FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices

Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer, *University of Illinois at Urbana–Champaign*

https://www.usenix.org/conference/osdi20/presentation/qiu

This paper is included in the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation

November 4–6, 2020

978-1-939133-19-9

Open access to the Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX

FIRM: An Intelligent Fine-Grained Resource Management Framework for SLO-Oriented Microservices

Haoran Qiu[1] Subho S. Banerjee[1] Saurabh Jha[1] Zbigniew T. Kalbarczyk[2]
Ravishankar K. Iyer[1,2]

[1]*Department of Computer Science* [2]*Department of Electrical and Computer Engineering*
University of Illinois at Urbana-Champaign

Abstract

User-facing latency-sensitive web services include numerous distributed, intercommunicating microservices that promise to simplify software development and operation. However, multiplexing of 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 via dynamic reprovisioning. Experiments across four microservice benchmarks demonstrate that FIRM reduces SLO violations by up to 16× while reducing the overall requested CPU limit by up to 62%. Moreover, FIRM improves performance predictability by reducing tail latencies by up to 11×.

1 Introduction

User-facing latency-sensitive web services, like those at Netflix [68], Google [77], and Amazon [89], are increasingly built as microservices that execute 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 times to process requests, i.e., latency spikes [4, 11, 22, 30, 35, 44, 53, 69, 98, 99]. Thus, it is critical to efficiently multiplex shared resources among microservices to reduce SLO violations.

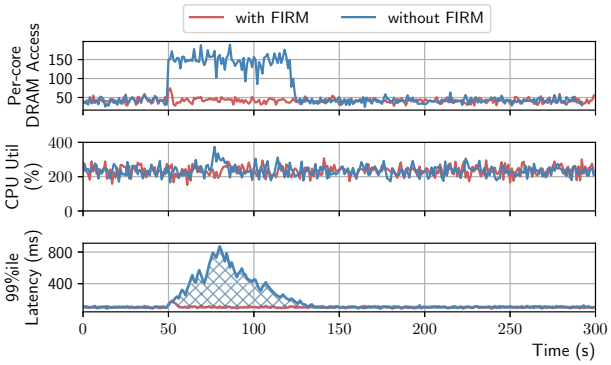


Figure 1: Latency spikes on microservices due to low-level resource contention.

Traditional approaches (e.g., overprovisioning [36, 87], recurrent provisioning [54, 66], and autoscaling [39, 56, 65, 81, 84, 88, 127]) 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, such as caches, memory, I/O channels, and network links, at fine granularity, and thus may not reduce SLO violations. For example, in Fig. 1, the Kubernetes container-orchestration system [20] is unable to reduce the tail latency spikes arising from contention for a shared resource like memory bandwidth, as its autoscaling algorithms were built using heuristics that only monitor CPU utilization, which does not change much during the latency spike. Second, significant human-effort and training are needed to build high-fidelity performance models (and related scheduling heuristics) of large-scale microservice deployments (e.g., queuing systems [27, 39]) that can capture low-level resource contention. 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 prob-

lems 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. Online 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 on whether or not a microservice instance is responsible for SLO violations.
- *Reinforcement learning (RL) driven mitigation of SLO violations 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, (b) scaling up/down, i.e., adding or reducing the amount of resources attached to a container, and (c) scaling out/in, i.e., scaling the number of replicas for services. 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 [67] and workloads [40,42,96,98] 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 an SLO violation mitigation framework for microservices by using fine-grained resource management in an application-architecture-agnostic way with multilevel 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 the microservice instances responsible for SLO violations by extracting critical paths and detecting anomaly instances in near-real time using telemetry data.
2. *RL-based SLO Violation 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 by 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 [20]. We demonstrate and validate this implementation on four real-world microservice benchmarks [34, 116] (in §4).

Results. FIRM significantly outperforms state-of-the-art RM frameworks like Kubernetes autoscaling [20, 55] and additive increase multiplicative decrease (AIMD) based methods [38, 101].

- It reduces overall SLO violations by up to 16× compared with Kubernetes autoscaling, and 9× compared with the AIMD-based method, while reducing the overall requested CPU by as much as 62%.
- It outperforms the AIMD-based method by up to 9× and Kubernetes autoscaling by up to 30× in terms of the time to mitigate SLO violations.
- It improves overall performance predictability by reducing the average tail latencies up to 11×.
- It successfully localizes SLO violation root-cause microservice instances with 93% accuracy on average.

FIRM mitigates SLO violations without overprovisioning because of two main features. First, it models the dependency between low-level resources and application performance in an RL-based feedback loop to deal with uncertainty and noisy measurements. Second, it takes a two-level approach in which the online critical path analysis and the SVM model filter only those microservices that need to be considered to mitigate 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 made possible by the rise of containerization [21, 70, 92, 108] and container-orchestration frameworks [19, 20, 90, 119] that enable modular, low-overhead, low-cost, elastic, and high-efficiency development and production deployment of SOA microservices [8,9,33,34,46,68,77,89,104]. A deployment of

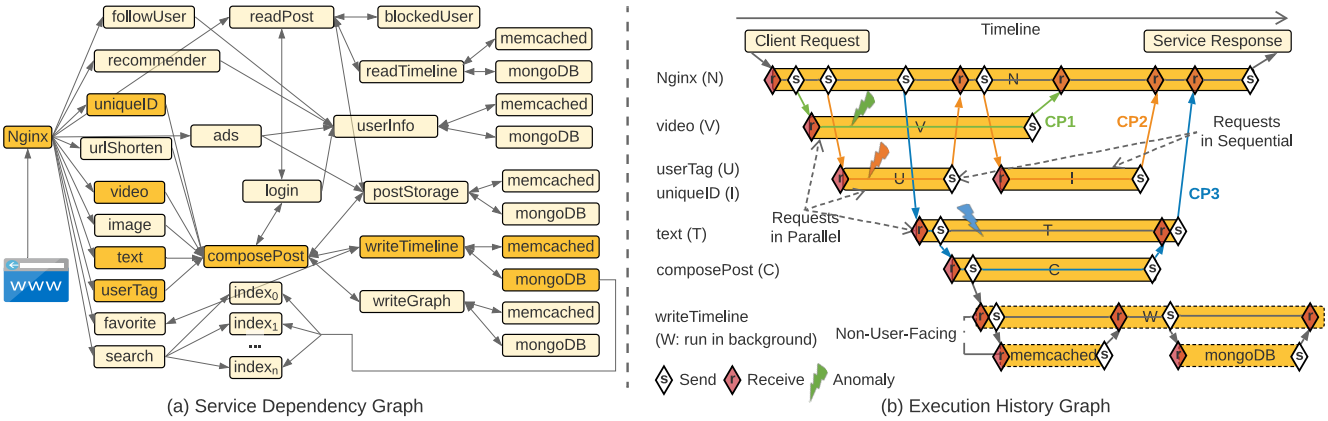
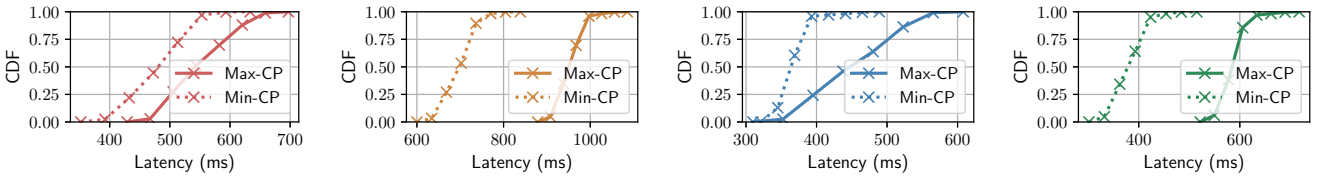


Figure 2: Microservices overview: (a) Service dependency graph of *Social Network* from the DeathStarBench [34] benchmark; (b) Execution history graph of a `post-compose` request in the same microservice.



(a) Social network service. **(b)** Media service. **(c)** Hotel reservation service. **(d)** Train-ticket booking service.

Figure 3: Distributions of end-to-end latencies of different microservices in the DeathStarBench [34] and Train-Ticket [116] benchmarks. Dashed and solid lines correspond to the minimum and maximum critical path latencies on serving a request.

such microservices can be visualized as a *service dependency graph* or an *execution history graph*. The performance of a user request, i.e., its end-to-end latency, is determined by the *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). It tells how requests are flowing among microservices by following parent-child relationship chains. Fig. 2(a) shows the service dependency graph of the *Social Network* microservice benchmark [34]. 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 darker yellow.

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) shows 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 [64, 125]. When we mention CP alone without the target microservice m, it means the critical path of the "Service Response" to the client (see Fig. 2(b)), i.e., end-to-end latency.

To understand SLO violation characteristics and study the relationship between runtime performance and the underlying resource contention, we have run extensive performance anomaly injection experiments on widely used microservice benchmarks (i.e. DeathStarBench [34] and Train-Ticket [116]) and collected around 2 TB of raw tracing data (over 4.1×10^7 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 because of underlying shared-resource contention and their sensitivity to this interference. Though other causes may also lead to CP evolution at real-time (e.g., distributed rate limiting [86], and cacheability of requested data [2]), it can still be used as an efficient manifestation of resource 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 by using *performance*

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

Case	Average Individual Latency (*ms*)						Total (*ms*)
	N	*V*	*U*	*I*	*T*	*C*	
<*V*,*CP*1>	13	603	166	33	71	68	614 ± 106
<*U*,*CP*2>	14	237	537	39	62	89	580 ± 113
<*T*,*CP*3>	13	243	180	35	414	80	507 ± 75

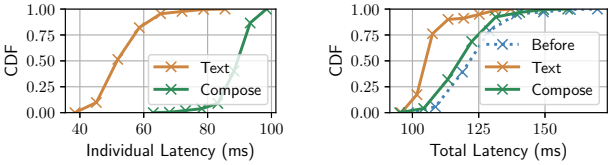


Figure 4: Improvement of end-to-end latency by scaling "highest-variance" and "highest-median" microservices.

anomaly injections.[1] Table 1 lists the changes observed in the latencies of individual microservices, as well as end-to-end latency. We observe as much as 1.2–2× variation in end-to-end latency 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× difference in median latency and 2.5× difference in 99th percentile tail latency across these CPs.

Recent approaches (e.g., [3,47]) 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 CP latency. 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 violations 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 the culprit. A common heuristic is to pick the one with the highest latency. However, we find that that rarely leads to the optimal solution. Consider Fig. 4. The left side shows the CDF of the latencies of two services (i.e., `composePost` and `text`) on the CP of the `post-compose` request in the Social Network benchmark. The `composePost` service has a higher median/mean latency while the `text` service has a higher variance. Now, although the `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. That phenomenon is shown on the right side of Fig. 4, which shows the end-to-end latency for the original configuration (labeled "Before") and after the two microservices were scaled from a single to two containers each (labeled "Text" and "Compose"). Hence, scaling microservices with higher variances provides better performance gain.

[1] *Performance anomaly injections* (§3.6) are used to trigger SLO violations by generating fine-grained resource contention with configurable resource types, intensity, duration, timing, and patterns, which helps with both our characterization (§2) and ML model training (§3.4).

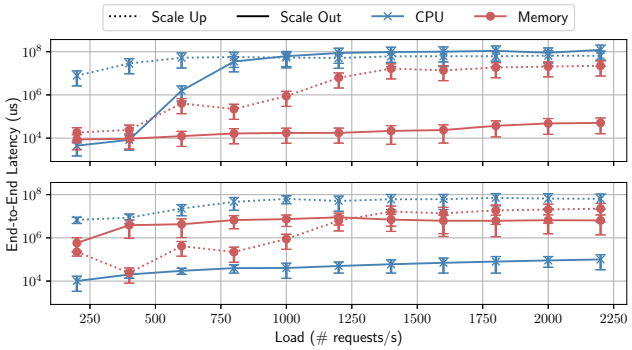


Figure 5: Dynamic behavior of mitigation strategies: *Social Network* (top); *Train-Ticket Booking* (bottom). Error bars show 95% confidence intervals on median latencies.

Insight 3: Mitigation Policies Vary with User Load and 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 (a) to *scale out* by spinning up a new instance of the container on another node of the compute cluster, or (b) to *scale up* by providing more resources to the container via either explicitly partitioning resources (e.g., in the case of memory bandwidth or last-level cache) or granting more resources to an already deployed container of the microservice (e.g., in the case of CPU cores).

As described before, recent approaches [23,38,39,56,65,84,94,101,127]) address the problem by building static policies (e.g., AIMD for controlling resource limits [38, 101], and rule/heuristics-based scaling relying on profiling of historic data about a workload [23, 94]), or modeling performance [39,56]. However, we found in our experiments with the four microservice benchmarks that such static policies are not well-suited for dealing with latency-critical workloads because the optimal policy must incorporate dynamic contextual information. That is, information about the type of user requests, and load (in requests per second), as well as the critical resource bottlenecks (i.e, the resource being contended for), must be jointly analyzed to make optimal decisions. For example, in Fig. 5 (top), we observe that the trade-off between

scale-up and scale-out changes based not only 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 also application-dependent because the trade-off curve inflection points change across applications, as illustrated in Fig. 5 (bottom).

3 The FIRM Framework

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

1. Based on the insight that resource contention manifests as dynamically evolving CPs, FIRM first detects CP changes and extracts critical microservice instances from them. It does so using the *Tracing Coordinator*, which is marked as (1) in Fig. 6.[2] The tracing coordinator collects tracing and telemetry data from every microservice instance and stores them in a centralized graph database for processing. It is described in §3.1.
2. The *Extractor* detects SLO violations and queries the Tracing Coordinator with collected real-time data (a) to extract CPs (marked as (2) and described in §3.2) and (b) to localize critical microservice instances that are likely causes of SLO violations (marked 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 reprovision resources for the critical instances (marked 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 the 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 model and setup are described in §3.4.
4. Finally, actions are validated and executed on the underlying Kubernetes cluster through the deployment module (marked as (5) and described in §3.5).
5. In order to train the ML models in the Extractor as well as the RL agent (i.e., to 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 marked 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

[2]Unless otherwise specified, (*) refers to annotations in Fig. 6.

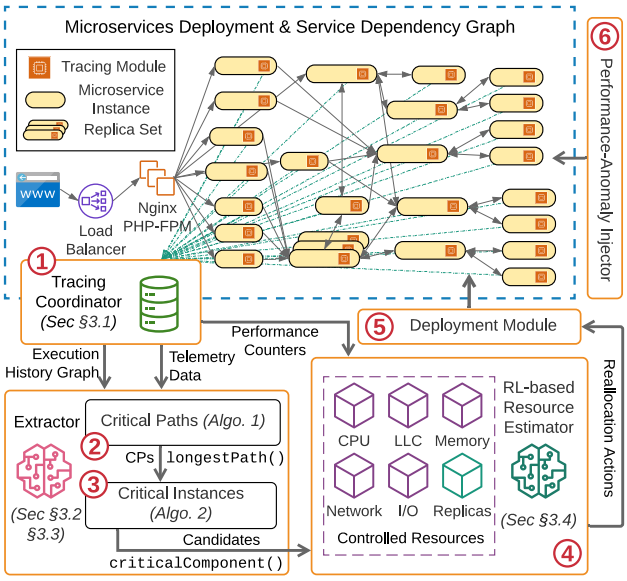


Figure 6: FIRM architecture overview.

performance [111–115]. 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 and records time, local profiling information, and RPC calls (e.g., source and destination services). The execution paths are combined to form the *execution history graph* (see §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 [95] and its open-source implementations, e.g., Jaeger [112] and Zipkin [115]. Each microservice instance is coupled with an OpenTracing-compliant [75] *tracing agent* that measures spans. 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 [72] 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 (see §3.2 and §3.3). Distributed clock drift and time shifting are handled using the Jaeger framework. In addition, the Tracing Coordinator collects telemetry data from the systems running the microservices. The data collected in our experiments is listed in Table 2. The distributed tracing and telemetry collection overhead is indiscernible, i.e., we observed a $<0.4\%$ loss in throughput and a $<0.15\%$ loss in latency. FIRM had a

Table 2: Collected telemetry data and sources.

cAdvisor [13] & **Prometheus** [82]
`cpu_usage_seconds_total, memory_usage_bytes,`
`fs_write/read_seconds, fs_usage_bytes,`
`network_transmit/receive_bytes_total, processes`
Linux perf subsystem [79]
`offcore_response.*.llc_hit/miss.local_DRAM,`
`offcore_response.*.llc_hit/miss.remote_DRAM`

maximum CPU overhead of 4.6% for all loads running in our experiments on the four benchmarks [34, 116]. With FIRM, the network in/out traffic without sampling traces increased by 3.4%/10.9% (in bytes); the increase could be less in production environments with larger message sizes [63].

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 §2 that a CP is the longest path in the request's execution history graph. Hence, changes in the end-to-end latency of an application are often determined by the slowest execution of one or more microservices on its CP.

We identify the CP in an execution history graph by using Alg. 1, which is a weighted longest path algorithm proposed to retrieve CPs in the microservices context. The algorithm needs to take into account the major communication and computation patterns in microservice architectures: (a) *parallel*, (b) *sequential*, 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 that overlap with each other in the execution history graph, e.g., U, V, and T in Fig. 2(b). Formally, for two child spans i with start time st_i and end time et_i, and j with st_j, et_j of the same parent span p, they are called *parallel* if $(st_j < st_i < et_j) \vee (st_i < st_j < et_i)$.
- *Sequential workflows* are characterized by one or more child spans of a 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 $t_{i\to p} \leq t_{p\to j}$, i.e., i completes and sends its result to p before j does. 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 a sufficient number of request executions, there is a violation of that inequality, then the services are not sequential.
- *Background workflows* are those that do not return values to their parent spans, e.g., W in Fig. 2(b). Background workflows are not part of CPs since no other span depends on their execution, but they may be considered responsible for SLO violations when FIRM's Extractor is localizing critical components (see §3.3). That is because background workflows may also contribute to the contention of underlying shared resource.

Algorithm 1 Critical Path Extraction

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

```
procedure LongestPath(G, currentNode)
    path ← ∅
    path.add(currentNode)
    if currentNode.childNodes == None then
        Return path
    end if
    lrc ← currentNode.lastReturnedChild
    path.extend(LongestPath(G, lrc))
    for each cn in currentNode.childNodes do
        if cn.happensBefore(lrc) then
            path.extend(LongestPath(G, cn))
        end if
    end for
    Return path
end procedure
```

Algorithm 2 Critical Component Extraction

Require: Critical Path CP, Request Latencies T

```
procedure CriticalComponent(G, T)
    candidates ← ∅
    T_CP ← T.getTotalLatency() ▷ Vector of CP latencies
    for i ∈ CP do
        T_i ← T.getLatency(i)
        T_99 ← T_i.percentile(99)
        T_50 ← T_i.percentile(50)
        RI ← PCC(T_i, T_CP)          ▷ Relative Importance
        CI ← T_99/T_50               ▷ Congestion Intensity
        if SVM.classify(RI, CI) == True then
            candidates.append(i)
        end if
    end for
    Return candidates
end procedure
```

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 the performance variability and level of request congestion. Variability represents the single largest opportunity to reduce tail latency. The two features are then fed into an incremental SVM classifier to get binary decisions, i.e., on whether that instance should have its resources

re-provisioned or not. The approach is a dynamic selection policy that 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 (i.e., per-CP variability) and the variability caused by congestion in the service queues of each individual microservice instances (i.e., per-instance variability).

Per-CP Variability: Relative Importance. Relative importance [62, 110, 122] 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 that the variance of each variable T_i makes toward explaining the total variance of T_{CP}. To do so, we use the Pearson correlation coefficient [12] (also called zero-order correlation), i.e., $PCC(T_i, T_{CP})$, as the measurement, and hence the resulting statistic is known as the variance explained [31]. The sum of $PCC(T_i, T_{CP})$ over all microservice instances along the CP is 1, and the relative importance values of microservices can be ordered by $PCC(T_i, T_{CP})$. The larger the value is, the more variability it contributes to the end-to-end CP variability.

Per-Instance Variability: Congestion Intensity. For each microservice instance in a CP, congestion intensity is defined as the ratio of the 99th percentile latency to the median latency. Here, we chose the 99th percentile instead of the 70th or 80th percentile to target the tail latency behavior. The chosen 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 handle only a subset of the 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 gains 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 paths in 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 [29,58] implemented using stochastic gradient descent optimization and RBF kernel approximation by `scikit-learn` libraries [91]. Triggered by detected SLO violations, both critical path extraction and critical component extraction are stateless and multithreaded; thus, the workload scales with

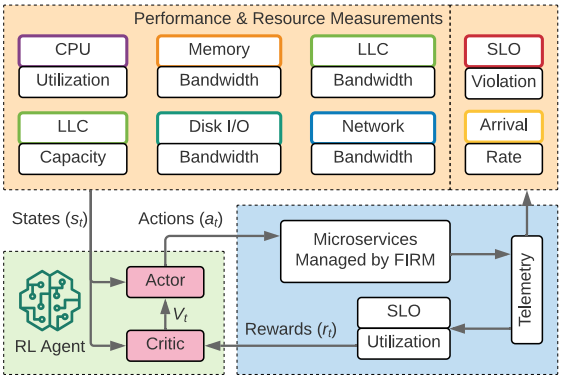


Figure 7: Model-free actor-critic RL framework for estimating resources in a microservice instance.

the size of the microservice application and the cluster. They together constitute FIRM's extractor (i.e., ② and ③). Experiments (§4.2) show that it reports SLO violation candidates with feasible accuracy and achieves completeness with §3.4 by choosing a threshold with a reasonable false-positive rate.

3.4 SLO Violation Mitigation Using RL

Given the list of critical service instances, FIRM's Resource Estimator, i.e., 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 by using measurements of tracing and telemetry data (see Table 2) collected from the Tracing Coordinator. When jointly analyzed, such data provides 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. We next give a brief RL primer before presenting FIRM's RL model.

RL Primer. An RL agent solves a *sequential decision-making problem* (modeled as a Markov decision process) by interacting with an environment. At each discrete time step t, the agent observes a *state of the environment* $s_t \in S$, and performs an *action* $a_t \in A$ based on its *policy* $\pi_\theta(s)$ (parameterized by θ), which maps *state space* S to *action space* A. At the following time step $t+1$, the agent observes an *immediate reward* $r_t \in R$ given by a reward function $r(s_t, a_t)$; the immediate reward represents the loss/gain in transitioning from s_t to s_{t+1} because of action a_t. The tuple (s_t, a_t, r_t, s_{t+1}) is called one *transition*. The agent's goal is to optimize the policy π_θ so as to maximize the expected *cumulative discounted reward* (also called the value function) from the start distribution $J = \mathbb{E}[G_1]$, where the return from a state G_t is defined

to be $\sum_{k=0}^{T}\gamma^k r_{t+k}$. The discount factor $\gamma \in (0,1]$ penalizes the predicted future rewards.

Two main categories of approaches are proposed for policy learning: value-based methods and policy based methods [5]. In value-based methods, the agent learns an estimate of the optimal value function and approaches the optimal policy by maximizing it. In policy-based methods, the agent directly tries to approximate the optimal policy.

Why RL? Existing performance-modeling-based [23, 38, 39, 56, 94, 101, 127] or heuristic-based approaches [6, 7, 37, 65, 84] suffer from model reconstruction and retraining problems because they do not address dynamic system status. Moreover, they require expert knowledge, and it takes 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 the 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. In particular, FIRM utilizes the deep deterministic policy gradient (DDPG) algorithm [59], which is a *model-free*, *actor-critic* RL framework (shown in Fig. 7). 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 (i.e., transitions between states), which are difficult to model precisely. When microservices are updated, the simulations of state transitions used in model-based RL are no longer valid.
2. The Actor-critic framework combines policy-based and value-based methods (i.e., consisting of an actor-net and a critic-net as shown in Fig. 8), and that is suitable for continuous stochastic environments, converges faster, and has lower variance [41].

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. The critic's estimation of the expected return allows the actor to update with gradients that have lower variance, thus speeding up the learning process (i.e., achieving convergence). 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 $\mathcal{D}$ that stores 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 pseudocode of the training algorithm is shown in Algorithm 3. RL training proceeds in episodes and each episode consists of T time steps. At each time step, both actor and critic neural nets are updated once.

Algorithm 3 DDPG Training

```
Randomly init Q_w(s,a) and π_θ(a|s) with weights w & θ.
Init target network Q' and π' with w' ← w & θ' ← θ
Init replay buffer D ← ∅
for episode = 1, M do
    Initialize a random process N for action exploration
    Receive initial observation state s_1
    for t = 1, T do
        Select and execute action a_t = π_θ(s_t) + N_t
        Observe reward r_t and new state s_{t+1}
        Store transition (s_t, a_t, r_t, s_{t+1}) in D
        Sample N transitions (s_i, a_i, r_i, s_{i+1}) from D
        Update critic by minimizing the loss L(w)
        Update actor by sampled policy gradient ∇_θ J
        w' ← γw + (1−γ)w'
        θ' ← γθ + (1−γ)θ'
    end for
end for
```

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

$$Q_w(s_t, a_t) = \mathbb{E}[r(s_t, a_t) + \gamma Q_w(s_{t+1}, \pi(s_{t+1}))]$$

$$\mathcal{L}(w) = \frac{1}{N}\sum_i (r_i + \gamma Q'_{w'}(s_{i+1}, \pi'_{\theta'}(s_{i+1})) - Q_w(s_i, a_i))^2.$$

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

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

Problem Formulation. To estimate resources for a microservice instance, we formulate a sequential decision-making problem which can be solved by the above RL framework. Each microservice instance is deployed in a separate container with a tuple of resource limits $RLT = (RLT_{cpu}, RLT_{mem}, RLT_{llc}, RLT_{io}, RLT_{net})$, since we are considering CPU utilization, memory bandwidth, LLC capacity, disk I/O bandwidth, and network bandwidth as our resource model.[3] This limit for each type of resource is predetermined (usually overprovisioned) before the microservices are deployed in the cluster and later controlled by FIRM.

At each time step t, utilization RU_t for each type of resource is retrieved using performance counters as telemetry data in ①. In addition, FIRM's Extractor also collects current

[3] The resource limit for the CPU utilization of a container is the smaller of $\hat{R}_i$ and the number of threads $\times$ 100.

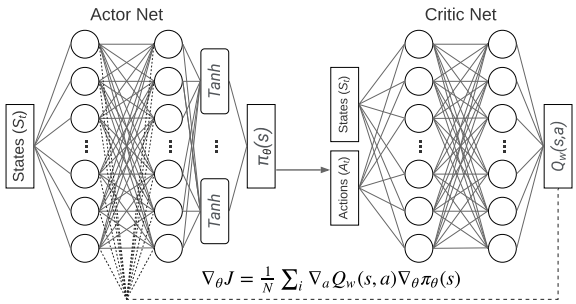


Figure 8: Architecture of actor-critic nets.

latency, request arrival rate, and request type composition (i.e., percentages of each type of request). Based on these measurements, the RL agent calculates the states listed in Table 3 and described below.

- *SLO maintenance ratio* (SM_t) 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 ($SM_t = 1$).
- *Workload changes* (WC_t) is defined as the ratio of the arrival rates at the current and previous time steps.
- *Request composition* (RC_t) is defined as a unique value encoded from an array of request percentages by using `numpy.ravel_multi_index()` [74].

For each type of resources i, there is a predefined resource upper limit $\hat{R}_i$ and a lower limit $\underset{\sim}{R}_i$ (e.g., the CPU time limit cannot be set to 0). The actions available to the RL-agent is to set $RLT_i \in [\hat{R}_i, \underset{\sim}{R}_i]$. If the amount of resource reaches the total available amount, then a scale-out operation is needed. Similarly, if the resource limit is below the lower bound, a scale-in operation is needed. The CPU resources serve as one exception to the above procedure: it would not improve the performance if the CPU utilization limit were 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 π_t that results in as few SLO violations as possible (i.e., $\min_{\pi_t} SM_t$) while keeping the resource utilization/limit as high as possible (i.e., $\max_{\pi_t} RU_t/RLT_t$). Based on both objectives, the reward function is then defined as $r_t = \alpha \cdot SM_t \cdot |\mathcal{R}| + (1-\alpha) \cdot \sum_i^{|\mathcal{R}|} RU_i/RLT_i$, where $\mathcal{R}$ 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 (i.e., for deployment) because of the time required to train such tailored models for user workloads, which might have significant churn. FIRM addresses the problem of rapid model training by using transfer learning in the domain of RL [14, 105, 106], whereby agents for SLO violation mitigation can be trained for either the general case (i.e., any microservices) or the specialized case (i.e., "transferred" to the behavior of individualized microservices). The pre-trained model used in the specialized case is called the base model or the source model. That approach is possible because prior understanding of a problem structure helps one solve similar problems quickly, with the remaining task being to understand the behavior of updated microservice instances. Related work on base model selection and task similarity can be found in [105, 106], but the base model that FIRM uses for transfer learning is always the RL model learned in the general case because it has been shown in evaluation to be comparable with specialized models. We demonstrate the efficacy of transfer learning in our evaluation described in §4. The RL model that FIRM uses is designed to scale since both the state space and the action space are independent of the size of the application or the cluster. In addition to having the general case RL agent, the FIRM framework also allows for the deployment of specialized per-microservice RL agents.

Table 3: State-action space of the RL agent.

State (s_t)
SLO Maintenance Ratio (SM_t), Workload Changes (WC_t), Request Composition (RC_t), Resource Utilization (RU_t)
Action Space (a_t)
Resource Limits $RLT_i(t), i \in \{\text{CPU, Mem, LLC, IO, Net}\}$

Table 4: RL training parameters.

Parameter	**Value**
# Time Steps × # Minibatch	300×64
Size of Replay Buffer	10^5
Learning Rate	Actor (3×10^{-4}), Critic (3×10^{-3})
Discount Factor	0.9
Soft Update Coefficient	2×10^{-3}
Random Noise	μ (0), σ (0.2)
Exploration Factor	ε (1.0), ε-decay (10^{-6})

Implementation Details. We implemented the DDPG training algorithm and the actor-critic networks using `PyTorch` [83]. The critic net contains two fully connected hidden layers with 40 hidden units, all using ReLU activation function. The first two hidden layers of the actor net are fully connected and both use ReLU as the activation function while the last layer uses Tanh as the activation function. 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 chose that setup because adding more layers and hidden units does not increase performance in our experiments with selected microservice benchmarks; instead, it slows down training speed significantly. Hyperparameters of the RL model are listed in Table 4. We set the time step for training the model to be 1 second, which is sufficient for action execution (see Table 6). The latencies of each RL train-

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

Performance Anomaly Types	Tools/Benchmarks
Workload Variation	`wrk2` [123]
Network Delay	`tc` [107]
CPU Utilization	iBench [24], `stress-ng` [100]
LLC Bandwidth & Capacity	iBench, `pmbw` [80]
Memory Bandwidth	iBench [24], `pmbw` [80]
I/O Bandwidth	Sysbench [102]
Network Bandwidth	`tc` [107], Trickle [117]

ing update and inference step are 73 $\pm$ 10.9 ms and 1.2 $\pm$ 0.4 ms, respectively. The average CPU and memory usage of the Kubernetes pod during the training stage are 210 millicores and 192 Mi, respectively.

3.5 Action Execution

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 amount of the resource available on that physical machine. FIRM assumes that machine resources are unlimited and thus does not have admission control or throttling. If an action leads to oversubscribing of a resource, then it is replaced by a scale-out operation.

- *CPU Actions:* Actions on scaling CPU utilization are executed through modification of `cpu.cfs_period_us` and `cpu.cfs_quota_us` in the `cgroups` CPU subsystem.
- *Memory Actions:* We use Intel MBA [49] and Intel CAT [48] technologies to control the memory bandwidth and LLC capacity of containers, respectively.[4]
- *I/O Actions:* For I/O bandwidth, we use the `blkio` subsystem in `cgroups` to control input/output access to disks.
- *Network Actions:* For network bandwidth, we use the Hierarchical Token Bucket (HTB) [45] queueing discipline in Linux Traffic Control. Egress `qdiscs` can be directly shaped by using HTB. Ingress `qdiscs` are redirected to the virtual device `ifb` interface and then shaped through the application of egress rules.

3.6 Performance Anomaly Injector

We accelerate the training of the machine learning models in FIRM's Extractor and the RL agent through performance anomaly injections. The injection provides the ground truth data for the SVM model, as the injection targets are controlled and known from the campaign files. It also allows the RL agent to quickly span the space of adverse resource contention behavior (i.e., the exploration-exploitation trade-off in RL). That is important, as real-world workloads might not experience all adverse situations within a short training time. We implemented a performance anomaly injector, i.e., (6), in which the injection targets, type of anomaly, injection time, duration, patterns, 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 the injector is configured and used) for the injector will be discussed in §4. The injector comprises seven types of performance anomalies that can cause SLO violations. They are listed in Table 5 and described below.

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

Network Delay. We use Linux traffic control (`tc`) to add simulated delay to network packets. Given the mean and standard deviation of the network delay latency, each network packet is delayed following a normal distribution.

CPU Utilization. We implement the CPU stressor based on iBench and `stree-ng` to exhaust a specified level of CPU utilization on a set of cores by exercising floating point, integer, bit manipulation and control flows.

LLC Bandwidth & Capacity. We use iBench and `pmbw` to inject interference on the Last Level Cache (LLC). For bandwidth, the injector performs streaming accesses in which the size of the accessed data 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` to generate 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 first creates test files that are larger than the size of system RAM. Then 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 Tricle for limiting the upload/download rate of a specific microservice instance.

Network Bandwidth. We use Linux traffic control (`tc`) to limit egress network bandwidth. For ingress network bandwidth, an intermediate function block (`ifb`) pseudo interface is set up, and inbound traffic is directed through that. In that way, the inbound traffic then becomes schedulable by the egress `qdisc` on the `ifb` interface, so the same rules for egress can be applied directly to ingress.

[4] Our evaluation on IBM Power systems (see §4) did not use these actions because of a lack of hardware support. OS support or software partitioning mechanisms [60, 85] can be applied; we leave that to future work.

4 Evaluation

4.1 Experimental Setup

Benchmark Applications. We evaluated FIRM on a set of end-to-end interactive and responsive real-world microservice benchmarks: (i) DeathStarBench [34], consisting of *Social Network*, *Media Service*, and *Hotel Reservation* microservice applications, and (ii) Train-Ticket [128], consisting of the *Train-Ticket Booking Service*. *Social Network* implements a broadcast-style social network with unidirectional follow relationships whereby users can publish, read, and react to social media 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* provides typical train-ticket booking functionalities, such as ticket inquiry, reservation, payment, change, and user notification. These benchmarks contain 36, 38, 15, and 41 unique microservices, respectively; cover all workflow patterns (see §3.2); and use various programming languages including Java, Python, Node.js, Go, C/C++, Scala, PHP, and Ruby. All microservices are deployed in separate Docker containers.

System Setup. We validated our design by implementing a prototype of FIRM that used Kubernetes [20] as the underlying container orchestration framework. We deployed the four microservice benchmarks with FIRM separately on a Kubernetes cluster of 15 two-socket physical nodes without specifying any anti-colocation rules. Each server consists of 56–192 CPU cores and RAM that varies from 500 GB to 1000 GB. Nine of the servers use Intel x86 Xeon E5s and E7s processors, while the remaining ones use IBM ppc64 Power8 and Power9 processors. All machines run Ubuntu 18.04.3 LTS with Linux kernel version 4.15.

Load Generation. We drove the services with various open-loop asynchronous workload generators [123] to represent an active production environment [17, 97, 118]. We uniformly generated workloads for every request type across all microservice benchmarks. The parameters for the workload generators were the same as those for DeathStarBench (which we applied to Train-Ticket as well), and varied from predictable constant, diurnal, distributions such as Poisson, to unpredictable loads with spikes in user demand. The workload generators and the microservice benchmark applications were never co-located (i.e., they executed on different nodes in the cluster). To control the variability in our experiments, we disabled all other user workloads on the cluster.

Injection and Comparison Baselines. We used our performance anomaly injector (see §3.6) to inject various types of performance anomalies into containers uniformly at random with configurable injection timing and intensity. Following the common way to study resource interference, our experiments on SLO violation mitigation with anomalies were designed to be comprehensive by covering the worst-case scenarios, given the random and nondeterministic nature of shared-resource interference in production environments [22, 78]. Unless otherwise specified, (i) the anomaly injection time interval was in an exponential distribution with $\lambda = 0.33s^{-1}$, and (ii) the anomaly type and intensity were selected uniformly at random. We implemented two baseline approaches: (a) the Kubernetes autoscaling mechanism [55] and (b) an AIMD-based method [38, 101] to manage resources for each container. Both approaches are rule-based autoscaling techniques.

4.2 Critical Component Localization

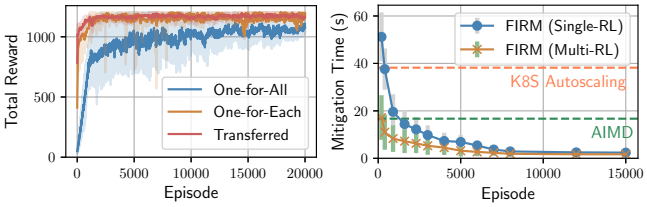
Here, we use the techniques presented in §3.2 and §3.3 to study the effectiveness of FIRM in identifying the microservices that are most likely to cause SLO violations.

Single anomaly localization. We first evaluated 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 increased the intensity of injected resource interference and recorded end-to-end latencies. The intensity parameter was chosen uniformly at random between [start-point, end-point], where the start-point is the intensity that starts to trigger SLO violations, and the end-point is the intensity when either the anomaly injector has consumed all possible resources or over 80% of user requests have been dropped or returned time. 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) and 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 vary significantly. In particular, FIRM's Extractor module achieved near 100% true-positive rate, when the false-positive rate was between $[0.12, 0.16]$.

Multi-anomaly localization. There is no guarantee that only one resource contention will happen at a time under dynamic datacenter workloads [40, 42, 96, 98] and therefore we also studied the container localization performance under multi-anomaly injections and compared machines with two different processor ISAs (x86 and ppc64). An example of the intensity distributions of all the anomaly types used in this experiment are shown in Fig. 9(c). The experiment was divided into time windows of 10 s, i.e., T_i from Fig. 9(c)). At each time window, we picked the injection intensity of each anomaly type uniformly at random with range [0,1]. Our observations are reported in Fig. 9(b). The average accuracy for localizing critical components in each application ranged from 92% to 94%. The overall average localization accuracy was 93% across four microservice benchmarks. Overall, we observe that the accuracy of the Extractor did not differ between the two sets of processors.

(a) ROC under single-anomaly. **(b)** Average accuracy under multi-anomaly. **(c)** Anomaly injection intensity and timing.

Figure 9: Critical Component Localization Performance: (a) ROC curves for detection accuracy; (b) Variation of localization accuracies across processor architectures; (c) Anomaly-injection intensity, types, and timing.



(a) Total reward. **(b)** SLO mitigation time.

Figure 10: Learning curve showing total reward during training and SLO mitigation performance.

4.3 RL Training & SLO Violation Mitigation

To understand the convergence behavior of FIRM's RL agent, we trained three RL models that were subjected to the same sequence of performance anomaly injections (described in §4.1). The three RL models are: (i) a common RL agent for all microservices (one-for-all), (ii) a tailored RL agent for a particular microservice (one-for-each), and (iii) a transfer-learning-based RL agent. RL training proceeds in episodes (iterations). We set the number of time steps in a training episode to be 300 (see Table 4), but for the initial stages, we terminate the RL exploration early so that the agent could reset and try again from the initial state. We did so because the initial policies of the RL agent are unable to mitigate SLO violations. Continuously injecting performance anomalies causes user requests to drop, and thus only a few request traces were generated to feed the agent. As the training progressed, the agent improved its resource estimation policy and could mitigate SLO violations in less time. At that point (around 1000 episodes), we linearly increased the number of time steps to let the RL agent interact with the environment longer before terminating the RL exploration and entering the next iteration.

We trained the abovementioned three RL models on the Train-Ticket benchmark. We studied the generalization of the RL model by evaluating the end-to-end performance of FIRM on the DeathStarBench benchmarks. Thus, we used DeathStarBench as a validation set in our experiments. Fig. 10(a) shows that as the training proceeded, the agent was getting better at mitigation, and thus the moving average of episode rewards was increasing. The initial steep increase benefits from early termination of episodes and parameter exploration. Transfer-learning-based RL converged even faster (around 2000 iterations[5]) because of parameter sharing. The one-for-all RL required more iterations to converge (around 15000 iterations) and had a slightly lower total reward (6% lower compared with one-for-each RL) during training.

In addition, higher rewards, for which the learning algorithm explicitly optimizes, correlate with improvements in SLO violation mitigation (see Fig. 10(b)). For models trained in every 200 episodes, we saved the checkpoint of parameters in the RL model. Using the parameter, we evaluated the model snapshot by injecting performance anomalies (described in §4.1) continuously for one minute and observed when SLO violations were mitigated. Fig. 10(b) shows that FIRM with either a single-RL agent (one-for-all) or a multi-RL agent (one-for-each) improved with each episode in terms of the SLO violation mitigation time. The starting policy at iteration 0–900 was no better than the Kubernetes autoscaling approach, but after around 2500 iterations, both agents were better than either Kubernetes autoscaling or the AIMD-based method. Upon convergence, FIRM with a single-RL agent achieved a mitigation time of 1.7 s on average, which outperformed the AIMD-based method by up to 9× and Kubernetes autoscaling by up to 30× 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 it on DeathStarBench benchmarks based on the hyperparameter tuned during training with the Train-Ticket benchmark. To understand the 10–30× improvement demonstrated above, we measured the 99th percentile end-to-end latency when the microservices were being managed by the two baseline approaches and by FIRM. Fig. 11(a) shows the cumulative distribution of the end-to-end

[5] 1000 iterations correspond to roughly 30 minutes with each iteration consisting of 300 time steps.

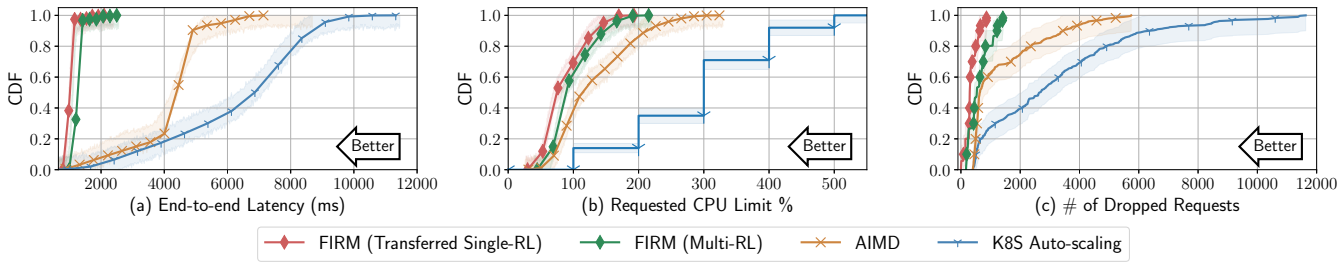


Figure 11: Performance comparisons (CDFs) of end-to-end latency, requested CPU limit, and the number of dropped requests.

latency. We observed that the AIMD-based method, albeit simple, outperforms the Kubernetes autoscaling approach by 1.7× on average and by 1.6× in the worst case. In contrast, FIRM:

1. Outperformed both baselines by up to 6× and 11×, which leads to 9× and 16× fewer SLO violations;
2. Lowered the overall requested CPU limit by 29–62%, as shown in Fig. 11(b), and increased the average cluster-level CPU utilization by up to 33%; and
3. Reduced the number of dropped or timed out user requests by up to 8× as shown in Fig. 11(c).

FIRM can provide these benefits because it 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 outperforms heuristics-based approaches.

5 Discussion

Necessity and Challenges of Modeling Low-level Resources. Recall from §2 that modeling of 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 and quantifiable performance gain while dealing with uncertain and noisy measurements [76, 120]. FIRM addresses the issue by modeling that dependency in an RL-based feedback loop, which automatically explores the action space to generate optimal policies without human intervention.

Why a Multilevel ML Framework? A model of the states of all microservices that is fed as the input to a single large ML model [81, 126] leads to (i) state-action space explosion issues that grow with the number of microservices, thus increasing the training time; and (ii) dependence between the microservice architecture and the ML-model, which sacrifices the generality. FIRM addresses those problems by incorporating a two-level ML framework. The first level ML model uses SVM to filter the microservice instances responsible for SLO violations, thereby reducing the number of microservices that need to be considered in mitigating SLO violations. That enables the second level ML model, the RL agent, to be trained faster and removes dependence on the application architecture. That, in turn, helps avoid RL model reconstruction/retraining.

Table 6: Avg. latency for resource management operations.

Operation	Partition (Scale Up/Down)					Container Start	
	CPU	Mem	LLC	I/O	Net	Warm	Cold
Mean (*ms*)	2.1	42.4	39.8	2.3	12.3	45.7	2050.8
Std Dev (*ms*)	0.3	11.0	9.2	0.4	1.1	6.9	291.4

Lower Bounds on Manageable 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, that is the minimum duration of latency spikes that any RM approach can handle. For transient SLO violations, which last shorter than the minimum duration, the action generated by FIRM will always miss the mitigation deadline and can potentially harm overall system performance. Worse, it may lead to oscillations between scaling operations. Predicting the spikes before they happen, and proactively taking mitigation actions can be a solution. However, it is a generally-acknowledged difficult problem, as microservices are dynamically evolving, in terms of both load and architectural design, which is subject to our future work.

Limitations. FIRM has several limitations that we plan to address in future work. First, FIRM currently focuses on resource interference caused by real workload demands. However, FIRM lacks the ability to detect application bugs or misconfigurations, which may lead to failures such as memory leak. Allocating more resources to such microservice instances may harm the overall resource efficiency. Other sources of SLO violations, including global resource sharing (e.g., network switches or global file systems) and hardware causes (e.g., power-saving energy management), are also beyond FIRM's scope. Second, the scalability of FIRM is limited by the maximum scalability of the centralized graph database, and the boundary caused by the network traffic telemetry overhead. (Recall the lower bound on the SLO violation duration.) Third, we plan to implement FIRM's tracing module based on side-car proxies (i.e., service meshes) [15] that minimizes application instrumentation and has wider support of programming languages.

6 Related Work

SLO violations in cloud applications and microservices are 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 [18,32,57,95,111–115] or middleware/OS-level instrumentation [10,16,63,109] (e.g., Sieve [109] utilizes a kernel module *sysdig* [103] which provides system calls as an event stream containing tracing information about the monitored process to a user application).

Root Cause Analysis. A large body of work [16, 35, 50, 52, 61, 63, 93, 109, 121, 124] provides promising evidence that data-driven diagnostics help detect performance anomalies and analyze root causes. For example, Sieve [109] leverages Granger causality to correlate performance anomaly data series with particular metrics as potential root causes. Pinpoint [16] runs clustering analysis on Jaccard similarity coefficient to determine the components that are mostly correlated with the failure. Microscope [61] and MicroRCA [124] 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 [51]. Seer [35] uses 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 [65,84]: (a) rule-based (commonly offered by cloud providers [6, 7, 37]), (b) time series analysis (regression on resource utilization, performance, and workloads), (c) model-based (e.g., queueing networks), or (d) RL-based. Some approaches combine several techniques. For instance, Auto-pilot [88] combines time series analysis and RL algorithms to scale the number of containers and associated CPU/RAM. Unfortunately, when applied to microservices with large scale and complex dependencies, independent scaling of each microservice instance results in suboptimal solutions (because of critical path intersection and insight 2 in §2), and it is difficult to define sub-SLOs for individual instances. Approaches for autoscaling microservices or distributed dataflows [39,56,81,126,127] make scaling decisions on the number of replicas and/or container size without considering low-level shared-resource interference. ATOM [39] and Microscaler [127] do so by using a combination of queueing network- and heuristic-based approximations. ASFM [81] uses recurrent neural network activity to predict workloads and translates application performance to resources by using linear regression. Streaming and data-processing scalers like DS2 [56] and MIRAS [126] leverage explicit application-level modeling and apply RL to represent the resource-performance mapping of operators and their dependencies.

Cluster Management. The emergence of cloud computing motivates the prevalence of cloud management platforms that provide services such as monitoring, security, fault tolerance, and performance predictability. Examples include Borg [119], Mesos [43], Tarcil [28], Paragon [25], Quasar [26], Morpheus [54], DeepDive [73], and Q-clouds [71]. In this paper, we do not address the problem of cluster orchestration. FIRM can work in conjunction with those cluster management tools to reduce SLO violations.

7 Conclusion

We propose *FIRM*, an ML-based, fine-grained resource management framework that addresses SLO violations and resource underutilization 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× while reducing the overall CPU limit by up to 62%. Overall, FIRM enables fast mitigation of SLOs by using efficient resource provisioning, which benefits both cloud service providers and microservice owners. FIRM is open-sourced at `https://gitlab.engr.illinois.edu/DEPEND/firm.git`.

8 Acknowledgment

We thank the OSDI reviewers and our shepherd, Rebecca Isaacs, for their valuable comments that improved the paper. We appreciate K. Atchley, F. Rigberg, and J. Applequist for their insightful comments on the early drafts of this manuscript. This research was supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under award No. 2015-02674. This work is partially supported by the National Science Foundation (NSF) under grant No. 2029049; by a Sandia National Laboratories[6] under contract No. 1951381; by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR), a research collaboration that is part of the IBM AI Horizon Network; and by Intel and NVIDIA through equipment donations. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF, Sandia National Laboratories, IBM, NVIDIA, or, Intel. Saurabh Jha is supported by a 2020 IBM PhD fellowship.

[6]Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

References

[1] Ivo Adan and Jacques Resing. *Queueing theory*. Eindhoven University of Technology Eindhoven, 2002.

[2] Bernhard Ager, Fabian Schneider, Juhoon Kim, and Anja Feldmann. Revisiting cacheability in times of user generated content. In *2010 INFOCOM IEEE Conference on Computer Communications Workshops*, pages 1–6. IEEE, 2010.

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

[4] 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.

[5] Kai Arulkumaran, Marc Peter Deisenroth, Miles Brundage, and Anil Anthony Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.

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

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

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

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

[10] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *OSDI*, volume 4, pages 18–18, 2004.

[11] 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.

[12] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. Pearson correlation coefficient. In *Noise Reduction in Speech Processing*, pages 1–4. Springer, 2009.

[13] cAdvisor. https://github.com/google/cadvisor, Accessed 2020/01/23.

[14] Luiz A. Celiberto Jr, Jackson P. Matsuura, Ramón López De Màntaras, and Reinaldo A.C. Bianchi. Using transfer learning to speed-up reinforcement learning: A case-based approach. In *Proceedings of 2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, pages 55–60. IEEE, 2010.

[15] Ramaswamy Chandramouli and Zack Butcher. Building secure microservices-based applications using service-mesh architecture. *NIST Special Publication*, 800:204A, 2020.

[16] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings International Conference on Dependable Systems and Networks*, pages 595–604. IEEE, 2002.

[17] Shuang Chen, Shay GalOn, Christina Delimitrou, Srilatha Manne, and Jose F. Martinez. Workload characterization of interactive cloud services on big and small server platforms. In *Proceedings of 2017 IEEE International Symposium on Workload Characterization (IISWC)*, pages 125–134. IEEE, 2017.

[18] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. The mystery machine: End-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 217–231, 2014.

[19] Docker Swarm. https://www.docker.com/products/docker-swarm, Accessed 2020/01/23.

[20] Kubernetes. https://kubernetes.io/, Accessed 2020/01/23.

[21] CoreOS rkt, a security-minded, standards-based container engine. https://coreos.com/rkt/, Accessed 2020/01/23.

[22] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[23] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource provisioning of web applications in heterogeneous clouds. In *Proceedings of the 2nd USENIX Conference on Web Application Development*, pages 49–60. USENIX Association, 2011.

[24] Christina Delimitrou and Christos Kozyrakis. iBench: Quantifying interference for datacenter applications. In *Proceedings of 2013 IEEE International Symposium on Workload Characterization (IISWC)*, pages 23–33. IEEE, 2013.

[25] Christina Delimitrou and Christos Kozyrakis. Paragon: QoS-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[26] Christina Delimitrou and Christos Kozyrakis. Quasar: Resource-efficient and QoS-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[27] Christina Delimitrou and Christos Kozyrakis. Amdahl's law for tail latency. *Communications of the ACM*, 61(8):65–72, 2018.

[28] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. Tarcil: Reconciling scheduling speed and quality in large shared clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 97–110, 2015.

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

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

[31] 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.

[32] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, pages 271–284, 2007.

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

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

[35] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 19–33, 2019.

[36] Mrittika Ganguli, Rajneesh Bhardwaj, Ananth Sankaranarayanan, Sunil Raghavan, Subramony Sesha, 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.

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

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

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

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

[41] Ivo Grondman, Lucian Busoniu, Gabriel A.D. 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.

[42] 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 *Proceedings of 2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629. IEEE, 2018.

[43] 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 *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, volume 11, pages 295–208, 2011.

[44] Todd Hoff. 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.

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

[46] 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.

[47] 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.

[48] Intel cache allocation technology. https://github.com/intel/intel-cmt-cat, Accessed 2020/01/23.

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

[50] 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.

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

[52] Saurabh Jha, Shengkun Cui, Subho Banerjee, Tianyin Xu, Jeremy Enos, Mike Showerman, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. Live forensics for HPC systems: A case study on distributed storage systems. In *Proceedings of the International Conference for High-Performance Computing, Networking, Storage and Analysis*, 2020.

[53] 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.

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

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

[56] 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 *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[57] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 34–50, 2017.

[58] 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.

[59] 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.

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

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

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

[63] 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 *Proceedings of International Conference on Cloud Computing*, pages 287–302. Springer, 2019.

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

[65] 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.

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

[67] 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.

[68] 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.

[69] Matt McGee. 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.

[70] Dirk Merkel. Docker: Lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239):2–2, 2014.

[71] 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.

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

[73] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. DeepDive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of 2013 USENIX Annual Technical Conference (USENIX ATC)*, pages 219–230, 2013.

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

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

[76] Karl Ott and Rabi Mahapatra. Hardware performance counters for embedded software anomaly detection. In *Proceedings of 2018 IEEE 16th Intl. Conf. on Dependable, Autonomic and Secure Computing, the 16th Intl. Conf. on Pervasive Intelligence and Computing, the 4th Intl. Conf. on Big Data Intelligence and Computing and Cyber Science and Technology Congress*, pages 528–535. IEEE, 2018.

[77] 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.

[78] Panagiotis Patros, Stephen A. MacKay, Kenneth B. Kent, and Michael Dawson. Investigating resource interference and scaling on multitenant PaaS clouds. In *Proceedings of the 26th Annual International Conference on Computer Science and Software Engineering*, pages 166–177, 2016.

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

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

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

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

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

[84] 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.

[85] Nauman Rafique, Won-Taek Lim, and Mithuna Thottethodi. Architectural support for operating system-driven CMP cache management. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, pages 2—-12. Association for Computing Machinery, 2006.

[86] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C Snoeren. Cloud control with distributed rate limiting. *ACM SIGCOMM Computer Communication Review*, 37(4):337–348, 2007.

[87] 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.

[88] 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.

[89] 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.

[90] 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.

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

[92] S. Senthil Kumaran. *Practical LXC and LXD: Linux Containers for Virtualization and Orchestration*. Springer, 2017.

[93] Syed Yousaf Shah, Xuan-Hong Dang, and Petros Zerfos. Root cause detection using dynamic dependency graphs from time series data. In *Proceedings of 2018 IEEE International Conference on Big Data (Big Data)*, pages 1998–2003. IEEE, 2018.

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

[95] 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.

[96] Akshitha Sriraman and Abhishek Dhanotia. Accelerometer: 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.

[97] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F. Wenisch. SoftSKU: optimizing server architectures for microservice diversity@ scale. In *Proceedings of the 46th International Symposium on Computer Architecture*, pages 513–526, 2019.

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

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

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

[101] 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 *Proceedings of 2015 54th IEEE Conference on Decision and Control (CDC)*, pages 1631–1636. IEEE, 2015.

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

[103] Sysdig. https://sysdig.com/, Accessed 2020/01/23.

[104] Davide Taibi, Valentina Lenarduzzi, and Claus Pahl. Processes, motivations, and issues for migrating to microservices architectures: An empirical investigation. *IEEE Cloud Computing*, 4(5):22–32, 2017.

[105] Matthew E. Taylor, Gregory Kuhlmann, and Peter Stone. Autonomous transfer for reinforcement learning. In *Proceedings of 2008 International Conference of Autonomous Agents and Multi-Agent Systems (AAMAS)*, pages 283–290, 2008.

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

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

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

[109] 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.

[110] 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.

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

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

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

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

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

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

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

[118] Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara. Workload characterization for microservices. In *Proceedings of 2016 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–10. IEEE, 2016.

[119] 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)*, pages 1–17, Bordeaux, France, 2015.

[120] 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.

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

[122] 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.

[123] wrk2: An HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2, Accessed 2020/01/23.

[124] Li Wu, Johan Tordsson, Erik Elmroth, and Odej Kao. MicroRCA: Root cause localization of performance issues in microservices. In *Proceedings of 2020 IEEE/IFIP Network Operations and Management Symposium (NOMS)*, pages 1–9, 2020.

[125] Cui-Qing Yang and Barton Miller. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 366–367, 1988.

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

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

[128] 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*, 14(8):1–1, 2018.

A Artifact Appendix

A.1 Abstract

FIRM is publicly available at https://gitlab.engr.illinois.edu/DEPEND/firm.git. We provide implementations for FIRM's SVM-based critical component extraction, RL-based SLO violation mitigation, and the performance anomaly injection. In addition, we provide a tracing data set of the four microservice benchmarks deployed on our dedicated Kubernetes cluster of 15 physical nodes. The data set was generated by running open-loop workload generation and performance anomaly injection.

A.2 Artifact Check-list

- **Algorithm:** FIRM's critical component extraction includes an algorithm to find the weighted longest path (i.e., critical path analysis) from the execution history graph of microservices.
- **Model:** FIRM's two-level machine learning architecture includes an SVM-based critical component extraction model and an RL-based SLO violation mitigation model. The latter one is designed based on deep deterministic policy gradient (DDPG).
- **Data set:** The artifact includes a tracing data set collected by running four microservice benchmarks [34, 116] in a 15-node Kubernetes cluster. The microservice benchmarks are driven by workload generation and performance anomaly injection.
- **Hardware:** Experiments can run on a cluster of physical nodes with Intel Cache Allocation Technology (CAT) [48] and Intel Memory Bandwidth Allocation (MBA) [49] enabled.
- **Required disk space:** Neo4j [72] requires 10 GB minimum block storage, and the storage size depends on the size of the database.
- **Set-up instructions:** Set-up instructions are available at the `README.md` file in the repository.
- **Public link:** https://gitlab.engr.illinois.edu/DEPEND/firm.git
- **Code licenses:** Apache License Version 2.0
- **Data licenses:** CC0 License

A.3 Description

A.3.1 How to Access

The artifact is publicly available at https://gitlab.engr.illinois.edu/DEPEND/firm.git.

A.3.2 Hardware Dependencies

Experiments can be run on a cluster of physical nodes with processors that have Intel CAT and MBA technologies enabled. They are required for last-level cache partitioning and memory bandwidth partitioning respectively.

A.3.3 Software Dependencies

Software dependencies are specified at the `README.md` file, which includes Kubernetes, Docker-Compose, and Docker.

A.3.4 Data Sets

The tracing data sets of four microservice benchmarks deployed on our dedicated Kubernetes cluster consisting of 15 heterogeneous nodes are also available. The data sets are not sampled and are from selected types of requests in each benchmark, i.e., compose-posts in the social network application, compose-reviews in the media service application, book-rooms in the hotel reservation application, and reserve-tickets in the train ticket booking application. A detailed description is available at `data/README.md`.

A.4 Installation

Installation instructions are specified at the `README.md` file in the repository.

A.5 Experiment Workflow

Experiments on physical clusters start from deploying the Kubernetes with FIRM. Microservice applications instrumented with the OpenTracing [75] standard are then deployed in the Kubernetes cluster. One can also use the instrumented microservice benchmarks in the repository for experiments. To drive the experiments, workload generators and performance anomaly injectors should be configured and installed accordingly. Then the training of FIRM's ML models is divided into two phases. In the first phase, the workflow stops at the SLO violation localization. The SVM model is trained with the feature data retrieved from the tracing coordinator and the label data from the performance anomaly injection campaign. In the second phase, the workflow continues and FIRM's RL agent is trained by interacting with the environment.

A.6 Experiment Customization

FIRM's multilevel ML modeling provides the flexibility of customizing the algorithms for both SLO violation localization and mitigation. The SVM model can be replaced by other supervised learning models or other heuristics-based methods. The DDPG algorithm used by the RL agent can also be replaced by other RL algorithms. The repository consists of the implementations of other alternative RL models such as proximal policy optimization (PPO) and policy gradient.

In addition, different types of resources in control are also configurable in the RL agent and the performance anomaly injector. That pluggability allows one to add or remove resources, and to change the actions associated with each type of resource.

A.7 AE Methodology

Submission, reviewing and badging methodology:

- https://www.usenix.org/conference/osdi20/call-for-artifacts