

B.Sc. (Business Analytics) Dissertation CA Report

Declarative Machine Learning Operations (MLOps) Platform

ExpOps

By

Poon Zhe Xuan

Department of Computer Science

School of Computing

National University of Singapore

2025/2026

B.Sc. (Business Analytics) Dissertation CA Report

Declarative Machine Learning Operations (MLOps) Platform

ExpOps

By
Poon Zhe Xuan

Department of Computer Science
School of Computing
National University of Singapore

2025/2026

Project ID: H414210

Advisor: Dr. Han Liang Wee Eric

Deliverables:

Report: 1 Volume

Contents

List of Figures	iii
List of Tables	iv
1 Introduction	1
1.1 Motivation	1
2 Literature Review	3
2.1 Approach: Declarative vs. Imperative	3
2.2 Reproducibility	4
2.3 Scalability	5
2.4 Project Objectives	6
3 Prototype	7
3.1 Design Considerations and Principles	7
3.2 System Architecture	8
3.3 Processes and Steps Pipeline	9
3.4 Pipeline Design	10
3.5 Distributed Execution on Clusters	11
3.6 Reproducibility Manager	12
3.7 Caching and Determinism	13
3.8 Metrics Logging and Charts	14
4 Future Plans	15
References	16
Appendices	18
Appendix A Titanic Project Example	A-1
Appendix B Example Code for Process and Step Decorator	B-1

Appendix C	Code for Static and Dynamic Charts	C-1
-------------------	---	------------

Appendix D	Key-Value Database Design	D-1
-------------------	----------------------------------	------------

D.1	Overview	D-1
D.2	Logical Model	D-1
D.3	Firestore Layout	D-1

Appendix E	Projects vs Runs	E-1
-------------------	-------------------------	------------

E.1	Overview	E-1
E.2	Projects	E-1
E.3	Runs	E-1

List of Figures

3.1	ExpOps System Design Diagram	8
3.2	Titanic Project DAG Pipeline with Processes and Steps	9
A.1	Process Graph of Titanic project	A-1
A.2	Process Graph of an ongoing run	A-1
A.3	Interactive process graph view showing detailed status information	A-2
A.4	Static Chart Node Modal	A-2
A.5	Dynamic Chart	A-3

List of Tables

2.1	Comparison of MLOps Platforms	3
3.1	Cache Key Composition for Process- and Step-Level Caching	13

Chapter 1

Introduction

1.1 Motivation

Machine learning (ML) has become a powerful toolkit for building intelligent systems across many industries (Amazon Web Services, 2025), yet the deployment and maintenance of ML applications in production remains notoriously complex. In practice, developing and launching an ML system is “fast and cheap,” but maintaining it over time is difficult and expensive (Sculley et al., 2015). Unlike traditional software, ML systems incur additional maintenance challenges. They include all the usual software upkeep issues, combined with ML-specific sources of technical debt (Sculley et al., 2015). This means that after the initial success of an ML model, keeping it running reliably (e.g., as data, requirements, or environments change) can require disproportionate effort.

Unlike conventional software development, machine-learning workflows are inherently experimental and iterative: practitioners routinely modify datasets, features, model architectures and hyperparameters in pursuit of incremental gains (Bermudez, 2024), which in turn induces tight couplings among data, code and configuration. This rapid iteration amplifies hidden dependencies and technical debt in production ML systems, making reproducibility and traceability first-order concerns (Elliott, 2025). Without rigorous provenance and versioning of datasets, models and configurations, results cannot be reliably recreated and failures become difficult to diagnose at scale. Consequently, production-readiness frameworks emphasize systematic testing, monitoring, data validation and configuration management as foundations for reliable, scalable operation (Elliott, 2025).

As an ML project matures from prototype to production, new operational challenges emerge. Models must be retrained with fresh data and integrated with downstream services on a regular basis and their performance should be continuously monitored to detect issues like data drift or accuracy degradation (Reddi, 2024). At the same time, the underlying infrastructure must handle these evolving workloads efficiently at scale, often leveraging distributed computing and parallel processing. In practice, large-scale ML training and inference are spread across multiple machines (i.e. distributed systems), using techniques such as data parallelism (splitting the dataset across nodes) or model parallelism (splitting the model itself) to accelerate computation (Belcic & Stryker, n.d.). Successfully operationalizing an ML pipeline thus involves not only building a good model but also managing continuous retraining, deployment integration, resource

scaling and performance monitoring in tandem.

Unfortunately, many existing solutions handle these steps ad-hoc. ML workflows are commonly scattered across multiple scripts, notebooks and specialized tools, which engineers then assemble together with a custom “glue code” (Scutari & Malvestio, 2023). This glue code typically consists of one-off scripts or wrappers to get data from one stage to the next, or to adapt outputs between libraries. Such brittle orchestration logic often leads to what Sculley et al. (2015) terms as “large masses of ‘glue code’” that lock in assumptions and create a maintenance nightmare. In practice, these fragmented pipeline “jungles” (Bogner et al., 2021) become hard to reproduce, debug, or evolve as projects grow. They accumulate operational debt (a form of technical debt) that slows down further experimentation and reliable deployment. For example, a small change in one script can break downstream steps and reproducing past results can be impossible if data or code versions were not tracked.

These challenges highlight the need for a unified framework that abstracts orchestration, enforces reproducibility and reduces the operational burden of maintaining ML workflows. We develop a declarative MLOps platform, i.e., users state what pipeline they want (data sources, transformations, training configuration and deployment target) rather than prescribing how to execute it. For example, a short YAML listing: `load(dataset=v1) → featurize(scale, encode) → train(model=xgboost, params=...)` is sufficient, the system infers dependencies and ordering. The specification is compiled into an explicit DAG that runs independent stages in parallel on distributed resources, exploits compute parallelism and caches intermediates for reproducible, incremental runs. This design reduces glue code and builds in retraining, rollback and auditability, which are functions that are typically brittle and labor-intensive in imperative, script-stitched pipelines.

Chapter 2

Literature Review

Modern MLOps platforms differ in how they define workflows, ensure reproducibility and scale execution. Table 2.1 summarises four representative systems: MLflow, Kubeflow, Flyte and Metaflow. The table highlights their core approaches (declarative vs. imperative), reproducibility & tracking mechanisms and scalability characteristics.

Table 2.1: Comparison of MLOps Platforms

Platform	Approach	Reproducibility	Scalability
MLflow	Imperative	Tracks experiments and environments via “Projects”.	Limited orchestration, tracking server may bottleneck.
Kubeflow	Imperative	Limited built-in tracking, consistency is user-managed.	High, Kubernetes-native with parallelized steps.
Flyte (Lyft)	Hybrid	Versioned tasks, data caching and automatic output logging.	High, built on Kubernetes with managed resources and parallelism.
Metaflow (Netflix)	Imperative	Versions data and steps, relies on user for deterministic runs.	Medium, steps can scale to AWS Batch or local compute.

2.1 Approach: Declarative vs. Imperative

One foundational design question for ML orchestration systems is whether to use a declarative approach (specifying what the pipeline should accomplish) or an imperative one (specifying how to perform each step). Traditional solutions have often been imperative: for example, many teams script pipelines with tools like Apache Airflow or write Kubeflow Pipelines using Python code, which tightly couples the workflow logic to the underlying infrastructure. This imperative pattern demands considerable platform-specific expertise and buries the high-level intent of the ML workflow in low-level implementation details. A recent survey of 23 MLOps platforms found that 76.4% expose primarily imperative APIs, leading to tightly coupled code and significant refactoring effort (averaging 41 hours) when porting pipelines across environments (Saxena, 2025).

By contrast, declarative MLOps has gained attention for its potential to abstract away infrastructure details and let users focus on pipeline logic. Declarative systems allow practitioners to define what steps to run and data dependencies, while the platform determines how to execute

them optimally. Only about 13.7% of surveyed platforms offered high-level declarative interfaces, but those that did showed a 67.8% reduction in implementation errors and nearly 59% decrease in maintenance effort across heterogeneous environments (Saxena, 2025). For example, Yang et al. (2025) propose a Declarative Data Pipeline (DDP) framework for large-scale ML services, where users describe data transformations and dependencies as JSON-based configuration blocks. The system automatically compiles these declarations into a directed acyclic graph (DAG), handles dependency resolution and optimizes execution through caching and state tracking. In this design, each dataset or intermediate artifact is treated as a first-class “anchor,” and the relationships between them fully determine the execution plan. This approach decouples workflow intent from execution logic, developers specify what data products to generate, while the framework decides how to schedule and execute them efficiently.

2.2 Reproducibility

Reproducibility is a cornerstone requirement in MLOps. Given the iterative, experiment-driven nature of ML workflows, teams need to reliably reproduce model training runs, compare results and trace data lineage. Many orchestration tools offer features to ensure that pipeline runs and model experiments can be repeated and audited. Experiment tracking systems such as MLflow put heavy focus on logging every detail of an experiment: parameters, data versions, model artifacts, metrics and even the code version for each run. MLflow’s tracking API and UI record these details, enabling researchers and engineers to compare model variants side by side and retrace steps when an experiment needs to be reproduced or debugged (Zaharia et al., 2018).

A crucial aspect of reproducibility is capturing the exact code, environment and dependencies used in an ML run. MLflow addresses this through its Projects component, which packages ML code in a reusable, reproducible form with an environment specification (Conda environment files or Docker containers) and entry points to run training or evaluation (Zaharia et al., 2018). Tools like Metaflow (Metaflow, n.d.) automatically version data artifacts and even snapshot code for each step, and Flyte (Flyte, n.d.) records every execution with a unique signature, making it possible to retrieve or recompute any artifact given the same inputs and code version.

Despite these features, there are still practical gaps in achieving “hands-off” reproducibility as many platforms still rely on the user to enforce certain practices. Manually setting random seeds in code is a commonly recommended step to make experiments repeatable (GeeksforGeeks, 2025), and most frameworks (TensorFlow, PyTorch, etc.) allow setting global seeds. However, relying only on user-set random seeds is error-prone and often insufficient. Even with a fixed seed, many operations in modern ML frameworks remain nondeterministic. For example, parallel GPU computations can sum floating-point numbers in varying orders, causing slight differences in results (Dong, 2025). Environment differences across workers or hardware types

further amplify this variability, causing models trained on separate nodes to diverge despite identical code and data (PyTorch, 2024).

For these reasons, reproducibility mechanisms must be enforced at the platform level, not left to individual scripts. A unified reproducibility manager that automatically sets and propagates global and per-task random seeds ensures consistent behavior across processes, models, and distributed workers. This guarantees that results remain deterministic even when experiments scale horizontally or execute on heterogeneous hardware (Matthew, 2025).

2.3 Scalability

As ML workloads grow, scalability becomes a core concern for orchestration platforms. Scalability has multiple dimensions: the ability to scale up (handle big data or intensive computation for a single workflow) and scale out (handle many concurrent workflows or serving many predictions), as well as efficiently manage distributed computing resources. Modern ML pipelines often need to process large datasets and train models with millions of parameters, which can exceed the capacity of a single machine. Orchestrators must therefore seamlessly extend to distributed execution environments. One common strategy for scalable ML pipelines is to build on cluster management systems like Kubernetes. Kubeflow (Kubeflow, n.d.) exemplifies this approach: it is an open-source toolkit designed to deploy and orchestrate ML workflows on Kubernetes, inheriting Kubernetes' ability to run on diverse cloud or on-premise infrastructure and to elastically scale services.

However, the trade-off is increased complexity. Running on a Kubernetes cluster introduces overhead in setup and maintenance. Users have to manage cluster configurations, container images and potentially Kubernetes-specific failure modes. As discussed earlier, Kubeflow's all-in approach to Kubernetes came at a cost: teams reported that Kubeflow could be fragile and burdensome to manage for large-scale workflows (Nelson, 2024). This suggests that while Kubernetes-based orchestration is powerful for scaling, it can lack flexibility for users who want to mix local and distributed work or who are not experts in DevOps.

Beyond compute scaling, the **scalability of tracking systems** themselves can also become a bottleneck. For instance, MLflow's centralized tracking server has been reported to struggle when the number of recorded runs grows very large (thousands of runs), where metadata retrieval and query latency degrade significantly (rom1504, 2020). This highlights that as teams scale up the number of experiments, even the experiment tracking component must be robust and scalable, not just the pipeline execution.

In addition to scaling across nodes and clusters, modern ML orchestration frameworks increasingly exploit task or process-level parallelism by executing independent DAG branches concurrently. For example, Flyte supports map-tasks that fan-out workloads across compute

nodes, enabling thousands of tasks to run daily in production environments (Jaishanker et al., 2022). In contrast, data-parallelism distributes a single model or dataset across multiple workers (Seong, 2024), where each processes different data shards and synchronizes results such as gradients. While process-parallelism enhances concurrency across tasks, data-parallelism boosts throughput within tasks, supporting both is essential for orchestration platforms to scale diverse ML workloads efficiently.

2.4 Project Objectives

As mentioned above, existing MLOps platforms reveal clear trade-offs. This project therefore proposes a **declarative, reproducible and scalable orchestration platform** that directly addresses these shortcomings.

Core Objectives and Addressed Gaps

1. Declarative Specification Format

What it does: Provides a high-level, machine-readable configuration (YAML/JSON) for defining data sources, transformations and model stages.

Gap addressed: Reduces the heavy reliance on imperative scripting seen in Kubeflow, improving portability and reducing refactoring effort across environments.

2. Validated Compiler and DAG Planner

What it does: Compiles the declarative specification into a validated **Directed Acyclic Graph (DAG)** that explicitly encodes task dependencies.

Gap addressed: Prevents hidden dependency errors and brittle pipelines common in imperative systems, ensures deterministic and portable execution.

3. Scalable Runtime with Caching and Reuse

What it does: Executes workflows seamlessly from local to cluster environments, automatically caching intermediate results.

Gap addressed: Overcomes scalability and efficiency limitations in MLflow (no orchestration) and Kubeflow (Kubernetes-only, high setup cost), avoiding redundant computation.

4. Built-in Reproducibility and Experiment Tracking

What it does: Automatically records manifests of code, data signatures, environment versions and outputs for every run, enforcing global random seeds and environment isolation.

Gap addressed: Eliminates issues from uncontrolled randomness and changes in software environments that cause inconsistent results across runs.

The platform will be evaluated through user surveys and pilot adoption by other students, who will apply it to their own machine learning projects to assess usability.

Chapter 3

Prototype

3.1 Design Considerations and Principles

ExpOps, short for *Experiment Operations*, is a declarative orchestration platform designed to streamline the management of computational experiments. While machine learning serves as the primary application domain, the platform’s architecture generalizes to any experiment-driven workflow that requires reproducibility, scalability and systematic execution.

ExpOps’ architecture is guided by core design principles that translate the goals of declarativity, reproducibility and scalability into concrete engineering decisions for execution, caching and observability.

- **Declarative over Imperative**

- *Rationale:* Simplify pipeline definition and minimize user logic coupling.
- *Design:* A YAML configuration file serves as the single source of truth, describing pipeline stages, dependencies and environment metadata. Each process and step is expressed as a pure, side-effect-controlled function with explicit I/O. The orchestrator automatically resolves the execution order, parallelism and scheduling.

- **Reproducibility and Determinism**

- *Rationale:* Guarantee identical outputs given identical inputs and code.
- *Design:* The system uses environment manifests for reproducible builds and enforces global and task-level random seeds. Hash-based identifiers (for code, configuration and data) ensure deterministic caching and result validation.

- **Scalability**

- *Rationale:* Enable seamless transition from local to distributed execution.
- *Design:* A scheduler interface abstracts compute providers (e.g., local threads, Slurm). All execution semantics remain consistent across backends, ensuring that results do not depend on execution topology.

- **Caching and Incremental Reuse**

- *Rationale:* Avoid redundant computation and enable efficient re-runs.

- *Design*: Each step result is indexed by a composite hash of its inputs, configuration and normalized function source. Deterministic hashing guarantees that only modified nodes are recomputed, supporting partial pipeline invalidation.
- **Data and Metadata Management**
 - *Rationale*: Support fast, isolated lookup and long-term artifact traceability.
 - *Design*: Metadata is stored in a key-value database for exact-match lookups, while large artifacts reside in an object store.
- **Extensibility and Maintainability**
 - *Rationale*: Allow independent evolution of system components.
 - *Design*: Interfaces for compute, storage, environment and tracking are defined via pluggable adapters, enabling incremental integration of new backends.

3.2 System Architecture

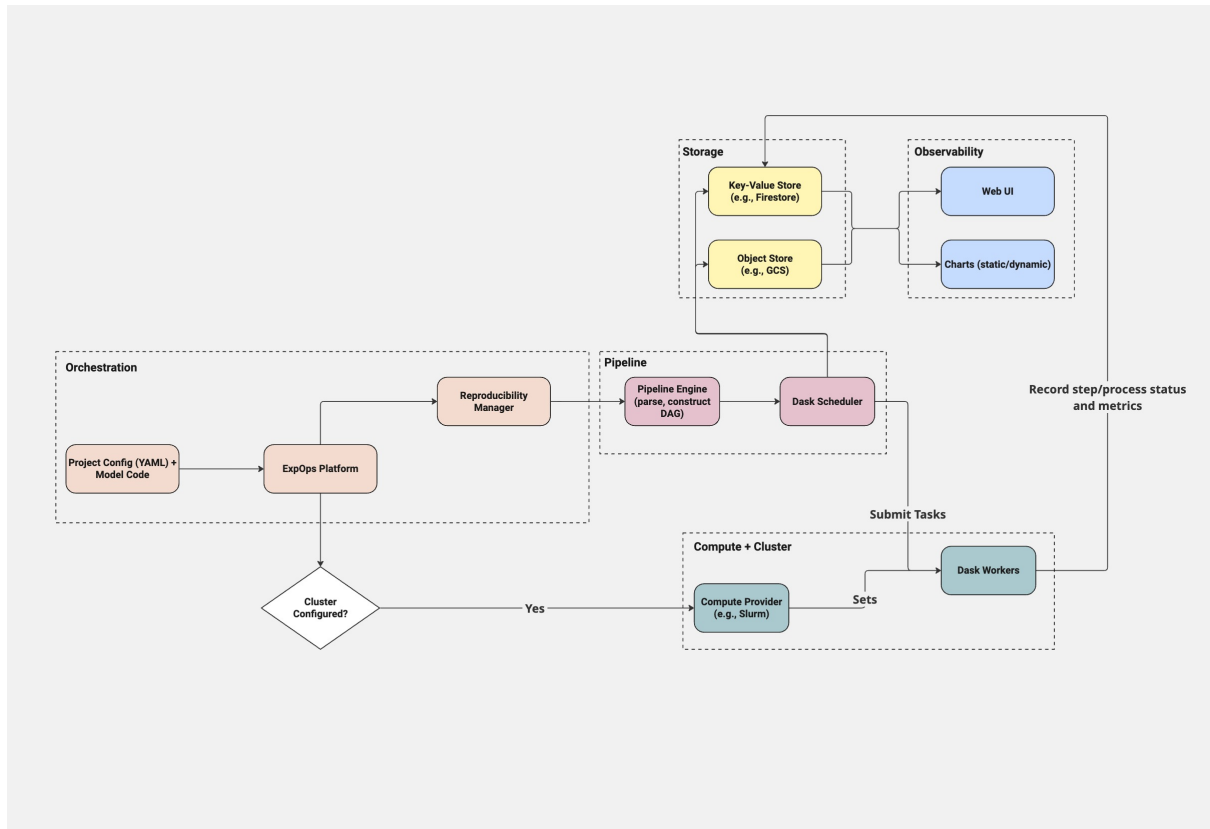


Figure 3.1: ExpOps System Design Diagram

Figure 3.1 illustrates the overall system architecture of ExpOps. The following sections follow this workflow’s logical order and will use a simple *Titanic* dataset example to illustrate

each component in practice. Refer to Appendix A for more details on the Titanic project example.

3.3 Processes and Steps Pipeline

ExpOps organizes machine learning workflows through a hierarchical structure of **processes** and **steps**. Processes are high-level workflow units that group related computational steps together while steps are atomic computational units within processes.

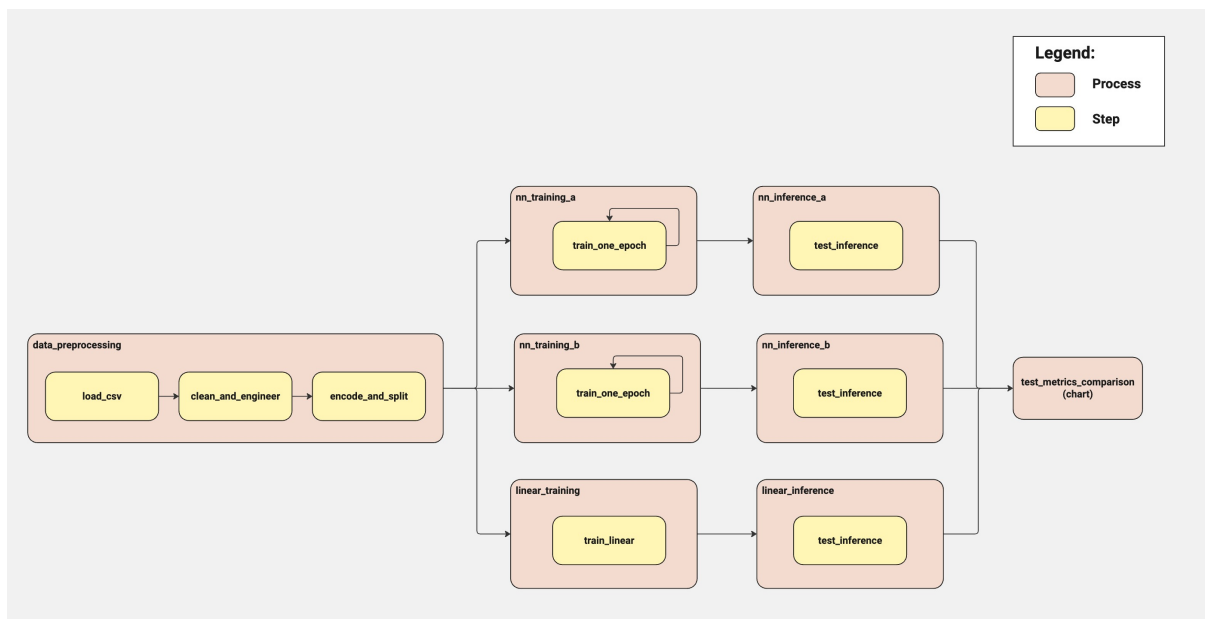


Figure 3.2: Titanic Project DAG Pipeline with Processes and Steps

3.3.1 Directed Acyclic Graph (DAG) Design

ExpOps represents workflows as a **Directed Acyclic Graph (DAG)**, where nodes denote processes and edges capture dependencies. This structure enables **process-level parallelism** by allowing independent branches to execute concurrently. For example, in Figure 3.2, the pipeline parallelizes the training of the two separate neural network models and another linear model. This design supports declarative orchestration: users specify only logical dependencies, while the system derives a deterministic execution order from the DAG, forming the foundation for distributed task scheduling across backends.

3.3.2 Process

Processes serve as the primary building blocks of ML pipelines and are defined using the `@process()` decorator.

Process code is defined with two input parameters: `data`, which is a dictionary containing the results of upstream processes keyed by upstream process name and `hyperparameters`, which is a dictionary containing the hyperparameter configuration for the process. The platform automatically injects upstream process outputs into the `data` dictionary based on the pipeline's dependency configuration, where each key corresponds to an upstream process name and the value contains that process's output results. The upstream results are first retrieved from the in-memory results in the context (elaborated in subsection 3.3.4), with a fall-back to loading from the cache if necessary. An example of process code is in Appendix B.

3.3.3 Step

Steps represent individual operations nested within processes that are executed sequentially and are defined using the `@step()` decorator. The results of individual steps are cached as well, tied to their process name, config, input parameters and function definition, this is further elaborated in section 3.7. As seen in Figure 3.2, steps can also form internal loops, such as iterative training across epochs. Additionally, step functions can be defined once and reused across multiple processes, promoting modularity and reducing code duplication.

3.3.4 Shared Context (Inter-Process Communication)

ExpOps provides transparent context and state management which provides access to outputs from previous processes/steps in memory, it operates behind the scenes without direct user interaction. Users never directly access the context system, they simply write processes and steps with standard parameters. This allows the platform to maintain reproducibility and declarative transparency, users write processes with standard parameters, while the system automatically tracks and injects the required data and state behind the scenes. In distributed mode, each worker gets its own copy of the context and the workers do not share memory or context state directly.

The context system also works with `log_metric` (elaborated in subsection 3.8.1) to enable saving of metrics to processes and steps. As the context system uses thread-safe context variables to track the current execution process and step, this helps to determine where to store the metrics and which process/step they belong to.

3.4 Pipeline Design

3.4.1 Data Flow and Dependencies with NetworkX

ExpOps implements its DAG using the NetworkX library (NetworkX Developers, n.d.), a Python package for constructing and analyzing graph structures. NetworkX offers a flexible

API for defining nodes, edges, and attributes, making it well suited for modelling workflow dependencies between processes. Its built-in algorithms for cycle detection and topological sorting are leveraged to validate pipeline integrity and prevent circular dependencies before execution.

```
pipeline:
  process_adjlist: |
    data_preprocessing nn_training_a
    data_preprocessing linear_training
```

The parser in ExpOps converts this adjacency list into a NetworkX DiGraph structure and validates that it forms a proper DAG (no circular dependencies).

3.4.2 Dask Scheduler Integration

To execute the pipeline efficiently, ExpOps integrates the Dask parallel computing library (Dask Development Team, n.d.). Dask is a flexible, open-source framework for distributed computation in Python, capable of scaling workloads seamlessly from a single machine to large clusters. It operates by constructing a **task graph**, a structure similar to NetworkX’s DAG, where each task represents a computation and edges represent dependencies.

The key reason for choosing Dask is its *dynamic task scheduling* and native compatibility with Pythonic workflows. Unlike static batch schedulers, Dask automatically determines execution order based on task dependencies, parallelizes independent tasks and handles worker distribution transparently. This allows the platform to scale from local development (threaded or process-based execution) to distributed backends (e.g., Slurm) without modifying user code.

3.5 Distributed Execution on Clusters

ExpOps provides support for distributed execution across compute clusters through a cluster management system built on Dask’s distributed computing framework (Dask Development Team, n.d.). This enables users to scale their machine learning pipelines across multiple nodes and cores while maintaining reproducibility and efficient resource utilization.

The platform supports multiple cluster provisioning backends through a pluggable provider architecture, with Dask managing job submission and worker orchestration. When cluster providers are unavailable, the system automatically falls back to local multi-threaded execution, ensuring that pipelines can always execute regardless of cluster availability.

Users can specify cluster settings, such as the provider, worker count, per-worker cores/memory directly in a separate `cluster_config.yaml` file, which will then be parsed at startup to initialise the chosen cluster backend with the specified resources and parameters.

3.6 Reproducibility Manager

ExpOps incorporates a comprehensive **Reproducibility Manager** that ensures reproducible machine learning experiments through systematic environment management and global random seed configuration. This is crucial for maintaining consistency across different runs and enabling proper scientific reproducibility.

The platform supports an automated setup and configuration of isolated virtual environments using multiple backend systems (conda, venv, pyenv). The user just needs to specify in the config the type of virtual environment they wish to use, the name of the environment and create a file with the dependencies (or inline in the config). The platform automatically creates separate environments for training and reporting phases, ensuring that visualization and analysis dependencies do not interfere with model training environments.

```
environment:
  venv:
    name: "pred-env"
    requirements_file: "projects/pred/requirements.txt"
  reporting:
    name: "pred-env-reporting"
    requirements_file:
      → "projects/pred/charts/requirements.txt"
```

In distributed execution scenarios, the platform ensures that all workers use the same Python interpreter and virtual environment as the main driver process. Specifically, when provisioning a cluster, the system passes `python=sys.executable` to Dask and prepends the virtual-environment's `PYTHONPATH` within each worker's launch script. This ensures that all worker processes start with the exact interpreter and environment used by the driver (Dask Developers, 2025), ensuring reproducible execution throughout the distributed system.

The reproducibility manager also automatically sets the global random seed as well as a global NumPy random seed during initialization. Additionally, if PyTorch and TensorFlow are available, their random seeds are also set. This setup covers most mainstream Python ML stacks, for example, scikit-learn (via NumPy's RNG), SciPy, PyTorch and TensorFlow, providing end-to-end seed control. During distributed execution, task-level deterministic seeding ensures reproducibility across workers and threads. The system exports a base seed and propagates it to all Dask workers, maintaining identical results across different worker assignments or thread scheduling.

3.7 Caching and Determinism

The platform provides caching at both process and step levels. Cache keys are derived from three independent SHA-256 hashes:

Table 3.1: Cache Key Composition for Process- and Step-Level Caching

Hash Type	Process-Level Definition	Step-Level Definition
input (ih)	Recursive upstream signatures (each predecessor's <code>ih/ch/fh</code>).	Call-time arguments to the step function.
config (ch)	Global configuration combined with this process's hyperparameters.	Global configuration used at runtime.
function (fh)	Composite hash derived from the process runner's normalized abstract syntax tree (AST) and signature, including nested <code>@step</code> ASTs and referenced step function hashes	Normalized AST and signature of the original <code>@step</code> function.

An **abstract syntax tree (AST)** represents the structural form of source code, abstracted from syntax details such as whitespace or comments. By hashing normalized ASTs rather than raw source text for the function hashes, the platform ensures that semantically identical code (e.g., with minor formatting changes) yields the same function hash, improving cache stability.

This design ensures that changes in inputs, configuration or code invalidate old results deterministically. The cache artifacts are written to stable absolute paths under the local filesystem and Object Store (e.g. Google Cloud Storage) when configured by the user. The index that maps the hashes to the cache path is stored in the configured key-value backend (e.g., Firestore, Redis) to enable $O(1)$ lookups, see Appendix D for more details.

3.7.1 Process and Step-level caching

Before scheduling a process node, the executor computes unified signatures using the process graph and context, combining the three hashes deterministically. A strict key-value lookup is then performed for the process. If it hits, the executor would schedule a lightweight "cached" placeholder while hydrating the process result into the driver context. If it misses, the executor schedules a real worker task.

When a `@step` function is called, the wrapper binds the original function's signature to actual arguments and computes the three hashes. After which, it performs a strict KV lookup. On hit, the cached result is returned and attached into the current `StepContext`. On miss, the step executes normally, its result being recorded.

3.8 Metrics Logging and Charts

Beyond deterministic caching, reproducible experimentation also depends on tracking of performance metrics. ExpOps includes an integrated metrics logging and visualization layer to monitor and analyze pipeline executions.

3.8.1 Log Metrics

ExpOps includes a lightweight metric-logging subsystem integrated into the shared context. Each running process or step can emit key-value metric pairs that are automatically routed to the configured key-value backend under a unique **probe path** (`<process>/<step>` or `<process>` if no step is active). This design enables downstream visualization components to subscribe selectively to relevant data streams without requiring explicit user wiring.

Metrics are typed and versioned for consistency. Numeric values are stored and indexed by step number, while non-numeric objects (e.g., lists) are serialized into JSON-safe snapshots.

3.8.2 Charts

Charts in ExpOps visualize logged metrics and outputs, enabling users to monitor training progress and evaluate results either in real time or after execution.

Static Charts

Static charts are executed as reporting tasks and produce chart artifacts (e.g., PNG).

Function Invocation: Functions are registered via the `@chart()` decorator. See Appendix C for an example code for static charts.

Data Access: For static charts, the platform resolves the configured `probe_paths` in the config above and injects a metrics dictionary directly into the chart function.

Execution: Static charts are integrated as specialized DAG nodes that can execute in parallel with other processes or after specific dependencies.

Dynamic Charts

Dynamic charts are rendered in the UI. They subscribe to live metrics via the chart's configured `probe_paths` and update on a fixed polling cadence. This is useful for charts that need to be updated in real-time, such as monitoring model loss curves across epochs.

Chapter 4

Future Plans

Thus far, ExpOps' prototype has focused on executing pipelines whose computational logic is authored by users in plain Python. Going forward, we will broaden the platform's reach and ease-of-use by making common ML workflows possible without explicit code.

Limitations of Reproducibility Manager

The current approach of the reproducibility manager comes with the assumption that the virtual environment is initialized on a file system that is accessible to all workers. We plan to address this limitation by distributing versioned container images. Another limitation is that the initialization of global random seeds might not cover components with independent RNGs that require explicit random seeds in model initialization. This will be addressed by injecting controlled random seeds into such components at runtime.

Component Library and Adapters

We will ship adapters for widely used ecosystems (e.g., scikit-learn and PyTorch), additional distributed execution backends (e.g., Ray) and a growing catalog of pre-defined processes (data splitting, scaling/encoding, model training, evaluation, and reporting). To further broaden accessibility, the platform will also provide a visual workflow builder that allows users, especially those less familiar with programming, to compose end-to-end pipelines through an interactive "drag and drop" interface rather than manual scripting.

Data Parallelism

The current prototype supports process-level parallelism by executing independent branches of the DAG concurrently. Next, we will add data-parallel execution for a *single* model. A practical use case of this data-parallel capability would be training a large CNN on millions of images: the dataset can be partitioned across several workers, each replicating the same model on different data shards, then aggregating gradients at each synchronization point to accelerate throughput.

Multi-Seed Replication

Reproducibility remains central, but exploration across randomness is equally important. We will introduce graph-level replication over random seeds: selected nodes (or subgraphs) can be declared *seed-mapped*, automatically fanning out into multiple replicas with distinct seeds.

References

- Amazon Web Services. (2025, October 18). *What is MLOps?* <https://aws.amazon.com/what-is/mlops/>
- Belcic, I., & Stryker, C. (n.d.). *What is distributed machine learning?* IBM Think. <https://www.ibm.com/think/topics/distributed-machine-learning>
- Bermudez, L. (2024, March 12). *Overview of MLOps* [Published in *machinevision*]. Medium. <https://medium.com/machinevision/overview-of-mlops-a07053fc2a80>
- Bogner, J., Verdecchia, R., & Gerostathopoulos, I. (2021). Characterizing technical debt and antipatterns in ai-based systems: A systematic mapping study. *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, 64–73. <https://doi.org/10.1109/TechDebt52882.2021.00016>
- Dask Developers. (2025). *Manage environments*. Dask. <https://docs.dask.org/en/latest/software-environments.html>
- Dask Development Team. (n.d.). *Dask*. <https://www.dask.org/>
- Dong, T. (2025, September 13). *Defeating nondeterminism in LLM inference: What it unlocks for engineering teams*. Propel. <https://www.propelcode.ai/blog/defeating-nondeterminism-in-llm-inference-ramifications>
- Elliott, S. (2025, March 3). *Reproducible workflows for compound AI: Reliable and scalable AI development*. Union.ai. <https://www.union.ai/blog-post/reproducible-workflows-for-compound-ai-reliable-and-scalable-ai-development>
- Flyte. (n.d.). *Flyte*. <https://flyte.org/>
- GeeksforGeeks. (2025, August 2). *Reproducibility in machine learning*. GeeksforGeeks. <https://www.geeksforgeeks.org/machine-learning/reproducibility-in-machine-learning/>
- Jaishanker, V. S., Bain, A., & Parthasarathy, V. (2022, October 4). *Flyte map tasks: A simpler alternative to apache spark*. <https://flyte.org/blog/flyte-map-tasks-a-simpler-alternative-to-apache-spark>
- Kubeflow. (n.d.). *Kubeflow*. <https://www.kubeflow.org/>
- Matthew, B. (2025, June 11). *Model versioning and reproducibility challenges in large-scale ML projects* [Article page on ResearchGate; uploaded Jun 11, 2025]. https://www.researchgate.net/publication/392595159_Model_Versioning_and_Reproducibility_Challenges_in_Large-Scale_ML_Projects
- Metaflow. (n.d.). *Metaflow*. <https://metaflow.org/>
- Nelson, D. (2024, January 26). *From kubeflow to flyte: A more reliable ML orchestration foundation*. aiXplain, Inc. <https://aixplain.com/blog/from-kubeflow-to-flyte-a-more-reliable-ml-orchestration-foundation/>

- NetworkX Developers. (n.d.). *Networkx*. <https://networkx.org/>
- PyTorch. (2024, November 26). *Reproducibility*. PyTorch. <https://pytorch.org/docs/stable/notes/randomness.html>
- Reddi, V. J. (2024). ML operations. In *Machine learning systems*. Harvard University. <https://www.mlsysbook.ai/contents/core/ops/ops.html>
- rom1504. (2020, April 18). *[fr] make MLflow tracking backend scale beyond thousand of runs* [Issue #2730 in mlflow/mlflow]. GitHub. <https://github.com/mlflow/mlflow/issues/2730>
- Saxena, S. (2025). Declarative mlops pipelines for enterprise platforms: A domain-specific language approach. *Sarcouncil Journal of Engineering and Computer Sciences*, 4(8), 34–41. <https://sarcouncil.com/download-article/SJECS-90-2025-34-41.pdf>
- Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). Hidden technical debt in machine learning systems. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, & R. Garnett (Eds.), *Advances in neural information processing systems* (Vol. 28). https://papers.nips.cc/paper_files/paper/2015/file/86df7dcfd896fcdf2674f757a2463eba-Paper.pdf
- Scutari, M., & Malvestio, M. (2023, April 22). Designing and structuring pipelines. In *The pragmatic programmer for machine learning*. <https://ppml.dev/design-code.html>
- Seong, S. (2024, April 12). *Data parallelism in machine learning training*. Medium. <https://medium.com/cloudvillains/data-parallelism-in-machine-learning-training-686ed9ab05fb>
- Yang, Y., Wang, R., Liu, X., Krishnan, A., Tao, Y., Deng, Y., Yao, K., Sun, P., Johnson, H., sinha, A., Golac, D., Friedland, G., Shakeel, U., Cooke, D., Sullivan, J., & Kong, C. (2025). Declarative data pipeline for large scale ml services. <https://arxiv.org/abs/2508.15105>
- Zaharia, M., Chen, A., Davidson, A., Ghodsi, A., Hong, S. A., Konwinski, A., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Xie, F., & Zumar, C. (2018). Accelerating the machine learning lifecycle with mlflow. *IEEE Data Engineering Bulletin*, 41(4), 39–45. https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf

Appendices

Appendix A

Titanic Project Example

To demonstrate and visualize the MLOps platform, we have created a simple frontend UI. The project used for demonstration is the popular Titanic dataset. We parallelize the training of two different models: a linear regression model and a neural network model. At the end, we compare the performance of the two models.

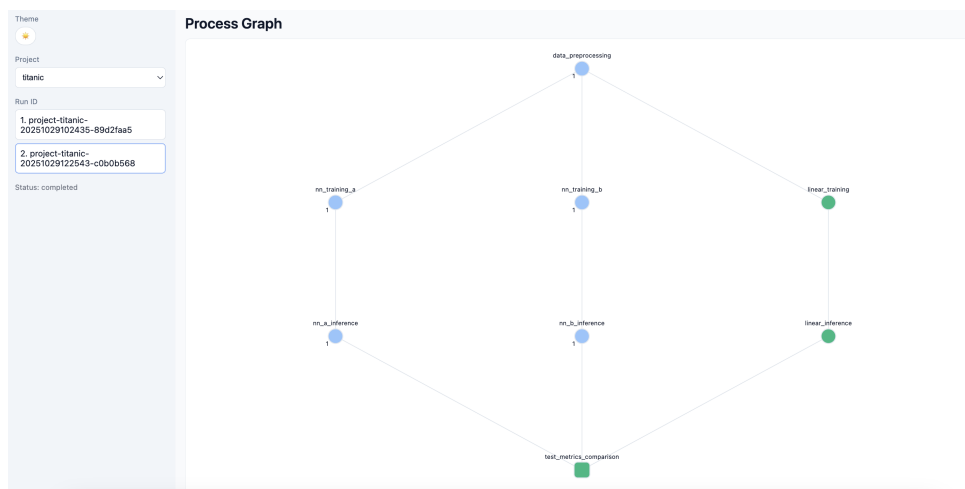


Figure A.1: Process Graph of Titanic project

The process graph is shown in Figure A.1. The color of the process nodes represents the current state of the process: green for completed, yellow for running, red for failed, blue for cached and grey for pending. Additionally, square nodes represent special static chart processes.



Figure A.2: Process Graph of an ongoing run

This graph enables users to see the current state of the run and the progress of the different processes running in parallel.

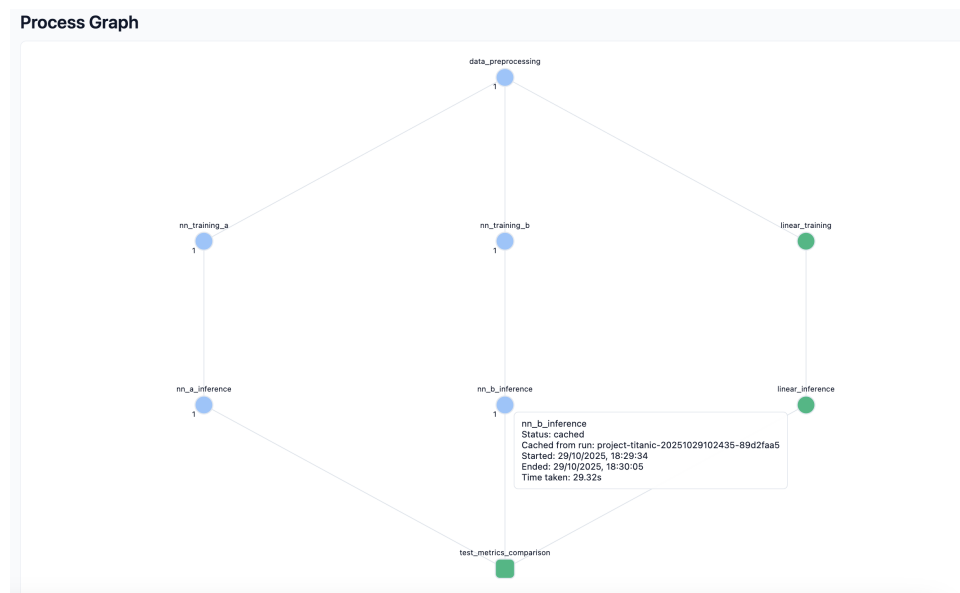


Figure A.3: Interactive process graph view showing detailed status information

As seen above in Figure A.3, when hovering over a process, users can see the information about the process such as the process name and execution time. Additionally for cached processes, the number at the bottom left shows the run number of the run that the process was cached from (refer to the left side to match the number to the run-id).

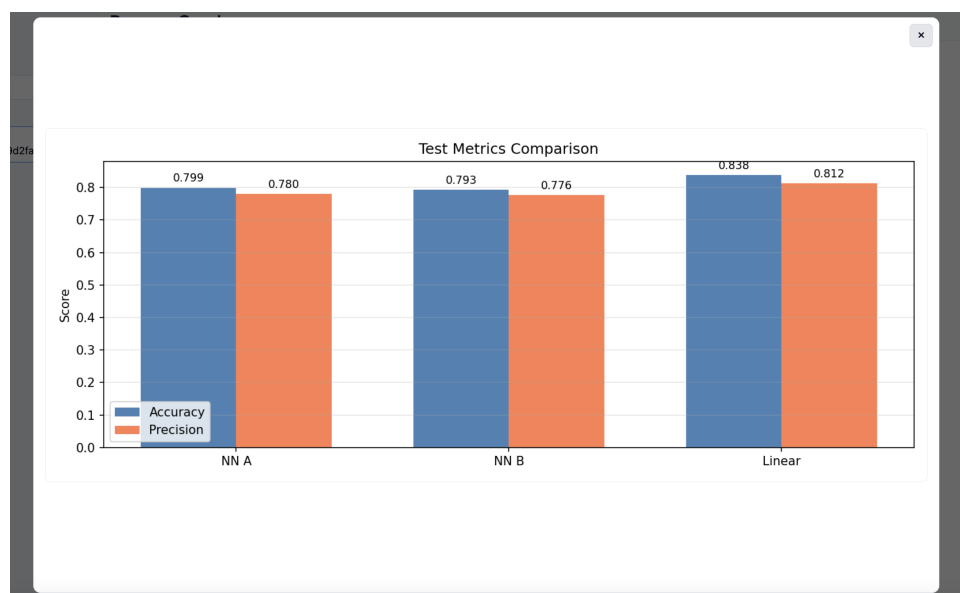


Figure A.4: Static Chart Node Modal

When clicking on a static chart node, a full-screen modal (A.4) will appear showing the chart.



Figure A.5: Dynamic Chart

The UI also shows dynamic charts (A.5) which are defined by the user and rendered on the front-end, allowing for real-time updates of certain metrics.

Appendix B

Example Code for Process and Step Decorator

```
@step()
def train_and_evaluate_nn(prepare_data, hyperparameters):
    clf = MLPClassifier(
        hidden_layer_sizes=hidden_layers,
        learning_rate_init=learning_rate,
        ...)
    """ Training Code """
    return {
        'model': clf
    }

@process()
def define_nn_training_process(data, hyperparameters):
    """NN Training process"""
    prep = data.get('data_preprocessing', {})
    result = train_and_evaluate_nn(prepare_data=prep,
        ↪ hyperparameters=hyperparameters)
    result['X_test'] = prep.get('X_test')
    result['y_test'] = prep.get('y_test')
    return result
```

Appendix C

Code for Static and Dynamic Charts

Project Config Reporting Section

```
reporting:
  enabled: true
  static_entrypoint: "projects/titanic/charts/plot_metrics.py"
  dynamic_entrypoint: "projects/titanic/charts/plot_metrics.js"
  charts:
    - name: "nn_losses"
      type: dynamic
      probe_paths:
        nn_a: "nn_training_a/train_and_evaluate_nn"
        nn_b: "nn_training_b/train_and_evaluate_nn"

    - name: "test_metrics_comparison"
      probe_paths:
        nn_a: "nn_a_inference/test_inference"
        nn_b: "nn_b_inference/test_inference"
        linear: "linear_inference/test_inference"
```

Static Charts Code

```
@chart()
def test_metrics_comparison(metrics, ctx):
    acc_series = metrics['linear'].get('test_accuracy', {})
    # chart plotting logic here
    ctx.savefig('test_metrics_comparison.png', dpi=150)
```

Dynamic Charts Code

```
import { chart } from '/mlops-charts.js';
chart('training_curves', (probePaths, ctx, listener) => {
  listener.subscribeAll(probePaths, (all) => {
    const series = ctx.toSeries((all.loss).train_loss);
```

```
        // render series with preferred charting library
    });
});
```

Appendix D

Key-Value Database Design

D.1 Overview

This appendix documents the key-value (KV) subsystem used for pipeline caching, run lifecycle, metrics, events, and charts indexing. Two backends are supported: (1) Redis for dev/test and (2) Google Firestore for durable production indexing with Pub/Sub events. Large artifacts (e.g., pickled outputs, charts) live in object storage; the KV holds only metadata and deterministic references.

D.2 Logical Model

- **StepCacheIndex**: exact-hash index for a single step execution.
- **ProcessCacheIndex**: exact-hash index for a process.
- **Run**: status, timestamps, stats, last_updated.
- **RunSteps**: per-run step records under the run document.
- **ChartsIndex (per-run)**: compact list of chart artifacts for UI.
- **Probe Metrics (per-run, per-path)**: metrics keyed by *probe_path*.

D.3 Firestore Layout

Root namespace: `mlops_projects/{project_id}`

- `step_indices/{process}:{step}:{ih}:{ch}:{fh}` → step cache record
- `process_indices/{process}:{ih}:{ch}:{fh}` → process cache record
- `runs/{run_id}` (doc) fields: `status`, `timestamps{start,end}`, `last_updated`
 - `steps/{process}.{step}` → per-run step record
 - `charts_index/index` (doc) → compact charts map (per-run)
- `metric/{run_id}/probes_by_path/{encoded}` → metrics for a single *probe_path*

Appendix E

Projects vs Runs

E.1 Overview

This appendix clarifies the distinction between **projects** (long-lived workspaces) and **runs** (single executions). Projects define configuration, code, and artifacts. Each invocation creates a new run with a unique `run_id` (e.g., `project-premier-league-20251021115816-cbdef7f3`).

E.2 Projects

What a project is

- A named, isolated workspace at `projects/<project_id>/` with `configs/`, `artifacts/`, `logs/`, `data/`, and `project_info.json`.
- The required config is `projects/<id>/configs/project_config.yaml`.
- Optionally, `configs/cluster_config.yaml` enables cluster execution.

Common actions (CLI)

```
1 # Create a project
2 python -m mlops.main create <project-id> [--description "text"]
   [--config path/to/base.yaml]
3
4 # List projects
5 python -m mlops.main list
```

E.3 Runs

What a run is

- A single pipeline execution for a project; identified by `run_id` generated as `project-{project_id}-{timestamp}-{suffix}`.
- Start a run with:

```
1 python -m mlops.main run <project-id> [--local]
```


Relationship to projects

- Each run belongs to exactly one project; its `run_id` is recorded in `project_info.json` with a `config_hash`.