TraceWeaver: Distributed Request Tracing for Microservices Without Application Modification

Sachin Ashok
University of Illinois
Urbana-Champaign
sachina3@illinois.edu

Vipul Harsh
University of Illinois
Urbana-Champaign
vharsh2@illinois.edu

P. Brighten Godfrey
University of Illinois
Urbana-Champaign and Broadcom
pbg@illinois.edu

Radhika Mittal
University of Illinois
Urbana-Champaign
radhikam@illinois.edu

Srinivasan Parathasarathy
IBM Research
spartha@us.ibm.com

Larisa Schwartz
IBM Research
lshwart@us.ibm.com

ABSTRACT

Monitoring and debugging modern cloud-based applications is challenging since even a single API call can involve many interdependent distributed microservices. To provide observability for such complex systems, distributed tracing frameworks track request flow across the microservice call tree. However, such solutions require instrumenting every component of the distributed application to add and propagate tracing headers, which has slowed adoption. This paper explores whether we can trace requests *without* any application instrumentation, which we refer to as request trace reconstruction. To that end, we develop TraceWeaver, a system that incorporates readily available information from production settings (e.g., timestamps) and test environments (e.g., call graphs) to reconstruct request traces with usefully high accuracy. At the heart of TraceWeaver is a reconstruction algorithm that uses request-response timestamps to effectively prune the search space for mapping requests and applies statistical timing analysis techniques to reconstruct traces. Evaluation with (1) benchmark microservice applications and (2) a production microservice dataset demonstrates that TraceWeaver can achieve a high accuracy of ~90% and can be meaningfully applied towards multiple use cases (e.g., finding slow services and A/B testing).

CCS CONCEPTS

• **Networks → Network monitoring**; **Network measurement**.

KEYWORDS

Non-Intrusive, Distributed Tracing, Graph Analysis, Microservices

1 INTRODUCTION

Modern "cloud native" applications built using a microservice architecture split their functionality into small logical units, which can be deployed as containers and automatically scaled (i.e., dynamically replicated to meet demand) with cluster management platforms like Kubernetes. Compared to monolithic applications, individual microservices are more manageable to build and maintain, easier to individually re-deploy in new versions, and easier to write in different languages and frameworks. The result is that modern applications are highly distributed, potentially involving hundreds or even over a thousand [47] individual microservice instances.

Unfortunately, such highly distributed applications are challenging to operate and debug [16, 18]. For example, an operator may wish to determine which service is responsible for inflating latency for a certain user-facing API call for a specific subset of traffic. But an API call may produce *children*, i.e., further API calls to other services issued in order to handle the parent call. The resulting tree of API calls may ultimately comprise many stages of backend calls to various services, making it hard to isolate the component responsible for a particular problem. For such troubleshooting, it is indispensable to have a **request trace**, which we define as a record of each request (API call) into a microservice, and recursively all requests that it spawns (children, children of children, etc.) to other microservices. That is, each record within the trace includes a start and finish time (known as a "span") and pointers to its child request records.

Distributed tracing frameworks such as Jaeger [35] and Zipkin [63], along with commercial solutions like Instana [33] and Datadog [23] and earlier research proposals like XTrace [27], help developers deal with the above problem. Because network communication is externally visible, it is easy for such frameworks to automatically log requests between microservices as isolated events, i.e., *individual spans*. But to produce full *request traces*, such systems require application developers to instrument their code, carrying identifiers that associate each request with its children, which is known as *context propagation*. Even within a single service, instrumentation is needed at a module level to track execution flow, requiring a context object to be passed as an argument when modules are invoked [44].

This application-level code modification is required for request tracing, no matter what underlying communication framework is used by the application (Apache Thrift [1], gRPC [2], etc.). Only the application-level modules can track the execution flow (i.e. which

Sachin Ashok, et al.

incoming request spawns which outgoing requests). This context-specific information cannot be automatically made available to the underlying communication frameworks without any application support. The code modification for context propagation can involve a considerable investment of time, given dozens or hundreds of microservices written by different teams. Some microservices can be more challenging (e.g., legacy apps) or completely impossible to modify (e.g., proprietary, binary code), resulting in incomplete request traces. Even when feasible, adding tracing instrumentation is often buried under a long list of feature requests. This is one major factor[1] that has limited adoption.

In this paper, we pose the question: to what extent can we accurately produce request traces without instrumenting applications? We refer to this problem as **non-intrusive request tracing**.

There is a large body of work on distributed tracing, that tackles a different, but related, problem – inferring that service X tends to depend on service Y, either using individual span-level information [15, 20, 48] or relying on request traces generated via context propagation [21]. In contrast, with non-intrusive request tracing, we seek finer-grained information about which specific requests resulted in which other specific requests, without relying on any application support (context propagation). This is usually more valuable (e.g., to debug problems with specific requests, or important groups of requests) but also fundamentally more challenging. Applications concurrently serve many incoming requests, making it difficult to determine which incoming requests led to which child requests.

Non-intrusive request tracing therefore remains an unsolved problem that is highly desired by industry [14]. A few existing proposals do attempt to tackle this problem [51, 54], but make several simplifying assumptions about the application threading models, which severely restricts their applicability. Our work, in contrast, seeks a more general solution that is not tied to a specific threading model.

Our approach is based on two observations. First, while it might not be possible to construct fully accurate request traces without leveraging application support or without making restrictive assumptions about the application's threading model [51, 54], even *approximate* request tracing (with good enough accuracy) can be highly useful. If accuracy is reasonably high, or if the user is presented with a small number of possible traces, one of which is indeed correct, this can dramatically accelerate performance debugging compared with having no information about a request's trace. Moreover, many profiling tasks don't require every *individual* trace to be reconstructed correctly; often, users want to understand a *sub-population* of requests – for example, what is the per-service processing latency for requests in the worst 5% end-to-end latency bracket? Or, what is the typical performance profile of a high profile user's latency-sensitive query type? These questions may be meaningfully answered even if individual traces have some occasional error.

Our second observation is that modern microservice environments provide new avenues for collecting useful information that we can leverage for non-intrusive request tracing. In particular, even though we treat the application itself as unmodifiable, we can see detailed information about the communication in and out of the application, e.g. through service mesh sidecars or eBPF hooks. We

detail how such tools can be effectively used to obtain span information (map an incoming request to the corresponding outgoing response, and get the request-response timestamps). Moreover, modern packaging of applications for Kubernetes environments makes it feasible to spin up applications in test environments, where they can be observed in a controlled way. We can leverage such test environments to infer the call graph, i.e. the sequence of backend services that an incoming request at a given service invokes.

We develop a system, TraceWeaver, that obtains the above information (span timings, call graph) and uses it for non-intrusive request tracing with usefully high accuracy. TraceWeaver is centered around a trace reconstruction algorithm that combines constraints obtained from the knowledge of the call graph and span timings to prune the candidate search space, along with soft statistical timing heuristics to map the incoming requests at each service with likely potential child requests. In particular, we estimate the timing distributions between parent and child requests, use it to score feasible parent-child mappings, and then select the highest-scoring feasible mapping by encoding it as a maximum independent set problem. But there is a chicken-and-egg problem – how do we even know the timing distribution between parent and child requests without knowing the mapping? We tackle this using an iterative joint estimation process to discover both the timing distribution and the mappings, which quickly converges to an accurate result. Finally, we augment our approach to accommodate a limited extent of dynamics in the call graph.

In our evaluation using applications from the DeathStarBench [28] benchmarking suite, we find that the best performing baseline approach has 70% accuracy in reconstructing traces, while our techniques boost this to 93%. Our preliminary analysis on a production dataset from Alibaba [12] running several customer-facing applications showcases a high accuracy in the range of 80%-99% even under variable, high system loads. Even though this is imperfect, Trace-Weaver's trace reconstruction may already be useful for a variety of tasks. We demonstrate two use cases: determining which back-end service caused a particular subset of API calls to be slow, and detecting changes in a service's performance profile while A/B testing.

However, we believe this is not the end of the story: we expect accuracy could be pushed increasingly high, in increasingly difficult environments, by incorporating more information into our reconstruction framework (e.g., commonalities in request contents). We describe such necessary and promising directions in future work (§7). Overall, we believe non-intrusive request tracing is a promising way to provide developers with a "free" debugging tool, without additional effort on their part. This work does not raise any ethical concerns.

2 PROBLEM DESCRIPTION

2.1 Terminology

We refer to Figure 1 (representing a microservice based application) to explain the terms used throughout the paper.
Span. A span is a request-response pair that represents one execution of an API call at a running service instance, with metadata comprising caller, callee, start time, end time, and API endpoint (a specific URL where clients send requests to interact with a server's functions [56]). In the above example, (R_1, R_8), (R_2, R_5) etc., are spans.

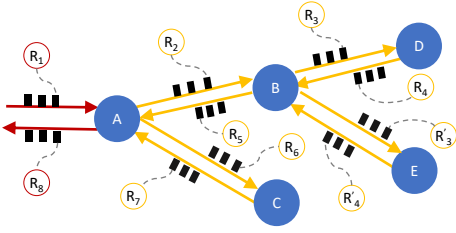[1]Others include cost of commercial solutions and performance overhead.

Figure 1: Call graph for an example application.

Call graph. At each service X, the call graph describes which other services are contacted by X to answer an incoming request. In Figure 1, as per the call graph, A talks to B and C to handle requests.

Dependency order. At each service X, we refer to the order in which backend calls are made as the *dependency order* at X. For the above example, the dependency order at A is the following – to respond to an incoming request, A calls B and C *sequentially* and once it receives a response from C, it returns a response. Similarly, the call graph at B is – to answer a request, B calls D and E *in parallel* and returns a response once it hears back from D and E.

Parent-child relationship. A parent-child relationship $R_1 \rightarrow R_2$ indicates that in the course of processing R_1, the microservice invoked R_2. In the above example, in order to process and respond to the incoming request R_1, A calls service B (R_2). Once it gets a response R_5 from B, it calls service C (R_6). Once it receives the response R_7 from C, it does some processing of its own and returns the response R_8 to the user. This makes the requests R_2 and R_6 children of R_1. Likewise, at Service B, follow-up requests R_3 and R'_3 ($'$ implies parallel calls) are children of the corresponding request R_2.

Request trace. A trace of a request r includes r and its response along with the records of the full tree of its descendants. The record for each request within the trace includes its span information (i.e., the caller, callee, API endpoint, start, and end time) and pointers to its child request records. In the above example, the root span starts when request R_1 arrives at front-end service node A (generally invoked by an external client), and ends when the corresponding response R_8 is returned. The remaining spans involve subsequent request-response pairs, e.g R_2-R_5, R_3-R_4, etc. which are child spans of the R_1-R_8 pair.

What is visible? Of the information listed above, we can readily obtain the span metadata (through eBPF, service mesh sidecars, etc). We can further infer the call graph (from production data or isolated test environments). We detail how such information can be obtained in §5. However, the parent-child relationships, and therefore the request trace, are not visible. That is the crux of the problem in this paper, as we discuss in more detail next.

2.2 Request Tracing

The key problem we target in this paper is whether we can construct complete *request traces*, i.e. map each request arriving at each service with the corresponding child requests. In the context of our example, we wish to map R_1 to R_2 and R_6 at A; R_2 to R_3 and R'_3 at B, and so on. We refer to this problem of obtaining individual request traces for all requests as *request tracing*.

2.2.1 Why is request tracing useful? Tracking the journey of a request through several components of a distributed application, that work in tandem to generate a response, is critical for various debugging and troubleshooting tasks. For example, let's say an operator is interested in knowing which microservice is majorly contributing to the delay for a small set of high priority client requests. To produce the answer, one must be able to map which backend requests were made to produce the response, for each of those high priority requests at each service. We detail more such use cases in §6, including A/B testing.

2.2.2 Why is instrumentation-based request tracing hard? One way to keep track of a request's journey through the application is to instrument all service components so that a unique *context identifier*, attached to an incoming request, is carried onto any requests to backend services that result from the incoming request. For example, R_1 carries a unique identifier that service A propagates to R_2, from which service B propagates to R_3, and so on. This is known as *context propagation*, and has been extensively explored through an active line of research [27, 35, 36, 39, 52, 63]. Given support for context propagation, spans from each microservice can be grouped by their context identifiers and spans with the same identifier can be stitched together to form a request trace.

Such context propagation must be inherently supported in the application logic, since only the application can track the execution flow across function boundaries. It cannot be provided as a plug-in feature by the underlying frameworks (e.g. the service mesh or RPC frameworks) which do not have the required visibility into the application logic and its internal execution flow (see Figure 2a).

Although copying identifiers seems simple at first glance, it involves major practical hurdles: all software components, maintained by different teams or different vendors, have to appropriately propagate context. This involves agreeing on an API to use consistently, but more importantly, instrumenting internal microservice code to carry identifiers across function calls and data structures.

OpenTelemetry [43] defines a standard API and distributed tracing frameworks like Jaeger and Zipkin (and other commercial solutions) provide instrumentation tools for context propagation, either in common libraries or via hooks that developers can explicitly call. However, it is up to individual app developers to use the APIs or leverage the available tools, and context propagation continues to be a difficult task that increases app development burden. As a result, it has seen limited adoption, especially for legacy applications which require significant developer effort to modify existing code (only 20% of enterprise applications, as of 2021 [53]). Furthermore, an application could use proprietary third party service components which may not be possible to instrument.

If software components have not yet been instrumented to propagate context, one could use program analysis techniques to automatically modify software components so that they pass request context. This approach, taken by systems such as [25], is error-prone and still requires developers to go through the changes suggested by the tool and approve them. Other approaches, such as Border-Patrol [37], require middleware instrumentation to modify event streams to run them sequentially at specific observation points for precise mapping, which results in high-performance penalties (10-15%) and makes them infeasible.

2.2.3 Non-intrusive request tracing. In the absence of context propagation, another strategy to construct request traces is to infer it based

on the available information. Specifically, one can match incoming requests at a service, say A, to outbound requests from A based on span metadata, thread-level data, or header parameters. One benefit of this approach is that it requires very little from app developers. This approach, which we refer to as *non-intrusive request tracing*, is the subject of our work.

2.2.4 Existing approaches for non-intrusive request tracing. Existing works such as vPath [54] and DeepFlow [51] solve non-intrusive request tracing for a restrictive set of applications assuming a specific threading model for the application. Specifically, for the threading model, it is assumed that there are no hand-offs between threads and no asynchronous calls, so that every outgoing request from a thread can be mapped to the most recent network event (request/response) picked up by that thread. Using this assumption, vPath (and DeepFlow) can record the thread t that picked up a request r_{in} and for any subsequent request r_{out} sent by t, until t picks up the next request, associates r_{in} with r_{out}. While this scheme is effective for applications that follow this threading model, these assumptions do not hold for several common application types – applications that hand over requests to communication threads of RPC libraries such as Thrift [1] or gRPC [2], or applications which employ event-driven, asynchronous communication to perform non-blocking I/O operations such as in Node.js [8] apps. For example, as illustrated in Figure 2b, due to non-blocking disk I/O behavior (1b, 2b) in the second case, the "next" (1st) outgoing event is an outgoing backend request (1c) due to the 1st request (1a) even though 2nd request is the last outstanding incoming request (2a). vPath (and DeepFlow) would incorrectly map 2a to 1c in this case.

2.2.5 Employing sidecars, eBPF hooks, or having access to the call-graph alone is insufficient. While infrastructure technologies like service meshes [34] and eBPF [26] can be valuable for observability, they are insufficient to solve the tracing challenge. Service mesh sidecars handle communication functions [13] such as load balancing, and retries and can intercept all traffic, enabling request observation/modification without altering the app code. eBPF, on the other hand, can run programs in the OS kernel, allowing monitoring/manipulation of system operations, such as network syscalls, with minimal overhead. Despite their usefulness, they alone cannot solve the tracing problem because if a sidecar (or an eBPF program) adds custom headers to an incoming request (e.g., as shown in Figure 2a), there is no guarantee that the application will propagate these headers to related outgoing backend requests. Without the propagation of these headers, linking the requests becomes impossible. Additionally, merely having access to the call graph – the set of backend services and their order queried by the application for a request – is inadequate as well. This limitation arises because multiple requests can concurrently traverse the same call graph, making it challenging to disambiguate requests solely based on call graph knowledge.

2.2.6 Our approach. We seek a more general solution for non-intrusive request tracing that is not restricted to specific threading models. Our solution, TraceWeaver, obtains visible information (call graph and span timings) using various avenues provided by modern microservice environments (e.g. eBPF hooks, sidecar proxies,
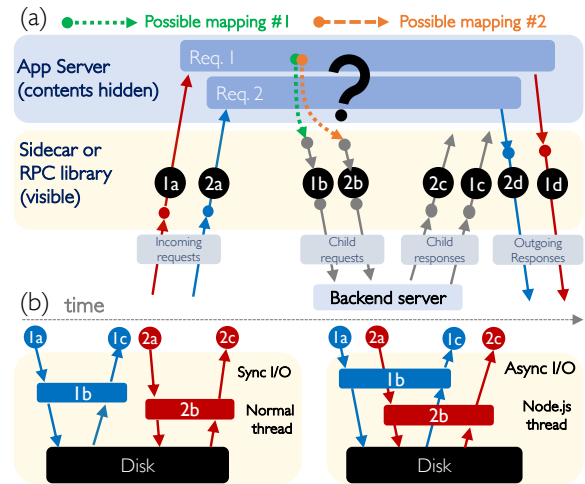

Figure 2: (a) Request forwarding chain shows how even a proxy or RPC library cannot discern the right mapping (1 or 2) as it is buried in app logic. (b) Non-blocking I/O in the second case (right) results in vPath/ DeepFlow's assumption (last incoming network event causing next outgoing network event in a thread) to break.

test environments etc). It then applies a novel timing analysis algorithm to construct *approximate* request traces from this information. TraceWeaver can be applied to each service individually, and can also co-exist with other solutions for non-intrusive request tracing (e.g., if a partial trace is available because some services propagate context or some services satisfy the assumptions of DeepFlow/vPath, TraceWeaver could fill in the gaps to complete the trace).

2.3 Related Problems: How they differ?

Several works under the umbrella term "distributed tracing" solve a variety of problems distinct from request tracing. One such problem, that multiple existing works try to tackle, is to obtain the service-level dependencies such as service A calls service B to answer inbound requests at A but not vice-versa. We refer to this problem of obtaining dependencies between services as "dependency mapping". The Mystery Machine [21] is a system for dependency mapping. It assumes context propagation and uses the request-traces thus obtained to derive a model of how services talk to each other. As discussed in §2.2.2, context propagation is burdensome for developers and hence has limited adoption in the real-world. Orion [20], Sherlock [15] and WAP5 [48] also tackle dependency mapping – they take as input span level data, and obtain the dependencies between services via analyzing delays in network traffic between when A receives a request and when A talks to B.

In contrast to dependency mapping, for request tracing, we seek finer-grained request traces, linking a specific request inbound at service A to another specific request outbound from A (and similarly at other services). Request tracing is more valuable than dependency mapping since request traces can be used to analyze performance of a single request or, more generally, any subset of requests. At the same time, disambiguating between far more numerous possibilities
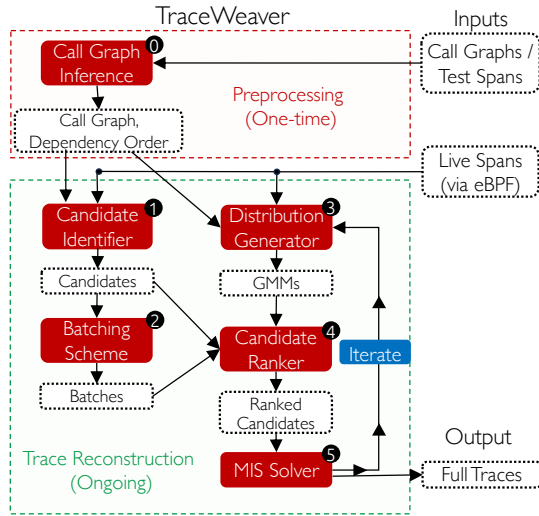
Figure 3: The distinct phases in TraceWeaver's system.

for request traces for every single request is fundamentally more challenging than finding which other services a particular service depends on. We tried to repurpose one of the above dependency mapping systems, WAP5, for non-intrusive request tracing and found its accuracy to be low (§6). Sherlock and Orion employ similar analysis, hence we expect them to have the same problems as WAP5.

Many systems take as input request traces, and analyze them to improve various aspects of the system. For instance, Snicket [17] optimizes sampling of request traces based on developer queries for efficient storage. [62] uses logs from services, tagged with identifiers, to learn models that can help in various storage related decisions. DQBarge [22] injects critical system information (e.g. load) onto requests, which is passed around so that services can optimize for quality/performance trade-off. All of the above systems assume capabilities similar to context propagation, but they can be used in conjunction with non-intrusive request tracing which can make request-traces available for their analysis.

3 TRACEWEAVER OVERVIEW

We enumerate the components of TraceWeaver (Figure 3):
Obtaining the inputs. To reconstruct traces, TraceWeaver requires live span data from running apps and the call graphs the spans follow, along with the dependency order (§ 2.1). TraceWeaver uses eBPF to hook onto networking syscalls (e.g., accept, recv, send, close) to collect live span information (caller, callee, API endpoint, start time, and end time). The call graphs and the dependency order can be provided directly by the operator (optional) or deduced by running tests on spans collected in a test environment. §5 describes more details.
Preprocessing. If operator-provided call graphs are available, TraceWeaver identifies dependencies by analyzing them. In their absence, spans generated in a test environment are used. Using these spans, TraceWeaver can learn two distinct types of dependencies regarding the application at each service S: (a) call graph: which other backend services does S call and (b) dependency order: the sequence in which S calls these backend services. These dependencies serve as constraints

which allow TraceWeaver to identify *candidate* mappings (the set of outgoing requests whose timings respect dependency requirements) for each request. Note, the preprocessing is run once and re-run only if the application is updated. We provide more details in §5.
Trace Reconstruction. Under large enough request arrival rates, each incoming request at service S can map to many different candidate outgoing requests for each backend service S calls. Our trace reconstruction algorithm (detailed in §4) scores and ranks these candidates to find the *best* mapping.
Using the output. While the output traces can be inspected individually for insights, learning aggregate statistics about application performance is another class of use case that is extremely useful in practice. For this, an operator specifies a filter which selects a subset of mapped traces that collectively form an "aggregate" trace, representing that subset's behaviour. A large class of use cases which leverage such aggregate traces can be derived from such a setup. For example, analyzing subset of traces that suffer tail latency to identify which straggler service(s) are responsible for delay inflation. Another interesting example is running an A/B test where comparing the performance of a subset of traces that used version B instead of A is useful. We showcase these use cases in §6.4.
Deployment modes. TraceWeaver allows for both offline and online deployment modes. In offline mode, spans are collected and stored to analyze on demand. In online mode, spans are passed to a running TraceWeaver instance which constructs request traces in real-time (detailed in §5.3).

4 TRACEWEAVER ALGORITHM

Inputs. Our reconstruction algorithm uses the following inputs (we detail how we obtain them in §5):
(1) Span information comprising the request-response pairs (e.g., R_1-R_8 pair from Figure 1) and the timing information.
(2) The call graph and the dependency order at each service.
Assumptions. TraceWeaver makes the following assumptions: (i) A parent span's response is sent out only after all child spans finish processing. (ii) Every successful request has a response or a completion acknowledgement. In other words, we assume request-response behaviour, commonly enforced by REST and gRPC endpoints. (iii) It is possible to know which unique call graph a given (completed) request falls into (e.g., via API endpoint, unique header parameters). (iv) Once identified, call graphs are either static with a well-defined structure or can display dynamism by traversing only a subset of the graph (e.g., due to caching, failures, or semantic reasons). We handle this class of dynamism in §4.2. We leave tackling other forms of dynamism, e.g., due to retries and quorums, to future work (§7).

For ease of exposition, we first describe our algorithm for the scenario where all spans follow the same call graph, i.e., no dynamic behaviour is induced due to caching or errors (§4.1), and then fold dynamism into it (§4.2).

4.1 Reconstruction for Static Call Graphs

We refer to Figure 1 to describe our algorithm and Table 4.1 lists all tunable parameters. For a given application, TraceWeaver decomposes the problem of reconstructing an entire request trace into solving independent optimization tasks at individual services i.e., in

Parameters	value
Max. size of an optimization batch	B = 30
Max. candidates for any given span	K = 5
Max. GMM components for modeling delay distribution	C = 5
Number of buckets used to estimate delay mean and variance	R = 10

Table 1: Parameters used in TraceWeaver's design.

Figure 1, mapping R_1-R_8 to R_2-R_5 at A is an independent optimization task w.r.t mapping R_2-R_5 to R_3-R_4 at B. Also, given that R_1 and R_2 must be received and sent by the same container of a service, the scope of each optimization task is limited to that granularity. Finally, the independently mapped pieces of the request trace can be trivially assembled in post-processing to form the full trace (since the outgoing R_2 at A and the incoming R_2 at B are the same and can be linked).

In essence, at each service S, TraceWeaver receives a dynamic number of spans (M) per time window and uses the spans and call graphs to do the following:

(1) Candidate identification: For each incoming span (a span whose request arrives at S, is serviced by S, and its response departs S) among the M spans, find candidate mappings using span timings and constraints derived from the call graph and the dependency order.

(2) Batching: Split the M spans into batches to restrict the scope of joint optimization to individual batches. Careful splitting minimizes the overlap of candidate mappings across spans in adjacent batches.

(3) Distribution generation: Estimate the "inter-span" time distributions as per the dependency order between the services in the call graph at S. In other words, if a dependency exists between R_1 and R_2 (or similarly, between R_5 and R_6), we estimate a probability distribution for the time between those events. In general, these probability distributions correspond to the time duration between the reception of an incoming parent request (child response) at S from a caller (callee) and the transmission of a dependent outgoing child request (parent response) at S to a callee (caller).

(4) Candidate ranking: Within each batch, score and rank candidate mappings for each span based on how well the timings of the candidate spans align, calculated using probability distributions from (3).

(5) Joint optimization: Solve each batch jointly, i.e., find an *assignment* of incoming spans to one candidate mapping each (by choosing among available candidate mappings), in such a way that the cumulative probability score of the batch of spans is maximized.

(6) Iteration: Repeat steps 3, 4 and 5 by feeding in the mappings from step 5 to step 3 to improve time distributions (step 3), candidate rankings (step 4), and the final assignment (step 5).

Step 1: Identifying Candidate Mappings. For each incoming span at a given service S, we define a candidate mapping as the set of candidate child spans, one for each backend service invoked by S as per the call graph, that is jointly feasible as explained below. We use timing constraints derived from the call graph and dependency order at that service to identify the set of jointly feasible candidate mappings for each incoming span. We refer back to Figure 1 to exemplify such constraints. Given that service A depends on B, we can constrain the set of child spans to B for the incoming parent span R_1-R_8 at A to the ones that satisfy the following criteria: (i) the child span's request R_2 was sent to B after R_1 arrived at A, and (ii)

the corresponding response R_3 from B for that child span's request arrived at A before the parent span's response R_8 left A. Likewise, we can constrain the set of feasible child spans to C. The dependency order (i.e. A invokes B and C sequentially) allows us to add additional constraints when considering R_1's candidate mappings for child spans at B and C – (iii) response (R_5) of a candidate child span must arrive from B at A before the request (R_6) of a candidate child span is sent from A to C. A set of candidate spans (i.e., a candidate mapping) satisfying all these constraints is deemed feasible.

Step 2: Creating Optimization Batches. To start, we split the M spans into non-overlapping batches for joint optimization. As step 4 will describe, TraceWeaver employs a solver to do the joint optimization. The intent of batching is to make using the solver feasible by limiting the size of the graph (proportional to batch size) passed to it. Our batching strategy employs the following algorithm to eliminate shared candidates across batches. At a high level, we try to make a cut (i.e., create batch boundaries) in such a way that no child spans satisfy constraints for parent spans in multiple batches; but we also limit the maximum size (B) if an appropriate batch boundary is not identified within that threshold.

(i) We sort all M incoming spans at the given service by start time, with ties broken by end time.

(ii) We consider every adjacent pair of spans (i, i+1) for a possible cut, where a cut denotes a batch ending with span i and the next batch starting with span i+1.

(iii) We make a cut if (a) the current batch size is larger than a threshold (B = 100) or if (b) (1) there are zero common candidates between span i+1 and the preceding span j with the latest end time (i.e., the span which finished last among all preceding spans 0 through i) and (2) the span j ends before span i+1.

This algorithm's runtime is $O(M)$. In Appendix A.2, we prove that the latter condition (b) ensures that span i+1 or any later span will not have any candidates in common with the previous batch.

Step 3: Constructing delay distributions. We construct delay distributions for caller/callee combinations per the call graph and the dependency order observed at S. For example, consider the scenario where service A (S = A in this case) initiates a call to service B and, upon receiving a response from B, subsequently calls service C as illustrated in Figure 1. In this context, we construct two distributions to encapsulate this scenario and the exhibited sequential dependence. These two distributions represent the time elapsed between (i) the arrival of a request at A and the subsequent transmission of the corresponding outgoing request to B and (ii) the reception of a response from B and the subsequent transmission of the corresponding outgoing request to C, respectively. Note, estimating such distributions is non-trivial since we naturally do not have the actual mappings between spans arriving at A and spans going from A to B, which could have given us the actual values of the processing delays. This is a chicken-and-egg problem: if the mappings were available, the distributions could be constructed trivially, but the distributions are needed to produce the inferred span mappings (the TraceWeaver output). Note that the real-time gaps between the incoming and the associated outgoing spans are not possible to obtain in a system without tracing (as this is the very problem TraceWeaver is attempting to solve). We resolve this by using a seed distribution followed by successive iterations, jointly discovering a distribution

and mapping that agree well with each other (as illustrated in Figure 9 in Appendix A.3). Such iterative techniques are commonly used in other domains for unsupervised optimization [32].

In the 1st iteration, we fit a Gaussian distribution for this delay t: $\mathcal{N}(\mu_{AB}, \sigma_{AB})$. Note that μ_{AB} can be estimated exactly without knowing the precise mapping. Since the mean of the differences equals the difference of the means, we can take the difference between the mean of arrival times of M external requests at A and the mean of departure times of M requests from A to B. To estimate σ_{AB}, we split M spans into $R = 10$ buckets and compute the empirical mean for each. We can calculate the approximate sample standard deviation ($\bar{\sigma}_{AB}$) across these bucket means (our samples). To obtain σ_{AB} (standard deviation of our population of M spans) we need to multiply that with $\sqrt{R}$ (as $\sigma_{AB} = \sqrt{n} * \bar{\sigma}_{AB}$ per central limit theorem [59], where n is no. of samples or bucket means). Note that this estimate is only approximate since σ_{AB} is computed over bucket means rather than individual data-points from the population (which we do not have). Nevertheless, this is sufficient to create initial "seed" distributions for the first iteration.

In subsequent iterations, the final output of the previous iteration (i.e., mapped spans) is available which gives us (our best estimate of) individual data-points. We use a much more robust Gaussian Mixture Model (GMM) [50] to fit these data-points using an expectation maximization (EM) algorithm [49]. A GMM is a powerful type of mixture model GMM(C), comprised of several Gaussians, each identified by $c \in \{1,..,C\}$, and parameterized by $\mu_{c,AB}$, $\sigma_{c,AB}$, and mixing weight $\pi_{c,AB}$. GMMs are known to be a universal approximator of densities [30, p. 65], given a finite number of Gaussian components C with enough parameters. To identify C, we run a sweep from $C = 1$ to 20 components and use the GMM (C) which minimizes the value from Bayesian Information Criterion (BIC) [58], a standard model selection tool used to mitigate over-fitting. Over successive iterations, the inferred mappings from previous iterations improve the delay distributions resulting in better future mappings. However, successive iterations that leverage a more complicated Gaussian Mixture Model only boost accuracy for scenarios where a simple Gaussian is insufficient to model the underlying delay distributions.

Step 4: Ranking candidate mappings. We use the delay distribution inferred in Step 3 to score each candidate mapping. Let Z denote a candidate mapping. In Figure 1, for service A, let incoming span R_1-R_8 be mapped to outgoing spans R_2-R_5 and R_6-R_7 according to Z. We define the score of mapping Z as

$$score(Z) = score_{AB}(t_1, t_2) + score_{BC}(t_5, t_6) + score_{CA}(t_7, t_8)$$

where t_1 is the time at which R_1 arrived at A, t_2 is the time at which R_2 was sent, t_5 is the time at which R_5 was received, t_6 is the time at which R_6 was sent, t_7 is the time at which R_7 was received, and t_8 is the time at which R_8 was sent. Note that all the candidate mappings obtained from step 1 are feasible, hence $t_1 <= t_2 <= t_5 <= t_6 <= t_7 <= t_8$ (or else Z would be infeasible). We set,

$$score_{AB}(t_1, t_2) = \log \mathcal{P}(t = t_2 - t_1 | \Theta) = \log \left(\sum_{i=1}^{C} \pi_i \, \mathcal{P}(t | \mu_i, \sigma_i) \right)$$

where $\mathcal{P}(t = t_2 - t_1 | \Theta)$ is the probability density function of the GMM described by Θ. Note, iteration 1 is just a special case of GMM with just 1 component such that $Score_{AB}(t_1, t_2) = \log \mathcal{P}[t = t_2 - t_1 | \mathcal{N}(\mu_{AB}, \sigma_{AB})]$. All possible candidate mappings are ranked using this scoring criteria.

Step 5: Joint optimization. Given the batch of spans where each incoming span has a set of candidate mappings, we cast the problem of finding the "best" mapping for each span as an optimization problem. The optimization attempts to find a set of mappings for the batch (one mapping per span) that maximizes the cumulative score of that set subject to two constraints: (1) an incoming span should only be assigned one mapping, (2) the same outgoing span must not be present in two different mappings. This construction turns out to be an instance of the multidimensional assignment problem which is NP-hard [41]. We propose an online algorithm (as illustrated in Figure 10 in appendix) which approximates the global maxima by finding the maximum independent set (MIS) and works as follows:
(i) We find the top $K = 5$ candidate mappings for each span.
(ii) Next, we transform each span's top K candidate mappings as vertices in a graph with weights proportional to their score.
(iii) Then, we add edges between vertices if they break constraints. That is, we add an edge between candidate mappings of the same span (to address constraint 1) and an edge between two candidate mappings that share a common outgoing span (to address constraint 2).
(iv) As each batch is fairly small, we can solve MIS exactly or nearly optimally using Gurobi [3], an off-the-shelf MIS solver.
(v) Once the mappings have been chosen for a batch, the outgoing spans of those mappings are deleted so that they cannot be assigned again to other incoming spans in subsequent batches. This only occurs if there are common candidates across batches, which we minimize via our choice of batch boundaries in Step 2.

4.2 Incorporating Dynamism

In section §4.1, we assumed the call graphs to be static. We now describe how TraceWeaver handles the predominant class of call-graph deviations, where requests can follow any valid subset of the usual call graph. This class of dynamism includes cases where some backend calls are not made due to (1) caching, (2) service failures, or (3) semantic reasons where the backend calls are unnecessary. To address such cases, we create a fuzzy version of our optimization that allows for a fraction of incoming spans to not have a complete mapping, i.e., their mapping does not have an outgoing span for each outgoing endpoint, thereby only traversing a subset of the call-graph. To identify and limit what fraction of such incoming spans are allowed this opportunity (to control fuzziness), we look at the externally observable discrepancy between what we expect given the call graph and what we observe in reality (e.g., instead of 1000 outgoing spans, we observe only 900). We fill up such discrepancies using phantom "skip" spans, such that the optimization can use them to fill in the gaps (e.g., span W $\rightarrow$ span X $\rightarrow$ skip span $\rightarrow$ span Z). In addition to limiting the number of skip spans, it is crucial to allow an incoming span opportunity to use it only according to its likelihood of having encountered dynamism. To realize both, we run the following algorithm:
(1) First, we calculate the total discrepancy between incoming requests and outgoing requests for each backend service over a large window (e.g., 10 secs) so that the observed discrepancy is not incidental due to the window being too small to see outgoing spans. This value is set as the total budget and skip spans are created accordingly.
(2) Next, each optimization batch is analyzed to estimate the max. degree of dynamism it could have encountered. This value directly corresponds to how many skip spans it may need at most. This value

has max quota $Q = X - Y$, where X is the total number of outgoing spans needed for this batch, and Y is the number of spans that can only be assigned to this batch given the constraints.
(3) Given the total budget and max. quota for each batch, we distribute skip spans to each batch using a water-filling algorithm [60]. This step iteratively distributes the spans to the most needy batches (i.e., with the highest Q values), stopping only when it runs out of total budget. This step also ensures that whatever error in estimation we make in calculating the total budget is distributed across batches.
(4) Finally, we run the same joint optimization described earlier with each batch having access to a certain number of skip spans as allocated previously. Also, we must adjust how we estimate the initial seed delay distributions if there are skip spans as the exact mean cannot be calculated as in §4.1 step 3. Hence, to initialize our optimization, we use delay distributions built on mappings obtained by running WAP5 [48]'s approach (described in §6)). Note this is only for the 1st iteration and for successive iterations, GMMs are leveraged to produce fully mapped traces in the usual manner.

5 SYSTEM DESIGN AND IMPLEMENTATION

Our prototype for TraceWeaver is implemented as a process which continuously ingests span info. and outputs the mapped spans (i.e., full traces). The key inputs required by TraceWeaver include span-level info. and the call graph with the dependency order. We discuss how we obtain these inputs in our system as well as alternate options.

5.1 Obtaining live span information

5.1.1 Capturing raw data.

(1) eBPF hooks (our approach). For a running service, we can employ eBPF hooks to obtain info. for requests/ responses of each parent and child span arriving at/ departing from that service. For instance, we implement the following eBPF-based approach for HotelReservation [28], a gRPC-based app used in our evaluation. Using eBPF, we hook onto networking syscalls (e.g., accept, send, recv, close) for relevant processes (filtering on PID), which enables us to capture info. on the wire without directly interfacing with or instrumenting the app.
(2) Alternatives: (a) In deployments running service meshes, span data is already available with sidecar proxies, which route requests/ responses via itself using iptable rules. These proxies intercept the HTTP connections and have visibility into the request and response headers, allowing them to gather span information (e.g., request/response mappings, timings). (b) Tools like Wireshark [61] and tcpdump [55] are also sufficient to capture raw network traffic.

5.1.2. Parsing and mapping requests/ responses.
Sidecar proxies and tools like Wireshark already contain suitable protocol analyzers or adapters, which allow for parsing serialized raw byte stream data. For example, raw data can be decoded to construct gRPC requests and map them to their corresponding responses by understanding the structured data encapsulated within the raw stream. For the eBPF-based approach, we can run a similar adapter as a separate user-space process, which ingests raw data from the kernel eBPF component to parse and produce the same output. For gRPC, one of the app layer protocols we tested on, we now describe an interesting challenge. Capturing requests/ responses from gRPC, which is built on top of HTTP/2 [6] (the next generation of HTTP protocol), can be tricky. This is due to the stateful header compression scheme (HPACK [5]),

which makes request/ response headers hard to parse. The HPACK-based compression feature involves the client and server maintaining a dictionary, mapping previously seen headers to unique numeric codes, and only passing these codes in further communication. Without access to this dictionary, we need to resort to sniffing from the beginning of a long-running HTTP connection in order to not miss important connection information available only at the initial phase. Also, the gRPC adapter must use a "man-in-the-middle" style approach to keep track of the dictionaries being actively updated at the client/ server. Our adapter uses standard HTTP libraries [7] to continuously ingest and parse this data, which allows us to read the plaintext HTTP header and payload. We currently do not handle encrypted traffic, but existing software solutions like Pixie [9], built on eBPF, have built-in mechanisms to trace the API between the application and the SSL/TLS library to capture it before it is encrypted.

5.2 Obtaining call graph and dependency order

5.2.1 Generating test spans.

(1) Test environments (our approach): To learn the call graph associated with a service, we can replay production traces (identified by query parameters or API endpoints) in a test environment. We do this one trace at a time so that the ensuing spans can be easily weaved together to form "test traces" due to the lack of competing candidates. We additionally apply a large artificial delay using Linux TC rules on each observed outgoing span to create varied examples. This last step ensures there is enough variety in these test traces to prevent ambiguity in learning dependencies. For example, in Figure 1, it can be tricky to determine whether A calls B and C serially or just appears so due to the quick completion of B's invocation. If delaying B's invocation also delays C's invocation, it would indicate serial ordering. Note that production traces could be clustered and a subset of representative requests from each cluster could be executed in isolation to reduce execution time. We leave this to future work.
(2) Alternatives: (a) If test environments are not feasible, test traces can also be generated from production environments by spinning up a tiny replica for the service of interest and re-directing one request at a time to it. This would also provide us with test spans which can be mapped together trivially. (b) A large volume of production traces can also be analyzed to find test traces, given that there might be periods of low load where the mappings are obvious and constructing traces is trivial. We leave the exploration of these options to future work.

5.2.2 Inferring call graphs and dependency order.
We use test traces to learn the call graph and dependency order using the analysis below. For each set of test traces which cover the same path, we model all the services it traverses as vertices in a graph. The aim is to obtain a graph with directed edges which would indicate dependencies. At service A in Figure 1, A and B are endpoints for incoming and outgoing spans. Here, A→B would indicate that an incoming span's request at A must precede an outgoing span's request to B. Similarly, since B and C are both outgoing endpoints, B→C would indicate that B's invocation must conclude (i.e., its response must come back) before C's invocation. For this graph, we initially add directed edges between every vertex pair (as every dependency is possible). By analyzing the traces from the test dataset, we remove the corresponding edge from the graph for every violating example of a dependency. Example, if a trace indicates that B finishes execution before or during C, we

remove C→B. By iterating through all test traces, we obtain a graph with edges representing genuine dependencies, which we can use as constraints. Similar approaches are espoused by [21, 40] to construct causal models for request execution which can also be leveraged.

5.3 Deployment modes

TraceWeaver allows for the following deployment modes.

(1) Offline: In this mode, span info. is collected from apps at runtime, and stored for later analysis. When an operator requires trace-level insights for a specific time period, TraceWeaver can selectively run the algorithm on spans from that period to produce the required traces.
(2) Online (enables tail-based sampling): In this mode, span info. is collected at runtime and sent to a live, running TraceWeaver instance which reconstructs traces in real-time. This online deployment mode of TraceWeaver also enables trace sampling. Sampling is an important use case for distributed tracing, where to reduce overhead, only a certain percentage of API calls are traced, but each is traced across its whole call tree using trace IDs. In this deployment mode, *all* requests and responses are collected *temporarily* by TraceWeaver for mapping. Next, TraceWeaver maps all the requests using the optimization algorithm and produces the fully mapped traces. Now, sampling can be done by the appropriate logging agent (either TraceWeaver itself or by interfacing with the app) at any given operator sampling rate and the rest of the traces can be deleted. This approach requires temporary storage of all spans for each sliding window of time (e.g., 1-5 seconds) where the window needs to chosen based off the known response latency distribution of the app (e.g., such that all plausible candidate mappings are captured within it). We evaluate the runtime performance of this online deployment setting in §6.

6 EVALUATION

6.1 Setup

We evaluate on benchmark apps from the DeathStarBench[28] suite – HotelReservation and Media Microservices (Fig. 7 & 8 in Appendix A.1), a Node.js based microservice demo[11], and on production traces from the Alibaba cluster dataset[12]. HotelReservation, Media Microservice, and Node.js apps comprise 6, 14, 7 different services respectively excluding a variable count of caching and database components (Memcached, Redis, and MongoDB). The microservices are orchestrated using Docker [24] and Kubernetes [38] and are run on 3 VMs running Ubuntu 16.04 (each provisioned with 4 cores and 4 GB of RAM). All VMs run within a bare-metal 32-core Intel Xeon Silver server. We use the wrk2 [57] load generator to create requests that hit each app's frontend (to generate test as well as live spans). We employ Jaeger [35] to collect ground truth of the end-to-end traces.
Baselines. We compare TraceWeaver against the following:
(i) WAP5 [48]: It tackles a weaker problem of dependency mapping by using the span-level information to compute the probability with which services invoke each other. For each child request r, WAP5 assigns it to the most recent parent request that is not yet assigned (or leaves it unassigned if there's no parent within a specific time window). Depending on the service the child request was transmitted to and the service the parent request was received from, appropriate frequency counters are updated, and the probability distributions are generated. Then, all such probability trees at each service are

aggregated to obtain the overall pattern, determining the probability that a service A calls another service B to process its requests. We re-purpose the tree-building algorithm of WAP5 for request tracing, picking the most likely child span requests as the mapping for the parent span's request r. While this version of WAP5 also employs mapping spans using probability scores, TraceWeaver differs from it considerably owing to techniques mentioned in §4: constraint-checking, near-perfect batching, GMM-based ranking, and joint optimization.
(ii) vPath [54]/ DeepFlow [51]: As previously mentioned, vPath assumes a specific threading model where threads pick up requests and process them to completion before switching to the next request. It assumes no request handoffs between threads, no async calls, and assumes access to the thread ID for the thread processing a request. While this can effectively stitch spans at services where these assumptions hold, it fails at services where it does not. DeepFlow employs identical assumptions so they're represented by our implementation of vPath in the results. For the tested apps which do request handoffs between threads internally, we do not have access to the ID of the final processing thread (we only have the gRPC thread ID that picked up the request) so DeepFlow/vPath is in-applicable. Similarly, no thread IDs are available in the Alibaba production dataset (possibly due to the nature of the apps as well as due to the overhead of capturing thread IDs). So in such cases, lacking thread IDs, vPath/DeepFlow is made to assume that all requests are handled by the same thread.
(iii) FCFS: We also construct a strawman where external requests arriving at a service A are assigned to the outgoing requests at service B based on their arrival and departure orders which works well if requests are processed in order with limited or no parallelism.

6.2 Benchmark applications

6.2.1 Accuracy vs. load. We begin by testing TraceWeaver's accuracy as we increase the load (in terms of requests per second), calculated based on each app's bottleneck. Increasing the load increases the level of concurrency among requests that are simultaneously processed by the app. Figure 4a shows the accuracy of end-to-end tracing for varying load for TraceWeaver, and the baselines, WAP5, vPath (DeepFlow) and FCFS across the benchmark apps. As load increases, high levels of parallelism kick in, reordering requests more frequently, so FCFS cannot keep up. Multiple apps we test on break vPath's assumptions as they employ RPC frameworks like Thrift [1] and gRPC [2]. The thread-based approach manages to get 80% accuracy at low loads, but accuracy quickly degrades with increasing load as the internal multiplexing of requests across threads is not visible to it. WAP5, whose statistical algorithm was designed for a different usecase and does not consider overlapping parallel requests carefully, experiences similar degradation in accuracy. Due to a combination of techniques such as constraint-checking, batching, ranking, and joint optimization, TraceWeaver is able to achieve high accuracy.
Top K accuracy: We also analyze the accuracy of one of top K mappings being correct and we find that accuracy to be highly boosted. A Top K (for low K, we use $K = 5$) output can still be very useful to an operator as they need only enumerate within a ranked list of 5 candidate mappings at each service for a problematic span which would enable fine-grained debugging with enhanced accuracy.

6.2.2 Accuracy vs response times. Developers are often interested in analyzing end-to-end traces which suffer tail latency. Figure 4b

(a) Accuracy vs. load (across multiple apps).

(b) Accuracy vs latency %ile (across multiple apps).

(c) Accuracy in the presence of dynamic behavior (caching).

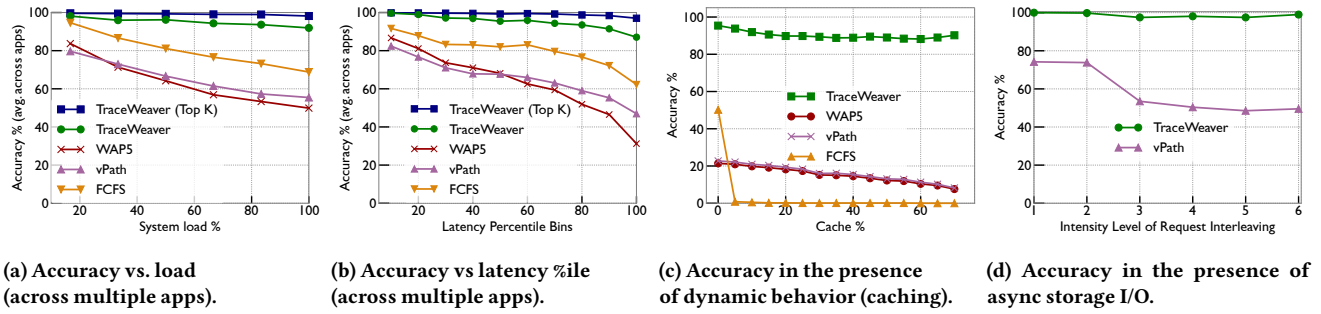**(d) Accuracy in the presence of async storage I/O.**

Figure 4: (a) Accuracy across increasing system load on all benchmarks. (b) Accuracy across requests binned by response times on all benchmarks. (c) Accuracy across increasing caching rates at one endpoint within the Hotel Reservation application. (d) Accuracy across increasing interleaving (1 is low and 6 is high) of async I/O reads within a single thread which cause inaccuracy in DeepFlow.

illustrates the accuracy for requests binned by their response time percentiles at single load level = 125 RPS (response times ranged from 40 ms to 225 ms for our applications). We can see that for the tail 10%ile requests, the accuracy of other baselines suffers, but TraceWeaver recovers the end-to-end traces with good accuracy.

6.2.3 Accuracy under increasing dynamism (caching). Behaviors such as caching and errors result in call graphs which differs from the static call graph we expect by traversing only a subset of the graph. We handle this deviation as mentioned in §4.2. To evaluate, we artificially insert caching into the search service of HotelReservation app, varying the cache hit probability from 5% to 80%. As we allow the optimization to apply call graph constraints in a fuzzy manner, TraceWeaver is able to get good accuracy in this dynamic scenario as illustrated in Figure 4c. Other approaches like FCFS and WAP5 fail to gracefully decline in accuracy due to caching causing high misalignment between the order of incoming and outgoing spans.

6.2.4 Accuracy in asynchronous settings. As mentioned previously, requests being handed off between threads within an app internally can break vPath/Deepflow's threading assumption (apps running gRPC/ Thrift already test this in Figure 4a). Yet another case is when async I/O causes multiple requests to be handled simultaneously by the same thread. Such async I/O can cause the last incoming request to not be responsible for the next outgoing request (as shown in Figure 2(b)). Figure 4d shows an experiment where we introduce such interleaving by creating async I/O requests for disk reads. We control interleaving by setting the standard deviation of the file size distribution. vPath/DeepFlow which depend on a synchronous model fail in such settings while TraceWeaver continues to perform well.

6.2.5 TraceWeaver ablation study. To understand the contribution of each component of TraceWeaver to the overall accuracy, we conduct an ablation study (by removing components of TraceWeaver incrementally) on the traces from the HotelReservation and Media Microservice apps. Figure 5 illustrates this experiment, showing a degradation in accuracy as expected when we incrementally remove the following optimizations: a. using invocation order to apply constraints (line 3), b. iterating to improve delay distributions (line 4), and c. batching to apply joint optimization across spans (line 5). Please note that not all optimizations benefit all apps equally. For instance, services with delay distributions that a simple Gaussian can represent
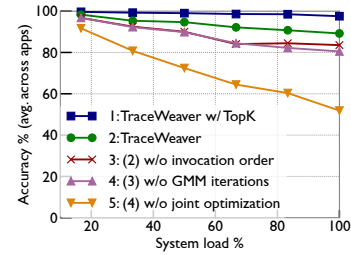

Figure 5: Component-level accuracy gains in TraceWeaver.

well will gain little from multiple iterations improving the GMM-based model, while others may see a substantial boost in accuracy (78% to 85% in one case). Similarly, a service where all backend invocations are done in parallel will not benefit from using constraints from the invocation order, as there aren't any serial dependencies.

6.3 Production traces

6.3.1 Accuracy vs. load multiple. Next, we evaluate on the Alibaba cluster dataset [12] similarly to the benchmark apps (accuracy vs. load). Our analysis considers a dataset spanning the traces corresponding to 15 most popular call graphs. The ground truth traces contain parent-child relationships between spans and start times and durations of individual spans. However, given the sanitized nature of a production dataset, call graph identifiers (i.e., API endpoint, port, header parameters) are notably missing. Therefore, for our analysis, we slice the trace data into buckets, one per call graph and evaluate TraceWeaver on each bucket. Another issue is that the production data is sampled at a rate of 0.5%, making the tracing challenge extremely easy (due to large spacings between adjacent spans in the sampled set). This is a common problem with any production dataset since operators must employ sampling to restrict the volume of logs. Therefore, to enable testing on production data, we employ the following strategy to compress the dataset to effectively increase the load while still evaluating on real (unchanged) service time distributions. To do this, we compress the time intervals between incoming spans in this trace dataset while, crucially, preserving the essential incoming → outgoing delay characteristics of the service. To clarify, suppose two incoming spans, S_1 and S_2 (belonging to trace T_1 and T_2)

(a) Accuracy on Alibaba's dataset (across 15 popular call graphs).

(b) Confidence scores for services in the Alibaba's dataset.

(c) Latency profile analysis of each service.

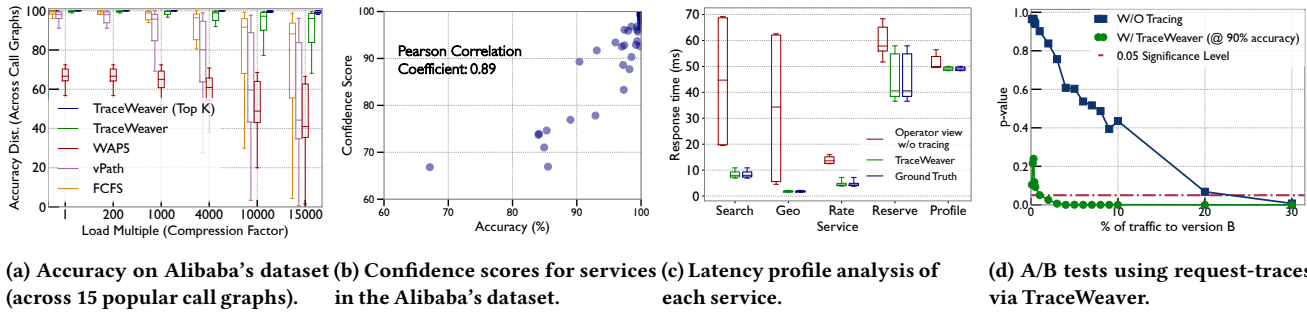(d) A/B tests using request-traces via TraceWeaver.

Figure 6: (a) Accuracy across increasing load multiple on Alibaba's dataset (boxplots presents accuracy %iles for 15 call graphs). (b) Confidence score compared to accuracy at specific services. (c) Latency profile at each service for requests above the 98%ile latency bracket. Boxplots represent the [5, 25, 50, 75, 95]%iles of corresponding distributions. (d) p-value for t-test comparing user satisfaction when $x\%$ of requests were sent to B vs when all requests were sent to A.

with arrival times t_1 and t_2, durations d_1 and d_2 are mapped to outgoing spans R_1 and R_2 respectively. To compress, we adjust the spacing ($t_2 - t_1$) to be much smaller using a compression factor cf while leaving d_1 and d_2 unchanged such that time gap for $S_1 \rightarrow R_1$ and $S_2 \rightarrow R_2$ remains the same. We also adjust the spacing between the outgoing spans by the same compression factor cf to ensure correctness. Effectively, a higher compression factor leads to higher concurrency and overlap in the traces creating many more plausible candidates which make non-intrusive request tracing harder which we can use to evaluate TraceWeaver. We refer to this compression factor as "load multiple" to signify the load increase. Please note, the trace data only indicates the service names for the caller and the callees but not the exact container ID. Therefore, we also normalize the load multiple by the number of replicas of a given service (i.e., for service S, effective_load = load_multiple / num_replicas) to recreate the load incident on a single container of that service (assuming load is balanced equally). The aim is to recreate realistic load but we increase the load multiple value further (i.e., to 15000) to evaluate the TraceWeaver's breaking point. Figure 6a shows our results on this dataset. As the load multiple increases, the accuracy drops for all algorithms, but TraceWeaver is still able to get high accuracy that is still practically usable.

6.3.2 Confidence scores (per-service). For each service S, we compute a confidence score, equal to 100% minus the % of incoming spans at S that either remained unmapped or weren't assigned their top choice mapping of outgoing spans. We find this confidence score to be well correlated with the accuracy with a very high Pearson correlation coefficient [46] of 0.89 (Figure 6b). Such a score can be used by operators to select an informed set of services to instrument if partial instrumentation of the application is possible (since instrumenting a small set of the X most difficult services among a total of Y services is much easier than instrumenting all of them).

6.4 Using approximate tracing for debugging

We showcase two use cases of how operators can use (imperfect) end-to-end traces derived from TraceWeaver for operational tasks.

6.4.1 Troubleshooting delays for slow requests. We emulate a performance anomaly scenario in the HotelReservation app (Fig. 7 in the appendix) by inflating the request-response span latency at the

Reserve and *Profile* services by 40ms for 10% of randomly selected requests. The operator's use-case is to *localize which service(s) are causing tail latency for the slowest 2% of requests (the chosen subset)?*

Answering this query requires complete traces for requests hitting the *frontend*, which have an e2e response latency > 98%ile and then computing the time spent at each service individually. Figure 6c presents our results. Note, for this app, spans can incur high delays at any set of services. So, in the absence of request traces, if an operator filters spans at each service by tail latency (top 2%), all services show up as contributing high latency (leading their debugging astray). This is because there are different subsets of requests which suffer high latency at each service. However, this view does not filter only requests that suffer the most cumulatively (and are hence in the top 2% overall). With full traces produced by TraceWeaver, an operator can now filter on traces (instead of spans) in the top 2% bracket. Note, these traces are in the top 2% because they suffer anomalous delays (which we injected) at two services simultaneously increasing their cumulative delay. Our results show us closely matching the ground truth in revealing *Reserve* and *Profile* as the culprits. Note that this aggregate query output, which is of interest to the operator, is not affected by a few inaccurate traces showcasing TraceWeaver's value.

6.4.2 A/B testing of a recommendation engine. In this scenario, operators want to run A/B tests for a new version (=B) of a recommendation algorithm and want to measure the effect of the new algorithm on user engagement. Usually, the service is modified so that a small percentage (=x) of requests are redirected to the newer version (=B) instead of the older version (=A). Thereafter, the two versions are compared using two-sample t-tests [31] to analyze if the difference between the user-satisfaction scores of the two groups (users served by A vs. B) is statistically significant or not. The test produces a p-value which is the probability of obtaining results at least as extreme as the observed results. If $p < 0.05$ (most commonly chosen threshold [45]), operators can conclude that the difference seen is statistically significant and that version B results in better user satisfaction than A. **Without request traces**, the operator cannot determine which user request was redirected to A or B. Note, it is not enough to analyze spans at A and B because user-satisfaction is not calculable at span-level and is only reflected end-to-end. Hence, the only way to compare user-satisfaction is to consider the aggregate user-satisfaction

score across all requests. If there is an increase in the overall user-satisfaction, one can conclude that the increase is due to the x% of the requests which were redirected to B. However, since B is untested, it is common to have a small x, say 1%, to minimize risk. If only a small number of requests are redirected to B, then the aggregate user satisfaction does not change much and conclusive evidence of improvement due to B cannot be obtained. The blue line in Figure 6d shows p-value remaining high for small x, indicating that there was inconclusive evidence for B being better. As x increases, the aggregate user-satisfaction score will pronouncedly reflect the changes due to B. **With request traces**, operators can separate the requests going to A and B, albeit with some error. The green line in Figure 6d shows that even when requests to A and B are separated with only 90% accuracy, the A/B test task of determining if B improves user satisfaction (as $p < 0.05$) can be completed. More importantly, it can be done with far less traffic redirected to B compared to the case w/o tracing (2% vs 20%).

6.5 Performance overhead

Presently, a single instance of TraceWeaver takes a few seconds (<5) to map 1000 spans on average (across our benchmarks) which is roughly ~200 RPS per container. However, we can further improve runtime by instantiating new instances of TraceWeaver which can handle disjoint sets of spans in parallel (e.g., instance A handles t=0 to 1s, B handles t=1 to 2s). The batching scheme mentioned in §4 helps identify such disjoint sets. However, systems handling production-scale loads often handle it by splitting traffic across replica containers, limiting per-container load. Hence, even in systems handling high load, a single online TraceWeaver instance usually only handles O(1000) RPS at most. Note, regardless of deployment mode (online v. offline), TraceWeaver runs off the critical path of the application.

6.6 Limitations

A single TraceWeaver instance is only required to reconstruct traces for spans within a single container because requests (of parent spans) arriving at container A do not result in backend requests (of child spans) being sent out of a different container B. This ultimately limits the concurrency that TraceWeaver needs to deal with to the resources of a single physical node, or more commonly, a single container within a node hosting multiple containers. However, it is possible that some apps may have extremely high parallelism within a single process, thus increasing the candidates TraceWeaver has to disambiguate among, worsening accuracy. However, most microservice systems prefer to avoid employing vertical scale-up [10] to handle high load [19] (further limiting parallelism) due to a variety of reasons like resource limitations, disruptive container restarts [29], etc., and instead prefer to horizontally scale-out [4].

TraceWeaver currently cannot handle cases where, for a given incoming span, the call graph is highly unpredictable (i.e., it is hard to know which call graph class a span belongs to). These include cases where the ensuing call graph after the reception of the span's request cannot be determined by standardized endpoints or requires custom application processing on the contents of the request to decide which backend endpoints are invoked via child spans. Note that many applications operate with predominantly static call graphs with dynamism exhibited mainly in the forms handled in §4.2. For other forms of dynamism, a clustering algorithm could better map spans to call graphs, using the endpoint as only one of many inputs (others could be span duration, candidate availability, header info., etc.).

TraceWeaver does not support head-based sampling [42], which makes the sampling decision when the request associated with a span arrives. To sample correctly, if a parent span is sampled, we want to keep all child spans associated with that parent span. However, to determine that association, TraceWeaver needs to (a) identify and compare the span's candidates (needing the response timestamp of the span) and (b) jointly optimize across all parent spans in the same optimization batch (requiring multiple parent spans sharing candidates with the span we're sampling to have finished processing). These properties are hard to satisfy in the case of head-based sampling, but TraceWeaver can support tail-based sampling as outlined in §5.3.

7 FUTURE WORK

Identifying thread affinity. Deepflow/ vPath makes restrictive assumptions about threading models, assuming no request hand-offs between threads which are common in modern apps (e.g., apps using gRPC/ Thrift). Tracking requests across such hand-offs (via monitoring syscalls handling critical section locks) could prove useful for non-intrusive request tracing. Even when a thread handles many requests concurrently (asynchronously), the ability to track requests to threads can help prune plausible candidates that TraceWeaver needs to consider, boosting accuracy. However, such fine-grained "taint-tracking" is expensive, and we leave its exploration to future work. **Handling variations in the call graph.** Handling variations beyond the ones mentioned in §4.2 can benefit apps employing quorums or where requests can invoke new, previously unseen endpoints. TraceWeaver can currently tackle the variations arising from spans traversing a subset of the call graph. Handling other forms of variations remains an interesting challenge and potential directions include looking at error codes in response headers to infer the number of retries and using eBPF to detect new syscalls.

8 CONCLUSION

Operating and debugging modern "cloud-native" applications with a microservice architecture is challenging due to their distributed nature. Our work introduces TraceWeaver, a system for non-intrusive request tracing that reconstructs request traces without requiring application-level code modifications. By leveraging span timings, call graph knowledge, and statistical timing heuristics, TraceWeaver significantly outperforms baseline approaches in accuracy. This high accuracy suggests that TraceWeaver can greatly reduce the manual effort typically required for tracing instrumentation, providing developers with an efficient and "free" debugging tool for managing and debugging complex microservice-based applications.

ACKNOWLEDGMENTS

REFERENCES

[1] Apache thrift. https://thrift.apache.org/.
[2] gRPC. https://grpc.io/.
[3] Gurobi Optimizer. https://github.com/Gurobi.
[4] Horizontal Pod Autoscaling. https://cloud.google.com/kubernetes-engine/docs/concepts/horizontalpodautoscaler.
[5] Hpack specification. http://http2.github.io/compression-spec/compression-spec.html.
[6] Http2. https://http2.github.io/.
[7] HTTP/2 parser. https://docs.python.org/3/library/http.client.html.
[8] Node.JS. https://nodejs.org/.
[9] Pixie. https://docs.px.dev/.
[10] Vertical Pod Autoscaling. https://cloud.google.com/kubernetes-engine/docs/concepts/verticalpodautoscaler.
[11] Algun. Node.js microservice. https://github.com/algun/jaeger-nodejs-example, 2019.
[12] Alibaba. Alibaba cluster trace program. https://github.com/alibaba/clusterdata, 2021.
[13] S. Ashok, P. B. Godfrey, and R. Mittal. Leveraging service meshes as a new network layer. In *Twentieth ACM Workshop on Hot Topics in Networks (HotNets)*, November 2021.
[14] O. Azizi. OpenTelemetry or eBPF? That is the Question. https://cloudnativeebpfdayna22.sched.com/event/1Auyh/opentelemetry-or-ebpf-that-is-the-question-omid-azizi-new-relic-pixie, 2022.
[15] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, page 13–24, New York, NY, USA, 2007. Association for Computing Machinery.
[16] P. Bailis, P. Alvaro, and S. Gulwani. Research for practice: Tracing and debugging distributed systems; programming by examples. *Commun. ACM*, 60(7):46–49, jun 2017.
[17] J. Berg, F. Ruffy, K. Nguyen, N. Yang, T. Kim, A. Sivaraman, R. Netravali, and S. Narayana. Snicket: Query-driven distributed tracing. In *Proceedings of the Twentieth ACM Workshop on Hot Topics in Networks*, HotNets '21, page 206–212, New York, NY, USA, 2021. Association for Computing Machinery.
[18] I. Beschastnikh, P. Wang, Y. Brun, and M. D. Ernst. Debugging distributed systems: Challenges and options for validation and debugging. *Communications of the ACM*, 59(8):32–37, Aug. 2016.
[19] G. Blinowski, A. Ojdowska, and A. Przybyłek. Monolithic vs. microservice architecture: A performance and scalability evaluation. *IEEE Access*, 10:20357–20374, 2022.
[20] X. Chen, M. Zhang, Z. M. Mao, and V. Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In *OSDI*. USENIX, January 2008.
[21] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. 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)*, pages 217–231, Broomfield, CO, Oct. 2014. USENIX Association.
[22] M. Chow, K. Veeraraghavan, M. Cafarella, and J. Flinn. DQBarge: Improving Data-Quality tradeoffs in Large-Scale internet services. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 771–786, Savannah, GA, Nov. 2016. USENIX Association.
[23] Datadog. Datadog. https://www.datadoghq.com/.
[24] Docker. Docker. https://www.docker.com/, 2013.
[25] DynaTrace. DynaTrace. https://www.dynatrace.com/.
[26] eBPF. ebpf hooks, 2014.
[27] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker. X-Trace: A pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*, Cambridge, MA, Apr. 2007. USENIX Association.
[28] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. 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, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.
[29] Github. Open Issue: Support in-place Pod vertical scaling in VPA. https://github.com/kubernetes/autoscaler/issues/4016.
[30] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
[31] W. S. Gosset. Independent two-sample t-test. https://en.wikipedia.org/wiki/Student%27s_t-test#Independent_two-sample_t-test, 2023.
[32] T. Hastie, R. Tibshirani, J. Friedman, and J. Franklin. The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer*, 27(2):460–468, 2005.
[33] Instana. Instana. https://www.ibm.com/cloud/instana.
[34] Istio. Istio. https://istio.io, 2021.
[35] Jaeger. Jaeger. https://www.jaegertracing.io/.
[36] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An end-to-end performance tracing and analysis system. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 34–50, New York, NY, USA, 2017. Association for Computing Machinery.
[37] E. Koskinen and J. Jannotti. Borderpatrol: isolating events for black-box tracing. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, Eurosys '08, page 191–203, New York, NY, USA, 2008. Association for Computing Machinery.
[38] Kubernetes. Kubernetes. https://kubernetes.io/, 2014.
[39] LightStep. LightStep. http://lightstep.com/, 2023.
[40] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar. Modeling the parallel execution of Black-Box services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 11)*, Portland, OR, June 2011. USENIX Association.
[41] D. M. Nguyen, H. A. Le Thi, and T. Pham Dinh. Solving the multidimensional assignment problem by a cross-entropy method. *Journal of Combinatorial Optimization*, 27(4):808–823, 2014.
[42] OpenTelemetry. Head Sampling. https://opentelemetry.io/docs/concepts/sampling/#head-sampling.
[43] OpenTelemetry. OpenTelemetry. https://opentelemetry.io/.
[44] Oracle. Oracle. https://docs.oracle.com/en-us/iaas/Content/Functions/Tasks/functionstracing.htm, 2023.
[45] K. Pearson. P-value. https://en.wikipedia.org/wiki/P-value#Usage, 2023.
[46] K. Pearson. Pearson correlation coefficient. https://en.wikipedia.org/wiki/Pearson_correlation_coefficient, 2023.
[47] M. Raney. What I Wish I Had Known Before Scaling Uber to 1000 Services. In *GOTO Chicago*, 2016.
[48] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera, and A. Vahdat. WAP5: Black-Box Performance Debugging for Wide-Area Systems. In *Proceedings of the 15th International Conference on World Wide Web*, WWW '06, page 347–356, New York, NY, USA, 2006. Association for Computing Machinery.
[49] scikit learn. Expectation Maximization. https://scikit-learn.org/stable/modules/mixture.html#expectation-maximization.
[50] scikit learn. Gaussian Mixture Models. https://scikit-learn.org/stable/modules/mixture.html.
[51] J. Shen, H. Zhang, Y. Xiang, X. Shi, X. Li, Y. Shen, Z. Zhang, Y. Wu, X. Yin, J. Wang, M. Xu, Y. Li, J. Yin, J. Song, Z. Li, and R. Nie. Network-centric distributed tracing with deepflow: Troubleshooting your microservices in zero code. In *Proceedings of the ACM SIGCOMM 2023 Conference*, ACM SIGCOMM '23, page 420–437, New York, NY, USA, 2023. Association for Computing Machinery.
[52] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.
[53] F. D. Silva and C. Rich. Broaden application performance monitoring to support digital business transformation, 2018.
[54] B. C. Tak, C. Tang, C. Zhang, S. Govindan, B. Urgaonkar, and R. N. Chang. Vpath: Precise discovery of request processing paths from black-box observations of thread and network activities. In *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, USENIX'09, page 19, USA, 2009. USENIX Association.
[55] tcpdump. tcpdump: a powerful command-line packet analyzer.
[56] T. P. Team. What is an API endpoint? https://blog.postman.com/what-is-an-api-endpoint/, 2023.
[57] G. Tene. Wrk2: a HTTP benchmarking tool based mostly on wrk. https://github.com/giltene/wrk2, 2019.
[58] Wikipedia. Bayesian Information Criterion (BIC). https://en.wikipedia.org/wiki/Bayesian_information_criterion.
[59] Wikipedia. Central Limit Theorem. https://en.wikipedia.org/wiki/Central_limit_theorem.
[60] Wikipedia. Water filling algorithm. https://en.wikipedia.org/wiki/Water_filling_algorithm.
[61] Wireshark. Wireshark: The world's most popular network protocol analyzer.
[62] G. Zhou and M. Maas. Learning on distributed traces for data center storage systems. In *4th Conference on Machine Learning and Systems (MLSys 2021)*, 2021.
[63] Zipkin. Zipkin. https://zipkin.io/.

A APPENDICES

Appendices are supporting material that has not been peer-reviewed.

A.1 Sample application topologies

Figure 7 illustrates the Hotel Reservation app consisting of 6 services, excluding a varying count of MongoDB and Memcached instances.
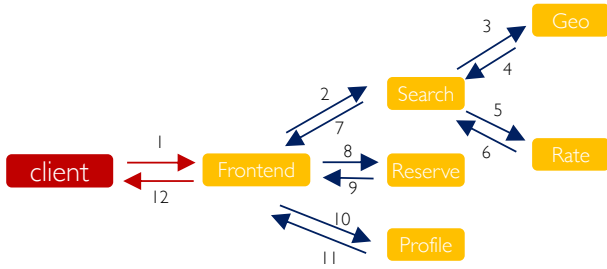
Figure 7: The Hotel Reservation application from the DeathStarBench suite [28].

Figure 8 illustrates the Media Microservices consisting of 14 services, excluding a varying count of MongoDB and Memcached instances. The invocation order of the spans is not shown for the sake of brevity.
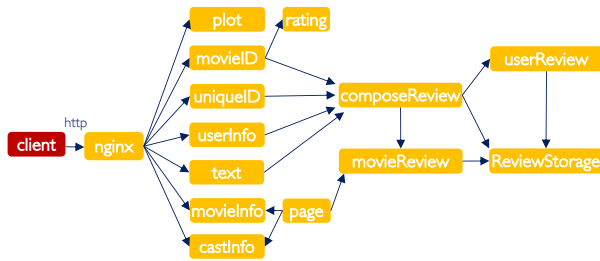

Figure 8: The Media Microservices application from the DeathStarBench suite [28].

A.2 Proof for the perfect cuts-based batching algorithm.

We provide a correctness proof for the algorithm that we use to identify batch boundaries based off "perfect cuts", where a perfect cut corresponds to creating a batch boundary where a pair of spans on either side of the boundary do not share a common candidate mapping. First, we define some terms to set the stage for proving the result.

(1) Let a span A be represented by a pair of start and end times, where start $\leq$ end, which we call its time window $T(A)$.
(2) Let there be a list of such spans sorted by start time with ties broken by end time.
(3) Let a candidate for a span i consist of a set of child spans that satisfy constraints (as discussed in §4.1). By definition, the child spans must have start time $\geq$ start time of span i and end time $\leq$ end time of span i.
(4) Let $Cut(i-1,i)$ represent a boundary in a list of spans such that a $Cut(i-1,i)$ creates a past set of spans with indices in the range $[0,i)$ and a future set of spans with indices in range $[i,K)$, where K is the size of the list. A $Perfect\ Cut(i-1,i)$ hence implies that the past and future set of spans do not share any common candidate mappings.
(5) Let $Pair(X,Y)$ represent a pair of spans with index X and Y respectively and $T(X,Y)$ represent the intersection of the $T(X)$ and $T(Y)$.

(6) Let index j_i represent a span with the latest end time among all spans with index $< i$.

For $Cut(i-1,i)$, consider the following two statements–

(1) $\mathcal{S}_1$: $Pair(j_i,i)$ do not share a candidate and span j_i ends before span i.
(2) $\mathcal{S}_2$: There exists a $Pair(x,y)$ sharing a candidate, where x and y are indices of spans before and after the $cut(i-1,i)$ respectively so that $x \leq i-1$ and $y \geq i$.

THEOREM A.1. $\mathcal{S}_1 \rightarrow not\ \mathcal{S}_2$. That is, if for a certain i, span i and span j_i do not share a candidate and span j_i ends before span i, then any span after i, including i, does not share a candidate with any span before i.

PROOF. Proof by Contradiction:

If $Pair(x,y)$ share a candidate, the candidate must exist in a $T(x,y)$ which is not a subset of $T(j_i,i)$ (as otherwise $Pair(j_i,i)$ would then trivially also share the same candidate).

However, this cannot be true for the following reasons:

(1) The earliest $T(x,y)$ can begin is at the start of span i, given that span i is the first span after the $Cut(i-1,i)$. Hence, $T(x,y)$ can, at the earliest, only begin at the start time of span i.
(2) The latest $T(x,y)$ can end is at the latest end time of any span before the $Cut(i-1,i)$, which is at the end of span j_i by definition and is included in $T(j_i,i)$ since span j_i ends before span i.
(3) Therefore, any $T(x,y)$ has to be a subset of $T(j_i,i)$ which creates a contradiction as the candidate which is possible for $Pair(x,y)$ must also be possible for $Pair(j_i,i)$

Hence, if we can guarantee $\mathcal{S}_1$ to hold, $\mathcal{S}_2$ cannot be true. □

A.3 Illustration of the various optimization steps.
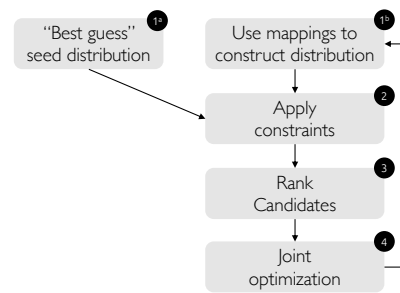
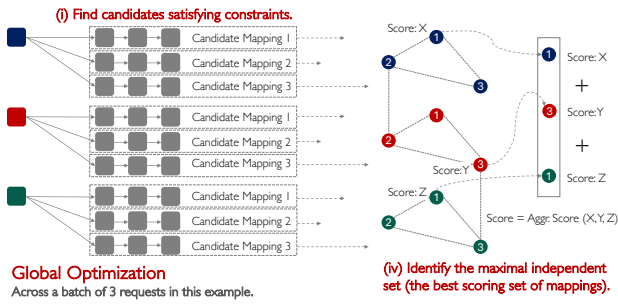**Figure 9: Constructing delay distributions (step 3) as described in S4.1**

Figure 10: Joint optimization (step 5) as described in S4.1