AgentSLM: Distilling Agentic Capability with Tool Reasoning

End-to-End Synthetic Dialog Generation, Fine-Tuning & Evaluation Pipeline — Customer Deployment Guide

AWS Marketplace | Container Product | Amazon MWAA | ECS Fargate | Amazon SageMaker | Amazon Bedrock Agents

Overview

AgentSLM automates the generation of comprehensive, high-accuracy conversational AI test scenarios directly from SOPs — and takes them all the way through fine-tuning, automated evaluation, and deployment as a production-ready inference endpoint.

The pipeline operates in four phases:

1. **Synthetic Data Generation** — Upload a SOP or playbook file and set of tools to S3. The pipeline extracts structure, fetches real user profiles from your RDS database, constructs synthetic user personas, and orchestrates multi-turn conversations between those simulated users and your existing **Amazon Bedrock Agent** (which uses a Lambda-backed custom action group for tool calling). The resulting dialog dataset is stored back to S3.
2. **SageMaker Fine-Tuning** — The generated conversations are used to launch a **Amazon SageMaker training job** that fine-tunes a base language model using LoRA/QLoRA. Zipped model weights are saved to `artifacts/<run_id>/training/` in S3 and the full unzipped weights are uploaded to `deployments/`.
3. **Automated Evaluation** — The fine-tuned model is evaluated by replaying conversation scenarios against the same Bedrock Agent. Each assistant turn is scored across five dimensions by a judge LLM. Conversations with scores are stored in `artifacts/<run_id>/inference/`.

4. **Bedrock Deployment** — The fine-tuned model is imported into **Amazon Bedrock** via the *Imported Models* feature and exposed as a serverless inference endpoint. An **Inference Lambda** acts as the production agent: it accepts user queries, calls a **Tool Retriever Lambda** (FAISS semantic search) to identify relevant tools, and then routes actual tool invocations to your Bedrock Agent's Lambda action group.

What You Provide

- An existing **Amazon MWAA** environment (Airflow 2.x or later).
- An existing **S3 bucket** for SOP uploads, tools(in openAPI schema) and pipeline outputs.
- An existing **RDS database** (PostgreSQL or MySQL) with a pre-created **Secrets Manager secret** holding the credentials — the pipeline fetches user records from this database to build synthetic personas.
- A **VPC** with at least two private subnets that have NAT Gateway access — the same VPC as your MWAA environment.
- A **security group** for ECS Fargate tasks with appropriate inbound and outbound rules configured for your VPC.
- An existing **Amazon Bedrock Agent** with at least one **Lambda-backed custom action group** that exposes your business tools. The pipeline uses this agent both for data generation and evaluation.
- A `tools.json` file (uploaded to `input/` in your S3 bucket) describing the tools available via your Lambda action group — used by the Tool Retriever at inference time.
- A `playbook.txt` file (uploaded to `input/sop_documents/` in your S3 bucket) containing your Standard Operating Procedure — this file triggers the pipeline and is used to generate synthetic training conversations.

Prerequisites

AWS Account Requirements

- Permissions to deploy CloudFormation stacks, create IAM roles, ECS services, and MWAA environments.

- An existing **S3 bucket** with **Versioning Enabled**. This bucket will hold your input documents, DAG code, and pipeline artifacts.
- An existing **RDS** instance reachable from your VPC (PostgreSQL or MySQL).
- A pre-created **Secrets Manager secret** containing your RDS credentials — see Step 3 below.
- At least **two private subnets** with NAT Gateway routes in different Availability Zones for the ECS and MWAA instances.
- **Amazon Bedrock model access** enabled for Anthropic Claude Sonnet in your AWS account and region.
- **Amazon SageMaker** service limits sufficient for a GPU-backed training job (e.g., `ml.g5.2xlarge` or `ml.p3.2xlarge`). Request a quota increase if needed before deployment.

Bedrock Agent with Lambda Action Group

The pipeline interacts with your business tools through an **Amazon Bedrock Agent** that has a **Lambda-backed custom action group**. This agent is used in two phases:

- **Data Generation** — Synthetic users send queries to the agent, which calls your Lambda action group to execute real tool logic. The resulting conversations form the training dataset.
- **Evaluation** — The lambda action group is used while replaying scenarios against the fine-tuned model to score response quality.

Before deploying AgentSLM, ensure you have:

- A deployed **Bedrock Agent** with a published alias — note the **Agent ID** and **Agent Alias ID**.
- A **Lambda function** registered as a custom action group on that agent. The Lambda must implement the Bedrock Agent action group invocation contract.
- The **Action Group name** (e.g., `my_tools_action_group`) — required as a CloudFormation parameter.
- A `tools.json` file describing each tool's name, description, and parameter schema — uploaded to `s3://<YOUR-BUCKET>/input/tools.json`.
- A pre-built **FAISS index** (`index.faiss`) and companion `metadata.json` — uploaded to `s3://<YOUR-BUCKET>/input/`. These files power the Tool Retriever semantic search. See Inference Endpoint & Tool Retriever for details on how to build them.

ℹ️ The Tool Retriever uses **Amazon Titan Text Embeddings V2** to embed tool descriptions and performs a FAISS cosine-similarity search at inference time. Enable Titan Embeddings V2 access in the Bedrock console before deploying.

Security Group for ECS

Create (or identify) a security group for the ECS Fargate tasks. Ensure it allows inbound traffic from your MWAA worker security group and outbound HTTPS access for AWS service calls. Contact your network administrator if you need assistance configuring security group rules.

Local Tools

- AWS CLI v2 with credentials configured, or access to the AWS Management Console.

SOP Document Structure

The pipeline processes Standard Operating Procedure (SOP) documents to extract business logic, rules, variables, and tool structures for test scenario generation. To ensure high parsing accuracy by the pipeline's LLM engine, the uploaded SOP files must be saved as UTF-8 encoded plain text (`.txt` or `.md`) files and adhere to a standard corporate SOP structure.

A standard corporate SOP document is composed of the following sections:

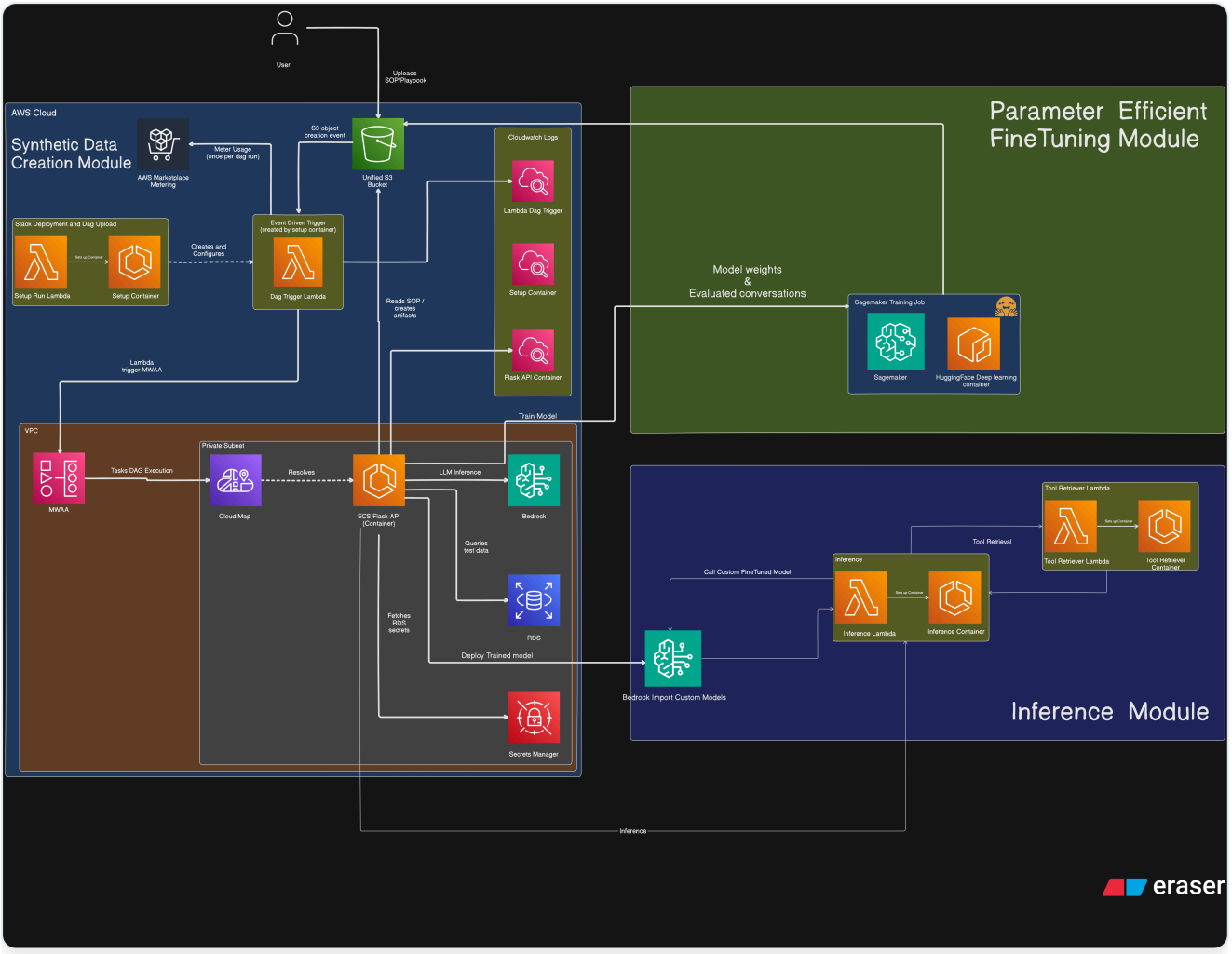
1. **Document Control:** A metadata block containing identifiers, version information, effective dates, ownership details, and department association.
2. **Purpose and Scope:** A high-level description of what the procedure achieves and who/what is within the scope of execution.
3. **Roles and Responsibilities:** Defines the entities involved (e.g. customer, agent/automated bot, system APIs) and their roles.
4. **Procedure:** Step-by-step description of the sequential workflow steps, validations, data collection, and tool usage.
5. **Exception Handling:** Explicit procedures for failed validation, error scenarios, or API integration errors.
6. **Technical Specifications:** Standard OpenAPI 3.0 YAML schemas defining any backend tools or APIs that are referenced and called during the workflow.

SOP Format Template

Below is the standard corporate template for the SOP document. Replace the placeholder sections (in brackets) with your actual process flow and API definitions. Emojis and conversational labels must be excluded from structural headers to maintain formatting cleanliness.

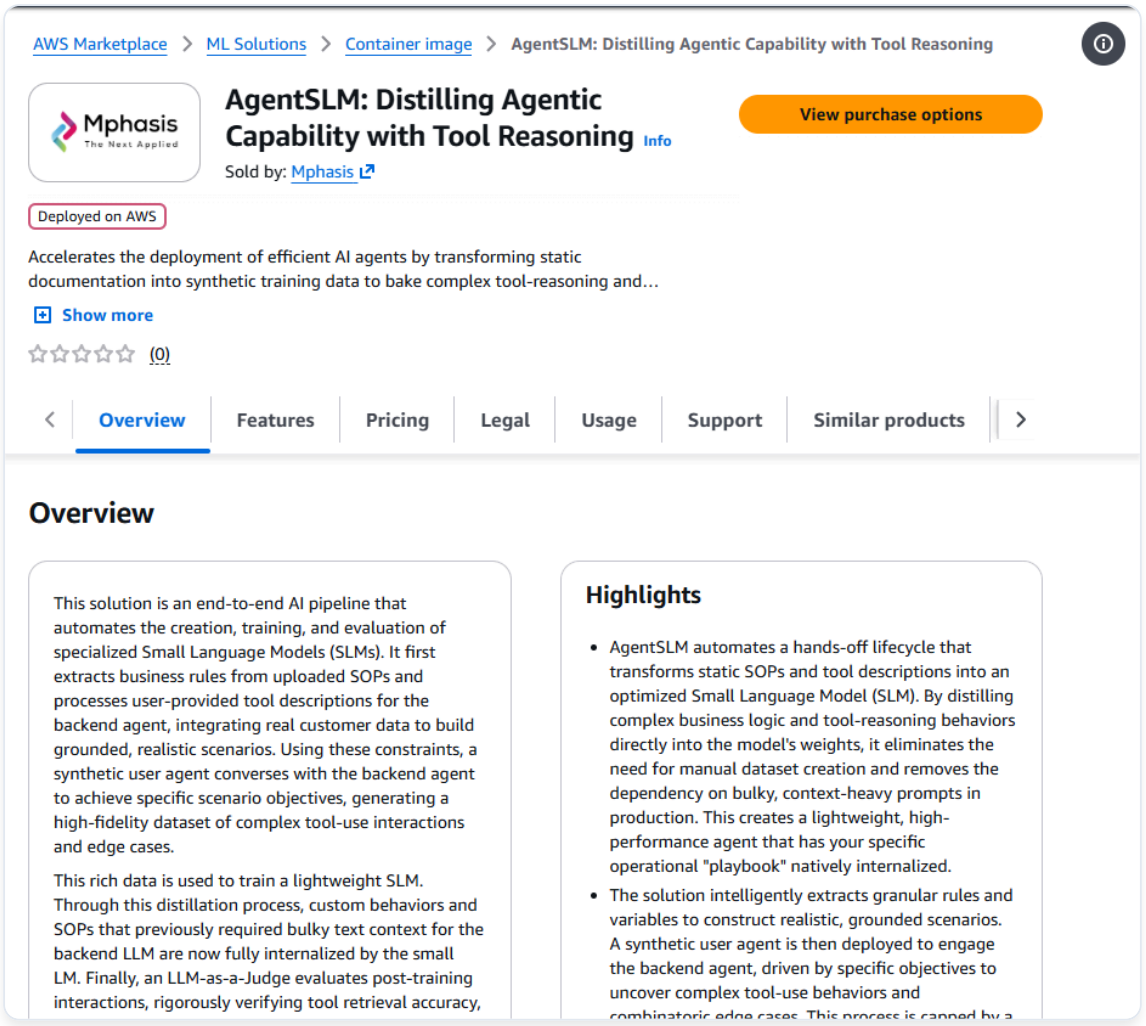
```
SOP-[ID]: [SOP Title]

1. Document Control
- SOP ID: [SOP-XXX-###]
- Title: [Process Title]
- Version: [Version Number]
- Department: [Department Name]
- Effective Date: [YYYY-MM-DD]
- Owner: [Owner Role or Name]

2. Purpose and Scope
[Describe the purpose, objective, and scope of this procedure.]

3. Roles and Responsibilities
- [Role Name 1]: [Description of what this role is responsible for]
- [Role Name 2]: [Description of what this role is responsible for]

4. Procedure
4.1 [Sub-Process A Title]
- Step 1: [Action description, details of user information to collect, and
input rules]
- Step 2: [Action description, including calling tools/APIs]
- Step 3: [Subsequent process step description]

4.2 [Sub-Process B Title]
- Step 1: [Prerequisite checks and action description]
- Step 2: [Action description, including calling tools/APIs]

5. Exception Handling
- [Scenario 1 (e.g. Validation Error)]: [Describe error path, user
notifications, and fallback action]
- [Scenario 2 (e.g. API/Tool Failure)]: [Describe error path, user
notifications, and fallback action]

6. Technical Specifications (OpenAPI Schema)
[Standard OpenAPI 3.0 YAML/JSON definitions for APIs referenced in the
procedure.(The same which you have in the tools.json file; paste it as a
YAML/JSON block here.)]
```

Architecture

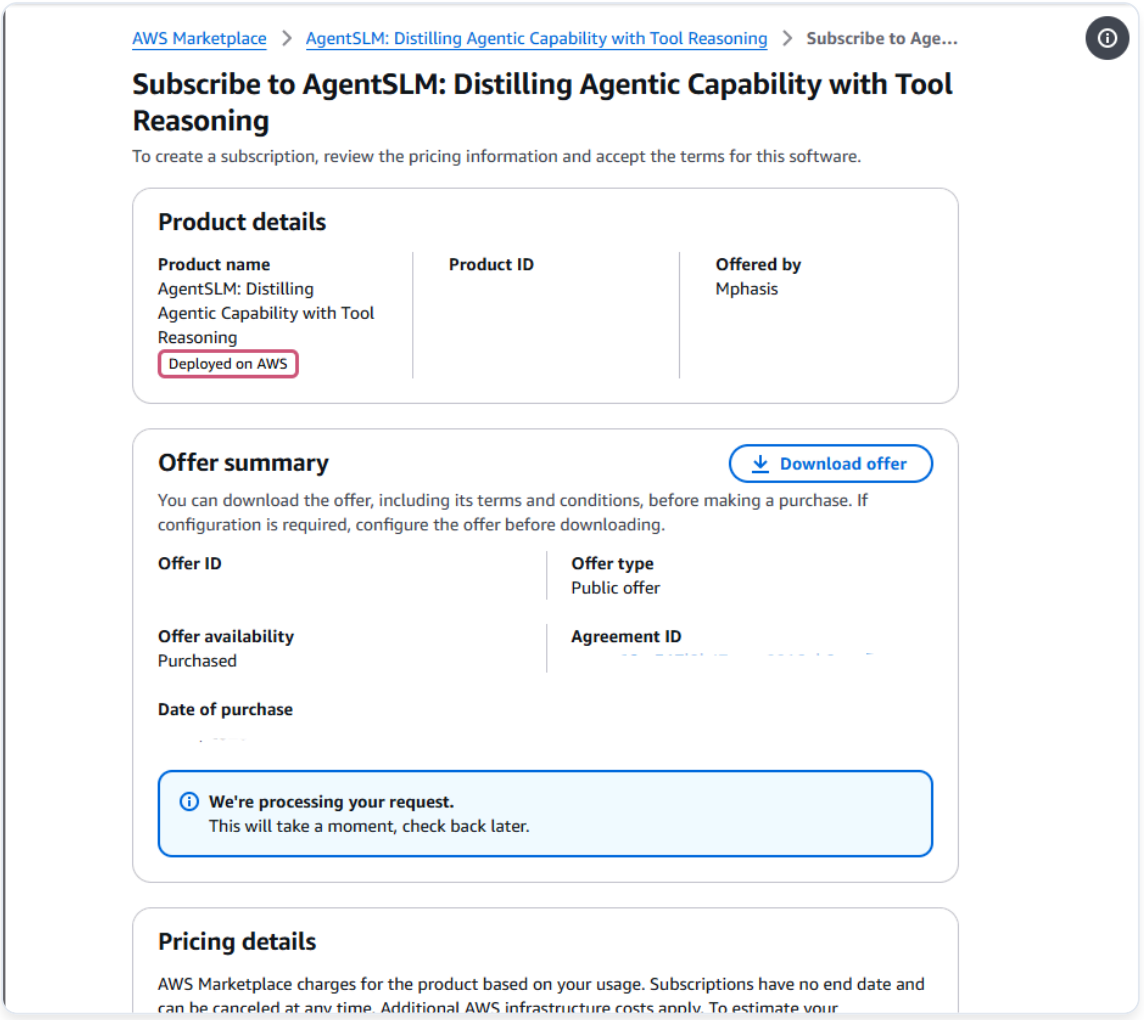


Step 1 — Subscribe on AWS Marketplace

1 **Find the product** on AWS Marketplace and click *Continue to Subscribe*.

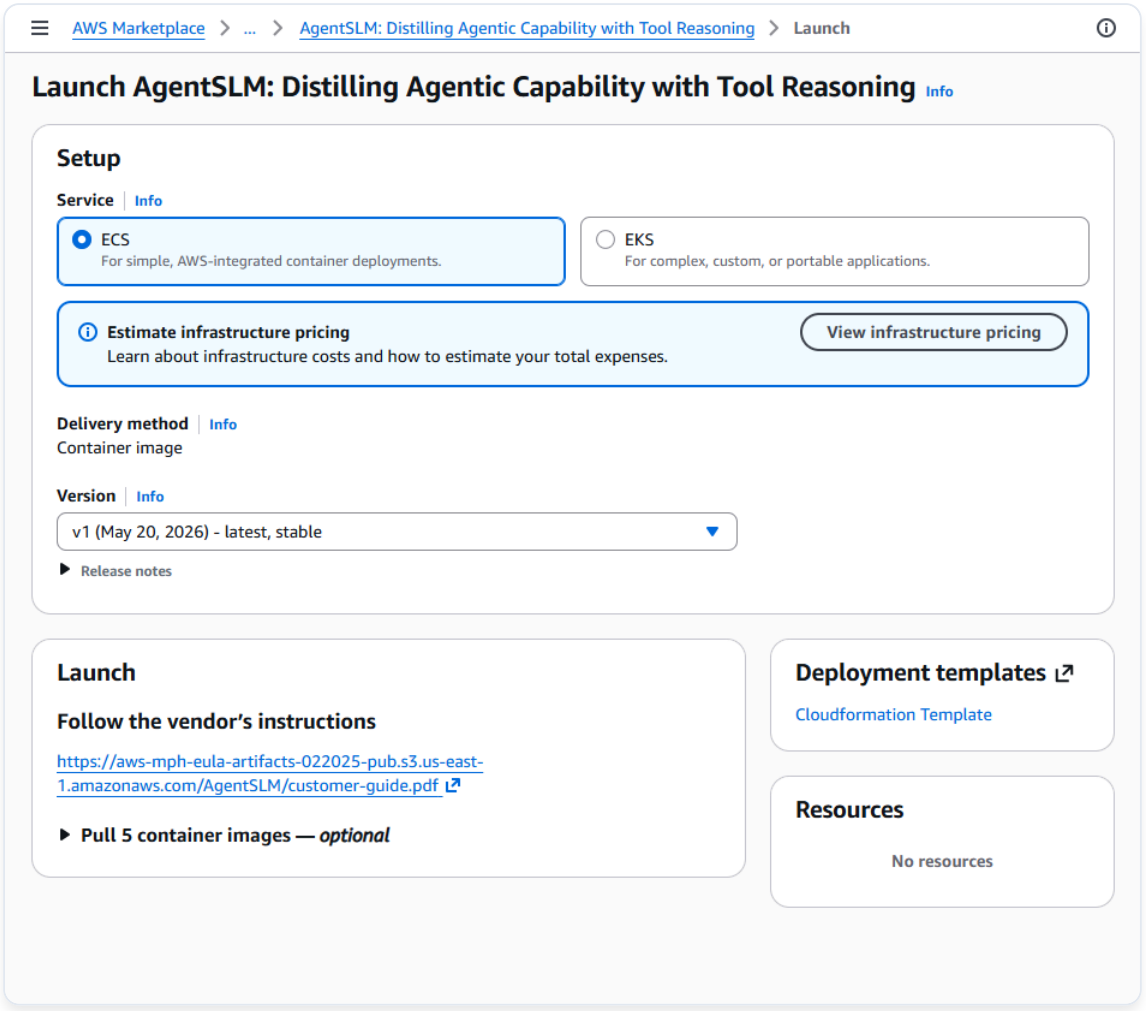


2 Accept the End User License Agreement and click *Continue to Configuration*.



3 Note the **Container Image URI** shown on the configuration page. You will enter this as the `ContainerImageUri` CloudFormation parameter in Step 4.

4 Click *Continue to Launch* → *Launch CloudFormation* to open the CloudFormation console with the template pre-loaded, or deploy manually using `AgentSLM_cloudformation_template.yml`.



Step 2 — Set Up Amazon MWAA

AgentSLM uses **Amazon Managed Workflows for Apache Airflow (MWAA)** to orchestrate the pipeline. You must have a running MWAA environment **before** deploying the CloudFormation stack. If you already have one, skip to Step 3 and note your environment name.

ℹ️ Creating a new MWAA environment takes **20–30 minutes**. Plan accordingly before you proceed to the CloudFormation deployment.

Create an MWAA Environment (Console)

1. Open the **Amazon MWAA** console and click *Create environment*.

2. **Name** — give it a name (e.g. `AgentSLM-mwaa`). You will enter this exact name as the `MWAAEnvName` CloudFormation parameter later.

3. **Airflow version** — select **2.9** or later.

4. **S3 bucket for DAG storage** — choose an existing S3 bucket (can be the same bucket you use for SOP documents) and set the DAG folder to `dags/`. You will upload `AgentSLM.py` to this folder in a later step.

5. **Networking**:
   - Select the **same VPC** you will use for the ECS Fargate service.
   - Select at least **two private subnets** in different Availability Zones.
   - Choose or create a **security group** for MWAA workers with appropriate outbound rules to reach ECS tasks.

6. **Environment class** — `mw1.small` is sufficient for most workloads. Scale up if you process many large SOP documents concurrently.

7. **Requirements file** — create a `requirements.txt` file with the required libraries and upload it to your MWAA S3 bucket. Point the *Requirements file* field to it (e.g. `s3://<YOUR-MWAA-BUCKET>/requirements.txt`).

8 Leave all other settings at their defaults and click *Create environment*. Wait for the status to change to **Available**.

Find Your Environment Name

Once the environment is **Available**, note the exact environment name from the MWAA console — it is case-sensitive. Enter this as the `MWAAEnvName` CloudFormation parameter.

Step 3 — Prepare Your Secrets Manager Secrets

The pipeline requires **two** Secrets Manager secrets to be created **before** deploying the CloudFormation stack.

Secret 1 — RDS Credentials (`DBPasswordSecretArn`)

The ECS container reads your RDS credentials at runtime from this secret. Provide its ARN as the `DBPasswordSecretArn` CloudFormation parameter.

🔒 The secret must use **exactly** these JSON key names — the application will fail to connect to the database if any key is missing or mis-spelled.

Required JSON Structure

```
{
  "DB_TYPE":     "postgresql",   <!-- or "mysql" -->
  "DB_USER":     "your_username",
  "DB_PASSWORD": "your_password",
  "DB_HOST":     "your-db.cluster-xxxx.us-west-2.rds.amazonaws.com",
  "DB_PORT":     "5432",         <!-- use "3306" for MySQL -->
  "DB_NAME":     "your_database_name"
}
```

Create the Secret (AWS CLI)

```
aws secretsmanager create-secret \
  --name "AgentSLM/rds-credentials" \
  --secret-string '{
    "DB_TYPE":     "postgresql",
    "DB_USER":     "dbadmin",
    "DB_PASSWORD": "<YOUR-PASSWORD>",
    "DB_HOST":     "my-db.cluster-xxxx.us-west-2.rds.amazonaws.com",
    "DB_PORT":     "5432",
    "DB_NAME":     "AgentSLM"
  }' \
  --region us-west-2
```

Copy the returned `ARN` — you will enter it as the `DBPasswordSecretArn` CloudFormation parameter.

Secret 2 — Agent Configuration (`AgentSecret`)

The pipeline also reads Bedrock Agent configuration and inference settings from a second secret named `AgentSecret`. You must create this secret before deployment and provide its ARN as the `AgentSecretArn` CloudFormation parameter.

> 🔒 Use **exactly** these JSON key names. The `TRAINED_MODEL_ARN` key will be **automatically populated** by the pipeline after the Bedrock model import completes — you may leave it empty at creation time.

Required JSON Structure

```
{
  "AGENT_ID":            "your-bedrock-agent-id",
  "AGENT_ALIAS_ID":      "your-bedrock-agent-alias-id",
  "LAMBDA_FUNCTION_NAME": "your-action-group-lambda-name",
  "LAMBDA_ACTION_GROUP": "your-action-group-name",
  "TRAINED_MODEL_ARN":   ""   <!-- populated automatically after model
deployment -->
}
```

Create the Secret (AWS CLI)

```
aws secretsmanager create-secret \
  --name "AgentSecret" \
  --secret-string '{
    "AGENT_ID":              "<YOUR-AGENT-ID>",
    "AGENT_ALIAS_ID":        "<YOUR-ALIAS-ID>",
    "LAMBDA_FUNCTION_NAME": "<YOUR-LAMBDA-NAME>",
    "LAMBDA_ACTION_GROUP":   "<YOUR-ACTION-GROUP-NAME>",
    "TRAINED_MODEL_ARN":     ""
  }' \
  --region us-west-2
```

Copy the returned `ARN` — you will enter it as the `AgentSecretArn` CloudFormation parameter.

Step 4 — Configure Your Bedrock Agent & Upload Tool Index

AgentSLM drives conversations through your own **Amazon Bedrock Agent**. You must grant the pipeline's ECS task role permission to invoke that agent, and upload the tool index files that power the semantic Tool Retriever at inference time.

4a — Collect Agent Identifiers

1. Open the **Amazon Bedrock** console and navigate to *Agents*.
2. Select your agent and note the **Agent ID** (e.g., `ABCDE12345`) and the **Agent Alias ID** of the published alias you want the pipeline to use (e.g., `TSTALIASID`).
3. Note the **Action Group name** attached to the agent — this is the action group backed by your Lambda function.
4. Note the **Lambda function name or ARN** of that action group's Lambda. The pipeline's ECS task role needs `lambda:InvokeFunction` on this function.

4b — Grant the Pipeline Access to Your Agent

The ECS task role created by the CloudFormation stack must be allowed to call your Bedrock Agent and its backing Lambda. After the stack is deployed (Step 5), attach the following inline policy to the ECS task role (`AgentSLMTaskRole` or similar — check the CloudFormation Outputs tab):

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["bedrock:InvokeAgent"],
      "Resource": "arn:aws:bedrock:<REGION>:<ACCOUNT_ID>:agent-
alias/<AGENT_ID>/<ALIAS_ID>"
    },
    {
      "Effect": "Allow",
      "Action": ["lambda:InvokeFunction"],
      "Resource": "arn:aws:lambda:<REGION>:<ACCOUNT_ID>:function:
<ACTION_GROUP_LAMBDA_NAME>"
    }
  ]
}
```

4c — Upload `tools.json`

1. **Create** `tools.json` — a JSON array where each element describes one tool available in your Lambda action group. Each entry needs at minimum a `name`, a `description` (used for embedding), and a `parameters` schema.

```
[
  {
    "name": "list_transactions",
    "description": "Lists recent transactions for a given
account.",
    "parameters": {
      "account_id": { "type": "string", "description": "The
account identifier." },
      "limit":      { "type": "integer", "description": "Max
number of results." }
    }
  }
]
```

Step 5 — Deploy the CloudFormation Stack

Option A: AWS Console

1. Open the **CloudFormation** console and click *Create stack → With new resources*.

2. Upload `AgentSLM_cloudformation_template.yml` or paste the Marketplace-provided template URL.

3. Enter a unique **Stack name** (e.g. `AgentSLM-prod`).

4. Fill in all parameters (see the Parameters Reference section below).

5. On the *Capabilities* page, check *I acknowledge that AWS CloudFormation might create IAM resources with custom names*.

6. Click *Submit* and wait for status `CREATE_COMPLETE` (~5–8 minutes).

CloudFormation Parameters Reference

Container Image

Parameter	Required	Description
`ContainerImageUri`	Yes	ECR image URI provided by AWS Marketplace after subscribing. Format: `<account>.dkr.ecr.<region>.amazonaws.com/<repo>:<tag>`
`SetupImageUri`	Yes	ECR image URI for the one-shot setup container. Format: `<account>.dkr.ecr.<region>.amazonaws.com/<setup-repo>:<tag>`

Networking

Parameter	Required	Description
`VpcId`	Yes	VPC ID. Must be the same VPC as your MWAA environment so Cloud Map DNS resolution works.
`ECSSubnetIds`	Yes	Comma-separated list of ≥ 2 **private** subnets in different AZs for ECS Fargate tasks. Must have NAT Gateway routes for outbound internet access (Bedrock, ECR, Secrets Manager).
`ECSSecurityGroupId`	Yes	Security group for ECS Fargate tasks with appropriate inbound and outbound rules. See Prerequisites.
`MWAASecurityGroupId`	Yes	Security group ID attached to your MWAA environment workers. Used to automatically add an inbound rule to the ECS security group.

Storage – S3

Parameter	Required	Description
`S3BucketName`	Yes	Name of your existing S3 bucket. Used for SOP document input and pipeline artifact/output storage.

Database – RDS Credentials

Parameter	Required	Description
`DBPasswordSecretArn`	Yes	ARN of a pre-existing Secrets Manager secret containing your RDS credentials. The secret **must** contain these exact JSON keys: `DB_TYPE`, `DB_USER`, `DB_PASSWORD`, `DB_HOST`, `DB_PORT`, `DB_NAME`. See Step 2 for how to create it.

Workflow – Amazon MWAA

Parameter	Default	Required	Description
`MWAAEnvName`	—	Yes	Name of your MWAA environment (not the ARN). Case-sensitive. Found in the Amazon MWAA console under *Environments*.

Bedrock Agent

Parameter	Required	Description
`BedrockAgentId`	Yes	ID of your Amazon Bedrock Agent (e.g., `ABCDE12345`). The pipeline uses this agent for synthetic data generation and evaluation.
`BedrockAgentAliasId`	Yes	Published alias ID of the Bedrock Agent (e.g., `TSTALIASID`). Use a stable alias — not the `DRAFT` alias — for reproducibility.
`AgentActionGroupName`	Yes	The name of the Lambda-backed custom action group on your Bedrock Agent. Used by the inference container to route actual tool calls.
`ActionGroupLambdaName`	Yes	Name (or ARN) of the Lambda function that implements the Bedrock Agent action group. The pipeline invokes this function directly for tool execution.

Tool Retriever & Inference

Parameter	Required	Description
`ToolRetrieverLambdaName`	Yes	Name of the Tool Retriever Lambda function (deployed from `tool_retriever_container/`). The inference Lambda calls this to perform FAISS semantic search over your tool index before every tool call.
`MarketplaceProductCode`	No	AWS Marketplace product code used for usage metering. Leave blank for internal testing or BYOL scenarios.

Step 6 — Verify DAG Availability

The CloudFormation stack automatically uploads the `AgentSLM.py` DAG and `requirements.txt` to your S3 bucket during deployment.

1. Wait ~20-30 minutes for the MWAA environment to finish creating and for the Airflow scheduler to pick up the new files.
2. Open the **Airflow UI** (MWAA console → your environment → *Open Airflow UI*).
3. Confirm that `sop_upload_synth` appears in the DAGs list.
4. **Enable the DAG:** Click the toggle switch next to the DAG name to turn it **On**.

Step 7 — (Optional) Set DAG Display Name in MWAA

You can set a cosmetic display name for the DAG shown in the Airflow UI by adding an MWAA environment variable.

How to Set It (Optional)

1. Open the **Amazon MWAA** console and select your environment.
2. Click **Edit**.
3. Under *Airflow configuration options*, add:

```
Key:   env.DAG_DISPLAY_NAME
Value: SOP Upload Synthetic Data
```

4. Click **Save** and wait for the environment update.

Step 8 — Verify the Pipeline

1. **Verify the stack deployed successfully** — open the CloudFormation console, select your stack, and confirm the status shows `CREATE_COMPLETE`.
2. **Trigger the pipeline** by uploading a SOP document to your S3 bucket under the `input/sop_documents/` prefix and tools.json into the `input/` prefix. The DAG is configured to automatically trigger on new SOP document uploads.

3. **Monitor the Airflow DAG run** in the Airflow UI. Open the MWAA console → your environment → *Open Airflow UI* and check the DAG runs list. Tasks will progress through *queued* → *running* → *success*. The full pipeline takes approximately 15–40 minutes depending on document size.

4. **Check S3 for output artifacts** — the finished QA dataset and intermediate files are written back to your S3 bucket under the `artifacts/` prefix.

Pipeline Phase: SageMaker Fine-Tuning

After synthetic conversations are generated and stored in S3, the Airflow DAG automatically submits a **SageMaker training job** to fine-tune a **Qwen3-4B Instruction tuned** model on the generated dataset. The training uses **LoRA / QLoRA** (via PEFT + TRL) for parameter-efficient fine-tuning with 4-bit quantization.

What Happens

1. **Dataset preparation** — Generated conversation JSON files are assembled into a tokenized JSONL dataset. Each record ends at an assistant turn so that the model learns to produce reasoning chains and tool calls in the correct format.

2. **SageMaker job launch** — The DAG submits a training job using the `training_container/` Docker image. The job reads the dataset from S3, fine-tunes the model, and writes adapter checkpoints to S3.

3 **Artifact storage** — Zipped model weights (LoRA adapters merged with the base model) are saved to:

```
s3://<YOUR-BUCKET>/artifacts/<run_id>/training/
```

Full unzipped weights (ready for Bedrock import) are uploaded to:

```
s3://<YOUR-BUCKET>/deployments/<run_id>/
```

ℹ️ Monitor job progress in the **SageMaker console** → **Training** → **Training jobs**. Logs are available in the associated CloudWatch log group. Training time depends on dataset size and instance type — typically 30–90 minutes on a `ml.g5.2xlarge`.

Pipeline Phase: Automated Evaluation

Once training is complete, the pipeline runs a **batch inference evaluation**: it replays representative scenarios through the fine-tuned model, routing each tool call through your Bedrock Agent's Lambda action group, and then scores every assistant turn using a judge LLM.

Evaluation Dimensions

Each assistant turn in every conversation is evaluated on five dimensions:

Dimension	Scale	What is Measured
`appropriateness_of_tool_usage`	0 or 1	Did the model correctly decide when to call a tool vs. respond directly? Did it invoke the Tool Retriever before any tool call, and call only tools returned by the retriever?
`faithfulness_of_synthesized_response`	0–10	Did the model accurately use the tool result or correctly convey what the user needs to do? No hallucinations or omissions.
`adherence_to_playbook_steps`	0–10	Did the model follow the SOP step sequence, collect required parameters correctly, and respect playbook logic in its reasoning and response?
`conversational_quality`	0–10	Was the response clear, concise, contextually aware, and goal-aligned without contradicting prior conversation turns?
`follow_up_and_memory`	0–10	Did the model ask for clarification when needed, avoid hallucinating parameter values the user never provided, and reuse previously collected information without redundant questions?

Evaluation Output

Conversations annotated with per-turn scores are stored at:

```
s3://<YOUR-BUCKET>/artifacts/<run_id>/inference/
```

Each file is a JSON array of conversation turns, each containing the assistant's reasoning, its final response, the tool calls made, and the five evaluation scores.

Pipeline Phase: Model Deployment to Amazon Bedrock

After evaluation, the pipeline **automatically** deploys the fine-tuned model to **Amazon Bedrock** using the *Imported Models* feature — no manual console steps are required. The DAG submits the import job directly from the model weights stored in S3 and monitors it until the model is active.

1. **Model weights in S3** — The full, unzipped model weights are written to `s3://<YOUR-BUCKET>/deployments/<run_id>/` at the end of the training phase. Bedrock reads them directly from this S3 path during import.

2. **Automatic Bedrock import** — The DAG programmatically calls the Bedrock `CreateModelImportJob` API, pointing to the `deployment/<run_id>/` S3 prefix. It polls until the import job reaches `Completed` status (typically 15–30 minutes).

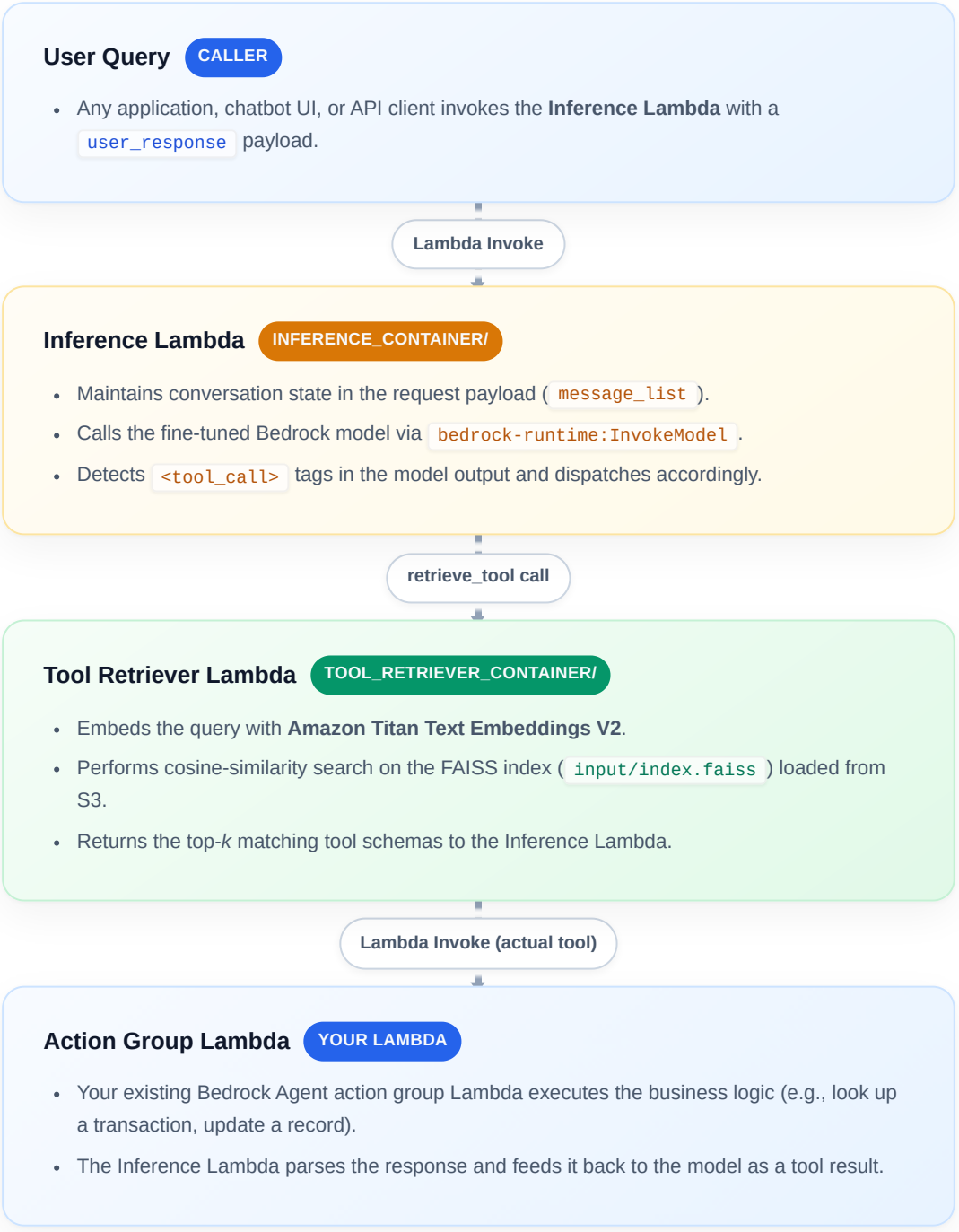
3. **ARN stored in Secrets Manager** — Once the import is complete, the pipeline automatically writes the resulting **Custom Model ARN** into the `AgentSecret` in AWS Secrets Manager. The Inference Lambda reads the ARN from this secret at runtime — no manual configuration is needed.

ℹ️ Imported models are billed per input/output token at standard Bedrock on-demand rates. No dedicated endpoint provisioning is required — Bedrock manages capacity automatically.

Inference Endpoint & Tool Retriever

The production inference layer consists of two Lambda functions that work in tandem to serve user queries through the fine-tuned model with dynamic tool selection.

How It Works



Configuring the Inference Lambda

Set the following environment variables on the Inference Lambda function (deployed from

`inference_container/`):

Environment Variable	Value
`CUSTOM_MODEL_ARN`	Full ARN of the imported Bedrock model — copied from the Bedrock console after import.
`TOOL_RETRIEVER_LAMBDA_NAME`	Name of the Tool Retriever Lambda function.
`LAMBDA_FUNCTION_NAME`	Name of your action group Lambda function.
`LAMBDA_ACTION_GROUP`	The action group name registered on your Bedrock Agent (e.g., `my_tools_action_group`).
`S3_BUCKET`	Your S3 bucket name — used to load `input/tools.json` at runtime.
`AWS_REGION`	AWS region where all resources are deployed.

⚠️ The Inference Lambda must have `lambda:InvokeFunction` permission on both the Tool Retriever Lambda and your Action Group Lambda, as well as `bedrock:InvokeModel` on the imported model ARN and `s3:GetObject` on your bucket's `input/` prefix.

S3 Artifact Structure

All pipeline inputs and outputs are organized under your S3 bucket as follows:

```
s3://<YOUR-BUCKET>/
├── input/
│   ├── sop_documents/          ← Upload SOP/playbook files here to trigger
the DAG
│   ├── tools.json              ← Tool schemas for the Tool Retriever
│
├── artifacts/
│   └── <run_id>/
│       ├── training/           ← Zipped LoRA adapter weights from
SageMaker
│       └── inference/          ← Evaluated conversations with per-turn
scores
│
└── deployments/            ← Full unzipped model weights (imported into
Bedrock)
```

ℹ️ Each pipeline run is identified by a unique `run_id` (typically a timestamp or UUID). You can find the `run_id` for a specific DAG run in the Airflow UI under the DAG run's configuration JSON.

Usage Billing & Metering

AgentSLM is an AWS Marketplace **Usage-Based** product. Billing is calculated based on the number of successful pipeline runs triggered.

ℹ️ Metering only occurs if the `MarketplaceProductCode` parameter is provided during CloudFormation deployment. If left blank, the pipeline will function normally but no marketplace usage will be recorded (useful for internal testing or BYOL scenarios).

Troubleshooting

ECS task keeps restarting / fails to start

- Check CloudWatch logs for the ECS task via the CloudWatch console.
- Confirm the ECS task role has `secretsmanager:GetSecretValue` on your `DBPasswordSecretArn`. An `AccessDeniedException` in logs means the role lacks this permission.
- Confirm the ECS subnets have NAT Gateway routes (ECR image pulls require outbound HTTPS 443).
- Verify `Amazon Bedrock model access` is enabled for Claude Sonnet in your account/region.

Airflow DAG tasks fail with "connection refused" or timeout

- Confirm MWAA and ECS are in the same VPC — Cloud Map DNS (`mwaa-internal.local`) only resolves within the same VPC.
- Verify the ECS security group allows inbound traffic from the MWAA worker security group.
- Confirm the ECS service has at least 1 running task (`runningCount: 1`).
- Check Cloud Map in the AWS console: **Service Discovery** → **Namespaces** → **mwaa-internal.local** → **flask-app** — the service instance should show a healthy IP.

Lambda is not triggered on S3 upload

- Confirm the setup container completed successfully during stack deployment (check the `AgentSLM-setup` log group).
- Confirm the object key starts with `input/sop_documents/` (prefix must include the trailing slash).
- Check that the S3 bucket notification exists: **S3 Console → your bucket → Properties → Event notifications**.
- Verify the Lambda resource policy allows `s3.amazonaws.com` from your account ID.

Lambda error: "Airflow returned HTTP 403"

- Confirm the Lambda execution role has `airflow:CreateCliToken` on your MWAA environment ARN.
- Verify `MWAA_ENV_NAME` (Lambda environment variable) matches exactly the name of your MWAA environment — it is case-sensitive.
- Ensure the MWAA environment is in `AVAILABLE` state.

DAG import error: "No module named 'requests'"

- The `requests` library must be available to MWAA workers. Add `requests` to your MWAA environment's `requirements.txt` file in the MWAA S3 bucket and update the MWAA environment to pick it up.

Secret read fails at runtime ("Invalid secret JSON")

- The Secrets Manager secret must contain all six keys exactly as named: `DB_TYPE`, `DB_USER`, `DB_PASSWORD`, `DB_HOST`, `DB_PORT`, `DB_NAME`. Missing or differently named keys will cause a runtime error in the container.
- Verify with: `aws secretsmanager get-secret-value --secret-id <YOUR-ARN> --query SecretString --output text`

Bedrock inference error ("ValidationException" or throttling)

- Ensure **Amazon Bedrock model access** is enabled for Anthropic Claude Sonnet 4.5 in your AWS account and region. Open the Bedrock console → *Model access* and enable the model.
- If using cross-region inference (the default), confirm the region you deployed into supports cross-region inference profiles.
- For throttling errors, the pipeline will retry automatically. For persistent quota issues, request a Bedrock service quota increase.

SageMaker training job fails or does not start

- Check the training job logs in **CloudWatch → Log groups → /aws/sagemaker/TrainingJobs**.
- Confirm your account has sufficient SageMaker GPU instance quota (e.g., `ml.g5.2xlarge`). Request a quota increase via **Service Quotas → Amazon SageMaker** if the job fails with a capacity error.
- Verify the SageMaker execution role has `s3:GetObject` and `s3:PutObject` on your S3 bucket's `artifacts/` and `deployments/` prefixes.
- Confirm the training container image URI is accessible — ensure the ECS task role (or SageMaker execution role) has `ecr:GetDownloadUrlForLayer` and related ECR pull permissions.

Bedrock model import fails ("InvalidRequestException")

- Confirm the model weights in `deployments/<run_id>/` are in a format supported by Bedrock Imported Models (e.g., Hugging Face `safetensors` format with a `config.json`).
- Verify the Bedrock service role used during import has `s3:GetObject` on the `deployments/` prefix and `s3:ListBucket` on the bucket.
- Imported Models is only available in select AWS regions — check the Bedrock documentation for current region availability.

Tool Retriever returns wrong tools or empty results

- Confirm `input/index.faiss` and `input/metadata.json` are present in your S3 bucket — run the setup script again if they are missing.
- Verify **Amazon Titan Text Embeddings V2** (`amazon.titan-embed-text-v2:0`) model access is enabled in the Bedrock console.
- If tools were recently added or changed, rebuild the FAISS index by re-running the setup script — the index must match the current `tools.json`.
- Check CloudWatch logs for the Tool Retriever Lambda for any `AccessDeniedException` errors from Bedrock or S3.

Inference Lambda: model returns no tool calls / conversation loops

- Confirm the `CUSTOM_MODEL_ARN` environment variable on the Inference Lambda is the correct ARN of the imported Bedrock model — not a foundation model ARN.
- Verify the Inference Lambda has `bedrock:InvokeModel` permission on the custom model ARN.
- Check that `input/tools.json` is present in S3 and readable by the Lambda's execution role (`s3:GetObject`).
- If the model enters a tool-call loop, the Inference Lambda caps iterations at 3 turns per user message. Review CloudWatch logs to identify which tool call is failing.

Support

For issues with this AWS Marketplace product, use the **Support** link on the Marketplace product page. When contacting support, provide:

- Your CloudFormation stack name and AWS region.
- The relevant CloudWatch log streams — ECS task logs, Lambda logs (Inference Lambda and Tool Retriever Lambda), and SageMaker training job logs.
- The error message and task ID from the Airflow UI.
- The `run_id` of the affected pipeline run (visible in the Airflow DAG run configuration).
- Your Bedrock Agent ID and alias ID, and the name of the Lambda action group.

> ✅ Before contacting support, check the Troubleshooting section above — most common issues are covered there.