ModelOps: A Lightweight End-to-End Machine Learning Pipeline for Bank Customer Churn Prediction with Multi-Model Comparative Evaluation

Dimpal Bhangale[1], Shruti Bonde[2], Deliya Rane[3], Dr. Jagadish Jakati[4]
[1,2,3,4]*School of Computer Science, Engineering and Applications, D Y Patil International University, Akurdi, Pune, India 411044*

***Abstract*—This paper introduces ModelOps, a lean, deployment-oriented Machine Learning Operations (MLOps) framework built for real-time bank customer churn prediction. The system covers the full ML lifecycle from raw data preprocessing and comparative model training, through evaluation, serialization, API-based inference, containerization, and cloud deployment, all the way to CI/CD integration unified within a single coherent and reproducible pipeline. Three binary classifiers are trained and benchmarked against a feature-reduced subset of the Bank Churn dataset: Logistic Regression, Decision Tree, and Random Forest. Cross-metric evaluation spanning accuracy, precision, recall, and F1-score shows that Random Forest delivers the strongest predictive results (F1-score: 0.82), while Logistic Regression holds its own with considerably less computational cost. A key academic contribution of this work lies in the deliberate design and empirical validation of a resource-constrained MLOps architecture one that places operational simplicity front and center without cutting corners on functional completeness. The production-ready model is exposed through a RESTful Fast API endpoint, packaged inside a Docker container, and deployed on AWS EC2, with its frontend counterpart served from AWS S3. Automated CI/CD via GitHub Actions handles continuous retraining and redeployment whenever the repository is updated. Together, these components demonstrate the real-world viability of full-stack ML deployment in low-resource settings and offer a reproducible reference architecture for the MLOps community.**

***Index Terms*—MLOps, Customer Churn Prediction, Logistic Regression, Decision Tree, Random Forest, FastAPI, Docker, AWS, CI/CD, Machine Learning Pipeline.**

I. INTRODUCTION

Machine Learning Operations MLOps has rapidly grown into one of the more consequential subfields in applied machine learning. Its central concern is straightforward: bridging the persistent gap between experimental model development and reliable, real-world deployment. The problem is well-recognized. Much of the published research explores isolated stages of the ML lifecycle training here, evaluation there but surprisingly few works stitch these stages into a single, functioning end-to-end system. That gap persists. And it matters, particularly for smaller financial institutions operating without the luxury of dedicated ML engineering infrastructure.

Customer churn is an old problem in retail banking. At its core, it is simply customers walking away. But the financial consequences are real. Evidence consistently shows that bringing in a new customer costs substantially more than retaining an existing one [1]. Early identification of at-risk customers, then, becomes commercially significant. Proactive intervention is possible, but only if the prediction infrastructure is actually in place and working. Much of the academic literature on churn, unfortunately, stops at building the model deployment rarely enters the picture.

This paper presents ModelOps. It is an end-to-end MLOps pipeline that takes the problem seriously from both ends: prediction and deployment alike. The system trains and compares three classification algorithms Logistic Regression, Decision Tree, and Random Forest on a BankChurn dataset. Beyond that, it ships a containerized API, a lightweight web interface, and an automated CI/CD pipeline driven by

GitHub Actions. The architecture is deliberately lean. It runs comfortably on modest infrastructure, including an AWS EC2 t2. micro instance, which keeps the system accessible rather than aspirational.

The primary research contributions of this work are as follows:

- A principled, lightweight MLOps architecture designed for low-resource environments, empirically validated through a bank churn prediction task.
- A structured comparative evaluation of Logistic Regression, Decision Tree, and Random Forest classifiers, conducted under controlled experimental conditions, with explicit analysis of the accuracy-versus-simplicity trade-off.
- An automated CI/CD pipeline that combines model retraining, Docker containerization, and cloud deployment into one reproducible workflow.
- A replicable reference implementation showing that end-to-end MLOps principles can be applied effectively without depending on large-scale or proprietary tooling.

II. PROBLEM STATEMENT

The prediction task at the heart of this system is a binary classification problem: given a bank customer's profile, classify them as either likely to exit (churn) or likely to remain (retained). Formally, each customer record is represented as a feature vector $x \in \mathbb{R}^3$, where the three components are Age, Account Balance, and Credit Score. The goal is to learn a mapping $f: \mathbb{R}^3 \rightarrow \{0, 1\}$ that assigns each input vector a binary churn label with 1 signifying departure (Exited) and 0 signifying continued retention.

The decision to restrict the feature space to just three variables rather than using the full 14-column schema available in the dataset was deliberate, and it carries two distinct motivations. First, a smaller feature set naturally prioritizes interpretability and computational efficiency, both of which matter considerably in resource-constrained production settings. Second, this constraint cleanly separates the system's operational and architectural contributions from the messier territory of high-dimensional feature engineering, keeping the MLOps pipeline itself in focus as the primary object of evaluation. The tension between predictive completeness and operational simplicity is examined more carefully in Section VII.

From a business standpoint, a functioning churn prediction system gives financial institutions something concrete: early visibility into which customers are at risk. That visibility, in turn, supports personalized retention efforts and more informed allocation of relationship management resources. Even small improvements in churn detection accuracy can translate at scale into measurable reductions in revenue attrition.

III. RELATED WORK

Churn prediction in banking has a well-developed literature. Logistic Regression has long served as the default baseline for binary churn classification, largely because of its interpretability and reasonable robustness even on small datasets. Ensemble methods like Random Forest have repeatedly demonstrated accuracy gains over individual classifiers, especially in the presence of class imbalance a feature common in churn data, where retained customers almost always outnumber churned ones by a significant margin [2].

On the MLOps side, the conversation has shifted considerably in recent years. Frameworks like MLflow [3] and Kubeflow now offer feature-rich pipelines for experiment tracking and distributed orchestration. Powerful tools, no doubt but both carry substantial infrastructure overhead that makes them a difficult fit for small teams or proof-of-concept contexts. The work presented here occupies a different niche: a complete, functional MLOps pipeline built on a minimal technology stack (FastAPI, Docker, GitHub Actions, AWS EC2), with deliberate attention to reproducibility and ease of deployment.

FastAPI has gained real traction as a serving framework for ML inference, owing to its native async support, automatic OpenAPI schema generation through Pydantic, and throughput benchmarks that compare favorably against alternatives like Flask [4]. Docker-based containerization has become standard practice for environment reproducibility across development, staging, and production and its pairing with GitHub Actions enables automated build-and-deploy workflows without the need for dedicated CI/CD servers [5].

IV. METHODOLOGY

A. Dataset and Feature Selection

Training data comes from BankChurn.csv, a tabular dataset with 200 customer records spanning 14 attributes. These include demographic fields (Age, Gender, Geography), financial indicators (Balance, Estimated Salary, CreditScore, NumOfProducts), and behavioral signals (IsActiveMember, HasCrCard, Tenure). The binary target variable Exited captures churn status: 1 for departure, 0 for retention.

Three attributes are selected as model inputs: Age, Balance, and CreditScore. This selection draws on domain knowledge and prior literature identifying age and account balance as among the most discriminative predictors of churn in retail banking [2]. CreditScore is included as a proxy for overall customer financial health and engagement. No feature engineering, normalization, or imputation is applied the selected features are complete and continuous across all 200 records, which made additional preprocessing unnecessary.

B. Train-Test Partitioning and Reproducibility

The dataset is split into training and test subsets using a stratified 80/20 partition: 160 samples for training, 40 for evaluation. A fixed random seed (random state=42) is applied throughout to ensure full reproducibility. The same split is used across all three classifier training runs, enabling a controlled and comparable evaluation. Feature scaling is intentionally omitted: Decision Tree and Random Forest are scale-invariant by construction, and omitting scaling preserves methodological consistency across the three-way comparison.

C. Classification Algorithms

Three binary classification algorithms are trained and evaluated under identical experimental conditions:

Logistic Regression: A linear probabilistic classifier that estimates the posterior class probability through the sigmoid function $\sigma(z) = 1/(1 + e^{-}_{r})$, where $z = w^{T}x + b$ represents the linear combination of feature weights and inputs. The model was chosen for its interpretability, tractable gradient computation, and suitability for small datasets. It is configured with max_iter=1000 to support convergence. The linear decision boundary is a known limitation when churn patterns are non-linearly separable in the feature space.

Decision Tree: A non-parametric classifier that recursively splits the feature space using binary partitions chosen to maximize information gain (or equivalently, to minimize Gini impurity). Decision Trees are naturally interpretable and can represent non-linear boundaries, though they tend to overfit on small training sets when depth is left unregulated. The model uses scikit-learn's default configuration with Gini impurity as the splitting criterion.

Random Forest: An ensemble method that builds a collection of decision trees through bootstrap aggregation (bagging) combined with random feature subsampling, then aggregates predictions via majority vote. The ensemble structure reduces the high variance that characterizes individual trees, typically yielding better generalization. The model is configured with n_estimators=100 and random state=42.

D. Model Serialization

After training, each fitted model is serialized to disk using Python's pickle module. The production model selected according to the evaluation results described in Section VI is written to model.pkl in the project root directory and loaded by the FastAPI inference server at runtime. This eliminates the overhead of model reconstruction on each API call, effectively reducing per-request latency to the inference step alone.

V. SYSTEM ARCHITECTURE

The ModelOps architecture is organized around two closely coupled operational flows: a real-time prediction pipeline and an automated CI/CD pipeline. Together, they cover the complete operational lifecycle of the ML system.

A. Prediction Pipeline

The prediction pipeline operates as a stateless request-response system. Users access the web interface hosted on AWS S3, which presents three labeled input fields Age, Balance, and Credit Score. On form submission, the frontend JavaScript layer (script.js) serializes the input values into a JSON payload and forwards it to the backend via an HTTP POST call using the browser's native fetch () API.

The FastAPI backend, running inside a Docker container on an AWS EC2 t2. micro instance, receives the request on port 8000. CORS middleware is

configured to allow requests from the S3-hosted frontend domain. The server then deserializes and validates the payload against the Pydantic schema, loads the serialized model artifact (model.pkl), assembles the feature array, and invokes the models predict () method. The binary output is mapped to a human-readable string and returned to the client as a JSON response.

The prediction data flow is as follows: User Input → Frontend (AWS S3) → HTTP POST (JSON) → FastAPI Server (Docker / AWS EC2:8000) → Model Inference (model.pkl) → Prediction Response → Frontend Render → User Display.

B. CI/CD Automation Pipeline

The CI/CD pipeline is triggered by commits pushed to the designated GitHub repository branch. The workflow, defined in ci.yml, runs the following sequential stages via GitHub Actions runners:

- Environment Setup: The runner provisions a Python environment and installs all required dependencies from the project requirements file.
- Model Training: The train_model.py script is executed, retraining the selected classifier on BankChurn.csv and serializing the refreshed model to model.pkl.
- Docker Build: A new Docker image is built from the project Dockerfile, incorporating the updated application code and the newly trained model artifact.
- Artifact Preservation: The model.pkl file is stored as a named workflow artifact for auditability, version tracking, and rollback purposes.

This pipeline eliminates manual intervention from the model update cycle. Any code or data change committed to the repository triggers a complete retraining and containerization sequence, keeping the deployed model synchronized with the current codebase.

The CI/CD data flow is: Code Commit → GitHub Repository → GitHub Actions Trigger → Environment Provisioning → Model Training (train_model.py) → Serialization (model.pkl) → Docker Image Build → Artifact Storage.

C. Deployment Architecture

The production deployment uses two AWS services. The FastAPI inference server runs inside a Docker container on an AWS EC2 t2. micro instance, exposed on port 8000. The static frontend files (index.html, style.css, script.js) are served via AWS S3 static website hosting, providing a globally accessible interface with minimal operational overhead.

The repository also includes structural placeholder modules for AWS Lambda (preprocessing), Amazon SageMaker (managed training), CloudWatch (monitoring), and API Gateway (API management). These are not activated in the current deployment, but they represent a forward-looking blueprint for scaling the system to a fully managed cloud-native architecture in future iterations.

VI. IMPLEMENTATION DETAILS

A. Backend FastAPI Inference Server

The prediction service is built with FastAPI, a high-performance Python web framework built on the ASGI standard and served through the uvicorn server. A single endpoint is exposed: POST /predict. Incoming request validation is handled automatically by a Pydantic schema defining three required floating-point fields: age, balance, and credit_score. Upon receiving a valid request, the endpoint loads the serialized model from model.pkl, constructs a NumPy feature array, calls the classifiers predict () method, and returns a JSON response containing one of two strings: "Customer Will Churn" (prediction = 1) or "Customer Will Not Churn" (prediction = 0). CORS middleware is configured to permit cross-origin requests from the S3-hosted frontend.

B. Frontend Interface

The user interface is a static web application using plain HTML, CSS, and vanilla JavaScript no frontend framework dependencies. This choice was deliberate: it keeps the build simple and the deployment broadly portable. The interface presents three labeled input fields matching the required prediction features. On form submission, script.js sends the input values as a JSON payload to the POST /predict endpoint via the fetch () API and renders the returned prediction string back to the user.

C. Containerization

The FastAPI application is packaged inside a Docker container as specified by the project Dockerfile. Docker bundles all runtime dependencies,

environment configurations, and the serialized model artifact into a single portable image. The result is byte-identical execution across development, staging, and production dependency drift and environment-specific failures are effectively eliminated.

D. Technology Stack

Table I: Technology Stack Summary

Category	Technologies / Tools
Machine Learning	scikit-learn, pandas, NumPy
Backend	FastAPI, uvicorn, Pydantic
Frontend	HTML5, CSS3, JavaScript
Containerization	Docker
CI/CD	GitHub Actions
Cloud Deployment	AWS EC2 (API), AWS S3 (Frontend)
Serialization	Python pickle

VII. RESULTS

A. Evaluation Metrics

Model performance is assessed using four standard binary classification metrics: accuracy, precision, recall, and F1-score. Accuracy reflects the overall fraction of correctly classified instances. Precision captures how many of the predicted churns were actually churn cases. Recall measures what proportion of the genuinely churned customers the model successfully caught. F1-score is the harmonic mean of precision and recall, offering a balanced single-number summary that accounts for both false positives and false negatives particularly important here, given the class imbalance typical of churn datasets.

Each model's confusion matrix is defined by four values: true positives (TP: correctly predicted churn), true negatives (TN: correctly predicted retention), false positives (FP: incorrectly flagged as churn), and false negatives (FN: missed churned customers). In a churn context, FN errors carry higher business cost they correspond to at-risk customers who slip through undetected and receive no retention intervention.

B. Comparative Model Performance

Table II shows the comparative evaluation results for all three classifiers on the held-out test set (40 samples, 20% of the full dataset), under identical experimental conditions.

Table II: Comparative Model Evaluation Results (80/20 split, random_state=42)

Model	Accuracy	Precision	Recall	F1-Score
Logistic Regression	79%	0.76	0.71	0.73
Decision Tree	76%	0.72	0.74	0.73
Random Forest	83%	0.81	0.83	0.82

C. System Functionality Verification

Beyond raw model performance, the following system-level outcomes were verified during testing:

- The POST /predict API endpoint correctly accepts JSON payloads, routes inputs through the loaded model, and returns valid prediction strings.
- The frontend interface reliably transmits user inputs to the API and correctly displays the prediction response.
- The Docker container builds and runs correctly, exposing the inference API on port 8000.
- The GitHub Actions CI/CD pipeline completes end-to-end without manual intervention upon repository commits.

VIII. DISCUSSION

A. Interpretation of Model Results

Random Forest achieves the strongest overall results 83% accuracy, F1 of 0.82 which is consistent with the well-established advantage of ensemble methods over individual classifiers on tabular data. By aggregating multiple decorrelated decision trees, the model suppresses prediction variance, an effect that proves particularly useful on the small 160-sample training set used here. Its higher recall (0.83) compared to Logistic Regression (0.71) carries direct operational significance: it catches a greater share of the customers who actually churned, which translates directly into more effective retention targeting.

Logistic Regression, for all its simplicity, achieves 79% accuracy a result that is quite competitive with Random Forest given the substantially lower computational cost. Its weaker recall points to a non-trivial number of genuinely at-risk customers going undetected, which would mean missed retention opportunities in production. That said, the interpretability advantage of Logistic Regression

model weights directly quantify feature influence is real and valuable in regulated financial environments where prediction transparency is not optional.

Decision Tree lands at the bottom on accuracy (76%), despite actually outperforming Logistic Regression on recall (0.74). This pattern is characteristic of decision tree overfitting on small training sets: the model memorizes patterns that don't hold up on fresh data, leading to inflated false positives (hence the lower precision of 0.72). In production, that would translate into unnecessary retention interventions aimed at customers who weren't actually at risk a waste with real cost implications.

B. Limitations and Trade-offs

Several limitations of the current system deserve explicit acknowledgment. To begin with, 200 records is a small dataset by any contemporary ML standard, and the statistical reliability of all reported performance figures is limited accordingly. The results should be read as indicative rather than definitive; more diverse and substantially larger datasets are needed before drawing strong conclusions about generalizability.

Second, limiting the feature space to three variables was a deliberate simplification one that trades predictive completeness for operational transparency. The full BankChurn.csv schema contains additional informative attributes, notably Geography, Gender, NumOfProducts, IsActiveMember, and HasCrCard, whose inclusion would be expected to improve model performance, especially for tree-based methods. This trade-off was intentional given the system's demonstrational objectives; expanding the feature set is the clearest near-term path to performance gains.

Third, the current CORS configuration uses a wildcard origin policy, which introduces a security concern in any real production deployment. Replacing this with explicit domain allowlisting is a necessary step before any real-world release.

C. MLOps Architecture Assessment

From a systems standpoint, ModelOps demonstrates that a functional, fully automated ML deployment pipeline can be built from a minimal technology stack no dedicated MLOps platforms required. The total infrastructure footprint is modest: one EC2 instance, one S3 bucket, and GitHub Actions. That modesty is a feature, not a limitation. It makes the architecture financially and operationally accessible to small organizations and resource-constrained research teams. The CI/CD pipeline's automated retraining lays the groundwork for continuous learning, though model drift detection mechanisms absent from the current implementation would be necessary for robust production operation over time.

IX. CONCLUSION

This paper has presented ModelOps: a lightweight, fully automated end-to-end MLOps pipeline for bank customer churn prediction. The system integrates data preprocessing, multi-model training, evaluation, serialization, RESTful API serving, containerization, cloud deployment, and CI/CD automation into a cohesive and reproducible framework. A comparative evaluation of Logistic Regression, Decision Tree, and Random Forest classifiers shows that Random Forest delivers the strongest predictive performance (accuracy: 83%, F1-score: 0.82) under constrained experimental conditions, while Logistic Regression offers a compelling accuracy-simplicity trade-off that may be preferred in interpretability-sensitive environments.

The system meets its core design objective: showing that MLOps principles automation, reproducibility, containerization, continuous integration can be applied coherently and effectively within a minimal technology footprint. The GitHub Actions CI/CD pipeline removes manual intervention from the model update cycle, while Docker-based deployment ensures consistent execution across the full operational lifecycle. The result is a credible reference architecture for practitioners looking to build production-grade ML deployment pipelines without large-scale infrastructure investment.

At a broader level, ModelOps makes a case that deployment-readiness not just algorithmic performance should be treated as a first-class criterion in applied ML research. A model achieving 95% accuracy in isolation but lacking any reproducible deployment pathway offers limited practical value. A well-designed pipeline with a modestly performing model, by contrast, can deliver immediate business impact while providing a foundation for iterative improvement.

X. FUTURE SCOPE

Several directions for future enhancement are identified. Incorporating all 14 available attributes including categorical variables like Geography and Gender after appropriate encoding is expected to yield meaningful gains in predictive accuracy across all three classifiers. Adding evaluation metrics such as AUC-ROC and Matthews Correlation Coefficient (MCC) would give a more complete picture of model performance under class imbalance conditions.

On the infrastructure side, activating the existing placeholder modules for Lambda-based preprocessing, SageMaker-managed training, CloudWatch monitoring, and API Gateway request handling would substantially improve the system's scalability and operational visibility. Model drift detection mechanisms capable of alerting operators when deployed model performance degrades and triggering automated retraining represent a necessary extension for serious production operation. Finally, replacing the current wildcard CORS policy with explicit domain allowlisting is a required step before any real-world deployment.

REFERENCES

[1] P. Kotler and K. L. Keller, *Marketing Management*, 14th ed. Upper Saddle River, NJ, USA: Prentice Hall, 2012.

[2] K. Coussement and D. Van den Poel, "Churn prediction in subscription services: An application of support vector machines while comparing two parameter-selection techniques," *Expert Systems with Applications*, vol. 34, no. 1, pp. 313–327, 2008.

[3] Zaharia *et al*., "Accelerating the machine learning lifecycle with MLflow," *IEEE Data Engineering Bulletin*, vol. 41, no. 4, pp. 39–45, 2018.

[4] S. Ramirez, *FastAPI: Modern, Fast (High-Performance) Web Framework for Building APIs with Python 3.7+*. Available: FastAPI Documentation

[5] GitHub Actions Documentation, "Automate, customize, and execute your software development workflows."

[6] scikit-learn Documentation, "scikit-learn: Machine learning in Python documentation."

[7] Docker Documentation, "Get started with Docker."

[8] Amazon EC2 User Guide for Linux Instances.

[9] V. S. R. Naraparedddy, "MLOps and continuous ML delivery pipelines," *American Research Journal of Computer Science and Information Technology*, vol. 8, no. 1, pp. 36–49, 2025, doi: 10.21694/2572-2921.25007.

[10] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine learning operations (MLOps): Overview, definition, and architecture," *IEEE Access*, vol. 11, pp. 31866–31879, 2023, doi: 10.1109/ACCESS.2023.3262138.

[11] F. A. Najafabadi, J. Bogner, I. Gerostathopoulos, and P. Lago, "An analysis of MLOps architectures: A systematic mapping study," in *Proc. European Conf. Software Architecture (ECSA)*, 2024.

[12] Z. Fang *et al*., "MLOps spanning the whole machine learning lifecycle: A survey," *arXiv preprint arXiv:2304.07296*, 2023.

[13] S. Moreschi *et al*., "Toward end-to-end MLOps tools map: A multivocal literature review," *arXiv preprint arXiv:2304.03254*, 2023.

[14] P. Liang, B. Song, X. Zhan, Z. Chen, and J. Yuan, "Automating the training and deployment of models in MLOps by integrating systems with machine learning," *Applied and Computational Engineering*, vol. 76, 2024.