IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 1, November 2026
Impact Factor: 8.2

Design and Prototype Implementation of a Modular AI ModelOps Platform for End-to-End Machine Learning Lifecycle Management

Gade Shekhar Satish[1], Borhade Mandar Dattatray[2], Fulsundar Manali Martand[3], Kawade Prerna Gajendra[4], Prof. S. K. Said[5]
Department of Artificial Intelligence and Data Science[1-4]
Project Guide[5]
Jaihind College of Engineering, Kuran, India

Abstract : *Machine learning models are often developed through experiments and notebooks, but moving them into reliable production systems requires coordinated data management, experiment tracking, model versioning, deployment, and continuous monitoring. Disconnected tools and manual release steps can reduce reproducibility and make model updates difficult to govern. This paper presents the design and prototype implementation of a modular AI ModelOps Platform that brings key lifecycle operations into a common system. The platform combines a React and TypeScript dashboard, FastAPI-based control-plane APIs, PostgreSQL application metadata, Redis-backed background services, MLflow experiment tracking and model registry, a model router, and Docker-based model serving. Its delivery workflow uses GitHub Actions for validation and image building, GitHub Container Registry (GHCR) for versioned artifacts, and a pull-based staging release checker on an Ubuntu host. Before promotion, a candidate image is checked for source and version metadata and tested through health and prediction requests; rollback safeguards preserve the previous deployment when validation fails. Functional verification has confirmed a staging prediction from the registered demo-classifier model version 5. Drift analysis and governed retraining are included in the intended lifecycle feedback design, while fully autonomous retraining is treated as an extension requiring further validation. The paper presents the system architecture, implementation workflow, related research, evaluation limits, and future development directions.*

Keywords: MLOps, ModelOps, machine learning lifecycle, experiment tracking, model registry, FastAPI, MLflow, Docker, continuous integration, continuous deployment, data drift, model serving.

1. INTRODUCTION

Machine learning is increasingly embedded in software products, decision-support tools, and data-driven services. Building a model with acceptable offline performance is only one part of the engineering task. A production model also requires repeatable data preparation, experiment tracking, model packaging, version control, secure deployment, inference interfaces, runtime monitoring, and a process for responding to changes in incoming data. When these activities are handled through separate scripts and manually coordinated tools, teams can lose track of which dataset, code revision, parameters, and model artifact produced a particular deployment [1].

Machine Learning Operations (MLOps) applies software engineering and operations practices to the machine learning lifecycle. It extends conventional CI/CD by introducing data and model validation, experiment and artifact lineage, model-specific testing, deployment controls, and feedback from production monitoring. Unlike ordinary application releases, model releases may change because of new training data, model parameters, feature definitions, or evaluation

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 1, November 2026 Impact Factor: 8.2

results. A reliable platform must therefore manage both software artifacts and model artifacts while preserving traceability across them [1], [2].

The AI ModelOps Platform described in this paper is designed to provide a modular, cost-conscious environment for managing these activities. It uses open-source components and container-based services rather than depending on a single commercial cloud platform. The current implementation includes application APIs for datasets, predictions, and model registry operations; MLflow-based tracking and registry services; model routing and containerized serving; and versioned CI/CD and staging-release automation. A web dashboard brings project, dataset, training, registry, prediction, deployment, and observability functions into one interface.

The primary objective is to reduce the operational gap between model experimentation and a repeatable deployment process. The platform is organized around a control plane that coordinates lifecycle operations and separate serving components that execute predictions. A monitoring and drift-feedback path is included in the architecture so that later stages can evaluate model behavior and initiate governed retraining. This paper describes implemented functions separately from planned extensions and does not claim unmeasured model-accuracy or throughput improvements.

II. LITERATURE SURVEY

Research on MLOps has moved from defining the discipline to evaluating platform capabilities, lifecycle architectures, and automated adaptation in production. The following studies provide the technical foundation for the design choices in this project.

A. MLOps principles and reference architecture

Kreuzberger, Kühl, and Hirschl [1] present a broad overview of MLOps, synthesizing principles, functional components, organizational roles, architectures, and workflows. Their work explains why operational machine learning requires more than model training: data preparation, model development, deployment, monitoring, and coordination must be treated as parts of a connected system. This motivates the modular separation used in the proposed platform between the control plane, lifecycle services, tracking and registry, model serving, and release automation.

B. Lifecycle monitoring and operational governance

Khattak et al. [2] examine Machine Learning Health Operations (MLHOps) in healthcare and discuss initial pipeline setup as well as long-term monitoring, distribution shifts, model updates, privacy, fairness, and interpretability. Although the application domain is healthcare, the study reinforces a general requirement: operational monitoring and governance must continue after a model is deployed. The proposed platform adopts this lifecycle perspective through versioned model artifacts, monitoring functions, and a feedback path for drift assessment and controlled updates.

C. Capabilities of existing MLOps platforms

Berberi et al. [3] evaluate 16 open-source MLOps platforms and compare capabilities such as experiment tracking, model development, orchestration, model testing, deployment, inference, and monitoring. Their review demonstrates that platforms differ in the range and depth of lifecycle features they provide. This is relevant to the present project because a small engineering team may need to integrate specialized tools rather than depend on a single product. The proposed system uses MLflow for experiment tracking and model registration, FastAPI for control-plane APIs, Docker for packaging, and GHCR for image distribution.

D. Architectural structure and process views

Amou Najafabadi et al. [4] conduct a systematic mapping study of MLOps architectures using 93 primary studies. They categorize 39 architecture components and synthesize process steps and stakeholder responsibilities. Their structural and process-oriented view supports the idea that an MLOps system should be designed as a coordinated set of responsibilities rather than a single training script. The proposed platform follows this approach by separating data and metadata services, training and evaluation, tracking and versioning, inference, deployment, and monitoring.

E. Distributed model update and serving

Rodrigues et al. [5] propose an MLOps architecture for near-real-time distributed stream learning. Their architecture identifies inference, model update, versioning, and monitoring as important components and uses containerized

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 1, November 2026 Impact Factor: 8.2

microservices to support updates. The target workload differs from the current project, which focuses on a controlled staging workflow rather than continuous high-rate stream learning. However, the study highlights the need to separate model serving from model update and to preserve clear version transitions.

F. Drift-aware retraining and deployment automation

Wong and Perumal [6] describe an AI-driven retraining architecture that combines drift detection, retraining scheduling, and data-governance controls. Their work is directly related to the planned feedback path of the AI ModelOps Platform: a drift signal should not automatically replace a deployed model without checking data quality, evaluation results, and release constraints. Ahmed et al. [7] further discuss the challenges of integrating machine learning into continuous deployment pipelines, including model validation, data governance, retraining cost, monitoring, and interoperability. These studies inform the platform's staged promotion and validation approach.

G. Research gap and project motivation

The reviewed literature identifies a common lifecycle spanning data, training, model storage, deployment, monitoring, and updating. Existing tools provide mature solutions for individual parts of this lifecycle, but teams still face integration decisions, configuration overhead, and the need to connect model-level evidence with software delivery practices [1], [3], [4]. The project therefore focuses on an implementation-oriented modular platform that combines model lifecycle APIs with image-based release management and a verifiable staging process. Its contribution is an integrated engineering prototype, not a claim that it replaces specialized enterprise MLOps suites.

III. EXISTING SYSTEM

In a basic machine learning workflow, data scientists commonly prepare datasets, train models, evaluate metrics, save model files, and expose predictions through application-specific code. Experiment details may be stored in notebooks or local files, and deployment may depend on manually built containers or scripts. This approach is sufficient for small demonstrations, but it can become difficult to reproduce when several datasets, experiments, model versions, and environments are involved.

A fragmented workflow creates several operational problems. First, data, code, and model versions may not be linked consistently, making it difficult to reproduce a deployed result. Second, the model registry and serving layer may be managed separately, increasing the risk of serving an unintended version. Third, deployment automation may validate application availability without testing a real inference request. Finally, monitoring and drift analysis may exist outside the release process, leaving teams without a defined governance path for model updates.

Commercial and open-source MLOps systems address many of these concerns, but their capabilities, infrastructure requirements, and integration models differ [3]. A project team may need to combine several services while maintaining a consistent interface and deployment process. The proposed platform addresses this integration problem with a small set of clearly separated components and a versioned release workflow.

IV. PROPOSED SYSTEM

The proposed AI ModelOps Platform is a modular architecture that manages the operational path from dataset registration to model inference and release verification. The application is divided into a presentation layer, a FastAPI control plane, lifecycle and tracking services, model-serving components, and a delivery layer. These components communicate through defined APIs and container/network boundaries. The design aims to keep training and control operations separate from the runtime process that serves predictions.

A. Presentation and control-plane layers

The presentation layer is implemented with React, TypeScript, and Vite. The existing frontend includes pages for projects, dataset management, training jobs, model registry, predictions, deployment management, and observability. These screens provide a consistent operator interface for lifecycle tasks. The backend control plane is implemented with FastAPI and exposes application APIs for dataset operations, model registry functions, and prediction requests,

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 1, November 2026
Impact Factor: 8.2

together with the platform's project and deployment capabilities. API validation and service-layer logic keep request handling separate from data access and ML execution.

B. Data, metadata, and asynchronous services

Dataset and project information is managed through backend services and relational metadata. PostgreSQL is used as the platform database for application records. The current MLflow tracking service uses a SQLite backend store and a local artifact directory, while Redis provides a queue and coordination layer for background work. A worker process is used for tasks that should not block interactive API requests. This separation allows the control plane to remain responsive while longer lifecycle operations are executed by supporting services. Dataset records and related metadata provide the basis for associating training work with its input data.

C. Experiment tracking and model registry

MLflow provides experiment tracking and model registry functionality. Experiment runs can record parameters, metrics, and artifacts, while registered model versions provide a reference for serving and lifecycle management. The platform distinguishes the version of the overall application release from the version of an ML model: for example, a GHCR image tag identifies a model-server software release, whereas a registry version identifies a particular trained model artifact. This distinction is important for traceability and rollback.

D. Training, evaluation, and drift feedback

The ML code is organized into modules for data handling, training, evaluation, inference, tracking, registry operations, drift analysis, and governance. A typical lifecycle begins with dataset selection and validation, continues through training and evaluation, and records the accepted model in the registry. Monitoring can compare incoming production data or prediction behavior with a baseline and produce drift evidence. The intended feedback workflow is: monitor model behavior, detect a potential drift condition, validate the data, schedule retraining, evaluate the candidate model, register an accepted version, and release it through the controlled deployment path. The complete autonomous retraining loop is not claimed as fully validated in the current implementation; it remains a governed extension of the architecture.

E. Model routing and containerized inference

The model router provides a central entry point for prediction requests and selects the registered model version to be served. The model server runs in a Docker container and exposes health and prediction endpoints. Containerization isolates the inference runtime and its Python dependencies from the control plane. A staging instance provides a controlled environment in which a candidate release can be tested before it replaces the currently active staging service.

F. CI/CD and pull-based staging deployment

GitHub Actions automates repository validation, test execution, semantic-version checks, and model-server image building. Successful release workflows publish versioned images to GitHub Container Registry. The staging host uses a system-level systemd timer and a pull-based release-check script to periodically inspect GitHub tags. When a newer release is available, the script verifies the image source and version labels, pulls the image, and invokes the deployment script. A candidate container is started on a temporary port and checked through health and real prediction requests. Only after these checks pass is the candidate promoted to the staging port. If a validation or promotion step fails, rollback safeguards are used to restore the previous staging container. This design avoids exposing the Docker daemon or requiring an inbound deployment connection from the public CI runner.

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 1, November 2026
Impact Factor: 8.2

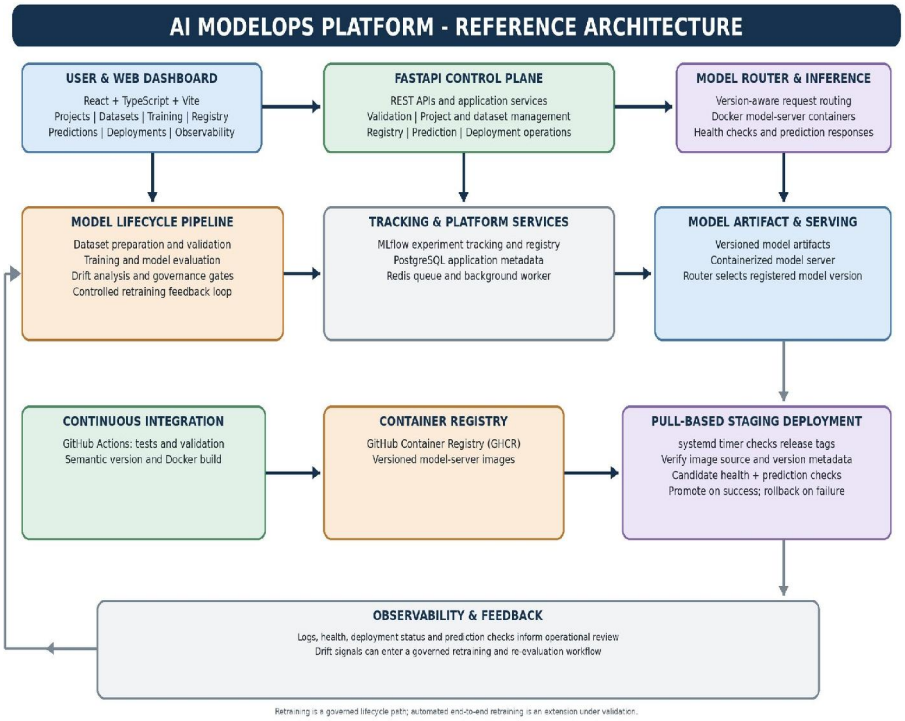


Fig. 1. Overall architecture and controlled model-release workflow of the AI ModelOps Platform.

V. IMPLEMENTATION STATUS AND FUNCTIONAL WORKFLOW

The implementation is being developed incrementally. The backend contains dedicated API modules for dataset management, prediction requests, and model registry operations. The frontend source contains corresponding application pages for key lifecycle activities. Docker Compose is used to run supporting services such as PostgreSQL and Redis, while MLflow provides the tracking and registry service. The model router and model server are deployed as separate services so that routing and inference can be updated independently.

The release workflow has been exercised with a versioned image published to GHCR and deployed to the Ubuntu staging host. During the verified release test, the checker detected a newer release tag, waited safely when the image was not yet available, and retried on a later polling cycle. After the image became available, source and version metadata were checked, a temporary candidate passed its health and prediction checks, and the candidate was promoted to staging. The final staging prediction endpoint returned HTTP 200 for the registered model demo-classifier, version 5, with prediction output [1]. This is a functional integration check; it is not a benchmark of predictive accuracy or serving throughput.

The implementation also includes failure-handling checks. A deliberately mismatched model version was used to make candidate validation fail, and a controlled port-conflict test exercised the deployment rollback path. These tests are intended to verify deployment safety rather than model quality. A formal evaluation using multiple datasets, model families, latency percentiles, resource utilization, and drift scenarios remains necessary.

A. End-to-end operational sequence

The intended operational sequence is summarized as follows: (1) a user creates a project and registers a dataset; (2) a training job consumes the selected data and produces evaluation evidence; (3) the accepted artifact is tracked and registered with a model version;

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429 Volume 6, Issue 1, November 2026 Impact Factor: 8.2

(4) the model server packages the selected artifact and exposes prediction endpoints; (5) GitHub Actions validates and builds a versioned serving image; (6) GHCR stores the release artifact; (7) the staging checker pulls, verifies, and tests the image before promotion; and (8) logs, health information, and prediction behavior provide feedback for subsequent operations. A drift event can enter a governed retraining workflow, subject to data and model validation gates.

B. Design considerations

The platform follows a modular approach to reduce coupling between user interface, API logic, model tracking, inference, and release automation. Version labels and registry versions provide traceability at different layers. Container-based serving creates a repeatable runtime environment, and candidate validation reduces the risk of promoting a broken release. The pull-based deployment method is appropriate for a small self-managed staging host because the host initiates outbound checks and pulls the artifact instead of exposing a privileged deployment interface to a remote runner. The present deployment target is staging; production high availability and multi-node orchestration are outside the verified scope of this version.

VI. EVALUATION SCOPE AND LIMITATIONS

The current evaluation is integration-oriented. It verifies that versioned release artifacts can be discovered, checked, pulled, started as a candidate, tested with health and prediction requests, and promoted to a staging service. It also exercises failure paths to confirm that an unsuccessful candidate does not replace the previously working staging deployment. These checks demonstrate the connection between the release pipeline and the inference service, but they do not measure the predictive quality of the model or the performance limits of the platform.

The current evidence is limited to a self-managed staging host and a functional demo-classifier prediction. No controlled results are reported for high-concurrency traffic, p95 or p99 inference latency, model accuracy across multiple datasets, drift-detection precision, retraining time, or multi-node availability. The architecture should therefore be considered an evolving prototype. A formal evaluation should compare manual and automated release workflows and report reproducibility, deployment success rate, rollback recovery time, resource utilization, API latency, and model-quality measures under normal and drifted data conditions.

VII. FUTURE SCOPE

- Complete and evaluate automated drift detection using explicit statistical thresholds and reference datasets.
- Connect drift events to governed retraining, candidate evaluation, approval, and registry promotion.
- Add data lineage, dataset versioning, feature validation, and reproducible training manifests.
- Expand model monitoring with prediction quality, latency, error rate, throughput, and resource metrics.
- Introduce role-based access control, audit trails, secrets management, and stronger API security.
- Add model explainability and fairness checks as configurable evaluation gates.
- Extend deployment from a single staging host to multi-environment and Kubernetes-based orchestration after resource and reliability testing.
- Evaluate deployment repeatability, rollback time, API latency, resource consumption, and behavior under simulated data drift.

VIII. CONCLUSION

This paper presents the design and prototype implementation of a modular AI ModelOps Platform intended to connect machine learning development with controlled deployment and operations. The platform combines a React and TypeScript dashboard, FastAPI control-plane APIs, relational metadata services, Redis-backed background processing, MLflow tracking and registry, model routing, Docker-based inference, GitHub Actions, GHCR, and pull-based staging deployment. A key implementation feature is the validation of candidate releases through image metadata, health checks, and prediction requests before promotion, with rollback safeguards for failures. A staging inference test has confirmed that a registered model can be served through the deployed workflow. The architecture also defines a governed feedback path for drift analysis and retraining, which requires further implementation and evaluation. The

IJARSCT
International Journal of Advanced Research in Science, Communication and Technology
International Open-Access, Double-Blind, Peer-Reviewed, Refereed, Multidisciplinary Online Journal
ISSN: 2581-9429
Volume 6, Issue 1, November 2026
Impact Factor: 8.2

project therefore provides a practical foundation for reproducible model lifecycle management while identifying clear steps toward more advanced monitoring, governance, and continuous training.

REFERENCES

[1] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture," IEEE Access, vol. 11, 2023, doi: 10.1109/ACCESS.2023.3262138.

[2] F. K. Khattak, V. Subasri, A. Krishnan, C. Pou-Prom, S. A. Koçak, E. Dolatabadi, D. Pandya, L. Seyyed-Kalantari, and F. Rudzicz, "MLHOps: Machine Learning Health Operations," IEEE Access, vol. 13, pp. 20374–20412, 2025, doi: 10.1109/ACCESS.2024.3521279.

[3] L. Berberi et al., "Machine Learning Operations Landscape: Platforms and Tools," Artificial Intelligence Review, vol. 58, art. no. 167, 2025, doi: 10.1007/s10462-025-11164-3.

[4] F. Amou Najafabadi, J. Bogner, I. Gerostathopoulos, and P. Lago, "An Architectural Perspective on MLOps: Structures, Processes, Tools, and Stakeholders," Information and Software Technology, vol. 193, art. no. 108029, 2026, doi: 10.1016/j.infsof.2026.108029.

[5] M. G. Rodrigues, E. K. Viegas, A. O. Santin, and F. Enembreck, "A MLOps Architecture for Near Real-Time Distributed Stream Learning Operation Deployment," Journal of Network and Computer Applications, vol. 238, art. no. 104169, 2025, doi: 10.1016/j.jnca.2025.104169.

[6] H. M. Wong and S. Perumal, "AI-Driven Model-Retraining Architecture to Sustain Operational Accuracy in Data-Drifting Environments," in Proc. 2025 IEEE Symposium on Wireless Technology & Applications (ISWTA), 2025, doi: 10.1109/ISWTA68114.2025.11329530.

[7] K. R. Ahmed et al., "DevOps for AI: Challenges and Opportunities in Integrating Machine Learning Models into Continuous Deployment Pipelines," in Proc. 2025 7th International Symposium on Advanced Electrical and Communication Technologies (ISAECT), 2025, doi: 10.1109/ISAECT68904.2025.11318824.