RESEARCH-ARTICLE

The Tale of Errors in Microservices

I-TING ANGELINA LEE, Washington University in St. Louis, St. Louis, MO, United States

ZHIZHOU ZHANG, Uber Technologies Inc, San Francisco, CA, United States

ABHISHEK PARWAL, Uber Technologies Inc, San Francisco, CA, United States

MILIND CHABBI, Uber Technologies Inc, San Francisco, CA, United States

Open Access Support provided by:

Uber Technologies Inc

Washington University in St. Louis

The Tale of Errors in Microservices

I-TING ANGELINA LEE*, Washington University in St. Louis, USA
ZHIZHOU ZHANG, Uber Technologies Inc., USA
ABHISHEK PARWAL, Uber Technologies Inc., USA
MILIND CHABBI, Uber Technologies Inc., USA

Microservice architecture is the computing paradigm of choice for large, service-oriented software catering to real-time requests. Individual programs in such a system perform Remote Procedure Calls (RPCs) to other microservices to accomplish sub-tasks. Microservices are designed to be robust; top-level requests can succeed despite errors returned from RPC sub-tasks, referred to as ***non-fatal errors***. Because of this design, the top-level microservices tend to "live with" non-fatal errors. Hence, a natural question to ask is "how prevalent are non-fatal errors and what impact do they have on the exposed latency of top-level requests?"

In this paper, we present a large-scale study of errors in microservices. We answer the aforementioned question by analyzing 11 Billion RPCs covering 1,900 user-facing endpoints at the Uber serving requests of hundreds of millions of active users. To assess the latency impact of non-fatal errors, we develop a methodology that projects potential latency savings for a given request as if the time spent on failing APIs were eliminated. This estimator allows ranking and bubbling up those APIs that are worthy of further investigations, where the non-fatal errors likely resulted in operational inefficiencies. Finally, we employ our error detection and impact estimation techniques to pinpoint operational inefficiencies, which a) result in a tail latency reduction of a critical endpoint by 30% and b) offer insights into common inefficiency-introducing patterns.

CCS Concepts: • **Software and its engineering** → **Cloud computing**; • **Computing methodologies** → **Modeling methodologies**; • **Computer systems organization** → **Reliability**.

Additional Key Words and Phrases: Microservices, Non-fatal Errors, Latency Estimation, Critical Path Analysis, RPC Errors, Performance Optimization

ACM Reference Format:
I-Ting Angelina Lee, Zhizhou Zhang, Abhishek Parwal, and Milind Chabbi. 2024. The Tale of Errors in Microservices. *Proc. ACM Meas. Anal. Comput. Syst.* 8, 3, Article 46 (December 2024), 36 pages. https://doi.org/10.1145/3700436

1 Introduction

Microservice architecture has become the preferred software architecture for the development of service-oriented systems in recent years [6, 16, 37, 40]. Unlike early monolithic systems, where functionalities were integrated into a single unit, microservices offer a decentralized approach, dividing the application into a collection of loosely coupled, independently deployable programs.

*Work done during a sabbatical as a visiting Research Scientist at Uber.

Authors' Contact Information: I-Ting Angelina Lee, Washington University in St. Louis, St. Louis, Missouri, USA, angelee@wustl.edu; Zhizhou Zhang, Uber Technologies Inc., Sunnyvale, California, USA, zzzhang@uber.com; Abhishek Parwal, Uber Technologies Inc., Sunnyvale, California, USA, abhishek.parwal@uber.com; Milind Chabbi, Uber Technologies Inc., Sunnyvale, California, USA, milind@uber.com.

ACM 2476-1249/2024/12-ART46
https://doi.org/10.1145/3700436

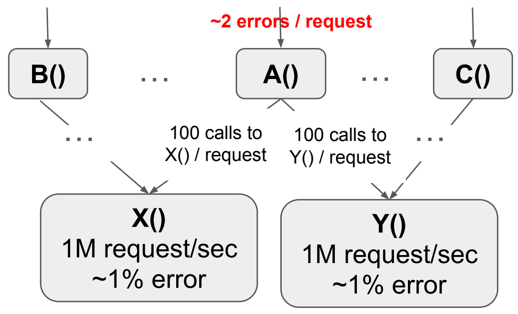
~2 errors / request
B()
A()
C()
100 calls to X() / request
100 calls to Y() / request
X() 1M request/sec ~1% error
Y() 1M request/sec ~1% error

Fig. 1. X is an interior API with 1 million queries per second and a 1% error rate, called by many upstream callers. A makes 100 RPCs each to X and Y in each of its requests, incurring ~ 2 erroring RPCs per request.

Each microservice handles specific business functionalities, promoting agile development and scalable deployment.

User requests entering a microservice system are broken down into smaller tasks (e.g., authentication, localization, payment processing) and executed by the relevant microservices, which may further decompose the tasks. Typically, each microservice is stateless, with all necessary context carried by the incoming request. The state, if necessary, is maintained in a per-service backing data store [1, 15, 30]. Inter-microservice communication is facilitated through standard messaging techniques like REST [18], gRPC [24], Thrift [44], and YARPC [3].

In the context of long-running microservices, an RPC is considered as ***erroring*** or ***failed*** if it returns a non-success status code (e.g., entity not found) to its caller. Such an erroring RPC typically does not result in the termination of the caller or the callee process; it only conveys the status of the operation.

Resiliency, a key feature of microservices, ensures continued operation despite failures in any part of the system. This isolation helps prevent system-wide failures common in monolithic architectures. However, the segregated nature of microservices can obscure inefficiencies. For instance, one part of the system might not know about an erroring RPC in another part, leading to redundant RPCs. Oftentimes, interior RPCs can fail, and yet the top-level user request succeeds nonetheless. We call such errors as "non-fatal" — specifically, given a top-level user request that succeeds, we define its erroring interior RPCs as ***non-fatal errors***; conversely if the corresponding user request fails, we define erroring interior RPCs as ***fatal errors***.

Non-fatal errors can increase operational costs and degrade user experience, yet they often go unnoticed in resilient microservice architectures. Although service owners monitor the error rates of individual microservices, they often lack a "global" perspective on the broader consequences of non-fatal errors. For example, consider an interior API `X()` depicted in Figure 1, which processes a million queries per second and has a 1% error rate. A user-facing endpoint `A()`, calling `X()` 100 times per inbound request, will likely experience one error out of 100 in its calls to `X()`. If the error returned from `X()` is non-fatal, it will not be apparent to the owners of `X()` or `A()`. If `A()` also interacts with another API `Y()` with similar error rates, the cumulative errors make `A()` experience about two errors in its RPC subgraph for each of its incoming requests, yet these issues remain hidden due to isolated monitoring. Unfortunately, non-fatal errors are common and can arise from system evolution [28, 41, 42], experimental features [12, 39], and migrations and decommissioning [21]. Preventable non-fatal errors can happen due to developers' inattention to performance. Developers working on high-level applications or business logic can be unaware of

```go
func GetUserInfo(user User) (Info, error) {
    info, err := CheckInAmericasDB(user)
    if  err == nil {
        return info, nil
    }
    info, err = CheckInEuropeDB(user)
    if err == nil {
        return info, nil
    }
    info, err = CheckInAsiaPacDB(user)
    if err == nil {
        return info, nil
    }
    return nil, NOT_FOUND
}
```

Listing (1) Simplistic If Version

```go
func GetUserInfo(user UserT) (Info, error) {
    switch user.Origin {
    case AMERICAS:
        return CheckInAmericasDB(user)

    case EUROPE:
        return CheckInEuropeDB(user)

    case ASIAPAC:
        return CheckInAsiaPacDB(user)

    default:
        return nil, NOT_FOUND
    }
}
```

Listing (2) Switch-Case Version

Fig. 2. Example of Golang code with non-fatal errors (left) and an alternative optimized version (right).

the performance consequences of simple coding idioms, which can leave them detached from the underlying layers of libraries and system software.

In this paper, we investigate non-fatal errors at user-exposed endpoints in production microservices. Our research questions explore the prevalence of non-fatal errors, their impact on latencies, and any common patterns that result in non-fatal errors. We systematically analyze the significance of non-fatal errors in microservices based on more than 6,000 microservices hosted inside the Uber company—a large-scale service-oriented software industry catering to real-time requests from more than a hundred million active users in multiple business domains. Unlike some of the giants from the early days of the internet [27, 42], which began as monolithic systems, Uber's system is unique — it is designed from the ground up using microservice architecture and has witnessed an explosion of microservices in a decade. This system serves as a prototypical setting for answering the research questions discussed in this paper. This setting provides a prototype for our investigation, with findings relevant to all microservice systems.

To make the notion of non-fatal errors concrete, consider the following example, a microservice API `GetUserInfo` which retrieves information about a user. The use may be present in only one of the regionally distributed databases —Americas, Europe, and Asia-Pacific. The developer follows a simplistic `if` check coding idiom that checks each database in some sequence until it finds the user, as shown in Figure 2. Notice here that if the user is found in `CheckInEuropeDB`, it would first resulted in a (non-fatal) error return from `CheckInAmericasDB`. Similarly, if the user is found in `CheckInAsiaPacDB`, this would have led to two prior RPCs that caused non-fatal errors. From the code point of view, there is no exceptional flow of control. In this code snippet, even if the developer may have prioritized the `CheckInAmericasDB` with knowledge of the common users, the tail latency will be dominated by finding the user in the `CheckInAsiaPacDB`, which incurs two non-fatal errors on its path. One may be able to avoid the non-fatal errors and its latency impact by utilizing some additional knowledge of the user's origin to directly query the appropriate database, as shown in Listing 2.

Empirical results from our study demonstrate that non-fatal errors are frequent and correlate with increased latency in critical services, although not all non-fatal errors significantly affect end-to-end latency.[1] Assessing their impact requires sophisticated analysis to estimate how non-fatal errors affect a request's RPC journey and to identify which endpoints would benefit most from error mitigation. We implemented our methodology across all endpoints at Uber, focusing

[1]Part of the data is publicly accessible at https://doi.org/10.5281/zenodo.13947828.

on a select few that led to valuable case studies, revealing and sometimes resolving operational inefficiencies.

Our contributions are threefold:

- We systematically study RPC errors in large-scale microservices, demonstrating that RPC errors are common and can significantly affect tail latencies.
- We develop a methodology to predict latency reductions and identify microservices with operational inefficiencies due to errors.
- We apply this methodology in an industrial context, presenting case studies that led to operational improvements, including one that resulted in a 30% latency reduction in a critical service.

The study presented in this paper highlights a significant, yet often overlooked, inefficiency in microservice architectures: the prevalence of non-fatal errors that do not cause top-level request failures but contribute substantially to increased latency. These errors, while not immediately fatal to system operations, can quietly degrade performance and user experience. The study presented in this paper not only quantifies the impact of these errors on latency in a large microservice ecosystem but also offers a practical methodology to reduce them. Our methods lead to measurable performance improvements and allow for better prioritization of APIs that need optimization. The methodology presented here guides anyone looking to optimize large-scale distributed systems towards better reliability and efficiency.

The rest of the paper is organized as follows. Section 2 provides background information. Section 3 presents the data analysis on non-fatal errors. Section 4 provides deeper analysis on the impact of non-fatal errors by applying the methodology and evaluating latency reduction potential of endpoints. Section 5 describes the methodology in detail. Section 6 discusses the case studies that used the methodology to identify and pinpoint operational inefficiencies. Section 7 summarizes related work. Finally, Section 9 offers conclusions.

2 Background

In this section, we define the terminologies and provide the background material necessary to understand our work.

2.1 Terminology

Endpoint vs. API.. Each microservice exposes one or more callable methods. One microservice is special, called the ***gateway***, which exposes its methods to external users invocable by an edge device such as a web browser or a mobile app; other microservices are internal and their methods can only be accessed through the gateway or another internal microservice. For clarity, we refer to gateway methods as ***endpoint***s and the methods in internal services as ***API***s.

Request vs. RPC.. A real-time query arriving at an endpoint in the gateway is called an ***external request***. To handle a request, the gateway endpoint invokes internal APIs via Remote Procedure Calls (RPCs). We refer to an *instance* of invoking a user-exposed endpoint simply as a ***request*** and an *instance* of invoking an internal API simply as an ***RPC***. Every request and its nested RPCs form a tree rooted at a gateway endpoint. Requests aggregated over all endpoints form a forest.

Successful vs. Failing Request. A ***request*** is deemed ***successful*** if the gateway endpoint responds with a success code (e.g., HTTP status code 200) to the edge device; otherwise, it is considered ***failed***. An RPC is deemed successful if the API returns a protocol-specific success code [2], otherwise, it is considered to have ***error***ed. It is important to emphasize that this study is not about analyzing errors or failures in the interconnection network stack; RPC libraries hide such errors from the applications, and RPC libraries are configured to automatically perform protocol-level retransmissions or retries [42]. Our focus is on the RPC error codes surfaced to the application layer, which can be "INVALID ARGUMENT" or "UNAUTHENTICATED," to name a few.

Fatal vs. Non-Fatal RPC.. Within a given failed request, if an interior RPC error propagates all the way to the root (i.e., the chain of RPCs between the root and the interior RPC all error out), we call such an RPC error ***fatal***. Conversely, if the interior RPC error does not propagate all the way to the root (e.g., some RPC along the chain to the root does not error), then we call such an RPC error ***non-fatal***. This definition implies that all RPC errors within a successful request are non-fatal. If none of the RPCs performed on the behest of a request experiences an error, this request is deemed ***error-free***.

2.2 Microservices at Uber and Their Generalizability

Uber hosts over 6,000 microservices. There is a diverse range of applications running, including, but not limited to: real-time data processing, user management, payment processing, advertising, maps, messaging, and notifications, experimentation, configuration management, analytics, and reporting, external API integrations, internal tools, and dashboards, machine learning and AI, caching, database, and document and blob storage. The complex software systems we analyze were developed by numerous teams, each employing different methodologies, languages, libraries, and tools. These teams also adhere to varying service-level metrics, such as latency, throughput, and error tolerance. Thus, despite being confined to one organization, our study captures a broad diversity of services.

Moreover, the widespread use of loosely coupled services, standardized communication protocols like RPC [24], Thrift [44], and YARPC [3], along with a strong reliance on distributed tracing for observability, suggests that the findings of this study are broadly applicable to other organizations employing similar microservice ecosystems [27, 34, 49]. These commonalities indicate that the patterns we identified, particularly those concerning the prevalence of non-fatal RPCs, their impact on latency, and strategies to optimize errors for reduced tail latency, are likely to hold in similar environments.

2.3 Distributed Tracing

It is difficult to trace the flow of requests as they traverse myriad APIs. Distributed tracing solves this problem by capturing and logging each RPC as the request traverses a distributed system.

Jaeger [8] is a distributed tracing tool suitable for monitoring requests in large-scale microservices. It integrates with modern application stacks using different RPC protocols and programming languages. Jaeger trace requests flowing through the microservice architecture by tagging them with metadata called ***trace context***, which includes a unique trace ID. The metadata is transmitted in-band (as part of the RPC) across the process boundaries. Jaeger instrumentation captures the duration of RPC, timestamp[2], and metadata (e.g., tags and logs) and combines them into an object called ***span***, which represents an RPC performed on behalf of the request. Spans are transmitted out of band (asynchronously, out of the critical path of the request) to Jaeger collectors. Collectors record spans in persistent storage, where they are later correlated via trace IDs into a single trace.

All services in Uber are a Jaeger-equipped instrument to track RPCs starting from the gateway at the sampling rate < 1%. Since tracing every request can inundate the system with voluminous data, at Uber, we trace every endpoint exposed by the user with an adaptive sampling rate < 1%. Once a request is marked for tracking at the gateway, its journey throughout the system is logged. Even at this sampling rate, we have access to ~840 Million traces formed out of ~210 Billion spans per day, or around 3 million spans per second. Our work described herein analyzes these traces

[2]Both start and end timestamp of a span are recorded using local machine time then converted to standard UTC time.

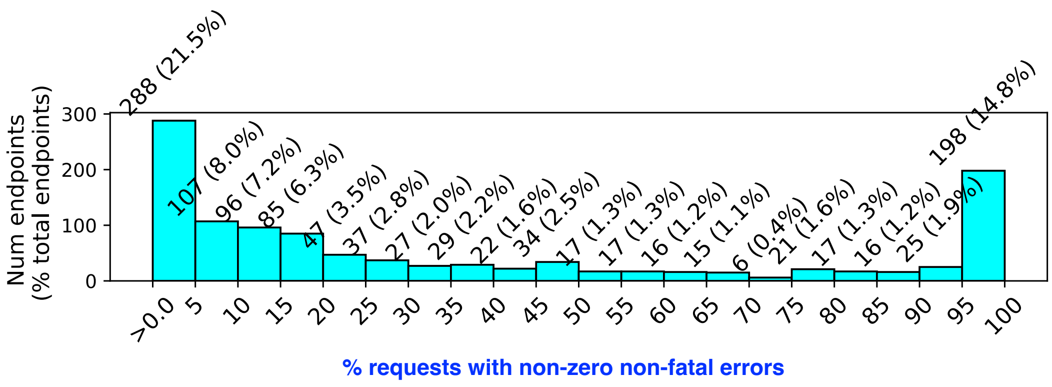

Fig. 3. Distribution of non-fatal errors over endpoints.

at scale. When describing our algorithms and evaluation, we use the terms ***request*** and ***trace*** interchangeably; similarly, we use the terms ***RPC*** and ***span*** interchangeably.

3 Characteristics of Microservice Errors

Uber has over 6,000 microservices with over 140K methods, and over 3,000 methods (endpoints) are user-exposed. We investigated a subset of 1,300 of these endpoints from the ***gateway*** service that showed activity in the trailing seven days via Jaeger tracing. We did not include the rest of the user-exposed endpoints into the study since they did not have enough incoming traffic[3]. In this section, we present our analysis and observations based on the data collected over traces from these endpoints. The dataset includes more than 1.7 million traces and 36 billion spans.

3.1 Non-fatal Errors are Common in Requests

Very few gateway requests fail; our study of sampled requests over all gateway endpoints showed that only ~0.65% of the requests failed. However, among the remaining ~99.35% requests,~29.35% requests contained at least one non-fatal error in their subtree of internal API calls. The remaining ~70% requests were error-free. By studying these non-fatal errors, we make the following observations.

To confirm that non-fatal errors are widespread rather than concentrated in a few highly utilized endpoints, we examined their occurrence across all user-exposed endpoints. Notably, 84% (1,120/1,341) of the endpoints encountered at least one non-fatal error during the study period, with only 16% (221 out of 1,341) consistently error-free.

Figure 3 displays a histogram correlating the fraction of requests with non-fatal errors (x-axis) to the number of endpoints (y-axis). The first histogram bin shows that 288 endpoints had between 0 and 5% of their gateway requests containing at least one non-fatal error. As the error rate increases, the number of affected endpoints decreases and then spikes again, where 198 endpoints have over 95% of their requests containing non-fatal errors—indicating these endpoints consistently operate with errors.

Observation 1. Most endpoints experience a non-trivial fraction of requests with non-fatal errors.

[3]It is not uncommon to have some endpoints unexercised for many days when they only serve specific situations such as holiday-theme, user unsubscription, payment cancellation, etc.

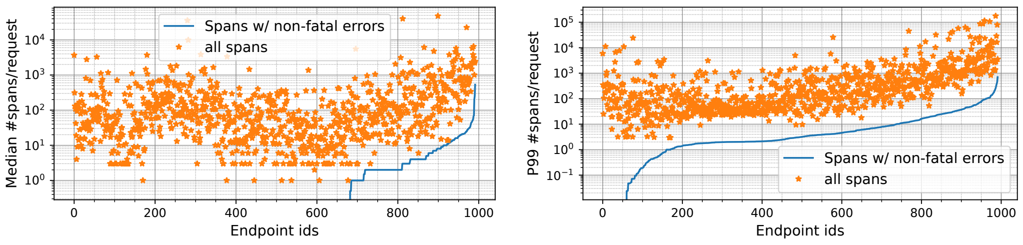

Fig. 4. Comparison of the number of spans with errors versus the total number of spans (in log scale) per successful gateway request. The left shows the request with the median number of spans for each endpoint and the right shows the P99. In both figures, the endpoint Ids are sorted in ascending order by the y-axis values corresponding to the number of spans with errors (the blue line). For every endpoint, there is one data point for the total number of spans and one data point for the number of spans with errors. Endpoints with less than 1,000 traces are not shown in both plots.

However, these data do not tell us how many non-fatal errors each gateway request likely experiences. Thus, we further compare the number of erroring spans against the total number of spans in a request and aggregate these counts across gateway requests for each endpoint.

Figure 4 illustrates this comparison, showing the number of erroring spans versus the total spans for successful gateway requests, plotted per endpoint on the x-axis. Since we collect multiple traces per endpoint, we show each endpoint's median (left) and 99th percentile (right) over the number of spans per request. We drop the endpoints with fewer than 1,000 sampled traces in the figure because the P99 computation needs at least 100 traces; we relied on 1,000 traces to get more statistically significant numbers. That leaves us with just under 1,000 endpoints. The blue line indicates the spans with errors, and the orange stars represent the total spans per request. At the median (left), nearly a hundred endpoints experience more than ten non-fatal errors per successful request. In P99 (right), over 200 endpoints see similar error counts. Some requests, both at the median and the P99, contain hundreds of non-fatal errors.

Out of all 11 Billion RPCs we observed during the last seven-day period, only 0.9% RPCs errored out. However, they disproportionately impacted 23% (12M out of 52M) traces. Thus, although the percentage of errors out of all RPCs is small, most traces contain at least one error because every request makes many RPCs and has a high chance of encountering an error.

Figure 5 shows the average percentage of error spans per trace for each endpoint with variance. Figure 6 plots the percentage of errors (y-axis) out of all RPCs made for a request. This figure illustrates a similar point to that of Figure 4 but is plotted from a different perspective — it plots the fraction of RPCs that errored out compared to the total number instead of the count. As we can see from the figure, some requests can have large fractions of their RPCs error out. These figures lead to our next observation.

Observation 2. Requests with non-fatal errors often contain multiple RPC errors.

3.2 Why Requests Experience Non-fatal Errors

Error Types. To understand why gateway requests experience non-fatal errors frequently, Table 1 shows the distribution of RPC errors found in traces with non-fatal errors. Perhaps surprisingly, our finding is different from a prior study [42], which had found that ***request hedging*** [11] causes a lot of `canceled` requests. With request hedging, the caller issues multiple copies of the same

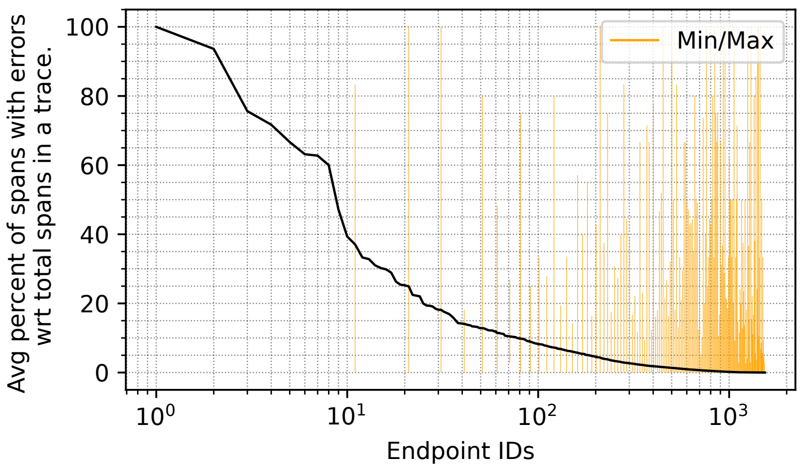

Fig. 5. Average percent of error spans per trace in each endpoint. The min-max bars show the variance.

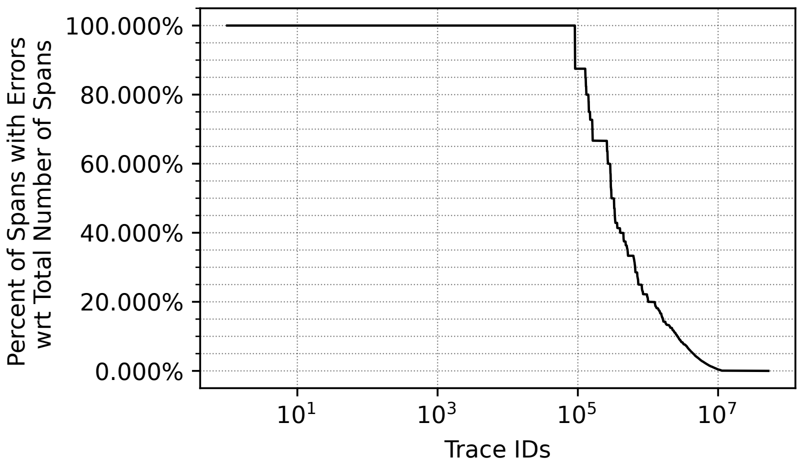

Fig. 6. Traces vs. percent error spans.

request to different backends, uses the first response it receives, and cancels the other outstanding ones.

In our study, we found that canceled requests constitute only a small fraction of the erroring RPCs; instead, a large fraction of non-fatal errors stem from `entity not found` (missing items, such as a user profile not found in the database), `aborted` (concurrency related issues, such as database transaction aborts), `failed precondition` (where the operation fails due to certain unmet pre-condition, such as failing to delete a user that does not exist in the database), and `resource exhausted` (intentional rate limits imposed on the servers to avoid overwhelm). The largest category, `entity not found` errors, is interesting — these errors often result from coding patterns

Table 1. gRPC errors in traces with non-fatal errors.

ENTITY NOT FOUND	42.46%	DEADLINE EXCEEDED	0.19%
ABORTED	23.18%	UNKNOWN	0.20%
FAILED PRECONDITION	16.47%	PERMISSION DENIED	0.10%
RESOURCE EXHAUSTED	16.29%	ALREADY EXISTS	0.06%
CANCELLED	0.31%	INTERNAL	0.05%
INVALID ARGUMENT	0.68%	UNAUTHENTICATED	$\sim$ 0.00%
UNAVAILABLE	0.01%	UNIMPLEMENTED	$\sim$ 0.00%

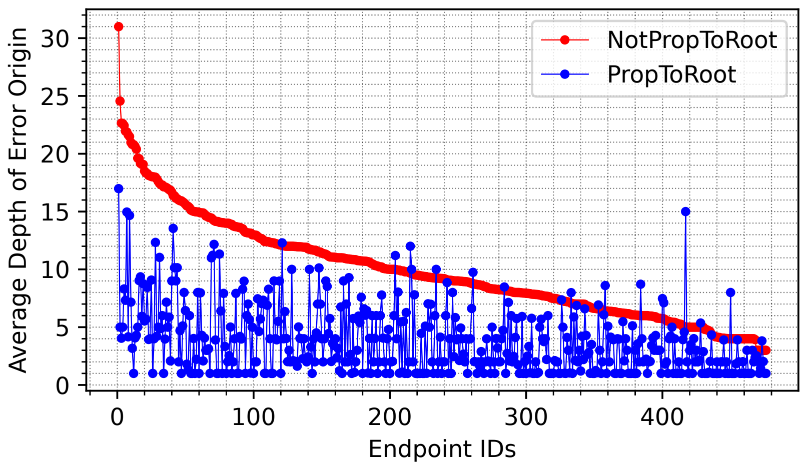

Fig. 7. Average depth of origin of errors propagating to root (fatal) vs. not propagating to root (non-fatal). Only endpoints with at least one trace with a fatal error are considered for comparison.

leading to potentially wasted work due to RPCs with non-fatal errors; this topic was illustrated with a code example in Section 1, or from failures of user customizations, which we elaborate further in Section 6.4. The `aborted` and `resource exhausted` errors are not entirely avoidable; however, knowing in which endpoints they are frequently encountered can help developers identify bottlenecks in microservice architectures. Finally, the `failed precondition` often results from operations that are meant to be best effort. Such errors may be hard to eliminate to reduce latency.

Observation 3. Majority of the non-fatal errors cluster around the following four categories: `entity not found`, `aborted`, `failed precondition`, and `resource exhausted`.

Beyond the types of error, we also collected data on the various characteristics of these errors.

Error Provenance Depth. We analyze the provenance of errors by measuring their depth from the root (the gateway RPC with depth zero). Figure 7 illustrates this comparison between non-fatal errors (`NotPropToRoot`, blue line) and fatal errors (`PropToRoot`, red line) across various endpoints. Only endpoints with at least one trace with a fatal error, in addition to at least a thousand total traces as explained previously, is required for this comparison, which left us with about 500 endpoints for analysis. The graph shows the average depth for each type of error, plotted for endpoints that have

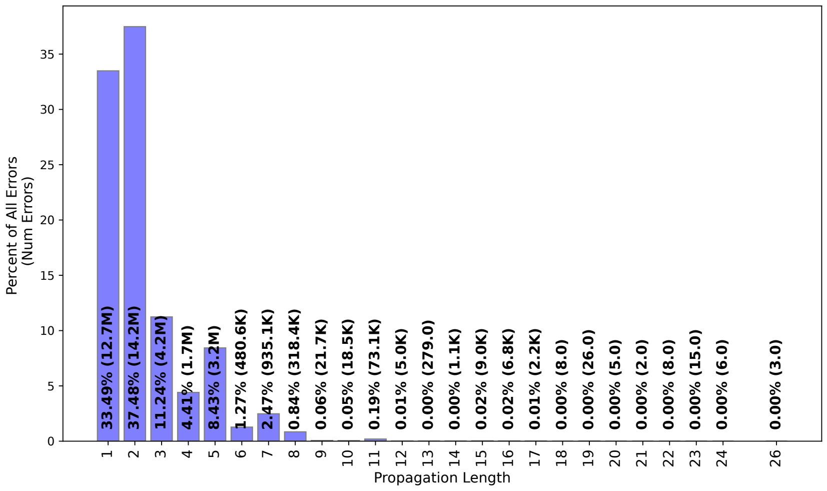

Fig. 8. Length of error propagation before they are squashed.

instances of both types of error. The analysis reveals that non-fatal errors typically occur deeper within the RPC graph than fatal errors for the same endpoint.

Error Propagation Length. We also collected the ***propagation length*** of errors, defined as the difference between the depth of the RPC where the error originates and the depth of the caller RPC that squashes the said error. Most errors have a short propagation length, as shown in Figure 8. More than 80% errors have a propagation length of three or fewer before they are squashed by an RPC caller. Only 0.3% errors have propagation lengths greater than 10. The longest propagation length observed was 26.

The data on these error characteristics leads to the next observation.

Observation 4. Non-fatal errors tend to originate deeper in the graph than fatal ones and have a short propagation length.

This makes intuitive sense, since microservices are resilient by design, — they tend to isolate and mitigate failures due to their loosely coupled nature. We hypothesize that many service APIs, especially those with lower depth (i.e., closer to the root-level gateway request in the call chain), are written to be robust to errors bubbling up from their callees.

API Resiliency. To test our hypothesis regarding API resilience, defined as an API's ability to prevent errors from propagating to the caller, we first set definitions for measuring this trait. For each API, we track two types of error counts: stopped errors (E_{stopped}), where the API does not return an error despite receiving one from its callees, and propagated errors ($E_{\text{propagated}}$), where both the API and at least one of its callees error out. The resiliency of an API is then calculated using the formula: $E_{\text{stopped}}/(E_{\text{stopped}} + E_{\text{propagated}})$.

Figure 9 shows the resilience scores for approximately 6,000 internal APIs called from the gateway service. The distribution reveals a bimodal pattern: APIs typically either always stop errors (about 29%) or always propagate them (about 63%), with only about 8% showing mixed behavior. This variability can be attributed to APIs reacting differently based on the specific callee and the type of error returned.

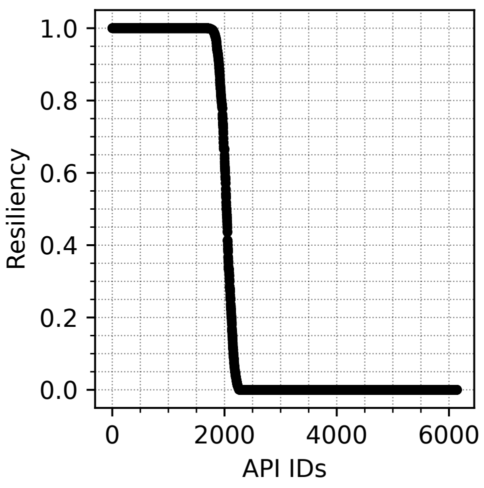

Fig. 9. Resiliency of an API to stop errors. Here all endpoints are internal endpoints that gateway service calls.

Observation 5. The resiliency of internal APIs exhibits a bimodal distribution—APIs predominantly either completely stop or propagate errors from subordinate RPCs.

This bimodal distribution supports our hypothesis that errors that originate deeper in the system are more likely to be mitigated by highly resilient APIs before reaching the top-level RPC.

3.3 Non-Fatal Errors Correlate with Prolonged Latency

We have also collected data from endpoints that experience both error-free requests and requests with non-fatal errors to compare their characteristics. Similarly to Figure 7, only those endpoints that have trace samples for both types of errors are shown in the figures below. Our analysis confirms a correlation between the presence of non-fatal errors, increased overhead, and prolonged latency.

Figure 10 compares the total number of spans per error-free request against that of requests with non-fatal errors. Here, we use the total number of spans in a trace as a proxy for the amount of work performed. The data is shown for each endpoint on the x-axis. Since the data are aggregated over many requests per endpoint, we show the median of the values on the y-axis. The blue line shows the median number of spans in error-free requests, whereas the orange line shows the median number of spans in requests with non-fatal errors. The number of spans in requests with non-fatal errors is almost always greater than those in error-free requests. On average, requests with non-fatal errors have 1.9× more spans than the error-free requests for the same endpoint.

Observation 6. Requests with non-fatal errors often perform more work than error-free requests.

We also collected and compared the latency of these requests. Figure 11 compares the median latency of error-free requests with those that have non-fatal errors. The x-axis represents different endpoints, and the y-axis is the median latency of requests. The blue line shows error-free requests whereas the orange line shows requests with non-fatal errors. The latency of requests with non-fatal errors is on average 1.8× more than the latency of error-free requests. The P99 latency (not shown) of requests with non-fatal errors is on average 2.9× more than the P99 latency of error-free requests.

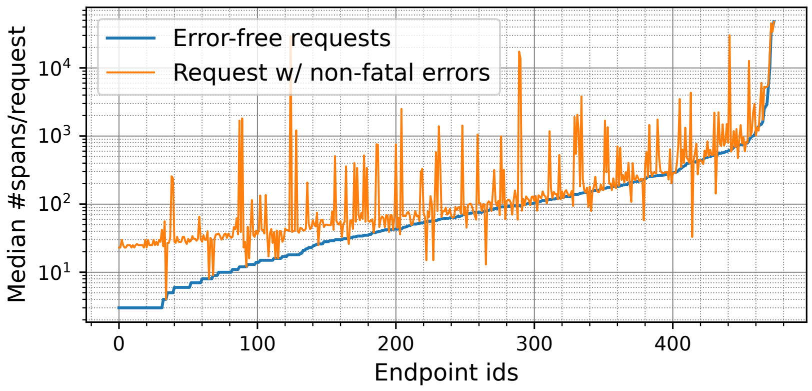

Fig. 10. No. of spans per request is higher for requests with non-fatal errors compared to that of error-free requests.

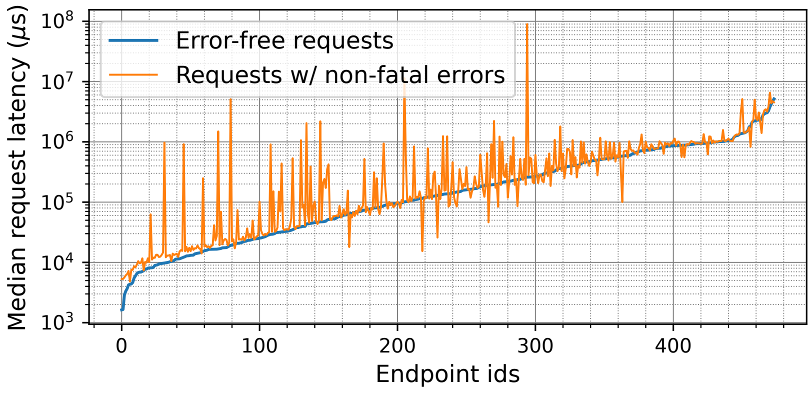

Fig. 11. Latency is higher for requests with non-fatal errors compared to that of error-free requests.

Observation 7. The latency for requests with non-fatal errors is often more than the latency for error-free requests.

4 Mitigating the Impact of Non-Fatal Errors on Latency: Challenges and A Solution

Data in Section 3 shows that non-fatal errors are common, and their occurrences seem to correlate with prolonged latency. Naturally, we wanted to know whether it is possible to identify and remove non-fatal errors that lead to inefficiencies and thereby improving request latency of the endpoints that suffer from non-fatal errors. Due to the large number of endpoints with non-fatal errors and limited engineering resources, it is not possible to investigate all these endpoints. In this section, we first describe the challenges in identifying the right endpoints and non-fatal errors to investigate, discuss the Latency-reduction estimator, a methodology we developed to assess more accurately impacts of non-fatal errors have on a given endpoint, present data showing that non-fatal errors

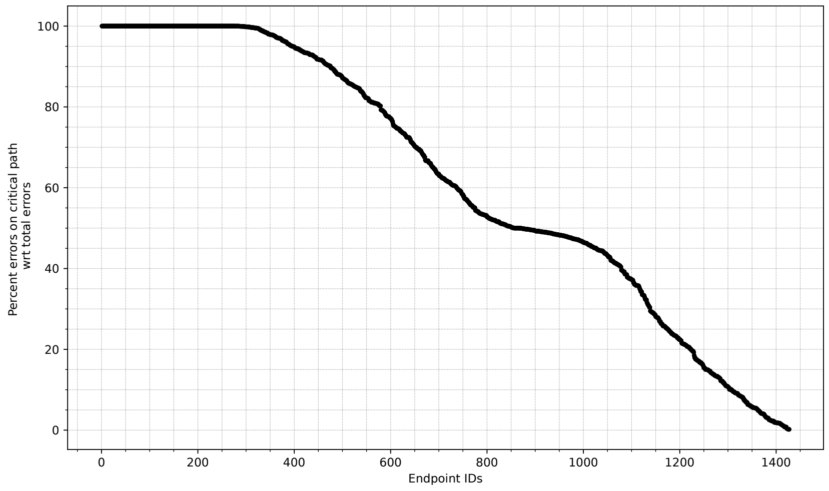

Fig. 12. Percentage of errors happening on the critical path w.r.t. total errors for each endpoint.

can indeed impact the latency of many endpoints, and how we can use the methodology to identify which endpoints warrant further investigation more effectively.

4.1 Challenges of Identifying Endpoints with Non-Fatal Errors to Investigate

To identify the right endpoints with non-fatal errors to investigate, one simple approach may be to simply start with endpoints that have the highests count of non-fatal errors. This simple approach does not work, because not all non-fatal errors contribute to the latency. RPCs within a request are highly asynchronous — APIs often make multiple concurrent [27, 55] downstream RPCs, and endpoint latency is determined by the critical path, which is the longest sequence of dependent tasks (RPCs) in the request. Importantly, many non-fatal errors do not occur on the critical path.

Figure 12 displays the percentage of errors in the critical path relative to total errors per endpoint, aggregating the errors in the traces of each endpoint and sorting them by the fraction of errors on the critical path. Notably, for about 15% of the endpoints, all errors are on the critical path. However, for most (85%) of the endpoints, not all errors affect the critical path. If an error is not on the critical path, it does not contribute to the observed latency.

Prior research like CRISP [55], DAPPER [14, 43], and CANOPY [29] has highlighted the importance of critical path analysis [52] in analyzing and optimizing latency in distributed systems.

What if we target endpoints with high counts of non-fatal errors on the critical path? This approach may seem to target only non-fatal errors that contributes to latency. Still, it does not work as intended because higher error counts do not necessarily translate to higher latency, even if one focuses on the critical path. Figure 13 shows a flamegraph that highlights APIs that contribute to the critical path of a particular endpoint — the longer the width of the bar, the longer the time such an API contributes to the endpoint latency. Furthermore, Figure 13 also shows the accumulated counts of non-fatal errors that occurred in certain APIs. As we can see in the figure, APIs with higher error counts do not necessarily correspond to higher contributions to the latency.

Finally, a different approach could be to measure the impact of non-fatal errors on the critical path following an algorithm analogous to [55]. However, this method is flawed as eliminating erroring spans could alter the critical path, meaning that the endpoint latency might not decrease proportionally to the eliminated error spans.

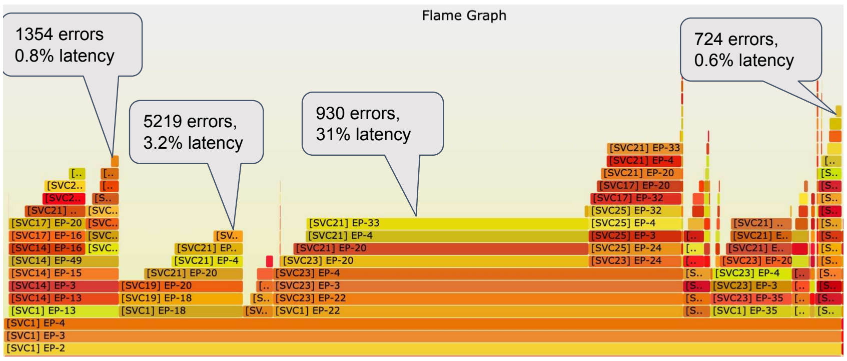

Fig. 13. Latency vs. error counts.

4.2 Latency-Reduction Estimator

To better estimate the impact of non-fatal errors, we need a more sophisticated strategy to compute the hypothetical latency if non-fatal errors were eliminated. We have developed a methodology, referred to as the ***latency-reduction estimator*** (or ***LR estimator*** for short), that computes the hypothetical latency of an "error-eliminated" trace given the observed trace, and thereby estimating the potential latency reduction.

The LR estimator operates under the assumption of a theoretical oracle that advises against invoking any RPC that will fail. This oracle is practical in our postmortem analysis of execution traces. It helps project an upper bound on latency reduction; however, the estimator also makes pragmatic estimations that adhere to certain understood program semantics.

Given a Jaeger trace of the original execution, the LR estimator computes the hypothetical latency of the ***error-eliminated*** trace by assuming that a) all RPCs errored out in the original execution have *zero* duration in this hypothetical execution, and b) all "dependences" among RPCs (causal relationships) are preserved. For instance, if span B starts only after span A completes in the original execution, span B starts only after span A completes in this hypothetical execution. For another instance, if some time in span A elapses before A invokes a subtask RPC span B in the original execution, the same amount of time is preserved in the error-eliminated trace. The LR estimator conservatively assumes that such dependences inferred from the original execution based on the Jaeger trace are dictated by the program semantics and should not change even if we are able to eliminate all non-fatal errors.

We use the example traces shown in Figure 14 to illustrate what LR estimator computes. The top of Figure 14 shows the original execution trace of a request, where spans C, I, and L errored out; the bottom shows its corresponding error-eliminated trace as inferred by LR estimator. In both traces, the critical path is highlighted in a striped pattern.

In the error-eliminated trace, eliminating span C allows spans D, E, and F to move forward without changing any dependencies among them, and shortens the duration of its parent, span B. A similar situation occurs for the elimination of span I, but span J does not move forward by the length of I, both because in the error-eliminated execution, it should remain that J occurs after H, and that its parent, span G performs the same amount of work after H completes before invoking J. Lastly, the elimination of span L does not impact the duration of its parent, span K, because the work necessary in K before invoking span M remains.

This example illustrates why eliminating non-fatal errors completely does not necessarily reduce latency by the amount equal to the length of the erroring spans eliminated. First, eliminating an

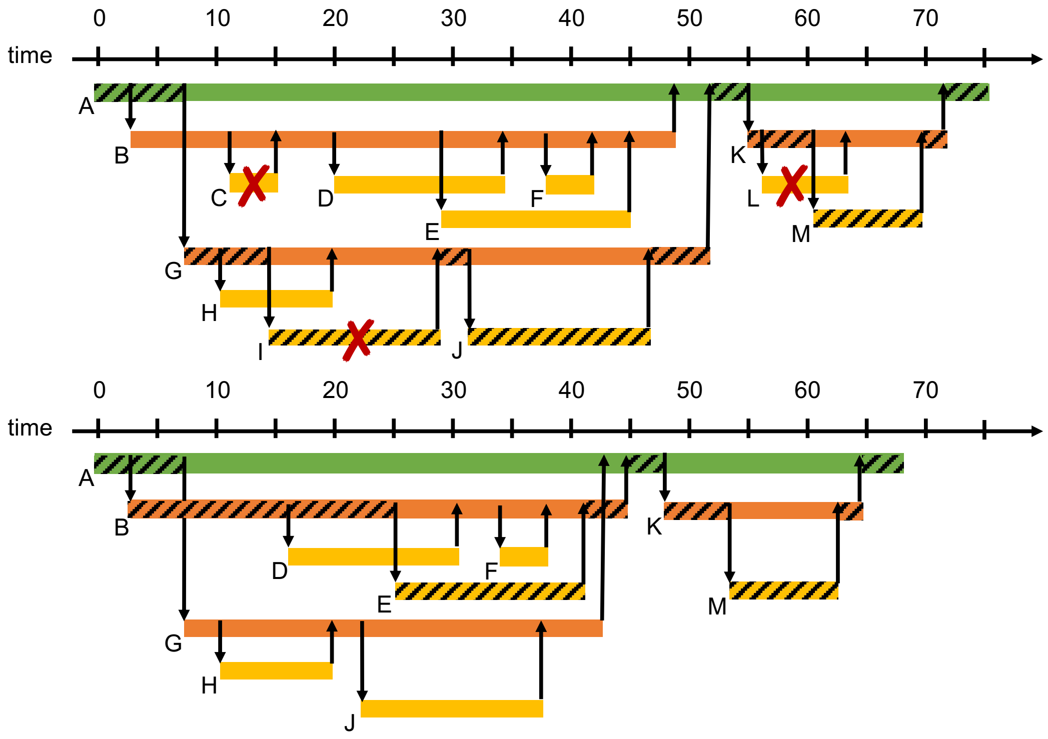

Fig. 14. An example illustrating the original (top) and error-free (bottom) execution traces. Each rectangle represents a span (an RPC call). The arrows indicate the start and end time of RPCs, where a downward arrow (from parent to child) indicates the invocation of a child and an upward arrow (from child to parent) indicates the termination of the child span. The critical path is highlighted with a striped pattern, and the red X indicates the span that errored out.

erroring span may not necessarily reduce the duration of its parent (e.g., L); or, even if it does, because of the changes in duration in its parent, the critical path may have shifted away (e.g., I), and now the latency is dictated by a different path, which must have a longer duration than the original path with latency reduction.

Finally, notice that, when we go from the original trace to the error-eliminated trace, not only does the critical path change, but the alternative critical path in the error-eliminated trace must also consider the impact of eliminating erroring spans and thus the latency of the error-eliminated trace is not simply the length of the alternative critical path in the original execution. The algorithm used in LR estimator (details in Section 5) accounts for these cascading effects when computing the hypothetical latency.

4.3 Measuring the Latency Reduction Potential Using the LR Estimator

Using LR estimator discussed above, we performed analyses on the same set of traces used in Section 3 and observed the following.

Table 2 bins the endpoints by their saving potential at different tail latencies. Tail-1% shows how much the worst 1% (averaged from 99th to 100th percentile) of the latency can be reduced by eliminating errors in those traces. Tail-5, Tail-10, Tail-20, and Tail-50 are defined similarly.

% latency	Number of endpoints				
reduction	Tail-1	Tail-5	Tail-10	Tail-20	Tail-50
(10, 20]	62	62	60	57	60
(20, 30]	24	22	22	26	25
(30, 40]	12	11	11	8	6
(40, 50]	8	10	9	8	13
(50, 60]	7	6	7	9	3
(60, 70]	7	7	7	6	8
(70, 80]	5	5	5	5	7
(80, 90]	6	6	6	5	2
(90, 100]	1	1	1	2	3

Table 2. Endpoint count with tail latency-saving potentials.

We notice that 62 endpoints can reduce their Tail-1% latency by 10-20%; 24 endpoints can reduce their Tail-1% latency by 20-30%; cumulatively, 26 endpoints can reduce their Tail-1% latency by $> 50\%$. Similarly, 60 endpoints can reduce their Tail-50% latency by 10-20% and 25 endpoints can reduce their Tail-50% latency by 20-30%. Combined, 23 endpoints can reduce their Tail-50% latency by $> 50\%$.

We ran similar analyses on interior APIs that can be reached from different gateway endpoints. We noticed that multiple gateway endpoints exercised ~6K interior APIs. Figure 15 shows the mean latency reduction potential of these internal APIs along with the error bars (orange colors) that represent the minimum and maximum observed latency reductions. $1,503$ of these APIs show a non-zero reduction, and the remaining ones are not shown on the graph. Some APIs (top ~ 500) show high savings with low variance, which means that they error out almost every time. The next 500 shows a high variance, which means that they may not be erroring out every time. However, those interior APIs with a higher mean saving potential deserve further investigation.

Observation 8. The LR estimator indicates a significant latency reduction potential in many endpoints and internal APIs.

We were intrigued to find some endpoints with more than a reduction potential 80% at Tail-50%. These tend to have smaller trace footprints and non-fatal errors occurring just below the root level, often designed into the system and not easily eliminateable. This may lead to overestimations in projected savings, a topic we will delve into in Section 6 with specific case studies.

We sought to determine whether non-fatal errors have a greater impact on tail latency compared to overall latency. Figure 16 illustrates the distribution of endpoint latency reductions across various tail percentiles, focusing only on endpoints where the reduction in Tail-1% is at least 10%. The plotted data points and trend lines suggest that the potential for latency reduction is generally higher for Tail-1% and Tail-5% than for Tail-10% and beyond.

The trend lines for higher tail percentiles exhibit lower R^2 values, indicating less perfect fits, which is expected due to the presence of many outliers. This variability suggests that, for many endpoints, tail latency is significantly more affected by errors compared to general latency conditions. In cases where Tail-1% and Tail-50% show similar reduction potentials, it implies that the endpoint consistently experiences errors. In contrast, if the tail-1% reduction is less than in tail-50%, other factors, in addition to errors, are likely to contribute to prolonged tail latencies.

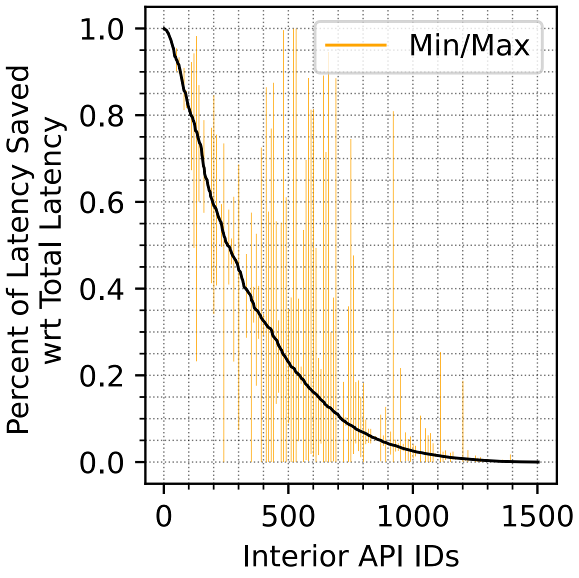

Fig. 15. Latency-saving potential of interior (non-gateway) APIs.

Observation 9. Estimates from LR estimator confirm that non-fatal errors have a greater impact on tail latency than on overall latency for many endpoints, indicating that errors are more prevalent at the tail of these distributions.

4.4 Discussion: The Benefits and Limitations of LR Estimator

The utility of the LR estimator becomes clear when comparing error counts against their actual impact on latency. Eliminating the most common errors does not necessarily yield significant performance improvements, as many of these errors do not occur in the critical path of a request. Our analysis (Figure 12) revealed that about 50% of errors occur off the critical path and therefore have a minimal impact on latency, while the remaining 50% directly contribute to prolonged response times. Even identifying errors on the critical path is not enough, as eliminating a single error may cause the path to shift, with new bottlenecks emerging. The LR estimator addresses this complexity by evaluating potential changes in the critical path when errors are removed, allowing it to project realistic latency savings and identify which optimizations will deliver the highest gains.

In large systems with thousands of endpoints, prioritizing the right areas for optimization is critical to ensure that investments in engineering time and resources are justified. The LR estimator enables this by focusing not on the most frequent errors, but on the ones that have the greatest potential to improve overall system performance. This approach allows organizations to concentrate their efforts on the endpoints where fixes will lead to the most significant latency reductions, making it a powerful tool to drive meaningful improvements in large-scale microservice architectures.

The LR estimator offers best-effort estimates but comes with certain limitations. This method optimistically assumes that all erroring spans could be eliminated, which is often not feasible in practice. Conversely, it adopts a conservative approach by making cautious assumptions about the dependencies among spans based on observed timings in a trace. Additionally, these dependencies are extrapolated from Jaeger traces, which may contain inaccuracies or missing information.

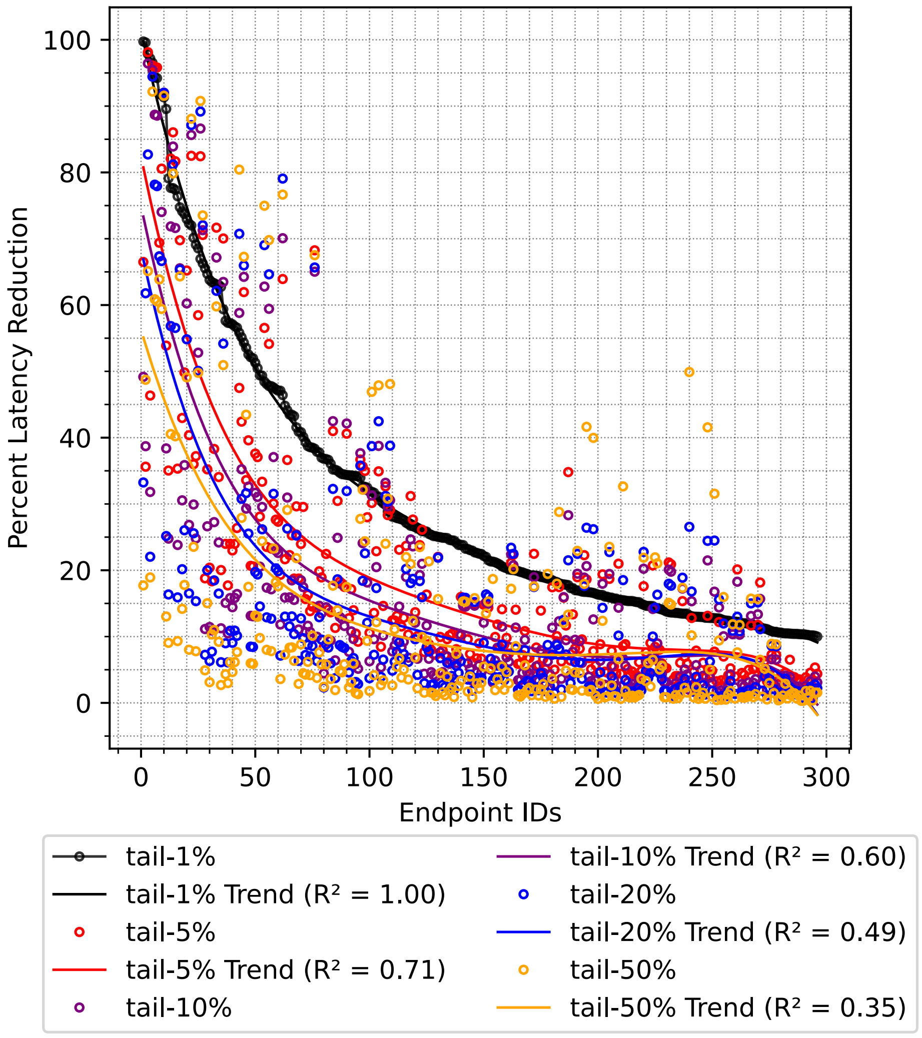

Fig. 16. Potential reduction in endpoint latency. Only endpoints whose Tail-1% reduction is at least 10% are plotted.

Given the absence of a definitive ground truth, it is impractical to individually evaluate all non-fatal errors and their dependencies throughout the system to determine which ones can be eliminated. Therefore, the LR estimator adopts a pragmatic approach, providing estimates of potential savings while respecting certain causal dependencies.

5 Design of the LR Estimator

This section provides the details of the LR estimator, invariant that it maintains and the algorithm it uses when computing the hypothetical (error-eliminated) latency for a given trace. The LR estimator must be fast and scalable so that it can process millions of traces each day to provide latency saving projects for thousands of endpoints. Although there are a few prior statistical methods proposed for latency projections such as [20, 54], we found them to be unsuitable for our needs because they all have one or more of the following shortcomings. Prior methods a) require hours to process a single endpoint, b) require repeated model tuning, and/or c) cannot reason about latency projection in the event of multiple non-fatal errors in a single trace.

5.1 Invariants Maintained

The Latency-reduction estimator *relaxes* the start/end time of a span to begin/end sooner, while *ensuring* that: (I) the spans in series in the original execution must remain in series in the induced error-eliminated execution, and (II) any "work" done in a parent span must remain intact and not be impacted by its children that errored out.

We describe the invariants more formally below, with the following terminology. We represent a span S's start and end times as S_{start} and S_{end}, respectively; we represent a span S seen in its original trace as $S^{observed}$ and its hypothetical error-eliminated executions as S^{hypo}. For two spans X and Y that are children of the same parent span P, we define the relationship X ***precedes*** Y, denoted as $X \prec Y$, iff $X_{end} \leq Y_{start}$. Furthermore, we say that X is an ***immediate predecessor*** of Y, denoted as $X \ll Y$, iff there exists no other child Z of P, s.t., $Z \prec Y$ and $X_{end} < Z_{end}$. We use notation $P \nprec Q$ to mean that P does not precede Q. Two spans P and Q are in parallel, denoted by $P \parallel Q$, iff, $P \nprec Q$ and $Q \nprec P$.

Considering a parent span P and its child C, C can finish its execution sooner for one or more of the following reasons: (I) it had an error (reduces to zero), (II) one of the descendants of C had an error, or (III) one of the siblings C finished early.

Property 1 (Error elimination)**.** If a span C returns an error in the observed execution, its duration will be zero in the error-eliminated execution.

$$\forall C : C^{observed}\text{errors} \Rightarrow C^{hypo}_{start} = C^{hypo}_{end} \tag{1}$$

Property 2 (Parent work retention)**.** Let P be the parent span, and let C and D be the children of P. The minimum work that P needs to perform before a child span can start must be maintained:

$$\left(nil \ll C^{observed}\right) \wedge \left(\Delta = C^{observed}_{start} - P^{observed}_{start}\right) \Rightarrow \quad C^{hypo}_{start} = P^{hypo}_{start} + \Delta \tag{2a}$$

$$\left(C^{observed} \ll D^{observed}\right) \wedge \left(Y^{hypo} \ll D^{hypo}\right) \wedge \left(\Delta = D^{observed}_{start} - C^{observed}_{end}\right) \Rightarrow \quad D^{hypo}_{start} \geq Y^{hypo}_{end} + \Delta \tag{2b}$$

For a child span C that $nil \ll C^{observed}$, i.e., C has no immediate predecessor in P (Equation 2a), the time gap between the start of P and the start of C, $\Delta = C^{observed}_{start} - P^{observed}_{start}$ is the minimum work done in the parent P to start C. This Δ work must remain in the parent before C can start even in the hypothetical error-eliminated execution.

On the other hand, for two spans such that $C^{observed} \ll D^{observed}$, i.e., C is D's immediate predecessor in the original execution (Equation 2b), the time gap between the end of C and the start of D, $\Delta = D^{observed}_{start} - C^{observed}_{end}$, is the minimum work done in the parent P to start D. If a new span, say Y, is chosen as the new immediate predecessor of D in the error-eliminated execution as a result of C changing its end time, D cannot start any sooner than Δ after Y's end.

Property 3 (Time shift)**.** Let C be D's immediate predecessor, i.e., $C \ll D$. Finishing C earlier by an amount δ allows D to begin its execution earlier by up to δ. D's end time shifts accordingly and possibly more if there is a latency reduction underneath D.

$$\left(C_{end}^{hypo} = (C_{end}^{observed} - \delta)\right) \wedge \left(C^{observed} \ll D^{observed}\right) \iff D_{start}^{hypo} \geq (D_{start}^{observed} - \delta) \tag{3}$$

Note that Property 3 is an if-and-only-if condition. That is, a span C shifting its end time earlier impacts the span that immediately follows it (and transitively anything that follows), but it does not impact anything that precedes or in parallel with C.

Property 4 (Series relationship)**.** When $C \prec D$ and D's start and/or end times shift to an earlier time because of a reduction in C's duration, D cannot start sooner than the end of another span X such that $X \prec D$.

$$\forall X : (X^{observed} \prec D^{observed}) \Rightarrow (X^{hypo} \prec D^{hypo}) \tag{4}$$

Referring back to Figure 14, span C erroring means that spans D and E (and transitively span F) shift by C's duration only and not any earlier. I erroring impacts only J. Since spans D, E, and F are shifted by at most C's duration, the work in its parent B, that is $C_{start} - B_{start}$ and $D_{start} - C_{end}$, remain intact. For J, $I \ll J$, and its minimum distance is $\Delta = J_{start} - I_{end}$ in the original execution; hence, when J shifts early in the error-eliminated execution, it maintains the Δ distance between itself and H. Doing so ensures that H and J do not overlap. Finally, since $L \parallel M$, L erroring does not impact M; as a consequence, work done in its parent $M_{start} - K_{start}$ remains intact.

Validity of the Assumed Invariants. These invariants tie back to the assumptions that we stated in Section 4.2 — a) all RPCs errored in the original execution have *zero* durations in the error-eliminated executions, and b) all "observed dependences" among RPCs are preserved, including minimal work done in the parent. Now we discuss why we chose to enforce these invariants. It may seem that the first assumption (i.e., zeroing out all errored RPCs) is not always applicable. Let us tie the erroring RPCs back to the category shown in Table 1. We actually find that many of the `entity not found` errors stem from suboptimal coding idioms that can potentially be optimized if the developer is made aware of them. For other types of errors that are not always possible to eliminate, the LR estimator still serves as a useful tool to point the developer to the right APIs to investigate for potential performance bottlenecks.

5.2 The Algorithm to Compute Latency Reduction

At a high level, the algorithm (shown in Listing 3) works as follows. `SpanTimeReduction` computes the latency reduction for the input span, which is boiled down to computing the maximum amount of reductions and shift of the spans allowable in the error-eliminated execution while maintaining the properties discussed in Section 5.1. It begins from the root of the trace and recursively applies the algorithm to its children. If the span errors (base case-I Line 25), the time reduction is its entire duration (Property 1). If the span has no children, there is no reduction (base case-II Line 27). Otherwise, the function sorts all children spans in descending order of their end times (Line 29) and finds all candidates (`lrcCandidates`) that can potentially become the last-returning child (LRC) and hence on the critical path in the error-eliminated execution; they are either the LRC seen in the observed execution (Line 33) or any other span in parallel with it (Line 36).

For each `lrcCandidate`, it computes its end time in the error-eliminated execution using `HypoSpanEndTime` (Line 40). The candidate with the maximum ending time will be the new LRC (Line 41). The time reduction of `span` is then the gap between the end times of the original LRC and the newly found LRC (Line 42).

```
 1 def HypoSpanEndTime(span, cache):
 2   if span in cache: return cache[span]
 3   # Get reduction in span and its descendants.
 4   reduction = SpanTimeReduction(span)
 5   newDuration = span.duration - reduction
 6   # Get the min distance with the immediate predecessor in the observed execution.
 7   pred = immediatePred(span)
 8   if (pred is not nil): minGap = span.start - pred.end
 9   else: minGap = span.start - span.parent.start
10   # Get candidates for span's immediate predecessor (IP) in the hypothetical
11   # execution, which includes span's IP in the original execution and all other
12   # spans that overlap with IP and end before IP in the original execution.
13   ipCandidates = GetImmediatePredCandidates(span)
14   # Compute immediate predecessor's endtime in the hypothetical execution.
15   predEnd = span.parent.start
16   for candidate in ipCandidates:
17     candidateEnd = HypoSpanEndTime(candidate, cache)
18     predEnd = max(predEnd, candidateEnd)
19   newEndTime = predEnd + minGap + newDuration
20   cache[span] = newEndTime
21   return newEndTime
22
23 def SpanTimeReduction(span):
24   # Base case: the span returns error.
25   if (span.hasError): return span.duration
26   # span did not error and has no children => zero reduction.
27   if len(span.children) == 0: return 0
28
29   sortedChildren = sortDescendingByEndTimes(span.children)
30   lrc = sortedChildren[0]  # lrc: last-returning child
31   # lrcCandidates includes LRC in observed execution and all other spans that
32   # overlap / in parallel with LRC in the observed execution.
33   lrcCandidates = [lrc]
34   for s in sortedSpans[1:]:
35     if happensBefore(s, lrc): break
36     lrcCandidates.append(s)
37   cache = {}
38   newEndTime = span.start
39   for s in lrcCandidates:
40     sEndTime = HypoSpanEndTime(s, cache)
41     newEndTime = max(newEndTime, sEndTime)
42   return lrc.end - newEndTime
43
44 timeSaved = SpanTimeReduction(rootNode)
```

Listing 3. Algorithm to compute the latency reduction for the error-eliminated execution from an observed execution.

`HypoSpanEndTime` computes the hypothetical end time of the given span S by accounting for the following two changes: (I) the latency reduction underneath S as a result of some descendant erroring, and (II) time shift of the predecessors of S, which makes S start and end sooner. It performs these steps:

(1) calls `SpanTimeReduction` on S, which returns the latency reduction underneath itself (Line 4).
(2) finds `minGap` as the minimum work to maintain in S's parent before S can start — the `minGap` is either the distance between the end of S's immediate predecessor and S_{start} in the original execution (Line 8) or between the start of S's parent and S_{start} in the original execution (Line 9) if S does not have an immediate predecessor (Property 2).

(3) computes `ipCandidates` of S by identifying its immediate predecessor I in the original execution and all spans that overlap with and end earlier than I (Line 13). These are the only potential candidates that could be the immediate predecessor of S in the error-eliminated execution (Property 4). It invokes `HypoSpanEndTime` recursively for each candidate, which reports end times in the error-eliminated execution; the candidate with the latest end time (`predEnd`) is chosen (Line 18).
(4) finally, computes S's end time in the error-eliminated execution as the summation of `predEnd`, `minGap`, and S's new duration (Line 19), per Properties 3-2.

`HypoSpanEndTime` caches the new end time computed for a span. Since a span can be an immediate predecessor to more than one span, the memoization serves as an optimization without impacting the correctness.

Correctness. For readability, we include a proof sketch here to provide intuition and include the full detail of the proof in Appendix A. We first define the term ***level***: given a trace, if its root span has no children, we say that the root is at level one; if the root span has children, then the root has level one plus the maximum level among its children. We use the term level similarly for trace — the trace has the same level number as its root span.

The `HypoSpanEndTime` and `SpanTimeReduction` are mutually recursive, and `HypoSpanEndTime` recursively calls itself. We first present a lemma showing that `HypoSpanEndTime` returns the correct hypothetical end time for a span with level n assuming that `SpanTimeReduction` computes the correct time reduction for a span with level n or less. We show this by inducting on the "rank" of a span among its siblings (i.e., spans that share the same parent). A span without any predecessor has rank 0, and a span with predecessor(s) has rank one plus the maximum ranks among its predecessor(s).

For the base case, `HypoSpanEndTime` returns the correct answers for the rank 0 spans, as there is no shift and the reduction boils down to `SpanTimeReduction` returning the correct time reduction, which satisfies the properties. For the inductive case, assuming `HypoSpanEndTime` returns the correct result for a span at rank $r - 1$ or less, we argue that it also returns the correct result for a span at rank r. Assume a rank-r span X with an immediate predecessor Y in the error-eliminated execution. The inductive argument makes the following two key points. First, the reduction time for X is computed correctly, as X has level n and we assume (for this lemma) that `SpanTimeReduction` computes the correct time reduction for a span with level n or less. Second, X correctly identifies Y (i.e., the span with end time `predEnd` in Lines 16-18). This is true because (I) Y must be in the set returned by `GetImmediatePredCandidates` in Line 13, and (II) every span in this set must have rank less than r so `HypoSpanEndTime` returns the correct result. The end time for X in the error-eliminated execution is fully captured by X's time reduction and the shift allowable for Y, and thus `HypoSpanEndTime` returns the correct result.

Finally, to show that the LR estimator computes the correct time reduction, we show that `SpanTimeReduction` returns the correct results by inducting on the level of the input trace. Here, the base case is a trace with one level (i.e., root span with no children). In this case, `SpanTimeReduction` simply returns the entire duration of the input span if it errors out (Line 25) or zero otherwise (Line 27), which is correct (Property 1). For the inductive case, since we know `SpanTimeReducdtion` returns the correct time reduction for spans at level $n - 1$ (by inductive hypothesis), we know that `HypoSpanEndTime` returns the correct answer for spans at level $n - 1$ as well (per lemma above). Then, the correctness of `SpanTimeReduction` for a span with level n boils down to showing that the code correctly identifies the hypothetical last-child returning, which it does. Thus, it follows that `SpanTimeReduction` returns the correct time reduction for a root span with level n.

Algorithmic Complexity. For a span with n children, the algorithm performs $O(n^2)$ work, leading to the asymptotic complexity of $O(N^2)$ for a trace with N spans. Typically, the number of children under a span is small. In practice, it runs fast — running the algorithm for a large trace of 10K spans took only 1.5ms, much faster than the time it takes to download the trace.

6 Case Studies

How should we evaluate the LR estimator? We have a correctness argument and we know that the implementation works correctly based on many small synthetic test traces. However, estimates from LR estimator are optimistic and best-effort projections, and not all may be achievable. Ultimately, what we want to know is whether LR estimator works "in the field" — does it indeed help developers identify inefficiencies and optimization opportunities in microservice architecture? This section details how we use LR estimator to prioritize endpoints for investigation and discuss our findings.

6.1 Selecting Endpoints to Investigate

The primary function of the LR estimator is to identify endpoints that warrant further investigation by providing an "upper bound" on potential savings based on the assumption that all error-prone spans could be eliminated. Endpoints with significant potential savings often contain numerous errors or long RPCs that frequently fail. Investigating these endpoints remains worthwhile even if not all errors can be resolved, as they are identified by the estimator as candidates for substantial improvements.

We systematically applied the LR estimator to sampled traces across endpoints/APIs, which calculated potential savings for requests in the Tail-x% for each endpoint, where x ranges from 1 to 100. We selected endpoints/APIs for further investigation based on high potential savings and a significant number of sampled traces collected over seven days. This approach led us to prioritize several endpoints for deeper analysis, a process involving collaboration with service owners, and an extensive review of the massive microservice codebase. This ongoing investigation has already yielded three notable case studies.

6.2 Visualizing Errors

We apply LR estimator to aggregate trace samples for an endpoint, producing data to visualize non-fatal errors for the investigation. The tooling maintains the call path starting from the gateway endpoint leading to every RPC that errors and is on the critical path (CP). Then it merges all such call paths belonging to the same endpoint generated from many traces into a compact calling context tree, as described in [4]. The script generates a call graph for each endpoint by aggregating its traces.

Flame graphs [22] have emerged as a standard technique for visualizing call path profiles [5]. We use flame graphs to show an aggregated calling context tree [5] of errors on the critical path. The width of the nodes is determined by the time contribution of the error to the total CP. Additionally, we display the error count for each node in the flame graph.

6.3 App-Launch Use Case

The Uber company has a substantial mobile user base and is heavily relying on certain workflows critical to its business. One such workflow is the mobile app launch, demanding low latency due to its importance and high usage volume. When initiated, this process sends a request to the `app-launch` gateway endpoint, triggering numerous RPCs for state machine transitions, database queries, and other APIs.

The LR estimator analysis identified the `app-launch` endpoint as a major contributor to latency, suggesting a potential reduction of 27. 8% in P99 latency. Generated flamegraphs revealed that two RPCs, `pool-provider` and `internal-provider`, consistently failed in tail requests. However, the `app-launch` still succeeded. The typical RPC chain involved sequences like `app-launch` →

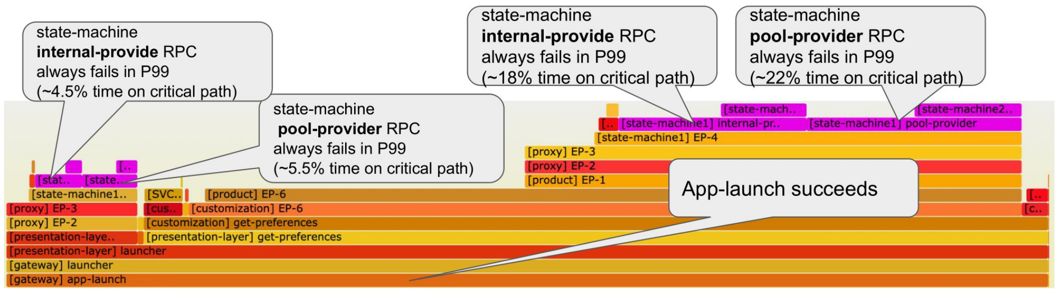

Fig. 17. pool-provider and internal-provider RPCs fail in the P99 profiles on two different call paths but app-launch succeeds. Wasteful pool-provider call consumes 27.8% of the critical path.

... → `state-machine` → `pool-provider` and `app-launch` → ... → `state-machine` → `internal-provider` (Figure 17). In contrast (not shown), in the P50 cases, there were no calls to `pool-provider` and the `internal-provider` succeeded. The `state-machine` has a very high query rate from many other callers and the errors it was seeing from the downstream `pool-provider` did not cause it to cross its error alert limits.

Upon further investigation by the state-machine service experts, we discovered that while the majority of requests were handled by the `internal-provider`, about 3% were handled by an `external-provider`. Previously, the system had a "pooling" feature, where the sequences of providers were tried: `internal-provider`, `pool-provider`, and `external-provider` in that order. Although this feature was discontinued, the `pool-provider` attempting code remained in the state-machine. Consequently, when the `internal-provider` failed, the system unnecessarily attempted the `pool-provider`, which invariably failed, before successfully proceeding to the `external-provider`.

The `pool-provider` RPC was wasteful since it would never succeed. Recognizing this inefficiency, the team promptly eliminated the redundant `pool-provider` RPC from the process. This optimization reduced the `app-launch` endpoint's tail latency by approximately 30%, closely aligning with LR estimator's projections. Additionally, since the state machine was integral to other endpoints, this change yielded up to a 10% latency reduction in several related endpoints.

6.4 User Customization Failure Use Case

The LR estimator highlighted `get-stores-view` as a top endpoint with a significant savings potential, which shows a list of stores to a querying user and also has a high traffic volume. The latency of this endpoint is business-critical.

The flame graphs (see Figure 18) indicated that it had numerous calls to the `get-user` endpoint, which failed without failing `get-stores-view`. `get-user` was called from 110 different RPC call paths. The total time on the critical path for `get-user` contributed to 57% of the P99 latency of `get-stores-view`. `get-user` is an API that provides various data about a user registered with the Uber company. However, it returns an error if the user being queried does not exist in the system. Many endpoints, including `get-stores-view`, allow users to query whether or not they are logged in/registered with Uber.

Many microservices call `get-user` frequently to customize their results based on user information. If the user is not logged in, generic, uncustomized results are returned. The failure of `get-user` in a query when called from one caller, say C_1 is not memoized as part of the request R when another caller, say C_2, invokes `get-user` again as part of R. These redundant invocations, happen from

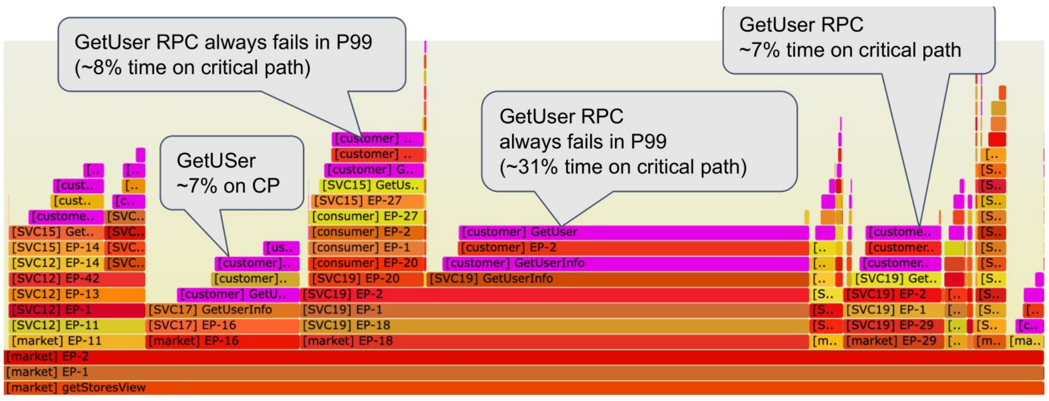

Fig. 18. get-stores-view encounters numerous non-fatal errors on various call paths for get-user in the P99. Failing get-user calls consumes ∼57% of the critical path.

various loosely coupled microservers. In addition to get-stores-view, several other endpoints are impacted by this issue.

A solution here is to perform request-level caching of error-returning calls throughout their RPC journey so that repeated attempts to make the same call with the same payload can simply replay the error eliding the actual call. This solution to this problem is in the conceptual stage due to its broad implications across our ecosystem and changes to the underlying RPC layer.

6.5 Parallel Fetch from Separate Data Stores

The next case is an internal API, called fetchInfo, which issues two independent data querying APIs concurrently (one to a mobile data store for mobile and one to a web data store). However, the invariant is that the data being queried can only be present in at most one data store, not both; hence, one of the APIs will always return a InfoNotFound error.

The concurrency design choice is meant to keep the latency low, but is heavy and incurs 50% wasteful work. The fact is that the calls ended up being scheduled and executed sequentially most of the time per Jaeger trace output, and this is how LR estimator was able to pinpoint the issue. A possible optimization is to embed additional context information in the incoming request, such as the type of the request (mobile vs. web) so that only the relevant downstream data store is called to avoid wasteful work. However, applying this optimization incurs significant engineering efforts due to many upstream callers. This example highlights both the overhead of microservice errors and the difficulty in redesigning for optimization, despite being obvious.

6.6 User Experience of the LR Estimator

Based on our experience thus far, we believe that LR estimator provides a good heuristic to bubble up endpoints worthy of investigations, both for debugging design choices and for latency optimization, as illustrated by our case studies. As discussed earlier, LR estimator can overproject savings. We investigated a few endpoints whose reported savings were not realizable. The false positive of the projected gain comes mainly from two scenarios: 1) inaccuracy of the trace (e.g., inaccurate tagging or mislabeling of errors) and 2) the nature of the errors. To give some examples, for the first case (trace inaccuracy), we have seen an instance where LR estimator projected a large saving because the second-level sub-RPC errored out while the top-level request succeeded. It turns out that, in this particular instance, the top-level request actually failed but the Jaeger trace was not correctly tagged, leading the LR estimator to assume that this is a case of a non-fatal error when it is not. For the second case (nature of errors), we have seen an instance where the code has some condition checks to perform early bailout of a complex operation, and the early bailout manifested as an RPC error. In this case, “eliminating” the error (i.e., removing the early bailout logic) is not desirable, as early bailout saves compute resources.

7 Related Work

Characterization. Several previous works have characterized microservices and RPCs on a similiar scale. Varshneya presented the design of microservices architecture at Netflix [48]. Luo et al. [34] study the characteristics of Alibaba traces and build a model to simulate call graphs. Huye et al. [27] present the microservice architecture in Meta, particularly focusing on microservice topology and traces of the request workflow. Zhang et al. [55] compute the critical path of large-scale microservice traces to guide optimization and anomaly detection. Gan et al. [19] present DeathStarBench and explore the implications of microservices in networking, OS, and cluster management. Zhang et al. [54] use a tree structure (L-Tree) to model the latency change at the aggregation level given certain conditions changes. Seemakhupt et al. [42] present a comprehensive characterization of RPCs at Google; specifically, they show common RPC errors and allude to RPC error removal to save resources in large-scale distributed systems for future study. These studies do not quantify non-fatal errors, their effect on request latencies, or project potential latency savings through error elimination.

Anomaly Detection and Root Cause Analysis. Prior art is rife with anomaly detection on microservice traces [25, 31–33, 38, 45, 46, 53, 56, 57]. Seer [20] combines supervised learning with distributed tracing to detect potential microservice performance anomalies. It has a deep neural network with both convolutional layers followed by long- and short-term memory layers. MicroRCA [50] uses unsupervised learning to pinpoint the root cause of performance issues in microservices. It reconstructs the topology graph to model the running microservice-based application, then subtracts a subgraph for anomaly detection. Similarly, algorithms for root cause analysis over execution traces have a long history in parallel and distributed computing. Ostrowski et al.[36] introduce a system to analyze end-to-end latency changes in large distributed applications and identify root causes. Wu et al. [51] diagnosing timing-related issues in distributed systems. Bohme et al. [9] analyze the root cause of wait states in MPI programs by reverse trace reply to detect performance bugs. Unlike these models and methods that do not differentiate non-fatal errors from error-free traces, latency-reduction estimator specifically analyzes and assesses the impact of such errors in unstructured distributed programs. It provides a targeted approach to diagnose and mitigate RPC-induced latency issues.

Performance and Error Troubleshooting. [47] and [17] employ distributed tracking to diagnose system performance problems, including performance degradation and errors. Ates et al. [7] develop an automation (VAIF) to enable additional logging based on observed performance degradation in distributed applications. Huang and Zhu [26] propose a new trace aggregation method that groups similar traces hierarchically to calculate detailed statistics and an automated tool analyzes them to identify performance problems. Mace et al. [35] tackle the difficulty of monitoring and troubleshooting in distributed systems with Pivot Tracing. Curtsinger and Berger introduce COZ [10], a causal profiler that performs performance experiments during runtime to predict potential optimizations. However, COZ, unsuitable for production environments due to its disruptive nature, primarily evaluates latency in single-process programs. In contrast, our method evaluates RPC systems and uses an algorithmic approach with real production data for postmortem analysis. Despite some similarities with these works, detecting the latency reduction potential, insights from a ground-up microservice environment, and the opportunity to save latency from non-fatal errors presented here are unique.

Bug Classification and Software Aging. Bug classification and software aging are critical to understanding performance degradation in modern software systems. Software aging is the gradual decline in performance or reliability of a software system due to resource exhaustion, error accumulation, and environmental changes, potentially causing failures. Grottke and Trivedi [23] distinguish between *Bohrbugs*, which are predictable, and *Mandelbugs*, which are complex and harder to diagnose due to chaotic behaviors. They propose strategies like *rejuvenation*, a preventive measure that aims to reset the states of the system to combat the effects of aging. Similarly, Dohi et al. [13] present methodologies for addressing software aging through rejuvenation, highlighting its role in maintaining long-term reliability in distributed systems. However, while rejuvenation techniques address aging, they do not fully resolve the complex and erratic behaviors observed in microservice architectures such as Uber, where non-fatal errors often result in significant latency issues. Our work focuses on addressing these operational inefficiencies specific to large-scale microservices.

8 Discussion

While our evaluation focuses entirely on request traffic, development culture(s), and program structures and behaviors specific to Uber's microservice ecosystem, the study encompasses a diverse set of services catering to a wide range of work streams. While the qualitative patterns we discovered may apply to other microservice ecosystems, quantitative metrics could differ. For example, Uber's microservice architecture is known for having a particularly wide and deep call graph [27], with extensive inter-service communication. This complexity of the RPC graph (a large number of nodes in the graph and wider and deeper graphs) increases the likelihood of encountering errors by a top-level request on its journey through the system compared to systems with an RPC graph with fewer nodes and a shallower graph depth. Our insights are most relevant to systems that have evolved organically with the microservice architecture, growing alongside the business. In contrast, more legacy monolithic systems, with their smaller RPC graphs, may exhibit different quantitative characteristics for errors and latency.

Additionally, error tolerance rates vary across microservice ecosystems and business domains. A system engineered for a higher RPC success rate (e.g., 99.9999%) will result in the top-level user request experiencing fewer errors in its journey and hence have a lesser impact on its latency compared to one designed for a lower success rate (e.g., 95%). Even within these systems, the size and structure of the RPC graph and the pathways through lower success rate nodes will significantly influence the quantitative impact on performance.

Although we do not have access to the internals of another large microservice system like ours, based on our observations of the Uber's microservice ecosystem, we believe that the following generalizations of the key observations made in Section 3 are plausible in any large microservice system:

(1) For requests that involve many RPCs and traverse multiple unique endpoints, encountering errors along the way is highly probable, even when the top-level request is ultimately successful. This is especially true unless every endpoint maintains an exceptionally low error rate. In a microservice ecosystem with numerous endpoints, it is nearly impossible to ensure that every endpoint has an extremely low error rate. While some endpoints may experience very few non-fatal errors, others may exhibit much higher error rates, warranting further investigation to improve the system's overall health (Observations 1 and 2).

(2) Non-fatal errors will tend to cluster around a few key error types, such as "entity not found" errors. These errors are often tolerable by the callers (Observation 3).
(3) Errors that originate deeper in the RPC graph are likely to be suppressed by higher-level callers, as each layer introduces an additional level of resiliency. In contrast, errors that originate closer to the root of the graph are more likely to be fatal (Observation 4).
(4) In large, well-engineered systems, predictability becomes crucial. Therefore, endpoints tend to exhibit bimodal resiliency behavior (Observation 5).
(5) Successful requests that contribute to extended tail latency often have a disproportionately high number of RPCs with non-fatal errors (Observations 6 and 7).
(6) Top-level requests may encounter the same error multiple times for the same payload and API as they traverse the system, due to a lack of information sharing in decoupled systems.

We leave a comprehensive exploration of these dimensions across multiple microservice ecosystems for future research.

9 Conclusions

In this paper, we analyze microservice systems, focusing on errors that do not result in failing the top-level request (non-fatal errors). Using Uber's 6,000+ microservices for our analysis, we show that non-fatal errors are common across thousands of services. We demonstrated that non-fatal errors can contribute significantly to latency and sometimes services tend to live with errors. We have developed the latency-reduction estimator that analyzes traces to pinpoint and quantify non-fatal errors and estimate how much latency can be reduced by eliminating non-fatal errors. Using the latency-reduction estimator, we characterized the nature of non-fatal errors at Uber's microservice ecosystem. Our experience indicates that non-fatal errors often stem from certain sub-optimal coding idioms arising from developers' inattention to performance. Using the latency-reduction estimator, we identified opportunities for optimization, including reducing the latency of a critical endpoint by 30%.

While non-fatal errors are often tolerated in resilient systems, this research suggests that their cumulative effect on performance and resource consumption is far from negligible. We can enhance overall system performance without requiring large-scale architectural changes by pinpointing and mitigating these errors. This work highlights the need for deeper analysis of non-fatal errors in large-scale microservice systems and offers a structured approach to reducing their impact. We believe that the methods presented in this work are applicable not only to Uber' ecosystem but also to other organizations facing similar challenges in service-oriented architectures. It is a step toward more efficient, scalable, and responsive systems, which underscores the importance of proactive error management to ensure long-term performance and reliability.

Acknowledgements

This work is supported in part by the National Science Foundation under Grants No. 1910568 and 1943456. We thank our shepherd Timothy Zhu and the anonymous reviewers for their valuable feedback.

References

[1] MySQL. https://www.mysql.com/. (Accessed on 07/30/2023).
[2] Status codes and their use in gRPC . https://grpc.github.io/grpc/core/md_doc_statuscodes.html.
[3] yarpc/yarpc-go: A message passing platform for go. https://github.com/yarpc/yarpc-go. (Accessed on 07/30/2023).
[4] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R Tallent. Hpctoolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience*, 22(6):685–701, 2010.

[5] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. PLDI '97, page 85–96, New York, NY, USA, 1997. Association for Computing Machinery.
[6] AppCentrica. Microservices the good bad and the ugly. https://www.appcentrica.com/the-rise-of-microservices/.
[7] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 165–170, 2019.
[8] Jaeger Authors. Jaeger: open source, end-to-end distributed tracing, 2022.
[9] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. Identifying the root causes of wait states in large-scale parallel applications. *ACM Trans. Parallel Comput.*, 3(2), jul 2016.
[10] Charlie Curtsinger and Emery D Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 184–197, 2015.
[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
[12] Anirban Deb, Suman Bhattacharya, Jeremey Gu, T Zhou, Eva Feng, and Mandie Liu. Under the hood of uber's experimentation platform. *Uber engineering.*, 2018.
[13] Tadashi Dohi, Kishor S Trivedi, and Alberto Avritzer. *Handbook of software aging and rejuvenation: fundamentals, methods, applications, and future directions.* World scientific, 2020.
[14] Brian Eaton, Jeff Stewart, Jon Tedesco, and N Cihan Tas. Distributed latency profiling through critical path tracing: Cpt can provide actionable and precise latency analysis. *Queue*, 20(1):40–79, 2022.
[15] Mostafa Elhemali, Niall Gallagher, Bin Tang, Nick Gordon, Hao Huang, Haibo Chen, Joseph Idziorek, Mengtian Wang, Richard Krog, Zongpeng Zhu, et al. Amazon {DynamoDB}: A scalable, predictably performant, and fully managed {NoSQL} database service. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1037–1048, 2022.
[16] Paul Durzan Ensono. The growth of microservices and why it matters to your business. https://www.datacenterdynamics.com/en/opinions/growth-microservices-and-why-it-matters-your-business/, Feb 2020.
[17] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):1–35, 2012.
[18] Roy Thomas Fielding and Richard N. Taylor. *Architectural Styles and the Design of Network-Based Software Architectures.* PhD thesis, 2000. AAI9980887.
[19] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.
[20] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems*, pages 19–33, 2019.
[21] Adam Gluck. Introducing domain-oriented microservice architecture. *Uber Engineering Blog*, 2020.
[22] Brendan Gregg. Blazing performance with flame graphs. In *27th Large Installation System Administration Conference.* USENIX Association, 2013.
[23] Michael Grottke and Kishor S Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109, 2007.
[24] A gRPC. high performance, open-source universal rpc framework, 2018.
[25] Xiaofeng Guo, Xin Peng, Hanzhang Wang, Wanxue Li, Huai Jiang, Dan Ding, Tao Xie, and Liangfei Su. Graph-based trace analysis for microservice architecture understanding and problem diagnosis. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1387–1397, 2020.
[26] Lexiang Huang and Timothy Zhu. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 76–91, 2021.
[27] Darby Huye, Yuri Shkuro, and Raja R Sambasivan. Lifting the veil on {Meta's} microservice architecture: Analyses of topology and request workflows. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 419–432, 2023.
[28] infoq. Controlled Chaos: Taming Organic, Federated Growth of Microservices. https://www.infoq.com/presentations/microservices-federated-growth/.
[29] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA, 2017. Association for Computing Machinery.
[30] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS operating systems review*, 44(2):35–40, 2010.

[31] Zeyan Li, Junjie Chen, Rui Jiao, Nengwen Zhao, Zhijun Wang, Shuwei Zhang, Yanjun Wu, Long Jiang, Leiqin Yan, Zikai Wang, et al. Practical root cause localization for microservice systems via trace analysis. In *2021 IEEE/ACM 29th International Symposium on Quality of Service (IWQOS)*, pages 1–10. IEEE, 2021.

[32] Dewei Liu, Chuan He, Xin Peng, Fan Lin, Chenxi Zhang, Shengfang Gong, Ziang Li, Jiayu Ou, and Zheshun Wu. Microhecl: High-efficient root cause localization in large-scale microservice systems. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 338–347. IEEE, 2021.

[33] Ping Liu, Haowen Xu, Qianyu Ouyang, Rui Jiao, Zhekang Chen, Shenglin Zhang, Jiahai Yang, Linlin Mo, Jice Zeng, Wenman Xue, et al. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*, pages 48–58. IEEE, 2020.

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

[35] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 35(4):1–28, 2018.

[36] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. Diagnosing latency in multi-tier black-box services. 2011.

[37] Gaspard Plantrou. Microservices are becoming the default application architecture choice – is it time to jump in? https://www.scaleway.com/en/blog/microservices-are-becoming-the-default-application-architecture-choice-is-it-time-to-jump-in/, September 2022.

[38] Haoran Qiu, Subho S Banerjee, Saurabh Jha, Zbigniew T Kalbarczyk, and Ravishankar K Iyer. {FIRM}: An intelligent fine-grained resource management framework for {SLO-Oriented} microservices. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 805–825, 2020.

[39] Murali Krishna Ramanathan, Lazaro Clapp, Rajkishore Barik, and Manu Sridharan. Piranha: Reducing feature flag debt at uber. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, pages 221–230, 2020.

[40] Chris Richardson. Service-oriented architecture. https://microservices.io/.

[41] Deepu Sasidhara. Microservices the good bad and the ugly. https://deepu.tech/microservices-the-good-bad-and-the-ugly/, October 2019.

[42] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

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

[44] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook white paper*, 5(8):127, 2007.

[45] Jacopo Soldani and Antonio Brogi. Anomaly detection and failure root cause analysis in (micro) service-based cloud applications: A survey. *ACM Computing Surveys (CSUR)*, 55(3):1–39, 2022.

[46] Gagan Somashekar, Anurag Dutt, Mainak Adak, Tania Lorido Botran, and Anshul Gandhi. Gamma: Graph neural network-based multi-bottleneck localization for microservices applications. 2024.

[47] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K Coskun, and Raja R Sambasivan. Automating instrumentation choices for performance problems in distributed applications with vaif. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 61–75, 2021.

[48] Ketan Varshneya. Understanding design of microservices architecture at netflix, 2021.

[49] Yingying Wen, Guanjie Cheng, Shuiguang Deng, and Jianwei Yin. Characterizing and synthesizing the workflow structure of microservices in bytedance cloud. *Journal of Software: Evolution and Process*, 34(8):e2467, 2022.

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

[51] Yang Wu, Ang Chen, and Linh Thi Xuan Phan. Zeno: Diagnosing performance problems with temporal provenance. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 395–420, 2019.

[52] C. Yang and B. Miller. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*, Los Alamitos, CA, USA, jun 1988. IEEE Computer Society.

[53] Chenxi Zhang, Zhen Dong, Xin Peng, Bicheng Zhang, and Miao Chen. Trace-based multi-dimensional root cause localization of performance issues in microservice systems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pages 894–894. IEEE Computer Society, 2024.

[54] Yazhuo Zhang, Rebecca Isaacs, Yao Yue, Juncheng Yang, Lei Zhang, and Ymir Vigfusson. Latenseer: Causal modeling of end-to-end latency distributions by harnessing distributed tracing. In *Proceedings of the 2023 ACM Symposium on Cloud Computing*, pages 502–519, 2023.

[55] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. CRISP: Critical path analysis of Large-Scale microservice architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 655–672, Carlsbad, CA, July 2022. USENIX Association.
[56] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Wenhai Li, and Dan Ding. Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study. *IEEE Transactions on Software Engineering*, 47(2):243–260, 2018.
[57] Xiang Zhou, Xin Peng, Tao Xie, Jun Sun, Chao Ji, Dewei Liu, Qilin Xiang, and Chuan He. 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*, pages 683–694, 2019.

A Correctness of the Latency-Reduction Estimator

This appendix provides the full correctness proof of the LR estimator. For readability, we include the symbols, algorithm, and its invariants again here.

For two spans X and Y that are children of the same parent span P, we define the relationship X ***precedes*** Y, denoted as $X \prec Y$, if and only if $X_{end} \leq Y_{start}$. Furthermore, we say that X is an ***immediate predecessor*** of Y, denoted as $X \ll Y$, iff there exists no other child Z of P, s.t., $Z \prec Y$ and $X_{end} < Z_{end}$. We use notation $P \nprec Q$ to mean that P does not precede Q. Two spans P and Q are in parallel, denoted by $P \parallel Q$, iff, $P \nprec Q$ and $Q \nprec P$.

Property 1 (Error elimination)**.** If a span C returns an error in the observed execution, its duration will be zero in the error-eliminated execution.

$$\forall C : C^{observed}\text{errors} \Rightarrow C^{hypo}_{start} = C^{hypo}_{end} \tag{1}$$

Property 2 (Parent work retention)**.** Let P be the parent span, and let C and D be the children of P. The minimum work that P needs to perform before a child span can start must be maintained:

$$\left(nil \ll C^{observed}\right) \wedge \left(\Delta = C^{observed}_{start} - P^{observed}_{start}\right) \Rightarrow \quad C^{hypo}_{start} = P^{hypo}_{start} + \Delta \tag{2a}$$

$$\begin{aligned}\left(C^{observed} \ll D^{observed}\right) \wedge \left(Y^{hypo} \ll D^{hypo}\right) \wedge \left(\Delta = D^{observed}_{start} - C^{observed}_{end}\right) \\ \Rightarrow \quad D^{hypo}_{start} \geq Y^{hypo}_{end} + \Delta\end{aligned} \tag{2b}$$

For a child span C that $nil \ll C^{observed}$, i.e., C has no immediate predecessor in P (Equation 2a), the time gap between the start of P and the start of C, $\Delta = C^{observed}_{start} - P^{observed}_{start}$ is the minimum work done in the parent P to start C. This Δ work must remain in the parent before C can start even in the hypothetical error-eliminated execution.

On the other hand, for two spans such that $C^{observed} \ll D^{observed}$, i.e., C is D's immediate predecessor in the original execution (Equation 2b), the time gap between the end of C and the start of D, $\Delta = D^{observed}_{start} - C^{observed}_{end}$, is the minimum work done in the parent P to start D. If a new span, say Y, is chosen as the new immediate predecessor of D in the error-eliminated execution as a result of C changing its end time, D cannot start any sooner than Δ after Y's end.

Property 3 (Time shift)**.** Let C be D's immediate predecessor, i.e., $C \ll D$. Finishing C earlier by an amount δ allows D to begin its execution earlier by up to δ. D's end time shifts accordingly and possibly more if there is a latency reduction underneath D.

```
def HypoSpanEndTime(span, cache):
  if span in cache: return cache[span]
  # Get reduction in span and its descendants.
  reduction = SpanTimeReduction(span)
  newDuration = span.duration - reduction
  # Get the min distance with the immediate predecessor in the observed execution.
  pred = immediatePred(span)
  if (pred is not nil): minGap = span.start - pred.end
  else: minGap = span.start - span.parent.start
  # Get candidates for span's immediate predecessor (IP) in the hypothetical
  # execution, which includes span's IP in the original execution and all other
  # spans that overlap with IP and end before IP in the original execution.
  ipCandidates = GetImmediatePredCandidates(span)
  # Compute immediate predecessor's endtime in the hypothetical execution.
  predEnd = span.parent.start
  for candidate in ipCandidates:
    candidateEnd = HypoSpanEndTime(candidate, cache)
    predEnd = max(predEnd, candidateEnd)
  newEndTime = predEnd + minGap + newDuration
  cache[span] = newEndTime
  return newEndTime

def SpanTimeReduction(span):
  # Base case: the span returns error.
  if (span.hasError): return span.duration
  # span did not error and has no children => zero reduction.
  if len(span.children) == 0: return 0

  sortedChildren = sortDescendingByEndTimes(span.children)
  lrc = sortedChildren[0]  # lrc: last-returning child
  # lrcCandidates includes LRC in observed execution and all other spans that
  # overlap / in parallel with LRC in the observed execution.
  lrcCandidates = [lrc]
  for s in sortedSpans[1:]:
    if happensBefore(s, lrc): break
    lrcCandidates.append(s)
  cache = {}
  newEndTime = span.start
  for s in lrcCandidates:
    sEndTime = HypoSpanEndTime(s, cache)
    newEndTime = max(newEndTime, sEndTime)
  return lrc.end - newEndTime

timeSaved = SpanTimeReduction(rootNode)
```

Listing 4. Algorithm to compute the latency reduction for the error-eliminated execution from an observed execution.

$$\left(C_{end}^{hypo} = (C_{end}^{observed} - \delta)\right) \wedge \left(C^{observed} \ll D^{observed}\right) \iff D_{start}^{hypo} \geq (D_{start}^{observed} - \delta) \tag{3}$$

Note that Property 3 is an if-and-only-if condition. That is, a span C shifting its end time earlier impacts the span that immediately follows it (and transitively anything that follows), but it does not impact anything that precedes or in parallel with C.

Property 4 (Series relationship)**.** When $C \prec D$ and D's start and/or end times shift to an earlier time because of a reduction in C's duration, D cannot start sooner than the end of another span X such that $X \prec D$.

$$\forall X : (X^{observed} \prec D^{observed}) \Rightarrow (X^{hypo} \prec D^{hypo}) \tag{4}$$

To argue that our algorithm computes the latency reduction correctly, we first precisely define what it computes. The algorithm computes the time saved by taking the original top-level RPC latency and subtract from it its hypothetical latency, where the hypothetical latency is derived from the error-eliminated execution. Importantly, the error-eliminated execution is implicitly derived from the original execution that satisfies the properties listed above.

Lemma 1. The LR estimator algorithm maintains the four properties stated above.

Proof. Property 1 is maintained directly by the code, per Line 25.

Property 2 is maintained directly by the code as well, per Lines 7-19. When computing the new end time of a span in the error-eliminated execution, Line 19 accounts for the minimum distance by adding `minGap` to the last immediate predecessor's new end time. The computation correctly maintains Property 2 provided that `ipCandidates` in Line 13 contains the correct candidate set of immediate predecessors in the error-eliminated execution and that `HypoSpanEndTime` works correctly (shown formally later).

Property 3 is maintained by the code as well. Let's suppose $C \ll D$ in the original execution and that C's end time shifted earlier by δ in the error-eliminated execution. The new start time for D is implicitly computed in Line 19 as `predEnd + minGap`. We know `minGap` is computed correctly according to its definition shown in Property 2 (Lines 7-9). There are two cases for `predEnd`. In the first case, C is D's immediate predecessor in both the original and error-eliminated execution. Then, `predEnd` is simply $C^{observed}_{end} - \delta$. Since $D^{observed}_{start}$ is simply $C^{observed}_{end}$, it follows that $D^{hypo}_{start} = D^{observed}_{start} - \delta$. The second case is that some other span X is D's immediate predecessor in the error-eliminated execution. Provided that X is included in the `ipCandidates` set in Line 16 (which we show formally in Lemma 3), then, it must be that $C^{hypo}_{end} < X^{hypo}_{end} < C^{observed}_{end}$ because otherwise X would not have become the new immediate predecessor for D. That means *predEnd* (i.e., X^{hypo}_{end}) is greater than $C^{observed}_{end} - \delta$ and it follows that $D^{hypo}_{start} > D^{observed}_{start} - \delta$. Lastly, note that the time shift of C or X impacts only the hypothetical end times of D, the hypothetical immediate successor (and transitively any of D's subsequent hypothetical successor), and does not impact the hypothetical end time calculations of other spans, establishing the if-and-only-if condition in Property 3.

Property 4 follows from Property 2 but requires some argument to show formally. For the purpose of contradiction, let's also suppose that span X precedes B in the original execution but becomes in parallel with B in the error-eliminated execution due to B shifting earlier because B's immediate predecessor (in the original execution) A erroring out or containing erroring spans underneath. (It cannot be that B precedes X because then shifting B earlier could not possibly make X and B overlap.) In the original execution, it cannot be that A precedes X because otherwise either A would not be an immediate predecessor of B; it cannot be that X precedes A because B can shift at most by A's duration while maintaining its minimum distance Δ by Property 2 and that is not enough shift to cause B to overlap with X. The only possibility remains is that A and X overlap while X precedes B in the original execution.

Given that A and X overlap in the original execution, X is included in the `ipCandidates` in Line 13 and considered when the code identifies the end time of the immediate predecessor in the error-eliminated execution (Lines 16-18). Since the new end time of B is computed as Δ away from the end time of its new immediate predecessor, which can be X or another span with later new end time (Line 19), X and B cannot become overlap in the error-eliminated execution. □

Note that Property 4 only states properties about two spans that are in series but does not say anything about two spans that are in parallel in the original execution. In fact, LR estimator allows two spans in parallel in the original execution to become in series in the inferred error-eliminated execution. This can happen, for instance, when span A precedes span B, and they overlap with another span X. If A errors out, it causes B to shift earlier but not X as A does not precede X. This shift may cause B to then precedes X in the error-eliminated execution, but this is fine. Either the critical path does not go through B in the original or error-eliminated executions and thus does not contribute to the time saving anyway, or the critical path goes through B in the original execution, now it may go through X in the error-eliminated execution instead, and the shift of B still contributes partially to the reduction of their parent RPC's latency. Moreover, note that Property 2 is stated in terms of the immediate predecessor in the original execution; thus, in this scenario, B would not be in the set of immediate predecessors for X.

First, we show that the code computes the correct candidate sets for the last-returning child (LRC) and for the immediate predecessor (IP) for a given span in the error-eliminated execution.

Lemma 2. *The code correctly computes the set of potential last-returning child for* `span` *in the error-eliminated execution in Lines 33-36.*

Proof. Let span A be the last-returning child (LRC) in the original execution. Let span B be the LRC in the error-eliminated execution. To show that the code correctly computes the set, we show that either B is A or B overlaps with A in the original execution. For the purpose of contradiction, suppose that B is in series with A. Then, B must precedes A in the original execution since otherwise A would not be the LRC in the original execution. By Property 4, B must still precede A in the error-eliminated execution, which contradicts with B being the LRC. (Note that even if A errors out, it can still be the LRC with zero duration if the new error end time makes it LRC). □

Lemma 3. *The helper function* `GetImmediatePredCandidates` *in Line 13 correctly computes the set of potential candidates of* `span`*'s immediate predecessor in the error-eliminated execution.*

Proof. Let span A be the immediate predecessor (IP) of span C in the original execution, and let span B be the IP of span C in the error-eliminated execution. To show that the code correctly computes the set of IP candidates for C in the error-eliminated execution, we must show that either A and B are the same span, or B is in parallel with A but has an earlier end time than A in the original execution. We know that B must have an earlier end time than A because otherwise A would not be the IP in the original execution for C. Now for the purpose of contradiction, let's suppose that B is in series with A with an earlier end time (i.e., $B \prec A$) in the original execution. By Property 4, B must still precede A in the error-eliminated execution, which contradicts with B being the IP for C in the error-eliminated execution. □

Before we formally show that the LR estimator computes the correct time reduction, we first define the term ***level***: given a trace, if its root span has no children, we say the root is at level one; if the root span has children, then the root has level one plus the maximum level among its children. We use the term level similarly for trace — the trace has the same level number as its root span. The final theorem proves correctness by inducting on the level of the trace. Before that, we first prove a lemma needed for the base case, that `HypoSpanEndTime` computes the correct hypothetical end times for a trace with a root span of level two, i.e., children of the root span are at level one and have no children themselves.e., level one).

Lemma 4. *Consider an input trace with a root span S of maximum level n.* `HypoSpanEndTime` *returns the correct hypothetical end times for its children spans with level $n-1$ provided that* `SpanTimeReduction` *returns the correct time reduction for spans with level $n-1$ or less.*

Proof. We shall ignore the use of `cache` for memoization as `HypoSpanEndTime` is idempotent and thus its use is simply an optimization.

When we invoke `HypoSpanEndTime` on children of S, `HypoSpanEndTime(S)` invokes `SpanTimeRedution` on the children with level $n-1$ or less. By assumption, `SpanTimeRedution` returns the correct time reductions for these spans.

Then, what we need to show is, with the correct reduction for each span in this trace, `HypoSpanEndTime` correctly returns the new end time for a child span. The `HypoSpanEndTime` is a recursive function that calls itself to compute the new end time for a span's immediate predecessor. We will show its correctness by inducting on the "ranks" of immediate predecessors that we have to traverse through before reaching the start of the block (which translates to the recursive depth of `HypoSpanEndTime`).

More formally, let's call a span with no immediate predecessors as rank 0. Then, a span A's rank is then 1+ the max rank among a A's predecessors. We will show inductively by rank that, the new end time returned by `HypoSpanEndTime` indicates the right amount of shift relative to the start of the root span S.

For a span A at rank 0, A_{end} is entirely determined by the reduction returned by `SpanTimeReduction` (as the initial values of `minGap` from Line 9 and `predEnd` from Line 15 cancel out,[4] implicitly maintaining Properties 2 and 3 relative to the start of the block). Thus, the returned new end time for such a span A is computed based on exactly the shift allowed while maintaining Properties 1-3 (and indirectly Property 4 by Lemma 1).

Now let's consider a span B at rank 1. The new end time is computed based on some span X where $X \ll B$ in the error-eliminated execution, B's `minGap` (the Δ in Property 2), and B's reduction returned by `SpanTimeReduction`. We know that `GetImmediatePredCandidates` in Line 13 contains both the immediate predecessor for B in the original execution, call it Y, and $\forall Z : Z \prec B \wedge Z \parallel Y \wedge Z_{end}^{observed} < Y_{end}^{observed}$. Since the $Y \ll B$ in the observed execution and Y is in `ipCandidates`, we know that `minGap` is computed correctly for B. Since B is at rank 1, any span in the `ipCandidates` set must be rank 0, for which `HypoSpanEndTime` returns the correct new end time, including X (which must be in the set by Lemma 3). Thus, we know that `predEnd` is also computed correctly for B and therefore `HypoSpanEndTime` returns the correct new end time for B.

Finally, by the inductive hypothesis, we assume that `HypoSpanEndTime` returns the correct new end time for any children span with rank less than or equal to $r-1$. We show that `HypoSpanEndTime` returns the correct new end time for a child span A of S with rank r. By Lemma 3, the immediate predecessor candidates for A in Line 39 must contain its immediate predecessor in the error-eliminated execution, and that the said candidate must have rank $r-1$ or less. Thus, by the inductive hypothesis, `predEnd` must be computed correctly, as well as the `minGap` and `newDuration` (since A must have level $n-1$ or less and thus by assumption `SpanTimeReduction` returns the correct time reduction for A). Thus, `HypoSpanEndTime` returns the correct new end time for A. □

Theorem. *The Latency-reduction estimator computes the correct time reduction for a given trace.*

Proof. We shall prove this theorem by inducting on the level of the input trace. Again, we refer to the root trace as S.

For the base case, a trace with one level (i.e., only root span S) simply invokes `SpanTimeReduction` on S and returns the correct time reduction for the trace — it is either S's entire duration if S errors out (Line 25) or zero otherwise (Line 27).

For the inductive case, we show that the LR estimator computes the correct time reduction for its root trace S with n levels, assuming that LR estimator computes the correct time reduction for spans

[4]Here the `ipCandidates` in Line 16 will be an empty set.

with $n - 1$ levels. The time reduction for S is essentially the time difference between the end of its last-returning child (LRC) in the observed execution and the end of the LRC in the error-eliminated execution. Since `SpanTimeReduction` computes the correct time reduction for S's children with level $n - 1$ (by inductive hypothesis), by Lemma 4, we know that `HypoSpanEndTime` computes the end time correctly for these children.

Thus, we just need to show that the new LRC of S in the error-eliminated execution is correctly identified. The code in Lines 33-36 collects a set of spans that include the LRC from the original execution and the spans that are parallel to it. By Lemma 2, we know that the new LRC must be within this set. By Lemma 4, we know that `HypoSpanEndTime` must return the correct new end times for the set of spans, thereby identifying the correct LRC in the error-eliminated execution. Thus, we conclude that `SpanTimeReduction` returns the correct time reduction for S, and the LR estimator computes the correct time reduction for a trace with n levels. □

Received August 2024; revised September 2024; accepted October 2024