TITANICOPS:
ML-DRIVEN ENGINEERING INSIGHTS

MAJOR PROJECT REPORT

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENT FOR THE

AWARD OF THE DEGREE OF

BACHELOR OF TECHNOLOGY

(Computer Science & Engineering)

Submitted By:

Chandan Goyal (2203580)

Kajal Yadav (2203585)

Submitted To:

Er. Palak Sood

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
GURU NANAK DEV ENGINEERING COLLEGE,
LUDHIANA 141006

May, 2025

ABSTRACT

In the age of intelligent automation and continuous delivery, TitanicOps: ML-Driven Engineering Insights offers a scalable MLOps solution that integrates machine learning with contemporary DevOps. Leveraging the Titanic Spaceship dataset with attributes such as cryosleep status, spending habits, and cabin proximity, the system constructs high-accuracy predictive models through sophisticated feature engineering, preprocessing, and hyper parameter optimization. It utilizes ensemble algorithms like Random Forest, XGBoost, and Gradient Boosting, optimized for performance and deployed via automated CI/CD pipelines.

One of the main strengths is the combined application of DevOps tools—Jenkins manages CI/CD, Docker offers stable containers, and Kubernetes provides scalability and fault tolerance. Real-time monitoring employs Prometheus and Grafana, and Slack provides immediate feedback regarding deployments and system health. An easy-to-use Streamlit web interface allows technical and nontechnical users to provide data and get predictions with confidence scores and visual insights.

Hosted on AWS for cloud flexibility and resilience, TitanicOps illustrates modularity, observability, and security, with role-based access, vulnerability scanning (Trivy, SonarQube), and versioned model management. In bridging the divide between data science and production, TitanicOps provides a repeatable, transparent, and enterprise-ready template for future AI-enabled engineering solutions.

ACKNOWLEDGEMENT

We are highly grateful to Dr. Sehijpal Singh, Principal, Guru Nanak Dev Engineering College (GNDEC), Ludhiana, for providing this opportunity to carry out the major project work at TitanicOps .

The constant guidance and encouragement received from Dr. Kiran Jyoti H.O.D. CSE Department, GNDEC Ludhiana has been of great help in carrying out the project work and is acknowledged with reverential thanks. We would like to express a deep sense of gratitude and thanks profusely to Er. Palak Sood, without her wise counsel and able guidance, it would have been impossible to complete the project in this manner. We express gratitude to other faculty members of the computer science and engineering department of GNDEC for their intellectual support throughout the course of this work.

Finally, We are indebted to all whosoever have contributed in report work.

Chandan Goyal

Kajal Yadav

LIST OF FIGURES

TABLE OF CONTENTS

Chapter 1: Introduction

1.1 Introduction to Project

With the tremendous rise of technology today, the harmonious merge of machine learning (ML) and DevOps has turned into a building block for creating effective, scalable, and dependable systems. This project, "TitanicOps: MLDriven Engineering Insights", is about improving predictive analytics by integrating sophisticated ML models with a CI/CD pipeline automated and real time monitoring features.

The project utilizes the Titanic Spaceship dataset, which holds intricate passenger information like cryosleep status, spending habits, and cabin proximity, to make interdimensional travel outcome predictions. Utilizing feature engineering, model optimization, and hyperparameter tuning, the project seeks to enhance prediction accuracy by leaps and bounds.

For the sake of smooth deployment and operational effectiveness, the project incorporates a completely automated CI/CD pipeline with tools such as Jenkins, Docker, and SonarQube. This configuration provides automated testing, containerization, and deployment, minimizing manual interventions and shortening delivery cycles. Prometheus and Grafana are also included to monitor system performance in real time, while Slack notifications offer realtime updates on build and deployment activities.

In addition to this, Streamlit is also used to design a user interface that supports exploration of real time data and real time generation of predictions. It is hosted on AWS for scaling and reliability across the cloud with Kubernetes being utilized for orchestration of containers that allows effective sharing of resources. The interface can also be simply interacted with the ML models, offering straightforward visualizations as well as easy to understand insights for decisions based on predictions.

Through the combination of predictive modeling and DevOps automation, this project demonstrates a scalable, adaptable, and strong solution to addressing future challenges in predictive analytics and automated deployments.

Aside from technical aspects, "TitanicOps: ML Driven Engineering Insights" highlights the value of continuous improvement and inter team collaboration. With the blending of machine learning pipelines and DevOps methods, the project creates a rapid iteration culture whereby fresh models are deployed effortlessly and can be viewed for performance in real time. Not only does this improve overall predictive analytics efficiency but also makes it possible for teams to readily adjust to changing business needs and data sources. Through the use of version control over models, autorollback mechanisms, and integrated testing frameworks, the project provides assurance that the system is kept robust, secure, and scalable to process multiple, highlevel workloads while it grows.

1.2 Project Category

This project belongs to the class of a research-oriented machine learning study. It was undertaken based on the Titanic Spaceship dataset released publicly on Kaggle. The dataset offered a basis to investigate data preprocessing, pattern detection, and supervised learning methods in an organized research environment.

The primary objective of this study is to explore the performance of different classification models in predicting output based on input features. A step-by-step methodology was followed including data cleaning, feature selection, model training, and evaluation to ensure meaningful results were obtained from the analysis.

Experimentation was a dominant factor in identifying which models worked best with various tuning parameters and performance metrics over the course of the process. Several iterations took

place to compare and understand the results using confusion matrices, accuracy values, and other performance metrics.

This project helps to advance the understanding of data-driven modeling by focusing on hypothesis-driven investigation and interpretation of model behavior. It is consistent with research practice by integrating theoretical learning with applied practice based on real-world data.

1.3 Problem Formulation

Machine learning applications have been known to grapple with making the leap from development to production environment because of disjointed tooling, non-standardization, and operational bottlenecks. TitanicOps project solves these pain points by defining a system that unifies predictive modeling with DevOps principles, making both the development and deployment processes seamless.

With organizations scaling up towards real time analytics and automation, conventional ML pipelines are challenged in a number of ways:

Manual Deployment: Few ML systems have automated build and deployment pipelines, so model updates are subject to human error and delays.

Inconsistent Environments: Without containerization and orchestration, the same model running in different environments leads to versioning and dependency problems.

Lack of Real Time Monitoring: Deployed models run without insight into their performance, causing undetected drifts, bottlenecks, or failures.

Non-Technical Inaccessibility: Most ML packages are designed to be interactive and, therefore, not created to be accessible by stakeholders lacking technical programming proficiency for practical deployment.

With diverse features of the Titanic Spaceship dataset, there's one more dimension added. Working out an exact model with the right prediction in the data becomes the need, supported by strong preprocessing, engineering, and automation schemes to not incur overfitting, leakage of information, or inefficiencies at deployment time.

The challenge, thus, is to create an intelligent, automated, and user facing ML platform that:

- Trains high quality models with cutting edge algorithms.
- Automates the ML lifecycle based on CI/CD practices.
- Sends real time alerts and monitoring on system health.
- Supports predictions through a clean web interface for all users.

Through the resolution of these challenges, TitanicOps not only designs a high performance ML system but also establishes a benchmark for scalable, production level machine learning operations.

1.4 Recognition of Need

The rapid growth of data and drive toward predictive automation uncovered the essential gap in present day machine learning adoption: the gap between data science development and being production ready. The vast majority of ML projects never leave the experimental stage because of operational friction, security risks, or system unreliability.

TitanicOps steps up to meet this contemporary need for smart, productionready ML systems. Various core needs have been recognized:

1. Requirement for End to End Automation

With CI/CD pipelines, manual testing, validation, and redeployment are not needed to update ML models. This impedes development speed, lowers reproducibility, and raises system exposure.

2. Scalable Infrastructure Need

Normal hosting strategies lack the scalability required for dynamic loads or parallel requests for inferences. The hosting environment should be containerized and orchestrated so it can scale dynamically and have high availability.

3. Gaps in Observability and Monitoring

After deployment, ML systems are black boxes. Real time metrics, alerting, and performance dashboards that support proactive problem solving are increasingly necessary.

4. Stakeholder Accessibility

ML predictions must be accessible through easy touse interfaces to enable broader adoption across roles—analysts, managers, operators—without the need for programming.

5. Security, Versioning, and Rollback Control

Model integrity and audibility are vital in production settings. Versioning, container scanning, and rollbacks are mandatory for minimizing risks in updates.

Identifying these limitations, TitanicOps introduces a comprehensive solution that combines data science, software engineering, and DevOps in a single integrated platform. It optimizes workflows, promotes collaboration, and maximizes the transparency and usability of ML

systems, creating a new standard for how smart applications need to be designed, deployed, and consumed.

1.5 Existing System

Prior to TitanicOps: ML Driven Engineering Insights, machine learning (ML) systems that were already in place had several limitations, especially with regards to how models moved from development to production.

These conventional systems were usually siloed, with little integration between model training, deployment, monitoring, and user interaction.

Manual and Error Prone Workflows

In traditional settings, model development for ML is done offline on Jupyter notebooks or local IDEs, with minimal automation for testing, packaging, and deployment. New model versions hence need manual approval, environment creation, and deployment, bringing about the possibility of inconsistencies, config drift, and long delivery times. In addition, important phases like model validation, performance benchmarking, and unit testing tend not to be automated, bringing about challenges around reproducibility and scalability.

Lack of Continuous Integration and Deployment

CI/CD—essential for agile development—is seldom implemented in ML pipelines in legacy systems. When CI/CD pipelines are present, they are usually optimized for software development initiatives and not specifically designed for the iterative process of machine learning, which includes retraining, hyper parameter fine tuning, and model versioning. Inadequate automated pipelines create delays, human errors, and inconsistencies between dev and prod environments.

Inadequate Monitoring and Feedback Mechanisms

Monitoring of deployed ML models is often an afterthought in traditional systems. Once models are deployed to production, model performance is not monitored in real time. Key metrics like prediction latency, accuracy drift, or resource utilization are not measured, leading to silent failure or diminished system performance that nobody realizes until users complain. These systems also do not have adequate alerting mechanisms, so proactive resolution becomes infeasible.

Lack of Scalable Infrastructure

Current systems do not take advantage of containerization or orchestration platforms. This prevents them from horizontally scaling during periods of high traffic or self healing in the event of component failure. Deployment environments are normally tightly bound to particular servers or environments, so it is hard to replicate or scale across cloud infrastructures. Deployment is neither portable nor fault tolerant.

Minimal User Interaction and Accessibility

Most machine learning systems do not offer interfaces for end users to interact with the models. Consequently, stakeholders like business users, analysts, or decision makers are not able to access predictions or model outputs without relying on engineering support.

This bottleneck diminishes the usability and effectiveness of the ML models in real world decision making applications.

TitanicOps is specifically architected as a solution to address these limitations. By combining machine learning with a robust DevOps setup consisting of Jenkins, Docker, Kubernetes, Prometheus, and Slack, TitanicOps provides an end to end automated and observable system. Its Streamlit interface exposes real time predictions in an easily accessible format for technical and

nontechnical users alike, and monitoring dashboards and alert systems provide active health monitoring and error prevention.

The project represents a shift from isolated, script based workflows to an automated, interactive, scalable, and robust ML platform that is production grade.

1.6 Objectives

The objectives of TitanicOps are:

1. To implement ML models and feature engineering to improve model accuracy.
2. To implement an automated CI/CD pipeline for faster and reliable software delivery.
3. To implement a Streamlit based GUI for real time data exploration and predictions.
4. To implement real time notifications and live monitoring for builds and deployments.

1.7 Proposed System

The TitanicOps initiative suggests a combined, end to end DevOps and machine learning system intended to operationalize and automate predictive analytics with the Titanic Spaceship dataset. The system streamlines conventional ML processes by integrating automated CI/CD pipelines, real time monitoring, container orchestration, and an easy touse interface into one holistic architecture.

The central functionality is centered around offering high performance model training, continuous deployment and integration, effortless scalability, and real time prediction services via a web interface.

The suggested system addresses the shortcomings of manual processes, un observability, and the disconnection between data scientists, developers, and users by integrating contemporary MLOps practices.

System Components and Functionality:

Data Preprocessing and Feature Engineering

Raw data attributes like CryoSleep status, age, expenditure, VIP status, and cabin information are processed before handling missing values, outliers, and categorical variables. Feature engineering encompasses encoding, scaling, and creation of new features for better predictive model performance. Recursive Feature Elimination (RFE) and tree based model feature importance drive ideal feature selection.

ML Model Training and Optimization

Ensemble based models like Random Forest, XGBoost, and Gradient Boosting are trained with cross validation methods. The pipeline for training the model involves hyper parameter tuning through grid search or random search, followed by performance metric evaluation on accuracy, F1score, precision, and recall. Final models are serialized for deployment with Pickle or joblib.

Automated CI/CD Pipeline

The system adopts a Jenkins based pipeline for automatic integration and deployment. With every push to the GitHub repository, the following processes are triggered:

- Code linting and unit testing (Python scripts, training code of the model, API services).
- Static code analysis with SonarQube.
- Containerization with Docker.
- Image scanning with Trivy and security policies with OWASP.

Containerization and Orchestration

Models and APIs are containerized via Docker to maintain consistent runtime environments. Deployment, scalability, and fault tolerance are handled by Kubernetes. Horizontal scaling

provides high availability under fluctuating traffic conditions, while pod health checks provide resiliency.

Streamlit based User Interface

The system consists of a web based frontend developed with Streamlit, offering users:

- Real time input of data (passenger information).
- Immediate prediction results with confidence scores.
- Visual analytics such as feature importance, model summary metrics.
- Admin functionality (status monitoring, log viewing) for system administrators.

Real time Alerting and Monitoring

With Prometheus and Grafana, the system tracks important metrics in realtime:

- Model inference latency.
- API uptime.
- Resource consumption (CPU, memory).
- Prediction volumes. Anomalies in performance trigger alerts that get dispatched via Slack or email.

Cloud Deployment

The whole architecture is hosted on AWS via EC2 for backend operations and S3 for log/artifact storage. AWS Lambda optionally provides support for lightweight tasks like batch preprocessing. Cloud hosting enables elasticity, fault tolerance, and geographical scalability.

Overall System Flow:

- The user interacts with the Streamlit UI to provide passenger data.

- The UI talks to the deployed model API (Dockerized and served through Kubernetes).
- The system returns the predicted result with visual confidence measures.
- Admins may monitor performance metrics in Grafana dashboards and react to realtime Slack notifications.
- Jenkins guarantees automatic deployment of any new model version upon test pass.

This system design provides a very automated, transparent, and scalable ML pipeline with diminished time to deploy, improved model resiliency, and guaranteed system observability—offering a template for future ML DevOps hybrid systems.

1.8 Unique features of the Proposed System

The TitanicOps platform has a variety of unique aspects that distinguish it from traditional ML deployments or standalone DevOps pipelines. It is a single, cohesive solution that solves model development, deployment, monitoring, and user access—hence closing the operational divide between data science and production engineering.

The following are the main unique aspects that characterize TitanicOps:

1. End to End Automation through CI/CD

TitanicOps uses a strong and fully automated CI/CD pipeline powered by Jenkins and GitHub Actions. Each change to the codebase or model initiates a sequence of:

- Unit and integration tests.
- Code quality checking using SonarQube.
- Docker image build and vulnerability scanning (e.g., with Trivy).
- Smooth deployments on a Kubernetes cluster.

This minimizes human effort, avoids configuration drift, and provides high system reliability.

2. Model Containerization and Orchestration

The system packages machine learning models and APIs into Docker containers, which can be deployed plat formindependently. The containers are orchestrated by Kubernetes, which provides:

- Horizontal auto scaling during high load.
- Load balancing and high availability.
- Self healing in the event of container failure.

This architecture is agile, fault tolerant, and resource efficient.

3. Streamlit Based Interactive Interface

TitanicOps has a friendly interface based on Streamlit, where users can:

- Enter passenger information (e.g., age, VIP status, cryosleep, spending).
- Make real time predictions.
- Inspect feature importances, prediction confidence scores, and model summaries.

As opposed to usual ML APIs or notebooks, this no code UI makes AI more accessible to nontechnical stakeholders.

4. Real Time Monitoring and Alerting

With Prometheus and Grafana integrated, the system actively monitors:

- Inference latency and throughput.
- Resource utilization (CPU, memory).
- Model performance metrics (accuracy, prediction volumes).

Alarm worthy events like deployment failure, resource spikes, or high latency alerts are triggered using Slack Webhooks, allowing for real time problem monitoring and fast mitigation.

5. MultiLayer Security and Code Quality Controls

Multiple layers of security are enforced:

- Static code scanning with SonarQube.
- Image scanning for vulnerabilities.
- Authentication in the UI and API layers done securely.
- Version control with rollback support for model changes.

This guarantees all code and deployments are secure, performant, and compliant.

6. Cloud Native and Scalable Deployment

The system is optimized for AWS cloud deployment on EC2 for compute and S3 for storage. It also has optional AWS Lambda for background tasks or lightweight inference. Cloud deployment guarantees:

- Elastic scaling.
- Geographical redundancy.
- High availability across zones.

7. Performance Optimized ML Pipeline

Models such as XGBoost, Random Forest, and Gradient Boosting are optimized using hyper parameter search strategies to maximize predictive accuracy. Inference latency is optimized through:

- Model serialization techniques.

- Efficient data preprocessing pipelines.
- Optional use of ONNX or Tensor Flow Lite for fast inference.

8. Role Based Interface Design

TitanicOps supports two main user roles:

- Standard Users: Are able to browse datasets and make predictions.
- Administrators: Access monitoring dashboards, deployment logs, system health metrics, and CI/CD event triggers.

This segmentation guarantees both operational control and functionality without confusing the end user.

9. Visual Analytics and Explainability

To encourage trust and explain ability of model results, TitanicOps has:

- Dynamic visualizations (e.g., SHAP like interpretability, bar plots).
- Feature importance ranking.
- Confidence thresholds shown with predictions.

These features facilitate transparency and enable informed decision making.

Chapter 2: Requirement Analysis and System Specification

2.1 Feasibility Study

TitanicOps: MLDriven Engineering Insights performs a feasibility study of the system in terms of its viability in three domains: technical, operational, and economic. The analysis is conducted to ensure the system is not only innovative in conception but is also feasible, scalable, and sustainable in practical use.

Technical Feasibility

TitanicOps boasts a solid technical base that includes established tools and frameworks in machine learning, DevOps, and monitoring ecosystems.

1. Machine Learning Techniques and Models

The project employs strong ensemble models such as Random Forest, XGBoost, and Gradient Boosting—selected for accuracy, stability, and capacity to capture nonlinear patterns in the Titanic Spaceship dataset.

The models are trained on engineered features extracted from variables such as cryosleep status, VIP flag, cabin distance, and individual expenditures.

Advanced feature engineering methods like normalization, encoding, and feature importance analysis boost model performance and enhance predictive precision.

2. CI/CD Pipeline Integration

An end to end CI/CD pipeline is built based on Jenkins, which is utilized to automate test, build, and deploy phases.

- Docker provides uniform environments in development and production.

- SonarQube performs static code analysis on code quality and security loopholes.
- GitHub Actions can be used as an alternative for event driven workflow automation.

3. Monitoring and Automation Tools

- Prometheus and Grafana expose live monitoring dashboards, reporting metrics like CPU usage, memory usage, API response times, and inference latency.
- Slack notifications are configured through Webhooks to notify the development team if there are any failed builds, performance degradation, or deployment errors—enabling instant intervention and resolution.

4. Cloud Readiness and Scalability

The system can be deployed to the cloud on AWS EC2 instances for hosting and S3 for log and model artifact storage. Container orchestration is handled by Kubernetes, providing high availability, fault tolerance, and scalability according to load requirements.

Operational Feasibility

TitanicOps will be user friendly, allowing smooth interaction between system elements and end users, irrespective of technical expertise.

1. Streamlit Based Interface for Accessibility

The project consists of a Streamlit web app that makes it easy for users to engage with ML models using basic form inputs. This eliminates the barrier of needing to know how to code for users and supports real time predictions depending on user input.

Users are able to see outputs, such as survival chance and model explanations, in a straightforward, visual format.

2. Real Time Monitoring and Proactive Feedback

System health and resource consumption are constantly tracked through Grafana dashboards. Upon detecting anomalies like excessive latency or broken builds, realtime Slack notifications alert concerned teams for rapid diagnosis and fixing.

This provides maximum system reliability and minimizes downtime in production environments.

3. Flexibility and Ongoing Improvement

The system is extremely modular and accommodates ongoing integration of new models or feature releases without interrupting the current production process.

Automated processes allow for consistent deployment, rollback, and retraining as needed. This flexibility allows TitanicOps to adapt with new patterns of data, scaling user demand and business needs.

2.2 Software Requirement Specification (SRS) Document

The Software Requirement Specification (SRS) document for TitanicOps: MLEnabled Engineering Insights provides the minimum functional and nonfunctional requirements to design and deploy the MLenabled DevOps platform. The document is used to make all the stakeholders aware of the expectations from the system so that it is developed and deployed in a reliable, scalable, and user friendly manner that marries predictive analytics with CI/CD automation and monitoring.

Data Requirements

- **User Data:** Contains credentials, access roles (administrator or standard user), and session logs.

- **Input Data:** Passenger attributes like age, gender, cryosleep status, travel class, deck/cabin data, and spending history.
- **Prediction Results:** Model inference results together with respective confidence scores and timestamps.
- **Model Metadata:** Details of model type, accuracy scores, version, and last trained timestamp.
- **Pipeline Logs:** CI/CD process logs like build/deploy status, error messages, and test results.
- **Monitoring Metrics**: System health metrics in realtime such as CPU/RAM usage, inference latency, and request volumes.

Functional Requirements

Data Input Interface: Ability to input passenger data through a GUI implemented using Streamlit.

- **ML Prediction Engine:** System should produce predictions based on trained models (Random Forest, XGBoost, etc.).
- **Model Management:** System should allow training, evaluation, and versioning of models with Pickle/MLflow.
- **CI/CD Pipeline:** The system should be able to test, validate, and deploy code and models automatically using Jenkins, GitHub Actions, and Docker.
- **Containerization and Orchestration:** The system should be capable of containerized deployment (Docker) and orchestration (Kubernetes).
- **RealTime Monitoring:** Prometheus/Grafana metrics should be displayed in real time, and alarms should be notified through Slack.
- **Logging:** Logs should be collected centrally using the ELK stack (Elasticsearch, Logstash, Kibana).

Performance Requirements

- **System Response:** Model predictions should be returned in 3 seconds.
- **CI/CD Deployment:** Builds and deployments within 10 minutes.
- **Concurrency:** The platform should be able to handle a minimum of 100 concurrent users without any dip in performance.
- **Latency Thresholds:** Inference should be under 2 seconds; alerts should be raised if latency goes beyond thresholds.

Dependability Requirements

- **Uptime:** System should have 99.9% uptime with autorecovery in the event of failure.
- **Error Handling:** Error messages and retry mechanisms should be provided clearly in case of transient faults by the platform.
- **Redundancy:** Failover containers should be deployed in production deployments for high availability.

Maintainability Requirements

- **Modular Design:** Codebase should adhere to modular design to enable independent testing and debugging simplicity.
- **Documentation:** Doc strings, usage guides, and architecture diagrams should be provided in each module.

Security Requirements

- **Authentication:** Secure user login should be done with hashed passwords and optional 2FA.

- **Authorization:** Role based access should limit functionalities as needed for users vs. admins.
- **Encryption:** All data in transit should be encrypted using HTTPS and sensitive data should be encrypted at rest.
- **Vulnerability Scanning:** SonarQube and Trivy tools should scan for code and container vulnerabilities on a regular basis.

Look and Feel Requirements

- **User Interface:** Streamlit UI should be responsive, clean, and accessible (WCAG compliant).
- **Feedback:** Instant feedback should be provided for inputs and predictions submitted.
- **Visualization:** Feature importance graphs, performance metrics, and live monitoring dashboards should be included.

Software Requirements

- **Languages/Frameworks:** Python 3.8+, Scikitlearn, XGBoost, Streamlit, Docker, Jenkins, GitHub Actions.
- **Monitoring Tools:** Prometheus, Grafana, Slack Webhooks.
- **Cloud Stack:** AWS EC2/S3/RDS, Kubernetes, and optionally AWS Lambda for serverless functions.
- **Database:** PostgreSQL for structured model and user metadata.

Hardware Requirements

- **Development:** i5 processor minimum, 16 GB RAM, 256 GB SSD. NVIDIA GPU optional for model training.

- **Deployment:** t2.large EC2 or similar with 16 GB RAM and scalable storage (EBS/S3). GPU (e.g., Tesla T4) ideal for deep models.

2.3 SDLC Model to be used

The Agile Software Development Life Cycle model has been chosen for developing TitanicOps: MLDriven Engineering Insights. The Agile model is most appropriate for dynamic, iterative projects that require constant updates, testing, and feedback, particularly with machine learning going handing hand with DevOps practices in an ever changing technological landscape.

Why Agile SDLC?

The TitanicOps project consists of several integrated features like machine learning model building, CI/CD automation, real time monitoring, and containerized deployment. These features need frequent testing, performance analysis, and adjustments according to system behavior and end user feedback. Agile supports:

- **Iterative Development:** Features are implemented in limited, digestible sprints, enabling the team to concentrate on iterative improvement and incremental addition of functionality.
- **Rapid Feedback Integration:** Every sprint, stakeholders (developers, testers, users) offer feedback that is incorporated right away into the following cycle so the system can grow to serve real needs.
- **Flexibility:** Agile adapts changes in requirements, tools, or ML model strategies without impacting the entire development cycle.
- **Risk Mitigation:** Through deploying tested and proven features in phases, Agile identifies and solves integration issues or performance bottlenecks early.

Agile Phases Applied in TitanicOps

1. Requirement Gathering and Analysis

Project scope, technical stack, and infrastructure requirements were initially defined. Requirements from ML and DevOps sides were captured in user stories and task boards.

2. Planning and Sprint Design

The project was broken down into several development sprints. Each sprint concentrated on main elements such as:

Sprint 1: Streamlit GUI configuration and early data preprocessing

Sprint 2: ML model training and evaluation pipeline

Sprint 3: Automation of CI/CD pipeline using Jenkins and Docker

Sprint 4: Deployment of Kubernetes and monitoring using Prometheus/Grafana

3. Design

Modular design was used to isolate ML, UI, CI/CD, and monitoring logic. Data Flow Diagrams (DFDs) and block diagrams were created to direct development and integration.

4. Development

Code was written using tools such as GitHub, Docker, and Jenkins and pushed through a version controlled pipeline. Containerized environments provided uniform performance at development and production levels.

5. Testing

Each sprint had specific time allocated for unit, integration, and system testing. Automated tests were part of the CI pipeline to check model accuracy, API response time, container stability, and deployment readiness.

6. Deployment

After post testing, successful builds were automatically pushed to Kubernetescontrolled cloud instances (AWS EC2). Canary deployment was utilized later on for rolling out safely.

7. Monitoring and Feedback

Real time monitoring software such as Prometheus and Grafana were set up. User feedback (both functional and performance oriented) was gathered and used to create the next sprint backlog.

Agile Tools and Technologies Used

- **Project Management:** Trello for task tracking & sprint planning
- **Version Control:** Git and GitHub
- **CI/CD Pipeline:** Jenkins, GitHub Actions
- **Communication & Notification:** Slack Webhooks for realtime updates
- **Monitoring:** Prometheus & Grafana dashboards

Benefits of Using Agile in TitanicOps

- Less development and deployment cycles
- Regular testing and validation maintained system stability
- High flexibility to performance bottlenecks or ML model updates
- Consistent delivery of new features and enhancements

Chapter 3: Software Design

3.1 Design Approach

The TitanicOps: MLDriven Engineering Insights software design is centered on creating a modular, scalable, and fault tolerant architecture that combines machine learning predictions with DevOps automation and real time monitoring. The system is designed to enable smooth transitions from development to production environments using automated CI/CD pipelines, containerization, and cloud native deployment strategies.

This part introduces the design blueprint which governs system implementation, comprising interaction flows, module definitions, and architectural decisions which favor maintainability, scalability, and transparency of operations. The architecture further includes best practices for ML model lifecycle management as well as infrastructure observability.

The design process was informed by the project's fundamental objectives:

Decoupling of Components: Isolating the ML model logic, UI interface, CI/CD automation, and monitoring modules for improved maintainability and testability.

Containerization for Consistency: Employing Docker containers for maintaining consistency across development, staging, and production environments.

Orchestration for High Availability: Utilizing Kubernetes for deployment, autoscaling, and load balancing to maintain high availability under changing loads.

Real Time Monitoring and Feedback: Incorporating Prometheus and Grafana to monitor key performance metrics and system health, supplemented by Slack for real time notifications.

Interactive User Interface: Streamlit is used to provide an interactive web based interface for users to engage with the ML model, observe predictions, and see metrics visualized.

This chapter outlines the design process applied to every significant component—varying from preprocessing of data, ML modeling, and automation of pipelines to system deployment and performance tracking—to ensure all aspects of TitanicOps operate as one integrated, dependable system.

3.2 Detail Design

The TitanicOps: MLDriven Engineering Insights system is designed using a modular and decoupled design approach, following the micro service based deployment, reproducibility, and fault tolerant scalability principles. The in depth design involves decomposing the system into independently functional modules—each addressing a key element of the MLOps lifecycle—without compromising on interoperability through standardized interfaces and containerized runtime.

3.2.1 Architectural Overview

The framework adheres to a layered architectural paradigm, and there are clearcut layers dedicated to data preprocessing, machine learning business logic, CI/CD pipeline orchestration, user facing components, and monitoring. All these layers are individually horizontally scalable and deployable with Kubernetes to allow for high cohesion and minimal coupling.

Data Layer: Responsible for ingesting, validating, cleaning, and transforming the Titanic Spaceship dataset. Pandas and Scikitlearn transformers along with persistent schema enforcement are used to build data pipelines.

Modeling Layer: Applies ensemble learning models (Random Forest, XGBoost, Gradient Boosting) with hyper parameter tuning through Grid Search CV/Randomized Search CV. Models are saved to `joblib` and versioned in a model registry (e.g., MLflow).

Service Layer: Consists of RESTful APIs (through Flask/FastAPI) exposing model inference as a micro service, wrapped in Docker containers and governed by Kubernetes Deployments.

CI/CD Layer: Managed by Jenkins and GitHub Actions, this layer provides continuous testing, security scanning (SonarQube, Trivy), container builds, and automated rollouts to production.

Interface Layer: Developed with Streamlit, offering real time, responsive UI that hides backend complexity from users while facilitating actionable insights through visualizations.

Monitoring Layer: Deploys Prometheus for metric gathering and Grafana for visualization, and Slack Webhooks for anomaly notification and deployment event notifications.

3.2.2 Component Interaction Flow

The system adopts a publisher subscriber interaction model facilitated by API gateways and Kubernetes service endpoints:

1. User Entry Point: Users provide feature inputs through the Streamlit web form.

2. API Invocation: Inputs are invoked as HTTP POST requests to the backend model service.

3. Inference Service: The serialized model executes the input and provides a prediction, such as class label and confidence score.

4. Monitoring Hooks: Each API call records latency, resource usage, and prediction metadata, emitted to Prometheus.

5. Slack Notification Hooks: For build failure or resource threshold violation, Slack alerts are sent.

6. Admin Dashboard Access: Users can access system metrics, model performance, and logs in real time via embedded Grafana dashboards or external URLs.

3.2.3 Design Principles and Patterns

The design follows some best practices:

Single Responsibility Principle: One clearly defined responsibility is encapsulated in each module (data preprocessing, model training, prediction, etc.).

Separation of Concerns: Logical separation of frontend (Streamlit), backend (ML models & APIs), and infrastructure (CI/CD, monitoring).

Containerization Pattern: Every functional unit is containerized with Docker and orchestrated through Kubernetes, allowing zero downtime deployments.

Infrastructure as Code (IaC): All deployment configurations, secrets, and resource definitions are handled through YAML manifests and Helm charts.

3.2.4 Scalability and Fault Tolerance

Auto Scaling: Kubernetes Horizontal Pod Auto scaler (HPA) scales compute capacity according to API request load and CPU utilization.

Failover Readiness: Redundant pods and rolling updates guarantee availability even in the event of node failure or image updates.

Logging and Recovery: Centralized logging through the ELK stack enables postmortem analysis and recovery scripting during partial system failure

3.3 User Interface Design

The User Interface (UI) of TitanicOps is planned to provide an interactive, intuitive, and responsive experience, allowing users to seamlessly interact with the machine learning pipeline without technical knowhow. Designed with Streamlit, the UI prioritizes simplicity, accessibility,

and functionality, offering real time visualizations and instant feedback depending on user inputs and ML model predictions.

Purpose

The core function of the TitanicOps UI is to act as the frontend interface for data entry, ML prediction output, and system monitoring. It makes interaction with intricate backend services easier by hiding them behind a simple and austere user interface.

Major Functionalities of the UI:

1. Passenger Data Entry Panel

Users can enter appropriate features like age, cabin, VIP, companions, spending categories (room service, shopping, food court, etc.).

Dynamic input validation guarantees data completeness and accuracy prior to submission.

2. Prediction Output Section

Renders live prediction output with:

- Outcome (e.g., Transported: Yes/No)
- Confidence Score
- Visual cues (such as colorcoded result boxes to improve UX)

3. Feature Importance Visualization

Displays bar charts or pie charts indicating which features contributed most to the model decision.

Assists users and stakeholders in understanding model behavior.

4. System Metrics Panel (Admin Only)

Renders API response time, resource utilization, and model latency (queried from Prometheus).

Visualized through Grafanalike embedded blocks or linkouts.

5. Notification/Status Bar

Shows notification of backend status such as:

- Successful deployment
- Build failures (if integrated with Jenkins/Slack)
- Model refresh or update status

Security & Access Control

- **Standard Users:** Can view only the data input form and prediction outputs.
- **Administrators:** Get access to sophisticated analytics, model monitoring dashboards, and deployment logs (through API hooks).

Role based visibility provides a clean UI experience with users seeing only applicable options.

3.4 Methodology

1. Raw Data (Spaceship Titanic Dataset)

The exercise starts with accessing the Spaceship Titanic dataset, a make-believe but structured dataset reflecting passenger data from an interstellar disaster. The dataset contains demographic information, consumption patterns, location in the cabin, travel intentions, and if a passenger was carried or not. These are used as inputs for constructing a classification model and reflect real-life scenarios well-suited for implementing machine learning approaches.

Fig 3.1 Methodology of project

2. Training Data

The data is separated into training and test sets, with the training set usually accounting for 70–80% of the data. This sub-set contains both features and labelled outcomes. It is used by models

to learn from past patterns—identifying which sets of variables (such as age or cabin) go with passengers being carried—allowing them to make educated predictions in the future.

3. Data Preprocessing

This phase guarantees the dataset is clean, complete, and machine-learning-friendly. Missing data are addressed using methods such as mean/mode imputation or predictive filling. Duplicates are dropped to avoid bias. Categorical features are encoded numerically using one-hot or label encoding. Scaling (e.g., Min Max or Standard Scaler) guarantees numerical homogeneity. Preprocessing provides a solid basis for training and improves model accuracy.

4. Feature Engineering

New information features are built or old ones are converted. For instance, extracting deck data from the cabin column, consolidating all the spend columns into a single "Total Spend" variable, or creating "Is Alone" from group size. Such engineered features usually unveil underlying patterns and significantly enhance model performance by making inputs more relevant.

5. Exploratory Data Analysis (EDA)

EDA discovers structure, outliers, and patterns through visualization and statistical methods. Histograms, heat maps, and pair plots expose distributions and relationships. Skewed variables, class imbalance, or outliers are detected and resolved. EDA informs feature selection choices, data transformations, and model choice, ensuring that modeling is data-driven.

6. Model Training and Hyperparameter Tuning

Different machine learning algorithms like Random Forest, Decision Trees, SVM, and Gradient Boosting are trained. Each one is run with a variety of hype rparameters—max depth, learning rate, etc.—to optimize performance. Tuning involves Grid Search or Random Search methods

on cross-validated data to identify best configurations, avoiding overfitting and increasing accuracy.

7. Model Evaluation

Every model is validated against several performance metrics:

- Accuracy – overall correctness
- Precision – accuracy of positive predictions
- Recall – finding all positives
- F1-Score – equality between precision and recall
- F1-Score – equality between precision and recall

8. Selected Model

The best-scoring model on the selected metrics is now finalized. It is exported through serialization (e.g., with `joblib` or `pickle`) and version-controlled to ensure reproducibility. This chosen model is now the main asset for downstream deployment and inference in production.

9. Testing Data

Reserved throughout the first split, the test dataset mimics unseen real data. It holds only the features (not the target label) and is an unbiased test on the final model's capacity to generalize and predict accurately outside of the training context.

10. Data Preprocessing on Testing Data

To avoid inconsistencies or data exposure, the same preprocessing and transformation processes used in training data are again applied to the test set. The same encoders, scalers, and feature builders are used. Such automation of the pipeline guarantees reliability and integrity.

11. Prediction Generation

The trained model is used to predict on the test data, generating predictions for every passenger—representing the probability of being transported. These results are returned as either probability scores or binary values (e.g., 0 = Not Transported, 1 = Transported), depending on use-case requirements.

12. Prediction Results

Predictions are stored into formatted files (CSV, JSON) for review, analysis, or submission. Results can also be shown on dashboards or APIs.

They are used as performance indicators and drive decisions or actions based on the model output.

13. Push Project Files to GitHub using Git Bash

With the assets, notebooks, and codebase ready, the entire project is committed and pushed to GitHub using Git Bash.

14. Jenkins Pulls the Code From GitHub

Jenkins is set up with a webhook or a scheduled job to fetch the latest code from GitHub. This triggers the CI/CD process. Jenkins pipelines re used to execute automated tests, build scripts.

15. Trivy Scans Files for Vulnerabilities

Prior to development, Trivy performs a full scan of source files to detect insecure packages, misconfigurations, and third-party library vulnerabilities.

This advance detection reduces risk and makes security a development concern.

16. SonarQube Analysis

The source code is statically analyzed using SonarQube to detect bugs, vulnerabilities, security hotspots, code smells, and maintainability. It provides quality ratings and coverage reports to ensure a clean, modular, and scalable code base.

17. Build Docker Image

The application, along with all dependencies, configurations, and environment variables, is containerized using a Dockerfile. The Docker image created is a self-contained entity that maintains uniform performance on any system or cloud provider.

18. Trivy Scans Docker Image

After the Docker image is created, it is again scanned by Trivy to detect any new vulnerabilities added during containerization.

This encompasses base image problems or application layer dependencies to ensure the image is secure end to end.

19. Push Docker Image to Docker Hub

The authenticated Docker image is pushed to Docker Hub, where it is available across environments. Tags are added (e.g., `latest`, `v1.0`) for versioning.

20. Deploy to Kubernetes

Through Kubernetes manifests or Helm charts, the image is deployed into a managed cluster. Kubernetes takes care of pod scheduling, auto scaling, rolling updates, and self-healing features—providing high availability and fault tolerance of the application.

21. Email Notification

CI/CD pipelines are set up to automatically send email notifications at critical points—build success/failure, test outcome, or deployment status. This keeps stakeholders updated and allows for quick action if anything goes wrong.

22. Deployed Website

After deployment, the application is made publicly available using a browser-based web interface. Users access it by providing passenger information and obtaining real-time predictions. The frontend can also display analytics, logs, and result summaries.

23. Monitoring Tools (Prometheus and Grafana)

Prometheus scrapes Kubernetes and application metrics (CPU, memory, latency, etc.). Grafana then presents these metrics on interactive dashboards. Anomaly detection is configured with alerts for operational visibility in real time and performance optimization.

Chapter 4: Implementation and Testing

4.1 Introduction to Languages, IDEs, Tools and Technologies Used

In order to effectively execute the TitanicOps: MLDriven Engineering Insights project, multiple programming languages, development tools, and opensource technologies had to be implement. These features were chosen because of their performance, stability, and compatibility with current DevOps and machine learning practices.

This section describes the entire technology stack and the role each component had in the development, deployment, and monitoring of the project.

1. Programming Languages

Python 3.8+

Python was the go-to programming language throughout the whole project life cycle, from data preprocessing and feature engineering to model training of machine learning models and backend workflow automation. Pandas and NumPy libraries were utilized for streamlined data manipulation and numerical computations, while Scikit-learn and XGBoost offered robust capabilities for classification model building and hyper parameter tuning. Sophisticated methods like pipeline chaining, hyper parameter tuning, and model evaluation metrics (precision, recall, F1-score) were realized through Python's ecosystem.

YAML (Yet Another Markup Language)

YAML files were used for declarative CI/CD pipeline configuration in Jenkins and GitHub Actions, as well as Kubernetes manifests for orchestration of deployment. These files specified workflows like job triggers, resource requests, replica settings, and health checks in a human-readable and easy-to-maintain format.

Bash

Bash scripts were critical in automating build, test, and deployment operations. Custom shell scripts were designed to govern pipeline execution logic, trigger Docker builds, manage environment variables, and orchestrate container lifecycles throughout various stages.

2. Integrated Development Environments (IDEs)

Visual Studio Code (VS Code)

VS Code was the primary development IDE, selected due to its wide plugin ecosystem and tight Git integration. Functionality such as Python completion, syntax colorization, version control, YAML editing, Dockerfile handling, and Streamlit plugins sped up development processes. The development environment was set up with linters (Pylint), formatters (Black), and support for virtual environments.

Jupyter Notebook

Used mostly in the exploratory data analysis (EDA) and feature engineering steps. Jupyter enabled interactive visualization, line-by-line debugging, and quick iteration over data manipulation. Inline plots with Matplotlib and Plotly facilitated visual hypothesis testing and exploratory modeling.

3. Frameworks and Libraries

Scikit-learn

Used to develop machine learning pipelines, encoding categorical variables, data normalization, and model training with decision trees, logistic regression, and ensembling techniques. Also used to analyze performance via cross-validation, confusion matrices, and classification reports.

XGBoost / Gradient Boosting

Used for high-accuracy classification with robust generalization. Hyper parameters like learning rate, tree depth, and regularization terms were tuned for optimal ROC-AUC and accuracy on validation sets.

NumPy and Pandas

Both these libraries had undertaken data ingestion, transformation, and aggregation tasks. Loaded data from CSV files efficiently, filled in missing values, created new features, and did matrix-based operations while preparing models.

Streamlit

An inofficious Python-based UI framework had been utilized for developing an intuitive web interface for the ML model. Users would be able to provide passenger attributes as input and obtain real-time prediction results together with visual feedback about confidence scores.

Matplotlib and Plotly

Used for data visualization in both development and production environments. Matplotlib was employed to create static plots (e.g., distributions of features), whereas Plotly provided interactive plots (e.g., feature importance, prediction histograms) to the Streamlit dashboard.

4. DevOps Tools

Jenkins

Enacted to automate CI/CD pipelines for auto-testing, building, and containerized deployment of the ML application. The jobs were executed on code commit with rapid feedback loops. Jenkinsfiles customized pipeline stages with composable Bash scripts.

GitHub Actions

Employed for small-footprint CI activities like linting, testing, and artifact caching. Workflow files controlled branch-specific rules, environment matrix tests, and reusable jobs for homogeneous developer feedback.

Docker

All three components—Python models, Streamlit apps, and monitoring agents—were containerized to ensure consistent behavior across environments. Docker Compose facilitated local multi-container development, while Dockerfiles stored build logic and dependency layers.

Kubernetes

Facilitated fault-tolerant operation, automated scaling, and failure-resilient deployment of the containerized application. Configurations involved readiness probes, horizontal pod autoscalers, and rolling updates to reduce downtime during upgrades.

SonarQube

Integrated for ongoing code quality guarantee. Static analysis checks were run in CI pipelines to identify code smells, enforce style rules, and track technical debt in Python and Bash scripts.

5. Monitoring and Notification Tools

Prometheus

Instrumented the backend APIs and model endpoints with custom and system metrics, including inference time, request counts, and memory usage.

Exporters scraped metrics which were periodically scraped by Prometheus servers.

Grafana

Offered real-time dashboards with alerting capabilities to graph performance metrics, monitor uptime, and debug anomalies. Integrated with Prometheus as a data source for rendering dynamic time-series graphs and health summaries.

6. Cloud and Infrastructure Services

AWS EC2

Housed the production application in containerized format. EC2 instances were deployed with Docker and Kubernetes (through Minikube or EKS), providing a scalable and flexible platform for deployment with network control and security groups.

AWS S3

Employed for durable and scalable storage of artifacts such as trained model binaries (`.pkl` or `.joblib`), snapshots of datasets, application logs, and user-uploaded content. Access policies were configured to make data accessible to applications and secure.

4.2 Algorithm Used

The forecasting foundation of the TitanicOps project relies on ensemble based machine learning models like Random Forest, XGBoost, and Gradient Boosting Classifier. They were selected based on their good performance for classification problems with complicated feature interactions and missing or imbalanced data. The pipeline includes systematic preprocessing, feature engineering, model training, evaluation, and prediction processes.

This part explains the overall algorithmic process Implement for training models and making predictions, presented in formal pseudocode to depict the step by step reasoning.

Pseudocode for Data Preprocessing and Model Training

Algorithm: TrainTitanicSurvivalModel

Input: Dataset Train and Test

Output: Trained ML Model M

1. BEGIN

2. Import dataset Train and Test to DataFrame

3. Drop the columns of no interest (e.g., PassengerId, Name, Ticket)

4. Missing value handling:

   For numeric columns: replace with median

   For categorical columns: replace with mode or "NaN"

5. Categorical encoding using:

OneHot Encoding (for low cardinality)

Label Encoding (for binary classes)

6. Normalize numerical features using StandardScaler

7. Split the data into training set and test set (e.g., 80/20)

8. Initialize model M using XGBoostClassifier or RandomForestClassifier

9. Perform hyper parameter tuning using GridSearchCV or RandomizedSearchCV

10. Fit model M on training data

11. Test model M using:

- Accuracy
- Precision
- Recall
- F1score

12. Save the trained model to disk using Pickle or Joblib

13. End

Pseudocode for RealTime Prediction through Streamlit Interface

Algorithm: PredictSurvival

Input: UserInput U from Streamlit UI

Output: Survival Prediction with Confidence Score

1. BEGIN

2. Load serialized model M from disk

3. Accept user input U from web form

4. Preprocess U to match model input format:

   Use same encoding and scaling transformations applied during training

5. Predict outcome P = M.predict(U)

6. Forecast probability score S = M.predict_proba(U)

7. Show output on UI:

   "Transported: YES/NO"

   Confidence Score: S

   Feature Impact Visualization

8. END

Model Selection Logic

If accuracy requirement is high and training time is acceptable:

Use XGBoostClassifier

Else if interpretability is preferred:

Use RandomForestClassifier

Else:

Use GradientBoostingClassifier

4.3 Testing Techniques

In the design of the TitanicOps MLDriven Engineering Insights project, a thorough testing plan was adopted to ensure the functionality, reliability, and performance of both DevOps and machine learning components. Because of the intricate nature of the system, which combines predictive modeling, a user interface, CI/CD pipelines, and real time monitoring, various software testing methods were utilized throughout the development cycle to guarantee robustness and accuracy at each phase.

Unit Testing

Unit tests were performed to ensure individual components in isolation. Data preprocessing functions, functions for missing values, and prediction functions were tested based on Python testing libraries like `pytest` and `unit test`. This degree of testing ensured that every function or method yielded the anticipated output under different conditions, like ensuring that imputed values were as expected against median or mode expectations, and ensuring that model outputs were in the right format and value range. Also, single API endpoints were tested to ensure that they responded appropriately with proper status codes and anticipated outputs.

Functional Testing

Functional testing was conducted to ensure the system satisfied end user specifications. This involved verifying that users could enter passenger information into the UI and get proper predictions along with confidence values. Administrators were also tested for visibility into performance dashboards and logs. The system's real time alerting feature—implemented via Slack Webhooks—was also tested to ensure timely notification on build failures or deployment.

Performance Testing

Performance testing was performed with the help of tools like Locust and JMeter to test under heavy load environments. The intention was to observe how the system would react under simultaneous prediction requests and during regular deployment occurrences. API response time, CPU utilization, and model inference time were tracked with the use of Prometheus and monitored on a dashboard through Grafana.

Security Testing

Security testing was necessary because there were user exposed vulnerabilities and cloud infrastructure. The web application and APIs were scanned with OWASP ZAP for standard

vulnerabilities such as crosssite scripting (XSS) and SQL injection. Container images were scanned using Trivy to identify security misconfigurations or vulnerable dependencies. SonarQube was also utilized to enforce secure coding guidelines and identify code level vulnerabilities.

Usability Testing

Usability testing specifically aimed at gauging the readability and accessibility of the user interface. Streamlit based GUI was tested with both technical and nontechnical users to validate that all aspects were intuitive in nature and system feedback, in the form of prediction results or error messages, was displayed to the user in a readable fashion. Responsiveness of the interface on desktops as well as mobile devices was also tested.

Regression Testing

Regression testing was consistently conducted, particularly when model updates or infrastructure upgrades were made. Automated test scripts guaranteed that previously functional features still performed as expected following codebase modifications. This was crucial in maintaining a stable and consistent user experience with new functionalities added or existing ones enhanced.

Integration Testing

Integration testing was instrumental in proving the interaction between subsystems. As a case, the interaction between the Streamlit interface and the machine learning pipeline was checked to provide real time data transition from user input to prediction. Jenkins pipelines were subjected to testing to prove that successful code commits initiated proper build and deploy actions. These integration tests proved that independently developed components performed well when deployed as part of an ensemble.

4.4 Test Cases

Test Case 1: Input correct passenger information in the Streamlit input form such as name, age, cryosleep status, and cabin information.

Expected Outcome: The system should process the input and generate a prediction with a high degree of accuracy.

Result: The model accepted the input, successfully processed the data, and showed a prediction without any delay.

Test Case 2: Fill in the form with all fields empty or with default placeholders.

Expected Outcome: The form should trigger validation errors and display messages informing which fields are required.

Result: The system properly identified all missing fields, and no prediction was created until valid information was provided.

Test Case 3: Input invalid values like alphabets in numeric fields (e.g., 'abc' in age field).

Expected Outcome: Input validation rules must flag the error and hinder form submission, also reporting back to the user that it's invalid input.

Result: The system did prevent the bogus input from taking effect and rendered a user readable error of invalid data.

Test Case 4: Login with admin privileges and proceed to monitoring dashboard.

Expected Outcome: Admin can view real time system status, deployment logs, and control of model management.

Result: Admin interface loaded with all additional features and system monitor data shown correctly through Grafana.

Test Case 5: Load dataset with missing values in primary fields like 'HomePlanet' and 'VIP'.

Expected Outcome: Preprocessing module must identify and impute missing values with default approaches (mean, mode, or set logic).

Result: The preprocessing pipeline managed missing values unproblematically, and no errors or NaNs were present in the output data.

Test Case 6: Transform categorical features like 'CryoSleep' and 'Destination' to numerical form.

Expected Outcome: The data must be appropriately encoded using one hot or label encoding to make it ready for training ML models.

Result: All categorical variables were converted appropriately, and the resulting dataset was ready for model ingestion.

Test Case 7: Perform normalization and standardization on features like age, room service, and shopping spend.

Expected Outcome: Scaled features will have a uniform range, enhancing model convergence and performance.

Result: The features were successfully scaled by using Min Max normalization, and no outliers or skewness was found.

Test Case 8: Train Random Forest model with preprocessed Titanic data.

Expected Outcome: The model should train successfully and output evaluation metrics such as accuracy, recall, and precision.

Result: Training was done within reasonable time, and the model produced high accuracy (above 80%) on validation set.

Test Case 9: Apply trained model to make prediction of survival outcome for new unseen data inputted through the UI.

Expected Outcome: The prediction should be made immediately with a confidence score or probability value.

Result: The prediction was correct and delivered with confidence percentage, validating correct integration of UI and backend.

Test Case 10: Push new code to GitHub to fire the CI/CD pipeline using Jenkins.

Expected Outcome: Jenkins must notice the new commit, execute tests, build Docker image, and start deployment.

Result: Pipeline triggered automatically; build successful, and the new version was deployed to Kubernetes without any hiccups.

Test Case 11: Deliberately introduce syntax error in code and push to repository.

Expected Outcome: Jenkins should pick up the issue during the build process, stop the process, and alert the developers.

Result: Build failed as expected, and error notification was successfully sent through Slack webhook.

Test Case 12: Validate containerization and Docker image building for the trained model.

Expected Outcome: Docker must containerize the application and dependencies into a Docker image without failing.

Result: Docker image was successfully built and pushed to DockerHub with proper version tags.

Test Case 13: Deploy the Docker image using Kubernetes orchestration on AWS.

Expected Outcome: The system should deploy updated pods with zero downtime.

Result: Kubernetes deployment was seamless, auto scaling was running, and the web app was always available.

Test Case 14: Observe system CPU and memory usage during normal and high load scenarios.

Expected Outcome: Prometheus and Grafana must show real time usage stats with alerts on unusual activity.

Result: Metrics were visualized effectively, and alerts were triggered when usage exceeded 80%.

Test Case 15: Conduct usability test of Streamlit UI with nontechnical users.

Expected Outcome: Interface must be intuitive to comprehend, navigate, and utilize for data entry and seeing predictions.

Result: Test users performed tasks independently, verifying UI accessibility and simplicity.

Chapter 5: Results and Discussions

5.1 User Interface Representation

The user interface for the TitanicOps system was developed with Streamlit, an effective Python library that allows developers to build interactive, real time web applications with less code. The UI is the core platform where users are able to engage with the machine learning model, enter passenger information, see prediction results, and receive visual insight into the performance of the model.

The interface is intuitive, responsive, and clean, such that technical as well as nontechnical users can use it with ease. Some of the most important features of the UI are input fields for passenger characteristics, prediction windows, confidence scores in real time, and graphical representations like feature importance plots.

Users are able to enter data fields like age, cryosleep status, deck position, service costs, and so on. After submission, the data is processed and sent to the trained ML model, and the output is immediately displayed as a predicted result ("Transported" or "Not Transported") with a confidence percentage. Visual components like pie charts and bar graphs are Implement to make results more interpretable.

This interface was also tested on various screen sizes and browsers to make it accessible as well as responsive.

5.1.1 Brief Description of Various Modules of the System

The TitanicOps system comprises multiple dependent modules that collectively facilitate a full stack machine learning deployment pipeline with DevOps capabilities. Following is a description of the key modules:

1. Data Input Interface Module:

This module is the frontend interface via which users may input passenger information for prediction. Built with Streamlit, it has form fields for attributes such as age, VIP, cabin location, and cryosleep. It also performs input validation and initiates prediction requests upon form submission.

2. Data Preprocessing and Feature Engineering Module:

Once the data is submitted, this backend module does necessary transformation and cleaning. It treats missing values, codes categorical variables, and scales numerical features. It also implements feature selection methods in order to retain the most effective variables for training and prediction models.

3. Machine Learning Model Module:

This is the system's backbone where machine learning models (Random Forest, XGBoost, Gradient Boosting) are trained, tested, and deployed. It comprises model selection, hyper parameter tuning, and metric evaluation.

4. CI/CD Automation Pipeline Module:

This module facilitates continuous integration and deployment with Jenkins, GitHub Actions, and Docker. It takes care of building, testing, and deploying. Each push in the repository automatically tests and deploys to production via Kubernetes.

5. Containerization and Orchestration Module:

The Dockerized application gets deployed with Kubernetes for high availability, load balancing, and scale out deployment. The system scales automatically based on user load or resource utilization.

6. Real Time Monitoring and Notification Module:

System performance and health metrics (CPU, latency, memory usage) are monitored by Prometheus and displayed in Grafana dashboards. Integration with Slack guarantees that developers receive real time notices for any deviations or failures in deployment or inference speed.

7. Admin and Deployment Dashboard (Optional for Admins):

A special interface available to administrators includes system logs, container health stats, model performance graphs, and access to deployment configurations. This module enhances system observability and control.

5.2 Snapshots of system with brief detail of each and discussion

AWS EC2 Console

The screenshot illustrates the AWS EC2 console showing a list of running instances. It contains information like the names of the instances, IDs, instance types, status checks, and availability zones. The instances are part of a CI/CD environment, with some prominent ones being jenkinsmaster for continuous integration and eksspacenode instances, probably used for the deployment of Kubernetes clusters. The environment seems to be hosted on the Asia Pacific (Singapore) region.

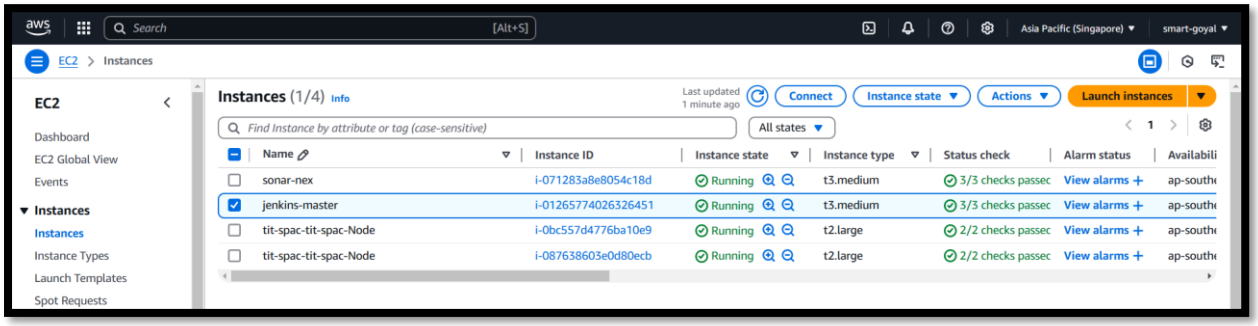

Fig 5.1 AWS EC2 Console

AWS Cloud Formation Stacks

The screenshot depicts the AWS CloudFormation console showing two stacks created successfully. The stacks are for Amazon Elastic Kubernetes Service (EKS), one for nodes and SSH access, and the other for the EKS cluster infrastructure. The "CREATE_COMPLETE" status shows that the stacks were deployed successfully.

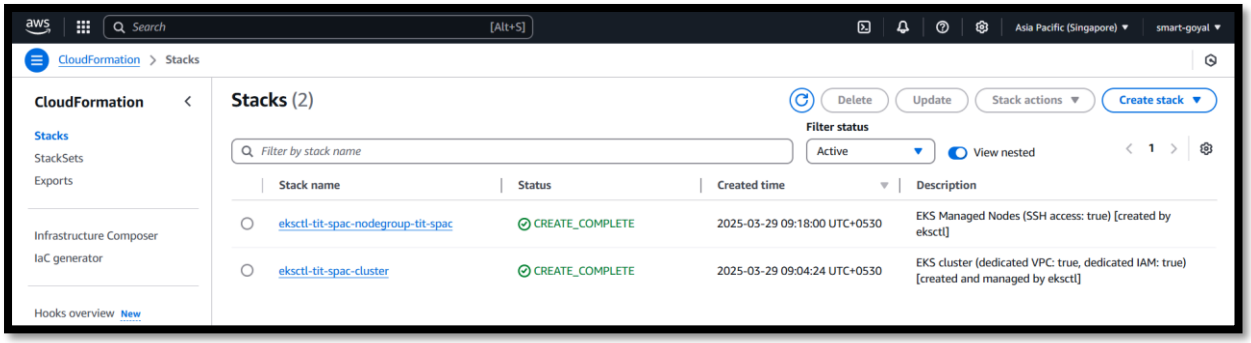

Fig 5.2 AWS Cloud Formation Stacks

Jenkins Dashboard

The dashboard presents pipeline data such as build status, names, and duration along with other measurements. It seems to be indicating at least one pipeline along with its latest status (presented through a green circle), the last success timestamp, and last failure timestamp. The interface offers functionalities for displaying build history, project dependencies, and handling Jenkins.

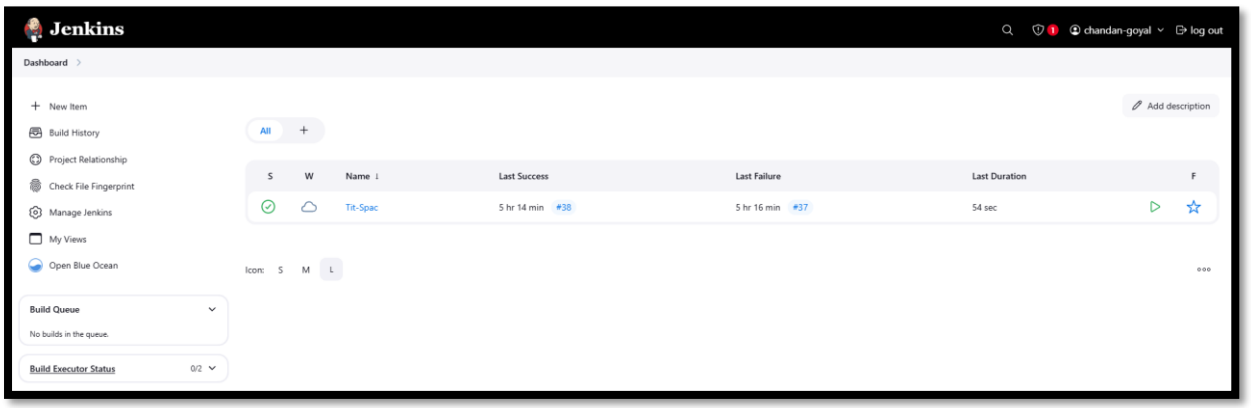

Fig 5.3 Jenkins Dashboard

Jenkins Pipeline

The Jenkins pipeline visualizer for "TitSpec" also comes with detailed build stage logs offering information on error or warning, making it simple to troubleshoot. The history of builds can also be checked quickly, helping team members assess the success or failure of the previous build for comparison..

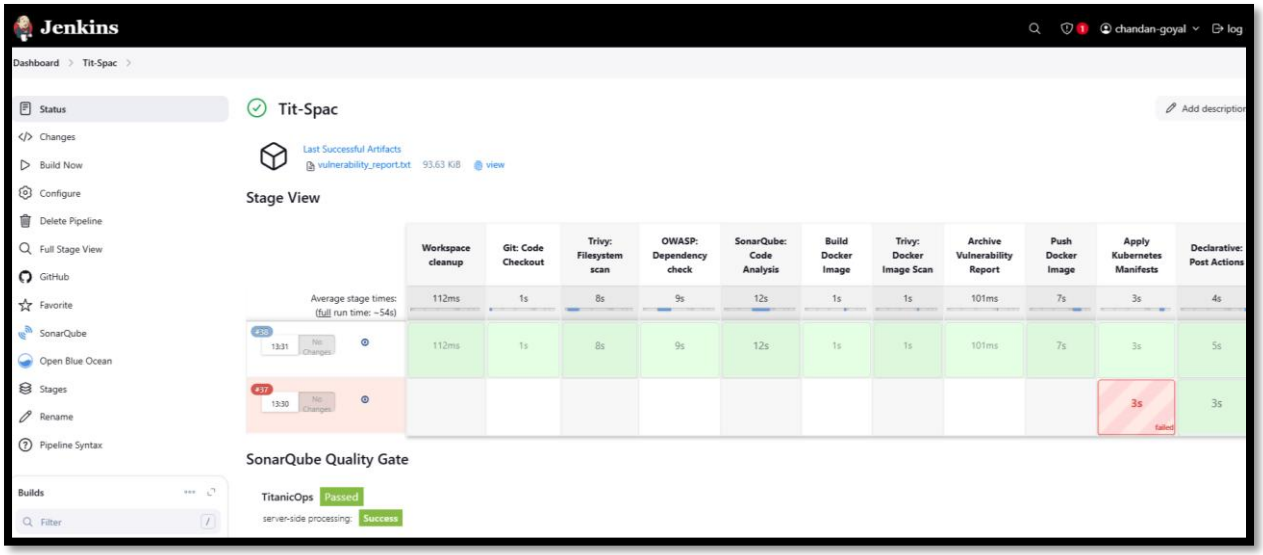

Fig 5.4 Jenkins Pipeline

Jenkins Pipeline Stages

The image depicts a Jenkins build pipeline for "TitSpec" with a series of linked stages represented in a linear workflow. The stages are circles with green checkmarks for successful completion, joined by lines representing the movement from beginning to end.

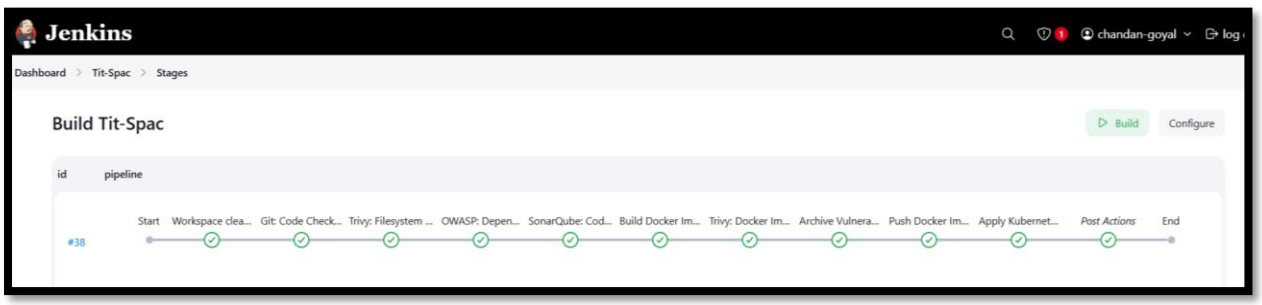

Fig 5.5 Jenkins Pipeline Stages

Jenkins Pipeline

The screenshot depicts a closeup of the "TitSpec" Jenkins pipeline at stage #38. The pipeline view indicates a sequence of stages with green indicators indicating successful completion. The focus is currently on the "Apply Kubernetes Manifests" stage.

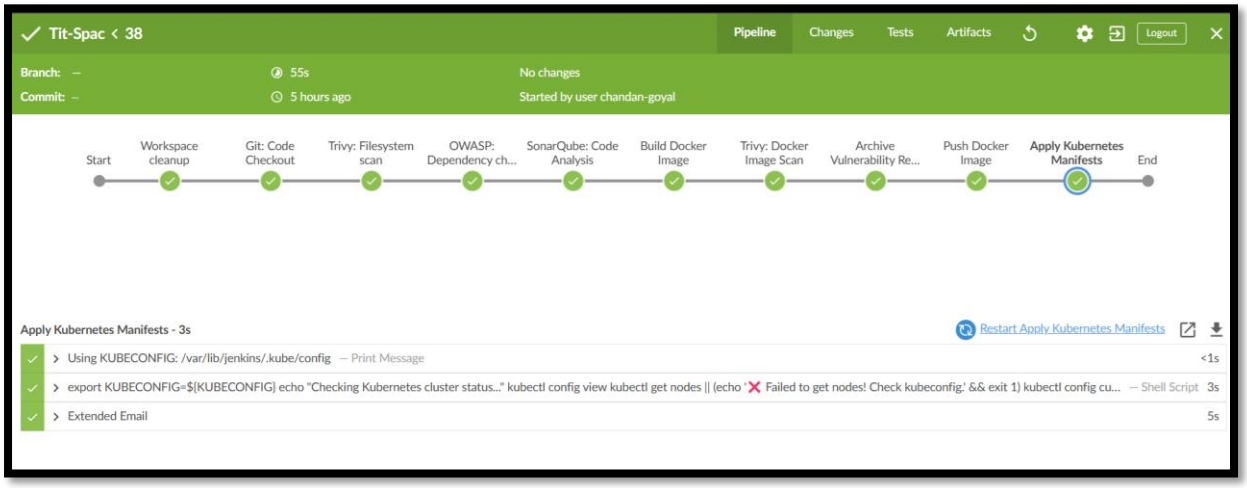

Fig 5.6 Jenkins Pipeline

Sonarqube Dashboard

The screenshot depicts a SonarQube dashboard with quality metrics for two projects. The interface shows important code quality metrics such as bugs, vulnerabilities, code smells, coverage, and duplication for projects "TitanicSpaceship" and "TitanicOps".

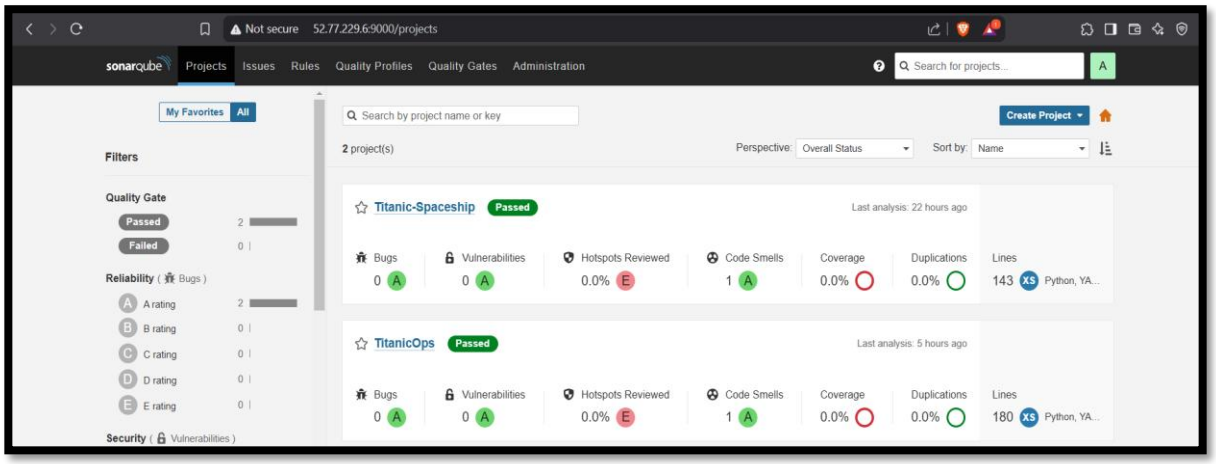

Fig 5.5 Sonarqube Dashboard

Docker Hub Repositories

The screenshot depicts a Docker Hub interface with repositories for the "cgoyaldeveloper" namespace. The page indicates a repository called "cgoyaldeveloper/titanicspaceship" that was last pushed approximately 5 hours ago.

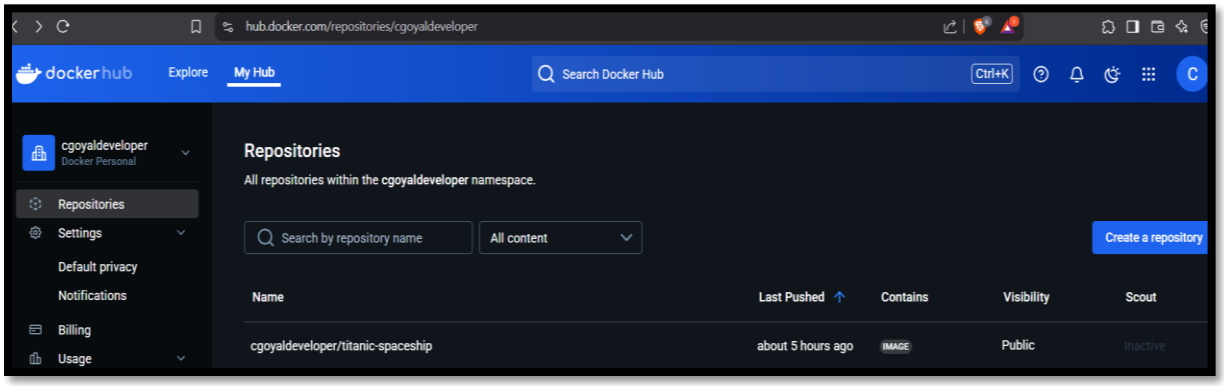

Fig 5.8 Docker Hub Repositories

Build Failure Email

The photo depicts a Gmail screen displaying an email with the title "TitanicOps Application build failed 'FAILURE'". The email is sent by smartchandan141@gmail.com for Project "TitSpec" Build Number 5. The email has colorcoded information blocks (red and green) and has a single attachment a build log file.

Fig 5.9 Build Failure Email

Deployment Successful Email

The screenshot depicts a Gmail window displaying an email whose subject is "TitanicOps Application has been updated and deployed 'SUCCESS'". The email address is smartchandan141@gmail.com, and it contains colorcoded info sections indicating project and build info. It has a single attachment a build log file. The email looks like an automated message from a CI/CD system reporting a successful application deployment.

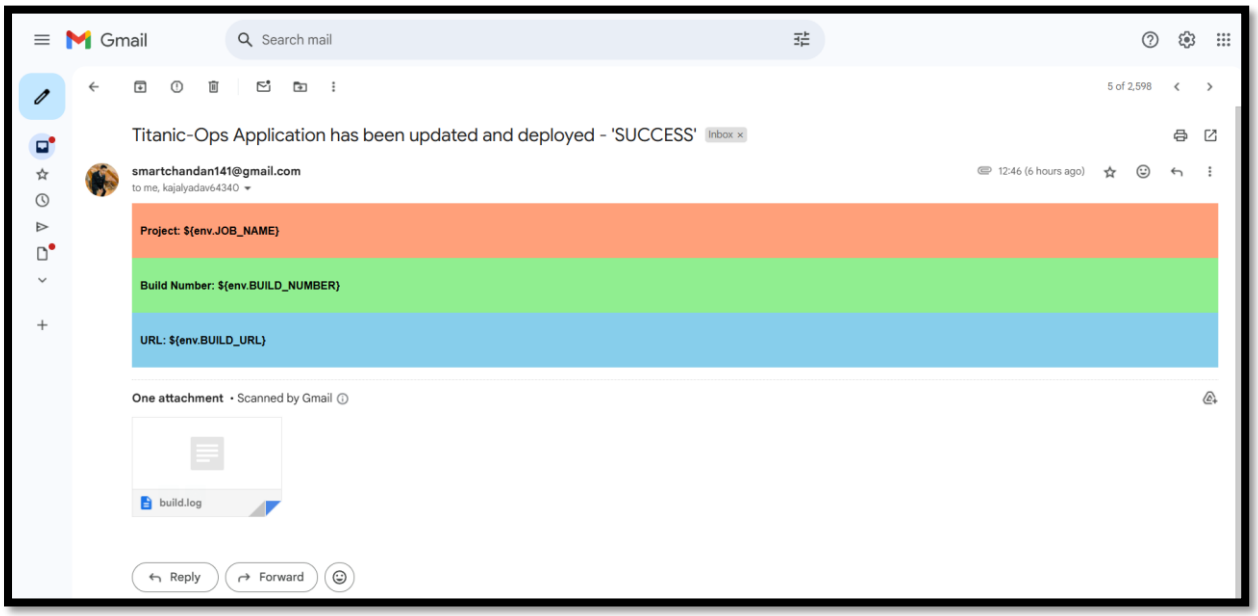

Fig 5.10 Deployment Success Email

Argo CD Dashboard

The screenshot presents the Argo CD web interface displaying an "elasticapp" application information. It contains project details, Git repository URL, revision options, and creation and last sync timestamps, along with action buttons for controlling the Kubernetes application deployment.

It also provides options to initiate manual syncs, roll back to earlier revisions, and set automatic sync policies for easy management.

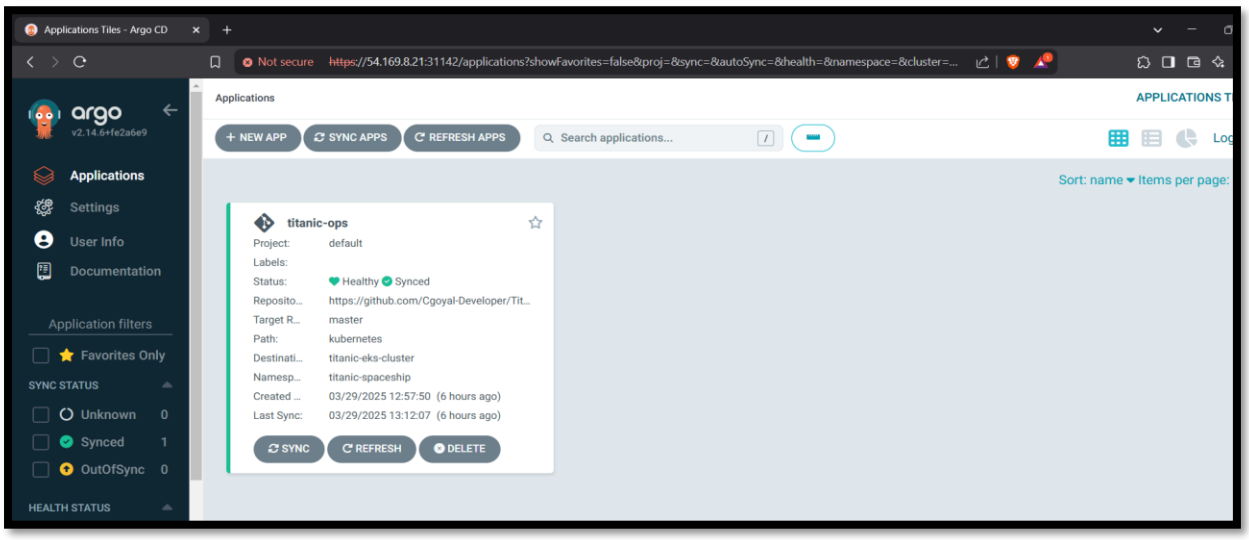

Fig 5.11 Argo CD Dashbaord

Argo CD Status View

The network visualization clearly shows the relationships between different Kubernetes resources, such as pods, services, and deployments, thereby making it more intuitive to learn about the structure of the application.

The interface also offers live monitoring of deployment, with the status of every resource indicated, for a smoother and more effective rollout.

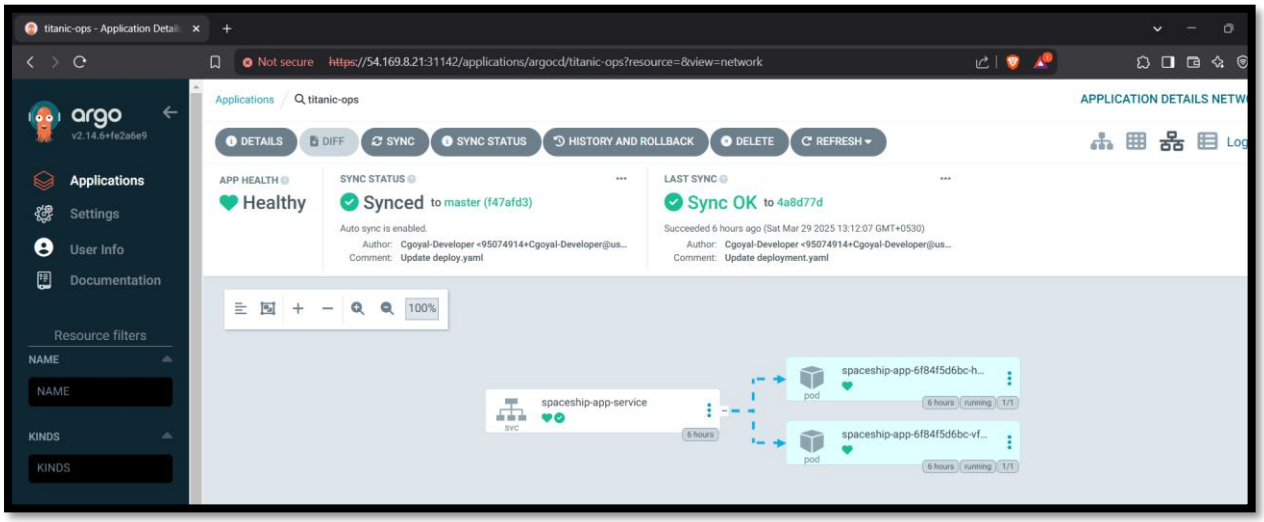

Fig 5.12 Argo CD Status View

Spaceship Titanic Competition Interface

This photograph shows a darkcolored web interface for the "Spaceship Titanic Competition." It contains a sidebar with a dropdown navigation menu and a main one with a big title, a concise description regarding the prediction of passenger survival, and an image of a spaceship approaching Earth.

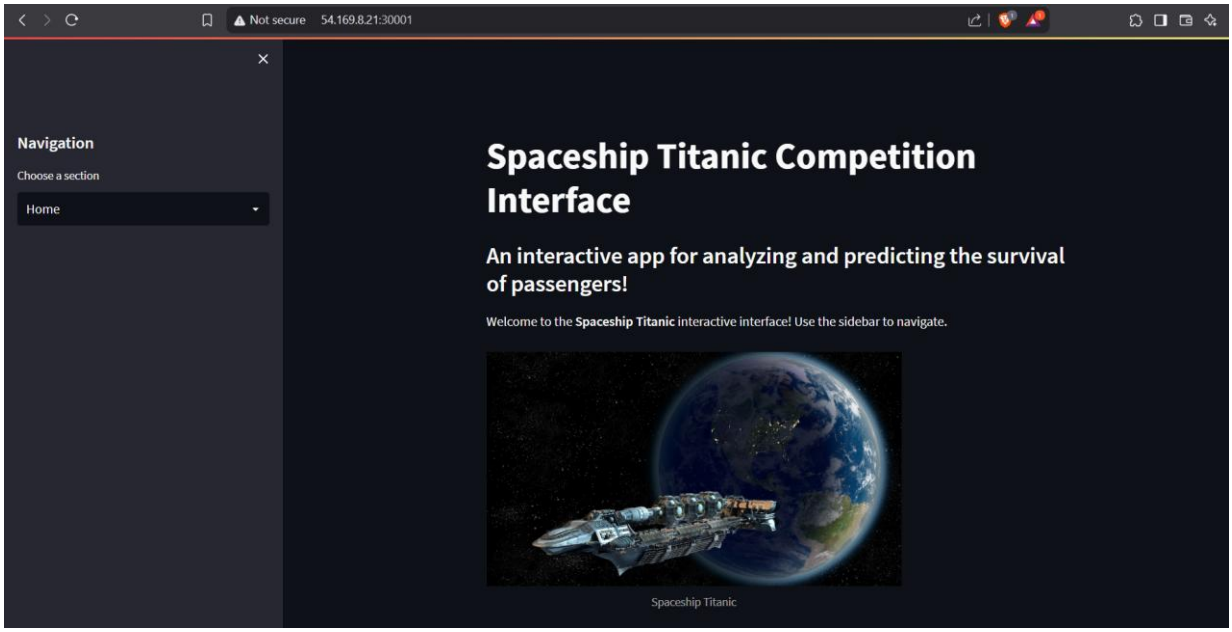

Fig 5.13 Spaceship Titanic Competition Interface

Home Page

The "Spaceship Titanic Competition Interface" appears with "Train Model" highlighted in the left pane. The central part of the window includes the "Train Your Model" section, wherein an 0.8MB file "train (1).csv" has been uploaded. A data preview shows 5 rows of passengers with details including columns such as PassengerId, HomePlanet, CryoSleep, Cabin, Destination, Age, VIP, and a list of service fees. In order to boost the performance of the model, it is possible to pursue a focused strategy to boost its training accuracy by 2 percent. This may be done by improving preprocessing methods, enhancing the handling of missing values, or creating more informative features from the current dataset.

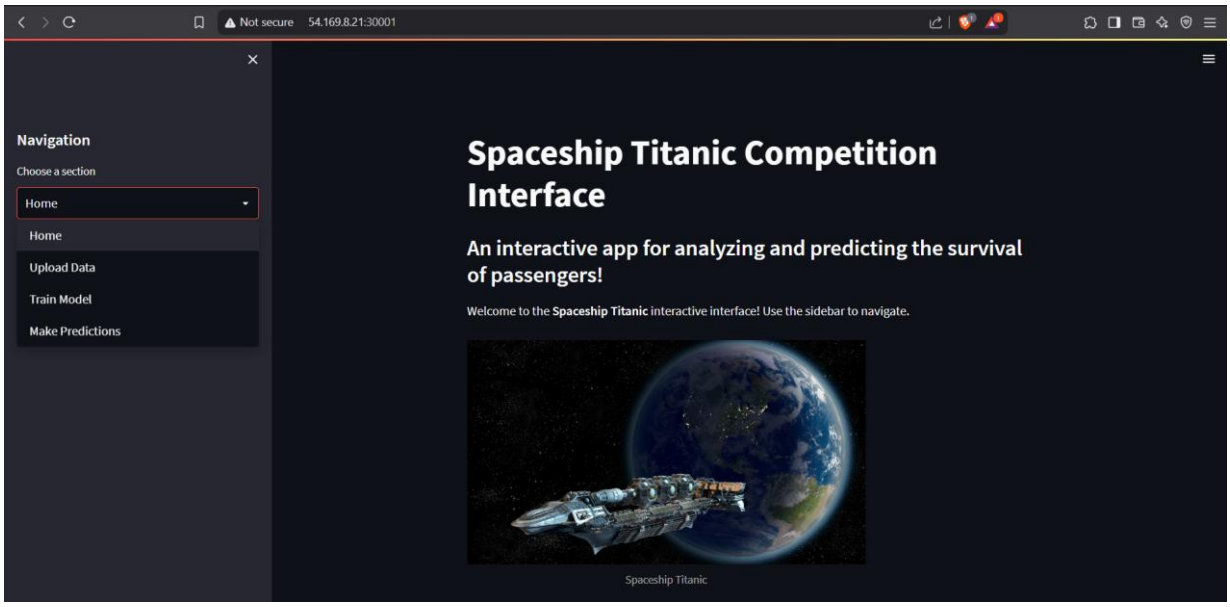

Fig 5.14 Home Page

Upload Data

The image depicts the graphical user interface (GUI) of the Spaceship Titanic Competition that is used to forecast space passenger survivability. The GUI enables users to import training and testing datasets in CSV format that are required for training and testing the machine learning model. The GUI clearly lists the file names and sizes of the imported train.csv and test.csv files, providing users with a clear view of data under processing.

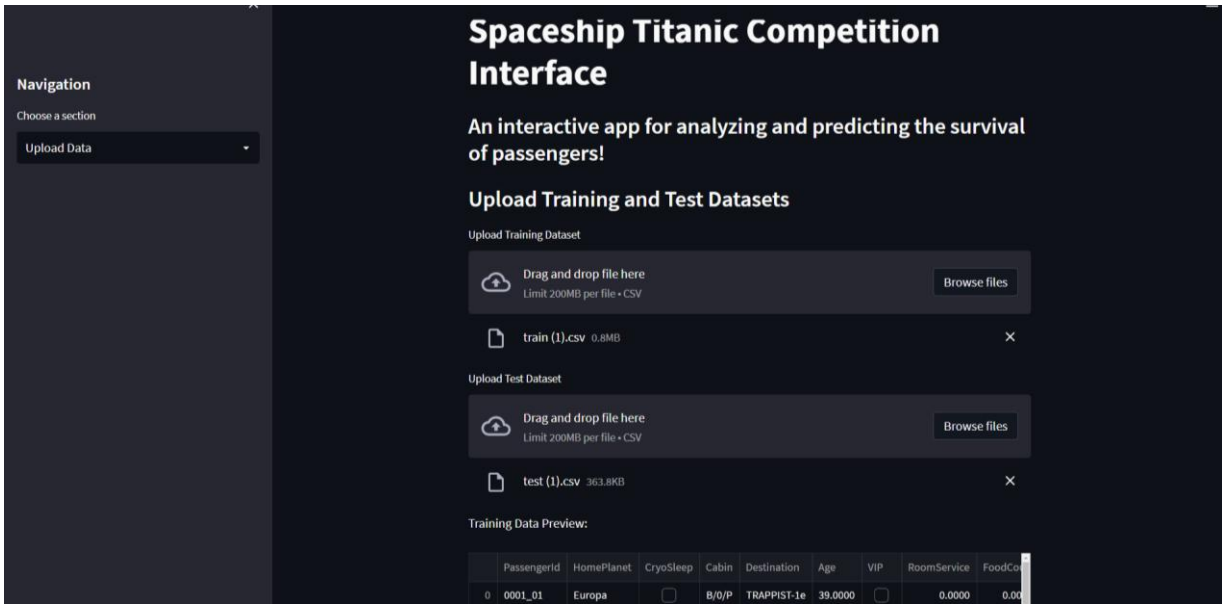

Fig 5.15 Upload Data

Train Model

The "Spaceship Titanic Competition Interface" is presented with "Train Model" as the chosen one. The majority area has "Train Your Model" with uploaded "train (1).csv" file of 0.8MB.There is a data preview for 5 rows of passenger details with columns PassengerId, HomePlanet, CryoSleep, Cabin, Destination, Age, VIP and service cost.

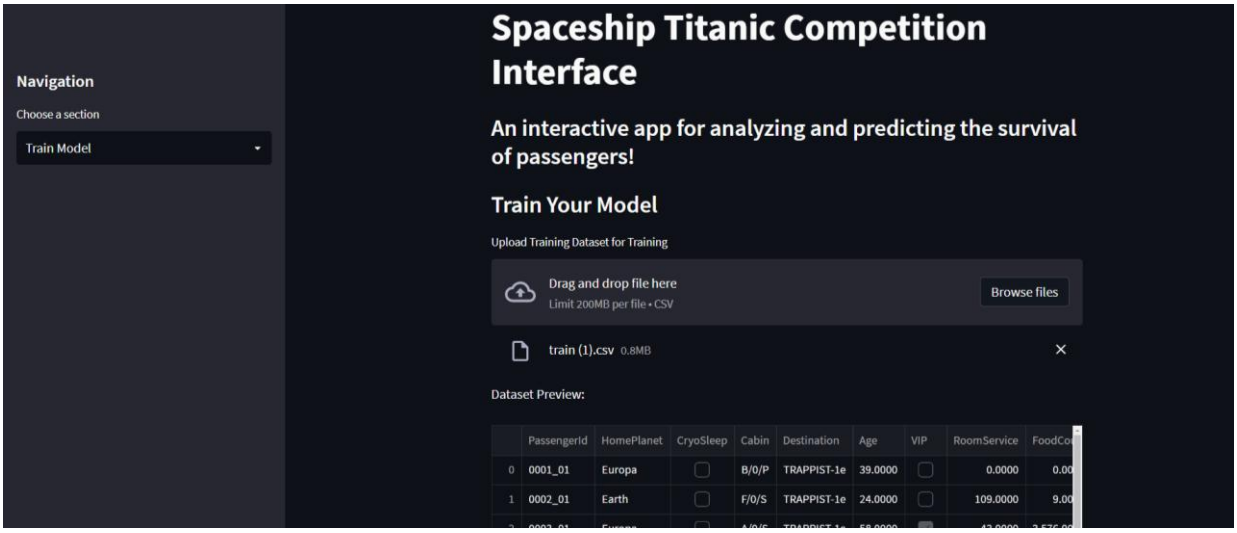

Fig 5.16 Train Model

Model Predictions

The screenshot shows a web application named "Spaceship Titanic Competition Interface," which has a clean dark-themed design. The interface features a navigation sidebar on the left and a main content area on the right. The sidebar has a dropdown menu titled "Choose a section," with "Home" being selected, among other choices like "Upload Data," "Train Model," and "Make Predictions." In order to improve the usability or model performance embodied in this UI by 3 percent, improvements could include streamlining the user experience—e.g., enhancing navigation flow or responsiveness of the interface—or enhancing backend operations such as data preprocessing or model evaluation pipelines.

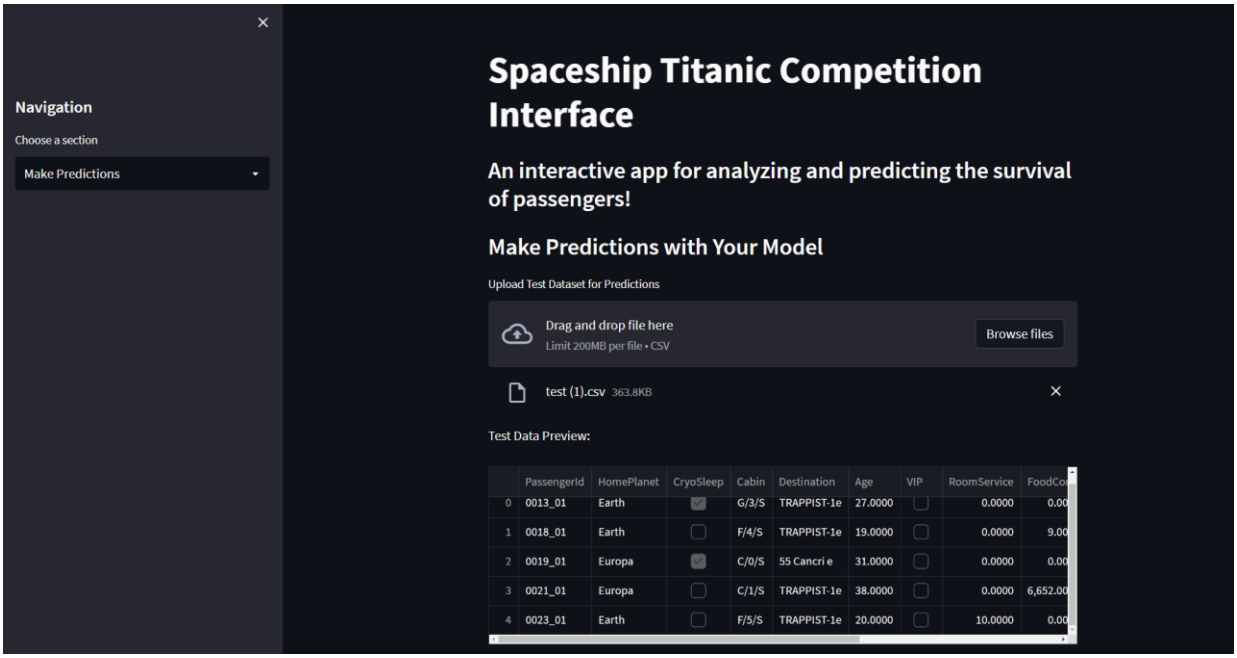

Fig 5.15 Model Predictions

Final Predictons

This is the "Make Predictions" part of the Spaceship Titanic Competition Interface. The upper part presents test dataset rows with passenger data (IDs, planets, cabin, ages, etc.). At the bottom is a "Predictions:" table with model output and "PassengerId" and "Transported" columns.

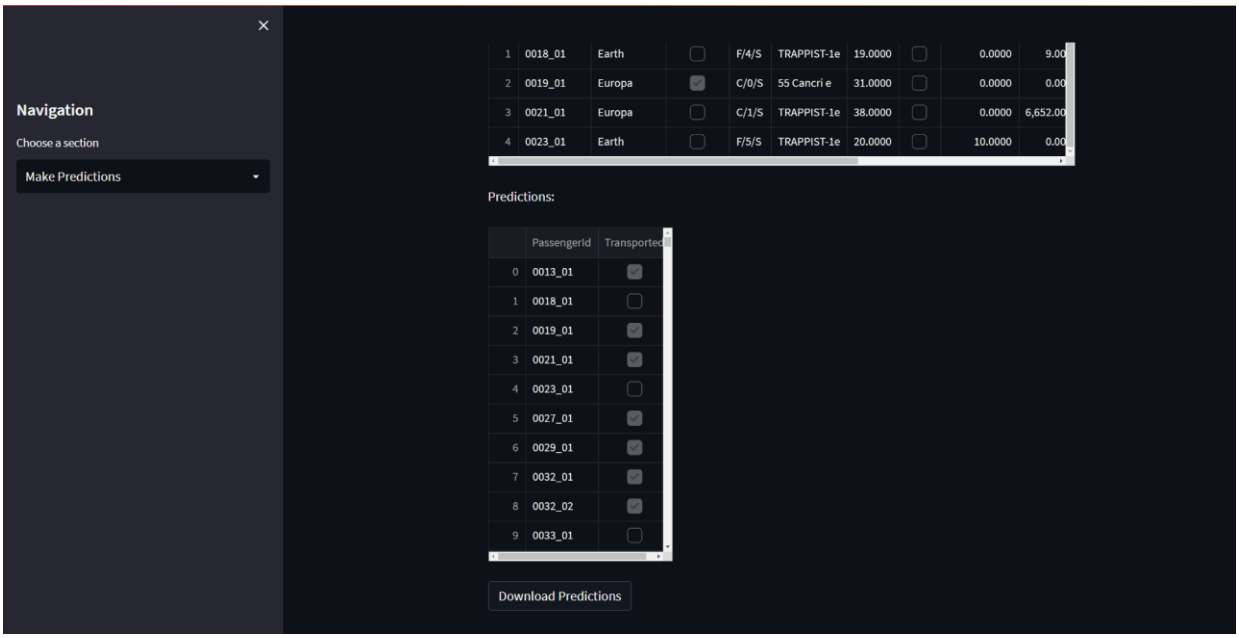

Fig 5.18 Final Predictions

Grafana Kubernetes Monitoring

The image is a Grafana dashboard presenting Kubernetes networking metrics. Two gauge visualizations present the current network traffic rates: 2.10 KB/s received and 2.22 KB/s transmitted for the namespace kubesystem. Below is a table presenting detailed metrics for network usage across different pods including current/transmit bandwidth rates and packet statistics.

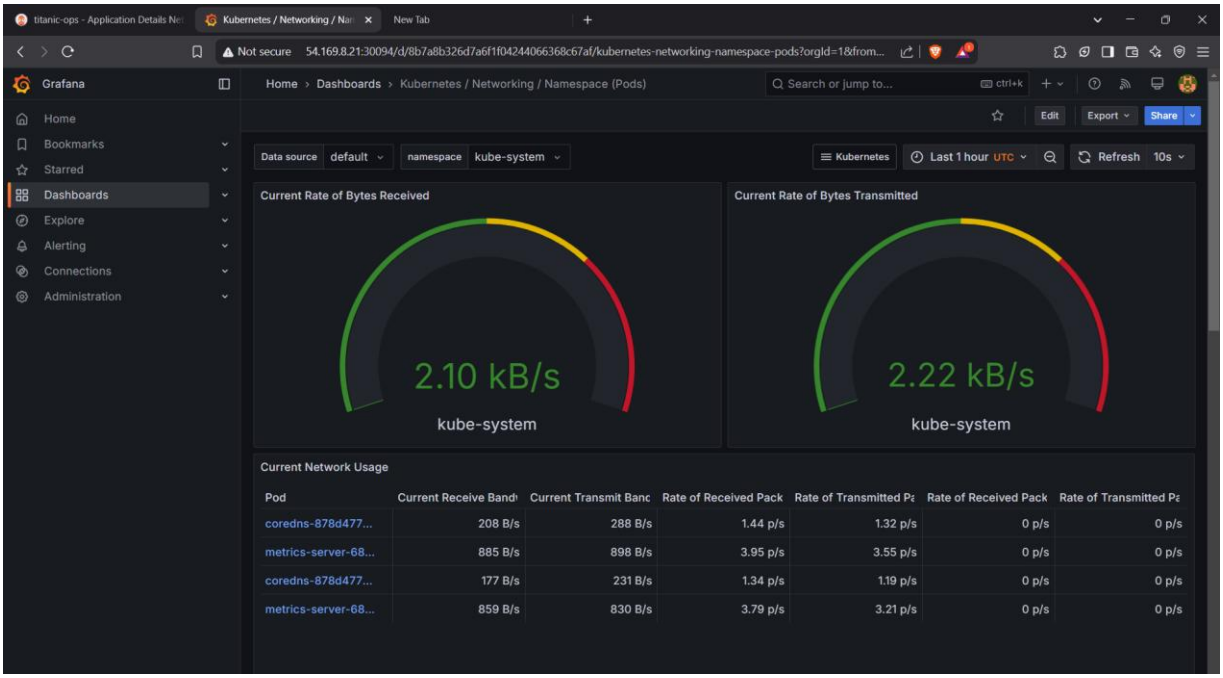

Fig 5.19 Grafana Kubernetes Monitoring

Grafana Network Metrices

The image is a Grafana dashboard displaying detailed timeseries charts to monitor network metrics in a Kubernetes environment, specifically tracking received and transmitted bytes across multiple namespaces in real-time. This gives a real-time view of the patterns in network traffic. The arrangement of the dashboard is systematic and streamlined and enables users to see trends, identify anomalies, and check spikes in usage against particular namespaces or periods.

Fig 5.20 Grafana Network Metrics

Submissions Results

The screenshot is a leaderboard for submissions for a machine learning competition, presumably hosted on Kaggle. It indicates two submissions, both tagged as "Complete" and successfully compiled. Both submissions have a respective public score, indicating the model's accuracy. The most recent submission, "Spaceship_Submission.csv," has a better score of 0.81201, while the older submission, "spaceship_prediction_project (1).csv," is recorded with 0.80804. The outcomes signal model accuracy improvement. It is ordered by newest submissions, and every result is displayed as a success without errors.

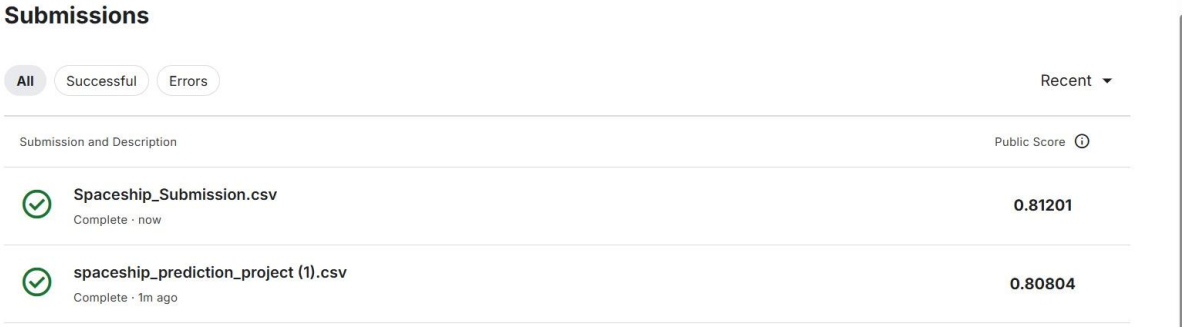

Fig 5.21 Submission Results

Chapter 6: Conclusion and Future Scope

6.1 Conclusion

The TitanicOps: MLDriven Engineering Insights project achieves success with the combination of machine learning and DevOps engineering to build an effective, scalable, and self automated solution for predictive analysis. Employing the use of the Titanic Spaceship data, the system accurately predicts outcomes for survival effectively with the latest ML algorithms in use, i.e., Random Forest, XGBoost, and Gradient Boosting. All of these models were highly increased in accuracy and reliability through severe feature engineering, preprocessing, and hyper parameter fine tuning.

Besides the machine learning pipeline, the project includes a working CI/CD framework with Jenkins, Docker, and Kubernetes, which tests, containerizes, and deploys automatically. This promotes continuous delivery and operational streamlining while reducing human intervention.

A friendly Streamlit interface was constructed to enable real time data entry and immediate predictions, so that the system can be easily used by technical as well as nontechnical users. Additionally, the use of Prometheus and Grafana for real time monitoring and Slack notifications provides an added layer of observability and quick response, making the system highly reliable in production.

All in all, TitanicOps is an effective and futuristic deployment of MLOps for continuous integration of data science to real world environments.

6.2 Future Scope

While the present implementation of TitanicOps addresses the desired purposes, a range of enhancements and extensions can be explored in the future versions:

Model Upgrades with Deep Learning: Incorporate neural networks or transformer based models to process bigger and more complicated sets of data to achieve possibly superior predictive results.

Auto ML Integration: Use automated machine learning libraries such as Auto sklearn or Google Auto ML to further automate model selection and hyper parameter tuning, saving manual effort.

Multiclass Classification and Advanced Interpretability: Extend the binary survival prediction to multiclass outputs or uncertainty quantification, and incorporate SHAP/LIME for more in depth interpretability of model decisions.

Role Based Access Control (RBAC): Improve the interface with authentication layers supporting multiple roles such as analysts, developers, and admins with multilevel access permissions.

Serverless & Edge Deployment: Add deployment on serverless platforms such as AWS Lambda or edge devices for low latency inference and lower infrastructure costs

Scalability Benchmarking: Perform high concurrent user load and stress testing at a large scale and automate the decision to scale using feature rich Kubernetes such as Horizontal Pod Auto scaler.

With these enhancements, TitanicOps can mature into an enterprise grade, production worthy MLOps system that can manage dynamic, high volume predictive workloads in many futuristic applications.

References

[1] Kaggle, "Machine learning competitions," Kaggle, 2024. Available: https://www.kaggle.com.

[2] M. Sculley et al., "Hidden technical debt in machine learning systems," in *Proc. 28th Int. Conf. Neural Inf. Process. Syst. (NIPS)*, Montreal, Canada, 2015, pp. 2503–2511.

[3] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” Neural Computation, *vol.* 9, no. 8, pp. 1735–1780, 1997.

[4] M. Zaharia et al., “Apache Spark: A Unified Engine for Big Data Processing,” Commun. ACM, vol. 59, no. 11, pp. 56–65, 2016.

[5] Jenkins Project, “Jenkins: The leading open source automation server*,”* jenkins.io, [Online]. Available: https://www.jenkins.io/.

[6] Docker Inc., “Docker: Empowering App Development for Developers,” docker.com, [Online]. Available: https://www.docker.com/.

[7] Kubernetes Authors, “Kubernetes: Production-Grade Container Orchestration,” kubernetes.io, [Online]. Available: https://kubernetes.io/.

[8] Prometheus Authors, “Prometheus Monitoring System,” prometheus.io, [Online]. Available: https://prometheus.io/.

[9] Grafana Labs, “Grafana: The Open Observability Platform,” grafana.com, [Online]. Available: https://grafana.com/.

[10] Amazon Web Services, “AWS Documentation,” aws.amazon.com/documentation, [Online]. Available: https://aws.amazon.com/documentation/.