Abstract
Modern user-facing, latency-sensitive web services include numerous distributed, intercommunicating microservices that promise to simplify software development and operation. However, multiplexing compute-resources across microservices is still challenging in production because contention for shared resources can cause latency spikes that violate the service-level objectives (SLOs) of user requests. This paper presents FIRM, an intelligent fine-grained resource management framework for predictable sharing of resources across microservices to drive up overall utilization. FIRM leverages online telemetry data and machine-learning methods to adaptively (a) detect/localize microservices that cause SLO-violations, (b) identify low-level resources in contention, and (c) take actions to mitigate SLO-violations by dynamic re-provisioning. Experiments across four microservice benchmarks demonstrate that FIRM reduces SLO violations by up to $16.7 \times$ while reducing the overall requested CPU limit by up to 62.3%. Moreover, FIRM improves performance predictability by reducing tail latencies by up to $11.5 \times$.

1 Introduction
Modern user-facing, latency-sensitive web services, like those at Netflix [64], Google [72], and Amazon [82], are increasingly built as microservices executing on shared/multi-tenant compute resources either as virtual machines (VMs) or as containers (with containers gaining significant popularity of late). These microservices must handle diverse load characteristics while efficiently multiplexing shared resources in order to maintain service level objectives (SLOs) like end-to-end latency. SLO violations occur when one or more “critical” microservice instances (defined in §2) experience load spikes (due to diurnal or unpredictable workload patterns) or shared-resource contention, both of which lead to longer than expected time to process requests, i.e., latency spikes [3, 9, 18, 26, 31, 48, 53, 54, 90, 91]. Thus, it is critical to efficiently multiplex shared resources among microservices to reduce SLO violations.

Traditional approaches (e.g., overprovisioning [32, 80], re-
current provisioning [49, 62], and autoscaling [35, 51, 61, 75, 78, 81, 117]) reduce SLO violations by allocating more CPUs and memory to microservice instances by using performance models, handcrafted heuristics (i.e., static policies), or machine-learning algorithms.

Unfortunately, these approaches suffer from two main problems. First, they fail to efficiently multiplex resources at fine granularity, such as caches, memory, I/O channels, and network links, and thus may not reduce SLO violations. For example, in Fig. 1, the Kubernetes container-orchestration system [16] is unable to reduce the tail latency spikes arising from contention for a shared resource like memory bandwidth, as its autoscaling algorithms are built using heuristics that only monitor CPU utilization, which does not change during the latency spike. Second, building high fidelity performance models (and related scheduling heuristics) of large-scale microservice deployments (e.g., queuing systems [23, 35]) that can capture low-level resource contention requires significant human-effort and training. Further, frequent microservice updates and migrations can lead to recurring human-expert-driven engineering effort for model reconstruction.

FIRM Framework. This paper addresses the above problems by presenting FIRM, a multilevel machine learning (ML)-based resource management (RM) framework to manage shared resources among microservices at finer granularity.
to reduce resource contention and thus increase performance isolation and resource utilization. As shown in Fig. 1, FIRM performs better than a default Kubernetes autoscaler because FIRM adaptively scales up the microservice (by adding local cores) to increase the aggregate memory bandwidth allocation, thereby effectively maintaining the per-core allocation. FIRM leverages online telemetry data (such as request-tracing data and hardware counters) to capture the system state, and ML models for resource contention estimation and mitigation. Telemetry data and ML models enable FIRM to adapt to workload changes and alleviate the need for brittle, hand-crafted heuristics. In particular, FIRM uses the following ML models:

- **Support vector machine (SVM)-driven detection and localization of SLO violations to individual microservice instances.** FIRM first identifies the “critical paths”, and then uses per-critical-path and per-microservice-instance performance variability metrics (e.g., sojourn time [1]) to output a binary decision of whether or not a microservice instance is responsible for SLO violations.

- **Reinforcement learning (RL)-driven mitigation framework that reduces contention on shared resources.** FIRM then uses resource utilization, workload characteristics, and performance metrics to make dynamic reprovisioning decisions, which include (a) increasing or reducing the partition portion or limit for a resource type, and (b) scaling-up/out or -down, i.e., adding or reducing the amount of resources attached to a container. By continuing to learn mitigation policies through reinforcement, FIRM can optimize for dynamic workload-specific characteristics.

**Online Training for FIRM.** We developed a performance anomaly injection framework that can artificially create resource scarcity situations in order to both train and assess the proposed framework. The injector is capable of injecting resource contention problems at a fine granularity (such as last-level cache and network devices) to trigger SLO violations. To enable rapid (re)training of the proposed system as the underlying systems [63] and workloads [36, 38, 89, 90] change in datacenter environments, FIRM uses transfer learning. That is, FIRM leverages transfer learning to train microservice specific RL-agents based on previous RL experience.

**Contributions.** To the best of our knowledge, this is the first work to provide a SLO violation mitigation framework for microservices using fine-grained resource management and in an application architecture agnostic way with multi-level ML models. Our main contributions are:

1. **SVM-based SLO Violation Localization:** We present (in §3.2 and §3.3) an efficient way of localizing microservice instances responsible for SLO violations by extracting critical paths and detecting anomaly instances in near-real time using telemetry data.

2. **RL-based Mitigation:** We present (in §3.4) an RL-based resource contention mitigation mechanism that (a) addresses the large state space problem and (b) is capable of tuning tailored RL agents for individual microservice instances using transfer learning.

3. **Online Training & Performance Anomaly Injection:** We propose (in §3.6) a comprehensive performance anomaly injection framework to artificially create resource contention situations, thereby generating the ground-truth data required for training the aforementioned ML models.

4. **Implementation & Evaluation:** We provide an open-source implementation of FIRM for the Kubernetes container-orchestration system [16]. We demonstrate and validate this implementation on four real-world microservice benchmarks [30, 107] (in §4).

**Results.** FIRM significantly outperforms state-of-the-art RM frameworks like Kubernetes autoscaling [16, 50] and additive increase multiplicative decrease (AIMD)-based methods [34, 93].

- It reduces overall SLO violations by up to 16.7× compared with Kubernetes autoscaling, and 9.8× compared with AIMD-based method, while reducing the overall requested CPU by as much as 62.3%.
- It outperforms AIMD-based method by up to 9.6× and Kubernetes autoscaling by up to 30.1× in terms of the time to mitigate SLO violations.
- It improves overall performance predictability by reducing the average tail latencies up to 11.5×.
- It successfully localizes SLO violation root-cause microservice instances with 93.8% accuracy.

**Why does FIRM work?** FIRM allows mitigation of SLO violations without overprovisioning, which we attribute to the following reasons. First, modeling dependency between low-level resources and application performance in an RL-based feedback loop to deal with uncertainty and noisy measurements; and Second, taking a two-level approach where the SVM model filters only those microservices that needs to be considered for mitigating SLO violations, thus making the framework application-architecture agnostic as well as enabling the RL agent to be trained faster.

## 2 Background & Characterization

The advent of microservices has led to the development and deployment of many web services that are composed of “micro”, loosely coupled, intercommunicating services, instead of large, monolithic designs. This increased popularity of service-oriented architectures (SOA) of web services has been possible with the rise of containerization [17, 65, 85, 99] and container-orchestration frameworks [15, 16, 83, 109] that enable modular, low-overhead, low-cost, elastic, and high-efficiency development and production deployment of SOA microservices [6, 7, 29, 30, 41, 64, 72, 82, 95]. A deployment of such microservices can be visualized as a service dependency graph or an execution history graph. Performance of a user request, i.e., its end-to-end latency, is determined by the

---

1 All data and source code used in FIRM will be made available freely under an OSS license on acceptance of the paper.
critical path of its execution history graph.

**Definition 2.1.** A Service Dependency Graph captures communication-based dependencies (the edges of the graph) between microservice instances (the vertices of the graph), such as remote procedure calls (RPCs). Fig. 2(a) shows the service dependency graph of the Social Network microservice benchmark [30]. Each user request traverses a subset of vertices in the graph. For example in Fig. 2(a), post-compose requests traverse only those microservices highlighted in the darker color.

**Definition 2.2.** An Execution History Graph is the space-time diagram of the distributed execution of a user request, where a vertex is one of send_req, recv_req and compute, and edges represent the RPC invocations corresponding to send_req and recv_req. The graph is constructed using the global view of execution provided by distributed tracing of all involved microservices. For example, Fig. 2(b) demonstrates the execution history graph for the user request in Fig. 2(a).

**Definition 2.3.** The Critical Path (CP) to a microservice \( m \) in the execution history graph of a request is the path of maximal duration that starts with the client request and ends with \( m \) [60, 115]. When used without the target microservice \( m \), we use CP to mean the critical path of the “Service Response” to the client (see Fig. 2(b)), i.e., end-to-end latency.

We have run extensive performance anomaly injection experiments on widely used microservice benchmarks (i.e.

---

**Figure 2:** Microservices overview: (a) Service dependency graph of Social Network from the DeathStarBench [30] benchmark; (b) Execution history graph of a post-compose request in the same microservice.

**Figure 3:** Distributions of end-to-end latencies of different microservices in the DeathStarBench [30] and Train-Ticket [107] benchmarks. The two lines in each figure correspond to the minimum and maximum latencies across all CPs on serving a request.

**Table 1:** CP changes in Fig. 2(b) under performance anomaly injection. Each case is represented by a <\textit{service}, CP> pair. \( N, V, U, I, T, \) and \( C \) are microservices from Fig. 2.

| Case        | Individual Latency (ms) | Total (ms) |
|-------------|-------------------------|------------|
| \(<V,CP1>\) | 3.2 231.6 89.9 24.5 35.8 47.6 | 234.8      |
| \(<U,CP2>\) | 2.3 70.2 344.6 28.9 25.7 61.3 | 375.8      |
| \(<T,CP3>\) | 1.9 74.3 99.4 25.4 193.1 54.0 | 249.0      |

DeathStarBench [30] and Train-Ticket [107]) and collected 2TB of microservice tracing data (over \( 4.1 \times 10^7 \) traces). Our key insights are as follows.

**Insight 1:** Dynamic Behavior of CPs. In microservices, the latency of the CP limits the overall latency of a user request in a microservice. However, CPs do not remain static over the execution of requests in microservices but rather change dynamically based on the performance of individual service instances due to underlying shared-resource contention and microservice sensitivity to this interference. For example, in Fig. 2(b), we show the existence of three different CPs (i.e., CP1-CP3) depending on which microservice (i.e., \( V, U, T \)) encounters resource contention. We artificially create resource contention using performance anomaly injections.\(^2\) Table 1 lists the changes observed in the latencies of individual microservices, as well as end-to-end latency. We observe as much as \( 1.6 \times \) variation in end-to-end latency.

\(^2\)Performance anomaly injection (§3.6) is used to trigger SLO violations by generating resource contention with configurable intensity, duration and timing.
across the three CPs. Such dynamic behavior exists across all our benchmark microservices. Fig. 3 illustrates the latency distributions of CPs with minimum and maximum latency in each microservice benchmark, where we observe as much as $1.6 \times$ difference in median latency and $2.5 \times$ difference in 99%-ile tail latency across these CPs.

Recent approaches (e.g., [2,42]) have explored static identification of CPs based on historic data (profiling) and have built heuristics (e.g., application placement, level of parallelism) to enable autoscaling to minimize latency of the CP. However, our experiment shows that this by itself is not sufficient. The requirement is to *adaptively capture changes in the CPs*, in addition to changing resource allocations to microservice instances on the identified CPs to mitigate tail latency spikes.

**Insight 2: Microservices with Larger Latency Are Not Necessarily Root Causes of SLO Violations.** It is important to find the microservices responsible for SLO violation to mitigate them. While it is clear that such microservices will always lie on the CP, it is less clear which individual service on the CP is responsible for the violation. A common heuristic is to pick the one with the highest latency. However, we find that rarely leads to the optimal solution. Consider Fig. 4, here the left figure shows the CDF of latencies of two services (i.e., composePost and text) on the CP of the post-compose request in Social Network benchmark. The composePost service has higher median/mean latency while text service has higher variance. Now, although composePost service contributes a larger portion of the total latency, it does not benefit from scaling (i.e., getting more resources) as it does not have resource contention. This behavior is shown in Fig. 4 (right), which shows the end-to-end latency for the original configuration (labelled “Before”) as well as when each of the two microservices are scaled from a single to two containers (labelled “Text” and “Compose”). Hence, scaling microservices with higher variance provides better performance gain.

**Insight 3: Mitigation Policies Vary with User Load and the Resource in Contention.** The only way to mitigate the effects of dynamically changing CPs, which in turn cause dynamically changing latencies and tail behaviors, is to efficiently identify microservice instances on the CP that are resource-starved or contending for resources and then provide them with more of the resources. Two common ways of doing so are to (a) *scale out* by spinning up a new instance of the container on another node of the compute cluster or (b) *scale up* by providing more resources to the container by either explicitly partitioning resources (e.g., in the case of memory or last-level cache) or by granting more resources to an already deployed container of the microservice (e.g., in the case of CPU cores).

As described before, recent approaches [19,34,35,51,61,78,87,93,117] address this problem by building static policies (e.g., AIMD for controlling resource limits [34,93], rule/heuristics-based scaling based on profiling historic data about a workload [19,87]), and performance modeling [35,51]. However, we find in our experiments with the four microservice benchmarks that such static policies are not well suited to deal with latency-critical workloads because the optimal policy must incorporate dynamic contextual information.

That is, information about the type of user requests, load (in requests per second), as well as the critical resource bottlenecks (i.e., the one being contented for), must be jointly analyzed to make optimal decisions. For example, in Fig. 5 (upper), we observe that trade-off for scale up vs. scale out not only changes based on the user load but also on the resource type. At 500 req/s, scale up has a better payoff (i.e, lower latency) than scale out for both memory and CPU bound workloads. However at 1500 req/s, scale out dominates for CPU and scale up dominates for memory. This behavior is application dependent because the trade-off curve inflection points change across applications, as illustrated in Fig. 5 (lower).

**3 The FIRM Framework**

In this section, we describe the overall architecture of the FIRM framework and its implementation using Fig. 6.

1. Based on the insight that resource contention manifests as dynamically evolving CPs, FIRM first detects CP changes and extracts critical microservice instances from it. This is done using the *Tracing Coordinator*, which is illustrated as in Fig. 6. It collects tracing and telemetry data from every microservice instance and stores it in a centralized graph database for processing. It is described in §3.1

2. The *Extractor* detects SLO violations and queries the Tracing Coordinator to collect real-time data to (a) extract CPs (illustrated as in §3.2) and (b) localize

\[3\]Unless otherwise specified, refers to annotations in Fig. 6.
critical microservice instances that are likely causes of SLO violations (illustrated as 3 and described in §3.3).

3. Using the telemetry data collected in 1 and the critical instances identified in 3, FIRM makes mitigation decisions to scale and reapportion resources for the critical instances (illustrated as 4). The policy used to make such decisions is automatically generated using RL. The RL-agent jointly analyzes contextual information about resource utilization (i.e., low-level performance counter data collected from CPU, LLC, Memory, I/O, and Network), performance metrics (i.e., per-microservice and end-to-end latency distributions), and workload characteristics (i.e., request arrival rate and composition) and makes mitigation decisions. The RL setup is described in §3.4.

4. Finally, actions are validated and actuated on the underlying Kubernetes cluster through the deployment module (illustrated as 5 and described in §3.5).

5. In order to train the RL-agent (i.e., span the exploration-exploitation trade-off space), FIRM includes a performance anomaly injection framework that triggers SLO violations by generating resource contention with configurable intensity and timing. This is illustrated as 6 and described in §3.6.

3.1 Tracing Coordinator

Distributed tracing is a method used to profile and monitor microservice-based applications to pinpoint causes of poor performance [102–106]. A trace captures the work done by each service along request execution paths, i.e., it follows the execution “route” of a request across microservice instances recording time, local profiling information, and RPC calls (e.g., source and destination services). These execution paths are combined to form the execution history graph (recall from §2). The time spent by a single request in a microservice instance is called its span. The span is calculated based on the time when a request arrives at a microservice and when its response is sent back to the caller. Each span is the most basic single unit of work done by a microservice.

The FIRM tracing module’s design is heavily inspired by Dapper [88] and its open-source implementations, e.g., Jaeger [103], Zipkin [106]. Each microservice instance is coupled with an OpenTracing [70]-compliant tracing agent that measures span. As a result, any new OpenTracing-compliant microservice can be integrated naturally into the FIRM tracing architecture. The Tracing Coordinator, i.e., 1, is a stateless, replicable data-processing component that collects the spans of different requests from each tracing agent, combines them, and stores them in a graph database [67] as the execution history graph. The graph database allows us to easily store complex caller-callee relationships among microservices depending on request types, as well as to efficiently query the graph for critical path/component extraction in §3.2 and §3.3. Distributed clock drift and time shifting are handled using the Jaeger framework. Additionally, the Tracing Coordinator collects telemetry data from the systems running the microservices. These data are listed in Table 2. The distributed tracing and telemetry collection overhead is indiscernible, i.e., <0.2% loss in throughput and <0.11% loss in latency.

3.2 Critical Path Extractor

The first goal of the FIRM framework is to quickly and accurately identify the CP based on the tracing and telemetry data described in the previous section. Recall from Def. 2.3 in §3 that a CP is the longest path in the request’s execution history graph. Hence, changes in end-to-end latency of an application is often determined by the slowest execution of one or more microservices on its CP.

We identify the CP in a execution history graph using Alg. 1. A weighted longest path algorithm is proposed to retrieve CPs. The algorithm needs to take into account the major communication and computation patterns in microservice architectures: (a) sequential, (b) parallel, and (c) background workflows.

- **Parallel workflows** are the most common way of processing requests in microservices. They are characterized by child spans of the same parent span overlapping with each other in the execution history graph, e.g., U, V, and T in Fig 2(b).
- **Sequential workflows** are characterized by one or more

| Table 2: Collected telemetry data and sources. |
|-----------------------------------------------|
| **cAdvisor** [11] & **Prometheus** [76]       |
| cpu_usage_seconds_total, memory_usage_bytes,  |
| fs_read/write_seconds, fs_usage_bytes,        |
| network_transmit/receive_bytes_total, processes |
| **Linux perf subsystem** [73]                 |
| offcore_response.*, llc_hit/miss.local_DRAM,  |
| offcore_response.*, llc_hit/miss.remote_DRAM   |

![Figure 6: FIRM architecture overview.](image)
Algorithm 1 Critical Path Extraction

Require: Microservice execution history graph $G$
Attributes: childNodes, lastReturnedChild

procedure LONGESTPATH($G$, currentNode)
1: path ← ∅
2: path.add(currentNode)
3: if currentNode.childNodes == None then
4:   Return path
5: end if
6: i ← currentNode.childNodes
7: getLatency(i)
8: path.extend(LONGESTPATH($G$, i))
9: for each $cn$ in currentNode.childNodes do
10:   if $cn$.happensBefore(i) then
11:     path.extend(LONGESTPATH($G$, cn))
12: end if
13: end for
14: Return path
15: end procedure

child span of the parent span that are processed in a serialized manner, e.g., $U$ and $I$ in Fig. 2(b). For two of $p$’s child-spans $i$ and $j$ to be in a sequential workflow, the time $t_{i→p}^j ≤ t_{p→j}^i$, i.e., $i$ completes and sends its result to $p$ before $j$. Such sequential relationships are usually indicative of a happens-before relationship. However, it is impossible to ascertain the relationships merely by observing traces from the system. If across many request executions, there is a violation of this inequality, then the services are not sequential.

- Background workflow are those which do not return values to their parent spans, e.g., $W$ in Fig. 2(b). Background workflows are not part of CPs but they may be considered to be culprits of SLO violations when FIRM’s Extractor is localizing critical components (in §3.3).

3.3 Critical Component Extractor

In each extracted CP, FIRM then uses an adaptive, data-driven approach to determine critical components (i.e., microservice instances). The overall procedure is shown in Alg. 2. The extraction algorithm first calculates per-CP and per-instance “features”, which represent performance variability and level of request congestion. This is because variability represents the single largest opportunity to reduce tail latency. These features are then fed into an incremental SVM classifier to get binary decisions, i.e., whether that instance should have its resources re-provisioned or not. This represents a dynamic selection policy, which is in contrast to static policies, as it can classify critical and noncritical components adapting to dynamically changing workload and variation patterns.

In order to extract those microservice instances that are potential candidates for SLO violations, we argue that it is critical to know both the variability of the end-to-end latency (per-CP variability) and the variability caused by congestion in per-instances service queue (per-instance variability).

Algorithm 2 Critical Component Extraction

Require: Critical Path CP, Request Latencies $T$

procedure CRITICALCOMPONENT($G$, $T$)
1: candidates ← ∅
2: $T_{CP}$ ← $T$.getTotalLatency() $\triangleright$ Vector of CP latencies
3: for $i ∈ CP$ do
4:   $T_i ← T$.getLatency($i$)
5:   $T_{99} ← T_i$.percentile(99)
6:   $T_{50} ← T_i$.percentile(50)
7: $RI ← PCC(T_i, T_{CP})$ $\triangleright$ Relative Importance
8: $CI ← T_{99}/T_{50}$ $\triangleright$ Congestion Intensity
9: if $SVM.classify(RI, CI) == True$ then
10:   candidates.append($i$)
11: end if
12: end for
13: return candidates
14: end procedure

Per-CP Variability: Relative Importance. Relative importance [58, 101, 112] is a metric that quantifies the strength of the relationship between two variables. For each critical path CP, its end-to-end latency is given by $T_{CP} = \sum_{i \in CP} T_i$, where $T_i$ is the latency of microservice $i$. Our goal is to determine the contribution the variance of each variable $T_i$ makes toward explaining the total variance of $T_{CP}$. To do this, we use the Pearson correlation coefficient [10], i.e., $PCC(T_i, T_{CP})$, as the measurement, and hence, the resulting statistic is known as variance explained [27].

Per-Instance Variability: Congestion Intensity. For each microservice instance in a CP, congestion intensity is defined as the ratio of 99th-percentile latency divided by the median latency. Here, we choose 99th percentile instead of 70th or 80th percentile to target the tail latency behavior. This ratio explains per-instance variability by capturing the congestion level of the request queue so that it can be used to determine whether it is necessary to scale. For example, a higher ratio means that the microservice could only handle some of requests but the requests at the tail are suffering from congestion issues in the queue. On the other hand, microservices with lower ratios handle most requests normally, so scaling does not help with performance gain. Consequently, microservice instances with higher ratios have a greater opportunity to achieve performance gain in terms of tail latency by taking scale-out or reprovisioning actions.

Implementation. The logic of critical path extraction is incorporated into the construction of spans, i.e., as the algorithm proceeds (Alg. 1), the order of tracing construction is also from the root node to child nodes recursively along the execution history graph. Sequential, parallel, and background workflows are inferred from the parent-child relationships of spans. Then, for each CP, we calculate feature statistics and feed them into an incremental SVM classifier [25, 52] implemented using stochastic gradient descent optimization and RBF kernel approximation by scikit-learn libraries [84].
3.4 SLO Violation Mitigation Using RL

Given the list of critical service instances, FIRM’s Resource Estimator, i.e., FIG. 4, is designed to analyze resource contention and provide reprovisioning actions for the cluster manager to take. FIRM estimates and controls a fine-grained set of resources including CPU time, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth. It makes decisions on scaling each type of resource or the number of containers using measurements of tracing and telemetry data (recall measurements from Table 2) collected from the Tracing Coordinator. When jointly analyzed, these provide information about (a) shared-resource interference, (b) workload rate variation, and (c) request type composition.

FIRM leverages reinforcement learning (RL) to optimize resource management policies for long-term reward in dynamic microservice environments. In particular, FIRM utilizes the deep deterministic policy gradient (DDPG) algorithm [55], which is a model-free, actor-critic RL framework (shown in Fig. 7).

Why RL? Existing performance-modeling-based [19, 34, 35, 51, 87, 93, 117] or heuristic-based approaches [4, 5, 33, 61, 78] suffer from model reconstruction and retraining problems because they do not tackle with dynamic system status. Moreover, they require expert knowledge with significant effort to devise, implement and validate their understanding of the microservice workloads as well as the underlying infrastructure. RL on the other hand is well suited for learning resource reprovisioning policies as it provides a tight feedback-loop for exploring action space and generating optimal policies without relying on inaccurate assumptions (i.e., heuristics or rules). It allows direct learning from actual workload and operating conditions to understand how adjusting low-level resources affects application performance. Further, FIRM’s RL formulation provides two distinct advantages:

1. Model-free RL does not need the ergodic distribution of states or the environment dynamics (transition between states), which are difficult to model precisely. When microservices are updated, the simulation of state-transition used in model-based RL is no longer valid.

Algorithm 3 DDPG Training

1. Randomly init $Q_\theta(s,a)$ and $\pi_\theta(a|s)$ with weights $w$ & $\theta$.
2. Init target network $Q'$ and $\pi'$ with $w' \leftarrow w$ & $\theta' \leftarrow \theta$.
3. Init replay buffer $D \leftarrow \emptyset$.
4. for episode $= 1, M$ do
5. Initialize a random process $\mathcal{N}$ for action exploration.
6. Receive initial observation state $s_1$.
7. for $t = 1, T$ do
8. Select and execute action $a_t = \pi_\theta(s_t) + \mathcal{N}$.
9. Observe reward $r_t$ and new state $s_{t+1}$.
10. Store transition $(s_t, a_t, r_t, s_{t+1})$ in $D$.
11. Sample $N$ transitions $(s_t, a_t, r_t, s_{t+1})$ from $D$.
12. Update critic by minimizing the loss $L(w)$.
13. Update actor by sampled policy gradient $\nabla \mathbb{E}_{D}[\mathcal{L}(w)]$.
14. $w' \leftarrow w + (1 - \gamma)w'$.
15. $\theta' \leftarrow \theta + (1 - \gamma)\theta'$.
16. end for.
17. end for.

2. Actor-critic combines policy-based and value-based methods, which is suitable for continuous, stochastic environment, converges faster, and has lower variance [37].

RL Primer. An RL agent solves a sequential-decision-making problem by interacting with an environment. At each discrete time step $t$, the agent observes a state of the environment $s_t \in \mathbb{S}$, and performs an action $a_t \in \mathbb{A}$ based on its policy $\pi_\theta(s)$ (parameterized by $\theta$), which maps state space $\mathbb{S}$ to action space $\mathbb{A}$. At the following time step $t+1$, the agent observes an immediate reward $r_t \in \mathbb{R}$ given by a reward function $r(s_t, a_t)$, representing the loss/gain in transitioning from $s_t$ to $s_{t+1}$ due to action $a_t$. The tuple $(s_t, a_t, r_t, s_{t+1})$ is called a transition. The agent’s goal is to optimize the policy $\pi_\theta$ so as to maximize the expected cumulative discounted reward from the start distribution $J = \mathbb{E}[G_t | s_0]$, where the return from a state $G_t$ is defined to be $\sum_{k=0}^{\infty} \gamma^k r_{t+k}$. The discount factor $\gamma \in (0, 1]$ penalizes the predicted future rewards.

Learning the Optimal Policy. DDPG’s policy learning is an actor-critic approach. Here the “critic” estimates the value function (i.e., the expected value of cumulative discounted reward under a given policy); and the “Actor” updates the policy in the direction suggested by the critic. Its estimation of the expected return allows for the actor to update with gradients that have lower variance, thus speeding up the learning process. We further assume that the actor and critic are represented as deep neural networks. DDPG also solves the issue of dependency between samples and makes use of hardware optimizations by introducing a replay buffer, which is a finite-sized cache $\mathcal{R}$ storing transitions $(s_t, a_t, r_t, s_{t+1})$. Parameter updates are based on a mini-batch of size $N$ sampled from the reply buffer. The pseudo-code of the training algorithm is shown in Algorithm 3.

In critic, the value function $Q_w(s_t, a_t)$ with parameter $w$ and its corresponding loss function are defined as:

$$Q_w(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q_w(s_{t+1}, \pi(s_{t+1}))]$$
\[ L(w) = \frac{1}{N} \sum_i (r_i + \gamma Q'_w(s_{i+1}, \pi'_w(s_{i+1})) - Q_w(s_i, a_i))^2. \]

The target networks \( Q'_w(s, a) \) and \( \pi'_w(s) \) are introduced in DDPG to mitigate the problem of instability and divergence when directly implementing deep RL-agents. In the actor component, DDPG maintains a parametrized actor function \( \pi_\theta(s) \) which specified the current policy by deterministically mapping states to a specific action. The actor is updated by:

\[ \nabla_a J = \frac{1}{N} \sum_i \nabla_a Q_w(s = s_i, a = \pi(s_i)) \nabla_\theta \pi_\theta(s = s_i). \]

**Problem Formulation.** To estimate resources for a microservice instance, we formulate it as a sequential decision-making problem which can be solved by the above RL framework. Each microservice instance is deployed in a container with resource limit \( RLT = (RLT_{cpu}, RLT_{mem}, RLT_{io}, RLT_{net}) \), since we are considering CPU utilization, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth as our resource model. This limit for each type of resource is predetermined before deployed (e.g., overprovisioned) in the cluster and later controlled by FIRM.

At each time step \( t \), utilization \( RU_i \) for each type of resource is retrieved using performance counters as telemetry data in \( \text{CPU} \). In addition, FIRM’s Extractor also collects current latency, request arrival rate, and request type composition (i.e., percentages of each type of request). Based on these measurements, RL agent calculates states listed in Table 3 and described below.

- **SLO violation ratio** \( (SV_i) \) is defined as SLO \_latency/current_latency if the microservice instance is determined to be the culprit. If no message arrives, it is assumed that there is no SLO violation \( (SV_i = 1) \).
- **Workload changes** \( (WC_i) \) is defined as the ratio of arrival rates of the current and previous time steps.
- **Request composition** \( (RC_i) \) is defined as a unique value encoded from an array of request percentages using numpy.ravel_multi_index() [69].

For each type of resources \( i \), there are predefined resource upper limit \( \hat{R}_i \) and lower limit \( \check{R}_i \) (e.g., CPU time limit can-

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
\textbf{State (\( s_i \))} & \textbf{Action Space (\( a_i \))} \\
\hline
SLO Violation Ratio \( (SV_i) \), Workload Changes \( (WC_i) \), Request Composition \( (RC_i) \), Resource Utilization \( (RU_i) \) & Resource Limits \( RLT(t), i \in \{CPU, Mem, LLC, IO, Net\} \) & \\
\hline
\end{tabular}
\end{table}

\text{not be set to 0). The actions available to the RL-agent is to set \( RLT_i \in [\check{R}_i, \hat{R}_i] \). If the amount of resource reaches the total available amount, then a scale-out operation is needed. The CPU resources serves as one exception to the above procedure: it would not benefit the performance if the CPU utilization limit is higher than the number of threads created for the service.}

The goal of the RL agent is, given a time duration \( t \), to determine an optimal policy \( \pi \), that results in as few SLO violations as possible \( (\min_\pi SV_i) \) while keeping the resource utilization/limit as high as possible \( (\max_\pi RU_i/RLT_i) \). Based on this objective, the reward function is defined as \( r_i = \alpha \cdot SV_i + |\check{R}_i| + (1 - \alpha) \cdot \sum_i \mid RU_i/RLT_i \mid \), where \( R \) is the set of resources.

**Transfer Learning.** Using a tailored RL agent for every microservice instead of using the shared RL agent should improve resource reprovisioning efficiency as the model would be more sensitive to application characteristics and features. However, such an approach is hard to justify in practice (deployment) because of the time required to train such tailored models for user workloads, which might have significant churn. FIRM addresses this problem of rapid model training using transfer learning in the domain of RL [12, 96, 97] where agents for SLO violation mitigation can be trained for the general case (i.e., any microservice) and the specialized case (i.e., “transferred” to the behavior of individualized microservices). This is possible as prior understanding of problem structure helps solve similar problems quickly, with the remaining task being to understand the behavior of updated microservice instances. We demonstrate the efficacy of transfer learning in our evaluation in §4. In addition to having the general case RL-agent, the FIRM framework also allows for deploying, a per-microservice RL-agent.

**Implementation Details.** We implemented the DDPG training algorithm and the actor-critic networks using PyTorch [77]. The critic net contains two fully connected hidden layers with 40 hidden units all using ReLU activation function. The actor net contains two fully connected hidden layer is using ReLU as the activation function for the first two layer, and using Tanh as the activation function for the last layer. The actor network has 8 inputs and 5 outputs, while the critic network has 23 inputs and 1 output. The actor and critic networks are shown in Fig. 8 and their inputs and outputs are listed in Table 3. We choose this setting as adding more layers and hidden units does not increase performance, instead, it slows down training speed significantly. Hyperparameters of the RL model are listed in Table 4. The latencies of each training update and inference step are 0.21 ± 0.1 ms and 40.5
FIRM’s Deployment Module, i.e., 5, verifies the actions generated by the RL agent and executes them accordingly. Each action on scaling a specific type of resource is limited by the total available amount of the resource on that physical machine. If the action leads to oversubscribing a resource, then it is replaced by a scale-out operation.

- **CPU Actions**: Actions on scaling CPU utilization are executed by modifying 
  `cpu.cfs_period_us` and 
  `cpu.cfs_quota_us` in cgroups CPU subsystem.
- **Memory Actions**: We use Intel MBA [44] and Intel CAT [43] technologies to control memory bandwidth and LLC capacity of containers, respectively. 5
- **I/O Actions**: For I/O bandwidth, we use cgroups blkio subsystem to control input/output access to disks.
- **Network Actions**: For network bandwidth, we use Hierarchy Token Bucket (HTB) [40] queuing discipline in linux Traffic Control. Egress qdiscs can be directly shaped by using HTB. Ingress qdiscs is redirected to virtual device IFB interface and then shaped by applying egress rules.

### 3.6 Performance Anomaly Injector

We accelerate the training of the RL-agent through performance anomaly injections of configurable intensity and timing. This allows us to quickly span the space of adverse resource contention behavior (i.e., the exploration-exploitation trade off in RL). This is very important as the real-world workloads might not experience all adverse situations within a short training time. We implemented a performance anomaly injector, i.e., 6, where the type of anomaly, injection time, duration, and intensity are configurable. The injector is designed to be bundled into the microservice containers as a file-system layer, the binaries incorporated into the container can then be triggered remotely during the training process. The injection campaigns (i.e., how it is used) for the injector will be discussed in §4. The injector comprises seven types of performance anomalies that can cause SLO violations which are listed in Table 5 and described below.

**Workload Variation.** We use wrk2 as the workload generator. It performs the multithreaded, multiconnection HTTP request generation to simulate client-microservice interaction. The request arrival rate and distribution can be adjusted to break the predefined SLO.

**Network Delay.** We use tc to delay network packets. Given the mean and standard deviation of the delay, each network packet is delayed following a normal distribution.

**CPU Utilization.** We implement the CPU stressor based on iBench and stress-ng to exhaust a specified level of CPU utilization on a set of cores.

**LLC Bandwidth & Capacity.** We use iBench and pmbw to inject interference on Last Level Cache (LLC). For bandwidth, the injector performs streaming accesses where the size is tuned to the parameters of the LLC. For capacity, it adjusts intensity based on the size and associativity of the LLC to issue random accesses that cover the LLC capacity.

**Memory Bandwidth.** We use iBench and pmbw for generating memory bandwidth contention. It performs serial memory accesses of configurable intensity to a small fraction of the address space. Accesses occur in a relatively small fraction of memory in order to decouple the effects of contention in memory bandwidth from contention in memory capacity.

**I/O Bandwidth.** We use Sysbench to implement the file I/O workload generator. It adjusts the number of threads, read/write ratio, and sleeping/working ratio to meet a specified level of I/O bandwidth. We also use Trickle for limiting the upload/download rate of a specific microservice instance.

**Network Bandwidth.** We use tc to limit egress network bandwidth. For ingress network bandwidth, an ifb interface is set up, and inbound traffic is directed through that. In this way, the inbound traffic becomes egress on the ifb interface, so same rules can be applied.

### 4 Evaluation

#### 4.1 Experimental Setup

**Benchmarks Applications.** We evaluate FIRM on a set of end-to-end interactive and responsive real-world microservice benchmarks: (i) DeathStarBench [30] consisting of Social Network, Media Service, and Hotel Reservation microservice applications, and (ii) TrainTicket [118] benchmark consisting of Train-Ticket Booking Service. Social Network implements a broadcast-style social network with unidirectional follow relationships where users can publish, read, and react to posts. Media Service provides functionalities such as reviewing, rating, renting, and streaming movies. Hotel Reservation is an online hotel reservation site for browsing hotel information and making reservations. Train-Ticket Booking Service pro-

---

**Table 4:** RL training parameters.

| Parameter                  | Value       |
|----------------------------|-------------|
| # Time Steps × # Minibatch | 300 × 64    |
| Size of Replay Buffer      | 10³         |
| Learning Rate              | Act(3 × 10⁻⁴), Critic(3 × 10⁻³) |
| Discount Factor            | 0.9         |

± 4 ms respectively.

**Table 5:** Types of performance anomalies injected to microservices causing SLO violations.

| Performance Anomaly Types          | Tools/Benchmarks |
|------------------------------------|------------------|
| Workload Variation                 | wrk2 [113]       |
| Network Delay                      | tc [98]          |
| CPU Utilization                    | iBench [20], stress-ng [92] |
| LLC Bandwidth & Capacity           | iBench, pmbw [74] |
| Memory Bandwidth                   | iBench [20], pmbw [74] |
| I/O Bandwidth                      | Sysbench [94]    |
| Network Bandwidth                  | tc [98], Trickle [108] |

---

7Our evaluation on IBM Power systems in §4 did not use these actions due to lack of hardware support. OS support or software partitioning mechanism [56, 79] can be applied, which we leave to future work.
vides typical train-ticket booking functionalities such as ticket enquiry, reservation, payment, change, and user notification. These benchmarks contain 36, 38, 15, and 41 unique microservices, respectively, cover all workflow patterns (recall from §3.2), and use various programming languages: Java, Python, Node.js, Go, C/C++, Scala, PHP, and Ruby. All microservices are deployed in separate Docker containers.

**System Setup.** We validate our design by implementing a prototype of FIRM using Kubernetes [16] as the underlying container orchestration framework. We deploy FIRM on a cluster of 15 two-socket physical nodes. Each server consists of 56–192 CPU cores and RAM varying from 500GB–1000GB. Nine of the servers use Intel Xeon E5s and E7s, while the remaining use IBM ppc64 Power8 and Power9. All machines run Ubuntu 18.04.3 LTS. The four microservice benchmarks are deployed and orchestrated using Kubernetes.

**Load Generation.** We continuously drive these services with various open-loop workload generators [113] to represent an active production environment, which include constant, diurnal, exponential distribution, and load with spikes. We uniformly generate workloads for every request type across all microservice benchmarks. The parameters to workload generators are the same as DeathStarBench. The workload generators and the microservice benchmark applications are never co-located (i.e., they execute on different nodes in the cluster). To control the variability in our experiments, we disable all other user workloads on the cluster.

**Injection and Comparison Baselines.** We use our performance anomaly injector (recall from §3.6) to inject various types of performance anomalies into containers uniformly at random. Unless further specified, (i) the anomaly injection time interval is in an exponential distribution with \( \lambda = 0.33 \) s\(^{-1}\), and (ii) the anomaly type and intensity are selected uniformly at random. We implement two baseline approaches: (a) the Kubernetes autoscaling mechanism [50] and (b) an AIMD-based method [34, 93] to manage resources for each container. Both approaches are rule-based autoscaling techniques.

### 4.2 Critical Component Localization

Here, we study the effectiveness of FIRM in identifying microservices that are most likely causing SLO violations using the techniques presented in §3.2 and §3.3.

**Single anomaly localization.** First, we evaluate how well FIRM localizes the microservice instances that are responsible for SLO violations under different types of single-anomaly injections. For each type of performance anomaly and each type of request, we gradually increase the intensity of injected resource interference and record end-to-end latency. The intensity parameter is chosen uniformly at random between [start-point, end-point], where the start-point is the intensity that triggers SLO violations, and the end-point is when either the anomaly injector consumes all possible resources or over 80% user requests are dropped. Fig. 9(a) shows the receiver operating characteristic (ROC) curve of root cause localization. The ROC curve captures the relationship between the false positive rate (x-axis) against the true positive rate (y-axis). The closer to the upper-left corner the curve is, the better the performance. We observe that the localization accuracy of FIRM, when subject to different types of anomalies, does not differ significantly. In particular, FIRM’s Extractor module achieves near 100% true positive rate, when the false positive rate is between [0.12, 0.15].

**Multi-anomaly localization.** Since there is no guarantee that only one resource contention can happen at the same time under dynamic datacenter workloads [36, 38, 89, 90]. We also study the container localization performance under multi-anomaly injections and compare machines with two different processor ISAs (x86 and ppc64). Intensity distribution of each anomaly type used in this experiment is shown in Fig. 9(c). The experiment is divided into time windows of 10s, i.e., \( T_i \) form Fig. 9(c)). At each time window, we pick injection intensity of each anomaly type uniformly at random with range [0,1]. Our observations are illustrated in Fig. 9(b). The average accuracy for localizing critical components in each application ranges from 92.8%–94.6%. The overall average localization accuracy is 93.8% across four microservice benchmarks. Overall, we observe that the accuracy of the Extractor does not differ between the two sets of processors.

### 4.3 RL Training & SLO Violation Mitigation

To understand to convergence behavior of FIRM’s RL agent, we train three RL models subjected to the same sequence of performance anomaly injections (described in §4.1). These three RL models are: (i) a common RL agent for all microser-
We observe that AIMD-based method, albeit simple, outperforms the Kubernetes autoscaling approach; but after 2500 iterations, both agents are better than Kubernetes autoscaling and the AIMD-based method. Upon convergence, FIRM with single-RL agent (one-for-all) and multi-RL agent (one-for-each) improves with episodes in terms of the SLO violation mitigation time; the starting policy at iteration 0-900 is no better than the Kubernetes autoscaling approach; but after 2500 iterations, both agents are better than Kubernetes autoscaling and the AIMD-based method. Upon convergence, FIRM with single-RL agent achieves average mitigation time of 1.7×, which outperforms the AIMD-based method by up to 9.6× and Kubernetes autoscaling by up to 30.1× in terms of the time to mitigate SLO violations.

### 4.4 End-to-End Performance

Here, we show the end-to-end performance of FIRM and its generalization by further evaluating on DeathStarBench benchmarks based on the hyperparameter tuned during training Train-Ticket benchmark. To understand the benefit of 10-30× improvement demonstrated above, we measure 99th percentile end-to-end latency when the microservices are managed by the two baseline approaches and by FIRM. Fig. 10(a) shows the cumulative distribution of the end-to-end latency. We observe that AIMD-based method, albeit simple, outperforms the Kubernetes autoscaling approach by 1.7× on average and by 1.6× in the worst cases. In contrast, FIRM:

1. outperforms both baselines by up to 6.9× and 11.5×, which leads to 9.8× and 16.7× fewer SLO violations;
2. lowers the overall requested CPU limit by 29.1-62.3%, shown in Fig. 10(b), and increases average cluster-level CPU utilization by up to 33%;
3. reduces user request drops by up to 8.6× in Fig. 10(c); and
4. multi-RL (one-for-each) model and single-RL (one-for-all) model in FIRM perform equally in terms of reducing end-to-end performance variability and requested resources.

This is because FIRM detects SLO violations accurately and addresses resource contention before SLO violations can propagate. By interacting with dynamic microservice environments under complicated loads and resource allocation scenarios, FIRM’s RL agent dynamically learns the policy, and hence outperform heuristics-based approaches.

---

6 1000 iterations correspond to roughly 30 minutes.
As shown in Table 6, the operations to scale resources for which automatically explores the action space to generate without overprovisioning. It is difficult to model the dependence between low-level resource requirements to quantifiable performance gain, all while dealing with uncertain and noisy measurements [71, 110]. FIRM addresses the issue by modeling that dependence in an RL-based feedback loop, which automatically explores the action space to generate optimal policies without human intervention.

### Why Multi-level ML Framework?
Modelling states of all microservices and feeding it as an input to a single large ML model [75, 116] lead to (i) state-action space expansion issues which grows with the number of microservices, thus increasing the training time; and (ii) dependence between microservice architecture and the ML-model, thus sacrificing the generality. FIRM addresses these problems by incorporating a two-level framework. The first ML model filters the microservice instances responsible for SLO violations using SVM, thereby reducing the number of microservices that needs to be considered for mitigating SLO violations. This enables the second ML model, the RL agent, to be trained faster and removes dependence on the application architecture (which helps avoid RL model reconstruction/retraining).

### Lower Bound on SLO Violation Duration for FIRM
As shown in Table 6, the operations to scale resources for microservice instances take 2.1–45.7 ms. Thus, this is the minimum duration of latency spikes that any RM approach can handle. For transient SLO violations, which are smaller than the minimum duration, the action will always miss the mitigation deadline and can potentially harm overall system performance. Predicting the spikes before they happen, and proactively taking mitigation actions can be a solution. However, this is a difficult problem as microservices are dynamically evolving, both in terms of load and architectural design. This will be subject of our future work.

### 5 Discussion

#### Necessity & Challenges of Modelling Low-level Resources
Recall from §2 that modeling resources at a fine granularity is necessary, as it allows for better performance without overprovisioning. It is difficult to model the dependence between low-level resource requirements to quantifiable performance gain, all while dealing with uncertain and noisy measurements [71, 110]. FIRM addresses the issue by modeling that dependence in an RL-based feedback loop, which automatically explores the action space to generate optimal policies without human intervention.

#### Why Multi-level ML Framework?
Modelling states of all microservices and feeding it as an input to a single large ML model [75, 116] lead to (i) state-action space expansion issues which grows with the number of microservices, thus increasing the training time; and (ii) dependence between microservice architecture and the ML-model, thus sacrificing the generality. FIRM addresses these problems by incorporating a two-level framework. The first ML model filters the microservice instances responsible for SLO violations using SVM, thereby reducing the number of microservices that needs to be considered for mitigating SLO violations. This enables the second ML model, the RL agent, to be trained faster and removes dependence on the application architecture (which helps avoid RL model reconstruction/retraining).

#### Lower Bound on SLO Violation Duration for FIRM
As shown in Table 6, the operations to scale resources for microservice instances take 2.1–45.7 ms. Thus, this is the minimum duration of latency spikes that any RM approach can handle. For transient SLO violations, which are smaller than the minimum duration, the action will always miss the mitigation deadline and can potentially harm overall system performance. Predicting the spikes before they happen, and proactively taking mitigation actions can be a solution. However, this is a difficult problem as microservices are dynamically evolving, both in terms of load and architectural design. This will be subject of our future work.

### 6 Related Work

SLO violations in cloud applications and microservices is a popular and well researched topic. We categorize prior work into two buckets: root cause analyzers and autoscalers. Both rely heavily on the collection of tracing and telemetry data.

#### Tracing and Probing for Microservices
Tracing for large-scale microservices (essentially distributed systems) helps understand the path of a request as it propagates through the components of a distributed system. Tracing requires either application-level instrumentation [14, 28, 88, 102–106] or middleware/OS-level instrumentation [8, 13, 59, 100].

#### Root Cause Analysis
A large body of work [13, 31, 45, 47, 57, 59, 86, 100, 111, 114] provides promising examples that data-driven diagnostics help detect performance anomalies and analyze root causes. For example, Sieve [100] leverages Granger causality to correlate performance anomaly data series with particular metrics as potential root causes. Microscope [57] and MicroRCA [114] are both designed to identify abnormal services by constructing service causal graphs that model anomaly propagation and by inferring causes using graph traversal or ranking algorithms [46]. Seer [31] leverages deep learning to learn spatial and temporal patterns that translate to SLO violations. However, none of these approaches addresses the dynamic nature of microservice environments (i.e., frequent microservice updates and deployment changes), which require costly model reconstruction or retraining.

#### Autoscaling Cloud Applications
Current techniques for autoscaling cloud applications can be categorized into four groups [61, 78]: (1) rule-based (commonly offered by cloud providers [4, 5, 33]), (2) time series analysis (regression on resource utilization, performance, and workloads), (3) model-based (e.g., queuing networks), or (4) RL-based. Some approaches combine several techniques. For instance, Auto-pilot [81] combines time series analysis and RL algorithms to scale number of containers and associated CPU/RAM. Unfortunately, when applied to microservices with large scale and complex dependencies, scaling of each microservice instance independently results in suboptimal solutions, and it is difficult to define sub-SLOs for individual instances. Approaches on autoscaling microservices or distributed dataflows [35, 51, 75, 116, 117] make scaling decisions for number of replicas and/or container size without considering low-level shared-resource interference. ATOM [35] and Microscaler [117] achieve this using a combination of queuing network- and heuristic-based approximations. ASFM [75] uses recurrent neural network activity to predict workloads and translates to resources using linear regression. Streaming and data-processing scalers like DS2 [51] and MIRAS [116] leverage explicit application-level modelling and apply RL to represent resource-performance mapping of operators and their dependencies.

#### Orchestration
In this paper, we do not address the problem of scheduling and orchestrating resources. There are several tools, e.g., Borg [109], Mesos [39], Tarcil [24], Paragon [21], Quasar [22], Morpheus [49], DeepDive [68], and Q-clouds [66], that provides such functionality. FIRM can work in conjunction with these resource orchestration tools to reduce SLO violations.

### 7 Conclusion

We propose **FIRM**, an ML-based, fine-grained, resource management framework that addresses SLO violations and resource under-utilization in microservices. FIRM uses a two-level ML model, one for identifying microservices responsible...
for SLO violations, and the other for mitigation. The combined ML model reduces SLO violations up to $16.7 \times$ while reducing overall CPU limit by up to 62.3%. Overall, FIRM enables fast mitigation of SLOs by using efficient resource provisioning, which benefits both cloud service providers and microservice owners.

References

[1] Ivo Adan and Jacques Resing. Queueing theory, 2002.

[2] Younsun Ahn, Jieun Choi, Sol Jeong, and Yoonhee Kim. Auto-scaling method in hybrid cloud for scientific applications. In *The 16th Asia-Pacific Network Operations and Management Symposium*, pages 1–4. IEEE, 2014.

[3] Ioannis Arapakis, Xiao Bai, and B Barla Cambazoglu. Impact of response latency on user behavior in web search. In *Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval*, pages 103–112, 2014.

[4] AWS auto scaling documentation. https://docs.aws.amazon.com/autoscaling/index.html, Accessed 2020/01/23.

[5] Azure autoscale. https://azure.microsoft.com/en-us/features/autoscale/, Accessed 2020/01/23.

[6] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Migrating to cloud-native architectures using microservices: an experience report. In *European Conference on Service-Oriented and Cloud Computing*, pages 201–215. Springer, 2015.

[7] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. Microservices architecture enables DevOps: Migration to a cloud-native architecture. *IEEE Software*, 33(3):42–52, 2016.

[8] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. Magpie: Online modelling and performance-aware systems. In *HotOS*, pages 85–90, 2003.

[9] Luiz André Barroso and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 4(1):1–108, 2009.

[10] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise reduction in speech processing*, pages 1–4. Springer, 2009.

[11] cadvisor. https://github.com/google/cadvisor, Accessed 2020/01/23.
quality in large shared clusters. In Proceedings of the Sixth ACM Symposium on Cloud Computing, pages 97–110, 2015.

[25] Christopher P Diehl and Gert Cauwenberghs. Svm incremental learning, adaptation and optimization. In Proceedings of the International Joint Conference on Neural Networks, 2003., volume 4, pages 2685–2690. IEEE, 2003.

[26] Jianru Ding, Ruiqi Cao, Indrajeet Saravanan, Nathaniel Morris, and Christopher Stewart. Characterizing service level objectives for cloud services: Realities and myths. In 2019 IEEE International Conference on Autonomic Computing (ICAC), pages 200–206. IEEE, 2019.

[27] Rob Eisinga, Manfred Te Grotenhuis, and Ben Pelzer. The reliability of a two-item scale: Pearson, Cronbach, or Spearman-Brown? International journal of public health, 58(4):637–642, 2013.

[28] Rodrigo Fonseca, George Porter, Randy H Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07), 2007.

[29] Yu Gan and Christina Delimitrou. The architectural implications of cloud microservices. IEEE Computer Architecture Letters, 17(2):155–158, 2018.

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

[31] Yu Gan, Yanqi Zhang, Kelvin Hu, Daijun 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, pages 19–33, 2019.

[32] Mrittika Ganguli, Rajneesh Bhardwaj, Ananth Sankaranarayanan, Sunil Raghavan, Subramony Seshia, Gilbert Hyatt, Muralidharan Sundararajan, Arkadiusz Chylinski, and Alok Prakash. Cpu overprovisioning and cloud compute workload scheduling mechanism, March 20 2018. US Patent 9,921,866.

[33] Google cloud load balancing and scaling. https://cloud.google.com/compute/docs/load-balancing-and-autoscaling, Accessed 2020/01/23.

[34] Panos Gevros and Jon Crowcroft. Distributed resource management with heterogeneous linear controls. Computer Networks, 45(6):835–858, 2004.

[35] Alim Ul Gias, Giuliano Casale, and Murray Woodside. ATOM: Model-driven autoscaling for microservices. In 2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS), pages 1994–2004. IEEE, 2019.

[36] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In 2007 IEEE 10th International Symposium on Workload Characterization, pages 171–180. IEEE, 2007.

[37] Ivo Grondman, Lucian Busoniu, Gabriel AD Lopes, and Robert Babuska. A survey of actor-critic reinforcement learning: Standard and natural policy gradients. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42(6):1291–1307, 2012.

[38] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. Applied machine learning at facebook: A datacenter infrastructure perspective. In 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA), pages 620–629. IEEE, 2018.

[39] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In NSDI, volume 11, pages 22–22, 2011.

[40] HTB - Hierarchy Token Bucket. https://linux.die.net/man/8/tc-htb, Accessed 2020/01/23.

[41] Steven Ihde and Karan Parikh. From a monolith to microservices + REST: the evolution of LinkedIn’s service architecture, March 2015. https://www.infoq.com/presentations/linkedin-microservices-urn/, Accessed 2020/01/23.

[42] Alexey Ilyushkin, Ahmed Ali-Eldin, Nikolas Herbst, Alessandro V Papadopoulos, Bogdan Ghit, Dick Epema, and Alexandru Iosup. An experimental performance evaluation of autoscaling policies for complex workflows. In Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering, pages 75–86, 2017.
[43] Intel cache allocation technology. https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

[44] Intel memory bandwidth allocation. https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

[45] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. Performance monitoring and root cause analysis for cloud-hosted web applications. In Proceedings of the 26th International Conference on World Wide Web, pages 469–478, 2017.

[46] Glen Jeh and Jennifer Widom. Scaling personalized web search. In Proceedings of the 12th international conference on World Wide Web, pages 271–279, 2003.

[47] Saurabh Jha, Shengkun Cui, Subho Banerjee, Tianyin Xu, Jeremy Enos, Mike Shoverman, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Understanding, detecting, and localizing failures in high-performance storage systems. In Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis, 2020.

[48] Anshul Jindal, Vladimir Podolskiy, and Michael Gerndt. Performance modeling for cloud microservice applications. In Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering, pages 25–32, 2019.

[49] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayananurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, et al. Morpheus: Towards automated SLOs for enterprise clusters. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 117–134, 2016.

[50] Autoscaling in kubernetes. https://kubernetes.io/blog/2016/07/autoscaling-in-kubernetes/, Accessed 2020/01/23.

[51] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 783–798, 2018.

[52] Pavel Laskov, Christian Gehl, Stefan Krüger, and Klaus-Robert Müller. Incremental support vector learning: Analysis, implementation and applications. Journal of machine learning research, 7(Sep):1909–1936, 2006.

[53] Latency is everywhere and it costs you sales - How to crush it, July 2009. http://highscalability.com/latency-everywhere-and-it-costs-you-sales-how-crush-it, Accessed 2020/01/23.

[54] It’s official: Google now counts site speed as a ranking factor, April 2010. https://searchengineland.com/google-now-counts-site-speed-as-ranking-factor-39708, Accessed 2020/01/23.

[55] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. arXiv preprint arXiv:1509.02971, 2015.

[56] Jiang Lin, Qingda Lu, Xiaoning Ding, Zhao Zhang, Xiaodong Zhang, and P Sadyayappan. Gaining insights into multicore cache partitioning: Bridging the gap between simulation and real systems. In 2008 IEEE 14th International Symposium on High Performance Computer Architecture, pages 367–378. IEEE, 2008.

[57] JinJin Lin, Pengfei Chen, and Zibin Zheng. Microscope: Pinpoint performance issues with causal graphs in micro-service environments. In International Conference on Service-Oriented Computing, pages 3–20. Springer, 2018.

[58] Richard Harold Lindeman. Introduction to bivariate and multivariate analysis. Technical report, Scott Foresman & Co, 1980.

[59] Haifeng Liu, Jinjun Zhang, Huasong Shan, Min Li, Yuan Chen, Xiaofeng He, and Xiaowei Li. Jcallgraph: Tracing microservices in very large scale container cloud platforms. In International Conference on Cloud Computing, pages 287–302. Springer, 2019.

[60] Keith Gerald Lockyer. Introduction to Critical Path Analysis. Pitman, 1969.

[61] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. A review of auto-scaling techniques for elastic applications in cloud environments. Journal of grid computing, 12(4):559–592, 2014.

[62] Michael David Marr and Matthew D Klein. Automated profiling of resource usage, April 26 2016. US Patent 9,323,577.

[63] Jason Mars and Lingjia Tang. Whare-map: heterogeneity in" homogeneous" warehouse-scale computers. In Proceedings of the 40th Annual International Symposium on Computer Architecture, pages 619–630, 2013.
[64] Tony Mauro. Adopting microservices at Netflix: Lessons for architectural design, February 2015. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, Accessed 2020/01/23.

[65] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. Linux journal, 2014(239):2, 2014.

[66] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-Clouds: managing performance interference effects for QoS-aware clouds. In Proceedings of the 5th European conference on Computer systems, pages 237–250, 2010.

[67] Neo4j - Native Graph Database. https://github.com/neo4j/neo4j, Accessed 2020/01/23.

[68] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13), pages 219–230, 2013.

[69] NumPy. https://numpy.org/doc/stable/index.html, Accessed 2020/01/23.

[70] OpenTracing. https://opentracing.io/, Accessed 2020/01/23.

[71] Karl Ott and Rabi Mahapatra. Hardware performance counters for embedded software anomaly detection. In 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), pages 528–535. IEEE, 2018.

[72] Dan Paik. Adapt or Die: A microservices story at Google, December 2016. https://www.slideshare.net/apigee/adapt-or-die-a-microservices-story-at-google, Accessed 2020/01/23.

[73] perf. http://man7.org/linux/man-pages/man1/perf.1.html, Accessed 2020/01/23.

[74] pmbw - Parallel Memory Bandwidth Benchmark. https://panthema.net/2013/pmbw/, Accessed 2020/01/23.

[75] Issaret Prachitmutita, Wachirawit Aittinonmongkol, Nasoret Pojjanasuksakul, Montri Supattatham, and Praisan Padungweang. Auto-scaling microservices on IaaS under SLA with cost-effective framework. In 2018 Tenth International Conference on Advanced Computational Intelligence (ICACI), pages 583–588. IEEE, 2018.

[76] The Prometheus monitoring system and time series database. https://github.com/prometheus/prometheus, Accessed 2020/01/23.

[77] PyTorch. https://pytorch.org/, Accessed 2020/01/23.

[78] Chenhao Qu, Rodrigo N Calheiros, and Rajkumar Buyya. Auto-scaling web applications in clouds: A taxonomy and survey. ACM Computing Surveys (CSUR), 51(4):1–33, 2018.

[79] Nauman Rafique, Won-Taek Lim, and Mithuna Thotethodi. Architectural support for operating system-driven cmp cache management. In Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, PACT ’06, page 2–12, New York, NY, USA, 2006. Association for Computing Machinery.

[80] Charles Reiss, Alexey Tumanov, Gregory R Ganger, Randy H Katz, and Michael A Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In Proceedings of the Third ACM Symposium on Cloud Computing (SoCC 12), pages 1–13, 2012.

[81] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmirek, 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.

[82] Cristian Satnic. Amazon, Microservices and the birth of AWS cloud computing, April 2016. https://www.linkedin.com/pulse/amazon-microservices-birth-aws-cloud-computing-cristian-satnic/, Accessed 2020/01/23.

[83] 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.

[84] scikit-learn. https://scikit-learn.org/stable/, Accessed 2020/01/23.

[85] S Senthil Kumaran. Practical LXC and LXD: linux containers for virtualization and orchestration. Springer, 2017.
[86] Syed Yousaf Shah, Xuan-Hong Dang, and Petros Zerfos. Root cause detection using dynamic dependency graphs from time series data. In 2018 IEEE International Conference on Big Data (Big Data), pages 1998–2003. IEEE, 2018.

[87] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In 2011 31st International Conference on Distributed Computing Systems, pages 559–570. IEEE, 2011.

[88] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[89] Akshitha Sriraman and Abhishek Dhanotia. Accelometer: Understanding acceleration opportunities for data center overheads at hyperscale. In Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, pages 733–750, 2020.

[90] Akshitha Sriraman and Thomas F Wenisch. µsuite: a benchmark suite for microservices. In 2018 IEEE International Symposium on Workload Characterization (IISWC), pages 1–12. IEEE, 2018.

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

[92] stressng. https://wiki.ubuntu.com/Kernel/Reference/stress-ng, Accessed 2020/01/23.

[93] Sonja Stüdli, M Corless, Richard H Middleton, and Robert Shorten. On the modified AIMD algorithm for distributed resource management with saturation of each user’s share. In 2015 54th IEEE Conference on Decision and Control (CDC), pages 1631–1636. IEEE, 2015.

[94] Sysbench. https://github.com/akopytov/sysbench, Accessed 2020/01/23.

[95] Matthew E Taylor and Peter Stone. Autonomous transfer for reinforcement learning. In AAMAS (1), pages 283–290. Citeseer, 2008.

[97] Matthew E Taylor and Peter Stone. Transfer learning for reinforcement learning domains: A survey. Journal of Machine Learning Research, 10:1633–1685, Jul 2009.

[98] tc - Traffic Control in the Linux kernel. https://linux.die.net/man/8/tc, Accessed 2020/01/23.

[99] Jörg Thalheim, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. Cntr: Lightweight OS containers. In 2018 USENIX Annual Technical Conference (USENIX ATC ’18), pages 199–212, 2018.

[100] Jörg Thalheim, Antonio Rodrigues, Istemi Ekin Akkus, Pramod Bhatotia, Ruichuan Chen, Bimal Viswanath, Lei Jiao, and Christof Fetzer. Sieve: actionable insights from monitored metrics in distributed systems. In Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference, pages 14–27, 2017.

[101] Scott Tonidandel and James M LeBreton. Relative importance analysis: A useful supplement to regression analysis. Journal of Business and Psychology, 26(1):1–9, 2011.

[102] Instana. https://docs.instana.io/, Accessed 2020/01/23.

[103] Jaeger: open source, end-to-end distributed tracing. https://jaegertracing.io/, Accessed 2020/01/23.

[104] Lightstep distributed tracing. https://lightstep.com/distributed-tracing/, Accessed 2020/01/23.

[105] SkyWalking, An application performance monitoring system. https://github.com/apache/skywalking, Accessed 2020/01/23.

[106] OpenZipkin - A distributed tracing system. https://zipkin.io/, Accessed 2020/01/23.

[107] Train Ticket - A train-ticket booking system based on microservice architecture. https://github.com/FudanSELab/train-ticket, Accessed 2020/01/23.

[108] Trickle - a lightweight userspace bandwidth shaper. https://linux.die.net/man/1/trickle, Accessed 2020/01/23.

[109] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In Proceedings of the European Conference on Computer Systems (EuroSys), Bordeaux, France, 2015.
[110] Xueyang Wang, Sek Chai, Michael Isnardi, Sehoon Lim, and Ramesh Karri. Hardware performance counter-based malware identification and detection with adaptive compressive sensing. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.

[111] Jianping Weng, Jessie Hui Wang, Jiahai Yang, and Yang Yang. Root cause analysis of anomalies of multilayer services in public clouds. *IEEE/ACM Transactions on Networking*, 26(4):1646–1659, 2018.

[112] Scott White and Padhraic Smyth. Algorithms for estimating relative importance in networks. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 266–275, 2003.

[113] wrk2 - An HTTP benchmarking tool based mostly on wrk. Retrieved from <https://github.com/giltene/wrk2>, Accessed 2020/01/23.

[114] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. MicroRCA: Root cause localization of performance issues in microservices. In *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2020.

[115] C Yang and Barton Miller. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed programs*, pages 366–367, 1988.

[116] Zhe Yang, Phuong Nguyen, Haiming Jin, and Klara Nahrstedt. Miras: Model-based reinforcement learning for microservice resource allocation over scientific workflows. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 122–132. IEEE, 2019.

[117] Guangba Yu, Pengfei Chen, and Zibin Zheng. Microscaler: Automatic scaling for microservices with an online learning approach. In *2019 IEEE International Conference on Web Services (ICWS)*, pages 68–75. IEEE, 2019.

[118] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 2018.