IoT

MDPI

Article

Benchmarking Service-Mesh Scalability in IoT-to-Cloud Microservice Continuums: A Two-Axis Taxonomy and Tail-Latency Evaluation

Michail Alexandros Kourtis [1,*], Achilleas Economopoulos [1], George Xilouris [1], Evangelos Markakis [2] and Ricard Vilalta [3]

[1] Institute of Informatics and Telecommunications, NCSR "Demokritos", 15341 Agia Paraskevi, Greece; aeconomopoulos@iit.demokritos.gr (A.E.); xilouris@iit.demokritos.gr (G.X.)
[2] Department of Electronic Engineering, Hellenic Mediterranean University, 71410 Heraklion, Greece; emarkakis@hmu.gr
[3] Centre Tecnològic de Telecomunicacions de Catalunya (CTTC/CERCA), 08860 Castelldefels, Barcelona, Spain; ricard.vilalta@cttc.es
* Correspondence: akis.kourtis@iit.demokritos.gr

Abstract

The proliferation of Internet of Things (IoT) devices operating in swarm-like formations—from autonomous device fleets to federated service networks—demands a computing continuum that spans edge nodes, fog gateways, and cloud back-ends. Service meshes such as Istio provide transparent traffic management, observability, and policy enforcement for the microservice fabrics that underpin these continuums, yet their scalability behaviour under diverse coordination patterns and increasing load intensity remains poorly characterised. This paper introduces a *two-axis taxonomy* for benchmarking service-mesh scalability in IoT-to-cloud microservice continuums: Axis A captures four scaling tiers (Baseline, Moderate, High, Extreme) derived from offered load and concurrency, while Axis B is measured by the *coordination-amplification factor* φ, the number of service-to-service calls that one request triggers, and groups the scenarios into *federated* patterns, in which services call only distinct downstream services, and *peer-to-peer* patterns, in which services also call peer instances of themselves. We instantiate the taxonomy across six representative IoT scenarios, each exercised through multi-phase Fortio load campaigns over an Istio-meshed Kubernetes cluster. Analysing 51 benchmark runs, we report percentile latency distributions (P50, P90, P99), load fidelity, and throughput–tail coupling. Because every scenario is driven with fixed-size payloads, per-request application processing is held constant, which lets us separate the effect of coordination structure from that of application weight. At matched application weight and offered load, peer-to-peer coordination adds only a modest tail premium over federated rollout (a factor of ≈1.2–1.8 at 100–200 RPS); the severe effects—up to an order-of-magnitude P99 growth from a scenario's own baseline to saturation, and achieved throughput collapsing to roughly half the requested rate—emerge specifically for heavy-compute peer-exchange workloads driven toward saturation. This interaction of load, coordination structure, and application weight is what the two-axis grid is designed to expose, and negligible-compute acknowledgement endpoints additionally provide a direct per-hop mesh-overhead measurement. We deliberately scope the study to the cloud-side segment of the continuum—the layer on which the service mesh runs—and within it to the single-node data plane, where mesh overhead can be measured in isolation from cross-node network variability; device and edge tiers enter only as offered load, and no claim is made about inter-tier transport. The A×B grid exposes otherwise hidden interaction effects between coordination topology and scale, providing actionable design guidance for service-mesh capacity planning and auto-scaling in IoT microservice deployments.

Check for updates

Academic Editors: Hyun-Ho Choi and Grigorios E. Koulouras

Received: 6 August 2026
Revised: 18 September 2026
Accepted: 8 October 2026
Published: 10 October 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

Keywords: service mesh scalability; IoT-to-cloud continuum; tail latency; Istio; microservice benchmarking; Kubernetes

1. Introduction

1.1. IoT-to-Cloud Microservice Continuums

The Internet of Things has evolved far beyond simple telemetry collection. Modern IoT deployments orchestrate heterogeneous fleets of devices—drones, robots, sensor arrays, electric-vehicle charging stations—that collectively perform cognitive and artificial-intelligence (AI) tasks such as anomaly detection, cooperative inference, and federated model training [1,2]. These device fleets increasingly resemble biological swarms: autonomous agents that coordinate locally yet contribute to a global objective [3,4].

Realising such swarm-like behaviour across the IoT-to-cloud *computing continuum* [5–7] poses substantial systems challenges. Data must flow from resource-constrained edge nodes, through fog or edge gateways, to cloud-hosted AI services and back, often within tight latency budgets. The microservice architectural style [8] has become the dominant paradigm for structuring the cloud-side (and increasingly edge-side) components of these systems, decomposing monolithic applications into independently deployable services that communicate over the network.

As the number of interacting microservices grows, so does the complexity of inter-service communication: load balancing, circuit breaking, retries, mutual TLS authentication, and distributed tracing must all be managed. Manually instrumenting each service for these cross-cutting concerns is error-prone and scales poorly, motivating the adoption of *service mesh* infrastructure.

Scope of the measurements. The object measured in this paper is the scalability of sidecar-based service meshes for microservice applications, on the cloud-side tier of an IoT-to-cloud continuum where the mesh runs. The device and edge tiers enter the measurements in one way only, as offered load: a fleet of N devices emitting requests at rate r presents an aggregate rate $N \cdot r$ to the entry services, and it is this aggregate that Axis A varies (Section 4.2). No claim is made about device-to-edge or edge-to-cloud transport. The continuum nevertheless defines the problem the measurements address, in two respects. It supplies the workloads: the six scenarios reproduce coordination patterns of IoT deployments—the canary rollout of an inference model, aggregation across charging stations, and peer exchange among turbine diagnostic nodes—which general-purpose microservice benchmarks do not contain (Section 3.6). It also supplies the condition under which the results matter: the IoT services benchmarked here answer within roughly 3 to 65 ms at low load, a range in which a few milliseconds per mesh hop is a material share of the latency, whereas it would be negligible beside a computation lasting minutes (Section 9.2). The consequences of the single-tier, single-node scope for external validity are examined in Sections 9.5 and 10.2.

1.2. Why Service-Mesh Scalability Matters

A service mesh interposes a network of sidecar proxies (typically Envoy [9]) alongside each microservice instance, creating a dedicated infrastructure layer for service-to-service communication [10]. Istio [11], one of the most widely adopted open-source service meshes, provides three capabilities that are particularly relevant to IoT microservice continuums. Through VirtualServices and DestinationRules it enables fine-grained traffic management—weighted splitting for canary deployments between model versions, timeouts, and fault injection—all without modifying application code. It also delivers observ-

ability essentially for free, automatically collecting request-level metrics, distributed traces, and access logs that give operators deep visibility into every service-to-service call. Finally, it enforces policy declaratively: mutual TLS, rate limiting, and authorisation rules can be applied mesh-wide, which matters for IoT deployments that span trust boundaries.

However, these capabilities come at a cost: each sidecar proxy introduces additional latency, CPU overhead, and memory consumption per request hop [12–14]. Understanding how this overhead scales under increasing load and different communication topologies is critical for capacity planning, auto-scaling policy design, and service-level objective (SLO) compliance in production IoT deployments [15].

1.3. Benchmarking Gap: Coordination Style × Scaling Conditions

Existing benchmarking studies of service meshes [10,12,14] and microservice platforms [16,17] typically evaluate performance along a single dimension: offered load. They report how latency and throughput degrade as requests per second increase. However, IoT microservice continuums introduce a second critical dimension: the *coordination topology* of a service, by which we mean the set of service-to-service calls that one request to the service triggers—which services call which, and how many times—as distinct from its *deployment* topology, which concerns how pods are placed on nodes. Section 4.1 formalises it as the invocation tree of a request, a static description of the call code in which repeated calls are counted with their multiplicity. Two services with identical deployment topologies can have entirely different coordination topologies, and it is the latter that determines how much traffic one external request generates inside the mesh. Whether the resulting overhead matters depends in turn on how long the services themselves compute: a few milliseconds per hop is negligible beside a computation of minutes but material beside one of tens of milliseconds, and Section 9.2 quantifies this relation for the benchmarked services. A federated learning aggregation service exhibits fundamentally different communication patterns (periodic bulk model updates) compared with a peer-to-peer cooperative control loop (continuous, low-latency inference exchanges among peers).

Ignoring this second dimension leads to misleading conclusions: a mesh configuration that performs well for federated rollout workloads may exhibit unacceptable tail latencies under peer-to-peer cooperative patterns at the same offered load. This paper addresses this gap by proposing a two-axis taxonomy that systematically varies both scaling conditions (Axis A) and coordination categories (Axis B), enabling a richer characterisation of mesh-mediated scalability.

The motivation is practical and time-critical: IoT deployments are moving from pilot-scale observability to production-grade AI and security-sensitive services, where SLO violations at P99 directly affect safety, operating cost, and compliance outcomes. Recent evidence from microservice benchmark suites [16], cloud-native latency analyses [12,18], and edge-IoT operational studies [19] consistently shows that control-plane convenience does not eliminate data-plane tail risk. A benchmarking method that jointly captures load intensity and coordination style is therefore necessary for deployment decisions, not just academic comparison.

1.4. Contributions and Paper Structure

This paper makes five contributions. At its centre is a two-axis benchmarking framework—Axis A for scale tiers and Axis B for coordination categories—for service-mesh scalability in IoT-to-cloud microservice continuums, accompanied by an operational procedure that maps an arbitrary service call graph onto the two axes (Section 4, Figure 1). To exercise the framework we design a suite of six use cases spanning federated and peer-to-peer coordination patterns, each driven through multi-phase load campaigns generated

with Fortio, the Istio team's open-source HTTP/gRPC load generator [20]. On this basis we report a comprehensive empirical evaluation of 51 benchmark runs across the A×B grid, covering P50/P90/P99 latency distributions, load fidelity, throughput–tail coupling, and cross-cell interaction effects. Crucially, we separate the effect of coordination structure from that of application weight by holding payloads constant across load tiers, and we use negligible-compute acknowledgement endpoints as a direct mesh-overhead probe, attributing the load-dependent tail growth predominantly to mesh and queueing (Section 8.4). From these results we derive design hypotheses for service-mesh capacity planning and auto-scaling in IoT microservice deployments.

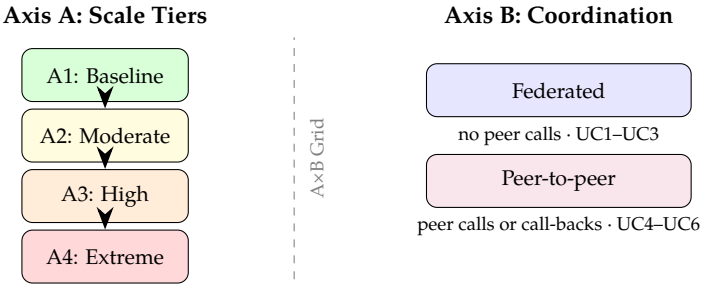


Figure 1. Two-axis benchmarking taxonomy. Axis A defines four scaling tiers from offered load and concurrency. Axis B is measured by the coordination-amplification factor φ, the number of service-to-service invocations that one request triggers (Section 4.1); for presentation, the six use cases are grouped into a federated category, whose services call only distinct downstream services, and a peer-to-peer category, in which some service calls a peer instance of itself or calls back its caller.

The remainder of this paper is organised as follows. Section 2 reviews background concepts. Section 3 provides a detailed comparison with related work. Section 4 presents the two-axis taxonomy. Section 5 details the scenario suite. Section 6 describes the experimental setup. Section 7 defines the metrics and statistical treatment. Section 8 presents the results. Section 9 discusses implications. Section 10 addresses threats to validity. Section 11 concludes.

2. Background

2.1. IoT/Edge-Cloud Continuum and Microservice-Based Platforms

The computing continuum—spanning far-edge IoT devices, fog and edge gateways, and centralised cloud data centres—has been extensively studied [21–23], with Moreschini et al. [5] formalising the *cloud continuum* as a substrate over which workloads are placed by latency, bandwidth, and governance constraints, and Ullah et al. [6] surveying orchestration across the cloud-to-things continuum together with its taxonomy and open challenges. Within this continuum, microservices have become the dominant structuring paradigm for IoT platforms [8]: decomposing applications into small, independently deployable services over HTTP/REST or gRPC enables rapid iteration, heterogeneity, and fine-grained scaling [24,25], with Kubernetes [26] as the de facto orchestrator across cloud and edge [19,27], extended to constrained devices by KubeEdge and lightweight distributions [28,29]. Dustdar et al. [7] frame these as *distributed computing continuum systems* that dynamically compose services across tiers, and Rosendo et al. [30] systematically review distributed intelligence across the edge-to-cloud continuum and underscore the need for cross-tier benchmarking methodologies—a gap our work directly addresses.

2.2. Swarm Intelligence: Consensus-Driven and Peer-to-Peer Coordination

Swarm intelligence, originally inspired by the collective behaviour of social insects [3], encompasses a family of decentralised, self-organising algorithms including ant colony

optimisation [31] and particle swarm optimisation [32]. Brambilla et al. [4] survey swarm robotics from an engineering perspective, identifying coordination mechanisms such as stigmergy, consensus, and direct communication.

In the IoT context, swarm-like behaviour manifests when large numbers of devices collaboratively perform tasks without centralised control. Examples include drone fleets performing cooperative inspection, electric-vehicle charging networks negotiating grid load, and sensor arrays performing federated anomaly detection. Federated learning [33,34] represents a prominent coordination pattern in which edge devices train local models and periodically aggregate them at a central server, preserving data locality and privacy. Two properties of such systems are easily conflated and are kept apart in this paper. The first is *how a decision is reached*: under *consensus* several participants agree on a common value, whereas under central control one component decides. The second is *how the participants communicate*: in a *federated* structure they exchange data only through a central coordinator or aggregator, whereas in a *peer-to-peer* structure they also exchange data directly with one another [35]. The two properties are independent—a consensus can be reached through peer-to-peer exchange just as through a central aggregator—and only the second, the communication structure, is used to group the scenarios of this paper (Section 4.3).

2.3. Service Mesh Architecture

A service mesh provides a dedicated infrastructure layer for handling service-to-service communication [10]. The dominant architecture interposes a sidecar proxy (typically Envoy [9]) alongside each service instance, intercepting all inbound and outbound traffic. A control plane (Istio's `istiod`) configures the proxies with routing rules, load-balancing policies, TLS certificates, and telemetry directives [11]. Li et al. [10] identify key service mesh capabilities: traffic management (routing, splitting, mirroring), resiliency (retries, circuit breaking, timeouts), security (mTLS, authorisation), and observability (metrics, traces, logs).

Recent developments in sidecar-less architectures offer promising alternatives. Istio's ambient mesh mode [36], which reached general availability in v1.24, replaces per-pod sidecars with shared per-node ztunnel proxies and optional Layer-7 waypoint proxies. Similarly, Cilium's eBPF-based service mesh [37] operates entirely within the Linux kernel, bypassing userspace proxy overhead for Layer-3/4 operations. Lee [38] provides a comparative analysis of four proxy models (per-pod sidecar, proxyless, eBPF kernel-native, and ambient), finding that proxyless models offer superior throughput at the cost of application coupling, while ambient mesh provides a promising middle ground. These architectural innovations are reshaping the performance trade-off landscape. We nonetheless focus our evaluation on the traditional sidecar model for three deliberate reasons. First, sidecar-based Istio remains the most widely deployed service-mesh configuration in production, so its scaling behaviour under IoT coordination patterns is the immediately actionable result for the majority of current deployments. Second, it provides a stable, well-understood baseline: the absolute gains reported for ambient and eBPF meshes (e.g., Lee [38], Simas et al. [39]) are expressed relative to sidecar Istio, so characterising the sidecar baseline across the A×B grid is a prerequisite for interpreting those gains under realistic coordination topologies. Third, and most importantly, our contribution is the two-axis method, which is architecture-agnostic by design; the same grid and mapping procedure can be re-run on ambient or eBPF data planes, and doing so is an explicit item of future work (Section 11.2) rather than a limitation of the framework itself.

2.4. Tail-Latency and Percentile-Based Evaluation

Dean and Barroso's seminal paper "The Tail at Scale" [40] demonstrates that in large-scale distributed systems, even rare high-latency events accumulate and dominate end-to-

end user experience. They advocate for percentile-based latency reporting (P90, P99, P99.9) rather than averages, which mask distributional pathologies. This insight is particularly relevant for IoT microservice continuums, where a single slow service call in a multi-hop chain can delay an entire cooperative inference cycle. Pelle et al. [18] study latency-sensitive cloud-native applications, quantifying how component placement and inter-service communication contribute to end-to-end latency. Sigelman et al. [41] describe distributed tracing infrastructure that enables per-hop latency attribution—a capability leveraged in our analysis to distinguish mesh overhead from application processing time.

3. Related Work

This section reviews the most closely related empirical studies and positions our contributions with respect to their scope, methodology, and findings. Table 1 provides a structured comparison across nine dimensions.

3.1. Service Mesh Performance and Overhead Studies

Elkhatib and Poyato [12] evaluate service-mesh frameworks for edge systems, quantifying their latency and resource overheads; however, their evaluation uses standard request/response patterns and does not vary coordination topology. Saleh Sedghpour et al. [14] empirically study service-mesh traffic-management policies for microservices, showing that policy configuration (retries, timeouts, and circuit breaking) materially affects latency and success rate under load. Neither study examines IoT-specific workloads or the interaction between coordination topology and scale.

Sedghpour and Townend [13] survey service mesh and eBPF-based alternatives, providing a taxonomic overview of architectural trade-offs but no original empirical data. Zhu et al. [42] contribute MeshInsight, a tool that dissects sidecar overhead at the system-call level, revealing that service meshes can incur up to 269% latency inflation and 163% vCPU overhead for benchmark applications. Their analysis is fine-grained and architecturally insightful but does not consider multi-tier load scaling or heterogeneous coordination patterns.

In the emerging sidecar-less space, Simas et al. [39] propose eSeMeshA, an eBPF-based acceleration framework that achieves over 70% throughput gains and 42% latency reduction compared to unoptimised Istio ambient mesh. Their work highlights the potential of kernel-level optimisation but evaluates only synthetic point-to-point workloads, not the cascading topologies studied here.

3.2. Microservice Benchmarking Suites

Gan et al. [16] introduce the DeathStarBench suite for microservices, demonstrating that P99 tail behaviour is highly sensitive to resource contention and queuing effects. The suite models social-network and hotel-reservation workloads—both following fan-out patterns typical of web applications—but does not include IoT-specific scenarios or systematic coordination-pattern variation. Sriraman and Wenisch [17] present μSuite, targeting latency-critical microservices with sub-millisecond SLOs. Their focus on low-latency RPCs complements our work, which targets the higher-latency regime characteristic of IoT inference pipelines. Ueda et al. [43] provide early workload characterisation for microservice architectures, identifying communication-intensive patterns that stress inter-service networking.

3.3. Cloud-Native Scalability Benchmarking

Henning and Hasselbring [15] propose a configurable method for benchmarking scalability of cloud-native applications, defining scalability as the ability to meet SLOs under increasing load through horizontal scaling. Their framework introduces systematic load generation with controlled ramp-up profiles and SLO-based assessment—principles

that inform our multi-phase Fortio campaign design. In a follow-up study, Henning and Hasselbring [44] benchmark stream-processing frameworks deployed as microservices on Kubernetes, finding approximately linear scalability across all four frameworks tested but with significant differences in per-instance resource requirements. Their work confirms that scaling behaviour is workload-sensitive, but evaluates only data-streaming patterns.

Eismann et al. [45] empirically examine microservice performance testing, showing that architecture and configuration choices strongly influence measured performance and complicate cross-study comparison. Rodrigues et al. [46] offer a recent systematic literature review of performance and scalability assessment in microservice architectures, identifying persistent gaps in comparative studies of architectural patterns and in end-to-end assessment approaches that combine SLO definition, operational profiling, and resource activation evaluation. Both reviews reinforce the need for the multi-dimensional benchmarking approach presented here.

3.4. Coordination and Architectural Pattern Evaluation

Meijer et al. [47] present an empirical study evaluating the performance behaviour of microservice architectural patterns (Gateway Aggregation, Gateway Offloading, Pipe and Filters), demonstrating that model-based predictions align with measurements. Karabey Aksakalli et al. [48] survey deployment and communication patterns in microservice architectures, identifying seven communication patterns and six communication challenges. These pattern-level investigations provide architectural context for our Axis B (coordination category) but do not measure performance under systematic load scaling.

3.5. IoT-Edge Platform Performance

In the IoT-edge domain, Čilić et al. [19] evaluate the performance of container-orchestration tools for Kubernetes at the edge, quantifying their overheads and operational variability. Vaño et al. [29] review cloud-native workload orchestration at the edge, identifying gaps between cloud-native tooling assumptions and edge resource constraints. Marchese and Tomarchio [49] extend Kubernetes with a load-aware orchestration strategy for microservices in shared clusters with dynamic resource patterns. Ahmad et al. [50] propose reactive and proactive auto-scaling strategies (Smart HPA and ProSmart HPA) that outperform Kubernetes' default HPA in microservice environments. Luo et al. [51] present a scalable system design for guaranteeing SLAs in shared microservice deployments through latency profiling and interference-aware placement. These studies address resource management and auto-scaling but do not evaluate service-mesh overhead under varying coordination topologies—a gap our work fills.

3.6. Research Gap and Our Contribution

Table 1 provides a structured comparison across nine evaluation dimensions. To keep the comparison meaningful, the table includes only studies that report original empirical benchmarking measurements along axes comparable to ours (load, topology, percentile latency, mesh overhead); purely conceptual or secondary works discussed above—capability taxonomies and architectural surveys [13,48], systematic literature reviews [46], edge-orchestration reviews [29], and resource-management systems without a mesh-overhead evaluation [51]—are positioned narratively but omitted from the grid because they do not produce comparable measurements. With that scope fixed, the table reveals that while existing studies address individual aspects—mesh overhead, benchmark suites, scalability methods, or IoT platforms—none jointly evaluate all dimensions relevant to service-mesh scalability in IoT continuums. Three gaps stand out in particular. No prior study treats coordination topology (federated versus peer-to-peer) as an independent variable alongside

load intensity. The IoT-specific communication patterns that motivate our work—periodic aggregation, cooperative inference, and peer exchange—are absent from existing benchmark suites. And the interaction between coordination topology, scaling tier, and per-request application weight—which, as we show, jointly govern tail latency—has not been characterised before. Our two-axis taxonomy fills these gaps by providing a structured framework for evaluating service-mesh scalability under the heterogeneous workload patterns characteristic of IoT-to-cloud continuums.

Table 1. Detailed comparison with related work across nine evaluation dimensions. ✓ = addressed; (✓) = partially addressed; – = not addressed.

Study	*Service Mesh*	*Multi-Tier Load*	*Coord. Topology*	*IoT Workloads*	*P99 Reporting*	*Sidecar Overhead*	*Auto-Scaling*	*Multi-Scenario*	**Key Focus/Limitation**
Li et al. [10] (2019)	✓	–	–	–	–	(✓)	–	–	Mesh capability taxonomy; no empirical benchmarks
Elkhatib & Poyato [12] (2023)	✓	✓	–	–	✓	✓	–	–	Service-mesh frameworks for edge; no topology var.
Saleh Sedghpour et al. [14] (2022)	✓	✓	–	–	✓	✓	–	–	Mesh traffic-management policies; no IoT patterns
Zhu et al. [42] (2023)	✓	–	–	–	(✓)	✓	–	–	Fine-grained sidecar dissection; single-load level
Simas et al. [39] (2025)	✓	–	–	–	(✓)	✓	–	–	eBPF acceleration; point-to-point only
Gan et al. [16] (2019)	–	✓	(✓)	–	✓	–	–	✓	DeathStarBench: web fan-out; no mesh evaluation
Sriraman & Wenisch [17] (2018)	–	✓	–	–	✓	–	–	✓	µSuite: sub-ms SLOs; no mesh, no IoT
Henning & Hasselb. [15] (2022)	–	✓	–	–	✓	–	✓	–	Scalability method; SLO-based; no mesh
Henning & Hasselb. [44] (2024)	–	✓	(✓)	–	✓	–	✓	✓	Stream processing scaling; no mesh, no IoT
Eismann et al. [45] (2020)	–	(✓)	(✓)	–	(✓)	–	–	–	Microservice performance testing; no mesh, no IoT
Meijer et al. [47] (2024)	–	✓	✓	–	–	–	–	✓	Architectural patterns eval; no mesh, no IoT
Ahmad et al. [50] (2025)	–	✓	–	–	–	–	✓	–	Auto-scaling HPA; no mesh, no coordination
Marchese & Tomarchio [49] (2025)	–	✓	–	(✓)	–	–	✓	–	Load-aware K8s scheduling; no mesh eval
Čilić et al. [19] (2023)	–	–	–	✓	–	–	–	–	K8s-at-edge orchestration perf.; no mesh, no scaling
This work	✓	✓	✓	✓	✓	✓	(✓)	✓	**Two-axis taxonomy; 6 IoT scenarios; 51 runs across A×B grid; P50/P90/P99**

4. Two-Axis Taxonomy for Service-Mesh Scalability

This section formalises the framework that structures our benchmarking evaluation. We use the term *taxonomy* in the operational sense of a classification scheme with explicit membership rules: it is a structured benchmarking grid whose two axes are defined by measurable properties of a workload (offered load and coordination topology), together with a procedure (Section 4.4) for assigning any service to a cell. The framework is intended to be prescriptive—it tells an experimenter which axis values to vary and how to read the resulting grid—rather than a descriptive catalogue of all possible IoT architectures. Figure 2

gives an end-to-end view of how the framework is defined and how the remainder of the paper instantiates, measures, and analyses it; the subsections that follow develop each layer in turn.

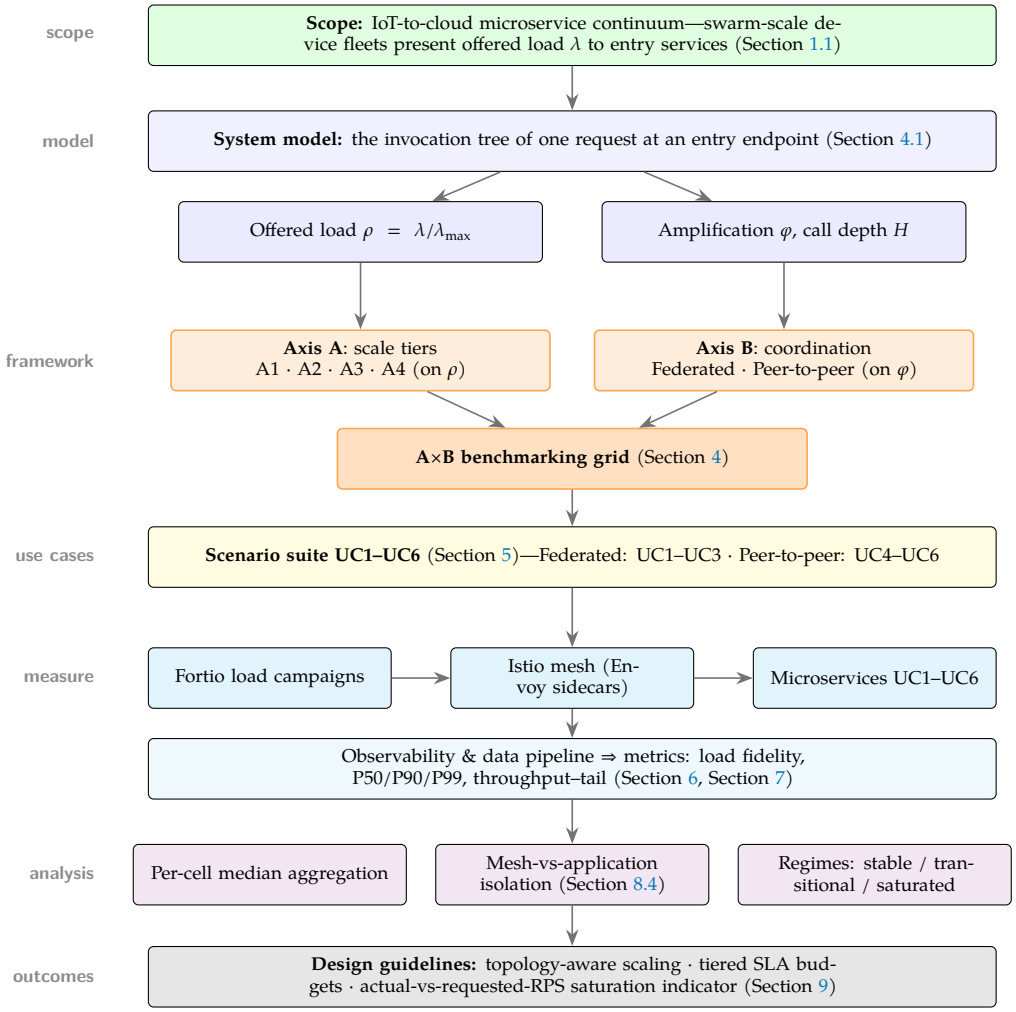


Figure 2. Architecture of the paper's scope and benchmarking framework. The study proceeds top-to-bottom: the IoT-to-cloud continuum supplies offered load to services whose requests are modelled as invocation trees; two measurable quantities—normalised load ρ and coordination-amplification factor φ—define Axis A and Axis B, whose product is the A×B grid; the grid is instantiated by the six-scenario suite and exercised through the Fortio → Istio → microservice measurement pipeline; per-cell aggregation together with the mesh-versus-application isolation then yields the three performance regimes and the design guidelines. Section references mark where each layer is developed.

4.1. System Model and Coordination-Amplification Factor

We model the work triggered by one client request as an *invocation tree*. The request enters at an *entry endpoint* r—in our scenarios, the endpoint driven by Fortio—and every service-to-service call it causes, whether directly or through intermediate services, appears as one edge of the tree. The tree is a static description of the deployed code for a given input: a call issued twice, or once per iteration of a loop, appears as two (or k) separate edges, so repeated calls between the same pair of services are counted with their multiplicity. Two quantities of this tree govern the cost that the service mesh adds to a request.

The first is the **coordination-amplification factor** $\varphi(r)$, the number of service-to-service invocations in the invocation tree of one request at r. Each invocation crosses the mesh once, so φ measures the load that one external request places on the sidecar data plane, and it is the quantity on which Axis B rests. The second is the **call depth** $H(r)$, the number of levels

of the tree, that is, the length of the longest chain of calls in which each call is issued by the callee of the previous one.

A **mesh hop** is one traversal of a sidecar pair—the egress Envoy proxy of the caller together with the ingress Envoy proxy of the callee—and costs latency L (Appendix G). It is not a network hop: because all pods are co-located on a single node (Section 6.1), every call traverses the loopback interface and the number of network hops is zero throughout. The request from the load generator to r is itself one mesh hop.

How much of this mesh cost lies on the critical path of a request depends on dispatch. The deployed handlers issue their downstream and peer calls with a blocking HTTP client, one after another, and return only once every call has been answered (Appendix E). Under this discipline all $1 + \varphi$ hops lie on the critical path, and the mesh contributes approximately $(1 + \varphi)L$ to the latency of each request. Were the calls instead issued in parallel, only one hop per level would remain on the critical path, and the contribution would fall towards $(1 + H)L$. The two bounds coincide for a simple chain and diverge for a wide fan-out; for the deployed services the first applies, which is why φ, rather than H, is the parameter used throughout the analysis. Concurrency across different requests is a separate matter: it is supplied by the parallel connections of the load generator (Section 5.3), and it is what produces queueing at the sidecars.

Table 2 lists φ and H for every endpoint exercised in the benchmark, derived from the call code of Appendix E. The factor ranges from $\varphi = 0$ for terminal acknowledgement endpoints to $\varphi = 6$ for the full-mesh exchange of UC6, in which each node calls its two peers and each peer forwards the message once more to its own two peers. Forwarding is bounded by a **time-to-live (TTL)** counter, an integer carried in every peer message: a receiving node forwards the message only while the TTL is positive and decrements it on forwarding, so a TTL of 1 permits exactly one further level. Axis A and Axis B are thus not free-standing labels but projections of two measurable quantities—offered load and φ—onto discrete tiers and categories.

Table 2. Coordination-amplification factor φ (service-to-service invocations per request) and call depth H for every benchmarked entry endpoint, derived from the call code of Appendix E. With blocking sequential dispatch the mesh contributes approximately $(1 + \varphi)L$ to each request.

UC	Category	Entry Endpoint	φ	H	Invocations Triggered
UC1	Federated	`/predict` (v1)	1	1	rating service
		`/predict` (v2)	0	0	none
UC2	Federated	`/forecast`	2	1	two actor `/result` calls
		`/result`, `/charge`	0	0	none
UC3	Federated	`/detect`	1	1	result store
UC4	Peer-to-peer	`/analyze`	3	2	peer, one peer forward, classifier
UC5	Peer-to-peer	`/schedule`	1	1	result store
		`/check` †	0	0	none (feedback branch not taken)
UC6	Peer-to-peer	`/analyze`	6	2	two peers, four peer forwards

† With the fixed test image the inspection score clears the threshold, so the feedback branch ($\varphi = 3$, $H = 2$) is not exercised; the single-decode latency of the measured runs (median P50 of 53.6 ms at A1–A2, against 57.8 ms for the single decode of UC3) confirms this.

4.2. Axis A: Scale Tiers

Axis A captures the intensity of the offered workload. To keep the axis transferable across hardware, we define each tier by a dimensionless load level rather than by absolute request rates alone. Let λ be the offered request rate in *requests per second* (RPS)—throughout this paper RPS denotes the rate aggregated across all client threads, never a per-thread rate—

let c be the number of concurrent connections (threads), and let $\lambda_{\max}$ be the empirically observed saturation throughput of the service under test (the rate beyond which achieved RPS no longer tracks requested RPS; Section 7.1). The normalised load is $\rho = \lambda/\lambda_{\max}$, and the four tiers correspond to load regimes $\rho \lesssim 0.1$ (Baseline), $0.1 < \rho \lesssim 0.4$ (Moderate), $0.4 < \rho \lesssim 0.85$ (High), and $\rho > 0.85$ or measured $\rho_{\text{achieved}} < \rho_{\text{requested}}$ (Extreme, i.e. the onset of saturation). For our testbed these regimes instantiate to absolute thresholds that we use as operational shorthand throughout. The *Baseline* tier A1 (RPS ≤ 10, threads ≤ 10) represents quiescent or lightly loaded conditions such as periodic telemetry ingestion or low-frequency model updates. The *Moderate* tier A2 (RPS in $(10, 50]$ or threads in $(10, 20]$) is steady-state operation with moderate concurrency, typical of normal daytime workloads. The *High* tier A3 (RPS in $(50, 200]$ or threads in $(20, 50]$) captures peak-hour conditions or event-driven spikes, such as a fleet of drones simultaneously uploading inspection images. Finally, the *Extreme* tier A4 (RPS > 200 or threads > 50) is a stress regime designed to expose breaking points, resource saturation, and tail-latency inflation. These thresholds are calibrated to the capacity of our testbed (Section 6) and the benchmarked services; in production the absolute RPS values would differ, but the relative tier structure remains applicable.

4.3. Axis B: Coordination Categories

Axis B describes how a service coordinates with other services, and it is measured by φ. For presentation, the six scenarios are grouped into two categories that differ in where the additional invocations go. In the **federated** category (UC1–UC3), an entry service calls distinct downstream services—a rating service, two charging-station actors, a result store—and no service on any execution path calls another instance of its own service or calls back a service that called it. In the **peer-to-peer** category (UC4–UC6), at least one execution path contains either a call to another instance of the same service (a *peer*), used to exchange intermediate results, or a call back to a service already on the path. The two analysis versions of UC4 exchange features as peers, the three diagnostic nodes of UC6 exchange and forward their results as peers, and in UC5 the object-processing service calls back the inspection service that invoked it. The category is read from the call code of the service across all of its branches (Appendix E), whereas φ describes the branch that a given request actually takes.

Two clarifications fix the meaning of these terms. First, the categories refer to communication structure only, not to how a decision is reached. A consensus, in which several nodes agree on a value, can be reached through peer-to-peer exchange, as in UC6, or through a central aggregator, and for that reason *consensus-driven* is not used as a category name. Second, the categories are descriptive, and the quantitative analysis uses φ (Table 2): under blocking dispatch the mesh cost of a request is set by its number of invocations, so two services with equal φ and equal application weight are expected to behave alike irrespective of category. The data are consistent with this expectation. The only pair of endpoints in the suite that is matched in application weight and close in φ but lies in different categories—UC3 `/detect` ($\varphi = 1$) and UC5 `/check` ($\varphi = 0$ on the measured branch), both performing the same medium image decode—differs in median P99 by a factor of 0.95–1.20 across A1–A3. The category differences reported in Section 8 are, at matched application weight, differences in φ; the categories are retained because they name the deployment patterns from which the scenarios are drawn, and because the peer-to-peer patterns are where the larger values of φ arise.

4.4. Operational Mapping Procedure

The two axes are assigned to any service—not only the six studied here—by the following procedure, which depends only on observable workload parameters and on the service

call graph (recoverable from distributed traces or from the deployment's routing manifests). This makes the classification reproducible and independent of the specific scenarios.

Axis A (load tier). Given a benchmark run with requested rate λ and concurrency c: (i) determine the saturation rate $\lambda_{\max}$ for the target endpoint from a ramp in which requested RPS is increased until achieved RPS plateaus; (ii) compute $\rho = \lambda/\lambda_{\max}$; (iii) assign the tier whose normalised-load regime contains ρ (Section 4.2). Where $\lambda_{\max}$ is not separately measured, the absolute RPS/thread thresholds of Table 3 are used as a testbed-calibrated proxy.

Axis B (coordination). Given the entry endpoint r of a service, construct the invocation tree of one request, either from distributed traces or from the call code, counting repeated calls with their multiplicity; $\varphi(r)$ is its number of edges and $H(r)$ its number of levels (Section 4.1). The service is placed in the peer-to-peer category if some execution path of its call code contains a call to another instance of the same service or a call back to a service already on the path, and in the federated category otherwise. Applying this rule to the suite yields UC1–UC3 in the federated category and UC4–UC6 in the peer-to-peer category, with the per-endpoint values of φ and H listed in Table 2; Table 3 summarises both axes.

Table 3. Two-axis taxonomy: definitions, thresholds, and use-case mapping. A run's Axis A tier is the higher of the tiers implied by its offered rate and by its concurrency, since the normalised load $\rho = \lambda/\lambda_{\max}$ responds to both; a demonstrated rate shortfall (achieved below 95% of requested) overrides both and places the run in A4.

Axis	Tier/Category	Criteria	Use Cases
A (Scale)	A1: Baseline	RPS $\leq$ 10 and threads $\leq$ 10	All
	A2: Moderate	RPS $\in (10, 50]$ or threads $\in (10, 20]$	All
	A3: High	RPS $\in (50, 200]$ or threads $\in (20, 50]$	All
	A4: Extreme	RPS > 200, threads > 50, or achieved $< 95\%$	UC2, UC4, UC6
B (Coord.)	Federated	Calls only to distinct downstream services; $\varphi \in \{0, 1, 2\}$	UC1, UC2, UC3
	Peer-to-peer	Some path calls a peer instance or calls back; $\varphi \in \{0, \ldots, 6\}$	UC4, UC5, UC6

4.5. Expected Performance Signatures

These hypotheses are not merely intuitive; they follow from two well-established results in queueing and distributed-systems theory. First, for an open system whose service utilisation is ρ, classical queueing analysis (the Pollaczek–Khinchine/Kingman approximation for the G/G/1 queue) gives a mean waiting time that grows as $\rho/(1-\rho)$, and the high percentiles of the waiting-time distribution diverge super-linearly as $\rho \to 1$. Because Axis A is defined directly in terms of ρ (Section 4.2), tail latency should be roughly flat at low tiers and then rise sharply as the Extreme tier approaches saturation. Second, Dean and Barroso's tail-at-scale analysis [40] shows that when a request must wait for φ downstream calls, the probability that at least one is slow compounds with φ, so the end-to-end tail is governed by the maximum over the fan-out. Combining the two: expected P99 is an increasing function of both ρ (Axis A) and the coordination-amplification factor φ (Axis B), and the two compound, since every additional invocation per request both raises sidecar utilisation and adds a mesh hop to the critical path. The steepest growth is therefore predicted in the high-ρ, high-φ corner of the grid. Concretely, we expect low-φ workloads to hold relatively stable tail latencies across Axis A tiers, because each request adds little load to the sidecars. High-φ workloads—in our suite the peer-to-peer scenarios UC4 and UC6—should instead show steeper tail growth under scaling, as each request triggers lateral peer calls that simultaneously raise sidecar utilisation and compound the tail-at-scale effect, producing queueing cascades. The A4 tier is where we expect the two axes to interact most visibly:

peer-to-peer workloads may cross into a qualitatively different regime ($\rho \to 1$ saturation, in which achieved RPS falls below requested RPS), while federated workloads degrade far more gracefully. Figure 3 visualises these hypothesised signatures across the A×B grid; Section 8.2 reports the extent to which each is borne out by the data (the peer-to-peer A4 saturation is confirmed; federated stability is confirmed only up to A3, the highest tier to which the heavy federated scenario was driven; and the predicted A3 transition appears as the onset of the matched-weight premium reported in Section 8.3).

		Axis A			
		A1	**A2**	**A3**	**A4**
Axis B	Federated	Low tail stable	Moderate stable	Mild increase	Moderate increase
	Peer-to-peer	Low tail stable	Rising tail	Significant inflation	Severe inflation

Figure 3. Expected performance signatures in the A×B grid. Colour intensity indicates expected P99 latency severity.

5. Scenario Suite and Workload Phases

5.1. Scenario Descriptions

We instantiate the two-axis taxonomy across six scenarios that exercise distinct microservice communication topologies, summarised visually in Figure 4. Each scenario is deployed as a set of Kubernetes microservices with Istio sidecar injection enabled. The algorithmic pseudocode, per-endpoint processing complexity analysis, and per-request latency decomposition for all six use cases are provided in Appendix D, Appendix E and Appendix F respectively.

Concrete applications behind the six scenarios.

Although the taxonomy is stated abstractly, each scenario is a stripped-down instance of a real IoT-to-cloud application from the deployments the testbed was built for; Table 4 gives the mapping, and we name the concrete roles here so the reader carries them through the abstract discussion that follows. UC1 is a canary rollout of an acoustic anomaly-detection model for predictive maintenance (two model versions v1/v2 behind a weighted route, with a downstream rating service). UC2 is an electric-vehicle charging swarm: a charging aggregator (Actor-A) fans a forecast request out to charging-station actors (Actor-B, Actor-C), and the deployment literally instantiates three charging instances. UC3 is a visual-inspection pipeline: a detection service analyses an image and forwards to a result store. UC4 is cooperative acoustic monitoring: two analysis peers split the incoming audio and exchange intermediate results before classification. UC5 is a dual-path scheduling application (image inspection plus a sensor-driven scheduler) with an inspection call-back. UC6 is wind-turbine acoustic diagnostics: three turbine diagnostic nodes (node1/2/3) each decode audio and exchange results with their peers to reach a collective decision. These domains are representative because each embodies one of the canonical coordination motifs a swarm-backing platform exhibits—centralised rollout, aggregation fan-out, staged pipeline, pairwise exchange, dual vertical/lateral paths, and full-mesh consensus—and it is the motif, not the specific domain, that determines how the service mesh scales.

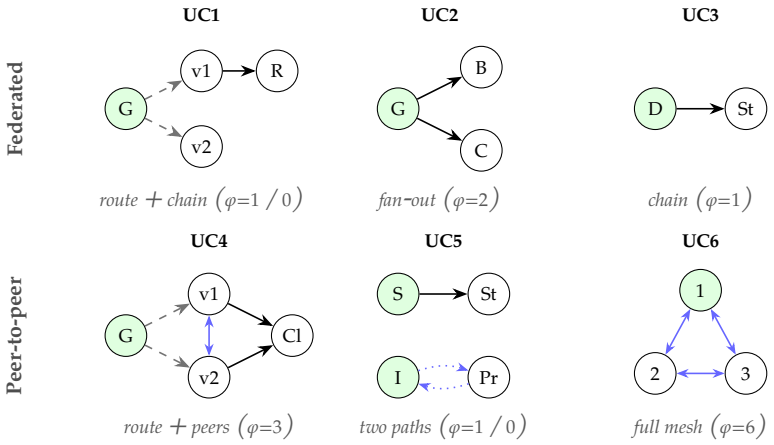


Figure 4. Communication topologies of the six use cases, with the coordination-amplification factor φ of each measured entry endpoint (Table 2; where a scenario has two values, they refer to its two branches or entry paths). Solid black arrows are service-to-service calls made on every request. Dashed arrows are *weighted routes*: in UC1 and UC4, G is the Istio route that sends each request to exactly one of v1 or v2, so it never calls both and no parallel call is implied, whereas in UC2 G is the gateway actor, which calls both actors. Blue double arrows are peer calls between instances of the same service; the dotted arrows of UC5 show the inspection call-back, present in the call code but not triggered by the benchmark input. Green nodes are the endpoints at which Fortio requests enter. Within a request, calls are issued one after another with a blocking client (Section 4.1); responses return along the reverse path and are omitted. Key: R = rating, B/C = charging-station actors, D = detection, St = result store, Cl = classification, S = scheduler, I = inspection, Pr = object processing, 1/2/3 = diagnostic nodes.

Table 4. The six scenarios as instances of real IoT-to-cloud applications, with the coordination motif each exercises and the concrete meaning of the node names used throughout the paper.

UC	Application Domain	Coordination Motif and Concrete Node Roles
UC1	Acoustic anomaly detection (predictive maintenance)	Weighted canary rollout; model versions v1/v2, downstream rating service
UC2	Electric-vehicle charging swarm	Aggregation fan-out; Actor-A aggregator → Actor-B/C charging stations (three instances)
UC3	Visual-inspection pipeline	Sequential chain; detection service → result store
UC4	Cooperative acoustic monitoring	Split-route peer exchange; analysis peers exchange, then classify
UC5	Dual-path scheduling	Vertical schedule path plus a lateral image-inspection feedback loop
UC6	Wind-turbine acoustic diagnostics	Full-mesh consensus; node1/2/3 decode audio and exchange with peers

From device swarms to service topologies.

The scenarios deliberately contain a small number of services, which warrants explanation given our swarm-scale motivation. In an IoT-to-cloud continuum, a swarm of N devices does not map to N cloud-side services; rather, the many devices funnel their requests into a far smaller set of coordination, inference, and aggregation services, and it is the topology and amplification among those services—not the raw device count—that governs how the service mesh scales. The device population enters our model through the offered load axis:

a swarm of N devices each emitting requests at rate r presents $N \cdot r$ RPS to the entry service, which is exactly what Axis A varies. What remains is to capture the canonical coordination motifs a swarm-backing platform exhibits, and the six scenarios are chosen to span them: centralised version rollout (UC1), one-to-many aggregation/fan-out (UC2), staged pipelines (UC3), cooperative pairwise exchange (UC4), dual vertical/lateral paths with feedback (UC5), and full-mesh collective inference (UC6). Thus a compact suite of services, exercised at swarm-scale request rates and across these motifs, reproduces the mesh-level pressures of large swarm deployments without requiring thousands of physical devices.

5.1.1. UC1: Weighted-Routing Inference

A single inference service is deployed in two versions (v1 and v2), with Istio traffic splitting routing 50% of requests to each version. Each request is routed by Istio to exactly one version; both versions decode the audio payload, which is the heavy processing class of Appendix F, after which v1 calls a downstream rating service and v2 answers directly ($\varphi = 1$ and $\varphi = 0$, Table 2). This scenario represents the simplest federated pattern: a route that selects one version per request, followed by at most one downstream call.

5.1.2. UC2: Multi-Actor Fan-Out

A coordination service composed of three actors, each deployed as a separate microservice with forecast and result endpoints. The primary actor is the entry endpoint at which external requests arrive and from which the fan-out originates, so it is labelled **G (gateway)** in Figure 4, as the entry is in UC1; unlike UC1's route, which selects one version per request, the gateway actor calls both secondary actors on every request. It is referred to as the gateway actor here, reserving Actor-B and Actor-C for the two secondary actors it calls. In the charging application these are a charging aggregator and two charging-station actors respectively. The gateway forwards each request to both secondary actors, whose `/result` endpoints acknowledge without calling further, giving a one-to-many fan-out of degree two ($\varphi = 2, H = 1$). The scenario stresses the mesh with a widened fan-out and per-actor latency variation rather than with depth.

5.1.3. UC3: Sequential Service Chain

A two-stage processing pipeline in which a front-end detection service analyses an incoming image and forwards the result to a back-end *result store*. In the visual-inspection application that the scenario abstracts, this second service persists each detection result: it performs a single JSON write and returns an acknowledgement, and its only role in the benchmark is to receive that one call. (The deployed endpoint path is `/register`, which is why the pseudocode of Appendix E retains that name.) The pattern is a chain, detection → result store, with Istio VirtualServices managing the routing between the two stages. The detection service issues one call, so $\varphi = 1$ and $H = 1$, and counting the request from the load generator each request traverses two mesh hops. The v1 branch of UC1—route to v1, then v1 → rating service—has exactly the same structure, so UC3 and the v1 branch of UC1 are both two-hop chains; UC1 differs only in the weighted route to v1 or v2 that precedes its chain.

5.1.4. UC4: Split-Route Peer Exchange

An analysis service deployed in two versions (v1, v2), with Istio splitting traffic 50/50 between them. Unlike UC1–UC3, the two analysis versions exchange intermediate results with each other as peers before calling a classification service, creating a cooperative processing loop.

How UC4 differs from a chain.

A request reaches v1 or v2 through the weighted route; the selected version sends its features to the other version, that version forwards them once more (TTL of 1, Section 4.1), and the selected version then calls the classifier, so that $\varphi = 3$ and $H = 2$ (Table 2). Two properties distinguish this from a chain. First, the number of peer invocations is set by the TTL rather than by the deployment: raising it adds further forwards, and hence further invocations, without adding a service. Second, the peer calls land on the same two analysis versions that also receive entry traffic, so that for every entry request those sidecars receive three calls rather than one, against the same replica set (Section 6.2); in a chain such as UC3 each service receives the entry rate exactly once. It is this concentration of load on shared sidecars, rather than the length of any single path, that produces the queueing reported at saturation in Section 8.6.

5.1.5. UC5: Dual-Path Pipeline with Call-Back

UC5 abstracts a charge-scheduling application with two independent entry endpoints, each forming its own path. On the scheduling path, a charging station posts a short array of sensor readings to `/schedule`; the scheduler, deployed in two versions split 50/50, computes a summary statistic over the array—a mean in v1 and a standard deviation in v2, which is why the two versions differ in cost while sharing a topology—and stores the resulting slot assignment in a result store, so that $\varphi = 1$. On the inspection path, a camera frame is posted to `/check`; the inspection service decodes and scores the frame and, only if the score falls below a threshold, hands it on to the object-processing service and to the next pipeline stage, whereupon the object-processing service marks the frame as processed and calls `/check` back. That call-back is what places UC5 in the peer-to-peer category (Section 4.3). With the fixed test image used throughout the benchmark, however, the score clears the threshold, so the measured `/check` requests return after the decode without triggering the call-back ($\varphi = 0$, Table 2). UC5 therefore belongs to the peer-to-peer category by its call structure, while its measured requests carry low amplification—one reason the analysis is stated in terms of φ rather than of category.

5.1.6. UC6: Full-Mesh Peer Amplification

Three processing nodes, each deployed as a separate microservice, perform local inference on incoming data and exchange results with both peer nodes for collective decision-making. This is the scenario with the highest amplification in the suite: each request triggers two peer calls, and with the deployed TTL of 1 each of those peers forwards the result once more to its own two peers, giving $\varphi = 2 + 4 = 6$ and $H = 2$ (Table 2). The full-mesh topology maximises peer traffic and exposes the mesh to the highest per-request overhead of any scenario we benchmark.

5.2. Fortio Phase Design

We first fix terminology, since it governs how the 51 figure is obtained. A *phase* is one Fortio invocation against a single endpoint at a fixed (RPS, threads, calls/duration) setting; a phase produces exactly one Fortio JSON result, which we call a *run*. A *campaign* is the ordered sequence of phases applied to one scenario, ramping from low to high load. For scenarios with several externally exercised endpoints (e.g., UC2's three actor endpoints `/forecast`, `/charge`, `/result`), each endpoint is driven by its own phase at every tier, so one tier contributes multiple runs; the "×3"/"×2" annotations in Table 5 denote this per-endpoint replication. Each phase (row) is characterised by its target RPS, concurrent threads (Fortio `-c`), and either an exact call count (Fortio `-n`, used where reproducible run length matters) or a timed duration of 30–60 s (Fortio `-t`, used for sustained-load percentiles); the **Calls** column reports the former

where applicable. Table 5 summarises the phase design, and the complete list of all 51 runs with their achieved metrics is given in Appendix A (Table A1).

Table 5. Scenario workload phases mapped to Axis A tiers. Duration is 30–60 s per phase (short phases for exact call counts).

UC	Phase	RPS	Threads	Calls	Tier
UC1	Warm-up	10	10	100	A1
	Baseline	10	5	600	A1
	Moderate	50	15	3000	A2
	High	100	25	6000	A3
	Peak	150	35	8995	A3
	Stress	200	50	12,000	A3
UC2	Baseline (×3)	10	5	600	A1
	Moderate (×3)	50	10	3000	A2
	High (×3)	100	20	6000	A3
	Peak (×3)	150	30	6750	A3
	Stress-1 (×3)	200	50	9000	A3
	Extreme (×3)	300	60	18,000	A4
UC3	Baseline	1	30	60	A1
	Low	5	2	150	A1
	Moderate	50	10	3000	A2
	High	100	25	30,000	A3
	Stress	200	50	3000	A3
UC4	Moderate	50	25	2250	A2
	High-1	100	50	3000	A3
	High-2	150	75	4500	A3
	Peak	200	100	12,000	A3
	Extreme-1	300	150	18,000	A4
	Extreme-2	400	200	–	A4
UC5	Baseline (×2)	5	2	150	A1
	Moderate (×2)	50	10	3000	A2
	High (×2)	100	25	30,000	A3
	Stress (×2)	200	50	3000	A3
UC6	Baseline	25	5	750	A2
	Moderate	50	10	1500	A2
	High	100	25	2000	A3
	Peak	200	50	–	A3
	Extreme-1	300	100	–	A4
	Extreme-2	400	150	–	A4

5.3. Rationale and Run Accounting

Parameter selection. The RPS, thread, and call values were chosen, not arbitrarily, to (i) span the four normalised-load regimes of Axis A from quiescent to saturated for each scenario, (ii) keep the concurrency-to-rate ratio in a realistic band by scaling threads roughly with RPS—Fortio's `-c` sets the number of connections the generator holds open, each served by its own goroutine, so the thread count fixes how many requests may be in flight simultaneously and is independent of the target rate `-qps`; it is the concurrency term c in ρ, and it is why a low-rate run at high concurrency still places queueing pressure on a sidecar and is tiered accordingly (Table 3)—and and (iii) place the highest phases at or beyond each scenario's measured saturation point λ_{max} so that the Extreme tier actually probes $\rho \to 1$. Because λ_{max} differs across scenarios—a heavy audio-decoding peer-to-peer service saturates at a far lower absolute RPS than a negligible-compute acknowledgement endpoint—the absolute RPS ladders differ across use cases even though they target the

same relative regimes. This is why, for example, UC6 reaches saturation behaviour already around 200 RPS whereas federated UC2 endpoints remain unsaturated at 300 RPS.

Which scenarios were driven to A4, and why. A4 coverage is partial, and we state the selection rule and its consequences explicitly rather than claiming the omitted runs would have been uninformative. The rule was to bracket the design space along Axis B: we drove to A4 the highest-amplification *peer-to-peer* scenarios (UC4 and UC6, selected a priori as the configurations most likely to saturate) together with the most heavily replicated *federated* scenario (UC2) as a reference, while UC1, UC3, and UC5 were run only up to A3 to bound total experiment time. We do not claim the omitted A4 runs would reveal nothing new—that cannot be known without running them. Their absence has one concrete consequence for the analysis, which we flag wherever it matters: because the only heavy federated scenario (UC1) was not driven to A4, the federated-versus-peer contrast at the Extreme tier cannot be read from the category aggregates, and the controlled evidence for the coordination effect is instead the matched-weight comparison of Section 8.3, where UC1 and UC4 are compared at identical load up to A3. Completing A4 coverage for all six scenarios—and in particular driving UC1 into saturation—is therefore listed as future work (Section 11.2) and would enable a fully symmetric A×B comparison at the Extreme tier.

How the 51 runs are obtained. Summing the per-scenario run counts of Appendix A gives the total directly: UC1 contributes 7 runs, UC2 contributes 18 (three actor endpoints across six tiers), UC3 contributes 5, UC4 contributes 6, UC5 contributes 9 (two entry endpoints across the ramp), and UC6 contributes 6, for $7 + 18 + 5 + 6 + 9 + 6 = 51$. UC2 dominates the count because its three independent actor endpoints are each benchmarked at every tier, which also supplies within-tier replication. The mapping from these runs to A×B cells (used in all aggregate plots) is: each run's Axis A tier follows its load regime (Table 5) and its Axis B category follows its scenario (Section 4.4); the resulting per-cell run counts are reported in Section 8.1.

6. Experimental Setup and Data Pipeline

6.1. Testbed

All experiments were conducted on a single-node Kubernetes cluster running Kubernetes v1.29 on an x86_64 machine with 16 CPU cores and 32 GB RAM (Figure 5). This single-node design is a deliberate methodological choice rather than a convenience: by colocating all pods on one host we remove inter-node network latency and topology variability, so that the latency differences we measure across the A×B grid are dominated by the service-mesh data plane (Envoy sidecar processing and intra-host queueing) and the fixed application processing, not by fluctuating physical-network conditions. In effect, the testbed measures a lower bound on mesh-induced tail inflation: a multi-node deployment would add network delay on top of—but would not remove—the sidecar and queueing effects characterised here. The consequences of this choice for external validity (absolute latency values, saturation thresholds) are examined in Sections 9.5 and 10.2. Each use-case scenario is deployed in a dedicated Kubernetes namespace (`uc1-inference`, `uc2-fanout`, `uc3-chain`, `uc4-peerexchange`, `uc5-dualpipe`, `uc6-fullmesh`) with Istio sidecar injection enabled at the namespace level.

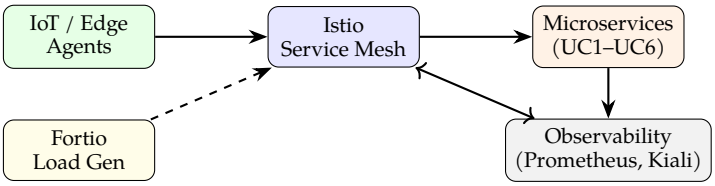


Figure 5. High-level architecture of the benchmarking testbed. In our experiments the abstract roles map concretely as follows: the *IoT/edge agent* role is played by the in-cluster Fortio load generator, which replays fixed audio/image/JSON payloads representative of edge devices (drones, charging stations, sensor arrays) through the Istio ingress gateway; the *microservices* block is instantiated by the six use-case deployments of Section 5 (e.g., the inference/rating services of UC1, the multi-actor fan-out of UC2, the full-mesh diagnostic nodes of UC6); and *observability* components collect request-level telemetry. Fortio runs as a meshed pod so that all generated traffic traverses the Envoy data plane.

6.2. Service Mesh Configuration

We deployed Istio v1.22 in its default profile. Automatic sidecar injection was enabled for every benchmark namespace through the `istio-injection: enabled` label, and traffic routing was governed by VirtualServices: for UC1 and UC4, requests are split 50/50 between service versions v1 and v2 using DestinationRules with version-based subset selectors. Each VirtualService sets a 5-s timeout on service-to-service calls to prevent indefinite hangs under load, while an Istio ingress gateway—fronted by HTTPRoute (Gateway API) or VirtualService route rules—exposes service endpoints to the Fortio load generator. Istio's default telemetry collects request-level metrics (request counts, latency histograms, response codes) and exports them to Prometheus. Service deployments use a fixed, small number of replicas throughout: each primary or versioned service runs with 2–3 replicas so that Istio's client-side load balancing across a destination is genuinely exercised, while terminal acknowledgement services (rating, classification, result store) run with a single replica. For example, UC4 deploys three replicas each of `analysis-v1` and `analysis-v2` behind a single Kubernetes Service (`analysis-service`). Replica count is deliberately held fixed across all runs so that the factors we vary are offered load (Axis A) and coordination structure (Axis B), not horizontal capacity; consequently the saturation we observe is at the configured replica level, and how tail latency responds to a replica-count sweep at fixed topology is an important dimension we leave to future work (Section 11.2).

Where the cross-cutting concerns actually appear. Section 1.1 motivates service meshes by the cross-cutting concerns they absorb—load balancing, circuit breaking, retries, mutual TLS and distributed tracing. Table 6 states where each of these is exercised in the scenario suite, and, equally important, where it is merely *configured*. The distinction matters for reading our results: a concern that is active in the data path contributes to every latency figure we report, whereas one that is configured but never triggered contributes only its standing cost. In particular, because all 51 runs returned HTTP 200 (Section 7.3), the retry and circuit-breaking policies were never invoked; what we measure is their presence in the filter chain, not their behaviour under failure. Exercising them deliberately, through fault injection, is a distinct study and is listed as future work (Section 11.2).

Table 6. Istio cross-cutting concerns and where the scenario suite exercises them. "Stressed" means the concern is active in the measured data path and varies across the grid; "configured" means it is deployed and contributes standing cost, but the benchmarks never drive it to act.

Concern	Status	Where It Appears in UC1–UC6
Weighted routing	Stressed	VirtualServices split traffic in every scenario: UC1 and UC4 50/50 across versions, UC3 70/30, UC5 50/50 across schedulers, UC2 40/30/30 across actors, UC6 40/40/20 across diagnostic nodes
Load balancing	Stressed	DestinationRules balance across the 2–3 replicas of each primary or versioned service; in the peer-to-peer scenarios the peer calls share that same replica set
Request timeout	Stressed	5 s ceiling on every service-to-service call; binds only near saturation, where it bounds the tail we report
Telemetry/tracing	Stressed	Per-hop metric collection on every call in all six scenarios, exported to Prometheus
Mutual TLS	Configured	Sidecar-to-sidecar encryption is on throughout, so its cost is inside every reported L, but it is not varied and no policy is exercised
Circuit breaking	Configured	Policies are deployed but never opened: no run produced a non-2xx response
Retries	Configured	Never triggered, for the same reason

6.3. Load Generation

Fortio [20] is an open-source load testing tool developed by the Istio team, designed specifically for benchmarking HTTP and gRPC services. We chose Fortio for its precise rate control, built-in percentile reporting, and native JSON output format. One naming point, since it affects how our archived data should be read: Fortio labels its rate field `QPS` (queries per second) in both its command-line interface and its JSON output. We use *RPS* (requests per second) throughout the text for accuracy, but the deposited dataset and analysis scripts retain Fortio's own `qps` field names so that they remain directly comparable with raw Fortio output; the two refer to the same quantity.

For each benchmark phase, Fortio is invoked with a target rate (`-qps`), a number of concurrent connections or threads (`-c`), and either a duration (`-t`, e.g., `1m0s`) or an exact number of calls (`-n`); POST payloads are sent with `-content-type` set to `application/json`, and custom headers are supplied through `-H` as needed. Fortio runs from within the cluster (as a Kubernetes pod with Istio sidecar) to ensure that all traffic traverses the mesh data plane. Each run produces a JSON result file containing detailed latency histograms, percentile values, achieved RPS, response code distributions, and connection statistics.

6.4. Data Extraction Pipeline

We developed a Python-based (Python 3.10.12, pandas 2.3.3) data extraction pipeline (Figure 6) that processes Fortio JSON results into a structured Excel workbook for analysis. Each result file is first parsed to extract requested and actual RPS, thread count, latency percentiles (P50, P75, P90, P99, P99.9), average/min/max/stddev latency, response codes, and connection statistics. From these, the pipeline derives a tail-to-median ratio (P99/P50) and a latency×throughput product, then merges the per-UC Excel files into a consolidated workbook annotated with Axis A tier and Axis B category. A companion plotting script

finally generates the heatmaps, line plots, box plots, and scatter plots used in the A×B grid analysis.

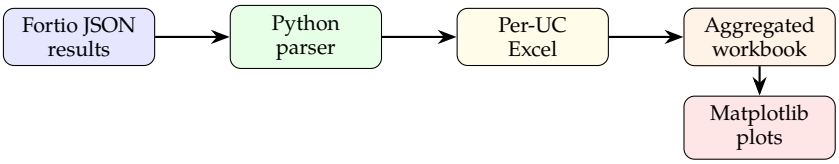


Figure 6. Data extraction and analysis pipeline. Fortio JSON results are parsed, aggregated into Excel workbooks with taxonomy annotations, and visualised.

7. Metrics and Statistical Treatment

This section defines the metrics extracted from each Fortio run and the statistical treatment applied before aggregation. Table 7 lists every metric, its unit, and its source; the subsections below motivate the choices that matter most for the analysis.

Table 7. Metrics definitions and sources.

Metric	Unit	Description
Requested RPS	req/s	Target rate specified to Fortio
Actual RPS	req/s	Achieved rate reported by Fortio
P50 latency	s	50th percentile of request latency
P90 latency	s	90th percentile of request latency
P99 latency	s	99th percentile of request latency
P99/P50 ratio	–	Tail-to-median amplification factor
Error rate	%	Fraction of non-2xx responses
Threads	–	Concurrent Fortio connections

7.1. Load Fidelity: Requested vs. Actual RPS

A critical prerequisite for valid benchmarking is that the load generator achieves the intended request rate. We report both the *requested RPS* (the target rate specified to Fortio) and the *actual RPS* (the rate achieved during the run). Significant divergence between these values indicates that the system under test (or the load generator) has reached a saturation point. In our dataset, most runs achieve >99% of the requested RPS, with notable exceptions in UC4 (400 RPS requested, 315.5 achieved) and UC6 (300–400 RPS requested, 185 achieved), signalling resource saturation at the A4 tier.

7.2. Latency Distribution: P50, P90, P99

Following Dean and Barroso [40], we report three latency percentiles for each run. The P50 (median) is the typical latency experienced by half of all requests; the P90 marks the onset of tail behaviour, the latency below which 90% of requests complete; and the P99 captures the "tail" that disproportionately affects user experience in fan-out architectures, the latency below which 99% of requests complete. We also report P99.9 where available, and compute the *tail-to-median ratio* (P99/P50) as a normalised measure of distributional spread.

7.3. Reliability: Response Codes and Error Rate

Each Fortio run records the distribution of HTTP response codes, from which we compute the success count (requests receiving a 200-series response) and the error rate (the fraction receiving non-2xx responses). In our dataset, all 51 runs achieved a 0% error rate: every request received an HTTP 200 response, with no non-2xx codes, timeouts, or dropped connections, even at A4 stress levels. We interpret this cautiously. A 200 response establishes *transport- and application-response success*—the request traversed the mesh, reached the backend, and returned a well-formed response within the 5-s timeout—but does not by itself

verify the semantic correctness of the returned payload, which we did not check against ground truth. The claim we make is therefore the narrower, well-supported one: Istio's timeout and circuit-breaking configuration prevented error propagation and connection failure under all tested loads, so the latency degradation we report occurs without a corresponding loss of availability—not the stronger claim of end-to-end application correctness.

7.4. Aggregation Strategy

For cross-cell comparisons in the A×B grid, we aggregate using the median of per-run metrics within each cell. The median is preferred over the mean because latency distributions are typically right-skewed, and the median is robust to outliers from individual runs affected by transient system events (e.g., garbage collection, background pod scheduling).

When a cell contains runs from multiple sub-services (e.g., UC2's three actors at the same tier), all runs contribute to the cell's aggregate, providing within-cell replication.

7.5. Threat Controls

To mitigate measurement artefacts, we applied four controls. Each scenario begins with a low-load warm-up phase that populates connection pools and triggers JIT compilation in the Envoy proxies. Benchmark phases within a scenario run sequentially rather than concurrently, avoiding cross-phase interference, and each scenario occupies a dedicated Kubernetes namespace, preventing resource contention between scenarios. Finally, every phase issues at least 100 requests (typically 600–30,000) so that percentile estimates are statistically stable.

8. Results

This section presents the empirical findings from 51 benchmark runs across the A×B grid. To keep the analysis navigable, it follows a fixed progression: we first report coverage (Section 8.1); then the headline tail-latency trend aggregated by Axis B category across Axis A tiers (Section 8.2); then the isolation of mesh from application cost that licenses interpreting that trend as infrastructure overhead (Section 8.4); then within-cell distributions and the throughput–tail relationship (Sections 8.5 and 8.6); and finally two scenario deep-dives, one federated and one peer-to-peer (Section 8.7). Throughout, "cell" denotes one (Axis B category × Axis A tier) combination, and aggregate values are per-cell medians (Section 7.4).

8.1. Benchmark Profile

Figure 7 shows the distribution of benchmark runs across the A×B grid. The federated category (UC1–UC3) contributes 30 runs distributed across all four tiers, while the peer-to-peer category (UC4–UC6) contributes 21 runs. The A3 (High) tier is the most densely populated, reflecting our emphasis on characterising performance in the operationally relevant peak-load regime. The A4 (Extreme) tier contains runs only from UC2, UC4, and UC6, as these scenarios were specifically designed to probe saturation behaviour.

Run counts by coordination category (Axis B) and scale tier (Axis A)

	A1	A2	A3	A4
Federated	7	5	15	3
Peer-to-peer	3	4	7	7

Figure 7. Heatmap of benchmark run counts by Axis B coordination category and Axis A scale tier.

8.2. Tail Latency Under Scaling

Figure 8 presents the median P90 and P99 latencies for each coordination category across Axis A tiers, with tiers assigned strictly by the Table 3 rules (so phases with more than 50 threads, or with achieved throughput below 95% of requested, fall in A4). Two cautions apply before these aggregates are read as a topology comparison. First, the per-category medians mix application weights: the federated aggregate is dominated by UC2's eighteen negligible-compute acknowledgement runs, so its low A4 median (17.5 ms) reflects those light endpoints rather than a heavy federated workload. Second, the federated heavy scenario (UC1) was driven only to 200 RPS/50 threads and never into saturation, whereas the heavy peer-to-peer scenarios were pushed to 400 RPS/200 threads; the two categories were therefore not driven to comparable ρ. The weight-mixed cell medians are 70.6, 24.2, 44.5, 17.5 ms (federated) and 67.1, 76.7, 100.0, 666.4 ms (peer-to-peer) at A1–A4. Because these conflate application weight, we do not treat them as the controlled result; the controlled comparison, at matched weight and load, is given in Section 8.3.

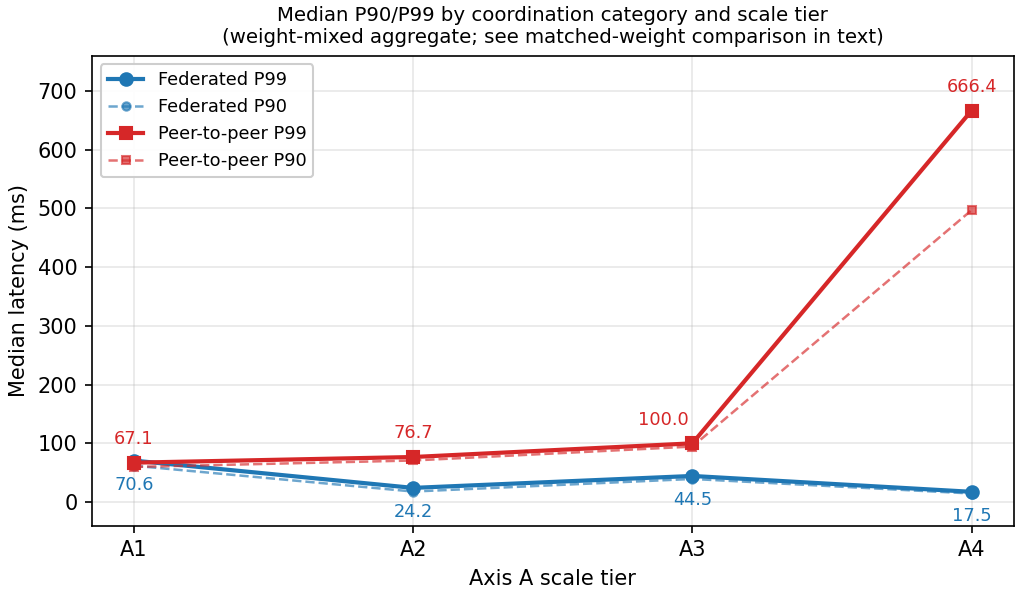


Figure 8. Median P90 and P99 latency versus Axis A scale tier for each Axis B coordination category (weight-mixed aggregate; tiers per Table 3). Because each category mixes application weights and the categories were not driven to comparable saturation, this view is descriptive; the controlled, matched-weight comparison is reported in Section 8.3.

Figure 9 displays the median P99 latency as a heatmap across the full A×B grid. The Peer-to-peer × A4 cell stands out at 666.4 ms, but it contains only heavy-compute peer-exchange runs driven into saturation, whereas the federated A4 cell contains only UC2's negligible-compute acknowledgements; the contrast therefore reflects application weight and saturation depth as much as coordination structure, which the matched-weight comparison (Section 8.3) disentangles.

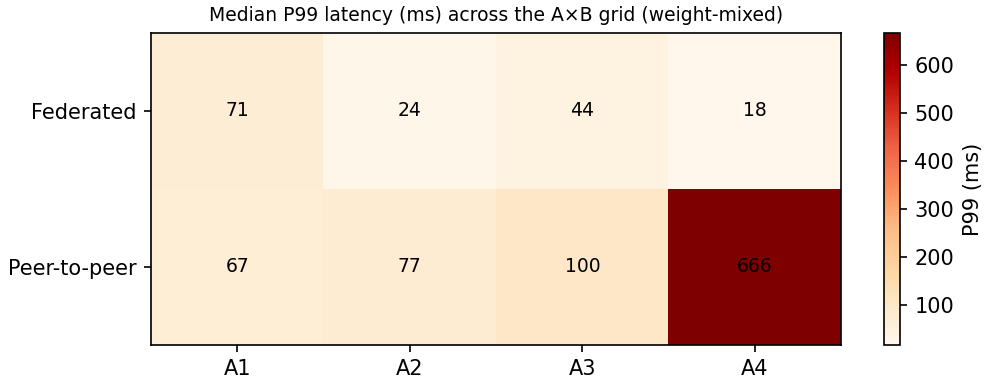


Figure 9. Heatmap of median P99 latency (ms) by Axis B coordination category and Axis A scale tier (weight-mixed; tiers per Table 3). The Peer-to-peer × A4 cell is the hotspot, but it contains only heavy-compute peer-exchange runs at saturation while the federated A4 cell contains only negligible-compute acknowledgements; Section 8.3 gives the weight-controlled comparison.

8.3. Coordination Effect at Matched Application Weight

Because the per-category aggregates above mix application weights, we isolate the effect of coordination structure by comparing two scenarios that share the same heavy audio-decoding entry endpoint: UC1 (federated—weighted routing with a single downstream rating call) and UC4 (peer-to-peer—split-route analysis with a lateral peer exchange). Both carry the HEAVY `librosa` decode (Appendix F), so a difference between them at equal offered load is attributable to coordination structure and the concurrency the pattern induces, not to application weight. Table 8 reports the pairing.

Table 8. Matched-weight, matched-load comparison. UC1 (federated) and UC4 (peer-to-peer) share the same HEAVY audio entry endpoint; both P99 values (ms) are read at equal offered RPS from Appendix A.

Offered Load	UC1 P99 (Fed.)	UC4 P99 (p2p)	Premium
100 RPS	117.2	139.7	1.19×
150 RPS	134.8	243.4	1.81×
200 RPS	237.8	345.1	1.45×

At matched weight and load the peer-to-peer premium is ≈1.2–1.8×: a real and consistent effect of coordination structure, but far from the order of magnitude the weight-mixed aggregates suggest. (UC4 additionally runs at higher thread counts than UC1 at the same RPS, so part of this premium is the extra concurrency the peer exchange induces; the concurrency-matched view below removes that objection.) The large effects seen elsewhere in this section are of two distinct kinds, which we separate deliberately: (i) *within-scenario* growth—UC6's P99 rises from 98 ms at 50 RPS to 1 121 ms at 400 RPS, ≈11× relative to its own baseline; and (ii) *throughput saturation*—the heavy peer-exchange scenarios are the only ones whose achieved RPS falls below requested (UC6 delivers 185 of a requested 400 RPS). Both are properties of the heavy peer-exchange corner of the grid under saturation rather than of coordination topology alone, and surfacing this interaction of load, structure, and application weight is precisely what the two-axis framework contributes.

Controlling for concurrency.

One objection to Table 8 is that the premium may be an artefact of concurrency, since at equal offered RPS the peer-to-peer scenario was driven with twice the threads of the federated one. The data admit the complementary comparison, which removes that objec-

tion by holding concurrency fixed and letting throughput vary (Table 9). At 25 threads the federated scenario sustains 100 RPS at a P99 of 117.2 ms while the peer-to-peer scenario sustains only 50 RPS at 100.0 ms; at 50 threads the figures are 200 RPS at 237.8 ms versus 100 RPS at 139.7 ms. In both pairs the federated scenario delivers exactly twice the throughput for a tail only 1.17–1.70× larger. The two views therefore bound the same effect from opposite directions: at matched throughput the peer-exchange pattern costs tail latency, and at matched concurrency it costs throughput. Taken together they indicate a genuine capacity cost of the coordination pattern rather than an artefact of how each scenario happened to be driven—though we note that neither pair is matched on both axes simultaneously, which only a dedicated thread- and rate-matched campaign could achieve (Section 11.2).

Table 9. Concurrency-matched view of the same two HEAVY-audio scenarios (UC1 federated, UC4 peer-to-peer). Holding the thread count fixed and letting throughput vary, the federated scenario delivers twice the RPS in both pairs.

Threads	UC1 RPS	UC1 P99 (ms)	UC4 RPS	UC4 P99 (ms)	Throughput Ratio
25	100	117.2	50	100.0	2.0×
50	200	237.8	100	139.7	2.0×

8.4. Isolating Mesh Overhead from Application Processing

A central concern when a benchmark mixes application logic (audio/image decoding) with infrastructure (the service mesh) is whether the reported latency trends reflect mesh behaviour or merely application computation. Our design isolates the two by construction, and the data let us quantify each component.

The application term is held constant. Every scenario is driven with fixed payloads—a single 1025 KB, 5 s WAV and a single 21 KB, 612×408 JPEG (Appendix F). For a fixed input the decoding cost (T_{audio}, T_{image}) is a deterministic, load-independent additive constant. Consequently, any change in latency as we move up the Axis A tiers cannot originate in application processing—it must come from the mesh data plane and from queueing. The load-dependent growth in tail latency reported in Section 8.2 is therefore predominantly an infrastructure effect rather than an artefact of heavier computation at higher load—with the caveat, quantified below, that near saturation this growth term can also absorb application-side CPU contention.

A pure mesh-overhead probe. The negligible-compute acknowledgement endpoints (UC2's `/result`; also `/rate`, `/classify`, `/register`, `/process`; Table A2) perform only JSON parsing and scalar operations, so a single-hop call to them measures the per-hop mesh latency L almost in isolation. Table 10 extracts L from UC2's single-hop `/result` endpoint across tiers: it climbs from ≈2.7 ms (P50)/4.7 ms (P99) at A1 to ≈9.4 ms/17.3 ms at A4. Because UC2 has zero application compute, this growth is entirely mesh and intra-host queueing—direct evidence that the sidecar data plane, not the workload, is what degrades under load.

Decomposing a saturating workload. UC6's entry endpoint costs $T_{\text{audio}} + 7L$ plus queueing (Appendix G). To make the application term explicit we estimate it from the median (not the tail) of a single heavy hop at baseline: $T_{\text{audio}} \approx \text{P50}_{\text{UC1,A1}} - L \approx 52.7 - 2.7 \approx 50$ ms, using the mesh floor L from Table 10. Subtracting this constant from UC6's P99 leaves the remainder shown in the right columns of Table 10, which rises from ≈48 ms at A2 to ≈1071 ms at A4. Two caveats bound the interpretation of that remainder. First, subtracting a load-independent constant from a percentile is valid only where the application term itself has no queue—that is, at low load; near saturation the remainder is an upper bound on mesh and queueing that may also absorb application-side CPU contention, consistent with the CPU-saturation outlier we attribute to the analytics service in Section 8.5, and

per-pod CPU series were not retained to separate the two. Second, the direct mesh floor measured on the negligible-compute UC2 `/result` endpoint (Table 10, left) is unaffected by this caveat, since those endpoints carry no application term. We therefore state the result conservatively: the constant-payload design and the ack-endpoint probe together attribute the load-dependent tail growth predominantly to mesh and queueing, while acknowledging an application-contention component near saturation that the present data cannot fully separate. The federated A4 cell, by contrast, is dominated by single-hop ack endpoints whose latency is essentially L alone, which is why its median is low.

Table 10. Per-hop mesh latency L measured on the negligible-compute, single-hop UC2 `/result` endpoint (zero application processing), and decomposition of UC6 `/analyze` P99 into a constant application term ($T_{\text{audio}} \approx 50$ ms, fixed payload) and a mesh+queueing remainder. The remainder grows ~22× while the application term is unchanged. Rows are labelled by offered load (RPS) because the two probe endpoints sit in different Axis A tiers at the same rate (UC2 stays sub-saturation; UC6 saturates); near saturation the remainder is an upper bound on mesh+queueing (see text).

	Mesh Floor *L* (UC2)		**UC6 `/analyze` P99**	
Load (RPS)	**P50 (ms)**	**P99 (ms)**	**App (ms)**	**Mesh+Queue (ms)**
10–50	2.7–2.9	4.7	50	48
100	4.8	12.5	50	89
200	7.6	15.5	50	300
300	9.4	17.3	50	646
400	–	–	50	1071

8.5. Distributional Behaviour

Box plots in Figures 10 and 11 reveal the within-cell variability of P99 latencies.

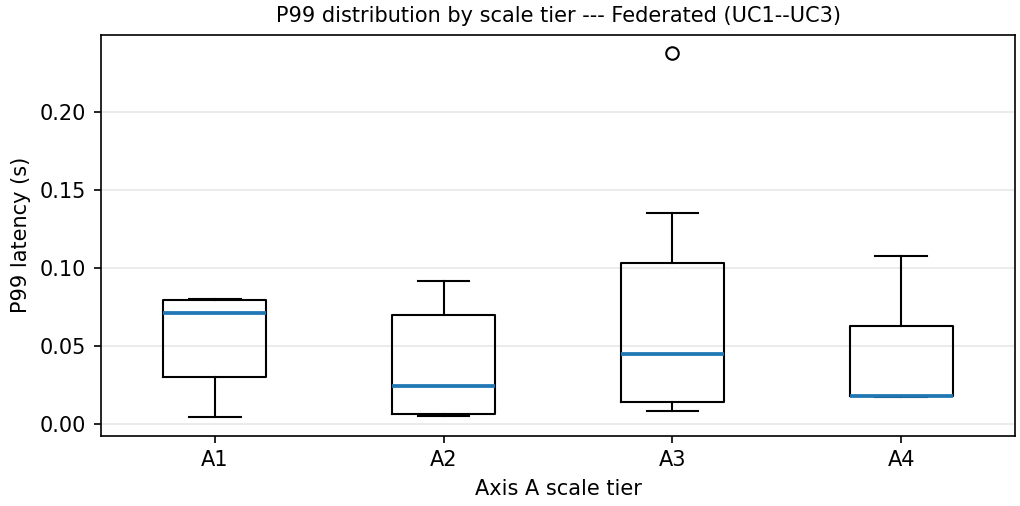


Figure 10. Box plot of P99 latency by Axis A scale tier for federated workloads (UC1–UC3). The distribution remains compact, with one outlier at A3.

For **federated workloads** (Figure 10), the interquartile range remains compact across all tiers, with a single outlier at A3 (UC1 at 200 RPS, P99 = 237.8 ms). This outlier corresponds to the highest-load phase of UC1, where the voice analytics service begins to saturate CPU resources.

For **peer-to-peer workloads** (Figure 11), the distributional spread increases dramatically with tier. At A4—which, with tiers assigned per Table 3, spans UC4 from 150 RPS/75 threads upward and UC6 from 200 RPS upward—P99 values range from 243 ms to 1121 ms,

with a median of 666.4 ms. The wide spread reflects the heterogeneity of the peer-to-peer scenarios' call structure and application weight: UC6's heavy audio nodes propagate results to peers through TTL-bounded forwarding (Appendix E), the most amplifying configuration ($\varphi = 6$), whereas UC4 ($\varphi = 3$) and UC5 ($\varphi \leq 1$ on its measured branches) amplify less and UC5's scheduler endpoint is only lightly loaded.

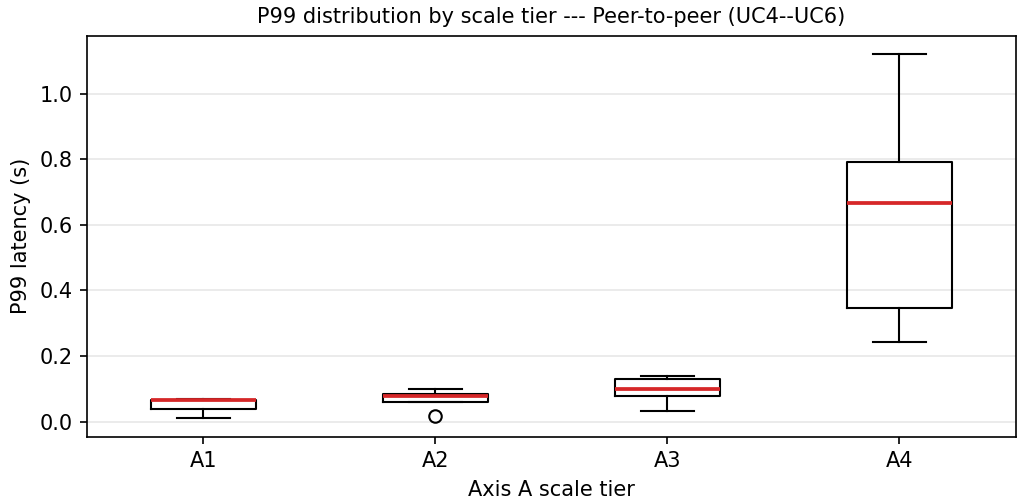


Figure 11. Box plot of P99 latency by Axis A scale tier for peer-to-peer cooperative workloads (UC4–UC6). The distribution widens substantially at A3 and A4.

8.6. Throughput–Tail Coupling

Figure 12 plots achieved RPS (not requested) against P99 latency for all 51 runs, with markers distinguishing coordination category and the four highest-latency runs annotated by use case and requested load. Because the abscissa is the delivered rate, a saturating run appears at its achieved throughput rather than its target—a distinction that is essential to reading the plot. Two regimes are visible. Below ∼150 RPS both categories cluster under 150 ms P99, peer-to-peer only slightly higher. The four runs with P99 above ≈0.6 s are all heavy peer-to-peer, but they fall into two qualitatively different groups. UC4 at requested 300 and 400 RPS keeps up with the offered load—achieving 296.9 and 315.5 RPS—while its P99 rises to 666.4 and 887.1 ms: throughput still scales, but the tail inflates. UC6 at requested 300 and 400 RPS behaves differently: its achieved throughput plateaus at 185.2 and 184.7 RPS, so these two points sit near 185 on the x-axis—well below 200, even though much more load was offered—while P99 climbs to 695.8 and 1 121.4 ms. Not all four high-latency points therefore have achieved RPS above 200; two of them are pinned at ≈185 precisely because the system cannot deliver more. This is the distinctive saturation signature: beyond the sustainable rate, additional offered load and concurrency no longer raise delivered throughput but accumulate in the queues, so the tail grows while the achieved rate is capped. The mechanism is the TTL-bounded peer forwarding of UC6 ($\varphi = 6$, Appendix E): each request's peer calls, and the further calls they trigger, converge on the shared Envoy sidecars, whose finite connection pools and processing capacity cap the sustainable request rate; once that cap is reached, extra concurrency lengthens sidecar queues rather than completing work. UC4's single-level exchange imposes a lighter amplification and so sustains higher throughput before the same effect sets in.

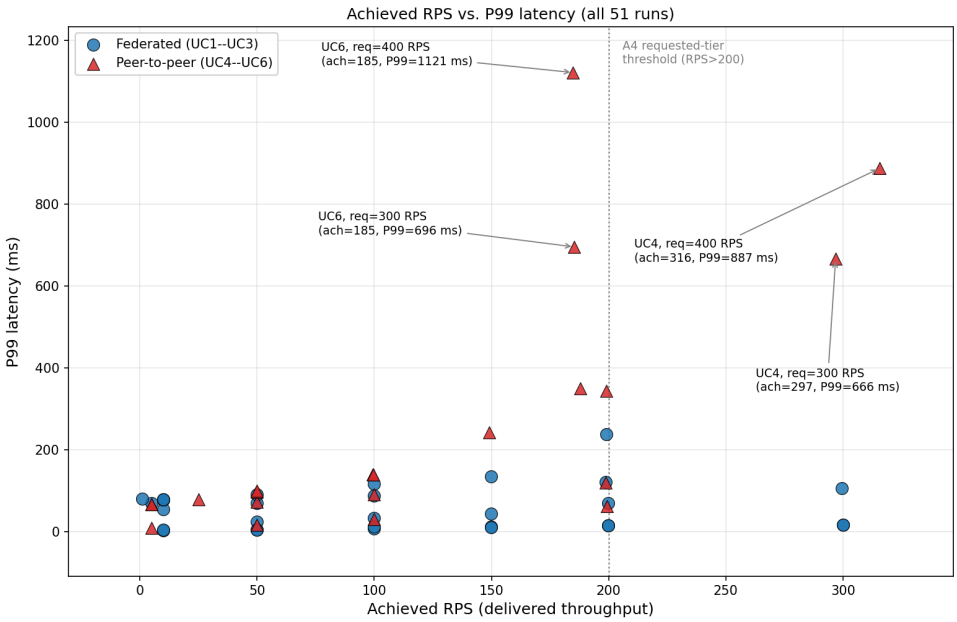


Figure 12. Achieved (delivered) RPS versus P99 latency for all 51 runs. Circles denote federated workloads (UC1–UC3), triangles peer-to-peer (UC4–UC6); the four highest-latency runs are annotated with their use case and requested load. Because the x-axis is achieved throughput, the two UC6 points sit near 185 RPS—they were offered 300–400 RPS but saturate below the delivered rate of the UC4 points, which keep up with their target.

8.7. Representative Scenario Deep Dives

Table 11 summarises the best and worst cells of the A×B grid by median P99 latency; because these cell medians mix application weights they are descriptive only, and the two deep dives that follow examine one federated and one peer-to-peer scenario individually.

Table 11. Summary of best and worst A×B cells by median P99 latency (tiers assigned per Table 3). These cell medians mix application weights and are descriptive only; the weight-controlled comparison is Table 8.

Rank	Cell	Dominant UC	Median P99 (ms)
Best	Federated × A4	UC2 (result endpoints)	17.5
2nd	Federated × A3	UC1 + UC2 + UC3	44.5
3rd	Federated × A1	UC1 + UC2 + UC3	70.6
Worst	Peer-to-peer × A4	UC4 + UC6	666.4
2nd	Peer-to-peer × A3	UC4 + UC5 + UC6	100.0
3rd	Peer-to-peer × A2	UC5 + UC6	76.7

8.7.1. Case Study 1: UC2 (Multi-Actor Fan-Out)—Federated Resilience

UC2 is the most extensively benchmarked federated scenario, with 18 runs across three sub-services (Actor-A, Actor-B, Actor-C) and six load tiers, and the three sub-services exhibit markedly different latency profiles. The primary actor's `/forecast` endpoint consistently shows the highest latencies (P99 = 55.0 ms at A1, 70.5 ms at A3, 107.2 ms at A4) because it fans out to the two secondary actors, a mini fan-out within the federated topology. The secondary actors' `/result` endpoints, performing only a simple acknowledgement with no downstream calls, stay fast throughout (P99 < 18 ms even at A4). This within-scenario variation illustrates that even within the federated category, the degree of downstream fan-out significantly influences tail latency. The A×B taxonomy correctly captures the aggregate behaviour, but designers should be aware of intra-scenario hotspots.

8.7.2. Case Study 2: UC6 (Full-Mesh Peer Amplification)—Peer-to-Peer Saturation

UC6 exhibits the most dramatic scaling behaviour. At A2 (50 RPS, 10 threads), P99 is a modest 98.2 ms, and at A3 (100 RPS, 25 threads) it rises to 139.4 ms. At A4 the system enters a qualitatively different regime: requesting 200 RPS yields an actual 188.1 RPS at P99 = 349.7 ms, 300 RPS yields 185.2 RPS at 695.8 ms, and 400 RPS yields only 184.7 RPS at 1121.4 ms. The actual RPS plateaus at ≈185, while P99 continues to rise—a classic sign of *queueing saturation*. Each incoming request triggers lateral exchanges with peer nodes, and in the deployed code a peer's exchange handler forwards once more to its own peers (a bounded, TTL-limited propagation; Appendix E), so each request triggers six service-to-service calls ($\varphi = 6$, Table 2). Under high concurrency the Envoy sidecar proxies become the bottleneck, queueing requests and inflating the tail sharply—though, as Section 8.4 cautions, part of this growth near saturation may reflect application-side CPU contention as well.

9. Discussion

9.1. What the A×B Taxonomy Reveals

The two-axis taxonomy exposes performance dynamics that would be invisible in a one-dimensional (load-only) analysis. A conventional benchmarking study reporting aggregate P99 across all scenarios would yield a misleading average, masking the divergence that emerges at high load between low- and high-amplification workloads, and between light and heavy application weights.

Read this way, the A×B grid resolves into three regimes, which we state carefully given the weight and saturation confounds discussed above. In *Regime I* (A1–A2) latencies are low and stable and, at matched weight, the two coordination patterns behave almost identically. *Regime II* (A3) is where, at matched application weight, the peer-to-peer premium becomes measurable—the matched-weight comparison of Section 8.3 puts it at ≈1.2–1.8×—the operationally useful boundary between comfortable headroom and latency risk. *Regime III* (A4) is outright saturation, reached in our data only by heavy-compute peer-exchange workloads: achieved RPS falls below requested RPS and P99 grows by up to an order of magnitude relative to that scenario's own baseline. We emphasise that federated heavy workloads were not driven into Regime III, so we do not claim they are inherently immune to it; what the data show is that the heavy peer-exchange corner saturates first. The practical implication survives this caveat: auto-scaling for peer-exchange services should trigger at the Regime I/II boundary, well before Regime III.

9.2. Sensitivity to Application Weight

The suite already contains a coarse but informative sensitivity axis in the per-request application weight, which spans several decades of processing cost—from NEGLIGIBLE JSON acknowledgements (≈0 ms) through LIGHT and MEDIUM image work to HEAVY audio decoding (≈50 ms) (Appendix F). Reading the grid along this third axis is instructive: negligible-weight endpoints (UC2's `/result`) stay flat and never saturate; the heavy audio endpoints are the ones that enter Regime III first; and the matched-weight comparison (Section 8.3) holds weight fixed to isolate coordination. Application weight is thus not merely a nuisance variable held constant—it is a control that the design already varies discretely, and the central result is precisely that coordination structure and application weight interact: peer coordination is comparatively benign for light endpoints and becomes dominant for heavy ones under load. We now make this quantitative rather than descriptive.

A discrete sensitivity analysis. Each benchmark run is driven at exactly one entry endpoint, so every run inherits that endpoint's weight class, and the suite therefore supports a sensitivity analysis over four populated classes without any additional experiment.

Figure 13a plots the median P99 of each class at each scale tier. Two things are visible. First, application weight is a first-order determinant of the tail: at the A3 tier, which is the only tier populated by all four classes, the median P99 rises monotonically from 15.5 ms (negligible) through 47.4 ms (light) and 90.9 ms (medium) to 137.1 ms (heavy)—a factor of 8.8 attributable to per-request compute alone, at identical offered load and with the mesh configuration unchanged. Second, the classes differ in how they respond to load, which is the sensitivity the reviewer's question is really about: negligible-weight endpoints grow only 3.7× across the whole range measured (4.7 ms at A1 to 17.5 ms at A4) and never saturate, whereas heavy endpoints grow 8.3× (80.0 ms to 666.4 ms) and are the only class that reaches saturation at all.

The ordering is not a composition artefact. Because the weight classes are unevenly distributed across coordination categories—all negligible-weight runs are federated, all light-weight runs peer-to-peer—the aggregate could in principle reflect category rather than weight. Figure 13b rules this out by splitting the A3 tier by category: within the federated category the median P99 rises 15.5 → 88.1 → 134.8 ms from negligible to medium to heavy, and within the peer-to-peer category 47.4 → 105.4 → 139.4 ms from light to medium to heavy. The ordering holds inside each category separately. The same panel corroborates the matched-weight result of Section 8.3 from an independent slice of the data: at matched weight and tier the gap between categories is modest—1.20× at medium weight and 1.03× at heavy—which is the same modest premium the controlled comparison reports, and far from the order-of-magnitude separation suggested by the weight-mixed cell medians.

The practical reading is that a capacity plan which ignores per-request compute cost will be wrong by roughly an order of magnitude before coordination structure is considered at all, and that structure matters most once weight is already high. The analysis remains discrete, over four classes fixed by the deployed services; a continuous payload-size sweep at fixed topology would turn it into a full response surface, and is a natural next step (Section 11.2).

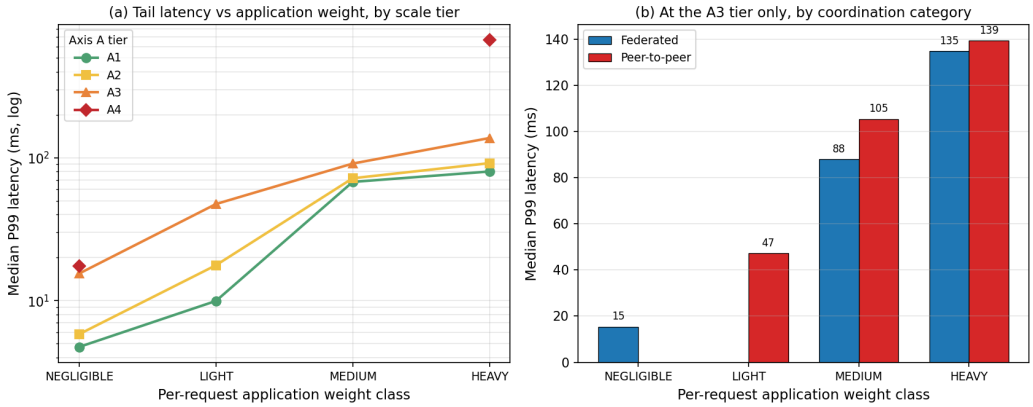


Figure 13. Discrete sensitivity of tail latency to per-request application weight, derived from the existing runs by classifying each by its entry endpoint. (**a**) Median P99 against weight class at each scale tier (log ordinate). A4 is sparsely populated—only the negligible and heavy classes were driven that far—so its two points are shown as markers rather than a line. (**b**) The A3 tier alone, split by coordination category, showing that the weight ordering holds within each category and is therefore not an artefact of how the categories are composed; at matched weight the two categories differ by only 1.03–1.20×.

When the mesh overhead matters. The overhead of the mesh has to be read against the computation that the services themselves perform. Under blocking dispatch the mesh contributes approximately $(1 + \varphi)L$ to a request whose own computation takes time T (Section 4.1), so its share of the request latency is

$$s = \frac{(1+\varphi)\,L}{T + (1+\varphi)\,L}. \quad (1)$$

With the per-hop latency measured on the negligible-compute probe endpoint (Table 10: $L \approx 2.7$ ms at low load, rising to ≈ 9.4 ms at the highest load at which the probe was measured), the share is substantial for the benchmarked services. For the heavy audio decode ($T \approx 50$ ms) it amounts to roughly 10% at $\varphi = 1$ and 27% at $\varphi = 6$ at low load, and to roughly 27% and 57% at the higher per-hop cost. For a service that computes for one minute, by contrast, even $\varphi = 6$ at the higher per-hop cost yields a share below 0.2%, and the mesh is irrelevant to its latency. The findings of this paper therefore apply to request–response services whose own computation lies between milliseconds and a few hundred milliseconds—the regime of the inference, aggregation and control services benchmarked here—and not to long-running batch computation. Equation (1) allows the reader to place a service of their own on this scale from its measured computation time and its φ. It should be noted that the equation describes the low-load baseline: near saturation the queueing component grows far faster than L itself (Table 10, right-hand columns), and that is where the tail inflation reported in Section 8 arises.

9.3. Design Implications for IoT Microservice Platforms

For IoT platforms deploying microservices under service-mesh governance, our measurements motivate several design hypotheses; we stress that these follow from the benchmarks but are not themselves validated here, and testing them is future work (Section 11.2). The most direct is topology-aware scaling: at a given application weight, peer-exchange services approach saturation earlier than federated ones, so auto-scaling controllers could fold coordination structure into their resource-allocation logic. Relatedly, mesh-level fan-out could be governed differently for the two categories—peer-exchange services may benefit from tighter per-destination rate limits and connection-pool caps to prevent queueing cascades. How such a limit should be set is itself a question: the paper's own saturation signature—achieved RPS falling below requested RPS—is a natural controller input, and adaptive-throttling methods that compute rate limits online for Web APIs [52] offer a concrete mechanism for turning that signal into a per-destination limit. Because peer-to-peer workloads consume a disproportionate share of the latency budget at high load, SLA specifications are better expressed as tiered budgets, with separate latency envelopes per coordination category rather than a single global target. Under extreme load it may even pay to apply an adaptive coordination fallback, temporarily switching from full peer-to-peer exchange to a hierarchical aggregation pattern that lowers the amplification factor, trading some inference quality for latency stability. Finally, where sidecar overhead is the dominant contributor to tail inflation, evaluating ambient mesh or eBPF-based alternatives [36,37] may yield substantial improvements, especially at the A3 and A4 tiers.

9.4. Practical Recommendations

These principles translate into four concrete recommendations for practitioners. First, profile the coordination topology before capacity planning: the number of downstream peer calls per request is the primary driver of tail-latency scaling, so a service with a large amplification factor (UC6 has $\varphi = 6$) places several times the sidecar load of a non-amplifying service on the mesh for each external request, and should be provisioned accordingly. Second, monitor the ratio of actual to requested RPS as a saturation indicator—once achieved RPS drops below about 95% of the target, the system has entered a degradation regime and should trigger auto-scaling or load shedding. Third, enforce SLAs on P99 rather than P50 or the average, since our data show P50 staying comparatively flat while P99 grows by up to an order of magnitude near saturation, so average-based SLAs systematically understate tail

risk. Fourth, benchmark with multi-phase ramps rather than single-point loads, because the value of the A×B framework lies precisely in the non-linear transitions between regimes that a single-RPS test would miss entirely.

9.5. Limitations

Several limitations should be considered when interpreting our results. The most fundamental, and we regard it as a severe one, is the single-node testbed. All experiments ran on one Kubernetes node. The absolute latencies and saturation thresholds reported here are therefore properties of that deployment and must not be read as multi-node performance: a distributed deployment would add inter-node network latency and scheduling variability on top of the effects we characterise, shifting both the absolute values and the rates at which saturation sets in. Having stated the severity, we should also state the reason for the choice, because it determines what the numbers are good for: co-locating all pods removes inter-node network variance, so the differences we measure across the A×B grid are attributable to the sidecar data plane and intra-host queueing rather than to the network. The figures are thus a lower bound on coordination cost—a multi-node deployment adds to them rather than removing them—and multi-node evaluation is the leading item of future work (Section 11.2).

The benchmarked services are also deliberately synthetic, using simplified inference logic such as random scoring and mean pixel intensity; production AI models with longer processing times would raise the latency baselines and could alter the relative weight of mesh overhead. A further constraint is that our resource view is latency-only—we report latency, throughput, and success rate but did not retain fine-grained per-pod CPU and memory time series, observing resource use only coarsely at the node level (where no sustained saturation appeared). We therefore localise the bottleneck indirectly, through the constant-payload argument, the negligible-compute mesh-floor probe, and the achieved-versus-requested RPS saturation signature of Section 8.4, rather than through direct sidecar counters; instrumenting Envoy and application containers with per-request CPU/memory accounting (via cgroup metrics or eBPF) is important future work. Finally, replication is limited (most cells hold 3–8 runs, enough for median-based analysis but not for confidence intervals or formal testing), scenarios were benchmarked in isolation rather than under the mixed-workload contention of a shared production cluster, and only the traditional sidecar-based Istio deployment was evaluated—ambient mesh and eBPF-based alternatives may scale quite differently, particularly for peer-to-peer workloads. **Replication, in both senses, and what it costs us.** Two distinct forms of replication bound these results and they have different consequences. The first is run replication, just described: 3–8 runs per cell support medians but not confidence intervals. The second is *service* replication—the number of pod replicas behind each service—and its consequence is more substantive than we previously made clear. Replica counts are held fixed throughout (2–3 for each primary or versioned service, 1 for terminal acknowledgement services; Section 6.2), which is what allows offered load and coordination structure to be the only factors varying across the grid. The price of that control is that every saturation point we report is a property of that particular horizontal capacity. We therefore cannot separate mesh-limited saturation—where the sidecar data plane is the binding constraint—from capacity-limited saturation, where the constraint is simply that there are too few replicas to absorb the offered rate. Both would present identically in our data, as achieved throughput falling below requested.

This interacts directly with Axis B, which sharpens rather than weakens the comparison. In UC4 and UC6 the peer calls are balanced across the same replica set that serves entry traffic, so that for every entry request those replicas and their sidecars receive three (UC4) and seven (UC6) calls rather than one (Section 5.1). The concentration is a genuine property of the coordination pattern, not an artefact of under-provisioning; but a replica-count sweep

at fixed topology is the experiment that would decide how much of the observed saturation is mesh-limited and how much is capacity-limited, and we name it as such rather than claiming the present data settle it.

Three further limitations bear directly on the reviewers' concerns. First, all downstream and peer calls are issued synchronously; asynchronous or parallel dispatch would lower the critical-path latency for the same call count, so our figures are a conservative, tail-worst-case bound rather than a characterisation of all dispatch styles. Second, the replica count is held fixed (Section 6.2), so the saturation points we report are specific to that horizontal capacity, and a replica-count sweep is needed to separate mesh-limited from capacity-limited saturation. Third—and most important for the federated-versus-peer contrast—the categories were not driven to comparable ρ: the federated heavy scenario reached only 200 RPS/50 threads and A4 was not run at all for UC1, UC3, and UC5, so our evidence that peer coordination degrades earlier rests on the matched-weight comparison (Section 8.3) rather than on the weight-mixed category aggregates, and a controlled study that drives both categories to $\rho \to 1$ at equal weight would settle the question definitively.

10. Threats to Validity

10.1. Internal Validity

Internal validity concerns whether the observed performance differences are causally attributable to the factors under study (coordination pattern and scale tier) rather than confounding variables.

Confounds addressed: We mitigated temporal confounds through sequential execution and warm-up phases. Namespace isolation prevents cross-scenario resource contention. Consistent Istio configuration across scenarios eliminates mesh-configuration bias.

Remaining concerns: The single-node testbed may introduce CPU and memory contention between the control plane (`istiod`), the Envoy sidecars, and the application containers. While namespace isolation limits logical interference, physical resource contention on a shared node cannot be fully eliminated. We coarsely monitored node-level CPU and memory utilisation during benchmarks and observed no sustained saturation, but, as noted in Section 9.5, fine-grained per-sidecar resource time series were not retained, so transient spikes cannot be ruled out.

10.2. External Validity

External validity concerns the generalisability of our findings to other settings.

Testbed representativeness: Our single-node Kubernetes cluster does not reproduce the network topologies, heterogeneous hardware, and multi-tenant resource contention of production IoT deployments. The absolute latency values reported here are specific to our testbed; the relative behaviour across the A×B grid is expected to generalise more robustly.

Continuum coverage: Our measurements cover one layer of the continuum—the cloud-side meshed cluster (Section 1.1)—and not the device or edge tiers, which enter only as offered load. Two consequences follow. First, an end-to-end latency budget for a real deployment must add device-to-edge and edge-to-cloud transport to the figures reported here, and those terms may dominate on constrained links; our results bound the cloud-side contribution, not the end-to-end total. Second, coordination that occurs between tiers—an edge gateway aggregating before forwarding, for instance—would change the effective call graph and hence the Axis B classification of the workload as a whole, even though the cloud-side segment we measure is unchanged. Extending the two-axis method across tiers, rather than within one, is the natural generalisation and is future work (Section 11.2).

Scenario coverage: Six scenarios, while diverse, cannot represent the full space of IoT coordination patterns. In particular, we do not evaluate broadcast/gossip patterns, hierar-

chical aggregation trees, or specifically dedicated hybrid patterns that combine federated and peer-to-peer elements.

Service mesh choice: Our evaluation uses Istio exclusively. Other service meshes (Linkerd, Consul Connect, Kuma) have different proxy architectures and may exhibit different performance profiles, particularly at extreme loads.

10.3. Construct Validity

Construct validity concerns whether our metrics and taxonomy accurately capture the theoretical constructs of interest.

Axis A thresholds: The RPS and thread-count thresholds defining our four tiers are calibrated to our testbed. Different hardware would require recalibration. We argue that the tier structure (not the absolute thresholds) is the transferable contribution.

Axis B classification: The binary classification into federated and peer-to-peer categories simplifies a continuous spectrum of coordination patterns, as formalised through the amplification factor φ in Section 4.1. Some scenarios (e.g., UC2's primary actor, which fans out to two secondary actors) exhibit hybrid characteristics. Following the mapping procedure of Section 4.4, we assigned each scenario to the category of its dominant amplification mode, acknowledging that a finer-grained, continuous treatment of φ (Section 11.2) might reveal additional structure.

10.4. Statistical Validity

Sample size: With 3–12 runs per A×B cell, our statistical power for detecting small effects is limited. We report medians rather than means to mitigate the influence of outliers, but formal hypothesis testing (e.g., Mann-Whitney U tests for cross-category comparisons) would require larger samples.

Non-independence: Runs within a scenario share the same service deployment and Kubernetes namespace. This introduces potential within-scenario correlation that violates the independence assumption of many statistical tests. Our median-based aggregation partially addresses this, but a mixed-effects model accounting for scenario-level clustering would be more rigorous.

11. Conclusions and Future Work

11.1. Key Findings

This paper has presented a benchmarking evaluation of service-mesh scalability in IoT-to-cloud microservice continuums spanning six scenarios and 51 multi-phase runs. Four findings stand out. First, tail latency is governed by the interaction of offered load, coordination structure, and per-request application weight: at matched application weight and load, peer-to-peer coordination adds a modest premium over federated rollout (≈1.2–1.8× at 100–200 RPS), not an order of magnitude; equivalently, at matched concurrency the federated scenario sustains exactly twice the throughput for a tail only 1.17–1.70× larger, so the coordination effect is a capacity cost rather than an artefact of how each scenario was driven. Second, the severe effects—up to ≈11× P99 growth from a scenario's own baseline, and achieved throughput collapsing to roughly half the requested rate—appear specifically in the heavy-compute peer-exchange corner of the grid driven toward saturation, which is the interaction the two-axis framework is built to surface. Third, the constant-payload design and negligible-compute acknowledgement endpoints let us attribute the load-dependent growth predominantly to mesh and queueing, with an acknowledged application-contention component at saturation. Fourth, all 51 runs returned a 0% error rate, so Istio preserved correctness even under saturation, at the cost of latency. In summary, the work contributes a two-axis benchmarking framework with an operational call-graph mapping procedure, a six-scenario suite grounded in real IoT

applications, and design hypotheses for capacity planning and auto-scaling that the framework itself can be used to test.

11.2. Future Work

Several directions remain for future investigation. The most pressing is to extend the evaluation to multi-node and geo-distributed clusters with realistic network topologies spanning edge and cloud tiers, and—in the same vein of broadening the infrastructure—to compare Istio against Linkerd, Cilium's eBPF data plane, and sidecar-less ambient mesh so that the architectural trade-offs can be quantified across the A×B grid. On the workload side, the synthetic endpoints should be replaced with production-grade ML models (YOLO for drone inspection, transformer-based anomaly detectors) to measure mesh overhead relative to genuinely compute-intensive processing, and multiple workloads should be benchmarked concurrently to capture the cross-workload interference and resource contention that a shared cluster imposes. Finally, two natural extensions of the framework itself are an adaptive mesh-configuration controller that auto-tunes Istio timeout, retry, and circuit-breaking parameters from real-time coordination-pattern detection, and a finer-grained Axis B that moves beyond the binary federated/peer-to-peer split to include hierarchical, gossip, and hybrid coordination patterns along the continuous φ scale. Beyond these, the reviews point to several concrete extensions we intend to pursue: a replica-count sweep to disentangle mesh-limited from capacity-limited saturation; a continuous payload-size (application-weight) sweep at fixed topology to convert the discrete weight classes of Section 9.2 into a response surface; an asynchronous/parallel-dispatch variant to quantify how much of the peer-exchange penalty is recovered by overlapping sub-calls; a campaign matched simultaneously on offered rate and concurrency (no pair in the present data is matched on both), driving the federated and peer-to-peer categories to comparable saturation at matched weight; and, not least, validating the design hypotheses of Section 9.3—in particular using the achieved-versus-requested-RPS signal to drive an adaptive per-destination rate limiter along the lines of [52]—rather than only deriving them from the benchmarks.

Author Contributions: Conceptualization, M.A.K.; methodology, M.A.K.; software, M.A.K. and A.E.; validation, M.A.K. and A.E.; formal analysis, M.A.K.; investigation, M.A.K. and A.E.; data curation, M.A.K.; writing—original draft preparation, M.A.K.; writing—review and editing, M.A.K., A.E., G.X., E.M. and R.V.; visualization, M.A.K.; supervision, G.X., E.M. and R.V. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by the European Union's Horizon Europe research and innovation programme under Grant Agreement No. 101135930 (COGNETS) and Grant Agreement No. 101225759 (PQ-NEXT), and by Project SHINE (PID2024-159781OB-I00) funded by MCIU/AEI/10.13039/501100011033 and by ERDF, EU.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The complete artefacts needed to reproduce every figure and table in this paper are openly available from Zenodo at https://doi.org/10.5281/zenodo.22307784 [53], released under CC BY 4.0. The archive contains the raw Fortio JSON result files for all benchmark phases, the derived spreadsheets, and the analysis scripts `build-dataset.py`, `fortio-plot.py` and `fortio-to-excel.py`. Running `build-dataset.py` with the `-verify` flag re-derives the Axis A/B labels from the raw JSON and asserts that all eight per-cell median P99 values reported in Section 8 are reproduced, exiting with an error on any mismatch. No third-party or access-restricted data were used.

Acknowledgments: 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 or the European Research Executive Agency (REA); neither the European Union nor REA can be held responsible

for them. During the preparation of this manuscript, the authors used generative AI-based tools for the purposes of language editing and structural revision. The authors have reviewed and edited the output and take full responsibility for the content of this publication.

Conflicts of Interest: The authors declare no conflicts of interest.

Appendix A. Full Workload Table

Table A1 presents the complete set of 51 benchmark runs with their Fortio parameters and key results.

Table A1. Complete benchmark run summary (51 runs).

UC	Tier	RPS_{req}	RPS_{act}	Thr.	P50 (ms)	P90 (ms)	P99 (ms)
UC1	A1	10	9.9	10	62.7	76.4	80.0
UC1	A1	10	9.9	10	55.0	74.7	80.0
UC1	A1	10	10.0	5	52.7	68.1	78.1
UC1	A2	50	49.9	15	64.1	82.6	91.5
UC1	A3	100	99.9	25	76.5	97.7	117.2
UC1	A3	150	149.8	35	88.9	115.9	134.8
UC1	A3	200	199.0	50	115.9	159.3	237.8
UC2	A1	10	10.0	5	10.7	14.2	55.0
UC2	A1	10	10.0	5	2.6	3.2	4.0
UC2	A1	10	10.0	5	2.7	3.7	4.7
UC2	A2	50	50.0	10	14.2	17.9	24.2
UC2	A2	50	50.0	10	3.1	4.0	5.8
UC2	A2	50	50.0	10	2.9	3.8	4.7
UC2	A3	100	100.0	20	22.9	29.0	33.9
UC2	A3	100	100.0	20	4.0	5.2	8.0
UC2	A3	100	100.0	20	4.8	9.4	12.5
UC2	A3	150	149.9	30	32.0	39.4	44.5
UC2	A3	150	150.0	30	5.1	9.4	12.7
UC2	A3	150	150.0	30	5.1	8.7	11.5
UC2	A3	200	199.7	50	50.1	62.7	70.5
UC2	A3	200	199.9	50	7.7	12.7	15.6
UC2	A3	200	199.9	50	7.6	12.5	15.5
UC2	A4	300	299.6	60	59.1	76.4	107.2
UC2	A4	300	299.9	60	8.9	14.1	17.5
UC2	A4	300	299.9	60	9.4	14.6	17.3
UC3	A1	5	5.0	2	54.6	61.9	70.6
UC3	A3	1	1.0	30	34.0	78.5	81.6
UC3	A2	50	50.0	10	61.0	68.3	69.9
UC3	A3	100	100.0	25	69.2	78.7	88.1
UC3	A3	200	198.7	50	74.5	95.3	121.8
UC4	A3	50	49.9	25	74.6	94.5	100.0
UC4	A3	100	99.6	50	103.2	131.4	139.7
UC4	A4	150	149.1	75	156.4	195.0	243.4
UC4	A4	200	199.0	100	233.8	295.0	345.1
UC4	A4	300	296.9	150	409.6	496.9	666.4
UC4	A4	400	315.5	200	625.2	751.0	887.1
UC5	A1	5	5.0	2	53.6	59.8	67.1
UC5	A1	5	5.0	2	6.8	8.7	9.9
UC5	A1	5	5.0	2	53.6	61.4	67.6
UC5	A2	50	50.0	10	10.7	13.9	17.7
UC5	A2	50	50.0	10	56.8	67.2	73.5
UC5	A3	100	100.0	25	19.9	27.3	31.4
UC5	A3	100	100.0	25	68.0	84.3	90.9
UC5	A3	200	199.4	50	33.0	46.1	63.3
UC5	A3	200	198.8	50	70.5	99.1	120.0

Table A1. *Cont.*

UC	Tier	RPS_{req}	RPS_{act}	Thr.	P50 (ms)	P90 (ms)	P99 (ms)
UC6	A2	25	24.9	5	61.0	74.3	79.9
UC6	A2	50	49.9	10	68.2	86.6	98.2
UC6	A3	100	99.4	25	87.6	121.5	139.4
UC6	A4	200	188.1	50	266.1	306.9	349.7
UC6	A4	300	185.2	100	542.9	607.7	695.8
UC6	A4	400	184.7	150	810.6	895.3	1121.4

Appendix B. Additional Plots Description

Several supplementary visualisations are included in the public dataset accompanying this paper (see the Data Availability Statement). A run-count heatmap (Figure 7) shows the distribution of benchmark runs across the A×B grid, which is useful for assessing coverage balance. Connection-latency distributions report the P50/P90/P99 of TCP connection-establishment times and confirm that Envoy connection pooling keeps connection overhead below 0.2 ms across all runs. Per-scenario latency profiles plot the individual P50/P90/P99 curves for each of the six scenarios for within-scenario trend analysis, and a tail-to-median ratio (P99/P50) heatmap visualises distributional spread across the grid, complementing the absolute P99 heatmap.

Appendix C. Run Validity Rule

The archived dataset contains 52 benchmark phases, of which 51 enter the analysis. One UC3 phase is excluded: it requested 5 RPS but achieved 0.6 RPS (12% of the requested rate) and completed only 18 requests. The exclusion follows a single rule applied uniformly: a phase that fails to reach its requested rate at low offered load (at or below 50 RPS) cannot be exhibiting saturation—the phenomenon this study measures—and instead indicates that the service was still warming up. Rate shortfalls at high load are genuine saturation and are retained; indeed they are the A4 trigger in Table 3.

The excluded phase is the first UC3 run executed and is separated from the rest of the suite by a wide margin on both relevant quantities: it is the only phase in the suite completing fewer than 60 requests, and the next-lowest achieved fraction across all other phases is 46%, recorded at 400 RPS where the shortfall is precisely the saturation being measured. The run is retained in the public archive rather than deleted, is flagged in the workbook's `runs_all` sheet, and is listed with its diagnostic values on a dedicated `excluded_runs` sheet, so that the effect of reinstating it can be checked directly.

Appendix D. Data Extraction Script Summary

The data extraction pipeline consists of three Python scripts. `fortio-to-excel.py` parses one or more Fortio JSON result files (single files, directories, or glob patterns) and exports a flat Excel workbook capturing requested and actual RPS, connections, latency percentiles (P50, P75, P90, P99, P99.9), average/min/max/stddev latency, HTTP response codes, error rates, connection statistics, and the derived tail-to-median ratio and latency×throughput product. `build-dataset.py` performs the analysis proper: it walks the per-scenario JSON directories, applies the validity rule of Appendix C, assigns each run its Axis A tier and Axis B category by the rules of Table 3, and writes an annotated workbook whose sheets separate all parsed runs, the analysed subset, the per-cell summary, and any excluded runs. Finally, `fortio-plot.py` reads that annotated workbook and generates the publication-ready visualisations used in the paper—the run-count heatmap, the P99 heatmap, the P90/P99 line plot, the P99 box plots by category, and the RPS-versus-P99 scatter plot—all rendered in Matplotlib (version 3.10.9) with consistent styling. All three

scripts are provided in the public dataset accompanying this paper (see the Data Availability Statement). Reproducibility is enforced rather than asserted: `build-dataset.py --verify` recomputes the eight per-cell median P99 values from the raw JSON and fails if any of them departs from the value printed in this paper.

Appendix E. Endpoint Pseudocode

The pseudocode below is grouped by use case using the same UC numbering, scenario names, and topologies as Section 5.1 and Figure 4; the service and endpoint names used here (e.g., Rating, Classification, Result-store, Object Processing services; `/predict`, `/forecast`, `/result`, `/analyze`) are the concrete instantiations of the abstract roles in those descriptions. To aid cross-reference: the entry endpoint of each procedure is the one driven by Fortio, downstream POST calls correspond to the mesh hops counted in Appendix G, and endpoints that only deserialise JSON and return an acknowledgement are the negligible-compute probes analysed in Section 8.4.

UC1—Weighted-Routing Inference

Algorithm A1 Audio Inference Service (v1/v2)

procedure PREDICT(*audio_bytes*)
  samples, *sr* ← **AudioDecode**(*audio_bytes*) ▷ librosa FFT decode, $O(n \log n)$
  duration ← |*samples*|/*sr*
  amplitude ← **Mean**(**Abs**(*samples*)) ▷ NumPy linear scan, $O(n)$
  score ← ComputeScore()
  if version = v1 **then** ▷ v1 calls downstream; v2 returns directly
    rating ← **POST**(RatingService, `/rate`, {*score*}) ▷ JSON, $O(1)$
  end if
  return {*score*, *rating*, *duration*, *amplitude*}
end procedure

Algorithm A2 Rating Service

procedure RATE(*json*) ▷ JSON payload, $O(1)$
  score ← *json*.score
  rating ← ThresholdCompare(*score*) ▷ scalar comparisons
  return {*score*, *rating*}
end procedure

UC2—Multi-Actor Fan-Out

Algorithm A3 Actor (A, B, C—identical)

procedure FORECAST
  metric ← **Random**() ▷ $O(1)$, no payload processing
  for *node* ∈ *Downstream_Nodes* **do** ▷ |*Peers*| = 2
    POST(*node*, `/result`, {*actor*, *metric*}) ▷ JSON, $O(1)$
  end for
  return {*actor*, *metric*}
end procedure
procedure RESULT(*json*) ▷ JSON payload, $O(1)$
  return Ack
end procedure

UC3—Sequential Service Chain

Algorithm A4 Detection Service (v1)

procedure DETECT(*image_bytes*)
  img ← **ImageDecode**(*image_bytes*, COLOR) ▷ cv2 JPEG decode, $O(w \cdot h)$
  intensity ← **Mean**(*img*) ▷ pixel-level mean, $O(w \cdot h)$
  result ← {*model*: *v*1, *intensity*}
  POST(ResultStore, `/register`, *result*) ▷ JSON, $O(1)$
  return *result*
end procedure

Algorithm A5 Detection Service (v2)

procedure DETECT(*image_bytes*)
  img ← **ImageDecode**(*image_bytes*, GRAYSCALE) ▷ cv2 JPEG decode, $O(w \cdot h)$
  edges ← **CannyEdge**(*img*) ▷ Sobel + NMS, $O(w \cdot h)$
  density ← **Sum**(*edges*) / **Size**(*edges*) ▷ $O(w \cdot h)$
  result ← {*model*: *v*2, *density*}
  POST(ResultStore, `/register`, *result*) ▷ JSON, $O(1)$
  return *result*
end procedure

Algorithm A6 Result Store (endpoint `/register`)

procedure REGISTER(*json*) ▷ JSON payload, $O(1)$
  return Ack
end procedure

UC4—Split-Route Peer Exchange

Algorithm A7 Monitoring Service (v1/v2)

procedure ANALYZE(*audio_bytes*)
  samples, *sr* ← **AudioDecode**(*audio_bytes*) ▷ librosa FFT decode, $O(n \log n)$
  duration ← |*samples*|/*sr*
  amplitude ← **Mean**(**Abs**(*samples*)) ▷ $O(n)$
  if version = v1 **then**
    damage ← (*duration* > 1.0) ∧ (*amplitude* > 0.05)
  else
    score ← **Random**()
  end if
  for *peer* ∈ *AnalysisPeers* **do** ▷ lateral peer exchange, |*Peers*| = 1
    POST(*peer*, `/exchange`, {*amplitude*, *ttl*: 1})
  end for
  severity ← **POST**(ClassificationService, `/classify`, {*duration*, *amplitude*, ...})
  return {*duration*, *amplitude*, *severity*}
end procedure
procedure EXCHANGE(*json*) ▷ peer-exchange handler; **TTL-bounded peer forwarding**
  Fuse(*json.amplitude*) ▷ merge peer's intermediate result, $O(1)$
  if *json.ttl* > 0 **then**
    for *peer* ∈ *AnalysisPeers* **do**
      POST(*peer*, `/exchange`, {*json.amplitude*, *ttl*: *json.ttl* − 1}) ▷ one further lateral hop
    end for
  end if
  return Ack
end procedure

Algorithm A8 Classification Service

procedure CLASSIFY(*json*) ▷ JSON payload, $O(1)$
  severity ← ThresholdCompare(*json*) ▷ scalar comparisons
  return {*severity*}
end procedure

UC5—Dual-Path Peer Pipeline

Algorithm A9 Image Inspection Service

procedure CHECK(*image_bytes*)
  img ← **ImageDecode**(*image_bytes*, GRAYSCALE) ▷ cv2 JPEG decode, $O(w \cdot h)$
  score ← **Mean**(*img*) ▷ pixel-level mean, $O(w \cdot h)$
  if *score* < *Threshold* **then**
    POST(ObjectProcessingService, `/process`, {*score*}) ▷ JSON, $O(1)$
  else
    POST(NextStage, `/process`, {*score*}) ▷ JSON, $O(1)$
  end if
  return {*score*, *dest*}
end procedure

Algorithm A10 Object Processing Service

procedure PROCESS(*json*) ▷ JSON payload, $O(1)$
  piece ← *json*
  piece.processed ← true
  POST(ImageInspector, `/check`, *piece*) ▷ JSON feedback loop
  return *piece*
end procedure

Algorithm A11 Scheduling Service (v1/v2)

procedure SCHEDULE(*json*)
  readings ← *json*.sensor_readings ▷ JSON array of length k
  if version = v1 **then**
    $feature$ ← **Mean**(*readings*) ▷ $O(k)$
  else
    $feature$ ← **StdDev**(*readings*) ▷ two-pass, $O(k)$
  end if
  POST(ResultStore, `/register`, {$feature$}) ▷ JSON, $O(1)$
  return {$feature$}
end procedure

Algorithm A12 Result Store (UC5; endpoint `/register`)

procedure REGISTER(*json*) ▷ JSON payload, $O(1)$
  return Ack
end procedure

UC6—Full-Mesh Peer Amplification

Algorithm A13 Diagnostic Node (node1/node2/node3 — identical)

procedure ANALYZE(*audio_bytes*)
  samples, *sr* ← **AudioDecode**(*audio_bytes*) ▷ librosa FFT decode, $O(n \log n)$
  feature ← **Mean**(**Abs**(*samples*)) ▷ $O(n)$
  for *peer* ∈ *PeerNodes* **do** ▷ |*PeerNodes*| = 2
    POST(*peer*, `/result`, {*node*, *feature*, *ttl*: 1}) ▷ JSON, $O(1)$
  end for
  return {*node*, *feature*}
end procedure
procedure RESULT(*json*) ▷ peer handler; **bounded lateral propagation**
  Fuse(*json.feature*) ▷ merge into local consensus state, $O(1)$
  if *json.ttl* > 0 **then**
    for *peer* ∈ *PeerNodes* **do**
      POST(*peer*, `/result`, {*node*, *json.feature*, *ttl*: *json.ttl* − 1}) ▷ one further peer hop
    end for
  end if
  return Ack
end procedure

The `ttl` field bounds the lateral propagation so the propagation terminates: with `ttl` = 1 each request triggers two peer calls that each trigger two further calls, giving a per-request amplification of $\varphi \approx 6$ (matched by the range stated in Section 4.1). These peer calls, which a federated fan-out does not make, are what place UC4 and UC6 in the peer-to-peer category, and the invocations they add are counted in φ (Table 2).

Appendix F. Per-Endpoint Processing Complexity

This section characterises the processing overhead of each microservice endpoint independently of the network layer. The goal is to provide a theoretical indicator of how much of the measured end-to-end latency (captured via Istio/Fortio) can be attributed to local computation versus network and service-mesh overhead.

For every endpoint we record the **operations** performed on the request payload before an HTTP response (or downstream request) is produced, the **asymptotic processing complexity**—where n is the number of audio samples ($n = sr \times dur$), $w \times h$ the pixel count, k the length of a JSON array, and P the number of downstream peers—and a qualitative **weight class** summarising the expected impact on measured latency. The weight classes are defined as follows:

NEGLIGIBLE Only JSON deserialisation, scalar comparisons, and/or random-number generation; processing time is constant and sub-millisecond.

LIGHT A single-pass linear scan over a small JSON array (`np.mean`, `np.std`); processing time is linear in a typically small input.

MEDIUM Pixel-level image decoding and a single-pass statistical reduction (`cv2.imdecode` + `np.mean`); processing time scales with image dimensions.

MEDIUM+ Image decoding, gradient computation (Sobel filters), and non-maximum suppression (`cv2.imdecode` + `cv2.Canny`); higher constant factor than plain MEDIUM at the same asymptotic class.

HEAVY Full audio decoding via FFT-based resampling (`librosa.load`) plus a linear reduction; processing time scales super-linearly with the number of audio samples.

The test payloads were held constant across all runs: a 1025 KB, 5-s WAV audio file ($n \approx$ 220,500 samples at 44.1 kHz) and a 21 KB, 612 × 408 JPEG image ($w \cdot h$ = 249,696 pixels).

Appendix F.1. UC1—Weighted-Routing Inference

Fortio entry: `POST /predict` with multipart audio.
Istio traffic split: v1 = 50%, v2 = 50%.
Pipeline: `/predict` (v1) $\xrightarrow{\text{POST}}$ `/rate`; `/predict` (v2) → response (no downstream call).

Hop	Endpoint	Operations	Complexity	Weight
Entry	`/predict` (v1)	`librosa.load` (FFT decode); `librosa.get_duration`; `np.abs+np.mean`; POST `/rate`	$O(n \log n)$	Heavy
Entry	`/predict` (v2)	`librosa.load` (FFT decode); `librosa.get_duration`; `np.abs+np.mean`; `random.uniform`	$O(n \log n)$	Heavy
Downstream	`/rate`	JSON deserialise; two scalar comparisons	$O(1)$	Negligible

Appendix F.2. UC2 — Multi-Actor Fan-Out

Fortio entry: two separate runs per phase — `POST /forecast` (JSON), `POST /result` (JSON).
Istio traffic split: 3 actors × 40%, 30%, 30% splits (A, B, C — identical code structure).
Pipeline: `/forecast` $\xrightarrow{P=2}$ `/result` on peers; `/result` → response.

Hop	Endpoint	Operations	Complexity	Weight
Entry	`/forecast`	`random.uniform`; iterate $P = 2$ peers, POST `/result` on each	$O(1)$	Negligible
Entry/Down	`/result`	JSON deserialise; construct ack	$O(1)$	Negligible

Appendix F.3. UC3—Sequential Service Chain

Fortio entry: `POST /detect` with multipart image.
Istio traffic split: v1 = 70%, v2 = 30%.
Pipeline: `/detect` (v1 or v2) $\xrightarrow{\text{POST}}$ `/register`.

Hop	Endpoint	Operations	Complexity	Weight
Entry	`/detect` (v1)	`np.frombuffer` ($O(1)$, zero-copy); `cv2.imdecode` (IMREAD_COLOR, $O(w \cdot h)$); `np.mean` over all pixels ($O(w \cdot h)$); POST `/register`	$O(w \cdot h)$	Medium
Entry	`/detect` (v2)	`np.frombuffer` ($O(1)$); `cv2.imdecode` (GRAYSCALE, $O(w \cdot h)$); `cv2.Canny` (Sobel + NMS, $O(w \cdot h)$); `np.sum/size` ($O(w \cdot h)$); POST `/register`	$O(w \cdot h)$	Medium+
Downstream	`/register`	JSON deserialise; echo payload	$O(1)$	Negligible

Appendix F.4. UC4—Split-Route Peer Exchange

Fortio entry: `POST /analyze` with multipart audio.
Istio traffic split: v1 = 50%, v2 = 50%.
Pipeline: `/analyze` (v1 or v2) $\xrightarrow{P=1,\ \texttt{ttl=1}}$ `/exchange` on the peer analysis instance (which fuses the value and forwards once more, bounded by `ttl`), and $\xrightarrow{\text{POST}}$ `/classify`.

Hop	Endpoint	Operations	Complexity	Weight
Entry	`/analyze` (v1)	`librosa.load` (FFT decode); `librosa.get_duration`; `np.abs+np.mean`; scalar comparison; POST `/exchange` (peer, `ttl` = 1); POST `/classify`	$O(n \log n)$	Heavy
Entry	`/analyze` (v2)	`librosa.load` (FFT decode); `librosa.get_duration`; `np.abs+np.mean`; `random.uniform`; POST `/exchange` (peer, `ttl` = 1); POST `/classify`	$O(n \log n)$	Heavy
Lateral	`/exchange`	JSON parse; fuse peer amplitude; if `ttl` > 0 forward to peer (`ttl` $- 1$); ack	$O(1)$	Negligible
Downstream	`/classify`	JSON deserialise; three scalar comparisons	$O(1)$	Negligible

Appendix F.5. UC5—Dual-Path Peer Pipeline

Fortio entry: two separate runs per phase—`POST /schedule` with JSON `{sensor_readings:[...]}`, `POST /check` with multipart image.
Istio traffic split (scheduler): v1 = 50%, v2 = 50%.
Pipelines:
`/schedule` (v1 or v2) $\xrightarrow{P=1}$ `/register`;
`/check` → `/process` on downstream peers, where `/process` (ObjectProcessing) $\xrightarrow{P=1}$ `/check` (feedback loop).

Hop	Endpoint	Operations	Complexity	Weight
Entry	`/schedule` (v1)	JSON parse; `np.mean` over sensor array ($k = 4$); POST `/register`	$O(k)$	Light
Entry	`/schedule` (v2)	JSON parse; `np.array` + `np.std` (two-pass, $k = 4$); POST `/register`	$O(k)$	Light
Entry	`/check`	`np.frombuffer` ($O(1)$); `cv2.imdecode` (GRAYSCALE, $O(w \cdot h)$); `np.mean` ($O(w \cdot h)$); POST to `/process` on downstream peers	$O(w \cdot h)$	Medium
Downstream	`/process`	JSON parse; set flag; POST `/check` ($P = 1$)	$O(1)$	Negligible
Downstream	`/register`	JSON parse; echo payload	$O(1)$	Negligible

Appendix F.6. UC6—Full-Mesh Peer Amplification

Fortio entry: `POST /analyze` with multipart audio.
Istio traffic split: node1 = 40%, node2 = 40%, node3 = 20%.
Pipeline (identical for all 3 nodes): `/analyze` $\xrightarrow{P=2,\ \texttt{ttl}=1}$ `/result` on peer nodes, each of which fuses the feature and forwards once more to its own $P = 2$ peers (TTL-bounded peer forwarding, $\varphi = 6$).

Hop	Endpoint	Operations	Complexity	Weight
Entry	`/analyze`	`librosa.load` (FFT decode); `np.abs+np.mean`; iterate $P = 2$ peers, POST `/result` (`ttl` = 1)	$O(n \log n)$	Heavy
Downstream/peer	`/result`	JSON parse; fuse feature; if `ttl` > 0 forward to $P = 2$ peers (`ttl` $- 1$); construct ack	$O(1)$	Negligible

Appendix F.7. Summary—Processing Complexity by Weight Class

Table A2. Processing weight classification of all endpoints exercised during Fortio/Istio benchmarking. Endpoints classified as Negligible contribute near-zero processing latency; any measured delay on these endpoints can be attributed almost entirely to network and service-mesh overhead.

Weight Class	Dominant Operation	Complexity	Affected Endpoints
Heavy	`librosa.load` (FFT decode)	$O(n \log n)$	UC1: `/predict` (v1, v2) UC4: `/analyze` (v1, v2) UC6: `/analyze` (all nodes)
Medium+	`cv2.imdecode` + `cv2.Canny`	$O(w \cdot h)$	UC3: `/detect` (v2)
Medium	`cv2.imdecode` + `np.mean`	$O(w \cdot h)$	UC3: `/detect` (v1) UC5: `/check`
Light	`np.mean`/`np.std` (JSON array)	$O(k)$	UC5: `/schedule` (v1, v2)
Negligible	JSON parse/scalar ops	$O(1)$	UC1: `/rate` UC2: `/forecast`, `/result` UC3: `/register` UC4: `/classify` UC5: `/process`, `/register` UC6: `/result`

Appendix F.8. Implications for Fortio/Istio Benchmarking

Two consequences for Fortio/Istio benchmarking follow from this classification. The first is the availability of pure network baselines: all UC2 endpoints and the lightweight downstream services (`/rate`, `/classify`, `/register`, `/process`, `/result`) carry Negligible processing cost, so latency measured on them reflects almost exclusively the network path and Istio sidecar overhead. The second is that the media payloads are controlled: for endpoints classified as Heavy, Medium+, or Medium, the processing cost is deterministic for a fixed input—audio endpoints (UC1, UC4, UC6) are governed by the sample count $n = sr \times duration \approx 220{,}500$ (a 5 s WAV at 44.1 kHz, 1025 KB), and image endpoints (UC3, UC5) by the pixel count $w \times h = 612 \times 408 = 249{,}696$ (a 21 KB JPEG). Under fixed inputs the processing component becomes a constant additive term, so differences in latency across mesh configurations can still be attributed to the network layer.

Appendix G. Per-Request Latency Decomposition

Let L denote the per-hop mesh latency (Envoy sidecar overhead), T_{audio} the audio-decode processing time, and T_{image} the image-decode processing time. Table A3 gives the resulting end-to-end latency of each benchmarked entry endpoint.

Table A3. End-to-end latency decomposition per Fortio request. L = mesh-hop latency (Envoy sidecar); T_{audio} = audio decode (`librosa.load`); T_{image} = image decode (`cv2.imdecode` + `np.mean`); T^{+}_{image} = image decode with edge detection (`cv2.Canny`). Negligible processing (<1 ms) is absorbed into L.

UC	Entry Point	End-to-End Latency (per Request)	Hops
UC1	`/predict` (v1)	$T_{\text{audio}} + 2L$	2
UC1	`/predict` (v2)	$T_{\text{audio}} + L$	1
UC2	`/forecast`	$3L$	3
UC2	`/result`	L	1
UC3	`/detect` (v1)	$T_{\text{image}} + 2L$	2
UC3	`/detect` (v2)	$T^{+}_{\text{image}} + 2L$	2
UC4	`/analyze`	$T_{\text{audio}} + 4L$	4

Table A3. *Cont.*

UC	Entry Point	End-to-End Latency (per Request)	Hops
UC5	`/schedule`	$2L$	2
UC5	`/check` (pass)	$T_{\text{image}} + L$	1
UC5	`/check` (feedback loop)	$T_{\text{image}} + 4L + T_{\text{image}}$	4
UC6	`/analyze`	$T_{\text{audio}} + 7L$	7

The hop counts assume the blocking, sequential dispatch of the deployed handlers (Section 4.1): every service-to-service call, including the TTL-bounded peer calls of UC4 and UC6, is answered before the handler proceeds, so each request traverses $1 + \varphi$ mesh hops (Table 2). The measurements are consistent with this: at low load the median P50 of UC6 `/analyze` is 64.6 ms, well above the approximately 53 ms that one audio decode and a single hop would give if its six peer calls lay off the critical path. The additional calls cost little at low load, where each hop is cheap, but under load they compound queueing at the shared sidecars and inflate the tail, which is the mechanism behind the interaction reported in Section 8.3. The `/check` (feedback loop) row describes a branch that the benchmark input does not trigger (Section 5.1).

References

1. Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A Survey. *Comput. Netw.* **2010**, *54*, 2787–2805. [CrossRef]
2. Al-Fuqaha, A.; Guizani, M.; Mohammadi, M.; Aledhari, M.; Ayyash, M. Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. *IEEE Commun. Surv. Tutor.* **2015**, *17*, 2347–2376. [CrossRef]
3. Bonabeau, E.; Dorigo, M.; Theraulaz, G. *Swarm Intelligence: From Natural to Artificial Systems*; Oxford University Press: Oxford, UK, 1999.
4. Brambilla, M.; Ferrante, E.; Birattari, M.; Dorigo, M. Swarm Robotics: A Review from the Swarm Engineering Perspective. *Swarm Intell.* **2013**, *7*, 1–41. [CrossRef]
5. Moreschini, S.; Pecorelli, F.; Li, X.; Naz, S.; Hästbacka, D.; Taibi, D. Cloud Continuum: The Definition. *IEEE Access* **2022**, *10*, 131876–131886. [CrossRef]
6. Ullah, A.; Kiss, T.; Kovács, J.; Tusa, F.; Deslauriers, J.; Dagdeviren, H.; Arjun, R.; Hamzeh, H. Orchestration in the Cloud-to-Things Compute Continuum: Taxonomy, Survey and Future Directions. *J. Cloud Comput. Adv. Syst. Appl.* **2023**, *12*, 135. [CrossRef]
7. Dustdar, S.; Casamayor Pujol, V.; Donta, P.K. On Distributed Computing Continuum Systems. *IEEE Trans. Knowl. Data Eng.* **2023**, *35*, 4092–4105. [CrossRef]
8. Dragoni, N.; Giallorenzo, S.; Lafuente, A.L.; Mazzara, M.; Montesi, F.; Mustafin, R.; Safina, L. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*; Springer: Cham, Switzerland, 2017; pp. 195–216. [CrossRef]
9. Envoy Proxy. Envoy Proxy Documentation. 2024. Available online: https://www.envoyproxy.io/docs (accessed on 20 May 2025).
10. Li, W.; Lemieux, Y.; Gao, J.; Zhao, Z.; Han, Y. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *Proceedings of the 2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*; IEEE: Piscataway, NJ, USA, 2019; pp. 122–130. [CrossRef]
11. Istio. Istio Service Mesh Documentation. 2024. Available online: https://istio.io/latest/docs/ (accessed on 20 May 2025).
12. Elkhatib, Y.; Poyato, J. An Evaluation of Service Mesh Frameworks for Edge Systems. In *Proceedings of the 6th International Workshop on Edge Systems, Analytics and Networking (EdgeSys)*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 19–24. [CrossRef]
13. Sedghpour, M.R.S.; Townend, P. Service Mesh and eBPF-Powered Microservices: A Survey and Future Directions. In *Proceedings of the IEEE International Conference on Service-Oriented System Engineering (SOSE)*; IEEE: Piscataway, NJ, USA, 2022; pp. 176–184. [CrossRef]
14. Saleh Sedghpour, M.R.; Klein, C.; Tordsson, J. An Empirical Study of Service Mesh Traffic Management Policies for Microservices. In *Proceedings of the 2022 ACM/SPEC International Conference on Performance Engineering (ICPE)*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 17–27. [CrossRef]
15. Henning, S.; Hasselbring, W. A Configurable Method for Benchmarking Scalability of Cloud-Native Applications. *Empir. Softw. Eng.* **2022**, *27*, 143. [CrossRef]

16. Gan, Y.; Zhang, Y.; Cheng, D.; Shetty, A.; Rathi, P.; Katarki, N.; Bruno, A.; Hu, J.; Ritchken, B.; Jackson, B.; et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*; Association for Computing Machinery: New York, NY, USA, 2019; pp. 3–18. [CrossRef]
17. Sriraman, A.; Wenisch, T.F. µSuite: A Benchmark Suite for Microservices. In *Proceedings of the 2018 IEEE International Symposium on Workload Characterization (IISWC)*; IEEE: Piscataway, NJ, USA, 2018; pp. 1–12. [CrossRef]
18. Pelle, I.; Czentye, J.; Doka, J.; Sonkoly, B. Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS. In *Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*; IEEE: Piscataway, NJ, USA, 2019; pp. 272–280. [CrossRef]
19. Čilić, I.; Krivić, P.; Podnar Žarko, I.; Kušek, M. Performance Evaluation of Container Orchestration Tools in Edge Computing Environments. *Sensors* **2023**, *23*, 4008. [CrossRef] [PubMed]
20. Fortio. Fortio: Fortio Load Testing Library, Command Line Tool, Advanced Echo Server and Web UI. 2024. Available online: https://fortio.org/ (accessed on 15 May 2025).
21. Shi, W.; Cao, J.; Zhang, Q.; Li, Y.; Xu, L. Edge Computing: Vision and Challenges. *IEEE Internet Things J.* **2016**, *3*, 637–646. [CrossRef]
22. Khan, W.Z.; Ahmed, E.; Hakak, S.; Yaqoob, I.; Ahmed, A. Edge Computing: A Survey. *Future Gener. Comput. Syst.* **2019**, *97*, 219–235. [CrossRef]
23. Botta, A.; de Donato, W.; Persico, V.; Pescapé, A. Integration of Cloud Computing and Internet of Things: A Survey. *Future Gener. Comput. Syst.* **2016**, *56*, 684–700. [CrossRef]
24. Burns, B.; Grant, B.; Oppenheimer, D.; Brewer, E.; Wilkes, J. *Borg, Omega, and Kubernetes*; Association for Computing Machinery: New York, NY, USA, 2016; Volume 14, pp. 70–93. [CrossRef]
25. Verma, A.; Pedrosa, L.; Korupolu, M.; Oppenheimer, D.; Tune, E.; Wilkes, J. Large-Scale Cluster Management at Google with Borg. In *EuroSys '15: Proceedings of the Tenth European Conference on Computer Systems*; Association for Computing Machinery: New York, NY, USA, 2015; pp. 1–17. [CrossRef]
26. Kubernetes. Kubernetes Documentation. 2024. Available online: https://kubernetes.io/docs/ (accessed on 20 May 2025).
27. Vayghan, L.A.; Saied, M.A.; Toeroe, M.; Khendek, F. A Kubernetes Controller for Managing the Availability of Elastic Microservice Based Stateful Applications. *J. Syst. Softw.* **2021**, *175*, 110924. [CrossRef]
28. Santos, J.; Wauters, T.; Volckaert, B.; De Turck, F. Towards Delay-Aware Container-Based Service Function Chaining in Fog Computing. In *Proceedings of the NOMS 2020—2020 IEEE/IFIP Network Operations and Management Symposium*; IEEE: Piscataway, NJ, USA, 2020; pp. 1–9. [CrossRef]
29. Vaño, R.; Lacalle, I.; Sowiński, P.; Sánchez-Julián, R.; Palau, C.E. Cloud-Native Workload Orchestration at the Edge: A Deployment Review and Future Directions. *Sensors* **2023**, *23*, 2215. [CrossRef] [PubMed]
30. Rosendo, D.; Costan, A.; Valduriez, P.; Antoniu, G. Distributed Intelligence on the Edge-to-Cloud Continuum: A Systematic Literature Review. *J. Parallel Distrib. Comput.* **2022**, *166*, 71–94. [CrossRef]
31. Dorigo, M.; Birattari, M.; Stutzle, T. Ant Colony Optimization. *IEEE Comput. Intell. Mag.* **2006**, *1*, 28–39. [CrossRef]
32. Kennedy, J.; Eberhart, R. Particle Swarm Optimization. In *Proceedings of ICNN'95—International Conference on Neural Networks*; IEEE: Piscataway, NJ, USA, 1995; Volume 4, pp. 1942–1948. [CrossRef]
33. McMahan, B.; Moore, E.; Ramage, D.; Hampson, S.; y Arcas, B.A. Communication-Efficient Learning of Deep Networks from Decentralized Data. In *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics (AISTATS)*; PMLR: Cambridge, MA, USA, 2017; pp. 1273–1282. Available online: https://proceedings.mlr.press/v54/mcmahan17a.html (accessed on 7 October 2026).
34. Kairouz, P.; McMahan, H.B. Advances and Open Problems in Federated Learning. *Found. Trends Mach. Learn.* **2021**, *14*, 1–210. [CrossRef]
35. Zhou, Z.; Chen, X.; Li, E.; Zeng, L.; Luo, K.; Zhang, J. Edge Intelligence: Paving the Last Mile of Artificial Intelligence with Edge Computing. *Proc. IEEE* **2019**, *107*, 1738–1762. [CrossRef]
36. Istio. Istio Ambient Mesh: Sidecar-Less Service Mesh. Ambient Mesh Reached GA in Istio v1.24. 2024. Available online: https://istio.io/latest/blog/2024/ambient-reaches-ga/ (accessed on 15 November 2025).
37. Cilium. Cilium: eBPF-Based Networking, Observability, and Security. 2024. Available online: https://cilium.io/ (accessed on 15 November 2025).
38. Lee, F.S. A Comparative Analysis of Service Mesh Proxy Architectures: From Sidecars to Ambient and Proxyless Models in Cloud-Native Environments. *Int. J. Mod. Comput. Sci. It Innov.* **2025**, *2*, 64–73.
39. Simas, A.J.; Rothenberg, C.E.; Pongrácz, G. eSeMeshA: eBPF-Based Service Mesh Acceleration for Cloud-Native Infrastructures. In *Proceedings of the 2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*; IEEE: Piscataway, NJ, USA, 2025; pp. 338–342. [CrossRef]
40. Dean, J.; Barroso, L.A. The Tail at Scale. *Commun. ACM* **2013**, *56*, 74–80. [CrossRef]

41. Sigelman, B.H.; Barroso, L.A.; Burrows, M.; Stephenson, P.; Plakal, M.; Beaver, D.; Jaspan, S.; Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report Dapper-2010-1, Google, 2010. Available online: https://research.google/pubs/dapper-a-large-scale-distributed-systems-tracing-infrastructure/ (accessed on 7 October 2026).
42. Zhu, X.; She, G.; Xue, B.; Zhang, Y.; Zhang, Y.; Kelley, A.; Krishnamurthy, A.; Anderson, T. Dissecting Overheads of Service Mesh Sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing (SoCC)*; Association for Computing Machinery: New York, NY, USA, 2023; pp. 142–157. [CrossRef]
43. Ueda, T.; Nakaike, T.; Ohara, M. Workload Characterization for Microservices. In *Proceedings of the 2016 IEEE International Symposium on Workload Characterization (IISWC)*; IEEE: Piscataway, NJ, USA, 2016; pp. 1–10. [CrossRef]
44. Henning, S.; Hasselbring, W. Benchmarking Scalability of Stream Processing Frameworks Deployed as Microservices in the Cloud. *J. Syst. Softw.* **2024**, *208*, 111879. [CrossRef]
45. Eismann, S.; Bezemer, C.P.; Shang, W.; Okanović, D.; van Hoorn, A. Microservices: A Performance Tester's Dream or Nightmare? In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*; Association for Computing Machinery: New York, NY, USA, 2020; pp. 138–149. [CrossRef]
46. Rodrigues, H.; Rito Silva, A.; Avritzer, A. Assessment of Performance and Its Scalability in Microservice Architectures: Systematic Literature Review. *J. Syst. Softw.* **2025**, *230*, 112500. [CrossRef]
47. Meijer, W.; Trubiani, C.; Aleti, A. Experimental Evaluation of Architectural Software Performance Design Patterns in Microservices. *J. Syst. Softw.* **2024**, *218*, 112183. [CrossRef]
48. Karabey Aksakalli, I.; Çelik, T.; Can, A.B.; Tekinerdoğan, B. Deployment and Communication Patterns in Microservice Architectures: A Systematic Literature Review. *J. Syst. Softw.* **2021**, *180*, 111014. [CrossRef]
49. Marchese, A.; Tomarchio, O. Enhancing the Kubernetes Platform with a Load-Aware Orchestration Strategy. *SN Comput. Sci.* **2025**, *6*, 224. [CrossRef]
50. Ahmad, H.; Treude, C.; Wagner, M.; Szabo, C. Towards Resource-Efficient Reactive and Proactive Auto-Scaling for Microservice Architectures. *J. Syst. Softw.* **2025**, *225*, 112390. [CrossRef]
51. Luo, S.; Lin, C.; Ye, K.; Xu, G.; Zhang, L.; Yang, G.; Xu, H.; Xu, C. Optimizing Resource Management for Shared Microservices: A Scalable System Design. *ACM Trans. Comput. Syst.* **2024**, *42*, 1–28. [CrossRef]
52. Arman, A.; Monticelli, M.; Firmani, D.; Leotta, F.; Mecella, M. An Adaptive Toolbox for Computing Throttling Rate Limits in Web APIs. *IEEE Trans. Serv. Comput.* **2026**, *19*, 141–152. [CrossRef]
53. Kourtis, M.A.; Xilouris, G. *Fortio Tail-Latency Benchmarks for Federated and Peer-to-Peer Microservice Coordination on an Istio Service Mesh*; Zenodo Dataset, v1; Zenodo: Geneva, Switzerland, 2026. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.