Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices

Zibo Wang, *University of Science and Technology of China and Microsoft Research;* Pinghe Li, *ETH Zurich;* Chieh-Jan Mike Liang, *Microsoft Research;* Feng Wu, *University of Science and Technology of China;* Francis Y. Yan, *Microsoft Research*

https://www.usenix.org/conference/nsdi24/presentation/wang-zibo

This paper is included in the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation.

April 16–18, 2024 • Santa Clara, CA, USA

978-1-939133-39-7

Open access to the Proceedings of the 21st USENIX Symposium on Networked Systems Design and Implementation is sponsored by

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

Autothrottle: A Practical Bi-Level Approach to Resource Management for SLO-Targeted Microservices

Zibo Wang[†§], Pinghe Li[¶], Chieh-Jan Mike Liang[§], Feng Wu[†], Francis Y. Yan[§]
[†]University of Science and Technology of China, [¶]ETH Zurich, [§]Microsoft Research

Abstract

Achieving resource efficiency while preserving end-user experience is non-trivial for cloud application operators. As cloud applications progressively adopt microservices, resource managers are faced with two distinct levels of system behavior: end-to-end application latency and per-service resource usage. Translating between the two levels, however, is challenging because user requests traverse heterogeneous services that collectively (but unevenly) contribute to the end-to-end latency. We present *Autothrottle*, a bi-level resource management framework for microservices with latency SLOs (service-level objectives). It architecturally decouples application SLO feedback from service resource control, and bridges them through the notion of performance targets. Specifically, an application-wide learning-based controller is employed to periodically set performance targets—expressed as CPU throttle ratios—for per-service heuristic controllers to attain. We evaluate Autothrottle on three microservice applications, with workload traces from production scenarios. Results show superior CPU savings, up to 26.21% over the best-performing baseline and up to 93.84% over all baselines.

1 Introduction

To ensure a seamless end-user experience, many user-facing latency-sensitive applications impose an SLO (service-level objective) on the end-to-end latency. Traditionally, cloud application operators resort to resource over-provisioning to avoid SLO violations, yet doing so unnecessarily wastes resources [21, 32]. Previous efforts have demonstrated significant savings if the excess resources could be harvested or reclaimed for co-located applications in a multi-tenant environment [13, 29, 30, 39, 57, 62].

A key enabler for such resource saving is SLO-targeted resource management. Its goal is to continuously minimize the total resources allocated, while still satisfying the end-to-end latency SLO. Unfortunately, modern cloud applications can be beyond current resource managers, due to the progressive shift from monolithic to distributed architecture [11, 25, 31, 40, 64]. They are a topology of cloud-native services or microservices[1], and user requests traverse a chain of execution dependencies among services of logic, databases, and machine learning (ML) model serving. Notably, this creates distinct levels of system behavior—the macro perspective reveals the end-to-end performance (e.g., user request latencies) and SLO, and the micro perspective is scoped to local measurements (e.g., service CPU usage) and resource control.

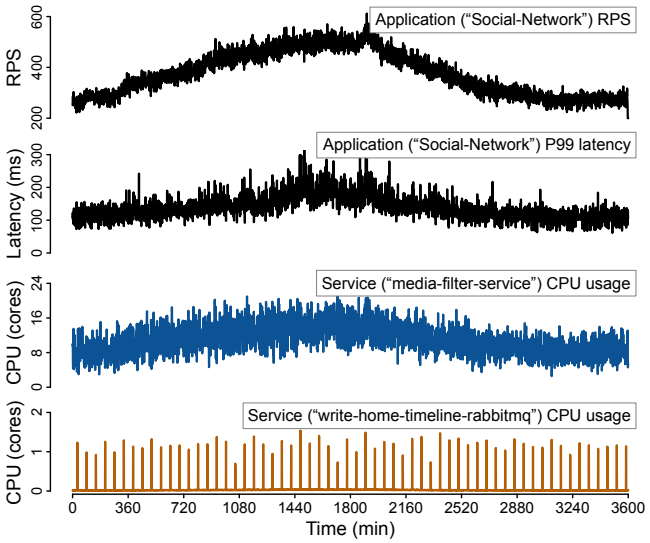


Figure 1: Individual microservices (bottom two panels) can exhibit vastly different resource usage patterns and short-term fluctuations. In addition, they do not necessarily have a strong correlation with the end-to-end application-level measurements (top two panels).

The distributed nature of microservices brings unique implications to resource management. First, heterogeneous services can exhibit vastly different resource usage patterns, due to how various user requests stress each service. The bottom two panels in Figure 1 contrast the CPU usage of two services in an application, Social-Network [22]. Second, application performance and per-service resource usage are measurements at different levels, without necessarily exhibiting a strong correlation (top two vs. bottom two panels in Figure 1). Translating between them requires knowing each user request's actual resource requirements and its service-to-service execution flow. Moreover, this execution chain incurs undesirable delays in observing effects of allocation changes on the end-to-end performance, further complicating resource management.

At first glance, it appears that resource managers could implicitly address the distributed nature by either considering application-wide dependencies [17, 28, 43, 44, 63] or employing heuristics with operator-defined rules on individual services [5]. The former centralizes resource control with a

[1] In this paper, we use "services" and "microservices" interchangeably.

global view of the service topology, while the latter delegates control to each service that acts on locally observed resource usage. Nevertheless, maintaining a global view is susceptible to topology changes and evolution [50, 64], and relying solely on local speculations may not achieve global optimality.

Instead, we embrace the distinct levels of distributed system behavior, and architecturally decouple mechanisms of application-level SLO feedback and service-level resource control. We design *Autothrottle*, a bi-level learning-assisted resource management framework for SLO-targeted microservices. The goal is to better use the visibility into application performance and SLO, to assist services in autonomously adjusting their own resource allocations. Autothrottle conveys this bridging "assistance" through *performance targets*, which translate the desired application performance to local proxy metrics measurable by services. Doing so hides low-level resource control details from the SLO feedback mechanism. In this paper, we use CPUs to discuss the framework design—not only is the CPU harder to manage due to its higher usage fluctuation over time [13, 22, 40], but it also has an immense impact on microservice response time [38, 57, 63].

At each microservice, Autothrottle locally runs a lightweight resource controller called *Captain*. Captain swiftly adjusts CPU allocations through OS APIs (e.g., CPU quota in Linux's cgroups), to ensure its governed microservice reaches the given performance target. Autothrottle represents this target using an unconventional metric—*CPU throttles*, namely the number of times a service exhausts its CPU quota in a time period. Not only are CPU throttles sufficiently cheap to sample at high frequency to enable Captains' timely adjustments, but we also observe that they have higher correlation with latencies than other proxy metrics such as CPU utilization (§5.3). These characteristics make CPU throttles an indicative target to track locally, for maintaining an end-to-end SLO. At the application level, Autothrottle employs a centralized SLO feedback controller called *Tower*. It observes the application workload measured by RPS (requests per second) and learns to determine the most cost-effective performance targets that maintain the SLO, using a lightweight online algorithm known as contextual bandits [14].

This paper makes the following key contributions:

- We examine unique implications that SLO-targeted microservices introduce to resource management (§2). Since there is no strong correlation between the end-to-end application performance and per-service resource usage, directly computing the optimal resource allocations is non-trivial.
- Autothrottle is a bi-level learning-assisted framework (§3), to embrace distinct levels of distributed system behavior. It separately designs mechanisms of application-level SLO feedback and service-level resource control, and introduces CPU-throttle-based performance targets to bridge them.
- Comprehensive experiments (§5) demonstrate Autothrottle's superior CPU savings over state-of-the-art heuristics and ML-based baselines, in three SLO-targeted applications: Train-Ticket [52], Social-Network [63], and Hotel-Reservation [22]. Compared with the best-performing baseline in each application, Autothrottle maintains the SLO for the 99th percentile latency while saving up to 26.21% CPU cores for Train-Ticket, up to 25.93% for Social-Network, and up to 7.34% for Hotel-Reservation, across four real-world workload patterns. Finally, running Social-Network over a 21-day period with production workloads from a global cloud provider, Autothrottle saves up to 35.2 CPU cores while reducing hourly SLO violations by 13.2×.

2 Background and Motivation

Our goal of SLO-targeted resource management is to minimize the total CPU allocations to microservice-based applications, while avoiding SLO violations on the user request latency. Following real-world findings [19], our SLO is an upper limit on tail latencies, specifically the 99th percentile (P99) request latencies unless otherwise noted.

2.1 Implications of microservices

Unlike monolithic applications, the distributed nature of microservices implies that multiple services collectively contribute to the end-to-end latency. This section presents observations, to motivate its implications on computing per-service resource allocations from the end-to-end latency SLO.

2.1.1 Service execution dependencies

As user requests traverse services, their end-to-end latency is a function of per-service performance (and hence resource usage). Being functionally different, services can consume resources differently. Moreover, service execution dependencies can introduce complex correlations to this function—not only are there various patterns such as parallelism, but services can also exhibit unexpected increases in resource demand.

An illustrative example is backpressure [22]—as an underprovisioned service undergoes performance degradation during request processing, the resource manager can misinterpret its idling parent's longer response time as the culprit. Simply identifying all parent-child relations does not fully solve the problem, as backpressure can vary subtly depending on service implementations. In one case we encountered, the CPU usage of a waiting parent unexpectedly increased with the number of requests, which was counterintuitive as waiting for child services should result in idle CPUs. Further investigations revealed that the parent service spawned a separate thread for each outstanding request (i.e., Thrift's `TThreadedServer` RPC model), leading to excessive thread maintenance and spurious context switching. An alternative implementation with non-blocking or asynchronous I/O (e.g., Thrift's `TNonBlockingServer`) eliminated the problem.

To grapple with the complexities arising from service dependencies, prior work considers how the end-to-end performance *directly* correlates with application-wide execution dependencies. However, maintaining an accurate and up-to-date global view of these dependencies is both challenging and costly. First, the interdependencies among services are constantly evolving during development, often with multiple versions of the same service coexisting [50], requiring frequent updates to the global view. Second, although ML-based resource management strategies [43, 63] have the potential to comprehend complex and large-scale service dependencies, they may entail substantial training and retraining expenses. Third, fine-grained distributed tracing (beyond the basic monitoring through sampling-based tracing) may be necessary for resource managers to observe and analyze service dependencies [23, 44], resulting in additional system overheads from increased instrumentation and telemetry collection.

Observation #1: *Maintaining an up-to-date global view of service dependencies can be impractical.*

2.1.2 Delayed end-to-end performance feedback

The chain of service execution dependencies brings about the *delayed effect*, i.e., a time delay for the impact of any changes in resource allocations or workloads to be fully observed in the end-to-end performance. This prevents resource managers from immediately responding to misallocations. The delayed effect is often amplified. One source is service queues—underprovisioning of resources will cause requests to accumulate in queues, and thus SLO violations are not detected until all queued requests are eventually processed or timed out. Even if resources are scaled at this point, it takes time to flush queues [22, 63]. Another amplification is that SLO is typically defined on aggregated performance data (e.g., percentiles), which require a sufficient number of requests to be profiled.

In light of the delayed effect, prior work [63] proposes to proactively predict the long-term impact of resource changes on the end-to-end performance. Such performance predictions are theoretically possible but they usually involve expensive data collection and model training. On the other hand, prematurely deploying ML models can result in a high percentage of mispredictions. For instance, our efforts to fully train Sinan's neural networks [63] for a 28-microservice application took 14+ hours, plus ∼6 hours to collect 20,000 training data points. Despite reproducing the published prediction accuracy, we observed that mispredictions can trick resource managers to overallocate at least 40.75% more CPU cores (§5).

Observation #2: *Predicting end-to-end application performance under the delayed effect can be unreliable.*

2.2 A practical approach

In light of the observations in §2.1, a more promising approach for SLO-targeted resource managers is to embrace the distributed nature of microservices by taking into account of the distinct levels of system behavior—the macro perspective reveals the end-to-end performance (e.g., user request latencies) and SLO, whereas the micro perspective is scoped to local measurements (e.g., service CPU usage) and control.

Naturally, these two levels can map to: ***(1)*** *application-level SLO feedback*, which compares the end-to-end performance and SLOs, and ***(2)*** *service-level resource control*, which computes resource allocations based on local measurements. In fact, if we architecturally decouple these mechanisms, it becomes feasible to position them close to their required inputs. Doing so brings the benefit of fast reaction, which opens up opportunities for resource managers to relax the requirement of computing the optimal resource allocations. Rather than striving to accurately model service dependencies (§2.1.1) or predict long-term application-wide behavior (§2.1.2), we can now employ lightweight service-level controllers that autonomously and swiftly adjust resource allocations, assisted by periodic guidance computed at the application level through a lightweight online learning approach.

In summary, SLO-targeted resource managers for microservices should incorporate the following design principles.

1. Decouple mechanisms of application-level SLO feedback and service-level resource control.
2. Rapidly drive per-service resource control with local performance targets and near-term prospects.
3. Achieve practicality through lightweight solutions.

3 The Autothrottle Framework

Following the design principles laid out in §2.2, we present Autothrottle, a practical and readily deployable resource management framework for SLO-targeted microservices.

3.1 Overview

Autothrottle is a bi-level learning-assisted framework, consisting of an application-wide global controller and per-service local controllers. The application-level controller is based on online learning, periodically assisting local resource control with its visibility into application workloads, end-to-end latencies, and SLO violations. The service-level controllers, on the other hand, are heuristic-based, continuously performing fast and fine-grained CPU scaling using local metrics as well as the assistance from the global controller.

The "assistance" bridging the two levels is based on the notion of *performance target*, a target performance level set by the application-wide controller for per-service controllers to attain. Autothrottle implements the performance target with *CPU throttle ratio*—the fraction of time a microservice is stopped by the underlying CPU scheduler. This design is motivated by the strong correlation between CPU throttles and service latencies revealed by our correlation test (§5.3).

Maintaining a CPU throttle ratio locally also allows tolerating a certain range of workload fluctuations (§5.3). When per-service controllers fail to rein in end-to-end latencies (e.g., the workload exceeds the tolerable range), the application-wide controller issues lower throttle targets to guide local controllers to allocate more CPUs. Conversely, higher throttle targets are assigned in the event of CPU over-provisioning.

Acting as a broker, the performance target allows Autothrottle to decouple the mechanisms of application-level SLO feedback and service-level resource control. Consequently, we are able to simplify the learning process of the application-level controller by concealing low-level resource details and avoiding the overhead of aggregating them, while enabling per-service controllers to focus on a self-contained in-situ task—reaching a given performance target using locally available information. Our bi-level design sets us apart from approaches that directly infer resource demands with proxy metrics (e.g., [5]) or machine learning (e.g., [63]).

Figure 2 depicts the architecture of Autothrottle. We refer to the per-service controllers as *Captains* (§3.2), and the application-wide controller as the *Tower* (§3.3).[2]

Autothrottle Captains. At the local level, each microservice runs a Captain instance, which periodically receives performance targets—CPU throttle ratios—from the Tower and strives to realize these targets using heuristic control. The heuristic control algorithm collects statistics on CPU usage and throttles, and employs two feedback control loops to scale CPUs up and down, respectively. This lightweight design ensures swift and fine-grained CPU autoscaling of the Captain even amid rapidly fluctuating workloads.

Autothrottle Tower. At the global level, the Tower leverages *contextual bandits* [14], a lightweight class of online reinforcement learning (RL), to dynamically determine suitable performance targets that maintain the SLO. It monitors application workload (e.g., RPS) and observes CPU allocations and end-to-end latencies (along with associated SLO violations) as feedback for its output targets. This online learning approach is directly applicable to any microservices, eliminating the need for extensive offline profiling or training.

Overall, Autothrottle takes a pragmatic stance and provides a resource management framework that is readily deployable across diverse latency-sensitive microservice applications. Next, we elaborate on Autothrottle from the bottom up, starting off with Captains (§3.2), followed by the Tower (§3.3).

3.2 Per-service controllers—*Captains*

Each Captain periodically (e.g., every minute) receives a target CPU throttle ratio from the Tower. Given a throttle target, Captain focuses on a self-contained, in-situ task—scaling up and down the CPUs made available to its governed service

[2]The air traffic control "tower" (application-wide controller) assigns "routes" (performance targets) to flights, while each "captain" (per-service controller) follows the assigned route by actually steering the aircraft.

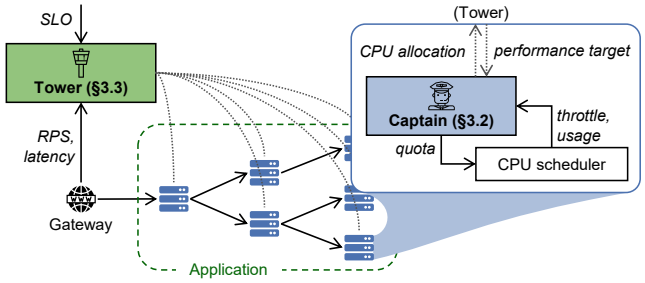


Figure 2: Autothrottle features bi-level resource management: The application-level learning-based controller (Tower), observing end-to-end latencies and workloads, periodically sets performance targets, expressed as CPU throttle ratios, for per-service heuristic controllers (Captains) to meet.

in order to meet the throttle target upon changing demand. Algorithms 1 and 2 present the pseudocode of Captain's main components, which we describe in detail below.

3.2.1 Resource metrics and knobs

Common CPU schedulers in the OS, such as the Linux CFS scheduler we use as a running example in this paper, assign each microservice a CPU quota (e.g., `cpu.cfs_quota_us`) to limit and isolate resource usage. To accomplish the task of maintaining a target CPU throttle ratio, Captain continuously collects two statistics exposed by the OS in each time window—CPU throttle count and CPU usage.

CPU throttle count. The Linux CFS scheduler maintains a CPU throttle count for each microservice in the variable `cpu.stat.nr_throttled`, which represents the cumulative number of CFS periods (100 ms by default) during which the CPU quota has been exhausted. Intuitively, if the CPU quota is used up early in a CFS period before a request can be fulfilled, the request will be approximately delayed by the remaining period, underscoring the importance of avoiding CPU throttles when maintaining latency SLOs. Anecdotal evidence in blog posts [15, 34] corroborates our intuition. To calculate the CPU throttle ratio over a time window, we divide the increase in the CPU throttle count (`cpu.stat.nr_throttled`) by the number of elapsed CFS periods.

CPU usage. The Linux CFS scheduler also reports the total CPU time consumed by a microservice as `cpuacct.usage`. This metric is particularly useful when the CPU is *over-provisioned*, as it reveals the actual (lower) CPU demand. Otherwise, this actual demand would be capped by the allocated CPUs if under-provisioned and thus remain unknown.

3.2.2 Multiplicative scale-up

In every time window of N ($N = 10$ by default) CFS periods, each Captain compares the measured CPU throttle ratio at its microservice with the target ratio. When the measured ratio exceeds the target, it indicates the CPU is under-provisioned,

Algorithm 1: Captain: scaling up and down

```
1  /* executes every N periods */
2  throttleCount = throttle count during last N periods;
3  throttleRatio = throttleCount/N;
4  margin = max(0, margin + throttleRatio − throttleTarget);
5  if throttleRatio > α × throttleTarget then
6  |  /* multiplicatively scale up */
7  |  quota = quota × (1 + throttleRatio − α × throttleTarget);
8  else
9  |  /* instantaneously scale down */
10 |  history = CPU usage history in the last M periods;
11 |  proposed = max(history) + margin × stdev(history);
12 |  if proposed ≤ β_max × quota then
13 |  |  quota = max(β_min × quota, proposed);
14 |  end
15 end
```

Algorithm 2: Captain: rollback mechanism

```
1  /* executes every period for N periods after each scale-down */
2  lastQuota = CPU quota before scale-down;
3  throttleCount = throttle count since scale-down;
4  throttleRatio = throttleCount/N;
5  if throttleRatio > α × throttleTarget then
6  |  /* revert to the previous (higher) quota before scale-down
7  |     with an additional allocation equal to the quota difference */
8  |  quota = lastQuota + (lastQuota − quota);
9  |  margin = margin + throttleRatio − throttleTarget;
10 end
```

demanding a prompt increase in the CPU quota to prevent imminent SLO violations at the application level.

To ensure that any desired target can be reached quickly within several steps, Captain increases the current CPU quota *multiplicatively*. We further make the size of the increase proportional to the difference between the measured CPU throttle ratio and the target ratio. This represents a form of proportional control, where a larger difference results in a larger stride in the CPU quota increase. The rationale is that when the difference is significant, a queue of requests is likely to have built up, thus requiring more CPUs to drain.

In practice, we find that the local workload arriving at a microservice is naturally bursty and irregular—regardless of the pattern of end-to-end requests—tricking Captains into spurious scale-ups. Hence, we execute the scale-up only when the CPU throttle ratio surpasses "$\alpha \times$ target ratio" ($\alpha \geq 1$), where α is a customizable weight that controls the sensitivity to transient load spikes. Correspondingly, the CPU quota is also multiplied by "1 + throttle ratio − $\alpha \times$ target ratio" in each step. The pseudocode is in Line 5–7 of Algorithm 1.

3.2.3 Instantaneous scale-down

Under frequent CPU throttling, Captain is forced to incrementally probe the actual CPU demand of the service. In contrast, when the measured throttle ratio is below the target ratio, the service's CPU demand has been adequately met. Consequently, historical CPU usage begins to more accurately reflect the actual (less throttled) CPU demand and help *instantaneously* determine the desired CPU quota.

Motivated by this characteristic of over-provisioning, the Captain maintains a sliding window of CPU usage over the most recent M ($M = 50$ by default) CFS periods, and calculates a new CPU quota based on two statistics from the sliding window: the maximum and the standard deviation of CPU usage. Specifically, the proposed quota is "max CPU usage + *margin* $\times$ standard deviation of CPU usage," where *margin* ≥ 0 is a dynamically tuned parameter that generally increases when the CPU throttle ratio exceeds the target ratio and decreases otherwise. To avoid unnecessary fluctuations in CPU allocation, the proposed quota is put into action only when it represents a significant-yet-moderate change. The details are described in Line 9–14 of Algorithm 1.

Our scale-down design draws inspiration from prior work [45,46], but differs in the carefully maintained parameter *margin* that depends on CPU throttles. Intuitively, if the CPU is recently throttled more often than desired, we should be more conservative by using a larger *margin* in the subsequent scale-down to avoid overreacting to momentary tranquility amid workload spikes; and vice versa. In summary, historical CPU usage in the sliding window allows for instantaneous scale-down, reclaiming extra CPU allocations in a single step.

3.2.4 Rollback mechanism after scaling down

Accidentally scaling up CPUs only leads to resource waste (and existing cloud applications tend to be over-provisioned); however, mistakenly scaling down the CPU allocation to any microservice may cause SLO violations at the application level. Thus, we introduce a fast rollback mechanism to the Captain to revert "reckless" scale-downs as follows.

After each scale-down, we continuously check whether it is "reckless"—if it has caused the CPU throttle ratio to exceed $\alpha \times$ target ratio—during *every* CFS period within the next N periods. We note that the triggering condition is the same as that used for scaling up, but due to the urgency of initiating a rollback, this check is performed more frequently, without waiting for the Captain's regular decision-making interval (N periods). After a rollback is triggered, the current CPU quota is restored to the previous (higher) quota used before the scale-down, plus an additional allocation equal to the difference between the two quotas. We grant slightly more CPUs to account for the potential processing delays that may have occurred since the erroneous scale-down. Details of the rollback mechanism are presented in Algorithm 2.

3.3 Application-level controller—*Tower*

In Autothrottle, Tower delegates the in-situ resource control to per-service Captains and only provides periodic assistance by dispatching the target CPU throttle ratios for Captains to

meet. This indirection via throttle targets effectively avoids the latency overhead associated with distributed tracing and logging, while retaining Tower's global perspective on end-to-end requests and SLO feedback.

We design Tower to compute a new target throttle ratio infrequently, e.g., once a minute, leaving ample time for tail request latencies and average CPU usage to stabilize as the new target settles in. Importantly, doing so minimizes the influence of Tower's previous decisions, simplifying the problem into a "one-step" decision-making process: Tower only needs to determine the optimal CPU throttle targets for the current step, without considering their long-term consequences.

This "one-step" nature motivates us to employ *contextual bandits*, a lightweight class of online reinforcement learning (RL) algorithms. In the taxonomy of RL, contextual bandits can be viewed as one-step RL and are well suited for real-time online scenarios in which the algorithm is required to learn efficiently from a limited amount of sample data.

3.3.1 Primer on contextual bandits

Recent work has modeled resource management with sequential decision-making paradigms and seen the application of multi-armed bandits [46, 47] and reinforcement learning [41, 44, 59]. Contextual bandits are intermediate between multi-armed bandits and the full-fledged RL [54].

Contextual bandits are like multi-armed bandits in that they are well suited to problems where an *action* (e.g., CPU throttle targets) taken at a step (e.g., one-minute interval) does not have long-term impact beyond that step. They receive a *cost* (negative reward) as feedback for the chosen action, and aim to minimize the cumulative cost (e.g., comprising CPU allocations and SLO violations). Conversely, contextual bandits also differ from multi-armed bandits by their ability to make decisions based on the observation of the system state, known as the *context* (e.g., RPS). This context can provide valuable information that aids in the learning process (§5.3).

In contrast to the full RL, which optimizes a sequence of future steps, contextual bandits only optimize the current step owing to their assumption that each chosen action only affects the immediate outcome without long-term consequences. Moreover, full RL typically demands extensive offline training before deployment as well as frequent retraining (e.g., upon significant changes in microservices), whereas contextual bandits are more lightweight (with simpler models) and suitable for online learning with considerably fewer samples.

In solving contextual bandit problems, a common approach is to train a cost-prediction model that estimates the cost of taking each action within a context. Due to their inherent partial observability, however, contextual bandits can only observe the costs of actions they select but not the costs of others. To enhance their performance and sample efficiency, a widely adopted improvement is to estimate the costs of unused actions via counterfactual estimates [12, 20, 48]. This approach reduces contextual bandit problems to cost-sensitive classification [35], which can then be addressed using standard supervised learning. We adopt this approach and refer the reader to Bietti et al. [14] for more details.

3.3.2 Realizing contextual bandits in Tower

Next, we describe the contextual bandit algorithm used in Tower. The algorithm operates with a step size of one minute, and it aims to learn to output an action that incurs the lowest cost given the observed context at each step.

Context. Tower selects the average RPS observed in the last step as the context because the optimal CPU throttle target depends on the RPS (§5.3). We refrain from predicting the RPS for the next step due to the inherent difficulty in accurately forecasting RPS; moreover, our Captains have been intentionally designed to tolerate short-term RPS fluctuations (§5.3). Other metrics such as CPU usage are not included in the context as they are merely the byproducts of applying a throttle target to an RPS, with the RPS serving as the primary causal factor. The composition of the workload (i.e., the distribution of different request types) is relevant, but our focus in this work remains on constant workload composition (Appendix A), following the setup in prior work [22, 63].

Action. Given an instantiation of the Captain's resource control algorithm, we search for a ladder of CPU throttle targets as the actions. The search is a one-time process for all applications. By default, our action space consists of 9 throttle targets, ranging from 0 to 0.3 (§4).

Reduction of action space. A microservice-based application can contain 10–1000s of services [11, 25, 31, 40, 64]. In the case of 9 throttle targets, generating a different CPU throttle target for each individual service would result in $9^{\#\text{services}}$ actions, rendering it infeasible for contextual bandits to learn. As a solution, Tower clusters microservices into two classes and outputs an action for each class, effectively reducing the action space to $9^2 = 81$. To implement the clustering, we use the standard k-means algorithm [37] to group microservices based on their average CPU usage. Our empirical results in §5.3 suggest a diminishing return beyond two clusters.

Cost function. We define the cost received per step as follows. When the SLO is met after the step, we only use the total CPU allocation as the cost, since the actual latencies below SLO matter no more. To this end, Tower requests Captains to send their actual CPU allocations as feedback every minute, and then normalizes the total allocation linearly into $[0, 1]$. On the other hand, when the SLO is violated, we set the cost to only contain the tail latency, linearly normalized to $[2, 3]$ considering the higher priority of SLO violations. We arrived at the two normalization ranges above based on their empirical performance compared with other ranges we tested, but we do not claim our cost function is the best.

Noise reduction for costs. Our contextual bandit algorithm learns online and updates its model weights on every (context, action, cost) tuple, i.e., most recent RPS, two throttle targets,

and the incurred cost. In reality, however, we observe highly noisy costs that result in confusion and poor performance of the model, supposedly due to the complex microservice system and the dynamics in Captains. To address this, we buffer and group recent samples using (context, action) as the index after quantizing the RPS. Given a new sample, we use the median cost of the group it falls into—rather than the cost computed for that individual sample—to update the model. Doing so significantly reduces the noise in costs and stabilizes the online learning process.

Exploration. Similar to multi-armed bandits and RL, contextual bandits rely on exploration to acquire knowledge about the costs of different actions, e.g., using ε-greedy [35] to choose a random action with a small probability of ε (the best action is selected otherwise). Despite a reduced action space, randomly exploring all 81 actions within a context remains inefficient as each sample requires one minute to complete; repeated sampling is further required to calibrate noisy cost estimates. To ensure efficient exploration without impeding online learning, we explore only the neighbors of the best action in the action space. Given a sorted ladder of the available CPU throttle target , $r_1 < r_2 < \ldots < r_9$, if the best action consists of (r_i, r_j), $1 \leq i, j \leq 9$, then each of its neighbors (r_i, r_{j-1}), (r_i, r_{j+1}), (r_{i-1}, r_j), (r_{i+1}, r_j) is explored next with an equal probability of $\varepsilon/4$ (subject to boundary conditions). The rationale is that the throttle target ladder is monotonic, allowing Tower to move upward or downward one step at a time without missing the optimal action.

4 Implementation

Our current implementation supports microservice applications deployed as pods on Kubernetes, but it can be easily extended to other container orchestration frameworks (e.g., OpenShift and Docker Swarm). Autothrottle is open-sourced at https://github.com/microsoft/autothrottle.

Captain. Each microservice is associated with a Captain co-located on the same worker node, so we deploy Captains as processes on worker nodes of the Kubernetes cluster. Captain implements the following three functionalities. First, it communicates with the Tower over a TCP socket, exchanging CPU throttle targets and allocations. Second, it collects CPU throttling and usage statistics from Linux cgroup API in every CFS period of 100 ms, as the input to the local resource controller. Third, it runs the resource controller for all microservices on the same worker node, and sets their CPU quotas (`cpu.cfs_quota_us`) accordingly. As Captain only comprises lightweight heuristic-based control loops, it does not require any pre-deployment training.

The pseudocode of Captain is outlined in Algorithms 1 and 2. Our default parameters are $N = 10$, $M = 50$, $\alpha = 3$, $\beta_{\max} = 0.9$, $\beta_{\min} = 0.5$. They can be adjusted accordingly. A larger N or M lowers sensitivity to the noise in CPU usage, hence slower reaction. α sets the supported range of throttle ratios to $(0, 1/\alpha)$. A smaller α increases the upper bound but decreases the tolerance on throttle ratio fluctuations. $\beta_{\max}$ and $\beta_{\min}$ prevent overly small or large allocation changes.

Tower. One instance of Tower runs globally alongside the application (i.e., in the same cluster), initialized with a user-specified SLO. It collects average RPS and tail latencies from the Locust workload generator, but can be extended to hook up to an application gateway. Furthermore, Tower receives the actual CPU allocations from Captains after dispatching CPU throttle targets to them every minute.

Tower leverages the widely used Vowpal Wabbit (VW) library [10] to implement contextual bandits. For each group of microservices, the model outputs one of the 9 throttle targets: 0.00, 0.02, 0.04, 0.06, 0.10, 0.15, 0.20, 0.25, and 0.30. Designed for efficient online learning, VW offers lightweight model options such as linear regression or a shallow neural network with a single hidden layer. We opt for a neural network model with 3 hidden units after performing an ablation study (§5.3), and train it with a learning rate of 0.5. The doubly robust estimator [20] is employed in the bandits for policy evaluation to estimate the costs of untaken actions. Moreover, we disable the native ε-greedy algorithm to implement our customized exploration strategy (§3.3.2). The specific VW usage is detailed in Appendix B.

Online training starts with an exploration stage, which allows VW to randomly explore how different CPU throttle targets would impact application latencies. During this stage, each randomly chosen action will be executed for 2 minutes. Only the second minute is used for cost calculation and training, in order to avoid interference from the previous chosen action. This exploration stage lasts $\sim$6 hours, during which application latencies may exceed the SLO.

After the exploration stage, Tower starts to exploit the best action, while still exploring neighboring actions with a total of 10% probability using ε-greedy. Tower runs every minute to collect last minute's (context, action, cost) sample. All recent samples are grouped using (context, action) as the index with RPS quantized into bins of 20, and each group's cost is defined as the median cost of the group. Since training each unique (context, action) only once is insufficient for contextual bandits, 10,000 training data points are sampled from these groups randomly. A contextual bandit model is then trained on these samples, and predicts the next best action based on RPS. As a reference, this training-and-prediction process takes less than one second in our setup.

5 Evaluation

We evaluate Autothrottle's superior resource saving with three SLO-targeted microservice applications, against state-of-the-art heuristic- and ML-based baselines. Major results include:

(1) Over the best-performing baseline in each application, Autothrottle maintains the given application P99 latency SLO,

while achieving a CPU core saving up to 26.21% for Train-Ticket, up to 25.93% for Social-Network, and up to 7.34% for Hotel-Reservation. Over all baselines, its savings can be up to 93.84%, 55.32%, and 83.99%, respectively.

(2) A 21-day study of Social-Network (with real-world workload trace from a global cloud provider) shows a saving up to 35.2 CPU cores, over the best-performing baseline. Meanwhile, it reduces hourly SLO violations from 71 to 5.

(3) Microbenchmarks evaluate Autothrottle's design and tolerance to workload fluctuations and load-stressing.

5.1 Methodology

Benchmark applications. We deploy three SLO-targeted microservice applications: ***(1)*** Train-Ticket [52], with 68 distinct services, ***(2)*** Hotel-Reservation from DeathStarBench [22], with 17 distinct services, and ***(3)*** Social-Network used in Sinan [63], a variant of the Social-Network application from DeathStarBench, with 28 distinct services including two ML inference serving services: a CNN-based image classifier and an SVM-based text classifier. These applications are representative of real-world microservices, with stateless services (e.g., business logic), data services (e.g., key-value stores), and gateways. Deployments are managed by Docker and Kubernetes. Parent-child service communications are through popular RPC frameworks such as gRPC and Thrift.

Application SLOs are specified on the hourly P99 latency [19]—1,000 ms for Train-Ticket, 200 ms for Social-Network, and 100 ms for Hotel-Reservation.

Comparison baselines. Baselines include ***(1)*** Kubernetes default autoscalers [5] (denoted as "K8s-CPU" and "K8s-CPU-Fast"), and ***(2)*** state-of-the-art ML-driven solution, Sinan [63].

K8s-CPU locally maintains each service's average CPU utilization, with respect to the user-specified CPU utilization threshold (e.g., 50%). Every m=15 seconds, it measures service's CPU usage, and computes the optimal allocation by "CPU usage / CPU utilization threshold." Then, it sets the CPU limit to the largest allocation computed in the last s=300 seconds. We also include a faster version called K8s-CPU-Fast, which has m=1 and s=20. Since Kubernetes relies on users to properly translate the application SLO to CPU utilization threshold, we manually try different thresholds to find the appropriate one for each experiment (Appendix F).

Sinan leverages ML models (e.g., a convolutional neural network and a boosted tree model) to globally assess each service's resource allocation. Starting with the open-sourced Sinan [8], we follow instructions to train application-specific models offline for 20+ hours. Since Sinan relies on users to properly set several hyperparameters, we manually tune for each application. During experiments, we run Sinan every second—given historical resource usage and latencies, Sinan tries to predict the optimal CPU allocation that is unlikely to violate the SLO over both the short and long terms.

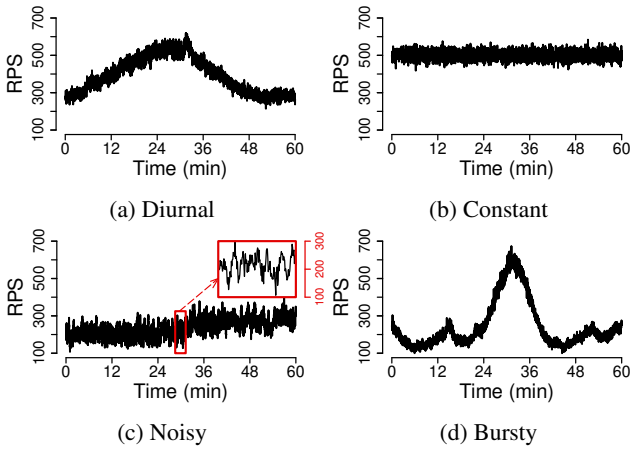


Figure 3: Our workload traces capture common patterns of RPS (requests per second) on an hourly basis. These patterns have been observed in real-world scenarios: Puffer streaming requests [60], Google cluster usage [58], and Twitter tweets [9]. We also recorded a full 21-day workload trace from a global cloud provider for long-term evaluation. We scale these traces accordingly for each benchmark application to saturate the cluster (Appendix E).

Experiment setup. We generate workloads with Locust [7], which is configured to mix application requests (Appendix A) to stress as many services as possible. Locust replays workload traces to reproduce RPS (requests per second). The first set of traces captures hourly RPS patterns, which are commonly observed in production environments: Puffer's streaming requests [60], Google's cluster usage [58], and Twitter tweets [9]. Figure 3 illustrates these patterns: diurnal, constant, noisy, and bursty. We also keep a full 21-day workload trace from a global cloud provider for long-term evaluation. Depending on the complexity of benchmark applications, we scale traces accordingly to saturate the cluster (Appendix E).

Each experiment ends when Locust finishes replaying a trace. For comparisons, we record the following per-hour measurements: ***(1)*** the average number of CPU cores allocated, and ***(2)*** the application end-to-end P99 latency.

Our testbeds consist of a 160-core cluster (over five 32-core Azure VMs with AMD EPYC 7763 processors) and a 512-core cluster (over six 64-core and four 32-core physical servers with Intel Xeon Silver 4216 processors).

5.2 Application SLO and resource saving

We evaluate the amount of CPU resources that Autothrottle saves over baselines, while every algorithm tries to maintain the hourly SLO over time. To ensure that all baselines can achieve their best results, we manually identify and tune their settings prior to experiments (Appendix F).

Table 1 summarizes empirical results on the 160-core cluster, and Autothrottle outperforms baselines in all applications.

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	30.4	58.0 (↓47.59%)	41.2 (↓26.21%)	278.4 (↓89.08%)
Constant	21.7	24.8 (↓12.50%)	27.3 (↓20.51%)	279.9 (↓92.25%)
Noisy	15.5	23.6 (↓34.32%)	17.7 (↓12.43%)	251.8 (↓93.84%)
Bursty	17.7	27.1 (↓34.69%)	21.9 (↓19.18%)	268.3 (↓93.40%)

(a) Train-Ticket application (SLO: 1,000 ms P99 latency)

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	77.5	93.9 (↓17.47%)	115.5 (↓32.90%)	162.7 (↓52.37%)
Constant	88.7	115.6 (↓23.27%)	118.8 (↓25.34%)	149.7 (↓40.75%)
Noisy	57.5	66.5 (↓13.53%)	105.1 (↓45.29%)	105.2 (↓45.34%)
Bursty	50.0	67.5 (↓25.93%)	99.7 (↓49.85%)	111.9 (↓55.32%)

(b) Social-Network application (SLO: 200 ms P99 latency)

Workload	Autothrottle	K8s-CPU	K8s-CPU-Fast	Sinan
Diurnal	15.3	15.7 (↓2.55%)	16.5 (↓7.27%)	45.5 (↓66.37%)
Constant	11.2	11.5 (↓2.61%)	11.3 (↓0.88%)	21.2 (↓47.17%)
Noisy	10.8	12.1 (↓10.74%)	11.6 (↓6.90%)	65.9 (↓83.61%)
Bursty	10.1	15.7 (↓35.67%)	10.9 (↓7.34%)	63.1 (↓83.99%)

(c) Hotel-Reservation application (SLO: 100 ms P99 latency)

Table 1: Average number of CPU cores that Autothrottle and baselines allocate to satisfy the SLO (and thus latencies are elided). Percentages in parentheses quantify Autothrottle's CPU savings over each baseline. The overall best-performing baseline for each application is highlighted in gray. For K8s-CPU and K8s-CPU-Fast, we manually search for their optimal utilization thresholds (to minimize the average CPU allocation), per application and workload trace (Appendix F).

We make the following observations, with respect to heuristic-based baselines. First, in Social-Network, Autothrottle saves up to 25.93% of CPU resources (or 17.5 cores) over K8s-CPU, and up to 49.85% of CPU resources (or 49.7 cores) over K8s-CPU-Fast. Delving into empirical results, Figure 4 suggests that tuning the baselines' CPU utilization thresholds does not make them outperform Autothrottle. Taking the diurnal workload as an example, the figure shows that Autothrottle is able to maintain the application SLO with the minimum CPU allocation—Autothrottle achieves a P99 latency of 178 ms with only 77.5 cores, whereas K8s-CPU achieves 177 ms with 115.5 cores and K8s-CPU-Fast achieves 171 ms with 93.9 cores, at best. When allocating a comparable number of CPUs (∼80 cores) to Autothrottle, K8s-CPU and K8s-CPU-Fast would violate the SLO, resulting in latencies of 252 ms and 418 ms respectively. Second, Autothrottle has a relatively low resource reduction on Hotel-Reservation. This is due to the application simplicity where requests traverse an average of only 3 microservices. A similar observation can be made for the constant workload trace, where the relatively static RPS pattern simplifies scaling decisions.

Furthermore, Table 1 shows that Autothrottle outperforms the ML-enabled baseline, Sinan. Its CPU saving is at least

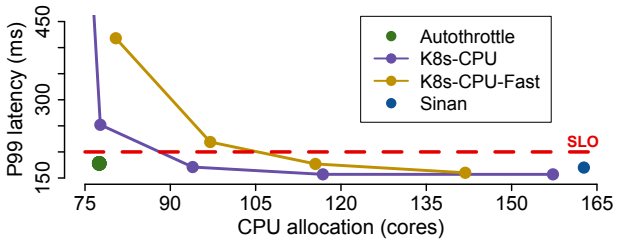


Figure 4: Application latency vs. CPU allocations, as we vary the two baselines' CPU utilization threshold for Social-Network under the diurnal workload trace. Dashed red line illustrates the 200 ms SLO. Autothrottle is able to maintain the SLO with the minimum CPU allocation.

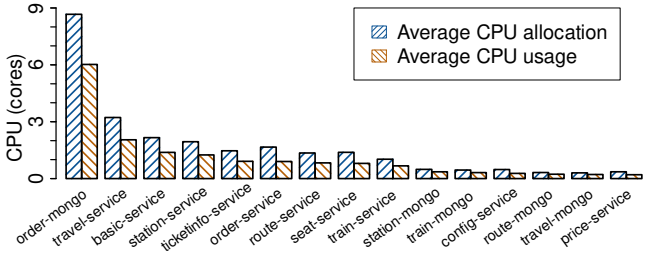


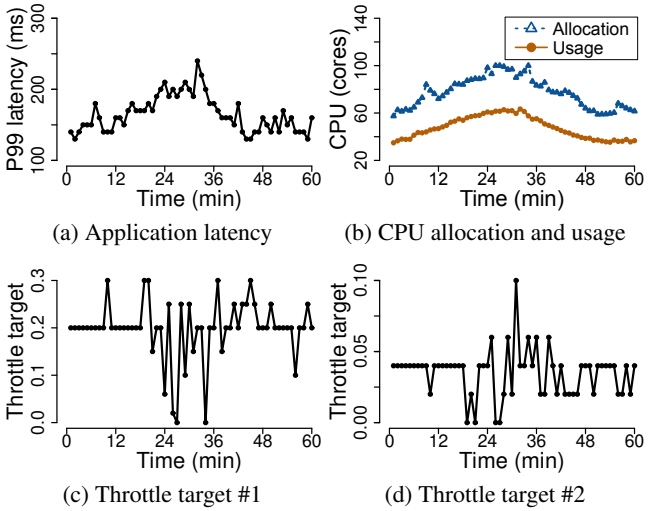
Figure 5: Autothrottle tailors CPU allocations to each microservice's resource usage. Figure shows top 15 microservices with the highest CPU usage in Train-Ticket under the diurnal workload trace.

40.75% (or 61 cores for Social-Network). Deeper investigations suggest two reasons for this gap. First, while we are able to achieve the model accuracy published by authors (e.g., training RMSE of 22.39 and validation RMSE of 22.07, for Social-Network) after 20+ hours of training, this non-negligible error can still mislead scaling decisions, especially for non-constant workloads. Second, in order to reduce training costs, Sinan learns to make relatively coarse-grained CPU allocation adjustments (i.e., ±1 core, ±10% cores, and ±50% cores).

Resource savings from Table 1 are due to Autothrottle's ability to tailor CPU allocations across services and over time. For example, Figure 5 looks at top 15 microservices with the highest CPU usage, under diurnal workload in Train-Ticket. We note that CPU allocation is noticeably lower for services with less CPU usage (e.g., `price-service`). Under the same workload, Figure 6 illustrates how Tower updates performance targets—as the RPS varies over time, Tower selects appropriate throttle targets to adjust CPU allocations and maintain the P99 latency. Note that per-minute P99 latencies are displayed in this figure, different from the hourly P99 latencies shown in the remaining evaluation.

5.3 Microbenchmarks

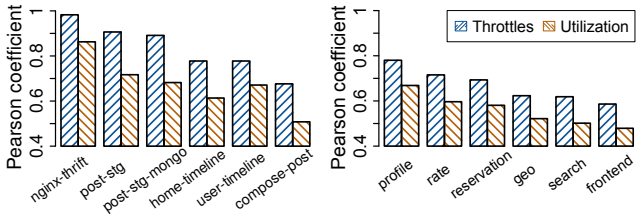
Correlation of proxy metrics to latencies. Compared with the prevalent proxy metric for estimating resource demand—



(a) Application latency (b) CPU allocation and usage

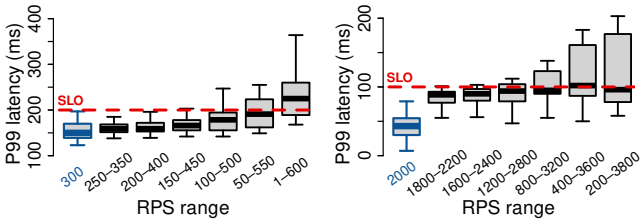
(c) Throttle target #1 (d) Throttle target #2

Figure 6: Measurements of Social-Network under diurnal workload. Figures (a) and (b) show the latency and CPU statistics achieved by Autothrottle. Figures (c) and (d) demonstrate how Tower adjusts throttle targets in response to time-varying workload for the two CPU usage groups (Appendix C).



(a) Social-Network (b) Hotel-Reservation

Figure 7: As a proxy metric, CPU throttles exhibit a higher correlation with application latencies than CPU utilization. The figure shows top microservices with highest CPU usage.



(a) Social-Network (b) Hotel-Reservation

Figure 8: Captain maintains latency SLO under some workload fluctuations. Boxplots show latency variances, from reusing the first blue boxplot's performance target.

CPU utilization, our use of CPU throttles is motivated by the higher correlation with application latencies as demonstrated by Figure 7. For each service in Social-Network, we manually set its CPU quota (i.e., `cpu.cfs_quota_us`) to 40 uniformly distributed values. Then, we measure CPU utilization, CPU throttle counts, and application P99 latency, at 300 RPS. We compute the Pearson correlation coefficient for ***(1)*** latency vs. CPU throttles, and ***(2)*** latency vs. CPU utilization. Figure 7a focuses on Social-Network microservices using the most CPU cores. In all cases, CPU throttles exhibit a higher correlation than CPU utilization, suggesting a stronger linear relationship. Figure 7b shows the same conclusion for Hotel-Reservation.

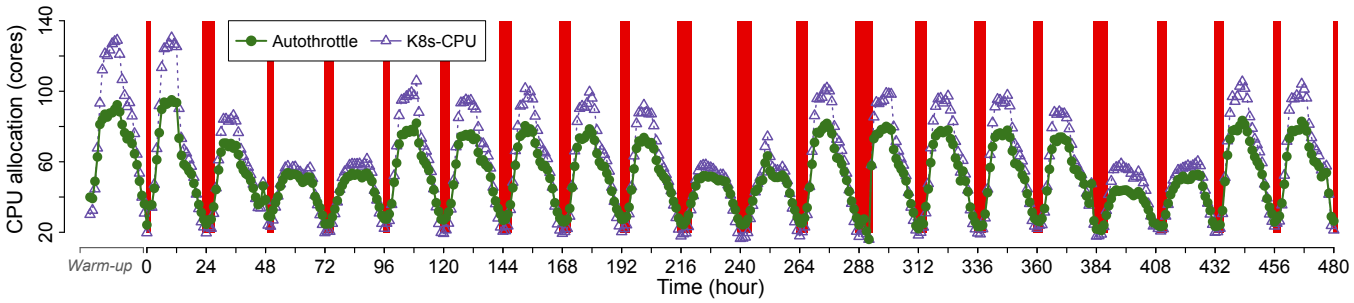
Recall that Captains continuously collect local CPU throttles for resource control (§3.2.2), and Tower distributes CPU-throttle-based performance targets (§3.3.2). A high correlation suggests that CPU throttling is indicative of the latency and suitable to track locally in Captain as a target for maintaining the SLO. The learning process in Tower can also be simplified given a clear relationship between CPU throttles and application latencies.

Tolerance to short-term workload fluctuations. Figure 8 shows that Captains can tolerate short-term local workload fluctuations, even with static throttle targets. The experiment starts by finding a throttle target for Social-Network's 200 ms SLO, at 300 RPS. Then, we reuse this target while instrumenting Locust to fluctuate RPS in a one-minute window for 60 minutes. The fluctuation ranges from 100 (i.e., RPS=250–350) to 600 (i.e., RPS=1–600). In Figure 8a, boxplots summarize the latency variance of 60 windows. Autothrottle can keep the application P99 latency under SLO for a fluctuation range up to 300 (i.e., RPS=150–450), or up to 500 (i.e., RPS=50–550) if we consider the median value instead. Similarly, Figure 8b shows RPS fluctuation tolerance up to 800 (i.e., RPS=1,600–2,400) for Hotel-Reservation.

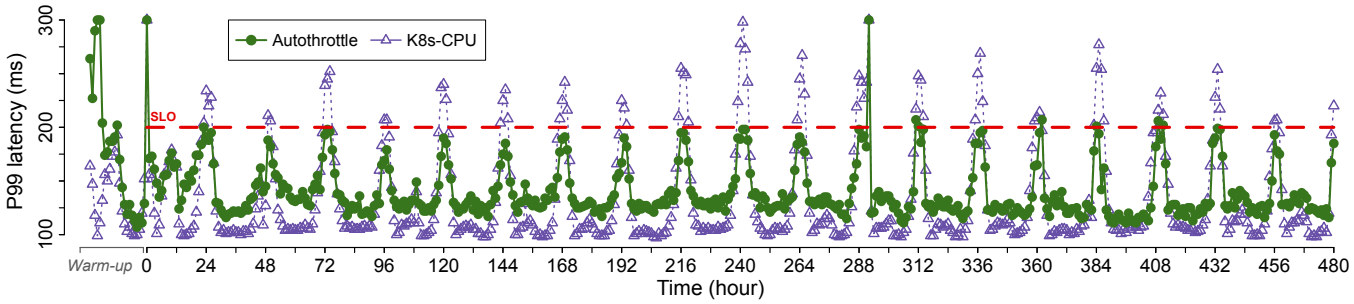
The tolerance to short-term workload fluctuations stems from the use of performance targets (vs. exact resource allocations), which hide service-level resource details from Tower and enable Captains to autonomously adjust resource allocations. This tolerance is vital as it frees Tower from the excessive recomputation of performance targets (§3.3.2).

Number of performance targets. Rather than generating separate performance targets for individual microservices, Tower clusters microservices into two categories based on their average CPU usage, reducing the action space to two targets (§3.3.2). To assess this design, we empirically compare the performance of 1, 2, 3, and 4 targets, under the constant workload trace. In each scenario, we manually search for the best-performing set of throttle targets that satisfy the SLO using the minimum number of CPU cores. For Social-Network, Autothrottle allocates 70.8, 55.9, 55.1, and 54.7 cores with 1 to 4 targets, respectively. Hotel-Reservation consistently uses the largest target (0.3) to meet the SLO on this trace, regardless of the number of targets. For Train-Ticket, the allocation is 18.6, 18.1, and 18.1 cores with 1 to 3 targets (exhaustive search is infeasible for 4 targets). Overall, these results suggest a diminishing return beyond 2 targets.

Load-stressing to the limit. We stress resource managers, by pushing Locust's RPS to the application's upper limit. This is the breaking point (before application crashing) when almost



(a) CPUs allocated by Autothrottle and the K8s-CPU baseline. Red boxes highlight hours of K8s-CPU's SLO violations.



(b) Social-Network's P99 latency, as achieved by Autothrottle and the K8s-CPU baseline. Dashed red line illustrates the 200 ms SLO.

Figure 9: A 21-day study on Social-Network with real-world workload trace from a global cloud provider. Compared with Autothrottle, the K8s-CPU baseline over-allocates an average of 12.1 and up to 35.2 cores, and triggers 71 hourly SLO violations.

all CPU cores are allocated. To this end, we stress Social-Network at constant RPS of 600 and 700, on the 160-core cluster. At 600 RPS, Autothrottle still achieves a CPU core saving of 27.67% and better SLO—it achieves a P99 latency of 202 ms with only 98.3 cores, whereas K8s-CPU achieves 216 ms with 135.9 cores and K8s-CPU-Fast achieves 235 ms with 133.1 cores. Finally, at 700 RPS, Autothrottle achieves a P99 latency of 452 ms with only 106.8 cores, whereas K8s-CPU achieves 600 ms with 153.1 cores, and K8s-CPU-Fast achieves 551 ms with 143.8 cores.

Ablation study for contextual bandits. We investigate two aspects that can impact Tower's contextual bandits. The first is the number of available throttle targets to choose from in the action space (§3.3.2). For the constant workload trace, reducing from 9 to 4 throttle targets results in over-allocating 5.6 CPU cores (or 10.03%) for Social-Network, and 0.7 CPU cores (or 3.49%) for Train-Ticket. The second is the use of neural networks (§4). Under various workload patterns on Social-Network, we test a linear model and neural networks with different numbers of hidden units, but their difference in CPU allocation is small. None of the tested models violates the SLO. We include the results in Appendix B.

5.4 Long-term evaluation

We perform a 21-day study with real-world workload trace from a global cloud provider. Experiments are performed with Social-Network on the 160-core cluster, and an hourly SLO of 200 ms is set on P99 latency. We compare Autothrottle with K8s-CPU, the best-performing baseline from §5.2. We use day 1 for training and tuning Autothrottle and K8s-CPU. For the former, we train the Tower's model. For the latter, we spend 24 man-hours to manually identify its best CPU utilization threshold.

Figure 9 illustrates the results over the entire period. Figure 9a shows the CPU core saving that Autothrottle achieves every hour, over the K8s-CPU baseline. First, Autothrottle can save up to 35.2 cores (or an average saving of 12.1 cores) over K8s-CPU. Second, although there are days when K8s-CPU allocates fewer CPUs (e.g., an hourly average of −2.77 CPU cores on day 4), these are also the days when K8s-CPU has a high chance of triggering SLO violations. In total, K8s-CPU violates the hourly SLO 71 times (highlighted by red boxes in Figure 9a). On the other hand, Autothrottle reduces SLO violations to only 5 times—an investigation reveals that these hours' workloads appear anomalous (i.e., recorded RPS jumps between 0 and ∼400) and unforeseen.

Figure 9b shows Social-Network's P99 latency per hour. One observation is that Autothrottle is able to continuously maintain a P99 latency closer to the 200 ms SLO. Since its P99 latency exhibits a much lower variance over time, this results in a more stable application performance.

5.5 Large-scale evaluation

We now show Autothrottle's scalability on the larger 512-core cluster. It allows us to push RPS beyond the breaking point of the 160-core cluster (§5.3), up to 1,200 on Social-Network (the upper limit for comparison baselines). To fully allocate

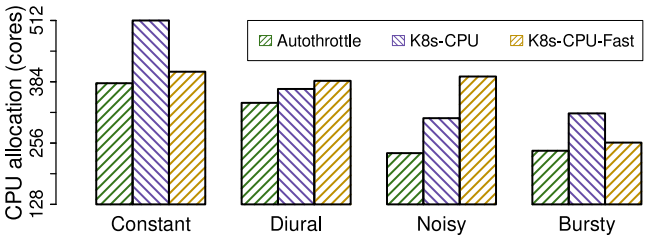


Figure 10: Number of CPU cores that Autothrottle and baselines allocate, to satisfy Social-Network's P99 SLO. Figure shows Autothrottle's scalability on the larger 512-core cluster.

all cores, we replicate Social-Network's CPU-intensive microservices: Nginx (×3) and ML-based image classifier (×6).

Figure 10 shows that Autothrottle is able to allocate fewer CPU cores while meeting Social-Network's 200 ms P99 SLO. Compared to the best-performing baselines, K8s-CPU and K8s-CPU-fast, Autothrottle saves up to 28.24% (or 150 CPU cores) and at least 5.92% (or 24 CPU cores). Finally, we note that K8s-CPU-Fast can have a higher CPU allocation than K8s-CPU, especially for the noisy workload trace. Since K8s-CPU-Fast is more sensitive to CPU utilization changes than K8s-CPU, it can sometimes accidentally scale down and lead to SLO violations. As a result, conservatively setting K8s-CPU-Fast results in the trade-off of higher CPU allocation.

6 Related Work

Cloud resource management. Cloud vendors have long offered services that enable elastic scaling of VMs and their associated resources according to user-defined rules [1, 3, 4]. In addition to rule-based scaling, researchers have proposed predictive scaling, which involves forecasting future demand and adjusting resource allocation in advance of any demand changes [2, 26, 42, 51]. Despite the cost effectiveness of these mechanisms in meeting SLOs, they are primarily designed for VMs (e.g., targeting monolithic applications or relying on VM-specific techniques such as live migration), and cannot be directly applied to microservices. Other cluster management frameworks [18, 27, 49, 55, 56] that schedule jobs to clusters may be used in conjunction with Autothrottle.

Vertical scaling of microservices. Vertical autoscalers adjust the resource limits in a fine-grained manner, e.g., milli-cores. Kubernetes Vertical Pod Autoscaler (VPA) [6] heuristically adjusts resource limits to maintain a user-specified utilization threshold. Autopilot [46] focuses on vertical scaling, selecting resource limits based on moving windows of historical usage and an ML technique akin to multi-armed bandit. Sinan [63] trains ML models to infer the likelihood of SLO violations given a set of proposed CPU limits. FIRM [44] reacts to SLO violations and pinpoints a microservice as the root cause, using reinforcement learning to scale up the service. A recent work [36] (also named "Autothrottle") adjusts the CPU quota of containers using closed-loop control to satisfy their individual network SLOs (e.g., throughput), rather than application latency SLOs. Autothrottle differs from these approaches with its bi-level design and the use of CPU throttle targets.

Horizontal and hybrid scaling of microservices. Horizontal autoscalers operate at a coarse-grained level by adjusting the number of replicas of a microservice. Kubernetes Horizontal Pod Autoscaling (HPA) [5] employs a mechanism similar to VPA at its core, except for choosing the appropriate number of pods to meet an input utilization threshold. GRAF [43] leverages graph neural networks to model service dependencies. COLA [47] uses a multi-armed bandit to collectively determine the number of replicas for each microservice. In addition, there are hybrid autoscalers that combine vertical and horizontal scaling and apply them selectively [24, 33]. Autothrottle focuses on vertical scaling due to its fine-grained and rapid reaction that empowers per-service controllers. As future work, we plan to explore the integration of horizontal scaling with Autothrottle.

Proxy metrics for estimating resource demand. In comparison to CPU throttles, alternative service-level proxy metrics fall short in maintaining end-to-end latency under workload changes. Kubernetes defaults to CPU utilization [5], but high CPU usage does not always indicate an issue if requests can still complete within the SLO [34, 47]. Queue length [61], the number of requests pending at a service, overlooks the complexity of individual requests, while queuing delay [16, 64] depends on the service's threading model [53] and may require manual instrumentation of each service. The scattered nature of queues across the application, OS, and network, further complicates precise measurement of queue length or queuing delay [63]. Regardless, dynamically adapting the thresholds for these metrics may require an application-level controller as proposed by Autothrottle.

7 Conclusion

Autothrottle is a bi-level learning-assisted resource management framework for SLO-targeted microservices. It decouples mechanisms of SLO feedback and resource control, and bridges them through CPU-throttle-based performance targets. Going forward, we are extending to additional resource types such as memory and storage, and exploring integrations with additional scaling strategies such as horizontal scaling.

Acknowledgments

We are grateful to Yang Yue and Jiayi Mao for their contributions in the early stage of this work. We thank Neeraja J. Yadwadkar for shepherding our paper and the anonymous reviewers for their helpful comments and feedback. We also thank Haidong Wang, Chuanjie Liu, Qianxi Zhang, Yawen Wang, Yu Gan, Fan Yang, Mao Yang, Lidong Zhou, and Victor Bahl for their support or insightful discussions.

References

[1] AWS Auto Scaling. https://aws.amazon.com/autoscaling/.

[2] AWS Predictive Scaling. https://docs.aws.amazon.com/autoscaling/ec2/userguide/ec2-auto-scaling-predictive-scaling.html.

[3] Azure Autoscale. https://azure.microsoft.com/en-us/products/virtual-machines/autoscale/.

[4] Google Cloud Autoscaler. https://cloud.google.com/compute/docs/autoscaler/.

[5] Kubernetes Autoscaling. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.

[6] Kubernetes Vertical Pod Autoscaler. https://github.com/kubernetes/autoscaler/tree/master/vertical-pod-autoscaler#vertical-pod-autoscaler.

[7] Locust: An Open Source Load Testing Tool. https://locust.io.

[8] Sinan Open-sourced Repository. https://github.com/zyqCSL/sinan-local.

[9] Twitter Data for Academic Research. https://developer.twitter.com/en/use-cases/do-research/academic-research/resources. Accessed in 2022.

[10] Vowpal Wabbit. https://vowpalwabbit.org.

[11] Adam Gluck. Introducing Domain-Oriented Microservice Architecture, 2020.

[12] Alekh Agarwal, Daniel Hsu, Satyen Kale, John Langford, Lihong Li, and Robert Schapire. Taming the monster: A fast and simple algorithm for contextual bandits. In *International Conference on Machine Learning*, pages 1638–1646. PMLR, 2014.

[13] Pradeep Ambati, Inigo Goiri, Felipe Frujeri, Alper Gun, Ke Wang, Brian Dolan, Brian Corell, Sekhar Pasupuleti, Thomas Moscibroda, Sameh Elnikety, Marcus Fontoura, and Ricardo Bianchini. Providing SLOs for Resource-Harvesting VMs in Cloud Platforms. In *OSDI*. USENIX, 2020.

[14] Alberto Bietti, Alekh Agarwal, and John Langford. A contextual bandit bake-off. *J. Mach. Learn. Res.*, 22:133–1, 2021.

[15] Dave Chiluk. Unthrottled: Fixing CPU Limits in the Cloud (blog post). https://engineering.indeedblog.com/blog/2019/12/unthrottled-fixing-cpu-limits-in-the-cloud/.

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

[17] Byungkwon Choi, Jinwoo Park, Chunghan Lee, and Dongsu Han. pHPA: A Proactive Autoscaling Framework for Microservice Chain. In *APNet*. ACM, 2021.

[18] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *SoCC*. ACM, 2015.

[19] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing Service Level Objectives for Cloud Services: Realities and Myths. In *ICAC*. IEEE, 2019.

[20] Miroslav Dudík, John Langford, and Lihong Li. Doubly robust policy evaluation and learning. *arXiv preprint arXiv:1103.4601*, 2011.

[21] David Lo et al. Towards Energy Proportionality for Large-scale Latency-critical Workloads. In *ISCA*, 2014.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *ASPLOS*. ACM, 2019.

[23] 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 *ASPLOS*. ACM, 2019.

[24] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven Autoscaling for Microservices. In *ICDCS*. IEEE, 2019.

[25] Giulio Santoli. Microservices Architectures: Become a Unicorn like Netflix, Twitter and Hailo, 2016.

[26] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. PRESS: Predictive Elastic Resource Scaling for Cloud Systems. In *CNSM*. IEEE, 2010.

[27] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. In *NSDI*. USENIX, 2011.

[28] Xiaofeng Hou, Chao Li, Jiacheng Liu, Lu Zhang, Shaolei Ren, Jingwen Leng, Quan Chen, and Minyi Guo. AlphaR: Learning-Powered Resource Management for Irregular, Dynamic Microservice Graph. In *IPDPS*. IEEE, 2021.

[29] Calin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, and Junhua Wang. PerfIso: Performance isolation for commercial latency-sensitive services. In *ATC*. USENIX, 2018.

[30] Seyyed Ahmad Javadi, Amoghavarsha Suresh, Muhammad Wajahat, and Anshul Gandhi. Scavenger: A Black-Box Batch Workload Resource Manager for Improving Utilization in Cloud Environments. In *SoCC*, 2019.

[31] Jeremy Cloud. Decomposing Twitter: Adventures in Service Oriented Architecture, 2013.

[32] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goirin, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards Automated SLOs for Enterprise Clusters. In *OSDI*, 2016.

[33] Anthony Kwan, Jonathon Wong, Hans-Arno Jacobsen, and Vinod Muthusamy. HyScale: Hybrid and Network Scaling of Dockerized Microservices in Cloud Data Centres. In *ICDCS*. IEEE, 2019.

[34] Cheuk Lam, Enlin Xu, and David Blinn. Kubernetes CPU Throttling: The Silent Killer of Response Time — and What to Do About It (blog post). https://community.ibm.com/community/user/aiops/blogs/dina-henderson/2022/06/29/kubernetes-cpu-throttling-the-silent-killer-of-res.

[35] John Langford and Tong Zhang. The Epoch-Greedy Algorithm for Multi-Armed Bandits with Side Information. *NIPS*, 2007.

[36] Kyungwoon Lee, Kwanhoon Lee, Hyunchan Park, Jaehyun Hwang, and Chuck Yoo. Autothrottle: Satisfying Network Performance Requirements for Containers. *IEEE Transactions on Cloud Computing*, 2022.

[37] Stuart Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, 1982.

[38] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. Serverless Computing: An Investigation of Factors Influencing Microservice Performance. In *ICCE*. IEEE, 2018.

[39] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving Resource Efficiency at Scale. In *ISCA*, 2015.

[40] 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 *SoCC*. ACM, 2021.

[41] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. Learning Scheduling Algorithms for Data Processing Clusters. In *SIGCOMM*. ACM, 2019.

[42] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. AGILE: Elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 69–82, 2013.

[43] Jinwoo Park, Byungkwon Choi, Chunghan Lee, and Dongsu Han. GRAF: A graph neural network based proactive resource allocation framework for SLO-oriented microservices. In *Proceedings of the 17th International Conference on emerging Networking EXperiments and Technologies*, pages 154–167, 2021.

[44] 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 *OSDI*. ACM, 2020.

[45] Gourav Rattihalli, Madhusudhan Govindaraju, Hui Lu, and Devesh Tiwari. Exploring potential for non-disruptive vertical auto scaling and resource estimation in Kubernetes. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, pages 33–40. IEEE, 2019.

[46] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at Google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[47] Vighnesh Sachidananda and Anirudh Sivaraman. Collective autoscaling for cloud microservices, 2021. arXiv:2112.14845.

[48] Tobias Schnabel, Adith Swaminathan, Ashudeep Singh, Navin Chandak, and Thorsten Joachims. Recommendations as treatments: Debiasing learning and evaluation. In *International Conference on Machine Learning*, pages 1670–1679. PMLR, 2016.

[49] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364, 2013.

[50] Vishwanath Seshagiri, Darby Huye, Lan Liu, Avani Wildani, and Raja R. Sambasivan. [SoK] identifying mismatches between microservice testbeds and industrial perceptions of microservices. *Journal of Systems Research*, 2(1), 2022.

[51] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011.

[52] Software Engineering Laboratory of Fudan University. Train Ticket: A Benchmark Microservice System. https://github.com/FudanSELab/train-ticket.

[53] Akshitha Sriraman and Thomas F. Wenisch. μTune: Auto-tuned threading for OLDI microservices. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 177–194, 2018.

[54] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: An introduction (second edition)*. MIT press, 2020.

[55] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *SoCC*, pages 1–16. ACM, 2013.

[56] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–17, 2015.

[57] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J. Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. SmartHarvest: Harvesting Idle CPUs Safely and Efficiently in the Cloud. In *EuroSys*. ACM, 2021.

[58] John Wilkes. Google cluster data – 2019 traces. https://github.com/google/cluster-data/blob/master/ClusterData2019.md, 2020.

[59] Zhengxu Xia, Yajie Zhou, Francis Y. Yan, and Junchen Jiang. Genet: Automatic curriculum generation for learning adaptation in networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 397–413, 2022.

[60] Francis Y. Yan, Hudson Ayers, Chenzhi Zhu, Sadjad Fouladi, James Hong, Keyi Zhang, Philip Levis, and Keith Winstein. Learning *in situ*: a randomized experiment in video streaming. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 495–511, Santa Clara, CA, February 2020. USENIX Association.

[61] Hailong Yang, Quan Chen, Moeiz Riaz, Zhongzhi Luan, Lingjia Tang, and Jason Mars. PowerChief: Intelligent Power Allocation for Multi-Stage Applications to Improve Responsiveness on Power Constrained CMP. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pages 133–146, 2017.

[62] Yanqi Zhang, Inigo Goiri, Gohar Irfan Chaudhry, Rodrigo Fonseca, Sameh Elnikety, Christina Delimitrou, and Ricardo Bianchini. Faster and Cheaper Serverless Computing on Harvested Resources. In *SOSP*. ACM, 2021.

[63] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-based and QoS-Aware Resource Management for Cloud Microservices. In *ASPLOS*. ACM, 2021.

[64] 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 *SoCC*. ACM, 2018.

Appendices

A Application workload details

We present the workload composition used in our experiments. Our workload generator, Locust, follows the ratios specified below when generating requests at a given RPS:

Train-Ticket:

- `Mainpage`: 29.41%
- `Travel`: 58.82%
- `Assurance`: 2.94%
- `Food`: 2.94%
- `Contact`: 2.94%
- `Preserve`: 2.94%

Hotel-Reservation:

- `Search`: 60%
- `Recommend`: 39%
- `Reserve`: 0.5%
- `Login`: 0.5%

Social-Network:

- `Read-home-timeline`: 65%
- `Read-user-timeline`: 15%
- `Compose-post`: 20%

B Vowpal Wabbit usage

The following VW parameters are used in our experiments.

- The doubly robust estimator [20] is employed for policy evaluation: `--cb_type dr`
- Number of available actions: `--cb_explore 81`
- The native ε-greedy is disabled to implement our customized exploration strategy (§3.3.2): `--epsilon 0`
- Number of hidden units in the neural network: `--nn 3`
- Learning rate: `-l 0.5`

We also compare different VW models—a linear model and neural networks with 2, 3, 4 and hidden units, on Social-Network under the same workload patterns (Figure 3). Figure 11 shows that Autothrottle perform similarly across different models. We select the neural network model with 3 hidden units, as it performs slightly better on the bursty workload (as indicated by the lower whiskers in boxplots).

C Microservice clustering

Tower clusters microservices into two groups based on their average CPU usage, denoted "High" and "Low", using a standard k-means clustering algorithm (§3.3.2). Table 2 presents a breakdown of the number of services in each group.

D Microservice replicas

Train-Ticket and Hotel-Reservation deploy each service with one replica. For Social-Network, we employ three replicas of `media-filter-service` except in the large-scale evaluation (§5.5). In §5.5, 6 replicas of `media-filter-service` and 3 replicas of `nginx-thrift` are employed.

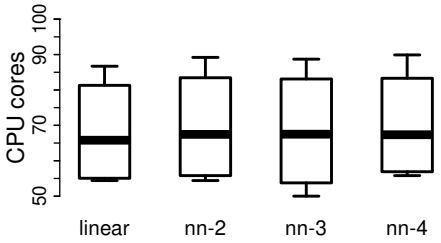


Figure 11: Different VW models—a linear model and neural networks with 2, 3, and 4 hidden units—perform similarly on Social-Network under various workloads (Figure 3).

Application	"High" group	"Low" group
Train-Ticket	8	60
Hotel-Reservation	6	11
Social-Network (160-core cluster)	1	27
Social-Network (512-core cluster)	2	26

Table 2: Number of services in each application assigned to the "High" and "Low" CPU usage groups.

E RPS range of workload traces

We scale the traces presented in Figure 3 to saturate the cluster for each application, as documented in Table 3.

F CPU utilization thresholds in K8s-CPU and K8s-CPU-Fast

In evaluating K8s-CPU and K8s-CPU-Fast, we test and select the best-performing CPU utilization threshold from the set $\{0.1, 0.2, \ldots, 0.9\}$, for each application and each workload trace. The selected thresholds are presented in Table 4.

G Evaluation methodology details

All experiments are performed using one-hour workload traces. Prior to testing, certain applications require additional preparations. We warm up Hotel-Reservation by sending 200 requests per second for 15 seconds and waiting for 60 seconds. For Social-Network, we populate the database with 962 users, 18,812 edges, and 20,000 posts. We then warm up for 3 minutes by incrementally increasing the RPS by 10% every 5 seconds, up to the initial RPS in the one-hour trace. The warm-up phase is excluded from the calculation of P99 latency and resource allocation.

Autothrottle shown in Table 1 is warmed up for 12 hours. The first 6 hours are the (random) exploration stage, followed by 6 hours of normal learning with ε=0.5. We use a separate 1-hour diurnal trace, which is different from the one used for testing but has the same RPS range. Warm up involves running 12 repetitions of this trace. During testing, exploration is turned off completely (with ε set to 0). For Hotel-Reservation,

Workload	Min RPS	Average RPS	Max RPS
Diurnal	145	262	411
Constant	152	200	252
Noisy	75	157	252
Bursty	62	163	442

(a) Train-Ticket

Workload	Min RPS	Average RPS	Max RPS
Diurnal	1721	2627	4003
Constant	1855	2002	2183
Noisy	793	1575	2470
Bursty	768	1633	4037

(b) Hotel-Reservation

Workload	Min RPS	Average RPS	Max RPS
Diurnal	227	394	656
Constant	390	500	588
Noisy	105	236	390
Bursty	104	245	648
Long-term (§5.4)	1	230	592

(c) Social-Network

Workload	Min RPS	Average RPS	Max RPS
Diurnal	479	787	1214
Constant	882	1001	1131
Noisy	232	472	771
Bursty	205	489	1266

(d) Social-Network, large-scale evaluation (§5.5)

Table 3: The RPS range of workload traces after being scaled to saturate the cluster for each application.

RPS is grouped into bins of 200 due to its high RPS, while other applications use the default bin size of 20.

H Captain performance

We demonstrate Captain's ability to achieve Tower's given performance target of CPU throttle ratio (§3.2). This is one factor that determines Autothrottle's effectiveness in maintaining the end-to-end SLO. To this end, Figure 12 dives into Social-Network and illustrates two services from "High" and "Low" CPU usage groups (§C): `media-filter-service` and `post-storage-service`. Two subfigures compare the target throttle ratio and the actual throttle ratio, over a period of 60 min. Captain's heuristics meets the targets relatively well and reacts quickly to target changes, especially when the target is low (Figure 12b). In Figure 12a, we note that the actual throttle ratio is lower than the target. The reason is that the throttle ratio is very sensitive to CPU allocation, especially when the target is high. As a result, Captain tries to err on the safe side, and it can over-allocate to avoid exceeding the targeted throttle ratio due to estimation errors.

Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.4	0.6
Constant	0.6	0.6
Noisy	0.5	0.7
Bursty	0.5	0.6

(a) Train-Ticket

Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.7	0.7
Constant	0.7	0.8
Noisy	0.6	0.7
Bursty	0.5	0.7

(b) Hotel-Reservation

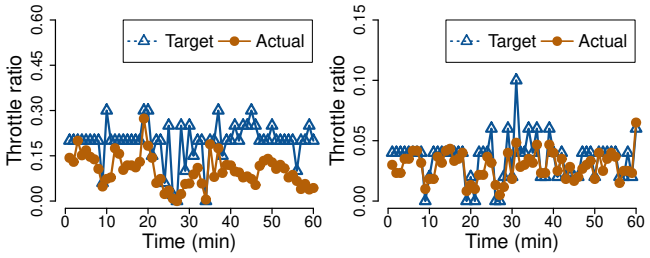
Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.5	0.5
Constant	0.5	0.6
Noisy	0.5	0.4
Bursty	0.5	0.4
Long-term (§5.4)	0.5	–

(c) Social-Network

Workload	K8s-CPU	K8s-CPU-Fast
Diurnal	0.6	0.7
Constant	0.5	0.8
Noisy	0.5	0.5
Bursty	0.5	0.7

(d) Social-Network, large-scale evaluation (§5.5)

Table 4: The best-performing CPU utilization thresholds for comparison baselines, per application and workload trace.



(a) `media-filter-service` (b) `post-storage-service`

Figure 12: Captain is able to follow the given performance target over time, by adjusting per-service resources.