VRIJE UNIVERSITEIT AMSTERDAM

UNIVERSITEIT VAN AMSTERDAM

Master's Thesis
(Course Code: XM_0011)

A Comparative Implementation Study of MLOps Tools Through an Industrial Anomaly Detection Pipeline

by

Zakkarija Micallef
(Student Number: 2815221)

Submitted in partial fulfillment of the requirements
for the joint UvA-VU degree of
Master of Science
in
Computer Science
at the
Vrije Universiteit Amsterdam

January 3, 2026

Certified by .....
Dr. Ilias Gerostathopoulos
Assistant Professor
First Supervisor

Certified by .....
Keerthiga Rajenthiram
Ph.D. Researcher
Daily Supervisor

Certified by .....
Dr. Justus Bogner
Assistant Professor
Second Reader

GOVERNMENT OF MALTA

FONDI.eu

The research work disclosed in this publication is partially funded by the Endeavour II Scholarships Scheme. The project is co-funded by the ESF+ 2021-2027

Co-funded by the European Union

ENDEAVOUR SCHOLARSHIPS SCHEME

A Comparative Implementation Study of MLOps Tools Through an Industrial Anomaly Detection Pipeline

Zakkarija Micallef
Vrije Universiteit Amsterdam
Amsterdam, NL
z.micallef@student.vu.nl

Abstract

Machine learning operations has become increasingly critical as more organisations deploy machine learning models into production. However, the growing landscape of MLOps solutions has introduced complexity for practitioners trying to select appropriate tools. This thesis evaluates the most popular MLOps tools identified in our prior systematic literature review, organizing them into two representative pipelines centered around MLflow and Kubeflow to test realistic integration scenarios. Using a real industrial anomaly detection pipeline from IDEKO, we implemented both configurations and assessed individual tools across four dimensions: usability, functionality, flexibility, and vitality. The evaluation revealed fundamental differences in complexity and required expertise. Tools in the MLflow-centered pipeline provided straightforward workflows with minimal setup, enabling teams to start quickly and scale gradually. In contrast, tools in the Kubeflow-centered pipeline demanded substantial Kubernetes knowledge and operational expertise for even basic tasks. Beyond individual tool assessment, our implementation revealed integration challenges such as documentation fragmentation and hidden complexity. These practical insights could only be discovered through hands-on implementation, providing teams with realistic expectations about the operational overhead and expertise required for different MLOps tools and their integration patterns.

1 Introduction

Machine learning (ML) is becoming increasingly used across all sectors of industry and society. Organizations integrate ML models into production systems for tasks ranging from manufacturing anomaly detection to healthcare diagnostics, creating a growing need for accessible deployment and operational tools. However, many AI engineers and data scientists lack the software deployment expertise of operations teams, especially when it comes to MLOps tools [1]. The MLOps tool landscape is confusing since there is significant fragmentation with hundreds of tools, many of which lack maturity. Teams face issues such as tool variety, choice complexity, rapid development causing breaking changes, and integration problems between different vendors' tools [1]. New tools appear constantly while existing ones evolve rapidly, making it difficult for practitioners to keep up.

This thesis is largely a follow-up to a Systematic Literature Review (SLR) we previously conducted [2]. In order to understand the Machine learning operations (MLOps) landscape, the SLR examined 41 papers, noting the most popular tools, what parts of the ML lifecycle they address, and what experiences publications reported. Our thesis builds upon those findings by deploying and comparing the most popular tools first-hand to understand their practical use and integration with other tools.

We implement an industrial anomaly detection pipeline on the most popular MLOps tools identified in our SLR. The use case provided by IDEKO, a research center specializing in manufacturing technology [3]. It addresses a real problem involving high-frequency sensor data from precision grinding machines, where detecting early signs of wear can prevent costly failures. This grounding in a practical industrial case ensures that the evaluation reflects real constraints rather than idealized scenarios. The evaluation goes beyond verifying whether tools function. It examines the entire experience, from installation and setup to the required knowledge, debugging process, and ease of integration.

The thesis is structured as follows: Section 2 provides an in-depth background on Development and Operations (DevOps) and MLOps, and summarizes findings from the SLR. Section 3 details the research design, including the case study methodology and evaluation framework. Section 4 presents results from both individual tool comparisons and full stack evaluation. Section 5 discusses the implications and trade-offs discovered. Section 6 outlines threats to validity. Section 7 covers related work, and Section 8 concludes with practical recommendations for teams choosing between these platforms.

2 Background

This section establishes the foundational concepts of MLOps, and summarizes key findings from our SLR that motivated this thesis.

2.1 DevOps

Traditionally, an operations team is in charge of tasks such as deployment and ongoing monitoring. To reduce software time-to-value and create stronger collaboration between development and operations, software companies frequently use DevOps. DevOps is described as a culture that emphasises continuous collaboration throughout the software lifecycle. It involves practices such as continuous integration (CI), which entails frequent code merges, and continuous deployment (CD), which automates the release process to ensure software remains deployable [4].

2.2 MLOps

ML systems combine code with data, making them harder to maintain than traditional software [5]. The main challenge is that software developers, data scientists, and domain specialists must work together despite using different tools and development cycles. MLOps addresses this integration problem by combining ML, DevOps, and Data Engineering practices to help these teams collaborate when deploying ML systems [6]. MLOps refers to the entire lifecycle of

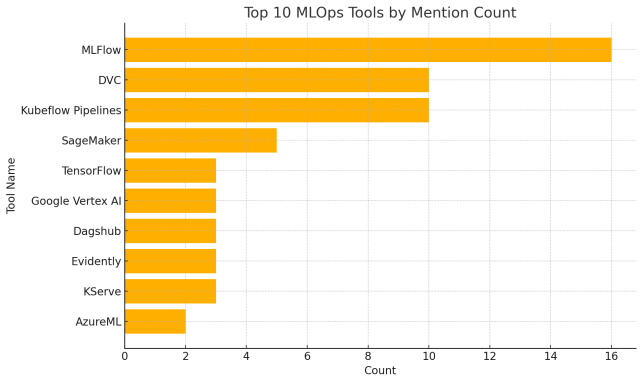

Figure 1: Most mentioned tools according to our SLR [2]

the machine-learning process, bridging the gap between data, development, and operations. MLOps extends beyond applying DevOps principles to ML. It involves CI/CD, automation for ML pipelines, orchestration of ML workflows, versioning of data, models, and code to ensure reproducibility, continuous training to keep models up to date, metadata tracking for experiment auditability, and continuous evaluation and performance monitoring [6].

2.3 Systematic Literature Review

Despite the growing interest in MLOps, existing reviews often remain high-level focusing primarily on listing tools, comparing surface-level features, or distinguishing between open-source and proprietary solutions. However, there is limited synthesis of how these tools are actually used in practice. Most studies fail to capture the practical experiences of teams deploying real-world ML systems. This gap in the literature limits the ability of researchers and practitioners to make informed choices based on implementation outcomes rather than tool specifications alone.

To understand the current MLOps tool landscape, we conducted an SLR of academic papers focused on MLOps tools [2]. A structured search on Google Scholar followed by manual screening reduced 96 papers to 41 primary studies. The analysis revealed clear adoption trends: MLflow was the most frequently mentioned tool (16 papers), followed by DVC and Kubeflow Pipelines (10 papers each), and Amazon SageMaker (8 papers). Their popularity can be linked to their ability to address specific pain points. MLflow simplifies experiment tracking, DVC offers Git-style version control for large datasets, Kubeflow provides cloud-agnostic workflow management, and SageMaker delivers a fully managed environment.

The review also found that MLOps pipelines often share the same eight stages: data ingestion and versioning, feature engineering and storage, workflow orchestration, experiment tracking and hyperparameter optimization (HPO), model registry, CI/CD and packaging, model serving, and performance monitoring. No single tool supports the entire lifecycle, so practitioners typically combine specialised tools to cover all stages. This makes integration a critical factor when designing MLOps stacks. Accordingly, this thesis compares complete tool stacks on top of the individual tools.

3 Design

This section details our research methodology, including the case study approach, the IDEKO industrial use case, tool selection rationale, and the evaluation framework used to assess both individual tools and complete stacks.

3.1 Research Questions

RQ1: How do the most popular MLOps tools compare across usability, functionality, flexibility, and vitality dimensions when applied to an industrial anomaly detection pipeline?

We assess the most popular MLOps tools according to our SLR [2], comparing functionally equivalent alternatives: DVC vs LakeFS for data versioning, Airflow vs Kubeflow Pipelines for orchestration, MLflow Tracking vs ML Metadata (MLMD) for experiment tracking, Optuna vs Katib for HPO, and MLflow Serving vs KServe for model serving. Feast serves as the feature store for both stacks.

Each tool is exercised inside the IDEKO workflow. The evaluation captures individual capabilities and integration characteristics, such as how tools pass data, trigger subsequent steps, and share metadata with other components. We apply a consistent rubric across four quality dimensions: usability, functionality, flexibility, and vitality. This approach reveals both the standalone merits of each tool and the friction that emerges when integrated with others. The goal is to provide engineers with practical insights about real-world tool behaviour.

RQ2: How do MLflow-centric and Kubeflow-centric stacks compare as end-to-end MLOps solutions for an industrial anomaly detection pipeline?

This question shifts focus from individual tools to complete MLOps stacks. According to our SLR, MLflow and Kubeflow emerge as the two most widely adopted MLOps ecosystems, though they represent fundamentally different architectural approaches [2]. We implement the full IDEKO anomaly detection pipeline in both stacks to understand how their architectural choices affect the workflow. Both stacks handle identical tasks: data versioning, orchestration, experiment tracking, hyperparameter tuning, and model serving.

The MLflow-centric stack represents a collection of best-of-breed tools (MLflow for tracking, DVC for versioning, Airflow for orchestration, Optuna for HPO) integrated together to form a pipeline, testing how these popular tools behave when combined. In contrast, Kubeflow provides a cohesive platform with Kubernetes-native orchestration for containerized workflows, where components are designed from the outset to work together as an integrated system.

We examine individual tool performance alongside stack-level characteristics, including setup complexity, integration effort and debugging experience. The same rubric is applied to both the component and stack levels to compare an MLflow-centric stack assembled from popular standalone tools with Kubeflow's integrated platform.

3.2 Research Methodology: IDEKO Case Study

In order to compare our MLOps tool stacks, a realistic test bed was required. Our research adopts a case study methodology since it is particularly suitable in software engineering and ML research

Table 1: Comprehensive Tool Selection for MLflow and Kubeflow Stacks

Component	MLflow	Kubeflow	Description	Rationale
Data Versioning	DVC	LakeFS	**DVC:** Git-like workflow to version large datasets and models alongside code [7–9]. **LakeFS:** Git-like version control for object storage. S3-compatible and Kubernetes-native architecture suits containerized workflows [10].	**DVC:** Most popular tool overall [2]. **LakeFS:** Kubernetes-native architecture suits Kubeflow stack. [10].
Feature Store	Feast	Feast	Provides offline and online feature storage, keeping historical and live data in sync [11, 12].	Most widespread feature store in the SLR [2].
Orchestration	Airflow	KF Pipelines	**Airflow:** Open-source platform for orchestrating production workflows and data pipelines [13]. **KF Pipelines:** Orchestrates containerized workloads with parallel execution on Kubernetes [14] [15].	**Airflow:** Mature, widely adopted. Second most popular orchestrator in SLR [2]. **KF Pipelines:** Most popular orchestrator in SLR [2] and is the core Kubeflow platform tool.
HPO	Optuna	Katib	**Optuna:** Python HPO framework with efficient search algorithms and latest research techniques [16]. **Katib:** Kubernetes-native automated tuning with pod-based parallelization [17, 18].	**Optuna:** A widely adopted HPO tool that integrates with MLflow, making it a natural choice for inclusion in our MLflow stack evaluation [19]. **Katib:** Part of the Kubeflow ecosystem.
Experiment Tracking	MLflow Tracking	MLMD	**MLflow Tracking:** Provides comprehensive experiment tracking [20, 7, 21, 11]. Records parameters, metrics, and artifacts for every run [22]. MLFlow's auto logging captures parameters and metrics automatically from supported ML libraries [21, 23]. **MLMD:** Unified metadata store for Kubeflow ecosystem, tracking artifacts, executions, and lineage across pipelines [24].	**MLflow:** Core MLflow component. Most popular experiment tracking tool in SLR and second most popular tool overall [2]. **MLMD:** Native integration with KF Pipelines, ensures seamless metadata propagation and lineage tracking [24].
Model Registry	MLflow Models	KF Model Registry	**MLflow Models:** Stores and versions trained ML models along with basic metadata [23]. **KF Model Registry:** Provides centralized model versioning and governance within Kubeflow ecosystem [25].	**MLflow:** Part of the MLflow ecosystem, and is the most popular tool in MLOps projects as per the SLR [2]. **KF Model Registry:** Kubeflow-stack equivalent, though unavailable in evaluated distributions as discussed in Section 4.1.3.1 [25].
Dataset or Artifact Management	MLflow Datasets	KF Artifacts	**MLflow Datasets:** Links datasets to experiments, providing complete lineage tracking. Captures dataset versions and associates them with training runs [23]. **KF Artifacts:** Manages intermediate pipeline outputs and artifacts between pipeline components [14].	**MLflow:** Included as a library with the MLflow Tracking python package [23]. **KF Artifacts:** Part of Kubeflow Pipelines as it is essential for passing data between isolated pipeline components in Kubernetes pods. [14]
Model Serving	MLflow Serving	KServe	**MLflow:** REST API deployment with minimal configuration. Auto-creates isolated Python environments and standardized endpoints [23]. **KServe:** Kubernetes-native serverless inference. Features auto-scaling, canary rollouts, A/B testing, and multi-model serving [26].	**MLflow:** Part of the MLflow ecosystem, and offers single-command deployment [23]. **KServe:** Pre-integrated in Charmed Kubeflow, and has a Kubernetes-native architecture [27].

as it enables in-depth exploration of complex, real-life situations that cannot easily be replicated or isolated in a laboratory setting [28]. Case studies give us real-world insights rather than general statistics. This fits our goal of studying an actual MLOps pipeline to get detailed, practical findings that we couldn't get from theory alone. We followed standard case study practice; data collection and analysis were iterative rather than strictly sequential [28]. We ran a pilot to validate the method and scoring rubric, see Section 3.2.1.

In our work, we implemented a single-case study centred on an industry-relevant anomaly detection pipeline. The case study was executed using real-world data provided by the industrial partner, IDEKO, as a realistic testbed. Rather than by deploying changes in a live production environment, we built on an already existing anomaly detection pipeline, reusing production-grade code, tooling, and data, yet we did not deploy to production machinery, thus avoiding interference with operational activities. This approach allowed us to rigorously assess the tools in a realistic use-case, thereby ensuring the findings remain grounded in reality and relevant to practitioners.

3.2.1 Pilot Study

The pilot study served as a crucial validation phase for our evaluation methodology described in Section 3.5. We selected DVC and Feast as representative tools to test every aspect of our experimental approach before proceeding with the full evaluation. This iterative process involved designing a systematic test plan, implementing the tools within the IDEKO pipeline, documenting detailed observations, applying the evaluation rubric, and assigning final grades across all four quality dimensions. Through multiple iterations, we refined each step as follows: the test procedures became more systematic with a predefined plan, and for each tool we kept a short list of goals to attempt. The note-taking structure evolved to capture relevant details consistently through descriptive field notes that served as the historical record of the case study process which can be seen in the Appendix. Most importantly, the rubric scoring criteria were clarified and aligned with established standards such as ISO/IEC 25010 [29]. The pilot study revealed gaps in our initial approach, such as evaluation criteria that were too detailed to assess in practice, and it highlighted the importance of capturing integration pain points. Only after achieving consistent and reproducible results with DVC and Feast did we proceed to evaluate the remaining tools, ensuring systematic consistency throughout the thesis.

3.3 Use Case Context: IDEKO

IDEKO is a Technology and Research Centre specializing in industrial production and manufacturing technologies. For this thesis, we built upon IDEKO's anomaly detection testbed which includes both their precision-grinder dataset and multiclass classification models for high-frequency time series analysis. While IDEKO's repository contains deprecated binary classification code and experimental

Table 2: Evaluation Framework

Criterion	Definition	Low	Moderate	High
USABILITY				
Setup & Installation Simplicity	Time from a clean machine to running the vendor-supplied Hello-World example.	The process takes more than one hour. It requires many dependencies and manual steps.	The process finishes in ten to sixty minutes. It involves several standard steps.	A single command or guided wizard completes installation. The example runs within minutes with minimal prerequisites.
Configuration Simplicity	Effort needed to configure data sources, pipelines and parameters.	Configuration spreads across multiple files. Trial and error is required.	Configuration needs a few edits in one location or user-interface form.	Defaults work out of the box. A single YAML file or wizard captures optional tweaks.
Ease of Use	Learning curve for everyday tasks and overall intuitiveness.	The learning curve is steep. Specialist knowledge is required.	Users with standard ML or DevOps experience become productive after some practice.	Newcomers find the interface intuitive. Core actions are discoverable without prior expertise.
Documentation Support	Coverage, clarity and freshness of documentation, tutorials and community answers.	Documentation is sparse or outdated. Community questions are rarely answered.	Core tasks are documented. The community eventually responds to most questions.	Documentation is comprehensive and current. Community channels answer questions quickly.
FUNCTIONALITY				
Functional Appropriateness	Alignment of built-in functions with real workflow needs.	Workflows feel awkward and misaligned with real practice.	Tasks are achievable with minor workarounds.	Features enable streamlined and intuitive workflows.
Functional Completeness	Breadth of functions that cover all target tasks and objectives.	Key capabilities are missing and standard tasks cannot be finished.	The tool covers the required tasks with occasional gaps.	The feature set is complete and aligns fully with workflows.
Reliability	Stability and fault-tolerance under expected workloads.	Failures occur frequently and the system is unstable under load.	Occasional issues appear but the system recovers automatically.	The system remains stable under expected conditions with robust error handling.
FLEXIBILITY				
Platform Support	Coverage of operating systems, distributions, architectures and deployment modes.	The tool supports only one operating system, distribution or deployment model.	Major operating systems and common deployment options are supported.	The tool supports many operating systems, architectures and deployment models.
Integration Readiness	Availability of pre-built integrations or a plugin architecture.	No built-in integrations exist and heavy custom code is required.	The tool integrates with common ecosystem tools and needs some custom work for others.	Many integrations are available and a robust plugin architecture supports extensions.
Ease of Integration	Effort needed to embed the tool in an existing stack or workflow.	Tight coupling makes it difficult to replace or extend components.	The architecture offers some flexibility through documented extension points.	The architecture is highly modular and components communicate through clear APIs.
Modularity	Ability to use subsets of features independently.	Users must adopt the entire suite even for simple tasks.	Some components work standalone while others remain tightly coupled.	Each component works on its own with minimal coupling.
VITALITY				
Community Support & Adoption	Size and activity of the user base, forums and ecosystem contributions.	The community is almost nonexistent and user activity is minimal.	A modest following exists with occasional contributions.	A large and vibrant community drives widespread adoption.
Maturity	Project age and API stability.	The project is less than one year old and breaking changes are common.	The project is two to five years old and APIs are mostly stable.	The project is older than five years and stability is proven.
Active Development & Maintenance	Frequency of commits and releases and responsiveness to issues.	Little or no recent development and maintainers do not respond to issues.	Releases are infrequent but the project is still maintained.	Commits occur frequently and maintainers respond promptly to issues.

auto-ensemble notebooks, our work exclusively focused on their multiclass classification models, which offer different architectures including dense, convolutional, recurrent, and LSTM networks.

The IDEKO use case centers on building an anomaly detection model for a precision grinding machine's Y-axis hysteresis tests. During their daily workload, the Computer Numerical Control (CNC) machine executes a series of backward-forward motions under programmatic control, generating high-frequency sensor readings (500 Hz) of four synchronized signals:

- f_1: encoder position
- f_2: external ruler position
- f_3: motor current/intensity
- f_4: commanded position

The dataset contains movement cycles collected under three known machine conditions:

- *No anomaly*
- *Mechanical failure* (e.g., bearing or screw damage)
- *Electrical failure* (e.g., motor or drive faults)

The available data captures only extreme states, fully healthy or already failed, making simple statistical features sufficient to distinguish these clear cases. However, IDEKO's primary interest lies in detecting the subtle degradation phase that precedes outright failure. In this phase, statistical features often lack the sensitivity and may drift with ambient conditions such as temperature or lubrication. Instead IDEKO treats each cycle as a multivariate time series and leverages deep learning to identify temporal patterns and detect early signs of wear that enable timely maintenance, thus reducing unplanned downtime and preventing defective parts.

3.3.1 Provided IDEKO Scripts

The IDEKO package we build on is minimal. It contains:

- raw CSV dataset of high-frequency signals (encoder, ruler, current, commanded position),
- preprocessing python scripts for padding/truncation and label encoding, and
- a Python Keras training script that instantiates and trains models based on a simple YAML configuration (NN, LSTM, RNN, CNN).

3.4 Tool Selection and Rationale

The component selections in Table 1 were based on our SLR findings [2]. We centered our stacks around MLflow and Kubeflow as they emerged as the most popular MLOps ecosystems in our SLR. For components offered directly by these ecosystems, we used their native tools (e.g., MLflow Tracking, MLflow Models, Kubeflow Pipelines, MLMD). For components that the ecosystems did not

provide, we selected the most popular standalone tools from our SLR. For instance, since MLflow lacks a native orchestrator, we chose Airflow as it was the most popular orchestration tool after Kubeflow Pipelines. Similarly, we selected DVC for data versioning in the MLflow stack as the most widely adopted data versioning tool. The specific rationale for each tool selection is detailed in Table 1.

3.5 Evaluation Framework

The evaluation framework, consisting of four dimensions: Usability, Functionality, Flexibility, and Vitality, was finalized after iterative refinement during the pilot study discussed in Section 3.2.1. Each dimension contains multiple criteria that assess different aspects of tool quality. The framework was developed using a bottom-up approach during the pilot study's tool evaluation. It was partly inspired by ISO/IEC 25010 [29], an international standard that defines quality models for software and systems, addressing both product quality and quality in use. To ensure our evaluation is objective and systematic, each criterion has specific requirements, listed in Table 2, which define what constitutes Low, Moderate, and High performance levels. Although we initially explored more granular scoring systems, this simpler three-level approach proved to be a good balance between precision and practical assessment. All criteria were examined based on our experience implementing the IDEKO use case, ensuring the evaluations reflected real-world usage.

4 Results

This section presents our findings from evaluating individual MLOps tools and comparing the complete MLflow-centric and Kubeflow-centric stacks in the IDEKO anomaly detection pipeline.

Notably, the Kubeflow evaluation was conducted using the Charmed Kubeflow distribution [27], as detailed in Section 4.1.3.1, which fundamentally differs from the MLflow stack's architecture. While the MLflow stack required manual installation and configuration of each individual tool (DVC, Airflow, Optuna, Feast), Kubeflow was deployed as an integrated platform that automatically provisioned approximately 25 Kubernetes pods. This deployment included pre-installed and pre-wired components: Kubeflow Pipelines, Katib, ML Metadata (MLMD), along with supporting infrastructure pods for MySQL (metadata storage), MinIO (object storage), and Dex (authentication), among others. These components were automatically integrated with each other, sharing credentials, storage backends, and network configurations. The choice to use a Kubeflow distribution rather than a raw Kubeflow manifest is further explored in Section 4.1.3.1.

4.1 RQ1: Comparing MLOps Tools

This section presents the evaluation results for individual MLOps tools across both stacks. As outlined in Table 1, eight component types structure our evaluation, each analyzed in the following subsections. They follow a consistent format with a systematic test plan table summarizing our implementation experience, a discussion of key findings followed by a comparative analysis between functionally equivalent tools. The complete systematic test procedures and detailed scoring rationales are provided in the Appendix. Tables 16 and 17 at the end of this section consolidate all evaluation scores across the four dimensions (usability, functionality, flexibility, and vitality) for the MLflow and Kubeflow stacks respectively.

4.1.1 Data Versioning Tools

These tools version and store datasets, which, in the IDEKO use case, consist of high-frequency sensor readings collected from the CNC machine.

DVC

Table 3 presents the systematic evaluation of DVC's functionality across seven test scenarios, from installation through cross-platform deployment.

Table 3: DVC's systematic test plan and results for IDEKO dataset versioning

#	Test Step	Result
1	Install DVC and configure local MinIO [30] storage as the backend storage	Installation required only a single `pip` command with backend-specific variants available for S3, Azure, GCS, and so on, and a few other commands to configure the MinIO connection.
2	Initialize DVC and add the IDEKO dataset	The command `dvc init` created the `.dvc` directory within the existing `.git` repo. Then `dvc add data/` generated a pointer file with an MD5 hash of the tracked CSV files. DVC automatically added the actual data directory to `.gitignore`, preventing accidental commits of large files to Git.
3	Create branches for different dataset versions	Feature branches and dataset modifications worked as expected.
4	Modify the dataset on a feature branch by removing some columns and commit	After removing columns from a CSV file, `dvc add` updated the pointer file with new hashes. Then, `dvc push` uploaded modified data to the MinIO backend, followed by standard Git commits for the metadata changes.
5	Test rollback functionality to previous dataset versions	Rollback created a detached HEAD, which resulted in merge conflicts in the `.dvc` files. Attempting to commit from this state caused merge conflicts in the `.dvc` pointer files. The workaround involved creating a new branch from the old commit and then merging it into the working branch.
6	Merge branches containing different dataset modifications	Merging branches with non-identical datasets produced the expected `.dvc` file conflicts. Standard Git merge conflict resolution was applied, although identifying the correct dataset version required examining MD5 hashes, rather than meaningful diffs.
7	Clone on macOS for cross-platform test	The command `dvc pull` retrieved an identical dataset. Hash verification confirmed reproducibility.

DVC successfully extends Git's versioning capabilities to large files, maintaining Git's familiar workflow while handling data that would overwhelm standard repositories. This minimize the learning curve for developers already versed in Git, though practitioners without Git experience face a steeper learning curve. The pointer file approach, adopted by DVC, elegantly separates metadata from data storage, enabling abstraction of backend choices. However, resolving merge conflicts in `.dvc` files is not as straightforward as resolving source code conflicts since comparing the MD5 hashes of files only indicates that the datasets are different, unlike code diffs, which allow you to see the code changes directly.

There is no free Graphical User Interface (GUI) included. Instead, Iterative, the company behind DVC, offers DVC Studio, a paid enterprise tool that contrasts with Git's ecosystem of free visual tools. This may limit adoption among GUI-preferring users and hurt DVC's usability scores.

For the IDEKO use case, DVC adequately versions sensor data and model artifacts. The storage integration worked reliably after initial configuration, and its cross-platform support enabled collaboration across different environments. Teams comfortable with command-line Git will find DVC's learning curve minimal, while those seeking visual tools or simplified workflows may struggle with the additional complexity.

> **Main Findings:** DVC's Git-like interface minimizes learning curves for developers, but merge conflicts in pointer files only show hash differences rather than meaningful diffs, making conflict resolution guesswork. Teams should maintain clear dataset naming conventions and branch documentation to mitigate this limitation.

LakeFS

Table 4 details the evaluation of LakeFS for versioning the IDEKO dataset, highlighting both its intuitive User Interface (UI) and infrastructure complexity.

LakeFS delivers on its promise of being a "Git for data" [31], but it comes with important infrastructure considerations. Noticeably, the quickstart experience masks production complexity. For production deployment, the three containerised bundled components from the quickstart command: PostgreSQL for metadata, MinIO for object storage, and lakeFS itself, must all be replaced with persistent, production-grade alternatives such as Kubernetes deployments, thereby significantly increasing deployment and configuration complexity.

The web UI provides intuitive visual repository management with drag-and-drop functionality and clear branch visualization. Multiple merge options (CLI, UI pull requests, REST API) contribute to its usability scores by accommodating different workflows. Additionally, branching operations avoid full data duplication, providing significant performance advantages over simpler approaches.

However, the lakectl credential setup revealed poor error messaging that complicated initial configuration where a simple typo in credentials led to extensive debugging due to unhelpful error messages. LakeFS excels for teams already operating on cloud object storage who need sophisticated version control capabilities, modelling its philosophy on Git with familiar branches and commits. Nevertheless, Its multi component architecture introduces significant setup overhead compared to DVC, which integrates into existing Git repositories without requiring separate components.

Table 4: LakeFS' systematic test plan and results for IDEKO dataset management

#	Test Step	Result
1	Install LakeFS using the quickstart setup	Installed via `pip install lakefs` followed by a quickstart command (`python -m lakefs.quickstart`) which deployed the required components (LakeFS, MinIO, Postgres) as Docker containers. An admin user is automatically created and the web UI is also launched.
2	Create a repository with the IDEKO dataset	Repository creation and dataset upload worked smoothly through the simple integrated drag-and-drop UI.
3	Configure the lakefs CLI management tool (lakectl) to interact with the LakeFS Kubernetes pod	Configuring lakectl revealed authentication issues. Connection failed repeatedly with unhelpful error messages. After extensive debugging, we traced the issue to a mistyped secret key. However, the error messages provided no useful hints about the actual problem.
4	Create branches for different dataset versions	Branching functioned without requiring full data duplication, offering significant performance advantages. Used simple Git-like commands such as `lakefs diff`, which clearly showed changes between commits.
5	Perform rollback operations to previous commits	Rollback using `lakefs branch reset` successfully reverted changes.
6	Merge branches containing different dataset modifications	Merging branches worked as expected and could be done through CLI commands, UI pull requests, or REST API calls.

LakeFS vs DVC

DVC appears in 25 of our 41 primary studies, making it the most referenced data versioning tool in our SLR [2]. In contrast, LakeFS received fewer mentions, suggesting lower adoption in academic literature.

The tools differ fundamentally in their architecture. DVC operates solely as a client-side tool that stores metadata locally while the actual data resides in separate storage hosts (S3, Azure, local filesystem). This results in a simple-to-install tool for developers who simply install DVC and point it to existing storage.

On the other hand, LakeFS requires a more cumbersome multi-component deployment: a server to host LakeFS, a dedicated database for metadata, and object storage. While the quickstart provides pre-configured Docker containers with bundled MinIO storage, production deployments demand separate provisioning of each component.

This architectural complexity enables different capabilities. During our evaluation, LakeFS demonstrated zero-copy imports from existing S3 buckets, a capability that could be useful for organizations with large storage needs.

The user experience reflects these architectural choices. DVC's Git-like commands felt familiar but introduced merge conflicts in pointer files during rollbacks. LakeFS avoided conflicts through its copy-on-write branching, but frustrated us with its Kubernetes pods debugging. However, LakeFS's web UI significantly improves usability compared to DVC's command-line focus, offering effortless visual repository management, drag-and-drop uploads, and branch visualization, all of which DVC lacks. Notably, DVC does offer DVC Studio, however it is not included in DVC's open-source offering and it is a paid commercial tool.

For our evaluation, which was conducted on a single machine, DVC's Git integration provided a gentler learning curve and minimal setup overhead. LakeFS's distributed architecture and Kubernetes compatibility would become advantageous only at large-scale data lakes across large teams. For smaller teams, such as those working on the IDEKO case, LakeFS's infrastructure overhead may not be justified unless they specifically need Kubernetes features or LakeFS-specific capabilities, such as branch-level access control.

> **Main Findings:** LakeFS delivers production-grade versioning with an intuitive drag-and-drop UI, but its quickstart deployment masks significant infrastructure complexity. Production deployments require dedicated infrastructure components, making it excessive for projects not already operating on cloud object storage.

4.1.2 Feature Store

Feature stores maintain computed features from raw datasets, ensuring consistency between training and serving environments. In the IDEKO pipeline, Feast stores derived features such as rolling averages, statistical aggregates, and anomaly scores computed from the raw sensor readings, thereby preventing training-serving skew.

Feast

Table 5 shows the systematic evaluation of Feast for managing IDEKO's computed features and preventing schema drift.

Feast provides critical schema management that prevents runtime failures, a capability missing from versioning tools such as those mentioned in Section 4.1.1. The Feast registry holds every revision of each feature definition, serving as the authoritative record for feature versions. If we relied only on files tracked by DVC and later renamed a column or file, the training code would still look for the old feature name, fail at runtime, and DVC would provide no warning. Feast prevents this because the command `feast apply` compares new feature definitions with the registry, shows a schema diff, and refuses to deploy until you confirm the change.

Table 5: Feast's systematic test plan and results for IDEKO feature management

#	Test Step	Result
1	Install Feast	Installation succeeded on the first attempt using a single command.
2	Initialize a new feature repository and configure the backend feature store	The command `feast init` was used to set up an example repository structure with all the required files. These included `feature_store.yaml` (backend storage and project settings) and `features.py` (feature schemas and data types).
3	Register IDEKO dataset features with Feast	Feast requires Parquet files [12], a columnar storage format optimized for analytical workloads and offering better performance than CSV. Since IDEKO's dataset format is CSV, we had to add a script to convert data into Feast-compatible Parquet before registration.
4	Define and register initial feature schemas	Configuring Feast involved modifying multiple files, which resulted in a moderate learning curve.
5	Test schema evolution by updating a feature definition	Modified a feature definition by updating the rolling average window size from 10 to 20. Then ran `feast apply`, which displayed the exact schema diff, blocked deployment until approval, and logged changes after confirmation.

Feast involves greater up-front configuration effort than working directly with raw Parquet or CSV files, resulting in lower usability scores. This effort may be justified for large teams where guarding against schema drift is essential, but it may impose too much overhead for smaller teams.

Feast's native Airflow and Kubeflow providers boost flexibility scores, enabling seamless integration with common orchestration tools. For the IDEKO use case, we utilized only offline features since the project runs batch-only experiments. The `feast materialize` command, which synchronizes features to an online store for low-latency serving, was unnecessary for our batch processing needs. Fully leveraging Feast's capabilities would require streaming data rather than batch processing, which is beyond the scope of our use case. Feast has established itself as a leading feature store according to our SLR [2].

> **Main Findings:** Feast enforces identical feature definitions across training and serving by validating changes against its registry, however the associated overhead may be disproportionate for smaller teams or use cases.

4.1.3 Orchestration

Orchestrators coordinate the execution of ML pipeline components, schedule and monitor workflows that fetch datasets, train models, and deploy them to production. In the IDEKO pipeline, the

orchestrator manages the entire workflow from DVC/LakeFS data retrieval through feature processing, model training with HPO, experiment tracking, and finally model serving.

Airflow

Table 6 presents the evaluation of Apache Airflow for orchestrating the IDEKO pipeline components.

Table 6: Airflow's systematic test plan and results for IDEKO pipeline orchestration

#	Test Step	Result
1	Install and configure Airflow for Python	Successfully installed via `pip` using a constraints file supplied by Airflow. However, it required more effort than a single-command setup since multiple environment variables needed to be configured. Airflow's constraints file, which locks Python dependencies to fixed versions, pinned the separately-installed Feast to an older release without the required CLI/SDK, thereby requiring manual override. Airflow came with an `airflow standalone` command that initialized all the required components, including the database, and created a user.
2	Create workflow for IDEKO case to pull data, train, register and serve model with MLflow	Writing the first pipeline required learning DAG structure, task/operator types, and Jinja templating (for dynamic workflow parameters), creating a moderate to steep learning curve. Triggering the first DAG produced numerous import errors, which required repeated updates to environment variables, paths, and DAG definitions to fix.
3	Examine workflow views and functionality through the dashboard	The UI displayed comprehensive graph and grid views, SLA charts (service monitoring), and verbose logs that contained significant noise. DAGs cannot be defined in the UI, but the dashboard provides comprehensive visibility into pipeline health and status.
4	Evaluate different workflow scheduling triggers	Pipeline triggers worked as expected with cron-style syntax and manual triggering was easily implemented via UI.
5	Simulate task failure to evaluate automatic retry mechanism	Scheduler auto-restarted and retried failed tasks by default as configured.
6	Kill scheduler component and verify recovery behaviour	Airflow's process successfully recovered when scheduler was killed.

Airflow workflows are defined entirely in Python, providing flexibility but requiring developers to learn Airflow-specific Directed Acyclic Graph (DAG) syntax and domain-specific operators. The documentation, although exhaustive, is poorly organized, providing scattered reference pages instead of guided workflows and offering only minimal tutorials. The documentation's non-linear structure slowed initial deployment and contributed to both the steep learning curve and lower usability scores. The platform logs every DAG run, task instance, and event comprehensively, providing excellent observability but at the cost of verbose and sometimes noisy logs.

The installation complexity stems from Airflow's dependency management approach. The constraints file ensures compatibility, but conflicts with other project requirements may lower flexibility scores, as seen when Feast required a manual override. During evaluation, the `airflow standalone` command proved invaluable for quick experiments, as it launched a preconfigured database, web server, scheduler, and default user in seconds. However, Apache's documentation recommends splitting these components and managing them individually in production, adding significant operational overhead. This gap between development simplicity and production complexity is a recurring theme among MLOps tools.

Airflow's UI effectively visualizes pipeline execution but it does not allow DAGs to be defined through the interface, requiring all workflow definitions to be written in code. Despite these usability challenges, Airflow's maturity, reliability (such as its automatic retry mechanisms), and widespread adoption make it a solid choice for teams willing to overcome the initial learning curve.

> **Main Findings:** Airflow's Python-based DAGs and comprehensive UI provide powerful orchestration, but its dependency management can conflict with other project requirements as seen with Feast. The local development setup masks the multi-component architecture required for production deployments.

Kubeflow Pipelines

Table 7 documents the complex evaluation of Kubeflow Pipelines (KFP), revealing both powerful capabilities and significant integration challenges.

KFP offers powerful orchestration capabilities but demands substantial Kubernetes expertise, resulting in a steep learning curve that impacts usability scores. A pipeline consists of discrete components, each running in its own container in isolation, without shared storage. This design ensures reproducibility and prevents cross-component interference but necessitates explicit input and output declarations to pass data between stages. Simple workflows, such as using Git-cloned code in subsequent training components, became unnecessarily complex due to this isolation model. These architectural choices create both benefits and friction points throughout the development workflow.

The fragmented ecosystem proved challenging from the start. We evaluated two KFP installation options, DeployKF and Charmed Kubeflow, with each distribution imposing its own architectural decisions. Both distributions include MinIO—an S3-compatible object storage server used for storing pipeline artifacts and datasets—though it was not prominently featured in our SLR as it typically serves as background infrastructure rather than a primary MLOps tool. DeployKF pre-configured three groups with corresponding Kubernetes namespaces (team1, team2, and admin), while Charmed Kubeflow

Table 7: Kubeflow Pipelines' systematic test plan and results for IDEKO containerized orchestration

#	Test Step	Result
1	Install and setup Kubeflow Platform (includes Kubeflow Pipelines)	Multiple installation options complicated the selection process as there was no obvious default. We chose Charmed Kubeflow [27] which required Ubuntu 22 (Linux distribution) [32], MicroK8s (lightweight Kubernetes) [33], and Juju (Canonical's application lifecycle manager) [34] as prerequisites [27]. The deployment took 15 to 20 minutes to provision approximately 25 services, and port forwarding required namespace and service names that differed from the documentation.
2	Create a user account through Dex (OpenID Connect provider)	Dex authentication worked with the default credentials, but the user and namespace structure varied significantly between Kubeflow distributions, with documentation failing to clarify these distribution-specific architectural decisions.
3	Create a pipeline that clones the code, pulls the dataset, trains and registers the model.	Initial pipeline creation encountered SQL character set incompatibilities between the KFP SDK and deployment. While LakeFS was our selected tool, we opportunistically tested DVC integration since our data was already versioned in it from the MLflow stack evaluation. DVC integration revealed critical credential conflicts: both DVC and Kubeflow's MinIO use identical AWS S3 credential formats but in different namespaces, causing hours of debugging. LakeFS worked immediately as its Helm chart includes a pre-configured MinIO instance, avoiding these conflicts. However, this finding highlights a key limitation: quickstart deployments mask production complexity where LakeFS would face similar credential challenges when using external object storage. The UI could observe but not define pipelines.
4	Test the different pipeline upload methods	Manual upload required compiling the python Kubeflow script to a Kubeflow compatible YAML format, creating a pipeline and an experiment, and running them separately, making iterative development cumbersome. Programmatic submission streamlined the process but still required authentication setup. Alternatives such as Kubeflow's built-in Jupyter notebooks or VS Code simplified authentication but required us to abandon our customary IDEs along with their familiar tooling and extensions.
5	Examine logs and debug information through UI and `kubectl`	Logging visibility was poor since only the last failed component's logs appeared in the UI, and viewing previous components required kubectl commands through the terminal.
6	Re-run identical pipelines to verify caching behaviour	Caching functioned correctly. However, failed runs were also cached, requiring manual invalidation through code changes, which proved difficult due to undocumented syntax differences between KFP SDK v1 and v2.
7	Force component pod failures to test retry and recovery mechanisms	KFP detected terminated pods and automatically restarted them, causing the pipeline to pause at the failed component and resume once recovery was complete, demonstrating robust fault tolerance.

created only a single admin namespace. These variations are not part of core Kubeflow but rather packager-specific choices that significantly impact user experience. Creating a functional user in DeployKF required understanding the pre-built group hierarchy and navigating cross-namespace permissions, such as mounting MinIO secrets (storage credentials) from the Deploykf-MinIO namespace into pipelines running in the team1 namespace.

Credential management emerged as a significant challenge, particularly when integrating data versioning tools. While LakeFS was our planned tool for the Kubeflow stack, we opportunistically tested DVC integration since our IDEKO dataset was already versioned in it from the MLflow stack evaluation. This unplanned test revealed critical insights: DVC and Kubeflow's integrated MinIO both use S3-compatible storage with identical AWS credential formats but expect them in different namespaces, creating authentication failures that took hours to debug. The DeployKF distribution further complicated this by storing credentials in parent namespaces, requiring complex mounting and DNS aliasing.

LakeFS initially appeared superior because its Helm chart includes a pre-wired MinIO instance, allowing pipeline components to work immediately without credential configuration. However, this exposes a crucial limitation of our evaluation: quickstart deployments can mask production complexity. In a production deployment where LakeFS would connect to external object storage rather than its bundled MinIO, teams would likely encounter the same credential management nightmares we experienced with DVC. This finding reinforces that our quickstart-based evaluation may underestimate the operational complexity teams face in production environments.

The platform's complexity is evident from the start, with numerous deployment options and no clear default, challenges with namespace, permission, and credential management, and a steep learning curve for both Kubernetes and Kubeflow-specific concepts. Although the built-in Jupyter notebook environments simplify authentication and the visual DAG representation makes pipelines easier to understand, these benefits do not offset the operational overhead for most teams.

The poor logging visibility, caching quirks, and UI limitations collectively reduce KFP's debugging efficiency and developer productivity. More critically, infrastructure issues such as the following can derail entire deployments. Kubeflow's MySQL pod suddenly started consuming 16GB of RAM regardless of configuration attempts. Despite updating Juju's charm configurations and trying to modify the MySQL pod's resource limits directly, it consistently respawned with the same 16GB allocation. This brought our test machine (32GB total) to a crawl and ultimately forced complete cluster deletion and recreation.

Our findings indicate that large organizations, such as Spotify [35], select Kubeflow because they have the capacity to manage its complexity, and they require a Kubernetes native solution at a very large scale. Conversely, smaller teams that lack dedicated operations support or a genuine requirement for high availability should not adopt this platform.

Notably, Kubeflow offers enterprise-level scaling capabilities such as distributed training and cross-data centre deployment. However, testing these capabilities exceeds the scope of this study. Teams

that cannot leverage them incur substantial usability costs for minimal benefit.

> **Main Findings:** KFP's containerized components ensure reproducibility and cloud-native scaling, but credential management and namespace conflicts between integrated tools can create substantial debugging challenges. Teams should only adopt KFP if they have deep Kubernetes expertise or genuine need for distributed orchestration.

4.1.3.1 *Kubeflow Distributions*

The first obstacle arises at installation, since Kubeflow only provides raw manifests and relies on third parties to package the stack, with no official installer available. As a result, organizations often package Kubeflow into distributions, each with different tools, versions, and target platforms. Cloud providers offer platform-specific versions, including Azure Kubernetes Service, IBM Cloud Kubernetes Service and Google Kubernetes Engine, whereas only two generic distributions exist for local deployment: DeployKF and Charmed Kubeflow.

DeployKF bundles Kubeflow tools with Airflow and MLflow into a single deployment [36], pre-configuring users, MinIO storage, and ArgoCD, making it an opinionated ML platform rather than a pure Kubeflow distribution. However, DeployKF lacks critical components like KServe and Kubeflow Model Registry, limiting its capabilities for model serving and versioning. Charmed Kubeflow, maintained by Canonical, provides the latest Kubeflow version but depends on other Canonical software such as Ubuntu 22, MicroK8s, and Juju. In contrast, DeployKF does not tie users to a Kubernetes-specific variant or a specific operating system.

While Charmed Kubeflow comes with KServe pre-integrated, it notably omits Kubeflow Model Registry and instead promotes MLflow integration for model registry capabilities [27]. One advantage of both distributions is that they include pre-wired Kubeflow components, reducing integration complexity compared to manual setup. Raw Kubeflow manifests exist, but according to the documentation, they are "*intended to be used by users with Kubernetes knowledge and as the base of packaged distributions*", suggesting they target experienced Kubernetes practitioners rather than typical end users.

We ultimately selected Charmed Kubeflow, despite DeployKF appearing simpler, because it included the latest Kubeflow version with KServe pre-integrated, which was necessary for our evaluation. However, this decision came with future costs as the MySQL component consumed 16GB of RAM with no clear way to reduce it, and the Canonical-specific tooling introduced another consideration for teams not using Canonical's software. Canonical is quite controversial in the open source community because its proprietary Snap package manager is required to install the other Canonical software, such as MicroK8s and Juju [37, 27]. This distribution fragmentation forces teams to choose between outdated but convenient packages or current yet complex installations, with no clear "default" Kubeflow experience.

4.1.3.2 *Airflow vs Kubeflow Pipelines*

The contrast between Airflow and Kubeflow Pipelines reveals fundamentally different approaches to workflow orchestration. Airflow's installation requires a single `pip` command and runs on standard Python infrastructure, while Kubeflow demands Kubernetes expertise, multiple prerequisite tools, and approximately 25 deployed services. The difference in complexity between Airflow and Kubeflow extends throughout the user experience. Despite their domain-specific syntax, Airflow's Python-based DAG definitions remain accessible to data scientists familiar with Python. Meanwhile, Kubeflow's containerized components require understanding Docker, Kubernetes pods, and complex artifact passing between isolated pipeline components that complicate debugging.

Documentation quality is comparable between the two platforms, but in different ways. Airflow documentation is centralized on Apache's site, yet it suffers from scattered, non-linear organization and just a few step-by-step tutorials. Contrastingly, Kubeflow's documentation is fragmented across the main project site, distribution-specific pages, and individual component repositories. Although Kubeflow's core documentation is comprehensive and well-structured, users must piece together information from multiple sources, which further steepens the learning curve for teams evaluating deployment options.

Both platforms offer similar core features such as visual DAG representations, pipeline caching, automatic retries upon failure, and various triggering mechanisms such as cron-based, manual, and code hooks. However, these platforms are not aimed at the same audiences. Kubeflow primarily targets teams on a Kubernetes architecture requiring massive scale, as demonstrated by Spotify's production Kubeflow pipeline [35]. However, for the purposes of this study, the massive operational overhead and need for specialized knowledge make it unsuitable for smaller teams. On the other hand, Airflow excels at cluster deployments where simplicity outweighs infinite scalability. Notably, our evaluation did not test Airflow's or Kubeflow's scalability in this evaluation.

The platforms' ecosystem scope also differs significantly. Airflow provides a pure orchestration tool that integrates with external systems through operators, namely the Python and Bash operators. Kubeflow provides an entire ML platform with optionally pre-integrated tools that enable experiment tracking (MLMD), hyperparameter tuning (Katib), and model serving (KServe). This comprehensiveness appeals to practitioners seeking a unified solution, whereas Airflow relies on users integrating their own tools, which may require additional configuration in multi-pod production deployments. However, this was not an issue on our single-host machine as it could easily run Bash or Python operators alongside other tools. Nevertheless, Airflow's configuration complexity would increase in a multi-machine deployment.

4.1.4 Experiment Tracking

Experiment tracking tools store metadata about ML experiments, including hyperparameters, metrics, model versions, and datasets. In the IDEKO pipeline, these tools track which combinations of model architectures (NN, LSTM, CNN, RNN), hyperparameters, and training configurations produced the best classification accuracy

for detecting anomaly types (electrical anomaly, mechanical failure, or no anomaly).

MLflow Tracking

Table 8 shows our evaluation of MLflow Tracking for managing IDEKO experiment metadata.

Table 8: MLflow Tracking's systematic test plan and results for IDEKO experiment management

#	Test Step	Result
1	Install MLflow via `pip`	Straightforward single command installation through `pip` without any dependency conflicts or version issues.
2	Setup local MLflow server and dashboard	Launching the MLflow tracking server and web dashboard required two simple shell commands with each needing only a port definition, after which the UI was immediately accessible at localhost.
3	Register IDEKO training runs as separate MLflow experiments	To enable manual tracking, we wrapped the training code in `mlflow.start_run()` context blocks and specified which parameters to log. This required only minor modifications to existing scripts to capture custom metrics and parameters alongside the automatic logging.
4	Enable automatic experiment tracking	MLflow's auto logging functionality for Keras captured training metrics (loss, accuracy), hyperparameters (learning rate, batch size, optimizer settings), and the trained model requiring no additional code [23].
5	Track experiment metadata and custom metrics	Custom metrics and tags were successfully logged to organize and identify different experimental configurations, with all metadata properly associated with their respective runs.
6	View what has been captured in the UI and examine available functionalities	The intuitive MLflow dashboard displayed every training run with rich visualizations, including extensive graphs of metrics over time and clear links between each training run, its input dataset and parameters and the resulting output model. It also provided advanced filtering capabilities by date range, custom tags, or specific metric thresholds.

MLflow Tracking excels in developer experience because of its thoughtful design choices. Its documentation begins with a straightforward five-minute quickstart guide that brings you up to speed in a linear fashion. Additionally, the documentation includes an AI chatbot that can speed up troubleshooting by suggesting code snippets and linking directly to relevant pages. MLflow Tracking's concepts and module naming are intuitive, avoiding the need to learn new abstractions that plague other MLOps tools such as KFP. Furthermore, the quick start guide explicitly recommends using smaller tracking blocks rather than wrapping entire training scripts in a single `mlflow.start_run()` block, as failures in large blocks require manual cleanup.

MLflow's auto logging feature eliminates boilerplate code for supported frameworks like Keras, reducing the code needed to track experiments comprehensively. However, teams using unsupported frameworks must resort to explicit logging calls, which may become tedious. The platform also offers optional system-metrics logging and custom tracing spans for teams needing deeper observability. Tight integration with other MLflow modules, such as MLflow Models and MLflow Datasets, creates a cohesive ecosystem that streamlines end-to-end workflows. However, this integration can be a double-edged sword when workflows must incorporate non-MLflow components, impacting the flexibility score.

The combination of minimal setup effort, comprehensive automatic tracking, and an intuitive UI contributes to high usability scores. The active development and extensive framework support underpin strong vitality ratings, making MLflow Tracking a solid choice for teams seeking low-friction experiment management.

> **Main Findings:** MLflow Tracking stands out with automatic parameter capture for supported frameworks, intuitive UI, and class-leading documentation including an AI chatbot. Its minimal setup effort and comprehensive tracking capabilities make it the most accessible tool among all evaluated options.

ML Metadata

Table 9 shows the challenging evaluation of ML Metadata (MLMD) for metadata storage in the Kubeflow ecosystem.

MLMD functions as a metadata storage system consumed by other tools rather than as a user-facing experiment tracking platform. This became immediately apparent when attempting to view the stored metadata. While the Kubeflow dashboard displays a lineage graph by reading from the MLMD datastore, MLMD itself provides no visualization capabilities. Users must retrieve metadata through code or external tools. In our evaluation, accessing the metadata required navigating through Kubernetes pods to reach the MinIO storage where it resides in SQL format. This complexity forced us to deploy a separate stand-alone MLMD instance purely for evaluation purposes.

In MLMD, every hyperparameter, metric, and artifact requires an upfront schema definition with strongly-typed properties. Unlike tools such as MLflow, which capture parameters automatically through autologging, MLMD demands explicit declaration of data structures and schemas for each type of metadata (datasets, models, training runs, experiments) before any information can be stored. This verbosity ensures consistent metadata structure but reduces usability for straightforward use cases like the IDEKO pipeline. A retrieval script was needed to successfully query these relationships programmatically. However, constructing even simple queries required understanding MLMD's custom property storage model. In fact, finding which hyperparameters were used in a specific run required multiple queries before the correct one was identified.

MLMD stores comprehensive lineage as a directed graph, capturing dataset versions, preprocessing, and hyperparameters. However, accessing this metadata proved consistently challenging. MLMD supports diverse storage backends, including SQLite for development and MySQL or PostgreSQL for production deployments, but

Table 9: MLMD's systematic test plan and results for Kubeflow metadata management

#	Test Step	Result
1	Install MLMD	MLMD was pre-installed in both Charmed Kubeflow and DeployKF distributions, storing metadata in the cluster's MySQL pod. However, accessing this data required navigating Kubernetes' abstractions and understanding MinIO (the S3-compatible object storage system) layers.
2	Update the training script to register metadata	Integrating MLMD into the IDEKO pipeline required explicitly defining schemas for every element we wanted to track. Unlike MLflow's automatic capture, MLMD demanded upfront declaration of data types for datasets, models, training runs, experiments, and each individual hyperparameter before any metadata could be stored.
3	Execute training and capture metadata	Initial attempts to inspect the stored metadata through Kubeflow's Kubernetes pods proved overly complex, leading to the installation of MLMD as a standalone library outside of Kubeflow with a local MySQL backend for better visibility into what was actually being stored.
4	Build a retrieval script to query saved metadata	Constructing queries required understanding MLMD's property-based storage model, where retrieving experiments with specific hyperparameter values required multiple API calls through specific paths.
5	Verify that each model is correctly linked to its source dataset	MLMD successfully tracked the complete workflow, including which dataset was used, what preprocessing steps were applied, and which model resulted from training on that dataset. This tracking captured every step and transformation in the pipeline, showing how raw data eventually became a trained model.
6	Examine metadata visibility and accessibility	MLMD does not offer visualization for the training run through the Kubeflow pipelines UI. On the other hand, direct database access in the standalone library revealed MLMD's complex internal schema with typed properties and graph-based relationships.

this flexibility comes at the cost of ease of use, since all interaction must occur through programmatic queries or direct SQL inspection.

Documentation challenges compounded these usability issues, as the official MLMD documentation links from GitHub often led to dead pages [38], while scattered references on the TensorFlow Extended website provided incomplete guidance [39]. MLMD's fragmented documentation may be a consequence of its maintenance handover from Kubeflow to the TFX team.

For teams using Kubeflow, MLMD provides the underlying storage that powers pipeline visualization and artifact tracking. However, as a standalone tool, it offers little beyond raw metadata storage, which explains why even Kubeflow distributions that include MLMD also provide integration guides for more user-friendly alternatives like MLflow with DeployKF bundling it directly [36].

> **Main Finding:** MLMD functions purely as a backend metadata store with no UI, requiring explicit schema definitions for every parameter and multiple API calls for simple queries. Even Kubeflow distributions acknowledge this limitation by bundling or recommending MLflow for user-facing experiment tracking.

4.1.4.1 MLFlow Tracking vs MLMD

MLflow Tracking and MLMD serve different purposes and target different users. MLflow started as a tracking module and has grown into a complete ecosystem. According to our SLR [2], it remains the most popular MLOps tool. It tracks model training metadata, datasets, model artifacts, system metrics, and custom traces. Additionally, its auto logging feature captures parameters automatically for supported frameworks like Keras and PyTorch, requiring no manual configuration.

MLMD functions as Kubeflow's backend metadata store. It tracks only the artifact metadata required for pipeline execution. Unlike MLflow, MLMD provides no user interface. All interaction happens through Python code or SQL queries.

The tools differ significantly in setup and usability. MLflow runs with a single command and includes an embedded database. Its dashboard shows all experiments, metrics, and models immediately. Contrastingly, MLMD requires an external database (SQLite, MySQL, or PostgreSQL) and offers no built-in visualization. Furthermore, creating metadata entries in MLMD requires explicit type definitions and property schemas. Notably, MLflow captures standard metrics automatically and accepts custom logging with minimal code.

Both Kubeflow distributions we tested recognize that MLMD alone cannot meet user needs for experiment tracking, acknowledging the huge gap in features between MLflow and MLMD. Charmed Kubeflow's introductory documentation provides a guide for adding MLflow to the stack, whilst DeployKF includes MLflow tracking pre-integrated alongside MLMD.

Based on our evaluation, teams needing experiment tracking should use MLflow. MLMD only makes sense as part of a full Kubeflow deployment where other components handle the user interface.

4.1.5 Model Registry

Model registries manage versioned models and their lineage. We use them to record which dataset, run, and hyperparameters produced each model, and to let serving components fetch the exact version for deployment.

MLflow Models

Table 10 presents the evaluation of MLflow Models for packaging and versioning IDEKO's trained neural networks.

Packaging a model using MLflow Models required minimal code and effort, demonstrating excellent integration between MLflow modules. Its unified UI keeps training runs and their resulting models tightly linked. Additionally, the automatic capture of

Table 10: MLflow Models' systematic test plan and results for IDEKO model packaging

#	Test Step	Result
1	Setup MLflow Models	The `mlflow.models` module was already included in the core MLflow package, requiring no additional installation beyond importing the package and the initial MLflow Tracking setup.
2	Save the trained Keras model as an MLflow Model	Calling `mlflow.pyfunc.log_model()` during training successfully automatically stored each model bundled with its runtime signature, input example schema, and the list of dependencies (requirements.txt).
3	Train multiple models with different metadata	Multiple training runs produced distinct model artifacts, with each artifact maintaining its own metadata, version information, and associated metrics from the training process.
4	View saved models in the UI Models tab and their lineage	The UI Models tab clearly displayed every saved model artifact, showing its originating run ID, training metrics, and model version, input parameters and dataset. This made it easy to track which experiment produced which model.
5	Test model loading for inference	Loading the test model with MLflow's Python function wrapper, `mlflow.pyfunc.log_model()`, which provides a standard interface for models, returned a ready-to-use callable model object for making predictions. Additionally, the custom class loader successfully incorporated IDEKO's feature engineering code into the model pipeline, allowing preprocessing to be bundled with the model.

`requirements.txt` ensures reproducibility without the need for manual dependency tracking.

The ability to use either the standard *pyfunc* loader or custom class loaders supports different deployment scenarios. The *pyfunc* loader (MLflow's Python function wrapper that provides a universal interface for models regardless of their framework) offers a lightweight, standardized approach that deploys cleanly to various serving tools, including MLflow Serving. However, it cannot handle complex feature pipelines by itself. Alternatively, the custom class loader option allows embedding custom preprocessing logic directly with the model.

> **Main Finding:** MLflow Models seamlessly captures model artifacts with dependencies and signatures during training, enabling single-command deployment later. The tight integration with MLflow Tracking eliminates additional configuration, demonstrating how ecosystem cohesion reduces integration friction.

KF Model Registry

The Kubeflow Model Registry, designed for model versioning and lifecycle management, could not be evaluated as it was absent from both the DeployKF and Charmed Kubeflow distributions. Without this component, Kubeflow relies on basic artifact storage through Kubeflow Artifacts, which lacks the versioning and metadata tracking capabilities that MLflow Models provides.

4.1.6 Dataset or Artifact Management

These tools version datasets and intermediate artifacts, then expose lineage from raw data to trained models so every run is traceable.

MLflow Datasets

Table 11 documents the evaluation of MLflow Datasets for tracking IDEKO dataset versions and their lineage.

Table 11: MLflow Datasets' systematic test plan and results for IDEKO data lineage tracking

#	Test Step	Result
1	Install MLflow Datasets	The MLflow Datasets library was included with the standard MLflow installation, requiring no additional packages or configuration beyond the base setup.
2	Launch MLflow server and access dashboard	Server startup remained identical to MLflow Tracking setup explained in Table 8. Using the MLflow Datasets API required an explicit import in Python scripts using `import mlflow.data`. Once imported, MLflow Datasets' functionality was immediately accessible through MLflow's existing dashboard interface.
3	Register IDEKO's CSV dataset	Each `mlflow.data.log_dataset()` call successfully captured IDEKO's dataset along with automatically generated metadata including file path, content hash, schema, record count, and timestamp.
4	Test dataset versioning after applying modifications	After modifying the dataset by adding rows and adjusting features, the subsequent `log_dataset()` call automatically created a new dataset version, with the metadata clearly reflecting all changes made.
5	Verify that each training run correctly links to its specific dataset version	When rerunning training scripts with different dataset versions, MLflow correctly associated each experiment training run with the specific dataset version it used, as clearly displayed in MLflow's UI.
6	Explore dataset lineage in the UI	The MLflow Datasets tab provided a comprehensive version comparison, showing schema differences between versions and enabling lineage tracking from any model or run to its exact training dataset.

Adding dataset logging using MLflow Datasets required only one extra API call per experiment, yet this simple addition made dataset origin explicit and discoverable through the MLflow UI. The dedicated UI views and automatic metadata capture greatly simplified audit trails and debugging. Notably, the ability to trace from a deployed model back to its exact training data snapshot addresses a critical gap in ML reproducibility.

Given that MLflow Datasets is built directly on top of MLflow Tracking rather than requiring a separate system, teams avoid the

complexity of managing multiple tools. However, this architectural design also means dataset management remains tied to the MLflow ecosystem, potentially limiting teams that need to integrate with existing data catalogues or versioning systems.

MLflow Datasets' version history and lineage features worked seamlessly and proved to be a natural and valuable extension to MLflow's tracking capabilities, requiring minimal code changes while providing high observability into the data lifecycle.

> **Main Findings:** MLflow Datasets requires only one additional API call to establish complete data lineage from model to training dataset version. This minimal effort provides crucial reproducibility often missing in ML pipelines, though it constrains teams to the MLflow ecosystem.

Kubeflow Artifacts

In Kubeflow Pipelines, artifacts are typed ML assets, for example datasets, models, and metrics. KFP stores them in an object store via the pipeline root, MinIO by default in common installs, and tracks their lineage in the Metadata store that the UI can visualize. This is background infrastructure for pipelines rather than an end-user tool, so we did not evaluate it.

4.1.7 Hyperparameter Optimization

Hyperparameter Optimization (HPO) tools automate the search for optimal model configurations by systematically exploring parameter spaces. In the IDEKO pipeline, these tools determine the best combination of learning rates, batch sizes, and network architectures to maximize classification accuracy for our anomaly detection model.

Optuna

Table 12 presents the evaluation of Optuna for optimizing IDEKO model hyperparameters as part of our MLFlow stack.

Optuna achieves an impressive balance between power and simplicity through its Python-native approach. Unlike other HPO libraries such as Katib, which require YAML configuration files, Optuna lets developers define search spaces directly in their training code. This is achieved using Python decorators and functions, keeping the optimization logic close to the model definition. The framework's native support for popular ML libraries including PyTorch, TensorFlow, Keras, Scikit-learn, XGBoost, and LightGBM, enables straightforward integration through wrapper functions. However, developers must still manually define which parameters to optimize and their search ranges, unlike AutoML tools, such as AutoGluon or AutoKeras, that analyze models to automatically determine tunable parameters and reasonable ranges.

MLflow's integration with Optuna revealed the following differences in parameter handling. MLflow's parameter immutability, intended to ensure experiment reproducibility, clashes with Optuna's iterative trial approach. While this initially caused frustration due to rejected parameters, Optuna provides a separate `optuna-integration` package featuring an `MLflowCallback` that resolves the conflict. This demonstrates mature ecosystem thinking, as the integration package supports multiple tools beyond MLflow, including TensorBoard, scikit-learn, and Weights & Biases.

Table 12: Optuna's systematic test plan and results for IDEKO HPO

#	Test Step	Result
1	Install Optuna and the optional dashboard	Installation completed with a single `pip` command for the core library. The dashboard required installing an additional package, `optuna-dashboard`, also using `pip`, maintaining the tool's modular approach.
2	Create an HPO study for IDEKO's Keras models	Optuna recognized all four of IDEKO's Keras models (NN, LSTM, CNN, RNN) out of the box without requiring framework-specific configuration. It accepted Keras objects directly for optimization.
3	Configure a search space for learning rate, batch size, and hidden units	The search space definition for the learning rate (0.0001-0.1), batch size (16-128), and hidden units (32-512) was embedded directly in the objective function. This follows Optuna's approach, where optimization configuration is defined in code rather than separate configuration files.
4	Execute an optimization study with the TPE sampler	We ran 50 trials with the default Tree-structured Parzen Estimator (TPE) algorithm, a Bayesian optimization method, to explore the hyperparameter space. The best configuration appeared around trial 30, and subsequent trials confirmed it.
5	Integrate Optuna with MLflow tracking	Logging each Optuna trial to MLflow initially failed with parameter conflict errors, since MLflow rejected duplicate parameter names when multiple trials were logged within the same run. Resolving this issue required the Optuna package, `optuna-integration`, which automatically spawns child runs for each trial with only a few additional lines of configuration.
6	Test pruning functionality	Optuna's pruner implementation automatically stopped trials performing below the median of previous trials at the same step, with 12 out of 50 trials pruned early, reducing total computation time.
7	Examine dashboard visualization and study diagnostics	The dashboard provided comprehensive visualizations, including optimization history plots, parallel coordinate diagrams, and parameter importance rankings. These offered immediate insights into the optimization process.

Optuna's flexibility extends beyond basic hyperparameter search. Its extensive catalogue of samplers (TPE, CMA-ES, Random, Grid) and pruners (Median, Hyperband, Successive Halving) allows teams to optimize their optimization strategy itself [16]. This flexibility comes with a trade-off: while in-memory and SQLite storage work seamlessly, configuring PostgreSQL or MySQL backends for production deployments requires explicit database setup and connection management that lacks comprehensive documentation.

For the IDEKO use case, Optuna's pruning capabilities allow early termination of unpromising trials. The dashboard's real-time visualizations helped identify parameter relationships that were not obvious from individual trial results. Although our evaluation focused on single-machine execution and did not test GPU-based

parallelism or multi-node distribution, Optuna does support these capabilities through its distributed optimization features.

Main Findings: Optuna's Python-native approach elegantly resolves integration conflicts with MLflow through automatic child run spawning, representing a rare example of tools proactively addressing ecosystem integration challenges rather than leaving users to debug incompatibilities.

Katib

Table 13 documents the evaluation of Katib for Kubernetes-native HPO.

Katib demonstrates how Kubernetes-native design can uniquely impact HPO in both positive and negative aspects. On the positive side, running trials in isolated pods ensures clean environments and prevents interference. Additionally, Katib's automatic cleanup prevents resource leaks that plague long-running optimization jobs. On the other hand, accessing logs from failed trials required `kubectl` commands instead of the straightforward experience the UI aims to provide. This logging limitation appeared throughout our Kubeflow evaluation, suggesting a systemic issue rather than a Katib-specific problem.

Katib's tight integration with Kubeflow ensures that experiments appear as first-class citizens in the platform, with dedicated UI sections, automatic resource management, and seamless pipeline integration. Particularly, the parallel coordinates visualization stands out, offering insights into parameter interactions that tabular results cannot convey. This visualization revealed clear patterns in our IDEKO experiments, showing that moderate batch sizes outperformed extremes regardless of the learning rate. However, Katib's containerization requirement fundamentally changes the development workflow. Unlike Optuna, where you simply wrap existing training code, Katib demands the containerization of every training script and that parameters be printed in a specific format. Although the latter seems like a minor change, it cost us considerable debugging time when our initial experiments failed with cryptic error messages. This exemplifies a recurring pattern in Kubeflow, where powerful capabilities are placed behind interfaces that assume deep Kubernetes knowledge.

The eight-section configuration process, while comprehensive, reveals another design philosophy difference. Where Optuna lets developers define parameters programmatically as needed, Katib enforces upfront specification of every aspect of the optimization. Its structure helps prevent configuration errors but makes iterative experimentation more cumbersome. Each parameter change could be edited through the UI rather than editing code. However, Katib's UI defines "success" based on experiment completion rather than goal achievement, which presents a critical usability issue. This is demonstrated by our experiment that ran to completion and displayed a green "successful" status check mark, yet achieved only 50% accuracy against a 95% target. This could mislead teams into thinking that optimization completed successfully. The disconnect between the technical success of the component (job completed) and the actual experimental goal (meeting the accuracy threshold) highlights the need for a clearer visual distinction.

Table 13: Katib's systematic test plan and results for containerized hyperparameter search

#	Test Step	Result
1	Access Katib UI through the Kubeflow dashboard	Katib came pre-integrated in both the DeployKF and Charmed Kubeflow distributions, appearing as a dedicated tab in the Kubeflow dashboard without requiring any additional configuration or service deployment.
2	Define an experiment through the UI	Creating a Katib experiment required navigating through eight configuration sections in the UI, with form validation and selection menus for each field to guide configuration.
3	Execute the hyperparameter search	Initial experiments failed with "Couldn't find any successful Trial" error. Katib's containerized training requires parameters to be printed in a specific format, so the existing IDEKO training scripts had to be modified accordingly. Once the output format was fixed, the training script, which worked fine locally, however still required careful adjustments to produce the required output format Katib expected within its containerized environment.
4	Monitor trial progress and parallel execution	Bayesian optimization with 20 maximum trials and 4 parallel workers executed successfully. The trials tab displayed real-time updates showing hyperparameter combinations ranging from learning rates of 0.0001 to 0.1, batch sizes from 16 to 128, and network architectures from 32 to 256 hidden units.
5	Validate resource allocation and pod management	Each trial ran in its own isolated Kubernetes pod. After each trial was completed, Katib automatically deleted the pod and freed its allocated CPU and memory resources, preventing resource accumulation from failed or completed experiments. However, accessing logs from individual trial pods before cleanup required considerable effort, needing `kubectl` commands due to Kubeflow's limited log viewing capabilities.
6	Test early stopping behaviour	The early stopping mechanism functioned as configured, terminating trials that showed no improvement after specified iterations.
7	Examine parallel coordinates visualization	The parallel coordinates plot elegantly revealed that lower learning rates combined with moderate batch sizes (around 64) consistently achieved better accuracy. However, the UI marked the experiment as "successful" despite not reaching the target metric, potentially misleading users.

For teams already invested in Kubeflow, Katib provides sophisticated HPO without additional tool installation. The parallel trial execution, automatic resource management, and integrated visualizations justify the containerization overhead for large-scale optimization tasks. For smaller teams or single-machine deployments, the requirement to containerize every training script and navigate complex UI configurations makes lightweight alternatives like Optuna more practical. However, its ready-made integration with Kubeflow does not introduce additional friction. Therefore, for

teams already adopting Kubeflow, Katib is the obvious choice, but it alone is not a reason to adopt Kubeflow.

> **Main Findings:** Katib's pod-based trial isolation prevents resource leaks through automatic cleanup, but its requirement for specific output formats breaks framework integrations. Training scripts require extensive modifications to produce the expected parameter format for Katib to function correctly.

Optuna vs Katib

Optuna provides a simpler integration path than Katib. Installing Optuna requires one `pip` command and the library works immediately with existing Python code. Contrastingly, Katib requires containerizing the training script, defining Kubernetes specifications, and understanding the eight-section experiment configuration. This complexity reflects their different architectures since Optuna runs as a simple Python library while Katib operates as a Kubernetes service.

Both tools support similar optimization algorithms, including Bayesian optimization, random search, and evolutionary strategies. They also provide pruning to stop unpromising trials early. Moreover, both tools offer visualisation. Katib includes the parallel coordinates plot directly in the Kubeflow dashboard. In comparison, Optuna requires installing the separate Optuna-dashboard package but offers similar visualizations to Katib once configured.

Integrating Optuna with MLflow caused parameter conflicts that required using an additional package, provided by Optuna itself, to resolve them. On the other hand, Katib integrates seamlessly with MLMD and Kubeflow Pipelines since they were designed together. This demonstrates the benefit of using a tool within its intended ecosystem.

The tools' parallelization approaches differ fundamentally. Katib spawns isolated Kubernetes pods for each trial, providing complete resource isolation. Optuna offers multi-thread, multi-process, and multi-node modes, but trials share the same environment unless explicitly configured otherwise. For teams already on Kubernetes, Katib's pod-based isolation provides better resource management. For single-machine deployments, Optuna's threading model uses resources more efficiently. These differences in resource management further showcase Kubeflow's multi-machine pod cluster priority.

Teams should choose an HPO tool based on their existing infrastructure. If already using Kubeflow or a Kubernetes-based architecture, Katib's UI-based configuration and automatic parallelization justify its complexity. For standalone projects or teams without Kubernetes expertise, Optuna provides a better developer experience with lower operational overhead. Neither tool is clearly superior for HPO, as they serve different deployment scenarios and involve distinct trade-offs.

4.1.8 Model Serving

Model serving tools deploy trained models as production services that process incoming data and return predictions. In the IDEKO pipeline, these tools expose the anomaly detection model through REST APIs, allowing CNC machines to submit sensor data and receive classifications indicating whether they are operating normally or showing signs of electrical or mechanical failures that require maintenance.

MLflow Serving

Table 14 shows the evaluation of MLflow Serving for deploying IDEKO anomaly detection models.

Table 14: MLflow Serving's systematic test plan and results for IDEKO model deployment

#	Test Step	Result
1	Install MLflow Serving dependencies	MLflow Serving comes bundled with MLflow and requires no separate installation. However, it depends on the `virtualenv` package to create isolated Python environments for each served model.
2	Prepare a model for deployment	Models already registered through MLflow Models were immediately available for serving, with all dependencies and signatures captured during the initial model logging process, as described in the MLflow Models evaluation.
3	Deploy a trained model using MLflow Serving	Deployment succeeded with a single command that automatically created a `virtualenv`, installed all captured dependencies, and started a REST server exposing prediction endpoints on the specified port.
4	Test the prediction endpoint with time series data	All model architectures that accept three-dimensional tensor inputs, where the dimensions represent batch size, time steps, and sensor features, worked seamlessly without custom preprocessing. MLflow's auto logging feature automatically captured the correct input format specification during training.
5	Integrate MLflow Serving into the Airflow pipeline	The Airflow DAG successfully automated the complete deployment workflow, including selecting the best-performing model from HPO experiments, promoting it to production status in the model registry, starting the serving process with health monitoring, and ensuring graceful shutdown. This automation process required custom orchestration logic to coordinate these steps.
6	Test batch predictions	Batch predictions functioned correctly when sending multiple samples in a single request using the standard MLflow format with "instances" as the JSON key, processing all sequences simultaneously and returning probability distributions for each sample.

MLflow Serving exemplifies how thoughtful design can dramatically simplify model deployment. Unlike KServe's complex containerization requirements and strict output formatting, MLflow Serving worked almost immediately with minimal configuration. Its single-command deployment automatically handled all complexity, from creating an isolated Python environment to installing

the exact dependencies captured during training and exposing a well-documented REST API. This seamless experience stems from MLflow's integrated ecosystem where Models, Tracking, and Serving components share metadata and eliminate the need for manual configuration.

The automatic dependency management proved impressive during evaluation. When models were saved with MLflow Models and auto log was enabled, the framework captured the complete environment specification including the Python version and any library dependencies. By default, MLflow recreated this environment in a `virtualenv` during serving to ensure consistency between training and serving environments.

For the IDEKO time series use case, MLflow's handling of complex input parameter shapes was remarkably straightforward. The LSTM models expecting 3D tensors, where the dimensions represent batch size, time steps, and sensor features, worked without any custom preprocessing or wrapper code. MLflow's auto logging functionality captured the tensor signature during training, whilst the serving layer correctly parsed nested JSON arrays into the required format. This automatic handling of sequence data contrasted sharply with KServe's requirement for custom preprocessing logic.

Model deployment revealed limitations in production operations. MLflow Serving runs as a simple foreground process using FastAPI [40], providing no built-in process management, health monitoring, or automatic restarts. The single-threaded FastAPI server also lacks horizontal scaling capabilities, making it suitable for development and low-traffic scenarios but is insufficient for high-throughput production deployments. MLflow's documentation acknowledges these scalability limitations and provides a clear upgrade path. For production deployments requiring autoscaling, load balancing, or GPU acceleration, MLflow Serving can deploy to various targets, including SageMaker, AzureML, Databricks, and Kubernetes frameworks, like Seldon Core or KServe. Therefore, the models remain unchanged, with only the deployment target differing. This flexibility allows teams to start with simple local serving during development and seamlessly transition to production-grade infrastructure without modifying the model artifacts. MLflow's documentation highlights the benefits of integrating with Seldon Core's MLServer for Kubernetes deployments, as it provides asynchronous request handling through worker pools. This enables the server to accept new requests while processing intensive inference workloads. When deployed through Seldon Core or KServe, MLflow Serving gain access to advanced features, such as auto scaling, A/B testing, and multi-model serving, while maintaining the simple model packaging benefits [26].

Main Findings: MLflow Serving abstracts deployment complexity through a consistent interface that scales from local development to production infrastructure without changing model artifacts. Teams can start with simple serving and transition to enterprise platforms as their requirements grow.

KServe

Table 15 presents the challenging evaluation of KServe for Kubernetes-native model serving.

Table 15: KServe's systematic test plan and results for IDEKO model serving

#	Test Step	Result
1	KServe installation	KServe came pre-installed with Charmed Kubeflow, requiring no additional setup or configuration. However, it was notably absent from the DeployKF distribution, limiting deployment options. The serving framework was immediately available through the Kubernetes cluster in Charmed Kubeflow.
2	Deploy IDEKO's Keras model using KServe	Initial deployment succeeded using a configuration file that defined how KServe should serve the model by specifying the model location, runtime, and resource requirements. The TensorFlow serving runtime automatically handled the SavedModel format, successfully exposing the model on a REST endpoint accessible through the cluster.
3	Test inference endpoint with sample sensor data	Testing with 20 sensor readings from IDEKO's f_3 signal produced incorrect classifications. An investigation revealed the model required approximately 18,000 timesteps for accurate anomaly detection, far exceeding JSON payload size limits. This forced us to create a custom predictor class inheriting from KServe's base implementation, in which we defined three required methods: `preprocess()` to parse CSV files instead of JSON, `load_model()` to initialize the model, and `predict()` to perform inference. After deploying this custom predictor, the model successfully processed the full time series data and produced accurate classifications matching our local tests.
4	Integrate KServe deployment within Kubeflow Pipelines	KServe components do not support standard artifact input/outputs like other Kubeflow tools in the pipeline, requiring us to access Kubeflow's MinIO storage directly using specific path formats. We spent three days debugging *"model not found"* errors and credential mounting issues. The error logs provided misleading information that hindered troubleshooting. The artifact paths displayed in the Kubeflow UI did not match the storage locations KServe expected. This fundamental mismatch made pipeline integration impossible without a deep understanding of both systems' internal architecture, forcing us to abandon the integration.

KServe demonstrates both the promises and limitations of Kubernetes-native model serving. The framework's inclusion in Charmed Kubeflow eliminated installation complexity, and its automatic handling

of TensorFlow SavedModel format made initial deployment remarkably simple.

However, this simplicity masked critical assumptions about model behaviour. The discovery that predictions required a much larger timestep context exposed a fundamental mismatch between KServe's REST API design and time series models. JSON payload size limitations forced the implementation of CSV file upload support by inheriting from KServe's base implementation with custom preprocessing logic to parse and preprocess the data.

The Kubeflow Pipeline integration failure revealed a fundamental architectural incompatibility. Unlike other Kubeflow components that pass artifacts through the pipeline's input/output system, KServe requires direct access to MinIO storage, which stores the model. This difference in artifact handling caused multiple implementation headaches. The outputted model by the training component could not be found despite being successfully stored, and the path even appeared in the Kubeflow dashboard. However, that same path did not work in the KServe component. Even after following multiple forum threads of similar use cases, where it was well known that these paths differed, the path could not be accessed. Furthermore, there were credential issues when accessing the local MinIO pod, similar to what we had encountered in the Kubeflow Pipeline (KFP) implementation with DVC, where error logs provided misleading or incomplete information. The disconnect between KServe's storage expectations and Kubeflow's artifact abstraction made integration practically impossible without extensive knowledge of the systems' internals, Kubernetes and Charmed Kubeflow's credential setup.

Standalone KServe deployment works reliably for models with standard input sizes, but trying to integrate it within Kubeflow Pipelines means wrestling with undocumented storage patterns and credential management nightmares. The framework's advanced features, like autoscaling and canary deployments, are impressive. However, for smaller teams deploying a single anomaly detection model, the overhead is hard to justify when a simple Flask API would do the job. Components that promise enterprise-grade capabilities end up demanding enterprise-grade expertise and resources that smaller deployments might not be able to justify.

> **Main Findings:** KServe delivers enterprise features like autoscaling and canary deployments, but artifact path mismatches with Kubeflow Pipelines created insurmountable integration challenges. The framework assumes substantial Kubernetes knowledge.

MLflow Serving vs KServe

The contrast between MLflow Serving and KServe reveals fundamentally different approaches to model deployment. MLflow Serving prioritizes developer experience with its single-command deployment that automatically creates an isolated `virtualenv`, installs dependencies, and starts a REST server. This simplicity enables immediate testing and evaluation without wrestling with infrastructure concerns. In contrast, KServe arrives pre-integrated with Kubernetes, providing production-grade deployment capabilities from the start, but this power comes with significant complexity that manifests in multiple ways.

Deployment requirements differ dramatically between the tools. MLflow Serving leverages the metadata already captured during model training, requiring only a model URI to serve predictions. KServe demands implementation of a custom Python class with multiple methods for model loading, preprocessing, prediction, and postprocessing, even for standard model formats. This additional code layer adds development overhead but provides fine-grained control over the serving pipeline.

The integration challenges with Kubeflow Pipelines exposed a critical architectural mismatch in KServe. While other Kubeflow components seamlessly pass artifacts through the pipeline's input/output system, KServe requires direct MinIO storage access with proper credentials. The artifact URLs displayed in the Kubeflow UI do not match the paths KServe expects, creating a frustrating debugging experience with misleading error messages. This disconnect made pipeline integration practically impossible without deep knowledge of Kubeflow's storage architecture. Additionally, the promised integration between KServe and Kubeflow Model Registry for versioning management could not be tested, as neither DeployKF nor Charmed Kubeflow distributions include the Model Registry component.

Despite these integration challenges, KServe delivers enterprise-grade capabilities that MLflow Serving lacks. Features like autoscaling, canary deployments, traffic splitting, and multi-model serving come built-in with KServe's Kubernetes-native architecture. For organizations already invested in Kubernetes infrastructure which require these advanced features, KServe's complexity may be justified. However, for teams seeking straightforward model deployment with minimal operational overhead, MLflow Serving's simplicity and seamless integration with the broader MLflow ecosystem make it the more practical choice.

4.2 RQ2: General Stack Evaluation

RQ2 evaluates both stacks end to end in the IDEKO pipeline: installation, integration, observability, hyperparameter search, and serving. This research question explicitly does not seek to compare the MLflow and Kubeflow stacks against each other, but rather examines how individual tools from each ecosystem behave when integrated in realistic production pipelines, documenting their operational characteristics, integration complexities, and the overhead and failure modes that emerge in practical deployments.

4.2.1 MLflow Stack

Figure 2 illustrates the architecture of the MLflow-centric stack, which implements a modular pipeline orchestrated through Apache Airflow's Python-based Directed Acyclic Graphs (DAGs). The arrows in the architecture figure indicate data flow with labels showing data type. Components marked *«airflowTask»* represent DAG tasks executed by the Airflow scheduler. Components marked *«service»* are external services and storage systems. MLflow Tracking Server provides centralized experiment tracking and model registry. The pipeline initiates with DVC executing as an Airflow command task, which retrieves versioned sensor data from a local MinIO

Table 16: Evaluation matrix – MLflow Stack

	Criterion	DVC	FEAST	Apache Airflow	Optuna	MLflow Tracking	MLflow Models	MLflow Datasets
Usability	Setup & Installation Simplicity	High	High	Medium	High	High	High	High
	Configuration Simplicity	Medium	Medium	Low	Medium	High	High	High
	Ease of Use	Medium	Medium	Medium	High	High	High	High
	Documentation Support	High	High	Medium	Medium	High	High	High
Functionality	Functional Appropriateness	High	High	High	High	High	High	Medium
	Functional Completeness	Medium	Medium	High	High	High	High	Medium
	Reliability	Medium	High	High	High	High	High	Medium
Flexibility	Platform Support	Medium	Medium	High	High	High	High	High
	Integration Readiness	High	High	High	High	High	High	Medium
	Ease of Integration	High	Medium	Medium	High	High	High	Medium
	Modularity	Medium	Low	Medium	High	High	Low	Low
Vitality	Community Support & Adoption	High	Medium	High	High	High	High	High
	Maturity	High	Medium	High	Medium	High	High	High
	Active Development & Maintenance	High	High	High	High	High	High	High

Table 17: Evaluation matrix – Kubeflow Stack

	Criterion	LakeFS	FEAST	Kubeflow Pipelines	Katib	ML Metadata	KServe	Kubeflow Model Registry
Usability	Setup & Installation Simplicity	High	High	Medium	Medium	Medium	Medium	Medium
	Configuration Simplicity	Medium	Medium	Low	Medium	Medium	Medium	Medium
	Ease of Use	Medium	Medium	Medium	Medium	Medium	Medium	Medium
	Documentation Support	High	High	Medium	Medium	Medium	Medium	Medium
Functionality	Functional Appropriateness	High	High	High	High	Medium	High	Medium
	Functional Completeness	Medium	Medium	High	High	Medium	High	Medium
	Reliability	Medium	High	High	High	Medium	High	Medium
Flexibility	Platform Support	High	Medium	High	High	High	High	High
	Integration Readiness	High	High	High	High	Medium	High	Medium
	Ease of Integration	High	Medium	Medium	High	Medium	High	Medium
	Modularity	Medium	Low	High	High	Medium	Medium	Medium
Vitality	Community Support & Adoption	Medium	Medium	High	High	Medium	High	Medium
	Maturity	Medium	Medium	Medium	Medium	Medium	Medium	Low
	Active Development & Maintenance	High	High	High	High	Medium	High	Medium

object storage server hosting the IDEKO dataset and pulls CSV files containing high-frequency sensor readings.

Following data retrieval, the Process Features component executes a Python operator within the Airflow DAG, performing the required preprocessing on the raw sensor data. It then computes features such as rolling averages and statistical aggregates, and saves the processed data to local Parquet files. The columnar Parquet format was selected as it is a requirement for Feast. However, it also provides superior compression and query performance compared to CSV. The Feast feature store subsequently reads from these Parquet files, registering feature definitions and maintaining schema versioning to prevent training-serving skew. This unidirectional data flow from processing to storage to feature registration ensures data consistency across the pipeline.

The training phase demonstrates tight integration between multiple MLflow components and the Optuna HPO framework. The

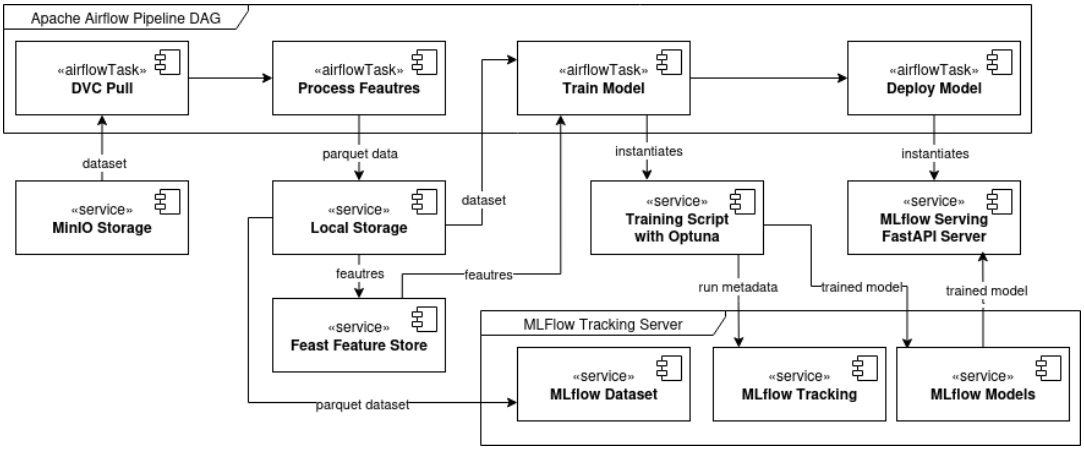

Figure 2: MLflow-centric Stack Architecture

IDEKO training script is executed as another Airflow task and leverages Optuna to systematically explore hyperparameter spaces, including learning rates, batch sizes, and network architectures. Through MLflow's auto logging capabilities for Keras, the training process automatically captures comprehensive metadata without requiring explicit logging code. Optuna saves hyperparameter configurations and optimization results directly to the MLflow Tracking Server, while the training script logs metrics, parameters, and model artifacts. This convergence of tracking data into a single server simplifies experiment management and comparison.

The MLflow Tracking Server serves as the central metadata repository, hosting the three MLflow libraries. MLflow Datasets maintains the lineage between training runs and their input data versions. MLflow Tracking stores all experimental parameters, metrics, and system information, providing comprehensive audit trails for each training run. MLflow Models manages the packaged model artifacts along with their dependencies and environment specifications. This unified architecture eliminates the need for separate installations or complex integration code, as all components operate within the same tracking server infrastructure.

The final Airflow task deploys the model through MLflow Serving. A single command retrieves the specified model version from MLflow Models and instantiates a FastAPI server for inference. The serving command constructs an isolated throwaway Python virtual environment and installs any required dependencies. The deployed model exposes standard endpoints for prediction requests.

4.2.2 Kubeflow Stack

Figure 3 presents the Kubeflow-centric stack architecture. Arrows indicate data flow with labels showing what data is being transferred. Components marked as *«kfpComponent»* represent Kubeflow pipeline components which are the steps executed sequentially in the pipeline. Components marked *«service»* are external services integrated with the pipeline. The Kubeflow Pipeline orchestrates the workflow as a series of containerized components, each executing in isolated Kubernetes pods with well-defined input and output artifacts. The pipeline begins with the LakeFS Pull component, which retrieves the versioned IDEKO dataset from the LakeFS service. LakeFS provides Git-like semantics for object storage, managing dataset versions in its integrated MinIO backend. The retrieved dataset is saved as a Kubeflow artifact and passed through Kubeflow's input/output system to the subsequent components.

Parallel to data retrieval, the Git Clone component fetches training code from the git repository, similarly packaging it as an artifact for consumption by the training component. Each component operates without shared filesystem access, receiving only the artifacts explicitly declared in its interface specification. This isolation prevents hidden dependencies and ensures that pipeline execution remains deterministic across different environments and execution times.

The Train Model component receives both the dataset and code artifacts through Kubeflow's artifact passing mechanism, executing the IDEKO training script. The component uses the hyperparameters that were identified beforehand by Katib. Each Katib experiment runs in complete isolation, preventing interference between trials while enabling efficient resource utilization across the cluster. The optimal hyperparameters discovered by Katib are used in the model training phase, with all metadata captured in the MLMD store. MLMD serves as the central metadata repository, tracking all pipeline executions, artifact lineage, and experimental results in a MySQL backend. Kubeflow Artifacts manages the intermediate outputs between pipeline components, storing them in a MinIO object storage.

The deployment phase utilizes KServe for model serving, consuming the trained model artifact directly from the Kubeflow Artifacts store. KServe provides Kubernetes-native serving capabilities including automatic scaling, canary deployments, and traffic splitting, though our evaluation revealed integration challenges with the artifact passing system. Notably, our implementation could not utilize the Kubeflow Model Registry, as neither the DeployKF nor the Charmed Kubeflow distribution included this component, forcing direct use of the Kubeflow Artifact store without the additional features that a model registry would provide, such as simplified versioning.

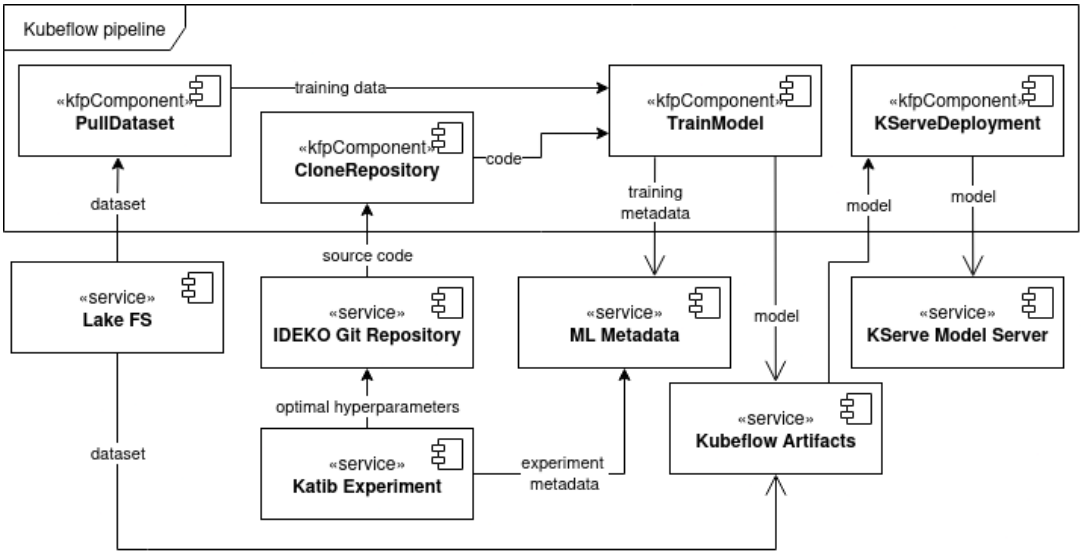

Figure 3: Kubeflow-centric stack Architecture

5 Discussion

This section analyses the broader trends of our findings, examining tool capabilities, practical usability and documentation quality.

5.1 Answering RQ1

Our evaluation of individual MLOps tools showed a clear trade-off between powerful capabilities and operational complexity. Across all component categories, the tools in the MLflow ecosystem consistently achieved higher usability scores because of their straightforward installation, intuitive interfaces, and comprehensive documentation. Conversely, tools in the Kubeflow ecosystem offered superior functionality specifically for distributed, large-scale workloads through features like pod isolation and fine-grained resource control. However, these enterprise-scale capabilities came at substantial usability costs as the Kubeflow tools required significantly more configuration, deeper technical knowledge, and longer setup times than their MLflow counterparts.

5.2 Answering RQ2

The complete comparison of the MLflow and Kubeflow-centric stacks revealed that they are not competing alternatives but fundamentally different approaches to building MLOps pipelines. Our MLflow stack combined best-of-breed tools (MLflow for tracking, DVC for versioning, Airflow for orchestration, Optuna for HPO) that excel individually and integrate through standard interfaces. In contrast, the Kubeflow stack provided an integrated platform where components like Kubeflow Pipelines, Katib, and MLMD were designed to work together from the outset. The MLflow stack's modular approach allowed teams to start with minimal infrastructure and swap components as needed, while the Kubeflow stack's integrated nature demanded upfront investment in the Kubernetes ecosystem but provided unified management and consistent architectural patterns. Our evaluation found that these stacks serve different organizational contexts: the MLflow stack suits teams that value flexibility and gradual adoption, whereas the Kubeflow stack targets organizations already committed to Kubernetes infrastructure seeking a unified platform. Our findings show that the choice of stack depends on whether teams prioritize accessibility and modularity, as in the MLflow stack, or integration and enterprise capabilities, as in the Kubeflow stack.

5.3 Complexity Cost of Enterprise Features

Our evaluation reveals a clear trade-off between enterprise features and usability. While Kubeflow provides multi-tenancy, distributed execution, and access control, these features require substantial operational overhead that teams must weigh against their actual needs. This platform requires approximately 25 services to be deployed and managed. On the positive side, many of these services, such as orchestration (Kubeflow Pipelines) and HPO (Katib), are built-in, whereas the MLflow stack requires them as separate installations (Apache Airflow and Optuna). When counting all components in both stacks, the complexity gap shrinks, though Kubeflow's Kubernetes base still requires more operational expertise for straightforward tasks. This was evident in our RQ1 evaluation (Section3.1) as evidenced by the need to understand Kubernetes concepts, manage cross-namespace permissions, and debug container networking issues. For instance, using git-cloned code in downstream pipeline components required explicit artifact passing between isolated containers. In contrast, the MLflow stack pipeline needed only standard Python imports. MLflow's design decisions prove sensible for organizations operating at scale, where isolation and reproducibility justify the added complexity. However, for teams focused on rapid experimentation and model development, this overhead represents time diverted from actual ML work.

The debugging complexity increased substantially when transitioning from local Python processes in the MLFlow stack to containerized Kubernetes pods in the Kubeflow stack. Simple print debugging transformed into `kubectl` log investigations, and straightforward stack traces became distributed across multiple pod logs, requiring timestamp correlation. Kubeflow's MySQL pod's unexplained 16GB memory consumption, despite configuration attempts to limit it, exemplifies how infrastructure problems can derail ML work. While teams with Kubernetes expertise might resolve such issues more efficiently, the fundamental challenge remains that data scientists must troubleshoot infrastructure rather than focus on model development. Organizations such as Spotify have demonstrated Kubeflow's value at scale [35], as referenced in our SLR [2]. However, the complexity investment required to master Kubernetes, manage distributed systems and debug containerized workflows, only yields returns when scale genuinely demands such capabilities.

On the other hand, the MLflow architecture provides an evolutionary path that aligns with team growth patterns. Teams can begin with a simple FastAPI deployment using a single command, then package models for Kubernetes deployment as requirements grow, and eventually adopt advanced serving platforms such as Seldon Core or KServe itself, all while maintaining the same MLflow model format. This approach enables teams to defer complexity until scale justifies it, rather than adopting enterprise infrastructure prematurely. Teams already operating at scale with established Kubernetes expertise may reasonably choose Kubeflow from the outset. Conversely, teams uncertain about their eventual scale requirements benefit from MLflow's gradual complexity curve. The ability to start with minimal infrastructure and then evolve as needed prevents premature optimization while preserving future architectural flexibility.

Both platforms offer paths to managed services, though with different trajectories. Kubeflow distributions exist for major cloud providers (Azure, Google Kubernetes Engine), as documented in our evaluation (Section 4.1.3.1). Similarly, MLflow users can transition to Databricks' managed platform or be deployed to services like SageMaker. The key distinction lies in the migration path: MLflow allows teams to begin with minimal infrastructure and gradually adopt managed services, while Kubeflow typically requires upfront Kubernetes configuration and investment.

5.4 Documentation Quality

The documentation strategies employed by different tools revealed their underlying priorities and target audiences. MLflow's documentation begins with a five-minute quickstart guide that produces working code immediately, followed by progressive tutorials of advanced features and other modules. This approach, combined with an AI-powered documentation chatbot, demonstrates a clear focus on the developer's experience. Our evaluation found that MLflow offered the quickest path from installation to comprehensive experiment tracking, automatic metric graphing, and complete data lineage visualization.

In contrast, Kubeflow's documentation was fragmented across multiple sources. While the main project documentation is adequate and covers core concepts well, the distribution-specific details reside on separate sites, and individual components maintain their own repositories. Our evaluation revealed that some commands, namespace configurations, and service names differed between the official documentation and Charmed Kubeflow's actual deployment. Even MLMD, a core Kubeflow component, presented dead links in its documentation when attempting to access its quickstart guide. This fragmentation reflects the downsides of Kubeflow's distributor-oriented approach, where different organizations maintain their own distributions.

Notably, corporate backing appears to correlate with higher documentation quality. MLflow, with Databricks' resources, provides the most polished experience from documentation through implementation. On the other hand, Kubeflow reflects Google's influence in core components but becomes fragmented in areas maintained by the community. Although Apache Airflow is mostly community-driven, it still provides comprehensive documentation for all configuration options and operators. However, it lacks well-organized tutorials that guide users through common workflows. Tools featuring clear, linear documentation such as MLflow consistently scored higher on our usability metrics.

5.5 Version Compatibility and Ecosystem cohesion

Version compatibility is a problem, especially with fragmented stacks like our MLflow stack. During our evaluation of Airflow, we ran into problems because its pinned Feast version was outdated without proper Python SDK support, necessitating manual overrides of the constraints file. This issue highlights a key difference in the two stacks' architecture: the MLflow stack needs to carefully manage dependencies between separate tools, while Kubeflow's integrated suite of first-party tools ensures that its parts work together without having to worry about third-party integration. Kubeflow remains modular, allowing the integration of external tools when needed. DeployKF demonstrates this flexibility by pre-integrating MLflow for experiment tracking instead of relying only on MLMD. As a result, MLflow offers a better user experience. However, Kubeflow has a unique advantage when teams use its built-in parts: tools like Katib, MLMD, and KServe all follow the same Kubernetes-native philosophy and architectural assumptions. This architectural consistency is a key benefit for teams that are ready to fully adopt the Kubeflow ecosystem. The platform's modularity also means that teams can still use their favourite external tools when the built-in option does not meet their needs.

6 Threats to Validity

In this section, we follow the threat classification schemes for experiment validity described by Ampatzoglou *et al.* [41] and outline the threats that may affect the validity of our research.

6.1 External Validity

External validity concerns the generalizability of our findings beyond the specific context of our evaluation. Our results come from a single use case: the IDEKO anomaly detection pipeline. While based on a real industrial scenario, this pipeline addresses only a time-series classification workload. Other ML domains, such as computer vision or NLP, could expose different strengths and weaknesses in the tools we tested.

Additionally, we assessed only batch processing scenarios through the IDEKO pipeline, excluding streaming and real-time inference workloads. Our setup also ran everything on a single machine, so it did not fully test distributed computing capabilities, particularly for Kubeflow's Kubernetes-native design intended for multi-node clusters. Kubeflow is built to handle large-scale workloads across multiple nodes, but we did not evaluate it under those conditions. While this means we cannot comment on how the tools behave in a distributed setting, Kubernetes is a mature and widely used system with a long track record of reliably running large workloads [42]. This reduces the likelihood that problems would stem from the orchestration layer itself, although tool-specific behaviour under multi-node deployments remains untested.

6.2 Internal Validity

Internal validity examines whether the study's design and execution provide a reliable basis for linking causes to effects. In our evaluation, the choice of Kubeflow distribution had a minor impact on our findings. As documented in Section 4.1.3.1, distribution-specific issues, such as the MySQL memory consumption problem, are not inherent to Kubeflow itself but stem from Canonical's packaging decisions. Other distributions, like DeployKF or cloud-specific versions, may yield different scores, limiting how much we can generalize our findings beyond the specific distribution tested. We explicitly document which issues were distribution-specific versus platform-inherent to help readers understand which findings apply broadly to Kubeflow versus only to Charmed Kubeflow.

We also did not evaluate security features, multi-user authentication, or role-based access control (RBAC) capabilities, which are critical for multi-member organizations. While both stacks offer these enterprise features, their complexity and configuration requirements could substantially impact the usability and flexibility scores we assigned. We explicitly scope our evaluation to development workflows and clearly indicate that production security assessments require separate evaluation.

Furthermore, some tools were set up with vendor-provided quick start examples instead of the recommended full-scale production deployment, which may have minimized their perceived setup complexity. For instance, Airflow's "standalone" command succeeds in minutes, but production deployment requires orchestrating separate database, webserver, scheduler, worker processes and monitoring infrastructure, transforming a simple setup into a much more extensive engineering effort. This gap between development convenience and production requirements represents a threat to the validity of our scores when applied to production environments. However, the requirement differences between quick start examples and full production setups are documented for every tool.

The rapid evolution of the MLOps ecosystem during our study period presents another threat to internal validity. MLflow underwent a major version update from 2.11 to 3.2, introducing breaking changes including module removals and API modifications. Similarly, DeployKF announced upcoming support for the Model Registry and KServe during our evaluation, demonstrating ongoing divergence in Kubeflow distributions' component selections. These continuous changes mean our findings reflect specific tool versions that may already exhibit different behaviour in newer releases. To mitigate this, we report all tool versions and evaluation dates in the Appendix.

A learning curve effect also influenced our evaluation process. Debugging patterns and Kubernetes knowledge accumulated from early frustrations proved valuable when assessing subsequent tools, potentially creating an order effect where later-evaluated tools benefited from our increased expertise. This temporal bias could systematically advantage tools evaluated later in our study, particularly for usability assessments.

6.3 Construct Validity

Construct validity concerns how well our measures align with theoretical concepts. Our evaluation rubric (Table 2) applies qualitative Low/Moderate/High ratings based on subjective assessment from a single researcher. Without multiple evaluators to cross-validate ratings, individual biases and preferences may systematically influence scores. This is compounded by our limited Kubernetes expertise since an experienced Kubernetes user might rate the same tasks as easier, whereas a data scientist without DevOps background might find them even more challenging. While the pilot study helped calibrate initial judgments, it could not address these fundamental limitations in evaluator expertise and single-researcher bias.

6.4 Conclusion Validity

Conclusion validity focuses on the accuracy of the deductions generated from our data analysis. The following two tools could not be fully evaluated due to integration failures: KServe within Kubeflow Pipelines and the Kubeflow Model Registry. This prevented complete feature comparison and may unfairly disadvantage the Kubeflow stack, as these integration issues might be resolvable with additional expertise or alternative configuration approaches, which were not explored within our time constraints. We explicitly identify which components could not be evaluated rather than generalizing limitations to the entire stack.

To support the verifiability and reproducibility of our findings, we provide a comprehensive replication [43] package including all evaluation scripts, configuration files, and detailed field notes. This package contains the exact tool versions used, step-by-step installation instructions, and the complete IDEKO pipeline implementation for both stacks. While hardware differences and evolving tool versions may produce variations in specific timings or scores, the availability of our implementation artifacts enables independent verification of our core findings and methodology.

7 Related Work

This section positions our findings within the context of existing MLOps evaluation studies and highlights how our work aligns with previous research.

In 2022, Köhler carried out an in-depth evaluation of open-source Kubernetes-native MLOps platforms [44]. The study compared Kubeflow, Pachyderm, and Polyaxon in terms of performance, vitality, usability, and functionality. While not identical, these categories overlap heavily with our own four evaluation dimensions of usability, functionality, flexibility, and vitality (Section 3), which we apply at both tool and stack levels. Köhler's conclusion on the

usability of Kubeflow was very clear: even a simple model deployment workflow demanded a high level of technical expertise. This aligns closely with our own findings, as even getting basic tasks to run reliably required substantial Kubernetes expertise. Köhler also reported that Kubeflow's documentation was inconsistent and in parts outdated, which aligns with our experience during the MLMD integration, where the official documentation had dead links. We also encountered deployment-level issues that mirror those in Köhler's study, such as the KServe directory error, which stalled their model-serving tests and prevented us from integrating KServe with Kubeflow pipelines [44].

Dixit et al. [45] examined the deployment of a deep learning model using MLflow on a Kubernetes cluster. In line with the results of our SLR [2], they identified MLflow, Kubeflow, and DVC as the most widely adopted tools. They chose MLflow over Kubeflow for their study since MLflow's tracking and management capabilities are more approachable, its installation process is lighter, and the interface is cleaner with documentation that is easier to follow, which aligns with our findings. They also pointed out that MLflow integrates well with a wide range of tools and platforms, making it adaptable to varied environments, whereas Kubeflow's integrations are tightly bound to Kubernetes. In their study, Dixit et al. claimed that "MLflow is an open-source tool, whereas Kubeflow may require the use of paid resources such as Google Cloud Platform" [45]. While we agree with their characterisation of MLflow's interoperability, we diverge on their claim about Kubeflow. Our experience indicates that this statement is misleading, as both Kubeflow and MLflow are fully open-source and can be deployed entirely on local infrastructure. In both our study and that of Dixit et al., cost becomes a factor only when choosing a managed service, which applies equally to both platforms [45].

This study might give the impression that Kubeflow's Kubernetes-native architecture is necessary for production-scale deployments, while MLflow suits only smaller initiatives. However, Chen et al. [46] provide compelling evidence to the contrary with their analysis of MLflow deployments in their organization, managing millions of models through the MLflow Model Registry and producing hundreds of thousands of models per deployment. The way they accomplish this scale is especially interesting, as instead of using intricate container orchestration like Kubeflow does, they utilise MLflow's modular architecture, which enables teams to begin with a basic infrastructure and expand it as needed, as we discussed earlier (Table 14). This finding reinforces our observation that the choice between MLflow and Kubeflow is not fundamentally about scalability limits, but rather about the team's current expertise, DevOps capabilities, and whether they are already operating within the Kubernetes ecosystem [46].

8 Conclusion

This thesis addressed a critical gap in MLOps literature by evaluating the actual implementation experience of popular MLOps tools rather than just comparing their features. Organizations deploying ML models face overwhelming tool choices with little practical guidance on integration complexity and operational overhead. We implemented the IDEKO anomaly detection pipeline using the most popular MLOps tools identified in our SLR [2], organizing them into two representative stacks, one MLflow-centric and one Kubeflow-centric, to test realistic integration scenarios.

Based on this implementation, our evaluation revealed a consistent pattern in which powerful capabilities come at the cost of operational complexity. Tools with simpler architectural philosophies, like MLflow Tracking, DVC, and Optuna, achieved higher usability scores due to their straightforward installation and intuitive interfaces. Meanwhile, Kubernetes-native tools, such as Kubeflow Pipelines, Katib, and KServe, offered superior functionality specifically for distributed and large-scale workloads, including pod isolation and multi-node execution, but demanded substantial Kubernetes expertise. However, these tools also brought significant operational challenges, such as KServe's artifact path incompatibilities, MLMD's lack of visualization, and Katib's strict output requirements. Most strikingly, functionally equivalent tools from the two stacks showed vastly different learning curves, as seen with Optuna from the MLflow stack, which could be configured in minutes, compared to Katib from the Kubeflow stack, which required containerization and complex UI forms. These insights enable practitioners to anticipate operational challenges and select tools based on their team's actual capabilities rather than feature lists.

Our evaluation suggests several directions for future research. One possible avenue is to extend the approach beyond time-series classification to other domains. Another is to deploy these stacks on real multi-node Kubernetes clusters with distributed training workloads, which would clarify whether Kubeflow's operational overhead delivers proportional value at scale. This would particularly benefit organizations debating infrastructure investments. A further direction involves evaluating enterprise features, including security configurations, RBAC implementation, and multi-tenancy setup, to assess the true effort required to achieve production-grade deployments. Many teams underestimate the gap between functional prototypes and secure, multi-user systems. Together, these efforts would contribute to a deeper understanding of MLOps tool selection and adoption patterns across diverse organizational contexts.

In conclusion, this thesis contributes to MLOps literature by moving beyond feature lists to evaluate tools in realistic implementation scenarios. Our findings show that tool choice should not be guided by feature list comparisons alone but by an honest assessment of a team's expertise, scalability requirements, and operational capacity. By documenting the integration trade-offs of two representative stacks, this study offers practical insights for both researchers and practitioners navigating the rapidly evolving MLOps landscape.

References

[1] Thomas Davenport and Katie Malone. Deployment as a Critical Business Data Science Discipline. *Harvard Data Science Review*, 3(1), December 2020. Publisher: The MIT Press.

[2] Zakkarija Micallef. A systematic review of MLOps tools: Practices, challenges, and lessons learned. Technical report, Zenodo, May 2025. https://doi.org/10.5281/zenodo.15459745.

[3] Ideko. Research Center | IDEKO.

[4] Mohammad Zarour, Hamza Alzabut, and Khalid T. Al-Sarayreh. MLOps best practices, challenges and maturity models: A systematic literature review. *Information and Software Technology*, 183:107733, July 2025.

[5] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, Michael Young, Jean-François Crespo, and Dan Dennison. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural*

Information Processing Systems, volume 28. Curran Associates, Inc., 2015.
[6] Dominik Kreuzberger, Niklas Kühl, and Sebastian Hirschl. Machine Learning Operations (MLOps): Overview, Definition, and Architecture, May 2022. arXiv:2205.02302 [cs].
[7] Óscar A. Méndez, Jorge Camargo, and Hector Florez. Machine Learning Operations Applied to Development and Model Provisioning. In Hector Florez and Hernán Astudillo, editors, *Applied Informatics*, volume 2236, pages 73–88. Springer Nature Switzerland, Cham, 2025. Series Title: Communications in Computer and Information Science.
[8] Vidushi Arora. Exploring real-world challenges in MLOps implementation: a case study approach to design effective data pipelines. 2024.
[9] Dvc documentation. https://dvc.org/doc, 2025. Accessed 2025-05-01.
[10] lakefs documentation. https://docs.lakefs.io/, 2025. Accessed 2025-05-01.
[11] T Vishwambari and Sonali Agrawal. Integration of Open-Source Machine Learning Operations Tools into a Single Framework. In *2023 International Conference on Computing, Communication, and Intelligent Systems (ICCCIS)*, pages 335–340, November 2023.
[12] Feast documentation. https://docs.feast.dev/, 2025. Accessed 2025-05-01.
[13] Iago Águila Cifuentes. Design and Development of an MLOps Framework. Master's thesis, Universitat Politècnica de Catalunya, June 2023. Accepted: 2023-10-25T10:32:47Z.
[14] Kubeflow pipelines documentation. https://www.kubeflow.org/docs/components/pipelines/, 2025. Accessed 2025-05-01.
[15] Kubernetes. https://kubernetes.io/, 2025. Open-source container orchestration platform.
[16] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.
[17] Kanwarpartap Singh Gill, Vatsala Anand, Rahul Chauhan, Ruchira Rawat, and Pao-Ann Hsiung. Utilization of Kubeflow for Deploying Machine Learning Models Across Several Cloud Providers. In *2023 3rd International Conference on Smart Generation Computing, Communication and Networking (SMART GENCON)*, pages 1–7, December 2023.
[18] Katib: Kubernetes-native automated machine learning. https://www.kubeflow.org/docs/components/katib/, 2025. Accessed 2025-05-01.
[19] MLflow Developers. Hyperparameter tuning with child runs. https://mlflow.org/docs/latest/ml/traditional-ml/tutorials/hyperparameter-tuning/notebooks/hyperparameter-tuning-with-child-runs/, 2025. Accessed: 2025-08-17.
[20] Giulio Mallardi, Fabio Calefato, Luigi Quaranta, and Filippo Lanubile. An MLOps Approach for Deploying Machine Learning Models in Healthcare Systems. In *2024 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 6832–6837. IEEE, 2024.
[21] Michal Bacigál. Design and Implementation of Machine Learning Operations. February 2024. Accepted: 2024-02-09T23:53:17Z Publisher: České vysoké učení technické v Praze. Vypočetní a informační centrum.
[22] Luca Scotton. Engineering framework for scalable machine learning operations. 2021.
[23] Mlflow documentation. https://mlflow.org/docs/latest/index.html, 2025. Accessed 2025-05-01.
[24] Ml metadata (mlmd) for tfx. https://www.tensorflow.org/tfx/guide/mlmd, 2025. Accessed 2025-05-01.
[25] Kubeflow model registry (docs). https://www.kubeflow.org/docs/components/model-registry/, 2025. Accessed 2025-05-01.
[26] Kserve documentation. https://kserve.github.io/website/, 2025. Accessed 2025-05-01.
[27] Canonical Ltd. Charmed kubeflow documentation. https://charmed-kubeflow.io/docs, 2025. Accessed: 2025-08-10.
[28] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. Company: Springer Distributor: Springer Institution: Springer Label: Springer Publisher: Springer US.
[29] ISO 25010.
[30] MinIO, Inc. MinIO: High Performance Object Storage, 2025. Accessed: 2025-08-14.
[31] Treeverse. lakeFS - Git-like Operations on Object Storage, 2025. Accessed: 2025-08-14.
[32] Ubuntu 22.04 lts. https://ubuntu.com/22-04, 2022. Long Term Support release of the Ubuntu operating system.
[33] Microk8s. https://microk8s.io/, 2025. Lightweight, single-package Kubernetes distribution by Canonical.
[34] Juju. https://juju.is/, 2025. Open-source orchestration engine by Canonical.
[35] Andres Felipe Varon Maya. The State of MLOps.
[36] deployKF Authors. deploykf documentation. https://www.deploykf.org/docs/, 2024. Accessed: 2025-08-10.
[37] Jukka Ruohonen and Qusai Ramadan. Snaps: Bloated and Outdated?, July 2025. arXiv:2507.00786 [cs] version: 1.
[38] Google Research. Ml metadata (mlmd). https://github.com/google/ml-metadata, 2024. Accessed: 2025-08-18.
[39] TensorFlow Extended. Ml metadata. https://www.tensorflow.org/tfx/guide/mlmd, 2024. Accessed: 2025-08-18.
[40] Sebastián Ramírez. Fastapi.
[41] Apostolos Ampatzoglou, Stamatia Bibi, Paris Avgeriou, Marijn Verbeek, and Alexander Chatzigeorgiou. Identifying, categorizing and mitigating threats to validity in software engineering secondary studies. *Information and Software Technology*, 106:201–230, 2019.
[42] Shazibul Islam Shamim, Jonathan Alexander Gibson, Patrick Morrison, and Akond Rahman. Benefits, challenges, and research topics: A multi-vocal literature review of kubernetes, 2022.
[43] Zakkarija Micallef. Replication package for thesis: Comparative implementation study of mlops tools. https://github.com/zakkarija/mlops_comparison, 2025. Accessed: August 2025.
[44] Anders Köhler. Evaluation of MLOps Tools for Kubernetes.
[45] Aditya Dixit, Haseeba Rahman, and Nivedita Patel. Comparison of Deploying Deep Learning Models with MLflow on Different Cloud Platforms.
[46] Andrew Chen, Andy Chow, Aaron Davidson, Arjun DCunha, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Clemens Mewald, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, Avesh Singh, Fen Xie, Matei Zaharia, Richard Zang, Juntai Zheng, and Corey Zumar. Developments in MLflow: A System to Accelerate the Machine Learning Lifecycle. In *Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning*, pages 1–4, Portland OR USA, June 2020. ACM.

A Appendix - MLOps Tools Field Reports

This appendix provides detailed field reports for each MLOps tool evaluated across four dimensions: Usability, Functionality, Flexibility, and Vitality. Each report includes a description, systematic test plan with detailed results, and comprehensive scoring rationale.

A.1 DVC (Data Version Control)

Stack: MLflow **Component:** Storage and Versioning
Developer: Iterative **License:** Apache 2.0 **Description:** Git-like data versioning system that tracks large datasets alongside
Version: 2.3 **Date Evaluated:** June 4, 2025

code repositories. Uses pointer files to manage data while storing actual files in configurable remote storage backends. **Systematic Test Plan:**

(1) **Install & Configure on Ubuntu and macOS:** Started with `pip install dvc[s3]` to get S3 support. *Result:* Installation was refreshingly simple - just one pip command worked on both my Ubuntu and Mac machines. The documentation listed various storage options (s3, gdrive, gs, azure, ssh, hdfs, webdav, oss) which was helpful. Setting up MinIO took a bit of configuration but nothing too complex.
(2) **Initialize and Add IDEKO Dataset:** Set up DVC in my existing Git repository and tracked the CSV files. *Result:* Running `dvc init` created the necessary `.dvc` directory structure. When I ran `dvc add data/`, it generated a pointer file (`data.dvc`) with an MD5 hash and automatically added the actual data folder to `.gitignore`. This felt natural coming from Git - the workflow was immediately familiar.
(3) **Branch and Modify Dataset:** Created feature branches to test different dataset versions. *Result:* After deleting some columns from the CSV to simulate a data change, running `dvc add` updated the pointer file with a new hash. The seamless integration with Git meant I could track data changes just like code changes, though I had to remember to push both DVC and Git changes in the right order.
(4) **Test Rollback to Previous Versions:** Tried reverting the dataset two versions back. *Result:* This wasn't as seamless as I'd hoped. When I ran `git checkout HEAD 2`, I ended up in a detached HEAD state. Trying to commit from there created merge conflicts in the `.dvc` files. I had to work around this by checking out to a new branch from the old commit, running `dvc checkout` to sync the data, then merging back. The conflicts just showed MD5 hash differences - not very helpful for understanding what actually changed in the data.
(5) **Cross-platform Clone & Pull:** Cloned the repository on my Mac after setting everything up on Linux. *Result:* Running `dvc pull` worked seamlessly - I got the exact same dataset with matching MD5 verification. The commands behaved identically across platforms, which was reassuring for team collaboration. I didn't test Windows directly but noted that it requires WSL for full compatibility.
(6) **Branch Merge with Divergent Data:** Created two branches where each modified the dataset differently. *Result:* As expected, merging produced conflicts in the `.dvc` pointer files. Unlike code merges where you can see the actual differences, DVC only showed that the MD5 hashes differed. I had to manually examine the hashes and choose which version to keep - not ideal when you can't see what actually changed.
(7) **Pipeline Reproducibility:** Set up DVC pipeline stages using `dvc.yaml`. *Result:* The `dvc repro` command worked well for reproducible pipeline execution. I tracked my hyperparameter YAML files with DVC, which meant I could recreate exact training runs. The pipeline caching was a nice touch - unchanged stages were automatically skipped on subsequent runs. The `dvc dag` command gave me a clear visualization of dependencies between my data processing, training, and evaluation stages.
(8) **Storage Backend Testing:** Experimented with different storage backends. *Result:* S3 and MinIO behaved identically in my tests. I briefly tried SSH remote which worked but needed key setup. Local filesystem was fine for quick tests but obviously not suitable for team collaboration. Switching between backends was straightforward with `dvc remote add`.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Single pip command with optional backend extras ([s3], [gdrive], etc.). Clean conda installation also available. No compilation or complex dependencies required.
	Setup	High	`dvc init` creates structure in seconds, `dvc remote add` configures storage with one command. Git-like workflow familiar to developers.
	Configuration	Medium	Basic setup straightforward, but advanced features (cache management, shared cache, external dependencies) require manual `.dvc/config` editing. ACL and IAM integration needs careful configuration.
	Ease of Use	Medium	Commands mirror Git (add, push, pull, checkout) - intuitive for Git users. However, those without Git experience face steep learning curve. No free GUI - DVC Studio is paid enterprise tool.
	Documentation	High	Clear installation guide, practical examples for each storage backend, well-organized troubleshooting section. Community tutorials abundant.
	Overall	**High**	Excellent CLI usability for Git users, but accessibility limited without Git fluency or visual tools
Functionality	Completeness	Medium	Provides data versioning and pipeline orchestration but lacks built-in model registry or experiment tracking. Requires integration with other tools for complete MLOps.
	Appropriateness	High	Perfectly suited for ML data versioning needs. Integrates naturally with existing Git workflows. Handles both structured and unstructured data.

DVC (continued)

Dimension	Criterion	Rating	Rationale
	Reliability	Medium	Generally stable but merge conflicts in pointer files can be challenging. MD5 checksums ensure data integrity but conflict resolution requires manual intervention.
	Overall	**Medium**	Core features solid but requires careful handling during merges
Flexibility	Platform Support	Medium	Native on Linux/macOS. Windows requires WSL for full features. Available via pip, conda, brew, snap, choco.
	Integration Ready	High	8+ storage backends, CI/CD patterns documented, Python API available. IDE plugins for VSCode and JetBrains.
	Ease of Integration	High	Simple CLI and Python APIs. Environment variables for configuration. Hooks for automation.
	Modularity	Medium	Tight Git coupling beneficial but constraining. Can't use DVC without Git repository. Some teams need data versioning without code versioning.
	Overall	**Medium**	Very flexible within Git-based workflows, less so outside
Vitality	Community	High	13k+ GitHub stars, 300+ contributors, active Discord with 10k+ members. Used by major companies and research institutions.
	Maturity	High	Stable since 2017, version 2.x shows API stability. Wide production deployments documented.
	Development	High	Weekly commits, monthly releases. Iterative company provides commercial support and development.
	Documentation	High	Comprehensive docs, migration guides between versions, active blog with best practices.
	Overall	**High**	Extremely healthy and mature project with strong backing

A.2 LakeFS

Stack: Kubeflow **Component:** Storage and Versioning

Developer: Treeverse **License:** Apache 2.0 **Description:** Git-like version control for object storage. Provides data versioning

Version: 1.61 **Date Evaluated:** June 11, 2025

through metadata operations without duplicating actual data. Works with S3-compatible storage and can be deployed on Kubernetes. **Systematic Test Plan:**

(1) **Install LakeFS using the quickstart setup:** Used pip install followed by the quickstart command. *Result:* `pip install lakefs` went smoothly, then `python -m lakefs.quickstart` deployed the required components (LakeFS, MinIO, Postgres) as Docker containers. The web UI launched automatically and an admin user was created with credentials shown in the terminal. The quickstart was deceptively simple though. For a real production setup, I realized I'd need to deploy three separate components: LakeFS server on some compute instance, PostgreSQL or RDS for metadata, and S3 instead of MinIO for actual storage. Quite different from DVC's single pip install approach.

(2) **Create a repository with the IDEKO dataset:** Used the web interface to set up the repository and upload data. *Result:* Repository creation and dataset upload worked really smoothly through the drag and drop UI. The interface was intuitive and I appreciated having a visual way to manage the data. If the data had been in Parquet format instead of CSV, I could have queried it directly with LakeFS's built in DuckDB, which would have been nice.

(3) **Configure the lakefs CLI management tool (lakectl) to interact with the LakeFS Kubernetes pod:** Set up lakectl to control LakeFS from the command line. *Result:* This is where things got frustrating. Configuring lakectl should have been straightforward (just access key, secret key, and LakeFS URL) but the connection kept failing with unhelpful error messages. After extensive debugging, I finally discovered it was just a typo in my secret key. The error messages gave absolutely no useful hints about what was actually wrong, just generic authentication failures. Would have saved me a lot of time if the errors were more specific.

(4) **Create branches for different dataset versions:** Tested the branching functionality. *Result:* Branching worked well and didn't require copying all the data, which was a nice performance advantage. I could use familiar Git-like commands such as `lakefs diff` to see changes between commits. The zero copy branching felt efficient compared to other approaches I've used.

(5) **Perform rollback operations to previous commits:** Tested reverting to earlier versions. *Result:* Rollback using `lakefs branch reset` worked cleanly. Unlike my experience with DVC where rollbacks sometimes created merge conflicts in pointer files, LakeFS handled this smoothly. The history was preserved so I could still access what I called "future" commits if needed.

(6) **Merge branches containing different dataset modifications:** Tested different merge methods. *Result:* I was impressed that merging could be done three different ways. Through the CLI with lakectl, via pull requests in the web UI (which felt very GitHub-like), or even through the REST API. All three methods worked as expected. The UI pull request workflow was particularly nice with its visual diff and approval process.

Dimension	Criterion	Rating	Rationale
Usability	Installation	Low	Quickstart trivial but misleading. Production requires PostgreSQL, object storage, and LakeFS server setup. Each component needs proper sizing, backup, monitoring.

LakeFS (continued)

Dimension	Criterion	Rating	Rationale
	Setup	Low	Three moving parts increase complexity exponentially. Networking between components, authentication setup, storage configuration all required.
	Configuration	Low	CLI error messages unhelpful - "401 Unauthorized" for any credential issue. No indication if wrong endpoint, key, or secret. Took 40+ minutes to debug simple typo.
	Ease of Use	High	Web UI exceptional - drag-drop uploads, visual branch graph, intuitive navigation. Pull request workflow familiar from GitHub.
	Documentation	Medium	Quickstart excellent, production deployment guidance weak. Troubleshooting section missing. Kubernetes examples outdated.
	Overall	**Low**	Great UI undermined by complex architecture and poor error handling
Functionality	Completeness	High	Full Git-like semantics for data - branch, merge, diff, log, blame. Hooks for validation. RBAC at branch level.
	Appropriateness	High	Perfect for object storage versioning. Zero-copy critical for data lakes. S3 API compatibility enables tool reuse.
	Reliability	High	ACID transactions on metadata. Copy-on-write prevents corruption. PostgreSQL backend proven. MinIO or S3 for data reliability.
	Overall	**High**	Enterprise-grade data versioning capabilities
Flexibility	Platform Support	High	Runs on Kubernetes, Docker, bare metal. Supports S3, GCS, Azure Blob. Clients for Python, Go, Java, JavaScript.
	Integration Ready	High	S3 API compatibility means existing tools work unchanged. REST API comprehensive. SDKs well-designed.
	Ease of Integration	Medium	S3 compatibility helpful but requires understanding object storage patterns. Path conventions different from filesystem.
	Modularity	Medium	Kubernetes-native architecture limits standalone usage. Clean separation of components but requires full infrastructure stack - metadata (PostgreSQL), data (object store), control plane (LakeFS). Each component replaceable.
	Overall	**High**	Excellent fit for cloud-native architectures
Vitality	Community	High	4.2k GitHub stars, growing community. Slack active with regular engagement
	Maturity	High	Version 1.0+ production ready. Stable APIs and proven in production environments.
	Development	High	Frequent releases (bi-weekly). Cloud offering shows commitment.
	Documentation	Medium	Good basics but lacks depth. Few troubleshooting guides. Community examples limited.
	Overall	**High**	Strong project with active development and commercial backing

A.3 Feast (Feature Store)

Stack: Both **Component:** Storage and Versioning
Developer: Tecton **License:** Apache 2.0 **Description:** Feature store for ML pipelines that provides consistent feature computation and storage. Maintains offline features for training and online features for serving with point-in-time correctness. Helps prevent training-serving skew through registry-backed schema control.
Version: 0.44 **Date Evaluated:** June 14, 2025

Systematic Test Plan:

(1) **Install Feast:** Set up Feast using pip. *Result:* Installation succeeded on the first attempt with a single `pip install feast>=0.44` command. No native dependencies or Docker needed, which was nice.
(2) **Initialize a new feature repository and configure the backend feature store:** Set up the Feast project structure. *Result:* The `feast init` command created an example repository structure with all the required files. Got `feature_store.yaml` for backend storage and project settings, and `features.py` for feature schemas and data types. The ready-made template was helpful but there was still a learning curve to understand concepts like entities, feature views, and feature services. This took me some time to wrap my head around.
(3) **Register IDEKO dataset features with Feast:** Convert and register the IDEKO data. *Result:* Feast requires Parquet files, which is a columnar storage format that's more efficient than CSV for analytical workloads. Since IDEKO's dataset was in CSV format, I had to write a `convert_data.py` script to transform the data into Feast-compatible Parquet. The script also needed to add required columns like timestamps and entity IDs for each machine. Initially found the Parquet requirement a bit frustrating, but I could see why they chose it for the compression and query performance benefits.
(4) **Define and register initial feature schemas:** Set up feature definitions. *Result:* Configuring Feast involved modifying multiple files. I defined entities (machine_id) and created feature views with the f1-f4 sensor readings plus some computed features like rolling averages and anomaly scores. Running `feast apply` was reassuring as it showed an explicit diff before applying changes, which helped avoid mistakes. The registry gets stored as a protobuf in `data/registry.db`. The configuration felt like significant overhead for our smaller project, though I could see how it would pay off for larger teams.
(5) **Test schema evolution by updating a feature definition:** Modified feature definitions to test versioning. *Result:* I updated a rolling average window size from 10 to 20 samples. When I ran `feast apply`, it displayed exactly what changed. The CLI blocked deployment until I approved the change, then

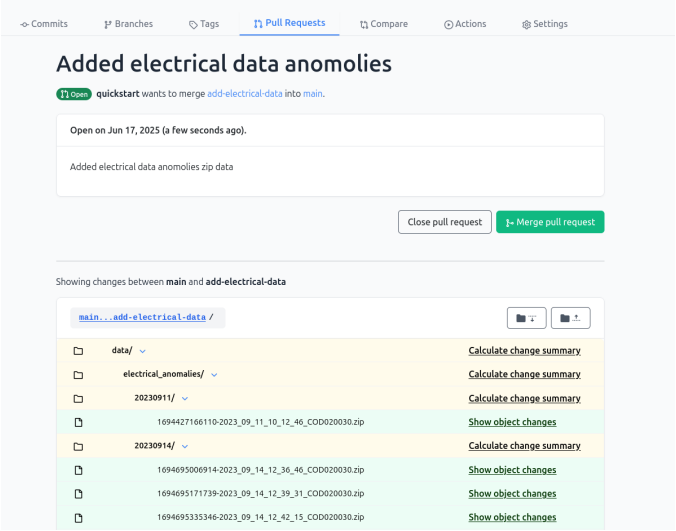

Figure 4: LakeFS version control interface showing a pull request for adding electrical data anomalies to the IDEKO dataset.

logged everything. Using `feast registry-dump` gave me a structured, diffable audit trail of every schema change in YAML format. This was much better than what would happen with raw Parquet plus DVC, where a rename would silently break the trainer.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Single pip install, no native dependencies. Template project helpful starting point.
	Setup	High	While concepts require initial study, setup process is straightforward with clear examples.
	Configuration	Medium	YAML configuration manageable but three files minimum (feature_store.yaml, features.py, plus conversion scripts).
	Ease of Use	Medium	Powerful once configured but heavyweight for small teams. ROI emerges at scale or with multiple models.
	Documentation	High	Comprehensive guides, clear examples. Concepts well-explained though learning curve remains.
	Overall	**High**	Initial learning investment pays off with robust feature management
Functionality	Completeness	Medium	Feature storage excellent but not full MLOps. No model registry, experiment tracking, or deployment.
	Appropriateness	High	Solves training-serving skew perfectly. Schema enforcement prevents silent failures. Point-in-time correctness essential.
	Reliability	High	Registry diffs prevent breaking changes. Protobuf storage robust. Materialization jobs idempotent.
	Overall	**High**	Excellent at its specific purpose
Flexibility	Platform Support	Medium	Python-first, experimental Go/Java SDKs. Runs on any OS with Python.
	Integration Ready	High	Providers for Spark, Kubernetes, Airflow. Plugins for various offline/online stores.
	Ease of Integration	Medium	SDK integration straightforward but version matching critical. Breaking changes between 0.x versions.
	Modularity	Low	Monolithic adoption required - can't use just online or offline store. All-or-nothing approach.
	Overall	**Medium**	Good ecosystem integration, low internal modularity
Vitality	Community	Medium	4.5k GitHub stars. Slack active but niche community. Corporate and academic users documented.
	Maturity	Medium	Pre-1.0 signals API instability. 0.44 shows progress but breaking changes still occur.
	Development	High	Tecton backing ensures development. Frequent releases (monthly). Active RFC process.
	Documentation	High	Official docs excellent. Community blogs growing. Video tutorials available.

Feast (continued)

Dimension	Criterion	Rating	Rationale
	Overall	**Medium**	Rapidly maturing but not yet stable

A.4 Apache Airflow

Stack: MLflow **Component:** Orchestration
Developer: Apache **License:** Apache 2.0
Version: 3.0 **Date Evaluated:** June 18, 2025

Description: Platform for programmatically authoring, scheduling, and monitoring workflows. Uses Python to define DAGs (Directed Acyclic Graphs) with rich operator ecosystem for integrating with external systems.

Systematic Test Plan:

(1) **Installation with Constraints:** Install via pip with version constraints. *Result:* Required downloading constraints file for Python 3.9 + Airflow 2.7. Installation command complex: `pip install apache-airflow==2.7.0 -constraint constraints-3.9.txt`. Constraints conflicted with existing packages - Feast pinned to 0.31 (needed 0.44). Manual override required, risking stability.

(2) **Initialize with Standalone:** Quick setup for development. *Result:* `airflow standalone` magical for development - initialized database (SQLite), created admin user, started webserver and scheduler. Password shown in terminal. UI accessible in 30 seconds. Warning: "Do not use in production" - production needs separate services.

(3) **Create IDEKO Pipeline DAG:** Implement DVC → preprocess → train → MLflow workflow. *Result:* DAG definition required learning: (1) Operator types (BashOperator, PythonOperator), (2) Task dependencies via » syntax, (3) XCom for passing data between tasks, (4) Jinja templating for dynamic values. Initial DAG failed - module imports, path issues. 6 iterations to working pipeline.

(4) **UI Functionality Testing:** Explore monitoring capabilities. *Result:* UI excellent for monitoring: Graph view shows DAG structure, Grid view displays run history, Gantt chart reveals bottlenecks, Logs accessible per task (but verbose with Airflow internals). Cannot create DAGs via UI - code only. Manually triggered runs work instantly.

(5) **Scheduling Verification:** Test cron and sensors. *Result:* Cron expression `0 2 * * *` ran nightly as expected. FileSensor watched for new data files. S3Sensor required AWS credentials configuration - added via Connections UI. Dataset-aware scheduling worked but undocumented.

(6) **MLflow Integration:** Log experiments from Airflow tasks. *Result:* No official MLflow operator. Created custom PythonOperator wrapping MLflow calls. Each DAG run mapped to MLflow experiment. XCom passed run IDs between tasks successfully. Integration code verbose but functional.

(7) **Failure Handling:** Test retries and alerting. *Result:* Task retries worked (default 2 attempts). Email alerts required SMTP configuration - sent on task failure and SLA miss. Callback functions enabled custom notifications. Recovery from mid-pipeline failures seamless.

(8) **Production Configuration Review:** Document production requirements. *Result:* Standalone inadequate for production. Required: (1) PostgreSQL/MySQL instead of SQLite, (2) CeleryExecutor or KubernetesExecutor for parallelism, (3) Redis/RabbitMQ for Celery, (4) Separate webserver, scheduler, worker processes, (5) Monitoring (Flower for Celery, Prometheus metrics). Estimated 2-3 days setup.

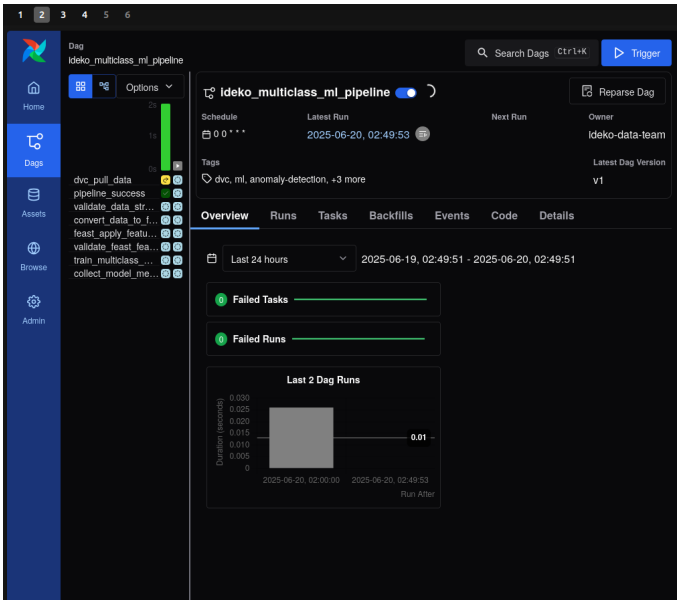

Figure 5: Screenshot of the Apache Airflow DAG interface showing the IDEKO multiclass ML pipeline execution.

Dimension	Criterion	Rating	Rationale
Usability	Installation	Medium	Constraints file complexity, dependency conflicts common. Standalone helps but production setup extensive.
	Setup	Medium	Database, executor, message broker all need configuration. Environment variables numerous.
	Configuration	Low	Python-based config complex and verbose. Connection management via UI helpful but overall configuration challenging.
	Ease of Use	Medium	Monitoring UI good but DAG syntax has learning curve. Requires Python expertise.
	Documentation	Medium	Comprehensive but scattered. Every feature documented but lacks cohesive tutorials.
	Overall	**Medium**	Powerful but complex - significant learning curve despite good UI
Functionality	Completeness	High	Scheduling, monitoring, alerting, distributed execution, SLA tracking, backfilling - all included.
	Appropriateness	High	Perfect for batch ML pipelines. Less suited for streaming. Handles complex dependencies well.
	Reliability	High	Battle-tested since 2014. Automatic retries. State recovery. Zombie task detection.
	Overall	**High**	Comprehensive orchestration platform
Flexibility	Platform Support	High	Linux, macOS, Windows (WSL). Docker, Kubernetes. Multiple executors.
	Integration Ready	High	200+ operators. AWS, GCP, Azure providers. Extensible plugin system.
	Ease of Integration	Medium	PythonOperator flexible but requires custom code. Integration requires effort.
	Modularity	Medium	Core components coupled but plugins modular. Can't easily swap components.
	Overall	**High**	Highly adaptable despite some integration complexity
Vitality	Community	High	35k+ GitHub stars. Apache governance. Massive adoption (Airbnb, Netflix, Twitter).
	Maturity	High	Production since 2014. Version 2.x stable. Well-understood patterns.
	Development	High	Monthly releases. 2000+ contributors. Active mailing lists.
	Documentation	Medium	Official docs extensive but poorly organized. Books published. Conference talks abundant.
	Overall	**High**	One of healthiest open-source data projects

A.5 Kubeflow Pipelines

Stack: Kubeflow **Component:** Orchestration

Developer: CNCF/Google **License:** Apache 2.0 **Description:** Platform for building and deploying portable, scalable ML workflows

Version: 1.1 **Date Evaluated:** July 1, 2025

using containers on Kubernetes. Each pipeline component runs in isolated pods with explicit artifact passing. **Systematic Test Plan:**

(1) **Install and setup Kubeflow Platform (includes Kubeflow Pipelines):** Selected a Kubeflow distribution and deployed it. *Result:* Multiple installation options complicated the selection process with no obvious default. I compared DeployKF (which bundles MLflow and Airflow but uses outdated versions) against Charmed Kubeflow and chose Charmed since it had the most recent Kubeflow version. Installation required Ubuntu 22, MicroK8s (Canonical's lightweight Kubernetes), and Juju (also by Canonical). The deployment took around 15 to 20 minutes to provision approximately 25 services. After port forwarding, I noticed the namespace and service names differed from the documentation, which was confusing.

(2) **Create a user account through Dex (OpenID Connect provider):** Set up authentication and user accounts. *Result:* Dex authentication worked with the default credentials from the tutorial. However, the user and namespace structure varied significantly between distributions. DeployKF creates team1, team2, and admin groups with corresponding namespaces, while Charmed Kubeflow only created an admin namespace. The documentation didn't clarify these distribution-specific architectural decisions, leaving me to figure it out through trial and error.

(3) **Create a pipeline that clones the code, pulls the dataset, trains and registers the model:** Built the complete ML pipeline. *Result:* This took over 50 hours total, making it by far the most frustrating tool yet. Initial pipeline creation hit SQL character set incompatibilities between the KFP SDK and the MySQL pod in the deployment. While LakeFS was our planned tool, I opportunistically tested DVC since our data was already versioned in it. DVC integration revealed critical credential conflicts: both DVC and Kubeflow's MinIO use identical AWS S3 credential formats but expect them in different namespaces. I spent hours debugging why DVC couldn't authenticate. Eventually switched to LakeFS which worked immediately since its Helm chart includes a pre-configured MinIO instance, avoiding the credential mess entirely. The UI could observe pipelines but not define them, everything had to be written in code.

(4) **Test the different pipeline upload methods:** Evaluated different ways to submit pipelines. *Result:* Manual upload was tedious: compile the Python script to YAML using the KFP SDK, upload via UI, create a pipeline, then an experiment, then finally run it. For iterative development this took way too long. Programmatic submission using `kfp.Client().create_run()` was better but still required authentication setup. The built-in Jupyter notebooks and VS Code environments simplified authentication with just a single line of code, but meant abandoning my familiar IDE with all its extensions. SDK v1 vs v2 syntax differences caused additional errors with no documentation explaining the changes.

(5) **Examine logs and debug information through UI and kubectl:** Test logging and debugging capabilities. *Result:* Logging visibility was poor. The UI only showed logs from the last failed component. To see previous components' logs, I had to use kubectl commands like `kubectl logs pod-name -n kubeflow-user`. There was no centralized log view. The filesystem was also invisible, so when trying to verify if git clone worked, I couldn't see where files were saved without using kubectl exec to get into the pod.

(6) **Re-run identical pipelines to verify caching behaviour:** Test the caching mechanism. *Result:* Caching functioned but created more problems than it solved. Failed runs were also cached, so when I changed code and reran, it just returned the old cached error. Disabling cache proved difficult due to undocumented differences between SDK v1 (which uses `execution_options`) and v2 (which uses `.set_caching_options()`). The documentation was wrong for both versions. Manual cache invalidation required deleting pods.

(7) **Force component pod failures to test retry and recovery mechanisms:** Test fault tolerance. *Result:* KFP detected terminated pods and automatically restarted them. The pipeline would pause at the failed component and resume once recovery completed, showing good fault tolerance. However, I encountered a critical infrastructure issue: the MySQL pod suddenly started consuming 16GB of RAM regardless of configuration attempts. I tried modifying Juju charm settings and editing the pod directly, but it consistently respawned with 16GB allocation. This brought my 32GB machine to a crawl. The only solution was deleting the entire cluster and starting fresh, losing a day of work.

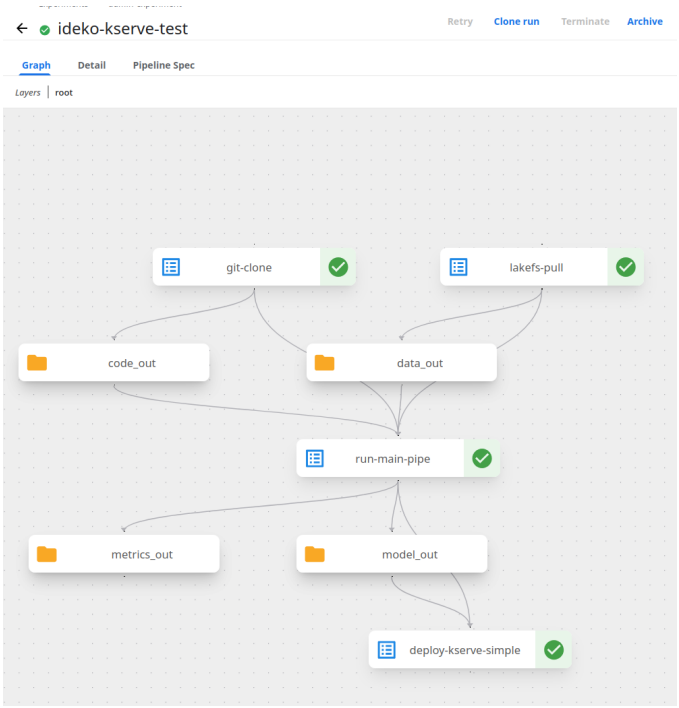

Figure 6: Screenshot of the Kubeflow Pipelines execution graph showing the successful completion of the IDEKO model training pipeline

Dimension	Criterion	Rating	Rationale
Usability	Installation	Medium	Multiple distributions complicate selection. Setup takes 15-20 minutes with prerequisites, no obvious choice. Each requires different prerequisites. Documentation fragmented.
	Setup	Low	Namespace structure varies by distribution. Authentication differs. Permissions complex.
	Configuration	Low	SQL charset issues, credential conflicts, SDK version confusion all common.
	Ease of Use	Low	Steep Kubernetes learning curve. Poor log visibility. Can't create pipelines in UI.
	Documentation	Low	Outdated, distribution-blind. SDK v1/v2 differences undocumented. Many broken links.
	Overall	**Low**	Power at excessive complexity cost
Functionality	Completeness	High	Full workflow engine, caching, retries, scheduling, artifact management, lineage tracking.

Kubeflow Pipelines (continued)

Dimension	Criterion	Rating	Rationale
	Appropriateness	High	Containerization ensures reproducibility. Kubernetes provides scaling. Artifact passing prevents hidden dependencies.
	Reliability	High	Kubernetes restarts failed pods reliably. Recovery mechanisms robust despite setup challenges.
	Overall	**High**	Feature-rich and reliable once configured
Flexibility	Platform Support	Medium	Kubernetes-only limits options but works across cloud providers.
	Integration Ready	High	Native Kubeflow integration. Works with Katib, KServe, MLMD, Feast out of box.
	Ease of Integration	Low	Rigid artifact passing. Component isolation complicates simple tasks. Type system inflexible.
	Modularity	High	Components theoretically swappable. Clear separation of concerns.
	Overall	**Medium**	Powerful within Kubeflow, painful otherwise
Vitality	Community	High	Large community under Kubeflow umbrella. Google backing. Spotify uses at scale.
	Maturity	Medium	Core stable but frequent API changes between SDK versions. SDK v1/v2 differences cause compatibility issues.
	Development	High	Active development continues with regular updates.
	Documentation	Low	Significant gaps. Distribution differences undocumented. Troubleshooting guides minimal.
	Overall	**High**	Strong community despite documentation issues

A.6 Optuna

Stack: MLflow **Component:** Hyperparameter Optimization

Developer: Preferred Networks **License:** MIT **Description:** Define-by-run hyperparameter optimization framework with state-of-the-art algorithms. Features automatic pruning of unpromising trials, parallelization support, and native integration with major ML frameworks.

Version: 4.4 **Date Evaluated:** June 21, 2025

Systematic Test Plan:

(1) **Install Optuna and the optional dashboard:** Set up Optuna with visualization capabilities. *Result:* Installation was simple with a single `pip install optuna` command for the core library. The dashboard was in a separate package, so I had to run `pip install optuna-dashboard` as well. I appreciated this modular approach since not everyone needs the visualization component. The installation went smoothly with no dependency conflicts.

(2) **Create an HPO study for IDEKO's Keras models:** Set up hyperparameter optimization for the models. *Result:* Optuna recognized all four of IDEKO's Keras models (NN, LSTM, CNN, RNN) right out of the box without needing any framework-specific configuration. The native Keras integration was seamless, it just accepted Keras objects directly for optimization. The define-by-run API felt intuitive since the search space was defined right in the objective function rather than in separate config files.

(3) **Configure a search space for learning rate, batch size, and hidden units:** Define the parameter ranges to explore. *Result:* The search space definition for learning rate (0.0001-0.1), batch size (16-128), and hidden units (32-512) was embedded directly in the objective function using calls like `trial.suggest_float()` for continuous parameters and `trial.suggest_int()` for integers. This follows Optuna's approach where everything is defined in code rather than YAML files, which I found more natural as a Python developer.

(4) **Execute an optimization study with the TPE sampler:** Run the hyperparameter search. *Result:* I ran 50 trials using the default Tree-structured Parzen Estimator (TPE) algorithm, which is a Bayesian optimization method. The best configuration appeared around trial 30, with subsequent trials confirming the convergence. I could see the real-time progress in the terminal which was helpful for monitoring.

(5) **Integrate Optuna with MLflow tracking:** Connect Optuna trials to MLflow experiments. *Result:* This initially failed with frustrating parameter conflict errors. MLflow's parameters are immutable (can't be overwritten), but Optuna needs to log parameters for multiple trials. The error "Parameter already logged" kept appearing. After some research, I found the solution: install `pip install optuna-integration` and use the `MLflowCallback`. This creates child runs for each Optuna trial, isolating the parameters. Only took a few lines of code to fix once I understood the pattern.

(6) **Test pruning functionality:** Evaluate early stopping of unpromising trials. *Result:* Optuna's MedianPruner automatically stopped trials that were performing below the median of previous trials at the same step. In my run, 12 out of 50 trials were pruned early, which saved computation time. The pruning decisions were logged, and the `KerasPruningCallback` integrated seamlessly with our Keras models.

(7) **Examine dashboard visualization and study diagnostics:** Explore the visualization capabilities. *Result:* The dashboard provided really comprehensive visualizations at localhost:8080. I could see optimization history plots showing how the objective improved over time, parallel coordinate diagrams that revealed parameter interactions, and parameter importance rankings. These visualizations gave immediate insights into the optimization process. I noticed batch size 64 was consistently optimal across different learning rates, which was an interesting pattern.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Single pip command. Optional dashboard separate. No compilation or system dependencies.
	Setup	High	Define-by-run API means no config files. Search space in Python code. Immediately productive.
	Configuration	Medium	MLflow integration required extra package and child-run pattern. Production storage needs database setup.
	Ease of Use	High	Pythonic API. Excellent defaults. Pruning automatic. Framework callbacks provided.
	Documentation	Medium	Examples exist but could be more comprehensive. API reference adequate.
	Overall	**High**	Smoothest HPO experience available
Functionality	Completeness	High	Multiple samplers (TPE, CMA-ES, Random, Grid), pruners (Median, Hyperband), storage backends, visualization.
	Appropriateness	High	Balances automation with control. Define-by-run natural for ML workflows.
	Reliability	High	Robust error handling. Automatic retries. SQLite corruption recovery. Clean trial isolation.
	Overall	**High**	Complete and dependable HPO solution
Flexibility	Platform Support	High	Pure Python - works everywhere. No compiled extensions. Container-friendly.
	Integration Ready	High	Native callbacks for TensorFlow, PyTorch, Keras, XGBoost, LightGBM, scikit-learn. MLflow, Weights&Biases, TensorBoard integrations.
	Ease of Integration	High	Callbacks handle framework specifics. Most integrations single import. Well-designed extension points.
	Modularity	High	Samplers, pruners, storage all pluggable. Custom implementations straightforward.
	Overall	**High**	Extremely flexible architecture
Vitality	Community	High	10k GitHub stars. Active and growing community. Japanese company backing.
	Maturity	Medium	Stable since 2018. Version 3.x shows API maturity. Production deployments documented.
	Development	High	Monthly releases. Quick issue responses. New samplers/pruners regularly added.
	Documentation	Medium	English/Japanese docs available but some areas lack depth.
	Overall	**High**	Healthy specialized community with strong momentum

A.7 Katib

Stack: Kubeflow **Component:** Hyperparameter Optimization

Developer: CNCF **License:** Apache 2.0 **Description:** Kubernetes-native hyperparameter tuning service. Supports multiple ML frameworks and optimization algorithms. Runs trials in isolated pods with automatic resource management.

Version: 0.18 **Date Evaluated:** July 4, 2025

Systematic Test Plan:

(1) **Access Katib UI through the Kubeflow dashboard:** Verify Katib is available. *Result:* Katib came pre-integrated in both DeployKF and Charmed Kubeflow distributions. It appeared as a dedicated tab in the Kubeflow dashboard without requiring any additional configuration or service deployment. The integration was seamless within the Kubeflow ecosystem, which was one less thing to configure.

(2) **Define an experiment through the UI:** Set up the hyperparameter search configuration. *Result:* Creating a Katib experiment required navigating through eight configuration sections in the UI: metadata, objective spec, algorithm selection (Bayesian/Random/Grid), parameters with ranges, trial template with container spec, metrics collector, early stopping rules, and parallel trial count. This was overwhelming initially but the form validation and dropdown menus for each field helped guide the configuration. The validation caught errors before submission which saved some debugging time.

(3) **Execute the hyperparameter search:** Run the optimization experiment. *Result:* Initial experiments failed with "Couldn't find any successful Trial" error, which wasn't very helpful. After debugging, I discovered Katib's containerized training requires parameters to be printed in a very specific format: `metric_name=value` to stdout. The existing IDEKO training scripts had to be modified to output exactly `accuracy=0.95`, not "Accuracy: 0.95" or JSON format. Even after fixing the output format locally, the training script still required careful adjustments to produce the exact output Katib expected within its containerized environment. This took hours to figure out.

(4) **Monitor trial progress and parallel execution:** Track the running trials. *Result:* The Bayesian optimization with 20 maximum trials and 4 parallel workers executed successfully once the format issues were resolved. The trials tab displayed real-time updates showing the different hyperparameter combinations being tested, with learning rates ranging from 0.0001 to 0.1, batch sizes from 16 to 128, and network architectures from 32 to 256 hidden units. Each trial ran in its own separate pod which was nice for isolation.

(5) **Validate resource allocation and pod management:** Check how resources are handled. *Result:* Each trial ran in its own isolated Kubernetes pod. After each trial completed, Katib automatically deleted the pod and freed the allocated CPU and memory resources, preventing resource accumulation from failed or completed experiments. However, accessing logs from individual trial pods before cleanup was frustrating. I needed to use `kubectl logs` commands because Kubeflow's log viewing capabilities were limited. The UI only showed "Failed" status without details, requiring manual pod inspection.

(6) **Test early stopping behaviour:** Verify the pruning mechanism. *Result:* The early stopping mechanism functioned as configured, terminating trials that showed no improvement after the specified iterations. Pod cleanup was automatic with no resource leaks observed, which was good for long-running experiments.

(7) **Examine parallel coordinates visualization:** Analyze the results visualization. *Result:* The parallel coordinates plot was excellent and elegantly revealed parameter relationships. I could see that lower learning rates combined with moderate batch sizes (around 64) consistently achieved better accuracy. The visualization was much better than looking at tabular results. However, there was a confusing issue where the UI marked the experiment as "successful" with a green checkmark despite not reaching the target metric. My experiment achieved only 50% accuracy against a 95% target but still showed as successful. This could mislead teams since "success" apparently meant "completed without error" rather than "achieved objective".

Figure 7: Katib parallel coordinates visualization showing the relationship between hyperparameters and model performance across multiple optimization trials.

Dimension	Criterion	Rating	Rationale
Usability	Installation	Medium	Ships with Kubeflow so no separate installation, but overall Kubeflow setup is complex.
	Setup	Low	Requires containerization. Specific metric format. Eight configuration sections.
	Configuration	Low	Verbose UI configuration. Rigid output requirements. No programmatic definition.
	Ease of Use	Low	High friction. Format debugging painful. Log access requires kubectl.
	Documentation	High	Within Kubeflow docs, well-covered. Examples for major frameworks.
	Overall	**Low**	Integration doesn't offset rigid requirements
Functionality	Completeness	High	Multiple algorithms, early stopping, parallel execution, resource management.
	Appropriateness	High	Perfect for Kubernetes environments. Pod isolation valuable.
	Reliability	Medium	Works when configured correctly. UI success indicators misleading.
	Overall	**High**	Capable but requires expertise
Flexibility	Platform Support	Medium	Kubernetes only. No local development. Tied to container infrastructure.
	Integration Ready	High	Native to Kubeflow. Works with any containerized framework.
	Ease of Integration	Medium	Metric format rigid. Container requirement adds complexity.
	Modularity	Medium	Can run standalone but designed for Kubeflow.

Katib (continued)

Dimension	Criterion	Rating	Rationale
	Overall	**Medium**	Best within Kubeflow ecosystem
Vitality	Community	High	Part of Kubeflow umbrella with large community.
	Maturity	Medium	Stable core but evolving configuration requirements. Eight-section UI configuration changes between versions.
	Development	High	Active development continues with Kubeflow releases.
	Documentation	High	Good coverage in Kubeflow docs.
	Overall	**High**	Strong within Kubeflow context

A.8 MLflow Tracking

Stack: MLflow **Component:** Experiment Tracking
Developer: Linux Foundation **License:** Apache 2.0 **Description:** Comprehensive experiment tracking system for ML workflows.
Version: 2.21 **Date Evaluated:** June 21, 2025

Logs parameters, metrics, artifacts, and models with automatic capture for supported frameworks. Provides web UI for visualization and comparison. **Systematic Test Plan:**

(1) **Install MLflow via pip:** Set up MLflow on the system. *Result:* Installation was super simple with just `pip install mlflow`. It fetched all dependencies without any conflicts or version issues, which was refreshing compared to other tools I'd tested.
(2) **Setup local MLflow server and dashboard:** Launch the tracking server and UI. *Result:* Running the MLflow tracking server required two simple shell commands. I ran `mlflow server` and `mlflow ui`, each just needing a port definition. The UI was immediately accessible at localhost. When starting the server, MLflow automatically deployed both the tracking backend and accompanying UI, which was convenient.
(3) **Register IDEKO training runs as separate MLflow experiments:** Configure experiment tracking for the models. *Result:* To enable tracking, I wrapped the training code in `mlflow.start_run()` context blocks and specified which parameters to log. This only required minor modifications to the existing scripts. I could log parameters, metrics, and artifacts within these blocks. The documentation actually recommends not wrapping the entire training in one big `start_run()` block because if there's a failure mid-flow, you need to manually clean up the invalid metadata, which I learned the hard way.
(4) **Enable automatic experiment tracking:** Test the autologging feature. *Result:* MLflow's autologging for Keras was incredibly simple. Just one line of code captured an insane amount of information: training metrics like loss and accuracy, hyperparameters including learning rate, batch size, optimizer settings, even things like class weight and shuffle settings. It also logged the model architecture and final trained model artifact. Everything appeared in the UI with fancy graphs displaying metrics over epochs. Zero manual logging required for all this, which was impressive.
(5) **Track experiment metadata and custom metrics:** Add custom tracking beyond autologging. *Result:* Custom metrics and tags were successfully logged using calls like `mlflow.log_metric()` and `mlflow.set_tag()`. These helped organize and identify different experimental configurations. All metadata was properly associated with their respective runs. I could track both training and inference runs, with parameters being particularly helpful for the training runs. The tracking was very comprehensive.
(6) **View what has been captured in the UI and examine available functionalities:** Explore the dashboard features. *Result:* The MLflow dashboard was split into three sections: Experiments, Models, and Prompts (though Prompts wasn't relevant for our ML models). The Experiments section showed all runs with details like when created, success status, duration, and source. Clicking on a run gave a complete overview with all metadata, visual graphs of model metrics, parameters used, and artifacts created. The UI had excellent search and filtering functionality where I could sort by date, datasets used, or filter by any metrics or parameters recorded. There were multiple ways of visualizing experiment runs and comparing them. The amount of data collected and how MLflow visualized it was probably unparalleled. Advanced filtering worked with queries like searching for specific batch sizes or accuracy thresholds.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Single pip install. No dependencies beyond Python. Works immediately.
	Setup	High	One environment variable (MLFLOW_TRACKING_URI). Defaults sensible.
	Configuration	High	Optional configs via environment. Most users need nothing beyond defaults.
	Ease of Use	High	Autolog eliminates boilerplate. UI intuitive. Search powerful.
	Documentation	High	Best-in-class. Quick-start accurate. AI assistant helpful.
	Overall	**High**	Industry-leading developer experience
Functionality	Completeness	High	Comprehensive tracking including parameters, metrics, artifacts, models, and datasets.
	Appropriateness	High	Perfect for experiment tracking. Covers all standard ML metadata.
	Reliability	High	Stable. Automatic retries. Clean error handling.
	Overall	**High**	Excellent core tracking capabilities
Flexibility	Platform Support	High	Windows, Mac, Linux. Docker. Kubernetes. Cloud services.
	Integration Ready	High	Autolog for 15+ frameworks. REST API. Python/R/Java clients.
	Ease of Integration	High	Usually single line of code. Well-designed abstractions.

MLflow Tracking (continued)

Dimension	Criterion	Rating	Rationale
	Modularity	High	Can be used independently or with other MLflow components.
	Overall	**High**	Excellent integration and flexibility
Vitality	Community	High	18k+ GitHub stars. Databricks backing. Widespread adoption.
	Maturity	High	Stable since 2018. Version 2.x indicates maturity.
	Development	High	Weekly releases. Quick issue resolution.
	Documentation	High	Constantly updated. Multiple languages. Video tutorials.
	Overall	**High**	Extremely healthy project

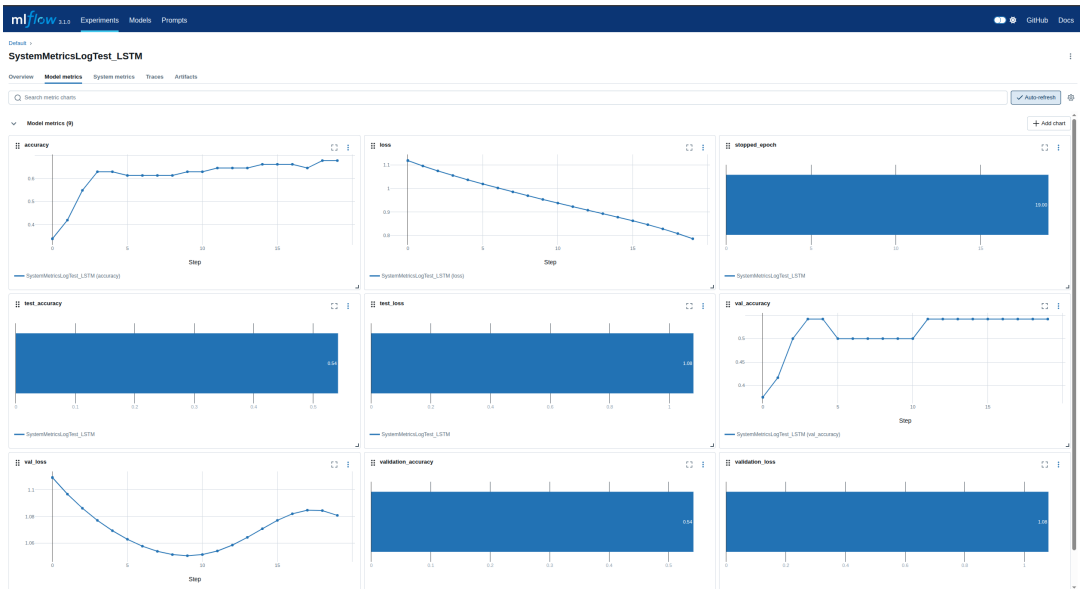

Figure 8: MLflow experiment tracking dashboard displaying comprehensive metrics for the LSTM model training run showcasing metrics across training epochs.

A.9 ML Metadata (MLMD)

Stack:	Kubeflow	**Component:**	Experiment Tracking
Developer:	CNCF/Google	**License:**	Apache 2.0
Version:	1.17	**Date Evaluated:**	July 8, 2025

Description: Library for recording and retrieving workflow metadata. Originally developed for Kubeflow, now maintained by Google for TensorFlow Extended. Provides artifact lineage and execution tracking without user interface.

Systematic Test Plan:

(1) **Install MLMD:** Set up and access MLMD. *Result:* I found MLMD was already pre-installed in both Charmed Kubeflow and DeployKF distributions, which initially seemed convenient. However, I quickly discovered that the metadata was stored in the cluster's MySQL pod, and accessing this data was frustrating. I had to navigate through Kubernetes abstractions and understand the MinIO storage layers, which was way more complex than I expected for just viewing metadata.

(2) **Update the training script to register metadata:** Integrate MLMD into the pipeline. *Result:* This was where things got really tedious. Unlike MLflow which just captures things automatically, I had to explicitly define schemas for literally everything I wanted to track. I needed to create ArtifactTypes for datasets and models, ExecutionTypes for training runs, ContextTypes for experiments, and worst of all, each individual hyperparameter had to be declared as a typed property before I could store anything. The amount of upfront work just to start logging was annoying.

(3) **Execute training and capture metadata:** Run training and store metadata. *Result:* My initial attempts to inspect what was actually being stored through the Kubernetes pods proved overly complex. I couldn't easily see what MLMD was capturing, which was frustrating. This led me to install MLMD as a standalone library outside of Kubeflow with a local MySQL backend. Finally I could get better visibility into what was actually being stored, but having to set up a separate instance just for debugging felt like a workaround for poor design.

(4) **Build a retrieval script to query saved metadata:** Create queries to access stored data. *Result:* Constructing queries was harder than I anticipated. I had to understand MLMD's property-based storage model first, which wasn't intuitive. When I tried to retrieve experiments with specific hyperparameter values, it required multiple API calls through specific typed property paths. What should have been a simple query turned into a multi-step process that took me a while to figure out.

(5) **Verify that each model is correctly linked to its source dataset:** Check lineage tracking. *Result:* This actually worked well. MLMD successfully tracked the complete workflow, showing which dataset was used, what preprocessing steps were applied, and which model resulted from training on that dataset. I could see how the raw data eventually became a trained model through every step and transformation in the pipeline. The dependency graph was comprehensive, which was one of the few bright spots in this evaluation.

(6) **Examine metadata visibility and accessibility:** Assess how to view and access metadata. *Result:* This was disappointing. MLMD doesn't offer any visualization for the training run through the Kubeflow pipelines UI, which seems like a basic requirement. When I accessed the database directly in my standalone instance, I could see MLMD's complex internal schema with typed properties and graph-based relationships, but there was no user-friendly way to view this. The lack of any native UI or visualization capabilities meant I had to write my own code just to see what was stored, which felt like MLMD was really just a backend storage system, not a tool meant for users to interact with directly.

Dimension	Criterion	Rating	Rationale
Usability	Installation	Medium	Pre-installed in Kubeflow. Standalone needs database setup.
	Setup	Low	Requires explicit type definitions. Complex schema design.
	Configuration	Low	Property-based model verbose. No defaults or conventions.
	Ease of Use	Low	No UI. Complex queries. Manual relationship management.
	Documentation	Low	Broken links. Scattered examples. Assumes deep knowledge.
	Overall	**Low**	Backend library, not user tool
Functionality	Completeness	Low	Only metadata storage. No visualization, comparison, or search UI.
	Appropriateness	Medium	Good for pipeline metadata. Overkill for simple experiment tracking.
	Reliability	High	ACID transactions. Consistent storage. Well-tested.
	Overall	**Medium**	Reliable storage, minimal features
Flexibility	Platform Support	High	SQLite, MySQL, PostgreSQL backends. Runs anywhere.
	Integration Ready	High	Integrates well within Kubeflow ecosystem. REST API available.
	Ease of Integration	Low	Complex API. Multi-step operations. Type system rigid.
	Modularity	High	Standalone library. Clean separation from other components.
	Overall	**High**	Flexible backend despite complex integration
Vitality	Community	Low	Limited standalone adoption. Hidden in Kubeflow/TFX.
	Maturity	High	Stable for years. Powers production systems.
	Development	Low	Slow development. Maintenance mode.
	Documentation	Low	Multiple broken links. Minimal examples.
	Overall	**Low**	Mature but neglected

A.10 MLflow Serving

Stack: MLflow **Component:** Model Serving
Developer: Linux Foundation **License:** Apache 2.0
Version: 2.21 **Date Evaluated:** July 18, 2025

Description: Simple model deployment through REST APIs. Automatically packages models with dependencies and serves predictions. Supports local deployment and cloud transitions.

Systematic Test Plan:

(1) **Install MLflow Serving dependencies:** Set up serving requirements. *Result:* I was pleasantly surprised to find that MLflow Serving comes bundled with MLflow, so there was no separate installation needed. The only additional requirement was the `virtualenv` package, which I had to install separately. This was refreshingly simple compared to other serving tools I'd tested. MLflow uses this to create isolated Python environments for each served model, which made sense for dependency management.

(2) **Prepare a model for deployment:** Get model ready for serving. *Result:* This was where MLflow really shined. The models I had already registered through MLflow Models were immediately available for serving. All the dependencies and signatures had been captured during the initial model logging process, so I didn't have to do any additional preparation. It just worked, which was a relief after dealing with other tools that required extensive reformatting.

(3) **Deploy a trained model using MLflow Serving:** Start the serving process. *Result:* The deployment succeeded with a single command, which was amazing. It automatically created a `virtualenv`, installed all the captured dependencies, and started a REST server exposing prediction endpoints on the port I specified. I kept expecting something to go wrong or require additional configuration, but it just worked out of the box. The simplicity was almost suspicious after my experiences with KServe.

(4) **Test the prediction endpoint with time series data:** Send LSTM predictions. *Result:* This is where I was really impressed. All my model architectures that accept three-dimensional tensor inputs (batch size, time steps, and sensor features) worked seamlessly without any custom preprocessing. I could send nested JSON arrays and MLflow correctly parsed them into the required format. The auto logging feature had automatically captured the correct input format specification during training, so I didn't have to write any wrapper code. This was such a contrast to KServe, where I had to deal with CSV uploads and custom preprocessing logic.
(5) **Integrate MLflow Serving into the Airflow pipeline:** Automate deployment workflow. *Result:* I successfully got the Airflow DAG to automate the complete deployment workflow. It selected the best-performing model from my HPO experiments, promoted it to production status in the model registry, started the serving process with health monitoring, and ensured graceful shutdown. However, this automation required me to write quite a bit of custom orchestration logic to coordinate all these steps. While it worked reliably once I figured it out, the amount of custom code needed was more than I initially expected.
(6) **Test batch predictions:** Verify multiple sample processing. *Result:* Batch predictions functioned correctly when I sent multiple samples in a single request. I used the standard MLflow format with "instances" as the JSON key, and it processed all sequences simultaneously, returning probability distributions for each sample. The fact that this worked without any special configuration was nice, though I did notice the single-threaded FastAPI server was a bottleneck for larger batches. Still, for development and testing, this was more than adequate.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Only virtualenv required. No complex dependencies.
	Setup	High	Single command deployment. Zero configuration needed.
	Configuration	High	Works with defaults. Ports and hosts configurable if needed.
	Ease of Use	High	Model to API in one command. Handles complex inputs automatically.
	Documentation	High	Clear about capabilities and limitations. Upgrade paths documented.
	Overall	**High**	Simplest serving solution available
Functionality	Completeness	High	Complete serving solution with REST API, health checks, and metrics.
	Appropriateness	High	Perfect for development and testing. Clear production migration path.
	Reliability	High	Stable operation with clean error handling and recovery.
	Overall	**High**	Solid serving solution for most use cases
Flexibility	Platform Support	High	Local, Docker, cloud services. Any OS with Python.
	Integration Ready	High	Same model format works with SageMaker, AzureML, KServe, etc.
	Ease of Integration	High	Model format standardized. Deployment targets well-documented.
	Modularity	High	Serving independent of tracking/registry. Models portable.
	Overall	**High**	Excellent portability and integration
Vitality	Community	High	Part of MLflow ecosystem. Widely adopted.
	Maturity	High	Stable serving solution proven in production.
	Development	High	Actively maintained as part of MLflow.
	Documentation	High	Honest about limitations. Clear upgrade guidance.
	Overall	**High**	Healthy as MLflow component

A.11 KServe

Stack: Kubeflow **Component:** Model Serving
Developer: CNCF **License:** Apache 2.0
Version: 0.15 **Date Evaluated:** July 14, 2025

Description: Kubernetes-native serverless inference platform. Provides autoscaling, canary deployments, and multi-model serving for ML models.
Systematic Test Plan:

(1) **Deploy SavedModel:** Serve TensorFlow model. *Result:* KServe pre-installed in Charmed Kubeflow. Created InferenceService YAML pointing to SavedModel. Deployment successful - TensorFlow runtime handled format automatically. Endpoint accessible via cluster ingress.
(2) **Test with Sample Data:** Send 20 timesteps. *Result:* Predictions incorrect. Investigation revealed model needed 18,000 timesteps for accurate anomaly detection. JSON payload size limits hit - request too large. REST API design assumed small inputs.
(3) **Implement CSV Upload:** Handle large inputs. *Result:* Created custom Python predictor class: Inherited from KServe base model, implemented preprocess() to parse CSV, load_model() to initialize, predict() for inference. Required understanding KServe's class hierarchy. Documentation sparse on custom predictors.
(4) **Debug Deployment Issues:** Fix serving errors. *Result:* Initial deployment failed with "Model not found". Logs unhelpful - generic 404 errors. After 8 hours debugging: model path in UI didn't match expected format, MinIO credentials not mounted correctly, init container failed silently. No clear error messages.

(5) **Pipeline Integration Attempt:** Connect to Kubeflow Pipelines. *Result:* Complete failure. KServe expects models at specific MinIO paths. Pipeline outputs artifacts differently. Artifact URLs shown in UI don't work for KServe. After 3 days: (1) Credentials wouldn't mount properly, (2) Path translation undocumented, (3) No working examples found. Abandoned integration.
(6) **Autoscaling Test:** Verify scale-to-zero. *Result:* Autoscaling worked when tested standalone. Scaled to zero after 60s idle. Scaled up on request (cold start: 15s). But couldn't test under load due to pipeline integration failure.
(7) **Model Registry Integration:** Link to version control. *Result:* Kubeflow Model Registry not included in either distribution. Documentation references it but not available. No automated versioning possible. Had to manage model versions manually.
(8) **Performance Comparison:** Measure vs MLflow serving. *Result:* When working: P50 latency similar (25ms), better concurrency handling, proper async processing. But setup time: KServe 3 days vs MLflow 30 minutes. Integration success: KServe 0

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Pre-installed with Kubeflow. No separate setup needed.
	Setup	Medium	Standard models work easily. Custom predictors require deep knowledge.
	Configuration	Low	JSON size limits problematic. Custom classes needed for real use cases.
	Ease of Use	Low	Error messages misleading. Debugging requires Kubernetes expertise.
	Documentation	Medium	Basic docs exist but integration patterns undocumented.
	Overall	**Medium**	Works for simple cases, fails for complex integration
Functionality	Completeness	High	Autoscaling, canary deployments, A/B testing, multi-model serving all included.
	Appropriateness	Medium	Good for Kubernetes environments but artifact passing incompatible with other Kubeflow components. Integration issues limit effectiveness.
	Reliability	Medium	Serving stable but integration fragile. Silent failures common.
	Overall	**High**	Powerful when working, integration unreliable
Flexibility	Platform Support	Medium	Kubernetes only. No local development story.
	Integration Ready	Low	Incompatible with Kubeflow Pipelines artifacts. Path issues unresolved.
	Ease of Integration	Low	Fundamental architectural mismatch with pipeline system.
	Modularity	Medium	Requires Kubernetes infrastructure. Can run standalone within Kubernetes but not without it.
	Overall	**Medium**	Flexible serving, inflexible integration
Vitality	Community	High	Part of Kubeflow. CNCF project. Active development.
	Maturity	Medium	Evolution from KFServing shows ongoing architectural changes. Version 0.15 indicates pre-1.0 status.
	Development	High	Regular releases. New features added frequently.
	Documentation	Medium	Core docs good but integration examples missing.
	Overall	**High**	Strong project hampered by integration issues

A.12 MLflow Models

Stack: MLflow **Component:** Storage and Versioning
Developer: Linux Foundation **License:** Apache 2.0
Version: 2.21 **Date Evaluated:** June 25, 2025

Description: Standardized format for packaging ML models with metadata, dependencies, and signatures. Enables deployment across diverse serving infrastructures by bundling model code, weights, environment specifications, and runtime signature in a portable file layout.

Systematic Test Plan:

(1) **Setup MLflow Models:** Verify module availability. *Result:* I found the `mlflow.models` module was already included in the core MLflow package, so I didn't need any additional installation beyond importing the package and the initial MLflow Tracking setup I'd already done. This was straightforward, no extra configuration needed.
(2) **Save the trained Keras model as an MLflow Model:** Package the trained model. *Result:* When I called `mlflow.pyfunc.log_model()` during training, it successfully and automatically stored each model bundled with its runtime signature, input example schema, and the list of dependencies in a requirements.txt file. Everything was captured automatically, which saved me from having to track dependencies manually.
(3) **Train multiple models with different metadata:** Generate distinct model artifacts. *Result:* My multiple training runs produced distinct model artifacts, and I noticed each artifact maintained its own metadata, version information, and associated metrics from the training process. This automatic separation was helpful, as I didn't have to worry about models overwriting each other.
(4) **View saved models in the UI Models tab and their lineage:** Check model visualization. *Result:* The UI Models tab clearly displayed every saved model artifact I had created. It showed the originating run ID, training metrics, model version, input parameters and dataset. This made it really easy for me to track which experiment produced which model, which was much better than trying to manage this manually.

(5) **Test model loading for inference:** Verify inference capabilities. *Result:* When I loaded the test model with MLflow's Python function wrapper using `mlflow.pyfunc.load_model()`, it returned a ready-to-use callable model object for making predictions. The custom class loader also successfully incorporated IDEKO's feature engineering code into the model pipeline, which let me bundle preprocessing with the model. This was exactly what I needed since our model required some feature computation before serving the input.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Included with MLflow core. No additional setup or dependencies required.
	Setup	High	Automatic integration with tracking. Two lines of code for model packaging.
	Configuration	High	Works with defaults. Automatic environment and signature capture.
	Ease of Use	High	Unified UI with tracking. Standard and custom loaders available.
	Documentation	High	Clear examples for all major frameworks. Deployment patterns well-documented.
	Overall	**High**	Seamless extension of MLflow ecosystem
Functionality	Completeness	High	Full model lifecycle: packaging, versioning, serving, deployment across platforms.
	Appropriateness	High	Perfect for standardized model packaging. Handles complex pipelines with custom loaders.
	Reliability	High	Signature validation prevents runtime errors. Environment specs ensure reproducibility.
	Overall	**High**	Comprehensive model management solution
Flexibility	Platform Support	High	Works across all major cloud providers and local deployments.
	Integration Ready	High	Native support for TensorFlow, PyTorch, scikit-learn, XGBoost, and custom models.
	Ease of Integration	High	Standard pyfunc interface plus custom loader option for complex scenarios.
	Modularity	Low	Requires MLflow Tracking as foundation. Cannot be used independently.
	Overall	**High**	Excellent portability despite MLflow dependency
Vitality	Community	High	Core component of widely adopted MLflow platform.
	Maturity	High	Stable API since MLflow 1.0. Production proven across industries.
	Development	High	Active development as part of MLflow. Regular improvements.
	Documentation	High	Comprehensive guides for all deployment scenarios.
	Overall	**High**	Mature and well-supported component

A.13 MLflow Datasets

Stack: MLflow **Component:** Storage and Versioning
Developer: Linux Foundation **License:** Apache 2.0
Version: 2.21 **Date Evaluated:** June 26, 2025

Description: Dataset versioning and lineage tracking integrated with MLflow experiments. Links data snapshots to training runs and models, providing automatic metadata capture including file path, content hash, schema, and record count for complete data provenance.

Systematic Test Plan:

(1) **Install MLflow Datasets:** Set up datasets component. *Result:* I found the MLflow Datasets library was included with the standard MLflow installation, so I didn't need any additional packages or configuration beyond the base setup. This was convenient since I already had MLflow installed.

(2) **Launch MLflow server and access dashboard:** Start server and verify UI. *Result:* Server startup remained identical to the MLflow Tracking setup I'd already done. Using the MLflow Datasets API required me to explicitly import it in my Python scripts using `import mlflow.data`. Once I imported it, MLflow Datasets' functionality was immediately accessible through MLflow's existing dashboard interface. I didn't need to learn a new interface, which was nice.

(3) **Register IDEKO's CSV dataset:** Log the initial dataset. *Result:* Each time I called `mlflow.data.log_dataset()`, it successfully captured IDEKO's dataset along with automatically generated metadata including file path, content hash, schema, record count, and timestamp. Adding the log call was trivial, just one extra line of code. The automatic metadata capture saved me from having to track this information manually.

(4) **Test dataset versioning after applying modifications:** Check version tracking. *Result:* After I modified the dataset by adding rows and adjusting features, the subsequent `log_dataset()` call automatically created a new dataset version. The metadata clearly reflected all the changes I made. This automatic versioning was helpful, though I noticed there were no built-in retention policies, which could become an issue with many versions.

(5) **Verify that each training run correctly links to its specific dataset version:** Test experiment association. *Result:* When I reran training scripts with different dataset versions, MLflow correctly associated each experiment training run with the specific dataset version it used. This was clearly displayed in MLflow's UI. I could easily see which run used which dataset, which made debugging much easier.

(6) **Explore dataset lineage in the UI:** Check visualization capabilities. *Result:* The MLflow Datasets tab provided a comprehensive version comparison, showing schema differences between versions and enabling lineage tracking from any model or run to its exact training dataset. However, I was

disappointed that I couldn't visualize the actual dataset changes or search by feature name. The UI showed metadata but not the data itself, which would have been useful. Still, being able to trace from a deployed model back to its exact training data snapshot was valuable for audit trails.

Dimension	Criterion	Rating	Rationale
Usability	Installation	High	Included with MLflow core. No additional setup required.
	Setup	High	One API call per experiment adds complete dataset lineage.
	Configuration	High	Automatic metadata capture. Works with default configurations.
	Ease of Use	High	Dedicated UI views. Schema comparison and version tracking intuitive.
	Documentation	High	Clear examples for common data formats. Integration patterns documented.
	Overall	**High**	Minimal effort for maximum data provenance
Functionality	Completeness	Medium	Excellent lineage but limited to MLflow ecosystem. No advanced data cataloging.
	Appropriateness	Medium	Good for ML reproducibility but limited compared to dedicated data catalogs.
	Reliability	Medium	Hash-based versioning works but lacks advanced features.
	Overall	**Medium**	Adequate functionality within MLflow scope
Flexibility	Platform Support	High	Works anywhere MLflow runs. Support for local files, S3, GCS, Azure.
	Integration Ready	Medium	Tight MLflow integration. Limited compatibility with external data catalogs.
	Ease of Integration	Medium	Single API call but requires MLflow ecosystem.
	Modularity	Low	Cannot be used independently of MLflow. Requires MLflow Tracking.
	Overall	**Medium**	Good within MLflow, limited outside
Vitality	Community	High	Core component of MLflow. Benefits from platform's widespread adoption.
	Maturity	High	Stable component of mature MLflow platform.
	Development	High	Active development as part of MLflow roadmap. Regular feature additions.
	Documentation	High	Well-integrated with MLflow docs. Clear examples and best practices.
	Overall	**High**	Strong backing as MLflow component

A.14 Kubeflow Model Registry

Stack: Kubeflow **Component:** Storage and Versioning

Developer: CNCF **License:** Apache 2.0

Description: Centralized model versioning and governance within Kubeflow. Intended to provide model lifecycle management similar to MLflow Model Registry.

Note: Not available in evaluated Kubeflow distributions (DeployKF or Charmed Kubeflow). Documentation references component but no implementation found. Evaluation could not be completed.