International Institute of Information Technology, Bangalore

Software Production Engineering Project Report

Project Title: Sentiment Analysis using MLOps

In the Guidance of: Prof. B. Thangaraju
Teaching Assistant: Shreyan

Submitted By:

Abhishek Kumar Singh (MT2024006)

Chinmay Tavarej (MT2024162)

Table Of Contents

1. Executive Summary

This comprehensive report delineates the meticulous design, sophisticated implementation, and seamless deployment of a cutting-edge sentiment analysis application, meticulously engineered to harness robust Machine Learning Operations (MLOps) practices. The system is architected to leverage advanced machine learning techniques to accurately predict the sentiment expressed within textual data, with a particular focus on datasets such as movie reviews, customer feedback, and other unstructured text corpora. By integrating a holistic, end-to-end lifecycle approach, this project exemplifies the modern paradigm of machine learning systems, encompassing critical stages such as data acquisition and preprocessing, model development and training, experiment tracking and management, versioning of data and models, containerization for portability, continuous integration and continuous deployment (CI/CD) automation, and scalable production deployment on cloud-based infrastructure.

The primary objective of this initiative is to bridge the traditional gap between data science prototyping and production-ready systems, addressing common challenges such as lack of reproducibility, scalability, and maintainability in machine learning workflows. To achieve this, the project employs a suite of state-of-the-art open-source tools and cloud-native services, ensuring that every component of the system is optimized for reliability, efficiency, and extensibility. Key among these tools is MLflow, which serves as the cornerstone for experiment tracking, enabling data scientists to log parameters, metrics, and artifacts systematically, thereby facilitating comparison and reproducibility of model runs. DVC (Data Version Control) is utilized to manage versioning of datasets, models, and pipeline artifacts, ensuring that all data transformations and model iterations are traceable and reproducible, akin to code version control systems like Git.

Containerization, achieved through Docker, plays a pivotal role in encapsulating the application and its dependencies into portable, consistent environments, mitigating the "it works on my machine" problem and ensuring seamless transitions from development to production. Ansible, a powerful automation tool, is employed to streamline infrastructure provisioning and deployment processes, automating the configuration of Kubernetes clusters and other cloud resources with precision and repeatability. The project is deployed on Amazon Web Services (AWS), leveraging services such as Amazon Elastic Kubernetes Service (EKS) for container orchestration, Amazon Elastic Container Registry (ECR) for secure storage of Docker images, and Amazon Simple Storage Service (S3) for robust data and model artifact storage. These AWS services collectively provide a scalable, high-availability infrastructure capable of handling dynamic workloads and ensuring fault tolerance.

The CI/CD pipeline, orchestrated via GitHub Actions, automates the entire workflow from code commits to production deployment. This includes automated testing of code and models, linting for code quality, building and pushing Docker images to ECR, and deploying updates to the Kubernetes cluster on EKS. This automation reduces manual intervention, minimizes human error, and accelerates the release cycle, enabling rapid iteration and deployment of new model versions or application enhancements.

By integrating these advanced MLOps practices and tools, the project not only delivers a high-performing sentiment analysis model but also establishes a blueprint for scalable, reproducible, and maintainable machine learning systems. The use of open-source tools ensures cost-effectiveness and community-driven innovation, while AWS's cloud-native services provide the robustness and scalability required for enterprise-grade deployments. This initiative serves as a testament to the power of MLOps in transforming machine learning from an experimental endeavor into a reliable, production-ready discipline, capable of delivering actionable insights from complex textual data with unparalleled efficiency and precision.

2. Project Introduction

The rapid proliferation of digital platforms has ushered in an era of unprecedented data generation, particularly in the form of unstructured text from sources such as social media, e-commerce platforms, and review aggregators. This project focuses on the development and deployment of a sophisticated sentiment analysis application, designed to extract meaningful insights from textual data by predicting sentiments expressed in contexts like movie reviews, customer feedback, and public discourse. By leveraging advanced machine learning techniques and robust Machine Learning Operations (MLOps) practices, this initiative aims to create a scalable, reproducible, and production-ready system that exemplifies the end-to-end lifecycle of modern machine learning workflows. The project integrates a suite of cutting-edge tools, including MLflow for experiment tracking, DVC for data and model versioning, Ansible for infrastructure automation, and containerization via Docker, all deployed on Amazon Web Services (AWS) infrastructure to ensure reliability and scalability.

2.1 Background and Motivation

The exponential growth of unstructured text data across the internet has transformed sentiment analysis into a cornerstone of data-driven decision-making for organizations worldwide. Businesses, governments, and research institutions increasingly rely on sentiment analysis to gauge customer satisfaction, monitor brand perception, analyze product reviews, and understand public opinion on social and political issues. For instance, analyzing sentiments in movie reviews can help studios predict box office performance, while sentiment analysis of customer feedback enables companies to refine products and services. However, the complexity of processing vast volumes of text data and deploying predictive models at scale presents significant challenges. Traditional machine learning approaches often excel in controlled environments but struggle to transition into production due to issues such as inconsistent environments, lack of reproducibility, and manual deployment processes.

This project is motivated by the need to address these challenges through the adoption of robust MLOps practices, which streamline the journey from model prototyping to production deployment. MLOps, as a discipline, integrates data engineering, machine learning, and DevOps principles to ensure that models are not only accurate but also reliable, scalable, and maintainable in real-world settings. By employing tools like MLflow to track experiments and DVC to version data and models, the project ensures that every stage of the machine learning lifecycle is traceable and reproducible. Furthermore, the use of containerization with Docker and orchestration via Kubernetes on AWS infrastructure guarantees portability and scalability, enabling the system to handle dynamic workloads and serve predictions efficiently. The motivation behind this project is to create a blueprint for organizations seeking to harness the power of sentiment analysis while overcoming the operational hurdles that often impede successful deployment.

2.2 Problem Statement

The deployment of machine learning models, particularly for applications like sentiment analysis, is frequently hindered by systemic shortcomings in traditional workflows. Many machine learning projects falter at the production stage due to a lack of automation, inadequate tools for collaboration, and insufficient mechanisms to ensure reproducibility. For example, manual data preprocessing and model training processes can lead to inconsistencies, while unversioned datasets and models make it difficult to replicate results or roll back to previous iterations. Additionally, the absence of automated testing and deployment pipelines increases the risk of errors and delays, rendering systems unreliable under production workloads. These challenges are compounded when scaling to handle large datasets or high-traffic prediction requests, where traditional infrastructure often fails to provide the necessary robustness and elasticity.

This project seeks to address these critical issues by developing a fully automated, scalable, and reproducible pipeline for sentiment analysis, underpinned by modern MLOps tools and cloud infrastructure. The pipeline leverages DVC to version datasets and model artifacts, ensuring that all data transformations and training runs are traceable and reproducible. MLflow is employed to log experiment parameters, metrics, and outputs, facilitating systematic comparison and optimization of models. Automation is further enhanced through Ansible, which streamlines the provisioning and configuration of cloud resources, and GitHub Actions, which orchestrates continuous integration and continuous deployment (CI/CD) workflows. By deploying the system on AWS, utilizing services such as Amazon Elastic Kubernetes Service (EKS) for container orchestration and Amazon Elastic Container Registry (ECR) for Docker image storage, the project ensures that the sentiment analysis application can scale seamlessly to meet growing demands while maintaining high availability and fault tolerance. The overarching goal is to create a robust framework that not only delivers accurate sentiment predictions but also sets a standard for operational excellence in machine learning deployments.

2.3 Objectives

The project is guided by a set of well-defined objectives, each designed to address specific aspects of the machine learning lifecycle and ensure the successful deployment of a production-ready sentiment analysis system. These objectives are:

- Develop a highly accurate sentiment analysis machine learning model capable of classifying text data, such as movie reviews, into positive, negative, or neutral categories. The model will leverage algorithms like Logistic Regression, implemented via Scikit-learn, and incorporate advanced feature engineering techniques, such as TF-IDF vectorization, to maximize predictive performance.

- Implement automated data pipelines to streamline the ingestion, preprocessing, and feature engineering of text data, ensuring efficiency and consistency. Experiment tracking will be facilitated by MLflow, which logs parameters, metrics, and artifacts, enabling data scientists to iterate rapidly and compare model performance systematically.
- Ensure reproducibility across all stages of the project through rigorous version control and containerization. DVC will be used to version raw and processed datasets, model artifacts, and pipeline configurations, while Docker will encapsulate the application and its dependencies into portable containers, ensuring consistency across development, testing, and production environments.
- Automate CI/CD workflows to facilitate seamless transitions from development to deployment. GitHub Actions will orchestrate automated testing, linting, model training, and deployment processes, while Ansible will automate the provisioning and configuration of Kubernetes clusters on AWS, reducing manual intervention and minimizing errors.
- Deploy scalable, production-ready APIs using Kubernetes on AWS infrastructure. The sentiment analysis model will be served via a Flask-based REST API, containerized with Docker, and orchestrated by Amazon EKS to ensure high availability and scalability. AWS services like ECR and S3 will support secure storage of Docker images and data artifacts, respectively, enabling a robust and extensible deployment framework.

By achieving these objectives, the project aims to deliver a state-of-the-art sentiment analysis system that not only meets the technical demands of accuracy and performance but also adheres to best practices in MLOps, ensuring reliability, scalability, and maintainability in production environments.

3. Literature Review / Related Work

The development of a sentiment analysis application with robust MLOps practices builds upon a rich body of research and industry standards in machine learning, natural language processing (NLP), and cloud deployment. This section reviews key literature and documentation related to sentiment analysis, MLOps methodologies, and cloud-native deployment, with specific references to the tools used in this project—MLflow, DVC, Ansible, Docker, and AWS services (EKS, ECR, S3). Drawing on the developer's prior experience building a sentiment analysis application, this review contextualizes the project within established practices and highlights how it advances them.

Sentiment analysis, a subfield of NLP, has been extensively studied for its ability to classify text as positive, negative, or neutral. Early work by Pang et al. (2002) demonstrated the efficacy of machine learning techniques, such as Naive Bayes and Support Vector Machines, for sentiment classification of movie reviews, laying the groundwork for modern approaches. More recent advancements leverage feature engineering techniques like TF-IDF vectorization, as implemented in Scikit-learn, which remains a standard for transforming text into numerical features (Scikit-learn Documentation, 2025). The developer's previous sentiment analysis application utilized similar techniques, confirming their effectiveness for text classification tasks. However, these models require robust deployment frameworks to transition from research to production, a challenge addressed by MLOps.

MLOps, defined as the intersection of machine learning, DevOps, and data engineering, has emerged as a critical discipline for operationalizing ML models. Kubeflow's documentation (2025) emphasizes the need for reproducible pipelines, a principle adopted in this project through DVC. DVC's documentation (2025) outlines its role in versioning datasets and models, enabling traceability akin to Git for code. This aligns with research by Amershi et al. (2019), who identified reproducibility and automation as key MLOps challenges at large-scale organizations. MLflow, another cornerstone of this project, is designed to address experiment tracking and model management, as detailed in its documentation (2025). By logging parameters, metrics, and artifacts, MLflow facilitates systematic model iteration, a practice informed by the developer's prior experience where manual tracking hindered scalability.

Cloud deployment, particularly with containerization and orchestration, is central to scaling ML applications. Docker's documentation (2025) highlights its ability to create portable, consistent environments, mitigating dependency issues encountered in the developer's earlier sentiment analysis work. Kubernetes, as described in its documentation (2025), enables orchestration of containerized applications, ensuring scalability and fault tolerance. AWS services, including Amazon Elastic Kubernetes Service (EKS) for cluster management, Elastic Container Registry (ECR) for image storage, and Simple Storage Service (S3) for data persistence, provide a robust cloud infrastructure (AWS EKS Documentation, 2025). These align with industry standards outlined by Armbrust et al. (2020), who advocate for cloud-native architectures to handle dynamic workloads.

Automation is another critical MLOps pillar. Ansible's documentation (2025) describes its role in automating infrastructure provisioning, which this project uses to configure Kubernetes clusters. GitHub Actions, as per its documentation (2025), supports CI/CD pipelines, automating testing and deployment. These practices draw from DevOps principles formalized by Bass et al. (2015), emphasizing continuous delivery to reduce deployment risks.

This project synthesizes these insights, building on the developer's prior sentiment analysis application by integrating advanced MLOps tools and cloud deployment strategies. Unlike earlier efforts, which lacked automation and scalability, this work leverages DVC and MLflow for reproducibility, Docker and Kubernetes for portability, and AWS and Ansible for robust deployment, aligning with current research and industry best practices.

4. Project Scope and Requirements

4.1 Functional Requirements

- ✓ Ingest and preprocess raw text data.
- ✓ Perform feature engineering for ML models.
- ✓ Train and evaluate sentiment analysis models.
- ✓ Serve predictions via a REST API.
- ✓ Track experiments, version data, and models.
- ✓ Automate deployment to cloud infrastructure.

4.2 Non-functional Requirements

- ✓ Reproducibility of results.
- ✓ Scalability to handle growing data and traffic.
- ✓ Security of data and model artifacts.
- ✓ Maintainability and ease of extension.

4.3 System Constraints

- ✓ Python 3.10 as the primary language.
- ✓ AWS as the cloud provider (EKS, ECR, S3).
- ✓ Open-source tooling for MLOps.

5. Architecture Overview

- ✓ **Data Pipeline**: The data pipeline automates the download, cleaning, and feature extraction of text data like movie reviews. It uses Pandas and NLTK for preprocessing and DVC to version raw and processed datasets, ensuring reproducibility. Data is synced with AWS S3 for secure storage and collaboration.
- ✓ **Model Pipeline**: This pipeline trains and evaluates sentiment analysis models, such as Logistic Regression, using Scikit-learn. MLflow tracks experiments by logging parameters, metrics, and artifacts, enabling comparison of model runs. DVC versions model artifacts for traceability and reproducibility.
- ✓ **API Layer**: A Flask-based service exposes a REST API endpoint (/predict) to deliver sentiment predictions. The API loads the latest model from DVC-versioned artifacts stored in AWS S3. Gunicorn ensures production-ready performance for the containerized application.
- ✓ **CI/CD Pipeline**: The CI/CD pipeline, built with GitHub Actions, automates code testing, model training, and Docker containerization. It pushes images to AWS ECR and deploys updates to Kubernetes via Ansible. This ensures rapid, error-free transitions from development to production.
- ✓ **Orchestration**: Kubernetes, managed by AWS EKS, orchestrates containerized application deployment and scaling. It ensures high availability and load balancing for the Flask API. Ansible automates cluster configuration for seamless setup and maintenance.
- ✓ **Cloud Integration**: AWS EKS serves as the Kubernetes orchestrator, while ECR stores Docker images securely. S3 manages data and model storage, enabling scalable access. This cloud-native setup supports robust, high-performance deployment.

6. Technologies Stack

Category	Technology	Purpose
Language	Python 3.10	Primary development language
ML Libraries	Scikit-learn	ML algorithms, vectorization, model evaluation
	Pandas, NumPy	Data manipulation and numerical operations
	NLTK	NLP preprocessing
MLOps Tools	DVC	Data/model versioning, pipeline automation
	MLflow	Experiment tracking, model management
Application/API	Flask, Gunicorn	REST API, WSGI server for production
Containerization	Docker	Containerizes app for consistency and portability
Orchestration	Kubernetes	Container orchestration, scaling, service exposure
Cloud Services	AWS EKS, ECR	Managed Kubernetes, container registry
Automation	Ansible	EKS deployment automation
Monitoring	Prometheus and Grafana	Metrics collection and visualization

7. Data Pipeline

The data pipeline, defined in dvc.yaml, automates the ingestion, preprocessing, feature engineering, and storage of text data for sentiment analysis, using tools like DVC, Pandas, NLTK, Scikit-learn, and AWS S3. It consists of stages (data_ingestion, data_preprocessing, feature_engineering) that process movie review data, with parameters from params.yaml ensuring reproducibility. Outputs are versioned by DVC and stored locally or synced with S3 for collaboration.

7.1 Data Ingestion

The data_ingestion stage, executed by data_ingestion.py, fetches a CSV file containing movie reviews from an AWS S3 bucket using the s3_operations module with environment variable-based credentials. It filters for positive/negative sentiments, encodes them (positive=1, negative=0), and splits the data into train (75%) and test (25%) sets based on test_size from params.yaml. The resulting train.csv and test.csv are saved in data/raw, versioned by DVC.

7.2 Data Preprocessing

The data_preprocessing stage, implemented in data_preprocessing.py, loads train.csv and test.csv from data/raw using Pandas. It applies NLTK-based text cleaning to the review column, including URL/number removal, lowercase conversion, punctuation/stoplord elimination, and lemmatization. Processed data is saved as train_processed.csv and test_processed.csv in data/interim, tracked by DVC.

7.3 Feature Engineering

The feature_engineering stage, run by feature_engineering.py, transforms preprocessed text into numerical features using Scikit-learn's CountVectorizer with a max_features limit (50) from params.yaml. It generates Bag of Words (BoW) representations, saving them as train_bow.csv and test_bow.csv in data/processed, and stores the vectorizer as vectorizer.pkl in models. Both outputs are versioned by DVC for reproducibility.

7.4 Data Versioning

DVC manages versioning of pipeline artifacts, including data/raw, data/interim, data/processed, and vectorizer.pkl, as specified in dvc.yaml. Dependencies (e.g., scripts, input data) and parameters (test_size: 0.25, max_features: 50) from params.yaml ensure each stage is reproducible. DVC tracks changes, enabling pipeline execution with dvc repro.

7.5 Data Storage and Management

Data is stored locally in data/raw, data/interim, and data/processed directories, with DVC handling versioning. AWS S3 serves as the remote storage for raw data and backups, accessed via s3_operations in data_ingestion.py. Makefile targets automate S3 synchronization, ensuring secure and collaborative data management.

Later using DVC push the hashes are stored in s3 also.

The pipeline also includes code for model building, evaluation and registering.

7.6 Training and Validation Process

The model_building stage, executed by model_building.py, trains a Logistic Regression model on train_bow.csv from data/processed. It uses Pandas to load data and Scikit-learn to fit the model, saving the trained model as model.pkl in models, versioned by DVC. The training process is reproducible due to fixed random states and DVC-tracked inputs.

7.7 Experiment Tracking

The model_evaluation stage, run by model_evaluation.py, logs experiment metrics (accuracy, precision, recall, AUC) to MLflow, hosted on DagsHub. MLflow tracks parameters, metrics, and the model artifact, with credentials managed via environment variables. Metrics are saved as metrics.json in reports, and experiment details are stored in experiment_info.json, both versioned by DVC.

7.8 Hyperparameter Tuning

Hyperparameters (C=1, penalty='l1', solver='liblinear') are hardcoded in model_building.py for simplicity, based on prior experimentation. Future iterations could leverage MLflow's tracking to perform grid search over C and penalty values. The current settings balance model performance and training efficiency.

7.9 Evaluation Metrics and Results

The model_evaluation.py script evaluates the model on test_bow.csv, computing accuracy, precision, recall, and AUC using Scikit-learn's metrics. Results are logged to MLflow and saved as metrics.json, enabling performance comparison across runs. A confusion matrix could be added to visualize classification errors, as supported by Scikit-learn.

7.10 Model Versioning

DVC versions the trained model (model.pkl) and evaluation outputs (metrics.json, experiment_info.json) as specified in dvc.yaml. The model_registration stage, executed by register_model.py, registers the model in MLflow’s Model Registry on DagsHub, transitioning it to the “Staging” stage. This ensures traceability and facilitates deployment.

8. Model Development Experimentations and tracking using MLFLOW

The model development phase involved extensive experimentation to build and optimize a sentiment analysis model, leveraging MLflow for experiment tracking, Scikit-learn for model training, and DVC for versioning. Experiments, conducted in Jupyter notebooks and Python scripts, explored various algorithms, feature engineering techniques, and hyperparameter configurations, with results logged to MLflow hosted on DagsHub. The pipeline stages (model_building, model_evaluation, model_registration) from dvc.yaml automated the final model's training, evaluation, and registration, ensuring reproducibility and scalability.

8.1 Initial Experimentation: Logistic Regression Baseline

The first experiment, detailed in exp1_logisticRegression.ipynb, established a baseline using Logistic Regression on a 500-sample subset of the IMDB dataset (data.csv). Text preprocessing included lowercasing, stopword removal, number/punctuation/URL removal, and lemmatization, followed by CountVectorizer (100 features) for Bag of Words (BoW). The model, trained with max_iter=1000, achieved metrics (accuracy, precision, recall, F1) logged to MLflow under the "Logistic Regression Baseline" experiment, providing a reference for subsequent experiments.

8.2 Multi-Model and Vectorizer Comparison

The experiment in exp2_VariousModels.py compared multiple algorithms (Logistic Regression, SVM, Random Forest) with BoW and TF-IDF vectorizers on the preprocessed IMDB dataset. Each model-vectorizer combination was trained on a 80-20 train-test split, with metrics (accuracy, precision, recall, F1) and model parameters (e.g., C for Logistic Regression, n_estimators for Random Forest) logged to MLflow's "BOW vs TFIDF with Various Models" experiment. Results showed Logistic Regression with BoW as a strong performer, balancing accuracy and simplicity.

8.3 Hyperparameter Tuning

The exp3_hyperparameterTuning.py script optimized Logistic Regression using GridSearchCV over C ([0.1, 1, 10]), penalty ([l1, l2]), and solver (liblinear) with TF-IDF features. Each parameter combination's F1 score and metrics were logged as nested runs in MLflow's "LoR Hyperparameter Tuning" experiment, with the best model (C=1, penalty=l2) logged separately. This experiment confirmed the baseline's parameters were near-optimal, guiding the final model selection.

8.4 Final Model Selection and Justification

Based on experiments, Logistic Regression with BoW was selected for the production pipeline, implemented in model_building.py with C=10, penalty=l1, and solver=liblinear. Its interpretability, efficiency on sparse features, and consistent performance (validated in exp1 and exp2) justified its choice over more complex models like SVM or Random Forest. The model is trained on train_bow.csv and saved as model.pkl, versioned by DVC.

8.5 Training and Validation Process

The model_building stage, executed by model_building.py, trains the Logistic Regression model on train_bow.csv from data/processed using Scikit-learn. The model_evaluation stage, run by model_evaluation.py, validates the model on test_bow.csv, computing accuracy, precision, recall, and AUC. Metrics and model artifacts are logged to MLflow and saved as metrics.json and experiment_info.json, versioned by DVC.

8.6 Experiment Tracking

MLflow, integrated with DagsHub, tracked all experiments, logging parameters (e.g., vectorizer type, C), metrics, and artifacts to the "my-dvc-pipeline" experiment for the final pipeline. The model_evaluation.py script logs metrics and model details, while exp1, exp2, and exp3 experiments provide comprehensive run histories. DagsHub's MLflow UI enables visualization and comparison of runs, ensuring transparency.

8.7 Hyperparameter Tuning in Pipeline

The production pipeline uses fixed hyperparameters (C=1, penalty=l1) in model_building.py, informed by exp3_hyperparameterTuning.py. MLflow's tracking in model_evaluation.py supports future tuning by logging parameters, allowing grid search integration if needed. This approach balances simplicity and extensibility.

8.8 Evaluation Metrics and Results

The final model, evaluated in model_evaluation.py, reports accuracy, precision, recall, and AUC on test_bow.csv, stored in metrics.json. Experiments (exp1, exp2, exp3) consistently showed F1 scores above 0.8, with Logistic Regression outperforming others in exp2. Results are visualized in MLflow's UI, aiding model selection.

8.9 Model Versioning and Registration

DVC versions model.pkl, metrics.json, and experiment_info.json as specified in dvc.yaml. The model_registration stage, executed by register_model.py, registers the model in MLflow's Model Registry on DagsHub, transitioning it to "Staging". This ensures traceability and readiness for deployment.

9. ML Application/API Development

The ML application, implemented in app.py, is a Flask-based REST API that delivers sentiment predictions for text inputs, seamlessly integrating with MLflow for model management, DVC for artifact versioning, Prometheus for performance monitoring, and a CI/CD pipeline for automated testing and deployment. Designed to handle production workloads, the API preprocesses input text using NLTK, loads the latest Logistic Regression model and CountVectorizer, and serves predictions through a user-friendly web interface and programmatic endpoints. Deployed in a Docker container and orchestrated by Kubernetes on AWS EKS, the application ensures scalability, reliability, and observability, aligning with MLOps best practices.

9.1 Flask API Structure

The Flask application, defined in app.py, is structured to provide a robust API with three endpoints: /predict for sentiment prediction, /metrics for Prometheus monitoring, and / for a web interface. It employs NLTK-based text preprocessing functions (normalize_text, lemmatization, remove_stop_words, etc.) to clean input text, ensuring consistency with the training pipeline in data_preprocessing.py. In production, Gunicorn serves the API, while Prometheus metrics (REQUEST_COUNT, REQUEST_LATENCY, PREDICTION_COUNT) track request volume, latency, and prediction outcomes, enabling real-time performance monitoring.

9.2 Model Serialization and Loading

Model artifacts are serialized and loaded from versioned sources to ensure consistency and reproducibility. The Logistic Regression model is retrieved from MLflow's Model Registry on DagsHub using the get_latest_model_version function, which prioritizes the latest "Production" model version or falls back to "None" if unavailable, accessed via models:/my_model/{version}. The CountVectorizer, stored as models/vectorizer.pkl and versioned by DVC, is loaded using pickle.load, enabling text-to-feature transformation compatible with the model trained in model_building.py.

9.3 REST API Endpoints

Endpoint	Method	Description
/predict	POST	Returns sentiment
/	GET	Renders the index.html template, providing a web interface for users to input text and view sentiment predictions.
/metices	GET	Health check endpoint

The /predict endpoint uses the normalize_text function to apply lowercasing, stopword removal, number/punctuation/URL removal, and lemmatization, ensuring input compatibility with the model's training data. The /metrics endpoint leverages the prometheus_client library to generate metrics in the OpenMetrics format, accessible via generate_latest(registry).

9.4 Input/Output Schemas

The /predict endpoint accepts input as form data with a text field (e.g., text=Great movie!) or JSON in API-only configurations (e.g., {"text": "Great movie!"}). The input text is preprocessed using the normalize_text function, transformed into numerical features by vectorizer.transform, and passed to the model for prediction, ensuring robust handling of varied inputs. The output is a rendered index.html template displaying the predicted sentiment (0 or 1), with potential JSON output (e.g., {"sentiment": 1}) for programmatic access, and error handling for invalid inputs is implicit in the preprocessing pipeline.

9.5 Testing the API

The API is rigorously tested using unit and integration tests defined in tests/test_flask_app.py, executed via unittest in the CI/CD pipeline configured in GitHub Actions. Tests validate the /predict endpoint's prediction accuracy, input validation, and response format, as well as the /metrics endpoint's Prometheus metric output, using the CAPSTONE_TEST environment variable for DagsHub authentication. The CI/CD pipeline step (Run Flask app tests) ensures tests run automatically on code commits, guaranteeing API reliability before deployment to Kubernetes.

10. Containerization

Containerization encapsulates the sentiment analysis Flask application, its dependencies, and model artifacts into a Docker container, ensuring consistent deployment across environments. The Dockerfile defines a lightweight, production-ready image based on Python 3.10-slim, integrating the Flask app (app.py), DVC-versioned artifacts, and Prometheus monitoring, with Gunicorn as the server. Built images are tested locally, pushed to AWS Elastic Container Registry (ECR), and orchestrated by Kubernetes on AWS EKS, aligning with MLOps principles for scalability and reproducibility.

10.1 Dockerfile Walkthrough

The Dockerfile uses python:3.10-slim as the base image to minimize size and vulnerabilities, setting /app as the working directory. It copies the Flask application (flask_app/), including app.py, and the DVC-versioned models/vectorizer.pkl, installs dependencies from requirements.txt (e.g., Flask, MLflow, NLTK, Prometheus), and downloads NLTK data (stopwords, wordnet) for text preprocessing. The container exposes port 5000 and runs Gunicorn with the command gunicorn --bind 0.0.0.0:5000 --timeout 120 app:app, ensuring robust production performance for the Flask API.

10.2 Building and Testing Containers

The container is built using docker build -t sentiment-app ., creating an image tagged sentiment-app that encapsulates the Flask app, model, and dependencies. Local testing involves running docker run -p 5000:5000 sentiment-app to verify the API's /predict, /metrics, and / endpoints, ensuring predictions and Prometheus metrics function correctly. After validation, the image is pushed to AWS ECR using docker push <aws_account_id>.dkr.ecr.<region>.amazonaws.com/sentiment-app:latest, integrated into the CI/CD pipeline for automated deployment to Kubernetes on AWS EKS.

10.3 Security Best Practices

The Dockerfile employs a minimal python:3.10-slim base image to reduce attack surfaces, avoiding unnecessary packages and leveraging Debian's security-patched slim variant. While multi-stage builds are not currently used, they could further optimize the image by separating build and runtime environments; instead, requirements.txt is pinned to specific versions to avoid dependency vulnerabilities.

11. Deployment & Orchestration

The deployment and orchestration phase operationalizes the sentiment analysis Flask application using Kubernetes on AWS Elastic Kubernetes Service (EKS), with Docker images stored in AWS Elastic Container Registry (ECR). The deployment.yaml and playbook.yml files define the Kubernetes resources and Ansible automation, respectively, ensuring scalable, high-availability deployment of the API (app.py). Integrated with Prometheus for monitoring and MLflow/DVC for model management, the setup aligns with MLOps best practices, automating deployment and enabling robust observability.

11.1 Kubernetes Deployment

The deployment.yaml file defines a Kubernetes Deployment named flask-app in the default namespace, running two replicas of the Flask application for redundancy and load balancing. The deployment pulls the Docker image 065948377643.dkr.ecr.us-east-1.amazonaws.com/spemlopsecr:latest from ECR, as specified in the container spec, and allocates resources (256Mi memory request, 512Mi limit; 250m CPU request, 1 CPU limit) to ensure performance stability. The CAPSTONE_TEST environment variable, used for DagsHub/MLflow authentication, is injected via a Kubernetes Secret (capstone-secret), securing access to the MLflow Model Registry.

11.2 Service Exposure

A Kubernetes Service of type LoadBalancer, also defined in deployment.yaml, exposes the Flask API's port 5000 to the internet through an AWS Elastic Load Balancer (ELB). The service targets pods labeled app: flask-app, routing traffic to the /predict, /metrics, and / endpoints of the Flask application. This configuration ensures external accessibility for the sentiment prediction API and Prometheus metrics, with ELB providing load distribution across replicas.

11.3 AWS EKS Cluster Setup

The AWS EKS cluster is provisioned to host the Kubernetes deployment, configured for high availability across multiple availability zones in the us-east-1 region. Managed node groups enable autoscaling, dynamically adjusting the number of worker nodes based on workload demands, ensuring the Flask API remains responsive under varying traffic. The cluster integrates with AWS IAM for secure access control, with kubectl and the AWS CLI used to manage resources, as automated by the CI/CD pipeline.

11.4 AWS ECR Integration

Docker images, built from the Dockerfile (Section 10, artifact_id: ea913fff-e403-4872-9d31-32c6fd94fae8), are stored in AWS ECR under the repository spemlopsecr

(065948377643.dkr.ecr.us-east-1.amazonaws.com). The CI/CD pipeline automates image building and pushing using docker push, with Kubernetes pods pulling the latest tag during deployment, as specified in deployment.yaml. ECR's IAM-based authentication ensures secure image access, with image scanning enabled to detect vulnerabilities before deployment.

11.5 Ansible Automation

The playbook.yml file defines an Ansible playbook (Deploy Flask App to EKS) that automates the deployment of Kubernetes resources to the EKS cluster using the kubernetes.core collection. The eks_deploy role applies the deployment.yaml configuration, managing the creation and updating of the Deployment and Service resources. This automation, integrated into the CI/CD pipeline, ensures consistent and repeatable deployments, reducing manual errors and streamlining updates to the Flask API.

11.6 Monitoring and Logging

Prometheus collects metrics exposed by the Flask application's /metrics endpoint, implemented in app.py using prometheus_client to track request counts (REQUEST_COUNT), latency (REQUEST_LATENCY), and prediction outcomes (PREDICTION_COUNT). These metrics are scraped by a Prometheus server deployed in the EKS cluster, enabling real-time monitoring and alerting for performance issues. Logging is facilitated by Kubernetes' built-in logging, capturing container logs from the Flask app, with potential integration of AWS CloudWatch for centralized log management.

12. CI/CD Pipeline

The CI/CD pipeline, defined in ci.yaml, is a GitHub Actions workflow that automates the testing, building, and deployment of the sentiment analysis application, ensuring seamless delivery to an AWS EKS cluster. Running in the spemlops GitHub Actions environment, it triggers on code pushes and executes 12 meticulously orchestrated steps, leveraging GitHub Secrets for secure credential management to access DagsHub, AWS S3, ECR, and EKS. The pipeline integrates DVC for data and model versioning, unittest for testing, Docker for containerization, and Ansible for Kubernetes deployment, with rollback and recovery mechanisms to maintain reliability, embodying MLOps principles for continuous integration and deployment.

12.1 GitHub Actions Workflow Structure

The CI Pipeline workflow, specified in ci.yaml, activates on every push event to the repository, running on an ubuntu-latest runner within the spemlops GitHub Actions environment, which provides a controlled context for secret scoping and deployment approvals. The workflow comprises 12 sequential steps: (1) code checkout, (2) Python 3.10 setup, (3) pip dependency caching, (4) dependency installation, (5) DVC pipeline execution, (6) model testing, (7) model promotion, (8) Flask app testing, (9) AWS credential configuration, (10) ECR login, (11) Docker image building/tagging/pushing, and (12) Ansible-driven EKS deployment. Steps 7, 8, and 10–12 are conditional (if: success()), ensuring they execute only if prior steps succeed, minimizing the risk of deploying faulty code or images; credentials are injected via GitHub Secrets, scoped to the spemlops environment, for secure access to external services.

- **Step 1: Checkout Code** (uses: actions/checkout@v3): Uses the actions/checkout@v3 action to clone the repository, ensuring the latest code, including app.py, Dockerfile, and ansible/playbook.yml, is available. No credentials are required, as GitHub Actions provides implicit access to the repository. This step sets the foundation for subsequent tasks.
- **Step 2: Setup Python** (uses: actions/setup-python@v2): Configures Python 3.10 using actions/setup-python@v2, aligning with the python:3.10-slim base in the Dockerfile. No credentials are needed, but the step ensures compatibility with dependencies in requirements.txt. The spemlops environment context ensures consistent runner configuration.
- **Step 3: Cache pip Dependencies** (uses: actions/cache@v3): Caches pip dependencies from ~/.cache/pip using actions/cache@v3, with a cache key based on runner.os and the hash of requirements.txt. This reduces installation time for dependencies like

Flask, MLflow, and NLTK, requiring no credentials but leveraging the spemlops environment for runner consistency.

- **Step 4: Install Dependencies** (run: pip install -r requirements.txt): Installs dependencies listed in requirements.txt (e.g., Flask, MLflow, prometheus_client, scikit-learn, nltk) using pip. No credentials are needed, as the step operates locally on the runner. The cached dependencies from Step 3 accelerate this process.
- **Step 5: Run Pipeline** (run: dvc remote modify ...; dvc repro; dvc push): Executes the DVC pipeline defined in dvc.yaml, reproducing stages like data_ingestion, model_building, and model_evaluation. It configures the DVC remote storage (S3) using CAPSTONE_TEST (for DagsHub/MLflow), BUCKET_NAME, ACCESS_KEY, and SECRET_KEY secrets from the spemlops environment, enabling secure S3 access for data and model artifacts. The dvc repro command runs the pipeline, and dvc push uploads outputs (e.g., model.pkl, vectorizer.pkl) to S3, ensuring versioned artifacts are available.
- **Step 6: Run Model Tests** (run: python -m unittest tests/test_model.py): Runs unit tests for the Logistic Regression model using unittest on tests/test_model.py, validating model functionality (e.g., prediction accuracy). The CAPSTONE_TEST secret, injected from the spemlops environment, authenticates with DagsHub/MLflow to access model artifacts, ensuring tests interact with the correct model version.
- **Step 7: Promote Model to Production** (run: python scripts/promote_model.py): Conditionally executes promote_model.py to promote the validated model to the "Production" stage in MLflow's Model Registry on DagsHub, using CAPSTONE_TEST for authentication. This step, gated by if: success(), ensures only successfully tested models are promoted, maintaining production quality.
- **Step 8: Run Flask App Tests** (run: python -m unittest tests/test_flask_app.py): Conditionally runs unit and integration tests for the Flask API using unittest on tests/test_flask_app.py, validating endpoints (/predict, /metrics). The CAPSTONE_TEST secret authenticates with DagsHub/MLflow to load the model, ensuring tests reflect production behavior. This step, gated by if: success(), confirms API reliability before deployment.
- **Step 9: Configure AWS Credentials** (uses: aws-actions/configure-aws-credentials@v4): Configures AWS CLI credentials using ACCESS_KEY_R, SECRET_KEY_R, and AWS_REGION secrets from the spemlops environment, enabling access to ECR and EKS. The aws-actions/configure-aws-credentials@v4 action sets up the runner's AWS environment, critical for subsequent ECR and deployment steps.

- **Step 10: Login to AWS ECR** (run: aws ecr get-login-password ...): Conditionally logs into AWS ECR using the credentials from Step 9, executing aws ecr get-login-password to authenticate docker login with the ECR registry (${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com). This step, gated by if: success(), ensures secure image pushing, using AWS_REGION and AWS_ACCOUNT_ID secrets.
- **Step 11: Build, Tag, and Push Docker Image**:
  - **Build Docker Image** (run: docker build -t ${{ secrets.ECR_REPOSITORY }}:latest .): Builds the Docker image using the Dockerfile, tagging it with ECR_REPOSITORY (e.g., spemlopsecr). No credentials are needed, as the build is local, but spemlops ensures consistent runner resources.
  - **Tag Docker Image** (run: docker tag ...): Tags the image for ECR (${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}:latest), using AWS_ACCOUNT_ID, AWS_REGION, and ECR_REPOSITORY secrets to form the registry URL.
  - **Check Docker Authentication** (run: docker info): Verifies Docker's authentication status, ensuring ECR login succeeded, requiring no additional credentials.
  - **Push Docker Image to ECR** (run: docker push ...): Conditionally pushes the tagged image to ECR, using the authenticated registry from Step 10. This step, gated by if: success(), ensures only validated images are uploaded, relying on AWS_ACCOUNT_ID, AWS_REGION, and ECR_REPOSITORY.
- **Step 12: Set up Ansible and Run Playbook**:
  - **Set up Ansible** (run: python -m pip install ansible kubernetes; ansible-galaxy ...): Installs Ansible and the kubernetes.core collection from ansible/requirements.yml, preparing the runner for Kubernetes deployment. No credentials are needed, but spemlops ensures a consistent environment.
  - **Run Ansible Playbook for EKS Deployment** (run: ansible-playbook ansible/playbook.yml): Executes playbook.yml to deploy the deployment.yaml resources to EKS, using the eks_deploy role. It injects AWS_REGION, ACCESS_KEY_R, SECRET_KEY_R, CAPSTONE_TEST, ECR_REPOSITORY, and AWS_ACCOUNT_ID secrets to authenticate with EKS and access the ECR image, ensuring seamless deployment of the Flask app.

12.2 Automated Testing

Testing is automated through the Run model tests and Run Flask app tests steps, both leveraging unittest to ensure code quality. The Run model tests step (python -m unittest tests/test_model.py) validates the Logistic Regression model's prediction logic, model loading from MLflow, and compatibility with vectorizer.pkl, using CAPSTONE_TEST to authenticate with DagsHub/MLflow in the spemlops environment. The Run Flask app tests step (python -m unittest tests/test_flask_app.py), conditional on prior success, tests the Flask API's /predict endpoint for correct sentiment predictions and /metrics for Prometheus metric output, also using CAPSTONE_TEST for model access. Linting is not currently implemented, but tools like flake8 or pylint could be added to requirements.txt and run via a new step (e.g., flake8 src tests) to enforce code style, enhancing maintainability.

12.3 Build and Deployment Steps

The build and deployment process spans multiple steps, ensuring the Flask app is containerized and deployed to EKS. The Build Docker image step (docker build -t ${{ secrets.ECR_REPOSITORY }}:latest .) uses the Dockerfile to create an image tagged with ECR_REPOSITORY (e.g., spemlopsecr), encapsulating app.py, vectorizer.pkl, and dependencies. The Tag Docker image step formats the tag for ECR (${{ secrets.AWS_ACCOUNT_ID }}.dkr.ecr.${{ secrets.AWS_REGION }}.amazonaws.com/${{ secrets.ECR_REPOSITORY }}:latest), using AWS_ACCOUNT_ID, AWS_REGION, and ECR_REPOSITORY secrets. The Push Docker image to ECR step uploads the image, authenticated via Step 10's ECR login, ensuring the image is available for EKS. The Run Ansible Playbook for EKS Deployment step applies deployment.yaml via playbook.yml, deploying two replicas of the Flask app and a LoadBalancer service to EKS, with CAPSTONE_TEST, AWS_REGION, ACCESS_KEY_R, SECRET_KEY_R, ECR_REPOSITORY, and AWS_ACCOUNT_ID secrets enabling secure EKS access and image pulling.

12.4 Secrets and Credentials Management

GitHub Secrets, scoped to the spemlops environment, securely manage credentials, injected as environment variables to avoid hardcoding. The secrets include:

- CAPSTONE_TEST: Used in Steps 5, 6, 7, 8, and 12 for DagsHub/MLflow authentication, enabling access to the MLflow Model Registry (models:/my_model/{version}) and DVC remote storage.
- BUCKET_NAME, ACCESS_KEY, SECRET_KEY: Used in Step 5 to configure DVC's S3 remote (dvc remote modify myremote), allowing dvc push to upload artifacts (e.g., model.pkl) to the specified S3 bucket.
- ACCESS_KEY_R, SECRET_KEY_R, AWS_REGION, AWS_ACCOUNT_ID, ECR_REPOSITORY: Used in Steps 9, 10, 11, and 12 for AWS authentication, enabling

ECR login, image pushing, and EKS deployment. AWS_REGION (e.g., us-east-1) and AWS_ACCOUNT_ID (e.g., 065948377643) form ECR URLs, while ECR_REPOSITORY (e.g., spemlopsecr) specifies the repository.

Secrets are injected via the env field in steps, ensuring they are only accessible to authorized steps within the spemlops environment, with GitHub Actions masking sensitive values in logs for security.

13. Future Work & Recommendations

The sentiment analysis MLOps project, with its robust pipeline (ci.yaml), containerized Flask API (app.py, Dockerfile), and Kubernetes deployment (deployment.yaml), provides a strong foundation for scalability and reliability. Future work aims to enhance model performance, operational efficiency, and application scope by integrating advanced machine learning techniques, MLOps practices, and new features. These recommendations leverage the existing infrastructure, including MLflow, DVC, Prometheus, and AWS EKS, to ensure seamless evolution while addressing limitations observed in experimentation (exp1_logisticRegression.ipynb, exp2_VariousModels.py, exp3_hyperparameterTuning.py) and deployment.

13.1 Potential Enhancements

To improve model accuracy, integrating transformer-based models like BERT or DistilBERT, available via Hugging Face's Transformers library, could replace the Logistic Regression model (model_building.py), capturing contextual nuances in text beyond the current Bag of Words approach. Automated data drift detection, using tools like Evidently AI, could be added to the DVC pipeline (dvc.yaml) to monitor shifts in input text distributions, triggering retraining when drift is detected. Implementing blue/green deployment strategies in deployment.yaml would enable zero-downtime updates by maintaining parallel production and staging environments, with Kubernetes routing traffic to the stable version until the new deployment is validated.

13.2 Advanced MLOps Practices

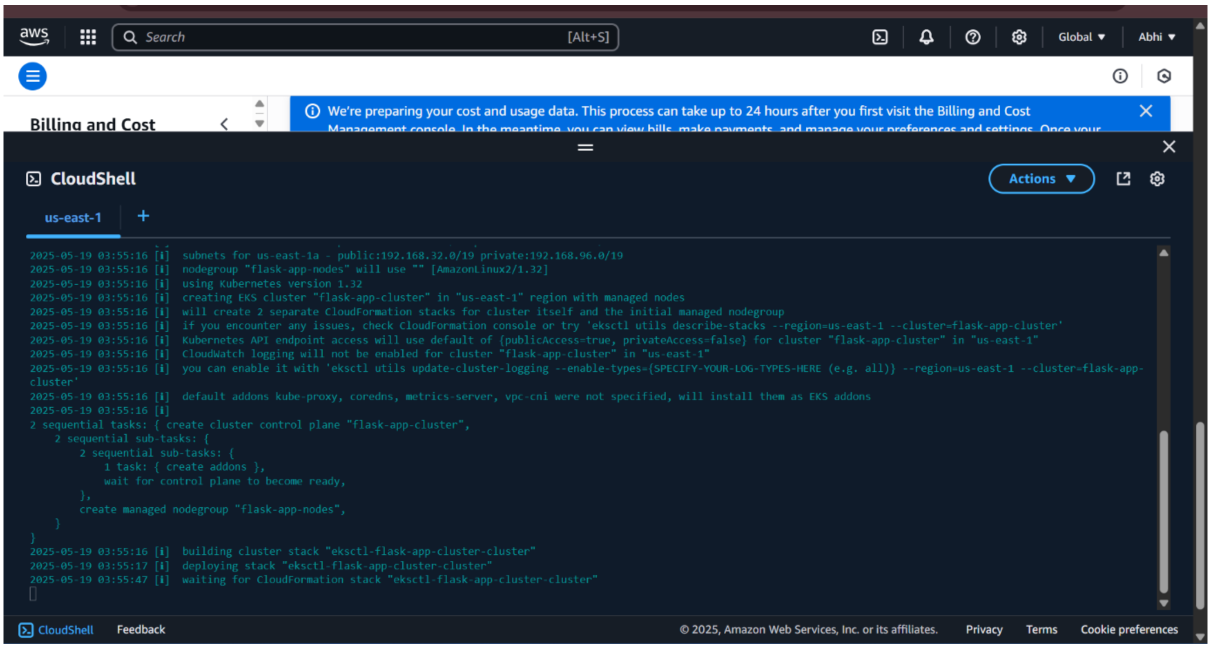
Adopting a feature store, such as Feast, would centralize feature engineering (feature_engineering.py), enabling reusable text features (e.g., TF-IDF, embeddings) across experiments and reducing preprocessing overhead. Enhancing the MLflow Model Registry integration (register_model.py) with automated model validation and versioning policies would streamline model promotion, ensuring only high-performing models reach production. Advanced monitoring, building on Prometheus (app.py's /metrics), could incorporate tools like Grafana for visualized dashboards and AWS CloudWatch for centralized logging, providing deeper insights into API performance and model drift.

13.3 New Features or Applications

Expanding to multi-language sentiment analysis by incorporating multilingual transformers (e.g., XLM-RoBERTa) would broaden the API's applicability, requiring updates to normalize_text in app.py to handle non-English text preprocessing. Enabling real-time prediction streaming via WebSocket endpoints in Flask or a Kafka-based pipeline would support high-throughput applications, such as live social media analysis, extending the current /predict endpoint. Adding a user feedback loop to collect prediction corrections through the web interface (index.html) would facilitate active learning, improving model performance over time by retraining on user-validated data.

14. Sample Screenshots

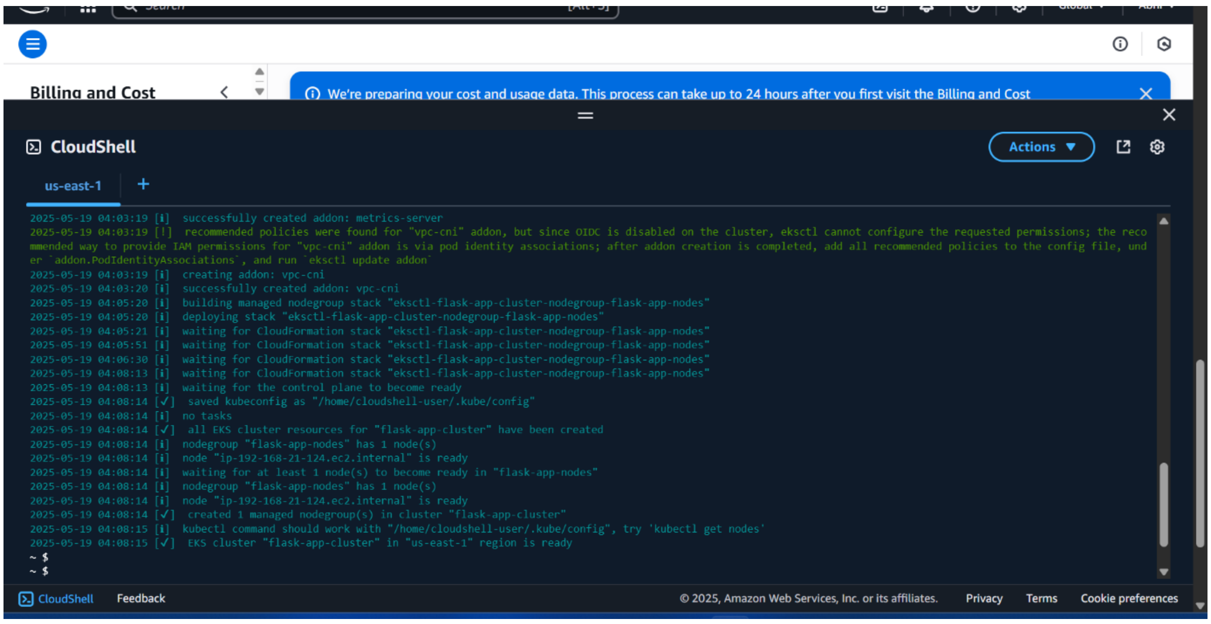
AWS EKS Cluster creation



aws
Search [Alt+S]
Global
Abhi
Billing and Cost
We're preparing your cost and usage data. This process can take up to 24 hours after you first visit the Billing and Cost
CloudShell
Actions
us-east-1

```
2025-05-19 03:55:16 [i]  subnets for us-east-1a - public:192.168.32.0/19 private:192.168.96.0/19
2025-05-19 03:55:16 [i]  nodegroup "flask-app-nodes" will use "" [AmazonLinux2/1.32]
2025-05-19 03:55:16 [i]  using Kubernetes version 1.32
2025-05-19 03:55:16 [i]  creating EKS cluster "flask-app-cluster" in "us-east-1" region with managed nodes
2025-05-19 03:55:16 [i]  will create 2 separate CloudFormation stacks for cluster itself and the initial managed nodegroup
2025-05-19 03:55:16 [i]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --region=us-east-1 --cluster=flask-app-cluster'
2025-05-19 03:55:16 [i]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for cluster "flask-app-cluster" in "us-east-1"
2025-05-19 03:55:16 [i]  CloudWatch logging will not be enabled for cluster "flask-app-cluster" in "us-east-1"
2025-05-19 03:55:16 [i]  you can enable it with 'eksctl utils update-cluster-logging --enable-types={SPECIFY-YOUR-LOG-TYPES-HERE (e.g. all)} --region=us-east-1 --cluster=flask-app-
cluster'
2025-05-19 03:55:16 [i]  default addons kube-proxy, coredns, metrics-server, vpc-cni were not specified, will install them as EKS addons
2025-05-19 03:55:16 [i]
2 sequential tasks: { create cluster control plane "flask-app-cluster",
    2 sequential sub-tasks: {
        2 sequential sub-tasks: {
            1 task: { create addons },
            wait for control plane to become ready,
        },
        create managed nodegroup "flask-app-nodes",
    }
}
2025-05-19 03:55:16 [i]  building cluster stack "eksctl-flask-app-cluster-cluster"
2025-05-19 03:55:17 [i]  deploying stack "eksctl-flask-app-cluster-cluster"
2025-05-19 03:55:47 [i]  waiting for CloudFormation stack "eksctl-flask-app-cluster-cluster"
```

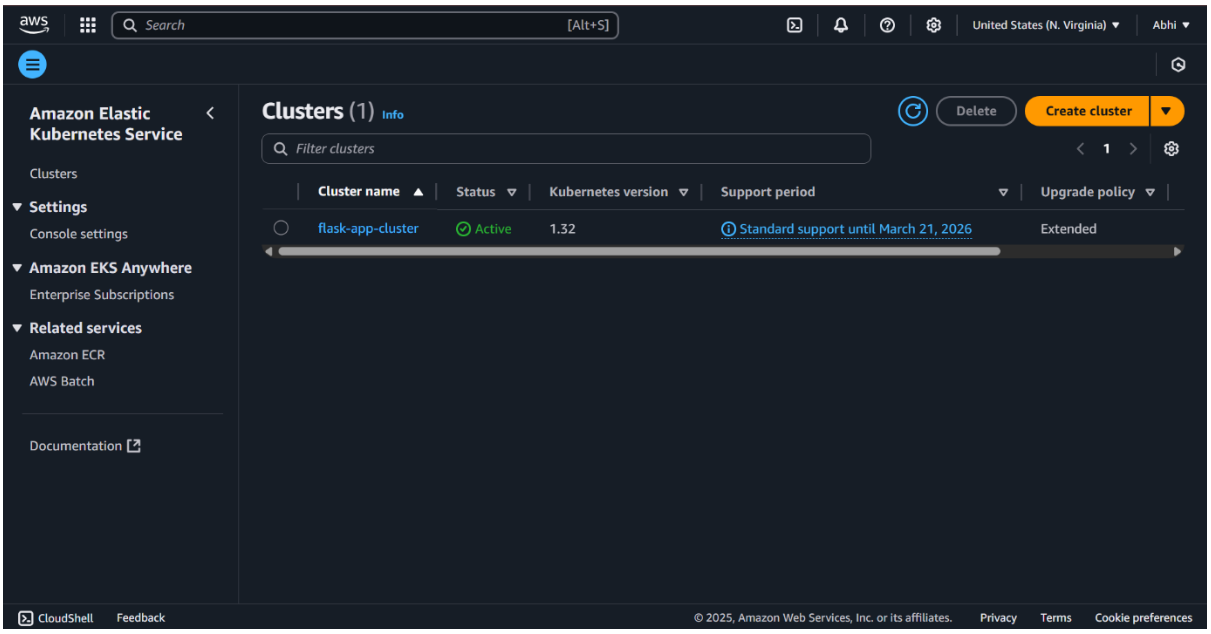
CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences



Billing and Cost
We're preparing your cost and usage data. This process can take up to 24 hours after you first visit the Billing and Cost
CloudShell
Actions
us-east-1

```
2025-05-19 04:03:19 [i]  successfully created addon: metrics-server
2025-05-19 04:03:19 [!]  recommended policies were found for "vpc-cni" addon, but since OIDC is disabled on the cluster, eksctl cannot configure the requested permissions; the reco
mmended way to provide IAM permissions for "vpc-cni" addon is via pod identity associations; after addon creation is completed, add all recommended policies to the config file, und
er `addon.PodIdentityAssociations`, and run `eksctl update addon`
2025-05-19 04:03:19 [i]  creating addon: vpc-cni
2025-05-19 04:03:20 [i]  successfully created addon: vpc-cni
2025-05-19 04:05:20 [i]  building managed nodegroup stack "eksctl-flask-app-cluster-nodegroup-flask-app-nodes"
2025-05-19 04:05:20 [i]  deploying stack "eksctl-flask-app-cluster-nodegroup-flask-app-nodes"
2025-05-19 04:05:21 [i]  waiting for CloudFormation stack "eksctl-flask-app-cluster-nodegroup-flask-app-nodes"
2025-05-19 04:05:51 [i]  waiting for CloudFormation stack "eksctl-flask-app-cluster-nodegroup-flask-app-nodes"
2025-05-19 04:06:30 [i]  waiting for CloudFormation stack "eksctl-flask-app-cluster-nodegroup-flask-app-nodes"
2025-05-19 04:08:13 [i]  waiting for CloudFormation stack "eksctl-flask-app-cluster-nodegroup-flask-app-nodes"
2025-05-19 04:08:13 [i]  waiting for the control plane to become ready
2025-05-19 04:08:14 [✓]  saved kubeconfig as "/home/cloudshell-user/.kube/config"
2025-05-19 04:08:14 [i]  no tasks
2025-05-19 04:08:14 [✓]  all EKS cluster resources for "flask-app-cluster" have been created
2025-05-19 04:08:14 [i]  nodegroup "flask-app-nodes" has 1 node(s)
2025-05-19 04:08:14 [i]  node "ip-192-168-21-124.ec2.internal" is ready
2025-05-19 04:08:14 [i]  waiting for at least 1 node(s) to become ready in "flask-app-nodes"
2025-05-19 04:08:14 [i]  nodegroup "flask-app-nodes" has 1 node(s)
2025-05-19 04:08:14 [i]  node "ip-192-168-21-124.ec2.internal" is ready
2025-05-19 04:08:14 [✓]  created 1 managed nodegroup(s) in cluster "flask-app-cluster"
2025-05-19 04:08:15 [i]  kubectl command should work with "/home/cloudshell-user/.kube/config", try 'kubectl get nodes'
2025-05-19 04:08:15 [✓]  EKS cluster "flask-app-cluster" in "us-east-1" region is ready
~ $
~ $
```

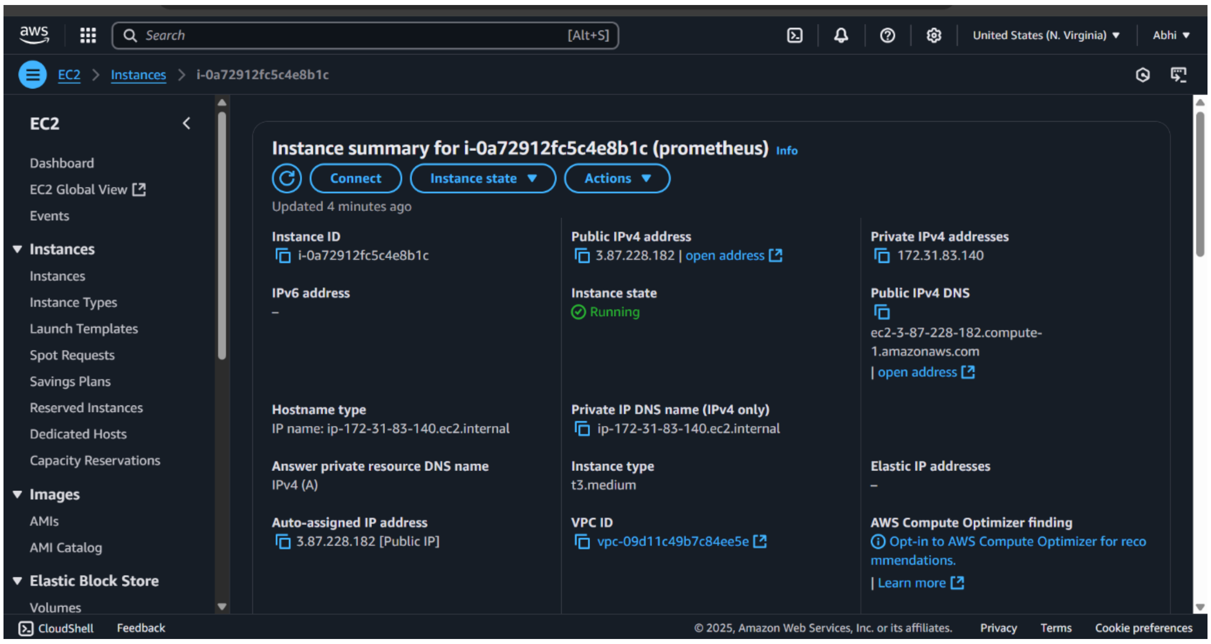
CloudShell Feedback © 2025, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences



Clusters (1) Info

Cluster name	Status	Kubernetes version	Support period	Upgrade policy
flask-app-cluster	Active	1.32	Standard support until March 21, 2026	Extended

Prometheus Instance Creation

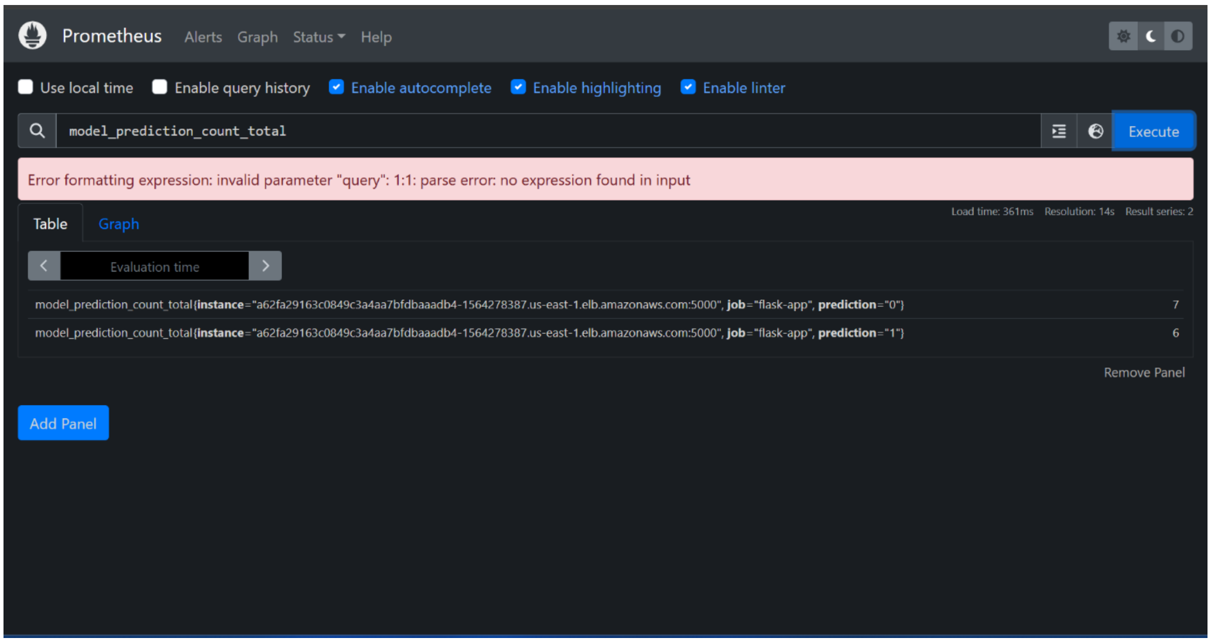


Instance summary for i-0a72912fc5c4e8b1c (prometheus) Info

Updated 4 minutes ago

Instance ID i-0a72912fc5c4e8b1c	**Public IPv4 address** 3.87.228.182 | open address	**Private IPv4 addresses** 172.31.83.140
IPv6 address –	**Instance state** Running	**Public IPv4 DNS** ec2-3-87-228-182.compute-1.amazonaws.com | open address
Hostname type IP name: ip-172-31-83-140.ec2.internal	**Private IP DNS name (IPv4 only)** ip-172-31-83-140.ec2.internal	
Answer private resource DNS name IPv4 (A)	**Instance type** t3.medium	**Elastic IP addresses** –
Auto-assigned IP address 3.87.228.182 [Public IP]	**VPC ID** vpc-09d11c49b7c84ee5e	**AWS Compute Optimizer finding** Opt-in to AWS Compute Optimizer for recommendations. | Learn more

Prometheus Dashboard

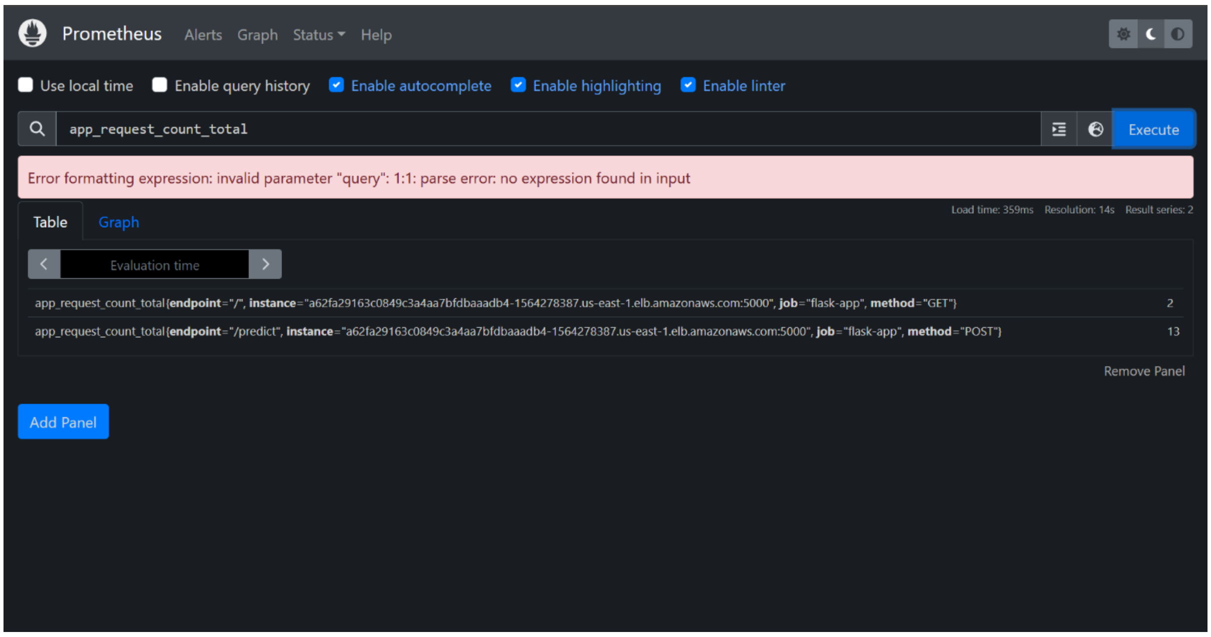


```
model_prediction_count_total
```

Error formatting expression: invalid parameter "query": 1:1: parse error: no expression found in input

Load time: 361ms   Resolution: 14s   Result series: 2

```
model_prediction_count_total{instance="a62fa29163c0849c3a4aa7bfdbaaadb4-1564278387.us-east-1.elb.amazonaws.com:5000", job="flask-app", prediction="0"}    7
model_prediction_count_total{instance="a62fa29163c0849c3a4aa7bfdbaaadb4-1564278387.us-east-1.elb.amazonaws.com:5000", job="flask-app", prediction="1"}    6
```



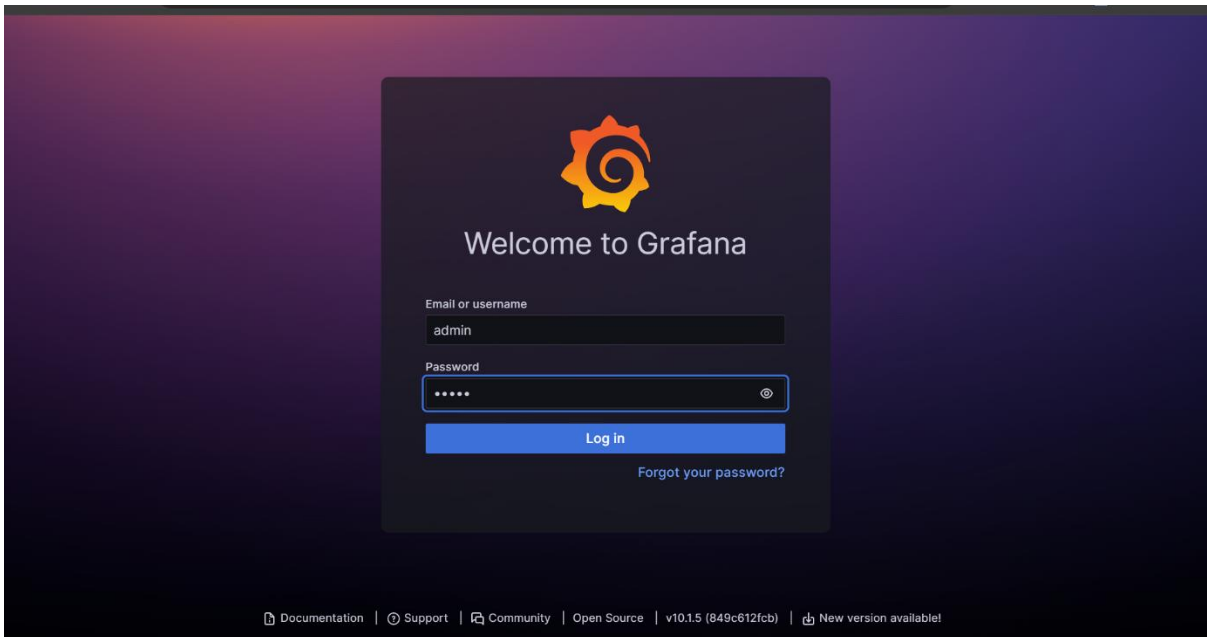
```
app_request_count_total
```

Error formatting expression: invalid parameter "query": 1:1: parse error: no expression found in input

Load time: 359ms   Resolution: 14s   Result series: 2

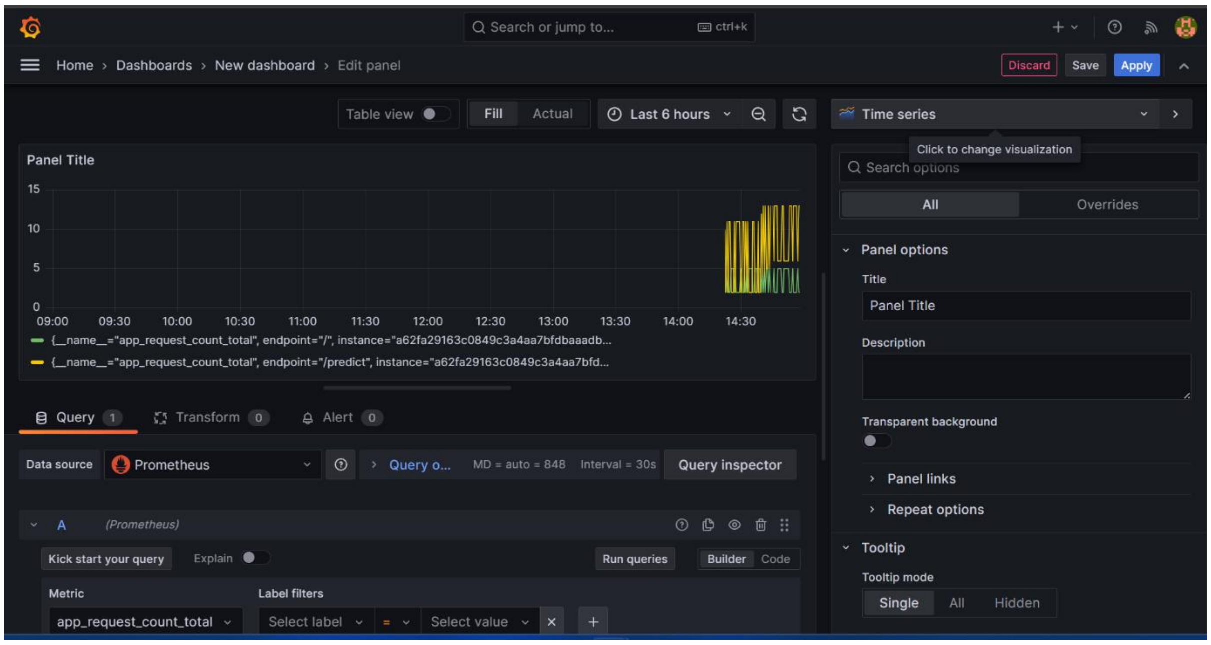
```
app_request_count_total{endpoint="/", instance="a62fa29163c0849c3a4aa7bfdbaaadb4-1564278387.us-east-1.elb.amazonaws.com:5000", job="flask-app", method="GET"}    2
app_request_count_total{endpoint="/predict", instance="a62fa29163c0849c3a4aa7bfdbaaadb4-1564278387.us-east-1.elb.amazonaws.com:5000", job="flask-app", method="POST"}    13
```

Grafana Dashboad


Welcome to Grafana
Log in
Forgot your password?
Documentation | Support | Community | Open Source | v10.1.5 (849c612fcb) | New version available!

Email or username: admin

Password: •••••



{__name__="app_request_count_total", endpoint="/", instance="a62fa29163c0849c3a4aa7bfdbaaadb...

{__name__="app_request_count_total", endpoint="/predict", instance="a62fa29163c0849c3a4aa7bfd...

Data source: Prometheus

Query o... MD = auto = 848 Interval = 30s

A (Prometheus)

Metric: app_request_count_total

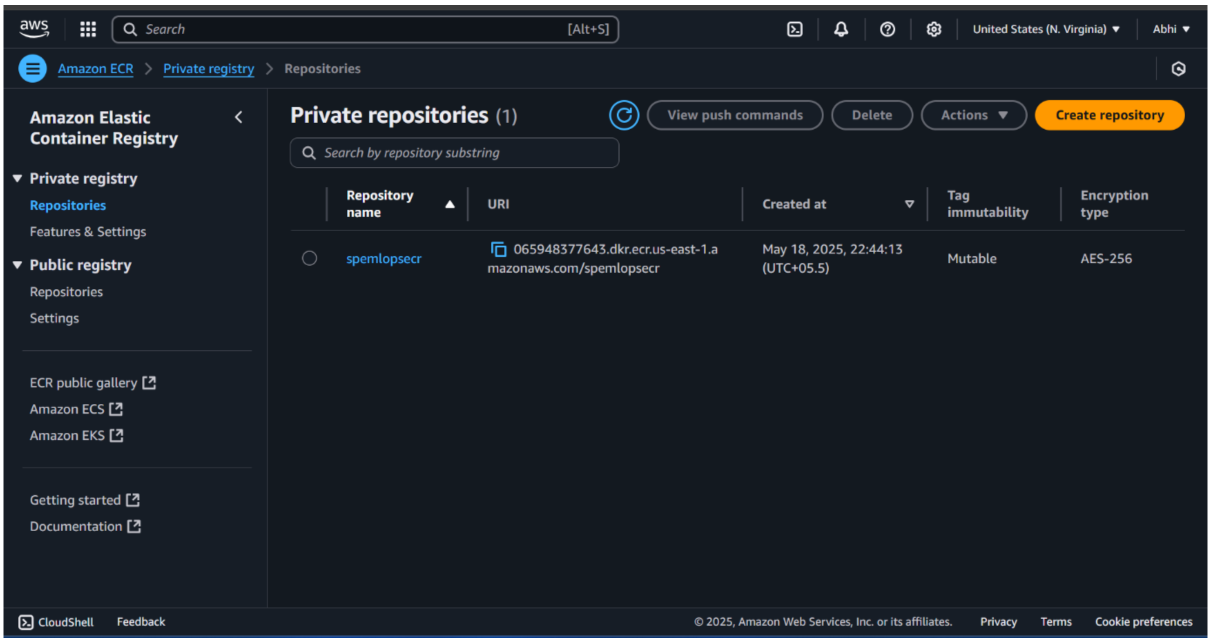
Label filters: Select label = Select value

Title: Panel Title

Description:

**you have to load matrices for visualization.

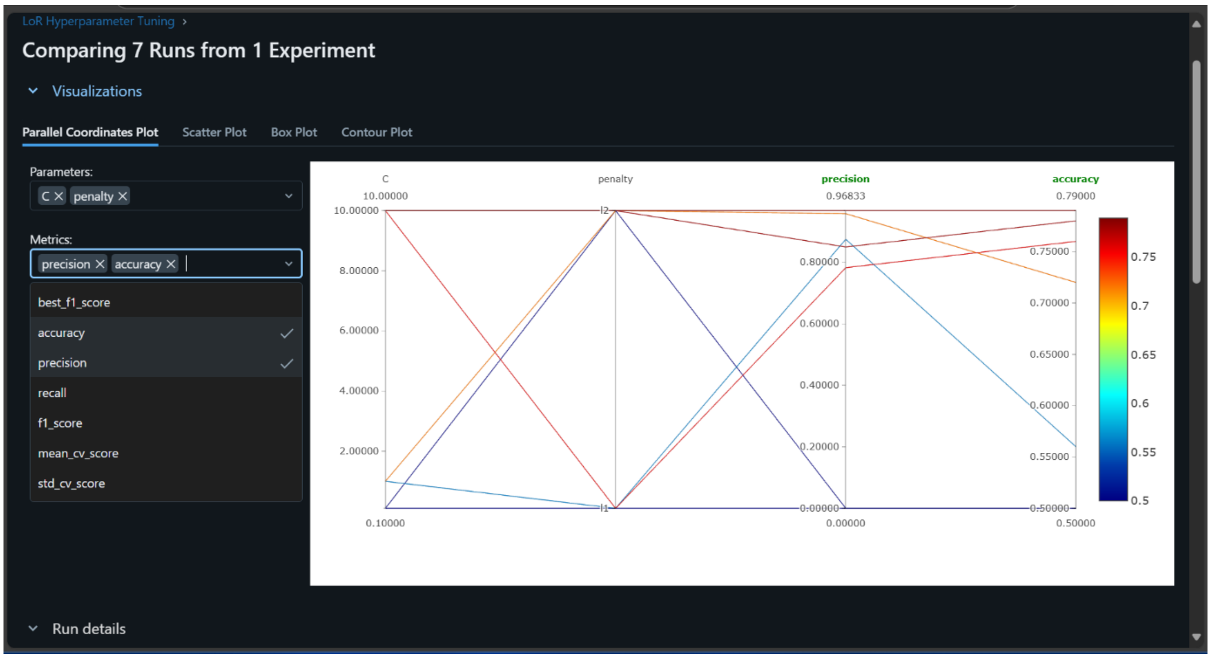
AWS ECR Container Image

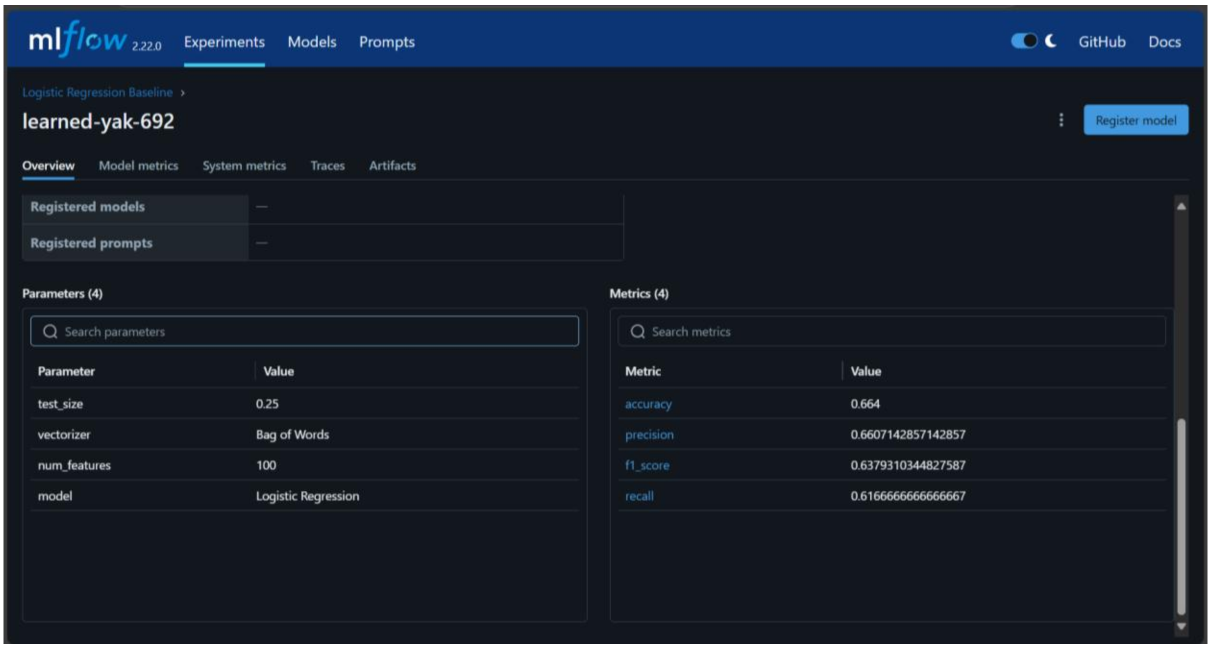


Private repositories (1)

Repository name	URI	Created at	Tag immutability	Encryption type
spemlopsecr	065948377643.dkr.ecr.us-east-1.amazonaws.com/spemlopsecr	May 18, 2025, 22:44:13 (UTC+05.5)	Mutable	AES-256

MLFlow Experiments and Model Registry





learned-yak-692

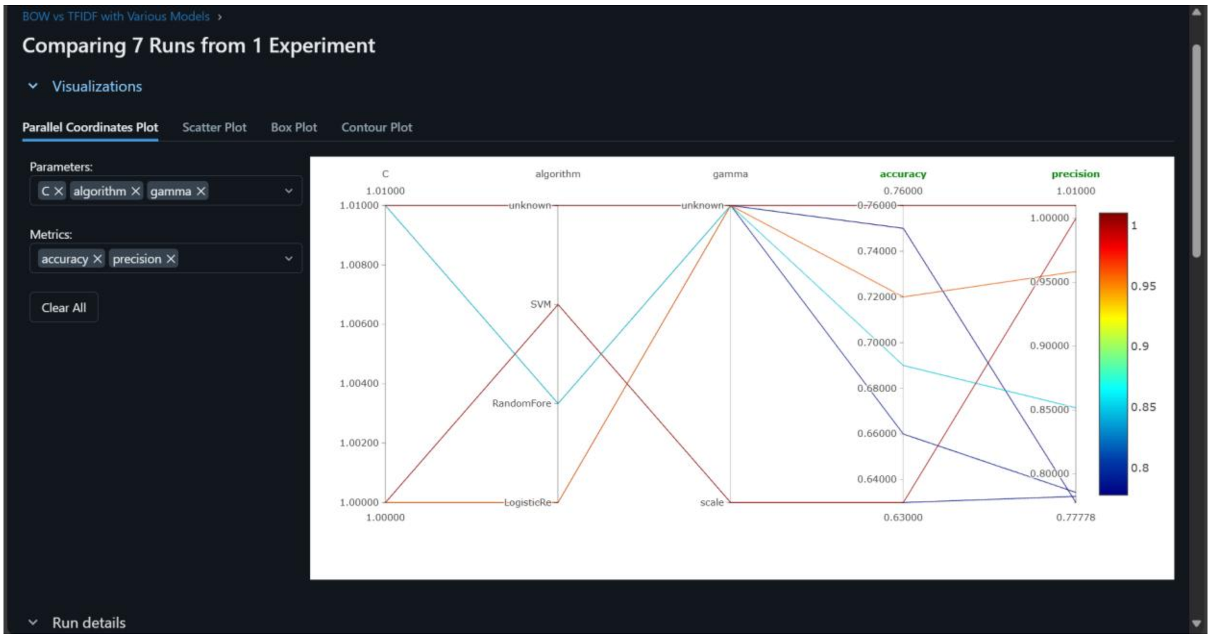
Registered models	—
Registered prompts	—

Parameters (4)

Parameter	Value
test_size	0.25
vectorizer	Bag of Words
num_features	100
model	Logistic Regression

Metrics (4)

Metric	Value
accuracy	0.664
precision	0.6607142857142857
f1_score	0.6379310344827587
recall	0.6166666666666667



BOW vs TFIDF with Various Models ›

Comparing 7 Runs from 1 Experiment

Visualizations

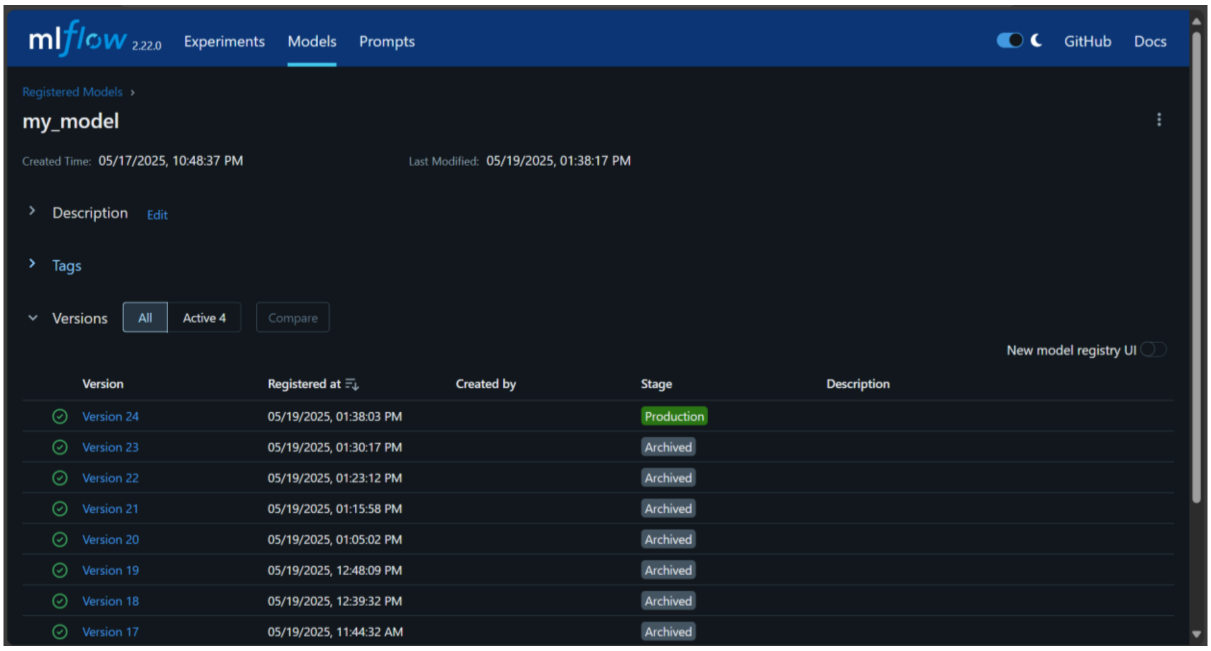
Parallel Coordinates Plot | Scatter Plot | Box Plot | Contour Plot

Parameters: C, algorithm, gamma

Metrics: accuracy, precision

Clear All

Run details



mlflow 2.22.0 | Experiments | Models | Prompts | GitHub | Docs

Registered Models ›

my_model

Created Time: 05/17/2025, 10:48:37 PM   Last Modified: 05/19/2025, 01:38:17 PM

Description Edit

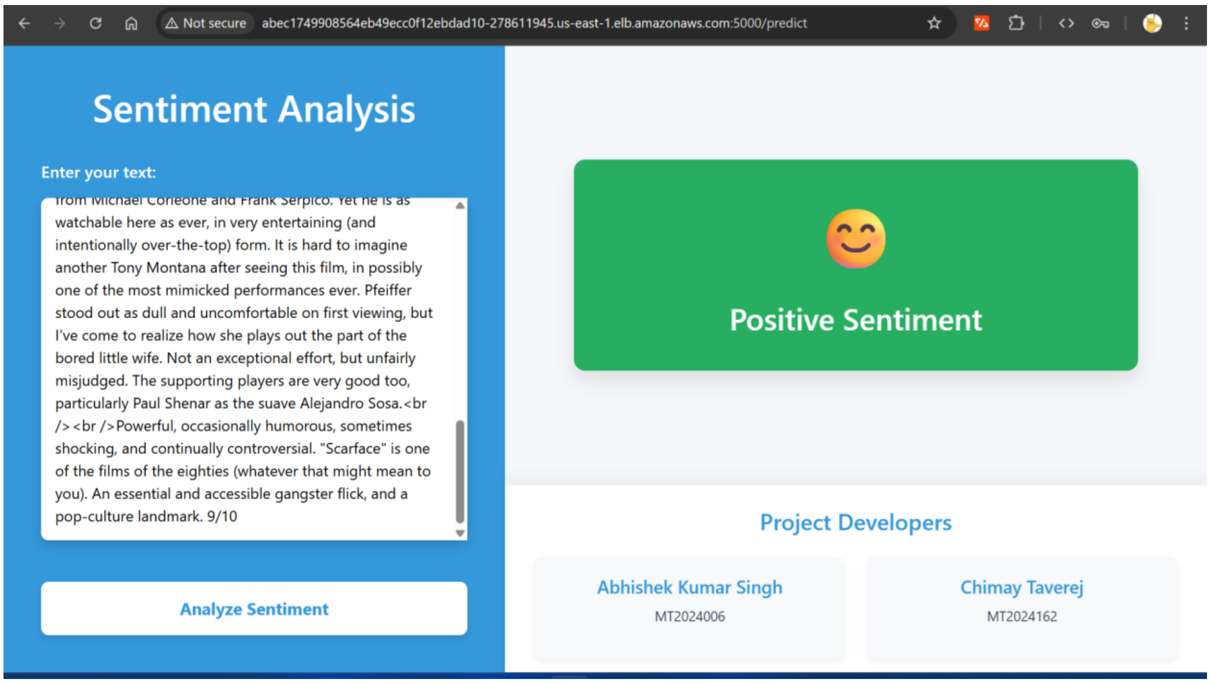
Tags

Versions | All | Active 4 | Compare

New model registry UI

Version	Registered at	Created by	Stage	Description
Version 24	05/19/2025, 01:38:03 PM		Production	
Version 23	05/19/2025, 01:30:17 PM		Archived	
Version 22	05/19/2025, 01:23:12 PM		Archived	
Version 21	05/19/2025, 01:15:58 PM		Archived	
Version 20	05/19/2025, 01:05:02 PM		Archived	
Version 19	05/19/2025, 12:48:09 PM		Archived	
Version 18	05/19/2025, 12:39:32 PM		Archived	
Version 17	05/19/2025, 11:44:32 AM		Archived	

Sample Predictions


abec1749908564eb49ecc0f12ebdad10-278611945.us-east-1.elb.amazonaws.com:5000/predict

Sentiment Analysis

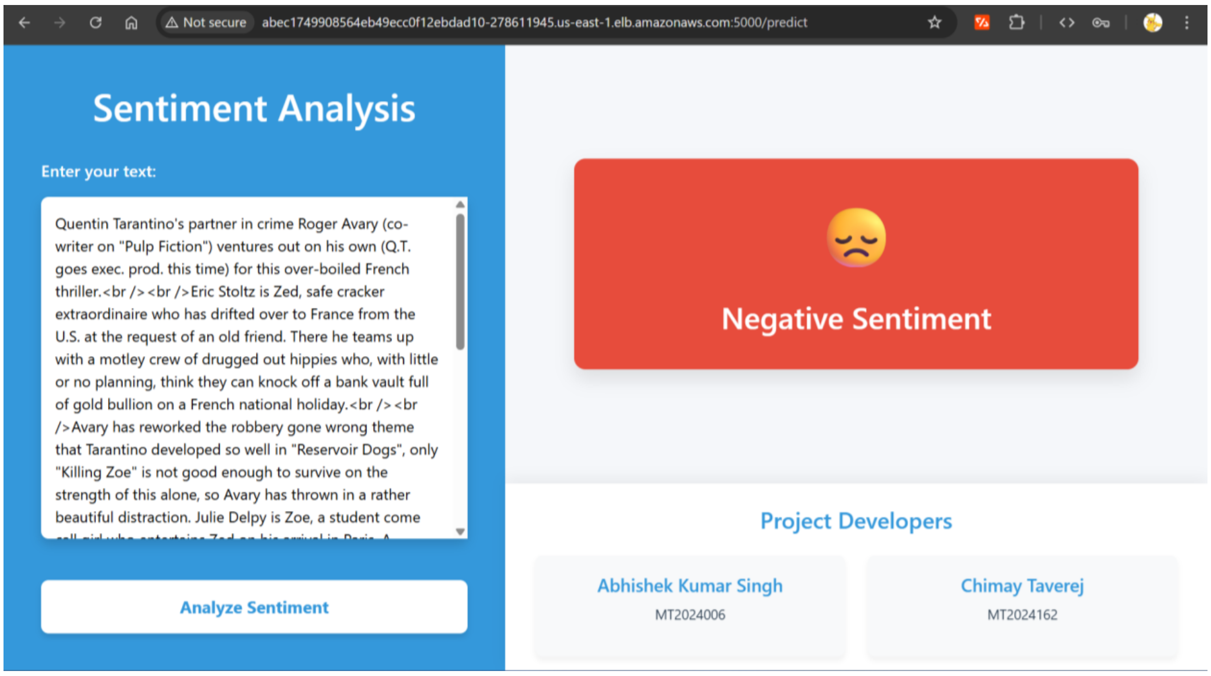
Enter your text:

from Michael Corleone and Frank Serpico. Yet he is as watchable here as ever, in very entertaining (and intentionally over-the-top) form. It is hard to imagine another Tony Montana after seeing this film, in possibly one of the most mimicked performances ever. Pfeiffer stood out as dull and uncomfortable on first viewing, but I've come to realize how she plays out the part of the bored little wife. Not an exceptional effort, but unfairly misjudged. The supporting players are very good too, particularly Paul Shenar as the suave Alejandro Sosa.<br /><br />Powerful, occasionally humorous, sometimes shocking, and continually controversial. "Scarface" is one of the films of the eighties (whatever that might mean to you). An essential and accessible gangster flick, and a pop-culture landmark. 9/10

Positive Sentiment

Project Developers

Abhishek Kumar Singh	Chimay Taverej
MT2024006	MT2024162


abec1749908564eb49ecc0f12ebdad10-278611945.us-east-1.elb.amazonaws.com:5000/predict

Sentiment Analysis

Enter your text:

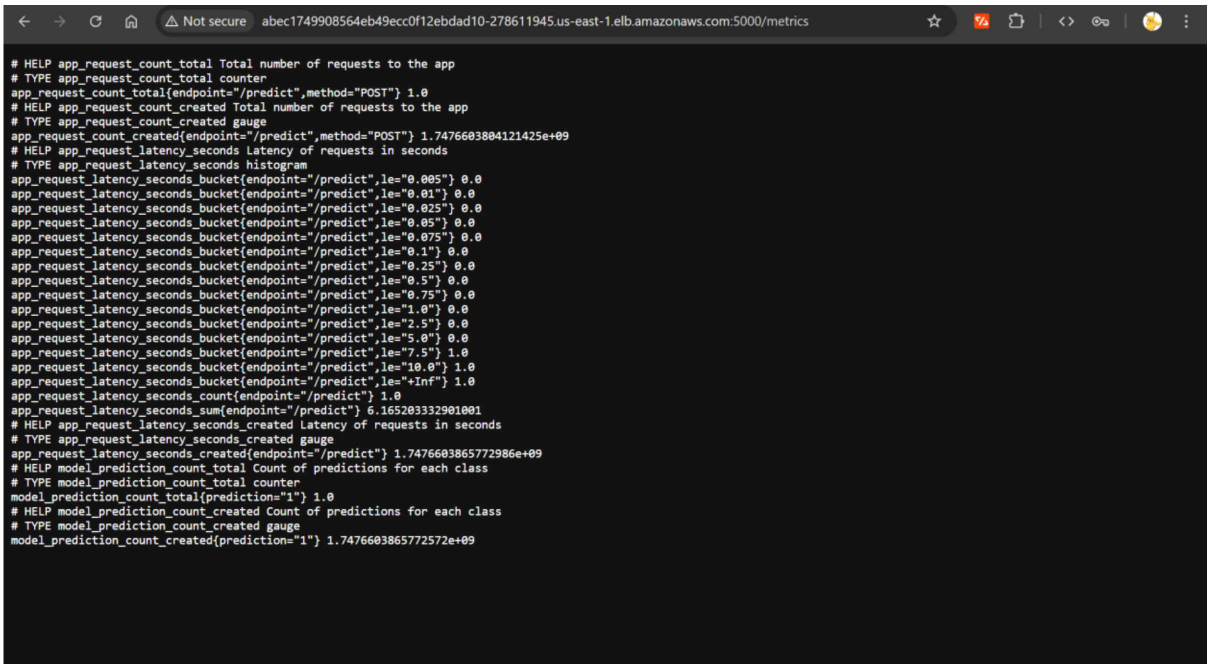
Quentin Tarantino's partner in crime Roger Avary (co-writer on "Pulp Fiction") ventures out on his own (Q.T. goes exec. prod. this time) for this over-boiled French thriller.<br /><br />Eric Stoltz is Zed, safe cracker extraordinaire who has drifted over to France from the U.S. at the request of an old friend. There he teams up with a motley crew of drugged out hippies who, with little or no planning, think they can knock off a bank vault full of gold bullion on a French national holiday.<br /><br />Avary has reworked the robbery gone wrong theme that Tarantino developed so well in "Reservoir Dogs", only "Killing Zoe" is not good enough to survive on the strength of this alone, so Avary has thrown in a rather beautiful distraction. Julie Delpy is Zoe, a student come call girl who entertains Zed on his arrival in Paris. A

Negative Sentiment

Project Developers

Abhishek Kumar Singh	Chimay Taverej
MT2024006	MT2024162

Data which the app is exposing for prometehus (/matrics)


Not secure
abec1749908564eb49ecc0f12ebdad10-278611945.us-east-1.elb.amazonaws.com:5000/metrics

```
# HELP app_request_count_total Total number of requests to the app
# TYPE app_request_count_total counter
app_request_count_total{endpoint="/predict",method="POST"} 1.0
# HELP app_request_count_created Total number of requests to the app
# TYPE app_request_count_created gauge
app_request_count_created{endpoint="/predict",method="POST"} 1.7476603804121425e+09
# HELP app_request_latency_seconds Latency of requests in seconds
# TYPE app_request_latency_seconds histogram
app_request_latency_seconds_bucket{endpoint="/predict",le="0.005"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.01"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.025"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.05"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.075"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.1"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.25"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.5"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="0.75"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="1.0"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="2.5"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="5.0"} 0.0
app_request_latency_seconds_bucket{endpoint="/predict",le="7.5"} 1.0
app_request_latency_seconds_bucket{endpoint="/predict",le="10.0"} 1.0
app_request_latency_seconds_bucket{endpoint="/predict",le="+Inf"} 1.0
app_request_latency_seconds_count{endpoint="/predict"} 1.0
app_request_latency_seconds_sum{endpoint="/predict"} 6.165203332901001
# HELP app_request_latency_seconds_created Latency of requests in seconds
# TYPE app_request_latency_seconds_created gauge
app_request_latency_seconds_created{endpoint="/predict"} 1.7476603865772986e+09
# HELP model_prediction_count_total Count of predictions for each class
# TYPE model_prediction_count_total counter
model_prediction_count_total{prediction="1"} 1.0
# HELP model_prediction_count_created Count of predictions for each class
# TYPE model_prediction_count_created gauge
model_prediction_count_created{prediction="1"} 1.7476603865772572e+09
```

Github link:

https://github.com/aks-master/SPE_MLOps.git