LatenSeer: Causal Modeling of End-to-End Latency Distributions by Harnessing Distributed Tracing

Yazhuo Zhang*
Emory University

Rebecca Isaacs†
Amazon Web Services

Yao Yue†
IOP Systems

Juncheng Yang
Carnegie Mellon University

Lei Zhang
Princeton University

Ymir Vigfusson
Emory University & Keystrike

Abstract

End-to-end latency estimation in web applications is crucial for system operators to foresee the effects of potential changes, helping ensure system stability, optimize cost, and improve user experience. However, estimating latency in microservices-based architectures is challenging due to the complex interactions between hundreds or thousands of loosely coupled microservices. Current approaches either track only latency-critical paths or require laborious bespoke instrumentation, which is unrealistic for end-to-end latency estimation in complex systems.

This paper presents LatenSeer, a modeling framework for estimating end-to-end latency distributions in microservice-based web applications. LatenSeer proposes novel data structures to accurately represent causal relationships between services, overcoming the drawbacks of simple dependency representations that fail to capture the complexity of microservices. LatenSeer leverages distributed tracing data to practically and accurately model end-to-end latency at scale. Our evaluation shows that LatenSeer predicts latency within a 5.35% error, outperforming the state-of-the-art that has an error rate of more than 9.5%.

CCS Concepts

• **General and reference** → **Performance**; • **Computing methodologies** → **Modeling methodologies**.

Keywords

microservices, distributed tracing, latency estimation, end-to-end latency

*Work done during the internship at Twitter.
†Work done at Twitter.

SoCC '23, October 30–November 1, 2023, Santa Cruz, CA, USA

ACM ISBN 979-8-4007-0387-4/23/11.
https://doi.org/10.1145/3620678.3624787

ACM Reference Format:
Yazhuo Zhang, Rebecca Isaacs, Yao Yue, Juncheng Yang, Lei Zhang, and Ymir Vigfusson. 2023. LatenSeer: Causal Modeling of End-to-End Latency Distributions by Harnessing Distributed Tracing. In *ACM Symposium on Cloud Computing (SoCC '23), October 30–November 1, 2023, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620678.3624787

1 Introduction

Latency estimation enables operators managing large-scale web applications [12, 33, 42, 49] to anticipate the impacts of potential changes before scaling their services, optimizing costs, adding features or adapting to changes in hardware configuration or cloud computing deployments [33, 44, 47, 83]. For instance, the operators may wish to assess how introducing a novel machine learning pipeline or migrating some subset of services to a different data center would affect customer-perceived latency. Such estimation is crucial to prevent performance regression and to protect the end-user experience: Google, Amazon, and Akamai have all noted significant drops in traffic or revenue following a modest (100+ ms) increase in latency [1, 22, 37, 76].

To be pragmatic, operators of web applications desire a latency estimation framework that meets the following goals.

(G1) Easy deployment. The system integrates with existing data collection systems, eschewing invasive and labor-intensive custom instrumentation.

(G2) Flexible scenarios. The system estimates end-to-end latency given hypothetical latency changes in any of its constituent services, informing advanced decision-making.

(G3) Realistic forecasts. The system should express confidence in its results—sound decisions require sound data and models.

In large-scale systems, however, these goals are challenging to fulfill. At scale, modern applications often comprise hundreds or thousands of loosely coupled microservices [13, 29, 45, 71, 77, 89], where a single user request may touch thousands of instances before generating a response [56, 71, 88]. In microservices architectures, a single service often interacts with many others, forming a complex web of serial or

parallel calls to fulfill a user request. Modifications in one can thus affect others, leading to varied impacts on the end-to-end latency of different requests.

State-of-the-art solutions fall short of meeting all three objectives. Several works [33, 63, 64] require explicit instrumentation that makes them difficult and expensive to deploy in different environments **(G1)**. Other methods, which use purely data-driven techniques, either do not target microservice-based web applications [30, 49, 75, 81], or are restricted to making predictions for a subset of services [16, 88] **(G2)**. Finally, the trend of leveraging deep machine learning models for latency estimation would attempt to draw causal conclusions through black-box methods that fundamentally can derive only associations [58, 59], and are thus set up to fail goal **(G3)** [67].

We present LatenSeer, a framework for estimating the end-to-end latency distribution for large-scale microservices applications, that is explicitly designed to fulfill goals **(G1)**—**(G3)**. As shown in the upper left of Fig. 1, LatenSeer piggybacks on the proliferation of distributed end-to-end tracing in large-scale systems, such as Jaeger [32] or OpenTelemetry [53], rather than demanding custom instrumentation **(G1)**. LatenSeer allows an operator (bottom left) to answer the question: *How will the end-users be impacted by some hypothetical changes to the latencies of the underlying microservices* **(G2)**? LatenSeer accepts hypothetical latency changes to services as inputs, and outputs the changed end-to-end latency distribution. We apply LatenSeer on two use cases of latency estimation: *service placement* (UC1)—reasoning about the end user latency impact of resource provisioning or service migration, and *slack analysis* (UC2)—determining the latency budget available to alter a microservice without impacting the overall response times of requests. We validate LatenSeer via null prediction [1] on distributed traces from two production sources to show that the underlying model is sound, and conduct controlled experiments using a social network microservices-based benchmark to evaluate the accuracy of latency estimation **(G3)**.

Inside LatenSeer, we cast the latency estimation problem as a causal model [2] of latency components through a simple principle: a complex service decomposes into the causal interactions between its constituent microservices. The end-to-end latency distribution for the entire application is thus the combination of each component's latency distributions.

LatenSeer overcomes multiple technical challenges that arise when forecasting latency in large-scale systems. First, distributed traces, which record the requests within distributed applications, often exhibit diverse execution paths.

[1]Null prediction refers to the experiments where LatenSeer derives the predicted latency distribution under conditions of zero injected delay.
[2]In distributed tracing, "causality" usually refers to the order of events or operations as they happened in relation to each other

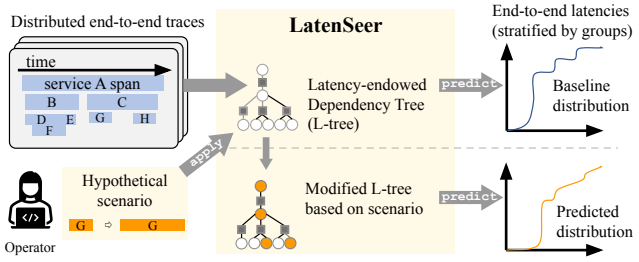


Figure 1: Overview of LatenSeer. Operators can pose hypothetical scenarios into a trace-driven causal model and predict how end-user latency would change relative to baseline.

Moreover, a single service might be involved in multiple top-level APIs. As a result, aggregating the distributed traces requires a data structure that is simultaneously succinct while maintaining sufficient diversity of requests to ensure accurate latency estimation [38]. A key idea in LatenSeer is the use of *set nodes* to efficiently cluster traces that exhibit similar execution paths.

Second, the latency of a parent service is a composite of its child services' latencies, underscoring the importance of discerning all causal relationships. Identifying causal dependencies from single requests is straightforward but aggregating them is complex due to clock skews and path-dependent executions. To address this, LatenSeer introduces the *succession time* alongside *set nodes* to aid in deducing the causal dependencies in child RPCs demonstrating similar execution execution paths

Finally, to improve the accuracy of latency estimation, LatenSeer profiles the joint distribution of child service latency, departing from the convenient but misleading assumption of mutual independence upon which most traditional approaches have relied. Joint latency profiling accounts for scenarios where the latency of different child services may be interrelated, thus making latency estimates more accurate than what previous models could achieve.

We implemented LatenSeer as a Python package, and use it in a social network prototype built from the DeathStarBench microservices benchmark (DSB) [26]. We evaluate LatenSeer's ability to make accurate latency predictions through two real world scenarios, service placement (UC1) and slack analysis (UC2). Our results show that LatenSeer predicts end-to-end latency distributions within a 5.35% error (D-statistic) with a mean of 2.7%. In contrast, the state-of-the-art gives a minimum error surpassing 9.5% with a mean of 14.5%. Finally, we evaluate LatenSeer using two production traces (Alibaba [45] and Twitter). We verify the soundness of latency prediction through null prediction experiments and demonstrate the scalability of LatenSeer on massive production workloads.

This work makes the following contributions.

- We propose trace-driven causal modeling as a methodology for answering interventional latency estimation questions in large microservices architectures.
- We detail the design of the LatenSeer framework for extracting causal inter-service dependencies from traces generated by off-the-shelf distributed tracing systems to generate causal models of end-to-end latency.
- We demonstrate LatenSeer's utility by accurately answering hypothetical questions on realistic scenarios with an estimation error within 5.35% (D-statistic), outperforming the state-of-the-art.
- We show LatenSeer's scalability through experiments on real-world traces from large-scale, complex production systems.

2 Background and related work

2.1 The latency estimation problem

Software systems are constantly evolving; these changes can affect latency. Tech giants such as Amazon, Google, and Netflix implement thousands of daily deployments and production changes across hundreds of services that comprise their production environments [24, 40, 68]. Estimating the impact of these frequent production changes ahead of time is crucial, especially for managing risk, ensuring system stability, and maintaining Service Level Objectives (SLOs).

Latency estimation is the task of predicting the impact of hypothetical service changes on end-to-end latency—the entire execution latency of requests to some top-level API endpoint across all the services involved in its execution. The operators of large-scale services are accountable for balancing the service response times with features, resource utilization, policy constraints, and costs. When evaluating potential changes to a service, the operators face the conundrum of assessing how the modifications might impact the response time—a critical factor in end-user experience.

To further elucidate, we highlight the importance of latency estimation by considering several "high-stakes" service changes.

Adding a new microservice. Introducing a new feature often involves adding one or more microservices. While this can improve customer experience, it may also slow down response times as requests navigate through the added service.

Scaling strategies. For a video streaming service, scaling up infrastructure can handle more users but may add latency due to increased server interactions or data replication across multiple sites.

Policy changes. Regulatory requirements might necessitate deployment changes, like storing user data in specific locations. This could lengthen end-to-end latency due to longer data transfer times. Latency estimation can guide operators in assessing and mitigating these impacts.

Latency estimation questions are *interventionist*: they concern hypothesizing how a running system will behave after it is changed. Conventional statistical and machine learning tools, relying on observational data, fall short in answering these queries due to their associative nature [60] and susceptibility to confounding variables [59]. Moreover, while randomized controlled trials offer more precise results, they are resource-heavy and inflexible [25, 74].

2.2 The state of the art

End-to-end latency estimation is important to engineering planning, performance analysis and optimization, however, the traditional approaches are deficient. To explain, we examine the main strategies deployed in practice.

Canarying releases. The canary release strategy, widely utilized in production, facilitates risk reduction by incrementally introducing software updates to a limited user group before broader deployment [82]. With scarce engineering resources, operators wish to conduct latency estimation *before* expensive engineering efforts are spent on the project. Such a feasibility review should precede any live traffic analysis or A/B experimentation—canarying—on the resulting component [79] to minimize cost and customer impact of potentially poor design decisions.

Service dependency graphs. A common technique for understanding the causal dependencies between services is to chart service dependencies, a method widely adopted in production [35, 45, 46, 73, 80]. Large-scale applications requests are represented as call paths through *service dependency graphs*, which capture communication-based dependencies between microservice instances. The call paths show how requests flow among microservices by following parent-child relationship chains. As such, service dependency graphs serve an important tools in discerning the complex interplay of services and optimizing application performance.

Many distributed tracing visualization tools [32, 35, 73, 90] aggregate traces to construct service dependency graphs at various detail levels (we will introduce distributed tracing in §2.3). Yet, none, to our knowledge, have integrated latency distribution into the service dependency graphs. Introducing latency estimation to a microservice-based dependency graph faces hurdles. Alibaba [45] shows that the latency of a service is stable among call graphs of similar topologies but varies significantly across different topologies, and the latency of a service is stable when the call rate varies. Some works [46, 80] use dependency graphs to guide autoscaling and service migration. Tprof [31] aggregated traces in a fine-grained manner of sub-span analysis for performance debugging.

While a service dependency graph outlines relationships between services, it fails to detail the micro-level dynamics of

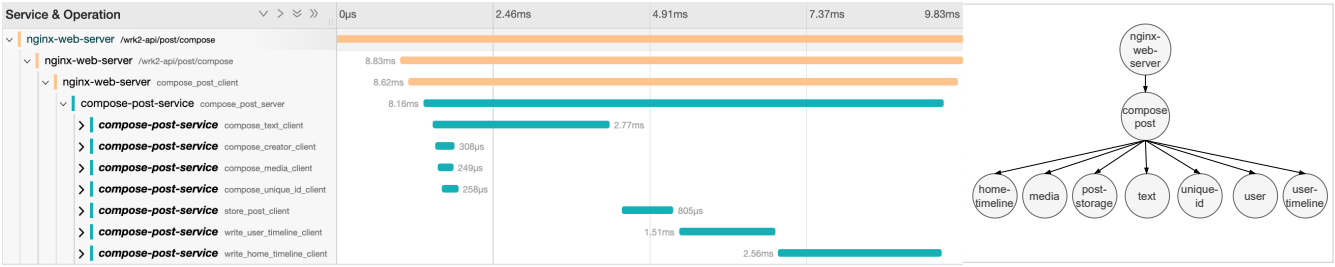


Figure 2: An example trace from Jaeger. The left diagram shows the trace in a timeline view, and the right diagram shows the service dependency graph generated by Jaeger.

Table 1: LatenSeer compared to state-of-the-art systems.

	LatenSeer	ORION	CRISP	WebPerf	MysteryMachine
G1	✓		✓		✓
G2	✓	✓		✓	✓
G3	✓	✓		✓	

execution order, including the nature of sibling relationships, whether parallel or sequential. For instance, the service dependency graph in Fig. 2 illustrates service `compose-post` must await the responses from all seven child services to complete its task. Yet, if we aim to comprehend how changes in the `media` service affect `compose-post`, the service dependency graph falls short. This limitation impedes accurate latency estimation, posing a risk of relying on models grounded in potentially incorrect assumptions about the interrelationships of child services.

Latency-critical paths. In microservice dependency graphs, the latency-critical path represents the slowest sequence of dependent tasks [84]. While its analysis facilitates the identification of optimization prospects and bottlenecks in distributed systems [7, 9, 12, 16, 21, 35, 61, 88], it can inadvertently obscure potential issues in off-path services. Hence, incorporating *slack* time analysis for off-path services is advocated, aiding in refined capacity planning and mitigating risks arising from shared-resource contention [16].

CRISP [88] uses critical path analysis on Uber traces to help developers understand and optimize important services. However, services off the latency-critical path are easily trimmed from the results. For example, service `media` in Fig. 2 are not shown in the final results of CRISP. A singular focus on the latency-critical paths paints an incomplete and misleading picture for end-to-end latency estimation, emphasizing the need for a more holistic approach that incorporates the dynamic roles of all services in the system.

Latency modeling and performance prediction. Gleaning causal dependencies in existing large-scale systems, such as building causal models of latency, has been explored in the past decade. Researchers from Google, for example, built theoretical frameworks to understand the latency profile of arbitrary black box services [37, 55], which, like more recent work over microservices [44], is focused around anomaly detection. More recent white-box approaches to performance modeling have also been proposed [2, 62].

Facebook's Mystery machine [12] shares the motivation of constructing a causal model using pre-existing data, predating modern tracing infrastructure, that works by initially hypothesizing all possible pairwise relationships and gradually rejecting the causality of each dependency through counterexamples. Regrettably, the ensuing predictions are brittle owing to incomplete methods for producing causal diagrams [34] and cannot handle various issues in tracing data, such as clock skew and missing span. CRISP [88] shares a similar motivation **(G1)** but lacks a causal treatise of latency estimation. Similarly, Orion [49] also uses latency propagation technique to calculate end-to-end latency. However, it focuses on known service DAGs and fails to address the causality in service dependencies. The study most closely aligned with our objectives is WebPerf [33], which crafts techniques surrounding service dependencies and latency propagation. However, WebPerf is specifically tailored to Microsoft's Azure environment and relies on Azure-specific hints to work. WebPerf employs domain-specific binary instrumentation, presenting a solution that isn't universally adaptable. In contrast, LatenSeer advances this approach by adopting a data-driven strategy, relying on trace data that are commonly available in today's production systems.

Latency prediction is a well-trodden field with substantial research in areas such as service selection [19, 85], service composition [3–5, 10, 87], and business process modeling [14, 66]. Notably, with the rising popularity of serverless workflows in many applications, there has been a surge in research efforts aimed at latency prediction for these workflows. These efforts predominantly focus on resource optimization and reducing communication latency in serverless workflows [17, 18, 50]. Distinctively, LatenSeer, not constrained by predefined workflows, offers a flexible solution to latency predictions in microservice-based applications.

2.3 The proliferation of distributed tracing

Distributed end-to-end tracing tracks requests' execution paths in a distributed system or application across different components, including frontend devices, backend services, and databases. Many production systems deploy distributed tracing frameworks, such as OpenTracing [54], Zipkin [90], and Jaeger [32], to track request traces and aggregated metrics, which is useful to examine hand-off between system components [8, 57, 72], and to troubleshoot practical system problems such as slow responses and errors [6, 20, 23, 27, 35, 43, 48, 53, 78]. The primary use cases of distributed end-to-end tracing systems concern active system monitoring [8, 57, 72], anomaly detection [6, 11, 27, 39, 47, 65, 69, 73], and root-cause analysis [55, 83].

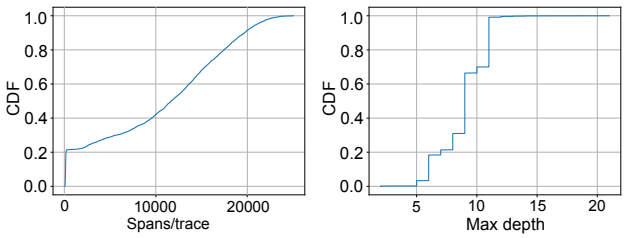
In our context, trace aggregation can be used to delineate the interdependencies among all services within a microservices system—allowing for data that is routinely collected to be useful for latency estimation **(G1)**. The left side of Fig. 2 shows a visual example of a trace constructed by Jaeger [32], a state-of-the-art open source tracing framework, for a compose post request to a benchmark of social network application [26]. An end-to-end *trace* represents an execution path through the system, whereas a *span* represents a logical operation from a function. A span maintains not only the causal relationship by keeping a reference to a parent span but also the runtime execution details, such as start and end timestamps to represent the duration of the operation. A trace can be considered as a directed acyclic graph (DAG) of spans, where each edge represents the causal relationships, such as RPC calls, between two spans [72]. Besides tracking causal relationships, a trace also captures temporal order between a group of child spans that are concurrently called by the same span.

We show an illustration of a service dependency graph, created from 20 traces of `compose-post` requests, in the right-hand-side diagram of Fig. 2. The completion of service `compose-post` is contingent on even other services, such as service `media` and service `text`. Below, we refer to the services that are invoked by the same "parent" service in a service dependency graph as *sibling nodes*.

2.4 Challenges of using distributed traces to predict latency

Distributed tracing confers a great opportunity for piggybacking latency estimation on existing data, providing end-to-end visibility, and revealing service dependencies. Yet this direction cannot provide end-to-end latency estimation out of the box due to multiple challenges.

(1) Raw traces fail to capture interdependencies. The perspective offered by each individual trace is too insular to fully encapsulate the complexity of the entire system.



(a) Median spans/trace=11676 (b) Median max call path depth=9

Figure 3: Twitter trace characteristics.

Yet aggregation techniques to produce service dependency graphs exhibit significant blind spots. Chief among them is the problem that sibling nodes may themselves have complex interdependencies that, when uncaptured, can result in an incomplete portrayal of the performance dynamics and intricacies, particularly in hypothetical scenarios, thereby undermining the accuracy of latency estimation. Nevertheless, a cautious and careful approach can overcome these limitations, as shown in §3, allowing distributed tracing to be valuable resource for latency estimation.

(2) Small inaccuracies can snowball. The inevitability of clock skew is a major issue for nuanced processing of distributed tracing data. In a distributed tracing deployment, spans are timestamped using the local machine's clock upon start and finish. However, the clocks on different machines in a distributed system invariably drift apart, even with periodic synchronization using the Network Time Protocol (NTP) [36, 41, 51, 52, 70]. This well-understood problem in distributed computer stems from various factors, such as clock hardware differences, environmental conditions, network latency, and the resolution of the system clock. Consequently, this can lead to misinterpretations in system performance analysis and incorrect event ordering [12, 88], with minor errors potentially amplifying to significantly impact conclusions on performance behavior.

(3) The problem of scale. A single trace is not representative of the full service—any interventional questions must account for the diversity of traces that execute in a production environment. At Twitter, for instance, a request involves 12,000 spans on average, with some traces encompassing as many as 25,000 (Fig. 3a). The call graphs for one endpoint comprise between 1 and 25,000 spans at tree depths between 2 to 22 (Fig. 3b) and encompass widths between 1–5,000. Uber, similarly, operates nearly 4,000 microservices, and a single request trace can have up to more than 11,000 spans nested 40 levels deep [88]. Existing tracing tools focus on providing the operator with a single, specific trace [31, 48], such as to identify an edge-case in Jaeger [86], or analyzing the path of a single request [48]. How to properly aggregate thousands, or even millions, of distributed traces to gather insights is a nascent and understudied area [88].

```
# Load distributed trace data
model = LatenSeer(data="s3://trace_bucket")
# UC1:Add hypothetical 30ms delay to service 'LBS'
scenario = model.apply('LBS'=lambda x:x+30)
# Get the changed end-to-end latency distribution
E2E = scenario.predict()
# UC2:Determine slack of 'LBS' service
LBSslack = model.slack('LBS')
```

Figure 4: LatenSeer Example. Estimate latency distribution if load balancing services are migrated to another data center with 30ms extra delay.

3 Design

LatenSeer is an offline tool for conducting latency estimation from distributed traces. This tool permits the system operators to infer how end-to-end latency would be affected when the latencies of specific services are changed. In this section, we describe how LatenSeer overcomes the aforementioned challenges to derive a causal model of steady-state latency from historic distributed trace data of the system.

Interface. Fig. 4 shows LatenSeer in action, estimating end-user latency distributions under the assumption that load-balancing services have been migrated to a distant data center (UC1). The operator first obtains a baseline model of latencies based on trace data. As input, the operator poses an interventional query to the model by defining a subset of services to move (*e.g.*, 'LBS'), and the delay incurred (or reduced) from the movement (*e.g.*, a normal distribution centered on 30 ms) triggering the latency propagation via `model.apply` which returns a latency-modified model, obtaining a predicted latency distribution as output (via `scenario.predict`). Separately, the operator also determines the available latency budget for the `LBS` service through a `model.slack` call, which returns CDFs of latency slacks for all services (UC2). The latency distributions in LatenSeer can be further stratified by arbitrary groups (*e.g.*, to assess latency impacts on customers in certain regions)—we focus on a single group for the clarity of presentation.

Key properties. To estimate end-to-end latency under potential alterations in any component services, LatenSeer constructs a model that fulfills the following properties: (1) it delineates the causal dependencies between microservices. (2) it recognizes the range of request routes within the system, accounting for the different paths a request might take and the specific services it might encounter. (3) it maintains the latency distributions of interactions (e.g., RPC or REST) between each pair of communicating services.

3.1 Modeling latency with invocation graph

We begin by elucidating the principles behind latency calculation using a single trace as an illustrative example.

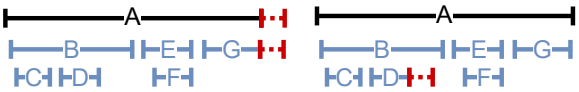


Figure 5: Child service latency may affect parent service.

The underlying concept is simple: the latency of a parent service can be deduced from the latencies of its child services. For example, Fig. 5 illustrates a trace consisting of a parent service A and six child services (B-G), where B, E, and G are on the latency-critical path. We define the following functions on a particular trace: $\mathcal{L}(x)$ the latency of service x, and $\mathcal{L}(\mathcal{N})$ which accounts for the network latency or other processing time on the latency-critical path. For instance, it includes the duration between B's finish time and E's start time. Consequently, the latency of the span A can be expressed as $\mathcal{L}(A) = \mathcal{L}(B) + \mathcal{L}(E) + \mathcal{L}(G) + \mathcal{L}(\mathcal{N})$.

Now we discuss how latency changes on different child services impact the latency of the parent service. Since services B, E, and G are on the latency-critical path, any delay in these services will increase the latency of A. However, the latency of A can also be affected by other services that are off the latency-critical path, such as C, D, and F in this example. We summarize four scenarios of latency change on a service and their impacts[3]:

(1) An increase in the latency of a child on the latency-critical path invariably leads to an increase in the parent latency
(2) A decrease in the latency of a child on the latency-critical path can result in unchanged or reduced parent latency, potentially altering the latency-critical path itself.
(3) When a child off the latency-critical path experiences increased latency, the parent latency might remain the same or increase, contingent upon changes to the latency-critical path.
(4) A decrease in the latency of a child off the latency-critical path leaves the parent latency unaffected.

Fig. 5 provides two examples demonstrating cases (1) and (3) respectively.

As previously discussed, the latency-critical path exhibits dynamism due to latency changes in different services. To assess the impact of these changes on the latency of a parent service, we construct an ***invocation graph*** for the child services. This graph captures the sequential execution order of the child services within a specific trace. In the invocation graph, an edge $\mathcal{E}(x, y)$ represents service y finishes before service x happens. Node *start*, *end*, and *sync* are virtual nodes that denote the starting, finishing, and synchronization points, respectively. For example, node G will not start until both nodes E and F have finished. Fig. 6 shows a comparison between a

[3]We can effectively calculate the latency changes on multiple services. For ease of presentation, we only show latency change on one service.

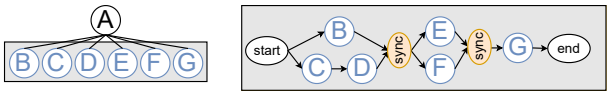


Figure 6: From service dependency tree to invocation graph.

standard service dependency tree and an invocation graph derived from the same trace (Fig. 5). Invocation graphs represents child relationships with finer granularity, a detail not captured by service dependency trees. Moreover, each node is endowed with a latency value corresponding to its service. The latency of the parent service can be expressed as $\mathcal{L}(A) = max(\mathcal{L}(B), \mathcal{L}(C)+\mathcal{L}(D))+max(\mathcal{L}(E), \mathcal{L}(F))+\mathcal{L}(G)$. Given this formulation, any alterations in a child service's latency allow for a direct computation of the resulting impact on the parent service's latency.

We now describe how the conventional service dependency tree can be converted to an invocation graph. We use *serial* and *parallel* to describe the relationship between any two spans. Consider, for example, spans B and E depicted in Fig. 5, we define the *succession time* as the difference between the starting time of E and the finishing time of B. Given that B starts before E from the vantage point of the parent observer, we say that B happens before E. If the succession time is positive, we declare the two spans to be serial. Conversely, a negative succession time, such as the overlapping time frames of E and F, indicates a parallel relationship between two spans.

The initial step involves identifying sync points, which serve to demarcate nodes into chronological groups. Each group is characterized such that services from a preceding group must complete their execution before the services in the succeeding group initiate. To establish these groups, we construct a graph $\mathcal{G}_p$ comprising child nodes and connect nodes that are in parallel relationships. Subsequently, we identify the connected components in the graph. Illustrating with the child nodes (B-G) in Fig. 6, we can construct a graph $\mathcal{G}_p$ consisting of nodes B, C, D, E, F, and G, interconnected through edges B-C, B-D, and E-F. Consequently, the connected components are $\{B, C, D\}$, $\{E, F\}$, and $\{G\}$. To complete this step, we arrange these groups chronologically based on the earliest start time among the nodes within each group.

The next step is to identify the causal orders among nodes within each group. Within each group, we establish connection between pairs of child nodes that are in serial relationship, forming a graph denoted as $\mathcal{G}_s$. In this graph, the maximal cliques, or *independent sets* in graph theory parlance [28] are the nodes with serial relationships. And the nodes within each maximal clique are then ordered by their starting timestamps. For example, in the first group $\{B, C, D\}$, maximal cliques are $\{B\}$ and $\{C, D\}$.

Drawing upon the aforementioned two steps, we can determine the invocation order of the child nodes. When a latency change occurs in any given service, we can identify the slowest path within the invocation graph, thus determining the overall latency of the parent service.

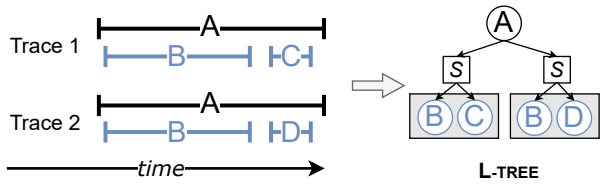


Figure 7: Aggregate traces to L-TREE.

3.2 Aggregating traces with L-TREE

We have presented the foundational principles of latency modeling. In this section, we detail the construction of a latency-endowed service dependency tree, referred to as the **L-TREE**, which aims to refine aggregate-level latency calculations.

The construction of the L-TREE is a top-down process by two main steps. Firstly, we iterate through traces to establish set nodes between parent spans and their direct children, merging identical spans where necessary. This forms an initial L-tree structured by set nodes, albeit without invocation graphs. Secondly, we traverse this preliminary tree to construct invocation graphs at each set node, drawing upon stored latency profiles. This produces a detailed L-tree. The subsequent parts of this section will delve deeper into each stage of this construction process.

Set node. The *set node* functions to effectively cluster together traces that exhibit similar invocation graphs. Each *regular node* in the L-TREE therefore denotes a span in the original traces. When multiple traces include the same span, the corresponding node in the L-TREE represents aggregate information of that particular span across the traces of the same call path. Each regular node records how many traces had the span to which it corresponds. A set node connects a regular node with a collection of regular child nodes and indicates an identical set of child nodes—the same set of spans[4]. We iteratively input traces, and merge the new input trace into L-TREE; if a new set of child spans occur, we create a new set node. Moreover, each set node remains a record of the number of traces it encompasses, thereby providing insights into the different branch ratios present in the L-TREE. For instance, Trace 1 and Trace 2 in Fig. 7 have same parent span A, but different sets of child spans $\{B, C\}$ and $\{B, D\}$. The right diagram in Fig. 7 shows a L-TREE for these two traces, where two set nodes are created to connect the parent node A and two different sets of child nodes.

Set nodes can cluster traces that exhibit similar invocation graphs, which effectively retain the diverse characteristics of calling relationships, thereby enhancing the modeling

[4]A set node effectively represents a *hyperedge* in a tree-based hypergraph.

of latency distribution. In constructing the tree from the top down, traces sharing the same set of child RPC calls from a single parent call are clustered at each level. This approach strikes a balance between coarse aggregation (grouping all spans without differentiation) and fine-grained aggregation (grouping traces only if they share identical invocation graphs).

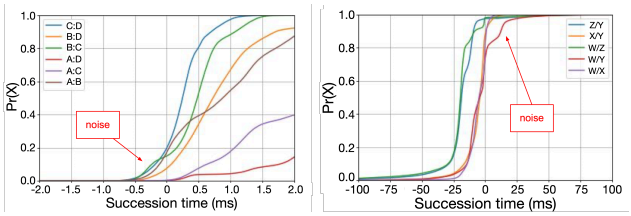
Joint latency profiling. Traditional service dependency modeling methods only store latency for each node independently [33, 49], which misses the finer-granular causalities between child services. L-tree maintains latency profiles differently, by jointly considering the latencies of sibling nodes. When forming a tree from a single trace and extending it to multiple traces, the latency data associated with each node is profiled as a distribution, rather than a single value. The set node manages the latency profiles of its child nodes. Besides maintaining the latency distribution for each child node, we also profile the succession time distribution between pairwise sibling nodes, which is crucial to construct an invocation graph for the sibling nodes.

Constructing invocation graphs in L-tree. Finally, we describe the construction of invocation graph in L-tree. This process resembles the one described in §3.1, but with a key modification: we use the median of the succession time distribution between two sibling nodes as a threshold to determine their serial and parallel relationships. We hypothesize that service invocations that occur in series should consistently have positive succession time between them, whereas parallel spans may show negative succession time in some traces. Unfortunately, in production settings, the measurements are not always clear-cut, as shown in Fig. 8, due to clock skew and other instrumentation artifacts. The adoption of the median value as the classification threshold serves to mitigate these challenges, promoting a more accurate delineation of serial and parallel relationships.

This concludes our construction of L-tree, where each regular node links to one or more set nodes indicating different call paths. Each set node maintains an invocation graph between the child nodes of the set node's parent node. Consequently, L-tree not only maintains similar call paths across different tree branches but also captures the sequential execution order of child services.

3.3 End-to-end latency estimation

LatenSeer estimates the end-to-end request time by combining the latency distributions in L-tree, propagating them bottom-up from leaves to the root. If an operator thus modifies the latency of specific nodes within the L-tree that corresponding to the services they tend to change, LatenSeer will infer the latency impact from these changes through the modified latency distribution of the root node. For example,



(a) Known serial siblings. **(b) Known parallel siblings.**

Figure 8: Succession time CDF for production services at Twitter over a 24-hour period.

service B is delayed by Δ_B in Fig. 7, LatenSeer propagates the increased latency Δ_B, calculates the changed latency Δ_A for service A, and outputs A's new latency distribution $\mathcal{L}(A)'$.

With the collection of traces, LatenSeer models node latency as probability distributions. Formally, for serial nodes with latency distributions represented as random variables X_1,... X_k, we define Add operator to estimate their combined distribution as: $\mathbb{P}(Z = z) = \sum \mathbb{P}(X_1 = x_1, ..., X_k = x_k)$, where $z = \sum_{i=1}^{k} x_i$. On the other hand, we define Max operator to estimate their combined latency: $\mathbb{P}(Z \leq z) = \mathbb{P}(X_1 \leq z, ..., X_k \leq z)$.

Node latency estimation. We estimate the latency distribution of node v through its set nodes and corresponding invocations graphs of their child nodes. Suppose that a set node s connects an invocation graph $\mathcal{G}$ containing m sync points. Assume there are p_j paths between two sync points and n_j nodes on j-th path. We use P_j^i to denote the set of nodes on j-th path between $(i-1)$th and ith sync points. Then the combined latency ℓ_i of nodes between $(i-1)$th and ith sync points can be calculated as:

$$\ell_i = \textsc{Max}(\textsc{Add}(P_1^i), ..., \textsc{Add}(P_{p_1}^i))$$

Finally, the latency of set node s is calculated as:

$$\mathcal{L}(s) = \textsc{Add}(\ell_1, ..., \ell_m)$$

Suppose the latency distribution of the node v depends on k set nodes, denoted as s_i. The latency $\mathcal{L}(v)$ can be expressed as a weighted sum of the latencies of these set nodes, represented as $\sum_{i=1}^{k} w_i \mathcal{L}(s_i)$, where $w_i \in [0, 1]$ denotes relative weights with $\sum_{i=1}^{k} w_i = 1$. The relative weight is determined based on the number of traces recorded by each set node.

End-to-end latency estimation. LatenSeer combines the estimated latency distributions of all nodes in the L-tree to produce one estimated distribution of end-to-end latency. We first inject the changed latency ($\Delta \in \mathbb{R}$) to the target nodes that correspond to the services for which the operator wants to intervene. (In causal modeling, this change emulates a do-operator [59, 60]). The algorithm works *bottom-up*, taking as input the L-tree, and the set of hypothetical services to be altered with their corresponding latency (Alg. 1). The changes

Algorithm 1 Latency Estimation

Require: L-TREE G; a "what-if" scenario S
Ensure: Estimated end-to-end latency distribution

```
1:  function PREDICT(G, S)
2:      r ← root node of G
3:      A ← APPLY(r, S)                          ▷ Nodes affected by scenario S
4:      d ← greatest depth among nodes in A
5:      while d > 0 do
6:          Nodes ← A[d]
7:          for each regular parent p of a node in Nodes do
8:              ℓ, affected ← LATENCYPROPAGATION(p, A)
9:              if affected then
10:                 update distribution of p with ℓ
11:                 A[p.depth].insert(p)
12:         d ← d − 1
13:     return updated distribution of root node r
```

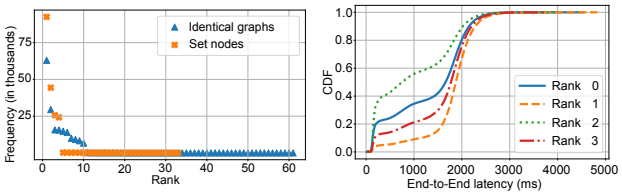
trigger bottom-up latency propagation within the L-TREE from the affected nodes toward the root. If a node displays variable latency, we recalibrate its parent node's latency using the ADD and MAX operators, following the structure of the invocation graph. Importantly, any modification in the latency of a parent node prompts a subsequent recalculation of its own parent, continuing this chain recursively up to the root node.

3.4 Making LatenSeer practical for production workloads

LatenSeer's design aims to solve latency estimation for real-world usages at scale. We discuss some design choices and optimizations that make LatenSeer practical, especially from our experiences when dealing with massive trace data at Twitter.

(1) Handling clock skew and miss data. To derive the serial and parallel relationships between nodes, we must compare the timestamps on two spans. These spans often represent RPCs that are likely emanating from different machines, which poses a potential issue as the clocks across these machines are not perfectly synchronized, thus could introduce inaccuracies in our comparisons. We mitigate this issue by using client-side timestamps, which record the times on the same machine where the RPC calls are invoked. In other words, the start and finish times of child spans that we use are the timestamps recorded from a machine where the parent span locates. In this manner, the duration of a child span consists of both its processing time and network latency. Handling missing or erroneous data in tracing systems is an open problem. We tackle this by truncating traces at the initial missing span connection. Comparisons on Alibaba traces revealed minimal latency differences between complete and truncated traces.

(2) Set nodes for clustering traces. As mentioned in §3.2, we introduce the concept of *set node* to cluster together traces



(a) Frequency of identical dependency graphs and set nodes. **(b) Latency distribution of different set nodes from same parent node.**

Figure 9: Set node justification via Twitter traces.

that exhibit similar invocation. We now provide a more detailed explanation behind this design decision. Given the diverse call patterns in large-scale systems, clustering traces based on identical dependency graphs is often impractical. For instance, a service invoking four `cache-get` operations and a service executing five `cache-get` operations would fall into separate clusters when using an identical clustering approach. Fig. 9a illustrates the frequency of identical dependency graphs and set nodes of a root node (a front-end service) in Twitter, derived from examining 190,087 traces with the same front-end API. The rank in Fig. 9a refers to the ranking of identical dependency graphs or set nodes based on their frequency. In this context, we focus exclusively on the first depth of the full graph: the root node and its immediate child nodes. Recall that the identical dependency graphs are those where child nodes exhibit precisely the same sequence of invocations, and the set node groups the traces which have the same set of child nodes. Fig. 9a reveals that the number of different identical dependency graphs is twice that of set nodes. This set node concept, therefore, offers a more practical and effective way of grouping traces for latency modeling in complex systems.

The adoption of set nodes is substantiated by their ability to identify similar call patterns in traces sharing the same set of child RPC calls. Conversely, different set nodes usually correspond to varying latency distributions, a consequence of the unique call paths each trace navigates, as exemplified in Fig. 9b. Further supporting this approach, we've observed that the top two ranks of identical service dependency graphs exhibit the same latency distributions and belong to the same set node (the finding is not shown in the paper due to space constraints.). This consistency lends further credibility to the concept of the set node, underscoring its practicality and relevance in understanding and modeling system latencies.

(3) Pre-merging *parallel spans*. The traces in large-scale applications tend to be complex, with a typical request touching tens to thousands of individual microservices, producing service dependency trees that are up to 20 levels deep. To reduce the complexity of computation, we presume some sibling spans are parallel and coalesce these spans into the same (regular) node. According to observations of a large

number of Twitter traces, we found most simultaneous RPCs to cache and storage are parallel. For example, hundreds of `cache-get` happening simultaneously often have a parallel relationship. Pre-merging such spans significantly reduce the L-TREE complexity.

(4) Building LatenSeer trees in parallel. LatenSeer must be efficient enough to process an enormous volume of traces for large-scale applications. To facilitate this, we parallelize the tree-building process. The traces are initially partitioned based on their top-level API endpoint, as identical API calls tend to exhibit similar calling patterns. Subsequently, traces within the same API endpoint are evenly distributed into the same bucket, and a subtree is constructed for each batch of traces. Finally, subtrees for traces with the same top-level API endpoint are merged. This merging process commences from the top and progresses downwards. When nodes are identical, they are merged along with their latency profiles. If nodes are distinct, the unique node is simply incorporated. This methodology ensures that the final tree accurately represents the diverse and complex calling patterns inherent in the large volume of trace data. We implemented L-Tree using a distributed data-parallel processing framework in Twitter. The framework can produce the results daily or weekly based on requirements.

4 Evaluation

In this section, we evaluate LatenSeer to answer the following questions.

- Can LatenSeer provide accurate end-to-end latency estimation results?
- How well is LatenSeer estimating the end-to-end latency with real-world use cases?
- How is LatenSeer compared with the state-of-the-art?
- How effectively can LatenSeer handle large-scale traces?

4.1 Experimental Setup

Prototype system. We implement LatenSeer[5] in ≈ 3,000 lines of Python3 code and test it with DeathStarBench microservice benchmark (DSB) [26] on *Social Network* application, which consists of 31 unique microservices. We leverage DSB's default workload generator to produce client requests. Furthermore, we fulfill all the missing function-level instrumentation in DSB.

We set up experiments on CloudLab [15] under two deployment topologies using three different physical sites, as shown in Fig. 10. We deploy the benchmark on 6 machines at the Utah site, 6 machines at the Wisconsin site, and 1 machine at the Clemson site. In both scenarios, we run the workload generator at the Clemson site on a node type "c6320"

[5]LatenSeer is open-sourced at https://github.com/yazhuo/LatenSeer, and the Twitter traces will be released upon legal approval.

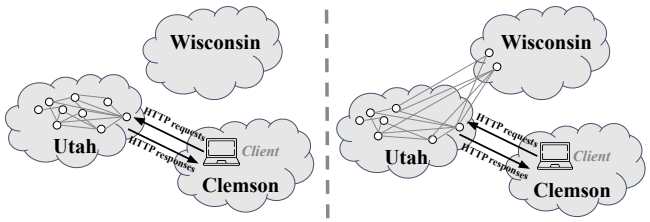


Figure 10: Prototype experiment setup using three Cloudlab sites. The left shows the single-site deployment where all microservices run in the Utah cluster; the right shows multi-site deployment where some microservices run in the Wisconsin cluster.

(Haswell 28-core with 256 GB RAM and 10 Gbps network). Microservices in Utah and Wisconsin ran on node types "xl170" (Broadwell 10-core with 64 GB RAM and 25 Gbps network) and "c220g5" (Skylake 20-core with 192 GB RAM and 10 Gbps network), respectively. RPC latencies within a single site are < 1 ms, while latencies between Utah and Wisconsin are in the range 38–42 ms.

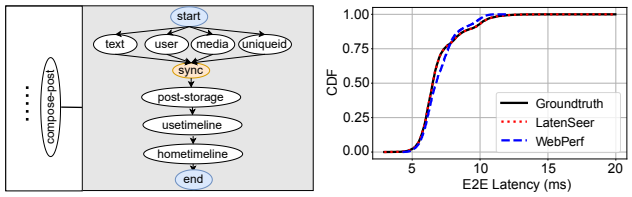
Methods. Both the traces and our measurements report the end-to-end request processing latency at the API gateway. We call the latency distribution that LatenSeer outputs for a specific service change scenario the *prediction*. We term a measured latency distribution for a given scenario *ground truth* and use it to quantify the accuracy of the corresponding prediction. When we exercise LatenSeer to make a prediction without perturbing the model latencies (typically to validate the model itself), we call the output latency distribution the *null prediction*.

Except for the sensitivity experiment that varies the request mix, the workload generator sends requests to three endpoints using the default read-dominated ratio of 1:3:6, with one `compose-post` request for every three calls to `read-user-timeline` and six `read-home-timeline` requests. We use the same request rate of 500 RPS in all cases.

Each experiment proceeds as follows. We first run our workload for 10 minutes in the baseline configuration to collect the traces for building the model in LatenSeer. Next, we inject an estimated latency delta (specifically, the average value from a ping test) into the model at the appropriate microservices and invoked `scenario.predict` to generate a latency prediction. We then run the workload again on the changed deployment for 10 minutes in order to measure the ground truth latency distribution. We compare prediction results with the ground truth. All traces are collected using Jaeger at a 10% sampling rate.

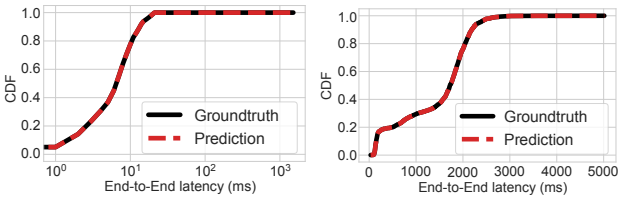
Metrics. For a prediction latency CDF F_{pred}, and a ground truth latency CDF F_{true}, we report prediction accuracy using two metrics:

(1) The D-statistic in the K-S (Kolmogorov-Smirnov) test, which is the difference between two CDF curves at the



(a) Ground truth (b) LatenSeer vs. WebPerf

Figure 11: LatenSeer compared with state-of-the-art.



(a) Alibaba (b) Twitter

Figure 12: Null prediction on production traces.

point of maximum divergence. It is defined as $D = \max_x \left|F_{\text{pred}}(x) - F_{\text{true}}(x)\right| \in [0, 1]$.

(2) The maximum and average of the relative latency error at each percentile. It is defined as $E = \left((F^{-1}_{\text{pred}}(y) - F^{-1}_{\text{true}}(y)\right) / F^{-1}_{\text{true}}(y) \in \mathbb{R}$, where $y \in [0, 1]$. A negative value implies that the prediction is lower than the ground truth; a positive value means that the prediction is higher latency than the ground truth. We focus on the mean and max relative error, denoted as E_{avg} and E_{max}.

The K-S statistic is a standard metric for comparing two CDFs, but because it captures only the most extreme point of misprediction, it can be large even when the two CDFs are mostly similar. In practice, it is also useful to know how well LatenSeer predicted latency across the entire distribution relative to the absolute values of the ground truth. Therefore we also report the relative error metric.

Traces. To evaluate scalability of LatenSeer in handling production-scale traces, we use two production traces from Twitter and Alibaba. The Twitter traces consist of up to 25,000 spans, with a depth spanning from 2 to 18 hops. The Alibaba traces possess up to 6,625 spans with 1 to 14 hops.

For latency estimation experiments, we operate our prototype system and gather traces using Jaeger [32] as part of the DSB deployment. The collected DSB trace set encompasses a variable number of spans, fluctuating between 5 and 33, with the maximum depth ranging from 2 to 6 RPC hops.

Baseline. We compare LatenSeer with WebPerf [33], the state-of-the-art latency estimation work. We reimplement WebPerf's model because it's not open-sourced, and tame WebPerf's model to fit our scenarios. WebPerf is based on customized low-level instrumentation for the extraction of causal dependency graphs, which we find not practical in today's tracing framework usages in most production environments. We resort to examining the DSB codebase to discern these dependencies and construct the corresponding graph to meet WebPerf's requirements.

4.2 Estimation Accuracy

Through code reading for DSB benchmark, we derive an true causal dependency graph. This graph is then compared with the L-TREE generated by LatenSeer. Our analysis shows that LatenSeer accurately replicated the same dependency relationships present in the true graph. Fig. 11a presents a section of the L-tree at its highest complexity, which also aligns with the trace depicted in Fig. 2.

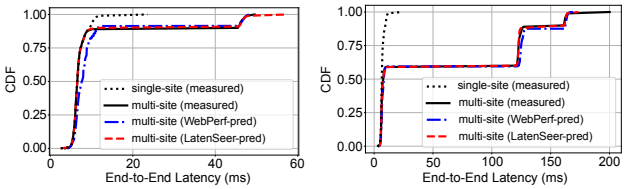
We then conduct a *null prediction* experiment to validate the soundness of our model. In this setup, we inject a "null" latency into the leaf nodes, triggering the latency propagation procedure to traverse all the nodes in the tree. We expect that the prediction aligns closely with the ground truth. As shown in Fig. 11b, LatenSeer precisely calculates the end-to-end latency distribution, while WebPerf generates larger errors with $D = 15\%$, $E_{max} = 8.8\%$ and $E_{avg} = -2.0\%$. The superior accuracy of LatenSeer over WebPerf can be attributed to the assumptions made by each tool regarding latency. WebPerf operates under the assumption that latencies on different components are independent of one another, while LatenSeer takes a more holistic approach by profiling the latencies jointly for sibling nodes.

Since the ground truth of causal dependency graphs for the production traces is not available, we limit our examination to whether the model accurately predicts the latency distribution of the input traces through null prediction experiments on two production traces. The results in Fig. 12 show that predictions align closely with the ground truth for both Alibaba traces (Fig. 12a) and Twitter traces (Fig. 12b), confirming that the internal service relationships are modeled faithfully.

4.3 Case Studies

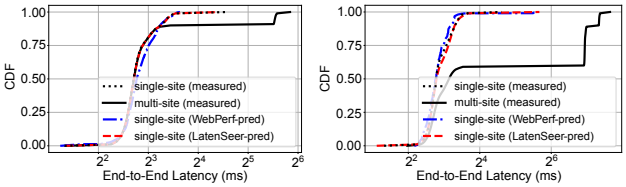
We evaluate how well LatenSeer models end-to-end request latency in a microservices environment using DSB. To examine the key properties, we focus on the two use cases: service placement (UC1) and latency slack (UC2).

4.3.1 Service placement (UC1) We study the accuracy of latency predictions under real, albeit controlled, conditions by comparing the predicted and measured latency distribution following wide-area service migration. In order to experimentally exercise as much of LatenSeer's model as possible, we use three API endpoints: `read-home-timeline`, `read-user-timeline`, and `compose-post`, and select three microservices for migration: `user-timeline-service`, `user-service`, and `media-service`.



(a) Migration of parallel services, affecting a single endpoint **(b) Migration of serial services, affecting two endpoints**

Figure 13: Prediction accuracy for migrating from local to remote.



(a) Parallel services migration, affecting a single endpoint **(b) Serial services migration, affecting two endpoints**

Figure 14: Prediction accuracy for migrating from remote to local.

- The `read-home-timeline` endpoint does not interact with any of the chosen services, hence unaffected by their migration.
- The `read-user-timeline` endpoint only interacts with `user-timeline-service`.
- The `compose-post` endpoint interacts with all three services, and invokes them both in parallel (`user-service` and `media-service`) and serially (`user-service` and `user-timeline-service`).

In the following experiments, we explore how accurately LatenSeer predicts end-to-end latency when selected microservices are migrated over the wide area network from one CloudLab site to another. We set up the experiments as described in §4.1, generating the model from the single-site deployment, measuring ground truth in the multi-site deployment, and using the average ping time measurement of 38.7 ms between the Wisconsin and Utah sites as the injected latency delta.

Migration that increases latency. Fig. 13 shows the prediction vs ground truth CDFs for experiments with two different pairs of services, each graph showing the top-level request latency distribution measured at the single-site (dotted line), the ground truth measured with the multi-site deployment (solid line), the prediction from WebPerf (dash-dot line), and the prediction from LatenSeer (dashed line). For the left-hand graph, Fig. 13a, the `user-service` and `media-service` were moved from Utah to the Wisconsin site. Only one endpoint, `compose-post`, should be affected by this migration. This endpoint comprises 10% of the default workload mix, and the two microservices have a parallel relationship within requests for that endpoint.

Fig. 13b shows the prediction when `user-timeline-service` and `user-service` were migrated. These microservices affect not only `compose-post`, with a serial relationship therein, but also impact `read-user-timeline`, which is called 30% of the time. Note that in both cases, latency is extended by different amounts depending on the number of cross-site calls introduced by the service migration.

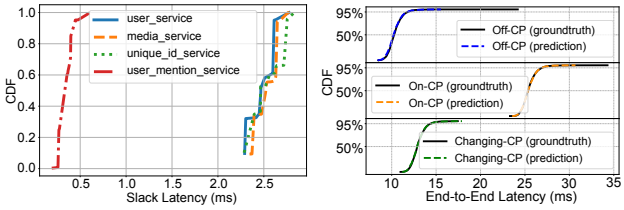
LatenSeer shows highly accurate predictions for both scenarios, as summarized in Table 2. Even though the K-S D statistic for LatenSeer for the right-hand graph is almost 5%, the average relative error is extremely low at -0.11%. In comparison, WebPerf shows much lower accuracy with 19% and -1.22% for D statistic and average relative error, respectively. Note how the shape of the CDF changes when some services are placed remotely: the jumps reflect the workload mix and how the three request types are affected (or not) by the migration. Thus Fig. 13a has just one jump at around the 90th percentile (because 10% of requests are `compose-post`), while Fig. 13b shows two jumps for `read-user-timeline` and `compose-post`, both of which touch the migrated services. The change in the shape of the latency distribution highlights that one cannot simply estimate the latency impact of service migration by offsetting the baseline distribution with a constant value and emphasizes the importance of LatenSeer's modeling techniques.

Table 2: Results of service placement experiments.

Experiments	Model	D(%)	E_{max}(%)	E_{avg}(%)
Parallel services	LatenSeer	1.31	2.30	-0.21
-> remote (Fig. 13a)	WebPerf	18.60	27.00	9.30
Serial services	LatenSeer	4.68	11.50	-0.11
-> remote (Fig. 13b)	WebPerf	19.04	28.00	-1.20
Parallel services	LatenSeer	1.45	1.10	0.22
-> local (Fig. 14a)	WebPerf	10.50	10.70	0.60
Serial services	LatenSeer	5.35	8.00	1.50
-> local (Fig. 14b)	WebPerf	9.50	8.90	-0.80

Migration that decreases latency. This experiment inverts the previous: we build the model from traces collected using the multi-site deployment and predict the single-site end-to-end latency distribution by injecting the negative delay value (-38.7ms) to the target nodes. Fig. 14 shows the results; the left-side plot delineates the effects on moving the parallel services (`user-service` and `media-service`), while the right-side plot shows the results of moving serial services (`user-service` and `user-timeline-service`). Once again, the prediction accuracy is very good, with average relative errors for both experiments below 2% and D-statistic less than 5.35%. As a comparison, WebPerf shows errors of more than 9.5%.

4.3.2 Slack Analysis (UC2) LatenSeer can be used to infer the latency budget of specific microservices: it traverses the L-TREE top-down, from the root node to all leaves, to



(a) Latency slack distributions output from LatenSeer **(b) Latency prediction from perturbation guided by latency slack**

Figure 15: Latency slack analysis.

compute the available latency slack for services. Fig. 15a shows four services with latency slack under our experimental workload: `unique-id-service`, `user-service`, and `media-service` have similar latency slack distributions — three of the services exhibited at least 2.3 ms slack, while the latency slack for `user-mention-service` is much smaller. Other services, not shown, had zero latency slack, meaning they were on the latency-critical path, and any increase in latency of these services would affect end-to-end latency.

We evaluate the accuracy of the latency slack identified by LatenSeer at individual services, using the slack to systematically perturb the latency distributions of services on and off the latency-critical path. For these experiments, we use the single-site deployment configuration and perturb latency in a controlled fashion by using the `tc` utility to induce delay at the network level of the Docker container of the target service. The same delay value is injected into the target service in LatenSeer's model, which then updates the latency slack at each node in a top-down fashion.

We evaluate the accuracy of slack estimates in each of the three ways that injected latency can perturb the L-TREE: (i) when the injected latency is off the latency-critical path; (ii) when the injected latency extends or reduces the duration of the latency-critical path; and (iii) when the injected latency changes the latency-critical path itself.

OFF-CP: For the first prediction experiment, we inject 2 ms of latency – well within the slack time – at `user-service` that is off the latency-critical path. Fig. 15b "Off CP" indeed confirms this, with the prediction almost unchanged from ground truth ($D = 4.26\%, E_{max} = -1.0\%, E_{avg} = -0.65\%$).

ON-CP: We then inject 5 ms of latency to `user-timeline` service that is inferred to be on the latency-critical path (zero latency slack). The new dominating latency distribution bubbles up through the levels of the L-TREE to the root node. Fig. 15b "On CP" shows the results with $D = 2.01\%, E_{max} = 0.7\%, E_{avg} = -0.11\%$.

CHANGING-CP: For the third injection experiment, we change the critical path by again adding latency to `user-service`. In this case, however, the injected latency of 5 ms exceeds the available slack, causing the change in end-to-end latency. We show the results in Fig. 15b "Changing CP". Note

Table 3: Results of sensitivity analysis on the injected latency for the experiment shown in Fig. 13a.

Diff from ping time(%)	D(%)	E_{max}(%)	E_{avg}(%)
-10	9.5	2.3	-0.96
-5	7.8	2.3	-0.58
-2	3.5	2.3	-0.36
-1	1.8	2.3	-0.29
+1	2.5	2.3	-0.01
+2	4.6	2.3	-0.06
+5	8.0	4.4	0.16
+10	9.6	8.6	5.40

that this experiment injects the same magnitude of latency change as the previous one (albeit into different microservices), but the resulting distribution is markedly different, and moreover, LatenSeer successfully predicts this difference ($D = 0.02\%, E_{max} = 0.3\%, E_{avg} = 0.1\%$).

4.4 Sensitivity Analysis

We conduct a sensitivity analysis on injected latency and on changes to the workload request mix, showing how the metrics degrade as the model diverges further from conditions experienced in the prediction environment.

Injected latency. In the service migration experiments, we use the average ping time between two sites to approximate the additional latency introduced by the new placement. This is obviously a low-fidelity value – single packet timings at the network layer are not the same as timings of RPC over TCP, across a WAN link, with varying payload size. However, we also claim that this is a realistic starting point in an industry setting, offering a simple-to-obtain, "good enough" value for many real-world scenarios that require only an approximate answer to a what-if question.

To better characterize the impact of such inaccuracy, we repeat the first service placement experiment with various latency injections as fractions of the ping value. Table 3 shows the results: the K-S statistic, D, in particular, reports increasingly large divergences between prediction and ground truth, although overall relative error does not vary much from the baseline we use in the experiments reported above.

Workload mix. LatenSeer predicts latency using the *historical* data. In the real world, it is not unusual for the mix of request types to change over time, and so we look here at how varying the relative request proportions affects the quality of the prediction. In this experiment, we first collect traces in the *single-site* scenario, which comprises a 50:50 mixture of request types `read-home-timeline` and `read-user-timeline`. Then, we migrate `user-timeline-service` to the multi-site scenario and collect ground truth using 90:10 and 20:80 mixtures.

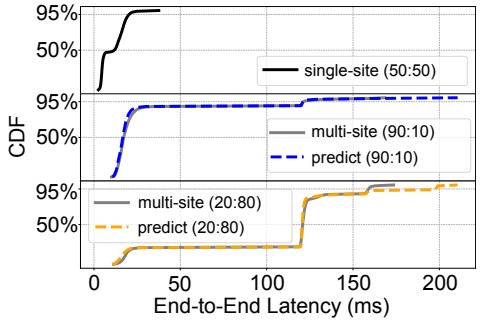


Figure 16: Latency prediction. Sensitivity to different workload distributions.

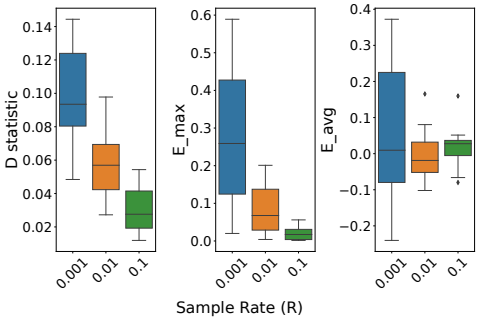


Figure 17: Error analysis. Relative errors of prediction for different sampling rates.

Fig. 16 shows the model's accurate predictions up to approximately the 90th percentile, but results degrade at the tail. This decline is attributed to a higher cache hit ratio during skewed workloads, a detail not covered by LatenSeer's modeling, hence the anticipated inaccuracy.

Threshold for succession time distribution. As discussed in §3.2, we choose the median value from the succession time CDF as our classification for all experiments. Here we repeat the first service placement experiment, adjusting the classification threshold between 1% and 99% to extract causal dependencies. Our results indicate same values for D, $E_{\max}$, and E_{avg} across all classification thresholds. This consistency can be attributed to the relatively low level of noise present within the benchmark environment. However, it's important to recognize that this threshold may not be universally applicable. It is recommended that the threshold be evaluated and potentially adjusted to suit the unique conditions of different production environments.

Varying trace sampling rate. As tracing systems only capture a subset of requests, we evaluate the accuracy of prediction with different sampling rates over 4 sets of migration experiments. We randomly sample the traces that are used to build the model in the previous experiments as *sample* traces, then we use the *sample* traces to build the model and conduct prediction. Unless otherwise stated, we repeat this experiment 20 times for each sampling rate.

The box plots in Fig. 17 show the error metrics (D, $E_{\max}$, and E_{avg}) between prediction from *sample* traces and ground truth for different sample rates. Overall, the D statistic is small, even for low sampling rates. Sampling with R = 0.001 results in E2E latency prediction with D statistic of between 0.05 and 0.14. E_{max} and E_{avg} show more over-prediction when the sampling rate is too low.

4.5 LatenSeer Performance

Building the LatenSeer model, or L-TREE, is dependent on the internal structure of the aggregated trace tree. In controlled experiments, we have found that constructing the LatenSeer from 30,000 DSB traces took approximately 11 minutes. The number of spans in our DSB trace set ranged from 5-33, and the depth extended from 2-6 RPC hops. However, this timing can significantly vary in real-world settings, owing to the complexity of the traces involved.

For instance, the number of spans in our DSB trace set ranged from 5-33, and the depth extended from 2-6 RPC hops. This complexity is dwarfed when we consider Alibaba and Twitter traces. Alibaba traces can contain up to 6,625 spans with depths between 1-14 hops. In contrast, Twitter traces can encompass up to 25,000 spans and depths ranging from 2-18 hops. Given these complexities, the time to build the L-TREE in real-world scenarios can be considerably longer. For 71,055 Alibaba traces, the model-building process requires approximately 66 minutes. Similarly, for 2,618 traces from Twitter, the construction process takes about an hour.

5 Conclusion

We present LatenSeer, a modeling framework for estimating end-to-end latency distributions in microservice-based web applications. LatenSeer can accurately predict interventional end-to-end latency by leveraging distributed tracing data. We evaluated LatenSeer in two realistic scenarios: service placement and latency slack analysis. Our evaluation shows that LatenSeer achieves high precision accuracy with an estimation error less than 5.35% (D-statistic), outperforming the start-of-the-art that has more than 9.5% estimation error. Moreover, our results on real-world production traces show that LatenSeer is practical and scalable enough to support complexities in production environments.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback and our shepherd Vivien Quéma for constructive suggestions. We are grateful to the organizations that have generously open-sourced and shared production traces. We thank CloudLab [15] for the infrastructure support for running experiments. We also appreciate the members of SimBioSys for their interest, insights, feedback, and support.

References

[1] Akamai Technologies. 2017. Akamai Online Retail Performance Report: Milliseconds Are Critical. Retrieved April 2022 from https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report.

[2] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. 2017. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*. USENIX Association, USA, 469–482.

[3] Mohammad Alrifai and Thomas Risse. 2009. Combining global optimization with local selection for efficient QoS-aware service composition. In *Proceedings of the 18th international conference on World wide web*. 881–890.

[4] Mohammad Alrifai, Dimitrios Skoutas, and Thomas Risse. 2010. Selecting skyline services for QoS-based web service composition. In *Proceedings of the 19th international conference on World wide web*. 11–20.

[5] Danilo Ardagna and Barbara Pernici. 2007. Adaptive service composition in flexible processes. *IEEE Transactions on software engineering* 33, 6 (2007), 369–384.

[6] Emre Ates, Lily Sturmann, Mert Toslali, Orran Krieger, Richard Megginson, Ayse K Coskun, and Raja R Sambasivan. 2019. An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*. Association for Computing Machinery, New York, NY, USA, 165–170.

[7] Paul Barford and Mark Crovella. 2000. Critical path analysis of TCP transactions. *ACM SIGCOMM Computer Communication Review* 30, 4 (2000), 127–138.

[8] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. 2004. Using Magpie for request extraction and workload modelling. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, Vol. 4. Association for Computing Machinery, New York, NY, USA, 18–33.

[9] David Böhme, Markus Geimer, Lukas Arnold, Felix Voigtlaender, and Felix Wolf. 2016. Identifying the root causes of wait states in large-scale parallel applications. *ACM Transactions on Parallel Computing (TOPC)* 3, 2 (2016), 1–24.

[10] Laura Carnevali, Riccardo Reali, and Enrico Vicario. 2021. Compositional evaluation of stochastic workflows for response time analysis of composite web services. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*. 177–188.

[11] Mike Y Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, Dave Patterson, Armando Fox, and Eric Brewer. 2004. Path-based failure and evolution management. In *1st USENIX Symposium on Networked Systems Design & Implementation (NSDI'04)*. USENIX Association, USA, 23–23.

[12] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI'14)*. USENIX Association, USA, 217–231.

[13] Jeremy Cloud. 2013. Decomposing Twitter: Adventures in Service-Oriented Architecture. https://www.infoq.com/presentations/twitter-soa/. Accessed: 2023-05-21.

[14] Vadim Denisov, Dirk Fahland, and Wil MP van der Aalst. 2019. Predictive performance monitoring of material handling systems using the performance spectrum. In *2019 International Conference on Process Mining (ICPM)*. IEEE, 137–144.

[15] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. 2019. The design and operation of CloudLab. In *2019 USENIX Annual Technical Conference (USENIX ATC'19)*. Association for Computing Machinery, New York, NY, USA, 1–14.

[16] Brian Eaton, Jeff Stewart, Jon Tedesco, and N Cihan Tas. 2022. Distributed Latency Profiling through Critical Path Tracing: CPT can provide actionable and precise latency analysis. *Queue* 20, 1 (2022), 40–79.

[17] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: Predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference*. 248–259.

[18] Simon Eismann, Johannes Grohmann, Erwin Van Eyk, Nikolas Herbst, and Samuel Kounev. 2020. Predicting the costs of serverless workflows. In *Proceedings of the ACM/SPEC international conference on performance engineering*. 265–276.

[19] Joyce El Hadad, Maude Manouvrier, and Marta Rukoz. 2010. TQoS: Transactional and QoS-aware selection algorithm for automatic Web service composition. *IEEE Transactions on Services Computing* 3, 1 (2010), 73–85.

[20] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. 2012. Fay: Extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)* 30, 4 (2012), 1–35.

[21] Brian Fields, Shai Rubin, and Rastislav Bodik. 2001. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th annual international symposium on Computer architecture*. Association for Computing Machinery, New York, NY, USA, 74–85.

[22] Tobias Flach, Nandita Dukkipati, Andreas Terzis, Barath Raghavan, Neal Cardwell, Yuchung Cheng, Ankur Jain, Shuai Hao, Ethan Katz-Bassett, and Ramesh Govindan. 2013. Reducing Web Latency: The Virtue of Gentle Aggression. In *Proceedings of the 2013 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM'13)* (Hong Kong, China) *(SIGCOMM '13)*. ACM, New York, NY, USA, 159–170. https://doi.org/10.1145/2486001.2486014

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

[24] Nicole Forsgren, Dustin Smith, Jez Humble, and Jessie Frazelle. 2019. *2019 Accelerate State of DevOps Report*. Technical Report. Google.Inc. http://cloud.google.com/devops/state-of-devops/

[25] Thomas R Frieden. 2017. Evidence for health decision making—beyond randomized, controlled trials. *New England Journal of Medicine* 377, 5 (2017), 465–475.

[26] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. 2019. 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 (ASPLOS'19)*. Association for Computing Machinery, New York, NY, USA, 3–18.

[27] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'19)*. Association for Computing Machinery, New York, NY, USA, 19–33.

[28] Fănică Gavril. 1972. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set

of a chordal graph. *SIAM J. Comput.* 1, 2 (1972), 180–187.

[29] Adam Gluck. 2020. Introducing Domain-Oriented Microservice Architecture. https://www.uber.com/blog/microservice-architecture/. Accessed: 2023-05-21.

[30] Herodotos Herodotou and Shivnath Babu. 2011. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *Proceedings of the VLDB Endowment* 4, 11 (2011), 1111–1122.

[31] Lexiang Huang and Timothy Zhu. 2021. tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces. In *Proceedings of the ACM Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 76–91.

[32] Jaeger 2023. Jaeger: Open Source, End-to-End Distributed Tracing. https://www.jaegertracing.io/. Accessed: 2023-05-21.

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

[34] Diviyan Kalainathan, Olivier Goudet, Isabelle Guyon, David Lopez-Paz, and MichÃ¨le Sebag. 2022. Structural Agnostic Modeling: Adversarial Learning of Causal Graphs. *Journal of Machine Learning Research* 23, 219 (2022), 1–62. http://jmlr.org/papers/v23/19-529.html

[35] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th symposium on operating systems principles*. Association for Computing Machinery, New York, NY, USA, 34–50.

[36] Kyle Kingsbury. 2013. The trouble with timestamps. https://aphyr.com/posts/299-the-trouble-with-timestamps. Accessed: 2023-05-30.

[37] Darja Krushevskaja and Mark Sandler. 2013. Understanding latency variations of black box services. In *Proceedings of the 22nd International Conference on the World Wide Web (WWW'13)*. Association for Computing Machinery, New York, NY, USA, 703–714.

[38] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing* (Santa Cruz, CA, USA) *(SoCC '19)*. Association for Computing Machinery, New York, NY, USA, 312–324. https://doi.org/10.1145/3357223.3362736

[39] Pedro Las-Casas, Giorgi Papakerashvili, Vaastav Anand, and Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *Proceedings of the ACM Symposium on Cloud Computing (SOCC'19)*. 312–324.

[40] Cate Lawrence. 2021. Deployment Frequency – A Key Metric in DevOps. https://humanitec.com/blog/deployment-frequency-key-metric-in-devops. Accessed: 2023-05-30.

[41] Yuliang Li, Gautam Kumar, Hema Hariharan, Hassan Wassel, Peter Hochschild, Dave Platt, Simon Sabato, Minlan Yu, Nandita Dukkipati, Prashant Chandra, et al. 2020. Sundial: Fault-tolerant clock synchronization for datacenters. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, USA, 1171–1186.

[42] Chieh-Jan Mike Liang, Zilin Fang, Yuqing Xie, Fan Yang, Zhao Lucis Li, Li Lyna Zhang, Mao Yang, and Lidong Zhou. 2023. On Modular Learning of Distributed Systems for Predicting {End-to-End} Latency. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. usenix, USA, 1081–1095.

[43] Lightstep. 2020. OpenTelemetry: Best Practices on Sampling. Retrieved December 2020 from https://opentelemetry.lightstep.com/best-practices/sampling/.

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

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

[46] Shutian Luo, Huanle Xu, Kejiang Ye, Guoyao Xu, Liping Zhang, Guodong Yang, and Chengzhong Xu. 2022. The power of prediction: microservice auto scaling via workload learning. In *Proceedings of the 13th Symposium on Cloud Computing*. Association for Computing Machinery, New York, NY, USA, 355–369.

[47] Jonathan Mace, Peter Bodik, Rodrigo Fonseca, and Madanlal Musuvathi. 2015. Retro: Targeted Resource Management in Multi-Tenant Distributed Systems. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. usenix, USA.

[48] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP'15)*. Association for Computing Machinery, New York, NY, USA.

[49] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Sameh Elnikety, Somali Chaterji, and Saurabh Bagchi. 2022. ORION and the Three Rights: Sizing, Bundling, and Prewarming for Serverless DAGs. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 303–320. https://www.usenix.org/conference/osdi22/presentation/mahgoub

[50] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–28.

[51] Ali Najafi and Michael Wei. 2022. Graham: Synchronizing Clocks by Leveraging Local Clock Properties. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. USENIX Association, USA, 453–466.

[52] Oleg Obleukhov. 2020. Building a more accurate time service at Facebook scale. https://engineering.fb.com/2020/03/18/production-engineering/ntp-service/. Accessed: 2023-05-30.

[53] OpenTelemetry 2023. OpenTelemetry: An Observability Framework for Cloud-Native Software. http://opentelemetry.io/. Accessed: 2023-05-21.

[54] OpenTracing 2023. OpenTracing: Vendor-Neutral APIs and Instrumentation for Distributed Tracing. http://opentracing.io/. Accessed: 2023-05-21.

[55] Krzysztof Ostrowski, Gideon Mann, and Mark Sandler. 2011. Diagnosing latency in multi-tier black-box services. In *4th International Workshop on Large-Scale Distributed Systems and Middleware (LADIS'11)*. USA.

[56] Maulik Pandey. 2019. Building Netflix's Distributed Tracing Infrastructure. Retrieved December 2022 from https://netflixtechblog.com/building-netflixs-distributed-tracing-infrastructure-bb856c319304.

[57] Austin Parker, Daniel Spoonhower, Jonathan Mace, Ben Sigelman, and Rebecca Isaacs. 2020. *Distributed Tracing in Practice: Instrumenting, Analyzing, and Debugging Microservices*. O'Reilly Media, USA.

[58] Judea Pearl. 2009. Causal inference in statistics: An overview. *Statistics surveys* 3 (2009), 96–146.

[59] Judea Pearl. 2019. The seven tools of causal inference, with reflections on machine learning. *Commun. ACM* 62, 3 (2019), 54–60.

[60] Judea Pearl and Dana Mackenzie. 2018. *The Book of Why: the new science of cause and effect.* Basic Books, USA.

[61] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, USA, 805–825. https://www.usenix.org/conference/osdi20/presentation/qiu

[62] Joy Rahman and Palden Lama. 2019. Predicting the end-to-end tail latency of containerized microservices in the cloud. In *2019 IEEE International Conference on Cloud Engineering (IC2E)*. IEEE, IEEE, USA, 200–210.

[63] Lenin Ravindranath, Jitendra Padhye, Sharad Agarwal, Ratul Mahajan, Ian Obermiller, and Shahin Shayandeh. 2012. Appinsight: Mobile app performance monitoring in the wild. In *Presented as part of the 10th {USENIX} symposium on operating systems design and implementation ({OSDI} 12)*. USENIX Association, USA, 107–120.

[64] Lenin Ravindranath, Jitendra Padhye, Ratul Mahajan, and Hari Balakrishnan. 2013. Timecard: Controlling user-perceived delays in server-based mobile applications. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. Association for Computing Machinery, New York, NY, USA, 85–100.

[65] Patrick Reynolds, Charles Edwin Killian, Janet L Wiener, Jeffrey C Mogul, Mehul A Shah, and Amin Vahdat. 2006. Pip: Detecting the Unexpected in Distributed Systems.. In *USENIX Symposium on the Networked Systems Design and Implemntation (NSDI'06)*, Vol. 6. USENIX association, USA, 9–9.

[66] Andreas Rogge-Solti, Wil MP van der Aalst, and Mathias Weske. 2014. Discovering stochastic petri nets with arbitrary delay distributions from event logs. In *Business Process Management Workshops: BPM 2013 International Workshops, Beijing, China, August 26, 2013, Revised Papers 11*. Springer, 15–27.

[67] Cynthia Rudin. 2019. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence* 1, 5 (2019), 206–215.

[68] Ruslan Meshenberg, Josh Evan. 2015. Netflix at AWS re:Invent 2015. https://netflixtechblog.com/netflix-at-aws-re-invent-2015-2bc50551dead. Accessed: 2023-05-30.

[69] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*.

[70] Justin Sheehy. 2015. There is No Now. *Commun. ACM* 58, 5 (apr 2015), 36–41. https://doi.org/10.1145/2733108

[71] Yuri Shkoro. 2019. Conquering Microservices Complexity at Uber with Distributed Tracing (Presentation). Received June 2021 from https://www.infoq.com/presentations/uber-microservices-distributed-tracing/.

[72] Yuri Shkuro. 2019. *Mastering Distributed Tracing*. Packt Publishing, USA.

[73] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc. https://research.google.com/archive/papers/dapper-2010-1.pdf

[74] Benjamin Speich, Belinda von Niederhäusern, Nadine Schur, Lars G Hemkens, Thomas Fürst, Neera Bhatnagar, Reem Alturki, Arnav Agarwal, Benjamin Kasenda, Christiane Pauli-Magnus, et al. 2018. Systematic review on costs and resource use of randomized clinical trials shows a lack of transparent and comprehensive data. *Journal of clinical epidemiology* 96 (2018), 1–11.

[75] Mukarram Tariq, Amgad Zeitoun, Vytautas Valancius, Nick Feamster, and Mostafa Ammar. 2008. Answering what-if deployment and configuration questions with wise. In *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*. Association for Computing Machinery, New York, NY, USA, 99–110.

[76] Parth Thakkar, Rohan Saxena, and Venkata N Padmanabhan. 2021. AutoSens: inferring latency sensitivity of user activity through natural experiments. In *Proceedings of the 21st ACM Internet Measurement Conference (IMC'21)*. Association for Computing Machinery, New York, NY, USA, 15–21.

[77] Sudhir Tonse. 2015. Scalable Microservices at Netflix. Challenges and Tools of the Trade. https://www.infoq.com/presentations/netflix-ipc/. Accessed: 2023-05-21.

[78] Mert Toslali, Emre Ates, Alex Ellis, Zhaoqi Zhang, Darby Huye, Lan Liu, Samantha Puterman, Ayse K. Coskun, and Raja R. Sambasivan. 2021. Automating instrumentation choices for performance problems in distributed applications with VAIF. In *SoCC '21: Proceedings of the Twelfth Symposium on Cloud Computing*.

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

[80] Kaushik Veeraraghavan, Justin Meza, Scott Michelson, Sankaralingam Panneerselvam, Alex Gyori, David Chou, Sonia Margulis, Daniel Obenshain, Shruti Padmanabha, Ashish Shah, et al. 2018. Maelstrom: Mitigating datacenter-level disasters by draining interdependent traffic safely and efficiently. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*. USENIX association, USA, 373–389.

[81] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th {USENIX} symposium on networked systems design and implementation ({NSDI} 16)*. USENIX Association, USA, 363–378.

[82] Alec Warner and Štěpán Davidovič. 2018. The Site Reliability Engineering Workbook Chapter: Canarying Releases. (2018).

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

[84] C-Q Yang and Barton P Miller. 1988. Critical path analysis for the execution of parallel and distributed programs. In *The 8th International Conference on Distributed*. IEEE Computer Society, IEEE, USA, 366–367.

[85] Tao Yu, Yue Zhang, and Kwei-Jay Lin. 2007. Efficient algorithms for web services selection with end-to-end qos constraints. *ACM Transactions on the Web (TWEB)* 1, 1 (2007), 6–es.

[86] Lei Zhang, Vaastav Anand, Zhiqiang Xie, Ymir Vigfusson, and Jonathan Mace. 2023. The Benefit of Hindsight: Tracing Edge-Cases in Distributed Systems. In *NSDI'23: Proceedings of the 20th USENIX Conference on Networked Systems Design and Implementation*. USENIX association, USA, 321–339.

[87] Yilei Zhang, Zibin Zheng, and Michael R Lyu. 2011. WSPred: A time-aware personalized QoS prediction framework for Web services. In *2011 IEEE 22nd international symposium on software reliability engineering*. IEEE, 210–219.

[88] Zhizhou Zhang, Murali Krishna Ramanathan, Prithvi Raj, Abhishek Parwal, Timothy Sherwood, and Milind Chabbi. 2022. CRISP: Critical Path Analysis of Large-Scale Microservice Architectures. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX association, USA, 655–672.

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

[90] Zipkin 2023. Zipkin: A Distributed Tracing System. http://zipkin.io/. Accessed: 2023-05-21.