Machine Learning for Autonomic System Operation in the Heterogeneous Cloud-Edge Continuum

Contract Number 101092912

D4.4 Final Version of AI Architecture and ML Models

Lead Partner	UCD
Contributing Partners	All partners
Owner / Main Author	UCD: John Byabazaire, Dimitris Chatzopoulos, Theodoros Aslanidis
Contributing Authors	CC: Marcell Fehér INRIA: Valéria Loscrì, Jiali Xu UTH: Alexandros Patras, Foivos Pournaropoulos, Maria Rafaela Gkeka, Nikos Bellas, Christos Antonopoulos, Spyros Lalis NTT: Andrea Pazienza, Massimiliano Rossi TUD: Rui Wang AUG : Dimitrios Akridas, Spyridon Evangelatos UNICAL: Gianluca Aloi, Antonio Iera, Michele Gianfelice
Reviewers	Raffaele Gravina (UNICAL), Filipe Sousa (FhP-AICOS)
Contractual Delivery Date	30/6/2025
Actual Delivery Date	19/7/2025
Version	1.0
Dissemination Level	Public

HORIZON EUROPE

The research leading to these results has received funding from the European Community's Horizon Europe Programme (HORIZON) under grant no. 101092912.

Change Log

Version	Summary of Changes
0.1	Initial skeleton prepared by UCD
0.2	Added draft of section 7
0.3	Added draft of section 5
0.4	Added draft for section 6
0.5	Added draft sections 3 and 11
0.6	Refined and homogenized all sections
0.7	Version released for internal review
1.0	Final version

List of Figures

List of Tables

Summary

This document presents the design of the MLSysOps Machine Learning (ML) framework. The framework intends to facilitate resource management and application orchestration across the computing continuum by integrating multiple ML models with the entire MLSysOps continuum. The framework's comprehensive components, from dynamic storage management to anomaly detection and ML model retraining and drift detection, show a robust approach to enhancing the functionality and efficiency of edge and cloud computing ecosystems. The design and implementation of the MLSysOps ML framework described here, along with certain ML approaches, lay the foundation for further development.

Abbreviations

Term	Definition
5GCN	5G Core Network
A2C	Advantage Actor Critic
ACL	Agent Communication Language
AI	Artificial Intelligence
ANOVA	Analysis of Variance
API	Application Programming Interface
AWS	Amazon Web Services
CFO	Carrier Frequency Offset
CFR	Channel Frequency Response
CIR	Channel Impulse Response
CL	Continual Learning
CNN	Convolutional Neural Network
CO	Classification Oriented
CPI	Cycles per Instruction
CPU	Central Processing Unit
DCAI	Data-Centric AI
DCO	Direct Current Offset
EC	Erasure Coding
FSL	Few-shot learning
gNB	Next-Generation Node B
HTTP	Hyper Text Transfer Protocol
IIoT	Industrial Internet of Things
KNN	K-Nearest Neighbors
LR	Logistic Regression
LSTM	Long Short-Term Memory
MAML	Model-Agnostic Meta-Learning
MDP	Markov Decision Process
MI	Mutual Information
ML	Machine Learning
MLaaS	Machine Learning as a Service
MLP	Multi-Layer Perceptron
PCA	Principal Component Analysis
PO	Phase Offset
PPO	Proximal Policy Optimization
RAN	Radio Access Network
RFE	Recursive Feature Elimination
RLNC	Random Linear Network Coding
RnF	Random Forest

RSS	Received Signal Strength
SDR	Software Defined Radio
SMOTE	Synthetic Minority Over-sampling Technique
SVM	Support Vector Machines
UE	User Equipment
XAI	Explainable AI
XGB	XGBoost

1 Introduction

The increasing volume and complexity of data generated by modern computing systems demand efficient and immediate processing, making edge computing essential for processing data closer to its source, reducing latency, enhancing real-time data analysis, and alleviating bandwidth constraints on central cloud servers. Energy-efficient micro-servers and powerful embedded devices further support this need. Edge computing extends cloud services closer to end-users by providing nodes with computing, storage, and networking resources at the network edge, such as IoT gateways and micro data centres. However, the scale and heterogeneity of cloud-edge computing require AI-driven management, as traditional rule-based approaches are insufficient, making autonomic computing systems that manage themselves based on high-level objectives a promising alternative.

MLSysOps aims to introduce AI-driven resource management and application deployment/orchestration mechanisms across the computing continuum. MLSysOps is designed to address the challenges of managing complex software ecosystems and the hardware that supports them. The framework integrates multiple ML models and agents to facilitate dynamic and efficient resource allocation, system monitoring, and performance optimization. The primary goal is to ensure that software applications run smoothly within containerized environments, with enhanced support for 5G infrastructure and large-scale machine learning model training.

This document introduces the MLSysOps ML framework. It comprises several key components, including dynamic storage management, trust management, anomaly detection, application placement and management, and integration with 5G. Each component is decoupled from the other, and each plays a crucial role in enhancing the management and optimization of software systems. This design ensures interoperability and avoids vendor lock-in. At the core of this design are MLSysOps agents. These specialized agents interact with the various system components and the MLConnector APIs to integrate machine learning (ML) models for decision-making and support for model monitoring and retraining. This multi-faceted interaction ensures seamless coordination and efficient system operations.

The rest of the document is structured as follows: Section 2 introduces and defines some of the terms necessary for the remainder of the document, including data-centric AI, reinforcement learning, explainable AI, and many others. Section 3 highlights the MLSysOps ML framework design. It highlights the core ML components, defines the different kinds of agents, and explains how these mediate the communication between the various system components. The section concludes by describing the proposed system design and briefly highlighting the role each component plays.

Subsequent sections highlight the ML integration at each layer defined in Section 3. Section 4 introduces trust assessment and management using online weighted anomaly detection. Section 5 highlights a novel hybrid physical layer authentication scheme for anomaly detection. Section 6 describes the use of ML-driven anomaly detection approaches to monitor 5G edge networks. Section 7 explores the use of reinforcement learning to minimize the latency of traffic routing for 5G users. The goal is to use reinforcement learning to select optimal routing paths based on metrics such as latency, bandwidth usage, and traffic load. Section 8 introduces a highly dynamic, ML-driven approach to adjust the redundancy level and geographical distribution of storage objects to minimize operating costs while satisfying high-level user performance requirements. Sections 9 and 10 describe facilities and mechanisms that aid application component placement, management, and reconfiguration, and describe how ML can be used to facilitate such services. Sections 11, 12, and 13 describe the mechanisms for ML model training, monitoring, and retraining. Also, the facilities for delivery and integration via the MLConnector API are discussed.

2 Background

Data-centric Artificial Intelligence

Data-Centric AI (DCAI)[1] is a paradigm shift in artificial intelligence development that emphasizes the importance of high-quality data over model complexity. Traditionally, AI development has been model-centric, where the primary focus is on improving algorithms and model architectures. In contrast, DCAI involves systematically engineering and improving datasets to enhance the performance of AI models. AI systems rely heavily on the data they are trained on. Inconsistent, noisy, or biased data can lead to poor model performance, regardless of the sophistication of the algorithms used. DCAI addresses these issues by treating data as a first-class citizen in the AI development process. By improving the quality of data, AI models can achieve better accuracy and generalization, which is crucial for real-world applications.

The key principles of DCAI are as follows:

- **Data Quality Over Model Complexity:** Improving the dataset itself is often more impactful than refining the model. This involves processes such as data cleaning, augmentation, labelling, and curation.
- **Programmatic Data Labelling:** Manual labelling is time-consuming and often impractical for large datasets. Programmatic labelling, which uses automated techniques to label data, is a more efficient approach. This method leverages subject matter expertise through coding rules or machine learning models to label data.
- **Cooperation:** Effective AI systems require the input of subject matter experts (SMEs) who can provide valuable insights into the data. DCAI promotes collaboration between SMEs and data scientists to ensure the data accurately reflects real-world conditions and knowledge.

The shift to a data-centric approach is critical for several reasons:

- **Improved Model Performance:** High-quality, well-curated data can significantly enhance the accuracy and reliability of AI models. This reduces the "garbage in, garbage out" problem, where poor data leads to poor model performance.
- **Efficiency in AI Development:** Focusing on data quality can streamline the development process. Better data reduces the need for extensive model tuning and complex architectures, saving time and resources.
- **Scalability and Adaptability:** Programmatic data labelling and systematic data management practices make it easier to scale AI solutions and adapt to new data or changing conditions without extensive rework.
- **Ethical and Responsible AI:** By involving SMEs and focusing on data quality, DCAI helps identify and mitigate biases in datasets, leading to more ethical and fair AI systems.

In contrast to the data-centric approach, the traditional model-centric AI approach focuses primarily on refining and improving the machine learning models themselves. This involves extensive work on developing sophisticated algorithms, optimizing model architectures, and fine-tuning parameters to achieve better performance. While this approach has been successful and has driven many advances in AI, it often treats the data as a static input that is fixed and unchangeable. As a result, significant time and resources are spent on iterative improvements to the model, sometimes overlooking the potential gains that can be made by enhancing the quality and relevance of the data.

DCAI presents several significant challenges despite its promising approach to enhancing machine learning systems[2]. One primary challenge is the requirement for extensive human involvement in labelling and curating data. This process can be time-consuming and labour-intensive, especially for complex datasets that require

1 Zha, Daochen, et al. "Data-centric artificial intelligence: A survey." *arXiv preprint arXiv:2303.10158* (2023).
2 Whang, Steven Euijong, et al. "Data collection and quality challenges in deep learning: A data-centric ai perspective." *The VLDB Journal* 32.4 (2023): 791-813.

subject matter expertise for accurate labelling. Another major challenge is the inconsistency and variability in data management practices across different teams and projects. Standardizing workflows and scaling AI solutions can be difficult when different teams use varying methods to develop AI solutions, manage data, and collaborate with quality teams. This lack of standardization makes it hard to apply lessons learned from one project to another, hindering the ability to scale successful solutions efficiently. Additionally, real-world data is often private and proprietary, posing further difficulties in sharing and leveraging data across different applications and organizations[3].

Ensemble Learning

Ensemble Learning (EL)[4] is an ML technique that combines multiple individual models, referred to as base learners, to create a more accurate and robust prediction model. The base learners can be weak or strong, and their combined output is often superior to that of any single model[5].

The basic concepts of Ensemble Learning are the following:

- **Base Learners:** The individual models that make up an ensemble. These can be weak learners, which are slightly better than random guessing, or strong learners, which have high predictive accuracy.
- **Weak Learner:** A model that performs slightly better than random chance. In ensemble learning, multiple weak learners are combined to create a strong learner.
- **Strong Learner:** A model that has high accuracy and predictive power.

We can distinguish four major categories of Ensemble Learning:

- **Bagging (Bootstrap Aggregating):**
  - **Process:** Multiple versions of a dataset are created using bootstrap sampling (sampling with replacement). Each version is used to train a different base learner.
  - **Examples:** Random Forests[6], where multiple decision trees are trained on different bootstrap samples and their outputs are averaged.
- **Boosting:**
  - **Process:** Base learners are trained sequentially. Each new model focuses on the errors made by the previous models.
  - **Examples:** AdaBoost[7], Gradient Boosting Machines (GBM)[8], and XGBoost[9].

- **Stacking:**
  - **Process:** Different models (base learners) are trained on the same dataset. The predictions of these models are then used as inputs for a meta-learner, which makes the final prediction.
  - **Examples:** Combining linear regression, decision trees, and neural networks where their outputs are used to train a logistic regression model as the meta-learner.

---

3 Zha, Daochen, et al. "Data-centric ai: Perspectives and challenges." *Proceedings of the 2023 SIAM International Conference on Data Mining (SDM)*. Society for Industrial and Applied Mathematics, 2023

4 Zha, Daochen, et al. "Data-centric ai: Perspectives and challenges." *Proceedings of the 2023 SIAM International Conference on Data Mining (SDM)*. Society for Industrial and Applied Mathematics, 2023.

5 Sagi, Omer, and Lior Rokach. "Ensemble learning: A survey." *Wiley interdisciplinary reviews: data mining and knowledge discovery* 8.4 (2018): e1249.

6 Breiman, Leo. "Random forests." *Machine learning* 45 (2001): 5-32.

7 Freund, Yoav, Robert Schapire, and Naoki Abe. "A short introduction to boosting." *Journal-Japanese Society For Artificial Intelligence* 14.771-780 (1999): 1612.

8 He, Zhiyuan, et al. "Gradient boosting machine: a survey." *arXiv preprint arXiv:1908.06951* (2019).

9 Chen, Tianqi, and Carlos Guestrin. "Xgboost: A scalable tree boosting system." *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016.

- **Voting:**
    - **Process:** Multiple models are trained on the same dataset. For classification, each model votes on the class, and the majority vote is taken as the final prediction. For regression, the average prediction is used.
    - **Examples:** Hard voting (majority class wins) and soft voting (average of predicted probabilities).

Unlike single-model approaches, ensemble learning leverages the diversity of multiple models to improve performance. It reduces overfitting by averaging out biases, handles more complex data patterns, and enhances generalizability. Single models may suffer from high bias or variance, but ensembles balance these errors more effectively.

Ensemble learning is crucial for improving model accuracy and robustness, especially when dealing with complex datasets or problems with high variance. It enhances models' predictive performance, making it a preferred choice in competitions and real-world applications where accuracy is paramount.

The primary benefits of ensemble learning include:

- **Improved Accuracy:** Combining multiple models often leads to better accuracy compared to a single model.
- **Reduced Overfitting:** Ensembles can reduce the risk of overfitting, particularly if the models are diverse, by averaging multiple models.
- **Better Generalization:** Ensembles tend to generalize better to new, unseen data.

These advantages make ensemble methods highly effective for both classification and regression tasks. Despite the effectiveness of ensemble learning in improving predictive accuracy and robustness, one should also be aware of the associated challenges and considerations. Ensemble learning presents several challenges and considerations that must be addressed to maximize its effectiveness. One major challenge is computational complexity, as training multiple models and combining their predictions can require significant computational resources and time. Another consideration is interpretability; ensembles, particularly those that incorporate numerous complex models, can be difficult to interpret, making it hard to understand how predictions are made. Additionally, the success of ensemble methods heavily depends on the diversity of the models used; ensuring that the base learners are diverse and make different errors is crucial for achieving the best performance and reducing overfitting.

Recent advances in ensemble learning have focused on improving fairness and integrating deep learning techniques. Fairness in ensemble learning aims to address biases in models to ensure equitable predictions across different demographic groups, mitigating issues of discrimination and promoting ethical AI practices. Additionally, ensemble learning in deep learning involves combining multiple deep neural networks to enhance robustness and accuracy. This approach leverages the strengths of various neural network architectures, resulting in models that are more resilient to errors and capable of delivering superior performance in complex tasks.

Continual Learning

Continual Learning (CL)[10], also known as Lifelong Learning, is an approach within machine learning that allows models to learn continuously from a stream of data, adapting to new information without forgetting previously

10 De Lange, Matthias, et al. "Continual learning: A comparative study on how to defy forgetting in classification tasks." *arXiv preprint arXiv:1909.08383* 2.6 (2019): 2.

acquired knowledge. This paradigm is crucial for developing systems that need to evolve over time, such as autonomous agents, personalized recommendation systems, and adaptive user interfaces[11].

CL strategies can be broadly categorized into three main approaches:

- **Architectural Approaches**: These involve modifying the neural network architecture to accommodate new information without disrupting existing knowledge. Techniques include adding new neurons or layers to the network, such as Progressive Neural Networks, which create subnetworks for new tasks while preserving the original network.
- **Regularization Approaches**: Regularization methods adjust the learning process to minimize the impact on previously learned tasks. Techniques like Elastic Weight Consolidation (EWC)[12] and Learning Without Forgetting (LWF)[13] constrain the updates to critical parameters, preventing significant changes that could lead to forgetting.
- **Memory-Based Approaches**: These methods maintain a buffer of previous data samples and are used during training to reinforce old knowledge. This helps the model remember past information while learning new tasks. Examples include experience replay and generative replay, where past experiences are either stored explicitly or generated by a model and used during training.

The main benefits of using CL methods can be summarized as follows:

- **Enhanced Adaptability**: CL systems can adapt to new information and changes in the environment without requiring complete retraining from scratch. This ability allows for more flexible and responsive AI models that remain relevant and accurate over time.
- **Improved Efficiency**: By leveraging previously learned knowledge, CL reduces the need for extensive data and computational resources when encountering new tasks. This efficiency can lead to faster development cycles and lower costs associated with model training.
- **Better Generalization**: CL helps models generalize better by incorporating diverse experiences over time. This generalization is crucial for creating robust AI systems capable of performing well in various scenarios and tasks.
- **Reduced Data Requirements**: Utilizing transfer learning techniques, CL systems can achieve high performance on new tasks with less data. This reduction in data requirements is particularly beneficial in domains where data collection is expensive or time-consuming.
- **Mitigation of Catastrophic Forgetting**: Catastrophic forgetting occurs when a model loses previously learned information upon learning new data, leading to a decline in performance on previous tasks. CL incorporates strategies to balance stability and plasticity, allowing systems to retain valuable insights while integrating new knowledge and ensuring consistent performance across tasks.
- **Incremental Improvement**: CL allows models to improve incrementally, similar to human learning processes. This incremental improvement fosters a more natural and continuous development of AI capabilities, aligning better with real-world applications where new data is continuously generated.

Despite its potential, CL faces several challenges:

11 Wang, Liyuan, et al. "A comprehensive survey of continual learning: theory, method and application." *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2024).

12 Kirkpatrick, James, et al. "Overcoming catastrophic forgetting in neural networks." *Proceedings of the national academy of sciences* 114.13 (2017): 3521-3526.

13 Li, Zhizhong, and Derek Hoiem. "Learning without forgetting." *IEEE transactions on pattern analysis and machine intelligence* 40.12 (2017): 2935-2947.

- **Scalability**: Efficiently scaling CL models to handle vast amounts of data and numerous tasks is a significant challenge.
- **Evaluation Metrics**: Robust metrics must be developed to evaluate the performance of CL systems across various tasks and over time.
- **Real-World Applications**: Bridging the gap between theoretical advancements and practical, real-world applications remains an ongoing research challenge.

Despite its potential, CL faces several challenges. One significant challenge is scalability, as efficiently scaling CL models to handle vast amounts of data and numerous tasks is difficult. Another challenge lies in developing robust evaluation metrics to assess the performance of CL systems across various tasks and over time. Additionally, bridging the gap between theoretical advancements and practical, real-world applications remains an ongoing research challenge.

Explainable Artificial Intelligence

Explainable AI (XAI)[14] is a field dedicated to making the decision-making processes of AI models more transparent and understandable to humans. As AI systems become increasingly complex and integrated into critical applications, the need for explanations that are comprehensible to stakeholders has grown. XAI aims to provide insights into how AI models arrive at their decisions, ensuring they are not only accurate but also trustworthy and aligned with human values and expectations[15].

Explainable AI (XAI) methods can be classified according to different criteria:

- The first criterion distinguishes between intrinsic and post-hoc methods. Intrinsic methods, such as Linear Regression, analyse models during training or are inherently interpretable. Post-hoc methods, like LIME[16], are applied to trained models to generate explanations.
- The second criterion differentiates between model-specific and model-agnostic methods. Model-specific methods, such as Layer-wise Relevance Propagation, are tailored to specific model types and require detailed knowledge of the model's inner workings. In contrast, model-agnostic methods, like SHAP[17], can be applied to any model type by manipulating inputs and features without delving deeply into the model's internal mechanics.
- The third criterion considers the scope of the explanation, distinguishing between local and global methods. Local methods, such as counterfactual explanations, focus on specific data instances or features. Global methods, like prototypes, examine overall model behaviour or dataset patterns.
- The fourth criterion focuses on the type of explanation provided. This includes intrinsically interpretable methods, knowledge extraction methods that use interpretable models to extract insights from complex models, feature influence methods that summarize statistics on features or their interactions, visualizations that use plots or graphs to convey explanations, and example-based explanations that concentrate on specific data instances.

Key properties of effective explanations include accuracy (performance on unseen data), fidelity (capturing the black box model’s behaviour), consistency (similarity across different models trained on the same task), stability (similarity between explanations for similar instances), comprehensibility (ease of human understanding), certainty (reflecting the certainty of predictions), degree of importance (showing the importance of each component), and novelty (ensuring the instance is well-represented by the data).

---

14 Hoffman, Robert R., et al. "Metrics for explainable AI: Challenges and prospects." *arXiv preprint arXiv:1812.04608* (2018).
15 Dwivedi, Rudresh, et al. "Explainable AI (XAI): Core ideas, techniques, and solutions." *ACM Computing Surveys* 55.9 (2023): 1-33.

16 Ribeiro, Marco Tulio, Sameer Singh, and Carlos Guestrin. "" Why should i trust you?" Explaining the predictions of any classifier." *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*. 2016.

17 Lundberg, Scott M., and Su-In Lee. "A unified approach to interpreting model predictions." *Advances in neural information processing systems* 30 (2017).

Explanation methods themselves have important properties, including expressive power (the structure of the explanation), translucency (the degree of insight into the model), portability (applicability across different ML models), and algorithmic complexity (computational overhead).

Human-friendly explanations should be contrastive, selective, consistent, probable, and generic. Contrastive explanations, which compare different scenarios, are particularly intuitive and align well with human thinking processes. Counterfactual explanations are popular for providing such intuitive, example-based explanations.

Figure 2.1 provides a visual representation of various XAI methods categorized according to these approaches.

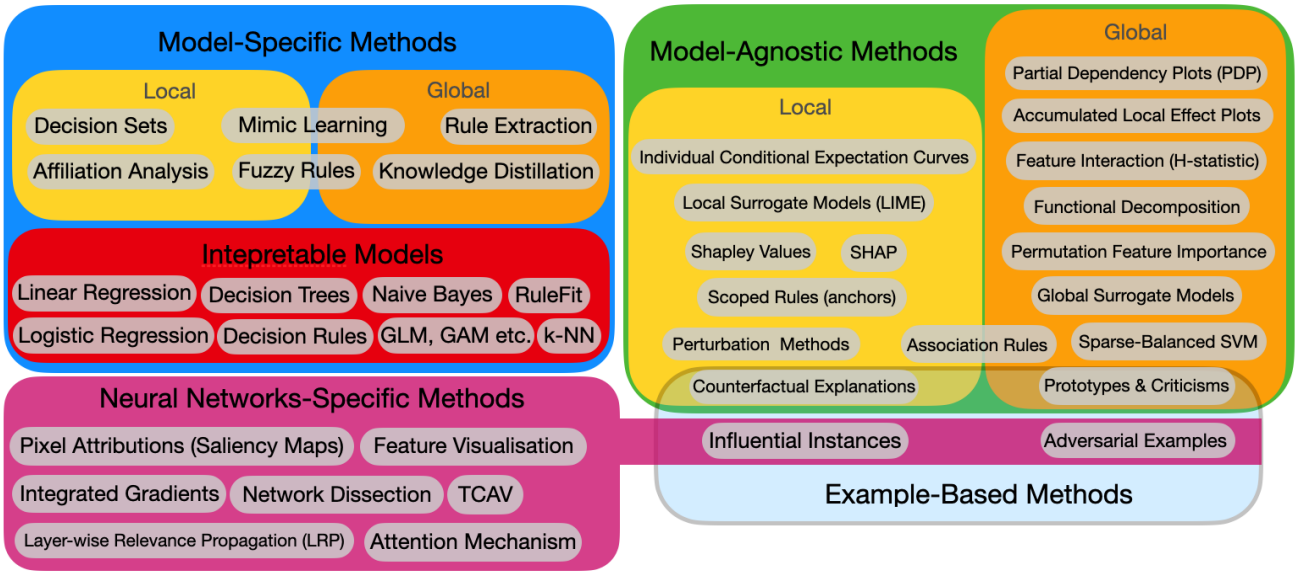


Figure 2.1: A taxonomy of Explainable AI methods

Federated Learning

Federated Learning (FL)[18] is an innovative approach to machine learning that enables the training of models across decentralized devices holding local data samples, without exchanging the data itself. This method addresses significant concerns related to data privacy, security, and accessibility, which are critical in various sectors such as healthcare, finance, and beyond.

At its core, federated learning allows multiple entities, often referred to as clients (such as mobile devices or edge servers), to train a shared machine learning model collaboratively. The primary advantage is that the raw data remains on the local devices, and only model updates, like gradients or weights, are transmitted. This

18 Zhang, Chen, et al. "A survey on federated learning." *Knowledge-Based Systems* 216 (2021): 106775.

ensures that sensitive information is not exposed or centralized, mitigating privacy risks. The general workflow of federated learning involves several key steps, as shown in Figure 2.2 below:

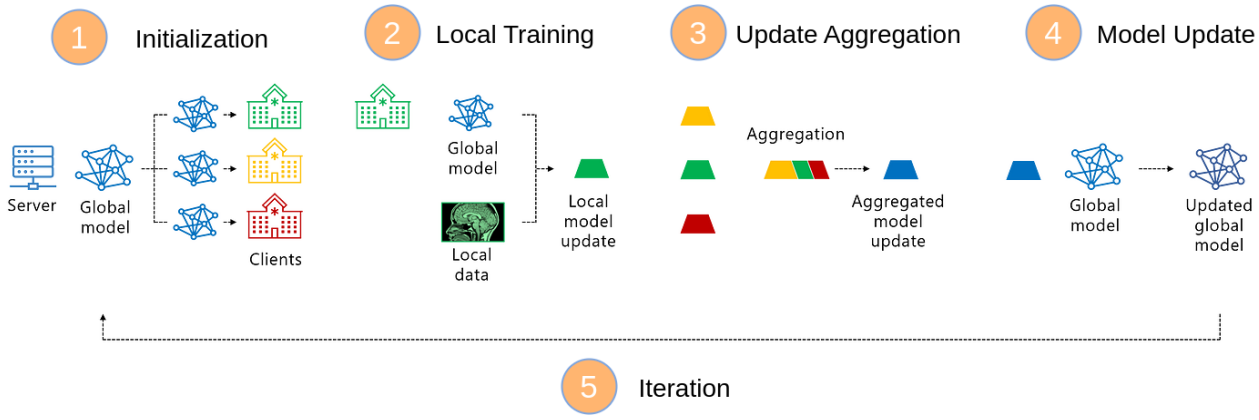


Figure 2.2: Workflow diagram of federated learning

1. **Initialization**: A global model is initialized at a central server and sent to all participating clients.
2. **Local Training**: Each client uses its local data to train the model independently.
3. **Update Aggregation**: Clients send the updated model parameters back to the central server.
4. **Model Update**: The central server aggregates these updates (typically through methods like federated averaging) to update the global model.
5. **Iteration**: This process is repeated iteratively until the model converges to a satisfactory performance level.

Federated learning can be categorized based on how the data is distributed across the clients:

- **Horizontal Federated Learning**: Involves data that is distributed across clients that have similar features but different samples.
- **Vertical Federated Learning**: Entails clients that have different features but the same set of samples.
- **Federated Transfer Learning:** Combines aspects of horizontal and vertical federated learning. It is often used to improve model performance on specific tasks using pre-trained models fine-tuned on local data.

In the realm of the Internet of Things (IoT), federated learning allows for the aggregation of insights from numerous distributed sensors and devices, fostering the development of intelligent systems that can operate efficiently without the need for massive data transfers. This approach is particularly beneficial for scenarios involving limited bandwidth or strict data privacy requirements.

While federated learning presents numerous advantages, it also faces several challenges:

- **Data Heterogeneity**: The variability in data distributions across clients can affect model performance and convergence.
- **Communication Overhead**: Frequent communication between clients and the central server can be resource-intensive and slow down the training process.
- **System and Model Robustness**: It is critical to ensure the reliability and robustness of the federated learning system, especially in the presence of unreliable or malicious clients.

Recent advancements in federated learning aim to address these challenges through innovative solutions. For instance, dynamic regularization techniques help manage data heterogeneity, while advanced encryption methods ensure the secure and efficient aggregation of model updates. As research in this field progresses,

federated learning is expected to become increasingly robust and scalable, further solidifying its role as a pivotal technology for privacy-preserving machine learning.

Few-shot Learning

Few-shot learning (FSL) is a subfield of ML and deep learning focused on training models to achieve high performance with very limited data. Unlike traditional supervised learning methods that require large amounts of labelled data, few-shot learning aims to enable models to generalize from only a few examples. This approach is particularly valuable in scenarios where acquiring extensive labelled datasets is impractical or expensive.

Few-shot learning is typically categorized based on the number of examples provided during training:

- **Zero-Shot Learning**: The model predicts classes it has never seen during training, leveraging semantic information or auxiliary data about the classes.
- **One-Shot Learning**: The model is trained with only one example per class. This requires the model to generalize from a single instance.
- **Few-Shot Learning**: This category involves more than one but still a very limited number of examples

Several approaches have been developed for few-shot learning:

- **Meta-Learning (Learning to Learn)**: Meta-learning involves training a model on a variety of tasks so that it can quickly adapt to new tasks with minimal data. Model-Agnostic Meta-Learning (MAML)[19] is a popular algorithm in this category. It focuses on finding a good initialization of model parameters that can be fine-tuned with a few examples.
- **Metric Learning**: This approach aims to learn a similarity metric to compare examples. Techniques such as Siamese networks[20] and prototypical networks[21] fall under this category. They focus on learning a distance function to measure how similar or dissimilar two examples are.
- **Data Augmentation**: This technique involves generating additional training examples by applying various transformations to the existing data. For instance, in computer vision, transformations like rotation, scaling, and flipping can be used to create new images from the limited available samples[22].
- **Generative Models**: Models like Generative Adversarial Networks (GANs)[23] and Variational Autoencoders (VAEs)[24] can generate new data samples that resemble the few available examples, thereby augmenting the training dataset.

Few-shot learning offers several significant benefits[25]:

- **Reduced Data Requirements**: It significantly reduces the amount of labelled data needed, making it feasible to develop machine learning models in data-scarce environments.
- **Cost and Time Efficiency**: Few-shot learning reduces the costs and time associated with data collection and labelling by minimizing the need for extensive labelled datasets.

19 Finn, Chelsea, Pieter Abbeel, and Sergey Levine. "Model-agnostic meta-learning for fast adaptation of deep networks." *International conference on machine learning*. PMLR, 2017.

20 Wang, Bin, and Dian Wang. "Plant leaves classification: A few-shot learning method based on siamese network." *Ieee Access* 7 (2019): 151754-151763.

21 Snell, Jake, Kevin Swersky, and Richard Zemel. "Prototypical networks for few-shot learning." *Advances in neural information processing systems* 30 (2017).

22 Shorten, Connor, and Taghi M. Khoshgoftaar. "A survey on image data augmentation for deep learning." *Journal of big data* 6.1 (2019): 1-48.

23 Robb, Esther, et al. "Few-shot adaptation of generative adversarial networks." *arXiv preprint arXiv:2010.11943* (2020)

24 Wei, Ruoqi, and Ausif Mahmood. "Optimizing few-shot learning based on variational autoencoders." *Entropy* 23.11 (2021): 1390.

25 Wang, Yaqing, et al. "Generalizing from a few examples: A survey on few-shot learning." *ACM computing surveys (csur)* 53.3 (2020): 1-34.

- **Enhanced Flexibility and Adaptability**: Models trained with few-shot learning can quickly adapt to new tasks with minimal additional training, making them versatile and suitable for dynamic environments.
- **Broad Applicability**: Few-shot learning has applications across various domains, including computer vision (e.g., image classification, object detection), natural language processing (e.g., text classification, sentiment analysis), and robotics (e.g., object manipulation, motion planning).

Despite its advantages, few-shot learning also presents several challenges. One challenge is model complexity, as few-shot learning models, especially those based on meta-learning, can be complex and computationally intensive. Another challenge is generalization, ensuring that models generalize well from a few examples to unseen data can be difficult, and poor generalization can lead to overfitting on the limited training data. Scalability is also a challenge, as applying few-shot learning to large-scale problems or domains with high variability remains difficult. Models must balance learning useful representations and maintaining computational efficiency. Lastly, evaluating few-shot learning models can be tricky because standard metrics used for traditional machine learning may not be directly applicable or sufficient to capture the nuances of performance in few-shot scenarios.

Reinforcement Learning

Reinforcement Learning (RL) is the subset of ML in which an agent is trained through a trial-and-error process by interacting with an environment and observing its response. More concretely, at each discrete timestep t, the agent observes the current environment's state s_t, executes an action a_t, and consequently receives the subsequent state s_{t+1} along with a reward r_{t+1}. Figure 2.3 presents the main RL loop. The possible states the agent can observe at any given timestep form a mathematical set; the state space, denoted as S. Similarly, the set containing the agent's possible actions in a specific environment is denoted as A. Finally, the reward given to the agent at each timestep is calculated based on the reward function R, which acts as the feedback mechanism directly influencing the agent's behaviour. A higher reward indicates more favourable behaviour from the agent's perspective. A sequence of interactions between the agent and the environment, starting from an initial state s_1 and concluding at a terminal state s_T, define an episode of length T. The agent's primary objective is to optimize the total cumulative reward $R_t = \sum_{t=i}^{T} r_{t+1}$ accumulated over an episode to achieve its goal. This optimization may also influence the length of an episode, depending on specific problem requirements. A policy $\pi(s)$ is a function that maps the observed state s to an action a, driving the decision-making strategy of the agent at each timestep. It shows the actions that the agent should take for every possible state $s \in S$.

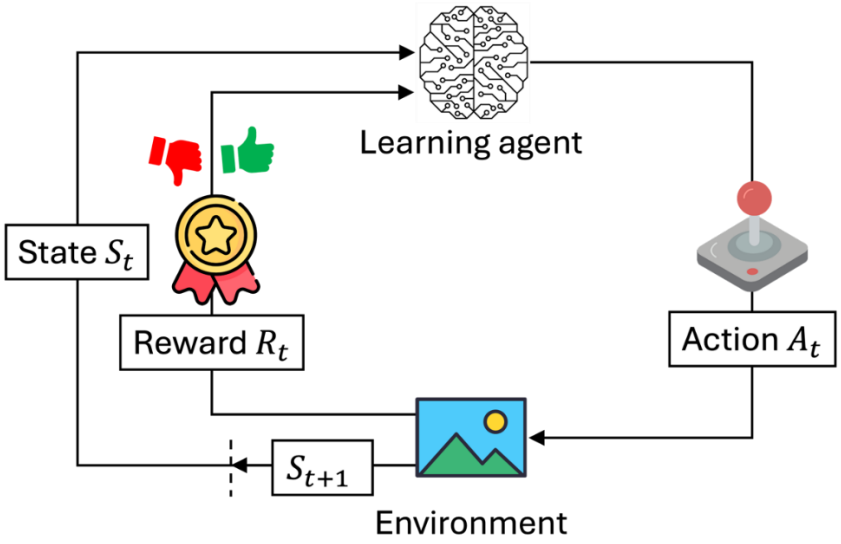


Figure 2.3: The core RL training loop

The agent, through its experiences, tries to improve its policy iteratively, either directly or indirectly. In addition to the policy, the agent uses two complementary functions to facilitate training. The first is the state-value function $V_\pi(s_t)$, which estimates the cumulative future rewards the agent can obtain from a given state. The other one is the action-value function $Q_\pi(s_t, a_t)$ which estimates the cumulative future rewards the agent can get performing a specific action at a given state. In simple terms, the value functions estimate how good it is for the agent to be in a particular state or take a specific action. A foundational concept in RL is the exploration-exploitation trade-off, where the agent strikes a balance between two opposing strategies. Exploitation means using already gained knowledge and choosing an action proven to work well, but it can also be suboptimal or problematic. Exploration means trying something new that may lead to better outcomes in the future. Finding the optimal balance between exploration and exploitation is a critical challenge in RL, aiming to maximize long-term goals. Regarding the mathematical formulation of RL, the system setting can be encapsulated as a Markov Decision Process (MDP). The name MDP refers to the fact that the RL problems follow the Markov property, meaning that the state transitions ($s_t \rightarrow s_{t+1}$) depend only on the current state and the action taken, and not on any previous states or actions.

2.1.1 Deep Reinforcement Learning

DRL has significantly boosted the performance of RL methods by leveraging deep learning capabilities. DRL has been successful in complex and dynamic environments where traditional RL approaches faced challenges with scalability. By employing deep learning models, DRL agents benefit from robust function approximations and the ability to handle larger state-action spaces. One advantage of DRL is its capacity for representation learning, allowing agents to extract meaningful features from raw sensory data and capture underlying patterns and structures. More concretely, deep learning can be incorporated into RL by using deep neural networks to learn the mapping of states into actions (policy) or estimating the cumulative reward from a specific state until the end of the episode (state value), as seen in Figure 2.4. However, it is essential to acknowledge the challenges and costs associated with DRL. Computationally, DRL models can be significantly resource-intensive. Moreover, the complexity introduced by DRL can reduce interpretability, making it challenging to understand the decision-making processes[26].

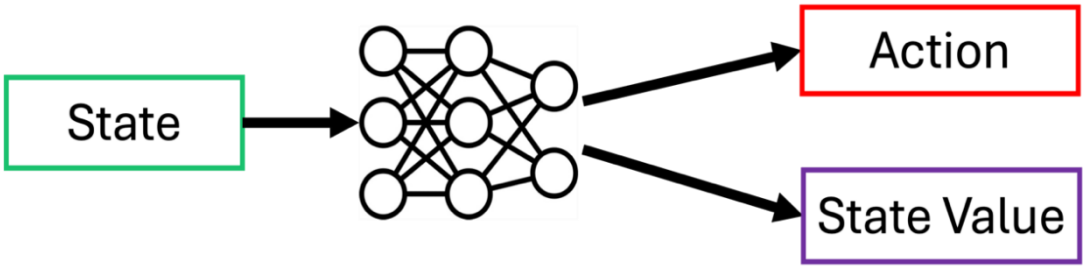


Figure 2.4: Depiction of how deep learning is incorporated into the RL paradigm

2.1.2 Categories of RL Algorithms

The available RL methods can be categorized according to various criteria. Here, we will recap the most important ones.

On-policy and Off-policy. Regarding how an RL algorithm utilizes experiences to update its policies or value functions, we can distinguish between on-policy and off-policy algorithms. On-policy algorithms directly learn the value of the policy being learned by making decisions based on the current policy and adjusting it in real-time based on the feedback received. In contrast, off-policy algorithms decouple the learning policy from the

26 Arulkumaran, Kai, et al. "Deep reinforcement learning: A brief survey." *IEEE Signal Processing Magazine* 34.6 (2017): 26-38.

decision policy, allowing the agent to learn optimal behaviour from exploratory actions derived from a different policy.

A critical component in off-policy algorithms is the *replay buffer*, which stores experience tuples (*state, action, reward, next state*) collected during the agent's interaction with the environment. This buffer enables the agent to revisit previous states, actions, and outcomes, facilitating learning from past experiences. The replay buffer serves multiple purposes: *(i)* it breaks the temporal correlations between consecutive learning samples, *(ii)* enriches the learning process by reusing past experiences, and *(iii)* improves the efficiency and stability of the learning algorithm. A replay buffer allows the integration of diverse experiences into the learning process, enhancing the agent's ability to generalize and adapt to the environment effectively.

Generally, on-policy algorithms offer more stable learning, better sample efficiency, and exploration-exploitation balance. However, they may hesitate to try new actions, and the convergence may be slow. On the other hand, off-policy algorithms offer data reusability because another policy can use the experience tuples stored from one off-policy algorithm. Also, they separate exploration and exploitation, providing a clear distinction between these two that allows exploration without sacrificing exploitation. Moreover, they are more flexible in adapting to new changes without affecting learning. Yet off-policy algorithms are more unstable, data inefficient, and do not balance exploration and exploitation well.

Value-based, Policy-based, and Actor-Critic. Suppose we focus more on what the RL agent tries to learn and update through interaction with the environment. In that case, we can categorize the algorithms into *value-based*, *policy-based*, and *actor-critic*. Value-based methods focus on estimating a value for each state or state-action pair that represents the expected cumulative reward that the agent can get. By iteratively updating value estimates based on observed rewards and state transitions, these methods enable the agent to make decisions that maximize long-term expected rewards. However, they may struggle with high-dimensional state spaces, where possible states become significantly large. Despite this limitation, value-based methods are powerful approaches to learning optimal policies based on value estimates.

On the other hand, policy-based methods directly try to learn the optimal policy, determining the action selection at each state. These methods aim to maximize the expected cumulative reward through iterative updates to the policy. The policy is updated toward higher expected rewards using techniques like policy gradients, which employ gradient ascent. This enables the agent to explore and improve its behaviour without relying on value estimation. However, these methods often encounter the challenge of high variance in the estimated state value, which can negatively impact the stability and convergence of the training processes. Despite this challenge, policy-based methods offer a direct and flexible approach to learning optimal policies.

Last, actor-critic methods combine the strengths of value-based and policy-based approaches by incorporating two distinct components: an actor and a critic. The actor learns the policy and makes action selections based on the current policy. At the same time, the critic acts like the actor's supervisor, evaluating their actions and providing feedback to guide the learning process toward better policies. This combination allows more efficient learning and improved decision-making in dynamic environments. These methods benefit continuous environments where actions are represented as real values by amortizing the variance in the state-value estimations.

Model-based and Model-free. Finally, another possible categorization in RL is whether the agent keeps a model of the environment throughout the learning process.

In model-based RL, the agent keeps an internal environment model representing the environment's dynamics. This model mimics the environment's responses to the agent's actions. The agent uses this model to simulate possible future trajectories and choose actions that lead to desirable outcomes. In contrast, in model-free methods, the agent directly learns a policy or value function without relying on an explicit model of the environment. Most techniques found in the literature primarily employ model-free approaches.

Model-based RL offers efficient planning and sample efficiency through model simulations. Also, if the model is accurate, it can be transferred and reused in similar environments. However, an inaccurate model highly

affects the quality of decision-making, leading to suboptimal and even catastrophic decisions. Moreover, building an accurate model is challenging and may require substantial computational resources. On the other hand, model-free approaches are simpler and safer to use as they avoid the model inaccuracy problem. Also, because no model exists, these methods can more easily be generalized and applied across different environments and tasks. Yet, model-free algorithms are usually more sample inefficient, requiring more interactions to learn an optimal policy. These methods may never reach the performance of a model-based approach with an accurate model. The decision-making is also less efficient as the lack of a model disables the agent's ability to plan.

Figure 2.5 presents a Venn diagram of the different RL methodologies. This diagram makes it clear that these RL categories may all overlap with each other. Actor-critic, value-based, and policy-based methods may or may not have a model, so that they may be model-free or model-based. In other words, retaining or not a model of the environment is an option that is completely independent of the method used, whether it is value-based, policy-based, or actor-critic.

Table 2.1 summarizes the benefits and limitations of the different types of RL and provides some notable examples of algorithms for each type.

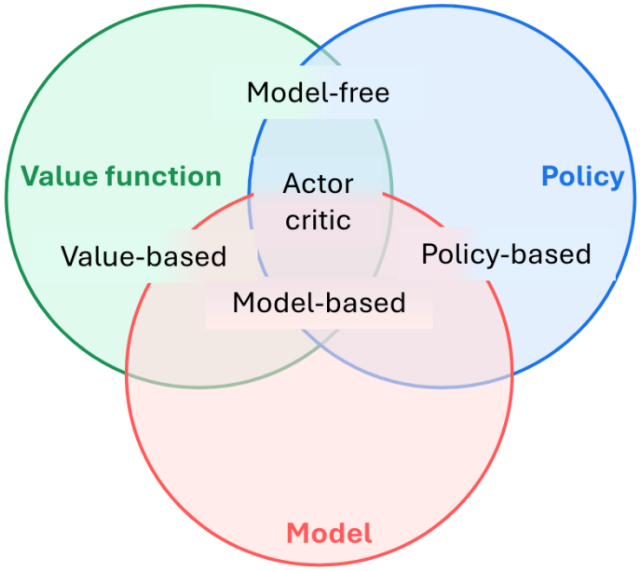


Figure 2.5: Different types of RL algorithms

Table 2.1 summarizes the benefits and limitations of the different types of RL and provides some notable examples of algorithms for each type.

Table 2.1: Comparison of RL algorithm types.

RL Category	Benefits	Limitations	Notable Algorithms
Value-based	Strong convergence properties	Struggle on high-dimensional and continuous action spaces	Q-learning, SARSA, DQN, DDQN
Policy-based	Efficient in large and continuous action spaces	Unstable and slow convergence due to high variance	REINFORCE, PPO, TRPO
Actor-critic	Improved stability	Increased complexity	DDPG, PPO, A2C, A3C, TD3, SAC

Model-free	Simpler to implement, with no need for an explicit model	Requires a large amount of interaction data, less sample-efficient	All of the above
Model-based	Sample efficient can plan ahead	Model inaccuracies can lead to poor performance, increased complexity	Dyna-Q, MuZero, World Models

Deep Q-Network

Deep Q-Network (**DQN**) represents a key RL agent architecture in our methodology. Leveraging a replay buffer, a target network, and a gradient clipping, DQN enables the network to learn from past experiences, enhancing decision-making and adapting to different network conditions. DQN employs the Q-learning concept, a technique aiming to learn a Q-function that assigns a value to each state-action pair to maximize cumulative reward over time. The Q-function is denoted as $Q(s,a;\theta)$, wherein s is the state, a is the action, and θ represents the weights of the DQN model neural network and has the form:

$$Q(s,a;\theta) = R_t + \gamma \max_{a'} Q\left(s', a'; \theta_{target}\right),$$

where R_t is the reward obtained after taking action a in state s at time step t, γ is the so-called discount factor, and the last term is the maximum predicted action value in the next state s', according to the target network. To mitigate the issue of sequential correlation in experience data, DQN uses a replay memory buffer. This buffer stores past experiences *(state, action, reward, new state)* and randomly samples mini-batches of experiences from the buffer during training. A target network is used to stabilize the network training. The target network is a copy of the main network that is periodically updated with the weights of the main network according to the following:

$$\theta_{target} \leftarrow (1-\alpha)\theta_{target} + \alpha\theta_{Q-net} \ ,$$

where θ_{target} are target network weights, $\theta_{Q\text{-}net}$ are the Q-network weights, and α is the target update interval. To ensure proper exploration of the environment, DQN can use strategies like ε-greedy, where the agent chooses a random action with probability ε instead of the action that maximizes the estimated Q-function. DQN training aims to improve the agent policy over time, letting it learn from past experiences and update its Q-function estimate. The DQN policy is deterministic: each state is directly mapped to an action without considering a probability distribution. Thence, DQN relies on estimating the action value (that is, Q), and the policy is implicitly defined as the action that maximizes this estimated action value. In contrast, models such as PPO and A2C handle stochastic policies.

Proximal Policy Optimization

Proximal Policy Optimization (**PPO**) is another crucial RL agent architecture we employ. Known for its stability and sample efficiency, PPO ensures robust learning by optimizing policies iteratively. Its role in the methodology is to enhance the adaptability of the network, especially in scenarios where dynamic adjustments are required to meet latency and performance targets. The policy returns a probability distribution over all possible actions for a specific state; namely, we have the distribution:

$$\pi_\theta(a_t|s_t),$$

representing the probability of selecting the action *in the state st, and the network parameters θ are optimized during training. Hence, parameters follow a direct policy optimization approach. During training, the goal is to directly improve the policy rather than focusing solely on estimating the action value. The* PPO algorithm is designed to ensure stability by avoiding excessively large updates in a single step.

In addition to the policy, PPO may involve a value function (state value) that estimates the value function of the state, helping to evaluate the goodness of the actions taken. This function is the following:

$$V(s) = \sum_{t=0}^{inf} \gamma_t \, R_{t+1},$$

where *V(s)* is the value of the state *s*, *γ* is the discount factor, and Rt+1 is the reward at time *t+1*. Moreover, a regularization term may involve a policy entropy, promoting diversity in the actions chosen by the agent. PPO computes a probability ratio between the new and old policies for each state-action pair, evaluating how much the new policy has improved compared to the previous one. A key element of PPO is advantage clipping, which limits the policy update within a range, avoiding excessive variations. For both PPO and A2C, the advantage is intended to be the difference between the reward obtained and the estimated value of the state.

Advantage Actor Critic

Advantage Actor Critic (**A2C**) is an actor-critic architecture that further strengthens our RL approach. With a focus on balancing exploration and exploitation, A2C facilitates the network's ability to make optimal decisions in real-time. The integration of A2C underscores our commitment to achieving a self-optimizing network that aligns with user and application demands. A2C algorithm combines elements of Advantage Learning (specifically Advantage Function) and Critic Learning. It is designed to train agents efficiently and help them make decisions in complex environments. A2C uses a policy (actor) that is a parameterized function mapping the environment states to probability distributions over actions. This policy represents the agent's behaviour. A2C uses a state value function (critic) that estimates the expected value of a state. This function helps evaluate the goodness of the actions taken.

The Advantage component of A2C comes from the Advantage Function, which measures how much better (or worse) an action is compared to the expectations of the value function. The agent's objective is to maximize the cumulative reward over time. The agent simultaneously learns an optimal policy (actor) and an accurate estimate of the value function (critic). A2C updates the model by using gradients calculated with respect to a combination of the loss from the policy (actor) and the loss from the value function (critic). During training, the agent interacts with the environment, collects experiences, and uses them to update its model.

3 ML architecture and system design

MLSysOps aims to manage a multitude of software systems and the underlying hardware infrastructure that hosts them.

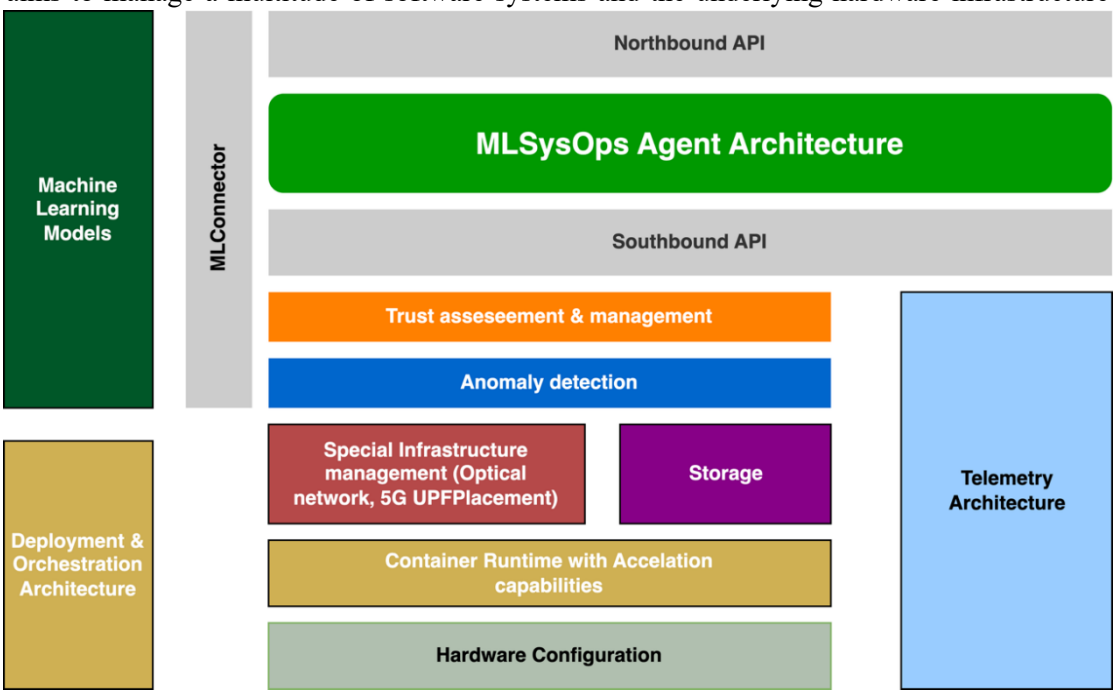


Figure 3.1 illustrates an abstract overview of the architecture. The framework's main building blocks are the agents and ML architecture. The different agents of the system interact with the rest of the environment with specific interfaces that are distinguished based on the entities that they interface with; 1) Northbound API: the system users (Application and System Administrator), 2) Southbound API: the various underlying software and hardware configurations and the telemetry system, 3) MLConnector: the plug-in ML models that help the agent make decisions.

The framework operates on the assumption that applications run within containers, with container management handled by the orchestration architecture across different levels of the slice continuum. Enhanced container runtime features offer application acceleration and an interface with the MLSysOps agent for efficient resource use. These agents can also adjust specific hardware settings.

This project also includes managing the 5G infrastructure and configuring the network of the datacenter cluster that trains large machine learning models. The MLSysOps framework treats these operations as special use cases with requirements. A similar approach applies to the storage component, which is differentiated as an infrastructure resource available for application usage.

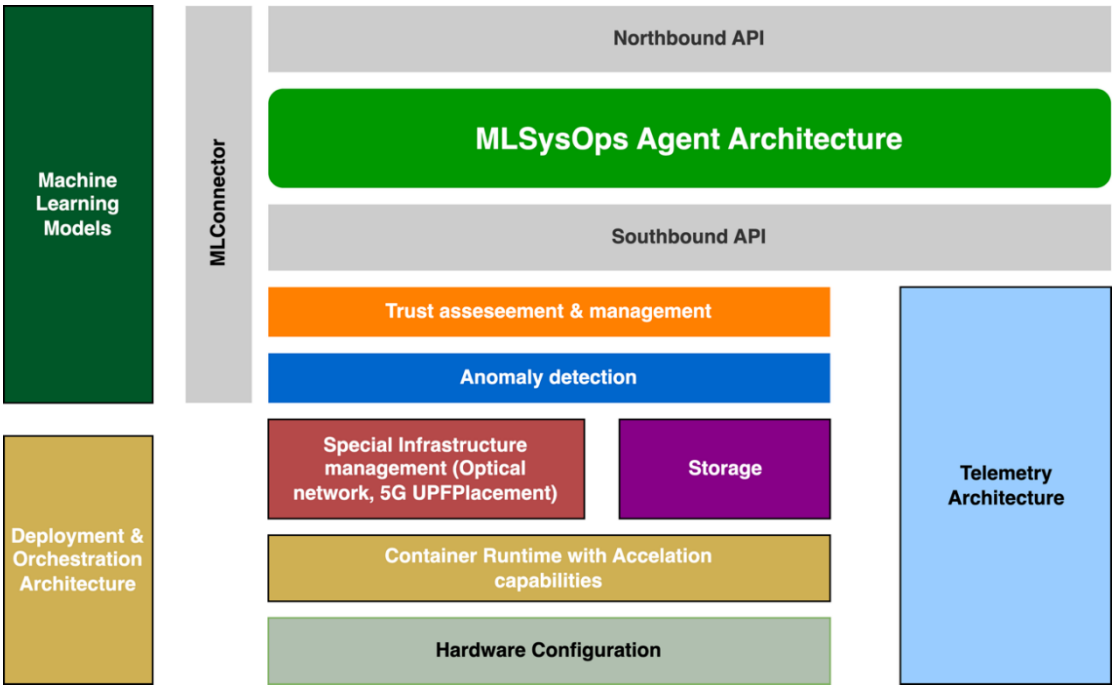


Figure 3.1: MLSysOps architecture

Additionally, the framework includes Anomaly Detection and Trust Assessment modules, which operate independently but use available interfaces to inform the MLSysOps agents. These modules can leverage ML models via the MLConnector to enhance their outcomes. The interactions between modules and the ML models are managed by the different types of agents that MLSysOps defines via the MLConnector. In the following section, we will delve into the different types of agents within the framework and their specific roles.

MLSysOps agents

In the MLSysOps framework, agents are indispensable, serving as the system's core intelligence. Agents at all levels shoulder several responsibilities, such as:

- **Data Interaction and Forwarding**: Agents are intermediaries for telemetry data collected from monitored computing nodes. They receive this data and perform essential tasks such as filtering, aggregating, and processing it. Subsequently, they provide this refined information as input to the selected machine learning model, enabling comprehensive analysis and informed decision-making.
- **High-Level Decision-Making**: Agents are responsible for making critical high-level decisions for the system, which includes carefully selecting and invoking the most appropriate ML model based on overall system optimization objectives and the unique agent architecture.
- **Action Consistency Assessment**: After receiving the ML model's output, agents assess whether the proposed action aligns with the application's specifications. Suppose the action deviates from the desired outcome or is invalid. In that case, agents promptly issue a negative feedback signal to the ML model, ensuring precise and effective decision-making within the system.
- **Transforming Information into Action**: Following the ML model's output, agents convert this information into actionable commands and optimization objectives. These directives are then directed towards external frameworks or are integrated into the lower levels of the agent's architecture.
- **Model Retraining Management**: Agents are tasked with initiating the retraining process of ML models upon detecting model drift. This action continuously adapts and enhances the ML model's performance over time.
- **Anomaly Detection and Trust Evaluation**: Agents assume control over anomaly detection mechanisms and trust evaluation processes, ensuring the system's reliability and robustness.

- **Responsiveness** to System Administrators' commands: Agents are designed to maintain responsiveness to signals from system administrators, allowing for the straightforward deactivation of ML with a simple human-initiated button press.
- **Transition to Conventional Techniques**: Agents facilitate a smooth and efficient transition to conventional optimization techniques, such as rule-based or heuristic approaches, when necessary.
- **Logging for Accountability**: Throughout these multifaceted responsibilities, agents maintain meticulous logs of their actions, complete with timestamps. This logging ensures accountability and transparency in their decision-making processes.

The MLSysOps framework defines three distinct types of agents. At the highest level, often referred to as the **continuum** or *entry-point* agent, the system begins with input information from an application scenario. The primary output at this stage comprises subgraphs, also known as clusters, which represent distinct application components and allocate the budget or resources required for running these clusters effectively. Moving down to the **cluster-level** or *infrastructure slice* agent, its role is to make crucial decisions regarding the nodes involved, the connections between them, and the equitable distribution of the budget. This includes allocating components to specific nodes, splitting the resource budget, and establishing the necessary connections. At the boundary of the infrastructure, **node-level** agents come into play with their responsibilities, which involve selecting the appropriate frequency and acceleration settings, selecting the type of hardware acceleration (GPUs, FPGAs, etc.) when available, and fine-tuning the system for optimal performance at a more granular level.

Figure 3.2 illustrates the internal architecture of the lowest-tier agent within the hierarchy, known as the Node Agent. Through the Southbound API interface, it receives telemetry and configuration information originating from the node on which it is installed. This information is then gathered by its Resource Collector entity. Subsequently, this data is transmitted in the form of Metrics to the Intent Target Module component, which addresses all the agent's objectives. Additionally, it is sent to the State Monitor component in the form of a Snapshot.

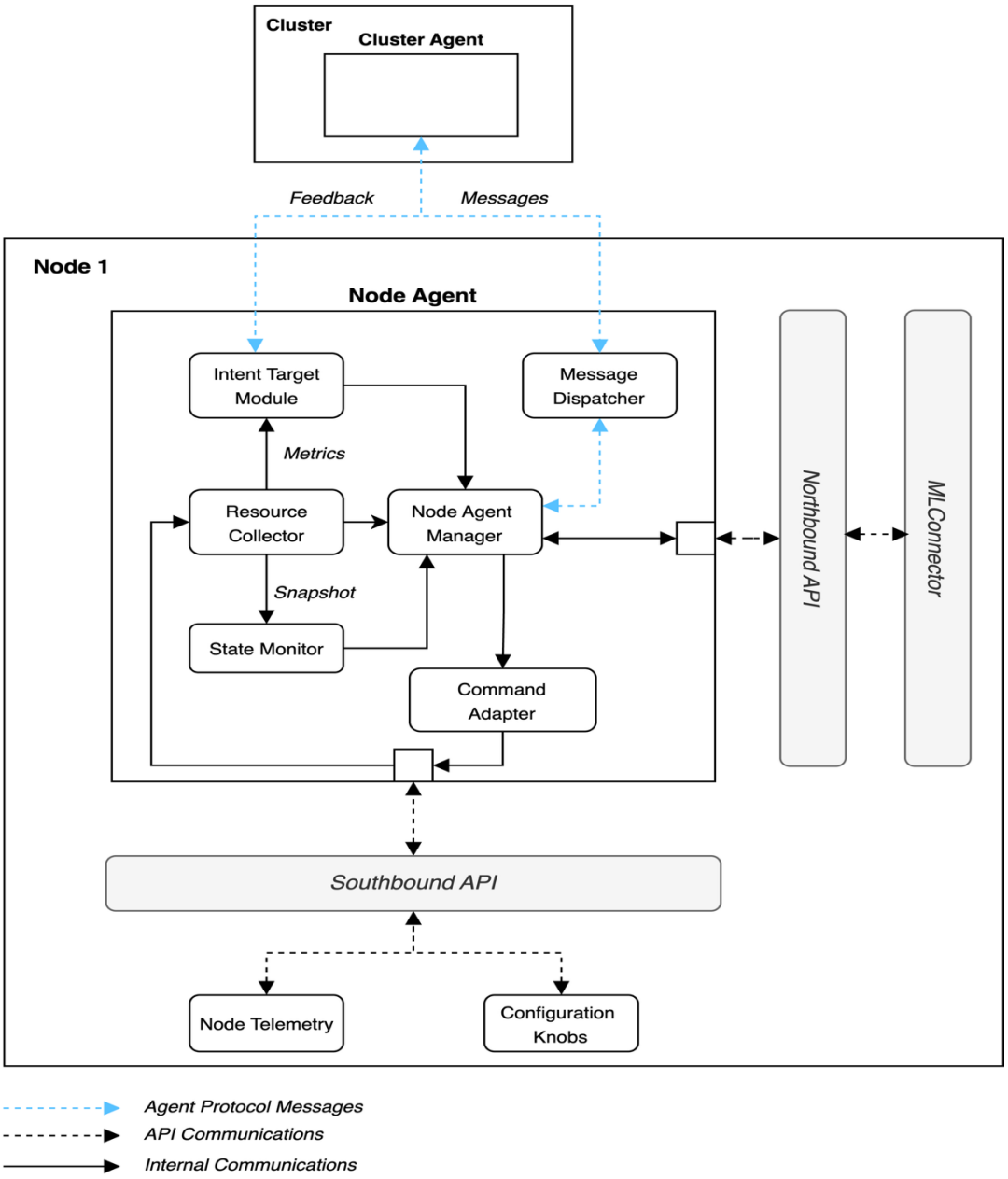


Figure 3.2: Node-level agent architecture

At the core of the Node Agent lies the Node Agent Manager, which serves as the central hub for consolidating information from the entities mentioned above and transmitting it to ML model services via the MLConnector interface. The latter can establish direct connections with local ML models installed on the physical computing node or access the ML-as-a-Service (MLaaS) component, which offers the latest ML models refined through reinforcement learning. The Node Agent Manager also efficiently oversees bidirectional communication with other agents within the hierarchy, thanks to the Message Dispatcher component. The outputs generated by the ML service are further processed and relayed to the Command Adapter component, which is tasked with converting these outputs into actionable commands and optimization objectives. The Node Agent Manager follows a predefined default behaviour if the ML service becomes unreachable or deactivated. In such cases, the Node Agent Manager directs actions to the Command Adapter component. Finally, the Command Adapter component forwards the appropriate commands to the remaining node components via the Southbound API interface.

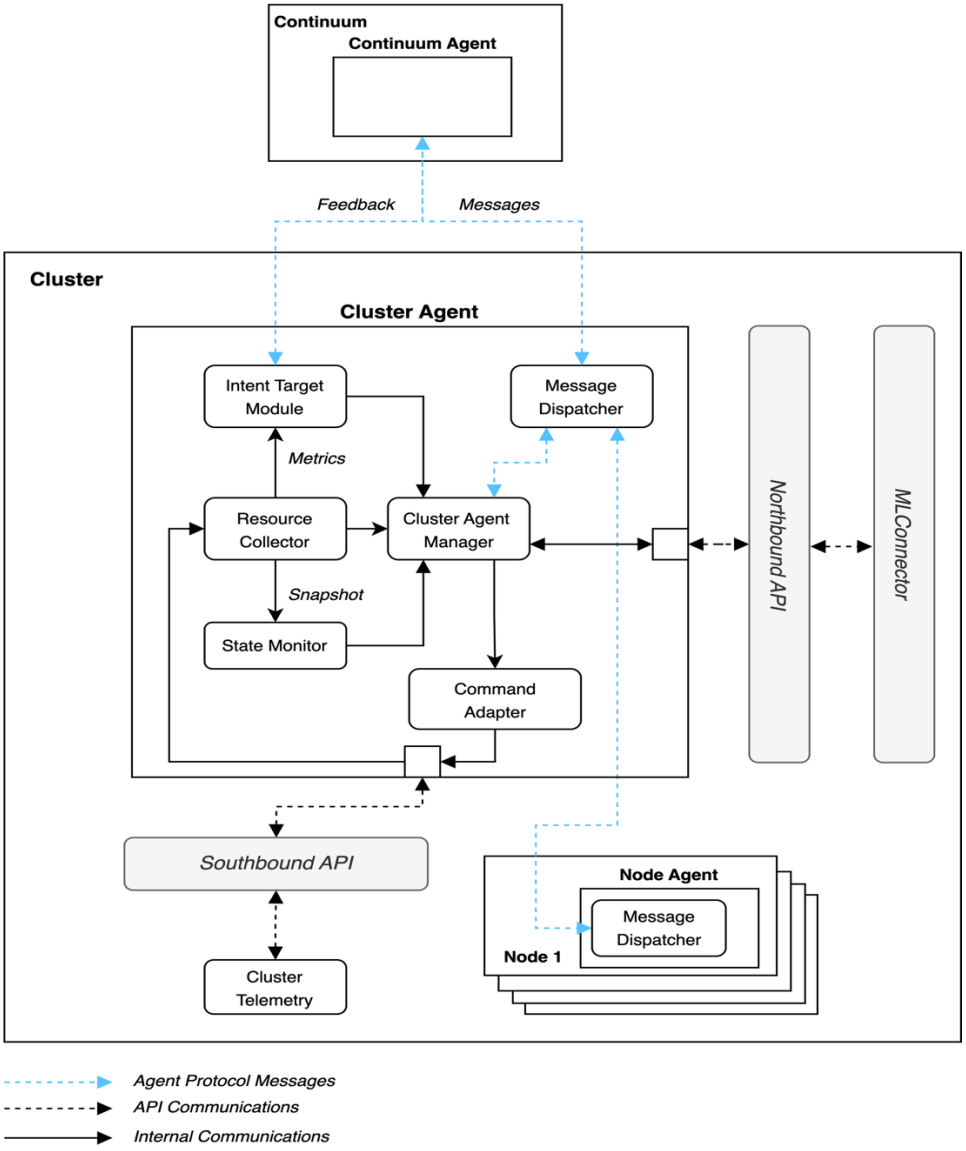


Figure 3.3: Cluster-level agent architecture

The internal architecture of the Cluster Agent, as illustrated in Figure 3.3 closely resembles that of the Node Agent. The Cluster Agent receives cluster telemetry data via the Southbound API interface, which subsequently routes it to its internal components. Within this framework, the Cluster Agent Manager assumes a pivotal role, collecting information from the Resource Collector, Intent Target Module, State Monitor, and Message Dispatcher and transmitting it to ML services through the MLConnector interface when available and connected. In the absence of the latter, the Cluster Agent Manager defaults to its standard behaviour, akin to the Node Agent. The command Adapter component also plays a critical role by converting the outputs from the Cluster Agent Manager into actionable commands and optimization objectives. It is important to note that each cluster corresponds to a set of nodes. These nodes have their Node agents, which communicate with the Cluster Agent using a predefined agent communication protocol, using their Message Dispatcher to ensure seamless interaction.

The Continuum Agent (Figure 3.4) is positioned at the highest level within the hierarchy as a pivotal communication hub connecting with all external actors through a bidirectional Northbound API interface. Handling incoming and outgoing requests and responses is the responsibility of the External Connector API. Furthermore, this API conveys the initial intents guiding the entire system to the Intent Target Module. The telemetry data, collected as Continuum Telemetry, enters the Continuum agents via the Southbound API interface. The information generated by all internal components of the Agent is then consolidated by the

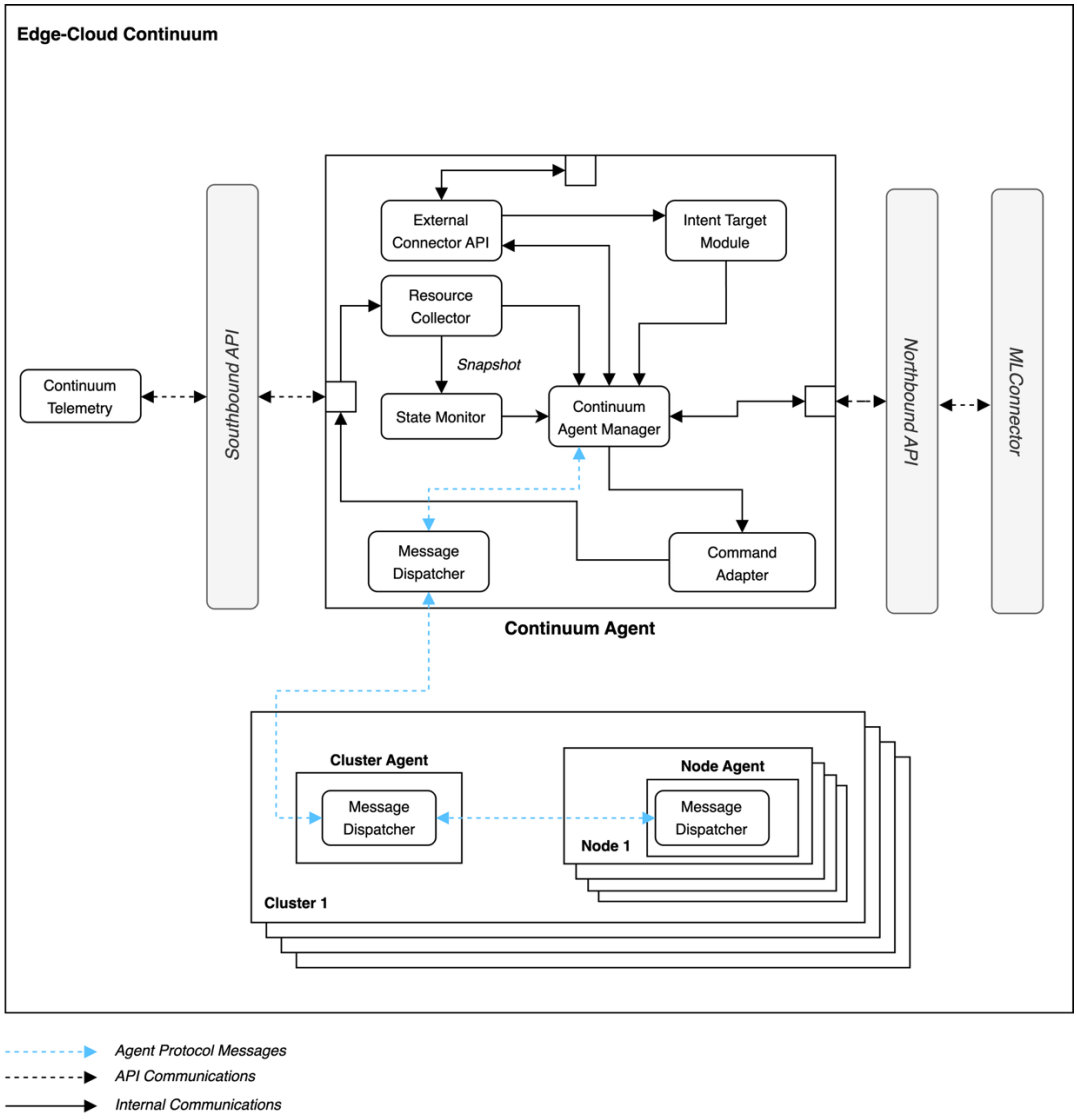


Figure 3.4: Continuum-level agent architecture

Continuum Agent Manager, which, in turn, conveys it to ML services, should they be available and active via the MLConnector interface or default to standard behaviour when unavailable. The resulting output action is subsequently translated into a command by the Command Adapter component and relayed to the entirety of Clusters and, consequently, Nodes throughout the system. The Continuum Agent engages with all agents within the hierarchy through a dedicated inter-agent communication protocol facilitated by the Message Dispatcher component.

System design

As described above, the MLSysOps framework defines distinct modules, each performing different roles. For seamless integration, we define different ML approaches to aid the decision process for each module. The following sections discuss each module's ML approach in detail. The goal of this section is to introduce the overall system design, each module, its relationship with the different MLSysOps agents, and how they interact with the ML repository. Figure 3.5 summarises the high-level system design.

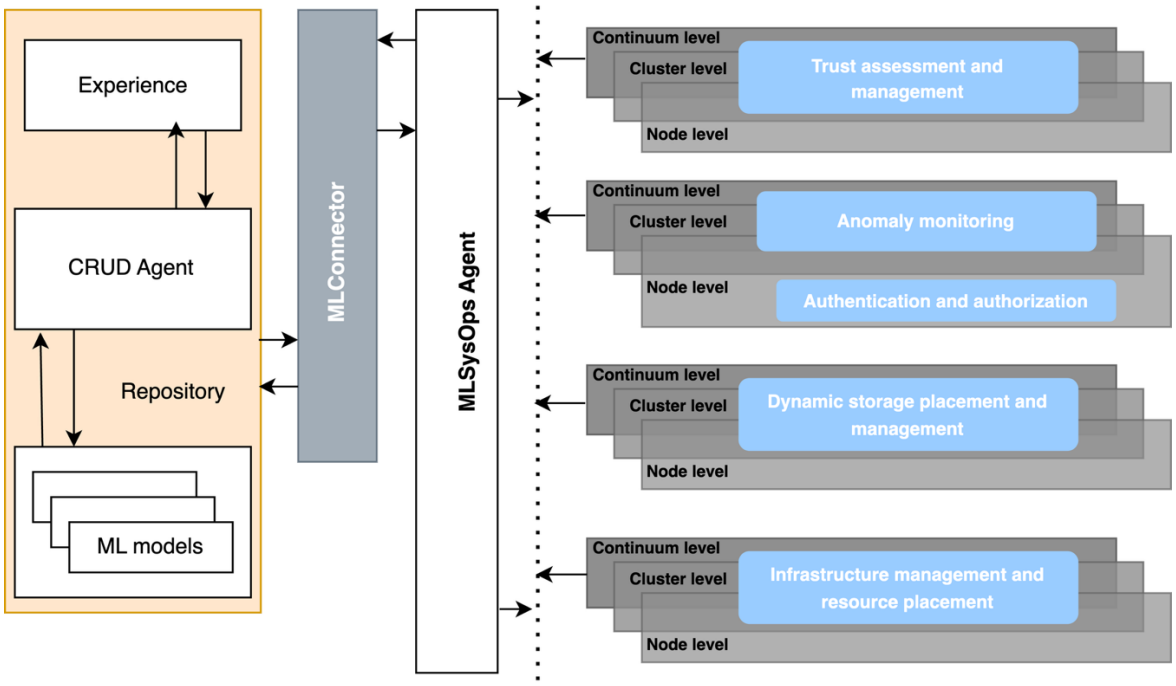


Figure 3.5: High-level ML system design

- ***Trust assessment and management*:** The trust assessment module is based on reputation and trust evaluation models. The system evaluates the identity, capability, and behaviour trust of nodes in the MLSysOps network. The behaviour trust evaluation uses a reputation-based trust evaluation model, which considers direct trust, indirect trust, and the rater's credibility to calculate and update the reputation of service providers.
- ***Device authentication, authorization, and anomaly detection*:** This module performs two core roles; Firstly, it defines a novel hybrid Physical Layer Authentication (PLA) scheme to distinguish between authenticated devices and malicious ones based on features extracted from the physical layer. This PLA scheme is designed to enhance the security of wireless communications by leveraging the unique characteristics of the physical layer, making it applicable to various communication technologies with limited adjustments. This operates at the node level. The other component of the module is anomaly detection. This continuously monitors all the nodes at different levels (node, cluster, and continuum) to identify attacks or system failures and distinguish between them. A re-allocation of the resources can follow in identifying such an event. The anomaly/attack detection functionality is installed on an independent node, which is able to receive all the data from the far-edge devices, extract the relevant features, and perform the detection process.
- ***Dynamic storage placement and management*:** This module offers a distributed object storage service for applications via an HTTP interface. It is offered via an API compatible with Amazon Simple Storage Service (S3), the de facto object storage interface with SDKs available in all major languages. Under the hood, the module extends the SkyFlok Secure Distributed Storage service and encompasses all three different-level agents.
- ***Infrastructure management and resource placement*:** This module performs various roles across all the three-agent levels (node, cluster and continuum level). It includes mechanisms that determine the management of resources and application execution at the initial deployment and during runtime. In each case, the cluster-level mechanism determines the placement of application components to one or more nodes in the platform. Then, the node-level mechanism is used to configure the node to optimize performance and reduce power dissipation.

4 Trust assessment and anomaly detection

The OWAD Mechanism

Open-World Anomaly Detection (OWAD) is a mechanism designed to address the limitations of traditional anomaly detection methods in dynamic, real-world environments. Unlike conventional approaches that assume a static definition of "normal" behavior based on initial training data, OWAD recognizes that normality can evolve over time due to factors such as changing user behavior, system updates, environmental shifts, or the introduction of new operational patterns.

The OWAD introduces a closed-loop system that continuously monitors, interprets, and adapts to these shifts. It integrates three key components: shift detection, which identifies when and how normality has changed; shift explanation, which pinpoints the specific features or patterns driving the change; and shift adaptation, which updates the anomaly detection model to align with the new normality while preserving prior knowledge. This mechanism works as follows (as shown in Figure 4.1):

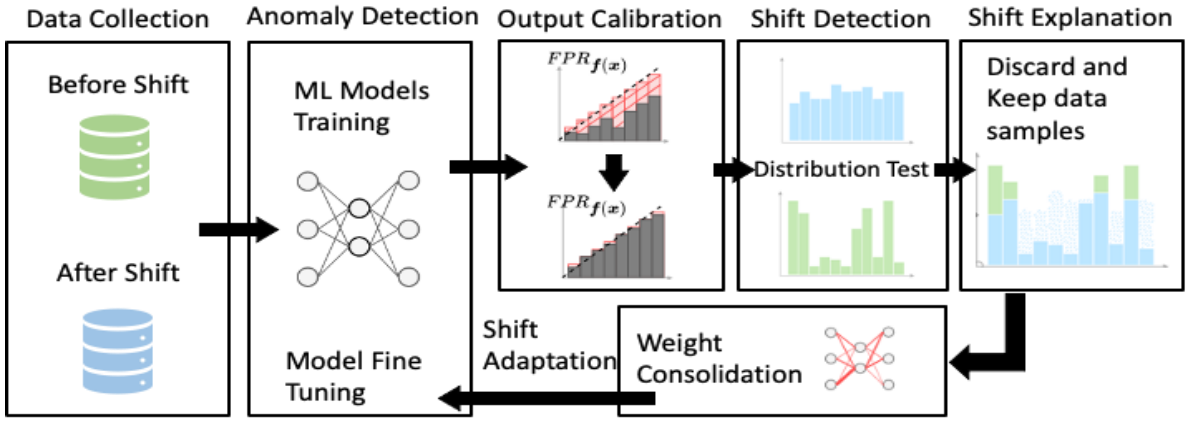


Figure 4.1: Architecture of the OWAD module

- **Data Collection:** Gathering relevant datasets that accurately reflect both normal and abnormal behavior in a system. The goal is to capture a comprehensive range of operational scenarios so that the model can learn what constitutes typical behavior and identify deviations from it. Proper labeling of known anomalies can significantly enhance the performance of supervised or semi-supervised detection models.
- **Anomaly Detection:** An initial model is trained on a dataset representing the presumed "normal" state. This model continuously processes incoming data streams, generating anomaly scores or predictions for each new data point. These scores indicate the degree of deviation from the learned normality pattern established during training.
- **Output Calibration:** The raw anomaly scores produced by the detection model undergo a transformation process. This step often involves converting these scores into calibrated probabilities or confidence estimates. Calibration techniques ensure that the scores are interpretable and comparable over time, providing a more reliable basis for subsequent drift analysis.
- **Shift Detection:** Statistical methods and change point detection algorithms are applied to monitor the stream of calibrated anomaly scores or underlying model confidence metrics over time. This continuous monitoring analyzes the distribution of these outputs, searching for significant deviations or trends that indicate a fundamental change in the characteristics of the incoming data deemed "normal" by the current model.
- **Shift Explanation:** Upon detection of a significant shift, techniques from Explainable AI (XAI) are employed. These techniques analyze the internal state of the anomaly detection model, the features of

the data triggering the shift signal, or the distribution of data before and after the detected change point. The goal is to identify *which* features, concepts, or data subspaces are primarily responsible for the observed normality shift.

- **Shift Adaptation:** Based on the insights gained from the shift explanation, the anomaly detection model undergoes modification. This adaptation leverages newly arrived data (often assumed to represent the new normality after the shift). It might involve strategies like incremental learning updates to model parameters, adjustment of decision thresholds, selective retraining focused on the features identified as shifted, or incorporating the explanation to guide safe model updates without catastrophic forgetting of previous knowledge. The adapted model then resumes anomaly detection on the evolving data stream.

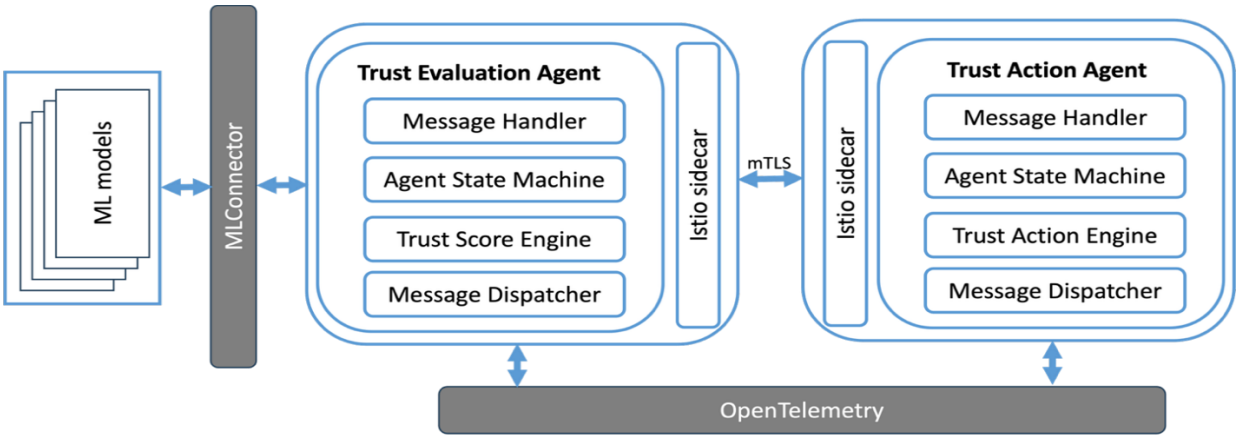


Figure 4.2: Architecture of the Trust assessment and anomaly detection Framework.

The trust assessment and anomaly detection system features two primary agents operating in tandem: a Trust Evaluation Agent and a Trust Action Agent supported by MLConnector, as presented in Figure 4.2.

Each agent follows a consistent internal workflow. Incoming messages are first processed by a Message Handler, which parses and routes inputs. The Agent State Machine then dynamically manages the agent's operational state, transitioning between phases like initialization, processing, and response. For the Trust Evaluation Agent, the core logic resides in the Trust Score Engine, which calculates trust metrics based on predefined algorithms or data analysis. Conversely, the Trust Action Agent relies on its Trust Action Engine to execute decisions—such as granting access or triggering alerts—based on trust evaluations. Finally, both agents utilize a Message Dispatcher to propagate results, alerts, or requests to external systems or other components.

The MLConnector operates independently, likely serving auxiliary functions like data provisioning (e.g., user history logs), system configuration, or integration with external services. Together, these elements form a cohesive trust-management ecosystem where evaluation and action are decoupled yet interoperable, enabling scalable and modular security or decision-making workflows.

Data Collection and Anomaly Data Generation

For the data collection, the hardware utilized consisted of two Raspberry Pi 5 single-board computers, each equipped with 8 GB of RAM. These compact yet capable devices served as the core data logging platforms, specifically configured to capture and record telemetry data streams. Leveraging the enhanced processing power and improved I/O capabilities of the Raspberry Pi 5 model, the units were deployed to continuously gather sensor readings and system metrics relevant to the experimental parameters. Their integrated connectivity options, including Gigabit Ethernet, dual-band Wi-Fi, and Bluetooth, facilitated both local sensor interfacing and remote data transmission during the collection process, ensuring robust and efficient acquisition of the required telemetry information.

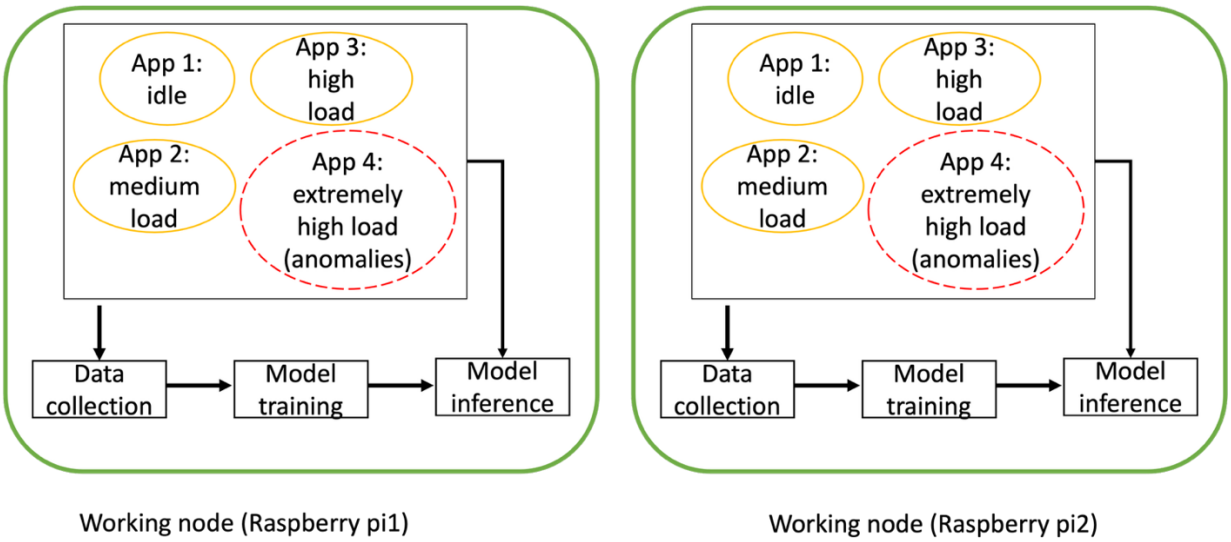


Figure 4.3: Data Collection Framework

The process depicted in Figure 4.3 begins with the simulation of four distinct application workload states on two independent Raspberry Pi 5 (8GB) devices. The stress-ng tool was employed to precisely generate controlled and reproducible levels of stress on the CPU and memory subsystems of each Pi. This specialized utility is designed specifically for loading and stressing computer systems in a configurable manner, making it ideal for simulating the target workload conditions: Application 1 simulated an idle condition, Application 2 a medium load, Application 3 a high load, and Application 4 an extremely high load incorporating anomalies. During this stress-ng-driven simulation phase on both devices, comprehensive telemetry data was collected, capturing detailed system behavior under these defined stress profiles. This collected telemetry dataset subsequently served as the input for the model training phase, where an analytical model was developed to recognize patterns and potential anomalies associated with the different load levels. Finally, the trained model was deployed for inference on the Raspberry Pi platforms, where its performance in identifying and classifying the simulated application states (idle, medium load, high load, and extremely high load with anomalies) was evaluated, again utilizing stress-ng to generate comparable load conditions. The entire sequence, workload simulation (via stress-ng) and telemetry collection, model training, and model inference, is then repeated for a second distinct experimental cycle, as indicated by the duplicated structure in the diagram.

The collected telemetry features provide granular insights into CPU utilization and memory behavior on the Raspberry Pi system. CPU metrics are captured per individual core (cores 0 through 3), detailing the cumulative time spent by each core in distinct operational states: idle (inactive), user (executing user application code), system (running kernel code), nice (processing low-priority user tasks), iowait (idle while waiting for I/O operations to complete), irq (servicing hardware interrupts), softirq (handling deferred interrupt processing), and steal (time lost in virtualized environments due to hypervisor servicing other virtual machines). Memory metrics offer a comprehensive view of system RAM usage, including the absolute memory_used_bytes (calculated as total memory minus free, buffers, and cached memory), the fundamental node_memory_MemTotal_bytes (total physical RAM available), and node_memory_MemFree_bytes (completely unused RAM). Crucially, the data also tracks node_memory_MemAvailable_bytes, an estimation of memory readily usable by new applications, incorporating reclaimable cache. Furthermore, kernel-level memory optimizations are reflected in node_memory_Buffers_bytes (temporary storage for raw disk blocks) and node_memory_Cached_bytes (the page cache holding frequently accessed file data for accelerated I/O performance). Collectively, these features form a detailed profile of system resource consumption under varying load conditions induced by stress-ng, enabling the analysis of CPU core balancing, workload distribution, I/O bottlenecks, memory pressure, and cache effectiveness, which are essential inputs for training and evaluating the anomaly detection model.

Anomaly Data Augmentation

The process begins by loading CSV files representing all combinations of CPU load (idle, medium, high) and memory load (idle, medium, high). For all states, the system calculates two key statistical parameters: the mean vector capturing central tendencies of all features, and the covariance matrix describing how these features vary together. When generating new data sequences, the system alternates between stable operational segments and transitional phases. For stable segments, it randomly samples directly from the original CSV data corresponding to a specific CPU-memory load combination. The length of these stable segments varies randomly between user-defined minimum and maximum values. When switching between different operational states within these stable segments, the system generates transitional data points using Gaussian interpolation between these stable segments.

For each transition point, the system calculates weighted averages of the mean vectors and covariance matrices from the departing and arriving operational states. It then samples from this interpolated multivariate Gaussian distribution to create new data points that smoothly transition between states. These generated points represent hybrid operational conditions that didn't exist in the original datasets but emerge naturally during state transitions in complex systems. We further transform this simulation into a more effective anomaly detection benchmark. During the transition phases, when the system currently performs mathematical interpolation, we could introduce deliberate anomaly injections. These would simulate real-world failure modes like sudden value spikes where a randomly selected feature gets multiplied by an abnormal factor, gradual sensor drift patterns that slowly deviate from expected values, or stuck-value scenarios where measurements become frozen for several consecutive observations.

Model training and Results

The anomaly detection process was implemented using an autoencoder neural network model trained on normalized telemetry data. The methodology began with data preparation, where multiple CSV files containing different CPU and memory load combinations were concatenated to form a comprehensive training dataset. The data was normalized using Min-Max scaling based on the parameters calculated from the training set to ensure all features were on a consistent scale between 0 and 1.

The autoencoder architecture (see Figure 4.4) consisted of a symmetric encoder-decoder structure with gradually decreasing and then increasing layer sizes, using ReLU activation functions. The encoder compressed the input features through four layers (reducing dimensions to 75%, 50%, 25%, and finally 10% of the original size), while the decoder mirrored this structure to reconstruct the original input. The model was trained using the Adam optimizer with mean squared error (MSE) as the loss function, which measures the reconstruction error between the input and output.

During testing, the trained autoencoder processed new data samples and calculated their reconstruction errors. The anomaly score for each sample was determined by computing the root mean squared error (RMSE) of its reconstruction. A threshold for anomaly detection was established based on the 99th percentile of reconstruction errors observed during training. Any test sample with an RMSE score exceeding this threshold was classified as an anomaly.

The evaluation process included loading known anomaly indices from a JSON file containing ground truth information about injected anomalies in the test data. Performance metrics such as accuracy, precision, recall, and F1 score were calculated by comparing the autoencoder's predictions against these known anomalies. The results were visualized in a comprehensive plot showing the anomaly scores of all test samples, with true anomalies highlighted in red and normal samples in blue, along with the decision threshold line. This visualization provided clear insight into the model's detection capability and the distribution of anomaly scores across the test dataset.

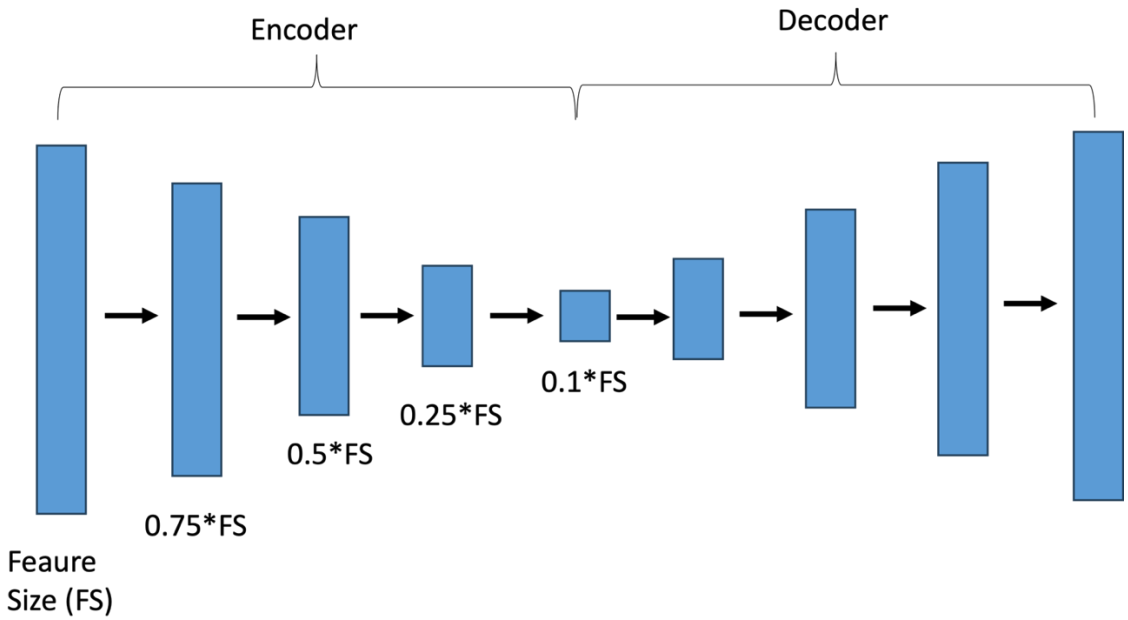


Figure 4.4: Autoencoder Architecture

The results demonstrate the performance of the autoencoder-based anomaly detection system on the test dataset of 2,000 samples. The model achieved an outstanding overall accuracy of 99.65%, correctly classifying both normal and anomalous samples with near-perfect precision. With a precision of 99.42% and a recall of 99.57%, the system showed remarkable balance in its ability to identify true anomalies while minimizing false alarms, as evidenced by the F1 score of 99.49%. The confusion matrix reveals only 4 false positives (normal samples incorrectly flagged as anomalies) and 3 false negatives (undetected anomalies) among the 692 known anomalies in the dataset, indicating highly reliable detection capabilities.

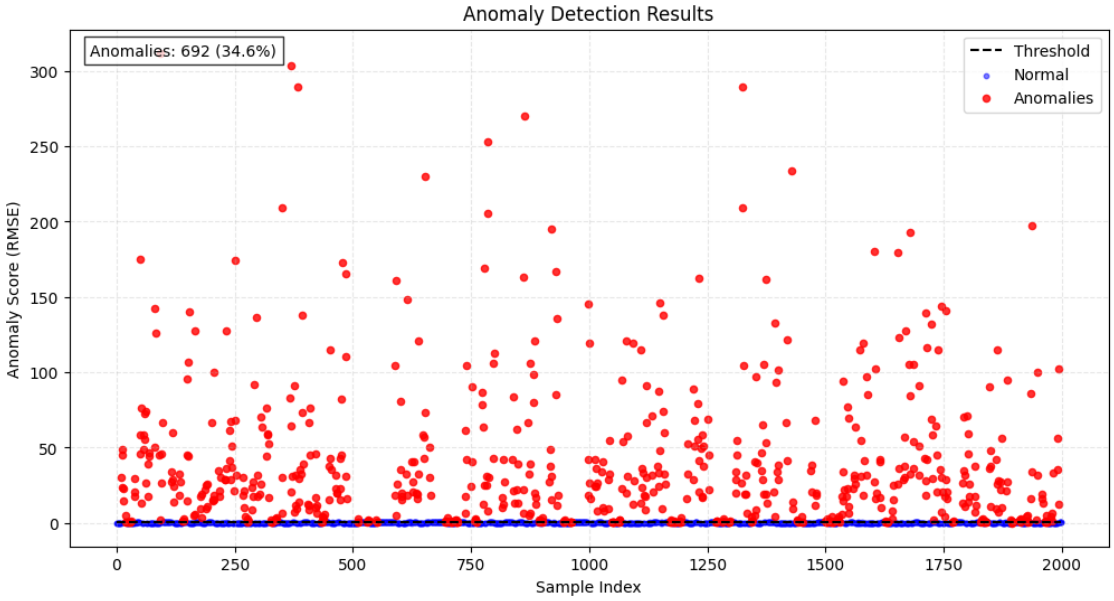


Figure 4.5: Anomaly Detection results

The anomaly scores in Figure 4.5 show a clear separation between normal and anomalous samples relative to the threshold of 0.253, with anomalies accounting for 34.6% of the test data. The classification results further confirm the model's consistent performance across both classes, achieving perfect 1.00 scores for normal samples and near-perfect 0.99 scores for the anomaly class in all metrics. These results validate the effectiveness

of the autoencoder architecture and the chosen thresholding strategy for detecting anomalies in the telemetry data, demonstrating both high sensitivity to abnormal patterns and strong specificity in maintaining low false alarm rates. The minor classification errors (0.35% of cases) primarily represent borderline cases where anomaly scores hovered near the decision boundary, suggesting the model operates with robust discriminative power across the entire dataset.

5 Anomaly detection using a novel hybrid physical layer authentication scheme

The proliferation of wireless networks and the critical nature of their applications have made security a paramount concern, particularly in detecting and preventing unauthorized access attempts. Our research within the MLSysOps project has developed a novel hybrid physical layer authentication (PLA) scheme that addresses the fundamental challenge of distinguishing between legitimate devices and potential security threats in multi-node networks. This work introduces an innovative approach that leverages inherent hardware impairments—specifically Carrier Frequency Offset (CFO), Direct Current Offset (DCO), and Phase Offset (PO)—combined with machine learning techniques to create a robust authentication framework capable of detecting anomalous behavior without prior knowledge of malicious device characteristics.

Architectural Design

To provide a clear, high-level view of this integrated strategy, Figure 5.1 illustrates how these security functions act as cohesive components within the broader MLSysOps ecosystem. The diagram depicts the ML Agents processing local data at the edge and communicating with the central MLSysOps platform via an MLConnector interface. This facilitates a critical operational loop: the agents forward security alerts and performance metrics for central analysis. At the same time, the platform deploys updated models to the agents, ensuring they remain robust and effective against evolving threats.

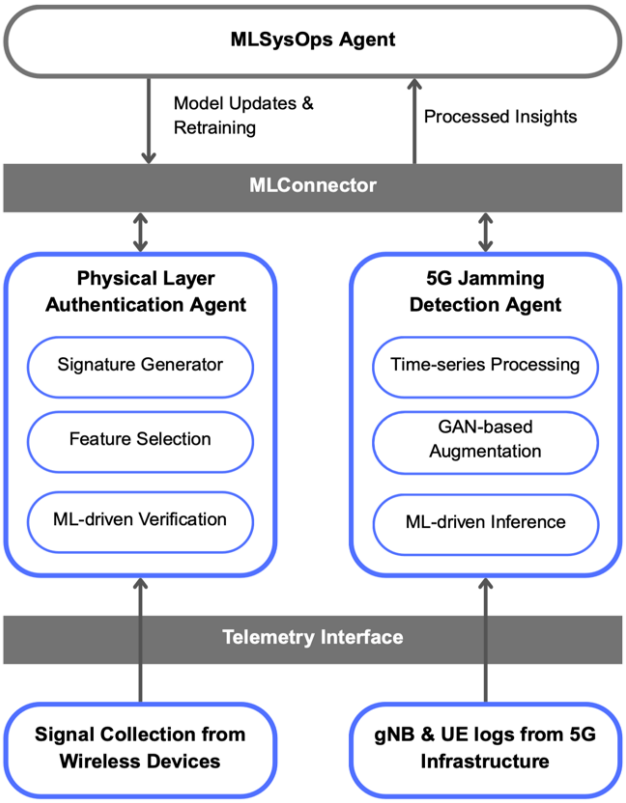


Figure 5.1: High-level Architecture of Wireless Anomaly Detection Agents

The Challenge: Securing Multi-Node Networks Against Sophisticated Threats

Traditional authentication mechanisms in wireless networks face increasing challenges as attack vectors become more sophisticated and networks grow more complex. Existing approaches typically suffer from several critical limitations:

- **Reliance on cryptographic methods alone**, which can be compromised through key distribution vulnerabilities and are computationally intensive for resource-constrained devices.

- **Channel-based authentication schemes** that depend on environmental factors like Received Signal Strength (RSS), Channel Impulse Response (CIR), or Channel Frequency Response (CFR), making them susceptible to variations in multipath fading and environmental changes.
- **Single-attribute authentication approaches** that fail to provide sufficient discrimination between devices, particularly when hardware characteristics overlap.
- **Limited scalability** in multi-node environments where the number of potential connections grows exponentially with network size.

To address these fundamental gaps, we developed a hybrid PLA scheme that synergistically combines multiple hardware impairments to create unique device signatures while employing advanced machine learning models for robust classification. For a comprehensive view of our authentication framework, Figure 5.2 illustrates the complete pipeline from signal collection through final device verification.

Core Innovation: Multi-Attribute Hardware Impairment Exploitation

Our scheme's foundation lies in recognizing that every wireless device exhibits unique hardware imperfections that manifest as consistent, measurable deviations in transmitted signals. Unlike channel-based features that fluctuate with environmental conditions, these hardware-induced signatures remain stable over time, providing reliable authentication markers.

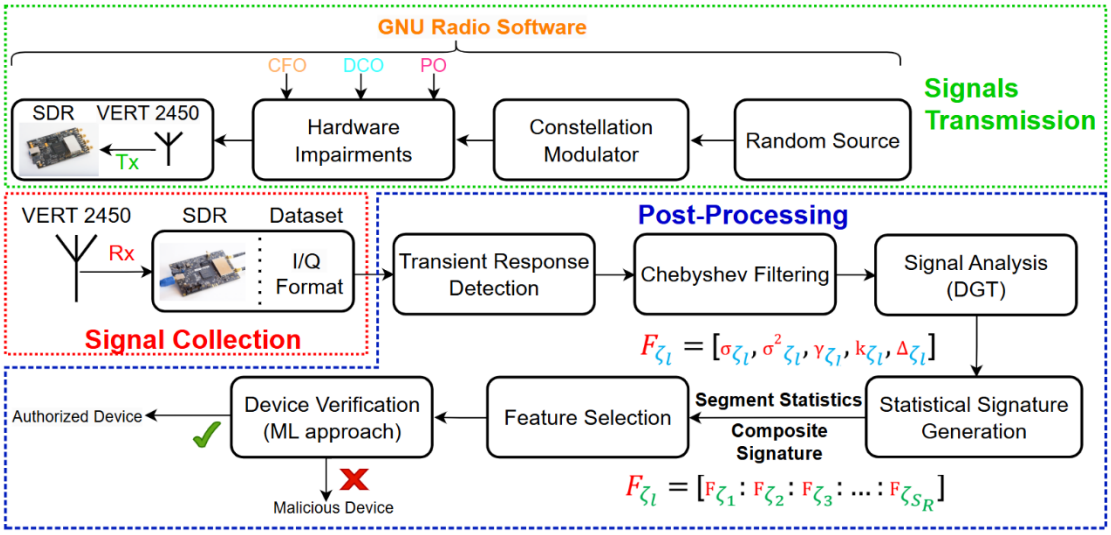


Figure 5.2: PHY authentication framework

5.1.1 Hardware Impairment Characteristics

The authentication framework exploits three primary hardware impairments:

1. **Carrier Frequency Offset (CFO)**: CFO arises from oscillator inaccuracies between transmitter and receiver and creates distinctive frequency shifts that vary between devices. Our measurements show CFO values ranging from 50 kHz to 200 kHz across different devices, with each maintaining consistent offsets under varying conditions.
2. **Direct Current Offset (DCO)**: Caused by mixer imbalances and ADC imperfections, DCO superimposes unique DC components on the transmitted signal. We observed DCO levels from 10 mV to 100 mV, providing additional discriminative features.
3. **Phase Offset (PO)**: Resulting from phase-locked loop variations and circuit mismatches, PO introduces consistent phase errors ranging from 11.5° to 45° in our experimental setup.

Figure 5.3 demonstrates the distinctive constellation patterns created by these impairments, highlighting their potential for device differentiation.

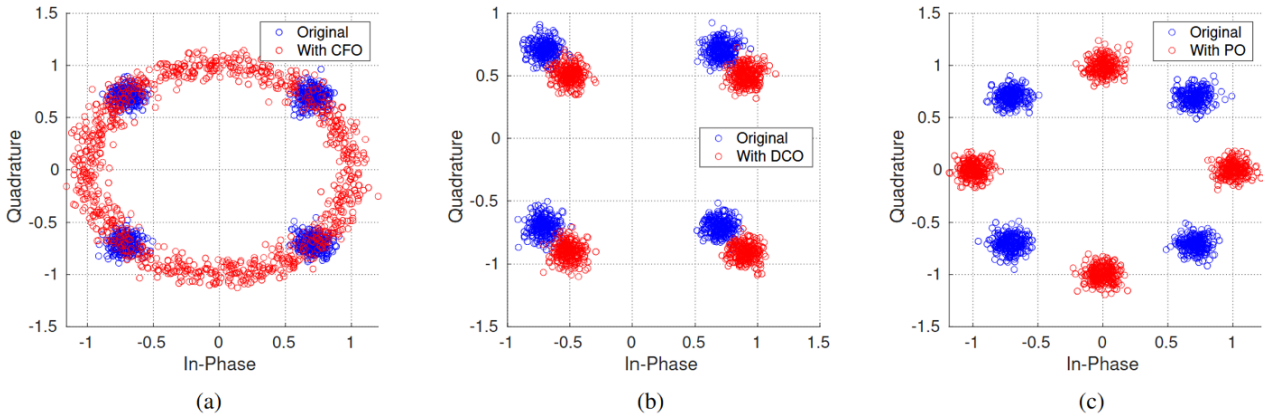


Figure 5.3: Effect of hardware impairment on RF data (a) CFO effect (b) DCO effect (c) PO effect

5.1.2 Advanced Signal Processing Pipeline

Our authentication process employs a sophisticated signal processing pipeline designed to extract and enhance these hardware signatures:

- **Transient Detection and Filtering**: We focus on the burst transient phase—the initial transmission period when a device transitions from idle to active state. This region contains the most distinctive hardware characteristics. Using amplitude-based Variance Trajectory detection, we precisely isolate these transients and apply a 4th-order Chebyshev filter to enhance signal quality while preserving authentication features.
- **Feature Generation via Discrete Gabor Transform**: The filtered signal is transformed using the Discrete Gabor Transform (DGT), which provides simultaneous time-frequency analysis. This two-dimensional approach captures both instantaneous and localized variations, revealing patterns invisible in traditional time or frequency domain analysis alone.
- **Statistical Feature Extraction**: From the DGT coefficients, we extract five statistical features per segment (S): standard deviation (σ), variance (σ^2), skewness (γ), kurtosis (k), and entropy (Δ). These metrics capture the unique distribution characteristics induced by hardware impairments.

Figure 5.4 illustrates the feature generation process, showing how the time-frequency surface is segmented and processed to create comprehensive device signatures.

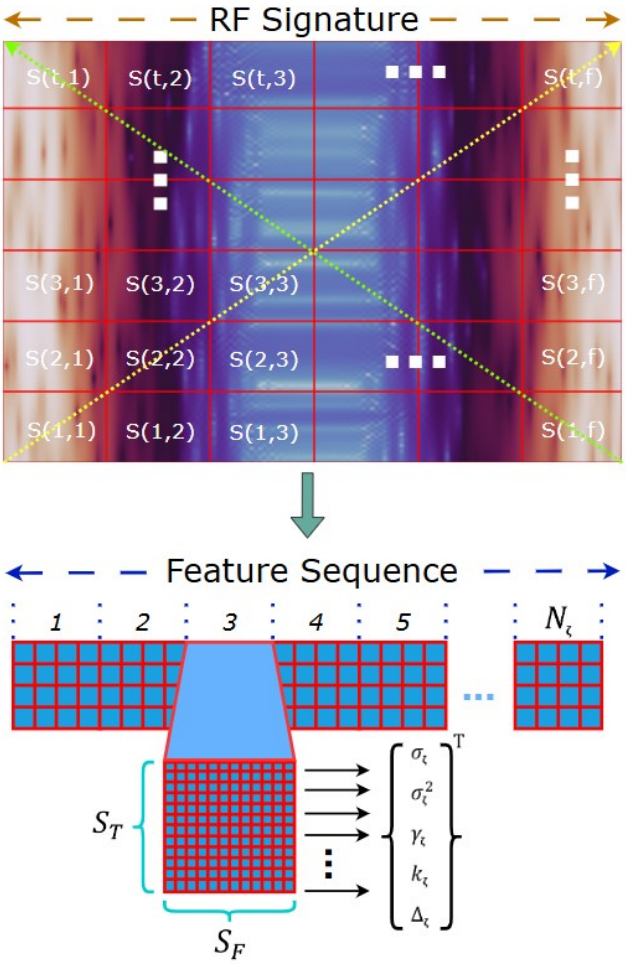


Figure 5.4: Feature generation using DGT approach

5.1.3 Machine Learning Integration for Robust Authentication

The extracted features serve as input to carefully selected machine learning models, each chosen for specific strengths in handling the authentication challenge. Our framework evaluates five different classifiers:

- **Random Forest (RnF)** and **XGBoost (XGB)**: These ensemble methods excel at capturing complex, non-linear relationships between hardware impairments, making them particularly effective for detecting subtle authentication patterns.
- **Support Vector Machines (SVM)**: Leveraging their ability to find optimal hyperplanes in high-dimensional spaces, SVMs provide fine-grained separation between legitimate and malicious devices.
- **Logistic Regression (LR)**: Offering probabilistic outputs essential for threshold-based authentication decisions, LR provides interpretable results while maintaining computational efficiency.
- **K-Nearest Neighbors (KNN)**: This intuitive approach excels when device relationships follow clear geometric patterns in the feature space.

To optimize feature selection, we employ four different approaches: Mutual Information (MI), Analysis of Variance (ANOVA), Principal Component Analysis (PCA), and Recursive Feature Elimination (RFE). This comprehensive evaluation ensures optimal performance across diverse network conditions.

5.1.4 Experimental Validation and Performance Analysis

Our experimental setup, depicted in Figure 5.5 uses a commercial BladeRF 2.0 Micro xA4 SDRs operating at 5 GHz with a 20 MS/s sampling rate. We created 12 distinct device profiles by systematically varying hardware impairment levels, simulating a diverse multi-node network environment.

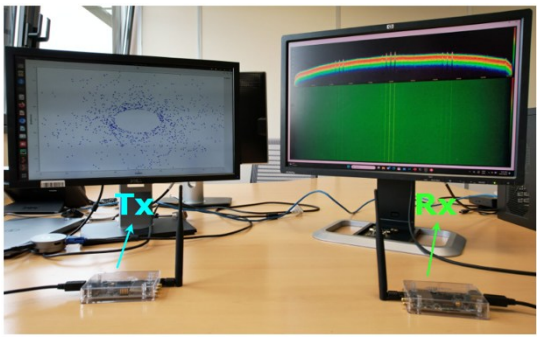


Figure 5.5: Experiment setup

The evaluation encompassed three challenging scenarios:

- **Scenario 1**: Focus on malicious device detection with 4 authorized and 8 malicious devices, testing the system's ability to identify threats.
- **Scenario 2**: Balanced environment with 6 authorized and 6 malicious devices, evaluating performance under equal class distributions.
- **Scenario 3**: Authorized-heavy configuration with 8 authorized and 4 malicious devices, assessing false alarm rates in legitimate-dominant networks.

5.1.5 Authentication Performance Results

Figure 5.6 presents the average detection rates across all ML and feature selection combinations, revealing several key insights:

Superior RnF-ANOVA Performance: Random Forest with ANOVA feature selection consistently achieved detection rates above 97% across all scenarios, demonstrating exceptional robustness in identifying both authorized and malicious devices.

Feature Selection Criticality: The choice of feature selection method significantly impacts performance, with ANOVA and MI generally outperforming PCA and RFE, particularly for ensemble methods.

Scenario Adaptability: While all approaches showed some performance degradation in more challenging scenarios, RnF-ANOVA maintained effectiveness with minimal performance loss.

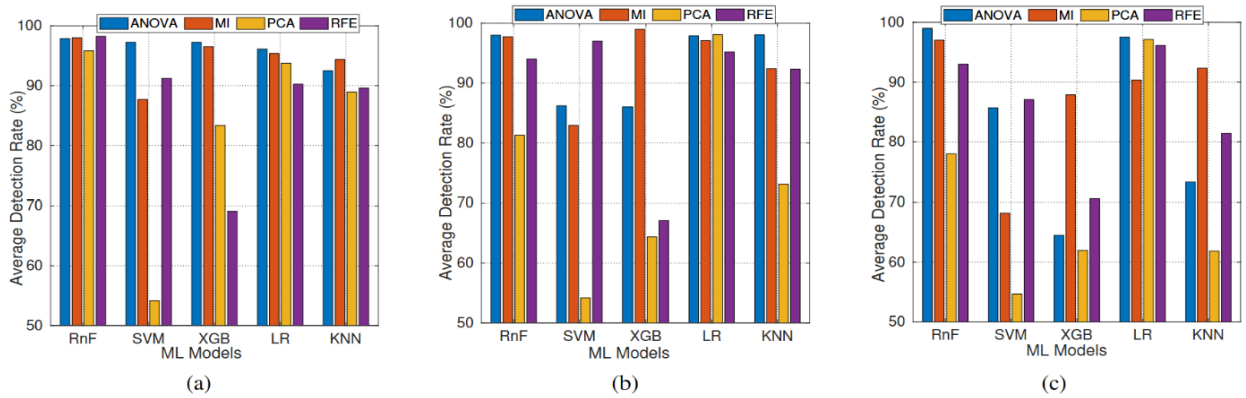


Figure 5.6: Average detection rate for different ML and FS models for (a) Scenario 1 (b) Scenario 2 (c) Scenario 3

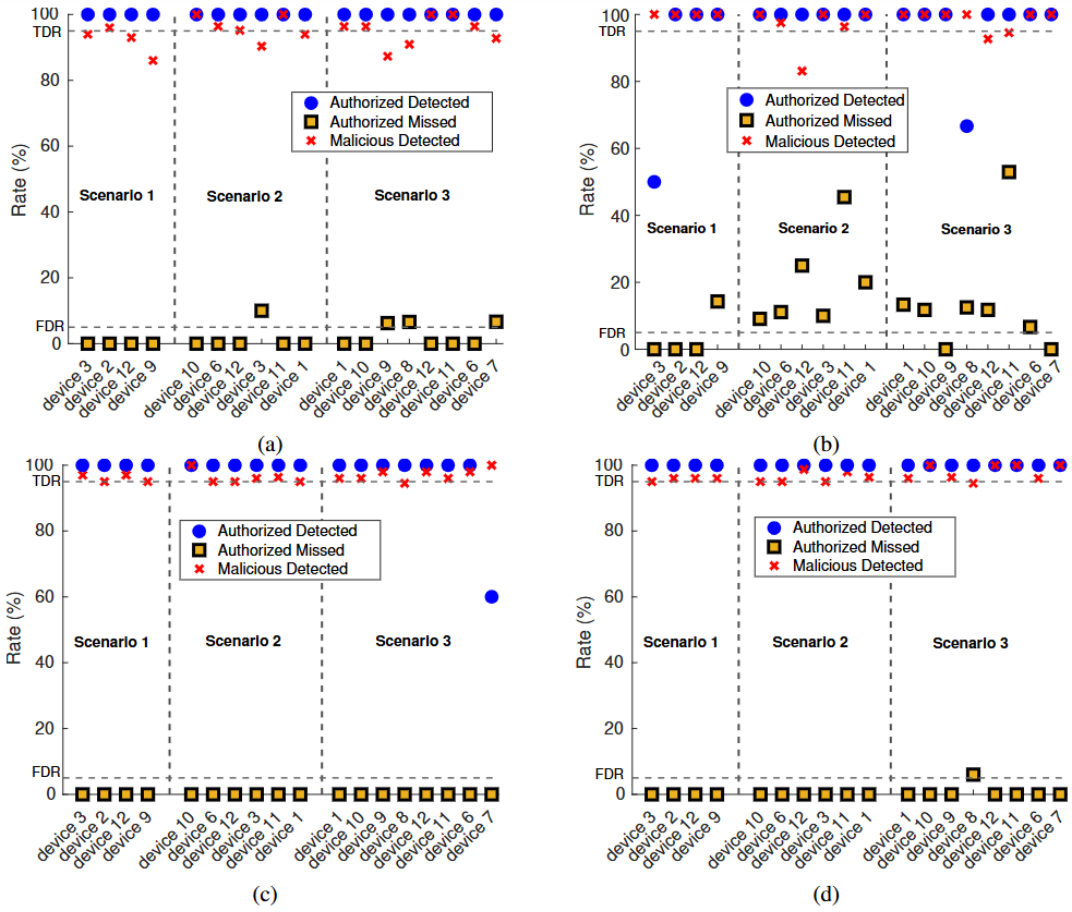


Figure 5.7: Authentication Rate for different combinations of FS and ML models (a) LR-ANOVA (b) LR-PCA (c) RnF-MI (d) RnF-ANOVA

Figure 5.7 provides detailed authentication rates for individual devices, highlighting the framework's consistent performance across different device types and attack patterns.

5.1.6 Robustness Under Varying Conditions

Critical to practical deployment is performance stability under different signal conditions. Figure 5.8 demonstrates our scheme's resilience across SNR variations from 1 to 15 dB:

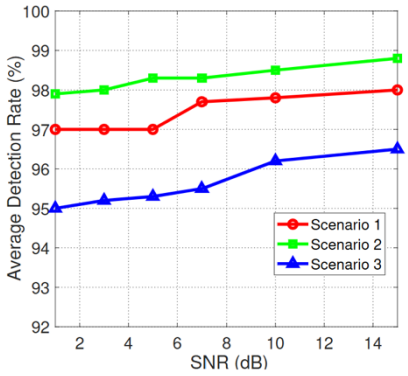


Figure 5.8: Average detection rate for different SNRs

The results show remarkable stability, with detection rates remaining above 95% even at SNR levels as low as 1 dB. This significantly outperforms previous approaches that showed substantial degradation below 10 dB SNR.

5.1.7 Computational Efficiency

For real-world deployment, we evaluated the computational requirements on a Jetson Nano Orin platform:

- **Mean inference time**: 3.714 ms per authentication decision.
- **Mean energy consumption**: 25.341 mJ per inference.

These metrics confirm the scheme's viability for real-time authentication in resource-constrained wireless networks. The dataset and code have been released to ensure reproducibility[27].

[27] https://github.com/PLA-AP/PLA

6 ML driven anomaly detection system to monitor 5G edge networks

The security and reliability of 5G edge networks are paramount, requiring advanced anomaly detection systems capable of identifying sophisticated threats. Our research within the MLSysOps project has focused on developing and evaluating robust, ML-driven approaches for this purpose, with a particular emphasis on detecting jamming attacks—a significant threat to wireless communication. Two key research efforts underpin our strategy: the comprehensive evaluation and system development, and the GANSec framework for enhancing model robustness through advanced data augmentation techniques.

Establishing Robust and Scalable Jamming Detection

A primary challenge in developing effective 5G anomaly detection is the need for systems that are both scalable and robust against complex, real-world threats. Traditional methods often fall short because they:

- Rely on costly specialized hardware like spectrum analyzers for RF fingerprinting.
- Focus on measurements from a single network entity (either RAN or UE), limiting a holistic view.
- Primarily address simplistic jamming profiles such as constant or reactive jammers, which produce overt disruptions.
- Are validated predominantly through simulations, which may not capture real-world operational dynamics and hardware interactions.

To address these critical gaps, we introduced an end-to-end solution for assessing jamming interference and supporting ML-based detection techniques.

For a visual overview of the methodology, Figure 6.1 illustrates the end-to-end pipeline including the testbed setup, data collection, preprocessing, and ML module.

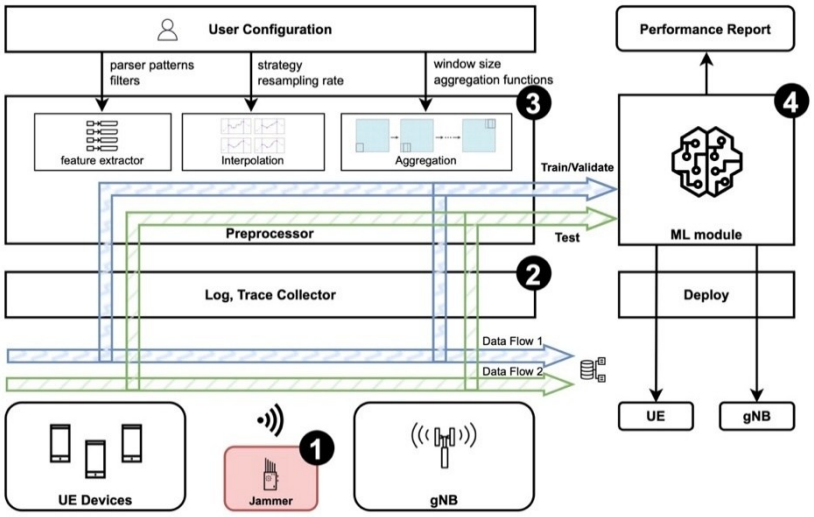


Figure 6.1: Architecture of the Jamming Detection Module

Core innovations include:

- **Leveraging Native Logs for Holistic, Cross-Layer Insights:** Our method uniquely collects and synchronizes native logs from both User Equipment (UE) and the Next-Generation Node B (gNB). This approach captures a comprehensive, multi-layer view of network behavior, including MAC-related metrics (signal quality, MCS index) and system-level insights (resource usage, scheduling decisions), without requiring costly external hardware. On the UE side, features like RSRP, RSRQ, and SINR are extracted from radio buffer logs, along with AT command records related to modem operations. From the gNB, traces include signal power, throughput, and packet transmission metrics (PUSCH, PUCCH).

- **Realistic Threat Modeling with a Power-Modulated Jammer:** A key component is a controllable, power-modulated jammer designed to simulate sophisticated interference. Unlike simpler jammers, this one periodically varies its amplitude using sinusoidal patterns to produce a less predictable noise profile, effectively challenging conventional detection methods. The jammer's behavior is defined by parameters such as sampling rate (r_samp), slot length (L slot), amplitude range, and pattern frequency (f pat).
  Figure 6.2 illustrates the tangible impact of this power-modulated jamming on key performance indicators like bandwidth, jitter, and packet loss.

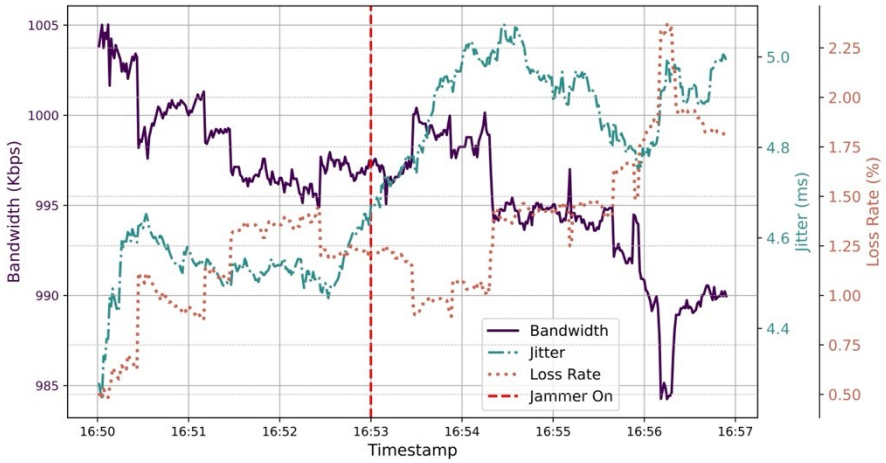


Figure 6.2: Comparison of interpolation techniques for handling missing/invalid data in log entries

- **Advanced Preprocessing for Rich Feature Extraction:** Raw logs are often noisy and irregularly timed. We employ a robust preprocessing pipeline to extract meaningful features and prepare data for ML models. This involves:
  - **Feature Extraction:** Using adaptable regular expression patterns to parse log messages and extract explicit measurements as well as contextual system-level signals.
  - **Interpolation:** Addressing uneven timestamps and missing data using techniques like forward filling for categorical features and polynomial interpolation (specifically, the Lagrange method) for quantitative metrics, ensuring a continuous time series.
  - **Aggregation:** Using a sliding window approach to compute statistical metrics (mean, standard deviation, median, amplitude, skewness, kurtosis) over features, capturing temporal patterns rather than just instantaneous values.
- **Rigorous ML Assessment with Emphasis on Robustness:** We evaluate several lightweight classifiers suitable for deployment in 5G environments (e.g., Support Vector Machines, K-Nearest Neighbors, Gradient Boosting, Random Forest). A cornerstone of its methodology is the use of a two-data-collection strategy. A primary dataset is used for training and initial validation. In contrast, an entirely separate "robustness test set"—collected under demonstrably different network conditions (e.g., varying UE-gNB distances, obstacles, temperature)—is used to measure model resilience to unseen operating conditions.
  The distinct feature distributions between the primary dataset and the robustness test set, highlighting the challenge for generalization, are visualized using KDE plots in Figure 6.3.

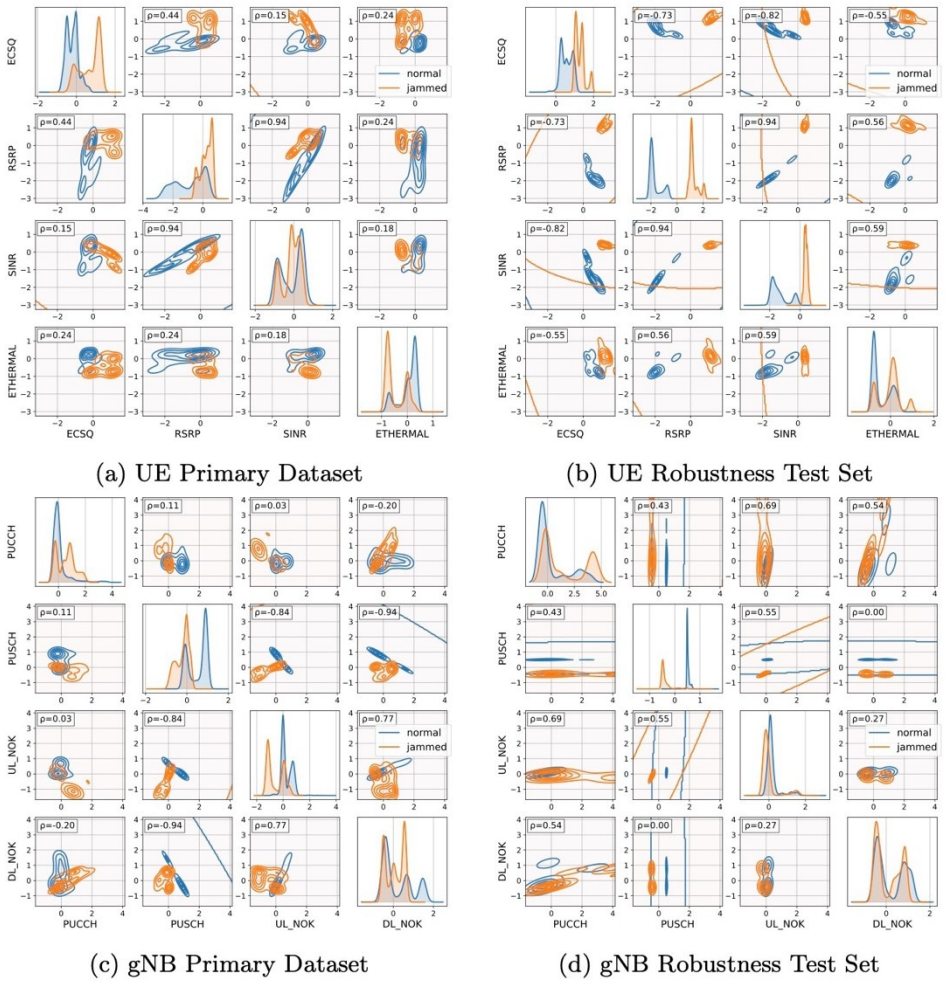


Figure 6.3: KDE analysis comparing the distributions of top-ranking features

Our experimental setup involved an open-source 5G core (Open5GS), a software-emulated RAN (srsRAN), bladeRF 2.0 micro SDRs for the gNB and jammer, and a OnePlus Nord 2T 5G as the UE.

The findings from the study were significant:

- Existing detection methods, benchmarked in our testbed, showed a performance drop from over 90% accuracy in controlled settings to below 70% under our more complex, varying conditions.
- In contrast, our log-based approach, leveraging its holistic data collection and robust preprocessing, maintained approximately 94% accuracy on the unseen robustness test set. The performance metrics of the evaluated classifiers are presented in Figure 6.4.

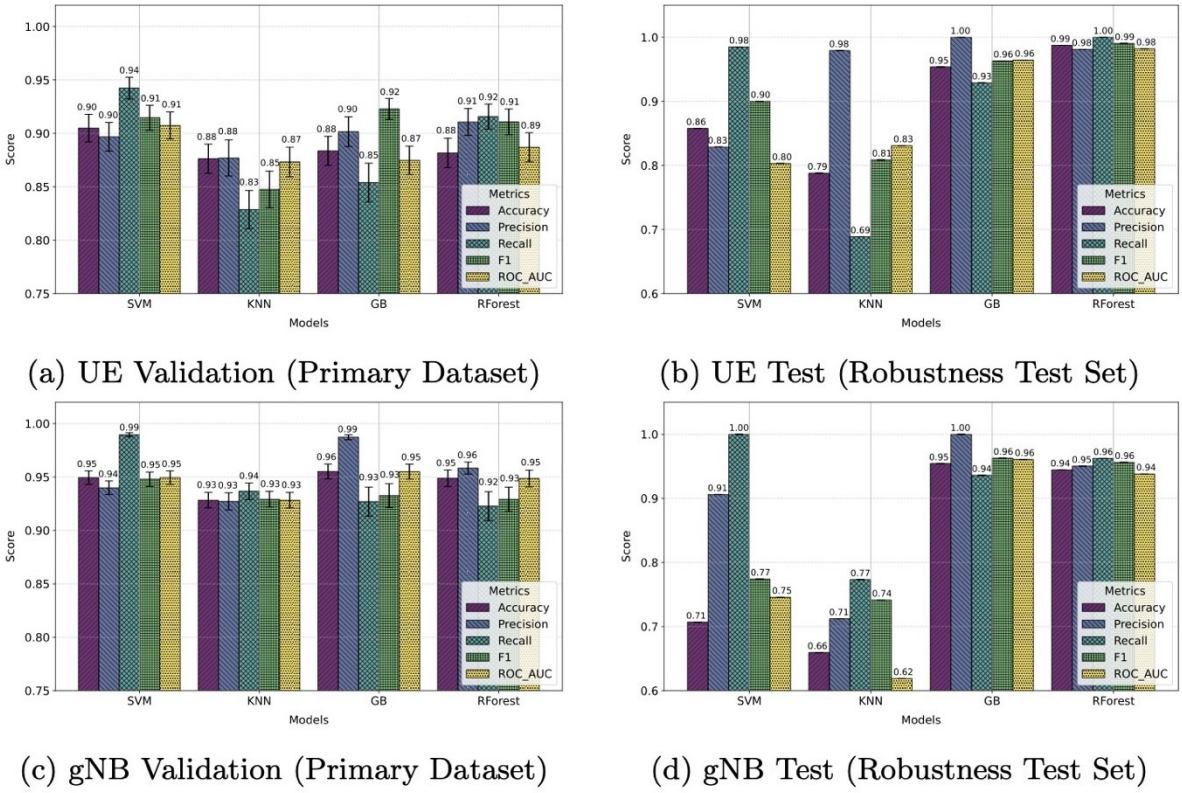


Figure 6.4: Accuracy, Precision, F1, Recall and ROC-AUC scores for each model

This demonstrates the efficacy in providing a scalable, cost-effective, and resilient solution for large-scale 5G jamming detection by harnessing native device logs and ensuring models are robustly evaluated.

GANSec: Enhancing Detection Robustness with Generative Data Augmentation

While our previous method provides a strong foundation, the performance and generalization of ML models can be further constrained by data limitations. Acquiring large-scale, diverse, and representative real-world wireless data is often prohibitively expensive and time-consuming. Wireless datasets frequently suffer from anomaly scarcity and class imbalance, hindering the training of reliable detection models. Traditional data augmentation techniques (e.g., noise injection, SMOTE) are often inadequate for the intricate nature of wireless time-series data, failing to capture temporal dependencies or underlying statistical properties. Moreover, many existing GAN-based augmentation approaches lack rigorous validation for robustness and generalization in diverse wireless conditions and offer limited control over the generation process.

An overview of the GANSec workflow, from data acquisition to downstream anomaly detection training, is provided in Figure 6.5.

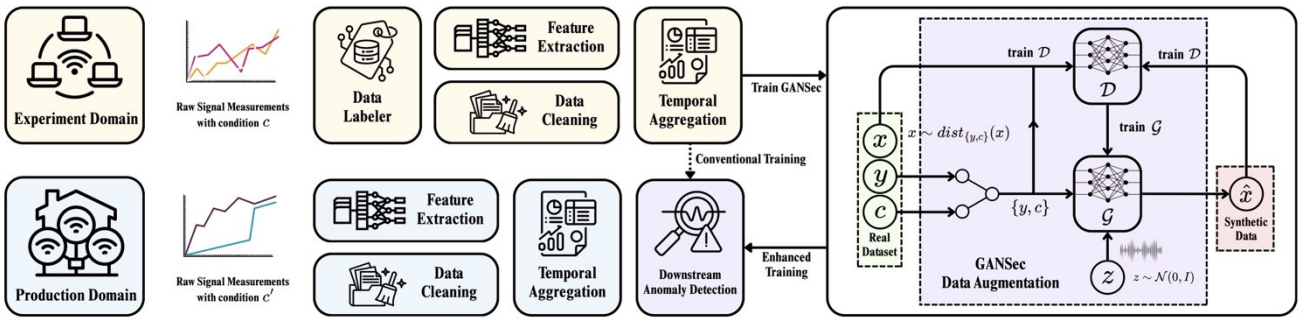
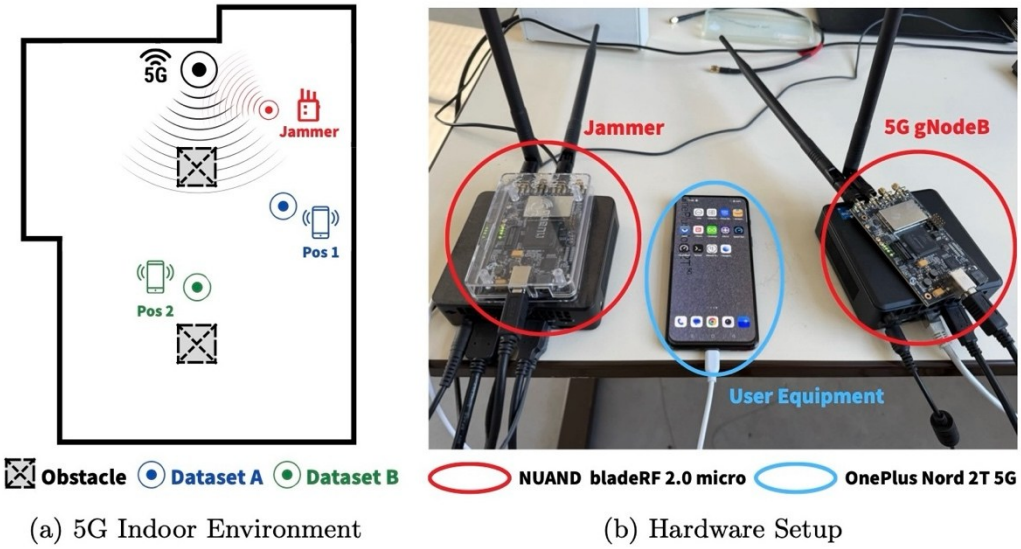


Figure 6.5: GAN-based Data Augmentation for Anomaly Detection in Wireless Security

Key aspects of GANSec include:

- **Tailored Conditional GANs for Wireless Data:** GANSec comprises a Generator (G) that learns to transform random noise vectors into synthetic data, and a Discriminator (D) that learns to distinguish these from real samples. The "conditional" aspect is crucial: GANSec takes the target label (e.g., 'normal' or 'jammed') and other contextual conditions (e.g., UE's distance to gNodeB, device model, modem temperature) as input during the generation process. This allows for the creation of contextually relevant synthetic data. We investigated different neural network backbones (MLP, LSTM, CNN) and two primary conditional training objectives for the Discriminator:
    - **Embedded Conditional (EC):** The Discriminator acts solely as a real/fake classifier for the given condition, emphasizing fidelity to the conditional data distribution.
    - **Classification Oriented (CO):** The Discriminator simultaneously predicts if the data is real/synthetic AND predicts its class label, encouraging the Generator to produce data that is not only realistic but also discriminative for the downstream task.
- **Specific Adaptations for Wireless Time-Series:** GANSec incorporates design considerations to handle the intricacies of wireless data effectively:
    - **Handling Temporal Dependencies:** Employs backbones like LSTMs and 2D CNNs suited for sequential data to capture crucial temporal correlations.
    - **Preserving Statistical Properties:** The framework is designed to learn and reproduce complex statistical properties of wireless signals, including those of log-scale metrics like RSRP (measured in dBm), which present unique challenges for neural networks.
- **Synthetic-Only Training Strategy for Robustness:** A cornerstone of the GANSec methodology is training the downstream anomaly detection model exclusively on the synthetic data generated by GANSec. This innovative strategy offers several advantages: generation of perfectly balanced datasets, potentially greater data diversity than the initial real samples, and enhanced privacy preservation by avoiding direct exposure of sensitive real-world measurements during detector training.
- **Cross-Scenario Evaluation for Rigorous Robustness Testing:** The experimental setup for GANSec involved a real-world 5G environment with a OnePlus Nord 2T 5G smartphone, two Nuand bladeRF 2.0 micro SDRs (as gNodeB and a power-modulated jammer), similar to the previous setup. The physical layout and hardware are depicted in Figure 6.6. Data was collected under two distinct conditions:
    - Scenario A: Used for training the GANSec models and baseline anomaly detectors. It included both normal operation and jamming anomalies, with a relatively balanced class distribution.
    - Scenario B: Collected under different UE positioning and with an additional obstacle, serving exclusively as the unseen test set to evaluate generalization capabilities. A 3-layer LSTM network was used as the downstream anomaly detector.

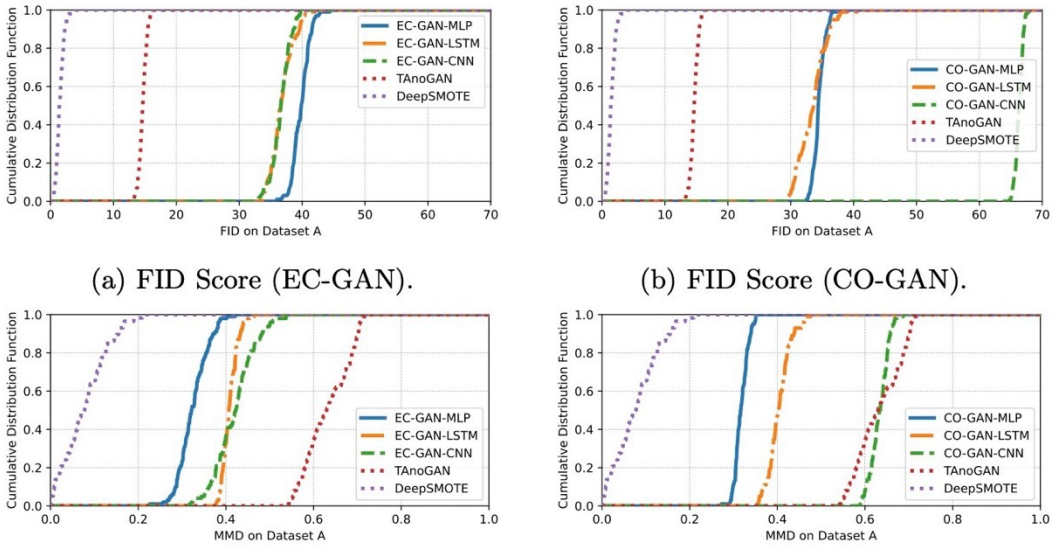


(a) 5G Indoor Environment (b) Hardware Setup

Figure 6.6: Experimental Setup for Collecting Dataset A and B

The GANSec study yielded compelling results:

- **Synthetic Data Quality:** While methods like DeepSMOTE produced samples very close to the original data (low FID/MMD scores), GANSec variants, particularly those with CNN backbones, generated a more diverse set of synthetic samples, hypothesized to be crucial for robustness. The distribution analysis of FID and MMD scores for various GANSec configurations and baselines is presented in Figure 6.7.



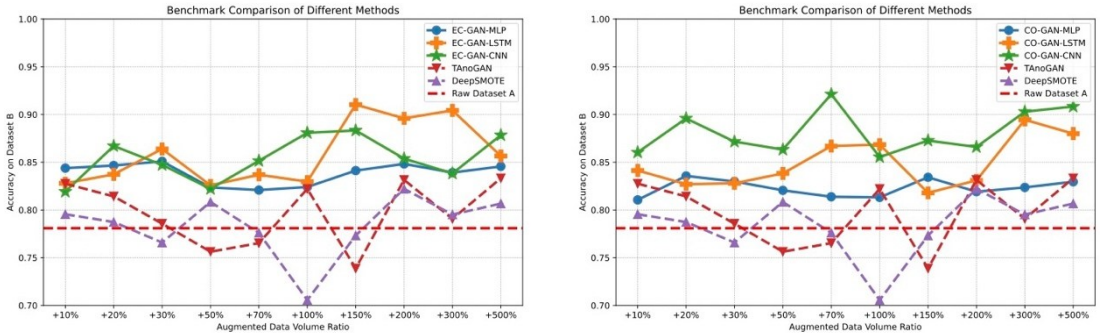
(a) FID Score (EC-GAN). (b) FID Score (CO-GAN).

(c) MMD Score (EC-GAN). (d) MMD Score (CO-GAN).

Figure 6.7: Distribution Analysis on Synthetic Dataset

- **Enhanced Generalization in Cross-Scenario Evaluation:** The baseline LSTM model trained on raw Scenario A data saw its accuracy drop from 92% (on Scenario A) to 78% when tested on the unseen Scenario B data, highlighting the significant generalization challenge. In contrast, models trained exclusively on data generated by GANSec variants (especially LSTM and CNN backbones with the CO-GAN objective) significantly outperformed all baselines on Scenario B. They achieved up to 92.13% accuracy on the unseen Scenario B dataset, showcasing substantially enhanced robustness and

generalization. The accuracy results on the unseen Dataset B across different augmentation ratios are detailed in Figure 6.8.



(a) EC-GAN (b) CO-GAN

Figure 6.8: Impact of the augmentation ratio (relative size of synthetic data used for training) on the accuracy achieved on the unseen Dataset B

This demonstrates that tailored conditional GANs, used with a synthetic-only training approach, can effectively build more robust wireless anomaly detection systems capable of handling unseen network conditions.

7 Minimizing the latency of traffic routing for 5G users using RL

Unlike previous generations of mobile networks, 5G will enable ultra-reliable user applications and support millions of sensors and devices within the Internet of Things (IoT). Together, these will bring new use cases to enable smart cities, autonomous vehicles, smart logistics, smart energy, and so on. Within this landscape, the quest for minimizing latency emerges as a paramount objective, vital for ensuring seamless interactions between users and data centers.

The primary focus of this study lies in the orchestration of a comprehensive strategy to minimize latency in 5G networks, specifically targeting the interaction between 5G base stations (also known as gNodeB) and Data Centers. An efficient 5G network with low latency is particularly significant for Industry 4.0, where real-time processing and immediate feedback are essential for automating and optimizing industrial processes. Reduced latency enhances the ability of machines to communicate seamlessly, improves the responsiveness of automated systems, and supports the development of smart factories where operations are dynamically adjusted based on real-time data.

In the face of escalating demand for swift and responsive communication, the methodologies, techniques, and considerations outlined in this report contribute to the crucial task of achieving optimal network performance. Additionally, energy efficiency and the utilization of green energy, performance efficiency, trusted tier-less storage, cross-layer orchestration including resource-constrained devices, resilience to imperfections of physical networks, trust, and security, are key elements in ensuring sustainable and robust network operations.

The purpose of this section is to provide a comprehensive overview of the strategy employed for the optimization of a 5G network, specifically focusing on minimizing latency between the gNodeB and target Data Centers. As the demand for high-speed, low-latency communication continues to surge, the optimization of 5G networks becomes paramount. This report delves into the methodologies, techniques, and key considerations adopted to achieve optimal network performance.

The User Plane Function (UPF) is the fundamental component of the 5G core infrastructure that enables such low-latency edge computing. A further essential component in a 5G architecture is the N3 interface, which connects the gNodeB Radio Access Network (RAN) to the UPF. The interface conveys user data from the RAN to the UPF for processing – the distance the UPF sits from the edge of the network depends on the latency and performance requirements of the user application.

This study extends to the intricate details of deploying the UPF within the target Data Center closest to the gNodeB. It explores the utilization of an edge-centric approach for distributing the user plane across the operator's country, demonstrating a commitment to reducing latency and enhancing overall network efficiency.

Leveraging Reinforcement Learning (RL) techniques at each layer, we aim to create a self-optimizing network capable of adapting to users' diverse needs and preferences in real-time. At the heart of our approach lies the integration of RL-inspired "best Path Finding" techniques, designed to minimize latency and enhance network efficiency.

A crucial component of the methodology is the implementation of RL-inspired “Best Path Finding” techniques, a strategic approach to minimizing latency. This involves selecting optimal paths based on KPIs, including latency, bandwidth usage percentages, traffic load, and CPU usage. The integration of these techniques aims to create a self-optimizing network that continually adapts to meet the demands of users and applications.

We recognize the pivotal role of real-time data processing in translating user preferences into actionable insights. By harnessing key performance indicators (KPIs) such as latency, bandwidth usage percentages, and traffic load, we strive to tailor network operations to suit individual user needs. Central to our approach is the recognition of users as pivotal entities shaping the network's landscape. By deploying the UPF within strategically selected Data Centers, we embrace an edge-centric paradigm aimed at enhancing user experience and reducing latency. By addressing the real-time data processing challenges inherent in 5G networks through

a user-centric lens, we contribute valuable insights to the ongoing discourse on the future of 5G connectivity and well-being.

The adopted approach revolves around the application of RL techniques at each layer of the network architecture. This involves the careful consideration of various KPIs collected from multiple Data Centers. The essence of the adopted approach lies in its ability to adapt to the dynamic and evolving nature of 5G networks, ensuring responsiveness to changing network conditions.

In the domain of network performance optimization, Artificial Intelligence (AI) and the Industrial Internet of Things (IIoT) are crucial, leveraging their adaptive capabilities to thrive in dynamic environments. Additionally, sustainability emerges as a paramount theme in these innovative solutions, underscoring the need for energy-efficient and environmentally responsible technologies.

RL is the cornerstone of our approach to optimizing the 5G network, particularly in the context of minimizing latency between the gNodeB and target Data Centers. RL, as a methodology, empowers the model to learn and adapt through interactions with its environment, making it well-suited for the dynamic and evolving nature of 5G networks.

RL Agents

The core of our optimization strategy is the RL agents, entities that learn to take actions in an environment to maximize a reward signal, as shown in Figure 2.3. The learning process involves the agent exploring the environment, observing the outcomes of its actions, and adapting its decision policy to maximize the reward over time. Learning algorithms based on predefined rules can guide this process. We defined agents and environments for each level of our architecture.

RL Environment

The RL environment is critical to the application of RL techniques within our network architecture. Our framework for environment definition is *Gymnasium*, a Python library derived from *OpenAI Gym*. This framework provides a standard API for communication between learning algorithms and environments, facilitating the development and comparison of RL algorithms.

Action, Observation, and Reward

For our latency optimization task, we define the action as selecting the edge Milan, Rome, and Cosenza, the observation as a triplet of dataset rows representing the feature status at the current timestep for each edge, and the reward as a function based on key performance indicators (KPIs). This reward function ensures that the selected edge minimizes latency while considering other crucial factors such as CPU usage and other metrics.

Policy for Target Data Center Selection

To achieve the primary goal of minimizing latency, a policy for target data center selection is defined within the RL environment. The episode ends when the current edge and the target edge are the same, aligning with the overarching objective of latency reduction.

The incorporation of DQN, PPO, and A2C as RL agents, coupled with the well-defined RL environment, forms the foundation of our methodology for optimizing the 5G network. This approach allows the network to adapt dynamically to diverse network conditions, providing a responsive and efficient communication infrastructure.

Methodology

We delve into the intricate details of the methodology employed for the 5G network optimization project. The overarching goal is to provide a clear understanding of how RL techniques are applied at various layers of the network architecture, along with the definition and explanation of the dataset that forms the basis of training and evaluation.

5G Network architecture with the elements involved in are shown in Figure 7.1. Milan, Rome, and Cosenza, the geographical locations where network optimization strategies are applied, represent the diversity of network conditions across the operator's country.

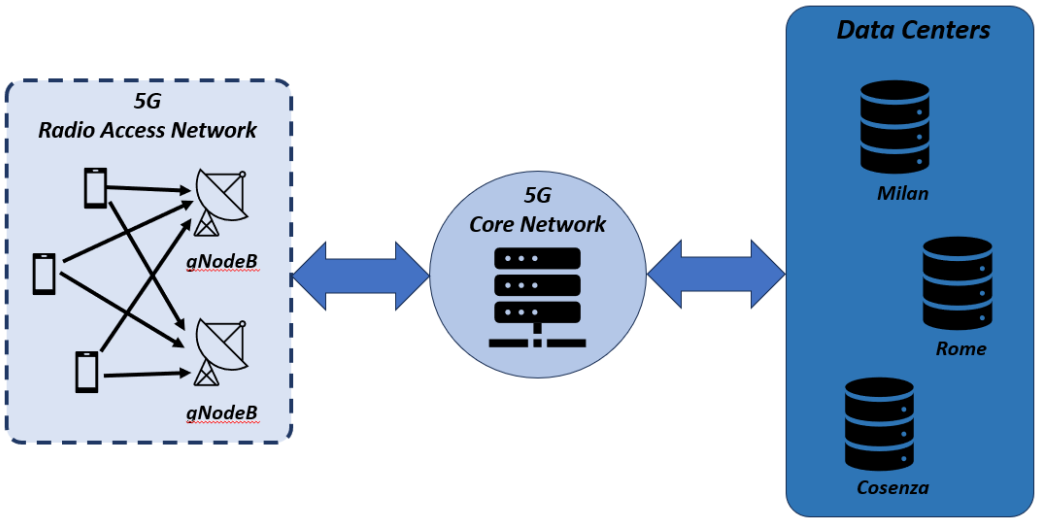


Figure 7.1: 5G Network elements involved. Each device is connected to the Core Network by means of a gNodeB. The core network selects the data center according to predefined KPIs

7.1.1 Dataset Overview

The foundation of our optimization strategy lies in the careful curation of a comprehensive dataset encompassing KPIs from multiple Data Centers. The selection of KPIs is critical in capturing the diverse aspects of network performance, ensuring a holistic representation of the network's state. The three chosen Data Centers, Milan, Rome, and Cosenza, serve as the geographical points of interest, each bringing unique characteristics and challenges to the optimization process.

In orchestrating our optimization strategy, the foundation lies in the deliberate selection of KPIs, each meticulously chosen to provide a nuanced understanding of the 5G network's operational dynamics:

- **start time** [timestamp]: This attribute represents the timestamp indicating the start time of a particular event or process within the network. It serves as a reference point for tracking the initiation of network operations or data transactions.
- **end time** [timestamp]: This attribute denotes the timestamp indicating the completion time of a specific event or process within the network. It marks the conclusion of network activities or data transactions and is used to measure the duration of these operations.
- **CPU usage** [%]: *CPU usage* refers to the percentage of the Central Processing Unit's capacity that is currently being utilized. It provides insights into the computational workload imposed on the system, which is crucial for optimizing resource allocation and ensuring efficient processing of network tasks.

- **memory usage** [%]: *Memory usage* represents the percentage of available Random Access Memory (RAM) that is currently in use. Monitoring memory usage is essential for detecting potential memory bottlenecks and optimizing memory allocation to prevent performance degradation or system failures.
- **disk usage** [%]: *Disk usage* denotes the percentage of storage capacity on the disk that is currently occupied. It offers insights into the utilization of storage resources, enabling administrators to manage disk space effectively and prevent storage-related issues such as disk saturation or data loss.
- **net in absolute** [kbps]: *Net in absolute* refers to the absolute amount of incoming network traffic measured in kilobits per second (kbps). It quantifies the rate at which data is received by the network interface, aiding in the analysis of network throughput and performance optimization.
- **net out absolute** [kbps]: *Net out absolute* represents the absolute amount of outgoing network traffic measured in kilobits per second (kbps). It indicates the rate at which data is transmitted from the network interface, facilitating the assessment of outbound data flow and network efficiency.
- **net in** [%]: *Net in* denotes the percentage of available network bandwidth that is currently utilized for incoming traffic. It helps monitor network congestion and identify potential bandwidth limitations that may impact the transmission of data packets.
- **net out** [%]: *Net out* signifies the percentage of available network bandwidth currently utilized for outgoing traffic. Similar to net in, monitoring the net out percentage aids in assessing network utilization and ensuring optimal allocation of bandwidth resources.
- **latency min** [ms]: *Latency min* represents the minimum latency observed during a specific period, measured in milliseconds (ms). It reflects the shortest time taken for data packets to travel between network endpoints, serving as a critical metric for evaluating network responsiveness and performance.
- **latency avg** [ms]: *Latency avg* denotes the average latency experienced over a given timeframe, measured in milliseconds (ms). It provides a comprehensive view of the network's overall responsiveness, considering fluctuations in latency and enabling administrators to identify trends or patterns in network performance.
- **latency max** [ms]: *Latency max* indicates the maximum latency observed within a specified interval, measured in milliseconds (ms). It represents the longest time taken for data packets to traverse the network, highlighting potential bottlenecks or areas of congestion that may impact overall network performance.
- **latency mdev** [ms]: *Latency mean deviation* quantifies the variability or dispersion of latency measurements around the average value, expressed in milliseconds (ms). It provides insights into the stability and consistency of network latency, aiding in the identification of outliers or irregularities that may require attention.
- **lost packets** [%]: *Lost packets* refer to the percentage of data packets that fail to reach their intended destination or are discarded during transmission. Packet loss can occur due to network congestion, hardware failures, or other factors, and monitoring this metric is essential for assessing network reliability and quality of service.

The centralized core network is located in Milan's central data center. Here, the network functions manage the control plane. The user plane is managed by the UPF, a virtualized network function deployed in Milan, Rome, and Cosenza data centers.

On each data center, we introduced a bandwidth cap to simulate a fixed limitation. Then we triggered data traffic on the N3 interface, which is the user plane interface between gNodeB and UPF. The data traffic is simulated through a script describing profiles. These profiles delineate patterns in data transmission, such as bandwidth consumption, peak hours, and resource utilization. The script considers three different time slots: Night, BusyHour, and Day, each corresponding to distinct traffic patterns. These classifications dictate the sleep interval, maximum throughput, maximum duration, and maximum CPU load for each simulation cycle. Each cycle randomizes required input values up to the maximum value foreseen for that specific time of the day. By

integrating diverse parameters and time-of-day considerations, the script can simulate various traffic profiles. Night Traffic is characterized by lower throughput, longer session durations, and moderate CPU load; busy Hour Traffic reflects peak usage periods with higher throughput, shorter session duration, and intensified CPU load; daytime Traffic exhibits moderate throughput, medium session duration, and CPU load, representing typical usage patterns during the day.

Furthermore, we introduced a real packet loss in each data center, which was different for each one. The telemetry setup, enriched with dynamic profiling based on peak-aware traffic scenarios, establishes a robust framework for gathering measurements of selected parameters.

We therefore devised a dataset from the triggered data traffic, which has been employed in our study. Before utilizing this data, several preprocessing steps are undertaken to ensure its quality and consistency. We exploited **Python** with the help of the ***Pandas*** library to pack data in a single ***DataFrame***. As different metrics may have varying scales and units, normalization is applied to standardize the features and bring them within a common range. Here, we employ the ***MinMaxScaler scaling algorithm from the scikit-learn*** Python library to transform the data to a [0, 1] interval.

The dataset also contains client IP addresses, which are encoded into numerical identifiers to facilitate processing and analysis. This step ensures uniformity in representing client identities within the dataset.

7.1.2 Features Weighting

The primary objective of this study is to construct an RL model tailored to the task of optimal edge selection. To achieve this goal, it is imperative to establish a precise definition of the optimal data center within the context of our study. Specifically, the model must acquire the capability to discern and select the data center that minimizes latency between the gNodeB and the chosen edge. Consequently, the RL Environment must be meticulously engineered to prioritize the minimization of latency as its principal objective. Nevertheless, it is crucial to acknowledge and adhere to additional KPIs beyond latency optimization. For instance, the scenario may arise where a data center with minimal latency is selected but concurrently exhibits maximal CPU utilization. Thus, it becomes imperative to delineate a coherent policy governing the selection of the target data center within the environment definition itself. This policy must effectively balance the optimization of latency with the consideration of other pertinent KPIs to ensure the overall effectiveness and efficiency of the RL model in its edge selection task.

The policy governing the selection of the target data center within the environment is established through the association of weights with each feature, thereby assigning importance to individual features contributing to the reward calculation process. These weights serve as parameters that dictate the relative significance of each feature in influencing the overall reward obtained by the RL agent. By assigning appropriate weights to different features, the policy effectively prioritizes certain performance metrics over others, guiding the RL agent's decision-making process toward optimizing resource allocation in the data center environment. This weighted approach ensures that the RL agent takes into account the multifaceted nature of data center operations, considering various factors such as latency, CPU utilization, memory usage, and network traffic, among others, in its pursuit of efficient resource allocation strategies. Consequently, the policy encapsulates the domain knowledge and objectives of the data center management, enabling the RL agent to make informed and effective decisions that align with the overarching goals of the system. In Table 7.1 are represented the weights for each feature.

Table 7.1: Features weights

Feature	Weight	Unit of Measurement
Start time	0	Timestamp
End time	0	Timestamp
CPU usage percent	0.6	Percent
Memory usage percent	0.5	Percent

Disk usage percent	0.5	Percent
Net in absolute	0	Kbps
Net out absolute	0	Kbps
Net in percent	0.7	Percent
Net out percent	0.7	Percent
Latency min	0	ms
Latency avg	1	ms
Latency max	0	ms
Latency mdev	0.2	ms
Lost percent	0.9	Percent

7.1.3 RL Algorithm Selection

The framework used for environment customization is ***Gymnasium***, an open-source Python library for developing and comparing reinforcement learning algorithms by providing a standard API to communicate between learning algorithms and environments, as well as a standard set of environments compliant with that API[28]. This is a fork of the *OpenAI* ***Gym*** library.

First, we defined action, observation, and reward as follows:

- *Action*: the edge selection (Milano, Roma, and Cosenza). From now on, network edge and data center terms are considered interchangeable.
- *Observation*: a triplet of dataset rows representing the feature status at the current timestep for each edge. The three data centers are identified by the indexes *(0,1,2)*.
- *Reward*: defined as $\sum_{i=1}^{n} \frac{W(i)}{D(i)}$, where *n* is the number of features, *W(i)* is the weight associated with feature i, and D(i) is the best value respecting KPI for each feature. The lower these values are, the higher the reward.

All environment methods are defined inside a *gym.Env* class; the main methods have been structured as follows:

- ***Step***: receives action as input and returns (*observation, reward, terminated, truncated, done).*
- ***Reset***: if called, defines the current edge by means of a random action.
- ***Calculate_reward***: receives observation and index of the edge target as input and returns the reward according to the formula above.

The episode ends when the current edge and the edge target are the same. The edge target is selected according to the policy described in the pseudocode shown in Algorithm 7.1. Hence, briefly speaking, the agent must learn how to predict the target data center.

[28] Towers, M., Kwiatkowski, A., Terry, J., Balis, J. U., De Cola, G., Deleu, T., ... & Younis, O. G. (2024). Gymnasium: A standard interface for reinforcement learning environments. *arXiv preprint arXiv:2407.17032*.

Algorithm 1 Best Target Data Center Selection

Require: the dataset, composed of a set of matrices wherein rows represent the data center and columns represent features
Ensure: an index identifying the best data center
observation: matrix of shape $(n\_datacenter, n\_features)$
$features\_weights$: a list of weights assigned to each feature
Initialize min_tot, an array of zeros of shape $(n\_features,)$
Initialize $data\_center\_best$, an array of zeros of shape $(n\_features,)$
Initialize $contrib\_tot$, an array of zeros of shape $(n\_features,)$
$weighted\_data \leftarrow observation * feature\_weights$
for $i = 1$ **to** n_features **do**
  Select the minimum value
  Select the index of the best value
  Append minimum value to min_tot
  Append the index to data_center_best
end for
for $idx, elem = 0$ **to** data_center_best **do**
  if elem is 0 **then**
    Append contrib_tot[0] to weighted_data[elem][idx]
  else if elem is 1 **then**
    Append contrib_tot[1] to weighted_data[elem][idx]
  else
    Append contrib_tot[2] to weighted_data[elem][idx]
  end if
end for
Define target_data_center_idx as the index corresponding to the max value of contrib_tot.

Algorithm 7.1: Best target data center selection based on dataset input and index output.

For the agent model training, we tested three different architectures: DQN, PPO, and A2C, using the *stable baseline3* library.

Models Parameters

For all models, we decided to define three main common parameters, as described below.

- *Learning rate*: indicates the speed at which the neural network adapts to the data during training. A higher learning rate implies more aggressive updates to the network weights.
- *Batch size*: indicates the number of experiences randomly sampled from the replay memory during each training step.
- *Discount factor* (γ): represents the discount rate for future rewards. It helps give more weight to immediate rewards compared to future ones. It is used in different ways depending on the model architecture.

The loss function is based on the difference between the current estimate of the Q-function and the target estimate and has the form:

$$\mathcal{L} = \mathrm{E}\left[\left(Q(s,a;\theta) - \left(r + \gamma max(Q(s',a';\theta_{target})\right)\right)^2\right]$$

The loss is minimized during training to learn the weights of the network. **DQN** selected architecture has two hidden layers of 64 neurons; other main parameters are described below, while the DQN loss evaluation scheme is depicted in Figure 7.2.

- *Replay buffer size*: specifies the maximum size of the replay memory.
- *Soft update coefficient* (τ): maintains a stable target version of the Q-Network.
- *Training frequency*: indicates how often the agent performs a training step after collecting new data.
- *Target network update frequency*: determines how often the target network weights are updated with the weights of the main network.
- *Exploration fraction*: defines the fraction of total training steps during which an exploration strategy (such as ε-greedy) is used.
- *Initial-Final exploration* (ε): specifies the initial-final value of ε in the ε-greedy strategy. ε controls the probability that the agent explores a new action instead of following the estimated optimal action.
- *Gradient steps*: specifies how many gradient steps are performed during each training step.
- *Target update interval (α)*: update the target network every target update interval environment steps.

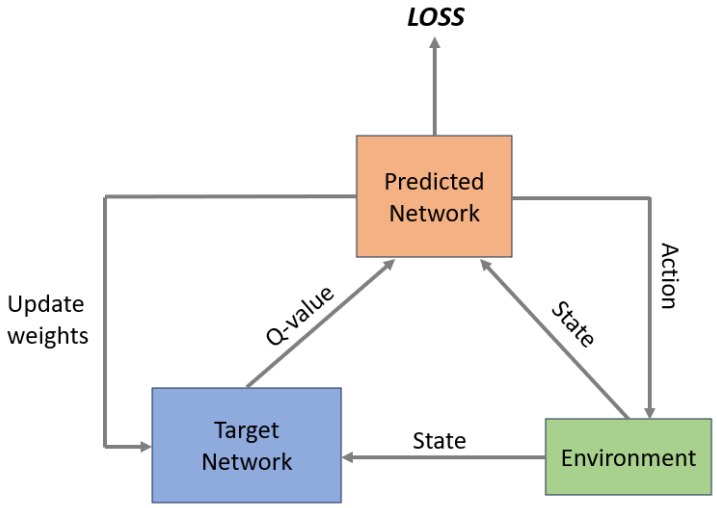


Figure 7.2: Diagram of the weight updates and loss evaluation in DQN

PPO loss function includes terms based on the probability ratio between the current policy and the previous policy, ensuring that the policy update is not too aggressive; this helps maintain stability during training. The function has the form:

$$\mathcal{L} = \mathrm{E}\left[\min\left(r(\theta)\cdot\hat{A}, \mathrm{clip}(r(\theta), 1-\epsilon, 1+\epsilon)\cdot\hat{A}\right)\right],$$

where $r(\theta)$ is the probability ratio, $\hat{A}$ an approximation of the advantage evaluated for each state-action, and ε is a clip hyperparameter. The selected architecture has two hidden layers of 64 neurons; other main parameters are described below. The PPO loss evaluation scheme is depicted in Figure 7.3.

- *Truncation factor* (λ): used in the calculation of the advantage function. It determines the trade-off between bias and variance in estimating advantages.
- *Advantage clipping parameter:* determines the maximum magnitude of the policy update during optimization. It limits the probability ratio between the new and old policies.
- *Clipping range*: another clipping parameter that can be used to control the maximum magnitude of the policy update.
- *Clipping range for the value function*: clipping parameter that can be used to control the maximum magnitude of the value function.
- *Entropy coefficient*: controls how much the policy entropy is involved as a regularization term. It contributes to maintaining diversity in the actions taken by the agent.
- *Value function coefficient*: this parameter determines how much the value function contributes to the total loss function. It controls the relative importance of the value function versus the policy.
- *Maximum gradient norm*: limits the gradient norm during the update to prevent gradient explosion problems.
- *Use SDE*: defines whether to use generalized State Dependent Exploration (gSDE) instead of action noise exploration.
- *Normalize advantage*: if activated, this normalization helps stabilize training and can be particularly useful when dealing with problems where reward scales may vary significantly.

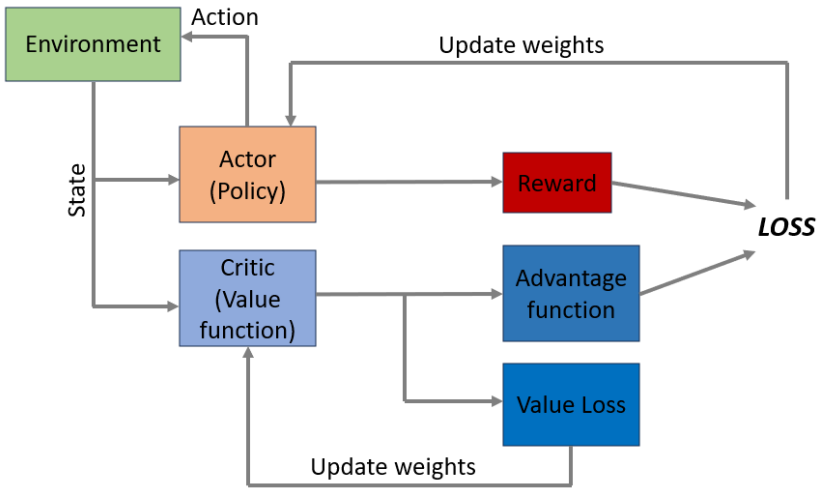


Figure 7.3: Diagram of the weight updates process and loss evaluation in PPO

The loss function for **A2C** includes terms for both the policy (actor) and the value (critic). The general formula for the A2C loss function is as follows:

$$Loss = Policy\ Loss + Value\ Loss,$$

with:

$$Policy\ Loss = -log(\pi_\theta(a|s)) \cdot \hat{A}$$
$$Value\ Loss = \left(\hat{V}(s) - V_\theta(s)\right)^2,$$

Wherein $\hat{V}(s)$ is the estimated advantage for state s calculated by using cumulative rewards and $V_\theta(s)$ the model's estimated value. A2C selected architecture has two hidden layers of 64 neurons; other main parameters are described below. The A2C loss evaluation scheme is depicted in Figure 7.4.

- *Truncation factor* (λ): used in the calculation of the advantage function. It determines the trade-off between bias and variance in estimating advantages.
- *Entropy coefficient*: is a measure of the uncertainty in the policy's output. The entropy coefficient balances the trade-off between exploration and exploitation by encouraging or discouraging the agent from taking uncertain actions.
- *Value function coefficient*: controls the weight of the value function (critic) in the overall loss function. It determines the importance of the value function compared to the policy in the learning process.
- *Maximum gradient norm*: limits the gradient norm during optimization to prevent the exploding gradient problem.
- *RMSProp* ε: a small value added to the denominator of the RMSProp update to avoid division by zero.
- *Use RMSProp optimizer*: a boolean parameter indicating whether to use the RMSProp optimizer. If set to True, RMSProp is used; otherwise, the Adam optimizer might be used.
- *Maximum gradient norm*: specifies the maximum norm for the gradients during optimization to prevent large updates.

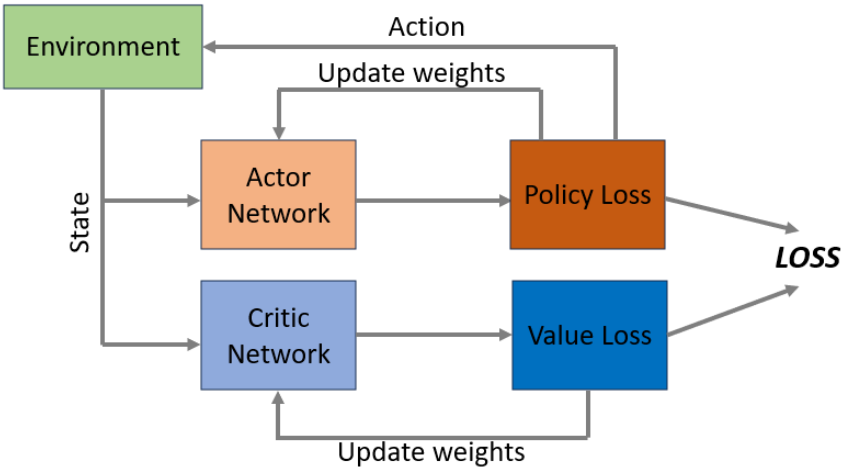


Figure 7.4: Diagram of the weight updates and loss evaluation in A2C

Models Fine-Tuning

As a preliminary test, we made two sets of experiments on two different machines:

1. NVIDIA Jetson AGX Xavier series.
2. Intel(R) Xeon(R) Silver 4210 CPU @ 2.20GHz, 10 vCPU, 32GB RAM.

We tried all the possible combinations among three different values for learning rate, batch size, and discount factor (i.e., gamma) parameters. The following dimensions have been selected:

- Learning rate: {0.001, 0.0001, 0.0002, 0.0003, 0.0004, 0.0005, 0.00005};
- Gamma: {0.45, 0.75, 0.99};
- Batch size: {64, 128, 256}.

These values have been selected as reasonable for the exam system. This step helped us select which model can perform better according to the task. Also, for a total number of timesteps of 3.000.000, we set the effective learning to start after the first 50.000 steps for the warm-up phase.

To analyze results, we have considered three different aspects:

- The maximum reward value achieved, meaning the maximum value of the run: a model shows good performance if that value lies in the convergence phase.
- Reward convergence and stability: a model shows good performance if the reward value becomes reasonably stable after a reasonable number of steps.
- Loss function behaviour: a model shows good performance if its loss functions are minimized/maximized according to the case.

7.1.4 Algorithm Selection Results

This section delves into the detailed results obtained from a series of experiments designed to assess and compare the performance of three prominent reinforcement learning algorithms: Deep Q-Network (DQN), Proximal Policy Optimization (PPO), and Advantage Actor-Critic (A2C). The objective of these experiments was to identify the most effective algorithm and its optimal configuration for the given task of optimizing latency in 5G networks for edges in Milan, Rome, and Cosenza.

DQN, PPO, and A2C represent three pillars of RL, each leveraging distinct strategies. The experiments aimed to provide insights into their performance nuances, particularly focusing on learning rate, gamma, batch size, maximum reward, and convergence time steps.

DQN Results

DQN results are reported in the following table.

Table 7.2: DQN hyper parameters configurations and results of training

Learning rate	Gamma	Batch size	Max reward	Convergence time steps	Notes
0.001	0.45	64,128	320	1M	High stability
0.001	**0.45**	**256**	**338**	**800K**	**High stability**
0.001	0.75	64	296	1.5M	High stability
0.001	0.75	128,256	300	1M	High stability
0.001	0.99	64,128,256	215	1M	High stability
0.0001	0.45	64,128,256	308	> 3M	Stable but slow convergence
0.0001	0.75	64,128,256	288	> 3M	Stable but slow convergence
0.0001	0.99	64,128,256	181	> 3M	Stable but slow convergence
0.0002	0.45	64,128,256	320	1.5M	High stability
0.0002	0.75	64,128,256	330	2M	High stability
0.0002	0.99	64	200	> 3M	Instability, no convergence
0.0002	0.99	128,256	208	2M	Stability
0.0003	0.45	64,128,256	330	1.5M	High stability
0.0003	0.75	64,128,256	315	1.5M	High stability

0.0003	0.99	64	193	>3M	Instability, no convergence
0.0003	0.99	128,256	228	1.7M	Stability
0.0004	0.45	64,128,256	323	1M	High stability
0.0004	0.75	64,128,256	305	1.5M	High stability
0.0004	0.99	64	230	1.8M	Stability
0.0004	0.99	128,256	205	1.5M	Stability
0.0005	0.45	64,128,256	325	1M	High stability
0.0005	0.75	64,128,256	317	1.5M	High stability
0.0005	0.99	64,128,256	245	1M	High stability
0.00005	0.45,0.75,0.99	64,128,256	154	NONE	Instability, no convergence

DQN, a foundational algorithm in RL, demonstrated varying performances across different configurations. These configurations include different learning rates, gamma values, and batch sizes, allowing for a comprehensive analysis of the model's stability and convergence characteristics. Observations from the experiments indicate:

- Configurations with lower learning rates generally exhibit high stability and efficient convergence.
- Moderate gamma values contribute to stable training, although excessively high values may lead to instability.
- Batch size variations show mixed results, with smaller batch sizes tending to offer more stable training dynamics.
- Extreme parameter values, such as a learning rate of 0.00005 combined with multiple gamma values, lead to unstable training behavior and hinder convergence.
- The loss analysis, derived from the three best-performing models, reveals that slower and more unstable decreases in loss are indicative of less effective learning.

Best DQN configuration:

- Learning Rate: 0.001
- Gamma: 0.45
- Batch Size: 256
- Max Reward: 338
- Convergence Time Steps: 800K

Notes: This configuration provides high stability and reasonable convergence time steps.

PPO Results

PPO results are reported in the following table.

Table 7.3: PPO hyper parameters configurations and results of training

Learning rate	Gamma	Batch size	Max reward	Convergence time steps	Notes
0.001	0.45	64,256	65	NONE	Instability, no convergence
0.001	0.45	128	292	NONE	Instability, no convergence
0.001	0.75	64,128,256	360	NONE	Instability, no convergence
0.001	0.99	64	500	>3M	Slow convergence

0.001	0.99	128,256	236	NONE	Instability, no convergence
0.0001	0.45	64	250	NONE	Instability, no convergence
0.0001	0.45	128,256	71	800K	Stable but low reward
0.0001	0.75	64,128,256	92	800K	Stable but low reward
0.0001	0.99	64	90	NONE	Instability, no convergence
0.0001	0.99	128,256	70	200K	Stable but low reward
0.0002	0.45	64	162	NONE	Instability, no convergence
0.0002	**0.45**	**128**,256	**97**	**300K**	**Stable but low reward**
0.0002	0.75	64	286	>3M	Instability, no convergence
0.0002	0.75	128,256	45	<100K	Too fast convergence, low reward
0.0002	0.99	64	82	>3M	Instability, no convergence
0.0002	0.99	128,256	49	>3M	Instability, no convergence
0.0003	0.45	64,128	110	>3M	Instability, no convergence
0.0003	0.45	256	68	600K	Stable but low reward
0.0003	0.75	64,128	260	NONE	Instability, no convergence
0.0003	0.75	256	45	700K	Stable but low reward
0.0003	0.99	64,128,256	178	NONE	Instability, no convergence
0.0004	0.45	64	390	NONE	Instability, no convergence
0.0004	0.45	128,256	180	NONE	Instability, no convergence
0.0004	0.75	64,128	266	NONE	Instability, no convergence
0.0004	0.75	256	194	NONE	Instability, no convergence
0.0004	0.99	64	330	NONE	Instability, no convergence
0.0004	0.99	128,256	88	NONE	Instability, no convergence
0.0005	0.45	64,128	340	NONE	Instability, no convergence
0.0005	0.45	256	116	NONE	Instability, no convergence
0.0005	0.75	64,128	285	NONE	Instability, no convergence
0.0005	0.75	256	98	NONE	Instability, no convergence
0.0005	0.99	64	286	NONE	Instability, no convergence
0.0005	0.99	128,256	140	NONE	Instability, no convergence
0.00005	0.45,0.75,0.99	64,128,256	75	500K	Stable but low reward

PPO, characterized by its stability and sample efficiency, exhibited robust performance across diverse configurations. Key observations from the experiments include:

- Across multiple configurations, instability and failure to converge were predominant outcomes, evidenced by the absence of a maximum reward and convergence within specified time steps.
- Higher learning rates and gamma values, combined with varying batch sizes, contributed to instability and hindered convergence.
- Several configurations exhibited stable but low reward outcomes, indicating suboptimal performance despite convergence.
- Notably, lower learning rates coupled with moderate gamma values and smaller batch sizes demonstrated relatively more stable behavior, although convergence was often slow.
- Conversely, higher learning rates and gamma values led to unstable training dynamics and failed to achieve convergence within the specified time steps.
- The loss analysis, derived from the three best-performing models, indicates comparable entropy loss across configurations, but training loss was notably better for the first model.

Best PPO Configuration:

- Learning Rate: 0.0002
- Gamma: 0.45
- Batch Size: 128
- Max Reward: 97
- Convergence Time Steps: 300K
- Notes: Despite stable but low rewards, this configuration demonstrates promising performance with a relatively shorter convergence time.

A2C Results

In Table 7.3 Instead, the performance evaluation of A2C models with the same hyperparameter configurations has been reported.

Table 7.4: A2C hyper parameters configurations and results of training

Learning rate	Gamma	Batch size	Max reward	Convergence time steps	Notes
0.001	0.45,0.75,0.99	64,128,256	43	2.5K	Convergence too quick
0.0001	0.45,0.75,0.99	64,128,256	45	3K	Convergence too quick
0.0002	0.45,0.75,0.99	64,128,256	44	3K	Convergence too quick
0.0003	0.45,0.75,0.99	64,128,256	45	2.9K	Convergence too quick
0.0004	0.45,0.75,0.99	64,128,256	45	3K	Convergence too quick
0.0005	0.45,0.75,0.99	64,128	45	3K	Convergence too quick
0.0005	**0.45**,0.75,0.99	**256**	**49**	**3K**	**Convergence too quick**
0.00005	0.45,0.75,0.99	64,128	47	3K	Convergence too quick
0.00005	0.45,0.75,0.99	256	46	2.9K	Convergence too quick

As we can see, no convergence is reached for all parameter configurations. This behavior could be due to the wrong parameter space explored with the selected configurations, and the default value for other specific parameters should probably be configured differently. Therefore, considering acceptable DQN results, further investigations and configurations have not been taken into consideration. The failure to achieve satisfactory convergence underlines the complexity of the problem and the need for further research to understand the underlying causes of this inefficiency better. The development of more effective strategies to improve the performance of the A2C model in this context remains a key area of interest for future studies.

Best A2C Configuration:

- Learning Rate: 0.0005
- Gamma: 0.45
- Batch Size: 256
- Max Reward: 49
- Convergence Time Steps: 3K
- Notes: No convergence is reached.

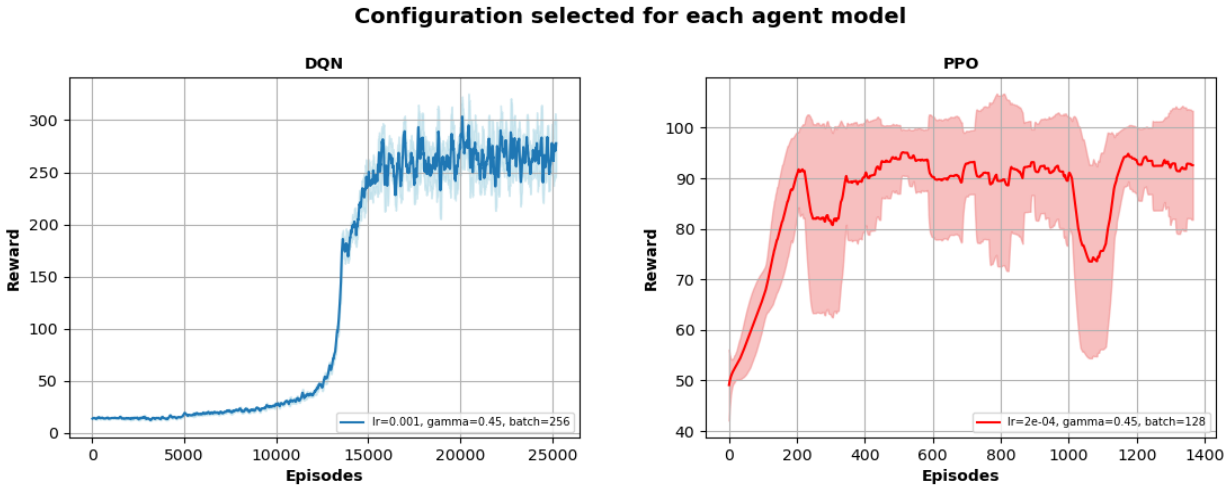


Figure 7.5: Best DQN and PPO reward functions

Best RL models configuration

Our findings reveal that **DQN** emerged as the superior performer, achieving a maximum reward of **338** with a learning rate of 0.001, gamma of 0.45, and batch size of 256, as shown in Figure 7.5. This configuration exhibited high stability and convergence during training. Conversely, PPO and A2C encountered challenges in achieving stable performance within the designated time frame.

These results highlight the importance of careful RL algorithm selection and hyperparameter tuning for achieving optimal performance in complex network optimization tasks. DQN's effectiveness can be attributed to its well-suited architecture for handling discrete action spaces (data center selection) and its ability to learn an effective policy from past experiences.

7.1.5 RL Training Strategies to avoid Overfitting

To ensure robust performance across different network conditions, we developed specific training strategies to prevent overfitting. These strategies are designed to address the inherent variability and complexity of real-world network environments and to guarantee that the learned policy can adapt effectively across heterogeneous operational conditions. Specifically, we employed empirical measurements from telco field operator experience to introduce best-practice knowledge within the optimization algorithm.

Overfitting in RL typically manifests when the agent memorizes specific patterns or configurations observed during training, thereby compromising its ability to generalize to unseen scenarios. This issue is particularly critical in 5G edge computing, where traffic patterns, resource availability, and environmental factors fluctuate dynamically. The adopted mitigation approach integrates empirical domain knowledge, policy regularization, feature diversity, and dynamic simulation to cultivate a more resilient learning process.

A key component of this strategy is the integration of expert-derived constraints that reflect real-world telco infrastructure practices. Specifically, the selection of the User Plane Function (UPF) — the virtualized network component responsible for routing user data — is restricted according to two operational thresholds. First, the candidate UPF must exhibit a CPU usage rate below **90%**, ensuring that selected resources are not already overburdened and thus avoiding performance degradation. Second, any new UPF must demonstrate a packet loss rate that improves upon the previously selected UPF by at least **20%**, thereby promoting continuous enhancement in data delivery quality.

We also implemented a refined reward system that issues positive rewards when latency and packet loss fall below dataset mean values, with penalties for higher values, encouraging the model to select high-performance UPFs. An additional accuracy bonus is applied when the selected UPF matches the optimal target, reinforcing

the selection of the highest-performing data centers. To build resilience against overfitting, we incorporated diverse traffic simulations across different times of day (Night, Busy Hour, Daytime), each with unique metrics for bandwidth, CPU load, and session duration. This simulation creates a varied dataset that closely reflects real-world conditions.

The Italian data centers in Milan, Rome, and Cosenza represent various network conditions and form the basis for testing our RL optimization strategies. Milan hosts the centralized core network with control plane functions, while each city has a UPF to manage the user plane. Each data center has a bandwidth cap to simulate real-world limitations. To build resilience against overfitting, data traffic was generated using a script that models different times of the day (Night, Busy Hour, Daytime) and traffic profiles (e.g., CPU load, bandwidth consumption). Each simulation cycle varies values such as throughput and session duration to reflect realistic usage patterns.

The goal of the RL model is to identify and select the optimal data center to minimize latency between the gNodeB and edge node. This RL environment emphasizes latency as the primary objective but incorporates additional Key Performance Indicators (KPIs) to ensure comprehensive performance optimization. For example, while low-latency data centers are preferred, high CPU utilization may affect overall network efficiency, so balancing multiple metrics is essential.

To facilitate optimal data center selection, we embedded a policy with weighted features within the RL environment, prioritizing latency and packet loss. This weighted approach allows the RL agent to consider multiple aspects of data center performance, ensuring decisions align with broader network goals and with empirical measurements.

To complement these constraints, the reward function employed by the RL agent was carefully redesigned to embed both performance optimization and sustainability considerations. The new reward function is constructed as a weighted aggregation of normalized key performance indicators (KPIs), ensuring that the agent learns to make decisions based on a multifactorial assessment of system health and efficiency. Each KPI is scaled into the [0, 1] range using MinMax normalization, allowing for consistent integration across heterogeneous metrics.

The reward function R used to guide the reinforcement learning agent during training is defined as a composite expression that combines a weighted sum of normalized performance features, an action-based correctness bonus, and two auxiliary penalty or reward terms related to key performance indicators (latency and packet loss). The full expression is given by:

$$R = \left(\sum_{i=1}^{n} f_i \cdot w_i \right)^2 + B + A_1 + A_2$$

In this formulation:

- f_i denotes the normalized value of the i-th feature (e.g., CPU usage, memory usage, average latency, etc.);
- w_i is the weight assigned to the i-th feature, reflecting its relative importance in the reward calculation;
- The squared term emphasizes the contribution of higher aggregated feature scores, promoting configurations that simultaneously optimize multiple KPIs.

The remaining additive terms incorporate policy-level reinforcement signals based on action correctness and quality of service metrics.

Correctness Bonus (*B*)

The scalar term *B* represents a correctness bonus, awarded when the agent's selected action (i.e., the chosen data center) aligns with the optimal target defined by the environment's policy. Formally:

B: a bonus based on the action's correctness:

$$B = \begin{cases} 8.0, & \text{if action} = \text{target} \\ 0, & \text{otherwise} \end{cases}$$

This component serves to reinforce correct predictions, significantly boosting the reward when the optimal decision is made.

The terms A1 and A2 are used to penalize suboptimal latency and packet loss conditions, further refining the reward to ensure robust performance across diverse network states.

Latency Bonus (A_1)

The bonus term A_1 is introduced to further incentivize selections that yield lower-than-average latency. It is defined as:

A_1: a bonus based on **latency average**:

$$A_1 = \begin{cases} 0.7, & \text{if latency\_avg} < \overline{\text{latency\_avg}} \\ 0, & \text{otherwise} \end{cases}$$

where $\overline{\text{latency\_avg}}$ represents the mean average latency across the training dataset. This component encourages the agent to prefer edges that maintain low latency under current conditions.

Packet Loss Penalty (A_2)

To penalize configurations that result in degraded data transmission reliability, the function includes a penalty term A_2 based on packet loss:

A_2: a penalty based on **packet loss**:

$$A_2 = \begin{cases} -0.3, & \text{if packet\_loss} > \overline{\text{packet\_loss}} \\ 0, & \text{otherwise} \end{cases}$$

where $\overline{\text{packet\_loss}}$ is the average packet loss in the dataset. This penalty discourages actions that would result in above-average packet loss, thereby promoting robust and reliable connections.

The complete reward function encapsulates both performance-oriented and policy-based components. The weighted sum captures the trade-offs between multiple KPIs, while the additive bonuses and penalties provide

targeted incentives and constraints that align with the overall objective: to dynamically select the best-performing and most sustainable edge node in a 5G network, under varying real-time conditions. This structure enhances the interpretability, robustness, and generalization capacity of the RL agent.

Moreover, to increase the training set's variability and representativeness, the dataset was extended using a traffic simulation engine that emulates distinct usage scenarios corresponding to different times of day: night, Busy Hour, and Daytime. These profiles simulate fluctuating throughput demands, CPU loads, and session durations, thus creating a broad and realistic spectrum of network conditions. This synthetic diversity ensures that the RL agent is exposed to a wide range of edge-case behaviors during training, thereby improving its generalization capabilities.

Finally, to further safeguard against overfitting, early stopping criteria and reward variance monitoring were adopted during training. Training episodes were terminated early if the reward function stabilized prematurely or exhibited a degenerate convergence pattern. Additionally, reward and loss evolution curves were periodically assessed to detect signs of memorization or performance collapse.

In summary, the integration of expert constraints, a carefully engineered and weighted reward function, synthetic traffic diversification, and continuous performance monitoring collectively contributes to the development of an RL model that is robust, adaptive, and well-suited for deployment in dynamic and resource-constrained 5G environments.

7.1.6 RL Evaluation

To assess the effectiveness of the RL agent in optimizing User Plane Function (UPF) selection across distributed edge data centers, we implemented a supervised evaluation pipeline that compares the agent's predictions against a ground truth reference. This ground truth is established through a deterministic labeling mechanism derived from domain-specific heuristics, ensuring that the evaluation is aligned with practical telco operations and service quality objectives.

The evaluation process begins with the ingestion of real-time or simulated **UPF telemetry data**, comprising key performance indicators (KPIs) such as CPU usage, memory utilization, disk capacity, network throughput (inbound and outbound), latency (minimum, average, and maximum), and packet loss. These features are extracted for each candidate data center—typically Milan, Rome, and Cosenza—and normalized for use as input to the RL environment.

This telemetry data is simultaneously processed along two paths:

1. **Ground Truth Labeling**: A rule-based decision module computes the optimal data center label for each observation based on a multi-criteria policy that balances latency minimization, resource availability, and sustainability metrics. This output constitutes the **Target** label and represents the best data center selection for the given network state, as determined by expert logic.
2. **RL Agent Inference**: In parallel, the trained RL agent receives the same telemetry observation and produces a **Predicted** label by selecting the data center it deems optimal according to its learned policy. This prediction represents the agent's decision-making outcome in the current environment state.

To quantify the accuracy of the RL policy, the predicted and target labels are compared using standard classification metrics:

- **Accuracy** measures the proportion of correct predictions across the total number of predictions:

$$\text{Accuracy} = \frac{\text{Correct Predictions}}{\text{All Predictions}} = 77\%$$

- **Precision** evaluates the fraction of correctly predicted optimal data centers out of all those predicted as optimal by the agent:

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} = 77\%$$

- **Recall** quantifies the fraction of truly optimal data centers that the agent successfully identified:

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} = 76\%$$

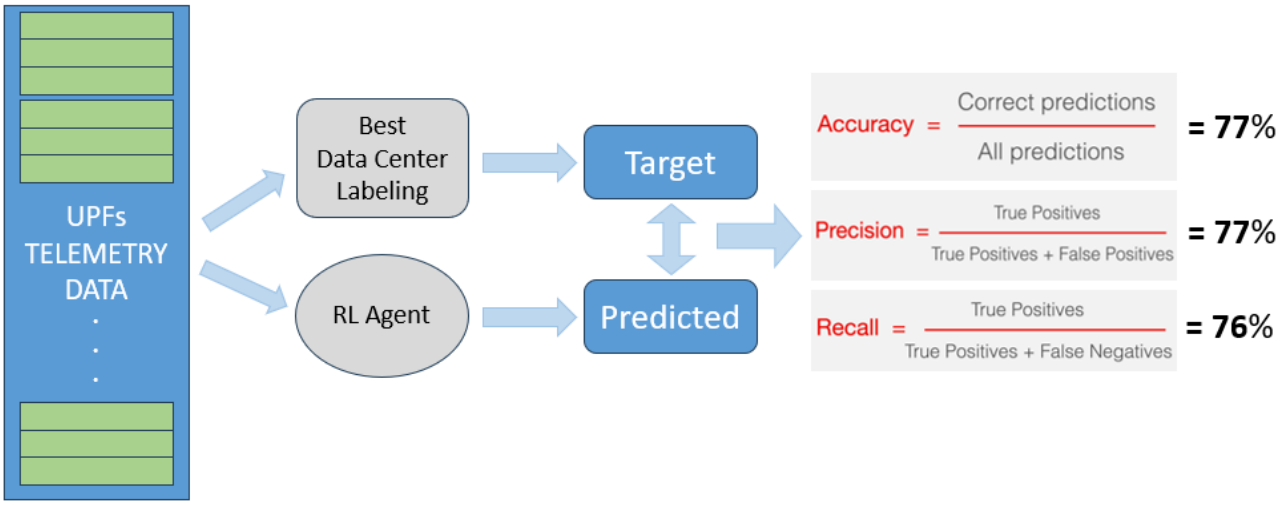


Figure 7.6: RL Evaluation Process

These results indicate a strong alignment between the RL agent's decisions and the expert-defined optimal selections, confirming that the agent has effectively learned the underlying policy from environment interaction. The high precision and recall values further demonstrate that the agent not only avoids frequent misclassifications but also maintains a reliable sensitivity to valid target selections.

This evaluation methodology serves a dual purpose: it provides a measurable validation of the RL agent's training effectiveness. It offers operational confidence for real-world deployment within a latency-sensitive, multi-metric optimization context. The results achieved thus far suggest that the deployed RL policy is capable of delivering high-quality edge data center selection, contributing to performance improvements and sustainability-aware orchestration within 5G network infrastructures.

Carbon Intensity: Including the Sustainability in the 5G Network Optimization

In the context of intelligent orchestration across the cloud-edge continuum, the integration of environmental sustainability metrics is of paramount importance. Among these, **carbon intensity** represents a critical parameter for evaluating the environmental impact of data center operations. Therefore, carbon intensity has been incorporated as a primary feature in the RL-based selection strategy for UPF placement, thereby enabling

the optimization process to account not only for performance indicators such as latency and packet loss but also for the greenhouse gas emissions associated with computational resource allocation.

Carbon intensity is defined as the amount of carbon dioxide (CO_2) emitted per unit of electricity consumed within a given geographical zone. It is typically expressed in **grams of CO_2 equivalent per kilowatt-hour (gCO_2-eq/kWh)**. This measure encapsulates the environmental burden of energy usage by quantifying the lifecycle emissions generated to produce electricity consumed at a particular location and time. Importantly, this metric includes not only direct CO_2 emissions, but also contributions from other greenhouse gases (GHGs), converted into CO_2-equivalent terms based on their 100-year global warming potential (GWP). For example, one gram of methane (CH_4) released into the atmosphere is considered to have the same climatic impact over a century as approximately 34 grams of CO_2.

The carbon intensity of a specific zone is fundamentally linked to its **electricity generation mix**—that is, the proportion of energy produced from renewable, low-carbon, and fossil-based sources. Regions relying heavily on coal, oil, or gas-fired power plants will exhibit significantly higher carbon intensity values compared to those sourcing electricity predominantly from hydroelectric, wind, solar, or nuclear energy. Temporal fluctuations also occur, as the energy mix changes over the course of the day in response to demand and intermittent renewable generation.

To ensure an accurate and science-based estimation of carbon intensity, this study relies on lifecycle emission factors, which are computed in accordance with the methodology outlined by the **Intergovernmental Panel on Climate Change (IPCC)** in its Fifth Assessment Report (2014)[29]. These emission factors incorporate all stages of electricity production, including fuel extraction and refinement, power plant construction and decommissioning, operational emissions during energy generation, and final disposal. The use of lifecycle analysis enables a comprehensive evaluation of the carbon cost associated with each kilowatt-hour consumed.

Carbon intensity values are retrieved dynamically via API from platforms such as **Electricity Maps**[30], which provide high-resolution, location-specific carbon intensity data. This data accounts for cross-border electricity exchanges, using flow-tracing methodologies to allocate emissions from imported electricity back to their origin countries and sources. Two main types of emission factors are used in this process:

1. *default factors*, which are standardized values derived from scientific literature,
2. *zone-specific factors*, which are grounded in localized emission reporting data.

For example, in Europe and the United States, publicly available direct power plant emission statistics can be combined with real-time electricity production data to calculate precise, regionally resolved emission factors.

Table 7.5: Default Emission Factors used by Electricity Maps

Mode	Emission factor (gCO2eq/kWh)	Category	Source
battery discharge	301	Renewable (default)	World average intensity by Electricity Maps
biomass	230	Renewable	IPCC 2014
coal	820	Fossil	IPCC 2014
gas	490	Fossil	IPCC 2014
geothermal	38	Renewable	IPCC 2014
hydro	24	Renewable	IPCC 2014
hydro discharge	301	Renewable (default)	World average intensity by Electricity Maps

[29] IPCC (2014) Fifth Assessment Report
[30] https://www.electricitymaps.com/

nuclear	12	Low-carbon	IPCC 2014
oil	650	Fossil	UK Parliamentary Office of Science and Technology
solar	45	Renewable	IPCC 2014
unknown	700	Fossil	Assumes thermal (coal, gas, oil)
wind	11	Renewable	IPCC 2014

Electricity Maps aggregates emission data from multiple sources, including peer-reviewed publications, governmental reports, and life-cycle meta-analyses. In this way, the platform supports both **operational emission factors**—which consider only the direct emissions during energy production—and **full lifecycle factors**, which are more appropriate for sustainability-aware system optimization.

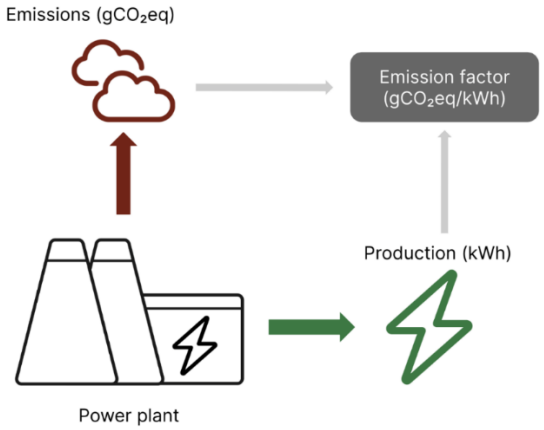


Figure 7.7: Schematic representation of emission factors

In the RL environment, carbon intensity is treated as a first-class citizen among the key performance indicators. A normalized carbon intensity score is computed for each candidate data center and incorporated into the agent's reward function with a significant weight (1.0), equal to that of latency. This modeling choice reflects the growing need to balance performance excellence with environmental stewardship in the design and deployment of modern 5G infrastructures. By embedding sustainability into the decision-making loop, the proposed solution contributes to the development of intelligent, low-carbon network systems aligned with the goals of the European Green Deal and the digital transition.

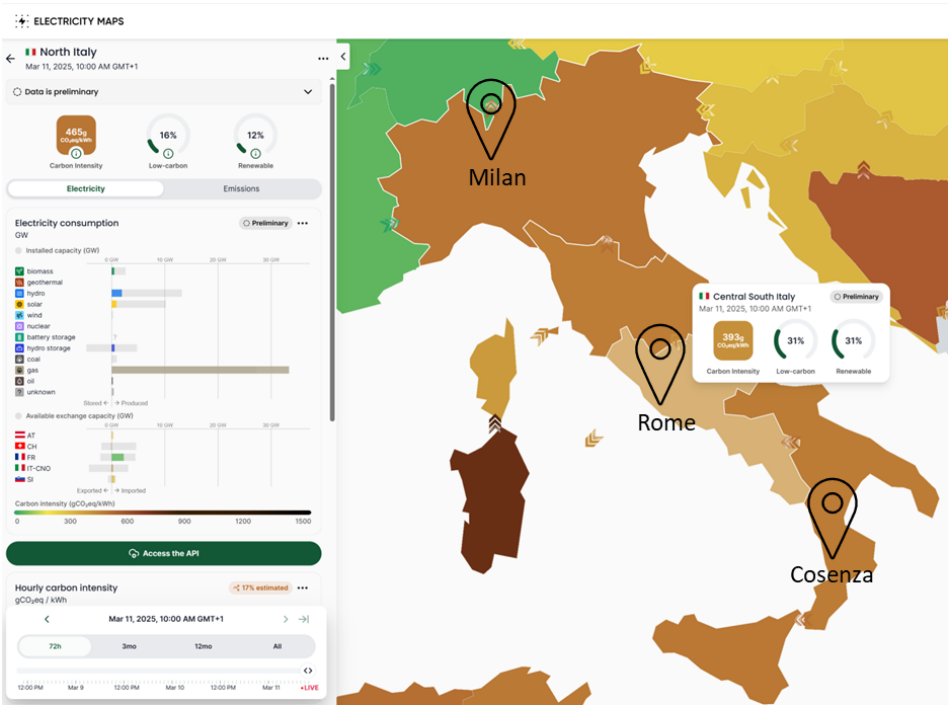


Figure 7.8: Electricity Maps of our Edge Nodes

To accurately capture the regional variability of carbon emissions, **carbon intensity values were dynamically retrieved via API** using the **geographical coordinates (latitude and longitude)** of the three target data center locations: Milan, Rome, and Cosenza. These values were sourced from Electricity Maps that provides zone-specific carbon intensity data derived from grid composition, cross-border electricity flows, and generation source emissions. The use of **flow-tracing algorithms** ensures that emissions associated with imported electricity are correctly attributed to their origin, thereby enhancing the precision of the carbon impact estimation.

In the RL framework developed for UPF optimization, **carbon intensity was treated as a primary input feature** and explicitly incorporated into the agent's **reward function**. The reward function aggregates a weighted sum of normalized features that reflect the operational state of candidate data centers. The **feature weights** used in this study are summarized below:

Table 7.6: Feature Weights with Carbon Intensity

Feature	Weight
CPU Usage (%)	0.6
Memory Usage (%)	0.5
Disk Usage (%)	0.5
Network In (%)	0.7
Network Out (%)	0.7
Latency Average (ms)	1.0

Latency mdev (ms)	0.2
Packet Loss (%)	0.9
Carbon Intensity (gCO_2/kWh)	**1.0**

The high weight assigned to carbon intensity (1.0)—equal to that of average latency—underscores the dual priority placed on both network performance and environmental impact. This design choice ensures that, all else being equal, the agent will favor routing decisions that lead to data centers with lower carbon footprints. As a result, the RL agent learns to strike a balance between selecting resources that optimize technical KPIs and those that contribute to sustainability objectives.

By embedding carbon-aware decision-making into the control logic of 5G service placement, the methodology presented in this study offers a tangible pathway toward greener and more energy-efficient network operations. This is particularly relevant for large-scale, geographically distributed infrastructures, where regional disparities in electricity generation mix can lead to significant variations in carbon intensity. Integrating these variations into the orchestration policy allows for context-aware optimization that extends beyond traditional QoS (Quality of Service) metrics.

In conclusion, the inclusion of real-time, location-specific carbon intensity as a decision-making factor enables the development of RL-driven network controllers that are not only performance-oriented but also climate-conscious. This aligns with broader European goals for digital sustainability and provides a scalable approach for minimizing the carbon impact of data-intensive edge services in the evolving landscape of 5G and beyond.

Real-time Inference Workflow

Our comprehensive evaluation of RL algorithms aimed at optimizing latency in 5G networks revealed that the DQN algorithm outperformed other tested models, specifically PPO and A2C. Through meticulous hyperparameter tuning, DQN achieved a maximum reward of 338, employing a learning rate of 0.001, a gamma value of 0.45, and a batch size of 256. This configuration demonstrated remarkable stability and convergence throughout the training process, highlighting DQN's superior capability in managing the discrete action space inherent in data center selection tasks.

In contrast, PPO and A2C faced significant challenges in attaining stable performance within the designated timeframe. Their inability to achieve comparable results underscores the importance of selecting an appropriate RL algorithm and fine-tuning its hyperparameters to meet the specific requirements of complex network optimization tasks.

The practical implementation of the selected RL model, specifically within the context of a real-time 5G network optimization scenario, is elucidated in the provided diagram in Figure 7.9. This diagram captures the end-to-end process, encompassing data collection, transmission, processing, and feedback.

Figure 7.9: Real-time Inference Workflow

The provided diagram illustrates the process of real-time inferences within a 5G network infrastructure, leveraging Prometheus for monitoring, a backend system for data handling, and a machine learning (ML) agent for processing and generating results. The process is detailed as follows:

1. Network Monitoring (Prometheus):
   - Prometheus is depicted as the monitoring system within the network. It is responsible for collecting metrics and data from various network components.
   - The network, monitored by Prometheus, gathers real-time data about network performance, user activity, and other relevant metrics.
2. Data Transmission to Backend:
   - The collected data is sent from Prometheus to the Backend system using HTTP POST requests.
   - Two types of data transmissions are shown:
     - `POST: Input` – This request sends the raw input data (metrics and performance indicators) from the network to the backend.
     - `POST: result` – This request sends processed results or inferences back to the network after being processed by the ML agent.
3. Backend System:
   - The Backend system acts as an intermediary, receiving data from the network and managing the data flow between the network and the ML agent.
   - The backend stores the incoming data and prepares it for further processing by the ML agent.
4. Communication with ML Agent (RabbitMQ):
   - The backend utilizes RabbitMQ, a message broker, to facilitate communication between the backend and the ML agent.
   - Data is published to RabbitMQ as `Input`, where it is queued and awaits processing by the ML agent.
5. ML Agent:
   - The ML agent is responsible for processing the input data and generating inference results.
   - The ML agent retrieves the queued data (Input) from RabbitMQ, processes it, and performs the necessary computations and analysis.
   - Once the processing is complete, the ML agent sends the results back to the backend using a `POST: results` request.
6. Result Integration and Feedback:
   - The backend receives the processed results from the ML agent.
   - These results are then sent back to the network (Prometheus) through a `POST: result` request.
   - The network can utilize these results to make real-time adjustments and optimizations, thereby improving performance and user experience.

This diagram and its associated processes illustrate a streamlined workflow for real-time data processing and inference within a 5G network, emphasizing the critical role of efficient data handling and communication between monitoring systems, backend infrastructure, and ML processing units. The integration of Prometheus, RabbitMQ, and ML agents ensures that the network can dynamically adapt to changing conditions, providing a robust solution for real-time network optimization.

Initially, the network's performance metrics and user activity data are continuously monitored by Prometheus, a robust monitoring system depicted in the diagram. Prometheus collects real-time data from various network components, ensuring comprehensive coverage of the network’s operational status.

Subsequent to data collection, the metrics are transmitted to a backend system through HTTP POST requests. The backend, represented centrally in the diagram, serves as the intermediary, receiving raw input data (`POST: Input`) from the network and processed results (`POST: result`) from the ML agent.

The backend leverages RabbitMQ, a message broker, to facilitate efficient communication with the ML agent. This step involves publishing the input data to RabbitMQ, where it is queued for processing. The ML agent then retrieves this data, performs the necessary computations, and generates inference results.

The processed results are sent back to the backend via `POST: results`, where they are subsequently relayed to Prometheus. This feedback loop enables the network to dynamically adjust and optimize its performance based on real-time insights, thereby enhancing overall efficiency and user satisfaction.

To support the described real-time inference process, a comprehensive API, defined using the OpenAPI Specification (version 3.0.1), underpins the communication between various system components. The JSON document outlines the API's endpoints, methods, parameters, and responses, ensuring a standardized approach to data exchange and management.

The API comprises several key endpoints categorized under three primary tags:

1. “DataCenter Controller” - responsible for managing data centers.
2. “Inference Result Controller” - overseeing the management of inference results.
3. “Snapshot Controller” - dedicated to the management of snapshots.

The paths section details the API endpoints and their respective HTTP methods, summaries, operation IDs, parameters, and responses. For instance, the endpoint “/api/v1/snapshots” supports both GET and POST methods under the “Snapshot Controller” tag:

- The GET method retrieves a list of snapshots, with possible responses including status codes 403 (Forbidden), 401 (Unauthorized), and 404 (Resource not found).
- The POST method allows the creation of a snapshot object, requiring a JSON request body. It also lists possible responses such as 403, 401, and 404, similar to the GET method.

Similarly, the “/api/v1/results” endpoint, managed by the “Inference Result Controller”, facilitates the retrieval and creation of inference result objects. The GET method includes optional query parameters for pagination (pageNumber and pageSize), and the POST method mandates a JSON request body to create a new inference result, with detailed error responses for common HTTP status codes.

The “/api/v1/datacenters” endpoint, under the “DataCenter Controller” tag, provides functionalities to list and create data center objects. Each method outlines the expected responses, emphasizing security and error handling.

In the components section, the document defines schemas for various objects such as DataCenterState, Snapshot, InferenceResult, and DataCenter. These schemas provide a detailed blueprint of the data structures used within the API:

- The DataCenterState schema encompasses properties like client_id, start_time, end_time, and various performance metrics (CPU, memory, disk usage percentages, network statistics, and latency measures).
- The Snapshot schema includes properties for snapshotId (UUID) and an array of DataCenterState objects.
- The InferenceResult schema describes properties such as resultId (UUID), snapshotId (UUID), action (integer), reward (float) and an optional creationDate field (Date) created at insertion time.
- The DataCenter schema details properties including datacenterId (integer), location (string), and description (string).

Therefore, this structured and detailed OpenAPI definition facilitates the interaction with the 5G network management API, providing clear guidelines for developers and stakeholders to integrate and utilize the API effectively. By adhering to standardized specifications and including comprehensive metadata, tags, paths, and components, the document ensures robust, secure, and efficient management of 5G network resources.

The synergy between the optimal RL model, real-time inference workflow, and a robust API framework facilitates an efficient and scalable solution for 5G network optimization. By leveraging the DQN algorithm's superior performance, the network can dynamically adapt to changing conditions, ensuring minimal latency and enhanced user experience. The detailed API definition supports seamless communication between system components, enabling real-time data processing and feedback loops essential for maintaining optimal network performance

Backend Architecture

The Architecture of the backend is depicted in Figure 7.10. It uses a Controllers-Services-Repositories pattern, a very common design pattern in software development, particularly within the context of web applications and microservices. This architecture promotes a clean, organized, and scalable codebase. It enhances maintainability, testability, and flexibility, making it easier to develop robust and scalable applications. This kind of pattern is particularly beneficial in complex systems where different aspects of the application need to evolve independently.

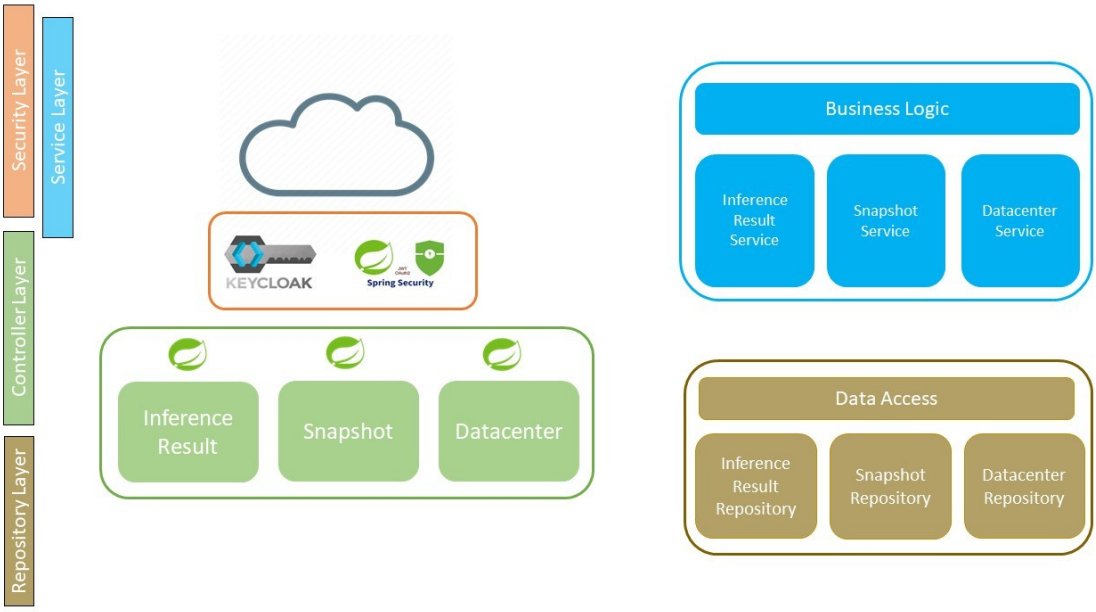


Figure 7.10: Backend Architecture

Security Layer

1. This layer ensures that only authenticated users can access the API and that users have the appropriate permissions to perform specific actions. It provides Authentication & Authorization.
2. Authentication
   - User Authentication: Verifies the identity of users trying to access the API. This typically involves checking user credentials (such as username and password) against Keycloak (OAuth2 provider).
   - Token Management: Handles authentication tokens (OAuth2 tokens against Keycloak). This includes issuing, validating, and invalidating tokens as needed.
3. Authorization
   - Access Control: Determines what authenticated users are allowed to do. This involves defining roles and permissions and enforcing them based on user roles. In this solution it is present the role 'OPERATOR' the only role that can use the API to ask for and inference result.
   - Method Security: Protects individual methods within services. Using annotations @PreAuthorize you can restrict access to certain methods based on user roles or specific conditions.

Services Layer

The Service Layer is where all the business logic take place, it plays a critical role in this solution. It guarantees the integrity of the information generated by the ML Agent and all the data passed to the algorithm for the inference phase. It also guarantees consistency and enforces business rules implementation.

Repository Layer

This role of the repository layer is abstracting the data access logic from the rest of the application. This separation of concerns helps in maintaining a clean architecture. In this solution it abstract data access logic for accessing DynamoDB database. Its roles are:

1. Data access Abstraction
   - Encapsulation of Data Logic: The repository layer encapsulates all the logic required to access data sources (DynamoDB in this case). This means that the details of how data is fetched, saved, updated, or deleted are hidden from the service and controller layers.
   - Consistent Interface: Repositories provide a consistent interface for data operations. This makes it easier to switch data sources or change the underlying data access technology without affecting other parts of the application.
2. CRUD Operations
   - Create, Read, Update, Delete: Offers basic CRUD operations.
3. Decoupling Business Logic from Data Access

Controller Layer

The controller layer is responsible for managing HTTP requests and responses, delegating business logic to the service layer, and ensuring that input is validated, and output is correctly formatted. This separation of concerns enhances the communication of the network with the ML Agent. A brief description of the functionalities of this layer can be found on the **Section 4**. of this document.

8 Dynamic storage placement and management

Object Storage systems are used for persistently storing and retrieving immutable blocks of data, with an industry-standard API pioneered by Amazon Web Services' Simple Storage Service (S3[31]). The basic concepts are the *object*, which contains the arbitrary data, its unique key, and optional metadata, as well as *buckets* which are containers of objects that define how and where objects are stored, control their access, assign lifecycle rules and additional attributes. Object storage services require users to create and configure a bucket before any objects can be added, and some of these configuration options, most notably the data location, cannot be changed later. Most systems store buckets in a single geographical region with a fixed level redundancy and no guarantees of performance metrics like latency or transfer speed.

When configuring a bucket, application developers must have strong assumptions about where objects will be accessed from and what will be the volume of traffic. If the actual demand originates much further, end users experience high latency. In case the bucket is rarely used, either from the beginning or demand decreases significantly over time, its high redundancy storage configuration costs more than what would be sufficient.

MLSysOps aims to introduce a highly dynamic, ML-driven approach to adjust the redundancy level and geographical distribution of buckets based on the actual traffic they experience in near real-time, to minimize operating costs while satisfying access speed requirements given by the MLSysOps application that owns the bucket. To achieve this, MLSysOps continuously monitors the origin and magnitude of demand for objects in managed buckets and employs an ML model to automatically trigger changes in the underlying storage representation of objects when the bucket is under, or overprovisioned, or the data location(s) is not aligned with the current demand.

The storage system being integrated into MLSysOps is the S3-compatible object storage extension of the SkyFlok[32] secure distributed file sharing and storage solution of CC. This platform uses a combination of encryption and erasure coding (specifically, *Random Linear Network Coding*) to create multiple redundant fragments of customer data, which are distributed to user-selected locations in 3rd party commercial cloud storage services like Amazon S3, OVH Cloud or Microsoft Azure. SkyFlok currently uses a fixed configuration of redundancy and locations for buckets. In MLSysOps, the system is extended with the ML-based dynamic storage re-configuration ability.

Problem description

The high-level purpose of the ML component in this context is to find the least expensive storage configuration for managed buckets that satisfies the performance requirements declared by the MLSysOps application and respects given geographical restrictions. Specifically, it needs to optimize the following objectives simultaneously:

1. Match the download speed requirement given by the user
2. Respect geographical restrictions (if any)
3. Minimize the redundancy of objects (prefer less data overhead)
4. Minimize storage cost (prefer storage locations with lower egress and storage costs)
5. Minimize the cost of changing the storage configuration (consider the cost of data migration itself)

Objectives 1 and 2 are directly derived from the user-given MLSysOps application descriptor. Objectives 3 and 4 ensure that the operational costs of the bucket are as low as possible, while objective 5 makes the model aware of the cost of its own actions. Since there are many possible storage configurations and the demand is changing over time, ML is particularly well suited to solve this problem.

In practice, the ML model is expected to detect and react to two common changes in traffic patterns. When the origin of bucket access shifts to a new geographical region, the data stored in the bucket should also shift to

[31] https://aws.amazon.com/s3/
[32] https://www.skyflok.com/

locations that are closer to the new demand. Furthermore, when the volume of traffic increases significantly, the model should find faster storage locations, even if they are more expensive, to match the performance requirements. Later if the demand diminishes, it should switch back to a cheaper storage configuration.

System design

The ML-driven object storage service is composed of two high-level components (see Figure 8.1). The SkyFlok platform is responsible for providing an S3-compatible interface to end-users while keeping track of buckets, their configuration (called *Storage Policy*) as well as metadata of the stored objects and ensuring that object contents are stored with the prescribed redundancy and distributed to the selected storage locations. The MLSysOps framework includes an ML-based component for dynamic storage management and stores the user-given high-level bucket requirements, as well as historical data of bucket traffic and storage location performance (both provided by SkyFlok). Its main purpose is to periodically re-evaluate whether current Storage Policies of the managed buckets are optimal, considering recent traffic and user-given requirements, and trigger the change if a better configuration is found.

Every SkyFlok bucket has an assigned Storage Policy, which defines the list of storage locations where the erasure coded fragments are located and the level of redundancy (how many of the total fragments are redundant). When a new object is uploaded to a bucket, its Storage Policy is loaded, object contents are processed accordingly, and the resulting encrypted-encoded fragments are distributed to the target storage locations. Download is a similar process in reverse (Figure 8.2): after obtaining the object metadata, the minimum number of fragments are transferred from the closest location(s), data is decoded, decrypted and streamed back to the client.

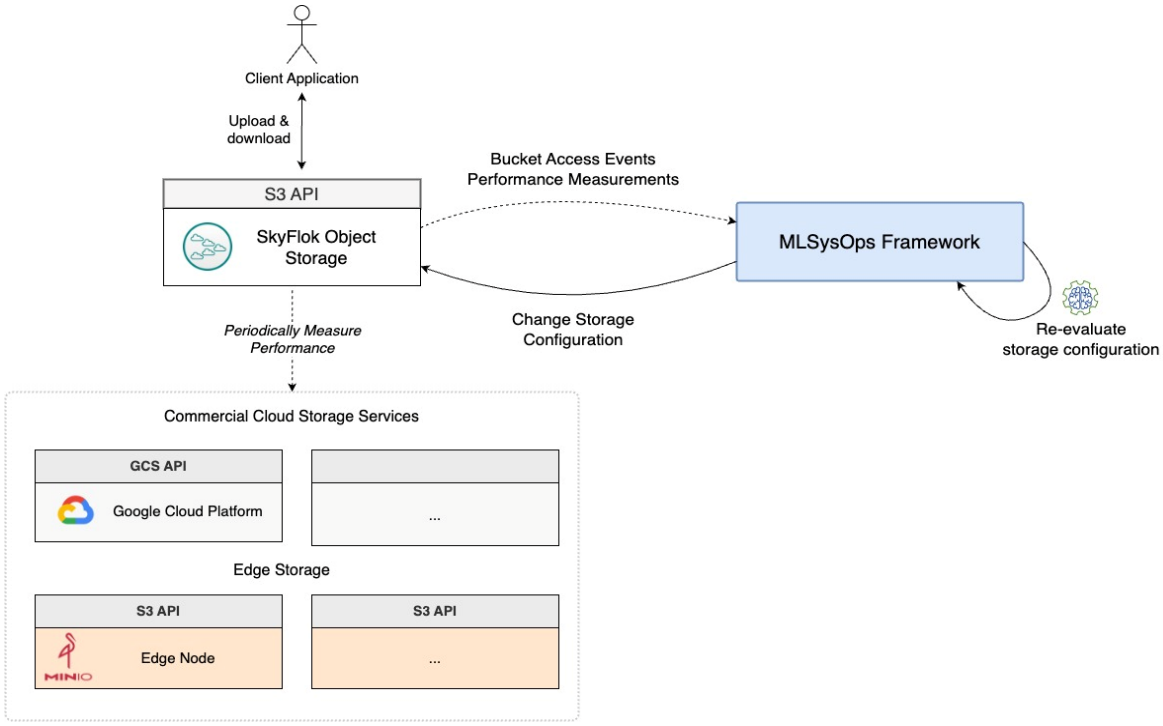


Figure 8.1: High-level overview of system components for ML-driven object storage

It is important to notice, that since the ML component is only able to change the storage representation of buckets, it can only affect the speed of just one of the download steps. Collecting data fragments can be accelerated if the new storage locations are either closer to the storage gateway or have inherently lower latency or higher throughput than the currently used ones. Changing the Storage Policy however does not affect the time of authorizing the incoming request, which is a constant-time operation that does not depend on the data

size or storage configuration, or decrypting the object contents, since encryption is always applied and cannot be turned off by the ML model.

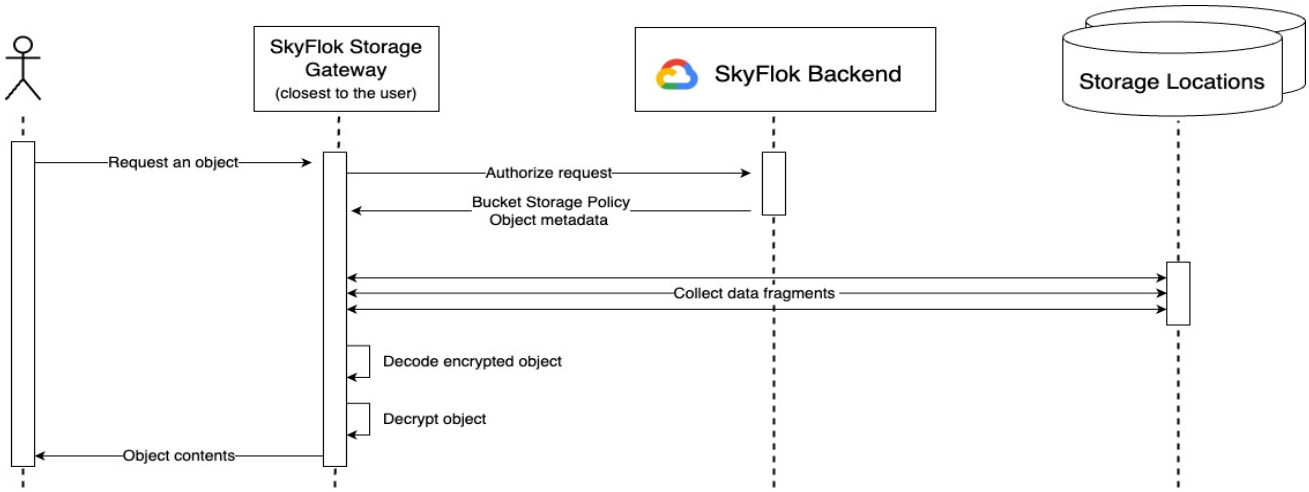


Figure 8.2: Object download process

Once a SkyFlok bucket is created, users of the MLSysOps framework can enable its ML-based management by including them in the MLSysOps application description and setting any of the following requirements:

- Maximum download latency
- Minimum download speed
- Restrictions on the storage locations (e.g., only GDPR-compliant locations)

For example, a bucket that stores small configuration files that need to be fetched quickly may define only a latency requirement of 150 ms. A different bucket used for large media storage (video, 3D models, AI models) could declare a required speed of 100 Mb/s and 300 ms latency. A bucket of logs or other low-priority data may not have any performance requirements and therefore is optimized for the cheapest storage regardless of traffic.

Note, that the current system design only considers performance requirements of object downloads (latency and speed) and location restrictions. In the future the same system can also consider requirements about upload speed and additional restrictions, such as mandatory high availability (e.g., always have at least one redundant fragment). Similarly, user-given requirements are currently considered static, but allowing changes during the lifetime of buckets may be supported later with minimal alterations to the system design.

When an object is downloaded by an end user, SkyFlok records relevant metrics and sends the following bundle of information to the ML component:

- Bucket name
- Timestamp
- End-to-end latency and speed of serving the request
- Geographical location of storage gateway that served the request (requests are automatically routed to the closest gateway instance using anycast DNS)
- Redundancy scheme of the object (number of total and redundant fragments)
- Storage locations where the fragments were transferred from
- Latency and speed of downloading each fragment
- Breakdown of time spent serving the request:
    - Authorization with SkyFlok Backend
    - Total wall clock time of downloading fragments
    - Decoding the encrypted object data
    - Decrypt the object data

Additionally, SkyFlok measures the latency and bandwidth of every available storage location once a day (including ones that are not currently used by any ML-managed buckets) and sends this information to the ML component.

When the ML component receives the notification of an object download event, it needs to decide whether a Storage Policy change is necessary and if so, what should be the new configuration. A change is necessary if either of the following criteria are met:

1. The bucket performance (either latency or bandwidth) is worse than the user requirements
2. The performance is acceptable, but a cheaper storage configuration would also satisfy the requirements

Evaluating Criteria 1 can be done directly by comparing the measured performance with the user requirements of the bucket. Note that even if the performance requirements do not warrant a storage policy change, the ML model must consider alternative configurations every time due to Criteria 2. Either way, it has to look at the characteristics of recent bucket traffic and try to find the cheapest Storage Policy that is close to the origin locations of significant portions of the experienced demand and use storage locations and a redundancy scheme that is expected to meet the performance requirements and location restrictions.

The access event notification and daily performance measurements contain all relevant data for the ML model to decide what to change in the current storage policy, and what the new values should be.

- Measured latency and speed are provided to assess Criteria 1
- The gateway location can be used to detect whether the origin of demand is shifting considerably
- A list of storage locations and the latency and speed of fragment/replica transfers are included so the model can detect if a storage location is experiencing a slowdown compared to its usual speeds
- The daily latency and bandwidth measurements of all storage locations provide information to the model about the expected performance of all alternative locations
- The time breakdown allows the model to infer the root cause of a slowdown and assess whether it can be fixed by moving fragments to different locations or changing the redundancy level

8.1.1 Additional Considerations

Besides the functional requirements discussed already, the ML component needs to consider several additional circumstances when generating a new storage policy and triggering a change.

Migrations are not free

Objective 5 states that the model should be aware of the cost of its actions and aim to minimize them. This objective is required because changing the storage configuration of a bucket has a one-time cost that may be significant compared to the savings it achieves. When the change involves moving fragments between locations or changing the redundancy level in a way that requires re-encoding data, the SkyFlok platform has to download and upload fragments to realize the change. For example, when increasing the redundancy from 3+1 to 3+2, the SkyFlok storage gateway must download 3 fragments of every object, generate the new redundant fragment and upload them to the new location. Depending on where the source fragments are stored, this operation costs \$0.01-\$0.12 per gigabyte of data, plus \$0.01 per gigabyte for uploading the new fragments from the Gateway to their destinations. While this seems like a negligible cost, migrating large buckets frequently might add up to a comparable amount to the monthly operating cost of the bucket itself.

To help the ML model evaluate potential new storage policies, the SkyFlok platform provides an API that calculates the cost of migrating an existing bucket from its current policy to a new theoretical one.

Significant changes only

The ML model should only trigger data migration when the change in either location or magnitude is sufficiently large. If the bucket is experiencing a steady demand with a few exceptional requests from distant users, or just a small proportion of downloads are served under the performance requirements, a large bucket may not be

worth migrating. Therefore, the model should recognize when the traffic pattern is changing considerably and only initiate inferring an alternative Storage Policy in this case.

Scale to zero

If a bucket stops receiving traffic completely, SkyFlok does not send any more download event notifications to the ML component. However, the lasting lack of traffic is an important signal that should result in migrating the bucket to the cheapest possible storage configuration. The ML model should therefore evaluate the fitness of the current policy not only on new incoming access events but also triggered periodically, even if no events are reported.

ML Approach

We propose decomposing the ML task into three sub-tasks that work together:

1. Predicting bucket traffic based on past traffic
2. Predicting storage location performance
3. Finding the optimal storage policy

For the first two tasks; *bucket traffic prediction*; the goal of this sub-task is to predict the magnitude and origin of traffic that will likely be experienced by a bucket, based on the historical traffic that has been recorded previously and *storage location performance prediction* predicts the latency and bandwidth of downloading data fragments from every storage location that is available in the object storage system. We propose the following strategies to tackle these tasks.

8.1.2 Download speed prediction

This section compares multiple regression models to predict file fragment download latency based on various features across different service providers in the US and EU. The goal is to evaluate different strategies for selecting backends in an erasure coding scheme described by parameters (n, k) where n is the total number of coded fragments and k is the minimum number of fragments required to reconstruct the original data. For this work, $n = 6$, and $k = 4$. We to achieve this, we compare the following strategies:

1. Selecting the nearest k backends.
2. Randomly choosing k backends.
3. Employing machine learning to predict and select the optimal set of k backends.

This comparison aims to identify the most efficient and reliable backend selection strategy for data reconstruction and download. This was modelled as a regression problem $y = f(X) + \in$, where X, included features such time of the day, distance from the backend to requesting gateway, and size of the file fragment, and y, represents download latency.

Dataset Description and Preprocessing

SkyFlok sends the following two event types in JSON format to the ML component. This is the State of the SkyFlok (environment).

```
interface ObjectAccess {
  bucket: string
  latency_ms: number
  speed_mbps: number
  client_lat: number
  client_lng: number
```

```
    timestamp: string
    reliability: {
        scheme: 'rlnc' | 'replication'
        replicas?: number
        rlnc_redundant_packets?: number
    }
    fragment_downloads: {
        storage_location_id: number
        data_size_bytes: number
        latency_ms: number
        speed_mbps: number
    }
    time_breakdown: {
        total_time_ms: number
        skyflok_auth_ms: number
        fragment_transfers_ms: number
        rlnc_decode: number
        aes_decrypt: number
    }
```

```
interface StorageLocationMeasurement {
    storage_location_id: number
    data_size_bytes: number
    latency_ms: number
    speed_mbps: number
}
```

The training and testing data were collect between October 2024 and January 2025. The training data included over 59 backends distributed across 9 service providers in 22 cities in US and Europe. However, for testing, we only considered 6 backends from US and EU. The Figure 8.3 below shows the distribution of these backends.

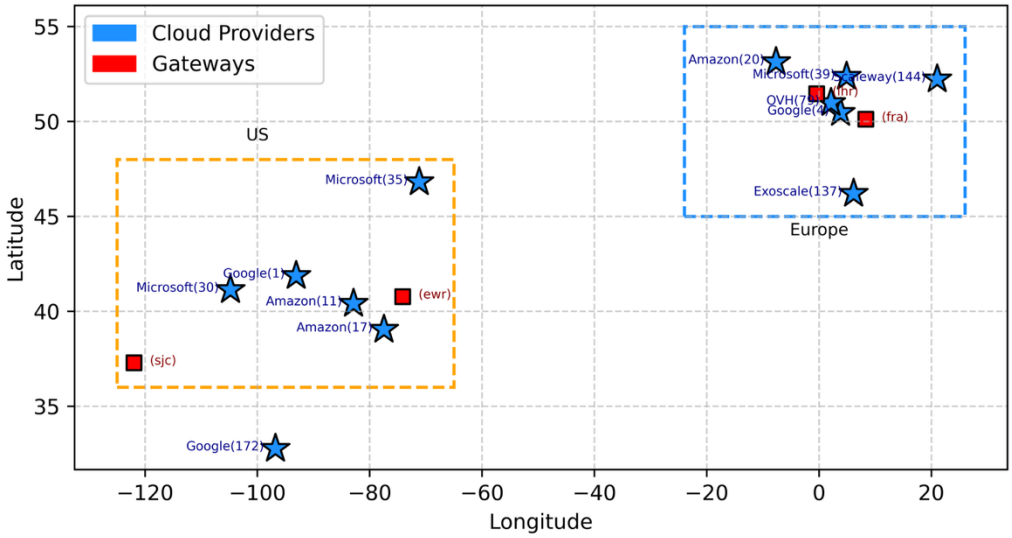


Figure 8.3: Distribution of backend and gateways used for testing ML models

Regression Models

To predict download latency, we built a regression model for each backend. We compared the following supervised learning regression approaches:

- Linear Regression: Linear regression models assume a linear relationship between input variables and the target variable. While this is one the simpler machine learning models, is still a useful and widely used statistical learning method[33] because it is computationally efficient.
- Random Forest Regression: In this, models constituent of the ensemble tree-structured predictors which are constructed using an injection of randomness[34]. During training, the method constructs multiple decision trees and outputs the mean prediction of these. As a result, this reduces overfitting and improving model stability and accuracy
- Gradient Boosting Regression: Unlike the above approach, in gradient boosting a sequential ensemble model builds regression trees in a stage-wise manner. Each new tree corrects errors from the previous trees, improving overall predictive performance[35].
- Extreme Gradient Boosting Regression (XGBRegressor): An optimized variant of the above method. It includes enhanced regularization techniques, and ability to handle of missing values.

Deployment

The machine learning models were. Deployed via the MLConnector described in the following section. Model training and deployment is completely decoupled from the file download gateways. This ensures scalability and ensures the gateways are not overloaded. The Figure 8.4 below summaries our machine learning pipeline.

The gateway can query the model in real-time for inference. This service also provides for explainability and drift monitoring of the machine learning models.

[33] https://link.springer.com/chapter/10.1007/978-3-031-38747-0_3
[34] https://escholarship.org/uc/item/35x3v9t4
[35] https://www.jstor.org/stable/2699986?seq=1

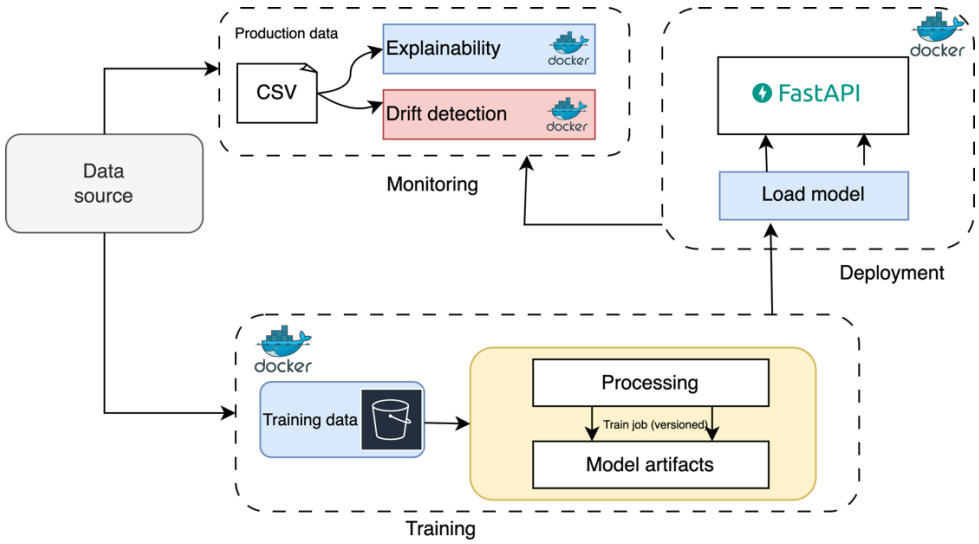


Figure 8.4: Machine learning pipeline

Evaluation

- Evaluation Metrics

  To evaluate the performance of the models, we used the R-squared, defined as

$$R^2 = 1 - \frac{\sum_{i=1}^{m}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{m}(y_i - \bar{y}_i)^2}$$

Where m is the number of samples, y_i is the actual value and $\hat{y}_i$ is the predicted value of the ith sample, and $\bar{y}_i$ denotes the mean value of actual value y_i.

- Hyperparameter Tuning

We used Grid Search for hyperparameter tuning. The method iteratively searches for a specific subset of hyperparameters by training the model on all possible combinations and then selecting the best-performing combination based on cross-validation of the scores. This approach ensures optimal parameter settings that enhance model accuracy and generalisation.

Result and discussion

We evaluated four regression models: linear regression, random forest, gradient boosting, and XGBRegressor on backends in the US and EU. We used R-squared (R^2) as the performance metric on training, validation, and testing and also on unseen data.

- Training, testing and validation

On the training, validation and testing sets, XGBRegressor achieved the highest R^2 of 0.9771, 0.8123 and 0.811 for US based backend, and 0.9782, 0.8217, and 0.8344 respectively. This shows a good balance between learning and generalization. Random forest also had closely comparable results. The Table 2.1 and Table 8.2 below summarise the training, validation and testing result for all the machine learning models.

Table 8.1: Comparing training, validation and testing R-squred for US based backends

	R-squared		
	Training	Validation	Testing
Linear regression	0.6733	0.6683	0.6689
Random forest	0.8248	0.7941	0.7995
Gradientboosting	0.8998	0.7987	0.8036
Xgbregressor	0.9771	0.8123	0.811

Table 8.2: Comparing training, validation and testing R-squared for EU based backends

	R-squared		
	Training	Validation	Testing
Linear regression	0.5186	0.5073	0.5077
Random forest	0.8369	0.8128	0.8106
Gradient boosting	0.9085	0.8168	0.8203
Xgbregressor	0.9782	0.8217	0.8344

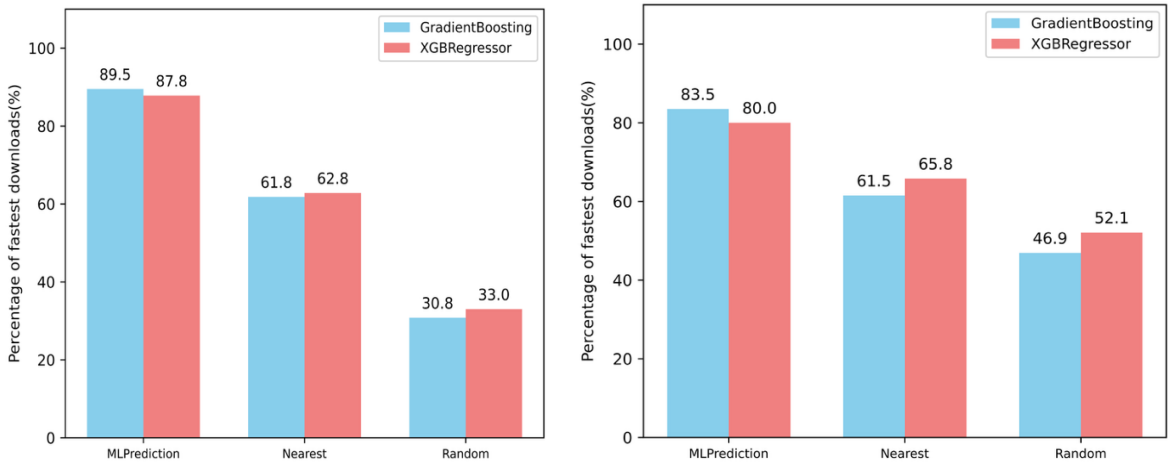


Figure 8.5: (a) Comparing model performance on unseen data for US backend (b) Comparing model performance on unseen data for EU backends

- Unseen data

Comparing he results of the unseen data, the ML-based backend selection performed better than the other two alternative, nearest and random. The result show, in the US based backends, the machine learning approach was able to predict the fastest backends 89.5% and 87.8% for gradient boosting and xboost respectively. In the EU based backends, it was 83.5% and 80% respectively. However, while xboost had better training, validation and

testing scores, gradient boosting achieved better performance on unseen data as show in the plots above. This means that gradient boosting model was able to generalise better than xboost.

8.1.3 Bucket traffic prediction

For traffic prediction, we are exploring three common forecasting methods: ARIMA, Prophet, and regression-based models. Each method offers a trade-off between interpretability, ease of use, and forecasting accuracy.

- ARIMA (AutoRegressive Integrated Moving Average) is a classical statistical model commonly used for stationary time series due its ability to capture temporal dependencies through autoregressive terms, differencing to remove trends, and moving averages of past errors[36]. It performs well on short-term forecasting tasks, but it requires manual tuning and it's less effective when the data exhibits strong seasonality.
- Prophet is an additive model designed for business time series data. It automatically decomposes time series into trend, seasonality (daily, weekly, yearly), and can capture holiday effects[37]. It performs well even if the data contains missing data or outliers and requires minimal tuning and is particularly effective for data with complex but interpretable seasonal patterns.
- Regression-based models leverage engineered features (e.g., hour of day, day of week, lag variables) to forecast future values using linear or non-linear regression techniques[38]. These models are flexible, but they rely heavily on the quality of feature engineering and may struggle to capture autocorrelation if not explicitly modelled.

8.1.4 Optimal Storage Policy Generation

When the storage management component examines an enrolled bucket for a potential storage policy change, it uses the output of the bucket traffic predictor and storage performance predictor sub-components to determine the expected end-user performance of potential new policies and to predict the operational savings.

We are planning to solve this problem with two competing approaches:

- A heuristic algorithm that encodes simple rules of how storage policy attributes should be changed in different situations
- Algorithms that use machine learning techniques

For this sub-task, we plan to explore Deep Reinforcement Learning (RL). RL involves an agent interacting with its environment and learning an optimal policy through trial and error. In recent years, deep learning, specifically deep neural networks, has become prominent in reinforcement learning applications, including games, robotics, and natural language processing. The agent interacts with the environment and takes certain actions to get maximum rewards. Figure 2.3 summarises the process.

[36] Shumway, R.H., Stoffer, D.S., Shumway, R.H. and Stoffer, D.S., 2017. ARIMA models. *Time series analysis and its applications: with R examples*, pp.75-163.
[37] Sivaramakrishnan, S., Fernandez, T.F., Babukarthik, R.G. and Premalatha, S., 2022. Forecasting time series data using arima and facebook prophet models. In Big data management in Sensing (pp. 47-59). River Publishers.
[38] Jadon, A., Patil, A. and Jadon, S., 2024, January. A comprehensive survey of regression-based loss functions for time series forecasting. In *International Conference on Data Management, Analytics & Innovation* (pp. 117-147). Singapore: Springer Nature Singapore.

9 Computation resource configuration using ML

MLSysOps manages a slice of infrastructure that can include a wide variety of computation and network resources. Resources may vary in kind and availability and are shared between different application components. Every application is deployed inside containers, and the default orchestrator schedulers focus on the fair scheduling of every process and the general availability of the resources. MLSysOps framework offers application developers the capability to provide specific performance targets for their applications deployed in the infrastructure. At the same time, it offers the infrastructure administrators the same option to set specific system operating targets (e.g. green energy usage, power profiles). It is responsible for deciding and applying suitable configurations that balance the application and system targets. Those targets might also change dynamically at runtime.

Problem description

The management of each application in the MLSysOps framework consists of two distinct phases: *i)* initial deployment and *ii)* adaptation. The initial deployment phase includes all the necessary steps from the point of the application submission until the successful deployment of the application components. Figure 9.1 shows a sequence of the interactions between the various system components with the MLConnector API for the initial deployment of an application. It focuses on the cluster- and node-level interactions, assuming a single cluster deployment for the continuum agent. As is shown in the figure, different independent decisions must be made at different levels. The agents at each level need to decide on different configuration options for the initial deployment, and they might need to use more than one ML model to decide on different aspects.

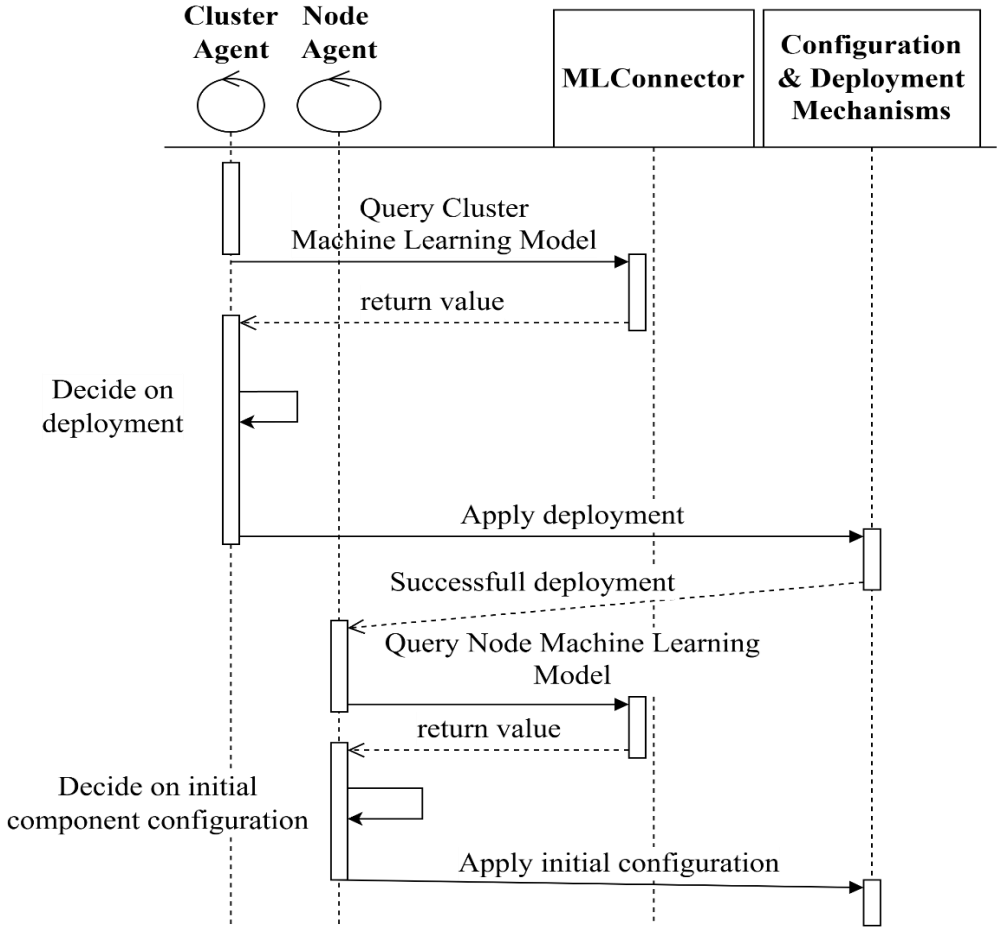


Figure 9.1 Initial deployment sequence. The ML models determine (i) the component placement and then, (ii) the initial node configuration.

The cluster agent needs to decide on which node each component should be placed, and for each component interaction which network connection should be used. It uses the MLConnector API to query the available ML models to get suggestions that will enhance its decision. After the decisions have been made, it proceeds with the component deployment, and then the node-level agent will decide on its local configuration parameters, which will be applied as soon as the component is successfully deployed on the node. At that point, the node-level agent also queries the available ML models through the MLConnector API.

A similar procedure takes place in the adaptation cycle, as shown in Figure 9.2. The agent in each level periodically runs an analysis and plan loop, analysing the current state of the system in conjunction with the application components' performance, to determine if a new plan is needed. The adaptation cycle in each agent runs independently from each other. A new plan might be needed if either the system or application targets are not met or if there is room for improvement. The ML models that suggest a new configuration should also provide a performance prediction for the new configuration, to allow the agents to make an informed decision.

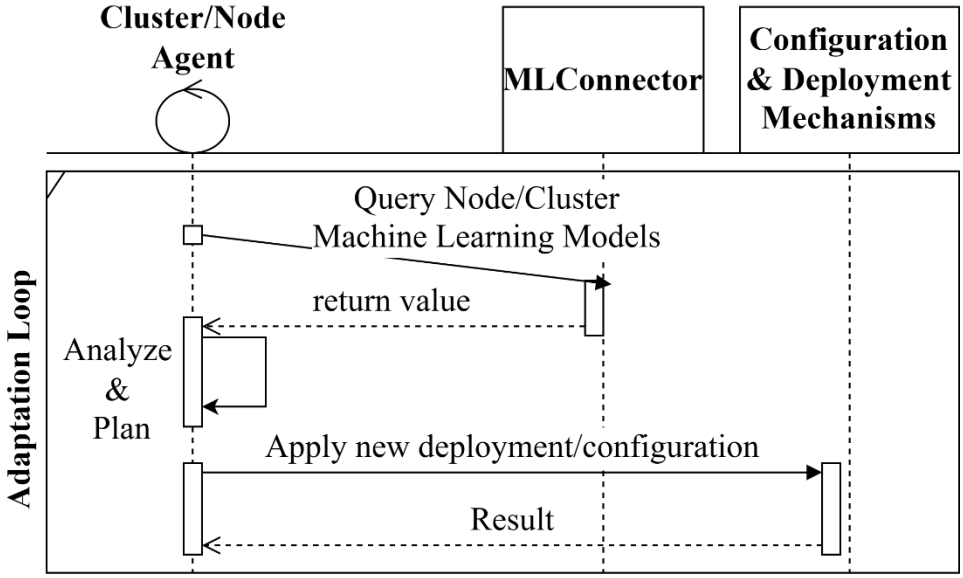


Figure 9.2 Adaptation Loop in both cluster and node level

Proposed approach

As it was described in the previous paragraph, different problems require different ML models. However, all these problems can be handled by RL agents that can be tailored to use both system- and application-level information as its state space and use different reward functions to train different underlying machine learning models for the various decisions on the configurations that must be made.

In the following subsections, we describe an approach for the state and action spaces for an RL agent in node level. The main assumption is that there is only one application component in the application description that can be dynamically adapted. The results that are presented operate on the initial deployment scenario.

The state space of the computation resources RL model in a single computation node includes:

- CPU Utilization for each core.
- Memory utilization for each node.
- Total power consumption.
- Application metrics from the component that has been deployed on the node.
- Features extracted from the application description and the components that have been placed on the node.
- Features extracted from the system description that have been submitted into the framework.

The **action space** is designed to contain the configuration options available for the cluster level. It has three different decisions:

- The type of the computation resource that will be allocated for the component (CPU, GPU, TPU, FPGA, etc.).
- The number of cores of the computation resource that will be allocated:
  - CPU: How many and which cores will be used?
  - GPU: In machines with multiple GPUs, decide on the number of GPU devices.
  - FPGA: How many and which fabric regions will be used?
- The operating frequency of the core in whatever granularity is available in the node.

For the **reward function** a cost model is used. The specific costs at the node level are:

- Computation cost is calculated from the power consumption of each computation resource.
- Application penalty is calculated as the difference between the current application metric and the application target.
- System penalty is calculated as the difference between the current system goal and the target.

The actual reward value is a function of the above cost to minimize the cost.

Data collection & experiment setup

The training of the proposed RL agent is performed using data produced by a realistic lab-based environment, and more precisely by the UTH research testbed. The telemetry system that is used in the MLSysOps framework will be used to collect and store the system and application metrics. On this testbed, we will run different scenarios with different aspects that we can focus on, in terms of node heterogeneity and application variation. The experiments will be split into different categories, such as cluster-level only, node-level only, and a combination of both. In all cases of experiments, different deployments and configurations are deployed and the telemetry system will record the results. These results will eventually be used by machine learning models for training and validation.

There are two modes of training that we utilize: i) online and ii) offline training. In the online training procedure, the RL agent decides on an action from the action space, applies the configuration in the real testbed, and then waits for the results. Based on the results, it calculates the reward and proceeds with the next step of training. However, this mode is extremely time-consuming and is mostly used on the deployed RL agent. On the other hand, in offline training, recorded data are used that were produced by executing every possible action and getting the result telemetry data from the system. The RL agent uses the recorded data in each step of the training process, significantly reducing the training time. This mode is mostly used for first-time training.

Regarding heterogeneity, one special type of (mobile) node that the UTH research testbed contains is the drone. To add more flexibility, the AeroLoop simulator (see D4.3) will help us conduct experiments at scale, simulating virtual UAVs and virtual Edge Nodes, without the overhead of performing field tests, which is a weather dependent as well as time-consuming procedure. Initially, we must ensure the proper functionality of the proposed mechanisms and intelligence via exhaustive testing before making the real experiments.

In addition, the simulator will assist in both offline and online training modes. For the former, we will have the opportunity to generate data based on the actions of the RL agent on a realistic system setup. Regarding online training, we will examine the adaptation capabilities of the RL agent. The latter will be (periodically) re-trained at runtime, while the application is running, depending on how well it performs and the dynamicity of its environment.

Computation resource allocation using RL

To validate our design, we have developed a proof-of-concept RL agent, that was trained to determine an optimal CPU core allocation for an application component, based on the application target.

An application container was prepared that runs ResNet18, a pre-trained image classification convolutional neural network on a batch of input images. The application metric of interest is the inference latency for each

individual image. The container runs for a specific interval (60 seconds) on a datacenter server machine of the UTH research testbed, including two AMD EPYC processors with 64 cores and 128 threads (hyperthreaded is enabled). During the execution of the container, real-time telemetry data were collected and stored in the telemetry database. The telemetry data that were collected and the telemetry collection parameters are shown in Table 9.1

Table 9.1 Telemetry metrics recorded

Metric	Values range [Units]	Collection Frequency
Average CPU utilization for each logical core	0.00 – 1.00	10 seconds
Total Average CPU Power Consumption for both sockets	0.00 – 400.00 [Watts]	10 seconds
Application metric (Inference latency)	0.00 – 1000.00 [ms]	1 second

The experiments involved deploying the application container with setting the number of CPU logical cores that could be used by the application. The number of cores ranged between 1 and 100. In these experiments, we assume that no other application components are running on the machine, but other processes from other users might be running. A sample of the final recorded data is summarized in Table 9.2.

Table 9.2 Sample of the recorded telemetry data

Cores Allocated	Application Metric	Average Power	Core 1 Util.	Core 2 Util.	Core 3 Util.	Core 4 Util.	…	Core 128 Util.
1	115,84	109,550	0,337	0,541	0,030	0,030	…	0,035
2	49,82	111,314	0,810	0,818	0,011	0,006	…	0,004
3	36,06	116,654	0,802	0,805	0,805	0,014	…	0,010
4	25,76	123,373	0,788	0,793	0,793	0,793	…	0,009
5	21,26	128,725	0,775	0,785	0,782	0,7875	…	0,009
6	19,02	130,744	0,761	0,773	0,775	0,7725	…	0,012
7	15,76	135,183	0,748	0,761	0,761	0,762	…	0,015
8	15,98	138,075	0,736	0,752	0,750	0,753	…	0,008
9	15,98	142,694	0,673	0,740	0,739	0,740	…	0,010
10	17,43	149,616	0,681	0,729	0,729	0,729	…	0,021
11	13,29	158,297	0,756	0,863	0,863	0,863	…	0,004
12	10,97	162,819	0,813	0,851	0,852	0,851	…	0,007
…	…	…	…	…	…	...	…	…
100	5,91	335,855	0,722	0,752	0,755	0,752	…	0,015

Figure 9.3 shows the results from the experiment runs, showcasing the need for an optimal core allocation. Assuming an application target of 7ms, the inference latency is initially improved by using more cores for the computation, increasing the power consumption at the same time, however, there is a point where the latency does not improve further, but the power consumption continues to increase.

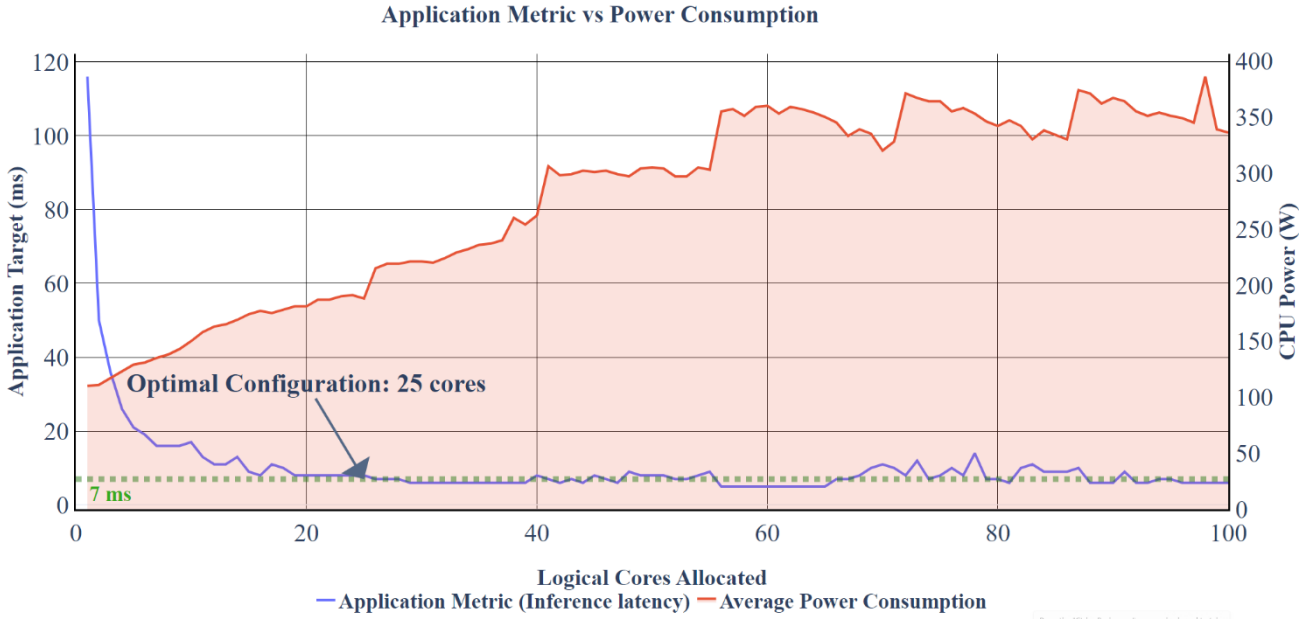


Figure 9.3 Telemetry metrics concerning the number of cores allocated. A handpicked optimal configuration is shown for the application target of 7ms.

The goal of the RL agent should be to successfully find a configuration that is optimal or at least near the optimal spot. To achieve this, we need to build a suitable reward function that will guide the RL agent to achieve the application target without spending more power than is needed. In Equation 9-1, we can see the reward function that we used in this proof-of-concept to train the RL agent. It was based on a prior work that explored a similar problem[39].

$$reward = \begin{cases} -100, & app\ metric\ >\ app\ target \\ -\sqrt{(app\_penalty\ +\ power\_cost)}, & app\ metric\ <=\ app\ target \end{cases}$$

Equation 9-1 Core allocation reward function

In Equation 9-1, the *app_penalty* term corresponds to the difference between the application metric and the application target, where the penalty is higher if the application performs better than it is required. On the other hand, the *power_cost* term corresponds to the power that was consumed to achieve this target, where the penalty is higher if the system consumes more power. The target of the RL agent is to gain a bigger reward, so it tries to minimize the penalties, achieving the desired goal.

[39] M. Han and W. Baek, "HERTI: A Reinforcement Learning-Augmented System for Efficient Real-Time Inference on Heterogeneous Embedded Systems," 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), Atlanta, GA, USA, 2021, pp. 90-102, doi: 10.1109/PACT52795.2021.00014.

The initial training results are shown in Table 9.3. It includes the results for 4 different application targets, on the same dataset. The agent was trained for 500000 timesteps. The agent managed to achieve some promising results on 2 of the 3 targets.

Table 9.3 Reinforcement Learning agent prediction results.

Application Latency Target (ms)	Mean Reward Value	Predicted Configuration (Cores to allocate)	Handpicked configuration (Cores to allocate)
5	-1616	57	61, 65
7	- 1188	25	25
10	-1070	35	15
15	-842	10	11

In the context of MLSysOps, this node level RL model will be used once the application components (e.g. the ResNet18 in this case) has been mapped to the specific node by the cluster-level RL model to adjust the configuration of the node for application execution. In this instance, the node level RL model determines an optimal CPU core allocation to optimize power dissipation and, at the same time, obey application latency constraints (first column of Table 9.3).

10 ML policies for cloud resource management

Resource utilization in cloud data centers remains inefficient, affecting both cost and sustainability. Modern platforms are increasingly aiming to minimize operational costs while meeting service level objectives (SLOs).

VM scheduling and resource management in modern cloud data centers remain challenging due to the dynamic behavior and diverse life spans of virtual machines (VMs). These factors complicate allocation and migration decisions, often resulting in underutilized hosts, or in SLO violations. Existing methodologies primarily focus on average CPU utilization, overlooking critical indicators such as peak utilization, burst patterns, and maximum CPU usage. These overlooked metrics are essential for overcommitment strategies and for avoiding SLO violations. LSTM models[40] are effective in capturing average utilization trends; however, they fail to predict sudden demand peaks.

Furthermore, VM lifetime provides useful information to guide proactive allocation decisions. Long-lived VMs may justify the cost of migration to balance load, while VMs with short remaining lifetimes do not offer the opportunity to amortize the migration cost. Current frameworks either do not incorporate lifetime predictions or assume predefined lifetime distributions.

Contributions. To address these challenges, we introduce *PeakLife*, an ML-based framework integrated with the MLSysOps software stack that optimizes VM migrations using an SLO-aware deep surrogate model. By jointly forecasting CPU utilization (both average and peak) and VM lifetimes, PeakLife enables proactive, system-wide decision-making. We evaluate PeakLife on the Azure dataset[41] using trace driven simulation. PeakLife performs remarkably close to an oracle strategy and when compared to baseline methods, it reduces unnecessary migrations by 41.33% and lower SLO violations by 34.98%.

PeakLife framework

10.1.1 Overview

We consider a typical cloud environment with multiple VMs running on a set of hosts. Following commercially available practices and the MLSysOps architecture, a centralized Resource Controller within the cluster-level agent manages (CPU) resource allocation, VM placement and consolidation, and host power-up/down. As shown in Figure 10.1, PeakLife includes two modules: the Predictor and the Cluster-level Coordinator. The Predictor, deployed locally on each host, works as a discrete-time controller that uses recent metrics to predict VM lifetime and future CPU utilization, to drive VM management decisions. The Cluster-level Coordinator receives the output of each Predictor and evaluates candidate VMs for migration to perform system-wide optimization.

[40] J. Shi, K. Fu, Q. Chen, C. Yang, P. Huang, M. Zhou, J. Zhao, C. Chen, and M. Guo, "Characterizing and orchestrating VN reservation in geodistributed clouds to improve the resource efficiency," in Proceedings of the 13th Symposium on Cloud Computing, 2022, pp. 94–109.
[41] E. Cortez, A. Bonde, A. Muzio, M. Russinovich, M. Fontoura, and R. Bianchini, "Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms," in Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 153–167.

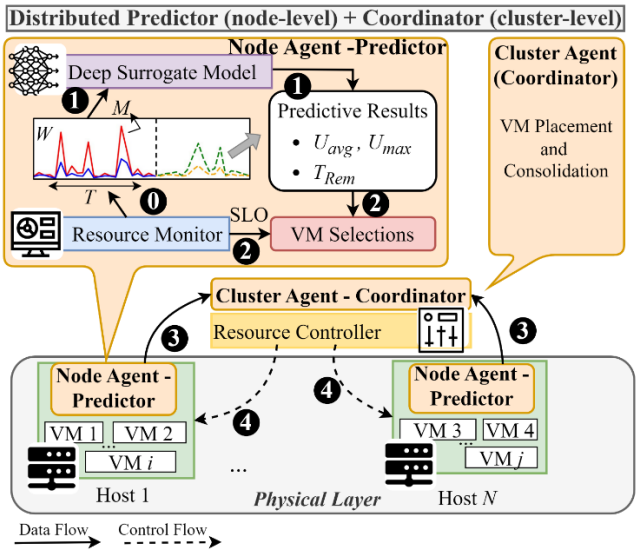


Figure 10.1: Overview of PeakLife. The Predictor runs locally on each node, while the Coordinator operates at the cluster-level

The Resource Monitor **(0)** monitors the state of the local VMs on each host and provides data to the Predictor. More specifically, the Resource Monitor collects the average and maximum CPU utilization during the previous window W of Δt time units for each VM, together with its current running time T, and the maximum utilization M within T. The data are fed into the Deep Surrogate Model **(1)**, which outputs future average (U_{avg}) and maximum (U_{max}) CPU utilization for multiple (k) time steps {t + 1, t + 2, ..., t + k}, as well as the remaining lifetime (T_{Rem}) for each VM. These predictions are used to produce a locally optimized set of Candidate VMs for migration **(2)** by excluding those likely to cause overutilization or that can be migrated cost-effectively.

The Coordinator **(3)** then performs a selection of the recipient hosts and global consolidation for energy efficiency based on the aggregated predictions. The decisions of the Cluster-Level Coordinator are implemented by the Resource Controller **(4)** . We discuss the architecture of the deep surrogate model in Section 10.1.2. The functionalities of Predictor and Cluster-level Coordinator are detailed in Section 10.1.3, together with the VM management algorithm.

10.1.2 Encoder-Decoder-based Deep Surrogate Model

The core of PeakLife is a deep surrogate model designed for joint prediction of CPU utilization and VM lifetimes. It follows a sequence-to-sequence (seq2seq) architecture. Figure 10.2 shows the architecture of the deep surrogate model.

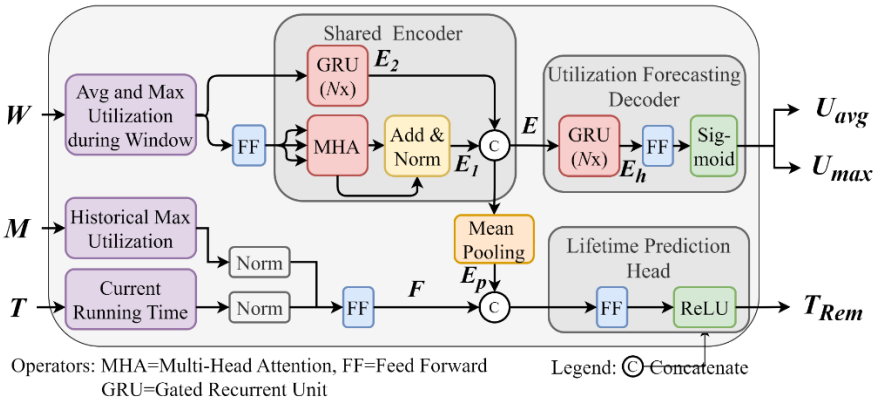


Figure 10.2: PeakLife Deep Surrogate Model

1) Shared Attention-GRU Encoder. The encoder extracts temporal and contextual features from the input sequence. The input W is first projected via a Feed Forward layer (FF) to obtain W_{seq}, which is then passed through a Multi-head attention (MHA) block [20] with n-heads, followed by the scaled-dot product attention operation to model long-range dependencies. The attention output is calculated as

$$W_{Att} = \text{MultiHeadAtt}\ (W_{seq}, W_{seq}, W_{seq})$$

W_{Att} is stabilized via residual connections and layer normalization to produce E_1. In parallel, W is processed through a stacked Gated Recurrent Unit (GRU) to capture local temporal dynamics and output E_2. E_1 and E_2 are concatenated to form the encoded representation $E = [E_1; E_2]$, which captures both local and global temporal dynamics of the input sequence W.

2) CPU Utilization Forecasting Decoder. To predict future utilization in a predefined output length, the decoder uses the last timesteps of E from the shared encoder, denoted E_L and pass through GRU layers to generate hidden states E_h. E_h is then passed through a feed-forward layer with the Sigmoid activation function to output average and maximum utilization predictions, U_{avg} and U_{max} in the range [0, 1]:

$$[U_{avg}, U_{max}] = \text{Sigmoid}\ (\text{FeedForward}(E_h))$$

3) Lifetime Prediction Head. To estimate T_{rem}, the lifetime prediction head processes the encoder representation E, and concatenates it with auxiliary features, i.e., maximum utilization M to t and current running time T. M and T are normalized to the same scale to ensure stability during training, and then passed through FF to obtain the high-dimension representation F. We use mean pooling to provide a global summary of the sequence E_P, aggregating the sequence-level features into a single vector, which can then be concatenated with auxiliary features F and passed to the lifetime prediction head. Finally, the combined representation is passed through an FF layer with ReLU activation to produce T_{Rem}.

Offline Model Training. For training, we use Ray to optimize the model hyperparameters and ensure training stability. The model uses two-layer GRUs (hidden size = 32), 8 attention heads, and a dropout of 0.1. It is trained for 100 epochs using the Adam optimizer with an initial learning rate of 0.001 and batch size of 8. For loss definition, we use Mean Squared Error (MSE) for utilization prediction and a combination loss of MSE and Mean Absolute Percentage Error (MAPE) for lifetime prediction. The total loss is defined as the summation of the above two and is used as the training loss function. Training loss has stably decreasing trends. Early stopping is used when the validation loss drops below the expected threshold L_T. The model is implemented in PyTorch. We construct training samples using a sliding-window approach. Utilization data are normalized to [0, 1] and lifetimes are log-normalized to balance scale differences.

Online Model Inference. The trained model predicts U_{avg} and U_{max} for several future time steps, and the remaining lifetime T_{Rem} for each VM. The outputs T_{Rem} are inverse-transformed to real values. The predicted results are then used to drive VM selections (see Figure 10.1).

10.1.3 Optimizing VM management decisions

Objective. To mitigate host overload and reduce unnecessary migrations, we model VM scheduling as an SLO-aware optimization problem. Specifically, the goal is to minimize SLO violation costs (C_{SLO}). An SLO violation occurs when a VM does not receive sufficient CPU resources, either due to host overload or temporary resource unavailability during migration. C_{SLO} for VM v can be modeled as

$$C^v_{SLO} \propto U^v_t \cdot Mig_v, \quad (3)$$

where Mig_v is the migration overhead factor for VM v. We next leverage multi-step predictions of utilization and incorporate proactive migration strategies. State-of-the-art VM consolidation strategies typically follow a four-step process[42]: (i) host overload detection, (ii) VM selection for migration, (iii) VM placement, and (iv) host consolidation (i.e., to power off underutilized hosts). PeakLife improves step (ii) by proactively selecting

[42]A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," Future generation computer systems, vol. 28, no. 5, pp. 755–768, 2012.

migration candidate VMs in a smarter way based on predicted CPU utilization and VM lifetimes. The general idea is summarized in the algorithm of Table 10.1. The inputs are defined as a set of active VMs V on each host and, for each VM, the respective predicted average or maximum utilization U^v, the current running time T^v_{curr}, predicted remaining lifetime T^v_{rem}, a predefined host utilization threshold θ_{util}, a VM lifetime filtering threshold θ_{ratio}, and a forecast window W.

Table 10.1: VM management algorithm

	Input: V, U^v, T^v_{rem}, θ_{util}, θ_{ratio}, W
	Output: VMs selected for migration M
1	**Initialize:** $M \leftarrow \emptyset$;
2	**Estimate Host Utilization** $U_{host,t}$ **Using** U^v **Across Future Steps.**
3	**Compute Weighted Utilization Metric:** $U_{weighted} = \frac{\sum_{t \in W} w_t * U_{host,t}}{\sum_{t \in W} w_t}$
4	Calculate θ^{adap}_{util} based on θ_{util} (Eq. 4)
5	**if** $U_{weighted} > \theta^{adap}_{util}$ **then**
6	**for** VM $v \in V$ **do**
7	**if** $T^v_{rem} < \theta_{ratio}$ **then**
8	**continue**;
9	**end**
10	**Compute cost**: $Cost^v_t = U^v_t \cdot Mig_v \cdot 1/T^v_{rem}$
11	**end**
12	**while** $U_{weighted} > \theta^{adap}_{util}$ **do**
13	**Select VM** v^* **for Migration**: $v^* \leftarrow \arg\min_{v \in V \setminus M} Cost^v_t$
14	$M \leftarrow M \cup \{v^*\}$;
15	**Update Host Utilization** $U_{host,t}$ with v^* **Removed Across** W.
16	**Recompute Weighted Utilization** $U_{weighted}$
17	**end**
18	**end**
19	**return** M

The algorithm first initializes the candidate set (line 1). Then, it calculates the estimated host utilization during the forecast window W based on the predicted utilizations (line 2). The algorithm calculates host utilization during a window of future scheduling periods as a weighted average. The weights w_t prioritize predictions for scheduling periods closer to the present time (e.g., $w_t = 1/(t + 1)$), highlighting immediate risks of overutilization while still considering future trends. Line 4 dynamically adjusts the utilization threshold θ_{util} based on the adaptive tuning mechanism described by

$$Gap_{Norm} = (U_{max} - U_{avg}) / U_{max}, \quad (4)$$

$$\theta^{adap}_{util} = \theta_{util} \cdot (1 - \alpha \cdot Gap_{Norm})$$

The adaptive tuning mechanism dynamically adjusts the utilization threshold of host nodes, optimizing resource efficiency while avoiding SLO violations while considering workload variability. Specifically, setting a higher utilization threshold would improve energy efficiency by packing more VMs on the same host. However, it may increase the risk of overutilization due to aggressive allocations and thus increase SLO violations. The threshold is tuned more aggressively when the workload is bursty (i.e., there is a significant difference between U_{max} and

U_{avg}, indicating a high risk of overloading if we do the allocation based on U_{avg}-only), or when operating under higher values of α.

Starting from line 5, candidate VMs for migration are identified when the host is predicted to be overutilized. We adopt a minimum cost-based VM selection strategy that incorporates predicted remaining lifetime, ensuring that migration overhead can be amortized over time. Specifically, we exclude VMs predicted to be short-lived (lines 7-9). For the remaining VMs, a cost score is computed (line 10) using Eq. 3, multiplied by the inverse of log-normed (so that extreme large values do not dominate the priority) remaining lifetime estimation ($1/T^{v}_{rem}$), to get VMs with shorter expected remaining life deprioritized.

Finally, a while-loop selects the VM with the lowest cost (lines 12-17) until the host utilization falls below θ_{util}. After each selection, the host utilization is updated to account for the removed VM. The candidate VMs set is returned as M (line 19). The complexity of the algorithm is O(nlogn), where n is the total number of VMs. Typically, $n \gg m$, where m is the total number of hosts.

The candidate VMs set M is then submitted to the Cluster-level Coordinator for host placement (see Section 10.1.1). The Cluster-level Coordinator employs the Best Fit Decreasing (BFD) algorithm to allocate VMs to destination hosts. The complexity of BFD is O(nlogn), where n is the total number of VMs to be migrated. As the last step, we apply the host consolidation method from[42], which powers off underutilized hosts by accommodating VMs that can be migrated away without overloading other hosts. The complexity of this last step is O(mlogm), where m is the total number of hosts.

Evaluation

10.1.4 Experimental Setup

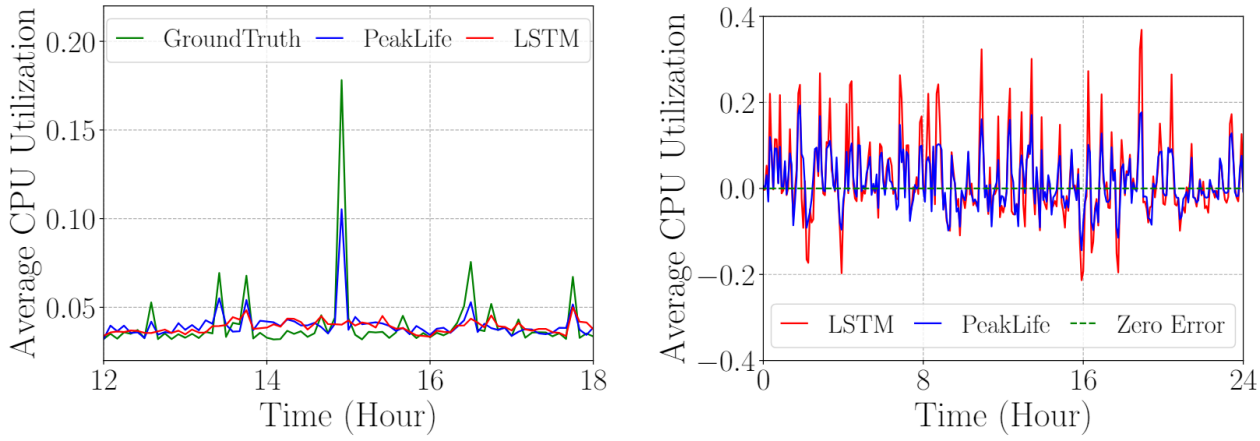
Model training and testing. The deep surrogate model of PeakLife is trained and tested using the Azure dataset. We use a one-day window of historical data to predict utilization for the next 30 minutes (6 steps, each 5 minutes). The model is refined weekly using data from the previous week.

Metrics for Evaluation. We evaluate CPU utilization prediction using Mean Absolute Percentage Error (MAPE) for accuracy, and Pearson Correlation (PCorr) for pattern similarity. For lifetime prediction, we use MAPE and analyze the error distribution of expected lifetimes on the test sets.

10.1.5 Utilization Prediction Results

We compare the prediction performance of PeakLife's surrogate model with two widely used prediction models: the analytical ARIMA model[43] and the LSTM-based model[44,45]. In all cases we have tuned the architecture (e.g., layer depth, hidden units) for optimal performance. All models are evaluated against a hold-out test set referred to as Ground Truth (GT), which is not used during training.

[43] R. N. Calheiros, E. Masoumi, R. Ranjan, and R. Buyya, "Workload prediction using arima model and its impact on cloud applications' QoS," IEEE transactions on cloud computing, vol. 3, no. 4, pp. 449–458, 2014.
[44] J. Liu, X. Tan, and Y. Wang, "Cssap: Software aging prediction for cloud services based on arima-lstm hybrid model," in 2019 IEEE International Conference on Web Services (ICWS). IEEE, 2019, pp. 283–290.
[45] A. I. Maiyza, N. O. Korany, K. Banawan, H. A. Hassan, and W. M. Sheta, "VtGAN: hybrid generative adversarial networks for cloud workload prediction," Journal of Cloud Computing, vol. 12, no. 1, p. 97, 2023.



(a) Zoomed view of peak prediction accuracy of a one-day segment

(b) Residual plot for volatile utilization prediction

Figure 10.3: Prediction results of PeakLife and LSTM-based model tested using typical average CPU utilization traces

Average Utilization. PeakLife prediction has a Pearson correlation of 0.93 and a 7.72% MAPE with Ground Truth, i.e., it achieves high accuracy in utilization pattern prediction. We present two representative VM cases with distinct characteristics in Figure 10.3. The first, shown in Figure 10.3a, exhibits low-variance utilization (mean value μ=0.041 and standard deviation σ=0.023) with periodic peaks. Both PeakLife and the tuned LSTM model track the general trend, but PeakLife captures peaks more accurately, although it tends to overestimate their amplitude. The results provided by ARIMA are excluded because of its poor performance. A summary evaluation of ARIMA is provided in the sensitivity discussion later in this section.

Figure 10.3b represents a more challenging case of average utilization prediction (μ=0.744, σ=0.115). The plot shows the residual error normalized against ground truth using one-day data. PeakLife incurs 8.43% MAPE and 0.9 correlation, outperforming LSTM that yields 11.19% MAPE and 0.77 correlation.

In summary, PeakLife effectively captures both low-variance and volatile utilization patterns, achieving high precision and trend prediction.

Sensitivity analysis and peak utilization prediction. Table 10.2 and Table 10.3 summarize model performance across different historical input window lengths $l \in \{72, 144, 288, 576\}$ corresponding to durations of 6 hours and up to 2 days, given the 5-minute sampling interval. For average utilization shown in Table 10.2, which exhibits relatively low variability ($\mu = 0.046$, $\sigma = 0.021$), PeakLife consistently achieves relatively high PCorr, and benefits from longer input histories, outperforming ARIMA and LSTM. For maximum utilization shown in Table 10.3, characterized by higher variability (μ=0.483, σ=0.137) and more irregular trends we use MAPE to evaluate the accuracy of predictions in capturing extreme values. With a l=288, similar to the results shown in Figure 10.3b, PeakLife achieves a MAPE of 8.54%, outperforming ARIMA (29.8%) and LSTM (11.05%). PeakLife also outperforms ARIMA and LSTM when trained with different lengths of historical data (i.e., when l is set to 72, 144, or 576). These results demonstrate the robustness of PeakLife in predicting diverse workload patterns.

Table 10.2: Pearson correlation for predictions on the average utilization (higher is better)

Historical Length (l)	Avg Utilization μ=0.046, σ=0.021		
	ARIMA	LSTM	PeakLife
l=72	0.767	**0.845**	0.838
l=144	0.765	0.884	**0.891**
l=288	0.784	0.924	**0.930**
l=576	0.814	0.920	**0.933**

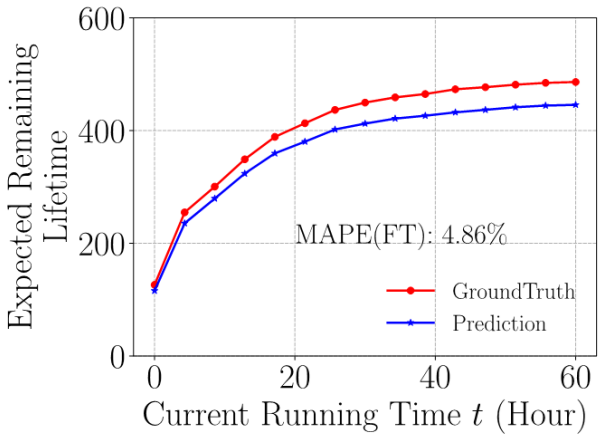
Table 10.3: MAPE for predictions on the maximum utilization (% – lower is better).

Historical Length (l)	Avg Utilization μ=0.483, σ=0.137		
	ARIMA	LSTM	PeakLife
l=72	31.76	14.7	**13.9**
l=144	33.59	12.03	**9.91**
l=288	29.8	11.05	**8.54**
l=576	28.87	12.08	**8.76**

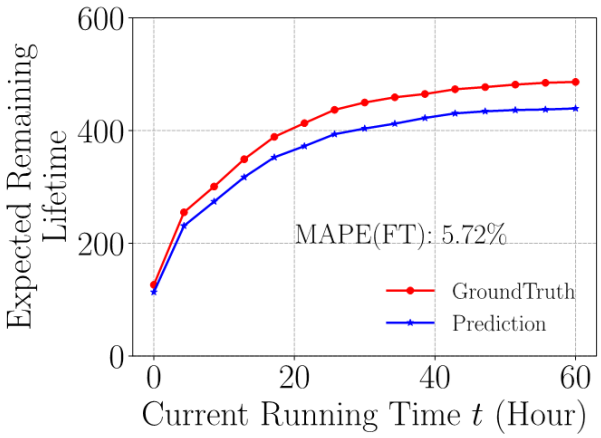
10.1.6 Lifetime Prediction Results

We evaluate the ability of PeakLife's surrogate model to predict the remaining VM lifetime in two cases: (1) for VMs already in the system, for which the model had the opportunity to observe their past behavior, and (2) for newly arriving VMs. For case (1), we train the model using historical data and keep 20% of the dataset as the test set. For case (2), we use a similar splitting strategy but ensuring that the test set contains entirely new VM IDs.

Results. In terms of prediction accuracy, PeakLife achieves a MAPE of 4.86% for case (1) (Figure 10.4a) and 5.72% for case (2) (Figure 10.4b), indicating its effectiveness in VM lifetime estimation. For both cases, the model tends to slightly underestimate the lifetime of long-running VMs, which is attributed to their lower prevalence in the dataset. Nevertheless, such estimations for long-running VMs (e.g., those exceeding 24 hours) remain sufficient for migration decisions.



(a) VM lifetimes predictions for VMs already seen on previous time periods.



(b) VM lifetimes predictions for unseen VMs

Figure 10.4: Predicted lifetime evaluated with two testing sets.

10.1.7 Improving VM and Resource Management

In this section, we evaluate the impact of PeakLife VM management. We show how utilization and lifetime, are translated into observable improvements in operational efficiency (e.g., migration counts, and SLO violations) in a cloud environment.

Simulation Setup. We use the extended CloudSim toolkit we developed in MLSysOps to implement software-in-the-loop simulations: the actual implementation of the Predictor and Cluster-level Coordinator operate within the simulation environment, which – in turn – simulates VM creation, lifetime, and time-varying resource requirements.

Our experiment uses a subset of the Azure dataset containing 200 VMs, across 20 hosts for one day (86,400 seconds), using 5-minute scheduling intervals. The baseline utilization threshold θ_{util} is set to 0.8, and the lifetime threshold θ_{ratio} is set to 0.5 hours.

Baselines. We compare PeakLife against the following baselines:

1) Local Regression Minimum Migration (LRMMT): A heuristic consolidation strategy that performs better than similar heuristic strategies[46].

2) LSTM prediction-based strategy for CPU utilization. We compare PeakLife-based results with those obtained using LSTM predictions for CPU utilization. Lifetime predictions are not produced by the LSTM-based model. To ensure a fair comparison, we also include results for a configuration of PeakLife where lifetime is not used for migration decision-making (the loop in lines 7-9 of the algorithm in Table 10.1 is skipped, and T_{rem} is eliminated from the VM cost calculation, in line 10). This variant, named PeakLife Util-only, also serves as an ablation analysis step for PeakLife.

3) Oracle relies on future knowledge. This ideal case assumes perfect knowledge of future CPU utilization and lifetimes, representing the best achievable performance.

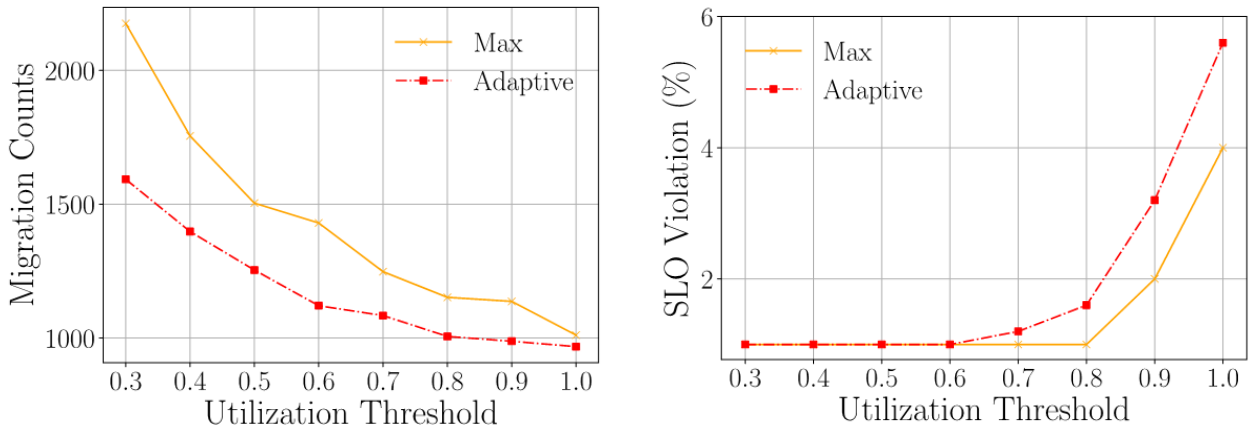
Metrics. We report total migration counts and SLO violation rates, which jointly reflect the efficiency of VM management and the system service quality.

Results. Figure 10.5 presents the experimental results. In terms of migration counts (Figure 10.5a) when applied to the average utilization workload, PeakLife reduces the average number of migrations to 1050 compared to LRMMT (1484 migrations, translating into a decrease of 41.33% for PeakLife) and LSTM-based strategies (1289 times), achieving results closer to Oracle (977 times). Specifically, PeakLife incurs fewer migrations when both utilization and lifetime predictions are incorporated, indicating better decision-making toward optimizing resource utilization. The reduction is also evident when comparing the total number of migrations induced by each strategy, showcasing the effectiveness of PeakLife in volatile conditions, and dealing with the "worst" situations when a conservative strategy needs to be generated.

For the SLO violation rate (Figure 10.5b), PeakLife also shows good performance, achieving the lowest violation rate (4.46%) among all baselines (except Oracle with the SLO violation rate of 4.41%). The incorporation of lifetime predictions further enhances its ability to prevent service disruptions caused by migrations, as seen in the difference between PeakLife Util-only and PeakLife. Compared to LRMMT (6.02%) and LSTM-based (6.03%), PeakLife strikes a balance between minimizing migrations and maintaining service quality.

In summary, PeakLife effectively reduces migration counts while maintaining low SLO violation rates. Thanks to the prediction accuracy of the deep surrogate model, PeakLife offers a lightweight solution with performance remarkably close to Oracle.

[46] P. Arroba, J. M. Moya, J. L. Ayala, and R. Buyya, "Dynamic voltage and frequency scaling-aware dynamic consolidation of virtual machines for energy efficient cloud data centers," Concurrency and Computation: Practice and Experience, vol. 29, no. 10, p. e4067, 2017.



(a) Migration counts. **(b) SLO violation rate**

Figure 10.5: Comparison of migration counts and SLO violation rate with the input traces of both average and maximum utilization

10.1.8 Model Prediction and Training Overhead

Prediction Overhead. The profiling of PeakLife is carried out on a system equipped with an Intel Core i9 CPU. Experiments show that PeakLife generates predictions in just 0.21 seconds per call. This translates to an overhead of 0.07% compared to the scheduling interval (set at 300 seconds), which is negligible. This overhead can be further reduced via batch inference and parallelism. The deep surrogate model is executed independently on each host to produce predictions for the VMs assigned to the specific node, enabling distributed inference with minimal latency. Moreover, PeakLife has a small memory footprint of about 1.6MB (measured using the PyTorch Profiler).

Training Time. PeakLife requires only 1.78 seconds per epoch during training thanks to its lightweight design. The low computational cost makes it feasible to apply periodic retraining with evolving workloads while maintaining competitive generalization performance.

In conclusion, PeakLife enables scalable and efficient deployment through its distributed architecture (see Figure 10.1). The Predictor runs independently on each host for low-latency inference, while the Cluster-level Coordinator aggregates precomputed predictions, ensuring minimal global overhead. With batch inference, multi-threading, and a small memory footprint, PeakLife can be seamlessly integrated into cloud infrastructures with negligible performance impact.

11 Model retraining

Model retraining can be triggered either automatically due to a detected performance degradation (i.e., more misclassifications than expected) or manually due to a planned update to consider expected changes on the input (i.e., to integrate more classes). Figure 11.1 depicts the integration of the monitoring agent on the MLConnector that is responsible for the calculation of the model drifting and the storage of valuable and representative samples in the context of data-centric AI, continual learning and few-shot learning. As introduced in the background section, continual learning allows AI models to learn continuously adapting to new information without forgetting previously acquired knowledge while few-shot learning enables model retraining with very limited data.

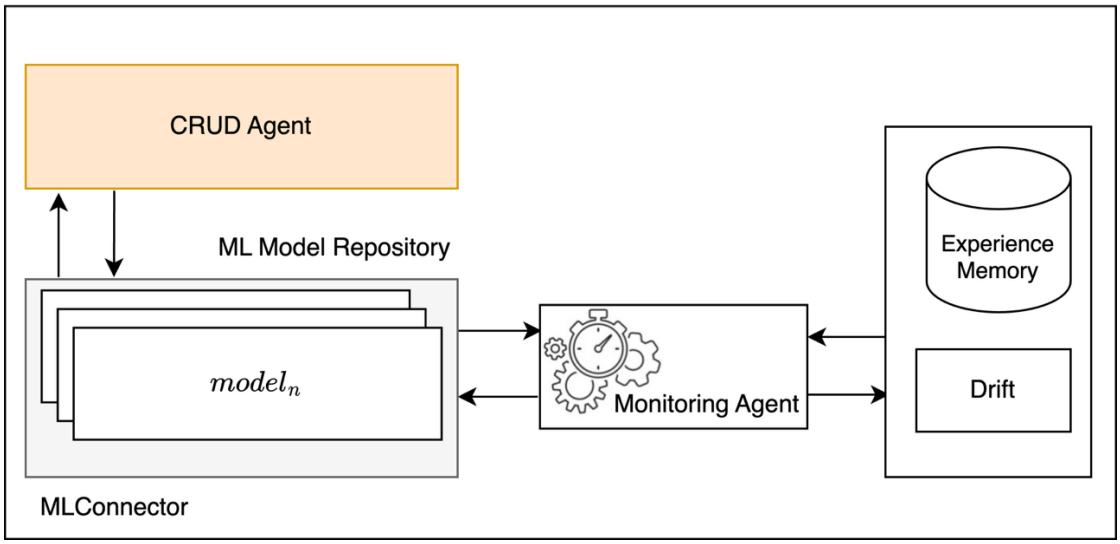


Figure 11.1: Condensed version of MLConnector that shows ML model monitoring.

Collaborative model training

Collaborative training approaches, such as Federated Learning[47] (FL), enable training models in a distributed manner while keeping the data decentralized. Usually, the devices that participate in FL are referred to as data owners. Data owners train the model locally for a small number of rounds and share the updated model with a centralised entity (i.e., a server). The server, in turn, in an aggregation phase, produces a global instance of the model. Finally, the updated global model is sent back to the data owners to start a new training epoch.

Resource-efficient learning and the importance of Split Learning

On-device training, even for a limited number of rounds, can be quite computationally intensive and demanding in terms of computing resources. Recently released mobile GPUs that are designed to support the training of NN models, still perform poorly. One factor is the large batch size requirement to ensure good accuracy and convergence speed. Thus, during training, large tensors containing the produced activations during forward propagation will be generated. Depending on the size of the model, this may require the device to allocate a considerable amount of memory to be able to train the model.

Figure 11.2 depicts, the portion of memory occupied by a process held in the main memory during the training process in FL (the data owner has the full model). This is significantly high for constrained devices to handle. For instance, there are small devices like the RPi 3, or the GPU of NVIDIA Jetson Nano, that cannot support such type of training. Also, even if a device can perform on-device training, its limited computing resources

[47] B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. y Arcas, "Communication-efficient learning of deep networks from decentralized data," in AISTATS. PMLR, 2017, pp. 1273–1282.

will cause the stragglers effect whereby slower devices can lead to unacceptably large training delays in the entire FL process. As a result, most FL applications consider models with fewer parameters.

To tackle this issue, Split Learning (SL) assigns to a compute node (i.e., a container running on a computationally capable device) the largest part of the model to perform the respective training process, whereas data owners only keep a small part of the model. This makes it possible for resource-constrained data owners to train deeper models. Moreover, SL is beneficial in terms of communication load. Namely, data owners need to exchange only the intermediate activations and gradients with the compute node, whereas in FL they have to exchange the parameters of the entire model with the aggregator -- this can be hundreds of MB. In the context of MLSysOps, we have developed SplitPipe, an SL framework that enables ML model training using significantly fewer resources.

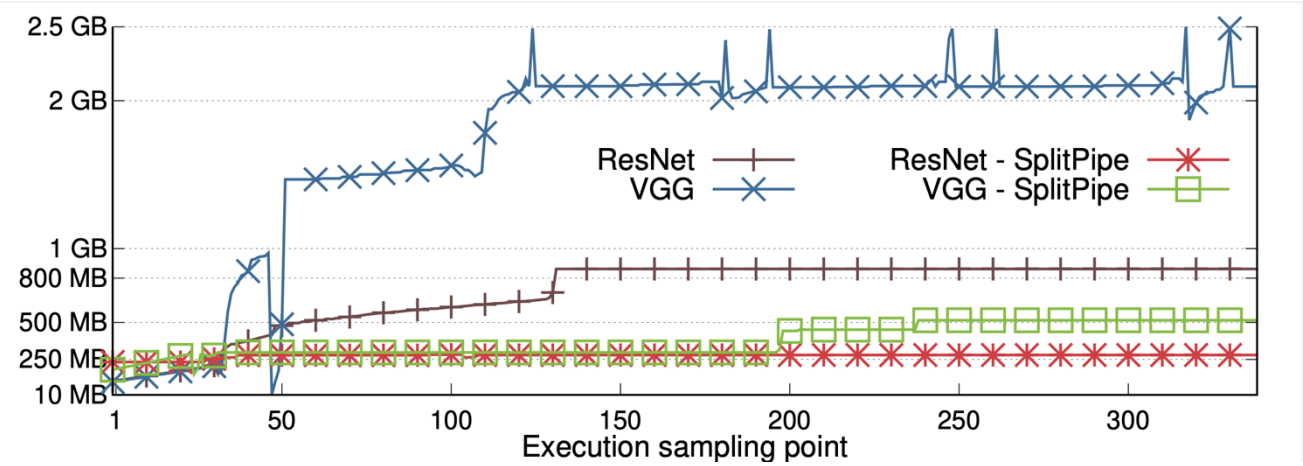


Figure 11.2: Memory usage measured on RPi 4, for training ResNet-101 and VGG-19, with FL and SplitPipe.

Parallel Split Learning

In SL protocols, the model is vertically split into multiple parts, with a subset of them being offloaded to more powerful compute nodes. Figure 11.3 depicts a model that is split into different parts through cut layers. The first and the last part of the model (before the first cut and after the last cut, respectively) are kept locally at the data owner, while the intermediate part is assigned to a compute node or further divided (via the possible cut) and assigned to two compute nodes.

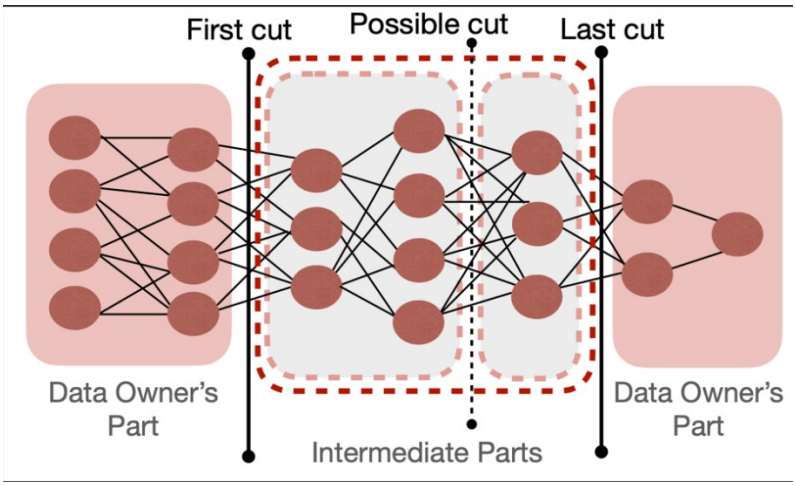


Figure 11.3: ML model partitioning.

Moreover, Figure 10.4 shows one training iteration between one data owner and one compute node (single hop). The data owner initiates each iteration. Firstly, during the forward propagation, the nodes exchange with each other (starting from the first model part) **forward()** requests containing the activations produced at the corresponding cut layers. These are the steps 1 and 2 in Figure 11.4. Then, during the back-propagation, following the opposite direction, and starting from the last model part the nodes compute the gradients and encapsulate the ones of the cut layers into **backward()** requests (steps 3a, 4a). Note that, when a node has computed the gradients, it can concurrently update the weights of the model part it is in charge of (steps 3b, 4b).

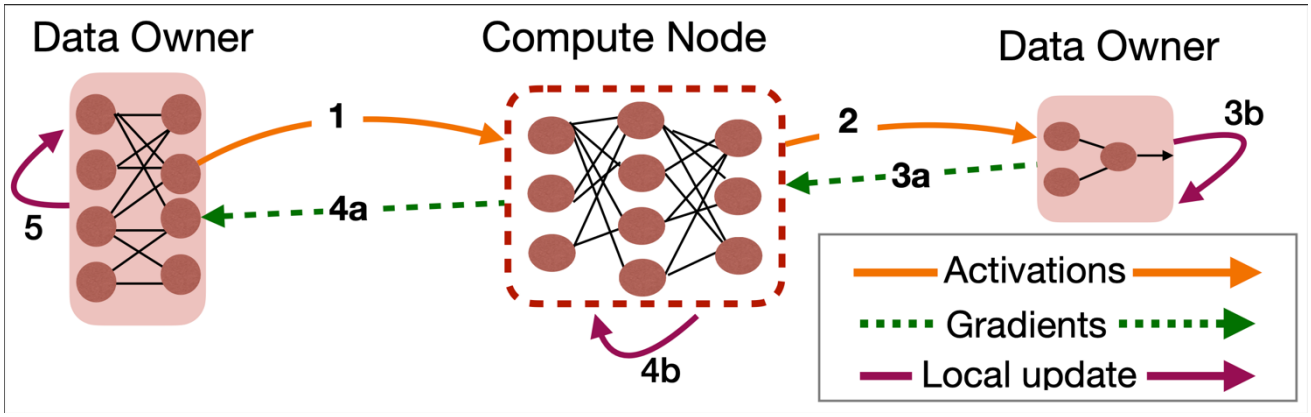


Figure 11.4: One batch update in SL with one compute node.

In the conventional SL approach, multiple data owners share a common instance of the intermediate model part and are served by the compute node in a round-robin fashion. However, the sequential serving of the data owners increases the training delay. Parallel SL speeds up training and enhances scalability.

As is shown in Figure 11.5, the compute nodes keep a different version of the model parts for each data owner. This allows each data owner to apply SL independently from other data owners. At the end of an epoch, the model parts from all the data owners are aggregated using techniques such as FedAvg, for this operation the data owners employ an aggregator. Notably, the intermediate model parts can be aggregated locally at the compute nodes, without any communication. Parallel SL performance is constrained by compute nodes' capacity, especially for numerous data owners. A relatively straightforward way to scale for a large number of data owners is to apply horizontal scaling, involving the addition of more compute nodes that can serve different sets of data owners. However, this introduces synchronization challenges for completing epochs, as compute nodes must aggregate intermediate model parts. Also, in Parallel SL, each compute node has to hold the entire intermediate part of the model for each data owner it is in charge of. This becomes problematic for models with extensive parameters, demanding substantial memory, and often requiring powerful (and costly) compute nodes.

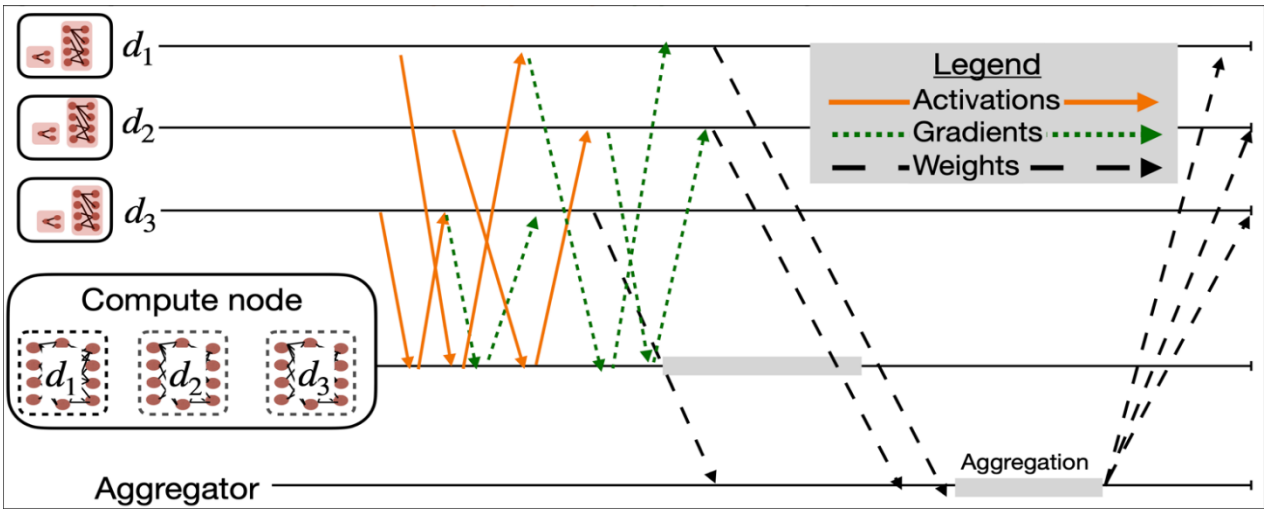


Figure 11.5: Parallel SL protocol with three data owners.

Multihop Parallel Split Learning

Going further, the intermediate model part can be split into smaller parts, which can be assigned to separate compute nodes. Specifically, by splitting further the model parts, we observe the following advantages:

(1) Resource relaxation: Memory and processing requirements for compute nodes are relaxed, enabling the use of less powerful and more affordable compute nodes. Figure 11.6 shows the reduction of the memory demands on the compute nodes while the multihop level increases. In Figure 11.6, the first/last cut layers are user-defined. While the intermediate model parts are calculated using SplitPipe which, finds the split policy that minimizes the training delay.

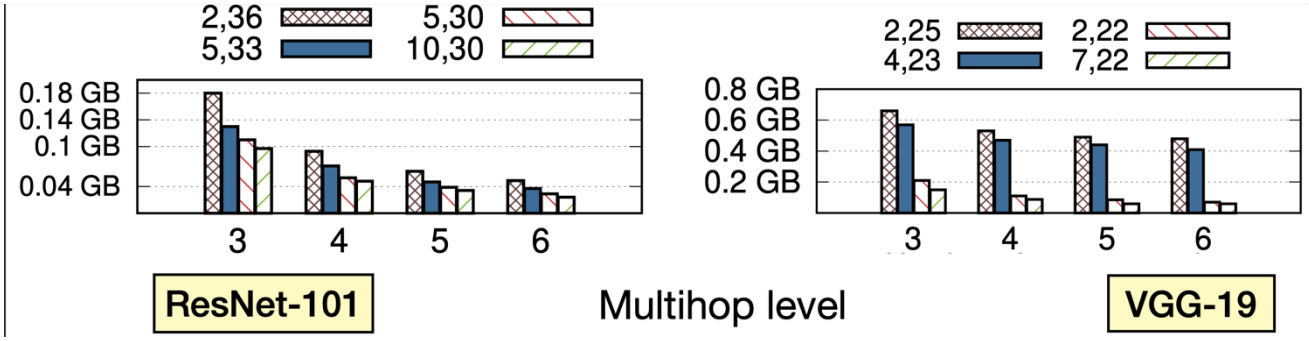


Figure 11.6: Memory demand for the compute node with the largest (memory-wise) model part for different multihop levels.

The smallest multihop level is 3 (i.e., one compute node), and the largest is 6 (i.e., four compute nodes). Also, for each model, we select different user-defined first and last cut layers. Note that VGG19 has 25 indivisible layers while ResNet101 has 37.

(2) Knowledge conceals: As the multihop level increases, each compute node is in charge of smaller model parts (i.e., consisting of a smaller number of layers) and hence has less knowledge about the ML model, which is an an important issue in SL. The research area which focuses on privacy concerns of SL mostly involves semi-honest attacks with a single split, whereas in our case we build upon the no-label sharing configuration (i.e., at least two splits). Also, the attacks depend on the received activations/gradients, which can be protected

with defence techniques. Only the compute nodes of PCAT[48] exploit the model part they are in charge of. But, even though PCAT outperforms other novel attacks, it remains sensitive to the number of offloaded layers, and hence one can challenge PCAT by increasing the multihop level.

(3) Pipeline Parallelism (PP): Splitting the model into multiple parts, enables PP, which is the combination of Data Parallelism (DP) and Model Parallelism (MP). The benefits of PP are (i) accelerated training and (ii) support for larger models. However, it is mainly used in centralized learning for a single source of data (i.e., one data owner). Even though DP has been adopted by decentralized learning (also referred to as client-based distributed learning) with SplitFed and other variants of Parallel SL, the concept of PP has not been widely explored in such a configuration. However, keep in mind that in this case, the implementation of PP is even more challenging as (i) the compute nodes store multiple instances of the model (one for each data owner), (ii) some model parts (usually the first and/or the last) are handled by resource-constrained devices, and (iii) the compute nodes do not have access to the data. Therefore, existing PP frameworks are not applicable.

In the context of MLSysOps, we developed SplitPipe, a distributed learning framework that combines SL with FL in a way that achieves better scalability. SplitPipe is a supplementary protocol to FL since it enables resource-constrained devices to participate in the training of large deep-learning models and restrains the straggler's effect. This is presented in Figure 11.2, where the on-device memory demands can be dropped up to 76% with SplitPipe. Furthermore, SplitPipe supports model splitting into multiple parts that are assigned in different compute nodes to (i) relax the memory requirements of each compute node, (ii) reduce the cost of compute nodes, and (iii) restrict the model's exposure to a single party. Data owners can choose the desired multihop level, which internally translates to a suitable partitioning of the model, with each part being assigned to a different compute node allowing pipelined parallelism (PP). In fact, given the desired multihop level, SplitPipe will optimize the selection of the intermediate split points (i.e., the model parts assigned to the compute nodes) with the objective of the minimization of the training latency.

In summary, the contributions of SplitPipe are:

1. SplitPipe is the first Parallel SL-based framework with multihop support. It is modular, easily extensible to support any model type, and is publicly available[49]
2. We provide and validate an analytical model for estimating the expected performance of SplitPipe. We show that the analytical model provides estimates of the measured system performance, with an error of less than 3.86%.
3. To the best of our knowledge, this is the first work that models and optimizes the splitting selection for multihop SL.
4. We evaluate SplitPipe for a wide range of scenarios using a realistic testbed. We show that the proposed protocol is robust to the stragglers effect and can significantly reduce the cost of the compute nodes with a slight increase in the training time.

[48] X. Gao and L. Zhang, "{PCAT}: Functionality and data stealing from split learning by {Pseudo-Client} attack," in 32nd USENIX Security Symposium (USENIX Security 23), 2023, pp. 5271–5288.
[49] https://github.com/jtirana98/MultiHop-Federeated-Split-Learning

12 Transfer Learning

This chapter presents two complementary applications of transfer reinforcement learning for autonomic resource management in heterogeneous cloud-edge environments: (i) transferable DRL agents for virtual machine (VM) management across diverse datacenter infrastructures, and (ii) cross-domain DRL agents for job placement in a cloud-edge continuum. Both approaches share the central challenge of **state–action dimension mismatch** when moving agents between infrastructures of different scale, topology or resource characteristics. We describe each use case's problem setting, system design, RL formulation, evaluation methodology and key results, and discuss how common abstractions enable seamless policy transfer.

Problem Description

Modern cloud and edge infrastructures evolve rapidly: hardware upgrades, topology reconfigurations, and workload shifts continually change the underlying resource graph. Classical DRL agents trained for one configuration become brittle when deployed on another, due to differences in the dimensionality or semantics of their state and action spaces. This **state–action mismatch** forces expensive agent redesign and retraining for every infrastructure change, undermining the goal of continuous, automated operations.

Transfer reinforcement learning aims to reuse knowledge from a source environment in a target one, reducing sample complexity and convergence time. Yet, most transfer RL methods assume a shared, fixed-dimensional state–action embedding; they cannot accommodate the heterogeneity of real datacenters or multi-datacenter continua. Addressing this gap requires **infrastructure-invariant representations** that abstract away domain-specific details while preserving the essential information needed for optimal control.

In the two studies presented here, we tackle two canonical resource management tasks—VM lifecycle management and job placement—by designing DRL agents whose **internal network architectures** incorporate a state-abstraction front end. This front-end maps heterogeneous, variable-sized inputs into a fixed-dimension, dense embedding, enabling the downstream policy network to remain unchanged across domains.

12.1.1 VM Management Use Case

- **Infrastructure**: Single datacenter with a fixed set of hosts and three different types of VMs (2, 4, and 8 CPU cores).
- **Broker Actions**:
  - **Instantiate VM** of a given type on a host.
  - **Terminate VM** on a host.

Figure 12.1 shows an example of the system model that this work is concentrated in.

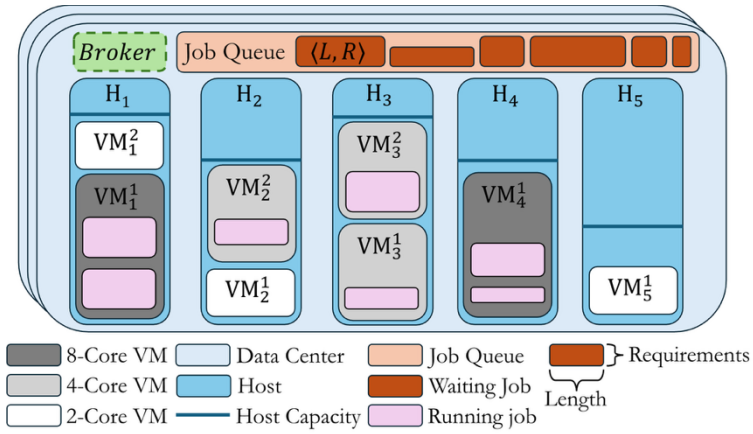


Figure 12.1: A data center with five hosts of different capacities and utilization. There are three types of VMs in the system. Some jobs are being executed while others are waiting in the queue

A cornerstone of this work is the **novel tree-based state representation** (Figure 12.2), designed to efficiently and compactly encode the hierarchical structure of the cloud infrastructure at any given time. The primary objective is to define a **fixed-size state representation** that supports an RL model which can be transferred seamlessly across infrastructures of varying sizes, thereby eliminating the state–action space mismatch problem that typically arises when infrastructures evolve or differ.

The tree structure captures the full resource hierarchy — from **datacenter → hosts → virtual machines (VMs) → jobs** — where each node encodes both its resource capacity and the number of child elements (e.g., number of VMs on a host, or number of jobs in a VM). By traversing the tree using a **preorder edge-count traversal**, this approach produces a flattened array that serves as a compact, informative embedding of the system's current state. To ensure compatibility across infrastructures of different sizes, **zero padding** is applied to the resulting array up to a predefined maximum length. This guarantees a **fixed-dimensional state vector**, allowing a single RL model architecture to operate across multiple datacenters with different scales and configurations — without needing to change the network architecture or suffer from state–action space mismatches.

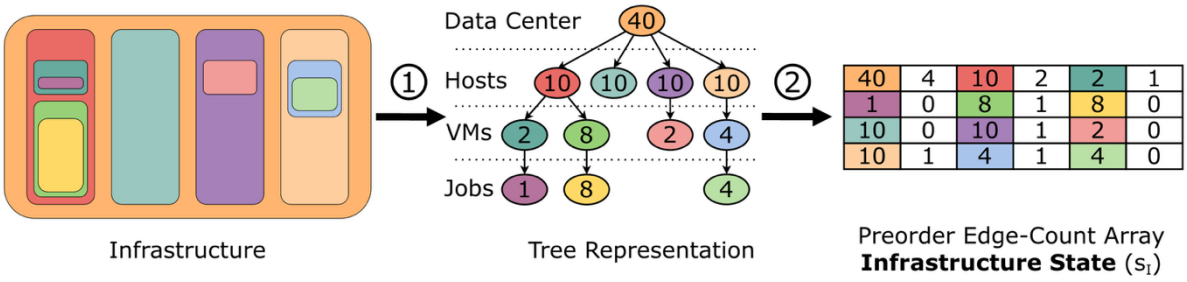


Figure 12.2: Tree representation of datacenter state and preorder edge-count traversal yielding a feature vector

Two key techniques were essential for enabling stable and effective transfer of the RL agents across infrastructures of varying size and scale: **action masking** and **weight freezing with zero-initialization**. These mechanisms address challenges related to variable host counts and zero-padded state representations, ensuring that a single policy network can generalize across domains without architectural changes.

Action Masking

The RL agent employs a **discrete action space**, where each action corresponds to the creation or termination of a VM on a particular host. The total number of possible actions therefore depends on the number of hosts available in the infrastructure. To support cross-domain transfer with a consistent action space, a **fixed maximum number of hosts** is defined, ensuring that the policy network always outputs a fixed-size action vector regardless of the actual host count.

In smaller infrastructures, this results in some actions being invalid (for example, attempting to manage VMs on non-existent hosts). To avoid this, the agent applies **action masking**: invalid actions are masked at each timestep, and the learning process focuses only on valid decisions. This approach stabilizes training and speeds up convergence in smaller domains, without requiring any changes to the policy network architecture.

Weight Freezing and Zero Initialization

The **tree-based state representation** is zero-padded to a predefined maximum size to ensure a fixed-length input to the policy and value networks. In smaller infrastructures, portions of the input vector correspond to non-existent elements (e.g., unused host slots) and are filled with zeros.

To prevent instability when transferring agents to larger infrastructures (where those previously unused portions become active), the associated neural network weights are:

- **initialized to zero**, ensuring that they do not influence policy decisions during initial training;
- **frozen** (excluded from gradient updates) when training in smaller infrastructures;
- **unfrozen** upon transfer to a larger domain, allowing the network to gradually learn new relevant features for the expanded state space.

Together, **action masking** and **weight freezing** play a critical role in mitigating state–action mismatch during transfer. They ensure that agents trained on smaller domains can be smoothly and safely adapted to larger or more complex infrastructures, while preserving their existing knowledge and avoiding catastrophic forgetting.

12.1.2 Job Placement Use Case

- **Infrastructure**: Multiple datacenters across cloud-edge continuum, each with distinct capacity limits and latencies. Three datacenter types: cloud, edge, far-edge (on-premises).
- **Broker Actions**:
  - **Place incoming job** on one of the available datacenters, subject to capacity and latency trade-offs respecting each job's soft deadline constraints.

Here, the state is represented as a concatenation of:

1. **Continuous features** (e.g., current load per datacenter).
2. **Discrete features** (e.g., datacenter type) mapped via an embedding layer.

Figure 12.3 presents the different components of the state space that the RL broker receives from the environment.

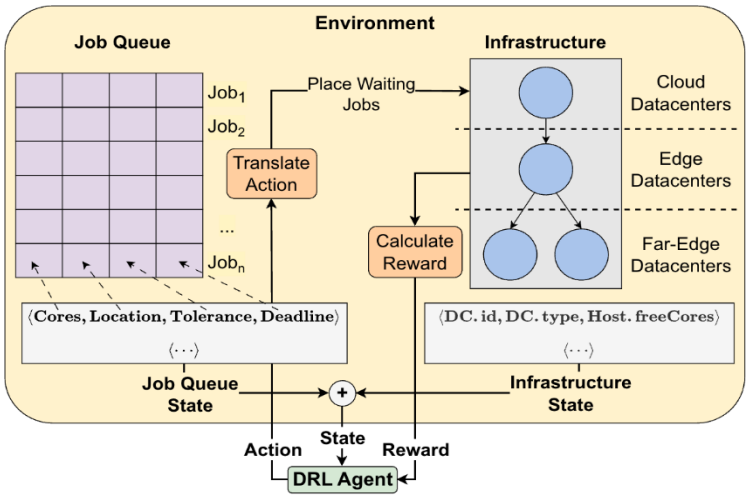


Figure 12.3: The DRL agent takes the state as input, selects an action, and receives a reward based on the environment's job placement response. This loop repeats until the episode ends

This work introduces a custom policy network for the RL broker. In this network, the raw observation features are first separated into a continuous feature stream and a categorical feature stream. For the categorical features, embeddings are created and for each independent feature a distinct identical MLP network is responsible. After the two streams are concatenated, an **adaptive residual** layer is introduced which is responsible in handling the environment instabilities upon transfers (Figure 12.4). By design, the size of the intermediate state representation is fixed and independent of the number of datacenters.

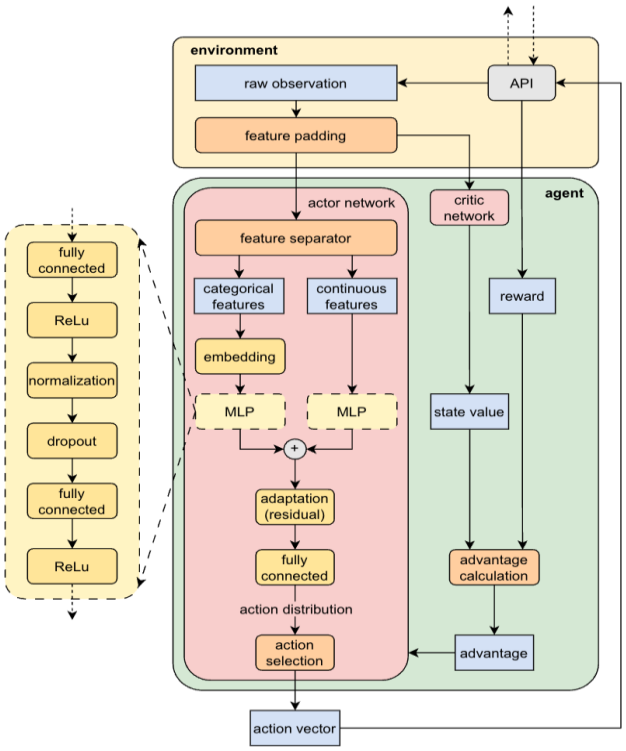


Figure 12.4: State abstraction module for job placement: separate MLPs for continuous/discrete inputs, followed by an adaptive residual fusion layer

ML Approach

Both use cases employ a **Proximal Policy Optimization (PPO)**[50] actor-critic algorithm for training, due to its stability and sample efficiency in continuous control tasks. Key elements of the RL formulation are summarized in Table 12.1

Table 12.1: Summary of RL formulation components for both use cases

Component	VM Management	Job Placement
State	Tree-traversal array + job-queue length	Datacenters load + jobs waiting characteristics
Action	{Instantiate VM type i on host h, Terminate VM v}	{Assign next job to datacenter d}
Reward	– Job queue: $-\alpha_1 \cdot$ jobs waiting – Cost: $-\beta_1 \cdot$ VM cores instantiated – Unutilization: $-\gamma_1 \cdot$ VM cores instantiated but unused – Invalid actions: $-\delta_1 \cdot$ action infeasible	– Job Placement Throughput: $\alpha_2 \cdot$ jobs placed – Placement Decision Quality: $\beta_2 \cdot$ Job QoS – Deadline: $-\gamma_2 \cdot$ deadlines violated

50 Schulman, John, et al. "Proximal policy optimization algorithms." *arXiv preprint arXiv:1707.06347* (2017).

Key Techniques Used and Novelties	Custom Tree-based state space, Action Masking, Weight Initialization & Freezing	Custom Policy Network with MLPs, Embedding and Residual Layers

Evaluation and Testing

Both studies use **CloudSimPlus** to simulate realistic cloud and edge infrastructures, with customizable host counts, and multi-datacenter scenarios.

- **VM Management**
  - **Source infrastructure**: Clusters with {4, 8, 16, 32} hosts, each host with 16 CPU cores.
  - **Target infrastructures**: All combinations of source and target cluster sizes were tested — that is, models trained on any one cluster size ({4, 8, 16, 32} hosts) were transferred to each of the other cluster sizes.
- **Job Placement**
  - **Source continuum**: Environment B - a cloud–edge continuum with four interconnected datacenters (1 cloud dataceneter, 1 edge datacenter, 2 far-edge datacenters)
  - **Target continuum**:
    - **Environment A** (smaller topology): 1 edge datacenter, 2 far-edge datacenters (no cloud datacenters)
    - **Environment C** (larger topology): Environment B + 1 additional edge datacenter, and 1 additional far-edge datacenter. In addition to the expanded topology, the workload pattern is also modified in this case to test the **robustness** of the RL model not only to infrastructure changes but also to significant shifts in job arrival patterns.
- **Workloads**: Synthetic datasets simulating bursty, non-stationary job arrival patterns, and variable demands.
- **Baselines:** heuristics, PPO from scratch.

Results

12.1.3 VM Management Use Case

DRL vs. Rule-based Comparison

Figure 12.5 compares our deep reinforcement learning (DRL) agent with three rule-based algorithms across three key optimization metrics: job queue ratio, allocated VM cores ratio, and unutilized VM cores ratio. These metrics are components of the DRL agent's reward function. After training the DRL agent for 2 million time steps, we evaluated the best episode's average metrics over five random seeds.

The DRL agent achieves a balanced optimization of all three metrics simultaneously, whereas each rule-based algorithm targets only one metric at the expense of others. Specifically, the rule-based algorithm minimizing job queue ratio (RB_q) prioritizes immediate job placement but incurs high resource inefficiency, reflected by the red colour points. The algorithm minimizing unutilization (RB_u) reduces unutilized cores by maintaining minimal VM allocation but results in longer job queues. The third rule-based approach (RB_a), which minimizes VM allocated cores by running resources only for the most demanding job, also leads to increased job queue times.

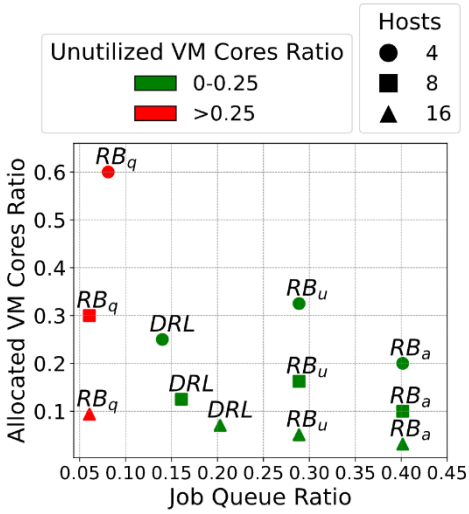


Figure 12.5: Performance metrics: DRL vs. rule-based (RB)

Overall, the DRL agent outperforms all rule-based approaches by effectively balancing job latency, resource allocation, and utilization, demonstrating its advantage in managing multiple competing objectives simultaneously.

Adapting to Infrastructure Changes

This experiment evaluates the agent's ability to adapt when transferred to infrastructures of different sizes. We compare two cases under the same job trace scenario, while varying the number of hosts:

1. An agent trained **from scratch** on the target infrastructure for 300 K timesteps.
2. An agent **transferred** from a source infrastructure after being pre-trained for 2 M timesteps, then fine-tuned for 300 K timesteps on the target infrastructure.

To comprehensively assess transferability, we consider all combinations of source and target cluster sizes, specifically: $4 \rightarrow \{8, 16, 32\}$, $8 \rightarrow \{4, 16, 32\}$, $16 \rightarrow \{4, 8, 32\}$, and $32 \rightarrow \{4, 8, 16\}$. The results of these experiments are summarized in Figure 12.6, and reveal the following key insights:

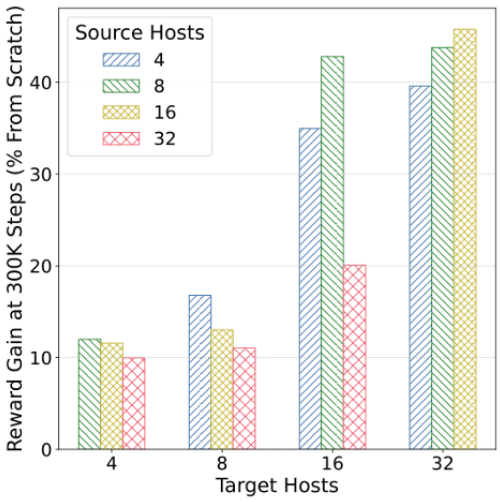


Figure 12.6: Reward gain (%) at 300K: transfer vs. scratch

Insight 1: **Larger clusters benefit the most from transfer learning.**

When transferring agents to larger infrastructures, the performance gains are more pronounced. Agents trained from scratch on larger clusters face a substantially larger state and action space, which increases training complexity and slows convergence. In contrast, transferred agents can leverage prior knowledge, enabling faster adaptation and significantly higher cumulative rewards within the same training duration. This trend is clearly visible in Figure 12.6, where reward improvements grow as the target cluster size increases.

Insight 2: **Source–target cluster size similarity impacts transfer effectiveness.**

The similarity between the source and target environments plays a key role in determining transfer efficiency. Agents transferred from source clusters of similar size to the target environment achieve better performance, as the underlying state distribution and dynamics are more closely aligned. For example, in the 16-host target scenario, agents transferred from the 8-host source perform better than those transferred from much smaller (4-host) or much larger (32-host) clusters. This indicates that choosing an appropriate source domain can further enhance transfer outcomes.

In addition to faster convergence, we frequently observe that transferred agents can even surpass the performance of agents that were trained and converged in the target environment. This effect arises because agents trained in a single infrastructure may become trapped in suboptimal local optima. Exposure to a different source environment provides the agent with more generalized knowledge, helping it to escape prior limitations and discover better policies.

In this experiment, transferred agents were fine-tuned in the target infrastructure for only **25 %** of the original full training time, yet they were able to surpass the previously converged target-only agents. This is possible because the transferred agents had already explored a wider portion of the state space during source training, accelerating their ability to adapt.

The results, presented in Figure 12.7, highlight another important insight:

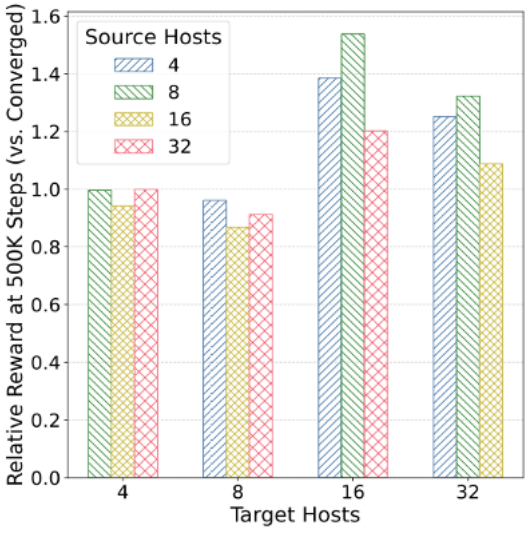


Figure 12.7: Relative reward: transfer (500K) vs. scratch (2M)

Insight 3: **Small-to-large transfers are more effective than the reverse.**

Agents trained in smaller infrastructures often outperform their fully converged counterparts when transferred to larger ones. This benefit is linked to the **weight freezing mechanism**: in smaller environments, certain parts

of the network are frozen, limiting overfitting. When transferred to larger infrastructures, these weights become active, allowing the agent to learn new information while retaining prior knowledge. In contrast, transferring agents from larger to smaller environments may result in **knowledge loss** or degraded adaptation due to mismatches between source and target state spaces. Nevertheless, even in these cases, transferred agents still outperform training from scratch — as shown in the **32→16** scenario, where the transferred agent exceeded the fully converged agent's reward after just 25 % of the training time.

Overall, the experiments demonstrate **up to 54 % reward improvement** compared to fully converged target-only agents, confirming the strong benefits of transfer learning in dynamic cloud environments.

Adaptation to Job Request Pattern Changes

In addition to infrastructure changes, we conducted experiments to analyze adaptation to job request pattern variations. Results show that agents adapt faster to workload changes since the state and action spaces remain unchanged. A transferred agent achieved up to 99% of the original reward on reduced load and maintained reasonable performance on higher loads. When tested on extended-duration workloads, the transferred agent quickly matched or surpassed a baseline trained over millions of steps, whereas direct training on long traces was unstable. These findings highlight the benefits of transfer learning and suggest that incremental or curriculum learning could further enhance adaptability.

12.1.4 Job Placement Use Case

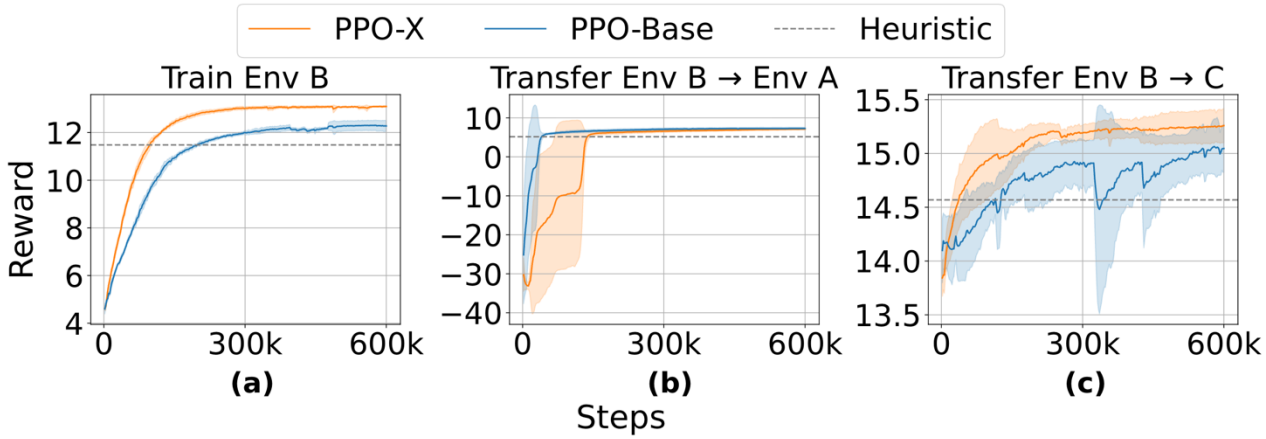


Figure 12.8: Reward comparison of proposed method vs. baselines

We compared our proposed PPO extension (PPO-X) against a vanilla PPO baseline and a heuristic approach across three environments with varying complexity and resource availability. Both PPO variants significantly outperformed the heuristic after 600k training steps. During training, PPO-X consistently achieved better results than the vanilla PPO baseline, particularly in more complex environments (Figure 12.8a).

However, in simpler environments with smaller state-action spaces, such as Environment A, PPO-X's larger model size sometimes resulted in slower convergence compared to vanilla PPO (Figure 12.8b). In these cases, the simpler PPO baseline may be preferred due to faster training and lower computational overhead.

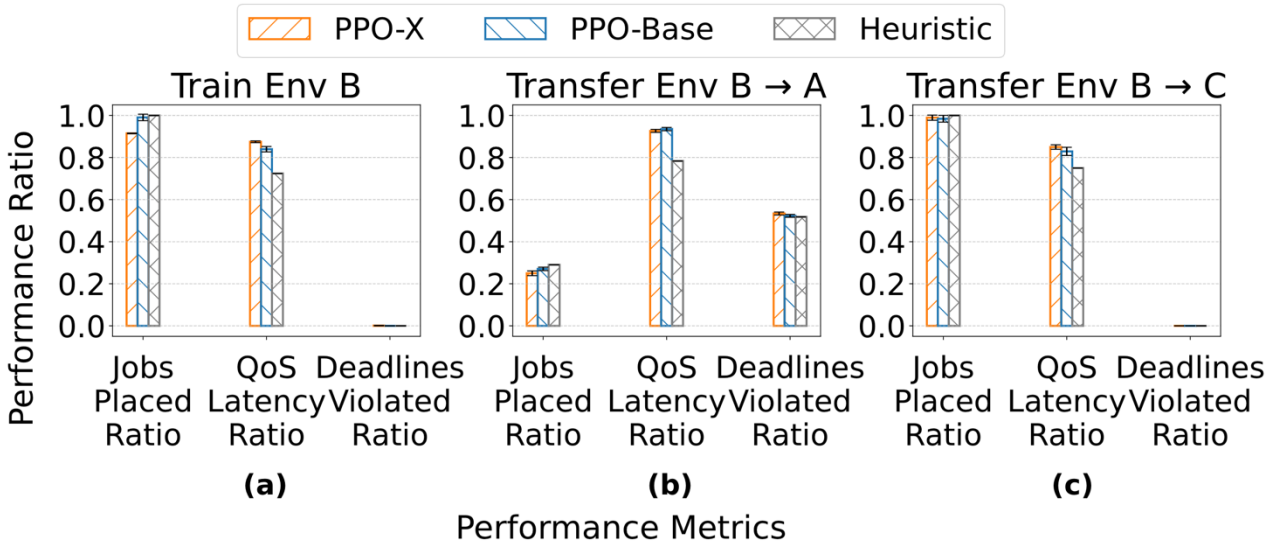


Figure 12.9: Performance metrics comparison of proposed method vs. baselines

Interestingly, practical system performance metrics shown in Figure 12.9 reveal that Environment A has worse job placement, QoS latency, and deadline violation ratios compared to the more resource-rich Environment B. This counterintuitive outcome is explained by Environment A's limited host availability, which creates a stressful workload pattern despite its smaller state-action space.

The advantages of PPO-X become clear under workload variability and sudden changes. When evaluated with a challenging workload trace (Figure 10.8c), PPO-X not only achieved higher cumulative rewards but also demonstrated greater stability and robustness across multiple runs, unlike the vanilla PPO baseline which showed instability. This improved stability and performance suggest that PPO-X is better suited for dynamic environments with fluctuating conditions, providing a more reliable and robust solution for real-world job placement challenges.

13 Delivery and integration of ML models

MLConnector

This section describes the ML API (MLConnector) design. It is based on Flask REST and is responsible for delivering all the machine learning needs within MLSysOps, from model registration, training and retraining, deployment to monitoring and drift detection. Figure 13.1 show the flow of processes within the MLConnector.

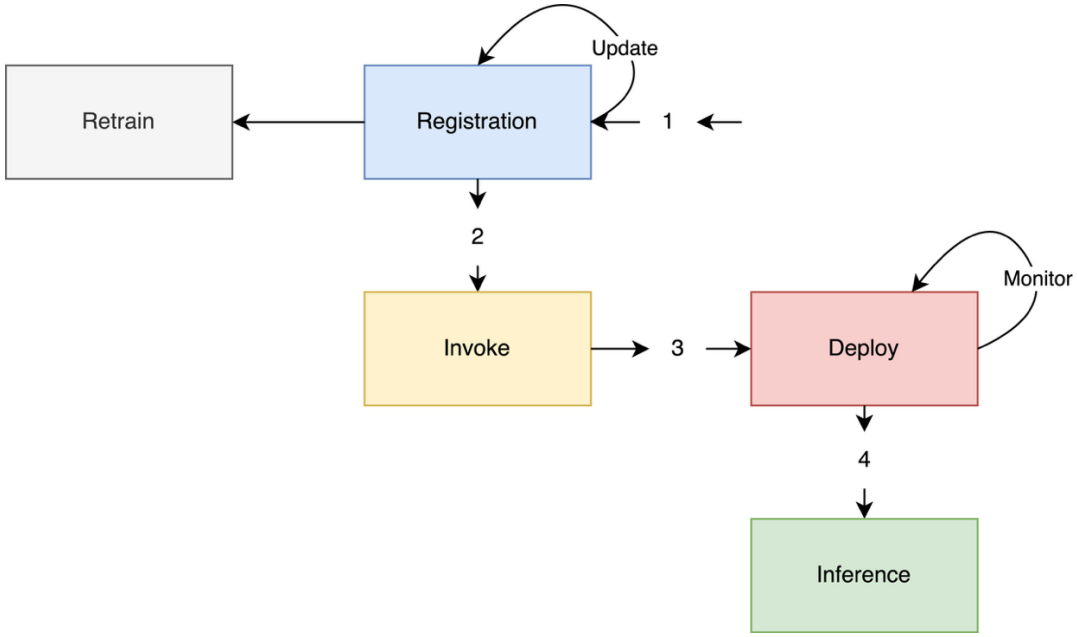


Figure 13.1: MLConnector machine learning flow

13.1.1 Declarative description of ML models

Each ML model is defined by a set of properties. These include model type, its internal hyperparameters, the minimum resources required to train and run that model, and its explainability characteristics. MLSysOps agents query the model repository with such parameters whenever they need to identify candidate models for a task.

```
MLInstanceID:
      -Type: String (10)
      -Description: Alphanumeric string defining the application instance
      -Example: "v18t4bnp2w"
ModelDescription:
      name:
            -Type: String
            -Description: Name of the machine learning model to be trained and
            deployed
            -Example: "RandomForest"
      type:
            -Type: String
            -Description: The type of model to be built. This can be classification,
            regression, or clustering. If the name specified in the model description
            conflicts with this parameter, then this parameter will be ignored
            -Example: [Classification, Regression, Clustering]
      label_no:
            -Type: INT
            -Description: If the model type is classification, then specify the number
            of labels. Otherwise, it will be ignored.
            -Default: 2
```

```
            -Example: 3
ModelPerformance:
      accuracy_metric:
            -Type: List of string
            -Description: The performance metric(s) used to evaluate model
            performance. One or more values can be specified. In cases where multiple
            metrics are defined, a prioritization scheme can be implemented. For now,
            the API just returns all the scores, and the calling agent decides.
            -Default: ["F1"]
            -Example: ["F1", "Recall", "Precision", "Accuracy"]
      threshold:
            -Type: List of INT
            -Description: The minimum percentage value for each of the metrics defined
            above. If more than one metric value is defined above, then the list above
            and this one should match.
            -Default: 85
            -Example: [86, 90, 86, 95]
Hyperparameters:
      -Type: Key value pair
      -Description: List of hyperparameters and the corresponding desired values.
      -Default: None
      -Example: {"max_depth":5, "n_estimators": 10, "criterion": "mse"}
Explanation:
      -Type: List of strings
      -Description: The level of explanation to return from the model. More than one
      level can be specified.
      -Default: None
      -Example: [None, low, medium, high]
Resources:
      - Key: value of the description that will be used to deploy the ML application
      - Example {"cpu":4, "gpu": 3, "disk":512, "ram":32, "archt": "arm64"}
Owner: [Node, Cluster, Continuum]
Environment:
      -OS: Ubuntu
      - container: Docker
```

As depicted by the dashed line in Figure 13.2, the design of the API is broken down into two main stages: the API for the initialization phase and the API for the deployment phase. In the first stage, we define the initialization API endpoints. At this stage, the MLConnector only sends the Model IDs to the node. These involve the following steps.

1. **Request**. Here, the continuum/cluster/node agent specifies the type of the ML model it needs using the ML model description defined above.
2. **Model description.** Once a matching model is found, the API returns a list of all the ML models that match the agent's requirements.
3. **Deployment REQUEST**. At this stage, the agent picks one model and sends back a request to API to deploy the selected ML model.
4. **Deployment instructions**. The API pushes the request to the queue for deployment and returns a deployment ID to the agent to be used in the next stage. This is a continuation of step 3 above.

13.1.2 API for ML model use and explanations delivery

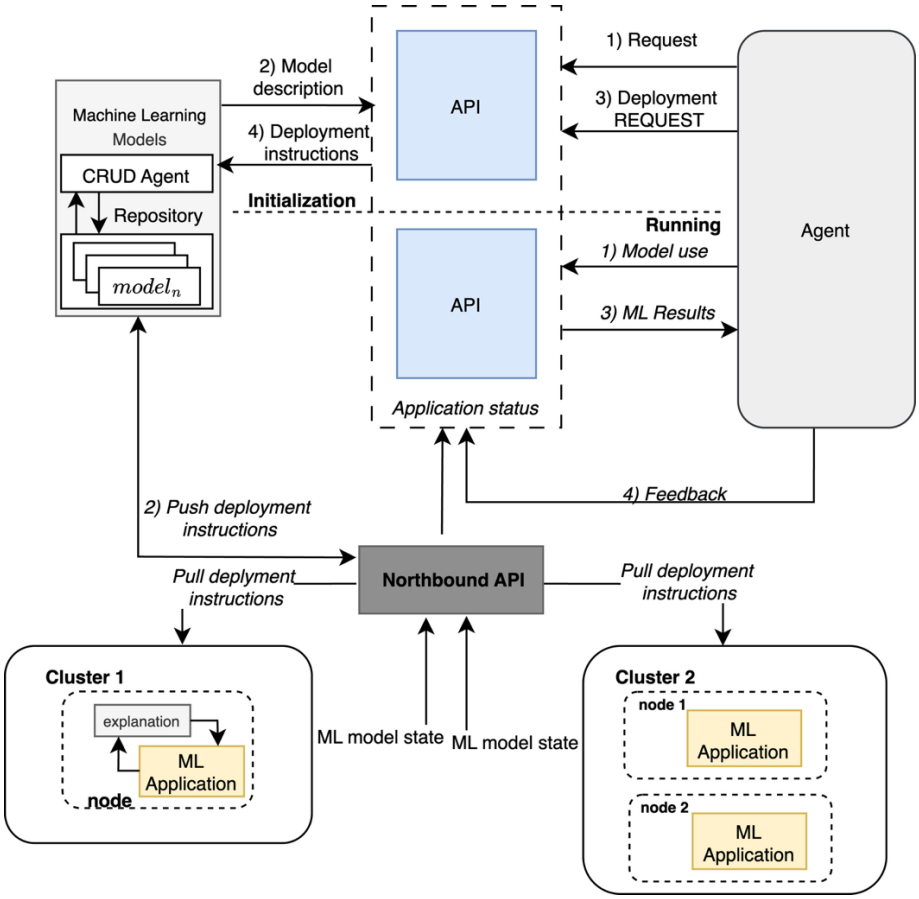


Figure 13.2: Process flow for ML interaction with API for training, deployment and monitoring

These endpoints are described in Table 13.1, and Figure 13.3 summarises the data flow of the endpoints.

Table 13.1: ML initialization endpoints

Action	HTTP Verb	URL Path	Description	Input	Response
Read	GET	/api/v1/model/	Read a collection of ML models available in the repository	None	List of all available ML model descriptions
Create	POST	/api/v1/model/	Add a new ML model to the repository	Model description	OK-
Read	GET	/api/v1/model/<model_id>	Read a particular ML model	String model ID	Model description

The second stage of the cycle is the deployment phase. After the agent acknowledges the deployment of an ML model, the request is pushed to a queue accessible by the continuum agent (a Redis queue being a potential implementation). Based on the resource requirements of the ML model to be deployed, a worker node(s) will be initialized. The resource negotiation is handled by the North and Southbound API. The MLConnector does not handle such resource allocation but can interact with other parts of the system (e.g. Northbound API) to make the required resources available.

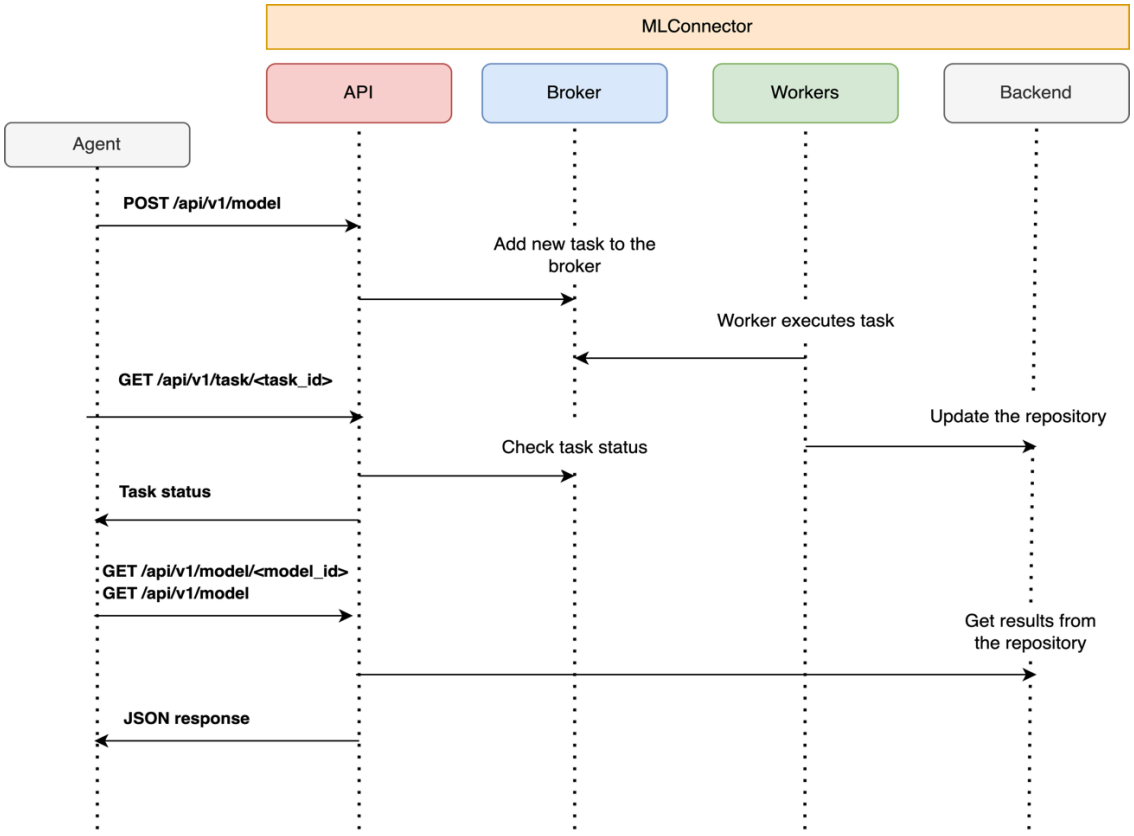


Figure 13.3: ML initialization data flow

This part of the system has the following actions.

1. **Model use**. Here, the agent interacts with the deployed API, for example, to make inferences or predictions. Input data will be required for this API call.
2. **ML results**. Here, the API returns prediction results. Based on these, the agent can request an explanation. The types and nature of explanations are defined in the next section.
3. **Reward**. Based on the return results and action performed, a reward will be returned to API to train the RL agent to improve its performance.

The API endpoints for this stage are defined in Table 13.2, and Figure 13.4 summarises the data flow of the endpoints.

Table 13.2: ML deployment endpoints

Action	HTTP Verb	URL Path	Description	Input	Response
Create	POST	/api/v1/deployment/	Request to deploy a new ML model	Model description	List of matching models
Create	POST	/api/v1/ deployment/ <model_id>	ACK a deployment request for an ML model	Model ID	Deployment ID
Read	GET	/api/v1/deployment/	Get an inference	Input data	Model output

		<inference_input>			
Read	GET	/api/v1/ deployment / < deployment _id>	Read a particular ML model deployment	Deployment ID	Deployment description
Read	GET	/api/v1/deployment/ <explanation_des>	Get the explanation on the deployed model	Inference input, Target var. level	Graph (see description below)

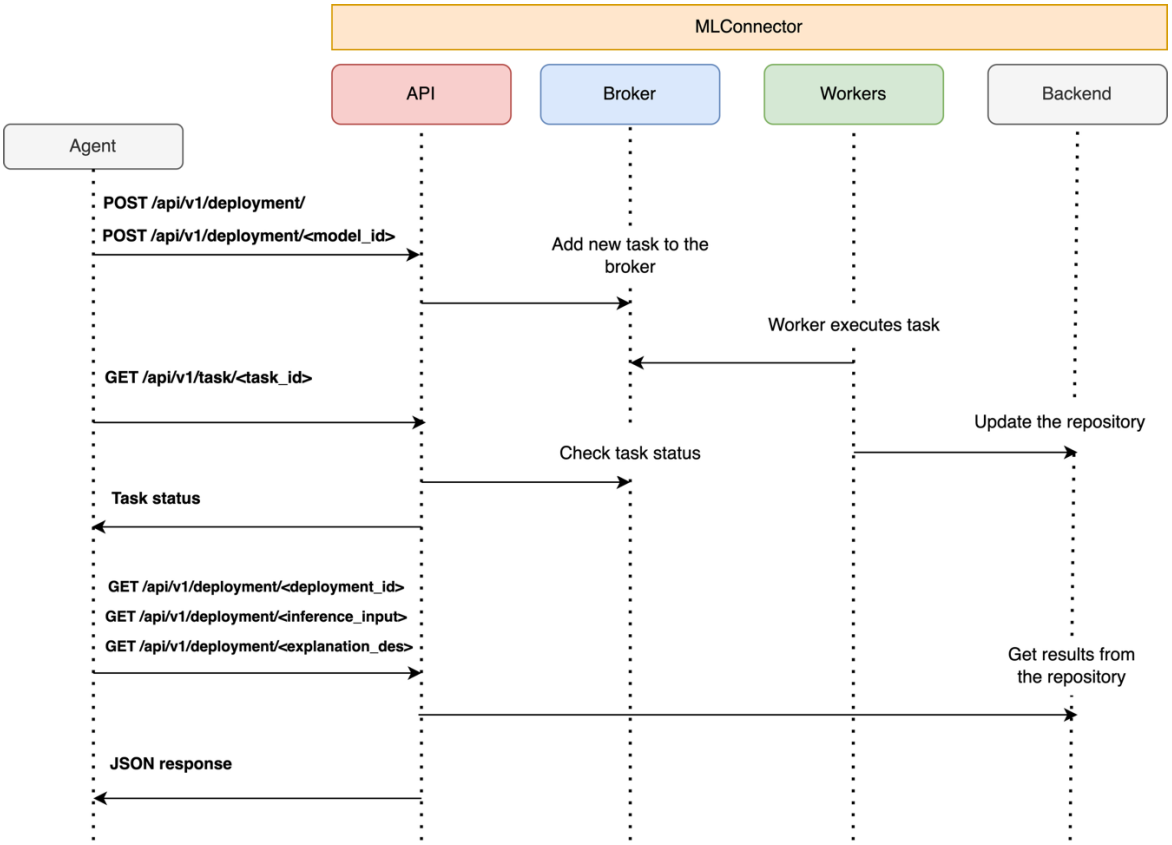


Figure 13.4: ML deployment endpoint data flow

Drift monitoring

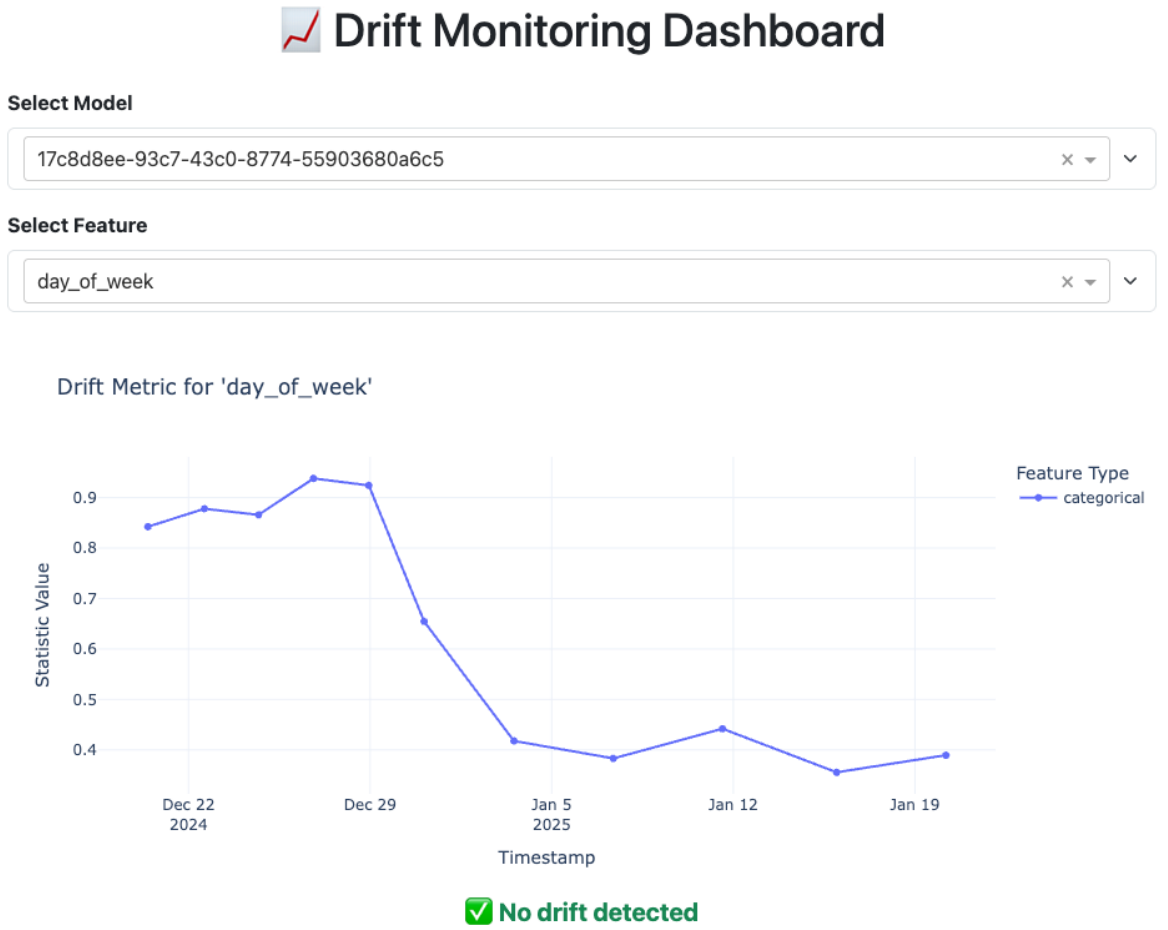
The MLConnector is also responsible for monitoring deployed machine learning model for performance. When machine learning models are deployed in real-world environments, the distribution of input data can evolve over time. This is known as data drift. Detecting and addressing data drift is important to maintaining model performance. During model registration, the user can enrol a model for drift detection with the MLConnector and specify a method to use. The MLConnector support three common methods.

1. Mean Shift
   It compares a numeric test dataset against a numeric training dataset, and produces a P-Value measuring the probability that the test data came from the same numeric distribution as the training data.
2. FourierMMD (Maximum Mean Discrepancy)
   Is a kernel-based method that measures the distance between two distributions by comparing their embeddings in a high-dimensional feature space. Uses characteristic kernels (e.g., Gaussian) to map data distributionsComputes a test statistic that captures both location and shape differences.
3. Kolmogorov–Smirnov Test (KS Test)

The KS test is a non-parametric test that compares the empirical cumulative distribution functions (ECDFs) of two datasets.

13.1.3 Usage

To enable drift on a model, the agent has to only update the drift property of the give ML model and specifying the method to use. The MLConnector then sets up a background service that monitors all the models. The service runs once every 24 hours. The results can then be view from the drift dashboard. Figure 13.5 show an example of one of the features of the regression models over a 2-month period.



Timestamp	Feature	Type	Statistic	P-Value	Drift Detected
2024-12-20 10:24:00	size	numerical	0.00033121	0.999995148	False
2024-12-20 10:24:00	download_time_ms	numerical	0.00065948	0.999995148	False
2024-12-20 10:24:00	hour	numerical	0.00733468	0.999995148	False
2024-12-20 10:24:00	minute	numerical	0.00500349	0.999995148	False

Figure 13.5: Drift monitoring dashboard

Explanations response designs

To understand how the API design works for explanation generation in the system, we briefly explain the design logic and our approach towards making the MLSysOps provide explanations. Based on this approach, we designed the declarative descriptions of the API and the relevant calls we described in Section 13.1. The latter returns the tuples of information necessary to create the explanations. We make a distinction between

explanations at the node level and higher levels (continuum and cluster) as the targets of the explanations differ. The Figure 13.6 the general flow of how different explanation can be retrieved by the agents at different levels.

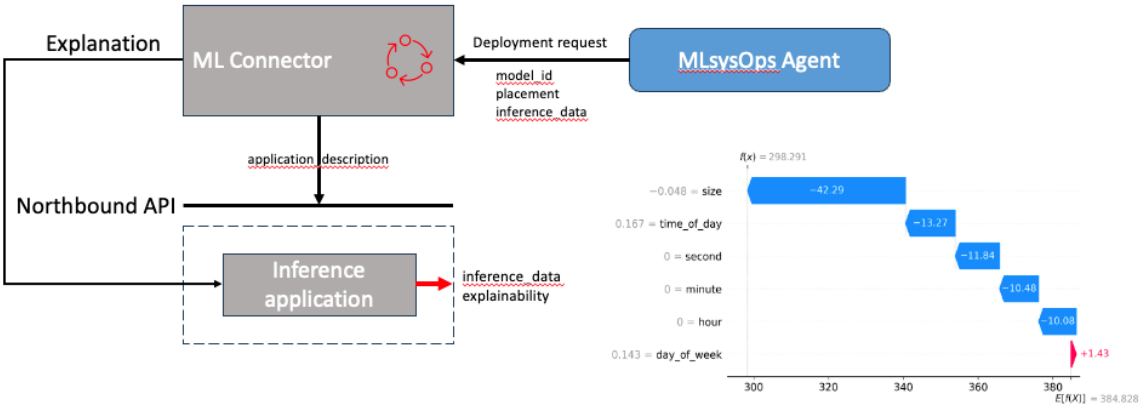


Figure 13.6: MLConnector explanations flow

Explanations at the node level have a very narrow scope of view, so local explanation techniques seem to be better suited here. Using Shapley values[51] for feature importance is a popular approach that can generate both local and global explanations regarding feature importance. We can use them to explain which parts of the input were most important for the action taken by an ML model at a particular time but also across many actions over a time window. SHAP is a well-established implementation of Shapley values for ML models that is easy to incorporate early on and offers many options, but more complex explanation approaches will be explored later. We plan to provide three different explanation categories that can be returned using the SHAP[52] explanation library.

1. Low level of detail (Two graphs).

   Force plot. This plot displays the importance of each feature (based on the Shapley values) in the model's output for the specific data instance. The importance of each feature can be inferred approximately, and the colours indicate a positive or negative influence on the model's output. (See Figure 13.7)

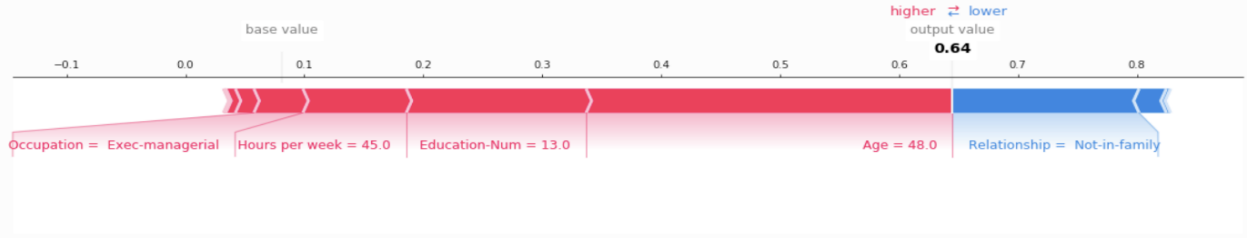


Figure 13.7: Force plot example from SHAP documentation[53]

Local bar plot. The plot shows the calculated Shapley Values in a bar plot from the most important feature to the least. The colours indicate a positive or negative contribution towards the model's output. (See Figure 13.8).

[51] Zou, J., & Petrosian, O. (2020). Explainable AI: Using Shapley value to explain complex anomaly detection ML-based systems. In *Machine learning and artificial intelligence* (pp. 152-164). IOS Press

[52] Lundberg, Scott M., and Su-In Lee. "A unified approach to interpreting model predictions." Advances in neural information processing systems 30 (2017).

[53] https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/decision_plot.html

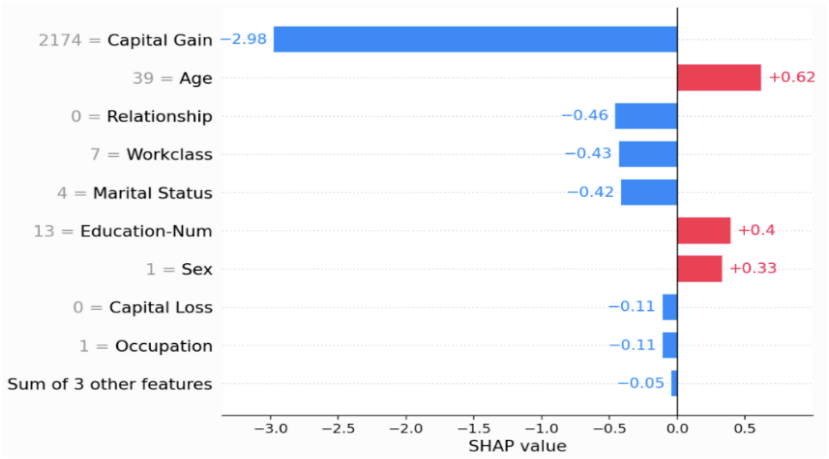


Figure 13.8: Local bar plot example from SHAP documentation[54]

2. Medium level of detail (Three graphs). It has the same graphs as the previous level plus an extra one, which is the Decision plot. It contains a similar type of information to the force plot but in greater detail. It gradually builds the model's output from the least impactful features to the more important ones (based on the Shapley values calculated). The actual Shapley values are not directly displayed, but the value of each feature is more visible. (See Figure 13.9).

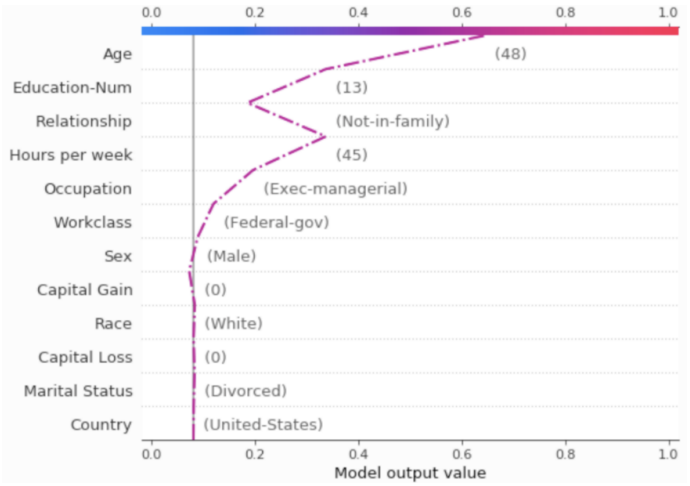


Figure 13.9: Decision plot example from SHAP documentation[55]

3. High level of detail (Four graphs). It has the same graphs as the previous level, plus the Waterfall plot. This plot combines the information provided by both the Decision Plot and the Local Bar plot. Since it has more information than the other graphs, it can be harder to read. (See Figure 13.10).

[54] https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/bar.html
[55] https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/decision_plot.html

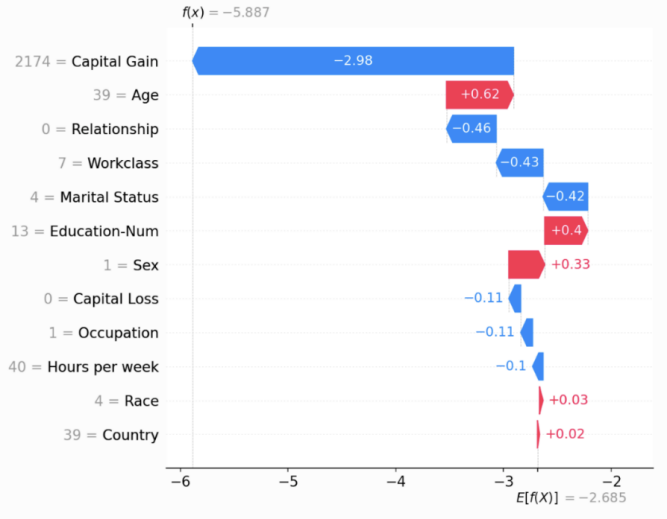


Figure 13.10: Waterfall plot example from SHAP documentation[56]

We plan to explore more techniques that may better suit explanation generation at higher levels. Even though Node-level explanations can stand on their own, we can use them as the building blocks for explanations at higher system levels. Explanations at cluster level and continuum level will follow two approaches based on who the viewer is (admin vs user) and what their interests are (system performance vs app performance).

13.1.4 User explanations at higher levels:

MLSysOps users will almost exclusively be interested in their application performance and health status, so the explanations should be tailor-made to satisfy these needs. This requires capturing node-level information for each application component and aggregating or combining them to application-wide explanations that focus on application-relevant parameters or performance metrics. This area is very sparsely explored, and currently, no well-established explanation method can successfully do that. We plan to explore extending the capabilities of Shapley Values to achieve that, as their mathematical properties are very desirable, but at the same time, we might find the need to create novel explanation methods that will be able to fit with the intricacies of an ML-based operated edge-cloud system.

13.1.5 Admin explanations at higher levels:

MLSysOps admins have many interests, which create a lot of different approaches towards explanations.

On the one hand, the admin needs to have a full view of the status of every application running on the system. The simplest thing we can do here is to provide an aggregated view of the explanations provided to every application user on the system. However, we might need to alter or change these explanations, or their level of detail at the very least, so that they are more aligned towards the interests of the admin. Keeping applications happy is of great interest to the admin, so when one of them is not, an explanation should be provided.

On the other hand, the admin is equally interested in the performance of the system itself. They need to have a view of how successfully the system goals are satisfied, and when not, an explanation should reveal what is wrong. These techniques need to consider the goals of each cluster agent and the goals of the continuum agent so that the resulting explanations are produced through these lenses. Such goals include resource utilization and energy consumption, green energy utilization, and so on. Having an ML-based operated edge-cloud system that can have different goals at different parts is a difficult task on its own but providing goal-relevant explanations on how such a system operates is even more challenging. No method currently exists that can do precisely that, so we first plan to adapt and extend some popular explanation techniques to facilitate these desired properties

[56] https://shap.readthedocs.io/en/latest/example_notebooks/api_examples/plots/waterfall.html

but also explore the possibility of creating new techniques that will better fit in this context. Our understanding of what these explanations will look like will be more concrete as the agent and ML implementation matures.

For both of those cases, we will have explanation generation on-demand for the time being, whether this is invoked by a user or the admin, to minimize the computational overhead from explanation generation.

14 Conclusion

In this document, we provide an AI-driven approach to the challenges posed by modern data-centric environments. The MLSysOps ML framework aids resource management and application orchestration across the computing continuum. By integrating multiple ML models and agents, MLSysOps ensures dynamic resource allocation, system monitoring, and performance optimization. This framework is crucial for maintaining smooth operation of software applications within containerized environments, especially in the context of 5G infrastructure and large-scale ML model training. Its modular design ensures interoperability and prevents vendor lock-in, while its AI-driven mechanisms enable efficient and autonomous system management. The framework's detailed components, from dynamic storage management to anomaly detection and ML model retraining, demonstrate a robust approach to enhancing the functionality and efficiency of edge and cloud computing ecosystems.

END OF DOCUMENT