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

Contract Number 101092912

# D3.3 Final Version of AI-ready MLSysOps Framework

| | |
|---|---|
| **Lead Partner** | NUBIS |
| **Contributing Partners** | UNICAL, NTT, UTH, CC, NUBIS, NVIDIA, INRIA, TUD |
| **Owner / Main Author** | Anastassios Nanos, Christos Panagiotou (NUBIS) |
| **Contributing Authors** | UNICAL (Raffaele Gravina, Marco Loaiza, Giancarlo Fortino)<br>NTT (Massimiliano Rossi, Kazem Eradatmand)<br>UTH (Alexandros Patras, Foivos Pournaropoulos, Nikolaos Bellas, Christos Antonopoulos, Spyros Lalis)<br>CC (Marcell Fehér)<br>NUBIS (Charalampos Mainas, Georgios Ntoutsos, Maria Goutha, Panagiotis Mavrikos, Konstantinos Papazafeiropoulos, Maria Rafaela Gkeka)<br>NVIDIA (Dimitris Syrivelis)<br>INRIA (Aya Moheddine, Jiali Xu, Valeria Loscri)<br>FhP-AICOS (Carlos Resende, João Oliveira, Filipe Sousa, Waldir Junior, José Costa)<br>TUD (Kaitai Liang) |
| **Reviewers** | Carlos Resende (FhP-AICOS), Christos Antonopoulos (UTH) |
| **Contractual Delivery Date** | M27 (31 March 2025) |
| **Actual Delivery Date** | 18/04/2025 |
| **Version** | 1.0 |
| **Dissemination Level** | Public |

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 | Outline by NUBIS. |
| 0.2 | Contributions collected by partners |
| 0.3 | Refinements and consolidation of contributions |
| 0.4 | Annotated text with comments from first round of reviews |
| 0.5 | Addressed comments after reviewing the document by the designated partners |
| 0.6 | Add sections from internal deliverables for clarification of concepts and mechanisms used in the project. |
| 0.7 | Annotated text with second round of reviews. |
| 0.8 | Final draft, with all review comments addressed. |
| 0.9 | Polish text, another iteration of reviews and comments to produce the final version. |
| 1.0 | Final version |

# Table of Contents

## List of Figures

## List of Tables

# Summary

Deliverable D3.3 represents the third and final iteration of the advanced mechanisms developed within the MLSysOps project, completing the evolution initiated in D3.1 and further developed in D3.2. This final iteration consolidates the enhancements and comprehensive developments of mechanisms designed for efficient resource allocation, adaptive configuration, and autonomous management within the heterogeneous cloud-edge continuum. D3.3 encapsulates the maturity reached by the MLSysOps AI-ready framework, highlighting the final advancements in telemetry infrastructure, computational resource management, adaptive storage distribution and networking along with the seamless deployment and orchestration across diverse computing infrastructures.

A first integrated version of the MLSysOps framework, including core components, supporting libraries, and mechanisms described throughout this deliverable, will be publicly released via the project's official open-source repository. This repository will also include detailed documentation, tutorials, and usage examples to support adoption and experimentation by the community. It is expected to be officially launched and accessible to the public by June 2025 (M30). Furthermore, slight differences may exist between the implementations described in this deliverable and those published in the repository, as the framework is subject to continuous refinement and enhancement. Therefore, the open-source repository should be considered the primary reference for the most up-to-date and operational version of the MLSysOps framework.

## Abbreviations

| | |
|---|---|
| ABI | Application Binary Interface |
| ABM | Agent Based Model |
| AI | Artificial Intelligence |
| API | Application Programming Interface |
| APN | Access Point Name |
| CID | Context Identifier |
| CPU | Central Processing Unit |
| CRI | Container Runtime Interface |
| DFX | Dynamic Function eXchang |
| DRAM | Dynamic random access memory |
| DVFS | Dynamic Voltage and Frequency Scaling |
| FAAS | Function As A Service |
| GPU | Graphics Processor Unit |
| HPC | High Performance Computing |
| IAAS | Infrastructure As A Service |
| IETF | Internet Engineering Task Force |
| JADE | Java Agent DEvelopment Framework |
| JID | Jabber Identifier |
| JVM | Java Virtual Machine |
| LLMs | Large Language Models |
| LwM2M | Lightweight Machine to Machine |
| ML | Machine Learning |
| MPSoC | MultiProcessor System On Chip |
| MQTT | Message Queuing Telemetry Transport |
| MSR | Model Specific Registers |

| | |
|---|---|
| OCI | Open Container Initiative |
| OpAMP | Open Agent Management Protocol |
| OS | Operating System |
| OTEL | OpenTelemetry |
| PAAS | Platform As A Service |
| PDU | Protocol Data Unit |
| PID | process identifier |
| PMU | Performance Monitor Unit |
| RAPL | Running Average Power Limit |
| RG | Requirements Group |
| RTOS | Real Time Operating System |
| SAAS | Software As A Service |
| SDF | Semantic Definition Format |
| SMB | Small-Medium Business |
| SMF | Session Management Function |
| UE | User Equipment |
| UPF | User Plane Function |
| vAccelRT | vAccel Runtime |
| VIP | Virtual IP |
| VM | Virtual Machine |
| VMM | Virtual Machine Monitor |
| WASM | Web Assembly |
| XMPP | Extensible Messaging and Presence Protocol |

# 1 Introduction

In the evolving landscape of cloud and edge computing, MLSysOps emerges as a pivotal endeavour, aiming to redefine the efficiency and adaptability of these technologies. D3.3 is the outcome of Work Package 3 and has been refined and finalized through an iteration of 3 deliverables (D3.1, D3.2). This series of deliverables focuses on the implementations of the basic infrastructure (telemetry and mechanisms), which feed the required data to the MLSysOps ML models (as realized in WP4) and implement the generated decisions, harnessing the power of artificial intelligence (AI) and machine learning (ML) to transform resource management in cloud and edge infrastructures.

## 1.1 System Scope and Architecture

The MLSysOps framework operates in the context of a heterogeneous, multi-layered computing continuum, ranging from centralized cloud infrastructures to resource-constrained far-edge devices. The objective of the framework is to enable autonomic, explainable, and adaptive system management by leveraging artificial intelligence, with minimal human intervention. The design of MLSysOps is guided by a system model that introduces the concept of a slice — a logical grouping of computing, networking, and storage resources across the continuum that is managed as a unit. Each slice is governed by its own deployment of the MLSysOps control plane and encompasses physical or virtualized resources at different layers.

The continuum model, shown in Figure 1, defines four layers:

- Cloud Infrastructure, where high-performance datacenters offer scalable resources for compute- and storage-intensive operations;
- Edge Infrastructure, comprising regional or institutional datacenters positioned closer to data sources and users;
- Smart-Edge, representing computational nodes with embedded intelligence (e.g., gateways, mobile nodes, or industrial controllers);
- and Far-Edge, which consists of highly constrained embedded devices (e.g., sensors, MCUs, UAVs) typically located at the network periphery.

This hierarchical abstraction enables MLSysOps to treat highly diverse resource pools as a cohesive infrastructure.

To manage the complexity of such environments, MLSysOps introduces a hierarchical agent-based architecture composed of three levels:

- Node Agents reside on individual nodes and expose configuration interfaces, monitor resource usage, and provide direct access to telemetry.
- Cluster Agents coordinate groups of nodes, aggregate telemetry, and issue deployment decisions or adaptation instructions.
- The Continuum Agent sits at the top level, interfacing with external stakeholders (via northbound APIs), receiving high-level intents and application descriptors, and coordinating decision-making across slices.

Each layer operates a Monitor–Analyze–Plan–Execute (MAPE) control loop, enabling autonomous adaptation based on local and global telemetry, system optimization targets, and ML-driven policies. Importantly, this architecture separates management logic from resource control, allowing for modular evolution and system introspection.

Figure 1 provides a visual abstraction of the MLSysOps domain model, highlighting the cloud-to-far-edge continuum layers, representative resource types, and the placement of MLSysOps components. This model underpins all subsequent design and prototyping activities described in this deliverable.

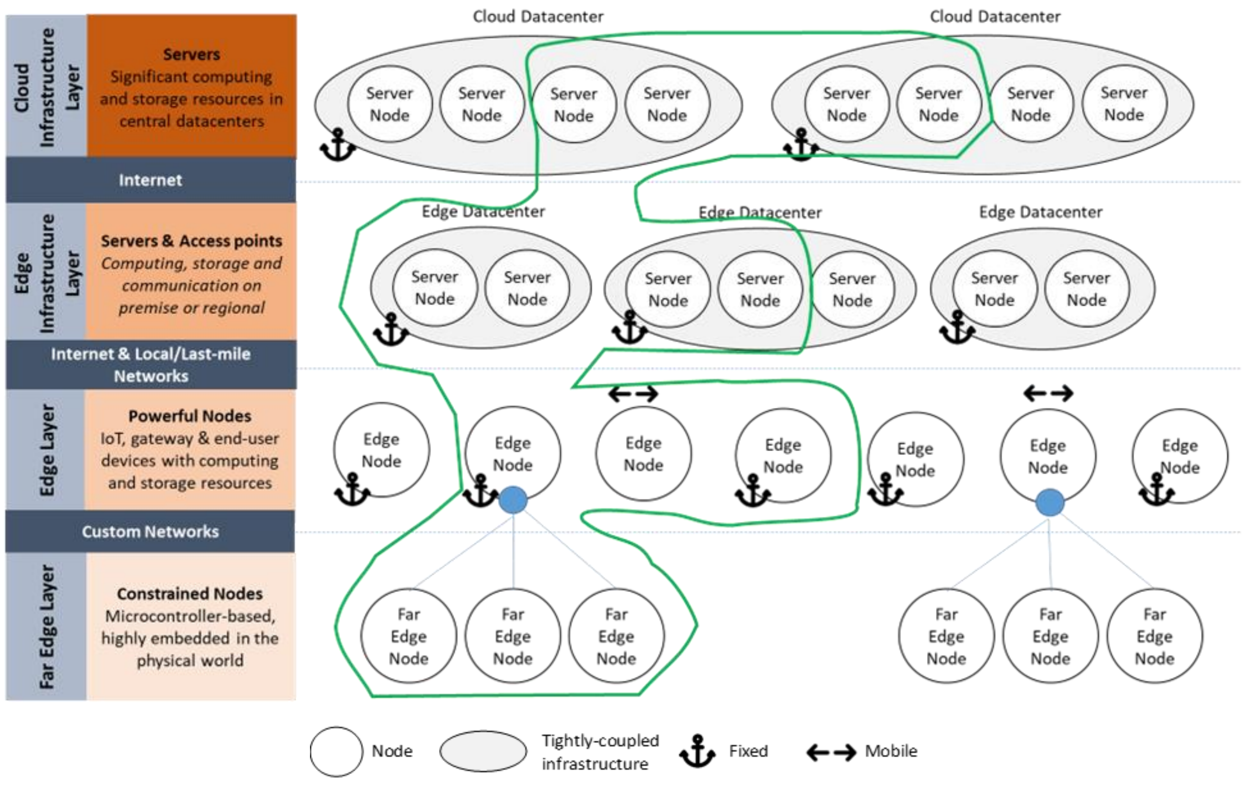

Cloud Infrastructure Layer
Servers
Significant computing and storage resources in central datacenters
Internet
Edge Infrastructure Layer
Servers & Access points
Computing, storage and communication on premise or regional
Internet & Local/Last-mile Networks
Edge Layer
Powerful Nodes
IoT, gateway & end-user devices with computing and storage resources
Custom Networks
Far Edge Layer
Constrained Nodes
Microcontroller-based, highly embedded in the physical world
Cloud Datacenter
Server Node
Edge Datacenter
Edge Node
Far Edge Node
Node
Tightly-coupled infrastructure
Fixed
Mobile

**Figure 1 Main layers of the continuum system, annotated with a slice, spanning the continuum**

D3.3 outlines the development activities involving advanced mechanisms for allocating, provisioning, and adapting resources within cloud and edge computing environments. This report also examines the underlying mechanisms of the AI-ready MLSysOps framework, as depicted in Figure 2, which illustrates the MLSysOps architecture. Components focused on by WP3 include the telemetry system responsible for data collection and processing, dynamic resource management mechanisms that optimize computational, storage, and network resources, and the deployment and orchestration mechanisms that ensure efficient application management across cloud and edge environments, along with the agents' component.

The MLSysOps agents, supported by ML models, analyse, predict, and optimize resource usage patterns and overall system performance by allocating, monitoring and configuring the different resources of the underlying layers via the mechanisms that are implemented in the context of WP3 and manifested in the current deliverable. This integration is a collaborative effort that draws on the diverse expertise of project partners, each contributing unique insights and solutions to the multifaceted challenges of cloud and edge computing. This collaborative approach is complemented by an iterative development process characterized by continuous testing and feedback loops. Such a process ensures that the mechanisms developed are not only effective in their current context but are also scalable and adaptable to future technological advancements and operational needs.

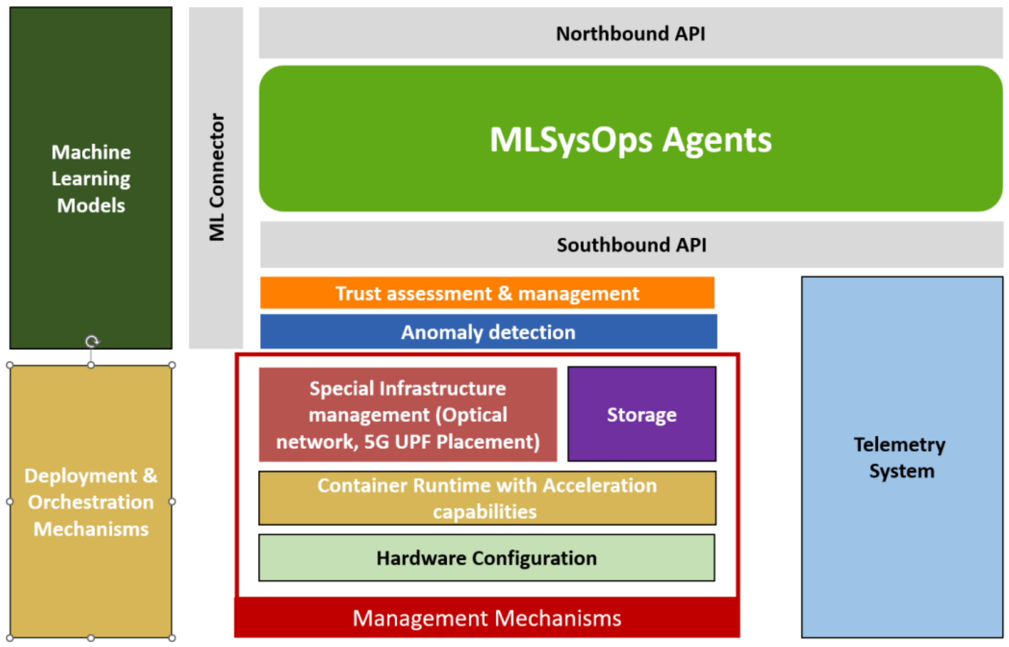

Machine Learning Models
ML Connector
Northbound API
MLSysOps Agents
Southbound API
Trust assessment & management
Anomaly detection
Special Infrastructure management (Optical network, 5G UPF Placement)
Storage
Container Runtime with Acceleration capabilities
Hardware Configuration
Management Mechanisms
Deployment & Orchestration Mechanisms
Telemetry System

**Figure 2 MLSysOps Architecture.**

## 1.2 Background

Key points of D3.3 deliverable are five fundamental concepts. AI-ready resource management signifies the capability of computing systems to autonomously manage and optimize resources using AI and ML models. The cloud-edge continuum represents the seamless integration of cloud computing with edge computing devices, enabling distributed and efficient processing. Energy efficiency is a cornerstone, emphasizing the importance of maximizing performance while minimizing energy consumption, a critical aspect of sustainable computing practices. Resource allocation and provisioning are processes involving the strategic distribution and management of computing, storage, and networking resources within an IT infrastructure. Lastly, adaptation in computing infrastructures highlights the ability of systems to dynamically adjust resource usage and operational parameters in response to changing conditions and demands.

In that context, Figure 3 depicts a comprehensive illustration of the MLSysOps hierarchical agent system's placement and its interactions with two other fundamental subsystems: container orchestration and telemetry. This agent hierarchy is structured in line with the orchestration architecture, and it is logically divided into three tiers. The communication among the three subsystems (agents, container orchestration, and telemetry) is facilitated through designated interfaces at each tier. Moreover, the agent system engages with the continuum level's system agents and integrates plug-in configuration policies that can use ML models at all levels. At every level, agents utilize mechanism plugins to implement commands for adjusting available configuration and execution mode options.

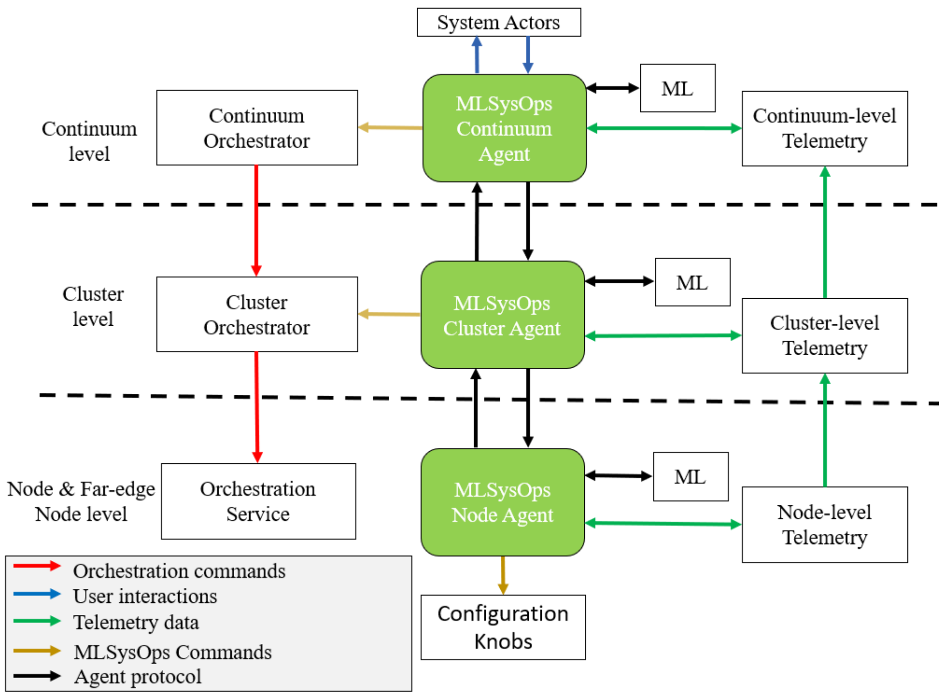


**Figure 3: MLSysOps high level hierarchical architecture.**

Node-level agents' interface with local telemetry systems and expose configuration knobs. Cluster-level agents coordinate resource allocation decisions across groups of nodes. At the top level, the continuum agent handles global orchestration, provides APIs to external actors, and aggregates telemetry data. ML-driven decisions can be made at every layer, using information for the respective layer. This layered approach facilitates scalability and separation of concerns while supporting collaboration across orchestration, telemetry, and ML systems. The agent infrastructure interacts through three distinct types of interfaces. The Northbound API provides access to application developers and system administrators. The Southbound API interfaces with the underlying telemetry collection and configuration mechanisms. The ML Connector allows ML models to be plugged into the framework and invoked for training, prediction, and explanation tasks.

The telemetry subsystem is built upon the OpenTelemetry specification and is responsible for collecting and processing metrics, logs, and traces. These are abstracted into hierarchical telemetry streams that feed the decision logic of the agents and the ML models. Data collection happens at the node level, where individual collectors expose metrics either in raw or aggregated formats. These are processed through transformation pipelines and propagated to cluster and continuum levels for higher-level aggregation and analysis.

Application deployment and orchestration are driven by declarative descriptions submitted by application developers and administrators. These descriptions capture the application's structure, resource requirements, and quality-of-service objectives. Deployment is handled through standard container orchestration tools, which are extended by the MLSysOps framework to support advanced placement decisions and runtime adaptation. For far-edge deployments, the framework introduces a proxy-based architecture involving embServe on constrained devices and a virtual orchestrator service running inside containerized environments. This approach allows resource-constrained devices to be seamlessly integrated into the same orchestration and telemetry flows as more capable edge and cloud nodes.

The object storage infrastructure builds upon and extends SkyFlok, a secure and distributed storage system. In MLSysOps, this infrastructure supports adaptive reconfiguration of bucket policies based on real-time telemetry and application usage patterns. The storage system exposes telemetry data regarding latency, bandwidth, and access frequency, enabling agents and ML models to optimize redundancy and placement decisions without disrupting ongoing operations.

The framework also includes specialized subsystems for anomaly detection and trust assessment. These modules analyze telemetry data to identify attacks or malfunctions and classify anomalies using ML models. Their outputs are exposed through the telemetry interface and used by higher-level agents to trigger remediation strategies or adapt orchestration plans. Trust levels for nodes are computed using a combination of identity, behaviour, and capability metrics, forming a reputation-based model that influences agent decision-making.

ML models play a central role in enabling the autonomic operation of the framework. Each level of the agent hierarchy may employ one or more models, which are integrated via the ML Connector API. These models receive structured telemetry input and produce configuration decisions, which are interpreted and enacted by the agents. The framework supports reinforcement learning, continual learning, and federated learning scenarios. In addition, explainability mechanisms are integrated into the ML workflows to allow system administrators and application developers to understand and audit the decisions made by the models.

MLSysOps effectively manages operations by leveraging telemetry data collected from each level, which provides essential insights. This data, combined with machine learning models, enhances the decision-making process, aligning with both the application's objectives and the system's requirements. Actions based on these decisions are cascaded and refined from the top level downwards. The final status and outcomes of these decisions are then made accessible to system Actors. The design and functionality of the telemetry system are further explored in Section 2.

The source code of the implementations is managed and tracked by the private GitLab of the project [1]. In D3.3, all source code repositories are migrated to the respective Github organization [2], where the Open-source project will be hosted (see Section 1.5).

**DISCLAIMER:** *Implementations and examples presented in this document may evolve, and the most up-to-date versions will be maintained in the project's open-source repository.*

## 1.3 Requirements

To achieve the design and implementation of a robust system that manages to address its objectives, as briefly presented previously the identification, analysis, and documentation of the requirements is of paramount importance, as it ensures that all functional expectations, operational constraints, and system assumptions are clearly understood and systematically addressed throughout the development lifecycle. MLSysOps carried out the identification of requirements using a bottom-up approach, emphasizing the core management and configuration functionalities that the framework was expected to support. These encompass aspects such as the structured representation of system and application configurations, automated deployment and orchestration mechanisms, effective resource usage and management at the node level, storage control, and mechanisms for assessing and maintaining trust. Additionally, the framework addresses network management and security in edge environments, the integration and control of 5G networks, optical networking within datacenters, and strategies for energy-efficient, sustainable computing. Underpinning all these areas is the use of continual and explainable machine learning models, which guide and optimize the above functionalities. The identified requirements are grouped into 10 distinct sets based on their core functionality.

**Table 1 RG1: Structured system and application descriptions requirements**

| id | Title | Description |
|---|---|---|
| R1.1 | Structured System Description | Describe system infrastructure declaratively, including nodes, resources, connectivity, and optimization targets |
| R1.2 | Structured Application Description | Describe application declaratively, including components, interactions, and QoS metrics |

**Table 2 RG2: Application deployment and orchestration requirements**

| id | Title | Description |
|---|---|---|
| R2.1 | Continuum Deployment | Deploy applications across all continuum layers with support for proxies at Smart-Edge |
| R2.2 | Flexible Connectivity | Support interactions via alternative connectivity paths and technologies |
| R2.3 | Adaptive Orchestration | Produce and adapt orchestration plans based on QoS targets and optimization goals |

**Table 3 RG3: Node level resource usage and management requirements**

| id | Title | Description |
|---|---|---|
| R3.1 | Node Configuration | Support physical parameter configuration (voltage, frequency, power, etc.) |
| R3.2 | FaaS Acceleration | Support dynamic hardware acceleration use in FaaS style |
| R3.3 | Concurrent ML Acceleration | Enable ML components to share acceleration resources with applications |
| R3.4 | Runtime Configuration | Adapt node and app configurations to meet QoS and system objectives |
| R3.5 | Telemetry Abstraction | Support telemetry collection despite heterogeneity |
| R3.6 | Far-Edge Virtualization | Provide virtualization capabilities for far-edge nodes |

**Table 4 RG4: Storage requirements**

| id | Title | Description |
|---|---|---|
| R4.1 | S3-Compatible Storage | Expose S3 interface across heterogeneous cloud-edge storage |
| R4.2 | Storage Monitoring | Monitor availability and performance of storage resources |

| | | |
|---|---|---|
| R4.3 | Access Pattern Monitoring | Track data access and modification patterns |
| R4.4 | Adaptive Storage Policies | Adjust storage choices and data encoding using ML |

**Table 5 RG5: Trust requirements**

| id | Title | Description |
|---|---|---|
| R5.1 | Trust Evaluation | Compute trust using reputation and behaviour analytics |
| R5.2 | Secure Communication | Encrypt and protect communication across layers |
| R5.3 | Trust Adjustment | Dynamically update trust levels and privileges |
| R5.4 | Trust-based Deployment | Use trust scores in resource allocation and deployment decisions |

**Table 6 RG6: Wireless network management and security at the edge requirements**

| id | Title | Description |
|---|---|---|
| R6.1 | QoS-Aware Connection Management | Adapt deployment based on QoS parameters from ML evaluation |
| R6.2 | Security-Aware Connection Management | Adapt deployment based on security parameters from ML evaluation |

**Table 7 RG7: Network management requirements**

| id | Title | Description |
|---|---|---|
| R7.1 | 5G UPF Automation | Automate deployment/configuration of UPF at edge |
| R7.2 | 5G Continuity | Ensure minimal downtime upon 5G core changes |
| R7.3 | 5G Deployment Metrics | Use 5G data to optimize UPF placement |
| R7.4 | Dynamic UPF Switching | Allow switching between centralized UPFs via ML |

**Table 8 RG8: Optical networking in the datacenter requirements**

| id | Title | Description |
|---|---|---|
| R8.1 | Optical Network Optimization | Generate ML-driven optical network configuration |
| R8.2 | Job Scheduling Optimization | Use ML to assign MPI ranks to minimize network use |

Table 9 RG9: Energy efficient and green computing in datacenters requirements

| id | Title | Description |
|---|---|---|
| R9.1 | Energy-Efficient Scheduling | ML-guided scheduling for QoS and energy efficiency |
| R9.2 | Node Energy Configuration | Configure nodes to balance QoS and energy use |
| R9.3 | Cross-DC Scheduling | ML-based inter-DC scheduling to maximize green energy/profit |

Table 10 RG10: Machine learning requirements

| id | Title | Description |
|---|---|---|
| R10.1 | ML Plug-and-Play | Plug in different ML models via abstraction layer |
| R10.2 | Scalable ML Architecture | Define scalable, layered responsibilities for ML models |
| R10.3 | Continual ML Training | Enable model drift detection and re-training |
| R10.4 | Resource-Aware Training | Schedule ML training with minimal interference |
| R10.5 | Explainable ML | Ensure ML models provide high fidelity and explainability |
| R10.6 | Manual Override | Allow disabling ML and reverting to manual/rule-based control |
| R10.7 | Telemetry Optimization | Use data-centric AI to reduce telemetry volume |
| R10.8 | HW-Agnostic ML Execution | Support ML execution on various hardware accelerators |

## 1.4 Background vs Foreground for orchestration mechanisms

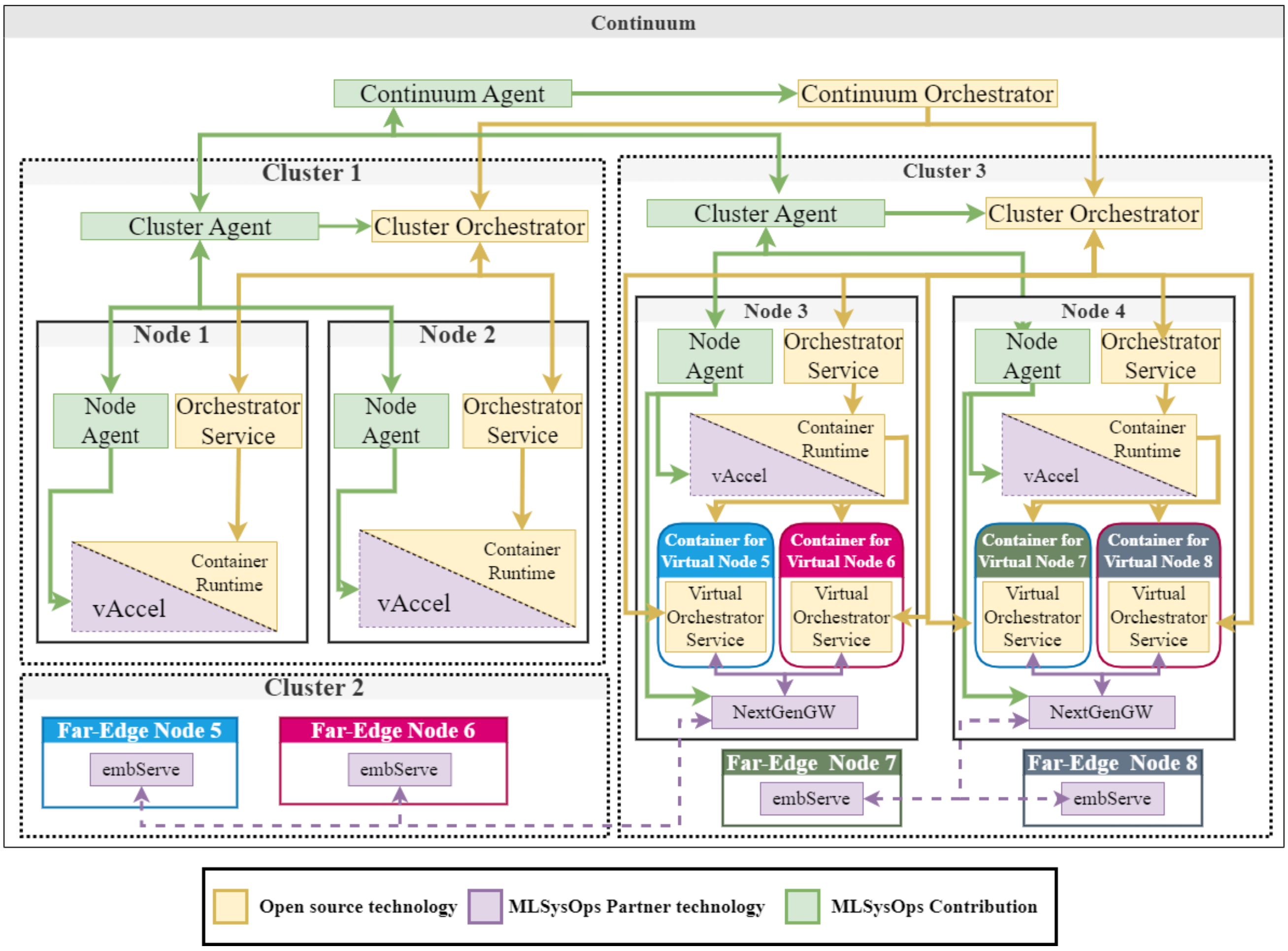


**Figure 4: MLSysOps Orchestration architecture.**

Figure 4 illustrates the layered orchestration model of the MLSysOps platform, structured across multiple clusters and far-edge nodes. The figure integrates both foundational (background) and innovative (foreground) components. Core infrastructure elements such as vAccel, Container Runtimes, Orchestrator Services, and embServe are leveraged from existing open-source and partner technologies. These ensure compatibility, performance, and modularity at each orchestration level.

Foreground contributions by MLSysOps are highlighted in green, including the development of the Continuum Agent, Continuum Orchestrator, and enhancements to Cluster and Node Agents. These enable a cross-layer orchestration paradigm across heterogeneous environments. Moreover, the concept of Virtual Nodes and their corresponding Virtual Orchestrator Services facilitates the efficient emulation and management of far-edge nodes within the cluster, significantly reducing latency and improving fault tolerance. This virtual-to-physical binding is further optimized through the NextGenGW and dynamic deployment of ML models to real far-edge devices, enhancing the responsiveness and adaptability of the system.

Additionally, enhancements to existing frameworks have been introduced to facilitate the deployment and lifecycle management of applications in the context of MLSysOps, as well as the execution mode of workloads, including the flexibility of adapting the execution to use hardware accelerators or not. Specifically, in the context of MLSysOps, we introduce the hybrid use of sandboxed container runtimes (such as kata-containers), alongside generic ones, and develop our own custom container runtime to support unikernel execution. Additionally, we enhance the vAccel framework to support a hinting mechanism for dynamically adapting the execution to the relevant hardware-specific plugin. More information about these mechanisms will be presented in the following sections.

## 1.5 Open Source Project

The implementation and prototyping activities described throughout this deliverable form the foundation of the first fully integrated version of the MLSysOps framework. This version includes a subset of the architecture's capabilities, such as key agent mechanisms, telemetry collectors, orchestration components, as well as ML-based policies and plugins that support specific use cases and continuum-layer configurations. These components are being progressively integrated and validated in relevant deployment environments, as part of the ongoing technical work under WP3 and WP4.

The results of the first fully integrated and tested release of the MLSysOps framework are not yet officially available to the public. This includes both the core software components and the corresponding documentation, tutorials, and usage examples. To ensure long-term usability, sustainability, and openness of the MLSysOps framework, the framework will be made publicly accessible via an open-source repository[1], and this release is planned for June 2025 (M30)[2]. The repository includes detailed guides for deploying and using the framework in representative scenarios, as well as instructions for extending it with custom plugins, telemetry sources, and orchestration policies. While the repository will include all essential components required to demonstrate and extend the framework, some modules described in this document might not be released as open-source due to intellectual property (IP) constraints, especially where third-party technologies or proprietary contributions are involved.

We emphasize this timeline to properly manage expectations regarding the current state of source code availability. As per the project's dissemination rules, this deliverable will be published on the MLSysOps website shortly after its submission to the European Commission. However, users should not expect the full framework to be immediately accessible or ready for use based solely on the content of this document. The official open-source release will be clearly communicated and promoted through the project's online channels (website, newsletter).

## 1.6 Outline

The rest of the document is structured as follows. Section 2 presents the telemetry component of MLSysOps architecture that is responsible for data collection (optionally combined with some pre-processing) from all the components of the system. Section 3 delves into the dynamic management of computational resources, allowing the optimization of processing capabilities while minimizing energy consumption. Section 4 focuses on the storage infrastructure, seeking to enhance adaptability and efficiency through intelligent data distribution and redundancy management. On the other hand, Section 5 focuses on the networking perspective, introducing the control mechanisms that pave the way for efficient and proactive network resource management. Further, Section 6 tackles application deployment and orchestration, implementing cutting-edge techniques for streamlined and efficient application operations across the cloud-edge continuum. Section 7 addresses the critical aspect of trust management and security, while Section 8 presents the agent architecture and technology along with the agent interaction, establishing robust mechanisms to ensure the integrity and reliability of operations within the infrastructure. Section 9 includes a detailed description of each component through

[1] https://github.com/mlsysops-eu/mlsysops-framework

[2] Originally planned for Sept 2025 (M33).

separate tables along with the git repository where the source code is maintained. The deliverable concludes in Section 10.

# 2 Telemetry System

The telemetry plane of MLSysOps collects the data necessary from all layers to drive the configuration decisions, potentially made using ML inference and continual training, with appropriate aggregation and abstraction towards the higher layers of the hierarchy. This section will give the technical insights of the MLSysOps telemetry system that supports the collection of performance metrics across MLSysOps resource management layers.

## 2.1 OpenTelemetry specification

The MLSysOps framework operates on different layers of the cloud-edge-far-edge continuum and manages highly heterogeneous systems and applications while simultaneously providing appropriate observability user interfaces, as illustrated in Figure 5. Given the diversity of tools and vendors involved, a vendor- and tool-agnostic protocol for collecting and transmitting telemetry data is essential.

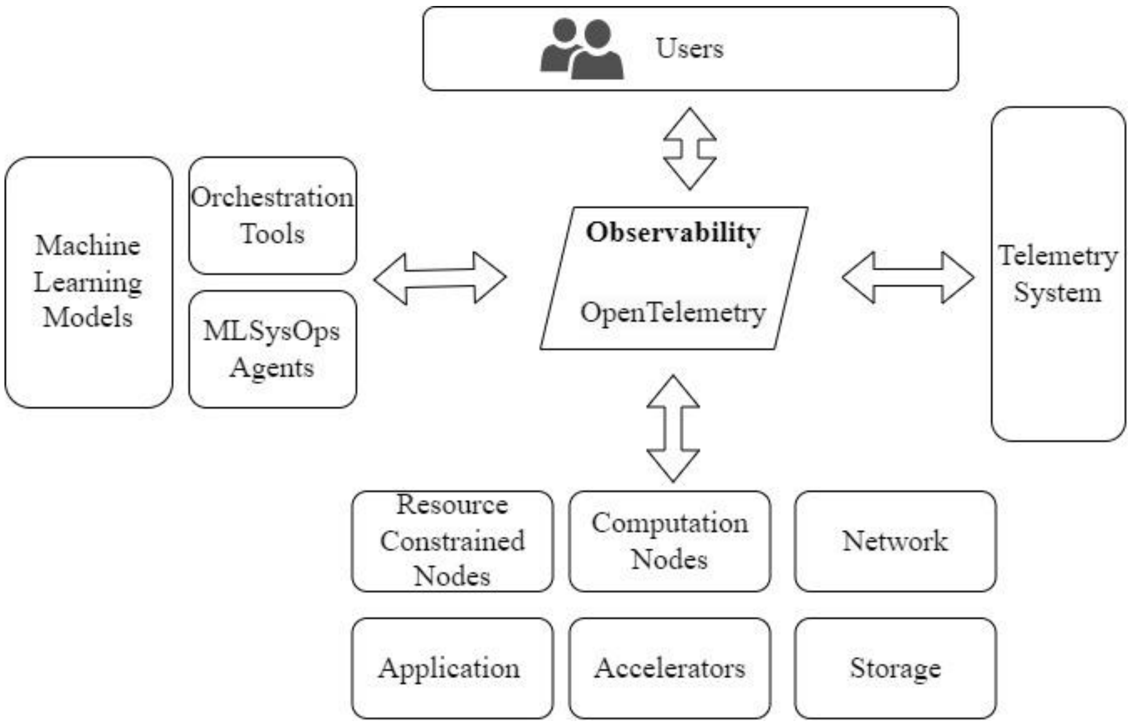


**Figure 5: High-level overview of the OpenTelemetry realisation in MLSysOps.**

OpenTelemetry [3], a well-defined open-source system, provides the foundation for MLSysOps observability capabilities. OpenTelemetry is an all-around observability framework, that handles all necessary signals (categories of telemetry data), such as traces, metrics, and logs. MLSysOps supports and uses all three signal categories. The basic signal that is exposed to the framework’s users are metrics, whereas logs and traces are used for debugging and profiling purposes.

## 2.2 Software components

OpenTelemetry offers the API as well as the software components that implement various telemetry functionalities, in the form of a Software Development Kit (SDK).

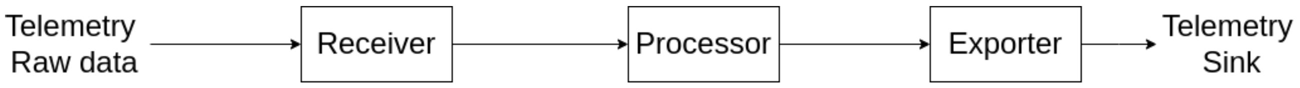


**Figure 6: Telemetry Pipeline.**

The central software component is the OpenTelemetry Collector (OTEL Collector), a vendor-agnostic implementation of the telemetry pipeline (as presented in Figure 6) that consists of three stages: i) receive, ii) process, and iii) export. This component is versatile and flexible, being able to receive telemetry data in multiple formats, process them in different ways, and export them to other systems. The OTEL Collectors can operate in two different modes: i) Agent and ii) Gateway mode. The main difference is that in Gateway mode, the collector receives telemetry data from other collectors (that, in turn, operate in Agent or Gateway mode). This makes the gateway a centralized aggregator for any underlying OTEL collectors.

The OTEL Collector [4] baseline implementation offers the minimum required functionality, which is a set of receivers and exporters that communicate using data conforming to the OpenTelemetry Data Specification, using either HTTP or gRPC protocols, as well as specific processing capabilities, like batching and memory control. The OpenTelemetry collector repository [4] includes multiple plugin implementations for all three stages of the telemetry pipeline, like Prometheus receivers & exporters, as well as a routing processor that can create different telemetry streams in the OTEL Collector pipeline. In MLSysOps, we fully leverage the available plugins to achieve the desired functionality for the OTEL Collectors at each level of the MLSysOps hierarchy: node, cluster, and continuum.

The OpenTelemetry specification defines a way to collect and transfer telemetry data without making any assumptions about the storage of the data. MLSysOps follows the same paradigm and does not make any assumptions for this matter, although, for the development of the framework, we use the Mimir metrics database [5], which is an open-source software that is suitable for the needs of our telemetry system. Mimir is deployed at the highest level (continuum), storing telemetry data and offering an API for flexible telemetry data querying through its proprietary PromQL interface,. PromQL [6] is quite versatile and powerful for time-series data, allowing different clients (such as ML models) to easily consume the data they need, using further aggregation and transformation functions. On top of the Mimir database, MLSysOps uses Grafana [7] for data visualization, Loki [8] for the logs messages database, and Grafana Tempo [9] for trace storage. All these components belong to the same ecosystem and work seamlessly with each other without a need for further configuration. We leverage the PromQL for the MLSysOps Telemetry API at the higher levels, implementing the necessary functionality for providing telemetry data to other entities of the system, and the Prometheus metric format for providing telemetry data at the same level.

The deployment of the OTEL Collectors in each node is performed using the appropriate containers, and the orchestration is done through the same orchestrator as the ones used for application components. This simplifies the management and the connectivity between the application components and the OTEL Collectors. The deployment configuration is done transparently and automatically by the MLSysOps framework, which is responsible for the initial deployment and configuration of the OTEL Collector pods on every node and layer, as well as the dynamic reconfiguration of the collectors at runtime.

## 2.3 Node level telemetry

On each node, the OTEL Collector operates in Agent mode. As Figure 7 illustrates, it receives data from any entity that needs to emit telemetry data into the telemetry system. This is done either through the available interfaces, as they are discussed in Section 2.7, or through the appropriate configuration of the OTEL Collector enabling the desired receiver. It then periodically pushes telemetry data to the OTEL Collector, which operates in Gateway mode at the cluster level. OTEL Collectors in each node can process the data in batches, keeping the overhead low. For instance, for an application component sending telemetry data at a high rate, the collector agent can store the data in memory (perhaps even process them to perform filtering and aggregation; see next) and then forward it to the gateway at a lower rate.

It is also possible to apply transformations and aggregations to the raw data before forwarding them to the gateway collector. Note that the OTEL Collector at the node level can route the raw and the transformed telemetry data to different exporters. The raw data exporting route, provides an endpoint that can be queried locally.

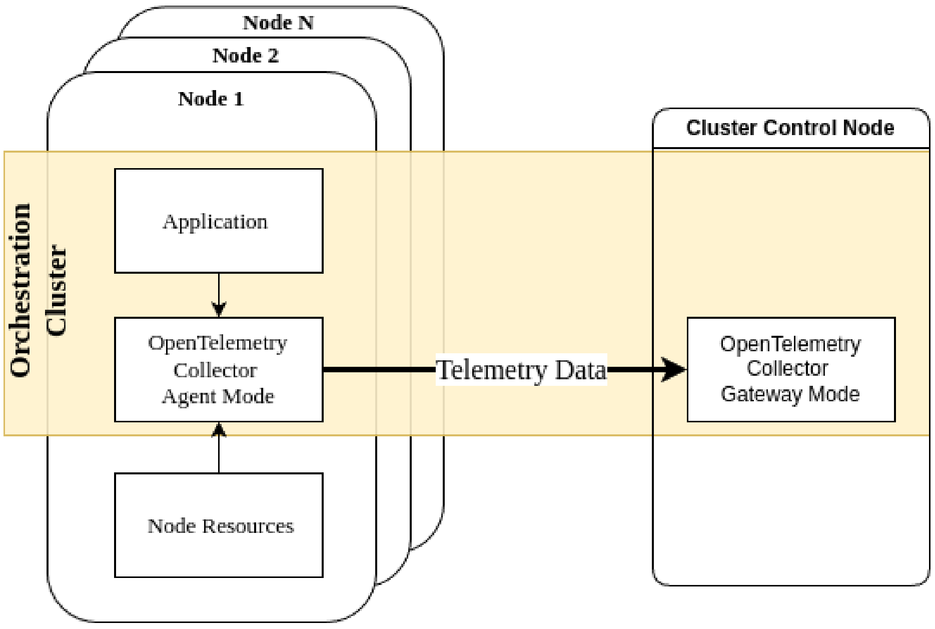


**Figure 7: OpenTelemetry Collector node deployment in MLSysOps Framework.**

### 2.4 Cluster level telemetry

At the cluster level, different components need to be monitored. The telemetry data in this layer must describe the status of the cluster rather than of a specific node. The main source of information for this level is the orchestration manager. There is, therefore, a dedicated OTEL Collector configured to collect metrics from the orchestrator, as depicted in Figure 8.

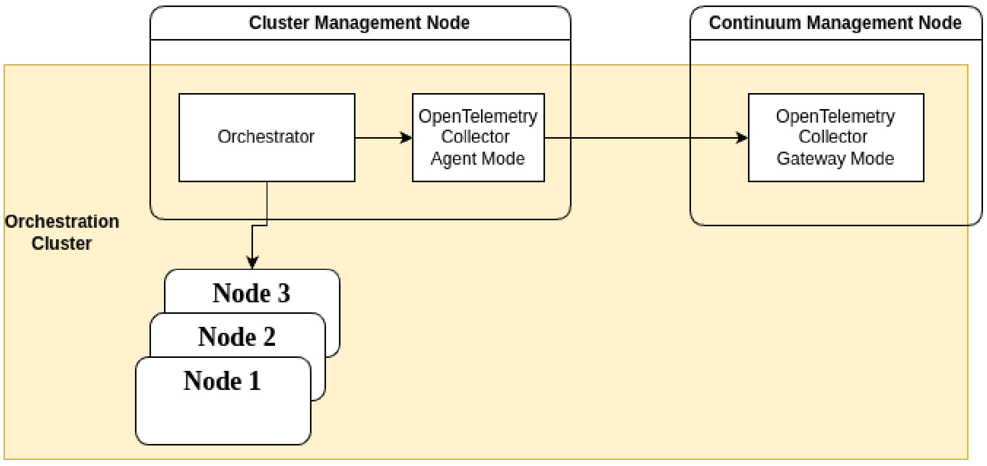

Cluster Management Node
Continuum Management Node
Orchestrator
OpenTelemetry Collector Agent Mode
OpenTelemetry Collector Gateway Mode
Orchestration Cluster
Node 3
Node 2
Node 1

**Figure 8: OpenTelemetry Collector cluster deployment in MLSysOps Framework.**

## 2.5 Continuum level telemetry

At the highest level of the MLSysOps framework, telemetry data is used not only for configuration decisions but also for informational and debugging purposes. This layer also includes components for telemetry data storage and visualization. Figure 7 illustrates the main components operating on the higher-level node (continuum).

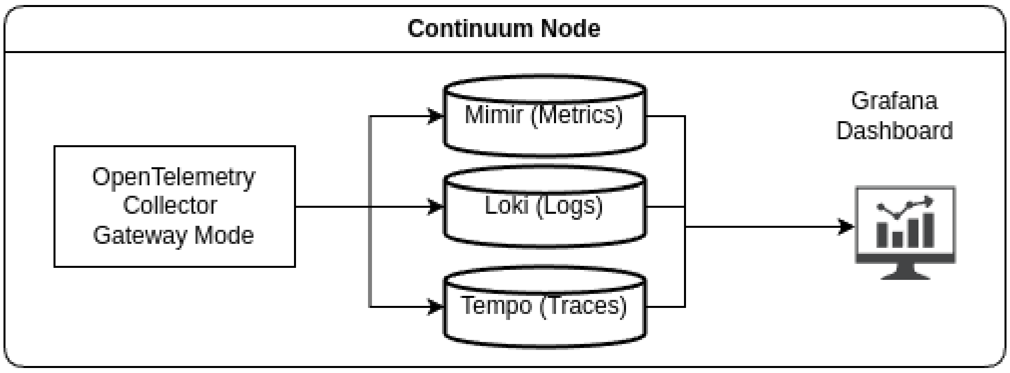


**Figure 9: MLSysOps Telemetry System continuum components.**

## 2.6 Telemetry system monitoring and adaptation

Gathering and processing telemetry data comes at a cost, both in terms of system resources and computational overhead. The frequency of data collection (high vs low data rates) and the origin of the data play a significant role in resource consumption. Some metrics are inherently more resource-intensive to collect, such as those obtained through external applications like perf. The overhead is higher for pull-based data collection, where the OTEL Collector actively polls other components for data updates. In contrast, push-based data collection involves components proactively sending telemetry data to the OTEL Collector. Application components often use push-based data transmission, allowing them to inject arbitrary telemetry data. Although push-based mechanisms offer flexibility, they also pose the risk of excessive data transmission and processing if not managed effectively. High data frequency can strain system resources and network bandwidth, particularly during peak traffic periods. All these functionalities need to be monitored and regulated to avoid problems with high utilization from the telemetry system itself. OTEL Collectors have built-in capabilities to share their own telemetry data, offering a clean way for any interested entity in the MLSysOps framework to monitor the behaviour of the telemetry system. Moreover, the MLSysOps telemetry system offers the necessary mechanisms to adapt various settings of the OTEL Collectors, such as the collection interval of the receivers. The adaptive configuration of the OTEL Collectors is handled by the MLSysOps agents by modifying the respective configuration files and restarting each collector.

## 2.7 MLSysOps Telemetry SDK & API

The MLSysOps framework offers integrators two pathways to interface with the telemetry system: i) use the OpenTelemetry SDK and its respective API or ii) employ the MLSysOps Telemetry SDK, which provides a simplified API. The former provides SDKs for a wide range of programming languages, enabling both manual instrumentation and automated instrumentation on compatible software systems. The latter serves as a wrapper on top of the OpenTelemetry SDK, abstracting away all the mundane code required to connect to an OpenTelemetry Collector and push metrics. This abstraction uses two function calls: one for pushing metrics and one for retrieving metrics. Instrumenting application components within the MLSysOps Framework is achievable with either of these options. MLSysOps Telemetry API/SDK is implemented in Python and C++ languages and is available in the opensource repository [2].

# 3 Mechanisms for AI-ready Computing Infrastructure

This section presents the foundational mechanisms that enable MLSysOps to monitor, configure, and optimize computing resources across heterogeneous platforms within the cloud-edge continuum. The focus is on node-level management, where different architectures—including x86-64, ARM, NVIDIA GPUs, and FPGA-based MPSoCs—are monitored and configured using hardware-specific interfaces. The section describes how telemetry data is collected via open-source tools such as Prometheus Node Exporter and NVIDIA GPU Exporter, and how configuration actions such as dynamic voltage/frequency scaling (DVFS), power capping, and clock adjustment are carried out depending on the platform capabilities. In addition, the section introduces the vAccel framework, which abstracts hardware acceleration through portable APIs and supports deployment in virtualized environments using gRPC and vsock communication. Together, these mechanisms establish the operational foundation for MLSysOps to deliver adaptive and ML-enhanced control over computing resources, while remaining hardware-agnostic and easily extensible.

## 3.1 Run-time Management of Configurable Parameters

The first step towards managing computational resources is to collect the necessary telemetry data to monitor the overall state of the system. The MLSysOps Framework includes a versatile telemetry system that can receive data in many formats, as described in Section 2. We use the Prometheus Node Exporter [10] to collect various system-level metrics from a computation node and transmit them to the MLSysOps telemetry system. These metrics include information about CPU usage, memory utilization, disk I/O, network activity, and other relevant statistics. The Prometheus Node Exporter collects metrics from the machine's operating system and hardware counters on resource usage and system performance. It also abstracts out all the low-level details of how to receive the available hardware metrics in different platforms and hardware architectures (x86 64-bit, Arm 32-bit and Arm 64-bit) using a common API. In the same spirit, we use NVIDIA Graphics Processing Unit (GPU) Exporter [11] to expose any available hardware and software metrics for the NVIDIA GPUs, on the nodes where such accelerators are available. Those two open-source tools were chosen due to their comprehensive supported metrics list and their easy interoperability with the OTEL Collector. More specifically, the OTEL Collector is configured to receive the data from both services with a pull logic at specific intervals. The list of available metrics for each of the used exporters, can be found on their respective repositories [10], [11].

In terms of configuring the computational resources, different API calls and tools are used according to the characteristics and capabilities of the hardware of each node. In the following section, we delve into the details of how telemetry and configuration are being implemented in each of the node platforms currently used in MLSysOps. The implementation of the node-level software components can be found at the MLSysOps GitHub public code repository.

## 3.2 x86-64 architecture

Intel's x86-64 architecture is dominant in Cloud and Edge platforms (e.g. datacenters, workstations, desktops, and laptops) and offers the richest set of telemetry and configuration settings among all computing platforms in a Cloud-Edge-IoT continuum. In all cases, the HW information that is available to the telemetry pipeline and the settings that a node agent can configure depend on the CPU microarchitecture.

***Telemetry***

To access the performance monitoring features on x86-64 CPUs, the Performance Monitoring Unit (PMU) is employed. The PMU is a dedicated hardware component that enables software to monitor diverse performance-related events, such as retired instructions, cache misses, etc. The PMU is accessed through *perf*, a performance analysis tool integrated into the Linux operating system and supported by the Linux kernel. *Perf* facilitates both kernel and user-space profiling, allowing the analysis of software behaviour at various levels. Due to the utilization of hardware performance counters *perf* has low overhead, meaning it minimally interferes with the system's performance during analysis.

In addition, Running Average Power Limit (*RAPL*) is an interface for reporting and managing the accumulated energy consumption of various CPU and DRAM power domains within a time window. There are multiple ways to read the *RAPL* registers in the latest Intel CPUs (Sandy Bridge microarchitecture and newer), including reading directly Intel's Model Specific Registers (MSRs) or using the *perf* API calls. Note that MSRs are special CPU registers that are used to control and configure various architectural features of the processor.

Both *perf* and *RAPL*, as well as other HW-, network-, and OS-level metrics, are exposed by the Node Exporter and are periodically fed into the telemetry pipeline.

***Configuration***

Setting the clock frequency and the voltage of a processor is done through the Dynamic Voltage and Frequency Scaling (DVFS) mechanisms provided by the operating system. On Linux systems, we change the clock frequency and the voltage by interacting with the *sysfs* entries associated with the *cpufreq* subsystem. The availability of these *sysfs* entries depends on the specific hardware and kernel configuration. For newer Intel architectures (Sandy Bridge and later generations), there is a more advanced scaling driver called *intel_pstate*. Intel CPU P-states represent the performance states of the CPU in terms of voltage-frequency internal control states. The *intel_pstate* driver exposes the P-states to user space, providing the ability to query and configure the CPU frequency at runtime. Our mechanism checks the availability of this driver and uses it instead of the basic driver in the *cpufreq* subsystem. The AMD processors are configured through the basic *acpi-cpufreq* subsystem.

Besides reading average power dissipation, the *RAPL* interface is also used to set a power limit on different CPU power domains within a user-defined time window. Note that when a system is power-capped, RAPL may dynamically reduce the clock frequency of the CPU domain to ensure that the power consumption remains below the specified limit. Therefore, power capping is a higher-level mechanism to control overall power consumption, and it often influences the dynamic adjustment of clock frequencies to achieve the specified power constraints. Setting the power cap is accomplished by writing the power limit to the respective MSR for the power domain we are interested in. The term "power domain" refers to different components within the processor that can be controlled or monitored independently in terms of power consumption. Even if the specific power domains differ between different microarchitectures, some common power domains in Intel's CPUs are the Core Power Domain (all CPU cores and their Level 1 and Level 2 caches), the Uncore Power Domain (Level 3 caches, memory controller, etc.), the Graphics Power Domain (for integrated GPUs), the DRAM Power domain, etc.

## 3.3 ARM architecture

ARM architecture CPUs are prominently used on the embedded and mobiles devices that form the Edge and Internet of Things ecosystem. Their main characteristic is the lower power consumption while achieving promising levels of performance. As MLSysOps framework management capabilities reach the Edge layer of the continuum, ARM CPUs are widely used in our testbeds, which include Raspberry PI boards, Xilinx MPSoC Development Boards etc. One differentiating fact for the ARM CPUs, compared to the x86-64 architecture, is the variation of form and sizes: ARM CPUs are used in Single Board Computers as standalone chips, in System on Chips paired with other hardware components (GPU, FPGA) as well as in more constrained devices with very low power needs or real-time requirements (e.g. ARM M). This variability needs appropriate handling from the MLSysOps framework and the respective mechanisms.

***Telemetry***

For nodes that have an ARM CPU, like the ZCU102 Development board that uses the 64-bit Quad-core Cortex-A53, we follow the same approach as for the x86-64 platforms by exposing *perf* and other HW-, network-, and OS-level metrics through the Node Exporter and feeding them into the telemetry system.

Some of these platforms do not have dedicated hardware that can report the power consumption. In this case, power monitoring is based on third party sensing hardware. We successfully managed to monitor the power consumption of the Raspberry Pi 3 using an INA226 sensor that measures the current that flows into the whole Raspberry board, thus calculating the power that is consumed from every peripheral and device on the board, including the CPU. On the other hand, the ZCU102 Development Board has dedicated INA226 sensors that measure the power on different power rails for the various parts of the MPSoC and the board (e.g. APU, FPGA fabric, DRAM etc). In both cases, we extended the Prometheus Node Exporter capabilities and implemented the necessary collectors to read the measurements of those sensors.

***Configuration***

ARM CPUs are compatible with the Linux *cpufreq* subsystem, enabling processor frequency control at runtime in the same way as the x86-64 processors.

## 3.4 NVIDIA GPUs

The management and monitoring of NVIDIA GPUs is based on the tools the NVIDIA driver and runtime offer. The basic software tools that we used is the *nvidia-smi* and *NVML*. We have currently tested the following functionalities on the following NVIDIA GPUs: Quadro K80, Quadro K1200, GeForce RTX4090 and different NVIDIA Jetson devices.

***Telemetry***

The telemetry data that concerns the NVIDIA GPUs are reported through the NVIDIA *nvidia-smi* command line tool. The latter is used by the NVIDIA GPU Exporter to collect the available metrics.

***Configuration***

To achieve better efficiency for the NVIDIA GPUs, the MLSysOps framework leverages the NVIDIA Management Library (NVML) [12] to set specific clock frequencies for the GPU. More specifically, the *pyNVML* Python library [13] is used to implement the necessary mechanism to interact with the NVIDIA GPU settings, to query and set the corresponding frequencies. NVIDIA GPUs have two basic clock domains that can be configured independently: i) memory and ii) graphics clock frequency. Both frequencies can change at runtime, altering the performance vs power ratio.

## 3.5 FPGA ZCU102 MPSoC

The FPGA SoC node is built around the Zynq UltraScale+ MPSoC, which combines programmable logic (in the form of FPGA fabric) with processing subsystems based on a 4-core Arm Cortex-A53 CPU (called Host unit). In such a platform, the user typically executes parts of the application in the Host Unit and exploits the FPGA fabric to run custom hardware accelerators. In MLSysOps, the accelerators can be used for speeding-up processing of the application components as well as for improving training/inference of the local ML models. ZCU102 is used as a Smart Edge platform in the MLSysOps ecosystem, especially when high-performance and low-power functionality is needed.

***Telemetry***

ZCU102 MPSoC includes both an ARM CPU and an FPGA Fabric. The details about the CPU metrics are mentioned in Section 3.3. Note that the telemetry system does not receive any data from the FPGA fabric, with the exception of power consumption, which is reported by power monitoring sensors on the board.

***Configuration***

FPGA Fabric is programmed using a generated bitstream binary, which is essentially a hardware implementation of a specific functionality (e.g. a computation kernel or a signal processing module). A fully programmed FPGA fabric cannot be altered at runtime easily, and the initial hardware design itself must support some form of configuration to enable a certain level of reconfiguration.

One way to achieve this, is to reprogram the entire fabric with a different bitstream, that has different properties. This is a time-consuming procedure and not feasible for most of the cases. As an alternative, we investigated the feasibility of using the *Dynamic Function eXchange* (DFX) [14] capability to accomplish a higher level of modification of the hardware kernels. This approach yielded negative results, as DFX introduces significant complexity for developers and causes noticeable performance overhead, making it impractical in most scenarios. The MLSysOps framework therefore adopted a simpler approach, which requires the FPGA MPSoC board to be reprogrammed with a system reboot. Although this method introduces downtime for the entire device, it simplifies programmability while maintaining optimal performance.

## 3.6 vAccel Extension for Hardware Acceleration

vAccel [15] is a framework designed to enable portable and secure hardware acceleration in multi-tenant environments. The core idea of vAccel is decoupling the user application from hardware-specific code. The vAccel runtime exposes to user application functions that can be accelerated. However, the actual hardware-specific code implementing these functions for a particular hardware accelerator device is provided in the form of so-called plugins, which are loaded at runtime. As a result, vAccel applications can migrate from one host to another without necessitating code modifications or re-compilation. At the same time, the vAccel modular design eliminates user code running on shared accelerators. Only code included in a vAccel plugin executes on the hardware accelerator, decreasing the effective attack surface[3].

Moreover, vAccel employs an API remoting approach to expose hardware acceleration inside virtual machines. vAccel uses a transport plugin to dispatch user API calls to the host over a VirtIO Sockets (vsock) device. On the host side, a vAccel Agent listens for acceleration requests, executes them on the available hardware devices, and responds with the computation result.

### *3.6.1 vAccel Runtime*

Figure 10 depicts vAccel runtime (vAccelRT), the core component of the vAccel framework. vAccelRT is basically a thin dynamic library. On the front-end, it exposes to the application an API consisting of a set of “accelerable” functions, while at the back-end it manages a set of plugins providing hardware implementations for the vAccel API.

The hardware accelerator-specific implementation of the API lives inside vAccel plugins. A vAccel plugin is a dynamic library which implements a subset of the front-end API for a particular hardware accelerator. When a vAccel application is being started, vAccelRT loads the available plugins on the platform inside the application's

[3] vAccel plugins are infrastructure-provided code that can be closely audited by the infrastructure provider. Hence, their functionality is only exposed at an API level.

address space. Subsequently, it is responsible for dispatching vAccel API calls made by the user application to the corresponding implementation of one of the available vAccel plugins.

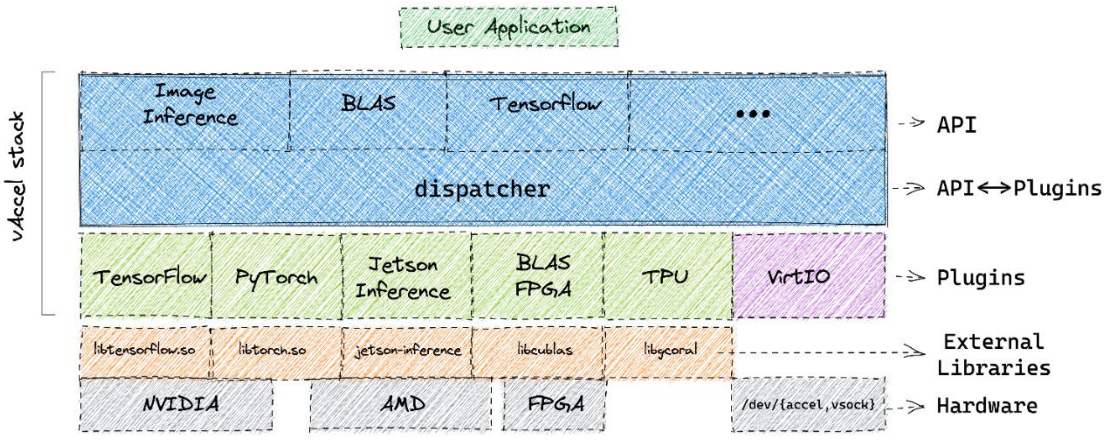


**Figure 10: vAccel runtime system.**

For example, Figure 11 depicts a vAccel application that consumes the Image Inference API of vAccel. When the application makes a call to the vAccel API, vAccelRT searches for any of the available plugins that implement the relevant call. If one is found, the call is dispatched to it (in this case, a jetson-inference-based plugin), which, in turn, executes the operation on the corresponding hardware accelerator device.

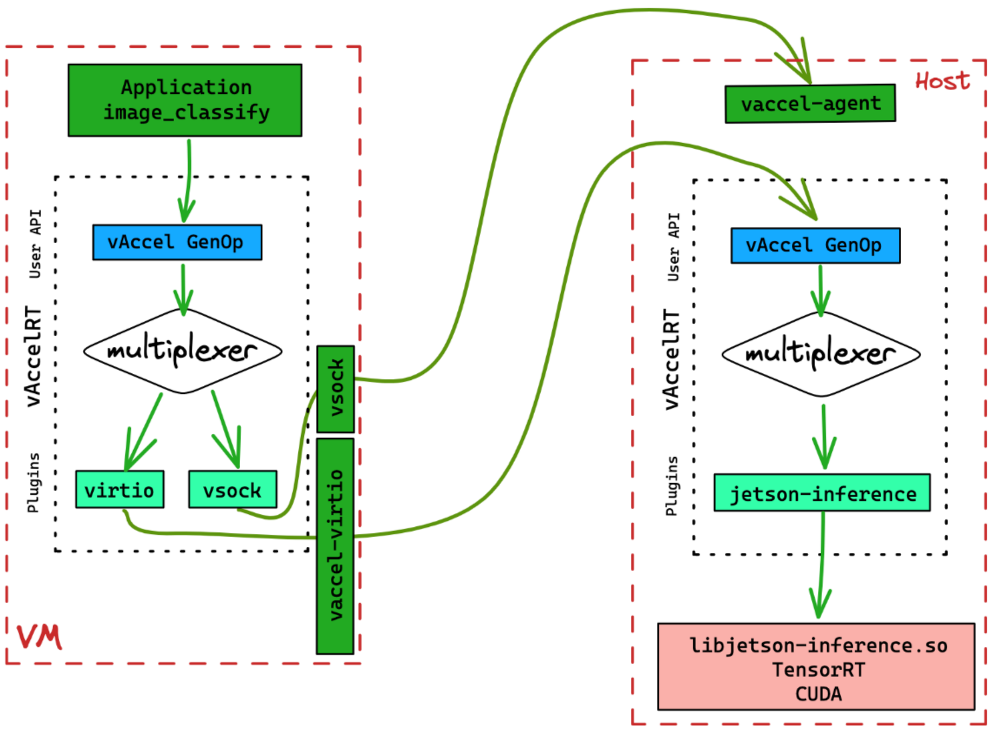


**Figure 11: A vAccel application using the Image Inference API of vAccel. vAccelRT dispatches the API calls to the jetson-inference plugin which performs the operation on GPU.**

Separating user- from hardware-specific code is the key concept of vAccel design that renders vAccel portable across hosts with varying hardware accelerator devices. The vAccel plugins are the components that provide the hardware implementation of vAccel functions. Consequently, migrating a vAccel application from one host to another is straightforward; we only need vAccel plugins implementing the required subset of the vAccel API on each one of these hosts (Figure 12).

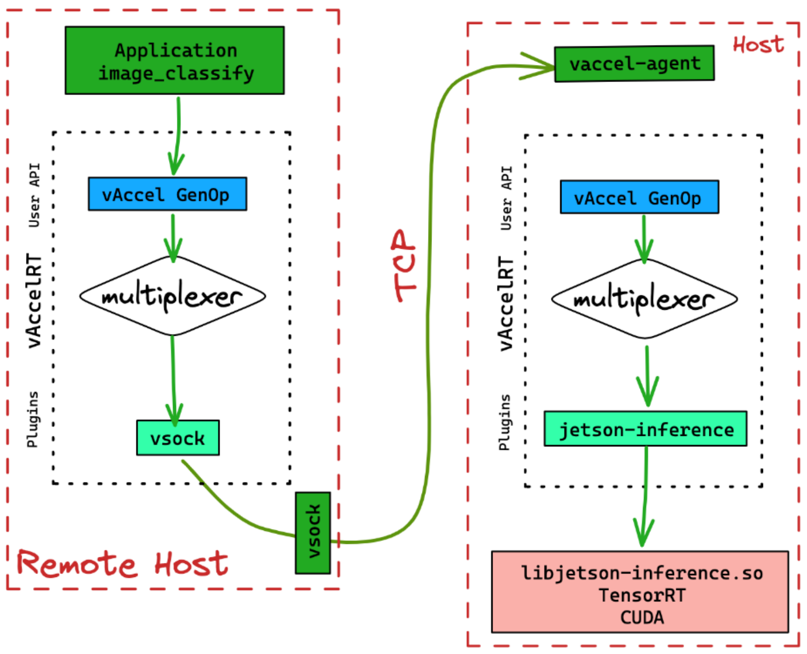


**Figure 12 The same Image inference vAccel application as in Figure 9, using a remote accelerator host.**

### 3.6.2 *Virtualization*

The vAccel framework exposes hardware acceleration inside virtual machines in a hypervisor-agnostic fashion by employing an API remoting approach. The natural granularity for vAccel to intercept code and offload it to the host is that of vAccel API calls. This differentiates vAccel from other API remoting frameworks such as rCUDA which on one hand operate at the level of the accelerator hardware driver, i.e. at a finer-grain level (which renders it more difficult to hide latency overhead) and on the other hand they are hardware-device specific, e.g. NVIDIA.

### 3.6.3 *Vhost communication channel*

The vAccel API remoting transport channel for an application executing inside a VM is vsock, which is a paravirtual interface for socket-like communication between a guest VM and its host operating system. On the guest side, the channel is exposed as a paravirtual device under /dev/vsock, whereas on the host side, the hypervisor exposes a UNIX socket. A vsock address consists of two parts, a 32-bit Context Identifier (CID) address and a 32-bit port number where ports below 1024 are privileged.

### 3.6.4 *gRPC*

vAccel builds the serialization of acceleration requests on top of the gRPC framework [14]. gRPC is a remote procedure call framework designed to enable efficient service communication across data centres or within microservices. It uses HTTP/2 at its transport layer and Protocol Buffers (protobuf) as its service description language, while it provides features for authentication, bidirectional streaming and flow control, blocking or nonblocking bindings, cancellation and timeouts, as well as telemetry. Finally, it allows generating bindings for various high-level programming languages and tools for creating boiler-plate code both for client and server implementations. vAccel API has been translated into protobuf definitions, and client and server implementations have been created based on these definitions. Currently, vAccel supports the synchronous flavour of gRPC API services while the potential performance benefits of adopting the asynchronous semantics as well is under evaluation. Client (guest) side implementation of the vAccel API remoting mechanism is built on top of the client definition of the vAccel gRPC service. The core functionality is built within a vAccel plugin, which implements the whole vAccel API. Its responsibility is to serialize vAccel API calls to gRPC calls and transmit them over the vsock-end of the communication channel.

### *3.6.5 Server (host) implementation*

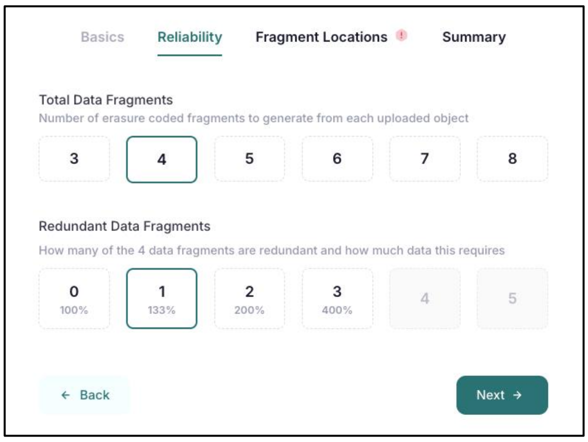
In order to handle acceleration requests originating from inside a guest VM, vAccel introduces a vAccel Agent application. The Agent is, essentially, a server binding on the UNIX end of the vsock channel, which the hypervisor exposes as a named socket on the host file system. It is implemented on top of the server definition of the vAccel gRPC service and it is essentially a vAccel application, meaning that it links against vAccelRT.

# 4 Mechanisms for AI-ready Storage Infrastructure

SkyFlok [16] is a cloud-based software as a service (SaaS) for secure file sharing and storage, developed and maintained by CC. It uses a combination of encryption and erasure coding to encode files into redundant data fragments, which are persistently stored in multiple, user-selected data centers operated by different cloud providers (e.g., Google, Amazon, Microsoft). SkyFlok has integrations to 15+ cloud storage providers, supporting over 100 distinct storage locations worldwide. While the original SkyFlok product is a browser-based web application designed for small-medium businesses (SMBs), it has recently been expanded with an S3-compatible object storage service that is meant to be either the primary or backup data storage solution for applications and systems.

## 4.1 Extension of SkyFlok for Adaptive Fragments Distribution

Currently, SkyFlok users manually select (as shown in Figure 13) the storage locations and level of redundancy during the setup process and the settings are immutable during the whole lifetime of buckets. The choice of storage parameters (locations and redundancy) directly affects the service quality for end users and applications accessing the bucket: users closer to the selected storage locations will experience lower latency, while higher redundancy results in higher availability and potentially faster downloads.

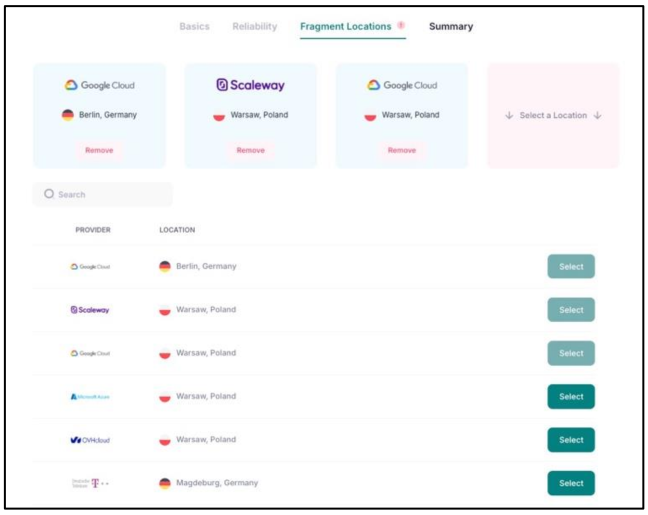


Total Data Fragments
Number of erasure coded fragments to generate from each uploaded object

3 [✓ 4] 5 6 7 8

Redundant Data Fragments
How many of the 4 data fragments are redundant and how much data this requires

| 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 100% | [✓ 133%] | 200% | 400% | | |



Selected: Google Cloud – Berlin, Germany; Scaleway – Warsaw, Poland; Google Cloud – Warsaw, Poland

| PROVIDER | LOCATION | |
|---|---|---|
| Google Cloud | Berlin, Germany | Select |
| Scaleway | Warsaw, Poland | Select |
| Google Cloud | Warsaw, Poland | Select |
| Microsoft Azure | Warsaw, Poland | Select |
| OVHcloud | Warsaw, Poland | Select |
| Deutsche Telekom | Magdeburg, Germany | Select |

**Figure 13: SkyFlok Object Storage: fixed erasure coding config and storage location selection.**

While this fixed configuration works well in situations where the volume and geographical distribution of demand are static and can be forecasted with high confidence, it suffers when either of them varies greatly over time or cannot be predicted reliably. Note, that exceptional traffic spikes are not the only reason for dynamic changes in access patterns, but it is common for data to lose value over time. When data is fresh, it is likely to be accessed significantly more frequently than months or years later. To ensure best performance, bucket creators must allocate storage resources for the highest anticipated demand and either keep this expensive configuration forever or must develop additional processes to manually migrate old data to a "colder" bucket with less redundancy and cheaper storage. Typically, object storage services only allow changing the storage tier between "hot" and "cold" based on the last time of access, but they do not offer high level of flexibility and control.

The SkyFlok Object Storage service is enhanced in MLSysOps with new capabilities that allow ML-based agents to monitor the real time traffic of buckets and adjust the storage configuration when significant changes are detected, while the bucket stays fully operational. This approach relieves the application developer from overprovisioning storage resources just to prepare for short time periods when data is frequently accessed, and from having to anticipate the volume or geographical origin of demand altogether.

SkyFlok offers new APIs for the MLSysOps platform that allow updating the storage configuration ("Storage Policy") of buckets enrolled in this dynamic scheme. This policy contains information about how object contents are processed when uploaded to the bucket. Most importantly, it includes the erasure coding configuration (number of total and redundant data fragments) as well as the storage locations where coded fragments are placed persistently.

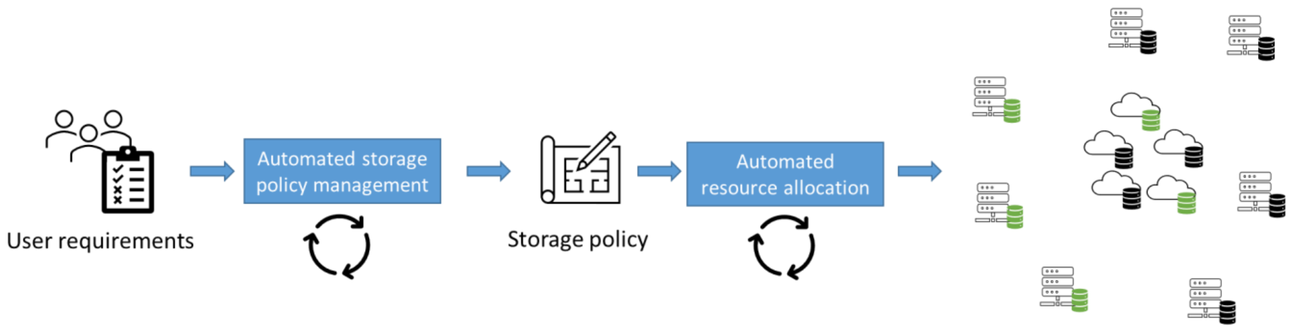

User requirements
Automated storage policy management
Storage policy
Automated resource allocation

**Figure 14: Automated storage configuration and resource allocation management.**

The process  starts with the application developer creating a bucket on the *SkyFlok S3 Developer Portal* and setting its initial Storage Policy based on the estimate of the volume and origin of users accessing the bucket. Then, the developer creates an API Key that allows the MLSysOps framework to update the Storage Policy of the bucket. The application developer then adds the bucket name, API key, high-level performance requirements and optionally storage location restrictions (e.g., a continent or region where all fragments must reside for legal/compliance reasons) to the MLSysOps application description, which is submitted to the MLSysOps platform. At this point, the bucket is fully set up, ready to be used by applications and managed by MLSysOps.

During the lifetime of the bucket, SkyFlok reports all object access events to the MLSysOps platform, which periodically checks whether the current Storage Policy is still best suited for the bucket's performance requirements. In case it detects either that the bucket underperforms, or a cheaper storage configuration would also satisfy the target performance, the ML model generates a new Storage Policy and invokes the policy update mechanism in SkyFlok. In turn, SkyFlok determines the specific actions that are needed to migrate existing objects of the bucket from their current configuration to the new Storage Policy and schedules changing each existing object.

Depending on the concrete change in the storage configuration, the amount of work SkyFlok needs to carry out varies greatly. It can be as little as deleting a single coded fragment per object (e.g., when changing from erasure coding 5+2 to 5+1), or as much as having to re-generate all fragments (e.g., erasure coding 3+1 to 4+1). The change may include any combination of moving existing fragments to different locations, as well as deleting or generating new ones.

After determining the steps needed for migration, a series of background tasks are launched that execute these jobs and create new versions of the bucket's objects according to the new Storage Policy. When all objects have been converted, the internal metadata of their old versions is atomically replaced by the new one, and no longer needed data fragments are scheduled for removal. By performing the potentially long-running transformation process in the background, the bucket stays fully usable during the operation: existing objects are available, and new data can be uploaded without the clients ever noticing the ongoing migration.

### *4.1.1 Telemetry for Storage Metrics*

When the storage configuration of a bucket is not well aligned with the current demand, several root causes are possible. For example, the redundancy level may be too high or too low, demand may be originating far away from the fragment locations, or some of the data centers that hold fragments may be experiencing degraded performance or a temporary outage.

To find a better Storage Policy, the MLSysOps framework needs up-to-date information about both the volume and geographical location of the demand arriving to managed buckets, as well as current and historical performance (latency and bandwidth) of all available storage locations. SkyFlok sends both data as telemetry. When an object in an MLSysOps-managed bucket is accessed, a notification is sent by SkyFlok, informing the framework of the bucket, timestamp and the country where the request came from (based on the client's IP address, which is not stored persistently). Additionally, after object downloads, SkyFlok reports the speed of retrieving data fragments from the storage locations, giving the MLSysOps framework near real-time information about their current performance. However, there may be storage locations that do not experience a lot of traffic and stay invisible to the MLSysOps framework, which therefore cannot assess if these would be suitable replacements of existing locations in an underperforming bucket. To this end, SkyFlok periodically measures the latency and bandwidth to all available locations and reports these results to the MLSysOps framework as well.

# 5 Mechanisms for AI-ready Networking Infrastructure

The Datacenter Job Scheduler Interface has been advanced to support additional parallelism dimensions, including the Mixture of Experts (MoE) dimension, to better accommodate evolving LLM model communication patterns. Backend development now allows the scheduler to generate Optical Switch Matrix configurations using an intermediate representation, which supports heterogeneous devices by converting permutations through device-specific drivers.

For the 5G Core Control Plane, the task outcome includes the Python-based software for managing and controlling monitoring scripts, enabling continuous profiling of critical network parameters under various traffic conditions. This telemetry framework provides detailed analysis of network and computational parameters ensuring robust monitoring and dynamic profiling.

## 5.1 Datacenter Job Scheduler Interface Development

NVIDIA datacenter use case is focused on reconfigurable physical topology enablement for AI clusters, which are primarily designed for AI training - not inference. Reconfigurable physical topology is implemented on top of an optical network that leverages optical switches to switch between different effective topologies. This is way more efficient than using Fat Tree topologies that always provide full bisection bandwidth between any node of the cluster, because a given AI workload mix that occupies the full cluster has well-defined traffic patterns that will never use all the available Fat Tree connections during their execution. By adapting the topology to the workload, we need to deploy fewer physical connections. This results in cheaper AI clusters that consume less power without sacrificing performance. The optical switch connectivity permutation is inferred by the placement of AI training jobs and their communication pattern. By placing the jobs appropriately, we make sure there is always an optical network topology that can serve them, despite the reduced networking connections.

More specifically, in the NVIDIA datacenter use case, MLSysOps machine learning input is a description of a batch of an arriving AI training job mix, each job’s connectivity requirements across the required GPU nodes and the current cluster occupancy. The training jobs that we are looking at are a mix of Deep Learning Recommender Models (DLRM) and Large Language Models (LLMs). These exhibit specific network traffic patterns on the datacenter network with well-defined periodicity. More specifically, DLRMs perform periodic all-to-all communication following a well-known algorithm and LLMs all-reduce communication.

NVIDIA has developed a series of best fit algorithms with some heuristics that are proven to perform acceptably in practice. What is currently missing is a wider insight on how the job scheduling decisions we make in each slot affect the cluster utilization in the long run. Currently, each job in the arriving batch is assigned resources in a First Come First Served manner, but we know that serving the batch jobs in different order results in different occupancy states of the cluster. The state captures both computation and network resources. Computation nodes (GPUs) are static and non-configurable, and their availability and utilization are mostly used as a metric. However, the network connections between the nodes and the GPUs, with their respective utilization, have a higher degree of configurability and can offer better results. The network configuration depends on the job scheduling decision. The network is implicitly configured by internal mechanisms of the network backbone that include Optical Circuit Switches (OCSs), based on the job placement. Therefore, the machine learning output is the order in which each job batch should be scheduled so that existing algorithms offer better cluster utilization across large periods of time (e.g. months of running workloads). Both the described job scheduler input and job scheduler output comprise well-defined JSON descriptions offered via a REST API and this is currently part of the NVON simulator functionalities that have been developed in the project.

The Datacentre Scheduler offers the cluster occupancy state and arriving workload batch as Telemetry outputs and gets as configurationthe order of execution of the workload. In more details, these parameters are:

- **Cluster GPU occupancy**
  *Definition:* GPU Instance Occupied along with remaining time and OCS uplink group that is currently part of.
  *Significance:* It reflects cluster occupancy and is a required machine learning system input.

- **Cluster OCS uplink occupancy per packet switch**
  *Definition:* For each packet switch in the network that is connected to an Optical Switch with a fixed number of physical links, this composite metric reflects the current occupancy of each such fixed link along with detailed allocation information (i.e. allocation timestamp and expected occupancy duration).
  *Significance:* It reflects cluster link occupancy and is a required machine learning system input.

- **Arriving Job batch**
  *Definition:* Description of arrived jobs batch. Each job in the batch reports number of GPUs required, network pattern, and approximate compute duration.
  *Significance:* It reflects resource request characteristics and is a required machine learning model input.

In the context of this deliverable a second iteration of the scheduler design has been carried out, factoring in the evolution of LLM model collective communication patterns. The best fit algorithms have been revisited in the described context. We have also developed a backend so that the scheduler generates the required Optical Switch Matrix configuration which can be fed to the optical switch device and implement the required permutation. We have developed an intermediate representation of OCS permutation which is converted by optical switch device specific device driver so heterogenous devices can be supported

## 5.2 Mechanisms in 5G Core Control Plane

In 5G core network the traffic can be divided into two categories: control and user plane. The control plane influences functions like authentication, mobility, and network slicing. The user plane is the user network traffic and can be influenced by the control plane, which can choose which network path has to be used and which user plane network functions will be engaged by the user plane. In this project, when the MLSysOps agent makes the decision about the best location to be used by the user plane to minimize latency, the control plane needs to be triggered in order to start the process of configuring the core network accordingly to comply with such decision.

There are several ways to trigger the data path switch, and many of them are strictly related to the 5G core network exposure capabilities. The goal is to remain as generic as possible, so that the solution can be easily integrated into different core network designs. We selected three possible solutions for this. The first approach is to configure a particular 5G network slice, which is good to be used with simulated User Equipment (UE) but could bring problems in compatibility with real UEs, which in some specific UE models don’t manage correctly the network slicing request. The second option is configuring the SMF (Session Management Function), that is, the control part of a PDU session. It configures tunnels, allocates IP addresses and configures traffic steering. This configuration is valid for the default network slice of the user. The third method is to configure the APN accordingly, but in this case, it is valid for all users using that APN. We choose to use the second option, namely, to configure the SMF. This way, we will continue to use the network slicing concept, which is important in 5G, but we will not use more than one slice, so that real UE will not face problems.

The 5G core network needs an API to trigger the configuration of the network slice that will make the User Equipment choose the newly selected UPF. In this API is filled with the IP address that will identify the UPF network function that has to be targeted. This way, the SMF Control Plane functions will have an influence on the data path used by the user plane.

Our telemetry system is designed for continuous monitoring and dynamic profiling of critical parameters, allowing a comprehensive understanding of the quality of connection between gNodeB with multiple UPF deployments across varied locations of datacentres. This segment systematically captures data related to computational and network-centric aspects, such as CPU and memory utilization in deployed UPFs, bandwidth usage, and maximum bandwidth limits between gNodeB and UPF. These metrics are pivotal in assessing their impact on latency and packet loss. To ensure the authenticity of our dataset, dynamic profiling is conducted under both peak and non-peak traffic conditions in the 5G network. This approach ensures that the collected data faithfully represents real-world scenarios, providing valuable insights for machine learning algorithms.

As a real example for better understanding the general network model, within our telemetry framework, the network comprises a gNodeB and three UPF instances deployed as Virtual Machines (VMs) in three locations with relevant distances that bring to different latencies. The gNodeB acts as the primary access point, facilitating communication between user equipment and the 5G network. The strategically positioned UPF deployments in different cities mirror real-world scenarios, considering diverse geographical influences. This setup enables us to evaluate telemetry parameters in a geographically dispersed 5G network, refining configurations for adaptability across various deployment scenarios. Such parameters and metrics are:

- **UPF CPU Usage**
  *Definition:* Percentage of CPU utilized in the machine hosting the UPF.
  *Impact:* Elevated CPU usage may impact communication latency between gNodeB and UPF, leading to processing delays, increased context switching, and CPU resource competition.
  *Consequence:* Decelerated handling of network-related tasks, resulting in heightened communication latency.

- **UPF Memory Usage**
  *Definition:* Percentage of memory utilized in the machine hosting the UPF.
  *Impact:* Increased memory usage may contribute to communication latency through packet buffering, queuing, and potential swapping.
  *Consequence:* Delays in the transmission and reception of data.

- **Traffic Load (Uplink and Downlink)**
  *Definition:* Absolute value of traffic load measured in Mbps between gNodeB and UPF in both directions.
  *Impact:* Higher traffic load can elevate latency due to congestion, queuing delays, and potential network bottlenecks.
  *Consequence:* Slower data transmission and processing.

- **Bandwidth Usage (Uplink and Downlink)**
  *Definition:* Percentage of bandwidth utilized between gNodeB and UPF in both directions, calculated relative to the maximum available bandwidth.
  *Significance:* Represents connectivity situations independently from absolute Traffic Load values, crucial for varying UPF deployments.

- **Packet Loss**
  *Definition:* Percentage of data packets that fail to reach their destination (gNodeB or UPF).
  *Significance:* Reflects communication reliability and performance.

- **Latency**
  *Definition:* Absolute value of latency/delay measured in milliseconds between gNodeB and UPF.
  *Significance:* Crucial for optimized UPF placement and usage to meet Quality of Service requirements for 5G services.

- **CO2 equivalent per kWh**
  *Definition:* Carbon intensity measures how clean the electricity consumption is in a zone at a given time. It represents how many grams of carbon dioxide (CO2) are released in the atmosphere for each kilowatt hour (kWh) of electricity consumed.
  In other words, carbon intensity represents the greenhouse gas footprint of 1 kWh consumed inside that zone. This footprint is measured in gCO2-eq (grams of CO2 equivalent), meaning that each type of greenhouse gas can be converted to its CO2 equivalent in terms of global warming potential over 100 year (for instance, 1 gram of methane emitted has the same global warming impact during 100 years as ~34 grams of CO2 over the same period). The carbon intensity of electricity generation of a zone is determined by the power production mix and their associated carbon intensity factors. Data are retrieved via API from Electricity Maps (https://app.electricitymaps.com/).
  *Significance:* Understanding the CO2 equivalent per kWh is vital for assessing the environmental impact of energy consumption. This metric varies based on the energy source and the efficiency of power generation.

Various open-source tools are employed to effectively measure the following parameters:

- Computational Parameters: Utilizing tools such as top, htop, and free.
- Network Parameters: Employing tools like mtr, iperf, and twampy.

Additionally, we developed a Python-based software designed to manage and control script modules based on Bash scripting. These scripts make use of the tools for continuous monitoring. The controller can generate simulated workloads on the network that align with usage profiles during both peak and non-peak hours. Note that the workload is to be applied to all deployed UPFs. Therefore, datasets are collected for each UPF deployment. This comprehensive approach ensures a detailed and accurate assessment of telemetry parameters within a 5G network environment. In conclusion, the telemetry setup, enriched with dynamic profiling based on peak-aware traffic scenarios, establishes a robust framework for gathering measurements of selected parameters. The network setup, deploying UPFs across diverse geographical locations, further enhances the adaptability of our configuration. By simulating real-world workloads, we aim to enhance the overall quality of the collected dataset.

## 5.3 Resource Management Mechanisms of Wireless Edge Networks

Within the IoT-Edge segment of the MLSysOps infrastructure, resource management mechanisms focus on efficiently orchestrating wireless communications between low-end nodes and edge nodes through various wireless technologies. Nodes in this infrastructure typically possess multiple wireless interfaces, dynamically activating specific communication technologies based on the requirements of running services/applications, external conditions (such as interference or channel quality), and internal conditions (such as battery levels or hardware health).

An MLSysOps agent is embedded to oversee this dynamic environment, monitoring system behaviour and dynamically assigning critical communication parameters such as frequency channels and modulation schemes based on the type of application and the communication protocol in use. These assignments can be determined using heuristic methods or machine learning (ML) algorithms. Additionally, the agent leverages real-time monitoring data, including telemetry composed of logs and signal samples (e.g., I/Q samples) collected at multiple network layers and from deployed testbeds, to detect and respond to anomalies or potential attacks. These logs provide detailed insights into communication patterns between connected nodes, edge nodes, and the broader network infrastructure, enabling nodes to follow standard protocols for joining networks and ideally associating with the nearest or most appropriate edge node.

To evaluate and demonstrate these mechanisms, we implemented our test scenario within a representative 5G network environment consisting of User Equipment (UE), Base Stations (gNodeB), and the 5G Core Network (5GCN). In this scenario, wireless edge nodes (categorized as UE) are served by the 5GCN, and their behaviours are continuously monitored to optimize resource allocation and swiftly identify anomalies or attacks using machine learning models.

When the 5G network is deployed, the communication between the UE and the 5GCN is monitored and logs related to this communication from the gNB level are considered. These logs give an insight about the quality of the communication established between the UE and gNB/5GCN. For example, the Signal-to-Noise Ratio (SNR), Cyclic Redundancy Check (CRC) from the logs and other parameters as the time taken for the message to be transmitted and received between UE and gNB, etc, provide valuable information about the network's performance & behaviour in addition to the reliability of the communication links.

Complementarily, logs captured directly from UE devices—implemented using Android smartphones (OnePlus)—offer precise and granular insights into device behaviours. Notably, specific log tags, including AT (modem command traces), RILJ (communication protocol logs), and RmcNwhdlr (network event management logs), have demonstrated a strong correlation with anomalous behaviour. Analyzing these logs facilitates anomaly detection, thereby enhancing system robustness and security.

By analysing these logs, it is possible to detect anomalies and patterns that could indicate a potential security threat or attack. Such patterns are considered as indications of normal and abnormal network behaviour. Leveraging Machine Learning (ML) techniques and mechanisms, the MLSysOps agent is trained on the extracted features from the logs (both gNB level and UE level) to detect anomalies or potential attacks. This approach enables the development of a sophisticated intrusion detection system that can proactively identify and respond to various types of attacks in a 5G network. Through the process of continuous monitoring and analysis of the gNB and UE logs, the MLSysOps agent can learn to differentiate between normal and abnormal activity and thereby enhancing the overall security.

We also consider wireless digital traces, and more specifically I/Q samples to identify devices based on hardware impairments such as Phase Offset (PO), Direct Current Offset (DCO), Carrier Frequency Offset (CFO) and analysing their impact on the generated signal by the transmitter. The authentication process and the detection of the illicit intruder is done at the far edge node.

Finally, the MLSysOps infrastructure integrates geo-localization techniques, enabling precise location tracking and isolation of compromised or malfunctioning nodes within the IoT-Edge environment. Specifically, the GPS positioning feature of Teltonika 5G routers is employed to geo-localize edge devices. An example API call to obtain positioning data is as follows:

```
GET {{BaseUrl}}/gps/position/status
(authorization via bearer token obtained through login)

Sample Response (200 OK):
{
  "success": true,
  "data": {
    "accuracy": "500",
    "fix_status": "1",
    "altitude": "143.3",
    "speed": "0",
    "timestamp": "1742549118",
    "satellites": "2",
    "longitude": "9.162780",
    "latitude": "45.442997",
    "angle": "0",
    "utc_timestamp": "1742545518"
  }
}
```

The agent continuously monitors telemetry metrics such as SNR, CRC errors, and anomalous I/Q samples. Upon detecting abnormalities, the agent extracts relevant resource usage data and the affected node's geographical location to promptly initiate appropriate isolation measures using trained ML models. This holistic approach applies broadly to various wireless technologies beyond 5G, including WiFi, underscoring the versatility and effectiveness of the MLSysOps resource management mechanisms.

# 6 Mechanisms for AI-ready Application Deployment and Orchestration

Section 6 presents the MLSysOps mechanisms for application deployment and orchestration across the cloud-edge continuum. While it builds on established technologies like Kubernetes, Karmada, and container runtimes (e.g., kata-containers or containerd), the focus is on foreground contributions: novel mechanisms for hybrid runtime support, orchestration adaptations tailored to multi-cluster deployments, and dynamic far-edge deployment on resource-constrained devices.

MLSysOps introduces:

- Unified orchestration mechanisms across Cloud and Edge environments, integrated into higher-level orchestrators like Karmada and K8s.
- Hybrid execution environments blending traditional and sandboxed runtimes.
- Low-level custom container runtimes supporting sandboxing through microVMs
- Optimized sandboxing through an efficient hypervisor and stripped down applications (single application kernels and unikernels)
- Integration of hardware acceleration abstraction for sandboxed workloads (vAccel on kata-containers)
- Runtime plugin hinting in vAccel for adaptive accelerator selection.
- Far-edge orchestration extensions, including virtualized orchestrators, proxy agents, and embServe-powered deployments.

Background on standard techniques such as container sandboxing or microVM operation is referenced for context, while the emphasis remains on MLSysOps-specific adaptations and implementations.

## 6.1 Software Deployment and Orchestration

In this section, we discuss the functionality of the mechanism that is designed to enable adaptive deployment and orchestration across the mobile-edge-cloud continuum. The mechanism focuses on distributed applications that follow the microservices paradigm, where the desired functionality is achieved through the interaction between several components. For instance, an application component can implement specific functions/services, which are invoked/used by other application components. The implemented mechanism is called Fluidity [17], and has been developed for the scope of the project for the cluster-level management.

We build our mechanism on top of Kubernetes, which is one of the most common solutions for deployment and orchestration of distributed applications. Kubernetes provides a high degree of scalability, resilience, and reliability. In addition, it can be used in heterogeneous nodes (e.g, running on top of a large variety of platforms and operating systems) with minimal effort. It also contains an active community that continuously assists the growth/improvement of the platform. Since a large part of the project, including the application use cases, focuses on nodes at the edge, which usually do not have the same resources as typical datacenter nodes, we build on top of a lightweight version of Kubernetes, called K3S [18].

In addition, we assume Karmada [19] for multi-cluster management, as it is compatible with the native Kubernetes API, offering high flexibility and seamless integration of existing tools. In MLSysOps, we leverage Karmada to break the boundaries of a single control-plane, allowing for more flexible management of compute and network resources on individual clusters, with their own, localized control-plane, while maintaining a birds-eye view from a single, aggregate control-plane. The key innovation behind Karmada is its ability to act as a central control plane for managing resources across diverse Kubernetes environments. This allows organizations to harness the full potential of Kubernetes without being limited by individual cluster boundaries. Karmada's architecture enables efficient resource allocation and optimization, making it a valuable tool for modern cloud-native deployments.

In this subsection, we describe the following aspects of MLSysOps:

1. Cluster-level application deployment and orchestration using Fluidity, which takes decisions regarding the initial application deployment as well as adaptation while the application is running. It also provides support for application-level traffic redirection, so that alternative communication interfaces can be exploited rather than the default one (that is also used for control traffic).
2. Continuum-level orchestration and integration with Karmada to support the application deployment to multiple Kubernetes clusters.

### *6.1.1 Fluidity*

Fluidity enables flexible application deployment and orchestration across the entire system continuum. It is designed to manage modular applications where the developer merely provides the individual components as containers and annotates them with their resource, deployment, and interaction requirements. Based on this information, Fluidity deploys the application components in the cloud, on edge nodes, and mobile IoT nodes, and adapts this deployment at runtime without any intervention from the application owner or system administrator. Notably, Fluidity is responsible for managing applications on a given slice of computational and sensing resources, which may span all layers of the continuum.

It is built as an extension of Kubernetes, which is used as the underlying mechanism for the basic pod/container deployment, health monitoring, and inter-pod communication (using Flannel [20] as an overlay network). Going beyond the standard functionality of Kubernetes, Fluidity supports flexible application deployment for mobile nodes and edge computing with transparent redirection of application traffic over different network interfaces supporting IP-based communication, which may employ different technologies at the physical layer.

More specifically, Fluidity introduces the following extensions on top of Kubernetes: (i) support for multiple node communication interfaces; (ii) application-level traffic redirection between different interfaces; (iii) management of mobile nodes serving as hosts for application components; (iv) flexible deployment, driven by continuum-aware system and application descriptions; (v) support for different deployment policies, separated from the core deployment mechanisms.

The system infrastructure spanning the cloud–edge–mobile continuum, as well as the application and its components, is described via suitable Kubernetes Custom Resource Definitions (CRDs). This way, Fluidity can introduce all the structural elements necessary to support a declarative, intent-driven deployment of the application in the continuum with the desired flexibility and runtime adaptation.

Our implementation is based on a custom Controller that follows the Kubernetes operator pattern. The Controller is a watcher, capturing events (e.g., addition, modification, and removal) on Kubernetes Custom Resources. More specifically, the Controller handles Custom Resources that implement the MLSysOps' formal application descriptions. Notably, both conventional and ML-based policies (by using the MLSysOps ML Connector API internally) can be used to trigger potential adaptations in the deployment plan at the initial deployment phase as well as during the operation of the application.

The possible actions that a policy can ask from the core Fluidity mechanism are: (i) *Deploy*, if the pod should be deployed to the node. (ii) *Remove*, if the pod is no longer needed on the node, and should be removed. (iii) Move, if the pod should be relocated from its old host to a new one. We implement relocation as a combination of deployment and removal on the respective hosts. (iv) *RedirectAppTraffic*, if it is desirable to perform application data redirection. This can also be combined with component deployment/relocation actions. The feasibility of this action is validated by the mechanism based on the capabilities of the nodes that host interacting application components (else the hint is ignored).

### *6.1.2 Application-level traffic redirection*

We refer to a component as service-providing if it has an ingress relation with at least another component, while the latter is mentioned as a client component.

The implemented mechanism actively exploits the ability of a mobile node to interact with an edge node via direct wireless communication (for example, WiFi), rather than via the default communication path (for example, 4G and the public Internet). To this end, additional actions are performed depending on whether the service-providing component is relocated (i) from the cloud to an edge node, (ii) from an edge node to the cloud, or (iii) between two edge nodes.

In the first case, the mobile node is instructed to activate its WiFi interface and connect to the wireless network of the edge node (sending the necessary information, such as the SSID and key of the network). This is done before the generation of the pod specification and the deployment on the edge node. This way, the WiFi connection delay overlaps with the component deployment delay. Once the old pod is removed, the control plane of our mechanism notifies the mobile node to redirect application traffic to the component instance on the edge node over WiFi. In turn, the mobile node interacts with the edge node to jointly set/adjust the routing rules that are associated with the service-providing component.

In case the service-providing component is relocated from the edge back to the cloud, the mechanism instructs the mobile node to disconnect from the edge WiFi network. The WiFi disconnection completely overlaps with the pod-file generation and pod deployment/removal, in a similar manner as the WiFi connection in the previous scenario. Subsequently, the mobile node is notified to redirect application traffic by restoring the routing rules. As above, this is done after removing the component instance from its old host.

A similar process is followed when the service-providing component is relocated between two edge nodes. More specifically, after deploying the pod on the new edge host and removing the pod from the old one, the control plane instructs the mobile node to redirect application traffic. In this case, the mobile node configures the WiFi to connect to the network of the new edge host and updates the routing rules for the communication between the application components.

We should note that this reconfiguration knob has the following limitations: (i) it is tested solely with Flannel as the network overlay for inter-pod communication, and (ii) caution should be exercised if used in combination with component relocations, as the routing rules enabling redirection must be set/added only after the kube-proxy component has completed the (re)configuration of all the required rules that need to change due to the component relocation.

### *6.1.3 Integration with Karmada*

Custom propagation policies provide a mechanism for controlling how resources are distributed and synchronized across Kubernetes clusters within a Karmada deployment. While Kubernetes offers default propagation policies out of the box, they may not always meet the unique needs of custom resource types. Custom propagation policies fill this gap by empowering users to define their own rules and strategies for resource propagation, ensuring greater flexibility and adaptability.

Karmada simplifies the process of creating and applying custom propagation policies through its intuitive API and powerful control plane. Users can define propagation policies based on various criteria such as resource type, namespace, labels, or annotations, tailoring them to their specific use cases. By leveraging Karmada's custom propagation capabilities, organizations can achieve finer-grained control over resource distribution and synchronization, optimizing performance and resource utilization.

Custom propagation policies offer several benefits over traditional, one-size-fits-all approaches. They enable organizations to tailor resource management strategies to their unique requirements, resulting in improved efficiency, resilience, and scalability. By embracing custom propagation policies, users can unlock the full potential of Karmada and Kubernetes, driving innovation and accelerating the pace of development.

When implementing custom propagation policies, it is essential to follow best practices to ensure reliability and consistency. This includes defining clear objectives and criteria for resource propagation, testing policies in controlled environments, and monitoring their performance in real-world scenarios. Additionally, it is important to document policies thoroughly and involve stakeholders in the decision-making process to ensure alignment with organizational goals and objectives. In MLSysOps, we integrate our custom application description to Karmada's custom propagation policies and, as a result, we can deploy multi-tier microservice-based applications using a custom CRD, propagated across the continuum accordingly.

Karmada is used as a mechanism for custom resource propagation to specific cluster(s) to implement the initial application deployment across multiple Kubernetes clusters. Subsequently, the selected Cluster Controllers (Fluidity instances) capture new operations (e.g., addition, removal, or update) of a given forwarded application description. It is the Cluster Controller's responsibility to deploy the components based on (i) flags (hints) in the application description and (ii) its custom policy implementation. The aforementioned hints indicate the cluster that each component needs to be deployed (by providing the respective cluster's unique identifier). It is the (custom) policy that decides the suitable component-to-node mapping.

Figure 15 provides a high-level visualization of the implemented solution. A user submits an application description via the Northbound API, which is received by the Continuum Agent running on top of Karmada. This agent's controller component processes the input and collaborates with its ML-based intelligence module to produce a new deployment plan. This plan is propagated downward through the Cluster Agent and then to the Node Agent, each of which may re-evaluate and adjust the plan based on localized telemetry and their own intelligence modules. Communication between agents uses the MLSysOps Agent Protocol, while interactions with ML models occur via the ML Connector API. Kubernetes, with the respective Kubelets, are responsible for executing deployment and reconfiguration decisions at the cluster and node levels. The policy plugins are explained in detail in Section 8.3.

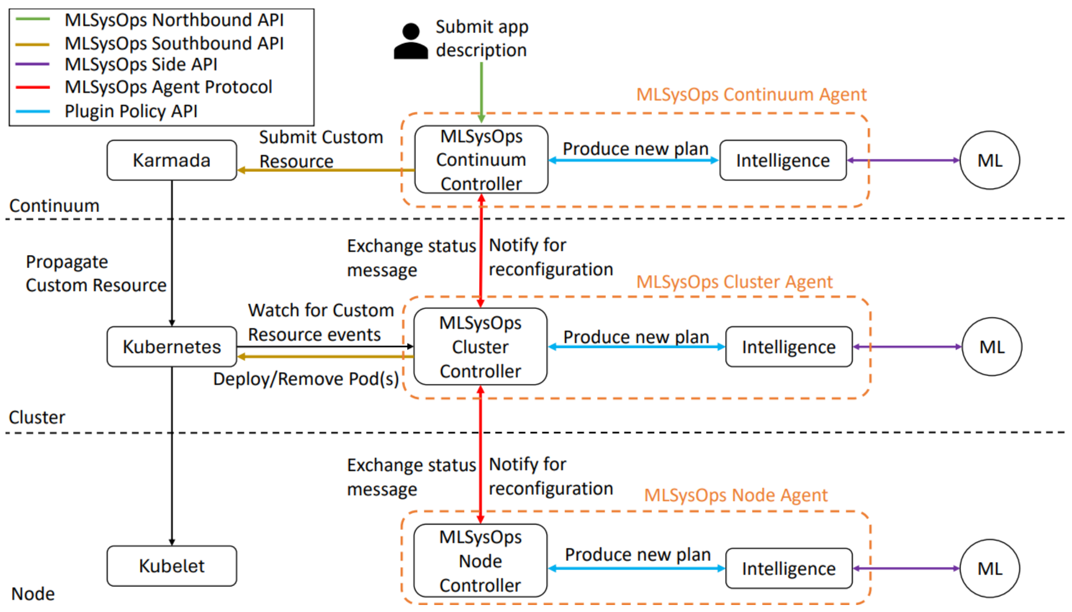


**Figure 15. Deployment mechanism architecture.**

## 6.2 Isolation by combining lightweight virtualization techniques

Running applications in the cloud has changed the way users develop their code, package and deploy their software components. During the past decade, applications were deployed in the cloud using conventional Virtual Machines (VMs) following the Infrastructure-as-a-Service (IaaS) model. Users would typically choose the setup of their virtual hardware, install their preferred OS, and deploy their application / service on top of that VM.

However, quite recently, the community has given rise to other approaches [21], [22] which were quickly adopted by cloud vendors, towards solutions that follow the paradigm of Platform-, Software-, and Function-as-a-Service (PaaS, SaaS, and FaaS respectively). These approaches offer performance and flexibility improvements over IaaS by decoupling the application from the infrastructure. Providing a common OS stack, maintained by the provider and optimized for the specific hardware it is running on, is much more efficient than exposing a generic virtual hardware interface. Additionally, users seek to maximize the number of requests handled while minimizing request/response latency. Apart from the cloud paradigm, Edge computing is slowly adopting these modes of operation, especially in the context of IoT and 5G [23].

Deployment options become far more compliated in the context of MLSysOps as applications need to be executed throughout the continuum, including cloud nodes, Edge nodes or even mobile / IoT nodes. Keeping the benefits of cloud application deployment while accommodating their execution in diverse enviromnets presents challenges. In MLSysOps we address that by employing hybrid execution modes, enabling isolation while preserving the ease-of-deployment features of the cloud paradigm. In what follows, we try to position current state-of-practice, as well as state-of-the-art and how in MLSysOps we combine techniques and mechanisms to facilitate the execution of applications across the computing continuum.

In the IaaS case, the burden of orchestrating and optimizing the system's software stack running on top of virtual hardware is passed to the user, while in the other cases, the vendor exposes a customized interface tailored to the application / service offered. The cloud-native [24] concept emerged from this trend as a need to reduce bloated interfaces and abstractions that introduced significant overhead for application deployment and execution. Containers have played an important role towards cloud-native embracement; they have revolutionized deployment by facilitating application packing and dependency tracking, and reducing the overheads of execution; however, this comes at the cost of security and isolation [25]. As a result, cloud vendors fall back to generic virtualization techniques: Microservice offerings are essentially VMs running on the vendor's custom systems stack, exposing a language runtime, a specific service such as a Database Management System (DBMS), a LAMP stack or just container host-side systems software. For instance, to provide a secure Serverless environment where users deploy their functions at will, cloud providers either: (a) spawn a VM per tenant, install their Serverless backends there, and keep it hot while the user submits functions to be executed; (b) spawn VMs which host containers per tenant, with the necessary software installed, and execute the user function there; or (c) spawn microVMs [26] per tenant where isolation is provided by the microVM monitor [27]

Figure 16 captures a snapshot of the traditional mode of execution for a generic VM on Linux/KVM using a standard user-space Virtual Machine Monitor (VMM). The Kernel-based Virtual Machine (KVM) module in the Linux kernel interfaces with the VMM, which essentially handles privileged operations (VMExits). So, when a privileged operation needs to be executed in the guest, the system traps it in the host's kernel-space (KVM), which, in turn, delivers this event to the monitor in user-space. Upon completion, the execution returns to the kernel, which, in turn, kicks the guest's vCPU via a VMEnter. This process is a design choice: for instance, QEMU supports a full operating system stack, emulates several architectures / features, and makes perfect sense for this code to be in user-space.

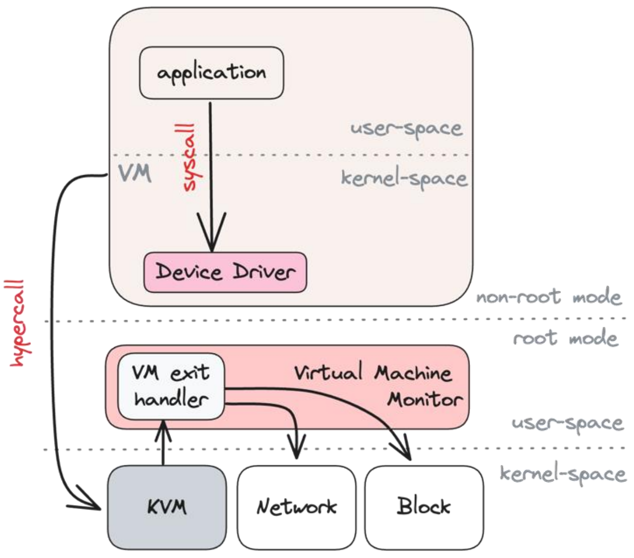


**Figure 16: Virtual Machine running on a generic user - space VMM on top of KVM.**

On the other hand, in the context of Serverless Computing [28], cloud-native applications, and lightweight execution, this process seems too complicated. The system does not need to hand over the event to user–space since the only operation needed can be performed in the kernel (network, storage I/O, etc.). The decoupling of the control path from the data path while performing high-performance I/O is the vhost approach [29].

Apart from optimizing the data path, researchers have done considerable work to minimize the overhead of VM spawning and execution. Several minimalistic approaches have been proposed regarding the systems software stack to facilitate fast and secure application execution in the cloud. For instance, Unikernel as Processes [30] describes a specialized VMM with several backends (e.g., seccomp, KVM, muen, etc.) that minimizes the attack surface by limiting the interface with the underlying layers. Cloud Hypervisor is an open-source VMM that runs on top of KVM [31]. Amazon's Firecracker [27] is a fork of rust-vmm [32], a lightweight VMM, built to deploy microVMs, which feature enhanced security and workload isolation over traditional VMs.

Most of these approaches use the Linux kernel as the guest OS. This implies that although the user just needs to execute a function, the cloud provider must spawn a Linux kernel guest, or a container, from scratch and then run the function in this environment. More importantly, in all the above cases, when a *VMexit* happens, the mode of execution still needs to be passed on to the monitor (first mode-switch) to service the exit and then back to KVM to resume the guest (second mode-switch).

Simplicity is key when designing an application execution stack: users want to run their code fast and get a result back. They do not care where the code will run if there is reproducible, fast, and secure execution. To this end, lightweight virtualization appears mutually beneficial to cloud vendors and users: the former increase resource utilization by consolidating more tasks to nodes; the latter enjoy fast service response times and (potentially) lower-cost services.

In MLSysOps, as mentioned above, we adopt a hybrid approach to balance the trade-offs between lightweight application execution and workload isolation/security. To this end, we enable the deployment of workloads in various execution modes, such as generic containers, sandboxed containers, and unikernels, using the appropriate virtualization mechanisms for each mode.

### *6.2.1 Efficient sandboxing of containers on edge nodes*

#### *6.2.1.1 Containers and their benefits*

Containers are lightweight, self-contained execution environments that encapsulate applications and their dependencies, providing a consistent and reproducible runtime environment. They enable the packaging of software in a manner that allows it to run reliably and consistently across different computing environments, including any computer hardware, infrastructure, or cloud environment. Achieving this versatility is made possible through a combination of operating system-level virtualization and resource isolation techniques. Containers abstract away the underlying infrastructure, allowing applications to be developed and deployed with minimal concern for specific hardware or software configurations. Unlike virtual machines, containers do not require a guest OS in each instance, resulting in smaller, faster, and highly portable units that can be executed on desktops, traditional IT systems, or in the cloud. Figure 17 shows a high-level overview of the node-level flow for a container spawn. The kubelet interfaces with the container runtime through CRI services like the image and runtime service, which connect to containerd. containerd-shim ensures containers run independently, while runc handles low-level container execution in compliance with the OCI specification.

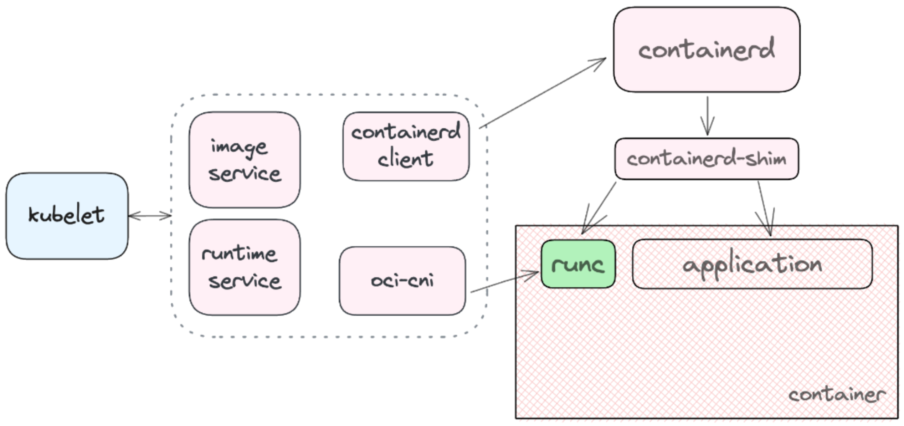


**Figure 17: High-level overview of generic container spawning in a k8s environment.**

#### *6.2.1.2 Limitations of containers*

Despite their numerous advantages, containers have certain limitations, particularly in terms of security and isolation. Although containers provide a level of isolation by leveraging operating system features, they still share the underlying operating system kernel. This shared kernel introduces potential security risks, as a compromise within the kernel could impact the security and integrity of all containers running on the same host. Additionally, containers may not provide sufficient isolation for certain sensitive workloads or applications with strict security requirements. Furthermore, containers may face challenges when handling specific types of workloads, such as those with strict real-time requirements or resource-intensive applications that demand fine-grained control over hardware resources. In MLSysOps, container deployments are considered only in single-tenant setups – when the underlying hardware is not shared by multiple stakeholders, the risks mentioned above are not critical.

#### *6.2.1.3 Sandboxing and its importance*

In multi-tenant setups though, containers present these limitations in terms of security, isolation and resource control; to mitigate this in MLSysOps, we employ container sandboxing. By encapsulating containers within microVMs, each with its dedicated kernel instance, stronger isolation and security are achieved. The use of microVMs ensures that any compromise within a specific microVM remains contained, mitigating potential security risks from the shared underlying kernel. Moreover, container sandboxing resolves the insufficient isolation for sensitive workloads by providing a secure and isolated execution environment for individual containers within each microVM. Figure 18 presents the high-level concept of container sandboxing using kata-containers. At the top, tools such as ctr, nerdctl, and crictl serve as interfaces for clients that issue container management commands. These tools interact with containerd, which is responsible for pulling images, creating containers, and supervising their lifecycle. Once a container is created, containerd uses containerd-shim to spawn and manage the container process independently. At the runtime level, the OCI runtime (runc) is invoked to create the container environment, applying the necessary namespaces and cgroups.

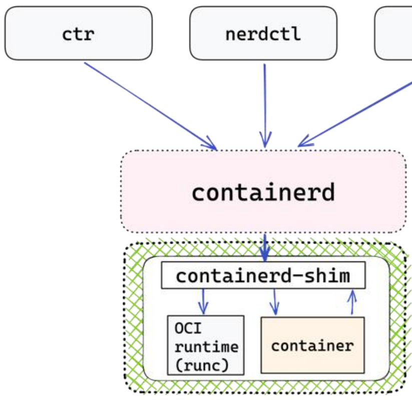


**Figure 18: Container sandboxing.**

#### 6.2.1.4 *Sandboxing in Edge Devices*

Edge devices often have constrained processing power, memory, and storage capacity, which can impact the performance and scalability of applications deployed on them. Secondly, the heterogeneous nature of edge devices introduces complexities. Different devices may have varying hardware capabilities, operating systems, and available hardware accelerators. This requires developing specialized software that can seamlessly run across diverse edge devices, accommodating their unique characteristics. Developers must account for compatibility issues, adaptability, and the need for device-specific optimizations. In addition to these challenges, multi-tenancy adds another layer of complexity. When multiple deployments share the same edge devices, isolation becomes crucial to ensure the security and integrity of each application and its data. Sandboxing techniques, such as microVMs, can be employed to provide isolated execution environments for individual deployments, mitigating the risk of interference or unauthorized access. However, deploying multiple sandboxed containers in a single edge device can be challenging due to resource limitations and potential application conflicts.

### *6.2.2 Sandbox container runtimes*

Kata Containers [33] is an open-source project that aims to provide a secure and lightweight runtime environment for containerized applications. It leverages hardware virtualization technologies to offer strong separation between containers while maintaining the performance advantages of lightweight containers. Launching a container in a micro VM, managed by a hypervisor, with its own kernel and root file system, ensures enhanced security and isolation, defending the application from remote execution, memory leaks, or unprivileged access, as well as protecting the host in case of untrusted or untested programs.

Kata-containers have been designed as a modular system, using various components, each with a unique task to actualize an end-to-end container spawn. The runtime is compatible with the containerd runtime shimv2 architecture and complies with the Open Container Initiative (OCI) runtime specification, making it Kubernetes-compatible with either CRI-O or the equivalent containerd implementation. A single runtime shim is also sufficient to manage all the OCI containers of an entire pod. Kata utilizes a guest agent, a daemon process, to establish robust communication between the guest and the host through a vshock socket operating on a ttRPCbased protocol. This approach enables the exchange of container management commands as well as carry the standard I/O streams between a container and its manager, eliminating also the need for multiple runtime calls. Such a structure ensures that Kata remains agnostic to the sandbox mechanisms, allowing a plethora of different hypervisors and different types of containers, such as WebAssembly (Wasm) or Linux, suiting different requirements and preferences.

There are currently two versions of the kata-containers runtime: the default runtime is written in Golang, while runtime-rs, which is implemented in Rust, is under development and was created because of a need for better container startup speed, resource consumption, stability, and security. In the context of the MLSysOps project, we ported AWS Firecracker [34] to the Rust runtime. While the Go runtime features plenty of different hypervisors (ACRN, Cloud Hypervisor, Firecracker, and QEMU), the Rust runtime has the built-in option of Dragonball, a hypervisor embedded in the runtime.

Our implementation provides the ability to generate a working Firecracker hypervisor instance that can spawn a VM and subsequently containers with all the needed features of a VMM together with full networking functionality, the option to jail the sandbox and its resources along with the capability to hot plug block devices, by patching them to pre-inserted dummy drives, as Firecracker does not support filesystem sharing. Additionally, we have integrated the vAccel as a part of kata-containers, both embedded in the runtime and as a standalone service, providing the option for extra computational power if the workload requires it, through hardware acceleration, without involving direct hardware/device access.

### *6.2.3 microVM optimizations*

The virtualization layer is a vital component of the software system stack, especially in multi-tenant edge computing. Nonetheless, this comes at the cost of consuming more resources and adding overhead to the overall execution of a workload. Consequently, the virtualization layer must be as lightweight as possible while not compromising the isolation and fair execution among the different tenants. Under these circumstances, traditional system-level virtualization is the only feasible solution to provide strong isolation. In system-level virtualization, the VMM creates an entire virtual machine, and a separate OS runs inside. In most cases, the VMM is a user-space application that interacts with the host OS to create and manage the VMs. As a result, researchers and engineers focus on reducing the overhead that the VMM induces and optimizing I/O performance.

In this context, microVMs have emerged. Instead of using an entire OS inside a VM, microVMs use a minimal kernel and only the necessary components to execute applications. The lightweight VMs are much more scalable since they can quickly boot and shut down while reducing resource consumption. Such virtual machines also require fewer functionalities from the underlying hypervisor. As a result, emerging hypervisors only support the necessary functionality, specifically for microVMs, reducing their codebase and the overhead of setting up the environment for the VM.

### *6.2.4 Virtual Machine Monitor*

A hypervisor can be simple and minimal; for example, the solo5 unikernel [35] highlights I/O in hardware virtualization. When a privileged operation occurs in the guest, the system traps (VMExit) to the host kernel (KVM), then returns control to the user space monitor. The monitor handles the request and resumes guest execution with KVM. In lightweight virtualization, this switch from kernel to user space adds overhead. For instance, during a network I/O request, control returns to the user space monitor to process it, which then makes a system call, returning control to the host kernel.

To entirely remove this overhead, we designed and implemented HEDGE, a minimal and simplistic VMM that resides inside the Linux kernel interacting directly with KVM without any intervention from the user space. HEDGE is a simple dispatch handler in the kernel that services a guest's needs. It provides an interface to the

KVM API, a VM execution environment for each of the VMs spawned, generic device handling (network & block), and a management layer to perform basic VM operations (create, destroy, dump console, etc.).

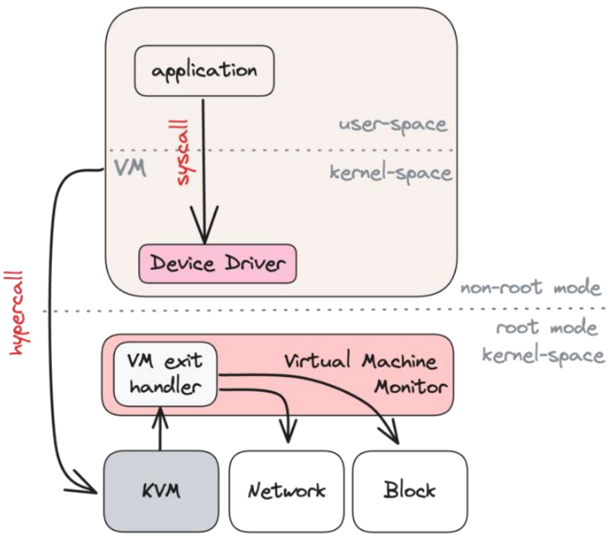


**Figure 19: A unikernel running as a VM on HEDGE.**

A significant challenge in this approach is that KVM targets user space processes, providing an API through file descriptors. Additionally, using KVM's API from within the kernel is not feasible because most necessary functions are exclusively used inside KVM. A solution to this issue involves creating glue code, which consists of wrappers for KVM functions, between HEDGE and KVM to expose all required functionality. Consequently, two small patches are needed to enable the use of HEDGE. In other respects, HEDGE operates similarly to most user space VMMs. For example, as in the case of QEMU/KVM, each VM is associated with one kernel thread, which implements the vCPU. The thread's life cycle begins when HEDGE receives a request to spawn a new VM and handles all privileged operations (VMExits).

One design choice worth noting is that the new kernel thread will have its own memory mappings (mm struct). Moreover, HEDGE allocates virtual memory to serve as the guest's memory and maps it to a virtual address within the newly created kernel thread's memory area. As a result, the kernel thread mimics a user space process, creating the impression for KVM that it is being used from user space. An important aspect of HEDGE's design is minimizing the noise VMMs generate while handling I/O requests. Performance is a primary goal of this project; to achieve this, the guest needs to run as uninterrupted as possible. Besides removing the mode switch overhead, HEDGE handles I/O requests with minimal overhead. The simple and minimal hypercall Application Binary Interface (ABI) from Solo5 aids in this regard. Network packets are formed by the guest, and when an I/O request occurs, HEDGE simply forwards the frame to the appropriate network interface. Receiving packets follow the opposite route. Each guest is associated with a virtual interface (TAP), and raw ethernet sockets are used to receive and send network packets on behalf of the guest.

Regarding block device support, HEDGE leverages the device mapper (DM) functionality to create a virtual block device mapped to a physical device. Using the block read/write hypercalls from Solo5 ABI, the guest makes I/O requests, which are translated into read/write calls in the kernel to the DM block device. The plan includes adding support for VirtIO to host additional unikernel frameworks and even the basic functionality of a Linux guest.

HEDGE provides a management interface, which currently is minimal and handles basic VM operations such as start, stop, etc. It can be managed both locally (user space) or remotely, making it suitable for situations where user space access is limited, such as edge nodes. Management can be done using the following commands:

- Load: Loads a module (VM image) and prepares it for deployment.
- Start: Executes the selected module.
- Stop: Stops the execution of a VM.

Users can select the block or net device, specify command line arguments for the guest, and view the guest's console output. They can also access statistics like boot and setup times, I/O operations, and general HEDGE stats such as the number of VMs and memory usage. The management interface can be accessed locally through procfs in the Linux kernel. When HEDGE is loaded, it creates two files and a directory under /proc:

- /proc/monitor: Controls the hypervisor and virtual machines.
- /proc/vmcons/VMID: Stores virtual machine output.
- /proc/vmstats/VMID: Contains stats for the hypervisor and virtual machines.

Alternatively, the network management interface allows interaction via UDP commands and file transmission over TFTP.

In the context of the MLSysOps, HEDGE is enhanced to support generic Linux distributions. We also implement basic virtio driver functionality for network and storage. As our focus is on lightweight execution, we port and successfully execute unikraft [36], rumprun [37] solo5 [35], and OSv unikernels [38]. Additionally, we integrate HEDGE with our custom container runtime to support the secure and efficient end-to-end deployment of workloads through the MLSysOps orchestrator to edge devices.

### *6.2.5 Single-application kernels and Unikernels*

Unikernels represent an innovative technology that offers exceedingly fast boot times, reduced memory consumption, enhanced performance, and superior security compared to containers. As such, unikernels are particularly beneficial in mixed and heterogeneous environments, notably for microservices and function deployments. However, they are often criticized for their complex build and deployment processes. To address this challenge, we introduce Bunny[4], a tool designed in the context of MLSysOps to provide a user-friendly build process for unikernels, making them as accessible as containers. The primary objective of Bunny is to offer a streamlined and unified approach to building and packaging unikernels. With Bunny, users can seamlessly construct applications as unikernels and package them as OCI images inclusive of all necessary metadata to run them with Urunc.

[4] https://github.com/nubificus/bunny

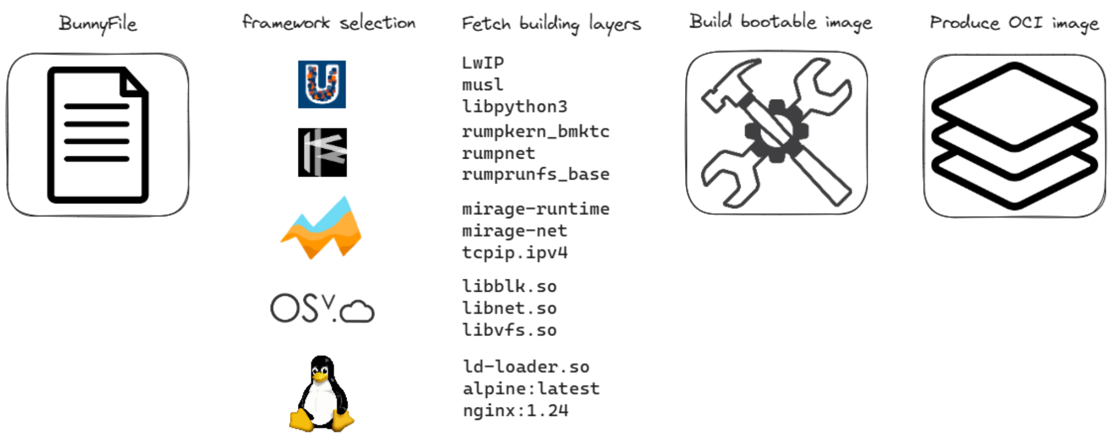


**Figure 20: Building and packaging an application with bunny as an OCI-compatible container image.**

In addition to unikernels, bunny can be used to build single-application kernels: a stripped down version of an application, bundled with its dependencies in a minimal, distroless rootfs, containing the linux kernel to boot and the binary to execute the application. Figure 20 illustrates the process of building and packaging an application using bunny.

Currently, bunny supports GNU/Linux for x86_64 and arm64 architectures. The main goals of bunny are to build and package unikernels and single-application kernels as OCI images. Packaging these binaries is agnostic of the frameworks and, hence, bunny can be used for any unikernel framework or similar technologies.

**bunnyfile: a Dockerfile for slim application kernels and unikernels**

The first step to achieve cloud-native integration of these kind of applications is to package them into an OCI-compatible container image. To achieve this, we follow standard practices to describe the application in a structured file format (YAML). The structure of a bunnyfile is shown in Table 11.

**Table 11: Bunnyfile Structure**

```
#syntax=harbor.nbfc.io/nubificus/bunny:latest   # [1] Set bunnyfile syntax for automatic recognition from buildkit.
version: v0.1                                   # [2] Bunnyfile version.

platforms:                                      # [3] The target platform for building/packaging.
  framework: unikraft                           # [3a] The unikernel framework.
  version: v0.15.0                              # [3b] The version of the unikernel framework.
  monitor: qemu                                 # [3c] The hypervisor/VMM or any other kind of monitor.
  architecture: x86                             # [3d] The target architecture.

rootfs:                                         # [4] (Optional) Specifies the rootfs of the unikernel.
  from: local                                   # [4a] (Optional) The source or base of the rootfs.
  path: initrd                                  # [4b] (Required if from is not scratch).
  type: initrd                                  # [4c] (optional) The type of rootfs (e.g. initrd, raw, block)
  include:                                      # [4d] (Optional) A list of local files to include in the rootfs
    - src:dst

kernel:                                         # [5] Specify a prebuilt kernel to use
  from: local                                   # [5a] Specify the source of a prebuilt kernel.
  path: local                                   # [5b] The path where the kernel image resides.

cmdline: hello                                  # [6] The cmdline of the app.
```

The relevant fields are described in Table 12.

**Table 12: Bunnyfile fields definition**

| | Description | Required | Value type | Default value |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 1 | instruct Buildkit to use bunny for parsing this file | yes | buildkit directive | - |
| 2 | API version of bunnyfile format. Current version is v0.1 | yes | string in major version format (vX) | - |
| 3 | Information about targeting platform | yes | - | - |
| 3a | The unikernel/libOS to target | yes | string | - |
| 3b | The unikernel/libOS version | no | string | latest |
| 3c | The VMM or any kind of monitor, where the unikernel will run on top | no | string | framework-dependent |
| 3d | The target architecture | no | string | bunny's host arch |
| 4 | Instructions about unikernel's rootfs | no | - | - |
| 4a | The base image or an image/location that contains a rootfs | no | "scratch", "local", "OCI image" | "scratch" |
| 4b | The path relative to the from field where a rootfs file resides | yes, if from == "local" | file path | - |
| 4c | The type of the rootfs | no | "raw", "initrd" | platform-dependent |
| 4d | Files from the build context to include in the rootfs | no | list of : | - |
| 5 | Information about a prebuilt kernel | no | - | - |
| 5a | The location where the prebuilt kernel resides | no | "local", "OCI image" | - |
| 5b | The path relative to the from field where a kernel binary resides | yes, if from is set | "local", "OCI image" | - |
| 6 | The command line of the application | no | string | - |

*The rootfs field:* The unikernel and libOS landscape consists of various frameworks and technologies, each with its own storage support. Users may need assistance navigating the different storage technologies supported by each framework. Bunny provides a common interface to allow users to easily define the contents of the application’s rootfs. Let's examine this component of the bunnyfile in more detail.

*The from field:* This field tells bunny to create or use an existing rootfs file, similar to Dockerfile's FROM instruction. It can have these values:

- scratch: Create the rootfs from scratch (starting with an empty directory).
- local: Use an existing rootfs file in the local build context.

- OCI image: Use an OCI image as a base for the rootfs or an OCI image with an existing rootfs file.

*The path field:* This field is used when "local" or "OCI image" is specified in the from field. It defines the location of an existing rootfs file, such as a locally created rootfs file (e.g., initrd, virtio-block) or one from another OCI image.

*The type field:* Some unikernel frameworks or similar technologies support a single type of storage, making this field unnecessary. If so, bunny will print the respective error message. However, some frameworks support multiple storage types. For example, Linux can use an initramfs, a virtio-block, or shared-fs as the rootfs. In these cases, this field is useful. Users can select a specific type of rootfs and bunny will build it. Currently, bunny can create the following types:

- initrd: A typical cpio file that guests can use as an initial rootfs.
- raw: In this case, bunny does not build any specific kind of file but instead copies the files specified by the user directly into the OCI image's rootfs. This type is useful for passing the entire OCI image's rootfs to the guest, either through share-fs or devmapper.

*The include field:* In this field, users can define the files from the local build context they want to include in the rootfs. It is equivalent to the COPY instruction in Dockerfile. The field accepts a list of entries with the following format:

- `<path_in the_local_build_context>:<path_inside_the_rootfs>`

### Example

#### *Using a Bunnyfile*

To append container runtime annotations to an existing Unikraft OCI image, we can define the bunnyfile as follows:

```
#syntax=harbor.nbfc.io/nubificus/bunny:0.0.2
version: v0.1
platforms:
        framework: unikraft
        monitor: qemu
        architecture: x86
kernel:
        from: unikraft.org/nginx:1.15
        path: /unikraft/bin/kernel
cmdline: "nginx -c /nginx/conf/nginx.conf"
```

In the above example, we specify the following details:

- We intend to use a Unikraft unikernel designed to execute on top of QEMU over an x86 architecture.
- The unikernel binary located at `/unikraft/bin/kernel` will be retrieved from the `unikraft.org/nginx:1.15` OCI image.
- The cmdline for the unikernel is defined as `nginx -c /nginx/conf/nginx.conf`.

Using this configuration file, bunny will fetch the specified OCI image and append the container runtime annotations. The OCI image can then be built with the following command:

```
docker build -f bunnyfile -t urunc/reuse/nginx-unikraft-qemu:test .
```

**Using a Dockerfile-like syntax**

In the case of the Dockerfile-like syntax file, we need to manually specify the annotations needed for the container runtime, using the respective labels. Therefore, to transform the above bunnyfile to the equivalent Containerfile:

```
#syntax=harbor.nbfc.io/nubificus/bunny:0.0.2
FROM unikraft.org/nginx:1.15


LABEL com.urunc.unikernel.binary="/unikraft/bin/kernel"
LABEL "com.urunc.unikernel.cmdline"="nginx -c /nginx/conf/nginx.conf"
LABEL "com.urunc.unikernel.unikernelType"="unikraft"
LABEL "com.urunc.unikernel.hypervisor"="qemu"
```

In the above file:

- We set the `unikraft.org/nginx:1.15` as the base for our OCI image.
- We manually specify through labels the `urunc` annotations.

We can build the OCI image with the following command:

```
docker build -f Containerfile -t urunc/prebuilt/nginx-unikraft-qemu:test .
```

### *6.2.6 Unikernels as containers*

To bridge the gap between containerized environments and unikernels, enabling seamless integration with cloud-native architectures, in MLSysOps, we introduce urunc [39]. Designed to fully leverage the container semantics and benefit from the OCI tools and methodology, urunc aims to become “runc for unikernels”, while offering compatibility with the Container Runtime Interface (CRI). Urunc uses hypervisors to launch unikernels from OCI-compatible images, helping developers and administrators to package, deploy, and manage software with cloud-native practices.

#### *6.2.6.1 urunc: a unikernel container runtime*

Figure 21 presents a high-level architecture diagram of urunc and its interaction with the rest of the components in a generic container environment.

To delve into the inner workings of urunc, the process of starting a new unikernel "container" via containerd involves the following steps:

1. Containerd unpacks the image onto a devmapper block device and invokes urunc.
2. Urunc parses the image's rootfs and annotations, initiating the required setup procedures. These include creating essential pipes for stdio and verifying the availability of the specified vmm.
3. Subsequently, urunc spawns a new process within a distinct network namespace and awaits the completion of the setup phase.
4. Once the setup is finished, urunc executes the vmm process, replacing the container's init process with the vmm process. The parameters for the vmm process are derived from the unikernel binary and options provided within the "unikernel" image.
5. Finally, urunc returns the process ID (PID) of the vmm process to containerd, effectively enabling it to handle the container's lifecycle management.

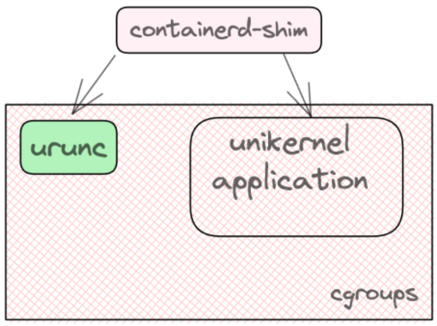


**Figure 21: Running an unpacked container image as a unikernel.**

urunc requires specific metadata in OCI container images  to support unikernels in a containerized environment,. These images must include the unikernel binary, configuration files, and other necessary files, along with metadata specifying how to run the unikernel. The metadata can be included as annotations or a specific file in the container's rootfs.

While urunc-formatted unikernel images are not meant for execution by other container runtimes, they can still be stored and distributed via generic container registries like Docker Hub or Harbor. This allows them to integrate with standard cloud-native workflows for building, shipping, and deploying applications.

Unikernels hold great potential for utilization in serverless deployments. With their lightweight nature, ultra-fast boot times, and singular purpose, unikernels align perfectly with the requirements of short-lived, single purpose serverless functions. By leveraging urunc, developers can seamlessly deploy and manage unikernel-based serverless applications in a cloud-native manner. Combining unikernels and serverless computing enables efficient resource utilization, rapid scaling, and optimal performance, opening possibilities for building highly efficient and responsive cloud-native applications.

More information on urunc, tutorials and hands-on instructions can be found at the urunc documentation website[5].

### *6.2.7 Extending application deployment to Serverless [40]*

Knative [41] is an extension to K8s that specifically targets the needs of serverless workloads. It provides an additional abstraction layer and automation for deploying and managing serverless functions or applications. Knative abstracts away much of the manual configuration and management tasks, offering a developer-friendly experience.

A key feature of Knative is *scale to zero*, meaning that resource allocation for a service occurs after its invocation. Knative will automatically release the service's resources when a service becomes idle. This feature aligns closely with MLSysOps concepts, where, especially at edge sites, resources are scarce; hence, since these resources are allocated on demand, we optimize for cost and resource utilization.

[5] https://urunc.io

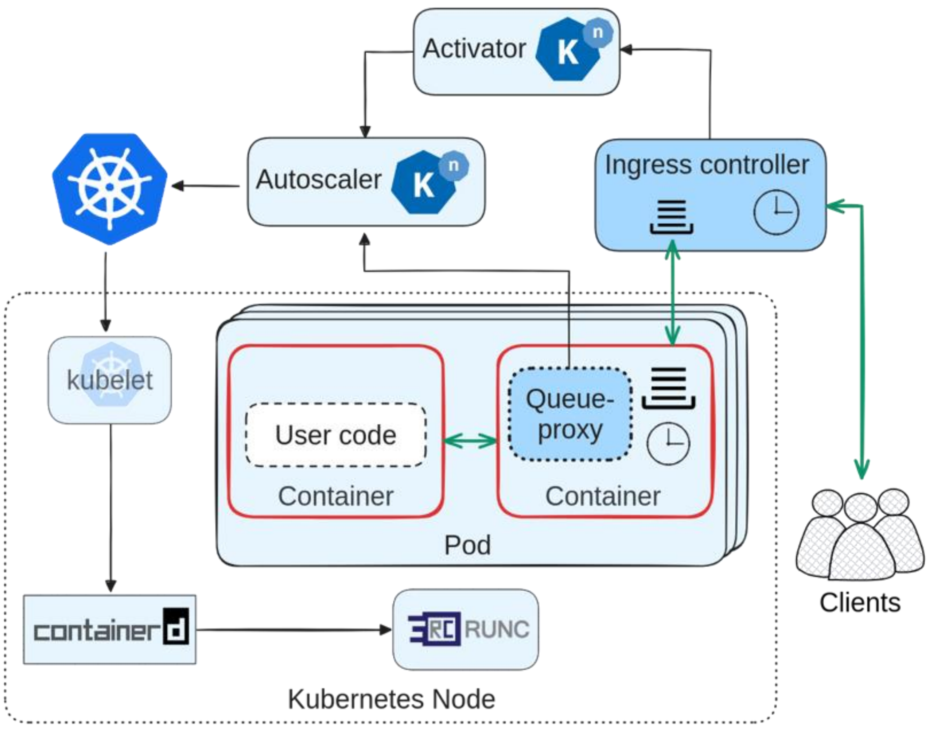


**Figure 22: Knative architecture overview**

Knative introduces the concepts of *serving* and *eventing*; Knative serving facilitates the management and deployment of serverless containers, whereas eventing facilitates event-driven workflows. Figure 22 showcases the high-level architecture of Knative. The basic building blocks, along with their functionality, are:

(a) the *Activator*, which starts or stops pods in response to incoming requests to a Knative *Service*;

(b) the *Autoscaler*, which manages the dynamic scaling of Knative Services based on demand;

(c) the *Queue Proxy*, which coordinates the flow of requests to and from the Knative Service.

It also communicates with the Autoscaler and Activator to manage the lifecycle of pods;

(d) the *Controller*, which manages the lifecycle of Knative services, including creation, updating, and deletion;

(e) the *Metrics Server*, which collects and exposes metrics related to the performance and behaviour of Knative services.

Knative's threat model [42] is a work-in-progress document developers consult to determine the severity of a reported exploit. Based on the tenancy model currently in use in the K8s ecosystem [43], Knative assumes a single tenant per cluster (*Clusters as a service*) as its security model. The project does not consider itself secure for the other tenancy options: *Namespaces as a service*, or *Control planes as a Service*. This option severely limits the applicability of Knative, as, for instance, a serverless provider willing to use Knative should expose a dedicated control plane for each tenant willing to submit arbitrary workloads to the underlying infrastructure.

The mitigation for allowing users to submit untrusted workloads to a Knative cluster is enhanced isolation mechanisms. Container runtimes, such as *Kata-containers* and *gVisor*, provide such mechanisms by using VM-based isolation for containers.

However, the VM-based sandboxing mechanisms induce significant overhead in the instantiation times of containers. Slow spawn times reduce the responsiveness and the scalability of a serverless system. As a result, there seems to be a trade-off between isolation and performance in serverless computing. The integration of lightweight, self-contained applications such as unikernels into a serverless system aims to eliminate this trade-off, providing VM-based isolation, while maintaining flexibility for the user (submitting arbitrary code) and fast spawn times.

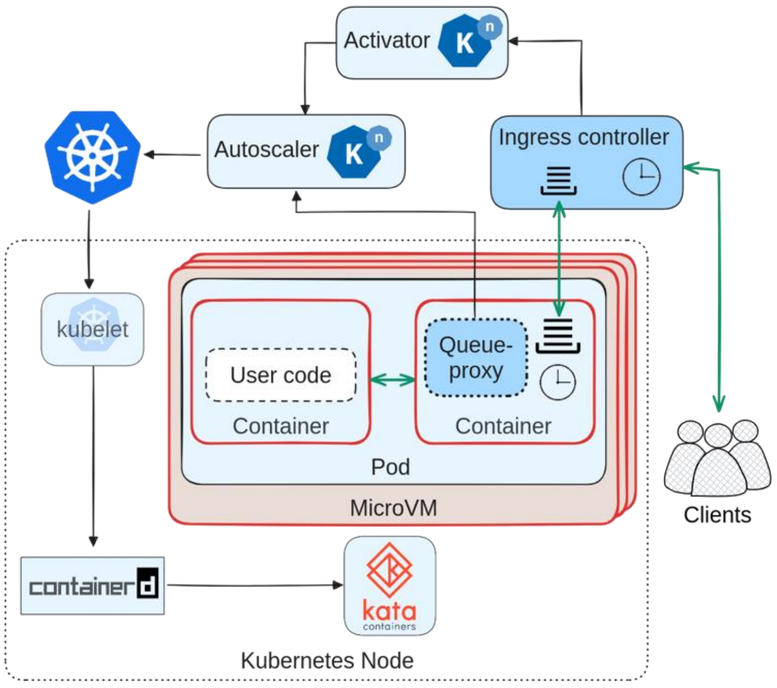


**Figure 23: Knative function pod with a sandbox container runtime (kata-containers)**

In addition, while sandboxing mechanisms like gVisor or Kata-containers reduce the exposed attack surface of the rest of the system, they do not address a potential vulnerability in the Knative stack. In particular, kantive's queue-proxy co-exists with the untrusted workload: the user-submitted workload, packaged in the user-container inside a sandbox. Figure 23 illustrates the coexistence of queue-proxy with user-container. To address the aforementioned potential vulnerability, in the scope of MLSysOps we change the design of Knative, separating the queue-proxy container from the user-container.

A key aspect of urunc is the separation of containers from unikernels. In particular, urunc determines the type of the container using OCI annotations. Depending on the type of the container, urunc either handles the spawning by itself or forwards the request to a generic container runtime (e.g., runc). In that way, urunc can seamlessly integrate with K8s, allowing generic container runtimes to handle platform-specific containers. We visualize the execution flow of spawning containers with urunc over K8s in Figure 24.

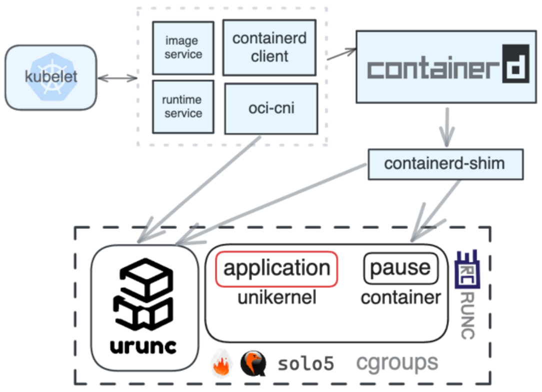


**Figure 24: Integration of urunc with K8s: example unikernel execution flow, including the pause container**

We exploit the inherent separation of containers within the same pod that urunc offers by spawning the queue-proxy container as a generic container. In contrast, the user-container is a unikernel.

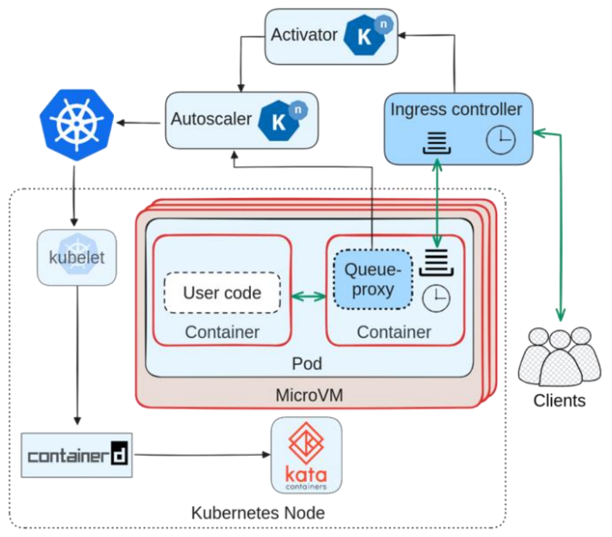


**Figure 25: Knative Function pod over sandboxed container runtimes**

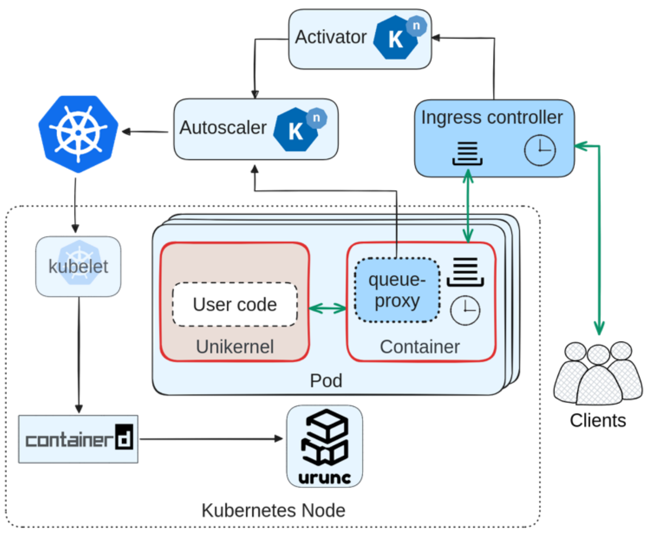


**Figure 26: Knative function pod over urunc**

**Figure 25** and **Figure 26** visualize the isolation boundaries for a Knative Function Pod, deployed on a bare-metal K8s, over kata-containers and urunc.

Such a design facilitates the complete separation between the platform stack and the user's function. Moreover, the user's function executes inside a unikernel, minimizing the attack surface and inheriting strong VM-based isolation from the rest of the stack. As a result, we can significantly reduce the exploiting capabilities of malicious parties.

Although the merits of unikernels have already been quantified in serverless frameworks [44], to the best of our knowledge, there has been no cloud-native integration with popular, state-of-practice tools such as k8s and Knative.

### *6.2.8 Hardware Acceleration*

In the context of edge devices, utilizing specialized hardware components or accelerators to offload and enhance specific computing tasks plays a vital role in boosting performance and efficiency. By leveraging hardware accelerators like GPUs or FPGAs, sandboxed containers can delegate computationally intensive tasks, leading to swifter and more efficient execution. This approach not only improves the overall application performance but also optimizes the utilization of resources on edge devices. Hardware acceleration empowers edge nodes to effectively handle challenging workloads, such as real-time data processing, AI inferencing, or video transcoding. The result is improved responsiveness, decreased latency, and increased energy efficiency, ultimately enhancing the capabilities of edge computing. In MLSysOps, we build and enhance the vAccel framework to enable interoperable hardware acceleration to workloads deployed as container images in various modes of execution: containers, sandboxed containers (in microVMs), and unikernels. More details on the vAccel framework are presented in Section 3.6.

The integration of the vAccel framework with the custom container runtimes is twofold:

- First, we integrate vAccel to the container runtimes we build and enhance (kata-containers) to support hardware acceleration functionality in instances that do not have direct access to hardware accelerator devices.
- Second, we enable vAccel in a multi-tenant Serverless environment using Knative, K8s, and our custom container runtimes.

The integration of vAccel with kata-containers is implemented in both runtimes (Go and Rust). In the Go runtime, we only support AWS Firecracker as the sandboxing mechanism, whereas in the Rust runtime we enable support for all available hypervisors.

### *6.2.9 Enhancing vAccel: Runtime Plugin Selection and Improved Exec API*

The virtualization acceleration framework (vAccel) is crucial for optimizing the performance of various computational workloads by offloading tasks to specialized hardware. To further enhance its functionality, two key components are introduced: hinting [45] for dynamic plugin selection; and robust argument parsing [46] for the exec API operation.

#### *6.2.9.1 Hinting for Smart Plugin Selection*

One of the core aspects of vAccel is its plugin architecture, which allows for flexibility and adaptability in accelerating different workloads. By integrating a hinting mechanism, vAccel can significantly improve its plugin selection process. Hinting involves providing the framework with additional context or metadata about the workload characteristics and requirements. This enables the MLSysOps agents to make informed decisions about which plugin vAccel should utilize, optimizing performance based on the current environment and task at hand.

For instance, in a cloud computing scenario, the hinting mechanism enables vAccel to switch between plugins optimized for CPU, GPU, or FPGA based on the hints from the MLSysOps agent, available resources and the specific demands of the workload. Furthermore, this selection process is dynamic, allowing for real-time changes in plugin utilization as the workload or resource availability shifts. By implementing hinting in vAccel we not only enhance its efficiency, but we also ensure that the most suitable acceleration resources are employed, leading to better overall system performance.

#### *6.2.9.2 Improved Argument Parsing for the Exec API Operation*

Another crucial enhancement for vAccel is the improvement of argument parsing within the exec API operation. The exec API is central to executing tasks within the vAccel framework, and providing a robust and user-friendly mechanism for argument parsing can greatly enhance its usability and functionality.

Enhanced argument parsing allows users to pack and unpack complex arguments with ease when using the exec API operation. This includes support for various data types, nested structures, and validation mechanisms to ensure the correctness of the inputs. By offering comprehensive documentation [47] and intuitive error messages, users can better understand and utilize the API, reducing the likelihood of errors and improving the overall developer experience.

Moreover, improved argument parsing can facilitate better integration with other systems and tools, making it easier to automate and manage workloads within the vAccel ecosystem. This leads to a more seamless and efficient workflow, as users can more effectively leverage the exec API to harness the power of vAccel for their specific needs [48].

## 6.3 Deployment Extensions for Far-Edge Devices

In MLSysOps, the deployment in Far-Edge Devices is achieved by using a service-oriented runtime environment that allows the deployment of workloads on resource-constrained devices along with the abstraction of such runtime to allow its integration with existing orchestration and deployment frameworks, namely Kubernetes. This is achieved using embServe (section 6.3.1), NextGenGW (section 6.3.2), Far-Edge node watcher (section 6.3.3) and the Far-Edge Kubelet (section 6.3.4), as Figure 27 depicts. The layer agnostic workload migration (section 6.3.5) implements a mechanism to allow the transparent migration of workloads between the Far-Edge and the other layers of the continuum. The Far-Edge Proxy Agent (section 6.3.6) introduces Far-Edge Nodes in the MLSysOps Agent framework defined in section 7. embServe and NextGenGW are Fraunhofer Portugal background work, which was enhanced in MLSysOps. Far-Edge node watcher, Far-Edge kubelet, the layer agnostic workload migration and the Far-Edge Proxy are foreground developed in the scope of MLSysOps.

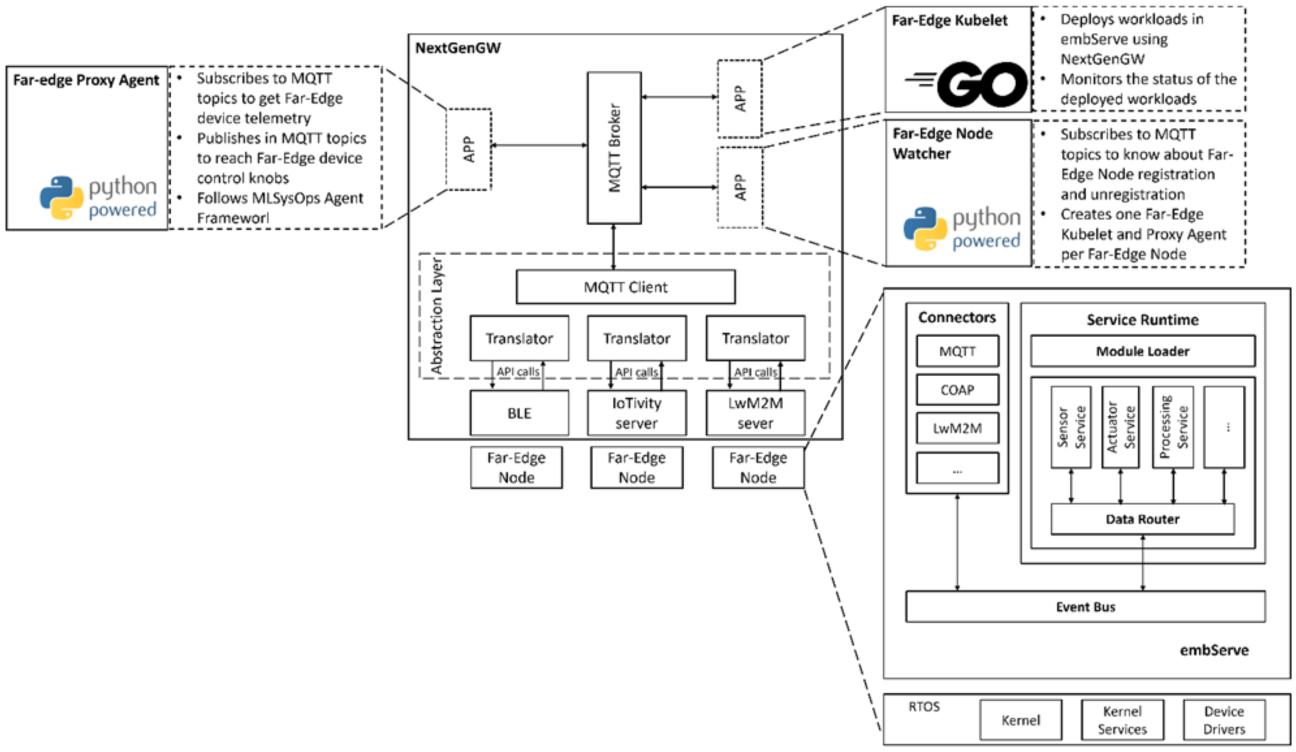


**Figure 27: Deployment Framework for Far-Edge Devices.**

### *6.3.1 embServe*

embServe is a runtime [49] that implements a service-based framework for devices that do not support traditional techniques for software deployment (i.e., containers). The runtime abstracts the application logic as a set of workloads that can be deployed and updated on demand.

Workloads are defined as entities that process inputs using a software component that defines which computations should be performed to produce outputs. They are composed of native code that is dynamically loaded and can interact with the device capabilities (i.e., its sensors, actuators, communication interfaces, etc.) using a well-defined API, and of metadata that contains all the information needed to load the workload correctly and map it to a standard protocol. In the scope of MLSysOps, we use these workloads to perform deployments on Far-Edge devices (resource constrained microcontroller-based devices). Section 6.3.1.1 details how workloads are deployed on embServe and how their status (CPU, memory, uptime and errors) is shared with interested clients (an improvement introduced in the scope of MLSysOps). Section 6.3.1.2 explains how embServe was adapted in the scope of MLSysOps to expose telemetry endpoints and control knobs that allow the monitoring and control of Far-Edge nodes. Section 6.3.1.3 explains how embServe services can be linked between each other to form complex applications. Section 6.3.1.4 details how, in the scope of MLSysOps, tinyML was added to embServe.

#### *6.3.1.1 Workload Deployment and Reporting*

embServe uses the concept of connectors to be generic regarding the supported communication protocols. We use a LwM2M connector to connect the Far-Edge devices to the NextGenGW (see section 6.3.2). To deploy new workloads, the connector uses the standard LWM2M Software Management object (urn:oma:lwm2m:oma:9) [50] and requires that the following steps are executed by the client:

- Create a new instance of the Software Management object;
- Deploy the service artifact using resource Package (ID 1);
    - Artifact will be validated by the runtime;
    - Update State resource (ID 7) is updated according to the result;
- Install the service by executing resource Install (ID 4);
    - Artifact will be saved to the Flash storage of the device;
    - Update State resource (ID 7) will be updated accordingly;
- Active the service by executing resource Activate (ID 10);
    - The service software component will be loaded and started;
    - The Activation State resource (ID 12) will change to 1 if the service is activated successfully.

The client can delete the instance of the Software Management object to remove the workload.

Workloads are packaged using the artifacts definition proposed by the Open Container Initiative (OCI) standard, which means they can be pushed and pulled from OCI-compliant registries providing full integration with Kubernetes image pull mechanisms. We present below an example of an embServe OCI artifact, which sets the "config.mediaType" as "application/vnd.embserve.v1" and uses the standard "application/vnd.oci.image.manifest.v1" with a custom OS in the image configuration.

```
application/vnd.oci.image.manifest.v1+json
{
   "schemaVersion": 2,
   "mediaType": "application/vnd.oci.image.manifest.v1 +json",
   "config": {
      "mediaType": "application/vnd.oci.image.config.v1 +json",
      "digest": "sha256:...",
      "size": 70
   },
   "layers": [{
      "mediaType": "application/vnd.embserve.v1+json",
      "digest": "sha256:...",
      "size": 4885,
      "annotations": {
         "org.opencontainers.image.title": "service. json"
      }
   }],
   "annotations": {
      "org.opencontainers.image.created": "2024-02-08 T20:35:34Z"
   }
}
```

```
# application/vnd.oci.image.config.v1+json
{
   "architecture": "arm",
   "os": "zephyr",
   "variant": "v7"
}
```

Workload CPU and memory usage are important metrics for workload orchestration. embServe reports this metrics using a custom LwM2M object, which includes 4 resources:

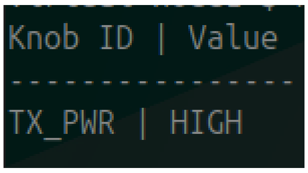
- "CPU Usage" provides the percentage of CPU used by the workload.
- "Memory Usage" provide in Bytes the memory used by the workload.
- "Uptime" provides in milliseconds the workload uptime.
- "Object", provides the identification of the lwm2m object (i.e. the deployed workload) to which the above metrics refer to.

#### *6.3.1.2 Telemetry and Control Knobs*

embServe currently provides telemetry and control knobs for monitoring and controlling its radio communication interface. The telemetry endpoint reports the RSSI values measured by the Far-Edge Node radio, while the control knob allows the client to control the Far-Edge radio TX power.

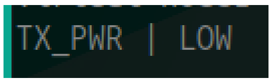
To report the RSSI values, embServe uses the Connection Monitoring LwM2M object (urn:oma:lwm2m:oma:4). The agent observes the property Radio Si*gnal Strength* and reports it as telemetry data. For the control knobs, embServe provides a BinaryAppDataContainer object instance (urn:oma:lwm2m:oma:19) with two instances of the *Data* property for TX and RX communication, similar to a shell interface. This object allows the agent to send commands to the node to interact with the knobs. For example:

- List available control knobs:
  - Send "mlsysops knobs list" to 19/0/0/0
  - Receive response on 19/0/0/1:



```
Knob ID | Value
----------------
TX_PWR | HIGH
```

- Set TX_PWR knob to LOW:
  - Send "mlsysops knobs set TX_PWR LOW" to 19/0/0/0
  - Validate that knob was changed by sending "mlsysops knobs get TX_PWR" to 19/0/0/0
  - Receive response on 19/0/0/1:



```
TX_PWR | LOW
```

One can use the same strategy used for the radio telemetry and control knob for exposing further telemetry data and control knobs, i.e., make use of the existing LWM2M endpoints to expose telemetry data (and when these do not fully meet the telemetry needs, extend them as defined by the standard) and of the BinaryAppDataContainer object to expose further control knobs

#### *6.3.1.3 Service Connections within Far-Edge Devices*

embServe provides a mechanism that allows clients to configure routes for each workload that can be used to create complex data streams between different Far-Edge devices. This allows the usage of the output of a workload as input of another workload in the Far-Edge, increasing the versatility of the runtime.

The LwM2M connector provides a custom object to configure the routes. This object uses a pair of URIs to define the origin and the destination of an event. Some examples can be observed in Table 13.

**Table 13 - embServe routes.**

| Origin URI | Destination URI | Description |
|---|---|---|
| local://1/temp avg | coap://ip:port/endpoint1 | Posts output with key temp avg to CoAP endpoint endpoint1 of the server with the target IP. |
| coap://endpoint1 | local://1/temp1 | Listens on CoAP endpoint endpoint1 and delivers posted data to the input with key temp1 of the service with ID 1. |
| coap://endpoint1 | coap://ip:port/endpoint1 | Listens on CoAP endpoint endpoint1 and forwards the posted data to the CoAP endpoint endpoint1 of the server with the target IP. |
| coap://endpoint1 | mqtt://ip:port/topic1 | Listens on CoAP endpoint endpoint1 and forwards the posted data to the MQTT topic topic1of the broker with the target IP. |

As shown in the table, the routing mechanism allows data forwarding scenarios, where the Far-Edge device is used to convert data between protocols or forward it to other devices.

All the possibilities provided by this mechanism are interesting for MLSysOps since it further expands the capabilities of the Far-Edge layer.

To create a new route, the client should create a new instance of the Data Route object (ID 35001) with the corresponding Origin URI (ID 27200) and Destination URI (ID 27201) resources. The interaction with the LwM2M connector is abstracted by the NextGenGw middleware (see section 6.3.2).

#### *6.3.1.4 TinyML on Far-Edge devices*

TinyML is a current trend that leverages constrained far-edge nodes to perform the inference of simple AI models. Several well-known AI frameworks already support this type of device such as TensorFlow with its Lite Micro variant or Apache microTVM. This trend is relevant for MLSysOps since it makes Far-Edge devices another potential target for deploying AI-based application components.

To enable inference of TinyML models, we added support for Tensorflow Lite Micro inside embServe. Far-Edge workloads interact with system resources such as sensors, network stack or kernel services using an SDK which bridges the workload code with the underlying real-time operating system. Therefore, the Tensorflow Lite Micro runtime was added to the base system. A new set of APIs was added to the embServe SDK, which allows workloads to load models, control the input and output tensors, and invoke the model's inference. A non-exhaustive list of the implemented APIs can be observed below.

```
struct tinyml_tensor {
    uint32_t size;
    enum tinyml_tensor_type type;
    void * data;
    enum tinyml_quantization_type quantization_type;
    struct tinyml_quantization_params quantization_params;
};

const void * tinyml_model_load(const uint8_t * model_buf, uint32_t size, uint32_t
tensor_arena_size);
int tinyml_model_tensor_info(const void * model, bool output, uint32_t index, struct
tinyml_tensor * tensor);
int tinyml_model_input(const void * model, uint32_t index, const struct tinyml_tensor *
tensor);
int tinyml_model_output(const void * model, uint32_t index, struct tinyml_tensor *
tensor);
int tinyml_model_invoke(const void * model);
int tinyml_model_unload(const void * model);
```

To create a TinyML workload, the target model is exported to a .tflite file, which is added to the workload source code as a C-flat buffer. The source code is then compiled in the same way as the normal Far-Edge workloads and uploaded to the OCI registry as an OCI artifact. It is of note that models are only limited by the amount of free memory, which must fit the workload and the tensor arena required for the model's inference.

Far-Edge nodes with TinyML capabilities are identified using a label (extra.resources.fhp/tinyml: true) added to the Node representation in Kubernetes. The MLSysOps framework can use this label to distinguish between AI-ready nodes and use the reported available memory to evaluate if a given model can be deployed to a specific node.

### *6.3.2 NextGenGW*

NextGenGW's [51] objective is to homogenise Far-Edge devices' data models and communication protocols into a single standard data model and communication protocol. This eases the integration of such highly heterogeneous devices in the higher layers of the IoT stack because it hides from the application developers and system integrators the particularities of each data model and communication protocol at the Far-Edge. We use MQTT [52] and Internet Engineering Task Force’s (IETF) Semantic Definition Format (SDF) [53] for such homogenization.

IETF SDF proposes normalising existing standard data models into the single description format that they are proposing. IETF proposes the creation of a new description format because the existing ones are tied to their particular goals and the context in which they are applied, so they cannot integrate all the particularities of the others. The proposed normalisation is achieved through a set of translators that exchange data representation between IETF SDF and other standards without losing each other's particularities.

Considering this, NextGenGW comprises the servers responsible for communicating with Far-Edge Devices. Each server is then associated with a translator, which translates the Far-Edge device data model and communication protocol into the IETF SDF representation bonded to MQTT and vice versa. Currently, the LwM2M server is supported and guarantees the connection with MLSysOps Far-Edge Devices and the embServe runtime that runs on them.

NextGenGW also comprises an MQTT Broker through which third-party applications can interact with the Far-Edge devices following the IETF SDF representation over MQTT. There is no restriction on the place of deployment of the Far-Edge nodes’ servers and their translators, MQTT broker and third-party applications. This means that they can be deployed in the same node or in different nodes as long as the network guarantees the connectivity between the MQTT Broker and the MQTT client on the translators and between the MQTT Broker and the third-party applications.

The IETF SDF binding with MQTT follows the IETF SDF hierarchy in sdfThings, sdfObjects, and sdfProperty, sdfActions and sdfEvents following the specification in [51]. This means that Far-Edge workload deployment and activation are achieved by following these steps:

**Step 1 - Instantiate a new software management endpoint:**

Sending a message with payload *"{"operation": "POST", "data": "{"label": <instance_id>}"}"* to MQTT topic *"<far_edge_identifier>/LWM2M_Software_Management"* creates a new instance of the software management object with the identification <*instance_id*> for the Far-Edge Device identified by <far_edge_identifier>. The "LWM2M_Software_Management" in the MQTT topic is the sdfObject of the sdfThing "<far_edge_identifier>".

**Step 2 – Load workload package:**

Sending a message with the payload *"{"operation": "POST", "data": <pkg_spec_json>}"* to the MQTT topic "*<far_edge_identifier>/LWM2M_Software_Management/<instance_id>/Property/Package"* loads the package specified in <*pkg_spec_json*>, which follows the specification in section 6.3.1, to the software management instance with the identifier <instance_id> of the Far-Edge Device identified by <far_edge_identifier>.

**Step 3 – Install workload:**

Sending a message with the payload {"operation": "POST"} to the MQTT topic *"<far_edge_identifier>/LWM2M_Software_Management/<instance_id>/Action/Install"* installs the wokload specified in instance <*instance_id*> on the Far-Edge Device identified by <*far_edge_identifier*>.

**Step 4 – Activate workload:**

Sending a message with the payload {"operation": "POST"} to the MQTT topic *"<far_edge_identifier>/LWM2M_Software_Management/<instance_id>/Action/Activate"* activates the workload specified in instance <*instance_id*> on the Far-Edge Device identified by <*far_edge_identifier*>.

Regarding the CPU, Memory, uptime and errors reported by the workloads deployed in the Far-Edge, the NextGenGW provides them when the message with the payload "{"operation": "GET"}" is published in the MQTT topic *"<far_edge_identifier>/Software_Package_Monitoring"*. The CPU, memory and uptime are then provided on the response topic option of the MQTT v5 specified in the publish request. The format of the response message is the following:

```
{
   "response_code": <resp_code_id >,
   "sdfObject": {
      "Software_Package_Monitoring":[{
         "label": <sw_pck_moni_instance_id>,
         "sdfProperty": {
            "Sofware_Package": "<worload_id_for_the_metrics>",
            "CPU_Usage": "<cpu_value>",
            "Memory_Usage": "<memory_value>",
            "Uptime": "<uptime_value>",
            "Error": "<errors_value>"
         }
```

```
          }]
      }
}
```

The same strategy is used for the other functionalities and mechanisms provided by embServe, such as the telemetry, control knobs and service connections. The LwM2M objects and resources are translated to SDF objects, properties, action and events, which are bound to MQTT following the mechanisms detailed in [51].

In the scope of MLSysOps, NextGenGW was improved to support sdfObjects with multiple instances, linking between objects and CBOR data format. Essential features for embServe deployments, status reporting and telemetry and control endpoints.

### *6.3.3 Far-Edge Node Watcher*

The Far-Edge Node Watcher is responsible for watching the appearance and disappearance of Far-Edge Nodes from the cluster. It subscribes to the MQTT topics "announce" and "unregister" where NextGenGW publishes such information. In the scope of MLSysOps, new Far-Edge Nodes are shared in the "announce" topic when they are registered in the LwM2M server. They are considered disconnected when they fail to respond to LwM2M heartbeat mechanism. When new nodes are announced in the "announce" topic, the Far-Edge Node Watcher creates the Far-Edge Kubelet and Far-Edge Proxy Agent associated with each node (see section 6.3.4 and section 6.3.6, respectively). When unavailable nodes are identified in the "unregister" topic, the Far-Edge Node Watcher deletes all the workloads running in such node, as well as the Far-Edge Kubelet and Far-Edge Proxy Agent associated with it.

### *6.3.4 Far-Edge Kubelet*

The Far-Edge Kubelet is built on top of the Kubernetes Virtual Kubelet, which is a Kubernetes implementation of the Kubelet that allows the integration of Kubernetes clusters with other APIs. In MLSysOps, we integrate Kubernetes with the NextGenGW API, i.e., with the Far-Edge nodes' abstraction provided through IETF SDF over MQTT.

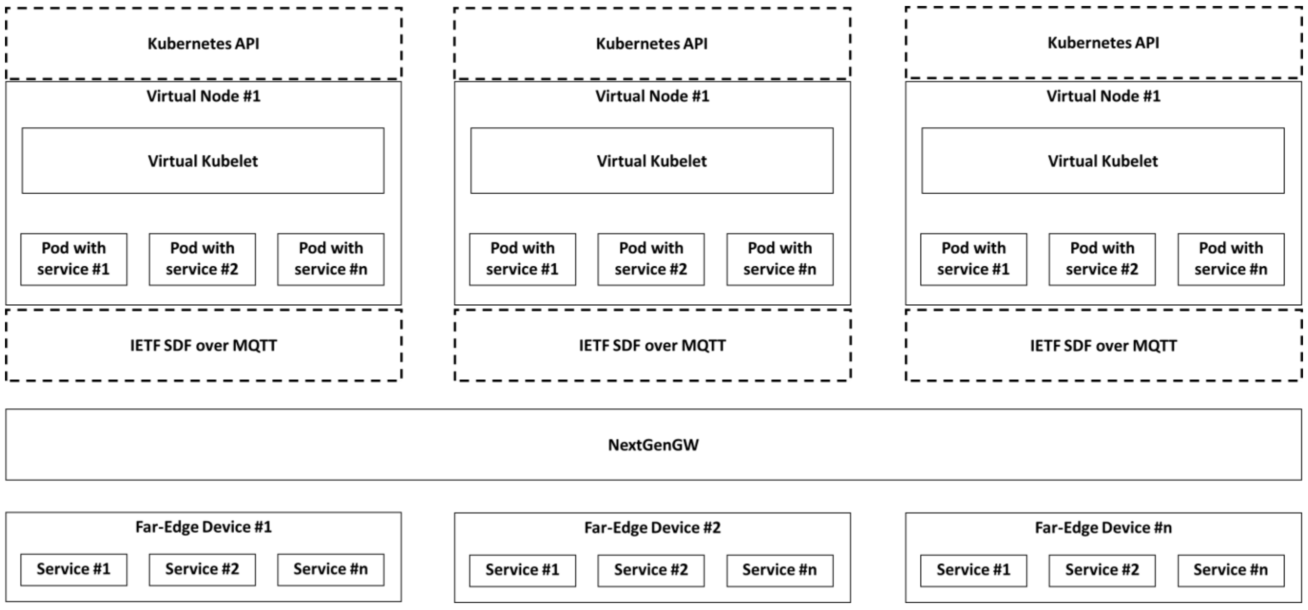


**Figure 28: Far-Edge Kubelet integration with Far-Edge Devices.**

Figure 28 overviews how the Far-Edge Kubelet uses the Virtual Kubelet to integrate the NextGenGW and the Far-Edge Devices running embServe with Kubernetes. The Far-Edge Kubelet is composed of the Virtual

Kubelet, which makes the bridge between the Kubernetes API and the NextGenGW's API. It is associated with the Virtual Nodes, which are the Kubernetes' virtual representation of the Far-Edge Nodes. Each Far-Edge device is virtualised to one Virtual Node, so it is managed by one Far-Edge Kubelet. When a Pod deployment command arrives at the Far-Edge Kubelet, the Far-Edge Kubelet decodes the Pod deployment command and deploys the workload contained on the command in the Far-Edge node associated with it. This is done through the NextGenGW abstraction using the MQTT topics and messages specified in section 6.3.2, which follows the IETF SDF bounding with MQTT.

The Far-Edge workloads are encoded as OCI compliant containers (see section 6.3.1). This means that the Far-Edge Kubelet pulls the Far-Edge workloads from an OCI-compliant registry and decodes them into the JSON format supported by embServe. The workload in this JSON format is the one used to interact with the NextGenGW on the step 2 specified in section 6.3.2.

It is also the responsibility of the Far-Edge Kubelet to report each workload CPU and memory usage, as well as uptime and errors. This is done by interacting with NextGenGW using the procedure defined in section 6.3.2 to get such metrics and proving them on the "GetMetricsResource" function of the PodHandlerConfig struct[6].

### *6.3.5 Layer agnostic workload migration*

As explained in section 6.3.1, the Far-Edge devices do not have the same resources as devices in the higher layers of the IoT stack (Smart Edge, Edge and Cloud), which means that for one workload to be deployed both in the Far-Edge and in the higher layers of the stack, such workload needs to have two versions, one compatible with the Far-Edge (in the MLSysOps scope, the embServe described in see section 6.3.1) and another compatible with the higher layers of the stack. On the other hand, Far-Edge devices are highly heterogeneous regarding their communication protocols and data models. To address this heterogeneity, we use the NextGenGW, which, as explained in section 6.3.2, abstracts Far-Edge devices heterogeneity with MQTT and IETF SDF. However, the resulting communication interface may not be compatible with the intended data consumers or consistent with the interface exposed by the workload version targeting higher layers of the IoT stack. Considering this, if a developer wants to create a Far-Edge version of a workload, which needs to adhere to a consumer-imposed communication interface or to the interface implemented by an already existing workload version targeting higher layers of the IoT stack, the MLSysOps framework should provide a mechanism for interchangeable use of Far-Edge and non-Far-Edge workload versions. Figure 29 presents an example of two workloads on the Cloud, Edge or Smart Edge that use HTTP to interact with each other. When creating a new version of workload B for the Far-Edge, such version might not support HTTP due to restrictions on the Far-Edge device that needs to use NextGenGW as a bridge to the rest of the cluster. Such differences between workload versions should be hidden to provide a seamless way to guarantee that workload migration to and from the Far-Edge occurs without impacting other workloads or clients that interact with it.

[6] https://pkg.go.dev/github.com/virtual-kubelet/virtual-kubelet/node/api#PodHandlerConfig

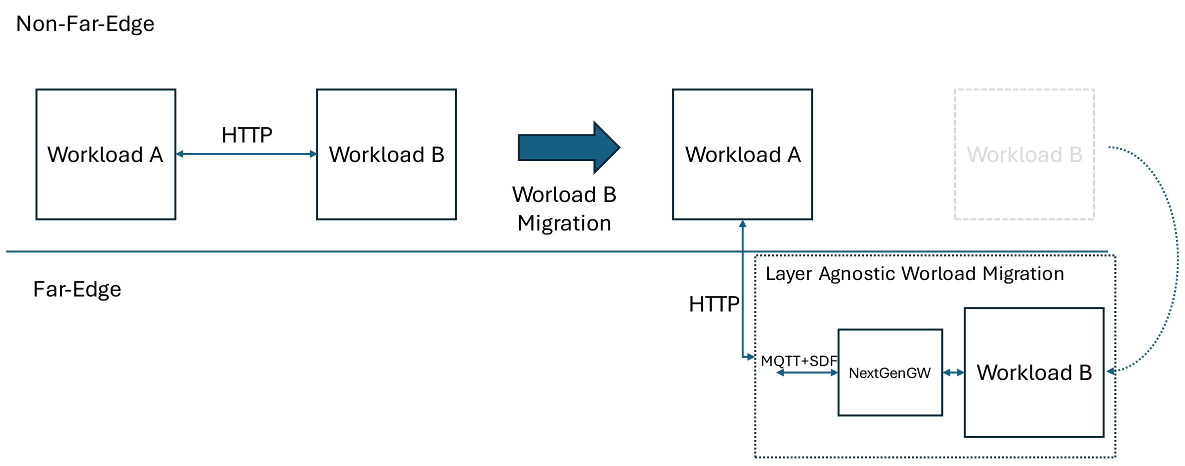


**Figure 29 – Example of Workload Migration to the Far-Edge**

The workload migration to the Far-Edge needs to guarantee that:

- From the orchestration perspective, workloads should behave equally amongst deployment layers (Far-Edge, Smart Edge, Edge and Cloud).
- Workloads running on different layers are not distinguishable by components outside the deployment system.
- The communication interface is the same regardless of the layer the workload is running on and the underlying network stack.

The layer agnostic workload migration strategy depicted in Figure 30 addresses the issue described above and meets the identified requirements. In this strategy, we use a communication interface "Adapter" per Far-Edge workload instance that adapts the MQTT+SDF messages to the interface imposed by workload clients or used by the workload originally designed for the higher layers of the IoT stack. This "Adapter" is associated with the Far-Edge workload it corresponds to by using the annotations in Table 14 on the workload Pod template, part of the workload Kubernetes deployment manifest.

**Table 14 - Communication Interface Adapter Annotations.**

| Annotation | Example value | Description |
|---|---|---|
| `"communication-adapter.fita/image"` | `"fhp/temperature-service-adapter:v0.0.1"` | Image to be used as the communication "Adapter" |
| `"communication-adapter.fita/ports"` | `"[{\"name\":\"http\", \"port:80\", \"protocol\": \"TCP\"}]"` | Ports exposed by the "Adapter" Pod |
| `"communication-adapter.fita/service-name"` | `"Temperature"` | Name of the property exposed by the Far-Edge service in the node's SDF representation |
| `"communication-adapter.fita/pull-policy"` | `"IfNotPresent"` | Image pull policy to be used in the "Adapter" deployment (Optional) |

The Far-Edge workload developer is responsible for implementing the interface "Adapter", which is deployed, and its lifecycle managed, by the Far-Edge kubelet. The Far-Edge kubelet links the "Adapter" with the Far-Edge Workload, so when it deploys the Far-Edge Workload, it also deploys the "Adapter". When it deletes the Far-Edge Workload, it also deletes the "Adapter". The "Adapter" runs as a standalone Pod in a node of the higher layers of the stack, controlled by a Kubernetes deployment with a single replica, whose IP address is the same of the Far-Edge workload instance. With such approach, the workloads or clients interacting with a workload with Far-Edge and non-Far-Edge versions are not aware of the particularities of the Far-Edge version and continue interacting with the workload independently of where it is deployed.

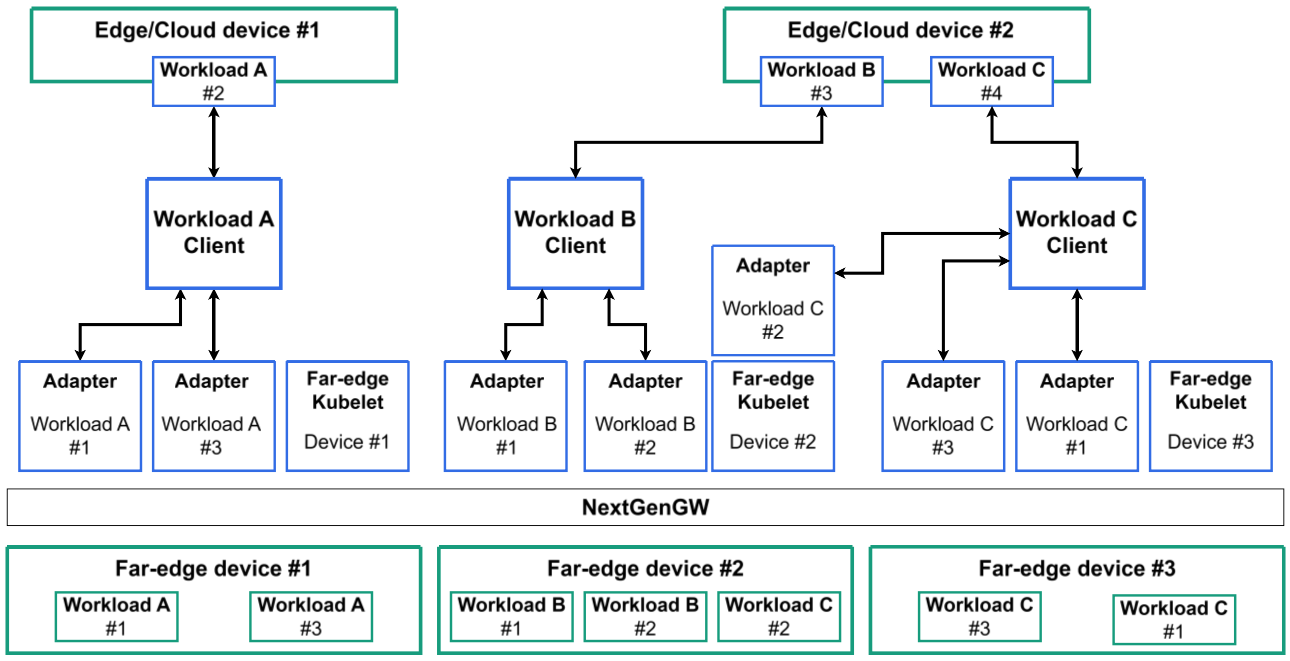


**Figure 30 - Layer agnostic workload migration strategy**

### *6.3.6 Far-Edge Proxy Agent*

The Far-Edge Proxy Agent is, along with the Far-Edge Kubelet and Far-Edge Node Watcher, the MLSysOps NextGenGW's clients. It uses NextGenGW abstraction to reach the control knobs and telemetry endpoints provided by embServe. The Far-Edge Proxy Agent follows the agent framework specified in section 7, collecting the telemetry data from MQTT in the SDF format and bridging it with the MLSysOps telemetry pipeline. The same bridging is done for the control knobs.

Because of the decoupling between the NextGenGW and the applications that interact with it, the Far-Edge Proxy Agent can be running on the same Node as the NextGenGW or in another Node connected to the MQTT broker.

# 7 Trust Management Mechanisms

## 7.1 Zero Trust Management Implementation

The implementation of trust management is split into the following stages:

- initialization and configuration stage
- trust evaluation
- trust update
- zero trust reputation credit management

The central criterion of trust management represents a specialized reputation credit evaluated for each of the corresponding framework agents.

### *7.1.1 Initialization & Configuration stage*

This stage involves the identification of the indices to be used for the reputation credit calculation, together with the setting of the initial values for this credit. It also includes the choice of a particular Machine Learning (ML) model via the MLSysOps Side API.

#### *7.1.1.1 Trust index definition engine*

The engine manages the appropriate indices addition, removal, and subsequent selection to be incorporated into the final equation of the reputation credit. The pool of the potential indices is formed based on the available data flows running from the telemetry, their granularity and on the capabilities of the available ML models (e.g., behaviour-based properties such as package dropping). The choice of either incorporating the whole pool into the computation of the index or a subset of the pool depends on the eventual achievement of a balance between the overall system performance and trust computation demands. The initial choice can be automatically managed utilizing the corresponding penalty terms, such as Lasso regularization.

#### *7.1.1.2 Weights assignment and set equation*

The weights are split into two categories: the weights that are assigned based on the authority and the weights that are assigned based on the results of an ML algorithm. The first category can be managed either based on the hierarchy of the node or based on a predefined configured value. The second category is learned by a dedicated ML model. The final equation for the reputation credit calculation is linear and represents a simple weighted sum where each of the previously chosen indices is weighted by both the weights of the first category and the weights of the second category. The resulting value is normalized to [0...1] range with the higher values associated with higher reputation.

#### *7.1.1.3 Initialize trust and reputation credit*

As soon as the weights and indices are defined, the initial trust reputation credits are computed per each available service throughout the hierarchy of the MLSysOps continuum. This initialization stage relies on the most recent data available in the storage for the telemetry, whose granularity may vary and depends on a particular configuration. The resulting credit values can be either broadcasted to the agents via telemetry or can be requested by different agents separately on an individual basis.

### *7.1.2 Trust Evaluation*

This stage represents an evaluation of reputation credits in a continuous manner after the initialization and configuration are finalized. The dynamic nature of indices and the learned weights in the previous stage allow dynamic evaluation of the corresponding credit values. Based on the rate and granularity of the telemetry flow, together with the corresponding performance of ML models used by trust management, the trust evaluation is performed in a continuous manner with the subsequent updates via telemetry channel.

##### *7.1.2.1 Resilient to distribution drift*

In open-world settings, normal behaviour can shift over time due to factors such as software updates, new user interactions, or evolving system environments. This normality shift can lead to a significant increase in false positives and false negatives, severely impacting model performance. To address this, we used OWAD, a framework designed to detect, explain, and adapt to normality shifts efficiently while minimizing manual labelling overhead. OWAD consists of three key components. First, it performs unsupervised shift detection by transforming model outputs into a calibrated distribution and applying statistical hypothesis testing to detect normality shifts. Next, it conducts shift explanation by identifying the most influential samples responsible for the shift and selecting them through an optimization-based approach to reduce labeling costs while maintaining high accuracy. Finally, OWAD ensures shift adaptation by assigning importance weights to model parameters, preventing catastrophic forgetting while generalizing to the new normality distribution.

### ***7.1.3 Trust Update***

Implementing trust updates involves notifying stakeholders about reputation credit changes, tuning thresholds for the linear equation for the reputation credit calculation and incorporating unsupervised anomaly detection for clustering. The ML training process includes running simulations for labelled data and fine-tuning based on dynamic situations. The initialization stage involves training an unsupervised anomaly detection model during a specified period. A reputation scale from 0 to 1 is established, with defined intervals. Confidence intervals indicate reliability. Trust updates are implemented by evaluating reputations, updating scores, and taking actions based on confidence levels. Continuous monitoring and iteration ensure the system's effectiveness and adaptability.

### ***7.1.4 Zero Trust Reputation Credit Management***

In case of zero trust, we set all reputation credits to zero except for a certificate trusted authority, which in turn enforces continuous verification of the identity of all involved services and users together with continuous authorization verifications. We use the Istio service mesh with a proper configuration to achieve the zero trust requirements.

##### *7.1.4.1 Architecture*

The Istio service mesh architecture [54] for zero trust management can be observed in Figure 31.

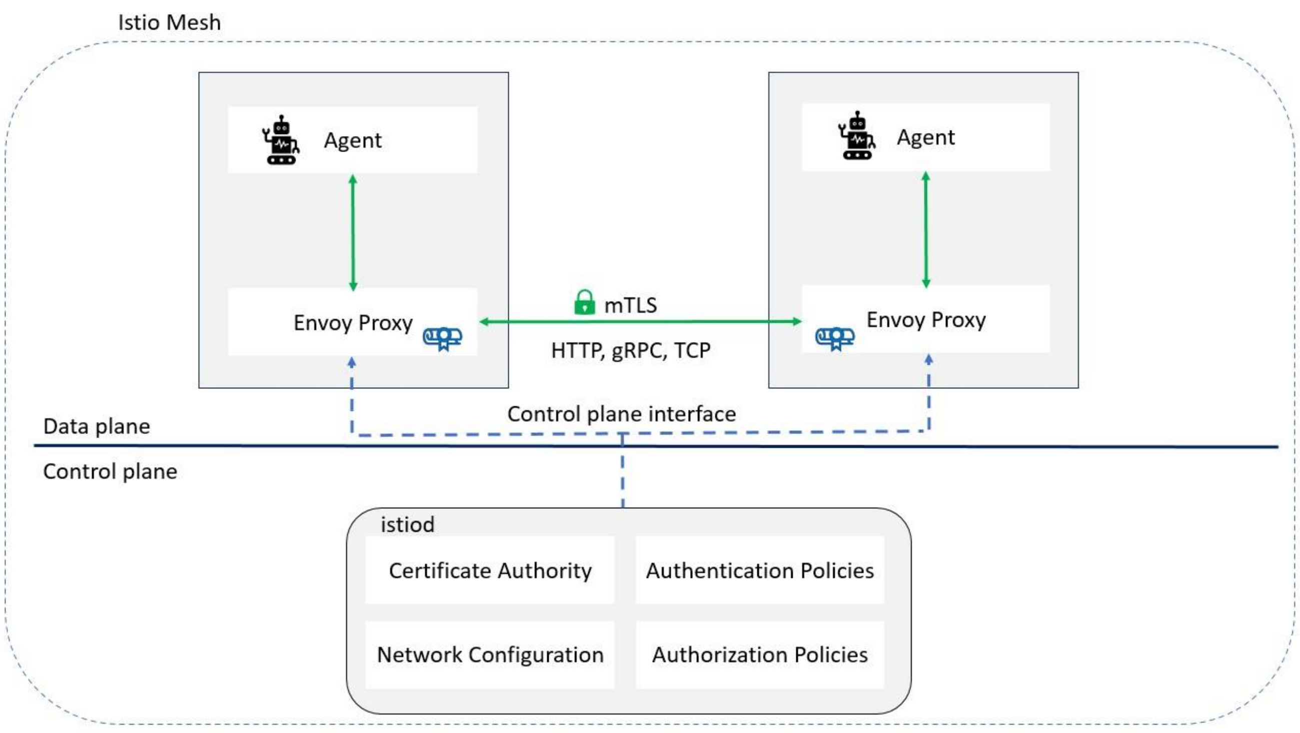


**Figure 31: Istio service mesh architecture for zero trust management.**

Every interacting entity (e.g., MLSysOps agent) is enhanced with an Envoy proxy that takes over the routine of both authentication and authorization on a peer-to-peer basis.

#### *7.1.4.2 Authentication*

The identity authentication is implemented based on a trusted authority which is responsible for certificate signing and management. It represents a self-signed root Certificate Authority (CA) that should be securely managed on the above-continuum level on an offline machine, and it will be used as a root CA for all workloads within a continuum. Subsequently, each cluster within the continuum will receive an intermediate CA issued based on the root CA. Moreover, the potential inter-cluster communication is possible as well in such case. Strong identities are securely provided to every workload using X.509 certificates. Istio agents, operating alongside each Envoy proxy, collaborate with istiod daemon to automate key and certificate rotation.

#### *7.1.4.3 Authorization*

The access control for incoming traffic in the server-side Envoy proxy is regulated by the authorization policy. Each Envoy proxy operates an authorization engine responsible for runtime request authorization. Upon receiving a request, the authorization engine assesses the request context against existing authorization policies, yielding an authorization outcome of either ALLOW or DENY. Operators define Istio authorization policies through .yaml files.

Implicit activation: Istio's authorization features are available immediately upon installation without the need for explicit enabling. To enforce access control on workloads, one can implement an authorization policy.

In the absence of applied authorization policies, Istio permits all requests to proceed unrestricted. Authorization policies support three actions: ALLOW, DENY, and CUSTOM. It has the flexibility to apply multiple policies, each with a distinct action, to enhance the security of your workloads. Istio examines policies in the following order: CUSTOM, DENY, and then ALLOW. For each action type, Istio initially verifies if a policy with the corresponding action exists and then assesses whether the incoming request aligns with the policy's specifications. If no match is found in a particular layer, the evaluation proceeds to the next layer.

Authorization policies are dynamically configured and managed by the MLSysOps framework. These policies are configured using an AuthorizationPolicy custom resource and comprise a selector, action, and a set of rules. The selector designates the policy's target, the action determines whether to allow or deny the request, and rules dictate when to trigger the action. Within the rules, the 'from' field specifies request sources, the 'to' field designates request operations, and the 'when' field outlines conditions for rule application. For instance, the following example showcases an authorization policy allowing access for two sources—the 'cluster.local/ns/default/sa/sleep' service account and the 'dev' namespace—to workloads labeled 'app:httpbin' and 'version:v1' in the 'foo' namespace. This access is granted only when requests contain a valid JWT token.

```
apiVersion: security.istio.io/v1beta1
kind: AuthorizationPolicy
metadata:
  name: httpbin-policy
  namespace: foo
spec:
  selector:
    matchLabels:
      app: httpbin
      version: v1
  action: ALLOW
  rules:
  - from:
    - source:
        principals: ["cluster.local/ns/default/sa/sleep"]
    - source:
        namespaces: ["dev"]
    to:
    - operation:
        methods: ["GET", "POST"]
    when:
    - key: request.auth.claims[iss]
      values: ["https://example.com"]
```

# 8 MLSysOps Agent Architecture

The agent component forms the core of the MLSysOps framework. It provides essential integration logic across all layers, connecting the configuration mechanisms of the underlying system, telemetry data collected from various system entities (e.g., application, infrastructure), and system configuration policies. Figure 32 illustrates the high-level architectural structure of the agent. The component exposes two interfaces—the Northbound and Southbound APIs—which offer structured methods for different system users to interact with it. The Northbound API targets application and policy developers, whereas the Southbound API is primarily intended for system administrators and mechanism providers.

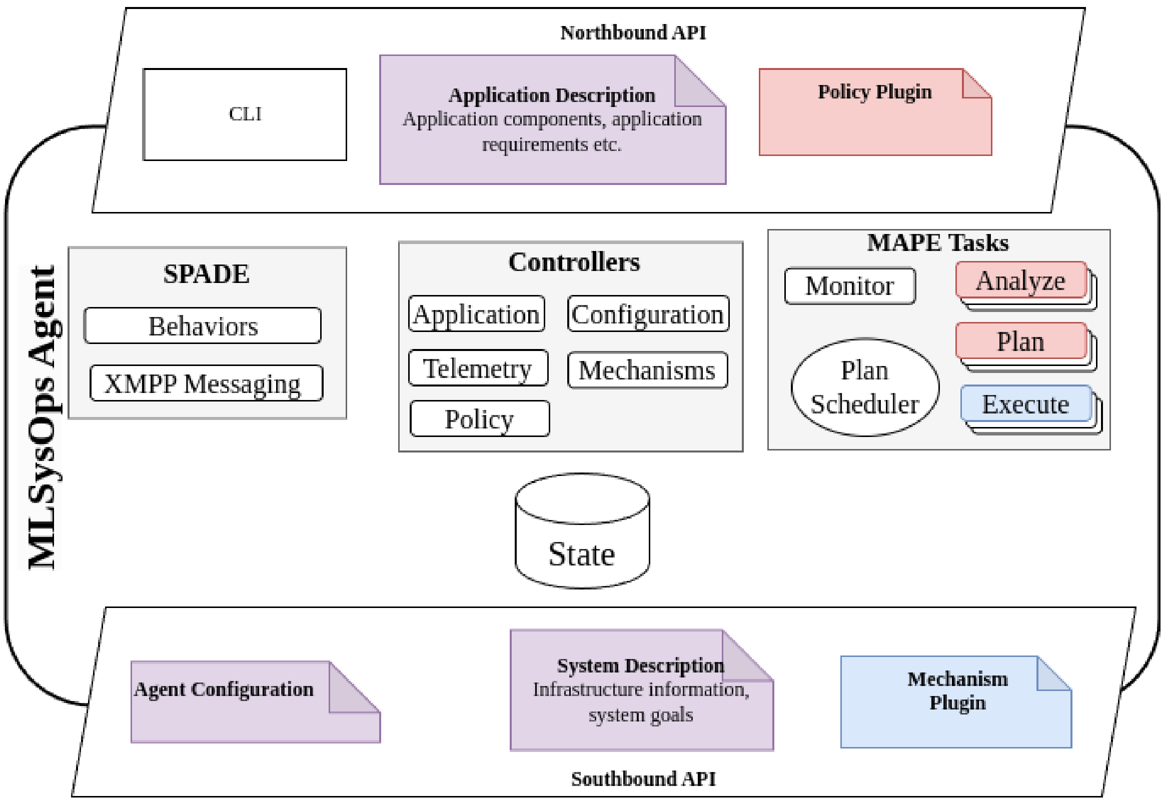


**Figure 32. MLSysOps Agent Architecture.**

The agent follows **MAPE (Monitor-Analyze-Plan-Execute)** paradigm, which was proposed in 2003 [55] to manage autonomic systems given high-level objectives from the system administrators, by using the same notion for the main configuration tasks, depicted as MAPE Tasks in Figure 32. The programming language of choice is Python, and leverages **SPADE Python multi-agent framework** [56] to form a network of agents that can communicate through XMPP protocol and a set of defined messages, providing any necessary functionality from internal tasks that are called behaviours. To achieve seamless operation between the various sub-modules, the agent implements a set of controllers that are responsible for managing the various external and internal interactions.

One important design goal of the agent was extensibility. This goal is achieved by defining simple yet powerful abstractions for two important actors interacting with the system: on one side, the policy developer, who implements the core management logic, and on the other side, the mechanism provider, who exposes the available configuration options for a subsystem. Both abstractions are integrated into the MLSysOps agent as plugin functionalities, specifically named **policy** and **mechanism plugins**. The agent's analysis, planning, and execution tasks depend on this plugin system to generate intelligent configuration decisions—provided by the installed policy plugins—and to apply those decisions to the underlying system via the available mechanism plugins.

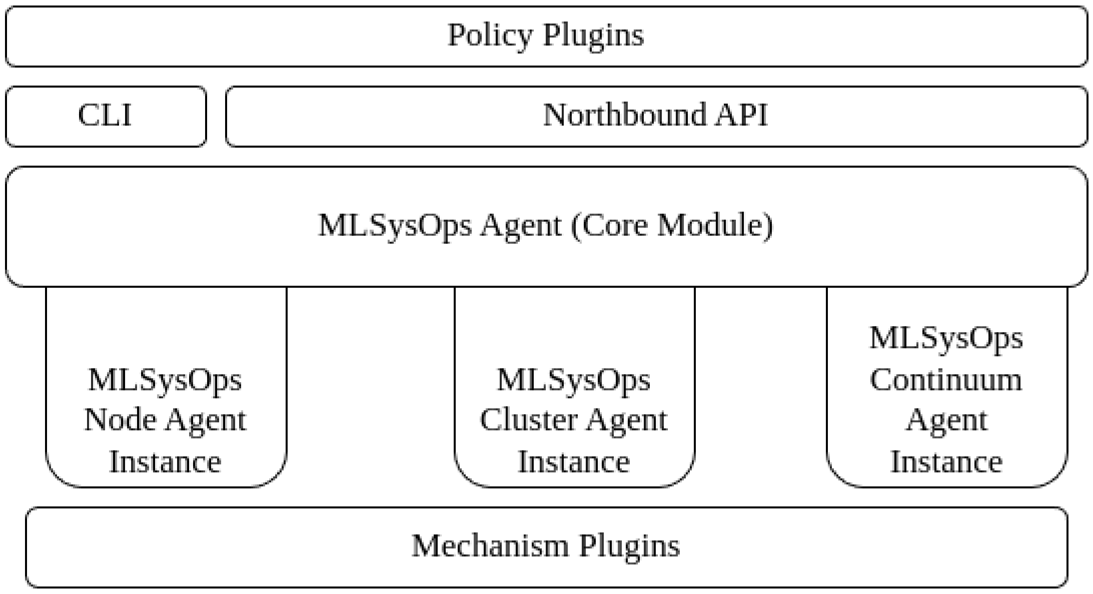

Policy Plugins
CLI
Northbound API
MLSysOps Agent (Core Module)
MLSysOps Node Agent Instance
MLSysOps Cluster Agent Instance
MLSysOps Continuum Agent Instance
Mechanism Plugins

**Figure 33. High-Level View of MLSysOps Agent components.**

The agent software is structured into different module types, as depicted in Figure 33:

- **Core Module** – Provides foundational functionalities shared by all agent instances (continuum, cluster, and node).
- **Layer-Specific Modules** – Offer customized implementations specific to the roles of continuum, cluster, or node agents.
- **External Interface Modules** – Facilitate interactions between the agent framework and external entities. These modules include the CLI, Northbound API, ML Connector, policy and mechanism plugins.

This modular architecture ensures consistency in core functionalities across all agents, while also supporting customization and extension for specific layers and external interactions. The following paragraphs provide further details for the MLSysOps agents.

## 8.1 MAPE Tasks & State

The primary responsibility of the agents is to manage the system's assets—entities and components that can be configured and/or must be monitored. Typical assets include application components, available configuration mechanisms, and the telemetry system. The MAPE tasks continuously monitor the state of these assets, analyze their condition, determine whether a new configuration plan is required, and, if so, create and execute the plan using the mechanism plugins. The Analyze and Plan tasks invoke the logic implemented in the policy plugins, whereas the Execution task uses the mechanism plugins.

### *8.1.1 Monitor*

The Monitor task runs periodically, collecting information from the environment and updating the agent's internal state. This information is sourced from the telemetry system, external mechanisms (via Southbound API mechanism plugins), and other external entities (e.g., other agents). Although there is only a single instance of the Monitor task, it is adaptive; its configuration can change at runtime based on the agent's current requirements. A fundamental configuration parameter is the frequency and type of information retrieved from the telemetry system. For example, when a new application is submitted to the system, additional telemetry metrics may need to be collected and incorporated into the internal state.

### *8.1.2 Analyze*

For each distinct managed asset, a separate Analyze task thread runs periodically. This thread invokes the corresponding method of the active policy plugin (see Section 8.3.1) for the specific asset, supplying all necessary inputs, including telemetry data and relevant system information (e.g., application and system descriptions). Policy plugins may implement the analysis logic using simple heuristics or employ machine learning models, either through the MLSysOps ML Connector or via any external service. This task also includes core logic to perform basic failure checks in the event of errors arising within the policy plugins.

The output of the Analyze task is a binary value (True or False), indicating whether a new configuration plan is required for the analyzed asset. If the result is True, a new Plan task is initiated.

### *8.1.3 Plan*

The Plan task is responsible for generating a new configuration plan and is executed once for each positive result produced by the Analyze task. The planning logic, implemented by the policy plugins, is invoked upon trigger and receives all necessary input data.

The output of this task is a dictionary containing values expected by each mechanism plugin. This dictionary represents the configuration plan to be applied by the respective configuration mechanisms. The result is pushed into a queue and forwarded to the Plan Scheduler (see Section 8.1.5).

### *8.1.4 Execute*

This task is invoked by the Plan Scheduler (see Section 8.1.5) once for each mechanism that must be configured in a given plan. Based on the dictionary provided by the plan, the corresponding mechanism plugin is called and supplied with the relevant configuration data. The new configuration is applied using a best-effort approach, without any retry logic, and the outcome is logged into the state (see Section 8.1.6). In the event of an error, it is expected that the subsequent run of the Analyze task will detect the issue and handle it appropriately.

### *8.1.5 Plan Scheduler*

Each agent supports the concurrent activation of multiple policy and mechanism plugins. As a result, different policies may generate configuration plans for the same mechanism simultaneously. This situation can lead to conflicts during plan execution, where multiple plans attempt to apply different—and potentially conflicting—configuration changes to the same mechanism at the same time. To handle such conflicts, the MLSysOps agent includes a Plan Scheduler module that processes the queued plans produced by Plan tasks (see Section 8.1.3) in a FIFO manner. The first plan in the queue is applied, and any subsequent plan targeting a mechanism already configured by a previous plan is discarded. The Plan Scheduler is designed to be extensible, allowing support for more advanced scheduling policies in the future.

For each scheduled plan, a single Execute task (see Section 8.1.4) is launched to apply the new configuration.

### *8.1.6 State*

This is the internal state (memory) of the agents. Each agent contains different information depending on its environment (continuum level, node type etc.). Some indicative information needed to be kept are information about the application descriptions as well as system and application telemetry. It is able to store historical snapshots of the telemetry data that has been acquired by the monitor task (see Section 8.1.1).

## 8.2 SPADE

SPADE (Smart Python Development Environment) [56] is a middleware platform for multi-agent systems written in Python, leveraging the capabilities of instant messaging to manage communication between agents. It simplifies the development of intelligent agents by combining a robust platform connection mechanism with an internal message dispatcher that efficiently routes messages to various integrated behaviours. Each agent in SPADE is identified by a unique Jabber ID (JID) and connects to an XMPP server using valid credentials. The use of XMPP not only provides persistent connections and reliable message delivery but also enables real-time presence notifications, which are essential for determining the availability of agents in a dynamic environment.

The selection of SPADE as the middleware for our multi-agent system is based on several key factors. Its native Python implementation allows for seamless integration with machine learning libraries and other smart applications, ensuring that sophisticated functionalities can be embedded directly within the agents. SPADE adheres to the FIPA standards, promoting interoperability with other agent platforms such as JADE, which is crucial for systems requiring diverse communication protocols. Furthermore, its modular architecture and open-source nature foster a vibrant community for continuous improvement, supporting extensibility through plugins, and custom behaviours. This robust, flexible design not only accelerates the development cycle but also provides a reliable foundation for building complex, intelligent multi-agent systems.

### *8.2.1 Behaviours*

In SPADE, a behaviour is a modular piece of functionality that encapsulates a specific task or activity for an agent. Behaviours determine how an agent reacts to incoming messages, processes information, or interacts with its environment. They can operate in different modes:

- Cyclic and Periodic behaviours are useful for performing repetitive tasks.
- One-Shot and Time-Out behaviours can be used to perform casual tasks or initialization tasks.
- The Finite State Machine allows more complex behaviours to be built.

This flexible structure allows us to efficiently delegate tasks without overcomplicating the agent’s core logic, ensuring clean and maintainable design.

Within our multi-agent system, each agent is assigned specific behaviours based on its role and the tasks it needs to perform. Each type of agent includes unique behaviours that enable it to carry out specialized tasks. For example, the Continuum Agent is responsible for interacting with both other agents and the user, incorporating behaviours for processing user requests, such as responding to a ping message to confirm aliveness, processing application and ML model descriptions, and checking the deployment status of these applications or models. And other behaviours to process agent’s interactions that are also common functionalities across the agents, for example handling subscriptions and heartbeat messages.

As previously mentioned, agents are structured across three layers: **Node Agents**, **Cluster Agents**, and a central **Continuum Agent**. Each type of agent is assigned a specific set of behaviours that align with its role in the system. These behaviours enable the agents to communicate, monitor health, process application logic, and adapt to runtime conditions. Below is a detailed explanation of each behaviour, followed by a description of which agent types implement them.

- **Heartbeat Behaviour:** This behaviour is used by node and cluster agents and is responsible for periodically sending a signal that indicates the agent is alive. These heartbeat messages are used by higher-layer agents to maintain an up-to-date view of active agents in the system.

- **Subscribe Behaviour:** Used by Node and Cluster Agents, this behaviour sends a subscription request to a higher-layer agent. It allows the agent to join the hierarchical structure and start reporting to its parent, establishing the control flow across layers.
- **Message Receiving Behaviour:** Present in all agent types, this behaviour allows an agent to handle incoming messages from other agents. These messages may contain commands, data updates, or coordination requests. It is essential for asynchronous interaction across the distributed system.
- **Message Sending Behaviour:** Also implemented by all agent types, this behaviour handles sending messages to other agents. It enables agents to initiate communication, send results, or trigger actions elsewhere in the system.
- **Management Mode Behaviour:** This behaviour allows agents to switch between different decision-making strategies. It is present across all agents and can dynamically toggle between heuristic control and machine learning-based approaches. This flexibility allows the system to adjust its intelligence level based on runtime context or user commands.
- **Policy & Mechanism Plugin Management Behaviour:** Implemented by all agents, this behaviour allows enabling or disabling plugins at runtime. It supports dynamic reconfiguration of agent logic and enhances adaptability without requiring redeployment.
- **HB Receiver Behaviour:** This behaviour is used in Cluster and Continuum Agents. It receives heartbeat signals sent by lower-layer agents (e.g., Nodes), updates their status, and maintains a local registry of active agents.
- **Check Inactive Agents Behaviour:** This behaviour complements the heartbeat mechanism. It is used in Cluster and Continuum Agents to scan the list of subscribed agents and identify those that have stopped sending heartbeats, indicating failure or disconnection.
- **Manage Subscription Behaviour:** Implemented by Cluster and Continuum Agents, this behaviour accepts and registers agents from a lower layer. It enables agents to expand their scope of management as new agents come online and request to be part of the system.
- **API Ping Behaviour:** Exclusive to the Continuum Agent, this behaviour allows external components such as the command-line interface to verify whether the agent is alive by sending ping messages through the North Bound API.
- **Check App Deployment Behaviour:** This behaviour is also unique to the Continuum Agent. It verifies that the components of an application have been properly deployed in the framework. It ensures application consistency across the infrastructure.
- **Check ML Deployment Behaviour:** Like the previous behaviour but focused on machine learning applications. It verifies that ML services are correctly deployed and ready for operation. This behaviour also exists only in the Continuum Agent.
- **App Process Behaviour:** Implemented in the Continuum Agent, this behaviour analyzes the application description and determines how and where to deploy its components. It interprets application specifications and translates them into deployment strategies.
- **ML Process Behaviour:** This behaviour, also integrated into the Continuum Agent, is responsible for managing the full lifecycle of machine learning endpoint deployments. It handles the deployment of new ML models or services, monitors the status of deployed models in real-time, supports redeployment in case of model updates or infrastructure changes, and manages the deletion of endpoints when they are no longer needed. This ensures a consistent and automated approach to maintaining ML services across the continuum infrastructure.

Each of these behaviours is designed to work in harmony within its respective agent, ensuring that the system remains modular, scalable, and responsive to dynamic environments. As we continue to develop our framework, we can further refine each behaviour to meet the specific requirements of our agents and enhance the overall efficiency of the multi-agent system.

Table 15 summarizes the behaviours types.

**Table 15: Behaviours per Agent.**

| Behaviour Name | Type | Type of Agent | | |
|---|---|---|---|---|
| | | Continuum | Cluster | Node |
| API Ping | Cyclic | x | | |
| Check inactive | Periodic | x | x | |
| Check _app_deployment | One Shot | x | | |
| Check_ml_deployment | One Shot | x | | |
| HB Receiver | Cyclic | x | x | |
| HeartBeat | Periodic | | x | x |
| ML_process | Cyclic | x | | |
| Manage Subscription | Cyclic | x | x | |
| Management mode | Cyclic | x | x | x |
| Message receiving | Cyclic | x | x | x |
| Message sending | One Shot | x | x | x |
| Process | Cyclic | x | | |
| Subscribe | Cyclic | | x | x |
| Policy management | Cyclic | x | x | x |

### *8.2.2 Messages*

SPADE agents communicate by exchanging discrete messages rather than direct method calls, embodying the “computing as interaction” paradigm of multi-agent systems. As previously mentioned, each agent is identified by a unique ID (JID) (username@domain) and connects to an XMPP server using this ID and a password. SPADE relies on the XMPP (Extensible Messaging and Presence Protocol) as the backbone for all inter-agent communication. This means that every message an agent sends is transmitted as an XMPP message stanza through the server to the target agent. By using XMPP’s standard messaging infrastructure, SPADE ensures that agents can reliably send and receive messages in real time, even across different hosts or network environments. In essence, the XMPP server mediates the exchange, routing each message to the intended recipient agent (identified by its JID) whether the agents reside on the same machine or are distributed over the Internet. This decoupled, server-mediated communication model provides a robust and standardized way for agents to interact, leveraging XMPP’s features for authentication, presence, and security (e.g. encryption) built into the protocol.

For this XMPP server, the open-source ejabberd service was selected due to its superior scalability, reliability, performance, security, ease of integration with SPADE, and strong community support. The configuration of the service is made on the config file providing a domain and each agent can register into it by using a jabber id and a password. Once the agent is registered it is ready to start exchanging messages with other registered agents.

**Message Dispatching and Templates**

Within each SPADE agent, an internal message dispatcher handles incoming and outgoing messages. This dispatcher functions like a mail sorter: when a message arrives for the agent, the dispatcher automatically places it into the proper “mailbox” for handling, and when the agent sends a message, the dispatcher injects it into the XMPP communication stream. The key to this routing is the use of **message templates**. Each behaviour (task) running within an agent can be associated with a message template that defines the criteria for messages it is interested in. A template can specify fields such as the sender’s JID, the message’s content or subject, thread/conversation ID, or metadata like performative and ontology. When an agent receives a message, the dispatcher compares the message against the templates of all active behaviours and delivers the message to the behaviour whose template it matches. In this way, templates act as filters to ensure each behaviour only processes relevant messages. For example, a template might match messages with a particular sender and a specific performative type, so that only messages from that sender with that communicative intent will trigger the associated behaviour. Messages that meet the template conditions are queued in the target behaviour’s mailbox, where the behaviour can retrieve them asynchronously. This template-based filtering and routing mechanism allows multiple behaviours to run concurrently in an agent without interfering with each other, as each behaviour will only pick up the messages meant for it. It provides a structured approach to message handling, simplifying the development of complex interactions (such as protocol exchanges) by separating them into different behaviour handlers listening for different message patterns.

**FIPA Standards for Structured Communication**

SPADE’s messaging model also draws from established standards to ensure that communications are well-structured and interoperable. In particular, SPADE supports message formats inspired by the FIPA (Foundation for Intelligent Physical Agents) Agent Communication Language standards. FIPA defines a set of message fields and interaction protocols intended to promote clear semantics and compatibility among agents. In SPADE, each message’s metadata can include standard FIPA-ACL fields like the *performative* (which describes the intent of the message, such as “inform” or “request”), the *ontology* (which defines the domain of discourse or vocabulary of the message content), and the *language* (the format of the message content). By allowing these fields in the message structure, SPADE ensures that every message carries not just raw data but also contextual information about how to interpret that data. Adhering to FIPA communication standards means that SPADE agents follow a common protocol syntax and semantics, which in principle makes it easier for them to interact with agents from other FIPA-compliant platforms. In other words, the use of well-defined performatives and message fields imposes a consistent structure on messages, reducing ambiguity and enhancing interoperability. This standards-based approach to message handling helps achieve a level of consistency in agent communication, so that the intent and context of messages are understood in a uniform way across different agents and systems. Ultimately, SPADE’s alignment with FIPA standards reinforces structured agent interactions and lays the groundwork for integration with the broader multi-agent ecosystem where such standards are followed.

### *8.2.3 Interactions – Coordination*

Agents rely on behavioural logic and message exchanges to interact and coordinate tasks across the layers of the continuum. Below is an example illustrating how subscription and heartbeat mechanisms are implemented using agent behaviours to facilitate this interaction.

**Subscription and Heartbeat Messages**

Subscription and heartbeat messages are essential processes used to register available nodes within the continuum and to maintain up-to-date information about the status of nodes and agents across the entire system.

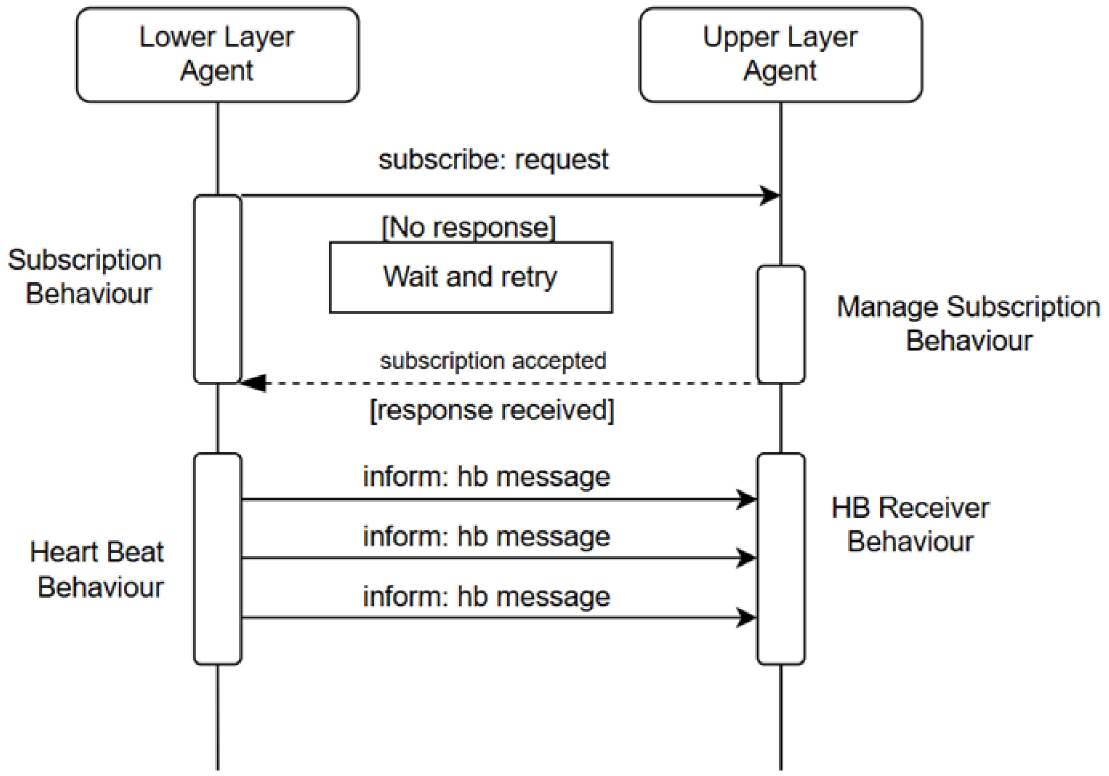


**Figure 34. Subscription and HB process.**

In the subscription process, a lower-layer agent sends a subscription request using the `subscribe` performative to an upper-layer agent. Since the behaviour is cyclic, the lower-layer agent continues to send subscription requests until it receives a `subscription accepted` message from the upper-layer agent. Once the acknowledgment is received, the agent stops the cyclic subscription behaviour and initiates the periodic heartbeat behaviour.

During the heartbeat phase, the lower-layer agent periodically sends heartbeat (HB) messages using the `inform` performative. The upper-layer agent, which runs a heartbeat receiver behaviour, constantly listens for these messages and updates its records based on the latest HB information.

As shown in Figure 32, the interaction between agents through message exchanges and behaviour logic enables the coordinated execution of various tasks across the continuum. In this example, the focus is on the subscription and heartbeat process. These coordinated mechanisms allow agents to collaborate and support broader functionalities within the framework, which are further detailed in the open-source repository [2].

## 8.3 Framework Plugins

The MLSysOps framework provides a structured plugin mechanism that enables a modular approach to integrating configuration decision logic with arbitrary mechanisms. Plugins are categorized into two types: core and custom. Core plugins are always enabled and are used by MLSysOps to perform essential management functions. In contrast, custom plugins can be configured and activated either during installation or at runtime, allowing flexibility and extensibility based on system requirements.

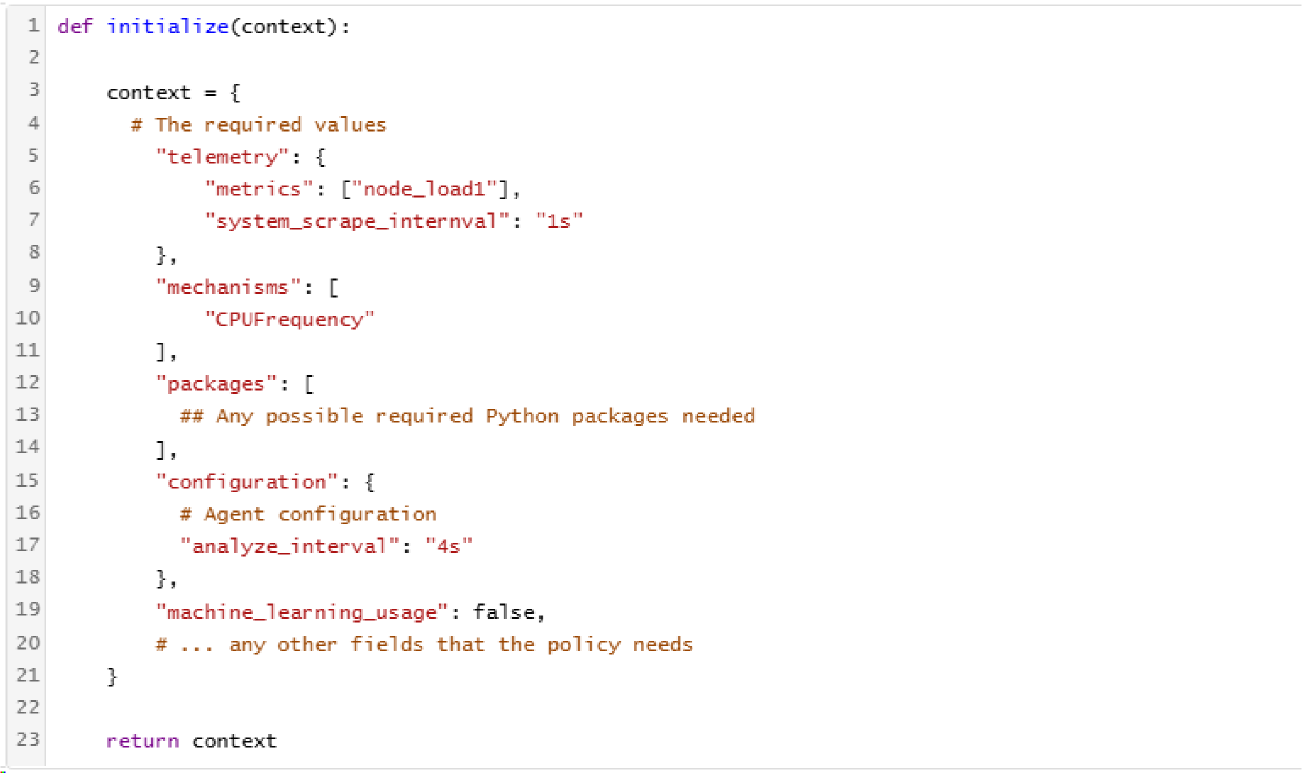
All AI-ready mechanisms described in this document can be potentially related with the MLSysOps agent as plugins. Section 8.3.3 includes a table listing the main core plugins that are integrated into the framework.

### *8.3.1 Policy Plugins*

Policy plugins are the components responsible for determining if a new adaptation is required and generating new configuration plans. They follow the MAPE paradigm, specifically implementing the Analyze and Plan tasks. A policy plugin is implemented as a Python module, which may import and use any external libraries, and must define three specific methods: (i) initialize, (ii) analyze, and (iii) plan. Each method requires specific arguments and must return defined outputs. Each method accepts a common argument, context, which can be used to maintain state between different method calls, as well as across consecutive invocations of the policy plugin. Policy plugins can be bound to a node or multiple applications; however, they need to decide on at least one mechanism.

The methods are described as follows:

**initialize**: This method contains the initialization configuration required by the agent. It is called during the plugin loading phase, and it must return the common context dictionary with specific values. An example is shown in Figure 35, where the policy declares the telemetry configuration it requires, the mechanisms it will analyze and manage, any custom Python packages needed by the script, and any additional agent configuration parameters. An important parameter is to declare if this policy will make use of machine learning - this enables the usage of the ML Connector interface and accordingly configures the mechanism that enables/disables machine learning usage in the framework.



```python
def initialize(context):

    context = {
      # The required values
        "telemetry": {
            "metrics": ["node_load1"],
            "system_scrape_internval": "1s"
        },
        "mechanisms": [
            "CPUFrequency"
        ],
        "packages": [
          ## Any possible required Python packages needed
        ],
        "configuration": {
          # Agent configuration
          "analyze_interval": "4s"
        },
        "machine_learning_usage": false,
        # ... any other fields that the policy needs
    }

    return context
```

**Figure 35. Example of node-level policy plugin initialize method.**

**analyze**: The purpose of this method is to analyze the current state of the system and running applications, and determine whether a new configuration plan might be required. In the example shown in Figure 36, the analyze function compares the current telemetry data for the application—retrieved using the application description—with the target value specified by the application. If the current value exceeds the defined threshold (target), the method concludes that a new plan is needed. In this example, it is assumed that the monitored application metric should remain below the specified target. The analyze method can also make use of the ML Connector interface, to make use of machine learning models deployed from that service.

```python
def analyze(context, application_description, system_description, current_plan, telemetry, ml_connector):

    # policy that checks if application target is achieved

    if telemetry['data']['application_metric'] > application_description['targets']['application_metric']:
      return True, context # It did not achieve the target - a new plan is needed

    return False, context
```

**Figure 36. Example of a node-level policy plugin analyze method.**

**plan**: This method decides if a new plan is needed, and if it is positive, generates a new configuration plan based on all available information in the system, including application and system descriptions, telemetry data, the current plan, and available assets. It may also leverage the ML Connector interface to invoke machine learning models. In the example shown in Figure 37, the plan method creates a new configuration for the CPU frequency of the node on which it runs. If the application target is not met, the method sets the CPU to the maximum available frequency; otherwise, it sets it to the minimum. The configuration values used in the plan are predefined and known to the policy developer, based on the specifications of the corresponding mechanism plugin (see Section 8.3.2 for examples).

```python
def plan(context, application_description, system_description, current_plan, telemetry, ml_connector
         , available_assets):

    if telemetry['data']['application_metric'] > application_description['targets']['application_metric']
       cpu_frequency_command = {
            "command": "set",
            "cpu": "all",
            "frequency": "max"
        }
    else:
       cpu_frequency_command = {
            "command": "set",
            "cpu": "all",
            "frequency": "min"
        }

    if new_plan != current_plan:
      new_plan = {
          "CPUFrequency": cpu_frequency_command
      }
      return new_plan, context


    return current_plan, context
```

**Figure 37. Example of a node-level policy plugin plan method.**

For both the *analyze* and *plan* methods, the arguments are as follows [2]:

- **application_description**: A dictionary or a list of dictionaries containing values from the submitted applications in the system (see Section 8.5.1).
- **system_description**: A dictionary containing system information provided by the system administrator (see Section 8.5.2).
- **current_plan**: The currently active plan for this policy. Since a previously generated plan may have failed, this argument allows the policy plugin to handle such scenarios appropriately.
- **telemetry**: A dictionary containing telemetry data from both the system and the applications.
- **ml_connector**: An object handler providing access to the ML Connector service endpoint within the slice. This argument is empty if the ML Connector service is not available [see documentation].

As described in Section 8.1, the above plugin methods are invoked and executed within the respective Analyze and Plan tasks. The Plan Scheduler ensures that any conflicts between different policy-generated plans are resolved and forwards them to the Execute tasks, which utilize the mechanism plugins to apply the configuration to the system. The declaration of machine learning model usage for each plugin enables MLSysOps to track where and when machine learning mechanisms are employed, monitor their performance, and disable plugins that utilize AI tools if requested. The plug-and-play support further allows for the dynamic modification of configuration logic, enabling agents to adapt to varying operational scenarios.

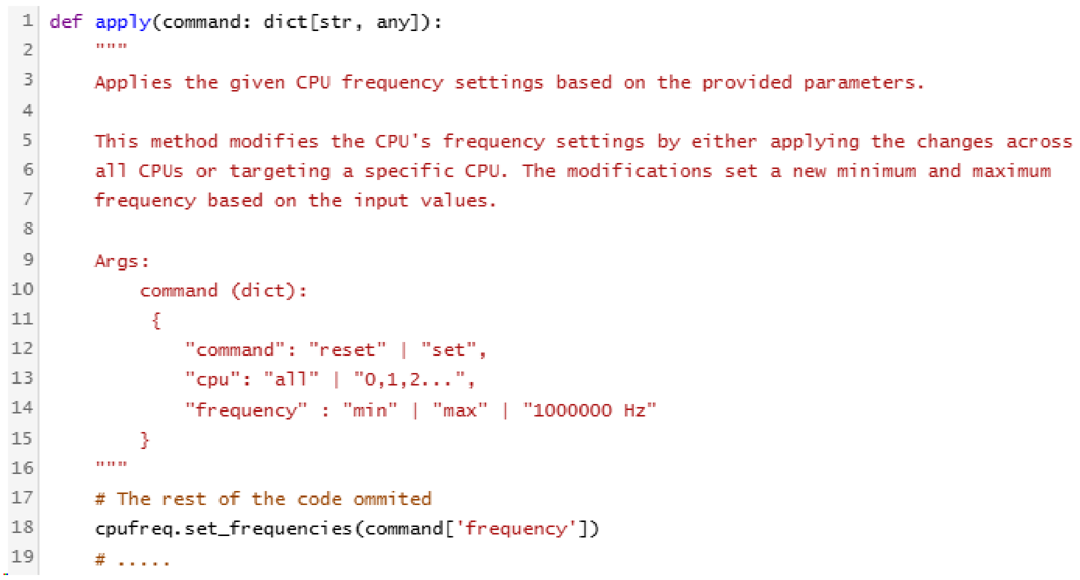
### *8.3.2 Mechanism Plugins*

The MLSysOps framework does not impose assumptions about the underlying system architecture, recognizing that real-world infrastructures often consist of heterogeneous systems with varying adaptation capabilities and operational requirements. Different types of nodes offer different configuration options, and nodes operating at higher levels of the continuum (e.g., cluster or continuum nodes) have distinct configuration needs. To ensure seamless integration—especially with the policy plugins—MLSysOps defines a standardized plugin interface for system administrators and mechanism providers, called **mechanism plugins**.

A mechanism plugin is provided as a Python script that implements three methods: (i) *apply*, (ii) *get_status*, and (iii) *get_options*. The plugin may use any required libraries, and it is assumed that any necessary packages are pre-installed along with the agent.

The methods are defined as follows:

**apply**: This is the primary method invoked by an Execute task (see Section 8.1.4). It accepts a single argument, command, which is a dictionary whose structure is defined and documented by the mechanism plugin. This dictionary is produced by the plan method of a policy plugin. The policy developer must be familiar with the available mechanism plugins in the system and the expected format of the command argument. Figure 38 shows an example of a CPU configuration plugin that utilizes supporting libraries, as described in Section 3.2. The expected dictionary structure is documented in the method's comment section, followed by the call to the underlying library to apply the specified configuration.



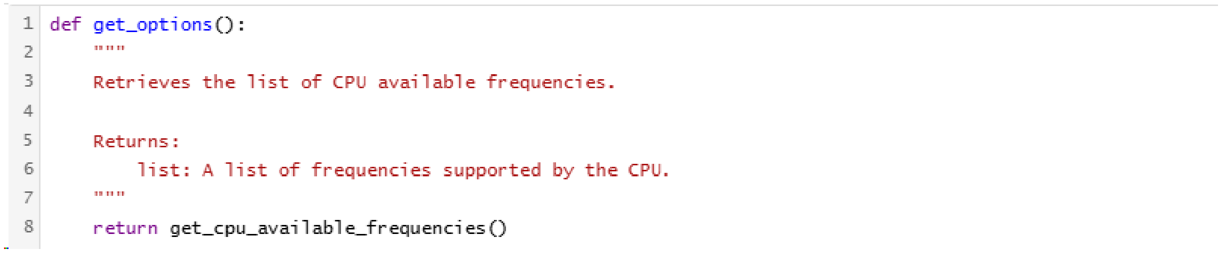
```python
def apply(command: dict[str, any]):
    """
    Applies the given CPU frequency settings based on the provided parameters.

    This method modifies the CPU's frequency settings by either applying the changes across
    all CPUs or targeting a specific CPU. The modifications set a new minimum and maximum
    frequency based on the input values.

    Args:
        command (dict):
         {
            "command": "reset" | "set",
            "cpu": "all" | "0,1,2...",
            "frequency" : "min" | "max" | "1000000 Hz"
        }
    """
    # The rest of the code ommited
    cpufreq.set_frequencies(command['frequency'])
    # .....
```

**Figure 38. CPU Frequency mechanism plugin apply method.**

**get_options**: It returns the available configuration options for the mechanism that is handled by the plugin. In the example in Figure 39, it returns the available CPU frequency steps, that can be used as values in the frequency key of the command dictionary of the apply method. This is meant to be used in a development environment, where MLSysOps framework provides suitable logging methods.



```python
def get_options():
    """
    Retrieves the list of CPU available frequencies.

    Returns:
        list: A list of frequencies supported by the CPU.
    """
    return get_cpu_available_frequencies()
```

**Figure 39. CPU Frequency mechanism get options method.**

The relationship and interaction between the policy and mechanism plugins are demonstrated in Section 8.3.4.

### *8.3.3 Core & Custom Plugins*

Core plugins provide essential policies and mechanisms that are shipped with the MLSysOps Framework by default. On the other hand, custom plugins are plugins made for specific use cases and can be built by third-parties.

The following tables briefly describe the initial plugins that have been developed, up to the time of writing of this document. Latest updates can be found on the open source repo of the project [2].

**Table 16. MLSysOps Policy Plugins.**

| Policy Name | Functionality | Type | Continuum Layer | Development Status |
|---|---|---|---|---|
| Static Node Components Placement | It places specific components on a node. | Core | Cluster | Completed |

| Policy Name | Functionality | Type | Continuum Layer | Development Status |
|---|---|---|---|---|
| Node Component Placement | Places application components on a node, based on application description criteria. | Core | Cluster | Completed |
| Cluster Component Placement | Places application components in a cluster, based on application description criteria. | Core | Continuum | Completed |
| Smart Agriculture Drone Management | Decides on the usage of a drone in a field. | Custom | Cluster | Completed |
| Smart Agriculture Application Prediction | It predicts the application performance of smart agriculture. Does not produce any plan. | Custom | Node | Under Development |
| Smart City Noise Prediction | Predicts the existence of people, using sound sensors. Does not produce any plan. | Custom | Node | Under Development |
| Smart City Cluster Management | Based on the node-level prediction metrics, it configures the application deployment. | Custom | Cluster | Under Development |
| CC Storage Gateway Placement | Decides on the CC storage gateway container. | Custom | Cluster | Under Development |

**Table 17. MLSysOps Mechanism Plugins.**

| Mechanism Plugin Name | Functionality | Type | Continuum Layer | Related Section |
|---|---|---|---|---|
| Fluidity | Manages application component placement in a Kubernetes cluster | Core | Cluster | Section 6.1.1 |
| ContinnumComponentPlacement | Places the components to specific clusters. | Core | Continuum | Section 6.1.3 |
| CPUFrequency | Configures the CPU frequency for the supported architectures. | Core | Node | Sections 3.2, 3.3 |
| NVIDAGPUFrequency | Configures the GPU Frequency for the supported architectures. | Core | Node | Section 3.4 |
| FPGAConfigurator | Configures the active bitrstream of Xilinx MPSoC FPGA. | Core | Node | Section 3.5 |
| vAccelPlugin | Configures the vAccel plugin used by a component. | Core | Node | Section 3.6 |

| Mechanism Plugin Name | Functionality | Type | Continuum Layer | Related Section |
|---|---|---|---|---|
| CCStorage | Configures the storage policy of a storage gateway. | Custom | Cluster | Section 4.1 |
| NetworkRedirection | Configures the network interfaces that are used by application components. | Custom | Cluster | Section 6.1.2 |
| ChangePodSpec | Configures the pod specifications for specific values. | Core | Node, Cluster | Sections 6.2 & 7 |

### *8.3.4 Plugins & MAPE Execution Flow*

Figure 40 illustrates the execution flow of the MAPE tasks and the integration of both policy and mechanism plugins. The Monitor task runs periodically, regardless of whether an application has been submitted, collecting telemetry data and updating the local state. When a new application is submitted to the system, a separate Analyze task thread is launched, which uses the analyze method of the corresponding policy plugin. Based on the result, the analysis session either terminates or triggers a background Plan task.

The Plan task then invokes the policy plugin's plan method to generate a new configuration plan. If a valid plan is produced, it is pushed into a FIFO queue. The Plan Scheduler periodically processes the plans in the queue and initiates an Execute task for each mechanism included in the output of the policy plugin's plan method. The Plan Scheduler enforces a constraint that prevents two different plans from applying configuration changes to the same mechanism within the same scheduling cycle.

Finally, the Execute task for each mechanism calls the apply method of the corresponding mechanism plugin. The results of the configuration are made visible to the next execution of the Analyze task, either via direct status retrieval or through the telemetry stream.

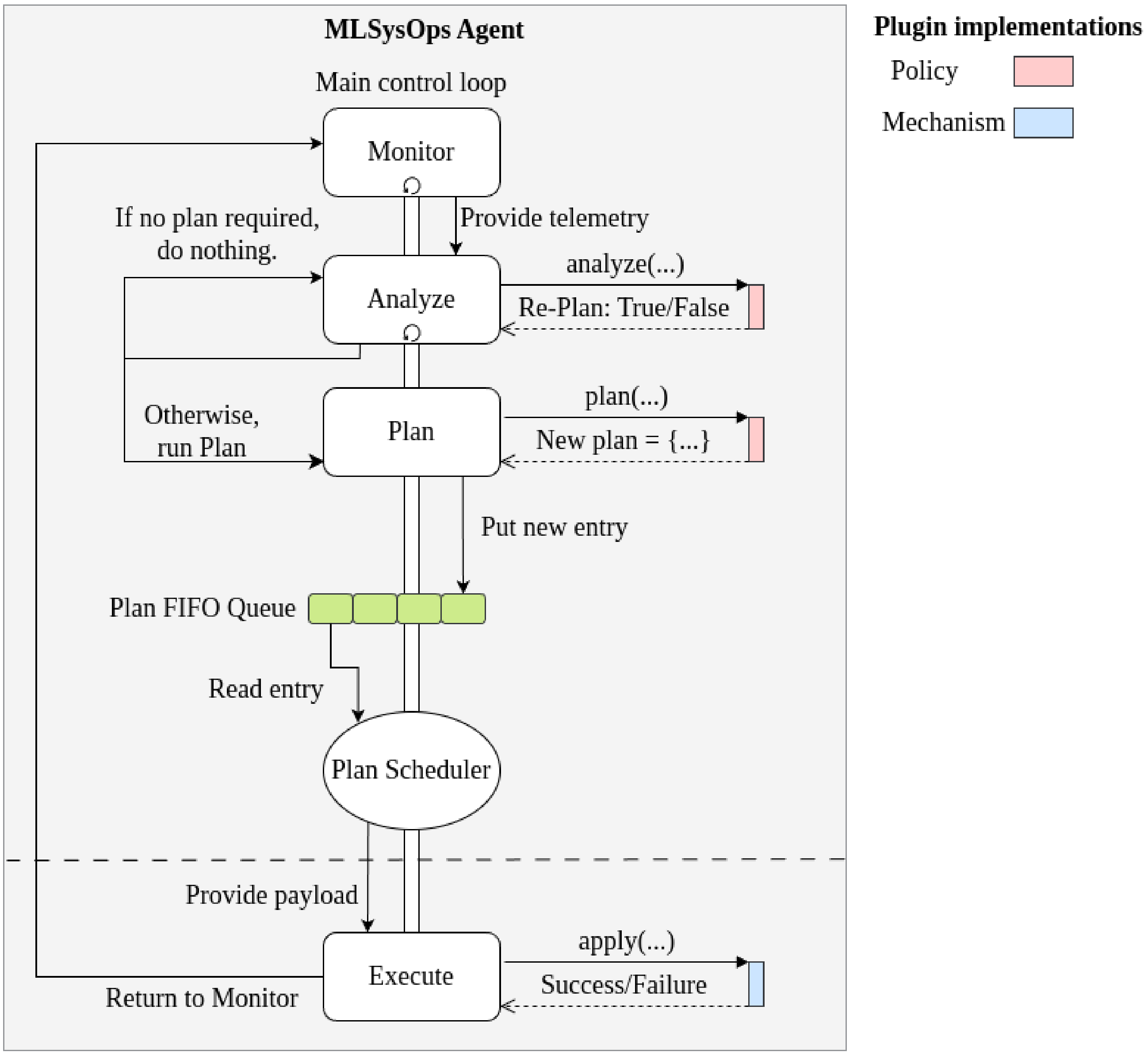


**Figure 40. Interaction between the MAPE tasks and the Policy & Mechanism plugins.**

## 8.4 Controllers

Controllers are responsible for coordinating all internal components of the framework, including the MAPE tasks, SPADE, Policy and Mechanism Plugins, and the Northbound and Southbound APIs.

**Application Controller**: Manages the lifecycle of the Analyze loop for each application submitted to the system. When a new application is submitted, a corresponding Analyze behaviour is initiated, and it is terminated when the application is removed.

**Policy & Mechanism Plugin Controllers**: Responsible for loading, initializing, and configuring policy and mechanism plugins. During runtime, these controllers provide updated information to the Application Controller, reflecting any changes in the policy API files.

**Agent Configuration Controller**: Handles external configuration commands received from other agents or via the Northbound API, and propagates them to the appropriate internal components. It is also responsible for loading the initial configuration file during startup.

**Telemetry Controller**: Manages the OpenTelemetry Collector for each agent, including initial deployment and runtime configuration. Since each collector operates as a pod within the cluster, the Node Agent coordinates with the Cluster Agent to request deployment and updates, as depicted in Figure 41. Additionally, this controller configures the Monitor task based on the telemetry metrics being collected.

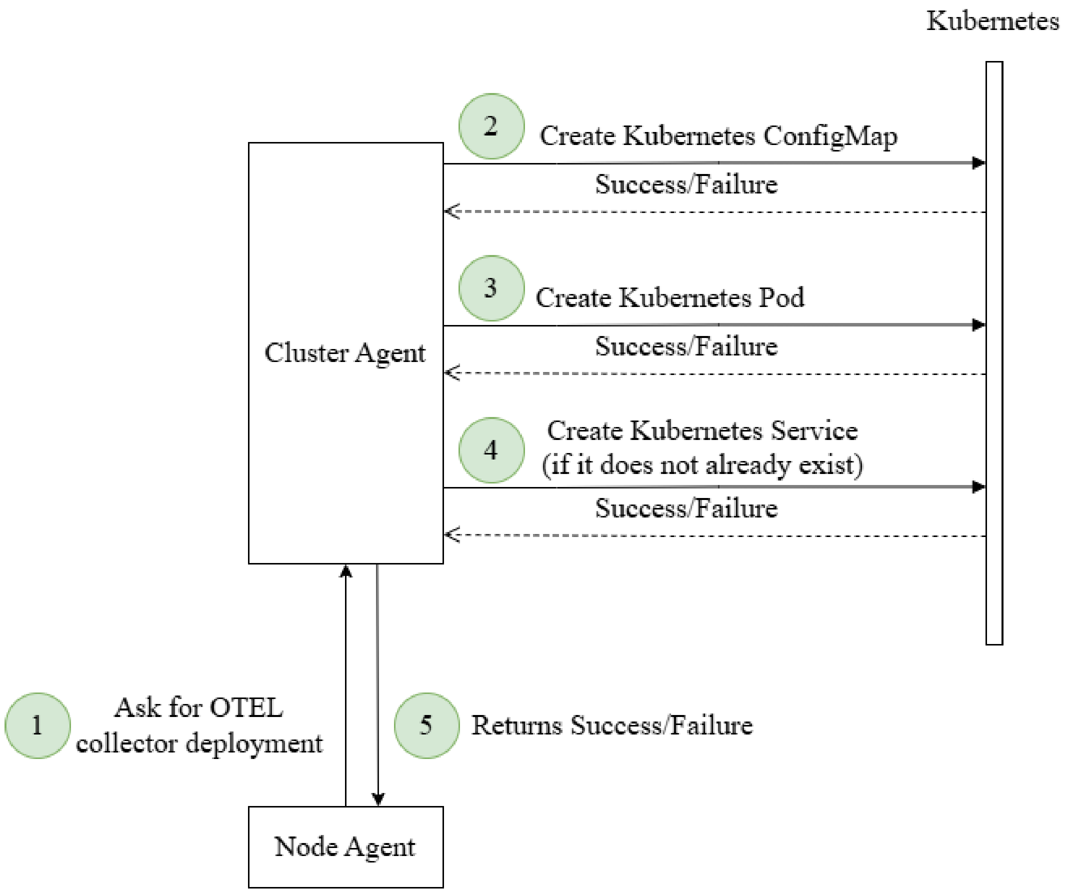


**Figure 41. OpenTelemetry Collectors configuration.**

## 8.5 Actor Provided Files

This section describes how the main actors of the framework can provide the respective files related to applications, agent configurations, and system descriptions. The formal specification, along with the respective documentation for the application, infrastructure, and agent configuration files, can be found as part of the open-source effort of the framework [2].

### 8.5.1 *Application Descriptions*

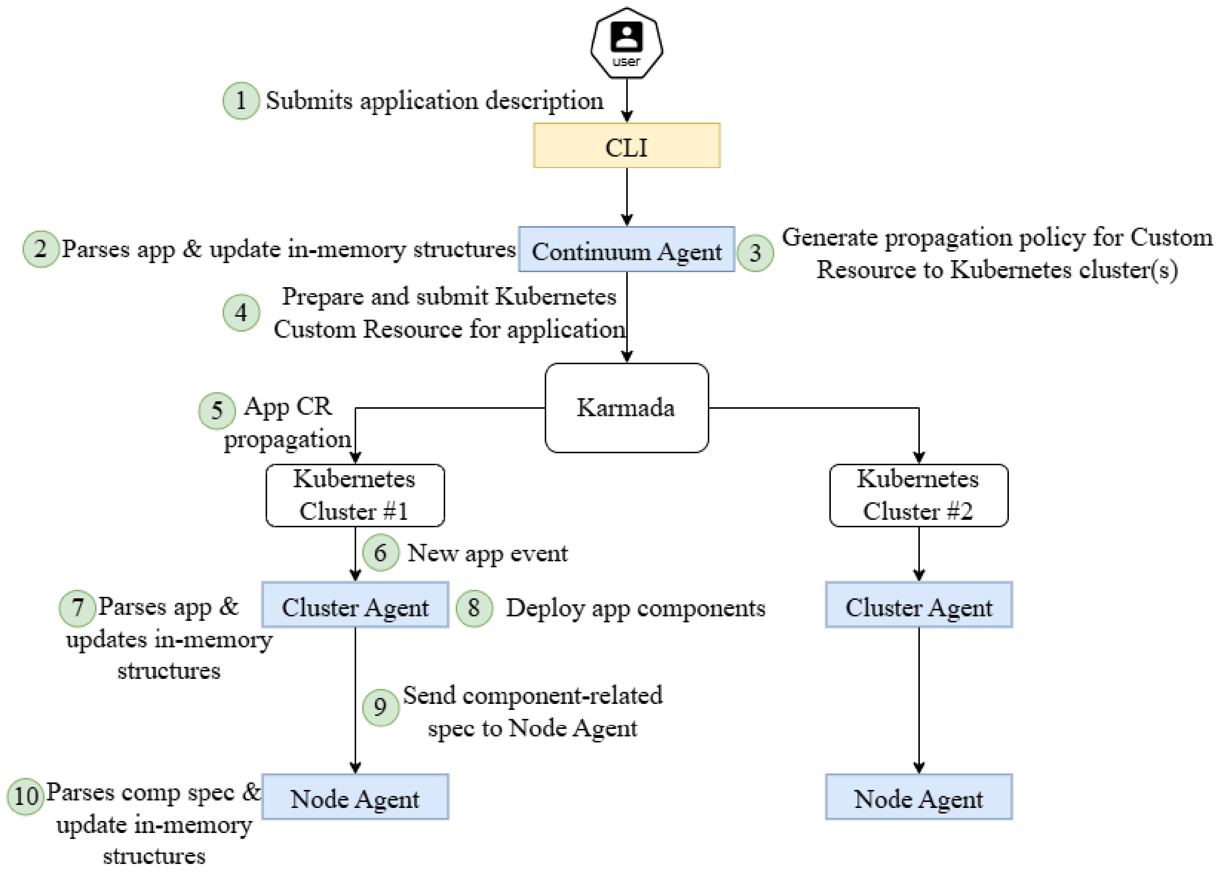

user
1 Submits application description
CLI
2 Parses app & update in-memory structures
Continuum Agent
3 Generate propagation policy for Custom Resource to Kubernetes cluster(s)
4 Prepare and submit Kubernetes Custom Resource for application
Karmada
5 App CR propagation
Kubernetes Cluster #1
Kubernetes Cluster #2
6 New app event
7 Parses app & updates in-memory structures
Cluster Agent
8 Deploy app components
Cluster Agent
9 Send component-related spec to Node Agent
10 Parses comp spec & update in-memory structures
Node Agent
Node Agent

**Figure 42. Application registration end-to-end flow**

The application owner, one of the main actors, interacts with MLSysOps by submitting the application description using the Command Line Interface (CLI) provided by the framework. The application description depicts the required deployment constraints (e.g., node-type, hardware, sensor requirements, etc.), which enable various filtering options for the continuum and cluster layers, that can decide the candidate clusters and nodes, respectively. Having as an example the registration of a given application, as shown in Figure 42, we perform a Top-Down propagation of the necessary information to each layer of the continuum. Initially, the Continuum agent creates a Kubernetes Custom Resource that is propagated to the available Kubernetes clusters. The Cluster agents follow the Kubernetes Operator pattern, so they are notified of application creation, update, or removal events. Each Cluster agent manages the components that match its cluster ID, if any. This information is provided by the Continuum agent in the application's Custom Resource. A given Cluster agent captures the application creation event, parses the description, and deploys the components based on the provided requirements. The component specifications are also sent to their host nodes, so that the Node agents can store relevant fields required for any potential reconfiguration/adaptation.

### *8.5.2 System Descriptions*

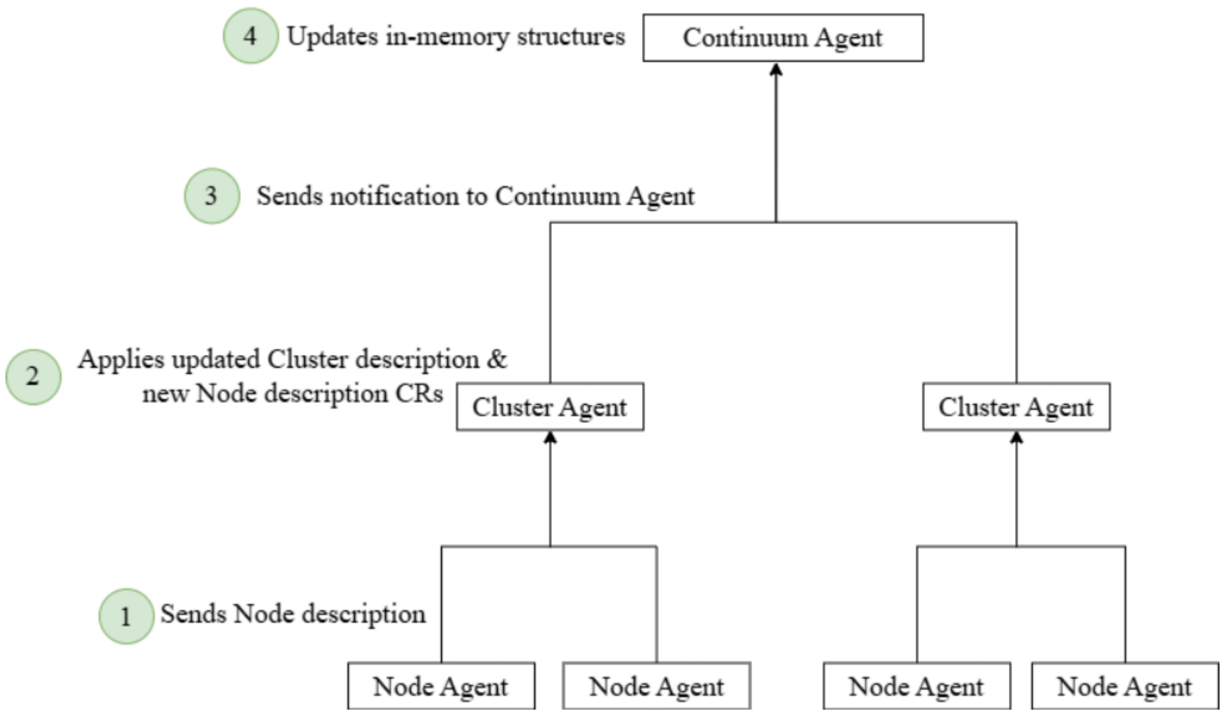

4 Updates in-memory structures
Continuum Agent
3 Sends notification to Continuum Agent
2 Applies updated Cluster description & new Node description CRs
Cluster Agent
Cluster Agent
1 Sends Node description
Node Agent
Node Agent
Node Agent
Node Agent

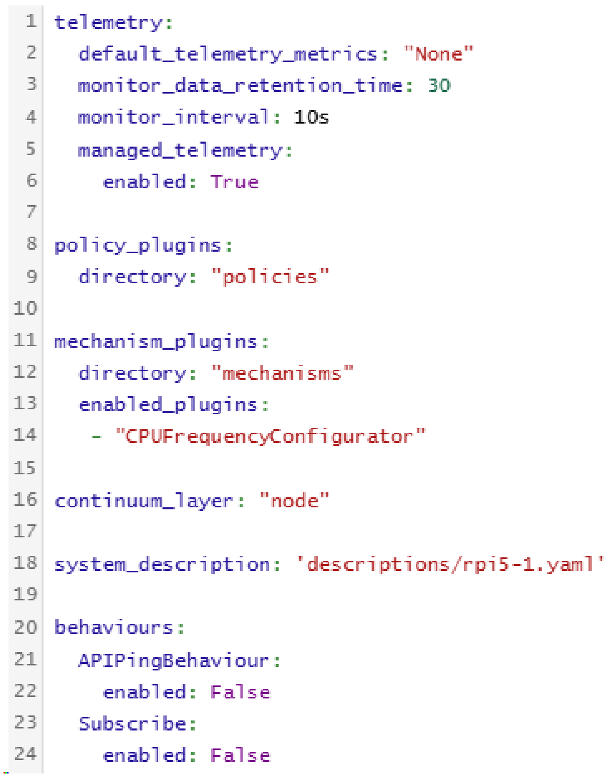
**Figure 43. Node description registration and propagation flow**

The infrastructure descriptions must be provided during the agent installation process. Taking the example of a Node description registration, it is propagated to the framework using a Bottom-Up approach, in contrast to the application registration solution (Top-Down), as shown in Figure 43. To this end, we follow the usual node registration protocols, e.g., node registration to a Kubernetes cluster. In our case, the Node agent sends the respective description to the Cluster agent, which transforms it into a Custom Resource and applies it to Kubernetes. In addition, the Cluster agent updates the Cluster formal description (also defined as a Custom Resource) with high-level information that can be used by the Continuum agent in order to perform filtering based on the available sensors, accelerators, and node types (e.g., to meet any relevant application deployment constraints). Finally, the Cluster agent notifies the Continuum agent, via the agent protocol, so that the latter can update its Cluster-related structures.

### *8.5.3 Agent Configuration*

Each agent uses a configuration file that defines its behaviour during instantiation. While agents operating at different layers of the continuum instantiate different components of the core MLSysOps framework, all agents running on nodes use the same base instance. However, since node characteristics may vary significantly, each agent can be individually configured using its corresponding configuration file.

An example configuration file[7] is shown in Figure 44, highlighting various configuration options. More details can be found in the documentation of the open-source project [2].



```yaml
telemetry:
  default_telemetry_metrics: "None"
  monitor_data_retention_time: 30
  monitor_interval: 10s
  managed_telemetry:
    enabled: True

policy_plugins:
  directory: "policies"

mechanism_plugins:
  directory: "mechanisms"
  enabled_plugins:
   - "CPUFrequencyConfigurator"

continuum_layer: "node"

system_description: 'descriptions/rpi5-1.yaml'

behaviours:
  APIPingBehaviour:
    enabled: False
  Subscribe:
    enabled: False
```

**Figure 44. MLSysOps Agent configuration file example.**

## 8.6 Northbound API & Command-line Interface

**The Northbound API (NB API)** serves as the main interface for external systems—such as user interfaces and automation tools—to interact with the MLSysOps agent-based orchestration framework. Designed as an HTTP-based RESTful service, it enables users to send commands, retrieve information, and monitor overall system status.

The NB API operates on a predefined IP and port, supporting secure, asynchronous communication with the Continuum Agent. This design allows for a modular and scalable control layer, effectively abstracting the internal complexity of the multi-agent system. As a result, it offers a clean, service-oriented interface for seamless integration with external management tools.

To ensure clarity and maintainability, the NB API is structured into four main categories, each aligned with a specific operational domain of the system. This modular organization reflects the core responsibilities and lifecycle stages of the MLSysOps framework, facilitating consistent and intuitive interaction for all users and systems.

**Applications:** Manage the lifecycle of deployed applications—from deployment to monitoring and removal.

[7] This is an indicative example, using values valid at the time of writing this document. The finalized version is expected to be provided with the official release of the open-source project.

| Method | Endpoint | Description |
|---|---|---|
| POST | /apps/deploy | Deploy an application. Requires app description in request body. |
| GET | /apps/list_all/ | Retrieve a list of all deployed applications in the framework. |
| GET | /apps/status/{app_id} | Get the current status of a specific application. |
| GET | /apps/apps/details/{app_id} | Fetch detailed metadata of an application. |
| GET | /apps/performance/{app_id} | Access performance metrics of a deployed application. |
| DELETE | /apps/remove/{app_id} | Remove (undeploy) a specific application. |

**ML Models:** Control the deployment and lifecycle of machine learning models integrated into the system.

| Method | Endpoint | Description |
|---|---|---|
| POST | /ml/deploy_ml | Deploy a machine learning model to the infrastructure. |
| GET | /ml/list_all/ | List all currently deployed ML models. |
| GET | /ml/status/{model_uid} | Check the status of deployment of an ML model. |
| DELETE | /ml/remove/{model_uid} | Remove an ML model from the system. |

**Infrastructure:** Register, list, and manage edge, cluster, and datacenter components that make up the continuum.

| Method | Endpoint | Description |
|---|---|---|
| POST | /infra/register | Register infrastructure components (edge node, cluster, etc.). |
| GET | /infra/list/ | List all registered infrastructure components. |
| DELETE | /infra/unregister/{infra_id} | Unregister and remove an infrastructure component |

**Management:** System-level controls for health checks and operational mode switching.

| Method | Endpoint | Description |
|---|---|---|
| GET | /manage/ping | Check continuum agent status (ping the continuum agent) |
| PUT | /manage/mode/{mode} | Change operational mode of the Agent (Heuristic or ML ). |

Each group exposes RESTful endpoints designed to facilitate interaction with the corresponding agents in the MLSysOps architecture.

The **MLS CLI** is a Python-based command-line tool designed for managing application deployments and system interactions within the **MLSysOps framework**. It provides functionalities for deploying and managing applications, infrastructure components, and machine learning (ML) models, as well as for querying system status and monitoring deployments. The CLI communicates with the **Northbound API**, enabling efficient interaction with the MLSysOps framework.

Key functionalities provided by the MLS CLI through the NB API include service health checks (ping), application deployment, and retrieval of system, application, and ML model statuses. This tool streamlines deployment and management workflows, offering an intuitive interface for interacting with the agent-based framework while ensuring efficient system operations.

The CLI is organized into distinct command groups, each responsible for managing a specific aspect of the system:

- **apps**: Manage application deployment, monitoring, and removal
- **infra**: Register and manage infrastructure components across the continuum
- **ml**: Handle the deployment and lifecycle of machine learning models
- **manage**: Perform general system operations, such as health checks and mode switching

This structured CLI design ensures that different user roles can efficiently interact with the system based on their specific needs, further reinforcing the modular and scalable nature of the **MLSysOps framework**.

The table presents an overview of the CLI commands currently available. These commands are indicative and may be updated or extended in the open-source release

**Table 18. CLI Commands.**

| Group | Command | Description | Parameters |
|---|---|---|---|
| APP | deploy-app | Deploy an application using a YAML file | yaml file using path or uri |
| | list-all | list the applications on the system | - |
| | get-app-status | Get the status of the application | App_id |
| | get-app-details | Get the details | App_id |
| | get-app-performance | get the performance metric of an application | App_id |
| | remove-app | removes an application from the system | App_id |
| INFRA | register-infra | register system description | yaml file using path or uri |
| | list | list infrastructure registered | infra_id (Datacenter or cluster id) |
| | unregister-infra | remove system description | infra_id |
| Management | set-mode | change between ML or Heuristic-normal mode | 0 for heuristic 1 for ML |

| | | | |
|---|---|---|---|
| /config | Set System Target | set infrastructure level targets | list of ids and list of targets |
| | Config Trust | Configure trust assesment | list of nodes id list of index and list of weights |
| | Ping | Ping the continuum agent | - |

| | | | |
|---|---|---|---|
| ML | deploy-ml | Deploy an application using a YAML file | yaml file using path or uri |
| | list-all | list the ml models deployed on the system | - |
| | get-status | Get the status of the ml models | model_uid |
| | remove-ml | removes a ml model from the system | model_uid |

# 9 Software Components

In this chapter, we include a detailed description of each component through separate tables detailing each component with information regarding the module name, system component, MLSysOps inventory ID, the responsible partners, main functions, submodules and interactions between submodules, layer, interfaces (internal and external), and execution environment requirements. The classification of each mechanism per system component follows the high-level architectural hierarchy and classifies each module to one of the following system components: System Actors, Continuum Orchestrator, Cluster Orchestrator, Orchestration Service, Continuum-level Telemetry, Cluster-level Telemetry and Node-level Telemetry.

For each component listed below, the IP status is indicated (foreground or background) along with whether it is currently open source, planned to be open source, not intended to be open source, or still undecided.

The rationale behind this structured presentation is to provide a clear and concise identification and understanding of each component's functionality and interactions, ensuring comprehensive documentation and easier reference.

## 9.1 Low-level container runtime

| **Module Name** | **System component** | **MLSysOps Inventory ID** | **Responsible Partner(s)** |
|---|---|---|---|
| Sandboxed Container Runtime (SCR) | **Orchestration Service** | 3.03.1 | NUBIS |
| **Description of the module** | | | |
| Main functions | Spawn Containers in a sandbox environment | | |
| Submodules | • Sandbox mechanism (hypervisor, secure enclave in software etc.)<br>• Container library (spawns a generic container) | | |
| Interaction between sub-modules | N/A | | |
| Layer | | | |
| **Requirements** | | | |
| RG2 (R2.1, R2.2) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • At least 4 CPU cores, 8GB of memory, 100GB storage<br>• Generic Linux system (systemd) running a variant of k8s / k3s etc.) | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| https://github.com/kata-containers/kata-containers | Background | yes | |
| https://github.com/nubificus/urunc | Foreground | yes | |

| | | |
|---|---|---|
| https://github.com/kata-containers/kata-containers/pull/8070 | Foreground | yes |
| https://github.com/nubificus/kata-containers/tree/rs-vaccel-fusion | Foreground | yes |
| **Notes** | | |
| | | |

## 9.2 Mechanism for Software Deployment and Orchestration

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Cluster Manager | **Continuum Orchestrator** | 3.01.1 | UTH |
| **Description of the module** | | | |
| Main functions | • Deploy/remove/relocate application components | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Cloud Infrastructure, Edge Infrastructure, Edge | | |
| **Requirements** | | | |
| RG2 (R2.1, R2.3) | | | |
| **Execution environment requirements (HW,SW)** | | | |
| • Control plane: At least 8GB of memory, 100GB storage<br>• Generic Linux system (systemd) running a variant of k3s. | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| https://mlsysops-gitlab.e-ce.uth.gr/mechanisms/fluidity | Foreground | Yes | |
| **Notes** | | | |
| Available at https://github.com/mlsysops-eu/mlsysops-framework. | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Node Manager | **Orchestration Service** | 3.03.9 | UTH |
| **Description of the module** | | | |

| | | | |
|---|---|---|---|
| Main functions | It uses the MLSYSOPS Node Mechanisms SDK to perform configurations on the node, based on the commands received from the Cluster Manager:<br>• Change CPU frequency<br>• Redirect application-level traffic between different network interfaces | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Cloud Infrastructure, Edge Infrastructure, Edge | | |
| **Requirements** | | | |
| RG2 (R2.2) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • Worker nodes: At least 1GB of memory and 16GB storage.<br>• Generic Linux system (systemd) running a variant of k3s. | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| https://mlsysops-gitlab.e-ce.uth.gr/mechanisms/fluidity-node-proxy | Foreground | Yes | |
| **Notes** | | | |
| It will be integrated into MLSysOps Node Agent, available at https://github.com/mlsysops-eu/mlsysops-framework. | | | |

| **Module Name** | **System component** | **MLSysOps Inventory ID** | **Responsible Partner(s)** |
|---|---|---|---|
| Continuum Orchestrator (karmada) | Continuum Orchestrator | 3.03.2 | NUBIS/UTH |
| **Description of the module** | | | |
| Main functions | Orchestrate the deployment of application and service descriptions based on the MLSysOps agents | | |
| Submodules | N/A | | |
| Interaction between sub-modules | Node-level Orchestrator, Cluster Orchestrator, MLSysOps Agents | | |
| Layer | Cluster, Cloud, Edge | | |
| **Requirements** | | | |
| RG2 (R2.1, R2.3) | | | |
| **Execution environment requirements (HW, SW)** | | | |

| | | |
|---|---|---|
| • At least 2 cores (virtual/physical), at least 4 GB RAM<br>• Tested on Ubuntu 20.04 and 22.04 | | |
| **Source code** | | |
| Link | IP | Open - Source |
| https://github.com/karmada-io/karmada | Background | yes |
| **Notes** | | |
| Will have to fork the repo to the MLSysOps Github organization, as custom changes related to MLSysOps are required. We have already identified bugs related to submariner (the networking component), as well as enhancements that will be proposed to be upstreamed. To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework), as a tagged version with scripts to be setup and configured for MLSysOps automatically. | | |

## 9.3 Deployment extensions for Far-Edge node

| **Module Name** | **System component** | **MLSysOps Inventory ID** | **Responsible Partner(s)** |
|---|---|---|---|
| Far-Edge Kubelet | **Orchestration Service** | 3.03.5 | FhP-AICOS |
| **Description of the module** | | | |
| Main functions | Bridges the Kubernetes API with the Far-Edge proxy (NextGenGW) API | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Smart-Edge, Edge | | |
| **Requirements** | | | |
| RG2 (R2.1), RG3 (R3.6) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • SW: Generic Linux system running a variant of k8s (not tested with k3s)<br>• HW: Hardware with the lowest resource in which it was tested: RPi4 (slow but functional) | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | Foreground | Yes | |
| **Notes** | | | |
| Not yet migrated to MLSysOps gitlab. To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Far-Edge node watcher | **Orchestration Service** | 3.03.6 | FhP-AICOS |
| **Description of the module** | | | |
| Main functions | Subscribes for Far-Edge node registration updates on the NextGenGW and propagates them into the Kubernetes environment by creating or deleting Far-Edge kubelets, as well as the Far-Edge Proxy agent. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Smart-Edge, Edge | | |
| **Requirements** | | | |
| RG2 (R2.1), RG3 (R3.6) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • SW: Linux OS with Python 3.8, Kubernetes library for Python, Paho MQTT library for Python<br>• HW: Hardware with the lowest resource in which it was tested: RPi4 | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | Foreground | Yes | |
| **Notes** | | | |
| Not yet migrated to MLSysOps gitlab. To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| embServe | **Orchestration Service** | 3.03.7 | FhP-AICOS |
| **Description of the module** | | | |
| Main functions | Runtime that adds service deployment capabilities to resource constrained Far-Edge nodes. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Far-Edge | | |

| Requirements | | |
|---|---|---|
| RG2 (R2.1, R2.2), RG3 (R3.1) | | |
| **Execution environment requirements (HW, SW)** | | |
| • SW: Zephyr OS<br>• HW: nRF52840 | | |
| **Source code** | | |
| Link | IP | Open - Source |
| N/A | Background | Not yet decided |
| **Notes** | | |
| This software module is part of FhP-AICOS background knowledge. We are discussing internally if we provide it in open source or binary. | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| NextGenGW | **Orchestration Service** | 3.03.8 | FhP-AICOS |
| **Description of the module** | | | |
| Main functions | Framework that homogenises Far-Edge node heterogeneity in a single communication protocol and data model. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Smart Edge, Edge | | |
| **Requirements** | | | |
| RG2 (R2.1), RG3 (R3.1, R3.5) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| SW: Generic Linux OS<br><br>HW: Hardware with the lowest resource in which it was tested: RPi4 | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | Background | Not yet decided | |
| **Notes** | | | |
| This software module is part of FhP-AICOS background knowledge. We are discussing internally if we provide it in open source or binary. | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Far-Edge Proxy Agent | **Node-level Telemetry Node-level Control Knob** | 3.03.9 | FhP-AICOS |
| **Description of the module** | | | |
| Main functions | Bridges NextGenGW API with MLSysOps agent framework. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Smart Edge, Edge | | |
| **Requirements** | | | |
| RG3 (R3.1, R3.5) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| SW: Generic Linux OS with Python 3.8, Kubernetes library for Python, Paho MQTT library for Python<br>HW: Hardware with the lowest resource in which it was tested: RPi4 | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | Foreground | Yes | |
| **Notes** | | | |
| Not yet migrated to MLSysOps gitlab. To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Layer agnostic workload migration | **Orchestration Service** | 3.03.10 | FhP-AICOS |
| **Description of the module** | | | |
| Main functions | Mechanism that allows the transparent migration of workloads between the Far-Edge and the other layers of the continuum. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |

| | | | |
|---|---|---|---|
| Layer | Smart Edge, Edge | | |
| **Requirements** | | | |
| RG2 (R2.1), RG3 (R3.6) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| SW: Generic Linux system running a variant of k8s<br>HW: Hardware with the lowest resource in which it was tested: Intel NUC 13 Pro 16GB DDR4 RAM 256GB NVMe | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | Foreground | Yes | |
| **Notes** | | | |
| Not yet migrated to MLSysOps gitlab, to be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

## 9.4 5G UPF analysis

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| 5G UPF analysis | **Cluster-level Telemetry** | 3.05.1 | NTT |
| **Description of the module** | | | |
| Main functions | Analyse target data center and apply ML logic for UPF switch | | |
| Submodules | • ML algorithm<br>• Agent | | |
| Interaction between sub-modules | Agent sends metrics to ML module | | |
| Layer | Cluster layers | | |
| **Requirements** | | | |
| RG7 | | | |
| **Execution environment requirements (HW, SW)** | | | |
| 10 vCPU, 32 GB RAM, 100 GB storage. Ubuntu 22.04.3 | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | Foreground | No | |

| Notes |
|---|
| For evidence here the demo video: https://youtu.be/EujiS2twBvI |

## 9.5 5G logs/metrics analysis for AAD

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| AAD 5G networks | **Cluster-level Telemetry** | 3.05.2 | INRIA |
| **Description of the module** | | | |
| Main functions | Analyse metrics from network and UE level and apply ML logic for attack detection | | |
| Submodules | • ML algorithm<br>• Agent | | |
| Interaction between sub-modules | Agent sends metrics to ML module | | |
| Layer | Cluster layers | | |
| **Requirements** | | | |
| RG7 | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • Ubuntu 22.04.3<br>• Python version 3.10+<br>• BladeRF SDR for RF signals<br>• Open5Gs for 5G core functionality<br>• Android version 12+ for UE | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| Not yet available | Foreground | Planned to be (not yet released) | |
| **Notes** | | | |
| | | | |

## 9.6 Agent framework mechanisms

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Communication Module | **System Actors** | 3.00.1 | UNICAL |
| **Description of the module** | | | |
| Main functions | Communication protocol between agents and the system. | | |

| | |
|---|---|
| Submodules | • XMPP service containerized (handle the communication between agents).<br>• Redis (to handle asynchronous calls and system information). |
| Interaction between sub-modules | N/A |
| Layer | Continuum and Cluster layers |
| **Requirements** | |
| RG2 (2.2) | |
| **Execution environment requirements (HW, SW)** | |
| • XMPP service configured and running on a container.<br>• Python Version 3.10<br>• Python libraries: spade, redis, aioxmpp. | |

| **Source code** | | |
|---|---|---|
| Link | IP | Open – Source |
| https://mlsysops-gitlab.e-ce.uth.gr/agent/containers | Foreground | Yes |
| **Notes** | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | |

| **Module Name** | **System component** | **MLSysOps Inventory ID** | **Responsible Partner(s)** |
|---|---|---|---|
| Northbound API | **System Actors** | 3.00.2 | UNICAL |
| **Description of the module** | | | |
| Main functions | Integrate the CLI user interface with the Agent Framework. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Continuum | | |
| **Requirements** | | | |
| RG2 (2.2) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • Host running the API endpoints.<br>• Python Version 3.10<br>• Python libraries: fast-api. | | | |

| Source code | | |
|---|---|---|
| Link | IP | Open – Source |
| https://mlsysops-gitlab.e-ce.uth.gr/agent/northbound-api | Foreground | Yes |
| **Notes** | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| CLI Module | **System Actors** | 3.00.3 | UNICAL |
| **Description of the module** | | | |
| Main functions | Facilitate interaction between the user and the application | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Continuum, cluster, node layers | | |
| **Requirements** | | | |
| RG2 (2.2) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • Python Version 3.10<br>• Python libraries: click, requests, yaml, json | | | |
| **Source code** | | | |
| Link | IP | Open – Source | |
| https://mlsysops-gitlab.e-ce.uth.gr/agent/mls-cli | Foreground | yes | |
| **Notes** | | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Agent Module | **System Actors** | 3.00.4 | UNICAL |
| **Description of the module** | | | |
| Main functions | Manage the orchestration of the system | | |

| | | | |
|---|---|---|---|
| Submodules | • Yaml file parser<br>• Deployment management<br>• Telemetry Data Handler | | |
| Interaction between sub-modules | N/A | | |
| Layer | Continuum, cluster, node layers | | |
| **Requirements** | | | |
| RG2 (2.1) | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • Server running xmpp service for agent communication.<br>• Python Version 3.10<br>• Python libraries: Spade | | | |
| **Source code** | | | |
| Link | IP | Open – Source | |
| https://mlsysops-gitlab.e-ce.uth.gr/agent | Foreground | yes | |
| **Notes** | | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

## 9.7 Object Storage

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Object Storage Gateway | **System Actors** | 3.00.6 | CC |
| **Description of the module** | | | |
| Main functions | Provides an S3-compatible object storage interface for MLSysOps applications | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Cloud Infrastructure, Edge Infrastructure | | |
| **Requirements** | | | |
| R4.1, R4.2, R4.4 | | | |
| **Execution environment requirements (HW, SW)** | | | |

| | | |
|---|---|---|
| • Python Version 3.11<br>• Python libraries: fastapi, gunicorn, starlette, uvicorn, fastapi-xml, aiohttp, pycryptodome<br>• Docker | | |
| **Source code** | | |
| Link | IP | Open – Source |
| https://mlsysops-gitlab.e-ce.uth.gr/storage/gateway | Foreground | No |
| **Notes** | | |
| | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Bucket Policy Generator | **System Actors** | 3.00.7 | CC |
| **Description of the module** | | | |
| Main functions | Triggers changing the storage policy of buckets based on real-time traffic | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Cloud Infrastructure, Edge Infrastructure | | |
| **Requirements** | | | |
| R4.1, R4.2, R4.4 | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • Python Version 3.11<br>• Python libraries: fastapi, tbd<br>• Docker | | | |
| **Source code** | | | |
| Link | IP | Open – Source | |
| https://mlsysops-gitlab.e-ce.uth.gr/storage/policy-generator | Foreground | yes | |
| **Notes** | | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | | |

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| Edge Storage | **System Actors** | 3.00.8 | CC |

| Description of the module | | |
|---|---|---|
| Main functions | Stores encrypted fragments of Object Storage objects | |
| Submodules | N/A | |
| Interaction between sub-modules | N/A | |
| Layer | Edge Infrastructure | |
| **Requirements** | | |
| R4.1, R4.2 | | |
| **Execution environment requirements (HW, SW)** | | |
| • MinIO | | |
| **Source code** | | |
| Link | IP | Open - Source |
| https://mlsysops-gitlab.e-ce.uth.gr/storage/edge-storage-node | Foreground | No |
| **Notes** | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | |

## 9.8 Telemetry library

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| MLSYSOPS Telemetry SDK Library | **Continuum-level Telemetry** | 3.07.1 | UTH |
| **Description of the module** | | | |
| Main functions | Offers an abstraction API for the OpenTelemetry SDK/API C++ & Python libraries. It can be used to push telemetry metrics and logs to the telemetry system. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | All | | |
| **Requirements** | | | |
| N/A | | | |
| **Execution environment requirements (HW, SW)** | | | |

- SW: It can be integrated in any software written in Python or C++.
- HW: None.

| Source code | | |
|---|---|---|
| Link | IP | Open - Source |
| https://mlsysops-gitlab.e-ce.uth.gr/telemetry/mls-telemetry-library | Foreground | Yes |
| https://mlsysops-gitlab.e-ce.uth.gr/telemetry/mlsysops-python-telemetry-library | Foreground | Yes |

| Notes |
|---|
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) |

## 9.9 Computation resource management

| Module Name | System component | MLSysOps Inventory ID | Responsible Partner(s) |
|---|---|---|---|
| MLSYSOPS Node Level Mechanisms SDK | **Orchestration Service** | 3.03.9 | UTH |

| Description of the module | |
|---|---|
| Main functions | It includes multiple functionality:<br>• **CPU/GPU frequency management**: It exposes a simple interface to interact with CPU & NVIDIA GPU frequency configuration libraries.<br>• **Network interface redirection**: A simple interface to change the network interfaces used by a container. |
| Submodules | N/A |
| Interaction between sub-modules | N/A |
| Layer | Node |

| Requirements |
|---|
| R3.1, R3.2, R3.3 |

| Execution environment requirements (HW, SW) |
|---|
| • SW: It can be integrated in any software written in Python. The underlying OS should be Linux based and have the cpufreq linux kernel module enabled. The GPU configuration capability needs the NVIDIA Management Library (NVML). |

| | | |
|---|---|---|
| • HW: Intel, AMD and ARM CPUs. NVIDIA GPUs.<br>• Network interfaces: It can change between different network connections: Wi-Fi, 4G, Ethernet. | | |
| **Source code** | | |
| Link | IP | Open - Source |
| https://mlsysops-gitlab.e-ce.uth.gr/mechanisms/node-level-mechanisms | Foreground | Yes |
| **Notes** | | |
| To be included in the open-source release (provisional link: https://github.com/mlsysops-eu/mlsysops-framework) | | |

| **Module Name** | **System component** | **MLSysOps Inventory ID** | **Responsible Partner(s)** |
|---|---|---|---|
| Job scheduler for DC with reconfigurable network topologies | System Actors | 3.00.5 | NVIDIA |
| **Description of the module** | | | |
| Main functions | Schedules AI training jobs on Datacenter clusters and appropriately configures network topology to reflect the job communication pattern. | | |
| Submodules | N/A | | |
| Interaction between sub-modules | N/A | | |
| Layer | Datacenter infrastructure | | |
| **Requirements** | | | |
| R8.2 | | | |
| **Execution environment requirements (HW, SW)** | | | |
| • NVIDIA Cloud native platform | | | |
| **Source code** | | | |
| Link | IP | Open - Source | |
| N/A | N/A | No | |
| **Notes** | | | |
| Source code is not yet publicly available | | | |

| **Module Name** | **System component** | **MLSysOps Inventory ID** | **Responsible Partner(s)** |
|---|---|---|---|
| vAccel | System Actors | 3.00.9 | NUBIS |
| **Description of the module** | | | |

| | |
|---|---|
| Main functions | vAccel is an open-source framework designed to enable flexible execution by mapping hardware-accelerate-able workloads to relevant hardware functions, thus decoupling applications from hardware-specific code. It aims at enhancing security by ensuring that consecutive executions on a hardware-accelerated platform do not leak sensitive data. |
| Submodules | Building on the vAccel framework, we develop custom mechanisms to enable runtime switching between vAccel plugins, as well as customized plugins, tailored to the MLSysOps UCs. |
| Interaction between sub-modules | Node-level Orchestrator, Cluster Orchestrator |
| Layer | Cluster, Cloud, Edge |
| **Requirements** | |
| R3.2, R3.3, R3.6 | |
| **Execution environment requirements (HW, SW)** | |
| • No specific requirements (diverse architecture support: amd64, aarch64, armv7l)<br>• Shared library environment (no alpine/statically linked envs, eg musl) | |

| **Source code** | | |
|---|---|---|
| Link | IP | Open - Source |
| https://github.com/nubificus/vaccel | Background | yes, enhanced |
| https://github.com/nubificus/vaccel-rust | Foreground | yes |
| https://github.com/nubificus/opencv-vaccel | Foreground | yes |
| https://github.com/nubificus/vaccel/pull/71 | Foreground | yes, merged |
| https://github.com/nubificus/vaccel/pull/97 | Foreground | Yes, merged |
| **Notes** | | |
| | | |

## 9.10 Requirements Map

The functionality that each components delivers, comes to address the functional requirements that were identified during the project execution. Table 19 maps the components that were described earlier in Section 9 with the identified requirements.

**Table 19 Requirements to components map**

| Inventory ID | Module Name | Requirement Group. (Requirements) |
|---|---|---|
| 3.03.1 | Sandboxed Container Runtime (SCR) | RG2 (R2.1, R2.2) |

| | | |
|---|---|---|
| 3.01.1 | Cluster Manager | RG2 (R2.1, R2.3) |
| 3.03.9 | Node Manager | RG2 (R2.2) |
| 3.03.2. | Continuum Orchestrator (karmada) | RG2 (R2.1, R2.3) |
| 3.03.5 | Far-Edge Kubele | RG2 (R2.1), RG3 (R3.6) |
| 3.03.6 | Far-Edge node watcher. | RG2 (R2.1), RG3 (R3.6) |
| 3.03.7 | embServe | RG2 (R2.1, R2.2), RG3 (R3.1) |
| 3.03.8 | NextGenGW | RG2 (R2.1), RG3 (R3.1, R3.5) |
| 3.06.1 | Far-Edge Telemetry and control knobs bridge | RG3 (R3.1, R3.5) |
| 3.05.1 | 5G UPF analysis | RG7 |
| 3.05.2 | AAD 5G networks | RG7 |
| 3.00.1 | Communication Module | RG2 (2.2) |
| 3.00.2. | Northbound API | RG2 (2.2) |
| 3.00.3 | CLI Module | RG2 (2.2) |
| 3.00.4 | Agent Module | RG2 (2.1) |
| 3.00.6 | Object Storage Gateway | RG4 (R4.1, R4.2, R4.4) |
| 3.00.7 | Bucket Policy Generator | RG4 (R4.1, R4.2, R4.4) |
| 3.00.8 | **Edge Storage** | RG4 (R4.1, R4.2) |
| 3.07.1 | MLSYSOPS Telemetry SDK Library | |
| 3.03.9 | MLSYSOPS Node Level Mechanisms SDK | RG3 (R3.1, R3.2, R3.3) |
| 3.00.5 | Job scheduler for DC with reconfigurable network topologies | RG8 (R8.2) |
| 3.00.9 | vAccel | RG3 (R3.2, R3.3, R3.6) |

# 10 Conclusion

MLSysOps is a project aimed at improving the efficiency and adaptability of cloud and edge computing environments by leveraging AI and ML for enhanced resource management. This document outlines the final design and implementation of the AI-ready mechanisms for the MLSysOps framework, which enables intelligent resource management across cloud-edge systems.

One key contribution of the MLSysOps framework is the telemetry system, which allows for fine-grained observability and node-level configuration. This system is crucial for monitoring and managing resources effectively. Additionally, the framework incorporates platform-specific mechanisms for x86-64, ARM, GPUs, and MPSoC platforms. These mechanisms dynamically control compute capabilities, ensuring optimal performance across different types of hardware.

Another significant aspect of the MLSysOps framework is the extension of the SkyFlok storage system. This extension utilizes ML agents for adaptive policy reconfiguration, making the storage system more responsive and efficient. Networking mechanisms are also integrated into the framework to facilitate datacenter job scheduling and 5G/wireless edge control, enhancing connectivity and data flow management.

The orchestration layer within the framework is designed to integrate various deployment tools, including container runtimes, serverless functions, unikernels, and far-edge deployment tools. This layer ensures secure sandboxing and accelerated execution paths, providing a robust environment for deploying and managing applications.

The MLSysOps framework employs an agent-based architecture and MAPE loops for decision-making processes. Plugins are available for extensibility in policies and mechanisms, allowing for customization based on specific needs. Furthermore, trust management, telemetry adaptation, and serverless-aware deployments contribute to enhancing autonomy, explainability, and adaptability in heterogeneous infrastructures. Together, these features establish a solid foundation for future ML-driven operations.

In conclusion, the MLSysOps framework represents a significant advancement in intelligent resource management for cloud and edge computing environments. Integrating advanced telemetry systems, platform-specific mechanisms, adaptive storage solutions, and comprehensive networking strategies, it provides a dynamic and efficient approach to modern computing challenges. The orchestration layer's ability to seamlessly incorporate diverse deployment tools further amplifies its robustness and flexibility. The MLSysOps framework uses a smart design with agents, MAPE loops, and customizable plugins to boost efficiency in different systems. With rising computing needs, MLSysOps will play a key role in advancing operations driven by machine learning.

# 11 Bibliography

[1] [Online]. Available: https://mlsysops-gitlab.e-ce.uth.gr.

[2] [Online]. Available: https://github.com/mlsysops-eu.

[3] "opentelemetry," [Online]. Available: https://opentelemetry.io/.

[4] "opentelemetry collector," [Online]. Available: https://github.com/open-telemetry/opentelemetry-collector.

[5] "https://grafana.com/oss/mimir/," Grafana Labs. [Online]. [Accessed 2025].

[6] "https://prometheus.io/docs/prometheus/latest/querying/basics/," Grafana Labs. [Online]. [Accessed 2025].

[7] "https://grafana.com/grafana/," Grafana Labs. [Online].

[8] "https://grafana.com/oss/loki/," Grafana Labs. [Online]. [Accessed 2025].

[9] "https://grafana.com/oss/tempo/," Grafana Labs. [Online]. [Accessed 2025].

[10] "Prometheus," [Online]. Available: https://prometheus.io/docs/guides/node-exporter/.

[11] "NVIDIA GPU Exporter," [Online]. Available: https://github.com/utkuozdemir/nvidia_gpu_exporter.

[12] "NVML - NVIDIA Management Library," [Online]. Available: https://developer.nvidia.com/nvidia-management-library-nvml.

[13] "pyNVML," [Online]. Available: https://pypi.org/project/pynvml/.

[14] "dfx - Dynamic Function Exchange," [Online]. Available: https://www.xilinx.com/video/hardware/dynamic-function-exchange-dfx.html.

[15] "vAccel," [Online]. Available: https://vaccel.org/. [Accessed 28 12 2023].

[16] "Skyflok," [Online]. Available: https://www.skyflok.com/. [Accessed 28 12 2023].

[17] F. Pournaropoulos, A. Patras, C. D. Antonopoulos, N. Bellas, S. Lalis, "Fluidity: Providing flexible deployment and adaptation policy experimentation for serverless and distributed applications spanning cloud–edge–mobile environments," Future Generation Computer Systems (FGCS), 2024. [Online]. Available: https://doi.org/10.1016/j.future.2024.03.031. [Accessed 10 April 2025].

[18] "k3s," [Online]. Available: https://k3s.io/.

[19] [Online]. Available: https://karmada.io/.

[20] “flannel,” [Online]. Available: https://github.com/flannel-io/flannel.

[21] *Thönes, Johannes. "Microservices." IEEE software 32.1 (2015): 116-116..*

[22] [Online]. Available: https://slurm.schedmd.com/.

[23] D. Sabella, A. Vaillant, P. Kuure, U. Rauschenback and F. Giust, “Mobile-edge computing architecture: The role of MEC in the Internet of Things,” *IEEE Consumer Electronics Magazine,* vol. 5, no. 4, pp. 84--91, 2016.

[24] [Online]. Available: https://www.cncf.io/about/faq.

[25] X. Lin, L. Lei, Y. Wang, J. Jing, K. Sun, Q. Zhou, X. Lin, L. Lei, Y. Wang and Ji, “A measurement study on linux container security: Attacks and countermeasures,” in *Proceedings of the 34th Annual Computer Security Applications Conference*, 2018, pp. 418--429.

[26] “AWS Lambda,” [Online]. Available: https://aws.amazon.com/lambda. Accessed: .

[27] “AWS FIRECRACKER,” [Online]. Available: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing. [Accessed 15 11 2023].

[28] E. J. Patterson, J. Schleier-Smith, V. Sreekanti, C.-C. Tsai, A. K. Qifan, P. Vaishaal, S. Joao, M. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. A. Popa, I. Stoica and D. A., “Cloud Programming Simplified: A Berkeley View on Serverless Computing,” *arXiv preprint arXiv:1902.03383,* 2019.

[29] “QEMU Internals,” [Online]. Available: http://blog.vmsplice.net/2011/09/qemu-internals-vhost-architecture.html. [Accessed 3 15 2023].

[30] D. Williams, R. Koller, M. Lucina and N. Prakash, “Unikernels as processes,” in *Proceedings of the ACM Symposium on Cloud Computing,* 2018, pp. 199-211.

[31] “cloud-hypervisor,” [Online]. Available: https://github.com/cloud-hypervisor/cloud-hypervisor. [Accessed 20 11 2023].

[32] “crosvm,” [Online]. Available: https://crosvm.dev/book/.

[33] “KATA Containers,” [Online]. Available: https://katacontainers.io/.

[34] “Firecracker – Lightweight Virtualization for Serverless Computing,” [Online]. Available: https://aws.amazon.com/blogs/aws/firecracker-lightweight-virtualization-for-serverless-computing/.

[35] [Online]. Available: https://github.com/Solo5/solo5. [Accessed 25 11 2023].

[36] S. Kuenzer, V.-A. Badoiu, H. Lefeuvre, S. Santhanam, A. Jung, G. Gain, C. Soldani, C. Lupu, S. Teodorescu and C. Raducanu, “Unikraft: fast, specialized unikernels the easy way,” in *Proceedings of the Sixteenth European Conference on Computer Systems*, 2021, pp. 376-394.

[37] A. Kantee and J. Cormack, “Rump kernels: no OS? no problems!,” *; login:: the magazine of USENIX \& SAGE*, vol. 39, no. 5, pp. 11-17, 2014.

[38] [Online]. Available: https://osv.io/. [Accessed 18 11 2023].

[39] “urunc,” [Online]. Available: https://blog.cloudkernels.net/posts/urunc.

[40] [Online]. Available: Mainas, C., Plakas, I., Ntoutsos, G., & Nanos, A. (2024). Sandboxing Functions for Efficient and Secure Multi-tenant Serverless Deployments. Zenodo. https://doi.org/10.5281/zenodo.11545513.

[41] [Online]. Available: https://knative.dev.

[42] [Online]. Available: https://github.com/knative/community/blob/main/working-groups/security/threat-model.md.

[43] [Online]. Available: https://kubernetes.io/blog/2021/04/15/three-tenancy-models-for-kubernetes.

[44] [Online]. Available: https://doi.org/10.1145/3342195.3392698.

[45] [Online]. Available: https://github.com/cloudkernels/vaccelrt/commit/ae2f46dead4355232c1412602e399e4b52bed4f5.

[46] [Online]. Available: https://github.com/cloudkernels/vaccelrt/commit/3ecabf25437892b5054673c00f8dc77ecd401442.

[47] [Online]. Available: https://github.com/cloudkernels/vaccelrt/blob/main/docs/vaccel_args.md.

[48] [Online]. Available: Goutha, M., Lagomatis, I., & Nanos, A. (2024). OpenCV hardware acceleration with vAccel. Zenodo. https://doi.org/10.5281/zenodo.12090076.

[49] J. Oliveira, F. Sousa and L. Almeida, “embServe: Embedded Services for Constrained Devices,” in *2023 IEEE 19th International Conference on Factory Communication Systems (WFCS)*, IEEE, 2023, pp. 1-8.

[50] “LWM2M,” [Online]. Available: https://www.openmobilealliance.org/release/LWM2M_SWMGMT/V1_0_2-20210119-A/HTML-Version/OMA-TS-LWM2M_SwMgmt-V1_0_2-20210119-A.html.

[51] C. Resende, W. Moreira and L. Almeida, “Nextgengw-a software framework based on mqtt and semantic definition format,” in *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, IEEE, 2023, pp. 141-148.

[52] [Online]. Available: https://mqtt.org/.

[53] [Online]. Available: https://www.ietf.org/archive/id/draft-onedm-t2trg-sdf-00.html#:~:text=The%20Semantic%20Definition%20Format%20(SDF,in%20the%20Internet%20of%20Things..

[54] [Online]. Available: https://istio.io/.

[55] J. Kephar and D. Chess, "The vision of autonomic computing," *Computer,* vol. 36, no. 1, pp. 41 - 50, 2003.

[56] "https://spade-mas.readthedocs.io/en/latest/model.html," SPADE Agent. [Online].

[57] "MINIO," [Online]. Available: https://min.io. [Accessed 29 10 2023].

[58] "zerotier," [Online]. Available: https://www.zerotier.com/.

[59] "JADE," [Online]. Available: https://jade.tilab.com/.

[60] "JASON," [Online]. Available: https://github.com/jason-lang/jason.

[61] "jacamo," [Online]. Available: https://github.com/jacamo-lang/jacamo.

[62] "pade," [Online]. Available: https://pade.readthedocs.io/en/latest/.

[63] "repast," [Online]. Available: https://repast.github.io/repast4py.site/index.html.

[64] "D2.2: Framework Architecture and APIs," 2023.

[65] "OpAMP," [Online]. Available: https://opentelemetry.io/docs/specs/opamp/#:~:text=Open%20Agent%20Management%20Protocol%20(OpAMP,package%20updates%20from%20the%20Server.. [Accessed 28 12 2023].

[66] "FIPA Foundation.," [Online]. Available: http://www.fipa.org.

[67] [Online]. Available: https://www.w3.org/TR/vocab-ssn/.

[68] "CCI," [Online]. Available: https://lov.linkeddata.es/dataset/lov/vocabs/ccp.

[69] "sdf objects," [Online]. Available: https://www.ietf.org/archive/id/draft-onedm-t2trg-sdf-00.html.

[70] "xmpp," [Online]. Available: https://xmpp.org/.

[71] N. B. Kalliopi Kravari, "A Survey of Agent Platforms," *Journal of Artificial Societies and Social Simulation,* 2015.

[72] [Online]. Available: https://spade-mas.readthedocs.io/en/latest/readme.html.

[73] [Online]. Available: http://www.fipa.org/specs/fipa00037/SC00037J.html.

[74] C. Pal, F. Leon, M. Paprzycki and M. Ganzha, “A Review of Platforms for the Development of Agent Systems,” *arXiv preprint arXiv:2007.08961,* 2020.

[75] Z. Wrona, W. Buchwald, M. Ganzha, M. Paprzycki, F. Leon, N. Noor and C.-V. Pal, “Overview of Software Agent Platforms Available in 2023,” *MDPI Information,* 2023.

[76] F. Bellifemine, G. Caire, A. Poggi and G. Rimassa, “JADE: A software framework for developing multi-agent applications. Lessons learned,” *Information and Software technology,* vol. 50, pp. 10-21, 2008.

[77] N. S. Boss, A. S. Jensen and J. Villadsen, “Building multi-agent systems using Jason,” *Annals of Mathematics and Artificial Intelligence,* vol. 59, pp. 373-388, 2010.

[78] R. H. Bordini and J. F. Hübner, “BDI agent programming in AgentSpeak using Jason,” in *International workshop on computational logic in multi-agent systems,* 2005.

[79] O. Boissier, R. H. Bordini, J. Hubner and Ricci, “Multi-agent oriented programming: programming multi-agent systems using JaCaMo,” 2020.

[80] X. H. Wang, D. Q. Zhang, T. Gu and H. K. Pung, “Ontology based context modeling and reasoning using OWL,” in *IEEE annual conference on pervasive computing and communications workshops,* 2004.

[81] Karmada. [Online]. Available: https://karmada.io/.

[82] “Karmada,” 2024. [Online]. Available: https://karmada.io/.

[83] “Karmada,” [Online]. Available: https://karmada.io/. [Accessed 21 6 2024].

[84] [Online]. Available: https://en.wikipedia.org/wiki/RCUDA.

[85] [Online]. Available: https://grpc.io/.

[86] [Online]. Available: https://www.cncf.io/blog/2023/12/12/karmada-brings-kubernetes-multi-cloud-capabilities-to-cncf-incubator.

[87] “flannel,” [Online]. Available: https://github.com/flannel-io/flannel.

END OF DOCUMENT