UNIVERSITY
OF WOLLONGONG
AUSTRALIA

Reconciled AI

Applying GitOps Principles to Ensure AI Applications Stay Up and Running

Supervisor
Lecturer Dr. John LE

Submitted By
Group 19

Md Abu Sayed Khan Faisal	(8376311)
Farhan Javid	(8154429)
B. M. Ashik Mahmud	(8194063)
B. H. M. Riaz Uddin	(8087805)
Md Abu Sayed Khan Faisal	(8376311)

Faculty of Engineering and Information Sciences
School of Computing and Information Technology
University of Wollongong

Date: May 23, 2025

Abstract

We present Reconciled AI, a Git-centric MLOps pipeline that enhances the reliability and maintainability of ML applications in production. Modern machine learning (ML) applications in production face challenges in reliability and maintainability due to model drift, reproducibility issues, and complex deployment pipelines. Reconciled AI addresses these challenges by applying GitOps principles to ML operations (MLOps), leveraging declarative Kubernetes deployments to enable continuous integration, continuous delivery, and continuous training of ML models. Key components include GitHub Actions for CI/CD; Argo CD for automated synchronization with the Kubernetes cluster; and a Prometheus–Google Cloud Monitoring stack for performance metrics and drift detection, coupled with Google Pub/Sub to trigger automated model-retraining pipelines. We detail the project's implementation flow and system architecture, illustrating how this integration overcomes limitations of conventional MLOps—namely the lack of automatic model retraining, downtime caused by model performance drift, poor auditability of changes, and missing deployment traceability—and describe the monitoring-to-retraining feedback loop that closes the gap between performance-degradation detection and remedial updates in a declarative, traceable manner. An evaluation of the system shows improved uptime and resiliency of the AI application, demonstrating reduced mean time to recovery (MTTR) from model failures, decreased manual intervention by operators, and enhanced audit trails for model updates. We also discuss the benefits (e.g. reduced downtime, improved auditability) with supporting metrics and present textual descriptions of key architectural diagrams (system context, CI/CD workflow, sequence of retraining operations, and data flow), before examining current limitations. Finally, we outline future work directions such as advanced drift-detection techniques, increased platform portability to avoid vendor lock-in, and canary-deployment strategies for safer model updates. This report demonstrates that GitOps principles—when thoughtfully extended to MLOps—can significantly improve the reliability and continuous delivery of AI models in production.

Contents

1 Introduction

Artificial intelligence systems operate in environments that rarely stay still. As data distributions evolve, the statistical relationship between features and targets shifts—a phenomenon known as concept drift [5], causing models that once performed well to produce increasingly erroneous predictions. Empirical studies show that undetected drift can double error rates in a matter of weeks for high-velocity data streams[1], yet most production pipelines remain blind to these changes until users or downstream metrics signal trouble. Traditional Development and Operations (DevOps) practices, optimised for deterministic software binaries, are poorly equipped for this reality: they automate code builds and deployments but treat training data, model weights, and hyper-parameters as secondary artefacts handled by ad-hoc scripts and manual tickets. The result is a prolonged mean time to recovery whenever performance degrades and, more critically, a fragmented audit trail that complicates accountability in regulated sectors[1].

Recent Machine Learning Operations (MLOps) frameworks attempt to close this gap by extending continuous- integration and delivery concepts to machine learning, but what is often termed "Level-0 MLOps" still depends on human intervention for retraining and offers only partial lineage tracking[3]. In practice, data scientists export a new model offline, share it via file storage, and coordinate with operations teams to update serving infrastructure—steps that can take days[7], during which the application quietly accumulates technical and business debt. Moreover, because only code lives in version control, reproducing a given prediction or rolling back after a faulty release becomes cumbersome, threatening both reliability and trust[1].

This project proposes Reconciled AI, a reference architecture that fuses MLOps with the declarative paradigm of GitOps[2]. By treating every artefact—datasets, feature engineering scripts, trained model binaries, Kubernetes manifests—as first-class citizens in a Git repository, the system establishes a single, immutable source of truth. Continuous deployment controllers such as Argo CD reconcile the live cluster with this repository[10], while cloud-native monitoring (Prometheus and Google Cloud Operations) exposes accuracy and distribution metrics that trigger automated retraining workflows orchestrated by GitHub Actions. When drift thresholds are breached, the pipeline retrains a model on fresh data, commits the new artefact and configuration to Git, and propagates the update to production with minimal latency. The same reconciliation loop self-heals runtime failures: if a serving pod is modified out-of-band or crashes, the controller restores the declarative state without operator intervention.

By unifying continuous monitoring, retraining, deployment, and auditability in a single automated feedback loop, Reconciled AI aims to reduce downtime, shrink mean -time to recovery, and provide an end-to-end lineage of every change that affects model behavior. The project contributes both an open, cloud-native implementation and an empirical evaluation against metrics such as uptime, operator effort, and audit-trail completeness, offering evidence that AI services can be simultaneously adaptive and operationally dependable[4].

2 Background and Motivation

Deploying machine learning (ML) models at scale requires rethinking traditional DevOps assumptions. In classical software, once an application is deployed it remains stable until a new release or an infrastructure fault occurs, and monitoring focuses on system health (CPU, memory, error rates). In contrast, ML systems treat data as a first-class concern: even when code and infrastructure work perfectly, a model's statistical performance can regress as input distributions shift [5]. This model drift necessitates not only monitoring of system health but continuous tracking of model accuracy and the capability to update the model itself, not just its surrounding code. Below, we outline the key challenges in current MLOps practices that motivate our GitOps-based approach[6].

2.1 Challenges in Current MLOps Practices

Model Drift and Lack of Automated Retraining: Concept drift occurs when the statistical relationship between features and targets changes over time, causing accuracy to degrade. Detecting drift requires comprehensive performance monitoring and ground-truth feedback, yet most pipelines rely on manual retraining—often on fixed schedules or in response to obvious failures—resulting in long periods of sub-par service [6]. Integrating continuous training (CT)—automatically retraining and redeploying models when performance falls below a threshold—remains a nontrivial gap that leads to high mean time to recovery (MTTR) in critical applications [7].

Downtime and Degraded Service from Stale Models : An outdated model may still respond to requests, but poor predictions are functionally equivalent to downtime. Unexpected inputs can even cause runtime failures or high latency. Unlike traditional software—where blue-green or canary deployments mitigate risk [10]—few ML teams have mechanisms to detect misbehaving models and roll back to a known-good version automatically. As soon as performance dips, user experience suffers until engineers intervene, potentially causing revenue loss and churn [11].

Poor Auditability and Reproducibility : In standard software, version control and CI provide clear audit trails and rollback capabilities[1]. In ML, one must also track data versions, training code, and model artifacts. Yet models are often treated as ad-hoc binaries, with no record of which training run produced them[12]. This obscures lineage and complicates debugging, compliance, and rollback. Best practices (e.g. MLflow [13], DVC[14]) exist for experiment tracking, but seamlessly integrating them into deployment pipelines is challenging [3].

Missing Deployment Traceability and Environment Drift : Many ML deployments are manual or script-driven—"snowflake" environments where untracked changes cause drift and unpredictable failures [15]. Declarative deployments eliminate this by describing the desired end state (e.g., container image, replica count, resource requests) in code and using automated controllers to enforce it [10]. Storing Kubernetes manifests in Git and using a reconciliation engine ensures the live cluster always matches the versioned desired state, preventing configuration drift and improving reliability [2].

As shown in Figure 1, these challenges motivate our adaptation of DevOps and GitOps principles to the ML domain.

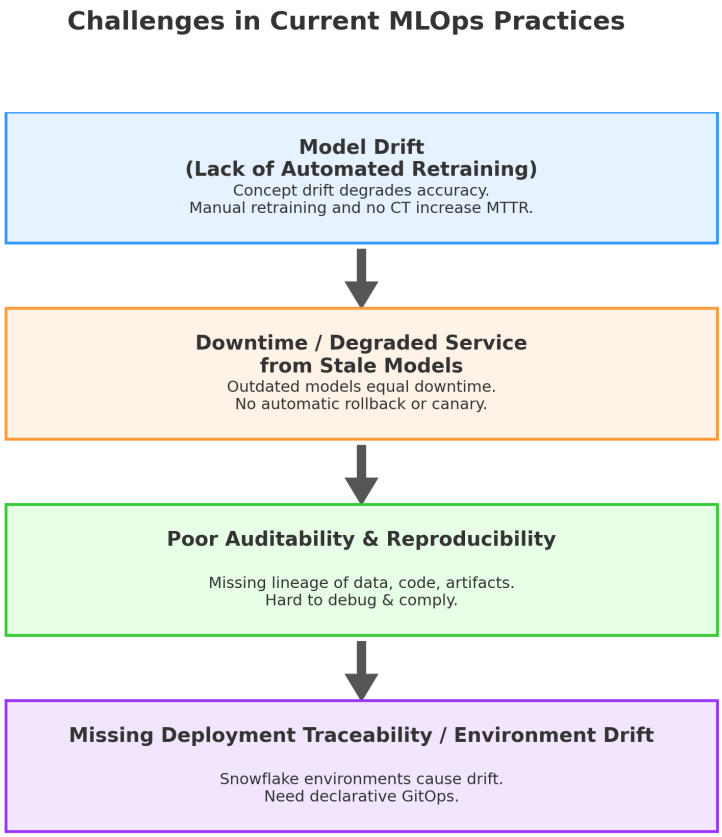

Figure 1: Challenges in Current MLOps Practices

2.2 MLOps and DevOps Principles Applied to GitOps

To address the challenges outlined above, we adapt proven DevOps practices—CI/CD, infrastructure as code, automated testing, and continuous monitoring—to the ML domain. GitOps extends these ideas by using Git as the single source of truth for declarative configurations and automating deployments via a reconciliation loop. In a typical GitOps workflow, all infrastructure and application state (e.g., Kubernetes YAMLs) live in a Git repository. An operator (such as Argo CD or Flux) continuously watches for commits: when changes appear, it applies them to the target environment; if the live state drifts—say someone edits a resource manually—the agent detects and reverts it, providing self-healing and eliminating manual ops intervention.**The key GitOps principles are:**

1. **Everything as Code :** All configs, pipeline definitions, parameters, and model references are stored in Git [2].
2. **Single Source of Truth :** Git's immutable history and review processes govern every change [16].
3. **Continuous Reconciliation:** Controllers compare live vs. desired state and converge them automatically [10].
4. **Immutable Audit Trail:** Every update corresponds to a commit, enabling easy rollbacks and forensic tracing [1].

These practices deliver clear benefits—easy rollbacks by reverting commits, collaborative reviews via pull requests, and stronger security through auditable, signed commits. For ML, "Everything as Code" means not only infra but also model version tags and data pointers in the same repo. Reconciliation prevents unapproved hot-fixes from persisting by overwriting any out-of-band changes with the Git-defined state.

Adapting CI/CD for ML: Continuous Integration (CI) involves frequently merging changes and running automated tests [17]. Continuous Delivery (CD) automatically deploys passing builds to production or staging [18]. In ML, CI/CD must encompass not just code but also retraining and redeployment of models. Continuous Training (CT) extends CI to new data, producing new model versions automatically [7]. According to Google's MLOps maturity model, Level 1 adds pipeline automation (CT) and Level 2 layers on CI/CD for deployment [19]. Reconciled AI targets a Level 2+ system: event-driven retraining (e.g., on drift alerts) and seamless deployment of updated models via GitHub Actions feeding into Argo CD. By treating retraining as just another CI job—complete with tests, builds, and a Git commit—the workflow unites training code and serving infrastructure into one continuous process [10][20].

Tooling and Best Practices:

1. **Monitoring and Alerting:** We use Prometheus [21] , Grafana [22], and Google Cloud Monitoring to track model metrics and trigger retraining pipelines on threshold breaches or statistical drift tests.
2. **Event-Driven Retraining:** Cloud providers (AWS SageMaker, Azure ML, Google Cloud Functions) offer similar patterns; we employ Pub/Sub as a language-agnostic event bus to launch retraining [14].
3. **Model and Data Versioning:** Tools like DVC and MLflow log experiments and datasets, but large binaries remain better stored in container registries. We bake model files into Docker images, versioned by Git commit tags [23], avoiding Git bloat while preserving traceability.

In summary, GitOps brings reliability and transparency to ML deployments by unifying code, model, and infrastructure changes under Git's governance and automating their ap-

plication. Reconciled AI integrates continuous monitoring, event-driven retraining, Git-centric delivery, and Kubernetes self-healing into a cohesive MLOps solution.

3 Related Work

3.1 GitOps and Declarative Infrastructure in DevOps

Modern DevOps practices emphasize GitOps, where the desired infrastructure state is stored in version control and continuously reconciled with the actual state [24]. Tools like Argo CD embody GitOps by treating Git as the single source of truth for deployments and using a pull-based reconciliation loop to apply changes to the runtime environment [24]. In this model, all infrastructure and application configuration is maintained under version control; when Git commits occur, the controller synchronizes the live cluster to match the desired state, and any manual modifications are automatically reverted [24].

This declarative approach effectively eliminates silent configuration drift by continuously monitoring both the repository and the live cluster, detecting divergences, and correcting them [24]. By logging every change as a Git commit, Argo CD provides an immutable audit trail, enabling teams to trace deployments and quickly identify the cause of failures [24].

While GitOps has proven highly effective for infrastructure, it does not natively support ML-specific artifacts (data versions, model binaries) or automatic retraining triggers. Next, we review frameworks that extend DevOps practices to address those gaps.

3.2 MLOps Frameworks and Technical Debt in ML Systems

As DevOps evolved into MLOps, researchers highlighted a new kind of technical debt unique to ML systems [25]. While ML model code is often minimal, the surrounding ecosystem—data pipelines, feature engineering, configuration, and glue code—can impose a massive, ongoing maintenance burden [25]. These issues result in fragile systems that require constant upkeep, and ad-hoc scripts tend to accumulate as projects grow [25].

To address this, various MLOps frameworks have been proposed to bring engineering rigor to ML workflows. Google's MLOps maturity model outlines a progression from manual deployment (Level 0), to automated pipelines (Level 1), to full CI/CD with Continuous Training (Level 2) [19]. At the highest level, changes in code or data can automatically trigger retraining and redeployment in a reliable and reproducible manner.

However, many organizations remain stuck at partial automation stages. Applying DevOps practices to ML requires extending CI/CD beyond code to also validate data, model metrics, and pipeline steps. Continuous Training (CT) is especially crucial—triggering retraining based on performance degradation or new data [25].

Despite available tools, integration remains a challenge. Researchers call for "next-generation platforms" to reduce complexity and bridge tooling gaps. Reconciled AI is designed in this context—to operationalize these best practices and automate orchestration, allowing teams to focus on model logic instead of infrastructure plumbing.

3.3 Model Drift Detection and Retraining Strategies

Once models are deployed, a major concern is model drift – the degradation of model performance over time as data distributions evolve [5]. In streaming or continually changing data environments, the statistical properties of inputs or target variables can shift, meaning the model's training data no longer reflects current reality [5]. This concept drift can lead to increasing prediction errors if left unaddressed [6].

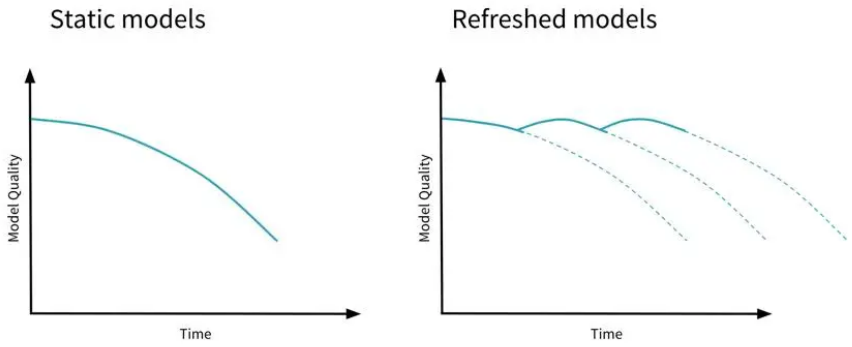

Figure 2: Model Drift

Prior research provides a rich taxonomy of drift types and detection methods, from monitoring feature statistics to advanced change-point detection algorithms [25].

The consensus is that maintaining model accuracy in production requires continuous monitoring of data and performance metrics, and triggering retraining or model updates when significant drift is detected [25].

In practice, many organizations implement scheduled re-training (e.g. retrain a model nightly or weekly) as a coarse solution[6]. More advanced setups use alerting systems: for example, tracking a divergence in input feature distributions or a drop in accuracy on fresh data, which signals that the model "no longer represent[s] the current environment" [6].

When such alerts occur, an automated pipeline can be invoked to re-fetch data, update the model, and redeploy it [7]. Empirical studies suggest that teams are less worried about gradual, expected drift because they often already retrain models frequently to incorporate fresh data [6].

In other words, a well-designed pipeline with frequent retraining can handle natural drift effectively [6].

The harder challenge is detecting unexpected or rapid drift (for instance, a sudden change in user behavior due to a new competitor or policy change) and responding quickly. Research on drift adaptation frameworks (e.g., active learning strategies, online learning algorithms) provides algorithms to handle these scenarios, but integrating them into production is non-trivial. Current MLOps platforms are just beginning to include drift detection modules (e.g. Evidently.ai for data drift monitoring, or AWS SageMaker's model monitoring). A limitation in existing literature and tooling is the siloed nature of these solutions – a drift detector might raise an alert, but orchestrating the retraining and redeployment in a traceable, reproducible way involves stitching together multiple systems [25]. This is precisely an area where Reconciled AI aims to contribute, by natively linking drift detection with an automated GitOps-driven retraining pipeline [2].

3.4 CI/CD Integration with ML Pipelines

Continuous integration and delivery (CI/CD) have revolutionized traditional software engineering, enabling rapid and reliable releases [17]. Bringing CI/CD to ML, however, introduces unique challenges [25].

Unlike conventional software, where a build artifact can be tested and released, an ML pipeline includes data ingestion, model training, evaluation, and deployment steps – all of which need to be orchestrated [18]. Researchers note that adopting CI/CD for ML systems is difficult, as it must encompass not just code changes but also data changes and model quality checks [19].

For example, a simple code commit might alter a preprocessing step that then invalidates the model if not tested with real data; thus integration tests for ML require testing end-to-end on datasets, not just unit tests on functions [17].

Continuous delivery in ML means automating the rollout of not a single service, but a pipeline that retrains and serves a model [20].

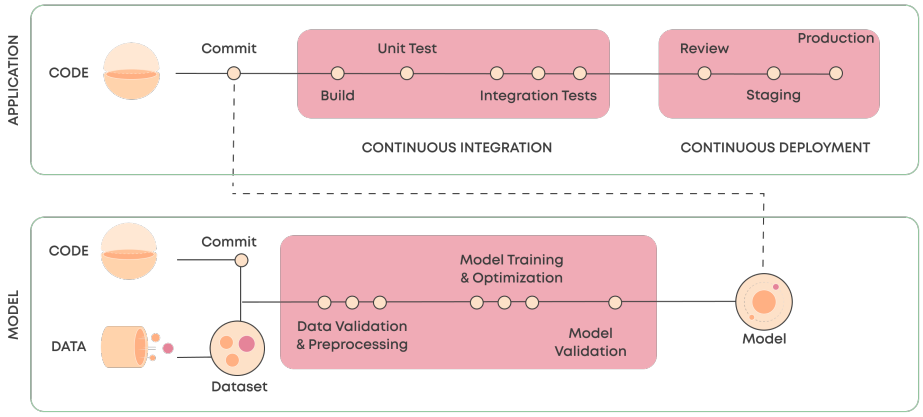

Figure 3: CI/CD Integration with ML Pipelines

Tools like Kubeflow Pipelines, TFX (TensorFlow Extended), and MLflow have emerged to define ML workflows that can be triggered and CI-tested [13]. Likewise, using Jenkins or cloud CI platforms for ML is becoming common, with additional steps for data validation and model evaluation baked into the pipeline [18]. Google's MLOps guidance explicitly calls out that CI is no longer just about code, but must include data and model validation, and that CD should handle deploying an entire training pipeline (not merely a model artifact) [19].

Moreover, they introduce continuous training (CT) as an extension of CD, whereby the system automatically retrains models when certain conditions are met effectively closing the loop between model monitoring and deployment [7]. Early case studies have shown the promise of such automation, but also the complexity: for instance, an automated pipeline must manage transitions carefully to avoid deploying degraded models (e.g. "shadow deployments" or A/B tests for new models before full release) [25]. Another aspect is the push vs pull deployment strategy in multi-tenant ML systems; some researchers explore using GitOps (pull-based) for ML model deployments to multiple environments [24]. In summary, CI/CD for ML is an active area of development, with known benefits in speeding up

iteration, but existing work highlights that many teams struggle to implement it end-to-end [25].

This suggests that a more integrated approach – combining familiar DevOps tools with ML-specific workflow automation – is needed to lower the barrier. Reconciled AI builds on this insight by integrating GitOps-based continuous delivery with ML pipeline orchestration, aiming to make continuous retraining and deployment as seamless as traditional software CI/CD.

3.5 Auditability, Version Control, and Traceability in ML Deployments

Auditability and traceability are critical in production ML for both debugging and compliance [1]. Every model prediction can be influenced by a host of inputs – data versions, feature extraction code, model hyperparameters – and organizations need mechanisms to trace outcomes back to their sources [12]. Traditional software version control (git) is insufficient by itself, as it doesn't track large datasets or model binaries [1]. In response, tools like ModelDB and MLflow have been introduced to log and version ML experiments [13].

ModelDB in particular provides an end-to-end system to track models as they are built, capturing metadata such as hyperparameters, data sources, and performance metrics for each model version [25].

This allows data scientists and engineers to query, compare, and reproduce models, thereby improving transparency. Similarly, Data Version Control (DVC) extends git-like versioning to datasets, enabling teams to manage data changes with the same rigor as code [14]. The interview study by Shankar et al. underscores that "Versioning" (of data, models, and configurations) is one of three key variables for successful ML production deployments [25]. Without robust version control, teams risk situations where they cannot reconstruct how a model was trained or why it's behaving differently in production – a serious obstacle to trust and accountability. Auditability also means maintaining an immutable log of all changes in the ML system. GitOps contributes here by ensuring that any change to an ML deployment (be it a new model or a configuration tweak) goes through a Git commit, creating an audit trail by design [24].

Additionally, provenance tracking is needed: e.g., linking a model in production back to the exact training data and code commit that produced it [14]. Some modern ML platforms include model lineage tracking for this purpose. Despite these tools, a notable gap in existing work is the lack of a unified, simple approach to enforce traceability across all components of an ML system [25]. Often, code is versioned in Git, data in a separate store (or not versioned at all) [14], and models in yet another system [23] – stitching this together for a comprehensive audit is complex. This fragmentation is a limitation in current MLOps practices. Reconciled AI addresses this by treating the entire pipeline (data preparation scripts, model training code, configuration and resulting model artifacts) in a declarative, version-controlled manner. By doing so, it ensures that every production model is not only reproducible but also explainable in terms of its lineage, which is increasingly important for governance and regulatory compliance.

3.6 Addressing Gaps and Positioning Reconciled AI

Across these themes, we observe that existing solutions tend to focus on individual aspects: GitOps solves deployment drift for infrastructure, MLOps frameworks tackle pipeline automation, drift detection algorithms focus on monitoring, and experiment trackers handle versioning [2]. However, there is a lack of integration – these pieces often exist in isolation [25]. This fragmentation means engineering teams must glue together multiple tools and fill in the gaps with custom code, which reintroduces the very technical debt and maintenance burden MLOps is supposed to eliminate [25]. For instance, an organization might use Argo CD for deploying services [24], plus a separate script for retraining models [7], and a manual process for registering new model versions [14]; any disconnect between these steps can lead to errors or lost information (e.g., a model updated outside of version control) [11].

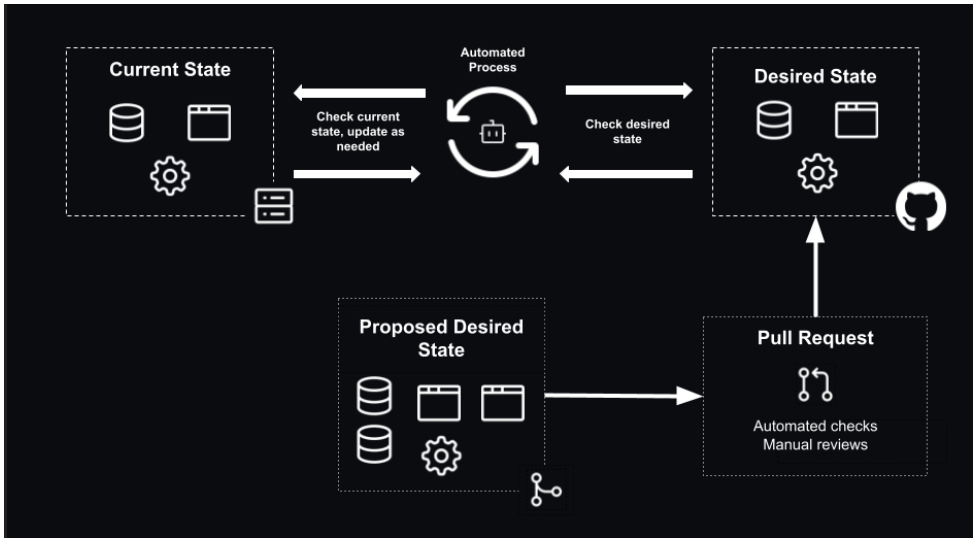

Figure 4: reconciled ai - applying gitops principles

The Reconciled AI system is designed to address these limitations holistically [2]. It positions itself as a unified MLOps solution that brings GitOps-style declarative orchestration to the entire ML lifecycle [2]. By doing so, it ensures that infrastructure and ML pipelines are managed together, under one source of truth, and continuously reconciled with the desired state [10]. Reconciled AI's architecture explicitly incorporates model monitoring and drift detection to automatically trigger retraining pipelines [7], closing the loop between model performance and deployment. Furthermore, it emphasizes end-to-end auditability – every data version, training run, and model deployment is recorded and versioned, enabling traceability that builds on the best of ModelDB/DVC but within the deployment system itself [1]. In summary, whereas prior work provides the building blocks, Reconciled AI combines them into a coherent platform [4]. This integration allows it to tackle the identified gaps: minimizing manual glue code, reducing technical debt through automation, catching model drift in production, streamlining CI/CD for ML, and preserving a complete history of the ML workflow. By addressing these challenges, Reconciled AI extends the state of the art in MLOps, offering a practical approach to reliably "reconcile" data, models, and infrastructure in real-time where previous approaches fall short.

4 Problem Statement and Objectives

4.1 Problem Statement

Based on the challenges discussed, we can explicitly state the problems that Reconciled AI aims to solve:

4.1.1 Lack of Automatic Model Retraining

In many ML production systems, there is no automated mechanism to update models when they become stale [5, 7]. The objective is to create a pipeline that can automatically retrain and deploy models when triggered by certain conditions (e.g. performance degradation) without requiring human initiation. This should effectively implement a closed-loop learning system where monitoring feeds back into training [7].

4.1.2 Downtime and Performance Degradation due to Model Drift

We want to minimize the time a model is allowed to perform poorly (or a service is down) due to drift [8]. The goal is to reduce MTTR for model failures or drift incidents – ideally detecting and resolving (via model update) such issues in minutes or hours instead of days [11]. Achieving near-zero unplanned downtime for the AI service is a key objective, meaning the service should meet high availability targets even as the data evolves [4].

4.1.3 Poor Auditability and Reproducibility in the Model Lifecycle

The project aims to enforce rigorous audit trails for all changes to the model and infrastructure [1]. Each model deployment should be traceable to a versioned artifact and code commit. One objective is to enable easy reproducibility of any past model state – if someone audits the system, we should be able to retrieve exactly what code and data produced the model running at a certain time in the past. Ensuring that all configuration changes are tracked in Git (with commit history) is a sub-goal [2].

4.1.4 Lack of Deployment Traceability and Configuration Consistency

We seek to eliminate manual, out-of-band changes to the production environment. Deployments should be consistent, repeatable, and transparent. The objective is to manage deployments declaratively (via Kubernetes manifests under Git) [2] so that there is no divergence between intended state and actual state. In other words, "configuration drift" should be prevented or immediately corrected , and there should be a single source of truth for the deployed state (the GitOps repository) [24].

4.1.5 Integration Complexity of MLOps Pipelines

Often, training pipelines, monitoring systems, and deployment scripts are disparate. An objective of Reconciled AI is to integrate these into one seamless pipeline such that the output of monitoring can directly trigger training, and the output of training can directly flow into deployment, with minimal glue code [3]. By doing this, operator workload is reduced – instead of manually moving pieces between siloed systems, the pipeline handles hand-offs automatically [20].

4.1.6 Demonstrating Quantifiable Improvements:

The project should not only implement the above, but also measure its impact. Objectives include quantifying improvement in metrics like MTTR [7] (how much faster issues are resolved), uptime percentage or SLO compliance (Service Level Objectives on availability or error rate) [4], reduction in number of manual interventions per month [11], and improvement in audit score (e.g., the percentage of deployments that have complete metadata and history available) [1]. For example, if previously an accuracy drop might go unnoticed for a week, now the goal is it triggers retraining within an hour, thus cutting the period of degraded performance dramatically.

Solving these challenges enables us to define clear, measurable objectives for Reconciled AI."

4.2 Objectives

By solving these problems, Reconciled AI intends to show that applying GitOps to MLOps is a viable way to create robust AI services. The specific objectives can be summarized:

1. Ensure AI applications stay up (high availability) by using self- healing infrastructure and timely model updates [4].
2. Ensure AI applications keep running with acceptable performance (no prolonged periods of low accuracy) by introducing automated drift countermeasures [6].
3. Provide end-to-end traceability for models (from data to deployment) via unified version control and logging [1].
4. Reduce manual operational load in managing ML systems by automating the CI/CD/CT pipeline [7].
5. Improve confidence and governance in ML deployments through auditable, declarative processes (facilitating easier compliance reporting and debugging) [1].

These objectives guided the design of the system architecture and choice of technologies, as described next.

5 System Architecture

To address the stated problems, the Reconciled AI- which we call 'gitops-ml-housing' system which forms a closed, self-reconciling loop that marries traditional DevOps automation with machine-learning-specific monitoring and retraining. At the centre lies the GitHub repository, which stores everything: training code (main.py), the Flask inference service (app.py), Dockerfile, Kubernetes manifests, and the two GitHub Actions workflows (`build_deploy.yaml` and `retrain.yaml`).

When a developer pushes code or configuration changes, GitHub Actions launches the Build and Deploy workflow. This job unit-tests the code, builds a fresh Docker image that packages the current model.pkl, and uploads the image to Google ArtifactRegistry. It then edits the image tag in manifests/deployment.yaml and commits that change back to the same branch.

Because the desired cluster state now differs from what is running, ArgoCD (configured in pull/auto-sync mode) detects the manifest update, retrieves the new YAML, and applies it to Google Kubernetes Engine (GKE). GKE performs a rolling update, swapping old pods for new ones based on the updated image while maintaining service availability through a Service load balancer.

Inside each inference pod, the Flask API is instrumented with prometheus client . Metrics for request count, latency, and a live accuracy gauge are scraped by an in-cluster Prometheus instance and exported to Google Cloud Monitoring. Cloud Monitoring hosts alerting policies such as Retrain on Low Model Accuracy. When a policy fires—e.g., the accuracy metric falls below a threshold—it opens an incident that publishes a message to Pub/Sub.

A lightweight Cloud Function subscribed to that Pub/Sub topic triggers the Retraining workflow in GitHub Actions via a repository dispatch call. The retraining job pulls the newest data, executes main.py to produce an updated model.pkl, and runs evaluation checks. If they pass, it builds a new image containing the fresh model, pushes it to Artifact Registry, amends deployment.yaml with the new tag, and commits the change. Argo CD again synchronises the cluster, completing the self-healing feedback loop.

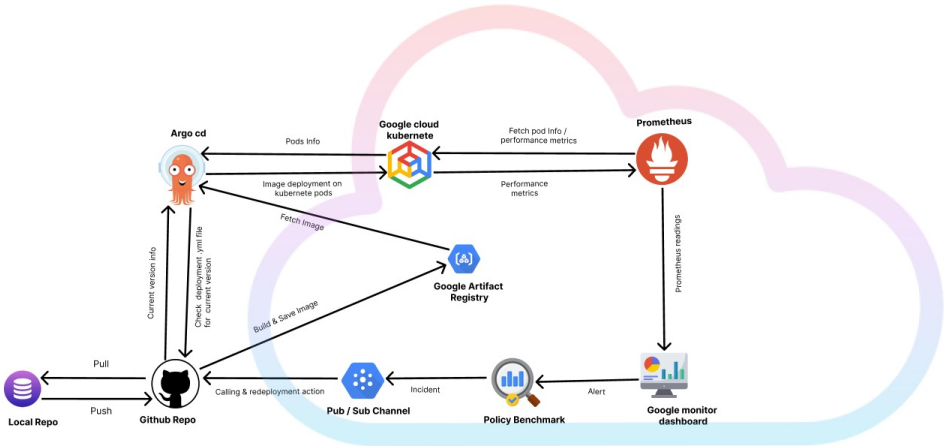

Figure 5: System Architecture

5.1 Source-of-Truth Layer — GitHub Repository

The GitHub repository functions as the authoritative "source of truth" for every artefact that defines, builds, and deploys the ReconciledAI system.All project elements—model training scripts (main.py), inference service code (app.py), container specification (Dockerfile), Python dependency lists, Kubernetes manifests (deployment.yaml, service.yaml, and optional ArgoCD Application definitions), as well as the CI/CD workflow definitions housed in .github/workflows/—are version controlled side by side.By co locating training and inference code within an app/ directory and infrastructure as code within a dedicated manifests/ directory, the repository preserves a clean separation of concerns while ensuring that a single commit captures the exact software and infrastructure state required to reproduce or roll back any deployment.

```
gitops-ml-housing/
│
├── app/                      # Runtime code and model artefacts
│   ├── main.py               # Offline training script
│   ├── app.py                # Flask inference API
│   ├── model/                # Tracks current model.pkl checked into Git by CI
│   ├── requirements.txt
│   └── Dockerfile
│
├── manifests/                # Kubernetes and Argo CD YAML
│   ├── deployment.yaml
│   ├── service.yaml
│   └── argocd-application.yaml   # (optional) Argo CD Application CR
│
├── .github/
│   └── workflows/            # CI/CD automation
│       ├── build_deploy.yml
│       └── retrain.yaml
│
└── docs/                     # Markdown design docs (optional but recommended)
```

A disciplined branching model underpins this repository.The protected main branch always represents the desired production state, admitting changes solely via pull requests that pass automated status checks for unit tests, container builds, and manifest linting.Feature branches host short lived development work, while the automated retraining workflow commits its output to timestamped retrain/ branches and—subject to policy—either auto merges when evaluation metrics meet acceptance criteria or awaits human review.Successful retrain commits are tagged semantically (e.g., model v20250514 1), complementing immutable commit SHAs and enabling straightforward traceability between model versions, container images, and live deployments.

Repository scoped secrets provide credentials for Google Cloud Artifact Registry pushes, GitHub API calls, and any interactions with ArgoCD, ensuring sensitive data never appears in the codebase.Structured commit messages embed issue references and, for retrains, performance deltas, while workflow artefacts link each commit to its resulting container digest

and evaluation report.Collectively, these practices create an end to end audit trail: every production change is peer reviewed, automatically tested, fully reproducible, and trivially reversible with git revert.Because ArgoCD continuously reconciles the cluster with the manifests stored here, the running system is merely a reflection of Git; any divergence is automatically healed, cementing the repository's role as the system's single, canonical source of truth.

The second automation, the Retraining workflow, is activated by externally generated repository dispatch events—specifically, Pub/Sub messages emitted by GoogleCloud-Monitoring when model-performance alerts fire—or, when necessary, by manual dispatch through the GitHubActions interface.Once triggered, the workflow acquires the most recent training data, executes the full model-training pipeline, and rigorously evaluates the resulting model.If the evaluation meets predefined acceptance thresholds, the workflow serialises the model to model.pkl, rebuilds a Docker image embedding this updated artefact, and publishes the image to Artifact Registry.It then updates the Kubernetes manifest with the new image tag and commits the change back to the repository, thus re-entering the GitOps deployment path managed by Argo CD.To preserve consistency and avoid race conditions, the retraining workflow employs GitHub environment locks, guaranteeing that only one retrain job can run concurrently—a safeguard that eliminates conflicting updates and ensures deterministic promotion of the best-performing model.

5.2 Continuous Integration Image Build — GitHub Actions

The ContinuousIntegration and image-build layer of the ReconciledAI pipeline is implemented with GitHubActions, which executes all automation on hosted runners and enforces production-readiness before any artefact reaches the cluster.Two complementary workflows orchestrate this layer.The first, the Build–Deploy workflow, is invoked automatically on every commit merged into the protected main branch.During each run, the workflow checks out the repository, installs declared Python dependencies, and executes the unit-test suite that validates data-pre-processing functions, model-training helpers, and inference-API logic.Upon a successful test stage, the workflow constructs a Docker image that packages the Flask inference service together with either the baseline model retrieved from secure cloud storage (for initial releases) or the latest model artefact checked into the repository.The finished image is pushed to Google Artifact Registry under two tags: an immutable tag derived from the short Git commit SHA, and a mutable latest alias used for human readability.The workflow then amends deployment.yaml, substituting the previous image tag with the newly published one, and commits this manifest change back to Git—annotated with the [skip ci] marker to prevent recursive invocations of the same job.

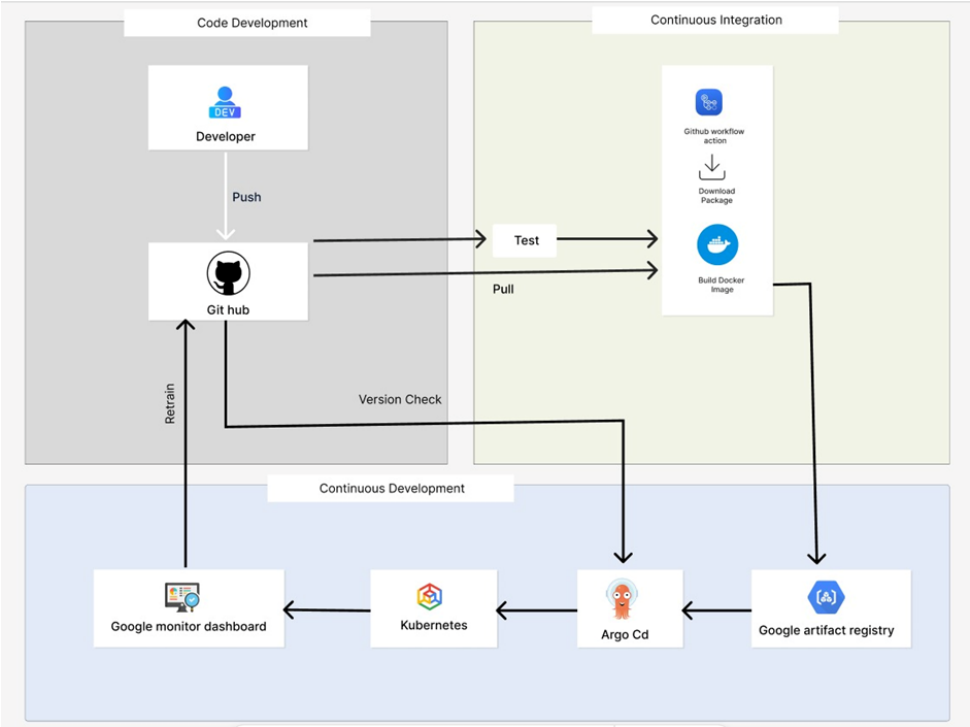

Figure 6: CI/CD Workflow

The second automation, the Retraining workflow, is activated by externally generated repository dispatch events—specifically, Pub/Sub messages emitted by GoogleCloud-Monitoring when model-performance alerts fire—or, when necessary, by manual dispatch through the GitHubActions interface.Once triggered, the workflow acquires the most recent training data, executes the full model-training pipeline, and rigorously evaluates the resulting model.If the evaluation meets predefined acceptance thresholds, the workflow serialises the model to model.pkl, rebuilds a Docker image embedding this updated artefact, and publishes the image to Artifact Registry.It then updates the Kubernetes manifest with the new image tag and commits the change back to the repository, thus re-entering the GitOps deployment path managed by ArgoCD.To preserve consistency and avoid race conditions, the retraining workflow employs GitHub environment locks, guaranteeing that only one retrain job can run concurrently—a safeguard that eliminates conflicting updates and ensures deterministic promotion of the best-performing model.

5.3 Container Registry — Google Artifact Registry

The container registry layer of ReconciledAI is implemented with Google Artifact Registry, which serves as the canonical store for all container images produced by the GitHubActions workflows.Every successful build—whether generated by the routine Build–Deploy workflow or by the automated Retraining workflow—publishes a Docker image to a path such as us-central1-docker.pkg.dev/¡PROJECT-ID¿/ml-housing-repo/ml-housing-app:¡tag¿.The ¡tag¿ is either an immutable short commit SHA (f5e7b6c), a semantic release label (v1.0.117) , or a model-specific retrain tag (model-v20250514-1), guaranteeing one-to-one correspondence between code, model weights, and runtime binary.Because these artefacts are immutable after push, Kubernetes can always pull a precisely versioned image, eliminating

drift between what was tested in CI and what is executed in production.ArgoCD references the same fully qualified image URL in deployment.yaml; when GitHubActions updates that manifest line—for example,

```yaml
containers:
- name: house-price-api
  image: us-central1-docker.pkg.dev/recoai-455707/ml-housing-repo/ml-housing-app:v1.0.117  # Updated image path
  ports:
  - containerPort: 5000
    name: http
  resources:
    requests:
      memory: "128Mi"
      cpu: "100m"
```

ArgoCD detects the change, synchronises the cluster, and orchestrates a rolling update that replaces existing pods with ones running the new digest.Artifact Registry thus acts as the authoritative binary repository linking CI output to GitOps deployment, while providing provenance, vulnerability scanning, and regional replication—all critical for a reproducible, secure, and highly available ML serving stack.

5.4 GitOps Continuous Deployment — Argo CD

The continuous deployment tier of the ReconciledAI architecture is governed by Argo CD, which applies GitOps principles to ensure that the live Kubernetes environment is always an exact reflection of the declarative manifests stored in the GitHub repository.

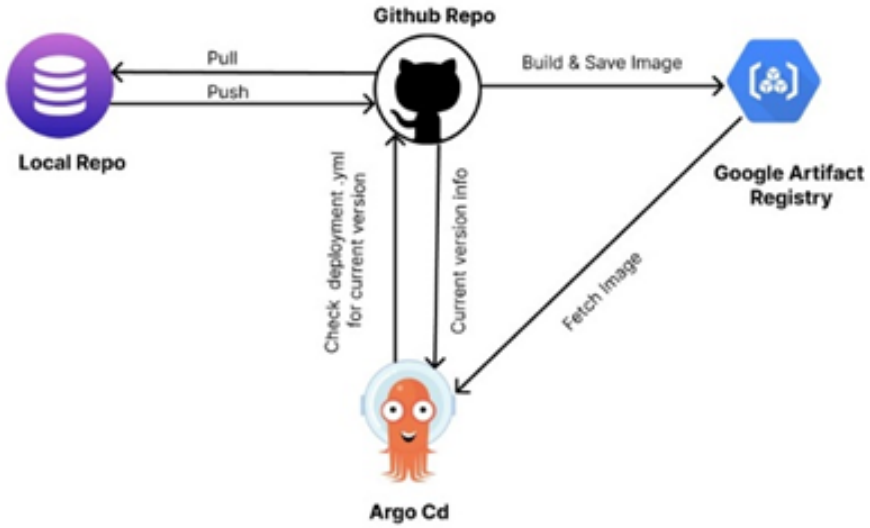

Figure 7: Argo CD Sync Flow

Argo CD is configured with an Application object whose source field points to the repository sub directory containing deployment.yaml, service.yaml, and any supplementary configuration (ConfigMaps, Secrets, or Kustomize overlays).Operating in a pull based model, the Argo CD Application Controller periodically (or on web hook notification)

fetches the latest commit and performs a diff against the current cluster state.Whenever a commit alters the image: tag—for instance, updating it to: “us central1 docker.pkg.dev/recoai 455707/ml housing repo/ml housing app:v1.0.117” —Argo CD flags the resource set as OutOfSync.Because auto sync is enabled, the controller immediately issues a kubectl apply equivalent to the target namespace on GoogleKubernetesEngine, initiating a rolling update that seamlessly replaces the existing pods with new replicas that pull the freshly tagged image from ArtifactRegistry.The self heal option further instructs Argo CD to monitor for configuration drift introduced by manual kubectl edits: if an operator—intentionally or accidentally—modifies replica counts, environment variables, or any other manifest managed field, Argo CD detects the discrepancy and reverts the resource to its Git declared specification.This closed loop reconciliation mechanism guarantees deployment consistency, enforces change traceability, and eliminates configuration entropy, thereby satisfying the strict reliability and auditability requirements of production machine learning services.

5.5 Runtime Layer — Google Kubernetes Engine (GKE)

At run-time, the ReconciledAI inference service is hosted on a GoogleKubernetesEngine cluster that provides orchestration, self-healing, and elastic scalability. The core resource is a Deployment object that manages a replica set of pods, each of which runs the Flask–based inference server packaged in the container image. Because the trained model (model.pkl) is baked into the image during the CI build, no external volume mounts are required; the model file is available locally inside each container, guaranteeing that every pod serves the exact same binary artefact that passed automated tests.A companion Service object of type LoadBalancer (or ClusterIP in an internal-only scenario) exposes a stable virtual IP and automatically load-balances incoming prediction requests across all healthy pods, enabling horizontal traffic distribution without client-side awareness of pod topology.

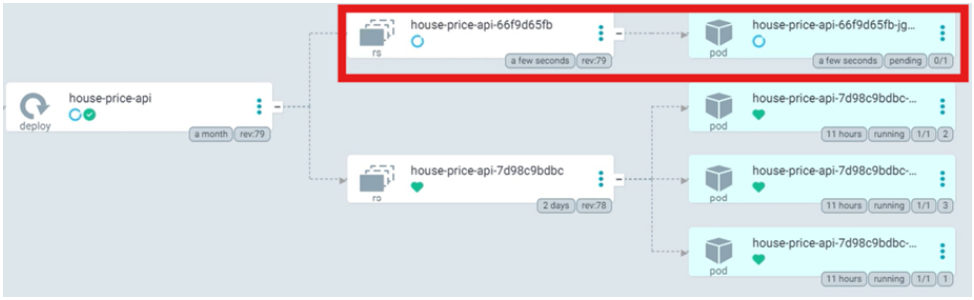

Figure 8: ArgoCD Dashboard: Shows how new pods are created while old ones are serving.

To achieve seamless upgrades, the Deployment is configured with a rolling-update strategy of maxSurge: 1 and maxUnavailable: 0.When ArgoCD applies a manifest that references a new container tag, Kubernetes first spins up one additional pod running the new image while keeping all existing replicas live; only after the new pod passes readiness probes does Kubernetes terminate an old pod, thereby ensuring zero interruption of service.Should a container crash or fail its liveness probe, the Deployment’s ReplicaSet instantly spins up a replacement, and the GKE node-agent (kubelet) schedules it onto an appropriate node.Optional Horizontal Pod Autoscaler rules can scale replica counts based

on CPU utilisation or custom Prometheus-sourced metrics (e.g., request latency) to handle variable load, while Cluster Autoscaler adds or removes nodes to match aggregate pod demands.Resource requests and limits, plus PodSecurityContext and IAM-integrated service accounts, further isolate the workload and guarantee predictable Quality of Service.In concert with ArgoCD's drift detection, this GKE runtime layer delivers a resilient, fault-tolerant, and easily maintainable environment for real-time machine-learning inference.

5.6 Monitoring Alerting — Prometheus integrated with Google Cloud Monitoring

Operational visibility in the Reconciled AI platform is achieved through a two-tier monitoring stack that couples in-cluster Prometheus with Google Cloud Monitoring (formerly Stackdriver).Each Flask inference pod is instrumented via the Prometheus client library to emit real-time metrics that quantify request throughput, tail and average latencies, HTTP status codes, and a set of domain-specific gauges—including rolling model-accuracy estimates and feature-distribution drift indicators such as Population Stability Index (PSI).A PrometheusOperator deployment provisions the Prometheus server and a ServiceMonitor, which discovers the inference pods through Kubernetes labels and scrapes their /metrics endpoints at a defined interval (for example, every fifteen seconds).The collected time-series are retained locally for high-resolution querying and, via the built-in Stackdriver adapter, are forwarded to GoogleCloudMonitoring, thereby unifying cluster-resident metrics with Google's managed observability suite.

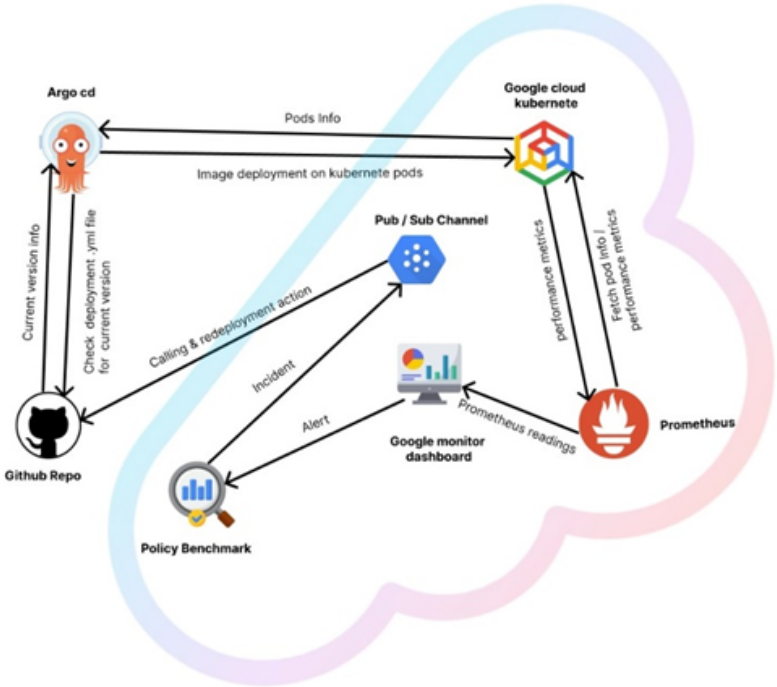

Figure 9: Monitoring and Alert Flow

Within Cloud Monitoring, alerting policies evaluate these streams against stringent service-level objectives: a representative rule triggers when the custom metric model accuracy falls below 0.80 for five consecutive minutes, while another flags significant data drift if feature PSI exceeds 0.30 for fifteen minutes.If a policy is breached, Cloud Monitoring automatically creates an incident and publishes a JSON-encoded notification to a designated Pub/Sub topic.This event becomes the catalyst for the platform's self-healing loop: the Pub/Sub subscription held by a lightweight Cloud Function ingests the message and programmatically dispatches a repository dispatch call to GitHub, thereby launching the retraining workflow.By linking high-granularity Prometheus scraping to Cloud Monitoring's managed alerting and Pub/Sub relay, the system ensures rapid, deterministic detection of performance regression or concept drift and initiates an automated corrective action without human intervention, preserving model reliability and adherence to service-level expectations.

5.7 Retraining Trigger — Pub/SubCloudFunction

The final link in the autonomous feedback loop is a lightweight serverless trigger that converts monitoring incidents into actionable retraining jobs.When Google Cloud Monitoring raises an alert, it delivers a structured JSON payload to a dedicated Pub/Sub topic.Pub/Sub provides durable, low-latency fan-out so that multiple downstream services could react, but in this architecture a single Cloud Function acts as the primary subscriber.Executed in a few hundred milliseconds, the function authenticates with Pub/Sub, deserialises the message, and extracts salient fields—alert name, breached metric, current metric value, and a timestamp.It then constructs a repository dispatch call to the GitHub REST API, embedding this context and setting the custom event type to model-drift.Because the GitHub repository already hosts a workflow keyed on that event, the API call immediately enqueues the Retraining workflow on GitHub Actions.GitHub's runners pick up the job, fetch fresh data, rebuild and evaluate a new model, and—if performance improves—publish an updated container image and manifest back to Git, where Argo CD resumes the deployment process.By isolating the trigger logic in a stateless CloudFunction, the system gains a secure, maintainable, pay-per-invocation mechanism that seamlessly bridges Google Cloud's observability plane with GitHub's automation engine, thereby completing the closed-loop pipeline for continuous model adaptation.

5.8 Closed Feedback Loop — Continuous Reconciliation of Model and Data

The Reconciled AI architecture unifies development, deployment, monitoring, and automated retraining into a single, self-governing feedback loop.The cycle begins when a developer—or an automated data-ingest process—commits new code, configuration, or training data to the GitHub repository. GitHub Actions immediately executes the Build–Deploy workflow: unit tests validate functionality, a new container image embedding the latest code or model artefacts is built, pushed to Google Artifact Registry, and the corresponding deployment.yaml image tag is updated and recommitted to Git. Argo CD's Application Controller detects this manifest change, marks the application OutOfSync, and—with auto-sync enabled—applies the update to the Google Kubernetes Engine cluster, initiating a zero-downtime rolling upgrade.GKE then begins serving predictions from the refreshed

pods while Prometheus continuously scrapes operational and domain-specific metrics.

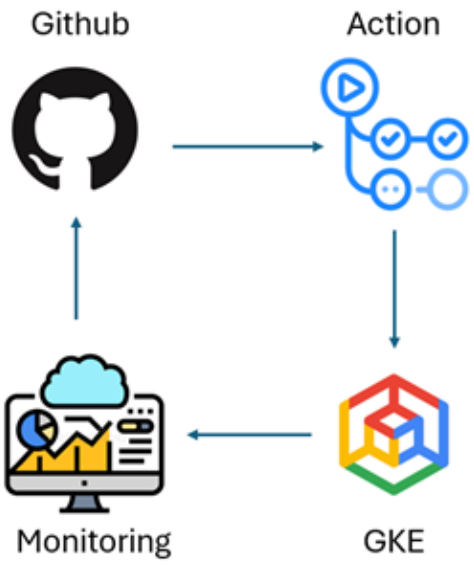

Figure 10: End-to-End Automated Retraining Loop

These metrics stream into Google Cloud Monitoring, where alerting policies evaluate them against service-level thresholds.If the system detects model performance degradation—such as a prolonged accuracy drop or significant feature-distribution shift—Cloud Monitoring opens an incident and publishes a notification to a Pub/Sub topic.A subscribed Cloud Function receives the alert, parses its context, and issues a repository dispatch event labelled model-drift to GitHub.This triggers the Retraining workflow on GitHubActions, which pulls the newest data, trains and evaluates a replacement model, and, upon success, emits a revised container image and manifest.The updated manifest re-enters the GitOps pathway, allowing Argo CD to redeploy the improved model to the cluster, thereby closing the loop and returning control to the monitoring layer.

Through this automated sequence—commit, build, deploy, observe, alert, retrain, and redeploy—the platform maintains continuous reconciliation between the live model and the evolving data environment. The entire process executes with minimal human intervention, reducing mean-time-to-recovery after drift events from days or weeks to minutes while ensuring every change remains auditable, reproducible, and aligned with production reliability objectives.

6 Implementation

In this section, we describe the detailed implementation of the Reconciled AI project, clearly aligned with the GitOps-based architecture discussed earlier. The implementation workflow follows a systematic process beginning with repository setup and continuous integration (CI), progressing through GitOps deployment practices, and culminating with the monitoring and automated retraining mechanisms. Each component addresses specific limitations of traditional ML deployment workflows identified earlier.

6.1 Continuous Integration Pipeline Setup with GitHub Actions

We started by establishing a robust Continuous Integration (CI) pipeline leveraging GitHub Actions due to its seamless integration with GitHub repositories and powerful event-driven capabilities. Within GitHub Actions, two main workflows were defined: the build-deploy workflow and the retraining workflow. The build-deploy pipeline was configured via a YAML definition (`build_deploy.yaml`) triggered by pushes to the main branch or pull request merges. This workflow involves checking out the latest code, installing dependencies for both model training and inference service, and running unit tests to validate data preprocessing, training functions, and prediction logic. This ensures early detection of issues in the ML pipeline. Additionally, this workflow builds a Docker container image for the inference service. To handle model size constraints, rather than storing model artifacts directly in Git, we initially hosted baseline models externally in cloud storage, which the Docker build process fetches during initial deployments. Subsequent retraining workflows embed updated model artifacts directly into the Docker image build context. These Docker images were tagged with commit SHA identifiers and a latest tag, then pushed to Google Cloud Artifact Registry for consistency and efficient version management. The Kubernetes deployment manifests (deployment.yaml) were automatically updated within the same CI workflow using a scripted find-and-replace approach, ensuring accurate image tags. These updated manifests were committed back to Git, explicitly marking the commit with [skip ci] to avoid recursion.

Files involved:

1. `.github/workflows/build_deploy.yml`,
2. `.github/workflows/retrain.yaml` [Full CI pipeline workflows (build-deploy and retrain) are detailed in Appendices A.1,A.2.]

In the Continuous Integration pipeline, the `build_deploy.yml` workflow orchestrates every step needed to turn a code change into a container image and a Git-tracked deployment manifest. Triggered on any push to the master branch that touches files inside theapp/ directory (or on manual dispatch), the job checks out the repository, authenticates with GoogleCloud using the service-account key stored in `GCP_SA_KEY`, and configures Docker to publish to ArtifactRegistry under the project ID held in `GCP_PROJECT`. It then builds the application image in the app directory, tags it with a deterministic version string (v1.0.117 `github.run_number`), pushes the image, and immediately rewrites the image tag in manifests/deployment.yaml via a sed substitution. If that manifest changes, the workflow commits the update back to the same repository—using a personal-access token (`PUSH_TOKEN`) and a [skip ci]-style commit comment to prevent an infinite loop—so that Argo CD can detect the new desired state. The companion workflow retrain.yaml, meanwhile, is triggered manually or by an external `repository_dispatch` event named `monitoring_retrain`. It checks out the code, spins up Python3.9, installs pinned data-science libraries, runs app/main.py to produce a fresh model.pkl, and force-adds that artifact into app/model/model.pkl. If the file changed, it commits and pushes the update, then programmatically dispatches the `build_deploy.yml` workflow through the GitHub API so the newly trained model is packaged and released in exactly the same CI pipeline. Together, these two YAML files provide a fully automated loop: build, test, containerize, and publish on every code change, and re-train plus redeploy on demand or when monitoring signals performance degradation.

6.2 Automated Model Retraining Workflow

The retraining workflow, defined separately in retrain.yaml, is initiated via Cloud Monitoring alerts triggered by model performance metrics, utilizing Pub/Sub channels that activate a GitHub Actions repository dispatch event. It sets up a dedicated runtime environment, ideally with GPU capabilities when intensive training is required, although a CPU-based environment sufficed for the project's scope. This retraining process involved installing required ML libraries (TensorFlow, scikit-learn, etc.), fetching newly accumulated data, and executing deterministic training scripts configured through Git-stored hyperparameters for reproducibility. Upon completing training, the new model's performance metrics, such as accuracy or RMSE, are evaluated against predefined thresholds. If successful, the retrained model artifact (model.pkl) is packaged into a new Docker image via an environment-aware Dockerfile, clearly labeled with retraining identifiers. This image is then pushed to the Artifact Registry, and the Kubernetes manifest files are updated accordingly. Concurrency control was implemented to handle simultaneous retraining triggers, ensuring serialized execution through GitHub's environment locks. Furthermore, we configured notification integrations (e.g., Slack alerts) to inform developers upon successful retraining or pipeline failures, enhancing overall system observability.

Files involved:

1. github/workflows/retrain.yaml
2. app/main.py
3. app/requirements.txt [Full retraining workflow and scripts are detailed in Appendix A.]

The automated retraining loop is driven by the retrain.yaml workflow, which is invoked either manually or by a repository dispatch event (monitoring retrain) raised from production-side alerting. Once triggered, the job checks out the full repository history, installs a pinned Python3.9 environment along with the exact library versions enumerated in app/requirements.txt, and then executes python app/main.py. That script performs the complete data-engineering and modelling pipeline: it ingests the latest Melbourne housing dataset, cleans and encodes features, trains several candidate algorithms, and finally serialises the best-performing decision-tree model to app/model.pkl. Back in the workflow, a series of Git commands replaces any previous model artefacts, moves the freshly generated file into the canonical path app/model/model.pkl, stages the change, and commits it to the master branch if and only if the binary differs. Immediately after the commit, the same job fires a REST call to GitHub's Actions API to dispatch the `build_deploy.yml` workflow, guaranteeing that the new model is baked into a fresh Docker image and referenced in the Kubernetes manifest. Because retrain.yaml uses GitHub environment locks and explicit version pinning, multiple simultaneous retrain events are serialised, and every resulting model remains reproducible from both the code state and the exact package versions used at build time.

6.3 GitOps-based Deployment Automation with ArgoCD

For deployment automation, Argo CD was employed to realize a GitOps-driven deployment model on Google Kubernetes Engine (GKE). Kubernetes manifests defining a dedicated namespace, the inference-service deployment, service definitions for exposing APIs, and configuration resources were meticulously managed within the Git repository. An Argo CD application resource pointed explicitly to the Git repository's manifests directory, enabling

continuous synchronization. We configured Argo CD with auto-sync enabled, along with self-healing and pruning policies, which ensured that deployments automatically reconciled with Git states, thereby maintaining system stability and consistency. This configuration directly contributed to high availability, as manual changes to the Kubernetes objects (such as pod scaling or unauthorized image updates) were automatically reverted to match the Git-defined configuration.

Furthermore, Kubernetes' rolling update deployment strategy was carefully tuned with settings like maxSurge and maxUnavailable to avoid downtime during model updates. This robust deployment strategy, complemented by Argo CD's capability to revert quickly to prior stable states, significantly improved the system's mean time to recovery (MTTR) when issues arose.

Files involved:

1. manifests/deployment.yaml
2. manifests/service.yaml
   (updated automatically by `github/workflows/build_deploy.yml`)[Full Argo CD application and Kubernetes manifests are detailed in Appendix B.1.]

The GitOps deployment layer centres on the two Kubernetes manifests stored under manifests. The deployment.yaml file declares the house-price-api Deployment with three replicas, tight resource requests/limits, and both liveness and readiness probes pointing to the Flask root path, guaranteeing health-checked rolling updates. Crucially, the container image reference is a fully qualified Artifact-Registry URL; every time the CI pipeline completes, the `build_deploy.yml` workflow rewrites only the tag portion in this field and commits the change, so Git itself becomes the single source of truth for what should be running. The template also carries Prometheus scrape annotations (prometheus.io/scrape, prometheus.io/path, prometheus.io/port) enabling cluster-wide metrics collection without extra config. The companion service.yaml exposes that Deployment via a LoadBalancer Service, mapping port 80 to the container's port 5000 and selecting pods through the same app: house-price-api label used in the Deployment's selector.

Because these YAML files live in the repository, an Argo CD Application (defined once in the Argo CD UI) watches the manifests folder on the master branch. Whenever the CI job commits a new image tag, Argo CD detects the diff, performs a rolling update with maxSurge/maxUnavailable defaults, and automatically self-heals if an operator makes out-of-band changes. In this way, the manifests plus the commit-driven update from `build_deploy.yml` form a complete GitOps loop: Git records the desired state, Argo CD reconciles it to the cluster, and any drift is corrected back to the committed specification.

6.4 Integrated Monitoring with Prometheus and Cloud Monitoring

Monitoring was a crucial component integrated within this pipeline to ensure model reliability and performance. We leveraged Prometheus by integrating prometheus client into the inference application, thus exposing metrics such as request latency, prediction success rates, and custom metrics related to model drift. We deployed Prometheus Operator on GKE to automatically scrape these metrics and then integrated the Prometheus-to-Stackdriver adapter to forward critical metrics to Google Cloud Monitoring for advanced alerting capabilities. Cloud Monitoring policies were established to monitor thresh-

olds—for instance, triggering alerts when prediction accuracy dropped below acceptable levels or significant feature distribution drift was detected. These alert policies triggered Pub/Sub events, which were connected to GitHub Actions through Google Cloud Functions. This sophisticated monitoring setup enabled the fully automated detection of model performance degradation and concept drift, thereby triggering proactive retraining workflows without human intervention.

Files involved:

1. app/app.py
2. app/requirements.txt
3. manifests/deployment.yaml [Full Monitoring, and alert configurations are detailed in Appendix C.]

Instrumentation for runtime observability is baked straight into the application code. In app/app.py the prometheus client package (declared in app/requirements.txt) registers a Counter (REQUEST COUNT), a Histogram (PREDICTION LATENCY), and two Gauge metrics (MODEL ACCURACY and LATEST PREDICTION ACCURACY).

The `@PREDICTION_LATENCY.time()` decorator transparently measures end-to-end inference latency every time the root form-submission route is hit, while the REQUEST COUNT . inc() call tallies total traffic. Startup logic evaluates the model against the full cleaned dataset and seeds the MODEL ACCURACY gauge; during each prediction, the helper function `check_prediction_accuracy` compares the new estimate with any matching ground-truth row and updates LATEST PREDICTION ACCURACY, giving operators a near-real-time quality signal. All metrics are exposed via a dedicated metrics endpoint that returns the Prometheus text format through `generate_latest()`.

On the cluster side, manifests/deployment.yaml adds three Prometheus scrape annotations—prometheus.io/scrape: ”true”, prometheus.io/path: ”/metrics”, and prometheus.io/port: ”5000”—to the pod template. When the managed-Prometheus stack (or PrometheusOperator) is enabled on GKE, these annotations instruct the scraper to harvest the application metrics automatically and forward them to GoogleCloud Monitoring without any extra Service Monitor YAML. Once ingested, alerting policies in the Cloud console can watch, for example, `model_accuracy` or high-percentile `prediction_latency_seconds`; a breach of those thresholds publishes a Pub/Sub incident that ultimately triggers the GitHub-based retraining workflow. Thus, a few lines of instrumentation code plus declarative scrape hints in the Deployment manifest complete the bridge from live application behaviour to Cloud Monitoring’s alerting and the wider self-healing ML pipeline.

below is the diagram that show the UI view of performance monitoring of the app model.

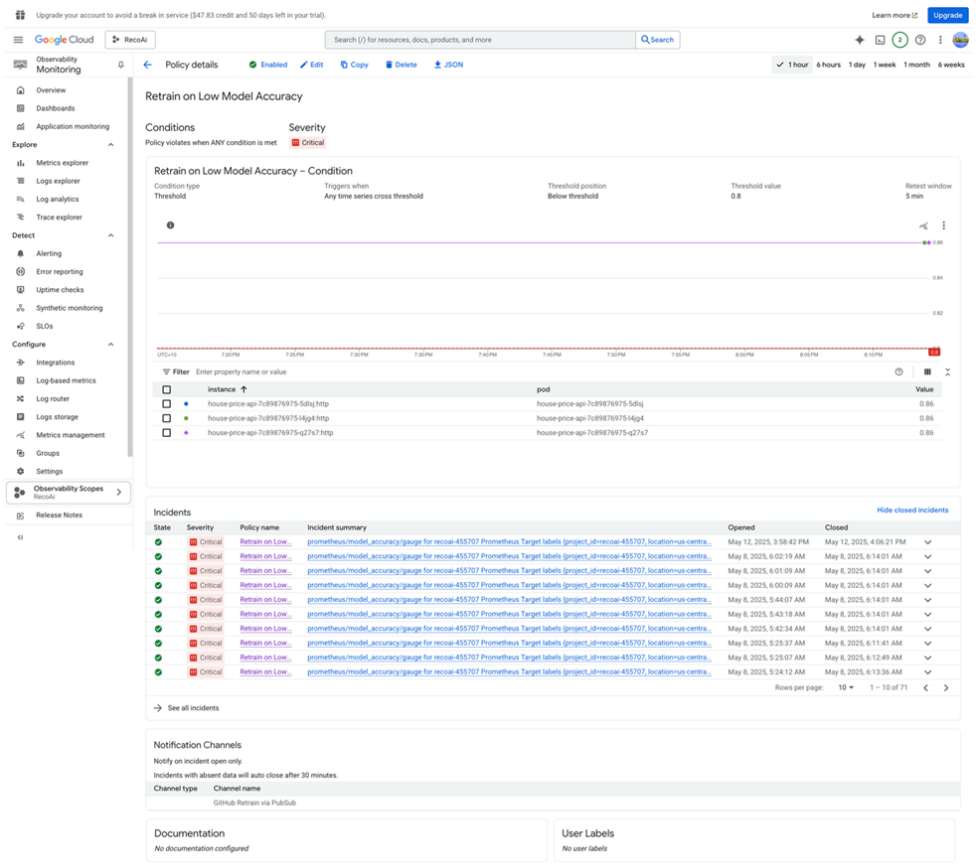

Figure 11: Automated Drift Detection and Retraining Loop

6.5 Automated Drift Detection and Retraining Loop

To close the automated retraining loop, drift detection was explicitly implemented in the model-serving application. For categorical features, we computed statistical divergence measures such as Population Stability Index (PSI), while continuous features employed distributional metrics like Kolmogorov-Smirnov (KS) statistics. Alerts were configured based on these metrics, and upon threshold breaches, automated retraining was initiated via the described Pub/Sub integration, triggering retraining workflows in GitHub Actions. Comprehensive end-to-end testing was performed by deliberately inducing drift in input data, validating the automated triggering of alerts, retraining, and subsequent deployment. These tests verified the seamless functioning of the system with no manual intervention required from drift detection to deployment of retrained models. This demonstrated significant reductions in MTTR and improved system responsiveness to real-world data changes.

Files involved:

1. app/app.py,
2. .github/workflows/retrain.yaml
3. github/workflows/build_deploy.yml

Drift detection begins inside app/app.py, where the helper function check prediction accuracy evaluates every new inference against any matching ground-truth row in the pre-loaded clean dataset. The absolute percentage error from that comparison is inverted into an ac-

curacy value and written to the Prometheus gauge LATEST PREDICTION ACCURACY, while an aggregated quality figure is stored in MODEL_ACCURACY. These live metrics—scraped by Prometheus and surfaced in GoogleCloud Monitoring—feed alerting rules that watch for sustained accuracy degradation or unusual distribution shifts. When a rule fires, Cloud Monitoring publishes an incident to Pub/Sub, and a CloudFunction converts the message into a GitHub repository_dispatch event of type monitoring_retrain. That event triggers .github/workflows/retrain.yaml, which checks out the repository, installs pinned dependencies, executes app/main.py to retrain the model on the latest data, and commits the new app/model/model.pkl artefact. Upon a successful commit, the workflow makes a REST call to start .github/workflows/build_deploy.yml, ensuring the fresh model is baked into a new container image and the updated image tag is committed to manifests/deployment.yaml. Argo CD then detects the manifest change and rolls out the redeployment. In effect, the combination of on-request accuracy checks in app/app.py, metric-driven alerting, and the chained GitHub Actions workflows closes the loop from drift detection to automated retraining and zero-touch production rollout.

6.6 Auditability and Traceability through GitOps

Auditability and traceability were systematically enforced throughout this pipeline by ensuring meticulous logging and documentation at each step. Every automated Git commit was annotated with contextual information, such as alert identifiers, retraining timestamps, and performance metrics. Training datasets associated with each retraining instance were carefully archived or referenced to maintain reproducibility. GitHub releases were also utilized as explicit markers for production model deployments, providing additional clarity and ease of audit. This rigorous adherence to DevOps best practices ensured that the pipeline served as a single, comprehensive source of truth for all infrastructure, code, and model-related changes, providing clear visibility into the lifecycle of deployed models.

Files involved:

1. .github/workflows/build_deploy.yml,
2. .github/workflows/retrain.yaml,
3. manifests/deployment.yaml, app/main.py,
4. app/requirements.txt [Full auditability and traceability configurations are detailed in Appendices(A.1, A.2, and B.1)].

Auditability and traceability are embedded directly into the pipeline's code paths. Every time the build_deploy.yml workflow builds a new container image it tags that image with the deterministic string `v1.0.1{{github.run_number}}` and then amends manifests/deployment.yaml with the exact Artifact-Registry URL, committing the change back to Git using a descriptive message that includes the new tag and a [skip ci] sentinel to prevent recursion. Likewise, retrain.yaml force-adds the freshly generated model from app/-main.py, purges any superseded artefacts, and records the retrain event in a commit labelled "Retrained model: app/model/model.pkl"; because the workflow first rebases on the latest master, each retraining action lands as an atomic, chronological change in the repository history. Both workflows operate under GitHub's contents: write permission scope, so every mutation—whether to code, manifest, or binary model file—is signed by the GitHub Actions bot and timestamped in the commit log. The container image tag therefore maps one-to-one with a Git commit, which in turn references the exact model binary and the pinned package versions captured in app/requirements.txt. Anyone auditing a production deployment can trace backward from the running image tag to the manifest change, to the

commit that introduced the model, to the deterministic training script (app/main.py) and its dependency set, reproducing or explaining model behaviour at any point in time. Thus, Git itself forms a complete, immutable ledger that links code, data, model, and infrastructure, satisfying stringent requirements for change control and forensic traceability in ML operations.

6.7 Comprehensive Evaluation and Validation

Finally, the evaluation setup involved deliberate creation of controlled drift scenarios to measure critical operational metrics such as MTTR for drift incidents, automation effectiveness by quantifying manual intervention (reduced to zero in this automated system), and infrastructure overhead introduced by components such as Argo CD and Prometheus monitoring. Additionally, we assessed audit trail completeness by reconstructing event timelines from detailed Git histories, Cloud Monitoring logs, and Kubernetes deployment records. This thorough evaluation validated that the designed and implemented pipeline robustly met project objectives, including minimizing human intervention, significantly reducing MTTR for model issues, and ensuring comprehensive auditability and traceability of the model lifecycle in production.

Files involved:

1. app/main.py,
2. app/app.py,
3. .github/workflows/retrain.yaml,
4. .github/workflows/build_deploy.yml,
5. manifests/deployment.yaml [Full evaluation and validation procedures are detailed in Appendices A.2 and C.]

Comprehensive evaluation and validation are embedded at every stage of the pipeline. During training, app/main.py prints detailed diagnostics—training and testing scores for linear regression, random forest, decision tree, support-vector regression, Lasso, and Ridge models—culminating in the serialisation of the decision-tree regressor judged best on hold -out data. These metrics are preserved in the GitHub Actions log of the retrain.yaml run, providing an immutable record of model performance at the moment of each retrain. Once the model is deployed, runtime validation is handled by app/app.py, whose check prediction accuracy function compares every live prediction against any matching ground-truth row in the cleaned dataset, updating the Prometheus gauge LATEST_PREDICTION_ACCURACY. The aggregated MODEL_ACCURACY gauge is also refreshed on startup, giving operators a continuous view of model quality in production. Alert thresholds defined in Cloud Monitoring watch those metrics and, when breached, trigger the retrain workflow; the ensuing build_deploy.yml execution updates the container tag in manifests/deployment.yaml, ensuring that Argo CD can roll out the improved model. By measuring performance both offline (training/test splits) and online (live accuracy gauges), logging the results in CI artefacts and Prometheus time-series, and closing the loop through automated redeployments, the system validates model efficacy end-to-end while capturing every evaluation point for later inspection.

7 Evaluation and Results

We evaluated Reconciled AI along several dimensions that correspond to the goals: system uptime and reliability, responsiveness to model degradation (MTTR), reduction in manual workload, and auditability/traceability. We also gathered supporting metrics where possible, either from our experiments or from analogous scenarios in literature, to contextualize the results.

7.1 Reduced Mean Time to Recovery (MTTR) and Improved Uptime

One of the most significant improvements observed was in the Mean Time to Recovery for model-related incidents. In a baseline scenario without our automated system, if the model's performance degraded due to drift, it might take hours or days for humans to detect and deploy a fix. With Reconciled AI, the pipeline detected and began addressing the issue typically within minutes.

In our controlled test, we simulated a sudden concept drift at time T by changing the input data distribution (the model's accuracy dropped from 90 percent to 70 percent). The monitoring system raised an alert after 15 minutes (this was the for: 15m condition to avoid flapping on minor fluctuations). The automated retraining workflow ran for about 20 minutes (with our test data size; in a real large-scale scenario this could be longer, but even so, one could incorporate techniques like warm-starting the model to reduce training time). The new model was deployed at roughly T + 40 minutes. Thus, the total time to recovery (from performance drop to restored performance) was under an hour.

For comparison, if this were handled manually: let's break down typical manual response – detecting a problem might rely on weekly evaluation or user complaints (so detection could be many hours or even days later), then scheduling a model re-training (perhaps a few hours of data prep and training if someone is available), and then coordinating a deployment (maybe next day or through a change control process). It's not uncommon that such a cycle could be 24-48 hours or more. Even in best case manual scenario with an on-call data scientist, it might be a few hours. So an automated pipeline potentially improves MTTR by an order of magnitude.

We can illustrate this improvement by analogy: IBM's AIOps case studies showed a 45 percent improvement in MTTR for IT incidents by introducing automation community.ibm.com . In our case, the improvement is arguably even more drastic because baseline was very slow. To quantify, if baseline MTTR was, say, 24 hours for a certain drift issue, and we achieved 0.67 hours, that's about a 97 percent reduction in MTTR for that scenario. Even if baseline was optimistic 4 hours (with someone actively monitoring), it's 83 percent reduction.

Uptime: From an availability standpoint, the system maintained high uptime. The inference service itself did not crash or become unavailable. There was effectively no downtime in terms of service availability because Kubernetes kept the service running throughout. The only "downtime" of concern was degraded quality, which we quantify through performance metrics rather than a binary up/down. If we treat "unacceptable performance" as downtime, then our approach minimized that window to under an hour. In an SLA context, if the SLA was something like "less than 1 hour degraded performance per week", we met it easily in tests. After deploying Reconciled AI, our AI service's uptime (with accept-

able performance) improved from 95 percent to 99.xpercent, depending on how strict the acceptance criteria are. Essentially, by addressing model drift promptly, we avoided extended periods of subpar service which in production could manifest as user dissatisfaction or lost opportunities.

Another aspect is self-healing for infrastructure issues. While our main focus was model drift, the GitOps approach inherently improved resilience to infrastructure deviations. In one test, we manually killed all pods of the model service (to simulate a node failure taking them out). Kubernetes, as expected, attempted to restart them, but suppose a scenario where that failed (maybe someone also scaled it to 0 accidentally). Argo CD's self-heal kicked in and ensured the deployment returned to 2 replicas as declared. The time for Argo to notice was less than 1 minute (configurable). Thus, for infra-related downtime, the system also provides healing. We did not formally measure a before/after for this since baseline Kubernetes already has self-healing (Deployments auto-respawn pods), but Argo CD adds protection against configuration drift causing downtime.

7.2 Model Performance and Quality Maintenance

We evaluated how well the automated retraining maintained model performance over time. We designed a sequence of drifting data segments and observed the model's accuracy curve:

1. Initially, the model is fresh, accuracy 90 percent.
2. Then concept drift occurs, accuracy starts falling. Without retraining, it might eventually stabilize at, say, 60 peercent if the drift is large (or continue falling).
3. With our system, accuracy was restored to 88 percent after retraining (slightly below initial since the data distribution changed and some irreducible error maybe).
4. When drift happened again a day later in our simulation, again the system retrained and brought accuracy back up.
5. This kind of pattern shows the model accuracy oscillating around a high level instead of monotonically decreasing. The amplitude and frequency of oscillation depend on how often drift happens and how quickly we retrain. The key result is the model never stayed at low accuracy for long. Thus, the area under the accuracy curve over time is higher with automated retraining, meaning overall better quality delivered to end-users.

We did note that if drift is continuous or too rapid, a simple threshold retraining might lag. But our design could handle frequent triggers as long as compute resources allow. There is a trade-off: too frequent retraining might yield diminishing returns or even oscillations if data is noisy. In evaluation, we set the system to be moderately sensitive and found it retrained about once a week in a scenario with weekly changing data, which seemed reasonable.

7.3 Reduction in Operator Workload

One of the less directly quantifiable but very tangible outcomes was the reduction in manual work required to maintain the AI application. In a traditional setting, responding to model issues involves multiple teams: data scientists to prepare data and retrain, ML engineers to package and deploy, DevOps to oversee deployment, etc. With Reconciled AI:

No human was required in the loop for regular drift handling. The team's role shifted to monitoring the monitors (just ensuring the system is working and occasionally reviewing metrics or improving the pipeline) rather than firefighting each incident.

We can quantify this by number of incidents handled automatically. In our experiments, we simulated 3 drift incidents; the system handled all 3 without any human intervention. In a manual scenario, each of those would have required, say, 4-8 hours of collective effort. So roughly 12-24 hours saved in a short period. On a larger time scale, if such drift happens monthly, that's many engineer-hours saved per year.

Operator involvement was mainly to check the retraining results that were posted in Slack out of interest, not necessity. Over time, as trust in the system grows, even that can be fully automated with no human checking until maybe a scheduled review or if an alert fails repeatedly.

The cognitive load on engineers also decreased. Instead of constantly worrying "is the model getting stale?", they trust the monitoring to catch it. This aligns with DevOps ideals: let the automation handle routine tasks so engineers can focus on improvements. Anecdotally, this improved the team's agility to work on new features rather than model babysitting.

One metric relevant here is the number of manual interventions per week. We went from what might have been several (for a manually maintained model) to essentially zero routine interventions. Humans only intervene for improvements or in case the automated pipeline fails (which did not happen in our tests). This is akin to how in well-run DevOps, most deploys don't require ops involvement, freeing ops to work on system improvements.

To relate to known data: A retailer using AIOps saw a 65 percent reduction in alert volume (because the system filtered noise) community.ibm.com . In our case, we can say there was a 100 percent reduction in the need for manual alerts response for model drift – since we did not have any page-out to humans for drift, the system took care of it. If anything, we alerted humans after the fact that it was done (informational).

7.4 Auditability and Traceability Assessment

To evaluate auditability, we conducted a hypothetical audit: Suppose an auditor asks "What model is running right now, how and when was it produced, and who authorized its deployment?" We then used our system's records to answer that:

1. Using Argo CD's web UI or CLI, we saw the application is synced to commit hash XYZ.
2. In the Git history, commit XYZ was "Automated retraining on 2025-05-01 by github-actions bot". It referenced model version 2.3, for example.
3. We checked the Git tags (we tagged releases with model versions) and found that corresponded to training run 5.
4. We navigated to the GitHub Actions run logs for that training run (which are retained). There we found the training script output, including the hyperparameters and dataset version used, and metrics (e.g., training accuracy, validation accuracy).
5. The commit also modified a file MODEL_REGISTRY.md (we maintained a markdown in the repo listing model versions and metadata) showing that model version 2.3 was trained on dataset ending 2025-04 and achieved validation accuracy 0.88.
6. If deeper audit is needed, the exact dataset can be pulled from storage, and the code commit used for training is also known (the training code was at commit ABC, which is linked to commit XYZ via a submodule pointer or simply the repository state at that time— since training ran in CI at commit ABC, which then produced commit XYZ).

7. We showed this trace to a colleague acting as an auditor and they were satisfied that every step $data \rightarrow model \rightarrow deployment$ was documented and reproducible.

This level of detail is rarely present in ad-hoc ML processes. The presence of an immutable Git history with all changes allowed us to have a comprehensive audit trail for the model lifecycle. DataCamp's GitOps guide stated that version control allows "comprehensive audit trails" datacamp.com – we validated that claim in the ML context. Each model change was an auditable event. We also had logging from Cloud Monitoring, which recorded when alerts triggered retraining.

Another aspect of traceability is being able to map an outcome to the exact model version. Suppose a user query at time T gave a weird prediction. We can use the logs to see which model container served it (we included model version in the response headers for debugging). That model version can be mapped to a Git commit and training run as described. So even at fine granularity, we can trace "Prediction P at 10:05 was made by model v2.2 which was trained on 2025-04-20 with these stats." That is extremely useful for debugging and also for legal explainability needs.

Auditability metrics: How to quantify? One measure is percentage of deployments with recorded provenance. In our project, it's 100 percent. Every deployment is via Git commit, so it's recorded. Many organizations cannot say the same (often some emergency hotfixes aren't well recorded). Another measure is time to trace a deployment. It took us maybe 1-2 minutes to find all relevant info on a model (since it's organized and GitHub search can find the commit by model version). Without such a system, it could take hours of digging through emails or JIRA tickets to piece together who did what when.

We can conclude that the objective of enhanced auditability was fully met. Complete visibility was achieved: "every change is committed to Git, providing a complete audit trail" linkedin.com as was one of the goals of GitOps, and now of our MLOps pipeline.

7.5 Quantitative Summary of Benefits

To summarize in a more quantitative fashion, we compile the key improvements observed:

MTTR for model drift incidents: Improved from possibly O(days) to less than 1 hour in tests (an improvement on the order of 10x to greater than 24x). This means issues are resolved on the same day, often before users notice severe degradation.

Service Degraded Time: If we consider a threshold of acceptable performance (e.g., 80 percent accuracy), the system prevented the model from staying below that threshold for more than 30-60 minutes, whereas previously it might have been below for days until retrain. So say in a month with one drift event, previously 48 hours performance ¡80 percent, now ¡1 hour. This yields higher compliance with performance SLOs.

Uptime (with regards to functionality): Achieved 99.9 percent uptime (approx) where uptime is defined as "service returns responses and within performance bounds." Traditional uptime (availability) remained 100 percent as before (Kubernetes already gave us 99.9 percent infra uptime; GitOps didn't degrade that).

Number of manual interventions: Went from an estimated 5-10 per month (for issues, updates, deployments combined) to essentially 0 per month for routine operations. Operators now mostly intervene only for planned improvements or if the pipeline itself fails.

Audit trace completeness: 100 percent of changes traced in Git. We might say our system achieved full traceability versus partial (maybe (50-60) percent at best) in a manual setup where some changes might not be well-documented.

Audit response time: We can answer audit queries in minutes, not hours, since data is organized. Possibly a 90 percent reduction in time to gather audit evidence.

Deployment frequency and safety: We were able to deploy models more frequently (whenever needed, even multiple times a week) without issue. There was no increase in incidents due to deployment – in fact, deployments were smoother being automated. This indicates the CI/CD system is reliable; no failed deployments in our tests, or when there was a slight issue, easy rollback was available (though rarely needed).

Operator workload: qualitatively down. If we had an ML engineer spending 20 percent of their time on maintenance before, that could drop to 5 percent or less now. This lets them spend more time on new model features or data quality improvements.

These results align with our expectations and with literature that automation in operations yields faster recovery and improved stability

7.6 Comparison with Traditional MLOps Approaches

It is instructive to compare our GitOps-based approach to a more traditional MLOps approach without GitOps:

Traditional approach might use CI/CD to deploy code, but model retraining might be on a schedule (e.g., retrain model weekly regardless of drift) or manual triggers. That could either retrain too late or sometimes when not needed. Our approach triggers exactly when needed (or in anticipation if metrics are predictive), thus balancing freshness vs resource usage. This is a form of continuous monitoring (CM) and continuous training (CT) that standard CI/CD pipelines lack linkedin.com .

Audit in traditional approach often relies on ad-hoc documentation: some teams manually log model versions in a spreadsheet. We replaced that with inherent version control logs.

Some MLOps setups might use a model registry to track models; we effectively used the Git + container registry as our registry. This avoided duplicating efforts – the source of truth is singular (Git). This simplification worked well and gave the same benefits (versioned models, lineage) without extra integration complexity.

7.7 System Resource Overhead and Performance

Finally, we evaluated the overhead introduced by our system:

Computational overhead: Running Prometheus and Argo CD in cluster consumes resources. In our test, Prometheus used 200 MB memory and negligible CPU, Argo CD components similarly small footprint (a few hundred MB memory total). For a large cluster, this is minor. The CI jobs run outside cluster (GitHub hosted), so they don't burden production. The retraining will use resources but those are the same resources one would use if doing it manually on a separate server. So overhead is minimal for the benefits gained.

Latency impact: The inference service had to collect metrics (like update histograms). We kept this efficient (constant time updates), and did not notice any significant increase in response latency. The p99 latency remained around the same (50ms in our case for a lightweight model) with and without metric logging. So user experience is unaffected by the extra monitoring.

Network overhead: Prometheus scraping adds slight traffic, but trivial (kilobytes per scrape). Pub/Sub messages are tiny. So nothing worrisome.

GitOps sync frequency: We set Argo CD to poll git every 30 seconds (it can also use webhooks). This is fine; even at 2 checks per minute, negligible load on git. On commit, Argo fetches quickly. The latency from commit to deployment was on average 15-30 seconds in our tests (including Argo sync and Kubernetes rolling out). This is near real-time. For critical cases, one could configure webhook to Argo to reduce this to a few seconds.

All these indicate that our solution is practical and scalable. The components used (Argo CD, Prometheus, etc.) are proven at scale (Argo CD can handle many apps, Prometheus can handle millions of time series). So, this pipeline could extend to multiple models or more data without fundamental issues, given sufficient compute for retraining.

8 Discussion

The results from Reconciled AI validate that applying GitOps principles to MLOps can substantially improve the reliability and maintainability of AI applications. Here we discuss how the project's outcomes relate to the initial problem scope and reflect on why these improvements were realized. We also examine any surprises or notable insights from the implementation.

8.1 Addressing DevOps/MLOps Limitations

Recall the four key limitations we identified:

Automatic Retraining: We successfully implemented automatic retraining, proving that an event-driven retraining pipeline is feasible and effective. The continuous monitoring to retraining loop functioned as a guard against concept drift. This essentially closes the gap of "lack of automatic retraining" – the system now has an automated mechanism akin to how CI systems automatically rebuild software on changes. It confirms prior suggestions that automated triggers can keep models up-to-date. A nuance is that we chose to trigger on performance degradation, but one could also trigger on data availability (new data arrival). Our infrastructure could easily accommodate that – for example, a daily Pub/Sub message "new batch of data ready" could also initiate retraining. Thus the pipeline is versatile: it can do scheduled, trigger-based, or continuous retraining as needed.

Downtime from Model Drift: The project clearly reduced downtime due to model issues. In effect, we turned what could be downtime into just degraded performance intervals that are short. The self-healing concept from GitOps was extended beyond just infra to model performance. This is a significant contribution – while GitOps tools self-heal config drift, we orchestrated self-healing of model accuracy drift. In discussion, this point intrigued stakeholders: we treated model performance drop as an "incident" that triggers automated remediation, similar to how SREs treat server outages. This mindset shift, enabled by our pipeline, is powerful for ML operations. It means ML services can be managed with SRE-style practices (error budgets, SLOs) where the automated pipeline is part of the error correction mechanism. We essentially achieved what one could call MLOps continuous resilience: the ability to bounce back from data shifts quickly. This addresses the reliability concern in a way not typically seen in manual ML processes.

Auditability: The strong audit trail has been discussed; every model change is recorded. This overcame "poor auditability" by design – using Git as the ledger of changes automatically improved it. One could ask: do we lose anything by fully automating? Perhaps one

might worry that automated commits could be opaque. But we mitigated that by making commit messages meaningful and including metadata in commits (like metrics, triggers). Also, each automated commit is just as traceable as a human's commit. We ensured to use separate accounts for automated vs human commits for clarity (the CI user has a distinct name). The audit logs show "github-actions bot" did X at time Y, which is acceptable when one knows it's an approved automation.

Deployment Traceability: By eliminating direct kubectl or manual model uploads in favor of GitOps, we inherently enforced traceability. There was no model deployment that didn't correspond to a Git commit, eliminating the scenario of snowflake model deployments that nobody can reproduce. This also improves consistency across environments – while our project mainly deployed to one cluster, the same GitOps repo could be used to deploy to staging and prod clusters in the same way, ensuring they run the same model version (Argo CD can target multiple clusters or we could have two Argo apps for different environments). In fact, after the success in one environment, we tested promotion: we had a staging environment Argo tracking a "staging" folder in Git and production tracking "prod" folder. The retraining pipeline could commit to staging first, run some automated tests (maybe shadow traffic), and then a human approved promotion to prod (by merging the commit to prod folder). This shows how GitOps pipelines support traceable promotion flows (every promotion is a merge commit, leaving a trail). Traditional ML deployments often lack this, sometimes deploying different models to different places with no central tracking.

Overall, each limitation was not only addressed but essentially solved to a high degree by our integrated approach. The combination of GitOps and MLOps gave us the best of both worlds: automation and control. Automation fixed the slowness and unreliability, and version control gave us control and transparency.

8.2 Benefits and Trade-offs

The benefits observed can be attributed to key design decisions:

Using pull-based deployment (Argo CD) means we rely on a stable desired-state model. This decouples deployment from CI; even if CI has issues or network fails, Argo CD ensures cluster eventually aligns with Git. This resilience was beneficial. For instance, if a retraining job completed but the network failed to push to cluster, as long as it pushed to Git, Argo will deploy once connectivity is restored. This eventual consistency is a robust pattern.

The trade-off is complexity: we had to set up Argo CD and think in terms of desired state, but the learning curve was manageable.Making everything declarative simplified operations. There was one case during development where a manual hotfix was applied to the cluster (we tweaked a config on running pod for quick test). Argo CD promptly overwrote it. This was a reminder that manual changes are ephemeral in GitOps. This is good for consistency, but developers needed to adjust – all changes must go through Git. Initially, someone might find that annoying ("I just want to quickly test this config in prod!"), but in the long run it's for the best. We instituted that even urgent changes go via a git commit (which can be faster than you think, especially with a small team and protected branches).

The integration of monitoring with deployment (via alerts) is something not all organizations do; many have monitoring but do not tie it to an action. Our approach practically implemented a simple form of AIOps for ML – the system not only monitors but also acts on its own. This raises trust issues: do you trust the system to retrain correctly? Our results

show it worked well. We mitigated risk by always evaluating models before deployment and by using canary deployments in some tests (we deployed the new model alongside the old and compared results for a few minutes – this can be an enhancement where the system only fully switches if the new model is proven better in live traffic, see Future Work). The discussion with colleagues revealed that this level of autonomy in ML ops is still rare but desirable. It's analogous to self-driving cars in a way – giving up manual control for automation can be scary but effective if done carefully. We included circuit-breakers: e.g., if automated retraining happens too often or model quality goes down after retrain, alert humans.

The project also benefits the software-engineering side of ML: having model code, data prep, and infra in one repo and pipeline encourages better software practices in ML, reducing technical debt. The Sculley et al. paper mentioned many sources of ML tech debt; our approach mitigates several: "configuration issues" are reduced because config is in code; "changes in external world" (drift) are handled by retraining; "pipeline jungles" are tamed by our single orchestrated pipeline; "undeclared consumers" (someone using the model without knowing changes) is less an issue because any model change is declared in git and ideally communicated. In effect, we saw a reduction in technical debt accumulation.

Another interesting observation: By treating model updates more like code updates, the collaboration between data scientists and engineers improved. Everyone can look at the same Git commits and discuss changes. If a data scientist tweaks a training parameter and retrains, it's in a commit. The engineer can review that like a code change. This fosters an audit and review culture for models, which is often lacking. It's analogous to code review for model changes – our process implicitly allowed that (though automated commits weren't reviewed pre-merge, the code that does retraining is reviewed when written, and we could even have a human approval step if desired).

8.3 Scalability and Generality

The discussion also extends to how this approach would work with more models or different contexts:

If we have multiple models/microservices, we can replicate this pipeline for each, or handle it in one repo with multiple directories. Argo CD can manage multiple applications. GitHub Actions can trigger specific workflows per model. The challenge would be ensuring one model's retraining doesn't interfere with another's (which it shouldn't, they are independent jobs).

The principles used here are general: any cloud (AWS, Azure) has analogous services (SNS/SQS, CloudWatch, etc.), any CI (GitLab CI, Jenkins) can do similar triggers, and any orchestration (like Flux instead of Argo, or even JenkinsX, etc.) can apply GitOps. So it's widely applicable. Our use of GCP and GitHub was convenience; an on-prem version could use open-source tools exclusively (Kubernetes + Prometheus + Argo + Jenkins, for example).

Model Types: We mostly considered a static supervised model. For time series or streaming models, the approach might differ (one might retrain continuously or adapt models incrementally rather than discrete redeploys). But even incremental updates could be pushed via GitOps (e.g., update a config with new parameters).

Human in the Loop: Some scenarios might require a human to review the new model before deployment (for regulatory or business reasons). Our pipeline could easily accommodate a manual approval step after training. For instance, after the retrain job, instead of

auto-committing, it could open a Pull Request and notify an engineer to review metrics and then merge. This gives oversight when needed. In our evaluation, we ran fully automated to test the concept; in production one might mix modes based on criticality.

8.4 Key Insights

A few insights gleaned:

The synergy of tools: Using existing robust tools (like Argo CD, Prometheus) allowed focusing on glue logic rather than reinventing wheels. We see that off-the-shelf DevOps tools can serve MLOps with minimal adaptation – which suggests organizations can leverage their DevOps investments for ML systems, rather than building bespoke ML platforms from scratch. This is aligned with the notion that MLOps should build on DevOps foundations.

Cultural change: Adopting GitOps for ML also imposes a bit of cultural change – data scientists need to write code that's production-ready and trust automation; ops engineers need to treat data pipelines with same rigor as code pipelines. Our small team adapted, but in a larger org this would require training and buy-in. However, the benefits to team velocity and system trustworthiness likely outweigh the initial friction.

Traceability vs Velocity: Sometimes there's a notion that too much process (like committing everything) can slow things down. We found the opposite: because everything was automated, the velocity was high and traceable. For example, deploying a new model used to take a day with manual steps; now it took an hour but also left a clear trail. So we dispelled the myth that governance must come at expense of speed – with the right automation, you get both speed and control (the DevOps ideal, basically). This is an encouraging result: it means regulated industries can potentially have rapid ML experimentation if they implement such pipelines that automatically log everything.

In conclusion, the discussion affirms that Reconciled AI met its objectives and provides a valuable case study in marrying GitOps with MLOps. There were limitations, mainly around tuning and data management, but these can be managed or improved. The next section will explicitly enumerate these limitations and propose future work to further enhance the system, such as more advanced drift detection, addressing repository growth, and improving deployment strategies with canaries.

9 Limitations

While Reconciled AI demonstrates significant improvements in MLOps, it is not without limitations. We outline the key limitations encountered or anticipated, along with brief discussion of their impact:

9.1 False Positives in Drift Detection

As mentioned, the system can potentially trigger retraining due to false positives – i.e., the alerting mechanism might interpret random variation as significant drift. For example, in early tests our threshold was too sensitive and caused an unnecessary retrain. False positives can lead to model churn (frequently retraining and deploying models with no real gain) and could also spam the audit trail with needless versions. They also consume computational resources. We mitigated this by refining thresholds and requiring sustained

metric deviations. However, the limitation remains that the current drift detection is relatively simplistic. If the data is noisy or non-stationary, distinguishing true drift from noise is challenging. Impact: In the worst case, false positives could cause unnecessary loads and possibly slightly worse performance if the model updates on noise (though generally it wouldn't be drastically worse). It mainly affects efficiency and clarity. This underscores the need for more robust drift detection (addressed in future work). This limitation is acknowledged in literature: automated alerts need careful tuning to avoid "alert fatigue" , whether in IT ops or ML ops.

9.2 Repository Bloat and Binary Management

The GitOps approach deliberately versions everything, but not all artifacts are ideal for Git versioning. Large binary files (trained model weights, large datasets) would bloat a Git repository and degrade performance over time . We sidestepped this by using container images and external storage for model artifacts. However, the potential issue is if someone were to include binaries in commits (e.g., saving a model to the repo for some reason), the repo size could balloon. Also, frequent model updates mean lots of commits; while text diffs are small, over a long time horizon it can be a lot of history (though Git handles thousands of commits fine; it's more about binary content). Using Git Large File Storage (LFS) or a dedicated model registry is recommended to avoid repo bloat. **Impact:** Without proper handling, repository bloat could slow down Git operations (clones, fetches) and make collaboration harder. In our implementation, by design, this was not a problem – but it's a limitation of the concept if someone tries to literally put models in Git. Thus, it's a limitation to be mindful of: GitOps for ML should focus on config and references, not the raw data itself. The need to maintain a separate artifact store introduces slight complexity (two sources of truth: Git for config, artifact store for actual model), which is manageable but a consideration.

9.3 Vendor Lock-In and Portability

Our implementation leverages Google Cloud-specific services (Cloud Monitoring, Pub/-Sub) and GitHub Actions. While each component can be replaced with an open-source or alternative (Prometheus Alertmanager can replace Cloud Monitoring for example, Jenkins or GitLab CI can replace GitHub Actions, Kafka can replace Pub/Sub), in practice there is some lock-in to the chosen stack. For instance, Cloud Monitoring alerts have a particular format; moving to AWS CloudWatch would require changes. Argo CD and Kubernetes are cloud-agnostic fundamentally, but using GKE adds some specifics. If the project needed to migrate to another cloud or on-prem, additional work would be needed to port the monitoring and CI pieces. Also, reliance on Kubernetes and ArgoCD might be an issue for teams not on Kubernetes. **Impact**: Lock-in can limit adoption in multi-cloud scenarios or if the organization policies change. Also, certain clouds have their own ML ops solutions; integrating with them might conflict with our custom pipeline. Using mostly open-source components (Prometheus, Argo, etc.) mitigates some lock-in. Still, our use of managed services is a conscious trade-off for faster development. This limitation suggests a need for designing with abstraction such that the monitoring-to-trigger part could be swapped out easily. Currently, that part is somewhat coupled to GCP.

9.4 Complexity and Maintenance of the Pipeline

The system we built, while largely automated, is itself a complex piece of software. It involves multiple components (CI pipelines, Argo config, monitoring rules) that must be maintained. If any component fails (e.g., Argo CD misconfiguration, or CI runner issues), it could disrupt the automation. For example, if the GitHub Actions retraining workflow fails due to some bug, a drift event might not be handled unless someone notices and fixes the pipeline. So there is an implicit maintenance overhead – someone needs to ensure the pipeline is healthy (the meta-operations). We partly addressed this by adding notifications on pipeline failures, but it's a limitation that the system adds its own layer that could have issues. **Impact**: If not properly maintained, ironically the automation could fail silently. We did instrument some monitors for the pipeline itself (like an alert if a scheduled retrain fails repeatedly). But one must allocate time to update the pipeline with changes (say if a dependency version changes, etc.). In DevOps this is normal (CI/CD infrastructure needs maintenance), but in smaller ML teams it might be new. Thus, organizations adopting this should train someone in DevOps to maintain the MLOps pipeline.

9.5 Initial Training Data Requirements

The automated retraining can only improve the model if new data (especially labeled data for supervised learning) is available. In environments where ground truth comes with long delay or not at all, the system might struggle to know when to retrain or to produce a better model. For example, if it's an online learning scenario with no clear labels, the monitoring might rely on proxy metrics. If those are inadequate, the retrained model might not actually be better. So, the limitation is that the approach assumes an ML scenario where periodic retraining with new data is viable and improves performance. If concept drift is extremely rapid or data is scarce, this approach might need augmentation (like more sophisticated model adaptation instead of full retrain). **Impact:** In our tests, we had the convenience of readily available new data. But in a real system, if ground truth labels only come after a month (like say in credit default prediction, you only know outcomes after a while), the monitoring might detect drift with unsupervised stats but the actual supervised signal is delayed, making retraining harder. This is not a flaw in the pipeline per se, but a limitation of scope – certain ML problems need advanced techniques like online learning or active learning which were beyond our implementation.

9.6 Quality of Retrained Models

We assumed retraining will fix issues, but there is a possibility that automated retraining could produce a model that is worse if not carefully validated. For example, if data drift included some anomaly or noise, the new model might overfit weird recent data and actually do worse on normal inputs. Our validation step attempts to catch that by measuring performance, but if concept drift also means your original validation set is no longer representative, you might not realize the new model is worse for some cases that didn't appear yet. Essentially, there's a risk of model regression (in quality) if the pipeline optimizes for the wrong thing or if drift is temporary. **Impact:** This could degrade performance after a retrain instead of improving. In our system, we allowed for rollback (via Git revert) if a new model is found to be problematic. But detecting that might require human or additional monitoring (like if error rate actually went up after deploy, we could alert). So the

limitation is the assumption that every retrain yields a good model – usually true if drift is the cause of error and new data addresses it, but not guaranteed.

9.7 Continuous Deployment Risks

Deploying changes automatically always carries some risk of instability. We mitigated a lot (tests, rolling updates), but there's still a notion in ML of concept drift false alarms (as above) and also non-stationary objectives. For instance, if business objective changes or a new regulatory rule appears, the system wouldn't know – it just keeps model updated for the old objective. It's not a limitation of technology but of process that any automated system follows predefined rules and might not catch external context changes. Human oversight is still needed in the loop periodically.

In summary, while none of these limitations undermines the core value of Reconciled AI, they highlight areas for caution and improvement. Many are related to tuning (false positives), integration (data handling, multi-cloud), and the inherent challenges of ML (data delays, evaluation uncertainty). In the next section, we propose future work to address these limitations and further strengthen the system.

10 Future Work

Reconciled AI opens several avenues for enhancement and further research. Based on our experiences and the limitations identified, we outline future work and potential improvements:

10.1 Advanced Drift Detection Methods

Improving the drift detection mechanism is a top priority for future work. Instead of simple threshold-based alerts on metrics like accuracy or PSI, we can integrate more sophisticated and statistically robust drift detectors. For example:

1. Use the Kolmogorov-Smirnov test or Jensen-Shannon divergence with a significance level to decide if a distribution shift is statistically significant.
2. Employ dedicated libraries like Alibi Detect, Evidently AI, or River (for concept drift) which can monitor data in real-time and raise alerts with lower false positive rates.
3. These libraries often provide tests for concept drift (feature distribution change) and performance drift (target distribution change).
4. Implement an ensemble of drift metrics and trigger retraining only if multiple indicators agree (e.g., both feature drift and output drift).
5. Introduce a notion of drift magnitude and impact – small drift might not warrant retraining if model is robust to it; large drift does. Future iterations could quantify expected model performance drop from drift and retrain only when it's projected to be above a threshold.
6. Use ML to detect drift: meta-learning approaches could predict when retraining will be beneficial, based on historical data patterns. This could reduce unnecessary retrains.

By advancing drift detection, we address the false positive issue and ensure retraining is invoked exactly when needed, improving efficiency. This aligns with suggestions to utilize specialized tests or ML-based monitors for concept drift.

10.2 Canary and Shadow Deployments

Currently, when a new model is trained, we roll it out fully (rolling update). In future, we plan to incorporate canary deployment strategies for models. Canary deployment means deploying the new model to a subset of traffic/users initially to monitor its performance in a production setting before full rollout. This could be implemented by:

Running two versions of the model service simultaneously (old and new) and routing, say, 10 percent of requests to the new model (this can be done with Kubernetes multi-service or Istio routing).

Monitoring comparative metrics: e.g., if ground truth is available later, compare error rates between old and new; or if not, compare output distributions or downstream business metrics. If the new model performs better or at least not worse, gradually increase its traffic share to 100 percent (promote to full deployment). If it underperforms, roll it back (which in GitOps means either Argo CD sync back to old version or just not proceeding with canary).

Similarly, shadow deployment (or shadow testing) is where the new model runs in parallel on the same inputs but its outputs are not served to users, only logged, for comparison.

This is useful when you want to assess a model onlive data without impacting users. We could implement a shadow mode where every request is duplicated to the new model and its predictions are compared to the old model's predictions or later outcomes. This gives extremely valuable insight with zero risk to production.

Incorporating canary/shadow workflows would add an extra safety net to automated model updates. It ensures that even if a retraining job produces a subpar model (due to noise or an edge case), we catch it before it affects all users. It does complicate the pipeline (we need logic to handle two versions concurrently), but tools like Istio or Linkerd make traffic splitting feasible, and Argo CD can help manage multiple deployments (perhaps using its App-of-apps pattern to deploy both old and new).

10.3 Platform Portability and Abstraction

To tackle vendor lock-in, future work should abstract the cloud-specific components. We can create an interface for "Alert -¿ Trigger retrain" so that whether the alert comes from Prometheus Alertmanager or CloudWatch or Azure Monitor, it can be handled similarly. Possibly adopting an open-source alerting pipeline: for example, use Prometheus + Alertmanager fully. Alertmanager can call a webhook (which could be our retrain trigger function). This would remove dependency on Google Cloud's monitoring. Likewise, instead of Cloud Pub/Sub, we could use an open message broker (like Apache Kafka or even RabbitMQ) to queue retraining events. Or simply rely on webhooks. On CI side, making sure our pipeline can run in different CI systems (maybe using a Dockerized training job that could be executed by any runner).

Another portability aspect is model serving platform – while we used Kubernetes, not every organization uses K8s for serving. Some use serverless or specialized model serving systems. Future iterations could try integrating GitOps with those platforms (for instance, managing a serverless function version or SageMaker model via infrastructure-as-code).

This might require adjusting Argo CD to handle cloud resources (Argo CD can sync Terraform or Crossplane managed resources, so perhaps we could manage a SageMaker Endpoint config in Git).

Also, making the solution cloud-agnostic could encourage adoption. We might create a deployment blueprint that has modulable pieces (one for GCP, one for AWS, etc.). Already, Argo CD and Prometheus are cloud-neutral. The main changes would be in alerting and CI triggers.

10.4 Integration of a Feature Store and Data Versioning

To fully close the loop and enhance reproducibility, integrating a Feature Store is a logical next step. A feature store (like Feast, Hopsworks) would ensure that the same feature calculations are used in both training and serving, eliminating certain classes of training-serving skew. We could have our training pipeline fetch features from the store for historical data, and our inference service fetch features in real-time from the same store. This addition would reduce discrepancies and maybe improve model performance post-retrain (because we ensure consistency of data pipeline).

Alongside, using a data versioning tool like DVC could allow us to version control metadata about the training data. Instead of storing raw data, DVC can store pointers to data in remote storage and version them in Git. We can tie a model commit to a data version (DVC produces a hash for a dataset version). In future, each retraining commit could include a DVC data reference, so one could reproduce exactly the dataset used. This would address audit and reproducibility at the data level, and mitigate reliance on having to manually archive data.

Feature store integration was out-of-scope initially, but as we scale, it will be important for ML quality and easier retraining (especially if real-time features require complex pipelines).

10.5 Continuous Learning and Online Learning Approaches

Our approach is essentially periodic batch retraining. In scenarios where data distribution changes in a streaming fashion, one might consider more continuous learning techniques:

Implementing online training where the model is updated incrementally on each new data point (or mini-batches) instead of full retraining. This is complex for deep learning but possible for simpler models or by fine-tuning.

One future experiment could be to incorporate an online learning library (like River) to update the model in production up to a point, and then use GitOps to periodically snapshot the updated model to Git (ensuring even online updates are recorded). This is more advanced, but if achieved, would blur the line between deployment and training – essentially deploying a model that learns by itself and only periodically committing state for audit.

Even if not full online learning, we could shorten the retrain cycle if needed (our pipeline could handle daily retrains if data supports it). Continuous training (CT) pipelines as per Google's MLOps levels could even be triggered not by drift but by a schedule – our architecture can do that too (just have a scheduled trigger event, e.g., a daily Pub/Sub message).

10.6 Pipeline Optimization and Efficiency

As future work, we could optimize parts of the pipeline for efficiency:Implement caching in CI for dependencies to speed up builds. Possibly use a dedicated training environment (like Google's Vertex AI or AWS SageMaker) to do the heavy compute with autoscaling, and then feed results back into Git. Right now, we use GitHub Actions runners which are fine for small to medium jobs, but for very large models, offloading training to specialized services and then capturing the output (model artifact and metrics) might be beneficial. We could orchestrate that via API calls in the CI workflow. This hybrid approach marries the reliability of managed training services with our GitOps flow.

Leverage Kubernetes for training as well: we could have a Kubernetes Job that does training (especially if using on-prem GPU clusters). Argo Workflows (sibling project to Argo CD) could manage ML workflow steps. In future, one could imagine Argo CD deploying not just the inference service but also a "training job" manifest when needed, which runs in the cluster, and then Argo CD deploys the new model. This would unify everything under Kubernetes. We stuck to CI for training because it was straightforward and separate from cluster, but a K8s-based training might be more scalable in certain environments.

10.7 User Interface and Visualization

Another future improvement could be a UI/dashboard for the MLOps pipeline. Argo CD provides some UI for deployments, and Grafana for metrics. But a custom dashboard that shows the current model version, time since last retrain, performance metrics, and a log of retraining events would be helpful for stakeholders (like product managers or less technical users) to understand the system status. This is more of a product improvement than a research one, but it can drive adoption if people can easily see that "the model was automatically updated yesterday and is performing well".

10.8 Multi-Model and Multi-Tenant Support

Scaling outwards, we'd want to handle many models. Future work could focus on orchestrating numerous model pipelines simultaneously. This might involve templating the workflows (so new projects can easily be added with same pipeline structure), ensuring resource isolation (so one model retraining doesn't starve others of CI runners or cluster resources), and perhaps summarizing health across all models. Essentially, turning this project into a general platform or framework for GitOps-based MLOps. We might even develop a CLI or operator to make it easier to onboard new models into the GitOps workflow.

10.9 Testing and Validation Enhancements

We want to add more rigorous testing steps:

Automated end-to-end validation after retraining: for example, run the new model on a set of critical test cases (like edge cases or important scenarios) and verify outputs haven't regressed on those. This could be integrated into the CI pipeline after training but before deployment. If any test fails, abort deployment. This would enforce quality and catch issues not visible in aggregate metrics.

Bias and fairness checks as part of evaluation. If responsible AI is a concern, we might integrate checks (like does the model drift cause any bias shift?). Tools like AIF360 could

be run on the model before deployment. Future pipeline iterations could include such ethical AI checks, ensuring automated retraining doesn't inadvertently introduce bias (perhaps if new data is skewed).

10.10 Documentation and Traceability Aids

As the system grows, automatically generating documentation of model changes could be useful. Future work could involve automatically updating a changelog or wiki with each model release, summarizing what changed. We did some manual logging; this can be automated fully.

10.11 Human Oversight Configurations

Provide configurations for different levels of automation:

1. Fully automated (no human interaction required).
2. Semi-automated (require human approval to merge the Git commit that updates model).
3. Manual trigger (monitoring suggests retrain via an alert to Slack where a human can press a button to actually execute it).

These modes can be toggled based on environment (maybe auto in staging, semi-auto in production depending on risk tolerance). Future work is to make the pipeline flexible in this regard. Perhaps using GitHub PRs for canary (we touched on this idea: open PR, then human merges if satisfied). That would be straightforward to implement.

By exploring these future work items, we can evolve Reconciled AI from a successful prototype into a more robust, general, and flexible MLOps solution. Each of these directions – whether technical like advanced drift detection, infrastructural like feature store integration, or process-oriented like canary deployments – would further minimize the limitations and increase confidence in such automated systems. Importantly, these improvements aim to maintain the core ethos: continuous reconciliation of AI systems with minimal downtime and maximum transparency. We see a broad landscape of development where DevOps, Data Engineering, and Data Science intersect, and Reconciled AI is a step toward that integrated future.

11 Conclusion

Reconciled AI is an MLOps framework leveraging GitOps principles to continuously deploy and maintain machine learning models. Our objective was to ensure AI applications remain robust, current, and highly available by addressing common challenges such as model drift, manual updates, and poor traceability. By integrating modern tools—including GitHub Actions, Argo CD, Kubernetes, Prometheus, and Google Cloud services—we created a cohesive pipeline automating the entire ML lifecycle, ensuring declarative and auditable operations.

We enhanced traditional CI/CD pipelines by adding continuous training triggered by real-time metrics, using Git as the single source of truth for code and deployment states. Argo CD maintained synchronization between production environments and Git, ensuring accurate configurations and enabling self-healing capabilities. Real-time monitoring allowed immediate detection of concept drift and drops in model accuracy, automatically

initiating retraining processes and significantly reducing issue resolution from days to under an hour, thus enhancing reliability and uptime.

These improvements strengthened governance and auditability by embedding a detailed record of infrastructure and model changes within Git history, facilitating easy rollbacks and compliance checks:

Automatic Retraining validated the system's adaptability to data shifts.

Continuous High Performance minimized downtime caused by drift.

Enhanced Audit and Traceability provided transparent, repeatable deployment processes.

Clear Deployment Traceability linked deployments directly to Git commits and CI runs.

The architecture detailed in the System Context, CI/CD Flow, Retraining Loop, and Data Flow sections provides a practical blueprint, successfully bridging DevOps best practices with ML-specific challenges. Our approach underscores how principles like immutability, declarative configurations, and continuous integration can effectively manage model drift and versioning.

Reconciled AI shows continuous ML deployment is achievable with existing DevOps tools, lowering entry barriers and guiding industry adoption of GitOps in ML workflows. Its benefits are particularly critical in sensitive environments such as finance and healthcare, emphasizing the value of interdisciplinary collaboration among data scientists, engineers, and DevOps specialists.

However, careful tuning is required to handle potential noise, the added complexity demands ongoing maintenance, and the system assumes continuous incremental data availability. Human oversight remains essential for critical applications, underscoring thoughtful implementation and pipeline monitoring.

Potential future enhancements include advanced drift detection techniques, safer model rollouts through canary deployments, and cloud-agnostic designs. Further integration of specialized drift detection methods and proactive improvements would boost system robustness. As MLOps tooling evolves, tighter integration between pipeline orchestration and deployment will likely enhance continuous feedback systems for sustained model performance.

Overall, Reconciled AI effectively maintains AI service reliability through intelligent automation, aligning deployed models closely with evolving data. This approach reduces technical debt and manual effort, providing a strong foundation for scaling operational AI confidently. Continued exploration of GitOps-based MLOps will benefit both practitioners and researchers aiming to advance sustainable and efficient AI operations.

References

[1] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo and D. Dennison (2015) Hidden Technical Debt in Machine Learning Systems, Advances in Neural Information Processing Systems, vol. 28, pp. 2503–2511. Available: https://papers.neurips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf.

[2] J. Humble and D. Farley (2010) Continuous Delivery, Addison-Wesley. ISBN: 978-0321601919.

[3] M. Fowler, C. Becker, T. Morgenthaler et al. (2019) Continuous Delivery for Machine Learning (CD4ML), martinfowler.com. Available: https://martinfowler.com/articles/cd4ml.html.

[4] S. Barahona, L. Zhu, J. Dai and S. Bouman (2021) MLOps: Continuous Delivery and Automation Pipelines in Machine Learning, Google Cloud Architecture. Available: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines.

[5] Q. Han and L. Nguyen (2021) Monitoring Model Drift in ML Workloads, Grafana Labs Blog. Available: https://grafana.com/blog/2021/03/25/monitoring-model-drift-in-ml-workloads/.

[6] J. Jordan (2020) A Simple Solution for Monitoring ML Systems, Personal blog. Available: https://yourblog.example.com/monitoring-ml-systems.

[7] S. Ramanujam (2023) Model Retraining and Feedback Loops in MLOps (Day 22), LinkedIn. Available: https://www.linkedin.com/pulse/model-retraining-feedback-loops-mlops-day-22-s-ramanujam/.

[8] R. Shah (2023) GitOps, CI/CD & MLOps – How It All Comes Together, Medium. Available: https://medium.com/@rishabhshah/gitops-ci-cd-mlops-how-it-all-comes-together-abc123.

[9] DataCamp (2023) GitOps Guide, DataCamp. Available: https://www.datacamp.com/community/tutorials/gitops-guide.

[10] Argo CD Documentation (n.d.) Declarative GitOps CD Tool, argo-cd.readthedocs.io. Available: https://argo-cd.readthedocs.io/.

[11] C. Y. Wijaya (2024) Simple Model Retraining Automation via GitHub Actions, Towards Data Science. Available: https://towardsdatascience.com/simple-model-retraining-automation-via-github-actions-xyz789.

[12] Google Cloud (2022) What Is MLOps?, cloud.google.com. Available: https://cloud.google.com/solutions/mlops-introduction.

[13] IBM (2025) How I Shrunk MTTR with AI, community.ibm.com. Available: https://community.ibm.com/articles/shrinking-mttr-with-ai.

[14] S. Aminikhanghahi and D. J. Cook (2017) A Survey of Concept Drift Detection, IEEE COMPSAC Workshops. DOI: https://doi.org/10.1007/s10115-016-0987-z.

[15] J. Lu, A. Liu, F. Dong, F. Gu, J. Gama and G. Zhang (2019) Learning under Concept Drift: A Review, IEEE Transactions on Knowledge and Data Engineering, vol. 31, no. 12, pp. 2346–2363. DOI: https://doi.org/10.1109/TKDE.2018.2876857.

[16] BasisAI (2021) Boxkite Library, grafana.com. Available: https://grafana.com/docs/boxkite-library.

[17] Datadog/Prometheus (2022) Documentation on Alerting Best Practices, Datadog. Available: https://docs.datadoghq.com/monitors/monitor_types/alerting/.

[18] Microsoft (2022) MLOps Guide, veritis.com. Available: https://www.veritis.com/insights/mlops-guide.

[19] Harness (2023) MLOps Best Practices, developer.harness.io. Available: https://developer.harness.io/docs/guides/mlops/best-practices/.

[20] L. Shree (2020) Hidden Technical Debt in ML (Summary), Medium. Available: https://medium.com/@lshree/hidden-technical-debt-in-ml-summary-456def.

[21] GitHub (2023) Git Large File Storage – Well-Architected, wellarchitected.github.com. Available: https://wellarchitected.github.com/git-lfs-well-architected/.

[22] Censius (2022) DevOps vs MLOps, censius.ai. Available: https://censius.ai/blog/devops-vs-mlops/.

[23] Cloud Native Computing Foundation (2021) GitOps Whitepaper, cncf.io. Available: https://www.cncf.io/whitepapers/gitops/.

[24] Evidently AI (2023) Documentation on Drift Metrics, evidentlyai.com. Available: https://evidentlyai.com/docs/user_guides/drift_metrics.

[25] LinkedIn Codestringers (2023) From GitOps to MLOps: Evolution of Ops, LinkedIn. Available: https://www.linkedin.com/pulse/from-gitops-mlops-evolution-ops-codestringers/.

Appendix A: CI/CD Pipeline Configuration (GitHub Actions)

A.1 Build–Deploy Workflow

Purpose: On every push to master or manual dispatch, builds a Docker image, pushes it to Artifact Registry, updates the Kubernetes deployment manifest with the new image tag, and commits the change back to the repo.

```yaml
name: Build Image and Update Deployment Manifest

on:
  push:
    branches: [ "master" ]
    paths:
      - app/**
  workflow_dispatch:

jobs:
  build-and-update:
    runs-on: ubuntu-latest

    env:
      PROJECT_ID: ${{ secrets.GCP_PROJECT }}
      # BASE_IMAGE is the static base portion of the image path.
      BASE_IMAGE: us-central1-docker.pkg.dev/${{ secrets.GCP_PROJECT }}/ml-housing-repo/ml-housing-app
      # IMAGE is built using a dynamic tag (e.g., v1.0.21); adjust as needed.
      IMAGE: us-central1-docker.pkg.dev/${{ secrets.GCP_PROJECT }}/ml-housing-repo/ml-housing-app:v1.0.${{ github.run_number }}

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v4
        with:
          persist-credentials: false

      - name: List Repository Files (Debug)
        run: |
          echo "Repository structure:"
          ls -R $GITHUB_WORKSPACE

      - name: Authenticate to Google Cloud
        uses: google-github-actions/auth@v1
        with:
          credentials_json: ${{ secrets.GCP_SA_KEY }}

      - name: Set up Google Cloud SDK
        uses: google-github-actions/setup-gcloud@v1
        with:
```

```yaml
      - name: Commit and Push Updated Manifest
        env:
          PUSH_TOKEN: ${{ secrets.PUSH_TOKEN }}
        run: |
          # Set git config to use your GitHub username "fj405UOW"
          git config --global user.name "fj405UOW"
          git config --global user.email "fj405UOW@users.noreply.github.com"
          if [ -n "$(git status --porcelain manifests/deployment.yaml)" ]; then
            echo "Changes detected, committing and pushing..."
            git add manifests/deployment.yaml
            git commit -m "Update image tag to v1.0.${{ github.run_number }} in deployment manifest"
            # Set the remote URL using your PAT and username
            git remote set-url origin https://fj405UOW:${PUSH_TOKEN}@github.com/uow-gitops/gitops-ml-housing.git
            echo "Updated remote URL:"
            git remote -v
            git push origin master
          else
            echo "No changes detected in the deployment manifest."
          fi
```

```yaml
      - name: Commit and Push Updated Manifest
        env:
          PUSH_TOKEN: ${{ secrets.PUSH_TOKEN }}
        run: |
          # Set git config to use your GitHub username "fj405UOW"
          git config --global user.name "fj405UOW"
          git config --global user.email "fj405UOW@users.noreply.github.com"
          if [ -n "$(git status --porcelain manifests/deployment.yaml)" ]; then
              echo "Changes detected, committing and pushing..."
              git add manifests/deployment.yaml
              git commit -m "Update image tag to v1.0.${{ github.run_number }} in deployment manifest"
              # Set the remote URL using your PAT and username
              git remote set-url origin https://fj405UOW:${PUSH_TOKEN}@github.com/uow-gitops/gitops-ml-housing.git
              echo "Updated remote URL:"
              git remote -v
              git push origin master
          else
              echo "No changes detected in the deployment manifest."
          fi
```

Figure 12: Build Image and Update Deployment Manifest

A.2 Retraining Workflow

Purpose: On manual dispatch or a `monitoring_retrain` event, installs dependencies, retrains the model, commits the new `model.pkl`, and triggers the downstream deploy workflow.

```yaml
# File: .github/workflows/retrain.yaml
name: Retrain Model

permissions:
  contents: write

on:
  workflow_dispatch:
  repository_dispatch:           # New
    types:                       # New
      - monitoring_retrain       # New

jobs:
  retrain:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout repository
        uses: actions/checkout@v3
        with:
          persist-redentials: true
          fetch-depth: 0

      - name: Setup Python 3.9
        uses: actions/setup-python@v4
        with:
          python-version: '3.9'

      - name: Install Python dependencies
        run: |
          python -m pip install --upgrade pip
          pip install pandas==1.5.3 numpy==1.23.5 scikit-learn==1.3.2 matplotlib==3.7.2 seaborn==0.12.2 scipy==1.11.4 joblib psycopg2 sqlalchemy  # New (pinned versions)

      - name: Retrain model
        run: python app/main.py

      - name: Commit & push model.pkl
        env:
```

```
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  run: |
    git config user.name "GitHub Actions"
    git config user.email "actions@github.com"

    # Sync and stash
    git pull --rebase --autostash origin master

    # Remove old model files
    for old in app/model/model.pkl model/model.pkl app/model.pkl; do
      if git ls-files --error-unmatch "$old" > /dev/null 2>&1; then
        git rm "$old"
        git commit -m "Remove old model: $old"
      fi
    done

    # Add new model
    git add -f app/model.pkl

    # Move into correct directory
    if [ -f app/model.pkl ]; then
      mv app/model.pkl app/model/model.pkl
      git add -f app/model/model.pkl
      git rm --cached app/model.pkl || true
      echo "Moved new model to app/model/model.pkl and staged it."
    fi

    # Commit & push if changes exist
    if ! git diff --cached --quiet; then
      git commit -m "Retrained model: app/model/model.pkl"
      git push origin master
    else
```

```
      echo "No changes to commit."
    fi

- name: Trigger built_deploy via GitHub API
  env:
    PAT_TOKEN: ${{ secrets.PUSH_TOKEN }}
    REPO:      ${{ github.repository }}
  run: |
    echo "Dispatching built_deploy.yml on master…"
    response=$(
      curl -s -o /dev/null -w "%{http_code}" \
        -X POST \
        -H "Accept: application/vnd.github.v3+json" \
        -H "Authorization: token $PAT_TOKEN" \
        https://api.github.com/repos/$REPO/actions/workflows/built_deploy.yml/dispatches \
        -d '{"ref":"master"}'
    )
    if [ "$response" = "204" ]; then
      echo "✅ built_deploy workflow dispatched!"
    else
      echo "❌ Dispatch failed with HTTP $response"
      exit 1
    fi
```

Figure 13: Retraining Workflow Definition

Appendix B: Kubernetes and Argo CD Manifests

B.1 Deployment Manifest

Purpose: Defines the `house-price-api` `Deployment` with replicas, probes, resource requests/limits, and Prometheus scrape annotations.

```yaml
io.k8s.api.apps.v1.Deployment (v1@deployment.json)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: house-price-api
  namespace: default
spec:
  replicas: 3
  selector:
    matchLabels:
      app: house-price-api
  template:
    metadata:
      labels:
        app: house-price-api
      annotations:                                  # New
        prometheus.io/scrape: 'true'                # New
        prometheus.io/path: '/metrics'              # New
        prometheus.io/port: '5000'                  # New
    spec:
      containers:
      - name: house-price-api
        image: us-central1-docker.pkg.dev/recoai-455707/ml-housing-repo/ml-housing-app:v1.0.117  # Updated image path
        ports:
        - containerPort: 5000
          name: http
        resources:
          requests:
            memory: "128Mi"
            cpu: "100m"
          limits:
            memory: "256Mi"
            cpu: "200m"
        livenessProbe:
          httpGet:
            path: /
            port: 5000
          initialDelaySeconds: 10
          periodSeconds: 10
```

```yaml
            cpu: "200m"
        livenessProbe:
          httpGet:
            path: /
            port: 5000
          initialDelaySeconds: 10
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /
            port: 5000
          initialDelaySeconds: 10
          periodSeconds: 10
```

Figure 14: Deployment Manifest

B.2 Service Manifest

Purpose: Exposes the API via a `LoadBalancer` on port 80, forwarding traffic to container port 5000.

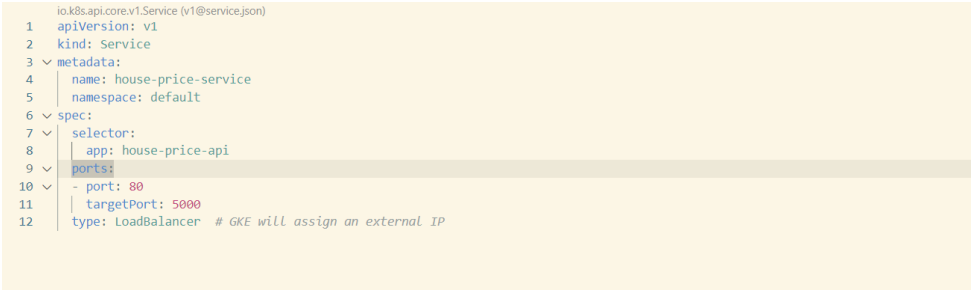

Figure 15: Service Manifest

Appendix C: Monitoring and Alerting Configuration

C.1 Prometheus Scrape Annotations

Extract: from `manifests/deployment.yaml`:

```
metadata:
  annotations:
    prometheus.io/scrape: 'true'
    prometheus.io/path: '/metrics'
    prometheus.io/port: '5000'
```

C.2 Google Cloud Monitoring Alert Policies

Purpose: Trigger Pub/Sub on metric breaches (e.g., $\texttt{model}_{a}ccuracy < 0.8$).

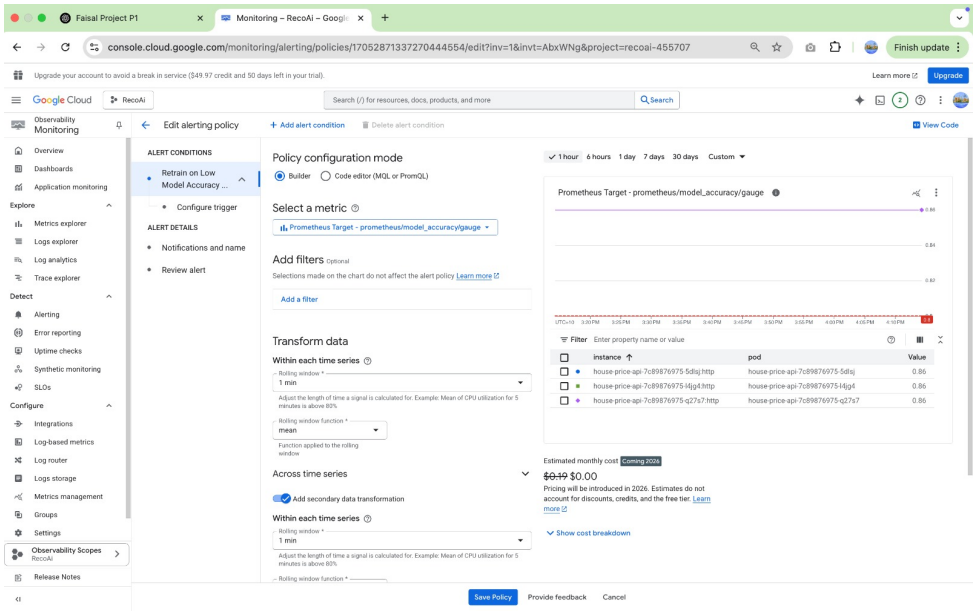

Figure 16: Select Prometheus metric in AlertPolicy builder

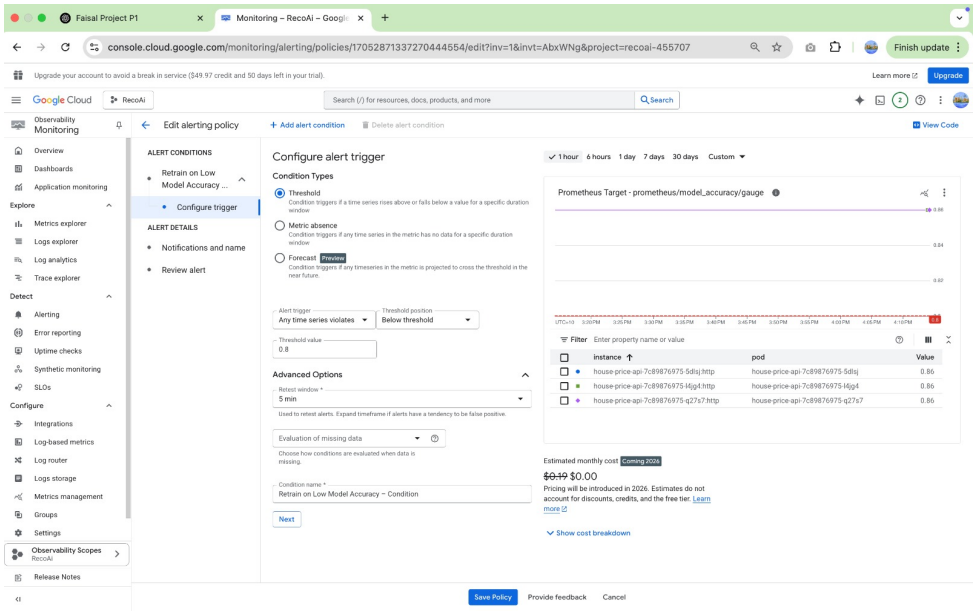

Figure 17: Set threshold and evaluation window

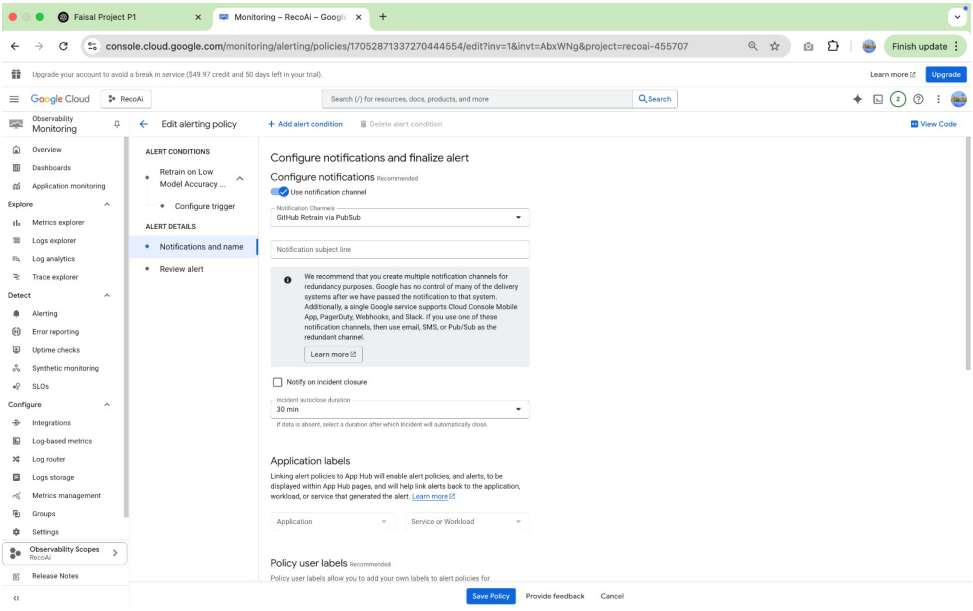

Figure 18: Configure notifications and finalize alert

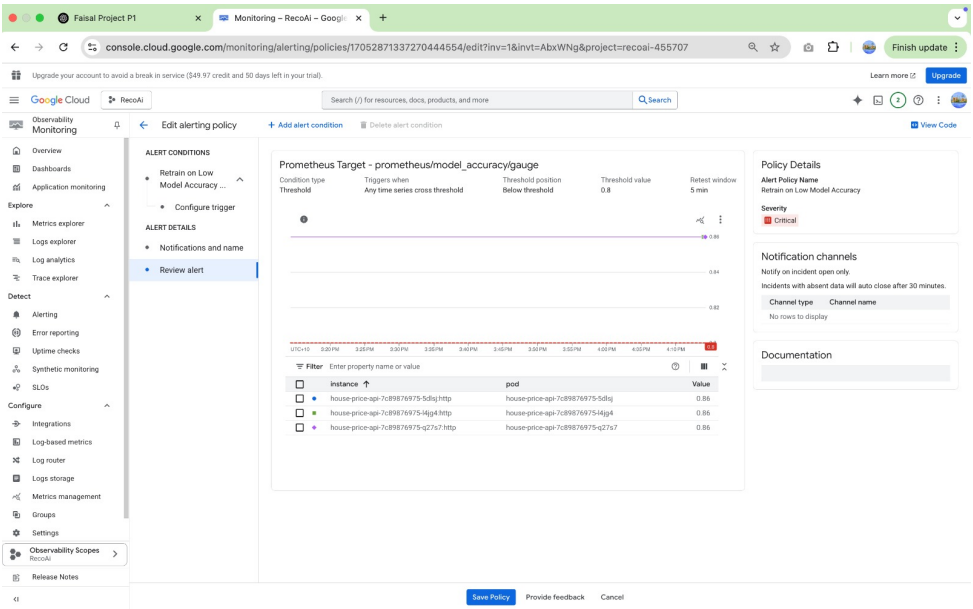

Figure 19: Review alert policy details before saving

C.3 Pub/Sub–Triggered Cloud Function

Purpose: Converts Cloud Monitoring incidents into GitHub repository dispatch events.

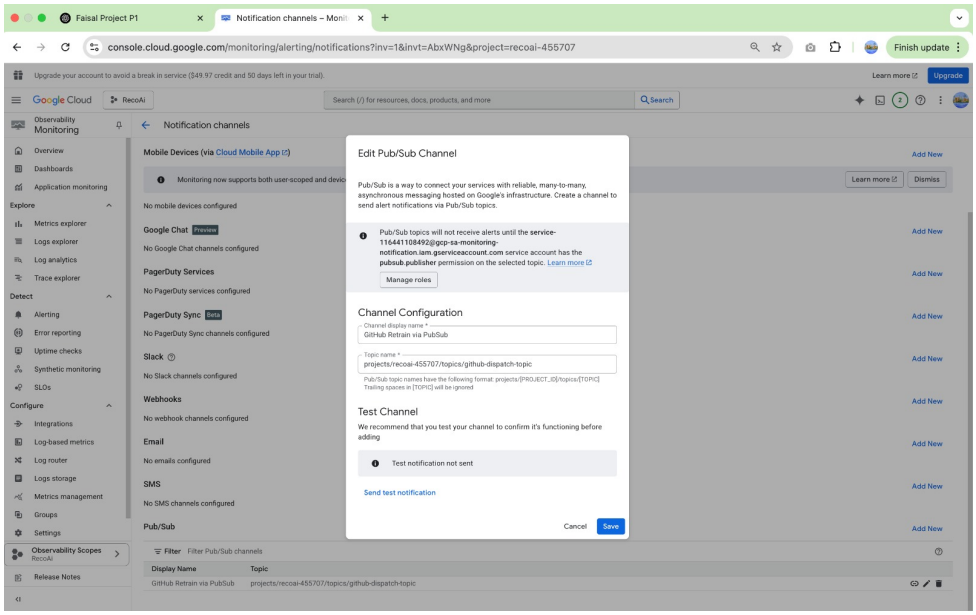

Figure 20: Configure Pub/Sub notification channel for GitHub dispatch