CSE 816: Software Production Engineering Project

Rotten Potatoes: A Kubernetes-Based MLOps Powered Application for Movie Review Analysis

Aayush Bhargav (IMT2022089) & Praveen Peter Jay (IMT2022064)

---

1 Introduction: Project Goals and Architecture

The **Rotten Potatoes** project is a full-stack platform that enables users to browse movie titles, read plot summaries, explore recent reviews, submit their own reviews, and view automatically computed sentiment-based movie scores. At its core, the system integrates an end-to-end **MLOps pipeline** designed to deliver a fully automated, scalable, and reproducible workflow that seamlessly handles data versioning, model training, application containerization, continuous deployment, and real-time monitoring.

The application is composed of three primary components:

1. **Backend (FastAPI):** Exposes the sentiment prediction API and manages all persistent movie, review, and user-generated data through a PostgreSQL database.
2. **Frontend (Streamlit):** Provides an interface for users to explore aforementioned features of the application.
3. **Machine Learning Model:** A production-ready Logistic Regression model responsible for classifying reviews into positive or negative sentiment.

The entire lifecycle is orchestrated using **Jenkins**, with **Ansible** automating configuration management and coordinating a robust **Kubernetes (Minikube)** cluster for containerized application deployment and scaling.

2 Required Manual Pre-requisite Steps

Before enabling the full CI/CD workflow, several one-time setup tasks must be performed. These steps prepare the environment for dataset versioning, Jenkins automation, containerization, Ansible-based deployment, and repository integration. This section serves as a complete guide for running the pipeline on any new system, with the reasonable assumption that both Jenkins and Ansible are pre-installed.

2.1 Fork the Base Repository

Begin by forking the main project repository:

`https://github.com/PraveenPeterJay/sentiment-mlops.git`

This creates a personal copy of the repository that Jenkins can clone, monitor, and build from.

2.2 Project Initialization and DVC Setup

```
mkdir /tmp/dvc_store
python manage_data.py 1
```

This creates the local DVC storage directory and generates the initial `data/train.csv` dataset (derived from the IMDb dataset `full_dataset.csv`).

Figure 1: A word cloud showcasing the full spectrum of DevOps technologies, tools, and components used throughout this project.

2.3 Versioning the Dataset with DVC

```
dvc init
dvc remote add -d mylocal /tmp/dvc_store
dvc add data/train.csv
dvc push
```

This registers the dataset with DVC and pushes its binary contents into the configured DVC store.

2.4 GitHub Repository Setup

```
git init
git remote add origin <your-forked-repo-url>
git push -u origin main
```

Publishing the project enables Jenkins to pull and track the repository.

2.5 Create the Ansible Vault File

Inside the `ansible/` folder, create a vault file to store deployment secrets:

```
cd ansible
nano rotpot_vault.yml
```

Add the following key–value pairs:

```
ansible_become_pass: <your-sudo-password>
db_host: <hostname>
postgres_db: <db-name>
postgres_user: <user>
postgres_password: <password>
```

Encrypt the vault file:

```
ansible-vault encrypt rotpot_vault.yml
```

Store the vault password as a Jenkins Credential (`Kind: Secret Text`) so the pipeline can decrypt the file during deployment.

2.6 Configure Ansible Inventory

Modify the `ansible_user` field in `ansible/inventory.ini` according to your target host machine.

2.7 Jenkins Pipeline Project Configuration

In the Jenkins UI:

- Create a new **Pipeline** job.
- Select **Pipeline script from SCM**.
- Choose **Git**, provide the URL of your forked repository, and select the `main` branch.
- Enable **Git SCM Polling** so Jenkins periodically checks for source updates. Also, provide the path for the Jenkinsfile in the repo.

This ensures Jenkins automatically triggers builds when new commits are pushed.

2.8 Add Credentials in Jenkins

Add Docker Hub username, password, email ID, and the Ansible Vault password to Jenkins following the steps:
`Manage Jenkins → Credentials → Global → Add Credentials`

2.9 Grant Jenkins Permission to Use Docker

```
sudo usermod -aG docker jenkins
sudo systemctl restart jenkins
```

This allows Jenkins to invoke Docker commands for building and pushing images.

2.10 Expose Jenkins for GitHub Webhooks

```
ngrok http --domain=<ngrok-fixed-domain> 8080
```

Use Ngrok's static domain feature to avoid reconfiguring GitHub webhooks after each restart. Keep Ngrok running in a dedicated terminal and register the public endpoint as your GitHub webhook so GitHub can instantly notify Jenkins on new commits (an alternative to SCM polling).

2.11 Fix DVC Store Permissions for Jenkins

```
sudo chmod -R 777 /tmp/dvc_store
```

This ensures Jenkins can access and pull DVC-tracked data during the pipeline run.

3 CI/CD Orchestration and Security Layer

The pipeline is driven by Jenkins and secured by industry-standard secret management tools.

3.1 Jenkins Pipeline (`Jenkinsfile`)

The pipeline is written in **Declarative Groovy**, acting as the single source of truth for the entire workflow.

- **Environment Variables:** Crucial parameters for Docker image naming (`BACKEND_IMAGE`, `FRONTEND_IMAGE`, `DOCKER_TAG`) are defined in the `environment` block, ensuring consistency across all build artifacts.
- **Secure Credential Injection:** Jenkins' built-in `credentials()` function is used to securely inject secrets as environment variables:
  - `DOCKER_CREDS`: Provides the Docker Hub username and password for image pushing.
  - `ANSIBLE_VAULT_PASSWORD`: Injects the password required to decrypt the `rotpot_vault.yml` file.
  - `EMAIL_ID`: Used for post-build notifications.

- **Single Stage Delegation:** Apart from polling, the entire build, train, and deploy process is delegated to a single `sh` step that executes the Ansible playbook, making the Jenkinsfile clean and readable.
- **Post-Build Actions:** The `post` block provides guaranteed execution of cleanup and notification steps:
  - **Notifications:** Uses the `mail` step to send detailed **Success** or **Failure** reports, including the `BUILD_NUMBER`, to the configured email ID.
  - **Cleanup:** `cleanWs()` ensures the workspace is cleared after every run, maintaining host hygiene.

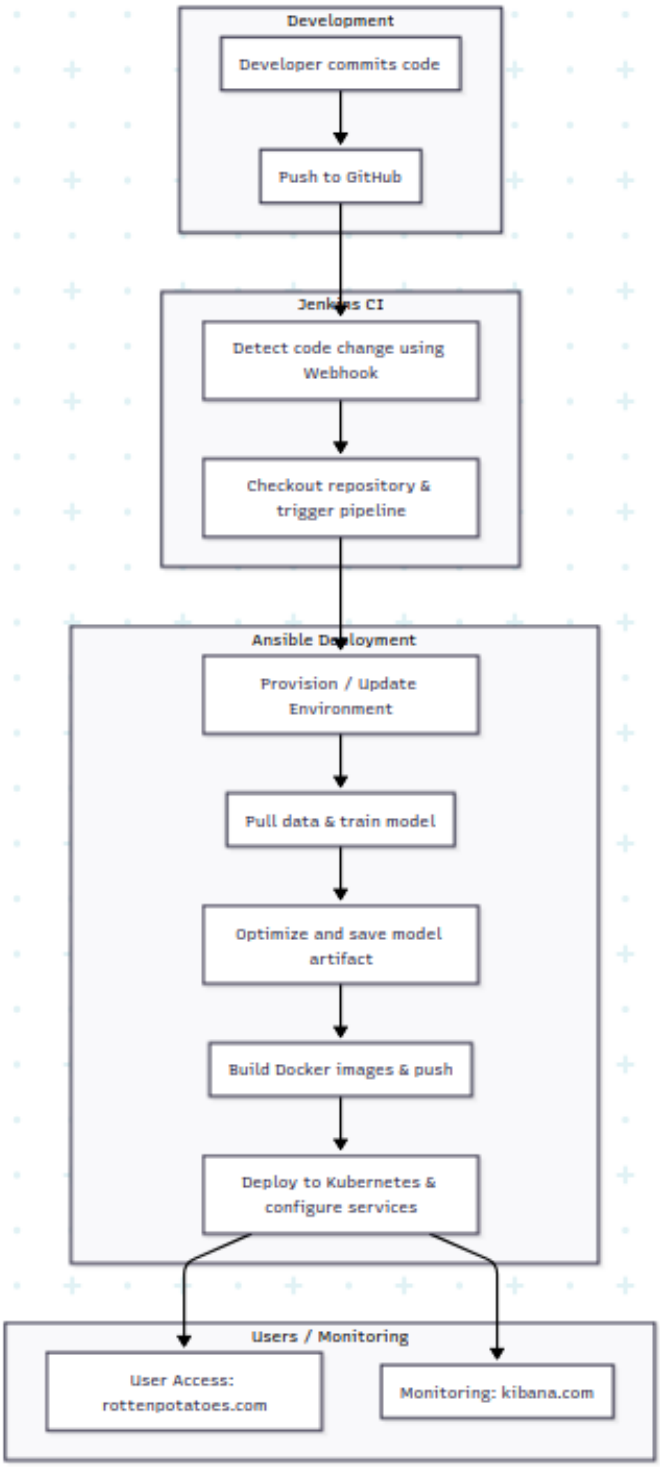

Figure 2: Flowchart illustrating the high-level MLOps pipeline, highlighting Jenkins for continuous integration and Ansible for deployment, ending with user access and monitoring.

```
pipeline {
    agent any

    environment {
        // Existing credential ID
        DOCKER_CREDS = credentials('dockerhub_credentials')

        // Image Names
        BACKEND_IMAGE = "${DOCKER_CREDS_USR}/mlops-backend"
        FRONTEND_IMAGE = "${DOCKER_CREDS_USR}/mlops-frontend"
        DOCKER_TAG = "latest"

        // Ansible vault password
        ANSIBLE_VAULT_PASSWORD = credentials('rotpot_vault_pass')

        // Email for notifications
        EMAIL_ID = credentials('email_id')
    }
```

Figure 3: Environment variables configured in the `Jenkinsfile`, demonstrating that no sensitive values are hard-coded. All secrets are securely referenced through credentials defined during the setup phase.

3.2 Ansible Security and Workflow Management

Ansible drives the deployment workflow with a strong emphasis on idempotency, reproducibility, and secure secret handling. The configuration is structured to ensure that each stage of the pipeline executes deterministically while maintaining strict control over sensitive information.

```
---
- name: "MLOps Pipeline: Configure, Build, Train, & Deploy on Host(s)'"
  hosts: server
  gather_facts: yes

  vars_files:
    - rotpot_vault.yml

  vars:
    # Variables passed from the Jenkinsfile environment
    dockerhub_user: "{{ lookup('env', 'DOCKER_CREDS_USR') }}"
    dockerhub_pass: "{{ lookup('env', 'DOCKER_CREDS_PSW') }}"
    k8s_namespace: "rottenpotatoes"
    kubeconfig_path: "/home/{{ ansible_user }}/.kube/config"
    backend_image: "{{ lookup('env', 'BACKEND_IMAGE') }}"
    frontend_image: "{{ lookup('env', 'FRONTEND_IMAGE') }}"
    docker_tag: "{{ lookup('env', 'DOCKER_TAG') }}"

  roles:
    - update_system
    - install_docker
    - install_kubernetes
    - set_up_workspace
    - train_model
    - use_latest_model
    - build_and_push_to_docker
    - prepare_kubernetes
    - deploy_on_kubernetes
    - configure_kibana
```

Figure 4: The Ansible playbook structure, illustrating a clean separation of concerns through well-defined modular roles, with environment variables dynamically injected from Jenkins during deployment.

- **Ansible Vault:** All sensitive application credentials–the `ansible_user` sudo password and PostgreSQL database secrets–are securely encrypted using Ansible Vault.
  - The main playbook (`playbook.yml`) loads these protected variables through `vars_files: rotpot_vault.yml`.
  - Jenkins provides the decryption key at runtime using `--vault-password-file vault-pass.txt`, enabling seamless–but secure–pipeline execution.

- **Role-Based Execution:** The automation logic is organized into well-defined Ansible roles, each encapsulating a specific operational responsibility. This modular structure improves maintainability, readability, and reusability across the pipeline. In the event that only certain tasks need to be performed, the role can simply be *commented out* from the playbook.
- **Dynamic Variable Lookup:** The playbook employs Ansible's `lookup('env', ...)` mechanism to dynamically fetch Docker credentials, registry details, and image identifiers from environment variables exported by Jenkins. This ensures CI/CD-driven configuration without hardcoding sensitive or environment-specific parameters.

4 System Pre-Configuration and Environment Preparation

Before any CI/CD automation or Kubernetes deployment can occur, the target host must be provisioned with a consistent, reproducible, and fully container-ready environment. This foundational layer is prepared through four Ansible roles—`update_system`, `install_docker`, `install_kubernetes`, and `set_up_workspace`—each responsible for a specific stage of infrastructure bootstrapping.

4.1 Base System Upgrade and Dependency Installation

The preparation process begins with updating the system package index and installing essential OS-level dependencies. These include certificate utilities, transport libraries, rsync for artifact synchronization, and Python's virtual environment tooling. This ensures that all subsequent roles operate on a clean, modern, and dependency-complete foundation, preventing compatibility issues during Docker, Kubernetes, or model-training operations.

4.2 Container Runtime Installation (Docker)

A stable container runtime is a prerequisite for both Minikube and the CI-driven Docker image builds. The `install_docker` role performs the following:

- Installs Docker Engine from the system package repositories.
- Ensures the Docker daemon is enabled and running, guaranteeing that Minikube can later use it as its runtime.
- Adds the Ansible user to the `docker` group, allowing the CI tasks, build steps, and Minikube commands to execute without root privileges.

With Docker configured as the active runtime, all subsequent image builds (backend, frontend, training environment) behave consistently across environments.

4.3 Kubernetes Tooling and Minikube Cluster Bootstrap

The `install_kubernetes` role equips the system with the tooling necessary for cluster operations and initializes the application's local Kubernetes environment.

- **kubectl Installation:** Installed in classic mode to ensure compatibility with all modern APIs used by the deployment manifests (Deployments, Services, PVCs, Ingress, HPA, etc.).
- **Minikube Provisioning:** If not already present, Minikube is downloaded as a standalone binary and configured to run using the Docker runtime. The role automatically:
  - purges any stale Minikube state,
  - starts a fresh cluster with all components (`API Server`, `Kubelet`, `Controller Manager`, `Scheduler`) validated,
  - updates the Kubernetes context so that all subsequent CI tasks operate against the correct cluster.
- **Ingress Addon Activation:** Minikube's built-in ingress controller is enabled, preparing the cluster to receive the application's ingress objects. This allows external traffic routing to both the frontend UI and monitoring stack (Kibana), bridging local hostname entries and in-cluster services.

At the end of this stage, the system hosts a clean, ready-to-use single-node Kubernetes cluster equipped for deploying deployments, PV/PVC-backed services, secrets, autoscalers, and ingress routes defined later in the pipeline.

4.4 Workspace Initialization and Artifact Synchronization

The `set_up_workspace` role standardizes the remote machine's file layout and prepares all necessary artifacts for the upcoming training and deployment stages.

- **Project Workspace Creation:** A dedicated workspace directory is created for all application components, ensuring clean isolation from system paths.
- **MLflow Experiment History Import:** The complete MLflow tracking directory is synchronized from the Jenkins host to the remote machine. This preserves prior runs, versioned artifacts, and lineage—an essential requirement for deterministic model promotion.
- **Source Code and Configuration Transfer:** All project files, configuration templates, and DVC pointer files are copied into the workspace while excluding DVC cache directories. This enables:
  - reproducible dataset retrieval,
  - deterministic model training,
  - and seamless templating of Kubernetes manifests with secrets, PV/PVC references, and image tags.

This pre-configuration layer ensures that every subsequent pipeline stage—training, artifact promotion, container builds, and full Kubernetes deployment—executes in a reproducible, isolated, and infrastructure-stable environment.

5 Application Components: Backend and Frontend

The Rotten Potatoes system is composed of two core components—a FastAPI-based backend and a Streamlit-powered frontend. Both services are fully containerized using Docker images, ensuring reproducibility, isolated dependencies, and seamless deployment inside Kubernetes.

5.1 Backend Service (FastAPI)

The backend exposes all core application functionality through a set of RESTful API endpoints. It is implemented in Python using `FastAPI`, and bundled into a Docker image that includes the MLflow-promoted sentiment analysis model at startup.

Core Responsibilities

- **Model Inference:** Loads the latest MLflow-registered model at container startup and executes sentiment classification for every submitted review. Endpoint: `/submit_review` accepts raw text, returns the predicted sentiment and the model version used.
- **Database Interaction:** The backend persists and retrieves application data using PostgreSQL via SQLAlchemy. Two primary tables are used:
  - `movies` – stores movie metadata including `id`, `name`, and `description`.
  - `movie_reviews` – stores user reviews including `review_id`, `movie_id`, `review` text, and `isPos` boolean indicating sentiment.

  API endpoints include:
  - `/movies` – returns a list of all available movies with their descriptions.
  - `/reviews/{movie_id}` – returns the most recent reviews for the specified movie.
  - `/score/{movie_id}` – computes the freshness score as the ratio of positive reviews to total reviews for the given movie and returns summary statistics:

$$\text{Freshness Score} = \frac{\text{Number of Positive Reviews}}{\text{Total Reviews}} \times 100$$

- **Structured Logging to Elasticsearch:** Each backend event is pushed to the in-cluster Elasticsearch instance at port `9200` as a structured JSON document with fields for event type, model version, movie ID, sentiment, and other metadata.

- **Stateless Containerization:** The backend Docker image contains:
  - FastAPI application code (`app.py`)
  - MLflow model artefacts (`ml_model/`)
  - Python dependencies (transformers, SQLAlchemy, requests, etc.)

  Environment variables configure database and Elasticsearch URLs at runtime.

The backend listens on port `8000`, exposed internally via a Kubernetes `ClusterIP` Service.

5.2 Frontend Service (Streamlit)

The frontend is a user-facing web application developed using `Streamlit`. It provides an interactive interface where users browse movies, view community reviews, and submit their own reviews for real-time sentiment analysis.

Core Responsibilities

- **Movie Browsing and Selection:** The app fetches the movie list from the backend and constructs a dynamic dropdown menu. Selecting a movie triggers:
  - a detailed description display,
  - real-time freshness score retrieval via `/score/{movie_id}`,
  - loading of recent community reviews.
- **Interactive Review Submission:** When a user writes a review, the frontend:
  - enforces a 5-word minimum validation,
  - displays loading indicators using Streamlit's `spinner()`,
  - sends a structured JSON payload to `/submit_review`,
  - renders the predicted sentiment using color-coded visual cues.
- **Dynamic Styling and UX Elements:** Reviews are displayed as color-highlighted quotes (`green` for positive, `red` for negative). Scores are represented with contextual labels: `CERTIFIED HOT`, `FRESH`, and `ROTTEN`.
- **Containerization:** The frontend runs inside a Docker image containing:
  - Streamlit runtime,
  - Python dependencies (`requests`, etc.),
  - The full UI script.

The frontend container listens on port `8501`, which is exposed through a Kubernetes `ClusterIP` Service.

5.3 End-to-End System Interaction

The complete workflow proceeds as follows:

1. User selects a movie via the Streamlit interface.
2. Frontend fetches movie details, score, and recent reviews from the backend.
3. User submits a review; frontend sends JSON data to the backend.
4. Backend:
   - processes sentiment using the MLflow model,
   - stores the review in PostgreSQL,
   - logs structured telemetry to Elasticsearch.
5. Frontend reloads data and updates the score in real time.

Both components remain loosely coupled, scalable, and fully portable due to their Docker-based encapsulation.

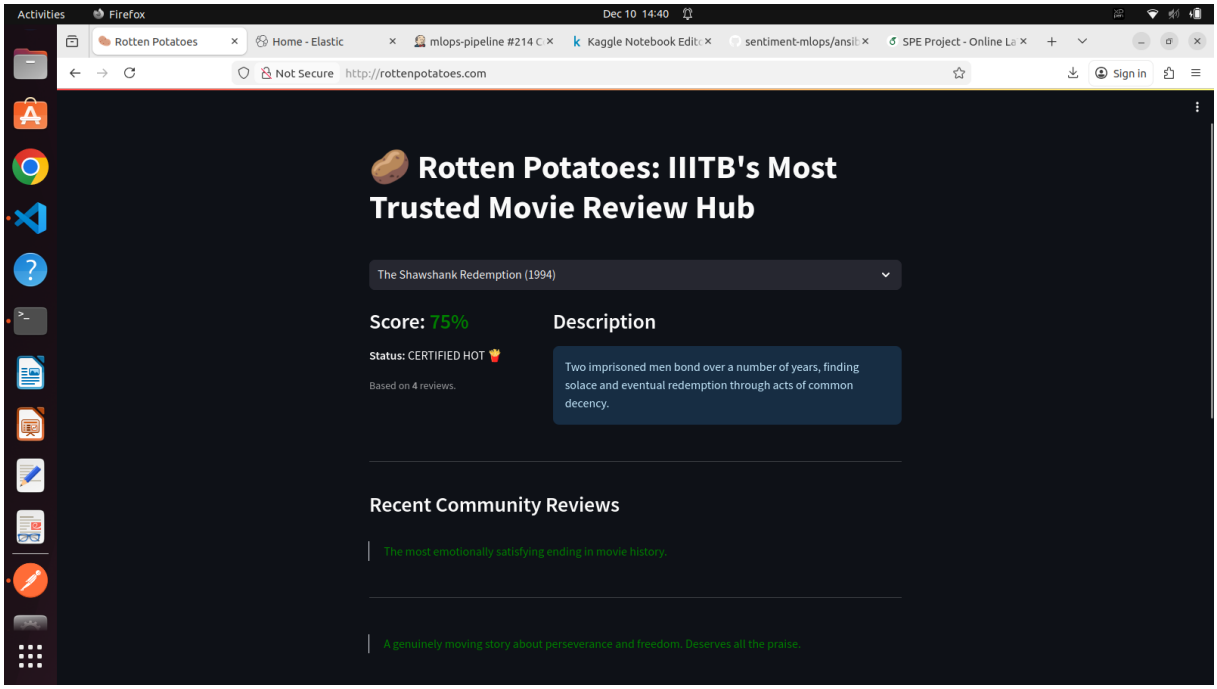

Figure 5: The **Rotten Potatoes** web application being accessed via Ingress.

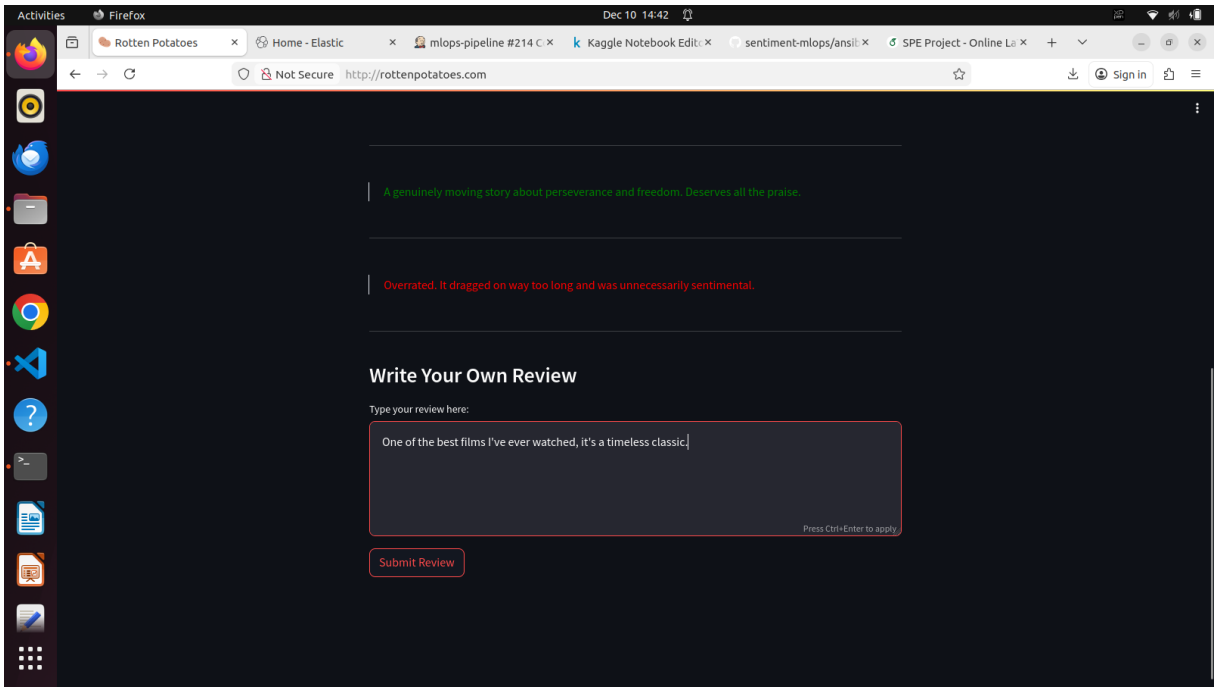

Figure 6: The review submission interface, allowing users to type and submit their movie reviews.

6 MLOps Core: Training, Artifact Promotion, and DVC

6.1 Sentiment Analysis Model

At the heart of the MLOps pipeline is a machine learning model that predicts the sentiment of user-submitted movie reviews. This model classifies reviews as **positive** or **negative**, which are then used to compute the

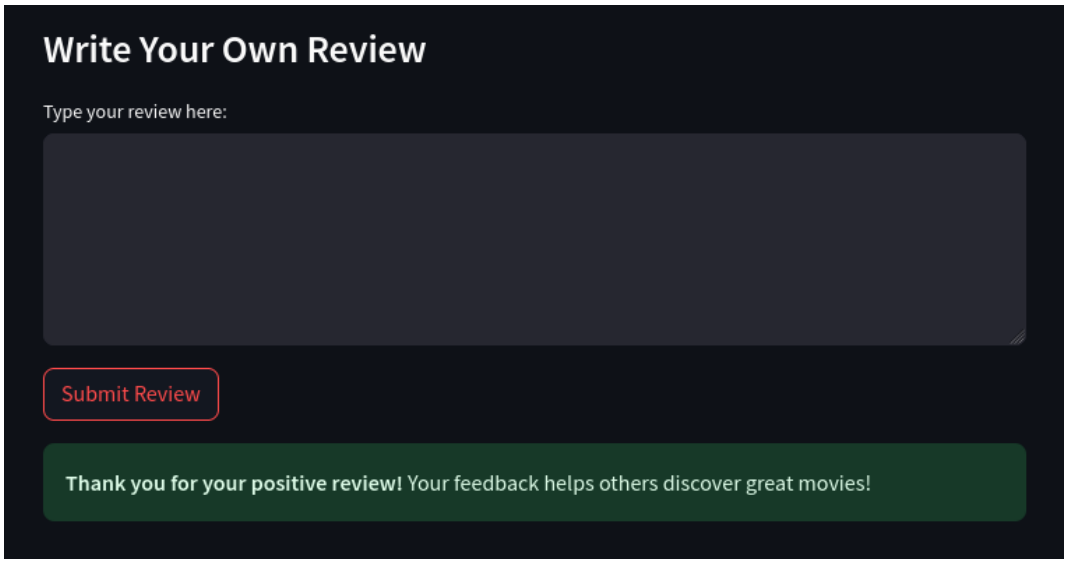

Figure 7: Confirmation screen displayed after a successful review submission and sentiment prediction.

freshness score for each movie (positive reviews divided by total reviews).

The training pipeline uses a lightweight linear classifier based on logistic regression. This design prioritizes fast training cycles, low inference latency, and stable performance—well-suited for the real-time sentiment classification requirements of the application.

6.2 Model Training and MLflow Integration (`train.py`)

The model is trained through the `train.py` script, which implements a reproducible and auditable training workflow built around MLflow:

- **Experiment Tracking via MLflow:** Every training execution triggers a new MLflow run. Configuration parameters (preprocessing, vectorizer settings, classifier type), training metrics, and environment details are automatically logged. This enables:
  - transparent comparison across model versions,
  - full reproduction of any past run,
  - a persistent audit log within the project.
- **Artifact Packaging:** MLflow stores the trained model along with its metadata and conda environment specification. These artifacts are kept in the persistent `mlflow_history` directory, ensuring continuity even across CI runs or container rebuilds.
- **Promotion of Latest Artifact:** After training completes, the newest MLflow model is identified and exported into a dedicated `ml_model/` folder. This folder acts as the handoff point for deployment, guaranteeing that the backend always embeds the latest validated model.

6.3 Data Version Control (DVC)

To ensure that all training runs operate on the exact same dataset version, the project uses **DVC** as the data management layer:

- The `train_model` role performs a `dvc pull` to retrieve `data/train.csv` from a local DVC remote (`/tmp/dvc_store`). This guarantees that the dataset used during CI training is always aligned with what is tracked in Git.
- The DVC metadata file (`train.csv.dvc`) contains the file checksum. This serves as a cryptographic fingerprint that prevents accidental drift in dataset versions across environments. When this is changed and pushed to GitHub, Jenkins is notified that there has been change in the training data, and triggers the pipeline.

- By locking each training run to an immutable dataset snapshot, experiment results remain reproducible and auditable regardless of when or where the training is executed.

6.4 Artifact Promotion and Containerization

Deployment readiness is automated through the Ansible `train_model` and `use_latest_model` roles, which orchestrates the full lifecycle from training to container embedding. The `train_model` role includes the following steps:

1. **Isolated Training Environment:** Each training run is executed inside a fresh virtual environment to avoid dependency contamination and ensure reproducible behaviour.
2. **Dataset Retrieval:** The dataset is fetched from the DVC remote at runtime. This guarantees that training uses the canonical version of `train.csv` associated with the current Git commit.
3. **MLflow History Mounting:** Jenkins provides a synchronized history of all previous MLflow runs. This history is linked as `mlruns/` inside the workspace so the new run seamlessly extends the experiment lineage.

The core function of the `use_latest_model role` is the execution of the following process:

1. **Model Artifact Resolution:** Once training finishes, the pipeline scans the MLflow directory to identify the most recent successful run. This avoids assumptions about run order and ensures promotion always uses the newest, valid artifact.
2. **Artifact Promotion:** The selected MLflow artifact is copied into `ml_model/`, which acts as the stable production candidate. Only a single promoted model exists at any time, simplifying deployment flows and preventing ambiguity.
3. **Container Embedding:** During the backend Docker build, the `ml_model/` directory is baked directly into the application image. This approach provides deterministic deployments: each backend image contains exactly one well-defined MLflow model, fully traceable back to the originating experiment.

Together, these steps implement the core principles of MLOps—versioned data, reproducible training, traceable artifacts, and deterministic deployment—while keeping the pipeline efficient and fully automated.

7 Kubernetes Deployment and Runtime Infrastructure

The application is deployed on Kubernetes using the `prepare_kubernetes` and `deploy_on_kubernetes` roles to ensure scalability, self-healing, and automated production rollouts. The cluster orchestrates the frontend, backend, database, ML model bundle, and the ELK monitoring stack while enforcing strict separation of concerns through namespaces, Secrets, and persistent storage resources.

7.1 Core Deployment Architecture

- **Namespace Isolation:** All Kubernetes objects—Deployments, Services, Secrets, ConfigMaps, PVCs, and Ingress resources—are deployed inside the dedicated namespace `rottenpotatoes`, ensuring strict scoping, cleaner lifecycle management, and isolation from cluster-wide workloads.
- **Backend Deployment:** The backend service is deployed as a stateless Deployment exposing port `8000`. The production container includes the most recently promoted model under `ml_model/`, ensuring deterministic inference across replica restarts. Configuration values such as API keys and database URLs are injected using Kubernetes Secrets.
- **PostgreSQL Deployment:** PostgreSQL operates as a single-replica Deployment exposing port `5432`. A dedicated **PVC (1 Gi)** provides durable storage for table data and WAL files, while credentials (username, password, database name) are securely mounted using `envFrom` and a `postgres-secret`. A ClusterIP Service, `postgres-service`, enables internal access for the backend.
- **Frontend Deployment:** The React application is deployed as a stateless Deployment served on port `8501`. A corresponding ClusterIP Service provides internal load balancing, while a NodePort or Ingress exposes the UI externally depending on the environment configuration.

- **Elasticsearch Deployment:** Elasticsearch (`port 9200`) runs as a single-node Deployment used for indexing application logs and analytics data. A **2 Gi PVC** stores shard data, index metadata, and cluster state, ensuring durability. An `initContainer` corrects storage directory permissions (UID 1000), preventing boot-time failures. A ClusterIP Service named `elasticsearch` facilitates stable internal connectivity for Kibana.
- **Kibana Deployment:** Kibana provides the visualization layer for the ELK stack, running on port `5601`. It consumes the Elasticsearch endpoint via the `ELASTICSEARCH_HOSTS` environment variable. A **NodePort Service** exposes Kibana externally, enabling direct dashboard access for system monitoring and log inspection.

7.2 Resource Requests, Limits, and Horizontal Pod Autoscaling (HPA)

Kubernetes provides fine-grained controls to manage how applications consume cluster resources. For the backend service particularly, we configure explicit **resource requests**, **resource limits**, and a **Horizontal Pod Autoscaler (HPA)** to ensure predictable performance under varying loads.

Resource Requests and Limits

- **Requests** define the minimum guaranteed CPU the scheduler must allocate to a Pod. The backend requests `100m` CPU, ensuring reliable placement without overcommitment.
- **Limits** define the maximum CPU a container may consume. If the backend attempts to exceed `200m`, Kubernetes throttles it to protect node stability.

By setting conservative requests and slightly higher limits, the backend remains lightweight in idle conditions while allowing controlled bursts of workload.

Horizontal Pod Autoscaler (HPA)

The Horizontal Pod Autoscaler dynamically adjusts the number of Pod replicas based on CPU utilization:

- The backend scales between 1 and 5 Pods.
- CPU utilization is monitored continuously, and if the target value of `5%` is exceeded, Kubernetes creates additional Pods automatically.
- When CPU usage drops, unused Pods are terminated to conserve resources.

This provides efficient scaling during traffic spikes while minimizing idle resource costs.

Observing the HPA in Action

To validate autoscaling behavior, we generate load on the backend service using `kubectl port-forward` and ApacheBench.

1. **Expose the backend service locally:**

   ```
   kubectl port-forward svc/backend -n rottenpotatoes 6969:8000
   ```

2. **Run a high-volume load test using ApacheBench:**

   ```
   ab 10000 -c 100 localhost:6969/movies
   ```

3. **Monitor the autoscaler in real time:**

   ```
   watch -n 0.5 kubectl get hpa -n rottenpotatoes
   ```

Step 2 sends 10,000 requests with a concurrency of 100. As CPU usage increases, the HPA detects the spike and scales the backend Deployment accordingly. Step 3 shows live updates of CPU utilization, desired replicas, and scaling activity.

Together, these steps demonstrate how Kubernetes automatically responds to workload spikes and provisions additional compute resources in real time.

7.3 Persistent Storage: PV and PVC Usage

Although most application components in the Rotten Potatoes deployment are stateless, two critical services require durable, node-independent storage: **PostgreSQL** (application database containing tables and metadata) and **Elasticsearch** (containing indexed logs and metadata). Kubernetes Persistent Volumes (PVs) and Persistent Volume Claims (PVCs) ensure that data persists even if pods are restarted, rescheduled, or updated.

Rationale for PV/PVC Usage:

- Stateless components (frontend, backend) do not require PVs and can be freely rescheduled.
- Stateful services (PostgreSQL and Elasticsearch) rely on PVC-backed volumes to maintain mission-critical data.
- Kubernetes dynamically binds each PVC to a suitable PV, ensuring:
  - data integrity,
  - resilience against pod failures,
  - consistent state across restarts,
  - seamless lifecycle management.

7.4 Configuration, Secrets, and Environment Management

- **Secrets:** The Postgres credentials are stored securely in Kubernetes Secrets. These are mounted using `envFrom` in the backend and Postgres Deployments, ensuring no secret appears in Git or in container layers.
- **Secure Backend Bootstrapping:** The backend reads all required connection strings, authentication keys, and environment variables from Secrets at startup, ensuring reproducible and secure behaviour across deployments.

7.5 Ingress and External Access Control

This subsection describes how Kubernetes Ingress objects expose the Rotten Potatoes application, how external traffic is routed, and which ports are used by each service.

- **Ingress Controller (NGINX):** An NGINX Ingress controller is used to perform host- and path-based routing from external HTTP traffic into cluster Services.
- **Host-based routing and Ingress rules:** Two primary Ingress objects map hostnames to internal services:
  - `frontend-ingress` — routes requests for `rottenpotatoes.com` to `frontend-service` on port `8501`.
  - `kibana-ingress` — routes requests for `kibana.com` to `kibana` service on port `5601`.
- **Host DNS mapping for local clusters:** For local Minikube deployments the Ansible `prepare_kubernetes` role maps the Minikube node IP into `/etc/hosts` so the hostnames `rottenpotatoes.com` and `kibana.com` resolve to the cluster node. This allows browser access to Ingress hostnames without external DNS.

7.6 Deployment Workflow and Rolling Updates

- **Declarative Manifests:** All resources—Deployments, Services, PVCs, Secrets, ConfigMaps, and Ingress objects—are maintained as YAML manifests and applied using the `deploy_on_kubernetes` role.
- **Rolling Updates:** Backend images containing the promoted ML model are deployed using Kubernetes' `RollingUpdate` strategy, ensuring:
  - zero downtime,
  - old pods terminate only after new pods pass readiness checks,
  - immediate automatic rollback if deployment health degrades.

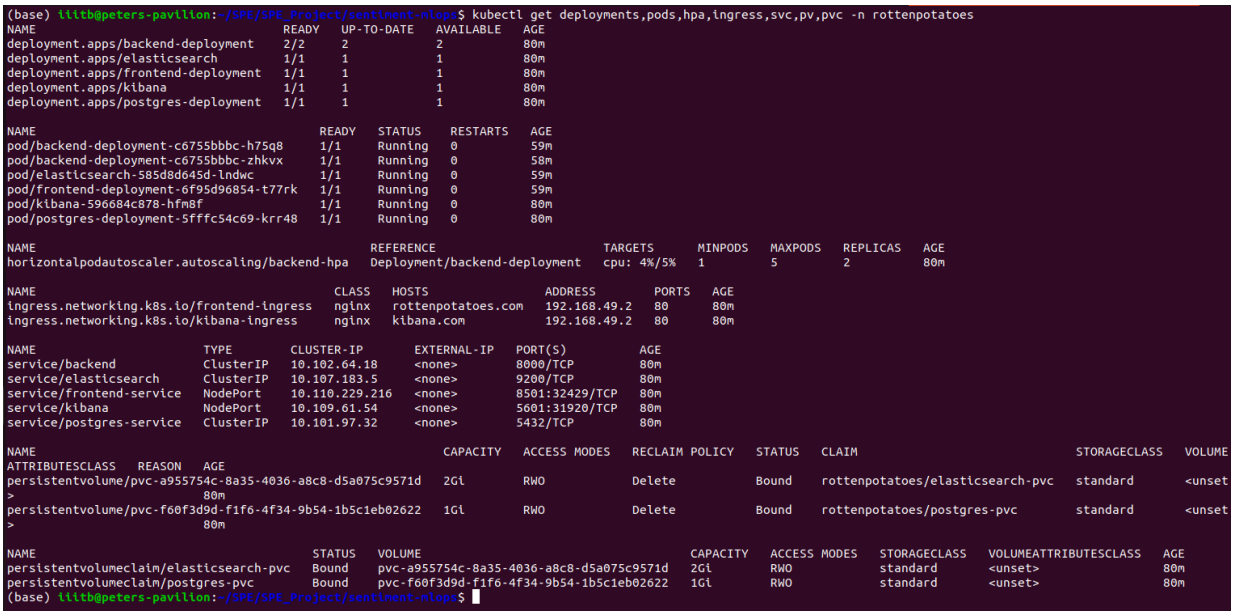

Figure 8: Output of `kubectl get` displaying all deployed Kubernetes artefacts in the `rottenpotatoes` namespace, including Deployments, Pods, the Horizontal Pod Autoscaler (HPA), Services, Ingress, PersistentVolume (PV) and PersistentVolumeClaim (PVC).

8 Advanced Monitoring with ELK Stack

The backend includes a tightly integrated **Elasticsearch** and `Kibana)` monitoring layer. Since the system runs on Kubernetes, all log ingestion and visualization is containerized, fully automated, and environment-agnostic.

8.1 Custom Structured Logging from `app.py`

The backend integrates a purpose-built, production-grade logging pipeline based on a custom Python handler named `SimpleElasticsearchHandler`. Instead of emitting unstructured text logs to stdout, all application events are transformed into rich JSON documents and shipped directly to the in-cluster `Elasticsearch` instance at `http://elasticsearch:9200`. This design provides immediate indexing, real-time searchability, and seamless compatibility with Kibana visualizations, without requiring Logstash or Beats.

1. Direct Log Shipping

Upon initialization, the logging handler dynamically constructs the ingestion endpoint using the configured index name:

`POST http://elasticsearch:9200/rotten_potatoes_logs/_doc`

Every time the application generates a log record, the handler intercepts it, serializes it into a structured JSON payload, and sends it to Elasticsearch using a raw HTTP POST request. This avoids traditional multi-stage log pipelines and ensures:

- **Zero external dependencies**: logs never touch the filesystem.
- **Real-time indexing**: documents become searchable within milliseconds.
- **Stable, deterministic schema**: Kibana dashboards remain consistent across deployments.

2. High-Resolution Log Document Structure

Each log event is transformed into a well-defined JSON object. The following fields are always included:

- **timestamp**: The handler captures a precise, ISO-8601 UTC timestamp at the moment of log emission. Kibana uses this as the primary time filter for dashboards.

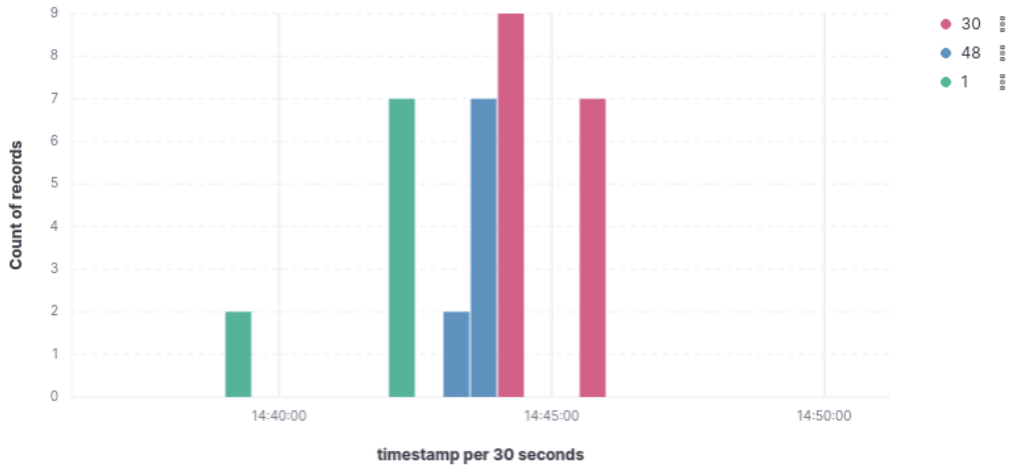

Figure 9: Time-series distribution of logs, with counts broken down per `movie_id`.

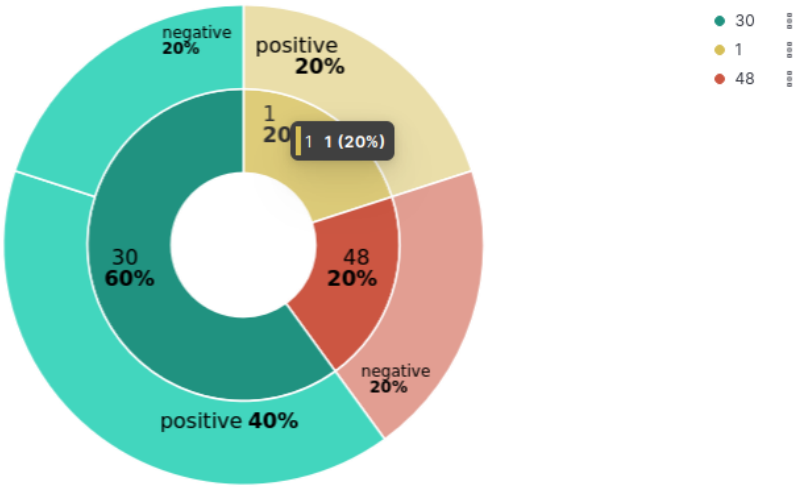

Figure 10: The inner donut shows the distribution of movie review submissions across `movie_id` values. The outer donut displays the distribution of model predictions, breaking down each movie's reviews into *positive* and *negative* categories.

- **level**: The Python logging level (`INFO`, `DEBUG`, `ERROR`, etc.) is recorded exactly as emitted, enabling severity-based filtering.
- **message**: The core message passed to `logger.info()`, `logger.error()`, etc. This contains a human-readable summary of the event (e.g., model load success, new prediction received).
- **logger**: Identifies the logger instance that produced the event. In the backend, most logs originate from a module-specific logger, enabling modular filtering.

In addition to these base attributes, the handler automatically ingests all values provided in the `extra={}` parameter of Python log calls. These fields become native JSON keys inside Elasticsearch, forming the backbone of the observability pipeline.

3. Automatic Enrichment with MLOps Metadata

A unique feature of this logging system is that the backend injects structured operational metadata directly into logs. The `record.extra` context is used to propagate domain-specific fields, which are essential for monitoring the behaviour of the deployed model and the overall system. Typical enriched fields include:

- `event_type` — distinguishes logical categories such as:
  - `MODEL_LOAD`

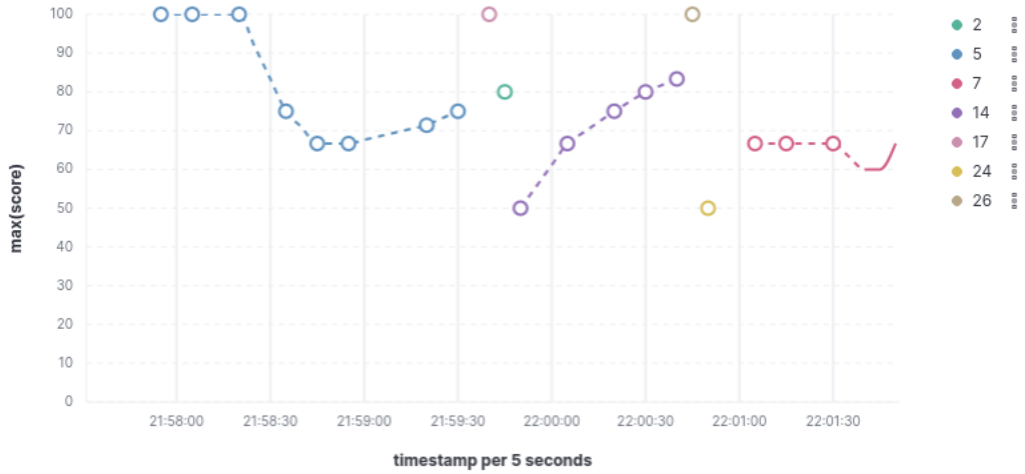

Figure 11: Temporal evolution of freshness scores for all queried movies across the observation window.

  - `PREDICTION`
  - `DB_WRITE`
  - `TRAFFIC`
  - `LIFECYCLE`
- `model_version`: the MLflow Run ID extracted from the loaded model artefact; enables version-level drift or performance tracking.
- `sentiment`: produced during inference requests; maps predicted output to `positive` or `negative`.
- `movie_id`: included for prediction events, allowing per-movie analytics.
- `endpoint`: recorded by traffic logs for tracing user interaction patterns (e.g., `/predict`, `/seed`, `/movies`).
- `status`: captures success or failure states for critical operations such as DB seeding, model loading, and inference.

These enriched fields transform log documents into telemetry-grade observability signals, enabling sophisticated queries like:

- Model performance and error rates grouped by `model_version`
- Prediction frequency per `movie_id`
- Endpoint-level traffic heatmaps
- Failure clustering based on `event_type`

This approach provides a lightweight, code-native alternative to tools like Fluentd or Logstash, while still maintaining full compatibility with Kibana dashboards. The plots in this report demonstrate the rich analytical capabilities available through Kibana, providing clear insights into application behaviour and activity.

9 Summary

This project successfully delivered a robust, full-stack platform for real-time movie review sentiment analysis, leveraging an end-to-end MLOps pipeline. The core achievement lies in the seamless integration and automation of industry-standard DevOps and MLOps tools to create a fully reproducible and scalable application.

9.1 Key Project Components and Technologies

The platform's architecture is built upon four fundamental pillars:

1. **Continuous Integration/Continuous Deployment (CI/CD):** The entire lifecycle, from code commit to production deployment, is orchestrated by **Jenkins** using a Declarative Groovy pipeline. Secure secret management is handled via Jenkins Credentials and **Ansible Vault**.
2. **Containerized Infrastructure:** The application (FastAPI backend and Streamlit frontend) and all supporting services (PostgreSQL, Elasticsearch, Kibana) are fully encapsulated in Docker containers, ensuring environment consistency.
3. **MLOps Core:** The sentiment analysis model is managed using **MLflow** for experiment tracking and artifact promotion, and **Data Version Control (DVC)** ensures dataset immutability and reproducibility for every training run.
4. **Orchestration and Scaling:** The application and its monitoring stack are deployed, managed, and scaled on a **Kubernetes (Minikube)** cluster, with configuration automated by **Ansible**. Critical features include:
    - Secure credential injection using Kubernetes Secrets.
    - Durable state management for PostgreSQL and Elasticsearch via **Persistent Volumes (PV/PVC)**.
    - Dynamic scaling for the backend via a **Horizontal Pod Autoscaler (HPA)**.
    - External access controlled by **Ingress** routing.

In conclusion, the Rotten Potatoes project serves as a comprehensive demonstration of modern software production engineering principles, successfully integrating CI/CD automation, a robust MLOps pipeline, and a fully containerized, scalable Kubernetes deployment.

Project Repository

All codes and documentation can be found here: https://github.com/PraveenPeterJay/sentiment-mlops