QONNECT: A QoS-Aware Orchestration System for Distributed Kubernetes Clusters

Haci Ismail Aslan(✉), Syed Muhammad Mahmudul Haque, Joel Witzke, and Odej Kao

Technische Universität Berlin, Berlin, Germany
{aslan, joel.witzke, odej.kao}@tu-berlin.de,
haque@campus.tu-berlin.de

Abstract. Modern applications increasingly span across cloud, fog, and edge environments, demanding orchestration systems that can adapt to diverse deployment contexts while meeting Quality-of-Service (QoS) requirements. Standard Kubernetes schedulers do not account for user-defined objectives such as energy efficiency, cost optimization, and global performance, often leaving operators to make manual, cluster-by-cluster placement decisions. To address this need, we present QONNECT, a vendor-agnostic orchestration framework that enables declarative, QoS-driven application deployment across heterogeneous Kubernetes and K3s clusters. QONNECT introduces a distributed architecture composed of a central Knowledge Base, Raft-replicated Resource Lead Agents, and lightweight Resource Agents in each cluster. Through a minimal YAML-based interface, users specify high-level QoS goals, which the system translates into concrete placement and migration actions. Our implementation is evaluated on a federated testbed of up to nine cloud–fog–edge clusters using the Istio Bookinfo microservice application. The system demonstrates dynamic, policy-driven microservice placement, automated failover, QoS-compliant rescheduling, and leader re-election after node failure—all without manual intervention. By bridging the gap between declarative deployment models and operational QoS goals, QONNECT transforms the cloud–edge continuum into a unified, self-optimizing platform.

Keywords: Cloud-fog-edge continuum · Orchestration · Self-optimizing clusters · QoS-aware deployment · Microservice placement and migration.

1 Introduction

The cloud–edge continuum has emerged as a response to the varying latency, privacy, and availability requirements of modern distributed applications [14, 16]. While cloud computing offers elasticity, scalability, and cost efficiency, its centralized model remains insufficient for latency-sensitive or connectivity-constrained scenarios, particularly at the mobile and edge levels [2, 7]. Fog computing mitigates these limitations by introducing intermediate layers that bring compute

and storage closer to data sources [3, 7, 10], but managing large-scale, heterogeneous fog infrastructures poses challenges in energy efficiency, security, and fault tolerance [3, 20, 21]. Edge computing extends this trend further, enabling near-device processing to support real-time workloads, yet it introduces its own constraints in terms of capacity, reliability, and exposure to distributed attack surfaces [1, 11, 15].

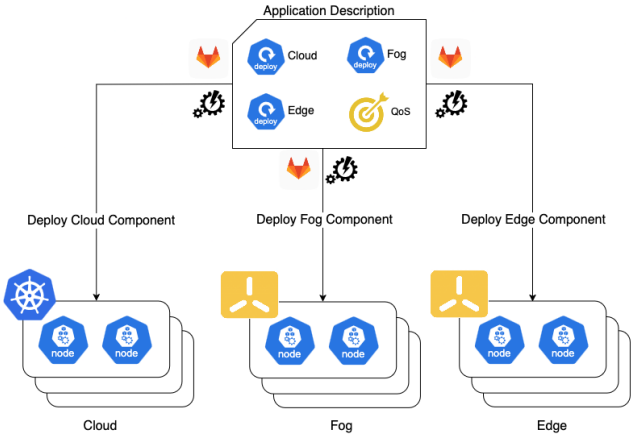


Fig. 1: Target approach.

Across the cloud–fog–edge continuum, no single model is universally optimal. Instead, hybrid approaches that coordinate centralized and decentralized resources offer better flexibility and performance [5]. Cloud-native orchestration tools like Kubernetes (K8s) have enabled scalable, infrastructure-agnostic application management in cloud environments [6]. Lightweight variants such as k3s and KubeEdge extend orchestration to fog and edge settings [25]. However, existing tools fall short in managing distributed, multi-layered applications with diverse QoS constraints [18, 23].

A robust cloud-to-edge orchestrator must meet seven key criteria [23]: (1) unified deployment across layers, (2) technology agnosticism, (3) multi-cloud capability, (4) dynamic runtime optimization, (5) extensible policy definitions, (6) support for both containers and virtual machines (VMs), and (7) production-grade robustness and security. Kubernetes partially satisfies these through autoscaling, extensibility, and VM integration, but lacks native support for QoS-aware scheduling and cross-cluster placement optimization.

While declarative tools like Helm and Kustomize simplify multi-service deployment, placing services based on latency, cost, or energy remains largely manual. The default Kubernetes scheduler is unaware of such QoS goals, requiring administrators to hardcode node placement and manually update CI/CD pipelines during initial deployment or disaster recovery [9].

To overcome these limitations, we propose QONNECT, an orchestration framework that automates the deployment of interconnected microservices across cloud, fog, and edge environments. Our solution aims to fulfill the key orchestrator requirements while minimizing manual intervention, guided by initiatives such as Swarmchestrate [13,24] and informed by real-world constraints. The envisioned approach is illustrated in Figure 1. We summarize our main contributions as follows:

- **Vendor-agnostic architecture.** A clean separation of global state (Knowledge Base), distributed decision-making (Raft-replicated RLAs), and local actuation (RAs) keeps the framework independent of cloud-provider APIs.
- **QoS-aware scheduler.** A weighted Borda count translates declarative vectors for energy, cost, and performance into concrete placements, requiring only a minimal YAML addition to existing manifests; the ranking component can be swapped for alternative algorithms.
- **Fault-tolerant control plane.** RLAs form a Raft quorum, automatically elect new leaders, and cooperate with RAs for transparent workload migration.

2 Background

2.1 Kubernetes and Lightweight Distributions

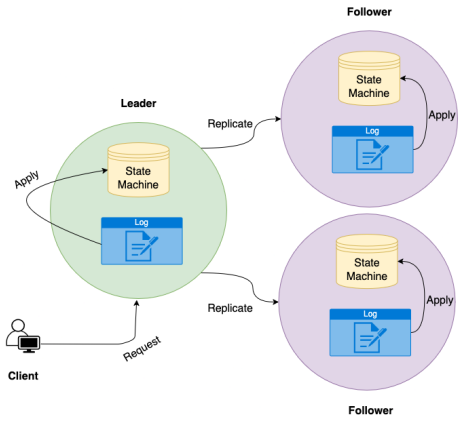
Kubernetes is the de facto standard for orchestrating containerized applications at scale. It decouples application deployment from infrastructure management using a declarative model driven by a central API server [19]. A typical cluster consists of a control plane (API server, scheduler, etcd, controller managers) and worker nodes that run containerized workloads grouped into pods. Deployments and ReplicaSets enable replication and lifecycle management. Kubernetes offers autoscaling, service discovery, self-healing, and declarative management of networking, storage, configuration, and secrets—supporting stateless, dynamic applications. To extend these features to resource-constrained environments, lightweight distributions like k3s bundle components and replace etcd with SQLite. Designed for edge and fog deployments, k3s supports Helm, CRDs, and runtimes like containerd, making it ideal for low-power or remote nodes.

2.2 Consensus and Orchestration

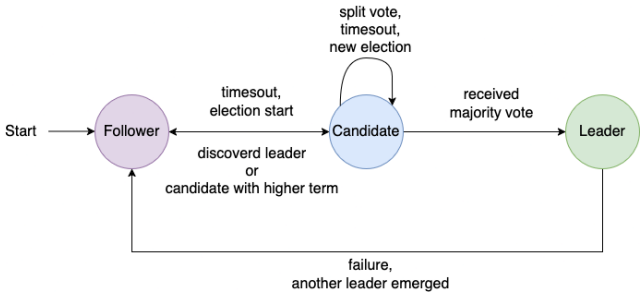
Reliable orchestration across distributed clusters requires strong consistency guarantees for a shared state. One commonly adopted [4,8] model is the *Replicated State Machine (RSM)*, where commands from clients are executed in the same order on all nodes. As shown in Figure 2a, this requires consistent log replication and fault tolerance to maintain synchronization across nodes.

Raft [17] is a consensus algorithm designed to implement RSMs with both understandability and practical deployment in mind. It simplifies consensus into three distinct components: *leader election*, *log replication*, and *safety*. A Raft

cluster consists of an odd number of nodes (e.g., 3 or 5), allowing the system to tolerate up to $\lfloor n/2 \rfloor$ node failures while maintaining quorum. Figure 2b illustrates the finite-state machine that governs each node in Raft. At any time, a node is in one of three states: *leader*, *follower*, or *candidate*. The leader handles client requests, appends them to its log, and replicates them to followers. Followers remain passive, responding only to messages from the leader or candidates.



(a) Replicated State Machine (RSM) model.



(b) Raft node states and transitions.

Fig. 2: Overview of Raft-based architecture components.

When a leader fails or becomes unreachable, followers won't receive its heartbeat and will consequently transition to candidates, starting a new election. A majority vote elects a new leader, who then resumes log replication. Log consistency is maintained by matching log indices and terms, ensuring that committed entries are preserved across leadership changes.

Raft's simplicity and robustness make it an ideal choice for managing distributed configuration states in orchestrators, particularly for multicluster coordination, leader tracking, or implementing consistent control logic across cloud-fog-edge environments.

2.3 QoS-Aware Scheduling and Borda Count

Traditional Kubernetes schedulers assign pods based on resource availability and node affinity, but lack native support for QoS-based policies (e.g., latency, cost, energy). To introduce such logic, external schedulers or controllers can rank candidate nodes using preference models.

One such model is the *Borda Count*, a voting method where each alternative (e.g., node) is assigned a score based on ranked preferences from multiple criteria. This allows fine-grained, weighted decisions for scheduling microservices with complex placement needs.

3 Architecture

OONNECT is a distributed orchestration framework designed to enable Quality-of-Service (QoS)-aware application deployment across heterogeneous cloud, fog, and edge environments. It abstracts away the complexity of coordinating multiple Kubernetes and K3s clusters, allowing operators and developers to specify high-level QoS goals through declarative policies. The system autonomously handles microservice placement, migration, and failover, adapting to dynamic infrastructure conditions while maintaining compliance with QoS constraints.

3.1 Requirements

The primary objective is to facilitate the cross-domain deployment and management of application components based on the user's QoS goals. This includes energy efficiency, cost and performance considerations (e.g., CPU, memory, storage, and bandwidth). Furthermore, the system must be capable of redeploying components upon failures or disruptions, ensuring that user applications continue to run uninterrupted. In summary, the framework must provide functional (FR) and non-functional (NFR) requirements such as:

- Flexible, user-directed domain placement (e.g., cloud, fog, or edge) without requiring manual setup.
- Adherence to user-defined goals for energy efficiency, cost, and application performance.
- Redeployment mechanisms when disruptions occur, maintaining the same QoS standards.
- Overall fault tolerance to handle unexpected events or resource contention.

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 .

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 based on operational and QoS requirements. The actual scheduling and instantiation of components in the user-selected domain shall be handled swiftly and transparently.

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:

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

Disruption Recovery and Redeployment (FR3) 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:

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

Fault Tolerance (NFR1) The framework shall exhibit fault-tolerant behavior, continuing to operate despite failures in hardware, software, or network resources. This involves employing backup instances to mitigate single points of failure.

3.2 Concept

The QONNECT architecture is influenced by the principles of *Clean Architecture*[1], which promote a clear separation of concerns across domain logic, use cases, and interface boundaries. This design choice enhances modularity, testability, and portability across diverse deployment environments. As illustrated in Figure 3, QONNECT consists of three core components: the *Knowledge Base (KB)*, the *Resource Lead Agent (RLA)*, and the *Resource Agent (RA)*—each fulfilling distinct roles in orchestrating services across distributed domains.

Each domain manages its own cluster(s). Cloud clusters typically run full Kubernetes; edge clusters use lightweight k3s; and fog clusters may use either, depending on available resources. Clusters consist of a control-plane and worker nodes. Worker nodes host application components and are monitored by a cluster-local RA, while at least one cluster on cloud includes an RLA to enable leadership and coordination.

The *Knowledge Base (KB)* acts as a global metadata repository for all clusters, nodes, and application deployments. It holds information such as available resources, node health, workload status, and user QoS requirements. While logically centralized, the KB can be backed by any reliable distributed storage (e.g.,

[1] https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html; last accessed: October 14, 2025

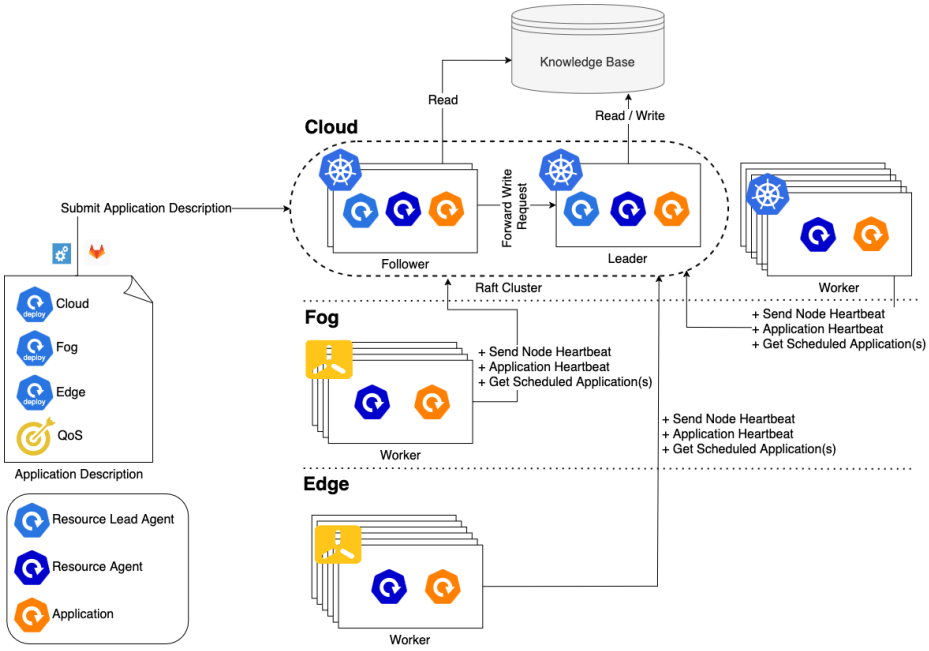


Fig. 3: Architecture overview: component interaction across cloud, fog, and edge domains.

replicated SQL/NoSQL databases or blockchain-based solutions), ensuring resilience and fault tolerance.

The *Resource Lead Agent (RLA)* is responsible for cluster coordination, system-wide scheduling, and consensus management. RLAs run on selected nodes and collectively form a Raft quorum for leader election and consistent decision-making. The elected RLA leader becomes the system's API entry point, serving orchestration requests and issuing deployment decisions based on user goals (e.g., minimizing cost, reducing latency, or conserving energy). The RLA evaluates all registered clusters against user-defined QoS constraints and dynamically selects the most suitable ones.

The *Resource Agent (RA)* is deployed in every application-hosting cluster. It interacts with the local Kubernetes or k3s API to receive and apply deployment instructions issued by the RLA. The RA also collects real-time telemetry from the cluster—covering node status, application health, and resource utilization—and forwards this data to the RLA. This ensures the orchestration loop remains aware of environmental changes and can trigger redeployments when necessary.

Each RA is stateless and reactive, ensuring fault-tolerant behavior with minimal footprint. RAs also act as execution proxies, translating higher-level scheduling decisions into native Kubernetes manifests and ensuring that inter-service

communication and dependency configurations are honored locally. The entire system is designed to support QoS-aware deployments while ensuring resilience and minimal manual intervention. If a cluster or node becomes unavailable, the RLA detects the failure and initiates a migration of the affected application to another suitable cluster within the same or a different domain, maintaining compliance with user goals.

By combining centralized decision-making (via the KB and RLA leader) with decentralized enforcement (via RAs), the architecture achieves scalable, cross-domain orchestration suited to dynamic, resource-constrained environments across the cloud–fog–edge continuum.

3.3 Implementation

This section details our system's implementation, justifying the claimed achievements. Our implementation[2] is open-sourced anonymously.

Raft Instead of a custom Raft implementation, we forked 'etcd/contrib/raftexample', retaining its bare-bones consensus layer (state machine, Write Ahead Log (WAL), snapshot) and replacing its key-value store/transport with a minimalist HTTP-based REST transport.

Scheduler The leader RLA in a Raft cluster serves the scheduler and the scheduler periodically:

1. Places pending services by assigning clusters/nodes using the Borda scorer, persisting decisions in the KB.
2. Re-queues stalled services; workloads with heartbeats older than a grace period are reset.

The Borda scorer's 'ScoreAndFilterNodes' method receives a node snapshot and QoS, proceeding via these steps:

1. Eligibility filter: Removes unready, unschedulable, or resource-pressured nodes; an empty result stops processing.
2. Per-attribute Borda ranking: Sorts slices and assigns Borda scores for energy, pricing, and capacity (CPU, memory, bandwidth, storage); lower values win for energy/pricing, higher for performance.
3. Capacity factor: Sums the four capacity Bordas into a single 'Capacity-Borda' representing raw node performance.
4. QoS-weighted node score: Calculated as

$$w = Energy\_\text{Borda} \cdot q_E + Pricing\_\text{Borda} \cdot q_P + Capacity\_\text{Borda} \cdot q_C,$$

where (q_E, q_P, q_C) are user weights (energy, pricing, performance), defaulting to $(1, 1, 1)$ if all are 0.0.

[2] https://github.com/dos-group/QONNECT; last-accessed: October 14, 2025

5. Threshold filter: Keeps nodes with score w greater than or equal to the arithmetic mean of all w values.
6. Cluster aggregation: Sums node scores per cluster, ordering clusters by retained score (total score as tie-breaker).
7. Return value: ID of the highest-ranked cluster and retained node names within it.

REST API Manifest portability across domains is enabled by two conventions: (1) Each domain manifest requires an Ingress with the application name as its first path segment, and (2) cross-domain placeholders (IP addresses for cloud-fog-edge clusters) are used instead of hard-coded addresses, which are replaced by the RA with concrete cluster IPs before manifest application. The 'application' field is a thin CRD-style YAML for scheduler-required metadata (name, labels, QoS).

Resource Agent The RA, which also follows the Clean Architecture principles, is responsible for the orchestration steps described in the following subsections.

Registration On startup, the RA checks for prior registration via the 'self' ConfigMap in the namespace. If present, cluster ID and role are loaded; otherwise, the cluster joins for the first time by determining its external IP, submitting it to the bootstrap RLA, and caching the returned UUID in a new ConfigMap. During bootstrap, auxiliary ConfigMaps for cluster IP mapping and running applications are also verified for subsequent operations.

Send node snapshot The RA uses the Kubernetes API to enumerate, filter control-plane nodes, and send the node snapshot to the RLA.

Polls cluster config The RA periodically fetches cluster ID $\rightarrow$ ingress IP mapping from the RLA via REST, updating the local ConfigMap for application deployment.

Polls applications The RA retrieves recently scheduled workloads from the RLA's dedicated REST endpoint. For each, it: (1) ensures/creates the target namespace, (2) applies Kubernetes objects, merging labels, and replacing cross-domain placeholders with concrete IPs from the cluster config, and (3) restricts deployments to scheduler-chosen worker nodes. On successful deployment, the RA adds/updates the applications ConfigMap with application ID, version, applied objects, and timestamp, serving as the single source of truth for health checks and documenting assigned workloads.

Send application heartbeat For each application in the applications ConfigMap, the RA reports status to the RLA: (1) healthy (all Deployments reached desired replica count), (2) progressing (at least one Deployment rolling out, within timeout), or (3) failed (e.g., crash-loop, missing objects, timeout, requiring manual intervention).

Cleanup application If the RLA responds to an application heartbeat with '404 Not Found', the RA concludes workload withdrawal, immediately deleting the Kubernetes namespace (removing deployed objects) and its ConfigMap entry, preventing future heartbeat cycles.

4 Evaluation

4.1 Setup

For evaluation, we established a testbed based on Kind[3] comprising nine clusters: three each for cloud, fog, and edge. Within each domain, clusters were assigned one of three profiles: energy-efficient, cost-efficient, or performance-efficient. Each cluster consists of one control-plane and two worker nodes. As clusters run in Docker on a single machine, real telemetry is unavailable; thus, representative energy, cost, and bandwidth values were manually assigned based on public data.

Energy Consumption Energy consumption was calculated using coefficients from Cloud Carbon Footprint's Appendix I[4] and the formula: $E = ((W_{\text{idle}} + W_{\text{max}})/2) \times \text{PUE} \times 1h/1000$, where PUE is power usage effectiveness and unitless. This yielded $E_{\text{AWS}} \approx 0.0024042$ kWh and $E_{\text{GCP}} \approx 0.0027335$ kWh. We assigned the lowest, midpoint ($E_{\text{mid}} \approx 0.0025689$ kWh), and highest values to energy-efficient, cost-efficient, and performance-efficient profiles, respectively.

Cost (EUR/h) Cost values, based on AWS On-Demand pricing[5] (assuming 1 EUR = 1 USD), were set as: $C_{\text{min}} = 0.0042$ EUR/h (`t4g.nano`) and $C_{\text{max}} = 32.7726$ EUR/h (`p4d.24xlarge`). These were assigned to cost-efficient, energy-efficient ($C_{\text{mid}} \approx 16.3884$ EUR/h), and performance-efficient profiles, respectively.

Network Bandwidth Network bandwidth values were derived from AWS specs (5 Gbps for `t4g.nano`, capped at 100 Gbps for `p4d.24xlarge` from 400 Gbps). We assigned 5 Gbps to cost-efficient, 52.5 Gbps (B_{mid}) to energy-efficient, and 100 Gbps to performance-efficient profiles.

The assigned parameters for each cluster profile are summarized in Table 1. We applied these same values to fog and edge clusters for consistency, acknowledging that real-world values would differ but maintaining clear distinctions across profiles. To simulate energy- and cost-efficient characteristics, we manually reduced reserved system resources on their worker nodes (5 GiB memory, 4 CPU cores, 2 GiB ephemeral storage) via `kubeadmConfigPatches` during setup.

[3] https://kind.sigs.k8s.io; last-accessed: October 14, 2025

[4] https://www.cloudcarbonfootprint.org/docs/methodology/#appendix-i-energy-coefficients

[5] https://aws.amazon.com/de/ec2/pricing/on-demand/ (Location Type: AWS Region, Region: US East (Ohio), OS: Linux)

Table 1: Assigned energy, cost, and bandwidth parameters for each cluster profile.

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

Test application For system evaluation, we adapted the Istio Bookinfo microservices application[6], a multi-service e-commerce system, with minimal configuration modifications and added a Kubernetes Ingress. The Bookinfo application comprises `productpage`, `details`, `reviews`, and `ratings` services. To simulate a cloud–fog–edge architecture, we deployed `productpage` on the cloud cluster, `details` and `reviews` on the fog cluster, and `ratings` on the edge cluster. Our deployment retained the overall structure but redistributed services to align with our experimental setup.

4.2 Results

To test the systems requirements we conducted a series of tests using the developed QONNECT prototype:

1. Deploy the test application prioritizing performance. Here QONNECT should schedule the application's components to the 'performance' clusters in each domain.
2. After initial deployment, we change the applications QoS requirement to prioritize energy efficiency instead. The components of the application should be redeployed to the corresponding 'energy' clusters.
3. After initial deployment, we simulate a cluster failing by deleting the RA within the 'performance' cluster of the edge domain. QONNECT should react accordingly be redeploying the ratings component to a different cluster within the edge domain.
4. Test the resilience against lead agent failure by deleting the RLA currently acting as raft leader. In our case that was the RLA of the 'energy' cluster.

Figure 4 shows the anticipated—and achieved—outcomes from executing the tests 1, 2 and 3. Test 4 also finished successfully, electing the RLA in the 'performance' cluster as new leader. Scripts conducting these tests can be found in the same repository together with our QONNECT prototype[7].

In addition to experimental validation, we provide a comparative analysis of QONNECT against existing orchestration frameworks in terms of architectural and functional capabilities. As summarized in Table 2, QONNECT combines QoS-aware scheduling, cloud–edge continuum support, distributed scheduling,

[6] https://istio.io/latest/docs/examples/bookinfo/noistio.svg; last-accessed: October 14, 2025

[7] https://github.com/dos-group/QONNECT/blob/main/resource-lead-agent/bruno-collection/swarmchestrate.json; last-accessed: October 14, 2025

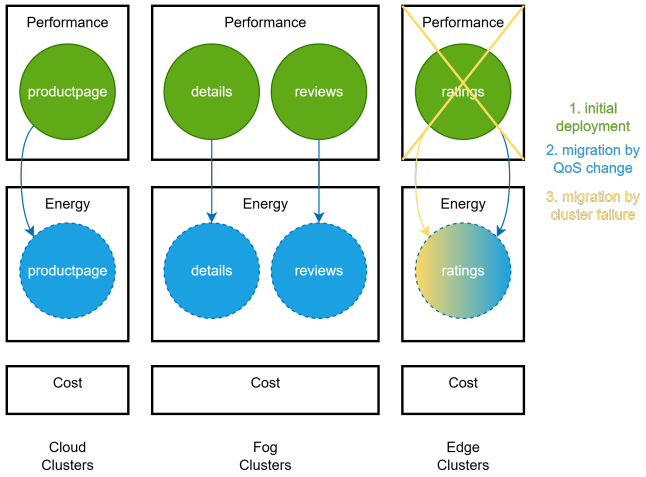


Fig. 4: QONNECT prototype behavior in tests 1 to 3. The color specifies the corresponding test case.

and fault tolerance within a unified framework. This highlights its potential to address limitations observed in prior systems and demonstrates its practical viability in diverse deployment environments.

5 Related Work

Several orchestration frameworks have emerged to address deployment and QoS challenges across the cloud–edge continuum. Taherizadeh et al. [22] propose a capillary computing model that migrates Docker-based microservices across Edge, Fog, and Cloud tiers to maintain proximity to mobile clients and balance load. Their vehicle-mounted prototype demonstrates significant improvements in response time and latency variation. MiCADO-Edge [12, 23] extends MiCADO with KubeEdge to orchestrate containers and VMs across heterogeneous infrastructures using TOSCA descriptors, supporting dynamic policy enforcement and multi-platform deployments in domains such as healthcare and video analytics. ACOA [18] introduces a distributed, per-application multi-scheduler architecture built on Kubernetes, enabling QoS-aware placement through customizable scheduling algorithms and consideration of latency, reliability, and resource availability. Nautilus [9] addresses dynamic runtime conditions through RL-based resource management, microservice migration, and latency-aware mapping of service graphs, achieving both QoS guarantees and resource efficiency. Complementing these system-specific efforts, Vaño et al. [25] survey the evolution of cloud-native orchestration at the edge, identifying trends in lightweight Kubernetes distributions, container runtimes, and emerging technologies such as WebAssembly-based virtualization.

Table 2: Comparison of orchestration systems.

Feature	QONNECT	MiCADO	ACOA	Nautilus	K8s (Vanilla)
Cloud–Edge	✔ Full support	✗ Cloud-focused; limited edge	✔ Full support	→ HPC-oriented	→ Not continuum-aware
QoS-aware scheduling	✔ Supports with self-healing	✗ No QoS placement	✔ QoS-based placement	→ Resource-focused; lacks QoS logic	→ No built-in constraint
Distributed scheduling	✔ Raft-based with RAs	✗ Terraform & policy keeper	✔ Multi-scheduler model	→ K8s default control	→ Single control plane logic
VM & container autoscaling	✔ VM and container aware	✔ Supports via Terraform & K8s	→ Focus on placement, not scaling	→ Scaling by base scheduler	✔ HPA and Cluster Autoscaler
Self-healing and failover	✔ Leader election, redeployment	→ Container restarts, no migration	→ Failover not emphasized	✔ Pod recovery, monitoring	✔ Supports via add-ons
User-level QoS specification	✔ Declarative YAML with QoS goals	→ Metric triggers in TOSCA	✔ Supports developer-level logic	→ No user QoS interface	→ Only resource requests/limits
Prototype / Testbed	✔ K8s & K3s clusters with Istio app	✔ Validated across cloud platforms	✔ Used in railway use-case	✔ Deployed on GPU clusters	✔ Widespread

6 Conclusion and Future Work

This work introduced QONNECT, a QoS-driven orchestrator for heterogeneous Kubernetes and K3s clusters spanning cloud to edge environments. The system enables intent-based deployment by translating high-level QoS vectors into concrete placement and migration decisions, using a modular scheduler architecture and a fault-tolerant Raft-based control plane. Through a federated nine-cluster testbed, QONNECT demonstrated its ability to maintain service-level objectives across dynamic conditions, automatically recovering from node and agent failures while preserving quorum and consistency. By demonstrating reliable, policy-driven orchestration under dynamic conditions, this work takes a step toward invisible infrastructure—where developers express what they need, and the system decides where and how to run it.

While promising, the prototype was evaluated under synthetic conditions. Future work will extend to large-scale, real-world deployments with richer QoS dimensions—especially latency—and integrate telemetry-driven energy models and predictive migration strategies.

Acknowledgments. This paper was partially supported by the Swarmchestrate project of the European Union's Horizon 2023 Research and Innovation programme under grant agreement no. 101135012.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abdoos, M., Taheri, H., Taeezadeh, A.: A proposed computation, which benefits from the cooperation of dew, edge, and cloud computations. Transactions on Emerging Telecommunications Technologies **31**(2), e3796 (2020). https://doi.org/10.1002/ett.3796
2. Apostu, A., Puican, F., Ularu, G., Suciu, G., Todoran, G.: Study on advantages and disadvantages of cloud computing - the advantages of telemetry applications in the cloud. In: Applied computer science. pp. 118–123. WSEAS Press, Greece (2013)
3. Bermbach, D., Pallas, F., Pérez, D.G., Plebani, P., Anderson, M., Kat, R., Tai, S.: A research perspective on fog computing. In: Braubach, L., Murillo, J.M., Kaviani, N., Lama, M., Burgueño, L., Moha, N., Oriol, M. (eds.) Service-Oriented Computing – ICSOC 2017 Workshops. pp. 198–210. Springer International Publishing, Cham (2018)
4. Beyer, B., Jones, C., Petoff, J., Murphy, N.R.: Site reliability engineering: how Google runs production systems. " O'Reilly Media, Inc." (2016)
5. Brogi, A., Forti, S., Ibrahim, A.: How to best deploy your fog applications, probably. In: 2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC). pp. 105–114 (2017). https://doi.org/10.1109/ICFEC.2017.8
6. Burns, B., Grant, B., Oppenheimer, D., Brewer, E., Wilkes, J.: Borg, omega, and kubernetes. Commun. ACM **59**(5), 50–57 (Apr 2016). https://doi.org/10.1145/2890784
7. Chen, S., Zhang, T., Shi, W.: Fog computing. IEEE Internet Computing **21**(2), 4–6 (Mar 2017). https://doi.org/10.1109/MIC.2017.39
8. Enes, V., Baquero, C., Rezende, T.F., Gotsman, A., Perrin, M., Sutra, P.: State-machine replication for planet-scale systems. In: Proceedings of the Fifteenth European Conference on Computer Systems. EuroSys '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3342195.3387543, https://doi.org/10.1145/3342195.3387543
9. Fu, K., Zhang, W., Chen, Q., Zeng, D., Guo, M.: Adaptive resource efficient microservice deployment in cloud-edge continuum. IEEE Transactions on Parallel and Distributed Systems **33**(8), 1825–1840 (2022). https://doi.org/10.1109/TPDS.2021.3128037
10. Iorga, M., Feldman, L., Barton, R., Martin, M., Goren, N., Mahmoudi, C.: Fog computing conceptual model (2018-03-14 2018). https://doi.org/10.6028/NIST.SP.500-325
11. Jedidi, A.: Dynamic trust security approach for edge computing-based mobile iot devices using artificial intelligence. Engineering Research Express **6**(2), 025211 (may 2024). https://doi.org/10.1088/2631-8695/ad43b5

12. Kiss, T., Kacsuk, P., Kovacs, J., Rakoczi, B., Hajnal, A., Farkas, A., Gesmier, G., Terstyanszky, G.: Micado—microservice-based cloud application-level dynamic orchestrator. Future Generation Computer Systems **94**, 937–946 (2019). https://doi.org/https://doi.org/10.1016/j.future.2017.09.050, https://www.sciencedirect.com/science/article/pii/S0167739X17310506
13. Kiss, T., Ullah, A., Terstyanszky, G., Kao, O., Becker, S., Verginadis, Y., Michalas, A., Stankovski, V., Kertesz, A., Ricci, E., Altmann, J., Egger, B., Tusa, F., Kovacs, J., Lovas, R.: Swarmchestrate: Towards a fully decentralised framework for orchestrating applications in the cloud-to-edge continuum. In: Barolli, L. (ed.) Advanced Information Networking and Applications. pp. 89–100. Springer Nature Switzerland, Cham (2024)
14. Mell, P., Grance, T.: The nist definition of cloud computing (2011-09-28 2011). https://doi.org/10.6028/NIST.SP.800-145
15. Mishra, K.N., Bhattacharjee, V., Saket, S., Mishra, S.P.: Security provisions in smart edge computing devices using blockchain and machine learning algorithms: a novel approach. Cluster Computing **27**(1), 27–52 (Nov 2022). https://doi.org/10.1007/s10586-022-03813-x
16. Moreschini, S., Pecorelli, F., Li, X., Naz, S., Hästbacka, D., Taibi, D.: Cloud continuum: The definition. IEEE Access **10**, 131876–131886 (2022). https://doi.org/10.1109/ACCESS.2022.3229185
17. Ongaro, D., Ousterhout, J.: In search of an understandable consensus algorithm. In: 2014 USENIX annual technical conference (USENIX ATC 14). pp. 305–319 (2014)
18. Orive, A., Agirre, A., Truong, H.L., Sarachaga, I., Marcos, M.: Quality of service aware orchestration for cloud–edge continuum applications. Sensors **22**(5) (2022). https://doi.org/10.3390/s22051755
19. Poulton, N.: The kubernetes book. Lean Publishing (2021)
20. Prakash, P., Darshaun, K., Yaazhlene, P., Ganesh, M.V., Vasudha, B.: Fog computing: issues, challenges and future directions. International Journal of Electrical and Computer Engineering **7**(6), 3669 (2017)
21. Stojmenovic, I., Wen, S., Huang, X., Luan, H.: An overview of fog computing and its security issues. Concurr. Comput.: Pract. Exper. **28**(10), 2991—-3005 (Jul 2016). https://doi.org/10.1002/cpe.3485
22. Taherizadeh, S., Stankovski, V., Grobelnik, M.: A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers. Sensors **18**(9) (2018). https://doi.org/10.3390/s18092938, https://www.mdpi.com/1424-8220/18/9/2938
23. Ullah, A., Dagdeviren, H., Ariyattu, R.C., DesLauriers, J., Kiss, T., Bowden, J.: Micado-edge: Towards an application-level orchestrator for the cloud-to-edge computing continuum. Journal of Grid Computing **19**(4), 47 (2021). https://doi.org/10.1007/s10723-021-09589-5
24. Ullah, A., Markus, A., Aslan, H.I., Kiss, T., Kovacs, J., Deslauriers, J., Murphy, A.L., Wang, Y., Kao, O.: Towards a decentralised application-centric orchestration framework in the cloud-edge continuum. In: 2025 IEEE 9th International Conference on Fog and Edge Computing (ICFEC). pp. 37–41 (2025). https://doi.org/10.1109/ICFEC65699.2025.00014
25. Vaño, R., Lacalle, I., Sowiński, P., S-Julián, R., Palau, C.E.: Cloud-native workload orchestration at the edge: A deployment review and future directions. Sensors **23**(4) (2023). https://doi.org/10.3390/s23042215