Towards Performance Robustness for Microservices

Divyanshu Saxena
UT Austin

Gaurav Vipat
UT Austin

Jiaxin Lin
Cornell University

Jingbo Wang
Purdue University

Isil Dillig
UT Austin

Sanjay Shakkottai
UT Austin

Aditya Akella
UT Austin

Abstract

Microservices are foundational to modern distributed applications, enabling modular design and scalability. However, they face performance variability due to environmental factors like workload burstiness, resource contention, and shared dependencies. Existing microservice controllers, such as autoscalers and admission controllers, struggle to ensure good performance, often causing several Service Level Objective (SLO) violations. We argue that this is because controller decision-making is uninformed, lacking guidance about robustness to environmental factors.

We propose the concept of run-time "performance robustness certificates" (PERCs) to address this limitation. A PERC provides statistical bounds on tail latencies of specific request types under a range of environmental perturbations. We show how to leverage a queueing-theoretic model of microservice performance to quickly derive actionable PERCs. We introduce Galileo, a framework that integrates PERCs with two state-of-the-art learned controllers to guide robust actions toward meeting SLOs. Experimental results with real-world benchmarks validate the effectiveness of PERCs in ensuring robust microservice performance.

1 Introduction

Modern microservice-based applications decompose functionality into tens to hundreds of independently managed services that collectively process diverse request types [1, 19, 21, 27]. While this modularization improves scalability and developer productivity, it also exposes applications to severe performance variability. Tail latencies, which dominate user-perceived performance, are highly sensitive to subtle environmental perturbations such as workload burstiness, background contention, and shifts in request composition. Our measurements on open-source benchmarks reveal that even modest changes in these *latent conditions* can induce sharp latency spikes: for example, a 20% increase in arrival rate for a SocialNetwork request type causes a 1.7× increase in its 99th-percentile (99p) latency despite adequate provisioning.

To mitigate such dynamics, microservice deployments typically rely on controllers, e.g., autoscalers [28, 32, 41, 44] and admission controllers [30, 46], that adjust CPU allocations or throttle requests to maintain service-level objectives (SLOs). However, as our study shows (Section 2), even state-of-the-art learning-based controllers lack robustness in dynamic environments. By relying solely on recent latency and utilization signals, they remain oblivious to latent environment shifts and react only after SLOs have already been violated, resulting in delayed and often suboptimal corrective actions.

We argue that robust microservice management requires controllers to proactively reason about performance under environmental perturbations. To this end, we introduce PERC*s*, or *Performance Robustness Certificates*, a new abstraction that provides certified upper bounds on end-to-end tail latencies across a bounded neighborhood of perturbations. A PERC, denoted (ε, θ), guarantees that, under all perturbations in a parameterized set ΔE around the current environment E, the ε-th percentile latency does not exceed θ. For instance, a PERC $(99p, 120\text{ms})$ certifies that the 99p latency remains below 120ms despite plausible changes in arrival rates, service speeds, or background interference. If this bound satisfies the SLO, the deployment is certified robust; otherwise, controllers can preemptively trigger corrective actions.

Designing PERCs that are both accurate and practical requires addressing three challenges: (1) the high-dimensionality of latent perturbations, (2) the analytical intractability of analyzing tail latency distributions across distributed microservices, and (3) the need for efficient runtime computation of PERCs. To meet these challenges, we develop a *Performance Reasoning Model* combining classical queueing theory, formal semantics for propagating perturbations, and data-driven estimation of model parameters. Our model captures each service as an $M/M/1$-PS queue and uses results from classical queueing theory to derive closed-form approximations for end-to-end latency distributions. We formalize the perturbations to the model, allowing a precise definition for PERCs. Finally, we analytically compute PERCs using only live end-to-end latency measurements within gradient approximation methods, allowing PERCs to be updated online with negligible overhead.

While PERCs provide the basis for reasoning, their effectiveness hinges on integration with controllers. We therefore design *Galileo*, a framework that unifies PERCs with reinforcement-learning-based autoscalers and admission controllers. Galileo employs *shields* [5, 9] that evaluate controller actions against PERCs: unsafe actions that risk violating SLOs are replaced with provably safe alternatives. During training,

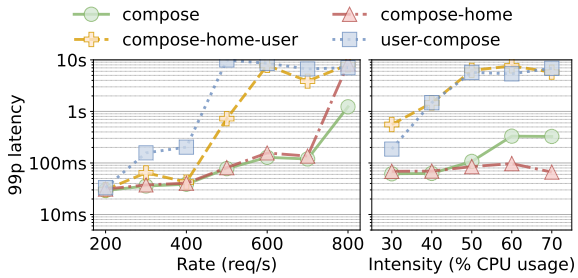


Figure 1: Performance variability for the `compose` request type of SocialNetwork benchmark application. Curves represent workloads comprising various request types, as indicated in the legend.

shields supply both penalties for unsafe actions and robustness rewards that guide the agent toward policies resilient to environment shifts. At inference, the same shields act as model-predictive safeguards, ensuring that deployed actions maintain robustness at runtime.

In summary, this paper makes the following contributions:

- We empirically demonstrate the sensitivity of microservice tail latencies to environmental perturbations and the fragility of current controllers (Section 2).
- We introduce PERCs, a new abstraction that provides certified worst-case tail latency bounds under bounded perturbations (Section 3).
- We develop a Performance Reasoning Model that enables efficient runtime computation of PERCs using queueing analysis, formal semantics, and gradient estimation techniques (Section 4).
- We design and implement Galileo, which integrates PERCs with learned autoscalers and admission controllers through shielding, reducing SLO violations by up to 99.4% with only up to 22% more average CPUs (Section 5–Section 7).

2 Background and Motivation

A microservice application comprises multiple long-running services, each executing in its own container and interacting with others. A user request triggers one service, which may in turn call other services to complete processing. These calls form a directed graph representing the sequence of microservices invoked for that request. Applications typically support multiple *request types*, where all requests of a given type consistently traverse the same subset of services.

2.1 Performance Variations in Microservices

Modern microservice deployments host critical business logic [1, 21, 27], requiring requests to meet strict latency constraints. Yet application latencies are highly sensitive to *latent factors* such as workload patterns and composition and resource contention from co-running jobs. Even small perturbations in these factors can lead to large performance shifts.

To illustrate, we deploy benchmarks from DeathStarBench [19] on a Kubernetes [2] cluster with fixed resources. Using wrk2 [3], we generate workloads and measure end-to-end latencies for a specific request type at fixed rates over

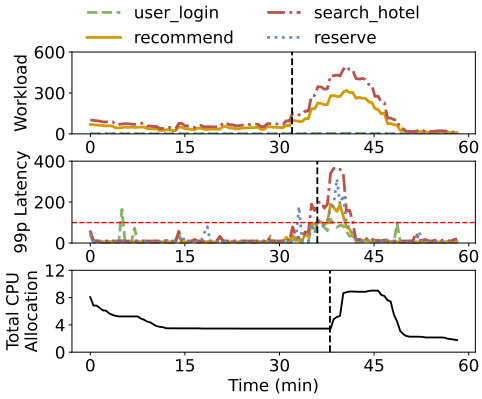


Figure 2: Execution of Autothrottle [41] over a 1-hour trace from Alibaba traces [27] on the Hotel Reservation benchmark (with four request types). Horizontal dashed line shows the SLO (100ms).

2-minute intervals, repeating the experiment under varying request rates, concurrent workloads, and background job intensities (details in Section 7). Figure 1 summarizes the results.

We observe two distinct regimes: (i) *Stable regime*: when processing capacity exceeds arrival rate, latencies remain predictable. (ii) *Overloaded regime*: when arrivals outpace capacity, queues grow rapidly, causing severe latency inflation. Even within the stable regime, small perturbations cause sharp latency increases; e.g., raising the `compose` request rate from 500 to 600 req/s (20% increase) in the Social Network benchmark increases 99p latency by 1.68× (77ms to 130ms).

Past an inflection point, latency growth accelerates. In the overloaded regime, increasing the `compose` request rate by 14% (700 to 800 req/s) drives a 10× jump in 99p latency due to queue buildup. The inflection point shifts with concurrent workloads: for `compose` alone, it is ∼700 req/s, but drops to 400 req/s when `home` and `user` requests co-run.

Background jobs further exacerbate latencies by creating contention, reducing effective processing rates. In our experiments, running CPU-intensive tasks on each cluster node and repeating measurements at 600 req/s shows that raising average background CPU utilization from 40% to 60% increases 99p latency for `compose` by 5.3× (62ms to 331ms). Despite configured CPU limits, co-running jobs contend for shared resources such as preemptions, cache, and memory bandwidth, degrading processing capacity and end-to-end performance.

As shown in Appendix A, extensive experiments with other DeathStarBench services show similar behavior.

2.2 Efficacy of Microservice Controllers

To manage performance variability, microservice controllers such as autoscalers and admission controllers dynamically adjust resource allocations and request rates. While platforms like Kubernetes [2] provide simple heuristic controls, recent work has explored learning-based techniques.

We evaluate two state-of-the-art learning-based controllers, Autothrottle [41] (autoscaling) and TopFull [30] (admission control), using two bursty, one-hour traces derived from the

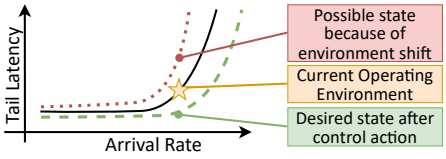


Figure 3: Each curve corresponds to a particular processing rate, with small environmental differences reflected by their proximity. Lower curves indicate higher rates. Suppose the system runs in an environment (marked ★) with high latencies. An autoscaler may allocate more CPU to move it to a lower-latency state (green dashed curve). Yet perturbations, e.g., a background job, can reduce the effective rate, pushing the system to the red curve with worse latencies.

open-source Alibaba cluster trace [27]. By examining their behavior under workload fluctuations and resource contention, we highlight their limitations in dynamic environments, underscoring the need for more robust performance management.

Figure 2 shows that Autothrottle quickly violates SLOs once the request arrival process changes (e.g., mean rate shifts), as at T=32 min (dashed line, top plot). Violations persist until request rates subside. This occurs because the controller lacks visibility into environment perturbations, only detecting the shift at T=36 min (middle plot) once enough latencies exceed the SLO. By the time corrective action is taken at T=38 min (bottom plot), queued requests already inflate latencies. Heuristic controllers suffer similarly [20, 34].

Further experiments, e.g., with the TopFull [30] admission controller and under co-running background jobs, reveal the same inefficacies (See full results in Appendix B).

2.3 Environment Impacts on Performance

Microservice controllers operate under the implicit assumption that the true environment remains close to measured conditions. In practice, small latent perturbations, e.g., changes in background load or request composition, can shift the environment significantly. Actions that would reduce latency under nominal conditions may fail or appear to worsen tail latency when the environment changes; e.g., an autoscaler may add CPUs to reduce latency, but background jobs or unanticipated contention may lower the microservices' effective processing rate, pushing the system to even higher latencies (Figure 3).

This is a fundamental limitation of reactive controllers: without explicit reasoning about potential perturbations, decisions are made based on short-term signals and may not maintain SLOs under varying conditions.

3 PERCs

For robust performance, controllers must anticipate how end-to-end latencies change under environmental perturbations. To this end, we introduce **Per**formance **R**obustness **C**ertificates (PERC*s*), an abstraction that provides provable bounds on end-to-end latencies across a set of perturbation scenarios. Below, we formally define PERCs and outline how our framework derives and applies them to robustify controllers.

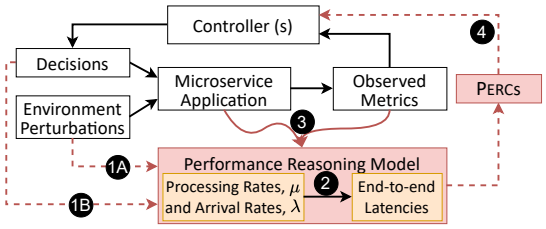


Figure 4: Overview of our approach: shaded boxes and dashed arrows represent new constructs introduced in our approach.

3.1 PERC Definition

Microservice application performance can vary widely across environments, as shown earlier. Here, *environment* encompasses both observable and latent factors. A PERC captures how the performance of a certain type of requests changes under perturbations to these factors by upper-bounding the latency within a neighborhood of the current environment:

Definition 1 (PERCs) *The* PERC, $P_{\varepsilon,E,T}$ *for a given environment* E*, target percentile* ε *and request type* T *is defined as the maximum possible* εp *latency for requests of type* T *across a range of perturbed environments, denoted by* Δ_E*, i.e.,*

$$P_{\varepsilon,E,T} = \max\{l_{\varepsilon,T}(E') \mid E' \in \Delta_E\} \quad (1)$$

where $l_{\varepsilon,T}(E)$ denotes the ε %-ile (or εp) of the end-to-end latency of a specific request type T under an environment E.

3.2 Overview of Our Framework

We now describe how our framework computes PERCs and uses them in microservice control loops (see Figure 7).

Capturing environment perturbations. A key challenge in computing PERCs, as defined in Equation (1), lies in the underspecification of the perturbation set Δ_E. The environment involves numerous uncontrollable factors like background interference, making it difficult to precisely define the magnitude of variation that constitutes E's local neighborhood.

To address this challenge, we design a *Performance Reasoning Model (PRM)*, rooted in classical queueing theory (see Figure 4). Each service is represented as an $M/M/1$-PS queue, characterized by its arrival and processing rates (Section 4.1). The key benefit of this model is that any environmental perturbation is captured as changes in these rates (arrow 1A). Then, instead of quantifying shifts across numerous environmental factors to capture Δ_E, operators only need to specify *worst case* perturbations in *arrival and processing rates across all queues* (termed as "parameters" of our PRM).

Mapping perturbations to end-to-end performance impact. Computing PERCs requires reasoning about how perturbations induce *tail-latency changes*, which is challenging.

To this end, building on the above, we define *a formal semantics* for our PRM, mapping changes in arrival and processing rates to end-to-end latency distributions (Figure 4; 2). The semantics enable us to precisely characterize the environment perturbation set (Section 4.2) and derive expres-

sions linking environment perturbations to the parameters of our PRM. These expressions are then used to compute PERCs.

Fitting the model to live applications. To compute accurate PERCs, we ideally require high-fidelity real measurements of the processing and arrival rates for all services, which we can then fit into our model. However, obtaining these from live applications has high overhead and is impractical.

We address this via a *data-driven approach* that fits parameters of our PRM to live *end-to-end measurements* from the running application (see 3 in Figure 4) and allows the model to closely resemble actual performance. We use *efficient gradient-estimation techniques* to compute these parameters quickly (Section 4.3). Thus, PERCs can be recomputed periodically at low overhead keeping them up-to-date under environment changes.

Integration with controllers. To leverage performance reasoning and PERCs systematically, learned microservice controllers need key design changes (arrow 4 in Figure 4).

To this end, we use our PRM to estimate the impact of a controller action before applying it (arrow 1B in Figure 4); this enables *model-predictive shielding* [8,9], a safe reinforcement learning technique (Section 5). During training, shielding and PERCs help craft reward signals that nudge controllers away from unsafe actions and towards robust decisions – i.e., the learned controller is incentivized to maintain PERCs within desired SLOs alongside primary objectives. During inference, shielding proactively checks whether a control action may cause SLO violations under environment perturbations, and if so, take a safe action identified using our PRM.

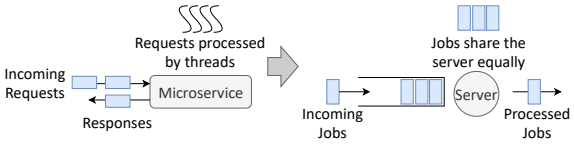
4 Performance Reasoning Model (PRM)

Rigorous performance bounds are hard to obtain in distributed systems due to dynamic, variable environments [22, 43]. In particular, predicting how perturbations affect SLO satisfaction remains a critical challenge central to PERCs, especially because tail latencies are hard to analyze.

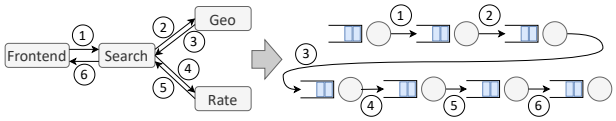
We address this issue with a novel approach that combines queueing theory, formal methods, and gradient estimation for scalable yet precise tail performance reasoning. Our approach models per-service and end-to-end processing in microservice applications, then deriving closed-form approximations for the end-to-end latency distributions; formalizes the impact of perturbations by defining a perturbation semantics for the model; and, armed with the model and semantics, applies lightweight data-driven fitting to compute PERCs efficiently.

4.1 Request Processing Model for Microservices

Our request-processing model captures the core execution behavior of microservices. Incoming requests enter a service queue (Figure 5a) and are executed by worker threads. Each service may use a fixed-size thread pool or spawn a new thread per request. Threads share the resources allocated to a microservice (e.g., CPU, memory), and upon request completion, the thread becomes available for the next queued request.



(a) A single microservice modeled as an $M/M/1$-PS queue.



(b) Services invoked by a user request modeled as a network of $M/M/1$-PS queues, where requests may re-enter other queues. Here, `geo` and `rate` are invoked sequentially.

Figure 5: Modeling the latency of a user request in a microservice applications using a network of $M/M/1$-PS queues.

Modeling individual services. This execution pattern is well captured by the $M/M/1$-PS queueing abstraction, where requests arrive as a Poisson process with rate λ, share the processing capacity of the server, are served at a rate μ (determined by the resources allocated to the service), and depart upon completion. This model is exact for thread-per-request service implementations. For fixed thread pools, it provides a close approximation when the pool size is sufficiently large, a common configuration used to reduce queuing delays.

Extending to microservice chains. Requests in a microservice application traverse a network of services, where one service may invoke another during processing (Figure 5b). While waiting for a downstream response, the thread is idle and does not consume resources, effectively causing the request to *exit* the queue. Once the response arrives, the thread resumes execution, which can be viewed as the request *re-entering* the queue. This behavior can be modeled by assigning a distinct $M/M/1$-PS queue to each hop along the request's path. This network of $M/M/1$-PS queues not only closely matches microservice processing but also allows the derivation of approximate closed-form expressions for end-to-end latencies as shown below, unlike other generic models, e.g., $G/G/1$ where such analysis is mathematically hard.

Expressions for end-to-end latency distributions and the latency function $l_{\epsilon,T}(E)$. We build on a classical result from queueing theory on the sojourn times (time between a job's arrival and its departure after processing) of a network of $M/M/1$-PS queues. We summarize the result below; please refer to Theorem 4.5.3 in [40] for details. The theorem implies that under the assumptions of overtake-free paths[1], for an ergodic[2] network of $M/M/1$-PS queues, the sojourn times at each queue are *independent* and *exponentially distributed*.

Mapping this to microservice applications, if a request traverses queues $(1,2,\ldots,K)$, the sojourn time at queue i is

[1]Overtake-free paths roughly imply that jobs from within a class are processed in the order they arrived; see [39] for the precise definition

[2]Ergodicity of a queueing system implies that every job completes in finite time (or the queue does not grow indefinitely over long period of time).

$\theta_i \sim Exp(\gamma_i)$), where $\gamma_i = (\mu_i - \lambda_i)$ is the parameter for the exponential distribution for the sojourn time, and μ_i and λ_i are the processing and arrival rates at the i-th queue, respectively. The corresponding end-to-end sojourn time for a request is given as $\theta = \sum_i \theta_i$ (summation over the queues along the request path). We can then approximate the sum of exponential variables, θ, using the Welch–Satterthwaite equation [3]:

$$\theta \sim \Gamma(\alpha, \beta); \quad \alpha = \frac{(\Sigma\gamma_i)^2}{\Sigma\gamma_i^2}, \; \beta = \frac{\Sigma\gamma_i^2}{\Sigma\gamma_i} \tag{2}$$

where $\Gamma(\alpha, \beta)$ is the Gamma distribution with parameters α and β. For simplicity, we will use the term *service parameters* to denote the processing and arrival rates (μ_i, λ_i) of individual queues, and the term *distribution parameters* to denote the end-to-end latency distribution (α, β).

Now the εp tail latency function $l_{\varepsilon,T}(E)$, used in the PERC Definition 1, can be expressed as:

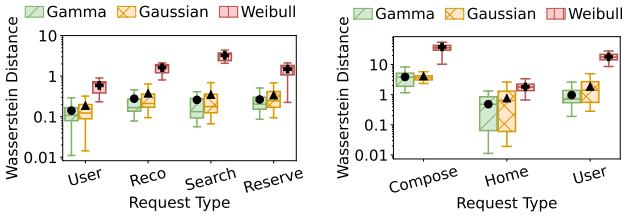
$$l_{\varepsilon,T}(E) = ppf(\Gamma(\alpha, \beta), \varepsilon) \tag{3}$$

where $ppf(D, \varepsilon)$, the probability point function for D, returns the ε percentile value for the distribution D. If we can estimate α and β under worst-case perturbations, we can compute the PERC using Equation (3).

Overall model definition. Note that different request types may invoke a different network of services. Therefore, they must be captured under different models. Formally, the model $\mathcal{M}_r$ for request type r, is given by $\mathcal{S}_r \times \mathcal{D}_r$, where $\mathcal{S}_r$ denotes the service parameters of all queues invoked by request type r, and $\mathcal{D}_r$ denotes the distribution parameters.

Dealing with complex communication patterns. Microservices may exhibit *asynchronous requests* with parallel service invocations that do not perfectly conform to our model described above. For instance, in Figure 5b, the services `geo` and `rate` may be invoked *in parallel* by `search`. In such cases, the end-to-end latency for a request is determined by the path (of queues) with the longer sojourn time. In contrast, our model assumes that all requests of a particular type follow the same path. This simplification is not problematic in practice: asynchronously invoked services typically have asymmetric processing rates, meaning one service dominates as the bottleneck. Consequently, even when services are invoked in parallel, the end-to-end latency is well-approximated by the sequence of queues containing the slower branch.

Empirical validation. We thoroughly validate this model empirically. We run the Hotel Reservation (HR) and Social Network (SN) benchmarks [19] (detailed setup in Section 7) over 150 different, but controlled environments using varying request rates (200-1000 reqs/sec), arrival processes (exponential and zipfian), and various request type mixes. For each environment, we collect end-to-end latencies for each request type, fit a Gamma distribution to 10% of the collected samples using Method of moments, and measure the Wasserstein distance [14] between the fitted distribution and 100% of the collected samples for that environment. We compare this model against Gaussians (used in [43]) and Weibull distributions (as a representative heavy-tailed distribution). Figure 6 shows that the average Wasserstein distances over all environments are 5-44% better than Gaussian and 73-95% better than Weibull for either applications – thus, validating our model.



(a) Hotel Reservation (b) Social Network

Figure 6: Wasserstein distances of proposed Gamma model vs. baselines. Y-axis is in log scale; markers show means.

We also evaluate the ratio of tail latencies of the fitted distribution to the full set of collected samples. Across different request types of both applications, we find that the tail latency ratios in the Gamma modeling are 6-49% better than the baselines, while being 0.83-1.01× the ground truth 99p latencies on average (see Appendix E.1 for detailed explanations).

This study shows that the Gamma distribution is indeed a good fit for modeling end-to-end latencies. Moreover, this model closely approximates the latency distributions for request types that do not perfectly align with our model assumptions. For instance, the `search` request type in HR and `compose` request type of SN includes artifacts such as asynchronous requests, but still our Gamma-distribution-based estimates only incur 8% and 20% error on average.

4.2 Semantics for Reasoning about Perturbations

The model, $\mathcal{M}_r$, described above only characterizes the current state of the microservice application. Computing PERCs, however, requires understanding how this model evolves under environmental perturbations. Here, we note a salient feature of using the queue-based abstraction: *any change in the environment will manifest as a change to the processing and/or arrival rates across services*. Thus, we can formalize these perturbations into *actions* over the model as follows:

$$\begin{aligned} \mathsf{P} &::= \mathsf{perturb}_\mu \; i \; \eta \mid \mathsf{perturb}_\lambda \; i \; \eta \\ \mathsf{actions} &::= \mathsf{P} \mid \mathsf{P} \circ \mathsf{P} \end{aligned} \tag{4}$$

where $\mathsf{perturb}_\mu$ (or $\mathsf{perturb}_\lambda$) is an *action over our model* that applies a perturbation factor η to the processing (or arrival) rate μ_i (λ_i) of queue i. Note that the definition above allows composition of multiple co-occurring perturbations, captured via the composition operator $\circ$. Thus, any environmental change can be represented as one or more such perturbation

[3] Using the Welch-Satterthwaite approximation, one can show that if $X_1, X_2, \ldots, X_n$ are independent Gamma random variables such that $X_j \sim \Gamma(a_j, b_j)$, then $Y = \sum X_i$ is distributed as the Gamma distribution, $\Gamma(p, q)$ where $p = \frac{(\sum a_j b_j)^2}{\sum a_j b_j^2}$ and $q = \frac{\sum a_j b_j^2}{\sum a_j b_j}$. Further, an exponential distribution is a special case of a Gamma distribution with rate parameter, $a_j = 1$.

actions applied across services.

We now define the *semantics* of these actions, that is, how these actions impact the model parameters. For a single action, we define the semantics as:

$$\forall i, \eta. [\![\mathsf{perturb}_\mu\ i\ \eta]\!](\mathcal{M}_r) = \mathcal{M}_r[\mu_i \mapsto \mu_i + \eta\mu_i, \quad \alpha \mapsto \alpha + \mathcal{P}_\alpha(\mu_i, \eta), \quad \beta \mapsto \beta + \mathcal{P}_\beta(\mu_i, \eta)] \tag{5}$$

where $[\![\mathsf{perturb}_\mu\ i\ \eta]\!](\mathcal{M}_r)$ denotes the application of the action $\mathsf{perturb}_\mu$ on the current model, $\mathcal{M}_r$, and $\mapsto$ denotes the update of a model parameter. The functions $\mathcal{P}_\alpha(\mu_i, \eta)$ and $\mathcal{P}_\beta(\mu_i, \eta)$ are *perturbation functions* that denote how much the distribution parameters α, β change when the processing rate μ_i of the queue i changes by a factor η. Similar semantics can be written for $\mathsf{perturb}_\lambda$. These semantics extend naturally to compositions: $[\![a \circ b]\!](\mathcal{M}_r) = [\![a]\!]([\![b]\!](\mathcal{M}_r))$ (see Appendix C.1).

Defining the range of perturbations Δ_E**.** We can now define Δ_E as the set of environments where the processing or arrival rates of any service deviate *up to* some maximum factor δ from their respective values in the current environment E. Formalizing in terms of the model actions, we have:

$$\Delta_E = \{\bigcirc_i [\![(\mathsf{perturb}_\mu\ i\ \eta_{\mu_i}) \circ (\mathsf{perturb}_\lambda\ i\ \eta_{\lambda_i})]\!](\mathcal{M}_r) \quad \forall i \in N(r), -\delta < \eta_{\mu_i}, \eta_{\lambda_i} < \delta\} \tag{6}$$

where $\bigcirc a_i$ denotes the composition of actions a_i for all queues i, and $N(r)$ denotes the set of services invoked by a request of type r. A larger δ corresponds to a broader perturbation set, while smaller values yield narrower sets.

The parameter δ serves as a tunable hyper-parameter of our framework, allowing operators to set the level of robustness they want the PERCs to reflect.

Worst-case perturbations to compute **PERCs.** Note that the worst tail latencies in Δ_E occur when the processing rates at all queues decrease by the maximum value possible (i.e., -δ), *and* arrival rates at all queues increase by the maximum value possible (i.e., δ). This is because microservice latencies are monotonic with arrival rates, and generally inversely proportional to processing rates (or, allocations) [28,32]. It is for this perturbation that we would like to compute the tail latency, which essentially acts as the PERC for the current environment. Thus, computing PERC boils down to applying the actions $[\![\mathsf{perturb}_\mu\ i\ -\delta]\!]$ and $[\![\mathsf{perturb}_\lambda\ i\ \delta]\!]$ for all queues i, using the semantics in Equation (14). Formally, this perturbation can be defined as the following composition of actions:

$$\forall i \in N(r), \bigcirc_i [\![(\mathsf{perturb}_\mu\ i\ -\delta) \circ (\mathsf{perturb}_\lambda\ i\ \delta)]\!](\mathcal{M}_r) \tag{7}$$

All we now require for PERC computation is the knowledge of the current parameters (α, β) and the perturbation functions $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ (to apply Equation (14)).

4.3 Computing PERCs Efficiently

In the ideal case, if we could accurately measure the service parameters (μ_i, λ_i) at each service, the current distribution parameters and the perturbation functions can be obtained from Equation (2). However, accurate measurements of service parameters are not feasible. First, extracting these service parameters requires significant instrumentation that adds overheads, for instance, by adding sidecar containers to each service pod that accurately measure how many requests arrived/exited at each service. This can add significant overhead, if done for all requests. Second, microservices being dynamic, complex systems, environment changes can easily lead to noisy μ_i and λ_i measurements; e.g., due to multiple threads releasing several requests at once, these parameters may see sudden spikes.

Data-driven inference of model parameters. We address the aforementioned challenges via a data-driven mechanism that relies on *only end-to-end latency measurements* used by controllers today. Our mechanism proceeds in two steps: first, we estimate the values of the distribution parameters using observed end-to-end latency data, and then, we estimate $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ via *gradient estimation methods*.

Step 1: Fitting the model to live data. At short, fixed intervals, we fit a Gamma distribution to recent end-to-end latencies for each request type, updating the distribution parameters (α, β). This assumes that the environment does not change significantly within this interval.

Step 2: Estimating $\mathcal{P}_\alpha$ *and* $\mathcal{P}_\beta$. We describe our technique for estimating $\mathcal{P}_\alpha$. Consider α as a function over service parameters, $\alpha = f_\alpha(\mu_1, \mu_2, \ldots, \lambda_1, \lambda_2, \ldots)$ (as in Equation (2)). Then for small η in Equation (5), we get the following expression using Taylor expansion:

$$\mathcal{P}_\alpha(\mu_i, \eta) = \eta\mu_i \frac{\partial f_\alpha}{\partial \mu_i} \approx \eta\, \mathcal{K}_{\alpha,\mu_i} \tag{8}$$

where $\mathcal{K}_{\alpha,\mu_i}$ is assumed approximately constant as long as the perturbation for μ_i remains within a small η neighborhood. Complete mathematical analysis for the above approximation is provided in Appendix C.2.

The important piece here is that the value of f_α at μ_i is already known (from step 1 above), and the value at $\mu_i + \eta\mu_i$ can be obtained by fitting another Gamma distribution over end-to-end latencies *collected with the processing rate* μ_i *tweaked by a small factor* η. In practice, we can achieve this effect by simply tweaking the current control action - resource allocation at service i - by a factor η as processing rates are directly proportional to allocated resources for each queue.

Similar analysis follows for $\mathcal{P}_\beta$: to tweak the arrival rate λ_i, we can tweak the rate limits for various request types, effectively tweaking the arrival rates at individual services.

Final **PERC** ***computation.*** With the functions $\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ estimated, we can apply the worst-case δ-perturbation to the semantics in Equation (14):

$$\alpha^* \mapsto \alpha + \sum_i \mathcal{P}_\alpha(\mu_i, -\delta) + \sum_i \mathcal{P}_\alpha(\lambda_i, \delta) \tag{9}$$

$$\beta^* \mapsto \beta + \sum_i \mathcal{P}_\beta(\mu_i, -\delta) + \sum_i \mathcal{P}_\beta(\lambda_i, \delta) \tag{10}$$

The resulting PERC can now be computed using Equation (3):

$$P_{\varepsilon,E,T} = ppf(\Gamma(\alpha^*, \beta^*), \varepsilon) \tag{11}$$

Algorithm 1 PERC Computation

Input: Request type r; tail percentile ε; number of perturbation samples N; current control (allocation/rate limits) $\mathbf{c}$; operator-specified perturbation budget δ.
Output: Worst-case tail latency estimate $P_{\varepsilon,E,T}$ (the PERC).

1: **procedure** COMPUTECERTIFICATE($\varepsilon, \mathbf{c}, r, \delta, N$)
2: $D \leftarrow$ COLLECTLATENCIES(r) ▷ Gather latencies for type r
3: $(\alpha, \beta) \leftarrow$ GAMMAFIT(D) ▷ Fit Gamma distribution
4: $\mathcal{P}_\alpha, \mathcal{P}_\beta \leftarrow$ COMPUTEGRADIENTS($r, \mathbf{c}, N$)
▷ Use gradient estimation to compute perturbation functions.
5: $\alpha^\star \leftarrow \alpha + \sum_i \mathcal{P}_\alpha(\mu_i, -\delta) + \sum_i \mathcal{P}_\alpha(\lambda_i, \delta)$
6: $\beta^\star \leftarrow \beta + \sum_i \mathcal{P}_\beta(\mu_i, -\delta) + \sum_i \mathcal{P}_\beta(\lambda_i, \delta)$
▷ Worst-case distribution parameters using Equations (9) and (10)
7: $P_{\varepsilon,E,T} \leftarrow ppf(\Gamma(\alpha^\star, \beta^\star), \varepsilon)$ ▷ Using Equation (3)
8: **return** $P_{\varepsilon,E,T}$

9: **procedure** COMPUTEGRADIENTS($\mathbf{c}, r, N$)
10: $\mathbf{A} \leftarrow$ PERTURBATIONMATRIX(N) ▷ See Algorithm 1 in [10]
11: **for** $j = 1 \rightarrow N$ **do** ▷ Make N perturbations
12: $\mathbf{c}^{(j)} \leftarrow$ UPDATECONTROL($\mathbf{c}, \mathbf{a}_j$) ▷ Perturb control.
13: $D^{(j)} \leftarrow$ COLLECTLATENCIES(r)
14: $(\alpha^{(j)}, \beta^{(j)}) \leftarrow$ GAMMAFIT($D^{(j)}$) ▷ Fit to updated latencies
15: $(\widehat{\nabla}\alpha, \widehat{\nabla}\beta) \leftarrow$ RECOVERGRADIENTS$\left(\{\alpha, \beta, (\alpha^{(j)}, \beta^{(j)})\}_{j=1}^{N}\right)$
▷ Estimate gradients via Algorithm 1 in [10]
16: **return** $(\widehat{\nabla}\alpha, \widehat{\nabla}\beta)$

Simultaneous perturbations for quick estimates. Computing the value of constants $\mathcal{K}$ in the above procedure relies on tweaking the control action and collecting updated latencies. A naïve implementation may perturb each service parameter one-by-one, making the overall process of estimating the perturbation functions (and hence, computation of PERCs) time-consuming. Instead, we take inspiration from a recent technique in gradient estimation [10] that allows simultaneous perturbation of multiple variables. The method takes a number of samples, each time perturbing a subset of the variables of interest at once, and then uses the resulting value of the function in all these samples to recover the partial gradients for each variable. This allows the computation of the quantity $\mathcal{K}$ for several service parameters using a small number of perturbations. We present the pseudocode in the function COMPUTEGRADIENTS in Algorithm 1.

4.4 Overall PERC Computation Algorithm

Our overall algorithm is presented in Algorithm 1. At a high level COMPUTECERTIFICATE() has these steps: **(1)** Given current environment E, we collect the end-to-end latency samples obtained by the execution of the microservice application (line 2). **(2)** We then fit a Gamma distribution over the collected samples to obtain distribution parameters (α, β) (line 3). **(3)** We then compute the perturbation functions as described in Equation (8) (line 4). We rely on Algorithm 1 presented in [10] to compute these using a small number of samples (lines 9–16). **(4)** We compute the updated distribution parameters using the worst-case perturbation factor δ (lines 5–6). **(5)** Finally, we compute $P_{\varepsilon,E,T}$ as per Equation (11) (line 7). In practice, we invoke the function COMPUTECERTIFICATE() periodically in order to get updated estimates if the underlying environment has changed.

5 Galileo: Robust Controllers Using PERCs

Existing controllers are non-robust (Section 2) because state observed at time t may omit latent factors that evolve further by $t+1$ when the action is applied. We introduce Galileo, a framework that treats robustness to such latent dynamics as a first-class training objective. Galileo extends *shielding* [5, 8, 9, 25], a safe RL technique that monitors agent actions against a system model and, upon detecting violations, substitutes safe backup actions. This ensures that outcomes satisfy a desired property: i.e., end-to-end performance meets SLOs despite environment perturbations.

The key construct in shielding is a *shield*, represented as a tuple $(\mathcal{M}, \mathcal{P}, \text{SAFE})$, where $\mathcal{M}$ is the model of the system, $\mathcal{P}$ is the property of interest and SAFE is a function that takes a control action a and computes a provably safe action $a^* = \text{SAFE}(a)$. The shield is *triggered* if a violates $\mathcal{P}$ under $\mathcal{M}$, in which case the safe action a^* is taken. During training, if the shield is triggered, a large negative penalty is provided to the agent – nudging the agent towards property satisfaction. The same shield can also be used at inference time, where unsafe actions are replaced by their SAFE counterparts – ensuring property satisfaction at run time.

Next, we describe how to evaluate the robustness property over our PRM (Section 5.1), how the shield integrates with controller training (Section 5.2), and how safe actions are computed (Section 5.3).

5.1 Shields in Galileo

Given a new action a_t (Figure 7), Galileo evaluates the desired $\mathcal{P}$ (microservice performance satisfies SLOs even under environment perturbations) under the PRM, $\mathcal{M}_r$. Evaluating this requires us to compute the expected PERC of the system *after* taking the action a_t, requiring semantics for executing a control action over our model.

Semantics for control actions. We extend our PRM (Section 4) to simulate control actions. Just as environment changes map to variations in queue arrival or processing rates, controller actions (e.g., adjusting resources or rate limits) can be expressed as service parameter updates in the model. Specifically, increasing CPU resources for service i changes the processing rate μ_i of its queue, while applying a rate limit to request type r modifies the arrival rates λ_j for all services it invokes ($j \in N(r)$; $N(r)$ defined in Equation (6)).

Formally, we can define two types of controller actions:

$$\mathsf{controls} ::= \mathsf{adjustCPU}\ i\ \eta \mid \mathsf{rateLimit}\ r\ \eta \tag{12}$$

where adjustCPU corresponds to the autoscaler action of updating CPU resources for queue i by a factor η, and rateLimit changes the rate limit for request type r by a factor η. Now, we can define their semantics in terms of the $\mathsf{perturb}_\mu$ and $\mathsf{perturb}_\lambda$ actions from Equation (4) as follows:

$$\begin{aligned} [\![\mathsf{adjustCPU}\ i\ \eta]\!](\mathcal{M}_r) &= [\![\mathsf{perturb}_\mu\ i\ \eta]\!](\mathcal{M}_r) \\ [\![\mathsf{rateLimit}\ r\ \eta]\!](\mathcal{M}_r) &= \bigcirc_{i \in N(r)} [\![\mathsf{perturb}_\lambda\ i\ \eta]\!](\mathcal{M}_r) \end{aligned} \tag{13}$$

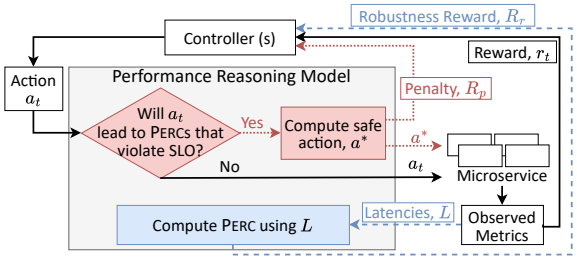


Figure 7: Integration of PERCs and Performance Reasoning for Galileo controllers.

where $[\![a]\!](\mathcal{M}_r)$ and $N(r)$ denote the usual (Equations (5) and (6)). Intuitively, increasing the CPU allocation of service i by η corresponds to scaling the processing rate of its queue by η. Similarly, scaling the rate limit of a request type by η translates to scaling the arrival rates at all services.

These semantics can then be used to compute the impact on model parameters (α and β) using Equation (5). In practice, this yields only an approximation of service behavior, since the computation of perturb actions assumes a small η (see Equation (8)); thus *underestimating* the true impact for large η. However, this approximation is still reasonable in practice: controllers typically apply actions with potential for performance degradation (e.g., reducing CPU or increasing rate limits) conservatively in small steps [41]. Thus, the approximation holds where it matters most – actions critical to performance are still reasonably captured.

Shield construction. The above semantics allow us to translate a new control action a_t into perturbation actions for our PRM. Further, we can also compound these actions with the actions corresponding to the worst-case δ-perturbations to the environment (defined in Equation (7)). Thus, we can compute the expected PERC under environment perturbations *after* the action a_t is taken. The shield in Galileo then checks: *Is this predicted* PERC *within the SLO?*

Formally, the evaluation of the property $\mathcal{P}$ for a new action a_t is done as follows: we apply a_t along with the worst-case actions (from Equation (7)) to our model:

$$\left(\bigcirc_i [\![(\mathsf{perturb}_\mu\ i\ -\delta) \circ (\mathsf{perturb}_\lambda\ i\ \delta)]\!]\right) \circ \left(\bigcirc_j [\![a_t[j]]\!]\right)(\mathcal{M}_r)$$

$\forall i \in N(r), j \in S(a_t)$, where $S(a_t)$ denotes the services impacted by the action, and $a_t[j]$ denotes the control action for service j. Say, applying the above actions results in distribution parameters (α', β'); we can now compute the *predicted* εp latency as $ppf(\Gamma(\alpha', \beta'), \varepsilon)$ (Equation (3)). This latency estimate acts as the PERC for the new control action a_t; if this is higher than the SLO, the shield is triggered.

5.2 Controller Training with Shields

Consider Figure 7 where the controller takes the statistics collected at time t to produce action a_t and constructs some reward function R_o using the statistics. Given the new action, a_t, we evaluate the estimated PERC under the model $\mathcal{M}_r$ as described above. If this PERC is higher than the SLO, the shield is triggered. To allow the agent to learn that the produced action was non-robust, a large negative penalty (denoted R_p) is provided (see Figure 7).

However, the R_p reward signal is inherently sparse: the agent may need to encounter many diverse environments before the shield is triggered. Moreover, the shield is valid only under the model assumptions; in environments that deviate from these assumptions, non-robust actions *may not* activate the shield. Detecting non-robustness *before* executing the action is difficult in such cases. These issues slow training and limit the agent's ability to proactively avoid unsafe actions.

Therefore, to improve learning, Galileo also incorporates a 'post-facto' reward, one computer *after* the action is taken. Specifically, using the latencies collected at time t, denoted L_t, Galileo computes PERCs using the algorithm described in Section 4.4. Then, Galileo incorporates these PERCs as a part of the reward signal *at each time step* by computing an additional reward function for robustness, $R_r = \mathsf{sigmoid}(\text{PERC} - \mathsf{SLO})$.

The final reward combines R_o, R_r, and R_p:

$$r = R_o + wR_r + R_p,$$

where w tunes the weight of the robustness reward. Intuitively, the R_p indicates 'will a control action lead to weak certificates that violate SLOs' and R_r indicates 'how close did the state resulting from the control action come to an SLO violation'.

5.3 Computing Safe Actions

The shield provides not only a feedback signal to the controller, but also computes a safe action SAFE(a). This is important for both training and inference. For controllers trained online (e.g., Autothrottle [41]), this ensures that 'exploration' steps do not impact microservice performance. For offline-trained controllers (e.g., TopFull [30]), this ensures that poor control actions do not impact the rewards observed by better actions in future time steps – leading to overall improved learning. At inference time, this can safeguard the microservice application from poor control actions resulting from incomplete training, unseen environments, or distribution shifts.

Given a control action a_t that violates the shield, Galileo computes a safe action a^* (see Figure 7) that is taken instead of a_t. In computing a^*, our insight is that end-to-end latencies in our model are monotonic with respect to arrival and processing rates of queues – higher arrival rates generally lead to higher latencies, and higher processing rates lead to lower latencies. Exploiting this observation, we compute a new action a by incrementing (or decrementing) a_t, along the direction of the gradients ($\mathcal{P}_\alpha$ and $\mathcal{P}_\beta$ from Section 4.2). We repeat this process until the PERC computed using some a is within the SLO, in which case it becomes the safe action a^*.

6 Implementation

PERC *Computation:* The PERC computation algorithm (Section 4.4) perturbs the current state to observe latency changes. In our implementation, an independent daemon process handles this. It first estimates current parameters (α, β) by fitting

a Gamma distribution (Method of Moments) to latency samples from the past 60 seconds (step 2 in Section 4.4). It then applies $N = 10$ perturbations (configurable; see Algorithm 1) using small Gaussian noise, collecting new samples for 20 seconds after each perturbation to estimate updated parameters (α', β') near the prior values (step 3 in Section 4.4). To maintain stability, (α', β') are constrained close to (α, β), and noise-induced gradient outliers are clipped. A full gradient estimation cycle takes ≈210 seconds, during which controllers continue using previously computed gradients.

Controller implementation: We implement Galileo for the online-learned autoscaler Autothrottle [41] and the offline-trained admission controller TopFull [30], using their original system architectures and name them **Galileo-ASc** and **Galileo-ADM**, respectively. We retain all states, training, and measurement procedures of the original controllers. Notably, we do not require any additional data for training Galileo-ADM – *we use the same workloads to fine-tune both TopFull and Galileo-ADM*. More details are provided in Appendix D.

Galileo is open sourced at `https://github.com/ldos-project/Galileo`.

7 Evaluation

We evaluate the accuracy of PERCs and their utility in developing robust controllers by asking these questions:

- Do the computed PERCs provide an *upper bound* for latencies under environment perturbations? (Section 7.1)
- Does the integration of PERCs lead to overall improved performance for microservice controllers? (Section 7.2)
- How do the different parameters used in Galileo impact end-to-end performance? (Section 7.3)
- How much overhead is added due to the PERC computation process? (Section 7.4)

Experiment Setup. We evaluate our approach on two DeathStarBench [19] applications - Hotel Reservation (**HR**) and Social Network (**SN**). We deploy both benchmark applications using Kubernetes [2] on a 64-core cluster (4 × 16-core Intel Xeon CPU@2.0GHz, each with 32GB of RAM) on Cloudlab [15]. We use another node on the cluster to generate load using the wrk2 [3] and Locust [4] tools.

7.1 Coverage Provided by PERC Estimates

Methodology: To evaluate whether PERCs bound worst-case latencies, we must capture latencies under perturbations that scale service arrival and processing rates by a factor δ (Section 4.2). A natural approach would be to systematically perturb controllable factors across services over multiple experiments and exhaustively explore neighborhoods of various environments; but, this is prohibitively data-intensive and still may miss uncontrollable latent effects such as resource contention or queueing delays in the communication stack. Instead, we let the application *run in the wild* with microservice controllers (Autothrottle or TopFull) and analyze performance *post-facto*, recording how arrival and processing rates evolved and what tail latencies were incurred, then comparing these with the PERCs for the corresponding environment.

In particular, we run the HR and SN applications for over 250 hours under diverse workloads. These workloads span arrival rates from 0–1500 requests/sec with bursty, noisy, periodic, and stable patterns; mixed request types; and scenarios both with and without time-varying background jobs consuming 40–80% of CPUs. This setup yields a wide range of application *environments*: in each 30-second window, we record end-to-end latencies, per-request-type arrival rates, and per-service CPU usage statistics. We treat the arrival rates and CPU usage as fingerprints of the environment. To exclude drastic shifts that lie well outside the scope of our PERCs, we prune any window whose arrival-rate variance across the preceding five windows exceeds 5× its mean arrival rate. After pruning, we obtain over 12,000 data points (**Test Set**) across both applications and all request types, each representing a distinct environment.

For each E in the Test Set, we define its neighborhood via three *perturbation balls*: (i) **Small** ($\delta = 0.05$), (ii) **Medium** ($\delta = 0.1$), and (iii) **Large** ($\delta = 0.2$). A δ ball, denoted $\Delta_{E,\delta}$, contains all test environments whose per-service arrival rates and CPU usage lie within a δ factor of those in E. We then compute the PERC using latencies from E and compare it against the 99th-percentile latencies observed across $\Delta_{E,\delta}$.

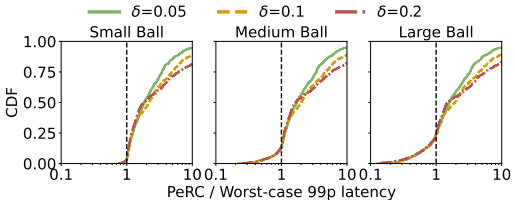
Tightness Metric: We define the Tightness metric to capture how the PERC compares to the worst-case 99p latency for a given request type:

$$Tightness_{T,\delta}(E) = \frac{P_{\varepsilon,E,T}}{max\{l_{\varepsilon,T}(E') \mid E' \in \Delta_{E,\delta}\}}$$

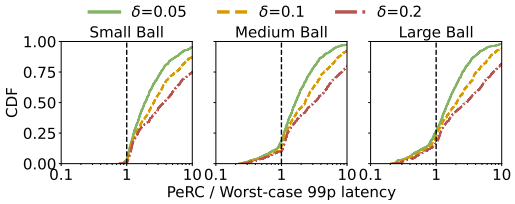
An PERC would have a tightness of 1 – a tightness larger than 1 implies that the PERC is a strict upper bound over the worst-case latencies observed empirically (desirable), while a tightness lower than 1 implies a weak bound.

Results: Figure 8 shows the CDF of Tightness over all environments in the Test Set for the `search` and `compose` request types of the HR and SN applications. We provide results for other request types in Appendix E.2. We use PERC-d to denote the PERC computed using $\delta = d$.

Coverage provided by PERCs: Figure 8 illustrates that the PERCs computed for the respective balls provide upper bounds over the worst-case latencies for most environments – PERC-0.05 provides an upper bound over the worst-case 99p latency in the small perturbation ball for 93-97% and 96-97% of the evaluated environments, across different request types in HR and SN, respectively. Similarly, PERC-0.1 and PERC-0.2 provide an upper bound for 78-93% and 75-93% of the tested environments in the medium and large perturbation balls, across the seven request types of HR and SN. In particular, **for small perturbations ($\delta = 0.05$), where our assumptions hold true, PERCs provide near-full coverage**. While larger perturbations may magnify the impact of approximations used to compute PERCs, our PERCs nevertheless



(a) HR: `search` Request Type



(b) SN: `compose` Request Type

Figure 8: Tightness of PERCs: CDF of the ratio of the computed PERC in an environment to the worst-case 99p latency under a perturbation ball. Curves close to the vertical line at 1 imply tight bounds, to the left of the vertical line imply weak bounds, and to the right of the vertical line imply a loose upper bound.

continue to provide reasonable upper bounds.

Impact of δ: As we increase the δ, PERCs provide more coverage. For the `search` request type, PERC-0.2 provides upper bounds over 4% and 9% more environments for the medium perturbation ball, compared to bounds provided by PERC-0.05 and PERC-0.1. However, a higher δ may also result in loose upper bounds: for instance, for the medium perturbation ball on the `compose` request type (Figure 8b), PERC-0.2 covers over 90% of perturbations, but the PERC is 9.7× the empirical worst-case latency on average. In contrast, PERC-0.05 and PERC-0.1 cover 81% and 85% of perturbations and are 3.2× and 5.3× higher than the worst-case empirical latency, respectively. This is because higher δs imply larger perturbations in the parameter space - some of which may not be captured in the environments we sampled. Similar trends are observed for the `search` request type for HR (see Figure 8a) and all other request types for both HR and SN (see Appendix E.2).

Does sampling hurt PERCs? Above, we have used all latency data available in a window of time, which is a common practice [41]. However, in some cases, this may not be feasible due to observability costs, necessitating sampling. We observe that reducing the number of samples only leads to a minor 3% drop in the coverage of the computed PERCs (see Appendix E.2 for detailed results).

7.2 Improved Efficacy of Galileo Controllers

Methodology: We evaluate Galileo by integrating PERCs with two microservice controllers - the Autothrottle autoscaler (**Galileo-ASc**) and the TopFull admission controller (**Galileo-ADM**). For each Galileo controller, we also use a variant that does not use shielding (Section 5.1), and only relies on the robustness reward, R_r (Section 5.2). We use a δ perturbation of 0.2 as it provides the best coverage (as shown above).

We evaluate all Galileo controllers and their baselines, separately, on the HR and SN applications over five 1-hour Alibaba traces [27] and replay them using Locust [4] with 10 threads. These traces incorporate various patterns, such as sudden bursts, gradual changes, stable arrivals, and periodic patterns, similar to the ones used in prior works [30, 41]. We evaluate the performance under two classes of environments: (i) varying arrival rates with a mix of request types (**Env-A**); and (ii) varying arrival rates with a mix of request types, in the presence of a CPU-stressing background job (**Env-B**). Together, these scenarios capture a variety of latents and environment perturbations affecting microservice performance.

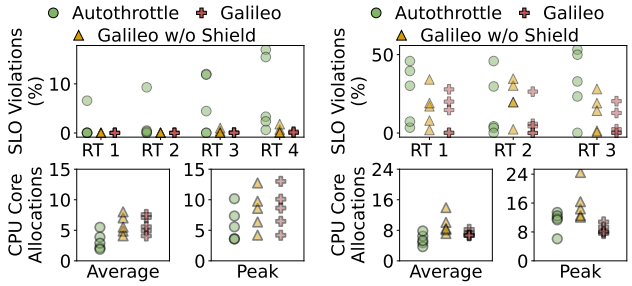
Metrics: We use SLO violations as the primary metric for comparing the Galileo controllers against the baselines, using the same *SLO of 100ms* for all request types. For the autoscaler, we additionally report the total CPU allocations, and for the admission controller, we also measure the overall goodput (the request rate permitted by the controller).

7.2.1 Galileo-ASc vs. Autothrottle

Figure 9 shows that for the Env-A family of workloads, the Galileo autoscaler is significantly better at meeting SLOs for both applications across all request types. The average SLO violations over all workloads incurred by Galileo-ASc in the HR benchmark are 98.4-99.4% fewer across the four request types than for Autothrottle (Figure 9a), **achieving just 0.05% absolute violations on average**. Similarly, across the three request types in the SN benchmark, Galileo-ASc leads to 49-76% fewer average SLO violations over all workloads, compared to Autothrottle. We observe similar 75-98% and 5-33% improvements across request types in the two benchmarks for the Env-B workloads as well (details in Appendix E.3).

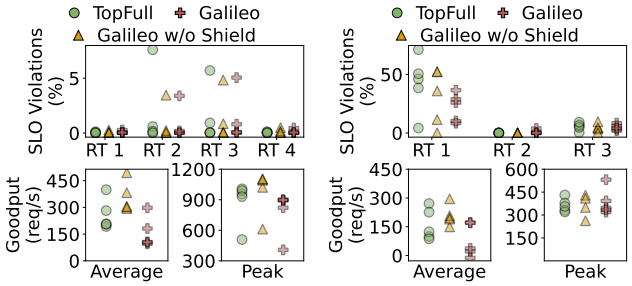
Sources of Improvement: The main reason for this improvement is Galileo-ASc's ability to quickly ramp up the core allocation *before* violations occur. This is evident in the average core allocations, where Galileo-ASc allocates up to 22% and 16% more cores than Autothrottle across all Env-A workloads, for HR and SN benchmarks, respectively. As an example, consider the time-series snippet of Galileo-ASc and Autothrottle in Figure 11a where Galileo-ASc allocates more CPUs *as soon as* latencies start to increase (at T=3.5 min; dashed vertical line in Figure 11a). In contrast, Autothrottle incurs a high cost only after several SLO violations have occurred (at T=10 min) before increasing its allocations. We observe similar results across other traces as well (see Appendix E.3 for full analysis).

Role of Shielding: Even without the shield applied, Galileo-ASc (denoted 'Galileo w/o Shield' in Figure 9) provides 93-99% and 28-60% fewer SLO violations, compared to Autothrottle, for HR and SN, respectively. This showcases the utility of the R_r function used in the training of Galileo-ASc (Section 5.2). Reaping the full benefits of PERCs, however, necessitates using shields. Consider again the example in Figure 11a, where Galileo-ASc (w/o Shield) ramps up CPU allocations because of the high PERCs computed at T=3.5



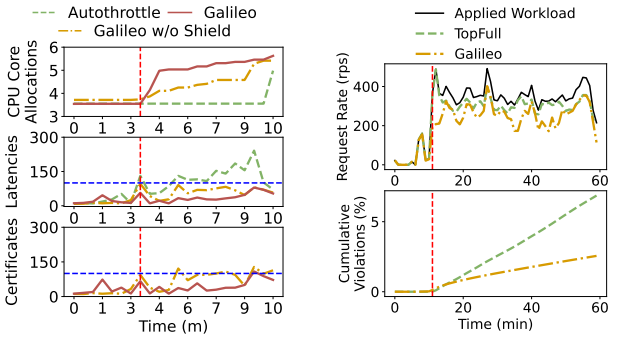
(a) Hotel Reservation (b) Social Network

Figure 9: Galileo-ASc vs. Autothrottle on workloads from Env-A. Different RTs stand for various request types (Table 2).



(a) Hotel Reservation (b) Social Network

Figure 10: Galileo-ADM vs. TopFull on workloads from Env-A. Different RTs stand for various request types (Table 2).



(a) PERCs provide high costs before latency spikes, leading to robust allocations.

(b) Galileo chooses a slightly lower rate limit to be robust to future perturbations.

Figure 11: Deep-dive into the actions of: (a) Galileo-ASc vs. Autothrottle, and (b) Galileo-ADM vs. TopFull, for one trace each - other traces have similar observations.

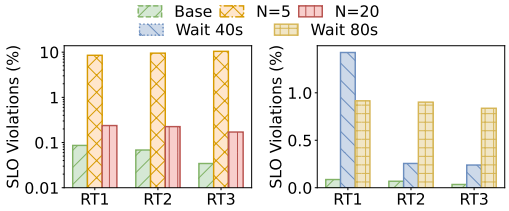


Figure 12: Sensitivity to number of perturbations and wait period.

min (third pane in Figure 11a). However, a more aggressive action is needed to keep the latencies in check. This is evident as the shield in Galileo-ASc is triggered at T=3.5 min, which overrides the small allocation increase with a *safer, larger* action (Section 5.3), thus maintaining consistently low latencies (lower even compared to Galileo-ASc w/o Shield).

Comparison against a naïve baseline: To further demonstrate that these benefits are indeed because of the PERCs and disambiguate the impact of additional rewards, we also compare a naïve 'robust' baseline, where the R_r reward function simply uses the observed 99p latency instead of PERCs. We find that this latency-based baseline still incurs nearly 5% SLO violations on average and that Galileo-ASc still provides 96-99% fewer average SLO violations across the two benchmarks (details in Appendix E.3.3).

7.2.2 Galileo-ADM vs. TopFull

Figure 10 shows that the Galileo-ADM generally incurs significantly smaller SLO violations for both benchmark applications for the Env-A workloads. Across the request types for the HR benchmark, Galileo-ADM incurs 0-6% fewer SLO violations than TopFull, averaged over the Env-A workloads.

Overall, Galileo-ADM achieves 0.5% average SLO violations for HR. Across the three request types in the Social Network benchmark, Galileo-ADM leads to 1-60% fewer SLO violations averaged over the Env-A environments. We observe 2-46% improvements in SLO violations across the two benchmarks as well for the Env-B workloads, and provide the detailed results in Appendix E.

Sources of Improvement: The higher SLO violations incurred by TopFull are because it is not conservative enough in setting the rate limit. Figures 10a and 10b show that Galileo-ADM achieves the significant improvements in SLO violations by (slightly) lowering the average goodput across Env-A workloads, by 40% and 35%, and peak goodputs by only 10% and 6%, for the HR and SN benchmarks, respectively. As an example, Figure 11b's cumulative violations show that while TopFull continues to use a high rate limit despite seeing SLO violations, the PERCs and shields used during the training of Galileo allow it to set a more conservative rate limit.

7.3 Sensitivity Analysis

We evaluate the sensitivity of Galileo controllers to the hyperparameters used in PERC computation. Gradient estimates rely on $N = 10$ perturbations with a wait of $p = 20$ seconds after each. We tweak N and p. Figure 12 shows that a smaller number of perturbations $N = 5$ reduces the accuracy of the gradients leading to poor end-to-end performance. On the other hand, $N = 20$ provides more accurate estimates but takes too long – roughly 8 minutes for a single gradient estimate, which also degrades the overall performance. Similarly, waiting too long ($p = 40$s or $p = 80$s) leads to longer gradient estimation process, degrading overall performance. Note that in spite of these choices, SLO violations continue to be low.

7.4 Overheads of Computing PERCs at Run-time

For Galileo controllers, PERCs are computed at run-time by a daemon process, in two key steps: fitting Gamma distributions to latency samples, and computing gradients using perturbed parameters (Section 6). We measure the time to fit Gamma parameters (α, β), as a function of the applied load. We observe that the time to fit Gamma parameters increases as the workload increases. This is expected because a higher load implies more samples, and hence larger time to fit the model. However, even for load up to 500 rps, this executes within 15ms in the tail (see Appendix E.5). We also measure the time spent in *computing* gradients from the perturbed values, across all traces from the Env-A and Env-B workloads, and find that it is 1.3ms on average, with a maximum of up to 27ms. The computational overhead of both of these steps is negligible compared to the total PERC computation loop of about 210 seconds.

8 Discussion

What happens when assumptions underlying the Performance Reasoning Model break? In computing PERCs (per Equation (11)), we rely on three key assumptions: (i) the microservices invoked by a request type in a given environment form an overtake-free network of $M/M/1$-PS queues, (ii) the environment remains largely stable over the duration of collecting samples to compute PERCs, and (iii) the perturbations are small enough for Taylor approximation in Equation (8) to hold. Real-world deployments may violate these assumptions, in which case PERCs can be weaker bounds than the true worst-case latencies. Crucially, the lightweight nature of PERC computation enables frequent recomputation, limiting the impact of such modeling inaccuracies. Further, our empirical study across the two benchmarks shows that such inaccuracies are infrequent – even for `compose` request type under large perturbation balls (where the above assumptions may not strictly hold), PERCs still provide coverage for over 73% of all observed environments (see Figure 8b). On the other hand, guaranteeing performance under failure events, e.g., a service crashing and stopping completely, requires mechanisms other than the controllers studied in this work and falls outside the scope of environment perturbations.

Can PERCs be applied to other microservice controllers in diverse settings? PERCs are general abstractions that extend to diverse settings, including clusters over heterogeneous hardware or multi-replica deployments. Our technique relies on empirical data to fit ditributions and derive gradients, which naturally applies to both of these settings. While this paper applies PERCs to autoscalers and admission controllers, these performance bounds can similarly guide other controllers, such as load balancers. However, constructing shields for load balancers introduces the challenge of precisely mapping a load-balancing decision to its induced changes in arrival and processing rates. Addressing this mapping remains an important direction for future work.

9 Related Work

Microservice Controllers. Prior work [28,31,32,34,41,44,45] has explored both learned and non-learned approaches for autoscaling. Learned approaches that train deep neural networks using extensive instrumentation [31,44] depend heavily on high-quality training data. Analytical models [28, 34, 45] capture end-to-end latency under varying resources and arrival rates, yet overlook perturbations, limiting robustness. AutoThrottle [41], as shown in Section 2.2, similarly lacks robustness. In general, existing autoscalers fail to model diverse environmental perturbations and their impact on latency. For admission control, non-learned methods such as Dagor and Wisp [38, 46] use priorities and thresholds but risk starvation [30]. TopFull [30] applies RL-based rate limiting but remains insensitive to latency perturbations and is non-robust.

Formal Analysis of Distributed Systems. Prior work on formal system analysis has largely targeted worst-case properties such as correctness [18, 35], liveness [37], and reliability [36]. While valuable for understanding large systems, these approaches cannot guide microservice controllers since they ignore performance. Recent efforts [6, 7] analyze network performance but rely on heavyweight solvers and static models, making them unsuitable for real-time PERC computation. Other methods estimate performance bounds via static analysis [22, 29] or fine-grained code execution measurements [11]. Similar techniques have been used in embedded systems [26, 33, 42]. All of these require extensive per-component instrumentation and struggle to compose individual measurements into end-to-end tail guarantees. Performal [43] combines formal analysis with real-time measurements but assumes Gaussian delay distributions, a model that is inaccurate (Section 4.1).

Bounds for Queueing Networks Traditional queueing-theoretic bounds [12, 13, 16, 24] are either deterministic network-calculus bounds, which are often loose, or stochastic MGF-based bounds, which are accurate mainly in large-delay or many-flow regimes. In contrast, our goal is to derive tail bounds that are easily parameterizable using "what-if" analysis, without relying on such scaling assumptions.

10 Conclusion

Ensuring robust performance in microservice applications is critical for meeting SLOs in dynamic environments. Existing controllers struggle with robustness due to their reliance on instantaneous performance metrics, which fail to account for latent environmental factors. To address this gap, we introduced PERCs, which provide statistical performance bounds under a range of perturbations. By leveraging a queueing-theoretic model, we enable efficient PERC computation, making PERCs suitable for real-time decision-making. Our framework, Galileo, integrates PERCs with controllers, improving their resilience. Through extensive evaluation, we demonstrate that our queueing modeling is accurate, PERCs are tight, and Galileo significantly reduces SLO violations.

Acknowledgments

We would like to thank our shepherd, Rishabh Iyer, and the anonymous reviewers for their invaluable feedback. We also thank members of the UTNS lab for their feedback. This material is based upon work supported by the U.S. National Science Foundation (NSF) under Grant Number 2326576, and in part under NSF Grants CNS-2107037, CNS-2112471, and ONR Grant N000142412779.

References

[1] uber.com – Introducing Domain-Oriented Microservice Architecture. https://www.uber.com/en-IN/blog/microservice-architecture/, 2020.

[2] Kubernetes. https://kubernetes.io/, 2023.

[3] wrk2. https://github.com/giltene/wrk2, 2023.

[4] Locust: An Open Source Load Testing Tool. https://locust.io/, 2024.

[5] Mohammed Alshiekh, Roderick Bloem, Rüdiger Ehlers, Bettina Könighofer, Scott Niekum, and Ufuk Topcu. Safe reinforcement learning via shielding. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, AAAI'18/IAAI'18/EAAI'18, 2018.

[6] Mina Tahmasbi Arashloo, Ryan Beckett, and Rachit Agarwal. Formal methods for network performance analysis. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 645–661, April 2023.

[7] Venkat Arun, Mina Tahmasbi Arashloo, Ahmed Saeed, Mohammad Alizadeh, and Hari Balakrishnan. Toward formally verifying congestion control behavior. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 1–16, 2021.

[8] Arko Banerjee, Kia Rahmani, Joydeep Biswas, and Isil Dillig. Dynamic model predictive shielding for provably safe reinforcement learning. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[9] Osbert Bastani. Safe reinforcement learning with nonlinear dynamics via model predictive shielding. In *2021 American Control Conference (ACC)*, pages 3488–3494, 2021.

[10] Jeremy Carleton, Prathik Vijaykumar, Divyanshu Saxena, Dheeraj Narasimha, Srinivas Shakkottai, and Aditya Akella. CONGO: Compressive online gradient optimization. In *The Thirteenth International Conference on Learning Representations*, 2025.

[11] Francisco J Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella. Upper-bounding program execution time with extreme value theory. In *13th International Workshop on Worst-Case Execution Time Analysis (2013)*. Schloss-Dagstuhl-Leibniz Zentrum für Informatik, 2013.

[12] Cheng-Shang Chang. On deterministic traffic regulation and service guarantees: a systematic approach by filtering. *IEEE Transactions on Information Theory*, 44(3):1097–1110, 1998.

[13] Florin Ciucu, Almut Burchard, and Jörg Liebeherr. A network service curve approach for the stochastic analysis of networks. In *Proceedings of the 2005 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '05, page 279–290, 2005.

[14] R. L. Dobrushin. Prescribing a system of random variables by conditional distributions. *Theory of Probability & Its Applications*, 15(3):458–486, 1970.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. The design and operation of CloudLab. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 1–14, July 2019.

[16] Markus Fidler. An end-to-end probabilistic network calculus with moment generating functions. In *2006, 14th IEEE International Workshop on Quality of Service*, pages 261–270, 2006.

[17] Abraham D. Flaxman, Adam Tauman Kalai, and H. Brendan McMahan. Online convex optimization in the bandit setting: gradient descent without a gradient. In *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, page 385–394. Society for Industrial and Applied Mathematics, 2005.

[18] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 469–483, May 2015.

[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang

Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. 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*, ASPLOS '19, pages 3–18, 2019.

[20] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina 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*, ASPLOS '19, page 19–33, 2019.

[21] Darby Huye, Yuri Shkuro, and Raja R. Sambasivan. Lifting the veil on Meta's microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, July 2023.

[22] Rishabh Iyer, Katerina Argyraki, and George Candea. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, April 2022.

[23] Ajaykrishna Karthikeyan, Nagarajan Natarajan, Gagan Somashekar, Lei Zhao, Ranjita Bhagwan, Rodrigo Fonseca, Tatiana Racheva, and Yogesh Bansal. SelfTune: Tuning cluster managers. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 1097–1114, April 2023.

[24] J.-Y. Le Boudec. Application of network calculus to guaranteed service networks. *IEEE Transactions on Information Theory*, 44(3):1087–1096, 1998.

[25] Shuo Li and Osbert Bastani. Robust model predictive shielding for safe reinforcement learning with stochastic dynamics, 2020.

[26] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007.

[27] 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, 2021.

[28] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Jian He, Guodong Yang, and Chengzhong Xu. Erms: Efficient resource management for shared microservices with sla guarantees. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ASPLOS 2023, page 62–77, 2022.

[29] Jiacheng Ma, Rishabh Iyer, Sahand Kashani, Mahyar Emami, Thomas Bourgeat, and George Candea. Performance interfaces for hardware accelerators. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 855–874, July 2024.

[30] 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, 2024.

[31] 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, November 2020.

[32] Vighnesh Sachidananda and Anirudh Sivaraman. Erlang: Application-aware autoscaling for cloud microservices. In *Proceedings of the Nineteenth European Conference on Computer Systems*, EuroSys '24, page 888–923. Association for Computing Machinery, 2024.

[33] Thomas Sewell, Felix Kam, and Gernot Heiser. High-assurance timing analysis for a high-assurance real-time operating system. *Real-Time Systems*, 53(5):812–853, 2017.

[34] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. Nodens: Enabling resource efficient and fast QoS recovery of dynamic microservice applications in datacenters. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 403–417, July 2023.

[35] Kausik Subramanian, Anubhavnidhi Abhashkumar, Loris D'Antoni, and Aditya Akella. Detecting network load violations for distributed control planes. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 974–988, 2020.

[36] Xudong Sun, Wenqing Luo, Jiawei Tyler Gu, Aishwarya Ganesan, Ramnatthan Alagappan, Michael Gasch, Lalith Suresh, and Tianyin Xu. Automatic reliability testing for cluster management controllers. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 143–159, July 2022.

[37] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 649–666, July 2024.

[38] Lalith Suresh, Peter Bodik, Ishai Menache, Marco Canini, and Florin Ciucu. Distributed resource management across process boundaries. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, page 611–623, 2017.

[39] J. Walrand and P. Varaiya. Sojourn times and the overtaking condition in jacksonian networks. *Advances in Applied Probability*, 12(4):1000–1018, 1980.

[40] Jean C. Walrand. An introduction to queueing networks. In *Prentice Hall International editions*, 1989.

[41] 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, April 2024.

[42] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.

[43] Tony Nuda Zhang, Upamanyu Sharma, and Manos Kapritsos. Performal: Formal verification of latency properties for distributed systems. *Proc. ACM Program. Lang.*, 7(PLDI), June 2023.

[44] 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*, ASPLOS '21, page 167–181, 2021.

[45] Yanqi Zhang, Zhuangzhuang Zhou, Sameh Elnikety, and Christina Delimitrou. Ursa: Lightweight resource management for cloud-native microservices. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 954–969, 2024.

[46] 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, page 149–161, 2018.

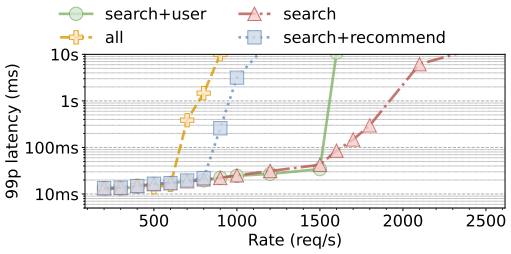
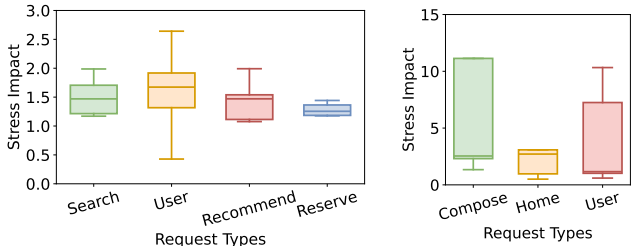


Figure 13: Performance variability for the `search` request type of HotelReservation benchmark application.



(a) Hotel Reservation (b) Social Network

Figure 14: Impact of background CPU stress jobs on 99p latencies of `search` and `compose` request types for Hotel Reservation and Social Network benchmarks, respectively.

A Performance Variations in Microservices

We repeat the experiment as mentioned in Section 2.1 with the Hotel Reservation benchmark. Figure 13 shows the performance variations for the `search` request type, as the arrival workload and the concurrent load is changed. A mere 6.6% increase in the request rate (from 1500 to 1600 req/sec) leads to a 1.7× increase in the latencies. As we observed with the Social Network application in Section 2.1, the variations are impacted by the presence of concurrent loads.

Figure 14 shows the impact of the presence of a background CPU stress job on various request types of the Hotel Reservation and Social Network benchmarks. We generate several workloads and measure end-to-end latencies in the presence and absence of background jobs. Then we compute the Stress Impact as the ratio of the 99p latencies in the presence of background jobs versus in nostress conditions. For the Hotel Reservation application, different request types can experience 1.09-1.55× increased 99p latencies on average, whereas for the Social Network application, request types can experience 2.4-4.7× higher 99p latencies on average.

B Efficacy of Existing Learned Controllers

B.1 Efficacy of TopFull in Adapting to Workload Changes

Similar to the issues we observed in Autothrottle, TopFull's admission control mechanism reacts too late to prevent SLO violations, leading to 99th-percentile latencies that can be up to 20× higher than the desired SLO (see Figure 15). The solid vertical line in Figure 15 shows that the increase in the

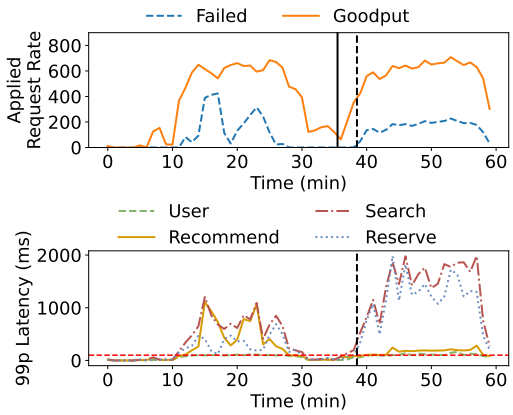
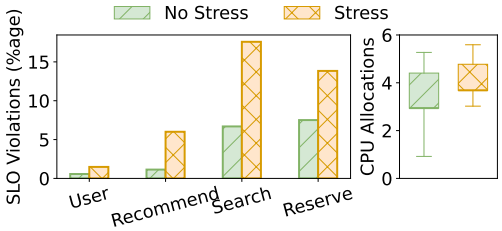
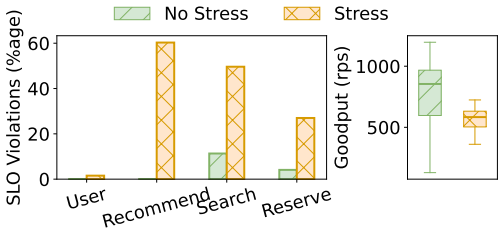


Figure 15: Execution of the TopFull [30] controller over a 1-hour trace from Alibaba traces [27] on the Hotel Reservation benchmark. Horizontal dashed line shows the SLO (100ms).



(a) Autothrottle learned autoscaler.



(b) TopFull admission controller.

Figure 16: Impact of running background CPU stress jobs on end-to-end performance of microservice controllers for the Hotel Reservation benchmark.

arrival rates starts at T=35.5 min, but the controller rate-limits requests at T=38.5 min when SLO violations start (see the dashed vertical line in Figure 15). Like Autothrottle, TopFull's reliance on just observed metrics precludes any information about environment perturbations, causing severe SLO violations.

B.2 Performance in the Presence of Background Jobs

These issues are exacerbated when these learned controllers are deployed in the presence of contending background jobs. Similar to Section 2.1, we start a CPU stressor job that consumes 50-80% of the CPUs in a time-varying manner and measure the end-to-end performance of Autothrottle and TopFull over a bursty 1-hour trace from the Alibaba trace dataset. Our findings are illustrated in Figure 16.

In the case of Autothrottle, SLO violations increase by

up to 3× for certain request types while for TopFull, the same background CPU stress job can lead to up to 60% SLO violations. Note that while both controllers do take the right action under the presence of background jobs – Autothrottle increases the overall CPU allocations and TopFull reduces the effective rate limit, the high SLO violation percentage indicate that the actions were not aggressive enough.

C Additional Details of the PRM

This section provides additional formalism and mathematical details for the Performance Reasoning Model (PRM) introduced in Section 4.

C.1 Semantics for Composition of Actions

The PRM utilizes two actions, namely $\mathsf{perturb}_\mu$ and $\mathsf{perturb}_\lambda$, for every queue in the abstract network of queues (Equation (4)). Actions across multiple queues can be easily composed together using the following semantics:

$$\begin{aligned}\forall i,j,\eta_X,\eta_Y.[\![(\mathsf{perturb}_X\ i\ \eta_X)\circ(\mathsf{perturb}_Y\ j\ \eta_Y)]\!](\mathcal{M}_r)\\ = \mathcal{M}_r[X_i \mapsto X_i+\eta_X X_i, Y_j \mapsto Y_j+\eta_Y Y_j,\\ \alpha \mapsto \alpha+\mathcal{P}_\alpha(X_i,\eta_X)+\mathcal{P}_\alpha(Y_j,\eta_Y),\\ \beta \mapsto \beta+\mathcal{P}_\beta(X_i,\eta_X)+\mathcal{P}_\beta(Y_j,\eta_Y)]\end{aligned} \tag{14}$$

where X,Y can be either of μ or λ and a,b are perturbations to different service parameters. Note that the effect of the two perturbations can be simply added because, under our modeling assumption, the sojourn times at each queue are independent (Section 4.1).

C.2 Expressions for Perturbation Functions

The *semantics* of $\mathsf{perturb}_\mu$ (and $\mathsf{perturb}_\lambda$) actions is defined as (same as Equation (5); repeated for completeness):

$$\begin{aligned}\forall i,\eta.[\![\mathsf{perturb}_\mu\ i\ \eta]\!](\mathcal{M}_r) = \mathcal{M}_r[\mu_i \mapsto \mu_i+\eta\mu_i,\\ \alpha \mapsto \alpha+\mathcal{P}_\alpha(\mu_i,\eta),\\ \beta \mapsto \beta+\mathcal{P}_\beta(\mu_i,\eta)]\end{aligned} \tag{15}$$

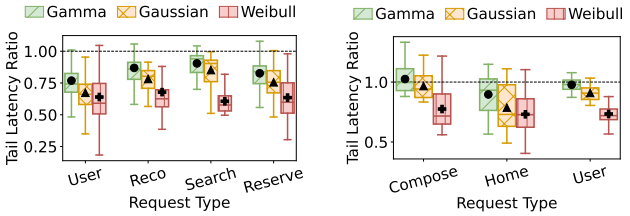
where the functions $\mathcal{P}_\alpha(\mu_i,\eta)$ and $\mathcal{P}_\beta(\mu_i,\eta)$ are the *perturbation functions*, that denote how much the distribution parameters α,β change when the processing rate μ_i of the queue i changes by a factor η.

Consider α as a function over service parameters, $\alpha = f_\alpha(\mu_1,\mu_2,\ldots,\lambda_1,\lambda_2,\ldots)$ (as in Equation (2)). Then for small η in Equation (5), Taylor expansion gives the following:

$$f_\alpha(\ldots,\mu_i+\eta\mu_i,\ldots) \approx f_\alpha(\ldots,\mu_i,\ldots)+\eta\mu_i\frac{\partial f_\alpha}{\partial \mu_i} \tag{16}$$

Comparing the above against Equation (5), we can express $\mathcal{P}_\alpha(\mu_i,\eta)$ as $\eta\mu_i(\partial f_\alpha/\partial\mu_i)$. To compute this, we would like to know the *gradient* of α with respect to the service parameter. Since the gradient function is not known, we leverage gradient-estimation techniques that allow calculation of the gradient of a function with respect to a given parameter without knowing the first-order function [17, 23]:

$$\mu_i\frac{\partial f_\alpha}{\partial \mu_i} \approx \frac{f_\alpha(\ldots,\mu_i+\eta\mu_i,\ldots)-f_\alpha(\ldots,\mu_i,\ldots)}{\eta} = \mathcal{K}_{\alpha,\mu_i} \tag{17}$$



(a) Hotel Reservation (b) Social Network

Figure 17: Ratio of the 99p data point on the fitted distribution to the actual 99p latencies. Markers show the mean values. Closer to 1 implies higher accuracy.

for some small η. Here, η serves only as a probe to estimate the local gradient. Once computed, we treat $\mathcal{K}_{\alpha,\mu_i}$ as approximately constant in the neighborhood of μ_i, allowing us to evaluate $\mathcal{P}_\alpha$ for arbitrary η values (as long as the perturbation remains within the small-η regime), as follows:

$$\mathcal{P}_\alpha(\mu_i,\eta) = \eta\,\mathcal{K}_{\alpha,\mu_i} \tag{18}$$

This provides us the result in Equation (8).

D Implementation Details

Table 1 shows the states and reward functions used by the two controllers, Autothrottle and TopFull, and how Galileo modifies them.

Galileo-ASc: We use a completely online controller for Autothrottle and Galileo-ASc. For the contextual bandit controller, we keep all hyperparameters same except for the exploration rate, which we set to 0.1, as reported in the original paper [41]. We use the same architecture as the original implementation but for the communication between the controller and the client (to obtain the latency and workload statistics) - we use gRPC. We observed better performance with the step size=30 seconds for Autothrottle on the bursty traces we considered (instead of the default 1 min suggested in the original paper [41]), and hence use the step size as 30s. We use the same 30-second step size for both Autothrottle and Galileo-ASc, leveraging all available latency data from the past 30 seconds to compute the PERC in R_r, with $w = 16$ chosen through hyperparameter tuning.

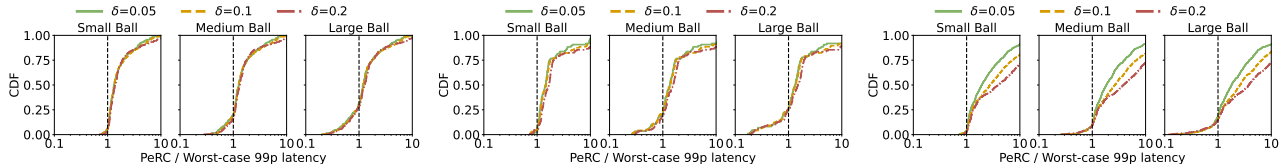
Galileo-ADM: TopFull uses two-step training: first on a Simulator with artificial DAGs, then fine-tuned on the real application. Since live latencies are not available for the first step (necessary for computing PERCs), we only apply the R_r reward function during fine-tuning. For both steps, we use the same set of hyperparameters as reported in the original paper [30]. For the second step, we fine-tune the provided base model (trained on the Simulator) for 800 episodes on each microservice – for both TopFull and Galileo-ADM.

Controller	Agent Description	Vanilla Reward Function, R_o	Updated Reward Function with PERCs

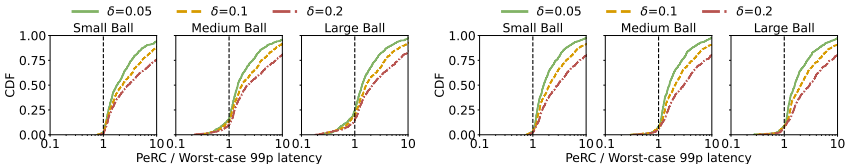
Autothrottle autoscaler [41]	Online-learned, contextual-bandit agent to decide throttle ratio targets for services, given a particular arrival request rate	If latencies violate the SLO, minimize latencies else minimize the allocations. The cost function seeks to balance minimal allocations and optimal latencies.
		`if latency <= SLO:`
		`  cost = allocation`
		`else:`
		`  cost = latency + 2`
			Add a η∗sigmoid(PeRC - SLO) term to the cost function, where η is a hyper-parameter.
			`if latency <= SLO:`
			`  cost = allocation`
			`else:`
			`  cost = latency + 2`
			`cost += η∗sigmoid(PeRC - SLO)`

TopFull admission controller [30]	Offline-trained, RL agent that uses PPO to learn the rate limit given the current goodput and observed latencies	Maximize the following reward function, where ΔGoodput corresponds to the change in the rate of successful requests, minus a penalty for violating the latency SLO.
		`reward = ΔGoodput -`
		`ρ∗max(0, latency - SLO)`
			Provides another penalty of the form η∗sigmoid(PeRC - SLO), where η is a hyper-parameter.
			`reward = ΔGoodput -`
			`ρ∗max(0, latency - SLO) -`
			`η∗sigmoid(PeRC - SLO)`

Table 1: Description of controllers where we applied PERCs and the modified reward functions. Highlighted text shows the augmented R_r reward function. PERCs are calculated for a pre-defined value of ε.



(a) HR: `recommend` Request Type (b) HR: `user` Request Type (c) HR: `reserve` Request Type



(d) SN: `home` Request Type (e) SN: `user` Request Type

Figure 18: Tightness of PERCs on the request types of: (a)-(c) Hotel Reservation, and (d)-(e) Social Network benchmarks. `search` and `compose` request types are shown in Section 7.1

	RT1	RT2	RT3	RT4
HR	user	recommend	search	reserve
SN	compose	home	user	–

Table 2: Mapping of request types (RT) in HR and SN apps.

E Detailed Evaluation Results

E.1 Complete Results for Empirical Validation of Our Request Processing Model

Methodology: We run the HR and SN applications from DeathStarBench under 150 different, but controlled environments using the wrk2 [3] tool, with varying request rates (200-1000 reqs/sec), arrival processes (exponential and zipfian), and various request type mixes. For each environment, we collect end-to-end latencies for each request type, fit a Gamma distribution to 10% of the collected samples using Method of moments, and measure similarity metrics (more below) between the fitted distribution and 100% of the collected samples for that environment. Our environment set is diverse – we observe 99p latencies ranging between 1ms to 10s across all request types.

Metrics: We use two metrics to evaluate the similarity: (i) Wasserstein distance [14] to measure the magnitude of difference between two probability distributions; and (ii) Tail latency ratio (ratio of the 99p data point on the fitted distribution to the ground truth 99p latency).

Baselines: An alternate method of approximating the end-to-end latency distributions is by using a Gaussian distribution (denoted **baseline**) as proposed previously in [43] for performance analysis of distributed systems. Similarly, heavy-tailed distributions are often thought to closely match system latencies, so we choose the Weibull distribution as a representative heavy-tailed distribution.

Results: The results for Wasserstein distances are presented in Section 4.1. We present the results for Tail latency ratios here. Improved modeling of end-to-end latency distributions results in more accurate tail percentile estimates, as demonstrated in Figure 17. Across the different request types of the HR application, the tail latency ratios in the Gamma modeling are 6-16% better than the baseline, and in terms of absolute values, the 99p data points derived via the Gamma distribution are 0.83-0.89× the ground truth 99p latencies on average. Similarly, for the SN application, the ratios in the Gamma modeling are 6-15% better than the baseline on average, and

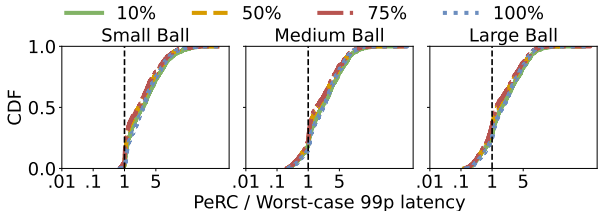


Figure 19: Impact of reducing the samples used for computing PERCs for the `compose` request type. PERCs computed for δ=0.2

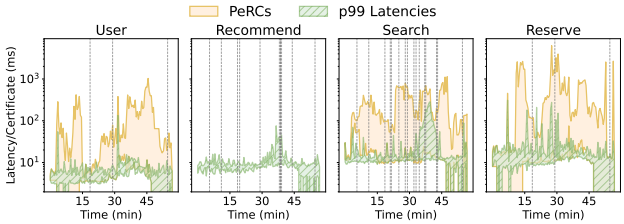


Figure 20: Trends of PERCs and p99 latencies. Vertical lines signify an increase in the total CPU allocation for services invoked by the request type.

the 99p data points are 0.93-1.01× of the ground truth 99p latencies, on average.

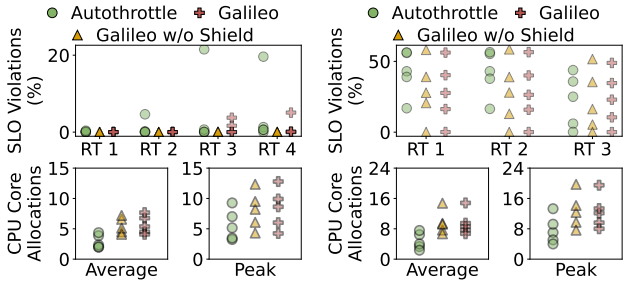
Sources of Improvement: The underlying reason for the improvement over the baselines is that the Gaussian distribution is not well-suited for dynamic microservice applications where the variances can be very high, leading to distributions with tails heavier than Gaussian's. Furthermore, a Gaussian fit can also result in negative values of latencies if the mean end-to-end latencies are small.

E.2 Complete Results for Tightness of PERCs

Figure 18 reiterates our findings from Section 7.1 that PERCs can provide tight bounds for various perturbation balls. For different amounts of perturbations, different δ values must be used for computing PERCs. Overall, using PERC-0.05[4] can provide 93-97% coverage over small perturbation balls, PERC-0.1 can provide 78-93% coverage over medium perturbation balls and PERC-0.2 can provide 75-93% coverage over large perturbation balls across the various request types of the HR benchmark (Figure 18).

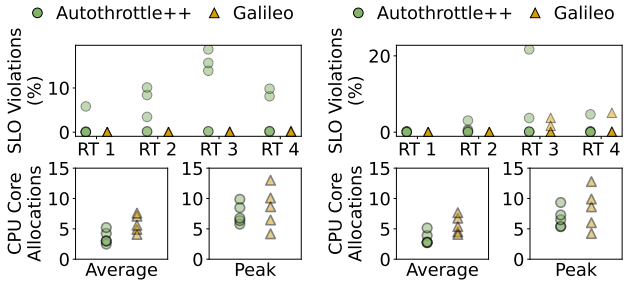
We also provide the full results for the sampling efficiency in Figure 19. We observe that reducing the number of samples does not lead to a significant drop in the accuracy of the computed PERCs. For the `compose` request type, even 10% of the samples collected are enough to get 97%, 81%, and 73% coverage for small, medium, and large perturbation balls, respectively, which is only slightly smaller than the coverage of 97%, 84%, and 75%, obtained from PERCs computed with all samples.

[4]Reusing the same notation as in Section 7.1



(a) HR: Env-B (b) SN: Env-B

Figure 21: Galileo-ASc vs. Autothrottle on: (a) Hotel Reservation; and (b) Social Network for Env-B.



(a) Env-A (b) Env-B

Figure 22: Galileo-ASc vs. Autothrottle++ on Hotel Reservation benchmark for (a) Env-A and (b) Env-B.

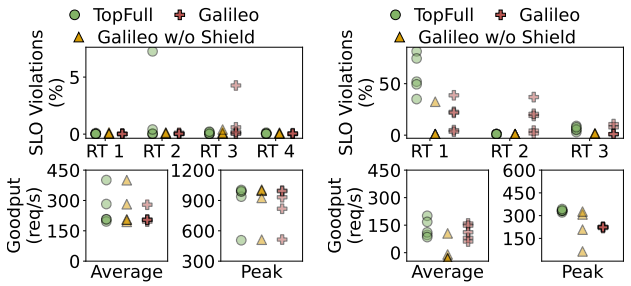
E.3 Additional Results for Galileo-ASc vs. Autothrottle

E.3.1 Deep dive into sources of improvement

In Section 7.2.1, we show that Galileo-ASc results in very few SLO violations and claim that this is because of Galileo's capability to make robust allocation decisions which are resistant to latent environment changes. While Figure 11a shows this for one such trace, we also observe similar results on other traces as well. Figure 20 demonstrates this across all traces for the HR application. Notably, the range of computed PERCs is higher than corresponding p99 latencies – this is because PERCs provide an upper bound over worst-case p99 latencies. We also observe allocation increases in the period T=15 min to T=45 min for all four request types, even though p99 latencies are within SLOs. This is because the PERCs breach SLOs and hence, Galileo-ASc decides to increase allocations in an attempt to make robust allocation decisions.

E.3.2 Results for the Env-B workloads

Continuing our findings in Section 7.2.1, Figure 21 shows that Galileo-ASc incurs between 62-75% fewer SLO violations across the four request types of HR and between 41-65% fewer SLO violations across the three request types of SN. Similar to the Env-A workloads, the improvements can be attributed to more responsive adjustments to allocations, 1-17% higher average allocations, and up to 15% higher average peak core allocation by Galileo-ASc.



(a) HR: Env-B (b) SN: Env-B

Figure 23: Galileo-ADM vs. TopFull on: (a) Hotel Reservation; and (b) Social Network application for Env-B.

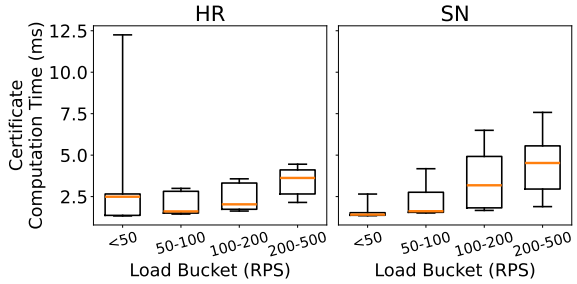


Figure 24: Computation time to fit Gamma parameters as a function of the applied load.

E.3.3 Comparison against a Naïve Baseline

We construct Autothrottle++, where we keep the same reward function as Galileo-ASc but replace the PERCs in the R_r reward with the scaled 99p latencies; and evaluate Autothrottle++ against Galileo-ASc for the HR benchmark. Figure 22 shows that Galileo-ASc outperforms Autothrottle++. Galileo-ASc incurs 28% and 27% fewer average violations across all workloads and request types for the HR Benchmark in Env-A and Env-B respectively, with only 1.6% and 6% greater average allocations. These results show that certificates provide additional context that enables more robust decision-making, and that a naïve incorporation of latencies into the controller reward function leaves performance on the table.

E.4 Additional Results for Galileo-ADM vs. TopFull

Continuing our findings in Section 7.2.2, Figure 23 shows that Galileo-ADM incurs 0-6% fewer SLO violations, on average, across the four request types of HR and 14-42% fewer SLO violations, on average, across the three request types of SN. Similar to the Env-A workloads, the root cause for this improvement is the slightly lower (1-24% on average) goodput allowed by Galileo-ADM.

E.5 Additional Results for Overhead Analysis

We measure the time to fit Gamma parameters (α, β), as a function of the applied load. Figure 24 shows the computation times for different load buckets for the HR and SN benchmark applications. We observe that the time to fit Gamma parameters increases with the applied load. This is expected because a higher load implies more samples, and hence larger time to fit the model. However, even for load up to 500 rps, this executes within 15ms in the tail. We observe some samples with high computation even for low loads for the HR application – we attribute this to the fact that when samples are too few, it can lead to incorrect parameters and additional checks.