usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

CRISP: Critical Path Analysis of Large-Scale Microservice Architectures

Zhizhou Zhang, *UC Santa Barbara;* Murali Krishna Ramanathan, Prithvi Raj, and Abhishek Parwal, *Uber Technologies Inc.;* Timothy Sherwood, *UC Santa Barbara;* Milind Chabbi, *Uber Technologies Inc.*

https://www.usenix.org/conference/atc22/presentation/zhang-zhizhou

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

NetApp®

CRISP: Critical Path Analysis of Large-Scale Microservice Architectures

Zhizhou Zhang
University of California, Santa Barbara

Murali Krishna Ramanathan
Uber Technologies

Prithvi Raj
Uber Technologies

Abhishek Parwal
Uber Technologies

Timothy Sherwood
University of California, Santa Barbara

Milind Chabbi
Uber Technologies

Abstract

Microservice architectures have become the lifeblood of modern service-oriented software systems. Remote Procedure Calls (RPCs) among microservices are deeply nested, asynchronous, and large in number, thus making it very hard to identify the underlying service(s) that contribute to the overall end-to-end latency experienced by a top-level request. State-of-the-art RPC tracing tools collect a deluge of data but provide little insight. We need sophisticated tools to bubble-up signals from a myriad of RPC traces to assist developers in identifying optimization opportunities, pinpointing common bottlenecks, setting appropriate time outs, diagnosing error conditions, and planning and managing compute capacity, to name a few.

In this paper, we present CRISP — a tool to perform critical path analysis (CPA) over a large number of traces collected from RPCs in microservices environments. CRISP provides three critical performance analysis capabilities: a) a *top-down* CPA of any specific endpoint, which is tailored for service owners to drill down the root causes of latency issues, b) a *bottom-up* CPA over all endpoints in the system — tailored for infrastructure and performance engineers — to bubble up those (interior) APIs that have a high impact across many endpoints, and c) an on-the-fly anomaly detection to alert potential problems.

We have applied CRISP's capabilities on Uber's entire backend system composed of ∼40K endpoints that cater to real-time requests from more than 100 million active daily users worldwide. Using the critical path as the basis of performance analysis has a) helped us identify five performance issues and optimization opportunities across two business-critical microservices, b) guided us in our future hardware choice that reduces end-to-end latencies, and c) reduced the false positives in anomaly detection by up to 50% while speeding up the training and inference by up to 28× and up to 67×, respectively, over the state of the art.

1 Introduction

Microservice architectures [23, 27, 28, 36, 43, 45, 56] have become the lifeblood of modern service-oriented software systems. As opposed to monolithic software development and deployment, in a microservice environment, the business logic is broken into individually deployable programs, which allow fast development and scalable deployment. Individual microservice *instances* interact with one another via remote procedure calls (RPCs). As microservices evolve with the business, they grow in number and their interactions become complex.

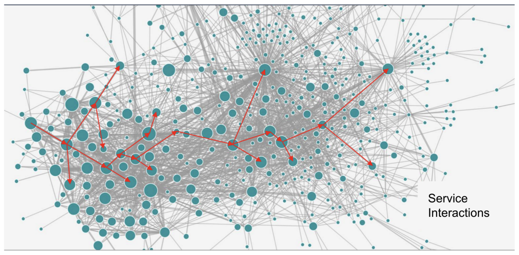


Figure 1: Complex microservice RPC call graph at Uber collected via Jaeger tracing.

Uber's backend is an exemplar of microservice architecture. Uber has ∼4,000 microservices interacting with each other via RPCs. Each microservice hosts a handful of APIs, leading to a total of about 40,000 unique RPC endpoints that can call one another in complex ways, as depicted in Figure 1. Hereafter, we use the terms *endpoint* and *API* interchangeably to mean a uniquely named functionality provided by a service. We use the terms *operation* and *RPC* interchangeably to mean an instance of invocation of such an API.

A service request arriving at an entry point API to the Uber backend systems undergoes multiple "hops" through numerous microservice RPCs before being fully serviced. The life of a request results in intricate microservice interactions. These interactions are deeply nested, asynchronous, and invoke numerous other downstream APIs. As a result of this complexity, it is very hard to identify which underlying service(s) contribute to the overall end-to-end latency [21, 32, 38, 44, 52, 53, 63] experienced by a top-level request. Answering this question is critical in many situations. For example:

- Identifying optimization opportunities for a top-level microservice (e.g., reducing tail latency)

- Identifying bottleneck APIs that affect numerous endpoints
- Setting appropriate time-to-live values for RPCs
- Diagnosing outages and error conditions
- Planning for computing and other capacity management

The critical path [59] is the longest chain of dependent tasks in a microservice dependency graph. Reducing the critical path length is necessary to reduce the end-to-end latency of a request. Hence, latency optimization efforts benefit from prioritizing the services that are on the critical path.

We have developed a tool, CRISP [1], to pinpoint and quantify performance problems in microservice architectures. CRISP uses the RPC tracing facility provided by Jaeger [6] and constructs the critical path through a request's graph of dependencies. The critical path may vary among requests; hence, CRISP computes the critical path per request. It then aggregates and summarizes critical paths from millions of requests. Finally, it presents them as digestible and actionable insights via rich heat maps [5] and flame graphs [31]. CRISP provides knobs to dissect the details with different percentile values that help in performance diagnoses.

As a full-fledged performance analysis tool, CRISP caters to various use cases via the following rich set of capabilities that scale to work on millions of traces:

- **Top-down analysis**: A top-down analysis of any specific endpoint of interest. This capability allows service owners to deep dive into their specific endpoint and pinpoints and quantifies bottlenecks encountered in the RPC dependency graph. Improving these bottlenecks should be the first-order priority to reduce the latency of the endpoint.
- **Bottom-up analysis**: A bottom-up analysis over all endpoints, which bubbles up and ranks by the impact of those interior APIs that cause the most latency across most endpoints. Optimizing these interior APIs reduces latency across numerous endpoints.
- **Neural network-based anomaly detection**: An automated anomaly detection system, which detects whether a request is exhibiting abnormal behavior compared with the past history of the endpoint. The system is trained per endpoint using an autoencoder-decoder machine learning technique [39]. This system is set up to expedite problem detection and alert developers. Basing the abnormality detection on the divergence in the critical path as opposed to the full call graph [39] not only makes the training and inference faster but also reduces false alerts.

Practical deployment of CRISP at Uber over a three-month period working on 40K endpoints while processing $\sim 200GB$ of traces with $\sim$18 million spans in $\sim$256 hours of CPU time per day has resulted in the following impact:

- Detection and narrowing down the causes of five latency impacting bugs in two business-critical services
- Identification of a 1.5$\times$ tail latency lengthening due to hardware choice and the resulting guidance for future hardware selection
- Up to 27.77$\times$ speedup in training, up to 66.85$\times$ speed up in inference, and 50% reduction in false alerts in identifying abnormality of service behaviors over the state of the art [39]

[1]named taking letters from **cri**tical and **sp**an

The rest of this paper is organized as follows: Section 2 motivates CRISP with a use case at Uber, Section 3 describes the Jaeger tracing framework, Section 4-6 describe the methodology, internals, and features of CRISP, Section 7 evaluates CRISP at Uber, Section 8 discusses the related work, and Section 9 offers our conclusions.

2 Motivating Example for CRISP

Fulfillment [8] at Uber is a platform to orchestrate and manage the lifecycle of orders and user sessions with millions of active participants. The Fulfillment platform is a foundational Uber capability that enables the rapid scaling of new verticals. The platform handles more than a million concurrent users and billions of trips per year that span over ten thousand cities. The platform handles billions of database transactions a day. Hundreds of Uber microservices rely on the platform as the source of truth for the accurate state of the trips and driver or delivery sessions. Events generated by the platform are used to build hundreds of offline datasets to make critical business decisions. Over 500 developers extend the platform using APIs, events, and code to build more than 120 unique fulfillment flows.

The `createOrder` endpoint allows capturing the requester's intent in the Uber backend. Intent can be to request a ride from one of the ridesharing lines of products, food booked and dispatched by one of the courier partners, or a package be delivered to a customer. This endpoint has a complex task dependency graph necessary for: a) determining order risk such as user fraud, sufficient user balance via authentication hold, b) ensuring the fare presented to the requester in the shopping phase is still valid, c) determining the benefits the requester is eligible for, d) enriching data with location information, and e) creating an order in the backend to start the matching process.

The tasks in this endpoint have grown organically as requirements evolved. This has led to an increase in p95 latency to 6 seconds, affecting user experience. The service itself is written in Java, and highly (both macro and micro) optimized using periodic profiling. However, the profiling offered no insights into downstream calls, where most time is spent. Quantitative insight into the causes of the latency was hard to analyze by looking at individual traces because each trace contains thousands of nested and overlapping RPCs.

There are numerous sampling- and instrumentation-based profilers [10, 11, 13, 29] for intra-service profiling. However, they do not collect metrics at the individual request level. The Fulfillment microservice (as most other microservices) is highly threaded; the work of an individual request may be partitioned among multiple threads within a process as well

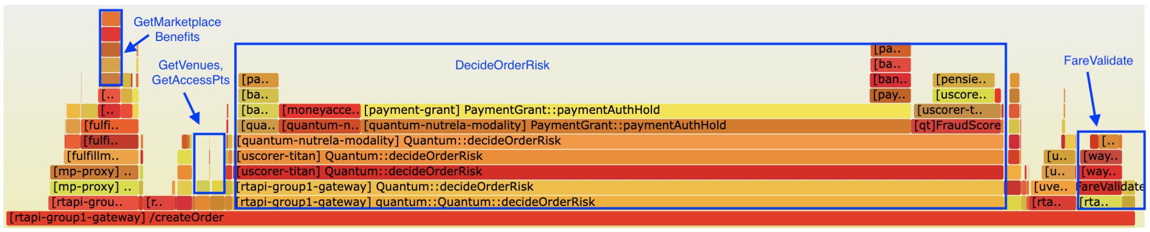


Figure 2: Critical path(s) of `createOrder` endpoint shown as a flame graph via CRISP after processing 100K Jaeger traces.

as multiple threads may be handling independent requests simultaneously. In such a setup, traditional profilers fail to highlight the causes of latencies incurred at an individual request level. Also, traditional profilers fail to capture IO waiting, task dependencies, and serialization patterns.

With CRISP, the development team performed a top-down critical path analysis of this endpoint over 100K traces (∼200GB of traces) and visualized the results as a flame graph as shown in Figure 2. Navigating the "hot" critical paths via the flame graph not only corroborated an existing hunch while offering quantitative guidance but also shed light on new optimization opportunities lurking in the wild. Below, we enumerate a few defects and optimization opportunities that became evident by inspecting CRISP-provided insights.

Async flow optimization: `decideOrderRisk` contributes to about 68% of the end-to-end P50 latency, revealing the following optimizations: a) aggressively use cache in `FraudScore` to reduce its latency and b) parallelize the calls beneath this big endpoint (e.g., `PaymentAuthHold` and `FraudScore`). In the long term, the team envisions using an asynchronous invocation of `paymentAuthHold` and using notification to the requester when a provider is assigned.

Unnecessary API serialization: There was an unnecessary serialization between `GetVenues` and `GetAccessPts`. These two RPCs can be done in parallel.

Avoidable server roundtrip for validation: `FareValidate` contributes to about 5% of the end-to-end P50 latency. This is a call that need not be performed every time. Trusted edge devices (e.g., company mobile app) can validate at the edge improving performance for trusted users and falling back to server validation if the fare has expired based on fare expiry TTL; untrusted apps will use the full server validation.

Caching over DB fetch: `GetMarketplaceBenefits` contributes to about 5% of the P50 latency. This can be served via a cache rather than a database read to fetch requester benefits.

3 Background

In this section, we first describe the microservice tracing infrastructure at Uber and then enumerate its shortcomings.

3.1 Distributed Tracing at Uber

Microservices run over several physical hosts, usually owned by multiple teams, and written in multiple languages. It is impossible to use traditional profilers [7, 13, 29] to gain insight into the events involved in processing a request. Because each physical host can have a separate clock, it is intractable to infer causality using time alone. Distributed tracing [47] encodes causality information in a distributed context, which is propagated across process boundaries. It provides a way to infer states across various services for the lifetime of a request.

At Uber, Jaeger [6] is used as the distributed tracing system. Jaeger provides clients for generating trace data and components for storage and retrieval of traces. Microservices instrumented with Jaeger clients produce OpenTracing [6] -compliant spans when receiving new requests and attach distributed context information (trace ID, span ID, custom key-value pairs). The "span" [46] is the primary building block of a distributed trace, which represents a serial unit of work done in a distributed system. Each span contains the following information:

- API name
- Start and finish timestamps
- Custom key-value pairs
- Span context and references (described below)

Each span may reference other spans with a causal relationship by span context. A span may reference a parent with the `ChildOf` relationship, indicating that the parent span waits for the child to finish a certain task. Multiple child spans can be referenced by the same parent and run concurrently.

While the source code is always instrumented, the overhead is controlled by a dynamic sampling rate, which is adjusted based on the traffic received by Internet-facing endpoints. No data is collected for traces that are not sampled. Specifically, adaptive sampling sets a target QPS for traces on a per root service-endpoint basis, which ensures that the number of samples on the external API request remains roughly constant. Jaeger does not support tail-based sampling [12].

Figure 3 depicts the Jaeger deployment at Uber. Jaeger is deployed as multiple components, with a `jaeger-agent` running on every host. All applications running on this host send spans to `jaeger-agent` over UDP [9]. `jaeger-agent` then forwards these spans to a `jaeger-collector`, which then buffers spans onto the Kafka [37] distributed event

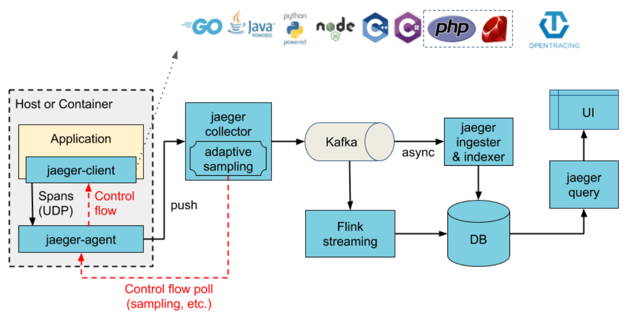


Figure 3: Jaeger deployment at Uber.

streaming platform. The spans buffered in Kafka have multiple consumers: `jaeger-ingester`, which inserts them into Docstore [3], a distributed SQL database, and allows for retrieving full traces; `jaeger-indexer`, which inserts them into Sawmill [4], a schema-agnostic logging platform that allows user-friendly search on spans fields. Additionally, spans are consumed by Apache Flink [15] jobs to produce multi-hop dependency graphs. Depending on the sampling configuration in effect, the backend processes around 400K-1M spans per second, which is approximately 20TB each day. Variance is common due to diurnal patterns.

3.2 Difficulties with Large-Scale Jaeger Traces

Despite their power, Jaeger traces are highly complicated. Jaeger provides a UI to filter traces by time ranges and also provides a UI to view the trace as a callgraph, as well as an expandable tree over a timeline. In spite of these facilities, the users of this manual workflow often complained about the following limitations to analyze endpoint latencies:

- Only first-level insights are possible from drilling down into microservice latencies and errors.
- Using a few Jaeger traces is insufficient to reach a reliable conclusion. Users can visualize and navigate only one Jaeger trace at a time. There is no aggregate summary of traces.
- A single Jaeger trace can be so complex that it is not humanly possible to browse and understand the details. Endpoints commonly have thousands of nodes in the RPC graph with 25-deep call chains and up to 40 spans overlapping in time. It is cumbersome to manually understand the critical path due to the asynchronous nature of calls.
- There is a lack of regular, performance-driven feedback tooling to optimize the workflow or downstream systems.

These challenges introduce a barrier to our developers in effectively using Jaeger to either detect anomalous situations or identify optimization opportunities.

4 CRISP Methodology

The fundamental difficulty in making sense out of a Jaeger trace is due to the complexity of the graph. Our premise is that while the whole graph is interesting in terms of data richness, it brings a lot of noise. There are many RPCs and call paths that are insignificant for a high-level analysis and optimization task. With this understanding, we shrink the graph to its quintessential element—the *critical path*—and aggregate many traces into a single summary that is still rich with call path information.

Critical Path Analysis [25, 59] (CPA) is a well-studied concept over directed acyclic graphs (DAG) formed out of computing graphs in parallel computing. The nodes in the DAG represent tasks (units of serial execution) and the edges represent dependencies between tasks. A node with an out-degree greater than one "spawns" children's tasks and a node with an in-degree greater than one waits ("syncs") for the children to finish. Total work is the sum of weights of all nodes and the critical path is the longest weighted path in the DAG.

Definition 1 (Critical Path)**.** In a task graph $G = (V, E)$ made of task vertices V and their dependency edges E, with two special vertices S (start node) and F (finish node), the critical path is a maximal-weight path from S to F. G may contain more than one critical path.

The critical path identifies the sequence of dependent computations that consume the most time. To speed up the service, it is strictly necessary to boost the components on the critical path.

RPCs among microservice operations have a parent-child hierarchical relationship and can be construed as a parallel computation DAG. The deriving critical path from Jaeger traces, however, has the following challenges:

- Unlike a traditional parallel computing DAG seen in the academic literature, the Jaeger traces do not provide clear "spawn" and "sync" events in the DAG.
- The parent spans in Jaeger traces carry no dependence information and so the information of the last "sync" child span is not directly available.
- In order to obtain the last "sync" child span, clock information is needed. However, the clocks on different machines where spans are collected are not time-synchronized.
- The critical path across all requests may not be unique. Services have diurnal patterns and different traces may exhibit different critical paths, which need to be aggregated, and yet "hot" critical paths need to be bubbled up.
- Since the service codes keep evolving, the critical path keeps changing.

We address these challenges in the next section.

We also mention in passing that the CPA is not a performance analysis panacea. Once the exposed latency on the critical path is eliminated, a new critical path may emerge which necessitates the need for an iterative profiling and optimization approach.

5 Critical Path Analysis

In this section, we detail how we compute, aggregate, and represent critical paths from many Jaeger traces for a given endpoint.

5.1 Deriving Critical Path from a Single Trace

CRISP's trace analysis exploits a map-reduce paradigm to process millions of traces belonging to each endpoint. To this end, each process loads an input Jaeger trace file (JSON format) and builds an n-ary tree, where each parent node is the RPC caller and the children nodes are the immediate downstream callees.

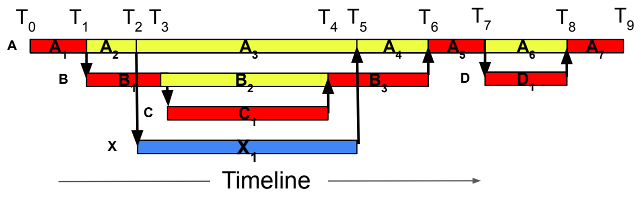
In order to compute the critical path through the trace, we need a computational DAG. To accomplish this under Jaeger/Opentrace trace format, we make use of the start and end times of children's spans. The start time in every immediate child creates a "spawn" event in the parent and splits its span at that point in time. Similarly, the end time in every immediate child creates a "sync" event in the parent and splits its span at the point in time. Thus, we transform the tree into a logical DAG for critical path construction.

Figure 4 shows an example DAG constructed from Jaeger traces by looking at span start and end times. In Figure 4, the span A is the root span, which invokes spans B, X, and D. The span B in turn invokes span C. The start time T_1 and its end-time T_6 of B create a spawn and sync points on A, respectively. Similarly, the spans X and D, create further segments in A. Similarly, B's child C, creates the spawn and sync points on B at T_3 and T_4, respectively.

Limitations of Jaeger/Opentrace format: One key limitation of the Jaeger is that the parent spans (a.k.a., `caller`) do not contain dependence information. Specifically, they lack the information of both start and end of `callee` RPC. Instead, it is the `callee` that stores both the ID of its parent and `callee`'s start and end time (per callee's local clock) in its own span. The implication of the constraint is that the dependency relationship needs to be inferred via clock information recorded in the callee span.

In addition, the inference can be inaccurate because of the clock skew that will be discussed in Section 5.2. Traditionally, the computation of the critical path depends on the last returned child of the parent spans [24]. In Jaeger traces, the last arriving child information is not directly recorded in the parent span. Instead, the last arriver needs to be inferred using the span end time for each child, which will be based on each child's local clock. Without correctly handling the clock skew, the critical path analysis can go wrong.

One may extend Jaeger tracing by making the callee return additional data to the caller. Unfortunately, ensuring that these changes are adopted universally across thousands of services is an engineering hurdle. Such changes also require support from different RPC libraries used by our system. Our solution, in contrast, does not require such large-scale system-wide



T_0 T_1 T_2 T_3 T_4 T_5 T_6 T_7 T_8 T_9
A
A_1 A_2 A_3 A_4 A_5 A_6 A_7
B
B_1 B_2 B_3
D
D_1
C
C_1
X
X_1
Timeline

Figure 4: Trace with root span A, its children B, X, and D. B further calls C. CRISP further segments each parent traces based on the start and end time of its children. The red-colored blocks represent the critical path through the trace.

```
def CP(root):
  path = [root]
  if len(root.child) == 0:
    return path
  children = sortDescendingByEndTime(root.children)
  lfc = children[0]
  path.extend(CP(lfc))
  for c in children[1:]:
    if happensBefore(c, lfc):
      path.extend(CP(c))
      lfc = c
  return path
```

Listing 1: Pseudocode to compute critical path.

changes but yet produces high quality results as we describe in the rest of this section.

5.1.1 Critical Path Algorithm

We, first, describe how we compute the critical path in a trace assuming perfectly synchronized clocks in this subsection. We expand to handle unsynchronized clocks in Section 5.2.

The process of computing the critical path (CP shown in Listing 1) on the logical DAG starts at the root node R—the endpoint under study. We sort all its children by their span *end time* and pick the last finishing child (LFC). The entirety of LFC is on the critical path. Let LFC_s be the start time of the LFC; we ignore all children spans of R that may start or end in the time intervening between the start and the end of LFC. We now look for the next child of R whose end time immediately precedes LFC_s and perform the same procedure iteratively until no child is left to process. Time not attributed to any child of R is attributed to the root span itself.

The process is also recursive. Once an LFC is identified, it recursively calls CP on its own children to distribute its time under its children. The result of the CP algorithm is a sequence of graph nodes with time associated with each one of them. Applying this algorithm to the trace shown in Figure 4, the critical path is represented by the fragments $A_1B_1C_1B_3A_5D_1A_7$.

There are two types of metrics associated with each node of the critical path — inclusive time and exclusive time. The "exclusive" time does not include the time spent in a node's callees. The "inclusive" time is the total wall clock time from the start to the end of the RPC on the specific node.

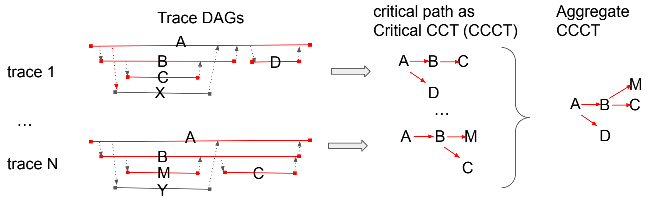


Figure 5: From trace to DAG to critical path (CCCT) to aggregate critical calling context tree. In the trace DAGs (left of the diagram) the x-axis is the flow of time. Horizontal lines are Jaeger spans and vertical lines are caller-callee relationships. Red-colored horizontal spans are on the critical path.

Since every node on the critical path encodes the information on how it was called, and since all call paths originate from a common root — the endpoint under investigation — it enables us to merge all call paths into a calling context tree (CCT) [14] by looking at their common prefixes. Consider the critical path $A_1B_1C_1B_3A_5D_1A_7$ for the trace in Figure 4. This path encodes the following call and return information: A calls B calls C returns to B returns to A calls D returns to A. With this, we can infer that there are the following call chains involved on the critical path: $A, A \to B, A \to B \to C, A \to B$, $A, A \to D$, and A. We can merge all these call paths into a CCT and call it a Critical Calling Context Tree (CCCT). This process is presented in the center section of Figure 5.

The calling context information makes it not only rich but also helps in aggregating critical paths from multiple traces described later in Section 5.3. A level of aggregation happens immediately within each trace processing: if the same endpoint appears multiple times on the critical path, we sum them as long as their call chains are exactly the same. For example, in the previous $A_1B_1C_1B_3A_5D_1A_7$ critical path example, we merge the multiple occurrences of call paths A_* and $A_* \to B_*$. This merger discards the ordering relationship between events, which we do not need for further analysis.

5.2 Challenges with the Clock Drift

The span start and end times recorded in Jaeger traces are both callee's local-machine time stamps converted to the standard UTC time. Machine clocks on two different physical machines drift [17, 49, 58] despite their periodic NTP-based synchronization. As a consequence of using local clocks, our critical path algorithm (if not corrected) can go wrong and sometimes lead to significant misattribution.

Span overlap problem: Figure 6 shows an ideal trace where the three spans A, B, and C are invoked one after another by the parent P. Most of the time should be attributed to the children. Figure 7 shows the trace for this example from our production, where the time recorded for the children spans have a small overlap; there is an overlap between the end of A and the start of B and the end of B and the start of C. In this case, the critical path

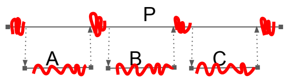


Figure 6: Ideal traces for a parent with three serialized children executions. Red lines show the critical path.

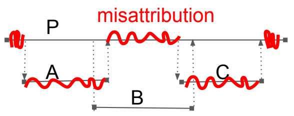


Figure 7: Actual traces due to clock drift. Red lines show the corresponding critical path.

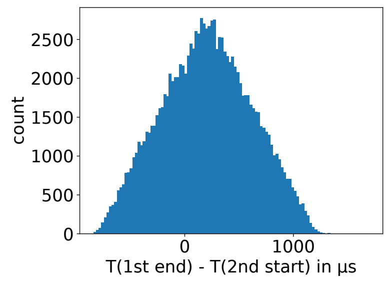


Figure 8: Distribution of time overlap recorded in Jaeger for two sequentially invoked RPCs. A positive value shows an overlap. The mean is 204.21μs and the max is 1696.00μs.

is not attributed to span B and instead attributed to the parent. Due to the clock drift, more than 50% of our traces recorded this type of span overlaps causing misattribution in critical paths.

We conducted a detailed study on the impact of such clock drift. Figure 8 plots the time overlap recorded in Jaeger traces of two sequentially invoked RPCs sampled over 118K traces. A positive value shows overlap and a negative value shows non-overlap. More than 50% of samples show an overlap. The P50 overlap is 204μs and the maximum overlap is 1696μs.

Based on this empirical observation, we tuned the `happensBefore(A, B)` part of our CP algorithm with the following relaxation:

- $A_{end} - threshold < B_{start}$, and
- No other children of the parent of P of A can start or end in the overlapped time range

The first condition allows a small `threshold` amount of overlap between the end of the previous span with the start of the next span. The second condition ensures that in the region of the allowed overlap, there is no other spawn and sync event, which ensures the parent-child serialization. The threshold is set to $1ms$.

Span overflow and underflow problems: In addition to the overlap, there can be overflow and underflow of child spans due to the clock drift. We enumerate these problems along with our pragmatic solutions below:

- A child span C may start before the start of the parent span P. In such cases, we truncate the start time of C till the start time of P. This may involve the recursive truncation of C's descendants.
- A child span C may end after the end of the parent span P. In such cases, we truncate the end time of C to the end time of P.

This may involve the recursive truncation of C's descendant.

- Although rare, a child span C may end before the start time of parent span P. Similarly, a child span C may start after the end time of the parent span P. In these cases, we completely drop the subtree formed by C for CPA.

This tailoring fixed our *CP* algorithm. The total time truncation over millions of traces was under 5% giving us the confidence that a significant part of the data was retained.

5.3 Aggregating Critical Paths

While one trace can be compressed into its essential critical path and represented as a CCCT, it may not be representative. Hence, we need to inspect numerous traces to derive a "typical" shape of the critical path. Distinct traces may exhibit different critical paths based on many things, such as calling parameters, scheduling decisions, system load, time of the day, and network delays, to name a few. Hence, a *summary* of *typical* components on the critical path is desired.

To this end, we merge all critical paths (represented as CCCTs) into a weighted, *aggregate CCCT*. We follow the tree merging process done in HPCToolkit [13]. The aggregate CCCT succinctly summarizes all call paths leading to critical path nodes in all traces; it captures the quantitative aspect by associating higher weights to those call paths that are often on the critical path. The weights of the nodes in such a tree would be the summation of the weights of the constituent call paths. Specifically, we provide different percentiles (e.g., P50, P95, P99) of the latency values, which are widely used for QoS purposes. Figure 5 exemplifies this process.

5.4 Workflow for Continuous CPA

Figure 9 depicts the workflow followed by CRISP for performing critical path analysis of microservice traces for all endpoints. The components belonging to CRISP are marked by the outermost rectangular box.

All services are instrumented to produce Jaeger traces during their RPCs. The instrumentation is enabled across languages such as Go, Java, Node.JS, and Python. The RPCs emit Jaeger spans into a common data store, which can be queried via SQL-style queries.

The CRISP workflow runs as a daily job. The workflow begins by collecting a list of endpoints. Each endpoint can be handled in parallel. Hence, we dedicate a handful of machines that shard the list of endpoints among them.

For each endpoint, CRISP queries the Jaeger data store (via `sawmill-query`) service to fetch a list of traceIDs. This query is set up to obtain the last two weeks' worth of traces. We then use these traceIDs to fetch the actual JSON traces (`jaeger-query`) service. We exploit IO parallelism here to fetch many traces concurrently.

We compute the critical path over each trace in parallel using the map-reduce paradigm. The set of critical paths obtained is fed into an aggregating process that summarizes and produces the daily critical path report for each endpoint (top-down analysis) and also produces overall metrics aggregated over all endpoints (bottom-up analysis). The results are injected into blob storage that can be easily navigated by a varied set of users, including service owners, performance engineers, and capacity managers. An offline anomaly detection model is also trained per endpoint result.

6 CRISP Features

We have developed tools to inspect critical paths for top-down performance analysis of specific endpoints, bottom-up analysis over all endpoints, and automatic anomaly detection over traces. We describe these features in this section.

6.1 Top-Down Analysis

We store the results of our CPA for each endpoint into profiles for investigation by service owners. CRISP provides the following means of visualization of CPA over each endpoint.

Flame graph: Flame graph [31] is a powerful way to visualize hierarchical call paths arising from profiling. The interactive visualization is easier to digest and investigate. Since we maintain the summarized critical paths as aggregate CCCTs, which are formed of many weighted call paths, it naturally avails itself to be represented as a flame graph.

If we chose all traces to represent a single flame graph, the critical path found in P99 latencies may dominate the flame graph and mask the other common cases. For that reason, we show three different flame graphs for different percentiles of latency values (e.g., P50, P95, and P99). We also produce differential flame graphs [30] that show how the critical paths change between two percentile values.

Heat map: Flame graphs are useful for navigating call chains but developers sometimes need access to an actual Jaeger trace that represents a given data so that they can inspect it in further detail. For this reason, CRISP provides the heat map view (see Figure 10), where the rows are the endpoints and the columns represent individual traces. Each cell in the heat map represents the exclusive time on the critical path and each cell is gradient colored based on its contribution (exclusive time) to the total latency. In this view, we collapse the call paths and accumulate the metrics from all call paths, reaching the same endpoint in a single row. However, for exploration, the developers have access to the top 5 call chains (not shown) for each endpoint, which is available by hovering over any row. In this view, the user can also choose percentile values and inclusive or exclusive metrics to sort the rows. Each column is also sorted by a high to low contribution for a given chosen metric. Selecting any trace takes the user to the Jaeger-UI to inspect the trace.

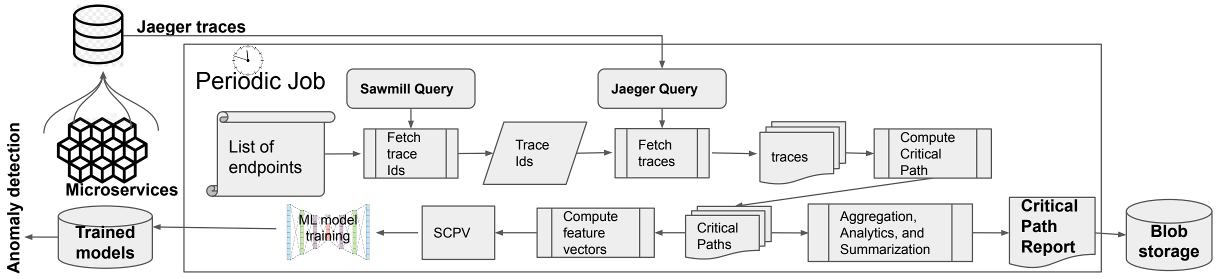
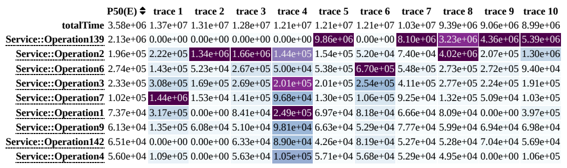


Figure 9: Schematic diagram of CPA over Jaeger traces.



	P50(E)	trace 1	trace 2	trace 3	trace 4	trace 5	trace 6	trace 7	trace 8	trace 9	trace 10
totalTime	3.58e+06	1.37e+07	1.31e+07	1.28e+07	1.21e+07	1.21e+07	1.21e+07	1.03e+07	9.39e+06	9.06e+06	8.99e+06
Service::Operation139	2.13e+06	0.00e+00	0.00e+00	0.00e+00	0.00e+00	9.86e+06	0.00e+00	8.10e+06	3.23e+06	4.36e+06	5.39e+06
Service::Operation2	1.96e+05	2.22e+05	1.34e+06	1.66e+06	1.44e+05	1.54e+05	5.20e+04	7.40e+04	4.02e+06	2.07e+05	1.30e+06
Service::Operation6	2.74e+05	1.43e+05	5.23e+04	2.67e+05	5.00e+04	5.38e+05	6.70e+05	5.48e+05	2.73e+05	2.72e+05	9.40e+04
Service::Operation3	2.33e+05	3.08e+05	1.69e+05	2.69e+05	2.01e+05	2.01e+05	2.54e+05	4.11e+05	2.77e+05	2.24e+05	1.91e+05
Service::Operation7	1.02e+05	1.44e+06	1.53e+04	1.41e+05	9.68e+04	1.30e+05	1.06e+05	9.25e+04	1.32e+05	5.09e+04	1.03e+05
Service::Operation1	7.37e+04	3.17e+05	0.00e+00	8.41e+04	2.49e+05	6.97e+04	8.18e+04	6.66e+04	8.09e+04	0.00e+00	3.97e+05
Service::Operation9	6.13e+04	1.35e+05	6.08e+04	5.10e+04	9.81e+04	6.63e+04	5.29e+04	7.77e+04	5.99e+04	6.94e+04	6.98e+04
Service::Operation142	6.51e+04	0.00e+00	0.00e+00	6.33e+04	8.90e+04	4.26e+04	8.19e+04	5.27e+04	5.28e+04	7.04e+04	5.69e+04
Service::Operation4	5.60e+04	1.09e+05	0.00e+00	5.63e+04	1.05e+05	5.71e+04	5.68e+04	5.29e+04	4.95e+04	0.00e+00	1.06e+05

Figure 10: Example heat map from 1000 traces. The result is sorted by the P50 percentile value of the exclusive time of each operation. Each cell is the accumulated time in μs.

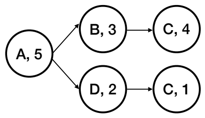
6.2 Bottom-Up Analysis

The objective of the bottom-up analysis is to derive insights from *all* endpoints and to bubble up those interior APIs improving which will improve many endpoints. The bottom-up analysis is a data-intensive process and needs access to critical paths from all endpoints. For this reason, we retain the aggregate CCCT computed for each endpoint from the top-down process, along with some additional statistics related to the overall graph structure. Once all endpoints are processed, the bottom-up analysis runs; it aggregates the statistics from each endpoint and quantifies the impact of each API over all other endpoints. The output of the bottom-up analysis is a descending priority list of top APIs that are often in many endpoints. Additionally, the bottom-up analysis produces various histograms over all traces taken together, which include the total number of times any API appears in any graph, the total number of times an API appears on the critical path, the number of unique APIs on the critical path, the critical path length, and the maximum degree of concurrency in a trace, among others. These graphs are intended to inform infrastructure and hardware engineers to better understand the current needs of our systems and aid capacity planning for the future.

6.3 Anomaly Detection

We also employ CRISP to pinpoint whether a new incoming trace (for a given endpoint) deviates from the normal execution behavior. For this purpose, we have trained a machine learning model and used it for inference.

During the offline training, we encode the critical path (CCCT) for each trace of an endpoint into feature vectors,



A	5
A→B	3
A→B→C	4
A→D	2
A→D→C	1

Figure 11: An example CCCT (left), the letters indicate name and the numbers indicate the exclusive time on the span. The corresponding SCPV (right).

which we call service critical path vectors (SCPV). We feed several SCPVs into an autoencoder to learn the normal execution pattern of the given service. During the online inference, the learned model will infer whether the given new trace is abnormal or not based on an anomaly score.

The architecture design, training, and inference of the autoencoder are derived from TraceAnomaly [39], which is the state-of-the-art framework for anomaly detection in microservices trace. The neural architectural details are described in Appendix B. The key difference between CRISP and TraceAnomaly is in the data encoding. TraceAnomaly uses a service trace vector (STV) which encodes every path in the trace and, in contrast, CRISP encodes only on the call paths for those spans that are on the critical path spans.

SCPV encoding: Figure 11 exemplifies encoding the critical path present as a CCCT into an SCPV. For each node in CCCT, it assigns weights based on its exclusive execution time. Notice that endpoint *C* occurs twice on the critical path, thus it is also encoded twice in the SCPV, given the call chain is different. The training set is a 2D matrix where each column is a feature (call path) and each row is the feature values of a given trace.

Using the call paths of spans only on the critical path, compared with the prior work that used all call paths in the entire graph, offers significant benefits. It reduces the feature dimensions; it reduces the training and inference time; and, most importantly, it improves the model accuracy. The impact of the CCCT-based encoding is substantial and evaluated in Section 7.3.

7 Experience and Evaluation

In Section 7.1, we describe one of our findings by applying the top-down analysis of CRISP at Uber, in Section 7.2 we

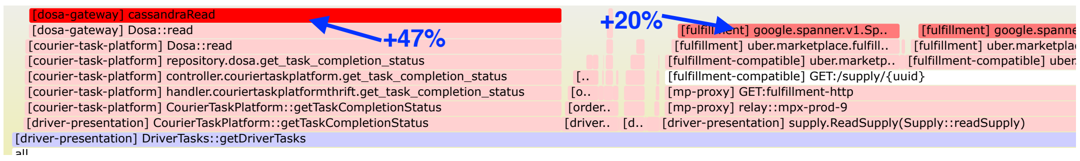


Figure 12: Differential flame graph for the `getDriverTask` endpoint. Red colors indicate the growth from P50 critical paths to P95 critical paths.

show valuable characteristics of microservices at Uber by applying the bottom-up analysis of CRISP. In Section 7.3, we empirically evaluate the anomaly detection capability of CRISP and in Section 7.4 we describe how we employed CRISP in guiding future hardware selection to reduce tail latency in our services.

7.1 Tail Latency Investigation via Top-Down Analysis

`getDriverTasks` is a business-critical endpoint in the driver-presentation service responsible for returning the task plan that a driver needs to perform. A sample task plan could be: passenger mask check, pickup passenger, pickup food, drop off passenger, and drop off food. This endpoint assembles the task plan and enriches it by calling numerous other microservices such as courier-task-platform.

Figure 12 shows the *differential* flame graph for the `getDriverTask` endpoint. The graph plots a difference between the critical paths seen in the traces with the P50 latency vs. P95 latency for the `getDriverTask` endpoint. The red-colored boxes show the growth in percentage time spent in P95 with regards to P50. The `getTaskCompletionStatus` API was absent in the P50 traces, whereas it occupies 47% of the total execution in P95 traces, contributing to the same amount of addition to the tail latency. This endpoint dependency makes a call to Cassandra—an expensive database read. Based on this insight from CRISP's differential flame graph views, we identified the root cause of performance variance and high tail latency. We recommend caching with timestamp filtering optimization as opposed to a database read to reduce the tail latency.

Trace processing overheads: Table 1 shows the overhead of analyzing the `getDriverTasks` endpoint discussed in this section running on 16 cores of an Intel Xeon Skylake machine clocked at 2.4 GHz.

Table 1: Overhead of top-down analysis of `getDriverTasks`.

Num Traces	Trace size	Processing time	Memory usage
10k	6.8 GB	48 sec	2.1 GB
20k	14 GB	109 sec	4.2 GB
40K	28 GB	232 sec	8.5 GB
80K	56 GB	553 sec	17.6 GB

Sparse sampling vs. quality of CPA: We observed that the sampling rate does not qualitatively affect the aggregate critical path results. We conducted an experiment where we first produced an aggregate critical path from 1 million traces. We also produced critical paths from randomly sampled 100K and 10K traces from the same data set. We noticed that the attribution of the top 20 services on the critical path, whether for 10K or 100K samples, was essentially the same as the one produced from 1M traces.

7.2 Systemic Insights via Bottom-Up Analysis

In this section, we show the result of running CRISP with bottom-up analysis on the collected trace dataset and some insight associated with the data. The dataset includes more than 1 million traces, ∼4k services, and ∼40k endpoints. It takes around 4 hours on 32-cores of a Intel Xeon Skylake machine clocked at 2.4 GHz.

Total RPCs per request: Figure 13 is a histogram of the total number of RPCs made per request, which is same as the total number of spans in a trace. On average there are 112 spans in a trace. However, there exist several large ones with a maximum of 275K spans. Such scale brings significant challenges for the developer to debug without proper reduction of the graph size.

Total endpoints in a trace: Figure 14 is a histogram of the total number of unique endpoints found in each trace. At most each trace has 1400 unique endpoints.

Latency distribution: Figure 15 plots the histogram of latencies observed in each of $\sim 1M$ traces. The tail is several orders of magnitudes longer than the mean or median.

RPC depth: Figure 16 is a histogram of the longest call chain found in each trace. The depth of the call chain is another measure of the complexity of traces. The average RPC depth is 8.5. The maximum observed depth is 36.

Unique caller: Figure 17 is a histogram of the number of the unique callers for each endpoint across one million traces. The number differs wildly as the mean value is just above 2 but the maximum value is 620.

Degree of concurrency: Figure 18 is a histogram of the maximum number of spans that overlap in time in each trace. This number gives the degree of concurrency (and hence a measure of the complexity) in our traces. Overall, the microservices show a high degree of concurrency. On average, the degree of concurrency is 21. The degree of concurrency often grows to 100s for more complicated services. The maximum degree of concurrency we observed in $\sim$ 1M traces was 3076.

Total RPCs on the critical path: Figure 19 is a histogram of the number of spans on the critical paths, which counts the number of RPCs made on the critical path. Besides a few outliers, the length of the critical path is short. On average, there are 33

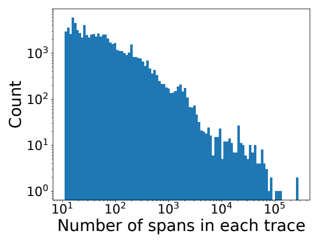


Figure 13: Histogram of the number of spans per trace.

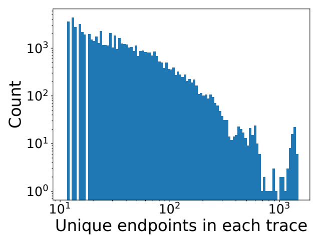


Figure 14: Histogram of number of unique endpoints per trace.

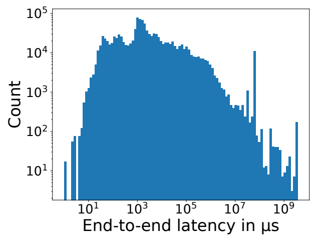


Figure 15: Distribution of latency among all traces.

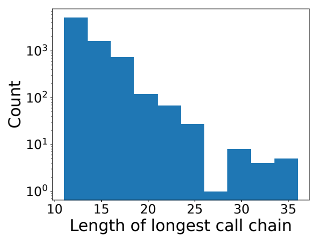


Figure 16: Histogram of longest call chain per trace.

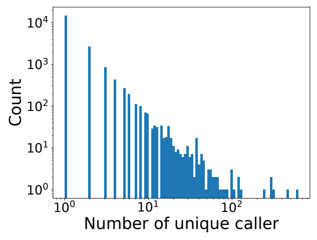


Figure 17: Histogram of the number of unique caller for each endpoint.

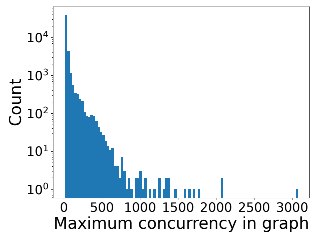


Figure 18: Histogram of the degree of the concurrency (max no. of overlapping spans) per trace.

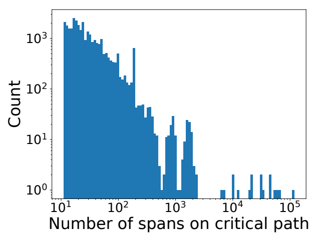


Figure 19: Histogram of the number of spans on the critical path per trace.

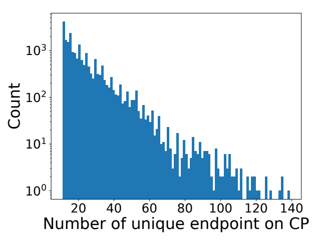


Figure 20: Histogram of the number unique endpoints on the critical path per trace.

RPCs on the critical path (in contrast, the entire graph in Figure 13 shows 112-275K RPCs in traces). The short critical path length allows the developer to investigate and debug easily.

Endpoints on the critical path: Figure 20 is a histogram of the unique endpoints on each critical path. Compared with the number of endpoints in the entire trace (Figure 14), the number of the endpoints on the critical path is an order of magnitude smaller (the maximums are 1400 vs. 140). The 10x size-reduction matches our observation of the 6 services we test for anomaly detection.

7.3 Empirical Analysis of Anomaly Detection

Here, we will evaluate CRISP's anomaly detection on six critical endpoints.

Methodology: We collect traces for six microservices in real production over a 14-day period. The training data for each case includes 20,000 traces and the testing data has 500 unseen traces for normal and abnormal data. To generate abnormal inference data, we drop 20% of the nodes in the graph and randomly shuffle the duration of the nodes as described in [26,39,48]. We did not use real anomalous traces for evaluation since we do not have a large number of labelled anomalous traces (i.e., we have a lot of false negatives). Also, the labeled data contains false positives and coordinating with hundreds of developers to verify the veracity of labeling is non-trivial.

We use TraceAnomaly [39] as the baseline against which we compare our results. We adopt the same architecture of the autoencoder and reuse their code. The main difference is that we use CRISP to preprocess the trace before feeding it into the autoencoder so that only paths appearing on the critical path information are included. A fundamental assumption is that any noticeable difference in the trace must impact the critical path.

Hardware: We use two machines in our evaluation: a CPU-only machine with 256 GB memory and a CPU+GPU machine with 128 GB memory. Most of the experiment is done on a machine with GPUs. It has 2 Quadro RTX 5000 GPUs and 2 socket Intel Xeon Gold 5218 CPU at 2.30GHz. The CPU machine has 2 sockets with Intel Xeon Silver 4214 CPU at 2.20GHz. Both machines run on Linux 4.14. The reason to use two machines is that for some experiments, the training data for TraceAnomaly cannot fit the GPU memory, whereas CRISP's training data always fits on GPU memory. In such cases, for a fair comparison, we also run the experiment on the 256 GB CPU-only.

Table 2 shows the empirical evaluation results of anomaly detection on 6 large online services at Uber. It captures the essential features such as the number of RPCs, unique endpoints, and call path diversity in these services. It also shows the training and inference time with both STV (prior art from TraceAnomaly) and SCPV (our work) data. Finally, the last 4 columns present the model accuracy in terms of precision and recall. In summary, using critical path via CRISP reduces the training time and inference time and improves the recall performance on top of the state of the art.

Training speedup: From the table, we can observe that CRISP offers up to 22× speedup for training compared with TraceAnomaly. Even the smallest speedup is more than 50%.

The reason for the speedup is that the training data from CRISP (SCPV) is one magnitude smaller than TraceAnomaly (STV) up to 25× for Service 6. The number of unique call paths on the critical path is significantly smaller than the total number of call paths in the entire graph (also see Figures 13-20). Furthermore, when the number of the trace and the dimension

Table 2: Evaluation results for large online services. Inference time is measured with 1000 traces. (TA*=TraceAnomaly.)

	No. of Unique endpoints	Max no. of spans	No. of callpaths/ features		Training Time		CRISP training speedup	Inference Time		CRISP inference speedup	Precision		Recall	
			STV	SCPV	TA*	CRISP		TA*	CRISP		TA*	CRISP	TA*	CRISP
Service 1	214	1429	5117	1186	70m (GPU)	46m (GPU)	1.52X	2.24s (GPU)	1.21s (GPU)	1.85X	1.0	0.998	0.986	0.992
Service 2	969	1724	9725	1860	100m (GPU)	50m (GPU)	2.00X	3.54s (GPU)	1.40s (GPU)	2.54X	1.0	1.0	0.958	0.984
Service 3	734	5320	20321	2154	150m (GPU)	50m (GPU)	3.00X	5.64s (GPU)	1.36s (GPU)	4.15X	1.0	1.0	0.5	0.982
Service 4	912	20001	25347	2715	1184m (CPU)	56m (GPU) 219m (CPU)	21.14X (GPU) 5.41X (CPU)	56.67s (CPU)	1.56s (GPU) 9.26s (CPU)	36.33X (GPU) 6.12X (CPU)	1.0	1.0	0.928	0.978
Service 5	768	6562	26404	2336	811m (CPU)	51m (GPU) 177m (CPU)	15.90X (GPU) 4.58X (CPU)	42.90s (CPU)	1.36s (GPU) 5.81s (CPU)	31.54X (GPU) 7.38X (CPU)	1.0	0.998	0.5	0.982
Service 6	1477	10992	28968	1151	1305m (CPU)	46m (GPU) 148m (CPU)	27.77X (GPU) 8.82X (CPU)	78.88s (CPU)	1.18s (GPU) 4.48s (CPU)	66.85X (GPU) 17.61X (CPU)	1.0	1.0	0.912	0.977

of the feature vector is large, the size of the training data of TraceAnomaly can easily exceed the memory of the GPU, which makes it unable to train. For such cases (Service 4, 5, and 6), we can still see more than 4× speedup even if we train both TraceAnomaly and CRISP on CPU machines. When CRISP is trained on the GPU machine, the speedup can easily exceed 15×. The faster training allows for more practical deployment.

Inference speedup: Similar to training speedup, the reduction in inference data size leads to a faster inference of CRISP. The smallest speedup is more than 1.85× whereas the largest speedup is over 66×. This lower latency allows us to batch many inferences together to exploit GPU throughput.

Precision: From Table 2, we can see that both TraceAnomaly and CRISP are capable of detecting the abnormal trace accurately. Autoencoders are capable of capturing the complex pattern of the graph. TraceAnomaly works slightly better than CRISP on 2 services, but overall accuracy is very high for both methods.

Recall: The recall is the part that differentiates the quality of results between TraceAnomaly and CRISP. Recall measures how many of the actual positives the model captures through labeling it as positive, (i.e., $\frac{True\_Positive}{True\_Postive+False\_Positive}$). When the recall is closer to 1, it indicates that the model makes fewer false-positive predictions (an anomaly in this case). From Table 2, it is clear that CRISP outperforms TraceAnomaly by a noticeable margin. Particularly for Service 3 and 5, half of the positive prediction of the anomaly is false, meaning all normal traces for inference are labeled abnormal by TraceAnomaly. To make sure the prediction is actually incorrect, we asked the service owners and verified that the normal inference testing traces are not showing any abnormal behaviors. On the contrary, CRISP's recall is close to 1. For Service 1 and 2, the performance of CRISP is slightly better than TraceAnomaly, as both models make relatively accurate predictions. CRISP shows more than 5% improvement for Service 4 and 6.

CRISP produces superior results on services with a large number of call paths. For instance, there are 912 endpoints in Service 4 but the total call paths is 25,347. Since there is more diversity among the shapes of the call chains on the entire graph, the SCPV encoding fails to capture its full variety; consequently, unseen call paths easily trigger a false positive in TraceAnomlay. In contrast, the critical path remains fairly stable when trained over a large corpus of traces, and consequently CRISP has fewer false positives.

7.4 CPA in Hardware Selection

In addition to the parent-child transitive relationships and times, Jaeger traces also contain additional information, such as the hostname on which the span was executed. Uber's data center consists of diverse hardware CPU SKUs. Services can be installed on different hardware versions. Hence, an API may run on different hardware on different requests.

We collected the critical path for one of our important services using CRISP and identified that a downstream operation was on the critical path. We further clustered the samples from the profiles by the CPU versions on which they were running. The violin plot in Figure 21 in Appendix A shows how the latencies vary on 2 prominent CPU SKUs: Intel Xeon Silver 4212 running at 2.2 GHz (SKU-A) and Intel Xeon Silver 4212R running at 2.4 GHz (SKU-B). The two SKUs are identical (same vendor, microarchitecture, cache size, etc.) with the only exception being that their CPU clock speeds are different. This mild (9%) difference in the clock speed has a profound impact on the behavior of the plotted service. The P50 value for SKU-A is 15% higher than that on SKU-B. Moreover, the tail latency on SKU-A is 1.5x higher than the one on SKU-A.

To summarize, a slightly faster CPU clock proves to have a significant impact on reducing the tail latency and overall latency. This difference has a significant impact on the overall capacity allocation since tail latency (e.g., P95) is often used in capacity allocation. This observation demands further, systematic investigation into classifying critical path components as CPU SKU sensitive vs. insensitive; also, such categorization helps data center-wide microservice schedulers to favor SKU-sensitive services on the critical path onto the SKUs where they exhibit superior performance.

8 Related Work

Critical Path Analysis (CPA) has been extensively explored in the shared-memory parallel programming paradigm [13, 20, 25, 40, 50, 54, 55, 59, 62] but less explored in distributed parallel systems. Unlike shared-memory and structured parallel programs, microservices use distributed parallel computing environments and are unstructured in nature.

Barford and Crovella [16] utilize critical path analysis for profiling and understanding TCP transactions and improve data transfer latency in web applications; however their scale is significantly smaller than the 4K services deployed over millions of CPU cores that we handle. Bohem et al. [18] employ tracing and CPA for MPI programs in HPC environments; this approach has not been employed in microservice environments. Kaldor et al. [33] develop an end-to-end tracing system (Canopy) for tracking requests from web-browsers/mobile to backend services; it handles billions of traces. A distinguishing feature of CRISP compared with Canopy is the use of CPA, which significantly reduces the data needed for analysis.

Qiu et al. [48] propose a fine-grained resource management framework based on microservice traces using CPA. They employ the insights for scheduling and other resource management to reduce CPU utilization. However, their work does not cover industry-scale deployment; they also do not facilitate performance bug or anomaly detection and cannot provide bottom-up system-wide performance insights.

Fields et al. [24] explore a hardware predictor to analyze the criticality of instructions by using CPA and use it to guide dynamic instruction scheduling. Venkataramani et al. [55] propose Global Critical Path (GCP) to predict system-level performance and optimize the performance of highly concurrent self-timed circuits. These approaches rely on the precise last arriver information, which is readily available in these cases. Our critical path computation in microservices also depends on knowing the last arriver. Unlike the aforementioned approaches, we do not have direct access to the last arriver in our distributed system. As a result, we need to use clock information from different hosts and adjust for clock skew to heuristically infer the last arriver.

Multiple tools have been developed to profile and debug large distributed and parallel systems. `lprof` [64] constructs request flow from logs and it is as good as the quality of logs; it has not been evaluated on microservices; it also does not provide CPA and hence suffers from a voluminous noisy data. Mace et al. [41] developed Pivot as a dynamic, extensible tracing system for inter-operating applications. Pivot employs a happen-before relationship between events to establish causality. Pivot does not build a critical path and hence pays equal attention to any causal relationship unlike CRISP. Chow et al. [19] build a system that utilizes a large number of request traces to validate hypotheses about causal relationships. Edgar [2] provides a summarized view of request traces, logs, and metadata in distributed systems. It does not employ sophisticated analyses or automated anomaly detection.

Several works have focused on microarchitectural aspects of microservices [34, 42, 52, 60, 61]. Most of these works are focused on how microservices utilize microarchitectural features, but ignore the end-to-end user request; in contrast, CRISP takes a higher-level approach and looks at the entire flow of requests through a chain of services.

Multiple works have studied anomaly detection in distributed systems. Liu et al. [39] use Deep Bayesian Network to detect the performance anomaly in an unsupervised manner. They utilize machine learning to learn the normal behavior pattern of the given dependency graph and try to detect the anomaly online. Gan et al. [26] propose a root cause analysis system for large-scale microservices using machine learning. The system uses Conditional Variational Autoencoders (CVAE) [51] to automatically generate the counterfactual training data. These approaches have used the entire call graph, leading to significant training and inference time. In contrast, CRISP uses only the critical path(s), leading to dramatic speedups while producing higher quality results.

9 Conclusions and Future Work

Microservices are the preferred architecture choice in modern service-oriented software systems. Large-scale microservices have tens of thousands of endpoints with complex, nested, and asynchronous. Prior work in profiling microservices has either focused on tracing techniques, which produce a lot of data, but lack in delivering insights, or on micro-architectural optimization within a service, ignoring the full picture of the life of a request through myriad services. This paper develops a tool, CRISP, which uses critical path analysis (CPA) over RPC traces to bubble-up interesting activities and discard noisy events. CRISP provides rich developer insights both for service owners and infrastructure engineers. In a short three-month deployment period, CRISP's analyses have sifted over 4,000+ services, 40,000+ endpoints, hundred of millions of traces, and tens of terabytes of data at Uber; as a result, CRISP has bubbled-up profiling results that helped developers understand and optimize important services. Employing the critical path, as opposed to the whole RPC trace, speeds up the training of models and on-the-fly inference for anomaly detection while also producing noticeably higher quality results.

Our future work involves enhancing CRISP to address other use cases such as setting the TTL values for downstream calls and bubbling up those downstream services that often return errors. We plan to expand our anomaly detection to include developers in the loop and improve traces with labelled data.

Availability

Parts of the code of this work are open-sourced [1].

Acknowledgement

This material is based upon work supported by the National Science Foundation under Grants No. 2006542 and 1763699. We thank our shepherd and the anonymous reviewers for their feedback. We also thank William Fulton and Sara Wilmes-Reitz for their help with proof-reading a draft version of this paper.

References

[1] CRISP: Critical Path Analysis of Microservice Traces. https://github.com/uber-research/CRISP.

[2] Edgar: Solving Mysteries Faster with Observability. https://netflixtechblog.com/edgar-solving-mysteries-faster-with-observability-e1a76302c71f. (Accessed on 11/30/2021).

[3] Evolving Schemaless into a Distributed SQL Database. https://eng.uber.com/schemaless-sql-database/. (Accessed on 01/12/2022).

[4] Fast and Reliable Schema-Agnostic Log Analytics Platform. https://eng.uber.com/logging/. (Accessed on 01/12/2022).

[5] Heat Map. https://en.wikipedia.org/wiki/Heat_map. (Accessed on 01/11/2022).

[6] Jaeger: open source, end-to-end distributed tracing. https://www.jaegertracing.io/. (Accessed on 12/01/2021).

[7] perf (linux) - wikipedia. https://en.wikipedia.org/wiki/Perf_(Linux). (Accessed on 01/12/2022).

[8] Uber's Fulfillment Platform: Ground-up Re-architecture to Accelerate Uber's Go/Get Strategy. https://eng.uber.com/fulfillment-platform-rearchitecture/. (Accessed on 01/12/2022).

[9] User Datagram Protocol. https://en.wikipedia.org/wiki/User_Datagram_Protocol. (Accessed on 01/11/2022).

[10] perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page, 2013.

[11] Profiling Go Programs. https://blog.golang.org/pprof, 2013.

[12] Head-based and tail-based sampling, rate-limiting. https://opentelemetry.uptrace.dev/guide/sampling.html#introduction, April 2022.

[13] ADHIANTO, L., BANERJEE, S., FAGAN, M., KRENTEL, M., MARIN, G., MELLOR-CRUMMEY, J., AND TALLENT, N. R. HPCToolkit: Tools for performance analysis of optimized parallel programs. *Concurrency and Computation: Practice and Experience 22*, 6 (2010), 685–701.

[14] AMMONS, G., BALL, T., AND LARUS, J. R. Exploiting Hardware Performance Counters with Flow and Context Sensitive Profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1997), PLDI '97, Association for Computing Machinery, p. 85–96.

[15] APACHE FLINK TEAM. Apache Flink: Stateful Computations over Data Streams. https://flink.apache.org/.

[16] BARFORD, P., AND CROVELLA, M. Critical path analysis of TCP transactions. *ACM SIGCOMM Computer Communication Review 30*, 4 (2000), 127–138.

[17] BLUEMATADOR. Time Drift (NTP). https://www.bluematador.com/docs/troubleshooting/time-drift-ntp.

[18] BÖHME, D., GEIMER, M., ARNOLD, L., VOIGTLAENDER, F., AND WOLF, F. Identifying the Root Causes of Wait States in Large-Scale Parallel Applications. *ACM Trans. Parallel Comput. 3*, 2 (jul 2016).

[19] CHOW, M., MEISNER, D., FLINN, J., PEEK, D., AND WENISCH, T. F. 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)* (2014), pp. 217–231.

[20] CURTSINGER, C., AND BERGER, E. D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 184–197.

[21] DELIMITROU, C., AND KOZYRAKIS, C. Amdahl's Law for Tail Latency. *Commun. ACM 61*, 8 (jul 2018), 65–72.

[22] EPANECHNIKOV, V. A. Non-parametric estimation of a multivariate probability density. *Theory of Probability & Its Applications 14*, 1 (1969), 153–158.

[23] FERDMAN, M., ADILEH, A., KOCBERBER, O., VOLOS, S., ALISAFAEE, M., JEVDJIC, D., KAYNAK, C., POPESCU, A. D., AILAMAKI, A., AND FALSAFI, B. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2012), ASPLOS XVII, Association for Computing Machinery, p. 37–48.

[24] FIELDS, B., RUBIN, S., AND BODIK, R. Focusing processor policies via critical-path prediction. In *Proceedings 28th Annual International Symposium on Computer Architecture* (2001), IEEE, pp. 74–85.

[25] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (New York, NY, USA, 1998), PLDI '98, Association for Computing Machinery, p. 212–223.

[26] GAN, Y., LIANG, M., DEV, S., LO, D., AND DELIMITROU, C. Sage: practical and scalable ML-driven performance debugging in microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (2021), pp. 135–151.

[27] GLUCK, A. Introducing Domain-Oriented Microservice Architecture). https://eng.uber.com/microservice-architecture/.

[28] GOLDBERG, Y. Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture. https://www.infoq.com/presentations/scale-gilt.

[29] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction* (New York, NY, USA, 1982), SIGPLAN '82, Association for Computing Machinery, p. 120–126.

[30] GREGG, B. Differential Flame Graphs. https://www.brendangregg.com/blog/2014-11-09/differential-flame-graphs.html.

[31] GREGG, B. The flame graph. *Communications of the ACM 59*, 6 (2016), 48–57.

[32] HAQUE, M. E., HE, Y., ELNIKETY, S., NGUYEN, T. D., BIANCHINI, R., AND MCKINLEY, K. S. Exploiting Heterogeneity for Tail Latency and Energy Efficiency. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2017), MICRO-50 '17, Association for Computing Machinery, p. 625–638.

[33] KALDOR, J., MACE, J., BEJDA, M., GAO, E., KUROPATWA, W., O'NEILL, J., ONG, K. W., SCHALLER, B., SHAN, P., VISCOMI, B., ET AL. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 34–50.

[34] KANEV, S., DARAGO, J. P., HAZELWOOD, K., RANGANATHAN, P., MOSELEY, T., WEI, G.-Y., AND BROOKS, D. Profiling a warehouse-scale computer. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture* (2015), pp. 158–169.

[35] KINGMA, D. P., AND WELLING, M. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114* (2013).

[36] KRAMER, S. The Biggest Thing Amazon Got Right: The Platform. https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/, October 2011.

[37] KREPS, J., NARKHEDE, N., RAO, J., ET AL. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (2011), vol. 11, pp. 1–7.

[38] LI, J., SHARMA, N. K., PORTS, D. R. K., AND GRIBBLE, S. D. Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2014), SOCC '14, Association for Computing Machinery, p. 1–14.

[39] LIU, P., XU, H., OUYANG, Q., JIAO, R., CHEN, Z., ZHANG, S., YANG, J., MO, L., ZENG, J., XUE, W., ET AL. Unsupervised detection of microservice trace anomalies through service-level deep bayesian networks. In *2020 IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)* (2020), IEEE, pp. 48–58.

[40] LIU, X., MELLOR-CRUMMEY, J., AND FAGAN, M. A New Approach for Performance Analysis of OpenMP Programs. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing* (New York, NY, USA, 2013), ICS '13, Association for Computing Machinery, p. 69–80.

[41] MACE, J., ROELKE, R., AND FONSECA, R. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), pp. 378–393.

[42] MARS, J., TANG, L., HUNDT, R., SKADRON, K., AND SOFFA, M. L. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-Locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture* (New York, NY, USA, 2011), MICRO-44, Association for Computing Machinery, p. 248–259.

[43] MAURO, T. Adopting Microservices at Netflix: Lessons for Architectural Design. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/, Feb 2015.

[44] MIRHOSSEINI, A., SRIRAMAN, A., AND WENISCH, T. F. Enhancing Server Efficiency in the Face of Killer Microseconds. In *Proceedings of the 25th International Symposium on High-Performance Computer Architecture (HPCA '19)* (2019), IEEE, pp. 185–198.

[45] NADAREISHVILI, I., MITRA, R., MCLARTY, M., AND AMUNDSEN, M. *Microservice Architecture: Aligning Principles, Practices, and Culture*, 1st ed. O'Reilly Media, Inc., 2016.

[46] OPENTRACING DEVELOPERS. OpenTracing: Span. https://opentracing.io/docs/overview/spans/.

[47] OPENTRACING DEVELOPERS. OpenTracing: What is Distributed Tracing? https://opentracing.io/docs/overview/what-is-tracing/.

[48] QIU, H., BANERJEE, S. S., JHA, S., KALBARCZYK, Z. T., AND IYER, R. K. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-Oriented Microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 805–825.

[49] S V, S. Time drift monitoring: Troubles of unsynchronized servers - site24x7 blog. https://www.site24x7.com/blog/time-drift-monitoring-troubles-of-unsynchronized-servers. (Accessed on 06/02/2022).

[50] SCHARDL, T. B., KUSZMAUL, B. C., LEE, I.-T. A., LEISERSON, W. M., AND LEISERSON, C. E. The Cilkprof Scalability Profiler. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures* (New York, NY, USA, 2015), SPAA '15, Association for Computing Machinery, p. 89–100.

[51] SOHN, K., LEE, H., AND YAN, X. Learning structured output representation using deep conditional generative models. *Advances in neural information processing systems 28* (2015), 3483–3491.

[52] SRIRAMAN, A., DHANOTIA, A., AND WENISCH, T. F. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of the 46th International Symposium on Computer Architecture* (2019), Association for Computing Machinery, p. 513–526.

[53] SRIRAMAN, A., LIU, S., GUNBAY, S., SU, S., AND WENISCH, T. F. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking* (2016).

[54] TALLENT, N. R., AND MELLOR-CRUMMEY, J. M. Effective performance measurement and analysis of multithreaded applications. In *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming* (2009), pp. 229–240.

[55] VENKATARAMANI, G., BUDIU, M., CHELCEA, T., AND GOLDSTEIN, S. C. Global critical path: A tool for system-level timing analysis. In *Proceedings of the 44th annual Design Automation Conference* (2007), pp. 783–786.

[56] VILLAMIZAR, M., GARCÉS, O., CASTRO, H., VERANO, M., SALAMANCA, L., CASALLAS, R., AND GIL, S. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)* (2015), pp. 583–590.

[57] WESTFALL, P. H., AND YOUNG, S. S. *Resampling-based multiple testing: Examples and methods for p-value adjustment*, vol. 279. John Wiley & Sons, 1993.

[58] WIKIPEDIA. Clock drift. https://en.wikipedia.org/wiki/Clock_drift.

[59] YANG, C.-Q., AND MILLER, B. Critical path analysis for the execution of parallel and distributed programs. In *[1988] Proceedings. The 8th International Conference on Distributed* (1988), pp. 366–373.

[60] YANG, H., BRESLOW, A., MARS, J., AND TANG, L. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (New York, NY, USA, 2013), ISCA '13, Association for Computing Machinery, p. 607–618.

[61] YASIN, A. A top-down method for performance analysis and counters architecture. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2014), IEEE, pp. 35–44.

[62] YOGA, A., AND NAGARAKATTE, S. Parallelism-centric what-if and differential analyses. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2019), pp. 485–501.

[63] ZHANG, Y., MEISNER, D., MARS, J., AND TANG, L. Treadmill: Attributing the Source of Tail Latency through Precise Load Testing and Statistical Inference. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016), pp. 456–468.

[64] ZHAO, X., ZHANG, Y., LION, D., ULLAH, M. F., LUO, Y., YUAN, D., AND STUMM, M. lprof: A non-intrusive request flow profiler for distributed systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 629–644.

A Violin Plot for Hardware Selection

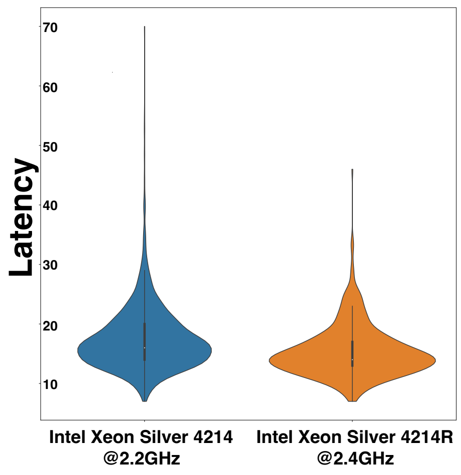


Figure 21: Violin plots of the exclusive execution time of a critical path operation with two different CPUs. The latency is in μs.

B Autoencoder Model Architecture

We choose the Deep Bayesian Network for anomaly detection given it is capable of learning complex patterns from the trace. We adopt the model from TraceAnomaly [39], which is the state-of-the-art framework for microservice trace based anomaly detection. Specifically, we adopt Variational Auto-Encoder (VAE) [35] to model the distribution pattern from the normal execution. VAE is an unsupervised learning that does not require a label, which can be expensive to obtain in our setting due to the volume of traces. Figure 22 depicts the architecture of VAE. It has three components: encoder, posterior flow, and decoder.

The encoder contains 1 hidden layer ($h_\phi(\mathbf{x})$) to learn the hidden features of SCPV. The goal is to learn the mean $\mu_{\mathbf{z}^{(0)}}$ and the standard deviation $\sigma_{\mathbf{z}^{(0)}}$ of the SCPV. $\mathbf{z}^{(0)}$ is sampled from diagonal Gaussian $\mathcal{N}(\mu_{\mathbf{z}^{(0)}}, \sigma_{\mathbf{z}^{(0)}}\mathbf{I})$ and served as the latent variable to fit the distribution. ε is a small constant vector that has been introduced to avoid numerical issues during the training [39]. SoftPlus is defined as $\text{SoftPlus}(\mathbf{x}) = \log(1+\exp(\mathbf{x}))$.

For the next step, posterior flow allows the network to learn more complex patterns of the trace. The input is $\mathbf{z}^{(0)}$ and after passing length of K flow it will become as $\mathbf{z}^{(K)}$.

Then, $\mathbf{z}^{(K)}$ will be passed into the decoder network to extract hidden features. Similarly, the purpose of those hidden features is to derive the mean $\mu_{\mathbf{x}}$ and standard deviation $\sigma_{\mathbf{x}}$ of the input trace vector. After that, the reconstructed $\mathbf{x}$ will be sampled from $\mathcal{N}(\mu_{\mathbf{x}}, \sigma^2_{\mathbf{x}}\mathbf{I})$

C Inference

When a new trace is given, the log-likelihood value will be computed against the model to detect whether the trace is abnormal or not. If the trace $\mathbf{x}$ is significantly different than the normal trace, the value of a trace $\log p_\theta(\mathbf{x})$ is noticeably smaller than the value of the normal traces. Instead of manually setting the threshold of anomaly, we follow the work from Liu et al. [39] and use Kernel Density Estimation (KDE) [22] to learn the distribution of the normal traces log-likelihood. Specifically, we adopt the p-value [57] approach and set the value as 0.001 to check if the probability of the log-likelihood value not following the learned distribution.

If the trace contains any unseen *call chain*, it will be regarded as abnormal. Training is a continuous process since the code evolves and the call paths keep changing over time. We use a sliding window of last 14 days of trace to keep our model up-to-date.

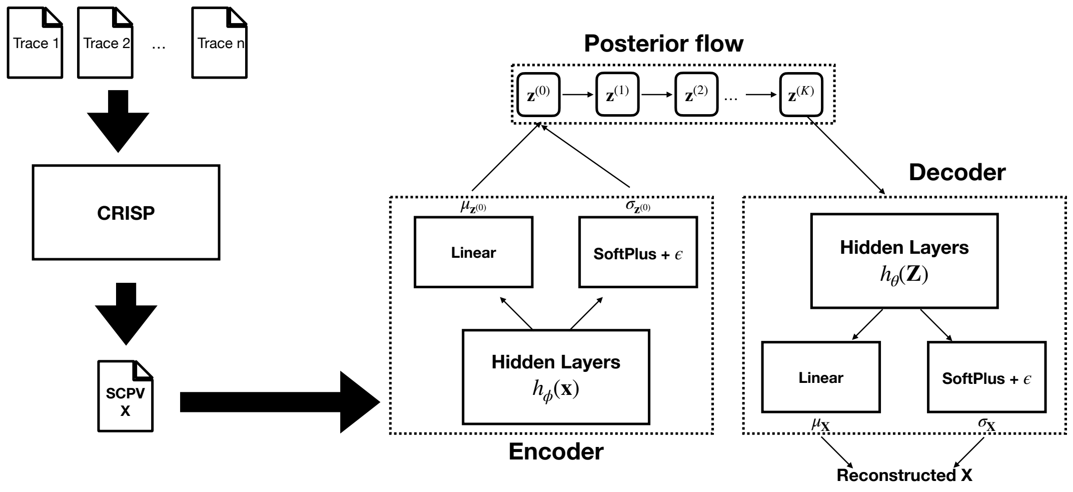


Figure 22: Architecture of neural network for anomaly detection.

Artifact Appendix

Abstract

This artifact includes the script to utilize CRISP as we presented in the paper.

Scope

The artifact allows: top-down analysis, bottom-up analysis, and preprocessing for anomaly detection. It does not come with any traces to analyze and those traces need to be provided by the user.

Contents

It contains the implementation of CRISP with corresponding script to run the analysis.

Hosting

The artifact is available at https://github.com/uber-research/CRISP under **atc-2022** branch.

Requirements

- Python3.6 is recommended to run the anomaly detection. Otherwise, any python3 version should be fine.
- Git is also needed.
- "git clone https://github.com/NetManAIOps/TraceAnomaly.git" is required under the root directory in order to run anomaly detection.

Setup

- "`python3.6 -m pip install -r requirements.txt`" to install the dependency for CRISP.
- "`python3.6 -m pip install -r TraceAnomaly/requirements.txt`" to install the dependencies for TraceAnomaly.
- You may need to install protobuf if the requirements.txt doesn't work in TraceAnomaly by "python3.6 -m pip install protobuf==3.12.4".
- specify "`TRACE_DIR`" in "bottom-up.sh", "top-down.sh", and "preprocess.sh".

Top-down Analysis

Before running, you need to specify the input, output, serviceName, operationName, and processor number in `top-down.sh`. Make sure the output directory already existed. To run the analysis, simpling typing

```
mkdir top-down-result
bash top-down.sh
```

By default, the script will use all processors to run. You can change the processor number with "`--parallelism`" knob in `top-down.sh` script.

The result will be in `top-down-result` folder. It will represent Figure 2, Figure 11, and Figure 13 in the original paper. The number and shape won't be exactly the same given the trace and endpoints are different from the paper.

Specifically, flamegraph like Figure 2 is generated as `flame-graph-P50.cct.svg`, `flame-graph-P95.cct.svg`, and `flame-graph-P99.cct.svg`. `criticalPaths.html` is like the heatmap in Figure 11 and please open it in browser. The differential flamegraph like Figure 13 can be viewed in `flame-graph-P50vsP95.cct.svg`

Bottom-up Analysis

To use the artifact, run

```
bash bottom-up.sh
```

The figure will be generated under "`result-bottom-up/`" folder, which looks like Figure 13 ˜ Figure 20 from the paper.

Anomaly Detection

Data Preprocessing run "`bash preprocess.sh`" to run generate the data for anomaly detection. Note each training, normal, and abnormal data needs to be parsed twice as it is shown in "`preprocess.sh`". The reasons is that we need to know the total number of call path to generate the data.

Training Please refer to https://github.com/NetManAIOps/TraceAnomaly.

Result Parsing Please go back to root directory when parsing the results. The trained model and the predicted results will be in "`TraceAnomaly/webankdata/`".

To parse the results, run "`python3.6 parse-rnvp.py -i path_to_rnvp_file`".