

PROJECT INFORMATION

Grant Agreement Nr: 101135012

Programme: HORIZON EUROPE

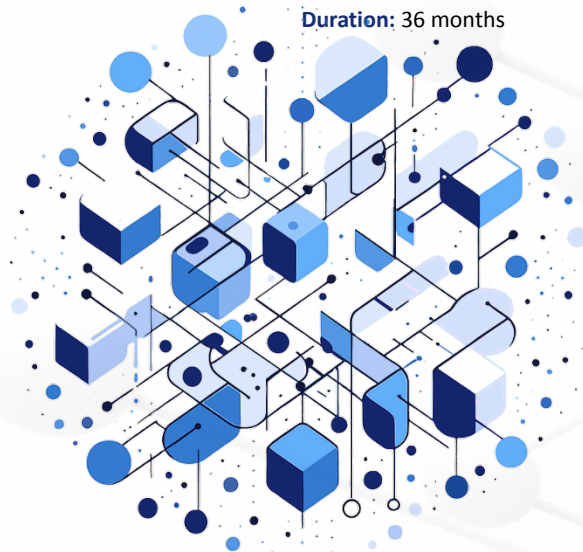
Call: HORIZON-CL4-2023-DATA-01

Topic: HORIZON-CL4-2023-DATA-01-04

Type of action: HORIZON Research and Innovation Actions

Start Date: 1 January 2024

Duration: 36 months



swarmchestrate

APPLICATION-LEVEL SWARM-BASED ORCHESTRATION ACROSS THE CLOUD-TO-EDGE CONTINUUM

D1.1 Logical Proximity and Distributed Matchmaking Algorithms

DOCUMENT INFORMATION

Work Package	WP1
Task	T1.1, T1.2, T1.3, T1.4, T1.5
Due Date	30/06/2025
Submission Date	01/07/2025
Lead Partner	TUB
Deliverable Type	R



Co-funded by
the European Union



UK Research
and Innovation

swarmchestrate.eu

DOCUMENT INFORMATION

Dissemination Level	PU
Version	0.1
Author(s)	Haci Ismail Aslan (TUB), Joel Witzke (TUB), Amy L. Murphy (FBK), Yiming Wang (FBK), Davide Berasi (FBK), Andras Markus (FEA)
Reviewer(s)	Jozsef Kovacs (SZTAKI) Andras Markus (FEA)

Document History

Version	Date	Contributor(s)	Short Description
0.1	01/06/2025	Haci Ismail Aslan (TUB) Joel Witzke (TUB)	Defined report structure and table of contents.
0.2	01/06/2025	Haci Ismail Aslan (TUB) Amy L. Murphy (FBK) Yiming Wang (FBK) Davide Berasi (FBK) Andras Markus (FEA)	Added Section 2 and revised Subsections 2.4 and Section 4.
0.3	05/06/2025	Haci Ismail Aslan (TUB) Amy L. Murphy (FBK) Yiming Wang (FBK)	Added Section 3, revised Section 4.
0.4	07/06/2025	Haci Ismail Aslan (TUB)	Added executive summary, Section 1 and 5.
0.5	09/06/2025	Haci Ismail Aslan (TUB) Amy L. Murphy (FBK) Andras Markus (FEA)	Version for internal peer review.
0.6	23/06/2025	Haci Ismail Aslan (TUB)	Implementing internal peer review comments.
1.0	29/06/2025	Haci Ismail Aslan (TUB) Amy L. Murphy (FBK)	Final version for submission.

Disclaimer

This document has been prepared by Swarmchestrate project partners as a result of work carried out within the Grant Agreement contract N° 101135012. Neither the Project Coordinator, nor any signatory party of the Swarmchestrate Project Consortium Agreement, nor any person acting on behalf of any of them makes any warranty or representation whatsoever, expressed or implied:

- with respect to the use of any information, apparatus, method, process, or similar item disclosed in this document, including merchantability and fitness for a particular purpose; or
- that such use does not infringe on or interfere with privately owned rights, including any party's intellectual property; or
- that this document is suitable to any particular user's circumstance; or
- assumes responsibility for any damages or other liability whatsoever (including any consequential damages) resulting from your selection to use this document or any information, apparatus, method, process, or similar item disclosed in this document.

The project is funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

This work was co-funded by UK Research and Innovation (UKRI) under the UK government's Horizon Europe funding guarantee (grant numbers 10101169 and 10102651).

This work was supported by Seoul National University (Institute of Engineering Research) and through grant NRF-RS-2023-00302083 by the National Research Foundation of Korea (NRF), which is funded by the Korea government, Ministry of Science and ICT (MSIT).

Copyright Notice

© 2024-2026 Swarmchestrate Consortium



Table of Contents

List of Figures.....	6
List of Tables.....	7
Abbreviations.....	8
Executive Summary.....	10
1. Introduction.....	11
1.1 Document Purpose.....	11
1.2 Document Structure.....	12
2. QoS-Driven Application Deployment in Swarmchestrator.....	13
2.1 State of the Art.....	13
2.2 Introduction to QoS-based Automated Application Deployment from Cloud to Edge.....	16
2.3 Logical Proximity.....	20
2.4 Simulating the Use of Logical Proximity.....	22
3. Demonstrating Intelligent Cloud-to-Edge Deployment.....	24
3.1 Background.....	24
3.1.1 Kubernetes.....	24
3.1.2 K3s.....	28
3.1.3 Raft.....	30
3.1.4 Clean Architecture.....	33
3.2 Functional and Non-functional Requirements.....	36
3.2.1 User-guided Domain Deployment (FR1).....	36
3.2.2 QoS-driven Deployment (FR2).....	36
3.2.3 Disruption Recovery and Redeployment (FR2).....	36
3.2.4 Non-functional requirement: Fault Tolerance (NFR1).....	36
3.3 Concept.....	37
3.3.1 Knowledge Base (KB).....	37
3.3.2 Resource Lead Agent.....	38
3.3.3 Resource Agent.....	39
3.4 Architecture.....	40
3.4.1 Infrastructure Model.....	41
3.4.2 Workload Model.....	42
3.5 Implementation.....	44
3.5.1 Knowledge Base.....	44
3.5.2 Resource Lead Agent.....	45
3.5.3 Resource Agent.....	48
3.6 Evaluation.....	50
3.6.1 Setup.....	50
3.6.2 Test Application.....	52

3.6.3 Evaluation Scenarios.....	53
3.6.4 Results and Observations.....	54
4. Runtime Optimization for Distributed Applications.....	59
4.1 Introduction.....	59
4.2 State of the Art.....	59
4.3 Problem Formulation and System Overview.....	62
4.3.1 AI.1 - Reconfiguration Trigger.....	63
4.3.2 AI.2 requirements prediction and AI.3 redeployment mechanism.....	65
5. Conclusions.....	66
6. References.....	67

List of Figures

Figure 2.1: Traditional deployment approach.....	18
Figure 2.2: An example offer provided by a resource agent.....	22
Figure 2.3: An example output of generated for ranking.....	23
Figure 3.1: A Kubernetes cluster and its components.....	25
Figure 3.2: Kubernetes Pod management [27].....	26
Figure 3.3: Kubernetes networking through Service [27].....	27
Figure 3.4: Kubernetes Ingress [27].....	27
Figure 3.5: K8s and K3s architecture comparison [8].....	29
Figure 3.6: K3s architecture.....	29
Figure 3.7: Replicated State Machine.....	31
Figure 3.8: Node states and the transitions among them.....	31
Figure 3.9: Clean architecture.....	34
Figure 3.10: Architecture Overview.....	39
Figure 3.11: Istio Bookinfo application architecture.....	54
Figure 3.12: Initial deployment of nine clusters (3 cloud, 3 fog, 3 edge).....	55
Figure 3.13: Final cluster topology after reducing worker nodes to 1 per cluster.....	56
Figure 3.14: Initial QoS-aware deployment to energy-efficient clusters (Task 1).....	57
Figure 3.15: Automatic application migration to performance-efficient clusters after QoS change.....	57
Figure 3.16: Rating component migrated from failed edge-performance to edge-energy.....	58
Figure 3.17: Raft election after the forced removal of the leader in cloud-energy, cloud performance becomes leader at term 7.....	59
Figure 4.1: Overview of the reconfiguration system.....	64

List of Tables

Table 3.1: Applications POST and PATCH endpoints' fields.....	48
Table 3.2: Resource Lead Agent REST API endpoints.....	48
Table 3.3: Assigned energy, cost, and bandwidth parameters for each cluster profile.....	52

Abbreviations

Term	Meaning
AI	Artificial Intelligence
API	Application Programming Interface
CD	Continuous Delivery
CI/CD	Continuous Integration / Continuous Delivery
CNI	Container Network Interface
CSP	Cloud Service Provider
FL	Federated Learning
FR	Functional Requirement
KB	Knowledge Base
K3s	Lightweight Kubernetes distribution for resource-constrained environments
K8s	Kubernetes
NFR	Non-Functional Requirement
OS	Operating System
P2P	Peer-to-Peer
PVC	Persistent Volume Claim
QoS	Quality of Service
RA	Resource Agent

Term	Meaning
RLA	Resource Lead Agent
RSM	Replicated State Machine
SLA	Service Level Agreement
TOSCA	Topology and Orchestration Specification for Cloud Applications
UUID	Universally Unique Identifier
WP	Work Package
YAML	YAML Ain't Markup Language (data serialization standard)

Executive Summary

This deliverable presents our work on resource selection based on the “logical proximity” notion and extends it by proposing strategies to orchestrate microservices in Cloud-to-Edge continuum, designed to support scalable, self-organizing deployment of microservices across the Cloud-to-Edge continuum. This deliverable presents a decentralized orchestration model developed under the Swarmchestrator project, inspired by swarm intelligence and designed to support scalable, self-organizing deployment of microservices across the Cloud-to-Edge continuum. At its core is the novel use of the *logical proximity* concept, which enables distributed matchmaking between application requirements and infrastructure capabilities. By quantifying resource suitability using multi-criteria cost functions and ranking methods such as Borda voting, the system autonomously forms optimal deployment groups (referred to as "swarms") that satisfy QoS goals like low latency, minimal energy use, and reduced cost.

To evaluate the feasibility and effectiveness of this approach, a full-stack prototype was developed and tested. Key achievements include:

- Design and implementation of a decentralized matchmaking algorithm using logical proximity and multi-objective QoS ranking.
- Development of a clean, modular orchestration framework based on containerized microservices and Kubernetes/K3s.
- Simulation-based validation of the matchmaking strategy using DISSECT-CF-Fog, showing effective swarm formation and resource selection.
- Robust fault-tolerant control plane based on Raft consensus for leadership among RLAs.
- Design of a system for runtime-optimization using AI techniques.

These contributions lay the groundwork for a self-adaptive, resilient orchestration layer that removes the need for centralized control or manual intervention during deployment. The system is designed to evolve toward runtime QoS adaptation and integration with broader orchestration mechanisms, making it a foundational element of the overall Swarmchestrator platform.

1. Introduction

1.1 Document Purpose

This deliverable, D1.1 Logical Proximity and Distributed Matchmaking Algorithms, provides the foundational principles, methodologies, and initial implementation details for the Swarmchestrator project's decentralized orchestration model across the Cloud-to-Edge continuum. It focuses on the development of intelligent matchmaking algorithms that consider Quality of Service (QoS) goals and logical proximity to enable dynamic, resilient, and optimized microservice deployment.

The purpose of this document is to:

- Introduce the logical proximity model as a decentralized decision-making mechanism.
- Present distributed matchmaking algorithms and their evaluation via simulation.
- Detail the implementation of intelligent application deployment across cloud, fog, and edge domains to provide a realistic test environment to further explore the concepts of logical proximity and swarm formation.
- Provide a basis for comparison with alternative approaches and prepare for integration with runtime optimization efforts described in later deliverables.

This deliverable is closely related to future system-wide orchestration strategies and complements deliverable D2.1, which provides the main architecture and TOSCA-based application specification format. Readers should refer to D2.1 for broader architectural context and application descriptors.

1.2 Document Structure

The structure of this document is as follows:

- **Section 2 – QoS-Driven Application Deployment in Swarmchestrator:** Reviews the state of the art in QoS-aware deployment frameworks and introduces Swarmchestrator’s approach to logical proximity and distributed matchmaking. It also includes a simulation-based evaluation of the proposed methods.
- **Section 3 – Demonstrating Intelligent Cloud-to-Edge Deployment:** Introduces a test bed as a feasibility exercise and details the technical architecture, functional and non-functional requirements, and implementation of the test bed methods. This section illustrates how Swarmchestrator operationalizes its deployment strategy using a test architecture based on container orchestration tools like Kubernetes and K3s.
- **Section 4 – Runtime Optimization for Distributed Applications:** Explores potential extensions of the system into runtime QoS adaptation, discussing how reconfiguration triggers, resource predictions, and decentralized learning may be leveraged.
- **Section 5 – Conclusions:** Summarizes the document’s contributions and outlines next steps for integrating the presented mechanisms with the broader Swarmchestrator system.
- **Section 6 – References:** Lists the academic and technical sources used throughout the document.

2. QoS-Driven Application Deployment in Swarmchestrator

2.1 State of the Art

The deployment of applications across the Cloud-to-Edge continuum has evolved significantly, with recent research focusing on meeting Quality of Service (QoS) requirements such as low latency, high reliability, and efficient resource utilization. Various frameworks have emerged to address these challenges by orchestrating microservices and containers across heterogeneous and dynamic infrastructures. This section reviews prominent approaches and technologies that represent the current state of the art in QoS-aware application deployment.

Orive et al. (2022) [1] design a distributed Kubernetes scheduler that takes per-application QoS requirements as input. Their architecture distributes scheduling decisions across nodes so that complex, application-specific policies can scale to a high number of nodes. Mutichiro et al. (2021) [2] focus on edge-specific scheduling with STaSA, a *service time-aware* algorithm. STaSA automatically assigns IoT service requests to KubeEdge nodes under real-time constraints, significantly reducing deadline violations and improving end-to-end latency compared to default schedulers. Kim et al. (2023) [3] similarly evaluate KubeEdge (a Kubernetes-based edge platform) and observe that default cross-node forwarding hurts performance. They propose a *local scheduling* scheme that handles traffic entirely at the local edge node, which yields higher throughput and lower latency in multi-edge clusters.

Taherizadeh et al. (2018) [4] introduced “Capillary Computing”, an early approach to dynamically orchestrate microservices across edge, fog, and cloud layers. Their architecture integrates two core services: a monitoring system that spans all three tiers, and a capillary container orchestrator that migrates Docker-based microservices based on workload dynamics and proximity to mobile users. Tested in a mobile scenario with a vehicle-mounted edge node, this system significantly reduced response time and latency variability compared to a fixed-cloud baseline, demonstrating effective QoS maintenance under mobility and load fluctuations.

Building on the need for robust orchestration frameworks, MiCADO-Edge [5] extends the MiCADO cloud orchestrator [6] by incorporating KubeEdge to support Cloud-to-Edge microservice management. MiCADO-Edge automates the deployment and runtime governance of microservices using user-defined QoS policies, while supporting heterogeneous environments including containers, virtual machines, and multi-cloud platforms (AWS, Azure, OpenStack). It is a TOSCA-based, vendor-neutral specification that facilitates flexible service descriptions, with real-world use cases demonstrating its effectiveness in video processing and secure healthcare data workflows.

Another key contribution is ACOA [1], a distributed scheduling framework designed for fine-grained, QoS-aware orchestration in the Cloud-to-Edge continuum. ACOA tackles challenges such as scalability, dynamic behavior, and application-specific constraints by supporting customizable scheduling algorithms and modeling both infrastructure and workloads. Its decentralized, multi-scheduler architecture—built on Kubernetes—enables application-specific optimization of latency, reliability, and resource usage without rigid infrastructure partitioning. A real-world validation in the railway sector underlines its practical viability for complex deployment scenarios.

Focusing on runtime adaptability, Nautilus [7] introduces a QoS-aware system for managing microservice-based, latency-sensitive applications under constrained and dynamic conditions. Nautilus combines a communication-aware service mapper with reinforcement learning-driven resource management and adaptive microservice migration strategies. It intelligently partitions application graphs and responds to system pressure and I/O contention in real time, achieving significant reductions in both latency violations and resource consumption. This makes it a robust solution for maintaining performance under dynamic workloads.

From a broader perspective, Vaño et al. (2023) [8] provide a survey of cloud-native technologies adapted for edge computing. Their work covers container engines, edge-specific OSs, lightweight Kubernetes distributions (e.g., K3s), and edge orchestration frameworks like KubeEdge. They highlight emerging trends such as WebAssembly-based runtime environments and heterogeneous workload orchestration, framing the trajectory of cloud-native evolution towards more lightweight, scalable, and adaptable edge solutions.

Beyond cloud-native containers, novel orchestrators have emerged. Bartolomeo et al. (2022) introduce Oakestra[9], a hierarchical framework with a global “root” orchestrator and multiple cluster-level schedulers. Oakestra allows rich SLAs – including latency bounds and resource needs (vCPU/vGPU counts, bandwidth, etc.) – and delegates placements first to clusters, then within clusters, dramatically reducing scheduling complexity. This approach was demonstrated by deploying a latency-critical augmented-reality service across heterogeneous edge clusters. Likewise, Bisicchia *et al.* (2023)[10] present a continuous orchestrator prototype built on Docker, which integrates QoS management into CI/CD pipelines and monitoring tools. Their system continuously enforces QoS-aware placements of multi-service applications across geo-distributed cloud-edge resources. Similarly, Rac and Brorsson (2023) [11] propose a cost-aware edge scheduler: in a 5G vehicular use case, their Kubernetes-based scheduler periodically reevaluates and migrates services in response to user movement, lowering operational cost without violating latency requirements.

Finally, Swarmchestrator presents a decentralized, application-centric orchestration model inspired by swarm intelligence. Unlike centralized orchestrators, it relies on multiple autonomous agents that collaboratively discover resources, negotiate and monitor deployments, and dynamically reconfigure services across cloud, fog, and edge environments. This decentralized approach enhances scalability and fault tolerance while

balancing QoS objectives such as latency, resource utilization, and energy efficiency, making it a promising direction for resilient, self-adaptive deployment infrastructures.

Unlike prior work such as the distributed Kubernetes scheduler [1], Swarmchestrator introduces a network of Resource Agents (RAs) that perform decentralized matchmaking based on a novel “logical proximity” metric.

In our model, each RA evaluates its own capacity against the application’s QoS priorities (bandwidth, cost, energy) and computes a normalized, weighted cost; candidates are then globally ranked—either via additive/multiplicative reliability adjustments or through a Borda voting scheme—without any central scheduler, as discussed in Section 2.

As we demonstrate in Section 3, leadership among RLAs is maintained through a lightweight Raft consensus layer that is decoupled from state storage, ensuring fault-tolerant, continuous operation even under node failures.

This fully decentralized, multi-objective orchestration goes beyond hierarchical frameworks like Oakestra or reinforcement-learning systems such as Nautilus by combining swarm-inspired agent collaboration, rigorous multi-criteria ranking, and dynamic reconfiguration across cloud, fog, and edge domains—enabling scalable, resilient deployments with minimal manual intervention.

2.2 Introduction to QoS-based Automated Application Deployment from Cloud to Edge

Cloud computing can be viewed as a model for on-demand access to vast shared resources, effectively creating an environment that appears to have unlimited capacity for running applications and services [12][13]. By leveraging pay-as-you-go pricing and broad network accessibility, organizations can dynamically provision computing power, storage, databases, and related resources without the capital expenses associated with traditional on-premises data centers. This elasticity not only enables rapid scaling in response to fluctuating workload demands but also allows faster innovation, as teams can build, test, and deploy solutions more quickly. Additionally, the promise of economies of scale contributes to lower operating costs, while globally distributed cloud infrastructures help reduce latency and enhance user experiences¹. Given the benefits, cloud-only approaches may not strictly follow the requisites when it comes to latency, security, and privacy. This is true in the case of mobile devices, where network performance can be unpredictable. In addition to this, as end devices and systems gradually increase the volumes of data, transporting it to remote cloud facilities can become increasingly unattractive from a cost or resource perspective [14]. Adding to these concerns are risks such as unplanned provider outages, dependence on internet connectivity, and potential vendor lock-in, which highlight the need for careful planning when adopting cloud computing. Even though the Cloud Service Providers (CSP) highlight a pay-as-you-go model in most cases, the customers need to commit for a certain period, which may not be cost-effective. As a result, organizations must diligently choose the right CSP and implement governance mechanisms and contingency strategies when relying on cloud-based architectures [15].

To address scenarios in which the inherent constraints of centralized cloud infrastructures, such as high latency or heavy data-transfer overhead, make it impractical to send all information to distant data centers, the paradigm of fog computing has emerged. In essence, fog represents a decentralized layer that lies between the cloud and end devices, placing processing and storage in closer proximity to data sources while still retaining access to scalable cloud resources [12][16]. This approach is particularly advantageous for latency-sensitive applications since fog nodes can offload tasks from the cloud, respond more quickly, and maintain a higher degree of location awareness. Moreover, fog can enhance data privacy by allowing personal or sensitive information to be aggregated and analyzed locally rather than continually transmitted to remote servers. Despite its advantages, fog computing introduces significant operational and architectural complexities. Dense fog deployments can result in very high numbers of geographically dispersed nodes, all of which must be managed (scaled, patched, and reconfigured) across diverse hardware and network environments, requiring immense effort. Additionally, distributing computation across large numbers of fog nodes can increase overall energy usage, making energy efficiency and consumption reduction essential design goals [17]. Moreover, system complexity grows with node count and geographic distribution: faults ranging from simple network latencies, message reordering, or loss to more severe conditions like network

¹ <https://aws.amazon.com/what-is-cloud-computing>. Last accessed: 2.6.2025

partitioning or Byzantine failures become more probable as the number of fog nodes increases [18]. Unlike centralized data centers, where physical access control can be arbitrarily strict and subject to audit, fog nodes operate in often hostile, unsupervised locations, making equivalent protections impractical and driving up maintenance costs [18]. Fog devices are also inherently vulnerable to traditional security attacks. As "small clouds" deployed close to end users but outside rigorous surveillance can be compromised via man-in-the-middle and other intrusion techniques, enabling eavesdropping or data hijacking [19]. Furthermore, regulatory and legal requirements, such as healthcare data residency mandates or GDPR's transparency provisions, pose challenges in fog environments: virtualization can obscure the physical location of sensitive data, complicating compliance when applications do not track where the data is located.

Edge computing shifts data processing away from centralized data centers to the network's edge—usually just one hop from IoT devices—so that computation happens closer to where data is generated [12]. It enables offloading of compute tasks, local data storage, and caching and can distribute requests and responses between cloud services and end users, thereby cutting latency and reducing bandwidth use [20]. Edge computing delivers pronounced advantages in domains that demand stringent latency guarantees, such as autonomous systems, real-time data analytics, and immersive virtual or augmented reality, by processing workloads near the data source, thereby minimizing network-induced delays [12][20]. Edge computing, while offering substantial advantages like reduced latency and localized data processing, presents significant disadvantages that deserve careful consideration. One major drawback is its limited processing power and storage capacity compared to centralized cloud systems. Edge devices typically have constrained resources, making them less capable of handling complex computations or large volumes of data effectively [21][22]. Another critical disadvantage is the potential for security risks due to the deployment of many edge devices across various locations. Studies indicate that edge computing configurations may expose sensitive data to higher risks of breaches, considering the decentralized nature of data handling and transmission [23]. Additionally, the lack of redundancy at the edge means that in the case of device failure, data could be lost or services disrupted [24].

Along the spectrum of computing resources from a single on-premises data center to a vast network of connected endpoints, practitioners debate whether a centralized hyperscale cloud or a fully decentralized edge better meets today's IT challenges. In truth, each approach yields distinct advantages and disadvantages, and a balanced orchestration that leverages all components of the continuum is often the most effective. Cloud-native container orchestration framework Kubernetes (K8s) [25], has long enabled scalable and production-grade management in cloud data centers, abstracting machine-specific details and facilitating seamless hardware upgrades. Moreover, with the advent of lightweight K8s distributions such as K3s² and KubeEdge³, these orchestration capabilities have been extended to resource-constrained edge environments, thereby enabling cloud-native

² <https://k3s.io>. Last accessed: 2.06.2025

³ <https://kubedge.io>. Last accessed: 2.06.2025

workload management from centralized facilities to the very edge [8]. While automated deployment and runtime management in the cloud have reached maturity, robust, production-level solutions for orchestrating applications seamlessly across the Cloud-to-Edge continuum remain underexplored [5][1].

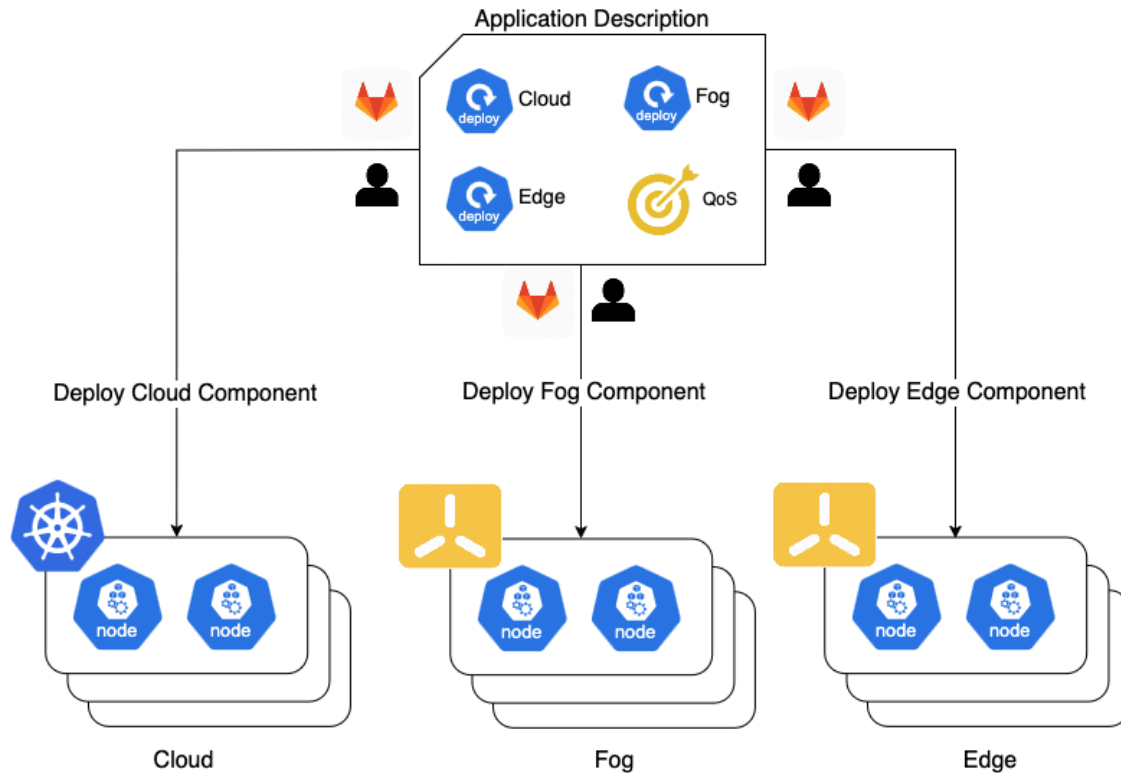


Figure 2.1: Traditional deployment approach

Figure 2.1 illustrates a scenario in which multiple Kubernetes clusters reside in the cloud and fog layers, with lightweight k3s clusters deployed at the fog and edge. Although Continuous Delivery (CD) tools like GitLab CI/CD⁴, GitHub Actions⁵, or Argo CD⁶ automate microservice deployment to specific clusters, cluster selection must currently be performed manually by a human administrator whenever QoS-based deployment is involved. Moreover, not all nodes within a given cluster may satisfy the QoS goals of a particular application, such as energy efficiency or cost. By default, the Kubernetes scheduler does not implement user-defined QoS-aware placement policies when selecting a node for an application. As a result, administrators are often required to explicitly specify the placement nodes in the manifest files and manually reconfigure both the CD pipeline and microservice configurations to ensure the required interconnections. The same level of manual intervention is also necessary during disaster recovery scenarios (e.g., cluster failure) to initiate re-deployment in a suitable alternative cluster that meets the application's QoS needs.

⁴ <https://docs.gitlab.com/ci>. Last accessed: 31.05.2025

⁵ <https://github.com/features/actions>. Last accessed: 31.05.2025

⁶ <https://argoproj.github.io/cd>. Last accessed: 31.05.2025

Our goal is to eliminate these manual interventions by automatically selecting suitable clusters based on QoS metrics and seamlessly redeploying microservices when disruptions arise, thereby creating a fully automated solution for Cloud-to-Edge orchestration. In our work, we focus on the QoS attributes of energy efficiency, cost, and performance (specifically CPU, memory, storage, and bandwidth).

2.3 Logical Proximity

During the Swarmchestrator application deployment process, upon receiving application descriptions submitted by the operators, expressed in the TOSCA format, the Swarmchestrator orchestrator, through the *Resource Agents (RAs)*, manages computational resources from various cloud and edge providers (e.g., Amazon, Microsoft), forming *logical groupings of resources* within the Swarmchestrator ecosystem, that are registered for discovery and deployment (see more details in D2.1). The *Knowledge Management* component, as elaborated on deliverable D3.1, acts as a distributed knowledge base, managing resource descriptions, interactions, discovery, and trust.

Specifically, Swarmchestrator supports microservices-based applications, described in TOSCA, covering four key aspects: (a) The details of application components such as container images, environment variables, etc; (b) The specific needs for application resources, such as cloud/edge instances, instance types, and hardware limits (CPU/RAM/Storage); (c) The desired QoS specifications, including performance, cost, energy efficiency, trust, placement, etc; and (d) the specification of custom metrics to be monitored by Swarmchestrator.

On the other hand, the RA manages computational resources, providing access to their resources. In Swarmchestrator, an RA is instantiated, when the resource provider registers their resources with attributes such as processing power, memory, hardware type, VM instances, pricing, locality, and energy metrics. Once instantiated, the RA connects to other RAs via a P2P network, forming a decentralised OS interface. The TOSCA description is submitted to the interface, where an RA receives it and initiates the deployment process with collaboration with other RAs.

At the deployment time, a random, lead RA will be selected to identify the resources that will satisfy the submitted microservices. The selected RA broadcasts the resource requirements to all available RAs, requesting resource offers. Each RA evaluates its capacity via Knowledge Management and classifies its coverage as either Partial (some requirements met), Full (all met), or Zero (none met), forming unique groups of potential resource offers. Specifically, logical proximity in the context refers to the closeness of the application's QoS requirements to the resources' capacities provided by each candidate resource offers. The resource offer that best matches the QoS requirements, i.e., with the highest *logical proximity*, will be selected. This optimal set, once configured, forms the Swarm.

Here, it is important to note that logical proximity abstracts away from geographical or network-hop distance—what we call physical proximity—and instead measures semantic fit between an application's QoS and a node's capabilities. This lets us rank and choose nodes that may be physically farther away but offer a closer match on, e.g., cost, reliability, or security policies.

The logical proximity estimation takes into account: the *application* QoS *goals* from the TOSCA description, and the dynamically obtained *reliability* metrics (e.g. failure frequency, availability, resource accuracy, etc.), representing the impact on the achievement of QoS goals, for each resource offer. Specifically, concerning the QoS goals, we primarily account

for four attributes including *latency*, *cost*, *bandwidth*, and *energy consumption*. Each of these attributes is defined with a *priority* reflecting the application operator's preferences.

Formally, we construct a cost function that computes a value of logical proximity for each resource offer. The offer that incurs the least cost will be chosen to form the Swarm. Let r^q be the raw data reflecting each QoS attribute q belonging to a set of QoS attributes Q , i.e., $q \in Q$. To account for QoS attributes lying in different scales of metric space, we normalise their cost value nor_q into the range of $[0, 1]$ as in Equation 1. Each normalised value is then weighted by its QoS priority p^q , and the total cost for an offer i is calculated as follow:

$$nor_q = \begin{cases} 0, & \text{if } \max(r_q) = \min(r_q) \\ \frac{r_q - \min(r_q)}{\max(r_q) - \min(r_q)}, & \text{otherwise} \end{cases} \quad (1)$$

$$total_cost_i = \sum_{q \in Q} p_q \cdot nor_{q,i} \quad (2)$$

Lastly, to incorporate reliability (R) into ranking, two approaches are used: (a) Additive, where R is subtracted from the total cost ($total_cost_i - R_i$), lowering costs for more reliable offers; and (b) Multiplicative, where R scales the total cost ($total_cost_i = (1 - R_i) \cdot total_cost_i$), adjusting cost proportionally to reliability.

In addition to considering the continuous cost function, we also propose an alternative with Borda Voting. This approach ranks offers based on their relative positions across QoS attributes. Each attribute is ranked independently (e.g., bandwidth in descending order, latency in ascending order), and offers receive scores based on their rank, with ties sharing the highest score for their position. Scores are then weighted by attribute priorities to determine the final ranking. Finally, reliability—either using an additive or multiplicative approach—is incorporated into the ranking process.

2.4 Simulating the Use of Logical Proximity

To evaluate and demonstrate the applicability of the proposed cost-based ranking and Borda voting algorithms, we conducted a simulation-based evaluation designed to provide realistic inputs for ranking. For this purpose, we used the DISSECT-CF-Fog simulation environment, which was selected by the project consortium during the initial planning phase. This tool plays a central role in Swarmchestrator, as it provides the first testbed to measure the proposed decentralised swarm formation and orchestration processes.

The simulator extension towards modelling resource agents, swarm agents and applications, and its integration with the aforementioned ranking algorithms are described in deliverable 4.2 in more detail, therefore, here we provide only a brief summary of the work achieved.

Thanks to the broad usability and capabilities of the DISSECT-CF-Fog simulator, we first configured the cloud, fog, and edge resources and their constraints (e.g., CPU, memory). These resources are represented by resource agents connected via a P2P network. Currently, the simulator focuses on the deployment phase of the submitted applications. When an application is received by a resource agent (referred to as the gateway RA), it initiates a broadcasting algorithm to collect offers from all resource agents. This process and the associated communication are also modeled through the simulator's network layer.

An example of a response containing offers provided by a resource agent is shown in Figure 2.2. For App-1, the corresponding resource agent can cover either *Component-1* and *Component-2*, or only *Component-3*, of the given application (see lines 9-10). The response also includes the energy, bandwidth, latency, and price properties (see lines 4-7), which describe the characteristics of the capacities offered by this particular resource agent.

```

1 {
2   "ResourceAgent": "Agent-1",
3   "Application": "App-1",
4   "Energy": "120",
5   "Bandwidth": "25000",
6   "Latency": "55",
7   "Price": "0.000000278",
8   "Offers": [
9     ["Component-1", "Component-2"],
10    ["Component-3"]
11  ]
12 }
```

Figure 2.2: An example offer provided by a resource agent

Once each response from the resource agents, providing the coverage of application components, is received, the gateway RA generates all possible combinations of these offers.

In each combination, every application component is covered exactly once, thus, a combination is a collection of resource agent and application component pairs. The simulator then iterates through each combination and creates the input for the ranking script, involving the previously mentioned capacity characteristics. Various aggregation strategies can be applied to the values, such as taking the mean, maximum, or sum. An example input and the result of the aggregation are shown in Figure 2.3.

It is worth mentioning that, at present, the reliability values are set to 1.0 and are not included in the ranking process due to the ongoing development of the Knowledge Management subsystem.

```

1 {
2   "qos_priority" : {
3     "energy" : 1.0,
4     "bandwidth" : 0.0,
5     "latency" : 0.0,
6     "price" : 0.0
7   },
8   "reliability" : [ 1.0,      1.0,      1.0,      1.0,      1.0      ],
9   "energy" :      [ 220.4,    286.5,    234.6,    269.8,    196.2    ],
10  "bandwidth" :    [ 12500.0,  25000.0,  62500.0,  125000.0,  6250.0    ],
11  "latency" :      [ 100.0,    50.0,    20.0,    30.0,    80.0      ],
12  "price" :        [ 5.556E-5, 5.56E-6, 2.7778E-4, 7.5556E-4, 1.1112E-4 ]
13 }

```

Figure 2.3: An example output of generated for ranking

Finally, the ranking script returns the suggested combination of offers, allowing the DISSECT-CF-Fog simulator to pick the provided combination for the deployment and swarm formation and proceed with modeling the runtime phase of the submitted application. The simulation results have also been summarised in a conference paper, which has been accepted for publication at ICFEC 2025 [26]. A more detailed description of the overall system for the simulation and the results can be found in deliverable report D4.2 since this is a joint work between WP1 and WP4.

3. Demonstrating Intelligent Cloud-to-Edge Deployment

This section serves as an exercise to test the feasibility of Swarmchestrator's objectives regarding intelligent orchestration across the Cloud-to-Edge continuum. Although the interaction between RAs and the impact of logical proximity were demonstrated in Section 2 and elaborated in deliverable D4.2, implementing the system to test the feasibility of our end goal is necessary to show that our proposal fulfills both the functional and non-functional requirements of an orchestrator. Thus, by demonstrating how key architectural components—such as the Resource Lead Agent, Resource Agent, and Knowledge Base—enable automated, QoS-driven application deployment, this work operationalizes the project's vision of scalable and fault-tolerant orchestration. The results presented here provide a critical foundation for further runtime adaptation and optimization tasks addressed in subsequent phases of the project.

The concept introduces a test bed for the distributed resource orchestration until the actual Swarmchestrator system is constructed. It must be noted that the main entities (such as RA, RLA, KB, etc.) implemented in this concept and its functionalities may vary in the other deliverables since the concept introduced in this report follows the architectural suggestions in deliverable D2.1. Here in D1.1, we demonstrate an alternative solution that stems from D2.1, but the functionalities we provide in this subsection for the main entities are abstracted. For example, KB has its own working package in Swarmchestrator, and is more complex than how we implemented in D1.1. All terms, acronyms, and definitions mentioned in this report should be assessed in the context of D1.1.

3.1 Background

3.1.1 Kubernetes

The following discussion is based on Kubernetes Documentation⁷, Saad Ali's "Kubernetes Design Principles" talk⁸, and "The Kubernetes Book" by Nigel Poulton [27].

Architecture. Kubernetes is an open-source platform for orchestrating containerized applications. A K8s cluster comprises two main parts: the control plane (or master node(s)) and the worker node(s) (which may be physical or virtual machines) as shown in Figure 3.1. The control plane is responsible for maintaining the desired state of the cluster and orchestrating workloads, while worker nodes execute the containerized applications packaged into Pods⁹.

The control plane includes several components:

⁷ <https://kubernetes.io/doc>. Last accessed: 3.6.2025

⁸ <https://kccna18.sched.com/event/IrkE>. Last accessed: 3.6.2025

⁹ <https://kubernetes.io/docs/concepts/workloads/pods>. Last accessed: 3.6.2025

- **API Server:** The central communication hub where all components interact using declarative REST APIs.
- **etcd:** A highly available, Raft-based distributed key-value store for cluster state.
- **Scheduler:** Responsible for assigning newly created Pods to suitable worker nodes based on resource availability, health, and other policies.
- **Controller Manager:** Runs various controllers (e.g., Node¹⁰, Deployment¹¹, and Job¹² controllers) that continuously monitor and reconcile the cluster state.
- **Cloud Controller Manager:** Hosts controllers that integrate with cloud provider services such as external load balancers. It separates cloud provider-specific tasks from core cluster logic. This component is usually absent in on-premises or local test clusters.

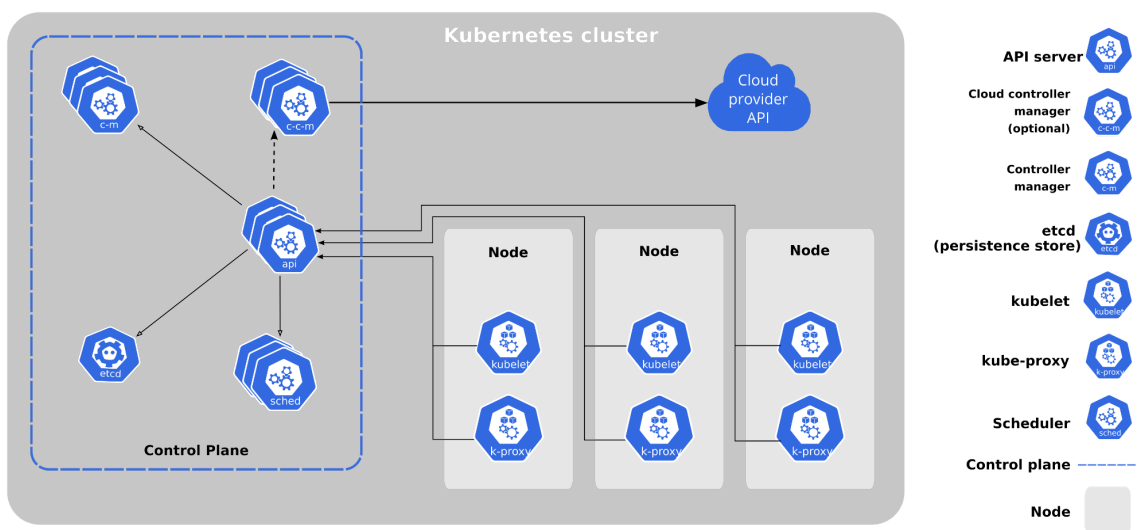


Figure 3.1: A Kubernetes cluster and its components

Worker nodes run essential components such as the kubelet (which manages Pod execution and reports node health), kube-proxy (for local cluster networking), and container runtime (which handles container lifecycle tasks).

Core Concepts: Pods, ReplicaSets, and Deployments. Kubernetes deploys and manages applications by grouping one or more related containers into a single entity known as a Pod. A Pod is the smallest deployable unit in K8s and serves as the atomic element for scheduling and horizontal scaling. Each Pod is augmented with metadata (e.g., names, labels¹³ and annotations¹⁴) that higher-level objects utilize for management.

Because Pods are ephemeral and do not provide built-in mechanisms for scaling or self-healing, Kubernetes employs ReplicaSets¹⁵ to maintain a specified number of Pod

¹⁰ <https://kubernetes.io/docs/concepts/architecture/nodes>. Last accessed: 3.6.2025

¹¹ <https://kubernetes.io/docs/concepts/workloads/controllers/deployment>. Last accessed: 3.6.2025

¹² <https://kubernetes.io/docs/concepts/workloads/controllers/job>. Last accessed: 3.6.2025

¹³ <https://kubernetes.io/docs/concepts/overview/working-with-objects/labels>. Last accessed: 3.6.2025

¹⁴ <https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations>. Last accessed: 3.6.2025

¹⁵ <https://kubernetes.io/docs/concepts/workloads/controllers/replicaset>. Last accessed: 3.6.2025

replicas. Building on ReplicaSets, Deployments enable seamless rollouts and rollbacks, ensuring zero downtime during updates, as shown in Figure 3.2.

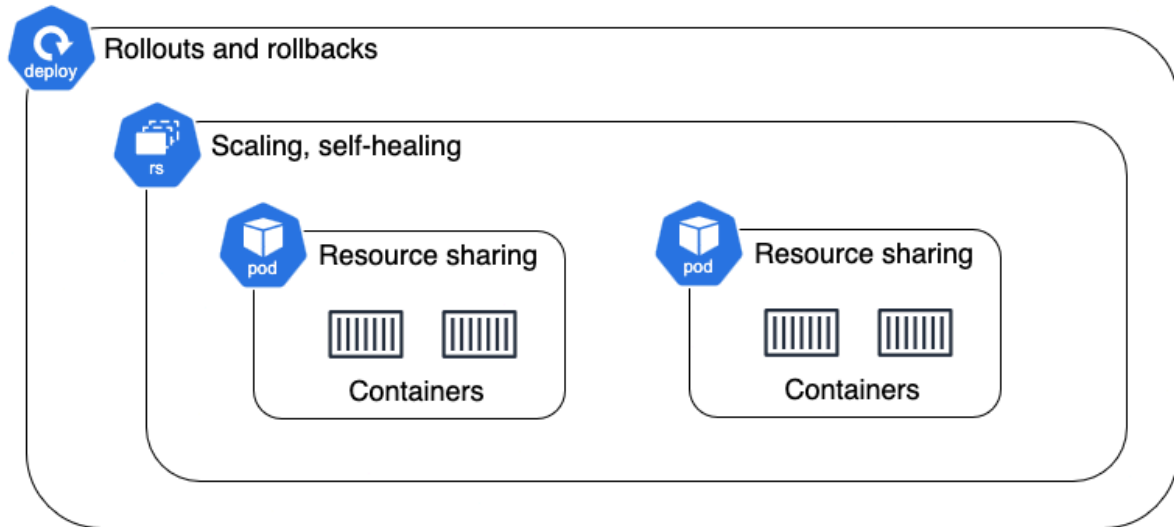


Figure 3.2: Kubernetes Pod management [27].

Networking and Service Discovery. Due to the transient nature of Pods, using their IP addresses for service-to-service communication is not advisable. Kubernetes addresses this by introducing the Service¹⁶ object, which provides stable networking to a set of Pods through a fixed IP address, DNS name, and port, as shown in Figure 3.3. In addition to offering consistent connectivity, Services also load balance incoming requests among the Pods.

Kubernetes supports multiple types of Services:

- **ClusterIP:** Provides a virtual IP address accessible only within the cluster.
- **LoadBalancer:** By default, a Service is ClusterIP, which is not accessible from outside the cluster. LoadBalancer exposes the service externally using a cloud provider's load balancer or, for bare metal, tools like MetalLB¹⁷.

Since allocating a separate LoadBalancer for each external application can be very expensive, Kubernetes also offers Ingress¹⁸. Ingress enables multiple external-facing applications to share a single load balancer by defining traffic routing rules via an Ingress object and implementing those rules through an Ingress controller as shown in Figure 3.4. While Kubernetes does not provide an Ingress controller by default, several open-source solutions (e.g., Ingress NGINX Controller¹⁹) are available.

¹⁶ <https://kubernetes.io/docs/concepts/services-networking/service>. Last accessed: 3.6.2025

¹⁷ <https://metallb.io>. Last accessed: 3.6.2025

¹⁸ <https://kubernetes.io/docs/concepts/services-networking/ingress>. Last accessed: 2.6.2025

¹⁹ <https://github.com/kubernetes/ingress-nginx>. Last accessed: 2.6.2025

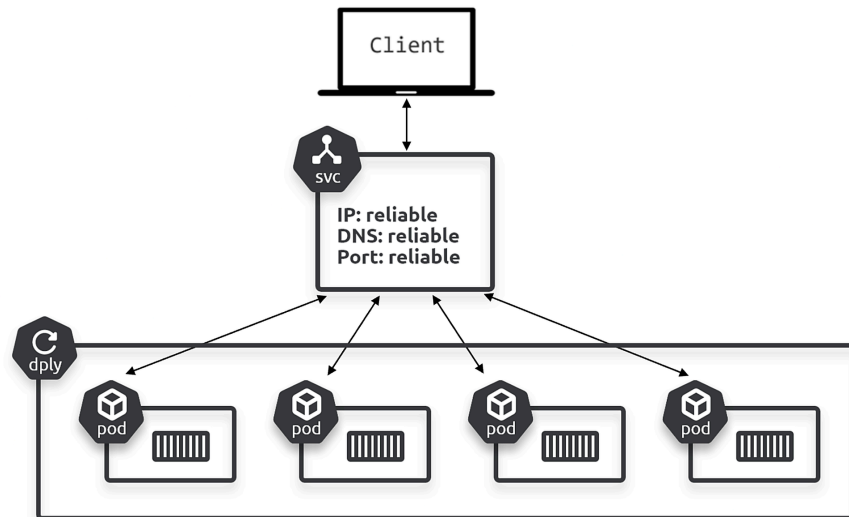


Figure 3.3: Kubernetes networking through Service [27].

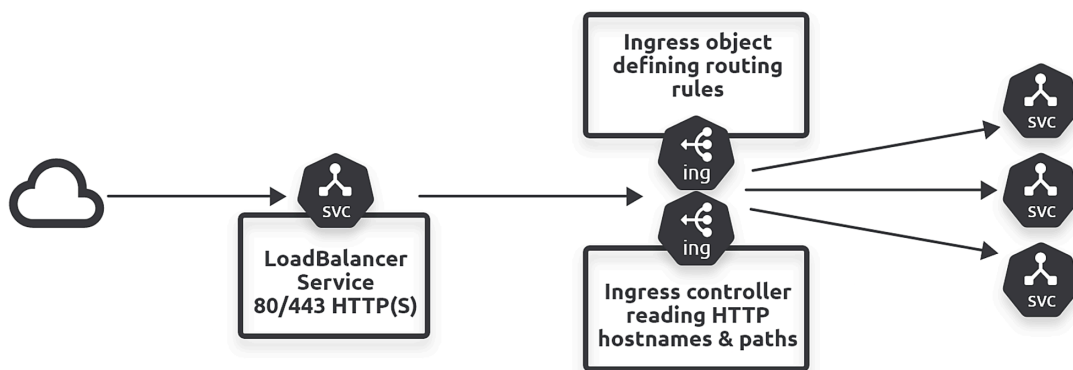


Figure 3.4: Kubernetes Ingress [27].

Storage, Configuration, and Security. For applications that require persistent storage, Kubernetes provides Persistent Volumes²⁰ (PV) and Persistent Volume Claims (PVC). A PV represents an external storage resource, while a PVC is used by a Pod to claim access. Kubernetes can dynamically provision a PV to satisfy a PVC or bind to an existing volume.

In addition to storage, Kubernetes manages application configuration and sensitive data through ConfigMaps²¹ and Secrets²². Resources can be grouped and isolated using Namespaces²³, which also facilitates the application of policies and quotas.

²⁰ <https://kubernetes.io/docs/concepts/storage/persistent-volumes>. Last accessed: 3.6.2025

²¹ <https://kubernetes.io/docs/concepts/configuration/configmap>. Last accessed: 3.6.2025

²² <https://kubernetes.io/docs/concepts/configuration/secret>. Last accessed: 3.6.2025

²³ <https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces>. Last accessed: 3.6.2025

Security is further enhanced by Kubernetes' Role-Based Access Control²⁴ (RBAC). RBAC defines permissions through Roles/ClusterRoles. This mechanism applies to both human users and non-human entities (e.g., ServiceAccounts²⁵), ensuring that Pods and other resources have the appropriate access to the API server.

A Typical Kubernetes Workflow. When deploying an application in Kubernetes, the process begins with a user (or an automated system) submitting a YAML manifest describing the desired state of the application with a variety of resources (e.g., Pods, Deployments, Services) and is sent to the Kubernetes API Server. The API Server validates the request, persists the resource specifications into etcd, and makes these changes visible to the rest of the cluster.

Kubernetes Scheduler and controllers constantly watch the API Server for new or updated resources. When a new resource (e.g., a Pod in a Deployment) appears, the scheduler determines which node should run the Pod. The Scheduler selects an appropriate node based on constraints such as resource availability, affinity/anti-affinity rules, and taints/tolerations. Once the Scheduler updates the Pod's node assignment, the node's kubelet sees the update and proceeds to create or update the Pod with the necessary container(s).

Each worker node regularly sends heartbeats (or status updates) to the API Server. These heartbeats update the node's status and health conditions, indicating whether the node is still alive and capable of running workloads. If a node fails to send heartbeats for a configured duration, the control plane marks that node as unhealthy. Consequently, Kubernetes may evict or reschedule Pods that were running on the unhealthy node, maintaining the desired state. In addition to node-level heartbeats, the kubelet also reports Pod status back to the API Server. This status includes information about container health, readiness checks, and liveness probes. When a Pod fails its readiness or liveness probes repeatedly, Kubernetes can take corrective actions, such as restarting the failing containers or removing the Pod altogether (which then triggers the ReplicaSet or Deployment controller to spin up a replacement Pod elsewhere).

3.1.2 K3s

The following discussion is based on the K3s documentation²⁶, the K3s GitHub repository, and additional insights from recent scientific literature [8].

Architecture. K3s is a lightweight K8s distribution optimized for resource-constrained environments (e.g., Edge, IoT, etc.). In contrast to the traditional Kubernetes design, a K3s cluster comprises two main parts as shown in Figure 3.5:

- **Server node(s):** This component is equivalent to the K8s master node. It consists of the control plane components, datastore components, kubelet, container runtime (e.g., containerd), and CNI. (Figure 3.6)

²⁴ <https://kubernetes.io/docs/reference/access-authn-authz/rbac>. Last accessed: 3.6.2025

²⁵ <https://kubernetes.io/docs/concepts/security/service-accounts>. Last accessed: 3.6.2025

²⁶ <https://docs.k3s.io>. Last accessed: 3.6.2025

- **Agent node(s):** This component is equivalent to a K8s worker node. It runs the kubelet, container runtime, and CNI and does not have any control plane or datastore components. (Figure 3.6)

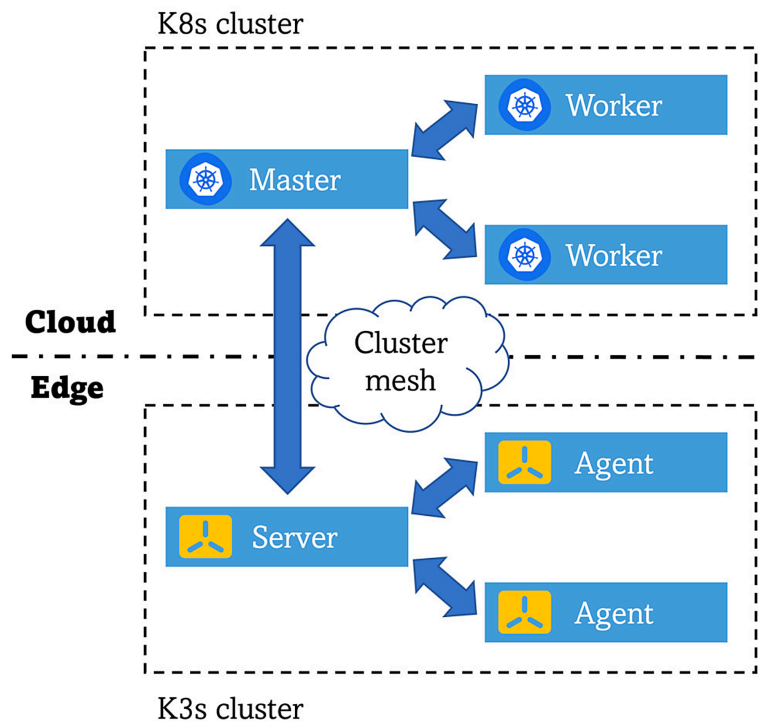


Figure 3.5: K8s and K3s architecture comparison [8].

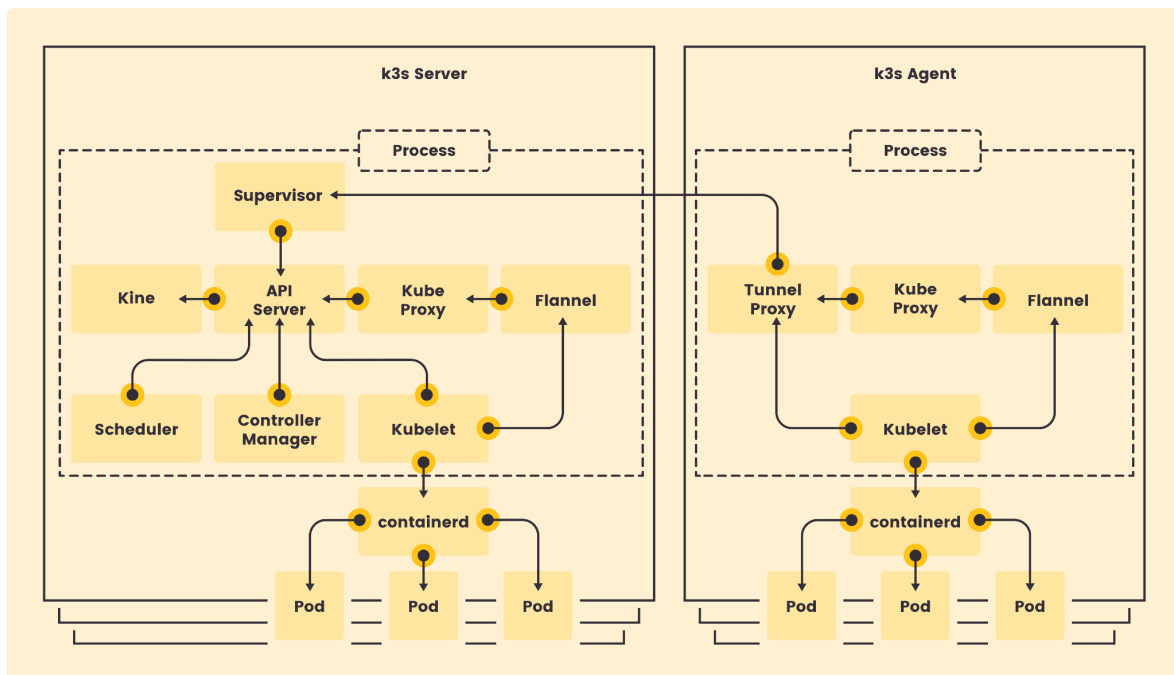


Figure 3.6: K3s architecture.

Differences from Upstream Kubernetes. K3s retains the core functionalities of upstream Kubernetes while incorporating modifications to reduce size and complexity:

- **Single Binary Distribution:** K3s packages most of the necessary components into a single binary of less than 100 MB.
- **Reduced Components:** In-tree storage drivers and cloud providers are removed. Out-of-tree alternatives (e.g., Container Storage Interface and Cloud Controller Manager) are used instead, further reducing the binary size.
- **Kine:** Instead of relying exclusively on etcd, K3s can use an embedded SQLite instance or an external database (e.g., MySQL or PostgreSQL) to store cluster state. This is accomplished through Kine, a small shim layer that translates between Kubernetes' etcd API and various SQL data stores. Kine allows for simpler, more flexible deployments, particularly useful in resource-constrained or edge environments, while preserving the Kubernetes control plane's expectations of an etcd-like key-value store.
- **Enhanced Security by Default:** TLS certificates are automatically managed, and agent nodes communicate with the control plane over a secure WebSocket tunnel.
- **Networking and Service Discovery:** K3s provides standard Kubernetes networking and service discovery features with a reduced footprint by bundling essential components:
 - **Flannel:** Implements the Container Network Interface (CNI) for pod networking.
 - **CoreDNS:** Offers DNS-based service discovery within the cluster.
 - **Traefik:** Serves as the default ingress controller.
 - **Klipper-lb:** Functions as an embedded load balancer for services requiring external exposure.

K3s is a distribution rather than a fork, maintaining close alignment with upstream Kubernetes through minimal patching and contributing back to open-source projects whenever possible.

3.1.3 Raft

Raft [28] is a consensus algorithm designed to manage a replicated log in a distributed system and ensure fault-tolerant operation. It follows the replicated state machine (RSM) approach (see Figure 3.7), where client commands are appended to a log, replicated across a cluster of nodes, and applied in the same order on each node's local state machine. This guarantees that under normal operation, all non-failing nodes maintain identical logs and execute the same sequence of commands, ensuring a consistent replicated state. Raft is both practical and easy to understand, enabling developers to build correct, production-ready systems.

A Raft cluster typically comprises an odd number of nodes (such as 3 or 5), which simplifies majority-based decision-making (majority-based quorum), reduces the likelihood of split votes, and allows the system to tolerate up to $\frac{n}{2}$ node failures. For clarity, Raft decomposes consensus into three core components: Leader election, log replication, and safety.

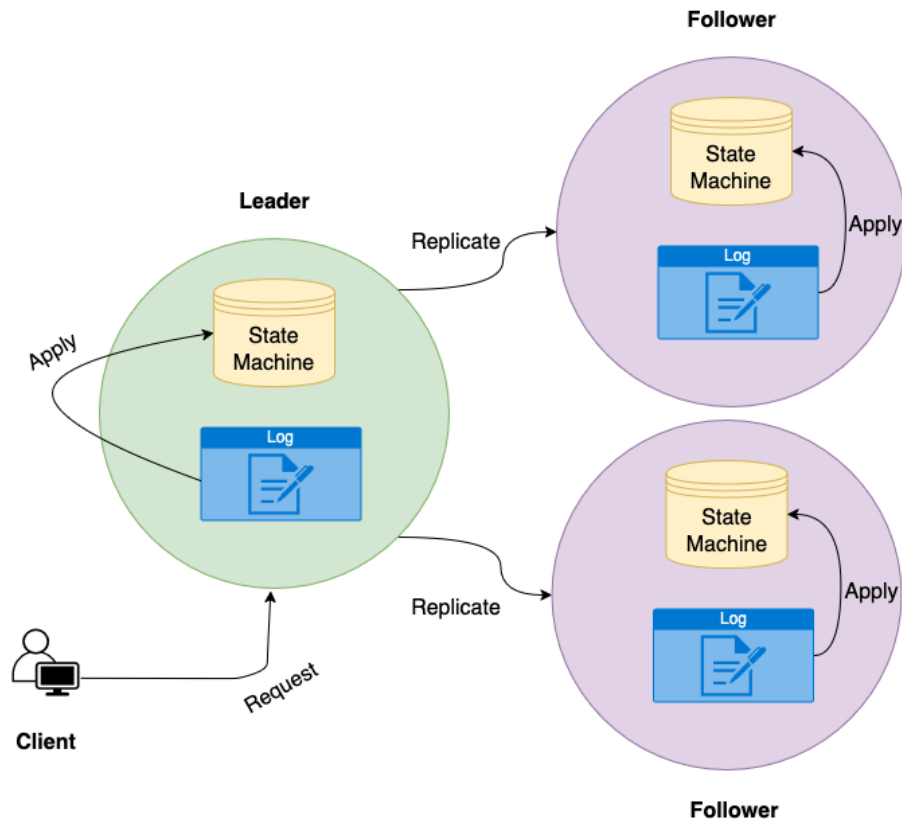


Figure 3.7: Replicated State Machine

At any point, a Raft node is in exactly one of three states: Follower, Candidate, or Leader, as shown in Figure 3.8. Raft divides time into terms-logical epochs that begin with a potential leader election and increment whenever a node starts an election. Nodes track the current term number, and the election rules ensure that no two leaders are elected in the same term.

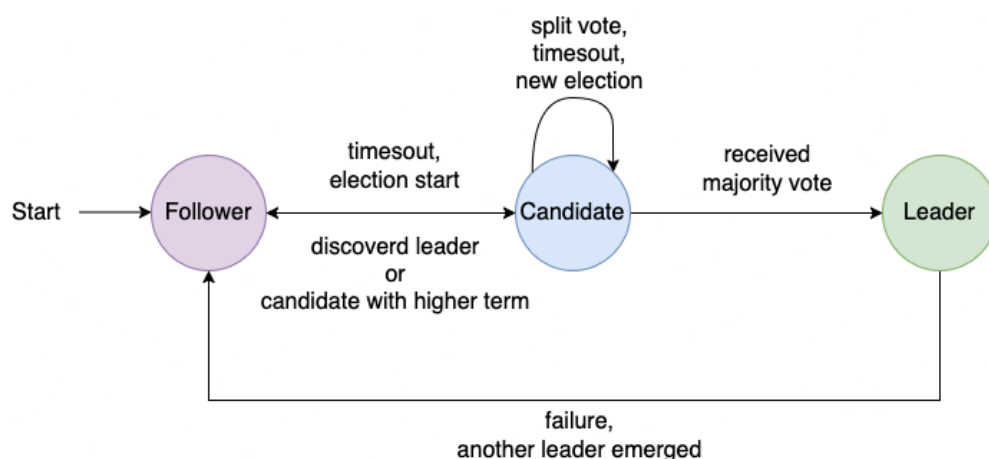


Figure 3.8: Node states and the transitions among them

Leader Election. When a Raft server starts, it begins as a follower and waits for messages from a leader or candidate. The current leader periodically issues heartbeat messages using the AppendEntries RPC without any log entries to confirm its authority. If a follower does

not receive such communication within a specified election timeout, it assumes no viable leader exists and transitions to the candidate state.

Once a node becomes a candidate, it increments its current term, votes for itself, and broadcasts RequestVote RPCs to all other servers in the cluster. To secure leadership, the candidate must collect votes from a majority of the servers. If successful, it becomes the new leader and immediately sends heartbeat messages to affirm its role and prevent additional elections.

There are three possible outcomes of a Raft election:

- **Victory:** The candidate gains votes from a majority of servers in the current term and becomes the leader.
- **Another leader emerges:** If the candidate receives an AppendEntries RPC with a term at least as large as its own, it recognizes that server as the leader and reverts to a follower.
- **No clear winner:** If several servers become candidates simultaneously, votes may split so that no candidate obtains a majority. In this case, each candidate waits for its (randomized) election timeout to expire, increments its term, and begins a new round of RequestVote RPCs.

To mitigate repeated split votes, Raft randomizes the election timeout for each server (e.g., uniformly within 150-300 ms), reducing the likelihood of simultaneous transitions to the candidate state.

Log Replication. After being elected as a leader, it begins processing client requests. If a follower receives a client request, it forwards the request to the leader. Each client request contains a command intended for the replicated state machines. The leader appends this command as a new entry in its write-ahead log (WAL) and sends AppendEntries RPCs in parallel to all followers to replicate the new entry. After the majority of the followers inform that the entry is written in their WAL, the leader commits the entry in its state machine and returns the result to the client. The leader also notifies followers that the entry has been committed so they can apply it to their respective state machines. This majority-based commit rule ensures durability and eventual execution by all non-failing servers. Each log entry contains an integer-based index and both the client command and the term number under which the leader added the entry to maintain consistency. Although normal operation maintains log consistency, leader crashes can leave some followers with extra or missing entries. A key property of Raft is that a leader never overwrites or deletes its own log entries. Conflicts are resolved solely by adjusting followers' logs to match the leader's. Once a follower's log is brought into agreement with the leader's, it remains consistent with that leader for the duration of the term.

Safety. While Raft's leader election and log replication mechanisms ensure that commands flow correctly under most circumstances, additional measures are necessary to guarantee that every state machine applies the same commands in the same order. A primary concern arises if a follower becomes temporarily unreachable while the leader commits new log entries and later returns, potentially becoming the leader without possessing all committed

entries. In such a scenario, the follower might overwrite missing log entries with new commands, leading to divergent command sequences across servers. Raft addresses this issue by enforcing a Leader Completeness Property: any server elected as leader in a new term must contain all entries committed in previous terms. This is achieved through an election restriction and a refined definition of log entry commitment. During elections, servers only vote for candidates whose logs are at least as up-to-date as their own. A candidate's RequestVote RPC includes information about its last log entry (term and index). If a voter's log is more up-to-date, it denies its vote. This ensures that only a server holding all previously committed entries can gather a majority vote and become the leader.

A leader considers an entry from its current term committed as soon as it is stored on a majority of servers. However, older log entries (from previous terms) are not committed merely by appearing on a majority of servers. Instead, Raft commits these older entries indirectly: once the leader commits a new entry from its current term, all preceding entries in the log are considered committed. This approach avoids corner cases where a majority of servers might appear to store a prior entry without guaranteeing its retention if a leader crashes and re-election occurs. Combining the above rules ensures that if a leader in a term T has committed a log entry, any future leader in that term $U > T$ will also contain that entry. This is because the majority of servers that accepted the entry will not grant votes to a candidate lacking it. The Leader Completeness Property implies that once an entry is committed, it cannot be lost. Suppose, by contradiction, that a leader in term T commits an entry, but a later leader in term U lacks that entry. By definition, the new leader must have obtained votes from a majority of servers, including at least one that had stored the committed entry. However, that server's log would have been at least as up-to-date as the candidate's, making it impossible for the new leader to lack the entry. This contradiction proves that committed entries always appear in every subsequent leader's log.

Finally, Raft's State Machine Safety property guarantees that once any server applies for a log entry at a specific index, no other server will apply for a different entry at that index. By enforcing the sequential application of log entries (in ascending index order) and ensuring that committed entries at the same index are identical across logs, Raft guarantees that all non-failing servers execute the same sequence of commands.

3.1.4 Clean Architecture

To ensure our Cloud-Edge orchestrator test bed is both robust and adaptable, its design is fundamentally based on the “Clean Architecture” pattern. We selected this architectural model because its emphasis on the separation of concerns is critical for a system that must manage components across the heterogeneous and evolving Cloud-to-Edge continuum. This subsection provides a detailed overview of the principles of Clean Architecture to offer the necessary context for the design decisions and structural organization of our orchestrator, which are detailed in the subsequent sections of this report.

Clean Architecture²⁷, popularized by Robert C. Martin (often referred to as Uncle Bob), aims to achieve a systematic separation of business logic from implementation details. By structuring an application in concentric layers—each with a specific responsibility—it becomes easier to maintain, extend, and test. The architecture's hallmark is that key business rules remain shielded from external technical concerns, fostering high cohesion around core functionality while minimizing coupling to outside frameworks or technologies.

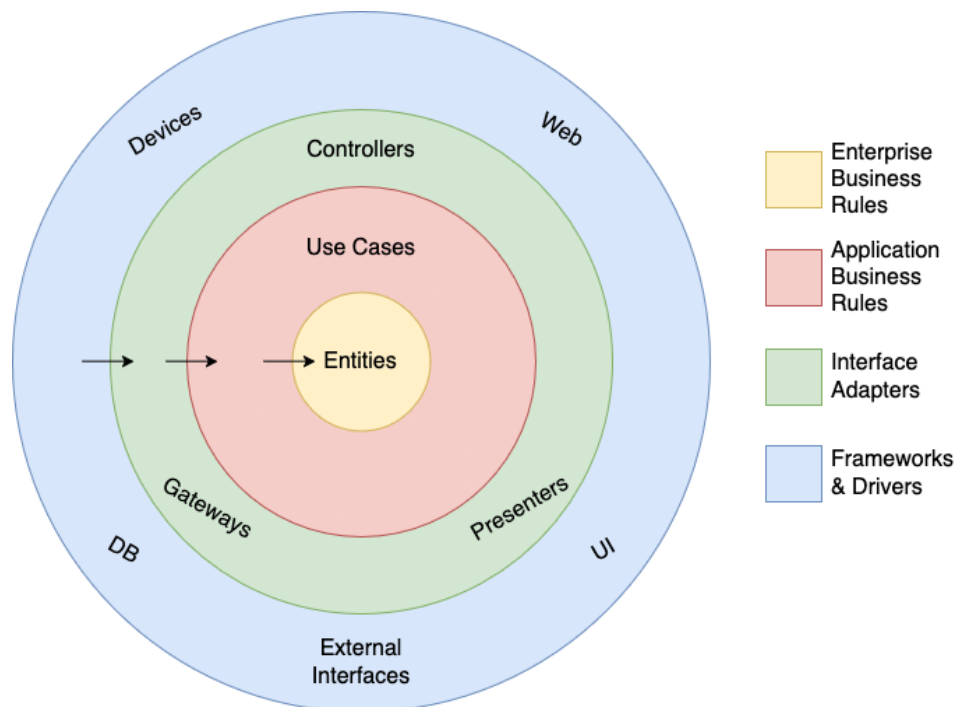


Figure 3.9: Clean architecture

Figure 3.9 illustrates a layered approach designed to isolate high-level rules from technical details. Moving inward toward the center reflects an increasing level of abstraction, while the outer rings deal with concrete implementations. The key principle, known as the Dependency Rule, stipulates that all source code dependencies must point inward. Consequently, no inner component should reference names or structures from an outer layer.

At the core are Entities, which contain fundamental business logic. They are meant to remain stable even if external concerns, such as user interfaces or security mechanisms, change.

Surrounding them are Use Cases, where application-specific business rules reside. This layer directs the flow of data to and from the inner entities, orchestrating the steps required to fulfill each scenario in the system.

The next level, Interface Adapters, adapts data to formats that internal or external elements can handle. In this layer, one might find presenters, views, and controllers for user interfaces, as well as components that manage the interaction with storage or external

²⁷ <https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html>. Last accessed: 3.6.2025

services. By restricting these adaptations to the outer ring, the innermost rules remain unaffected by infrastructure or framework concerns.

Finally, Frameworks and Drivers occupy the outermost circle, where tools such as databases or web servers are located. Minimal code is placed here, serving primarily as a conduit to the adjacent inner layer. This separation helps ensure that whenever external technologies become obsolete or require replacement, core functionality remains intact and can be tested without those external dependencies.

Adhering to this layered design results in systems that are:

- **Independent of Frameworks:** The architecture is independent of a certain library or platform. Frameworks become optional tools rather than constraints that dictate the system's structure.
- **Testable:** Core business rules are testable without UI, database or other external components.
- **Independent of UI:** User interfaces can be changed or replaced—e.g., switching from a web UI to a console UI—without requiring alterations to the core business rules.
- **Independent of Database:** Databases can be changed (e.g., from SQL to NoSQL) with minimal effort, and core business logic is not tied to any specific storage technology.
- **Independent of Any External Agency:** The business logic is unaware of external services; thus, it remains unaffected by changes in external services.

By ensuring that data passed across these boundaries is kept in simple structures free from any framework-specific details, the architecture preserves the independence of the inner layers and mitigates the impact of changes in external tools or interfaces.

3.2 Functional and Non-functional Requirements

To systematically guide the development of our Cloud-to-Edge orchestrator, it is crucial to first establish a clear set of capabilities and operational goals. This chapter defines these essential criteria by presenting the core functional and non-functional requirements that the system must fulfill. These requirements serve as the foundational blueprint that dictates the orchestrator's architecture and provides the benchmarks for its subsequent evaluation, ensuring it can effectively manage applications across the continuum as intended.

3.2.1 User-guided Domain Deployment (FR1)

The system shall deploy each application component to the domain cluster (cloud, fog, or edge) as specified by the user. This capability must include:

- **User Configuration:** Allowing users to designate which domain(s) particular components should be placed in based on their operational and QoS requirements.
- **Seamless Provisioning:** Handling the actual scheduling and instantiation of components in the user-selected domain swiftly and transparently.

3.2.2 QoS-driven Deployment (FR2)

The system shall deploy application components based on user-defined QoS goals, ensuring optimal alignment with specified requirements such as energy consumption, operational cost, and performance. This includes:

- **QoS-Aware Scheduling:** Selecting deployment locations by evaluating available resources against user QoS targets (e.g., cost, energy, and performance).
- **Dynamic Adaptation:** Adjusting deployment decisions dynamically if the QoS parameters or resource availability change.

3.2.3 Disruption Recovery and Redeployment (FR2)

The system shall detect any disruptions (e.g., node failures or resource overloads) affecting running application components and automatically redeploy them to another cluster within the same domain if needed. This requirement involves:

- **Continuous Monitoring:** Tracking cluster health, resource usage, and application status to identify problems promptly.
- **Automatic Migration:** Re-deploying affected components in alternative clusters according to the user's QoS configurations and preferences.

3.2.4 Non-functional requirement: Fault Tolerance (NFR1)

The framework shall exhibit fault-tolerant behavior, continuing to operate despite failures in hardware, software, or network resources. This involves redundancy, employing backup instances to mitigate single points of failure.

By satisfying these functional and non-functional requirements, the proposed system will ensure reliable, high-performance, cost-effective, and energy-efficient deployments across cloud, fog, and edge domains.

3.3 Concept

This subsection presents the high-level architecture of our system, which revolves around three major components: the Knowledge Base (KB), the Resource Lead Agent (RLA), and the Resource Agent (RA). Figure 3.10 provides a visual overview of how these components fit together.

The concept introduces a test bed for the distributed resource orchestration until the actual Swarmchestrator system is constructed. It must be noted that the main entities (such as RA, RLA, KB, etc.) implemented in this concept and its functionalities may vary in the other deliverables since the concept introduced in this report follows the architectural suggestions in deliverable D2.1. Here in D1.1, we demonstrate an alternative solution that stems from D2.1, but the functionalities we provide in this subsection for the main entities are abstracted. For example, KB has its own working package in Swarmchestrator, and is more complex than how we implemented in D1.1. All terms, acronyms, and definitions mentioned in this report should be assessed in the context of D1.1.

Our system is divided into three administrative domains: cloud, fog, and edge. Each domain can contain multiple clusters: the cloud domain may run one or more full-fledged Kubernetes clusters; the edge domain, to save resources, may run one or more lightweight K3s clusters; and the fog domain may host either K8s or K3s clusters, depending on available capacity. Every cluster comprises one or more nodes (physical or virtual), each classified as either a control plane or a worker component. Clusters that run user applications must include one worker node running a Resource Agent (RA). Clusters that participate in the leadership group must run a Resource Lead Agent (RLA) on one of their nodes. Consequently, a single cluster can fulfill both the execution and leadership roles by running both RLA and RA.

The Knowledge Base (KB) holds global state and metadata for all k8s/k3s clusters, cluster nodes, and applications. The Resource Lead Agent (RLA) manages leadership (via Raft), scheduling logic, and the system's API server. The Resource Agent (RA) ensures that clusters receive the correct deployments while continuously reporting their node and application health to the RLA.

3.3.1 Knowledge Base (KB)

In Swarmchestrator, we introduce a separate knowledge layer that is likewise used to manage state. An alternative strategy could rely on a Raft-based data store for each Raft-enabled cluster to keep the system state, where the data store itself manages distributed leadership and data replication. In such a setup, all system information would live under the data store's consensus, and traditional Raft followers would forward read requests to their leader for state consistency. In contrast, our method only uses Raft for determining leadership among the RLAs. We do not manage our system state in Raft itself. Instead, we leave all system data in the KB, thereby allowing Raft followers to process read requests directly (since the KB acts as the canonical data store). This diverges from more conventional Raft-based systems, in which followers typically forward every request to the leader to

ensure linearizability [28]. By decoupling storage from Raft, our approach avoids substantial overheads and remains flexible in how it handles data.

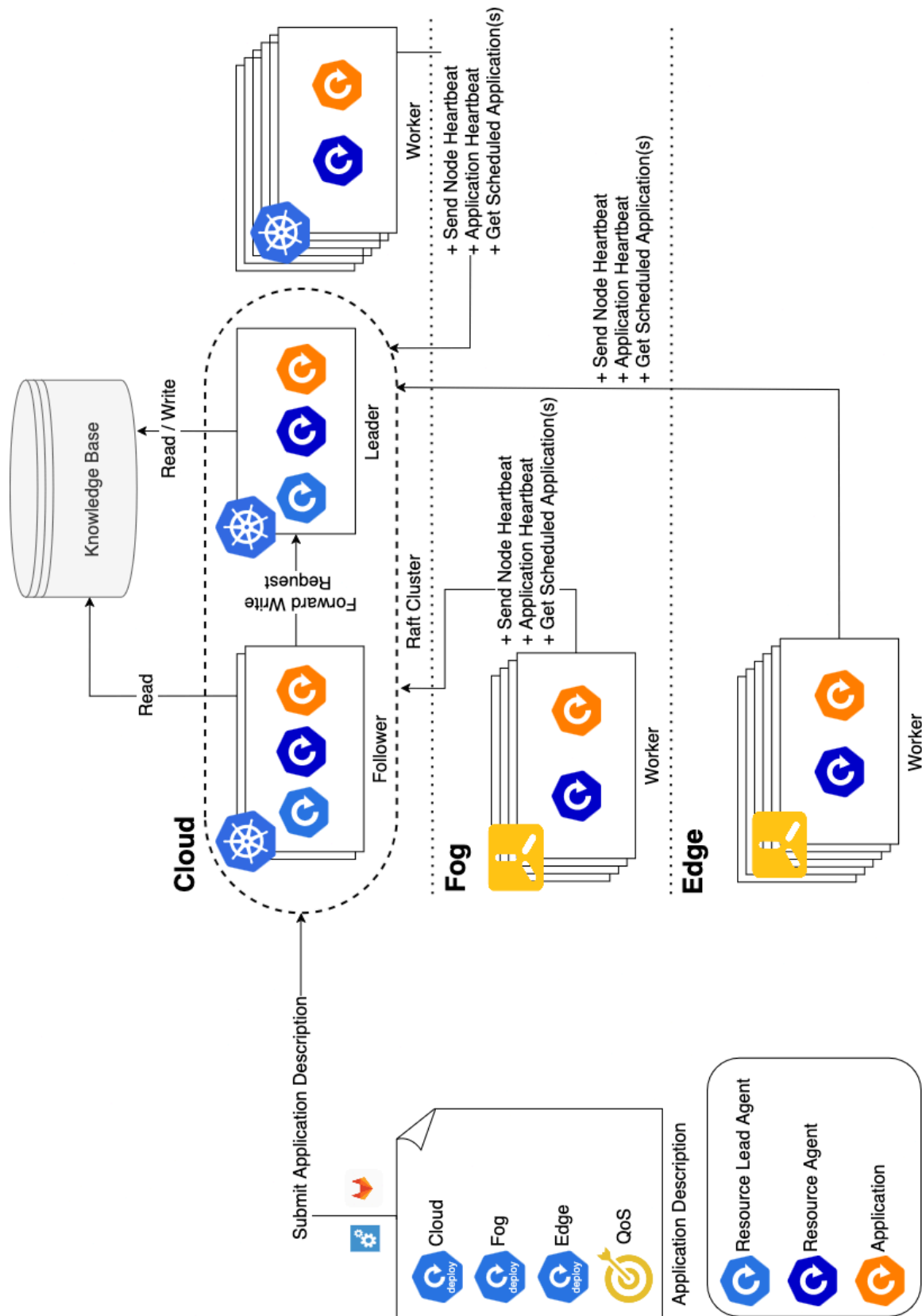


Figure 3.10: Architecture Overview

3.3.2 Resource Lead Agent

The RLA handles high-level orchestration tasks that require a global view of the system. It is installed on a certain number of cloud clusters that form a Raft cluster. As part of the Raft cluster, exactly one RLA instance acts as the leader, while every other RLA instance acts as a follower. Its key components are:

- API Server
- Raft
- Scheduler

API Server. API server exposes REST endpoints for CRUD (create/read/update/delete) operations on clusters, nodes, and applications. Read operations are handled by both leader and follower RLAs, but write operations (or any changes) are routed to the leader RLA in the Raft cluster, ensuring consistent storage in the KB.

Raft. The Raft layer inside every RLA-enabled cluster exists solely to provide leadership consensus. All RLAs bootstrap into a single Raft cluster so that, at any given moment, exactly one cluster is recognized as the leader and the remaining clusters act as followers. Election time-outs and heartbeats keep this leadership view fresh, allowing the system to recover quickly if the current leader becomes unavailable.

Unlike conventional deployments, where a Raft log replicates user data, our design stores all applications and cluster states in the Knowledge Base (KB). The Raft log, therefore, carries only the minimal metadata required to agree on leadership and membership. By keeping the consensus layer this lightweight, we gain fast fail-over and avoid the overhead of shipping large state snapshots among clusters. Followers can answer read-only API requests directly against the KB, whereas mutating requests are transparently forwarded to the current leader, giving the system a single, strongly ordered write path without burdening Raft with full data replication.

This minimalist arrangement cleanly separates responsibilities: Raft guarantees that exactly one RLA coordinates scheduling and updates, the KB holds the authoritative data, and Resource Agents simply forward their requests to any RLA instance, leaving the RLAs themselves to decide whether a given operation is served locally or routed to the leader.

Scheduler. The Scheduler runs exclusively on the current RLA leader and is responsible for turning high-level application requests into concrete placement decisions across the cloud, fog, and edge domains. Its control loop wakes up at regular, configured intervals and executes two complementary tasks: assigning pending applications and recovering stalled ones.

First, the Scheduler queries the KB for applications that have any service definition (cloud, fog, or edge) but lack a corresponding cluster assignment. It then gathers the latest inventory of clusters and their nodes, grouping them by domain. For every unassigned service, the Scheduler invokes a pluggable scoring component that ranks candidate nodes against the application's QoS goals (energy, pricing, performance). The highest-ranked

cluster, together with a filtered list of suitable nodes, is written back to the application record in the KB. An update is committed only when all services of the application have received a valid cluster assignment, ensuring that partially scheduled workloads do not block future re-evaluations.

Secondly, the Scheduler scans for stalled applications whose last heartbeat in the KB is older than a configured grace period and whose status is neither failed nor progressing. For each such service, it clears the existing cluster and node references in the KB, thereby returning the application to the pending pool. On the next scheduling cycle, the application will be treated like a fresh request and matched to a new cluster that satisfies its QoS goals.

The ranking logic itself is encapsulated behind an interface so that different multi-objective optimization algorithms can be experimented with; in our prototype, we employ a Borda-count-based scorer to balance the various QoS dimensions. By limiting all writes to the leader RLA and persisting decisions only in the KB, the Scheduler remains stateless and can be restarted or migrated without losing context.

The Scheduler communicates with the KB exclusively through an interface. Because every KB operation, creating, listing, or updating clusters, nodes, and applications, is accessed via this interface, the physical datastore can be swapped without touching scheduler logic. A relational database, a NoSQL store, or even a blockchain ledger can therefore act as the KB, provided it implements the same set of methods for cluster, node, and application management. This extra layer of abstraction keeps the scheduling component independent of storage details and future-proofs the design against changes in persistence technology.

3.3.3 Resource Agent

The Resource Agent (RA) is deployed on one worker node in every cluster that is expected to run user workloads. Its life cycle is governed by a periodic control loop that executes four independent tasks: synchronizing the current pool of RLA endpoints, exporting node information, deploying newly scheduled jobs, sending application heartbeats, and cleaning up any running jobs that are no longer necessary. The loop itself is triggered at a configurable interval, allowing operators to tune the trade-off between control-plane freshness and network overhead. Conceptually, the RA plays the role of a worker: it never initiates communication with peer RAs or with the KB but instead interacts only with the pool of RLAs discovered at registration time.

Upon start-up, the RA registers its cluster with any reachable RLA provided by the cluster admin as a bootstrapping configuration, receiving in return the current list of all RLA endpoints. The agent keeps no knowledge of which RLA is the leader or the follower. It simply directs every subsequent request to one entry in the pool in a round-robin fashion. At a configurable interval, it refreshes the pool of RLAs for any changes. This pool is stored on disk so the agent can reconnect after a restart without waiting for external configuration. The registration request includes basic cluster metadata, after which the RA begins sending periodic status reports that describe only worker nodes. K8s control plane nodes are intentionally excluded because they cannot host user workloads.

RA collects a snapshot of worker-node resources like CPU capacity, memory, and pressure indicators such as PID, memory, or disk pressure using the Kubernetes API and forwards that snapshot to one of the RLAs. In a separate request, the RA asks whether any services have been scheduled for its cluster but are still pending deployment. For every returned service definition, the agent substitutes domain placeholders with concrete IP addresses and adds additional metadata if added by the user, and applies the manifests locally. Finally, the RA assembles a status report for every application currently running on the cluster and transmits these heartbeats to an RLA. Using the response to the request, RA also cleans up any unnecessary running services.

Because each of the four steps is issued as an independent call, the RA remains responsive even if one action is delayed or fails. The next loop simply retries the affected step with a different RLA. Although the agent is not completely stateless, it caches the RLA pool and records basic metadata about the applications it has deployed. This local state is limited to what is necessary for reconnecting and correlating heartbeats. With no need to serve inbound traffic and only a lightweight state on disk, additional RAs can be added linearly as new clusters join the cloud, fog, or edge domains, and any agent can restart without coordination overhead. All cluster-level orchestration, leadership, and scheduling decisions continue to reside in the RLA/KB control plane, preserving a clear separation between control and data paths.

RLAs never initiate network calls to RAs. Instead, RAs continuously poll RLAs and push status reports back, mirroring the design pattern of Kubernetes. Consequently, all traffic flows outbound from RAs to RLAs, keeping the system loosely coupled and eliminating the need for inbound connectivity on the RAs.

3.4 Architecture

The components reason about two orthogonal views of the world:

1. Infrastructure view – what physical or virtual resources exist, and what is their real-time condition?
2. Workload view – what is the desired state of the application, what quality targets (QoS) do they declare, and what is the actual state?

Both views are stored in the KB, but they are not served directly from the database. All-access is mediated by the RLA's REST API-server component, which exposes a versioned REST interface. Through that interface:

- Cluster administrators can list, patch, or delete clusters and create, list, patch, or delete applications without touching the KB.
- RAs register the cluster, refresh node snapshots, and send application heartbeats.

3.4.1 Infrastructure Model

A Cluster record identifies an administrative unit that runs its own K8s/K3s control plane. Besides a monotonically increasing UUID, it has:

- Type – Refers to the administrative domain of the cluster: cloud, fog, or edge.
- Role – Refers to the role of the cluster. It can either be a worker cluster, which includes only an RA responsible for deploying user workloads, or a master cluster, which also includes an RLA and is part of the Raft cluster.
- RaftID – This can be null if the cluster is just a worker. If it's a master cluster, it has a unique integer ID used by Raft to identify the nodes as part of the Raft cluster.
- IP – The IP address of the cluster's ingress. It is used by applications for service-to-service communication using a path-based pattern, e.g., cluster_IP/app-name/...
- CreatedAt – Timestamp with a zone of the first successful registration.
- UpdatedAt – Timestamp with a zone of the last change to any cluster-level field.

Every physical or virtual machine that is part of the cluster is represented as a Node object. Every Node belongs to exactly one Cluster.

- Name – The name of the node is unique within the cluster, but it's not globally unique.
- Location – The location of the cluster node, using a time zone identifier (e.g., Europe/Berlin). Reserved for future scheduling features.
- CPU – The number of CPUs the node has.
- CPUArch – The architecture of the CPU (e.g., amd64 or arm64).
- Memory – The amount of memory of the node in bytes.
- NetworkBandwidth – The network bandwidth of the node in bps (bits per second).
- EphemeralStorage – The ephemeral storage of the node in bytes.
- Energy – The energy consumption of the node in kWh (kilowatt hour).
- Pricing – The pricing of the node in EUR/h (euros per hour).

These attributes are mostly static, changing rarely. On the other hand, dynamic ones such as IsReady, IsSchedulable, and pressure flags: PID, Memory, and Disk are updated by the RA at each heartbeat.

Nodes are also associated with CreatedAt and UpdatedAt timestamps with a zone. The UpdatedAt field is important because nodes that are not fresh (i.e., those whose status has not been updated for a certain period of time) are removed from consideration by the scheduler.

3.4.2 Workload Model

The entity is split into two groups of attributes:

- Declarative intent – the information that comes straight from the user when an application is created or patched.
- Observed state – values that are filled in and continuously maintained by the orchestration layer once the application has been admitted.

Declarative Intent. These fields define the desired state and change only through explicit user action:

- Name – a global, human-readable identifier.
- Labels – free-form key/value pairs that will be copied onto the generated Kubernetes objects.
- QoS – a triplet {Energy, Pricing, Performance} whose entries range from 0.0 (low importance) to 1.0 (high importance); the scheduler treats them as weights.
- CloudSvc / FogSvc / EdgeSvc – optional JSON blobs (one per administrative domain) containing the exact Kubernetes manifests: Deployments, Secrets, ConfigMaps, and so on for that slice. At least one of the three blobs must be present.
- CloudSvcVersion / FogSvcVersion / EdgeSvcVersion – monotonically increasing UUIDs that bump every time the user changes the corresponding service blob or QoS.

Observed State. Added by the leader RLA after scheduling and continuously updated by the RAs.

- CloudSvcCluster / FogSvcCluster / EdgeSvcCluster – UUID of the cluster that is supposed to deploy or run the respective service objects.
- CloudSvcNodes / FogSvcNodes / EdgeSvcNodes – comma-separated list of node names that are supposed to be run or running the actual workload.
- CloudSvcStatus / FogSvcStatus / EdgeSvcStatus – Status of the workload (e.g., Progressing | Healthy | Failed).
- CloudSvcHeartbeat / FogSvcHeartbeat / EdgeSvcHeartbeat – latest timestamp reported by the responsible RA.

Every application record also carries two system-related attributes:

- ID – a monotonic, ever-growing UUID assigned at admission time.
- CreatedAt – set once when the admission request is first accepted; never changes.
- UpdatedAt – refreshed automatically whenever any desired and observed field is modified.

3.5 Implementation

We decided to adopt a mono-repository layout instead of maintaining multiple repositories. The root of the repository is therefore organized into the following top-level directories:

- knowledge-base/: Our KB implementation using PostgreSQL. This directory includes the following components:
 - docker-compose.yaml spins up a local Postgres instance.
 - SQL migrations (managed with golang-migrate).
- resource-lead-agent/: Implementation of the RLA. This directory includes the following components:
 - source code of RLA.
 - Kubernetes manifests (Deployment, Service, ConfigMap, ...) and Kustomize manifest.
 - test files for unit and integration test.
 - Swagger specification for the REST API.
 - Bruno collection for the REST API.
- resource-agent/: implementation of the RA. This directory includes the following components:
 - source code of RA.
 - Kubernetes manifests (Deployment, Service, ConfigMap, ...) and Kustomize manifest.
- sample-application/: evaluation workload. This directory includes the clone of Istio bookinfo application from Github²⁸ with minimal tweaks.
- scripts and configs: The root directory includes the following components:
 - script to setup evaluation environment using Kind²⁹ (docker based multi-node K8s cluster) clusters.
 - configurations for Kind clusters.

3.5.1 Knowledge Base

The KB within the scope of this deliverable is a simple, shallow database that can be easily replaced in the future. While the primary intuition, design, and architecture for the KB is discussed in deliverable D3.1, the goal for the generic KB implementation used in this investigation was to stand-in for the final system, allowing our evaluation to proceed in parallel to the development of the final, Swarmchestrator KB.

The KB is implemented using a PostgreSQL³⁰ database (v17.4) that persists the exact objects defined in the Infrastructure and Workload models. Each model entity maps one-to-one to a

²⁸ <https://github.com/istio/istio/blob/1.25.2/samples/bookinfo/platform/kube/bookinfo.yaml>. Last accessed: 4.6.2025

²⁹ <https://kind.sigs.k8s.io>. Last accessed: 4.6.2025

³⁰ <https://www.postgresql.org>. Last accessed: 5.6.2025

relational table, no additional tables or views are introduced. Schema changes are applied through plain SQL migration files using `golang-migrate`³¹, and the most recent version of which is given in our implementation³².

All primary keys are generated as UUID v7 values, which are monotonically increasing and therefore sortable. All timestamps are recorded in UTC. Foreign-key constraints preserve referential integrity between the three relations; no additional triggers or stored procedures are required.

3.5.2 Resource Lead Agent

The following paragraphs cover the implementation of the RLA³³. All high-level ideas have already been explained in Section 3.3, “Concept”. It is written in the Go³⁴ programming language and built on top of `cobra`³⁵ CLI tool, `go-chi`³⁶ router, `viper`³⁷ configuration management tool and `etcd-raft`³⁸ lightweight implementation of Raft algorithm and followed the clean architecture approach described in Section 3.1.4, “Clean Architecture”.

Project layout.

- `algo/` – algorithms: `borda` for node scoring and `Raft` for leadership.
- `cmd/` – RLA entry point. Starts the HTTP server, the Raft peer, and the scheduler ticker.
- `domain/` – immutable value objects and validation logic; acts as Clean Architecture's entity (core business) ring.
- `internal/` – configuration parsing and REST API layer; acts as Clean Architecture's interface-adapter ring.
- `resource-lead-agent/` – use-case layer that implements application-specific business rules around the domain.
- `external/kb/` – external service layer for KB access.
- `external/leader/` – HTTP client to forward mutating requests to the current leader.

Raft. We did not implement the Raft ourselves instead we forked `etcd/contrib/raftexample`³⁹ which is an example Raft-based key-value store. We trimmed the original Raft example down to the bare-bones consensus layer: the Raft state machine plus its WAL + snapshot logic remain, while the sample key-value store and its transport were dropped and replaced by a minimalist HTTP-based REST transport.

³¹ <https://github.com/golang-migrate/migrate>. Last accessed: 5.6.2025

³² <https://github.com/dos-group/swarmchestrator-alternative/tree/main/knowledge-base/migrations>. Last accessed: 5.6.2025

³³ <https://github.com/dos-group/swarmchestrator-alternative/tree/main/resource-lead-agent>. Last accessed: 5.6.2025

³⁴ <https://go.dev>. Last accessed: 5.6.2025

³⁵ <https://github.com/spf13/cobra>. Last accessed: 5.6.2025

³⁶ <https://github.com/go-chi/chi>. Last accessed: 5.6.2025

³⁷ <https://github.com/spf13/viper>. Last accessed: 5.6.2025

³⁸ <https://github.com/etcd-io/raft>. Last accessed: 5.6.2025

³⁹ <https://github.com/etcd-io/etcd/tree/release-3.5/contrib/raftexample>. Last accessed: 5.6.2025

Scheduler. The scheduler wakes up on a configurable interval to:

1. place pending services – assign clusters + nodes using the Borda scorer, then persist the decisions in the KB.
2. re-queue stalled services – any workload whose last heartbeat is older than a configurable grace period is considered stalled, its cluster/node bindings are cleared, and it re-enters the pending pool for the next cycle.

Scoring. Borda implements the interface expected by the scheduler. Its single public method `ScoreAndFilterNodes` receives (i) the up-to-date node(s) snapshot and (ii) the QoS requirements. The routine proceeds as follows:

1. **Eligibility filter** – remove every node that is not ready, unschedulable or under resource pressure. An empty result stops processing and returns no placement.
2. **Per-attribute Borda ranking** – for each of the attributes: energy, pricing, capacity (CPU, memory, bandwidth, storage) the slice is sorted and a Borda score is assigned. Low values win for energy/pricing, high values win for capacity attributes.
3. **Capacity factor** – the four capacity Bordas are summed into a single `CapacityBorda` that represents the raw performance of the node.
4. **QoS-weighted node score** –

$$w = Energy_{Borda} \cdot q_E + Price_{Borda} \cdot q_P + Capacity_{Borda} \cdot q_C$$

where (q_E, q_P, q_C) are the user weights (energy, price, performance) to represent the priority scores for each QoS target. If priority scores are not given by the user, we assume that all weights are equal to 1.0.

5. **Threshold filter** – compute the arithmetic mean of all w ; keep only nodes whose score is greater than or equal to that mean.
6. **Cluster aggregation** – for each cluster sum (i) all node scores and (ii) scores of the retained nodes. Clusters are ordered by the retained score; ties are broken with the total score.
7. **Return value** – the ID of the highest ranked cluster and the list of retained node names belonging to that cluster.

REST API. The Bruno⁴⁰ collection and generated OpenAPI contract lives in the repository, but two routes deserve special attention: `POST/v1/applications` and `PATCH/v1/applications/{id}`. Both receive up to four YAML manifests as shown in table 3.1. The RLA REST API endpoints are also given in Table 3.2.

⁴⁰ <https://www.usebruno.com>. Last accessed 6.6.2025

Field Name	Purpose
application	Minimal metadata (name, labels, QoS)
cloud_svc	K8s objects destined for the <i>Cloud</i> domain
fog_svc	K8s objects destined for the <i>Fog</i> domain
edge_svc	K8s objects destined for the <i>Edge</i> domain

Table 3.1: Applications POST and PATCH endpoints' fields

Verb & Path	Action
POST /clusters	register cluster
PATCH /clusters/{id}	update cluster (leader only)
PUT /clusters/{id}/nodes	upsert node snapshot
GET /cluster/config	obtain mapping (cluster ID to ingress IP)
POST /applications	admit application
PATCH /applications/{id}	update application
GET /clusters/{cid}/applications/scheduled	list workloads for worker
POST /clusters/{cid}/applications/{aid}/status	application heartbeat

Table 3.2: Resource Lead Agent REST API endpoints

Two conventions make the manifests portable across domains:

- Ingress rule – Each domain manifest must contain an Ingress whose firstpath segment equals the application name.
- Cross-domain placeholders – Whenever a Service from another domain is referenced, we do not hard-code an address; instead we use one of the tokens:

[CLOUD_CLUSTER_IP] [FOG_CLUSTER_IP] [EDGE_CLUSTER_IP]

The Resource Agent replaces these placeholders with concrete cluster IPs immediately before applying the manifest.

The application field is a thin CRD-style yaml that carries only the metadata (name, labels, QoS) required by the scheduler. This is to keep things simple for the implementation of the

proposed system, but in reality, the Swarmchestrate project will utilize TOSCA-formatted files as described in deliverable D2.1.

3.5.3 Resource Agent

The following paragraphs cover the implementation of the RA⁴¹. All high-level ideas have already been explained in Section 3.3. It is also written in the Go programming language and built on top of the cobra CLI tool, viper configuration management tool and the official Kubernetes Go client libraries⁴² and also followed the clean architecture approach described in Section 3.1.4.

Project layout.

- cmd/ – RA entry point. It starts the HTTP endpoint, performs the first-time cluster registration and then launches the ticker that (i) pushes the current node snapshot, (ii) polls for scheduled workloads that are still pending deployment, (iii) sends heartbeat updates for every application already deployed in this cluster. All three subtasks are run at the same configurable interval and (iv) polls clusters' configs (mapping of cluster-ID and ingress IP).
- domain/ – immutable value objects and validation logic; acts as Clean Architecture's entity (core business) ring.
- resource-agent/ – use-case layer that implements application-specific business rules around the domain.
- external/resource-lead/ – external service layer for resource-lead-agent access.

Registration. At start-up, the RA checks whether the local cluster is already registered.

It tries to read the *self* ConfigMap in the dedicated *swarmchestrate* namespace.

- ConfigMap present – the persisted cluster-ID and role are loaded into memory and the agent resumes its regular duties.
- ConfigMap absent – the cluster is joining for the first time. The RA determines its external IP and submits it to the bootstrap resource-lead-agent instance configured by the administrator during deployment. The resulting UUID is written to a newly created ConfigMap and cached in memory.

⁴¹ <https://github.com/dos-group/swarmchestrate-alternative/tree/main/resource-agent>. Last accessed: 6.6.2025

⁴² <https://github.com/kubernetes/client-go>. Last accessed: 6.6.2025
<https://github.com/kubernetes/api>. Last accessed: 6.6.2025
<https://github.com/kubernetes/apimachinery>. Last accessed: 6.6.2025

During the same bootstrap phase, the agent also verifies that the auxiliary ConfigMaps for clusters (keeps mapping of clusters and their IP) and applications (running applications). All subsequent operations rely on their presence.

Send node snapshot. The RA invokes the k8s API and enumerates every node and filters out any control-plane nodes, and sends it to the RLA.

Polls cluster config. Since clusters may join or leave at any time, the RA periodically fetches the current cluster ID \rightarrow ingress IP mapping from the RLA via REST and refreshes the local ConfigMap which is used during application deployment.

Polls applications. Through the dedicated REST endpoint, the RA retrieves every workload recently scheduled onto the local cluster. For each application then:

1. ensures that the target namespace exists (creating it on demand);
2. applies every shipped Kubernetes object, merges the application labels, and replaces all cross-domain placeholders ([`CLOUD_CLUSTER_IP`], [`FOG_CLUSTER_IP`], [`EDGE_CLUSTER_IP`]) with the concrete IPs obtained during the poll-cluster-config step;
3. restricts every Deployment to the exact worker node set chosen by the scheduler.

Once the deployment finishes successfully, the RA adds (or updates) an entry in the applications ConfigMap that records the application ID, its current version, the list of applied Kubernetes objects, and the deployment timestamp. This local register serves as the single source of truth for later health checks and clearly documents which workloads the cluster is responsible for.

Send application heartbeat. For every application recorded in the applications ConfigMap the RA derives a status and reports it to the RLA:

- Healthy - every Deployment has reached its desired replica count;
- Progressing - at least one Deployment is still rolling out and the deployment age is below a configurable timeout;
- Failed - any other situation (crash-loop, missing objects, timeout) that requires manual intervention.

Cleanup application. Whenever the RLA answers an application heartbeat with 404 Not Found the RA concludes that the workload has been withdrawn from this cluster. It immediately deletes the corresponding Kubernetes namespace (removing all objects created during deployment) and removes the application's entry from the ConfigMap so that it is no longer considered in future heartbeat cycles.

3.6 Evaluation

3.6.1 Setup

For evaluation, we set up our testbed clusters using Kind. We created nine clusters—three each for cloud, fog, and edge—and, within each domain, assigned one cluster to each of three profiles: energy-efficient, cost-efficient and performance-efficient. Each cluster has three nodes: one control-plane node and two worker nodes. Because all clusters run in Docker within a single machine, we do not have real billing, energy or bandwidth telemetry. Instead, we assigned representative values manually based on publicly available data.

Energy Consumption. We used the energy coefficients from Cloud Carbon Footprint's Appendix⁴³ and applied:

$$E = \frac{(W_{idle} + W_{max})}{2} \times PUE / 1000 \times 1 h$$

where,

W_{idle} = Idle CPU average power draw (W),

W_{max} = Maximum CPU average power draw (W),

PUE = Power Usage Effectiveness (unitless).

This yields:

$$E_{AWS} = \frac{0.74 + 3.50}{2} \times 1.135 / 1000 \times 1 h \approx 0.0024062 kWh,$$

$$E_{GCP} = \frac{0.71 + 4.26}{2} \times 1.10 / 1000 \times 1 h \approx 0.0027335 kWh.$$

The midpoint between AWS and GCP is:

$$E_{mid} = \frac{E_{AWS} + E_{GCP}}{2} \approx 0.00256985 kWh$$

Thus, we assigned:

- **Energy-efficient:** lowest value (AWS, 0.0024042 kWh)
- **Cost-efficient:** midpoint value (0.0025689 kWh)
- **Performance-efficient:** highest value (GCP, 0.0027335 kWh)

Cost (EUR/h). From AWS On-Demand pricing⁴⁴, *t4g.nano* bills \$0.0042/h and *p4d.24xlarge* bills \$32.7726/h. Assuming 1 EUR = 1 USD, we directly use:

⁴³ <https://www.cloudcarbonfootprint.org/docs/methodology/#appendix-i-energy-coefficients>. Last accessed: 6.6.2025

⁴⁴ <https://aws.amazon.com/de/ec2/pricing/on-demand/>. (Location Type: AWS Region, Region: US East (Ohio), OS: Linux). Last accessed: 6.6.2025

$$C_{min} = 0.0042 \text{ EUR/h},$$

$$C_{max} = 32.7726 \text{ EUR/h},$$

$$C_{mid} = \frac{C_{min} + C_{max}}{2} \approx 16.3884 \text{ EUR/h}.$$

Thus, we assigned:

- **Cost-efficient:** lowest cost (0.0042 EUR/h)
- **Energy-efficient:** midpoint cost (16.3884 EUR/h)
- **Performance-efficient:** highest cost (32.7726 EUR/h)

Network bandwidth. AWS specs list "Up to 5 Gbps" for *t4g.nano* and "400 Gbps" for *p4d.24xlarge*. Per requirement, we capped performance at 100 Gbps. Thus:

$$B_{min} = 5 \text{ Gbps},$$

$$B_{max} = 100 \text{ Gbps},$$

$$B_{mid} = \frac{5+100}{2} = 52.5 \text{ Gbps}.$$

Thus we assigned:

- **Cost-efficient:** 5 Gbps
- **Energy-efficient:** 52.5 Gbps
- **Performance-efficient:** 100 Gbps

Cluster Profile	Energy (kWh)	Cost (EUR/h)	Bandwidth (Gbps)
Energy-efficient	0.0024042	16.3884	52.5
Cost-efficient	0.0025689	0.0042	5
Performance-efficient	0.0027335	32.7726	100

Table 3.3: Assigned energy, cost, and bandwidth parameters for each cluster profile.

For the fog and edge clusters, we used the same assigned values for energy, cost and bandwidth as the cloud clusters for consistency. In reality, the energy consumption, network bandwidth and costs for fog and edge devices would be different from those of cloud data center nodes. However, because the assigned values do not conflict across different cluster domains (cloud, fog, and edge) and maintain clear distinctions among energy-efficient, cost-efficient and performance-efficient categories within each domain, we reused the same parameters. To further simulate the characteristics of energy-efficient and cost-efficient

clusters compared to performance-efficient clusters we manually reduced the reserved system resources. Specifically, for cost-efficient and energy-efficient worker nodes, we reduced the memory by 5 GiB, CPU cores by 4 and ephemeral storage by 2 GiB. These adjustments were applied through *kubeadmConfigPatches* during cluster setup.

3.6.2 Test Application

To evaluate the system functionality, we did not develop a custom application from scratch. Instead, we forked the existing Istio Bookinfo application deployment. We minimally modified the configuration to suit our experimental requirements. Additionally, we introduced a Kubernetes Ingress object to expose the application through our ingress controller.

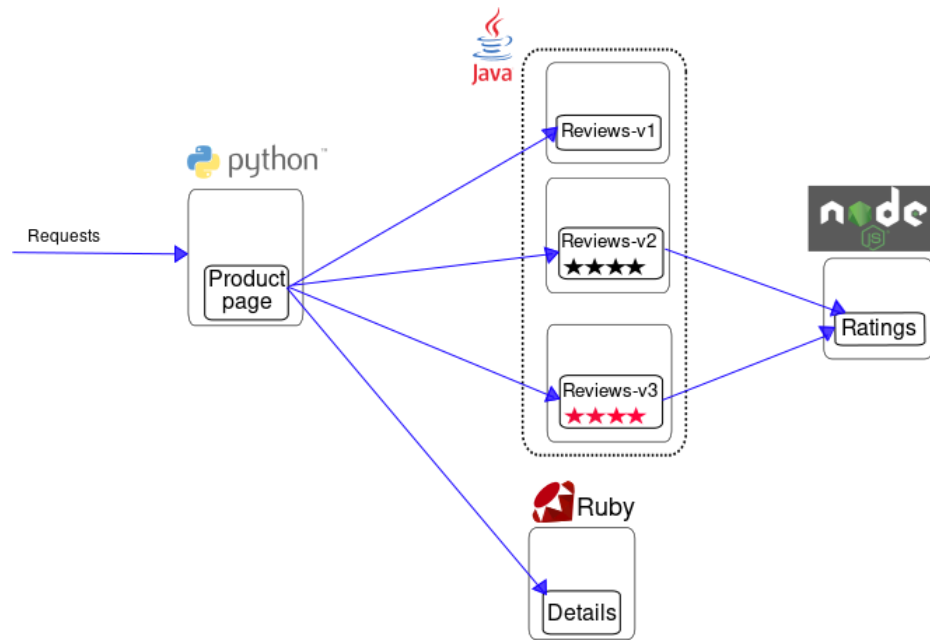
The Bookinfo application simulates a real-world microservices-based e-commerce system that provides information about a book, including its details and user reviews. The application is composed of the following components:

- **productpage:** The frontend service that displays the book information by aggregating data from the details and reviews services.
- **details:** Provides descriptive information about the book.
- **reviews:** Provides user reviews for the book and interacts with the ratings service to display ratings.
- **ratings:** Supplies numerical star ratings for each review.

To emulate a distributed architecture aligned with Cloud-to-Edge paradigms, we deployed the microservices across our clusters as follows:

- **Cloud cluster:** Hosts the productpage service.
- **Fog cluster:** Hosts the details and reviews services.
- **Edge cluster:** Hosts the ratings service.

Figure 3.11 shows the Istio Bookinfo architecture, which our test application is based on. While we retained the overall structure, our deployment reassigns services across the cloud, fog and edge clusters to reflect our experimental setup.

Figure 3.11: Istio Bookinfo application architecture⁴⁵

3.6.3 Evaluation Scenarios

To evaluate the system's compliance with its functional and non-functional requirements, we designed a series of tasks that test provisioning, dynamic adaptation, fault tolerance and redundancy handling. Each task is associated with a specific goal and an expected system behavior.

Task 1: Initial Deployment Based on QoS (FR1, FR2).

- **Objective:** Evaluate the system's ability to provision applications across domains in a QoS-aware manner.
- **Action:** Deploy the test application with a defined QoS goal (e.g., prioritizing energy efficiency or cost).
- **Expectation:** The application should be automatically deployed to a combination of clusters (cloud, fog and edge) that best meet the specified QoS profile. All services must be instantiated in appropriate clusters without manual placement.

Task 2: Dynamic QoS Update and Rescheduling (FR2).

- **Objective:** Test the system's support for adapting to changing QoS requirements.
- **Action:** Update the QoS goal of a running application (e.g., from energy efficiency to performance).

⁴⁵ <https://istio.io/latest/docs/examples/bookinfo/noistio.svg>. Last accessed: 6.6.2025

- **Expectation:** The system should automatically migrate the application to a new set of clusters that better satisfy the updated QoS goal. Services must be removed from the previous clusters to avoid redundancy or resource waste.

Task 3: Cluster Failure and Automatic Migration (FR3, NFR1).

- **Objective:** Test the system's fault tolerance and automatic recovery capability.
- **Action:** Simulate a cluster failure by removing a resource agent (RA).
- **Expectation:** The application should be automatically rescheduled to another cluster with the closest matching QoS profile. The system should detect the failure and restore the application.

Task 4: Redundancy and Failover (NFR1).

- **Objective:** Validate the redundancy and leadership election mechanism in case of failure.
- **Action:** Manually remove the leader RLA from the cluster.
- **Expectation:** A new leader should be automatically elected from among the remaining active RLAs. System coordination should resume without user intervention or loss of control logic.

3.6.4 Results and Observations

This section summarizes the system's behavior while executing the evaluation scenarios. The observations demonstrate the orchestration framework's ability to provision, adapt and recover in response to dynamic QoS goals and cluster-level failures.

Initial Cluster Setup and Configuration. Figure 3.12 shows the initial deployment of the nine clusters across cloud, fog, and edge domains. Each domain had three clusters: one energy-efficient, one cost-efficient, and one performance-efficient.

```

swarmchestrator % kubectl config get-clusters
NAME
kind-fog-performance
kind-cloud-energy
kind-edge-energy
kind-edge-cost
kind-edge-performance
kind-fog-energy
kind-cloud-performance
kind-fog-cost
kind-cloud-cost
swarmchestrator %

```

Figure 3.12: Initial deployment of nine clusters (3 cloud, 3 fog, 3 edge)

Each cluster was originally configured with two worker nodes. However, due to excessive CPU usage on the host machine, we reduced the number of worker nodes to one per cluster. The updated node layout is shown in Figure 3.13.

However, even after this reduction, system instability persisted. As a result, we permanently deleted the *cloud-cost* cluster, followed by the *fog-cost* and *edge-cost* clusters. This left us with only six active clusters—two per domain (energy-efficient and performance-efficient profiles).

```

swarmchestrator % kubectl get nodes --context=cloud-energy
NAME                                STATUS    ROLES    AGE     VERSION
cloud-energy-control-plane           NotReady control-plane 17m     v1.33.0
cloud-energy-worker                  Ready     <none>      17m     v1.33.0
swarmchestrator % kubectl get nodes --context=cloud-cost
NAME                                STATUS    ROLES    AGE     VERSION
cloud-cost-control-plane            Ready     control-plane 17m     v1.33.0
cloud-cost-worker                   Ready     <none>      17m     v1.33.0
swarmchestrator % kubectl get nodes --context=cloud-performance
NAME                                STATUS    ROLES    AGE     VERSION
cloud-performance-control-plane     Ready     control-plane 17m     v1.33.0
cloud-performance-worker            Ready     <none>      17m     v1.33.0
swarmchestrator % kubectl get nodes --context=fog-energy
NAME                                STATUS    ROLES    AGE     VERSION
fog-energy-control-plane             Ready     control-plane 16m     v1.33.0
fog-energy-worker                    Ready     <none>      16m     v1.33.0
swarmchestrator % kubectl get nodes --context=fog-cost
NAME                                STATUS    ROLES    AGE     VERSION
fog-cost-control-plane              Ready     control-plane 16m     v1.33.0
fog-cost-worker                     Ready     <none>      16m     v1.33.0
swarmchestrator % kubectl get nodes --context=fog-performance
NAME                                STATUS    ROLES    AGE     VERSION
fog-performance-control-plane       Ready     control-plane 16m     v1.33.0
fog-performance-worker              Ready     <none>      16m     v1.33.0
swarmchestrator % kubectl get nodes --context=edge-energy
NAME                                STATUS    ROLES    AGE     VERSION
edge-energy-control-plane            NotReady control-plane 16m     v1.33.0
edge-energy-worker                   Ready     <none>      15m     v1.33.0
swarmchestrator % kubectl get nodes --context=edge-cost
NAME                                STATUS    ROLES    AGE     VERSION
edge-cost-control-plane              Ready     control-plane 15m     v1.33.0
edge-cost-worker                     Ready     <none>      15m     v1.33.0
swarmchestrator % kubectl get nodes --context=edge-performance
NAME                                STATUS    ROLES    AGE     VERSION
edge-performance-control-plane       Ready     control-plane 15m     v1.33.0
edge-performance-worker              Ready     <none>      15m     v1.33.0
swarmchestrator %

```

Figure 3.13: Final cluster topology after reducing worker nodes to 1 per cluster

Importantly, this also meant that one of the three follower RLA nodes was no longer available from the beginning of the evaluation. The system operated throughout all remaining tasks with only two Raft nodes, demonstrating the ability to maintain quorum despite the reduced number of RLA (Raft node).

Task 1: Initial Deployment Based on QoS. The application was first deployed with a QoS goal emphasizing energy efficiency (*energy: 1.0*). As shown in Figure 3.14, the system placed the *productpage* on *cloud-energy*, the *details* and *reviews* services on *fog-energy*, and the *ratings* component on *edge-energy*. The application was accessible through the ingress controller, and the UI rendered successfully


```

~ % kubectl get pods -n istio-sample --context=cloud-energy
NAME                READY   STATUS    RESTARTS   AGE
productpage-v1-5f64ff4ddf-8tlx5  1/1     Running   0           11m
~ % kubectl get pods -n istio-sample --context=fog-energy
NAME                READY   STATUS    RESTARTS   AGE
details-v1-687487c595-14cs2    1/1     Running   0           11m
reviews-v1-56747b7d8f-pq4m4    1/1     Running   0           11m
reviews-v2-5597f468f7-vjcqt    1/1     Running   0           11m
reviews-v3-75567c4497-ps9j2    1/1     Running   0           11m
~ % kubectl get pods -n istio-sample --context=edge-energy
NAME                READY   STATUS    RESTARTS   AGE
ratings-v1-7d96b99975-2d9j4    1/1     Running   0           11m
~ %

```

Figure 3.14: Initial QoS-aware deployment to energy-efficient clusters (Task 1)

Task 2: QoS Update and Redeployment. The QoS goal was updated from *energy: 1.0* to *energy: 0.0, performance: 1.0*. Figure 3.15 shows the system automatically cleaned up the running services from the *-energy clusters and rescheduled them to the *-performance clusters.

```

resource-agent % kubectl get pods -n istio-sample --context=cloud-energy
No resources found in istio-sample namespace.
resource-agent % kubectl get pods -n istio-sample --context=fog-energy
No resources found in istio-sample namespace.
resource-agent % kubectl get pods -n istio-sample --context=edge-energy
No resources found in istio-sample namespace.
resource-agent % kubectl get pods -n istio-sample --context=cloud-performance
NAME                READY   STATUS    RESTARTS   AGE
productpage-v1-59b57fbfb4-bhklc  1/1     Running   0           11s
resource-agent % kubectl get pods -n istio-sample --context=fog-performance
NAME                READY   STATUS    RESTARTS   AGE
details-v1-6f9ff5cc97-tc2r5    1/1     Running   0           110s
reviews-v1-84cffbdcf-zc8xb     1/1     Running   0           94s
reviews-v2-8bbc894bc-pf2vz     1/1     Running   0           94s
reviews-v3-ffd956d7-jk6wr      1/1     Running   0           93s
resource-agent % kubectl get pods -n istio-sample --context=edge-performance
NAME                READY   STATUS    RESTARTS   AGE
ratings-v1-55c9b6c9b4-qp2xq    1/1     Running   0           2m1s
resource-agent %

```

Figure 3.15: Automatic application migration to performance-efficient clusters after QoS change

Task 3: Cluster Failure Recovery. To simulate cluster failure, we manually deleted the RA on edge-performance. As a result, the cluster could no longer communicate with the rest of the system, and the ratings service on that cluster became isolated. Figure 3.16 shows that the system detected this failure and redeployed the ratings service to edge-energy. The remaining services on cloud-performance and fog-performance continued to operate unaffected.

Task 4: Redundancy and Leadership Continuity. Although one follower RLA node was removed early during the setup phase (due to resource pressure and deletion of cost clusters), the remaining two RLA nodes successfully maintained Raft quorum throughout the evaluation.

For the leadership continuity task, we used a separate control-plane layout: three cloud clusters—*cloud-energy*, *cloud-cost*, and *cloud-performance*—each ran an RLA, while *fog-energy* and *edge-energy* hosted only workload pods. The leader initially resided in *cloud-energy*. After deliberately deleting that RLA, the surviving RLAs in *cloud-cost* and *cloud-performance* detected the missing heartbeat, started a new election (term 7), and unanimously elected *cloud-performance* as the new leader (see Figure 3.17). With two RLAs still online, quorum and scheduling operations continued uninterrupted.

```

swarmchestrator % k delete -k resource-agent/deploy --context=edge-performance
serviceaccount "resource-agent" deleted
clusterrole.rbac.authorization.k8s.io "resource-agent" deleted
clusterrolebinding.rbac.authorization.k8s.io "resource-agent" deleted
configmap "resource-agent" deleted
deployment.apps "resource-agent" deleted
swarmchestrator % kubectl get pods -n istio-sample --context=edge-performance
NAME                                READY    STATUS    RESTARTS   AGE
ratings-v1-6dbccc64d5-679x4        1/1      Running   0           55m
swarmchestrator % kubectl get pods -n istio-sample --context=edge-energy
NAME                                READY    STATUS    RESTARTS   AGE
ratings-v1-584669bdb9-k8z8t        1/1      Running   0           5m9s
swarmchestrator % kubectl get pods -n istio-sample --context=cloud-performance
NAME                                READY    STATUS    RESTARTS   AGE
productpage-v1-7787db5f54-bh6xj    1/1      Running   0           55m
swarmchestrator % kubectl get pods -n istio-sample --context=fog-performance
NAME                                READY    STATUS    RESTARTS   AGE
details-v1-68bbcdf579-dnqg6        1/1      Running   0           55m
reviews-v1-76f7f9dccc-9csvz        1/1      Running   0           55m
reviews-v2-7868474f77-xdlbh        1/1      Running   0           55m
reviews-v3-6f7c477689-gzxx6        1/1      Running   0           55m
swarmchestrator %

```

Figure 3.16: Rating component migrated from failed edge-performance to edge-energy

Relying on the results, we show the proposed system worked robustly, handling offer selection mechanism when a cluster is down, and migrates to the new clusters with no issue.

```

[15.938ms] [rows:0] SELECT * FROM "applications"
2025/05/03 13:27:34 no applications found ...
2025/05/03 13:27:34 no applications found ...

2025/05/03 13:27:44 /app/external/kb/repository/pg/application.go:122
[9.410ms] [rows:0] SELECT * FROM "applications"

2025/05/03 13:27:44 /app/external/kb/repository/pg/application.go:122
[11.948ms] [rows:0] SELECT * FROM "applications"
2025/05/03 13:27:44 no applications found ...
2025/05/03 13:27:44 no applications found ...
received OS signal: terminated
2025/05/03 13:27:52 INFO server shutting down .....
i[redacted]2 ~ % kubectl logs resource-lead-agent-94fc48995-t2dr7 -n swar
mchestrator --context=cloud-energy
error: error from server (NotFound): pods "resource-lead-agent-94fc48995-t2dr7"
not found in namespace "swarmchestrator"
~ % █

-- kubectl logs resource-lead-agent-94fc48995-gvdw6 -n swarmchestrator --context=cloud-cost -f -- 80x18
raft2025/05/03 13:27:53 INFO: 2 [term: 6] received a MsgVote message with higher
term from 3 [term: 7]
raft2025/05/03 13:27:53 INFO: 2 became follower at term 7
raft2025/05/03 13:27:53 INFO: 2 [logterm: 6, index: 4, vote: 0] cast MsgVote for
3 [logterm: 6, index: 4] at term 7
raft2025/05/03 13:27:53 INFO: raft.node: 2 lost leader 1 at term 7
raft2025/05/03 13:27:53 INFO: raft.node: 2 elected leader 3 at term 7
(string) (len=46) "handle stalled apps - not a leader - returning"
(string) (len=48) "schedule pending apps - not a leader - returning"
(string) (len=48) "schedule pending apps - not a leader - returning"
(string) (len=46) "handle stalled apps - not a leader - returning"
(string) (len=48) "schedule pending apps - not a leader - returning"
(string) (len=46) "handle stalled apps - not a leader - returning"
(string) (len=46) "handle stalled apps - not a leader - returning"
(string) (len=48) "schedule pending apps - not a leader - returning"
(string) (len=48) (string) (len=46) "schedule pending apps - not a leader - retu
rning"
"handle stalled apps - not a leader - returning"

-- kubectl logs resource-lead-agent-94fc48995-hl52t -n swarmchestrator --context=cloud-performance -f -- 80x18
raft2025/05/03 13:27:53 INFO: 3 [logterm: 6, index: 4] sent MsgVote request to 1
at term 7
raft2025/05/03 13:27:53 INFO: 3 [logterm: 6, index: 4] sent MsgVote request to 2
at term 7
raft2025/05/03 13:27:53 INFO: raft.node: 3 lost leader 1 at term 7
raft2025/05/03 13:27:53 INFO: 3 received MsgVoteResp from 2 at term 7
raft2025/05/03 13:27:53 INFO: 3 has received 2 MsgVoteResp votes and 0 vote reje
ctions
raft2025/05/03 13:27:53 INFO: 3 became leader at term 7
raft2025/05/03 13:27:53 INFO: raft.node: 3 elected leader 3 at term 7
{"level":"error","msg":"send raft message","error":"Post \"http://172.19.0.11:80
80/raft\": read tcp 10.244.1.31:55764->172.19.0.11:8080: read: connection reset
by peer"}

```

Figure 3.17: Raft election after the forced removal of the leader in cloud-energy, cloud performance becomes leader at term 7

4. Runtime Optimization for Distributed Applications

4.1 Introduction

Up to this point, our discussions have focused on the deployment phase of Swarmchestrate, specifically taking the application-defined QoS requirements and ensuring that selected resources will meet those requirements. After deployment, the system will move into an operational phase in which the Swarmchestrate monitoring system (detailed in D2.1 Section 4) will collect key metrics, both infrastructure metrics commonly found in cloud, fog, and edge based systems as well as metrics defined by the applications at specification time.

At runtime, Swarmchestrate is tasked with ensuring that the application-defined QoS constraints are continuously maintained in the presence of changes due to the application itself, the swarm resources it is executing on, as well as other environmental changes. This is accomplished in two primary ways. First, the previously mentioned monitoring system, developed in WP2, collects, aggregates and evaluates metrics over time, *detecting* violations. In parallel, the component focused on here will employ AI techniques for *predicting* QoS violations.

Both detection and prediction systems will rely on an additional component, to be implemented later in this WP, to propose modifications to the application configuration that will either restore the violated application-defined QoS constraints or will pre-emptively modify the configuration so that a predicted violation does not occur.

This section describes the AI-based prediction system and provides an overview for the complete sub-system.

4.2 State of the Art

Developing distributed AI techniques for the runtime optimization of distributed applications necessitates reviewing current approaches in Federated Learning, Time Series analysis, and Reinforcement Learning. These learning techniques and paradigms offer promising mechanisms for decentralized intelligence, particularly in systems with constrained communication, privacy requirements, or dynamic topology, as in the case of the Swarmchestrate approach. A discussion of some of these techniques applied to resource allocation in cloud-edge computing environments is also provided in deliverable D2.1, Section 5.2.

Federated Learning (FL) is a machine learning technique in which multiple devices or entities (clients) collaboratively train a model without explicitly exchanging their local data. A defining characteristic of federated learning is data heterogeneity, as local datasets may have diverse distributions. FL offers an effective privacy-preserving and communication-efficient learning method for training models across decentralized agents,

which is particularly relevant in a swarm-based system like Swarmchestrator, where data is inherently distributed and potentially device-specific.

Originally introduced by McMahan et al. (2017) [29], the canonical FL setup consists of a central server and numerous edge clients; at each round, the server sends the current global model to a subset of randomly picked clients which locally train it on their own private data; then, each local model is sent back to the server where the global model is updated by averaging it with the received local models. The process continues until convergence. Since its introduction, numerous extensions to classical FL have emerged to address challenges such as data heterogeneity and unbalanced client participations.

Instead of randomly selecting the participating clients at each round, different participant selection mechanisms have been proposed to improve training efficiency in FL. For instance, Oort [30] proposed to prioritize the use of clients having both data that offers the greatest utility in improving model accuracy and the capability to run training quickly. This is particularly relevant in the cloud-to-edge scenario of Swarmchestrator, where clients may not all be available for FL training at a given time, and may have diverse data distributions and computational capabilities.

Another relevant research direction is personalized FL, where the goal is to allow each client to train a personalized model tailored to its specific needs while still benefiting from collaborative learning. For example, FedPer [31] proposes to partition the models into a shared base component and a client-specific personalization layer, which is kept private by each device. Another notable method is Ditto [32], which incorporates a regularization term in the local objectives to encourage the personalized models to remain close to the optimal global model while still allowing local adaptation. Personalized models are particularly useful in cloud-to-edge scenarios like Swarmchestrator, where devices running the same application may have very different environments and objectives.

Asynchronous FL frameworks like FedAsync [33] propose algorithms to handle scenarios where client updates arrive at different times, which is a common occurrence in real distributed setups. These techniques reduce idle times and consider the updates from late or infrequent contributors without degrading model performance.

FL has been successfully applied in the context of distributed runtime orchestration. For instance, FAuNO [34] presented a semi-asynchronous federated learning framework to optimize task offloading in edge computing scenarios. FAuNO employs an actor-critic Reinforcement Learning approach where local actors learn node-specific dynamics, while a federated critic aggregates experiences across agents to elicit cooperation and increase performance.

Time-Series (TS) analysis and forecasting encompass statistical and machine learning techniques for understanding and predicting patterns in series of data points. During runtime, each device in the Swarmchestrator system continuously produces monitoring data that reflects its current operational state. Understanding patterns in these time series and

predicting their future behaviour is particularly relevant for proactively reallocating resources or triggering swarm reconfiguration.

In recent years, the time series field has experienced significant advancements, moving from traditional statistical approaches to more advanced deep learning techniques. Liu et al. (2023) [35] introduced iTransformer, which repurposes the standard transformer architecture for the task of multivariate TS forecasting. Unlike previous transformer-based approaches, iTransformer inverts its input dimensions and embeds time points into *variate tokens* instead of temporal tokens. This enables the attention mechanisms to model inter-variate relationships and achieves state-of-the-art performance on real-world datasets, offering better generalization and efficiency, and making it a strong candidate for time series forecasting tasks.

Han et al. (2025) [36] proposed SOFTS, an efficient model for multivariate TS forecasting employing a novel STar Aggregate-Redistribute (STAR) module. STAR creates a global core representation by aggregating all input series, which is then redistributed and integrated with individual series representations to enable effective channel interactions. SOFTS achieves strong performance while maintaining linear computational complexity and offers an efficient alternative to transformer-based models.

Recent research has also introduced foundation models for TS forecasting, pre-trained on large TS datasets and applicable to diverse tasks and domains. However, such models often require substantial computational resources. To address this limitation, Ekambaram et al. (2025) [37] developed Tiny Time Mixers (TTM), a small and efficient pre-trained model based on the TSMixer [38] architecture, which is a stack of multi-layer perceptrons. TTM offers strong zero/few-shot capabilities while maintaining computational efficiency. The pre-trained weights of different model's versions are publicly available.

In the context of distributed applications, Ju et al. (2021) [39] proposed a Proactive Pod Autoscaler (PPA) that leverages a forecasting model to predict future workloads and proactively autoscale resources. Compared to the default Kubernetes autoscaler, this PPA approach achieves better efficiency of resource utilization and better application cloud performance. Moreover, its model-agnostic framework also allows the integration of newer, more accurate forecasting methods.

Reinforcement Learning (RL) is a machine learning framework where an agent learns to make intelligent decisions by interacting with an environment to maximize a cumulative reward. RL algorithms enable the agent to discover optimal policies through trial-and-error exploration and without requiring labeled data. This makes RL especially suitable for orchestrating systems in cloud and edge environments, where explicit supervision is often unavailable. Traditional single-agent RL methods such as Q-Learning, Deep Q-Networks (DQN) [33] and Proximal Policy Optimization (PPO) [40] have been successfully applied in resource allocation, task scheduling, and auto-scaling problems [41]. These methods rely on the definition of a reward function that encapsulates high-level QoS goals such as latency, throughput, or energy consumption.

However, the single-agent formulation assumes a global observation space and a central decision-making, which does not adapt well to the distributed nature of Swarmchestrator. These limitations motivate the use of **Multi-Agent Reinforcement Learning (MARL)** in a cooperative setting, where multiple agents jointly learn to achieve a common objective.

Early works in MARL adopted Independent Q-Learning (IQL) approaches [42], where each agent learns its own policy independently. While simple, IQL suffers from unstable training because the evolving policies of the agents result in a non-stationary environment. To overcome this, Foerster et al. (2017) [43] and Lowe et al. (2020) [44] operate in a Centralized Training with Decentralized Execution (CTDE) paradigm, where global information is used during training to facilitate learning. Their solutions are extensions of actor-critic policy gradient methods: the *critic* is provided with extra information, such as the policies or observations of other agents, while the *actor* relies solely on locally available data. After training, only the actors are deployed at runtime, ensuring that the system operates without centralized coordination.

Rashid et al. (2018) [45] proposed QMIX, a value-based solution in the CTDE setup that learns a joint Q-function estimating the common reward for the team of agents. This joint function is then decomposed into individual per-agent action-value functions, where each agent uses only its own local observation to infer its action.

MARL has been effectively applied to complex orchestration tasks in distributed computing contexts. For instance, Han et al. (2021) [46] employ a multi-agent actor-critic algorithm to improve the long-term throughput rate of request processing in edge-cloud systems. Similarly, Pei et al. (2024) [47] employ a Multi-Agent Deep Q-Network in a task scheduling framework to optimize both response times and operational costs. In their approach, each agent learns its own optimal policy, and training is stabilized through the technique of experience replay and the use of target networks. The above-discussed RL methodologies have the potential to be adapted to the Swarmchestrator runtime optimization process, particularly when the simulator implements digital twins of the demonstrators: the optimal reconfiguration policy can be learnt using simulations with verifiable rewards.

4.3 Problem Formulation and System Overview

Facilitating runtime reconfiguration requires knowing the application-level QoS goals and monitoring the state of the running system. The former arrives from the application specification itself while the latter is provided by the Swarmchestrator monitoring subsystem. When arriving data leads to a prediction of a violation of the QoS goals, this runtime reconfiguration subsystem will output a reconfiguration proposal to the orchestrator for its evaluation and eventual execution.

We envision three sub-components to the reconfiguration system, as seen in Figure 4.1. First, AI.1 continuously collects information from the monitoring system, processing it to predict a QoS violation. Following such a prediction, a second component AI.2 will propose modifications to the resource requirements to address the expected violation. If additional

resources are required, the requirements will be fed in AI.3 to the same mechanism used at deployment time to identify resources to be added to the swarm. The resulting reconfiguration plan will be returned to the orchestrator where it can be further evaluated and possibly enacted.

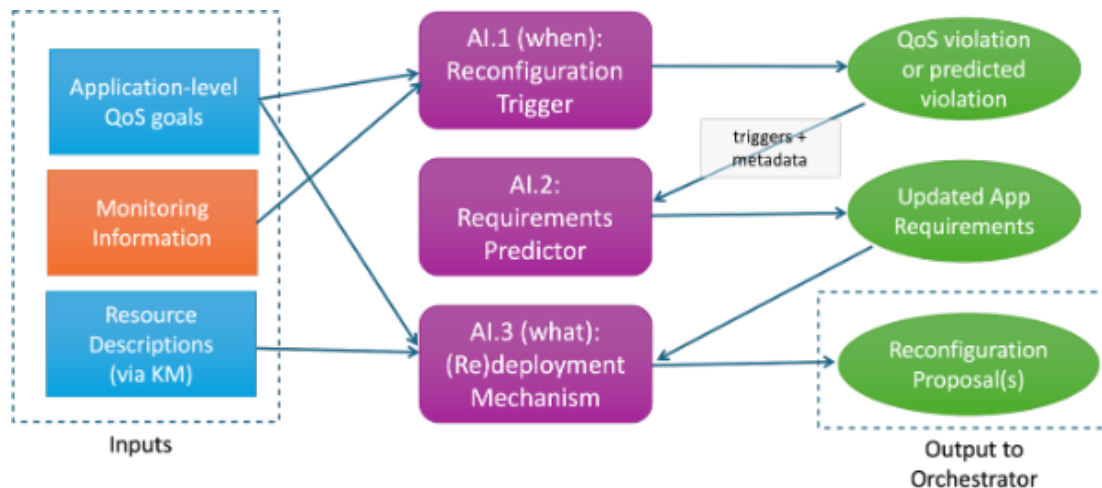


Figure 4.1: Overview of the reconfiguration system

4.3.1 AI.1 - Reconfiguration Trigger

The first step in this runtime reconfiguration subsystem is the prediction that a QoS parameter is likely to be violated in the near future. While the application level QoS requirements are static and easily identified based on the original application specification, the monitored information requires additional attention.

Specifically, we must consider *what* information and at *what frequency* to monitor specific attributes, as well as the location at which the monitored information should be collected and processed.

What to monitor: The Swarmchestrator monitoring system has the ability to monitor many infrastructure attributes of typical cloud, fog, and edge based resources, e.g., CPU, memory, latency, etc, and each at a configurable frequency. While our initial tendency is to collect “as much information as possible”, the overhead of doing so has the potential to negatively impact the functioning of the resources and thus the application components running on them. Therefore establishing the information and frequency at which to monitor is a critical element in our investigation.

Where to collect: The decision about where to collect and process the collected data is primarily one of evaluating the overhead imposed on the system in combination with the effectiveness of the solution. To clarify, we assume that a hypothetical, centralized prediction system using all information from all resources would provide the conditions to

make the “best” prediction possible. Nevertheless, such a system will incur an unreasonably large overhead both in terms of the data collection in a distributed system as well as the memory and processing time for a “large” centralized decision making algorithm. On the other hand, monitoring less information, e.g., on a single device, runs the risk of ignoring critical parameters that affect the QoS, especially latency. Therefore, we are exploring multiple options for where to collect, as well as the associated problem of “*where to process*”. To kick off the efforts, we start with the centralized prediction system where all information of resources can be gathered at a single processing resource to perform the prediction, serving as an indicative performance upper bound. We will then at a later phase investigate other more distributed information collection and prediction frameworks.

To set a baseline for the achievable performance of the prediction system, we will consider a centralized system that collects as much information as possible via the monitoring system. As this is unrealistic in the long run for an actual system, we will be using simulation as detailed in D4.2, as much as possible. Alternatively, it may be possible to utilize such a centralized system during the learning phase, but to reduce the amount of collected information during a subsequent, and much longer, runtime phase.

We have performed preliminary experiments to evaluate the feasibility of using time-series forecasting models to predict the future values of the monitored metrics. For this analysis, we considered the “Burn cpu Burn” dataset [48] which contains the CPU load data and various application-level metrics collected over a one-month period from a cluster of machines running multiple applications. We applied the iTransformer model [35] to forecast future CPU loads on a single machine, using historical time series data of all recorded variables along with time-based features. Our preliminary investigation shows the promise of the model, but the system requires additional modification to demonstrate the concrete results in the Swarmchestrator context. Moreover, the attention mechanism in the Transformer architecture produces an attention map over the input variates, which is a probability distribution indicating the model’s focus on each feature when predicting future CPU load. This map can be leveraged to rank the features by importance and consequently prune the less relevant ones to reduce the computational overhead. Finally, other typical baselines, such as Autoregressive Integrated Moving Average (ARIMA) or LSTM, will also be explored in our next phase study. Based on the positive results of this initial study, our plan is to apply those identified baselines more directly to the Swarmchestrator scenario in a testbed environment. We have begun evaluating the option of running simple, benchmark applications with predictable behavior in one of the testbed environments offered by the project partners. This will allow us to experiment with the solution both on a single device, which is a direct translation of the work to date, as well as employ the same mechanism in a centralized manner, collecting information about the running, distributed application and its resources across multiple hosts. This scenario will allow us to quantify the overhead of the distributed monitoring as well as evaluate the amount of training data necessary to achieve acceptable prediction performance.

In parallel we will explore the application of federated learning, specifically grouping resources with similar characteristics to share models. In this way we expect to reduce the

training time w.r.t. the single node solution, while achieving results closer aligned to a centralized solution. Further, the overhead of metric sharing will be reduced, making the solution more acceptable for long-term use.

In all cases, the centralized solution will serve as the benchmark for component performance.

4.3.2 AI.2 requirements prediction and AI.3 redeployment mechanism

After a QoS violation is predicted, the role of component AI.2 is to identify changes to the current resources that will allow the violation to be avoided. While specific mechanisms for this have not yet been explored, we envision using an application-defined rule-based system, e.g., to propose the addition or removal of resources depending on the violation. Of course, this is a suboptimal solution and, in the future, we plan to explore the option of introducing a learning-based approach for this component. Specifically, we envision that the application-defined scaling policy can be improved through Reinforcement Learning. This RL-based approach could rely either on the runtime exploration of the environment or on a simulated environment (*i.e.* digital twin in D4.1) to learn the optimal scaling strategy. While the feasibility of this approach has not yet been explored, we believe it represents a promising research direction worthy of further investigation.

The updated resource requirements predicted by AI.2 are then passed to component AI.3. This component is responsible for either removing resources or identifying new resources to accomplish the new requirements. While removing resources from the swarm is relatively straightforward, the allocation of new resources will utilize a mechanism similar to that used for resource identification at deployment time and described above, specifically the formulation and development of the decentralized solution where candidate swarm offers are formed among Resource Agents, and a dedicated Resource Agent will make the decision based on the cost accounting both the QoS requirements (associated with priority) and resource description (associated with reliability).

The refinement and integration of these components will continue through the end of the project.

5. Conclusions

This deliverable presented the foundational mechanisms developed within the Swarmchestrator project for intelligent, decentralized deployment of applications across the Cloud-to-Edge continuum. We built upon the key architectural components as given in D2.1 and D3.1, and detailed how these elements cooperate to realize a scalable and resilient orchestration framework.

A central contribution of this work is the concept of logical proximity, which enables resource selection and swarm formation based on multi-dimensional QoS requirements, including latency, bandwidth, energy consumption, cost, and reliability. By modeling logical proximity as a cost function and introducing both numeric and Borda-based ranking algorithms, we demonstrated how deployment decisions can be effectively decentralized while aligning with user-specified QoS goals.

To evaluate the feasibility and effectiveness of our matchmaking algorithms, we integrated them into a simulation environment (DISSECT-CF-Fog) and validated the selection and formation processes under realistic resource configurations. Our experiments confirmed that the proposed methods can accurately identify optimal resource groupings across heterogeneous cloud, fog, and edge environments, laying the groundwork for automated deployment with minimal manual intervention.

On the implementation side, we showcased the clean-architecture-based design of the orchestration components, including the stateless, pluggable scheduler and the fault-tolerant RLA consensus mechanism built on Raft. Our design choices, such as decoupling state management from consensus and isolating workload deployment via RAs, ensure robustness, modularity, and extensibility of the Swarmchestrator control plane.

To outline a path toward AI-assisted scheduling decisions, we explored runtime optimization in Section 4. We also presented the problem formulation and provided an overview of the reconfiguration system that will be used in future work to refine optimal scheduling decisions.

In summary, this deliverable establishes the algorithmic, architectural, and implementation groundwork for the Swarmchestrator orchestration system. Future deliverables will build upon this work by extending matchmaking to runtime reconfiguration, integrating trust and reliability models into scheduling decisions, and evaluating the full-stack system under diverse real-world scenarios.

6. References

- [1] A. Orive, A. Agirre, H.-L. Truong, I. Sarachaga, and M. Marcos, "Quality of Service Aware Orchestration for Cloud-Edge Continuum Applications," *Sensors (Basel)*, vol. 22, no. 5, Feb. 2022, doi: 10.3390/s22051755.
- [2] B. Mutichiro, M.-N. Tran, and Y.-H. Kim, "QoS-Based Service-Time Scheduling in the IoT-Edge Cloud," *Sensors*, vol. 21, no. 17, p. 5797, Aug. 2021.
- [3] S.-H. Kim and T. Kim, "Local Scheduling in KubeEdge-Based Edge Computing Environment," *Sensors*, vol. 23, no. 3, p. 1522, Jan. 2023.
- [4] S. Taherizadeh, V. Stankovski, and M. Grobelnik, "A Capillary Computing Architecture for Dynamic Internet of Things: Orchestration of Microservices from Edge Devices to Fog and Cloud Providers," *Sensors*, vol. 18, no. 9, p. 2938, Sep. 2018.
- [5] A. Ullah, H. Dagdeviren, R. C. Ariyattu, J. DesLauriers, T. Kiss, and J. Bowden, "MiCADO-Edge: Towards an Application-level Orchestrator for the Cloud-to-Edge Computing Continuum," *Journal of Grid Computing*, vol. 19, no. 4, pp. 1–28, Nov. 2021.
- [6] "MiCADO—Microservice-based Cloud Application-level Dynamic Orchestrator," *Future Generation Computer Systems*, vol. 94, pp. 937–946, May 2019.
- [7] K. Fu, W. Zhang, Q. Chen, D. Zeng, and M. Guo, "Adaptive resource efficient microservice deployment in cloud-edge continuum," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, no. 8, pp. 1825–1840, Aug. 2022.
- [8] R. Vaño, I. Lacalle, P. Sowiński, R. S-Julián, and C. E. Palau, "Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions," *Sensors*, vol. 23, no. 4, p. 2215, Feb. 2023.
- [9] G. Bartolomeo, S. Bäurle, N. Mohan, and J. Ott, "Oakestra: an orchestration framework for edge computing," in *Proceedings of the SIGCOMM '22 Poster and Demo Sessions*, pp. 34–36.
- [10] G. Bisicchia, S. Forti, E. Pimentel, and A. Brogi, "Continuous QoS-compliant Orchestration in the Cloud-Edge Continuum," Oct. 2023, doi: 10.1002/spe.3334.
- [11] "Cost-aware Service Placement and Scheduling in the Edge-Cloud Continuum," *ACM Transactions on Architecture and Code Optimization*, Mar. 2024, doi: 10.1145/3640823.
- [12] S. Moreschini, F. Pecorelli, X. Li, S. Naz, D. Hastbacka, and D. Taibi, "Cloud continuum: The definition," *IEEE Access*, vol. 10, pp. 131876–131886, 2022.
- [13] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," Sep. 28, 2011. doi: 10.6028/NIST.SP.800-145.
- [14] Chen, Songqing and Zhang, Tao and Shi, Weisong, "Fog Computing." Accessed: Jun. 02, 2025. [Online]. Available: <https://doi.org/10.1109/MIC.2017.39>
- [15] Apostu, A. and Puican, F. and Ularu, G. and Suciu, G. and Todoran, G., "Study on Advantages and Disadvantages of Cloud Computing - The Advantages of Telemetry Applications in the Cloud," Technische Informationsbibliothek (TIB). Accessed: Jun. 02, 2025. [Online]. Available: <https://www.tib.eu/en/search/id/BLCP%3ACN084898312/Study-on-Advantages-and-Disadvantages-of-Cloud/>
- [16] Michaela Iorga and Larry Feldman and Robert Barton and Michael Martin and Nedim Goren and Charif Mahmoudi, "Fog Computing Conceptual Model," Mar. 14, 2018. doi: 10.6028/NIST.SP.500-325.
- [17] P. Prakash, K. G. Darshaun, P. Yaazhylene, M. V. Ganesh, and B. Vasudha, "Fog Computing: Issues, Challenges and Future Directions," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 7, no. 6, pp. 3669–3673, Dec. 2017.
- [18] D. Bermbach et al., "A Research Perspective on Fog Computing," *Service-Oriented Computing – ICSOC 2017 Workshops*, pp. 198–210, 2018.
- [19] I. Stojmenovic, S. Wen, X. Huang, and H. Luan, "An overview of Fog computing and its security issues," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 10, pp. 2991–3005,

- Jul. 2016.
- [20] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge Computing: Vision and Challenges," *IEEE Internet Things J.*, vol. 3, no. 5, pp. 637–646, Oct. 2016.
 - [21] M. Abdoos, H. Taheri, and A. Taezadeh, "A proposed computation, which benefits from the cooperation of dew, edge, and cloud computations," *Transactions on Emerging Telecommunications Technologies*, vol. 31, no. 2, p. e3796, Feb. 2020.
 - [22] H. Zhang, Z. Zhang, L. Zhang, Y. Yang, Q. Kang, and D. Sun, "Object Tracking for a Smart City Using IoT and Edge Computing," *Sensors (Basel)*, vol. 19, no. 9, Apr. 2019, doi: 10.3390/s19091987.
 - [23] A. Jedidi, "Dynamic trust security approach for edge computing-based mobile IoT devices using artificial intelligence," *Eng. Res. Express*, vol. 6, no. 2, p. 025211, Jun. 2024.
 - [24] K. N. Mishra, V. Bhattacharjee, S. Saket, and S. P. Mishra, "Security provisions in smart edge computing devices using blockchain and machine learning algorithms: a novel approach," *Cluster Computing*, vol. 27, no. 1, pp. 27–52, Nov. 2022.
 - [25] "Borg, Omega, and Kubernetes," *Communications of the ACM*, Apr. 2016, doi: 10.1145/2890784.
 - [26] A. Ullah *et al.*, "Towards a Decentralised Application-Centric Orchestration Framework in the Cloud-Edge Continuum," Apr. 01, 2025. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/2504.00761>
 - [27] N. Poulton and P. Joglekar, *The Kubernetes Book - Second Edition*. Packt Publishing.
 - [28] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *2014 USENIX annual technical conference (USENIX ATC 14)*, 2014, pp. 305–319.
 - [29] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, and B. A. Y. Arcas, "Communication-efficient learning of deep networks from decentralized data," *AISTATS*, vol. 54, pp. 1273–1282, Feb. 2016.
 - [30] F. Lai, X. Zhu, H. V. Madhyastha, and M. Chowdhury, "Oort: Efficient federated learning via guided participant selection," *arXiv [cs.LG]*, Oct. 12, 2020. [Online]. Available: <http://arxiv.org/abs/2010.06081>
 - [31] M. G. Arivazhagan, V. Aggarwal, A. K. Singh, and S. Choudhary, "Federated learning with personalization layers," *arXiv [cs.LG]*, Dec. 02, 2019. Accessed: Jun. 04, 2025. [Online]. Available: <http://arxiv.org/abs/1912.00818>
 - [32] T. Li, S. Hu, A. Beirami, and V. Smith, "Ditto: Fair and robust federated learning through personalization," *arXiv [cs.LG]*, Dec. 08, 2020. Accessed: Jun. 04, 2025. [Online]. Available: <http://arxiv.org/abs/2012.04221>
 - [33] C. Xie, S. Koyejo, and I. Gupta, "Asynchronous federated optimization," *arXiv [cs.DC]*, Mar. 10, 2019. [Online]. Available: <http://arxiv.org/abs/1903.03934>
 - [34] F. Metelo, A. Oliveira, S. Racković, P. Á. Costa, and C. Soares, "FAuNO: Semi-Asynchronous Federated Reinforcement Learning Framework for Task Offloading in Edge Systems," *arXiv [cs.AI]*, Jun. 03, 2025. Accessed: Jun. 04, 2025. [Online]. Available: <http://arxiv.org/abs/2506.02668>
 - [35] Y. Liu *et al.*, "ITransformer: Inverted Transformers are effective for time series forecasting," *arXiv [cs.LG]*, Oct. 10, 2023. [Online]. Available: <http://arxiv.org/abs/2310.06625>
 - [36] L. Han, X.-Y. Chen, H.-J. Ye, and D.-C. Zhan, "SOFTS: Efficient multivariate time series forecasting with series-core fusion," *arXiv [cs.LG]*, Apr. 22, 2024. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/2404.14197>
 - [37] V. Ekambaram *et al.*, "Tiny Time Mixers (TTMs): Fast Pre-trained Models for Enhanced Zero/Few-Shot Forecasting of Multivariate Time Series," in *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, Nov. 2024. Accessed: Jun. 03, 2025. [Online]. Available: <https://openreview.net/forum?id=3O5YCEWETq¬eId=RdR2bNLzSY>
 - [38] S.-A. Chen, C.-L. Li, N. Yoder, S. O. Arik, and T. Pfister, "TSMixer: An all-MLP architecture for time series forecasting," *arXiv [cs.LG]*, Mar. 10, 2023. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/2303.06053>
 - [39] L. Ju, P. Singh, and S. Toor, "Proactive autoscaling for edge computing systems with Kubernetes,"

- arXiv [cs.DC]*, Dec. 19, 2021. doi: 10.1145/3492323.3495588.
- [40] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv [cs.LG]*, Jul. 19, 2017. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/1707.06347>
 - [41] G. Zhou, W. Tian, R. Buyya, R. Xue, and L. Song, "Deep reinforcement learning-based methods for resource scheduling in Cloud computing: A review and future directions," *arXiv [cs.DC]*, May 09, 2021. doi: 10.1007/S10462-024-10756-9.
 - [42] M. Tan, "Multi-agent reinforcement learning: Independent vs. Cooperative agents," in *Machine Learning Proceedings 1993*, Elsevier, 1993, pp. 330–337.
 - [43] J. Foerster, G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson, "Counterfactual Multi-Agent Policy Gradients," *arXiv [cs.AI]*, May 24, 2017. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/1705.08926>
 - [44] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," *arXiv [cs.LG]*, Jun. 07, 2017. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/1706.02275>
 - [45] T. Rashid, M. Samvelyan, C. S. de Witt, G. Farquhar, J. Foerster, and S. Whiteson, "QMIX: Monotonic value function factorisation for deep multi-agent reinforcement learning," *arXiv [cs.LG]*, Mar. 30, 2018. Accessed: Jun. 03, 2025. [Online]. Available: <http://arxiv.org/abs/1803.11485>
 - [46] Y. Han, S. Shen, X. Wang, S. Wang, and V. C. M. Leung, "Tailored learning-based scheduling for kubernetes-oriented edge-cloud system," *arXiv [cs.DC]*, Jan. 16, 2021. [Online]. Available: <http://arxiv.org/abs/2101.06582>
 - [47] L. Pei, C. Xu, X. Yin, and J. Zhang, "Multi-agent Deep Reinforcement Learning for cloud-based digital twins in power grid management," *J. Cloud Comput. Adv. Syst. Appl.*, vol. 13, no. 1, pp. 1–12, Oct. 2024.
 - [48] "Burn CPU Burn." Accessed: Jun. 04, 2025. [Online]. Available: <https://kaggle.com/model-t4>