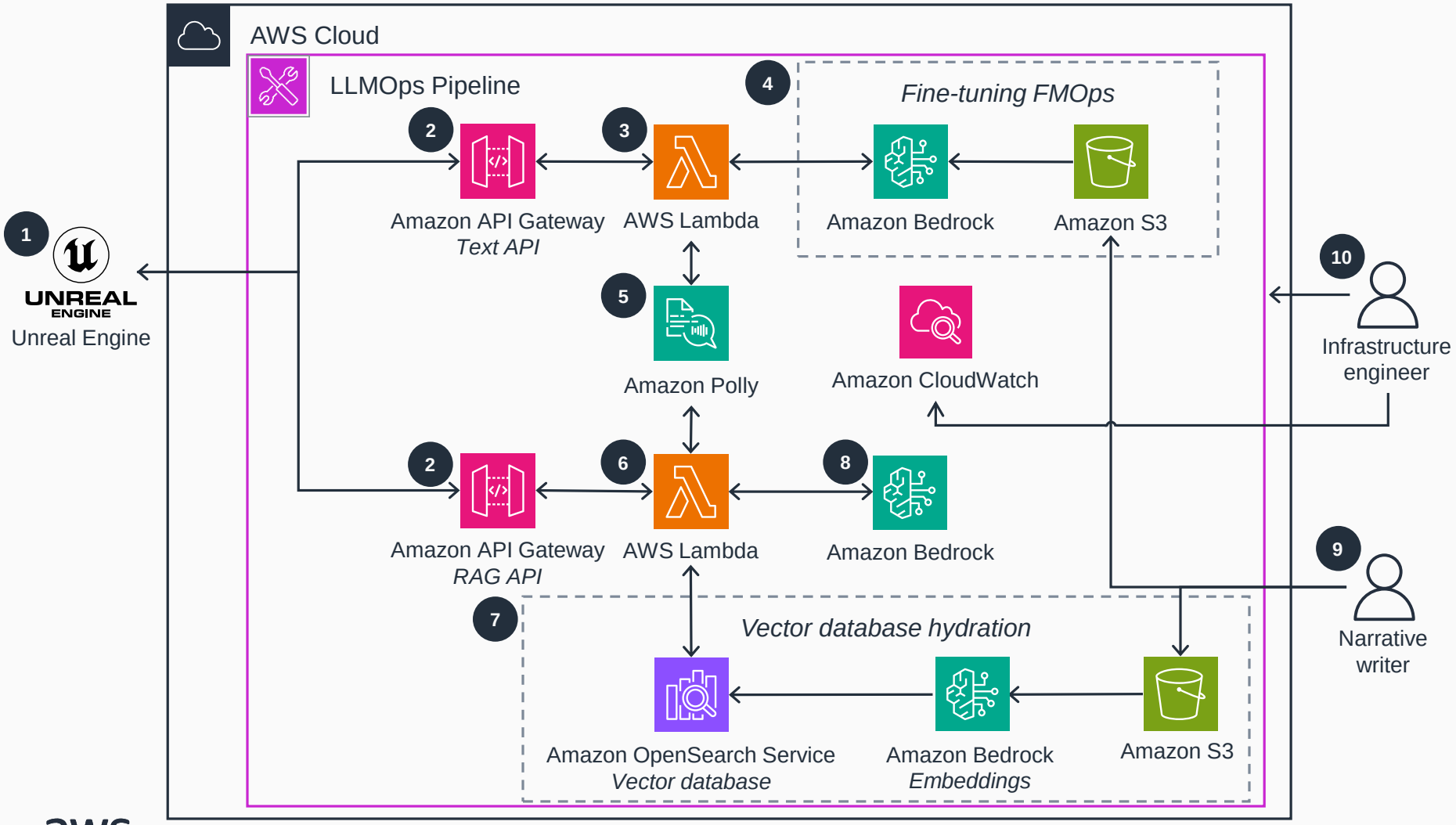
Guidance for Dynamic Non-Player Character (NPC) Dialogue on AWS

Overview

This architecture diagram shows an overview workflow for hosting a generative AI NPC on AWS.



1. Game clients interact with the NPC running on Unreal Engine.
2. Requests for generated text responses from the NPC are sent to a Text API **Amazon API Gateway** endpoint. Requests that require game-specific context from the NPC are sent to a retrieval-augmented generation (RAG) **API Gateway** endpoint.
3. **AWS Lambda** handles the NPC text requests and sends them to LLMs hosted on **Amazon Bedrock**.
4. Base LLMs and LLMs customized through fine-tuning provide a generated text response.
5. The generated text response is sent to **Amazon Polly**, which in turn returns an audio stream of the response. The audio format is returned to the NPC and delivered as NPC synthesized speech.
6. For RAG NPC requests, **Lambda** submits the request to **Amazon Bedrock** to generate a vectorized representation from the embeddings model. **Lambda** then searches for relevant information from an **Amazon OpenSearch Service** vector index.
7. **OpenSearch Service** provides a similarity search capability to provide relevant context that augments the generated text request based on the vectorized representation of the request from **Amazon Bedrock**.
8. The relevant context and original text request are sent to LLMs hosted on **Amazon Bedrock** to provide a generated text response. **Amazon Polly** then delivers the response as synthesized speech to the NPC.
9. Game narrative writers add game-specific training data to create custom models using the FMOps process or add game lore data to hydrate the vector database.
10. Infrastructure and DevOps engineers manage the architecture as code using the **AWS Cloud Development Kit (AWS CDK)** and monitor the Guidance using **Amazon CloudWatch**.

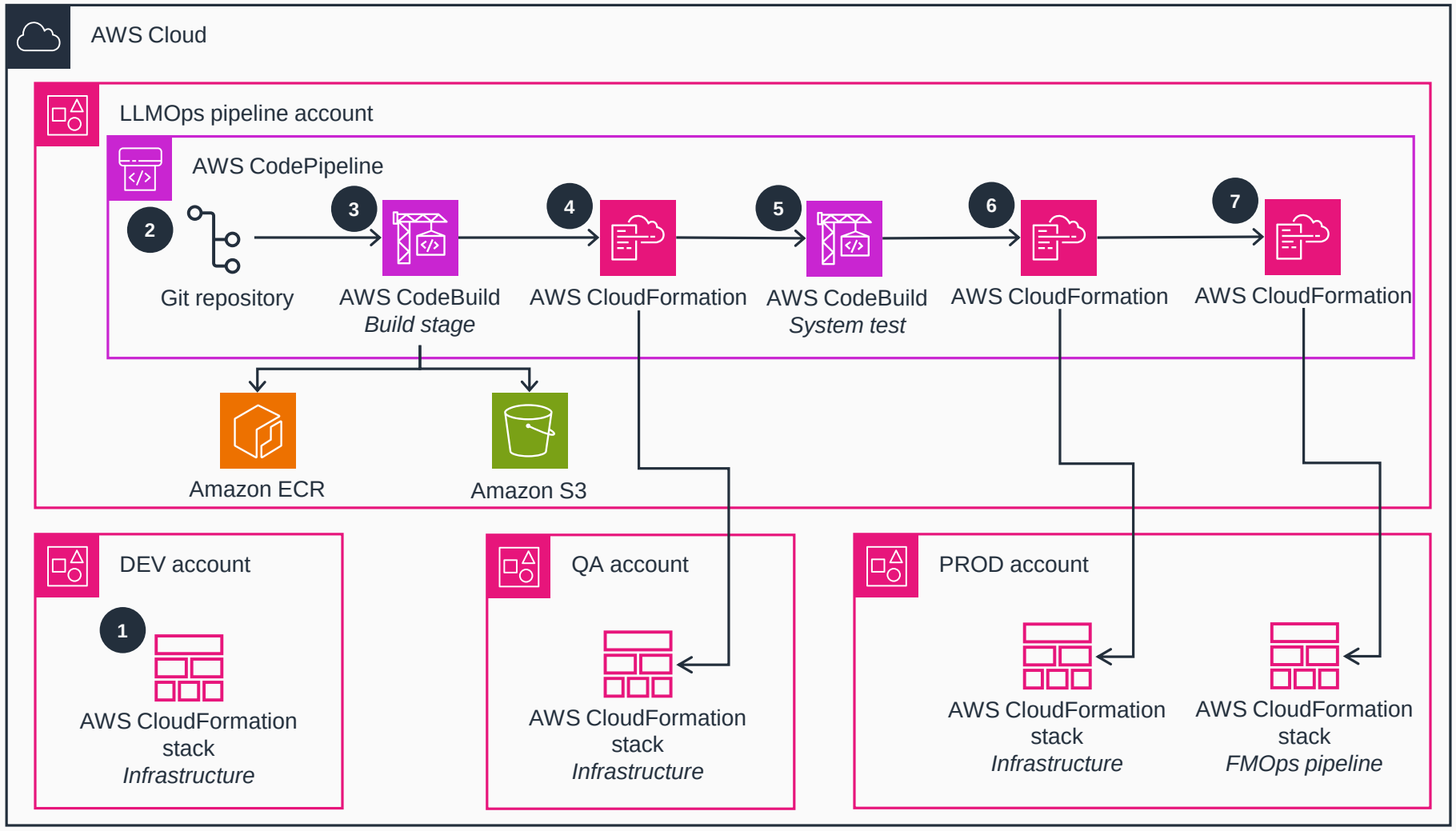
Reviewed for technical accuracy February 10, 2025

AWS Reference Architecture

Guidance for Dynamic Non-Player Character (NPC) Dialogue on AWS

LLMOps Pipeline

This architecture diagram shows the processes of deploying an LLMOps pipeline on AWS.



1. Infrastructure engineers build and test the codified infrastructure using **AWS CDK**.
2. Updates to infrastructure code are committed to the Git repository, invoking the continuous integration and continuous deployment (CI/CD) pipeline within the Toolchain AWS account.
3. Infrastructure assets, such as docker containers and **AWS CloudFormation** templates, are compiled and stored in **Amazon Elastic Container Registry (Amazon ECR)** and **Amazon Simple Storage Service (Amazon S3)**.
4. The infrastructure is deployed to the quality assurance (QA) AWS account as a **CloudFormation** stack for integration and system testing.
5. **AWS CodeBuild** initiates automated testing scripts that verify that the architecture is functional and ready for production deployment.
6. Upon successful completion of all systems tests, the infrastructure is automatically deployed as a **CloudFormation** stack into the Production (PROD) AWS account.
7. The FMOps pipeline resources are also deployed as a **CloudFormation** stack into the PROD AWS account.

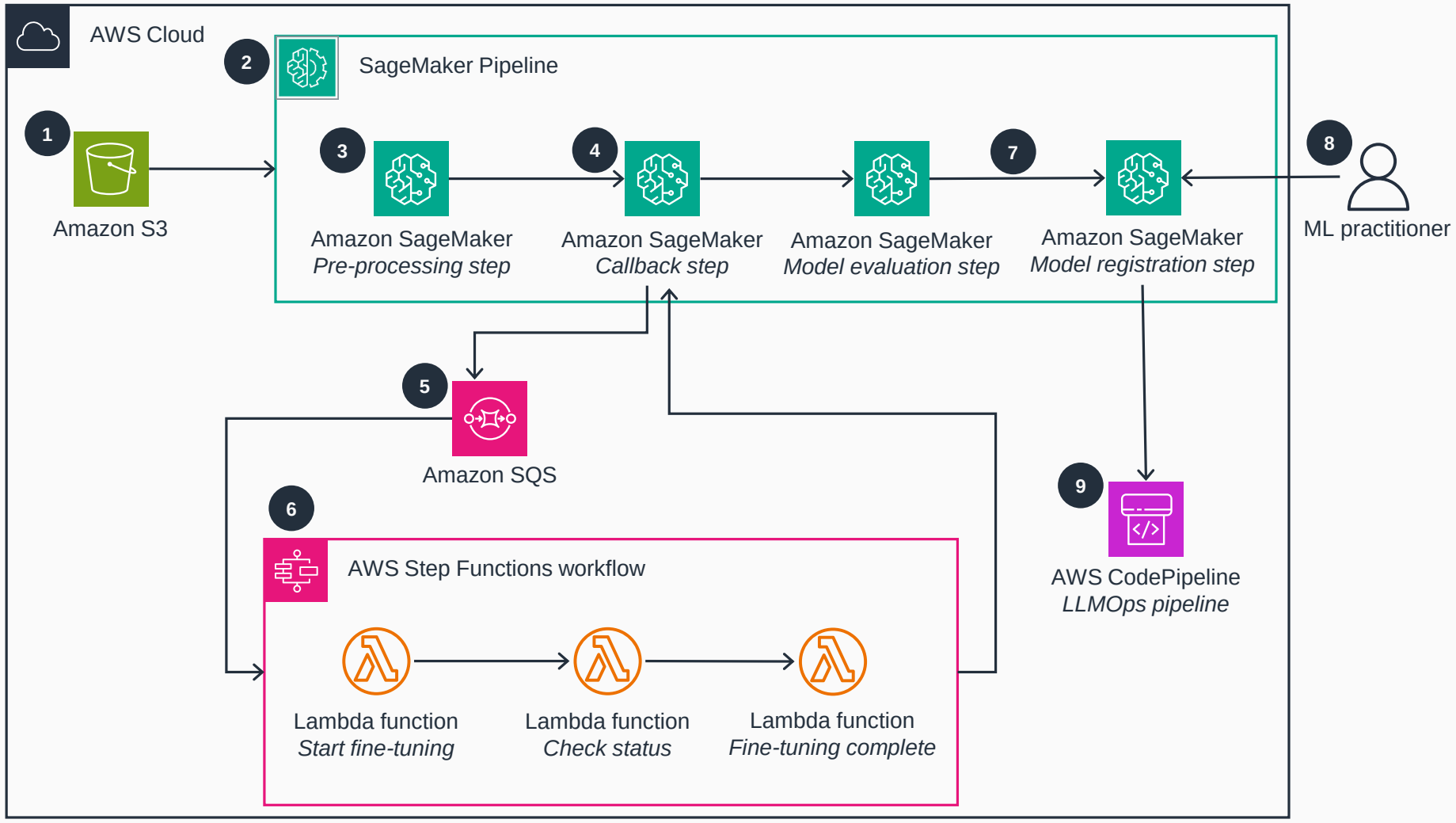
Reviewed for technical accuracy February 10, 2025

AWS Reference Architecture

Guidance for Dynamic Non-Player Character (NPC) Dialogue on AWS

FMOps Pipeline

This architecture diagram shows the process of tuning a generative AI model using FMOps.



1. Game lore text documents are uploaded to an **S3** bucket.
2. The document object upload event invokes **Amazon SageMaker Pipelines**.
3. The preprocessing step runs a **SageMaker** processing job to pre-process the text documents for model fine-tuning and model evaluation.
4. The callback step allows **SageMaker Pipelines** to integrate with other AWS services by sending a message to an **Amazon Simple Queue Service (Amazon SQS)** queue. After sending the message, **SageMaker Pipelines** waits for a response from the queue.
5. **Amazon SQS** manages the message queue that coordinates tasks between the **SageMaker Pipelines** and the **AWS Step Functions** workflow.
6. The **Step Functions** workflow orchestrates the process of fine-tuning the LLM. Once a model has been fine-tuned, **Amazon SQS** sends a success message back to the **SageMaker Pipelines** callback step.
7. The model evaluation step runs a **SageMaker** processing job to evaluate the fine-tuned model's performance. The tuned model is stored in the **Amazon SageMaker Model Registry**.
8. Machine learning (ML) practitioners review the tuned model and approve it for production use.
9. An **AWS CodePipeline** workflow is invoked to deploy the approved model into production.

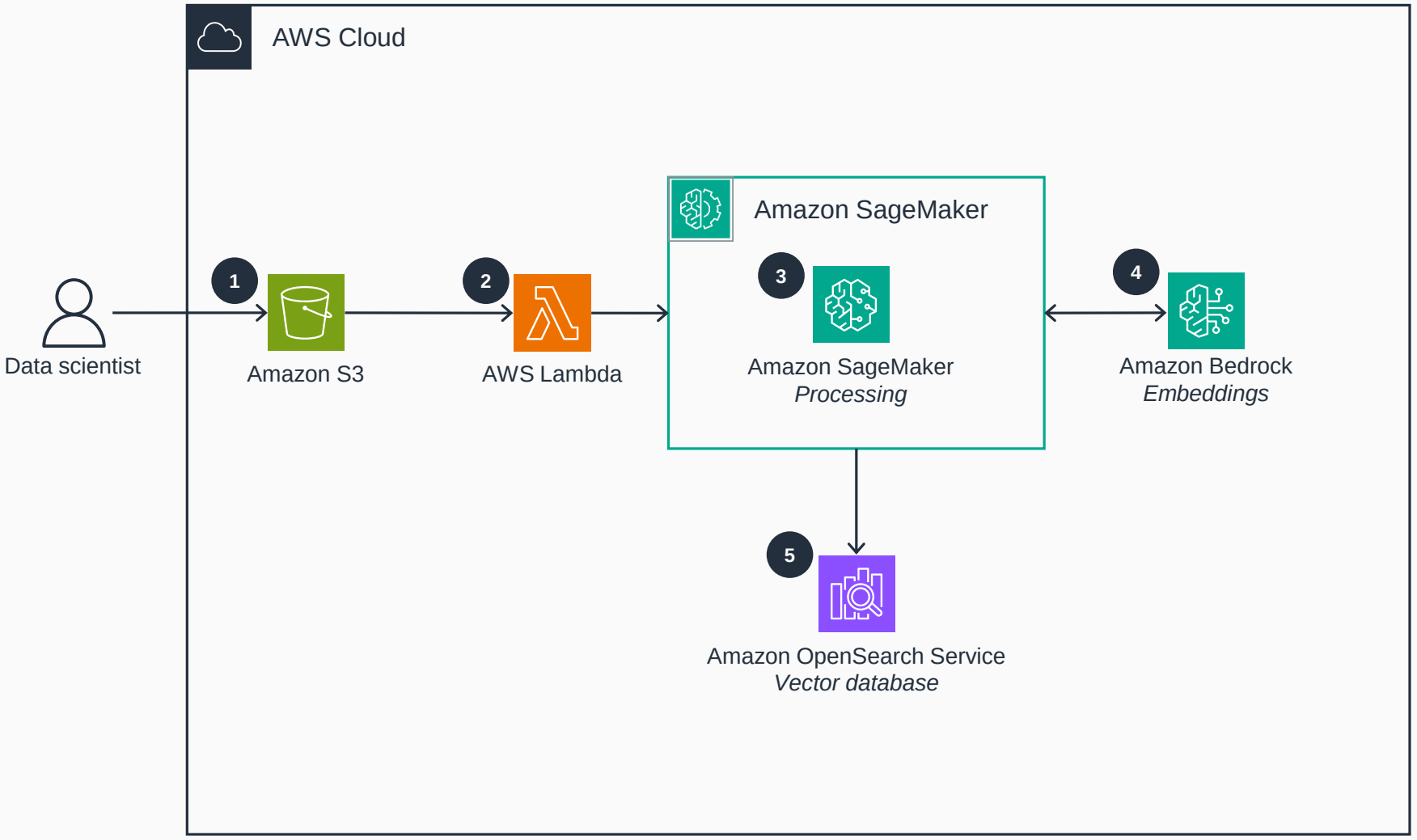
Reviewed for technical accuracy February 10, 2025

AWS Reference Architecture

Guidance for Dynamic Non-Player Character (NPC) Dialogue on AWS

Database Hydration

This architecture diagram shows the process for database hydration by vectorizing and storing gamer lore for RAG



1. A data scientist uploads game lore text documents to an **S3** bucket.
2. The object upload invokes a **Lambda** function to launch a **SageMaker** processing job.
3. A **SageMaker** processing job downloads the text document from **Amazon S3** and splits the text into multiple chunks.
4. The **SageMaker** processing job then submits each chunk of text to an **Amazon Titan** embeddings model hosted on **Amazon Bedrock** to create a vectorized representation of the text chunks.
5. The **SageMaker** processing job then ingests both the text chunk and the vector representation into **OpenSearch Service** for RAG.

Reviewed for technical accuracy February 10, 2025

AWS Reference Architecture