Deployment Pipeline

Code - https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/tree/production
Video -
https://drive.google.com/file/d/14j2F7C7ytSy3sLw8IUjoGD3pgAVxljje/view?usp=sharing

Overview

This project aims to forecast energy demand for a given location by utilizing weather data. The main objective is to predict the energy demand based on current weather conditions, helping stakeholders better prepare for future energy needs. The system also extends beyond energy forecasting by incorporating stock market predictions. It provides insights into how forecasted energy demand may affect energy companies and their stock prices. The project leverages multiple machine learning models to generate these predictions, making it a powerful tool for both energy management and financial forecasting.

Working of Product

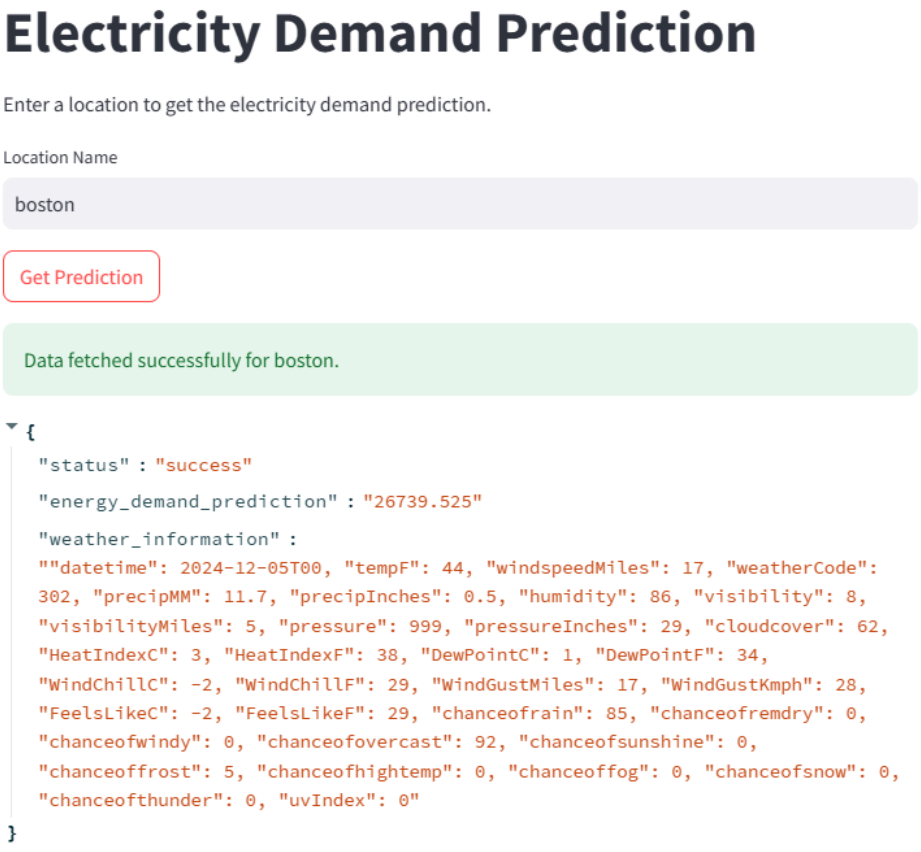
The product works by collecting data from two key sources: a Demand API that provides historical energy demand data, and a Weather API that supplies weather data. This data is preprocessed and stored in DVC (Data Version Control), ensuring that the data is well-managed and reproducible for future use.

To predict energy demand, the project experimented with machine learning models, including XGBoost, Linear Regression (LR), and Long Short-Term Memory (LSTM) models. These models are trained using historical weather and demand data. The energy demand prediction is based on weather patterns and other factors, allowing the system to forecast demand for the next hour as well as the following day on average.

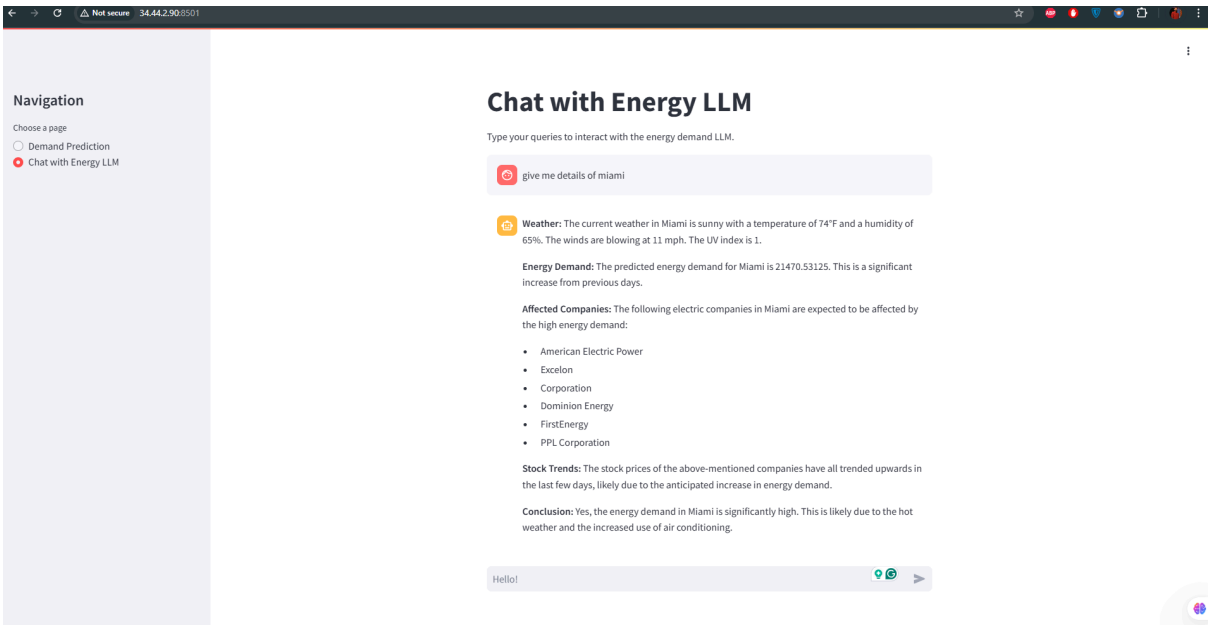
The backend of the application allows users to input a location (such as "Boston"), and the system handles the prediction process in several stages. First, the location is passed through a Geo API to fetch the geographical coordinates. Using these coordinates, the backend retrieves the past 7 hours of weather data from the Weather API. A sliding window approach of size 6 is then applied to normalize the weather data for the last six hours. This preprocessed data is subsequently fed into the trained machine learning models to predict energy demand for the next hour and the following day.

In addition to predicting energy demand, the project integrates a Language Learning Model (LLM) to offer financial stock recommendations. The process begins by predicting energy demand based on the weather data and location coordinates. Once the energy demand is forecasted, the system identifies the nearest energy companies to the given location that are publicly traded on the stock market. The stock price, stock history, and balance sheet data of these companies are retrieved. The LLM then analyzes all the information and generates a comprehensive report. This report includes a detailed overview of the predicted energy demand, weather conditions, and an analysis of the potential impact on the stock prices of these energy companies. The report also predicts whether the stock prices of these companies are likely to rise or fall based on the forecasted demand and market data.

The system is deployed with an API endpoint that allows users to interact with the application. Users can input a location, and the backend processes the data and returns both the energy demand prediction and a stock market analysis for the nearest energy companies. This API serves as the main interface for users to access the predictive models and financial insights generated by the system.



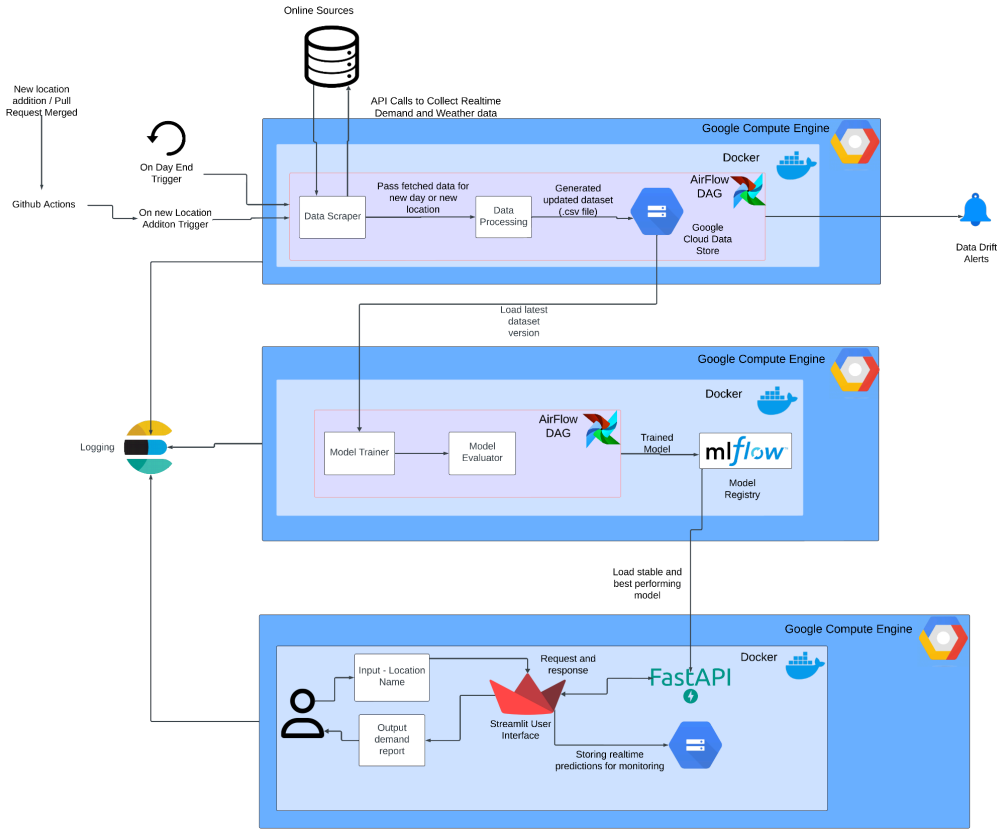
Example UI: Demand prediction



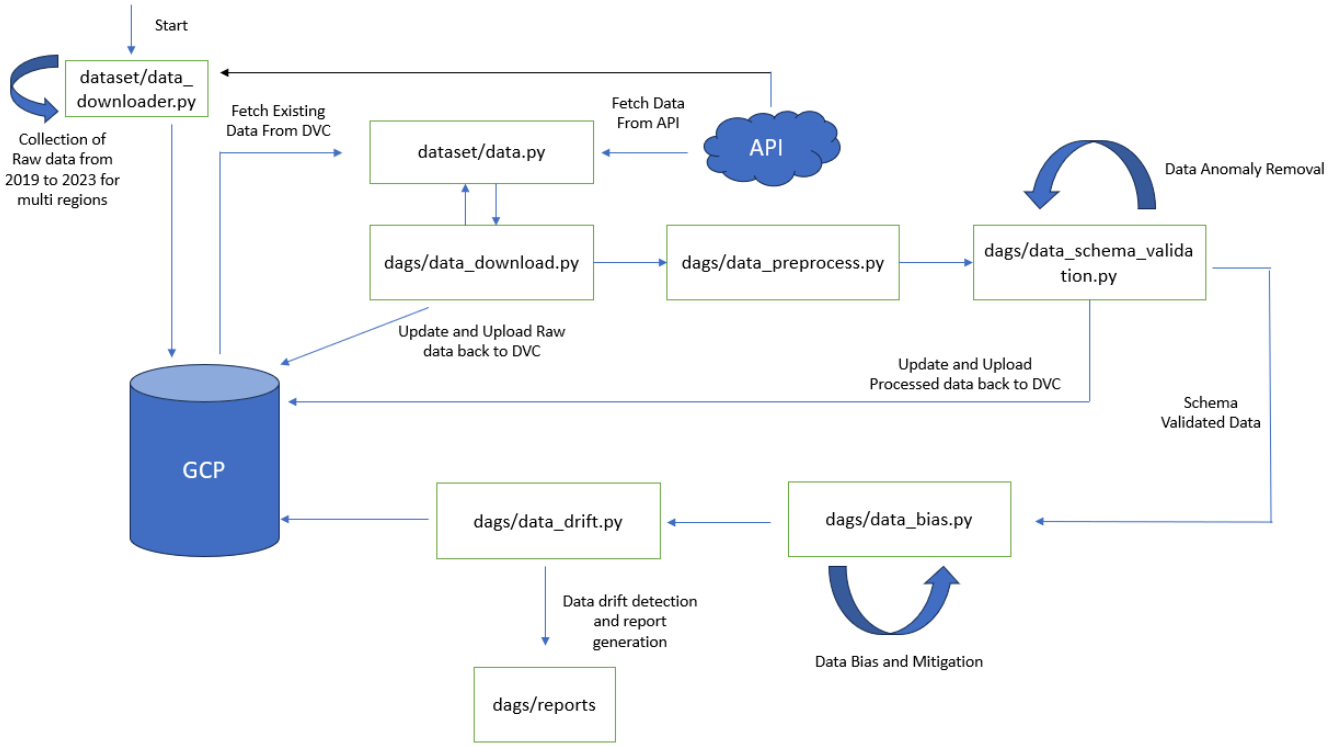
Example UI: LLM report

Project flow chart

1) Project flow



2) Data processing flow

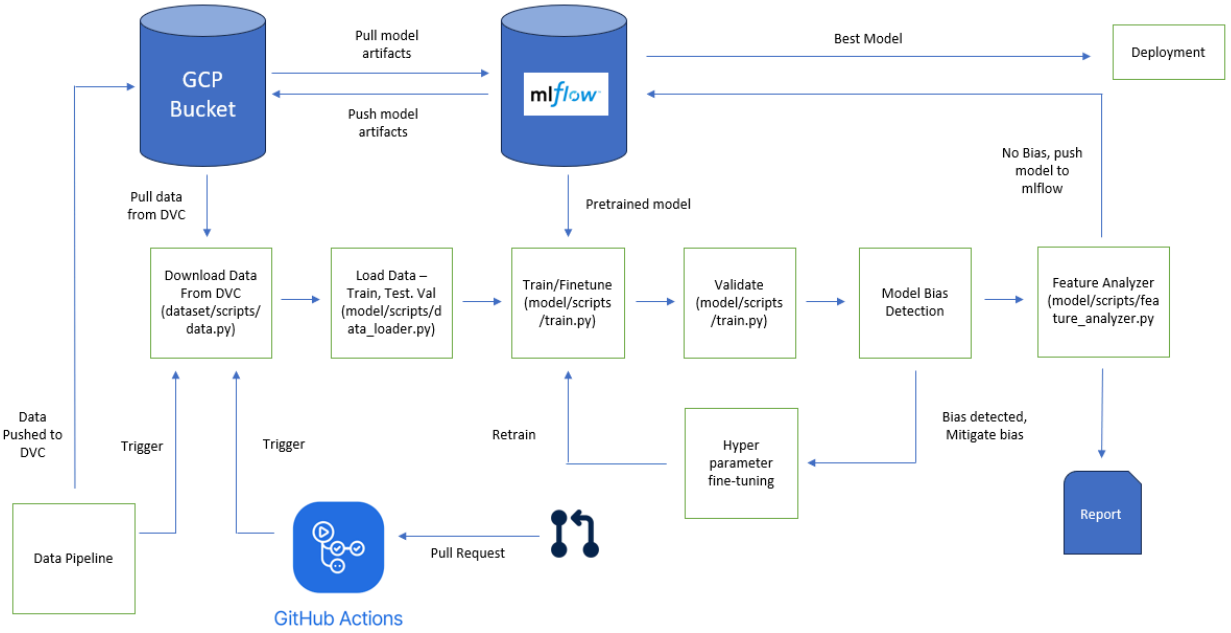


The diagram outlines a comprehensive data engineering pipeline designed to collect, process, validate, and analyze data from various sources. The pipeline leverages DVC for data version control and GCP as the primary storage solution.

The process begins by fetching data either from DVC or an external API. The collected data undergoes thorough cleaning and preprocessing to ensure data quality and consistency. Subsequently, the data is validated against a predefined schema to maintain data integrity.

Once validated, the processed data is stored back in DVC, enabling reproducibility and traceability. Additionally, the pipeline incorporates data quality checks, including anomaly detection and data drift monitoring. By proactively identifying and addressing these issues, the pipeline ensures the reliability and accuracy of the data.

3) Model training flow



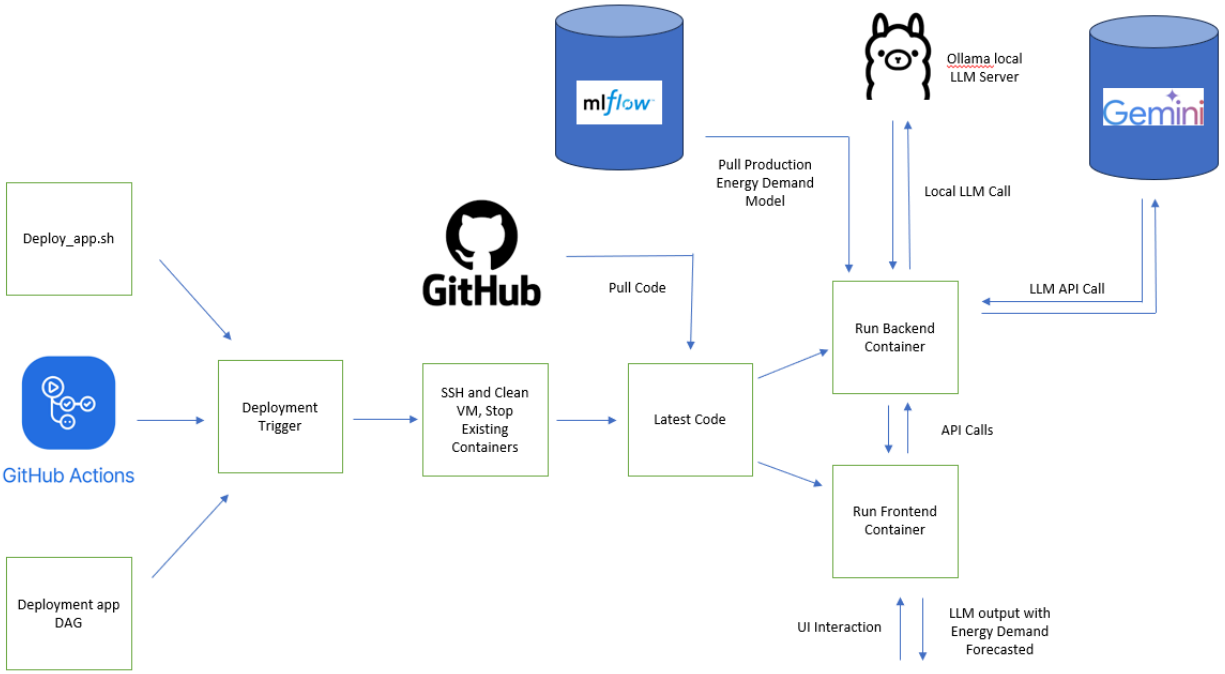
This diagram illustrates the complete pipeline for training, validating, and deploying machine learning models, with a focus on data flow, automation, and bias detection. The pipeline begins with data pushed to DVC (Data Version Control), which serves as the central repository for versioned datasets. When triggered, data is pulled from DVC into the pipeline, where it is downloaded and preprocessed using dedicated scripts. The data is then divided into training, validation, and test sets.

The training process starts with a pre-trained model retrieved from MLflow. The model is fine-tuned or trained on the new data, followed by validation to assess its performance. The pipeline includes a model bias detection step, where any bias in the model is identified. If bias is detected, mitigation measures are applied, followed by hyperparameter fine-tuning to optimize model performance. Once validated and deemed unbiased, the best-performing model is pushed back to MLflow as the production model, ready for deployment. Additionally, a feature analysis report is generated to provide insights into feature importance and any observed biases.

The workflow is integrated with GitHub Actions for automation, enabling retraining upon pull requests or changes in the dataset. Model artifacts and results are stored in a GCP Bucket, ensuring centralized access to all resources. This pipeline ensures robustness, fairness, and reproducibility in the ML model lifecycle.

4) Deployment

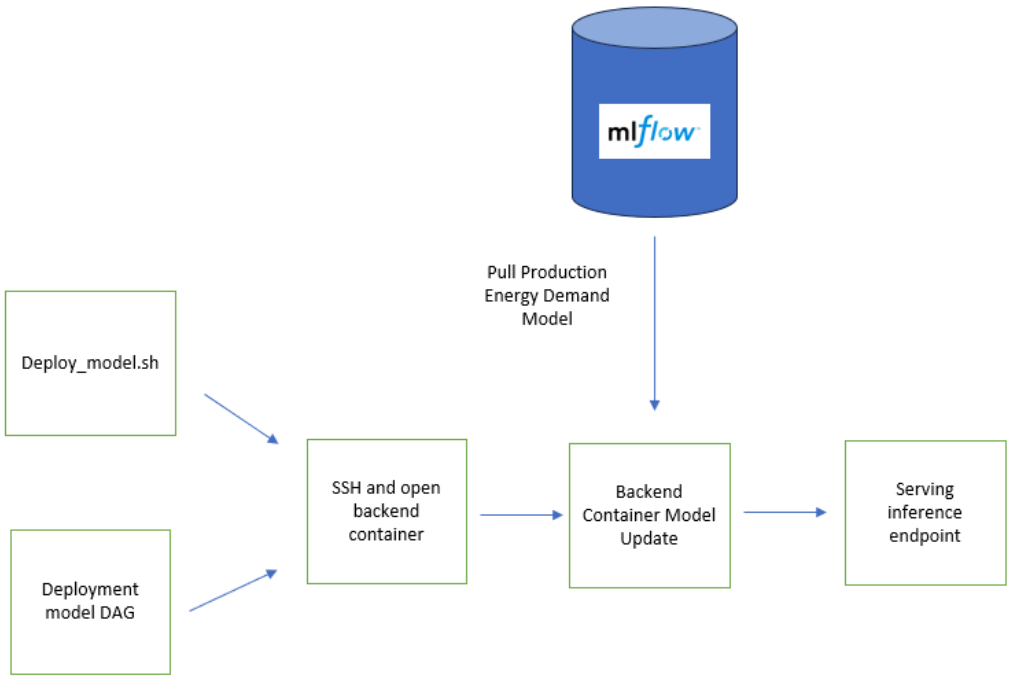
- Deploy App Flow



The diagram showcases the deployment and interaction workflow of the energy demand forecasting application, emphasizing automation, model management, and containerization. Deployment of the application begins with a trigger, which can originate from three sources: a `deploy_app.sh` script, a GitHub Actions workflow, or a deployment DAG executed via Airflow. Once triggered, the process involves SSHing into the target VM, stopping any existing containers, and pulling the latest application code from the GitHub repository.

Upon fetching the latest code, the system initiates the deployment of backend and frontend containers. The backend container integrates with the MLflow server to retrieve the production energy demand model and uses APIs to handle weather, stock, and financial data. The backend also connects to a local LLM server (Gemini or Ollama) to generate comprehensive analytical reports based on user queries. The frontend container facilitates user interaction, sending input queries to the backend for processing. This tightly coupled automation ensures a seamless workflow for deploying updates and delivering accurate forecasts and financial recommendations with minimal manual intervention.

- Deploy Model Flow

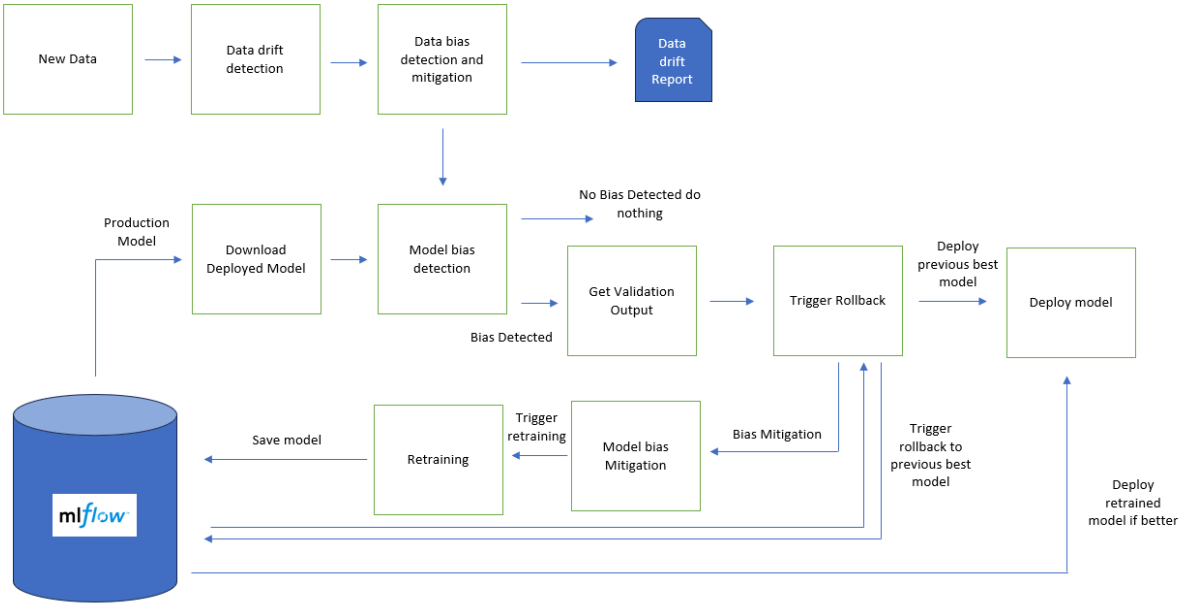


The provided diagram depicts a streamlined process for deploying machine learning models into production. The pipeline initiates with a shell script (`Deploy_model.sh`) that triggers the deployment workflow, visualized as a directed acyclic graph (DAG). This DAG outlines the sequence of steps required to successfully deploy the model.

The first critical step involves establishing a secure connection to the backend container using SSH. This container serves as the deployment environment for the model. Once the connection is established, the model is updated within the container, which might involve copying new model files, modifying configuration settings, or rebuilding the model image.

The final stage in the deployment process is the creation of an inference endpoint. This endpoint acts as a gateway for clients to interact with the deployed model. By sending requests to this endpoint, clients can obtain predictions generated by the model. The endpoint ensures seamless communication and facilitates the utilization of the model's capabilities.

5) Monitoring Flow



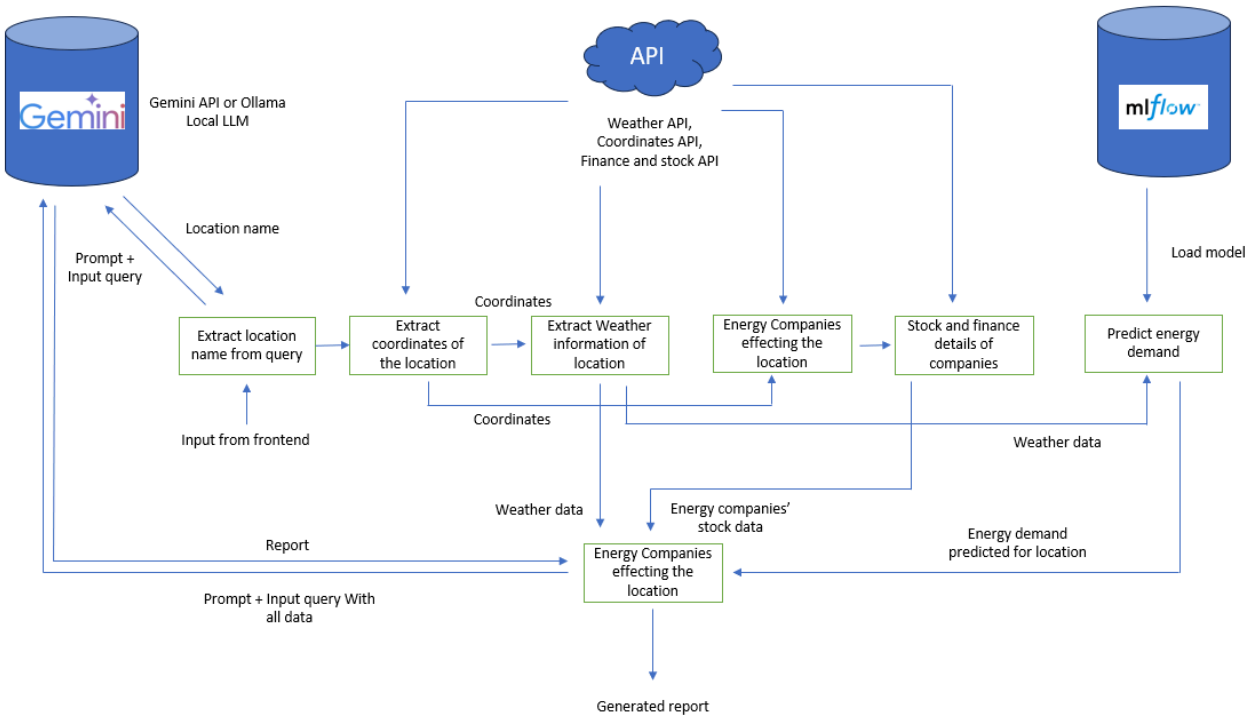
Data Ingestion and Analysis The process begins with the introduction of new data. This data is meticulously analyzed to detect any significant shifts in its distribution compared to the original training data, a phenomenon known as data drift. If data drift is identified, the system further scrutinizes the data for potential bias.

Bias Detection and Mitigation Should bias be detected, the system employs various mitigation techniques to rectify the issue. These techniques might include data augmentation, algorithmic adjustments, or rebalancing the training dataset.

Model Deployment and Monitoring Once the data has been processed and any bias has been addressed, the model is deployed into production. However, the monitoring process doesn't cease here. The model's performance is continuously evaluated to identify any emerging bias in its predictions.

Retraining or Rollback If bias is detected in the deployed model, the system triggers a decision-making process. Depending on the severity and nature of the bias, the system may opt for retraining the model on a corrected dataset or reverting to a previously reliable model. Retraining involves incorporating bias mitigation techniques to ensure the new model is more equitable.

6) Backend Flow



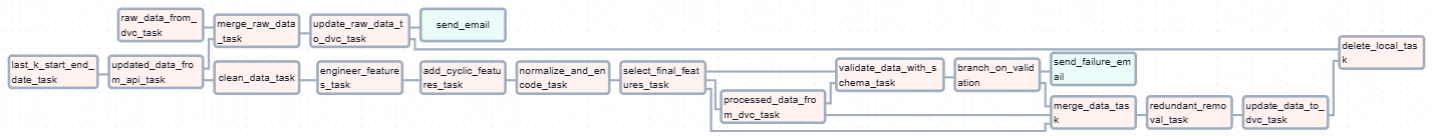
The diagram illustrates the workflow of an energy demand forecasting and financial stock recommendation system, showcasing its integration with multiple APIs, an MLflow server, and a local LLM such as Gemini or Ollama. The system begins with user input from the frontend, typically in the form of a query containing a location name. The input is processed to extract the location name and corresponding coordinates using a coordinates API. Once the coordinates are determined, they are used to fetch weather data for the specified location from a weather API. Simultaneously, the system identifies energy companies operating in or near the location using a mapping process and gathers their financial details, including stock prices, historical performance, and balance sheets, through a finance and stock API.

The MLflow server plays a critical role in the pipeline by providing the best-trained energy demand forecasting model. This model predicts energy demand based on the extracted weather data, offering insights into both immediate and future energy needs. These predictions are combined with the weather data, stock details, and financial summaries of nearby energy companies. This consolidated information is then processed by the local LLM, which generates a comprehensive report. The report not only includes the predicted energy demand but also provides actionable insights into how the demand might influence the stock prices of the identified companies. This end-to-end flow ensures a seamless interaction between weather forecasting, energy modeling, financial analysis, and natural language reporting.

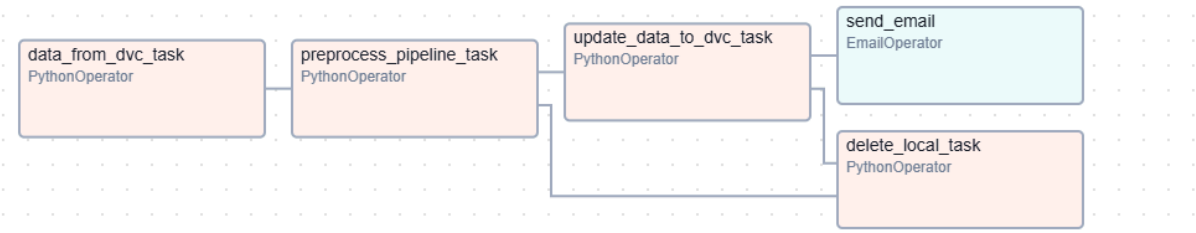
DAG flow charts

(Information on each DAG - https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/README.md)

1) New data download and preprocess DAG

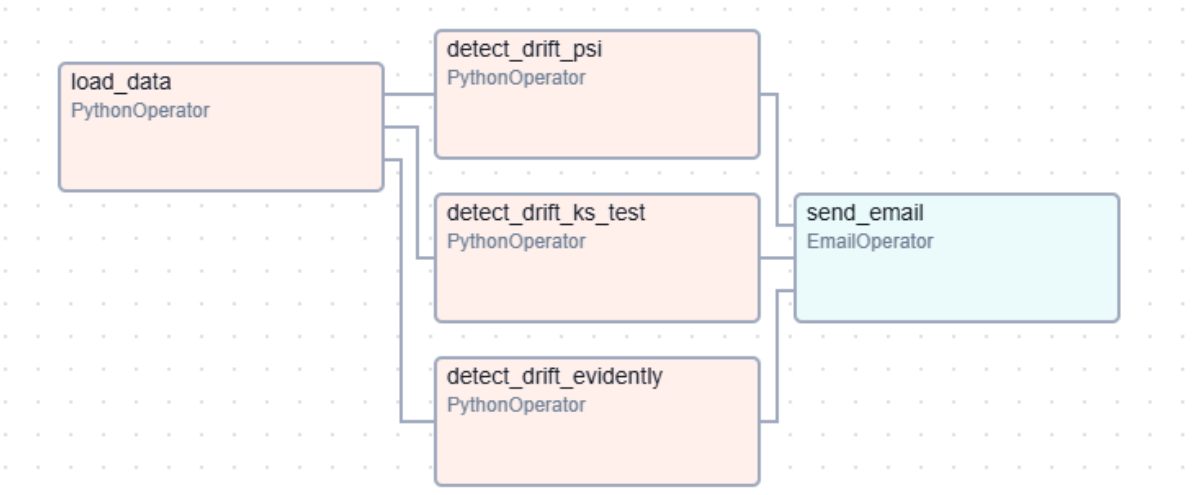


2) Raw data download DAG and preprocess DAG

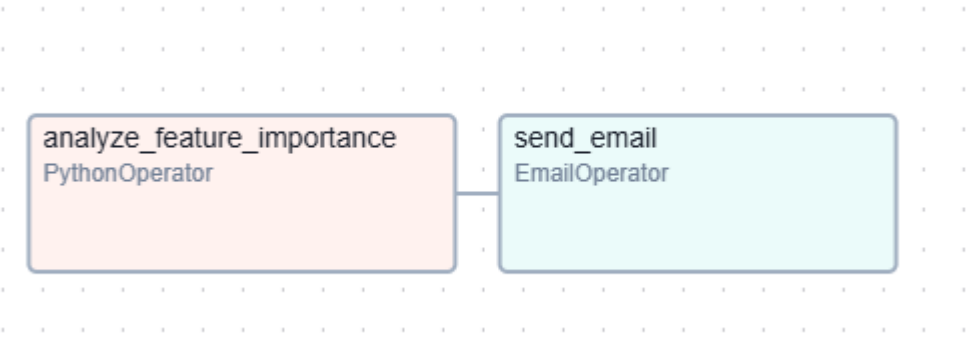


3) Data bias detection and mitigation DAG

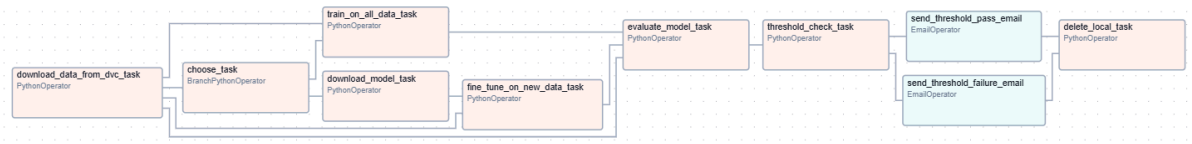
4) Data drift detection and report generation DAG



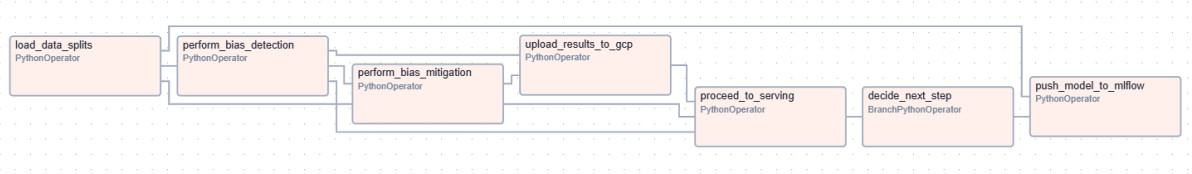
5) Data feature importance analysis DAG



6) Model training and evaluation DAG

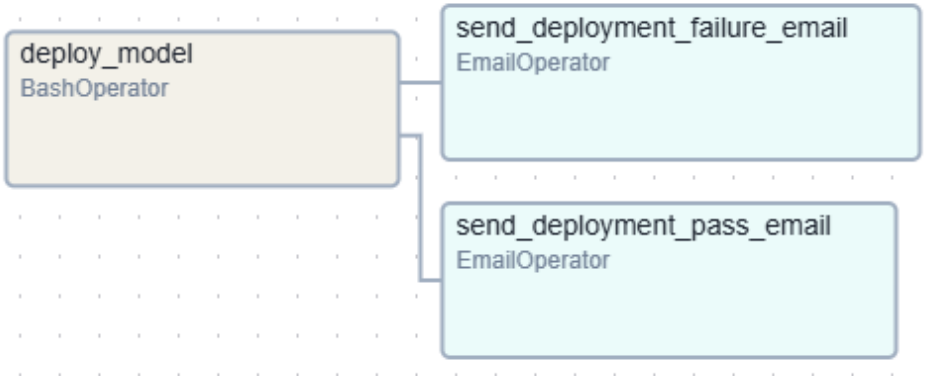


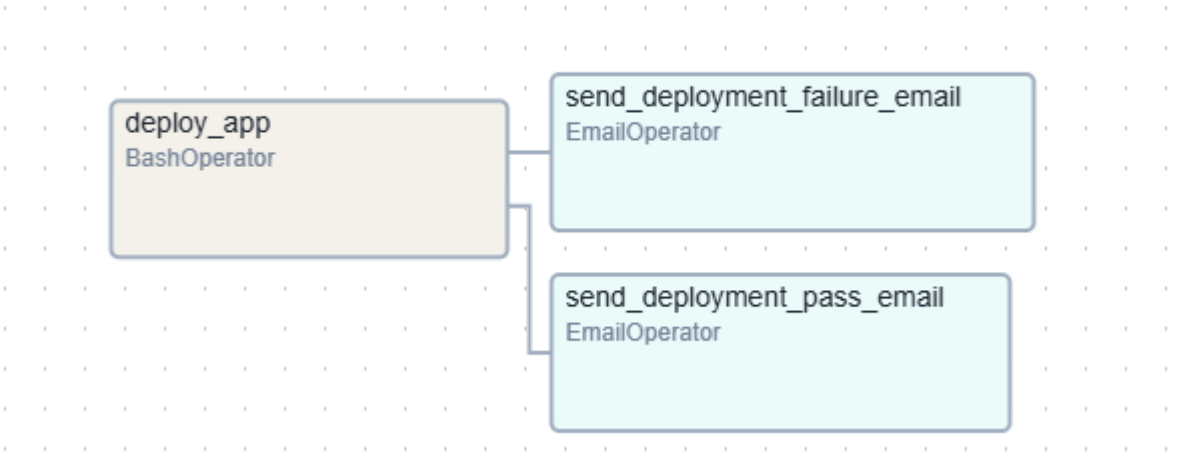
7) Model bias detection DAG



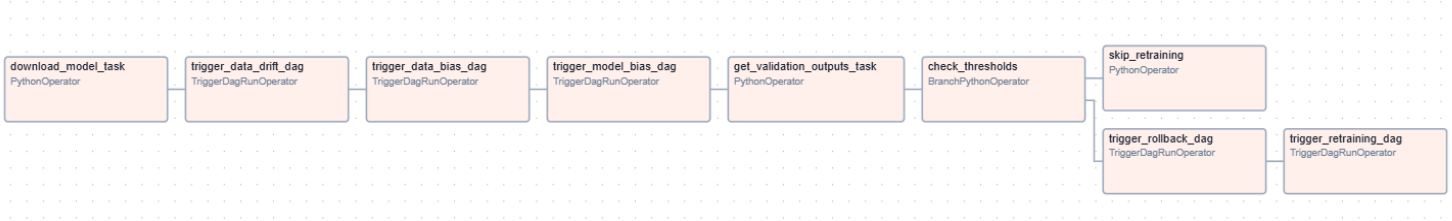
8) Retraining model DAG

9) Deploy model and deploy app DAGs

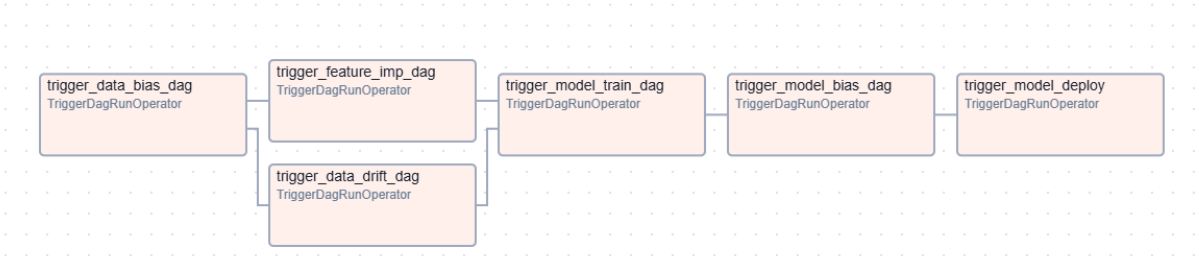




10) Deployment mentoring DAG



11) Master DAG

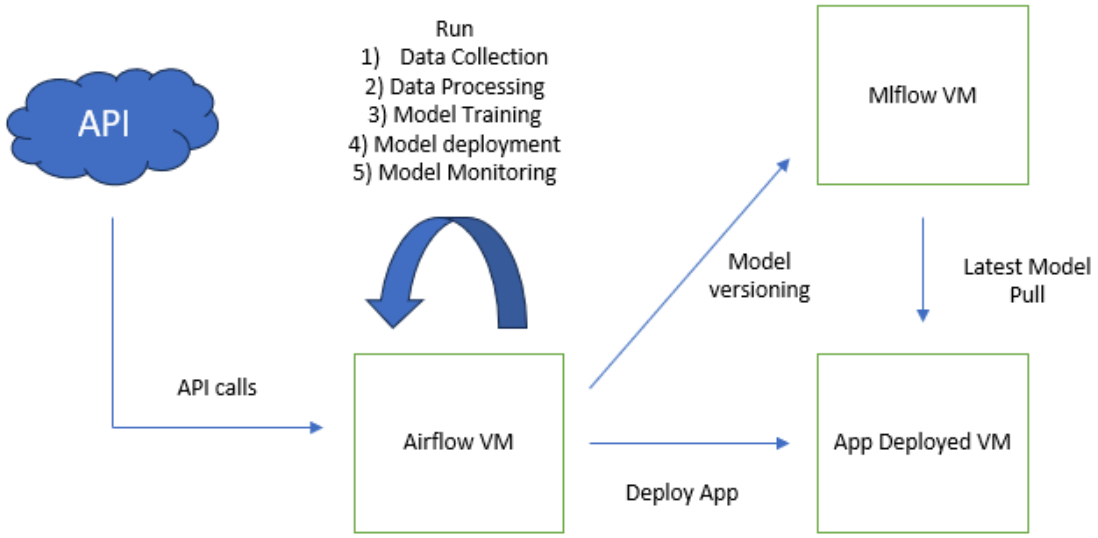


Cloud Deployment

Deployment Service - GCP Compute Engines: Airflow, MLflow, RAG Backend, and Frontend

The deployment of the energy demand forecasting and financial stock recommendation system is managed across three Google Cloud Platform Compute Engine instances. Each VM is dedicated to a specific service to ensure efficient resource allocation and manageability. The first VM hosts Airflow, which is responsible for orchestrating various tasks through DAGs. Airflow ensures the smooth execution of workflows with data collection, model training, evaluation, and deployment. The second VM is dedicated to the MLflow server, where the trained models and artifacts are stored, and the best-performing model is designated as the production model. The third VM is reserved for the RAG

(Retrieval-Augmented Generation) system and model inference, where the backend processes the energy demand prediction and financial queries, interacting with both the MLflow server and external APIs. These VMs work to deploy, maintain, and serve the application, ensuring that the system remains scalable and resilient.



Deployment Automation - GitHub Actions, DAGs, and Scripts

The deployment of the energy demand forecasting application and its components is automated through a combination of GitHub Actions, Airflow DAGs, and shell scripts. GitHub Actions is utilized for continuous integration and continuous deployment (CI/CD), ensuring that any changes made to the repository are automatically tested and deployed to the respective VMs. This enables fast and efficient deployment cycles, reducing manual intervention and speeding up the release of updates and fixes. Additionally, the `deploy.sh` script is an essential part of the automation, as it can be used independently or in conjunction with Airflow to deploy the application. The script SSHs into the appropriate VM, performs a git clone operation to pull the latest code, and runs the app containers. This method provides a streamlined approach for deploying updates across multiple environments. The third method for deploying the application involves using an Airflow DAG, which triggers the `deploy.sh` script.

Logging

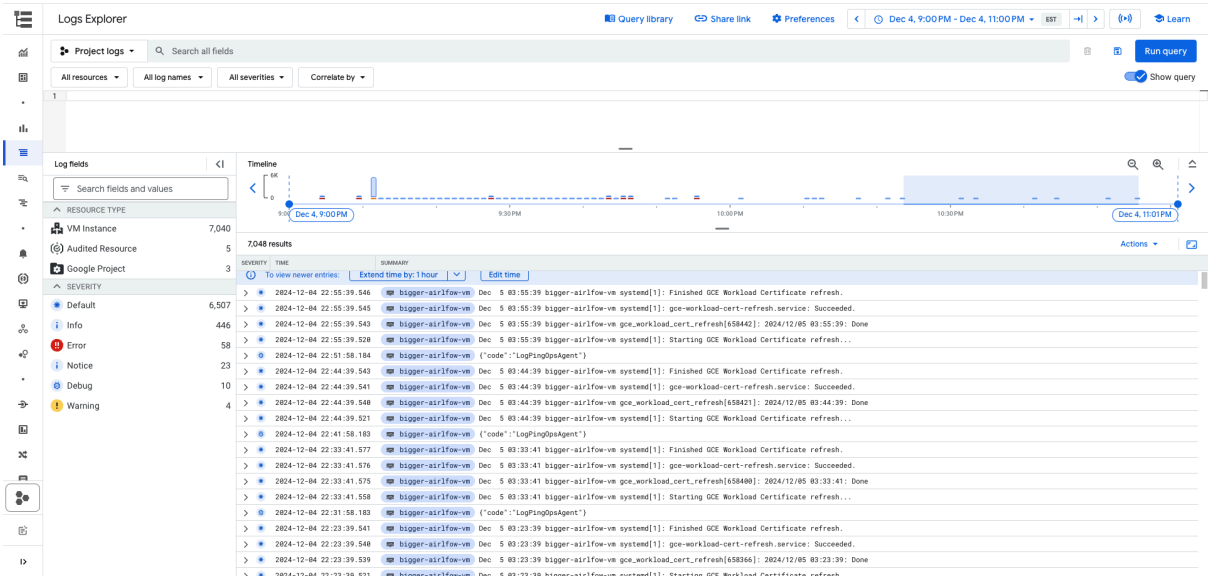
Logging is a critical component of our application, ensuring transparency, traceability, and maintainability across the pipeline. We used Google Cloud Logging for log monitoring. We implemented the Google Cloud Ops Agent service, which in turn uses an open-source log collector called Flunetd to capture logs from the VMs and send them to Google Cloud Logging.

The main Features of Logging are:

1. **Centralized Log Management:** All logs from VMs hosting Airflow and the application backend are centrally managed in Google Cloud Logging.
2. **Granular Logging**: Each DAG and its corresponding tasks generate logs to facilitate debugging, monitoring, and auditing.
3. **Structured and Queryable Logs**: Logs are structured in JSON format, making them easily searchable and filterable in the Google Cloud Logging Console.
   Ex:


```
{
        insertId: "1d66a0yfhb6qt5"
        jsonPayload: {
                line: "731"
                message: "::endgroup::"
                module: "taskinstance.py"
                severity: "INFO"
                timestamp: "2024-12-05T01:28:34.767+0000"
        }
        labels: {
                agent.googleapis.com/log_file_path:
                "/home/aakashmahesha/logs/dag_id=test_airflow_dag/run_id=man
                ual__2024-12-05T01:28:31.081543+00:00/task_id=print_hello/at
                tempt=1.log"
                compute.googleapis.com/resource_name: "bigger-airlfow-vm"
        }
        logName: "projects/tactical-runway-437316-k1/logs/airflow_logs"
        receiveTimestamp: "2024-12-05T02:11:59.213737644Z"
        resource: {
        labels: {
                instance_id: "3670832847913233795" (instance_name:
                bigger-airlfow-vm
                )
                project_id: "tactical-runway-437316-k1"
                zone: "us-central1-a"
        }
        type: "gce_instance"
        }
        timestamp: "2024-12-05T02:11:58.510141723Z"
}
```


4. **Real-time Monitoring**: Google Cloud Logging enables real-time monitoring of application health and task execution.



5. The **Google Cloud Ops Agent** is configured to collect logs from the following sources:
    1. **Airflow Task Logs**: Captures logs for each task in the DAGs.
    2. **Application Logs**: Includes backend and frontend container logs.
    3. **MLflow Logs**: Captures model training and inference logs.
    4. **System Logs**: Monitors VM health and resource utilization.

Repository connection

The setup-scripts/config.json contains the configuration details. Here the repository URL is configured along with the deployment VM details. The repository is connected to the demployment VM. On running setup-scripts/setup_vm.sh and setup-scripts/deploy_app.sh, the github code is cloned to the VM.

setup-scripts/remove.sh script is used to clean the VM. It deletes the repository from the VM. (https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/tree/dev/setup-scripts)

Replication steps

1) Reserve a VM

- Open GCP and reserve a compute engine with minimum 100GB persistent storage. You can reserve one with GPU, but not mandatory

2) Setup VM

- We are using password login instead of keys, which needs password setup.
- SSH into the VM and run `sudo nano /etc/ssh/sshd_config`
- Change to - PasswordAuthentication yes
- Next set the password and give root access to the user `sudo passwd username`
- We can then ssh using password. Setup sshpass (used to login with password)
- apt-get install sshpass && apt-get install jq

- Next install docker on the VM. For Debian OS -(https://docs.docker.com/engine/install/debian/
- Setup docker access
  `sudo usermod -aG sudo <USERNAME>`
  `sudo usermod -aG docker <USERNAME>`
  `newgrp docker`

3) Setup Environment and code
- Setup and update the credentials in **setup-scripts/config.json**
- Run setup_vm.sh to setup environment in the newly allocated vm- `cd setup-scripts && ./setup_vm.sh`

4) Deployment App
- Deployment can be done using deploy_app.sh, github action (CI/CD) or Airflow DAG

Manual Deployment
- Run deploy_app.sh to deploy and run the model and LLM - `cd setup-scripts && deploy_app.sh`
- Run deploy_model.sh to deploy just the model inside the LLM backend - `cd setup-scripts && deploy_model.sh`

Github Action Deployment
- Create a pull request with changes to the files - backend/app.py and backend/rag.py and deployment is triggered and merge to production

Using DAG
- Run the deployment_model_dag to deploy only the new ML model into the backend container
- Run the deployment_app_dag to deploy only entire backend LLM with new model inside the backend container

Model monitoring

Threshold - rollback and retraining

- **Download Model Artifacts**
  The DAG starts by downloading the latest model artifacts to evaluate the currently deployed model.

- **Data and Model Quality Checks**

  Data Drift Detection: Checks for significant changes in input data distribution.

Data Bias Detection and Mitigation: Identifies and mitigates biases in the input dataset.

Model Bias Detection and Mitigation: Ensures the model predictions are not biased.

- **Model Validation**
  Using the processed test dataset, the model is validated to extract key performance metrics such as:
  1. Mean Squared Error (MSE)
  2. Mean Absolute Error (MAE)
  3. R^2 Score

- **Threshold Verification**
  The extracted metrics are compared against predefined thresholds to decide whether the model needs retraining.
  1. If metrics are within thresholds: The pipeline skips retraining.
  2. If metrics fall outside thresholds: Retraining is triggered.

- **Conditional Retraining**
  If retraining is necessary, the DAG triggers a separate retraining and evaluation pipeline.

- **Rollback Logic**:
  If retraining is not possible or fails, the system can load a previous, stable model version from GCS or the model registry for redeployment.

Notifications

- Airflow is configured to send email notifications for key events in the pipeline, such as training completion, validation failure, or bias detection alerts. This ensures continuous monitoring and prompt action when necessary

Video Submission

https://drive.google.com/file/d/14j2F7C7ytSy3sLw8IUjoGD3pgAVxljje/view?usp=sharing

Code details

- Deployment -
  Scripts -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/tree/dev/setup-scripts

  DAGs -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/deployment_app_dag.py

https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/deployment_model_dag.py

- Github action scripts
  Airflow deployment -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/.github/workflows/airflow-workflow.yaml

  Model deployment -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/.github/workflows/deploy.yaml

- Connection to repository
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/setup-scripts/README.md

- Monitoring and retraining DAGs
  Deployment monitoring -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/deployed_monitoring_dag.py

  Model retraining DAG -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/model_retrain_evaluate_dag.py

  Master DAG -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/master_dag.py

- Env configuration
  Local env setup -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/master_dag.py

  VM env setup is handled by setup-scripts/setup_vm.sh script

  Airflow cfg -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/dags/master_dag.py

  Airflow server -
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/airflow-config/docker-compose.yaml
  https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/airflow-config/Dockerfile

Mlflow server interaction - https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/model/scripts/mlflow_model_registry.py

Logging - https://github.com/MLOPS-IE7374-Fall2024-G9/mlops-project/blob/dev/airflow-config/google-ops-agent-config.yaml