# So, you want to trace your distributed system? Key design insights from years of practical experience

Raja R. Sambasivan*, Rodrigo Fonseca†, Ilari Shafer‡, Gregory R. Ganger*

*Carnegie Mellon University, †Brown University, ‡Microsoft*

CMU-PDL-14-102

April 2014

**Parallel Data Laboratory**

Carnegie Mellon University

Pittsburgh, PA 15213-3890

## Abstract

*End-to-end tracing captures the workflow of causally-related activity (e.g., work done to process a request) within and among the components of a distributed system. As distributed systems grow in scale and complexity, such tracing is becoming a critical tool for management tasks like diagnosis and resource accounting. Drawing upon our experiences building and using end-to-end tracing infrastructures, this paper distills the key design axes that dictate trace utility for important use cases. Developing tracing infrastructures without explicitly understanding these axes and choices for them will likely result in infrastructures that are not useful for their intended purposes. In addition to identifying the design axes, this paper identifies good design choices for various tracing use cases, contrasts them to choices made by previous tracing implementations, and shows where prior implementations fall short. It also identifies remaining challenges on the path to making tracing an integral part of distributed system design.*

**Acknowledgements:** We thank the members and companies of the PDL Consortium (including Actifio, APC, EMC, Emulex, Facebook, Fusion-io, Google, Hewlett-Packard Labs, Hitachi, Huawei Technologies, Intel, Microsoft Research, NEC Laboratories, NetApp, Oracle, Panasas, Riverbed, Samsung, Seagate, STEC, Symantec, VMWare, and Western Digital) for their interest, insights, feedback, and support. This research was sponsored in part by two Google research awards, NSF grant #CNS-1117567, and by Intel via the Intel Science and Technology Center for Cloud Computing (ISTC-CC). While at Carnegie Mellon, Ilari Shafer was supported in part by a National Science Foundation Graduate Research Fellowship.

**Keywords:** Cloud computing, Distributed systems, Design, End-to-end tracing

## 1 Introduction

Modern distributed services are large, complex, and increasingly built upon other similarly complex distributed services. For example, many Google services use various internal services (e.g., for ads and spell-checking) and have been deployed atop infrastructure services like Bigtable [10], which itself is spread across 100s of nodes and built atop other services like the Google File System (GFS) [21] and the Chubby lock service [7]. Even "simple" web applications generally involve multiple tiers, some of which are scalable and distributed. Management and development tasks (e.g., performance debugging, capacity planning, and problem diagnosis) are always difficult, and are made even more so in such environments because traditional *machine-centric* monitoring and tracing mechanisms [32, 46] are not effective. In particular, they cannot provide a coherent view of the work done by a distributed service's nodes and dependencies.

To address this issue, recent research has developed new tools and approaches based on *workflow-centric* tracing techniques, which we collectively refer to as "end-to-end tracing" [2, 5, 9, 11, 19, 20, 23, 25, 37, 38, 40, 43–45, 47, 48, 53]. End-to-end tracing captures the detailed workflow of causally-related activity within and among the components of a distributed system. For example, for a request-based distributed service, each individual trace would show the work done within and among the service's components to process a request (see Figure 1 for an example).

As was the case with single-process debugging tools applied to single-process applications, the detailed information provided by end-to-end tracing about how a distributed service processes requests is invaluable for development and administration. To date, end-to-end tracing has been shown to be sufficiently efficient when using sampling (i.e., less than 1% runtime overhead [40, 43]) to be enabled continuously and has proven useful for many important use cases, including anomaly detection [5, 11], diagnosis of steady-state correctness and performance problems [11, 19, 20, 37, 40, 43], profiling [9, 43], and resource-usage attribution [18, 47]. As such, there are a growing number of industry implementations, including Google's Dapper [43], Cloudera's HTrace [13], Twitter's Zipkin [50], and others [14, 49]. Looking forward, end-to-end tracing has the potential to become the fundamental substrate for providing a global view of intra- and inter-datacenter activity in cloud environments.

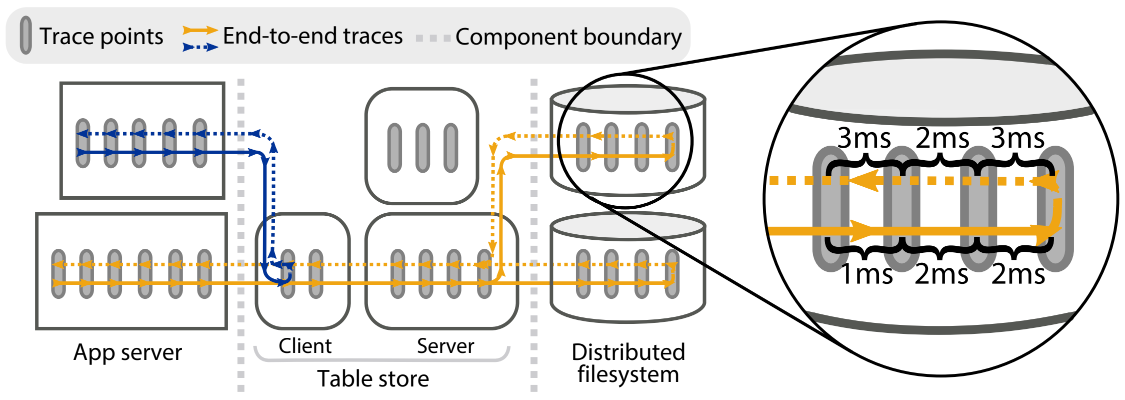

**Figure 1: Two end-to-end traces.** Such traces show the workflow of causally-related activity within and among the components of a distributed system and (optionally) other distributed services it depends on. A trace can potentially include information about the *structure* of the workflow (i.e, the causal order of work executed, amount of concurrency, and locations of forks and joins), its performance, and resource usage. However, only a subset of this information needs to be collected for different use cases. In this case, the traces show the structure and inter-trace-point latencies of two requests submitted to a hypothetical application server that is backed by a shared table store and a shared distributed filesystem. The data needed by the top request is in the table store's cache, whereas the data needed by the bottom request must be retrieved concurrently from two storage nodes. Trace points are simply markers indicating the locations in a component's software reached by a workflow.

Unfortunately, despite the strong emerging interest in end-to-end tracing, there exists too little information to guide designers of new infrastructures. Most problematically, existing literature treats end-to-end tracing as a generic "one-size-fits-all" solution for many of its possible use cases (e.g., diagnosis of steady-state problems or resource attribution). But, our experiences designing two of the most well-known tracing infrastructures (Stardust [40, 47], and X-Trace [19, 20]) and building tools on top of them and Dapper [43] have proven to us that such generality is unwarranted. For example, our initial experiences [41] building and using Spectroscope [40], a tool that uses end-to-end traces to automatically localize the source of performance changes, were immensely frustrating. Our initial design for this diagnosis tool reused a tracing infrastructure that had previously been used for resource-attribution tasks, but this "one-size-fits-all" belief resulted in a very limited tool that incurred high overheads. Only after our initial failure did we realize that tracing infrastructures designed with resource-attribution tasks in mind will not show critical paths in the face of on-demand evictions, may not show synchronization points, and are incompatible with certain forms of sampling to reduce overhead, thus limiting their utility for diagnosis. Our experiences modifying the original tracing infrastructure (Stardust [47] to one that was useful for diagnosis tasks (the revised Stardust [40]) helped inform many of the insights presented in this paper.

The basic concept of tracing is straightforward—instrumentation at chosen points in the distributed service's code produces data when executed, and the data from various points reached for a given request can be combined to produce an overall trace. But, from our experiences, we have learned that four key design axes dictate the utility of the resulting traces for different use cases: which causal relationships should be preserved, how causal relationships are tracked, how to reduce overhead by sampling, and how end-to-end traces should be visualized. Designing tracing infrastructures without knowledge of these axes and the tradeoffs between choices for them will lead to implementations that are not useful for their intended purposes. And, of course, efficiency concerns make it impractical to support all possible use cases. Indeed, since these axes have not been previously identified and are not well understood, many tracing implementations have failed to live up to their potential.

This paper helps guide designers of tracing infrastructures. Drawing on our experiences and the past ten years of research on end-to-end tracing, we distill the key design axes of end-to-end tracing and explain tradeoffs associated with the options for each. Beyond describing the degrees of freedom, we suggest specific design points for each of several key tracing use cases and identify which previous tracing implementations do and do not match up. We believe that this is the first paper to identify these design axes and show how different choices for them can drastically affect a tracing infrastructure's usefulness.

The remainder of this paper is organized as follows. Section 2 discusses use cases for and the basic anatomy of end-to-end tracing. Sections 3–6 describe the design axes and their tradeoffs. Section 7 applies these insights to suggest specific design choices for each of several use cases as well as to analyze the suitability of existing end-to-end tracing infrastructures for those use cases. Section 8 discusses some challenges and opportunities that remain in realizing the full potential of end-to-end tracing.

## 2 Background

This section describes relevant background about end-to-end tracing. Section 2.1 describes its key use cases. Section 2.2 lists the three commonly used approaches to end-to-end tracing and Section 2.3 describes the architecture of the approach advocated in this paper.

### 2.1 Use cases

Table 1 summarizes the key use cases of end-to-end tracing and lists tracing implementations suited for them. Note that some of the listed infrastructures were initially thought to be useful for a wider variety of use cases than those attributed to them in the table. For example, we initially thought that the original Stardust [47]

| Use case | Implementations | |
|---|---|---|
| Anomaly detection | Magpie [5] | Pinpoint [11] |
| Diagnosing steady-state problems | Dapper [43]<br>Pip [37]<br>Pinpoint [11] | Stardust‡ [40]<br>X-Trace [20]<br>X-Trace‡ [19] |
| Distributed profiling | ETE [23]<br>Dapper [43] | Whodunit [9] |
| Resource attribution | Stardust [47] | Quanto [18] |
| Workload modeling | Magpie [5]<br>All others | Stardust [47] |

**Table 1: Main uses of end-to-end tracing.** This table lists the key use cases for end-to-end tracing and tracing implementations suited for them. Some implementations appear for multiple use cases. The revised versions of Stardust and X-Trace are denoted by *Stardust*‡ and *X-Trace*‡. Almost all tracing implementations can be used to model or summarize workloads, though the types of models that can be created will differ based on the design choices made for them.

would be useful for both resource attribution and diagnosis. Similarly, Google's Dapper has proven less useful than initially thought because it cannot be used to detect certain types of anomalies [35]. It is these mismatches between "thought to be useful for" and "actually useful for" that this paper hopes to minimize.

**Anomaly detection**: This diagnosis-related use case involves identifying and debugging problems that manifest as rare workflows that are extremely different (e.g., fall in the 99.9th percentile) from other workflows. Such problems can be related to correctness (e.g., component timeouts or failures) or performance (e.g., a slow function or excessive waiting for a slow thread). They may manifest as workflows that are extremely different than others with regard to their *structures*—(i.e, the causal order of work executed, amount of concurrency, and locations of forks and joins)—latencies, or resource usages. Magpie [5] identifies both correctness- and performance-related anomalies, whereas Pinpoint's [11] anomaly detection component focuses solely on correctness problems.

**Diagnosing steady-state problems**: This is another diagnosis-related use case, which involves identifying and debugging problems that manifest in many workflows (and so are not anomalies). Such problems affect the 50th or 75th percentile of some important metric, not the 99.9th. They may manifest in workflows' structures, latencies, or resource usages, and are generally performance related—for example, a configuration change that modifies the storage nodes accessed by a set of requests and increases their response times. Pip [37], the revised version of Stardust (Stardust‡ [40]), both versions of X-Trace [19, 20], Dapper [43], and parts of Pinpoint [11] are all most useful for diagnosing steady-state problems.

**Distributed profiling**: The goal of distributed profiling is to identify slow components or functions. Since the time a function takes to execute may differ based on how it is invoked, profilers often maintain separate bins for every unique calling stack, so full workflow structures need not be preserved. Whodunit [9] is explicitly designed for this purpose and can be used to profile entire workloads. Dapper [43] and ETE [23] show visualizations that help profile individual workflows.

**Resource attribution**: This use case is designed to answer question "Who should be charged for this piece of work executed deep in the stack of my distributed system's components?" It involves tying work done at an arbitrary component of the distributed system to the client or request that originally submitted it. Quanto [18] and the original version of Stardust [47] are most useful for resource attribution. The former ties per-device energy usage to high-level activities (e.g., sensing or routing) in distributed-embedded systems. The latter ties per-component resource usages (e.g., CPU time or disk time) to clients in distributed storage systems or databases. We note that resource-attribution-based tracing can be especially useful for accounting and billing purposes, especially in distributed services shared by many clients, such as Amazon's EC2 [51].

**Workload modeling**: This catch-all use case involves using end-to-end traces to create workload summaries, which can then be used for later analyses or inferences. One example in this vein is Magpie [5], which clusters its traces to identify those unique sets of workflows that are representative of the entire workload. Stardust [47] can be used to create queuing models that answer "what-if" questions (e.g., "What would happen to the performance of workload A if I replaced the CPU on a certain distributed system component with a faster one?"). Almost any tracing implementation can be viewed as useful for workload modeling. But, the types of models that it can be create will be dictated by the design choices made for it.

## 2.2 Approaches to end-to-end tracing

Most end-to-end tracing infrastructures use one of three approaches to identify causally-related activity: metadata propagation, schemas, or black-box inference. This paper focuses on design decisions for tracing infrastructures that use the first, as they are more scalable and produce more accurate traces than those that use the other two. However, many of our analyses are also applicable to the other approaches.

**Metadata propagation**: Like security, end-to-end tracing works best when it is designed as part of the distributed system. As such, many implementations are designed for use with white-box systems, for which the components can be modified to propagate metadata (e.g., an ID) delineating causally-related activity [9,11,18–20,37,40,43,47]. All metadata-propagation-based implementations identify causality between individual functions or trace points, which resemble log messages and record the fact that a particular point in the system was reached at a particular time. To keep runtime overhead (e.g., slowdown in response time and throughput) to a minimum so that tracing can be "always on," most tracing infrastructures in this category use sampling to collect only a small number of trace points or workflows.

**Schema-based**: A few implementations, such as ETE [23] and Magpie [5], do not propagate metadata, but rather require developers to write *temporal join-schemas* that establish causal relationships among variables exposed in custom-written log messages. Schema-based approaches are not compatible with sampling, since they delay determining what is causally related until after all logs are collected. Therefore, they are less scalable than metadata-propagation approaches.

**Black-box inference**: Several end-to-end tracing implementations [2,6,25,28,38,44,45,53] do not modify the traced systems. Rather, they infer causality by either correlating variables or timings from pre-existing logs [6,25,45,53] or making simplifying assumptions [44]. Though the idea of obtaining end-to-end traces without software modification is appealing, these approaches cannot properly attribute causality in the face of asynchronous behaviour (e.g., caching, event-driven systems), concurrency, aggregation, or code-specific design patterns (e.g., 2-of-3 storage encodings), all of which are common in distributed systems.

## 2.3 Anatomy of end-to-end tracing

Figure 2 shows the anatomy of most metadata-propagation-based end-to-end tracing infrastructures. The software components work to identify work done in the distributed system, preserve the chosen causal relationships, limit overhead, optionally persist trace data to storage, create traces, and present traces to developers. They include individual trace points, the causal-tracking mechanism, the sampling mechanism, the storage component, the trace construction code, and the presentation layer.

Developing such an infrastructure requires answering two conceptual design questions that will dictate the infrastructure's fundamental capabilities. The first is: "what causal relationships should be preserved?" Preserving all of them would result in too much overhead, yet preserving the wrong ones will will yield useless traces. Section 3 describes which causal relationships should be preserved for the use cases identified in the previous section.

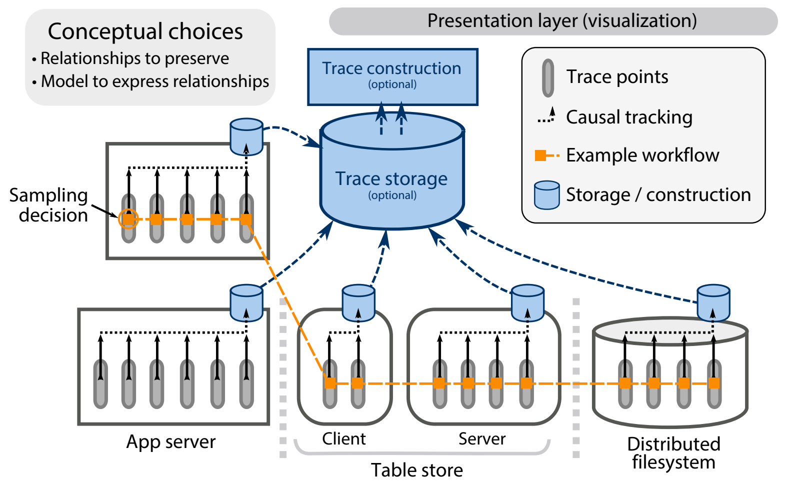

**Figure 2: Anatomy of end-to-end tracing.** The elements of a typical metadata-propagation-based tracing infrastructure are shown.

The second conceptual design question is: "What model should be used to express relationships?" Specialized models can only represent a few types of causal relationships, but can be stored and retrieved more efficiently; expressive models make the opposite tradeoff. The most popular specialized model is a directed acyclic tree, which is sufficient for expressing sequential, concurrent, or recursive call/reply patterns (e.g., as observed in RPCs). Forks and concurrency are represented by branches. It is used by the original X-Trace [20], Dapper [43], and Whodunit [9]. Dapper additionally optimizes for call/reply patterns by using a storage schema that is optimized for retrieving them. Pinpoint [11] uses paths, which are sufficient for representing synchronous behaviour and event-based processing.

Though useful, trees cannot be used to represent nodes with multiple parents. This corresponds to cases where a single distributed-system event is causally dependent on several previous events. Examples include joins or processing that depends on several inputs. Since preserving joins is important for diagnosis tasks (see Section 3.1.4), the revised version of Stardust [40] and the revised version of X-Trace [19] use general directed acyclic graphs (DAGs) instead of trees. The original version of Stardust [47] also uses DAGs to establish relationships between original submitters of work and aggregated activity. Pip [37] has the same expressive power as DAGs through its representation of arbitrary messages among its tasks.

We now briefly describe the basic components of the infrastructure. The causal-tracking mechanism propagates metadata with workflows to preserve the desired causal relationships. It is critical to end-to-end tracing, and key design decisions for it are described in Section 4.

Individual trace points indicate locations in the codebase accessed by individual workflows. Executing a trace point creates a trace-point record, which contains information about the executed trace point, associated metadata, and any additional data developers may want to capture (e.g., the current call stack). Trace points are often embedded in commonly used libraries (e.g., RPC libraries) and added by developers in important areas of the distributed system's software [19, 40, 43]. Though many design decisions for where to add trace points are similar to those for logging, trace points can also help distinguish workflow structures (see Section 4.2) for examples of how these *structural trace points* can be used to identify forks and joins). Alternatively, binary re-writing or interposition techniques like aspect-oriented programming [27] can be used to automatically add trace points at function boundaries [9, 17] or other locations, such as forks and joins.

Most tracing infrastructures use sampling techniques to limit runtime and/or storage overhead. *Coherent sampling* is often used, in which either all or none of a workflow's trace points are sampled. For example, by using coherent sampling to persist (i.e., store to stable storage) less than 1% of all workflows' trace-point records, the revised version of Stardust [40] and Dapper [43] impose less than a 1% runtime overhead. The choice of what causal relationships are to be preserved dictates the sampling technique that should be used. Section 5 describes different sampling techniques and their tradeoffs.

The storage component persists trace-point records. The trace construction code joins trace-point records that have related metadata to construct traces of causally-related activity. These components are optional, since trace-point records need not be persisted if the desired analyses can be performed online. For example, for some analyses, it is sufficient to propagate important data with causally-related activity and read it at executed trace points.

Several good engineering choices, as implemented by Dapper [43], can minimize the performance impact of persisting trace-points records. First, on individual components, records of sampled trace points should be logged asynchronously (i.e., off the critical path of the distributed system). For example, this can be done by copying them to an in-memory circular buffer (or discarding them if the buffer is full) and using a separate thread to write trace points from this buffer to local disk or to a table store. A MapReduce job can then be used to construct traces. Both Stardust [47] and Dapper [43] suggest storing traces for two weeks for post-hoc analyses before discarding them.

The final aspect of an end-to-end tracing infrastructure is the presentation layer. It is is responsible for showing constructed traces to users and is important for diagnosis-related tasks. Various ways to visualize traces and tradeoffs between them are discussed in Section 6.

## 3 Which causal relationships should be preserved?

Since preserving causally-related activity is the ultimate goal of end-to-end tracing, the ideal tracing infrastructure would preserve all true or *necessary* causal relationships, and only those. For example, it would preserve the workflow of servicing individual requests and background activities, read-after-write accesses to memory, caches, files, and registers, data provenance, inter-request causal relationships due to resource contention (e.g., for caches) or built-up state, and so on.

However, it is difficult to know what activities are truly causally related. As such, tracing infrastructures resort to preserving Lamport's happens-before relation ($\rightarrow$), which states that if $a$ and $b$ are events and $a \rightarrow b$, then *$a$ may have influenced $b$*, and thus, $b$ might be causally dependent on $a$ [29]. But, the happens-before relation is only an approximation of true causality: it can be both too indiscriminate and incomplete at the same time. It can be incomplete because it is impossible to know all channels of influence, which can be outside of the system [12]. It can be too indiscriminate because it captures irrelevant causality, as *may have influenced* does not mean *has influenced*.

Tracing infrastructures limit indiscriminateness by using knowledge of the system being traced and the environment to capture only the *slices* of the general happens-before graph that are most likely to contain necessary causal relationships. First, most tracing infrastructures make assumptions about boundaries of influence among events. For example, by assuming a memory-protection model, the tracing infrastructure may exclude happens-before edges between activities in different processes, or even between different activities in a single-threaded event-based system (see Section 4 for mechanisms by which spurious edges are removed). Second, they may ask developers to explicitly add trace points in areas of the distributed system's software they deem important and only track relationships between these trace points [11, 19, 20, 37, 40, 43, 47].

Different slices are useful for different use cases, but preserving all of them would incur too much overhead (even the most efficient software taint-tracking mechanisms yield a 2x to 8x slowdown [26]). As such, tracing infrastructures work to preserve only the slices that are most useful for how their outputs will be used. The rest of this section describes slices that have proven useful for various use cases.

## 3.1 Intra-request slices

When developing a tracing infrastructure, developers must choose a slice of the happens-before graph that defines the workflow of a request as it is being serviced by a distributed system. Work created by the submitting request that is performed before the request responds to the client must be considered part of its workflow. However, latent work (e.g., data left in a write-back cache that must be written to disk eventually) can either be considered part of the submitting request's workflow or part of the request that forces that work to be executed (e.g., via an on-demand cache eviction). This observation forms the basis for two *intra-request* slices—submitter-preserving and trigger-preserving—that preserve different information and are useful for different use cases. We first identified these slices and the differences between them while trying to understand why the original Stardust [47] wasn't useful for diagnosis tasks.

Section 3.1.1 and Section 3.1.2 describe the tradeoffs involved in preserving the submitter-preserving and trigger-preserving slices in more detail. Section 3.1.3 lists the advantages of preserving both intra-request slices. Section 3.1.4 discusses the benefits of delineating concurrent behaviour from sequential behaviour and preserving forks and joins in individual traces. Table 2 shows intra-request slices most useful for the key uses of end-to-end tracing.

### 3.1.1 The submitter-preserving slice

Preserving this slice means that individual end-to-end traces will show causality between the original submitter of a request and work done to process it through every component of the system. It is most useful for resource attribution, since this usage mode requires that end-to-end traces tie the work done at a component several levels deep in the system to the client, workload, or request responsible for originally submitting it. Quanto [18], Whodunit [9], and the original version of Stardust [47] preserve this slice of causality. The two leftmost diagrams in Figure 3 show submitter-preserving traces for two WRITE requests in a distributed storage system. Request one writes data to the system's cache and immediately replies. Sometime later, request two enters the system and must evict request one's data to place its data in the cache. To preserve submitter causality, the tracing infrastructure attributes the work done for the eviction to request one, not request two. Request two's trace only shows the latency of the eviction. Note that the tracing infrastructure would attribute work the same way if request two were a background cleaner thread instead of a client request that causes an on-demand eviction.

### 3.1.2 The trigger-preserving slice

The submitter-preserving trace for request one shown in Figure 3 is unintuitive and hard to understand when visualized because it attributes work done to the request after the client reply has been sent. Also, latent work

| **Intended use** | **Slice** | **Preserve forks/joins/concurrency** |
|---|---|---|
| Anomaly detection | Trigger | Y |
| Diagnosing steady-state problems | ” | ” |
| Distributed profiling | Either | N |
| Resource attribution | Submitter | ” |
| Workload modeling | Depends | Depends |

**Table 2: Suggested intra-flow slices to preserve for various intended uses.** Since the same necessary work is simply attributed differently for both trigger- and submitter-preserving slices, either can be used for profiling. The causality choice for workload modeling depends on what aspects of the workload are being modeled.

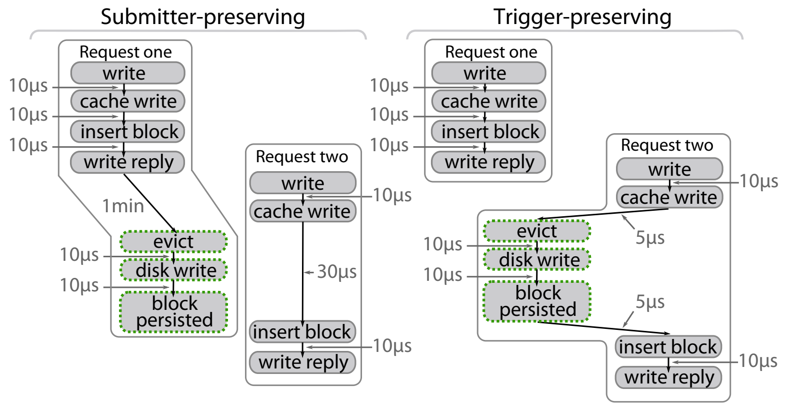

**Figure 3: Traces for two storage system WRITE requests when preserving different slices of causality.** Request one places its data in a write-back cache and returns immediately to the client. Sometime later, request two enters the system and must perform an on-demand eviction of request one's data to place its data in the cache. This latent work (highlighted in dotted green) may be attributed to request one (if submitter causality is preserved) or request two (if trigger causality is preserved). The one minute latency for the leftmost trace is an artifact of the fact that the traces show latencies between trace-point executions. It would not appear if they showed latencies of function call executions instead, as is the case for Whodunit [9].

attributed to this request (i.e., trace points executed after the reply is sent) is performed in the critical path of request two. In contrast, trigger causality guarantees that a trace of a request will show all work that must be performed before a client response can be sent, including another client's latent work if it is executed in the request's critical path. The right two traces in Figure 3 show the same two requests as in the submitter-preserving example, with trigger causality preserved instead. Since these traces are easier to understand when visualized (they always end with a client reply) and always show all work done on requests' critical paths, trigger causality should be preserved for diagnosis tasks, which often involve answering questions of the form "Why is this request so slow?"

Indeed, switching from preserving submitter causality to preserving trigger causality was perhaps the most important change we made to the original version of Stardust [47] (useful for resource attribution) to make it useful for diagnosis tasks [40]. Many other tracing implementations implicitly preserve this slice of causality [11, 19, 37, 43].

### 3.1.3 Is anything gained by preserving both?

The slices suggested above are the most important ones that should be preserved for various use cases, not the only ones that should be preserved. Indeed, preserving both submitter causality and trigger causality will enable a deeper understanding of the distributed system than is possible by preserving only one of them. For example, for diagnosis, preserving submitter causality in addition to trigger causality will allow the tracing infrastructure to answer questions such as "Who was responsible for evicting my client's cached data?" or, more generally, "Which clients tend to interfere with each other most?"

### 3.1.4 Preserving workflow structure (concurrency, forks, and joins)

For both submitter-preserving causality and trigger-preserving causality, preserving workflow structure—concurrent behaviour, forks, and joins—is optional. It is not necessary for some use cases, such as resource attribution or profiling. However, it is useful to preserve them for diagnosis tasks. Preserving concurrency and forks allows developers to diagnose problems due to excessive parallelism or too little parallelism. Additionally, preserving joins allows developers to diagnose excessive waiting at synchronization points and allows them to easily identify critical paths.

The original version of X-Trace [20] used trees to model causal relationships and so could not preserve joins. The original version of Stardust [47] used DAGs, but did not instrument joins. To become more useful for diagnosis tasks, in their revised versions [19,40], X-Trace evolved to use DAGs and both evolved to explicitly include APIs for instrumenting joins.

### 3.2 Preserving inter-request slices

In addition to relationships within a request, many types of causal relationships may exist between requests. This section describes the two most common ones.

**The contention-preserving slice**: Requests may compete with each other for resources, such as access to a shared variable. Preserving causality between requests holding a resource lock and those waiting for it can help explain unexpected performance slowdowns or timeouts. Only Whodunit [9] preserves this slice.

**The-read-after-write-preserving slice**: Requests that read data (e.g., from a cache or file) written by others may be causally affected by the contents. For example, a request that performs work dictated by the contents of a file—e.g., a map-reduce job [15]—may depend on that file's original writer. Preserving read-after-write dependencies can help explain such requests' behaviour.

## 4 How should causal relationships be tracked?

All end-to-end tracing infrastructures must employ a mechanism to track the slices of intra-request and inter-request causality most relevant to their intended use cases. To avoid capturing superfluous relationships (e.g., portions of undesired slices or false causal relationships), tracing infrastructures "thread" metadata along with individual workflows and establish happens-before relationships only to items with the same (or related) metadata [9,11,19,20,37,40,43,47,48]. Section 4.1 describes different types of metadata and tradeoffs between them.

In general, metadata can be propagated by storing it in thread-local variables when a single thread is performing causally-related work, and encoding logic to propagate metadata across boundaries (e.g., across threads, caches, or components) in commonly used libraries. We argue that systems should be designed with the ability to propagate reasonably generic metadata with their flow of execution and messages, as this is a key underlying primitive of all tracing infrastructures we describe.

Though any of the approaches discussed below can preserve concurrency by establishing happens-before relationships, additional instrumentation is needed to capture forks and joins. Such *structural trace points* are discussed in Section 4.2. Of course, the causal-relationship model used by the tracing infrastructure must also be expressive enough to represent concurrency, forks, and joins.

### 4.1 Tradeoffs between metadata types

Per-workflow metadata can either be static or dynamic. Dynamic metadata can additionally be fixed-width or variable-width. There are three main issues to consider when determining which type of metadata to use. First is size. Larger metadata will result in larger messages (e.g., RPCs) or will constrain payload size. Second is brittleness (or resilience) to lost or unavailable data. Third is whether the approach enables immediate availability of full traces (or other data needed for analysis) without trace construction.

Comparing the three approaches, fixed-width approaches limit metadata size compared to variable-width approaches. All fixed-width approaches are also brittle to data availability or loss, though in different ways and to differing degrees. Dynamic, variable-width approaches can be extremely resilient to data loss, but at the cost of metadata size. Additionally, dynamic, variable-width approaches are often necessary to avoid trace construction. Table 3 summarizes the tradeoffs between the various metadata-propagation approaches. The rest of this section describes them in more detail.

**Static, fixed-width metadata**: With this approach, a single metadata value (e.g., a randomly chosen 64-bit workflow ID) is used to identify all causally-related activity. Tracing implementations that use this method must explicitly construct traces by joining trace-point records with the same metadata. When doing so, they must rely on clues stored with trace-point records to establish happens-before relationships. For example, to order causally-related activity within a single thread, they must rely on an external clock. Since network messages must always be sent by a client before being received by a server, tracing infrastructures that do not rely on synchronized clocks might establish happens-before relationships between client and server work using network send and receive trace points on both machines. To identify concurrent work within components, tracing implementations that use this approach might establish happens-before relationship via thread IDs. Pip [37], Pinpoint [11], and Quanto [18] use static, fixed-width metadata.

This approach is brittle because it will be unable to properly order activity in cases where the external clues are lost (e.g., due to losing trace-point records) or are unavailable (e.g., because developers are not blessed with the ability to modify arbitrary sections of the distributed system's codebase). For example, if thread IDs are lost or are not available, this approach might not be able to properly identify concurrent activity within a component.

**Dynamic, fixed-width metadata**: With this approach, simple logical clocks (i.e., single 64-bit values), in addition to a workflow ID, can be embedded within metadata, enabling tracing infrastructures to encode happens-before relationships without relying on external clues. To limit metadata size, a single logical timestamp is used. Vector clocks are not feasible with fixed-width metadata because they would require metadata as wide as the number of threads in the entire distributed system. At each trace point, a new random logical-clock value is chosen and a happens-before relationship is created by storing both new and old logical-clock values in the corresponding trace record. Counters that are incremented at each trace point could also be used to implement logical clocks, but would be insufficient for ordering concurrent accesses. Both versions of X-Trace [19, 20] use dynamic, fixed-width metadata. Dapper [43] and both versions of Stardust [40, 47] use a hybrid approach that combines the previous approach and this one. For example, Stardust [40, 47] relies on an external clock to order activity within components and uses logical clocks to order inter-component accesses.

The dynamic, fixed-width approach is also brittle because it cannot easily order trace-point records when a subset of them are lost. For example, if a single trace-point record is lost, this approach will be unable to order the two trace fragments that surround it because both will have completely different logical-clock values for which no explicit happens-before relationship exists. Hybrid approaches, which do not change metadata values as often, are slightly less susceptible to this problem than approaches that always change metadata between trace points. Other approaches are also possible to reduce brittleness, but at the expense of space.

| **Type** | **Resilient (~Brittle)** | **Traces avail. immediately** | **Constant size** | **Use cases** |
|---|---|---|---|---|
| Static | | | ✓ | All |
| Dynamic, fixed-width | | | ✓ | ” |
| Hybrid, fixed-width | — | | ✓ | ” |
| Dynamic, variable-width | ✓ | ✓ | | ” |

**Table 3: Tradeoffs between metadata types.** Static and dynamic, fixed-width approaches are of constant size (e.g., a minimum of one or two 64-bit values), but are brittle and do not enable immediate use of trace data. Dynamic variable-width approaches can enable resiliency by incorporating interval-tree clocks and can be used to obtain traces immediately, but the resulting metadata can be very large (e.g., its size could be proportional to the amount of intra-request concurrency and number of functions executed). Hybrid approaches represent a good inflection point because they are less brittle than pure static or dynamic approaches and are of constant size (e.g., a minimum of two 64-bit values).

**Dynamic, variable-width metadata**: With this approach, metadata assigned to causally-related activity can change in size in addition to value. Doing so would allow metadata to include interval-tree clocks [3] instead of simple logical clocks. Like vector clocks, interval-tree clocks reduce brittleness since any two timestamps can be compared to determine if they are concurrent or if one happened before another. But, unlike vector clocks, interval-tree clocks can grow and shrink in proportion to the number of active threads. In contrast, variable-width vector clocks cannot shrink and so require width proportional to the maximum number of threads observed in a workflow. Vector clocks also require globally unique, well-known thread IDs [3]. Currently, no existing tracing infrastructure uses vector clocks or interval-tree clocks.

Tracing infrastructures that wish to make full traces (or other data that requires tying together causally-related activity) available immediately without explicit trace construction must use dynamic, variable-width metadata. For example, tracing infrastructures that use dynamic, variable-width metadata could carry executed trace-point records within metadata, making them immediately available for use as soon as the workflow ends. Whodunit [9] is the only existing tracing implementation that carries trace-point records (i.e., function names) in metadata. To reduce metadata size, heuristics are used to reduce the number of propagated trace-point records, but trace fidelity is reduced as a result.

### 4.2 How to preserve forks and joins

For the static and dynamic, fixed-width metadata-propagation approaches discussed above, forks and joins can be preserved via one-to-many and many-to-one trace points. For the static approach, such trace points must include clues that uniquely identify the activity being forked or waited on—for example, thread IDs. For dynamic, fixed-width approaches, one-to-many trace points should include the current logical-clock value and the logical-clock values that will be initially used by each of the forked descendants. Join trace points should include the current logical-clock value and the logical-clock values of all events that must complete before work can proceed. Dynamic, variable-width approaches can infer forks and joins if they include interval-tree clocks.

An alternate approach, used by Mann et al. [31], involves comparing large volumes of traces to automatically determine fork and join points.

## 5 How should sampling be used to reduce overhead?

Sampling determines which trace-point records are persisted by the tracing infrastructure. It is the most important technique used by end-to-end tracing infrastructures to limit runtime and storage overhead [9, 19, 40, 43, 47]. For example, even though Dapper writes trace-point records to stable storage asynchronously (i.e., off the critical path of the distributed system), it still imposes a 1.5% throughput and 16% response time overhead when persisting all trace points executed by a web search workload [43]. When using sampling to capture just 0.01% of all trace points, the slowdown in response times is reduced to 0.20% and in throughput to 0.06% [43]. Even when trace-point records need not be persisted because the required analyses can be performed online, sampling is useful to limit the sizes of analysis-specific data structures [9].

There are three fundamentally different options for deciding what trace points to sample: head-based coherent sampling, tail-based coherent sampling, or unitary sampling. Coherent sampling methods, which guarantee that all or none of the trace points executed by a workflow will be sampled, must be used if traces showing causally-related activity are to be constructed. Additionally, head-based sampling will often result in high overheads if it is used to preserve submitter causality. Figure 4 illustrates the tradeoffs between the different sampling schemes when used to preserve different causality slices. The rest of this section further describes the sampling schemes.

**Head-based coherent sampling**: With this method, a random sampling decision is made for entire workflows at their start (e.g., when requests enter the system) and metadata is propagated along with workflows

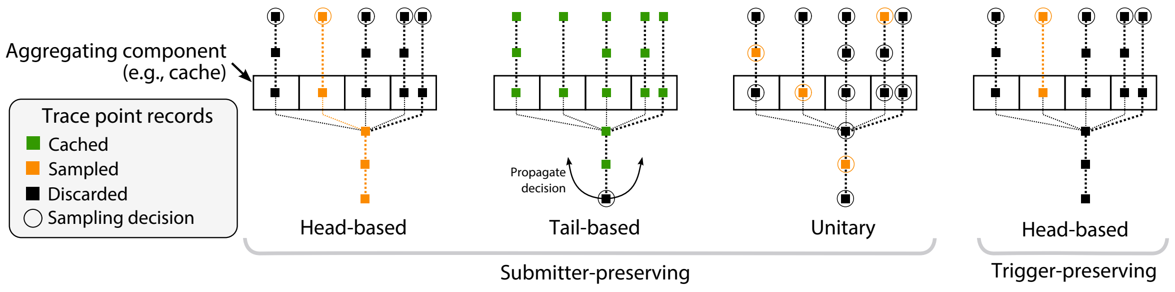

**Figure 4: Trace points that must be sampled as a result of using different sampling schemes and preserving different causality slices.** In this example, the right-most workflow in each of the four diagrams causes an on-demand eviction and, as part of this process, aggregates latent work stored in other cache blocks. Head-based sampling greedily decides whether to sample workflows at their start. As such, when preserving submitter causality using head-based sampling, all trace points executed by workflows that aggregate latent work (e.g., the workflow forcing the on-demand eviction in this example) must be sampled if any one of the aggregated set was inserted into the system by a sampled workflow. With each aggregation, the probability of sampling individual trace points will increase, resulting in high storage and runtime overheads. Since tail-based sampling defers sampling decisions to when workflows finish, it does not inflate the trace-point-sampling probability as a result of aggregations. However, it requires that records of trace points executed by workflows that leave latent work be cached until the latent work is aggregated. Unitary sampling also does not suffer from sampling inflation because it does not attempt to coherently capture workflows' trace-point records. However, with unitary sampling, data required to obtain traces or for needed analysis must be propagated with metadata and stored in trace-point records. The rightmost diagram shows that head-based sampling can be used to preserve trigger causality with low overhead because latent work is always attributed to the aggregator. As such, only the sampling decision made for the aggregator matters when deciding whether to sample the trace points it executes.

indicating whether to collect their trace points. The percentage of workflows randomly sampled is controlled by setting the workflow-sampling percentage. When used in conjunction with tracing infrastructures that preserve trigger causality, the workflow-sampling percentage and the trace-point-sampling percentage (i.e., the percentage of trace points executed that are sampled) will be the same. Due to its simplicity, head-based coherent sampling is used by many existing tracing implementations [19, 40, 43].

Head-based coherent sampling will not reduce runtime and storage overhead for tracing infrastructures that preserve submitter causality. This is because the effective trace-point-sampling percentage will almost always be much higher than the workflow-sampling percentage. To understand why, recall that preserving submitter causality means that latent work is attributed to the original submitter. So, when latent work is aggregated by another request or background activity, trace points executed by the aggregator must be sampled if *any one* of the aggregated set was inserted into the system by a sampled workflow. In many systems, this process will result in sampling almost all trace points deep in the system. For example, if head-based sampling is used to sample trace points for only 0.1% of workflows, the probability of sampling an individual trace point will also be 0.1% before any aggregations. However, after aggregating 32 items, this probability will increase to 3.2% and after two such levels of aggregation, the trace-point-sampling percentage will increase to 65%. The leftmost diagram in Figure 4 illustrates this inflationary process for one level of aggregation. The overall effective trace-point-sampling percentage depends on several parameters, including the workflow-sampling percentage, the number of aggregation levels, and the number of trace points between aggregation levels.

When developing the revised version of Stardust [40], we learned of this incompatibility between head-based coherent sampling and submitter causality the hard way. Head-based sampling was the first feature we added to the original Stardust [47], which previously did not use sampling and preserved submitter causality. But, at the time, we didn't know anything about causality slices or how they interact with different sampling techniques. So, when we applied the sampling-enabled Stardust to our test distributed system, Ursa Minor [1], we were very confused as to why the tracing overheads did not decrease. Of course, the root cause was that

Ursa Minor contained a cache very near the entry point to the system, which aggregated 32 items at a time. We were using a sampling rate of 10%, meaning that 97% all trace points executed after this aggregation were always sampled.

**Tail-based coherent sampling**: This method is similar to the previous one, except that the workflow-sampling decision is made at the end of workflows, instead of at their start. Delaying the sampling decision allows for more intelligent sampling—for example, the tracing infrastructure can examine a workflow's properties (e.g., response time) and choose only to collect anomalous ones. But, trace-point records for every workflow must be cached somewhere until the sampling decision is made for them. Because many workflows can execute concurrently, because each request can execute many trace points, and because workflows with latent work will remain in the system for long periods of time, such temporary collection is not always feasible.

Tail-based sampling avoids inflating the trace-point-sampling percentage because it does not commit to a sampling decision upfront. As such, it can be used to preserve submitter causality with low runtime and storage overhead. For workflows that carry aggregated work, tail-based sampling guarantees that either all or none of the trace points executed by workflows whose work has been aggregated are sampled. Accomplishing this requires maintaining a mapping between aggregators' workflow IDs and the IDs of the workflows whose work they have aggregated. The second-leftmost diagram in Figure 4 illustrates the trace-point records that must be cached when preserving submitter causality with tail-based sampling. Due to its high memory demand, tail-based sampling is not used by most tracing infrastructures.

Some tracing infrastructures use a hybrid scheme, in which they nominally use head-based coherent sampling, but also cache records of recently executed trace points in per-node circular buffers. The circular buffers are often sized to guarantee a request's trace-point records will not be evicted as long as it's execution time does not exceed the $50^{th}$ or $75^{th}$ percentile. This technique allows tracing infrastructures to backtrack and collect traces for non-sampled workflows that appear immediately anomalous (e.g., fail or return an error code soon after starting execution). However, it is not sufficient for performance anomalies (e.g., requests that take a very long time to execute).

**Unitary sampling**: With this method, developers set the trace-point-sampling percentage directly and the sampling decision is made at the level of individual trace points. No attempt is made at coherence (i.e., capturing all trace points associated with a given workflow), so traces cannot be constructed using this approach. This method is best for use cases, such as resource attribution, where the information needed for analysis can be propagated with workflows (assuming dynamic, variable-width metadata) and retrieved at individual trace points directly.

In addition to deciding how to sample trace points, developers must decide how many of them to sample. Many infrastructures choose to randomly sample a small, set percentage—often between 0.01% and 10%—of trace points or workflows [9, 19, 40, 43]. However, this approach will capture only a few trace points for small workloads, limiting its use for them. Using per-workload sampling percentages can help, but this requires knowing workload sizes a priori. A more robust solution, proposed by Sigelman at al. [43], is an adaptive scheme, in which the tracing infrastructure aims to always capture a set rate of trace points or workflows (e.g., 500 trace points/second or 100 workflows/second) and dynamically adjusts the trace-point- or workflow-sampling percentage to accomplish this set goal. Though promising, care must be taken to avoid biased results when the captured data is used for statistical purposes. For distributed services built on top of shared services, the adaptive sampling rate should be based on the tracing overhead the lowest-tier shared service can support (e.g., Bigtable [10]) and proportionately propagated backward to top-tier services.

## 6 How should traces be visualized?

Good visualizations are important for use cases such as diagnosis and profiling. Effective visualizations will amplify developers' efforts, whereas ineffective ones will hinder their efforts and convince them to use other tools and techniques [30, 39]. Indeed, Oliner et al. identify visualization as one of the key future challenges in diagnosis research [34]. This section highlights common approaches to visualizing end-to-end traces. The choices between them depend on the visualization's intended use, previous design choices, and whether precision (i.e., the ability to show forks, joins, and concurrency) is preferred over volume of data shown. Furthermore, the underlying trace representation limits which visualizations can be used. DAGs can support any of the approaches in this section. All but flow graphs can also be built from directed trees.

Table 4 summarizes the tradeoffs among the various visualizations. Figure 5 shows how some of the visualizations would differ in showing requests. Instead of visualizing traces, Pip [37] uses an expectation language to describe traces textually. Formal user studies are required to compare the relative benefits of visualizations and expectations, and we make no attempt to do so here.

**Gantt charts (also called swimlanes)**: These visualizations are most often used to show individual traces, but can also be used to visualize multiple requests that have identical workflows. The Y-axis shows the overall request and resulting sub-requests issued by the distributed system, and the X-axis shows relative time. The relative start time and latency (measured in wall-clock time) of items shown on the Y-axis are encoded by horizontal bars. Concurrency can easily be inferred by visually identifying bars that overlap in X-axis values. Forks and joins must also be identified visually, but it is harder to do so. Both ETE [23] and Dapper [43] use Gantt charts to visualize individual traces. In addition to showing latencies of the overall request and sub-requests, Dapper also identifies network time by subtracting time spent at the server from the observed latency of the request or sub-request.

**Flow graphs (also called request-flow graphs)**: These directed-acyclic graphs faithfully show requests' workflows as they are executed by the various components of a distributed system. They are often used to visualize and show aggregate information about multiple requests that have identical workflows. Since such requests are often expected to perform similarly, flow graphs are a good way to preserve precision, while still showing multiple requests. Fan-outs in the graph represent the start of concurrent activity (forks), events on different branches are concurrent, and fan-ins represent synchronization points (joins). The revised version of Stardust [40] and the revised version of X-Trace [19] visualize traces via flow graphs.

**Call graphs and focus graphs**: These visualizations are also often used to show multiple traces, but do not show concurrency, forks, or joins, and so are not precise. Call graphs use fan-outs to show functions accessed by a parent function. Focus graphs show the call stack to a chosen component or function, called the "focus node," and the call graph that results from treating the focus node as its root. In general, focus graphs are best used for diagnosis tasks for which developers already know which functions or components are problematic. Dapper [43] uses focus graphs to show multiple requests with identical workflows, but owing to its RPC-oriented nature, nodes do not represent components or functions, but rather all work done to execute an RPC at the client and server. Note that when used to visualize multiple requests with different workflows,

| | **Precision** | | | **Many flows?** | |
|---|---|---|---|---|---|
| | **Forks** | **Joins** | **Conc.** | **Same** | **Different** |
| Gantt charts | I | I | I | Y | N |
| Flow graphs | Y | Y | Y | Y | N |
| Call & focus graphs | N | N | N | Y | Y* |
| CCTs | N | N | N | Y | Y |

**Table 4: Tradeoffs between trace visualizations.** Different visualizations differ in precision—i.e., if they can show forks, joins and concurrency ("Y"), or if it must be inferred ("I"). They also differ in their ability to show multiple workflows, and whether those multiple workflows can be different. To our knowledge, these visualizations have been used to show traces that contain up to a few hundred trace points. Note that though call graphs and focus graphs are sometimes used to visualize multiple different workflows, they will show infeasible paths when used to do so.

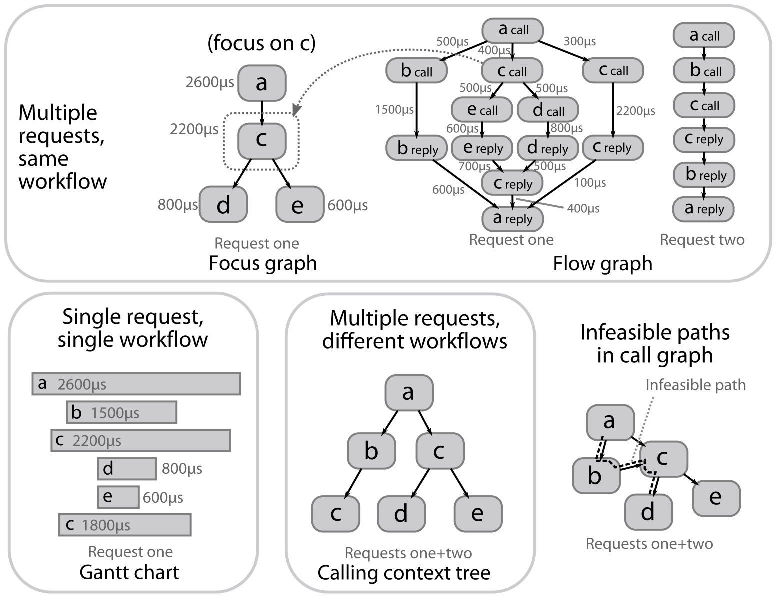

**Figure 5: Comparison of various approaches for visualizing traces.** Gantt charts are often used to visualize individual requests. Flow graphs allow multiple requests with identical workflows to be visualized at the same time while showing forks, joins, and concurrency. However, they must show requests with different workflows separately (as shown by requests one and two). CCTs trade precision for the ability to visualize multiple requests with different workflows (e.g., an entire workload). Call graphs can also show multiple workflows, but may show infeasible paths that did not occur in an actual execution. For example, see the $a \rightarrow b \rightarrow c \rightarrow d$ path in the call graph shown, which does not appear in either request one or two.

call graphs can show infeasible paths [4]. This is demonstrated by the $a \rightarrow b \rightarrow c \rightarrow d$ path for the call graph shown in Figure 5.

**Calling Context Trees (CCTs) [4]**: These visualizations are best used to show multiple requests with different workflows, as they guarantee that every path from root to leaf is a valid path through the distributed system. To do so in a compact way, they use fan-outs to show function invocations, not forks, and, as such, are not precise. CCTs can be constructed in amortized constant time and are best used for tasks for which a high-level summary of system behaviour is desired (e.g., profiling). Whodunit [9] uses CCTs to show profiling information for workloads.

## 7 Putting it all together

Based on the tradeoffs described in previous sections and our experiences, this section identifies good design choices for the key uses of end-to-end tracing. We also show previous implementations' choices and contrast them to our suggestions.

### 7.1 Suggested choices

The italicized rows of Table 5 show suggested design choices for key use cases of end-to-end tracing. For the causal-tracking mechanism, we suggest the hybrid static/dynamic, fixed-width approach for most use cases because it requires constant size, reduces the need for external clues, and is less brittle than the straightforward dynamic, fixed-width approach. Static metadata is also a good choice if the needed external clues (e.g., to establish happens-before relationships) will always be available or if many events that require clues, such as forks, joins, and concurrency, need not be preserved. Developers should consider using variable-width

| | | Design axes | | | | |
|---|---|---|---|---|---|---|
| **Use** | **Name** | **Sampling** | **Causality slices** | **Forks/ joins/conc.** | **Metadata** | **Visualization** |
| Anomaly detection | *Suggested* | *Coherent (T)* | *Trigger* | *Yes* | *S/DF* | *Flow graphs* |
| | Magpie [5] | No | Any | ” | None | Gantt charts (V) |
| | Pinpoint [11] | ” | Trigger | No | S | Paths |
| Diagnosing steady-state problems | *Suggested* | *Coherent (H)* | *Trigger* | *Yes* | *S/DF* | *Flow graphs* |
| | Stardust‡ [40] | ” | ” | ” | ” | ” |
| | X-Trace‡ [19] | ” | ” | ” | DF | ” |
| | Dapper [43] | ” | ” | Forks/conc. | S/DF | Gantt charts & focus graphs |
| | Pip [37] | No | ” | Yes | S | Expectations |
| | X-Trace [20] | ” | Trigger & TCP layers | Forks/conc. | DF | Call graphs & network layers |
| | Pinpoint [11] | ” | Trigger | No | S | Paths |
| Distributed profiling | *Suggested* | *Unitary* | *Either* | *No* | *DV* | *CCTs* |
| | Whodunit [9] | ” | Submitter | ” | ” | ” |
| | Dapper [43] | Coherent (H) | Trigger | Forks/conc. | S/DF | Gantt charts & focus graphs |
| | ETE [23] | No | Any | No | None | Gantt charts |
| Resource attribution | *Suggested* | *Unitary* | *Submitter* | *No* | *DV* | *None* |
| | Stardust [47] | No | ” | Forks/conc. | S/DF | Call graphs |
| | Quanto [18] | No | ” | No | S | None |
| Workload modeling | *Suggested* | *Depends* | *Depends* | *Depends* | *Depends* | *Flow graphs or CCTs* |
| | Magpie [5] | No | Depends | Yes | None | Gantt charts (V) |
| | Stardust [47] | No | Submitter | Forks/conc. | S/DF | Call graphs |

**Table 5: Suggested design choices for various use cases and choices made by existing tracing implementations.** Suggested choices are shown in italics. Existing implementations' design choices are qualitatively ordered according to similarity with our suggested choices. For comparison, two schema-based approaches, Magpie and ETE, are also included. The revised versions of Stardust and X-Trace are denoted by *Stardust‡* and *X-Trace‡*. Static approaches to metadata propagation are denoted by *S*, Dynamic, fixed-width approaches by *DF*, Hybrid, fixed-width approaches by *S/DF*, and dynamic, variable-width approaches by *DV*. *(V)* indicates that a variant of the stated item is used.

approaches if feasible. For use cases that require coherent sampling, we conservatively suggest the head-based version when it is sufficient, but tail-based based coherent sampling should also be considered since it subsumes the former and allows for a wider range of uses. The rest of this section explains design choices for the various use cases.

**Anomaly detection**: This use case involves identifying rare workflows that are extremely different from others so that developers can analyze them. As such, tail-based coherent sampling should be used so that traces can be constructed and so that the tracing infrastructure can gauge whether a workflow is anomalous before deciding whether or not to sample it. Either trigger causality or submitter causality can be preserved with low overhead. But, the former should be preferred since workflows that show trigger causality show critical paths and are easier to understand when visualized. To identify anomalies that result from excessive parallelism, insufficient parallelism, or excessive waiting for one of many concurrent operations to finish, implementations should preserve forks, joins, and concurrent behaviour. Flow graphs are best for visualizing anomalies because they are precise and because anomaly detection will, by definition, not generate many results. Gantt charts can also be used.

**Diagnosing steady-state problems**: This use case involves diagnosing performance and correctness problems that can be observed in many requests. Design choices for it are similar to anomaly detection, except that head-based sampling can be used, since, even with low sampling rates, it is unlikely that problems will go unnoticed.

**Distributed profiling**: This use case involves sampling function or inter-trace-point latencies. The inter- and intra-component call stacks to a function must be preserved so that sampled items can be grouped together based on context, but complete traces need not be constructed. Since call stacks can be represented compactly [9], these requirements align well unitary with sampling and dynamic, variable-width metadata-propagation approaches. Combined, these options allow for trace-point records to be carried as metadata and profiles to be collected online. If metadata size is a concern, fixed-width metadata, combined with head-based or tail-based sampling can be used as well, but online profiling will not be possible. Call stacks do not need to preserve forks, joins, or concurrency. CCTs are best for visualizing distributed profiles, since they can show entire workloads and infeasible paths do not appear.

**Resource attribution**: This use case involves attributing work done at arbitrary levels of the system to the original submitter, so submitter causality must be preserved. Resource attribution is best served by dynamic, variable-width metadata-propagation approaches and unitary sampling. This combination will allow client IDs of aggregated items to be carried in metadata, thus enabling immediate, online analyses without having to construct traces. If metadata size is a concern, tail-based sampling and fixed-width metadata could be used instead, but online, immediate analyses will not be possible. Head-based sampling can be used, but will likely result in high overheads because it will result in sampling almost all trace points after a few levels of aggregation. Though forks, joins, and concurrency need not be preserved, DAGs must be used as the underlying data model to preserve relationships between original submitters and aggregated work. Visualization is not necessary for this use case.

**Workload modeling**: The design decisions for this use case depend on what properties of the workload are being modeled. For example, when used to model workloads, Magpie [5] aims to identify a set of flows and associated resource usages that are representative of an entire workload. As such, it is useful for Magpie to preserve forks, joins, and concurrent behaviour. If traces for this use case are to be visualized, flow graphs or CCTs should be used, since they allow for visualizing multiple traces at one time.

## 7.2 Existing implementations' choices

Table 5 also lists how existing tracing implementations fit into the design axes suggested in this paper. Tracing implementations are grouped by the use case for which they are most suited (a tracing implementation may be well suited for multiple use cases). For a given use case, tracing implementations are ordered according to similarity in design choices to our suggestions. This ordering shows that that tracing implementations suited for a particular use case tend to make similar design decisions to our suggestions for that use case. The rest of this section describes key cases where our suggestions differ from tracing implementations' choices.

For anomaly detection, we suggest tail-based sampling, but both Magpie [5] and Pinpoint [11] do not use any sampling techniques whatsoever. Collecting and storing trace points for every request guarantees that both implementations will not miss capturing any rare events (anomalies), but also means they cannot scale to handle large workloads. Magpie cannot use sampling, because it does not propagate metadata. Pinpoint is concerned mainly with correctness anomalies, and so does not bother to preserve concurrency, forks, or joins.

For diagnosing steady-state problems, we suggest that forks, joins, and structure be explicitly preserved, but Dapper [43] cannot preserve joins because it uses a tree as its model for expressing causal relationships, not a DAG. Recent work by Mann et al. [31] focuses on learning join-point locations by comparing large volumes of Dapper traces. Dapper traces are then reformatted to show the learned join points. Pip [37] also differs from many other tracing implementations in that it uses an expectation language to show traces. Pip's expectation language describes how other components interact with a component of interest and so is similar in functionality to focus graphs. Both are best used when developers already have a component-of-interest in mind, not for problem-localization tasks.

Both the revised version of Stardust [40] and the revised version of X-Trace [19] were created as a result of modifying their original versions [20, 47] to be more useful for diagnosis tasks. Both revised versions independently converged to use almost the same design choices. Sambasivan et al. initially tried to use the original version of Stardust, which was designed with resource attribution in mind, for diagnosis, but found it insufficient, motivating the need for the revised version. The original X-Trace was designed by Fonseca et al. to help with diagnosis tasks. But, as a result of experiences applying X-Trace to additional real systems [19], they eventually found the design choices listed for the revised version to be more useful than the ones they originally chose.

For distributed profiling, existing infrastructures either meet or exceed our suggestions. For resource attribution, existing implementations do not use sampling and hence cannot scale.

## 8 Challenges & opportunities

Though end-to-end tracing has proven useful, many important challenges remain before it can reach its full potential. They arise in collecting and presenting trace data, as a result of the complexity and volume of traces generated by today's large-scale distributed systems. Also, we have only touched the tip of the iceberg in developing analysis techniques for end-to-end traces; many opportunities remain to better exploit this rich data source.

### 8.1 Challenges in trace collection

As instrumented systems scale both in size and workload, tracing infrastructures must accommodate larger, more complex, traces at higher throughput, while maintaining relevance of tracing data. Though head-based sampling meets the first two criteria of this key challenge, it does not guarantee trace relevance. For example, it complicates diagnostics on specific traces and will not capture rare bugs (i.e., anomalies). Conversely, tail-based sampling, in which trace points are cached until requests complete, meets the relevance criteria, but not the first two.

An in-between approach, in which all trace points for requests are discarded *as soon as* the request is deemed uninteresting, seems a likely solution, but important research into finding the trace attributes that best determine when a trace can be discarded is needed before this approach can be adopted. An alternate approach may be to collect low-resolution traces in the common case and to increase resolution only when a given trace is deemed interesting. However, this approach also requires answering similar research questions as that required for the in-between approach.

Another challenge, which end-to-end tracing shares with logging, involves trace interpretability. In many cases, the developers responsible for instrumenting a distributed system are not the same as those tasked with using the resulting traces. This leads to confusion because of differences in context and expertise. For example, in a recent user study, Sambasivan et al. had to manually translate the trace-point names within end-to-end traces from developer-created ones to ones more readily understood by general distributed-systems experts [39]. To help, key research must be conducted on how to define good instrumentation practices, how to incentivize good instrumentation, and how to educate users about how to interpret instrumented traces or logs. Research into automatic instrumentation and on the fly re-instrumentation (e.g., as in DTrace [8]) can also help reduce instrumentation burden and help interpretability.

A final important challenge lies in the integration of different end-to-end tracing infrastructures. Today's distributed services are composed of many independently-developed parts, perhaps instrumented with different tracing infrastructures (e.g., Dapper [43], Stardust [40, 47], Tracelytics [49], X-Trace [19, 20], or Zipkin [50]). Unless they are modified to be interoperable, we miss the opportunity to obtain true end-to-end traces of composed services. The provenance community has moved forward in this direction by creating the Open Provenance Model [33], which deserves careful examination.

### 8.2 Challenges in visualization

As the volume and size of end-to-end traces increase, a key challenge lies in understanding how to visualize them effectively. The techniques described in Section 6 often only scale to a few hundred trace points at best, but many distributed services can generate a lot more. For example, Wang [52] describes how tools like Graphviz [22] cannot effectively visualize HDFS traces, which, due to its large write sizes of 64MB or more, can generate individual traces comprised of 1,000s of trace points. Even navigating graphs with 100s of trace points was challenging for users in a study by Sambasivan et al. [39]. Higher-level summaries of large end-to-end traces could help, but research is needed into how to preserve appropriate attributes, include multiple levels of detail, and summarize and collapse similar subgraphs in meaningful ways. Interactivity seems paramount for allowing users to filter, query, and display only relevant information.

### 8.3 Opportunities in trace analysis

The use cases of end-to-end tracing described in this paper represent only a handful of all potential ones. Significant opportunities remain to discover more. For example, one recent research effort focuses on using end-to-end traces to automatically identify and assign extra resources to bottlenecked services in large distributed systems [36]. Many research opportunities also remain for the use cases already identified in this paper. For example, for diagnosis, longitudinal comparisons across traces are useful to identify outliers or undesirable differences between distributed systems components. Spectroscope [40], which uses statistical techniques to identify timing variations among graphs and simple heuristics to compare their structures, is an initial step, but is not sufficient. Research into whether more advanced techniques, such as graph kernels [42] and frequent-subgraph mining [24], can be used for such comparisons is needed. Along these lines, Eberle et al. [16] present potentially useful techniques for identifying structural anomalies. Despite their promise, it remains to be seen whether these techniques can scale while simultaneously accounting for the structure, labels, timings, and domain-specific semantics present in end-to-end traces.

## 9 Conclusion

End-to-end tracing can be implemented in many ways, and the choices made dictate the utility of the resulting traces for different development and management tasks. Based on our experiences developing tracing infrastructures and past research on the topic, this paper provides guidance to designers of such infrastructures and identifies open questions for researchers.

## References

[1] Michael Abd-El-Malek, William V. Courtright II, Chuck Cranor, Gregory R. Ganger, James Hendricks, Andrew J. Klosterman, Michael Mesnier, Manish Prasad, Brandon Salmon, Raja R. Sambasivan, Shafeeq Sinnamohideen, John Strunk, Eno Thereska, Matthew Wachs, and Jay Wylie. Ursa minor: versatile cluster-based storage. In *FAST'05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, December 2005. Cited on page 12.

[2] Marcos K. Aguilera, Jeffrey C. Mogul, Janet L. Wiener, Patrick Reynolds, and Athicha Muthitacharoen. Performance debugging for distributed systems of black boxes. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003. Cited on pages 1 and 4.

[3] Paulo S. Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks: a logical clock for dynamic systems. In *OPODIS '08: Proceedings of the 12th International Conference on Principles of Distributed Systems*, 2008. Cited on page 11.

[4] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *PLDI '97: Proceedings of the 11th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997. Cited on page 15.

[5] Paul Barham, Austin Donnelly, Rebecca Isaacs, and Richard Mortier. Using Magpie for request extraction and workload modelling. In *OSDI '04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004. Cited on pages 1, 3, 4, 16, and 17.

[6] Ledion Bitincka, Archana Ganapathi, Stephen Sorkin, and Steve Zhang. Optimizing data analysis with a semi-structured time series database. In *SLAML '10: Proceedings of the 1st USENIX Workshop on Managing Systems via Log Analysis and Machine Learning Techniques*, 2010. Cited on page 4.

[7] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006. Cited on page 1.

[8] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *ATC '04: Proceedings of the 2004 USENIX Annual Technical Conference*, 2004. Cited on page 18.

[9] Anupam Chanda, Alan L. Cox, and Willy Zwaenepoel. Whodunit: transactional profiling for multi-tier applications. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems*, 2007. Cited on pages 1, 3, 4, 5, 7, 8, 9, 11, 13, 15, 16, and 17.

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006. Cited on pages 1 and 13.

[11] Mike Y. Chen, Anthony Accardi, Emre Kiciman, Jim Lloyd, David Patterson, Armando Fox, and Eric Brewer. Path-based failure and evolution management. In *NSDI '04: Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation*, 2004. Cited on pages 1, 3, 4, 5, 6, 8, 9, 10, 16, and 17.

[12] David R. Cheriton and Dale Skeen. Understanding the limitations of causally and totally ordered communication. In *SOSP '93: Proceedings of the 14th ACM Symposium on Operating Systems Principles*, 1993. Cited on page 6.

[13] Cloudera HTrace. http://github.com/cloudera/htrace. Cited on page 1.

[14] Compuware dynaTrace PurePath. http://www.compuware.com. Cited on page 1.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. In *OSDI '04: Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation*, 2004. Cited on page 9.

[16] William Eberle and Lawrence B. Holder. Discovering structural anomalies in graph-based data. In *ICDMW '07: Proceedings of the 7th IEEE International Conference on Data Mining Workshops*, 2007. Cited on page 19.

[17] Ulfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. Fay: extensible distributed tracing from kernels to clusters. In *SOSP '11: Proceedings of the 23nd ACM Symposium on Operating Systems Principles*, 2011. Cited on page 5.

[18] Rodrigo Fonseca, Prabal Dutta, Philip Levis, and Ion Stoica. Quanto: tracking energy in networked embedded systems. In *OSDI '08: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 2008. Cited on pages 1, 3, 4, 7, 10, and 16.

[19] Rodrigo Fonseca, Michael J. Freedman, and George Porter. Experiences with tracing causality in networked services. In *INM/WREN '10: Proceedings of the 1st Internet Network Management Workshop/Workshop on Research on Enterprise Monitoring*, 2010. Cited on pages 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 16, 18, and 19.

[20] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-Trace: a pervasive network tracing framework. In *NSDI '07: Proceedings of the 4th USENIX Symposium on Networked Systems Design and Implementation*, 2007. Cited on pages 1, 2, 3, 4, 5, 6, 9, 10, 16, 18, and 19.

[21] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003. Cited on page 1.

[22] Graphviz - graph visualization software. http://www.graphviz.org. Cited on page 19.

[23] Joseph L. Hellerstein, Mark M. Maccabe, W. Nathaniel Mills III, and John J. Turek. ETE: a customizable approach to measuring end-to-end response times and their components in distributed systems. In *ICDCS '99: Proceedings of the 19th IEEE International Conference on Distributed Computing Systems*, 1999. Cited on pages 1, 3, 4, 14, and 16.

[24] Ruoming Jin, Chao Wang, Dmitrii Polshakov, Srinivasan Parthasarathy, and Gagan Agrawal. Discovering frequent topological structures from graph datasets. In *KDD '05: Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2005. Cited on page 19.

[25] Soila P. Kavulya, Scott Daniels, Kaustubh Joshi, Matt Hultunen, Rajeev Gandhi, and Priya Narasimhan. Draco: statistical diagnosis of chronic problems in distributed systems. In *DSN '12: Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks*, 2012. Cited on pages 1 and 4.

[26] Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In *VEE '12: Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, 2012. Cited on page 6.

[27] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *ECCOP'96: Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997. Cited on page 5.

[28] Eric Koskinen and John Jannotti. BorderPatrol: isolating events for black-box tracing. In *Eurosys '08: Proceedings of the 3rd ACM SIGOPS European Conference on Computer Systems*, 2008. Cited on page 4.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978. Cited on page 6.

[30] Zhicheng Liu, Bongshin Lee, Srikanth Kandula, and Ratul Mahajan. NetClinic: Interactive visualization to enhance automated fault diagnosis in enterprise networks. In *VAST '10: Proceedings of the 2010 IEEE Symposium on Visual Analytics Science and Technology*, 2010. Cited on page 14.

[31] Gideon Mann, Mark Sandler, Darja Krushevskaja, Sudipto Guha, and Eyal Even-dar. Modeling the Parallel Execution of Black-Box Services. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*, 2011. Cited on pages 11 and 17.

[32] Matthew L. Massie, Brent N. Chun, and David E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7), July 2004. Cited on page 1.

[33] Luc Moreau, Ben Clifford, Juliana Freire, Joe Futrelle, Yolanda Gil, Paul Groth, Natalia Kwasnikowska, Simon Miles, Paolo Missier, Jim Myers, Beth Plale, Yogesh Simmhan, Eric Stephan, and Van den Bussche. The open provenance model core specification (v1.1). *Future Generation Computer Systems*, 27(6), June 2010. Cited on page 19.

[34] Adam J. Oliner, Archana Ganapathi, and Wei Xu. Advances and challenges in log analysis. *Communications of the ACM*, 55(2), February 2012. Cited on page 14.

[35] Personal communication with Google engineers, 2011. Cited on page 3.

[36] Personal communication with researchers at Carnegie Mellon University, 2012. Cited on page 19.

[37] Patrick Reynolds, Charles Killian, Janet L. Wiener, Jeffrey C. Mogul, Mehul Shah, and Amin Vahdat. Pip: detecting the unexpected in distributed systems. In *NSDI '06: Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation*, 2006. Cited on pages 1, 3, 4, 5, 6, 8, 9, 10, 14, 16, and 17.

[38] Patrick Reynolds, Janet L. Wiener, Jeffrey C. Mogul, Marcos K. Aguilera, and Amin Vahdat. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the 15th ACM International World Wide Web Conference*, 2006. Cited on pages 1 and 4.

[39] Raja R. Sambasivan, Ilari Shafer, Michelle L. Mazurek, and Gregory R. Ganger. Visualizing request-flow comparison to aid performance diagnosis in distributed systems. *IEEE Transactions on Visualization and Computer Graphics (Proceedings Information Visualization 2013)*, 19(12), December 2013. Cited on pages 14, 18, and 19.

[40] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *NSDI'11: Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011. Cited on pages 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 16, 18, and 19.

[41] Raja R. Sambasivan, Alice X. Zheng, Eno Thereska, and Gregory R. Ganger. Categorizing and differencing system behaviours. In *HotAC II: Proceedings of the 2nd workshop on Hot Topics in Autonomic Computing*, 2007. Cited on page 2.

[42] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12, November 2011. Cited on page 19.

[43] Benjamin H. Sigelman, Luiz A. Barroso, Michael Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical Report dapper-2010-1, Google, April 2010. Cited on pages 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 16, 17, and 19.

[44] Byung C. Tak, Chunqiang Tang, Chun Zhang, Sriram Govindan, Bhuvan Urgaonkar, and Rong N. Chang. vPath: precise discovery of request processing paths from black-box observations of thread and network activities. In *USENIX '09: Proceedings of the 2009 USENIX Annual Technical Conference*, 2009. Cited on pages 1 and 4.

[45] Jiaqi Tan, Soila P. Kavulya, Rajeev Gandhi, and Priya Narasimhan. Visual, log-based causal tracing for performance debugging of mapreduce systems. In *ICDCS '10: Proceedings of the 30th IEEE International Conference on Distributed Computing Systems*, 2010. Cited on pages 1 and 4.

[46] The `strace` system call tracer. `http://sourceforge.net/projects/strace/`. Cited on page 1.

[47] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R. Ganger. Stardust: tracking activity in a distributed storage system. In *SIGMETRICS '06/Performance '06: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, 2006. Cited on pages 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 18, and 19.

[48] Brian Tierney, William Johnston, Brian Crowley, Gary Hoo, Chris Brooks, and Dan Gunter. The NetLogger methodology for high performance distributed systems performance analysis. In *HPDC '98: Proceedings of the 7th International Symposium on High Performance Distributed Computing*, 1998. Cited on pages 1 and 9.

[49] Tracelytics. `http://www.tracelytics.com`. Cited on pages 1 and 19.

[50] Twitter Zipkin. `https://github.com/twitter/zipkin`. Cited on pages 1 and 19.

[51] Matthew Wachs, Lianghong Xu, Arkady Kanevsky, and Gregory R. Ganger. Exertion-based billing for cloud storage access. In *HotCloud '11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing*, 2011. Cited on page 4.

[52] William Wang. End-to-end tracing in HDFS. Technical Report CMU-CS-11-120, Carnegie Mellon University, July 2011. Cited on page 19.

[53] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. Detecting large-scale system problems by mining console logs. In *SOSP '09: Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, 2009. Cited on pages 1 and 4.