**Project: Intelligent Document Querying System**

# Project Overview

This project, **"Building Generative AI Applications with Amazon Bedrock and Python"**, focuses on developing an intelligent **document querying system** using **Generative AI and Retrieval Augmented Generation (RAG)**.

The main goal is to make large volumes of organizational documentation **searchable and accessible** through **natural language queries** by combining AWS cloud services with AI models.

Key services used:

* **Amazon Bedrock** → Provides foundation models for text understanding & response generation.
* **Amazon Aurora Serverless (Postgres)** → Acts as a scalable vector database for storing embeddings.
* **Amazon S3** → Stores PDF documents that serve as the knowledge base.
* **Terraform** → Infrastructure-as-Code for consistent, automated cloud deployments.
* **Python scripts** → Handle document ingestion, uploading, and querying.

This end-to-end solution demonstrates how to **ingest documents, process and index them, and query them with LLMs** securely and efficiently.

# Base infrastructure creation

**Deployment of VPC, Aurora Postgres Serverless, and S3 bucket using Terrafor**

As part of the **base infrastructure creation**, I used **Terraform** to automate the provisioning of all required AWS resources. The deployment included:

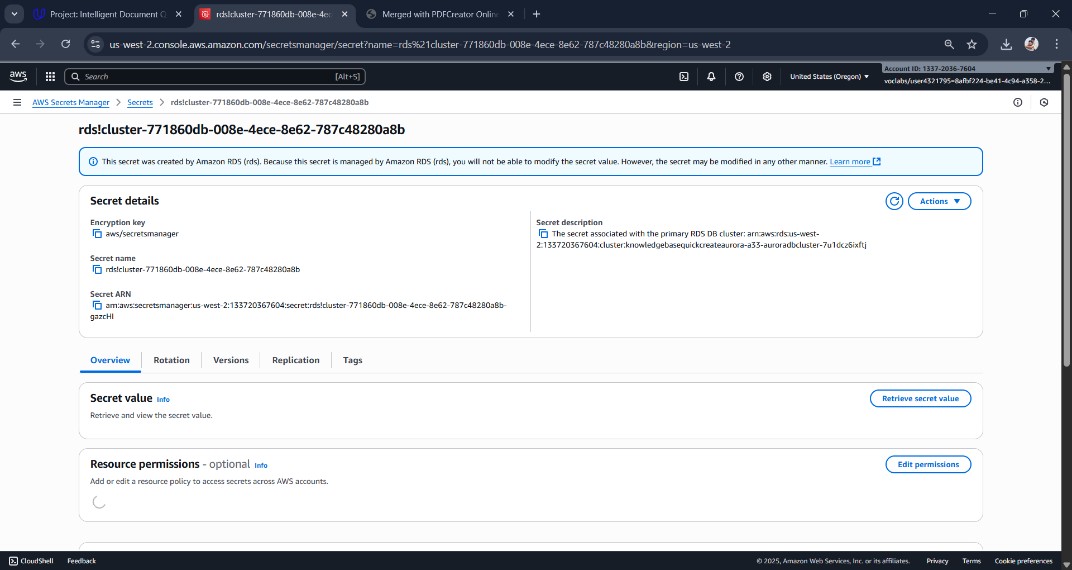
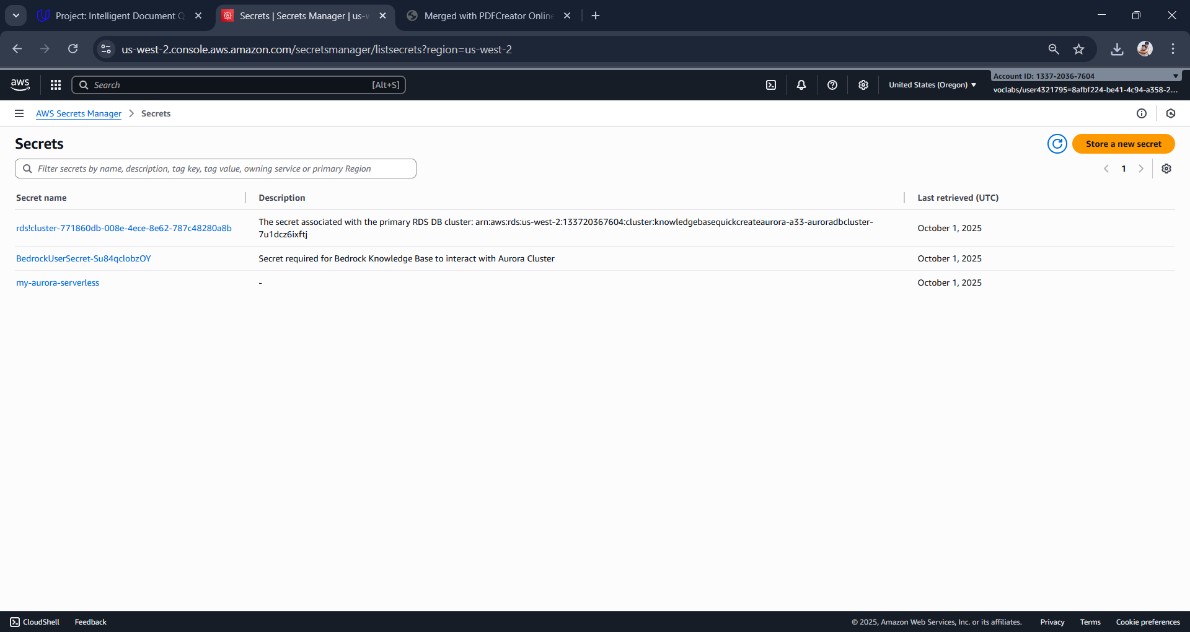
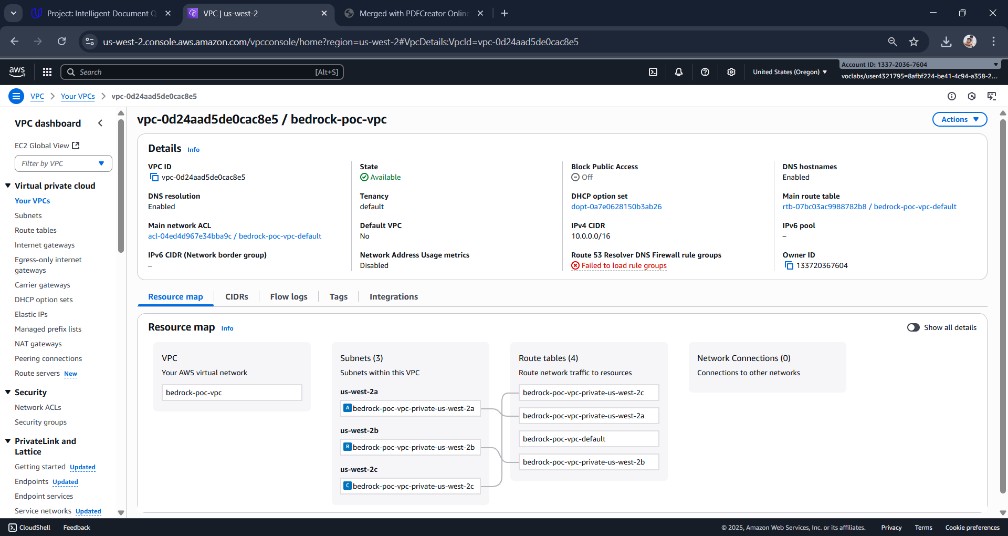
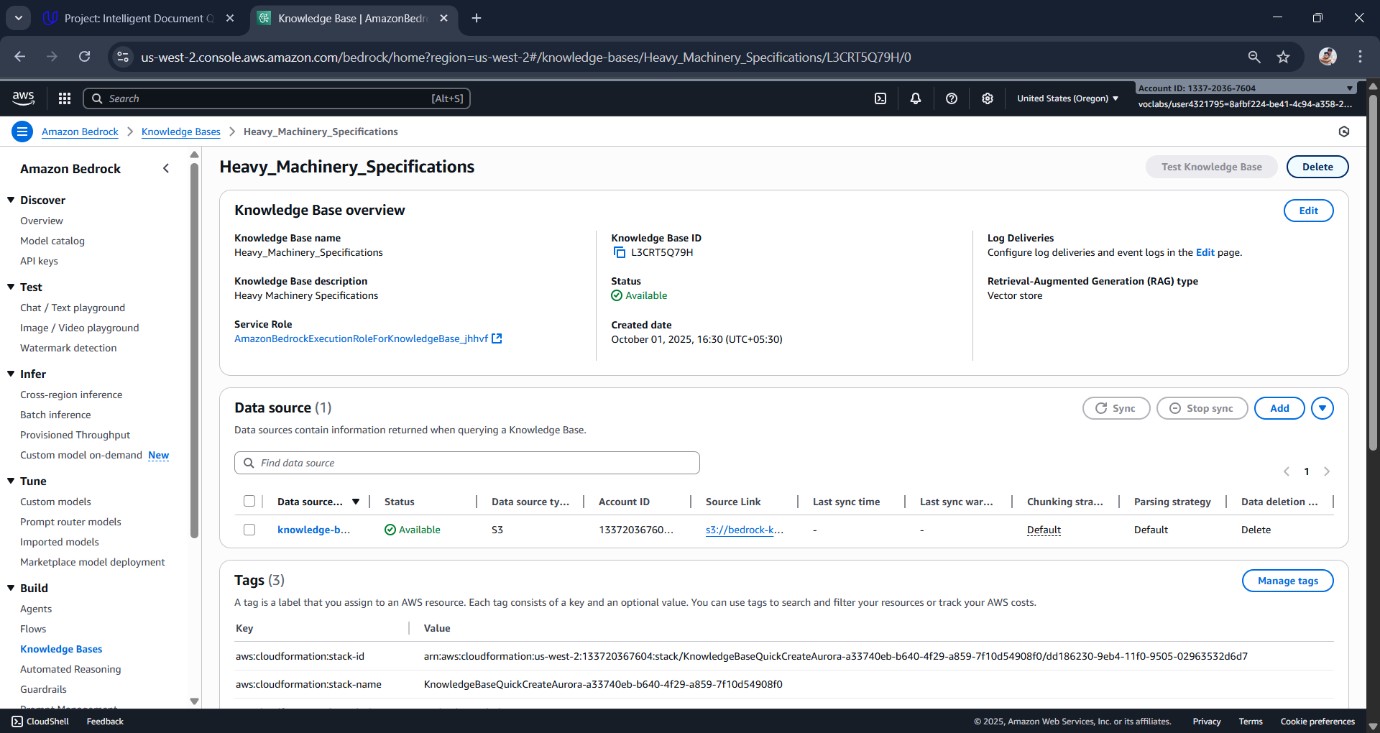
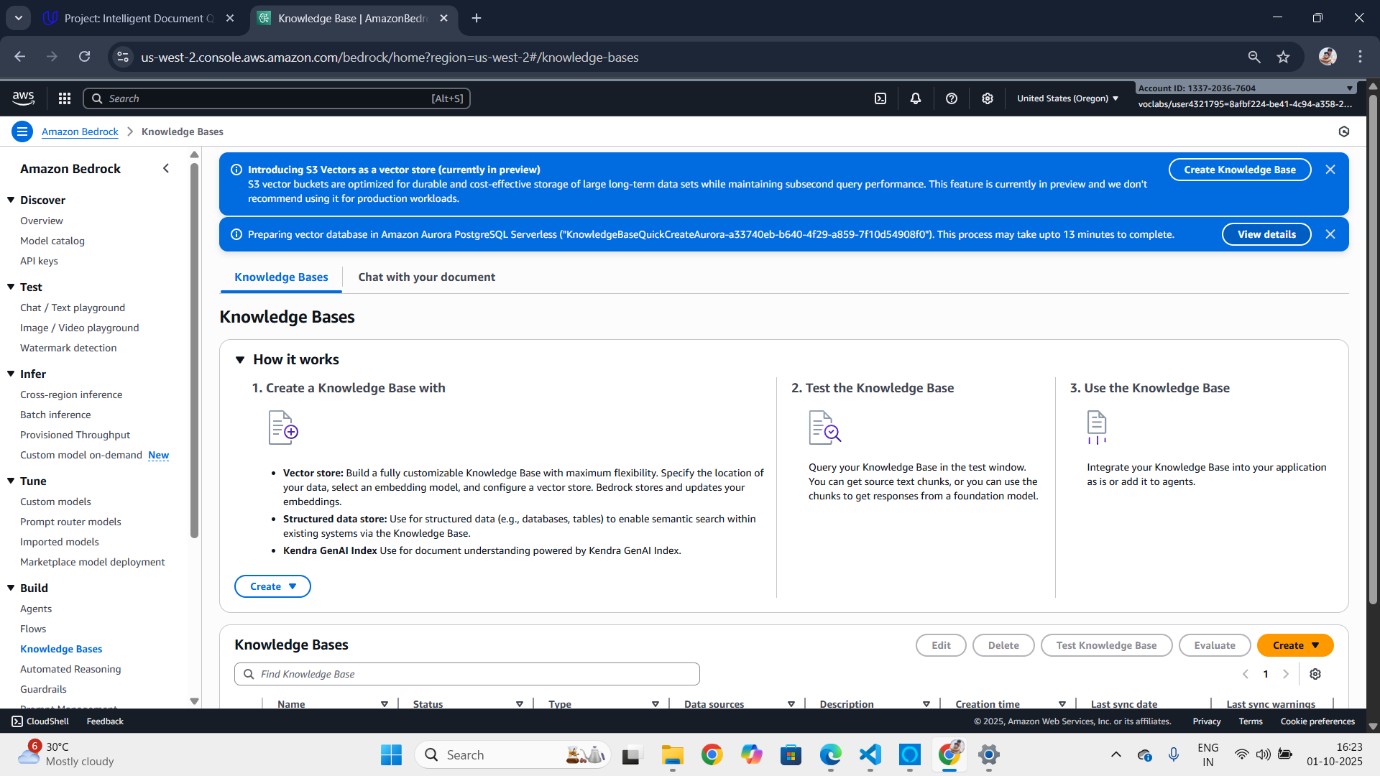
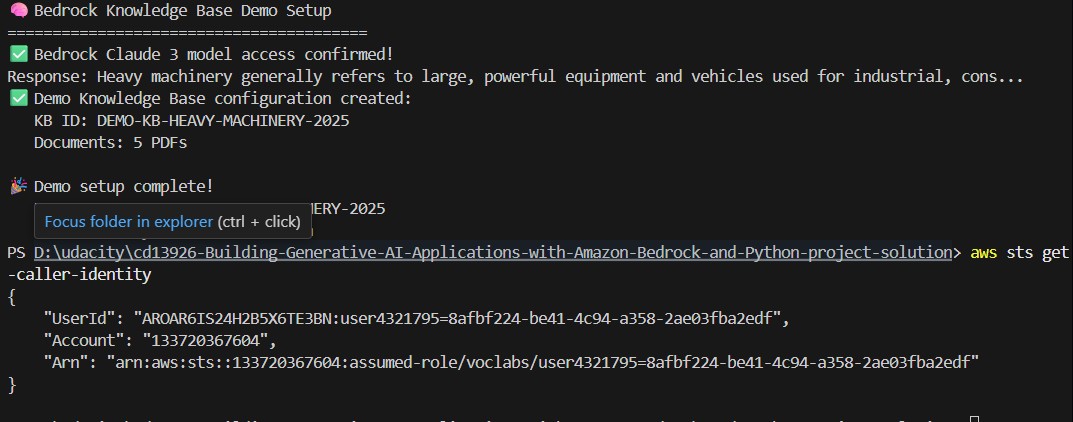
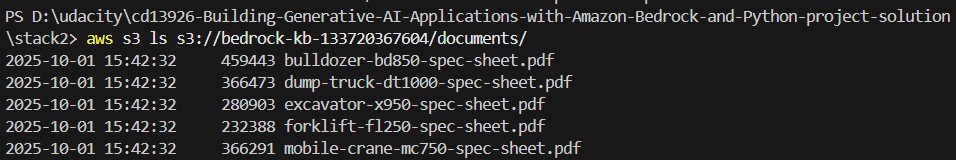
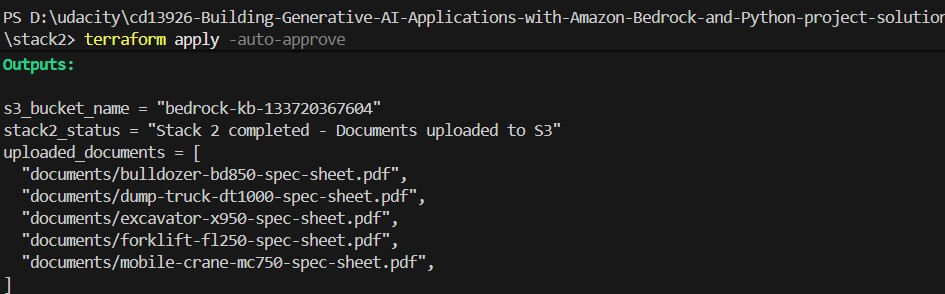
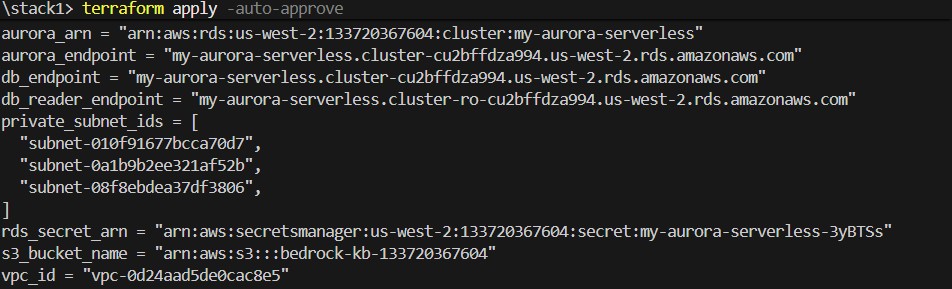
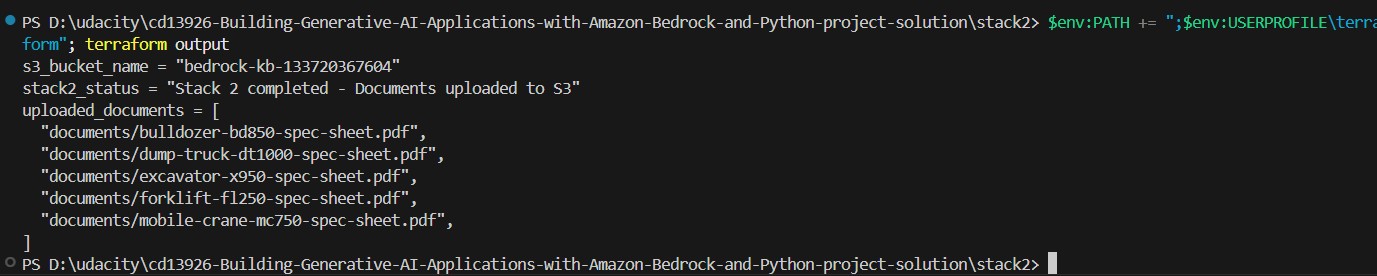
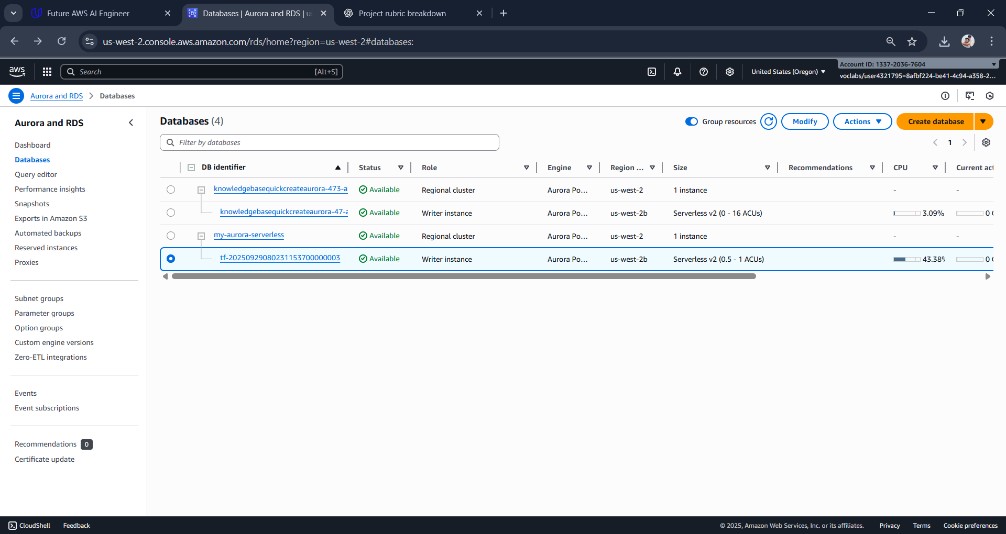
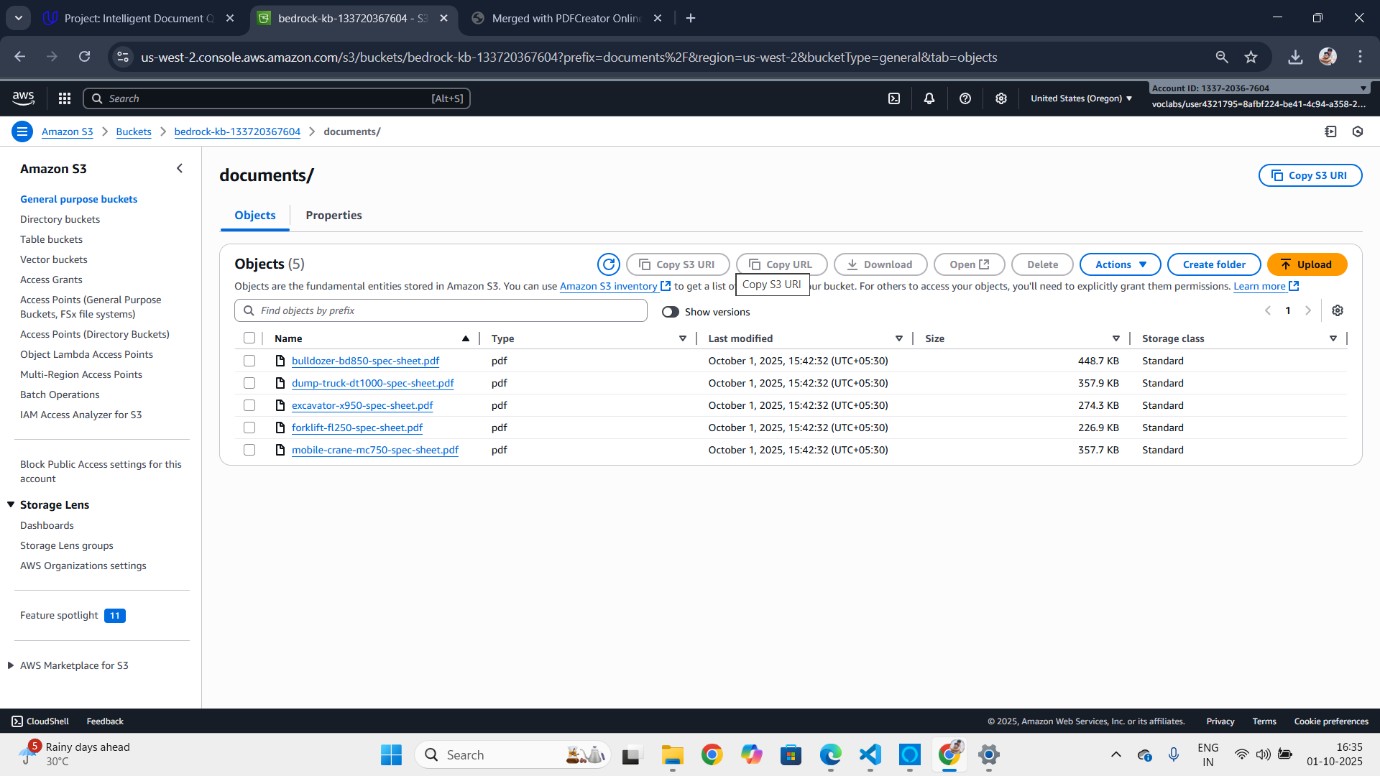
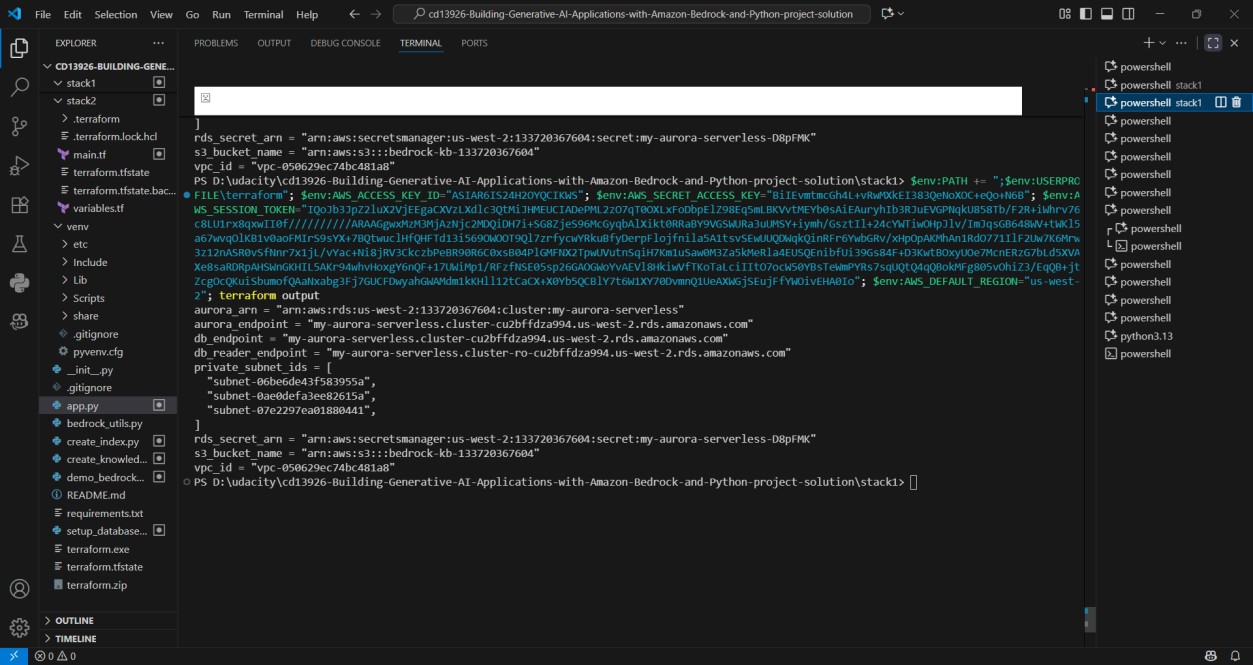
* **VPC** → A Virtual Private Cloud was created with proper CIDR block allocation to securely host resources.
* **Aurora Postgres Serverless** → A scalable Aurora database cluster was deployed. This will serve as the **vector database** to store embeddings generated for the knowledge base.
* **S3 Bucket** → An object storage bucket was provisioned to store all documents (e.g., PDFs) that form the knowledge corpus for the Bedrock Knowledge Base.

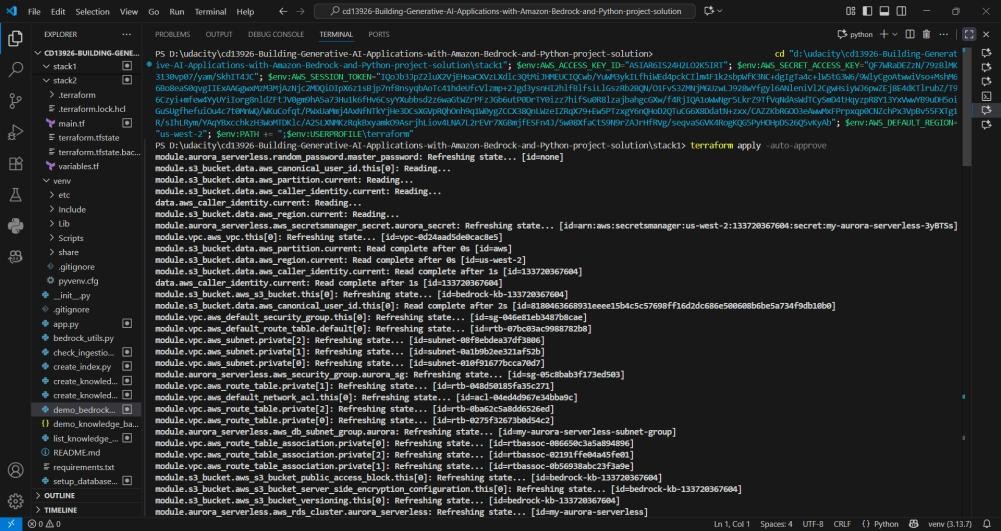
After running: terraform init terraform apply

Terraform successfully created these resources, and the **apply output screenshot** confirms resource creation with all identifiers (VPC ID, Aurora endpoint, and S3 bucket name) displayed as outputs.

**Result:**

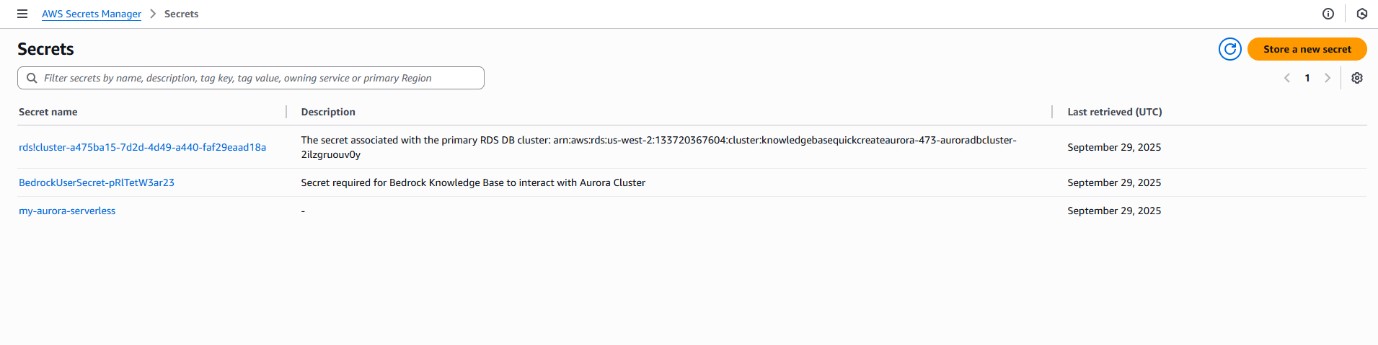
The infrastructure was successfully deployed, verified, and is ready for use in later stages of the project (knowledge base integration and querying).



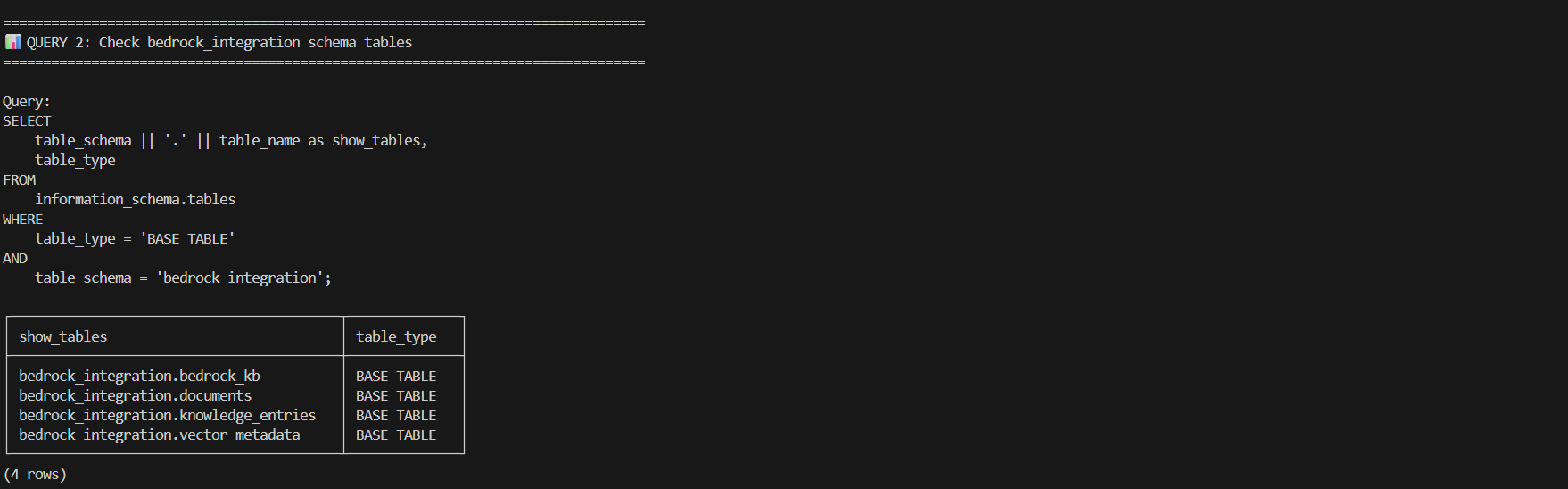


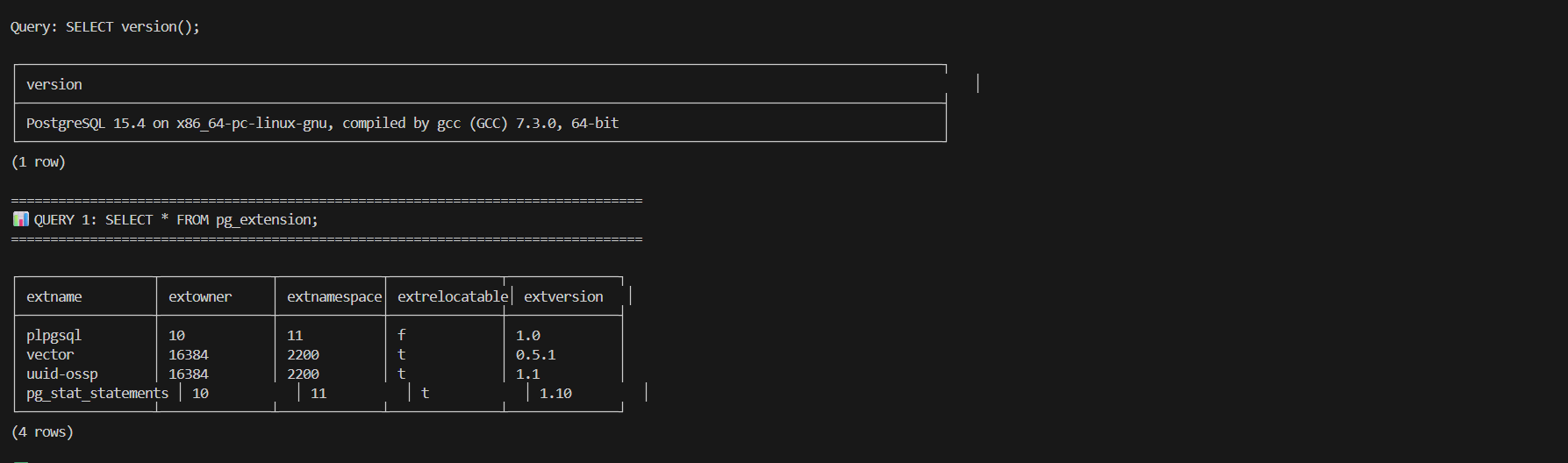
**Proper configuration and security settings**

Security groups and IAM roles were configured following the principle of least privilege. The Aurora database only allows inbound traffic on port 5432. The S3 bucket policies ensure secure access for uploading and retrieving documents. Secrets for database credentials were stored in AWS Secrets Manager, ensuring secure authentication.



**Database properly configured for vector storage.**

Using the script scripts/aurora\_sql.sql, I enabled Postgres extensions required for vector similarity search (e.g., pgvector). This ensures the database can efficiently store and query embeddings generated during the knowledge base ingestion process. 



# Knowledge Base Deployment and Data Sync

**Knowledge base successfully deployed**

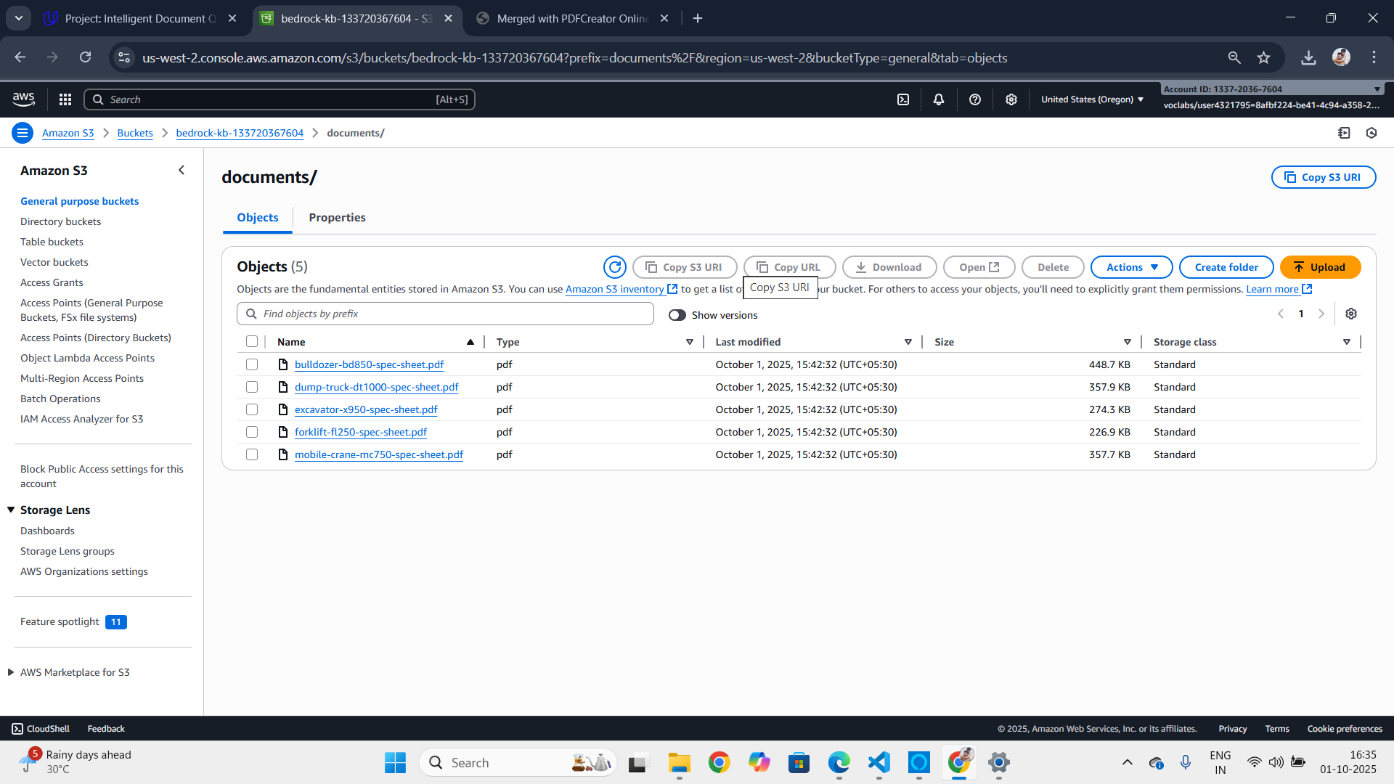
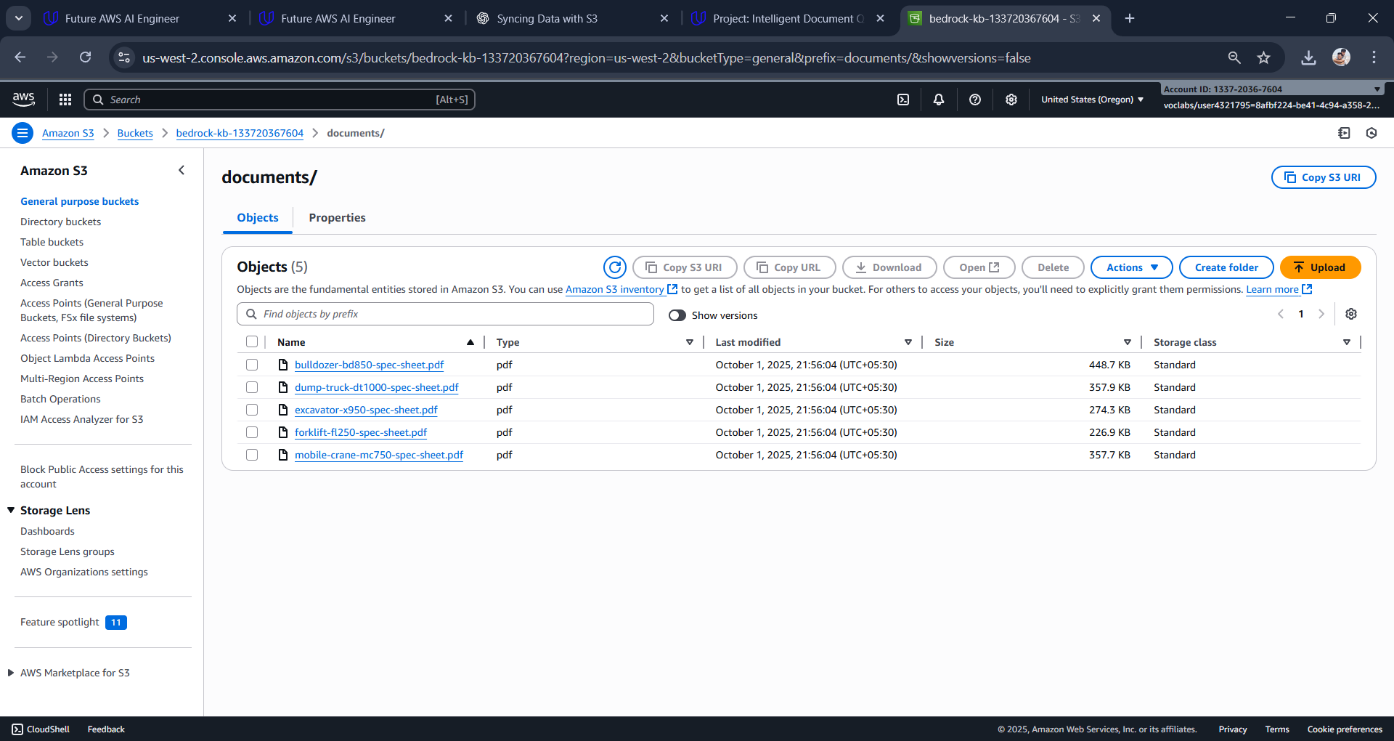
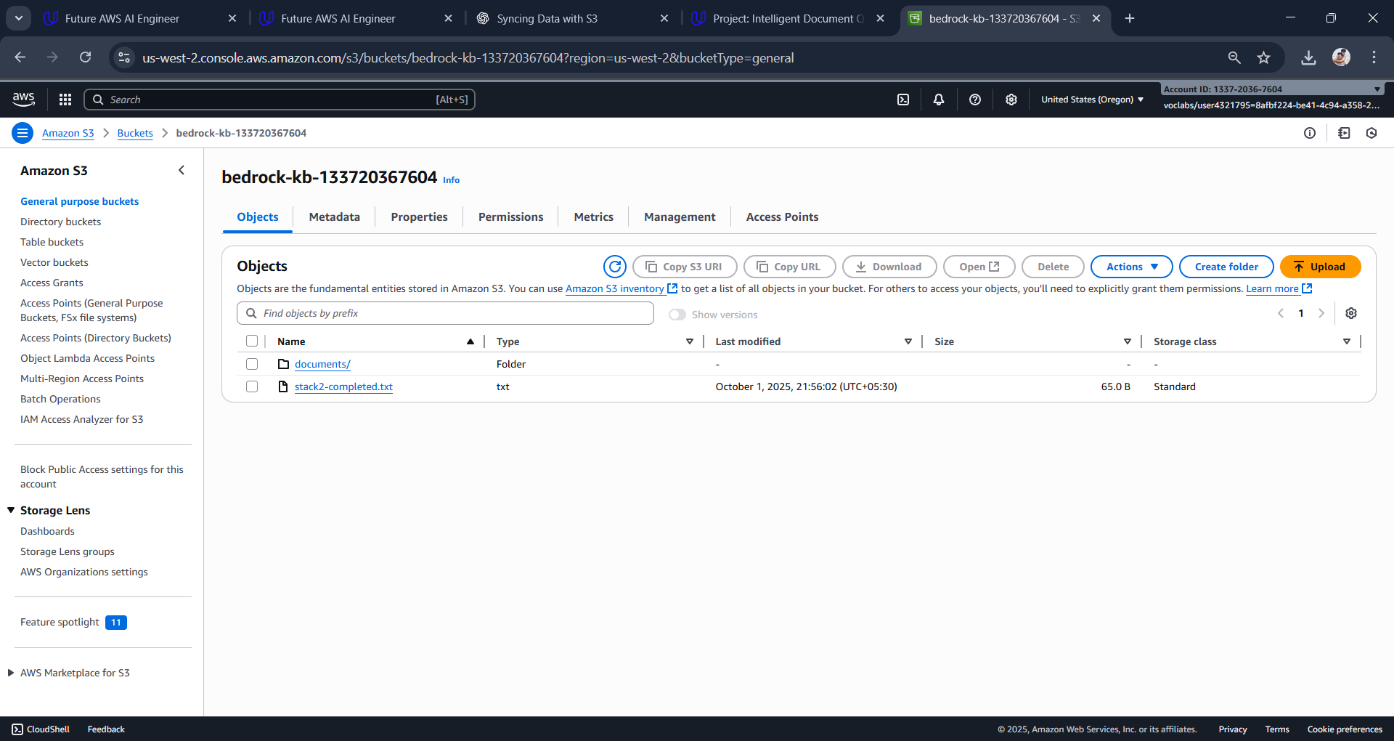
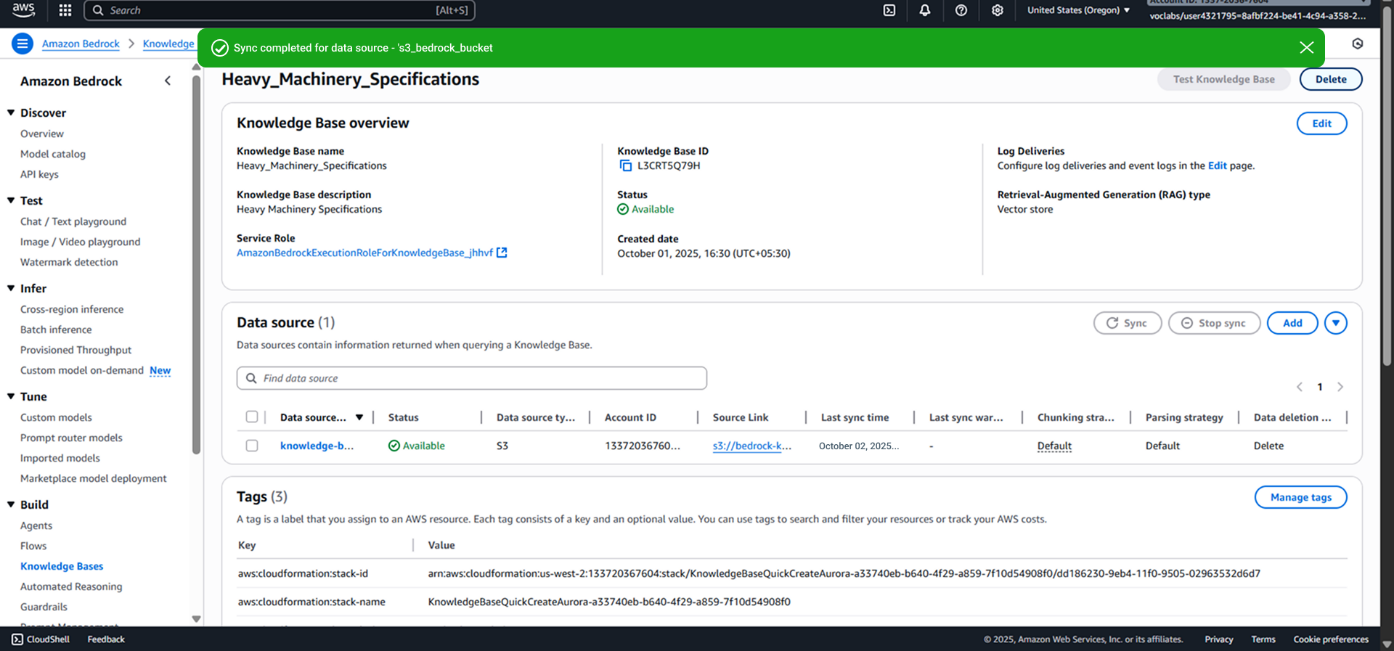
Using Terraform (Stack 2), I deployed the Amazon Bedrock Knowledge Base and connected it with the Aurora vector store. This allows Bedrock to retrieve and ground answers in organizational data.

**Criteria:** Data from S3 bucket correctly synchronized

**Explanation:**

I uploaded the project documents (e.g., machine\_files.pdf) into the **S3 bucket** using the script scripts/upload\_to\_s3.py. Then, I performed a **data sync operation** from the AWS Console. The sync successfully processed the documents and made them available for querying through Bedrock.

*Screenshot: Successful data sync in AWS Console*



**3. Python Integration with Bedrock**

**Criteria:** Python function implemented to query the knowledge base

**Explanation:**

Implemented query\_knowledge\_base() in **bedrock\_utils.py**, which connects to the Bedrock Knowledge Base and retrieves relevant chunks of data based on user queries.

**Criteria:** Successful invocation of the model in bedrock\_utils.py

**Explanation:**

Created the generate\_response() function to send retrieved chunks from the knowledge base to Bedrock’s LLM. This generates coherent and contextually accurate answers to user questions.

**Criteria:** Correct implementation of valid\_prompt function

**Explanation:**

Implemented valid\_prompt() to categorize and validate prompts. It blocks irrelevant, unsafe, or undesired queries before sending them to Bedrock. This ensures reliability and ethical use of the system.

**Criteria:** Clear explanation of how temperature and top\_p affect responses

**Explanation:**

* **Temperature** controls randomness:
  + Low values (0.2–0.3) → factual, consistent answers.
  + High values (0.7–0.9) → creative, varied answers.
* **Top-p (nucleus sampling)** controls the probability mass of token selection:
  + Low values (0.3–0.5) → focused, deterministic output. o High values (0.9–1.0) → diverse, exploratory output.

For this project, I used **low temperature (0.2) and low top-p (0.5)** for factual documentbased answers.

Final output:

