Slowpoke: End-to-end Throughput Optimization Modeling for Microservice Applications

Yizheng Xie
Brown University

Di Jin
Brown University

Oğuzhan Çölkesen
Brown University

Vasiliki Kalavri
Boston University

John Liagouris
Boston University

Nikos Vasilakis
Brown University

Abstract

SLOWPOKE is a new system to accurately quantify the effects of hypothetical optimizations on end-to-end throughput for microservice applications, without relying on tracing or a priori knowledge of the call graph. Microservice operators can use SLOWPOKE to ask what-if performance analysis questions of the form *"What throughput could my retail application sustain if I optimized the shopping cart service from 10K req/s to 20K req/s?"*. Given a target service and its hypothetical optimization, SLOWPOKE employs a performance model that determines how to selectively slow down non-target services to preserve the relative effect of the optimization. It then performs profiling experiments to predict the end-to-end throughput, as if the optimization had been implemented. Applied to four real-world microservice applications, SLOWPOKE accurately quantifies optimization effects with a root mean squared error of only 2.07%. It is also effective in more complex scenarios, *e.g.*, predicting throughput after scaling optimizations or when bottlenecks arise from mutex contention. Evaluated in large-scale deployments of 45 nodes and 108 synthetic benchmarks, SLOWPOKE further demonstrates its scalability and coverage of a wide range of microservice characteristics.

1 Introduction

The microservice architecture has emerged as a prevailing approach for constructing modern distributed applications [27, 31, 44, 58]. By decomposing applications into independently developed and deployed services, microservices simplify cross-team collaboration and scalable deployment [20, 27, 36]. The throughput of an application is fundamental to its scalability, cost efficiency, and quality of service under high incoming traffic, *e.g.*, traffic spikes during overlapping periods of global activity on social media platforms [17].

Improving the throughput of individual services—even with simple techniques such as vertical or horizontal scaling—does not necessarily yield end-to-end throughput improvements, due to complex service interactions in real deployments [11]. As a result, estimating how the end-to-end throughput of a microservice application would change after optimizing one or more services remains a challenging problem. A common practice is to implement selected optimizations and evaluate their performance in a canary environment [41, 47]; yet, this approach leaves the hard task of identifying the most promising changes to engineering teams. *What-if* analysis provides a principled approach to address this problem by quantifying the effects of hypothetical optimizations ahead of time and with high accuracy. Example optimizations include allocating additional resources, improving runtime efficiency through techniques such as multithreading, or migrating to new frameworks or cloud platforms; for instance, by adopting a high-capacity data store in data-as-a-service infrastructures [25].

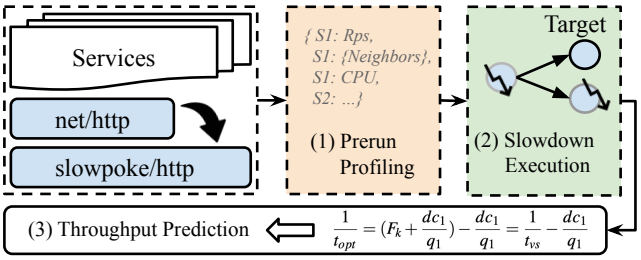


Fig. 1: SLOWPOKE overview. Given a microservice application running on Kubernetes, SLOWPOKE (1) pre-runs experiments to collect per-service information, (2) selectively slows down non-target services and measures the resulting throughput, and (3) quantifies throughput improvements using a performance model.

Existing what-if analysis techniques, such as causal profiling [16], quantify the impact of optimizing specific sections of code in short-running, multithreaded applications on *a single machine*. The core idea behind causal profiling is to introduce *virtual speedups* by carefully pausing concurrent execution threads. A virtual speedup has the same relative effect as a hypothetical speedup and can be used to accurately estimate

end-to-end performance before applying the real optimization. Unfortunately, the state-of-the-art causal profiler, Coz [16], cannot be applied to distributed settings. Coz assumes a single monolithic application and can only predict the effect of optimizing local functions by pausing application threads. On the other hand, based on distributed tracing [2, 18, 30, 45], existing microservice profiling tools [15, 29, 34, 51, 56, 57] identify critical paths and predict latency. However, without an accurate causal model and visibility into the services' internals, these approaches cannot answer what-if questions about throughput, since there is no well-defined correlation between latency and throughput.

This paper presents SLOWPOKE, a system that accurately quantifies end-to-end throughput improvements of hypothetical optimizations. SLOWPOKE introduces a performance model that estimates the throughput of a microservice application by carefully slowing down non-target services using a lightweight distributed coordination mechanism. Given a microservice application, a target service for optimization, and a predefined workload, SLOWPOKE runs experiments in a *pre-production environment*. As shown in Fig. 1, SLOWPOKE first collects per-service information in a pre-run profiling stage, then computes and applies the appropriate slowdowns at runtime, and finally predicts the resulting throughput improvement using SLOWPOKE's performance model.

Evaluated on four real-world microservice applications and 108 synthetic benchmarks that cover a wide spectrum of microservice topologies, execution models, and communication primitives, SLOWPOKE provides accurate throughput predictions, with errors ranging between -4.35–4.88% (RMSE—root mean squared error: 2.07%) for real applications, and -7.61–5.65% (RMSE: 1.89%) for synthetic benchmarks.

This paper makes the following contributions:

- SLOWPOKE, the first system for accurate optimization prediction in complex microservice architectures (§2). SLOWPOKE treats microservices as black boxes communicating via HTTP or gRPC. SLOWPOKE does not require application instrumentation or a priori knowledge of the service call graph.
- A novel performance model for quantifying end-to-end throughput improvements of hypothetical optimizations (§3). It supports multithreaded services, dynamic call graphs, and hybrid workloads with various request types. The model formalizes the prediction problem as a linear program and demonstrates the bottleneck equivalence between slowed-down and optimized executions.
- A lightweight distributed slowdown mechanism that enables coordinated pausing across complex microservices, allowing SLOWPOKE to accurately quantify the effects of hypothetical optimizations at scale (§4).

SLOWPOKE is MIT-licensed, open-source software; available at github.com/atlas-brown/slowpoke.

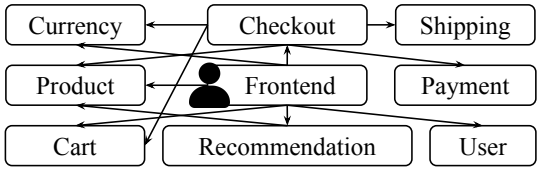


Fig. 2: Topology of OnlineBoutique. Vertices represent independently developed services that expose APIs. Edges denote various types of network communication between services.

2 Slowpoke Overview

This section first discusses the challenges of achieving accurate throughput optimization prediction for microservice applications (§2.1). It then provides an overview of SLOWPOKE and its core components (§2.2).

2.1 Motivating Example

Consider the topology of the shopping application OnlineBoutique [3] shown in Fig. 2. To place an order or perform a product search, users send HTTP requests to the `frontend` service, which communicates with other services via network protocols, such as gRPC or HTTP. As demand scales—*e.g.*, through market expansion—developers are often tasked with optimizing throughput to sustain performance under higher loads. But should the team accelerate `recommendation` queries by 30% through costly model retraining *or* improve `cart` lookups by 40% via implementing a high-capacity in-memory key-value store? Choosing the right optimization is critical for achieving meaningful end-to-end throughput gains within limited engineering and financial budgets.

Challenges and complexity: Predicting the end-to-end impact of such hypothetical optimizations before implementing them is challenging in today's complex microservice applications. For a given workload, assume that each service `S` can sustain throughput t_S when other services are sufficiently fast (*i.e.*, not bottlenecks). Throughput optimization prediction is thus reduced to accurately estimating $\min t_S$ before and after applying an optimization to a target service.

Given a microservice call graph, one approach is to estimate t_S *analytically*, by directly predicting which service would become the bottleneck after the optimization. However, this unknown throughput t_S depends on many factors, including deployment characteristics, types of requests, request rates, service states, and interactions with other services. Even with distributed tracing, the lack of visibility into service internals prevents accurate estimation of t_S.

Another approach is to estimate t_S *experimentally*, by measuring the throughput of service `S` when it is saturated, which is often impractical. When deploying `S` as part of the entire system, ensuring no other services become a bottleneck requires substantial resource over-provisioning to all other services when `S` is far from saturation. More importantly, pro-

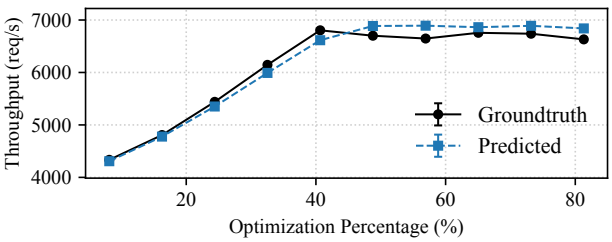


Fig. 3: Profiling results from SLOWPOKE for the `Cart` service in OnlineBoutique with mixed types of user requests. The x-axis is the percentage of optimization applied to `Cart` service, *e.g.*, 50% optimization means a 50% reduction in average execution time per request. The y-axis is the end-to-end application throughput.

visioning more resources may not even increase a service's throughput, as the service can be single-threaded or bottlenecked by mutex contention. When deploying `S` in isolation, it requires capturing a substantial amount of end-to-end request traces and developing a workload driver capable of replaying both calls to and responses from other services to stress-test `S`, at arbitrary speeds. Unfortunately, even with a sophisticated replayer, changing the timescale of captured traces is error-prone, as it depends on the behavior of other services, which are unknown.

Alternatively, while one could, in principle, simulate an optimization by provisioning additional resources to the target service, this approach is often impractical given the aforementioned problems in resource provisioning. Furthermore, it fails to accurately capture the impact of other types of optimizations, such as algorithmic enhancements or migration to a more efficient execution framework.

2.2 Optimization Prediction with Slowpoke

SLOWPOKE addresses these challenges by performing *bottleneck-equivalent* executions that selectively slow down non-target services, introducing the same relative effect as optimized executions (see §3). Bottleneck equivalence implies that the slowed-down execution will exhibit a bottleneck on the same service as the hypothetically optimized execution. As a result, SLOWPOKE can accurately estimate the effects of hypothetical optimizations on end-to-end throughput, without requiring application-level instrumentation or a priori knowledge of the throughput capacity of individual services.

To predict the effects of throughput optimizations, SLOWPOKE runs in a pre-production environment that mimics the production environment. Developers replace the communication libraries with SLOWPOKE's runtime and provide three inputs: (1) a target service, (2) its optimization, and (3) a workload consisting of end-user requests. SLOWPOKE is agnostic to the actual optimization implementations and only requires the expected performance improvement of the target service, *e.g.*, a 50% reduction in its average processing time.

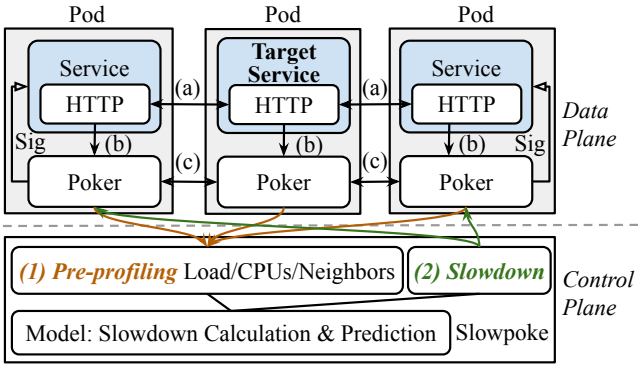


Fig. 4: Architecture of SLOWPOKE. SLOWPOKE runs on the control plane and orchestrates profiling experiments on microservice applications deployed on Kubernetes. Each microservice runs in a pod alongside a POKER process. Services interact through instrumented communication libraries via a data channel (a). During the pre-profiling phase, each POKER collects information from instrumented libraries through a standard UNIX pipe (b). SLOWPOKE aggregates this information and computes the slowdowns using its performance model. During the slowdown execution phase, the POKER co-located with the target service sends slowdown messages to its neighboring POKERs through TCP control channels (c). Other POKERs will further propagate slowdown messages and pause the execution of non-target services using POSIX signals.

Fig. 3 shows an example profiling result from SLOWPOKE for the `Cart` service in OnlineBoutique under varying optimizations and a mixed workload comprising different types of user requests (§5.1). Without actually implementing any optimization, SLOWPOKE accurately quantifies the effect of optimizing the `Cart` service on end-to-end throughput. By selecting the point on the x-axis corresponding to the expected performance improvement of the target service (even when concrete optimizations have not yet been defined), developers can use the curve to immediately obtain the predicted end-to-end throughput of the application. For example, developers can conclude that reducing `Cart`'s average processing time by 40% achieves optimal throughput, and that further optimization yields no additional benefits.

SLOWPOKE design: Fig. 4 presents an overview of SLOWPOKE. SLOWPOKE targets microservice applications deployed in containerized environments managed by Kubernetes. Its core logic runs on the control plane of the orchestration platform. Each service runs inside a container within a Kubernetes pod and communicates with other services via SLOWPOKE's communication libraries with traffic-monitoring capabilities. These libraries serve as drop-in replacements for existing HTTP and gRPC libraries, simplifying SLOWPOKE integration into existing infrastructures. In addition, SLOWPOKE deploys a lightweight controller program, POKER, alongside each microservice instance within the same container. POKER is written in C and is responsible for collecting service information, slowing down non-target services

determined by SLOWPOKE's performance model (§3), and communicating with neighboring POKERs to achieve coordinated slowdown (§4).

SLOWPOKE operates in two phases. During the *pre-profiling* phase, it performs experiments to collect information about the deployment and request load. It then feeds the collected information into its performance model (§3). The performance model determines how much to selectively slow down non-target services in order to produce a bottleneck-equivalent version of the optimized application. In the *slowdown* phase, SLOWPOKE slows down non-target services using its coordinated slowdown mechanism (§4) and measures the throughput of the bottleneck-equivalent execution. At the end of this phase, SLOWPOKE computes the predicted end-to-end throughput, as if the optimization had been implemented, and produces profiling summaries shown in Fig. 3.

Runtime orchestration: SLOWPOKE extends Go's HTTP and gRPC libraries to enable non-intrusive service instrumentation and collection of system information. These extensions are API-compatible with the standard communication libraries (Fig. 4 (a)), allowing developers to adopt SLOWPOKE with minimal integration effort .

The instrumented libraries intercept handler invocations to record two runtime metrics per service: (1) the number of requests received, and (2) its downstream services that receive outgoing requests. The first metric is used during the *pre-profiling* phase to estimate the request distribution across services and, during the slowdown phase, to inform the co-located POKER about how much slowdown to inject and when. The second metric is used during the *slowdown* phase to identify neighboring services and establish TCP connections between POKERs, which are then used for inter-POKER communication during the slowdown phase. POKER collects metrics from the communication library through a standard UNIX pipe (Fig. 4 (b)).

During the pre-profiling phase, each POKER collects service-level metadata from the instrumented communication libraries, including load statistics such as the number of calls per second, the directly connected neighbor services, and the number of allocated CPUs. This information is aggregated by SLOWPOKE to compute per-service slowdowns using its performance model. During the *slowdown* phase, SLOWPOKE restarts the deployment with slowdowns enabled by setting environment variables in the YAML configuration to inform POKER of the intended slowdown duration for its co-located service. When the target service (the one hypothetically optimized) receives a predefined number of incoming requests, its instrumented communication library notifies the co-located POKER (Fig. 4 (c)). POKER then initiates slowdown propagation by communicating with neighboring POKERs. Upon receiving the slowdown messages, POKERs co-located with non-target services further propagate messages to their neighbors and pause the corresponding service processes by sending POSIX signals.

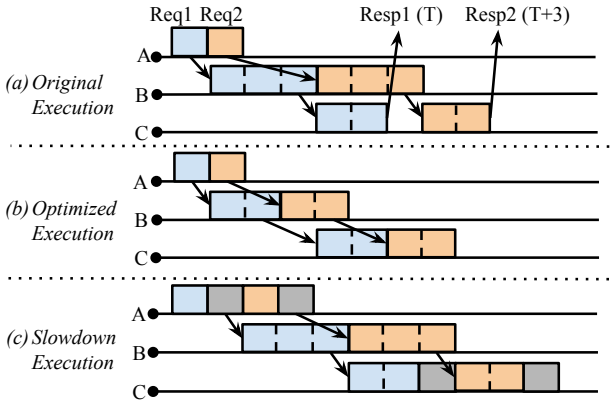


Fig. 5: Execution graphs under different scenarios. Each solid block represents the processing time of a service for a user request, *e.g.*, three time units for service `B` in (a). Blocks of the same color correspond to the same user request. Arrows represent network calls between services, triggered upon completion of the caller's processing block. For simplicity, network transmission latency is omitted. (a) Baseline execution graph: a request is completed and returned to the user every three time units. (b) Optimized execution: service `B`'s processing time is reduced from three to two time units, improving overall throughput. (c) Slowed-down execution: an artificial delay of one time unit per request is introduced in services `B` and `C`. Gray blocks indicate the induced slowdowns.

3 Slowpoke Performance Model

This section presents the mathematical model on which SLOWPOKE relies to quantify the end-to-end throughput improvement of an optimization to a target service by selectively slowing down non-target services. It begins with a simplified example for exposition purposes (§3.1). It then generalizes to (§3.2) tackle the complex nuances of real-world applications.

3.1 Model Intuition

Consider an application with three services (Fig. 5) and assume that services are single-threaded and CPU-bound—implying a reciprocal relationship between processing time and throughput. Each user request generates exactly one call to each service. Service `A` acts as the frontend: it receives user requests, processes them for one time unit, calls `B`, and waits for a response before replying to the user. Similarly, `B` processes the call for three time units and then calls `C`. Lastly, `C` processes the call for another two time units and returns. The application throughput is determined by the longest processing time among all services (*i.e.*, `B`):

$$t_{original} = \frac{1}{\max(p_A, p_B, p_C)} = \frac{1}{3} \tag{1}$$

where p_S is the average processing time per request in S.

Suppose an optimization plan optimizes target service `B` by reducing p_B by $d = 1$ time unit, as shown in Fig. 5(b). The

Tab. 1: Variables used in the model. RU can be any unit for the particular resource (*e.g.*, CPU time).

Var.	Description	Unit
t	End-to-end application throughput	Req/s
c	Number of calls per user request	1
q	Quota of a specific resource	RU/s
u	Resource consumed per call	RU/Req

application's throughput improves:

$$t_{opt} = \frac{1}{\max(p_A, p_B - d, p_C)} = \frac{1}{2} \tag{2}$$

However, this equation cannot be computed directly without knowing p_A, p_B, and p_C, which requires precise estimation of processing times of all services. Rather than directly optimizing B, SLOWPOKE *virtually speeds up* the target service by slowing down other services by d time units, *e.g.*, one time unit in Fig. 5(c). The resulting throughput is given by:

$$t_{vs} = \frac{1}{\max(p_A + d, p_B, p_C + d)} = \frac{1}{3} \tag{3}$$

Note that t_{opt} and t_{vs} are connected by $\max(p_A, p_B - d, p_C) = \max(p_A + d, p_B, p_C + d) - d$, therefore:

$$\frac{1}{t_{opt}} = \frac{1}{t_{vs}} - d = 2 \tag{4}$$

Using this relationship, the model predicts t_{opt}—the throughput resulting from the actual optimization—by leveraging t_{vs}, the *observed throughput* after artificially injecting slowdowns to non-target services, *without* requiring knowledge of service internals and the bottleneck.

3.2 Generalization

Here we generalize the model to any type of resource that has a limit per time unit. Assume there are c calls to the service for each user request, each call takes u resource, and q is the quota for such resource. u can be interpreted as the average resource consumption per call for non-deterministic and non-cumulative per-request usage, *e.g.*, memory consumption varying with the input and cache state. For any feasible throughput t, the total resource usage must not exceed the quota for any resource:

$$uct \le q \quad \text{or equivalently} \quad \frac{uct}{q} \le 1 \tag{5}$$

The units of all variables are listed in Tab. 1. Each constraint inequality represents a single resource constraint, and each service can have multiple such constraints. Perhaps unintuitively, a mutex-protected resource can also be described in this fashion. Let u be the average resource consumption—*e.g.*, CPU time—a service call requires inside the critical section, then by the nature of mutexes, *uct* needs to be less than 1 since no two threads can enter the critical section simultaneously. In practice, q may be slightly less than 1 due to locking overhead. Nevertheless, this formulation can describe the bottlenecks of various synchronization primitives.

The end-to-end throughput is the maximal t that satisfies all the resource constraints. The throughput bottleneck is the resource whose usage constraint reaches equality. This linear program is implicitly solved when measuring the end-to-end throughput of the application.

Dynamic call graphs and hybrid workload: The actual number of calls to each service varies based on the application logic. A service may be called multiple times within one user request. Service calls can also be dynamic: a service may decide whether to call another service based on the incoming request. The workload can also affect the number of calls a service receives. If a workload is a mix of two types of requests and only one of them uses the service, then the number of calls received depends on the ratio of the mixture.

The model encapsulates these behaviors by defining c as the average ratio between the number of calls received and the number of user requests. We assume the distribution of request type remains steady throughout the entire workload.

Multi-core services and general resource quota: In Section 3.1 we assumed the services were single-threaded. In this formulation, we relax that assumption by treating CPU time as a resource. Here, q becomes the CPU time the service gets per second—if 2 CPUs are dedicated to this service, $q = 2$. Accordingly, u becomes the average CPU time needed to handle each service call.

The concepts of quota and usage generalize to other types of resources as well. One notable example is mutex, where q is 1 CPU and u is time spent inside the mutex by each call to this service. This constraint exists *alongside* the regular CPU time constraint. Either constraint can become the service's bottleneck, and the linear program correctly captures that.

Bottleneck-equivalent transformation: Assuming an optimization reduces usage u_1 of service 1 by d, the throughput is then given by the solution to the new constraint set:

$$\text{maximize } t \text{ in } \begin{cases} \dfrac{u_1 - d}{q_1} c_1 t \le 1 \\ \dfrac{u_i}{q_i} c_i t \le 1 \quad (i \ne 1) \end{cases} \tag{6}$$

For simplicity, this can be rewritten as:

$$\begin{gathered} \text{maximize } t \text{ in } F_i t \le 1 \\ \text{where } F_1 = \frac{u_1 - d}{q_1} c_1 \text{ and } F_i = \frac{u_i}{q_i} c_i \quad (i \ne 1) \end{gathered} \tag{7}$$

In other words, the throughput of the optimized application t_{opt} is the solution to the linear program in Eq. 7. However, t_{opt}

cannot be obtained by solving this linear program because u_i and q_i cannot be accurately measured without extensive tracing instrumentation. Nor can it be obtained by measuring the application throughput because one cannot reduce the usage to $u_1 - d$ without actually implementing the optimization.

Similar to the transformation from Eq. 2 to Eq. 3, SLOWPOKE adds $\frac{dc_1 t}{q_1}$ to the left-hand side of all constraints, transforming the problem into

$$\text{maximize } t \text{ in } (F_i + \frac{dc_1}{q_1})t \leq 1 \tag{8}$$

This new linear program does not yield the same solution t as the one in Eq.7. Yet, it preserves the exact same bottleneck. Suppose the k-th constraint reaches equality in Eq. 7, resource k will be the throughput bottleneck in the optimized application. Given t is the same in every constraint, it also means F_k is the largest among all F_i. Consequently, the left-hand side of the k-th constraint in Eq. 8 will also be the largest among all constraints. Therefore, the k-th constraint in Eq. 8 also reaches equality. In other words, given the solution to Eq. 8, denoted as t_{vs}, one can recover the solution to Eq. 7, denoted as t_{opt}, using the following equation:

$$\frac{1}{t_{opt}} = (F_k + \frac{dc_1}{q_1}) - \frac{dc_1}{q_1} = \frac{1}{t_{vs}} - \frac{dc_1}{q_1} \tag{9}$$

Implicit solution through experiment: Expanding the definition of F_i in Eq. 8, we get

$$\text{maximize } t \text{ in } \begin{cases} \frac{u_1}{q_1} c_1 t \leq 1 \\ (\frac{u_i}{q_i} + \frac{dc_1}{q_1 c_i}) c_i t \leq 1 \quad (i \neq 1) \end{cases} \tag{10}$$

While Eq. 7 cannot be implicitly solved by throughput measurement experiments due to the negative term for constraint 1, Eq. 8 can be. By adding $\frac{dc_1}{q_1 c_i}$ seconds of "pause" to resource consumption for each service call, we can construct a version of the application whose throughput—once experimentally measured—implicitly solves Eq.8.

Measurement requirement: When using the performance model, only d, q_1, and c_i needs to be known in order to calculate the appropriate delay and predict the throughput. Although u_i and q_i are part of the model formulation, they do not need to be explicitly known, since they are implicitly accounted for during measurement.

4 Slowdown Mechanism

The mechanism to slow down non-target services is crucial for constructing the experiment for bottleneck-equivalent transformation. As pointed out in Section 3, a resource bottleneck can be preserved in the bottleneck-equivalent transformation only if its usage can be paused by the slowdown mechanism. SLOWPOKE implements slowdown by pausing services using

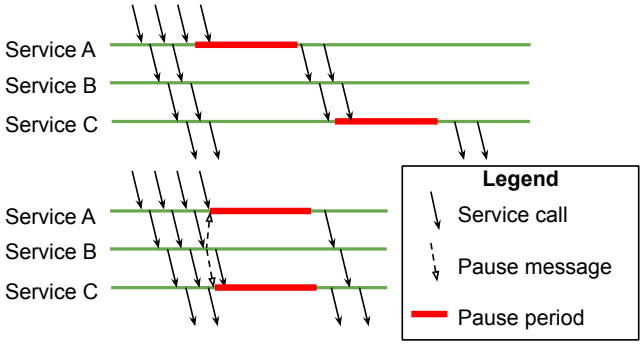


Fig. 6: Uncoordinated (top) *vs.* coordinated (bottom) pausing..

POSIX signals [4]. When launching each service, POKER starts first and spawns the service as a child process. It also sets a new process group for the service process. To pause a service, POKER first uses an unblockable `SIGSTOP` signal that pauses the service's entire process group. Then POKER sleeps for the desired duration, calculated as the time the service must remain paused according to the model. Finally, POKER sends a `SIGCONT` signal to resume the service process.

This implementation correctly preserves general CPU time bottlenecks, including those within synchronization primitives. It does not accurately preserve network or disk I/O bottlenecks because they have extensive buffering and caching inside the OS kernel.

One notable property of POKER's pausing is that it prevents all CPU usage during the pausing period, even if the service did not fully utilize the CPU outside of the period. In contrast, cgroup [23] —another potential pausing approach—controls CPU usage by counting actual total CPU time used by the service, and therefore can not slow down the service as SLOWPOKE needs, if the CPU quota is not fully used. As a result, SLOWPOKE can accurately predict throughput even when a mutex-induced bottleneck exists in other services.

Batching: POSIX signals can incur non-negligible overhead due to the need for context switching, especially when the original service has a low per-call processing time. To mitigate this overhead, SLOWPOKE accumulates pauses and executes them in batches, thereby reducing the overall overhead. We evaluate the overhead of POKER's pausing mechanism and the effect of batching in § 5.4.

Pause-induced latency: One subtle yet important scalability issue hides in the exact choice of when to perform the pauses. Suppose we implement pauses in a straightforward way: let POKER pause the local service based on the incoming requests it has seen. This uncoordinated local pausing scheme has a serious scalability problem: the average latency during profiling phase increases rapidly as the number of services on the critical path increases.

To illustrate this problem, consider the scenario shown in Fig. 6. Service `A` calls service `B`, which calls service `C`, and service `B` is the optimization target. Suppose service `A` and `C` enter

Tab. 2: Benchmark Summary.

Benchmark	Services	LOC	Sources
OnlineBoutique	9	1088	[3, 52]
HotelReservation	6	608	[20, 52]
SocialNetwork	6	532	[20, 39, 52]
MovieReview	12	913	[8, 20, 52]

the pausing period after seeing three calls. Because pausing a service stops it from making more calls, the propagation of the pause across services will be essentially serialized, causing requests to experience very high latency—proportional to the number of services along the path. Moreover, subsequent requests will also encounter serialized pauses and elevated latency, leading to a higher average latency overall.

In an ideal scenario, this will not lead to throughput degradation because the bubble in service `C` can be filled by queueing many calls before service `A` enters the pausing period to keep it busy. However, the unavoidable increase in average latency—and, by Little's law, in-flight requests—will lead to higher resource consumption (*e.g.*, memory, opened connections) and contention (*e.g.*, scheduling, memory cache), ultimately skewing the measurement of maximal throughput.

Coordinated pausing: To resolve the problem, POKER coordinates pause periods among services. Instead of each POKER pausing its service independently, the target service determines when *all* services should pause. Each POKER establishes additional control channels with its neighbors. The POKER running alongside target service initiates the pausing by sending slowdown messages to its neighbors. Using a gossip protocol, the pause eventually propagates across the entire application. When a POKER receives a slowdown message, it further propagates the message to all directly connected neighbors and pauses its co-located service for the currently batched pause duration. This decentralized approach is preferred over a global controller or pairwise connections, as it avoids all-to-all network connectivity and distributes propagation across services, thereby reducing the average overhead imposed on any single service.

The coordination is demonstrated in Fig. 6 (bottom). The pause messages coordinate services and reduce overall latency. Pause messages originating from the same initial pause at the target service are considered part of the same *round*. Each pause message includes a round number to ensure that no service pauses twice in the same round. Consider all service calls triggered by a user request as forming a graph—the service call graph. Some service calls may be delayed because their corresponding services are paused by POKER. However, assuming that slowdown messages do not travel slower than service calls, no path in the service call graph will encounter pauses for the same round. This ensures that the extra latency introduced by SLOWPOKE no longer increases with the number of services in the application.

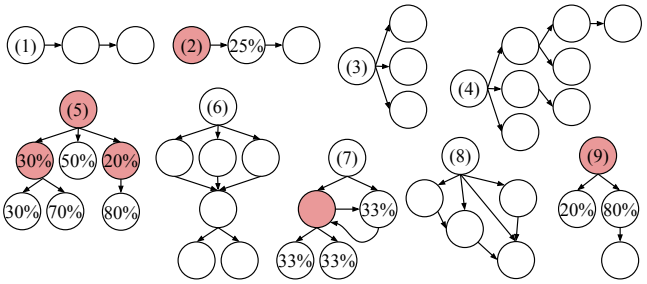


Fig. 7: Topologies of synthetic benchmarks. Each block represents a service, with arrows denoting call dependencies between services. Red blocks indicate services that issue dynamic calls, with the probability of a call reaching a callee service specified within the callee block. Topologies of synthetic benchmarks include: (1) Chain, (2) Dynamic chain, (3) Fan-out, (4) Unbalanced tree, (5) Dynamic tree with a two-level dynamic path, (6) Directed Acyclic Graph (DAG) with a relaying service, (7) Dynamic path with cycles (via different endpoints), (8) DAG with five services, and (9) Dynamic tree simulating a cache hit/miss scenario.

We evaluate the effectiveness of coordinated pausing in Section 5.4 and show that it significantly improves average latency compared to uncoordinated pausing, without requiring substantially more in-flight requests when using a closed-loop workload generator.

5 Evaluation

We evaluate SLOWPOKE's prediction accuracy across a diverse set of well-known applications (§5.1) and 108 synthetic benchmarks that capture a broad spectrum of microservice characteristics (§5.2). We further demonstrate SLOWPOKE's effectiveness in scenarios involving realistic scaling optimizations, mutex-induced bottlenecks, and large-scale microservice deployments on a 45-node cluster (§5.3). Finally, we analyze key design decisions contributing to SLOWPOKE's accuracy and examine its runtime overheads (§5.4).

Experimental setup: We conduct all experiments on Kubernetes v1.29.14 deployed on AWS EC2. Each worker node is an `m5.large` instance with 2 vCPUs (2.5 GHz Intel Xeon Platinum 8259CL), 10 Gbps network bandwidth, 8 GB RAM, and 32 GB GP3 EBS volumes. The control node and the client node—on which workload generators run on—are `m5.2xlarge` instances with 8 vCPUs, 32 GB RAM, and 32 GB GP3 volumes. All nodes run Ubuntu 22.04. The size of clusters varies from 5 to 45 nodes depending on the number of services. Each service in our experiments occupies one node due to interference between cgroup resource limits and SLOWPOKE's pausing mechanism, as discussed in the limitations section (§7). The target service initiates slowdown propagation every 100 incoming requests (batching size).

Benchmarks: Tab. 2 summarizes the four open-source applications used in the evaluation. All services communicate via HTTP. *OnlineBoutique* is an online shopping application

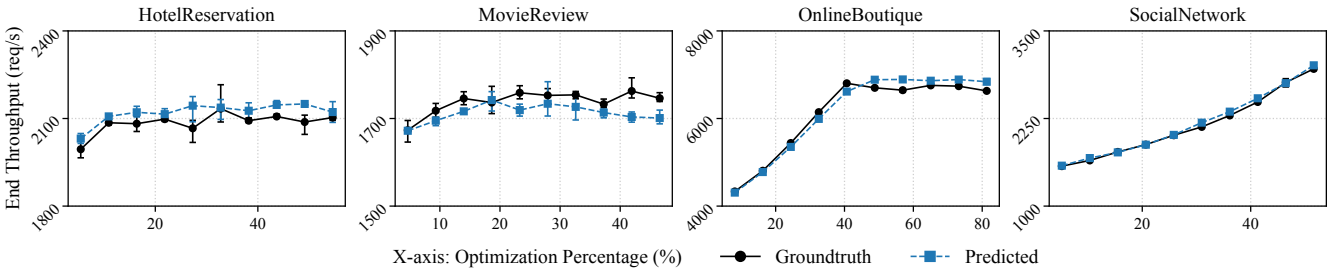


Fig. 8: Estimation errors across four well-known applications. The x-axis denotes the optimization percentage of the service's processing time, *e.g.*, a 50% optimization corresponds to a 50% reduction in the target service's average processing time per request. The range of percentages varies across applications due to differences in target service's processing times.

and its workload consists of 75% indexing requests (homepage views, product browsing, and cart interactions) and 25% operational requests (currency updates, cart modifications, and checkouts). *HotelReservation* is a hotel booking application and its workload is composed of 80% search queries for hotels in a specific area and 20% reservation requests. *SocialNetwork* is a social media application and its workload distribution includes 60% homepage timeline views, 30% personal timeline views, and 10% post submissions. *MovieReview* is a review-sharing application and its workload consists of 90% page views and 10% review submissions.

To further evaluate SLOWPOKE's accuracy across diverse microservice characteristics, we extend the Hydragen [42] microservice benchmark generator to support dynamic call graphs, expressed as probabilities of invoking downstream services. Using the extended generator, we generate 9 synthetic topologies shown in Fig. 7. Each topology is experimentally evaluated under three different configuration parameters: communication protocol (gRPC or HTTP), request concurrency (sequential or concurrent calls), and target service placement (regardless if it is an existing bottleneck), resulting in 108 total configurations. Additionally, we generate a large-scale microservice benchmark with a balanced tree topology comprising 43 services to evaluate SLOWPOKE's scalability (§5.3).

Workload generation: To generate workloads for all benchmarks, we use `wrk`, a closed-loop workload generator, configured with 8 threads and 1024 connections for real-world benchmarks. Given the diversity of synthetic benchmarks, coarse-grained connection settings can overload the system, leading to inaccurate throughput measurements. To address this, we apply a heuristic strategy that reduces the number of connections when the observed throughput is significantly lower than the expected throughput, estimated from the configured processing times. We apply a consistent workload across all phases of each experiment, including pre-profiling, slowdown execution, and optimization.

Each throughput measurement includes a 3-second warm-up phase to estimate the expected execution time for the workload, followed by a measurement phase that runs until the workload completes. We repeat each experiment five times and retain the three data points closest to the median for analysis due to the inherent variance in throughput measurements (§5.1). We calculate the error percentage based on the throughput predicted by SLOWPOKE and the ground truth, measured after applying the hypothetical optimization. Negative errors indicate underestimation, and positive errors indicate overestimation. For each group of predictions, we report both the error range (negative to positive) and the root mean squared error (RMSE) to indicate the spread.

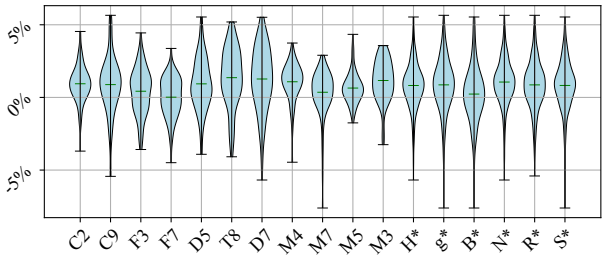
5.1 Real-world Microservice Benchmarks

This experiment evaluates SLOWPOKE's prediction accuracy across the four open-source applications listed in Tab. 2.

Methodology: To perform controlled experiments with varying optimization percentages, we introduce an artificial spinning of 1000 µs to target services and simulate optimizations by removing different fractions of the spinning time. We select target services with varying request loads and simulate optimizations by uniformly reducing the spinning time across 10 experiments, resulting in varying optimization percentages, as shown in Fig. 8. Specifically, we select `Cart` (accessed by 45% of user requests) in *OnlineBoutique*, `HomeTimeline` (70%) in *SocialNetwork*, `Profile` (80%) in *HotelReservation*, and `MovieReviews` (90%) in *MovieReview*. Each benchmark is initialized with predefined deterministic states—such as user data in *SocialNetwork* and review entries in *MovieReview*—before execution, following prior work [52].

Key results: Fig. 8 illustrates the prediction accuracy of SLOWPOKE across all applications, with error percentages ranging between -4.35–4.88% (RMSE: 2.07%). *OnlineBoutique* reports errors ranging between -2.88–4.25% (RMSE: 2.38%). *SocialNetwork* yields errors ranging between -2.19–2.89% (RMSE: 1.74%). *HotelReservation* shows errors ranging between -2.29–4.88% (RMSE: 2.16%). *MovieReview* exhibits errors ranging between -4.35–1.35% (RMSE: 1.95%).

Analysis: The results demonstrate that SLOWPOKE generalizes across diverse applications, accurately predicting throughput improvements with reasonable errors, given the inherent variance in throughput measurements. For example, the stan-



C2—Chain (length 2)
C9—Chain (length 9)
F3—Fan-out (width 3)
F7—Fan-out (width 7)
D5—DAG (5 services)
T8—Unbalanced tree
D7—DAG (7 services)
M4—Dynamic chain
M7—Dynamic tree
M5—Dynamic cycle
M3—Dynamic cache
H*—HTTP, g*—gRPC
B*—target is bottleneck
N*—target is not bottleneck
R/S*—concurrent/sequential

Fig. 9: Prediction errors for synthetic experiments grouped by different characteristics. Green lines indicate mean values.

dard deviation of ground truth (without slowdowns) throughput, measured across three runs per configuration, reaches up to 58 req/s (mean: 2134 req/s) in *HotelReservation*. High errors may stem from SLOWPOKE's overheads (§5.4) and from Eq.9, whose form can numerically amplify noises in t_{vs}.

Overall, SLOWPOKE accurately predicts performance improvements across a wide range of throughput optimizations. It predicts gains between 1565–3000 req/s in *SocialNetwork*, and between 4301–6824 req/s in *OnlineBoutique*, where the target services are critical bottlenecks. It also effectively identifies cases where optimizations yield minimal gains, as observed in *MovieReview* and *HotelReservation*.

5.2 Synthetic Benchmarks

In this experiment, we use synthetic benchmarks to evaluate the impact of diverse microservice topologies and configurations on SLOWPOKE's prediction accuracy.

Methodology: We use the 9 topologies shown in Fig. 7 to generate 108 synthetic benchmarks with varying configurations, including different communication protocols, request concurrency models, and target service placements. For example, in the chain topology, selecting the head, middle, or tail service as the target yields three distinct configurations. All service processing times are randomly generated from a Gaussian distribution with a mean of 700 µs and a standard deviation of 300 µs. For each configuration, we generate optimizations ranging from 10% to 100% of the target service's processing time, resulting in 108 $\times$ 10 experiments in total. Ground truth is obtained by directly modifying the processing time within the benchmark generator.

Key results: Across configurations, SLOWPOKE's errors range between -7.61–5.65% (RMSE: 1.89%). The heatmap in Appendix A shows that underestimations are more common at higher optimization levels, *e.g.*,-7.61% at a 90% optimization, where the target service's processing time approaches zero,

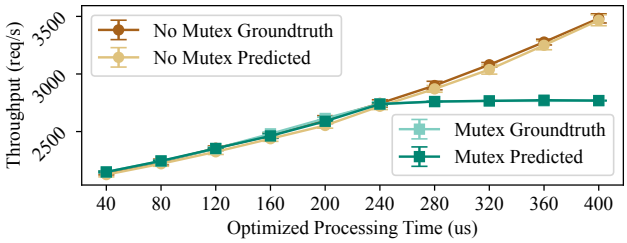


Fig. 10: Throughput prediction with mutex bottleneck.

resulting in an unrealistically fast service. As a result, the throughput measured during slowdown execution is lower due to profiling overhead, leading to underestimation.

Analysis: SLOWPOKE predicts throughput improvements across diverse microservice topologies and configurations with high accuracy, achieving error ranges comparable to those observed in real-world applications. To examine the impact of different topologies and configurations on prediction accuracy, we aggregate all results by different characteristics, including whether the target service is an existing bottleneck, as shown in Fig. 9. Out of 1080 experiments, 300 experiments have the target service as an existing bottleneck, 382 experiments result in underestimation, 540 experiments use sequential calls, and 540 experiments use HTTP protocol. SLOWPOKE maintains high prediction accuracy across all topologies, with slightly larger error ranges in scenarios involving more services and greater complexity, *e.g.*, in dynamic topology where multiple dynamic paths are exercised.

5.3 Slowpoke in Action

This experiment evaluates SLOWPOKE's prediction accuracy in three scenarios: realistic optimizations, mutex-induced bottlenecks, and large-scale deployments.

Optimization by scaling: This experiment explores SLOWPOKE's effectiveness in predicting throughput improvements under realistic optimizations, including horizontal and vertical scaling. For horizontal scaling, we scale `Cart` in *OnlineBoutique* from 1 to 2 replicas by adding an EC2 instance of the same type. For vertical scaling, we scale `Hometimeline` in *SocialNetwork* from 2 to 4 CPUs by deploying the optimized service on an `m5.xlarge` instance with 4 vCPUs. SLOWPOKE requires an expected optimization effect—specifically, by how much the throughput of the target service improves after its optimization—as input to predict the end-to-end throughput improvement. We assume a linear benefit for horizontal scaling. Given that vertical scaling does not lead to linear improvement in practice, we first estimate the throughput improvement of `Hometimeline` on 4 vCPUs by provisioning 8 vCPUs to other services, ensuring that `Hometimeline` is saturated to the best extent possible. This estimated value is used as input to SLOWPOKE in all slowdown experiments. The results show that SLOWPOKE accurately predicts the through-

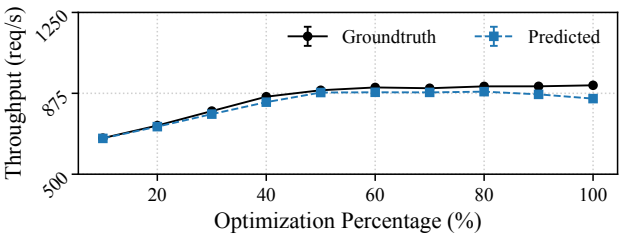


Fig. 11: Large-scale deployment with 43 services.

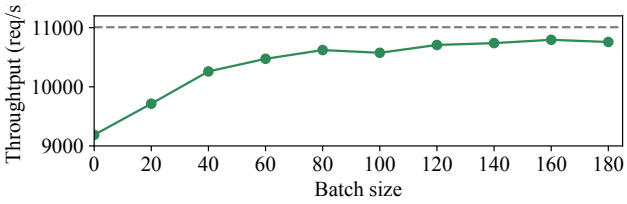


Fig. 12: Single-service throughput *vs.* batch size.

put improvement of both optimizations, with errors of **-2.41%** for *OnlineBoutique* and **-4.60%** for *SocialNetwork*.

Handling mutex-protected resources: This experiment explores whether SLOWPOKE's slowdown mechanism can correctly handle the throughput prediction of an optimized application even when the new bottleneck arises from mutex contention. We use a simple two-service application where each service exposes a single endpoint. The client calls `A` which in turn calls `B`. Each service runs on a 2-CPU machine. Within `A`'s endpoint, each request spins the CPU for 800 µs. Within `B`'s endpoint, each request spins the CPU for 350 µs while holding a mutex. We then evaluate SLOWPOKE's effectiveness on 10 different optimizations, corresponding to processing time reductions between 0 and 400 µs. For comparison, we repeat the same experiments with the mutex in `B` removed. Fig. 10 presents the actual and predicted throughputs for both the mutexed and unmutexed versions of `B`. Because both `A` and `B` run on 2 CPUs, but most of `B`'s processing is serialized by a mutex lock, the mutexed version of the application initially gains performance as `A` is optimized but plateaus at 2750 req/s when the bottleneck shifts to `B`. In contrast, when `B` is not mutexed, `A` remains the bottleneck throughout the experiments, therefore, the post-optimization throughput continues to increase. In both cases, SLOWPOKE predicts the throughput with less than 1% error across all optimizations, demonstrating that its slowdown mechanism remains effective with synchronization-induced bottlenecks.

Large-scale deployment: This experiment explores SLOWPOKE's effectiveness in a large-scale deployment comprising 43 services using a synthetic balanced tree topology (width is 6, height is 3). Each service's processing time is randomly assigned following the same procedure as in prior synthetic experiments. We select `service4` at level 2—having both parent and child services—as the optimization target, and apply 10 optimizations reducing its processing time.

Fig. 11 presents SLOWPOKE's prediction in this setup with errors ranging between -7.11–0.12% (RMSE: 1.79%). The largest error occurs under extreme optimizations, where the target service's processing time approaches zero. This experiment demonstrates that SLOWPOKE accurately predicts throughput improvements even in large-scale deployments.

5.4 Batching and Coordinated Pausing

This set of experiments evaluates SLOWPOKE's batching and coordinated slowdown mechanism.

Pause batching: We first evaluate SLOWPOKE's overhead as we vary the batching size using a simple two-service call graph. The user-facing service accepts HTTP requests and calls an inner service, which executes a busy-spinning loop for 50 µs before returning. The application's baseline throughput, without any instrumentation, is 10960 req/s. We use POKER to add a no-op slowdown to the service, meaning that it sends the signals but sets its sleep time to 0 s. As shown in Fig. 12, the service achieves a throughput of 9180 reqs/s (19.4% overhead) without batching. Increasing the batching size in steps of 20 gradually improves throughput, reaching 10795 req/s (1.93% overhead) and stabilizing at batch sizes of 80 and onwards.

Pause coordination: To demonstrate the necessity of pause coordination in SLOWPOKE, we construct three chain topologies, with 3, 6, and 9 services, each service deployed on a 2-CPU machine. The root service has a processing time of 1 ms per request, while inner services have processing times randomly selected from the range of 100–150 µs. We select the root service as the optimization target and measure throughput for a hypothetical optimization that reduces its processing time by 600 µs. Coordinated slowdown is implemented using the mechanism described in Section 4, with a batch size of 50 requests. Uncoordinated slowdown is implemented by having each POKER count received requests independently and sleep after every 50 requests.

The first experiment evaluates the impact of pause coordination on prediction accuracy using the 9-service chain topology. We fix the number of connections to 128 for coordinated pausing and gradually increase it to 352 for uncoordinated pausing, which maximizes the measured throughput. The results in Tab. 3 show that coordinated pausing results in a prediction error of only -4.3%, whereas uncoordinated pausing, even with the highest number of connections, results in a -17% prediction error.

To further illustrate the root cause of the problem, we measure the throughput of all three applications with different connection numbers (32–512), with results shown at the top of Fig. 13. Without coordinated pausing, the application requires substantially more open connections to reach peak throughput. Specifically, it needs 96, 256, and 352 open con-

Tab. 3: Prediction accuracy comparison between SLOWPOKE's coordinated and uncoordinated pausing. The ground truth throughput with 128 connections is 3541 req/s.

Setting (conns)	Tput.	Pred.	Rel. error
SLOWPOKE (128)	1680	3387	-4.35%
Uncoordinated pause (128)	956	1340	-62.2%
Uncoordinated pause (352)	1562	2939	-17.0%

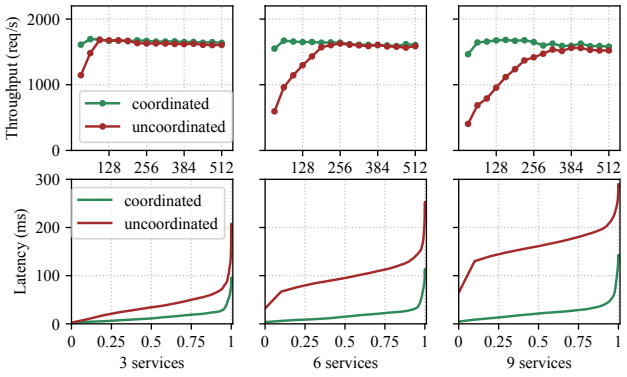


Fig. 13: Throughput and latency with coordinated and uncoordinated pausing. Top row: throughput measured with varying connection parameter in `wrk`. The applications have a simple chain topology with 3, 6, and 9 services. Bottom row: latency at different percentiles for the same set of applications.

nections for 3, 6, 9 services in the application, respectively. In contrast, with coordinated pausing, applications reach peak throughput using no more than 128 open connections. Next, we use an open-loop workload generator with a fixed request rate of 1500 req/s to record the latency distribution. As shown in Fig. 13, the 90%ile latency remains stable with coordinated slowdown, demonstrating the efficiency and scalability of the coordination mechanism. The 99%ile latency increases slightly as the number of services grows, which is expected since the coordination is not a strict global synchronization, which is infeasible in practice.

6 Related Work

SLOWPOKE is related to a large body of prior work on distributed tracing, critical path analysis, modularity, performance modeling, resource management, and service meshes.

Distributed tracing: Distributed tracing techniques have been extensively explored to gain visibility and diagnose performance issues in distributed systems [2, 9, 18, 45]. Pivot Tracing [33], Canopy [30], DeepFlow [43], 3MileBench [53], Hindsight [54], and Black-box performance analysis [10] extend distributed tracing for greater visibility in diverse environments and under different settings.

The core difference between these approaches and SLOWPOKE is that, traces alone can only capture service dependencies and the total time spent in each service, whereas the questions answered with SLOWPOKE require accurate modeling to capture the complex dependencies and behaviors *after* optimizations. The two approaches are synergistic, as SLOWPOKE could potentially leverage tracing information to extract necessary dependency information required for its analysis.

Critical path analysis: Critical path analysis aims to identify latency bottlenecks in complex distributed applications. Mystery Machine [15] and LatenSeer [56] reconstruct dependency relations based on tracing information and identify critical paths for latency, used for prediction of end-to-end latency. CRISP [57] introduces a framework for identifying critical paths in microservices and leveraging trace analysis to pinpoint sources of high latency. SnailTrail [24] generalizes critical path analysis for online analysis of long running dataflow. These latency analyses may enable throughput prediction if all the services are CPU-bound and handle requests in a sequential and synchronous way, but not in the general cases that SLOWPOKE handles. Conversely, SLOWPOKE cannot identify critical paths for latency optimizations nor predict latency improvements.

Modularity: Modular applications predate the advent of microservices [50], with prior work developing profiling techniques and exploring optimization opportunities in modular systems. Transactional profiling [14] tracks transactions across shared memory, events, or stages, and measures their interference. Active dependency discovery [13] perturbs system behavior to reveal dynamic dependencies between components. Intra-box analysis [21] introduces variable delays on network and disk traffic to confirm causal relationships between events. DiSh [35] and Fractal [26] focus on distributing opaque modular components in shell scripts to optimize performance. While effective in characterizing complex dependencies and improving performance, these approaches do not quantify the potential gains of performance optimizations. Developers could leverage SLOWPOKE to evaluate hypothetical optimizations, complementing prior profiling and optimization techniques in modular applications.

Performance modeling: Blocked-time analysis [37] uses detailed white-box logging and full execution replay to quantify I/O bottlenecks. Indy [22] predicts process performance on a single machine by tracing resource usage and calculating predictions based on available hardware resources. These systems predict potential system performance, yet require white-box visibility into resource usage, whereas SLOWPOKE implicitly takes them into account within experiments. Causal profilers [12, 51] targeting parallel programs on multiple nodes fail to directly and effectively apply to long-running microservices, which have different execution models that involve multiple concurrent requests.

Microservice resource management: Microservice resource management focuses on allocating resources efficiently while minimizing latency and cost. FIRM [38], Erms [32], and

Sinan [55] use learning-based techniques to efficiently provision resources to meet application-level latency requirements. Accelerometer [46] provides fine-grained profiling to identify resource usage hotspots in microservice environments. Autothrottle [49] uses throttle measurement to maximize efficiency of CPU provisioning in the application while meeting performance targets. Kraken [48] uses live traffic testing to help stress production systems with realistic workloads. Concord [28] employs theoretically optimal scheduling policies to improve throughput in datacenter applications. While these systems can be useful in identifying potential bottlenecks and improving performance, they cannot quantify the throughput improvements of hypothetical optimizations.

Service meshes: Existing production-grade service meshes such as Istio [6], Linkerd [7], and Cilium [5] offer robust platforms for managing microservices, featuring traffic management, security, and observability. MeshInsight [59] enhances performance analysis by profiling service meshes through inter-service communication metrics and network latency. While these systems provide valuable insights and actionable capabilities, they do not directly leverage this information to guide optimization planning. SLOWPOKE could be integrated with service meshes to leverage their rich observability and traffic control features.

7 Discussions and Limitations

Although SLOWPOKE is effective in many realistic scenarios, the current implementation may not be applicable to all possible uses encountered in production microservice deployments. This section discusses SLOWPOKE's limitations and extensions for future work.

Resource sharing across services: The SLOWPOKE performance model assumes exclusive resource use by each service. However, modern cloud deployments multiplex resources across services to improve utilization and reduce costs. Future work could extend the SLOWPOKE model to support such sharing by introducing additional constraints:

$$\left(\sum_s \frac{u_s c_s}{q_i} + \frac{dc_1}{q_1}\right) t \leq 1$$

where u_s represents the resource usage of service s, c_s denotes the service call ratio, and q_i is the total quota allocated to the shared resource. To accommodate this extension, the slowdown mechanism needs to coordinate pauses across all services sharing the same resource.

Supporting other applications: The current version of SLOWPOKE implements runtime components in Go, embedded within communication libraries. However, the underlying model (§3) and slowdown mechanism (§4) are language-agnostic. Supporting additional applications and services requires porting these components to other languages or frameworks capable of instrumenting layer-7 communications. Such extensions would broaden SLOWPOKE's applicability across complex microservices.

Compatibility with I/O-bound bottlenecks: The performance model generalizes across various types of bottlenecks, including CPU contention, mutex contention, and network saturation. However, the slowdown mechanism, implemented via `SIGSTOP` signals, has limited effects on the OS kernel. For instance, once a network packet is pushed into the kernel, `SIGSTOP` cannot halt subsequent kernel and NIC processing. For future work, network bottlenecks could be addressed by using a sidecar (*e.g.*, Istio [6]) or I/O throttling to slow down network transmission.

Compatibility with cgroup resource limiting: Container orchestration platforms such as Kubernetes and Docker Compose, rely on Linux control groups (cgroup) [23] to enforce resource quotas when specified. The CPU cgroup controls CPU quota by restricting service's total CPU time within consecutive time intervals, called "periods". The quota is refreshed at the beginning of each period. If a POKER pause begins in the middle of a period, the service might have already used up its CPU quota for that period, and would have been paused by the scheduler regardless. Such interaction can partially or fully nullify POKER's intended pause, causing the service to run much faster than expected. Notably, CPU allocation mechanisms without such "catch up" mechanisms, *e.g.*, virtual machines or CPU affinity [1], remain compatible with SLOWPOKE. Given that cgroup contains a pausing mechanism `cgroup.freeze` [23], modifying the kernel to extend the CPU quota refreshing strategy could enable cgroup to support SLOWPOKE's pausing semantics.

Autoscaling: SLOWPOKE's current model supports prediction only for services with static resource allocation. SLOWPOKE can still augment autoscaling systems [19, 40]—where resource provisioning is dynamically adjusted—by predicting the impact of optimizations, including both horizontal and vertical scaling.

8 Conclusion

SLOWPOKE quantifies the end-to-end throughput improvements of hypothetical optimizations for microservices. By introducing a novel performance model, selective service slowdowns, and an effective coordinated pausing mechanism, SLOWPOKE accurately predicts these effects across a variety of microservice applications and deployment scenarios. SLOWPOKE is MIT-licensed, open-source software; available at github.com/atlas-brown/slowpoke.

Acknowledgements

We thank the NSDI'26 reviewers for their feedback; our shepherd, Suman Nath, for his guidance; the NSDI'26 Artifact reviewers for their time and effort. We also thank

Chameleon Cloud for providing resources; Vivek Unnikrishnan, Devon Lewis, and Christina Stepin for their help in the early stages of the project; and Yuhang Song, Evangelos Lamprou, and Zekai Li for helping test artifact. This material is based upon research supported by NSF awards CNS-2247687, CNS-2312346, CNS-2237193, and CNS-2440334; DARPA contract no. HR001124C0486; a Fall'24 Amazon Research Award; a Google ML-and-Systems Junior Faculty award; and a BrownCS Faculty Innovation Award.

References

[1] CPU Affinity. https://www.gnu.org/software/libc/manual/html_node/CPU-Affinity.html. [Online; accessed April 2025].

[2] Jaeger: Open source, end-to-end distributed tracing. https://www.jaegertracing.io. [Online; accessed April 2025].

[3] Online boutique benchmark. https://github.com/GoogleCloudPlatform/microservices-demo.

[4] signal(7) — Linux manual page. https://man7.org/linux/man-pages/man7/signal.7.html. [Online; accessed April 2025].

[5] The eBPF-powered Cilium service mesh. https://cilium.io/blog/2021/12/01/cilium-service-mesh-beta/. [Online; accessed April 2025].

[6] The Istio service mesh. https://istio.io. [Online; accessed April 2025].

[7] The Linkerd service mesh. https://linkerd.io. [Online; accessed April 2025].

[8] The Movie Database (TMDB) — themoviedb.org. https://www.themoviedb.org/. [Online; accessed April 2025].

[9] Zipkin: A distributed tracing system. https://zipkin.io. [Online; accessed April 2025].

[10] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, page 74–89, New York, NY, USA, 2003. Association for Computing Machinery.

[11] Emre Baran. Performance and scalability of microservices. https://www.cerbos.dev/blog/performance-and-scalability-microservices. [Online; accessed April 2025].

[12] Zachary Benavides, Keval Vora, and Rajiv Gupta. Dprof: distributed profiler with strong guarantees. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[13] Aaron Brown, Gautam Kar, and Alexander Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *2001 IEEE/IFIP International Symposium on Integrated Network Management Proceedings. Integrated Network Management VII. Integrated Management Strategies for the New Millennium (Cat. No. 01EX470)*, pages 377–390. IEEE, 2001.

[14] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. *SIGOPS Oper. Syst. Rev.*, 41(3):17–30, March 2007.

[15] 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 Conference on Operating Systems Design and Implementation*, OSDI'14, page 217–231, USA, 2014. USENIX Association.

[16] Charlie Curtsinger and Emery D. Berger. Coz: finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, page 184–197, New York, NY, USA, 2015. Association for Computing Machinery.

[17] Anca Agape Daniel Boeve, Kiryong Ha. Throughput autoscaling: Dynamic sizing for facebook.com. https://engineering.fb.com/2020/09/14/networking-traffic/throughput-autoscaling/, 2020.

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

[19] Guilherme Galante, Luis Carlos Erpen De Bona, Antonio Roberto Mury, Bruno Schulze, and Rodrigo da Rosa Righi. An analysis of public clouds elasticity in the execution of scientific applications: a survey. *Journal of Grid Computing*, 14(2):193–216, 2016.

[20] 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.

[21] Haryadi S Gunawi, Nitin Agrawal, Andrea C Arpaci-Dusseau, J Schindler, and RH Arpaci-Dusseau. Deconstructing commodity storage clusters. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 60–71. IEEE, 2005.

[22] Jonathan C Hardwick, Efstathios Papaefstathiou, and David Guimbellot. Modeling the performance of e-commerce sites. In *27th International Conference of the Computer Measurement Group*, volume 105, page 3, 2001.

[23] Tejun Heo. Control group v2. https://docs.kernel.org/admin-guide/cgroup-v2.html, 2025.

[24] Moritz Hoffmann, Andrea Lattuada, John Liagouris, Vasiliki Kalavri, Desislava Dimitrova, Sebastian Wicki, Zaheer Chothia, and Timothy Roscoe. SnailTrail: Generalizing critical paths for online analysis of distributed dataflows. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 95–110, Renton, WA, April 2018. USENIX Association.

[25] Will Hu. Improving performance and capacity for espresso with new netty framework. https://www.linkedin.com/blog/engineering/open-source/espresso-new-netty-framework, 2019.

[26] Zhicheng Huang, Ramiz Dundar, Yizheng Xie, Konstantinos Kallas, and Nikos Vasilakis. Fractal: Fault-tolerant shell-script distribution. In *23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26)*, Renton, WA, May 2026. USENIX Association.

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

[28] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 466–481, New York, NY, USA, 2023. Association for Computing Machinery.

[29] Yurong Jiang, Lenin Ravindranath Sivalingam, Suman Nath, and Ramesh Govindan. Webperf: Evaluating what-if scenarios for cloud-hosted web applications. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 258–271, New York, NY, USA, 2016. Association for Computing Machinery.

[30] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA, 2017. Association for Computing Machinery.

[31] Shutian Luo, Huanle Xu, Chengzhi Lu, Kejiang Ye, Guoyao Xu, Liping Zhang, Yu Ding, Jian He, and Chengzhong Xu. Characterizing microservice dependency and performance: Alibaba trace analysis. In *Proceedings of the ACM symposium on cloud computing*, pages 412–426, 2021.

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

[33] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[34] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. ORION and the three rights: Sizing, bundling, and prewarming for serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 303–320, Carlsbad, CA, July 2022. USENIX Association.

[35] Tammam Mustafa, Konstantinos Kallas, Pratyush Das, and Nikos Vasilakis. DiSh: Dynamic Shell-Script distribution. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 341–356, Boston, MA, April 2023. USENIX Association.

[36] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture*. " O'Reilly Media, Inc.", 2016.

[37] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th USENIX*

Symposium on Networked Systems Design and Implementation (NSDI 15), pages 293–307, Oakland, CA, May 2015. USENIX Association.

[38] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[39] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, AAAI'15, page 4292–4293. AAAI Press, 2015.

[40] 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.

[41] Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald C Gall. We're doing it live: A multi-method empirical study on continuous experimentation. *Information and Software Technology*, 99:41–57, 2018.

[42] Mohammad Reza Saleh Sedghpour, Aleksandra Obeso Duque, Xuejun Cai, Björn Skubic, Erik Elmroth, Cristian Klein, and Johan Tordsson. Hydragen: A microservice benchmark generator. In *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*, pages 189–200. IEEE, 2023.

[43] Junxian Shen, Han Zhang, Yang Xiang, Xingang Shi, Xinrui Li, Yunxi Shen, Zijian Zhang, Yongxiang Wu, Xia Yin, Jilong Wang, Mingwei Xu, Yahui Li, Jiping Yin, Jianchang Song, Zhuofeng Li, and Runjie Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 420–437, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Yuri Shkuro. Conquering microservices complexity @uber with distributed tracing. https://www.infoq.com/presentations/uber-microservices-distributed-tracing/, 2019.

[45] 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.

[46] 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*, ASPLOS '20, page 733–750, New York, NY, USA, 2020. Association for Computing Machinery.

[47] Alexander Tarvo, Peter F Sweeney, Nick Mitchell, VT Rajan, Matthew Arnold, and Ioana Baldini. Canaryadvisor: a statistical-based tool for canary testing. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 418–422, 2015.

[48] Kaushik Veeraraghavan, Justin Meza, David Chou, Wonho Kim, Sonia Margulis, Scott Michelson, Rajesh Nishtala, Daniel Obenshain, Dmitri Perelman, and Yee Jiun Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 635–651, Savannah, GA, November 2016. USENIX Association.

[49] Zibo Wang, Pinghe Li, Chieh-Jan Mike Liang, Feng Wu, and Francis Y Yan. Autothrottle: A practical bi-level approach to resource management for slo-targeted microservices. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 149–165, 2024.

[50] Matt Welsh, David Culler, and Eric Brewer. Seda: An architecture for well-conditioned, scalable internet services. *ACM SIGOPS operating systems review*, 35(5):230–243, 2001.

[51] Adarsh Yoga and Santosh Nagarakatte. A fast causal profiler for task parallel programs. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 15–26, 2017.

[52] Haoran Zhang, Konstantinos Kallas, Spyros Pavlatos, Rajeev Alur, Sebastian Angel, and Vincent Liu. Mucache: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)*, pages 221–238, 2024.

[53] Jun Zhang, Robert Ferydouni, Aldrin Montana, Daniel Bittman, and Peter Alvaro. 3milebeach: A tracer with teeth. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '21, page 458–472, New York, NY, USA, 2021. Association for Computing Machinery.

[54] Lei Zhang, Zhiqiang Xie, Vaastav Anand, Ymir Vigfusson, and Jonathan Mace. The benefit of hindsight: Tracing edge-cases in distributed systems. In *20th*

USENIX Symposium on Networked Systems Design and Implementation (NSDI 23), pages 321–339, Boston, MA, April 2023. USENIX Association.

[55] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G. Edward Suh, and Christina Delimitrou. Sinan: ML-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 167–181, New York, NY, USA, 2021. Association for Computing Machinery.

[56] Yazhuo Zhang, Rebecca Isaacs, Yao Yue, Juncheng Yang, Lei Zhang, and Ymir Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, SoCC '23, page 502–519, New York, NY, USA, 2023. Association for Computing Machinery.

[57] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, 2022.

[58] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. Overload control for scaling wechat microservices. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 149–161, 2018.

[59] Xiangfeng Zhu, Guozhen She, Bowen Xue, Yu Zhang, Yongsu Zhang, Xuan Kelvin Zou, Xiongchun Duan, Peng He, Arvind Krishnamurthy, Matthew Lentz, Danyang Zhuo, and Ratul Mahajan. Dissecting service mesh overheads, 2022.

A Full Results for Synthetic Benchmarks

Fig. 14 shows the error heatmap across all synthetic experiments discussed in Section 5.2. Each row represents a single call graph, and each cell indicates the estimation error for a specific optimization percentage (10%–100%), where a 50% optimization corresponds to a 50% reduction in target service's average processing time per request. The color intensity reflects the magnitude of the error. Red blocks denote overestimation, whereas blue blocks indicate underestimation.

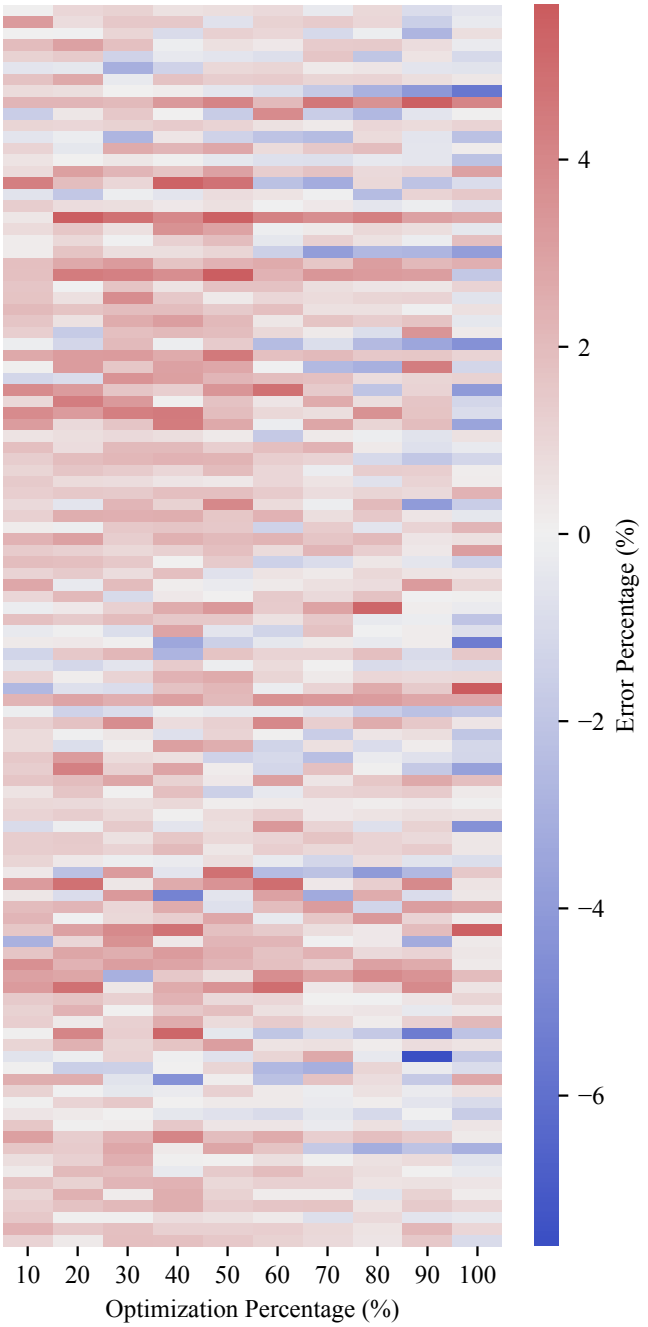


Fig. 14: Error heatmap of SLOWPOKE across all synthetic call graph configurations.