Fast and Efficient Scaling for Microservices with SurgeGuard

Anyesha Ghosh
The University of Texas at Austin
Austin, Texas, USA
aghosh@utexas.edu

Neeraja J. Yadwadkar
The University of Texas at Austin
Austin, Texas, USA
neeraja@austin.utexas.edu

Mattan Erez
The University of Texas at Austin
Austin, Texas, USA
mattan.erez@utexas.edu

Abstract*— The microservice architecture is increasingly popular for flexible, large-scale online applications. However, existing resource management mechanisms incur high latency in detecting Quality of Service (QoS) violations, and hence, fail to allocate resources effectively under commonly-observed varying load conditions. This results in over-allocation coupled with a late response that increase both the total cost of ownership and the magnitude of each QoS violation event. We present SurgeGuard, a decentralized resource controller for microservice applications specifically designed to guard application QoS during surges in load and network latency. SurgeGuard uses the key insight that for rapid detection and effective management of QoS violations, the controller must be aware of any available slack in latency and communication patterns between microservices within a task-graph. Our experiments show that for the workloads in DeathStarBench, SurgeGuard on average reduces the combined violation magnitude and duration by 61.1% and 93.7%, respectively, compared to the well-known Parties and Caladan algorithms, and requires 8% fewer resources than Parties.

Keywords – Cloud computing, microservices, serverless, quality-of-service, resource management, datacenters

I. Introduction

User-facing cloud applications are increasingly moving away from traditional monolithic services to a model comprising numerous single-purpose and loosely-coupled *microservices* [1], [6], [11], [12]. Microservices improve the modularity of applications, isolate errors, facilitate development and deployment, and can scale quickly and at fine granularity to meet changing demand. However, microservice-based applications raise challenges in resource management [19], [33] due to the complex application topologies and the large, unpredictable request rate surges seen in current deployments [9], [10].

Prior work attempts to tackle this problem using a variety of heuristics [16], [18], [25], [29], optimization techniques [30], and machine learning (ML) algorithms [15], [21], [26], [28], [31], [33], but all fall short in one important aspect—*they do not effectively meet tight latency objectives in the face of rapidly varying conditions, such as input load surges and temporary network delays.* Recent studies point to average request rates that are $2-3\times$ higher during a surge with much higher instantaneous request rates [7], [8]. Because current controllers respond too slowly, systems resort to wasteful approaches such as provisioning for maximum load [8] or rate limiting requests [9], [10].

We present SurgeGuard, a resource controller for microservice-based applications that is specifically designed to manage application QoS during surges in incoming load, network latency, or other disruptions to steady-state behavior. Unlike prior systems, SurgeGuard is able to both respond to events with extremely low latency to minimize their impact, and also accurately identify root-cause slowdowns to focus resources where they are most beneficial. SurgeGuard reflects three key insights: (1) relying on averaged metrics and measurements inherently delays a response to surges; (2) decisions must account for the overall task graph and provide resources to address the root cause of slowdowns; and (3) resource allocation should prioritize those services that benefit the most from the additional resources.

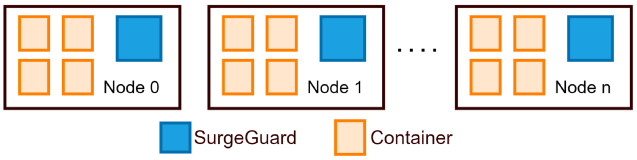


Fig. 1: Each node contains one instance of SurgeGuard managing resources for the containers on that node.

SurgeGuard is designed to be lightweight, decentralized, and to scale well to multiple nodes (Fig. 1). It relies only on local state and makes decisions without relying on other nodes in the cluster, making it robust to changes in container placement and node memberships. SurgeGuard consists of two complementary units. **FirstResponder** embodies the first insight and provides a fast path for managing very short surges using a kernel module that quickly identifies QoS violations using *per-packet metrics instead of relying on aggregated or averaged metrics over time*. **Escalator** utilizes the second and third insights to create a slower-path user-space controller that efficiently allocates resources for longer, yet still transient violations. Escalator *introduces a novel per-service metric that separates an observed QoS violation into two critical components*: slowdown from the service itself and slowdown resulting from insufficient processing down-stream from the service. This metric is both effective and cheap to compute, enabling Escalator to avoid over-provisioning a service that is experiencing a surge only because of downstream effects. Escalator can correctly identify the root cause of a surge even when the bottleneck service does not itself exhibit any QoS violations. Furthermore, Escalator *also considers the sensitivity of each service to an increased resource allocation*

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00 ©2024 IEEE

when provisioning resources in response to a surge. We emphasize that the novelty of Escalator is specifically with respect to handling surges rather than attaining better steady state performance compared to prior work.

In contrast to SurgeGuard, existing resource management approaches do not identify needed additional resources correctly or respond too slowly. Existing work can be broadly divided into two categories: heuristic- and ML-based controllers. Heuristic controllers (e.g., Parties [16], Caladan [18], and Dirigent [35]) rely on averaged metrics and are similar to Escalator in response time (Table I). However, these existing controllers treat each container in isolation and ignore inter-container dependencies, often wrongly identifying the root cause of a violation [20] and hence making ineffective allocation decisions. We solve this issue with our new metrics.

ML-based controllers learn dependencies between containers and manage end-to-end, rather than per-container QoS targets [19], [21], [33]. While they correctly identify the root causes of violations and manage resources well for steady-state QoS behavior, they suffer from significant latency overheads that make them unable to react to rapid changes. ML-based controllers require low-noise metrics and use a centralized inference server to make and enforce allocation decisions. This incurs significant inter-node communication overheads for collecting container metrics and sending decisions, resulting in a decision granularity on the order of seconds even when the inference itself takes tens to hundreds of milliseconds [33].

SurgeGuard is able to effectively match the correct root-cause identification of the ML approaches with Escalator while exceeding the decision granularity of heuristic approaches with FirstResponder. We summarize our major contributions below:

- We identify key limitations of prior approaches in managing microservice QoS during transient surges (III).
- We show that commonly used microservice threading models (II-A) induce hidden dependencies that are not handled by existing per-container management algorithms. We present new metrics and algorithms that enable correct upscaling even in the presence of these hidden dependencies (III-B).
- We demonstrate the importance of resource sensitivity for making effective allocations and present a low-overhead technique to enable sensitivity-aware scaling (III-C).
- We use our insights to design SurgeGuard: a controller specifically designed to effectively detect and mitigate both short (IV-A) and long load surges (IV-B).
- We introduce *violation volume*: a new metric that accounts for both the magnitude and duration of QoS violations while evaluating controller performance (II-D).
- We evaluate SurgeGuard and show that it significantly reduces the violation volume from large transient surges vs. the well known Parties (61.1% avg. reduction) and Caladan (93.7% avg. reduction) algorithms (VI).

[1]Caladan relies on a custom networking stack with visibility into runtime queues; its update interval is far larger with the Linux networking stack.

Controller Type	Controller	Dependence Aware?	Distributed?	Update Interval
ML	Sinan	Yes	No	>1s [19]
Heuristic	PARTIES	No	Yes	500ms [16]
	Caladan∗[1]	No	Yes	5-20us [18]
	SurgeGuard	**Yes**	**Yes**	**∼0.2ms**

TABLE I: Comparison of SurgeGuard with existing controllers.

II. Motivation and Related Work

We briefly overview microservices, and policies used for QoS management of monolithic and microservice applications.

A. Microservice-Based Applications

In a microservice based architecture, applications are partitioned into short, fine-grained components called services. Services are deployed in individual containers, and communicate with each other using remote-procedure calls (RPCs). An incoming user request triggers a sequence of RPC calls based on the application logic (Fig. 2). The services and the flow of RPCs between them comprise the *application task-graph*.

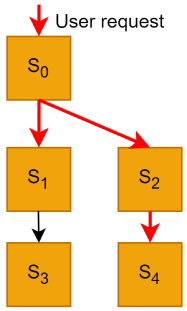


Fig. 2: A simple microservice application task graph

Microservice Threading or Connection Models. Current RPC frameworks provide two major models to maintain inter-service connections. The first model, called *connection-per-request* in this paper, opens a new connection or application thread for each new RPC between containers, incurring the cost of creating a connection on every request. The second, called *fixed-size threadpool* in this paper, maintains a fixed size pool of opened connections for each container. This amortizes the overhead of creating and establishing a new connection and is hence recommended for high request rate deployments [3]. These threading models are independent of the inter-service communication protocol and can be used both for synchronous (e.g., `grpc sync`) and asynchronous (e.g., `grpc async`, message queues) communication.

The threadpool size can be provisioned using Little's Law:

$$\text{ThPoolSize} = \text{DesiredReqRate} * \text{DownstreamLatency} \quad (1)$$

B. Resource Management for Microservices

Resource scaling techniques for microservices fall into two categories: *horizontal* scaling (or autoscaling) [2], [4], [34] and *vertical* scaling. As SurgeGuard is a vertical scaling controller, we focus on vertical scaling in this paper, and discuss the interactions of SurgeGuard with horizontal scaling in VII.

Vertical Scaling. Vertical scaling responds to QoS violations by allocating more resources to containers on a given machine in response to a QoS violation. Vertical scaling approaches draw heavily from resource management controllers developed for monolithic applications to allocate shared resources like cores [16], [25], [30], [33], the last-level cache (LLC) [16], [25], [30], [33], [35], frequency (power) [16], [35], memory bandwidth [22], and IO bandwidth [32].

Parties [16] proposes a resource controller to manage allocations for multiple latency-critical jobs. Caladan [18] and Shenango [29] use network queueing delay to rapidly make upscaling decisions. CliTE [30] uses Bayesian optimization to find resource allocations. Dirigent [35] uses progress tracking to reallocate resources at fine timescales. Parslo [27] uses gradient descent to automatically calculate per-service targets from end-to-end targets. Balm [22] extends previous work by allocating memory bandwidth along with cores and the LLC.

The above approaches treat each container in isolation and do not to capture inter-container dynamics and communication patterns, resulting in ineffective upscaling decisions [20]. ML techniques like Sinan [33] and Sage [19] use supervised learning to determine efficient resource allocations from end-to-end performance targets. They solve the prior issues at the cost of a high detection and inference overhead making them unsuitable for managing transient violations [19], [21].

C. Request Rate Surge Management

The microservice architecture is particularly appealing for user-facing applications that encounter sudden *surges* in the request rate (load) [6], [11] because it promises the ability to quickly scale up to meet demand [2]. However, current resource manager cannot respond fast enough to the large surges observed in recent studies [9], [14] (the average request rate during a surge is $2-3\times$ higher than the base request rate, with the instantaneous request rate being much higher [7], [8]). As a result, current systems resort to less-than-ideal techniques, including provisioning for maximum load [8], rate limiting requests [9], [10], and request batching [9], [10]. However, these methods waste resources due to overprovisioning [8], degrade application QoS by inducing more retries and longer latencies [7], [9], or only work for predictable surges [8].

Our work, SurgeGuard, uses vertical scaling to efficiently manage resources during surges without incurring the overheads of these heavy approaches.

D. Violation Volume Metric

We design a new metric to report the impact of a QoS violation – *violation volume*. We define the violation volume as the violation magnitude-duration product, illustrated as the total area of the output latency graph that lies above the desired QoS target (Fig. 3). Violation volume accounts for both the duration and the magnitude of the QoS violations in a unified metric, providing a more complete picture while comparing the performance of QoS management controllers as compared to just using tail latency (which ignores the violation duration) or violation frequency (which ignores the violation magnitude).

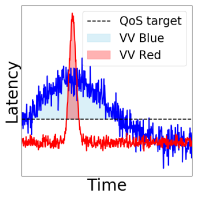


Fig. 3: Violation Volume (VV): VV Red < VV Blue, though red has a higher tail latency.

III. SurgeGuard Design Features

We present three key insights that explain why existing resource controllers for microservices fall short in meeting the desired QoS for applications and discuss how these guide the design of the core features of SurgeGuard.

A. Fast & Efficient Detection of QoS Violations

Detecting QoS violations quickly is important, especially for violations resulting from brief load or execution surges. During the surge and before detection, queues build up without any mitigation. This increases both the magnitude and duration of missed deadline surges, requiring extra resources to process the queued requests after mitigation is eventually applied. Fig. 4 shows a simple example comparing the violation volume and core allocation required to tackle a surge with a 4s duration using an ideal controller that, on detecting a surge, allocates the exact amount of cores needed to overcome it (instead of increasing allocations step-by-step as in real controllers). We observe that a detection delay of 1s (typical for ML-based controllers) results in a violation volume that is $4.75\times$ larger as compared to a delay of 0.5s (typical for Parties) and $24\times$ larger as compared to a delay of 0.2ms, while also needing $40-75\%$ more cores to manage the spike.

Prior Approaches. Existing approaches for identifying QoS violations are slow because they use averaged metrics such as execution time, queueing delay and performance counter information to make allocation decisions. To achieve stable results, these metrics must be averaged over many samples [16], [18], [33], increasing the detection latency and reducing the sensitivity of the violation detector. Additionally, ML based controllers need to report container metrics to a centralized inference server, which adds a significant communication latency to the detection time.

Design Feature #1: SurgeGuard rapidly detects QoS violations to avoid otherwise-costly queue buildup. We design a kernel module (FirstResponder) that tracks and detects QoS violations using *per-packet progress* instead of using averaged metrics. FirstResponder hooks onto the earliest point in the receiver-side network stack (`netif_receive_skb` [5] in Linux). It intercepts each incoming packet, reads the progress information, detects QoS violations (if any), and then forwards the packet for further network processing. The FirstResponder design ensures that QoS violations can be detected as early as possible with very low overhead (~0.2us/packet). Further details on FirstResponder are provided in IV-A.

B. Threading-Model Aware Upscaling

As shown in II-A, current RPC frameworks provide two major models to maintain inter-service connections: connection-per-request and fixed-size threadpool. The choice of threading/connection models introduces *hidden inter-container dependencies* which complicate resource management.

As an example, consider the effect of a request rate Surge on a two-service (c1 and c2) application. If c1 uses a connection-per-request model (Fig. 5(a)), the higher request rate spawns more RPC threads in both services, increasing

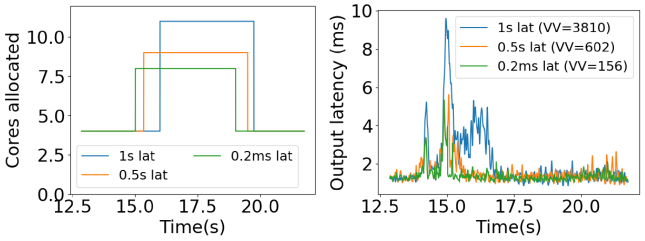


Fig. 4: Slower detection (detection delay = 1s) causes an increase in the violation volume (VV) compared to faster detection (delay = 0.5s), requiring more resources to process the queued requests.

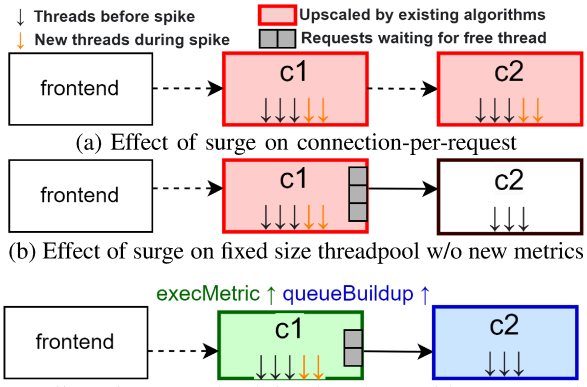


Fig. 5: Impact of threading model: Dashed arrows denote thread-per-request while solid arrows denote fixed-size threadpools. Existing controllers upscale correctly in (a) but fail to upscale c2 when using fixed size threadpools (b). The new SurgeGuard metrics correctly upscale both c1 and c2 in response to a Surge (c).

thread contention and hence the execution time of both c1 and c2. However, with a fixed-size threadpool (Fig. 5(b)), the concurrency between c1 and c2 is fixed, and the higher request rate is not seen by the downstream container (c2). The extra requests rapidly queue up in c1 waiting for the threadpool to return a free connection, thus increasing the execution time of c1, while the execution time of c2 remains unchanged. Note that the threadpool-induced queueing is *implicit* – it takes the form of several application threads either constantly polling for an available connection or waiting to be woken up by an interrupt once a connection becomes available.

Prior Approaches. Existing per-container controllers work well for the thread-per-request model (Fig. 5(a)), correctly scaling both c1 and c2 to manage the request surge. However, due to the hidden dependencies introduced by the fixed thread-pool model (Fig. 5(b)) these controllers only upscale c1 and still miss the deadlines because c2 is provisioned for the initial (lower) request rate. Controllers like Shenange [29] or Caladan [18] that use explicit network queueing metrics (e.g. queueing time/occupancy) also fail to detect the *implicit* threadpool induced queues and cannot upscale containers correctly.

Existing ML-based controllers like Sinan [33] or Sage [19] use complex models to infer inter-container dependencies. However, as before, their long detection and inference time makes them too slow for managing transient violations.

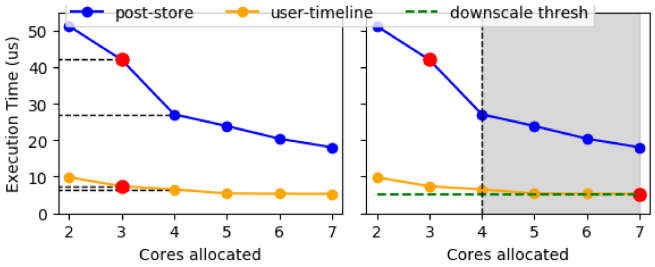


Fig. 6: Execution time graph vs. number of allocated cores (*aka. sensitivity curve*) for two services of socialNetwork, with current allocations in red. In the left graph, latency is more effectively decreased by upscaling post-store irrespective of the relative slowdown of the two services. In the right graph, the execution time of user-timeline is just above the downscale threshold: user-timeline holds on to 7 cores, though allocating 4 cores provides similar performance.

Design Feature #2: To rapidly and correctly manage QoS violations, SurgeGuard uses novel threading-model aware upscaling metrics that account for inter-container dependencies. We introduce two new metrics: (1) an execution time metric (*execMetric*) decoupled from the time waiting for a free connection, and (2) a queue-buildup metric (*queueBuildup*) that is attributed to downstream slowdowns.

- *Execution Metric (execMetric):* This excludes the time spent waiting for a free connection from the total container execution time:

$$execMetric = \text{execTime} - \text{timeWaitingForFreeConn} \quad (2)$$

An increase in *execMetric* reflects a true slowdown in a container, and indicates that it should be upscaled (Table II). While this metric is sufficient to upscale containers using a thread-per-request model, it cannot detect hidden dependencies, as the time spent waiting for a free connection is not accounted for in any container.

- *Queue Buildup Metric (queueBuildup):* This metric is designed to account for the hidden inter-container dependencies.

$$queueBuildup = \frac{\text{execTime}}{execMetric} \quad (3)$$

A large value of *queueBuildup* at a container indicates that more time is lost to the hidden dependencies (waiting to obtain a free connection or thread), showing that the downstream throughput is unable to keep up with upstream throughput. Hence, on detecting an increase in *queueBuildup* at a container, *downstream* containers should be upscaled to reduce downstream latency and increase downstream throughput (Table II).

Fig. 5(c) shows how these metrics allow SurgeGuard to correctly upscale both c1 and c2 in the presence of hidden dependencies. The surge increases the thread concurrency and hence *execMetric* at c1, resulting in c1 being upscaled. The threadpool-induced queueing at c1 increases *queueBuildup*, resulting in the downstream container (c2) being upscaled.

C. Resource-Sensitivity Aware Allocations

Per-container management approaches attempt to meet QoS targets for each container by allocating additional resources

to the containers experiencing a violation. However, this can be wasteful: services that marginally benefit from additional resources can be prioritized over those that can benefit significantly and better help meet end-to-end QoS targets (Fig. 6(left)). Additionally, when a container experiencing a violation has a nearly flat sensitivity curve, these controllers are extremely sensitive to the downscale threshold and are unable to quickly reverse allocations made in response to a transient violation: containers are allowed to hog a large amount of extra resources for little benefit (Fig. 6(right)).

Prior Approaches. Previous approaches are designed for monolithic applications and rely on extensive profiling to find sensitivity curves [17]. However, the sensitivity curve of a microservice depends on several additional factors such as request rate, application bottlenecks, allocations of other services etc., which exponentially increases the profiling overhead and makes existing profiling based techniques impractical.

Design Feature #3: SurgeGuard determines resource sensitivities with minimal overhead and collectively considers containers across the task-graph while making allocations. We design a very low overhead online-profiling technique to find the sensitivity information for containers at run time.

To do this, we create an array (execAvg) that stores the exponential running average of the execution time of each container for each observed allocation. These values are updated every time SurgeGuard reads the updated per-container metrics shared by the containers(red arrows in Fig. 7).

$$\text{execAvg}[\text{container}][\#\text{cores}] = \alpha * \text{execAvg}[\text{container}][\#\text{cores}] + (1-\alpha) * \text{newObservedTime}[\text{container}]$$

The sensitivity value (sens) for any number of cores can then easily be calculated by looking up the percentage reduction in average execution time achieved by allocating an extra core.

$$\text{sens}[\text{container}][\#\text{cores}] = 1 - \frac{\text{execAvg}[\text{container}][\#\text{cores}+1]}{\text{execAvg}[\text{container}][\#\text{cores}]}$$

We use a large value of α ($\alpha = 0.5$) to weight newer execution times quite heavily, ensuring that sensitivity values account accurately for current conditions. We use it to prevent containers from hogging cores by periodically revoking a core from containers where the execAvg matrix indicates that it will not have a significant impact on the execution time (revoking a core if sens[container][#cores-1]<0.02 works well). Additionally, we use it during upscaling by preferentially allocating cores to containers where the sens matrix shows a high sensitivity to core allocations.

IV. SurgeGuard Architecture

SurgeGuard manages two resources: core allocations and per-core frequency[2]. Fig. 7 shows the overall architecture of SurgeGuard. SurgeGuard consists of two complementary units: a kernel module (FirstResponder) that manages frequency and a user-space controller (Escalator) that manages both frequency and core allocations.

[2]SurgeGuard can be easily extended to other resources (§7), we leave them out here to focus on the novel algorithm and design aspects of SurgeGuard.

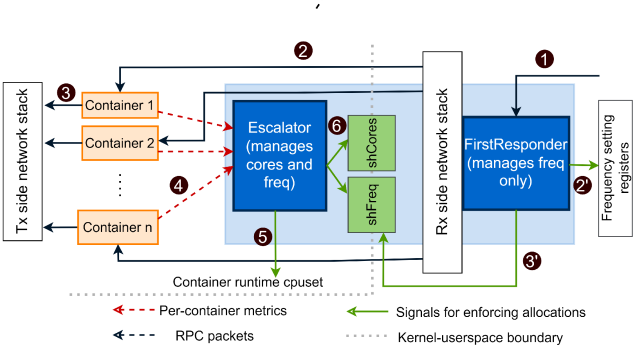


Fig. 7: High level block diagram of SurgeGuard, showing the sequence of operations on receiving an RPC packet.

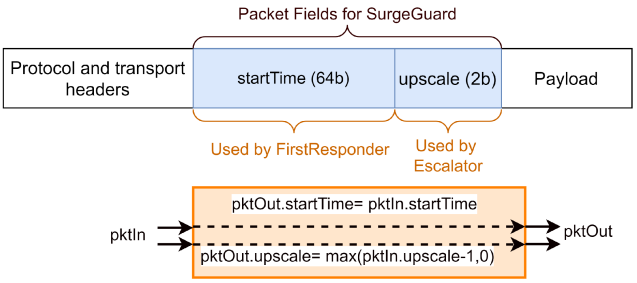


Fig. 8: Additional metadata fields for SurgeGuard.

FirstResponder implements Design Feature #1, using per-packet progress tracking to provide a *fast path for detecting and managing extremely short violations*. Escalator complements FirstResponder by managing resource allocations during *longer QoS violations*. Escalator implements efficient resource allocations using Design Features #2 and #3. SurgeGuard does not specify any particular resource-allocation policy per se, and we use that of Parties in our evaluation. SurgeGuard requires some extra metadata fields in the RPC packets (Fig. 8), which are described in greater detail below.

Fig. 7 shows the basic operation of SurgeGuard. When an RPC packet is received ❶ at the receive (rx) side of the network stack, it is first intercepted by FirstResponder. FirstResponder reads some SurgeGuard metadata fields from the RPC packet (described below) and calculates the *per-packet slack* which it uses to determine QoS violations. It then forwards the packet to the normal network processing path which routes the RPC to the appropriate containers ❷. If a QoS violation is detected by FirstResponder, it updates the frequency of the desired containers ❷'. FirstResponder periodically writes the updated frequencies to a shared memory region (shFreq) to synchronize state with Escalator ❸'. This forms the fast path of detecting and managing QoS violations.

Once the containers finish executing, the container runtimes calculate the SurgeGuard metadata fields for any outgoing RPCs ❸ as described below. The runtimes also calculate averaged metrics and periodically communicate them with Escalator using shared files/pipes ❹. Escalator uses these metrics to update core and frequency allocations ❺, and periodically synchronizes state with FirstResponder by writing updates to shCores and shFreq ❻. This forms the slower and

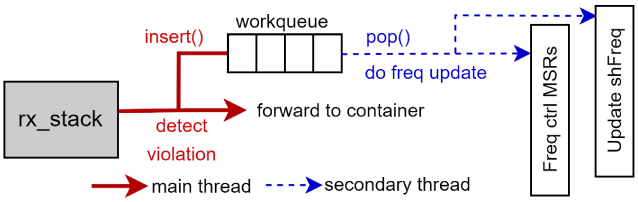


Fig. 9: FirstResponder uses a coordinator-worker design to shift the long latency of frequency updates off the network stack critical path.

more precise path of detecting and managing QoS violations.

SurgeGuard is carefully designed to be completely decentralized (Fig. 1). Decisions are made locally at each node based on the preset parameters, observed incoming requests, and locally-computed metrics. SurgeGuard does not require any visibility into or explicit communication with SurgeGuard modules on other nodes.

SurgeGuard Parameters. SurgeGuard uses two configurable parameters for each container to make allocation decisions: the expected execution time (expectedExecMetric) and the expected elapsed time since the start of the job (expectedTimeFromStart). These values can either be set by the user or obtained through online profiling. Following the approaches of Dirigent [35] and Nightcore [23], we set these values by profiling the application at low load and setting the parameters to twice the values measured at low load.

SurgeGuard Metadata Fields. To implement progress-tracking and threading-model aware allocation, SurgeGuard adds two metadata fields to each RPC packet (Fig. 8). The *startTime* field denotes the starting timestamp of the job and is used by FirstResponder for progress tracking. It is set in the first container and propagated unchanged by the remaining containers (Fig. 8). The *upscale* field is used as an upscaling hint by Escalator to upscale downstream containers in response to a QoS violation (II-C"Queue Buildup Metric"). A container is considered for upscaling if pktIn.upscale>0 (Table II). This field is set at the container where the violation is detected, and propagated by subsequent containers (Fig. 8), decreasing by 1 at each successive container. This ensures that only a limited number of downstream containers are upscaled in response to an upstream QoS violation. This design allows SurgeGuard to be completely decentralized – upscaling hints piggyback on data packets and are automatically sent across nodes to the appropriate containers without requiring SurgeGuard to possess any global knowledge.

A. FirstResponder

FirstResponder implements Design Feature #1 (§3.1) and tracks *per-request slack* at each container.

Per-Packet Slack Calculation. We determine slack by calculating the difference between the expected and observed progress towards the end-to-end target:

$$\text{slack} = \text{expectedTimeFromStart} - \text{observedTimeFromStart} \quad (4)$$

$$\text{observedTimeFromStart} = \text{currentTime} - \text{pkt.startTime} \quad (5)$$

TABLE II: Upscaling candidates based on violation cause.

Detected condition at container c	Upscaling candidates
pkt.upscale > 0	Container c
queueBuildup violation	Downstream containers, set pkt.upscale
execMetric violation	Container c

Detecting a negative slack at any container indicates that the request progress is lagging, and an end-to-end QoS violation is likely unless downstream containers are upscaled.

FirstResponder Implementation. We implemented FirstResponder as a per-node host-side kernel module. This allows us to read the packet fields for the slack calculation without requiring an additional kernel-userspace crossing, and hence minimizes the latency and overhead of FirstResponder.

FirstResponder hooks into the receive side of the networking stack, intercepting each incoming packet before it is routed to its destination container. It reads the startTime field from the packet and calculates the per-request slack. If the slack is negative, it detects a QoS violation and upscales the frequency of both the violating container and downstream containers present on the same node. It operates on a per-request basis instead of using averaged statistics, upscaling containers as soon as a violation is detected on any request.

As FirstResponder lies on the critical path of the kernel-side networking stack, it is carefully designed to minimize added latency. FirstResponder uses a two thread coordinator-worker design (Fig. 9). The main thread lies on the critical path and detects slack violations. On detecting a violation, it inserts a frequency update work item in the work queue of the secondary thread. Meanwhile, the secondary thread lying off the critical path polls its work queue for work items. It pops each work item off of the work queue and performs the frequency update by writing to machine-specific registers (MSRs). The secondary thread then informs Escalator of the update by writing the updated frequency to shFreq.

Mitigating Frequent Updates. Because FirstResponder does not compute averages, the measured slack can be noisy and lead to unnecessary updates. To mitigate this, once an upscaling decision is made for a path through the task-graph, FirstResponder does not change the frequency for that path any further for a time window (∼2x of the end-to-end request latency worked well). This reduces the rate of frequency modifications and reduces the overhead incurred by the coordinator and worker threads.

B. Escalator

Escalator is a user-space controller that implements Design Features #2 and #3 to identify threading- and sensitivity-aware upscaling and downscaling candidates. *Escalator's contribution lies in our techniques for determining these candidates, not in deciding which resources to allocate.* In this paper, we combine our candidate identification mechanisms with an existing resource allocation algorithm (Parties) to provide a complete resource management solution.

At the start of each decision cycle, Escalator reads the per-container metrics collected by the container runtimes (Fig. 7). It then assigns a score to each container indicating the importance of scaling it. For this, Escalator checks three conditions: (1) whether an pkt.upscale hint has been received due to an upstream *queueBuildup*, (2) whether *queueBuildup* at this container exceeds a threshold (`QUEUE_TH`), and (3) whether (*execMetric*/expectedExecMetric) exceeds a threshold (`EXEC_TH`). If any of these conditions is true, the score of the upscaling candidates (as determined from Table II) is incremented by 1. This ensures that containers failing more checks have higher scores than containers failing fewer checks.

Once the upscaling candidates are identified, Escalator can use any algorithm to decide upscaling preferences and how many extra units of each resource should be allocated to each candidate (we used the Parties algorithm). While upscaling resources, we first prioritize containers with the higher scores. Among containers with the same score, we prioritize the containers with the highest sensitivity to cores and allocate one core at a time to these containers.

While deallocating resources, we first deallocate resources from containers with a score of zero, i.e., those that are not marked as upscaling candidates by any condition. We use the Parties algorithm to determine downscaling priorities and to make decisions on which resource should be deallocated. If there are no containers with a score of zero, we use sensitivity-based revocation (Design Feature #3) to revoke cores from containers with low sensitivity to core count.

V. Experimental Setup

Experimental Setup. We conduct all our experiments on Chameleon [24], which provides reconfigurable bare-metal nodes in a cloud-like environment. We use a cluster of four 2-socket Intel Xeon 6242 (Cascade Lake) server nodes, each with 64 logical cores and 224GB of memory running Ubuntu-18.04. The client runs on a separate 6-core Intel Xeon node. We use the open-loop workload generator `wrk2` [13] as our client, and modify it to generate input load spikes. We we use the default `intel_pstate` governor except when controlling per-core frequencies with the `userspace` frequency governor. We measure the energy using `perf`, subtracting the idle energy consumption to accurately measure the energy consumption by the application.

On each node, we use 3 logical cores for SurgeGuard, 16 cores for network processing and other OS tasks, and 52 logical cores for the workload. We initialize per-container allocations to achieve the highest steady-state throughput (request rate) using a total of 34 cores for the foreground application. The remaining 18 cores can be allocated on demand by the controllers for managing surges; In deployment, these cores could be used to run background applications while the microservice workload is in steady state.

Controllers Evaluated. We evaluate three controllers:

- Parties [16]: We implement the Parties controller in C++ following the code open-sourced by the authors.
- CaladanAlgo [18]: We implement the Caladan algorithm as a userspace controller. Since we do not use Caladan's custom networking stack, and lack visibility into the network queues, we use our proposed *queueBuildup* metric for the queueing delay measurement of CaladanAlgo.
- SurgeGuard: Our proposed SurgeGuard.

We allow CaladanAlgo to allocate hyperthreads individually. For the other controllers, we always allocate both the hyperthreads of a physical core to the same container. We set the same per-container QoS limits for all three controllers, following the methodology of IV "SurgeGuard Parameters".

TABLE III: Details of the evaluated workloads. Threadpool size of ∞ denotes connection-per-request model.

Workload	Action	Task-graph Depth	RPC	Threadpool Size
`CHAIN`	-	5	Thrift	512
socialNetwork	`ReadUserTimeline`	5	Thrift	512
	`ComposePost`	8	Thrift	512
hotelReservation	`searchHotel`	11	gRPC	∞
	`recommendHotel`	5	gRPC	∞

Workloads. *Benchmarks:* We use realistic publically available microservice workloads – socialNetwork and hotelReservation from DeathStarBench [20]. We use two actions for each of these workloads, representing a range of task-graph sizes, RPC frameworks and threading patterns. Table III shows further details about the chosen workloads.

`CHAIN` Microbenchmark: We also evaluate a microbenchmark that comprises a chain of five services, each performing arithmetic work (a large vector accumulate). It uses the same threading and connection models as the Thrift-based workloads in DeathStarBench [20].

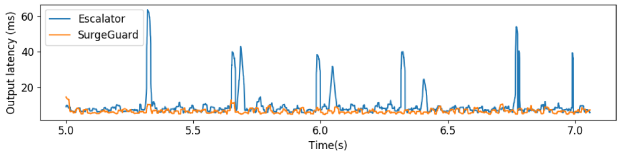
VI. Evaluation

We evaluate SurgeGuard on a range of surge scenarios. We detail our results in the following subsections, finding that:

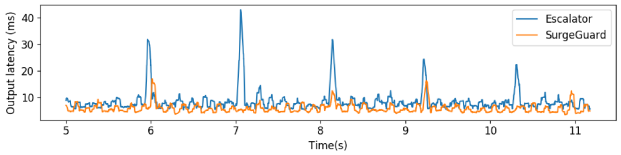
- FirstResponder successfully manages short surges, reducing the latency during surges by ∼40x for 100us long surges, and ∼3x for 2ms long surges (VI-A).
- Escalator efficiently manages longer surges, reducing violation volume by an average of 19-61% while requiring 2-8% fewer cores than Parties(VI-B).
- Each Escalator mechanism contributes to reducing the violation volume (VV): new metrics reduce it by 12.5%, sensitivity-based allocations by 49%, and combining them reduces the VV by 74% across workloads (VI-B).
- SurgeGuard scales well to multiple nodes, reducing violation volume by 39%, and using 16% fewer cores than the baselines (VI-B) when using 4 nodes.

A. Managing Short Surges With FirstResponder

We demonstrate the effectiveness of FirstResponder on short surges by comparing the performance of Escalator to the complete SurgeGuard consisting of Escalator+FirstResponder. Fig. 10 shows the timeline graphs for two different surge durations: 100us and 2ms, demonstrating different aspects of the benefit provided by FirstResponder. Escalator (and other averaging based controllers) are unable to detect very short



(a) Surge Duration = 100us, Surge Freq. = 1 surge/0.5s



(b) Surge Duration = 2ms, Surge Freq. = 1 surge/1s

Fig. 10: Timeline graphs for short request rate surges using CHAIN. The instantaneous request rate during the surge is 20x the request rate outside the surge. FirstResponder decreases the violation volume of such short surges by 98% for (a) and 88% for (b) compared to Escalator alone.

surges (Fig. 10(a)), resulting in a large increase in the output latency. The per-packet violation detection of FirstResponder is able to rapidly detect and increase core frequency in response to the surge, absorbing the increase in the request rate without incurring any increase in the output latency.

As the duration of the surge grows (Fig. 10(b)), the resulting increase in the averaged execution time is detected by Escalator, which allocates more cores to meet the latency target. This can be seen in the latency graph – latency during the surges is lower in Fig. 10(b) compared to Fig. 10(a) despite having a longer surge duration (2ms vs 100us). Even in this case, the fast detection and upscaling of FirstResponder helps mitigate the surge impact till Escalator detects the violation and upscales cores. SurgeGuard absorbs the surges much more effectively compared to just Escalator.

Additionally, Fig. 10(a) and Fig. 10(b) show that the relative benefit provided by FirstResponder decreases as the surge duration grows. This is expected as the early detection of FirstResponder provides a proportionally smaller head-start with an increase in the surge duration.

B. Managing Longer Surges With Escalator

We evaluate SurgeGuard on longer request rate surges by injecting 2s long request rate surges every 10s into the input request rate. We vary the request rate during the surge to 1.25x, 1.5x and 1.75x of the base request rate. We measure the overhead by measuring both the average number of cores used and the energy consumed. *As Escalator provides nearly the entire performance benefit of SurgeGuard for these longer surges (<0.3% performance difference between Escalator and SurgeGuard), we do not separate the performance impact of Escalator and FirstResponder in this subsection.* We present results using our violation volume metric; the results and trends are similar for tail latency (P98 latency) as well.

Single Node Results. Fig. 11 shows that, on average, *SurgeGuard decreases the violation volume by 19% for 1.25x surge, 43% for 1.5x surge and 61% for 1.75x surge while requiring 2-8% fewer cores and 2-4% less energy as compared to Parties.* The average performance of CaladanAlgo is poor, primarily because it has a high violation volume for `searchHotel` and `recommendHotel`. These workloads use a connection-per-request model and do not have any explicit or implicit queues. Hence, the value of *queueBuildup* remains fairly stable even during the surge, resulting in CaladanAlgo being unable to properly upscale resources.

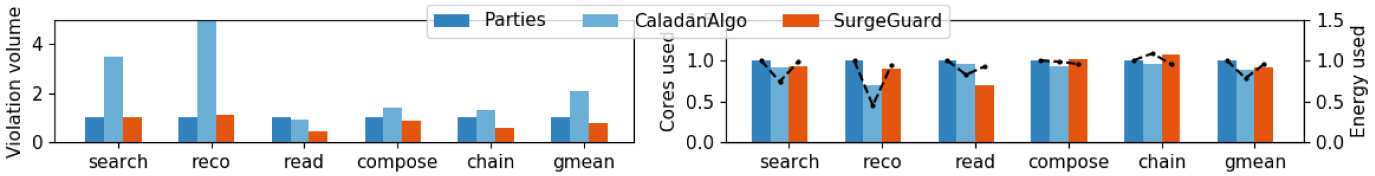
SurgeGuard provides a benefit over Parties for all the evaluated workloads. For `searchHotel` and `recoHotel`, the benefit comes primarily from the better core allocation decisions made with our sensitivity-aware allocations. `CHAIN`, `readUserTimeline` and `composePost` use a limited threadpool model and hence also derive a benefit from the threading-model aware upscaling. We also see that the benefit of SurgeGuard increases as the surge magnitude increases. This is because the impact of the inefficient allocations made by Parties and CaladanAlgo is magnified with larger surges.

SurgeGuard's resource sensitivity-aware allocations also helps reduce the number of cores used, as containers are not allowed to hog cores from which they do not derive a meaningful benefit. However, a reduction in usage is not guaranteed – the freed cores can simply be taken up by other, more core-sensitive containers to manage the increased request rate. Because of these contradictory factors, SurgeGuard achieves a relatively small (2-8% for cores, 2-4% for energy) reduction in resource usage compared to the Parties baseline.

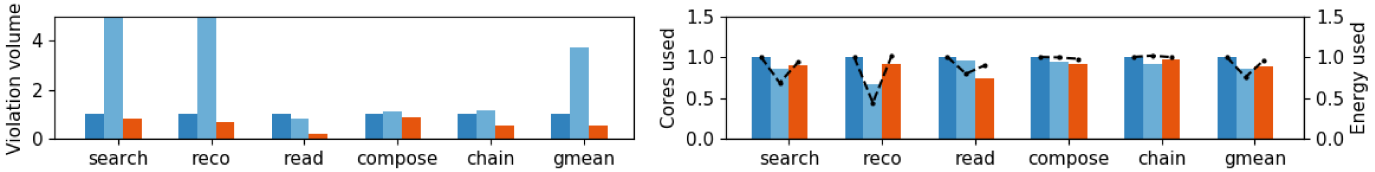
Effect Of Changing Surge Duration. Fig. 12 shows the effect of changing the surge duration from 0.1s to 5s on `recommendHotel` (which uses connection-per-request) and `readUserTimeline` (uses fixed threadpool). The request rate during the surge is set to 1.75x of the base request rate. We see that SurgeGuard outperforms both the baseline controllers for all the surge durations. Additionally, as the surge duration increases, the violation volume of SurgeGuard w.r.t. both Parties and CaladanAlgo also improves (43.4%→56.5% improvement over baseline from 0.1s to 5s). This is due to similar reasons as before – the effect of the efficient core allocations made by SurgeGuard is more prominent for larger surges. As before, the energy used by SurgeGuard vs. the baselines also remains fairly stable (∼1) across surge durations. The exception is `recommendHotel` when using CaladanAlgo as the baseline, where SurgeGuard's energy consumption is 2.5x of CaladanAlgo. This is because, as described before, CaladanAlgo fails to upscale containers when the workload uses the connection-per-request model, resulting in a much lower energy consumption (7.4x lower than SurgeGuard for 5s surge) but also a much higher violation volume (251x higher than SurgeGuard for 5s surge) as compared to SurgeGuard.

Node Scaling. We evaluate the effect of increasing the number of nodes by extending our experiments to 1, 2 and 4 nodes (Fig. 13). We inject a 2s long surge every 10s and set the request rate during the surge to 1.75x of the base request rate.

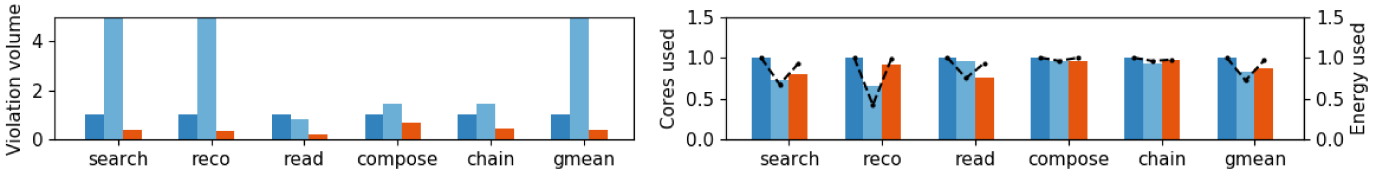
SurgeGuard significantly outperforms both Parties and CaladanAlgo for all configurations. As the number of nodes increases, the resource constraints also decrease and both Parties and CaladanAlgo inefficiently allocate the increasingly



(a) Normalized violation volume (left) and cores and energy used (right) with 1.25x surge.

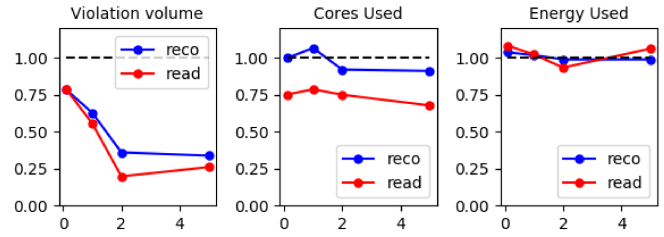


(b) Normalized violation volume (left) and cores and energy used (right) with 1.5x surge.

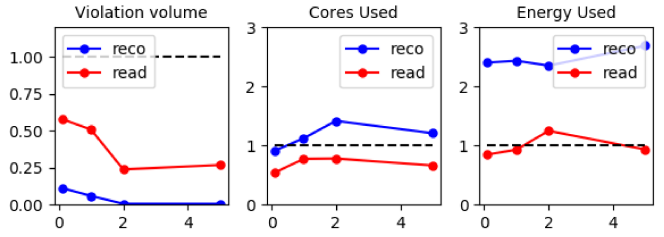


(c) Normalized violation volume (left) and cores and energy used (right) with 1.75x surge.

Fig. 11: Normalized violation volume and resources for different magnitudes of the request rate surge. All results are normalized to Parties. In the graphs on the right, the bars show the cores used while the lines show the energy used. The workload names are abbreviated: search is `searchHotel`, reco is `recommendHotel`, read is `ReadUserTimeline`, compose is `ComposePost`.

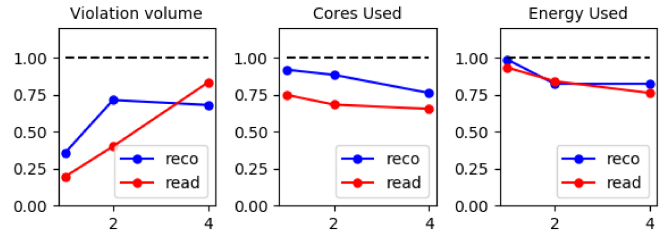


(a) Performance of SurgeGuard normalized to Parties

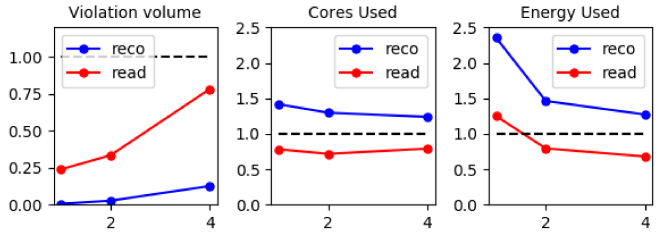


(b) Performance of SurgeGuard normalized to CaladanAlgo

Fig. 12: Effect of varying surge duration from 0.1s to 5s on SurgeGuard, normalized to two baselines - (a) Parties and (b) CaladanAlgo. Values <1 show improvement over baseline.



(a) Performance of SurgeGuard normalized to Parties



(b) Performance of SurgeGuard normalized to CaladanAlgo

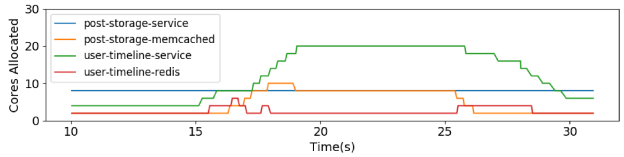
Fig. 13: Effect of increasing number of nodes from 1 to 4 on SurgeGuard, normalized to two baselines - (a) Parties and (b) CaladanAlgo. Values <1 show improvement over baseline.

large number of cores to the application to mitigate the surge. SurgeGuard's efficient resource allocations hence provides an increasing benefit of requiring fewer cores (6.5%→16.4% improvement) and lower energy consumption (14.2%→28.3% improvement) with increasing number of nodes. The benefit of SurgeGuard on the violation volume has an opposite trend – decreasing (67.2%→51.4% improvement) as the number of nodes increases. This is because distributing the application across a larger number of nodes makes it less likely that a particular container hogs a substantial fraction of the cores and prevents other containers from scaling up their allocation.

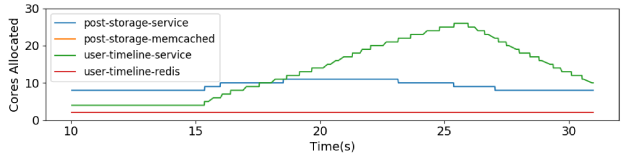
C. Per-Component Benefit Analysis

We explain how our new metrics and sensitivity based resource revocations enable efficient core allocations over time and complement each other to manage application QoS.

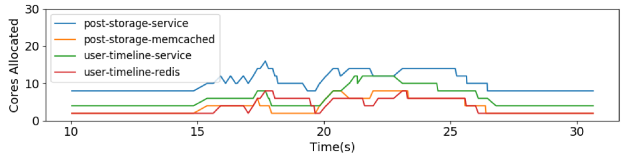
Timeline of Core Allocations. Fig. 14 shows the allocations over time for some services of `readUserTimeline`, injecting a 10s long request rate surge starting at 15s. As



(a) Timeline of core allocations for Parties



(b) Timeline of core allocations for CaladanAlgo



(c) Timeline of core allocations for SurgeGuard

Fig. 14: Core allocations over time for readUserTimeline in response to a 1.75x request rate surge starting at 15s and ending at 25s.

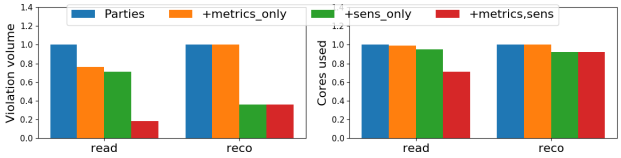


Fig. 15: Performance breakdown of Escalator

`readUserTimeline` uses a fixed-size threadpool model, the extra requests queue up in user-timeline-service waiting for a free connection. As a result, both Parties and CaladanAlgo keep allocating cores to user-timeline-service during the surge, giving it almost 50% of the total cores in the system. While this decreases the latency of user-timeline-service, it is unable to manage the surge as downstream containers (post-storage-service and post-storage-memcached) are starved of cores.

In contrast, SurgeGuard, allocates extra cores both to user-timeline-service and to the downstream containers on detecting the surge at t=15s, spreading out cores more evenly between containers. We also see that SurgeGuard reverses core allocations based on their sensitivity, revoking cores from the containers between 18-20s and 23-25s. These mechanisms allow SurgeGuard to efficiently manage large surges by detecting hidden dependencies, directing resources correctly to the containers that benefit from them, and preventing containers from unnecessarily hogging resources.

Performance Breakdown Of Escalator Mechanisms. We use Parties as the base resource allocator and compare the performance of Parties, Parties with our new metrics, Parties with the sensitivity-based allocations, and the complete Escalator controller using Parties as the base allocator (i.e., including both the new metrics and the sensitivity-based downscaling). Fig. 15 shows the impact of these components on the violation volume and the cores used.

First, `readUserTimeline` exhibits a 23.5% reduction in violation volume with the new metrics, but `recommendHotel` does not benefit. This is because `recommendHotel` uses unlimited threadpools (execMetric=execTime for unlimited threadpools, so there is no impact from the new metrics). `ReadUserTimeline` uses a fixed-size threadpool – using our new metrics allows us to infer the hidden dependencies and improve performance by allocating resources across the task-graph (Fig. 14).

We also see that applying sensitivity-based allocations without using our new metrics decreases both the violation volume (by 28% and 63% for `readUserTimeline` and `recommendHotel`) and the cores used (by 5% and 8%). This is because sensitivity-based revocation frees cores from insensitive containers, allowing them to be allocated to containers that derive a larger benefit. However, without using the new metrics, the freed cores are quickly reallocated to user-timeline-service by the Parties allocator, which limits the benefits achieved by the sensitivity-based downscaling. Combining these mechanisms together into Escalator compounds their benefits – the sensitivity based revocations free extra cores from insensitive containers while the new metrics account for hidden dependencies to correctly upscale candidates across the task graph, resulting in significantly higher performance than when using either mechanism alone.

D. SurgeGuard Overheads

SurgeGuard requires three cores on each node: one for Escalator, and one for each of the two threads of FirstResponder. Having dedicated cores for the two FirstResponder threads ensures that the latency critical primary thread does not contend with the secondary thread. We assign the two FirstResponder threads to hyperthreads on the same physical core to lower the latency of access to the shared workqueue. In all the experiments, the CPU utilization of the cores allocated to SurgeGuard remains below 3%.

The primary thread of FirstResponder adds a latency of 0.26us on the packet processing path ($< 0.5\%$ of the packet processing time). Adding a workitem in the workqueue shared between the primary and secondary threads requires an additional 0.44us. The secondary thread requires 2.1us to read an item from the workqueue and update the frequency MSR – this latency lies off the application processing path and does not contribute to any overheads. The overheads are negligible overall and adding FirstResponder does not change the load-latency curve of the application at steady state.

VII. Discussion

Interaction with Other Controllers. SurgeGuard is designed to rapidly detect QoS violations and make efficient allocations. While it is able to capture complex inter-service dynamics to a large extent, it does not have the expressiveness of ML/gradient-search algorithms like Sage or Sinan. We envision that these heavier techniques periodically set steady-state allocations, while SurgeGuard manages allocations in between to tolerate surges. The heavy ML models can thus be run less frequently. This saves system resources, cuts communication costs, and supports a longer decision interval

(enabling more accurate and complex ML controllers) without negatively impacting the QoS. Also, the Escalator algorithms for selecting up/down-scaling candidates can be integrated with existing autoscalers and controllers like Shenango [29] that rely on optimized networking stacks.

Interaction with Autoscaling Algorithms. Similar to the benefits SurgeGuard provides to ML-based controllers, we expect the ability of SurgeGuard to better tolerate surges to also benefit horizontal-scaling controllers, by managing QoS and preventing request buildup while the autoscaler launches a new container. Autoscaling algorithms can also benefit from our insights and new metrics to identify scaling candidates.

Extending SurgeGuard to Other Resources. SurgeGuard can be easily extended to manage resources beyond cores and frequency. As FirstResponder is designed to respond to very short spikes, it can manage any resources that can be quickly upscaled and have an immediate impact on the execution time (e.g. memory bandwidth for bandwidth constrained services). On the other hand, Escalator's candidate selection can be used to manage any resource by combining it with any existing resource allocation algorithm.

VIII. Conclusion

We identify a number of issues with existing microservice resource management schemes and use our insights to design SurgeGuard. SurgeGuard is designed to be fast, lightweight, and to require minimal workload changes and profiling information. We evaluate SurgeGuard using the Parties algorithm as a base; however, our solutions are widely applicable and can be used by a wide range of resource controllers and autoscaling algorithms in the future for managing QoS during surges.

IX. Acknowledgements

We thank our reviewers for their insightful feedback. We also acknowledge the Texas Advanced Computing Center (TACC) at The University of Texas at Austin for providing computational resources that have contributed to the research results reported within this paper. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation. Additionally, this work is supported in part by the NSF grant #2212579.

References

[1] Amazon microservices. [Online]. Available: https://aws.amazon.com/microservices/

[2] Aws autoscaling. [Online]. Available: https://aws.amazon.com/autoscaling/

[3] "grpc best practices." [Online]. Available: https://grpc.io/docs/guides/performance/

[4] Kubernetes. [Online]. Available: https://kubernetes.io/

[5] Linux network receive function. [Online]. Available: https://docs.kernel.org/networking/kapi.html#c.netif_receive_skb

[6] Netflix microservices. [Online]. Available: https://netflixtechblog.com/the-netflix-cosmos-platform-35c14d9351ad

[7] Spike management at aws. [Online]. Available: https://aws.amazon.com/blogs/database/running-spiky-workloads-and-optimizing-costs-by-more-than-90-using-amazon-dynamodb-on-demand-capacity-mode/

[8] Spike management at facebook. [Online]. Available: https://engineering.fb.com/2018/02/12/production-engineering/how-production-engineers-support-global-events-on-facebook/

[9] Spike management at netflix. [Online]. Available: https://netflixtechblog.com/migrating-critical-traffic-at-scale-with-no-downtime-part-1-ba1c7a1c7835

[10] Spike management at twitter. [Online]. Available: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2022/stability-and-scalability-for-search

[11] "Twitter microservices." [Online]. Available: https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/rebuild_twitter_public_api_2020

[12] Uber microservices architecture. [Online]. Available: https://www.uber.com/blog/crisp-critical-path-analysis-for-microservice-architectures

[13] wrk2. [Online]. Available: https://github.com/giltene/wrk2

[14] S. Boucher, A. Kalia, D. G. Andersen, and M. Kaminsky, "Putting the" micro" back in microservice," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 645–650.

[15] R. Chen, J. Wu, H. Shi, Y. Li, X. Liu, and G. Wang, "Drlpart: A deep reinforcement learning framework for optimally efficient and robust resource partitioning on commodity servers," in *Proceedings of the 30th International Symposium on High-Performance Parallel and Distributed Computing*, 2021, pp. 175–188.

[16] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 107–120.

[17] N. El-Sayed, A. Mukkara, P.-A. Tsai, H. Kasture, X. Ma, and D. Sanchez, "Kpart: A hybrid cache partitioning-sharing technique for commodity multicores," in *2018 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 2018, pp. 104–117.

[18] J. Fried, Z. Ruan, A. Ousterhout, and A. Belay, "Caladan: Mitigating interference at microsecond timescales," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 281–297.

[19] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: practical and scalable ml-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 135–151.

[20] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. 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*, 2019, pp. 3–18.

[21] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices," in *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, 2019, pp. 19–33.

[22] D. Gureya, V. Vlassov, and J. Barreto, "Balm: Qos-aware memory bandwidth partitioning for multi-socket cloud nodes," in *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, 2021, pp. 435–438.

[23] Z. Jia and E. Witchel, "Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 152–166.

[24] K. Keahey, J. Anderson, Z. Zhen, P. Riteau, P. Ruth, D. Stanzione, M. Cevik, J. Colleran, H. S. Gunawi, C. Hammock, J. Mambretti, A. Barnes, F. Halbach, A. Rocha, and J. Stubbs, "Lessons learned from the chameleon testbed," in *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20)*. USENIX Association, July 2020.

[25] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015, pp. 450–462.

[26] D. Masouros, S. Xydis, and D. Soudris, "Rusty: Runtime interference-aware predictive monitoring for modern multi-tenant systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 1, pp. 184–198, 2020.

[27] A. Mirhosseini, S. Elnikety, and T. F. Wenisch, "Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices," in *Proceedings of the ACM Symposium on Cloud Computing*, 2021, pp. 442–457.

[28] R. Nishtala, V. Petrucci, P. Carpenter, and M. Sjalander, "Twig: Multi-agent task management for colocated latency-critical cloud services," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 167–179.

[29] A. Ousterhout, J. Fried, J. Behrens, A. Belay, and H. Balakrishnan, "Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 361–378.

[30] T. Patel and D. Tiwari, "Clite: Efficient and qos-aware co-location of multiple latency-critical jobs for warehouse scale computers," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2020, pp. 193–206.

[31] S. Wang, Y.-H. Zhu, S.-P. Chen, T.-Z. Wu, W.-J. Li, X.-S. Zhan, H.-Y. Ding, W.-S. Shi, and Y.-G. Bao, "A case for adaptive resource management in alibaba datacenter using neural networks," *Journal of Computer Science and Technology*, vol. 35, no. 1, pp. 209–220, 2020.

[32] Y. Yuan, M. Alian, Y. Wang, R. Wang, I. Kurakin, C. Tai, and N. S. Kim, "Don't forget the i/o when allocating your llc," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 112–125.

[33] Y. Zhang, W. Hua, Z. Zhou, G. E. Suh, and C. 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*, 2021, pp. 167–181.

[34] Z. Zhou, C. Zhang, L. Ma, J. Gu, H. Qian, Q. Wen, L. Sun, P. Li, and Z. Tang, "Ahpa: adaptive horizontal pod autoscaling systems on alibaba cloud container service for kubernetes," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 37, no. 13, 2023, pp. 15 621–15 629.

[35] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *Proceedings of the twenty-first international conference on architectural support for programming languages and operating systems*, 2016, pp. 33–47.

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

C_1 We identify several key limitations of prior approaches in managing microservice QoS during transient surges. We present new metrics and techniques that enable task-graph and sensitivity aware upscaling for complex task graphs which contain hidden inter-service dependencies.

C_2 We use our insights to design SurgeGuard: a controller specifically designed to effectively detect and mitigate both short and long load surges using different response paths.

C_3 We introduce *violation volume*: a new metric that accounts for both the magnitude and duration of QoS violations while evaluating controller performance.

C_4 We evaluate SurgeGuard on applications from DeathstarBench and our CHAIN microbenchmark and show that it reduces the violation volume from large transient surges vs. the well known Parties (61.1% avg. reduction) and Caladan (93.7% avg. reduction) algorithms.

B. Computational Artifacts

A_1 https://doi.org/10.5281/zenodo.12591422, see `apps` and `controllers`

A_2 https://doi.org/10.5281/zenodo.12591422, see `wrk2_spike`

Artifact ID	Contributions Supported	Related Paper Elements
A_1	C_1, C_2, C_4	Table 1 Figures 9-14
A_2	C_3	Figures 9-14

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This artifact includes the code for SurgeGuard along with the applications for our experiments. The application code is present under `apps`. We take the DeathstarBench source code from https://github.com/delimitrou/DeathStarBench and modify it to report metrics to SurgeGuard using shared files, and implement the protobuf changes shown in Figure 7 in the paper. Additionally, we create a microbenchmark (`CHAIN`) which creates a chain of 5 services, each of which does some arithmetic work. It uses the Thrift protocol and the threading models used by DeathStarBench.

The code for SurgeGuard distills contributions C_1 and C_2, and can be found in `controllers`. We also provide the code which we use to evaluate the Parties and Caladan algorithms in the same folder as well.

Expected Results

Overall, SurgeGuard should provide a lower violation volume for any given spike pattern as compared to the baselines – Parties and CaladanAlgo. It should also have the same or lower core usage as compared to Parties in all cases, and have a lower core usage than Caladan for all workloads other than hotelReservation. The performance benefit of SurgeGuard over the baselines should increase with longer and larger spikes. This shows that our insights and the resulting SurgeGuard design is able to more effectively manage spikes (due to the lower violation volume) and achieve efficient core allocations (due to the lower core usage).

Expected Reproduction Time (in Minutes)

Building the applications and setting up the configuration files for the experiments takes around 1 hour. Executing each experiment takes 2 minutes. We repeat the experiments multiple times for spike configuration, and also gather data for multiple spike configurations, resulting in the experiments taking a few hours to complete in total. The analysis step is currently performed manually, and takes around 1 hour.

Artifact Setup (incl. Inputs)

Hardware: Hardware: Our experiments used a cluster of four 2-socket Intel Xeon 6242 (Cascade Lake) server nodes each with 64 logical cores and 224GB of memory running Ubuntu-18.04. For the controllers that manage frequency, we disable the `intel_pstate` and `acpi_cpufreq` frequency governors and use the `userspace` frequency governor to have precise control over core frequencies. For the rest, we use the default `intel_pstate` governor. The initial frequencies of the cores are set to 1.6GHz.

Software: All the software dependencies required by the application are specified in the respective Dockerfiles, which automatically install the required dependencies. Other than that, the dependencies are `gcc-9.4`, `docker-v21.0+`, `docker-compose-v1.13+`, `python-v3.8+`, `asyncio-v3.4`, `aiohttp-v3.9`, `libssl-dev`, `zlib1g-dev`, `luarocks-v2.4.2` and `luasocket-v3.1.0`. Note that these versions are just the ones we used for our experiments, the applications and controllers work well with other versions of these packages as well. Except for docker, these packages are downloaded using `apt` or `pip`. Docker can be downloaded from https://docs.docker.com/desktop/install/ubuntu/ or from https://download.docker.com/linux/ubuntu/dists/focal/pool/stable/amd64/.

Datasets / Inputs: The datasets used by the applications are taken from DeathstarBench and are summarized below:

- socialNetwork - `socfb-Reed98` (used for our experiments, taken from Facebook Networks https://networkrepository.com/socfb-Reed98.php) and

`ego-twitter` (subset of the Twitter social network graph)
- mediaMicroservices - `tmdb`, (taken from the IMDB database)
- hotelReservation - dataset created by the authors of DeathstarBench

For socialNetwork, we initialize the databases by generating and storing 30 posts (length and contents of each post are randomly generated) for each user.

Installation and Deployment: All requirements for installation and deployment have been specified in "Software". The instructions for building and deploying the applications and running the experiments are provided in `apps/README` and `controllers/README`.

Artifact Execution

Briefly, the order of operations for running the experiments is as follows: (1) The applications are built using `docker build` and deployed using the `docker-compose.yml` files in their respective directories. (2) The initial core allocations and the per-service parameters are specified in a config file (see `controllers/README` for a description of the contents of the config file, and `controllers/sample_config` for a sample config file. (3) The controller code is initialized to reflect the number of available cores on the system (4) The workload generator (A_2) and the controller are run in parallel to send HTTP requests into the deployed application following the desired spike patterns while simultaneously managing allocations using the controller.

For our experiments, we allocate 52 cores/node for the application, and reserve the remaining cores for running the controllers, network processing and other OS tasks. We initialize the work-load to use 2/3rd of the remaining cores (the remaining cores would be allocated to background tasks, and revoked on demand, in real deployments). The initial per-container core allocations can be set arbitrarily – we set them by searching for the allocations that can support the highest request rate using these cores. We set the base request rate in our experiments to be slightly less than the knee of the load latency curve achieved using our initial allocations.

The user needs to specify the per-service parameters (desiredExecMetric and desiredTimeFromStart, see section 4 in the paper) in the config files. To do this, we run the workload at low load for 1-2 mins and periodically collect/calculate the desired parameters from each service. For example, to calculate desiredExecMetric, we periodically read the per-service execMetric values from each container. After collecting the values for 1-2 mins, we set the final desired parameter values to 2x of the average parameter values obtained at low load. We then write these values into the config files. Note that this is not the only way of setting the per-container targets, they can be set by the user through other types of profiling and the multiplication factor between the desired parameter values and the profiled values can be changed to set tighter or looser bounds.

Artifact Analysis (incl. Outputs)

The analysis step is simple, comprising of collecting the violation volume and tail latencies from all the experiment runs, and averaging them appropriately to obtain comparisons between the evaluated controllers. The controllers report the average core usage during the experiments, which we average as well. For each spike pattern, we collect 17 data-points for each controller. While averaging these data-points, we exclude the best and worst data-points to remove extreme outliers, and average the remaining 15 data-points to report the final value.

B. Computational Artifact A_2

Relation To Contributions

This artifact includes our open-loop workload generator, which is a modified version of the `wrk2` workload generator present at https://github.com/giltene/wrk2. We modify it to generate input load spikes for our experiments (C_4) and to calculate and report the violation volume metric(C_3).

Expected Results

Running the workload generator provides a histogram of the request latencies during the experiment and the violation volume.

Expected Reproduction Time (in Minutes)

There is no inherent time limit imposed by the workload generator. For our experiments, we chose to warm up the system for 30s and collect data over the next 60s.

Artifact Setup (incl. Inputs)

Hardware: No specific hardware requirements.

Software: Our changes do not require any additional software packages as compared to the basic `wrk2` workload generator. `wrk2` uses lua to generate HTTP requests – we used luarocks-v2.4.2 and luasocket-v3.1.0.

Datasets / Inputs: Our changes introduce new input parameters for specifying the magnitude and duration of the spikes and the end-to-end QoS limit. They are described in more detail in `wrk2_spike/README.md`.

Installation and Deployment: The workload generator simply needs to be compiled and run as detailed below, it does not have any other installation/deployment steps.

Artifact Execution

The workload generator is compiled using `wrk2_spike/Makefile`. Once the application (A_1) is deployed, the workload generator is used to send HTTP requests to the deployed application and collect per-request latency information throughput the experiment.

We ran all experiments with the options `-threads 16 -connections 256`, as we noticed that setting fewer threads or open connections led to the workload generator becoming a performance bottleneck in some cases. The other important parameters are set as follows:

- -rate: The request rate at steady state, set to be slightly below the knee of the load-latency curve.

- -spikerate: The request rate during the spike, set to 1.25x, 1.5x, and 1.75x of the base request rate (Figure 10).
- -spikelen: The duration of the spike, set to 0.1s-5s (Figures 10, 11).
- -qos: The target QoS limit, used for calculating the violation volume

Artifact Analysis (incl. Outputs)

The output of the workload generator is a histogram of the request latencies and the violation volume during the experiment. Further analysis of the output is not a part of this artifact, it must be done by the user based on their requirements.