of choice and follow the steps..

Console

To add a knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. On the agent details page, choose the working draft from the **Working draft** section.
4. For the **Knowledge bases** section, choose **Add**.
5. Choose a knowledge base that you have created and provide instructions for how the agent should interact with it.
6. Choose **Add**. A success banner appears at the top.
7. To apply the changes that you made to the agent before testing it, choose **Prepare** before testing it.

API

To associate a knowledge base with an agent, send an AssociateAgentKnowledgeBase request with a Agents for Amazon Bedrock build-time endpoint.

The following list describes the fields in the request:

- The following fields are required:

Field	Short description
agentId	ID of the agent
agentVersion	Version of the agent
knowledgeBaseId	ID of the knowledge base

- The following fields are optional:

Field	Short description
description	Description of how the agent can use the knowledge base
knowledgeBaseState	To prevent the agent from querying the knowledge base, specify `DISABLED`

Test an Amazon Bedrock agent

After you create an agent, you will have a *working draft*. The working draft is a version of the agent that you can use to iteratively build the agent. Each time you make changes to your agent, the working draft is updated. When you're satisfied with your agent's configurations, you can create a *version*, which is a snapshot of your agent, and an *alias*, which points to the version. You can then deploy your agent to your applications by calling the alias. For more information, see Deploy an Amazon Bedrock agent.

The following list describes how you test your agent:

- In the Amazon Bedrock console, you open up the test window on the side and send input for your agent to respond to. You can select the working draft or a version that you've created.
- In the API, the working draft is the `DRAFT` version. You send input to your agent by using InvokeAgent with the test alias, `TSTALIASID`, or a different alias pointing to a static version.

To help troubleshoot your agent's behavior, Agents for Amazon Bedrock provides the ability to view the *trace* during a session with your agent. The trace shows the agent's step-by-step reasoning process. For more information about the trace, see Trace events in Amazon Bedrock.

Following are steps for testing your agent. Select the tab corresponding to your method of choice and follow the steps.

Console

To test an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agents** section, select the link for the agent that you want to test from the list of agents.
4. The **Test** window appears in a pane on the right.

   > **Note**
   >
   > If the **Test window** is closed, you can reopen it by selecting **Test** at the top of the agent details page or any page within it.

5. After you create an agent, you must package it with the working draft changes by preparing it in one of the following ways:
   - In the **Test** window, select **Prepare**.
   - In the **Working draft** page, select **Prepare** at the top of the page.

   > **Note**
   >
   > Every time you update the working draft, you must prepare the agent to package the agent with your latest changes. As a best practice, we recommend that you always check your agent's **Last prepared** time in the **Agent overview** section of the **Working draft** page to verify that you're testing your agent with the latest configurations.

6. To choose an alias and associated version to test, use the dropdown menu at the top of the **Test window**. By default, the **TestAlias: Working draft** combination is selected.

7. To test the agent, enter a message and choose **Run**. While you wait for the response to generate or after it is generated, you have the following options:

   - To view details for each step of the agent's orchestration process, including the prompt, inference configurations, and agent's reasoning process for each step and usage of its action groups and knowledge bases, select **Show trace**. The trace is updated in real-time so you can view it before the response is returned. To expand or collapse the trace for a step, select an arrow next to a step. For more information about the **Trace** window and details that appear, see Trace events in Amazon Bedrock.
   - If the agent invokes a knowledge base, the response contains footnotes. To view the link to the S3 object containing the cited information for a specific part of the response, select the relevant footnote.

   You can perform the following actions in the **Test** window:

   - To start a new conversation with the agent, select the refresh icon.
   - To view the **Trace** window, select the expand icon. To close the **Trace** window, select the shrink icon.
   - To close the **Test** window, select the right arrow icon.

You can enable or disable action groups and knowledge bases. Use this feature to troubleshoot your agent by isolating which action groups or knowledge bases need to be updated by assessing its behavior with different settings.

To enable an action group or knowledge base

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agents** section. select the link for the agent that you want to test from the list of agents.
4. On the agent's details page, in the **Working draft** section, select the link for the **Working draft**.
5. In the **Action groups** or **Knowledge bases** section, hover over the **State** of the action group or knowledge base whose state you want to change.

6. An edit button appears. Select the edit icon and then choose from the dropdown menu whether the action group or knowledge base is **Enabled** or **Disabled**.
7. If an action group is **Disabled**, the agent doesn't use the action group. If a knowledge base is **Disabled**, the agent doesn't use the knowledge base. Enable or disable action groups or knowledge bases and then use the **Test** window to troubleshoot your agent.
8. Choose **Prepare** to apply the changes that you have made to the agent before testing it.

API

Before you test your agent for the first time, you must package it with the working draft changes by sending a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

See code examples

> **Note**
>
> Every time you update the working draft, you must prepare the agent to package the agent with your latest changes. As a best practice, we recommend that you send a GetAgent request (see link for request and response formats and field details) with a Agents for Amazon Bedrock build-time endpoint and check the `preparedAt` time for your agent to verify that you're testing your agent with the latest configurations.

To test your agent, send an InvokeAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint.

See code examples

> **Note**
>
> The AWS CLI doesn't support InvokeAgent.

See code examples

The following fields exist in the request:

- Minimally, provide the following required fields:

Field	Short description
agentId	ID of the agent
agentAliasId	ID of the alias. Use `TSTALIASID` to invoke the `DRAFT` version
sessionId	Alphanumeric ID for the session (2–100 characters)
inputText	The user prompt to send to the agent

- The following fields are optional:

Field	Short description
enableTrace	Specify `TRUE` to view the trace.
endSession	Specify `TRUE` to end the session with the agent after this request.
sessionState	Includes `sessionAttributes` that remain throughout the session and/or `promptSessionAttributes` that remain throughout the InvokeAgent turn. These attributes give the agent context.

The response is returned in bytes in the `chunk` object. If the agent queried a knowledge base, the `chunk` includes `citations`. If you enabled a trace, a `trace` object is also returned. If an error occurs, a field is returned with the error message. For more information about how to read the trace, see Trace events in Amazon Bedrock.

Trace events in Amazon Bedrock

Each response from an Amazon Bedrock agent is accompanied by a *trace* that details the steps being orchestrated by the agent. The trace helps you follow the agent's reasoning process that leads it to the response it gives at that point in the conversation.

Use the trace to track the agent's path from the user input to the response it returns. The trace provides information about the inputs to the action groups that the agent invokes and the knowledge bases that it queries to respond to the user. In addition, the trace provides information about the outputs that the action groups and knowledge bases return. You can view the reasoning that the agent uses to determine the action that it takes or the query that it makes to a knowledge base. If a step in the trace fails, the trace returns a reason for the failure. Use the detailed information in the trace to troubleshoot your agent. You can identify steps at which the agent has trouble or at which it yields unexpected behavior. Then, you can use this information to consider ways in which you can improve the agent's behavior.

View the trace

The following describes how to view the trace. Select the tab corresponding to your method of choice and follow the steps.

Console

To view the trace during a conversation with an agent

Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

1. In the **Agents** section, select the link for the agent that you want to test from the list of agents.
2. The **Test** window appears in a pane on the right.
3. Enter a message and choose **Run**. While the response is generating or after it finishes generating, select **Show trace**.
4. You can view the trace for each **Step** in real-time as your agent performs orchestration.

API

To view the trace, send an InvokeAgent request with a Agents for Amazon Bedrock runtime endpoint and set the `enableTrace` field to `TRUE`. By default, the trace is disabled.

If you enable the trace, in the InvokeAgent response, each `chunk` in the stream is accompanied by a `trace` field that maps to a TracePart object. Within the TracePart is a `trace` field that maps to a Trace object.

Structure of the trace

The trace is shown as a JSON object in both the console and the API. Each **Step** in the console or Trace in the API can be one of the following traces:

- PreProcessingTrace – Traces the input and output of the pre-processing step, in which the agent contextualizes and categorizes user input and determines if it is valid.
- Orchestration – Traces the input and output of the orchestration step, in which the agent interprets the input and invokes API operations and queries knowledge bases. Then the agent returns output to either continue orchestration or to respond to the user.
- PostProcessingTrace – Traces the input and output of the post-processing step, in which the agent handles the final output of the orchestration and determines how to return the response to the user.
- FailureTrace – Traces the reason that a step failed.

Each of the traces (except `FailureTrace`) contains a ModelInvocationInput object. The ModelInvocationInput object contains configurations set in the prompt template for the step, alongside the prompt provided to the agent at this step. For more information about how to modify prompt templates, see Advanced prompts in Amazon Bedrock. The structure of the `ModelInvocationInput` object is as follows:

```
{
    "traceId": "string",
    "text": "string",
    "type": "PRE_PROCESSING | ORCHESTRATION | KNOWLEDGE_BASE_RESPONSE_GENERATION |
 POST_PROCESSING",
    "inferenceConfiguration": {
        "maximumLength": number,
        "stopSequences": ["string"],
        "temperature": float,
        "topK": float,
        "topP": float
    },
    "promptCreationMode": "DEFAULT | OVERRIDDEN",
```

```
    "parserMode": "DEFAULT | OVERRIDDEN",
    "overrideLambda": "string"
}
```

The following list describes the fields of the ModelInvocationInput object:

- `traceId` – The unique identifier of the trace.
- `text` – The text from the prompt provided to the agent at this step.
- `type` – The current step in the agent's process.
- `inferenceConfiguration` – Inference parameters that influence response generation. For more information, see Inference parameters.
- `promptCreationMode` – Whether the agent's default base prompt template was overridden for this step. For more information, see Advanced prompts in Amazon Bedrock.
- `parserMode` – Whether the agent's default response parser was overridden for this step. For more information, see Advanced prompts in Amazon Bedrock.
- `overrideLambda` – The Amazon Resource Name (ARN) of the parser Lambda function used to parse the response, if the default parser was overridden. For more information, see Advanced prompts in Amazon Bedrock.

For more information about each trace type, see the following sections:

PreProcessingTrace

```
{
    "modelInvocationInput": { // see above for details }
    "modelInvocationOutput": {
        "parsedResponse": {
            "isValid": boolean,
            "rationale": "string"
        },
        "traceId": "string"
    }
}
```

The PreProcessingTrace consists of a ModelInvocationInput object and a PreProcessingModelInvocationOutput object. The PreProcessingModelInvocationOutput contains the following fields.

- `parsedResponse` – Contains the following details about the parsed user prompt.
  - `isValid` – Specifies whether the user prompt is valid.
  - `rationale` – Specifies the agent's reasoning for the next steps to take.
- `traceId` – The unique identifier of the trace.

OrchestrationTrace

The Orchestration consists of the ModelInvocationInput object and any combination of the Rationale, InvocationInput, and Observation objects. For more information about each object, select from the following tabs:

```
{
    "modelInvocationInput": { // see above for details },
    "rationale": { ... },
    "invocationInput": { ... },
    "observation": { ... }
}
```

Rationale

The Rationale object contains the reasoning of the agent given the user input. Following is the structure:

```
{
       "traceId": "string",
       "text": "string"
    }
```

The following list describes the fields of the Rationale object:

- `traceId` – The unique identifier of the trace step.
- `text` – The reasoning process of the agent, based on the input prompt.

InvocationInput

The InvocationInput object contains information that will be input to the action group or knowledge base that is to be invoked or queried. Following is the structure:

```
{
```

```
    "traceId": "string",
    "invocationType": "ACTION_GROUP | KNOWLEDGE_BASE | FINISH",
    "actionGroupInvocationInput": {
        "actionGroupName": "string",
        "apiPath": "string",
        "verb": "string",
        "parameters": [
            {
                "name": "string",
                "type": "string"
                "value": "string"
            },
            ...
        ],
        "request": {
            "content": {
                "content-type": [
                    {
                        "name": "string",
                        "type": "string",
                        "value": "string"
                    }
                ]
            }
        }
    },
    "knowledgeBaseLookupInput": {
        "knowledgeBaseId": "string",
        "text": "string"
    }
}
```

The following list describes the fields of the InvocationInput object:

- `traceId` – The unique identifier of the trace.
- `invocationType` – Specifies whether the agent is invoking an action group or a knowledge base, or ending the session.
- `actionGroupInvocationInput` – Appears if the `type` is `ACTION_GROUP`. For more information, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock. Contains the following input for the action group that is invoked:
  - `actionGroupName` – The name of the action group that the agent will invoke.

- `apiPath` – The path to the API operation to call, according to the API schema.
- `verb` – The API method being used, according to the API schema.
- `parameters` – Contains a list of objects. Each object contains the name, type, and value of a parameter in the API operation, as defined in the API schema.
- `requestBody` – Contains the request body and its properties, as defined in the API schema.

- `knowledgeBaseLookupInput` – Appears if the `type` is `KNOWLEDGE_BASE`. For more information, see Knowledge bases for Amazon Bedrock. Contains the following information about the knowledge base and the search query for the knowledge base:
  - `knowledgeBaseId` – The unique identifier of the knowledge base that the agent will look up.
  - `text` – The query to be made to the knowledge base.

Observation

The Observation object contains the result or output of an action group or knowledge base, or the response to the user. Following is the structure:

```
{
    "traceId": "string",
    "type": "ACTION_GROUP | KNOWLEDGE_BASE | REPROMPT | ASK_USER | FINISH"
    "actionGroupInvocation": {
        "text": "JSON-formatted string"
    },
    "knowledgeBaseLookupOutput": {
        "retrievedReferences": [
            {
                "content": {
                    "text": "string"
                },
                "location": {
                    "type": "S3",
                    "s3Location": {
                        "uri": "string"
                    }
                }
            },
            ...
        ]
```

```
    },
    "repromptResponse": {
        "source": "ACTION_GROUP | KNOWLEDGE_BASE | PARSER",
        "text": "string"
    },
    "finalResponse": {
        "text"
    }
}
```

The following list describes the fields of the Observation object:

- `traceId` – The unique identifier of the trace.
- `type` – Specifies whether the agent's observation is returned from the result of an action group or a knowledge base, if the agent is reprompting the user, requesting more information, or ending the conversation.
- `actionGroupInvocationOutput` – Contains the JSON-formatted string returned by the API operation that was invoked by the action group. Appears if the `type` is `ACTION_GROUP`. For more information, see Define OpenAPI schemas for your agent's action groups in Amazon Bedrock.
- `knowledgeBaseLookupOutput` – Contains text retrieved from the knowledge base that is relevant to responding to the prompt, alongside the Amazon S3 location of the data source. Appears if the `type` is `KNOWLEDGE_BASE`. For more information, see Knowledge bases for Amazon Bedrock. Each object in the list of `retrievedReferences` contains the following fields:
  - `content` – Contains `text` from the knowledge base that is returned from the knowledge base query.
  - `location` – Contains the Amazon S3 URI of the data source from which the returned text was found.
- `repromptResponse` – Appears if the `type` is `REPROMPT`. Contains the `text` that asks for a prompt again alongside the `source` of why the agent needs to reprompt.
- `finalResponse` – Appears if the `type` is `ASK_USER` or `FINISH`. Contains the `text` that asks the user for more information or is a response to the user.

PostProcessingTrace

```
{
```

```
    "modelInvocationInput": { // see above for details }
    "modelInvocationOutput": {
        "parsedResponse": {
            "text": "string"
        },
        "traceId": "string"
    }
}
```

The PostProcessingTrace consists of a ModelInvocationInput object and a PostProcessingModelInvocationOutput object. The PostProcessingModelInvocationOutput contains the following fields:

- `parsedResponse` – Contains the `text` to return to the user after the text is processed by the parser function.
- `traceId` – The unique identifier of the trace.

FailureTrace

```
{
    "failureReason": "string",
    "traceId": "string"
}
```

The following list describes the fields of the FailureTrace object:

- `failureReason` – The reason that the step failed.
- `traceId` – The unique identifier of the trace.

Manage an Amazon Bedrock agent

After you create an agent, you can view or update its configuration as required. The configuration applies to the working draft. If you no longer need an agent, you can delete it.

Topics

- View information about an agent
- Edit an agent
- Delete an agent

- Manage the action groups of an agent
- Manage agent-knowledge bases associations

View information about an agent

To learn how to view information about an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. On the agent details, you can view the following information:
   - The **Agent overview** section contains the agent configuration.
   - The **Tags** section contains tags that are associated with the agent. For more information, see Tag resources.
   - The **Working draft** section contains the working draft. If you select the working draft, you can view the following information:
     - The **Model details** section contains model and instructions used by the agent's working draft.
     - The **Action groups** section contains the action groups that the agent uses. For more information, see Create an action group for an Amazon Bedrock agent and Manage the action groups of an agent.
     - The **Knowledge bases** section contains the knowledge bases associated with the agent. For more information, see Associate a knowledge base with an Amazon Bedrock agent and Manage agent-knowledge bases associations.
     - The **Advanced prompts** section contains the prompt templates for each step of the agent's orchestration. For more information, see Advanced prompts in Amazon Bedrock.
   - The **Versions** and **Aliases** sections contain versions and aliases of the agent that you can use for deployment to your applications. For more information, see Deploy an Amazon Bedrock agent.

API

To get information about an agent, send a GetAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`. See code examples.

To list information about your agents, send a ListAgents request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. See code examples. You can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

To list all the tags for an agent, send a ListTagsForResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the agent.

Edit an agent

To learn how to edit an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To edit the agent configuration

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agent overview** section, choose **Edit**.

4. Edit the existing information in the fields as necessary.
5. When you're done editing the information, choose **Save** to remain in the same window or **Save and exit** to return to the agent details page. A success banner appears at the top. To apply the new configurations to your agent, select **Prepare** in the banner.

You might want to try different foundation models for your agent or change the instructions for the agent. These changes apply only to the working draft.

To change the foundation model that your agent uses or the instructions to the agent.

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Model details** section, choose **Edit**
6. Select a different model or edit the instructions to the agent as necessary.

> **Note**
>
> If you change the foundation model, any prompt templates that you modified will be set to default for that model.

7. When you're done editing the information, choose **Save** to remain in the same window or **Save and exit** to return to the agent details page. A success banner appears at the top.
8. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

To edit the tags associated with an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. In the **Tags** section, choose **Manage tags**.

5. To add a tag, choose **Add new tag**. Then enter a **Key** and optionally enter a **Value**. To remove a tag, choose **Remove**. For more information, see Tag resources.
6. When you're done editing tags, choose **Submit**.

API

To edit an agent, send an UpdateAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same. For more information about required and optional fields, see Create an agent in Amazon Bedrock.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

To add tags to an agent, send a TagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the agent. The request body contains a `tags` field, which is an object containing a key-value pair that you specify for each tag.

To remove tags from an agent, send an UntagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the agent. The `tagKeys` request parameter is a list containing the keys for the tags that you want to remove.

Delete an agent

To learn how to delete an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To delete an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

2. Select **Agents** from the left navigation pane.
3. To delete an agent, choose the option button that's next to the agent you want to delete.
4. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the agent, enter **`delete`** in the input field and then select **Delete**.
5. When deletion is complete, a success banner appears.

API

To delete an agent, send a DeleteAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`.

By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

See code examples

Select a topic to learn about how to manage the action groups or knowledge bases for an agent.

Topics

- Manage the action groups of an agent
- Manage agent-knowledge bases associations

Manage the action groups of an agent

After creating an action group, you can view, edit, or delete it. The changes apply to the working draft version of the agent.

Topics

- View information about an action group
- Edit an action group
- Delete an action group

View information about an action group

To learn how to view information about an action group, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about an action group

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, choose an action group for which to view information.

API

To get information about an action group, send a GetAgentActionGroup request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `actionGroupId`, `agentId`, and `agentVersion`.

To list information about an agent's action groups, send a ListAgentActionGroups request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentVersion` for which you want to see action groups. You can include the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

See code examples

Edit an action group

To learn how to edit an action group, select the tab corresponding to your method of choice and follow the steps.

Console

To edit an action group

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, select an action group to edit. Then choose **Edit**.
6. Edit the existing fields as necessary. For more information, see Create an action group for an Amazon Bedrock agent.
7. To define the schema for the action group with the in-line OpenAPI schema editor, for **Select API schema**, choose **Define with in-line OpenAPI schema editor**. A sample schema appears that you can edit. You can configure the following options:
   - To import an existing schema from Amazon S3 to edit, choose **Import schema**, provide the Amazon S3 URI, and select **Import**.
   - To restore the schema to the original sample schema, choose **Reset** and then confirm the message that appears by choosing **Confirm**.
   - To select a different format for the schema, use the dropdown menu labeled **JSON**.
   - To change the visual appearance of the schema, choose the gear icon below the schema.
8. To control whether the agent can use the action group, select **Enable** or **Disable**. Use this function to help troubleshoot your agent's behavior.
9. To remain in the same window so that you can test your change, choose **Save**. To return to the action group details page, choose **Save and exit**.
10. A success banner appears if there are no issues. If there are issues validating the schema, an error banner appears. To see a list of errors, choose **Show details** in the banner.

11. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To edit an action group, send an UpdateAgentActionGroup request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same. You must specify the `agentVersion` as `DRAFT`. For more information about required and optional fields, see Create an action group for an Amazon Bedrock agent.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Delete an action group

To learn how to delete an action group, select the tab corresponding to your method of choice and follow the steps.

Console

To delete an action group

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, choose the option button that's next to the action group you want to delete.
6. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the action group, enter **`delete`** in the input field and then select **Delete**.
7. When deletion is complete, a success banner appears.

8. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To delete an action group, send a DeleteAgentActionGroup request. Specify the `actionGroupId` and the `agentId` and `agentVersion` from which to delete it. By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Manage agent-knowledge bases associations

After creating an agent, you can add more knowledge bases or edit them. Adding and editing take place within the working draft. To carry out these operations, choose an agent from the **Agents** section and then choose the **Working draft** in the **Working Draft** section.

Topics

- View information about an agent-knowledge base association
- Edit an agent-knowledge base association
- Disassociate a knowledge base from an agent

View information about an agent-knowledge base association

To learn how to view information about a knowledge base, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a knowledge base that's associated with an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Knowledge bases** section, select the knowledge base for which you want to view information.

API

To get information about a knowledge base associated with an agent, send a GetAgentKnowledgeBase request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the following fields:

To list information about the knowledge bases associated with an agent, send a ListAgentKnowledgeBases request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentVersion` for which you want to see associated knowledge bases.

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

See code examples

Edit an agent-knowledge base association

To learn how to edit an agent-knowledge base association, select the tab corresponding to your method of choice and follow the steps.

Console

To edit an agent-knowledge base association

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Action groups** section, select an action group to edit. Then choose **Edit**.
6. Edit the existing fields as necessary. For more information, see Associate a knowledge base with an Amazon Bedrock agent.
7. To control whether the agent can use the knowledge base, select **Enabled** or **Disabled**. Use this function to help troubleshoot your agent's behavior.
8. To remain in the same window so that you can test your change, choose **Save**. To return to the **Working draft** page, choose **Save and exit**.
9. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To edit the configuration of a knowledge base associated with an agent, send an UpdateAgentKnowledgeBase request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same. You must specify the `agentVersion` as `DRAFT`. For more information about required and optional fields, see Associate a knowledge base with an Amazon Bedrock agent.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Disassociate a knowledge base from an agent

To learn how to disassociate a knowledge base from an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To dissociate a knowledge base from an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose an agent in the **Agents** section.
4. On the agent details page, for the **Working draft** section, choose the working draft.
5. In the **Knowledge bases** section, choose the option button that's next to the knowledge base that you want to delete. Then choose **Delete**.
6. Confirm the message that appears and then choose **Delete**.
7. To apply the changes that you made to the agent before testing it, choose **Prepare** in the **Test** window or at the top of the **Working draft** page.

API

To disassociate a knowledge base from an agent, send a DisassociateAgentKnowledgeBase request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `knowledgeBaseId` and the `agentId` and `agentVersion` of the agent from which to disassociate it.

To apply the changes to the working draft, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Include the `agentId` in the request. The changes apply to the `DRAFT` version, which the `TSTALIASID` alias points to.

Customize an Amazon Bedrock agent

After you have set up your agent, you can further customize its behavior with the following features:

- **Advanced prompts** let you modify prompt templates to determine the prompt that is sent to the agent at each step of runtime.
- **Session state** is a field that contains attributes that you can define during build-time when sending a CreateAgent request or that you can send at runtime with an InvokeAgent request. You can use these attributes to provide and manage context in a conversation between users and the agent.
- Agents for Amazon Bedrock offers options to choose different flows that can optimize on latency for simpler use cases in which agents have a single knowledge base. To learn more, refer to the performance optimization topic.

Select a topic to learn more about that feature.

Topics

Advanced prompts in Amazon Bedrock

After creation, an agent is configured with the following four default **base prompt templates**, which outline how the agent constructs prompts to send to the foundation model at each step of the agent sequence. For details about what each step encompasses, see Runtime process.

- Pre-processing
- Orchestration
- Knowledge base response generation
- Post-processing (disabled by default)

Prompt templates define how the agent does the following:

- Processes user input text and output prompts from foundation models (FMs)
- Orchestrates between the FM, action groups, and knowledge bases
- Formats and returns responses to the user

By using advanced prompts, you can enhance your agent's accuracy through modifying these prompt templates to provide detailed configurations. You can also provide hand-curated examples for *few-shot prompting*, in which you improve model performance by providing labeled examples for a specific task.

Select a topic to learn more about advanced prompts.

Topics

Advanced prompts terminology

The following terminology is helpful in understanding how advanced prompts work.

- **Session** – A group of InvokeAgent requests made to the same agent with the same session ID. When you make an `InvokeAgent` request, you can reuse a `sessionId` that was returned from the response of a previous call in order to continue the same session with an agent. As long as the `idleSessionTTLInSeconds` time in the Agent configuration hasn't expired, you maintain the same session with the agent.
- **Turn** – A single `InvokeAgent` call. A session consists of one or more turns.
- **Iteration** – A sequence of the following actions:

  1. (Required) A call to the foundation model
  2. (Optional) An action group invocation
  3. (Optional) A knowledge base invocation
  4. (Optional) A response to the user asking for more information

  An action might be skipped, depending on the configuration of the agent or the agent's requirement at that moment. A turn consists of one or more iterations.
- **Prompt** – A prompt consists of the instructions to the agent, context, and text input. The text input can come from a user or from the output of another step in the agent sequence. The prompt is provided to the foundation model to determine the next step that the agent takes in responding to user input

- **Base prompt template** – The structural elements that make up a prompt. The template consists of placeholders that are filled in with user input, the agent configuration, and context at runtime to create a prompt for the foundation model to process when the agent reaches that step. For more information about these placeholders, see Placeholder variables in Amazon Bedrock agent prompt templates). With advanced prompts, you can edit these templates.

Configure the prompt templates

With advanced prompts, you can do the following:

- Turn on or turn off invocation for different steps in the agent sequence.
- Configure their inference parameters.
- Edit the default base prompt templates that the agent uses. By overriding the logic with your own configurations, you can customize your agent's behavior.

For each step of the agent sequence, you can edit the following parts:

- **Prompt template** – Describes how the agent should evaluate and use the prompt that it receives at the step for which you're editing the template. When editing a template, you can engineer the prompt with the following tools:
  - **Prompt template placeholders** – Pre-defined variables in Agents for Amazon Bedrock that are dynamically filled in at runtime during agent invocation. In the prompt templates, you'll see these placeholders surrounded by $ (for example, `$instructions$`). For information about the placeholder variables that you can use in a template, see Placeholder variables in Amazon Bedrock agent prompt templates.
  - **XML tags** – Anthropic models support the use of XML tags to structure and delineate your prompts. Use descriptive tag names for optimal results. For example, in the default orchestration prompt template, you'll see the `<examples>` tag used to delineate few-shot examples). For more information, see Use XML tags in the Anthropic user guide.

  You can enable or disable any step in the agent sequence. The following table shows the default states for each step.

Prompt template	Default setting
Pre-processing	Enabled

Prompt template	Default setting
Orchestration	Enabled
Knowledge base response generation	Enabled
Post-processing	Disabled

> **Note**
>
> If you disable the orchestration step, the agent sends the raw user input to the foundation model and doesn't use the base prompt template for orchestration. If you disable any of the other steps, the agent skips that step entirely.

- **Inference configurations** – Influences the response generated by the model that you use. For definitions of the inference parameters and more details about the parameters that different models support, see Inference parameters for foundation models.
- **(Optional) Parser Lambda function** – Defines how to parse the raw foundation model output and how to use it in the runtime flow. This function acts on the output from the steps in which you enable it and returns the parsed response as you define it in the function.

  Depending on how you customized the base prompt template, the raw foundation model output might be specific to the template. As a result, the agent's default parser might have difficulty parsing the output correctly. By witing a custom parser Lambda function, you can help the agent parse the raw foundation model output based on your use-case. For more information about the parser Lambda function and how to write it, see Parser Lambda function in Agents for Amazon Bedrock.

  > **Note**
  >
  > You can define one parser Lambda function for all of the base templates, but you can configure whether to invoke the function in each step. Be sure to configure a resource-based policy for your Lambda function so that your agent can invoke it. For more information, see Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function.

After you edit the prompt templates, you can test your agent. To analyze the step-by-step process of the agent and determine if it is working as you intend, turn on the trace and examine it. For more information, see Trace events in Amazon Bedrock.

You can configure advanced prompts in either the AWS Management Console or through the API.

Console

In the console, you can configure advanced prompts after you have created the agent. You configure them while editing the agent.

To view or edit advanced prompts for your agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. In the left navigation pane, choose **Agents**. Then choose an agent in the **Agents** section.
3. On the agent details page, in the **Working draft** section, select **Working draft**.
4. On the **Working draft** page, in the **Advanced prompts** section, choose **Edit**.
5. On the **Edit advanced prompts** page, choose the tab corresponding to the step of the agent sequence that you want to edit.
6. To enable editing of the template, turn on **Override template defaults**. In the **Override template defaults** dialog box, choose **Confirm**.

   > **Warning**
   >
   > If you turn off **Override template defaults** or change the model, the default Amazon Bedrock template is used and your template will be immediately deleted. To confirm, enter **`confirm`** in the text box to confirm the message that appears.

7. To allow the agent to use the template when generating responses, turn on **Activate template**. If this configuration is turned off, the agent doesn't use the template.
8. To modify the example prompt template, use the **Prompt template editor**.
9. In **Configurations**, you can modify inference parameters for the prompt. For definitions of parameters and more information about parameters for different models, see Inference parameters for foundation models.
10. (Optional) To use a Lambda function that you have defined to parse the raw foundation model output, perform the following actions:

> **Note**
>
> One Lambda function is used for all the prompt templates.

a. In the **Configurations** section, select **Use Lambda function for parsing**. If you clear this setting, your agent will use the default parser for the prompt.

b. For the **Parser Lambda function**, select a Lambda function from the dropdown menu.

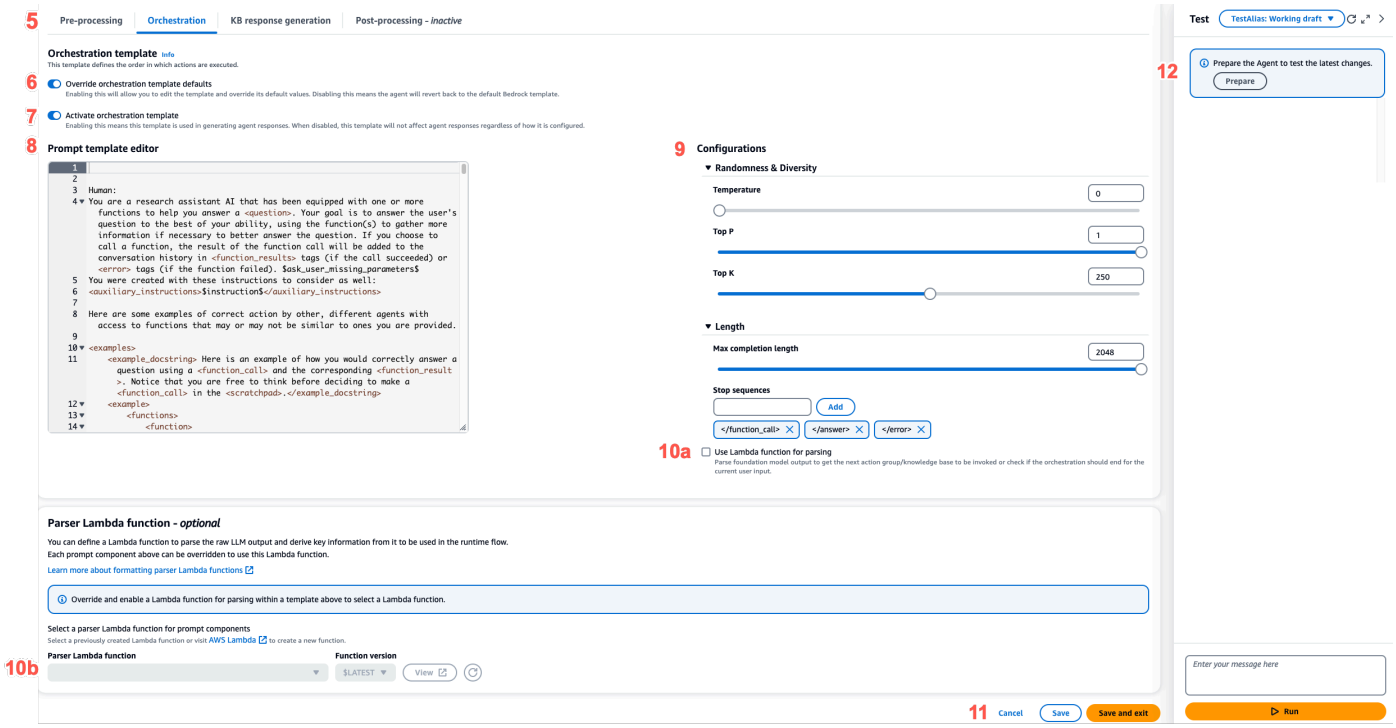
> **Note**
>
> You must attach permissions for your agent so that it can access the Lambda function. For more information, see Resource-based policy to allow Amazon Bedrock to invoke an action group Lambda function.

11. To save your settings, choose one of the following options:

    a. To remain in the same window so that you can dynamically update the prompt settings while testing your updated agent, choose **Save**.

    b. To save your settings and return to the **Working draft** page, choose **Save and exit**.

12. To test the updated settings, choose **Prepare** in the **Test** window.



API

To configure advanced prompts by using the API operations, you send an [UpdateAgent](#) call and modify the following `promptOverrideConfiguration` object.

```
"promptOverrideConfiguration": {
    "overrideLambda": "string",
    "promptConfigurations": [
        {
            "basePromptTemplate": "string",
            "inferenceConfiguration": {
                "maximumLength": int,
                "stopSequences": [ "string" ],
                "temperature": float,
                "topK": float,
                "topP": float
            },
            "parserMode": "DEFAULT | OVERRIDDEN",
            "promptCreationMode": "DEFAULT | OVERRIDDEN",
            "promptState": "ENABLED | DISABLED",
            "promptType": "PRE_PROCESSING | ORCHESTRATION |
 KNOWLEDGE_BASE_RESPONSE_GENERATION | POST_PROCESSING"
        }
    ]
```

```
}
```

1. In the `promptConfigurations` list, include a `promptConfiguration` object for each prompt template that you want to edit.
2. Specify the prompt to modify in the `promptType` field.
3. Modify the prompt template through the following steps:
   a. Specify the `basePromptTemplate` fields with your prompt template.
   b. Include inference parameters in the `inferenceConfiguration` objects. For more information about inference configurations, see Inference parameters for foundation models.
4. To enable the prompt template, set the `promptCreationMode` to `OVERRIDDEN`.
5. To allow or prevent the agent from performing the step in the `promptType` field, modify the `promptState` value. This setting can be useful for troubleshooting the agent's behavior.
   - If you set `promptState` to `DISABLED` for the `PRE_PROCESSING`, `KNOWLEDGE_BASE_RESPONSE_GENERATION`, or `POST_PROCESSING` steps, the agent skips that step.
   - If you set `promptState` to `DISABLED` for the `ORCHESTRATION` step, the agent sends only the user input to the foundation model in orchestration. In addition, the agent returns the response as is without orchestrating calls between API operations and knowledge bases.
   - By default, the `POST_PROCESSING` step is `DISABLED`. By default, the `PRE_PROCESSING`, `ORCHESTRATION`, and `KNOWLEDGE_BASE_RESPONSE_GENERATION` steps are `ENABLED`.
6. To use a Lambda function that you have defined to parse the raw foundation model output, perform the following steps:
   a. For each prompt template that you want to enable the Lambda function for, set `parserMode` to `OVERRIDDEN`.
   b. Specify the Amazon Resource Name (ARN) of the Lambda function in the `overrideLambda` field in the `promptOverrideConfiguration` object.

Placeholder variables in Amazon Bedrock agent prompt templates

You can use placeholder variables in agent prompt templates. The variables will be populated by pre-existing configurations when the prompt template is called. Select a tab to see variables that you can use for each prompt template.

Pre-processing

Variable	Models supported	Replaced by
$functions$	Anthropic Claude Instant, Claude v2.0	Action group API operations and knowledge bases configured for the agent.
$tools$	Anthropic Claude v2.1	
$conversation_history$	All	Conversation history for the current session
$question$	All	User input for the current `InvokeAgent` call in the session.

Orchestration

Variable	Models supported	Replaced by
$functions$	Anthropic Claude Instant, Claude v2.0	Action group API operations and knowledge bases configured for the agent.
$tools$	Anthropic Claude v2.1	
$agent_scratchpad$	All	Designates an area for the model to write down its thoughts and actions it has taken. Replaced by predictions and output of the previous iterations in the current turn. Provides the

Variable	Models supported	Replaced by
		model with context of what has been achieved for the given user input and what the next step should be.
$any_function_name$	Anthropic Claude Instant, Claude v2.0	A randomly chosen API name from the API names that exist in the agent's action groups.
$conversation_history$	All	Conversation history for the current session
$instruction$	All	Model instructions configure d for the agent.
$prompt_session_attributes$	All	Session attributes preserved across a prompt
$question$	All	User input for the current `InvokeAgent` call in the session.

You can use the following placeholder variables if you allow the agent to ask the user for more information by doing one of the following actions:

- In the console, set in the **User input** in the agent details.
- Set the `parentActionGroupSignature` to `AMAZON.UserInput` with a CreateAgentActionGroup or UpdateAgentActionGroup request.

Variable	Models supported	Replaced by
$ask_user_missing_parameters$	Anthropic Claude Instant, Claude v2.0	Instructions for the model to ask the user to provide required missing information

Variable	Models supported	Replaced by
$ask_user_missing_information$	Anthropic Claude v2.1	
$ask_user_confirm_parameters$	All	Instructions for the model to ask the user to confirm parameters that the agent hasn't yet received or is unsure of.
$ask_user_function$	All	A function to ask the user a question.
$ask_user_function_format$	All	The format of the function to ask the user a question.
$ask_user_input_examples$	All	Few-shot examples to inform the model how to predict when it should ask the user a question.

Knowledge base response generation

Variable	Model	Replaced by
$query$	All	The query generated by the orchestration prompt model response when it predicts the next step to be knowledge base querying.
$search_results$	All	The retrieved results for the user query

Post-processing

Variable	Model	Replaced by
$latest_response$	All	The last orchestration prompt model response
$question$	All	User input for the current `InvokeAgent` call in the session.
$responses$	All	The action group and knowledge base outputs from the current turn.

Parser Lambda function in Agents for Amazon Bedrock

Each prompt template includes a parser Lambda function that you can modify. To write a custom parser Lambda function, you must understand the input event that your agent sends and the response that the agent expects as the output from the Lambda function. You write a handler function to manipulate variables from the input event and to return the response. For more information about how AWS Lambda works, see Event-driven invocation in the AWS Lambda Developer Guide.

Topics

- Parser Lambda input event
- Parser Lambda response
- Parser Lambda examples

Parser Lambda input event

The following is the general structure of the input event from the agent. Use the fields to write your Lambda handler function.

```
{
    "messageVersion": "1.0",
    "agent": {
```

```
        "name": "string",
        "id": "string",
        "alias": "string",
        "version": "string"
    },
    "invokeModelRawResponse": "string",
    "promptType": "ORCHESTRATION | POST_PROCESSING | PRE_PROCESSING |
 KNOWLEDGE_BASE_RESPONSE_GENERATION ",
    "overrideType": "OUTPUT_PARSER"
}
```

The following list describes the input event fields:

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from the Lambda function. Agents for Amazon Bedrock only supports version 1.0.
- `agent` – Contains information about the name, ID, alias, and version of the agent that the prompts belongs to.
- `invokeModelRawResponse` – The raw foundation model output of the prompt whose output is to be parsed.
- `promptType` – The prompt type whose output is to be parsed.
- `overrideType` – The artifacts that this Lambda function overrides. Currently, only `OUTPUT_PARSER` is supported, which indicates that the default parser is to be overridden.

Parser Lambda response

Your agent expects a response from your Lambda function that matches the following format. The agent uses the response for further orchestration or to help it return a response to the user. Use the Lambda function response fields to configure how the output is returned.

```
{
    "messageVersion": "1.0",
    "promptType": "ORCHESTRATION | PRE_PROCESSING | POST_PROCESSING |
 KNOWLEDGE_BASE_RESPONSE_GENERATION",
    "preProcessingParsedResponse": {
        "isValidInput": "boolean",
        "rationale": "string"
    },
    "orchestrationParsedResponse": {
        "rationale": "string",
```

```
        "parsingErrorDetails": {
            "repromptResponse": "string"
        },
        "responseDetails": {
            "invocationType": "ACTION_GROUP | KNOWLEDGE_BASE | FINISH | ASK_USER",
            "agentAskUser": {
                "responseText": "string"
            },
            "actionGroupInvocation": {
                "actionGroupName": "string",
                "apiName": "string",
                "verb": "string",
                "actionGroupInput": {
                    "<parameter>": {
                        "value": "string"
                    },
                    ...
                }
            },
            "agentKnowledgeBase": {
                "knowledgeBaseId": "string",
                "searchQuery": {
                    "value": "string"
                }
            },
            "agentFinalResponse": {
                "responseText": "string",
                "citations": {
                    "generatedResponseParts": [{
                        "text": "string",
                        "references": [{"sourceId": "string"}]
                    }]
                }
            },
        }
    },
    "knowledgeBaseResponseGenerationParsedResponse": {
       "generatedResponse": {
            "generatedResponseParts": [
                {
                    "text": "string",
                    "references": [
                        {"sourceId": "string"},
                        ...
```

```
                    ]
                }
            ]
        }
    },
    "postProcessingParsedResponse": {
        "responseText": "string",
        "citations": {
            "generatedResponseParts": [{
                "text": "string",
                "references": [{
                    "sourceId": "string"
                }]
            }]
        }
    }
}
```

The following list describes the Lambda response fields:

- `messageVersion` – The version of the message that identifies the format of the event data going into the Lambda function and the expected format of the response from a Lambda function. Agents for Amazon Bedrock only supports version 1.0.
- `promptType` – The prompt type of the current turn.
- `preProcessingParsedResponse` – The parsed response for the `PRE_PROCESSING` prompt type.
- `orchestrationParsedResponse` – The parsed response for the `ORCHESTRATION` prompt type. See below for more details.
- `knowledgeBaseResponseGenerationParsedResponse` – The parsed response for the `KNOWLEDGE_BASE_RESPONSE_GENERATION` prompt type.
- `postProcessingParsedResponse` – The parsed response for the `POST_PROCESSING` prompt type.

For more details about the parsed responses for the four prompt templates, see the following tabs.

preProcessingParsedResponse

```
{
    "isValidInput": "boolean",
```

```
    "rationale": "string"
}
```

The `preProcessingParsedResponse` contains the following fields.

- `isValidInput` – Specifies whether the user input is valid or not. You can define the function to determine how to characterize the validity of user input.
- `rationale` – The reasoning for the user input categorization. This rationale is provided by the model in the raw response, the Lambda function parses it, and the agent presents it in the trace for pre-processing.

orchestrationResponse

```
{
    "rationale": "string",
    "parsingErrorDetails": {
        "repromptResponse": "string"
    },
    "responseDetails": {
        "invocationType": "ACTION_GROUP | KNOWLEDGE_BASE | FINISH | ASK_USER",
        "agentAskUser": {
            "responseText": "string"
        },
        "actionGroupInvocation": {
            "actionGroupName": "string",
            "apiName": "string",
            "verb": "string",
            "actionGroupInput": {
                "<parameter>": {
                    "value": "string"
                },
                ...
            }
        },
        "agentKnowledgeBase": {
            "knowledgeBaseId": "string",
            "searchQuery": {
                "value": "string"
            }
        },
        "agentFinalResponse": {
            "responseText": "string",
```

```
            "citations": {
                "generatedResponseParts": [
                    {
                        "text": "string",
                        "references": [
                            {"sourceId": "string"},
                            ...
                        ]
                    },
                    ...
                ]
            }
        },
    }
}
```

The `orchestrationParsedResponse` contains the following fields:

- `rationale` – The reasoning for what to do next, based on the foundation model output. You can define the function to parse from the model output.
- `parsingErrorDetails` – Contains the `repromptResponse`, which is the message to reprompt the model to update its raw response when the model response can't be parsed. You can define the function to manipulate how to reprompt the model.
- `responseDetails` – Contains the details for how to handle the output of the foundation model. Contains an `invocationType`, which is the next step for the agent to take, and a second field that should match the `invocationType`. The following objects are possible.
  - `agentAskUser` – Compatible with the `ASK_USER` invocation type. This invocation type ends the orchestration step. Contains the `responseText` to ask the user for more information. You can define your function to manipulate this field.
  - `actionGroupInvocation` – Compatible with the `ACTION_GROUP` invocation type. You can define your function to determine action groups to invoke and parameters to pass. Contains the following fields:
    - `actionGroupName` – The action group to invoke.
    - `apiName` – The name of the API operation to invoke in the action group.
    - `verb` – The method of the API operation to use.
    - `actionGroupInput` – Contains parameters to specify in the API operation request.

- `agentKnowledgeBase` – Compatible with the `KNOWLEDGE_BASE` invocation type. You can define your function to determine how to query knowledge bases. Contains the following fields:
  - `knowledgeBaseId` – The unique identifier of the knowledge base.
  - `searchQuery` – Contains the query to send to the knowledge base in the `value` field.
- `agentFinalResponse` – Compatible with the `FINISH` invocation type. This invocation type ends the orchestration step. Contains the response to the user in the `responseText` field and citations for the response in the `citations` object.

knowledgeBaseResponseGenerationParsedResponse

```
{
   "generatedResponse": {
        "generatedResponseParts": [
            {
                "text": "string",
                "references": [
                    { "sourceId": "string" },
                    ...
                ]
            },
            ...
        ]
    }
}
```

The `knowledgeBaseResponseGenerationParsedResponse` contains the `generatedResponse` from querying the knowledge base and references for the data sources.

postProcessingParsedResponse

```
{
    "responseText": "string",
    "citations": {
        "generatedResponseParts": [
            {
                "text": "string",
                "references": [
                    { "sourceId": "string" },
                    ...
```

```
                ]
            },
            ...
        ]
    }
}
```

The `postProcessingParsedResponse` contains the following fields:

- `responseText` – The response to return to the end user. You can define the function to format the response.
- `citations` – Contains a list of citations for the response. Each citation shows the cited text and its references.

Parser Lambda examples

To see an example parser Lambda function for a specific prompt template, select from the following tabs. Also shown are example input events sent to the function and responses from it. The `lambda_handler` function returns the parsed response to the agent.

Pre-processing

Example function

```
import json
import re
import logging

PRE_PROCESSING_RATIONALE_REGEX = "<thinking>(.*?)</thinking>"
PREPROCESSING_CATEGORY_REGEX = "<category>(.*?)</category>"
PREPROCESSING_PROMPT_TYPE = "PRE_PROCESSING"
PRE_PROCESSING_RATIONALE_PATTERN = re.compile(PRE_PROCESSING_RATIONALE_REGEX,
 re.DOTALL)
PREPROCESSING_CATEGORY_PATTERN = re.compile(PREPROCESSING_CATEGORY_REGEX, re.DOTALL)

logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default
 PreProcessing prompt
def lambda_handler(event, context):
```

```
    print("Lambda input: " + str(event))
    logger.info("Lambda input: " + str(event))

    prompt_type = event["promptType"]

    # Sanitize LLM response
    model_response = sanitize_response(event['invokeModelRawResponse'])

    if event["promptType"] == PREPROCESSING_PROMPT_TYPE:
        return parse_pre_processing(model_response)

def parse_pre_processing(model_response):

    category_matches = re.finditer(PREPROCESSING_CATEGORY_PATTERN, model_response)
    rationale_matches = re.finditer(PRE_PROCESSING_RATIONALE_PATTERN,
 model_response)

    category = next((match.group(1) for match in category_matches), None)
    rationale = next((match.group(1) for match in rationale_matches), None)

    return {
        "promptType": "PRE_PROCESSING",
        "preProcessingParsedResponse": {
            "rationale": rationale,
            "isValidInput": get_is_valid_input(category)
            }
        }

def sanitize_response(text):
    pattern = r"(\\n*)"
    text = re.sub(pattern, r"\n", text)
    return text

def get_is_valid_input(category):
    if category is not None and category.strip().upper() == "D" or
 category.strip().upper() == "E":
        return True
    return False
```

```
{
    "agent": {
        "alias": "TSTALIASID",
```

```
        "id": "AGENTID123",
        "name": "InsuranceAgent",
        "version": "DRAFT"
    },
    "invokeModelRawResponse": " <thinking>\nThe user is asking about the
 instructions provided to the function calling agent. This input is trying to gather
 information about what functions/API's or instructions our function calling agent
 has access to. Based on the categories provided, this input belongs in Category B.
\n</thinking>\n\n<category>B</category>",
    "messageVersion": "1.0",
    "overrideType": "OUTPUT_PARSER",
    "promptType": "PRE_PROCESSING"
}
```

```
{
  "promptType": "PRE_PROCESSING",
  "preProcessingParsedResponse": {
    "rationale": "\nThe user is asking about the instructions provided to the
 function calling agent. This input is trying to gather information about what
 functions/API's or instructions our function calling agent has access to. Based on
 the categories provided, this input belongs in Category B.\n",
    "isValidInput": false
  }
}
```

Orchestration

The following are example functions for Anthropic Claude 2 and Anthropic Claude 2.1.

Anthropic Claude 2

```
import json
import re
import logging


RATIONALE_REGEX_LIST = [
    "(.*?)(<function_call>)",
    "(.*?)(<answer>)"
]
RATIONALE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_REGEX_LIST]
```

```
RATIONALE_VALUE_REGEX_LIST = [
    "<scratchpad>(.*?)(</scratchpad>)",
    "(.*?)(</scratchpad>)",
    "(<scratchpad>)(.*?)"
]
RATIONALE_VALUE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_VALUE_REGEX_LIST]

ANSWER_REGEX = r"(?<=<answer>)(.*)"
ANSWER_PATTERN = re.compile(ANSWER_REGEX, re.DOTALL)

ANSWER_TAG = "<answer>"
FUNCTION_CALL_TAG = "<function_call>"

ASK_USER_FUNCTION_CALL_REGEX = r"(<function_call>user::askuser)(.*)\)"
ASK_USER_FUNCTION_CALL_PATTERN = re.compile(ASK_USER_FUNCTION_CALL_REGEX, re.DOTALL)

ASK_USER_FUNCTION_PARAMETER_REGEX = r"(?<=askuser=\")(.*?)\""
ASK_USER_FUNCTION_PARAMETER_PATTERN = re.compile(ASK_USER_FUNCTION_PARAMETER_REGEX,
 re.DOTALL)

KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX = "x_amz_knowledgebase_"

FUNCTION_CALL_REGEX = r"<function_call>(\w+)::(\w+)::(.+)\((.+)\)"

ANSWER_PART_REGEX = "<answer_part\\s?>(.+?)</answer_part\\s?>"
ANSWER_TEXT_PART_REGEX = "<text\\s?>(.+?)</text\\s?>"
ANSWER_REFERENCE_PART_REGEX = "<source\\s?>(.+?)</source\\s?>"
ANSWER_PART_PATTERN = re.compile(ANSWER_PART_REGEX, re.DOTALL)
ANSWER_TEXT_PART_PATTERN = re.compile(ANSWER_TEXT_PART_REGEX, re.DOTALL)
ANSWER_REFERENCE_PART_PATTERN = re.compile(ANSWER_REFERENCE_PART_REGEX, re.DOTALL)

# You can provide messages to reprompt the LLM in case the LLM output is not in the
 expected format
MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE = "Missing the argument askuser for
 user::askuser function call. Please try again with the correct argument added"
ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE = "The function call format
 is incorrect. The format for function calls to the askuser function must be:
 <function_call>user::askuser(askuser=\"$ASK_USER_INPUT\")</function_call>."
FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE = 'The function call format
 is incorrect. The format for function calls must be: <function_call>
$FUNCTION_NAME($FUNCTION_ARGUMENT_NAME=""$FUNCTION_ARGUMENT_NAME"")</
function_call>.'
```

```
logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default
 orchestration prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))

    # Sanitize LLM response
    sanitized_response = sanitize_response(event['invokeModelRawResponse'])

    # Parse LLM response for any rationale
    rationale = parse_rationale(sanitized_response)

    # Construct response fields common to all invocation types
    parsed_response = {
        'promptType': "ORCHESTRATION",
        'orchestrationParsedResponse': {
            'rationale': rationale
        }
    }

    # Check if there is a final answer
    try:
        final_answer, generated_response_parts = parse_answer(sanitized_response)
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    if final_answer:
        parsed_response['orchestrationParsedResponse']['responseDetails'] = {
            'invocationType': 'FINISH',
            'agentFinalResponse': {
                'responseText': final_answer
            }
        }

        if generated_response_parts:
            parsed_response['orchestrationParsedResponse']['responseDetails']
['agentFinalResponse']['citations'] = {
                'generatedResponseParts': generated_response_parts
            }

        logger.info("Final answer parsed response: " + str(parsed_response))
        return parsed_response
```

```
    # Check if there is an ask user
    try:
        ask_user = parse_ask_user(sanitized_response)
        if ask_user:
            parsed_response['orchestrationParsedResponse']['responseDetails'] = {
                'invocationType': 'ASK_USER',
                'agentAskUser': {
                    'responseText': ask_user
                }
            }

            logger.info("Ask user parsed response: " + str(parsed_response))
            return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    # Check if there is an agent action
    try:
        parsed_response = parse_function_call(sanitized_response, parsed_response)
        logger.info("Function call parsed response: " + str(parsed_response))
        return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    addRepromptResponse(parsed_response, 'Failed to parse the LLM output')
    logger.info(parsed_response)
    return parsed_response

    raise Exception("unrecognized prompt type")

def sanitize_response(text):
    pattern = r"(\\n*)"
    text = re.sub(pattern, r"\n", text)
    return text

def parse_rationale(sanitized_response):
    # Checks for strings that are not required for orchestration
    rationale_matcher = next((pattern.search(sanitized_response) for pattern in
 RATIONALE_PATTERNS if pattern.search(sanitized_response)), None)

    if rationale_matcher:
```

```
        rationale = rationale_matcher.group(1).strip()

        # Check if there is a formatted rationale that we can parse from the string
        rationale_value_matcher = next((pattern.search(rationale) for pattern in
 RATIONALE_VALUE_PATTERNS if pattern.search(rationale)), None)
        if rationale_value_matcher:
            return rationale_value_matcher.group(1).strip()

        return rationale

    return None

def parse_answer(sanitized_llm_response):
    if has_generated_response(sanitized_llm_response):
        return parse_generated_response(sanitized_llm_response)

    answer_match = ANSWER_PATTERN.search(sanitized_llm_response)
    if answer_match and is_answer(sanitized_llm_response):
        return answer_match.group(0).strip(), None

    return None, None

def is_answer(llm_response):
    return llm_response.rfind(ANSWER_TAG) > llm_response.rfind(FUNCTION_CALL_TAG)

def parse_generated_response(sanitized_llm_response):
    results = []

    for match in ANSWER_PART_PATTERN.finditer(sanitized_llm_response):
        part = match.group(1).strip()

        text_match = ANSWER_TEXT_PART_PATTERN.search(part)
        if not text_match:
            raise ValueError("Could not parse generated response")

        text = text_match.group(1).strip()
        references = parse_references(sanitized_llm_response, part)
        results.append((text, references))

    final_response = " ".join([r[0] for r in results])

    generated_response_parts = []
    for text, references in results:
        generatedResponsePart = {
```

```
            'text': text,
            'references': references
        }
        generated_response_parts.append(generatedResponsePart)

    return final_response, generated_response_parts


def has_generated_response(raw_response):
    return ANSWER_PART_PATTERN.search(raw_response) is not None

def parse_references(raw_response, answer_part):
    references = []
    for match in ANSWER_REFERENCE_PART_PATTERN.finditer(answer_part):
        reference = match.group(1).strip()
        references.append({'sourceId': reference})
    return references

def parse_ask_user(sanitized_llm_response):
    ask_user_matcher = ASK_USER_FUNCTION_CALL_PATTERN.search(sanitized_llm_response)
    if ask_user_matcher:
        try:
            ask_user = ask_user_matcher.group(2).strip()
            ask_user_question_matcher =
 ASK_USER_FUNCTION_PARAMETER_PATTERN.search(ask_user)
            if ask_user_question_matcher:
                return ask_user_question_matcher.group(1).strip()
            raise ValueError(MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE)
        except ValueError as ex:
            raise ex
        except Exception as ex:
            raise Exception(ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE)

    return None

def parse_function_call(sanitized_response, parsed_response):
    match = re.search(FUNCTION_CALL_REGEX, sanitized_response)
    if not match:
        raise ValueError(FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE)

    verb, resource_name, function = match.group(1), match.group(2), match.group(3)

    parameters = {}
    for arg in match.group(4).split(","):
```

```
        key, value = arg.split("=")
        parameters[key.strip()] = {'value': value.strip('" ')}

    parsed_response['orchestrationParsedResponse']['responseDetails'] = {}

    # Function calls can either invoke an action group or a knowledge base.
    # Mapping to the correct variable names accordingly
    if resource_name.lower().startswith(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX):
        parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'KNOWLEDGE_BASE'
        parsed_response['orchestrationParsedResponse']['responseDetails']
['agentKnowledgeBase'] = {
            'searchQuery': parameters['searchQuery'],
            'knowledgeBaseId':
 resource_name.replace(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX, '')
        }

        return parsed_response

    parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'ACTION_GROUP'
    parsed_response['orchestrationParsedResponse']['responseDetails']
['actionGroupInvocation'] = {
        "verb": verb,
        "actionGroupName": resource_name,
        "apiName": function,
        "actionGroupInput": parameters
    }

    return parsed_response

def addRepromptResponse(parsed_response, error):
    error_message = str(error)
    logger.warn(error_message)

    parsed_response['orchestrationParsedResponse']['parsingErrorDetails'] = {
        'repromptResponse': error_message
    }
```

Anthropic Claude 2.1

```
import logging
import re
```

```
import xml.etree.ElementTree as ET

RATIONALE_REGEX_LIST = [
    "(.*?)(<function_calls>)",
    "(.*?)(<answer>)"
]
RATIONALE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_REGEX_LIST]

RATIONALE_VALUE_REGEX_LIST = [
    "<scratchpad>(.*?)(</scratchpad>)",
    "(.*?)(</scratchpad>)",
    "(<scratchpad>)(.*?)"
]
RATIONALE_VALUE_PATTERNS = [re.compile(regex, re.DOTALL) for regex in
 RATIONALE_VALUE_REGEX_LIST]

ANSWER_REGEX = r"(?<=<answer>)(.*)"
ANSWER_PATTERN = re.compile(ANSWER_REGEX, re.DOTALL)

ANSWER_TAG = "<answer>"
FUNCTION_CALL_TAG = "<function_calls>"

ASK_USER_FUNCTION_CALL_REGEX = r"<tool_name>user::askuser</tool_name>"
ASK_USER_FUNCTION_CALL_PATTERN = re.compile(ASK_USER_FUNCTION_CALL_REGEX, re.DOTALL)

ASK_USER_TOOL_NAME_REGEX = r"<tool_name>((.|\n)*?)</tool_name>"
ASK_USER_TOOL_NAME_PATTERN = re.compile(ASK_USER_TOOL_NAME_REGEX, re.DOTALL)

TOOL_PARAMETERS_REGEX = r"<parameters>((.|\n)*?)</parameters>"
TOOL_PARAMETERS_PATTERN = re.compile(TOOL_PARAMETERS_REGEX, re.DOTALL)

ASK_USER_TOOL_PARAMETER_REGEX = r"<question>((.|\n)*?)</question>"
ASK_USER_TOOL_PARAMETER_PATTERN = re.compile(ASK_USER_TOOL_PARAMETER_REGEX,
 re.DOTALL)


KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX = "x_amz_knowledgebase_"

FUNCTION_CALL_REGEX = r"(?<=<function_calls>)(.*)"

ANSWER_PART_REGEX = "<answer_part\\s?>(.+?)</answer_part\\s?>"
ANSWER_TEXT_PART_REGEX = "<text\\s?>(.+?)</text\\s?>"
ANSWER_REFERENCE_PART_REGEX = "<source\\s?>(.+?)</source\\s?>"
```

```
ANSWER_PART_PATTERN = re.compile(ANSWER_PART_REGEX, re.DOTALL)
ANSWER_TEXT_PART_PATTERN = re.compile(ANSWER_TEXT_PART_REGEX, re.DOTALL)
ANSWER_REFERENCE_PART_PATTERN = re.compile(ANSWER_REFERENCE_PART_REGEX, re.DOTALL)

# You can provide messages to reprompt the LLM in case the LLM output is not in the
 expected format
MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE = "Missing the parameter 'question' for
 user::askuser function call. Please try again with the correct argument added."
ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE = "The function call format is
 incorrect. The format for function calls to the askuser function must be: <invoke>
 <tool_name>user::askuser</tool_name><parameters><question>$QUESTION</question></
parameters></invoke>."
FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE = "The function call format is incorrect.
 The format for function calls must be: <invoke> <tool_name>$TOOL_NAME</tool_name>
 <parameters> <$PARAMETER_NAME>$PARAMETER_VALUE</$PARAMETER_NAME>...</parameters></
invoke>."

logger = logging.getLogger()


# This parser lambda is an example of how to parse the LLM output for the default
 orchestration prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))

    # Sanitize LLM response
    sanitized_response = sanitize_response(event['invokeModelRawResponse'])

    # Parse LLM response for any rationale
    rationale = parse_rationale(sanitized_response)

    # Construct response fields common to all invocation types
    parsed_response = {
        'promptType': "ORCHESTRATION",
        'orchestrationParsedResponse': {
            'rationale': rationale
        }
    }

    # Check if there is a final answer
    try:
        final_answer, generated_response_parts = parse_answer(sanitized_response)
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
```

```
        return parsed_response

    if final_answer:
        parsed_response['orchestrationParsedResponse']['responseDetails'] = {
            'invocationType': 'FINISH',
            'agentFinalResponse': {
                'responseText': final_answer
            }
        }

        if generated_response_parts:
            parsed_response['orchestrationParsedResponse']['responseDetails']
['agentFinalResponse']['citations'] = {
                'generatedResponseParts': generated_response_parts
            }

        logger.info("Final answer parsed response: " + str(parsed_response))
        return parsed_response

    # Check if there is an ask user
    try:
        ask_user = parse_ask_user(sanitized_response)
        if ask_user:
            parsed_response['orchestrationParsedResponse']['responseDetails'] = {
                'invocationType': 'ASK_USER',
                'agentAskUser': {
                    'responseText': ask_user
                }
            }

            logger.info("Ask user parsed response: " + str(parsed_response))
            return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response

    # Check if there is an agent action
    try:
        parsed_response = parse_function_call(sanitized_response, parsed_response)
        logger.info("Function call parsed response: " + str(parsed_response))
        return parsed_response
    except ValueError as e:
        addRepromptResponse(parsed_response, e)
        return parsed_response
```

```
    addRepromptResponse(parsed_response, 'Failed to parse the LLM output')
    logger.info(parsed_response)
    return parsed_response

    raise Exception("unrecognized prompt type")


def sanitize_response(text):
    pattern = r"(\\n*)"
    text = re.sub(pattern, r"\n", text)
    return text


def parse_rationale(sanitized_response):
    # Checks for strings that are not required for orchestration
    rationale_matcher = next(
        (pattern.search(sanitized_response) for pattern in RATIONALE_PATTERNS if
 pattern.search(sanitized_response)),
        None)

    if rationale_matcher:
        rationale = rationale_matcher.group(1).strip()

        # Check if there is a formatted rationale that we can parse from the string
        rationale_value_matcher = next(
            (pattern.search(rationale) for pattern in RATIONALE_VALUE_PATTERNS if
 pattern.search(rationale)), None)
        if rationale_value_matcher:
            return rationale_value_matcher.group(1).strip()

        return rationale

    return None


def parse_answer(sanitized_llm_response):
    if has_generated_response(sanitized_llm_response):
        return parse_generated_response(sanitized_llm_response)

    answer_match = ANSWER_PATTERN.search(sanitized_llm_response)
    if answer_match and is_answer(sanitized_llm_response):
        return answer_match.group(0).strip(), None
```

```
    return None, None


def is_answer(llm_response):
    return llm_response.rfind(ANSWER_TAG) > llm_response.rfind(FUNCTION_CALL_TAG)


def parse_generated_response(sanitized_llm_response):
    results = []

    for match in ANSWER_PART_PATTERN.finditer(sanitized_llm_response):
        part = match.group(1).strip()

        text_match = ANSWER_TEXT_PART_PATTERN.search(part)
        if not text_match:
            raise ValueError("Could not parse generated response")

        text = text_match.group(1).strip()
        references = parse_references(sanitized_llm_response, part)
        results.append((text, references))

    final_response = " ".join([r[0] for r in results])

    generated_response_parts = []
    for text, references in results:
        generatedResponsePart = {
            'text': text,
            'references': references
        }
        generated_response_parts.append(generatedResponsePart)

    return final_response, generated_response_parts


def has_generated_response(raw_response):
    return ANSWER_PART_PATTERN.search(raw_response) is not None


def parse_references(raw_response, answer_part):
    references = []
    for match in ANSWER_REFERENCE_PART_PATTERN.finditer(answer_part):
        reference = match.group(1).strip()
        references.append({'sourceId': reference})
    return references
```

```
def parse_ask_user(sanitized_llm_response):
    ask_user_matcher = ASK_USER_FUNCTION_CALL_PATTERN.search(sanitized_llm_response)
    if ask_user_matcher:
        try:
            parameters_matches =
 TOOL_PARAMETERS_PATTERN.search(sanitized_llm_response)
            params = parameters_matches.group(1).strip()
            ask_user_question_matcher =
 ASK_USER_TOOL_PARAMETER_PATTERN.search(params)
            if ask_user_question_matcher:
                ask_user_question = ask_user_question_matcher.group(1)
                return ask_user_question
            raise ValueError(MISSING_API_INPUT_FOR_USER_REPROMPT_MESSAGE)
        except ValueError as ex:
            raise ex
        except Exception as ex:
            raise Exception(ASK_USER_FUNCTION_CALL_STRUCTURE_REMPROMPT_MESSAGE)

    return None


def parse_function_call(sanitized_response, parsed_response):
    match = re.search(FUNCTION_CALL_REGEX, sanitized_response)
    if not match:
        raise ValueError(FUNCTION_CALL_STRUCTURE_REPROMPT_MESSAGE)

    tool_name_matches = ASK_USER_TOOL_NAME_PATTERN.search(sanitized_response)
    tool_name = tool_name_matches.group(1)
    parameters_matches = TOOL_PARAMETERS_PATTERN.search(sanitized_response)
    params = parameters_matches.group(1).strip()

    action_split = tool_name.split('::')
    verb = action_split[0].strip()
    resource_name = action_split[1].strip()
    function = action_split[2].strip()

    xml_tree = ET.ElementTree(ET.fromstring("<parameters>{}</
parameters>".format(params)))
    parameters = {}
    for elem in xml_tree.iter():
        if elem.text:
            parameters[elem.tag] = {'value': elem.text.strip('" ')}
```

```
    parsed_response['orchestrationParsedResponse']['responseDetails'] = {}

    # Function calls can either invoke an action group or a knowledge base.
    # Mapping to the correct variable names accordingly
    if resource_name.lower().startswith(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX):
        parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'KNOWLEDGE_BASE'
        parsed_response['orchestrationParsedResponse']['responseDetails']
['agentKnowledgeBase'] = {
            'searchQuery': parameters['searchQuery'],
            'knowledgeBaseId':
 resource_name.replace(KNOWLEDGE_STORE_SEARCH_ACTION_PREFIX, '')
        }

        return parsed_response

    parsed_response['orchestrationParsedResponse']['responseDetails']
['invocationType'] = 'ACTION_GROUP'
    parsed_response['orchestrationParsedResponse']['responseDetails']
['actionGroupInvocation'] = {
        "verb": verb,
        "actionGroupName": resource_name,
        "apiName": function,
        "actionGroupInput": parameters
    }

    return parsed_response


def addRepromptResponse(parsed_response, error):
    error_message = str(error)
    logger.warn(error_message)

    parsed_response['orchestrationParsedResponse']['parsingErrorDetails'] = {
        'repromptResponse': error_message
    }
```

Example request

```
{
    'agent': {
        'alias': 'TSTALIASID',
```

```
        'id': 'AGENTID123',
        'name': 'InsuranceAgent',
        'version': 'DRAFT'
    },
    'invokeModelRawResponse': ' To answer this question, I will:\\n\\n1.
 Call the GET::x_amz_knowledgebase_KBID123456::Search function to search
 for a phone number to call.\\n\\nI have checked that I have access to the
 GET::x_amz_knowledgebase_KBID23456::Search function.\\n\\n</scratchpad>\\n\
\n<function_call>GET::x_amz_knowledgebase_KBID123456::Search(searchQuery=\"What is
 the phone number I can call?\")',
    'messageVersion': '1.0',
    'overrideType': 'OUTPUT_PARSER',
    'promptType': 'ORCHESTRATION'
}
```

Example response

```
{
    'promptType': 'ORCHESTRATION',
    'orchestrationParsedResponse': {
        'rationale': 'To answer this question, I will:\\n\\n1. Call the
 GET::x_amz_knowledgebase_KBID123456::Search function to search for a phone
 number to call Farmers.\\n\\nI have checked that I have access to the
 GET::x_amz_knowledgebase_KBID123456::Search function.',
        'responseDetails': {
            'invocationType': 'KNOWLEDGE_BASE',
            'agentKnowledgeBase': {
                'searchQuery': {'value': 'What is the phone number I can call?'},
                'knowledgeBaseId': 'KBID123456'
            }
        }
    }
}
```

Knowledge base response generation

Example function

```
import json
import re
import logging

ANSWER_PART_REGEX = "<answer_part\\s?>(.+?)</answer_part\\s?>"
```

```
ANSWER_TEXT_PART_REGEX = "<text\\s?>(.+?)</text\\s?>"
ANSWER_REFERENCE_PART_REGEX = "<source\\s?>(.+?)</source\\s?>"
ANSWER_PART_PATTERN = re.compile(ANSWER_PART_REGEX, re.DOTALL)
ANSWER_TEXT_PART_PATTERN = re.compile(ANSWER_TEXT_PART_REGEX, re.DOTALL)
ANSWER_REFERENCE_PART_PATTERN = re.compile(ANSWER_REFERENCE_PART_REGEX, re.DOTALL)

logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default KB
 response generation prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))
    raw_response = event['invokeModelRawResponse']

    parsed_response = {
        'promptType': 'KNOWLEDGE_BASE_RESPONSE_GENERATION',
        'knowledgeBaseResponseGenerationParsedResponse': {
            'generatedResponse': parse_generated_response(raw_response)
        }
    }

    logger.info(parsed_response)
    return parsed_response

def parse_generated_response(sanitized_llm_response):
    results = []

    for match in ANSWER_PART_PATTERN.finditer(sanitized_llm_response):
        part = match.group(1).strip()

        text_match = ANSWER_TEXT_PART_PATTERN.search(part)
        if not text_match:
            raise ValueError("Could not parse generated response")

        text = text_match.group(1).strip()
        references = parse_references(sanitized_llm_response, part)
        results.append((text, references))

    generated_response_parts = []
    for text, references in results:
        generatedResponsePart = {
            'text': text,
            'references': references
        }
```

```
        generated_response_parts.append(generatedResponsePart)

    return {
        'generatedResponseParts': generated_response_parts
    }

def parse_references(raw_response, answer_part):
    references = []
    for match in ANSWER_REFERENCE_PART_PATTERN.finditer(answer_part):
        reference = match.group(1).strip()
        references.append({'sourceId': reference})
    return references
```

Example request

```
{
    'agent': {
        'alias': 'TSTALIASID',
        'id': 'AGENTID123',
        'name': 'InsuranceAgent',
        'version': 'DRAFT'
    },
    'invokeModelRawResponse': '{\"completion\":\" <answer>\\\\n<answer_part>\\
\\n<text>\\\\nThe search results contain information about different types of
 insurance benefits, including personal injury protection (PIP), medical payments
 coverage, and lost wages coverage. PIP typically covers reasonable medical
 expenses for injuries caused by an accident, as well as income continuation,
 child care, loss of services, and funerals. Medical payments coverage provides
 payment for medical treatment resulting from a car accident. Who pays lost wages
 due to injuries depends on the laws in your state and the coverage purchased.
\\\\n</text>\\\\n<sources>\\\\n<source>1234567-1234-1234-1234-123456789abc</
source>\\\\n<source>2345678-2345-2345-2345-23456789abcd</source>\\\
\n<source>3456789-3456-3456-3456-3456789abcde</source>\\\\n</sources>\\\\n</
answer_part>\\\\n</answer>\",\"stop_reason\":\"stop_sequence\",\"stop\":\"\\\\n\\\
\nHuman:\"}',
    'messageVersion': '1.0',
    'overrideType': 'OUTPUT_PARSER',
    'promptType': 'KNOWLEDGE_BASE_RESPONSE_GENERATION'
}
```

Example response

```
{
```

```
    'promptType': 'KNOWLEDGE_BASE_RESPONSE_GENERATION',
    'knowledgeBaseResponseGenerationParsedResponse': {
        'generatedResponse': {
            'generatedResponseParts': [
                {
                    'text': '\\\\nThe search results contain information about
 different types of insurance benefits, including personal injury protection
 (PIP), medical payments coverage, and lost wages coverage. PIP typically covers
 reasonable medical expenses for injuries caused by an accident, as well as income
 continuation, child care, loss of services, and funerals. Medical payments coverage
 provides payment for medical treatment resulting from a car accident. Who pays lost
 wages due to injuries depends on the laws in your state and the coverage purchased.
\\\\n',
                    'references': [
                        {'sourceId': '1234567-1234-1234-1234-123456789abc'},
                        {'sourceId': '2345678-2345-2345-2345-23456789abcd'},
                        {'sourceId': '3456789-3456-3456-3456-3456789abcde'}
                    ]
                }
            ]
        }
    }
}
```

Post-processing

Example function

```
import json
import re
import logging

FINAL_RESPONSE_REGEX = r"<final_response>([\s\S]*?)</final_response>"
FINAL_RESPONSE_PATTERN = re.compile(FINAL_RESPONSE_REGEX, re.DOTALL)

logger = logging.getLogger()

# This parser lambda is an example of how to parse the LLM output for the default
 PostProcessing prompt
def lambda_handler(event, context):
    logger.info("Lambda input: " + str(event))
    raw_response = event['invokeModelRawResponse']

    parsed_response = {
```

```
        'promptType': 'POST_PROCESSING',
        'postProcessingParsedResponse': {}
    }

    matcher = FINAL_RESPONSE_PATTERN.search(raw_response)
    if not matcher:
        raise Exception("Could not parse raw LLM output")
    response_text = matcher.group(1).strip()

    parsed_response['postProcessingParsedResponse']['responseText'] = response_text

    logger.info(parsed_response)
    return parsed_response
```

Example request

```
{
    'agent': {
        'alias': 'TSTALIASID',
        'id': 'AGENTID123',
        'name': 'InsuranceAgent',
        'version': 'DRAFT'
    },
    'invokeModelRawResponse': ' <final_response>\\nBased on your request, I
 searched our insurance benefit information database for details. The search
 results indicate that insurance policies may cover different types of benefits,
 depending on the policy and state laws. Specifically, the results discussed
 personal injury protection (PIP) coverage, which typically covers medical
 expenses for insured individuals injured in an accident (cited sources:
 1234567-1234-1234-1234-123456789abc, 2345678-2345-2345-2345-23456789abcd). PIP may
 pay for costs like medical care, lost income replacement, childcare expenses, and
 funeral costs. Medical payments coverage was also mentioned as another option that
 similarly covers medical treatment costs for the policyholder and others injured in
 a vehicle accident involving the insured vehicle. The search results further noted
 that whether lost wages are covered depends on the state and coverage purchased.
 Please let me know if you need any clarification or have additional questions.\\n</
final_response>',
    'messageVersion': '1.0',
    'overrideType': 'OUTPUT_PARSER',
    'promptType': 'POST_PROCESSING'
}
```

Example response

```
{
    'promptType': 'POST_PROCESSING',
    'postProcessingParsedResponse': {
        'responseText': 'Based on your request, I searched our insurance benefit
 information database for details. The search results indicate that insurance
 policies may cover different types of benefits, depending on the policy and
 state laws. Specifically, the results discussed personal injury protection
 (PIP) coverage, which typically covers medical expenses for insured individuals
 injured in an accident (cited sources: 24c62d8c-3e39-4ca1-9470-a91d641fe050,
 197815ef-8798-4cb1-8aa5-35f5d6b28365). PIP may pay for costs like medical care,
 lost income replacement, childcare expenses, and funeral costs. Medical payments
 coverage was also mentioned as another option that similarly covers medical
 treatment costs for the policyholder and others injured in a vehicle accident
 involving the insured vehicle. The search results further noted that whether lost
 wages are covered depends on the state and coverage purchased. Please let me know
 if you need any clarification or have additional questions.'
    }
}
```

Control session context

For greater control of session context, you can modify the SessionState object in your agent. The SessionState object contains two types of attributes that you can use to provide conversational context for the agent during user conversations.

- **sessionAttributes** – Attributes that persist over a session between a user and agent. All InvokeAgent requests made with the same `sessionId` belong to the same session, as long as the session time limit (the `idleSessionTTLinSeconds`) has not been surpassed.
- **promptSessionAttributes** – Attributes that persist over a single turn (one InvokeAgent call). You can use the $prompt_session_attributes$ placeholder when you edit the orchestration base prompt template. This placeholder will be populated at runtime with the attributes that you specify in the `promptSessionAttributes` field.

The general format of the SessionState object is as follows.

```
{
    "sessionAttributes": {
        "<attributeName1>": "<attributeValue1>",
        "<attributeName2>": "<attributeValue2>",
```

```
        ...
    },
    "promptSessionAttributes": {
        "<attributeName3>": "<attributeValue3>",
        "<attributeName4>": "<attributeValue4>",
        ...
    }
}
```

You can define the session state attributes at two different steps.

- When you set up an action group and write the Lambda function, include `sessionAttributes` or `promptSessionAttributes` in the response event that is returned to Amazon Bedrock.
- During runtime, when you send an InvokeAgent request, include a `sessionState` object in the request body to dynamically change the session state attributes in the middle of the conversation.

Session attribute example

The following example uses a session attribute to personalize a message to your user.

1. Write your application code to ask the user to provide their first name and the request they want to make to the agent and to store the answers as the variables *<first_name>* and *<request>*.
2. Write your application code to send an InvokeAgent request with the following body:

```
{
    "inputText": "<request>",
    "sessionState": {
        "sessionAttributes": {
            "firstName": "<first_name>"
        }
    }
}
```

3. When a user uses your application and provides their first name, your code will send the first name as a session attribute and the agent will store their first name for the duration of the session.

4. Because session attributes are sent in the Lambda input event, you can refer to these session attributes in a Lambda function for an action group. For example, if the action API schema requires a first name in the request body, you can use the `firstName` session attribute when writing the Lambda function for an action group to automatically populate that field when sending the API request.

Prompt session attribute example

The following general example uses a prompt session attribute to provide temporal context for the agent.

1. Write your application code to store the user request in a variable called *<request>*.
2. Write your application code to retrieve the time zone at the user's location if the user uses a word indicating relative time (such as "tomorrow") in the *<request>*, and store in a variable called *<timezone>*.
3. Write your application to send an InvokeAgent request with the following body:

```
{
    "inputText": "<request>",
    "sessionState": {
        "promptSessionAttributes": {
            "timeZone": "<timezone>"
        }
    }
}
```

4. If a user uses a word indicating relative time, your code will send the `timeZone` prompt session attribute and the agent will store it for the duration of the turn.
5. For example, if a user asks **`I need to book a hotel for tomorrow`**, your code sends the user's time zone to the agent and the agent can determine the exact date that "tomorrow" refers to.
6. The prompt session attribute can be used at the following steps.
   - If you include the $prompt_session_attributes$ placeholder in the orchestration prompt template, the orchestration prompt to the FM includes the prompt session attributes.
   - Prompt session attributes are sent in the Lambda input event and can be used to help populate API requests or returned in the response.

Optimize performance for Amazon Bedrock agents

This topic provides describes optimizations for agents with specific use cases.

Topics

Optimize performance for Amazon Bedrock agents using a single knowledge base

Agents for Amazon Bedrock offers options to choose different flows that can optimize on latency for simpler use cases in which agents have a single knowledge base. To ensure that your agent is able to take advantage of this optimization, check that the following conditions apply to the relevant version of your agent:

- Your agent contains only one knowledge base.
- Your agent contains no action groups or they are all disabled.
- Your agent doesn't request more information from the user if it doesn't have enough information.
- Your agent is using the default orchestration prompt template.

To learn how to check for these conditions, select the tab corresponding to your method of choice and follow the steps.

Console

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Agent overview** section, check that the **User input** field is **DISABLED**.
4. If you're checking if the optimization is being applied to the working draft of the agent, select the **Working draft** in the **Working draft** section. If you're checking if the optimization is being applied to a version of the agent, select the version in the **Versions** section.
5. Check that the **Knowledge bases** section contains only one knowledge base. If there's more than one knowledge base, disable all of them except for one. To learn how to disable knowledge bases, see Manage agent-knowledge bases associations.

6. Check that the **Action groups** section contains no action groups. If there are action groups, disable all of them. To learn how to disable action groups, see Edit an action group.
7. In the **Advanced prompts** section, check that the **Orchestration** field value is **Default**. If it's **Overridden**, choose **Edit** (if you're viewing a version of your agent, you must first navigate to the working draft) and do the following:
   a. In the **Advanced prompts** section, select the **Orchestration** tab.
   b. If you revert the template to the default settings, your custom prompt template will be deleted. Make sure to save your template if you need it later.
   c. Clear **Override orchestration template defaults**. Confirm the message that appears.
8. To apply any changes you've made, select **Prepare** at the top of the **Agent details** page or in the test window. Then, test the agent's optimized performance by submitting a message in the test window.
9. (Optional) If necessary, create a new version of your agent by following the steps at Deploy an Amazon Bedrock agent.

API

1. Send a ListAgentKnowledgeBases request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the ID of your agent. For the `agentVersion`, use `DRAFT` for the working draft or specify the relevant version. In the response, check that `agentKnowledgeBaseSummaries` contains only one object (corresponding to one knowledge base). If there's more than one knowledge base, disable all of them except for one. To learn how to disable knowledge bases, see Manage agent-knowledge bases associations.
2. Send a ListAgentActionGroups request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the ID of your agent. For the `agentVersion`, use `DRAFT` for the working draft or specify the relevant version. In the response, check that the `actionGroupSummaries` list is empty. If there are action groups, disable all of them. To learn how to disable action groups, see Edit an action group.
3. Send a GetAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the ID of your agent. In the response, within the `promptConfigurations` list in the `promptOverrideConfiguration` field, look for the PromptConfiguration object whose

promptType value is ORCHESTRATION. If the promptCreationMode value is DEFAULT, you don't have to do anything. If it's OVERRIDDEN, do the following to revert the template to the default settings:

a. If you revert the template to the default settings, your custom prompt template will be deleted. Make sure to save your template from the basePromptTemplate field if you need it later.

b. Send an UpdateAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. For the PromptConfiguration object corresponding to the orchestration template, set the value of promptCreationMode to DEFAULT.

4. To apply any changes you've made, send a PrepareAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Then, test the agent's optimized performance by submitting an InvokeAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint, using the TSTALIASID alias of the agent.
5. (Optional) If necessary, create a new version of your agent by following the steps at Deploy an Amazon Bedrock agent.

Deploy an Amazon Bedrock agent

When you first create an Amazon Bedrock agent, you have a working draft version (DRAFT) and a test alias (TSTALIASID) that points to the working draft version. When you make changes to your agent, the changes apply to the working draft. You iterate on your working draft until you're satisfied with the behavior of your agent. Then, you can set up your agent for deployment and integration into your application by creating *aliases* of your agent.

To deploy your agent, you must create an *alias*. During alias creation, Amazon Bedrock creates a version of your agent automatically. The alias points to this newly created version. Alternatively, you can point the alias to a previously created version of your agent. Then, you configure your application to make API calls to that alias.

A *version* is a snapshot that preserves the resource as it exists at the time it was created. You can continue to modify the working draft and create new aliases (and consequently, versions) of your agent as necessary. In Amazon Bedrock, you create a new version of your agent by creating an alias that points to the new version by default. Amazon Bedrock creates versions in numerical order, starting from 1.

Versions are immutable because they act as a snapshot of your agent at the time you created it. To make updates to an agent in production, you must create a new version and set up your application to make calls to the alias that points to that version.

With aliases, you can switch efficiently between different versions of your agent without requiring the application to keep track of the version. For example, you can change an alias to point to a previous version of your agent if there are changes that you need to revert quickly.

To deploy your agent

1. Create an alias and version of your agent. Select the tab corresponding to your method of choice and follow the steps.

   Console

   To create an alias (and optionally a new version)

   1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
   2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
   3. In the **Aliases** section, choose **Create**.
   4. Enter a unique name for the alias and provide an optional description.
   5. Choose one of the following options:
      - To create a new version, choose **Create a new version and to associate it to this alias**.
      - To use an existing version, choose **Use an existing version to associate this alias**. From the dropdown menu, choose the version that you want to associate the alias to.
   6. Select **Create alias**. A success banner appears at the top.

   API

   To create an alias for an agent, send a CreateAgentAlias request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. To create a new version and associate this alias with it, leave the `routingConfiguration` object unspecified.

See code examples

2. Deploy your agent by setting up your application to make an InvokeAgent request (see link for request and response formats and field details) with an Agents for Amazon Bedrock runtime endpoint. In the `agentAliasId` field, specify the ID of the alias pointing to the version of the agent that you want to use.

To learn how to manage versions and aliases of agents, select from the following topics.

Topics

- Manage versions of agents in Amazon Bedrock
- Manage aliases of agents in Amazon Bedrock

Manage versions of agents in Amazon Bedrock

After you create a version of your agent, you can view information about it or delete it. You can only create a new version of an agent by creating a new alias.

Topics

- View information about versions of agents in Amazon Bedrock
- Delete a version of an agent in Amazon Bedrock

View information about versions of agents in Amazon Bedrock

To learn how to view information about the versions of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To view information about a version of an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose the version to view from the **Versions** section.

4. To view details about the model, action groups, or knowledge bases attached to version of the agent, choose the name of the information that you want to view. You can't modify any part of a version. To make modifications to the agent, use the working draft and create a new version.

API

To get information about an agent version, send a GetAgentVersion request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentVersion`.

To list information about an agent's versions, send a ListAgentVersions request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`. You can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

Delete a version of an agent in Amazon Bedrock

To learn how to delete a version of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To delete a version of an agent

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.

3. To choose the version for deletion, in the **Versions** section, choose the option button next to the version that you want to delete.
4. Choose **Delete**.
5. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the version, enter **`delete`** in the input field and choose **Delete**.
6. A banner appears to inform you that the version is being deleted. When deletion is complete, a success banner appears.

API

To delete a version of an agent, send a DeleteAgentVersion request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

Manage aliases of agents in Amazon Bedrock

After you create an alias of your agent, you can view information about it, edit it, or delete it.

Topics

- View information about aliases of agents in Amazon Bedrock
- Edit an alias of an agent in Amazon Bedrock
- Delete an alias of an agent in Amazon Bedrock

View information about aliases of agents in Amazon Bedrock

To learn how to view information about the aliases of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To view the details of an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.

2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose the alias to view from the **Aliases** section.
4. You can view the name and description of the alias and tags that are associated with the alias.

API

To get information about an agent alias, send a GetAgentAlias request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. Specify the `agentId` and `agentAliasId`.

To list information about an agent's aliases, send a ListAgentVersions request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and specify the `agentId`. You can specify the following optional parameters:

Field	Short description
maxResults	The maximum number of results to return in a response.
nextToken	If there are more results than the number you specified in the `maxResults` field, the response returns a `nextToken` value. To see the next batch of results, send the `nextToken` value in another request.

To view all the tags for an alias, send a ListTagsForResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the alias.

Edit an alias of an agent in Amazon Bedrock

To learn how to edit an alias of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To edit an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. In the **Aliases** section, choose the option button next to the alias that you want to edit.
4. You can edit the name and description of the alias. Additionally, you can perform one of the following actions:
   - To create a new version and associate this alias with that version, choose **Create a new version and associate it to this alias**.
   - To associate this alias with a different existing version, choose **Use an existing version and associate this alias**.

To add or remove tags associated with an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. Choose the alias for which you want to manage tags from the **Aliases** section.
4. In the **Tags** section, choose **Manage tags**.
5. To add a tag, choose **Add new tag**. Then enter a **Key** and optionally enter a **Value**. To remove a tag, choose **Remove**. For more information, see Tag resources.
6. When you're done editing tags, choose **Submit**.

API

To edit an agent alias, send an UpdateAgentAlias request. Because all fields will be overwritten, include both fields that you want to update as well as fields that you want to keep the same.

To add tags to an alias, send a TagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the alias. The request body contains a `tags` field, which is an object containing a key-value pair that you specify for each tag.

To remove tags from an alias, send an UntagResource request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint and include the Amazon Resource Name (ARN) of the alias. The `tagKeys` request parameter is a list containing the keys for the tags that you want to remove.

Delete an alias of an agent in Amazon Bedrock

To learn how to delete an alias of an agent, select the tab corresponding to your method of choice and follow the steps.

Console

To delete an alias

1. Sign in to the AWS Management Console, and open the Amazon Bedrock console at https://console.aws.amazon.com/bedrock/.
2. Select **Agents** from the left navigation pane. Then, choose an agent in the **Agents** section.
3. To choose the alias for deletion, in the **Aliases** section, choose the option button next to the alias that you want to delete.
4. Choose **Delete**.
5. A dialog box appears warning you about the consequences of deletion. To confirm that you want to delete the alias, enter **`delete`** in the input field and choose **Delete**.
6. A banner appears to inform you that the alias is being deleted. When deletion is complete, a success banner appears.

API

To delete an alias of an agent, send a DeleteAgentAlias request (see link for request and response formats and field details) with an Agents for Amazon Bedrock build-time endpoint. By default, the `skipResourceInUseCheck` parameter is `false` and deletion is stopped if the resource is in use. If you set `skipResourceInUseCheck` to `true`, the resource will be deleted even if the resource is in use.

See code examples