Azure AI Human Detection Pipeline

Author: Amanda Sumner
Course: Advanced Python
Program: Data Science
Date: September 2025

GitHub repository

Output samples

1. Introduction

This report documents the development of a serverless AI pipeline for an Advanced Python course assignment. The project's purpose was to create an automated system to detect humans in images using Microsoft Azure. The solution is built with Python on Azure Functions, integrating with Azure Blob Storage and Azure AI Vision. This report covers the project's architecture, a code walkthrough, and a summary of my key learnings.

This serverless AI pipeline has several practical applications. One key use is **automated security camera monitoring**, where the system processes camera feeds and generates reports or alerts when a person is detected. The architecture can also be used for **automated content moderation**, such as scanning user-uploaded photos on social media for policy violations. This event-driven and scalable design makes it highly versatile for real-time image analysis.

2. Project Architecture

The project is an automated, serverless pipeline that demonstrates a core concept in modern cloud computing: event-driven architecture. This design allows the application to respond to events (like a file upload) without needing to be constantly running, which is both scalable and cost-effective.

1. Data ingestion (Azure Blob Storage): The process begins when an image file is uploaded to the designated input container, `camera-feed`, in Azure Blob Storage. This is the event that triggers the entire pipeline.

2. Event trigger (Event Grid): As soon as a new file is created in the `camera-feed` container, Azure Blob Storage publishes an event to an Event Grid topic. An event subscription is configured to forward this specific event to the Azure Function. This is the mechanism that "wakes up" the serverless function.

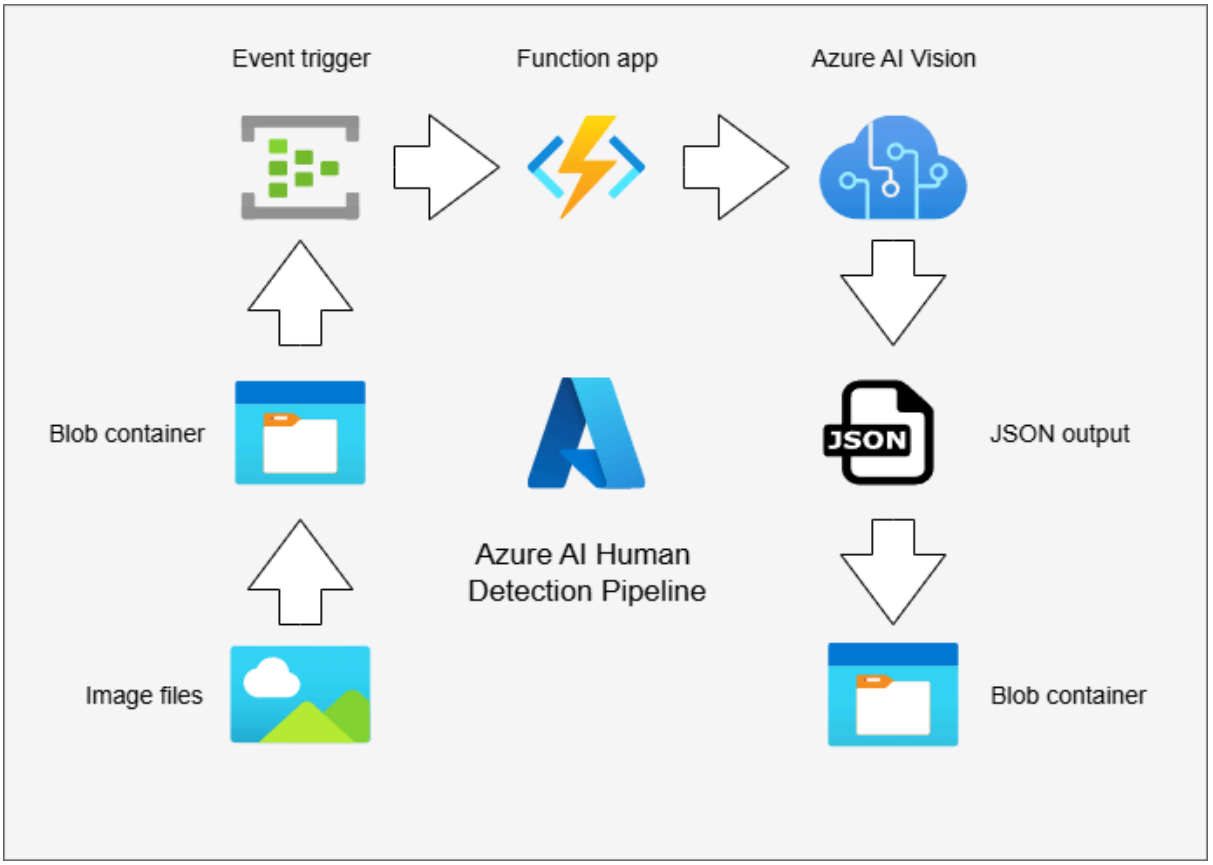
3. Serverless processing (Azure Functions): The Event Grid event triggers the Python-based Azure Function. The function's code then performs the following steps:

- It downloads the image data from the provided URL.
- It sends the image data to the Azure AI Vision service for analysis.
- It receives the analysis results, which include bounding box coordinates and confidence scores for any detected people.

4. AI analysis (Azure AI Vision): The Azure AI Vision service, a component of the Azure AI platform, uses its pre-trained models to perform the computer vision task. The service returns a structured JSON response containing the analysis.

5. Output storage (Azure Blob Storage): If people are detected in the image, the function generates a JSON report and uses an output binding to automatically save this report to a separate container, `analysis-results`, in Azure Blob Storage. This completes the end-to-end pipeline.

The diagram below visualizes the flow of data and the interaction between the key components.



3. Code Walkthrough

The project's core logic is contained within a single Python script, `function_app.py`, which uses the Python v2 programming model for Azure Functions. This approach uses decorators to define triggers and bindings, simplifying the code and removing the need for separate configuration files. I chose to handle the entire process, from data ingestion to analysis, within a single function named `ImageAnalysisFunction`.

Triggers and Bindings

The function is configured with a Blob trigger, which is linked to an Event Grid subscription. This makes the function event-driven and allows it to scale automatically.

- `@app.event_grid_trigger`: This decorator is the function's entry point. It tells the Azure Functions host to invoke the `ImageAnalysisFunction` whenever a new event is received from Event Grid. The incoming event payload, which contains metadata about the uploaded image, is passed to the function as the `event` parameter.
- `@app.blob_output`: This decorator is an output binding that handles saving the final analysis report. It specifies that a new file should be created in the `analysis-results` container. The report content is provided to this binding via the `outputBlob` parameter.

Authentication and Data Handling

The function uses a system-assigned managed identity. This eliminates the need to hard-code API keys in the code or configuration files.

- **Managed identity:** The `DefaultAzureCredential` is used to authenticate with Azure AI Vision and Azure Blob Storage. This credential automatically uses the function app's managed identity to acquire an access token, providing a secure and seamless way to connect to other Azure services.
- **Data flow:** The function's internal logic first checks if the received event is a `Microsoft.Storage.BlobCreated` event. If it is, the code extracts the blob's URL and name from the event payload. The image data is then downloaded directly from the blob using a `BlobClient`. This approach ensures that the function can process images of any size without being limited by memory.

Core Analysis Logic

Once the image data is available, the function performs the core AI task.

- The `ImageAnalysisClient` from the `azure-ai-vision` SDK is initialized using the environment variables for the endpoint and key.

- The `analyze` method is called, requesting the `VisualFeatures.PEOPLE` feature. This sends the image data to the AI service.
- The function then evaluates the `analysis_result` object. An `if` condition checks for the presence of people in the returned data. If people are detected, a detailed JSON report is structured, including the confidence score and bounding box coordinates for each person.

This JSON report is then passed to the output binding via `outputBlob.set()`, which automatically handles writing the file to the designated container in Azure Blob Storage.

```
import azure.functions as func # Core library for Azure Functions
import logging # For writing log messages
import os # To access environment variables
import json # For working with JSON data

# Client libraries for Azure AI Vision and Azure Blob Storage
from azure.ai.vision.imageanalysis import ImageAnalysisClient
from azure.ai.vision.imageanalysis.models import VisualFeatures
from azure.core.credentials import AzureKeyCredential

# Libraries for interacting with Azure Blob Storage
from azure.storage.blob import BlobClient
from azure.identity import DefaultAzureCredential # Used for managed
identity authentication

# Create a Function App instance
app = func.FunctionApp()

# Define the function's trigger and output bindings using decorators.
# The event_grid_trigger decorator configures the function to be invoked
by an Event Grid event.
# The 'arg_name="event"' specifies that the incoming event payload will be
passed to the 'event' parameter.
@app.event_grid_trigger(arg_name="event")

# The blob_output decorator defines a binding for writing data to Azure
Blob Storage. It specifies the output container ('analysis-results') and
the file naming convention.
@app.blob_output(arg_name="outputBlob",
path="analysis-results/{name}.json", connection="AzureWebJobsStorage")
```

```
def ImageAnalysisFunction(event: func.EventGridEvent, outputBlob:
func.Out[str]):
    """
      This function processes an image upload event, analyzes the image
      for humans, and saves a JSON report to Azure Blob Storage.
    """
      logging.info(f"Python EventGrid trigger processed an event:
      {event.get_json()}")

    try:
        # Parse the JSON payload from the EventGrid event
        event_data = event.get_json()

            # Check if the event has a 'url' key, which is expected for a
            valid blob creation event
        if 'url' in event_data:
            # Extract the blob URL and filename from the event payload
            blob_url = event_data['url']
            blob_name = blob_url.split('/')[-1]

            logging.info(f"Blob created at: {blob_url}")

            # Use managed identity to authenticate. DefaultAzureCredential
            automatically finds the correct credentials (e.g., from App
            Service managed identity).
            credential = DefaultAzureCredential()

            # Create a BlobClient to download the image data
            blob_client = BlobClient.from_blob_url(blob_url=blob_url,
            credential=credential)
            image_data = blob_client.download_blob().readall()

            # Get the endpoint and key for Azure AI Vision from
            application settings (environment variables).
            endpoint = os.environ["VISION_ENDPOINT"]
            key = os.environ["VISION_KEY"]

            # Initialize the AI client with the credentials
            client = ImageAnalysisClient(endpoint=endpoint,
            credential=AzureKeyCredential(key))

            # Send the image data to the AI service for analysis.
            # The 'VisualFeatures.PEOPLE' flag requests person detection.
            analysis_result = client.analyze(
                image_data=image_data,
                visual_features=[VisualFeatures.PEOPLE])
```

```
            # Initialize the report dictionary
            analysis_report = {
                "image_name": blob_name,
                "human_detected": False,
                "detection_details": []
            }

            # Check if the AI service detected any people. The 'people'
            attribute contains a list of detected person objects.
            if analysis_result.people and analysis_result.people.list:
                analysis_report["human_detected"] = True

                # Iterate through each detected person to extract details
                for person in analysis_result.people.list:
                    analysis_report["detection_details"].append({
                        "confidence": person.confidence,
                        "bounding_box": {
                            "x": person.bounding_box.x,
                            "y": person.bounding_box.y,
                            "width": person.bounding_box.width,
                            "height": person.bounding_box.height
                        }
                    })

            logging.info("Analysis complete. Generating JSON report.")

            # Write the analysis report to the output binding.
            # The 'json.dumps' method converts the Python dictionary to a
            JSON string.
            outputBlob.set(json.dumps(analysis_report, indent=4))
            logging.info(f"JSON report saved to
            analysis-results/{blob_name}.json")

        else:
            # Log a warning if the event payload is not a valid blob
            creation event.
            logging.warning("Event payload is missing a 'url' key.
            Skipping processing.")

    except Exception as e:
        # Log any errors that occur during the function's execution
        logging.error(f"An error occurred: {str(e)}")
```

4. Key learnings and conclusion

This project was a comprehensive learning experience that provided insight into the entire cloud development lifecycle. The challenges encountered were not just bugs, but valuable lessons in core Azure concepts.

Key Learnings

I gained hands-on experience in several crucial areas:

- **Azure resource management:** I learned to provision and configure resources in the Azure Portal, including creating a Function App, a Storage Account, and an Azure AI Vision service. I also learned how to link these services using application settings and connection strings.
- **Visual Studio Code integration:** The entire development and deployment workflow was handled through the Azure Functions extension in VS Code. This proved to be a powerful tool for developing, debugging, and seamlessly deploying code to the cloud.
- **Deployment workflow:** I gained practical experience with the deployment process itself, including how Python code is packaged and deployed to a remote Linux host.

Conclusion

The most significant challenges involved understanding and configuring the relationships between different cloud services. I successfully solved issues related to:

- **Permissions and identity:** The Function App's managed identity required the explicit assignment of IAM roles to enable secure communication with the storage account and the AI service.
- **Trigger and binding conflicts:** I resolved a conflict between the BlobTrigger and the EventGridTrigger to ensure the function was correctly discovered by the Azure Portal.

In conclusion, this project successfully built a robust, end-to-end serverless AI pipeline on Azure. It provided valuable experience in diagnosing and resolving complex cloud-native issues, which I consider a crucial skill for my career as a data scientist.