It's Time to Retry: Taming Retry Storms by Treating Retry Capacity as a Shared Resource

Yazhuo Zhang Giuseppe Steduto Luca Renna Ana Klimovic
ETH Zurich

Abstract

Retries are essential for masking transient failures in distributed systems, but when sustained or systemic failures occur, retries can create positive feedback loops that amplify load and prolong outages. Mechanisms such as backoff, circuit breakers, or retry budgets regulate each client independently. In practice, this is not enough as service providers need a way to control aggregate retry load across clients.

Our key insight is that retry capacity should be treated as a shared resource and enforced where aggregate demand is visible: at the proxy between clients and the downstream service. We propose Arolla, a retry admission mechanism that observes the aggregate retry stream and admits retries with a token bucket that combines time-based and event-based refill. This enables automatically reducing retry admission during severe failures and increasing it as the service recovers. To prevent aggressive clients from monopolizing scarce retry capacity, Arolla uses an attempt-aware cost function that makes large retry attempts progressively more expensive, without requiring per-tenant state. Arolla requires no application code changes as it does not replace client-side retry logic: clients continue to decide *when* to retry and *how* to back off, Arolla simply decides *which* retry to admit.

We implement Arolla as an Envoy filter for Istio and show that across a wide range of failure scenarios in a realistic microservice testbed, Arolla prevents retry-induced collapse and enables fast recovery, while maintaining fairness and low overhead. We further reproduce a real-world retry-storm outage from its public postmortem and show that Arolla automatically recovers the system without operator intervention.

1 Introduction

Modern distributed services routinely experience transient failures, such as momentary network disruptions, latency spikes, or brief pod restarts, which cause requests to time out even when components are healthy on average [9, 19, 29, 41, 44]. To cope with such disruptions, large-scale systems widely rely on automatic retries in client libraries and RPC frameworks [10, 12, 26, 39, 50]. When a request fails due to a transient error or timeout, the client issues additional attempts in the hope that a subsequent try will succeed. This mechanism is highly effective at masking transient failures.

However, retries work well only when the failure is transient. When failures are systemic, such as capacity shortfalls, dependency outages, or correlated slowdowns, retries become destructive. Each failed request triggers additional retries, increasing load and causing more failures, forming a positive feedback loop that displaces first attempts and drives success rates below those of the original failure alone. If the retry-induced load exceeds the service's remaining capacity, the system enters a metastable failure [43]: an overloaded state that persists even after the original trigger is gone. Retry storms are a recurring cause of prolonged outages in large-scale systems [40, 43, 48, 69].

Client-side retry controls, such as exponential backoff, circuit breakers, and retry budgets, are widely deployed to prevent retry storms. However, they are not sufficient in practice. We analyze 21 retry-related production incidents from public postmortems (2011–2025) and identify four recurring reasons: (1) clients cannot distinguish systemic from transient failure; (2) aggregate retry load is not controlled, because each client library, SDK, and application team independently chooses when to retry, how many times, and how aggressively to back off; (3) aggressive clients crowd out well-behaved ones; and (4) client-side controls are fragile and fail to scale.

Industry has begun shifting retry control to the service-mesh proxy layer (Envoy, Linkerd), where infrastructure proxies can observe aggregate retry traffic. However, existing proxy-layer controls are not coupled to downstream health. Envoy's retry budget [26] caps concurrent retries as a fixed fraction of active requests, which is too permissive during a severe outage and too restrictive during minor faults. Circuit breakers (e.g., Envoy's outlier detection [25]) rely on operator-configured thresholds that are difficult to tune and often unstable across workloads.

We present Arolla, a retry admission mechanism that addresses this gap. Arolla treats retry capacity as a shared resource at each downstream service and bounds it using a

mixed-refill token bucket. The refill rate is coupled to downstream success: the budget tightens as the service degrades and expands as it recovers. For fairness across clients, Arolla provides an attempt-aware cost function that makes large retry attempts progressively more expensive, such that aggressive clients do not crowd out well-behaved ones.

Arolla runs as an Envoy filter deployed on sidecars and ingress gateways, requiring no changes to application code or client libraries. We evaluate Arolla in a realistic microservice testbed under a wide range of failure scenarios. In contrast to uncontrolled retries, circuit breakers, and Envoy retry budget, Arolla consistently prevents sustained outages and enables rapid recovery once faults are resolved. Recovery time scales from seconds under mild faults to tens of seconds under severe conditions. Arolla is robust across parameter choices and fairly allocates retry capacity across heterogeneous clients. We also reproduce a real-world retry-storm outage from its public postmortem and show that Arolla recovers the system automatically, without operator intervention. We will open-source Arolla prototype and our postmortem experiment suite.

Summary of contributions:

- We analyze 21 retry-related production incidents and characterize why client-side retry controls and existing proxy-layer controls fail to prevent retry storms in open systems.
- We propose Arolla, a retry admission mechanism that bounds aggregate retry load using a mixed token bucket and enforces fairness via an attempt-aware cost function.
- We implement Arolla as an Envoy HTTP filter with native Istio integration, requiring zero modifications to application code or client libraries.
- We evaluate Arolla on a live microservice testbed under severe failure scenarios. Arolla robustly prevents retry-induced collapse and outperforms existing proxy controls with negligible overhead. We also reproduce a real retry-storm outage from its public postmortem and show that Arolla recovers the system automatically.

2 Retries in Distributed Systems

Modern Internet services are increasingly built as distributed microservice applications: user-facing platforms such as Netflix, Uber, Amazon, and Slack operate hundreds to thousands of independent services that communicate over the network [14, 30, 65, 71]. A single user action routinely fans out into dozens of inter-service requests along a call graph, each a potential point of transient failure. In such systems, when a client sends a request to a downstream service and it fails, due to a timeout, an error response, or a transient network disruption, the client may retry: it sends an additional attempt for the same logical operation.

2.1 When Retries Work and When They Fail

A retry is beneficial when it resolves a *transient* failure: a momentary blip affecting a single request or a brief window of requests, where a subsequent attempt is likely to succeed because the underlying cause has already cleared. We define retry efficiency as the fraction of admitted retries that succeed. Under transient failure, retry efficiency is high.

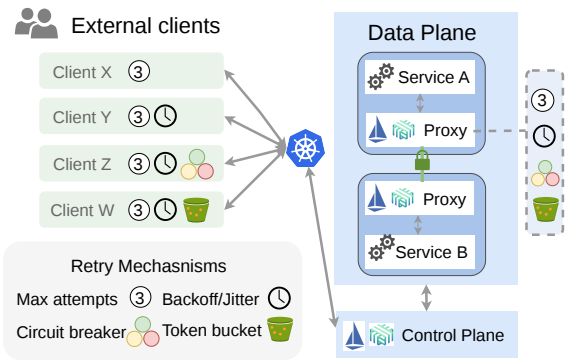


Figure 1: Retry control points in a service-mesh deployment. External clients apply one or more of four local mechanisms: maximum attempts, backoff with jitter, circuit breaker, and token bucket. Service-mesh proxies (sidecar and ingress) can apply similar controls.

Retries become harmful when the failure is *systemic*: a sustained condition, such as a capacity shortfall, a dependency outage, or a correlated slowdown, which affects many clients simultaneously and persists across retry windows. In this case, retry efficiency approaches zero. To make matters worse, the additional retry traffic displaces first attempts, reducing the overall success rate below what the original failure rate alone would predict. This creates a destructive positive feedback loop: failures produce retries, which produce more load, which causes more failures. If the retry-induced load exceeds the service's remaining capacity, the system enters a sustained metastable failure state in which retries become the dominant source of traffic [43]. §2.3 analyzes 21 such incidents from public postmortems across cloud providers, SaaS platforms, and infrastructure services.

2.2 Client-Side Retry Control

To limit retry volume, client libraries provide several mechanisms (Fig. 1). These mechanisms are often implemented in the client logic of RPC frameworks like gRPC [39], cloud service API frameworks like the AWS S3 SDK [59], and in general-purpose resilience libraries such as Resilience4j [56].

Maximum attempt limits. The most basic control is a static cap on the number of times a single request can be retried (typically 1–3 times) [10, 39]. While this bounds the total work generated by an individual request, it provides no defense against concurrent storms: if N failing clients each retry 3 times, the downstream is still hit by $3N$ uncoordinated retries.

Backoff and jitter. Backoff is a scheduling policy that spaces out subsequent retry attempts, typically using exponentially increasing delays with added randomness (jitter) to prevent synchronized thundering herds [12]. While backoff reduces

the instantaneous retry rate from a single client, it does not cap the aggregate volume: N backing-off clients still eventually produce $O(N)$ retries.

Circuit breaker. A circuit breaker is a state-machine mechanism that tracks request outcomes over a sliding window, tripping "open" to block all outbound traffic if the failure or slow-call rate exceeds a threshold, and transitioning to "half-open" after a cooldown to tentatively test recovery [52, 56]. Its key limitation is that it is binary (passing or blocking all traffic) and relies on static thresholds that are highly sensitive to the specific workload and failure mode [57].

Token bucket. The token bucket, a primitive adapted from network rate-limiting, is a local mechanism that throttles retries to protect the downstream during outages [59]. The client maintains a bounded bucket of tokens. Each successful request deposits a fractional token (e.g., 0.1 tokens), while issuing a retry consumes a full token (e.g., 1.0 token) up to a maximum attempt limit. If the bucket lacks sufficient tokens, the client drops the retry rather than sending it to the network.

Server-published throttling. Server-published throttling is a hybrid approach in which the downstream server publishes token-bucket parameters (e.g., via gRPC service config) that clients apply locally [39]. This gives operators some influence over retry behavior, but enforcement remains client-side: each client runs its own bucket independently.

All client-side mechanisms share a fundamental limitation: they bound individual retry rates but are inherently blind to the aggregate downstream load. Next, we analyze production incidents to demonstrate exactly how this lack of global coordination causes these mechanisms to fail under stress.

2.3 Characterizing Retry-Related Production Incidents

To understand why retry storms persist despite widespread adoption of modern resilience patterns, we collected and analyzed 21 retry-related production incidents from public postmortems (2011–2025). The dataset covers major cloud providers (AWS [2–4, 6, 7], GCP [32–36], Cloudflare [54]), widely used SaaS platforms (Slack [53], Discord [22], CircleCI [17]), and popular infrastructure or platform services (Spotify [20], Allegro [48], Honeycomb [42], Elastic Cloud [24], Datadog [21], Coinbase [18], Signal [60]).

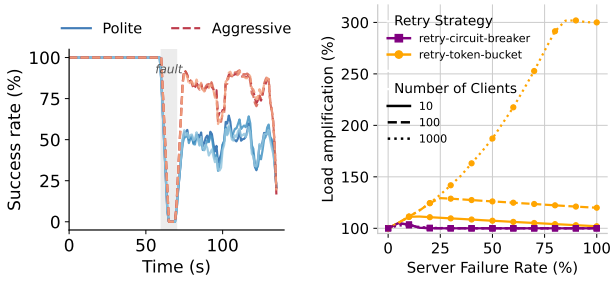
While prior work has discussed retries as a primary sustaining mechanism in metastable failures [43], our analysis of these incidents highlights why current mitigation strategies fail in practice. We draw four observations.

Observation 1: Clients blindly retry on systemic failures. Client-side retries are fundamentally designed to silently absorb transient anomalies, such as dropped network packets or brief latency spikes. However, in all 21 incidents we analyzed, the failures were systemic. Our analysis reveals that these systemic outages always involved two components: a latent root cause and an activating trigger. The root causes were deep architectural vulnerabilities, such as underprovisioned resources [20], software bugs and misconfigurations [17,32,54], and single points of failure (SPOFs) [4,33]. These latent flaws lay dormant until activated by a routine, such as a physical network interruption [6], a high load surge [48], or a software release [21]. Crucially, a client receiving a timeout or error response cannot distinguish between a momentary network blip and a severe systemic failure. Lacking this global context, clients blindly apply transient error handling to systemic faults. Instead of allowing the degraded system time to recover, the automated wave of retries amplifies the underlying fault into a catastrophic capacity shortfall.

Observation 2: Uncoordinated clients produce unbounded retry surges. When a shared dependency degrades, the resulting load is the sum of many independent retry decisions. In the Discord [22], Signal [60], and Coinbase [18] incidents, millions of external applications simultaneously attempted to reconnect after sessions were severed by backend congestion or database failovers. The same pattern held internally: during the Elastic Cloud outage [24], independent proxy instances issued concurrent retries against a degraded ZooKeeper server, and the combined request rate exceeded what the server could absorb. In both settings, each client's retry decision was locally reasonable, but the aggregate was unbounded.

Observation 3: Heterogeneous clients exhibit disparate retry behaviors. In several incidents, a small subset of aggressive clients dominated recovery capacity at the expense of well-behaved ones. During the Spotify outage [20], legacy desktop clients lacked any backoff logic and issued a constant stream of requests that saturated all available service capacity; modern clients correctly executing exponential backoff found no open slots when they attempted to reconnect. "Polite" backoff was effectively punished, the more a client yielded, the less likely it was to succeed. A similar pattern emerged during the Google App Engine incident [32], where a tight retry loop in just 1.29% of applications exhausted the global authentication capacity and crowded out the remaining 98% of healthy traffic. In both cases, a few aggressive clients determined whether the whole system recovered.

Observation 4: Client-side controls failed, while postmortems recommend the same class of fixes. In several of the incidents we examined, client-side retry controls were already in place but did not contain the failure. Allegro's static circuit breaker did not trip because the flash-sale traffic diluted the error rate below its threshold [48], while Slack's exponential backoff was not aggressive enough to protect its degraded database [53]. The lessons-learned sections of these postmortems commonly recommend adopting or tightening the same class of client-side mechanisms: installing a circuit breaker, enabling exponential backoff with jitter, adding a retry budget, or tuning existing parameters more conservatively. The next section examines why these approaches are not sufficient.



(a) Aggressive retry behavior monopolizes recovery capacity **(b)** Client-side controls fail to scale.

Figure 2: Why client-side retry controls fail.

3 Limitations of Existing Mechanisms

We discuss the shortcomings of existing retry control mechanisms on both the client side (§3.1) and the server side (§3.2).

3.1 Why Client-Side Retry Control Fails

Based on the characterization study (§2.3), we identify four structural reasons why client-side retry controls fall short.

(a) Clients cannot distinguish systemic from transient failure. A client that receives a timeout or a 503 error has no way to determine whether the failure is a one-off blip that a retry will likely resolve or a systemic overload affecting all clients simultaneously. All client-side retry policies, regardless of backoff strategy or gating logic, make the retry decision from locally observable information alone.

(b) Aggregate retry load is not controlled. Retry policy is not set in one place. Each client library, SDK, and application team independently chooses when to retry, how many times to retry, and how aggressively to back off, with no visibility into the decisions of its peers. The result is that no operator-controllable knob caps the aggregate load reaching a shared dependency. Table 1 illustrates the extent of this fragmentation across eight widely used client libraries: maximum attempts range from 1 to unbounded; initial backoff spans two orders of magnitude (0 ms to 2 s); retryable status codes vary from "network errors only" to 'all exceptions." When services using different SDKs share a downstream dependency, standardizing retry behavior globally is infeasible: the client population is distributed, heterogeneous, and frequently composed of third-party or legacy software whose retry behavior is entirely opaque to the operator.

(c) Aggressive retries monopolize recovery capacity. When server capacity is degraded, every admitted retry displaces capacity that could have served a first attempt or another tenant's retry. Retry capacity is, in effect, a shared resource, yet no allocation mechanism governs its use. Clients that retry more aggressively (shorter backoff, more attempts, more instances) consume a disproportionate share of the remaining capacity, while well-behaved clients see their success rates decline further. To illustrate, we ran an experiment with six clients sharing a single service instance: three polite clients configured with up to 3 retries and three aggressive clients configured with up to 10. We injected a 100% failure at t=60 s and cleared the fault shortly after. As shown in Fig. 2a, the aggressive clients monopolize the recovery capacity, achieving a much higher success rate than the polite clients, whose success rates plummet as they yield to the aggressive ones.

Table 1: Default retry parameters in popular client libraries. "—" = retries disabled by default; † = deadline-based, not attempt-based. ‡ = unlimited retries if the RPC never left the client, at most 1 retry if it reached the library but not the server.

Library / SDK	Lang.	Max retries	Initial backoff	Retryable conditions
gRPC (built-in) [39]	Multi	—	—	Transparent retry‡
google-api-core [31]	Py	120 s†	1 s	500, 503, 429, conn. err
aws-sdk-go-v2 [5, 8]	Go	3	rand[0, 2) s	500–504, throttle
axios-retry [61]	JS	3	0	net err + 5xx (idemp.)
Envoy [27]	C++	1	25 ms	none
Resilience4j [55]	Java	3	500 ms	all exceptions
urllib3/Retry [64]	Py	10	0	conn/read err (idemp.)
Tenacity [63]	Py	∞	0	all exceptions

(d) Client-side controls are fragile and fail to scale. Operators already deploy retry controls, such as exponential backoff with jitter [12], circuit breakers [56], and token budgets [26, 59], but these are highly fragile when static parameters do not match the specific failure mode. More importantly, even if each client adopts best-known retry logic, these mechanisms become unreliable as the client population grows. Brooker [13] shows that when many short-lived clients (as in serverless and container deployments) share a dependency, each client observes too few requests to estimate the true failure rate. As a result, circuit breakers trip early or oscillate between open and closed states, and token buckets drain too slowly to prevent the aggregate retry load from scaling as $O(N)$ in the number of clients, regardless of how well any individual client is tuned. We implemented a retry-behavior simulator that reproduces these findings and extends them to the full range of server failure rates (0%–100%), shown in Fig. 2b.

Summary. Client-side retry control is local, fragile, and fundamentally uncontrollable in open systems. Clients cannot distinguish systemic from transient failure, cannot control aggregate behavior across heterogeneous SDKs, cannot fairly allocate retry capacity among competing tenants, and existing controls are routinely misconfigured. Yet in every incident we studied, operators eventually restored service by throttling load at a server-side control point, manually, under pressure, during an outage. This suggests that retry control should be enforced at a shared on-path enforcement point where aggregate demand is visible. Next, we discuss the existing proxy-layer controls and their limitations.

3.2 Proxy-Layer Retry Control

As modern microservice applications grow to hundreds or thousands of services, operators increasingly move network-layer functionality such as routing, encryption, observability, and retries into a service mesh. In systems such as Istio, every inter-service request traverses an infrastructure proxy (e.g., Envoy), providing an enforcement point independent of application code [47]. At the downstream service boundary, the proxy can observe requests from multiple callers, distinguish retries from first attempts using request metadata, and apply a common admission policy.

This placement is increasingly reflected in production systems. For example, the Kubernetes Gateway API community recently identified budgeted retries as a frequently requested traffic-management feature [49]. Existing proxy-layer mechanisms, however, still regulate retries using local proxy state. Envoy's retry budget limits concurrent retries to a configurable fraction of active and pending upstream requests [26]. Linkerd similarly limits retry volume using a `retryRatio` and a `minRetriesPerSecond` floor [50].

These mechanisms do not directly couple retry admission to downstream recovery. For example, with a 20% Envoy retry budget, a caller proxy with 100 active and pending requests may permit 20 concurrent retries even if none of those requests is succeeding. Multiple caller proxies can independently contribute additional retries to the same degraded dependency. Moreover, an application-issued retry arrives at Envoy as a new HTTP request, so its initial upstream attempt bypasses the router's retry-budget check. Arolla instead gates identifiable retries at the protected service's inbound proxy and adjusts retry capacity based on successful downstream completions. §4 describes the admission policy in detail.

4 Arolla Design

We propose Arolla, a retry admission mechanism that fills the enforcement gap identified in §3. Arolla runs at the proxy on the path between callers and the downstream service, where it observes the aggregate retry stream and applies a budget to bound the number of retries. Arolla does not replace client-side retry logic: clients continue to decide *when* to retry and *how* to back off, which means there is no need for client application or library code changes. Arolla decides *whether* each retry is admitted.

4.1 Retry Capacity as a Shared Resource

We frame retry admission as resource allocation. Each retry consumes downstream processing capacity that could serve a first attempt. A service's ability to absorb retries without degradation is finite and is a shared resource across all callers, analogous to CPU or network bandwidth in a cluster scheduler [28]. We bound this resource with a token bucket that consumes (drains) one token per admitted retry.

There are two classical refill disciplines for a token bucket, each parameterized by a refill rate and a bucket capacity:

- **Time-refill bucket** (t, C)**.** The bucket receives t tokens per second regardless of traffic, capped at C. This guarantees a minimum admission rate: it preserves retry capacity during low-traffic periods and lets clients probe a server for recovery after failure.

- **Event-refill bucket** (r, C)**.** Each successful response deposits r tokens, capped at C. This provides feedback: as the downstream service degrades, successes decline, deposits decrease, and the budget tightens automatically.

Arolla combines time and event-based refill disciplines, inspired by Isaacs [45], which show that neither discipline alone suffices for retry control and the two can be combined. In contrast to prior work, Arolla applies the mixed discipline token-bucket at a new control point: it places a bucket on the retry admission path that is *shared across all callers* of the protected downstream service. With this design, event refill causes the bucket's available tokens to grow more slowly as the service degrades, while time refill sustains a small rate of token replenishment that allows clients to probe for recovery after a failure.

Composition across call chains. In a multi-hop call chain, each service boundary has its own retry-admission bucket. These buckets compose naturally. When a downstream service degrades, the upstream service sees fewer successful responses, so its event-based refill slows and the number of available tokens in its bucket grows more slowly as well. No explicit cross-service coordination is required: each proxy updates its bucket using only the responses it already observes from its downstream service.

4.2 Multi-tenant Retry Allocation

Arolla also needs a mechanism to ensure fairness among callers that send requests to a common downstream service, such that callers that retry more aggressively than others do not necessarily consume capacity before others (§3.1c). Instead, Arolla seeks to allocate retry capacity based on need.

A natural approach is to track each tenant explicitly and enforce per-tenant quotas (e.g., max-min fair shares). In principle, this can provide strong isolation. In practice, however, it requires a tenant identifier on every request, per-tenant state at every proxy, and some mechanism to reset or age quota state over time. This complexity is difficult to justify when the retry path is a fraction of total traffic.

Instead, Arolla uses the retry *attempt number* itself as a lightweight signal of aggressiveness. Early retries are relatively cheap: they often reflect transient failures and still have a reasonable chance of succeeding. Later retries are different. A higher number of attempt is a stronger signal that the request is stuck in a persistent failure mode and is more likely to waste scarce downstream capacity. Arolla exploits this distinction by charging later retry attempts more heavily. In this way, fairness emerges from the admission cost function itself, without requiring per-tenant state.

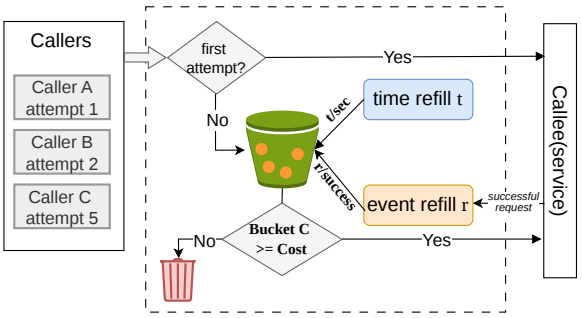


Figure 3: Arolla's admission path

Cost function. Given a configurable *threshold* τ (default 3), the cost of admitting a retry at attempt a is:

$$cost(a) = \begin{cases} 1 & \text{if } a \le \tau, \\ a(a-1)/2 & \text{if } a > \tau. \end{cases} \quad (1)$$

The first few retries are intentionally cheap, while later retries become expensive quickly. With the default τ: attempt 4 costs 6 tokens, attempt 5 costs 10, attempt 6 costs 15. An aggressive caller at attempt 5 drains 50% of a C=20 bucket in a single admission; a polite caller at attempt 2 pays one token and drains 5%. This pricing makes aggressive retries self-limiting under contention while preserving access for more conservative retries.

Fig. 3 visualizes Arolla's admission path and Algorithm 1 gives the full admission logic. The filter holds one bucket per protected service: a token count B initialized to C and a timestamp *last* tracking the previous refill. First attempts are admitted unconditionally (line 8); For each retry, the filter first refills the bucket by $t \cdot \Delta$ tokens for the elapsed interval (capped at C), then looks up the attempt-aware cost c from Equation 1. If $B \ge c$, the filter subtracts c tokens from B and admits the retry; otherwise the retry is rejected. On each successful response, the bucket deposits r tokens (line 9-12).

4.3 How Arolla Addresses the Limitations of Client-Side Control

We now revisit the four problems of client-side retry control identified in §3.1 and explain how Arolla addresses each.

Aggregate health signal. Clients cannot distinguish systemic from transient failure. Arolla instead uses aggregate downstream outcomes at the proxy. When successes continue, event-based refill keeps the retry-admission bucket replenished; when successes collapse, refill slows automatically and retry admission tightens. This lets Arolla react to downstream health without requiring clients to classify failures explicitly.

Aggregate enforcement. Client-side controls bound retries per caller, but they do not bound the total retry load arriving at a service. Arolla addresses this by placing a single shared retry-admission bucket at each service boundary. All callers draw from the same bucket, regardless of how many clients exist or how their retry policies differ, so the protected service sees a direct bound on aggregate admitted retry traffic.

Algorithm 1 Mixed-refill retry admission with attempt-aware pricing.

```
1:  Params: r (event refill), C (capacity), t (time refill), τ (cheap threshold)
2:  State: B ← C;   last ← now
3:  function TIMEREFILL
4:      now ← current time
5:      B ← min(B + t · (now − last), C)
6:      last ← now
7:  function ONREQUEST(req)
8:      if req.attempt = 1 then return ADMIT                ▷ first attempts are never gated
9:      TIMEREFILL
10:     c ← COST(req.attempt)
11:     if B < c then return REJECT
12:     B ← B − c;   return ADMIT
13: function ONRESPONSE(resp)
14:     if resp is success then
15:         TIMEREFILL
16:         B ← min(B + r, C)                               ▷ event refill
17: function COST(a)
18:     if a ≤ τ then return 1
19:     else return a(a−1)/2
```

Fair sharing under contention. When retry capacity becomes scarce, aggressive callers can otherwise consume a disproportionate share simply by retrying more often. Arolla's attempt-aware cost function makes progressively later retries increasingly expensive, so aggressive retry behavior becomes self-limiting under contention. This preserves admission for less aggressive retries without requiring per-tenant state or explicit fair-share accounting.

Robustness and scalability. Client-side retry controls are fragile because enforcement is replicated across many callers, each operating on partial and noisy observations. Arolla centralizes enforcement at the service boundary, where the proxy observes the aggregate retry stream directly. This avoids per-client sample starvation and makes control naturally scale with the deployment: each proxy protects its own downstream service instance, and mixed refill provides a broader safe operating range across workloads and failure conditions.

5 Implementation

We implement Arolla as an Envoy Wasm filter deployed in the Istio service mesh [47]. The filter requires no modification to application code or Istio's control plane.

Retry identification. The filter distinguishes retries from first attempts using a per-request attempt counter derived from two HTTP headers. The `x-envoy-attempt-count` header is set by upstream Envoy sidecar for mesh-internal retries. For external traffic, we read a client-provided attempt header (e.g., `X-Attempt-Number`). Requests with counter $= 1$ are first attempts (always admitted); requests with counter > 1 are retries (subject to admission).

State and overhead. The filter maintains a single aggregate bucket per service: a token count, a last-refill timestamp, and the parameter set, totaling tens of bytes. Per-request processing is $O(1)$: two header reads, a time-based refill update, one comparison, and, on admission, a token decrement; successful responses add one token increment. The bucket is shared across HTTP contexts within an Envoy worker and requires no locking beyond the worker's single-threaded event loop. We quantify CPU and memory overhead in §6.5.

Configuration. Arolla is configured through Istio's `pluginConfig` and `WasmPlugin`, with default parameters r (0.1), C (20), t (1.0), and, for the fairness variant, $\tau(3)$.

Deployment. Istio's sidecar injection provides per-hop enforcement. Arolla is injected at phase `AUTHN`, which is before Istio authentication filters, so retries are gated before any routing. For services outside the mesh, the filter runs on an ingress gateway, providing retry admission at the boundary. Arolla scales with the protected service: each replica maintains its own bucket, so total retry capacity grows with service capacity. Because requests are load-balanced across replicas, per-replica buckets approximate a global retry budget while preserving scalability and avoiding cross-proxy coordination.

Compatibility Rejected retries are failed immediately with a retryable response (e.g., HTTP 429), allowing clients to continue using their existing retry policies. Because Arolla operates entirely at the proxy layer, it requires no application changes or cross-service coordination.

6 Evaluation

We answer the following questions:

- Can Arolla prevent retry-induced collapse under realistic failure scenarios? (§6.2)
- Does Arolla enforce fair allocation across heterogeneous clients? (§6.3)
- Can Arolla mitigate real-world retry-storm outages? (§6.4)
- What is Arolla's runtime overhead? (§6.5)
- How sensitive is Arolla to its parameter choices compared to existing approaches? (§6.6)

6.1 Testbed and Methodology

Application. We evaluate on Online Boutique [37], an 11-service e-commerce application developed by Google. Services are written in five languages (Go, Python, C#, Java, Node.js) and communicate over gRPC. We enable retries at every service's sidecar using Istio `VirtualService` policies: up to 3 retries per hop on 5xx and transport-level errors, with jittered exponential backoff and per-try timeouts calibrated to each service's measured healthy p99 latency.

Deployment. We deploy Online Boutique on a Kubernetes cluster of four worker nodes (32 cores and 64 GB of RAM each), running Istio 1.27. Each application service runs with 2 replicas and up to 4 CPU cores per replica; the ingress gateway runs with 2 replicas. Arolla is installed on all sidecars and the ingress gateway via Istio's `WasmPlugin` resource. We restart all pods between runs to eliminate residual state (e.g., connection pools, Wasm VM state, retry queues) that could otherwise bias results.

Load generation. We use an open-loop load generator that issues requests at a fixed rate regardless of backend latency. The generator runs on a dedicated 32-core client node. External clients retry failed requests up to 3 times on 5xx and 429 responses by default.

Failure injection. We inject failures using Istio `VirtualService` fault injection, returning HTTP 503 responses at the `cartservice` sidecar for a configurable fraction of requests during the fault window.

Policies. We compare Arolla against three baselines:

- No control: All retries are admitted. This represents uncontrolled retry behavior.
- Circuit breaker: Envoy's per-service outlier detection with ejection threshold $\theta = 50\%$ error rate and base ejection time 5 s.
- Envoy retry budget: Envoy's default configuration, limiting concurrent retries to 20% of active upstream requests with a minimum of 3.

Metrics. We report the following metrics:

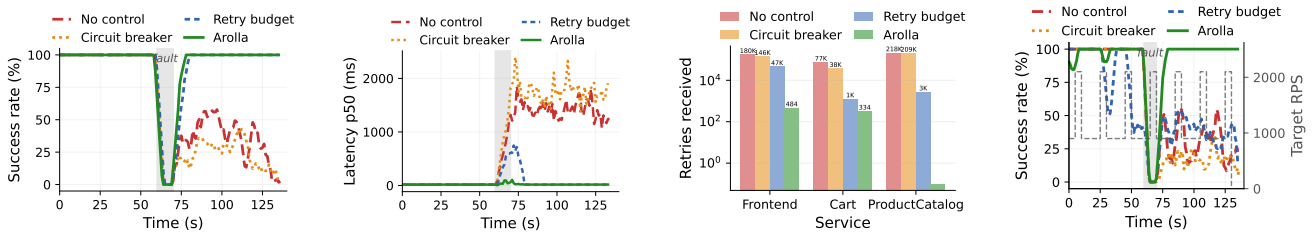
- Success Rate: successful RPS at the frontend.
- Recovery time: seconds from fault removal until the client-observed success rate reaches $\geq 95\%$.
- Latency: p50/p99 of E2E request latency at the frontend.

6.2 Effectiveness

When a service experiences partial failure, retries can amplify load and prevent recovery even after the fault is removed. We evaluate whether Arolla breaks this feedback loop and restores availability. We first study one representative scenario in detail, then sweep three axes of fault severity: offered load, failure rate, and fault duration.

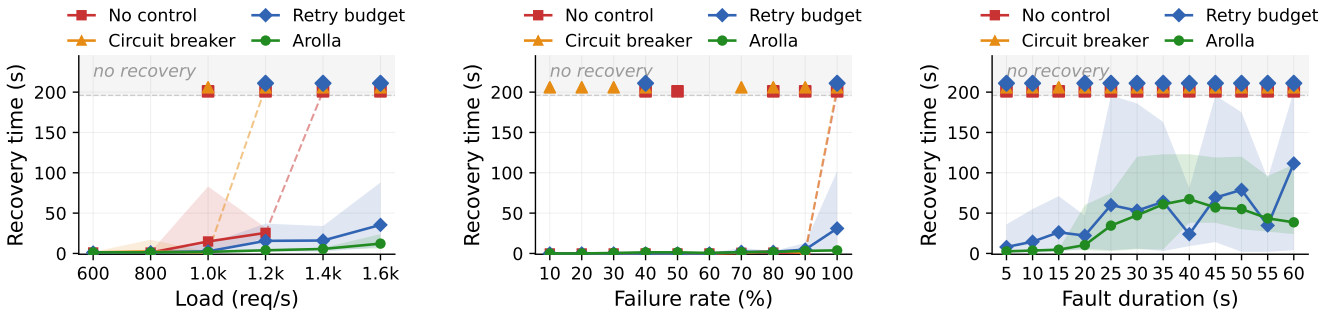
6.2.1 Sustained failure

Setup. We target the `POST /cart` endpoint, which fans out to `productcatalogservice` (to validate the SKU) and `cartservice` (to add the item to the session cart backed by Redis). Both hops inherit the mesh-wide retry policy (§6.1), and the external client retries up to 3 on failure. We drive 1,200 RPS of open-loop traffic to the frontend, which is 70%



(a) Success rate under steady load. **(b)** Frontend p50 latency. **(c)** Total retries to each service. **(d)** Success rate under bursty load.

Figure 4: Effectiveness under steady and bursty load. We inject a 100% abort fault at `cartservice` for 10 s. Arolla recovers under both workloads, whereas Envoy's retry budget recovers under steady load but does not sustain full recovery under bursty load within the observation window.



(a) Recovery time vs. load. **(b)** Recovery time vs. failure rate. **(c)** Recovery time vs. failure duration.

Figure 5: Recovery time across three axes of failure severity. Shaded bands span the min-max range of 10 repeated runs per point. Markers in the top "no recovery" region indicate runs that never recovered.

of the cluster's healthy-state capacity. To induce metastability: we inject a 100% abort fault at `cartservice` for 10 s and use a tight `perTryTimeout` to trigger retries quickly. Arolla runs on every sidecar and on the ingress gateway with default parameters r=0.1, t=1.0, and C=20. Circuit breaker and Envoy retry budget also use their default settings.

Results. All four policies sustain 1,200 RPS at 100% success before the fault. During the 10 s fault window,success drops to 0% for all policies because every `cartservice` call is aborted. The policies diverge sharply after fault removal.

No recovery: With no control and circuit breaker, the system does not recover within the measurement window (Fig. 4a). Success rate briefly rebounds after fault removal, then remains low and eventually decays toward zero. Their frontend p50 latency stays high throughout recovery, roughly 1–2 s, versus a healthy baseline near 15 ms (Fig. 4b), indicating persistent backlog rather than transient disruption. Fig. 4c shows that both policies allow large retry volumes to continue circulating through the request path. At the frontend, no control and circuit breaker deliver about 181K and 146K retries, respectively; at `productcatalogservice`, 218K and 209K mesh retries; and at `cartservice`, 77K and 39K retries.

Recovery under steady load: Retry budget and Arolla both recover in the run shown in Fig. 4a. However, retry budget recovers more slowly and exhibits a pronounced latency spike, while Arolla returns to steady-state behavior more quickly (Fig. 4b). Arolla also admits substantially fewer retries across the three services (Fig. 4c).

Recovery under bursty load: Fig. 4d shows that bursty arrivals challenge stability even before fault injection. Envoy's retry budget exhibits pronounced success-rate fluctuations as bursts trigger retries, while Arolla maintains near-100% success apart from brief initial dips. The injected fault then drives all policies to approximately zero success. After fault removal, Arolla quickly returns to 100% success and sustains it through subsequent bursts. In contrast, retry budget, no control, and circuit breaker remain at low, fluctuating success rates throughout the remaining observation window. These results show that Arolla both limits burst-induced instability and enables recovery after the injected fault.

Why non-control and circuit-breaker fail to recover. The non-recovery cases share the same root cause: after the fault is removed, retries continue to consume the limited capacity needed for recovery.

Under no-control case, retries are unconstrained. A failed `POST /cart` can therefore expand into a multi-level retry tree across the client, frontend, and downstream services. Once the cart path becomes overloaded, these retries compete with fresh requests for service capacity, so the system remains trapped in a low-goodput state.

With circuit breaker, retries are not eliminated; instead, recovery becomes oscillatory. When the failure rate crosses the ejection threshold, the overloaded pod is ejected and temporarily receives no traffic. When the ejection interval expires, queued demand and retries return at once, immediately pushing the service back into failure. This repeated on/off behavior prevents the system from settling into sustained recovery.

Why retry budget and Arolla help. Both retry budget and Arolla improve over no control and circuit breaker because they suppress retry amplification after the fault is removed, allowing useful work to make progress again.

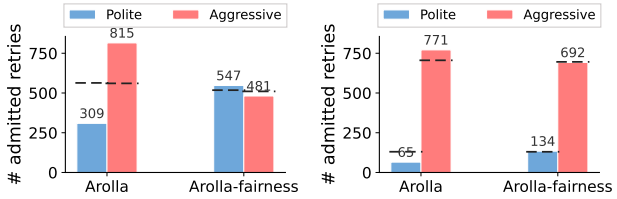
With retry budget, retry admission is tied to the number of in-flight requests. In the run shown in Fig. 4, this is sufficient to reduce retry pressure enough for the system to recover. However, this coupling also makes the mechanism sensitive near the recovery tipping point: if the post-fault burst drains quickly enough, the system recovers; if not, elevated in-flight demand permits more retries, which prolongs overload and can re-trigger the feedback loop. We will discuss this sensitivity in the broader fault-space below.

With Arolla, retry admission is driven by successful downstream responses rather than in-flight load. During the fault, successful responses stop, so event refill halts and the retry bucket drains. Time refill contributes at most t=1 retry/s, which allows only a small probe stream while rejecting the vast majority of retries. As a result, the cart path does not accumulate a large retry backlog, and overload does not spill over to `productcatalogservice`. Once the fault is removed, fresh requests can complete immediately, successful responses resume event refill, and retry capacity expands only as the downstream service proves it can handle load.

6.2.2 Generality across failure conditions

To test whether Arolla's effectiveness extends beyond the single scenario in §6.2.1, we independently sweep three axes of fault severity: offered load, failure rate, and fault duration. At each setting, we repeat the experiment 10 times; the plotted marker shows the median recovery time, the shaded band spans the min-max range across runs, and runs that never recover are shown in the top "no recovery" region. Across all three sweeps, Arolla is the only mechanism that recovers in every tested condition while maintaining a relatively tight spread. In contrast, no control and circuit breaker fail much earlier, while retry budget works over a broader range but is noticeably less stable.

Recovery vs. load (Fig. 5a). We sweep offered load from 600 to 1,600 RPS with a fixed 10 s, 100% abort fault. Arolla recovers in every run across the full range: recovery time stays near zero at light load and increases only gradually at the highest offered loads. Retry budget remains effective over much of the range as well, but becomes unstable from about 1,200 RPS onward, where the recovery time increase and some runs fail to recover entirely. In contrast, once load is high enough, neither no control nor circuit breaker recovers.



(a) All six clients dispatch at 200 RPS (equal base load). **(b)** Polite at 50 RPS each, aggressive at 350 RPS each

Figure 6: Fairness of retry admission across heterogeneous clients. Three *polite* clients (up to 3 retries per request) and three *aggressive* clients (up to 10 retries per request) issue the same `POST /cart` workload during a 20 s, 50% abort fault at `cartservice`. Bars report the number of retries admitted by Arolla, summed within each client group. The dashed line marks the expected allocation target.

In contrast, once load is high enough, neither no control nor circuit breaker recovers.

Recovery vs. failure rate (Fig. 5b). We sweep the abort percentage from 10% to 100% at 1,200 RPS with a fixed 10 s fault. Arolla again recovers in every run and keeps recovery time low across the full range. Retry budget is similarly effective through most of the sweep, but becomes unstable at the most severe point, 100% abort, where recovery time spreads widely and some runs fail to recover. No control and circuit breaker degrade much earlier: both become unreliable at moderate abort rates, and by high abort rates they largely or entirely stop recovering.

Recovery vs. fault duration (Fig. 5c). We sweep fault duration from 5 s to 60 s at 1,200 RPS and 100% abort. Arolla recovers at every duration we test, although recovery time increases with fault duration and shows a wider spread for longer faults. This pattern is consistent with larger post-fault backlogs after longer disruption windows. No control and circuit breaker fail to recover at all tested durations, including the shortest 5 s fault. Retry budget again falls in between: it often recovers, but with high variability, and several durations show both very wide spreads and no-recovery runs.

6.3 Fairness

This experiment evaluates whether attempt-aware retry allocation (§4.2) prevents aggressive callers from taking a disproportionate share of retry capacity. When multiple clients share a downstream service, retry capacity should reflect underlying traffic demand rather than who retries most aggressively. A client that sends more base traffic should receive more retry capacity, because it generates more legitimate retry demand. However, that client should not receive an additional advantage simply by issuing more retry attempts per request, nor should lower-load clients be crowded out entirely. We therefore evaluate fairness under both equal-load and skewed-load traffic mixes.

Setup. We build on the `POST /cart` setup of §6.2, but replace the single client population with six heterogeneous clients. Three polite clients issue at most 3 client-side retries per request, while three aggressive clients issue at most 10. We use a 20 s, 50% abort fault at `cartservice` so that retry capacity remains scarce but nonzero. We compare two Arolla variants: the aggregate-only bucket from §4.1 and the attempt-aware allocation policy from §4.2. We evaluate two traffic mixes: equal load, where all six clients send 200 RPS, and skewed load, where polite clients send 50 RPS each and aggressive clients send 350 RPS each.

Equal load (Fig. 6a). With equal base traffic, the two groups should receive similar retry capacity. Aggregate-only admission does not achieve this: it favors aggressive clients because they retry more times per request and therefore hit the shared bucket more often. As a result, aggressive clients receive substantially more admitted retries than polite clients. Attempt-aware allocation largely removes this advantage and brings the two groups much closer to parity. Concretely, aggregate-only Arolla admits 815 retries from aggressive clients versus 309 from polite clients, whereas the attempt-aware variant shifts admission to 481 and 547, respectively.

Skewed load (Fig. 6b). When aggressive clients send more base traffic, they should also receive more retry capacity. The fairness goal is therefore not equal allocation, but allocation that roughly tracks traffic volume while preventing the higher-load group from gaining an extra share through more aggressive retrying. Aggregate-only Arolla over-allocates retry capacity to aggressive clients. Attempt-aware allocation narrows this gap. Aggressive clients still receive more admitted retries, as they should, but polite clients are no longer crowded out by the combination of higher traffic volume and more aggressive retry behavior.

Summary. Attempt-aware allocation makes retry admission track base traffic rather than retry aggressiveness. Under equal load, it moves the two groups toward parity; under skewed load, it preserves the heavier group's larger share while limiting the extra advantage from more aggressive retrying.

6.4 Case Study: The Slack Outage

To test whether Arolla addresses a real production failure mode beyond our synthetic microservice scenario, we reproduce the dynamics of Slack's February 22, 2022 outage [53].

The incident. Slack's data path routes client queries through an application tier that consults a Memcached cluster via the mcrouter proxy; on cache misses the request falls through to a sharded MySQL (Vitess) backend. On the day of the incident, a routine service discovery software (Consul) update restarted 25% of the memcached nodes, flushing their caches. The resulting cache-miss spike routed a high fraction of queries directly to MySQL, and queries over the group-direct-message (GDM) keyspace were particularly expensive: they required scatter-gather across every Vitess shard. Database query latency rose; the application layer interpreted slow responses as transient failures and retried with exponential backoff. Retry load prevented the database from draining its queue, and the cache could not rebuild from an overloaded DB. The system entered a metastable state that persisted until operators manually throttled incoming load.

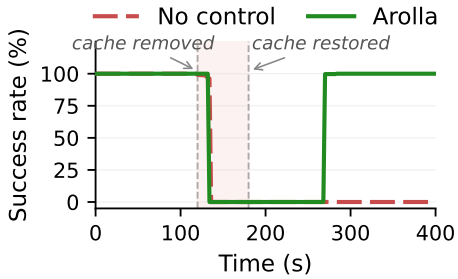
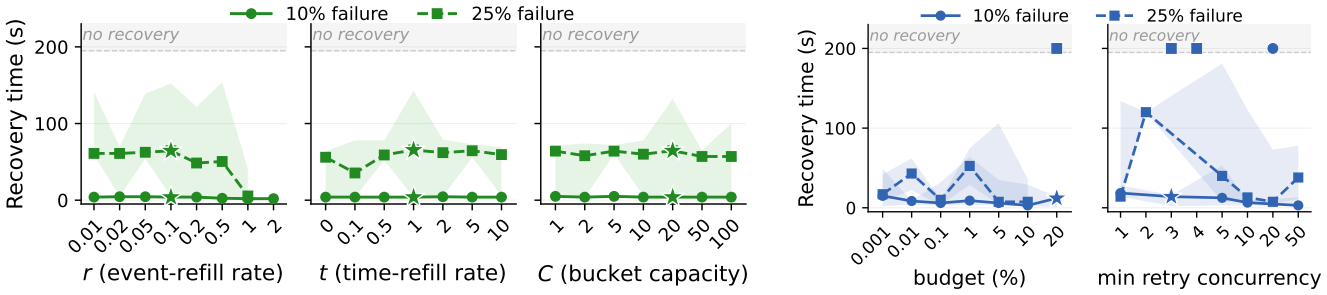


Figure 7: Slack outage reproduction. The cache is removed at t=120 s and restored (cold) at t=180 s. Without Arolla, success rate collapses to 0% and does not recover. With Arolla, success rate initially drops but recovers to 100% within ∼2 minutes after cache restoration.

Reproduction and Arolla's role. We reproduce the essential dynamics on a 6-node Kubernetes cluster: one control-plane node, three memcached replicas, a MySQL instance with memory capped at `450MiB`, and a single node running mcrouter plus a mock application implementing the read-through cache pattern. At t=120s we remove all memcached replicas from the cluster, forcing a cache-miss spike that exceeds MySQL's capacity; clients retry failed requests with exponential backoff up to 10 attempts. Without Arolla (Fig. 7), the client-observed success rate collapses to 0% within 15 s of the fault and remains there for the full 10-minute observation window, reproducing the self-sustaining retry storm described in the postmortem. With Arolla deployed at the application's outbound sidecar, success rate also drops to 0% initially because the cache is empty and MySQL cannot serve all the requests at the same time, but the retry storm is throttled at the sidecar rather than reaching the DB. Each cache-miss query now has a real chance to complete, cache entries accumulate, subsequent queries hit the cache, and after ∼2 minutes the system returns to 100% success without operator intervention.

6.5 Overhead

We measure Arolla's CPU and memory overhead on Online Boutique by comparing no control, empty Wasm, and full Arolla under healthy `POST /cart` traffic. We test steady traffic at 600 RPS and bursts alternating between 180 seconds at 450 RPS and 60 seconds at 1,050 RPS, averaging 600 RPS. Each configuration and workload has five trials, with a 240-second warm-up and a 720-second measurement window. We randomize configuration order, fix worker placement, and restart proxies between configurations. CPU counters and working-set memory are sampled every five seconds; results



(a) Arolla parameters.

(b) Retry budget parameters.

Figure 8: Parameter sensitivity under a 10 s and 25 s, 100% `cartservice` abort at 1,200 RPS. Stars mark default values; the top shaded band indicates no recovery. **(a)** Arolla's recovery time varies only modestly across wide sweeps and all tested settings recover. **(b)** Retry budget is more sensitive, and some settings fail under the severe fault.

average the two replicas and five trials.

Under steady traffic, Arolla increases CPU usage over no control by 10.1% at the gateway, 9.5% at frontend, and 7.3% at cartservice. Its additional cost over empty Wasm is 3.7%, 3.4%, and 1.5%, respectively. This processing includes per-request state, header checks, metrics, and bucket refill after successful responses. We also examine overhead during traffic bursts: at the gateway, Arolla adds 4.1% over empty Wasm at 450 RPS and 4.6% at 1,050 RPS. Including the Wasm filter, overhead relative to no control remains near 11.5% in both phases, indicating approximately proportional CPU cost as traffic increases. All requests succeed with no observed mesh retries, so these costs reflect processing that remains active under healthy traffic.

Memory overhead is largely a fixed cost of the loaded Wasm configuration. Empty Wasm adds approximately 40-43 MiB per idle proxy, with little further memory added by Arolla. Under steady traffic, full Arolla uses 33–39 MiB more than no control, while its working set changes by less than 1 MiB between quiet and burst phases. These results suggest that most of the extra memory is already present with empty Wasm; enabling Arolla's logic or increasing traffic adds little further memory.

We have also explored integrating Arolla into Envoy's native traffic-control mechanisms, with policy configuration exposed through Istio and Gateway API. This approach implements retry admission directly in the native data plane, offering a path to reducing the CPU and memory overhead.

6.6 Parameter Sensitivity

So far, we evaluate Arolla, retry budget, and circuit breaker at default settings. We next test whether the results are robust to parameter choices. In practice, a retry-admission mechanism should tolerate imperfect tuning. Here we focus on Arolla and retry budget. We omit circuit breaker because its behavior depends on many tuning knobs and was already ineffective in our earlier results.

Setup. For each approach, we sweep one parameter at a time while holding the others at their default values(marked by stars in Fig. 8). We evaluate two fault durations, 10 s and 25 s, both with 100% abort at `cartservice` under 1,200 RPS offered load. Arolla has three parameters: event-refill rate r (swept 0.01–2.0), time-refill rate t (0–10), and bucket capacity C (1–100). Retry budget has two: budget percentage (0.001%–20%) and minimum retry concurrency (1–50).

Results. Arolla is largely insensitive to parameter choice across all three sweeps (Fig. 8a). Changing parameters over wide ranges has only a modest effect on recovery time. The dominant factor is fault severity rather than parameter choice: the 25 s fault duration consistently recovers more slowly than the 10 s fault duration. More importantly, every Arolla configuration we test recovers.

Retry budget is substantially more sensitive to parameter choice (Fig. 8b). Larger budget percentages and higher minimum retry-concurrency floors make the policy more permissive, which can increase retry pressure on an already overloaded service. But smaller values are not uniformly better either, because an overly restrictive budget leaves too little slack to adapt across different loads and fault severities. As a result, retry budget shows no clear tuning trend. Nearby settings can behave quite differently, and the best value under one fault duration need not remain best under another.

Overall, Arolla appears easier to configure: across the ranges we test, default and nearby settings all recover, and even fairly extreme values remain effective. Retry budget appears more sensitive to tuning, with no single parameter setting performing consistently well across the fault conditions we evaluate.

7 Discussion

Assumptions. Arolla targets retry storms caused by failures and misconfiguration, not adversarial clients attempting to evade admission control. Within the mesh, retry attempts are tracked by Envoy via `x-envoy-attempt-count`, which is set by the upstream sidecar and cannot be rewritten by the

application; mesh-internal retries therefore carry a trusted attempt number. At the ingress, external SDKs may provide an attempt header such as `X-Attempt-Number`, but assumes that external clients report their initial retry attempts honestly. Supporting zero-trust clients would require the proxy to infer retry identity independently, which we leave to future work.

When Arolla does not help. Arolla specifically targets overload sustained or amplified by retries; it is not a general overload-control mechanism. If first-attempt traffic alone exceeds the service's sustainable capacity, limiting retries cannot restore stability. For the same reason, Arolla does not address cold-start avalanches dominated by first attempts. Arolla also requires retries to traverse an Envoy proxy where they can be identified and gated. Our current implementation therefore does not control raw TCP traffic or application-layer retries that bypass the sidecar.

Can Arolla itself become a failure point? Arolla is designed to avoid introducing a shared bottleneck. Admission is enforced locally at each Envoy proxy using only per-upstream bucket state; there is no centralized coordinator service on the request path. As a result, Arolla scales with the number of service instances: when a service scales out, retry admission capacity scales out with the proxies that protect it.

8 Related Work

Metastable failures. Bronson et al. [11] introduced metastable failures as degraded states sustained by feedback loops such as retries. Huang et al. [43] surveyed public incidents and found retries to be a common sustaining mechanism; our study follows a similar methodology but focuses on retry-control failures. Alvaro and Isaacs et al. [1, 46] develop a CTMC-based framework that predicts when a system parameterization will exhibit metastability, but does not propose a runtime mitigation. Arolla is complementary: it provides the runtime control that such analyses can help configure.

Adaptive retry and circuit-breaking mechanisms. Recent work has proposed adaptive retry and circuit-breaking policies for microservices. Sedghpour et al. [57, 58] adapt per-client retry behavior based on observed metrics, while Tavori et al. [62] propose RetryGuard, which enables or disables retries per service based on local signals. These approaches improve over static tuning, but they still do not enforce a shared retry budget across all callers or allocate retry capacity fairly under contention. Service meshes also provide proxy-layer controls. Envoy's retry budget limits retries generated by an outbound proxy as a fraction of active and pending upstream requests [26], and the budget is coupled to in-flight load rather than downstream recovery, therefore high outstanding demand can permit more retries even when the service is unhealthy. Arolla instead gates identifiable retries at the protected service's inbound proxy, where callers at that enforcement point share a common admission bucket. Its event-based refill couples retry capacity to successful downstream completions, while attempt-aware pricing prevents aggressive callers from consuming disproportionate retry capacity without maintaining per-tenant state. Montesi and Weber [52] survey circuit-breaker deployment strategies and observe that proxy-side placement can protect services against misbehaving clients, an early observation consistent with the proxy-layer enforcement point that Arolla operationalizes.

Many-to-one overload and receiver-driven control. Retry storms share the many-to-one pattern long studied in networking as *incast* [66]: many senders converge on a single receiver, the shared bottleneck saturates, and timeout-driven retransmissions amplify the load. Receiver-driven protocols such as Homa [51] and in-network credit schemes such as BFC [38] gate each sender on explicit permission, so aggregate inbound traffic cannot outrun the bottleneck's service rate. TVA [70] and Phalanx [23] extend this idea to adversarial senders: a sender must first obtain a capability before its packets are forwarded. Arolla attacks a similar problem at a different layer: it controls application-level retries at the service-mesh proxy and couples admission to downstream success.

Overload control in microservices. Prior microservice overload controllers regulate total offered load using priorities, credits, pricing, or learned admission policies, including DAGOR [72], Breakwater [15], Protego [16], TopFull [68], and Rajomon [67]. Arolla targets the narrower, complementary problem of admitting retries specifically, so retry-induced amplification does not crowd out capacity needed for first attempts. Because Arolla always admits first attempts and only gates retries, it composes naturally with these schemes: a service can deploy an overload controller for total load and Arolla for retry-specific admission.

9 Conclusion

Retry storms are common in production, but client-side defenses remain fragile because callers regulate retries independently without a global view of aggregate load. We presented Arolla, a retry-admission mechanism that bounds retry load and fairly allocates capacity across heterogeneous callers. On a realistic microservice testbed, Arolla recovers from retry-induced collapse, remains robust across parameter settings, and prevents aggressive clients from dominating retry capacity. We also reproduce a production outage and show that Arolla automatically suppresses the retry storm. More broadly, effective retry control requires a service-level view of aggregate caller load.

Acknowledgments

We would like to thank Tom Anderson, Rebecca Isaacs, Berni Schiefer, Ismail Oukid, Tyler Neely, and Daniel Ritter for their valuable insights and input during discussions. We also thank our colleagues from the EASL Group, Systems Group, and our anonymous reviewers, for helpful feedback and suggestions. Yazhuo Zhang is supported by the Swiss National Science Foundation (project TMSGI2_218019).

References

[1] Peter Alvaro, Rebecca Isaacs, Rupak Majumdar, Kiran-Kumar Muniswamy-Reddy, Mahmoud Salamati, and Sadegh Soudjani. Formal analysis of metastable failures in software systems. *arXiv preprint arXiv:2510.03551*, 2025.

[2] Amazon Web Services. Summary of the amazon ec2 and amazon rds service disruption in the us east region. https://aws.amazon.com/message/65648/, 2011. Accessed: 2026-03-27.

[3] Amazon Web Services. Summary of the october 22, 2012 aws service event in the us-east region. https://aws.amazon.com/message/680342/, 2012. Accessed: 2026-03-27.

[4] Amazon Web Services. Summary of the amazon dynamodb service disruption and related impacts in the us-east region. https://aws.amazon.com/message/5467D2/, 2015. Accessed: 2026-03-27.

[5] Amazon Web Services. aws-sdk-go-v2: aws/retry/jitter_backoff.go. https://github.com/aws/aws-sdk-go-v2/blob/main/aws/retry/jitter_backoff.go, 2020. Accessed: 2026-04-21.

[6] Amazon Web Services. Summary of the aws service event in the northern virginia (us-east-1) region. https://aws.amazon.com/message/12721/, 2021. Accessed: 2026-03-27.

[7] Amazon Web Services. Summary of the amazon dynamodb service disruption in the northern virginia (us-east-1) region. https://aws.amazon.com/message/101925/, 2025. Accessed: 2026-03-27.

[8] Amazon Web Services. Configure retries and timeouts — AWS SDK for Go v2. https://docs.aws.amazon.com/sdk-for-go/v2/developer-guide/configure-retries-timeouts.html, 2026. Accessed: 2026-04-21.

[9] Peter Bailis and Kyle Kingsbury. The network is reliable: An informal survey of real-world communications failures. *Queue*, 12(7):20–32, 2014.

[10] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site reliability engineering: how Google runs production systems*. " O'Reilly Media, Inc.", 2016.

[11] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, pages 221–227. ACM, 2021.

[12] Marc Brooker. Timeouts, retries, and backoff with jitter. Amazon Builders' Library https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/, 2019. Accessed: 2026-03-02.

[13] Marc Brooker. Fixing retries with token buckets and circuit breakers. https://brooker.co.za/blog/2022/02/28/retries.html, 2022. Accessed: 2026-04-19.

[14] Keertana Chidambaram, Qiuling Xu, Ko-Jen Hsiao, and Moumita Bhattacharya. Rebuilding netflix video processing pipeline with microservices. https://netflixtechblog.com/rebuilding-netflix-video-processing-pipeline-with-microservices-4e5e6310e359, 2024. Accessed: 2026-04-20.

[15] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for μs-scale RPCs with Breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 299–314. USENIX Association, 2020.

[16] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 725–738. USENIX Association, 2023.

[17] CircleCI. Db performance issue - incident report for circleci. https://circleci.statuspage.io/incidents/hr0mm9xmm3x6, 2015. Accessed: 2026-03-27.

[18] Coinbase. Brief incident post-mortem: January 6–7, 2021. https://www.coinbase.com/blog/brief-incident-post-mortem-january-6-7-2021, 2021. Accessed: 2026-04-19.

[19] Ting Dai, Jingzhu He, Xiaohui Gu, and Shan Lu. Understanding Real-World Timeout Problems in Cloud Server Systems. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 1–11, April 2018.

[20] David Poblador i Garcia. Incident management at spotify. https://labs.spotify.com/2013/06/04/incident-management-at-spotify/, 2013. Accessed: 2026-03-27.

[21] Laura de Vesine, Rob Thomas, and Maciej Kowalewski. Failure is inevitable: Learning from a large outage, and building for reliability in depth at Datadog. https://www.datadoghq.com/blog/engineering/rethinking-reliability/, 2025. Accessed: 2026-04-19.

[22] Discord Engineering. Connectivity issues - incident report for discord. https://status.discordapp.com/incidents/dj3l6lw926kl, 2017. Accessed: 2026-03-27.

[23] Colin Dixon, Thomas E Anderson, and Arvind Krishnamurthy. Phalanx: Withstanding multimillion-node botnets. In *NSDI*, volume 8, pages 45–58, 2008.

[24] Elastic. Elastic cloud january 18, 2019 incident report. https://web.archive.org/web/20201107233826/https://www.elastic.co/blog/elastic-cloud-january-18-2019-incident-report, 2019.

[25] Envoy Proxy Project. Circuit breakers: Outlier detection. https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/cluster/v3/outlier_detection.proto. Accessed: 2026-04-19.

[26] Envoy Proxy Project. Circuit Breakers: Retry Budget. https://www.envoyproxy.io/docs/envoy/latest/api-v3/config/cluster/v3/circuit_breaker.proto. Accessed: 2026-04-19.

[27] Envoy Proxy Project. Router — HTTP filters: retry policy. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/router_filter, 2026. Accessed: 2026-04-21.

[28] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *8th USENIX symposium on networked systems design and implementation (NSDI 11)*, 2011.

[29] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM Conference*, pages 350–361. ACM, 2011.

[30] Adam Gluck. Introducing domain-oriented microservice architecture. https://www.uber.com/us/en/blog/microservice-architecture/, 2020. Accessed: 2026-04-20.

[31] Google. google.api_core.retry.retry_base — source. https://googleapis.dev/python/google-api-core/latest/_modules/google/api_core/retry/retry_base.html, 2017. Accessed: 2026-04-21.

[32] Google Cloud. Google app engine incident #15025. https://status.cloud.google.com/incident/appengine/15025, 2015. Accessed: 2026-04-19.

[33] Google Cloud. Google cloud networking incident #19009. https://status.cloud.google.com/incident/cloud-networking/19009, 2019. Accessed: 2026-03-27.

[34] Google Cloud. Google compute engine incident #19008. https://status.cloud.google.com/incident/compute/19008, 2019. Accessed: 2026-03-27.

[35] Google Cloud. Cloud filestore listinstances api failed with error code 429 globally. https://status.cloud.google.com/incidents/X8SNkK2BPyCrc1sveeiu, 2022. Accessed: 2026-03-27.

[36] Google Cloud. Multiple gcp products are experiencing service issues. https://status.cloud.google.com/incidents/ow5i3PPK96RduMcb1SsW, 2025. Accessed: 2026-03-27.

[37] Google Cloud Platform. Online boutique: Cloud-native microservices demo application. GitHub, 2024. Accessed: 2026-04-01.

[38] Prateesh Goyal, Preey Shah, Naveen Kr Sharma, Mohammad Alizadeh, and Thomas E Anderson. Backpressure flow control. In *Proceedings of the 2019 Workshop on Buffer Sizing*, pages 1–3, 2019.

[39] gRPC. Retry Authors. https://grpc.io/docs/guides/retry/, 2025. Accessed: 2026-04-16.

[40] Haryadi S Gunawi, Mingzhe Hao, Riza O Suminto, Agung Laksono, Anang D Satria, Jeffry Adityatama, and Kurnia J Eliazar. Why does the cloud stop computing? lessons from hundreds of service outages. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pages 1–16, 2016.

[41] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, et al. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies (FAST)*, pages 1–14, 2018.

[42] Fred Hebert. Incident review: What comes up must first go down. https://www.honeycomb.io/blog/incident-review-what-comes-up-must-first-go-down/, 2023. Accessed: 2026-03-27.

[43] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 73–90, 2022.

[44] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray failure: The achilles' heel of cloud-scale systems. In *Proceedings of the 16th Workshop*

on Hot Topics in Operating Systems (HotOS), pages 150–155. ACM, 2017.

[45] Rebecca Isaacs. Analyzing metastable failures. Keynote Address Abstract, 16th Biennial Conference on Innovative Data Systems Research (CIDR), 2026. Available at https://www.cidrdb.org/cidr2026/keynotes.html.

[46] Rebecca Isaacs, Peter Alvaro, Rupak Majumdar, Kiran-Kumar Muniswamy-Reddy, Mahmoud Salamati, and Sadegh Soudjani. Analyzing metastable failures. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS)*, pages 172–178. ACM, 2025.

[47] Istio Project. Istio service mesh. https://istio.io, 2024. Accessed: 2026-03-02.

[48] Michał Kosmulski. Postmortem — why allegro went down. https://allegro.tech/2018/08/postmortem-why-allegro-went-down.html, 2018. Accessed: 2026-03-27.

[49] Kubernetes SIG Network. GEP-3388: Http route retry budget. https://gateway-api.sigs.k8s.io/geps/gep-3388/, 2024. Accessed: 2026-04-16.

[50] Linkerd Project. Service Profiles. https://linkerd.io/2.12/reference/service-profiles/. Accessed: 2026-04-17.

[51] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.

[52] Fabrizio Montesi and Janine Weber. Circuit breakers, discovery, and API gateways in microservices. *arXiv preprint arXiv:1609.05830*, 2016.

[53] Laura Nolan. Slack incident on 2-22-22. https://slack.engineering/slacks-incident-on-2-22-22/, 2022. Accessed: 2026-03-27.

[54] Matthew Prince. Cloudflare outage on november 18, 2025. https://blog.cloudflare.com/18-november-2025-outage/, 2025. Accessed: 2026-03-27.

[55] Resilience4j. Retry. https://resilience4j.readme.io/docs/retry, 2023. Accessed: 2026-04-21.

[56] Resilience4j. Circuitbreaker. https://resilience4j.readme.io/docs/circuitbreaker, 2025.

[57] 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 *IEEE International Conference on Cloud Engineering (IC2E)*, pages 32–42. IEEE, 2023.

[58] Mohammad Reza Saleh Sedghpour, Cristian Klein, and Johan Tordsson. An empirical study of service mesh traffic management policies for microservices. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE)*, pages 17–27. ACM, 2022.

[59] Amazon Web Services. Retry behavior. https://docs.aws.amazon.com/sdkref/latest/guide/feature-retry-behavior.html, 2026. Accessed: 2026-04-16.

[60] Signal and Hacker News Community. Signal is experiencing technical difficulties (Hacker News discussion). https://news.ycombinator.com/item?id=25803010, 2021. Accessed: 2026-04-19.

[61] Softonic. axios-retry. https://github.com/softonic/axios-retry, 2025. Accessed: 2026-04-21.

[62] Jhonatan Tavori et al. RetryGuard: Preventing self-inflicted retry storms in cloud microservices applications. *arXiv preprint arXiv:2511.23278*, 2025.

[63] Tenacity. Tenacity. https://tenacity.readthedocs.io/en/latest/, 2026. Accessed: 2026-04-21.

[64] The urllib3 Project. *Utilities — urllib3 Documentation*. Read the Docs, 2026. Accessed: 2026-02-24.

[65] Coburn Watson, Scott Emmons, and Brendan Gregg. A microscope on microservices. https://netflixtechblog.com/a-microscope-on-microservices-923b906103f4, 2015. Accessed: 2026-04-20.

[66] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. Ictcp: Incast congestion control for tcp in data center networks. In *Proceedings of the 6th International COnference*, pages 1–12, 2010.

[67] Jiali Xing, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. Rajomon: Decentralized and coordinated overload control. In *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 2025.

[68] Jiali Xing et al. TopFull: An adaptive top-down overload control for SLO-oriented microservices. In *Proceedings of the ACM SIGCOMM Conference*. ACM, 2024.

[69] David Yanacek. Using load shedding to avoid overload. Amazon Builders' Library, 2019. Accessed: 2026-03-02.

[70] Xiaowei Yang, David Wetherall, and Thomas Anderson. A dos-limiting network architecture. *ACM SIGCOMM Computer Communication Review*, 35(4):241–252, 2005.

[71] Yazhuo Zhang, Rebecca Isaacs, Yao Yue, Juncheng Yang, Lei Zhang, and Ymir Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 502–519, 2023.

[72] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling WeChat microservices. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, pages 149–161. ACM, 2018.