PES Innovation Lab

Project Report

Summer Internship 2025

MiCozServices

Interns

Name	SRN
Lekhyasree M	PES1UG23CS328
Nathan P	PES2UG23CS368
Navneet N	PES2UG23CS371
Suchitra S	PES2UG23CS608

Mentors

Achyuth YS
Animesh ND
Smaran J

PES Innovation Lab
PES University
100 Feet Ring Road,
BSK III Stage,
Bangalore - 560085

Abstract

Program performance, such as throughput and latency, are central concerns for developers in time sensitive applications, such as large scale distributed services like Machine Learning (ML) pipelines, user facing web services, and so on.

For these large, distributed applications, we define latency as the time taken to respond to a request sent to the application and throughput as the number of requests an application can serve given a unit of time (For example the number of requests an application can serve in a second).

These applications are designed as small, (usually) independent chunks of code, called microservices. This lets applications distribute easier, scale faster, and allow manipulations/updates with finer granularity than monolithic (single binary) applications. These microservices are connected to one another with network calls, forming a directed graph, showing patterns of data movement and processing.

Latency and throughput now take into account the runtime of each service along with the network transmission time for communication between services. As these applications grow, it becomes harder to determine where code optimizations will be most effective at reducing overall latency and increasing throughput. Improving just one important service can sometimes lead to a significant, exponential improvement across the entire system, while optimizing a service which is not as important will lead to almost no improvement in overall performance.

Causal Profiling is a method of profiling being used today on monolithic which quantifies the performance gain in the entire system when a certain portion of code is sped up. It does this by running experiments where it speeds up certain portions of code 'virtually' by stalling all other threads of execution running in parallel with the portion of code to be sped up. This answers the question: *"If this part of the code were faster then would the system really perform better"*

Our aim is to test the possibility of applying causal profiling to microservices. Can the principles of causal profiling and virtual speedup be applied to microservice architectures to answer the question ***"If this service in my architecture were faster then would the system really perform better?"***

To this end we come up with theoretical models to apply virtual speedup to services and perform exhaustive empirical testing on many simulated microservice architectures as well as some real microservices. We prove that it is possible to apply causal profiling to certain microservice graphs and can be applied on real microservices to find which service is most worth optimizing.

Contents

Chapter 1

Introduction

Developers have been concerned with the performance metrics of latency and throughput of applications for decades.

A 2017 report by Akamai [1] shows that a 100ms increase in latency in website load time can lead to a 7% decrease in customer conversion rate. Performance is not just a luxury, it's a requirement which, if a system fails to meet, can cause massive losses.

Before the adoption of distributed systems, most applications were developed as monoliths, running on a single server or mainframe.

Developers looked to profiling tools like perf or gprof, profilers which tell the developer which functions / portions of code run for the longest amount of time. This information can mislead developers, just because a portion of code runs for the longest amount of time does not necessarily mean optimizing it will lead to overall performance gain.

This is due to code today running in parallel / concurrently with other code. Code depends on other code for outputs, they contend for shared resources. Code can cause slowdowns elsewhere, speeding up a function or line of code can cause unforeseen improvements or deterioration in performance.

Traditional profilers fail to account for these causal relationships.

Causal profiling tools like **COZ** [2] solve this issue by explicitly quantifying the impact of speeding up a line of code by various amounts. See: Section 2.1 for more details.

As organizations shifted towards distributed systems, to allow for more scalability and avoid a single point of failure. This evolution introduced the concept of microservices, applications that can be decomposed into small independent services distributed across machines that communicate over networks to execute one overall assignment. Microservices gained popularity due to their advantages in resilience, scalability and versatile deployment.

Existing microservices are usually profiled by relying on extensive observability on each service and need manual work to be done by developers, who read logs and traces to pinpoint issues and bottlenecks in the system, a very unintuitive and tedious process. Any method of profiling which can provide intuitive results and reduction in manual work in order to find system bottlenecks is valuable.

1.1 Problem Statement

This project aims to, through systematic case studies and exhaustive empirical testing determine whether implementing causal profiling's virtual speedup principles at the service level within a microservice architecture produce significant results, in the sense that the profile correctly predicts which service is worth optimizing in a microservice architecture for best reduction in latency.

We specifically constrained the scope of this project to explore optimizing latency and not throughput, but future work can include predicting the effects of optimization on the throughput of the system.

Chapter 2

Literature Survey/Related Work

2.1 Causal Profiling & COZ

Causal profiling is a performance analysis methodology that pinpoints meaningful optimizations that will have system-wide performance improvement, rather than merely highlighting where maximum system time is spent. Unlike traditional profilers which only report the code segments with highest execution times, causal profiling addresses the concern: Would making this code faster actually improve end-to-end performance?

Causal profiling was first described in the paper *"COZ: Finding Code that Counts with Causal Profiling" Emery D Berger and Charlie Curtsinger* [2]

The COZ profiler implements causal profiling for monolithic C/C++/Rust binaries and introduces the concept of running live performance experiments to determine the impact of optimizations.

Traditional profilers frequently mislead developers into optimizing "hot" code, emphasizing the sections of their code that execute for the longest duration. However, optimizing the code that runs for the longest time will not always lead to a proportional improvement in overall performance.

This is because code is not always on the critical path of execution.

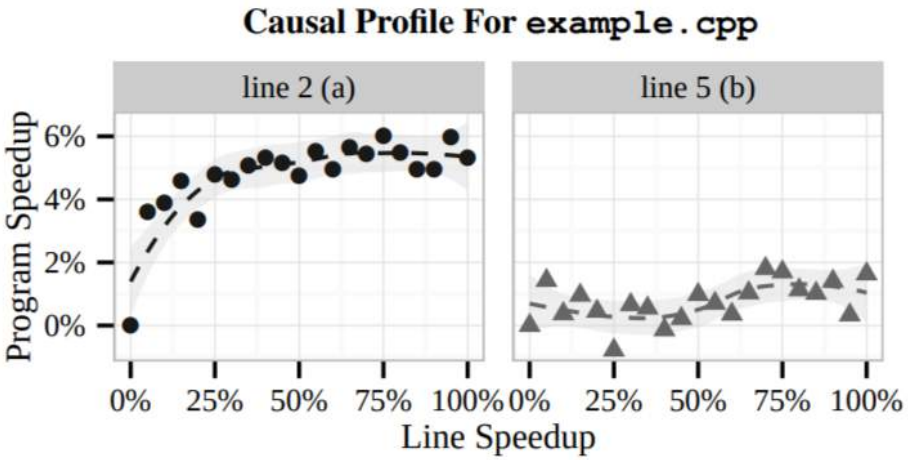

Figure 2.1: The image shows a causal profile of an example program, shown in the COZ paper. The graphs compare: The amount of speedup to a line on the x-axis versus the corresponding total program speedup on the y-axis for two lines of code (a) and (b).

This critical path, defined as the longest sequence of unparallelizable operations, dictates the overall execution time. Only by accelerating code on this critical path can the total execution time be reduced.

Causal profiling addresses this by running performance experiments, during program execution. Each experiment calculates the impact of any potential optimization by virtually speeding up code i.e inserting pauses that slow down all other threads of execution running concurrently. The key insight is that this slowdown has the same relative effect as running that line faster, thus 'virtually' speeding it up.

This makes it clear that speeding up line(a) will give more program speedup compared to speeding up line(b). The developer is clear on where they should be spending their optimization efforts.

Causal profiling works on the basis of the facts that:

1. Modern programs run code in parallel.
2. Code is executed multiple times.The performance improvement gained by speeding up a line of code is not always linear

3. Only optimizations on the critical path improve the total system performance.
4. Critical path of execution of a program may shift when some code is optimized. Thus making it useless to optimize code beyond a certain point.

2.2 Microservices

Microservice architectures have become a standard for building scalable, maintainable and robust systems. In a microservice setup, an application is implemented by splitting it into isolated and loosely independently deployable services, each running as a separate process, which communicate over network protocols.

The execution path that processes each request sent to a microservice can be modeled as a DAG: directed acyclic graph with nodes representing instances of services and edges representing calls to other services, creating complex dependencies.

Each microservice application can have multiple execution paths depending on the type of request. Determining the bottleneck service and identifying how local improvements affect overall performance is a particularly difficult task.

Services in a microservice architecture are usually deployed as binaries on Virtual Machines: A computer system that is implemented in software rather than hardware and/or Containers: a standard unit which isolates and encapsulates software for deployment.

Microservices are popular for building scalable and modular systems, but their performance is heavily influenced by complex inter-service dependencies. Common patterns include:

1. Services frequently rely on others to complete tasks outside their scope waiting for responses
2. Calling multiple services in parallel Interacting with shared resources like databases.

Microservices inherently involve intricate causality between components. The

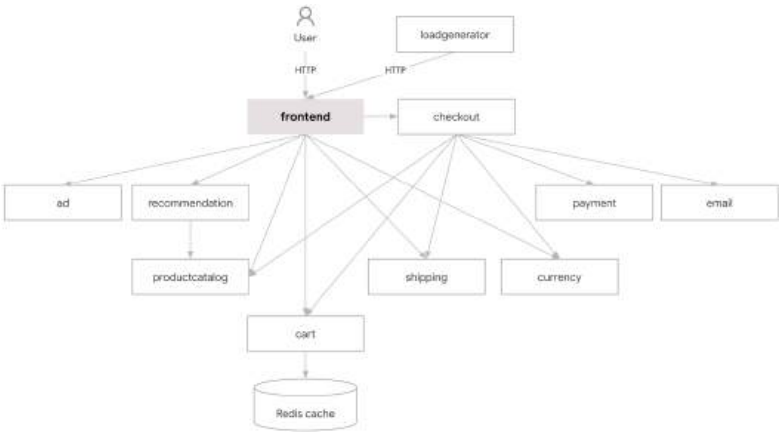

Figure 2.2: The image shows the architecture of Google's online boutique microservice demo, with over 10 distinct services.

basis on which causal profiling works (on monolithic applications) also applies to microservice architectures.

In any microservice architecture, the call graph for a client request represents which services call which, contains a critical path, the longest chain of sequential activities. This path forms the bottleneck of the system. Optimizing a service on the critical path improves performance, at least until the critical path shifts. Optimizing any services outside the critical path yields no gain.

Standard profiling as outlined before relies on extensive observability and a lot of manual work. There exist some methods of profiling which account for causality, techniques like critical path (link) tracing or analysis on time series data using statistical techniques like Granger Causality [3]. But the novelty of causal profiling is in explicitly being able to quantify the amount of performance gain to be expected when a service is sped up.

It's important to note that despite the above definition of a service in a microservice architecture, going forward we consider request handlers running in each binary to be services, and not the binary itself. So one binary can consist of multiple services, if it has code to service multiple different types of requests. This is because it doesn't make much sense to speed up an entire binary and nor is all the code in a binary executed when a certain type of request is made.

Chapter 3

Assumptions & Scope

Causal profiling for microservices was tested and researched on, assuming the conditions of a real world data center:

1. Where actual network latencies between services deployed on different machines are very less to almost negligible.
2. Services themselves are deployed on virtual machines and/or inside containers, on one or more machines.

The following cases are not considered in scope of this project:

1. Shared Resources, resources that can only be held by a certain amount of services at a time, were not considered.
2. Microservice architectures in which services call others using non-blocking network calls (services don't block waiting for a response and continue computation, blocking at a later time)
3. Architectures with inherently asynchronous components like message queues. We deal only with quantifying performance gain in the request-response cycle of a microservice system.

Chapter 4

Causal Profiling Model & Virtual Speedup Hypothesis

We begin by drawing an analogy between causal profiling in monolithic applications and its potential adaptation to distributed microservice systems.

4.1 Virtual Speedup Model

While COZ profiles lines of code, finding which line of code is worth optimizing, we move our focus to which service is worth optimizing in a microservice architecture.

Just as COZ stalls all threads of execution except the one being tested for speedup, our model stalls all other branches of execution running concurrently with the target service to be virtually sped up.

Our Hypothesis is: Speeding up a service is equivalent to Stalling all other parallel branches of execution.

Consider this microservice DAG path for example:

The timing diagram of the given architecture is as follows, showing the time at which instances of each service are called and how long they run for. Here, there are three concurrently running branches of execution. It's clear to see that if we send a request to service A, we can expect a response back in 10 units of time.

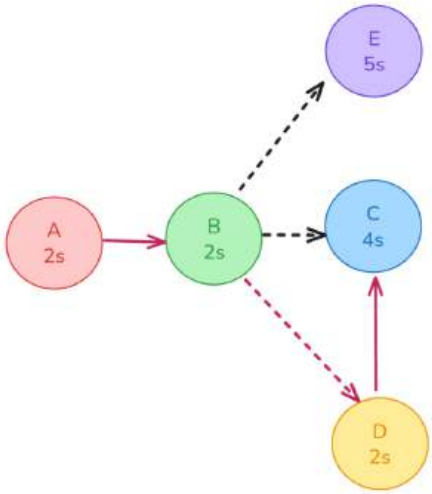

Figure 4.1: Consider this microservice DAG path for example:

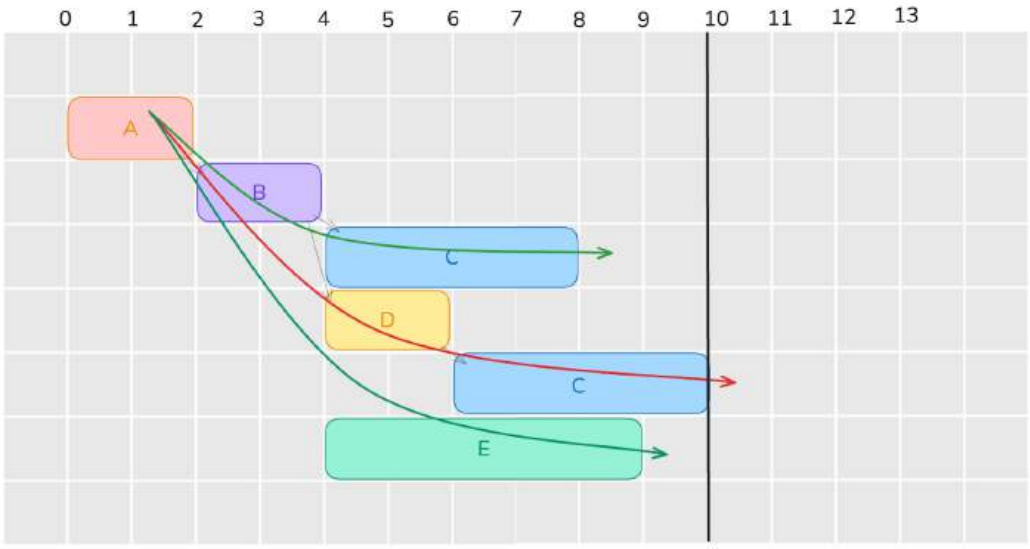

Figure 4.2: First, actually speeding up service C by 1 time unit reduces total execution time by 1 unit.

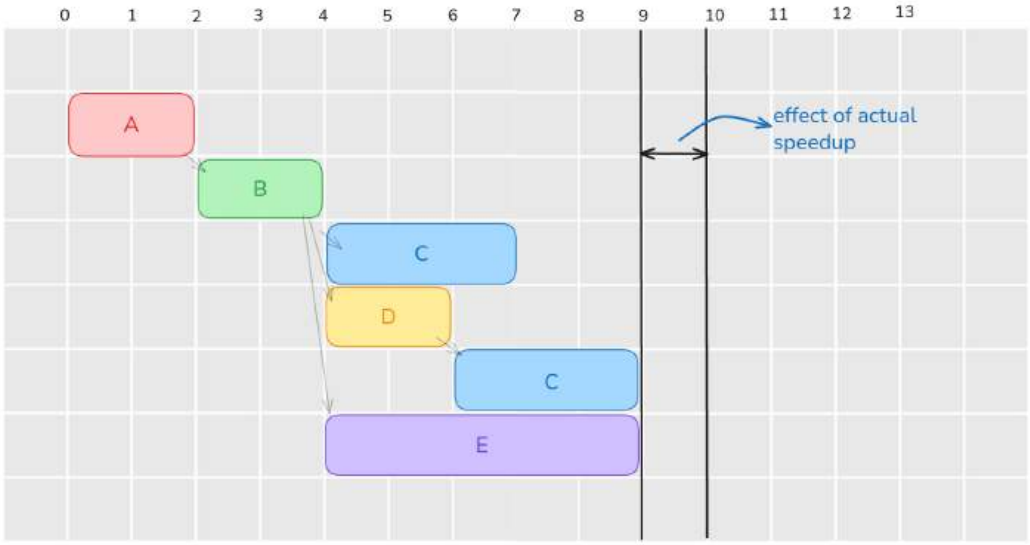

Figure 4.3: To virtually speed up a service running on one branch of execution, we need to stall all other branches of execution that run concurrently with the service. Here we virtually speed up service C by applying stalls of 1 unit time.

To virtually speed up a service running on one branch of execution, we need to stall all other branches of execution that run concurrently with the service. Here we virtually speed up service C by applying stalls of 1 unit time.

The insight here is that stalling a branch of execution entails stalling any one of the services running on that branch, as by definition, all services on a branch are directly or indirectly dependent on all services on the same branch.

To find the effective runtime, after applying virtual speedup we simply subtract the additional added delays.

$$\text{Effective Runtime} = \text{Total Runtime} - \text{Stalls Added}$$

$$\text{Effective Runtime} = \text{Total Runtime} - n \cdot d$$

where
n: number of times the service to be sped up ran (number of instances of the service)
d: delay injected each time the service ran

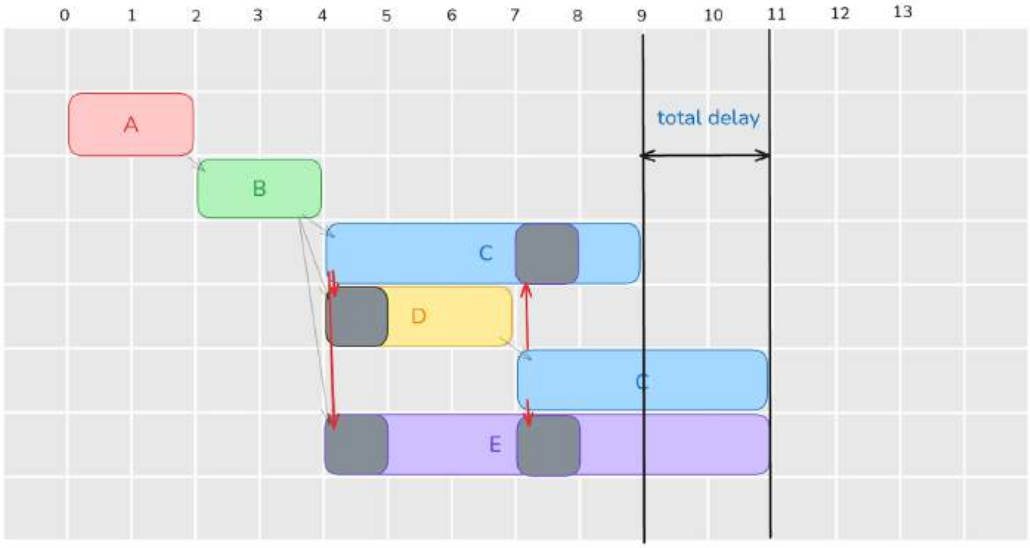

Figure 4.4: The grey blocks show the inserted delays(inserted when C runs, shown by the red arrows)

Causal profiling has the distinct advantages over traditional critical path tracers, which only tell the developer what the critical path of execution of their system is.

Since stalling other branches of execution is logically equivalent to actually speeding up a service, causal profiling is able to show when optimizing a service does not lead to any more improvement in performance due to a shift in critical path.

For example, speeding up C by more than 1 unit does not lead to more than 1 unit of total speedup

These DAGs are useful for modeling the execution tree of a particular end-point of a microservice architecture.

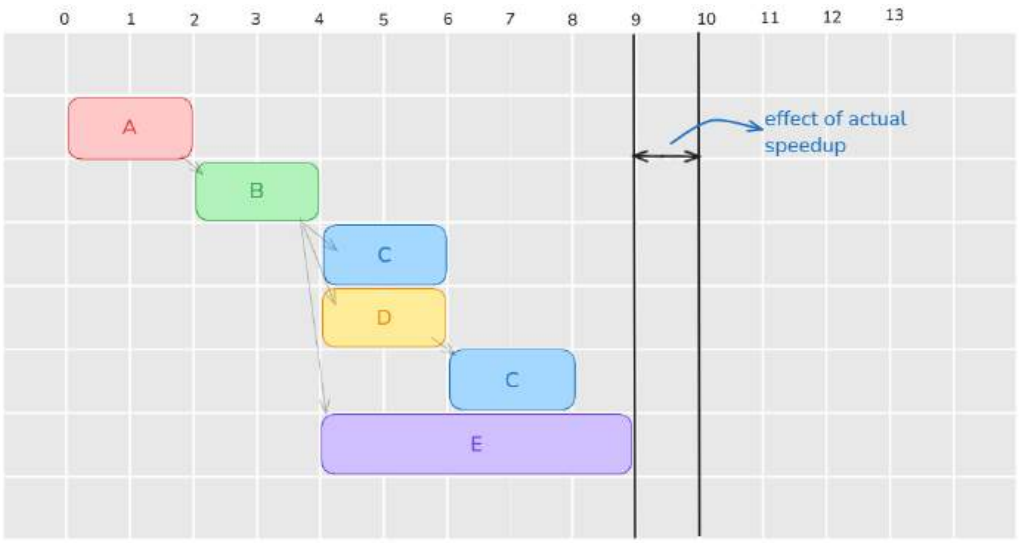

Figure 4.5: For example, speeding up C by more than 1 unit does not lead to more than 1 unit of total speedup

4.2 Technical Methodology

In order to implement this model of virtual speedup, we need to define some requirements, When a service is to be virtually sped up: At least one service on all other branches of execution must be stalled and this must be the only service that is stalled on that branch.

Each service instance maintains a local delay counter, while the overall architecture uses a shared global delay counter.

The global counter, managed by a separate node called the global counter store or the delay store, represents the total intended delay across the system and can be accessed or incremented through a dedicated API.

In contrast, a service's local counter tracks how much delay that specific instance has already served.

To synchronize with the rest of the system, a service compares its local counter with the global counter and sleeps for the difference. This brings it in line with the total delay expected by the architecture.

When a service needs to impose a stall across the entire system (stall other

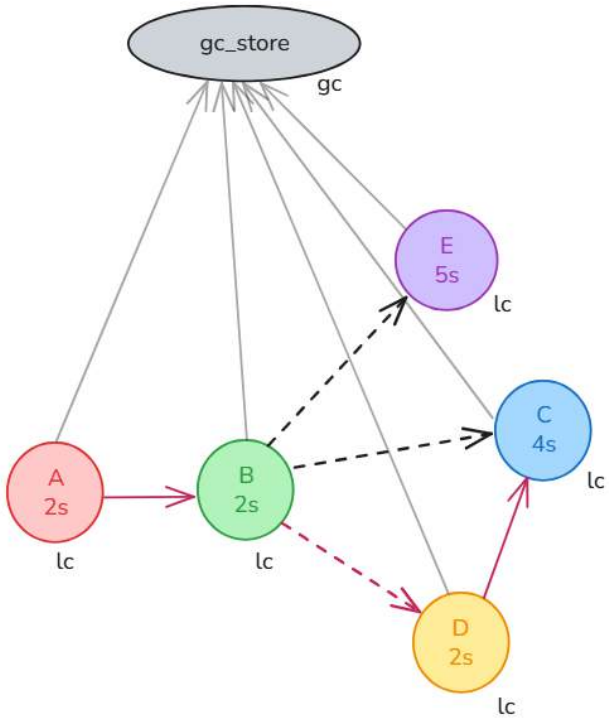

Figure 4.6: To this end, we adopt a global and local delay counter approach similar to COZ

services to virtually speed itself up), it simply increments both the global delay counter and its own local counter.

Before a service instance exits, it is required to sync its local counter with the global counter. This approach means we avoid the need for periodic polling of the global delay counter throughout execution, which could create scalability issues in large microservice deployments where hundreds of instances might frequently query the delay store.

Delaying just before exiting is equivalent to delaying during execution. From the perspective of other services, the response is simply observed to arrive later. To compute the effective runtime of the system when a service has been virtually sped up, we subtract the global delay counter value from the raw end to end latency measured.

All the logic to do with delays will be handled by middleware (interceptors) which act before and after a service's typical execution. This middleware is instrumented into the application code of each service. Along with some wrapper functions around operations like network calls. See Section 4.3

In addition:

1. Each service also exposes an API to set itself as the target of virtual speedup, which takes a parameter of how much it should speed up by.
2. Each request call chain is identified by a Universally Unique Identifier (UUID), which is set at the first service in the call chain, and propagated forward to all other services, this UUID is used to query the delay store for the global delay of the current call chain.

4.3 Propagation of Delays & Network Call Wrappers

In a chained or nested call structure, it is possible for multiple services on the same branch to each attempt to serve the local delay themselves. This would lead to redundant stalling, breaking the constraint of effectively stalling each branch only once.

To resolve this, we implement a delay propagation protocol through middleware, where every service before making a call to another service it depends on, passes along its current local delay counter as part of the request metadata, upon receiving the request, the next service in the chain performs the following sequence of actions:

1. Check if it is the target of virtual speedup, in which case it increments its local counter as well as makes an API call to the delay store to increment the global delay count.
2. Continues its regular execution
3. Just before returning a response back to its caller, it fetches the global delay counter from the delay store, and compares it to its own local delay count.
   (a) If global and local counter values match, then this service has already caught up with global delay and a stall is not needed.
   (b) But if local counter value is lesser than global, then the service sleeps for the difference, and updates its local delay counter accordingly.
4. It sends back its local delay counter as metadata in the response back to its callee. The callee then inherits back this local delay if it is higher than its own local delay.

This propagation of delays ensures that only one service ever gets stalled on a branch of execution. This propagation of metadata such as UUID's and local delay counters is done by wrapping all network calls to other services with custom code which performs the task of sending and receiving metadata.

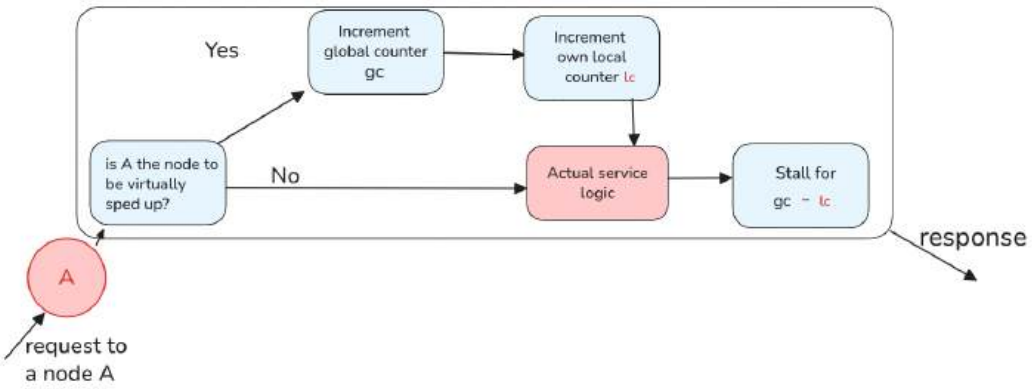

Figure 4.7: Tracing a request to node A

Chapter 5

Randomized DAG generation & Testing

In order to prove our virtual speedup model accurately reflects the effects of actually speeding up a service, we:

1. Model the execution path of user facing endpoints of microservice architectures as DAGs
2. Apply random amounts of virtual speedup and compare the total runtime after applying the same amount of actual speedup.

To this end, we have built an automatic microservice DAG generation pipeline which:

1. Generates DAGs
   (a) With a random number of nodes, where each node does computational work for a randomly generated fixed amount of time, the time is fixed by intentionally busy waiting for a fixed duration.
   (b) Each node is set to randomly and intermittently call other services, in between its computation work, either one other service, or many others in parallel.
2. Deploys each node in the DAG as a Golang web server in Docker containers, with custom code generated for each service.

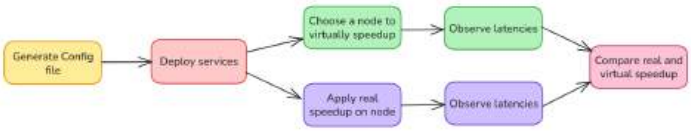

Figure 5.1: Testing workflow

3. Exhaustively tests for all services asserting that the reduction in end to end latency shown by virtually speeding up said service matches with the reduction shown when the service is actually sped up by the same amount

The reduction in latency shown by virtually speeding up a service accurately reflects the reduction in latency when the service is actually sped up by the same amount for all 60+ randomly generated DAGs tester.

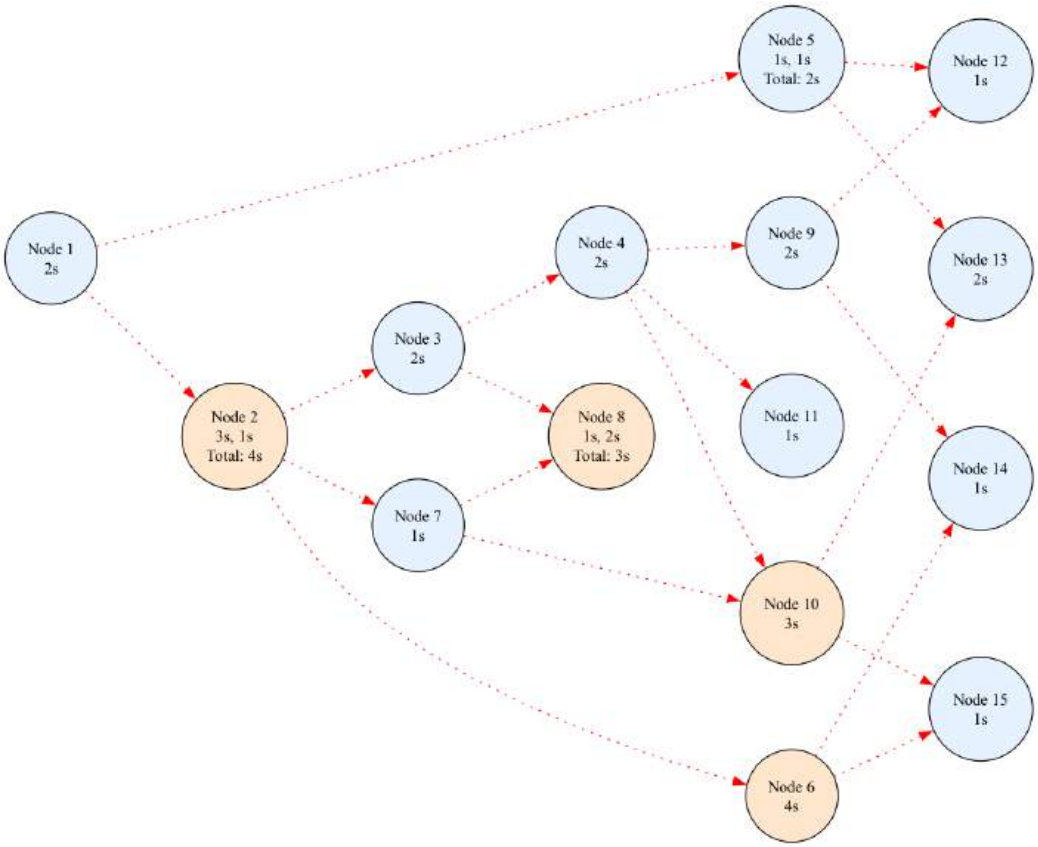

Figure 5.2: Here are some other DAGs where all calls are parallel. (Dotted red lines). Our counter approach works here as well.

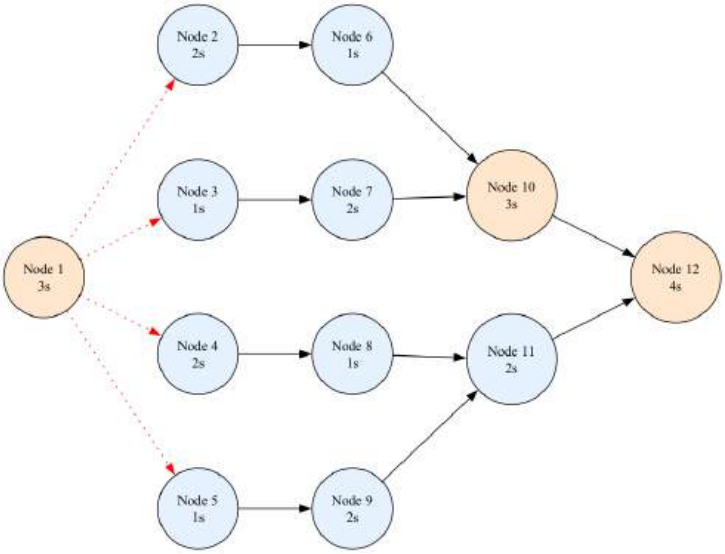

Figure 5.3: Randomly Generated DAG

Field	**Node 10**	**Node 9**
Port	5010	5009
Speedup Amount	1	1
Virtual Speedup Duration	12.0142	13.0131
Real Speedup Duration	12.0046	13.0043

Table 5.1: Speedup Results for Nodes **9** and **10**

We observe that the Base Time of the above microservice DAG is 13s. Performing virtual speedup and real speedup on node 10, we get a runtime of 12s. Similarly, applying speedup on node 9 yields a runtime of 13s, which is the same as the baseline. This shows that speeding up node 9 does not improve the latency, as expected, since it does not lie on the critical path

Here are some other randomly generated DAGs:

Our virtual speedup model not failing in any of these cases, gives credibility to the theory that, Speeding up a service == Stalling all other parallel branches of execution.

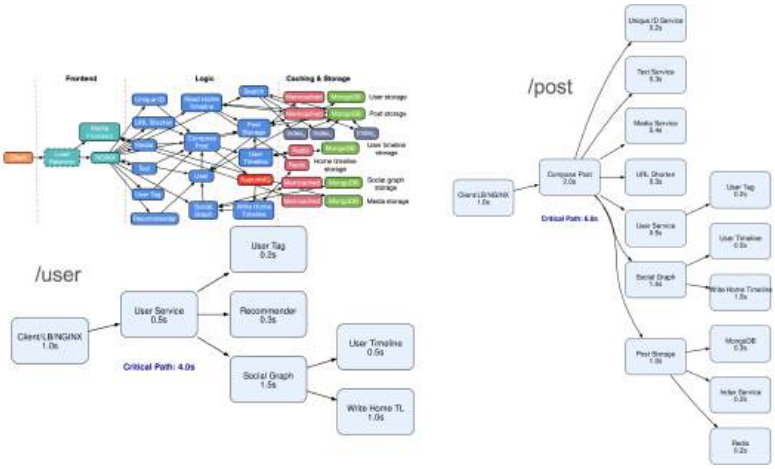

Figure 5.4: Enter Caption

5.1 Modelling real microservices

We also model some real microservice architectures to further validate our virtual speedup model, and show its applicability beyond just random dags.

Chapter 6

Testing on a real microservice architecture

So far we have exhaustively tested the applicability of our causal profiling model on many simulated microservice architectures. Each service was precisely controlled, down to how long each service does computational work for, and when and how it calls other services.

Real microservice architectures have unpredictable latencies, as the runtime of a service in the architecture can be different based on a variety of factors and conditions. Some of the factors that can affect runtime are: How processes were scheduled onto the CPU, cache misses, I/O wait times etc...

We now have to contend with variation in end to end latency values, hence we send multiple requests to an endpoint and analyze distributions of latencies using summary statistics methods. A distribution refers to a function that shows the possible values of latency and how frequently they occur.

In order to tell whether latencies have reduced overall after speeding up a service, we use two methods:

1. Distribution Tests:
    (a) Null hypothesis tests which are used to compare two distributions of data
    (b) We make use of the one tailed Mann Whitney U test, a non-

parametric (not assuming any standard underlying distribution of data) test which compares two distributions stochastically. A distribution of data X is said to be stochastically lesser than another distribution Y if for any given value, a random value from X is more likely to be greater than it than a random value from Y

(c) All tests were performed at the 95% confidence level. Which is the probability that a sample accurately represents the population.

2. Cumulative Distribution Functions:

   (a) CDFs are functions of a distribution of data which describe the probability that a random variable from the distribution takes a value less than a particular value.

   (b) CDF is estimated non-parametrically using np.quantile() which works on the basis of Hyndman-Fan sample quantile estimator.

   (c) CDF plots can be used to more easily infer, graphically, if speedup had any effect on latencies

We also use the inverse of the CDF to estimate the p99 (99th percentile. I.e latency that is worse than 99% of the other latency values in the distribution) latency of the distribution, we compare the p99 latency of two distributions to quantify the amount of latency reduction.

6.1 Load Testing / Latency data collection

In order to collect end to end latency for all endpoints in the architecture, we load test the system, model actual usage of the system by simulating multiple users sending requests to the system concurrently.

This is done using a python load testing tool called Locust, we outline our load testing methodology below:

1. Avoiding Coordinated Omission Coordinated omission is a well-known measurement bias: If a load generator waits to send the next request until the previous one completes (this is called a closed workload), then when the system under test slows down or stalls, the tester sends fewer requests. This masks the true latency experience of users. In essence, latency percentile metrics become overly optimistic because

slow requests suppress new load being sent. To combat this, we adopt an open workload model: requests are sent at a fixed rate regardless of response times. This reflects how real users behave, if the system is slow, user request arrivals don't pause. Each request's latency is measured independently, so we capture the full latency distribution even when the system has a backlog of requests to service.

2. Estimating Throughput capacity & Load testing at a moderate load. We start by performing a closed workload test increasing the number of concurrent users until we hit a maximum RPS ceiling, we then open load test the system using half of this maximum RPS, so as to capture latencies during typical user behaviour without overloading and destabilizing the services.

3. Number of Requests sent. Around 300 requests were always sent to the endpoint being tested, this is so that we have enough latency data points to infer parameters about the underlying distribution with high confidence level.

6.2 Test Environment

1. The microservice architecture was set up on one machine, with each service running in a separate container with CPU and memory resource limits.

2. Load testing software was run on another machine, both of these machines were connected via Ethernet cable.

3. The microservices CPU and memory usage was constantly monitored to ensure that the system was not overloaded, as latency data points from an overloaded system do not reflect the responsiveness of the system under typical conditions. This observability is set up using tools like node-exporter and prometheus along with grafana which is used to visualize the data.

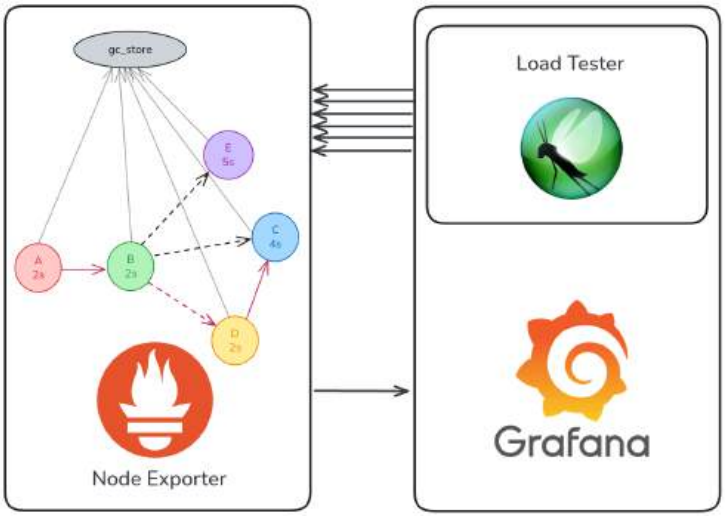

Figure 6.1: Testing Overview

Our testing took place on an ECommerce website microservice, with the following architecture:

Let's specifically take up one of the cases tested empirically, looking at the /cart endpoint which invokes the cart service. Which in turn calls the ad and product services in parallel. Through manual runtime measurements we know that the cart service is bottlenecked on the ad service. Which means that the ad service is on the critical path of execution when a request to /cart is being processed, while the product service is not, speeding up the ad service should result in a reduction in latencies while speeding up the product service should not.

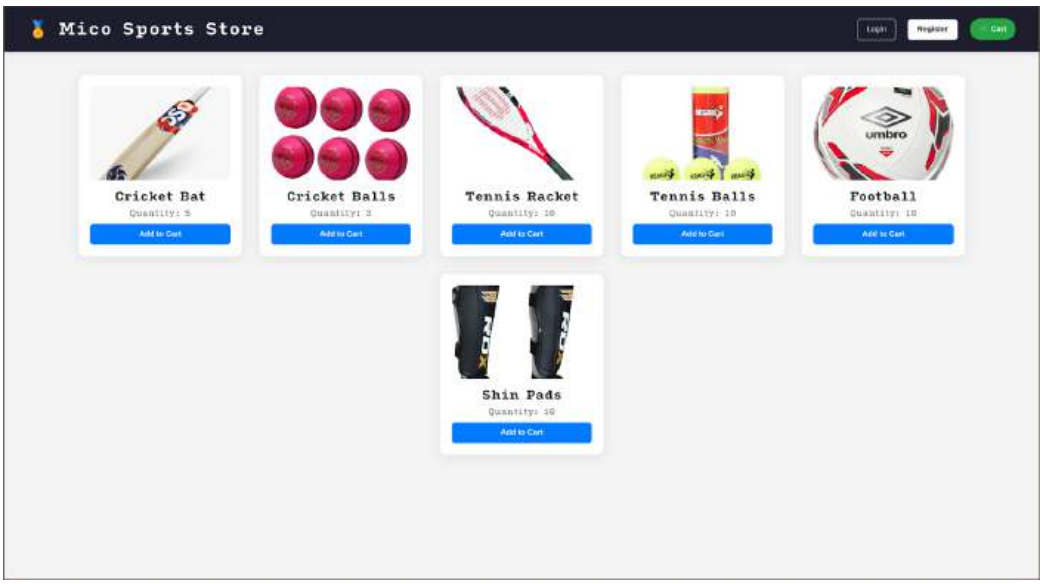

Figure 6.2: ECommerce Website

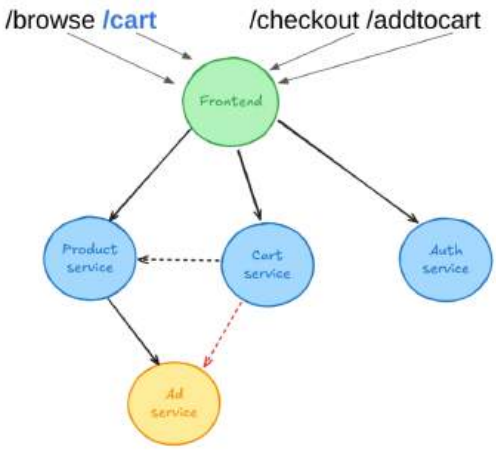

Figure 6.3: The architecture consists of 5 nodes containing multiple services, with 4 major user facing endpoints (excluding the login and register endpoints) worth profiling.

Chapter 7

Results

Our model of virtual speedup was able to correctly predict which service is worth optimizing and which is not.

7.1 Ad Service

The below graph shows the results of speeding up the ad service by 60% of its runtime on the /cart endpoint.

1. The left graph shows the raw latency distribution, more left skewed is better.
2. The right graph shows the cumulative distribution function of each distribution, once again left is better, showing less latency for each percentile value

From the CDF we clearly see that virtual speedup shows significant improvement in performance from the baseline, the difference between p99 values of baseline test and virtual speedup test is ¿90ms The Mann Whitney distribution comparison test confirms this and tells us that the latency distribution after virtually speeding up the ad service is stochastically lesser than the baseline.

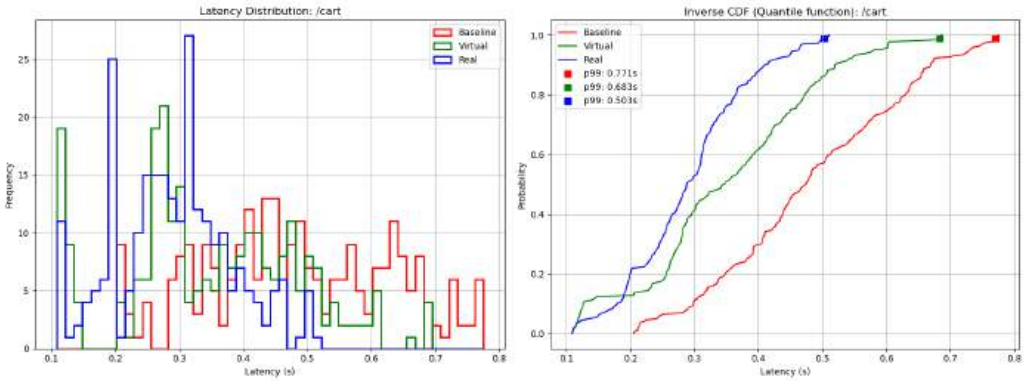

Figure 7.1: ad service, looking at the cart endpoint

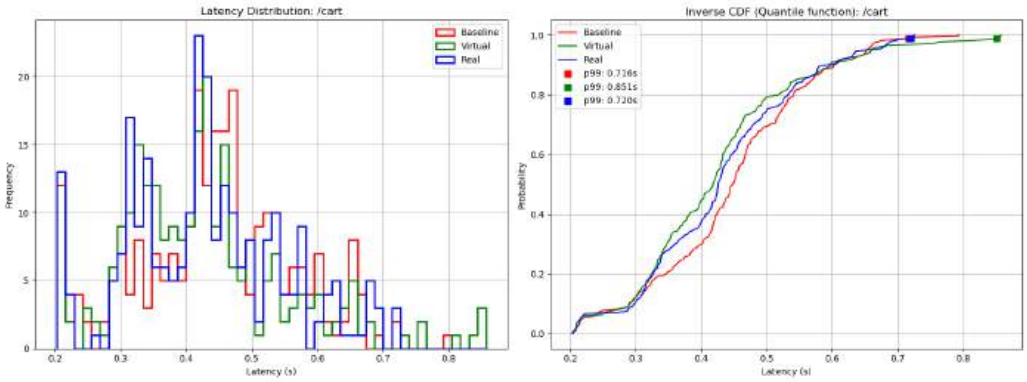

Figure 7.2: product service, looking at the cart endpoint

7.2 Product Service

The above graph shows the results of speeding up the product service by 60% of its runtime on the end to end latencies of the /cart endpoint.

Notice how the CDF plots are a lot more closely coupled together, telling us that there isn't much improvement in latency for the /cart endpoint by optimizing the product service.

This is confirmed by the Mann Whitney distribution comparison test that tells us that the latency distribution after virtually speeding up the product service is stochastically the same as the baseline (no real improvement).

Our virtual speedup model also correctly predicts that while speeding up the product service will not have an effect on the /cart endpoint's performance, it will reduce the latency of the /browse endpoint

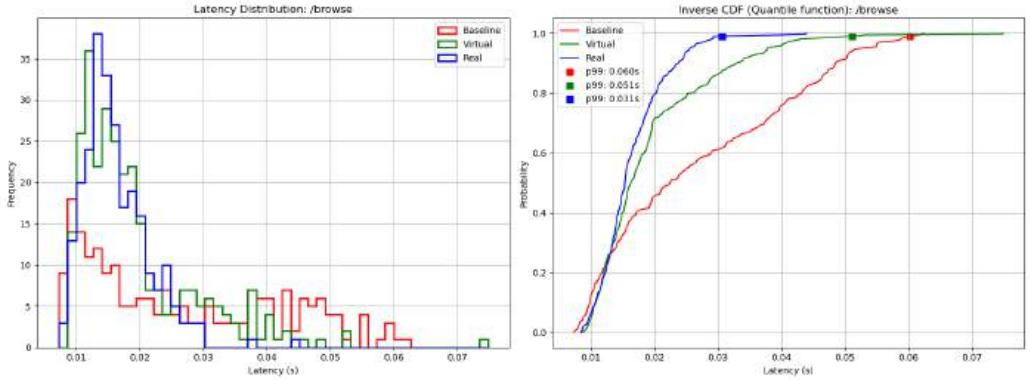

Figure 7.3: Speeding up the product service, looking at the browse endpoint

The above graph shows the results of speeding up the product service by approx. 60% of its runtime on the end-to-end latencies of the /browse endpoint.

Chapter 8

Conclusions and Future Work

Our causal profiling model successfully shows which service is worth optimizing and which is not for a particular endpoint in a real microservice with all its unpredictability and non-determinism.

8.1 Conclusions

1. It is possible to causally profile some microservice DAGs
2. This applies not just to simulated microservice DAGs but real microservices, as long as certain assumptions are made and appropriate statistical analysis methods are used.

8.2 Future Work

As for the future scope and work that can be built on top of the current base:

1. Reducing the margin of error between the latencies shown by really speeding up a service, and latencies predicted by virtually speeding up a service.
2. Incorporating microservice architectures where:
   (a) There are shared resources that are limited to a fixed number of services at a time.
   (b) Services making non-blocking network calls to other services
   (c) There are inherently asynchronous components like message queues
3. Relaxing some of the assumptions made about latencies between services and the latency to get and increment the global delay store begin negligible.
4. Pivoting to take a more fine grained approach, moving our focus from which service in a microservice architecture is worth optimizing $\rightarrow$ Which line of code in which service is worth optimizing. One approach to this could be integrating the COZ profiler itself into our work, and may help overcome some of the assumptions made above and extend the scope of the project.

Bibliography

[1] Akamai Technologies, "Akamai online retail performance report: Milliseconds are critical." https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report, 2017. [Online; accessed 3-August-2025].

[2] C. Curtsinger and E. Berger, "COZ: Finding Code that Counts with Causal Profiling," Tech. Rep. UM-CS-2015-008, University of Massachusetts Amherst, 2015.

[3] Jörg Thalheim and Antonio Rodrigues and Istemi Ekin Akkus and Pramod Bhatotia and Ruichuan Chen and Bimal Viswanath and Lei Jiao and Christof Fetzer, "Sieve: Actionable Insights from Monitored Metrics in Microservices." https://arxiv.org/abs/1709.06686, 2017. [Online; accessed 3-August-2025].