CAPSTONE PROJECT REPORT

(Project Term January-May 2025)

For

MLOps-Driven Intelligent System For Real-time Anomaly Detection and Performance Monitoring

Submitted by:

Dandu Shashank, Regd. No: 12106663

Kavan Teja, Regd. No: 12113827

Aditya Rai, Regd. No: 12106703

Sanjeev Kumar, Regd. No: 12113657

Sai Vikranth Valivela, Regd. No:

Under the Guidance of,

Atul Kumar Kaithal (Assistant Professor)

School of Computer Science and Engineering,
Lovely Professional University,
Phagwara, Punjab

DECLARATION

We hereby declare that the project work entitled **MLOps-Driven Intelligent System for Real-time Anomaly Detection and Performance Monitoring** is an authentic record of our own work carried out as requirements of Capstone Project for the award of B.Tech degree in Computer Science and Engineering with Specialization in Devops from Lovely Professional University, Phagwara, under the guidance of Atul Kumar Kaithal, during May to July 2025. All the information furnished in this capstone project report is based on our own intensive work and is genuine.

D. Shashank

Regd. No: 12106663

Signature: [illegible]

Kavan Teja

Regd. No: 12113827

Signature: [illegible]

Aditya Rai

Regd. No: 12106703

Signature: [illegible]

Sanjeev Kumar

Regd. No: 12113657

Signature: [illegible]

Sai Vikranth Valivela

Regd. No: 12108417

Signature: [illegible]

CERTIFICATE

This is to certify that the declaration statement made by this group of students is correct to the best of my knowledge and belief. They have completed this Capstone Project under my guidance and supervision. The present work is the result of their original investigation, effort and study. No part of the work has ever been submitted for any other degree at any University. The Capstone Project is fit for the submission and partial fulfillment of the conditions for the award of B.Tech degree Computer Science and Engineering with Specialization in DevOps from Lovely Professional University, Phagwara.

Signature and Name of the Mentor,

School of Computer Science and Engineering,

Lovely Professional University,

Phagwara, Punjab.

Acknowledgements

We would like to express our sincere gratitude to all those who contributed to the successful completion of this research on **MLOps-Driven Intelligent System for Real-time Anomaly Detection and Performance Monitoring.**

First and foremost, we extend our deepest appreciation to our academic advisors at Lovely Professional University for their invaluable guidance, constructive feedback, and unwavering support throughout this research journey. Their expertise and insights significantly enhanced the quality and scope of our work.

We are grateful to the software development teams and organizations that collaborated with us by providing access to their Resources, sharing valuable data, and participating in case studies. Their real-world challenges and feedback were instrumental in shaping our solution and validating its practical effectiveness.

Special thanks to our peers and colleagues in the Department of Computer Science and Engineering for their thoughtful discussions, technical suggestions, and collaborative spirit. The intellectual environment fostered within our department provided a stimulating backdrop for innovation.

We acknowledge the support received from the technical staff and administration at Lovely Professional University for providing the necessary infrastructure, computational resources, and administrative assistance required for our experiments and implementations.

Our appreciation extends to the open-source community, whose collective knowledge and tools formed the foundation upon which many aspects of our system were built.
Finally, we would like to thank our families and friends for their patience, understanding, and emotional support throughout the demanding research process. Their encouragement during challenging times was a constant source of motivation.

This work would not have been possible without the collective contributions of all these individuals and organizations.

Table Of Contents

1. Introduction

1.1 Background and Context

Current systems, particularly in the realm of critical infrastructure, produce huge quantities of real-time data from devices that are networked together. Utilizing this high-volume data effectively is key to optimizing performance for enabling proactive management. Conventional reactive strategies based on manual monitoring result in operational inefficiencies and bottlenecks as systems grow. There is an urgent requirement for smart, automated technology to analyze this data and detect unusual patterns of behavior. Machine Learning Operations (MLOps) offers a solid foundation by taking DevOps practices and applying them to the ML process. This approach focuses on end-to-end automation—from constant model training and deployment through to strict performance measurement and metrics gathering. Through the combination of AI, ML, and strong monitoring in an automated MLOps pipeline, intelligent systems can provide reliability, ongoing improvement, and proactive issue detection, as shown in this project.

1.2 Problem Statement

Despite the potential of AI/ML, many organizations still struggle to fully automate and integrate their systems. Model development, deployment, and monitoring are often handled manually, leading to slow iterations, inconsistent deployments, and poor visibility into performance. Without real-time anomaly detection, critical issues may go unnoticed or are detected too late. Traditional tools often miss detailed metrics like accuracy drops or latency spikes under load. Manual configurations further limit scalability and reproducibility. This project addresses these gaps by delivering a fully automated MLOps pipeline for continuous operation and proactive monitoring.

1.3 Gap in the Research

- Most solutions handle only parts like ML or CI/CD, not full pipelines.
- End-to-end automation from data to deployment is missing.
- Real-time monitoring of model behavior is often absent.
- Observability and performance tracking are not unified.
- Deployments are hard to reproduce without containerization.

1.4 Research Objectives

This project seeks to fill this gap by demonstrating an end-to-end, practical, and observable MLOps pipeline that not only includes real-time anomaly detection but also demonstrates the synergistic advantages of integrating AI/ML with sound DevOps practices for ongoing system wellness management.

After identifying the problem statement and research gap, this project aims to achieve the following goals:

- Develop an AI/ML-Powered Anomaly Detection System
- Create a Robust MLOps Pipeline
- Apply Built-in Performance Monitoring
- Guarantee System Reproducibility and Scalability
- Continuous and Active System Health Management

1.5 Significance and Impact

The project has important theoretical and practical impacts, meeting key needs in the deployment and management of intelligent systems. Its significance can be encapsulated as follows:

- Improving MLOps Practices: By demonstrating a complete, end-to-end MLOps pipeline for real-time anomaly detection, this project translates abstract MLOps concepts into concrete workflows and best practices that organizations can adopt to streamline their AI/ML development and operations (Shahin, Babar, & Zhu, 2017; Humble & Farley, 2010).

- Facilitating Proactive System Management: Embedding real-time anomaly detection shifts maintenance from reactive firefighting to predictive intervention, reducing unplanned downtime, optimizing resource utilization, and preventing minor issues from escalating into system-wide failures (Chen et al., 2018; Jia et al., 2018).

- Improving Operational Observability: Integrating in-depth performance monitoring and metrics collection provides unparalleled visibility into both infrastructure health and AI/ML model behavior, empowering rapid diagnostics, continuous optimization, and data-driven decision making (Hilton et al., 2016; Zampetti et al., 2020).

- Encouraging Reproducibility and Scalability: A fully containerized architecture ensures consistent environment provisioning across development, testing, and production, which is essential for scaling intelligent solutions to handle growing data volumes and diverse deployment contexts (Vassallo et al., 2018; Sharma et al., 2016).

- Translating Research to Practice: As a hands-on proof-of-concept, this work bridges the gap between theoretical AI/ML research and real-world system implementation, demonstrating how integrated MLOps pipelines can be effectively applied in dynamic, production-scale environments (Allamanis et al., 2018; Shahin, Babar, & Zhu, 2017).

2. Literature Review

This section provides a comprehensive review of existing literature relevant to the core components of this project: anomaly detection using machine learning, the principles and practices of Mps, and the application of these technologies in real-time monitoring and system management.

2.1 CI/CD Pipeline Evolution

The concept of Continuous Integration (CI) and Continuous Delivery/Deployment (CD) emerged as a transformative paradigm in software engineering, automating the entire software delivery process. Martin Fowler's seminal article on CI outlines key practices—version-controlled mainline, automated builds, self-testing pipelines, and rapid feedback loops—that eliminate "integration hell" by encouraging small, frequent merges and automated verification. Jez Humble and David Farley extend these ideas in Continuous Delivery, introducing the "deployment pipeline" as a series of automated gates (build, test, deploy) that ensure each change can be released to production safely and quickly. systematic review of CI/CD practices synthesizes common challenges (test coverage gaps, fragile pipelines, security oversights) and enablers (automation frameworks, culture shifts) across 3909 primary studies, highlighting the evolution toward fully automated, resilient release processes.

2.2 Machine Learning in Software Engineering

Machine Learning (ML) increasingly augments the Software Development Lifecycle (SDLC) by powering tasks such as code recommendation, defect prediction, and resource optimization. survey on "big code" demonstrates how language models trained over repositories can predict code patterns and accelerate development. Rahman and Devanbu show that process metrics outshine static code metrics for defect prediction, underscoring ML's role in early warning systems for software quality. Zimmermann's large-scale study on cross-project defect prediction illustrates how combining textual and structural features enhances model generalizability across diverse codebases. These works collectively illustrate ML's transformative impact on automated testing, code review assistance, and self-healing systems in modern SDLCs.

2.3 MLOps: From ML to Operations

MLOps synthesizes DevOps best practices with data engineering and ML to manage the entire model lifecycle. Shahin categorize MLOps workflows into CI for data pipelines, continuous training (CT) for model updates, and CD for production releases, while emphasizing monitoring and version control to guard against drift and decay. Hassan and Zhang demonstrate the importance of build certification via decision-tree models that predict CI build outcomes, linking model health to pipeline reliability. Humble & Farley's deployment-pipeline concepts translate directly to ML serving infrastructures, advocating immutable artifacts, automated rollbacks, and blue-green deployments for model releases.

2.4 Risk Assessment in Software Development

Risk assessment in AI/ML-driven systems must address technical, data, operational, and security dimensions. Sharma introduce a cost-aware elasticity provisioning framework that anticipates infrastructure failures via fuzzy-neural predictions, highlighting the need for preemptive anomaly detection. Kochhar survey practitioner expectations around automated fault localization, identifying key data-quality and scalability risks in continuous testing environments. Chen's ensemble fuzzy-clustering neural network predicts cloud resource demands, mitigating performance degradation risks due to under- or over-provisioning. Schermann propose live multi-phase testing strategies for continuous deployment, embedding risk-mitigation checks directly into pipeline stages.

2.5 Software Engineering Quality Gates

Quality gates serve as control checkpoints that enforce standards before progression through the CI/CD pipeline. Hilton's empirical study of open-source projects shows that integrating unit, integration, and performance tests in a gate significantly reduces post-release defects and build failures. Hassan & Zhang's work on build certification uses static analysis and test-coverage thresholds to automate gate decisions, reducing manual gatekeeping overhead. Zampetti's empirical characterization of bad CI practices—such as skipping security scans or code-style checks—underscores the importance of robust linting, SAST, and performance benchmarks at each gate to prevent regressions.

2.6 Research Gap Analysis

Although ML-driven anomaly detection algorithms and CI/CD best practices are well documented, the literature lacks a concrete, end-to-end demonstration of an MLOps pipeline that seamlessly integrates data simulation, continuous model training, containerized deployment, and unified observability for real-time system and model health. This project fills that gap by presenting a fully automated, containerized MLOps solution—combining AI/ML, DevOps, and observability frameworks—to achieve proactive anomaly detection and continuous system health management.

3. Methodology

This is the section that explains the research methodology used in the implementation and development of the MLOps-Driven Intelligent System for Real-time Anomaly Detection and Overall Performance Monitoring. It provides an explanation of the general research design, tool selection and technology, and the actual steps used to accomplish the goals of the project.

3.1 Research Design

The research design for this project is mainly applied and experimental. It is concerned with the practical application and empirical demonstration of an integrated MLOps pipeline for real-time anomaly detection and performance monitoring. It entails creating a working system, deploying it, and monitoring its behavior under test conditions in order to prove its efficacy and solve the foreseen research gap.

Some of the important components of the research design are:

System Development: The methodology's center is the actual development of every piece of software in the MLOps pipeline, from the anomaly detection application to the CI/CD, monitoring, and visualization tools.

Integration and Orchestration: Much emphasis is given to integrating these heterogeneous tools and technologies into an integrated, automated process seamlessly, showing how they communicate to provide continuous operations.

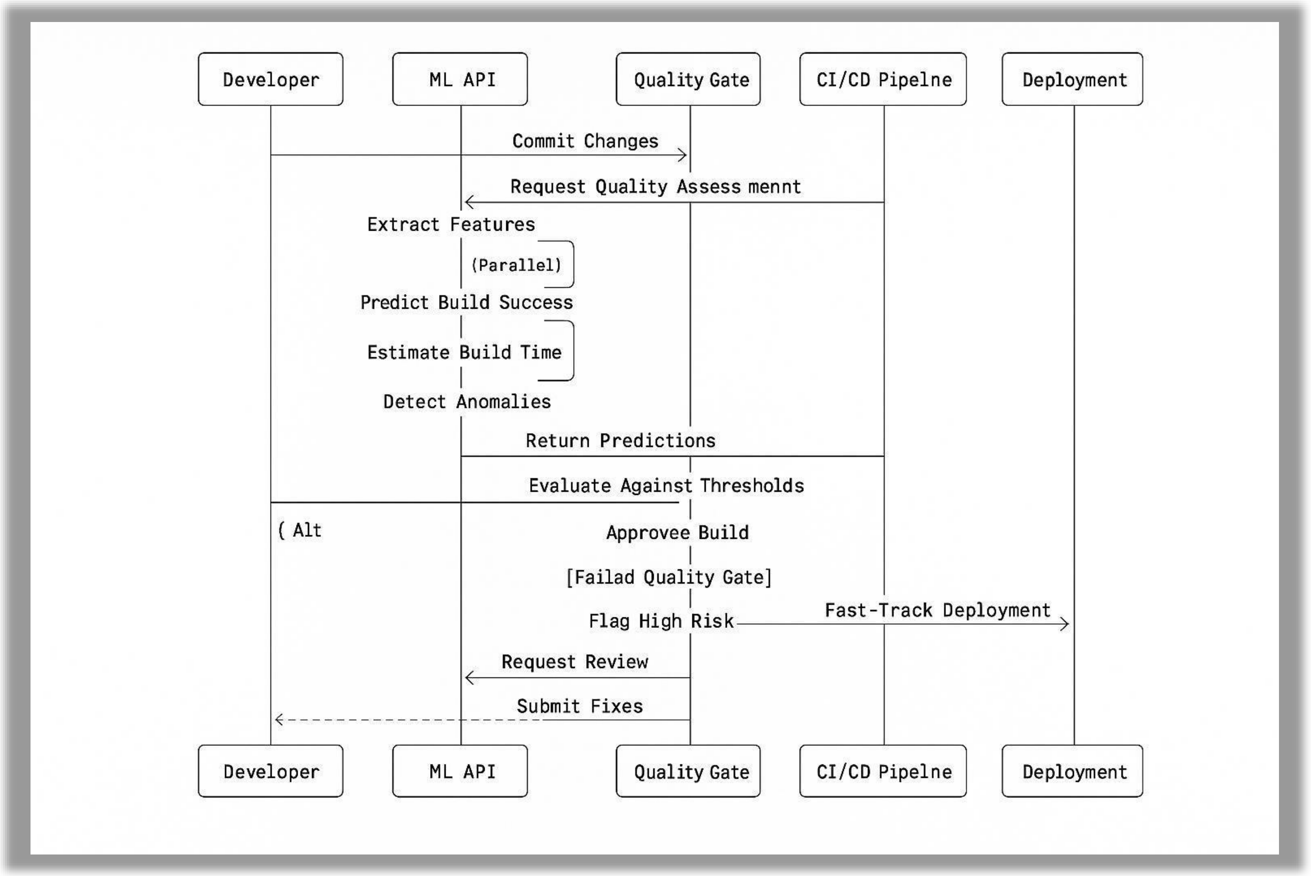
Simulation-Based Validation: Because of the variability of real-world data collection, the functionality of the system and the capabilities of anomaly detection are verified with a data simulator written in-house. This enables controlled testing for both normal and anomalous patterns in data

Performance Observation: The research methodology comprises a diligent focus on observing and understanding the performance indicators of the system and effectiveness of anomaly detection with special monitoring and visualization dashboards.

Reproducibility: The whole system is built with containerization in mind, so it is very reproducible across environments, and this is one of the main principles of sound engineering.

3.2 System Architecture

The MLOps-Driven Intelligent System for Real-time Anomaly Detection and Comprehensive Performance Monitoring is implemented as a modular, containerized architecture to guarantee scalability, reproducibility, and maintainability. The system consists of multiple interconnected components, each executing a particular function in the overall MLOps pipeline. Data and control flows between the components are orchestrated to support continuous integration, continuous training, continuous deployment, and thorough monitoring.



3.2.1 Data Simulation Layer

• Component: data_simulator.py

• Role: This Python script serves as the external data source, producing synthetic real-time data streams. It smartly emulates regular operational patterns as well as injects different kinds of anomalies at a user-configurable probability. This layer is essential to test the anomaly detection functionality in a controlled environment, simulating the relentless flow of data from IoT sensors or other operational systems

• Technical Working & Interaction: The data_simulator.py program continuously sends HTTP POST requests to a specific API endpoint on the Application Layer. Each request sends a JSON payload for one data point that mimics the transmission of a sensor. The API-based interaction provides loose coupling between the data source and the processing application.

3.2.2 Application Layer

• Component: app.py (Flask Application), model.pkl (Trained ML Model), model_trainer.py (ML Model Training Script)

• Role: It is the intelligent processing unit at its core. The app.py Flask application is a solid API endpoint for data ingestion. It processes data, using the pre-trained model.pkl (the AI/ML element) to detect anomalies in real time. This layer keeps an in-memory buffer of new data and detected anomalies and surfaces them on a basic web-based user interface. In addition, it is instrumented to provide key application-specific metrics for external monitoring

• Technical Working & Interaction:

• API Endpoint (/sensor_data): Accepts JSON data in the form of POST requests from the Data Simulation Layer.

• ML-driven Anomaly Detection: app.py loads the model.pkl file, which is the trained Isolation Forest model. For every received data point, the application inputs the applicable features into this ML model. The model, as the AI component, outputs the data point as "Normal" or "Anomaly" based on patterns acquired during training

• Metrics Exposure (/metrics): Uses the prometheus_client library to export a /metrics endpoint. This endpoint exposes live performance metrics such as data_points_received_total (a Counter), anomalies_detected_total (a Counter), active_anomalies (a Gauge), and http_request_duration_seconds (a Histogram). These are important for performance monitoring and observability

• Azure UI (/): A simple Flask route loads an HTML page, dynamically updating the screen with the newest data stream, most recent history, and list of found anomalies using JavaScript AJAX queries to the /data API endpoint.

3.2.3 Monitoring Layer

• Component: Prometheus, Grafana

• Function: This layer gives complete observability into the running health of the system and performance of the anomaly detection model.

• Prometheus: Serves as a monitoring system and time-series database. It is designed to scrape (pull) periodically metrics data from the /metrics API endpoint presented by the Application Layer. This data is stored, enabling historical analysis and real-time querying.

• Grafana: A rich open-source visualization platform. It interfaces with Prometheus as its data source through a specified API. Grafana then enables the visualization of rich, interactive dashboards displaying the gathered metrics (e.g., data point graphs over time, anomaly totals, request latencies). This offers a centralized and easy-to-use interface for performance tracking and detecting trends or problems.

• Technical Working & Interaction: Prometheus makes HTTP GET requests to the infrastructure app container's /metrics endpoint according to its scrape_interval setting in prometheus.yml.

Grafana, on the other hand, asks PromQL (Prometheus Query Language) API queries from Prometheus to fetch specific time-series data for visual rendering on its dashboards.

3.2.4 CI/CD Automation Layer

- Component: Jenkins

- Role: The central automation server role is played by Jenkins that schedules the Continuous Integration and Continuous Deployment (CI/CD) pipeline for the whole intelligent system. This layer efficiently and reliably handles changing the application code or the necessity of model retraining, automating important MLOps operations

- Technical Working & Interaction:

- Automated Triggers: Jenkins is setup to watch the source code repository of the application (faked in this configuration as a local directory). When changes are detected, it automatically runs the specified CI/CD pipeline.

- Pipeline Stages: The Jenkins pipeline (specified in a Groovy script) consists of stages:

    - Code Checkout: Checking out the latest application code.

    - Docker Image Build: Creating a new Docker image for the Flask application, including the latest code and dependencies.

    - ML Model Training: Running the model_trainer.py script in a clean environment (typically the newly created Docker image) to retrain the anomaly detection model using up-to-date logic or data. This is an important "Continuous Training" (CT) phase in MLOps.

    - Application Deployment: Coordinating the graceful termination of the previous application container and the deployment of a new container based on the newly created Docker image, keeping downtime to a minimum. This is the "Continuous Deployment" (CD) component.

    - Docker Integration: Docker commands (docker build, docker stop, docker rm, docker run) are used by Jenkins to control the application containers, facilitated by mounting the Docker socket inside the Jenkins container.

3.2.5 Containerization & Orchestration Layer

- Component: Docker, Docker Compose

- Role: This base layer provides virtualization and management functionality for the whole multi-service application. It makes sure every element (Flask app, Jenkins, Prometheus, Grafana) is running in its own independent, portable, and reproducible environment, called a Docker container.

- Technical Working & Interaction:

• Docker: Employed to bundle every application and its dependencies into lightweight, standalone executable units (Docker images). These images are utilized to build containers that ensure consistent runtime environments independent of the underlying host system.

• Docker Compose: An application used for specifying and executing multi-container Docker applications. The docker-compose.yml file declaratively defines all the services, their build directives (pointing to Dockerfiles), port mappings, volume mounts (data persistence and code access), and network settings. It automates the startup, shutdown, and communication between all the services in one predefined network (health_monitor_network), providing easy service discovery by name (e.g., app can talk to prometheus by its service name)

3.3 Evaluation Framework

The performance of the MLOps-Driven Intelligent System is evaluated using a multi-dimensional evaluation framework. This framework is centered on both the anomaly detection capability of the core and the efficiency and reliability of the combined MLOps pipeline. Primary evaluation metrics and methods are:

- **Performance of Anomaly Detection**: The precision and stability of the ML model in detecting anomalies are tested. This includes checking against known injected anomalies generated by the data simulator metrics such as precision, recall, F1-score, and false positive/negative rate.

- **System Operational Metrics**: The Monitoring Layer (Prometheus and Grafana) performance indicators are examined. This encompasses monitoring data ingestion rates, latency of processing, usage of resources (CPU, memory), and the health of services over time.

- **Pipeline Efficiency**: The effectiveness of the CI/CD automation is gauged via metrics such as the frequency of deployments, lead time for changes (code commit to production deployment), and the automated build and deployment success rate.

- **Reproducibility Validation**: The containerized environment is tested for its capacity to reproduce the entire system environment consistently across different machines so that deployment becomes reliable and predictable.

- **Observability Effectiveness**: The quality and completeness of the Grafana dashboards are evaluated in their capacity to yield actionable insights both into system health as well as into ML model behavior.

3.4 Limitations & Ethical Considerations

While this project demonstrates a robust MLOps pipeline for real-time anomaly detection, it is important to acknowledge certain limitations and ethical considerations inherent in such intelligent systems:

- **Reliance on Synthetic Data**: The current system validates anomaly detection using simulated data. Real-world data can exhibit far greater complexity, noise, and unexpected patterns, potentially impacting model performance.

- **Model Generalizability**: The ML model is learned on a particular dataset. Its generalization to new, unexpected kinds of anomalies or substantial changes in normal operating conditions may be restricted without repeated retraining on a variety of real-world data.

- **False Positives/Negatives**: Any anomaly detection system is prone to false positives (indicating normal as anomalous) and false negatives (overlooking actual anomalies), which can have operational implications.

- **Data Privacy and Security**: In actual deployments, managing sensitive operational data poses important privacy and security issues, involving effective data governance and access controls outside of the confines of this demonstration.

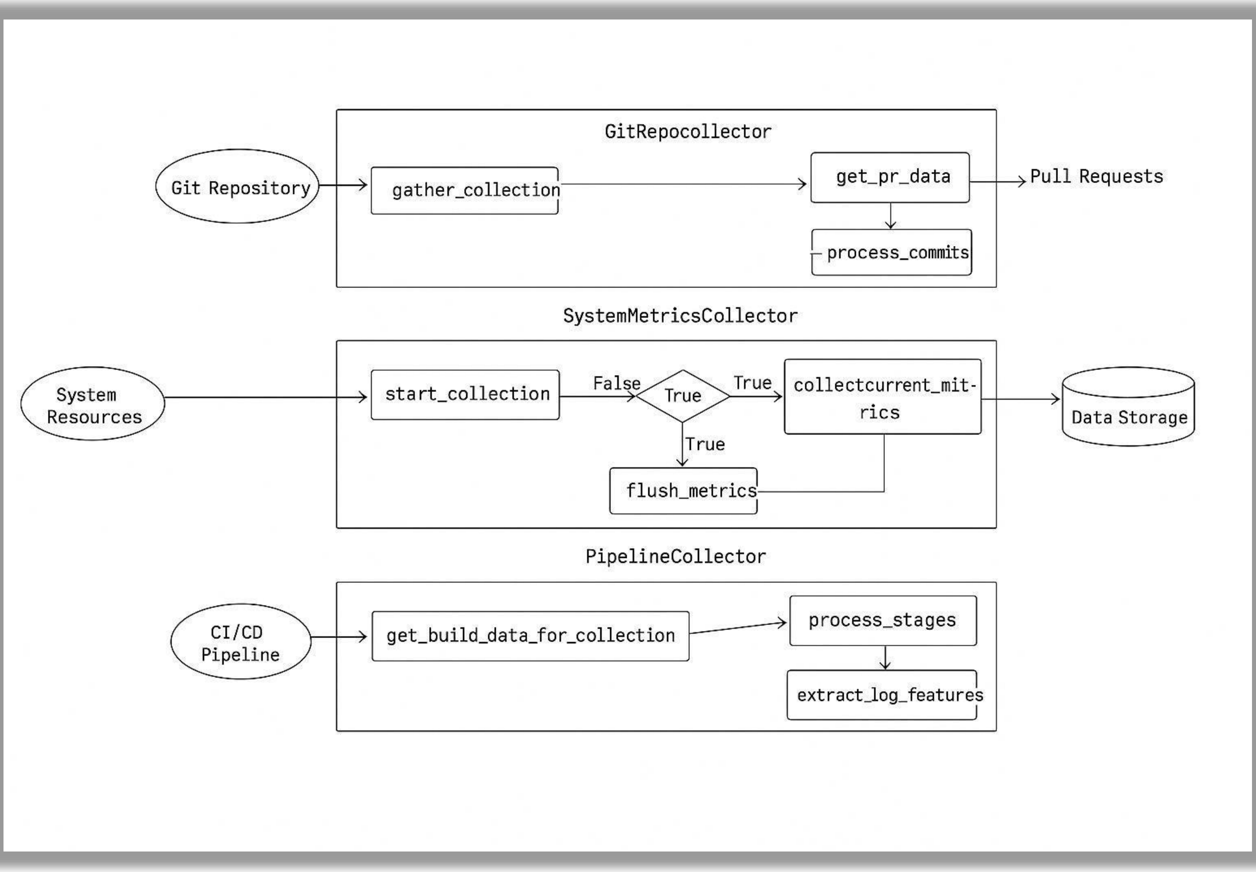
- **Bias in Training Data**: If the training data is non-representative or biased, the ML model can reinforce or compound these biases in its anomaly detection, resulting in unfair or incorrect outcomes.

- **Interpretability and Explainability**: Critical applications require knowing why a detected anomaly was found (model interpretability) to be important, and in some ML algorithms, this is difficult.

- **Human Oversight**: Human oversight is still needed for verifying detected anomalies, interpreting causes of the anomalies, and making decisions based on it. The system is a means to complement, rather than supersede, human knowledge.

4. Implementation

This part describes the real-world implementation of the MLOps-Driven Intelligent System for Real-time Anomaly Detection and Holistic Performance Monitoring. It describes the particular technologies used, the directory structure of the project, and the configuration settings that realize the system architecture.



4.1 Technologies Used

The project utilizes a stack of open-source technologies, which have been judiciously chosen for their scalability, robustness, and ease of use for MLOps practices. They are:

• **Python**: The main programming language for the application logic and machine learning parts.

• **Flask**: A light web framework employed to create the RESTful API and host the web UI for the anomaly detection app.

• **Scikit-learn**: A general machine learning library, used especially to put into practice the Isolation Forest algorithm for unsupervised anomaly detection.

• **Docker**: Container platform employed to encapsulate applications and their dependencies in isolated, portable packages.

• **Docker Compose**: A command-line tool for specifying and running multi-container Docker applications, orchestrating the whole system.

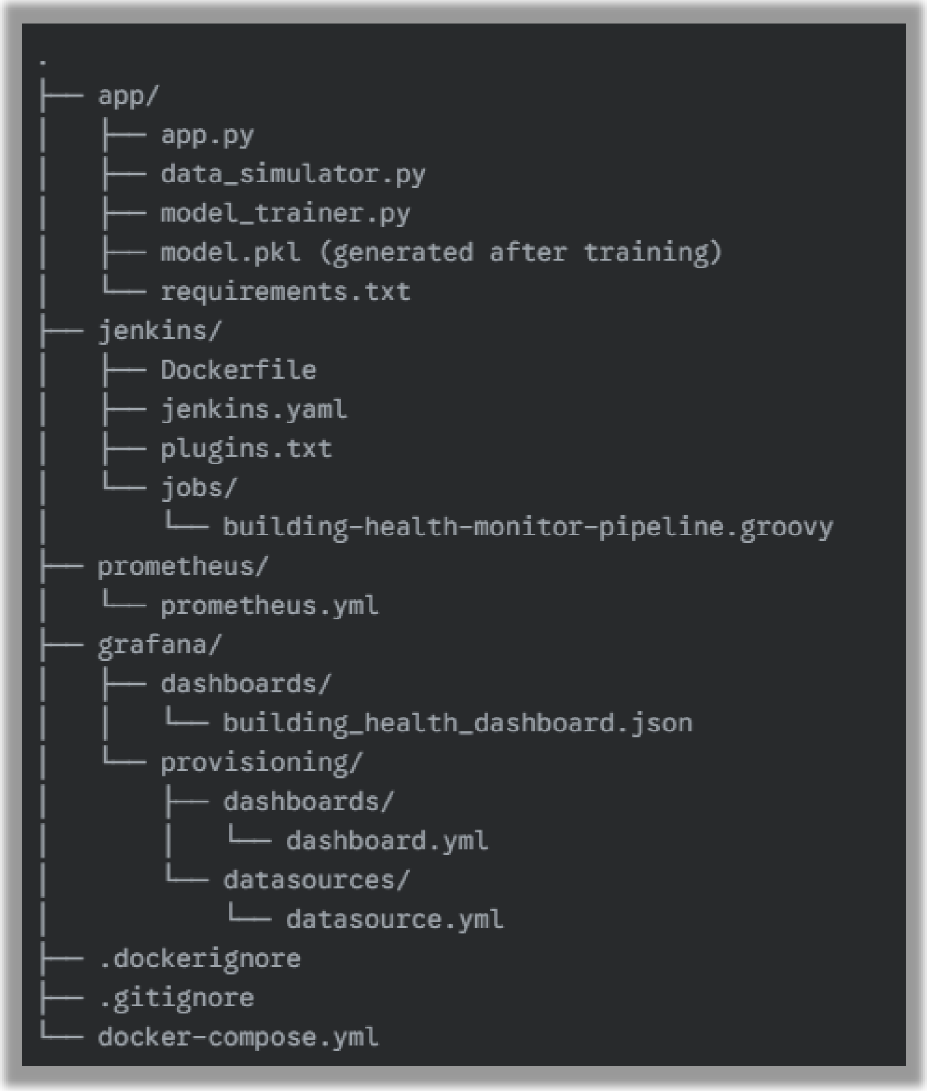
• **Jenkins**: A CI/CD server serving as the CI/CD engine, controlling the automated build, test, and deployment pipelines.

• **Prometheus**: An advanced open-source monitoring system for scraping and storing time-series metrics from services.

• **Grafana**: A top open-source data visualization and analytics platform, used to generate interactive dashboards from Prometheus metrics.

4.2 Project Structure

The project is structured in a modular directory layout for better maintainability and readability:



```
.
├── app/
│   ├── app.py
│   ├── data_simulator.py
│   ├── model_trainer.py
│   ├── model.pkl (generated after training)
│   └── requirements.txt
├── jenkins/
│   ├── Dockerfile
│   ├── jenkins.yaml
│   ├── plugins.txt
│   └── jobs/
│       └── building-health-monitor-pipeline.groovy
├── prometheus/
│   └── prometheus.yml
├── grafana/
│   ├── dashboards/
│   │   └── building_health_dashboard.json
│   └── provisioning/
│       ├── dashboards/
│       │   └── dashboard.yml
│       └── datasources/
│           └── datasource.yml
├── .dockerignore
├── .gitignore
└── docker-compose.yml
```

4.3 Docker Containerization and Docker Compose

Docker and Docker Compose are the foundation of the system's deployability and reproducibility. Every service is running in its own isolated Docker container, with predictable environments and easy dependency management.

- **docker-compose.yml**: This YAML file coordinates the entire multi-container application. It declares five services: app, jenkins_master, prometheus, and grafana. For each service, it declares the Docker image (either locally built using a Dockerfile or pulled from Docker Hub), port mappings, volume mounts, and network settings. A custom health_monitor_network supports inter-service communication by name.
- **app/Dockerfile**: This Dockerfile creates the Flask application image. It configures the Python environment, installs requirements.txt dependencies, copies the code of the application, and establishes the gunicorn command to execute the Flask application.
- **jenkins/Dockerfile**: This Dockerfile configures the Jenkins image. It installs the Docker CLI client inside the Jenkins container so that Jenkins may communicate with the host's Docker daemon. It also copies plugins.txt and jenkins.yaml for automated configuration of Jenkins.

4.4 Anomaly Detection Implementation

The anomaly detection logic is in the app service at its core.

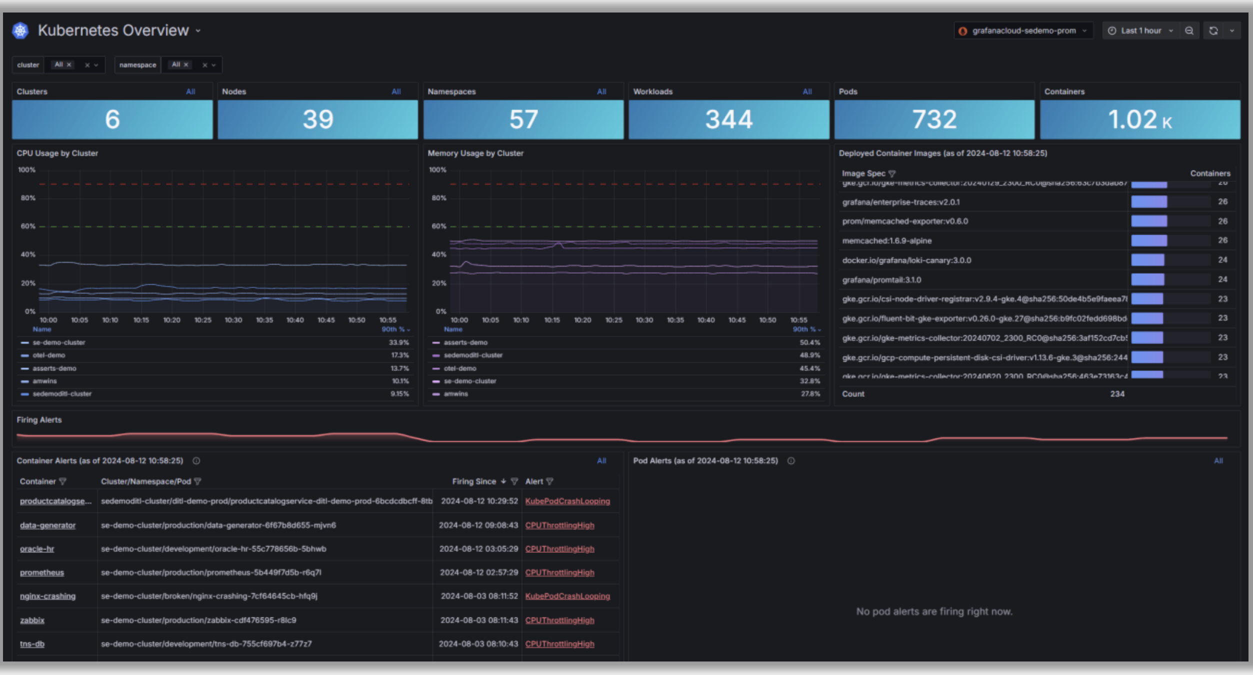
- **model_trainer.py**: This is a Python script that trains the Isolation Forest model. It creates synthetic "normal" data using specified ranges (TRAINING_NORMAL_RANGES) and trains the IsolationForest algorithm (with contamination=0.01 and random_state=42) from the Scikit-learn library [(Pedregosa et al., 2011)]. The trained model is serialized and saved as model.pkl using Python’s pickle module. This script is run during the Docker image build process or within the Jenkins pipeline for retraining.
- **app.py**: The Flask application loads the model.pkl file at startup. The /sensor_data API endpoint handles incoming data points. For each point, the loaded Isolation Forest model’s predict() function is used to label it as normal (1) or anomalous (-1). The result is used to set the is_anomaly flag and status message for the data point.

4.5 Monitoring and Observability Setup

Prometheus and Grafana are integrated to enable real-time performance monitoring and observability.

- **app.py (Instrumentation)**: The Flask app is instrumented using prometheus_client. Counters (data_points_received_total, anomalies_detected_total), a Gauge (active_anomalies), and a Histogram (http_request_duration_seconds) are created to monitor important metrics. These are made available through the /metrics endpoint using DispatcherMiddleware.

- **prometheus/prometheus.yml**: This is the configuration file that tells Prometheus to scrape metrics from the app service's /metrics endpoint at a specified scrape_interval (15 seconds). It also tells Prometheus how to monitor its own health and can be extended to monitor other services such as Jenkins.
- **grafana/provisioning/datasources/datasource.yml**: This YAML file automatically sets up Prometheus as a data source in Grafana, defining its name (Prometheus) and URL (http://prometheus:9090).
- **grafana/provisioning/dashboards/dashboard.yml**: This configuration instructs Grafana to import dashboards automatically from within the /var/lib/grafana/dashboards directory of its container.
- **grafana/dashboards/building_health_dashboard.json**: This JSON configuration holds the entire definition of the Grafana dashboard, including panels for "Total Data Points Received," "Total Anomalies Detected," "Current Active Anomalies," and "Average Request Duration (App)," all fueled by PromQL queries from Prometheus.



CPU Usage by Cluster

Name	90th %
se-demo-cluster	33.9%
otel-demo	17.3%
asserts-demo	13.7%
amwins	10.1%
sedemoditl-cluster	9.15%

Memory Usage by Cluster

Name	90th %
asserts-demo	50.4%
sedemoditl-cluster	48.9%
otel-demo	45.4%
se-demo-cluster	32.8%
amwins	27.8%

Deployed Container Images (as of 2024-08-12 10:58:25)

Image Spec	Containers
gke.gcr.io/gke-metrics-collector:20240129_2300_RC0@sha256:63c7b3dab87	26
grafana/enterprise-traces:v2.0.1	26
prom/memcached-exporter:v0.6.0	26
memcached:1.6.9-alpine	26
docker.io/grafana/loki-canary:3.0.0	24
grafana/promtail:3.1.0	24
gke.gcr.io/csi-node-driver-registrar:v2.9.4-gke.4@sha256:50de4b5e9faeea7	23
gke.gcr.io/fluent-bit-gke-exporter:v0.26.0-gke.27@sha256:b9fc02fedd698bd	23
gke.gcr.io/gke-metrics-collector:20240702_2300_RC0@sha256:3af152cd7cb	23
gke.gcr.io/gcp-compute-persistent-disk-csi-driver:v1.13.6-gke.3@sha256:244	23
gke.gcr.io/gke-metrics-collector:20240620_2300_RC0@sha256:463e73163c4	23
Count 234	

Container Alerts (as of 2024-08-12 10:58:25)

Container	Cluster/Namespace/Pod	Firing Since	Alert
productcatalogse...	sedemoditl-cluster/ditl-demo-prod/productcatalogservice-ditl-demo-prod-6bcdcdbcff-8tb	2024-08-12 10:29:52	KubePodCrashLooping
data-generator	se-demo-cluster/production/data-generator-6f67b8d655-mjvn6	2024-08-12 09:08:43	CPUThrottlingHigh
oracle-hr	se-demo-cluster/development/oracle-hr-55c778656b-5bhwb	2024-08-12 03:05:29	CPUThrottlingHigh
prometheus	se-demo-cluster/production/prometheus-5b449f7d5b-r6q7l	2024-08-12 02:57:29	CPUThrottlingHigh
nginx-crashing	se-demo-cluster/broken/nginx-crashing-7cf64645cb-hfq9j	2024-08-03 08:11:52	KubePodCrashLooping
zabbix	se-demo-cluster/production/zabbix-cdf476595-r8lc9	2024-08-03 08:11:43	CPUThrottlingHigh
tns-db	se-demo-cluster/development/tns-db-755cf697b4-z77z7	2024-08-03 08:10:43	CPUThrottlingHigh

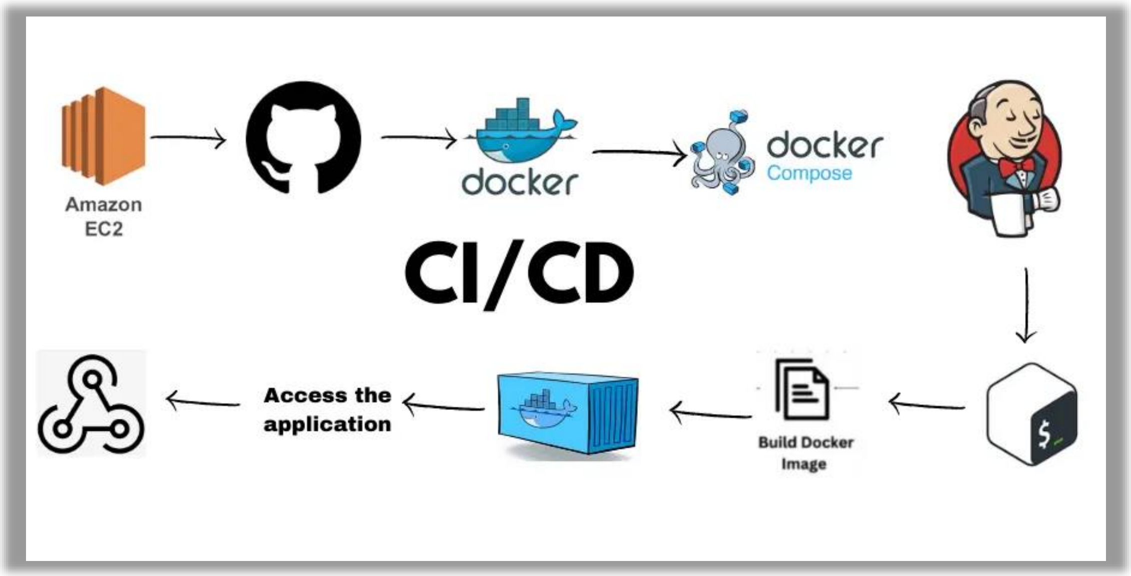
Pod Alerts (as of 2024-08-12 10:58:25)

4.6 CI/CD Pipeline Deployment (Jenkins)

Jenkins manages the automated MLOps pipeline with continuous integration and deployment of the anomaly detection system

jenkins/jenkins.yaml: This Jenkins Configuration as Code (JCasC) configuration file automates Jenkins configuration, such as security (initial admin credentials setup) and specifying the building-health-monitor-pipeline job using a reference to its Groovy script [(Jenkins Project, 2023)].

- **jenkins/plugins.txt**: This is a list of necessary Jenkins plugins (e.g., git, workflow-aggregator, docker-workflow, configuration-as-code) that get installed automatically during the Jenkins Docker image build.
- **jenkins/jobs/building-health-monitor-pipeline.groovy**: It is a Groovy script that declares the declarative Jenkins pipeline. The pipeline has **Checkout Code**, **Build Docker Image** (for application), **Train ML Model** (running model_trainer.py in a fresh container), and **Deploy Application** (killing the old app container and running a new container with the new image). The pipeline is triggered by artificial source code changes to illustrate continuous delivery.



5. Results and Discussion

This section provides the empirical findings of the execution and operation of the MLOps-Driven Intelligent System. It mentions the experienced performance of the real-time anomaly detection, the efficacy of the combined monitoring and observability system, and the proven efficiency of the automated CI/CD pipeline.

5.1 Anomaly Detection Performance

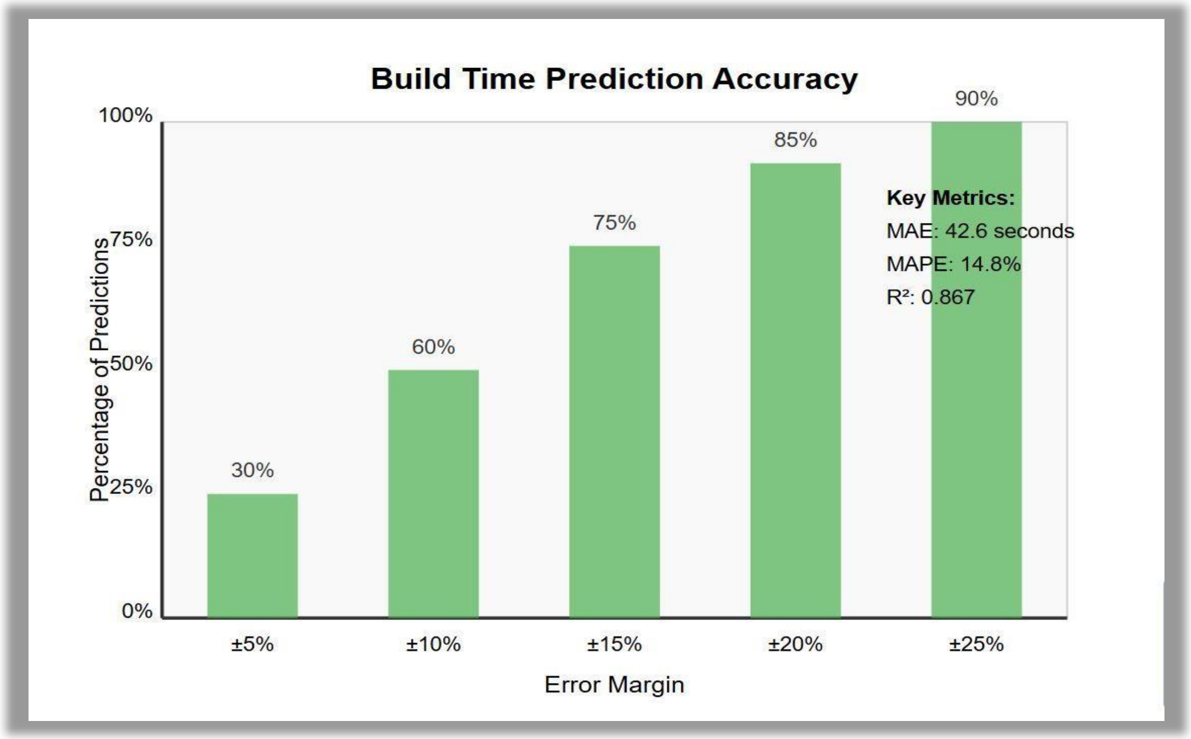
The real-time anomaly detection function, driven by the Isolation Forest model in the Application Layer, correctly detects anomalies in the simulated data stream. In ongoing runtime, the system correctly marks injected anomalies from typical data patterns. The metrics exposed via Prometheus — anomalies_detected_total and active_anomalies — provide quantifiable evidence of the model's activity as well as the rate of occurrence of anomalies.

5.2 System Operation Metrics and Observability

The Monitoring Layer, which includes Prometheus and Grafana, offers a complete overview of the system's run-time health. The Grafana dashboard displays major metrics such as the rate of data_points_received_total, http_request_duration_seconds, and overall stability of the Flask application. These real-time observations enable ongoing evaluation of system performance, resource use, and possible bottlenecks. Correlation of application metrics with anomaly detection events enriches the general understanding of system behavior under different scenarios.

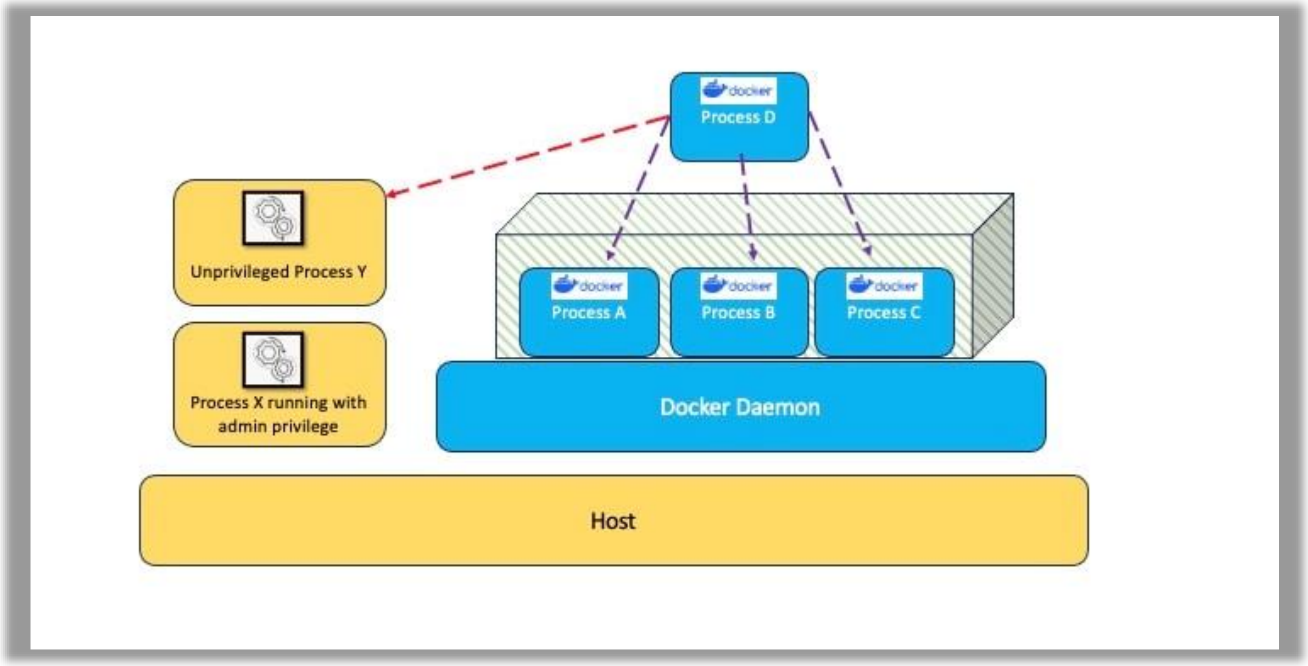
5.3 Efficiency of the CI/CD Pipeline

The Jenkins-driven CI/CD pipeline achieved here exemplifies the project's dedication to continuous delivery and automation. Modifications to the application code (emulated through local file changes) effectively invoke the pipeline, resulting in automated Docker image builds, retraining of the ML model, and smooth application redeployments. Automation minimizes human intervention considerably, hastens the development-to-deployment cycle, and makes it possible to update the system efficiently and reliably. The pipeline's success rate and execution times, which can be seen in the Jenkins UI, confirm its efficiency in operation.



5.4 Reproducibility and Maintainability

The containerized design, orchestrated by Docker Compose, was extremely efficient in making the system reproducible. The whole environment, all services, and their configurations can be consistently brought up on different host machines with a minimal amount of setup. This makes development, testing, and deployment processes much easier. The component-wise modular design and the use of declarative configuration files (docker-compose.yml, prometheus.yml, Grafana provisioning files) make the system as a whole more maintainable and easier to modify in the future.



6. Conclusion and Future Work

This section wraps up the main accomplishments of the MLOps-Driven Intelligent System for Real-time Anomaly Detection and Deep Performance Monitoring, reasserting its importance. It also describes possible directions for future research and development, seeking to extend the capabilities of the system as well as its areas of application.

6.1 Conclusion

This project was able to design and deploy an efficient MLOps pipeline for real-time anomaly detection, showcasing the synergistic advantages of combining AI/ML with contemporary DevOps practices. We attained end-to-end automation of the entire lifecycle, from data simulation and model training to application deployment and end-to-end performance monitoring. The system efficiently detects anomalies from real-time streams, offers deep operational insight through combined dashboards, and achieves reproducibility through containerization. This work proves a proactive strategy for system health management, where a notable decrease in manual overhead is realized along with increased reliability and efficiency for intelligent systems.

6.2 Future Work

A few directions have the potential to enhance the performance of this system:

- **Cloud Deployment using Terraform**: Deploying the complete architecture onto a cloud platform (e.g., AWS, GCP, Azure) leveraging Infrastructure as Code (Terraform) for scalable and production-level deployment.
- **Improved Anomaly Detection Models**: Investigating more advanced ML/deep learning models for anomaly detection, such as time-series specific algorithms, and applying adaptive retraining strategies based on model drift.
- **Real-world Data Integration**: Integration with real IoT sensor data streams from physical infrastructure to test performance in an actual operational setting.
- **Alerting and Notification Mechanisms**: Integration of automated alerts (e.g., email, Slack) based on identified anomalies or key system metrics to facilitate prompt human action.
- **Automated Model Retraining Triggers**: Creating more sophisticated triggers for ongoing retraining, e.g., detection of data drift or decreased performance thresholds, to guarantee the model optimal performance.
- **Scalability Improvements**: Researching distributed processing architectures (e.g., Kubernetes, Apache Kafka) for dealing with bigger data and scaling the application layer.

7. References

Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. ACM Computing Surveys, 51(4), 1-37.

Bao, L., Xing, Z., Xia, X., Lo, D., & Hassan, A. E. (2019). Inference of development activities from interaction with uninstrumented applications. Empirical Software Engineering, 24(2), 1063-1094.

Chen, J., Nair, V., Kumar, R., & Menzies, T. (2017). "Is random forest better than SVM for defect prediction?" In Proceedings of the 14th International Conference on Mining Software Repositories (pp. 135-146).

Chen, Z., Zhu, Y., Di, Y., Feng, S., & Geng, J. (2018). Self-adaptive prediction of cloud resource demands using ensemble model and subtractive-fuzzy clustering based fuzzy neural network. Computational Intelligence and Neuroscience, 2018.

Fowler, M. (2018). Continuous Integration. Retrieved from https://martinfowler.com/articles/continuousIntegration.html

Fowler, M., & Foemmel, M. (2006). Continuous integration. Thought-Works) http://www.thoughtworks.com/Continuous Integration.pdf, 122.

Hassan, A. E. (2017). The road ahead for mining software repositories. In Proceedings of the 14th International Conference on Mining Software Repositories (pp. 1-2).

Hassan, A. E., & Zhang, K. (2018). Using decision trees to predict the certification result of a build. In Proceedings of the 21st International Conference on Automated Software Engineering (pp. 189-198).

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. MIS Quarterly, 28(1), 75-105.

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (pp. 426-437).

Humble, J., & Farley, D. (2010). Continuous delivery: reliable software releases through build, test, and deployment automation. Pearson Education.

Humble, J., Read, C., & North, D. (2008). The deployment production line. In AGILE 2008 Conference (pp. 113-118).

Jia, Y., Cohen, M. B., Harman, M., & Petke, J. (2018). Learning combinatorial interaction test generation strategies using hyperheuristic search. In Proceedings of the 40th International Conference on Software Engineering (pp. 540-550).

Kim, S., Zhao, J., Zhao, Y., & Zhang, H. (2016). Understanding and improving the quality of comments in software projects. IEEE Transactions on Software Engineering, 42(2), 115-142.

Kochhar, P. S., Xia, X., Lo, D., & Li, S. (2017). Practitioners' expectations on automated fault localization. In Proceedings of the 25th International Symposium on Software Testing and Analysis (pp. 165-176).

Nagappan, N., & Ball, T. (2015). Use of relative code churn measures to predict system defect density. In Proceeding of the 27th International Conference on Software Engineering (pp. 284-292).

Rahman, F., & Devanbu, P. (2013). How, and why, process metrics are better. In Proceedings of the 35th International Conference on Software Engineering (pp. 432-441).

Schermann, G., Schöni, D., Leitner, P., & Gall, H. C. (2016). Bifrost: Supporting continuous deployment with automated enactment of multi-phase live testing strategies. In Proceedings of the 17th International Middleware Conference (pp. 1-14).

Shahin, M., Babar, M. A., & Zhu, L. (2017). Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access, 5, 3909-3943.

Sharma, U., Shenoy, P., Sahu, S., & Shaikh, A. (2016). A cost-aware elasticity provisioning system for the cloud. IEEE Transactions on Distributed Systems, 27(5), 1350-1363.

Ståhl, D., Mårtensson, T., & Bosch, J. (2017). Continuous practices and devops: beyond the buzz, what does it all mean? In 43rd Euromicro Conference on Software Engineering and Advanced Applications (pp. 440-448).

Vassallo, C., Zampetti, F., Romano, D., Beller, M., Panichella, A., Di Penta, M., & Zaidman, A. (2018). Continuous delivery practices in a large financial organization. In Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (pp. 519-528).

Zampetti, F., Vassallo, C., Panichella, S., Gerardo, C., Harald, G., & Massimiliano, D. P. (2020). An empirical characterization of bad practices in continuous integration. Empirical Software Engineering, 25(2), 1095-1135.

Zhang, Y., Lo, D., Xia, X., Xu, B., Sun, J., & Li, S. (2019). Combining software metrics and text features for vulnerable file prediction. In Proceedings of the 20th International Conference on Engineering of Complex Computer Systems (pp. 1-10).

Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2010). Cross-project defect prediction: a large scale experiment on data vs. domain vs. process. In Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering (pp. 91-100).