1. DEPLOYMENT

What is deployment?

Deployment is the process of making a software application or system available for use in a production environment. It involves activities such as code compilation, configuration management, and testing to ensure the application operates as intended. Deployable artifacts are released to the target environment, and infrastructure may be provisioned as needed. Post-deployment verification and monitoring are essential to ensure the application's correct functionality, and rollback plans are prepared for contingency.

**Objective of our project deployment:**

Orchestrating an end-to-end pipeline for the development and deployment of a sentiment detection model involves creating a seamless and automated workflow that spans the entire lifecycle of the model. The designed solution aims to streamline the monitoring and assessment of maintenance activities within Thameslink to quickly identify areas of concern thereby leading to prompt response and continuous improvement while being scalable, secure and highly available for data storage.

* 1. Pre-requisites and Interfaces:

Before proceeding with the deployment, the system must be configured with the pre-requisites mentioned below:

* Twitter account : Required to connect to Twitter Developer account and to generate authentication keys and tokens to connect to Twitter API.
* Azure account : To integrate with VSCode for building, deploying and managing the entire orchestration on cloud premises. Create an Azure account for students to get the Azure Student Subscription, which comes with a free 100$ credit to use the necessary Azure resources.
* VSCode : Required to build, integrate and deploy the Python codes.

The following interfaces have been identified for the end-to-end deployment process :

* The first interface is between Twitter (X) and VSCode , to pull the tweets with specific keywords by connecting to Twitter API , using Python as the underlying programming language.
* The second interface is between VSCode and Azure Cloud Ecosystem (specifically Azure serverless function) to build and deploy the python code in VSCode to Azure.
* The final interface is between Azure Cloud (specifically Azure SQL Database) and Power BI (for data visualization).

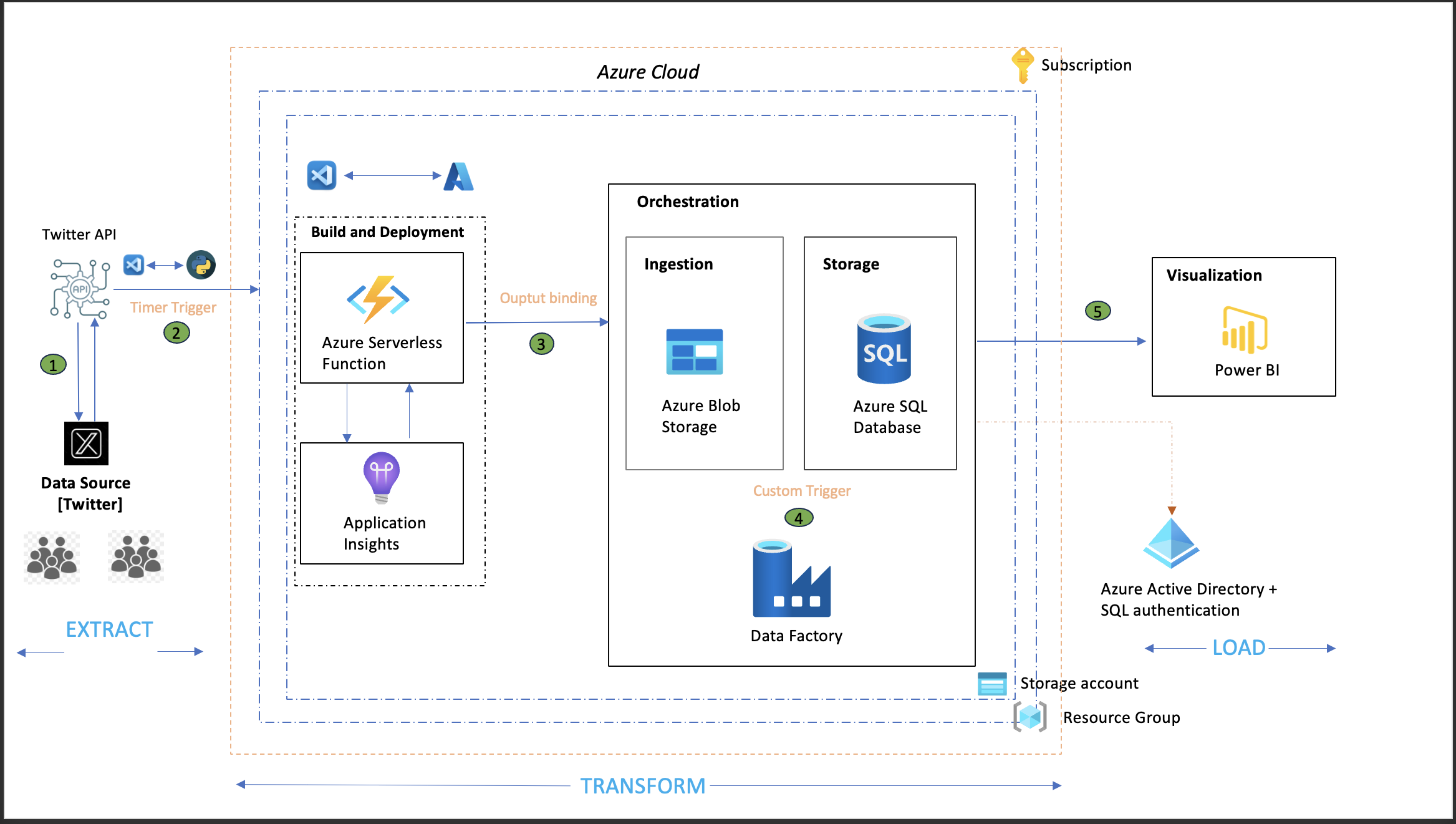




1 2 3

1.2 Deployment Architecture:

In the deployment architecture, we embark on a journey that begins with Twitter data acquisition through the Twitter API. Visual Studio Code (VS Code) serves as our development and integration hub, orchestrating the entire process. Leveraging Azure's serverless function capabilities, we deploy sentiment analysis functions with output bindings, seamlessly storing the analyzed data into an Azure Blob Storage. The synchronized data is then efficiently processed and orchestrated into Azure SQL Database using Azure Data Factory, forming a vital link in our data pipeline. Finally, the enriched data is visualized in Power BI, providing insightful analytics to empower the Thameslink maintenance team with real-time sentiment insights derived from the tweets. This architecture combines the flexibility of serverless computing, the power of Azure services, and the visual storytelling capabilities of Power BI for a comprehensive and impactful sentiment analysis solution.



1.2.1 Extraction Phase:

*Step 1: Accessing Twitter Developer Platform*

To initiate the process, navigate to the Twitter Developer website and log in using the Twitter account credentials. Once logged in, proceed to create a new project on the platform and add an application to it. This step also involves generating essential authentication keys and tokens required to connect to the Twitter API programmatically. Specifically, we will need the API key, API secret key, Access token, and Access token secret.

*Step 2: Connecting to Twitter API with Tweepy and generated API credentials*

With the obtained API credentials, the next step involves connecting to the Twitter API using the Tweepy library in Python. Tweepy is a widely used Python library for interacting with the Twitter API, offering convenient functionalities for accessing and extracting tweets. Tweepy acts as a wrapper around the Twitter API, providing a convenient interface for sending requests and receiving responses.

*Step 3: Tweet Extraction and Storage*

Transitioning to the extraction phase, our primary goal is to retrieve tweets that specifically contain predefined keywords. This task is efficiently tackled by leveraging the robust search functionalities offered by the Twitter API through the Tweepy library. The extracted tweets (in JSON format) are flattened before storing them in a Dataframe. To enhance the organization and maintainability of our codebase, we decided to adopt a modular code strategy by organizing our code into main modules and corresponding submodules and the submodules being called by the main module during execution.

1.2.2 Transformation Phase:

Transformation Phase 1 [Build and Deploy]

The first phase of transformation involves integrating the python code (end-to-end model) present in VSCode with Azure.

*Step 1: Creating a resource group and storage account.*

In Azure, a Resource Group is a logical container for resources deployed in a region. It helps you manage and organize related Azure resources, such as virtual machines, storage accounts, and databases, as a single administrative unit. Resource groups enable you to manage and monitor resources collectively, apply policies, and control access and permissions.

A Storage Account, on the other hand, is a fundamental Azure resource that provides scalable and secure cloud-based storage. It supports various types of data storage services, including blobs (for unstructured data), tables (for NoSQL data), queues (for message communication), and files (for file storage).

*Step 2: Creating an Azure serverless function app*

After creating a resource group and storage account, the next step is to create an Azure serverless function app. Azure Functions allow us to run event-triggered code without managing the underlying infrastructure. They automatically scale based on demand and are an excellent choice for serverless computing scenarios. VSCode can be integrated with Azure functions by installing the necessary Azure extensions. After writing/ modifying function code in VSCode, we can use the debugging features in VS Code to test your function locally. Once satisfied, deploy the function to Azure using the Azure Functions extension.

*Step 3: Creating a timer trigger*

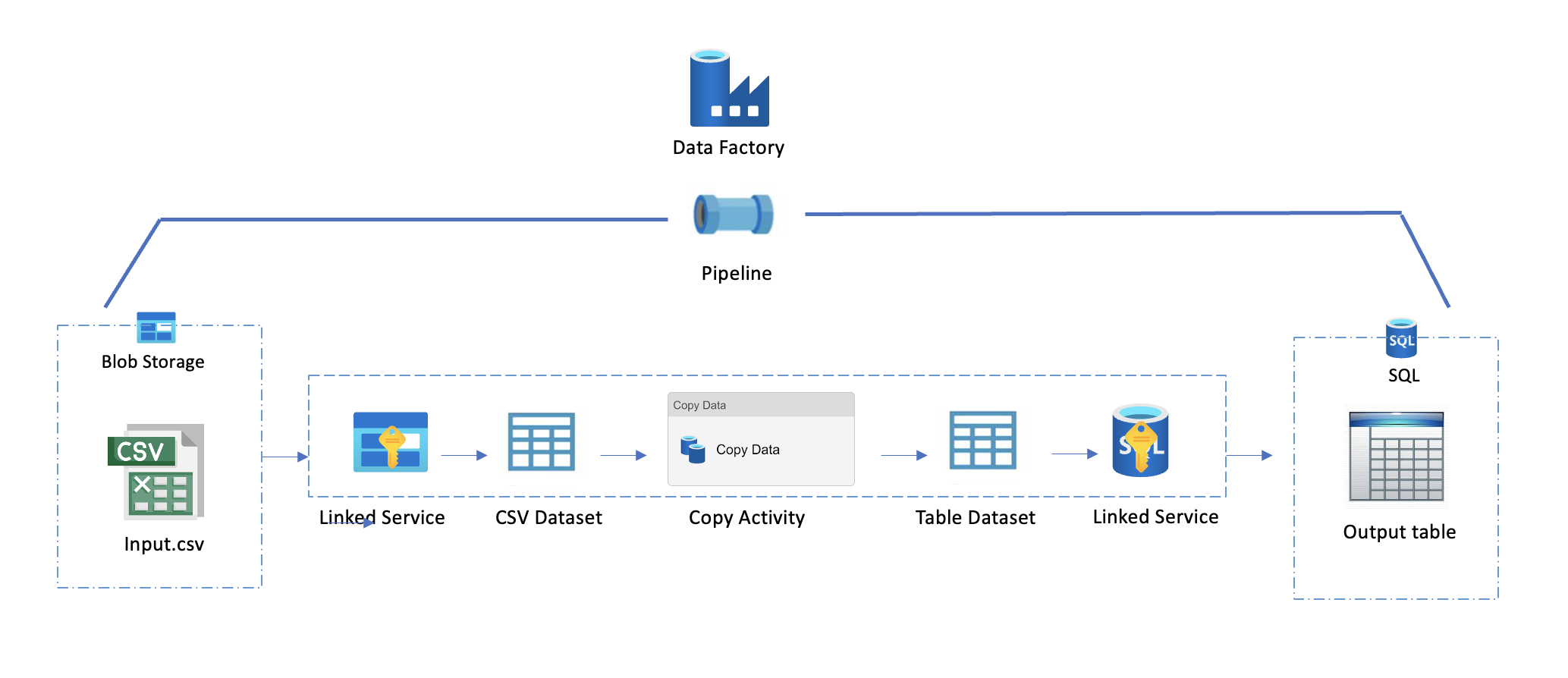
Once function app is created, create a new function in Azure portal with timer trigger. A timer trigger is a type of trigger that helps to run a function on a specified schedule , similar to a cron job. Eg : "0 0 0 \* \* \*" [ runs everyday at midnight ]. In our scenario the trigger was scheduled to run once per day to accumulate all the tweets, run the model on the data collected and retrieve suggestions and recommendations to improve the services of Thameslink.

*Step 4: Creating an output binding storage*

An output binding storage defines the target destination or resource in a declarative manner within the function's configuration function.json file. Here the output binding is Azure Blob Storage which indicates output of Azure functions will be stored in a blob container to then move it to Azure SQL database in the final transformation phase.

Transformation Phase II [Orchestration]

Orchestration involves defining and managing pipelines and activities within the data integration and transformation processes. Here we have used, Azure Data Factory to copy the data from Azure Blob Storage (source) to Azure SQL Database (sink) using copy as the data movement activity with custom trigger.



The following steps were undertaken to perform the orchestration after creating an Azure Data Factory instance and connecting it to the Azure storage account:

*Step 1: Define source and sink*

Azure Blob Storage acts as the source, holding a versatile range of structured, semi-structured, and unstructured data, specifically in the form of CSV files. The destination, or sink, is an Azure SQL Database designed to handle relational data. The orchestrated flow involves extracting, transforming, and loading (ETL) data from the diverse CSV files stored in Azure Blob Storage into a structured format suitable for relational storage in Azure SQL Database.

*Step 2: Create linked services for source and sink*

Linked services store the connection information and credentials needed to connect to the data sources. For the source, these linked services encapsulate the information required to establish a connection with Azure Blob Storage, where the CSV files reside. Meanwhile, the linked service for the sink encapsulates the connection details for the Azure SQL Database, the relational storage destination.

*Step 3: Create datasets for source and sink*

Datasets define structure of the data and to also indicate how it should be consumed by activities in the pipeline.

*Step 4: Pipeline creation in Azure Data Factory*

Creating a pipeline and adding a "Copy Data Activity" is a fundamental step in orchestrating data movement between different data sources. Once pipeline is created and the activity is added, configure the copy data activity by configuring the source and sink datasets to point to source and sink providing necessary details. Map the source and sink schemas correctly. Also, add a custom trigger to facilitate movement of data from Blob Storage to SQL Database.

*Step 5: Execute the pipeline*

Run the pipeline to execute the data copy operation. Monitor the progress, success, or any errors in the Azure Data Factory Monitoring interface.

1.2.3 Loading Phase:

The interface for the loading phase involves utilizing an Azure SQL database as the final storage repository, providing a secure and scalable solution for data storage. To enhance data analysis and presentation, Power BI is employed as the visualization tool for creating interactive and insightful dashboards. This interface architecture ensures a seamless flow of data from storage to visualization, empowering users to derive meaningful insights and make informed decisions based on the stored information in the Azure SQL database.

To establish connection between the interfaces, we require the below from Azure :

* Azure Server name (eg : <server\_name>.database.windows.net)
* Database name , username and password (required if using SQL server authentication).
* Update Azure server firewall settings to include the IP address of the machine trying to connect to the SQL database.

1.3 Challenges:

* The major challenge encountered was an ODBC Driver compatibility issue thereby failing to establish a direct connection between the Azure Function App and SQL database. Leveraging the capabilities of Azure Data Factory proved to be a strategic resolution. By orchestrating data movement between the source and sink using custom triggers, compatibility challenges were bypassed, ensuring seamless and reliable data transfer. This not only resolved the immediate connection issue but also introduced a more robust and scalable solution for managing data flow within the Azure ecosystem.
* The Tweetnlp library (a custom library), designed for irony and sentiment analysis, was facing recognition issues within the Azure environment, which led to a build failure. The library was checked with the Azure Function version and Python version and no incompatibilities found. The possible option to resolve the conflict was to import the source code of the library into Azure environment, which was quite challenging. As an easy fix, we decided to demonstrate the end-to-end data pipeline and orchestration using the output csv file obtained from the model.
  1. Best Practices for Sustained High Performance of Deployed Models:
* Continuous Monitoring and Resource Scaling:

1. Use monitoring tools like Azure Monitor and Azure Application Insights to track the deployment process in real-time. Both Azure Monitor and Application Insights seamlessly integrate with various Azure services, making it easier to monitor the entire application stack.
2. Adjust the compute resources allocated to the deployed model dynamically based on changes in demand or data volume. Also define auto-scaling rules based on specific metrics, such as CPU usage or request rates, to trigger automatic adjustments to resource allocation.

By combining continuous monitoring with dynamic resource scaling, you can ensure that your deployed models are both performant and cost-effective, adapting to changing demands in real- time.

* Data Drift Monitoring: Data drift occurs when the statistical properties of the input data used for training a machine learning model change in the production environment. This shift in data distribution can impact model performance, leading to a degradation in predictions.

1. Azure Machine Learning Data Drift allows you to monitor the distribution of features in the incoming data and compare it with the distribution of the training data.
2. Integrate data drift monitoring with automated model retraining pipelines. When significant drift is detected, trigger a retraining process to update the model with the most recent data.

* Documentation: Maintain comprehensive documentation about the model, its deployment, and any updates or changes made over time.
* Feedback Loops: Establish feedback loops to collect information about model predictions and their real-world outcomes.
* Logging and Auditing: Ensure that the modular code has logging mechanisms to capture information for debugging and troubleshooting.
* Model Retraining: Schedule periodic retraining of the model using new and updated data to ensure model remains accurate and relevant to changes in data.