🩺 Visual Search RAG System with GenAIOps

A comprehensive **GenAIOps (LLMOps)** solution for an e-commerce Visual Search system. This project allows users to upload an image of a product and find similar items in a catalog using a "Caption-then-Embed" strategy powered by GPT-4o and Azure AI Search.

It features a robust **Governance & Guardrails** layer and a unified **Monitoring** system.

🌟 Key Features

- **Multimodal Search**: Uses GPT-4o to generate detailed visual descriptions of images, enabling semantic search using text embeddings.
- **Vector Store Agnostic**: Switch between **Azure AI Search** (Production) and **ChromaDB** (Local/Dev) via configuration.
- **Governance & Guardrails**:
    - **Safety Validator**: Detects prompt injection, jailbreaks, and unsafe content (using Azure Content Safety).
    - **Compliance Checker**: Detects and redacts PII (GDPR/HIPAA compliance).
    - **Hallucination Detection**: Checks responses for uncertain language.

🛠️ Prerequisites

- **Python 3.10+**
- **Azure Subscription** with:
    - **Azure OpenAI Service** (Deployments: gpt-4o, text-embedding-3-small)
    - **Azure AI Search** (Optional if using ChromaDB)
    - **Azure Content Safety** (Optional but recommended)
    - **Azure Monitor** (Optional)
- **Docker** (for containerization)

🚀 Setup & Installation

1. **Clone the Repository (optional if you're not using GitHub)**

```
None
git clone <your-repo-url>
cd week_14_llmops
```

2. **Create a Virtual Environment**

```
None
python -m venv venv
source venv/bin/activate  # On Windows: venv\Scripts\activate
```

3. **Install Dependencies**

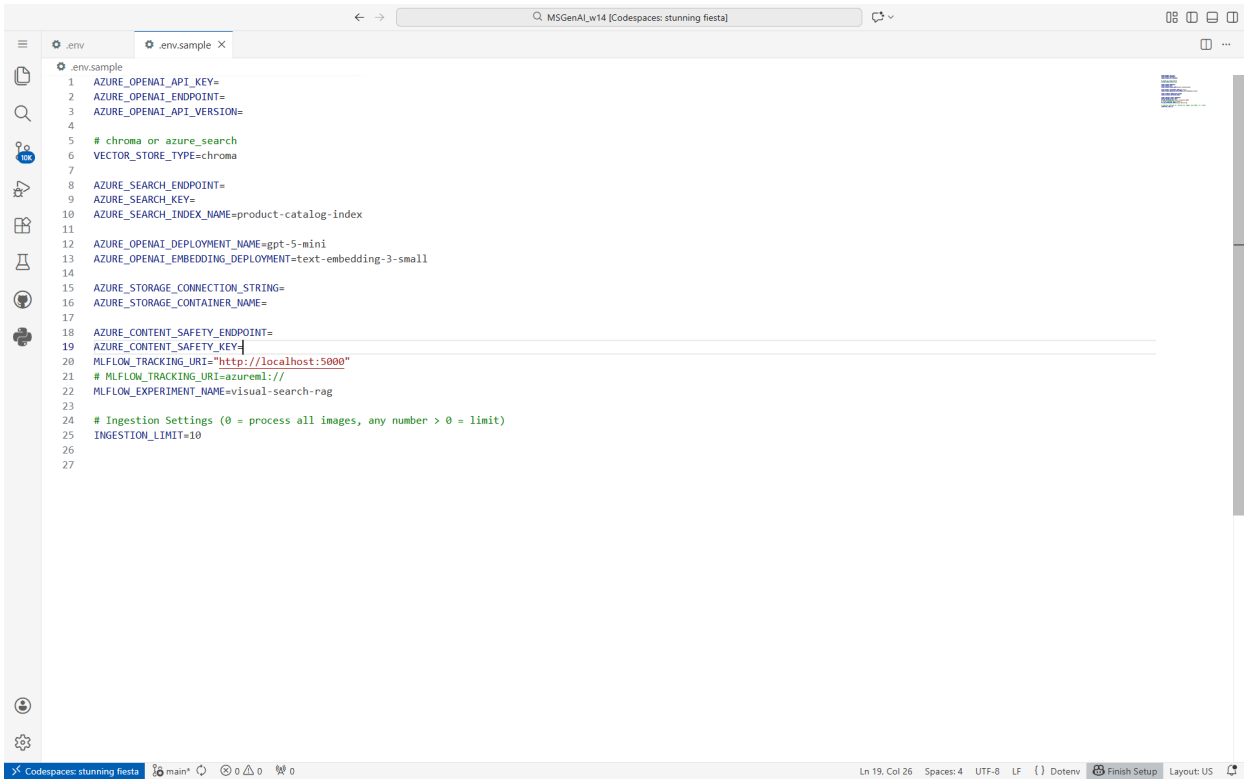
```
None
pip install -r requirements.txt
```

4. **Configure Environment**
    - Copy the sample configuration file:

```
None
cp .env.sample .env
```

    - Open .env and configure:
        - VECTOR_STORE_TYPE: Set to chroma (local) or azure_search (cloud).
        - AZURE_OPENAI_*: Add your keys and endpoints.
        - MLFLOW_TRACKING_URI: Set tracking URI (default http://localhost:5000 for local).

NOTE: We recommend **gpt-5-mini** and **text-embedding-3-small** models for LLM and Embedding model deployments respectively.



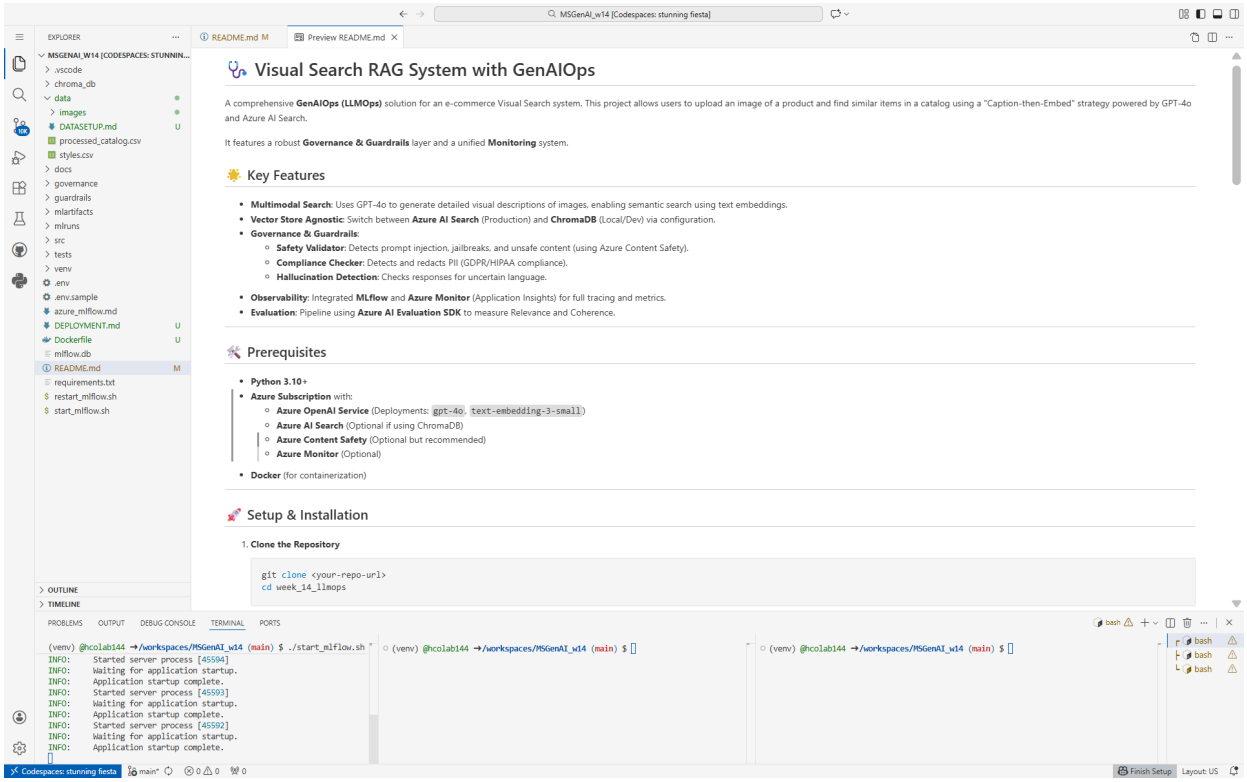
🏃 Usage Guide - The LLMOps Pipeline

1. Start Monitoring (MLflow)

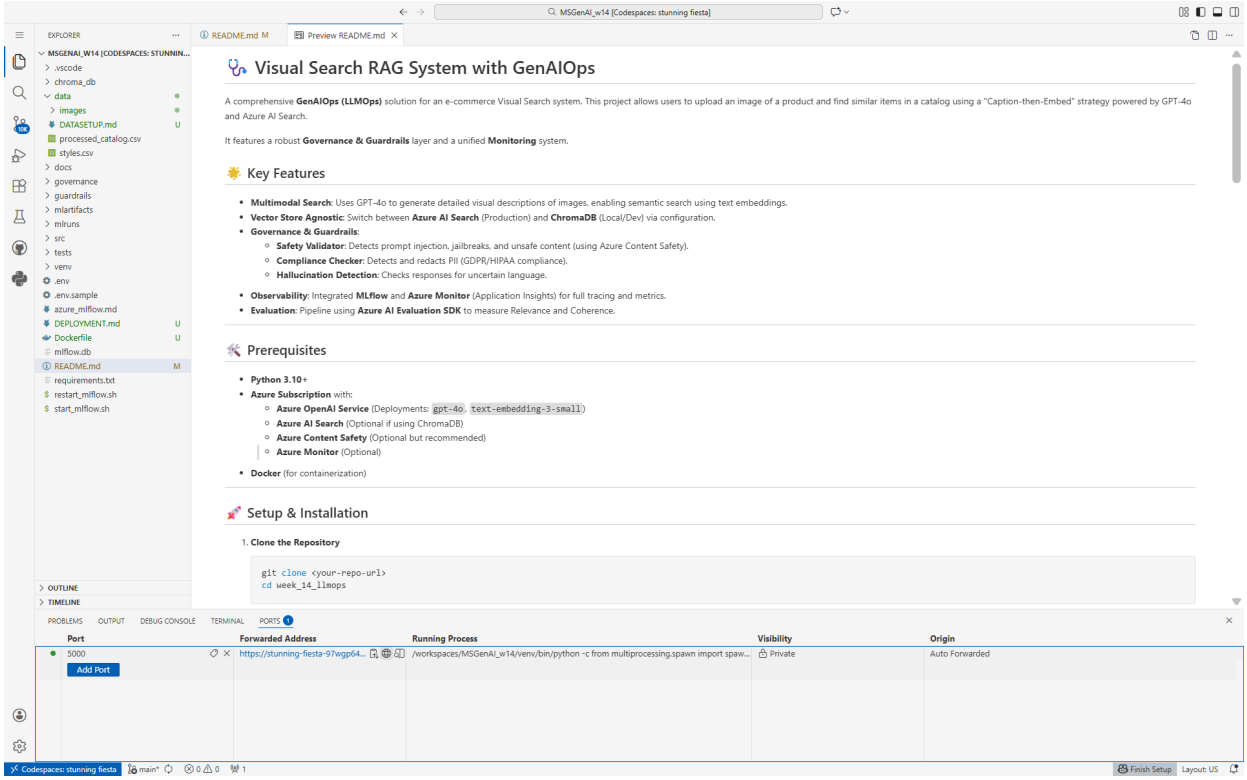
Start the local MLflow server to track all experiments and traces.

None

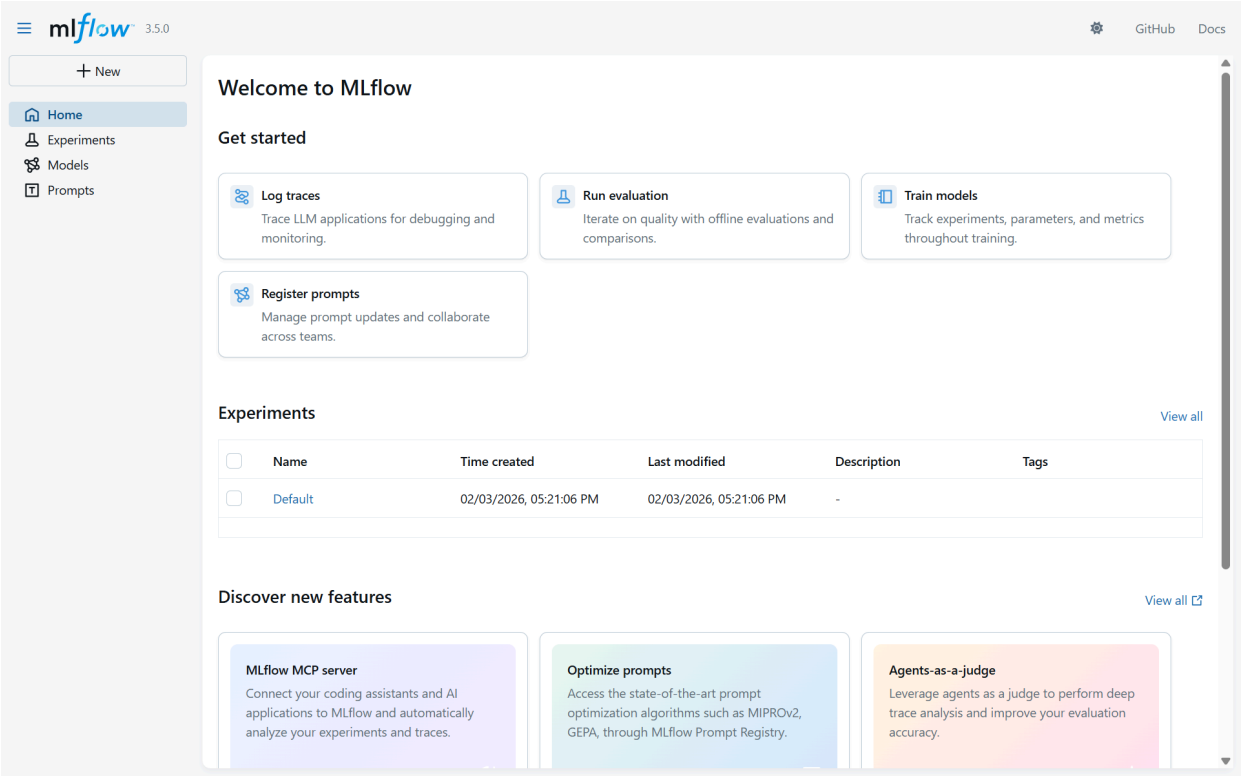
```
./start_mlflow.sh
```



- The dashboard can be viewed at http://localhost:5000.



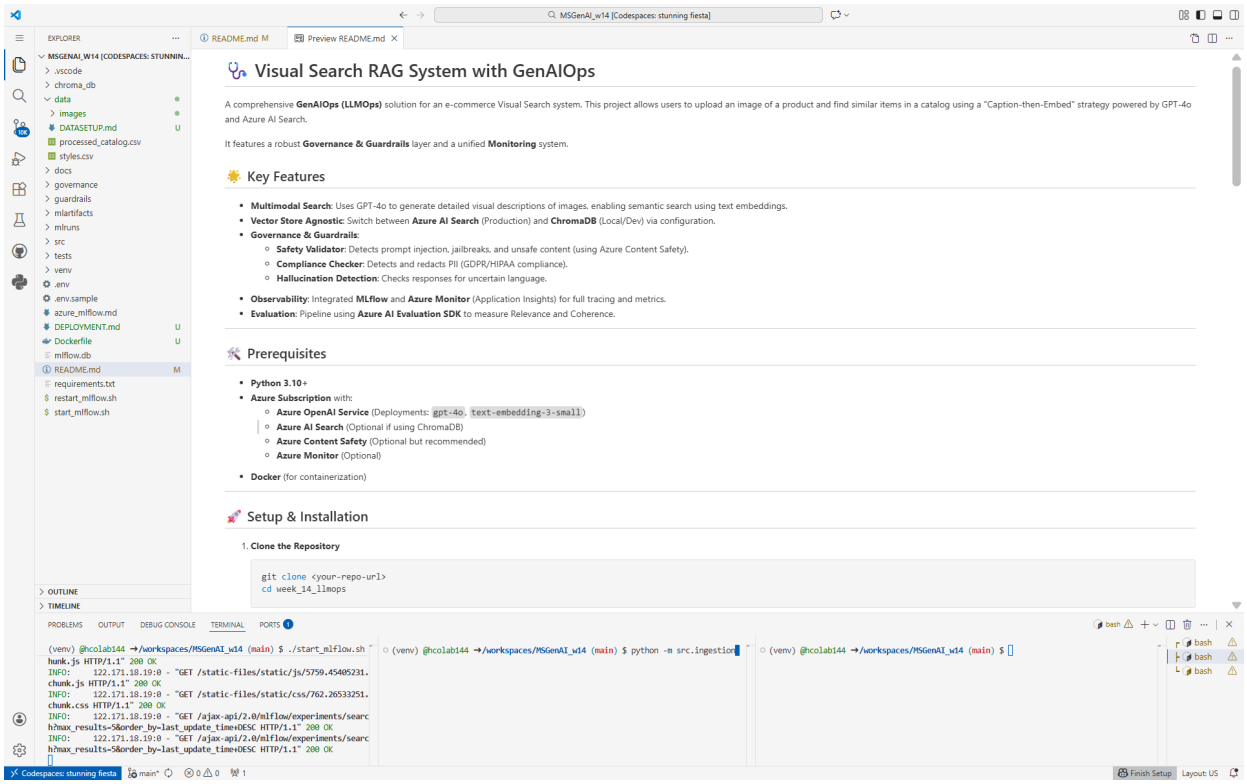
The MLFlow Dashboard opens in a new tab.



2. Data Ingestion

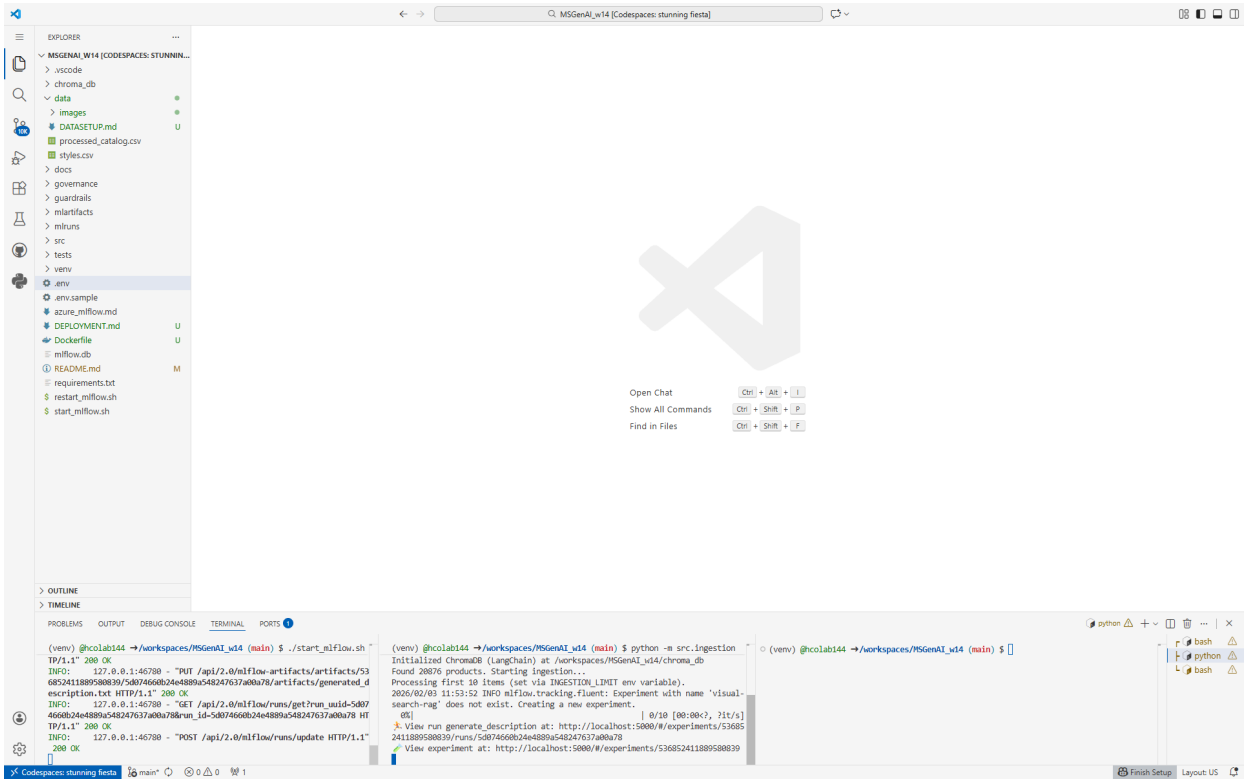
Populate the vector index (Chroma or Azure) with the product catalog.

```
None
# Ingest data into Vector Store
python -m src.ingestion
```

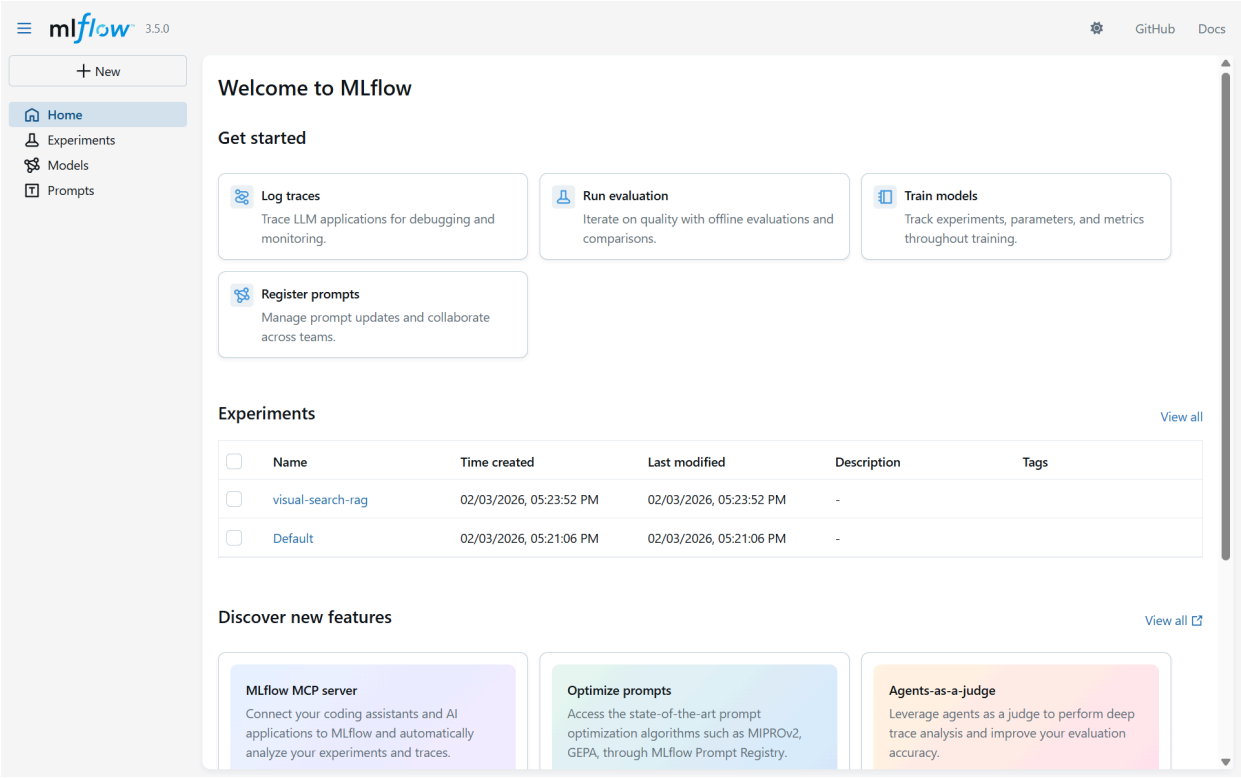


NOTE: The ingestion process typically takes a few minutes depending on the number of images. In this demo, we've used 10 images by defining the environment variable in .env file

```
None
# Ingestion Settings (0 = process all images, any number > 0 = limit)
INGESTION_LIMIT=10
```



You can track the experiments from the MLflow dashboard.

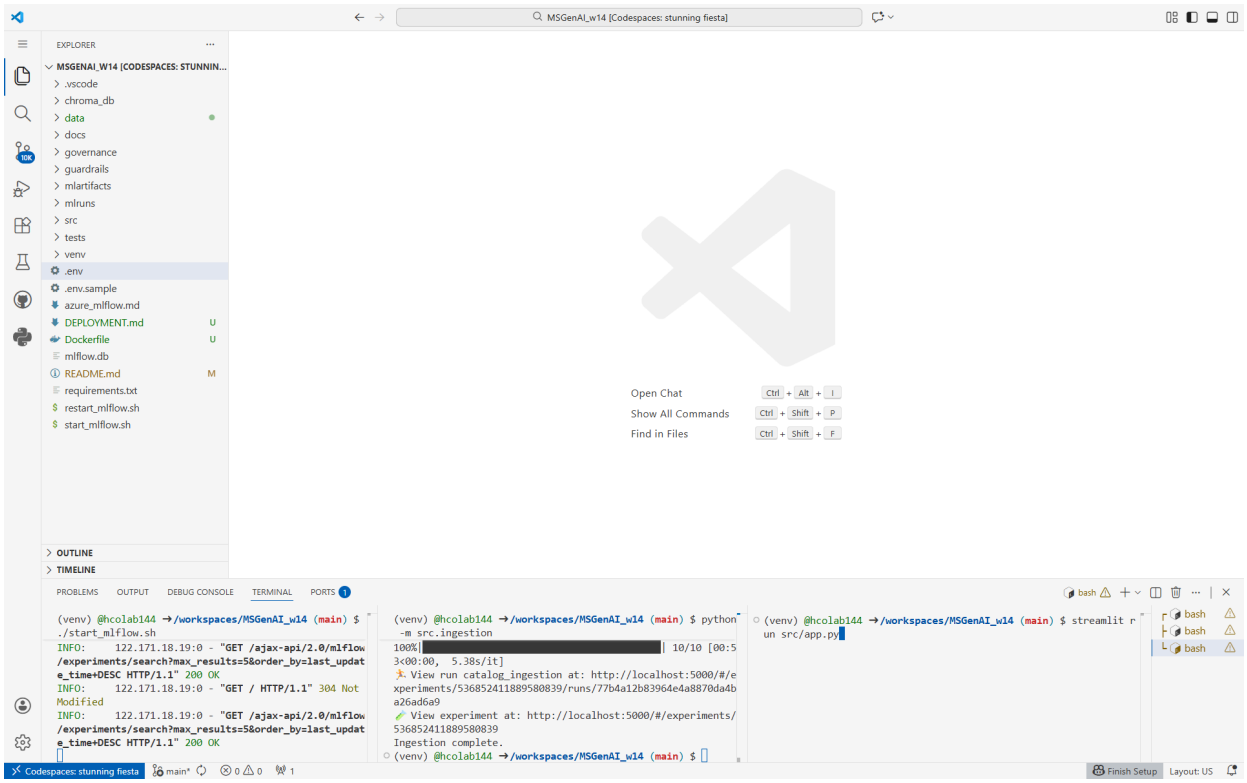


3. Running the Application

Launch the Streamlit interface. This handles the **RAG flow** with **Governance** checks.

```
None
streamlit run src/app.py
```

- **Governance in Action**: Try uploading an unsafe image or simulating a prompt injection to see the GovernanceGate block it.
- **Monitoring**: Check MLflow traces for every search request.



Integrating MLflow with Azure Machine Learning (AML)

A Comprehensive Step-by-Step Guide

This guide explains how to configure your Python applications to use Azure Machine Learning as a centralized backend for **MLflow Tracking**. This setup allows you to manage experiment logs, metrics, models, and GenAI traces in a unified, cloud-based environment.

---

1. Prerequisites

Before starting, ensure you have the following Azure resources provisioned:

- **Azure Machine Learning Workspace:** The central hub for your ML operations.
- **Azure Resource Group:** The container holding your AML workspace.

- **Python Environment:** Python 3.8+ is recommended.

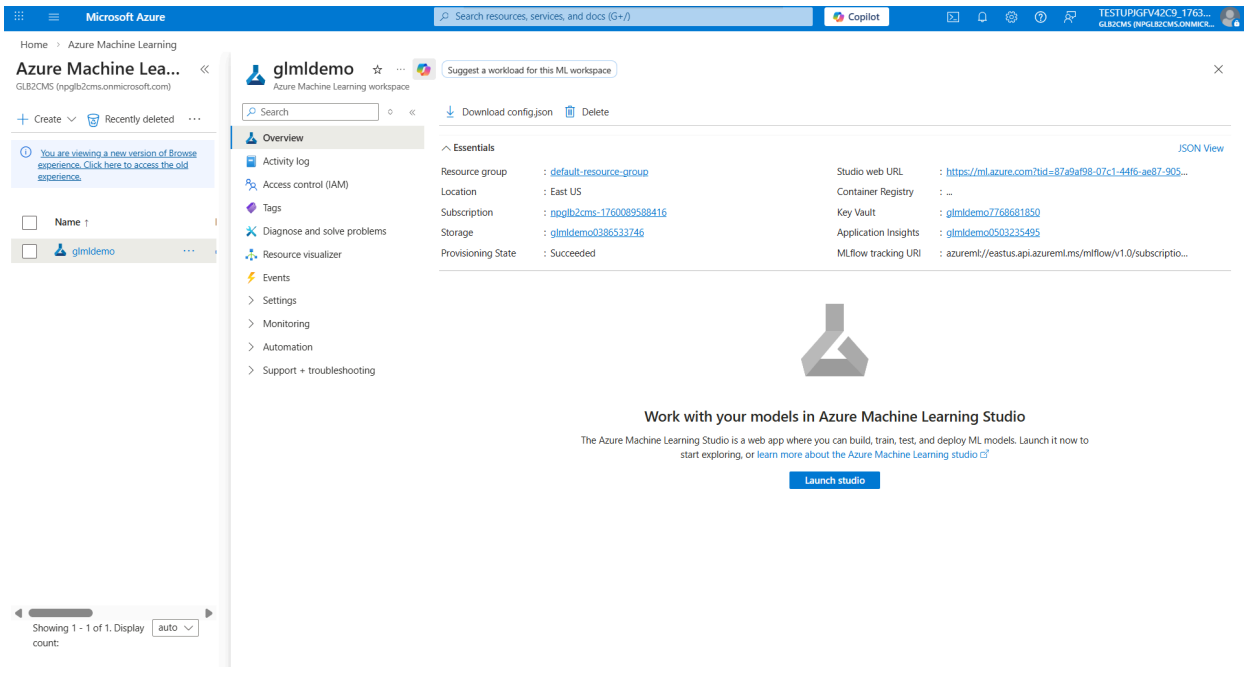
---

2. Retrieve the MLflow Tracking URI

To point MLflow to Azure, you must obtain the workspace-specific Tracking URI.

Option A: Via Azure Portal (UI)

1. Navigate to the Azure Portal.
2. Open your **Azure Machine Learning Workspace**.
3. On the **Overview** page, locate the **MLflow tracking URI** in the top-right section (you may need to click "JSON View" or "See more" if it is hidden).
4. Copy the URI. It follows this format: azureml://<region>.api.azureml.ms/mlflow/v1.0/subscriptions/<sub-id>/resourceGroups/<rg-name>/providers/Microsoft.MachineLearningServices/workspaces/<workspace-name>



Environment Configuration

Your application uses the MLFLOW_TRACKING_URI environment variable to determine where to send data.

For Local Development

Update your .env file or export the variable in your terminal to set tracking URI to localhost.

For production deployment (when deployed using Azure App Services), use the **Azure ML Tracking URI**.

```
None
# Local tracking (disabled)
# MLFLOW_TRACKING_URI="http://localhost:5000"

# Azure ML Tracking
MLFLOW_TRACKING_URI="azureml://eastus.api.azureml.ms/mlflow/v1.0/..."
```

Implementation and Verification

Update the .env file with the MLflow tracking URI

```
# MLFLOW_TRACKING_URI="http://localhost:5000"
MLFLOW_TRACKING_URI=azureml://eastus.api.azureml.ms/mlflow/v1.0/subscriptions/45c4849b-fed4-4092-a8f3-a0f98e38644b/resourceGroups/default-resourc
MLFLOW_EXPERIMENT_NAME=visual-search-rag

# Ingestion Settings (0 = process all images, any number > 0 = limit)
INGESTION_LIMIT=10
```

Authentication and Security

Azure Machine Learning requires authentication. By using azure-identity, you can avoid hardcoding credentials.

For Local Development

Simply authenticate via the Azure CLI. The library will use your developer identity:

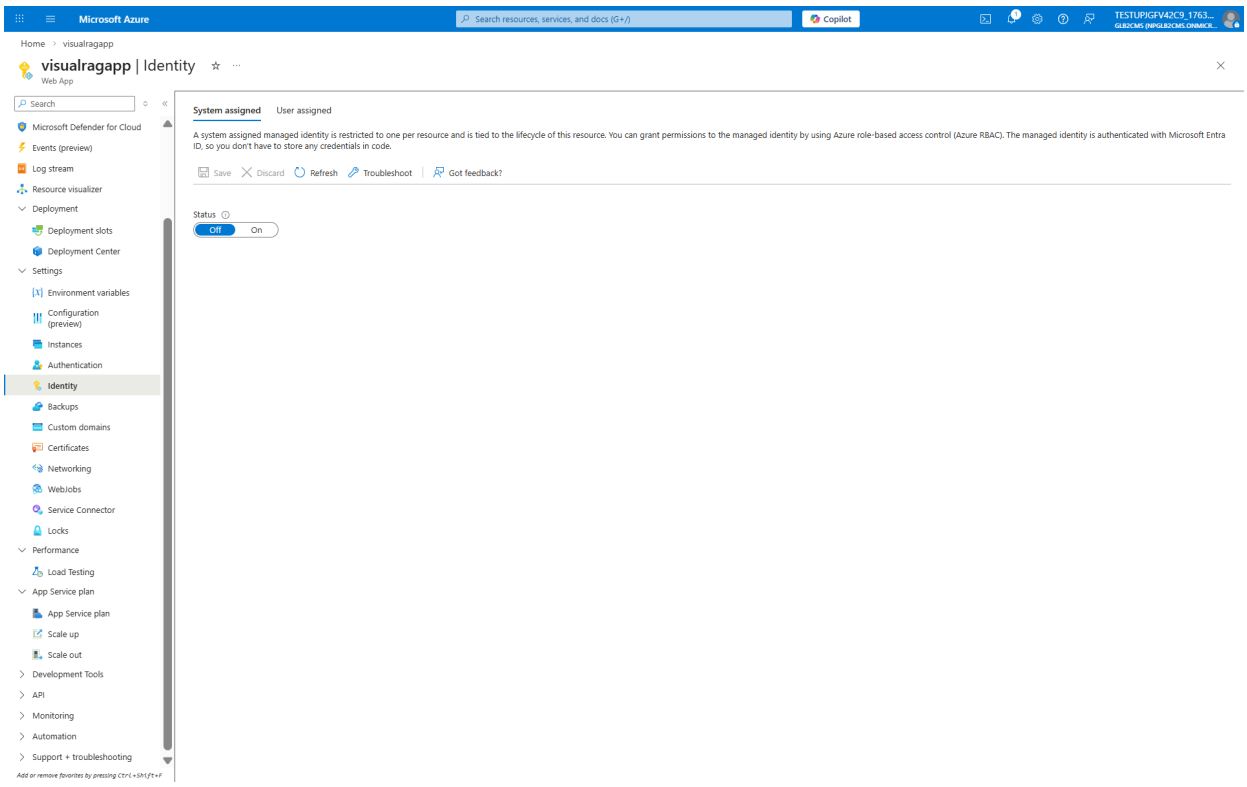
```
None
az login
```

For Production Application (Passwordless Login)

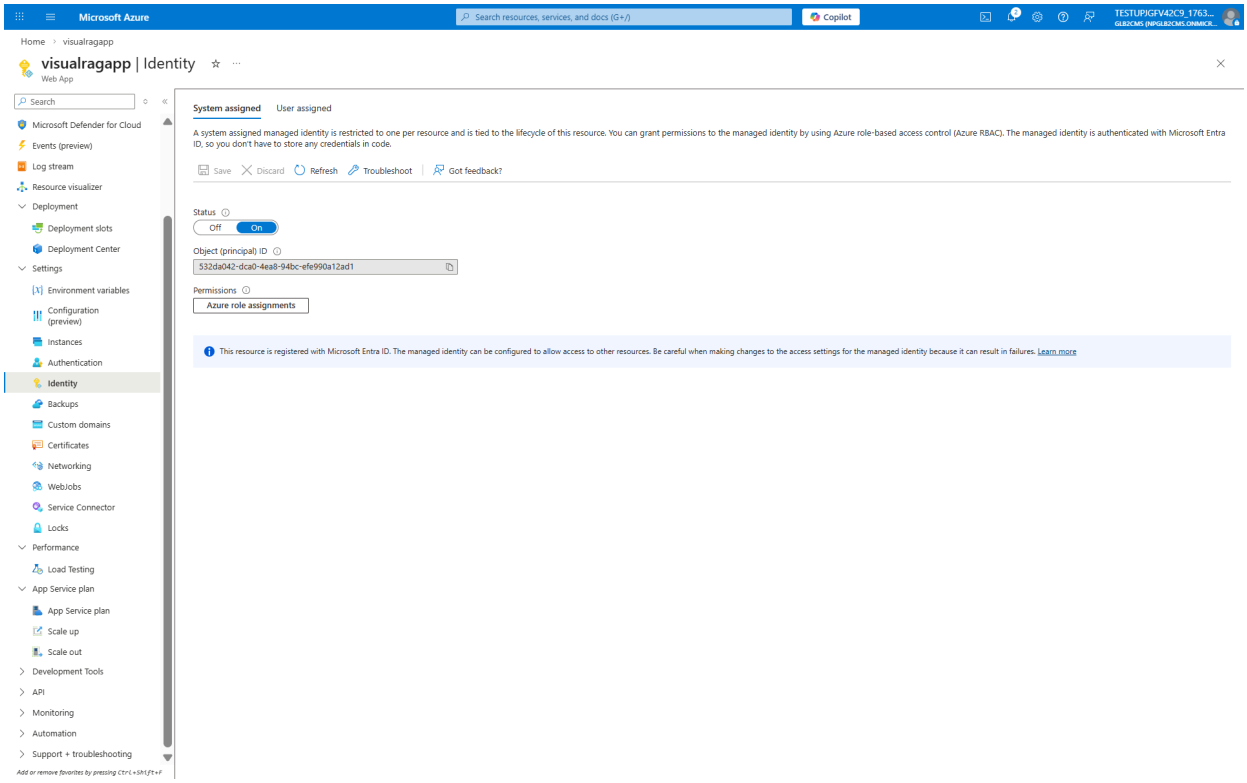
For production deployment, we'll use Managed Identity to ensure secure, credential-free communication happens between the RAG application and Azure Machine Learning Workspace. We'll define roles that will allow for the communication to happen between these Azure services.

NOTE: This section assumes that you've created **Azure App Service.** You may revisit this section when deployment to Azure App Service is complete.

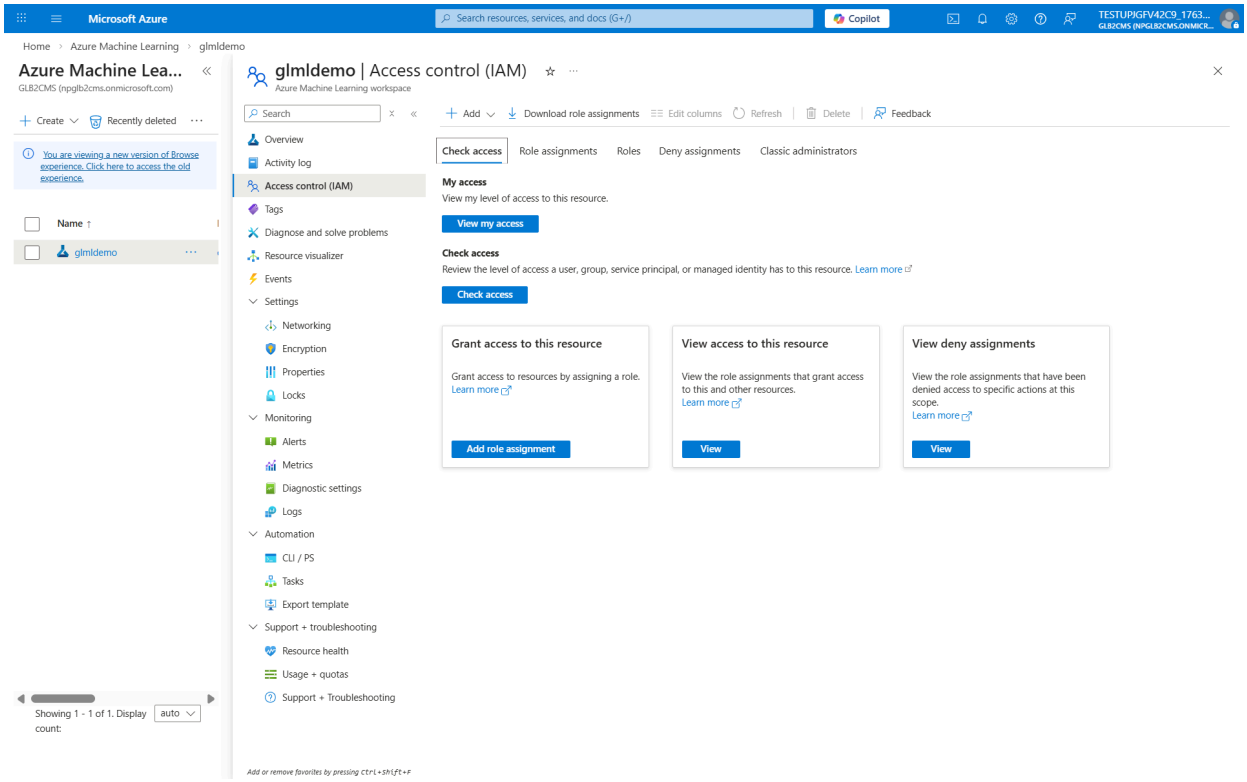
- Enable Identity:
    - Go to **App Service** > **Settings** > **Identity**.



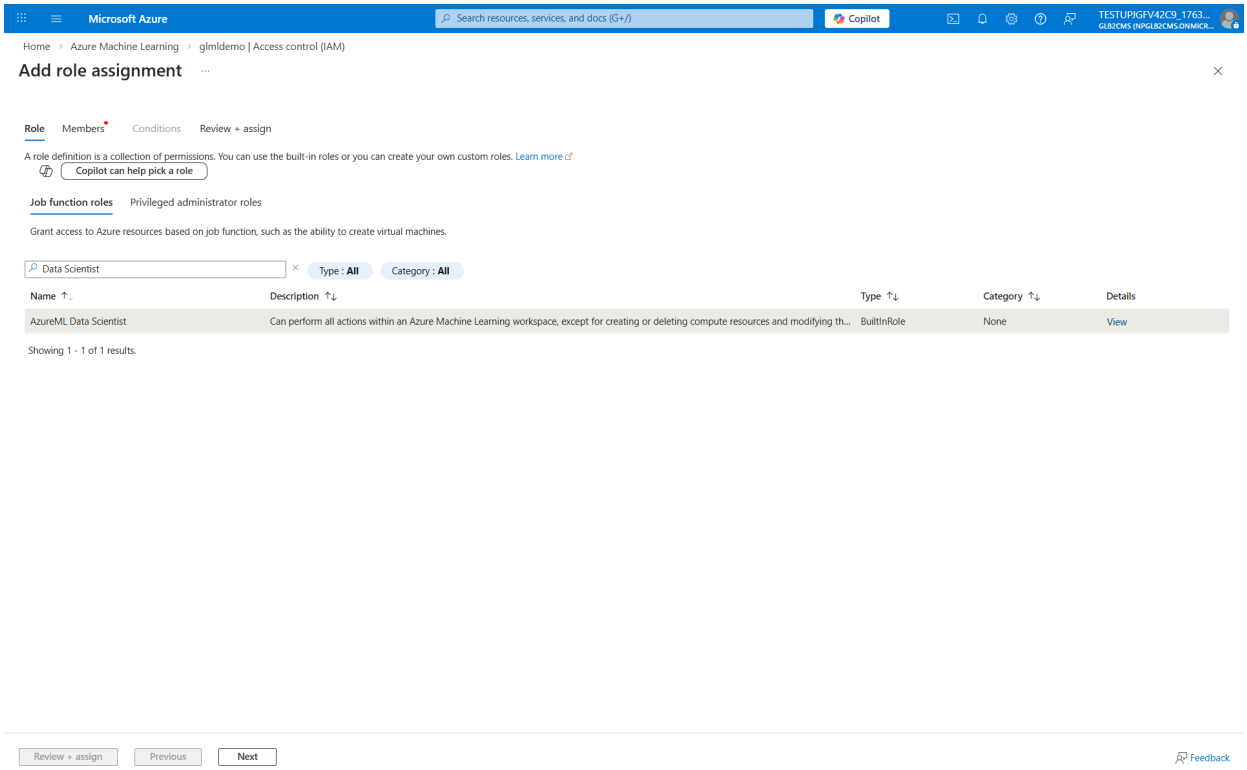
    - Set Status to **On** (System assigned) and click **Save**.



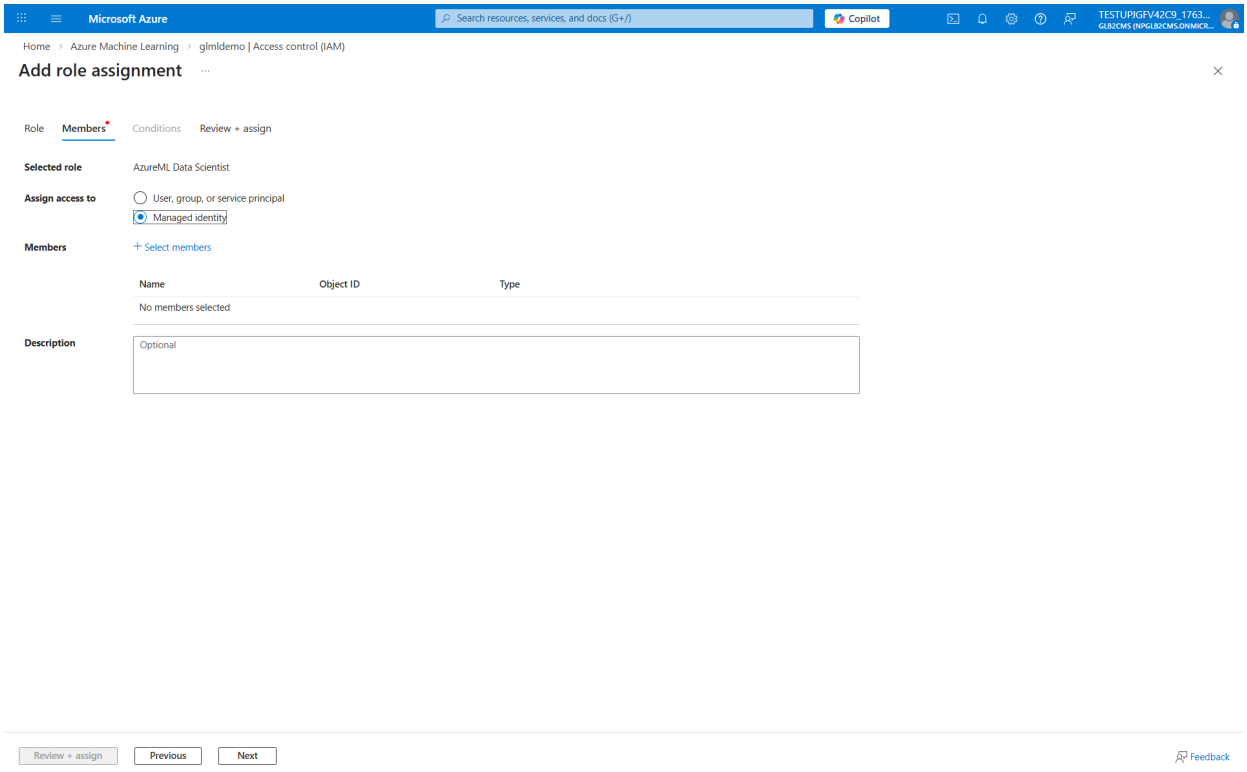
- Grant Permissions:
    - **Navigate to your Azure** Machine Learning Workspace.
    - Go to **Access control (IAM)** > **Add** > **Add role assignment**.



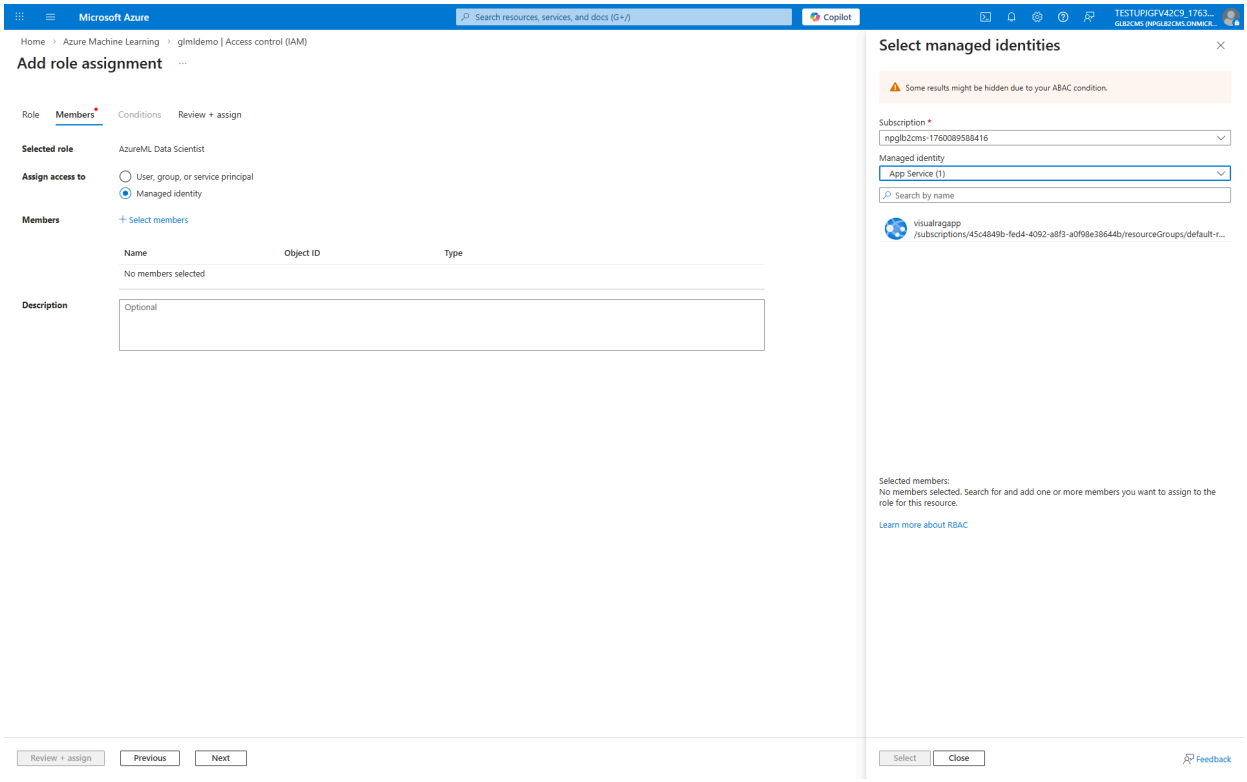
- Role: **AzureML Data Scientist** (allows logging and reading experiments).

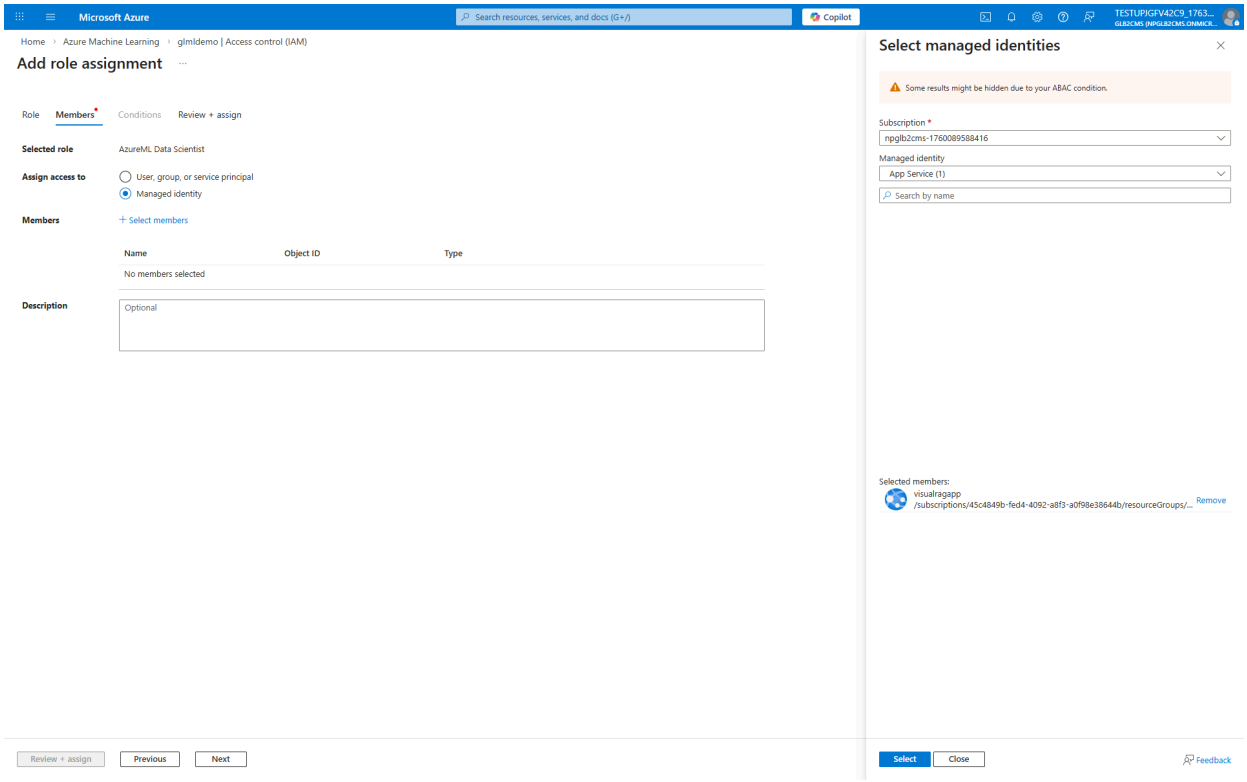


- Assign access to: Managed Identity.

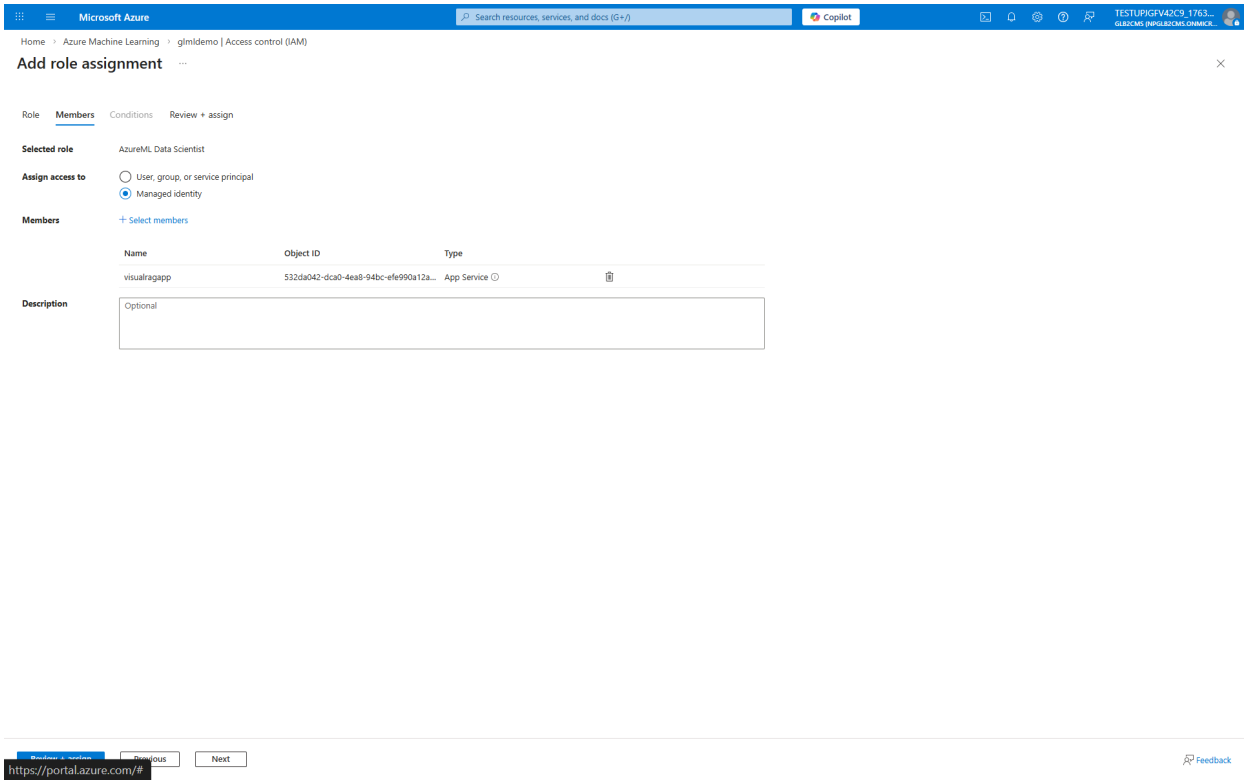


- Member: Select your specific **App Service instance**.'

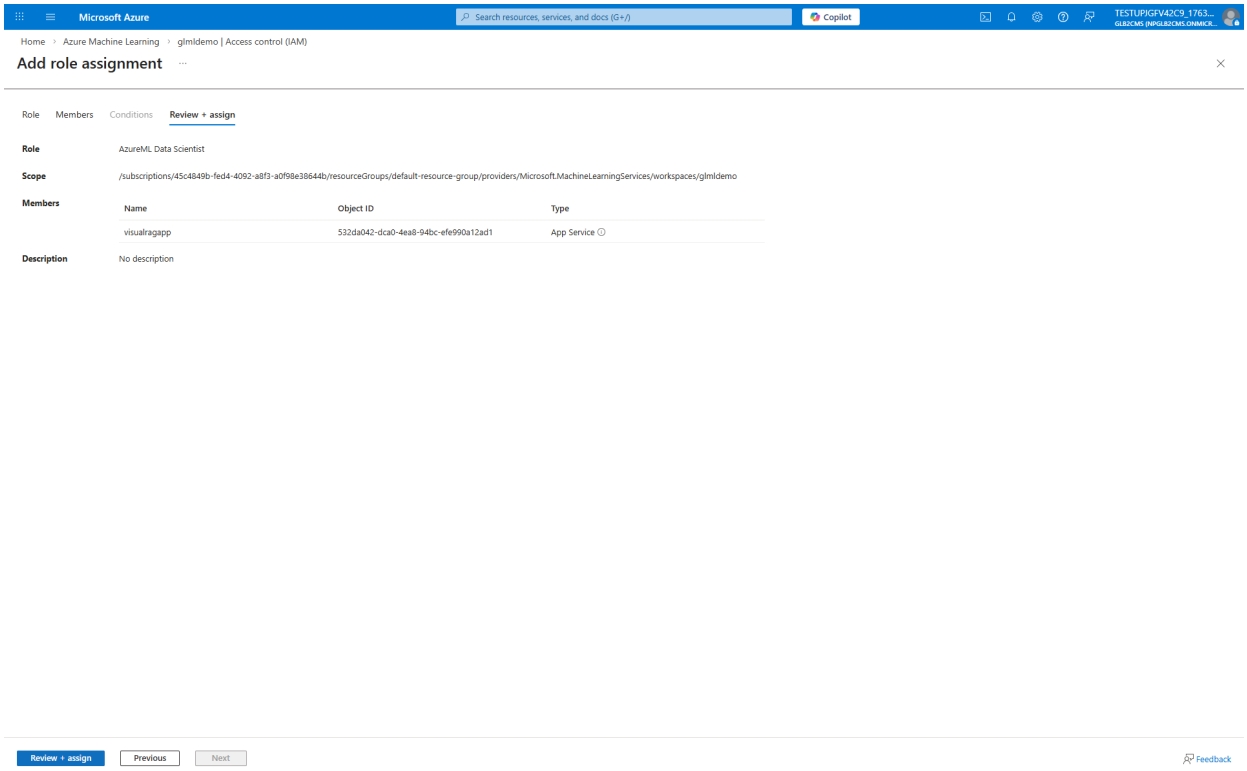




Click on **Next**

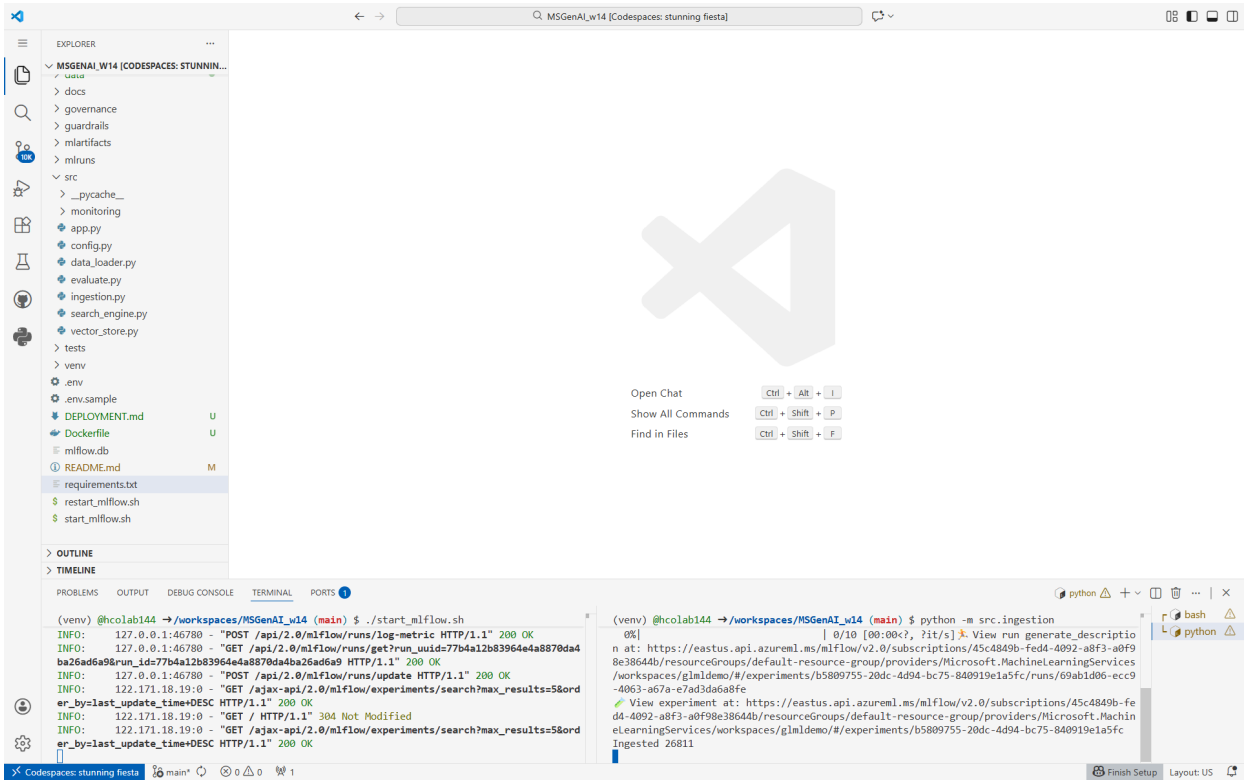


Click on **Review + assign**

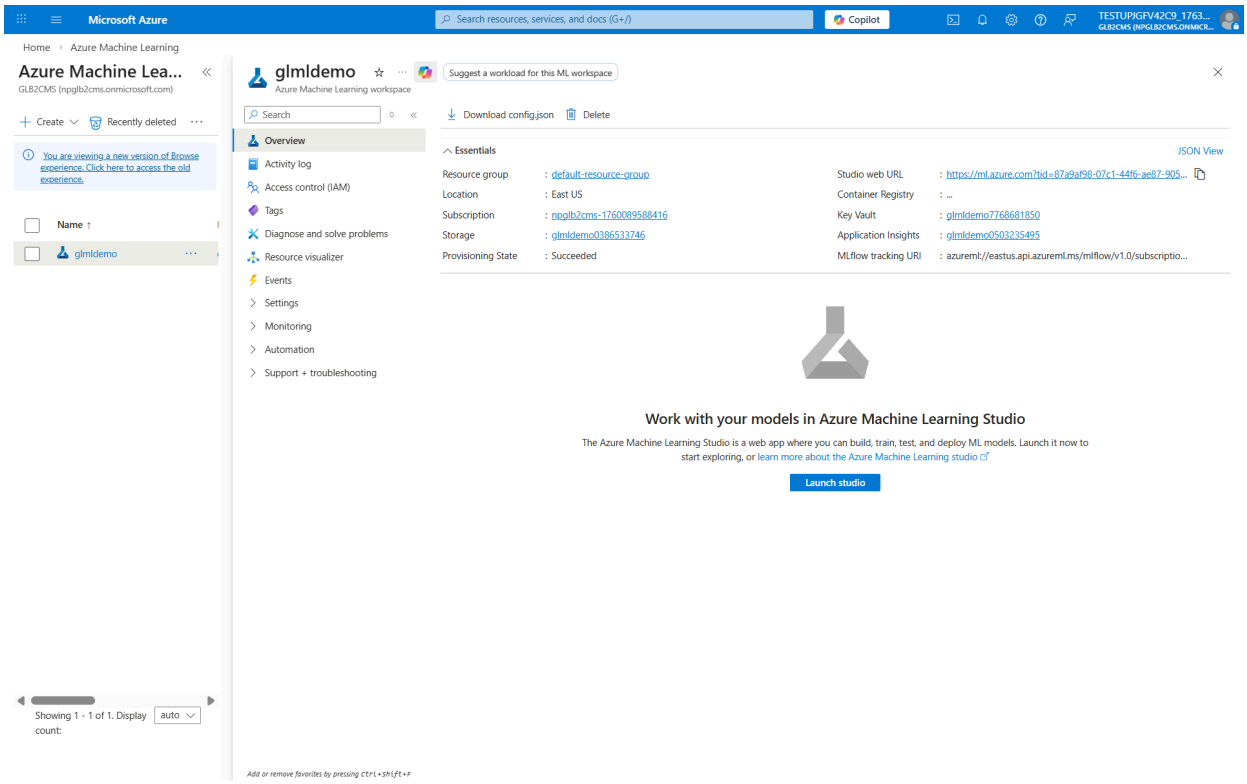


Tracing via Azure ML Workspace

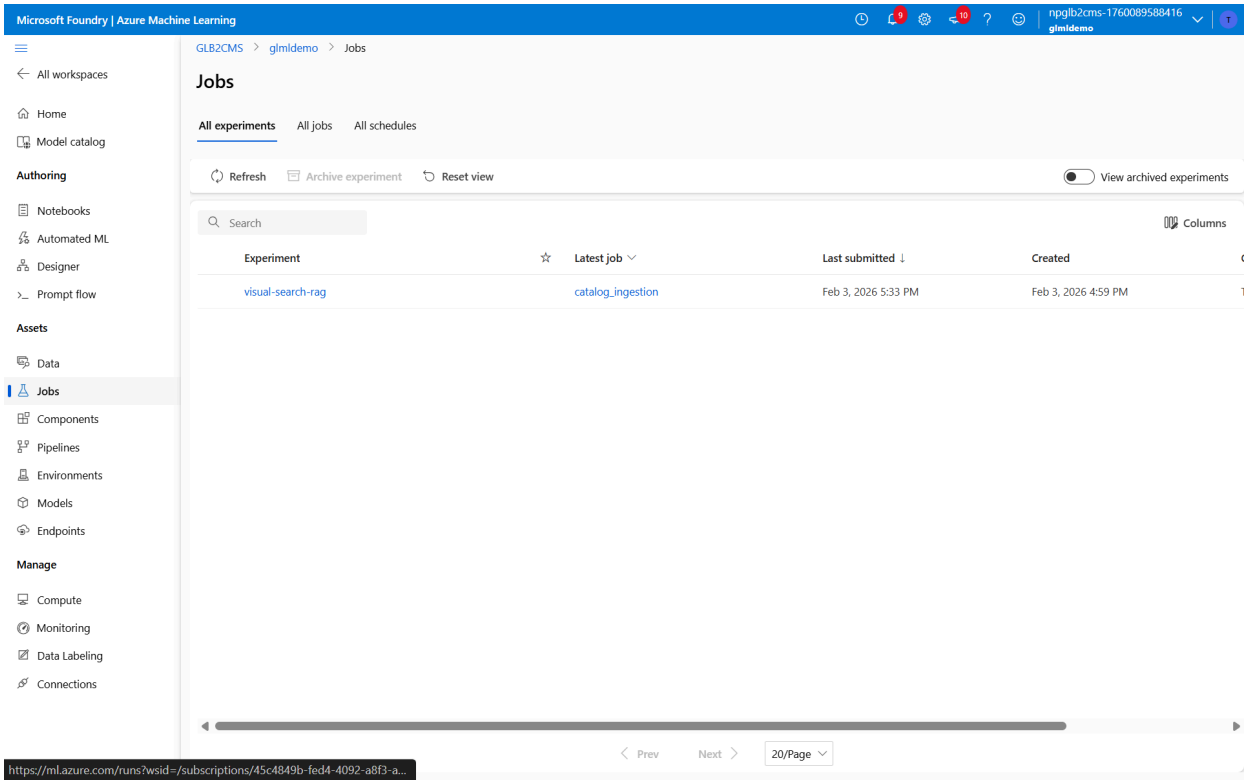
Run your script: python src/ingestion.py.



Open **Azure Machine Learning Studio**.

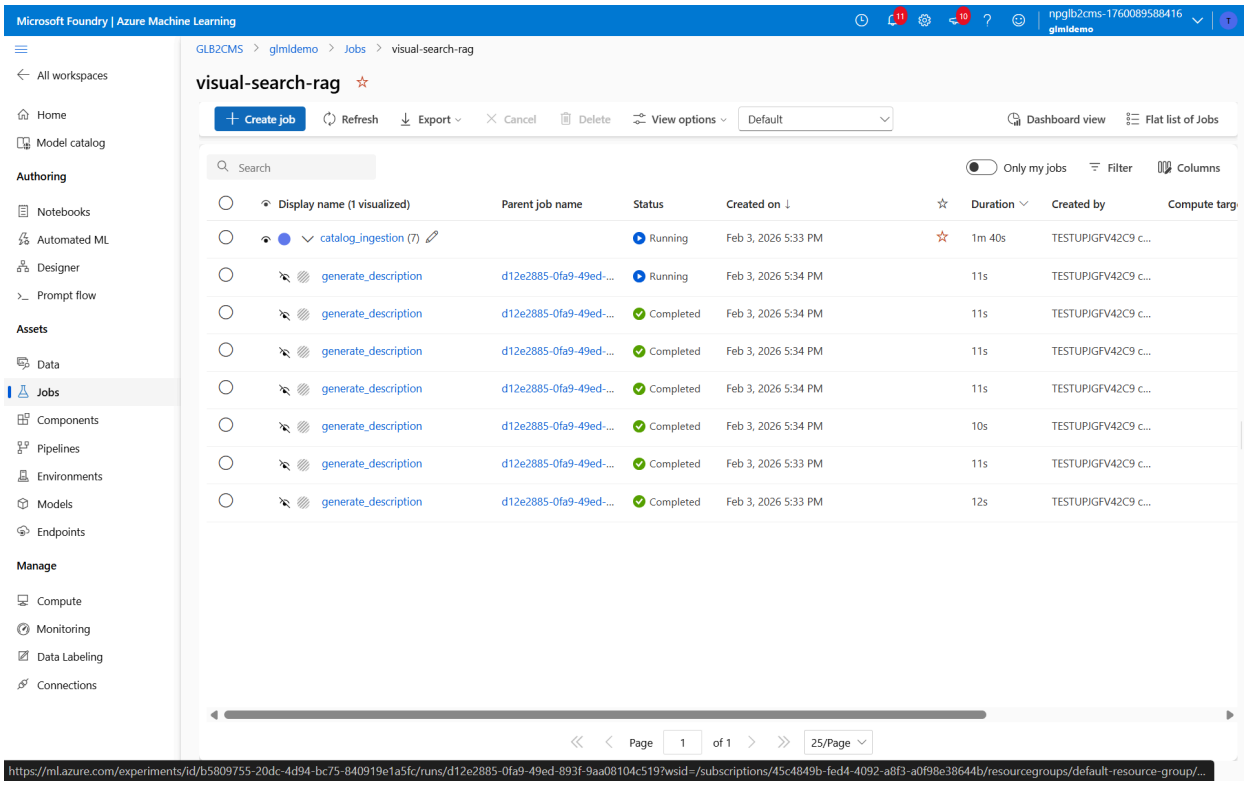


Navigate to the **Jobs** tab in the left sidebar

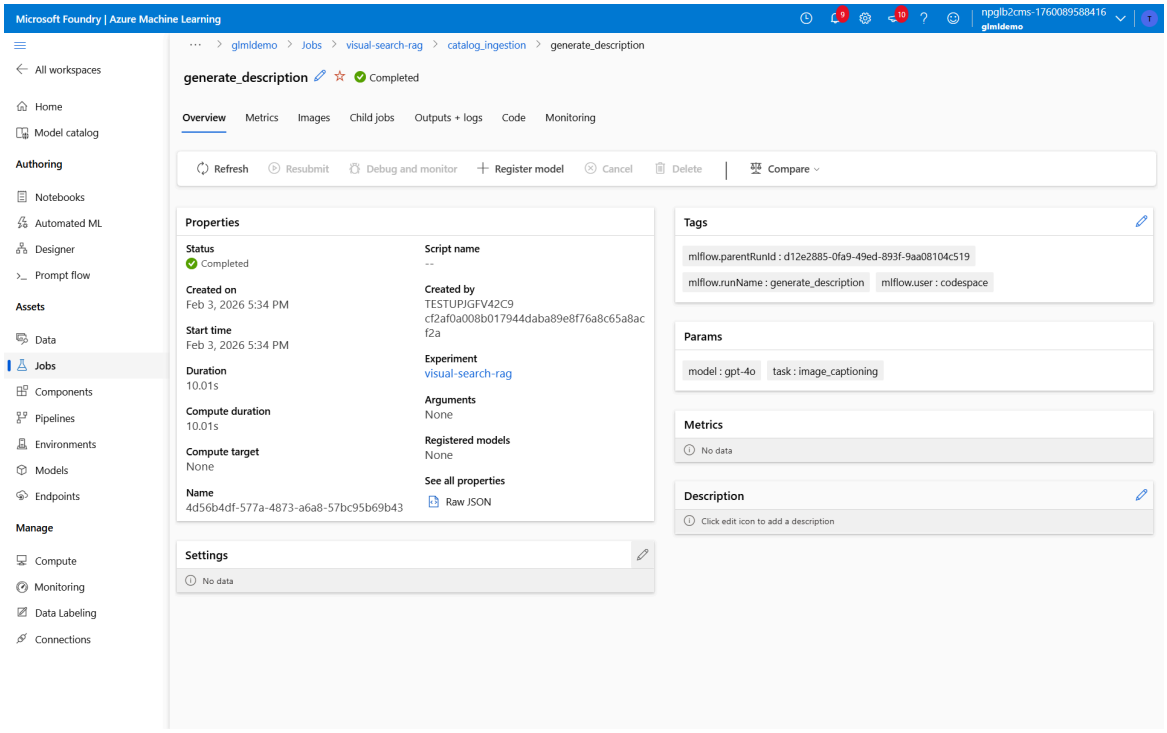


.

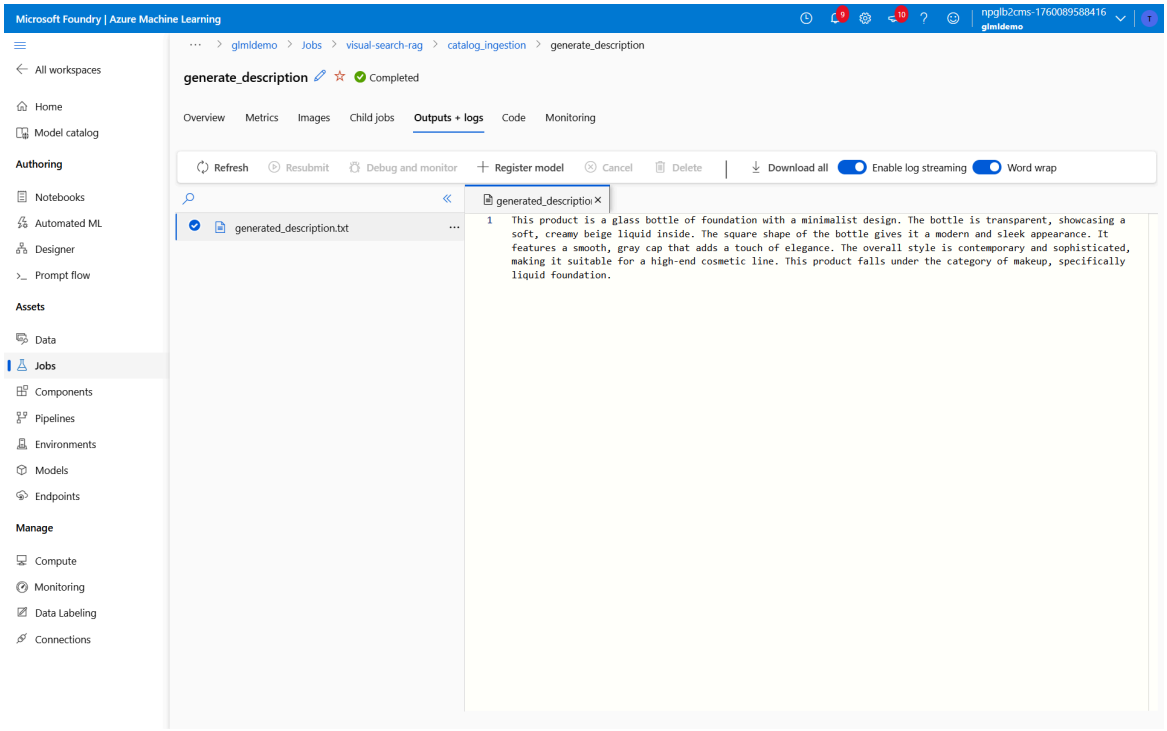
Select your experiment (e.g., visual-search-rag).



You will see your runs, metrics, and traces listed there.



The Outputs + logs tab provides the Inputs - Output pair for each LLM Chain.

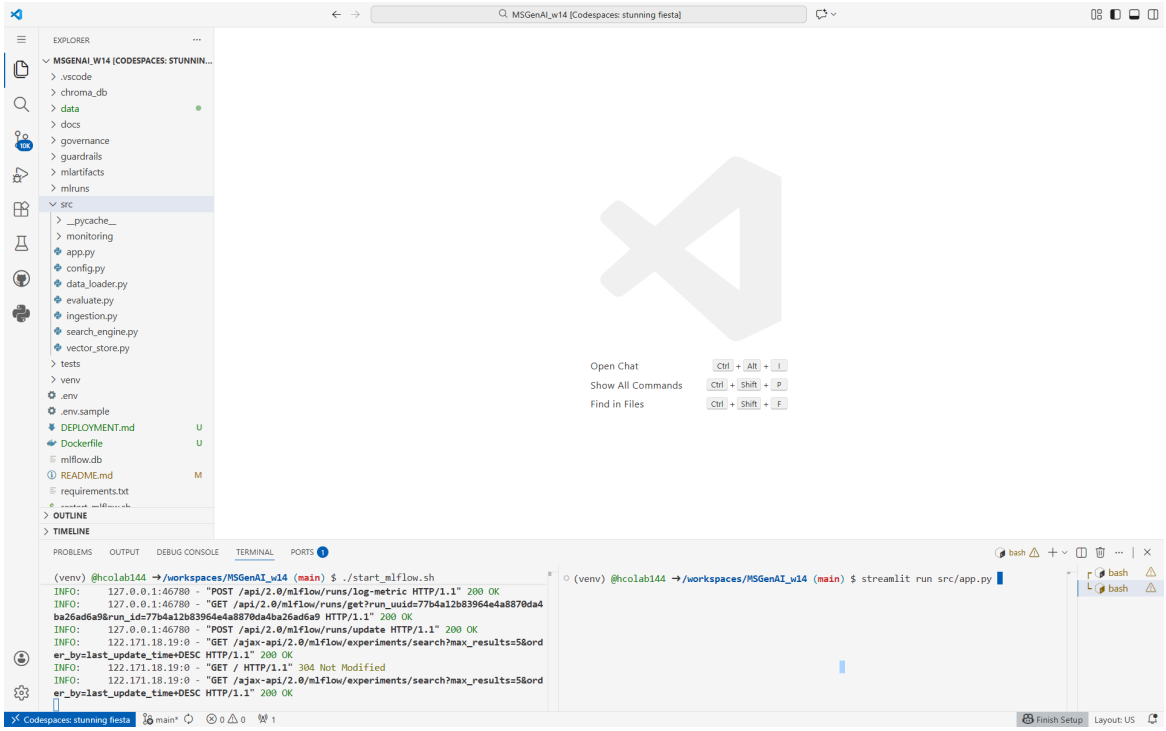


Streamlit Application

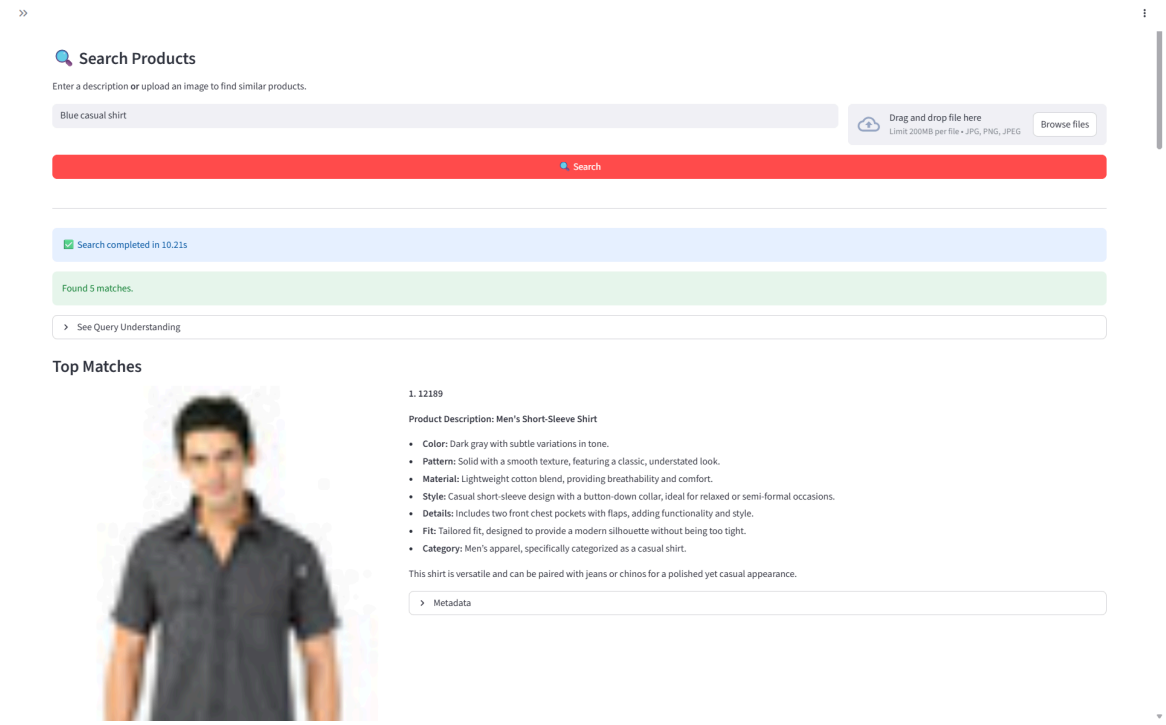
To run the streamlit application, run the following command

None

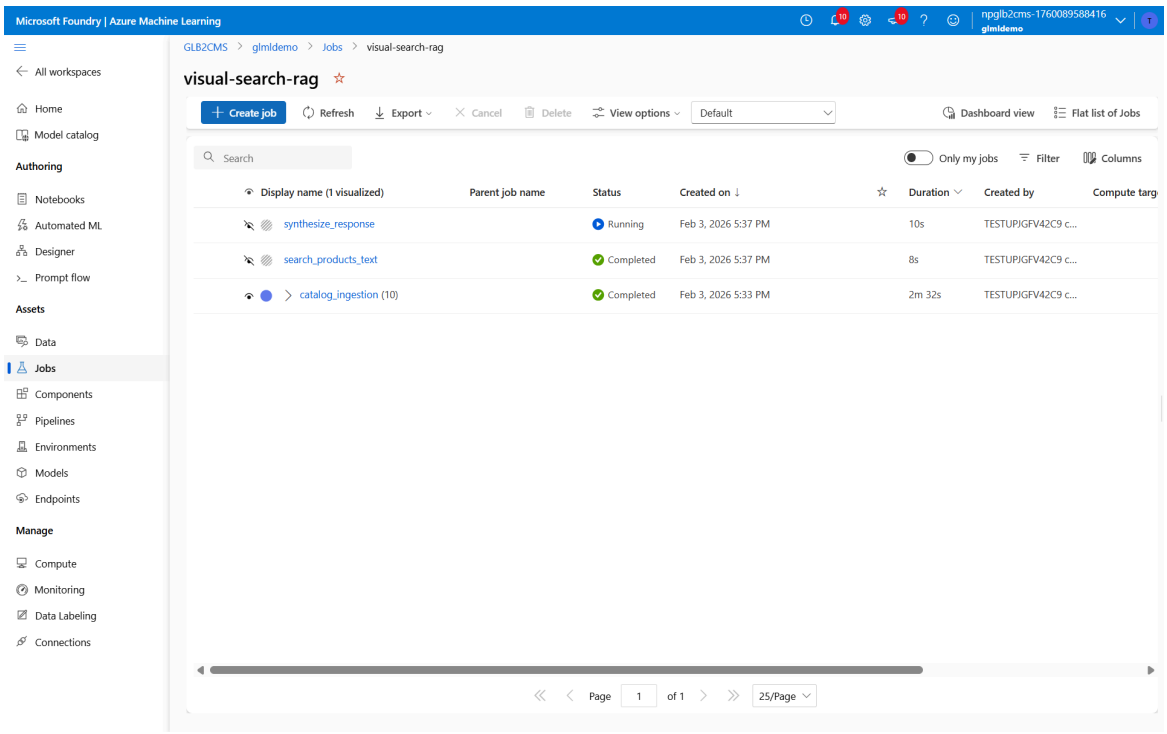
```
streamlit run src/app.py
```



The streamlit application shows the results for the user query. “Blue casual shirt”



The tracking server keeps track of the LLM calls



Note: There is occasionally a 1–2 minute propagation delay for logs and GenAI traces to appear in the Azure ML Studio UI compared to a local MLflow dashboard.

🛡️ Governance & Guardrails

The system uses a layered safety approach located in governance/ and guardrails/.

- **Governance Gate** (governance.governance_gate): Orchestrates checks.
- **Safety Validator**: Blocks harmful content.
- **Compliance Checker**: Redacts PII.

See Governance README for details.

📊 Monitoring

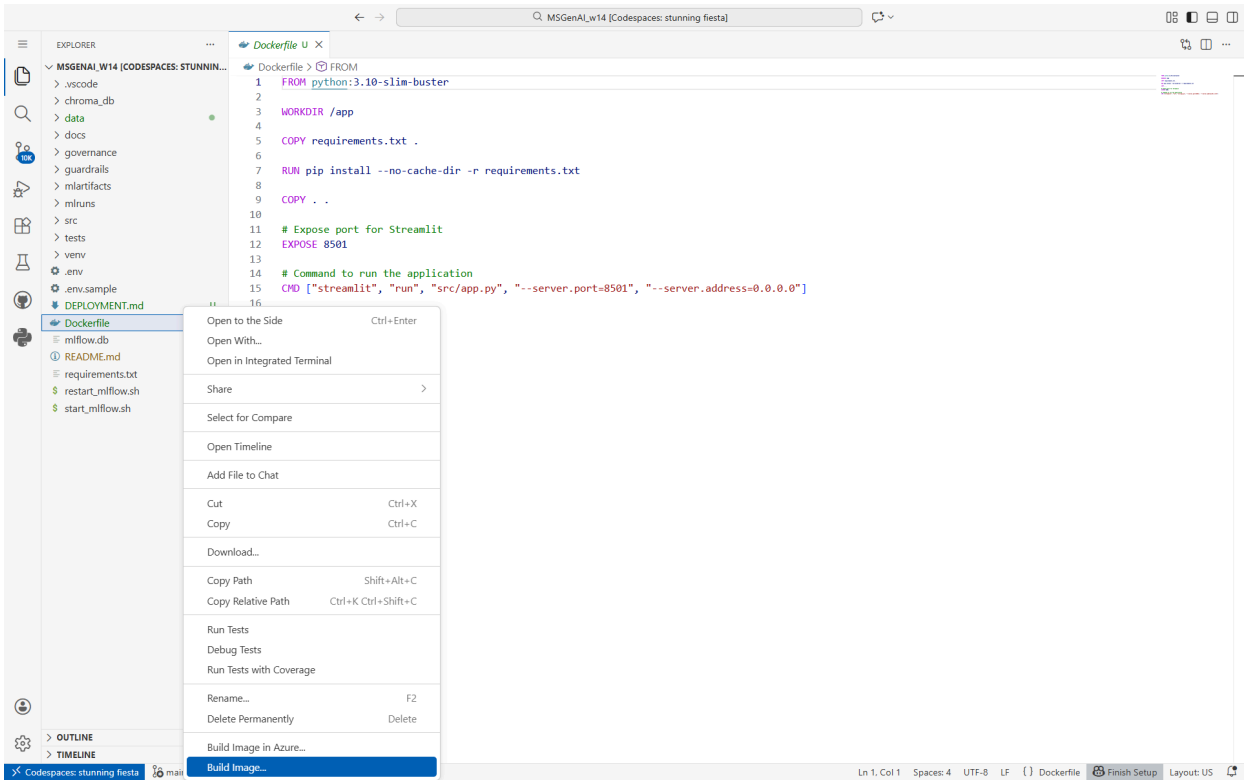
Unified monitoring via monitoring/.

- **Logger**: Structured JSON logging.
- **Metrics**: Token usage, latency, cost.
- **Traces**: Distributed tracing via OpenTelemetry.

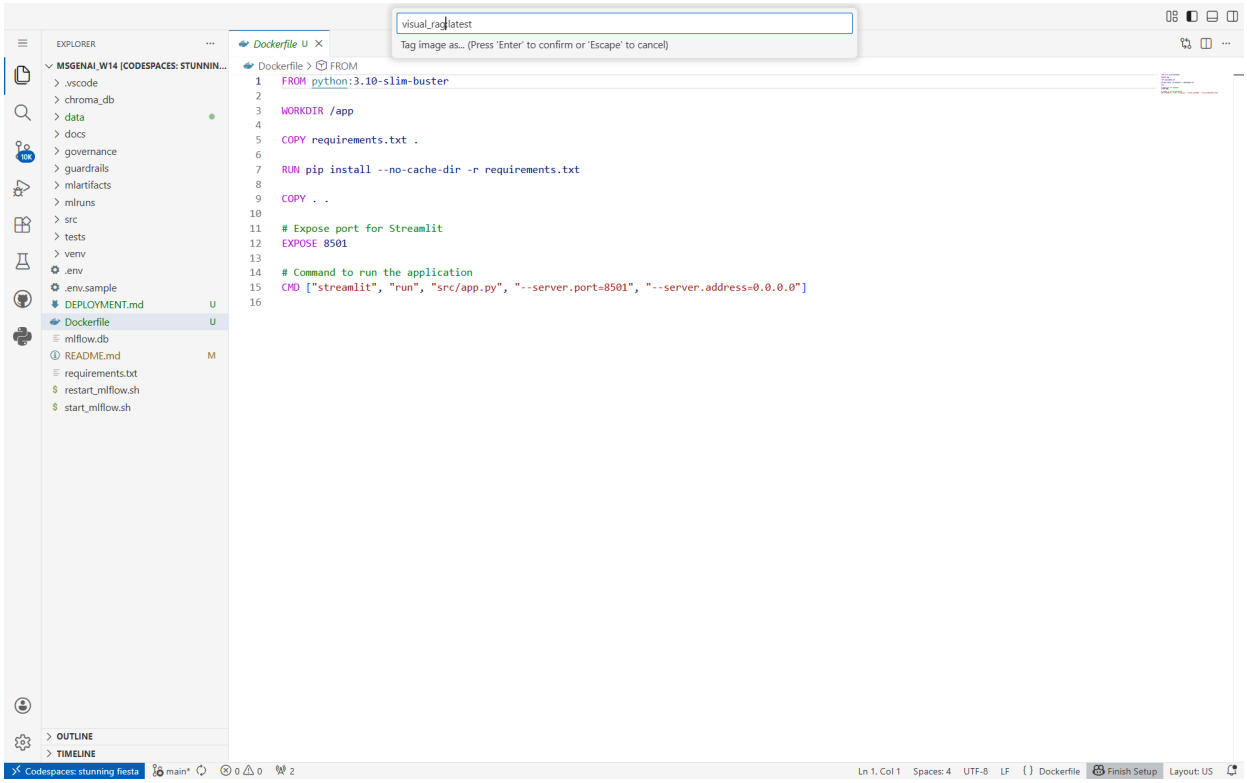
See Monitoring README for details.

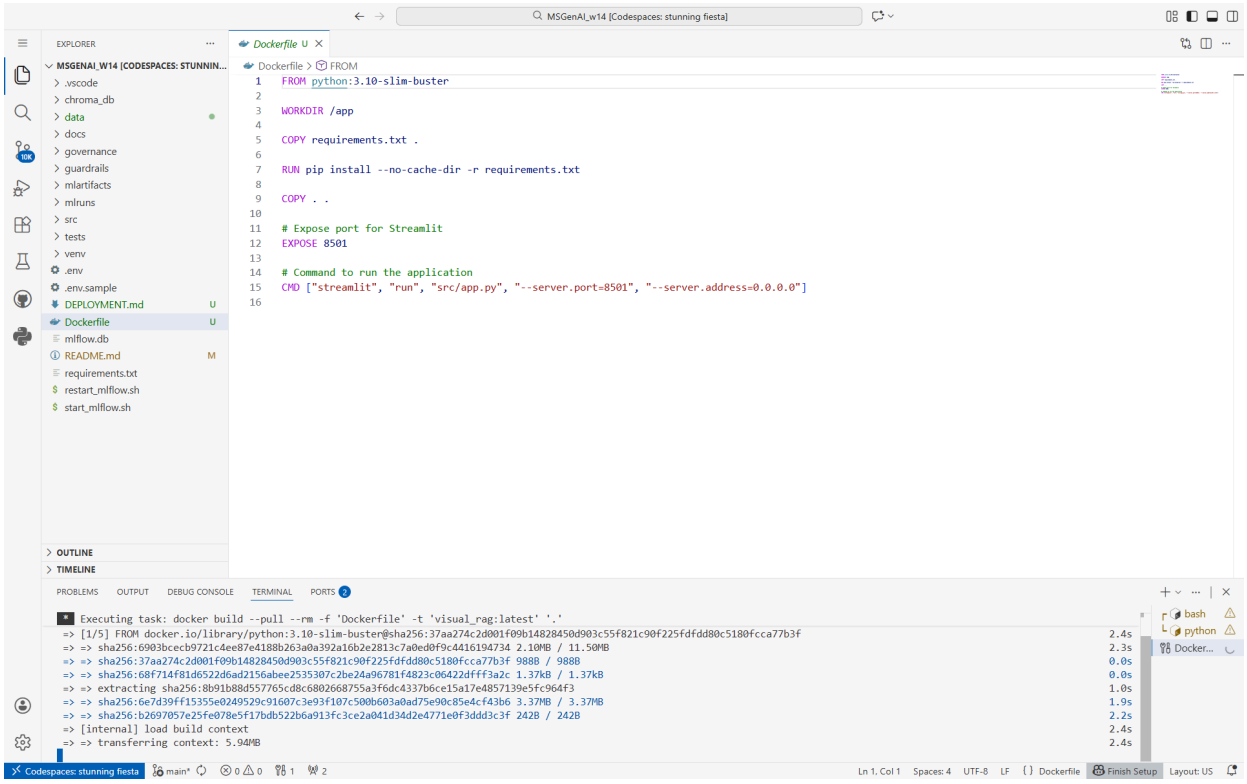
Build Docker Image

Select the **Docker Image** and click on **Build Image**

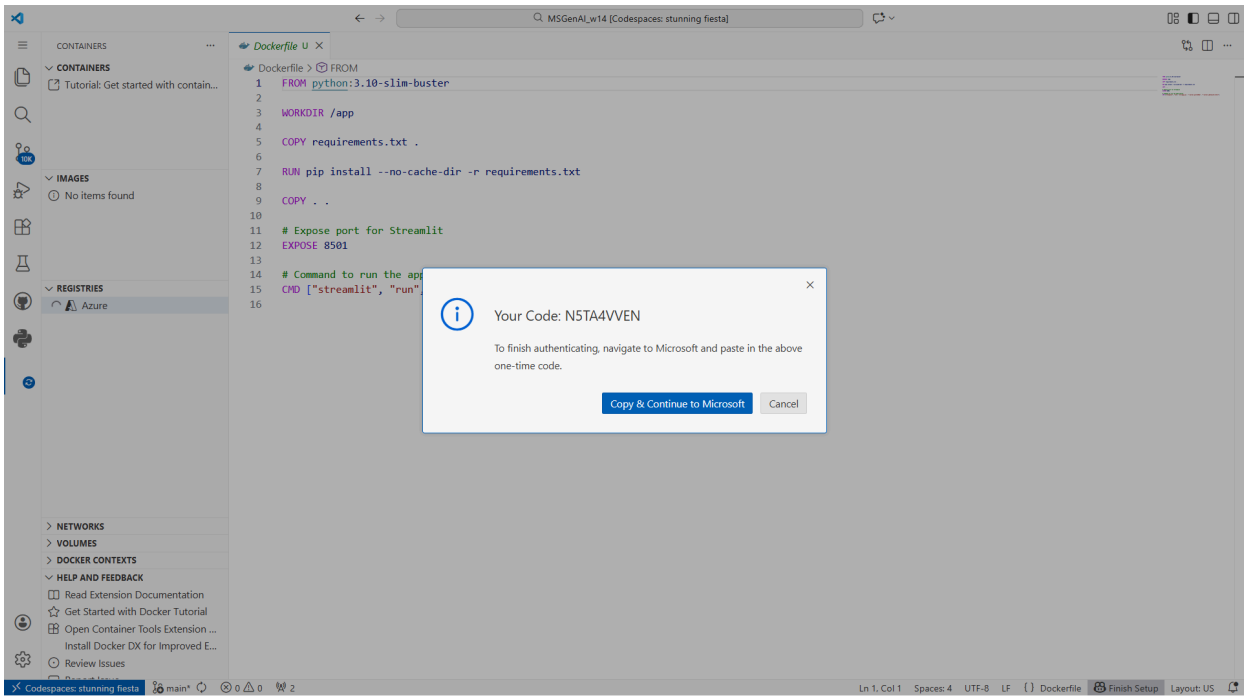


Enter the Image Tag and click Enter.

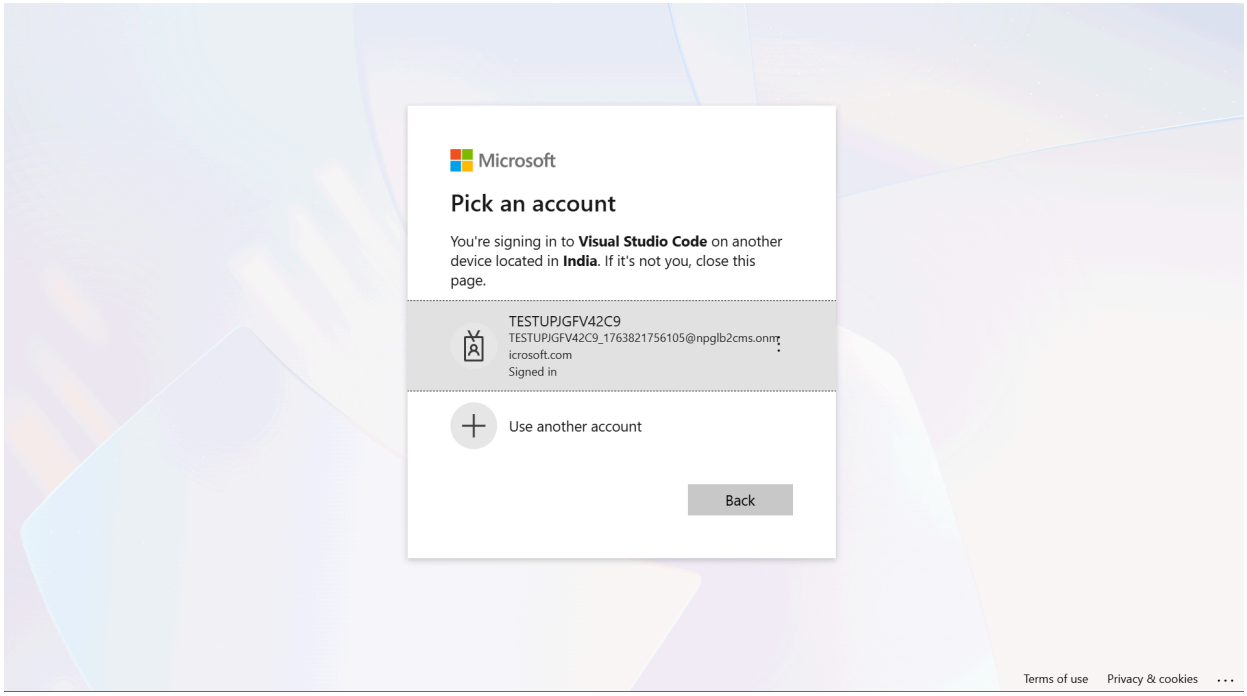


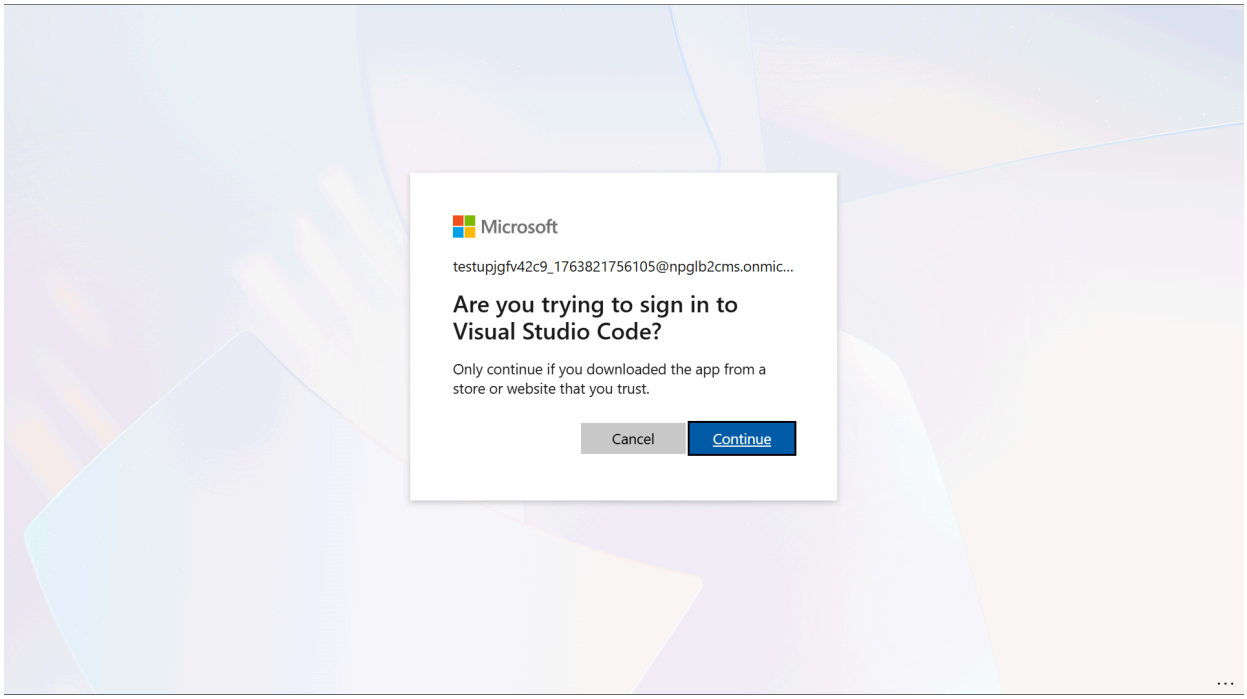


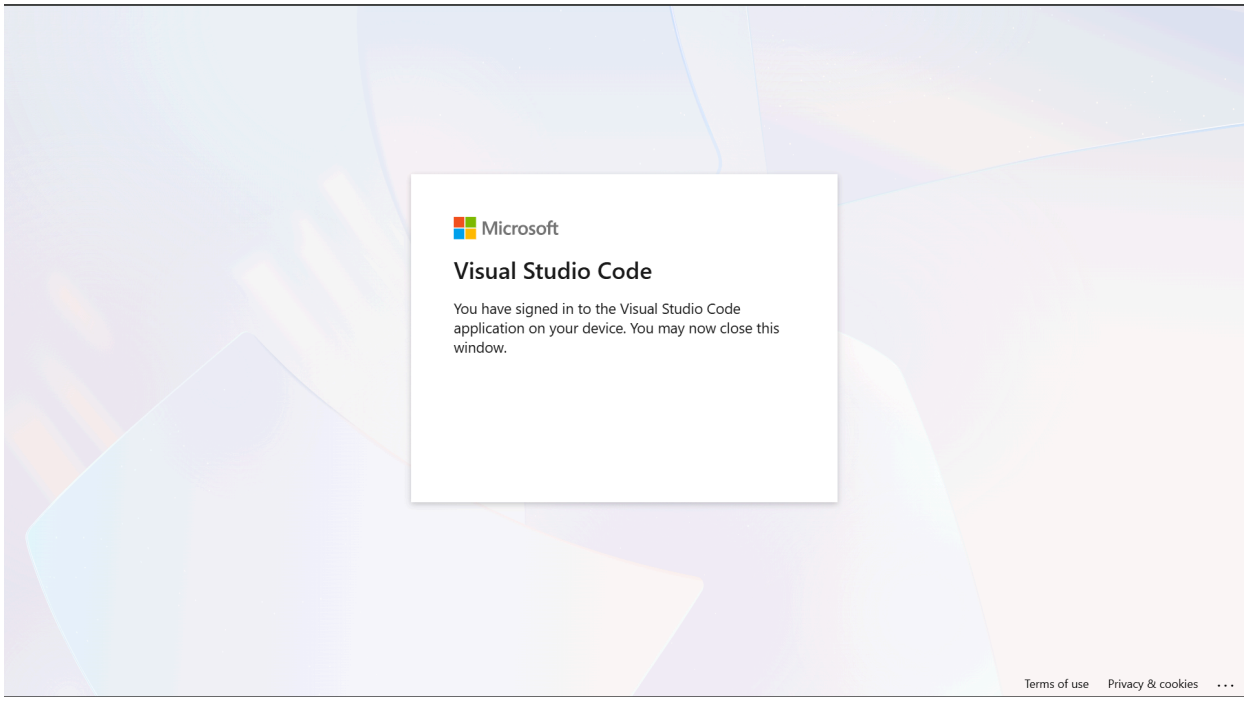
Once the Docker image is built, navigate to the **Container Tools** extension in VS Code and click on **Azure**.
This will prompt you to login using your **Azure account**.

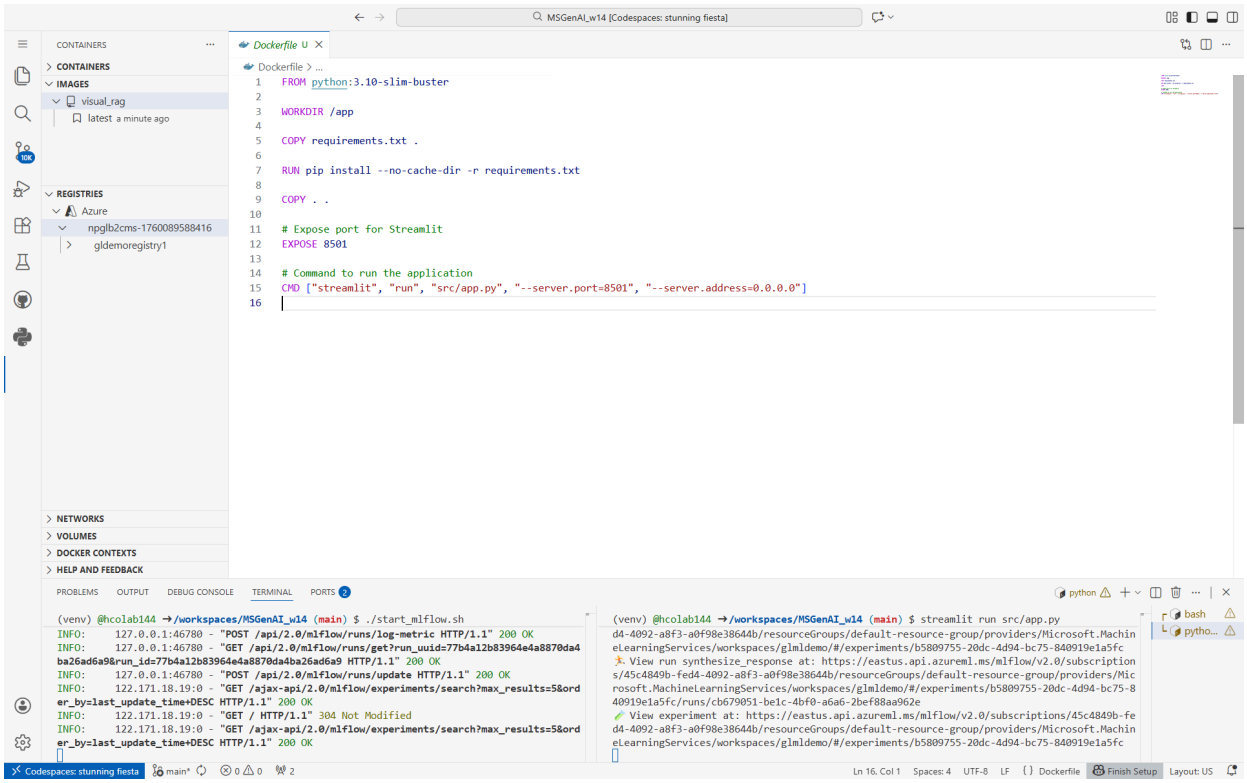


Login via your Azure account.

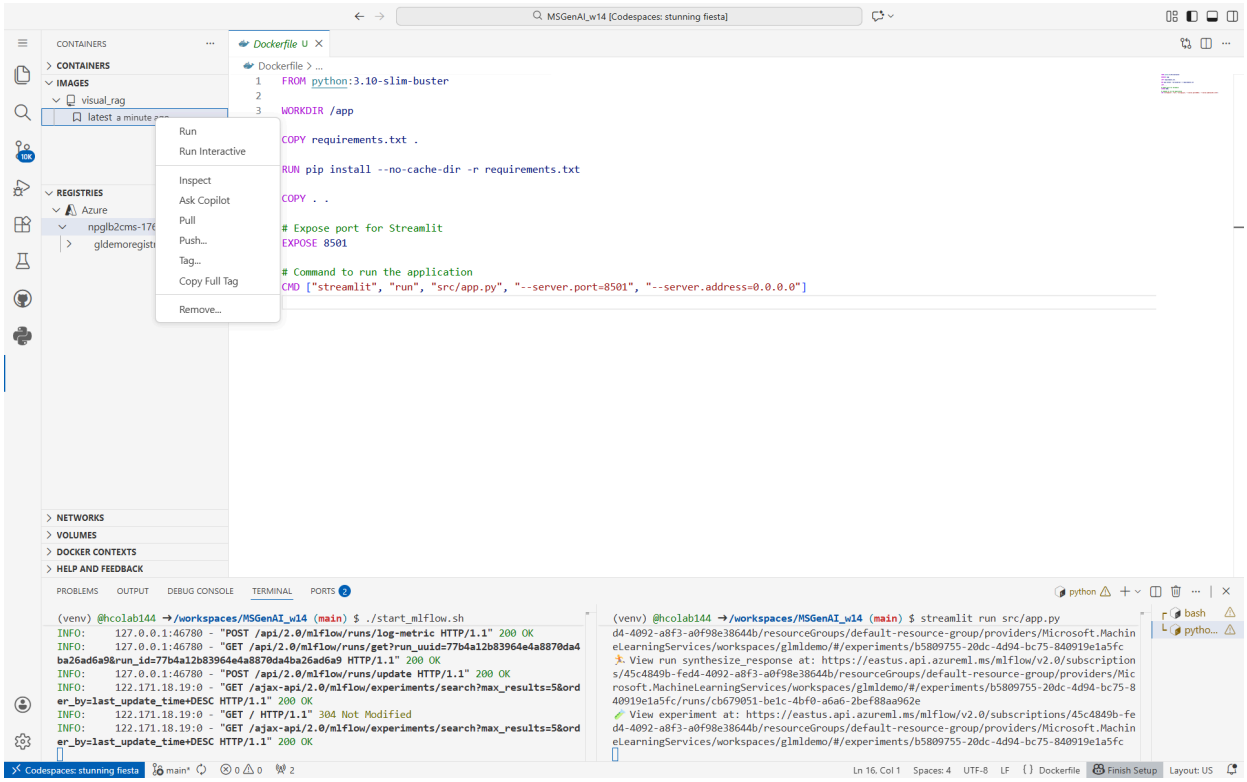








Click on the Docker Image and click on **Push**



NOTE: The above steps assume that you've already created the Azure Container Registry. The steps for creating a registry can be found in Week 9 and Week 13 MLS sections

---

Azure App Service

NOTE: Once the Docker image has been pushed to the ACR, the next step is to deploy the Docker image using a service such as **Azure App Services.**
You may refer to the MLS content of Week 9, Week 13 for the step-by-step instructions on how to create an Azure App Service and deploy the application using Docker Image via Azure Container Registry (ACR)

1. Navigate to your **App Service** in the Azure Portal.
2. Go to **Settings > Configuration** (or **Environment variables** in newer interfaces).
3. Click **+ Add** to create a new setting.

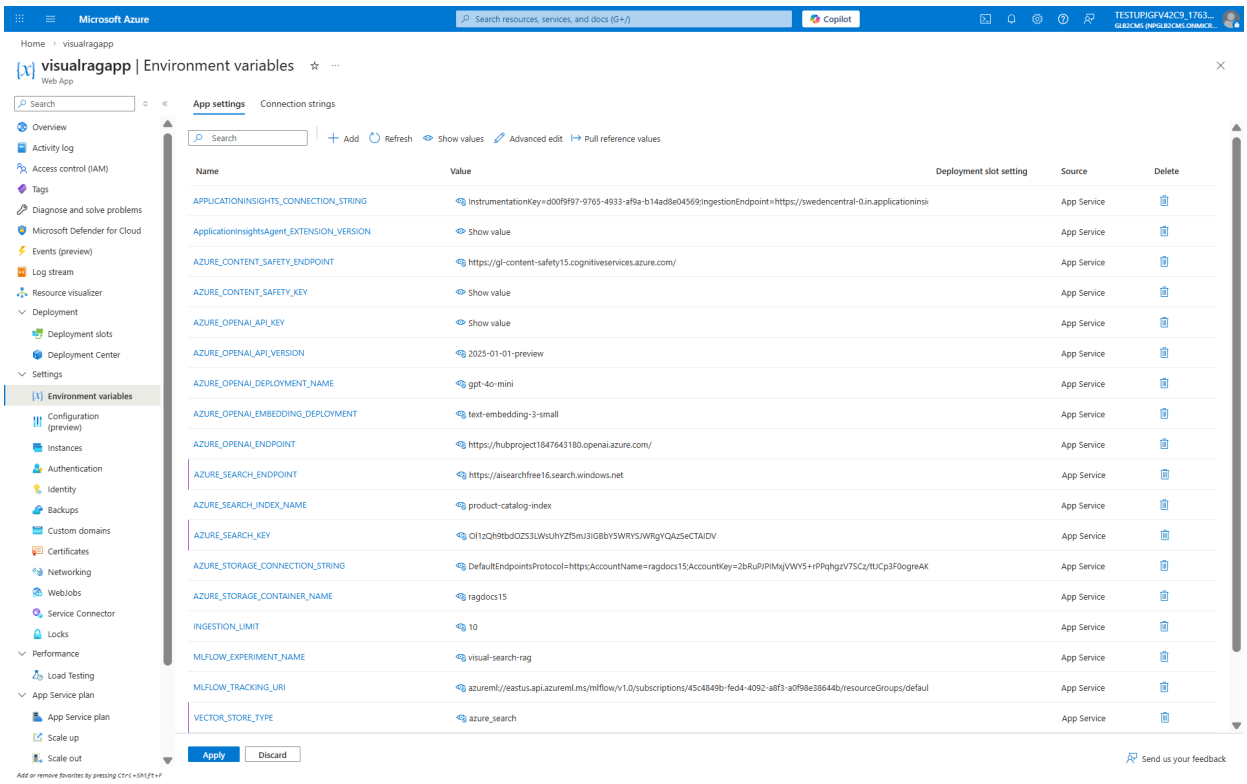
For instance to add MLFLOW tracking URI, click Add and set the following:

- **Name:** MLFLOW_TRACKING_URI
- **Value:** <Your Azure ML Tracking URI>

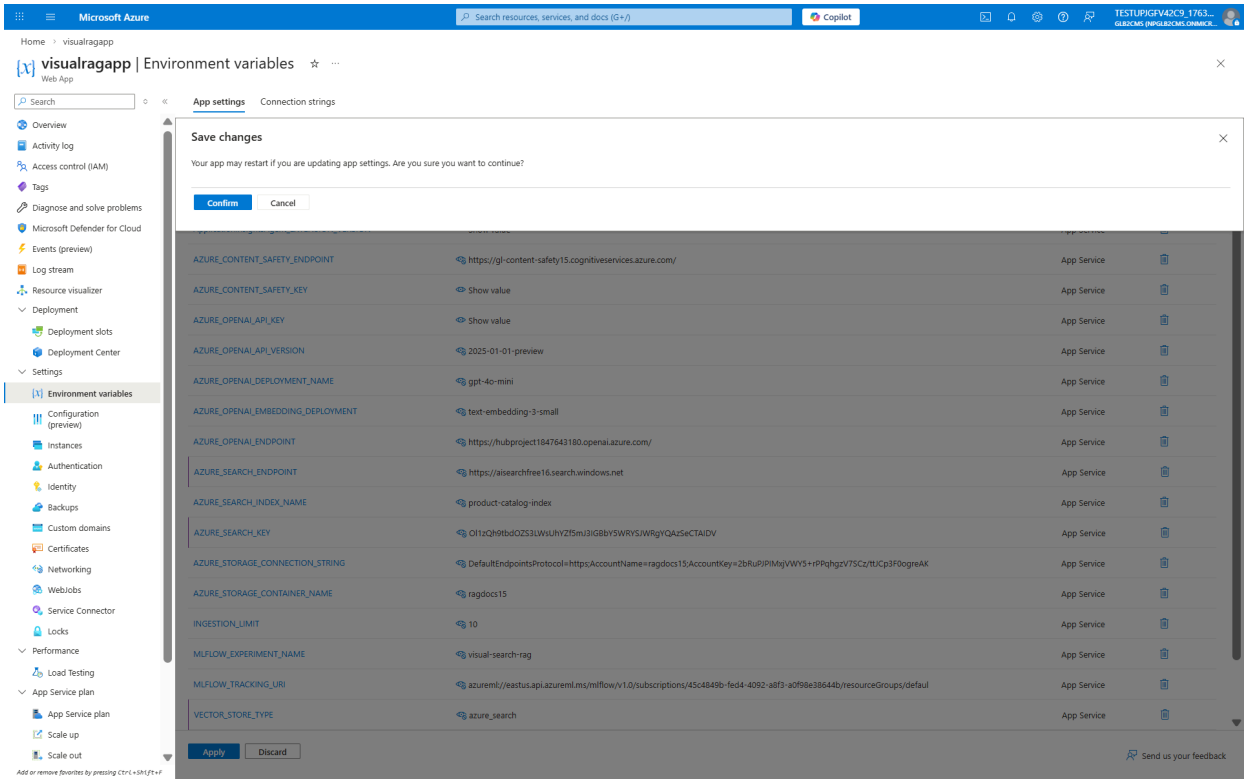
4. Click **Save**.

Enter all the environment variables required for the application and click **Apply**

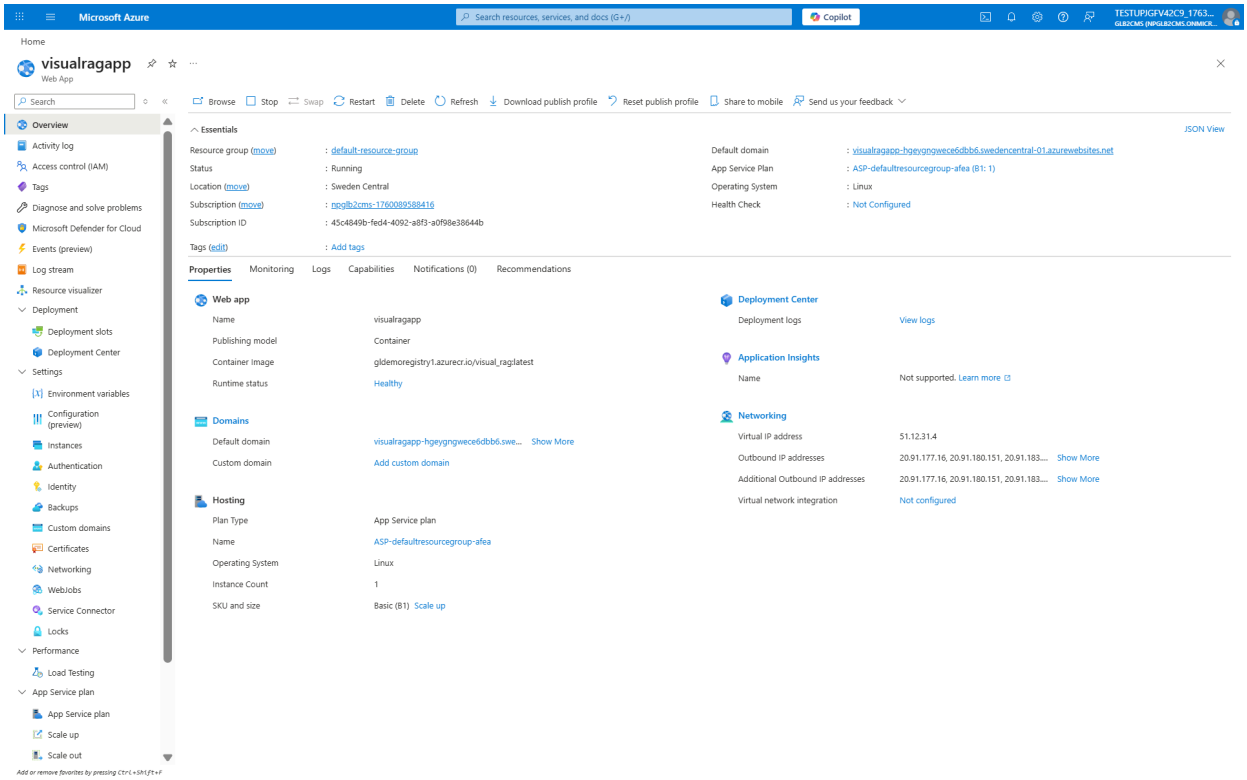
NOTE: Make sure the variable **VECTOR_STORE_TYPE** is **azure_search**



Save the changes and click **Confirm**.



Click on the **URL** defined in the **Default domain**.

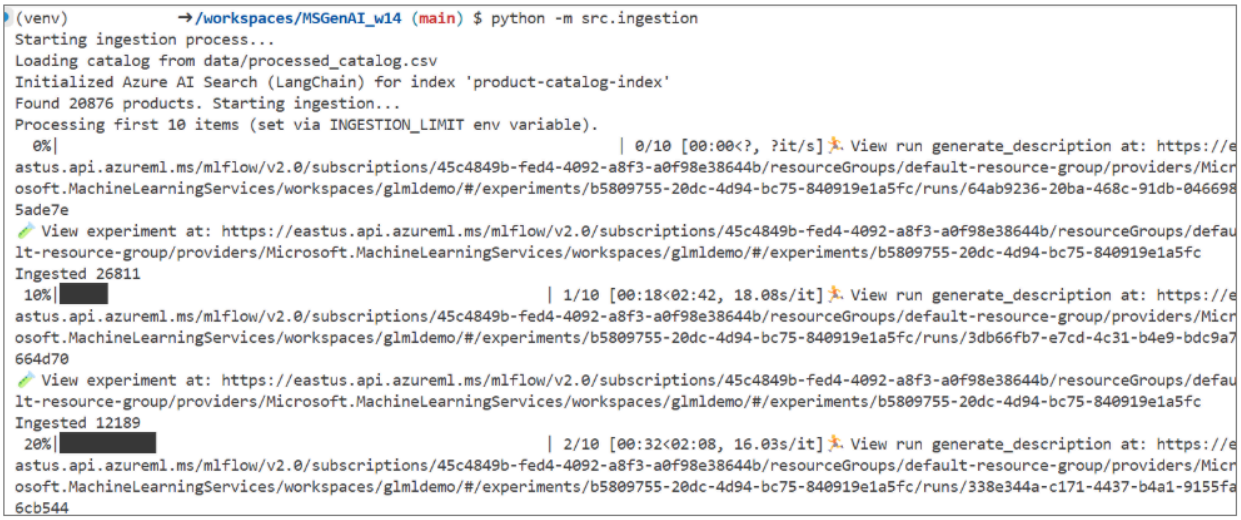


NOTE:
For production purposes, we will ingest the images using Azure AI search.
Update the .env file to **azure_search**

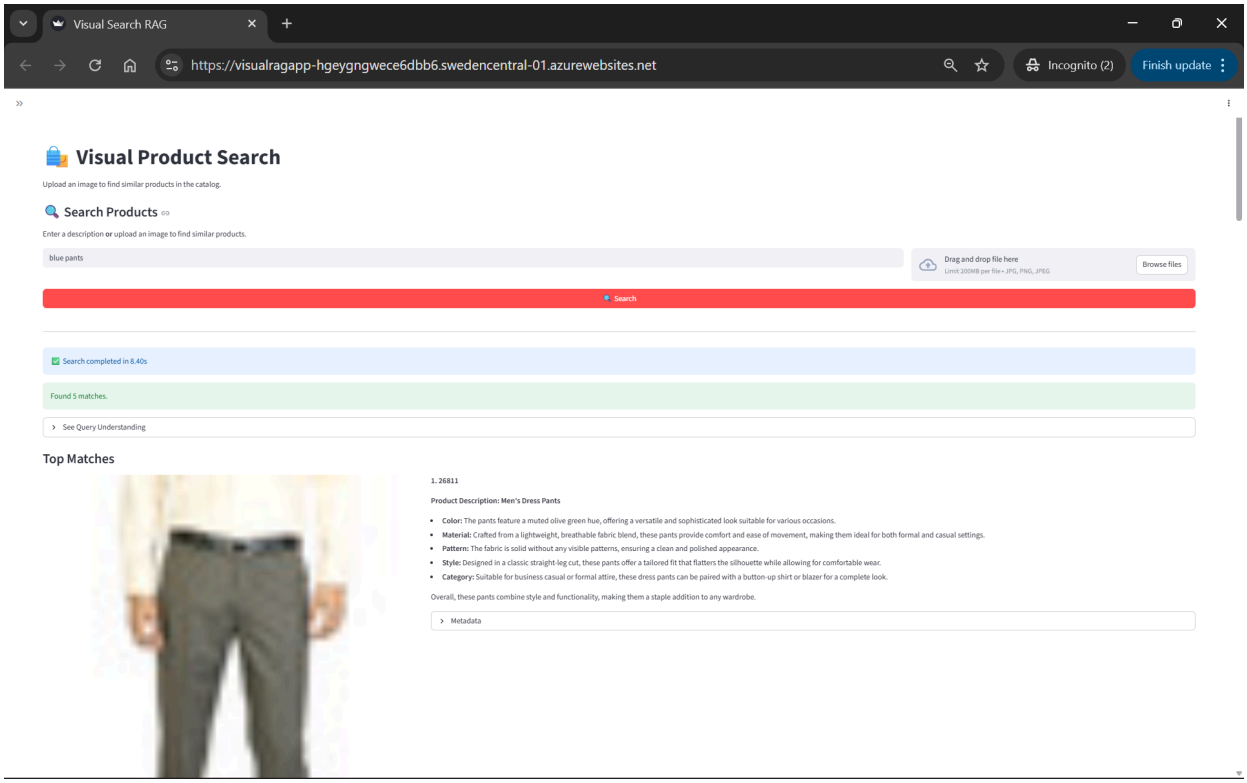
```
None
.env file
# Vector Store Selection
# Options: "chroma" (Local) or "azure_search" (Production)
VECTOR_STORE_TYPE=azure_search
```

Run the following command to populate the search index in Azure AI Search before searching in the deployed application.

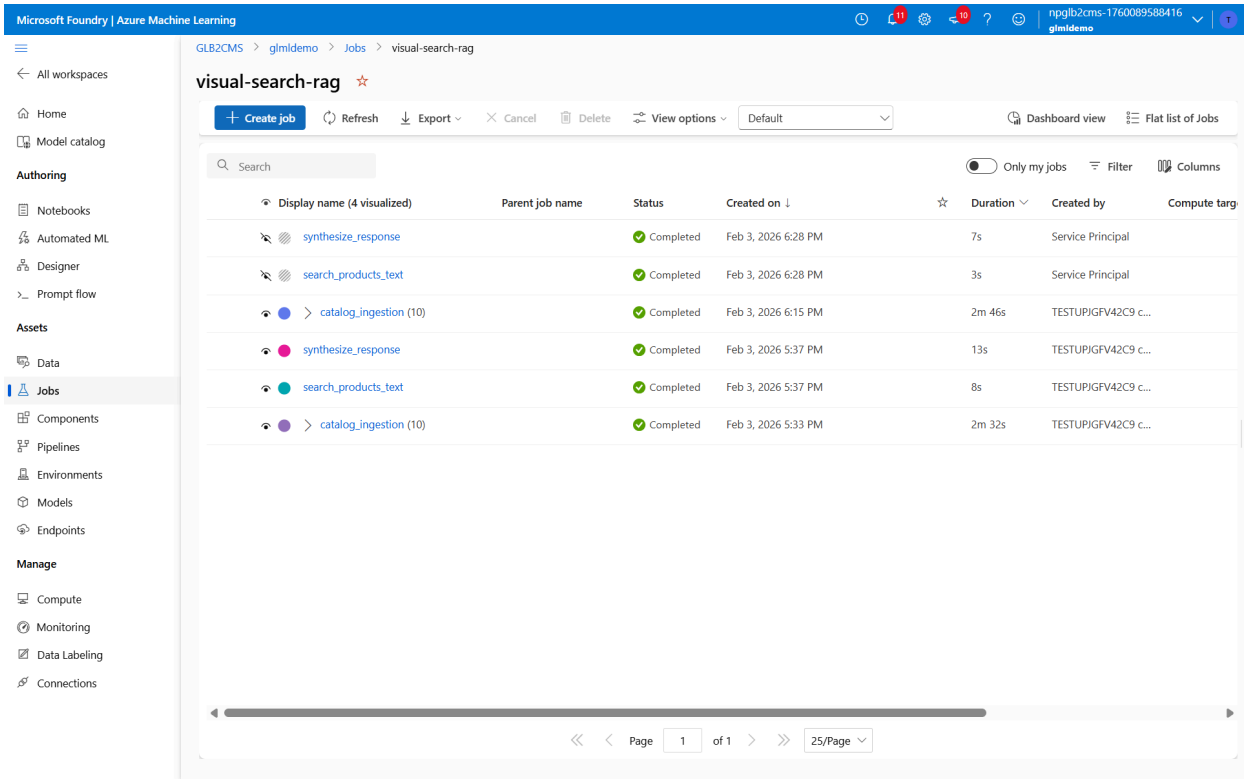
```
None
python -m src.ingestion
```



In the deployed application, enter the search query and press **Search**.
The search results are displayed in the sections below.



The tracking server in ML Studio automatically populates the **jobs** and the **corresponding results**.



Monitoring

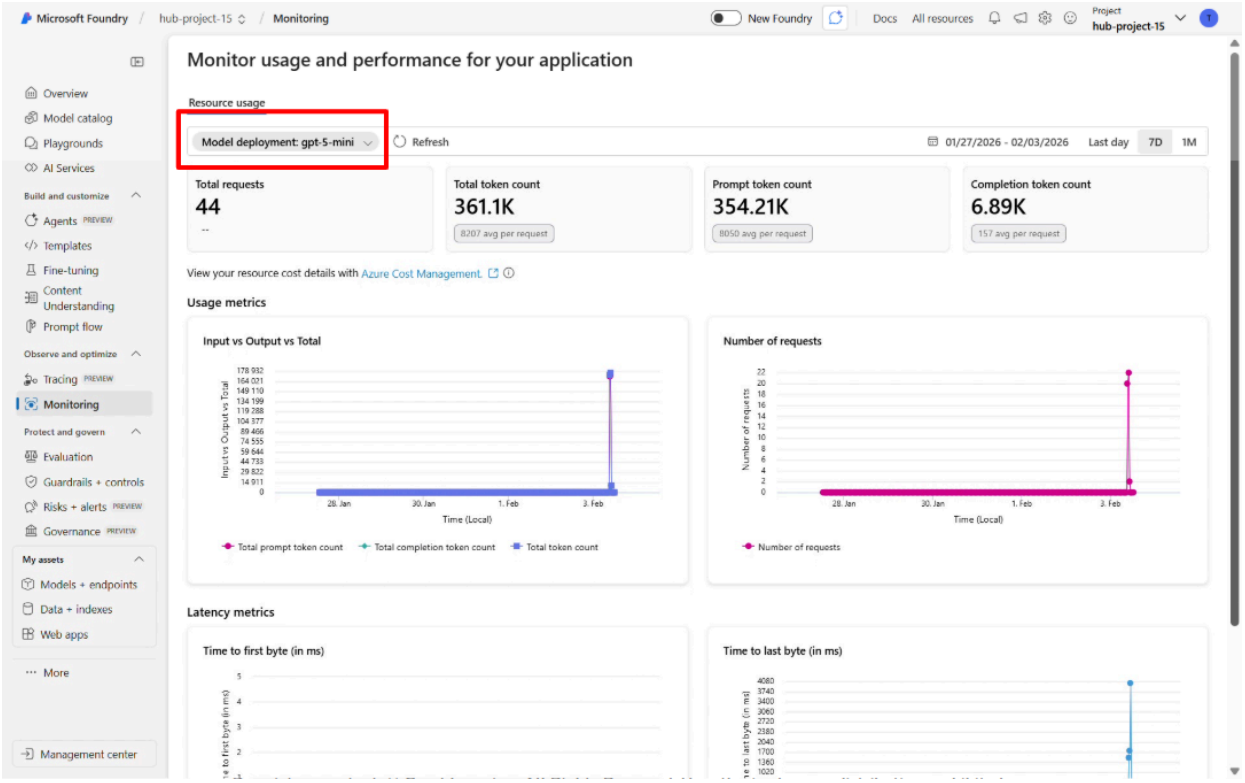
Monitoring of LLM applications provides the following benefits:

- Detection of "Model Drift" and "Knowledge Decay"
- Economic and Performance Visibility (Cost & Latency)
- Regulatory Compliance and Auditability
- Closing the "Feedback Loop" (Continuous Improvement)
- Safety and Guardrail Enforcement

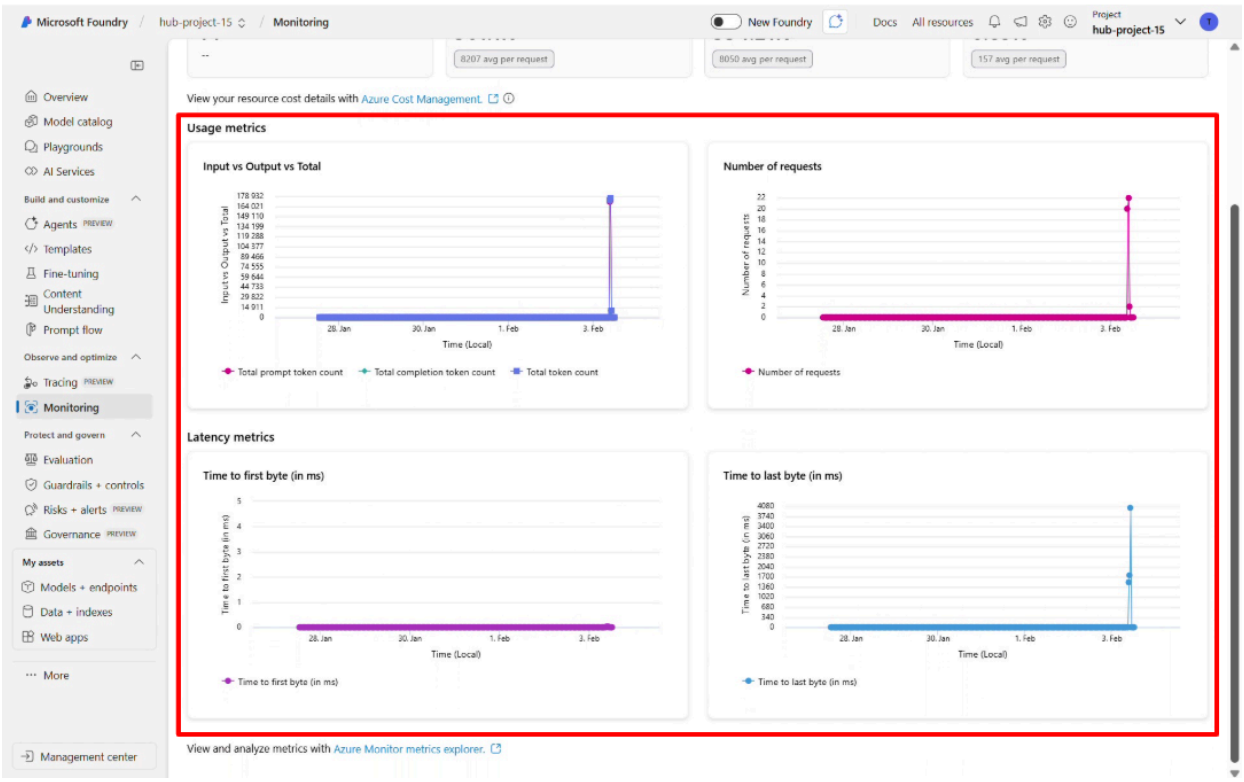
To ensure the RAG application remains a reliable, enterprise-grade asset, we implement a multi-layered monitoring strategy using **Azure App Services, Azure AI Search, Azure ML Workspace (or MLflow)** and **Azure AI Foundry**

Azure Foundry Service

- Metrics that can be monitored in the Azure Foundry Service include:
- Request/Response Latency: Measure metrics such as Time to First Token (user-perceived latency), Time per Output Token (the "reading speed" of the AI), Total Request Latency (The end-to-end round trip.)
- Number of Requests: The total number of requests addressed by the model



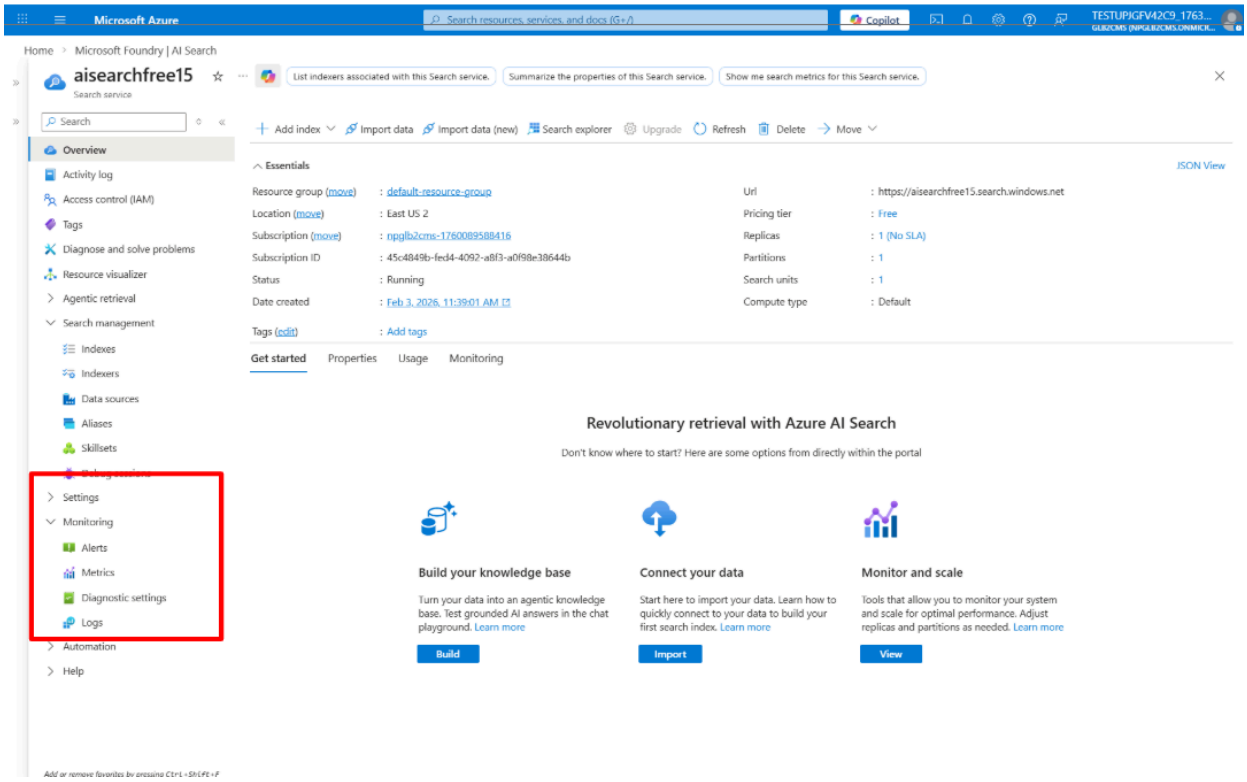
The Usage metrics for the model are shown as illustrated (Total Tokens, Number of Requests, Latency metrics)



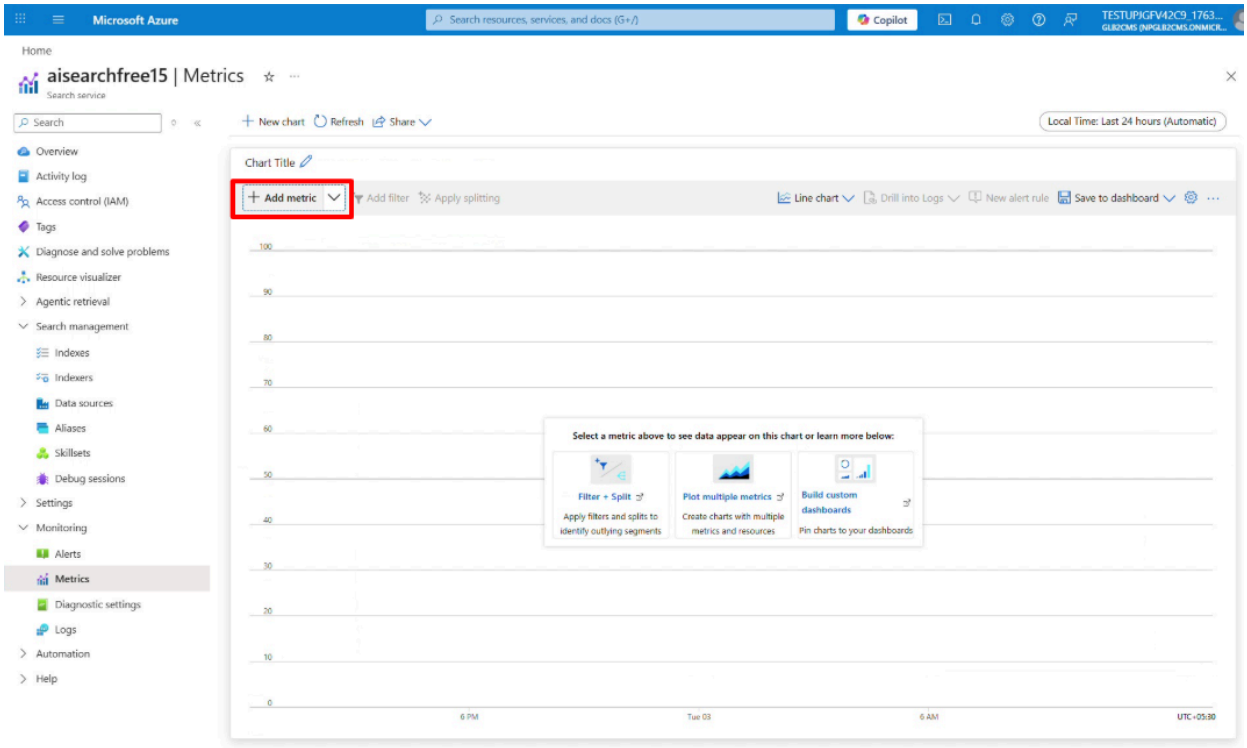
Azure AI Search

Metrics that can be monitored in the Azure Foundry Service include:

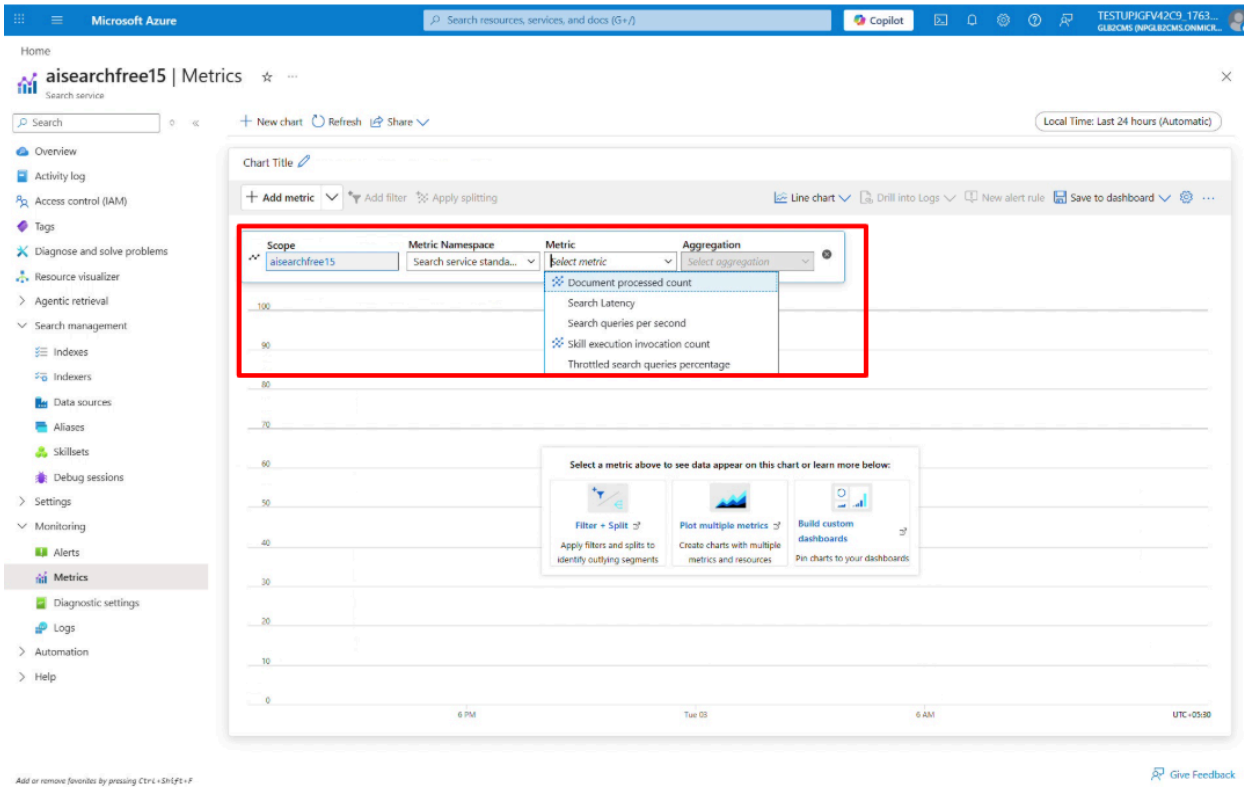
- **Search Latency & Throughput**: Track the time taken to retrieve PDF chunks to ensure the "R" in RAG does not become a bottleneck.
- **Index Health & Freshness**: Monitor document ingestion success rates and ensure the vector store reflects the most recent insurance policy updates.
- **Query Performance**: Analyze "Zero-Result" queries to identify gaps in the knowledge base or issues with the embedding model.
- **Throttling Detection**: Monitor Search Unit (SU) utilization to preemptively scale during high-traffic periods (e.g., catastrophe season).



Navigate to **Metrics** in **Monitoring.**



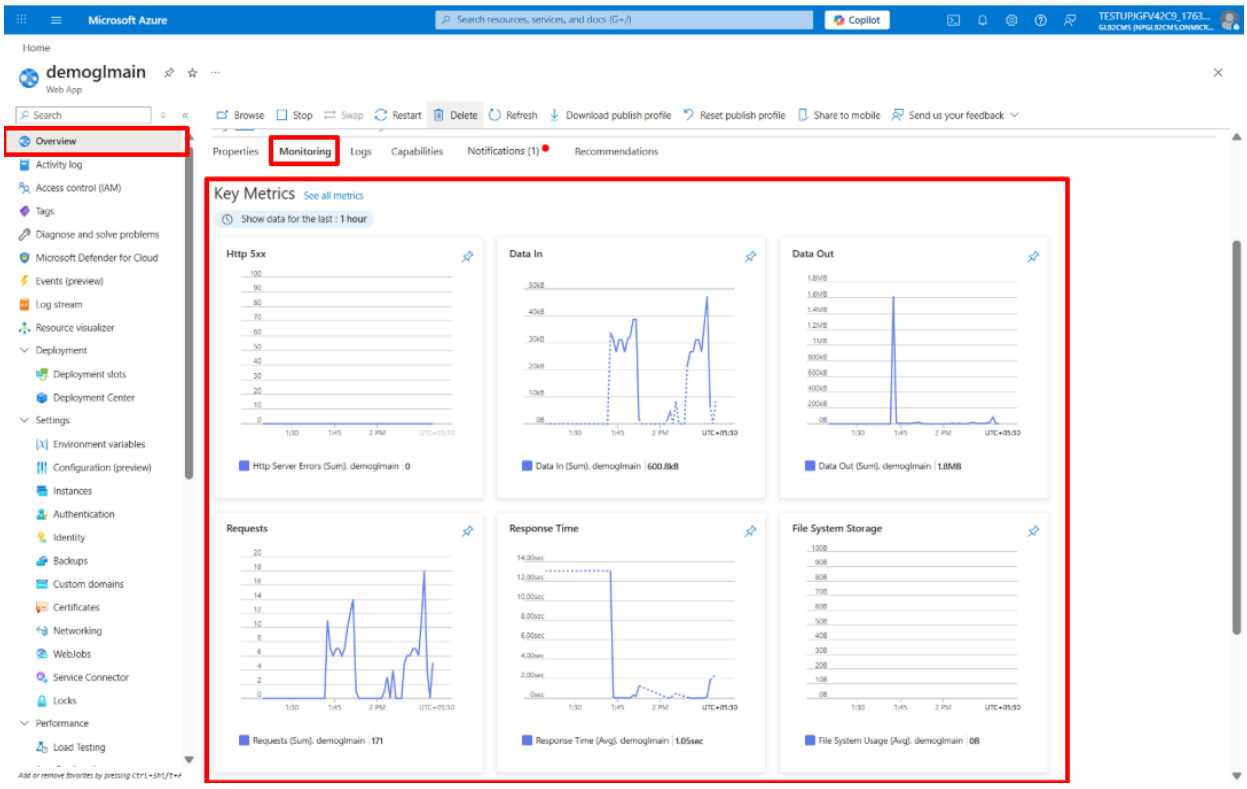
Select the appropriate metric to be monitored



Azure App Services

Metrics that can be monitored in the Azure Foundry Service include:

- **Request/Response Latency**: Measure end-to-end "Time to First Token" (TTFT) to ensure claims adjusters receive insights in under 2 seconds.
- **Resource Utilization**: Monitor CPU and Memory "spikes" during complex PDF parsing or heavy LLM orchestration tasks to optimize App Service Plan costs.
- **Auto-scaling Triggers**: Set alerts based on concurrency limits to ensure the system scales horizontally during peak claim filing hours.



Go to **Monitoring** -> **Metrics** in Azure App Services for detailed overview