Adaptive Circuit Breaker Configuration for Transition from Istio Sidecar to Ambient Mode*

Jiwon Yoo, Seoyeon Kang, Kahyun Kim, Hyeyeon Hyeon, and Seongmin Kim†

Sungshin Women's University, Seongbuk-gu, Seoul, Korea
{220246055, 20211036, 20211039, 20221143, sm.kim}@sungshin.ac.kr

Abstract

Microservice Architecture (MSA) improves scalability and resilience in cloud-native systems, yet its tightly coupled inter-service dependencies often lead to cascading failures. To mitigate such risks, service meshes employ circuit breakers (CB) as key resilience mechanisms. However, Istio's recent shift from sidecar to ambient mode fundamentally changes CB behavior and parameter sensitivity. This study experimentally investigates how the transition of the Istio service mesh data plane from sidecar to ambient mode affects the operational behavior of its circuit breaker (CB) mechanism. The results show that the ambient architecture, which manages connections at the node level through shared proxies, exhibits earlier throttling and higher contention under identical configurations. Throughput stability was restored only when parameters were tuned proportionally to pod count. These findings highlight that legacy CB configurations optimized for the sidecar model cannot be directly reused in ambient environments, emphasizing the need for architecture-aware parameter tuning and providing practical guidance for optimizing resilience policies in next-generation service mesh deployments.

Keywords— Istio, Service mesh, Circuit breaking, Sidecar, and Ambient

1 Introduction

The Microservice Architecture (MSA) decomposes large-scale service systems into multiple independent components, enabling each service to be deployed, scaled, and recovered autonomously. This architectural paradigm enhances operational efficiency and system resilience in cloud-native environments and has become the foundational framework for global-scale platforms, such as Netflix, Google, and Amazon [3, 10, 1]. However, as MSA-based systems evolve, inter-service communication grows increasingly complex, and dependencies among services become tighter. This heightened interconnectedness often results in cascading failures, where a single point of failure propagates through multiple services, amplifying its impact on overall system stability.

To mitigate such risks, the Circuit Breaker (CB) pattern has been introduced as a representative resilience mechanism in microservice-based systems [16]. CB enhances system stability by isolating failing services through a fail-fast mechanism and restoring connectivity once recovery is detected (fail-recovery). Therefore, proper CB configuration is crucial for maintaining Service Level Agreement (SLA) compliance and ensuring resilience in distributed service environments [11, 15].

With the emergence of service meshes, a new infrastructure layer has been introduced to manage inter-service communication independently of application logic. By delegating traffic management, security, and fault control to a dedicated proxy layer, service meshes simplify resilience control and observability. Among them, Istio has recently introduced a new data-plane model called ambient mode, which replaces traditional per-pod proxies (sidecar) with a node-level proxy (zTunnel) and an optional L7 Waypoint proxy [4]. This architectural shift changes how CB operates and how connection

*Proceedings of the 9th International Conference on Mobile Internet Security (MobiSec'25), Article No. W3, December 16-18, 2025, Sapporo , Japan.

†Corresponding author

pools are managed, potentially altering the behavior of resilience mechanisms that were optimized for sidecar-based deployments [9].

Despite this architectural transition, CB parameters are still configured at the pod level by cluster operators. In practice, cluster operators often migrate existing pod configurations without re-tuning, relying on previously validated settings to minimize service disruption and configuration overhead [12] As a result, existing configurations are often reused in the new environment without proper re-evaluation. However, deriving appropriate configurations for Ambient Mode from existing per-Pod settings is non-trivial, as multiple inter-dependent parameters influence CB triggering behavior. This raises concerns about their effectiveness under different architectural scopes, per-pod (sidecar mode) vs. per-node (ambient mode). Understanding these behavioral differences is essential for maintaining performance stability during migration.

Accordingly, we conduct an empirical analysis of how the transition from Sidecar to ambient Mode influences Circuit Breaker (CB) performance. In particular, we examine the behavior of three key parameters—maxConnections, maxRequestsPerConnection, and http1MaxPendingRequests—under identical experimental conditions in both modes. Also, we consider the quality-of-service (QoS) when varying CB parameters under regular and extreme load conditions. Our evaluation results demonstrate that ambient mode maintains more stable throughput, while sidecar mode shows aggressive rejection behavior and shorter tail latency. Based on the findings, we derive parameter tuning guidelines optimized for the ambient mode to enhance resilience and ensure stable traffic management.

2 Background & Related work

2.1 Circuit Breaker in Service Mesh

A service mesh is a dedicated infrastructure layer that manages inter-service communication independently of application logic. By separating the control plane, which governs traffic policies and configurations, from the data plane, which processes actual network traffic, service meshes enable consistent enforcement of security, routing, and fault-tolerance policies across microservices [2].

Among existing implementations, Istio is the most widely adopted framework due to its rich policy abstractions and high extensibility. Initially, Istio employed a sidecar-based architecture, where each pod runs an Envoy proxy responsible for managing connection pools, request queues, and resilience policies on a per-service basis [8]. While this design provides fine-grained control over service behavior, it also introduces additional operational and resource overhead as the number of microservices increases [17].

Circuit breaking in Istio is implemented indirectly through the `DestinationRule` resource [6], which specifies connection pool and outlier detection limits rather than providing a standalone circuit breaker object. When the thresholds defined by these parameters are exceeded, excessive requests are either rejected or delayed, preventing cascading failures.

Table 1 summarizes the major configuration parameters that realize circuit-breaking functionality in Istio [5]. The key parameters include `maxConnections`, which limits the number of concurrent upstream connections; `http1MaxPendingRequests`, which defines the maximum number of pending HTTP requests that can be queued; and `maxRequestsPerConnection`, which determines how many requests a single connection can handle before being recycled. Together, these mechanisms enable the system to fail fast during transient faults and recover gracefully once stability is restored.

2.2 Istio Architectural Models: Sidecar vs. Ambient

Istio supports two data-plane models, sidecar and ambient. These modes fundamentally differ in their traffic-processing scope and fault-isolation granularity.

Sidecar. In the traditional sidecar model [8], an Envoy proxy is injected into every Pod, and each proxy independently manages request queues, connection pools, retry logic, and circuit-breaking configurations at the L7 (HTTP) layer, as illustrated in Figure 1. While this architecture provides strong

Table 1: Major Istio Circuit Breaker Configuration Parameters

Category	Parameter	Description
Connection Pool	`maxConnections`	Limits the maximum number of concurrent upstream connections. When the limit is exceeded, new connections are immediately rejected (Fail-Fast).
	`http1MaxPending Requests`	Limits the maximum number of pending HTTP requests. When the queue becomes saturated, new requests are blocked to prevent latency propagation.
	`maxRequests PerConnection`	Limits the number of requests handled per connection. When the threshold is reached, the connection is recycled to reduce long-lived load.

fault isolation and detailed traffic control, it also introduces several limitations. First, resource overhead increases linearly with the number of pods, since each pod operates a dedicated Envoy instance that consumes CPU and memory resources. Second, state inconsistency may arise because each pod maintains an independent circuit-breaker state, leading to unbalanced request handling and uneven connection saturation across services. Finally, latency amplification occurs as multiple L7 proxies add extra network hops, contributing to higher tail latency.

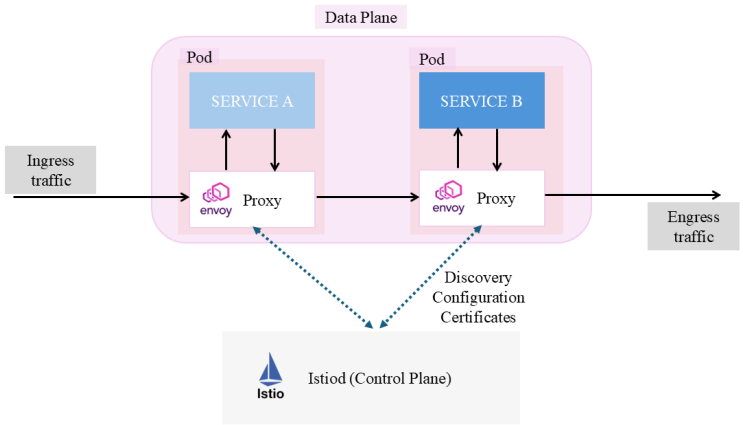


Figure 1: Istio Service mesh sidecar architecture

Ambient. To address these challenges, Istio introduced the ambient mode in 2023 [7]. As shown in Figure 2, ambient mode eliminates per-pod proxies and replaces them with a node-level L4 proxy (ztunnel) and an optional L7 waypoint proxy. In this model, all pod traffic on the same node is routed through ztunnel, shifting circuit-breaker enforcement from the per-pod level to the per-node level.

This architectural shift offers several advantages. First, the traffic path becomes unified, as ztunnel aggregates traffic from multiple pods and centralizes connection management. Also, the resource pool is shared, meaning that circuit-breaker parameters operate across node-level resources rather than isolated per-pod pools. Finally, the control overhead is reduced, since fewer proxies simplify configuration distribution and improve scalability.

In summary, while the sidecar model prioritizes service-level isolation and fine-grained control, the ambient model emphasizes efficiency and simplified management through shared resources. This architectural transition may alter circuit-breaker behavior and its sensitivity under varying workload

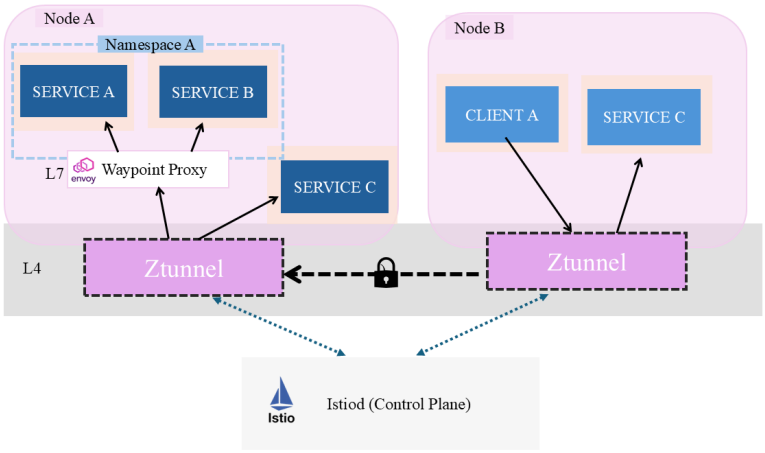


Figure 2: Istio Service mesh ambient architecture

conditions.

2.3 Related Work

Several studies have explored adaptive control of circuit breakers (CB) in microservice systems. Sedghpour et al. proposed an adaptive circuit breaker controller that dynamically adjusts CB thresholds and retry parameters based on the ratio of latency to queue length, thereby improving throughput and availability under heavy load [14, 13]. Although these studies have advanced adaptive CB configuration techniques, they primarily focused on tuning strategies within a fixed architecture (e.g., sidecar).

In contrast to previous approaches, this study focuses on how the underlying data-plane architecture, specifically, the transition from per-pod (sidecar) to per-node (ambient) proxies, affects circuit breaker (CB) behavior and overall performance. To this end, we empirically examine how Istio's architectural transition influences CB behaviors by evaluating differences in QPS, latency, and success ratio under identical configurations across both modes.

3 Problem Statement and Evaluation Setup

In this study, we focus on answering the following research questions (RQs):

RQ1. How do circuit breakers behave differently between sidecar and ambient modes under identical configurations?

RQ2. How should CB parameters be reconfigured to maintain stability during the transition to ambient mode?

RQ3. How does ambient mode's shared architecture affect CB sensitivity under varying load conditions?

These questions are designed to clarify the behavioral differences and configuration implications of Istio's circuit-breaking mechanism across different data-plane architectures. To address them, we conduct experiments using the Istio benchmark sample application, Bookinfo, under identical load conditions across both operational modes. The Bookinfo application composes of a simple service chain, productpage–details–reviews–ratings. Note that we select this benchmark because its clearly

defined inter-service call path and error propagation behavior make it well-suited for analyzing circuit breaker dynamics.

We focus on the connection pool–based control mechanism because it enables direct observation of connection- and request-level blocking effects during traffic bursts. Note that the impact of outlier detection is reserved for future investigation. In addition, we evaluate the service performance using three key indicators: success rate, queries per second (QPS), and P95/P99 Latency. The overall experimental setup and parameter configurations are summarized in Table 2.

Table 2: Experimental setup and configurations.

Category	**Configuration**
Cluster Setup	Kubernetes (1 master, 2 workers)
Istio Version	1.23.1 — Separate namespaces within same cluster for Sidecar and Ambient modes
Applications	*Bookinfo* (simple chain)
Load Generator	Fortio (`fortio/fortio:latest`), QPS 100–40,000, duration 30s
Experiment Modes	Sidecar / Ambient (with zTunnel and Waypoint)
CB Mechanism	(1) Connection Pool (2) Outlier Detection
Test Parameters	`maxConnections`, `maxRequestsPerConnection`, `http1MaxPendingRequests`
Measured Metrics	Avg/Max latency, QPS, Success rate (2xx), Failure rate (5xx)
Test Cases	Regular Load / Extreme Load (load scaling per mode)
Data Collection	Fortio JSON logs → CSV extraction → visualization
Evaluation Goal	Compare CB behavior/sensitivity between Sidecar and Ambient modes; derive tuning guidelines

Each experiment was performed under two traffic conditions: 1) regular and 2) extreme. In the regular condition, we conducted a sensitivity analysis under normal load, measuring latency (P50/P90/P99), QPS, success ratio, and error rate with the following parameters: QPS of 100, 200, and 500; concurrency of 8, 16, and 32; and a duration of 30 seconds. In the case of extreme condition, we evaluated overload scenarios designed to trigger circuit breaker activation, using relatively higher traffic levels (QPS: 1000, 2000, 4000; concurrency: 64, 128, 256; duration: 30 seconds), respectively.

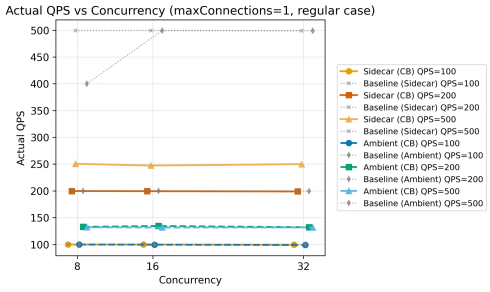
The experiment focused on three key parameters of Istio's connection pool configuration: *maxConnections, maxRequestsPerConnection*, and *http1MaxPendingRequests*. Each parameter was evaluated independently while keeping the remaining parameters at their default values defined in the official Istio and Envoy upstream standards [6]. Specifically, maxConnections and http1MaxPendingRequests were varied from 1 to 32 (1, 2, 4, 8, 16, 32) to observe how increasing connection and pending-request limits affect blocking and latency behavior. For maxRequestsPerConnection, values of 0, 1, 2, 4, 8, and 16 were tested to compare the impact of enabling connection reuse (KeepAlive) versus imposing strict request-per-connection limits. Across all tests, other circuit breaker parameters (e.g., `outlierDetection`) remained disabled or set to default, ensuring that observed performance variations originated solely from connection pool adjustments.

4 Impact of Configuration on Circuit Breaking

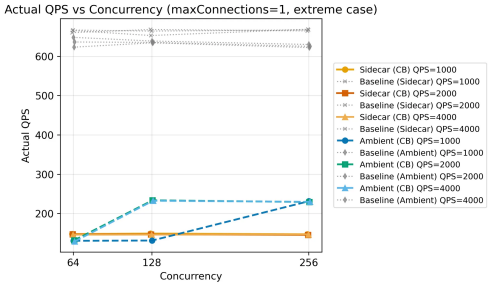
In this section, we analyze the performance of Istio **Connection Pool**–based circuit breaker by varying the aforementioned three key control parameters.

4.1 Impact of Concurrent Connections

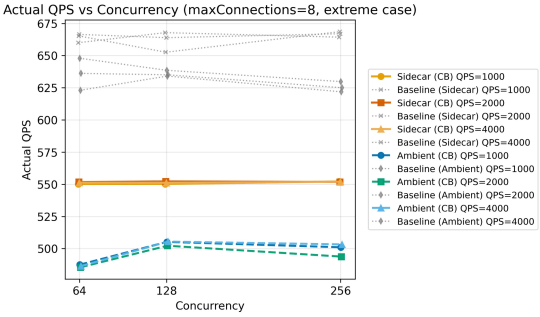
The parameter *maxConnections* controls the maximum number of concurrent connections allowed between services. Lower values restrict connection parallelism, causing faster queue saturation and potentially increasing request blocking under high load. We examine the performance impact of varying this parameter in sidecar and ambient modes in terms of throughput (Figure 3) and latency (Figure 4) under identical load conditions.



(a) Regular case (maxc = 1)



(b) Extreme case (maxc = 1)

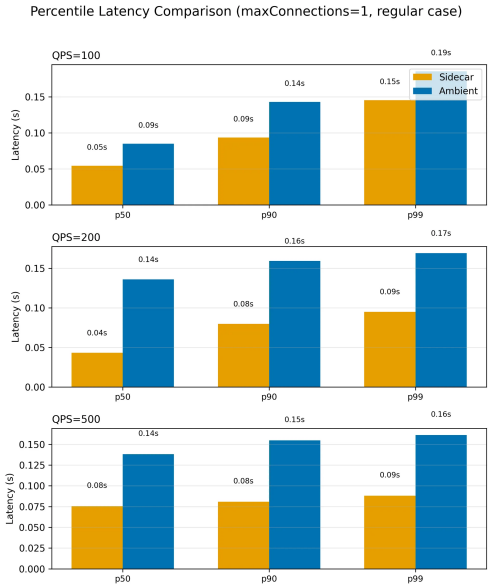


(c) Extreme case (maxc = 8)

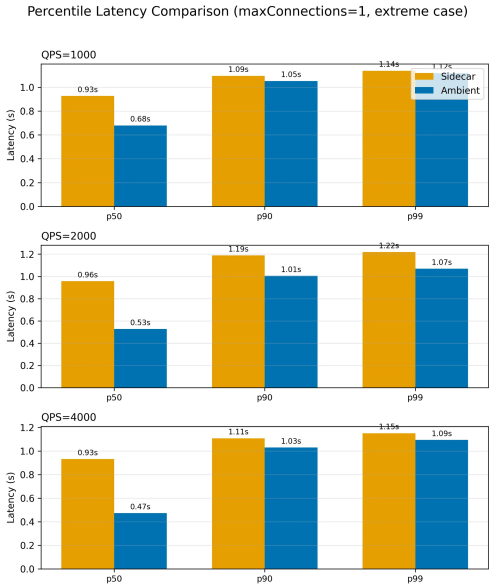
Figure 3: Throughput comparison between sidecar and ambient modes. Note, maxc denotes maxConnections.

Throughput. Under low-load conditions (QPS 100, Concurrency 8), both sidecar and ambient modes achieved over 99% success rates and met their target throughput (Figure 3a). However, when the connection limit was severely constrained (maxConnections = 1), a noticeable performance gap emerged between the two modes. At QPS 500, the sidecar mode maintained throughput around 250 QPS, while the ambient mode was limited to 130–140 QPS due to its shared connection pool. In the sidecar architecture, each pod's Envoy proxy independently manages its own pool, ensuring minimal parallelism even under restrictive settings. In contrast, multiple pods share a single namespace-level TCP session pool in the ambient mode, resulting in sequential request handling and reduced service-level efficiency.

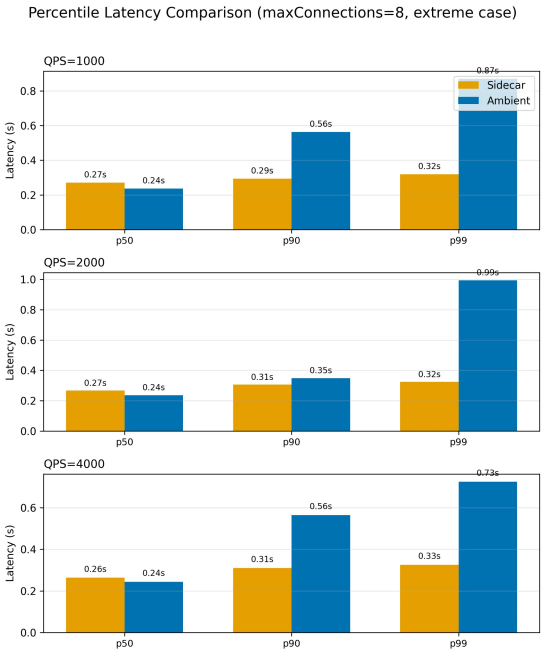
Under high-load conditions, the difference became more evident (Figure 3b). When maxConnections is set to 1, sidecar throughput rapidly declined as per-pod queues saturated, whereas the ambient mode temporarily sustained higher throughput (approximately 130–240 QPS) through node-level session sharing. This L4-layer sharing alleviated early bottlenecks, but as maxConnections increased (≥ 8), contention within the shared zTunnel pool emerged (Figure 3c). The increasing number of concurrent sessions introduced coordination overhead between zTunnel and Waypoint, gradually offsetting the benefit of shared distribution.



(a) Regular case (maxc = 1)



(b) Extreme case (maxc = 1)



(c) Extreme case (maxc = 8)

Figure 4: Latency comparison between sidecar and ambient modes (maxConnections).

In summary, the sidecar mode preserved stable throughput through per-pod isolation, while the ambient mode exhibited throughput saturation and moderate decline beyond the mid-load range. These findings reveal an inherent trade-off between scalability and stability, as the ambient architecture becomes increasingly sensitive to contention within its shared L4 resource domain.

Latency. Under moderate load (QPS 100–500) with maxConnections = 1, the sidecar mode consistently delivered lower latency across all percentiles (Figure 4a). Median latency remained below 0.1 s, while ambient mode showed slightly higher tail values ($p_{99} \approx 0.16$–0.19 s). This difference stems from sidecar's per-pod connection pools, which allow parallel request handling and shorter queues, whereas ambient serializes traffic through a node-level zTunnel, introducing marginal delay. Thus, pod-level isolation yields lower and more uniform latency under normal load. At higher loads (QPS 1000–4000) (Figure 4b), the pattern reversed. In sidecar, the L7 queue saturated quickly, pushing tail latency above 1 s ($p_{99} > 1.1$ s).

In contrast, ambient maintained smoother latency growth ($p_{99} \approx 0.6$–0.8 s) because its layered data plane separates L4 connection management (zTunnel) from L7 routing (Waypoint), preventing deep queue buildup. When the connection limit increased (maxConnections = 8) (Figure 4c), both modes showed similar median latency ($p_{50} \approx 0.25$ s), indicating handshake overhead was removed. However, ambient exhibited higher tail latency ($p_{99} \approx 0.7$–1.0 s) than sidecar ($p_{99} \approx 0.3$ s) due to contention within the shared zTunnel pool, where multiple pods compete for limited L4 resources. Sidecar maintained stable latency because per-pod Envoy isolation prevented congestion and resource contention.

In summary, sidecar mode provides consistently lower tail latency through distributed queue isolation, whereas ambient offers smoother scaling under extreme load but becomes sensitive to contention once connection limits relax. This reflects a trade-off between per-pod responsiveness and node-level scalability.

4.2 Impact of Number of Requests

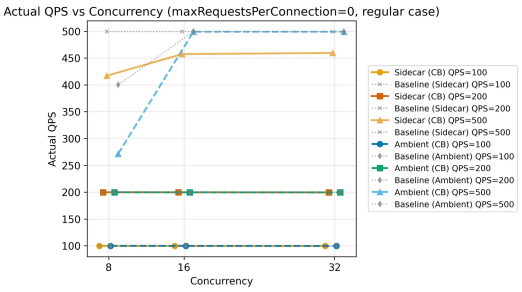
The parameter *maxRequestsPerConnection* defines the number of requests that can be served over a single TCP connection. It directly governs the persistence of HTTP KeepAlive sessions and the efficiency of connection reuse, thereby affecting request throughput and connection management behavior.

Throughput. Under moderate load (QPS ≤ 500), changing maxRequestsPerConnection had minimal influence on overall throughput (Figure 5a and Figure 5c). At higher loads (QPS ≥ 1000), however, throughput diverged sharply depending on this parameter (Figure 5b and Figure 5d). When maxRequestsPerConnection = 1 (no reuse), every request initiated a new TCP/TLS handshake, accumulating connection-setup overhead. In sidecar, each pod maintained an independent pool, allowing partial handshake parallelism, but Envoy reconstructed the L7 routing path for every request, sharply reducing effective throughput and increasing CPU use. In ambient, a single node-level zTunnel pool serialized pod traffic, yet shared connection management mitigated handshake overhead, resulting in higher QPS than sidecar under high concurrency.

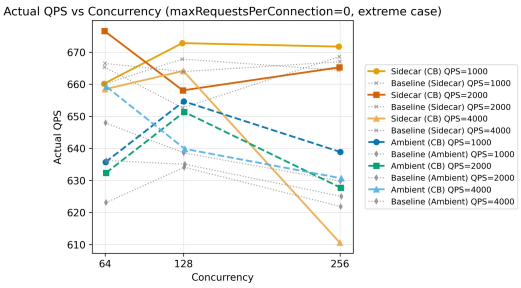
With maxRequestsPerConnection = 0 (unlimited reuse), throughput initially rose through persistent connections. Beyond Envoy's processing capacity, however, new requests accumulated in internal queues; throughput then plateaued and degraded as the system reached saturation. Ambient maintained steadier throughput under overload because its L4-level zTunnel distributed traffic more evenly and reduced L7 overhead by delegating application-layer routing to the Waypoint proxy.

It is worth noting that sidecar achieved higher concurrency under non-reuse settings but suffered steeper throughput collapse when persistent connections overloaded the proxy. Ambient delivered more stable performance, particularly under sustained high traffic, highlighting its resilience to long-lived connection stress.

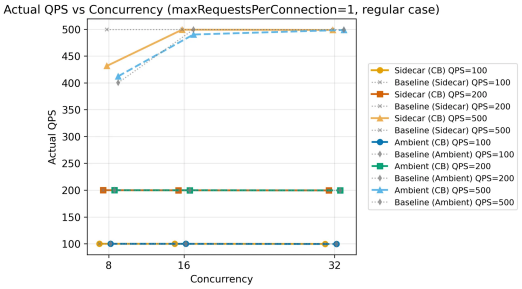
Latency. Latency characteristics exhibited an opposite trend to throughput, as shown in Figure 6. With maxRequestsPerConnection = 0 (unlimited reuse) (Figure 6a and Figure 6b), sidecar kept long-lived L7 connections that queued multiple requests on the same channel. Once Envoy's processing capacity was exceeded, internal queues saturated, sharply increasing mean and tail latency. In ambient,



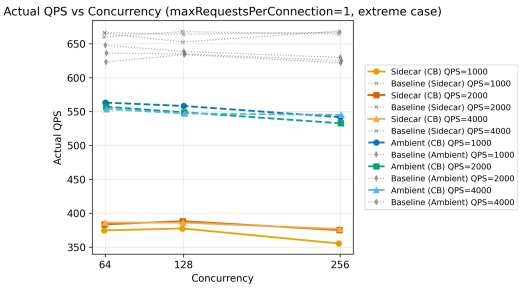
(a) Regular case (maxr = 0)



(b) Extreme case (maxr = 0)



(c) Regular case (maxr = 1)



(d) Extreme case (maxr = 1)

Figure 5: Throughput comparison between sidecar and ambient modes. Note, maxr denotes maxRequestsPerConnection.

L4-level reuse through zTunnel and L7 delegation to the Waypoint proxy limited queue buildup and maintained smoother latency growth, even under overload.

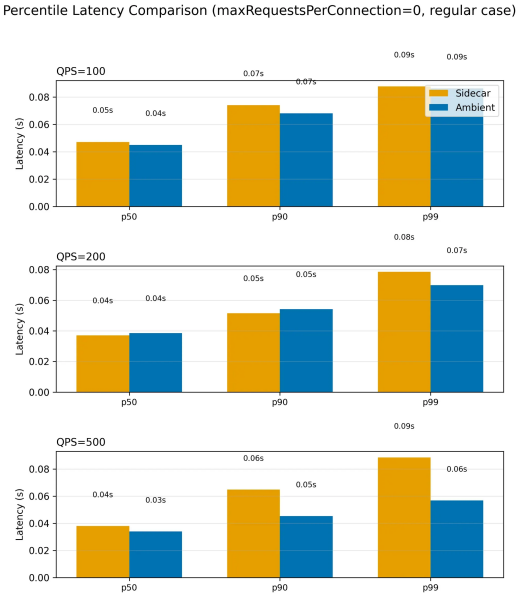
When maxRequestsPerConnection = 1 (no reuse) (Figure 6c and Figure 6d), sidecar achieved slightly lower latency since each pod's Envoy created connections in parallel, distributing handshake overhead. Ambient processed sessions sequentially, producing modestly higher average and p99 latency due to serialization overhead. Under extreme load (QPS $\geq$ 1000), these trends intensified: sidecar experienced pronounced tail spikes (p99 $>$ 1 s) from queue saturation and CPU exhaustion, whereas ambient's layered L4/L7 design kept queues short and suppressed tail-latency escalation.

In summary, sidecar incurred severe latency penalties under persistent connection overload, while ambient mitigated latency growth through its hierarchical data-plane separation, offering more stable and resilient performance.

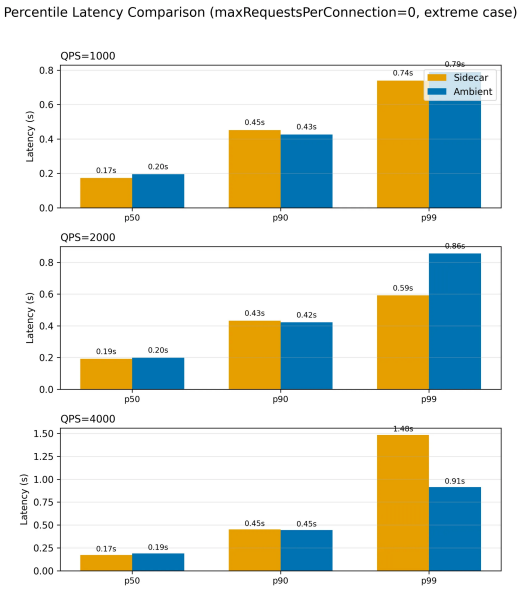
4.3 Impact of Queued HTTP Requests

The parameter *http1MaxPendingRequests* defines the maximum number of requests that can be concurrently queued within the proxy. Smaller queue limits cause excess requests to be rejected immediately, thereby lowering average latency but potentially reducing overall throughput. Conversely, larger queues permit more pending requests to accumulate, which improves request retention under high load but increases tail latency due to prolonged waiting times.

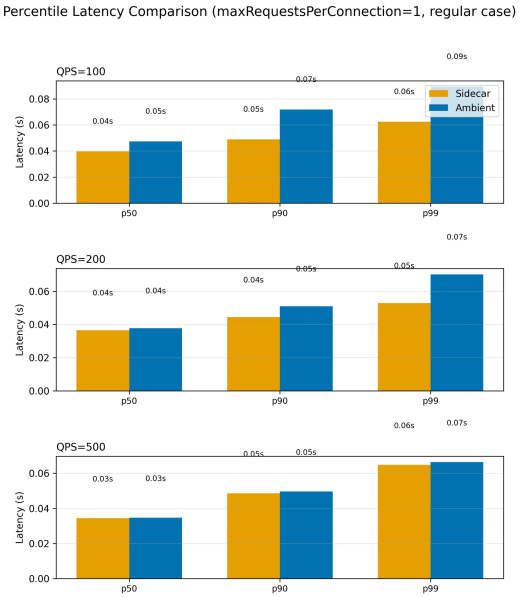
Throughput. Under regular load conditions (QPS $\leq$ 500), sidecar and ambient modes exhibited nearly identical throughput across all concurrency levels (Figure 7a). A slight increase was observed in ambient at QPS = 500, likely due to lightweight parallelism between the zTunnel and Waypoint layers.



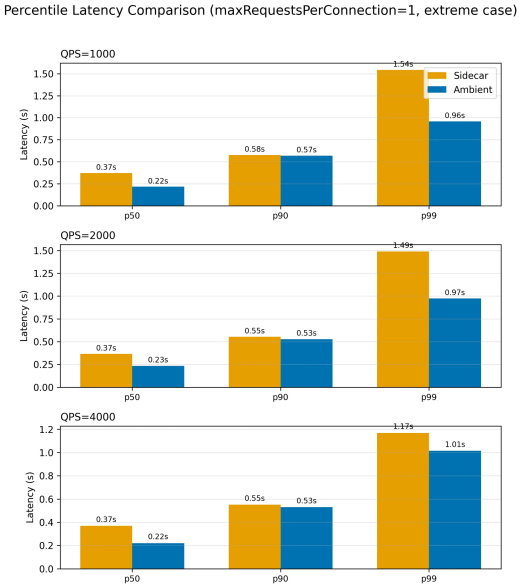
(a) Regular case (maxr = 0)



(b) Extreme case (maxr = 0)

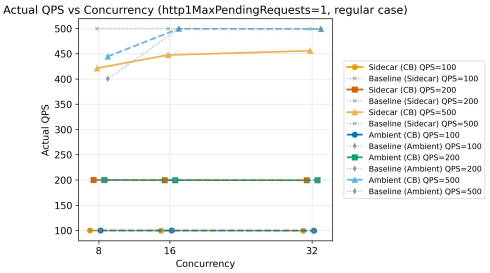


(c) Regular case (maxr = 1)

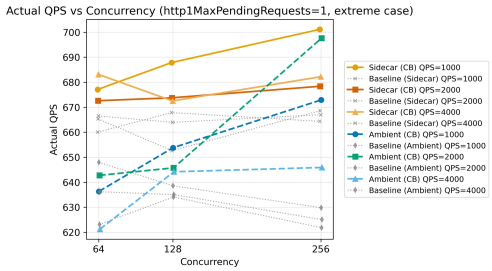


(d) Extreme case (maxr = 1)

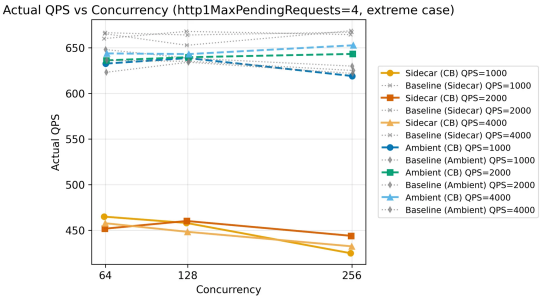
Figure 6: Latency comparison between sidecar and ambient modes (maxRequestsPerConnection).



(a) Regular case (http1max = 1)



(b) Extreme case (http1max = 1)



(c) Extreme case (http1max = 4)

Figure 7: Throughput comparison between Sidecar and Ambient modes (http1MaxPendingRequest).

Overall, queue limits had negligible influence on throughput when the system was not under stress.

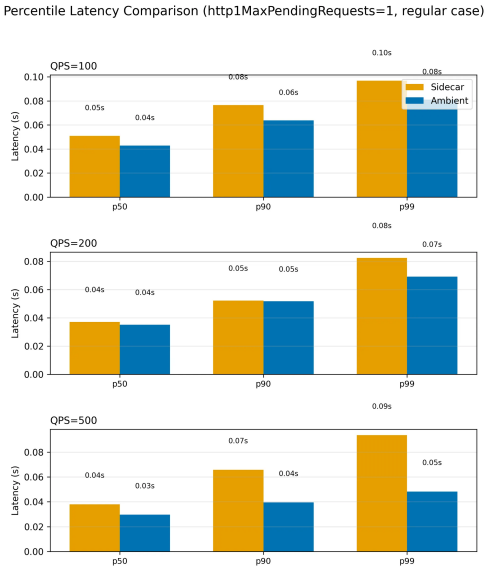
Under extreme load (QPS $\geq$ 1000), throughput behavior diverged more clearly with queue size (Figure 7b). With http1MaxPendingRequests = 1, sidecar showed modest throughput gains as concurrency increased, driven by early rejection of overflow requests. Because Fortio counts both successful and failed (HTTP 503) transactions as completed operations, these rapid rejections inflated the measured QPS relative to actual processing efficiency. Ambient, in contrast, maintained steady but slightly lower throughput, as its shared zTunnel serialized sessions but avoided oscillations under load. When the queue limit was relaxed (http1MaxPendingRequests $\geq$ 4), sidecar's per-pod L7 queues began to saturate, causing throughput to plateau and then decline. Ambient maintained balanced throughput by distributing pending requests across its layered zTunnel–Waypoint structure.

In summary, sidecar benefits from short-term throughput gains under tight queue constraints through early rejection but suffers from L7 saturation as queues expand. Ambient provides steadier, more balanced throughput by mitigating local queue buildup through multi-layer request distribution.

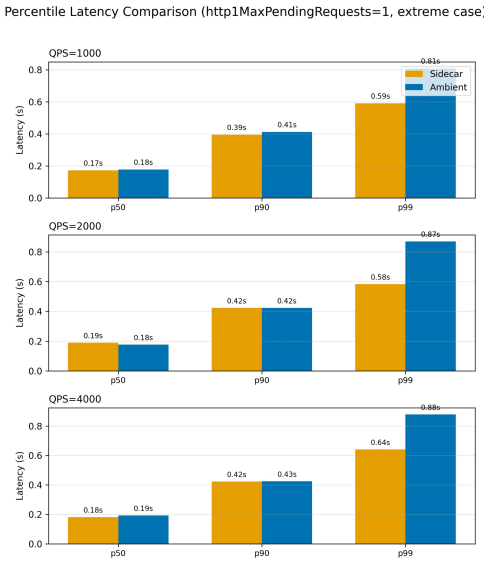
Latency. Again, latency patterns generally mirrored the inverse of throughput trends. Under regular load, smaller queues (http1MaxPendingRequests = 1) reduced average latency in both modes by rejecting excess requests immediately, though retry operations caused a slight rise in tail latency.

Under extreme load (Figure 8b), the two modes diverged. Sidecar achieved low average latency through early rejection but suffered unstable tail latency due to retry overhead and transient bursts. Ambient, while maintaining similar average latency, showed higher p99 values as session serialization through the shared zTunnel introduced occasional waiting delays.

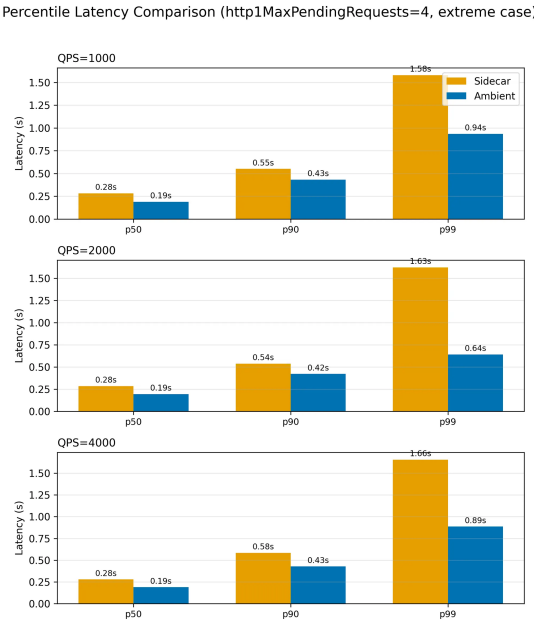
When the queue limit increased ($\geq$ 4) (Figure 8c), sidecar accumulated pending requests in its L7 buffer, resulting in sharp tail-latency growth. In contrast, ambient maintained moderate average latency and controlled tail-latency escalation by distributing requests across its layered zTunnel–Waypoint



(a) Regular case (http1max = 1)



(b) Extreme case (http1max = 1)



(c) Extreme case (http1max = 4)

Figure 8: Latency comparison between Sidecar and Ambient modes (http1MaxPendingRequest).

structure, which mitigated deep queue buildup.

5 Implications and Lesson Learned

The experimental findings highlight that architectural transitions in service meshes fundamentally alter the operational scope of CB. Parameters optimized for the sidecar model cannot be directly applied to the ambient architecture, as the shift from per-pod to shared node-level proxies changes how rejection, queuing, and load isolation occur. In particular, the absence of rapid L7-level rejection in ambient mode can unintentionally cause throughput degradation or connection saturation when legacy CB configurations are reused without adjustment. Accordingly, architecture-aware reconfiguration of CB parameters is required, as discussed in the following lessons learned.

Lesson 1: Reconfiguring Connection Limits for Ambient Architecture. Applying identical connection-related parameters leads to significantly earlier throttling in the ambient environment, necessitating the relaxation of configuration thresholds. Experimental results show that *maxConnections* must be increased in ambient mode to achieve a throughput comparable to that of the sidecar architecture.

Lesson 2: Trade-off between Performance and Stability. The sidecar mode enhances system stability by enabling rapid fail-fast activation to prevent cascading failures, but this comes at the cost of higher average latency and lower throughput. In contrast, the ambient mode improves resource efficiency and reduces average latency, yet exhibits slower recovery and longer tail latency under stress. Therefore, when stability and fault isolation are prioritized, a fail-fast–oriented (sidecar-type) configuration is preferable, whereas a throughput-oriented (ambient-type) configuration is more suitable when operational efficiency is the primary objective.

Lesson 3: Adaptive Thresholding. In dynamically changing traffic environments, static CB thresholds are insufficient to respond effectively. Experimental results revealed cases where the circuit breaker was triggered either prematurely or with excessive delay during sudden traffic surges. Therefore, an adaptive circuit breaker mechanism is needed. For example, it can continuously adjust its activation threshold according to real-time service indicators such as QPS, latency, and error rate.

6 Conclusion

This study experimentally analyzed how Istio's connection pool–based circuit breaker behaves differently across the sidecar and ambient data-plane architectures. Even under identical configurations, the two modes exhibited distinct performance sensitivities arising from their proxy-layer design. The results highlight the importance of mode-specific parameter tuning, rather than directly reusing sidecar configurations in ambient environments. These findings provide practical guidance for optimizing circuit breaker settings in node-based service meshes and lay the groundwork for adaptive, autonomous resilience control in future frameworks. As future work, we plan to extend the analysis to outlier detection, another key reliability mechanism, and evaluate diverse benchmark applications with complex service chains in large-scale MSA environments.

7 Acknowledgments

This work is partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2024-00351898), the MOTIE under Training Industrial Security Specialist for High-Tech Industry (RS-2024-00415520) supervised by the Korea Institute for Advancement of Technology (KIAT), and the MSIT under the ICAN (ICT Challenge and Advanced Network of HRD) program (No. IITP-2022-RS-2022-00156310) supervised by the Institute of Information & Communication Technology

References

[1] Amazon Web Services, Inc. Microservices on aws. Technical report, Amazon Web Services, 2017. Whitepaper. PDF: https://docs.aws.amazon.com/pdfs/whitepapers/latest/microservices-on-aws/microservices-on-aws.pdf.

[2] Ramaswamy Chandramouli, Zack Butcher, and James Callaghan. Service mesh proxy models for cloud-native applications (nist sp 800-233). Technical report, National Institute of Standards and Technology, 2024. Accessed: 2025-11-06.

[3] GeeksforGeeks Contributors. System design netflix: A complete architecture. https://www.geeksforgeeks.org/system-design-netflix-a-complete-architecture/, 2025. Accessed: 2025-11-06.

[4] John Howard, Ethan J. Jackson, Yuval Kohavi, Idit Levine, Justin Pettit, and Lin Sun. Introducing ambient mesh. https://istio.io/latest/blog/2022/introducing-ambient-mesh/, 2022. Accessed: 2025-11-06.

[5] Istio Authors. Circuit breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/, 2025. Accessed: 2025-11-06.

[6] Istio Authors. Destination rule – connection pool settings. https://istio.io/latest/docs/reference/config/networking/destination-rule/, 2025. Accessed: 2025-11-06.

[7] Istio Authors. Istio ambient mesh architecture overview. https://istio.io/latest/docs/ambient/architecture/, 2025. Accessed: 2025-11-06.

[8] Istio Authors. Istio architecture (sidecar-based deployment). https://istio.io/latest/docs/ops/deployment/architecture/, 2025. Accessed: 2025-11-06.

[9] Istio Authors. Sidecar or ambient? https://istio.io/latest/docs/overview/dataplane-modes/, 2025. Accessed: 2025-11-06.

[10] Ruslan Meshenberg. Microservices at netflix scale: First principles, tradeoffs, lessons learned. In *GOTO Amsterdam 2016*, 2016.

[11] Muhammad Miraj and Ahmad Nurul Fajar. Model-based resilience pattern analysis for fault tolerance in reactive microservice. *Journal of Theoretical and Applied Information Technology*, 100(9):3075–3093, 2022. Accessed: 2025-11-06.

[12] Red Hat, Inc. Migrating from service mesh 2 to service mesh 3, 2024. Red Hat OpenShift Service Mesh 3.0 Documentation, accessed: 2025-11-07.

[13] Mohammad Reza Saleh Sedghpour, David Garlan, Bradley Schmerl, Cristian Klein, and Johan Tordsson. Breaking the vicious circle: Self-adaptive microservice circuit breaking and retry. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*, pages 32–42, 2023.

[14] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. Service mesh circuit breaker: From panic button to performance management tool. 04 2021.

[15] Falahah Suprapto, Kridanto Surendro, and Wikan Sunindyo. Circuit breaker in microservices: State of the art and future prospects. *IOP Conference Series: Materials Science and Engineering*, 1077:012065, 02 2021.

[16] Will Zhang. Improving microservice reliability with istio, 2020.

[17] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, XiongChun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting overheads of service mesh sidecars. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 142–157, New York, NY, USA, 2023. Association for Computing Machinery.