usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices

Jiali Xing, Akis Giannoukos, Paul Loh, Shuyue Wang, and Justin Qiu, *University of Pennsylvania;* Henri Maxime Demoulin, *DBOS, Inc;* Konstantinos Kallas, *University of California, Los Angeles;* Benjamin C. Lee, *University of Pennsylvania*

https://www.usenix.org/conference/nsdi25/presentation/xing

This paper is included in the
Proceedings of the 22nd USENIX Symposium on
Networked Systems Design and Implementation.

April 28–30, 2025 • Philadelphia, PA, USA

978-1-939133-46-5

Open access to the Proceedings of the
22nd USENIX Symposium on Networked
Systems Design and Implementation
is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

Rajomon: Decentralized and Coordinated Overload Control for Latency-Sensitive Microservices

Jiali Xing[1], Akis Giannoukos[1], Paul Loh[1], Shuyue Wang[1], Justin Qiu[1], Henri Maxime Demoulin[2], Konstantinos Kallas[3], and Benjamin C. Lee[1]

[1]University of Pennsylvania, USA. [2]DBOS, Inc, USA. [3]University of California, Los Angeles, USA.

Abstract

Microservices are increasingly central for cloud applications due to their flexibility and support for rapid integration and deployment. However, applications often experience overload or sudden traffic surges that exceed service capacity, resulting in increased latency or service failures. Moreover, microservices are decentralized, interdependent, and multiplexed, exacerbating risks from overload.

We present RAJOMON, a market-based overload control system for large microservice graphs. RAJOMON controls overload through distributed rate-limiting and load shedding. Clients attach tokens to requests and services charge a price for each API, dropping requests with insufficient tokens. Tokens and prices propagate through the entire call graph, piggybacking on requests and responses. Thus, RAJOMON is the first decentralized, end-to-end overload control system.

We implement and evaluate RAJOMON on a setup of up to 140 cores and on a variety of applications from academia and industry. Experiments indicate RAJOMON protects microservice goodput and tail latency from substantial demand spikes, even in the case of mixed request types and deeper service graphs. For high-load scenarios, RAJOMON reduces tail latency by 78% and increases goodput by 45% when compared against state-of-the-art overload control for microservices.

1 Introduction

Microservices enable extensible and flexible software architectures, advantages that have led to their adoption in many large, production systems [6, 7]. Effectively managing overload, where demand exceeds the system's capacity for computation, is essential for performance and reliability. Demand spikes might arise from expected events, such as sales or holidays, or unpredictable variations in client behavior. Supply inefficiencies might arise from hardware issues, such as power throttling, or software issues such as system misconfiguration or garbage collection. Overload can severely degrade user experience and system reliability [3, 52]. It is the principal culprit for cascading failures within service graphs [2].

Microservice applications, compared to monolithic systems, have several characteristics that make overload control more challenging. First, microservices are deployed automatically and managed in a decentralized manner [19]. Second, microservices are highly interdependent, creating the risk that congestion at one node can propagate along the execution path, turning localized overload into cascading failures [13]. Finally, microservice nodes are often multiplexed, serving multiple request interfaces. Each interface may involve different downstream call graphs but share the same nodes, making it difficult to detect which interfaces are overloaded [24, 25].

An effective, responsive overload control framework should incorporate three elements: (1) decentralized detection and decisions to improve responsiveness; (2) coordination across the call graph to control load early in the execution path, ideally at client side; and (3) the ability to distinguish and manage overload at interface granularity.

Current systems handle request surges with load shedding and rate limiting (Table 1), but they do not provide the key elements for effective overload control. Centralized approaches struggle to respond to overload in a timely manner because they rely on extensive telemetry and computation [34]. Without coordination across the call graph, many requests are processed by some nodes and dropped by others, wasting computational resources and lengthening queues. [52]. Without interface granularity, overloaded and idle interfaces are treated equally, causing head-of-line blocking and dropping requests [3]. These limitations hinder the practical deployment of previously proposed frameworks.

We respond to these challenges and propose RAJOMON to achieve key desiderata with tokens and price tables. Clients are granted *tokens* at predefined rates, and these tokens are attached to each request and its sub-requests. Each microservice maintains a table of *prices*, which specify the number of tokens required for an interface. First, RAJOMON decentralizes overload control as clients rate limit themselves when their token holdings are insufficient while services reject requests with insufficient tokens. Second, RAJOMON coordinates control across the microservice graph without extra

Frameworks	Decentral-ization	Coordi-nation	Client Rate Limiting	Interface Granularity	Time Scale
Dagor	✓	Pair-wise	✗	✓	1s
Breakwater	✗	✗	✓	✗	< 1ms
BreakwaterD	✓	✗	✓	✗	< 1ms
TopFull	✗	✓	✗	✓	10s
Rajomon	✓	✓	✓	✓	< 1s

Table 1: Related work of microservice overload control, with time scales represented in seconds (s) and milliseconds (ms).

network overheads. When a service experiences overload, it increases prices and attaches updated prices to responses, thereby informing upstream services and end-users about congestion, who then proactively limit new requests before they are issued to overloaded execution paths. Finally, prices are updated independently for each interface based on their downstream interfaces, ensuring that overloaded interfaces are properly throttled while idle ones remain unaffected.

Rajomon is designed and implemented as a gRPC interceptor package that does not modify application code [5]; we will release the source code. We evaluate Rajomon using service graphs, with varied depths and sizes, from academic benchmarks and industrial traces. Compared to state-of-the-art baselines [3, 34, 52], Rajomon improves goodput (*i.e.*, throughput within latency requirements) by 117–266% and reduces latency by 33–46% under challenging system conditions. When traffic surges, Rajomon goodput recovers within a second, four times faster than baselines. Rajomon's differentiates between concurrent requests across different interfaces, increasing goodput by 45–245% and reducing latency by 78–94%.

2 Motivation

Overload occurs when computational demands exceed system capacity, causing requests to queue, violate service-level obligations, or crash the service. For instance, the release of Taylor Swift's 'Midnights' album drew overwhelming traffic and caused a global outage of Apple Music and Spotify [15]. Another example is Coinbase's Super Bowl ad, which crashed its own website [37]. Such outages cause financial losses and harm user experience; 61% of users immediately turn to competitors when they have difficulty accessing websites [17].

We motivate an overload control framework that targets microservice-based applications, detailing their characteristics that lead to unique challenges. We address these challenges by specifying desiderata for the control framework. We also describe limitations of existing solutions.

2.1 Microservices

An application structured using microservices breaks down functionality into separate components, each of which is implemented in its own service [13, 25]. Each service provides an interface (API) that supports specific requests and communicates with other services, through remote procedure calls (RPCs), to perform an end-to-end task. By tracking services invoked for the task, we can construct a call graph. Several properties of the microservice computation – decentralized operation, interdependent requests, multiplexed services – pose significant challenges for overload control.

Decentralized Operation. Microservices are loosely coupled, through RPCs, and each is usually implemented, maintained, and managed by a different organization or team [19]. This decentralized structure makes traditional, monolithic overload control methods impractical because there is no single entity that has ownership or visibility over all services. Furthermore, large microservice applications tend to include hundreds or even thousands of services [25]. At this scale, even distributed tracing incurs significant delays. For instance, Dapper takes up to 15 seconds to propagate trace data, while Jaeger also operates asynchronously, adding further delay [11, 36]. As a result, centralized control mechanisms are unable to respond quickly enough because they first need to observe the state of all services.

Interdependent Requests. Microservice-based applications have graph structures with complex dependencies. According to Alibaba, 30% of graphs are wide and exhibit a fan-out pattern in which a service calls many others [24]. Many other graphs are deep and exhibit long chains of services. An overloaded service can cause cascading quality-of-service violations throughout the graph [13]. When an upstream node exhibits increased queuing delay, request queues have already lengthened in downstream nodes [25].

Multiplexed Services. Services are often multiplexed and shared by multiple applications and types of requests, which invoke different application programming interfaces (APIs). Approximately 90% of call graphs contain multiplexed services that are included in other call graphs and interact with various upstream and downstream services [24,26]. With such multiplexing, one type of request might overload a shared service, causing head-of-line blocking and affecting the processing of other non-overloaded interfaces [25]. On the other hand, overload control actions suitable for one interface could be too conservative and starve other interfaces [34]. Determining the optimal rate for concurrent APIs given the varying capacities of downstream servers is challenging.

2.2 Control Desiderata

We design and implement Rajomon to address the unique challenges from microservices. Our framework supports three key properties for effective overload control.

Decentralized Decisions. Every client rate limits and every service sheds load independently. Decentralized decisions require exchanging and propagating load signals. Yet no service blocks on communication or commands from other services, ensuring timely reaction to overload.

Coordinated Control. Services propagate load metadata, recursively and eventually, to upstream services and clients. Metadata flows through the microservice graph based on paths taken by varied request types. Thus, services are aware of downstream overload and proactively reduce the rate of upstream requests. Such collaborative load shedding reduces wasted computation throughout the graph.[1]

Interface Granularity. Clients and services monitor and control overload for each interface separately. Control at interface granularity is more efficient than prior approaches, which operate at client granularity [3], because overload often occurs at specific interfaces rather than broad services or certain clients. RAJOMON applies rate limits and active queue management differently for each request type. Its approach can be viewed as *virtual multi-queues* for head-of-line blocking in the microservice context.

2.3 Existing Approaches

There exist several established approaches for overload control, but none adequately address the challenges inherent to microservice-based applications, as shown in Table 1.

Dagor implements decentralized overload control [52]. Each server monitors its queuing delay to determine the admission level. Dagor attaches a priority score to each request and expresses admission level as a priority cut-off. Each server drops requests that fail to meet this cut-off. Services communicate admission levels to their senders so that load is shed one tier before hotspots within the graph.

Breakwater implements single-layer overload control to manage interactions between one server and its clients [3]. The front-end service adjusts total credits based on queuing delays and then issues credits to each client to explicitly control the number of requests they can send. The front-end increases server utilization by over-issuing credits and then dropping requests when the excess of credits overload the server.

TopFull is an adaptive overload control framework for microservice applications that operates in a top-down manner [34]. It dynamically controls API load by adjusting request rates at the front-end. It first clusters APIs according to their graph structure and shared nodes. For each cluster, reinforcement learning throttles requests at API granularity based on observed end-to-end goodput and latencies.

Empirical Comparison. For motivation, Figure 1 presents throughput, dropped requests, and total demand for the *Compose Post* request in a social network application. Initially, all frameworks are able to match goodput to total demand. But when demand spikes at 5 seconds, performance degrades.

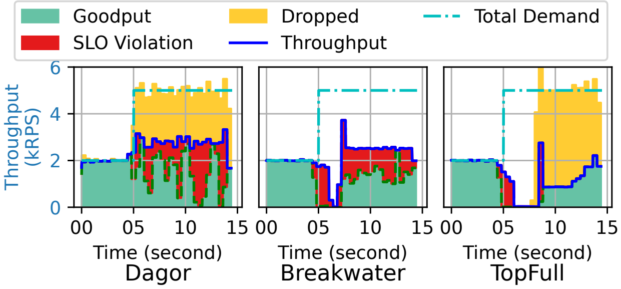


Figure 1: Existing OC approaches failed to handle the overload.

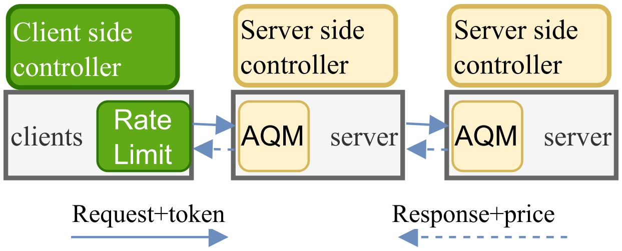


Figure 2: RAJOMON architecture.

Dagor lacks end-to-end coordination, shedding load only at nodes that precede hotpsots in the graph. This wastes resources and slows response time as many requests are processed by other services but are ultimately dropped. Breakwater, enabled only at the front-end, is unaware of real-time load conditions at downstream services such that overload causes service failure for 2 seconds. Moreover, Breakwater's decisions are misaligned with actual back-end capacities, leading to incorrect rate limits and causing half of requests to violate service-level obligations. TopFull's centralized reinforcement learning policy also experiences service failure for 4 seconds. Although it eventually recovers, communication and computation overheads delay system adaptation.

3 RAJOMON Design

Tokens function as a universal currency for computations requested by a sender (clients or upstream services) from a receiver (downstream services). Unlike conventional token bucket systems, RAJOMON's tokens accompany requests as they traverse the call graph. From the sender's perspective, tokens dictate each client's capacity to send requests. From the receiver's perspective, tokens signify each request's relative priority and enable collective agreement on which requests to shed first during overload (*i.e.*, those with fewer tokens).

RAJOMON employs tokens for two reasons. For clients, tokens permit differentiated rate limiting, which is important for microservices with multiple interfaces and diverse load dynamics at each interface. Clients should issue different

[1] Note that Dagor also implements collaborative load shedding, but does so between single sender-receiver pairs, which limits scalability to larger applications [52].

request types at different rates.

For services, tokens permit coordinated load shedding, which is important for efficiency. Applications often exhibit wide fan-outs in which one node in the call graph issues requests to and waits for multiple downstream nodes [24]. When load shedding is required, downstream nodes should coordinate and drop all sub-requests originating from the same request. Because all sub-requests must complete, the system need not compute for some when others are dropped. Tokens, attached to requests, provide the needed metadata about the original request and avoid additional communication for coordinated load shedding.

Prices represent the current demand and supply of computational resources at each service. Practically, prices set a threshold for the number of tokens required for a request's admission. Prices rise to curtail admissions during overload and fall when load eases. Prices propagate from downstream nodes toward clients, providing updated signals about system load to all services in the graph. A price is set for each interface. A table holds all prices for a service's interfaces.

Prices are necessary for sharing congestion information across nodes. Because each service interface calls different downstream services, its computational capacity depends on related downstream capacities. Determining whether a particular interface is overloaded requires callees that share their congestion levels with callers. Prices permit this coordination and decentralized overload control at interface granularity.

Controllers. Each node runs a local RAJOMON controller (Figure 2). The **Client-Side Controller** prevents clients from overwhelming the system by regulating the rate of outgoing requests. It generates tokens at a rate defined by its policy. Before a client issues a request, its controller ensures sufficient tokens are available for the corresponding interface. If not, it rate-limits and drops the request. Otherwise, it removes tokens from its pool and attaches them to the outgoing request. Periodically, the controller updates its price table based on the latest information received from the server.

The **Server-Side Controller** performs admission control, comparing tokens attached to incoming requests with the price of the requested interface. If tokens are insufficient, the request is dropped. Otherwise, the request is admitted and tokens are attached to any subsequent requests sent to downstream services. The controller periodically updates prices based on local queuing delays and feedback from downstream services to reflect the graph's current load conditions. Price updates are communicated back to upstream services and clients as part of the response metadata. Thus, controllers independently assess and manage load for each call path, enabling decentralized yet coordinated adaptive queue management.

3.1 Mechanisms for Overload Detection

The visibility of load conditions is limited because tasks require computation across many microservices loosely connected by RPCs. Measuring resource availability and service capacity is complicated by diverse graph topologies and request types. Simple indicators for overload in a single server, such as CPU utilization or queuing delay [3,52], do not apply.

RAJOMON supports decentralized overload detection for loosely connected services. A service calculates its price for computation based on local queuing delays. Each of its interfaces sets its own price, allowing the system to track and manage congestion separately. Moreover, to capture end-to-end load conditions and quantify the total price a request must pay, prices accumulate from back-end services to clients.

When a service receives a price update from downstream callees, it updates the total price and shares with upstream callers. This backward propagation of prices informs clients and callers of the overload status for all downstream interfaces. Prices circulate asynchronously without requiring a centralized message broker.

3.2 Mechanisms for Overload Control

Each RAJOMON controller acts independently, making decisions about rate-limiting or adaptive queue management based on requests' attached tokens and interfaces' prices.

RAJOMON's control decisions account for call graph topology to minimize wasted computation. To reduce waste in parallel computation, which arises when one caller's requests fan out to multiple callees, RAJOMON uses token holdings to coordinate load shedding across downstream services. Overloaded services collaboratively shed sub-requests originating from the same caller's request, thereby avoiding unnecessary computation in parallel branches of the service graph.

To reduce waste in sequential computation, which arises when a client requests computation from a chain of microservices, RAJOMON controls load earlier in the call graph. Ideally, load is shed at clients or upstream services. By the time a request arrives at downstream services, its computation has already consumed significant service capacity and, for efficient goodput, should be completed if possible.

3.3 Policies for Client Control

Policies are devised to tune system responsiveness to overload while maintaining flexibility. These policies optimize the rate and granularity at which tokens and prices are updated.

Asynchronous Token Generation. The Client-Side Controller requires policies for token generation and spending, which should ensure stable throughput and smooth transitions between rate limiting and load shedding levels. RAJOMON uses a Poisson process to replenish tokens, which disperses token generation and client requests across time.

In contrast, generating tokens at some constant rate tends to produce bursty requests and oscillating goodput as clients simultaneously or synchronously replenish their token holdings. For example, when every client replenishes tokens every

10ms, they all send more requests at the beginning of the 10ms period. Randomizing the number of replenished tokens ameliorates this problem but does not fully resolve it.

Randomized Token Spending. RAJOMON selects a uniform random number of tokens for each outgoing request, yielding a range of tokens attached to requests such that the number of dropped requests increases gradually as the price increases. This policy provides a smoother reaction to overload and enhances system responsiveness.

Attaching a deterministic number of tokens to requests destabilizes throughput. A simple strategy spends all available tokens on the next request. But because clients may hold a similar number of tokens, the all-in strategy produces requests with similar numbers of tokens. Under this policy, token-based adaptive queue management is ineffective because the controller cannot distinguish priority across requests.

3.4 Policies for Server Control

Regularly updating and sharing prices are essential for timely overload control. Prices must reflect system dynamics and load conditions from nodes throughout the call graph.

Proportional Price Updates. The Server-Side Controller compares queuing delay against a threshold at regular intervals, determining whether price must be updated for a particular service interface. Threshold and interval are configurable parameters that typically range from 1ms to 20ms.

RAJOMON increases price by an increment that is proportional to queuing delay. The greater the queuing delay in excess of the threshold, the greater the price increase. Practically, when queuing delay exceeds the threshold by 1ms, the price increases by approximately 3 to 13 tokens. When queuing delay is less than half the threshold, the price decreases by 1 token. Otherwise, price is unchanged.

Traditionally, additive-increase / multiplicative-decrease (AIMD) algorithms adjust rates in token bucket mechanisms (*e.g.*, SEDA [41, 42]). AIMD divides admission rate by a constant factor (*e.g.*, two) during overload. But it throttles incoming requests by the same factor regardless of overload severity. AIMD cannot precisely control load.

Lazy Price Propagation. The Server-Side Controller sends prices to upstream nodes in the call graph with a lazy update policy. Lazy updates mitigate overheads associated with price updates. Because microservice functions and their corresponding RAJOMON controllers run on multiple threads, overheads could arise from ensuring thread safety for price updates. Because applications are implemented with tiers or layers of microservices, an overhead added to each layer can easily accumulate in deep call graphs and increase end-to-end latency. These overheads are modest when RAJOMON need not attach metadata to every response in the call graph. Upon sending a response, the controller attaches its price information with a configured probability, *e.g.*, 20%, updating the upstream services with the current local prices.

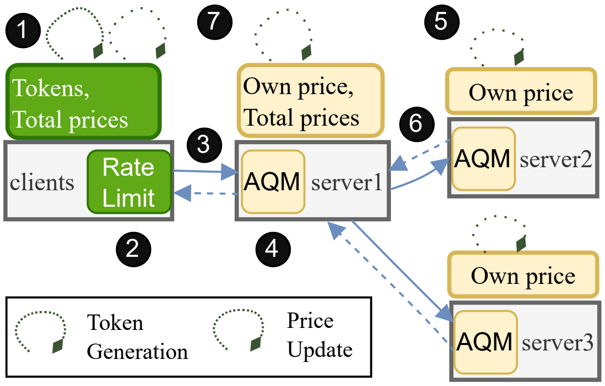


Figure 3: RAJOMON example.

Maximum Total Price. The Server-Side Controller determines the price of its computation by combining its local price with downstream price data. A service's price is the price of local computation plus the maximum of prices published by relevant downstream services. This policy emphasizes the most overloaded downstream service, allowing rapid responses to hotspots. Moreover, this policy is easy to implement. RAJOMON simply forwards a request's token holdings through the graph as it proceeds from client to back-end.

In contrast, an intuitive policy would resemble supply chains. A service's price is its local price plus the sum of prices for all relevant downstream services. If prices are additive, then token spending must also be additive. Whenever a request fans-out and triggers parallel sub-requests for downstream service, its token holdings must be split, allocated, and attached to those sub-requests. We must split tokens so adaptive queue management at back-end services are effective, but doing so adds complexity and latency.

3.5 Example

Figure 3 provides an example of RAJOMON's operation. A client requests computation from three services. Suppose server 1's `login` interface calls both server 2's `authentication` and server 3's `homepage` interfaces.

❶ The **Client-Side Controller** generates tokens periodically as determined by its policy (*e.g.*, 10 tokens every 10ms). ❷ The client issues a request to `login` if it holds enough tokens for the interface. Here, the client holds 10 tokens and `login`'s price is 5 tokens. ❸ The client spends 5 tokens, attaching them to the outgoing `login` request.

❹ The **Server-Side Controller** checks the request's tokens against the price, admits the request, and attaches 5 tokens to each sub-request to `authentication` and `homepage`. ❺ The server periodically updates prices based on queuing delays, increasing prices for `homepage` and `authentication` to 7 and 8 tokens, respectively. ❻ Servers 2 and 3 drop incoming requests, which hold 5 tokens, and responds to server 1 with updated prices. ❼ Server 1 adjusts its price table based on local queuing delay and the maximum price of downstream services (*i.e.*, 8 tokens). When server 1 receives and drops incoming requests with insufficient token holdings, it will respond to upstream clients with updated prices.

This example illustrates how RAJOMON coordinates load shedding and rate limiting for microservice graphs. For fanned-out microservices, suppose Servers 2 and 3 must drop half their requests. Since sub-requests inherit tokens from their originating request, the servers can identify and drop the half with fewer tokens attached. The servers would focus on completing the remaining sub-requests and better translate computation into goodput.

For a chain of microservices, when Server 1 receives increased prices from Servers 2 and 3, it immediately adjusts its price and drop more incoming requests. Doing so mitigates overload earlier in the graph and reduces load shedding at Servers 2 and 3. Moreover, Server 1 propagates its increased price to clients and prompt more aggressive rate limiting.

4 Implementation

The RAJOMON prototype is implemented as an open-source gRPC package [48] in 948 lines of Go code [5, 9]. The implementation includes three parts. First, for overload detection, RAJOMON implements a goroutine that periodically measures queuing delays for other running goroutines. Go uses goroutines instead of threads and our implementation is analogous to prior techniques that monitor thread queuing delays [3, 52]. Second, for price tables, RAJOMON uses synchronous maps with built-in support for concurrent use [16]. Third, RAJOMON implements gRPC interceptors for service receivers, service senders, and clients. Interceptors implement rate-limiting and adaptive queue management (§3).

Baselines and implementation. We compare RAJOMON to state-of-art microservice overload control frameworks—(i) Dagor, (ii) Breakwater, (iii) Breakwaterd, (iv) TopFull—which were introduced in Section 2.3. Breakwaterd is a distributed implementation of Breakwater that rate limits across all sender-receiver pairs in the service graph. This was suggested as a future work, but was neither implemented nor evaluated in the original study [3].

For fair comparison, we implement all baselines in Golang as gRPC libraries. The re-implementation follows the original design, but as a gRPC interceptor, we removed Breakwater's dependencies on certain hardware or operating system kernels [3, 12, 32]. We open source our implementation for Dagor, Breakwater(d), and TopFull [23, 46, 47].

Benchmarks. We evaluate RAJOMON using academic benchmarks and industry traces from Alibaba (Table 2). We re-implement academic benchmarks, Social Network and Hotel Reservation, in GoLang with gRPC and Redis based on code from [49]. SLOs for each interface are set to 5× its 95th-percentile latency under normal conditions.

We use microservice application traces from Alibaba [24] to generate synthetic benchmarks (traces are not deployable applications). We consume service call graphs and each service's computation times to produce corresponding service instances in Golang. Each service runs a busy-loop to emulate each request's computation. We focus on three highly demanded microservice endpoints—S_102000854, S_149998854, S_161142529—which we denote *S1*, *S2*, and *S3*. Endpoints are representative of call graphs of different sizes (10-20, 20-30, 30-40 nodes).

App	Interfaces	Depth	Width	#Nodes	SLO (ms)
Social Network	Compose Post	4	3	9	90
	User Timeline	4	2	7	20
	Home Timeline	4	1	5	20
Hotel Reservation	Search Hotel	4	3	9	60
	Reserve Hotel	3	2	5	25
Alibaba	S1	5	10	20	250
	S2	9	10	33	600
	S3	6	6	18	375

Table 2: Microservice benchmarks and SLOs.

Parameter Tuning. Overload control frameworks commonly involve four key parameters. First, queuing threshold specifies when overload control is triggered. Second, update frequency specifies how often control variables such as token holdings and prices adjust. Third, step width specifies the magnitude of the updates to control variables. Finally, client-side parameter include the token generation rate in RAJOMON and the expiration time in Breakwater. We allow Breakwaterd to have two sets of parameters—one for the front-end service and another for the other services—resulting in twice the number of parameters compared to Breakwater.

We use Bayesian optimization to tune overload control parameters for each application on its slowest request interface (*e.g.*, *Compose Post* for *Social Network* and *Search Hotel* for *Hotel Reservation*). The objective function for tuning is a weighted sum of goodput and tail latency, specifically defined as $goodput - 10 \times \max(0, latency - SLO)$, following similar methods from prior work [3, 4, 34]. In each iteration of the optimization procedure, we generate a traffic surge from 80% to 200% of the application's maximum sustainable load and then measure goodput and tail latency.

For TopFull, we adhere to their transfer learning process. We implemented a graph-based simulator to first train the RL model in the simulated environment before fine-tuning it on the real application. We use the same reinforcement learning model and parameters as described in the original paper.

5 Evaluation

We evaluate RAJOMON performance characteristics to answer the following questions.

1. How does RAJOMON affect the goodput and latency of microservice graphs under various loads? (§5.2)
2. How does RAJOMON impact the time required for goodput to recover from an overload condition? (§5.3-5.4)
3. What is the impact of RAJOMON on the goodput and latency of multiple, concurrent request interfaces? (§5.5-5.7)
4. How large is RAJOMON's parameter variation compared to other control schemes?(§5.8)
5. What is the latency overhead of RAJOMON compared to other control schemes? (§5.9)

5.1 Experimental Setup

Testbed. We perform most experiments on seven CloudLab m510 servers, equipped with 8-core Intel Xeon D-1548 CPUs, 64GB DDR4 RAM, 256GB NVMe SSDs, and 10GB NICs, unless otherwise stated. For *Social Network* and *Hotel Reservation* experiments involving multiple interfaces (§5.5, 5.6), we scaled up to seven c220g2 servers, each equipped with 20-core Intel Xeon E5-2660 v3 CPUs, 160GB RAM, and 10GB NICs. All servers run Ubuntu 18.04 LTS with kernel 4.15.0. Services are deployed with Kubernetes [21]. One dedicated node runs the control plane and clients while all other nodes run microservices. All nodes communicate through a single router. Each experiment for the *Social Network* or *Hotel Reservation* applications runs on eight nodes, while the Alibaba application runs on twenty nodes.

Load Generation and Measurement. We extend *ghz* for load generation [8]. On the client node, 1000 workers establish 1000 concurrent gRPC connections and issue requests asynchronously to the application. Requests follow a Poisson process with some rate specified by the experiment. We randomize input data for *Hotel Reservation* and *Social Network* requests because experiments focus on success rate rather than the content of responses.

Each experiment consists of a warmup (5s) and overload (10s) phase unless otherwise noted. For experiments with a single request type, we warm up with load at 80% of the application's maximimum sustainable throughput subject to its SLOs and then overload to some rate specified by the experiment. For experiments with multiple concurrent request types, we warm up with fast, computationally light requests and then overload with slow, computationally heavy requests.

We measure end-to-end request latency during overload and report 95th-percentiles for latency and goodput (*i.e.*, number of requests processed within the SLO). We do not report warm-up performance unless otherwise noted.

5.2 Performance for Single Interfaces

Figure 4 presents data for *Search Hotel* from the *Hotel Reservation* app and *Compose Post* from the *Social Network* app. Each experiment includes a warm-up phase—4k requests per second (RPS) for *Search Hotel* and 2k for *Compose Post*—followed by an overload phase with varied load spikes on x-axis. Data is the average of five experiments with error bars indicating variance in latency and goodput.

Results. For *Search Hotel*, RAJOMON ensures tail latency remains below 200ms and sustains goodput of approximately 3k RPS during overload. Goodput drops only marginally as traffic load increases. In contrast, baseline techniques cannot maintain performance as load increases. Once load exceeds 12k RPS, their tail latencies are five times that of RAJOMON's and their goodputs are less than half of RAJOMON's. TopFull maintains lower, steady goodput at 1k RPS but suffers from much higher tail latencies of 750ms.

For *Compose Post*, RAJOMON's tail latency increases modestly above the 80ms SLO, approaching 100ms, while goodput remains stable above 2k RPS. In contrast, baseline techniques exhibit a piecewise linear trend. Tail latencies spike to 800ms when load is 5k RPS and then stabilize. Goodput drops to 500 RPS and then stabilizes. Dagor is a bit more robust and sustains performance until load is 7k RPS.

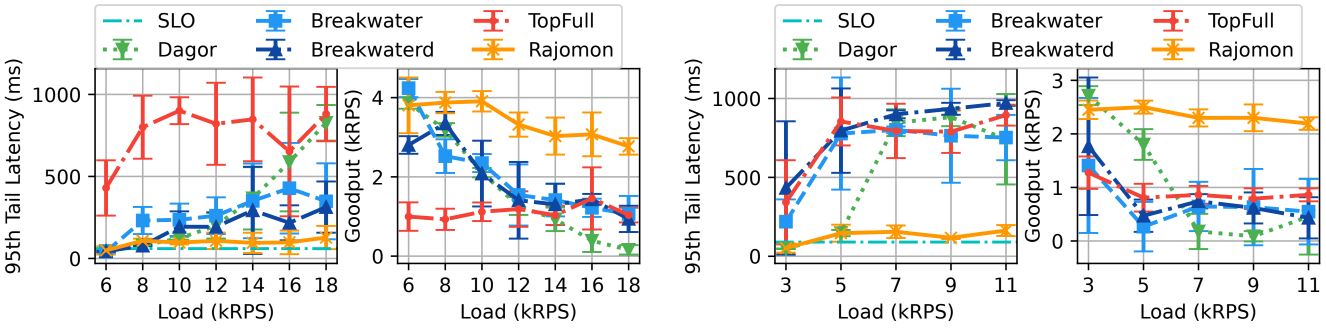
Discussion. RAJOMON's decentralized control delivers uniquely robust performance. It keeps tail latency low by quickly shedding load. Price updates regulate the rate of incoming requests from upstream clients and services. Moreover, RAJOMON's collaborative control keeps goodput high, even under heavy load conditions, as back-end services drop sub-requests that originate from the same upstream request.

In contrast, Breakwater is designed specifically for front-end services and lacks downstream visibility, which delays and diminishes any adaptation to overload in back-end services. Less effective adaptation means lower goodput and higher latency, especially under heavy load because requests can quickly accumulate in queues before overload is detected.

Dagor sheds load to counter request incast, but wastes server-side computation as client-side rate limiting would be more efficient. Shedding becomes more aggressive as load increases, which leads to lower goodput and higher latency.

Breakwaterd extends Breakwater to each sender-receiver pair in the graph but suffers for the same reasons as Dagor. Requests are rate-limited and dropped only in the node preceding an overloaded hotpsot because overload signals do not propagate through the graph. As load increases, more partially processed requests are ultimately dropped.

TopFull is limited by its coarse timescales. Within the 10-second overload phase, TopFull experiences an initial period of service unavailability followed by a slow recovery as it begins adaptation. Delayed detection and response means lower goodput and higher latency, which is consistent with its own experimental evaluation [34]. Although it is slow to



(a) Tail latency and Goodput during overload of *Search Hotel*

(b) Tail latency and Goodput during overload of *Compose Post*

Figure 4: Overall performance of RAJOMON and other baselines for *Search Hotel* and *Compose Post* requests.

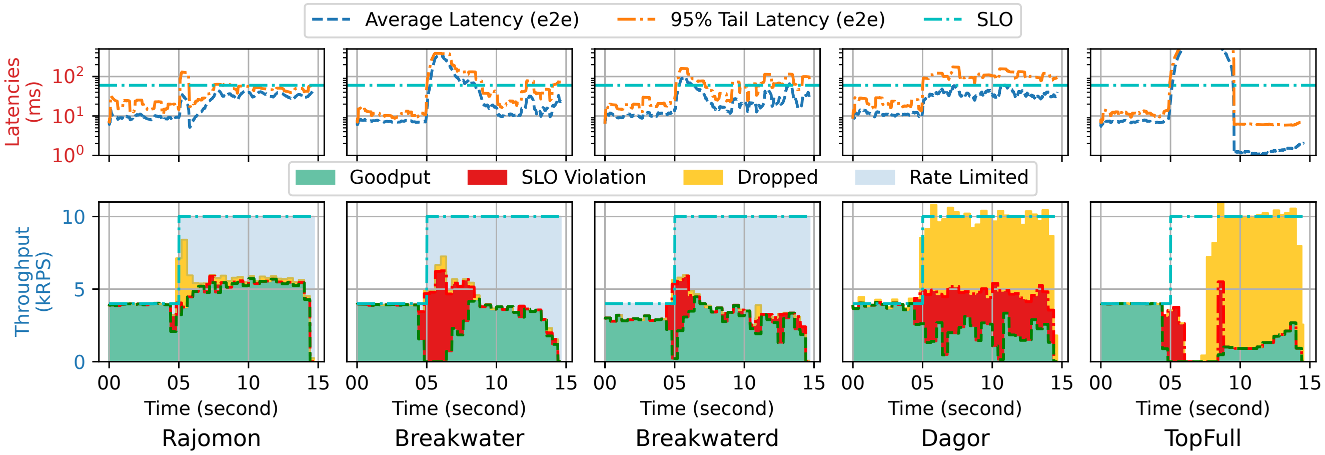


Figure 5: The throughput and tail latency under traffic surge of *Search Hotel* (single interface).

respond, TopFull's capable reinforcement learning model can eventually stabilize performance at medium and high load.

Across all baselines, performance losses are more pronounced for *Compose Post* because its back-end services require more intensive Redis writes. When not managed effectively, overload produces much greater increases in latency.

5.3 Control Dynamics for Single Interface

Figure 5 illustrates system dynamics. Client demand is 4k RPS during warm-up followed by 10k RPS during overload. The first row shows latency across time. The second row shows how requests are either rate-limited, dropped, or processed with a response time that may or may not satisfy SLOs.

Results. When overload begins at 5 seconds, RAJOMON drops many requests and then, over the next 2 seconds, steadily recovers goodput until it reaches 5k RPS. Most excess load is controlled through rate limiting thereafter. Latency initially spikes to 100ms but quickly returns to SLO target.

Breakwater(d)'s responses to overload do not occur until 2 to 4 seconds after its onset. During these periods, 50% to 100% of requests violate the SLO target latency. Eventually, latency returns to target but goodput stabilizes at lower levels, below the 4k RPS during warm-up.

Dagor maintains throughput of 5k RPS and aggressively drops the additional 5k RPS of overload. However, 95th percentile latency exceeds 100ms and most of its throughput violates the SLO. TopFull encounters 4 seconds of unavailability due to overload, followed by many dropped requests and slow goodput recovery.

Discussion. RAJOMON promptly detects server-side congestion and drops excess requests at 5 seconds. After this initial reaction, RAJOMON propagates updated prices upstream to clients, which rate limits subsequent requests. According to RAJOMON's lazy update policy, price updates are included in 20% of responses, which explains the 2-second transition from server-side load shedding to client-side rate limiting, which stabilizes latency and maintains goodput.

In contrast, Breakwater(d) experiences delays and inefficiencies due to a lack of coordination. The Breakwater front-end requires 2 seconds to detect queuing delays from back-end overload. Even after detection, back-end delays are higher than those at the front-end. Since Breakwater issues credits based on front-end delays, this mismatch adds another 2 seconds to load adjustment even though Breakwater is tuned to update credits every 1ms.

Breakwaterd responds more quickly as it reduces load at intermediate tiers. When the back-end becomes overloaded, mid-tiers rate-limit and queue incoming requests, which eventually time out. However, timed-out requests do not signal upstream services or clients, leading to wasted computation on requests that are ultimately discarded. This inefficiency explains Breakwaterd's sub-optimal goodput.

Dagor's lack of rate limiting leads to many dropped requests, which slows down the system. Dagor drops requests only in the nodes preceding hotspots. For four-tier *Search Hotel*, most services continue processing 10k RPS even though half of these requests are dropped at the second-to-last tier. This wastes upstream computation and increases latency.

TopFull's centralized, front-end-only design inherits the limitations of both Breakwater and Dagor. It detects overload only after end-to-end goodput drops, preventing timely reactions to overload and causing extended service downtime. Moreover, load is shed only at the front-end, which wastes computation on requests that cannot satisfy SLOs and contribute to goodput. Reinforcement learning should eventually adapt to new load conditions but over a long timescale.

5.4 Recovery Time

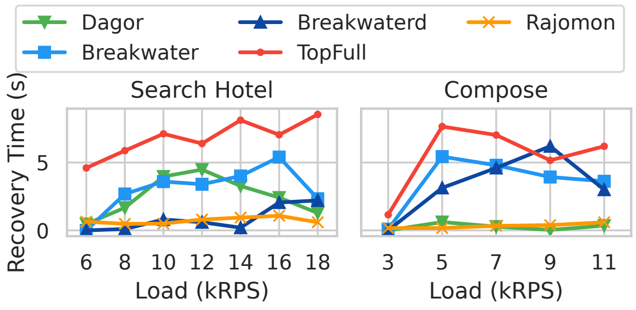


Figure 6: Recovery time of Rajomon and baselines for *Search Hotel* and *Compose Post* request.

Figure 6 plots average recovery time under varying load conditions. We define *recovery time* as the period between the onset of overload and goodput stabilization at some level.

Results. Rajomon consistently maintains sub-second recovery regardless of load conditions. Although baselines demonstrate sub-second recovery at low loads, they often struggle at higher loads. For Breakwater, recovery time rises to around 4 seconds as load increases. For Breakwaterd and Dagor, recovery times vary depending on the workload. Breakwaterd responds slowly for *Social Network* and Dagor for *Hotel Reservation*; recovery times range from 3 to 5 seconds depending on the severity of overload. Topfull is slow and often requires more than 5 seconds to recover.

Discussion. Rajomon is the only framework that consistently recovers in less than a second. Its decentralized overload control is fast and responsive, regardless of load.

In contrast, baseline techniques present a trade-off between recovery time and stable goodput. Breakwaterd and Dagor's decentralized designs help them react to overload and reach equilibrium faster, but at the expense of goodput. They respond quickly, in part, because their stable goodput after the onset of overload is much lower than Rajomon's. Similarly, as load increases, Breakwater reports lower goodput and shorter recovery times. TopFull suffers from longer recovery times due to its centralized architecture.

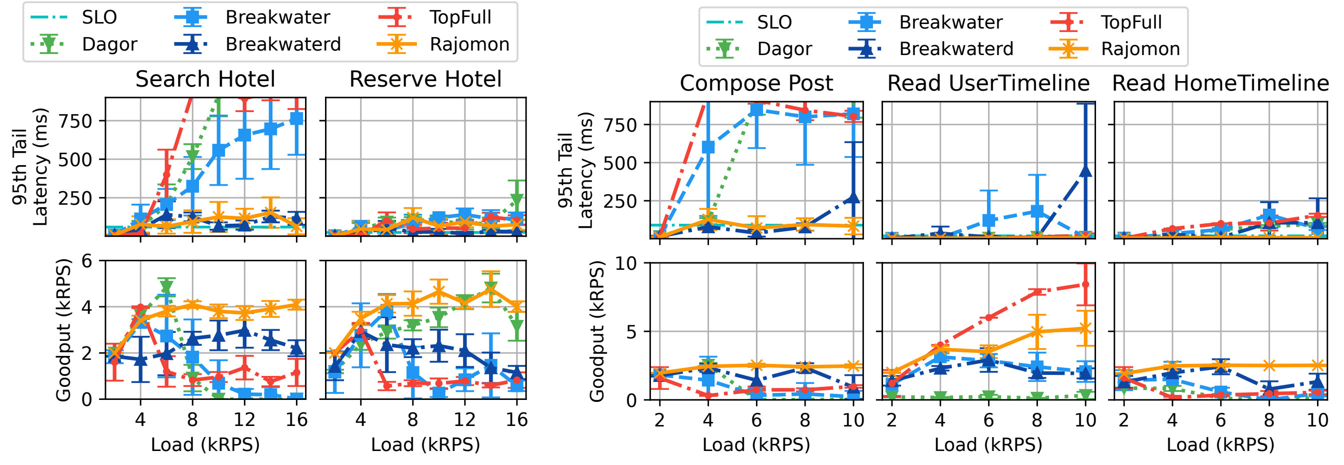
5.5 Performance for Multiple Interfaces

Figure 7 details Rajomon's performance for concurrent requests to multiple interfaces. For the *Hotel Reservation* application, we generate *Reserve Hotel* requests at a certain rate during warm-up and add an equal amount of *Search Hotel* traffic during overload. For the *Social Network* application, we generate *Read User Timeline* requests during warm-up and add *Compose Post* and *Read Home Timeline* traffic during overload. Note x-axis reports load per interface and, for example, $x = 10$k RPS in the *Social Network* application means 30k RPS total load from three request types.

Results. In Figure 7a, Rajomon maintains a latency of approximately 100ms and a goodput of 4k RPS for both *Search Hotel* and *Reserve Hotel* requests, even as the load increases. In contrast, Breakwater, Dagor, and TopFull maintain low tail latencies for *Reserve Hotel* but exhibit increased latencies for *Search Hotel* as the load increases. The goodput of most baselines for both interfaces drops from 4k to 1k as load approaches 14k RPS. Two exceptions are Breakwaterd for *Search Hotel* and Dagor for *Reserve Hotel*, both of which maintain goodput above 2k RPS while sustaining low tail latencies.

In Figure 7b, Rajomon maintains latency within SLOs across all request types. It sustains *Compose Post* and *Read Home Timeline* goodput at 2.5k RPS, while *Read User Timeline* goodput increases to 5k RPS as load intensifies. Most baselines exhibit reduced goodput and increased latencies for *Compose Post* once the load surpasses 4k RPS. For *Read User Timeline* and *Read Home Timeline*, baseline goodput diminishes to less than half of Rajomon's performance as the load nears 10k RPS. Notably, Topfull achieves higher *Read User Timeline* goodput beyond 6k RPS but at the expense of *Compose Post* and *Read Home Timeline* interfaces. Breakwaterd stands out by maintaining latency within SLOs for most interfaces; however, it experiences a decline in goodput.

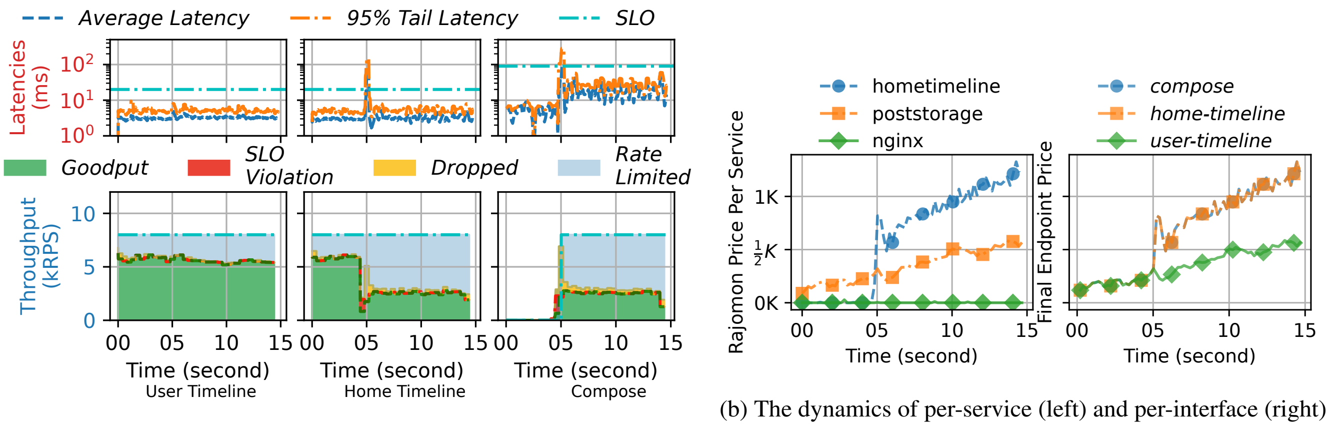
Discussion. Rajomon demonstrates low latency and high goodput even under heavy concurrent loads from multiple request types. It achieves this by combining AQM with rate-



(a) The performance of OC on concurrent *Search Hotel* and *Reserve Hotel* requests.

(b) The performance of OC on concurrent *Compose Post*, *Read User Timeline*, and *Read Home Timeline* requests.

Figure 7: The performance of RAJOMON and baselines on concurrent requests (up to 30k RPS in total).



(a) The dynamics of the latency and throughput.

(b) The dynamics of per-service (left) and per-interface (right) prices.

Figure 8: The dynamics of the throughput, latency, and prices for concurrent *Social Network* requests.

limiting and assessing the maximum sustainable load for each interface separately. RAJOMON's advantage is particularly evident for *Read User Timeline*, which has the potential for higher goodput and lower latency because these requests do not share hotspots with *Compose Post* and *Read Home Timeline*. Only RAJOMON manages to realize this potential by avoiding head-of-line blocking. Per-interface pricing effectively virtualizes a multi-queue system and sets distinct admission rates for each interface.

Dagor distinguishes between interfaces, but its adaptive queue management lacks coordination across the service graph. The same inefficiencies observed in Figure 5 persist: The massive load shedding in mid-tier services increases the latencies of *Search Hotel* and *Compose Post* requests. This finding highlights that interface granularity must be combined with end-to-end coordination to be fully effective.

Breakwater are oblivious to interfaces and suffer from head-of-line blocking because, for instance, they cannot rate limit the slow requests (*Search Hotel* and *Compose Post*) independently. As a result, other interfaces also suffer from low goodput. If Breakwater could differentiate between interfaces, they could issue fewer credits for slow interfaces, ensuring performance for all interfaces. Breakwaterd achieves low tail latencies but at the expense of reduced goodput. This occurs because Breakwaterd reduces load in intermediate tiers without coordinating with frontend and end users, as previously discussed in Sections 5.2 and 5.3. Therefore, effective rate-limiting mechanisms should incorporate interface granularity and end-to-end coordination to optimize request rates.

TopFull cluster interfaces based on shared hotspots and implements admission control for each cluster. In our case, *Search Hotel* and *Reserve Hotel* form one cluster, *Compose*

Post and *Read Home Timeline* form another, and *Read User Timeline* forms a third. This approach can sometimes work. The control agent sets a 6k RPS rate for *Read User Timeline* while keeping latency within SLOs. But, for other clusters, the lack of decentralized control and client-side rate-limiting constrains performance. Although TopFull can set different rates for *Search Hotel* and *Reserve Hotel*, its slow reaction times prevent it from handling even 1k RPS effectively.

5.6 Control Dynamics for Multiple Interfaces

Figure 8 analyzes system dynamics for the *Social Network* application's multiple request types. Figure 8a follows the same interpretation as Figure 5, except for the presence of multiple concurrent requests. Figure 8b details service prices at interface granularity. Recall that the price of a service is the maximum of all the downstream prices paid by its subrequests.

Results. Figure 8a indicates a negligible number of dropped requests and SLO violations. Once *Compose Post* requests arrive, RAJOMON immediately rate limits *Read Home Timeline*. Figure 8b illustrates how RAJOMON dynamically adjusts prices for different services to reflect their load. During warm-up, *post storage* service is the only hotspot and its price dictates prices for all request types. During overload, *post storage*'s prices keep increasing slowly so *Read User Timeline* interface remains inexpensive. In contrast, *home timeline*'s price increases substantially within half a second, indicating an emerging hotspot. The price increase propagates upstream such that *Compose Post* and *Read Home Timeline* both become much more expensive.

Discussion. RAJOMON's dynamic prices play a crucial role. First, prices reflect real-time load and delay across services. When *home timeline*'s delays are immediately reflected in *Compose Post* and *Read Home Timeline* prices, ensuring that overload control is precise and timely.

Furthermore, prices at interface granularity isolate *Read User Timeline* requests, which do not involve the overloaded *home timeline* service. These requests are neither rate limited nor dropped, ensuring consistent performance even when other parts of the system are under stress.

5.7 Performance for Alibaba Traces

We further evaluate three real-world traces from Alibaba (Section 4). We construct a service graph that is the union of *S1*, *S2*, and *S3*, which makes it much larger than *Hotel Reservation* and *Social Network*. We start *S1* and *S3* during warm-up and then add *S2* during overload.

Results. Figure 9 indicates RAJOMON maintains latencies below 200ms and goodput at 4k RPS for all interfaces and varied loads. Collectively, the baselines are less effective. As load approaches 8k-10k RPS, latency increases exponentially beyond 500ms and goodput falls by half.

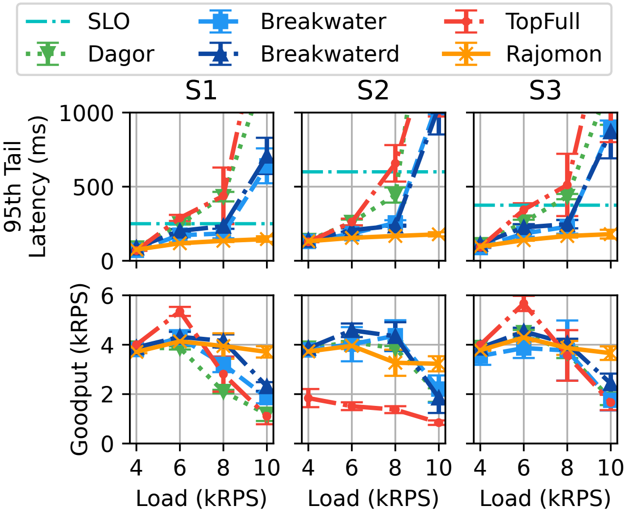
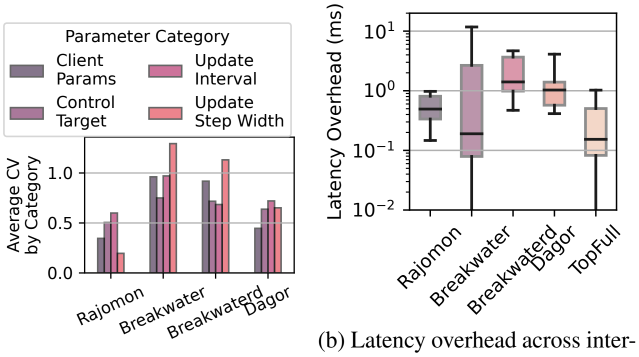


Figure 9: The performance impact of RAJOMON and baselines on large service graphs (up to 30k RPS in total).

Discussion. RAJOMON manages overload efficiently at scale, maintaining low latencies and high goodput even under extreme load. In contrast, baselines struggle under high load. Breakwater's rate limiting scales better and achieves consistently lower latencies than Dagor's adaptive queue management (*i.e.*, dropping requests) as detailed in §5.2. TopFull's RL agents set effective admission rates for *S1* and *S3*. But the reason for *S2*'s rate is unknown because TopFull is a black-box that does not detail the interactions between its RL agents. Yet its centralized approach for dropping requests suffers from higher latencies and lower goodput for all interfaces.

5.8 Parameter Variability



(a) CV of tuned parameters.

(b) Latency overhead across interfaces.

Figure 10: Comparison of parameter variability and latency overhead.

Overload control frameworks must be tuned for each application to be effective [3, 34]. A robust framework should be less sensitive to tuning parameters. Such a framework would reduce tuning time and effort, while providing more consistent performance across varied applications and system conditions, given specific parameter values.

We quantify robustness with the Coefficient of Variation (CV) for tuned parameters.[2] CV measures the relative variability of parameters, permitting comparisons across control frameworks. We calculate CV of each framework's optimal parameters in Table 3 across the three applications in Table 2.[3]

Figure 10a indicates RAJOMON performs well with similar parameters for varied applications. It exhibits the lowest variability in tuned parameter values, indicating we need not extensively tune and optimize for each application.

Breakwater(d) exhibits higher variability, indicating the difficulty of re-tuning its parameters for each application and highlighting the risks from inconsistent performance. Breakwater's credit-based rate limiting requires precise control over the total number of credits issued and held by each client, making performance more sensitive to parameters. Dagor exhibits lower variability than Breakwater(d) because priority-based adaptive queue management does not cause abrupt changes in traffic and tends to be more robust [3].

5.9 Overhead Comparison

We assess overheads by measuring differences in median latency with and without control for a system with modest 2k RPS load. Figure 10b presents the distribution[4] of latency overheads across eight service interfaces in Table 2.

RAJOMON's overhead is low for two reasons. First, it maintains and propagates only necessary state such as tokens per request and price per interface. In contrast, Dagor updates large matrices at all microservices that track request priorities and determine admission levels based on a priority cut-off: (B,U). These matrices, which grow linearly with the number of interfaces and user groups, suffer from lock contention. Note that Dagor and Breakwaterd's accumulate across tiers in deep microservice graphs, harming end-to-end latency.

Second, RAJOMON performs local price calculations asynchronously and off the critical path. TopFull also incurs low overheads because it uses only one token bucket at the front-end and its reinforcement learning decisions are asynchronous. In contrast, Breakwater calculates a client's new credit holdings c_x^{new} before every response. This overhead lengthens queues and harms latency, especially under higher loads.

6 Related Work and Discussion

Network Congestion Control Extensive work has addressed congestion control at the network layer [1, 18, 20, 22, 28, 44, 51]. RAJOMON, however, accounts for microservice-specific factors like call graphs and interdependencies. Future work could integrate RAJOMON with these mechanisms to improve end-to-end performance.

Load Balancing and Autoscaling. RAJOMON complements existing resource management strategies. While autoscaling increases service capacity, it is inherently slow and typically requires 40–200 seconds [30, 33–35, 38–40, 50]. In contrast, RAJOMON provides rapid overload control on a subsecond timescale, mitigating performance issues before autoscaling adjustments take effect. Once autoscaling provisions additional resources, the prices fall accordingly to accommodate higher throughput. Moreover, the tokens and prices primitives provide insight into request characteristics and downstream loads. This information can enhance load balancing and job scheduling decisions. Finally, even with optimal scheduling and load balancing, systems can experience overload due to unexpected traffic bursts that exceed capacity [14, 29, 31, 43, 45]. In such scenarios, RAJOMON serves as a safeguard, maintaining performance when other strategies are insufficient.

Malicious Clients. RAJOMON assumes trusted clients, such as proprietary mobile apps or Internet-of-Things devices that account for most traffic [27]. However, untrusted clients may attempt to exploit the framework for more resources or disrupt services. To prevent this, RAJOMON would require a server-side validation module to ensure token spending aligns with token generation policy and history. Trust models and validation techniques are an avenue for future research.

Dynamic Service Graphs. We focused on applications with deterministic execution paths. In practice, microservices may have dynamic call paths. If call paths depend on request metadata, such as branching based on payload, these requests can be treated as different interfaces upon receipt. If call paths are random and unknown upon receipt, for example, due to cache misses, one extension for RAJOMON could use probability theory to update its price table based on the expected value of downstream costs. For instance, if the price is 10 for a cache hit and 50 for a miss, with a 10% miss rate, the local price should be 15, based on expectation. Future work could explore other pricing policies for dynamic paths and flexible microservice settings.

7 Conclusion

RAJOMON presents a novel, market-based overload control system for microservices, using tokens and price tables for distributed rate limiting and load shedding while coordinating control throughout service graphs. RAJOMON significantly outperforms state-of-the-art baselines, improving goodput by more than 45% and reducing latency by more than 78% under high load. Its fast and precise control, with sub-second recovery times, ensures system reliability and efficiency in complex microservice systems.

[2] CV is ratio of standard deviation and mean [10].

[3] TopFull's RL control is excluded.

[4] Maximum, 75th, 50th, 25th percentile, and minimum.

References

[1] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. pfabric: minimal near-optimal datacenter transport. *SIGCOMM Comput. Commun. Rev.*, 43(4):435–446, aug 2013.

[2] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. *Site Reliability Engineering: How Google Runs Production Systems*. " O'Reilly Media, Inc.", 2016.

[3] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 299–314. USENIX Association, November 2020.

[4] Inho Cho, Ahmed Saeed, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Protego: Overload control for applications with unpredictable lock contention. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 725–738, Boston, MA, April 2023. USENIX Association.

[5] Cloud Native Computing Foundation. gRPC, A high performance, open source universal RPC framework. https://grpc.io/, 2024.

[6] Adrian Cockcroft. Evolution of Microservices - Craft Conference. https://www.slideshare.net/adriancockcroft/evolution-of-microservices-craft-conference, June 2024.

[7] Adrian Cockcroft. Microservices Workshop - Craft Conference. https://www.slideshare.net/adriancockcroft/microservices-workshop-craft-conference, June 2024.

[8] Bojan D. ghz · gRPC benchmarking and load testing tool. https://ghz.sh/, 2023.

[9] Alan AA Donovan and Brian W Kernighan. *The Go programming language*. Addison-Wesley Professional, 2015.

[10] Brian S Everitt and Anders Skrondal. *The Cambridge dictionary of statistics*. Cambridge University Press, 2010.

[11] Cloud Native Computing Foundation. Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/.

[12] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297. USENIX Association, November 2020.

[13] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[14] Zhaoyu Gao, Anubhavnidhi Abhashkumar, Zhen Sun, Weirong Jiang, and Yi Wang. Crescent: Emulating heterogeneous production network at scale. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1045–1062, Santa Clara, CA, April 2024. USENIX Association.

[15] Stephanie Giang-Paunon. Taylor Swift's 'Midnights' album crashes Spotify, leaving fans shocked; nearly 8,000 outages reported. https://www.foxbusiness.com/entertainment/taylor-swifts-midnights-album-crashes-spotify-fans-shocked-8000-outages-reported, 2022.

[16] Go. sync package - sync - Go Packages. https://pkg.go.dev/sync#Map, 2024.

[17] Google. What Users Want Most from Mobile Sites Today. https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/what-users-want-most-from-mobile-sites-today/, 2012.

[18] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't matter when you can JUMP them! In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 1–14, Oakland, CA, May 2015. USENIX Association.

[19] Boris Grubic, Yang Wang, Tyler Petrochko, Ran Yaniv, Brad Jones, David Callies, Matt Clarke-Lauer, Dan Kelley, Soteris Demetriou, Kenny Yu, and Chunqiang Tang. Conveyor: One-Tool-Fits-All Continuous Software Deployment at Meta. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 325–342, Boston, MA, July 2023. USENIX Association.

[20] Chi-Yao Hong, Matthew Caesar, and P. Brighten Godfrey. Finishing flows quickly with preemptive scheduling. *SIGCOMM Comput. Commun. Rev.*, 42(4):127–138, aug 2012.

[21] Kubernetes. Production-Grade Container Orchestration. https://kubernetes.io/, 2024.

[22] Wenxin Li, Xin He, Yuan Liu, Keqiu Li, Kai Chen, Zhao Ge, Zewei Guan, Heng Qi, Song Zhang, and Guyue Liu. Flow scheduling with imprecise knowledge. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 95–111, Santa Clara, CA, April 2024. USENIX Association.

[23] Loh, Paul and Xing, Jiali. Breakwater gRPC Implementation. https://github.com/pennsail/breakwater-grpc. Accessed: 2024-09-16.

[24] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing Microservice Dependency and Performance: Alibaba Trace Analysis. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, pages 412–426, New York, NY, USA, November 2021. Association for Computing Machinery.

[25] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, and Chengzhong Xu. An In-Depth Study of Microservice Call Graph and Runtime Performance. *IEEE Transactions on Parallel and Distributed Systems*, 33(12):3901–3914, December 2022. Conference Name: IEEE Transactions on Parallel and Distributed Systems.

[26] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, pages 355–369, New York, NY, USA, November 2022. Association for Computing Machinery.

[27] Daniele Molteni. Landscape of API Traffic. https://blog.cloudflare.com/landscape-of-api-traffic, January 2022.

[28] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A Receiver-Driven Low-Latency Transport Protocol Using Network Priorities (Complete Version). *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, August 2018. arXiv: 1803.09615.

[29] Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. Zero downtime release: Disruption-free load balancing of a multi-billion user website. 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*, SIGCOMM '20, page 529–541, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for Infrastructure-as-a-Service. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, San Jose, CA, June 2013. USENIX Association.

[31] Yipei Niu, Fangming Liu, and Zongpeng Li. Load balancing across microservices. In *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pages 198–206, 2018.

[32] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. Shenango: Achieving high CPU efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 361–378, Boston, MA, February 2019. USENIX Association.

[33] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. GRAF: a graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, CoNEXT '21, pages 154–167, New York, NY, USA, December 2021. Association for Computing Machinery.

[34] Jinwoo Park, Jaehyeong Park, Youngmok Jung, Hwijoon Lim, Hyunho Yeo, and Dongsu Han. Topfull: An adaptive top-down overload control for slo-oriented microservices. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 876–890, New York, NY, USA, 2024. Association for Computing Machinery.

[35] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[36] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver,

Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[37] Sebastian Sinclair. Coinbase site crashes from traffic as super bowl ads spark public interest. *Blockworks*, 2022.

[38] Enge Song, Yang Song, Chengyun Lu, Tian Pan, Shaokai Zhang, Jianyuan Lu, Jiangu Zhao, Xining Wang, Xiaomin Wu, Minglan Gao, Zongquan Li, Ziyang Fang, Biao Lyu, Pengyu Zhang, Rong Wen, Li Yi, Zhigang Zong, and Shunmin Zhu. Canal mesh: A cloud-scale sidecar-free multi-tenant service mesh architecture. In *Proceedings of the ACM SIGCOMM 2024 Conference*, ACM SIGCOMM '24, page 860–875, New York, NY, USA, 2024. Association for Computing Machinery.

[39] Akshitha Sriraman and Thomas F Wenisch. µTune: Auto-Tuned Threading for {OLDI} Microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, 2018.

[40] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y. Yan. 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)*, pages 149–165, Santa Clara, CA, April 2024. USENIX Association.

[41] Matt Welsh and David Culler. Overload management as a fundamental service design primitive. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, EW 10, pages 63–69, New York, NY, USA, July 2002. Association for Computing Machinery.

[42] Matt Welsh, David Culler, and Eric Brewer. SEDA: an architecture for well-conditioned, scalable internet services. *ACM SIGOPS Operating Systems Review*, 35(5):230–243, October 2001.

[43] Qizhen Weng, Wencong Xiao, Yinghao Yu, Wei Wang, Cheng Wang, Jian He, Yong Li, Liping Zhang, Wei Lin, and Yu Ding. MLaaS in the wild: Workload analysis and scheduling in Large-Scale heterogeneous GPU clusters. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 945–960, Renton, WA, April 2022. USENIX Association.

[44] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowtron. Better never than late: meeting deadlines in datacenter networks. *SIGCOMM Comput. Commun. Rev.*, 41(4):50–61, aug 2011.

[45] Bartek Wydrowski, Robert Kleinberg, Stephen M. Rumble, and Aaron Archer. Load is not what you should balance: Introducing prequal. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 1285–1299, Santa Clara, CA, April 2024. USENIX Association.

[46] Xing, Jiali. Dagor gRPC Implementation. `https://github.com/pennsail/dagor-grpc`. Accessed: 2024-09-16.

[47] Xing, Jiali. TopFull gRPC Implementation. `https://github.com/pennsail/topdown-grpc`. Accessed: 2024-09-16.

[48] Xing, Jiali and Giannoukos, Akis. Rajomon Source Code. `https://github.com/pennsail/rajomon`. Accessed: 2024-09-16.

[49] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. {MuCache}: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 221–238, 2024.

[50] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. 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*, pages 167–181, 2021.

[51] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: admission control for performance-critical rpcs in datacenters. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 1–18, New York, NY, USA, 2022. Association for Computing Machinery.

[52] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, pages 149–161, New York, NY, USA, October 2018. Association for Computing Machinery.

A Alibaba Production Trace Samples

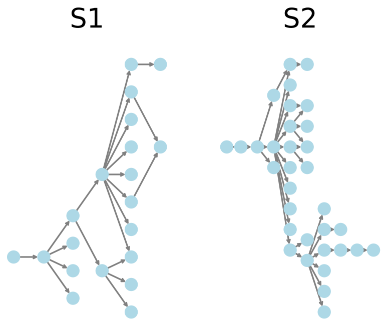


Figure 11: Sampled Alibaba services.

Figure 11 illustrates the service call graphs, *S1*, *S2*, and *S3*, sampled from Alibaba production traces. While shown separately, these graphs share some common microservices. In our experiments, the three graphs were deployed together across 20 servers.

B Overload Control Parameters

Mechanism	**Control Target**	**Update Interval**
Dagor	queuing threshold	threshold update rate
Breakwater	queuing threshold	C_{total} update rate
Breakwaterd	queuing threshold	C_{total} update rate
RAJOMON	queuing threshold	price update rate
Mechanism	**Update Step Width**	**Clients**
Dagor	α, β	u_{max}
Breakwater	α, β	initial credit, client expiration
Breakwaterd	α, β	initial credit, client expiration
RAJOMON	price update step	token update rate

Table 3: Tuned parameters for overload control mechanisms, categorized by their function.

Table 3 shows the parameters tuned for each overload control mechanism.