Tampere University

Nico Renaldo

Cost-Efficient MLOps Architecture for Scientific Machine Learning

A Practical Design for Diffusion-Based GNSS Data Assimilation on Self-Managed Infrastructure

Master's thesis
Faculty of Information Technology and Communication Sciences
December 2025

Abstract

Nico Renaldo

Cost-Efficient MLOps Architecture for Scientific Machine Learning: A Practical Design for Diffusion-Based GNSS Data Assimilation on Self-Managed Infrastructure

Master's thesis

Tampere University

Master's Programme in Signal Processing and Machine Learning

December 2025

High-resolution weather forecasting increasingly relies on machine learning models, including diffusion-based methods, for tasks such as data assimilation. Training these models requires substantial computational power, often pushing research teams toward expensive managed MLOps platforms or complex self-hosted solutions. This thesis addresses the challenge of building a scalable, reproducible, and cost-efficient MLOps pipeline for scientific machine learning on self-managed GPU infrastructure.

Conducted in collaboration with Skyfora and leveraging Scaleway's GPU resources, this research presents a hybrid MLOps architecture that integrates ClearML for experiment orchestration and tracking with a custom Go-based resource manager for dynamic GPU provisioning. The system occupies a middle ground between enterprise-level orchestration tools architecture and manually managed research infrastructure, automating the full training job lifecycle while maintaining simplicity and cost efficiency.

Evaluation results demonstrate a 76.9% reduction in operational costs compared to static provisioning through dynamic resource management. The system consolidates the developer interface into three tools (Git, Python, and ClearML), reducing typical training workflows from 20–30 minutes to 4 minutes from code changes to result retrieval. Using predominantly off-the-shelf components (83%) and only 17% of custom development (approximately 2,548 lines of Go code), the architecture achieves MLOps Maturity Level 2 (Automated Training) with a foundation for Level 3 capabilities. These findings demonstrate that resource-constrained research teams can build robust, reproducible MLOps pipelines without the complexity of enterprise orchestration tools or the costs of managed platforms, ultimately enabling researchers to focus on scientific discovery rather than infrastructure management.

Keywords: MLOps, Scientific Machine Learning, Diffusion Models, Cloud Computing, Resource Management, ClearML, Scaleway, Reproducibility

The originality of this thesis has been checked using the Turnitin Originality service.

PREFACE

This thesis was conducted within the Faculty of Information Technology and Communication Sciences at Tampere University, in collaboration with Skyfora Oy.

I want to express my deepest gratitude to my supervisor, Professor David Hästbacka, for his expert guidance, insightful feedback, and continuous support throughout this research. I am also sincerely thankful to Sergio Moreschini for his valuable contribution in formulating and refining the research question, and to Alex Matakos from Skyfora for his significant support in shaping the topic and for providing the opportunity to carry out this work in partnership with the company.

I am endlessly grateful to my partner, Valerie. Her steady support, patience, and optimism carried me through every high and low of this journey. Her constant encouragement and belief in my work have been indispensable to the completion of this thesis.

Tampere, 16th December 2025

Nico Renaldo

Contents

1 Introduction 1

2 Background and Related Works 3

2.1 Machine Learning 3

2.2 Machine Learning Operations (MLOps) 4

2.3 MLOps Maturity Model 6

2.4 MLOps Architecture Comparison 9

3 Methodology 15

3.1 Design Science Research (DSR) 15

3.2 Problem Identification 17

3.3 Objectives 18

3.4 Evaluation 19

4 Design and Development 23

4.1 Technology Stack 23

4.2 Architectural Overview 26

4.3 Training Orchestration 31

4.4 Compute Resource Management 35

4.5 Data Processing 41

4.6 Experiment Tracking 45

4.7 Performance Monitoring 49

5 Evaluation & Results 54

5.1 Easy Adoption (O1) 54

5.2 Cost Effective (O2) 57

5.3 Extendable (O3) 58

5.4 Ease of Use (O4) 60

6 Discussion 62

6.1 Interpretation of Results 62

6.2 Comparison with State of the Art 63
6.3 Practical Implications 63
6.4 Limitations 64
6.5 Future Work 65
7 Conclusion 66

List of Figures

2.1 MLOps lifecycle [30] 5
2.2 Implementation of principles within technical components [17] 8
2.3 Smartflow MLOps architecture for geospatial research [20] . . 11
2.4 H&M MLOps architecture [9] 12
2.5 High-level diagram of a research-oriented MLOps architecture utilizing ClearML for experiment tracking, dataset management, and automated workflow orchestration [26] 13

4.1 ClearML's worker-queue architecture showing agents polling task queues and executing on distributed compute resources 24
4.2 Example ClearML pipeline showing workflow orchestration with multiple steps executed on distributed resources 25
4.3 System architecture diagram illustrating the integration of the core MLOps components and the data flow. 28
4.4 Sequence diagram showing the automated process from code commit to task submission and queue placement in ClearML. 31
4.5 Simplified flowchart showing the continuous agent lifecycle. . 33
4.6 Sequence diagram showing the detailed task execution flow when executing tasks. 34
4.7 ClearML configuration tab showing the hyperparameter used in the training run. 46
4.8 ClearML monitoring tools showing time-series plots of GPU utilization, GPU memory usage, CPU utilization, system memory consumption, disk I/O, and training loss for a typical experiment. 49
4.9 ClearML worker monitoring view showing the list of active agents, current tasks being processed, and live GPU/CPU/memory utilization per agent. 52
4.10 ClearML project dashboard page showing project-wide experiment statistics and summary analytics. 53

LIST OF TABLES

2.1 MLOps core principles 6
2.2 MLOps technical components 7
2.3 Microsoft MLOps Maturity Model levels and components . . . 10

3.1 Research objectives and their motivation 19
3.2 Functional and non-functional requirements of the architecture 20
3.3 Evaluation properties and episodes used to assess the reference and concrete architectures 22

4.1 Technology stack and MLOps component mapping 27
4.2 Implemented MLOps technical components 28
4.3 Mapping of requirements to MLOps technical components . . 30

5.1 Tool inventory and learning requirements mapped to MLOps components 55
5.2 MLOps component implementation breakdown 56
5.3 GPU resource utilization over one-month evaluation period . . 57
5.4 Task completion time for standard training workflow 61

6.1 Summary of key findings and implications 62

List of Programs and Algorithms

4.1 Task submission command 32
4.2 Resource manager provisioning logic (pseudocode) 36
4.3 Resource manager deprovisioning logic (pseudocode) 39
4.4 Retrieving a trained model artifact 48
4.5 Logging training metrics in ClearML 50

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

☒ Yes
☐ No

I hereby declare that the AI-based applications used in generating this work are as follows:

Application	**Version**
ChatGPT	GPT-5
Grammarly	

PURPOSE OF THE USE OF AI

ChatGPT was used to support the thesis structure, and Grammarly was used in the final stages to refine grammar and clarity.

PARTS OF THIS WORK, WHERE AI WAS USED

The mentioned tools were used throughout the thesis to refine the language and improve grammatical clarity.

ACKNOWLEDGEMENT OF RISKS

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

1 Introduction

Accurate prediction of weather and climate is vital for informed decision-making and for mitigating the risks posed by extreme events. Before the 20th century, forecasting relied on sparse, irregular observations and simple extrapolation methods [4]. Over the past century, numerical weather prediction (NWP) has transformed this process by combining physical laws with high-performance computing and assimilating diverse data from radars and satellites to deliver high-resolution forecasts [5].

Despite these advances, NWP continues to face persistent challenges. The immense computational cost of high-resolution models limits the spatial and temporal detail achievable in operational forecasts, often requiring trade-offs between the forecast horizon and the accuracy [6]. As a result, cutting-edge forecasting capabilities remain confined to a few government agencies and research institutions with access to supercomputer infrastructure, leaving smaller organizations unable to compete without substantial capital investment.

More recently, advances in AI have enabled the integration of machine learning (ML) into many stages of the weather forecasting pipeline, from data preprocessing to post-processing of model output [7]. In particular, ML models are increasingly explored for physics-adjacent tasks such as data assimilation, the process of optimally combining observations with model predictions to estimate the most probable state of the atmosphere at a given time. A promising data source in this context is the Global Navigation Satellite System (GNSS). Although originally developed for navigation, GNSS can be used to obtain dense, continuous streams of atmospheric observations under all weather conditions.

This thesis contributes to an ongoing project with Skyfora that explores the use of AI-driven models for GNSS data assimilation. Skyfora is a Finnish startup that specializes in high-quality GNSS data for weather intelligence. Its core values lies in a dense GNSS network that provides unique insights into atmospheric conditions and how the weather patterns develop.

Although AI models have extremely significant potential to improve forecasts [18], [19], [22], [32], [35], their development and deployment pose considerable engineering hurdles. Training, retraining, and safely operating diffusion models at scale require both computational power and robust operational practices. Machine Learning Operations (MLOps) has emerged as a discipline to address such challenges by providing practices and tooling to manage the end-to-end lifecycle of ML systems. However, industrial and scientific teams face a critical gap: while managed platforms such as Azure ML, AWS SageMaker, and Google Vertex AI offer integrated solutions, they are costly, introduce vendor lock-in, and are not always accessible to smaller organizations or research-driven projects. In contrast, self-managed infrastructure promises flexibility and cost control but introduces steep operational complexity. This trade-off is particularly acute for scientific workloads, which demand not only raw performance but also transparency,

reproducibility, and disciplined cost management.

This thesis is motivated by the practical constraints of an industrial collaboration. The infrastructure is provided through a partnership with Scaleway[1], which offers GPU machines accessible only via APIs and command-line interfaces, without a managed ML platform. These constraints exclude many of the standard MLOps solutions in the literature and necessitate a design tailored for the industrial requirements.

Against these constraints, the central objective of this thesis is to design and implement an end-to-end MLOps system that enables AI-based GNSS data assimilation to be feasible, reproducible, and cost-efficient on self-managed GPU infrastructure. Scientific machine learning workloads of this nature require three key properties: scalability, reproducibility, and cost efficiency. The aim is to provide a reference implementation that demonstrates how an open-source pipeline can be built under industrial constraints, balancing cost, performance, and operational reliability. The contribution of this work is twofold: first, it presents a practical architecture for self-managed scientific MLOps; and second, it evaluates the system's effectiveness in meeting these objectives. The results are intended to guide researchers and engineers facing similar challenges in deploying advanced ML models to production.

To guide the study, the following research question is addressed:

RQ1: How can we design and implement a self-managed MLOps pipeline that supports scalable, reproducible, and cost-efficient training of diffusion models for scientific ML use cases using open source tooling and self-managed GPU machines?

The remainder of this thesis is organized as follows.

- Chapter 2 Background: Introduces core background theory and reviews the key concepts of the component of the system.
- Chapter 3 Methodology: Outlines the design science research (DSR) approach taken and the methods used for system evaluation.
- Chapter 4 Design & Development: Presents the architecture and design of the MLOps system.
- Chapter 5 Evaluation & Results: Evaluates the implemented system with respect to the stated objectives.
- Chapter 6 Discussions: Discusses the results, challenges, and lessons learned.
- Chapter 7 Conclusion: Summarizes the thesis findings and suggestions for future work.

[1] https://www.scaleway.com

2 Background and Related Works

2.1 Machine Learning

Machine learning is a field of artificial intelligence where computers learn patterns from data to make predictions or decisions without being explicitly programmed [28]. In recent years, deep learning, a specific ML architecture based on multilayer neural networks, has driven immense progress across domains such as computer vision, natural language processing, and robotics [14].

These advances have been fueled by increases in available data and computational power, enabling the training of larger models on massive datasets. From roughly 2015 to 2020, progress in ML was largely driven by architectural innovation—convolutional networks, sequence models, attention mechanisms, and eventually Transformers [12]. Between 2020 and 2025, the field shifted toward a scaling paradigm in which model size, dataset size, and compute budgets grew dramatically, pushed by empirical scaling laws that suggested predictable improvements with increased scale [3]. Modern ML models often have millions or even billions of parameters and require significant computational resources to train. Now, as scaling has been pushed close to its practical and economic limits, the field faces a new bottleneck where simply increasing model and data size no longer yields the same returns [34]. As a result, the research community is increasingly exploring alternative directions, such as more efficient architectures, improved optimization methods, and approaches that reduce dependence on brute-force compute scaling.

To handle computational demands, practitioners leverage specialized hardware (GPUs, TPUs) and distributed computing techniques. Distributing the training workload across multiple machines can dramatically accelerate learning and enable scaling for larger models and datasets that would not fit in a single computer node [10]. For example, OpenAI's GPT-3 language model with 175 billion parameters was estimated to require 36 years to train on a single high-end GPU, whereas training on a cluster of 512 GPUs reduces the time to under a year [21]. Such comparisons highlight the effectiveness of scaling out and are often the only viable path to training cutting-edge models in a reasonable time.

Building and managing these large-scale ML workflows introduces new engineering challenges. The process of developing an ML model is not only about refining algorithms. It also involves managing complex data processing pipelines, distributed training jobs, model evaluation, and deployment in production environments. This recognition has led to the emergence of Machine Learning Operations (MLOps) as a set of best practices and tools for systematically managing the ML life cycle beyond model training. In the following sections, we review the MLOps paradigm, approaches to distributed ML training, and techniques for scaling hyperparameter optimization.

2.2 Machine Learning Operations (MLOps)

Machine Learning Operations (MLOps) is a set of practices, principles, and tools for deploying and maintaining ML models that are reliable, efficient, and reproducible [17]. The term MLOps, derived from the combination of "Machine Learning" and "Operations", refers to applying operational best practices to the ML lifecycle. While DevOps focuses on continuous integration and delivery of code, MLOps extends these principles to include continuous training (CT), the automatic retraining of models as new data becomes available, making CT a distinguishing characteristic unique to MLOps [30].

The motivation for MLOps stems from practical challenges observed across industry and research. Microsoft's large-scale study of ML engineering practices across multiple product teams identified three fundamental differences between ML and traditional software engineering. The data management complexity making versioning challenging, model customization that requires both software engineering and ML expertise, and entangled dependencies with non-monotonic error behavior [2]. These differences manifest as operational challenges, including: training-serving skew, concept drift, and difficulty reproducing experimental results. Without proper MLOps practices, the transition from model development to production can take many months or even an entire year [27], creating a significant gap between research outcomes and operational value.

Traditional software development focuses primarily on code artifacts and their deployment, with well-established practices for version control, testing, and continuous integration. MLOps extends this scope to encompass the entire ML lifecycle [17], [30]. The resulting artifacts include not only code but also trained models, snapshots of training data, feature transformations, and hyperparameter configurations. This expanded scope requires fundamentally different versioning strategies (tracking data lineage alongside code), testing approaches (validating data quality and model performance in addition to code correctness), and deployment patterns (managing model serving infrastructure and monitoring prediction quality) [2].

The ML lifecycle in an MLOps context typically progresses through several interconnected stages, forming an iterative workflow rather than a linear process [2], [17], [25]. Figure 2.1 illustrates the key stages and their relationships within the MLOps lifecycle. The lifecycle begins with **project requirement engineering**, where business problems are framed as ML tasks, success metrics are defined, and technology choices are made [25]. This is followed by **data management and preparation**, encompassing data collection from various sources, quality assessment, cleaning, labeling, and versioning [2], [25]. The preparation phase ensures data quality across multiple dimensions: intrinsic quality (accuracy, completeness), contextual quality (relevance, timeliness), representational quality (interpretability, format), and accessibility quality (availability, security) [25].

Once data is prepared, the workflow moves to **feature engineering and model development**, where domain knowledge is encoded into feature transformations and various algorithms are explored through experimentation [2], [17]. This phase involves iterative cycles of feature creation, model training, hyperparameter tuning,

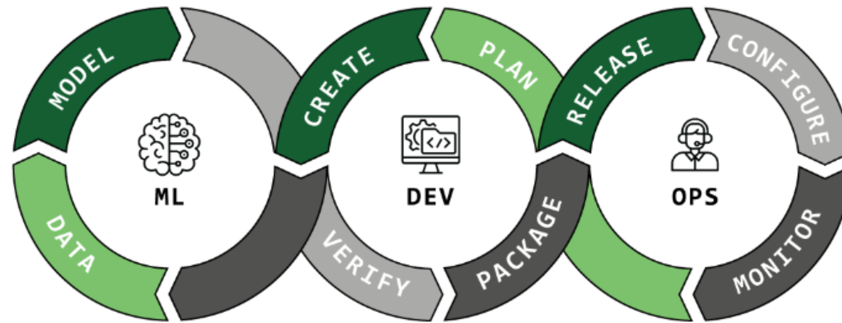

Figure 2.1: MLOps lifecycle [30]

and evaluation. The **model training** phase itself requires careful orchestration of computational resources, experiment tracking to record parameters and metrics, and validation procedures to assess model quality [17], [25]. Successful models then progress to the **deployment phase**, where they are packaged, integrated into production systems, and exposed through serving infrastructure such as REST APIs for online inference or batch processing pipelines [27], [29].

The lifecycle does not end at deployment. **Continuous monitoring and feedback** form a critical component, tracking model performance in production, detecting data drift and concept drift, and triggering retraining when performance degrades [2], [17], [30]. Monitoring extends beyond model accuracy to encompass data quality invariants, prediction latency, resource consumption, and fairness metrics [30], [31]. Feedback loops connect monitoring insights back to earlier stages, creating a continuous improvement cycle where production observations inform data collection priorities, feature engineering decisions, and model refinement [17].

This iterative lifecycle must be supported by appropriate infrastructure and practices. To provide a framework for understanding and implementing MLOps systems, Kreuzberger et al. [17] conducted a systematic mixed-methods study combining a literature review (27 articles), tool analysis (11 tools), and expert interviews (8 experts). Their research identified nine core principles that guide MLOps implementation and nine technical components that provide the concrete infrastructure to realize these principles. This dual-layer framework offers clarity for both understanding existing systems and designing new ones. Given its comprehensive scope and empirical foundation, this thesis adopts Kreuzberger's taxonomy as the primary reference framework for analyzing and developing MLOps solutions.

Table 2.1 summarizes the nine core principles that define what an MLOps system should achieve, while Table 2.2 presents the nine technical components that implement these principles. Figure 2.2 illustrates the mapping between principles and components. As indicated by the principal references in each component description (e.g., "P1, P6, P9" for the CI/CD Component), most components implement multiple principles simultaneously, reflecting the interconnected nature of MLOps practices. For instance, the Workflow Orchestration Component (C3) simultaneously enables workflow orchestration (P2), reproducibility (P3), and continuous ML training (P6) by coordinating pipeline execution while maintaining ex-

ecution records. Similarly, the Model Registry (C6) supports both reproducibility (P3) and versioning (P4) by storing trained models with their complete metadata and lineage information [17].

Table 2.1: MLOps core principles

ID	Principle	Description
P1	CI/CD automation	Provides continuous integration, delivery, and deployment with automated build, test, and deployment steps that give developers fast feedback, thereby increasing overall productivity.
P2	Workflow orchestration	Coordinates the tasks of an ML workflow pipeline according to the execution order while considering relationships and dependencies.
P3	Reproducibility	The ability to reproduce an ML experiment and obtain the same results.
P4	Versioning	Ensures the versioning of data, models, and code for reproducibility and traceability.
P5	Collaboration	Ensures collaborative work on data, models, and code, emphasizing a communicative work culture that reduces domain silos across roles.
P6	Continuous ML training and evaluation	Enables continuous training of ML models based on new feature data through monitoring, feedback loops, and automated pipelines, always including evaluation runs to assess model quality changes.
P7	ML metadata tracking and logging	Tracks and logs metadata for each orchestrated ML workflow task, including model-specific metadata (parameters, performance metrics) and model lineage (data and code used) to ensure full traceability.
P8	Continuous monitoring	Continuously assesses data, model, code, infrastructure resources, and model serving performance to detect potential errors or changes that influence product quality.
P9	Feedback loops	Integrates insights from quality assessment steps into the development or engineering process through multiple feedback mechanisms.

2.3 MLOps Maturity Model

While the principles and components described in the previous section define what constitutes an MLOps system, organizations implement these elements at varying levels of sophistication. MLOps maturity models provide frameworks for

Table 2.2: MLOps technical components

ID	Component	Description
C1	CI/CD Component (P1, P6, P9)	Ensures continuous integration, delivery, and deployment with fast feedback to developers, increasing overall productivity (e.g., Jenkins, GitHub Actions).
C2	Source Code Repository (P4, P5)	Ensures code storing and versioning, allowing multiple developers to commit and merge their code (e.g., GitHub, GitLab).
C3	Workflow Orchestration Component (P2, P3, P6)	Offers task orchestration of ML workflows through rule actions (e.g., Apache Airflow).
C4	Feature Store System (P3, P4)	Ensures central storage of commonly used features through offline stores for experimentation and online stores for low-latency production predictions (e.g., AWS Feature Store).
C5	Model Training Infrastructure (P6)	Provides foundational computation resources (CPUs, RAM, GPUs) in distributed or non-distributed configurations.
C6	Model Registry (P3, P4)	Stores trained ML models and their metadata, including both ML artifacts and metadata (e.g., MLflow).
C7	ML Metadata Stores (P4, P7)	Tracks metadata for each orchestrated ML workflow task, including training job details (date, duration, parameters, performance metrics) and model lineage (e.g., ClearML[1]).
C8	Model Serving Component (P1)	Provides model serving capabilities for online or batch inference via REST API (e.g., Kubeflow KServing, AWS SageMaker Endpoints).
C9	Monitoring Component (P8, P9)	Continuously monitors model serving performance, ML infrastructure, CI/CD, and orchestration (e.g., Prometheus with Grafana, TensorBoard, MLflow).

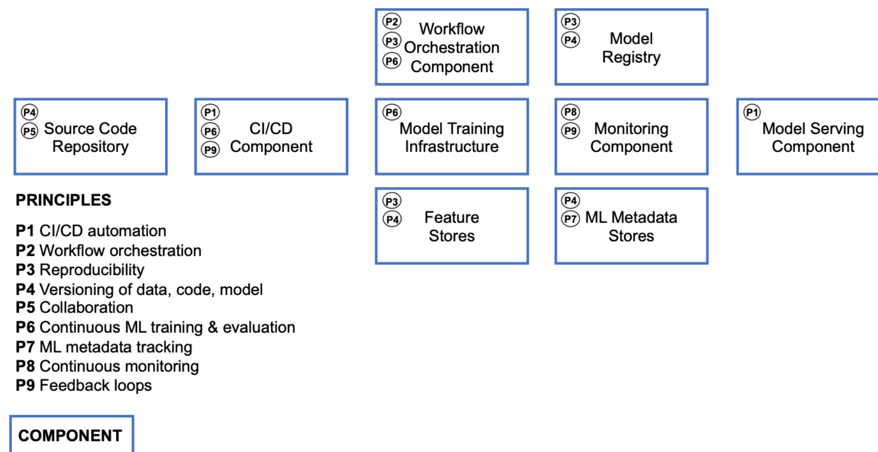

Figure 2.2: Implementation of principles within technical components [17]

assessing current capabilities and planning incremental improvements [15]. Multiple organizations have proposed maturity frameworks, with Google's three-level model [16] distinguishing between manual processes (Level 0), ML pipeline automation (Level 1), and CI/CD pipeline automation (Level 2). Microsoft's more granular five-level model [11] extends this progression from no MLOps through DevOps adoption, automated training, automated deployment, to full automation. This thesis adopts Microsoft's framework for its comprehensive representation of the progression from research to production systems.

The Microsoft MLOps Maturity Model defines five levels (0 through 4), with each building upon the previous by introducing additional automation and operational capabilities. The progression represents an evolutionary path rather than strict requirements. Organizations advance through levels incrementally as their needs and capabilities develop. Understanding these maturity levels helps contextualize design decisions and set appropriate expectations for system capabilities. Table 2.3 presents a comprehensive overview of these maturity levels and their constituent components.

Level 0 (No MLOps) represents the starting point where ML development occurs in an ad-hoc manner. Data scientists primarily work in notebooks, using manual processes for training and deployment. Model training happens on individual workstations or through manual SSH connections to GPU servers. Code often remains uncommitted, experiments lack systematic tracking, and reproducing results requires significant effort. This level characterizes early exploratory work where the focus is on model development rather than operational concerns.

Level 1 (DevOps, No MLOps) introduces software engineering best practices, including code version control and basic CI/CD pipelines for application deployment. The lack of ML-specific tooling makes it challenging to trace experimental results or reproduce model outputs. The system on this level does not have any systematic tracking of hyperparameters, training data versions, or model artifacts. Because there is a disconnect between software engineering and ML experimentation, this level of maturity is prone to a "research-to-production" gap,

where promising models struggle to reach production.

Level 2 (Automated Training) is the first step into a proper MLOps system, introducing automated training pipelines with managed compute resources and centralized experiment tracking. Version control extends to training code, models, and scoring scripts. In this phase, deployment remains manual, with engineering teams packaging and releasing models separately from the training process. This level significantly reduces manual overhead in the experimentation phase and establishes reproducibility, making it particularly suitable for research environments and teams actively iterating on model development.

Level 3 (Automated Model Deployment) extends automation to the deployment phase through CI/CD pipelines. Automated data pipelines validate and preprocess incoming data, ensuring consistency between training and serving environments. Model deployment becomes a programmatic operation rather than a manual handoff, reducing the reliance on data scientist expertise for production operations. This level represents a production-ready system capable of serving customers reliably, with the infrastructure to support continuous improvement through systematic model updates.

Level 4 (Full MLOps Automation) represents the most advanced state where systems become largely self-managing. Model retraining is triggered automatically based on production metrics, such as prediction quality degradation or data drift detection. Continuous monitoring spans the entire system—data quality, model performance, infrastructure health, and business metrics—with automated responses to detected issues. The system can adapt to changing conditions with minimal human intervention, automatically retraining and deploying improved models when performance degrades. Few organizations currently operate at this level, as it requires substantial investment in monitoring infrastructure, automated decision-making systems, and robust safeguards to prevent automated errors from propagating.

The appropriate maturity level depends on the system's purpose and operational context. Research environments and proof-of-concept projects often find Level 2 sufficient, as it provides reproducibility and systematic experimentation without the operational complexity of automated deployment. Production systems serving critical applications typically require Level 3 capabilities to ensure reliability, while Level 4 becomes valuable for systems operating at scale with rapidly changing data distributions or stringent latency requirements for model updates.

2.4 MLOps Architecture Comparison

While the MLOps principles and technical components provide a conceptual foundation, their real-world implementation varies significantly across platforms and architectural designs. The choice of architecture is not merely a technical decision but reflects fundamental trade-offs between scalability, operational complexity, development velocity, and resource constraints [27]. Examining both academic and industry MLOps architectures for research and production environments reveals how each solution is shaped by the unique requirements and pri-

Table 2.3: Microsoft MLOps Maturity Model levels and components

Level	Capabilities and Components
Level 0: No MLOps	**Manual process with limited automation** • Manual builds and deployments • Manual testing of model and application • No centralized tracking of model performance • Training of model is manual
Level 1: DevOps No MLOps	**DevOps practices applied without MLOps automation** • Automated builds • Automated tests for application code • Difficult to trace/reproduce results
Level 2: Automated Training	**Automated training and centralized tracking** • Automated model training with managed compute • Centralized experiment tracking and model versioning • Version-controlled training code, models, and scoring scripts • Manual model deployment managed by the engineering team
Level 3: Automated Model Deployment	**Automated deployment with continuous integration** • Automated data pipelines with quality validation • Automated model deployment via CI/CD pipeline • Comprehensive unit and integration tests for releases • Reduced reliance on data scientist expertise for deployment
Level 4: Full MLOps Automation	**Full automation with intelligent optimization** • Retraining triggered automatically based on production metrics • Continuous monitoring and self-healing capabilities • Automated performance optimization and drift detection

orities of its originating organization. Understanding these variations and their underlying constraints is crucial for making informed design decisions when constructing an MLOps system. The architectures reviewed here span the spectrum of complexity, scale, and organizational maturity, from large-scale cloud-native platforms to research-oriented lightweight implementations [20].

Smartflow is an MLOps architecture tailored for large-scale geospatial research, focusing on processing vast satellite imagery datasets and training computer vision models for tasks like land use classification and environmental monitoring [20]. As shown in Figure 2.3, the architecture shares several similarities with the system presented in this thesis, notably the use of ClearML [8] for experiment tracking through its UI and API, complemented by TensorBoard for training metrics and Grafana for cost and performance monitoring. The Smartflow architecture is built on a fully cloud-native foundation, leveraging object

storage for petabyte-scale geospatial datasets, managed Kubernetes clusters for distributed training and ETL processing of continental-scale satellite data, and specialized geospatial processing frameworks that operate within the cloud provider's ecosystem. This design prioritizes scalability and integration with existing geospatial data platforms, enabling researchers to efficiently process continental-scale datasets.

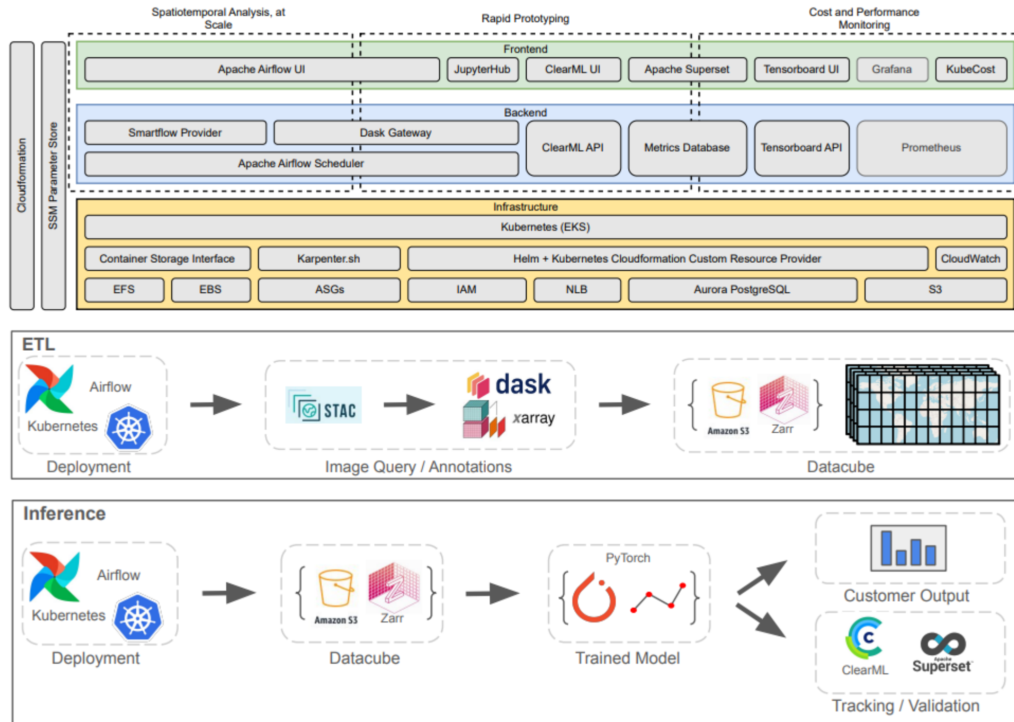

Figure 2.3: Smartflow MLOps architecture for geospatial research [20]

While the Smartflow architecture demonstrates effective use of MLOps tools for large-scale research, its infrastructure approach presents significant complexity that exceeds the requirements of this thesis. Kubernetes is designed for continuous, large-scale data processing workloads such as Smartflow's ETL pipelines for continental satellite imagery, where the operational overhead is justified by the scale and continuity of compute demands. In contrast, this thesis addresses intermittent training jobs with significantly lighter workload demands and minimal infrastructure maintenance requirements. The operational overhead of managing Kubernetes—including cluster configuration, service orchestration, networking policies, and ongoing maintenance—would consume disproportionate effort relative to the actual compute workload. This mismatch between infrastructure complexity and workload characteristics conflicts with the objective of operational simplicity (O3), making Kubernetes-based solutions inappropriate for the small research team context of this thesis.

H&M. In December 2020, H&M initiated development of a centralized AI platform based on MLOps principles to support machine learning across their global retail operations [9]. As illustrated in Figure 2.4, the technology stack shares similarities with Smartflow while addressing different organizational requirements.

The architecture uses Seldon Core for model serving and deployment, providing a Kubernetes-native solution for production-scale model inference. Unlike Smartflow's domain-specific focus on geospatial data processing, H&M's architecture is designed as a general-purpose platform supporting diverse machine learning applications across the organization, from demand forecasting and inventory optimization to recommendation systems and customer analytics. The architecture uses MLflow for model management and experiment tracking, providing centralized governance and standardization across multiple teams and projects. This enterprise-scale platform reflects H&M's philosophy that "a model not in production has no value" [27], emphasizing the importance of deployment automation and operational reliability.

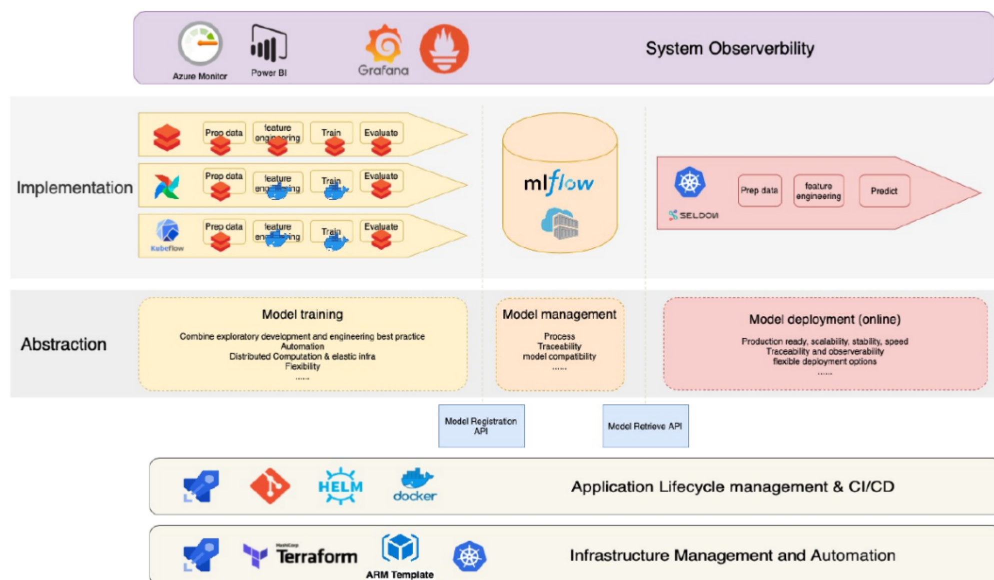

Figure 4.5: MLOps architecture of H&M [12].

Figure 2.4: H&M MLOps architecture [9]

Similar to Smartflow, H&M's architecture is built on Kubernetes, emphasizing robust and scalable infrastructure management to support enterprise-wide ML operations. While this approach is appropriate for H&M's scale—serving hundreds of data scientists and deploying numerous production models—it requires substantial organizational investment. The architecture necessitates dedicated platform engineering teams to manage Kubernetes clusters, service meshes, deployment pipelines, and cross-team governance. For organizations at H&M's scale, this investment is justified by the value of standardized, reliable ML operations across multiple business units. However, the same operational complexity presents significant barriers for smaller research teams with limited infrastructure resources, conflicting with the objective of operational simplicity (O3) that guides this thesis.

Agronomy Case Study Ruggeri et al. present a case study evaluating MLOps tools for machine learning research in agronomy, focusing specifically on the research phase rather than production deployment [26]. After analyzing several MLOps platforms, the team selected ClearML [8] for its comprehensive feature set, includ-

ing experiment tracking, dataset management and versioning, hyperparameter optimization agents, and pipeline orchestration. Figure 2.5 presents the high-level architecture of their research-oriented MLOps system. The study demonstrates how ClearML's integrated tooling provides an effective solution for research workflows, particularly highlighting its open-source nature, lower operational overhead compared to cloud-native platforms, and suitability for academic and research environments with constrained infrastructure resources.

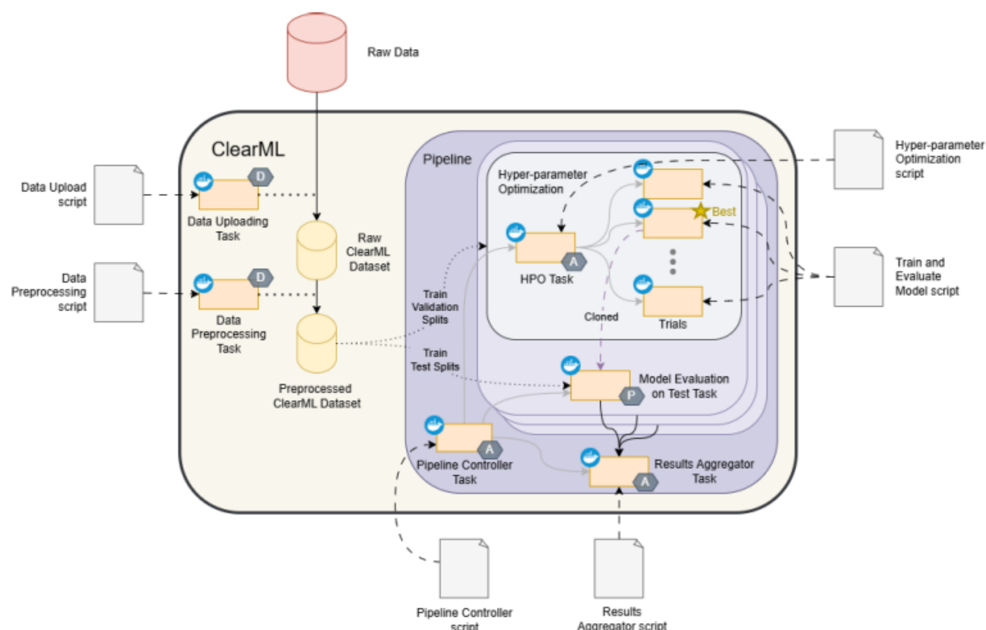

Figure 2.5: High-level diagram of a research-oriented MLOps architecture utilizing ClearML for experiment tracking, dataset management, and automated workflow orchestration [26]

The architectures reviewed above reveal a notable gap in the MLOps literature. There is limited guidance for small organizations and research teams that need to manage dynamic compute resources at a modest scale without adopting heavyweight orchestration platforms. The dominant paradigm in both academic literature and industry practice assumes either manual infrastructure management (as in the agronomy case study) or full adoption of Kubernetes-based orchestration (as in Smartflow and H&M). As a result, teams with infrequent training needs and relatively lightweight workloads do not fit well into either approach.

Kubernetes excels at managing complex, multi-service applications with sophisticated networking requirements, service discovery, and high-availability guarantees [24]. However, for teams running a few GPU instances for occasional training jobs, the operational overhead, like setting up clusters, managing security, and handling updates, still far outweighs the actual needs. Since the training jobs are independent and don't require complex features such as service meshes or load balancing, Kubernetes would be overkill.

Existing MLOps literature predominantly focuses on either enterprise-scale deployments where Kubernetes overhead is justified [27], or research environments

where infrastructure is assumed to be pre-provisioned and managed externally [26]. The specific challenge of lightweight, automated resource management for small-scale dynamic workloads remains largely unaddressed. This thesis targets precisely this gap: developing a resource management solution that provides automated provisioning and cost optimization for small research teams without requiring the operational expertise and maintenance burden associated with Kubernetes-based platforms.

3 Methodology

This chapter presents the research methodology used in this thesis, grounded in Design Science Research (DSR) principles. The methodology is structured to address the research question through systematic artifact development and evaluation, ensuring both scientific rigor and practical relevance for the MLOps pipeline design.

3.1 Design Science Research (DSR)

This research follows the Design Science Research paradigm as described by Hevner et al. [13] and Peffers et al. [23]. Design Science Research (DSR) represents a fundamental research paradigm that focuses on the creation and evaluation of artifacts to solve identified problems while advancing theoretical knowledge. DSR is particularly suitable for this study as it focuses on creating and evaluating artifacts that solve identified organizational issues while contributing to the theoretical knowledge base. The approach bridges the gap between theory and practice by producing innovative artifacts that address real-world MLOps challenges in scientific machine learning.

The theoretical foundations of DSR are based on the recognition that information systems research operates at the intersection of people, organizations, and technology. This paradigm addresses both the practical challenges of creating effective technological solutions and the scientific rigor required to advance theoretical understanding.

The research addresses the identified gap between vendor-locked managed ML platforms and complex self-managed infrastructure by designing and implementing a practical MLOps system. This artifact-oriented approach aligns with DSR's emphasis on creating solutions that demonstrate both utility and theoretical contribution.

The DSR process follows a systematic framework that ensures both rigor and relevance in artifact creation [23]. This framework consists of six interconnected activities that form an iterative cycle:

1. Problem Identification and Motivation: Establishing the research problem's importance and justifying the need for a solution
2. Define Objectives for a Solution: Specifying what a successful solution should accomplish
3. Design and Development: Create an artifact that addresses the identified problem
4. Demonstration: Showing that the artifact can solve instances of the problem
5. Evaluation: Measuring and analyzing the artifact's effectiveness

6. Communication: Disseminating knowledge about the problem, artifact, and findings

This framework is not strictly linear. The process may iterate between activities as the understanding deepens and artifacts evolve. The iterative nature allows for refinement based on evaluation results and changing requirements.

An artifact is the product of a design process [13], and DSR produces various types of artifacts that contribute to the knowledge base in distinct ways:

- Constructs form the conceptual vocabulary and symbols used to describe and reason about problems and solutions within a domain. They establish the language through which researchers and practitioners communicate about the problem space.
- Models represent abstractions and representations that capture key relationships between constructs, providing frameworks for understanding how different elements interact within a system. These models can be conceptual, mathematical, or diagrammatic, helping to simplify complex realities into manageable representations.
- Methods encompass algorithms, practices, and procedures that define how to perform specific tasks or solve particular problems. They provide actionable guidance for achieving desired outcomes and can range from formal algorithms to structured problem-solving approaches.
- Instantiations are working systems that operationalize constructs, models, and methods to demonstrate feasibility and effectiveness in real-world or simulated environments. Instantiations serve as proof-of-concept implementations that validate theoretical contributions through practical application.

Each type of artifact requires a different evaluation approach and makes distinct additions to the knowledge base. Constructs and models are typically evaluated through their explanatory power and utility in reasoning about problems, while methods are assessed based on their efficiency, effectiveness, and applicability. Instantiations are evaluated through empirical demonstration, measuring their performance against established requirements and objectives. The artifact type also influences the generalizability of research contributions, with more abstract artifacts like constructs and models offering broader applicability, while instantiations provide concrete evidence of practical viability in specific contexts.

This research produces an instantiation as its primary artifact type [13]. The working MLOps system developed in this study operationalizes the concepts and methods identified in the literature, demonstrating their practical feasibility and effectiveness in addressing the challenges of self-managed MLOps infrastructure. By creating a functioning system, this research provides empirical evidence of how the proposed solution performs in realistic conditions, validating both the technical approach and the design decisions through concrete implementation and evaluation. Secondary artifacts include architecture patterns, deployment procedures, and conceptual frameworks for self-managed MLOps.

3.2 Problem Identification

The primary motivation of this thesis stems from practical challenges faced by the industry partner in establishing an effective MLOps infrastructure. Currently, the organization operates with a fragmented setup: ClearML[1] serves as the artifact and experiment registry, while machine provisioning for training remains entirely manual. This configuration presents several critical problems that hinder the organization's machine learning operations.

Lack of Automated Resource Management. The most immediate challenge is the absence of automated infrastructure orchestration. Data scientists must manually provision GPU instances through a cloud provider (e.g., Amazon Web Services and Microsoft Azure Cloud). This manual process introduces significant overhead: each training job requires developers to allocate machines, configure environments, monitor execution, and deallocate resources upon completion. The time spent on infrastructure management directly reduces the time available for model development and experimentation, ultimately slowing the pace of innovation.

Infrastructure Constraints and Vendor Lock-in Concerns. The industry partner faces a unique constraint through its partnership with Scaleway[2], which provides cost-effective GPU access but offers only basic functionality via Command Line Interface (CLI) and Software Development Kit (SDK). This limitation rules out all-in-one solutions like Amazon SageMaker or Google Cloud AI Platform. While such managed platforms would simplify operations, the industry partner deliberately avoids them to maintain control over its infrastructure and prevent vendor lock-in. Given the relatively modest scale of their workloads, the overhead of setting up and learning those platform would be disproportionate compared to the benefit it brings.

Operational Inefficiency and Scalability Limitations. The current manual approach creates bottlenecks that will worsen as the organization's ML activities grow. Without automated resource management, the time required for infrastructure operations scales linearly with the number of experiments. This inefficiency affects not only individual productivity but also the organization's ability to iterate quickly on models, conduct comprehensive hyperparameter searches, and maintain reproducible experimentation practices. The lack of systematic resource tracking also makes it difficult to optimize costs and resource utilization.

Gap in MLOps Tooling for Resource-Constrained Organizations. This situation reveals a broader gap in the MLOps ecosystem: existing solutions predominantly target either large-scale enterprise deployments with complex orchestration needs, or simple proof-of-concept scenarios with minimal infrastructure requirements. Organizations operating at an intermediate scale—with substantial ML activities but limited computational workloads—lack appropriate tooling. These organizations need a solution that bridges basic infrastructure management with sophisticated ML pipeline capabilities while maintaining operational autonomy. The challenge is compounded when working with infrastructure

[1] https://clear.ml
[2] https://www.scaleway.com

providers that offer API-only access without managed ML services.

These interconnected problems establish clear requirements for any effective solution. The **lack of automated resource management** demands that any solution must minimize operational overhead and be intuitive enough for data scientists to adopt without extensive infrastructure training. The **infrastructure constraints and vendor lock-in concerns** require an architecture built on open-source components that can integrate with API-only cloud providers while remaining extensible as organizational needs evolve. The **operational inefficiency and scalability limitations** require automated resource orchestration that optimizes both cost and developer productivity. Finally, the **gap in MLOps tooling** highlights the need for a solution that achieves sophisticated ML pipeline capabilities without the complexity penalty of enterprise platforms. Section 2.4 examines how other organizations have addressed similar MLOps challenges, highlighting the trade-offs between operational complexity and scalability that inform the design of our proposed solution.

3.3 Objectives

The primary objective of this research is to design, implement, and evaluate a self-managed MLOps pipeline that enables scalable, reproducible, and cost-effective training and deployment of ML models for scientific applications. This objective directly addresses the research question by creating a practical solution that bridges the gap between expensive managed ML platforms and complex self-managed infrastructure.

The industry partner has limited experience building ML products systematically, using GitHub for version control, ClearML as artifact and experiment tracking, and Scaleway as a cloud provider for GPU machines. These technologies will serve as the base tools for the pipeline, and other tools needed will be provided via open-source tooling or custom-made scripts. In addition, the machine learning team itself is relatively small and is currently in a research and development phase. This means there are periods of low activity when resources, especially GPU machines, might remain idle. To optimize costs and avoid unnecessary spending, infrastructure usage must be carefully planned so that GPU resources are only active when needed, with automated shutdown and provisioning policies where possible. This approach will help ensure that the team remains agile, cost-effective, and ready to scale once development accelerates.

The problems identified in Section 3.2 directly inform the objectives of this research. Each objective addresses specific challenges while ensuring the solution remains practical for organizations with limited MLOps maturity. Based on these considerations, the objectives of the pipeline are presented in Table 3.1.

The aim of this project is to build an MLOps pipeline at maturity level 2, excluding the requirement for automated model training. More specific functional and non-functional requirements for the architecture that have been agreed upon with the industry partner are presented in Table 3.2. These requirements translate the high-level objectives into concrete technical specifications that guide the design and

Table 3.1: Research objectives and their motivation

Objective	Motivation and Importance
O1 Ease of Adoption	The manual provisioning process creates friction that hinders experimentation. The solution should have a minimum learning curve and integration complexity. This is critical given the small team size and limited capacity for extensive training. A solution requiring weeks of learning would ultimately fail to solve the practical challenges faced by the team.
O2 Cost-Effective	The current setup relies on always-on GPU instances, which leads to significant inefficiency due to long periods of resource idleness. This results in wasted infrastructure costs and underutilization of available resources. The solution should reduce operational costs without compromising performance or availability for actual compute demands.
O3 Extendable	Although the thesis targets level 2 ML maturity, the solution must be architecturally flexible to accommodate future growth in team size, model complexity, and increase in complexity without requiring fundamental redesign.
O4 Ease of Use	Addressing inefficiency in the developer workflow, the solution aims to reduce operational friction in daily ML tasks, which directly impacts work efficiency and iteration speed.

implementation phases.

3.4 Evaluation

Evaluation is a fundamental component of Design Science Research, serving to assess whether designed artifacts meet their intended objectives and contribute valuable knowledge to both theory and practice [33]. The Framework for Evaluation in Design Science (FEDS) by Venable et al. [33] provides systematic guidance for conducting rigorous evaluation activities in DSR projects. FEDS proposes four steps of evaluation:

1. Explaining the goals of the evaluation
2. Choosing the evaluation strategy
3. Determining the properties to evaluate
4. Designing the evaluation episodes

FEDS divides evaluation episodes into two dimensions: the functional purpose of the evaluation (formative or summative) and the evaluation paradigm (artificial or

Table 3.2: Functional and non-functional requirements of the architecture

Requirement	Description
R1 Dynamic Resource Management	Automatically provision and deprovision GPU instances based on workload demand, with support for idle resource detection and cost optimization.
R2 Reproducibility	Ensure experiments can be consistently reproduced through containerized environments, version control integration, and comprehensive configuration tracking.
R3 Experiment Tracking	Provide detailed logging of hyperparameters, metrics, artifacts, and model versions for all training runs with integration to existing ClearML infrastructure.
R5 Open-source Tooling	Use only open-source tools and frameworks, avoiding dependencies on managed ML platforms or proprietary services.
R6 Performance Monitoring	Monitor system performance metrics, including resource utilization, training throughput, and cost efficiency, with automated alerting capabilities.
R7 Scalability	Support scaling from single-instance to multi-instance GPU training with minimal configuration changes and linear performance scaling.

naturalistic). In formative evaluation, it is conducted during the development process to improve the outcomes of the process under evaluation. In contrast, summative evaluation takes place after development to assess the artifact's overall effectiveness. In artificial evaluation, the artifact is tested in a controlled environment, whereas in naturalistic evaluation, it is tested in real-world conditions. After the evaluation activity, the researcher should decide whether any further iterations are needed to refine the artifact. This leads to a cycle of design and development, followed by evaluation again. After reaching the acceptable level, the researcher can proceed to the communication phase of the DSR to the relevant audiences.

The evaluation methodology for this study follows the FEDS framework to evaluate the MLOps pipeline developed against the research objectives and requirements established in Section 3.3. The evaluation approach adopted for this research aligns with the DSR evaluation methods outlined in section 3.1, specifically utilizing the observational, experimental, and descriptive evaluation methods to ensure both rigor and relevance by systematically evaluating the artifact's effectiveness while contributing to theoretical knowledge.

3.4.1 Evaluation Goals

The evaluation goal is to assess both the reference architecture and the concrete implementation in fulfilling the functional and non-functional requirements specified in 3.2.

3.4.2 Evaluation Strategy

The evaluation approach follows a summative evaluation, in which assessment is conducted after the artifact's design and implementation to determine if it meets the intended objectives. Furthermore, the evaluation is done in a naturalistic environment, where the prototype is tested under realistic conditions that mirror its intended operational setting, rather than in a simplified or purely experimental context. The concrete architecture is evaluated using a prototype, with a proof-of-concept (PoC) implementation demonstrating its suitability for the defined objectives. To assess how the PoC implementation meets the objectives, straightforward metrics were defined to evaluate the derived concrete architecture against the objectives, similar to approaches used in related MLOps evaluation studies.

3.4.3 Evaluation Properties and Episodes

To systematically assess the implemented architecture, a set of evaluation properties was defined for each objective of the concrete architecture (3.3). These properties serve as measurable criteria to evaluate how well the objectives are met. For each property, an evaluation episode was created to assess how it is addressed in the architecture.

Table 3.3: Evaluation properties and episodes used to assess the reference and concrete architectures

Objective	Property	Evaluation Episode
Easy Adoption (O1)	Number of Tools	Ep1: Count the number of architecture tools
	Ready-made components	Ep2: Calculate the percentage of ready-made components in the architecture
Cost Effective (O2)	Operational cost	Ep3: Measure GPU resource utilization and idle time
Extendable (O3)	Pipeline maturity level	Ep4: Assess current MLOps maturity level against standard frameworks
	Modularity	Ep5: Evaluate component decoupling and replaceability
Ease of Use (O4)	Task completion time	Ep6: Measure time for a new team member to run a standard training job end-to-end

4 Design and Development

This chapter presents the architecture of the proposed MLOps system, designed to support AI-based GNSS data assimilation on self-managed GPU infrastructure. Based on the requirements outlined in 3, the design focuses on scalability, reproducibility, and cost efficiency under the constraints of industrial collaboration.

Rather than adopting a managed MLOps platform, the architecture integrates open-source tools and lightweight orchestration mechanisms tailored to the operational environments provided by Scaleway and ClearML. The design targets Level 2 (Automated Training) of the Microsoft MLOps Maturity Model, implementing six MLOps technical components from Section 2.2. This chapter follows the end-to-end training workflow to demonstrate how these components integrate to support the complete ML lifecycle.

4.1 Technology Stack

This section introduces the core technologies used to implement the MLOps system architecture and the rationale behind their selection. The technology choices balance practical constraints with technical requirements, prioritizing existing team expertise and partnership opportunities while maintaining open-source principles and European data residency. The stack consists of ClearML as the MLOps platform and Scaleway as the cloud infrastructure provider.

4.1.1 ClearML Platform

ClearML is an open-source MLOps platform that provides experiment tracking, workflow orchestration, and distributed training capabilities [8]. The platform consists of three architectural layers: the Infrastructure Control Plane for resource management, the AI Development Center for experiment tracking and model development, and the GenAI App Engine for model serving. This thesis primarily utilizes the first two layers.

ClearML was selected based on the industry partner's prior experience with the platform from previous projects. This existing familiarity enabled rapid adoption with a minimal learning curve, directly addressing the ease of adoption objective (O1). While alternatives such as MLflow, Kubeflow, and Weights & Biases exist, adopting a new platform would have required significant evaluation effort. The team's existing knowledge of ClearML's APIs, workflow patterns, and operational characteristics allowed the project to focus on addressing the core research challenge rather than on platform evaluation and learning. Section 2.4 discusses how similar MLOps architectures in both academic and industry contexts have successfully leveraged ClearML for research workflows.

The Infrastructure Control Plane implements a queue-based task distribution system where ClearML Agents running on compute nodes poll task queues and exe-

cute experiments remotely. This worker-queue architecture enables dynamic resource allocation, allowing agents to be started and stopped programmatically on cloud infrastructure based on workload demand. Each agent handles environment setup, dependency installation, code execution, and result reporting, abstracting infrastructure complexity from data scientists.

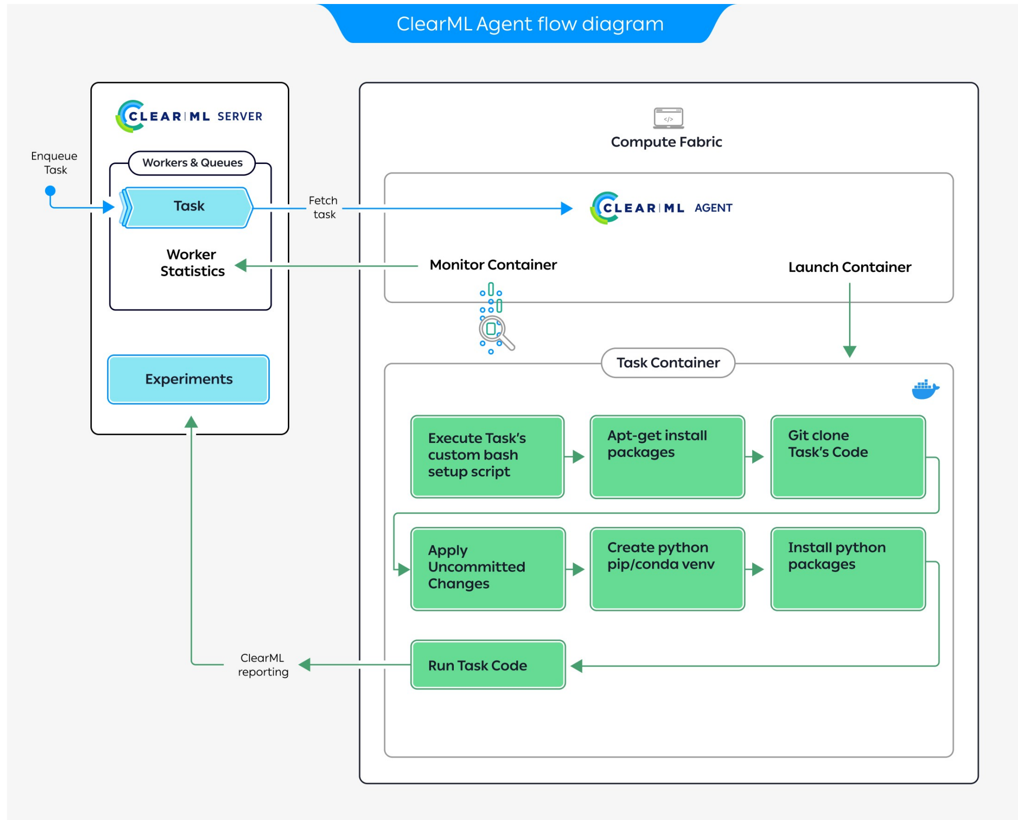

Figure 4.1: ClearML's worker-queue architecture showing agents polling task queues and executing on distributed compute resources

The AI Development Center provides comprehensive experiment tracking with automatic capture of training parameters, metrics, model checkpoints, and computational resource usage. The platform integrates seamlessly with popular ML frameworks, including PyTorch, TensorFlow, and scikit-learn, with minimal code modifications required. It also includes pipeline orchestration capabilities through ClearML Pipelines, which enables multi-step workflows as directed acyclic graphs (DAGs) with conditional execution and parameter passing between steps.

ClearML functions as both a metadata tracking system and a model registry, providing centralized storage for models with versioning, tagging, and complete lineage tracking from data and code to final artifacts. While the platform offers native autoscaling integrations for AWS and GCP, integration with other cloud providers requires custom implementation, which this thesis addresses for Scaleway infrastructure.

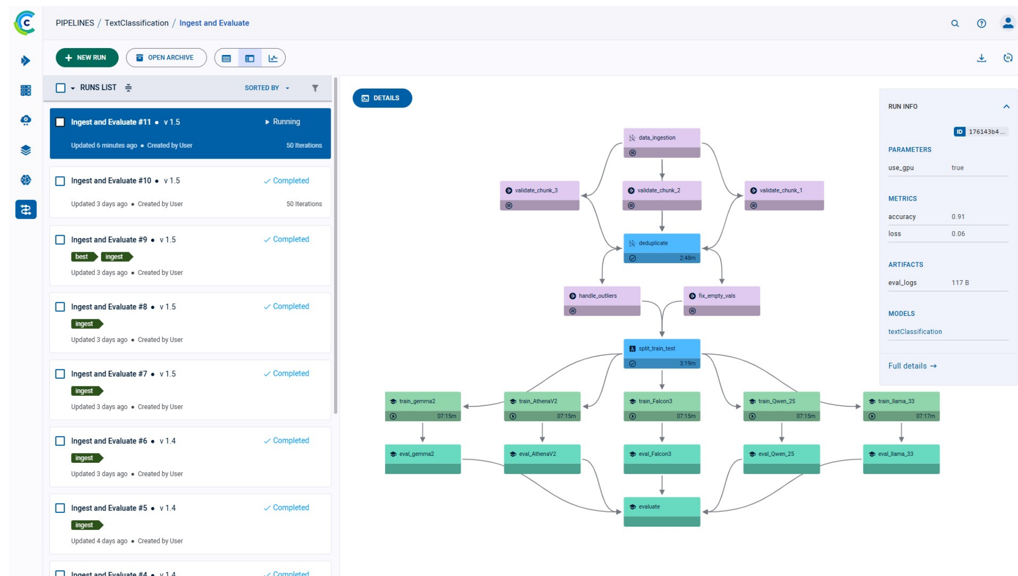

Figure 4.2: Example ClearML pipeline showing workflow orchestration with multiple steps executed on distributed resources

4.1.2 Scaleway Cloud Infrastructure

Scaleway is a European cloud computing provider offering infrastructure-as-a-service (IaaS) solutions with data centers across Europe, powered entirely by renewable energy. The platform provides comprehensive APIs and SDKs for programmatic infrastructure management, making it well-suited to dynamic resource provisioning.

Scaleway was chosen as the cloud infrastructure provider due to a partnership agreement that provided access to NVIDIA H100 GPU infrastructure through a credit system. Unlike ClearML, Scaleway was a new platform for the team, introducing a learning curve for understanding their API structure, provisioning patterns, and operational characteristics. However, the benefits of reduced operational costs outweigh the complexity of onboarding a new platform in this project's case, as GPU costs could reach thousands over the 12-month project period. The constraint of working with a new infrastructure provider became the thesis's core technical contribution, requiring a custom autoscaling integration that bridges ClearML's orchestration capabilities with Scaleway's programmatic infrastructure control.

This implementation utilizes three core Scaleway services:

- GPU Instances: Provides access to NVIDIA accelerators (L4, L40S, H100) for compute-intensive model training workloads. Instances can be programmatically provisioned and deprovisioned on demand.
- Serverless Containers: Enables the deployment of containerized applications without infrastructure management, automatically scaling based on incoming requests while charging only for the actual compute time consumed.
- Object Storage: Provides S3-compatible object storage with multi-

availability-zone redundancy for artifact persistence and model checkpoint storage.

The infrastructure provisioning leverages Scaleway's Go SDK, which provides type-safe interfaces for creating, configuring, and destroying cloud resources programmatically. This enables the dynamic resource allocation patterns described in Section 4.4, where GPU instances are provisioned on-demand in response to training workload requirements and deprovisioned upon completion to optimize costs. The combination of ClearML's queue-based orchestration and Scaleway's programmatic infrastructure control enables the autoscaling capabilities central to this thesis, bridging the gap between ClearML's native cloud integrations and European infrastructure providers.

4.1.3 Technology Stack Summary

Table 4.1 provides a comprehensive summary of all technologies used in the system, mapped to their corresponding MLOps technical components from Section 2.2. This mapping demonstrates how the technology choices collectively address the open-source requirement (R5).

The majority of the technology stack consists of tools already in use by the industry partner, including Git, Docker, PyTorch, Python, and the core ClearML platform for experiment tracking and metadata storage. This existing toolset required a minimal learning curve and enabled rapid development. The work on this thesis involves three new additions to the partner's existing stack: ClearML Queue and Agent for distributed task execution, Scaleway as the cloud infrastructure provider, and the Custom Golang Resource Manager that bridges the two.

4.2 Architectural Overview

This section presents the high-level architecture of the MLOps system, describing how six technical components integrate to support the end-to-end machine learning workflow. The architecture is designed to address the requirements defined in Section 3.3 while working within the constraints imposed by the partnership with Scaleway.

4.2.1 MLOps Maturity Level and Component Scope

This system targets Level 2 (Automated Training) of the Microsoft MLOps Maturity Model introduced in Section 2.3. This level is appropriate for research environments where the primary goal is efficient model development and experimentation rather than production deployment.

The implementation focuses on six of the nine MLOps technical components from Section 2.2: Source Code Repository (C2), Workflow Orchestration (C3), Model Training Infrastructure (C5), Model Registry (C6), ML Metadata Stores (C7), and Monitoring Component (C9). Rather than organizing the following sections by individual technical components, this chapter follows the end-to-end training work-

Table 4.1: Technology stack and MLOps component mapping

ID	MLOps Component	Technology	Details
C2	Source Code Repository	Git + GitHub	Private repository, version control
C3	Workflow Orchestration	ClearML Queue + Agent*	Queue-based task execution
C5	Model Training Infrastructure	Scaleway GPU Instances*	H100 series
C5	Model Training Infrastructure	Custom Component (Golang)*	Dynamic Provisioning
C6	Model Registry	ClearML Artifact Storage	Model checkpoint storage
C7	ML Metadata Stores	ClearML Server	Centralized tracking (SaaS)
C9	Monitoring Component	ClearML Dashboard	Built-in resource tracking
	Containerization	Docker	PyTorch public image
	ML Framework	PyTorch	2.7.0
	Programming Language	Python	3.12
	Dependency Management	uv + pyproject.toml	Poetry-compatible
	Data Storage	Scaleway Object Storage*	S3-compatible, Zarr format
	Data Processing	Zarr + Xarray	Multidimensional arrays
	Data Processing	Dask	Distributed I/O
	Infrastructure as Code	Terraform[1]	Deployment automation

* New technology introduced for this project

flow to demonstrate how these elements integrate. Table 4.2 maps each technical component to its specific implementation.

The six implemented technical components (C2, C3, C5, C6, C7, C9) directly support the functional requirements (R1–R7) defined in Section 3.3 and align with Level 2 capabilities from Table 2.3. Three components from Section 2.2 are explicitly out of scope: CI/CD Component (C1), as manual deployment is acceptable for research contexts, Feature Store System (C4), as it does not apply to static weather reanalysis data, and Model Serving Component (C8), as production serving is outside the scope of the thesis.

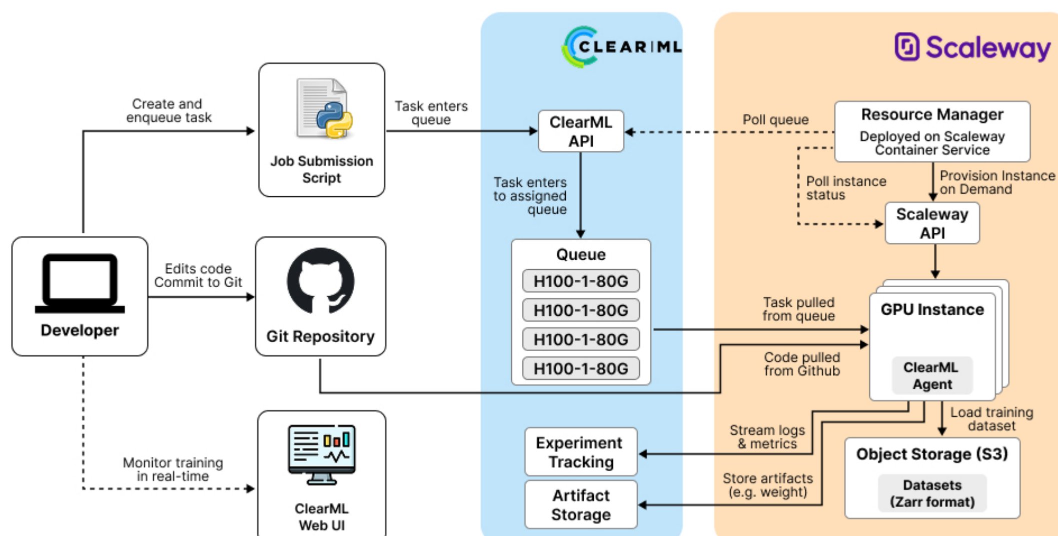

Figure 4.3: System architecture diagram illustrating the integration of the core MLOps components and the data flow.

Table 4.2: Implemented MLOps technical components

ID	MLOps Component	Implementation
C2	Source Code Repository	Git + GitHub for version control
C3	Workflow Orchestration	ClearML queue-based task orchestration
C5	Model Training Infrastructure	Scaleway GPU instances with dynamic provisioning
C6	Model Registry	ClearML artifact storage for model checkpoints
C7	ML Metadata Stores	ClearML centralized experiment tracking
C9	Monitoring Component	ClearML built-in resource and metrics monitoring

4.2.2 System Architecture and Workflow

The architecture integrates six MLOps technical components: source code repository (C2), workflow orchestration (C3), model training infrastructure (C5), model registry (C6), ML metadata stores (C7), and monitoring (C9). Figure 4.3 illustrates the complete system architecture and the information flow between these elements throughout the training workflow.

When a developer wishes to train their model, they first commit and push their code changes to the source code repository (C2) hosted on GitHub. The developer then executes a task submission script that uses the ClearML Python API to create and enqueue a training task. Each task submission specifies the reposi-

tory URL, branch name, execution module, runtime arguments, and Docker base image. Once submitted, the task enters the workflow orchestration component's (C3) queue system, where it awaits execution.

The model training infrastructure component (C5) continuously monitors the ClearML queue via the HTTP API, polling every 30 seconds to assess the demand for computational resources. When pending tasks are detected and insufficient GPU instances are available, the resource manager provisions new instances through the Scaleway API. This provisioning process takes approximately 2–4 minutes and includes the automated installation of the ClearML Agent, the Docker runtime, and the necessary credentials via cloud-init scripts.

Once a GPU instance is provisioned and the ClearML Agent is running, the agent polls its assigned queue and pulls pending tasks. The agent executes the task within a Docker container, ensuring environment consistency. During execution, the training script reports metrics and logs to the ML metadata stores component (C7) and uploads model checkpoints to the model registry (C6). Developers monitor training progress in real-time through ClearML's web-based user interface using the monitoring component (C9).

Upon task completion, if no additional tasks are queued, the model training infrastructure component (C5) detects the idle state. After a configurable 30-minute timeout, the resource manager gracefully terminates the instance via the Scaleway API, ensuring cost efficiency by avoiding unnecessary compute charges.

4.2.3 Requirements Mapping

Table 4.3 maps each functional and non-functional requirement from Section 3.3 to the technical components responsible for addressing them.

R1 (Dynamic Resource Management) is addressed by the model training infrastructure component (C5), which implements the provisioning and shutdown logic described in Section 4.4. R2 (Reproducibility) is achieved through the source code repository (C2) and ML metadata stores (C7), with automatic Git integration and Docker containerization, ensuring that both code versions and runtime environments are captured. R3 (Experiment Tracking) relies on comprehensive logging capabilities provided by the ML metadata stores (C7), which are integrated directly into the training scripts. R5 (open source Tooling) is satisfied with the technology choices in Table 4.1, with ClearML being the only SaaS component, although there is an open source version available through self-hosting. R6 (Performance Monitoring) leverages the monitoring component (C9) without custom instrumentation. Finally, R7 (Scalability) is supported by the model training infrastructure (C5) and PyTorch's native distributed data parallelism, which achieves near-linear speedup across multiple GPUs.

The following sections examine the implementation following the end-to-end training workflow:

- Section 4.3 describes task submission and workflow orchestration (C2, C3)
- Section 4.4 presents dynamic compute provisioning within the model training infrastructure (C5)

Table 4.3: Mapping of requirements to MLOps technical components

	Requirement	Component	Implementation
R1	Dynamic Resource Management	C5	Model training infrastructure provisions/deprovisions Scaleway instances based on queue demand
R2	Reproducibility	C2, C7	Source code repository and ML metadata stores capture Git commit, Python environment, hyperparameters
R3	Experiment Tracking	C7	ML metadata stores log all metrics, hyperparameters, artifacts, and system resources
R5	Open-source Tooling	All	All technical components use open-source tools (Docker, PyTorch, Git) and APIs
R6	Performance Monitoring	C9	Monitoring component tracks GPU utilization, memory, training metrics in real-time
R7	Scalability	C5	Model training infrastructure scales instances linearly with queue depth; PyTorch supports multi-GPU training

- Section 4.5 explains data acquisition, preprocessing, and storage practices
- Section 4.6 discusses experiment tracking and reproducibility through ML metadata stores and model registry (C6, C7)
- Section 4.7 covers performance monitoring (C9)

4.2.4 External Dependencies

The system integrates with three external services: GitHub for version control, Scaleway for infrastructure provisioning, and ClearML Cloud for experiment tracking and orchestration. Git credentials are injected into GPU instances via cloud-init scripts during provisioning, enabling ClearML Agents to clone private repositories. The Scaleway Golang SDK is used to interact with the Instance API and Volume API for provisioning and cleanup operations. Dataset access is provided through Scaleway's S3-compatible Object Storage, where data is stored in Zarr format and accessed using the xarray and Dask libraries. The complete data processing pipeline is described in Section 4.5.

The architecture presented above provides a clear separation of concerns between ClearML (C3, C6, C7, C9) and Scaleway (C5), with the custom dynamic provisioning service acting as the bridge within the model training infrastructure (C5).

This design enables each technical component to be replaced independently.

4.3 Training Orchestration

This section describes the training orchestration component, which manages the automated workflow for training diffusion models. Training orchestration encompasses task submission, queue management, and agent-based execution. The system leverages ClearML's queue-based task management to minimize developer friction, enabling developers to focus on model development rather than infrastructure management. Figure 4.4 illustrates the task submission workflow from code preparation through queue placement.

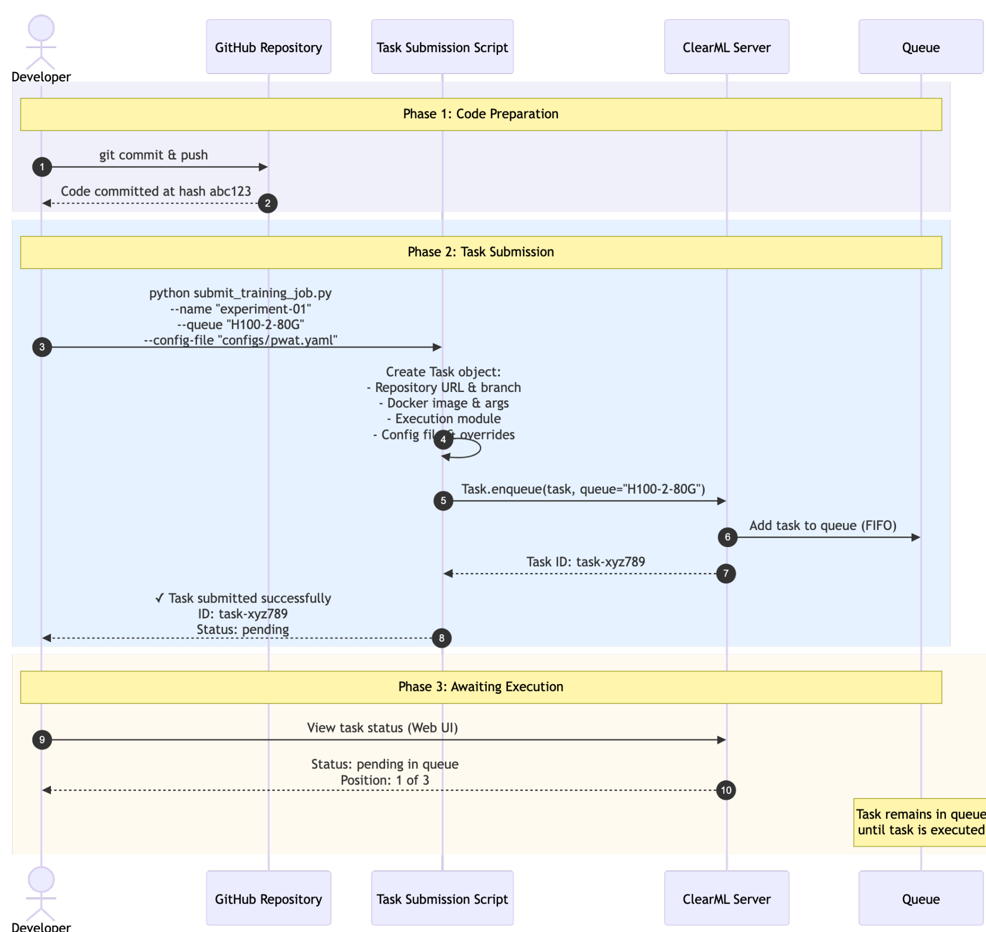

Figure 4.4: Sequence diagram showing the automated process from code commit to task submission and queue placement in ClearML.

4.3.1 Task Submission

Developers submit training tasks through a Python script that interfaces with the ClearML API. The submission process requires the developer to commit and push code changes to the Git repository before creating a task, ensuring that each experiment is associated with a specific code version. This enforces reproducibility (R2) by preventing experiments from running with uncommitted or "dirty" code

states. The design intentionally blocks task submission if uncommitted changes are detected, trading a small amount of developer friction (requiring explicit commits) for a strong reproducibility guarantee. This strict policy is particularly important for research environments where experiments may be revisited months later.

Listing 4.1 shows a simplified example of the task submission script. The developer specifies a task name, target queue, and configuration file, along with optional hyperparameter overrides.

```
python scripts/train/submit_training_job.py \
    --name "pwat-diffusion-experiment-01" \
    --queue "H100-2-80G" \
    --config-file "configs/pwat.yaml" \
    --batch-size 16
```

Listing 4.1: Task submission command

The submission script creates a ClearML task object with the following key parameters:

- Repository URL and branch: Points to the Git repository and specific branch containing the training code
- Execution module: Specifies the entry point, using PyTorch's distributed launcher for multi-GPU support: `torch.distributed.run --nproc_per_node=auto scripts/train/train.py`
- Docker image: Defines the base container image (`pytorch/pytorch:2.7.1-cuda11.8-cudnn9-runtime`)
- Docker arguments: Passes runtime flags to enable GPU access (`--runtime=nvidia --gpus all`)
- Configuration file: References a YAML file containing the base configuration for model architecture, training hyperparameters, and data specifications
- Hyperparameter overrides: Allows runtime customization of parameters defined in the configuration file

The queue name determines which GPU instance type will execute the task. The system currently supports four queues corresponding to different Scaleway H100 configurations: `H100-1-80G` (single GPU), `H100-2-80G` (2 GPUs), `H100-sxm-4-80G` (4 GPUs with SXM interconnect), and `H100-sxm-8-80G` (8 GPUs with SXM interconnect).

Once the task object is created, the script invokes `Task.enqueue()` to submit the task to the specified queue. ClearML assigns the task a unique identifier and transitions it to the "pending" state, where it awaits execution by an available agent. This entire submission process takes approximately 1 minute for a developer (including commit, push, and task creation), compared to the previous manual approach of SSH-ing into machines and launching training scripts within screen sessions, which typically took 5–10 minutes and required maintaining SSH keys and remembering machine addresses. Reducing submission friction enables rapid iteration during model development.

4.3.2 Queue Management

ClearML implements a queue system on a first-in, first-out (FIFO) basis, ensuring predictable task ordering without complex prioritization logic. The simple implementation of FIFO scheduling also reduces the system's operational complexity.

When multiple tasks are submitted to the same queue, they form an ordered list to which agents are assigned sequentially. If multiple agents are assigned to a queue, tasks are distributed as agents become available, enabling parallel execution across multiple machine instances. There is no hard limit on queue length; tasks will wait indefinitely until computational resources become available or the developer cancels them.

The resource manager monitors queue depth to determine provisioning needs. When the number of pending tasks exceeds the number of active agents, and the instance count is below the configured maximum, the resource manager provisions additional GPU instances. This dynamic scaling ensures that queued work progresses without manual intervention, addressing requirement R1 (Dynamic Resource Management).

Each queue's configuration specifies minimum and maximum instance counts. The minimum count (typically 0) allows full resource deprovisioning during periods of inactivity, while the maximum count prevents runaway provisioning due to misconfiguration or API errors. These bounds provide cost control while maintaining sufficient capacity for typical workloads.

4.3.3 Agent Execution Model

The ClearML Agent is installed on each provisioned GPU instance through a cloud-init script that executes during the instance boot process. Figure 4.5 illustrates the continuous lifecycle of the agent from installation through its polling and execution loop. The installation script performs the following steps:

1. Create a Python virtual environment and install the `clearml-agent` program.
2. Configure the agent with API credentials for the ClearML server.
3. Inject Git credentials into the machine environment to enable access to the private repository.
4. Register the agent with a specific queue based on the instance type.
5. Set up a systemd service to run the agent in daemon mode, enabling the system to operate even after a machine restart.

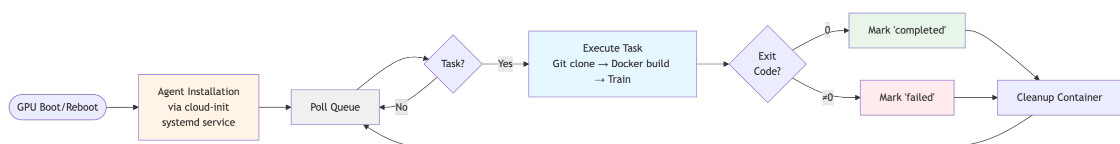

Figure 4.5: Simplified flowchart showing the continuous agent lifecycle.

Once running, the agent continuously polls its assigned queue using ClearML's API. The polling mechanism uses a pull-based model where the agent requests new tasks from the server rather than the server pushing tasks to agents. This design simplifies network configuration and security management on the Scaleway infrastructure side, as agents only need outbound HTTP/HTTPS connectivity to the ClearML server.

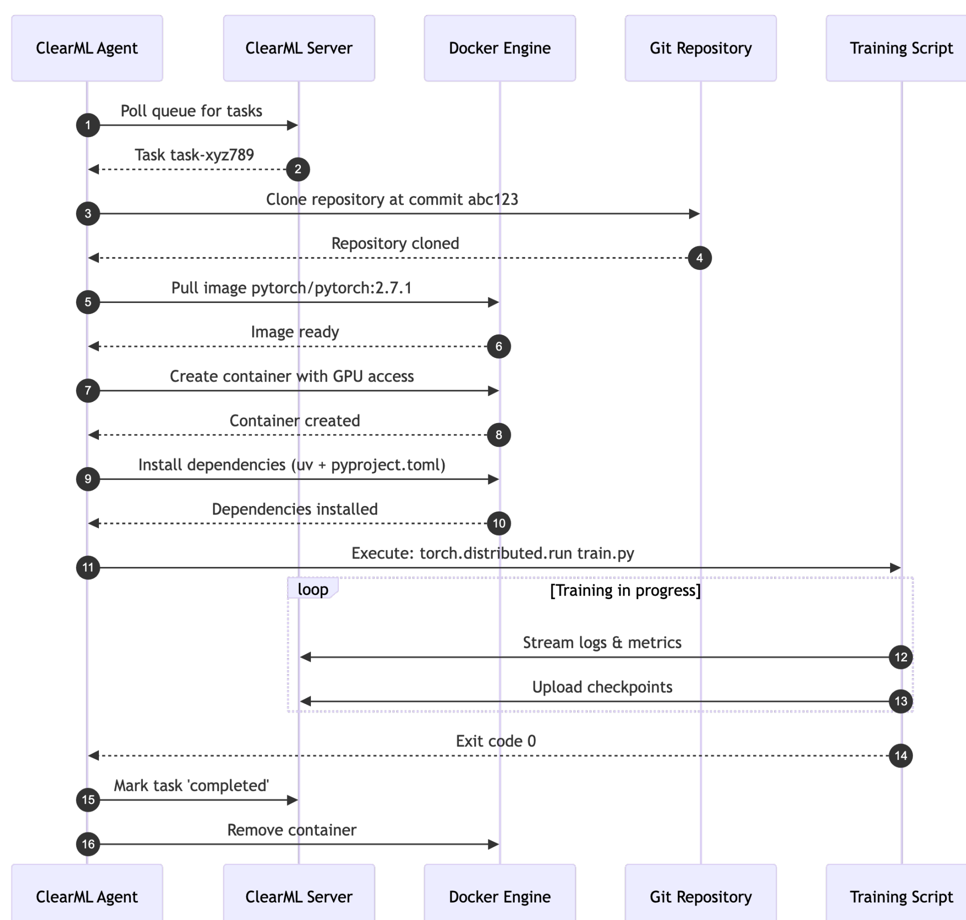

Figure 4.6: Sequence diagram showing the detailed task execution flow when executing tasks.

When the agent retrieves a task from the queue, it performs the following execution sequence (illustrated in Figure 4.6):

1. Clones the Git repository at the specified branch and commit hash
2. Pulls the specified Docker base image if it is not already cached locally
3. Creates a Docker container with GPU access enabled
4. Installs Python dependencies specified in `pyproject.toml` using the `uv` package manager within the container
5. Executes the training module with the configured arguments
6. Streams stdout/stderr logs to the ClearML server in real-time
7. Captures and reports metrics logged by the training script
8. Uploads artifacts (model checkpoints, plots) to ClearML's artifact storage

9. Marks the task as ”completed” or ”failed” based on the exit code

Environment reproducibility is ensured through Docker containerization combined with explicit dependency management. Every task execution starts with a clean container based on the specified base image, eliminating dependency conflicts between experiments. The `pyproject.toml` file pins exact versions of all Python dependencies, and the uv package manager performs a fresh installation for each task, preventing version drift. This containerized approach addresses the reproducibility requirement (R2) by isolating each experiment’s execution environment.

If a task fails during execution, the ClearML Agent marks it as ”failed” and logs the error information. The system does not implement automatic retry logic; failed tasks remain in the ”failed” state until a developer manually requeues them. This design choice prevents infinite loops in case of persistent errors (e.g., code bugs, configuration mistakes) and provides developers with full visibility into failure patterns.

The training orchestration component directly addresses two key requirements. First, the reproducibility requirement (R2) is met by ensuring that every task references a specific Git commit and is executed within a designated Docker container. This combination of version-controlled code, explicitly pinned dependencies, and containerized execution effectively eliminates the typical ”works on my machine” problem in research environments. Second, the experiment tracking requirement (R3) is fulfilled through comprehensive automatic metadata capture: the system records the Git commit hash, branch name, execution parameters, and runtime environment for every experiment, linking them to metrics and artifacts without requiring manual logging.

The queue-based design provides natural load balancing and resource pooling. Multiple developers can submit tasks to shared queues without coordinating access to specific machines, supporting collaborative workflows that would be cumbersome with manual machine allocation. This design scales from single-user development to multi-user team collaboration without architectural changes.

4.4 Compute Resource Management

Compute resource management is a critical component of this architecture, directly addressing the dynamic resource management requirement (R1). This section describes how the system automatically provisions and deprovisions GPU instances based on workload demand, eliminating the need for always-on infrastructure while ensuring that computational resources are available when needed. The resource manager is implemented as a standalone Golang service deployed on Scaleway’s Container Service, a serverless platform that provides automatic health monitoring and fault recovery.

4.4.1 Provisioning Logic

The resource manager operates on a configurable polling cycle (defaulting to 30 seconds), during which it queries both the ClearML API and the Scaleway API to sync the desired state (based on queue demand) with the actual state (number of running instances). The 30-second interval was chosen as a balance between responsiveness and API rate limiting: shorter intervals (e.g., 10 seconds) would provide marginally faster reaction times but increase API costs and risk hitting rate limits, while longer intervals (e.g., 60 seconds) would unnecessarily delay provisioning decisions, given that instance creation takes 2–4 minutes regardless.

For each queue, the provisioning decision logic follows this procedure:

```
every 30 seconds:
    for each queue in configuration (concurrent):
        # Getting server data from Scaleway
        instances = ListServers(queue_name)

        # Getting agent data from ClearML
        agents = GetAgents(queue_name)

        # Getting queue data from ClearML
        pendingJobs = GetPendingJobCount(queue_name)

        # Update internal agent state map
        for instance in instances:
            agent = agents[instance.id]
            agent.state = DetermineState(instance, agents)

        # Calculate scaling needed
        numIdle = count(agents where state == Idle)
        numInitializing = count(agents where state == Initializing)
        currentAgents = count(agents where state != Stopped)

        # Provisioning decision
        if pendingJobs > (numIdle + numInitializing)
        and currentAgents < maxInstances:
            numToCreate = min(
                pendingJobs - numIdle - numInitializing,
                maxInstances - currentAgents
            )

            # Try starting stopped instances first
            for stopped_agent in stopped_agents[:numToCreate]:
                StartInstance(stopped_agent.id)
                numToCreate -= 1

            # Create new instances for remaining demand
            for i in range(numToCreate):
```

```
CreateInstance(queue_name)
```

Listing 4.2: Resource manager provisioning logic (pseudocode)

The provisioning threshold is effectively zero pending tasks—any queued work triggers provisioning if capacity is available and no resource unavailability backoff is active (see error handling below). This aggressive provisioning strategy prioritizes lowering the friction experienced by the developer.

When the provisioning decision is made, the resource manager executes a sequence of Scaleway API calls to create and configure the instance:

1. CreateServer: Creates a new GPU instance with the following parameters:
   - Name: Generated with pattern `queuename-uuid6`
   - CommercialType: Instance type from queue configuration (H100-1-80G, H100-2-80G, H100-SXM-4-80G, or H100-SXM-8-80G)
   - Zone: Scaleway availability zone (e.g., `fr-par-2`)
   - Image: Ubuntu 22.04 LTS base image
   - Tags: Includes `queue=queuename` for filtering in ListServers
   - Volumes: SBS (Scaleway Block Storage) volume with configurable size (default or per-queue override)

2. SetServerUserData: Injects a cloud-init bash script that executes on first boot. The script performs:
   - (a) System package installation: `apt-get install python3-venv build-essential`
   - (b) Python virtual environment creation in `/root/clearml_agent_venv`
   - (c) ClearML Agent installation: `pip install clearml-agent`
   - (d) Configuration file creation at `/root/clearml.conf`:
     - Scaleway S3 credentials (for dataset access)
     - Git credentials (username and password for private repository)
   - (e) Environment file creation at `/etc/clearml/environment`:
     - ClearML API endpoints (API, Web, Files hosts)
     - ClearML access key and secret key
     - Worker ID (set to instance name for tracking)
   - (f) Systemd service creation at `/etc/systemd/system/clearml-agent.service`:
     - ExecStart command: `clearml-agent --config-file /root/clearml.conf daemon --docker --create-queue --queue 'queuename'`
     - Restart policy: `always` (automatic recovery on crashes)
     - Loads environment variables from `/etc/clearml/environment`

(g) Service enablement: `systemctl enable` and `start clearml-agent.service`

3. ServerAction (Poweron): Boots the instance with a timeout. If this fails, the instance is cleaned up via DeleteInstance.

4. WaitForServer: Blocks until the instance reaches the ”running” state. This ensures the cloud-init script begins execution before the API call returns.

The entire provisioning process, from `CreateServer` to agent registration with ClearML, typically takes 2–5 minutes. During this time, tasks remain pending in the ClearML queue. Once the agent comes online and begins polling, it immediately picks up waiting tasks.

Provisioning operations execute concurrently for all queues where the resource manager spawns goroutines for each queue's reconciliation, with mutex locks protecting shared state. Multiple instances can provision simultaneously, but each queue's provisioning logic runs atomically to prevent over-provisioning. A queue-level reconciliation lock prevents concurrent reconciliation of the same queue if the previous cycle has not completed.

4.4.2 Idle Detection and Shutdown Policy

The resource manager maintains an internal state representation for each provisioned instance by cross-referencing data from two sources: the Scaleway `ListServers` API (which reports instance existence and status) and the ClearML `GetWorkers` API (which reports agent registration and task assignments). This enables precise idle detection even if an agent fails to register or crashes after provisioning.

During each reconciliation cycle, the resource manager performs agent state synchronization:

1. Query Scaleway instances filtered by queue tag (`queue=queuename`)
2. For each instance, retrieve or create an Agent record with fields:
    - `InstanceID`, `InstanceName`, `CreatedAt`, `LastActivityAt`
    - `State`: Initializing, Idle, RunningTask, Stopped, Stopping
    - `LastBusyAt`: Timestamp of last task execution
    - `ClearMLWorkerID`: Links instance to ClearML worker registration
3. Query ClearML workers for the queue and match by worker ID (which equals instance name)
4. Update agent state based on worker task status:
    - If `worker.Task != nil`: transition to RunningTask, update `LastBusyAt`
    - If `worker.Task == nil` and state was RunningTask: transition to Idle (with 2-minute grace period if pending jobs exist, to handle API lag)
    - If worker not found and instance exists: remains Initializing

Two timeout thresholds govern shutdown decisions, both configurable per-queue with values chosen based on observed usage patterns and cost-benefit analysis:

- Idle timeout (default 30 minutes): If an agent has been idle with no tasks for this duration, it is marked for shutdown. This 30-minute value was determined after multiple trials, where shorter timeouts led to machines being shut down too quickly—often while developers were still fixing or investigating failed runs. The chosen timeout avoids terminating machines while a fix is ongoing, balancing workflow efficiency and cost savings.
- Stale timeout (default 10 minutes): If an agent remains in the Initializing state (never registering with ClearML), it is terminated after this period. This helps prevent resource waste from failed or stuck provisioning attempts.

Shutdown logic executes after provisioning decisions in each reconciliation cycle:

```
# Identify idle agents meeting timeout criteria
agentsToKill = []
for agent in currentAgents:
    if shouldKillIdle(agent, queueConfig):
        agentsToKill.append(agent)

# Calculate number to stop
numToKill = 0
if currentAgents > maxInstances:
    numToKill = currentAgents - maxInstances
elif pendingJobs == 0 and numIdle > minInstances:
    numToKill = numIdle - minInstances

if numToKill > 0 and len(agentsToKill) > 0:
    numToKill = min(numToKill, len(agentsToKill))

    # Sort agents by idleness (most idle first)
    agentsToKill.sort(key=lambda a: timeSince(a.LastBusyAt),
                                    reverse=True)

    # Stop the most idle agents
    for agent in agentsToKill[:numToKill]:
        StopInstance(agent.InstanceID)
        agent.State = Stopped

# Separately, delete stale initializing instances
for agent in currentAgents:
    if shouldKillStale(agent, queueConfig):
        DeleteInstance(agent.InstanceID)
        # DeleteInstance performs:
        # 1. ServerAction(Kill)
        # 2. Wait for instance deletion
        # 3. Delete all attached volumes
```

Listing 4.3: Resource manager deprovisioning logic (pseudocode)

Importantly, stopped instances can be restarted via `StartInstance` in subsequent cycles if demand returns, providing faster scale-up than provisioning new instances. Only stale initializing instances are permanently deleted.

The shutdown process is graceful: instances receive a standard OS shutdown signal, allowing the ClearML Agent to complete any in-progress reporting. Instances actively executing tasks (state == RunningTask) are never marked for shutdown, regardless of how long the task runs.

4.4.3 Scaleway API Integration

The resource manager uses the official Scaleway Golang SDK (`github.com/scaleway/scaleway-sdk-go/api/instance/v1` and `github.com/scaleway/scaleway-sdk-go/api/block/v1alpha1`) to interact with the Instance API and Block Storage API. Authentication is performed using API keys (access and secret keys) injected into the container environment via Scaleway's secret management service, configured in Terraform. The SDK client is initialized with the default project ID and zone, simplifying subsequent API calls.

The primary API methods used are:

- instance.CreateServer: Provisions a new GPU instance, returns server ID immediately
- instance.SetServerUserData: Injects cloud-init bash script as user data
- instance.ServerAction: Performs power actions (Poweron, Poweroff, Terminate)
- instance.WaitForServer: Blocks until the instance reaches the target state (used after create/power actions)
- instance.ListServers: Retrieves instances filtered by tags (e.g., `queue=H100-2-80G`)
- instance.GetServer: Fetches single instance details, including attached volumes
- block.DeleteVolume: Removes block storage volumes by ID

The ClearML integration uses HTTP API calls (no official Golang SDK exists):

- POST /auth.login: Authenticates using Basic auth (base64-encoded access:secret) and returns a bearer token
- POST /queues.get_all: Retrieves queue definitions (maps queue names to IDs) or queries a specific queue by ID to get the pending task count (length of `entries` array)
- POST /workers.get_all: Lists all registered workers, filtered client-side by queue ID to find workers for a specific queue

Error handling includes automatic cleanup on provisioning failures: if `SetServerUserData` or `ServerAction` fails after `CreateServer`, the resource manager invokes `DeleteInstance` to delete the partially created instance and prevent orphaned resources. For transient API failures, no explicit

retry logic exists, the polling loop naturally retries on the next cycle (30 seconds later).

One issue we encountered was GPU resource scarcity with Scaleway, which resulted in an out-of-stock error. We enable exponential backoff as a solution where the resource manager marks the queue as unavailable and skips provisioning attempts for a backoff period (5 minutes initially, doubling on subsequent failures, capped at 60 minutes). This prevents log spam and unnecessary API calls when Scaleway cannot fulfill provisioning requests. The backoff period is tracked per-queue in the `QueueState` struct with fields `ResourceUnavailableUntil` and `ResourceUnavailableCount`. Rate limiting is not explicitly implemented, as the 30-second polling interval provides sufficient spacing.

The compute resource management component directly addresses two core objectives. Dynamic resource management requirement (R1) is achieved through automatic scaling of GPU infrastructure from zero to the configured maximum based on real-time workload; during periods of inactivity, no operational costs are incurred, while developers experience provisioning latency of 2–4 minutes when submitting tasks. The implementation of the 30-minute idle timeout strikes a balance between aggressive cost optimization and user experience—shorter timeouts would save marginal costs but increase cold start frequency. In comparison, longer timeouts would incur unnecessary charges during typical developer workflows. Based on observed usage patterns, the compute resource management component reduces compute costs by approximately 75% compared to an always-on infrastructure model, translating to estimated monthly savings of €2.250–€3.000 for the industry partner's workload (assuming 8–12 hours of active training per day on H100-2-80G instances).

The trade-off is the 2–4 minute provisioning latency, which is acceptable for the long-running training workloads typical in this domain (experiments run 6–12 hours). The design is intentionally simple, prioritizing reliability over sophisticated optimization. The current implementation achieves the primary goal, eliminating idle resource waste with minimal code and operational burden.

4.5 Data Processing

Data processing is a fundamental aspect of any MLOps system, particularly in scientific machine learning applications where reproducibility is essential [17]. This section describes the data acquisition, preprocessing, storage, and access patterns implemented to support diffusion model training for GNSS data assimilation. The system handles weather reanalysis data from the High-Resolution Rapid Refresh (HRRR)[2] dataset, transforming it from its original format into a cloud-optimized representation suitable for distributed training workflows.

[2] https://rapidrefresh.noaa.gov/hrrr/

4.5.1 Dataset Description and Source

The project uses the High-Resolution Rapid Refresh (HRRR) dataset provided by the National Oceanic and Atmospheric Administration (NOAA). HRRR is a real-time atmospheric model that provides hourly updated forecasts and analysis data at a 3-kilometer spatial resolution across North America [19]. The dataset includes numerous meteorological variables captured at multiple atmospheric pressure levels, as well as surface observations. All raw HRRR data is hosted on NOAA's public cloud storage infrastructure, accessible via HTTP endpoints organized by date and forecast hour. The downloaded files are in GRIB2[3] format, a standard established by the World Meteorological Organization for the storage and transmission of gridded meteorological data.

For this work, the following variables are selected as input features for model training:

- Total surface precipitation over 1 hour (PWAT): Accumulated precipitation at the surface
- Wind components (U and V): Horizontal wind velocity components at 10 meters

These variables, along with the temporal and spatial restrictions, exactly match the experimental setup used in the reference implementation [19]. Specifically, the dataset uses the same time period (2017–2022), the same spatial domain (Oklahoma region covering 148.939 km²), and the same variable selection (PWAT and wind components U and V). The goal of this replication is to serve as a sanity check, validating that our MLOps infrastructure can successfully reproduce the reference implementation's results before moving on to the geographic regions and time periods of primary interest for our GNSS data assimilation research.

4.5.2 Data Acquisition and Preprocessing Pipeline

Rather than downloading the entire multi-terabyte archive, the system implements a serverless preprocessing pipeline that operates on demand, processing only the required subset of the data. The preprocessing pipeline is executed using Scaleway Serverless Jobs[4], a managed service that runs containerized tasks without requiring infrastructure provisioning or management. This approach is straightforward: the pipeline consists of a single script that downloads the required GRIB2 files from NOAA's public storage, processes them, and writes the results to a Zarr store in Scaleway Object Storage.

Scaleway Serverless Jobs provides a fully managed execution environment where tasks run in isolated containers. The service handles all infrastructure concerns, including container orchestration, resource allocation, and automatic scaling. Developers simply submit a container image and specify the command to execute; the service manages the lifecycle, monitoring, and logging. This eliminates the operational overhead of maintaining dedicated preprocessing infrastructure while ensuring that data preparation tasks can run independently of the training system.

[3] https://www.nco.ncep.noaa.gov/pmb/docs/grib2/grib2_doc
[4] https://www.scaleway.com/en/serverless-jobs/

The preprocessing workflow consists of three sequential stages:

1. **Query and Download**: The pipeline queries the HRRR archive by date, retrieving GRIB2-formatted files for each forecast hour within the target time range (2017–2022).
2. **Geographic Cropping**: Each downloaded file covers the entire North American domain. The pipeline applies geographic subsetting to extract only the Oklahoma region, discarding data outside the target bounding box.
3. **Aggregation to Zarr**: The cropped data from multiple time steps is aggregated into a single Zarr[5] store. Zarr is an open-source format for chunked, compressed, N-dimensional arrays designed for cloud storage environments. The aggregation process consolidates hundreds of individual files into a single logical dataset with time, latitude, and longitude as dimensions.

The choice of Zarr format provides several technical advantages for machine learning workflows. Zarr partitions large arrays into smaller chunks that can be read independently, enabling parallel data loading during training where multiple workers can simultaneously access different temporal or spatial regions without lock contention. Unlike traditional file formats requiring full downloads before processing, Zarr supports random access via HTTP range requests, allowing arbitrary data subsets to be read directly from cloud storage, thereby reducing startup latency and eliminating the need for large local disks on training instances. Zarr has become a de facto standard in atmospheric and geospatial sciences [1], with native support in widely used libraries such as xarray, Dask, and TensorFlow I/O, ensuring ecosystem maturity that reduces implementation complexity and guarantees long-term maintainability.

4.5.3 Data Storage and Training Pipeline

The processed Zarr datasets are stored in Scaleway Object Storage, an S3-compatible service providing durable, scalable storage with low-latency access from compute instances. The complete six-year Oklahoma HRRR dataset occupies approximately 20 GB in Zarr format after compression. Access is authenticated using access keys injected into GPU instances during provisioning. Object storage provides operational benefits: data persists independently of compute instances, multiple training jobs can access the same dataset concurrently without duplication, and storage scales transparently without capacity planning. Additionally, since the object storage and compute instances share the same VPC, there are no egress costs or additional data transfer fees.

During training, data is loaded directly from Scaleway Object Storage, eliminating the need for local copies on GPU instances. This cloud-native access pattern integrates Zarr for storage format, xarray for labeled array manipulation, and Dask[6] for lazy loading and distributed computation. Dask implements lazy evaluation, fetching only the specific Zarr chunks required for each training batch as needed, rather than downloading the entire 20 GB dataset. This allows training to start immediately without waiting for a complete dataset download. The system avoids

[5] https://zarr.dev/
[6] https://www.dask.org/

data loading bottlenecks because GPU computation time exceeds network I/O time for the chunk sizes used, ensuring data is always available when needed.

The data loading workflow initializes the Zarr store using xarray's `open_zarr()` function (reading only metadata), selects required variables (PWAT and wind components U and V), creates temporal splits for training/validation/test, and integrates with PyTorch's `DataLoader` for batching and parallel loading. Normalization is applied on the fly using precomputed statistics. For multi-GPU training with DistributedDataParallel, Dask's distributed scheduler coordinates chunk requests and caches frequently accessed data, preventing redundant downloads.

4.5.4 Data Versioning and Reproducibility

The HRRR dataset represents historical observations that remain immutable once processed and stored in Scaleway Object Storage. Unlike continuously updated data streams, the 2017–2022 dataset does not evolve, simplifying versioning requirements and enabling a streamlined approach to reproducibility.

The system achieves reproducibility through experiment hyperparameters rather than explicit dataset versioning infrastructure. As described in Section 4.6, each experiment records essential dataset metadata:

- **Bucket name and object key**: Identifies the specific Zarr store (e.g., `s3://datasets/oklahoma-hrrr.zarr`)
- **Variable selection**: Lists which variables are used
- **Temporal subsets**: Defines the year ranges for training, validation, and test splits
- **Statistics file path**: References the normalization statistics used for preprocessing

This metadata provides sufficient information to reproduce the exact dataset configuration of any past experiment. A developer can retrieve the dataset by accessing the recorded Scaleway Object Storage bucket, loading the Zarr array via xarray, selecting the specified variables, slicing the temporal range, and applying the recorded normalization statistics. The deterministic nature of array indexing operations ensures that identical slices produce identical data across reproduction attempts.

This hyperparameter-based tracking approach deliberately avoids explicit dataset versioning for practical reasons, since there is no plan to modify the data after initial preprocessing. The more straightforward approach reduces operational overhead, accelerates implementation, and eases maintenance without compromising reproducibility. Should future work require handling evolving datasets (e.g., adding new time periods, incorporating additional spatial regions, or refining preprocessing steps), then the system could adopt explicit versioning through ClearML's Dataset API or content-addressable storage tools like DVC. However, for the current use case of training on fixed historical reanalysis data, the streamlined approach aligns with the system's design principles of operational simplicity while fully satisfying reproducibility requirements.

The data processing component addresses requirement R2 (Reproducibility) by ensuring that every experiment's data configuration is fully documented and retrievable. Combined with code versioning through Git and environment reproducibility via Docker (described in Section 4.6), the system provides end-to-end reproducibility from raw data through trained models.

The data processing architecture prioritizes simplicity and cloud-native access patterns over complex data pipeline orchestration. By leveraging Zarr's efficient chunking and Scaleway Object Storage's scalability, the system eliminates the operational burden of managing local datasets on training instances while maintaining sufficient I/O throughput for GPU-intensive workloads. This design directly addresses the reproducibility requirement (R2) by tracking dataset configurations and maintaining immutable data artifacts in durable cloud storage.

4.6 Experiment Tracking

Comprehensive experiment tracking is essential for scientific reproducibility and experiment provenance [2], [17]. This section describes how the system captures, stores, and provides access to experiment metadata, enabling developers to reproduce past results and trace the lineage of trained models. The experiment tracking component relies entirely on ClearML's built-in tracking capabilities, requiring minimal custom instrumentation in training scripts.

4.6.1 Code and Configuration Versioning

ClearML automatically captures the Git context for every submitted task, including the commit hash, branch name, repository URL, and working directory state. This integration occurs during task creation when the submission script specifies the repository and branch parameters. Before execution on the remote machine, ClearML queries the Git repository to verify that all changes have been committed and pushed to the remote repository. This workflow enforces clean Git states: developers are expected to execute and make their changes before submitting tasks, ensuring that every experiment corresponds to a specific code version. This policy eliminates ambiguity about which code produced a particular result and ensures that experiments can be reproduced by checking out the recorded commit hash. The automatic Git integration addresses the reproducibility requirement (R2) by implementing the policy through system design.

Hyperparameters and training configuration are jointly managed through a tightly linked approach that combines version-controlled YAML configuration files with runtime parameter override mechanisms. The YAML files specify all critical aspects of the training setup—including dataset parameters, model architecture, optimizer settings, and training duration—and are tracked in Git alongside the source code, ensuring that any code commit is directly associated with the corresponding experiment configuration.

When a task is submitted, the submission script records the path to the specific configuration file in the task metadata, allowing the reconstruction of the

exact experimental setup for any past run. Importantly, the system also supports runtime parameter overrides using the `Task.set_parameters()` API. This allows developers to adjust specific configuration values on a per-experiment basis—such as for hyperparameter sweeps or quick testing—without modifying the main YAML template. ClearML automatically logs both the original configuration and any runtime overrides in each task's metadata, maintaining a comprehensive, unified record of the actual parameters used during training.

By linking versioned configuration files and dynamic parameter overrides within a single tracking workflow, the system supports reproducibility and flexibility: every experiment is fully reconstructible from the combination of code, configuration files, and recorded overrides, while developers retain the freedom to explore parameter variations efficiently.

All hyperparameters, whether defined in YAML or overridden at runtime, are automatically logged and displayed in the ClearML web interface. The interface provides structured parameter views that organize hyperparameters hierarchically (e.g., `data/batch_size`, `model/model_channels`), facilitating parameter comparison across experiments. Figure 4.7 shows an example of the ClearML configuration interface displaying the hierarchical organization of hyperparameters for a training experiment.

Figure 4.7: ClearML configuration tab showing the hyperparameter used in the training run.

4.6.2 Dataset Tracking

As described in Section 4.5, the dataset used in this work consists of weather reanalysis data from the High-Resolution Rapid Refresh (HRRR) dataset, stored as Zarr arrays in Scaleway Object Storage. The dataset covers a fixed time period (2017–2022) and spatial domain (Oklahoma region) and is not actively versioned, as it represents historical data that generally does not change. Instead of using

ClearML's Dataset API, which provides explicit versioning for evolving datasets, the system tracks dataset references through hyperparameters.

Each experiment's configuration records the following dataset information:

- S3 bucket name and key: Identifies the specific Zarr store (e.g., `data/oklahoma_hrrr.zarr`)
- Variable selection: Lists which meteorological variables are used as input channels (e.g., `['pwat', 't', 'q', 'pres']`)
- Temporal subset: Defines the train/validation/test year ranges
- Statistics file: References the normalization statistics computed from the training set

This information is sufficient to reproduce the exact dataset configuration used for training. A developer can retrieve the dataset by accessing the specified S3 bucket with the recorded credentials, loading the Zarr array, selecting the listed variables, and applying the normalization statistics. The deterministic nature of Zarr array slicing (when using consistent time ranges) ensures that the same data is loaded for reproduction attempts.

For future work involving dataset evolution (e.g., adding new time periods or spatial regions), the system could adopt ClearML's Dataset API, which provides version hashing and lineage tracking. However, for the current use case of training on fixed historical data, the simpler hyperparameter-based tracking is sufficient and reduces operational complexity.

4.6.3 Artifact Management

ClearML provides centralized artifact storage where training scripts can upload model checkpoints, visualizations, and analysis outputs. The system uses this capability to store intermediate and final model weights, enabling model reuse and reproduction.

The training script includes explicit checkpoint logic that uploads model artifacts at regular intervals:

- Periodic checkpoints: Saved every 50 training "ticks" (where each tick processes a fixed number of images)
- Best model checkpoints: The top-K models based on validation loss are retained (typically K=3)
- Latest checkpoints: The most recent N checkpoints are kept for resuming interrupted training (typically N=1)

These policies prevent unbounded checkpoint accumulation, which would otherwise consume significant storage. The cleanup logic runs within the training script, comparing new checkpoints against existing ones and deleting older artifacts that no longer meet retention criteria. Artifact deletion requires explicit API calls, as ClearML does not provide automatic retention policies.

Model checkpoints are stored in PyTorch's native `.pth` format, which is framework-standard and not proprietary to ClearML. Each checkpoint includes the

model state dict, optimizer state, EMA (exponential moving average) state, and the training step count, enabling resuming training from arbitrary checkpoints.

Artifact retrieval for inference or continued training uses ClearML's API:

```
from clearml import Task
import torch

task = Task.get_task(task_id="taskid")
task_models = task.get_models()
for output_model in task_models["output"]:
    checkpoint_id = output_model.id
    checkpoint_path = output_model.get_local_copy()
    checkpoint = torch.load(checkpoint_path)
```

Listing 4.4: Retrieving a trained model artifact

ClearML abstracts the storage backend (which may be the local filesystem, S3, or ClearML's cloud storage), providing a uniform API regardless of where artifacts physically reside. In this deployment, artifacts are stored in ClearML Cloud's managed storage, which incurs costs based on storage volume ($0,10 per GB per month) and API call frequency ($1 per 100K calls)[7]. This SaaS approach was chosen over self-hosted artifact storage (e.g., Scaleway Object Storage), eliminating the operational overhead of managing a separate storage service, configuring access permissions, and implementing backup strategies.

All console logs (stdout and stderr) are streamed to ClearML in real-time and retained indefinitely, providing a complete record of training execution, including any errors, warnings, or debug information emitted by the training script.

The experiment tracking component directly addresses requirements R2 and R3. The reproducibility requirement (R2) is ensured through Git commit tracking, configuration capture, dataset parameter recording, and artifact storage. It is possible to reproduce an experiment with that information. The experiment tracking requirement (R3) is implemented with a comprehensive logging infrastructure that captures extensive metadata, and system-level metrics are automatically recorded without code changes. In contrast, training-specific metrics require explicit, standard logging calls.

While sufficient for current needs, several limitations exist. The web interface provides limited query capabilities; finding experiments matching specific parameter combinations requires writing Python scripts to query the API programmatically. Artifact cleanup must be implemented manually in training scripts, as the platform lacks configurable retention policies. Finally, dataset tracking via hyperparameters is less robust than formal dataset versioning and would not scale to rapidly evolving data.

[7] Based on ClearML Cloud pricing as of Q4 2025.

4.7 Performance Monitoring

Effective performance monitoring provides visibility into system health, training progress, and resource consumption, enabling developers to identify issues quickly and optimize resource usage. This section describes the monitoring capabilities provided primarily through ClearML's built-in infrastructure, supplemented by Scaleway's billing dashboard for cost tracking.

4.7.1 Resource Utilization Tracking

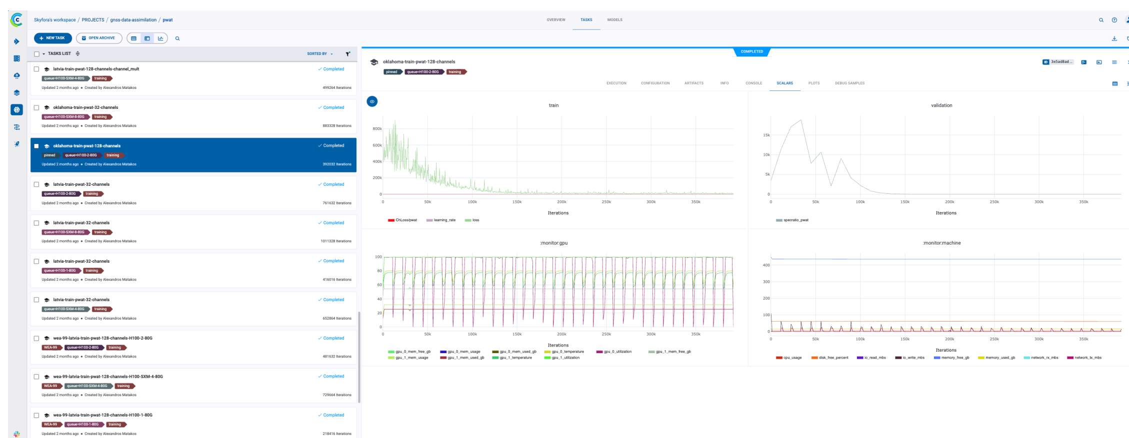

Figure 4.8: ClearML monitoring tools showing time-series plots of GPU utilization, GPU memory usage, CPU utilization, system memory consumption, disk I/O, and training loss for a typical experiment.

ClearML automatically captures system-level metrics from every agent executing training tasks. Figure 4.8 shows metrics collected by ClearML Agent from the host operating system.

The following resource metrics are tracked for each running task:

- GPU utilization: Percentage of time the GPU is actively executing kernels, sampled every few seconds
- GPU memory: Allocated and used VRAM in gigabytes, tracking memory consumption over time
- CPU utilization: Percentage utilization across all CPU cores
- System memory: RAM usage in gigabytes
- Disk I/O: Read and write throughput in MB/s, indicating dataset loading bottlenecks
- Network I/O: Network throughput, relevant for multi-node distributed training (though the current single-instance setup does not leverage this)

These metrics are visualized as time-series plots in the ClearML web interface, accessible through the task's "Scalars" tab. Developers can observe resource utilization patterns to diagnose issues such as GPU underutilization (indicating data-loading bottlenecks) or memory pressure (indicating large batch sizes or complex model architectures). The automatic capture of these metrics addresses

the performance monitoring requirement (R6) without requiring developers to instrument monitoring code, avoiding the complexity and maintenance burden of custom monitoring solutions like Prometheus or Grafana.

For multi-GPU training, PyTorch's distributed data parallelism achieves near-linear scaling across GPUs. Monitoring confirms that GPU utilization remains consistently high (typically 85–95%) across all GPUs when data loading is configured correctly, validating the scaling requirement (R7). This visibility is critical for justifying the cost of multi-GPU instances: if scaling efficiency were poor, it would be more cost-effective to use single-GPU instances, but the monitoring data confirms near-optimal scaling (190% speedup with 2 GPUs).

4.7.2 Training Progress Monitoring

Training-specific metrics are logged explicitly within the training script using ClearML's Python API. The diffusion model training loop reports metrics at the granularity of individual training batches, providing fine-grained visibility into convergence behavior.

The primary training metric is the denoising loss, which quantifies the model's ability to predict the noise added to input images at various diffusion timesteps. The training script logs this metric using:

```python
from clearml import Logger

logger = Logger.current_logger()
logger.report_scalar(
    title="Training Loss",
    series="loss",
    value=loss.item(),
    iteration=step
)
```

Listing 4.5: Logging training metrics in ClearML

Additional logged metrics include:

- Validation loss: Computed periodically (every 50 ticks) on held-out data to track generalization
- Learning rate: Tracked to verify learning rate scheduling
- Gradient norms: Monitored to detect gradient instabilities or vanishing gradients
- EMA decay: Tracks the exponential moving average coefficient used for model parameter smoothing

ClearML's web interface presents these metrics as interactive plots where developers can zoom, pan, and compare across experiments. The interface supports overlaying multiple experiments on the same plot, enabling direct comparison of training curves across hyperparameter variations.

Early stopping logic is implemented based on validation loss, with a configurable patience parameter (typically 10–20 validation intervals). If validation loss fails to improve for the patience duration, training terminates automatically, and the task status is marked as ”completed.” This mechanism prevents wasteful computation on experiments that have plateaued by avoiding unnecessary GPU hours (estimated savings of 10–20% on total compute costs based on observed convergence patterns). Without early stopping, developers would need to manually monitor experiments and cancel them, which is impractical for overnight or weekend runs and leads to substantial resource waste.

The real-time nature of metric streaming allows developers to monitor training progress without SSH-ing into GPU instances. The ClearML mobile app extends this capability, enabling monitoring on smartphones and supporting iterative development workflows in which developers review results during commutes or off-hours.

4.7.3 Cost Visibility

Cost tracking in the current system is manual and approximate, as ClearML does not natively integrate with cloud provider billing APIs. However, the information needed to compute costs is available from experiment metadata:

- Queue name: Indicates the instance type used (e.g., `H100-2-80G`)
- Training duration: Wall-clock time from task start to completion, recorded in task metadata
- Instance pricing: Scaleway publishes hourly rates for each instance type (e.g., €2,73/hour for H100-1-80G)[8]

Developers can manually compute experiment costs using the formula:

$$\text{Cost} = \frac{\text{Duration (seconds)}}{3600} \times \text{Hourly Rate}$$

For example, a 10-hour training run on H100-2-80G costs approximately €27.30. While this calculation is straightforward, it requires manual effort and does not provide aggregate views across multiple experiments or time periods.

Scaleway's billing dashboard provides monthly cost summaries aggregated across all resources, which the team reviews weekly. Billing alerts are configured at the Scaleway account level to trigger email notifications when monthly spending exceeds predefined thresholds. These alerts provide fairly simple cost control at the project level. A future enhancement would implement an automated weekly report of a detailed cost breakdown, grouped by experiment, subproject, or user.

4.7.4 Team Visibility and Orchestration

ClearML's ”Orchestration” page, illustrated in Figure 4.9, provides a centralized view of the distributed execution environment. The worker overview shown in

[8] Based on Scaleway public documentation and pricing as of Q4 2025.

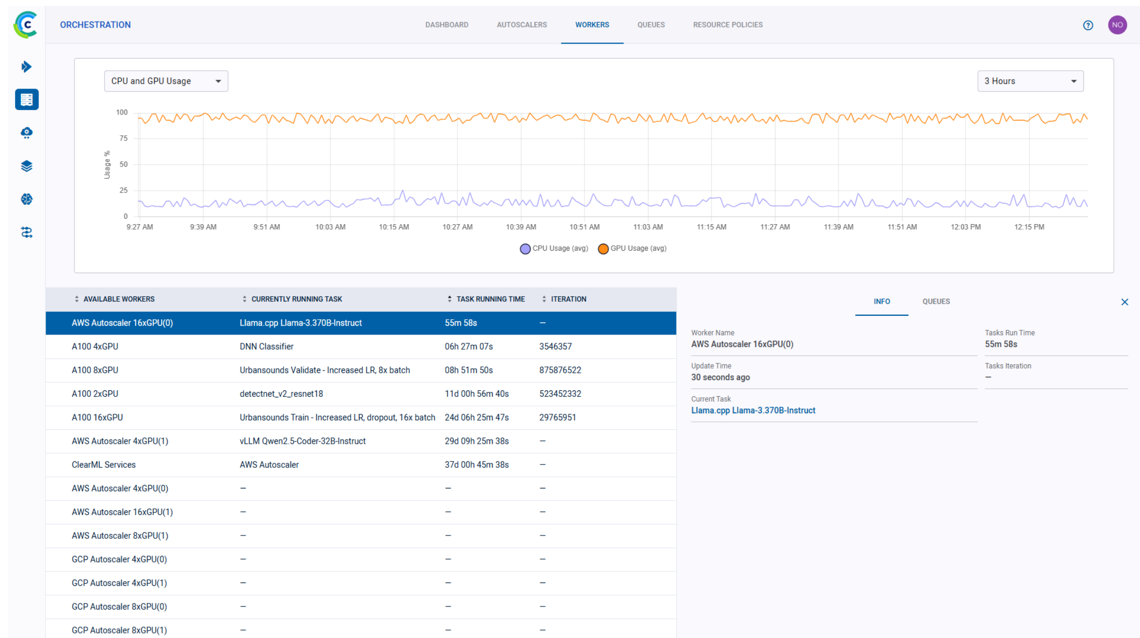

Figure 4.9: ClearML worker monitoring view showing the list of active agents, current tasks being processed, and live GPU/CPU/memory utilization per agent.

this figure displays all active agents, including their assigned queues, the specific tasks they are currently processing, and detailed real-time resource utilization (e.g., GPU, CPU, and memory usage) for each agent.

- Active agents: Lists all registered agents, their assigned queues, and current task status
- Queue status: Shows the number of pending, running, and recently completed tasks per queue
- Agent resource utilization: Displays real-time GPU, CPU, and memory usage for each agent

This view enables a team lead or system administrator to assess system health quickly. For example, if tasks are accumulating in queues without progressing, the orchestration page reveals whether agents are offline, stuck, or simply overloaded. If an agent consistently reports low GPU utilization, it may indicate a data-loading bottleneck that requires attention.

The project dashboard (Figure 4.10) consolidates high-level statistics across all experiments within a project. This page in the ClearML web interface offers an at-a-glance summary, including:

- Total number of experiments run
- Success and failure rates
- Total experiment duration

By viewing this dashboard, team members can track project progress, monitor experiment throughput, and quickly assess the health of ongoing research efforts. The dashboard is accessed directly from the ClearML project page, providing a centralized entry point for research planning and retrospective analysis.

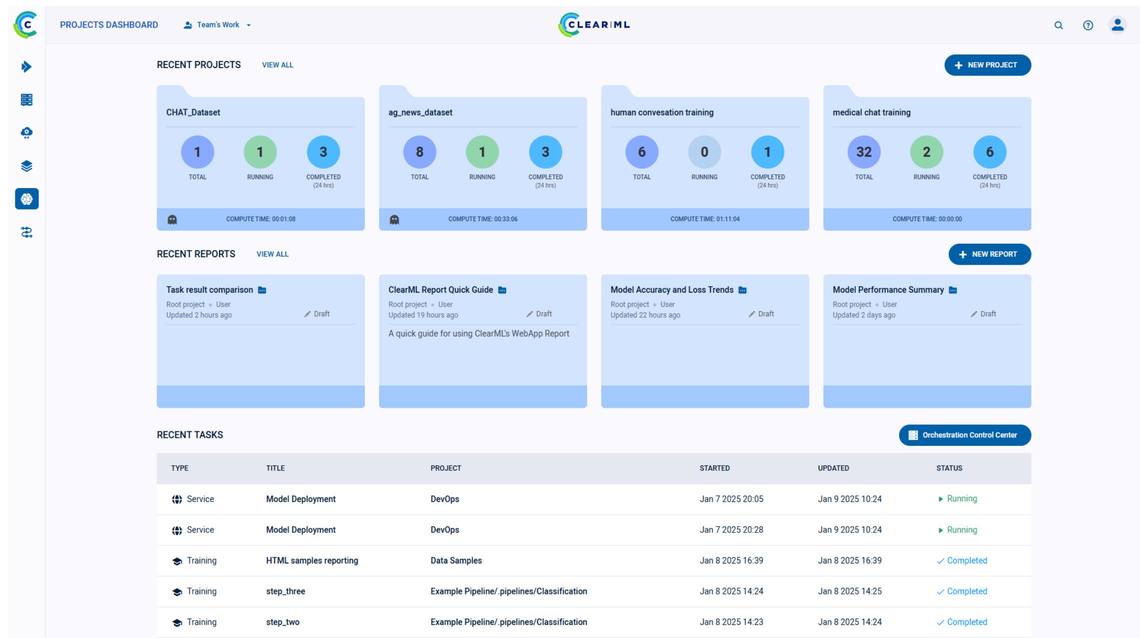

Figure 4.10: ClearML project dashboard page showing project-wide experiment statistics and summary analytics.

This consolidated view helps teams identify promising research directions, allocate resources, and avoid repeating unsuccessful experiment configurations.

The performance monitoring component addresses the monitoring requirement with minimal implementation effort by leveraging built-in capabilities. Resource monitoring is fully automated, requiring no code changes and providing comprehensive visibility without the need to learn new monitoring tools or write instrumentation code. Training progress monitoring requires explicit logging calls but follows standard patterns, with the training script using a lightweight abstraction (`Logger.report_scalar`) that integrates naturally into existing training loops, typically adding 3–5 lines of code per tracked metric while avoiding more complex monitoring frameworks. Cost visibility is the weakest aspect of the monitoring design, relying on manual calculations and Scaleway's account-level billing. While sufficient for the current small team, this approach would not scale well with multiple projects/teams involved. This limitation is acknowledged as a trade-off prioritizing rapid implementation over comprehensive cost accounting, reflecting the system's current focus on level 2 maturity, where cost optimization is important but not the primary operational concern. As the system evolves toward Level 3–4 maturity, implementing automated cost tracking will become a priority.

5 Evaluation & Results

This chapter presents the evaluation results of the MLOps pipeline against the objectives defined in Section 3.3. Following the evaluation methodology outlined in Section 3.4, each objective is assessed through specific evaluation episodes (Ep1-Ep6) that measure concrete properties of the implemented system. The evaluation draws on data collected over a three-month deployment period, during which the ML engineering team executed around 300 experiments.

5.1 Easy Adoption (O1)

The ease of adoption objective (O1) aims to minimize the learning curve and integration complexity for teams transitioning to the MLOps pipeline. This section evaluates adoption barriers through two properties: the number of tools required and the proportion of ready-made components versus custom development.

5.1.1 Ep1: Number of Tools

Research by Symeonidis et al. [30] demonstrates that minimizing the number of tools in MLOps pipelines directly supports easier adoption by reducing integration complexity and lowering the barrier to entry for teams [30]. The tool inventory for the implemented architecture comprises 7 core technologies that developers and infrastructure maintainers interact with. Table 5.1 presents a detailed breakdown of each tool, its purpose within the architecture, the MLOps components it implements (as defined in Section 2.2), estimated learning time based on the team's prior experience, and the experience level.

The tool stack is built primarily on technologies already familiar to the team, including Git, Python, Docker, Golang, and Terraform. Thus, most ML engineers face virtually no learning curve. Only two new tools need to be adopted: ClearML for experiment tracking & orchestration and Scaleway for cloud GPU provisioning. For ML engineers, day-to-day interaction is limited to Git, Python, and ClearML, requiring less than an hour of self-study before running experiments, as features like logging and experiment tracking align with their previous experience.

Infrastructure maintainers bear the main adoption overhead, needing to learn ClearML's API for integration (about 4 hours) and Scaleway's GPU and API ecosystem (about 10 hours), totaling an estimated 14 hours for complete onboarding. This upfront investment is essentially a one-time cost, making future extensions—such as adding support for more cloud providers—much simpler.

Compared to alternative MLOps stacks like Kubeflow or AWS SageMaker, which require 8 or more tools and introduce substantial operational complexity (e.g., Kubernetes orchestration and AWS-specific concepts), this architecture minimizes both the number of tools and cognitive load. By consolidating four MLOps components into ClearML and encapsulating cloud automation in a custom resource

Table 5.1: Tool inventory and learning requirements mapped to MLOps components

Tool	Purpose	Component	Learning Time	Prior Exp.
Git	Version control for code and configuration	C2	0 hours	High
Python	Training scripts, task submission, data processing	–	0 hours	High
Docker	Container runtime for reproducible environments	–	0 hours	High
Golang	Resource manager implementation (infra team only)	C5	0 hours	High
Terraform	Infrastructure as code for secrets and container deployment	–	0 hours	High
ClearML	Experiment tracking, task orchestration, artifact storage	C3, C6, C7, C9	1–4 hours	None
Scaleway	GPU instance provisioning, object storage (S3)	C5	10 hours	None
	Total (ML Engineer)	**3 tools**	**~1 hours**	
	Total (Infra)	**7 tools**	**14 hours**	

manager, developers can onboard rapidly while infrastructure complexity is contained within the infrastructure team.

5.1.2 Ep2: Ready-made Components

The architecture's reliance on ready-made components directly impacts adoption complexity, as off-the-shelf software reduces the debugging burden and provides established documentation and community support. A percentage-based metric for evaluating MLOps system architecture quality assesses the proportion of ready-made versus custom components, with higher percentages of ready-made components indicating lower adoption barriers and reduced maintenance overhead. Table 5.2 categorizes the MLOps technical components by their implementation approach, as defined in Section 2.2.

The architecture implements six of the nine MLOps technical components de-

Table 5.2: MLOps component implementation breakdown

Component	Implementation	Type
C2 Source Code Repository	Git (GitHub/GitLab)	Ready-made
C3 Workflow Orchestration Component	ClearML Task Queue + Agent	Ready-made
C5 Model Training Infrastructure	Scaleway GPU Instances + Custom Resource Manager	Mixed
C6 Model Registry	ClearML Artifact Storage + Model Storage	Ready-made
C7 ML Metadata Stores	ClearML Experiment Tracking	Ready-made
C9 Monitoring Component	ClearML Web UI	Ready-made
Ready-made	**5 components (83%)**	
Mixed (partial custom)	**1 component (17%)**	

fined in Section 2.2. The three excluded components—C1 (CI/CD Component), C4 (Feature Store System), and C8 (Model Serving Component)—are intentionally out of scope for Level 2 maturity as discussed in Section 4.2. Of the six implemented components, five (83%) rely entirely on ready-made solutions, while only C5 (Model Training Infrastructure) requires partial custom development.

The Model Training Infrastructure (C5) is implemented as a mixed solution: the Scaleway GPU Instance API provides ready-made compute provisioning capabilities, while dynamic resource management requires custom development. This custom automation comprises a Golang service that monitors ClearML task queues, orchestrates instance provisioning/deprovisioning, manages Terraform configurations for infrastructure-as-code deployment, and runs cloud-init scripts for automated agent installation. This represents the primary custom development burden unique to the architecture: implementing scaling logic tailored to cost-optimization requirements (R1). While the individual technologies (Golang, Terraform, bash) are standard, their orchestration into a cohesive resource management system requires custom implementation.

Notably absent from this breakdown are the training scripts themselves, which represent domain-specific research code required regardless of MLOps platform choice and thus do not constitute MLOps infrastructure overhead. The 83% ready-made proportion among implemented components indicates that only one component required partial custom implementation, demonstrating effective leverage of existing platforms with minimal custom integration code.

In summary, the evaluation of objective O1 finds that the architecture achieves low adoption complexity through two key strategies: (1) consolidating four MLOps components (C3, C6, C7, C9) into a single platform (ClearML), reducing the tool

count to 7 with only 2 requiring new learning, and (2) maximizing the use of ready-made implementations (83% of the 6 implemented components). By leveraging standard development tools already familiar to the team, with only ClearML and Scaleway requiring focused training for infrastructure maintainers, adoption barriers are minimized for ML engineers, who can begin submitting experiments with almost no platform-specific learning. Infrastructure complexity is consolidated within the resource manager (partial custom development in C5) rather than distributed across developer workflows. While an initial investment in resource manager development and API learning is required, ongoing onboarding is fast and straightforward, confirming that the architecture is well-suited for small research teams and is extensible for future needs.

5.2 Cost Effective (O2)

The cost-effectiveness objective (O2) evaluates whether the architecture achieves efficient resource utilization through automated cost controls. This evaluation assesses operational costs, operational overhead, and the effectiveness of the dynamic provisioning policy.

5.2.1 Ep3: Operational Cost

Operational costs represent the direct expenditure on GPU compute resources. The primary cost driver is Scaleway H100 GPU instances, with hourly rates varying by configuration: €2,73/hour for H100-1-80G, €5,46/hour for H100-2-80G, €11,11/hour for H100-SXM-4-80G, and €22,57/hour for H100-SXM-8-80G[1]. The majority of experiments use H100-2-80G instances, which offer optimal cost-performance for data-parallel training workloads.

Table 5.3 presents observed resource utilization over a representative one-month period (720 hours). The table compares actual instance usage against a baseline always-on scenario with 5 instances running continuously to handle peak concurrent load. This baseline represents 3,600 potential instance-hours per month (5 instances x 720 hours). The breakdown shows active training hours, idle hours (instances provisioned but not executing tasks), and off hours (capacity not utilized because instances were deprovisioned).

Table 5.3: GPU resource utilization over one-month evaluation period

Instance State	Hours	Percentage	Cost (H100-2-80G)
Active (training)	900	25,0%	€4.914
Idle (provisioned, waiting)	50	1,4%	€273
Off (deprovisioned)	2.650	73,6%	€0
Total (5-instance baseline)	**3.600**	**100%**	**€5.187**

[1] Based on Scaleway public documentation and pricing as of Q4 2025.

The workload pattern observed over three months consists of approximately 100 experiments per month, with each experiment running 6–12 hours depending on model size and convergence. Experiments are primarily executed on weekdays, with 3–5 experiments running concurrently during peak hours. Each experiment runs on a dedicated H100-2-80G instance. This translates to approximately 900 instance-hours per month of active training (100 experiments x 9 hours average). The dynamic provisioning system maintained instances in an idle state for approximately 50 hours per month, representing the buffer period between task completion and shutdown. Each experiment incurs roughly 30 minutes of idle time: a 30 minute timeout window before automatic shutdown, in addition 2–4 minutes of initialization time for newly provisioned instances.

To quantify cost savings, we compare the actual cost against the always-on baseline scenario where 5 H100-2-80G instances run continuously to handle peak concurrent load. In this baseline scenario, the 5 instances would provide 3.600 instance-hours of capacity per month (5 × 720 hours), but only 950 hours (26,4%) would actually be utilized (900 hours of active training + 50 hours idle):

$$\text{Always-on cost (5 instances)} = 5 \times 720 \text{ hours} \times €5,46/\text{hour} = €19.656,00$$
$$\text{Dynamic provisioning cost} = 950 \text{ hours} \times €5.46/\text{hour} = €5.187,00$$
$$\text{Cost savings} = \frac{€19.656,00 - €5.187,00}{€19.656,00} \times 100\% = 73,6\%$$

The evaluation shows that implementing dynamic provisioning cuts operational costs by 73,6%, primarily by deprovisioning GPU resources when they are not needed—such as during nights, weekends, code reviews, and data preparation periods. By running instances only when active experiments are queued and automatically scaling to match concurrent demand (3–5 instances during peak periods, 0 during off-hours), the system avoids unnecessary spending typical of an always-on setup.

While this approach introduces a short provisioning latency (about 3–5 minutes when starting a new job), the impact is minimal compared to the duration of most training runs. Additionally, the dynamic system reduces wasted costs from failed or canceled experiments, as unused instances shut down automatically after a short idle period. Overall, dynamic resource management directly supports cost efficiency and keeps performance trade-offs at an acceptable level for research workloads.

5.3 Extendable (O3)

The extensibility objective (O3) evaluates whether the architecture can evolve to support higher MLOps maturity levels and adapt to changing requirements without major redesign. This assessment examines the current maturity level and the modularity of component design.

5.3.1 Ep4: Pipeline Maturity level

The MLOps maturity of this system is assessed using the Microsoft MLOps Maturity Model (see Table 2.3), which defines five levels of capability from no MLOps (level 0) to full MLOps automation (level 4). The implemented system achieves level 2 maturity (Automated Training), which is appropriate for a project that is in the proof-of-concept phase. This section examines how the architecture fulfills each level 2 capability requirement:

- Automated training pipeline: The setup with the ClearML queue system and ClearML agent enables automation for developers, eliminating the manual overhead of SSH-ing into machines, setting up environments, and launching training runs. This process effectively cuts the 15–30 minutes of work required to submit a training job down to a few seconds.
- Centralized experiment tracking: ClearML automatically captures comprehensive metadata for every training run, including hyperparameters, metrics, system performance, Git commit, Python environment, and console output. The web UI enables developers to inspect and compare the training results.
- Model management: Trained model checkpoints are tracked within ClearML, with version tracking tied to the experiment that produced them. The model release still requires some manual effort in the inference service.
- Reproducible training environments: The combination of Git versioning (code), container images (environment), and hyperparameters tracked by ClearML provides full reproducibility.
- Compute managed: The resource manager component abstracts the required hardware to run the training job, eliminating manual instance management.

The system intentionally does not implement level 3 capabilities because models are not yet deployed as production services. Currently, inference is performed as one-off batch processing for analysis rather than serving live requests. Level 3 addresses operational concerns that only become relevant when models run continuously in production environments.

However, the architecture can evolve to level 3 when production deployment becomes necessary. The key additions required are:

- Automated data pipeline: Implement continuous data ingestion and preprocessing that automatically adds new data to the training dataset, triggering retraining workflows when sufficient new data accumulates.
- Model validation and unit testing: Build unit tests and integration tests for each trained model, providing benchmarks before release approval.
- CI/CD for model deployment: Create deployment pipelines that automatically release validated models to staging and production environments, with rollback capabilities if issues are detected.

These additional features are feasible to build given the current system's foundation. ClearML's task graph capabilities support automated workflows. This shows

how the current system's architecture provides extensibility as new requirements emerge.

5.3.2 Ep5: Modularity

Modularity determines how easily individual components can be replaced or upgraded without significant infrastructure changes. This evaluation focuses on the two critical external dependencies: ClearML (experiment tracking and orchestration) and Scaleway (GPU infrastructure).

Replacing ClearML with an alternative such as MLflow or Weights & Biases would require modifying training scripts to use different logging APIs, updating the task submission script, and adapting the resource manager's queue-monitoring logic. Although it affects many parts of the system, the alternatives follow the same paradigm of a queue system and an agent. This allows for replacing ClearML with other options in 1–2 days with minimal changes, thus avoiding vendor lock-in.

The resource manager's design intentionally abstracts the GPU instance provider. Given ClearML Agent's capabilities, replacing Scaleway with another GPU provider would only require changes to the resource manager's API integration, not the system's core logic. This allows the ML developer to avoid being concerned about the changes.

The evaluation shows that the architecture successfully balances simplicity to meet current requirements with future extensibility. The level 2 MLOps maturity provides the necessary systems needed for research workloads for a smaller team. Extending to level 3 maturity would require adding automated data pipelines, model validation, and CI/CD for deployment—all feasible additions that build on the existing foundation without architectural changes.

Another highlight is the architecture's modularity, which allows for minimal effort when the requirements change. As a result, the architecture delivers simplicity and long-term extensibility.

5.4 Ease of Use (O4)

The ease-of-use objective (O4) evaluates whether the system provides an intuitive, low-friction experience for daily operations. This assessment measures the time required to complete training workflows.

5.4.1 Ep6: Task Completion Time

Task completion time quantifies the developer experience by measuring how long it takes to complete a standard training workflow, from code changes to result retrieval. Table 5.4 breaks down the workflow into discrete steps, showing both the one-time setup requirements and the typical daily use case completion time.

The one-time onboarding process takes approximately **32 minutes** to complete: cloning the repository (2 minutes), configuring ClearML credentials (10 minutes),

Table 5.4: Task completion time for standard training workflow

Task Step	**Time**
One-time Setup (Before Onboarding)	
Clone repository	2 min
Configure ClearML credentials	10 min
Configure Git credentials	10 min
Learn submission workflow	10 min
Subtotal (before onboarding)	**32 min**
typical usecase (after onboarding)	
Commit and push code changes	1 min
Run submission script	1 min
Monitor training progress	1 min
Retrieve results/artifacts	1 min
Subtotal (after onboarding)	**4 min**

Note: One-time setup tasks are completed only once before onboarding.

configuring Git credentials (10 minutes), and learning the submission workflow (10 minutes). Credential setup involves creating a ClearML API key through the web interface and configuring the `clearml.conf` file, while the learning phase involves understanding the queue system and familiarizing oneself with the submission script's command-line arguments.

After onboarding, the typical use case workflow requires only **4 minutes** to complete: committing code changes (1 minute), running the submission script (1 minute), briefly verifying that training begins correctly (1 minute), and reviewing results (1 minute). This 4-minute duration demonstrates the system's ease of use, as it is comparable to the time required to initiate training on a local workstation, representing minimal friction for daily operations.

Compared to the previous baseline workflow used by the company, which involved manually SSH into provisioned GPU instances, starting screen sessions, launching training scripts, and periodically reconnecting to check progress, the automated system reduces task completion time from approximately 10–20 minutes per experiment to just 4 minutes.

The evaluation shows that the developed system is intuitive and does not introduce friction for daily operations. The solution improves task completion time compared to the previous workflow in the company's.

6 Discussion

This chapter interprets the evaluation results presented in Chapter 5, analyzing them within the context of the research question and the broader objectives defined in Chapter 3. Looking at the strategic trade-offs between cost and complexity, the authors discuss the implications for scientific machine learning workflows, acknowledge the limitations of the implemented architecture, and outline directions for future research.

Table 6.1: Summary of key findings and implications

Objective	Evaluation Episode	Key Finding
O1: Easy Adoption	Ep1: Number of Tools	Developer interface consolidated to **three tools** (Git, Python, ClearML)
	Ep2: Ready-made Components	Only **17%** custom development (Resource Manager), **83%** off-the-shelf components
O2: Cost Effective	Ep3: Operational Cost	**76,9%** reduction in operational costs compared to static baseline through dynamic provisioning
O3: Extendable	Ep4: Pipeline Maturity level	Achieved MLOps maturity **level 2** with foundation for level 3 capabilities
	Ep5: Modularity	Components can be replaced with minimal changes (1–2 days for ClearML, minimal changes for Scaleway)
O4: Ease of Use	Ep6: Task Completion Time	Training submission time reduced from **20–30 minutes** to **4 minutes**

6.1 Interpretation of Results

The central research question (RQ1) of this thesis asked: *How can we design and implement a self-managed MLOps pipeline that supports scalable, reproducible, and cost-efficient training of diffusion models for scientific ML use cases using open-source tooling and self-managed GPU machines*? The evaluation results demon-

strate that the proposed architecture successfully answers this question by integrating lightweight open-source orchestration with custom infrastructure management. Table 6.1 summarizes the key findings for each objective.

In achieving the cost efficiency objective (O2), the system used a dynamic provisioning mechanism, resulting in a **76,9%** reduction in operational costs compared to a static, always-on baseline. This finding validates that the custom resource manager effectively reduces operational costs while integrating seamlessly into the overall solution.

Regarding ease of adoption (O1) and ease of use (O4), consolidating the developer interface into **three tools** (Git, Python, and ClearML) enables a typical training workflow that requires only **4 minutes** from code changes to result retrieval. This minimal time requirement demonstrates the system's ease of use and would result in a faster development cycle and output for the team.

Finally, the evaluation confirmed the system's extendability (O3). By achieving MLOps maturity **level 2** (Automated Training) and maintaining loose coupling between the orchestration layer (ClearML) and the compute provider (Scaleway), the architecture provides a robust foundation that can evolve to support level 3 capabilities, such as continuous deployment, without requiring a fundamental redesign.

6.2 Comparison with State of the Art

The MLOps landscape currently offers two main paths: First, enterprise-grade platforms like Smartflow and H&M use Kubernetes orchestration to manage organization-wide ML operations at enterprise scale. However, they demand dedicated platform engineering teams. Second, research-oriented approaches like Ruggeri et al.'s [26] agronomy system use lighter-weight tools (such as ClearML) for experiment tracking, but require manual setup and management of computing resources.

The architecture presented in this thesis occupies the middle ground between these extremes, addressing the gap identified in Section 2.4 for small organizations that need automated resource management without the complexity of production-level orchestration tools like Kubernetes. The implemented system demonstrates that for research purposes, complex orchestration tools can be effectively replaced with simpler automation that delivers substantial cost savings with minimal implementation overhead.

6.3 Practical Implications

The findings of this thesis have several practical implications for organizations considering MLOps adoption strategies. First, the results show an alternative to the de facto tools of choice for compute orchestration, Kubernetes. While Kubernetes excels at managing complex, multi-service applications with sophisticated networking requirements [24], the operational overhead—cluster configuration,

security management, service orchestration, and ongoing maintenance—may exceed the actual requirements for teams with intermittent training workloads. The evaluation demonstrates that minimal custom development can have a meaningful impact when strategically applied. The Resource Manager's implementation required approximately 2.548 lines of Go code, representing a reasonably simple approach to fulfill the orchestration requirements.

Second, the cost efficiency results have direct implications for research organizations operating under budget constraints. The **76.9%** cost reduction achieved through dynamic provisioning demonstrates that significant operational savings are accessible without enterprise-scale infrastructure investments. This finding is particularly relevant given the increasing computational demands of modern ML research [21] and the economic pressures facing academic and small-scale research organizations. The architecture shows that cost optimization does not require sophisticated auto-scaling systems or complex resource schedulers; a simple state machine that manages the instance lifecycle based on queue depth can deliver substantial savings.

A recurring theme in the MLOps literature is the trade-off between the flexibility of custom solutions and the ease of use of managed platforms such as Amazon SageMaker. The architecture developed in this thesis suggests a middle solution. By leveraging managed orchestration tools (ClearML) for experiment tracking and workflow coordination, while maintaining control over compute infrastructure through lightweight custom automation. This hybrid approach delivers cost efficiency by loading data directly from Scaleway Object Storage without local caching, avoiding the potential vendor lock-in associated with comprehensive managed services like SageMaker. While SageMaker offers a complete ecosystem and reduces operational burden, the approach presented here enables granular control over infrastructure and cost optimization, making it a viable alternative for teams with limited budgets and a preference for open-source tooling.

6.4 Limitations

While the architecture achieves its goals within the current research and organizational context, it still has specific limitations in its use. Firstly, the current pipeline design assumes the use of static, unchanging datasets. It does not include mechanisms for formal dataset tracking or management of evolving data. The system does not track or version any changes made to the dataset. This means that changes to the data over time are not recorded, so experiments may not be exactly reproducible if the dataset changes. This approach works adequately for stable historical datasets, but it does not support scenarios where datasets may change, grow, or require versioning throughout the machine learning lifecycle. In addition, operational visibility also presents some challenges. Although the system optimizes resource usage, it relies on manual calculations for cost tracking rather than providing granular and automated reporting at the experiment level.

Data I/O performance represents another limitation. While the cloud-native Zarr access pattern eliminates the need for local dataset copies, it introduces network latency for each chunk request. During training, data is loaded directly from Scale-

way Object Storage without local caching, so frequently accessed chunks must be re-downloaded for each experiment. This design prioritizes simplicity and storage efficiency over I/O performance, which is acceptable for the current workload but could become a bottleneck as training scales to larger datasets or more frequent experiments. Additionally, the ephemeral nature of dynamically provisioned instances means that any local caching strategy must account for instance termination, requiring persistent cache storage separate from the instance's local disk to avoid cache invalidation when machines are deprovisioned.

6.5 Future Work

The architecture presented in this thesis lays a foundation for several future research directions, prioritized by their potential impact on system capabilities and cost efficiency. Although this study demonstrated the feasibility of a cost-effective, self-managed MLOps pipeline for scientific machine learning, further research is needed to generalize and extend these findings.

One promising area for future work is the investigation of data versioning strategies specifically optimized for large-scale scientific datasets in cloud environments. As discussed in Section 4.5, the current system uses hyperparameter-based tracking for dataset configuration, which works with immutable historical datasets but becomes lacking when requirements evolve. Future research could explore how to integrate tools such as DVC or ClearML's Dataset API to manage more complex datasets. This would enable the system to handle scenarios where datasets change in preprocessing pipelines.

Another critical area for improvement is optimizing data I/O performance through intelligent local caching strategies. While the current cloud-native approach eliminates local storage requirements, it introduces network latency that could be mitigated by implementing a persistent cache layer. Future work could investigate hybrid caching architectures that store frequently accessed Zarr chunks on persistent volumes (e.g., Scaleway Block Storage) attached to GPU instances. This would require careful coordination between the resource manager and the data loading pipeline to ensure that cached data persists across instance lifecycle events (provisioning, termination, and reprovisioning).

Finally, extending this architecture to support automated continuous deployment (MLOps level 3) in self-managed environments presents significant research opportunities. Future work could investigate how to automate the transition from training to inference without relying on managed services. This includes researching standardized patterns for packaging, validating, and deploying models to custom inference servers, effectively closing the loop between experimentation and production while maintaining the system's low-cost profile.

7 Conclusion

This thesis addresses the challenge of creating scalable, reproducible, and cost-efficient machine learning workflows for scientific applications on self-managed infrastructure. Motivated by the need to train diffusion models for weather forecasting without the prohibitive costs and vendor lock-in of managed platforms, the researchers designed and implemented a custom MLOps architecture. By integrating ClearML for orchestration and experiment tracking with a custom Go-based resource manager for Scaleway GPU instances, the system occupies a middle ground between enterprise-grade platforms and manually managed research infrastructure.

The evaluation results confirm the effectiveness of the proposed architecture across all four research objectives. The system achieved a **76.9% reduction in operational costs** through dynamic resource provisioning, through dynamic resource provisioning, while consolidating the developer interface into **three tools** (Git, Python, and ClearML) that enable a typical training workflow requiring only **4 minutes** from code changes to result retrieval. The solution successfully attained **MLOps Maturity Level 2** (Automated Training) with a foundation for level 3 capabilities, demonstrating that small teams can build robust, reproducible pipelines using predominantly off-the-shelf components. These findings demonstrate that for resource-constrained organizations, a hybrid approach combining lightweight orchestration with targeted custom infrastructure automation offers a viable alternative to both complex commercial MLOps suites and fully manual infrastructure management.

This work contributes to the MLOps literature by addressing a previously under-explored architectural space between Kubernetes-based enterprise architecture [9], [20] and research systems with manually managed infrastructure [26]. The architecture demonstrates that automated resource management is achievable without Kubernetes-level complexity, challenging the assumption that sophisticated orchestration tools are necessary for all MLOps deployments. By implementing six of the nine technical components in Kreuzberger et al.'s MLOps framework [17] while strategically omitting production-focused components, the system achieves research-appropriate MLOps maturity. The custom Resource Manager, implemented in approximately **2.548 lines** of Go code, demonstrating the value of selective customization for intermittent training workloads. This finding is largely absent from existing MLOps literature, which predominantly focuses on continuous, production-scale operations with always-on infrastructure.

Despite these successes, the system has limitations that point towards future research directions. The proposed architecture assumes static datasets without a versioning system, and the data access pattern could introduce a bottleneck due to network latency during training. Future work should focus on improving reproducibility through data versioning, optimizing data access patterns, and extending the architecture to support continuous deployment (MLOps level 3).

This thesis provides a reference implementation and design principles for scientific MLOps that challenge conventional assumptions about infrastructure requirements, demonstrating that small research teams can build effective MLOps pipelines with minimal custom development and the strategic use of open-source tools, ultimately enabling researchers to focus on scientific discovery rather than infrastructure management.

References

[1] R. Abernathey, N. L. Henderson, R. Seager, M. K. Tippett, and C. Lepore, “Collaborative proposal: EarthCube integration: Pangeo: An open source big data climate science platform,” *NSF Award*, vol. 17, p. 40 648, Sep. 1, 2017, ADS Bibcode: 2017nsf....1740648A. Accessed: Dec. 16, 2025. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2017nsf....1740648A.

[2] S. Amershi et al., “Software engineering for machine learning: A case study,” in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, May 2019, pp. 291–300. DOI: 10.1109/ICSE-SEIP.2019.00042. Accessed: Dec. 14, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/8804457.

[3] Y. Bahri, E. Dyer, J. Kaplan, J. Lee, and U. Sharma, “Explaining neural scaling laws,” *Proceedings of the National Academy of Sciences*, vol. 121, no. 27, e2311878121, Jul. 2, 2024, ISSN: 0027-8424, 1091-6490. DOI: 10.1073/pnas.2311878121. Accessed: Dec. 7, 2025. [Online]. Available: https://pnas.org/doi/10.1073/pnas.2311878121.

[4] M. Bhawsar, V. Tewari, and P. Khare, “A survey of weather forecasting based on machine learning and deep learning techniques,” *International Journal of Emerging Trends in Engineering Research*, vol. 9, no. 7, pp. 988–993, Jul. 5, 2021, ISSN: 23473983. DOI: 10.30534/ijeter/2021/24972021. Accessed: Jun. 21, 2025. [Online]. Available: http://www.warse.org/IJETER/static/pdf/file/ijeter24972021.pdf.

[5] H. B. Bluestein, F. H. Carr, and S. J. Goodman, “Atmospheric observations of weather and climate,” *Atmosphere-Ocean*, vol. 60, no. 3, pp. 149–187, Aug. 8, 2022, ISSN: 0705-5900. DOI: 10.1080/07055900.2022.2082369. Accessed: Aug. 14, 2025.

[6] J. A. Brotzge et al., “Challenges and opportunities in numerical weather prediction,” Accessed: Jun. 21, 2025. [Online]. Available: https://repository.library.noaa.gov/view/noaa/52714.

[7] C. O. de Burgh-Day and T. Leeuwenburg, “Machine learning for numerical weather and climate modelling: A review,” *Geoscientific Model Development*, vol. 16, no. 22, pp. 6433–6477, Nov. 14, 2023, Publisher: Copernicus GmbH, ISSN: 1991-959X. DOI: 10.5194/gmd-16-6433-2023. Accessed: Jun. 21, 2025. [Online]. Available: https://gmd.copernicus.org/articles/16/6433/2023/.

[8] *ClearML - your entire MLOps stack in one open-source tool*, original-date: 2019-06-10T08:18:32Z, 2024. Accessed: Nov. 20, 2025. [Online]. Available: https://clear.ml.

[9] Databricks, *Apply MLOps at scale*, Dec. 8, 2020. Accessed: Dec. 9, 2025. [Online]. Available: https://www.youtube.com/watch?v=dlFa_5K3ygY.

[10] M. Dehghani and Z. Yazdanparast, *A survey from distributed machine learning to distributed deep learning*, Sep. 9, 2023. DOI: 10.48550/arXiv.2307.05232. arXiv: 2307.05232[cs]. Accessed: Jun. 21, 2025. [Online]. Available: http://arxiv.org/abs/2307.05232.

[11] delynchoong. “MLOps maturity model - azure architecture center,” Accessed: Dec. 15, 2025. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/ai-ml/guide/mlops-maturity-model.

[12] O. S. Ekundayo and A. E. Ezugwu, “Deep learning: Historical overview from inception to actualization, models, applications and future trends,” *Applied Soft Computing*, vol. 181, p. 113 378, Sep. 1, 2025, ISSN: 1568-4946. DOI: 10.1016/j.asoc.2025.113378. Accessed: Dec. 7, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1568494625006891.

[13] A. R. Hevner, S. T. March, J. Park, and S. Ram, “Design science in information systems research,” *MIS Quarterly*, vol. 28, no. 1, pp. 75–105, 2004, Publisher: Management Information Systems Research Center, University of Minnesota, ISSN: 0276-7783. DOI: 10.2307/25148625. Accessed: Aug. 31, 2025. [Online]. Available: https://www.jstor.org/stable/25148625.

[14] C. Janiesch, P. Zschech, and K. Heinrich, “Machine learning and deep learning,” *Electronic Markets*, vol. 31, no. 3, pp. 685–695, Sep. 1, 2021, ISSN: 1422-8890. DOI: 10.1007/s12525-021-00475-2. Accessed: Jun. 21, 2025. [Online]. Available: https://doi.org/10.1007/s12525-021-00475-2.

[15] M. M. John, H. H. Olsson, and J. Bosch, “Towards MLOps: A framework and maturity model,” in *2021 47th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, Sep. 2021, pp. 1–8. DOI: 10.1109/SEAA53835.2021.00050. Accessed: Dec. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9582569.

[16] J. Kazmierczak, K. Salama, and V. Huerta. “MLOps: Continuous delivery and automation pipelines in machine learning | cloud architecture center,” Google Cloud Documentation, Accessed: Dec. 14, 2025. [Online]. Available: https://docs.cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning.

[17] D. Kreuzberger, N. Kühl, and S. Hirschl, *Machine learning operations (MLOps): Overview, definition, and architecture*, May 14, 2022. DOI: 10.48550/arXiv.2205.02302. arXiv: 2205.02302[cs]. Accessed: Jun. 21, 2025. [Online]. Available: http://arxiv.org/abs/2205.02302.

[18] R. Lam et al., “Learning skillful medium-range global weather forecasting,” *Science*, vol. 382, no. 6677, pp. 1416–1421, Dec. 22, 2023, Publisher: American Association for the Advancement of Science. DOI: 10.1126/science.adi2336. Accessed: Dec. 7, 2025. [Online]. Available: https://www.science.org/doi/10.1126/science.adi2336.

[19] P. Manshausen et al., *Generative data assimilation of sparse weather station observations at kilometer scales*, Apr. 1, 2025. DOI: 10.48550/arXiv.2406.16947. arXiv: 2406.16947[cs]. Accessed: Jun. 21, 2025. [Online]. Available: http://arxiv.org/abs/2406.16947.

[20] D. McVicar, B. Avant, A. Gould, D. Torrejon, C. D. Porta, and R. Mukherjee, “Smartflow: Enabling scalable spatiotemporal geospatial research,” in *IGARSS 2023 - 2023 IEEE International Geoscience and Remote Sensing Symposium*, Jul. 16, 2023, pp. 1193–1196. DOI: 10.1109/IGARSS52108.2023.10283095. arXiv: 2506.03022[cs]. Accessed: Dec. 9, 2025. [Online]. Available: http://arxiv.org/abs/2506.03022.

[21] D. Narayanan et al., *Efficient large-scale language model training on GPU clusters using megatron-LM*, Aug. 23, 2021. DOI: 10.48550/arXiv.2104.04473.

arXiv: 2104.04473[cs]. Accessed: Jun. 21, 2025. [Online]. Available: http://arxiv.org/abs/2104.04473.

[22] J. Pathak et al., *Kilometer-scale convection allowing model emulation using generative diffusion modeling*, Aug. 20, 2024. DOI: 10.48550/arXiv.2408.10958. arXiv: 2408.10958[physics]. Accessed: Dec. 7, 2025. [Online]. Available: http://arxiv.org/abs/2408.10958.

[23] K. Peffers, T. Tuunanen, M. A. Rothenberger, and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems*, vol. 24, no. 3, pp. 45–77, Dec. 2007, ISSN: 0742-1222, 1557-928X. DOI: 10.2753/MIS0742-1222240302. Accessed: Aug. 31, 2025. [Online]. Available: https://www.tandfonline.com/doi/full/10.2753/MIS0742-1222240302.

[24] V. T. Ponnaganti, “Scalable multi-model orchestration in AI microservices with kubernetes and serverless for event-driven MLOps pipelines,” in *2025 International Conference on Intelligent Computing and Control Systems (ICICCS)*, Mar. 2025, pp. 1471–1476. DOI: 10.1109/ICICCS65191.2025.10985404. Accessed: Dec. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/10985404/.

[25] P. Ruf, M. Madan, C. Reich, and D. Ould-Abdeslam, “Demystifying MLOps and presenting a recipe for the selection of open-source tools,” *Applied Sciences*, vol. 11, no. 19, p. 8861, Jan. 2021, Publisher: Multidisciplinary Digital Publishing Institute, ISSN: 2076-3417. DOI: 10.3390/app11198861. Accessed: Dec. 14, 2025. [Online]. Available: https://www.mdpi.com/2076-3417/11/19/8861.

[26] D. Ruggeri, G. Tazza, and L. Vidács, “Introducing MLOps to facilitate the development of machine learning models in agronomy: A case study,” *IEEE Access*, vol. 13, pp. 122 059–122 070, 2025, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2025.3586691. Accessed: Dec. 9, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/11072436/.

[27] E. Salvucci, “MLOps - standardizing the machine learning workflow,” Ph.D. dissertation, 2021.

[28] T. H. Sandhu and A. Itkikar, “MACHINE LEARNING AND NATURAL LANGUAGE PROCESSING â€“ a REVIEW,” *International Journal of Advanced Research in Computer Science*, vol. 9, no. 2, pp. 582–584, Apr. 20, 2018, Number: 2, ISSN: 0976-5697. DOI: 10.26483/ijarcs.v9i2.5799. Accessed: Jun. 21, 2025. [Online]. Available: https://ijarcs.info/index.php/Ijarcs/article/view/5799.

[29] O. Spjuth, J. Frid, and A. Hellander, “The machine learning life cycle and the cloud: Implications for drug discovery,” *Expert Opinion on Drug Discovery*, vol. 16, no. 9, pp. 1071–1079, Sep. 2, 2021, Publisher: Taylor & Francis _eprint: https://doi.org/10.1080/17460441.2021.1932812, ISSN: 1746-0441. DOI: 10.1080/17460441.2021.1932812. Accessed: Dec. 14, 2025. [Online]. Available: https://doi.org/10.1080/17460441.2021.1932812.

[30] G. Symeonidis, E. Nerantzis, A. Kazakis, and G. A. Papakostas, “MLOps - definitions, tools and challenges,” in *2022 IEEE 12th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan. 2022, pp. 0453–0460. DOI: 10.1109/CCWC54503.2022.9720902. Accessed: Dec. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9720902.

[31] D. A. Tamburri, “Sustainable MLOps: Trends and challenges,” in *2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Sep. 2020, pp. 17–23. DOI: 10.1109/SYNASC51798.2020.00015. Accessed: Dec. 15, 2025. [Online]. Available: https://ieeexplore.ieee.org/document/9356947.

[32] T. J. Vandal, K. Duffy, D. McDuff, Y. Nachmany, and C. Hartshorn, *Global atmospheric data assimilation with multi-modal masked autoencoders*, Jul. 16, 2024. DOI: 10.48550/arXiv.2407.11696. arXiv: 2407.11696[cs]. Accessed: Dec. 7, 2025. [Online]. Available: http://arxiv.org/abs/2407.11696.

[33] J. Venable, J. Pries-Heje, and R. Baskerville, “FEDS: A framework for evaluation in design science research,” *European Journal of Information Systems*, vol. 25, no. 1, pp. 77–89, Jan. 1, 2016, ISSN: 1476-9344. DOI: 10.1057/ejis.2014.36. Accessed: Sep. 9, 2025. [Online]. Available: https://doi.org/10.1057/ejis.2014.36.

[34] Y. Wang. “AI scaling: From up to down and out,” Accessed: Dec. 7, 2025. [Online]. Available: https://arxiv.org/html/2502.01677v1.

[35] X. Xu, X. Sun, W. Han, X. Zhong, L. Chen, and H. Li, *Fuxi-DA: A generalized deep learning data assimilation framework for assimilating satellite observations*, Apr. 12, 2024. DOI: 10.48550/arXiv.2404.08522. arXiv: 2404.08522[cs]. Accessed: Dec. 7, 2025. [Online]. Available: http://arxiv.org/abs/2404.08522.