



Research on Lifecycle Configuration and Reclamation Strategies for Edge Nodes Based on Microservice Architectures

Hangjie Zheng

GCP NetInfra, Google, Inc., Sunnyvale, CA 94089, USA.

How to cite this paper: Hangjie Zheng. (2025) Research on Lifecycle Configuration and Reclamation Strategies for Edge Nodes Based on Microservice Architectures. *Advances in Computer and Communication*, 6(5), 286-292.
DOI: 10.26855/acc.2025.12.005

Received: October 23, 2025

Accepted: November 21, 2025

Published: December 18, 2025

***Corresponding author:** Hangjie Zheng, GCP NetInfra, Google, Inc., Sunnyvale, CA 94089, USA.

Abstract

Frequent state transitions, transient connectivity, and heterogeneous service deployments in edge computing environments have rendered traditional static resource management approaches increasingly inadequate. The decentralized and latency-sensitive nature of edge systems demands lifecycle-aware mechanisms that can dynamically respond to workload fluctuations, service drift, and constrained resource availability. Microservice architectures—characterized by modularity, lightweight orchestration, and autonomous scalability—provide structural affordances to address these challenges. This study proposes a closed-loop lifecycle strategy tailored to edge nodes, encompassing both operational configuration and decommissioning phases. The framework integrates runtime-driven service composition, container-level resource quota tuning using HPA/VPA policies, and snapshot-based state externalization for node deactivation, combined with lightweight cross-node service migration. A resource reclamation mechanism is further introduced based on priority-weighted scoring, enabling fine-grained reclamation scheduling in high-concurrency environments. The proposed approach enhances system resilience, reduces resource waste, and ensures service continuity in volatile edge scenarios. Evaluation suggests its applicability to real-world deployments in distributed edge clusters with dynamic topologies and uneven resource distributions.

Keywords

Microservice architecture; Edge nodes; Lifecycle configuration; Resource reclamation

Edge nodes have emerged as critical enablers of low-latency computation and localized intelligence, yet their operational states remain highly volatile due to fluctuating workloads, device-level constraints, and environmental instability. Traditional centralized configuration models fail to account for these dynamic lifecycle variations, often leading to misallocated resources, service disruptions, and overall system inefficiency. In contrast, microservice-based architectures provide the modularity and runtime elasticity needed to respond to such volatility. The challenge lies in synchronizing service deployment and resource provisioning during active phases, while ensuring safe service teardown and seamless recovery during node retirement. Addressing both ends of the lifecycle is essential for robust edge system performance.

1. Lifecycle Modeling and Management Requirements for Edge Nodes Based on Microservice Architectures

1.1 Fine-Grained Lifecycle Staging and State Transition Logic of Edge Nodes

In real-world deployments, edge nodes exhibit distinct lifecycle phases that evolve dynamically over time [1]. These can be categorized into six primary states: Cold Start, Initializing, Running, Idle, Draining, and Terminated. Each

phase corresponds to a specific operational responsibility and resource status. For instance, Cold Start involves image pulling and network initialization; Running handles active request processing; Idle maintains baseline connections with minimal resource consumption. Transitions between states are triggered by runtime metrics and system feedback—for example, a node may enter Idle after maintaining sub-5% CPU utilization and zero active sessions for 10 consecutive minutes. Likewise, if a node is scheduled for retirement or becomes unresponsive, it should enter the Draining phase to gracefully offload services. Under microservice-based architectures, the entire lifecycle must be modeled as a finite-state machine integrated with the scheduler, enabling fine-grained orchestration and preventing resource waste or service drift.

1.2 Workload Patterns and Resource Signatures Across Lifecycle Stages

Edge nodes exhibit highly differentiated workload profiles and resource consumption behaviors across lifecycle stages. These variations span CPU load, memory usage, bandwidth demand, and disk I/O operations. For example, during Cold Start, network throughput and storage I/O peak momentarily due to image pulling and dependency setup, while CPU usage remains minimal. In contrast, the Running phase involves sustained CPU and memory consumption due to concurrent request processing, with consistently moderate-to-high bandwidth use. The Idle state demands minimal resources, maintaining only basic listeners and health probes. Draining and Terminated phases involve service state persistence and log flushing, resulting in elevated I/O pressure. These evolving resource signatures serve as critical inputs for orchestration decisions. A multi-dimensional metric model—such as a CPU-I/O-Bandwidth triangular mapping (see Figure 1)—can guide service mounting and configuration choices in real time. Lifecycle-aware modeling of such resource characteristics improves scheduling precision and prevents reactive lag during load fluctuations.

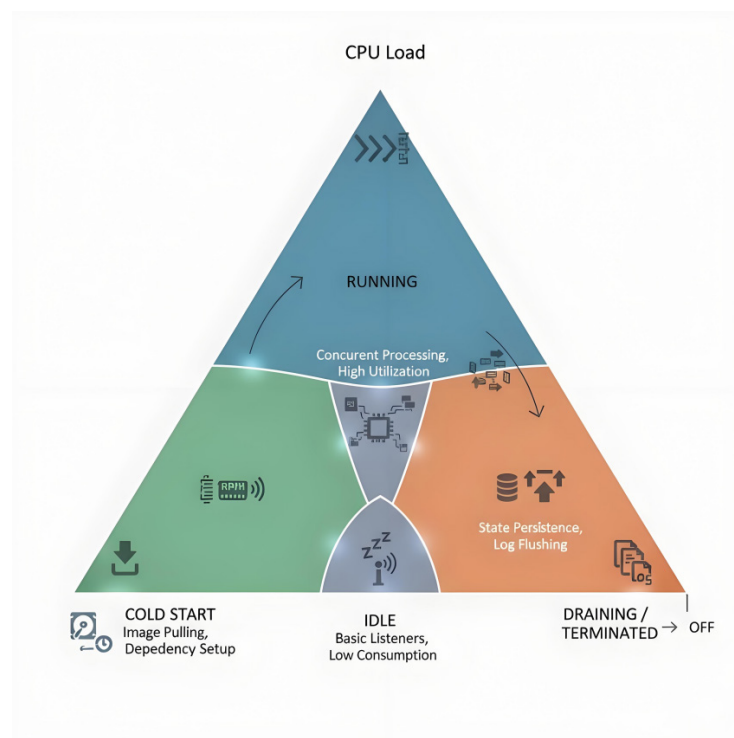


Figure 1. Lifecycle Stage–Resource Signature Mapping for Edge Nodes.

1.3 Consequences of Missing Lifecycle Awareness in Edge Node Management

The absence of lifecycle-aware mechanisms remains a key cause of inefficiency and instability in many edge computing systems. Without accurate lifecycle recognition, schedulers may allocate workloads to nodes that are either draining or approaching decommission, resulting in failed service responses and repeated cold starts that can cascade into full-path bottlenecks. Idle nodes, when not correctly identified, may continue to occupy resources without

contributing active services, leading to resource deadlocks. In microservice architectures—where service dependencies are often tightly interwoven—lifecycle mismanagement can cause service drift, delayed health checks, and inconsistent orchestration states. A real-world instance involves an edge-based video inference platform that continued to dispatch image processing tasks to a draining node, leading to state loss and inference interruption. The disruption cascaded across the pipeline, causing over 12 minutes of service outage. These examples underscore that lifecycle awareness is not merely a matter of efficiency, but a prerequisite for service continuity and orchestration integrity in distributed microservice-based edge systems [2].

2. Design of Lifecycle-Aware Configuration Mechanisms Supported by Microservice Architectures

2.1 Dynamic Task Assembly via Microservice-Based Functional Decoupling

Edge nodes should not rely on static service bundles or pre-fixed application images. Instead, service deployment must align with lifecycle stages through on-demand dynamic assembly [3]. Microservice architectures enable such flexibility by decoupling task logic into lightweight, independently deployable components. For instance, during the “Cold Start” phase, only configuration and log agent services may be required; in the “Running” phase, a full stack including task processor, monitoring sidecars, and synchronization services is activated. Services communicate through APIs (e.g., REST, gRPC), allowing runtime composition based on actual functional needs. Container orchestrators like Kubernetes support this logic by leveraging Deployment templates and init-containers to define stage-specific service bundles. As shown in Table 1, each lifecycle stage corresponds to a distinct set of functional microservices, highlighting the adaptive alignment between decoupled service composition and node operational roles.

Table 1. Mapping Between Lifecycle Stages and Microservice Functional Assemblies

Lifecycle Stage	Example Functional Microservices
Cold Start	Config init, dependency check, log collector
Running	Task processor, data router, state sync, load balancer
Idle	Health probes, monitor agent, low-power maintenance

2.2 Service Mounting Mechanisms Driven by Lifecycle State Detection

Dynamic service deployment in edge environments must rely on precise integration between lifecycle state detection and microservice mounting. Nodes continuously report runtime states via probes (e.g., Kubernetes Liveness/Readiness) and metric collectors like Prometheus. These indicators—such as CPU usage, request latency, and task queue depth—are parsed by orchestration systems to determine matching deployment actions. For instance, entering the “Running” state may trigger the mounting of business services alongside observability and tracing modules. In contrast, entering “Idle” may initiate the removal of nonessential services, preserving only resource monitors. Sidecar-based injection (via Istio or Envoy) allows seamless addition of policy agents, configuration proxies, and logging components in response to state changes. This tight integration between lifecycle feedback and mounting behavior significantly enhances orchestration flexibility and system responsiveness [4].

2.3 Resource Adaptation Strategies Driven by Multi-Dimensional Runtime Metrics

In microservice-based edge systems, resource adaptation must be both reactive and predictive, guided by multi-dimensional runtime metrics. Modern platforms like Kubernetes offer tools such as Horizontal Pod Autoscaler (HPA) and Kubernetes Event-Driven Autoscaling (KEDA), which scale services based on real-time metrics, including CPU utilization, query rate, and latency thresholds. For example, if a container maintains CPU usage over 80% for five minutes, the scheduler may expand replicas and increase memory limits. When resources are constrained, non-core services are scaled down or de-prioritized. More advanced strategies include forecasting models (e.g., Kalman filters, LSTM networks) that anticipate load spikes, enabling preemptive resource allocation. Combined with Kubernetes

constructs like `LimitRange` and `ResourceQuota`, these methods establish fine-grained, automated, and lifecycle-sensitive resource governance for edge environments [5].

3. Resource Configuration and Service Scheduling Strategies During Operational Phases

3.1 Resource Priority Allocation Model Design During Runtime

In edge computing environments, the limited resource capacity of nodes necessitates differentiated allocation strategies across lifecycle stages. To ensure optimal utilization under multi-task concurrency, this study introduces a Resource Allocation Priority Index (RAPI) model driven by three weighted factors: task criticality, service dependency depth, and node health score. The model computes priority through the equation

$$\text{RAPI} = \alpha \times \text{TaskCriticality} + \beta \times \text{ServiceDependencyDepth} + \gamma \times \text{NodeHealthScore},$$

where task criticality measures a service's contribution to the core operational chain, service dependency depth reflects its hierarchical coupling within the microservice topology, and node health quantifies dynamic system metrics such as CPU usage, memory pressure, and latency.

When a node enters the Running state, the scheduler ranks all active services according to their RAPI values, allocating computation and bandwidth preferentially to those with higher scores while throttling lower-priority services to prevent contention [6].

As shown in Table 2, weighting coefficients vary significantly across lifecycle stages—Cold Start prioritizes initialization completeness, Running focuses on throughput stability, Idle emphasizes energy conservation, and Draining safeguards data persistence. The RAPI model is dynamically maintained through Prometheus-based metric collection and Kubernetes scheduling feedback, transforming resource allocation from static provisioning to intelligent, context-aware decision-making, thereby enhancing system stability and resource matching precision.

Table 2. Resource Allocation Priority Rules Across Lifecycle Stages

Lifecycle Stage	Weights (α, β, γ)	Allocation Logic Description
Cold Start	(0.6, 0.2, 0.2)	Preserve initialization completeness; reserve bandwidth and cache
Running	(0.5, 0.4, 0.1)	Emphasize task criticality and dependency depth for throughput
Idle	(0.2, 0.1, 0.7)	Minimize nonessential service usage; maintain health probes
Draining	(0.3, 0.2, 0.5)	Ensure data persistence and graceful service handoff

3.2 Container-Level Dynamic Quota and Scheduling Fine-Tuning Mechanism

Container-level resource governance must provide fine-grained runtime adjustments to accommodate load variability while preventing oscillatory scaling. The practical implementation decomposes into three cooperating layers: metric ingestion \rightarrow decision control \rightarrow enforcement layer. Metric ingestion leverages Prometheus/cAdvisor and a Sidecar autotuner to collect multi-dimensional signals (CPU, RSS, page faults, socket counts, P95/P99 latency, queue depth). Time-series aggregation feeds a decision controller that uses a hybrid control policy: horizontal autoscaling (HPA/KEDA) for burst concurrency, vertical resizing (VPA or controlled cgroup adjustments) for sustained resource pressure; both governed by hysteresis windows and cooldown timers to suppress thrash. A sample control rule: if container CPU $> 80\%$ and latency > 100 ms for 5 consecutive minutes \rightarrow trigger HPA add-replica (+1); if CPU $> 90\%$ for 10 minutes \rightarrow apply stepped VPA-driven increase of requests/limits (split into N increments) to avoid abrupt resource jumps. Enforcement is realized through Kubernetes constructs (`LimitRange`, `ResourceQuota`, Pod Priority, QoS classes) and a safety Operator (CRD) that validates and gates automated VPA actuation (e.g., cap on absolute increase, deny for non-critical pods). Adding a lightweight forecasting module (exponential smoothing or LSTM) enables pre-warming replicas, minimizing cold-start penalties. The control loop keeps audit logs and snapshot checkpoints; if scaling oscillation occurs (>3 scale events within a time window), the system enters a throttled mode, reverting to the last stable configuration. As shown in Figure 2, this feedback loop closes the gap between metric detection and safe enforcement, balancing responsiveness with operational stability [7].

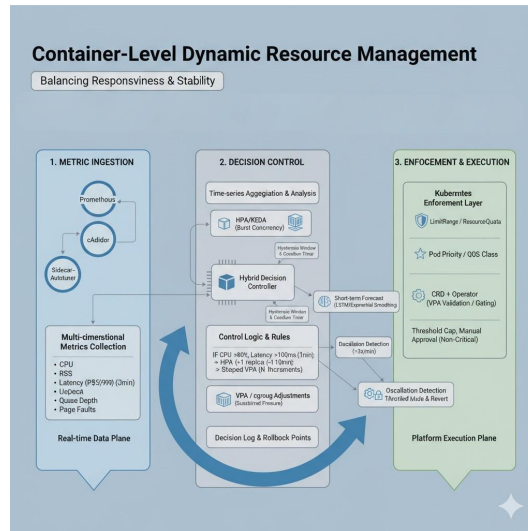


Figure 2. Control loop for container-level autoscaling and resource enforcement.

3.3 Cross-Node Service Migration and Collaborative Deployment in Edge Clusters

In edge computing scenarios, service migration across nodes is often necessitated by resource saturation, network congestion, or lifecycle transitions. To ensure continuity and system stability during such migrations, this section proposes a coordination strategy based on affinity rules, real-time load metrics, and network topology awareness. Each node continuously monitors its health status, and when predefined thresholds are breached (e.g., CPU > 85%, memory > 90%, sustained P99 latency), it enters a de-load state and triggers the scheduler to compute a candidate set of alternative nodes. Selection prioritizes nodes within the same geographic zone to minimize service drift and inter-node latency. Prior to migration, a lightweight snapshot-based state file is generated, and service metadata is synchronized via a registry system (e.g., Consul), ensuring consistency of runtime context and configuration. To reduce cold-start latency, idle nodes may engage in pre-warming routines, loading service images and dependencies in advance. During live migration, a service mesh infrastructure manages traffic proxying, enabling intelligent request rerouting and gray release for seamless handover. This collaborative strategy enhances edge resource utilization, improves service resilience, and stabilizes throughput under dynamic and resource-heterogeneous conditions typical of real-world edge clusters.

4. Resource Reclamation and Service Rebuilding Mechanisms During Node Decommissioning

4.1 State Tagging and Migration Control Mechanism in the Service Retirement Phase

When an edge node reaches the retirement phase of its lifecycle, the system must ensure a smooth service offboarding process while maintaining data integrity [8]. A retirement mechanism centered on state tagging, migration control, and dependency decoupling is therefore essential. Upon receiving the retirement signal, the scheduling controller immediately updates the service registry (e.g., Consul or Etcd) to mark the node and its running instances as Draining. Once this state is activated, no new tasks are assigned to the service, while existing sessions are allowed to complete gracefully, ensuring transactional integrity. Subsequently, a service snapshot procedure is triggered to export runtime states, cache data, and context dependencies into persistent storage, thereby achieving state externalization. If the retiring node hosts distributed or streaming workloads, the migration manager initiates a lightweight instance transfer strategy, selecting optimal target nodes according to network topology and latency thresholds. The process leverages image synchronization and configuration inheritance to minimize downtime. In parallel, all linked dependencies—API gateways, message queues, and database pools—are dynamically updated in the control plane to eliminate dangling references. A configurable grace period further protects against abrupt disconnections or session loss. Finally, through unified logging and callback verification, the entire service retirement becomes traceable and auditable, ensuring that edge clusters maintain operational stability and security during high-concurrency node transitions.

4.2 Asynchronous Resource Cleanup and Multi-Phase Release Strategy Design

After the service retirement process is completed, the system enters the resource reclamation stage. Unlike centralized architectures, edge clusters must reclaim resources without interrupting ongoing workloads or compromising platform continuity, which necessitates a hybrid approach combining asynchronous execution with phased release. Once a service instance transitions from Draining to Terminated, the lifecycle controller activates an Async Cleanup Daemon, which monitors the cleanup queue and processes tasks according to defined priorities.

The first phase, known as the Lightweight Memory and Cache Release Layer, removes ephemeral resources such as container caches, temporary logs, file descriptors, and I/O streams. These operations run in an independent thread pool to prevent blocking of the control plane.

The second phase, the Unmounting and Network Detachment Layer, unbinds virtual network interfaces, detaches mounted volumes and temporary namespaces, and updates the Resource Descriptor Table to prevent misallocation during future scheduling [9].

The third phase, the Persistent Process and Volume Reclamation Layer, invokes the Volume Manager to perform secure data wiping, integrity verification, and dependency detachment. To ensure consistency and prevent race conditions, a Deferred Reclaim Lock mechanism is implemented, which temporarily locks the released resource until the cleanup event is fully acknowledged. Throughout all stages, an EventBus asynchronously reports execution status to the Reclaim Log, enabling full traceability.

This multi-phase asynchronous strategy significantly reduces system blocking risks, decouples resource reclamation from runtime scheduling, and enables safe, high-speed resource reuse under high concurrency, ensuring long-term operational stability of edge nodes.

4.3 Construction of a Resource Reclamation Scheduling Mechanism Based on Priority Weights

After the asynchronous cleanup framework is established, determining which resources should be reclaimed first and which should be retained becomes critical. To achieve precise and safe scheduling, this study introduces a Resource Reclamation Priority Scheduling (RRPS) mechanism that evaluates three dimensions—service type, task interruption cost, and runtime duration—to calculate a composite score guiding the reclamation order. The core formula is defined as:

$$RRPS = \omega_1 \times \text{ServiceType} + \omega_2 \times \text{TaskInterruptCost} + \omega_3 \times \text{RuntimeDuration}$$

where ω_1 , ω_2 , and ω_3 represent weighted coefficients that are dynamically optimized using regression analysis and historical performance data. The scheduler ranks service instances according to their RRPS values, with higher scores indicating higher reclaim priority.

In practice, when node load exceeds the predefined threshold or a retirement signal is triggered, the scheduler generates a real-time reclamation list and builds a Hierarchical Scheduling Tree to represent inter-resource dependencies. Upper-level nodes correspond to interrupt-tolerant services (e.g., log collectors or monitoring probes), while lower-level nodes represent mission-critical services (e.g., data-processing pipelines and persistent storage). Reclamation commands are issued in controlled batches through an event-driven dispatcher, with the Reclaim Lock Manager ensuring mutual exclusion and preventing race conditions or deadlocks.

As shown in Table 3, typical services are categorized into four classes: log collection, main data processing, state synchronization, and resource monitoring. Each class is assigned a different reclamation priority according to its interruption cost and runtime duration. This mechanism enables the system to automatically balance release speed and service stability during overload or migration scenarios, achieving fine-grained control and adaptive optimization of resource reclamation.

Table 3. Priority Scoring Model for Edge Node Service Reclamation

Service Type	Interrupt Cost (0–1)	Runtime (min)	Overall Reclaim Priority
Log Collection	0.2	5	High (Release First)
Main Data Processing	0.9	60	Low (Release Last)
State Synchronization	0.7	25	Medium
Monitoring Probe	0.5	10	Medium-High

5. Conclusion

Edge nodes operate under highly dynamic conditions characterized by resource heterogeneity, short lifecycle spans, and volatile workloads, necessitating lifecycle-aware mechanisms for resource configuration and reclamation. This study presents a comprehensive design for edge node lifecycle management under microservice architectures, including a runtime priority-based allocation model, container-level quota fine-tuning, cross-node collaborative migration, and a multi-stage reclamation strategy. Together, these components form a closed-loop control system that governs the full service lifecycle—from activation to retirement. The proposed mechanisms demonstrate high adaptability, minimal system intrusion, and strong extensibility, making them well-suited for real-world deployment in heterogeneous and decentralized edge environments. This work lays a practical foundation for resilient resource orchestration and intelligent lifecycle operations. Future research may incorporate inference-aware scheduling, energy-efficient policies, and data locality strategies to further enhance scheduling intelligence and system-level self-adaptation in complex edge computing contexts.

References

- [1] Cao Y, He Y, Zhang C. IMLMA: An Intelligent Algorithm for Model Lifecycle Management with Automated Retraining, Versioning, and Monitoring. *J Electron Res Appl*. 2025;9(5):233-48.
- [2] Yang M, Wang X, Cai B, et al. Full stack optimization of microservice architecture: systematic review and research opportunity. *Cluster Comput*. 2025;28(15):1005.
- [3] Otero M, García MJ, Fernandez P. An extensible lightweight framework for distributed telemetry of microservices. *Sustain Comput Inform Syst*. 2025;46:101100.
- [4] Tusa F, Clayman S, Buzachis A, et al. Microservices and serverless functions—lifecycle, performance, and resource utilisation of edge based real-time IoT analytics. *Future Gener Comput Syst*. 2024;155:204-18.
- [5] Alshuqayran N, Ali N, Evans R. A model-driven architecture approach for recovering microservice architectures: Defining and evaluating MiSAR. *Inf Softw Technol*. 2025;186:107808.
- [6] Fávero FL, Almeida DRN, Affonso JF. A Systematic Mapping Study on the Modernization of Legacy Systems to Microservice Architecture. *Appl Syst Innov*. 2025;8(4):86.
- [7] Anand J, Karthikeyan B. Dynamic priority-based task scheduling and adaptive resource allocation algorithms for efficient edge computing in healthcare systems. *Results Eng*. 2025;25:104342.
- [8] Lokesh G, Baseer KK. A meta-heuristic approach-aided multi-objective strategy with optimal resource allocation via fault tolerant and priority-based scheduling for load balancing in cloud. *Wirel Netw*. 2024;31(3):1-23.
- [9] Karatza DH, Stavrinides LG. Resource allocation and aging priority-based scheduling of linear workflow applications with transient failures and selective imprecise computations. *Cluster Comput*. 2024;27(4):5473-88.