PathFence: Reducing Cross-Path Dependencies in Microservices

Xuhang Gu
Louisiana State University
Baton Rouge, USA
xgu5@lsu.edu

Qingyang Wang
Louisiana State University
Baton Rouge, USA
qwang26@lsu.edu

Abstract

Maintaining low and consistent response times is crucial for mission-critical, user-facing applications (e.g., e-commerce sites and social media platforms) that are built on microservices architectures. However, through extensive benchmarking of microservice applications in cloud environments, we find that response time stability in microservices applications is fragile, with delays ranging from milliseconds to seconds, even under moderate CPU utilization level (e.g., 60%). An important cause of this instability is cross-path dependency, where multiple execution paths, triggered by different user requests, share certain common component microservices. As a result, a slowdown in one execution path (e.g., due to a transient bottleneck) can propagate and degrade the performance of other execution paths that share the same microservices. To address this challenge, we propose PathFence, a novel approach that reduces the impact of cross-path dependencies in microservices applications by isolating workloads from different execution paths at shared microservices. By dynamically allocating software resources (e.g., thread and connection pools) with the awareness of execution path and optimizing concurrency level on shared microservices, PathFence significantly improves response time stability. Based on three real-world workload traces and three representative microservices benchmarks, we show that PathFence reduces the 99th percentile response time by up to 80% and decreases the number of dropped requests by over 90%.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **General and reference → Performance**.

Keywords

Microservices, Dependency, Resource Scheduling, Scalability

ACM Reference Format:
Xuhang Gu and Qingyang Wang. 2025. PathFence: Reducing Cross-Path Dependencies in Microservices. In *The 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25), July 20–23, 2025, Notre Dame, IN, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3731545.3731583

HPDC '25, Notre Dame, IN, USA

ACM ISBN 979-8-4007-1869-4/2025/07
https://doi.org/10.1145/3731545.3731583

1 Introduction

Motivation. In recent years, modern user-facing latency-sensitive web applications such as those at Alibaba [36], Netflix [37], and Twitter [13] have increasingly adopted microservices architectures, which decompose monolithic systems into lightweight, loosely coupled microservices. While microservices architectures offer several benefits, such as better scalability, easier cross-team development, and simpler deployment [15, 16], they also introduce complex inter-service dependencies that create significant performance challenges [41, 45].

One major challenge is the significantly increased dependencies among numerous component microservices in the system, which can cause performance instability when handling a high number of concurrent user requests. When a user request enters the microservices system, it typically triggers a series of Remote Procedure Calls (RPCs) between related microservices, forming an execution path. The request cannot be completed until all required downstream tasks (microservices) along the execution path have been executed. Therefore, if a performance anomaly (e.g., due to resource saturation) occurs in a downstream microservice, it can cause the related worker threads in all its upstream microservices to wait for the response, degrading the performance of the entire execution path. We refer to this phenomenon as *cross-service queue overflow* [47], where a performance anomaly in a downstream microservice creates a queuing effect (i.e., worker threads waiting for a response) in all its upstream microservices. Even worse, the performance degradation of one execution path can propagate and further delay other execution paths if they share microservices with the original execution path. This occurs because every microservice has a limited number of worker threads to handle all incoming requests. If the limited worker threads of a shared microservice are consumed by one execution path that experiencing performance anomaly, all other execution paths that share the same microservice will be delayed due to contention for those worker threads (more detail in Section 2).

Unfortunately, as reported in Alibaba microservices trace [36], (1) 60% of their execution paths have a depth greater than 4, and (2) 5% of microservices (called "hotspot") are shared by 90% of execution paths in their application. The extended depth of these execution paths increases the likelihood of performance anomalies, as more microservices are involved in processing each request. Additionally, the widespread sharing of microservices across various paths increases the risk of performance interference, where performance anomalies in one path can affect others. As a result, these cross-path dependencies make it more challenging to achieve consistent performance stability in microservice applications.

Limitation of state-of-the-arts. Existing studies aimed to mitigate performance instability in microservice applications can be categorized into two approaches: hardware resource management [23,

41] and software optimization [44, 45, 50]. Hardware resource management focuses on optimizing resource allocation among microservices to mitigate performance anomalies caused by resource contention. For instance, FIRM [41] introduces a dynamic hardware resource adaptation strategy that responds to bursty workloads by allocating additional resources to bottleneck microservices, thereby alleviating temporary overloads and minimizing SLO violations. Conversely, software optimization focuses on enhancing the microservice software stack to reduce computation overhead. For example, μConAdapter [32] proposed an adaptive thread pool allocation mechanism for bottleneck microservices to improve CPU efficiency when handling high concurrency.

However, both hardware resource management and software optimization have inherent limitations in addressing performance instability caused by cross-path dependencies in microservices. These approaches primarily focus on optimizing the bottleneck microservices themselves (e.g., by adding hardware resources or optimizing software stack) while overlooking the broader system dynamics, specifically the larger set of microservices that share execution paths with the bottlenecks. Although these microservices may not be the direct source of performance anomalies, they can still cause performance interference across multiple execution paths due to cross-path dependencies.

Key insights and contributions. In this paper, we first present an empirical study on the response time instability of microservices caused by cross-path dependencies, based on extensive measurements of the representative microservices application SocialNetwork [16]. We illustrate two common factors that can cause millibottlenecks (bottlenecks lasting less than a second) in runtime microservices applications: interference from internal collocated containers and bursty workloads from external users, both of which are prevalent in real-world cloud environments. Our findings show that even a minor queuing effect in a downstream microservice, triggered by a millibottleneck, can lead to cross-service queue overflow along its execution path. In cases where upstream microservices are shared across multiple execution paths, these queuing effects can further propagate delays to other dependent execution paths. Notably, due to the widespread queuing effects caused by cross-path dependencies, we observed dropped requests at the system's weakest link, leading to TCP retransmissions and response times extending up to several seconds.

To address the response time instability caused by cross-path dependencies, we propose PathFence, a novel software optimization approach that reduces the impact of cross-path dependencies in microservices applications by isolating the workload on shared microservices for different execution paths. PathFence firstly introduces execution path-aware resource management by partitioning software resources, such as worker threads, into distinct reservations, each dedicated to a specific execution path. Requests are then assigned to their respective reservations, ensuring that the performance of one execution path does not affect the performance of other execution paths. In addition, PathFence adopts an online profiling and auto-tuning framework that continuously monitors the real-time system metrics (e.g., system throughput, response time) and dynamically adjusts the reservation sizes to ensure that it always achieves the peak performance of each execution path. Overall, we make the following key contributions:

- An empirical demonstration of response time instability in microservices applications due to cross-path dependencies, showing that queuing effects originating in a downstream microservice can propagate upstream along the long execution path of dependencies and further spread to other execution paths;
- A novel approach, PathFence, that reduces the impact of cross-path dependencies in microservices applications by isolating the software resources of shared microservices for dependent execution paths;
- An evaluation of PathFence using three representative microservices benchmarks, showing that PathFence can effectively mitigate the response time instability caused by cross-path dependencies, reducing the 99th percentile response time by up to 80% and decreasing the number of dropped requests by over 90%.

Limitations of the proposed approach. We conclude two limitations of PathFence that can be improved. First, PathFence is a software optimization approach that requires awareness of the workload type of different requests (e.g., direct downstream microservice) in a component microservice. Thus, we need the tracing information of each type of requests to collect all the invoked microservices in the execution path. The updates of the application (e.g., adding new microservices or changing the execution path) may require the re-tracing and re-implementation of the system. Second, the scheduling of software resources requires real-time profiling of system metrics (e.g., system throughput, response time, and concurrency level), the robustness of the optimization of software resources may be affected by the accuracy of the profiling mechanism.

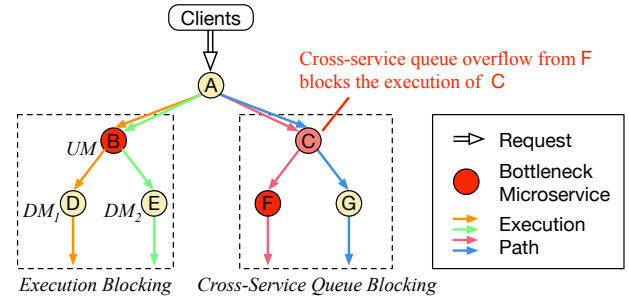
2 Background and Motivation

2.1 Blocking Effects in Microservices

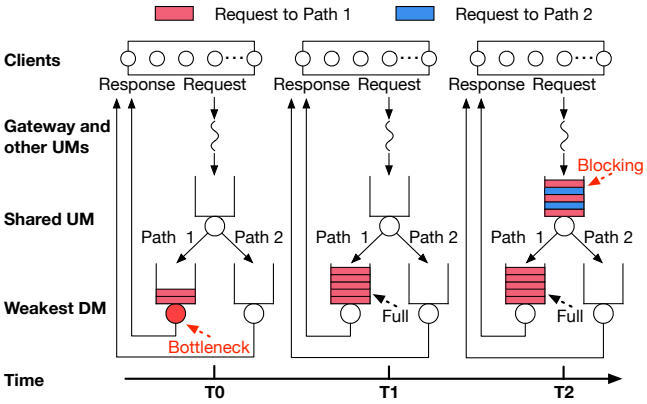
Microservice applications decompose the monolithic design of business logic into loosely coupled microservices, introducing complex inter-service dependencies. This architectural shift presents a significant challenge in maintaining system-wide performance stability: resource contention in a single component microservice can block the execution of other microservices, leading to *blocking effects* that this paper refers to, causing cross-path dependencies.

Blocking Effects within an Execution Path. A user request triggers a series of RPC calls to multiple microservices, forming an execution path. Each RPC call connects an *upstream microservice* (UM) to a *downstream microservice* (DM). Due to the existence of dependent tasks in the execution path, a microservice may not perform its next task (e.g., call its downstream microservices or reply its upstream microservices) until receiving replies from its downstream microservice. As a result, resource saturation in any microservice can disrupt the execution flow, blocking the execution of both upstream and downstream microservices along the execution path.

Blocking Effects across Different Paths. In addition, blocking effects can propagate across multiple execution paths when they share common microservices, amplifying the impact of resource saturation. Fig. 1a illustrates two types of blocking effects across different paths: *execution blocking* and *cross-service queue blocking*.



(a) Execution blocking and cross-service queue blocking.



(b) Conceptual illustration of cross-service queue blocking. At $T0$, a bottleneck starts on the weakest DM in Path 1, filling up its queue. At $T1$, the bottleneck DM reaches its queue limit, causing queue overflow to the UM. At $T2$, the queue overflow propagates to the shared UM, blocking the execution of all dependent DMs in Path 2.

Figure 1: Cross-path dependencies caused by blocking effects in microservices applications.

First, when resource saturation occurs on a shared microservice, it can directly block its own request processing and the calls to downstream microservices. As a result, the execution of all the execution paths that share the microservice will be blocked, which is called execution blocking. For example, if microservice-B experiences resource saturation, it will block the load of microservice-D and microservice-E. Second, when resource saturation occurs on a local microservice of one execution path, it can indirectly block the execution of other execution paths by occupying the shared upstream resources. This is because resource saturation on the bottleneck microservice will cause message queues and thread pools to fill up. Once the local queues reach capacity, requests start to overflow to their upstream microservices, leading to a phenomenon called Cross-Service Queue Overflow. As a result, subsequent requests from other execution paths that share the same microservices also become blocked, which is called cross-service queue blocking. For example, if microservice-F experiences resource saturation, it will block the load of microservice-C once its queue overflows. In addition, the load of all DMs (e.g., microservice-G) will also be blocked due to the queued requests at the shared upstream microservice (microservice-C).

Execution blocking occurs when resource saturation happens directly on a shared upstream microservice. In such a scenario, the shared upstream microservice becomes a bottleneck, blocking the execution of all dependent downstream microservices across different execution paths. In practice, microservices operators can identify the shared upstream microservices and take proactive measures to mitigate execution blocking. For example, they can allocate more computational resources to those shared upstream microservices, such as increasing CPU and memory limits. However, cross-service queue blocking is more challenging to detect and mitigate because resource saturation and the resulting blocking effects occur on different microservices. In such a scenario, resource saturation can happen on any DM in one execution path and block the execution of other execution paths. Simply increasing the computational resources of the shared UM could not solve the problem. Given the fact that microservice applications usually have (1) long execution paths with multiple microservices and (2) multiple execution paths that have shared microservices, cross-service queue blocking is severe and common in microservices applications. In this paper, we focus on the cross-service queue blocking effects in microservices applications and investigate the impact of these effects on response time instability.

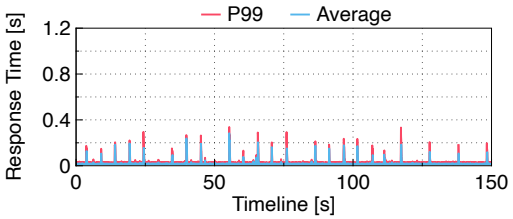
2.2 TCP Retransmission Timeout

When a microservice encounters blocking effects, its application-level queue (e.g., thread pool) begins to fill up. Once the queue reaches capacity, the microservice can no longer accept new requests, causing incoming requests to be placed in its TCP backlog. If additional requests continue to arrive, the TCP backlog will eventually overflow. At this point, any additional requests will be dropped, resulting in TCP retransmissions. Typically, the TCP retransmission timeout (RTO) is calculated based on the round-trip time (RTT) between the sender and receiver. The minimum RTO is set to 1 second and doubles with each subsequent retransmission [40].

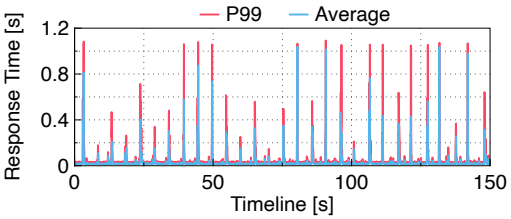
In traditional monolithic n-tier systems, operators typically mitigate TCP backlog overflow by limiting the number of concurrent connections from an upstream tier to a given service. However, in microservice applications, setting an appropriate limit on upstream concurrent connections becomes significantly more challenging due to the diverse and often large number of upstream ingress microservices [46]. For example, Alibaba reports that more than 5% of its production microservices have more than 16 ingress upstream microservices [36]. In such scenarios, the total volume of incoming requests can far exceed the capacity of a single microservice to handle concurrent requests, leading to TCP retransmissions.

2.3 Motivation Experimental Results

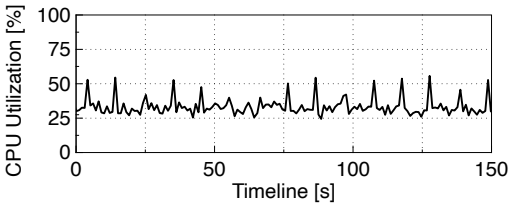
To highlight how the cross-path dependencies in microservices can cause response time instabilities, we conduct a series of experiments using the SocialNetwork benchmark from DeathStarBench [16]. We create very short internal colocation interference (millibottleneck) to a single component microservice and observe the response time instability of the entire microservices application. More details of the experimental setup can be found in Section 3.



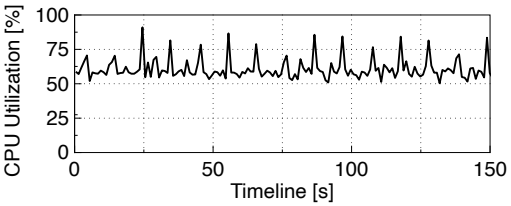
(a) Moderate response time fluctuations at 4000 clients



(b) Large response time fluctuations at 8000 clients

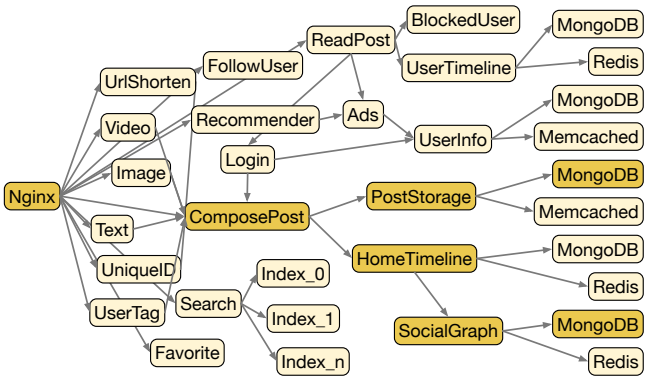


(c) Low CPU util. of bottleneck microservice at 4000 clients

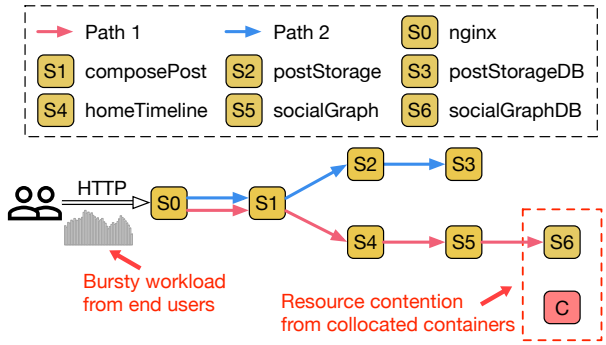


(d) Moderate CPU util. of bottleneck microservice at 8000 clients

Figure 2: Interference on a single microservice causes system wide response time instability due to the cross-path dependencies. (a) and (c) show the results with 4,000 clients, while (b) and (d) show the results with 8,000 clients, where both cases are far away from resource saturation.



(a) The Architecture of SocialNetwork.



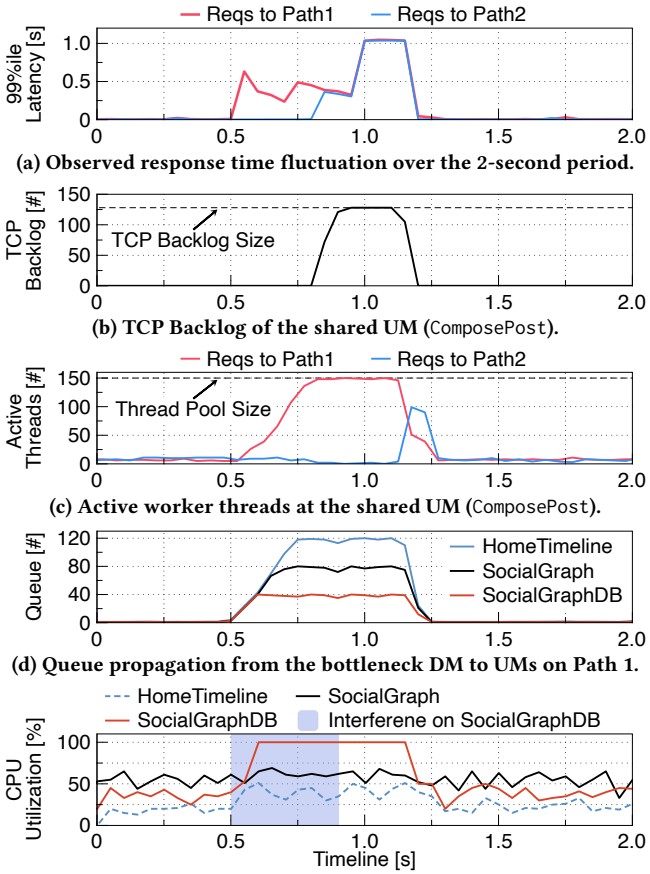
(b) Two representative execution paths in SocialNetwork. External users may generate bursty workloads, and an internal microservice may be collocated with microservices from other execution paths or applications on the same host, leading to performance interference.

Figure 3: SocialNetwork benchmark execution dependency graph and experimental setup.

Fig. 2a and Fig. 2b show the 99th percentile and average end-to-end response time of the system under moderate and high workloads, respectively. Fig. 2a shows that the system exhibits some response time fluctuations at a workload of 4,000 normal users. Fig. 2b illustrates more significant response time fluctuations, ranging from milliseconds to over a second, even though the average CPU utilization remains around 60%, well below saturation, making it difficult to identify the root cause. Due to the cross-path dependencies, system-wide performance instability is caused by the very short resource contention (millibottleneck) within a single microservice. Such large response time fluctuations can degrade user experience and potentially result in significant business revenue loss [28, 29]. In the following section, we further analyze two common factors contributing to millibottlenecks and large response time fluctuations: internal colocation interference and external bursty workloads.

3 Key Observations

In this section, we analyze the system behavior under two factors: internal colocation interference and external bursty workload, using SocialNetwork benchmark application [16]. SocialNetwork is a microservices application that simulates a social media platform. The benchmark consists of more than 30 microservices (as shown in Fig. 3a), each responsible for a specific function, such as user authentication, posting messages, or generating social graphs. The microservices communicate with each other through RPC calls. We deploy the benchmark on a Kubernetes cluster, where each microservice runs in its own container. To manage resource allocation dynamically, we leverage Kubernetes' auto-scaling mechanism, which adjusts the number of replicas for each microservice based on CPU utilization. Specifically, the auto-scaling policy is configured to scale out when CPU utilization exceeds 70% for 30 seconds, and to scale in when utilization falls below 30% for 30 seconds. To simulate user behavior, we use a closed-loop RUBBoS workload generator [3] that emulates a configurable number of external users navigating the website. We choose two representative execution paths to demonstrate the results, as shown in Fig. 3b. Each



(a) Observed response time fluctuation over the 2-second period.

(b) TCP Backlog of the shared UM (`ComposePost`).

(c) Active worker threads at the shared UM (`ComposePost`).

(d) Queue propagation from the bottleneck DM to UMs on Path 1.

(e) CPU utilization of the microservices along Path 1. A millibottleneck is observed in the downstream microservice `SocialGraphDB`.

Figure 4: Fine-grained analysis of the experiments over a 2-second time span. A millibottleneck caused by interference from a collocated container at the downstream microservice `SocialGraphDB` (e) triggers queue propagation to upstream services (d). Requests along Path 1 dominate worker threads (c), leading to the TCP backlog filling up on the upstream microservice `ComposePost` (b). This results in prolonged response times, as seen in (a), due to TCP retransmissions.

execution path involves a series of microservices that process a specific type of user request. In addition to the gateway microservice, the two execution paths share a common upstream microservice (`ComposePost`). In the following, we will demonstrate how the performance degradation from one execution path can propagate to another execution path due to cross-path dependencies. We first present two common factors that contribute to millibottlenecks in microservices runtime environments: internal interference from collocated containers and external bursty workloads from users, both of which are prevalent in real-world cloud environments. The results indicate that the impact of a millibottleneck on a downstream microservice is not limited to its own execution path, but can propagate to other execution paths, causing system wide response time instabilities We then show that the system capacity (maximum throughput) is limited by the weakest execution path.

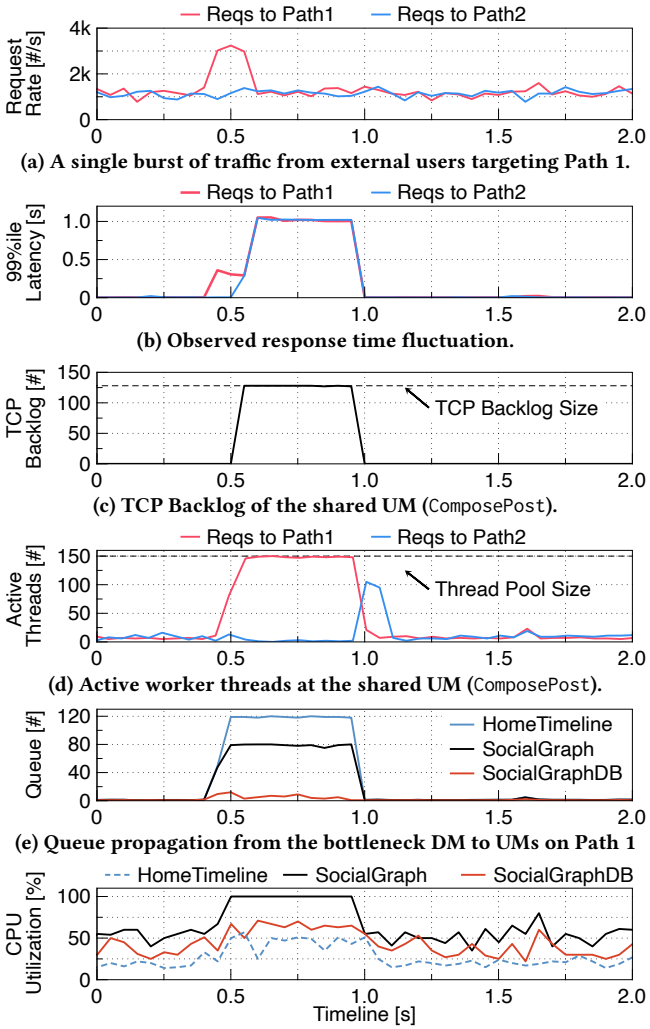
To induce internal colocation interference, we explicitly colocate a container running CPU-intensive computation tasks alongside containers hosting the benchmark microservices application. These tasks are CPU-bound and consume 100% of the available CPU resources. In real-world cloud environments, such CPU saturation is common, as containers from different tenants or applications are scheduled independently and may aggressively utilize their allocated CPU quotas to maximize performance or resource usage efficiency. To induce external bursty workloads, we use the Httperf [2] benchmarking tool to generate bursts of traffic by sending a sudden influx of requests to the microservice application, mimicking spikes in user activity.

3.1 Millibottlenecks Caused by Internal Colocation Interference

Internal colocation interference arises when multiple microservices share the same physical host, leading to resource contention. Microservices are typically deployed as containers, which are lightweight, portable units that package an application along with dependencies [27]. These containers are often hosted on shared or multi-tenant compute resources, such as virtual machines (VMs) in cloud environments. When multiple containers are collocated on the same host, they must compete for critical resources like CPU, memory, and network bandwidth. This competition can cause resource contention and degrade performance, particularly when one or more containers aggressively consume shared resources [35, 41].

To demonstrate the impact of internal colocation interference on response time, we explicitly colocate a container running compute-intensive tasks with the container hosting the `SocialGraphDB` microservice. `SocialGraphDB` is a downstream microservice in execution Path 1, which is not shared with the other execution path (Path 2), as illustrated in Fig. 3b. Fig. 4 shows a fine-grained timeline analysis (with a 50ms time interval) over a 2-second time span of the experiment. Fig. 4e shows the CPU utilization of the microservices along Path 1. Interference from the collocated container causes a millibottleneck in the `SocialGraphDB` microservice. As a result, the local queue of `SocialGraphDB` quickly fills up during the millibottleneck period, causing requests to queue in the upstream microservices (`SocialGraph` and `HomeTimeline`), as shown in Fig. 4d. When the queued requests overflow into the shared upstream microservice `ComposePost`, its thread pool becomes saturated with waiting requests from Path 1, as shown in Fig. 4c. Consequently, no idle worker threads are available to process incoming requests, affecting both Path 1 and other execution paths. This leads to the TCP backlog of `ComposePost` filling up, as shown in Fig. 4b. Once the TCP backlog reaches the limit (typically 128), additional requests will be dropped, triggering TCP retransmissions. As a result, the response time for the dropped requests will exceed 1 second. Fig. 4a shows the response time fluctuations during the interference period.

In this representative experiment, it is challenging to set an optimal queue size for the microservice `ComposePost`, as it has more than 7 ingress upstream microservices (see Fig. 3a). The total incoming requests can far exceed the number of concurrent requests that `ComposePost` can handle, particularly when some queue slots are occupied by downstream microservices during millibottlenecks.



(a) A single burst of traffic from external users targeting Path 1.

(b) Observed response time fluctuation.

(c) TCP Backlog of the shared UM (`ComposePost`).

(d) Active worker threads at the shared UM (`ComposePost`).

(e) Queue propagation from the bottleneck DM to UMs on Path 1

(f) CPU utilization of the microservices along Path 1. The burst triggers a millibottleneck in the microservice `SocialGraph`.

Figure 5: An illustration of how a single burst of traffic triggers a millibottleneck, leading to performance degradation that propagate across different execution paths.

As a result, all the requests (regardless of which execution path they will reach) that traverse through the shared microservice `ComposePost` will be impacted by the downstream millibottlenecks caused by the interference from the collocated containers.

3.2 Millibottlenecks Caused by External Bursty Workload

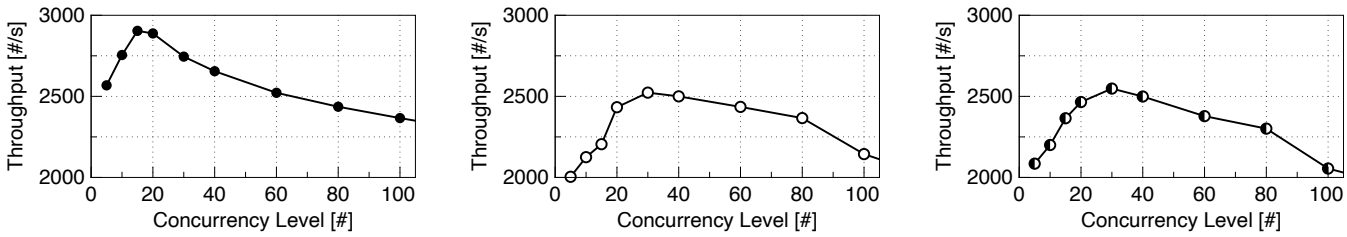
External bursty workloads arise when a microservice application experiences a sudden increase in incoming requests, temporarily overloading the system and causing response times to spike. Bursty workloads are common in user-facing applications, such as e-commerce sites, social media platforms, and search engines, where user traffic can fluctuate significantly throughout the day [14, 44]. These workloads can lead to unpredictable response times and overall performance degradation, especially when the system is already under heavy load [31].

To demonstrate the impact of external bursty workloads on the response time, we send a small volume burst of traffic with one type of user request to the microservice application, traversing through the execution Path 1 (see Fig. 3b). The burst of traffic quickly reaches the weakest microservice (`SocialGraph`), creating a millibottleneck. Fig. 5 shows a representative 2-second snapshot, with each metric monitored at a fine-grained 50ms interval. Fig. 5a shows a burst of traffic that suddenly reaches execution Path 1 at around 0.5s. This burst causes transient CPU saturation (millibottleneck) at the weakest microservice (`SocialGraph`) along the path, as shown in Fig. 5f, leading to requests queuing. The local queue quickly fills up during the millibottleneck period, causing requests to overflow into the upstream microservice (`HomeTimeline`), as shown in Fig. 5e. Similar to the internal colocation interference scenario discussed in Section 3.1, when the queued requests overflow into the shared upstream microservice `ComposePost`, its thread pool becomes saturated with waiting requests from Path 1 (Fig. 5d). As a result, no idle worker threads remain to handle incoming requests from either path, leading to the TCP backlog of `ComposePost` filling up, as shown in Fig. 5c. Once the backlog reaches its limit, additional requests are dropped, triggering TCP retransmissions and resulting in significantly prolonged response times for those requests, as shown in Fig. 5b. These observations demonstrate that even a single bursty workload can substantially degrade system-wide performance.

3.3 System Capacity Limited by the Weakest Execution Path

System capacity (maximum throughput) is the maximum number of requests that an application can process per unit time, which is usually used to measure how fast a system can handle the incoming workload. To achieve the maximum throughput, practitioners typically optimize the concurrency level (i.e., number of concurrent requests) of a system [33, 45]. A higher concurrency level can improve throughput by allowing the system to process more requests concurrently. However, if the concurrency level is set too high, it can lead to resource contention, where requests compete for limited resources such as CPU, memory, and network bandwidth. This contention can result in increased queueing delays, degraded performance, and, ultimately, longer response times. On the other hand, setting the concurrency level too low can result in underutilization of system resources, thereby limiting throughput and efficiency of the system.

To maximize the throughput of a microservice system, practitioners typically conduct offline tuning experiments [48] or sandbox testing [27] to determine the optimal concurrency level for each component microservice. However, these approaches do not differentiate between workload types for requests on microservices that are shared across multiple execution paths. In this work, we observe that when handling mixed types of workloads, the overall throughput of a shared upstream microservice is limited by the weakest execution path. As a result, the shared microservice becomes underutilized once the weakest execution path reaches



(a) Requests to Path 1 only (b) Requests to Path 2 only (c) Half requests to Path 1 and half to Path 2

Figure 6: Optimal concurrency level for `ComposePost` shifts as the workload distribution changes. The performance is limited by the weakest execution path when the microservice serves multi-type of requests as shown in (c).

its capacity, limiting the throughput of other execution paths and preventing further adjustments to the optimal concurrency level.

To investigate throughput limitations and the shifting of optimal concurrency levels on a shared microservice, we conduct a series of experiments under different workload distributions. These experiments use the same benchmark as shown in Fig. 3, where the upstream microservice `ComposePost` is shared by two execution paths (Path 1 and Path 2). Fig. 6 shows how the optimal concurrency level of the shared microservice `ComposePost` varies under different workload distributions. When there are only requests sent to Path 1, an optimal concurrency level of 15 can achieve the maximum throughput (around 2900 req/s), as shown in Fig. 6a. However, when all requests are sent to Path 2, the optimal concurrency level shifts to 30 and the maximum throughput is around 2600 req/s, as shown in Fig. 6b. In addition, when the workload is evenly distributed between Path 1 and Path 2, the optimal concurrency level and maximum throughput are similar to the case when all requests are sent to Path 2. This indicates that the overall throughput of the shared upstream microservice is limited by the weakest execution path (Path 2). As a result, the resource utilization of the microservices on Path 1 is underutilized. Therefore, to achieve the maximum throughput of a microservice system, it is essential to optimize the concurrency level of the shared upstream microservice with awareness of the workload types of the requests.

4 Design and Implementation

PathFence is designed to reduce the impact of cross-path dependencies by scheduling the software resources of a microservice with the awareness of the execution paths of incoming requests. To achieve it, PathFence divides the software resources (e.g., worker threads and database connections) of a microservice into multiple reservations. Each reservation is assigned to a specific execution path, allowing the microservice to handle requests of that path independently. Then PathFence dynamically optimizes the number of concurrent requests to the level that achieves the peak throughput for each execution path. This approach ensures that the system can handle the maximum number of requests without causing software resource contention, thereby eliminating the impact of a performance anomaly in one execution path on other execution paths.

Asynchronous invocations between microservices are considered as a common practice to reduce the inter-service dependencies and improve the performance of microservices applications [47], which typically use event-driven processing models. In such architecture, any idle thread handles a request upon event notification, meaning that threads are not associated with specific requests [45]. However, despite their advantages, asynchronous invocations are not always practical in real-world deployments. First, asynchronous invocations require application developers to explicitly manage request state, synchronization, and potential race conditions [9]. Implementing asynchronous invocations also requires additional effort and introduces additional complexity to the system. For example, Apache Thrift requires an additional library libevent [6] to handle asynchronous events. Moreover, debugging asynchronous systems is inherently more difficult, as the control flow is non-linear and issues such as race conditions or deadlocks may not manifest consistently [51]. Asynchronous operations can also lead to unexpected latency under high load, especially when underlying system resources like I/O buffers or threads are saturated [26]. In such cases, the intended performance benefits may be negated, and critical user-facing operations may experience delays.

The design goal of PathFence is to reduce the impact of cross-path dependencies in microservices applications, similar to the benefits of asynchronous invocations, while still using synchronous invocation mechanisms. In such a way, PathFence minimizes implementation complexity and can be easily integrated into existing microservices applications with minimal additional programming effort.

4.1 Scheduling Software Resources with Execution Paths Awareness

PathFence improves microservice performance stability in microservices by isolating software resources (e.g., thread pool and database connections) into multiple reservations, each dedicated to a specific execution path. This design prevents resource contention across different execution paths and effectively reduces cross-path dependencies.

Fig. 7 shows an example of thread pool design to illustrate the software resource management of PathFence. When a request enters the microservice, it is first placed in a task queue, awaiting processing by a worker thread. To identify the execution path of each request, PathFence checks the downstream microservice (DM) that the request is sent to. If no downstream microservice is determined, the request is considered to be a local request. Since the local requests only require worker threads processing without additional I/O communication overhead, PathFence schedule those

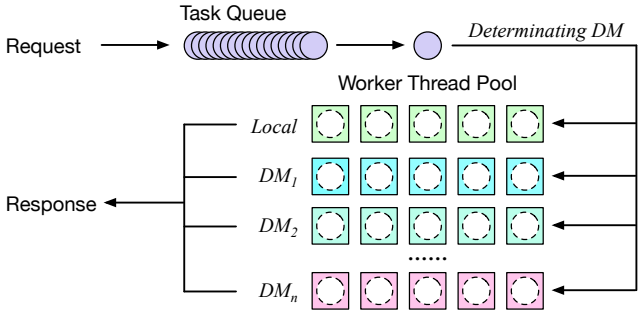


Figure 7: Thread pool design of PathFence. We identify the execution path of each request by its downstream microservice. Each execution path is assigned with a pool reservation.

Algorithm 1: PathFence's Software Resource Management

```
   /* Initialization */
1  sum_permit_DMs ← 0
2  for DM_id in DM_ids do
3      semaphore_DM.set(DM_id, permit_DM_id)
4      sum_permit_DMs ← sum_permit_DMs + permit_DM_id
5  semaphore_local.set(pool_size − sum_permit_DMs)
   /* Acquire a semaphore for a request type */
6  acquire(type):
7      if type = local then
8          if semaphore_local.tryAcquire() then
               /* Acquired from local reservation */
9          else
10             target_DM ← DM with fewest active threads
11             semaphore_DM.acquire(target_DM)
12     else if type = DM then
13         semaphore_DM.acquire(DM_id)
   /* Release a semaphore after execution */
14 release(type):
15     if type = local then
16         semaphore_local.release()
17     else if type = DM then
18         if semaphore_local.getQueueLength() > 0 then
19             semaphore_local.release()
20         else
21             semaphore_DM.release(DM_id)
```

requests with a higher priority than those that need to be sent to downstream microservices.

To manage the allocation of software resources, PathFence employs a semaphore-based algorithm to regulate the access for different execution paths. The semaphore-based algorithm is implemented by extending the Java Semaphore class, as shown in Algorithm 1. It includes two functions: `Acquire()` acquires a semaphore for a specific request type, and `Release()` releases the semaphore

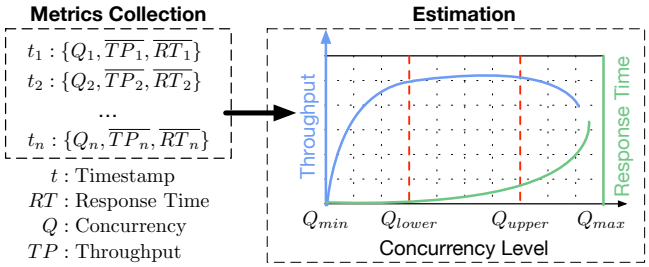


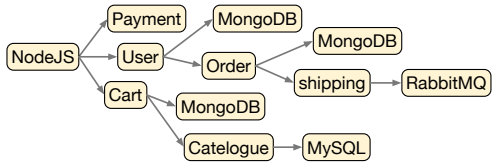
Figure 8: SCT model overview.

after the request has been processed. The algorithm only prioritizes local requests to prevent them from being blocked by other requests (e.g., those waiting for an I/O response). However, it can be easily extended to adjust priorities based on downstream microservices' performance and priority requirements.

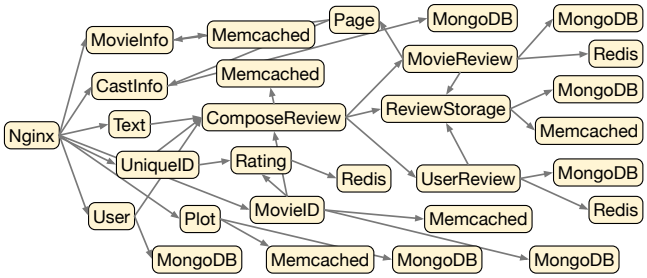
During initialization, PathFence sets up a semaphore for each execution path and assigns a concurrency limit (*permit_DM_id*) to control the number of worker threads allocated to that path. The optimal value of *permit_DM_id* is determined by the SCT model, which is described in Section 4.2. When a request arrives, PathFence assigns a worker thread from the thread pool based on the downstream microservice that the request targets. If a downstream microservice is identified, PathFence acquires a semaphore from the corresponding reservation to ensure that resource allocation respects the concurrency limit for that execution path. For local requests, PathFence first attempts to acquire a semaphore from the local reservation. If the local reservation is full, it instead acquires a semaphore from the other reservations with the fewest active worker threads. This prioritization ensures that local requests are not blocked by others (e.g., those waiting for an I/O response). After processing a request, PathFence releases the allocated resource. It first attempts to release the semaphore back to the local reservation if there are queued local requests waiting for a permit. Otherwise, it returns the semaphore to the corresponding reservation.

4.2 Optimizing Concurrency Level for Each Execution Path

PathFence dynamically adjusts the concurrency level for each execution path based on real-time system conditions, such as external workload distribution and internal hardware scaling. To achieve this, we implement the Scatter-Concurrency-Throughput (SCT) model, an online management framework that effectively recommends and reconfigures the concurrency level for n-tier systems [33]. We adapt the SCT model to the microservices environment, integrating it with our PathFence software resource management design. According to the Utilization Law in Queuing Theory, a server's throughput increases with the number of concurrent requests until it reaches saturation. At saturation, the throughput reaches its maximum and remains constant until it decreases due to non-trivial multithreading overhead, while the response time increases nearly linearly with the number of concurrent requests. The optimal concurrency level is the number of concurrent requests that achieves peak throughput without causing the server to become saturated. Therefore, SCT defines the minimum concurrency level



(a) The Architecture of SockShop.



(b) The Architecture of MediaService.

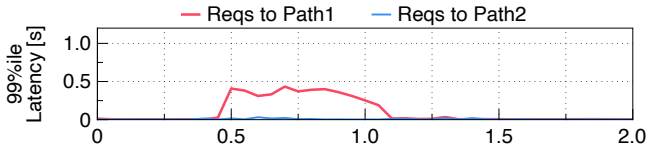
Figure 9: Topology of SockShop and MediaService.

that achieves the maximum throughput as the lower bound of a rational concurrency setting, denoted as Q_{lower}, and the maximum concurrency level that achieves the peak throughput as the upper bound of a rational concurrency setting denoted as Q_{upper}.

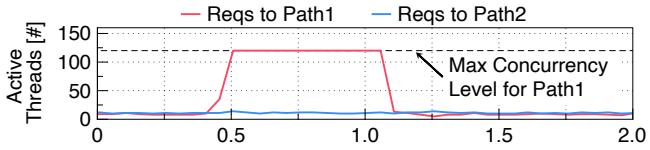
SCT dynamically optimizes concurrency levels by continuously monitoring system runtime metrics, such as throughput, response time, and concurrency, to predict the optimal concurrency level for each execution path, as shown in Fig. 8. To collect the runtime metrics of each execution path, PathFence only records the arrival and departure timestamps of each request within each microservice at millisecond granularity. These timestamps are then processed to calculate a series of tuples of each type of request (Q_i, TP_i, RT_i) during a short time window (e.g., 3 minutes). Where Q_i is the concurrency level, TP_i is the throughput, and RT_i is the response time of the type of requests. Since the real-time concurrency level typically changes, we use $[Q_{min}, Q_{max}]$ to denote the range of the concurrency level collected in the Metrics Collection Phase. Given each tuple, we can calculate the average throughput $\overline{TP_n}$ and the average response time $\overline{RT_n}$ for each concurrency level Q_n in the range $[Q_{min}, Q_{max}]$ and estimate the rational concurrency range $[Q_{lower}, Q_{upper}]$. To have a low response time while maintaining the maximum throughput, we set the concurrency level to the lower bound (Q_{lower}) of the rational concurrency range. If the concurrency level is lower than Q_{lower}, the throughput is not maximized. If the concurrency level is higher than Q_{lower}, the response time increases nearly linearly with the concurrency level. Therefore, we set the concurrency level to Q_{lower} to achieve the peak throughput while maintaining a low response time.

4.3 Implementation

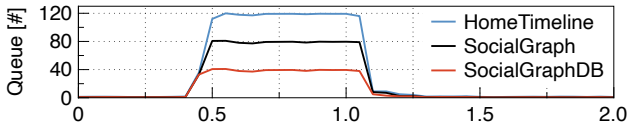
To integrate our work into the three benchmark applications evaluated in Section 5, we implement PathFence in two versions: Apache Thrift [5] in C++ and Spring Boot [12] in Java, ensuring compatibility with a wide range of microservices applications. In the Apache Thrift (used by SocialNetwork and MediaService benchmarks) implementation, we integrate PathFence semaphore-based resource



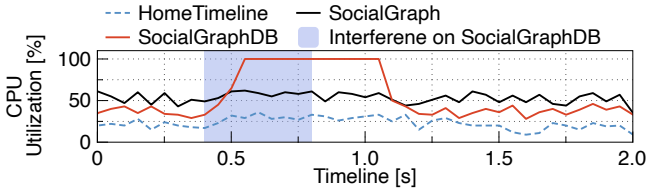
(a) Observed response time spike over the 2-second period



(b) Active worker threads at the shared UM (`ComposePost`)



(c) Queue overflow from the bottleneck DM to UMs on Path 1



(d) CPU utilization of the microservices along the Path 1. A millibottleneck was observed in the DM `SocialGraphDB`.

Figure 10: A fine-grained analysis over a 2-second time span of the repeat experiment from Section3.1 with PathFence, showing the reducing of cross-path dependencies on the same long call chains (see Fig. 3b). (d) A millibottleneck caused by interference from a collocated container at the downstream microservice `SocialGraphDB`. (c) Queue overflow propagation from Path 1 to upstream microservices. (b) Unlike the previous experiments, PathFence limits the number of worker threads for Path 1, preventing Path 2 from competition for available worker threads. (a) As a result, the response time spike is limited to the execution path experiencing the millibottleneck.

management within Thrift's server models. By extending Thrift's thread pool and connection management, we enable dynamic concurrency control to mitigate cross-path dependencies. In the Spring Boot (used by SockShop benchmark) implementation, we integrate PathFence with Spring's ThreadPoolTaskExecutor, allowing execution path-aware resource allocation.

Both implementations allow PathFence to dynamically adjust concurrency levels and optimize response time stability. Any microservices application built upon these two frameworks can be easily integrated with PathFence with minimal additional programming effort.

5 Evaluation

In this section, we first validate the effectiveness of PathFence in reducing the impact of cross-path dependencies on response time using SocialNetwork benchmark application in Section 5.2. We

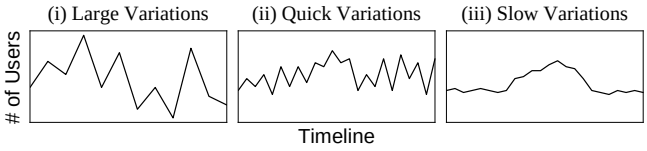


Figure 11: Workload traces.

then compare the performance of PathFence with the state-of-the-art solution (i.e., Asynchronous invocations) using two additional applications (i.e., SockShop and MediaService) in Section 5.3.

5.1 Experimental Setup

Benchmarks. We use three open-source microservices benchmark applications in total: SockShop [4], SocialNetwork, and MediaService from DeathStarBench [16]. SockShop is an online shopping website where users can browse socks, add them to their shopping cart, and place orders. SocialNetwork implements a broadcast-style social network website with uni-directional follow relationships, where users can create, view, and comment on posts. MediaService is a video streaming service that allows users to upload, view, and comment on videos. The microservices architecture graphs of the three microservices applications are shown in Fig. 3 and Fig. 9.

Testbeds. We conduct the experiments on a private cloud with 20 server nodes, each has 8 CPU cores and 32GB of memory running on Ubuntu 22.04. We deploy the microservices applications on the Kubernetes cluster [10].

Workloads To generate background workload, we adopt the RUBBoS workload generator [3] to simulate a configurable number of concurrency users navigating the website. Each user follows a Markov chain model to navigate among web pages, with an average 7-second Poisson distributed thinking time between every two consecutive requests. We control the background workload intensity by varying the number of concurrent users.

5.2 Reducing of Cross-Path Dependencies

We repeat the experiment from Section 3.1 with PathFence enabled to demonstrate that it significantly reduces the impact of cross-path dependencies on response time. Specifically, we use the same topology as Fig. 3b shows, where the upstream microservice `ComposePost` is shared by two execution paths (namely Path 1 and Path 2). To induce a performance anomaly on Path 1 and analyze its impact, we create a millibottleneck of the same length on the downstream microservice `SocialGraphDB`. Meanwhile, we

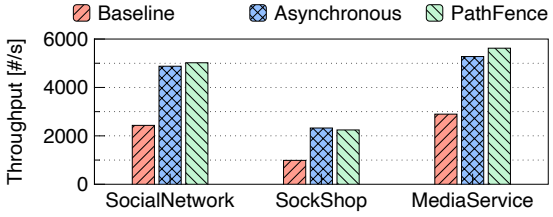


Figure 12: System maximum throughput comparison between PathFence and Asynchronous microservices.

Table 1: Response time and dropped requests comparison between baselines and PathFence under three bursty work traces. The applications are short for SockShop, SocialNetwork, and MediaService, respectively.

App	Workload Variations	Average Response Time [ms]			99th Percentile Response Time [ms]			Dropped Requests [#]		
		Base	Async.	Our	Base	Async.	Our	Base	Async.	Our
SS	Large	249	171	153	3013	378	394	1833	0	0
	Quick	217	165	158	2374	319	327	797	0	0
	Slow	208	161	143	1178	293	299	273	0	0
SN	Large	314	255	223	7045	1094	1037	5981	173	189
	Quick	289	253	218	3097	1075	1021	2973	134	181
	Slow	273	244	252	1329	598	633	533	0	0
MS	Large	445	397	283	3470	1233	1092	3987	239	204
	Quick	416	311	332	3511	1204	997	2429	131	87
	Slow	391	297	286	1573	871	453	1266	0	0

fix the concurrency level limit of Path 1 on the shared microservice `ComposePost` to 120.

Fig. 10 shows the fine-grained analysis of the experiments over a 2-second time span. At the beginning of the experiment, interference from a collocated container creates a millibottleneck on the downstream microservice `SocialGraphDB` of Path 1, as shown in Fig. 10d. The millibottleneck triggers queue overflow to upstream microservices along Path 1, as shown in Fig. 10c. Once the overflow reaches the shared upstream microservice `ComposePost`, the queued requests quickly consume the available worker threads. Unlike the previous experiments, PathFence limits the total number of threads for each execution path, preventing the competition for available worker threads for different paths, as shown in Fig. 10b. This results in a response time spike that is limited to the execution path with the millibottleneck (see Fig. 10a).

The results show that PathFence effectively reduces the impact of cross-path dependencies on response time. The response time spike is limited only to the execution path that experiences performance anomaly and does not propagate to other execution paths. Therefore, the overall response time of the system is significantly reduced, and the spike is limited to the execution path with the millibottleneck. In the following, we will evaluate the performance of PathFence with auto-tuning the concurrency level for each execution path under three realistic workload distributions.

5.3 Comparative Performance

Workloads. We conduct a series of experiments using three bursty workload traces, as shown in Fig. 11. These workloads are collected from real-world production traces and further characterized in [17].

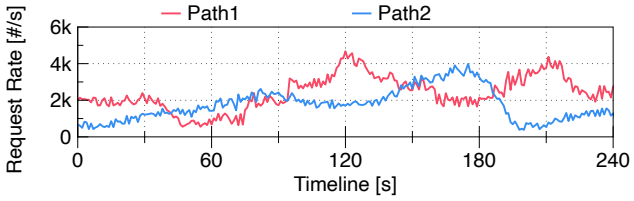
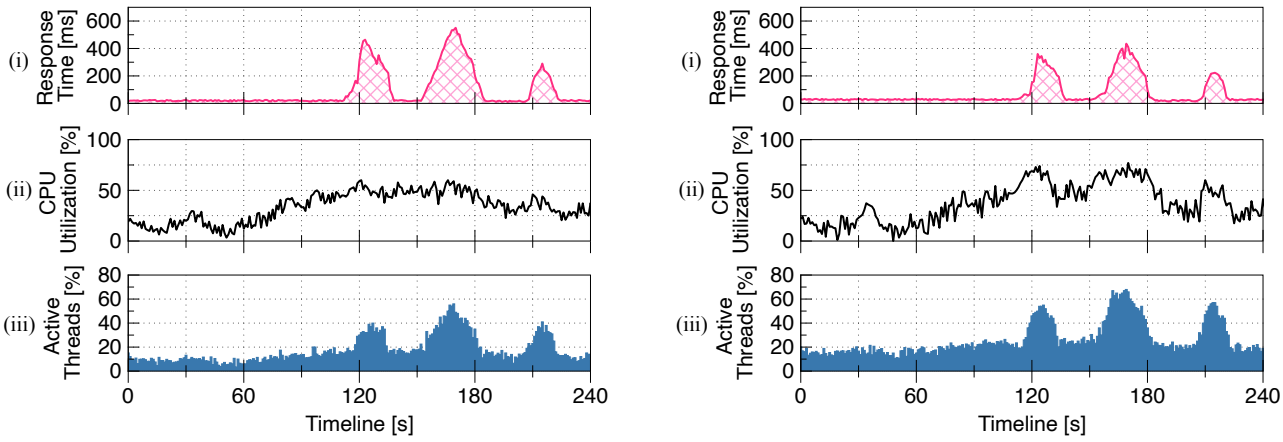


Figure 13: Workload Variation Trace.



(a) Asynchronous microservices

(b) PathFence

Figure 14: Performance comparison between Asynchronous microservices and PathFence under the workload shown in Fig. 13. (i) Average response time. (ii) CPU utilization and (iii) active worker threads of the shared upstream microservice `ComposePost`.

To evaluate system behavior under bursty conditions, we modify the workload by adjusting the number of concurrent users accessing a representative execution path, while keeping the concurrent users accessing other execution paths constant. As a result, hardware resource saturation occurs only in the targeted execution path when the workload reaches its saturation point, whereas resource utilization in other execution paths remains well below saturation (e.g., CPU utilization stays below 40%).

Baseline and asynchronous counterpart. We consider the original versions of the applications, which employ synchronous servers, as the baseline. In addition, asynchronous servers are considered as a common practice to reduce the inter-service dependencies and improve the performance of microservices applications [45] with the drawback of implementation difficulty. Asynchronous servers accept new requests and process them in an event-driven mechanism using only one or a few worker threads. The worker threads return to the event loop once they send a request to a downstream microservice without waiting for the response. Therefore, the worker threads are not associated with specific requests, reducing the potential impact of queued requests overflow [47]. We adopt the asynchronous counterparts to compare the performance by replacing each microservice of the three applications with an asynchronous server and also apply the SCT model to automatically tune its concurrency level limitation of each microservice..

Results. Fig. 12 shows the system maximum throughput comparison of 3 benchmarks between PathFence and Asynchronous microservices. In general, PathFence achieves similar throughput with the asynchronous counterpart, and both of them significantly outperform the baselines. Table 1 shows the response time and dropped requests comparison between baselines and PathFence under three bursty work traces. Compared with the other two applications, SockShop has a simpler microservices topology and fewer execution paths. Both asynchronous microservices and PathFence can eliminate the dropped requests and achieve the lowest response time. On the other hand, SocialNetwork has the most complex microservices topology and the most execution paths, and PathFence and Asynchronous microservices achieve similar performance. Additionally, MediaService has the heaviest requests (a single request takes the longest time to process). PathFence outperforms the asynchronous counterpart in terms of tail latency and dropped requests. This is because Asynchronous microservices use a global worker thread pool and do not isolate the workload for different execution paths. As a result, the limited number of worker threads are quickly occupied by those requests with long processing time, leading to long tail latency. Thus, PathFence achieves the lowest response time with the fewest dropped requests.

To further evaluate the performance of PathFence and asynchronous microservices under more complex conditions, we vary the workload distribution of two representative execution paths in the SocialNetwork benchmark, as illustrated in Fig. 13. Fig. 14 presents a timeline analysis comparing the performance of PathFence and asynchronous microservices under this dynamic workload. During periods of low workload, when no performance anomalies are present, the asynchronous architecture achieves a slightly better average response time (approximately 5ms lower than PathFence). This is because the Asynchronous microservices use only a few worker threads to handle requests when the system is not saturated, leading to less overhead for thread management. However, when the system is under high workload (at least one execution path reaches the saturation point), PathFence outperforms the asynchronous microservices in terms of both average and tail latency. This performance gain is attributed to PathFence 's ability to allocate concurrency levels on a per-path basis, enabling each execution path to reach its peak throughput and maximizing hardware resource utilization. On the other hand, even though asynchronous microservices allocate more worker threads to handle requests during the high workload (due to the implemented tuning of the STC model), they still suffer from performance degradation due to the cross-path dependencies. This is because asynchronous microservices do not isolate the workload for different execution paths, leading to underutilization of hardware resources.

In conclusion, while asynchronous microservices offer benefits under light workloads by reducing thread management overhead,

they fall short in scenarios with high concurrency and complex execution path interactions. PathFence addresses these challenges by isolating execution paths and dynamically adjusting concurrency levels, resulting in stable and efficient performance across a range of workload intensities. The results highlight the importance of path-aware resource management in mitigating cross-path interference and maximizing system throughput in microservices architectures.

6 Related Work

Queue management in microservices. Efficient queue management is crucial for reducing queue overflows and enhancing the performance of microservices [11, 38]. Several studies have explored various queue management strategies to reduce queuing delays and improve throughput, including intelligent scheduling mechanisms [32] and priority-based queues [21]. However, existing approaches often overlook the shared nature of microservices across multiple execution paths, a scenario that can lead to uncontrolled queue propagation and, consequently, system-wide performance degradation. In addition, optimizing queue size has proven to reduce queue overflows and improve the performance of internet servers, as demonstrated in studies [34, 48]. One straightforward approach, such as increasing the number of threads in each component microservice, might help delay or prevent queue overflows and request drops. However, setting a queue size too large can introduce significant performance issues, a problem that has been analyzed in previous research [26, 31]. The challenge lies in determining the optimal queue size for each component under mixed and dynamic workloads, especially for microservices receiving multiple ingress traffic streams from various upstream services (e.g., `ComposePost` microservice in Fig. 3a).

Asynchronous architectures of microservices. Asynchronous architectures have been widely studied for building high performance applications [30, 39, 45]. Asynchronous servers typically employ an event-driven model, where an event loop manages multiple tasks concurrently. Rather than dedicating a separate thread to each task, the server listens for events, such as completed I/O operations or incoming network requests, and processes them as they occur. This model allows a single thread or a small thread pool to efficiently handle many requests [7, 8]. However, while the concept is appealing, the practical implementation of asynchronous architectures in microservices presents significant challenges. Asynchronous servers are often difficult to program and debug due to the complex and obscured control flow, making them more challenging to manage compared to traditional thread-based models [47, 49].

Dynamic resource allocation and scaling. Dynamic scaling strategies (e.g., Amazon Auto Scaling [42]), which aim to balance load and improve system resilience under dynamic conditions, are a commonly used mechanism for managing large-scale web applications, allowing them to adapt to changing workloads and ensuring performance stability [22, 25, 41, 44]. However, transient millibottlenecks and cross-service queue propagation often evade detection by traditional dynamic scaling mechanisms. This limitation arises because the control window for these mechanisms generally operates on a minute-level granularity (e.g., Amazon CloudWatch [1], which monitors at one-minute intervals to prevent over-sensitivity to fluctuating workloads by default). Millibottlenecks, which last for sub-second durations, are too short for these systems to detect and respond to in real-time [20, 43]. While dynamic scaling is effective for handling more gradual workload shifts, it may fail to address response time instability caused by short-lived bottlenecks or cross-service queue propagation in microservice architectures. This highlights the need for more fine-grained scaling solutions that can quickly react to transient performance issues in microservices.

Cross-service dependencies in microservices. A significant challenge in microservice systems is the management of cross-service dependencies, especially in environments with multiple execution paths. These dependencies can lead to contention for shared resources, resulting in blocking and performance degradation across the entire system [16, 24, 50]. To understand cross-service dependencies, Luo et al. [36] conducted an in-depth analysis of microservice call graphs using large-scale tracing data, comparing their structural properties with those of traditional directed acyclic graphs (DAGs). CRISP [50] presents a tool for critical path analysis in microservices, leveraging extensive RPC tracing data collected from Uber's production systems. Gu el al. [18] exploit runtime dependencies among microservices from the perspective of external users, and present Grunt Attack [19], a coordinated attack strategy that degrades the performance of targeted microservices by leveraging these dependencies. Our work builds on these concepts by proposing a novel approach to dynamically allocate resources, ensuring that execution paths are isolated to mitigate the impact of cross-path dependencies.

7 Conclusion

In this paper, we investigated the impact of cross-path dependencies on response time stability in microservices applications. Through extensive benchmarking and empirical analysis, we demonstrated that even minor bottlenecks in a single microservice can propagate across execution paths, causing significant response time fluctuations and system-wide performance degradation. To address this challenge, we introduced PathFence, a novel approach that isolates execution paths by dynamically allocating software resources to prevent queuing effects from propagating across shared microservices. By integrating execution path awareness into thread scheduling and concurrency control, PathFence effectively minimizes the impact of cross-path dependencies. Our evaluation across multiple microservices benchmarks demonstrated that PathFence reduces the 99th percentile response time by up to 80% and decreases the number of dropped requests by more than 90%. Overall, PathFence provides a practical and efficient solution to improve response time stability in microservices applications, making it a valuable contribution to performance optimization in modern cloud environments.

8 Acknowledgments

We thank the anonymous reviewers for their valuable feedback and suggestions. This research has been partially funded by the National Science Foundation under CISE's OAC-2430345. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] 2017. Amazon CloudWatch introduces High-Resolution Custom Metrics and Alarms. https://aws.amazon.com/about-aws/whats-new/2017/07/amazon-cloudwatch-introduces-high-resolution-custom-metrics-and-alarms/.

[2] 2019. The httperf HTTP load generator. https://github.com/httperf/httperf. Accessed: 2024.

[3] 2019. RUBBoS: Bulletin Board Benchmark. https://projects.ow2.org/view/rubbos/. Accessed: 2023.

[4] 2023. Sock Shop: A Microservices Demo Application. https://github.com/microservices-demo/microservices-demo. Accessed: 2024.

[5] 2024. Apache Thrift. https://thrift.apache.org/. Accessed: 2024.

[6] 2024. libevent: An Event Notification Library. https://libevent.org/. Accessed: 2025.

[7] 2024. NGINX. http://nginx.org/. Accessed: 2024.

[8] 2024. Node.js. https://nodejs.org/en/. Accessed: 2024.

[9] William Abernethy. 2019. *Programmer's Guide to Apache Thrift*. Simon and Schuster.

[10] The Kubernetes Authors. 2025. Kubernetes. https://kubernetes.io/.

[11] Ataollah Fatahi Baarzi and George Kesidis. 2021. Showar: Right-sizing and efficient scheduling of microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 427–441.

[12] Broadcom. 2024. *Spring Boo.* https://spring.io/projects/spring-boot

[13] Jeremy Cloud. 2013. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.slideshare.net/InfoQ/decomposing-twitter-adventures-in-serviceoriented-architecture.

[14] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.

[15] Yu Gan, Mingyu Liang, Sundar Dev, David Lo, and Christina Delimitrou. 2021. Sage: practical and scalable ml-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 135–151.

[16] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, et al. 2019. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *ASPLOS*. 3–18.

[17] Anshul Gandhi, Mor Harchol-Balter, Ram Raghunathan, and Michael A Kozuch. 2012. Autoscale: Dynamic, robust capacity management for multi-tier data centers. *TOCS* 30, 4 (2012), 1–26.

[18] Xuhang Gu, Jianshu Liu, and Qingyang Wang. 2023. A BlackBox Approach to Profile Runtime Execution Dependencies in Microservices. In *2023 IEEE 9th International Conference on Collaboration and Internet Computing (CIC)*. IEEE, 116–120.

[19] Xuhang Gu, Qingyang Wang, Jianshu Liu, and Jinpeng Wei. 2024. Grunt Attack: Exploiting Execution Dependencies in Microservices. In *2024 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 115–128.

[20] Xuhang Gu, Qingyang Wang, Qiben Yan, Jianshu Liu, and Calton Pu. 2024. Syncmillibottleneck attack on microservices cloud architecture. In *Proceedings of the 19th ACM ASIA Conference on Computer and Communications Security*. 799–813.

[21] Mor Harchol-Balter, Takayuki Osogami, Alan Scheller-Wolf, and Adam Wierman. 2021. Correction to: Multi-server queueing systems with multiple priority classes. *Queueing Systems* 99 (2021), 397–398.

[22] Rubaba Hasan, Timothy Zhu, and Bhuvan Urgaonkar. 2024. AutoBurst: Autoscaling Burstable Instances for Cost-effective Latency SLOs. In *Proceedings of the 2024 ACM Symposium on Cloud Computing*. 243–258.

[23] Md Rajib Hossen, Mohammad A Islam, and Kishwar Ahmed. 2022. Practical efficient microservice autoscaling with QoS assurance. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. 240–252.

[24] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable failures in the wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 73–90.

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

[26] Daeho Jeong, Youngjae Lee, and Jin-Soo Kim. 2015. Boosting {Quasi-Asynchronous}{I/O} for better responsiveness in mobile devices. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. 191–202.

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

[28] Ron Kohavi, Randal M Henne, and Dan Sommerfield. 2007. Practical guide to controlled experiments on the web: listen to your customers not to the hippo. In *13th ACM SIGKDD*. 959–967.

[29] Ron Kohavi and Roger Longbotham. 2007. Online experiments: Lessons learned. *Computer* 40, 9 (2007), 103–105.

[30] Maxwell N Krohn, Eddie Kohler, and M Frans Kaashoek. 2007. Events Can Make Sense.. In *USENIX Annual Technical Conference*. 87–100.

[31] Zijun Li, Quan Chen, Shuai Xue, Tao Ma, Yong Yang, Zhuo Song, and Minyi Guo. 2020. Amoeba: Qos-awareness and reduced resource usage of microservices with serverless computing. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 399–408.

[32] Jianshu Liu, Shungeng Zhang, and Qingyang Wang. 2023. μConAdapter: Reinforcement Learning-based Fast Concurrency Adaptation for Microservices in Cloud. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*. 427–442.

[33] Jianshu Liu, Shungeng Zhang, Qingyang Wang, and Jinpeng Wei. 2020. Mitigating large response time fluctuations through fast concurrency adapting in clouds. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 368–377.

[34] Guangqiang Luan, Pu Pang, Quan Chen, Shuai Xue, Zhuo Song, and Minyi Guo. 2022. Online thread auto-tuning for performance improvement and resource saving. *IEEE Transactions on Parallel and Distributed Systems* 33, 12 (2022), 3746–3759.

[35] Shutian Luo, Chenyu Lin, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, Huanle Xu, and Chengzhong Xu. 2024. Optimizing resource management for shared microservices: a scalable system design. *ACM Transactions on Computer Systems* 42, 1-2 (2024), 1–28.

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

[37] Tony Mauro. 2015. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.

[38] Amirhossein Mirhosseini, Sameh Elnikety, and Thomas F Wenisch. 2021. Parslo: A gradient descent-based approach for near-optimal partial slo allotment in microservices. In *Proceedings of the ACM Symposium on Cloud Computing*. 442–457.

[39] David Pariag, Tim Brecht, Ashif Harji, Peter Buhr, Amol Shukla, and David R Cheriton. 2007. Comparing the performance of web server architectures. *ACM SIGOPS Operating Systems Review* 41, 3 (2007), 231–243.

[40] Vern Paxson, Mark Allman, J Chu, and Matt Sargent. 2011. RFC 6298: Computing TCP's retransmission timer.

[41] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. 2020. {FIRM}: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th ({OSDI} 20)*. 805–825.

[42] Amazon Web Service. 2024. Amazon Auto Scaling Documentation. https://aws.amazon.com/documentation/autoscaling. Accessed: 2024.

[43] Huasong Shan, Qingyang Wang, and Qiben Yan. 2017. Very short intermittent ddos attacks in an unsaturated system. In *International Conference on Security and Privacy in Communication Systems*. Springer, 45–66.

[44] Jiuchen Shi, Hang Zhang, Zhixin Tong, Quan Chen, Kaihua Fu, and Minyi Guo. 2023. Nodens: Enabling Resource Efficient and Fast {QoS} Recovery of Dynamic Microservice Applications in Datacenters. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 403–417.

[45] Akshitha Sriraman and Thomas F Wenisch. 2018. {μTune}:{Auto-Tuned} Threading for {OLDI} Microservices. In *OSDI 18*. 177–194.

[46] Qingyang Wang, Xuhang Gu, and Calton Pu. 2024. A Study of Response Time Instability of Microservices at High Resource Utilization in the Cloud. In *2024 IEEE 6th International Conference on Cognitive Machine Intelligence (CogMI)*. IEEE, 111–116.

[47] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. 2017. A study of long-tail latency in n-tier systems: Rpc vs. asynchronous invocations. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 207–217.

[48] Qingyang Wang, Shungeng Zhang, Yasuhiko Kanemasa, Calton Pu, Balaji Palanisamy, Lilian Harada, and Motoyuki Kawaba. 2019. Optimizing n-tier application scalability in the cloud: A study of soft resource allocation. *ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOMPECS)* 4, 2 (2019), 1–27.

[49] Shungeng Zhang, Qingyang Wang, Yasuhiko Kanemasa, Jianshu Liu, and Calton Pu. 2020. DoublefaceAd: A new datastore driver architecture to optimize fanout query performance. In *Proceedings of the 21st International Middleware Conference*. 430–444.

[50] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. {CRISP}: Critical Path Analysis of {Large-Scale} Microservice Architectures. In *USENIX ATC 22*. 655–672.

[51] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 2019. Latent error prediction and fault localization for microservice applications by learning from system trace logs. In *Proceedings of the 2019 27th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*. 683–694.