Roshanfer: Achieving Performance Resilience in Cloud Microservices

Farzad Mohammadi
Imperial College London
London, United Kingdom
f.mohammadi24@imperial.ac.uk

Theo Akande*
IMC Trading
Amsterdam, Netherlands
theoakande1@gmail.com

Marios Kogias
Imperial College London
London, United Kingdom
m.kogias@imperial.ac.uk

Abstract

Dynamic and unpredictable load affects the performance resilience of cloud services. Frequent and sporadic overloads cause performance degradation that manifests as increased latency, SLO violations, and reduced goodput. Despite the plethora of mechanisms to deal with overload, ranging from auto-scaling and service degradation to careful admission control and load shedding, microservice-based services remain vulnerable to such problems. Existing overload control frameworks depend on slow, reactive designs, and fail to cater to the needs of modern services with dynamic call graphs and changing workloads.

We present Roshanfer, a proactive overload control system for microservice-based deployments. With Roshanfer, microservices are overload-proof by design because Roshanfer proactively controls and strictly bounds the number of admitted requests. To do so, it depends on a novel mechanism that creates closed systems between microservice pairs. This design allows fast backpressure and request queue buildup only at the service gateway, which is the centralized point for admission control and policy enforcement. To our knowledge, Roshanfer is the first overload control mechanism to deal with dynamic call graphs at a request granularity, and it is also supported by a robust TLA+ model that provides further correctness guarantees. Our evaluations on well-known benchmarks show that Roshanfer avoids SLO violations and goodput loss across various service call graphs and SLOs, even with concurrent requests for multiple APIs. Additionally, Roshanfer reduces 99th percentile latency by 54% and achieves 1.55× goodput in a large-scale service based on Alibaba traces compared to Rajomon, the current state-of-the-art.

CCS Concepts: • **Computer systems organization** → **Cloud computing**.

*Work done while affiliated with Imperial College London

EuroSys '27, Rabat, Morocco

ACM ISBN 979-8-4007-2971-3/2027/04
https://doi.org/10.1145/3842654.3848578

Keywords: Performance Resilience, Overload Control, Microservices

ACM Reference Format:
Farzad Mohammadi, Theo Akande, and Marios Kogias. 2027. Roshanfer: Achieving Performance Resilience in Cloud Microservices. In *22nd European Conference on Computer Systems (EuroSys '27), April 19–23, 2027, Rabat, Morocco.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3842654.3848578

1 Introduction

Performance resilience remains a long-standing goal and open problem for modern datacenter and cloud services. Both academia [13, 14, 56, 70, 81, 86] and industry [49, 82] have been trying to design systems that deliver their latency and throughput Service Level Objectives (SLOs), despite fluctuations in workload and incoming load. Specifically, they try to do so in cases where resource demand is higher than resource supply, a situation described as *overload* [78]. If not handled properly, overload can lead to throughput collapse [13, 78, 86] and SLO violations, impacting customers, revenue, or even service development [23, 24, 62].

There are multiple alternatives for dealing with overload at different layers of the stack, levels of granularity, response times, and costs. Control plane solutions operate at a second or minute granularity and commonly target long-lasting overloads. They observe certain metrics, e.g., CPU utilization or end-to-end application latency, and they react by auto-scaling [64, 65, 76, 83, 85], i.e., provisioning more resources, dropping requests through admission control [56, 70], or gracefully degrading certain application features [1, 41, 49]. Dataplane solutions operate at a finer granularity and target shorter-term bursts for which control planes could not react fast enough. They operate at the level of individual requests and selectively drop certain requests to maintain a high *goodput*, i.e., the rate of requests that complete within the latency SLO. Dataplane solutions can target either individual applications, such as SEDA [78] and Breakwater [13], or microservice-based deployments, such as Rajomon [81]. In most cases, production deployments depend on a combination of these approaches to deal with both sudden load spikes and longer-term overloads.

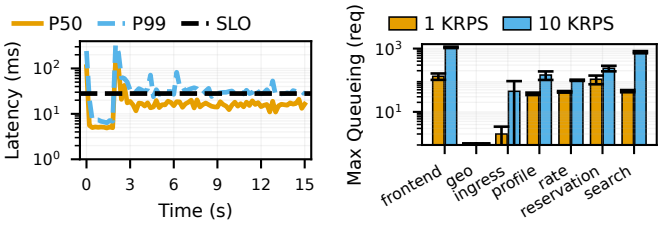
In this work, we focus on the problem of overload control for microservice-based deployments [27, 30, 47, 72], specifically on approaches that can handle overload within milliseconds and eliminate latency SLO violations. The way

modern services are built and deployed, i.e., as sets of loosely coupled microservices communicating over Remote Procedure Calls (RPCs), combined with the volatility of workload patterns [16, 47, 72], makes this a particularly challenging problem. Bottlenecks can change over time depending on the workload, making them hard to identify, sporadic overloads cause services to frequently oscillate between normal and overloaded conditions [80], and different parts of the service, e.g., different APIs or microservices, can interfere with each other [85], while queues can build up anywhere inside the deployment, substantially degrading performance resilience.

By carefully studying prior work [81, 86] that tackles the same problem, we identify several limitations that harm their efficiency and applicability and challenge their ability to deliver what they were initially deployed for, i.e., to maintain high goodput under overload. First, all prior approaches operate in a reactive manner, which can be slow and leads to SLO violations. Second, they allow for arbitrary and uncoordinated request drops throughout the service to deal with the reactive loop imperfections, which leads to precious resource waste, especially when resources are scarce under load, since requests can be dropped after they have been partially served. Third, none of the prior works can deal with dynamic call graphs, i.e., services whose execution path for even a specific API is not known a priori at the top-level request, which are common in production microservice-based deployments [47]. Finally, all of them depend on a large number of parameters with different knobs in each microservice, which makes them very brittle across workload changes and requires extensive training before deployment.

Our main insight is that overload control for microservice-based deployments can be proactive without any performance loss. By carefully designing and enforcing *closed systems* [25, 71] between microservices, we can tightly bound queuing inside services, pushing most of the queueing to the service boundary, i.e., the gateway. This splits the deployment into two parts: a performance-resilient core that requires only minimal configuration, receives new requests only when it completes existing ones, and guarantees strict upper bounds on queueing and consequently tail latency, and a centralized component at the gateway that leverages the resilience of the core and enforces admission policies.

We design, formalize using TLA+, and implement Roshanfer, a performance-resilient overload control scheme for modern microservice-based deployments. Roshanfer uses a credit-based mechanism that precisely controls and strictly bounds queuing in each microservice based on a mechanism inspired by the bandwidth-delay product (BDP) [52, 60] that dynamically configures per-microservice queue limits. Given this fine-grained and robust mechanism, Roshanfer is, to the best of our knowledge, the first microservice overload control system that deals with dynamic service graphs without any simplifying assumptions. Moreover, it does not allow for request drops inside the service, thus improving goodput,



(a) Latency dynamics

(b) Queueing before and after overload

Figure 1. High queueing and SLO violations happen with Rajomon [81]. Offered load jumps from 1 to 10 KRPS at $t = 2$.

while not requiring any reconfiguration or training across workload changes, allowing it to dynamically adapt to different workloads. Roshanfer follows a sidecar design [12, 15, 34] and remains agnostic to application logic and compatible with existing microservices implemented in any language, thus improving its deployability.

In our evaluation, we compare Roshanfer with Dagor [86] and Rajomon [81], the current state-of-the-art systems in microservice overload control. We show that Roshanfer completely eliminates SLO violations in benchmarks representing academic setups [22], industry large-scale deployments [47] with 30 microservices, and dynamic call graphs [18]. In a realistic large-scale service modelled after Alibaba production traces [18, 47], Roshanfer achieves up to 54% lower tail latency and 1.55× goodput compared to Rajomon, the state-of-the-art, even when offered load is 14× the capacity.

This paper makes the following contributions:

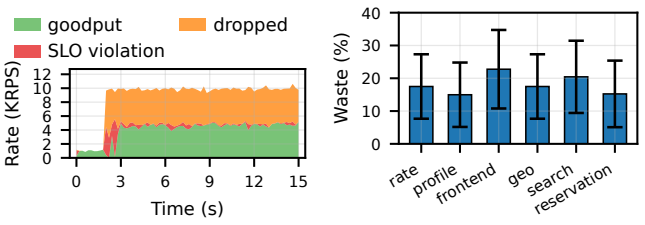
- An analysis of state-of-the-art overload control systems for microservice-based deployments and their design pitfalls that lead to performance unpredictability (§2).
- An architecture for microservice-based deployments comprising a resilient, efficient service core based on closed systems with tightly bounded queues and a centralized admission-control component, which is, to our knowledge, the first overload control approach that handles dynamic service graphs without simplifying assumptions (§3–§4).
- The formalization and implementation of that mechanism in a sidecar architecture for compatibility and ease of deployment (§5).

2 Background and Motivation

In this section, we zoom in on existing overload control mechanisms for microservice-based deployments that operate at a fine, i.e., request-level, granularity, and identify three pitfalls that motivate the need for a new approach to designing such systems.

2.1 Pitfall 1: Reactive Nature

Existing approaches to overload control [13, 81, 86] implement a reactive control loop. They monitor a certain system



(a) Goodput dynamics

(b) Resource waste across microservices

Figure 2. Goodput loss and resource waste under overload with Rajomon [81]. All requests are for the `search-hotel` API from the *Hotel Reservation* benchmark.

metric, such as queueing delay, and, through a mechanism that can vary across different approaches, control the rate of incoming requests into the system.

We experiment with Rajomon [81], the state-of-the-art system for overload control in microservice-based deployments, to identify the problems with this approach. Rajomon depends on a dynamic pricing model, where the per-microservice price is controlled according to the current demand and clients without enough credits are unable to send requests. Rajomon also allows individual microservices to drop requests to accommodate sudden price fluctuations. We create a simple overload scenario using the *Hotel Reservation* benchmark [22] and the `search-hotel` API. During the first 2 seconds, we send 1 KRPS of requests (well below capacity), and then we increase the load to 10 KRPS, exceeding capacity and overloading the service. We set the SLO for the 99^{th}-percentile (P99) latency to 28ms, which is 5× the P99 latency when the system is not overloaded (one request at a time). We measure and report the P99 latency in every 200ms window.

Observations: Figure 1 summarizes the experiment results. In Figure 1a we observe that Rajomon violates the latency SLO during overload. To investigate the root cause, we measure and plot the amount of queuing at each microservice in the call graph before and after the load increase. In Figure 1b, we observe the following. Before overload, microservices have enough resources to handle incoming requests, so queuing is minimal. Under overload, queues build up significantly (up to 10× in `frontend` for instance) at microservices despite Rajomon's overload control, which manifests as queuing delays and latency SLO violations.

In the same experiment, we also study the service goodput, i.e., the rate of requests completing within the latency SLO, which is a metric extensively used [13, 81] during overload as it combines throughput and latency. Such services are commonly user-facing and therefore latency-critical. Figure 2a plots goodput over time and categorizes requests into three groups: *goodput*: requests that finish within SLO, *dropped*: dropped requests, *SLO violated*: requests that violate the SLO. We make the following observations. After the load spike, a significant fraction of requests violate SLOs due to Rajomon's slow reaction, causing an initial goodput drop. Note, however, that the service still spends resources to serve those requests. For cloud tenants, this means paying for resources they cannot leverage. Even after Rajomon stabilizes, request dropping at backend microservices and SLO violations still occur, leading to more resource waste.

To understand microservice efficiency under overload, we measure the per-microservice resource waste and plot it in Figure 2b. We define resource waste as the fraction of RPCs that a microservice successfully processes but whose initial top-level request ultimately fails due to either a downstream RPC drop or an SLO violation. We observe that the `frontend` (up to 35%) and `search` (up to 33%) microservices incur higher resource waste due to their downstream dependencies, which increase the chance that their processed RPCs fail.

Root-cause analysis: To reason about the previous results and explain why Rajomon fails to offer performance resilience, given that there are still SLO violations and goodput fluctuations, we have to understand some of Rajomon's internals, which also generalize to other existing overload control systems [13, 56, 86]. Rajomon monitors the per-microservice queuing delay and adjusts prices only after queuing delay crosses its configured threshold. So, it reacts only after the problem manifests. *Slow detection* means that by the time the system reacts, many requests have already queued up, causing inevitable SLO violations. Rajomon, though, also suffers from *slow propagation*, given that the overload signal needs to propagate through the call graph back to the clients. In the above experiment, the longest path in the call graph has only 3 hops, yet Rajomon requires over a second to recover after the demand surge. To compensate for the slow reaction, microservices drop requests independently, further wasting resources.

2.2 Pitfall 2: Dynamic Call Graphs

We observe that widely deployed services [27, 30, 47, 72] follow a specific structure. Clients access the service through a *gateway*, which acts as the single entry point to the service. The gateway often implements load balancing, protocol translation, and request routing [36, 43]. Then, requests expand into a directed acyclic graph (DAG) of RPCs among RPC endpoints, flowing from intermediate microservices to leaf microservices [47, 72]. Microservices frequently fan out to many dependencies and are shared by many upstream microservices (fan-in). The callee microservice is commonly referred to as *downstream* by the caller, while the caller is *upstream* for the callee.

The prior state-of-the-art system, Rajomon [81], makes a sweeping simplification, assuming that the service call graph and the trajectory of each request are static and known when the request enters the system. Although this assumption simplifies the design of the overload control mechanism,

e.g., in Rajomon it allows microservices to calculate request prices based on deterministic downstreams, it is not realistic. In practice [18, 47], the trajectory of each request is dynamic and hard to predict even for a single service. The choice of a downstream microservice can depend on the execution in the caller or the execution of other downstream microservices. Consider, for example, the case of a web server that needs to access certain data and uses a database and a caching layer, e.g., memcached [53]. The web server will first try the cache, and only if the data is not there will it try the database, thus making the call graph and the set of microservices involved in serving a request unpredictable. Another example is a shopping application, where the arguments of a payment request can determine where to forward it, such as to a credit-card payment microservice or a debit-card alternative.

Existing overload controllers fail to deal with dynamic call graphs due to their coarse-grained overload signal aggregation. Both Rajomon and Dagor [86] rely on queuing delay changes measured periodically, which all possible call graphs of a service contribute to. This implicit and aggregated approach prevents identifying situations where, for example, one possible call graph is overloaded while others are not, which leads to SLO violations at the tail [16].

2.3 Pitfall 3: Configuration and Training Needs

Existing overload controllers rely on a plethora of parameters and configurations, leading to unrealistic deployment requirements. For example, given a service with M microservices, Rajomon requires $3\times M+2$ parameters. Our experience when deploying existing systems has revealed the following challenges: i) navigating the large space of parameters requires a pre-training phase that lasts hours, ii) the optimal parameter values can differ significantly from service to service and from workload to workload, and iii) any SLO value change requires re-training parameters because the SLO is directly used in the utility function for training. Given workload unpredictability and service dynamism, these limitations lead to brittle and time-consuming deployments.

3 Design

For performance resiliency, we set the following Resilience Properties (RP) for Roshanfer: **RP1: Bounded queuing.** Roshanfer should proactively bound queueing at individual microservices. Since avoiding SLO violations is a collaborative task among several subsystems, Roshanfer relies on two assumptions to provide SLO-bounded tail latency: 1. Head-of-line (HoL) blocking delay is minimized at each microservice using an appropriate CPU scheduler (e.g., a processor-sharing (PS) scheduler for microservices with high service time variability [79]). 2. The network is not congested and packets are delivered in a timely manner. Under these two conditions, bounded queueing prevents SLO violations and bounds the tail latency to the SLO, which is of paramount importance in production systems as failing to do so damages revenue [24, 62] or completely disrupts the service [23]. During overload, fast drops are preferred to SLO violations as requests can be retried [31, 69].

RP2: Stable goodput. Roshanfer should maintain stable goodput even under varying overload levels. **RP3: No partial request processing.** Roshanfer should avoid request drops at microservices to minimize resource waste. **RP4: Explainable adaptivity.** Roshanfer should be able to adapt to workload changes and support any kind of dynamic call graph without requiring offline training or per-microservice configuration, which would make the system brittle and hard to explain. In addition to these resilience properties, Roshanfer should not impose any restrictions on the service architecture or implementation language for ease of deployment purposes.

3.1 Overview

Roshanfer splits a microservice-based deployment into two separate parts, each dealing with different concerns. On one side, the service core includes all the microservices. It depends on carefully and tightly bounded queues between microservices that compose closed systems [71] and uses a credit-based mechanism to manage those. We later show how to statically define the specific per-queue bounds, which are independent of the current system load, thus enabling fully proactive queue bounds that do not suffer from Pitfall 1. Roshanfer picks the per-queue bounds such that they maximize throughput while minimizing the potential queuing time, thus avoiding overload in each microservice by construction and allowing fast *backpressure* to build up to the edge of the service core whenever a microservice approaches saturation. Once a request enters this service core, it is guaranteed neither to encounter unexpected queueing nor to cause overload. So, tightly bounded queueing and the credit-based mechanism enable Roshanfer to guarantee **RP1** and **RP2**.

On the other side, ingress, or the service gateway, controls the requests that enter the service core, separating the performance-resilient part from the unpredictable world. During overload, queues start building up in ingress given the fast backpressure from the service core. Ingress is the only place in Roshanfer where requests can be dropped, achieving **RP3** since no request is dropped after having consumed service resources. Given the performance-resilient and configuration-free core, ingress is also the only place that Roshanfer requires configuration, i.e., the target SLOs, to control the per-request waiting time there. This design decision significantly reduces the configuration space and allows Roshanfer to adapt to any microservice-based deployment, thus achieving **RP4**.

Figure 3 shows the different components comprising Roshanfer. On the left, *Ingress* queues requests during overload and

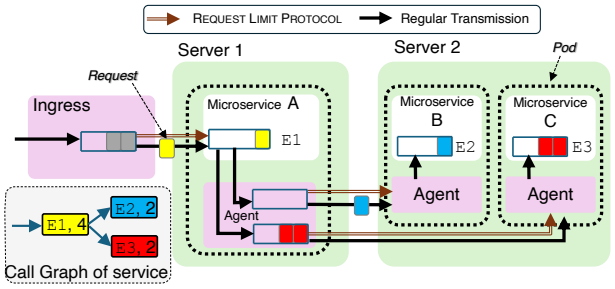


Figure 3. Overview of Roshanfer architecture. Numbers in the call graph show endpoint limits.

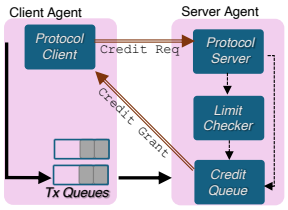


Figure 4. Agent's logical components and their interactions.

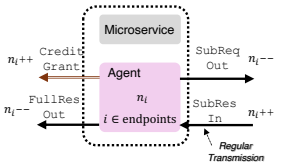


Figure 5. Active request n_i counting for endpoint i.

controls end-to-end tail latency, keeping it under the SLO by adjusting the size of its centralized queue through proactive request dropping when its queue is full. Roshanfer remains agnostic to application logic and employs a sidecar model [34], making it compatible with any RPC-based service, independent of the implementation language or application internals. The *Agent*, i.e., Roshanfer's sidecar, implements the Request Limit Protocol, i.e., the credit-based mechanism that limits the number of requests in each endpoint and forwards requests among microservices.

3.2 Agent

Roshanfer uses a credit-based mechanism to cap the number of requests in the service. This mechanism is receiver-driven and treats each RPC endpoint [1] separately, with each endpoint having its own limit. Client Agents must acquire a credit from server Agents before transmitting. Agents maintain per-endpoint request counts and their respective limits. Servers grant a credit for an endpoint only if the request count is less than the limit. Per-endpoint limits prevent per-endpoint overload.

Agents control request flow through the exchange shown in Figure 4. The client Agent intercepts all outgoing requests from the microservice and queues them internally. For every request, it sends an explicit `Credit Request` message to the server Agent and waits for a response. The server Agent pushes the request to the Credit Queue and consults the Limit Checker. The Limit Checker controls the number of requests in the microservice by deciding whether to grant a credit. If there are available credits, i.e., the outstanding request count is less than the per-endpoint limit, the server Agent returns a credit and the client can send a request. If not, the `Credit Request` remains in the credit queue until a credit becomes available, i.e., backpressure propagates upstream. Credits become available when the number of *active* requests in the downstream microservice goes down. When this happens, the Agent decides which `Credit Request` to grant based on a configurable prioritization policy.

Agents remain transparent to endpoint internals and count the *active* requests for every endpoint by monitoring network traffic. An active request is a request that i) the endpoint just sent a credit for, ii) is currently being serviced, or iii) is waiting to send a downstream request. The Limit Checker tracks active requests n_i for every endpoint i by monitoring four events (Figure 5). It increments the count when the Agent grants a credit to an upstream microservice (`Credit Grant`) and decrements it when the microservice returns a response (`FullRes Out`). Crucially, it also decrements the number of active requests when a sub-request is issued to the downstream microservice (`SubReq Out`), and increments it when the sub-request returns (`SubRes In`). By treating RPC endpoints as black boxes, Agents can serve different types of microservices, i.e., leaf nodes or intermediate nodes that have one or multiple downstream dependencies.

Agents support fan-in and fan-out patterns too. The centralized nature of the Credit Queue in Agents (Figure 4) inherently enables the fan-in pattern. For fan-out patterns, however, if we naively updated the number of active requests for each sub-request, all sub-requests from a single parent would be counted as independent sub-requests. To prevent this, Roshanfer relies on a distributed request ID set by Ingress and carried by each request and inherited by its sub-requests. Thus, Agents decrement the active count only once per unique parent request ID, regardless of the fan-out degree.

Algorithm 1: Queue adjustment algorithm at Ingress.

```
1  Function AdjustQueueSize(e2e_p99_i)
2      smoothed_i ← α * e2e_p99_i + (1 − α) * smoothed_i;
4      err ← (smoothed_i − SLO_i)/SLO_i;
5      u_i ← time-weighted queue i utilization;
6      if err > err_d then
7          Q*_i ← Q*_i / adj_d;
8      else if err < err_i then
10         if u_i > 0.8 × Q*_i then
11             Q*_i ← Q*_i + (-err)× adj_i;
12         else
14             Q*_i ← max(u_i, Q*_i / adj_d, MIN_LIMIT);
```

3.3 Overload Control at Ingress

Bounding the number of requests in the service with the Request Limit Protocol enables Ingress to centrally apply admission control and drop requests if necessary during

[1] We assume that a microservice can have multiple RPC endpoints

overload. Ingress communicates with Agents using the same Request Limit Protocol, acting as a client to those Agents. When any microservice is at capacity, i.e. overloaded, backpressure will propagate, leading to extra requests queueing in Ingress waiting for credit. Ingress maintains separate queues for each service API.

Given the bounded tail latency at the performance-resilient service core, Roshanfer can control requests' end-to-end latency by configuring the waiting delay at Ingress, i.e., the size of the Ingress queue. Unlike other overload control mechanisms that require configuring a set of parameters distributed across the entire deployment, Roshanfer depends on this single parameter. In the simple case of a service with a single API, there is a maximum queue size Q where queueing beyond that leads to SLO violation. Note that this Q only depends on the latency SLO and end-to-end tail latency at the service, and not on the current incoming load. Thus, at any time, the actual queue size of an API needs to satisfy $q \leq Q$.

In the more general case with multiple APIs that might also share common microservices, APIs might interfere with each other, thus making the configuration of Q_i more challenging, as Q_i assumes API i is using the service exclusively, which can lead to SLO violations. Thus, Roshanfer requires a tighter upper bound, Q_i^*, which depends on the rate and mix of APIs receiving load at each point in time. In the general case we have: $q_i \leq Q_i^* \leq Q_i$. This defines a subspace of allowed queue size configurations that will not lead to SLO violations in an N-dimensional space, assuming there are N different APIs. There are multiple ways to identify that space, leveraging the predictability provided by the service core either online or offline and using tools such as machine learning or discrete event simulations.

In our current design of the Ingress, we chose a simple additive-increase/multiplicative-decrease (AIMD) algorithm to approximate Q_i^*. The intuition behind the algorithm is to keep the Q_i^* low in anticipation of bursts to avoid SLO violations or interference with other APIs and only increase it if necessary, i.e., if the queue utilization increases, while again guaranteeing that the end-to-end latency stays within the target SLO.

Algorithm 1 describes the control loop logic. The algorithm runs every 200ms or 100 responses. It maintains an estimate of the P99 end-to-end latency using an Exponential Weighted Moving Average (EWMA) and computes the relative error to the SLO (line 4). If the error is bigger than err_d (e.g., 0), then it decreases Q_i^* by a factor. If the error is smaller than err_i (e.g., -0.3), it means the end-to-end tail latency is well below the SLO, so Q_i^* can increase without risk. Simply increasing Q_i^* every time the error is small, though, will leave Roshanfer vulnerable to latency overshoots during sudden load spikes. To avoid this, Ingress only increases Q_i^* when this is necessary, i.e., the queue is actually being used, by checking if the queue is almost full (line 10). When the queue utilization is low, Q_i^* decreases to not incur high waiting delay during load spikes (line 14).

The above algorithm does not assume a physically centralized Ingress. Thus, it allows for a highly scalable deployment where independent Ingress instances are responsible for a specific API or a few APIs. Different Ingress instances do not need to explicitly synchronize with each other, since they implicitly communicate through the perceived changes to the end-to-end latencies and the delays to receive credits from the service core.

3.4 Dynamic Call Graphs

So far, we have only explained how the Request Limit Protocol works in the case of static microservice graphs. Unlike static graphs, where Ingress has a separate queue and can detect a bottleneck through fast queue buildup and backpressure, dynamic graphs can be partially bottlenecked, where only a fraction of requests hit the bottleneck path. This will slow down backpressure propagation, and Ingress will not react quickly enough to start dropping requests, thus leading to SLO violations.

We extend the Request Limit Protocol to quickly detect and report the existence of a bottleneck, even in the case of dynamic graphs, through a conservative approach that considers the worst case among all possible execution paths and applies backpressure accordingly. We consider the case of dynamic call graphs to be similar to the case of a fan-out pattern, since a microservice can send to *any* downstream microservice but it actually sends to *only one* of them at runtime. Thus, in this case the Request Limit Protocol will now ask all downstream microservices for credit but only use the one needed. Agents immediately return unused credits with a specialized `Credit Return` message. This way, Roshanfer can avoid cases where Ingress optimistically accepts requests anticipating that they will end up in the non-bottlenecked path, but instead end up in the bottlenecked one, leading to SLO violations. Through this mechanism, Roshanfer still preserves bounded queuing and instead drops extra requests quickly so they can be rerouted or retried (**RP1**).

Because Roshanfer operates at the granularity of every RPC, it does not require prediction of dynamic paths. An Agent observes all outgoing requests and their destinations. This enables Roshanfer to discover new paths on their first use (e.g., when a feature becomes available programmatically [49]). Thus, Roshanfer can update active paths and remove any inactive paths periodically. Such dynamic path discovery enables Roshanfer to adapt to dynamism in the call graph without explicit configuration.

3.5 An End-to-End Example

Here we present a simple example of how Roshanfer works, shown in Figure 3. For simplicity, this example assumes that endpoint limits are known and fixed. In this setup, we have a

single API with a simple fan-out pattern. The call graph spans three endpoints `E1`, `E2`, and `E3`. The limit for each endpoint is shown next to it in the call graph. Figure 3 captures a snapshot of the deployment where endpoints `E1`, `E2`, and `E3` have 4, 2, and 2 active requests, respectively. `E3` seems to be the bottleneck as there are 2 requests waiting for credit in `Microservice A` Agent's request transmission queue. Given that `E1` depends on `E3`, the back-pressure has successfully propagated to `E1` as `E1` has also reached its limit, despite `E2` not being a bottleneck. Request Limit Protocol naturally propagated the back-pressure to Ingress as we see 2 pending requests waiting there.

4 Determining Request Limits

So far, we have assumed that Agents are configured with per-queue limits and have explained how they enforce them via the Request Limit Protocol. We will now describe how Agents calculate those limits and explain how they can change in the case of microservices with multiple endpoints.

4.1 Per Endpoint Limits

Agents need to set the limit high enough to guarantee full microservice utilization and maximize throughput, but not higher, because doing so would lead to queueing and potential overloads. Previous work shows that CPU is the main bottleneck contributing to performance degradation in microservice-based deployments [76, 83, 85]. Thus, we focus on CPU cores and set the active-request limit high enough to keep all cores allocated to a microservice busy. The limit should also account for enough requests to hide the delay during the Request Limit Protocol, when a request is accounted for but not actively occupying a CPU, e.g., while a `Credit Grant` is in flight, to prevent CPUs from remaining idle at that time. Based on that rationale and inspired by the bandwidth-delay product (BDP) [52, 60], we calculate the number of requests that can be processed in a microservice during this idle time, and add it to the total request limit for each core.

Equation 1 formalizes this description for leaf nodes in the microservice graph. L_i is the limit of endpoint i, N_{cores} is the number of CPU cores allocated to the microservice, RTT is the average round-trip time between Agents, and s_i is the average service time at endpoint i. The second term is equivalent to BDP but for compute at the application layer, where $N_{\text{cores}} \times \frac{1}{s_i}$ and RTT correspond to service rate and delay, respectively. The delay of 1 RTT corresponds to waiting for the `Credit Grant - SubReq Out` exchange. We round up the second term of Equation 1 to ensure at least one request is pending when it evaluates to less than 1.

$$L_i = N_{cores} + \left\lceil N_{cores} \times \frac{1}{s_i} \times RTT \right\rceil \quad (1)$$

For non-leaf nodes, the formula becomes slightly more complex (Equation 2) because it now has to account for the idle time waiting to send requests to the downstream microservice. N_{calls} corresponds to the number of sequential downstream calls the microservice has to make.

$$L_i = N_{cores} + \left\lceil N_{cores} \times \frac{1}{s_i} \times (N_{calls} + 1) \times RTT \right\rceil \quad (2)$$

The Supplementary Materials explain this formula in detail. Note that using these formulas, Agents can also dynamically change per-endpoint limits. Agents use EWMA estimators for RTT and per-endpoint service times by tracking the inputs and outputs of the microservice, while N_{cores} is set by the cluster manager when provisioning each microservice. Thus, Roshanfer can handle changing workloads over time. Finally, limits depend only on the rounded-up ratio of RTT to service time, not on their exact values, making limits resilient to service time variability.

4.2 Microservices with Multiple Endpoints

Roshanfer's ideas extend to microservices with multiple endpoints, too. We assume the generic case where there are no dedicated cores for specific endpoints and there is no way to know the request ratio among the different endpoints, as this can change over time. Each endpoint should be able to temporarily consume all cores for its microservice. Thus, we calculate the per-endpoint limits following the same logic described in §4.1, considering each endpoint in isolation.

We separate the case of leaf and non-leaf microservices with multiple endpoints to understand how Roshanfer addresses challenges unique to each one. For non-leaf microservices, Roshanfer needs to ensure that if any of the downstream microservices becomes a bottleneck, the back-pressure created in the upstream microservice will not prevent the other non-bottlenecked endpoints from making forward progress. Thus, to account for any scenario where the downstream of one or more endpoints becomes a bottleneck, Roshanfer allows each endpoint limit to act independently, considering only requests active for that endpoint. Note that despite introducing overcommitment, the number of requests at a non-leaf microservice is still strictly bounded.

Treating leaf microservices the same, however, can lead to unnecessary queueing, as there is no downstream microservice that can create a problem by becoming a bottleneck. We introduce the *global limit* (GL) which describes the total number of credits in a leaf microservice with multiple endpoints. In this case, an Agent should check both the per-endpoint limit and GL before issuing a credit.

GL should be at least equal to the maximum per-endpoint limit among all collocated endpoints to guarantee full utilization by any single endpoint. Going lower than that would imply that at least one endpoint is throttled and cannot use all resources allocated to the microservice. On the upper bound, given that each endpoint cannot use more than its

limit, it does not make sense to configure the limit higher than the sum of all the per-endpoint limits, as a higher limit will lead to unnecessary queuing. So, $max(L_i) \leq \text{GL} \leq \sum_i L_i$.

Setting GL = $max(L_i)$ reduces the number of requests in the microservice and pushes the microservice to operate closer to a run-to-completion model. Increasing the level of overcommitment increases the number of concurrent requests, and if the microservice scheduler (OS or runtime) implements such a policy, the microservice moves more toward a PS model. Consequently, microservices that expect high variability in their service times should set GL higher than $max(L_i)$ [26, 38].

5 Implementation

Roshanfer consists of around 7K lines of C++ code (LOC). The bulk of the code is in the sidecar implementation for parsing RPCs and implementing the Request Limit Protocol. A *sidecar* is a lightweight proxy [19, 67] deployed on the same machine as each microservice. Sidecars intercept all incoming and outgoing RPCs and talk to the microservice over `localhost`. Agent and Ingress share the same codebase. Roshanfer currently supports the gRPC protocol for Unary RPCs by relying on the nghttp2 [75] library for HTTP/2 framing and HPACK [59] (de)compression, but follows a modular architecture, which allows it to be easily extended to other RPC protocols. We build Request Limit Protocol on top of UDP to avoid the overhead of TCP, given that its messages fit in a single packet.

We implement Roshanfer with high performance, low latency, and ease of deployment in mind; hence, we prefer technologies that are widely available in a cloud setting. The sidecar is a multi-threaded application with each worker thread running an event loop on `io_uring` [9] for asynchronous I/O operations (a single worker can process up to 40 KRPS). Worker threads avoid lock-based synchronization and instead rely on atomic operations to share state. `io_uring` enables zero-copy I/O and removes context switch and syscall overheads via shared ring buffers between user and kernel space for pending syscalls and kernel-space polling. Roshanfer leverages adaptive batching [10] for I/O operations within a loop iteration. Roshanfer is publicly available at https://github.com/roshanfer-project.

Credit Queue Policy. Given that Roshanfer supports different request admission policies in the Agent to pick among the upstream requests that have asked for a credit, we implement two sample policies: a Round-Robin (RR) and a Weighted Fair Queue (WFQ) policy. The RR policy alternates among the different endpoints. The WFQ policy, inspired by Aequitas [84], supports three priorities, but it can easily be extended to more levels.

TLA+ Model. We formalize Roshanfer using TLA+. This enables exhaustive exploration of intricate behaviors over all possible traces for a given configuration. We verify the following critical properties, providing important guarantees for the correctness and performance of Roshanfer with respect to the configured limits.

- **Request Bound (DR1)**. *The number of requests active at any given endpoint E is strictly bounded by the sum of the local limits at E and the bounds of E's immediately downstream microservice endpoints.*
- **Deadlock Freedom**. *All requests admitted at Ingress will receive a response, assuming that processing inside the microservices terminates.*
- **Work Conservation**. *Credit requests will only be queued at a microservice S if either S, or some microservice downstream from S, has a number of active requests that meets or exceeds its local limit.*

The *Request Bound* property is critical for avoiding SLO violations, ensuring that there is no unbounded queueing internal to the service. *Deadlock Freedom* ensures that the credit allocation system can never permanently halt the progress of the service, while *Work Conservation* asserts that Roshanfer does not meaningfully interfere with throughput; the Request Limit Protocol will not queue requests unless a microservice is genuinely at capacity. The full TLA+ specification is presented in Supplementary Materials.

6 Evaluation

We evaluate our design and implementation to answer the following questions:

1. Can Roshanfer achieve bounded tail latency and stable goodput? (§6.2)
2. Can Roshanfer eliminate SLO violations during sudden load spikes? (§6.3, §6.4)
3. How does Roshanfer shift queuing across the deployment? (§6.5)
4. How does Roshanfer behave with multiple APIs? (§6.6)
5. Does Roshanfer work in the presence of dynamic call graphs? (§6.7)
6. How does Roshanfer perform in a large-scale real-world service? (§6.8)
7. What is the overhead of running Roshanfer? (§6.9)
8. What is the impact of overcommitment and credit distribution policy? (§6.10)
9. Is Ingress sensitive to values of parameters used in Algorithm 1? (§6.11)

6.1 Evaluation Setup

Testbed: We use 25 c220g2 CloudLab machines to run all experiments (3 workload machines and 22 deployment machines). We use Kubernetes [42] to orchestrate services. We disable Hyper-Threading and set all machines to run at maximum frequency to achieve the best performance.

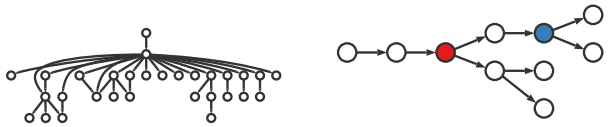
Load Generation and Metric Collection: We use a custom open-loop [71] workload generator implemented in Go

to generate requests according to a Poisson process. We measure all metrics in an end-to-end manner at the workload generator. SLO-violating requests are successful requests with end-to-end latency beyond the SLO. Goodput is the rate of requests finished without an SLO violation. We execute each experiment 3 times and present the average across 3 repetitions, with error bars indicating variance across executions.

Baselines: We compare Roshanfer with two state-of-the-art data plane microservice overload control systems: Rajomon [81] and Dagor [86]. Dagor depends on an approach that drops requests at Ingress based on priorities and the current load in the system. It allows each microservice to further drop requests independently, and only propagates overload information one level back to the upstream microservice. We provide a description of how Rajomon works in §2.1. We use the official implementation of Rajomon [58] and reuse their Dagor implementation [57]. We do not compare against control plane overload control systems, TopFull [56] and Galileo [70], because they take minutes to adjust to overload (for instance, each iteration of Galileo takes 210s and requires multiple iterations to converge), and previous work shows inferior performance compared to our chosen baselines [81]. For fair comparisons and a realistic setting where clients reach the gateway only, we unify the architecture of all baselines. External clients just send their requests without participating in overload control. The gateway receives all requests before forwarding them to the service. This gateway hosts Rajomon's client logic and Dagor's gateway logic. We tune the parameters for Rajomon using Bayesian optimization as suggested in prior work [81], and we report the optimal configurations in Supplementary Materials. For each benchmark, we run the tuner for 50 iterations with 10 initial points.

Benchmarks: We use 4 benchmarks to evaluate Roshanfer. We use the *Hotel Reservation* and *Social Network* benchmarks from DeathStarBench [22]. In addition, we use a benchmark with dynamic call graphs and another large-scale benchmark (30 microservices) generated using DGG [18] based on Alibaba traces [47] to represent real-world microservice-based deployments (shown in Figure 6). The SLO for each API is 5× the P99 latency when the system is unloaded (one request at a time).

Roshanfer Configuration: To avoid HoL blocking delay, overcommitment should be configured based on service time variability. Since first-come, first-served (FCFS) and PS scheduling are optimal for low and high service time variability, respectively [79], overcommitment should be set to 0 and 1, respectively (§4.2). Given that leaf microservices in our benchmarks do not exhibit extreme service time variability, we set overcommitment to 0 in all leaf microservices. We do not prioritize any API in our experiments, for a fair comparison with Rajomon. For Ingress, we set err_d to -0.05, err_i to -0.3, adj_d to 1.5, adj_i to 1, MIN_LIMIT to 4, and α to 0.5 for all experiments.



(a) Large-scale graph (30 microservices)

(b) Dynamic call graph (colored nodes randomly pick their downstream microservices with unknown probabilities)

Figure 6. Benchmarks based on Alibaba traces [18, 47].

6.2 Bounded Tail Latency and Stable Goodput

We start by running an experiment where we vary the offered load to a single service, exceed its capacity, and observe how tail latency and goodput behave. We choose `search-hotel` from *Hotel Reservation* and `compose-post` from *Social Network*, which have the most complex call graphs. The SLOs are set to 28ms and 40ms for `search-hotel` and `compose-post`, respectively.

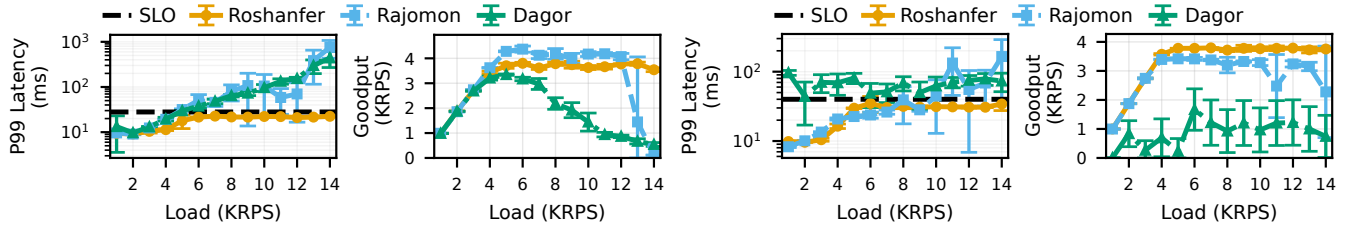
Figure 7 summarizes the experiment results. For both APIs, we observe that Roshanfer always stays within the latency SLO (**RP1**) and maintains a stable goodput (**RP2**) even in cases where the offered load is more than 3× the capacity. Rajomon and Dagor, on the other hand, violate the latency SLO while their goodput decreases as the offered load increases. When the offered load is 2× the maximum goodput, Roshanfer achieves 75% and 20% lower tail latency than Rajomon for `search-hotel` and `compose-post`, respectively. Dagor performs worse than Rajomon across the board.

Roshanfer achieves SLO-capped tail latency because of bounded and controlled queueing delay inside the service and at Ingress, respectively. With Dagor, the lack of end-to-end coordination leads to high queueing delay and SLO violations. Thus, microservices resort to frequent request drops, causing goodput loss. End-to-end price aggregation in Rajomon allows better coordination among microservices, leading to higher goodput and lower tail latency compared to Dagor. However, due to divergence between the test and training environments and late reactions, it violates the SLO (Pitfalls 1 and 3). Thus, microservices resort to dropping requests, which explains goodput decline (Pitfall 2).

6.3 Bounded Tail Latency under Sudden Load Spikes

We now plot the latency and goodput as a function of time for the `search-hotel` API to study sudden load spikes. For the first 2s, the load is 1 KRPS (underloaded), and then increases to 10 KRPS to overload the service. We measure and report tail latency for requests completed within each 200ms window. Figure 8 shows the results.

We see that Roshanfer does not need to react when the load jumps. While enough requests are admitted into the service to sustain goodput (**RP2**), it drops any extra requests



(a) `search-hotel` API **(b)** `compose-post` API

Figure 7. Goodput and tail latency of Roshanfer and baselines under varying levels of overload.

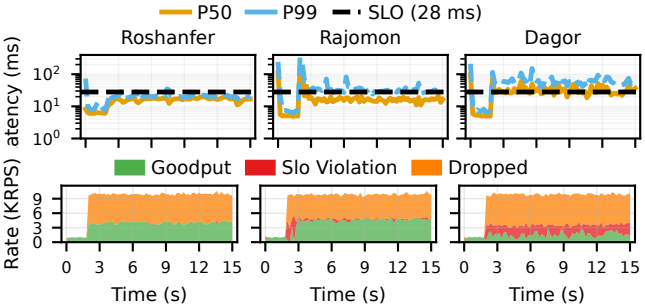
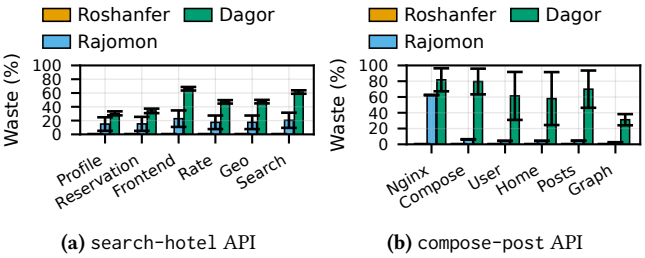


Figure 8. Control dynamics with `search-hotel`.



(a) `search-hotel` API **(b)** `compose-post` API

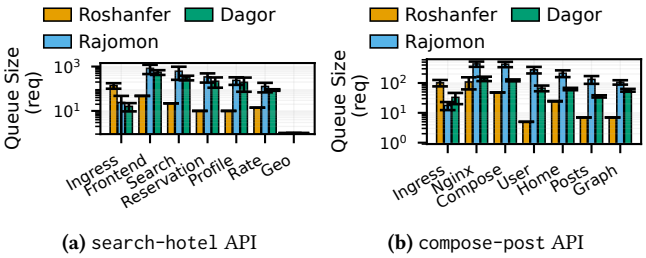
Figure 9. Resource waste across overload control systems.

at Ingress to avoid SLO violations (**RP1**). Both Dagor and Rajomon take time to adjust to load changes, while suffering from goodput loss and SLO violations during overload (Pitfall 1). Dagor is unable to control queueing delay throughout the experiment due to a lack of end-to-end coordination. In contrast, Rajomon achieves lower tail latency but still suffers from SLO violations. Similar results for the `compose-post` API are reported in Supplementary Materials.

6.4 Minimal Resource Waste

For the previous experimental setting, we also study resource utilization for the two APIs during overload. We focus on resource waste, i.e., the CPU resources wasted on requests that eventually violate their SLOs or that are dropped by one of the downstream microservices.

Figure 9 shows the percentage of resource waste in every microservice of the two APIs. Roshanfer only drops requests at Ingress where they have not consumed any resources yet (**RP3**). In addition, few requests have response times higher than the SLO (less than 1% for an SLO defined at P99, hence no overall SLO violation). Therefore, the overall resource



(a) `search-hotel` API **(b)** `compose-post` API

Figure 10. Queuing under overload.

waste under Roshanfer is near zero. On average, Rajomon and Dagor waste up to 63% and 81% of resources, respectively. Rajomon performs better than Dagor as it tries to drop requests as early as possible in the call graph through price propagation. Resource waste results can also be interpreted as duals of Figure 7. For example, Figure 7b shows that Dagor's goodput can decline to less than 1 KRPS, which is 75% lower than the 4 KRPS that Roshanfer achieves, showing that the rest of the resource utilization has been wasted.

6.5 Bounded Queueing at Services

For the experiments in §6.3 and §6.4, we study the maximum queuing during overload and how it is distributed across the service to understand how Roshanfer can shift queues from microservices to Ingress. Figure 10 shows the maximum queue sizes in microservices for both benchmarks.

For Roshanfer, we see that Ingress has the biggest queue as Request Limit Protocol pushes all extra requests to Ingress. On the other hand, all microservices have smaller queues compared to both Rajomon and Dagor. Dagor and Rajomon, however, allow up to 9× (in the `frontend` microservice) higher queuing at microservices compared to Roshanfer, with less queuing at Ingress. This shows that both baselines allow higher queueing inside the service, which explains their SLO violations.

6.6 Performance With Multiple Concurrent APIs

Overload control in the presence of multiple APIs is much more challenging than the single API case because i) the SLO for each API is defined based on unloaded P99 latency without the presence of other APIs (a fixed target), and ii) the call graphs of APIs can share endpoints, creating competition among microservices during overload. We run a

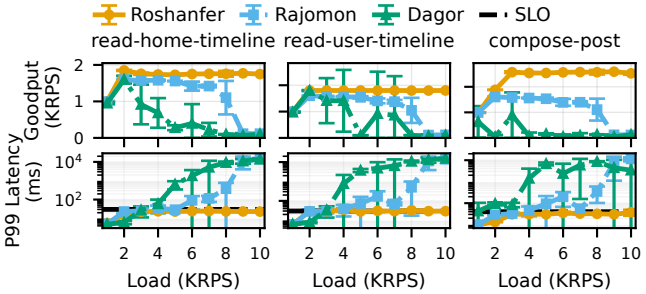


Figure 11. Tail latency and goodput with 3 APIs.

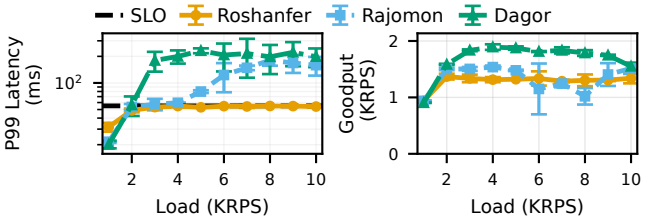


Figure 12. Tail latency and goodput with dynamic call graphs.

similar set of experiments, but now exercise three APIs of the *Social Network* service at the same time. The microservice `post`, for example, is shared by all three external APIs. An overload in one API can potentially increase processing and queuing delay for other APIs. We consider equal load levels for the `read-user-timeline`, `read-home-timeline`, and `compose-post` APIs of the *Social Network* benchmark and no request priorities. We target the worst-case scenario where all APIs increase load at the same time. Our goal is to explore how overload controllers deal with multiple APIs with different SLOs. For brevity, we only show experiments with *Social Network* here, and corresponding experiments on *Hotel Reservation* appear in the Supplementary Materials.

Figure 11 shows tail latency and goodput results with varying offered load. Again, Roshanfer bounds the tail latency to the corresponding SLOs under any offered load. In terms of goodput, Roshanfer also achieves higher levels compared to the baselines. Both Rajomon and Dagor start to lose goodput after 3 KRPS. Although Rajomon avoids SLO violations at low loads, it quickly starts deteriorating as the load increases.

Roshanfer is able to deliver performance resilience with multiple concurrent APIs because credits are issued at RPC endpoint granularity. Moreover, requests for different APIs are queued at independent Ingress instances to avoid HoL blocking delays, while the Go scheduler in each microservice efficiently and fairly multiplexes resources among the different RPC endpoints. This allows Roshanfer to achieve high goodput and low tail latency for all APIs. Dagor and Rajomon fail to avoid goodput loss and SLO violations across APIs for the same reasons as in the single API case.

6.7 Resilience to Dynamic Call Graphs

To evaluate performance with dynamic call graphs, we use another benchmark where requests for a single API can have

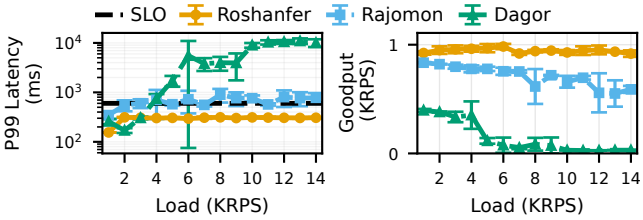


Figure 13. Tail latency and goodput in the Alibaba benchmark.

three possible call graphs. The service times of these call graphs have a coefficient of variation (CV) of 66% to represent real-world services [18, 47], while the SLO is fixed to 55ms. Figure 12 shows P99 latency and goodput under increasing offered loads. Roshanfer again consistently controls the tail latency regardless of the load and the dynamism in the call graph (**RP4**). In contrast, both Dagor and Rajomon start violating the SLO at 2 KRPS offered load and get worse as the load increases (Pitfall 2). Higher goodput with Dagor and Rajomon is expected despite their lack of support for dynamic call graphs. These systems rely on queueing delay as the overload signal. However, with dynamic call graphs, this signal is neither accurate nor stable, and hence they are less efficient at throttling the excess load. As they admit more requests, resource-intensive call graphs increase the tail latency, while lightweight call graphs drive throughput.

This tension reveals the innate trade-off in dynamic call graphs. Controlling tail latency requires throttling when the heaviest call graph is overloaded, leading to lower goodput. Conversely, admitting more requests violates the SLO to achieve higher goodput. Roshanfer adopts the first option as it is required for performance resilience (**RP1**).

6.8 Roshanfer at Scale

In this experiment, we evaluate Roshanfer in a large-scale benchmark based on Alibaba traces (shown in Figure 6a). The SLO of this service is 600ms, and we use the same setup as in the previous experiment. Results are shown in Figure 13. As expected, Rajomon outperforms Dagor in both tail latency and goodput because of end-to-end coordination. Despite this, Rajomon starts violating the SLO when the offered load reaches 3 KRPS. Roshanfer keeps the tail latency under 300ms (50% of the SLO) while achieving higher goodput compared to baselines. The results demonstrate Roshanfer's ability to operate in large-scale real-world deployments.

6.9 Roshanfer's Overhead

We evaluate the cost of Roshanfer and of providing performance resilience in end-to-end deployments. As a baseline, we run the *Hotel Reservation* benchmark with the `search-hotel` API and allow microservices to communicate directly without sidecars or any form of overload control. Figure 14 shows both median and P99 latency for Roshanfer and Plain (no overload control) deployments. Roshanfer performs similarly to Plain, with 1.2ms higher P50 latency and 1ms lower

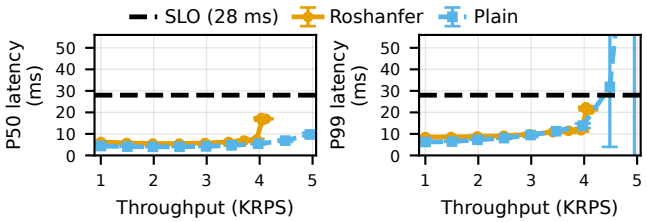
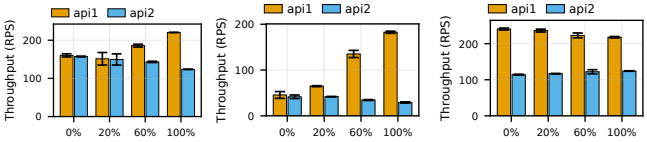


Figure 14. Roshanfer's overhead.



(a) 1:2 ratio, equal priority. **(b)** 1:10 ratio, equal priority. **(c)** 1:2 ratio with 2:1 priority.

Figure 15. Impact of overcommitment.

P99 latency at 3.5 KRPS offered load. The results demonstrate Roshanfer's low latency overhead from sidecar request processing and the Request Limit Protocol. The *ms*-level service times almost completely mask Roshanfer's μs-scale communication and processing delays, even for microservices with multiple calls in their call graph.

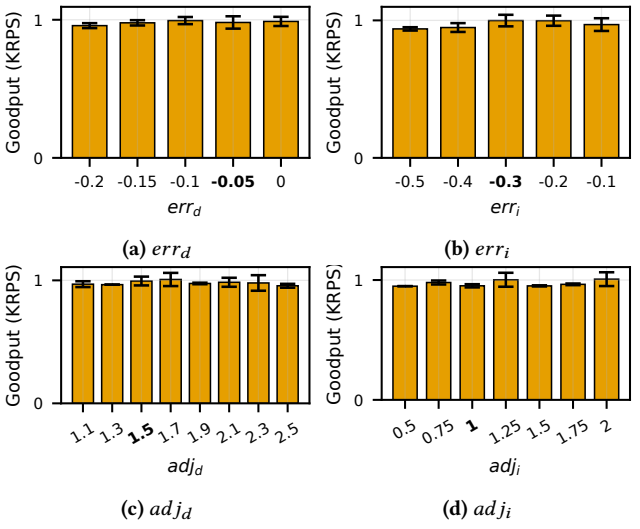
6.10 Impact of Overcommitment and Request Admission Policy

We isolate how the global limit trades off with overcommitment using one busy-looping leaf microservice that exposes two endpoints. For a 1:2 service time ratio (Figure 15a), goodput is equal at low overcommitment and converges to a fair CPU split as overcommitment grows; for 1:10 (Figure 15b), the slower endpoint initially takes most CPU under low overcommitment, yet both endpoints again approach fair share as overcommitment rises because more short requests finish within their SLO. Thus, for leaf microservices, higher overcommitment emulates PS, whereas lower overcommitment emulates run-to-completion FCFS.

To study the impact of request admission policy, we repeat the experiment with a 1:2 service time ratio but with a 2:1 credit distribution ratio using a simple Weighted Round Robin (WRR) policy. Figure 15c shows the results. In contrast to Figure 15a, which depends on a Round-Robin policy, with WRR, the endpoints can achieve a fair share of CPU resources, even with 0 overcommitment. Therefore, the request admission policy should be considered jointly with overcommitment when setting the global limit.

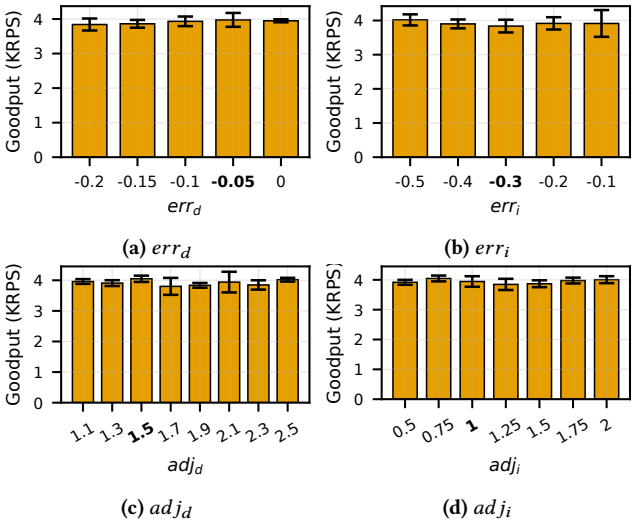
6.11 Ingress Parameter Sensitivity

We evaluate the sensitivity of Ingress to the values of its main parameters used in Algorithm 1. We sweep each parameter while keeping the others fixed at the values reported in §6.1. For each value, we overload the benchmark and measure the achieved goodput. We repeat this process for our large-scale benchmark based on Alibaba traces and for the *Hotel Reservation* benchmark. We select the sweep range based on the semantics of the parameters. For example, err_d should be below 0 because the desired P99 latency should be less than the SLO. Furthermore, err_i should be smaller than err_d so that there is a "do nothing" region in the algorithm to prevent the maximum queue size from oscillating.



(a) err_d **(b)** err_i **(c)** adj_d **(d)** adj_i

Figure 16. Ingress parameter sensitivity in the Alibaba benchmark.



(a) err_d **(b)** err_i **(c)** adj_d **(d)** adj_i

Figure 17. Ingress parameter sensitivity in the Hotel Reservation benchmark (`search-hotel` API).

Figure 16 and Figure 17 show goodput versus different parameter values for the Alibaba and *Hotel Reservation* benchmarks, respectively. The bold labels represent the default values presented in §6.1. The results show that, regardless of the parameter values, goodput stays at 1 KRPS and 4 KRPS for the Alibaba and *Hotel Reservation* benchmarks, respectively. These goodput values agree with the maximum achieved goodput in §6.2 and §6.8. Within each benchmark, the results show the resilience of Ingress with respect to its parameters. Across the benchmarks, the results show that a

fixed set of parameter values works well and does not require re-tuning.

The resilience of Ingress is due to bounded queues at the service. Since extra queueing can only happen at Ingress, Algorithm 1 can use its queue utilization as a proxy for the offered load. Given this knowledge, the maximum queue size is increased if and only if the end-to-end latency is below the SLO and the offered load is high enough; otherwise, it is decreased. Such a design avoids an unnecessary increase in the maximum queue size, making Ingress insensitive to its parameters.

7 Discussion

Limitations. Roshanfer calculates endpoint limits under the following assumptions: First, endpoint limits only consider CPU bottlenecks, as prior work shows CPU to be the main resource affecting performance of microservices [76, 83, 85]. Roshanfer can be adapted to other resource bottlenecks, e.g., disk or GPU, given that the formulas introduced in §4 are not CPU-specific. The formulas abstract the service rate $\frac{1}{s_i}$ and the level of concurrency N_{cores}, metrics that also correspond to other resources. Yet, we leave the exploration of multi-resource overload control to future work. Second, active request counting (§3.2) relies on the assumption of synchronous unary RPCs in which each RPC is single-threaded, which is common [4, 22, 73]. In the presence of multi-threaded requests, one can adjust the level of concurrency N_{cores} accordingly to support them.

Although we acknowledge the potential for lock contention [14] to become a bottleneck for certain highly concurrent applications, we believe that this is a resource allocation problem and not an overload control one. Roshanfer assumes that a microservice can fully utilize its allocated resources and builds on this assumption. If a microservice cannot do so due to frequent synchronization, the control plane should provision it with fewer resources, which is an orthogonal problem.

Message Queues. Roshanfer assumes synchronous nested RPCs. Asynchronous messaging through brokers such as Kafka [7] and RabbitMQ [66] violates this assumption and can introduce unbounded queueing that degrades performance resilience. Message queues split call graphs into subgraphs with nested synchronous RPCs within each one. Each subgraph has its own SLO, and independent instances of Roshanfer can be deployed within each subgraph. Those independent Roshanfer instances can therefore retain overload control for each subgraph.

Dynamic Call Graphs and Autoscaling. With dynamic call graphs, there is a trade-off between avoiding SLO violations and achieving high resource utilization (§6.7). Roshanfer's main goal is performance resilience, so it favors avoiding SLO violations. The resulting underutilization stems from resource imbalances across paths that reach capacity at different loads. Autoscalers can periodically rebalance resources between bottleneck and underutilized paths to mitigate this issue. A bottleneck path is one where all of its microservices have filled their bounded queues and the corresponding queue at Ingress is non-empty. Thus, Roshanfer can streamline autoscaler designs by providing strong bottleneck signals.

Handling Failures. Roshanfer can be used to improve failure handling. First, Roshanfer automatically self-clocks when a target microservice suffers a fail-slow failure [46]. This is because a slower microservice will naturally drain its queue and respond to credit requests more slowly. Second, in the case of a fail-stop failure, Roshanfer caps the blast radius to the number of requests allowed by the endpoint limit. A timeout-based mechanism, similar to Envoy [19], can deal with the lost requests by resending them and updating the active request count. Roshanfer relies on the assumption of timely `Credit Request` and `Credit Grant` messages. Although this assumption can be violated in congested and lossy networks, these messages can be sent in a higher priority class to guarantee their delivery [28, 52, 60].

Replication. Roshanfer can be extended to support replication. Each replica of a microservice can have its own Agent, which maintains per-replica endpoint limits and active request counts. Client Agents can use available load balancing policies (e.g., round-robin) to pick the target replica. We leave a full exploration of replicated microservices with Roshanfer to future work.

8 Related Work

Overload Control for Microservice-Based Deployments. At the control plane, there are three types of systems for dealing with overload in microservice-based deployments: i) Admission rate controllers [56, 70, 74]: adjust the admission rate of individual APIs at the gateway, with TopFull [56] and Galileo [70] using reinforcement learning (RL), ii) Service quality controllers [1, 41, 49]: gracefully degrade the service quality to serve higher throughput but with lower quality, iii) Auto-scalers [11, 29, 48, 50, 64, 65, 68, 76, 77, 83, 85]: automatically add more resources to match the offered load. Although these systems are necessary for overloads lasting minutes to days, they are not fast enough to react to load spikes at millisecond timescales. For instance, each iteration in Galileo [70] takes 210s and assumes a stable workload between iterations.

Data plane overload controllers [81, 86] are designed to target short-term overloads. However, their reactive nature prevents them from achieving performance resilience during sudden load spikes. Specifically, Rajomon [81] relies on queuing-delay thresholds for overload detection and signaling. However, our evaluations show this slow reaction can

take seconds, leading to SLO violations, while individual microservices resort to request drops, creating resource waste. In addition, they rely on simplifying assumptions such as static call graphs and a pre-training phase, which are unrealistic for real-world deployments.

Service meshes such as Istio [34] and Cilium [15] expose rate limiting [37] and circuit breaking [35], but these mechanisms do not provide performance resilience. They combine a strictly local, per-microservice view with sender-driven control, which makes global tuning brittle across large microservice graphs and still allows fan-in to overload a downstream bottleneck microservice.

Overload Control for Monolithic Applications. Handling overload in monolithic applications is a smaller challenge because it does not require end-to-end coordination. SEDA [78] is one of the seminal works in this space. Breakwater [13] improves on SEDA by combining client-side rate limiting with server-side queue management. Similar to Roshanfer, Breakwater uses a credit-based approach to limit the number of admitted requests based on the application's queuing delay under high load. Protego [14] tackles flaws in both SEDA and Breakwater when RPCs face lock contention. Recently, Atropos [32] identifies slow requests as the root cause of overload and cancels them to prevent overload and open up capacity for faster requests.

Network Congestion Control. Roshanfer and related research on overload control borrow ideas from congestion control algorithms [2, 3, 8, 28, 44, 51, 52, 60], especially receiver-driven credit schemes [28, 52, 60]. Yet, they deal with different bottlenecks. Notably, Aequitas [84] shares similar goals with Roshanfer, but focuses entirely on the transport layer, which is orthogonal to Roshanfer.

Stream Processing Systems. Apache Flink [5] and similar stream engines use credit-based flow control under overload, but their operator graphs are mostly static chains, so credits identify overloaded operators and limit buffer bloat [6] rather than enforce tight bounds across the graph. Roshanfer instead targets microservices: its credits share compute among many upstream microservices under dynamic call graphs and volatile workloads.

End-host RPC Scheduling. Many systems improve interactive performance through host-local RPC scheduling [17, 20, 21, 33, 39, 40, 45, 54, 61, 63], multiplexing CPUs among co-located applications on one machine rather than tackling distributed microservice-based deployments. Caladan [21] reduces co-location interference but not overload control; Coresync [55] argues that scheduling and overload control should be co-designed, yet remains confined to monolithic applications. Roshanfer caps per-microservice queuing while preserving headroom for schedulers to multiplex RPC endpoints with heterogeneous service times and SLOs.

9 Conclusion

We present Roshanfer, a system that offers performance resilience even under sudden, short load spikes in microservices. We take a proactive approach to prevent overload before it happens, as opposed to reacting to it, which leads to performance degradation. Through extensive evaluation, we show how Roshanfer meets its design goals across a variety of realistic benchmarks and workloads while preserving ease of deployment.

Ethics. This work does not raise any ethical issues.

Acknowledgments

We thank our shepherd, our anonymous EuroSys reviewers, and the members of the LSDS group at Imperial for their constructive comments and suggestions. This work is supported by an ERC Starting Grant for the CloudNG project (grant agreement ID 101220079).

References

[1] Kapil Agrawal and Sangeetha Abdu Jyothi. 2025. Cooperative Graceful Degradation in Containerized Clouds. In Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1. ACM, Rotterdam Netherlands, 214–232. doi:10.1145/3669940.3707244

[2] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data center TCP (DCTCP). In Proceedings of the ACM SIGCOMM 2010 Conference (New Delhi, India) (SIGCOMM '10). Association for Computing Machinery, New York, NY, USA, 63–74. doi:10.1145/1851182.1851192

[3] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. 2013. pFabric: minimal near-optimal datacenter transport. In Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM (Hong Kong, China) (SIGCOMM '13). Association for Computing Machinery, New York, NY, USA, 435–446. doi:10.1145/2486001.2486031

[4] Vaastav Anand, Deepak Garg, Antoine Kaufmann, and Jonathan Mace. 2023. Blueprint: A Toolchain for Highly-Reconfigurable Microservice Applications. In Proceedings of the 29th Symposium on Operating Systems Principles. ACM, Koblenz Germany, 482–497. doi:10.1145/3600006.3613138

[5] Apache Flink. 2019. A Deep-Dive into Flink's Network Stack. https://flink.apache.org/2019/06/05/a-deep-dive-into-flinks-network-stack/#credit-based-flow-control Accessed: 2025-09-13.

[6] Apache Flink. 2025. Network Memory Tuning Guide. https://nightlies.apache.org/flink/flink-docs-master/docs/deployment/memory/network_mem_tuning/ Accessed: 2025-09-15.

[7] Apache Software Foundation. 2025. Apache Kafka. https://kafka.apache.org/ Accessed: 2025-09-15.

[8] Serhat Arslan, Yuliang Li, Gautam Kumar, and Nandita Dukkipati. 2023. Bolt: Sub-RTT Congestion Control for Ultra-Low Latency. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, 219–236. https://www.usenix.org/conference/nsdi23/presentation/arslan

[9] Jens Axboe. 2025. liburing. https://github.com/axboe/liburing Accessed: 2025-09-15.

[10] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In 11th USENIX

Symposium on Operating Systems Design and Implementation (OSDI 14). USENIX Association, Broomfield, CO, 49–65. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[11] Romil Bhardwaj, Kirthevasan Kandasamy, Asim Biswal, Wenshuo Guo, Benjamin Hindman, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2023. Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 623–643.

[12] Jingrong Chen, Yongji Wu, Shihan Lin, Yechen Xu, Xinhao Kong, Thomas Anderson, Matthew Lentz, Xiaowei Yang, and Danyang Zhuo. 2023. Remote Procedure Call as a Managed System Service. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, 141–159. https://www.usenix.org/conference/nsdi23/presentation/chen-jingrong

[13] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2020. Overload Control for µs-scale RPCs with Breakwater. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, Virtual Event, 299–314. https://www.usenix.org/conference/osdi20/presentation/cho

[14] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. 2023. Protego: Overload Control for Applications with Unpredictable Lock Contention. In 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). USENIX Association, Boston, MA, 725–738. https://www.usenix.org/conference/nsdi23/presentation/cho-inho

[15] Cilium. 2025. Cilium. https://docs.cilium.io/en/stable/overview/component-overview/ Accessed: 2025-09-15.

[16] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. Commun. ACM 56, 2 (2013), 74–80. doi:10.1145/2408776.2408794

[17] Henri Maxime Demoulin, Joshua Fried, Isaac Pedisich, Marios Kogias, Boon Thau Loo, Linh Thi Xuan Phan, and Irene Zhang. 2021. When Idling is Ideal: Optimizing Tail-Latency for Heavy-Tailed Datacenter Workloads with Perséphone. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (Virtual Event, Germany) (SOSP '21). Association for Computing Machinery, New York, NY, USA, 621–637. doi:10.1145/3477132.3483571

[18] Fanrong Du, Jiuchen Shi, Quan Chen, Pu Pang, Li Li, and Minyi Guo. 2025. Generating Microservice Graphs with Production Characteristics for Efficient Resource Scaling. In Proceedings of the 39th ACM International Conference on Supercomputing. ACM, Salt Lake City USA, 895–910. doi:10.1145/3721145.3725761

[19] Envoy Project. 2025. Envoy Proxy. https://www.envoyproxy.io/ Accessed: 2025-09-15.

[20] Joshua Fried, Gohar Irfan Chaudhry, Enrique Saurez, Esha Choukse, Í nigo Goiri, Sameh Elnikety, Rodrigo Fonseca, and Adam Belay. 2024. Making Kernel Bypass Practical for the Cloud with Junction. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association, Santa Clara, CA, 55–73. https://www.usenix.org/conference/nsdi24/presentation/fried

[21] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating Interference at Microsecond Timescales. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, Virtual Event, 281–297. https://www.usenix.org/conference/osdi20/presentation/fried

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 2019. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems. In Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems. ACM, Providence RI USA, 3–18. doi:10.1145/3297858.3304013

[23] Google. 2018. Example Error Budget Policy. In The Site Reliability Workbook, Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy (Eds.). O'Reilly Media, Sebastopol, CA. https://sre.google/workbook/error-budget-policy/ Accessed: 2026-05-13.

[24] Google. 2023. Why Does Speed Matter? https://web.dev/learn/performance/why-speed-matters Accessed: 2026-05-13.

[25] Steven D. Gribble. 2001. Robustness in Complex Systems. In Proceedings Eighth Workshop on Hot Topics in Operating Systems. IEEE, Elmau, Germany, 21–26. doi:10.1109/HOTOS.2001.990056

[26] Linsong Guo, Danial Zuberi, Tal Garfinkel, and Amy Ousterhout. 2025. The Benefits and Limitations of User Interrupts for Preemptive Userspace Scheduling. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). USENIX Association, Philadelphia, PA, 1015–1032. https://www.usenix.org/conference/nsdi25/presentation/guo

[27] Einas Haddad. 2015. Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow. https://www.uber.com/blog/service-oriented-architecture/ Accessed: 2025-09-15.

[28] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In Proceedings of the Conference of the ACM Special Interest Group on Data Communication (Los Angeles, CA, USA) (SIGCOMM '17). Association for Computing Machinery, New York, NY, USA, 29–42. doi:10.1145/3098822.3098825

[29] Rubaba Hasan, Timothy Zhu, and Bhuvan Urgaonkar. 2024. AutoBurst: Autoscaling Burstable Instances for Cost-effective Latency SLOs. In Proceedings of the 2024 ACM Symposium on Cloud Computing (Redmond, WA, USA) (SoCC '24). Association for Computing Machinery, New York, NY, USA, 243–258. doi:10.1145/3698038.3698530

[30] Mazdak Hashemi. 2017. The Infrastructure Behind Twitter: Scale. https://blog.x.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale Accessed: 2025-09-15.

[31] Tamás Hauer, Philipp Hoffmann, John Lunney, Dan Ardelean, and Amer Diwan. 2020. Meaningful Availability. In 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). USENIX Association, Santa Clara, CA, 545–557. https://www.usenix.org/conference/nsdi20/presentation/hauer

[32] Yigong Hu, Zeyin Zhang, Yicheng Liu, Yile Gu, Shuangyu Lei, Baris Kasikci, and Peng Huang. 2025. Mitigating Application Resource Overload with Targeted Task Cancellation. In Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles. ACM, Lotte Hotel World Seoul Republic of Korea, 270–285. doi:10.1145/3731569.3764835

[33] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (Virtual Event, Germany) (SOSP '21). Association for Computing Machinery, New York, NY, USA, 588–604. doi:10.1145/3477132.3483542

[34] Istio. 2025. Istio. https://istio.io/ Accessed: 2025-09-15.

[35] Istio. 2025. Istio Circuit Breaking. https://istio.io/latest/docs/tasks/traffic-management/circuit-breaking/ Accessed: 2025-09-15.

[36] Istio. 2025. Istio Gateway. https://istio.io/latest/docs/concepts/traffic-management/#gateways Accessed: 2025-09-15.

[37] Istio Authors. 2025. Istio Rate Limiting. https://istio.io/latest/docs/tasks/policy-enforcement/rate-limit/ Accessed: 2025-09-15.

[38] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving Microsecond-Scale Tail Latency Efficiently with Approximate Optimal Scheduling. In Proceedings of the 29th Symposium on Operating Systems Principles. ACM, Koblenz Germany, 466–481. doi:10.1145/3600006.3613136

[39] Yuekai Jia, Kaifu Tian, Yuyang You, Yu Chen, and Kang Chen. 2024. Skyloft: A General High-Efficient Scheduling Framework in User Space. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (Austin, TX, USA) (SOSP '24). Association for Computing Machinery, New York, NY, USA, 265–279. doi:10.1145/3694715.3695973

[40] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-Defined Scheduling Across the Stack. In Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (Virtual Event, Germany) (SOSP '21). Association for Computing Machinery, New York, NY, USA, 605–620. doi:10.1145/3477132.3483548

[41] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building More Robust Cloud Applications. In Proceedings of the 36th International Conference on Software Engineering. ACM, Hyderabad India, 700–711. doi:10.1145/2568225.2568227

[42] Kubernetes. 2025. Kubernetes. https://kubernetes.io/ Accessed: 2025-09-15.

[43] Kubernetes SIG Network. 2025. Gateway API. https://gateway-api.sigs.k8s.io/ Accessed: 2025-09-15.

[44] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (Virtual Event, USA) (SIGCOMM '20). Association for Computing Machinery, New York, NY, USA, 514–528. doi:10.1145/3387514.3406591

[45] Jiazhen Lin, Youmin Chen, Shiwei Gao, and Youyou Lu. 2024. Fast Core Scheduling with Userspace Process Abstraction. In Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (Austin, TX, USA) (SOSP '24). Association for Computing Machinery, New York, NY, USA, 280–295. doi:10.1145/3694715.3695976

[46] Ruiming Lu, Yunchi Lu, Yuxuan Jiang, Guangtao Xue, and Peng Huang. 2025. One-Size-Fits-None: Understanding and Enhancing Slow-Fault Tolerance in Modern Distributed Systems. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). USENIX Association, Philadelphia, PA, 359–378.

[47] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. 2021. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In Proceedings of the ACM Symposium on Cloud Computing. ACM, Seattle WA USA, 412–426. doi:10.1145/3472883.3487003

[48] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. 2022. Erms: Efficient Resource Management for Shared Microservices with SLA Guarantees. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (Vancouver, BC, Canada) (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 62–77. doi:10.1145/3567955.3567964

[49] Justin J. Meza, Thote Gowda, Ahmed Eid, Tomiwa Ijaware, Dmitry Chernyshev, Yi Yu, Md Nazim Uddin, Rohan Das, Chad Nachiappan, Sari Tran, Shuyang Shi, Tina Luo, David Ke Hong, Sankaralingam Panneerselvam, Hans Ragas, Svetlin Manavski, Weidong Wang, and Francois Richard. 2023. Defcon: Preventing Overload with Graceful Feature Degradation. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 607–622.

[50] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F. Wenisch. 2021. Parslo: A Gradient Descent-based Approach for Near-optimal Partial SLO Allotment in Microservices. In Proceedings of the ACM Symposium on Cloud Computing (Seattle, WA, USA) (SoCC '21). Association for Computing Machinery, New York, NY, USA, 442–457. doi:10.1145/3472883.3486985

[51] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication (London, United Kingdom) (SIGCOMM '15). Association for Computing Machinery, New York, NY, USA, 537–550. doi:10.1145/2785956.2787510

[52] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. 2018. 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 (SIGCOMM '18). ACM, Budapest, Hungary, 221–235. doi:10.1145/3230543.3230564

[53] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013. Scaling Memcache at Facebook. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13). USENIX Association, Lombard, IL, 385–398. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/nishtala

[54] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association, Boston, MA, 361–378. https://www.usenix.org/conference/nsdi19/presentation/ousterhout

[55] Bhaskar Pardeshi, Eric Stuhr, and Ahmed Saeed. 2025. CoreSync: A Protocol for Joint Core Scheduling and Overload Control of μs-Scale Tasks. In 2025 IEEE 33rd International Conference on Network Protocols (ICNP). IEEE, Seoul, Republic of Korea, 1–11. doi:10.1109/ICNP65844.2025.11192365

[56] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. 2024. TopFull: An Adaptive Top-Down Overload Control for SLO-Oriented Microservices. In Proceedings of the ACM SIGCOMM 2024 Conference. ACM, Sydney NSW Australia, 876–890. doi:10.1145/3651890.3672253

[57] PennSail. 2025. Dagor gRPC. https://github.com/pennsail/dagor-grpc Accessed: 2025-09-15.

[58] PennSail. 2025. Rajomon. https://github.com/pennsail/rajomon Accessed: 2025-09-15.

[59] Roberto Peon and Hervé Ruellan. 2015. HPACK: Header Compression for HTTP/2. RFC 7541. doi:10.17487/RFC7541

[60] Konstantinos Prasopoulos, Ryan Kosta, Edouard Bugnion, and Marios Kogias. 2025. SIRD: A Sender-Informed, Receiver-Driven Datacenter Transport Protocol. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). USENIX Association, Philadelphia, PA, 451–471. https://www.usenix.org/conference/nsdi25/presentation/prasopoulos

[61] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In Proceedings of the 26th Symposium on Operating Systems Principles (Shanghai, China) (SOSP '17). Association for Computing Machinery, New York, NY, USA, 325–341. doi:10.1145/3132747.3132780

[62] Mia Primorac, Katerina Argyraki, and Edouard Bugnion. 2021. When to Hedge in Interactive Services. In 18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21). USENIX Association, Virtual Event, 373–387. https://www.usenix.org/conference/nsdi21/presentation/primorac

[63] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. 2018. Arachne: core-aware thread management. In Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (Carlsbad, CA, USA) (OSDI'18). USENIX Association, USA, 145–160.

[64] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, Virtual Event, 805–825.

[65] Haoran Qiu, Weichao Mao, Chen Wang, Hubertus Franke, Alaa Youssef, Zbigniew T. Kalbarczyk, Tamer Başar, and Ravishankar K. Iyer. 2023. AWARE: Automate Workload Autoscaling with Reinforcement Learning in Production Cloud Systems. In 2023 USENIX Annual Technical Conference (USENIX ATC 23). USENIX Association, Boston, MA, 387–402.

[66] RabbitMQ. 2025. RabbitMQ. https://www.rabbitmq.com/ Accessed: 2025-09-15.

[67] Will Reese. 2008. Nginx: the high-performance web server and reverse proxy. Linux Journal 2008, 173 (2008), 2.

[68] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, and John Wilkes. 2020. Autopilot: workload autoscaling at Google. In Proceedings of the Fifteenth European Conference on Computer Systems (Heraklion, Greece) (EuroSys '20). Association for Computing Machinery, New York, NY, USA, Article 16, 16 pages. doi:10.1145/3342195.3387524

[69] Harshit Saokar, Soteris Demetriou, Nick Magerko, Max Kontorovich, Josh Kirstein, Margot Leibold, Dimitrios Skarlatos, Hitesh Khandelwal, and Chunqiang Tang. 2023. ServiceRouter: Hyperscale and Minimal Cost Service Mesh at Meta. In 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23). USENIX Association, Boston, MA, 969–985. https://www.usenix.org/conference/osdi23/presentation/saokar

[70] Divyanshu Saxena, Gaurav Vipat, Jiaxin Lin, Jingbo Wang, Isil Dillig, Sanjay Shakkottai, and Aditya Akella. 2026. Towards Performance Robustness for Microservices. In 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26). USENIX Association, Renton, WA, 93–112. https://www.usenix.org/conference/nsdi26/presentation/saxena

[71] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In 3rd Symposium on Networked Systems Design & Implementation (NSDI 06). USENIX Association, San Jose, CA, 239–252. https://www.usenix.org/conference/nsdi-06/open-versus-closed-cautionary-tale

[72] Korakit Seemakhupt, Brent E. Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C. Snoeren, Arvind Krishnamurthy, David E. Culler, and Henry M. Levy. 2023. A Cloud-Scale Characterization of Remote Procedure Calls. In Proceedings of the 29th Symposium on Operating Systems Principles. ACM, Koblenz Germany, 498–514. doi:10.1145/3600006.3613156

[73] Sockshop. 2023. Sock Shop : A Microservice Demo Application. https://github.com/ocp-power-demos/sock-shop-demo Accessed: 2026-05-13.

[74] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. 2017. Distributed Resource Management across Process Boundaries. In Proceedings of the 2017 Symposium on Cloud Computing. ACM, Santa Clara California, 611–623. doi:10.1145/3127479.3132020

[75] Tatsuhiro Tsujikawa. 2025. nghttp2. https://github.com/nghttp2/nghttp2 Accessed: 2025-09-15.

[76] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. 2024. Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association, Santa Clara, CA, 149–165.

[77] Ziliang Wang, Shiyi Zhu, Jianguo Li, Wei Jiang, K. K. Ramakrishnan, Yangfei Zheng, Meng Yan, Xiaohong Zhang, and Alex X. Liu. 2022. DeepScaling: microservices autoscaling for stable CPU utilization in large scale cloud systems. In Proceedings of the 13th Symposium on Cloud Computing (San Francisco, California) (SoCC '22). Association for Computing Machinery, New York, NY, USA, 16–30. doi:10.1145/3542929.3563469

[78] Matt Welsh, David Culler, and Eric Brewer. 2001. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01). ACM, Banff, Canada, 230–243. doi:10.1145/502034.502057

[79] Adam Wierman and Bert Zwart. 2012. Is Tail-Optimal Scheduling Possible? Operations Research 60, 5 (2012), 1249–1257. doi:10.1287/opre.1120.1086

[80] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. 2024. Load Is Not What You Should Balance: Introducing Prequal. In 21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24). USENIX Association, Santa Clara, CA, 1285–1299.

[81] Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, Justin Qiu, Henri Maxime Demoulin, Konstantinos Kallas, and Benjamin C. Lee. 2025. Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices. In 22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25). USENIX Association, Philadelphia, PA, 21–36. https://www.usenix.org/conference/nsdi25/presentation/xing

[82] David Yanacek. 2019. Using Load Shedding to Avoid Overload. Amazon Builders' Library. https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload/ Accessed: 2026-05-13.

[83] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. 2021. Sinan: ML-based and QoS-aware Resource Management for Cloud Microservices. In Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (Virtual USA, 2021-04-19). ACM, Virtual Event, 167–181. doi:10.1145/3445814.3446693

[84] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: Admission Control for Performance-Critical RPCs in Datacenters. In Proceedings of the ACM SIGCOMM 2022 Conference. ACM, Amsterdam Netherlands, 1–18. doi:10.1145/3544216.3544271

[85] Yanqi Zhang, Zhuangzhuang Zhou, Sameh Elnikety, and Christina Delimitrou. 2024. Ursa: Lightweight Resource Management for Cloud-Native Microservices. In 2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA). IEEE, Edinburgh, United Kingdom, 954–969. doi:10.1109/HPCA57654.2024.00077

[86] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In Proceedings of the ACM Symposium on Cloud Computing. ACM, Carlsbad CA USA, 149–161. doi:10.1145/3267809.3267823