Canopy: An End-to-End Performance Tracing And Analysis System

Jonathan Kaldor† Jonathan Mace* Michał Bejda† Edison Gao† Wiktor Kuropatwa†
Joe O'Neill† Kian Win Ong† Bill Schaller† Pingjia Shan† Brendan Viscomi†
Vinod Venkataraman† Kaushik Veeraraghavan† Yee Jiun Song†

†Facebook *Brown University

Abstract

This paper presents Canopy, Facebook's end-to-end performance tracing infrastructure. Canopy records causally related performance data across the end-to-end execution path of requests, including from browsers, mobile applications, and backend services. Canopy processes traces in near real-time, derives user-specified features, and outputs to performance datasets that aggregate across billions of requests. Using Canopy, Facebook engineers can query and analyze performance data in real-time. Canopy addresses three challenges we have encountered in scaling performance analysis: supporting the range of execution and performance models used by different components of the Facebook stack; supporting interactive ad-hoc analysis of performance data; and enabling deep customization by users, from sampling traces to extracting and visualizing features. Canopy currently records and processes over 1 billion traces per day. We discuss how Canopy has evolved to apply to a wide range of scenarios, and present case studies of its use in solving various performance challenges.

1 Introduction

End-users of Facebook services expect a consistently performant experience. However, understanding, analyzing, and troubleshooting performance can be difficult. End-users can perform a wide range of actions and access Facebook through heterogeneous clients, including web browsers and mobile apps, which offer varying levels of information and control over the application. Each action, such as loading a page on Facebook.com, entails complex executions spanning clients, networks, and distributed back-end services.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132749

Dynamic factors also influence performance, such as continuous deployment of new code, changing configurations, user-specific experiments, and datacenters with distinct characteristics. The metrics and events relevant to performance are diverse and continually changing; different endpoints may have different metric definitions, while some endpoints encompass end-to-end flows that cross multiple layers of the system stack. Engineers need ready access to performance data, ideally tailored to the problem they're trying to solve and the questions they typically ask.

This paper describes the design and implementation of Canopy, Facebook's end-to-end performance tracing infrastructure. Canopy is a pervasive tracing infrastructure that records and processes *performance traces*, combining performance data from the end-to-end execution path with structured and causally-related execution traces, to enable customized performance analysis. Canopy extends prior tracing systems such as X-Trace [24] and Dapper [54]; at its core, Canopy constructs traces by propagating identifiers through the system to correlate information across components. However, Canopy addresses three broader challenges we have faced at Facebook in using tracing to solve performance problems.

First, end-to-end performance data is heterogeneous, with multiple execution models and wide variations in the granularity and quality of data available to be profiled. Consuming instrumented data directly is cumbersome and infeasible at scale because engineers and tools must understand the mappings from low-level event data to higher-level semantic structures across all components of Facebook's stack. However, designating a higher-level model in instrumentation leads to issues because instrumentation is pervasive, includes legacy components, must anticipate future components, and is difficult to change even when integrated into common infrastructure.

Second, there is a mismatch in granularity between the high-level, aggregated, exploratory analysis that operators perform in practice, and the rich, fine-scale data in a single trace. Evaluating interactive queries over raw traces is computationally infeasible, because Facebook captures over one billion traces per day, with individual traces comprising thousands of events. Nonetheless, rapidly solving performance problems requires

tools that can efficiently slice, group, filter, aggregate, and summarize traces based on arbitrary features.

Finally, end-to-end performance over the whole stack means many engineers share the same tracing infrastructure. Only a small fraction of data in each trace may be relevant to each engineer, pertaining to specific features or products under investigation. However, by design, traces contain *all* of the data necessary for *any* engineer to identify issues. This presents an information overload, and motivates the need for both (1) generalized interfaces as an entry-point for casual usage and high-level scenarios, and (2) deep customization of everything from the traces to sample, to the features to extract, to the ways to navigate the extracted data.

Canopy addresses these challenges with a complete pipeline for extracting performance data from system-generated traces across the stack, including browsers, mobile applications, and backend services. Canopy emphasizes customization at each step of the pipeline, and provides a novel separation of concerns between components to allow for their individual evolution. At development time, Facebook engineers can instrument their systems using a range of APIs tailored to different execution models. At runtime, Canopy maps the generated performance data to a flexible underlying event-based representation. Canopy's backend pipeline receives events in near-realtime; reconstructs a high-level trace model that is more convenient for analyzing and querying; extracts user-specified *features* from traces; and outputs the results to datasets for customized aggregate visualizations.

Canopy has been deployed in production at Facebook for the past 2 years, where it generates and processes 1.3 billion traces per day spanning end-user devices, web servers, and backend services, and backs 129 performance datasets ranging from high-level end-to-end metrics to specific customized use cases. This paper makes the following contributions:

- A decoupled design for tracing that separates instrumentation from the trace model, enabling independent evolution and the composition of different execution models.
- A complete pipeline to transform traces into a custom set of extracted features to enable rapid analysis.
- A set of customizable tracing components to provide multiple views of the same data for different use cases.

The rest of this paper proceeds as follows. We discuss previous experiences with tracing and motivate Canopy in §2. In §3-4 we describe Canopy's design and implementation. We present case studies and evaluation in §5. Finally, we discuss our experiences, related work, and future challenges in §6-8.

2 Motivation

For several years prior to Canopy, a number of tracing systems were developed at Facebook to address various single- and cross-system performance scenarios. Each system was specialized for a specific use case and difficult to extend to other domains. For example, backend services had RPC call-tree instrumentation that was difficult to adapt to other execution models like event loops. Browser page-load tracing focused on joining data from the client and a single server, and had difficulty scaling to cover other backend systems. Mobile applications had standalone OS-provided tracing that lacked the ability to look at aggregate production data. In each of these cases, analysis over traces was confined to fixed workflows, with limited customization for per-domain features and slow iteration time for new features or analyses.

This siloed approach to tracing led to several problems. It was difficult to get cross-system insights when the tools themselves didn't cross all systems, and it meant that engineers needed to understand multiple tracing tools and when to use and switch between them. Each tracing system evolved to meet specific needs and respond to gaps in other tracing systems, but no single system could handle the broad spectrum of issues that arise in practice. The type of analysis supported by each system was also fairly rigid and changes took days to deploy.

Based on these experiences we designed Canopy, a flexible multi-domain tracing system for consuming and analyzing trace data in aggregate. Canopy addressed a growing need for a single tracing system, applicable to many domains, that engineers could customize to quickly target relevant information.

2.1 Canopy in Action

We begin by motivating Canopy with an example of how engineers typically investigate regressions in our web stack. When a performance issue arises in one component, its symptoms might manifest in a different part of the stack; diagnosing performance issues thereby requires a global view, yet granular attribution, of performance.

When an end-user loads a page on Facebook.com, it is assembled from numerous *page pieces* that are developed by many different product groups. Facebook's core framework code combines page pieces, and runs them both on web servers and within client browsers. In March 2017, the average time needed to display the initial portion of a particular page regressed by 300ms, or approximately 13%, illustrated in Figure 1a. Canopy calculates this metric by recording end-to-end traces of requests, including execution in both the browser and throughout Facebook backends. Canopy backends receive traces in near-realtime, derive a range of metrics including page load latency from each trace, and pipe them to various long-lived datasets of performance metrics.

To investigate the regression, engineers typically start with a breakdown of page load time across the constituent components. Figure 1b shows how server execution time and network time were relatively unchanged, whereas browser execution time and resource-fetching time (CSS, JS, etc.) were elevated.

To calculate these metrics, Canopy derives the *critical path* of requests, using comprehensive timing and dependency information from the client and across all server-side processes. Canopy's instrumentation APIs capture this for a range of different execution models (*e.g.* threads, events, continuations, queues, RPCs); internally, Canopy unifies the information using a generic underlying representation for trace data.

The next step in the investigation was to diagnose the change in resource loading time. The useful high-level metrics for resource loading relate to a server-side mechanism called *early flush*. Early flush optimizes the page load time by predicting and preemptively sending batches of resources that might be needed by the client. Canopy measures the success of early flush by capturing information when resources are received on the client, then incorporating them into critical path metrics. Figure 1c plots the prediction accuracy for JavaScript and CSS resources – 80% accuracy means that 80% of the resources sent were actually necessary for the initial page display. The page load regression corresponded to a 5% drop in CSS prediction accuracy and an additional 10kB of CSS before the page could display (1d); this unnecessary CSS took the place of useful resources, forcing clients to wait for the next batch of resources before the page could display (an early flush *miss*).

These metrics pointed to a problem at the page granularity, but do not yet point to what caused the change. Canopy's datasets further break down metrics at the granularity of page pieces, and grouping by page piece (1e) identified the culprit – the UserInput page piece had added an additional 10kB of CSS to the page load critical path. By comparison, other components were unchanged. This pointed at the root cause – the launch of a new feature in the UserInput page piece changed the set of resources that it required. The early flush component was unaware of this change, and grouped resources suboptimally. On the client, the UserInput page piece stalled waiting for resources that weren't initially available, increasing its time on the critical path (1f). Engineers fixed the regression by re-configuring the early flush mechanism.

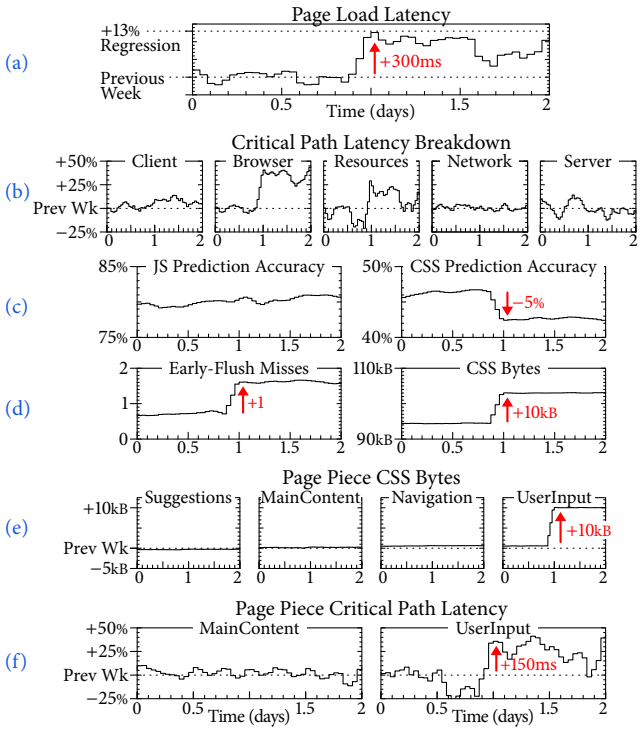


Figure 1: Using Canopy, an engineer quickly identifies UserInput as responsible for a 300ms page load time regression (cf. §2.1).

2.2 Challenges

This example illustrates how a range of different information, related across system components, is necessary to diagnose problems. However, it is difficult to predict the information that will be useful or the steps that engineers will take in their analysis. In practice at Facebook, successful approaches to diagnosing performance problems are usually based on human intuition: engineers develop and investigate hypotheses, drawing on different kinds of data presented in different ways: individual traces, aggregates across many traces, historical trends, filtered data and breakdowns, etc. In the research literature, a recent study of Splunk usage [3] drew similar conclusions that human inference tends to drive analysis, while the use of automated techniques is "*relatively rare*". The goal of Canopy is to allow human-generated hypotheses to be tested and proved or disproved quickly, enabling rapid iteration. To achieve this, we must overcome three broad challenges.

Modeling Trace Data Performance traces incorporate a wide range of information from all components on the execution path of requests, from clients through backend services. Causal relationships between events provide further information about the concurrency and dependencies of execution. In their most general form, these traces are directed, acyclic graphs (DAGs), with nodes representing events in time, and edges representing causality [23, 31]. However, exposing traces at this granularity is inappropriate for two reasons.

First, in order to reconstruct high-level concepts, engineers must understand and interpret trace events and their causal relationships. For example, a segment of processing might delimit itself with start and end events, derive its latency from event timestamps, and describe interesting features using intermediary events. Similarly, causal edges might be classified as network communication if they link events annotated with different hostnames. However, many engineers and teams participate in recording traces, making ad-hoc decisions about the events and information to record. There is wide variation in the granularity and quality of data captured by different software versions, and missing data is common due to best-effort delivery from remote clients. In practice it is infeasible

to consume traces of this granularity, as few, if any, engineers understand all mappings to high-level concepts.

Second, events are an inappropriate abstraction to expose to engineers adding instrumentation to systems. Engineers must understand how to describe their system's execution model – *e.g.* segments of computation, RPC calls – in terms of events, and do so in a coherent manner across all system components. However, researchers and practitioners consistently describe instrumentation as the most time consuming and difficult part of deploying tracing frameworks [22–24, 28, 50, 53].

To bridge these challenges, previous tracing frameworks proposed that a higher-level trace representation should be a first-class primitive [12, 54, 60]. This approach aligns instrumentation abstractions with the system's execution model, making it easier to instrument systems; it also directly encodes meaningful high level concepts in traces, making them easier to consume. For example, Dapper models traces as trees of *spans* – segments of computation – which naturally describe executions in Google's RPC-based infrastructure and can be intuitively visualized as a call stack. [54]

However, promoting a first-class trace model led to several problems in the open-source community. Frameworks lost generality; for example, Zipkin [60] users struggled to instrument execution models not easily described as span trees, such as queues [65], asynchronous executions [41, 42, 44], and multiple-parent causality [9, 10, 43, 45]. Changing trace models entailed exhaustively revisiting system-level instrumentation [4, 8, 11, 13, 14], or elaborate workarounds such as inferring new attributes [22, 35, 46]. Mismatched models affected compatibility between mixed system versions; *e.g.* Accumulo and Hadoop were impacted by the "*continued lack of concern in the HTrace project around tracing during upgrades*" [4, 5, 7]. Some tracing frameworks suffered "*severe signal loss*" [40] because a simpler trace model discarded useful causal relationships.

We experienced similar challenges at Facebook. Instrumenting core frameworks is useful for making tracing widely available and reducing the need for every system to maintain its own instrumentation. However, there are a wide variety of custom system designs, third party code, and other issues that limit use of those core frameworks. Furthermore, instrumentation is typically compiled into services as they are released, so even when tracing is integrated into core frameworks, a single trace may cross through multiple services each with their own version of instrumentation. Thus, instrumentation must be flexible and adaptable in order to inter-operate correctly both within the trace, and across system components.

Analyzing Data Engineers need to be able to view, filter, and aggregate metrics based on arbitrary features across all traces, *e.g.* to view changes in specific counters, track durations of well-defined subphases, compare metrics before and after a regression; etc. However, traces are an inappropriate level of granularity for ad-hoc analysis, even when the they are abstracted to a higher level model as described in the previous challenge. This is for two reasons.

First, traces are very rich, with an individual trace comprising potentially tens of thousands of performance events. Queries over traces, including historical data, can apply to an extremely large volume of traces – Canopy currently records 1.3 billion traces per day. Simultaneously we want to support engineers analyzing traces at interactive time scales. It is computationally infeasible to directly query this volume of traces in real-time in response to some user query.

Second, it is cognitively demanding to expect users to write queries at the granularity of traces. It should not be necessary to know how to calculate high-level metrics such as page load time, critical path breakdown, or CSS byte attributions (cf. §2.1); in practice it is non-trivial to derive many such features from traces. However, unless we provide further abstractions, users will have to consume trace data directly, which entails complicated queries to extract simple high-level features.

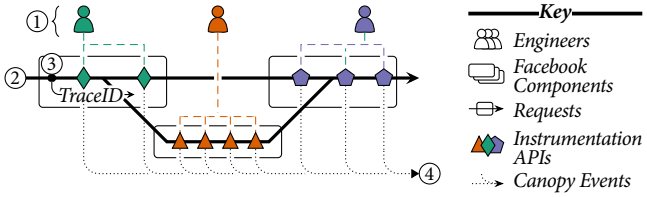
Supporting Multiple Use Cases Facebook traces have evolved over time as operators find new data they are interested in, and new subsystems integrate with our tracing systems. Traces are useful to different people in different ways, and we continually encounter new use cases for tracing, new metrics and labels of interest, new aggregations and breakdowns, etc. The volume of data in a trace can overwhelm users and make it difficult to distill the parts relevant to particular use cases.

This presents a delicate trade-off; on one hand, we want simplicity and ease-of-use for the broadest possible audience, implying a shared, curated, comprehensive entry point for users. On the other hand, users should be able to view traces through a lens appropriate for their particular task or component, *i.e.* avoiding features and metrics that pertain to other components. We must also support end-to-end analysis for problems that span multiple components, and support drilling-down to the lowest-level details of individual traces.

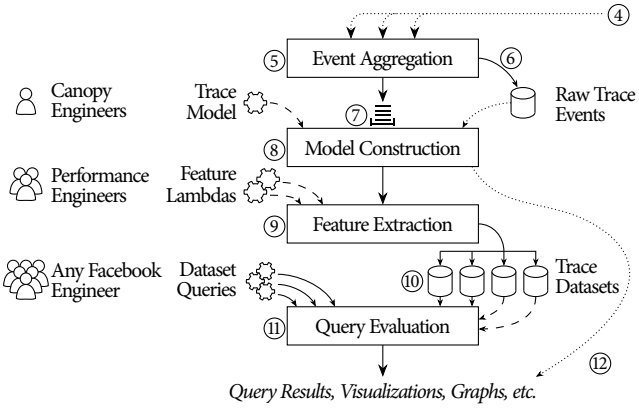
These challenges imply that users should have control over the data presented to them, especially in terms of exploratory analysis. This includes piping data to custom backends; viewing data through custom visualizations; extracting custom features and metrics; and ensuring low QPS systems and underrepresented execution types are sufficiently sampled.

3 Design

Canopy addresses these challenges by providing a pipeline for extracting performance data from system-generated traces across the stack. Canopy emphasizes user-customization at each step of the pipeline, and provides a separation of concerns between components to allow for their individual evolution. In this section we describe Canopy's high level design and the flow for processing and analyzing performance traces.



(a) Engineers instrument Facebook components using a range of different Canopy instrumentation APIs (①). At runtime, requests traverse components (②) and propagate a TraceID (③); when requests trigger instrumentation, Canopy generates and emits events (④).



(b) Canopy's tailer aggregates events (⑤), constructs model-based traces (⑧), evaluates user-supplied feature extraction functions (⑨), and pipes output to user-defined datasets (⑩). Users subsequently run queries, view dashboards and explore datasets (⑪,⑫).

Figure 2: Overview of how (a) developers instrument systems to generate events and (b) Canopy processes trace events (cf. §3.1).

3.1 Canopy Overview

Figure 2 illustrates Canopy's architecture. We refer to the numbers in the figure in our description. To begin, Facebook engineers *instrument* system components to record performance information (①). Canopy provides several instrumentation APIs to capture different aspects of performance, *e.g.* counters, logs, causal dependencies, etc.

At runtime, incoming requests to Facebook will traverse the instrumented system components (②). To relate performance events to requests, Canopy assigns each request a unique TraceID and propagates it along the request's end-to-end execution path (③), including across process boundaries and when requests fan out and in. When instrumentation is triggered, Canopy generates *events* capturing performance information and causality with respect to prior events during execution (④). Internally, all instrumentation APIs map down to a common underlying event representation.

Canopy routes events to the *tailer*, its sharded backend processing pipeline. We shard by TraceID, so that events for each trace route to the same tailer instance. Upon receiving them, the tailer aggregates events in memory (⑤) and persists them to storage (⑥). Once the tailer determines all events have been received for a request, they are queued for processing (⑦). Processing begins by mapping events to a *trace model* (⑧), which provides a single high-level representation for performance traces that unifies the different instrumentation models and APIs used by Facebook developers. Next, Canopy evaluates user-supplied *feature lambdas* (⑨) which extract or compute interesting features from each modeled trace. Users bundle their feature lambdas with a dataset configuration that specifies predicates for filtering unwanted traces and directions for where to output the extracted features (⑩); typically, datasets are piped to Scuba [1], an in-memory database designed for performance data.

Finally, Facebook engineers can query datasets directly and view visualizations and dashboards backed by the datasets (⑪). In addition to user-configured datasets, Canopy provides several shared datasets and visualizations containing common high-level features, plus tools for drilling down into the underlying traces if deeper inspection is needed (⑫).

3.2 Instrumentation APIs

Instrumentation broadly comprises three tasks: 1) propagating the TraceID alongside requests as they execute, to associate performance data generated by different components; 2) recording the request structure, *e.g.* where and when it executes, causality between threads and components, and network communication; and 3) capturing useful performance data, *e.g.* logging statements, performance counters, and stack traces.

Each Canopy instrumentation API performs a slice of these tasks depending on what best aligns with the component or programming language in question. Canopy's low-level libraries in several languages allow users to manually log events and pass TraceIDs between threads; however most APIs layer higher-level concepts on top of these. For instance, in most Facebook components, causality tracking is handled automatically as part of a RequestContext interface that is solely responsible for passing around metadata like TraceIDs. Instead of events, some higher-level libraries have constructs for annotating segments of processing, such as try-with-resources statements in Java:

```
try (Block b = Canopy.block("Doing some work")) { ... }
```

Conversely, Facebook's web servers are heavily continuation based, so Canopy does not expose these concepts because it is difficult to manually track causality through asynchronous calls and accurately attribute performance counters to work done. Instead, Canopy's PHP instrumentation library only supports noting points in time and wrapping functions to profile, *e.g.*:

```
Canopy()->inform('Evicting Cache Entry');
Canopy()->measure('Evicting', $evictFunction);
```

```
struct Event {
  1: required string traceID;
  2: required string type;
  3: required string id1;
  4: optional string id2;
  5: optional i64 sequenceNumber;
  6: required i64 timestamp;
  7: optional map<string,string> annotations;
}
```

Figure 3: Thrift [55] declaration of Canopy events.

Internally, Facebook's web runtime will: propagate the TraceID; track causality through continuations; snapshot lightweight performance counters exposed by the server; and generate events that capture the order of, and relationships between, asynchronous function calls. In several languages Canopy provides orthogonal performance APIs, *e.g.* to register user-defined counters which Canopy will regularly record; to enable and disable stack trace sampling; and to capture OS-level information.

There are several reasons decoupling instrumentation APIs is a successful approach. First, for any particular instrumentation task, engineers have access to a narrow API that naturally fits the specific task at hand. This lowers the barrier to entry for engineers to adopt tracing, and reduces instrumentation errors. Second, by restricting engineers to narrow APIs, the recorded data is more robust because APIs hide the underlying representation; this avoids inconsistencies that arise when developers are responsible for manually constructing and reporting trace elements. Third, there is no single perspective on instrumentation, and no first-class system model, so we can introduce APIs to capture new aspects of performance without invalidating existing instrumentation; this is necessary to support the wide spectrum of custom system designs, third party code, and legacy instrumentation. This has also enabled us to integrate third-party tracing libraries with Canopy, such as HTrace [12] and Cassandra tracing [6]

3.3 Trace Events

Canopy does not designate a single execution or performance model to be the only possible trace representation. Instead, Canopy traces have a generic underlying representation based on events. When instrumentation is triggered, it generates and emits events that encode performance information and causal relationships to other events. Together, the events of a trace form a directed, acyclic graph (DAG). Event DAGs make minimal assumptions about execution models, performance data, or causality, and can describe any computation [31].

Figure 3 outlines Canopy's event definition. Canopy attaches the request's TraceID to each event, so that the subsequent processing stage can accumulate events related to each request. Canopy has an implicit and extensible set of event *types* that determine how Canopy's backend will interpret the event. Instrumentation libraries map high-level concepts down to events, and annotate performance information. Annotations are opaque to the event representation, but enable Canopy's backend to reconstruct the high-level concepts.

To record structural information, instrumentation libraries generate multiple related events which can be later recombined to reason about the structures. For example, a segment of computation might comprise a labeled start event (*e.g.*, "`Doing some work`") and an end event referencing the start event. Canopy uses sequence numbers and timestamps to order events within the same process or thread, and random IDs to relate events to shared concepts. For example, to capture inter-process causality, we typically generate and log an EventID on the sender, propagate both the TraceID and EventID to the receiver, then also log the EventID on the receiver.

Canopy does not enforce or restrict a mapping from high-level concepts to events, and different instrumentation APIs or system versions may represent the same concepts in different ways. New instrumentation libraries can reuse existing common event types or define new ones; we currently define 22 event types. This loose coupling is important for compatibility, since a single trace will cross through multiple services that must inter-operate across different instrumentation library versions. See §5.3 for a case study.

3.4 Modeled Traces

Canopy users do not consume events directly; instead Canopy constructs a *modeled trace* from events. Modeled traces are a higher-level representation of performance data that hide inconsistencies in the low-level events, which may arise due to different component and instrumentation versions.

In its current iteration, Canopy's model describes requests in terms of execution units, blocks, points, and edges. Execution units are high level computational tasks approximately equivalent to a thread of execution; blocks are segments of computation within an execution unit; points are instantaneous occurrences of events within a block; and edges are non-obvious causal relationships between points.

Specific logic for constructing model elements varies by event type. For example, segments of computation with explicit begin and end events naturally map to a block element; intermediary events in the same thread typically annotate the block or create points within the block, depending on their type. On the other hand, an instrumentation API for queues might instead generate events for the enqueue, dequeue and complete operations (similarly for locks, request, acquire, and release); mapping these events to the trace model entails creating several blocks in potentially multiple execution units, and describing their dependencies explicitly using edges.

Edges represent causality, both between processes and machines, as well as between blocks and execution units within the same process. Edges are especially useful for patterns such as streaming between long-running execution units, bidirectional communication; and indirect or application-level block dependencies. For brevity we do not record some causal edges

within an execution unit; due to their nested structure, execution units, blocks, and points have implicit causality that is sufficiently captured by timestamps and sequence numbers.

All trace elements can be annotated with performance data, such as messages, labels, stack traces, sets of values, and counters with an associated unit. Performance data is decoded from event annotations, with type information in the annotation key. Counters have no strict rules governing their usage; in practice they either represent absolute values at a point, or deltas over the duration of a block or edge. On edges, annotations let us convey more information about relationships beyond just causal ordering, *e.g.* timing and queuing information is useful to accurately understand critical path dependencies, a limitation of prior systems [35, 46].

All Canopy traces are backed by the same underlying trace model. This may seem counter-intuitive given that user customization is a goal of Canopy. However, this approach provides a necessary bridge across all system components and performance data. This allows analysis *across* system components; without it we risk siloed and incompatible performance models, and a loss of generality. So far, we have successfully incorporated a wide range of systems and performance data. By decoupling the trace model from instrumentation, we can also update the model as necessary to incorporate new structures, causal relationships, and types of performance data.

3.5 Trace Datasets

Trace-derived datasets are Canopy's high-level output. They comprise information collected across many traces, and are the main access point for Facebook engineers doing performance analysis. Each row of a dataset corresponds to some element in a trace – *e.g.* one high level dataset has a row per trace and aggregates statistics such as overall latency, country of origin, browser type, etc; conversely, a dataset for deeper analysis of web requests has one row for each page piece generated by each request, *i.e.* it is keyed by (TraceID, PagePieceID). The case study in §2.1 used both of these datasets.

Features Each column of a dataset is a feature derived from a trace. A feature might simply be a label taken from a trace element. It can also be a straightforward aggregation, such as the count of block occurrences or the change in a counter value over a particular range. More elaborate features may consider structural relationships, or compute statistics along the critical path between two points in the trace. Features may be any database column type; in practice, we make heavy use of numerics, strings, stacktraces, sets, and lists.

Extraction Canopy transforms traces to features by applying feature extraction functions:

```
f: Trace → Collection<Row<Feature>>
```

Functions are stateless and operate over a single trace at a time. A row corresponds to an element in the trace and comprises features computed over that element or related elements.

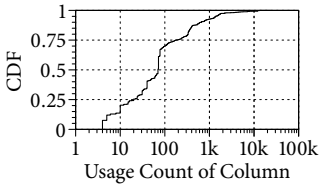


Figure 4: Frequency of columns occurring in dataset queries, for 2,852 columns in 45 datasets, from 6 months of queries to Canopy's main querying UI.

The main requirements for feature extraction are to: 1) enable engineers to iterate and deploy changes quickly; 2) be extensible to support new and unanticipated types of analysis; and 3) encourage modular and composable extraction functions, since custom datasets are often assembled as composites of other, higher level datasets. To this end, Canopy provides several ways to write feature extraction functions (cf. §4.4). The choice of features to extract, and what elements to extract them from, is done on a per-dataset or per-query level; we do not make this decision globally. Features can be extracted as traces arrive or retroactively, and new feature extraction functions can be quickly deployed to production.

Analysis Canopy outputs features to subsequent processing pipelines, including both online in-memory table stores and persistent storage for offline batch analysis. These databases can quickly query large datasets, which enables the kind of interactive, ad-hoc analysis we presented in §2.1.

Although pre-computing features can leave a large quantity of data from each trace unused, in practice we find it sufficient for several reasons. First, the source events and modeled traces remain stored in our storage layer and can be retrieved for either individual viewing or batch ad-hoc analysis. When engineers form hypotheses about possible new features, they can revisit existing traces with the new extraction functions. Second, features are causally related, so dataset queries can combine and break down data in myriad ways; that is, we can slice data by different dimensions without having to infer or compute causality between distributions. In practice most problems are solved by exploring different combinations of a small set of features, and combining common features such as latency with less common features (cf. §2.1 and §5.4). To illustrate, Figure 4 plots the usage frequency for 2,852 columns from 45 datasets, aggregated over 6 months of queries to Canopy's main dataset-querying UI. The most popular column is page load latency from the main Canopy dataset, used by 23,224 queries. However, most columns contributed to fewer than 100 queries, with a median of 70.

4 Implementation

In this section we present details of Canopy's implementation.

4.1 Canopy Client Library

Canopy's instrumentation APIs internally map down to a core client API that provides two main functions: starting a trace, and logging an event. Figure 5 illustrates this core API, and we refer to the numbers in the figure in our descriptions.

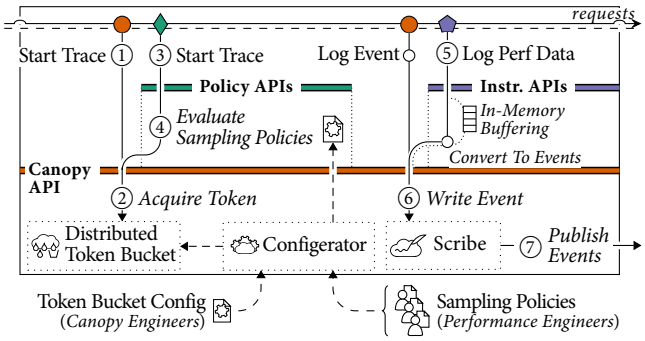


Figure 5: Components (*e.g.* web and backend services) invoke Canopy client APIs to start traces (①,③) and log data (⑤) (cf. §4.1).

Initiating Traces Due to the number of requests and the potentially rich set of captured detail, tracing all Facebook requests is infeasible. Instead Canopy only initiates traces for a subset of requests[1]. A call to start a trace (①) must first acquire a token from a distributed token bucket (②). The token bucket rate-limits traces both globally and per-tenant. Upon acquiring a token, the call generates and returns a TraceID, otherwise tracing is disabled. Rate limiting is necessary to limit undesirably high volumes or bursts of trace data that can occur due to, *e.g.* too-aggressive instrumentation. By default, Canopy allocates tenants 5 traces per second, a relatively low rate. Changing this limit is done by Canopy engineers.

Sampling Policies From the user's perspective, rate limiting is an insufficient granularity for sampling traces, for several reasons: low QPS API calls will be underrepresented; users cannot target requests with specific properties; different tenants cannot initiate traces from the same entry point; and it takes a long time to change the rate of generated traces, since it involves interacting with Canopy engineers.

To address this, we introduce a higher-level *sampling policy* abstraction for users to specify how to sample traces. Each sampling policy specifies: 1) an owner, *e.g.* a user, team, etc.; 2) a sampling strategy, either a probability or a rate; 3) restraints that the request must satisfy, *e.g.* a specific endpoint, datacenter, region, Gatekeeper [57] feature, etc.; 4) a lifetime for the policy; and 5) the level of detail to record – the coarsest level collects just a few high level events; finer levels capture internal service details. These options enable users to express policies targeting a range of different scenarios. For example, rate-based policies and restraints enable users to target low QPS systems and infrequent events. Canopy also allows teams to mark a codepath as having been reached; they can then define policies that target those particular paths.

For web and mobile requests, before acquiring a token, a call to initiate a trace (③) will first consult the user-defined sampling policies to determine whether any user is interested in the request (④). If a policy matches, the call invokes the policy's sampling strategy, and only if it passes do we finally acquire a token from the relevant tenant's bucket (②). This approach gives users fine-grained control over which traces to target, independent of Canopy engineer involvement, while the underlying token bucket provides safety in case of configuration errors. A local Configerator [57] daemon manages Canopy's sampling and token bucket configurations in-memory and synchronizes global updates in real-time, thereby enabling users to rapidly deploy sampling policies.

[1]Other scenarios also trigger tracing, such as scroll events and client interactions.

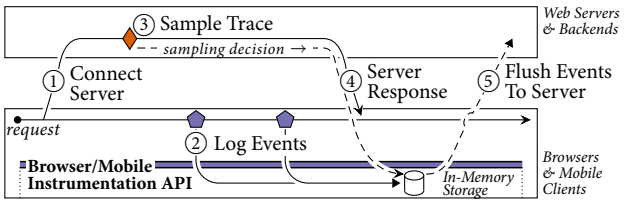


Figure 6: Clients initially cache events in memory. If the server samples the trace, clients flush their events to the server.

In some scenarios, such as for backend services, we only support a subset of sampling policy features. When integrating Canopy in new frameworks or languages, we typically begin by interacting directly with the distributed token bucket, and gradually introduce support for sampling policies as needed.

Generating Events

For sampled traces, calls to Canopy's instrumentation APIs (⑤) will internally log events (⑥). Some instrumentation APIs buffer data in memory before flushing events; primarily in systems that won't benefit from partial trace data when crashes occur. Finally, events are flushed to Scribe (⑦), a distributed messaging system for log data [27].

Opportunistic Tracing Canopy typically evaluates sampling decisions at the beginning of a request. However, for a subset of use cases there are restraints that can only be evaluated partway through the request; for example, a web request which begins on the client but has server-side restraints. Figure 6 illustrates how Canopy captures events that precede the sampling decision. After a client makes a request (①) to a server that might sample a trace, the client opportunistically persists in memory any events (②). After the server makes a sampling decision (③), it communicates that decision back to the client in the response (④). The Canopy client then flushes or discards the pending events (⑤). In general, opportunistic tracing is only supported when the number of traced components are small and known; it is infeasible for large fanouts or multiple hops, because turning tracing off requires informing all systems to which we have already propagated the TraceID.

4.2 Canopy Pipeline

Canopy's tailer is a sharded backend process that receives, processes, and persists trace data. The tailer balances two priorities: to ensure all traces get processed, and to do so as quickly

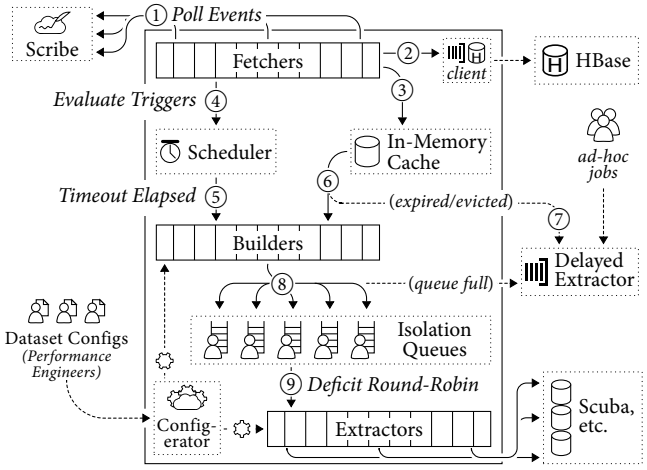


Figure 7: Tailer architecture; see §4.2 for description.

as possible – ideally in near-real time. This is challenging because of exceptional cases such as fluctuations in load, missing and delayed events, and resource contention from expensive datasets. In this section we describe the tailer's constituent components, illustrated in Figure 7.

Routing Events Scribe [27] routes events from instrumented Facebook components to tailers, and shards events by TraceID to ensure all events for a given trace will route to the same tailer. A pool of *fetcher* threads run a loop of polling Scribe (①) and handling received events. Fetchers immediately write each event to HBase for persistent storage (②), then insert into an in-memory cache (③). The cache groups events by TraceID, and expires a trace 15 minutes after its last event was added. If the cache is full, the least recently used trace is evicted by discarding all of its cached events, which account for the bulk of its memory footprint. To prevent a trace re-entering the cache with future events, we maintain an auxiliary cache of TraceIDs seen in the past 5 hours.

Handling Events Next, fetchers must determine when the trace an event belongs to is ready to be processed, *i.e.* all the trace's events have been received. However, we have no guarantee that events will be received in-order, or even at all — events may be dropped, components may not be instrumented, and clients may fail to send data. End-user clients with intermittent or slow network connectivity are particularly problematic: there is a long tail of clients that both take a long time to successfully transmit their events *and* have outlier performance that is important to track.

Our approach targets the common case that trace events arrive unhindered and temporally proximate. We pattern match incoming events and initiate trace processing after sensible timeouts. For each event, fetchers evaluate a set of *triggers*, which examine the event for distinguishing markers or features that can categorize the request (④). If a trigger matches, and the trace-trigger pair is not already scheduled, the fetcher schedules the trace to be processed after a trigger-specified *timeout*. Fetchers evaluate and schedule triggers regardless of whether the event was successfully cached. Example triggers include processing traces 10 minutes after their first event is received, 10 minutes after the first client browser event, and one day after identifying a trace from Facebook's automated build system (due to their long duration). In most cases Canopy receives all of a trace's events before a timeout elapses. Occasionally, Canopy will process a trace only to receive more events later. In this case, triggers are evaluated again, and it is processed twice. However, in practice this occurs rarely; less than 0.03% of traces are processed more than once, with the vast majority from traces that include user clients.

Load Shedding A pool of *builder* threads initiate processing for traces after timeouts elapse (⑤). For each trace-trigger pair, a builder will first retrieve the trace's events from cache (⑥) before processing. If its events expired from the cache, the builder abandons the trace-trigger pair and instead enqueues it to a separate persistent global queue which is read by a separate tier of machines called *delayed extractors* (⑦). Delayed extractors process traces from this queue, performing the same actions as the tailer but retrieving their events from HBase. Delayed extractors also process ad-hoc user jobs, *e.g.* to extract new features from old traces. In practice most traces (99.5%) are successfully processed directly from cache, since timeouts exploit the temporal locality of events.

Model Construction To construct a modeled trace, for each event builders inspect its type (*e.g.* "Edge Begin", "Add Annotation", etc.) and delegates to an event handler for that type. Event handlers mutate trace state by either creating model elements or adding information to existing elements. Since instrumentation may be buggy or incomplete, builders make further passes over the trace to: 1) handle incomplete model elements, *e.g.* blocks or edges missing end events; 2) correct any errors, *e.g.* points that occur outside the bounds of their containing block; 3) infer additional model elements, *e.g.* information from services without data; and 4) align timestamps across distinct hosts using causal edges, establishing a single timeline relative to the start of the trace. Builders annotate model elements with any changes or corrections — we track model errors in several datasets so that Canopy engineers can revisit and fix the model or instrumentation. Finally, builders validate a set of invariants on the trace, *e.g.* ensuring the trace has no cycles (which should be impossible).

Feature Extraction Builders next evaluate user-defined *dataset configs*. A dataset config comprises: 1) an owner; 2) a filtering predicate; 3) an extraction expression; and 4) output database information. Builders apply the filtering predicates to determine valid configs, then enqueue the trace into one or more *isolation queues* based on the config owners (⑧). Isolation queues are short, and if a builder cannot enqueue a trace due

to capacity restrictions it abandons the trace and writes the TraceID to the delayed extractor queue.

A separate pool of *extractor* threads poll isolation queues for traces, using deficit round-robin [52] to fairly share CPU time between queues (⑨). Extractors evaluate extraction expressions and output to the configured databases. Most datasets output to either Scuba [1], an in-memory database for real-time aggregation and queries, or Hive [59] tables, for offline processing as part of a general data warehouse pipeline.

Fair sharing is necessary for two reasons: arbitrary user-supplied extraction expressions are potentially very expensive; and users may misconfigure sampling policies to collect too many traces. We offload surplus processing to delayed extractors to prevent lag from impacting other users.

4.3 Scalability

Routing Events We shard Canopy's backend by TraceID, so for any given trace, all events will route to the same tailer. Consequently, there is no communication between tailers and Canopy's backend trivially scales by provisioning more machines. Routing events to tailers is Canopy's main scalability bottleneck; TraceIDs are random, so events generated by one component can potentially route to any tailer. Canopy currently generates and processes approximately 1.16 GB/s of event data. However, Scribe can scale several orders of magnitude beyond this, and this is a small fraction of the overall capacity of Facebook's Scribe infrastructure

Events and Datasets Canopy has two secondary scalability concerns. As instrumentation in Facebook components has matured, the number of events per trace has steadily grown: ~160 for backend services; >4500 for browser traces; and >7500 for mobile. This impacts the runtime overheads of generating trace data, which are more pronounced for verbose traces (cf. §5.1). To address this, we introduced multiple levels of trace detail that users can specify in their sampling policies, as described in §4.1. The number of users of Canopy and generated datasets has also grown over time, from 35 datasets in November 2015 to 129 datasets in April 2017.

Load Shedding Under normal operation, fetchers consume <5% CPU, builders ~25%, and extractors >55%. Under heavy load, isolation queues back up, and traces above each owner's fair share offload to delayed extractors. On rare occasions builders fail to keep up with the rate of traces, causing lag to accrue between triggers timing out and a builder picking up the trace; consequently, builders encounter more traces expiring from cache. Due to the default expiry of 15 minutes, builder lag does not exceed 5 minutes; beyond that all traces expire, and it is cheap to clear expired traces. On several prior occasions, issues with HBase caused fetcher threads to exhaust client-side buffers and block on writing events to HBase. When this occurs events accumulate in Scribe, since fetchers only receive events from Scribe by polling. Fetchers can lag by up to 3 days due to Scribe's data retention guarantees; however in practice they only lag by between 20 and 40 seconds.

(a) *Find the earliest client-side point*
```
Begin = ExecUnits | Filter(name="Client") | Points | First
```

(b) *Find the client-side "display done" marker.*
```
DisplayDone = ExecUnits | Filter(name="Client") | Points |
              Filter(marker="display_done") | First
```

(c) *Calculate display done latency.*
```
DisplayDone | Timestamp | Subtract(Begin | Timestamp) |
 RecordAs("display_done")
```

(d) *Find all network resource loading on the critical path.*
```
ResourceEdges = CriticalPath(Begin->DisplayDone) | Edges |
                Filter(type="resource")
```

(e) *Calculate the total resource loading time on the critical path.*
```
ResourceEdges | Duration | Sum | RecordAs("res_load")
```

(f) *Calculate the total CSS bytes on the critical path.*
```
ResourceEdges | Filter(resource="css") | Counter(bytes) |
 Sum | RecordAs("css_bytes")
```

Figure 8: Extraction expressions for features used in §2.1.

4.4 Feature Extraction

Canopy provides a domain-specific language (DSL) for describing features as pipelines of functions. Each function specified in the DSL is optimized and compiled into an efficient underlying implementation. Figure 8 illustrates DSL expressions for several features from the case study from §2.1.

Feature extraction is Canopy's most heavily iterated component. We designed Canopy's DSL when many features were straightforward functional extractions of trace data. The DSL initially offered a limited set of transforms with modest expressibility, and we expected to regularly extend the DSL with new functions as necessary. In practice, computed features grew more complex than expected, and the DSL had to incorporate more general-purpose features like data structures. Furthermore, engineers found the DSL insufficiently expressive for speculative or exploratory feature analysis, and were forced to jump to another language to write a user-defined function, then wait for redeployment. To address this, for ad-hoc queries we currently allow for integration with iPython notebooks (§4.5), and we are expanding support for dataset extraction to allow these to be stream processed online.

4.5 Querying and Visualization

Canopy provides multiple entry points to consume trace data, illustrated in Figure 9. Typically, engineers start from an aggregate view of the collected trace data. Figure 9a shows a visualization for comparing function calls between two populations to identify regressions in the Facebook mobile app. Figure 9b is a customized view for browser traces that visualizes time spent in different critical path components; the visualization is interactive and users can drill down into the data, view distributions, and select samples. Engineers also interact with

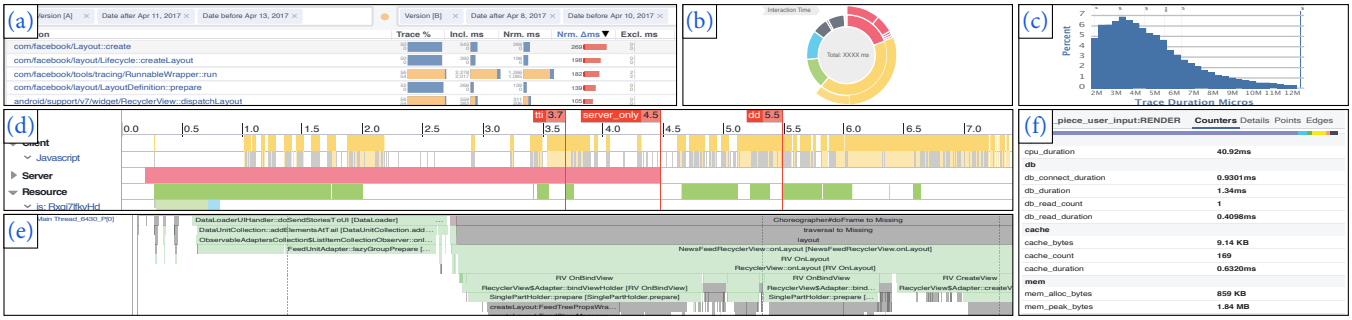


Figure 9: Engineers can use aggregate visualizations (a-c) to explore features. They can continue to drill down to individual traces (d-e) or elements within the trace (f). All visualizations support customizations to focus on relevant data for that view (cf. §4.5)

		Mean	1st	25th	50th	75th	99th
WWW	CPU	6.76%	11.11%	7.31%	6.86%	6.44%	4.44%
	Wallclock	2.28%	7.11%	2.18%	2.14%	2.38%	2.70%
Android	Cold Start	8.57%	5.29%	4.88%	10.78%	13.37%	0.00%
Service	ServiceA Wallclock	8.15%	6.83%	7.03%	7.15%	7.53%	10.44%
	ServiceB Wallclock	0.76%	0.72%	0.79%	0.80%	0.79%	0.38%

Table 1: Latency increase with tracing enabled (Mean and Percentiles).

datasets through visualizations and query interfaces provided by the underlying datastores. For example, Scuba [1] supports a subset of SQL[2] and provides a wide range of visualization and charting options (9c).

Each of these views provides ways to filter and slice features, enabling engineers to quickly explore the data along different dimensions. These also link between each other, so engineers can always jump from one custom view to other relevant views, or to the underlying view for the datastore.

At some point in an investigation, engineers may need to pivot from viewing aggregate data to exploring a single trace. From aggregate views, individual traces matching the filtered characteristics can be found and then explored. These views may also be customized based on the type of trace and nature of the investigation. Web traces have custom markers denoting important phases and execution units grouped by type (9d); mobile traces display sampled profiling data as a flame chart (9e). Engineers can inspect properties of individual model elements (9f), customized to group, aggregate, and display certain properties. Due to Canopy's single underlying trace model, a single trace may be viewed through many different visualizations, each emphasizing different aspects.

To analyze traces and datasets programmatically, Canopy integrates with iPython and we provide several iPython templates to get started. Engineers can load individual traces; write DSL functions or UDFs and process traces in bulk; or load and explore datasets from backend datastores.

[2]Scuba SQL supports grouping and aggregations, but not joins. By default Scuba subsamples data beyond 30 days and gives confidence bounds for aggregation queries. Scuba queries typically complete within a few seconds.

5 Evaluation

Canopy has been deployed in production at Facebook for the past 2 years. In this section we present case studies where Canopy has enabled Facebook engineers to detect and diagnose performance problems. We examine Canopy's overheads and load-shedding properties, and show that in addition to addressing the challenges described in §2.2, Canopy:

- Enables rapid performance analysis across heterogeneous systems with different execution and performance models;
- Supports many users concurrently and enables customization for different use cases;
- Enables independent evolution of the trace model, to adapt to new use cases and execution types;
- Scales to a large number of traces with low overhead.

5.1 Overheads

Table 1 details how the latency distributions change when tracing is enabled, for requests loading Facebook.com, starting the Android app, and two backend services (ServiceA and ServiceB). The median wallclock time for web requests increases by 2.14% with tracing enabled, and by 10.78% for Android cold-start at the finest granularity of tracing. In all cases, wallclock time increases less than CPU time, since some tasks happen off the critical path. For backend services, overheads are typically proportional to request duration: ServiceA requests are short and instrumented in detail, incurring 8.15% overhead; conversely ServiceB requests are longer and do not extend the default instrumentation, so incur < 1%. Overheads are higher in clients than on servers because we often lack system-level methods for capturing data and instead rely on application-level methods. Finally, Canopy's backend components require less than 0.1% of available datacenter resources.

5.2 Load Shedding

In this experiment we evaluate the tailer's ability to isolate tenants, manage load spikes, and prevent tenant interference. We mirror traffic from one production tailer for a period of 2 hours and 40 minutes, and simulate a load spike by gradually inserting additional load for the Mobile tenant (Figure 10a).

Load increases at t=10, t=40, and t=70; we sustain maximum load from t=70-115; and revert to normal at t=115.

Load introduced at t=40 pushes Mobile above its fair share. At t=50, the 10-minute triggers start elapsing for these traces, and Mobile's isolation queue backs up (10b). From t=50-125, the queue is consistently at or near its capacity of 5 traces, and some Mobile traces are scheduled for delayed extraction (10c). However, traces from non-mobile tenants remain unaffected.

Figure 10d plots the processing lag for traces on the delayed extractor, *i.e.* the time between being queued and processed. From t=0-90, the delayed extractor is under-utilized and processes traces immediately. At t=90, traces are queued faster than they can be processed, increasing the processing lag. After load reverts to normal at t=115, the delayed extractor continues processing traces and clears its backlog by t=155.

Figure 10e plots output rows generated for non-Mobile tenants, comparing the experiment to the source production tailer whose traffic we mirror. The figure shows little discrepancy between experiment and production, and demonstrates how Canopy isolates non-Mobile tenants from the load spike.

In practice, load spikes can happen for a variety of reasons. The distributed token bucket assumes a fixed cost per trace initiation, but in practice the number of events generated for one trace can vary significantly. New instrumentation that generates additional events can therefore increase the cost per trace. Changes to dataset configs can also change the cost of processing a trace, *i.e.* by introducing expensive extraction functions. This is usually caught by a canary system for changes, but sometimes the effects aren't observed for hours. For example, an instrumentation bug increased the number of blocks in each browser trace; this increased backend costs because several datasets for browser traces do work proportional to the number of blocks. However, lag only began accumulating the following morning, corresponding to the daily traffic peak.

5.3 Trace Model Evolution

Canopy has enabled us to evolve our trace model as new use cases arise, without having to revisit existing system instrumentation. Figure 11 illustrates several successive refinements we made to our model of RPC communication. Initially we modeled RPC communication using a block-based model (11a) analogous to Dapper's spans [54]. Each block corresponded to a request within one process and was identifiable by a unique block ID (*e.g.* **A**, **B**, etc.). When making an RPC, the sender (**A**) would generate a block ID (**B**) for the child and include the ID within the call. To record the RPC, we used four event types corresponding to client send (CS), server receive (SR), server send (SS), and client receive (CR), and annotated events with the block ID to enable later reconstruction of the RPC structure in the trace. However, we were unable to express more fine-grained dependencies within services, such as time blocked waiting for the RPC response.

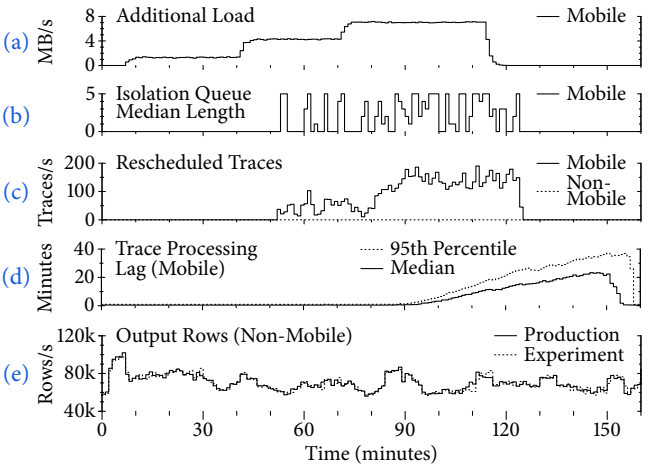


Figure 10: Load-shedding experiment using traffic mirrored from a production tailer: (a) we insert additional Mobile Load; (b) Mobile exceeds its fair share and its queue fills up; (c) excess Mobile traces are sent for delayed extraction; (d) the delayed extractor lags behind processing excess Mobile traces but clears the backlog after load reverts to normal; (e) non-Mobile traffic is unaffected and processed at the same rate as on the production tailer.

Our second model (11b) introduced execution units, distinct from blocks, so that processes could more naturally depict their internal events. We added new instrumentation to measure idle time and incorporated that information into the model. We reused the same event types as our initial instrumentation, and simply relaxed the model to allow client receive events to occur within a different block (**C**) from client send (**A**). However, while this approach enabled accurate measurement of client idle time, it did not account for time between receiving and processing the RPC response, which primarily occurs in two ways: backend servers that have dedicated network IO threads read responses from the network and enqueue them while the main thread is busy completing other work (11c); and asynchronous execution models can demarshall responses in the same threads that run the response handler, but defer the handler to a later time (11d) – this is common in web browsers. To handle these cases, we added a new event type called client queue (CQ) generated when a response is received and queued to be processed, and extended the trace model's RPC structure with metadata to indicate this queuing time. Finally, as more systems at Facebook integrated with Canopy we encountered a wider range of patterns, such as one-way RPCs, RPCs that send multiple responses, and more general streaming and pub-sub communication. This led us to decouple our model's representation of RPCs into sets of edges as described in §3.4, where an edge is a one-way communication between any two blocks (11e). With this, any two blocks can indicate network communication by generating a send event on sender (**A**, **D**) and a receive event on the receiver (**B**, **C**). Although we no longer require the four event types of the initial model (11a), the existing instrumentation is still

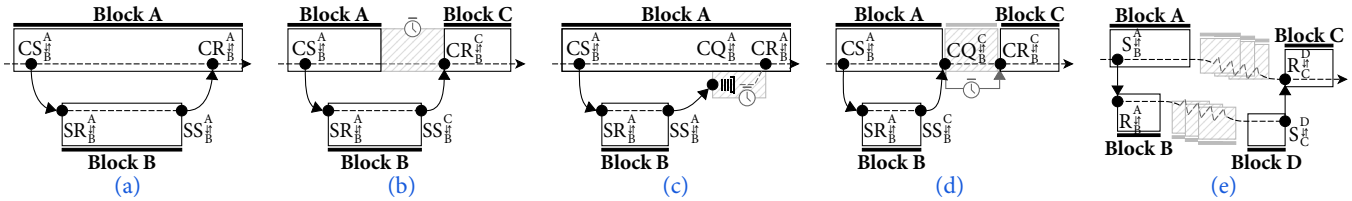
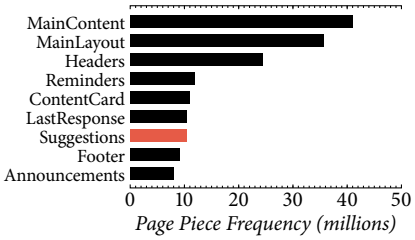
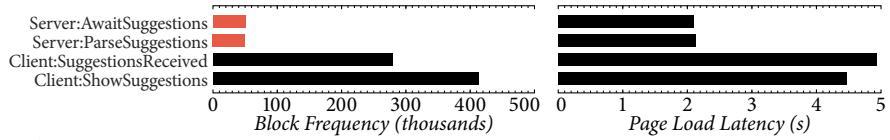


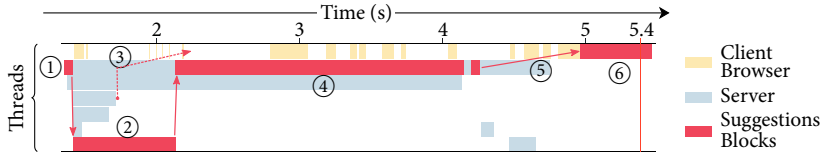
Figure 11: Evolution of Canopy's instrumentation and model for RPCs and network communication (cf. §5.3)



(a) The low-priority 'Suggestions' page piece unexpectedly appears on the execution critical path in 10.4M requests, initiating the investigation.



(b) 'Suggestions' appears on the critical path in two circumstances: on slow clients that receive suggestions before the page is done loading; and on the server during initial page construction.



(c) 'Suggestions' blocks server execution (②) and delays flush (ideal ③, actual ⑤).

Figure 12: An engineer sees something unusual on the critical path and investigates, ultimately finding a server execution issue (cf. §5.4)

compatible with each revision of the model, as they simply translate to a pair of edges in the current model.

Other Integrations Canopy's flexible model lets us incorporate the output of other trace systems and profilers. For example, to integrate HTrace [12] with Canopy, we map HTrace spans to blocks; timeline annotations to points within the block; and parent-child relationships to edges between blocks. Parent-child relationships in HTrace only convey hierarchy and not fine-grained timing information, so the edges map between the block begin points (and similarly for the ends).

5.4 Case Studies

Causal Ordering The Facebook.com page load process is carefully ordered to try and get important page-pieces displayed early, with distinct phases for groups of page-pieces. One of the features extracted for analysis is the set of page-pieces on the critical path for these phases. An engineer looking at this data was surprised to see a low-priority 'Suggestions' page-piece sometimes appearing on the critical path of the highest-priority phase (Figure 12a). To investigate, the engineer checked which specific 'Suggestions' blocks were appearing on the critical path, and cross-referenced with the average request latency when present (Figure 12b). When 'Suggestions' is on the client-side critical path, the high latency provides a sufficient explanation – the server sent all page-piece phases but the client was slow to process them. However, it was unexpected to find 'Suggestions' on the server-side critical path. The engineer inspected an affected trace and discovered the root cause: a bug in the server execution environment, where some asynchronous continuations were treated differently; as a result, the server may wait for these continuations even though the code wasn't. Figure 12c illustrates the execution timeline of an affected request: when the server received the initial request, it executed an asynchronous call to 'Suggestions' (①). However, it unexpectedly blocked waiting for this continuation (②). Normally, the server would have flushed some initial high-priority page pieces (③); instead, execution was blocked waiting for further computation to complete (④) which delayed flushing data to the client (⑤) and eventually displaying the page (⑥). This issue only occurred in a subset of requests; however, Canopy was still able to surface these as an issue. The customized feature extraction exposed this as an issue, and was also able to identify how often this happened and the estimated overall impact.

Regressions in Subpopulations Sometimes shifts happen as a result of internal changes, but sometimes those changes are exogenous. An example of this occurred in December 2016 and January 2017, when internet to the Indian subcontinent was disrupted on multiple occasions, due to a cyclone and multiple submarine cable cuts, respectively. This caused regressions in performance for affected end-users, and was large enough to cause a global regression in pageload latency of ~50ms. Country is an extracted feature, so splitting the data on that axis clearly showed the difference, while the critical path breakdown showed that the largest change for those end-users was in network and resource downloads, as expected. Here, Canopy was able to help engineers rapidly identify the cause and prevent long-term investigation for something ultimately outside of their control.

Identifying Improvements In addition to dataset exploration, Canopy traces can also be used to perform hypothetical analysis. Engineers responsible for the early flush component described in §2.1 improved page load time by considering the cause for the performance breakdowns in the critical path. Their analysis showed that clients were spending an inordinate amount of time on the critical path waiting for additional resources (CSS/JavaScript) before the page load was complete. This occurred because Facebook.com already sent down resources at the start of the page that were expected to be needed later on, but it was possible for page pieces to require additional resources after construction. They predicted that sending down a second wave of resources midway through the page load, particularly those typically used by low priority page-pieces, would lead to a noticeable performance improvement, which was borne out when the feature was implemented.

Mobile Clients Canopy is used across many domains at Facebook to analyze production trace data, including mobile applications. Canopy helps to identify resource bottlenecks and expensive function calls during various phases of important flows like *cold-start* (the time to start the application) and *long-scroll-frames* (other application work preventing timely display updates). Engineers used Canopy's feature extraction of call-stacks to identify a slow-building regression over multiple releases caused by a gradual increase in the number of initialization modules. Ad-hoc queries with Python notebooks have also been used to investigate metric quality; a query found cold-start traces with large gaps with nothing executing, which led engineers to find that their startup metric was including background app starts that they wanted to exclude.

Exploratory Analysis Features determined to be generally useful for a broad set of users are exposed in general datasets for engineers to use for quick investigations. Sometimes there's a slow request, and engineers just want quick help to find the cause. As an example, in February 2017, an engineer noticed a poorly performing endpoint which frequently timed out, and so examined features from a few traces. One of these was database time along the critical path, which led the engineer to find and fix an expensive query without an index, speeding up the DB query by 99% and page load time by 50%.

Custom Analysis For longer or more complex investigations, engineers can define custom views. Canopy allows these teams to quickly set up a customized table, and reuse many of the features from general tables, without forcing all features to exist in a shared global table. As an example, search results were broken up into several page-pieces, where the first may be empty. The latency of this page was defined as the display of the first non-empty page-piece. Engineers defined this metric as a complex extracted feature in a dataset, added existing, modular critical path features to further break it down, and then used this dataset for long-term monitoring of the flow.

6 Experiences

Some observations from our experiences with Canopy:

- Common datasets give casual users an entry point without exposing them to the underlying trace structure. Since not all features are applicable to all trace types, our common datasets vary by domain. For example, we widely use critical path breakdowns for web stack analysis, but not for mobile, where it is more difficult to correctly instrument all blocking (locks, CPU scheduling, framework code, etc.). Instead, mobile datasets center on common resource counters.
- It is an open question whether a DSL can express all possible analyses, or if a general purpose language is required.
- Canopy is applicable to average case, edge case, and tail latency analysis, because at scale, many strange behaviors are frequent enough to identify as statistical outliers. One limitation is investigating a specific instance or request that wasn't sampled, *e.g.* a single user error report of an anomaly.
- Canopy is suitable for tracing any action with a well-defined begin and end. We currently trace user-facing systems (web, mobile), backend systems (services, data analytics), and data provenance scenarios (*e.g.* the end-to-end lifecycle of a user notification). Trace durations vary by multiple orders of magnitude, which we handle with per-domain triggers and timeouts. The main scenarios we've encountered where Canopy's assumptions don't apply are long running dataflow pipelines and scenarios with multiple beginning points.
- Our most effective and scalable instrumentation strategies include instrumenting core frameworks, and capturing data from existing profilers and loggers.
- Common instrumentation errors include conflating TraceID propagation with tracing (and wanting to trace all requests), and propagating TraceIDs incorrectly (causing traces that never end). The latter leads to a large number of events being generated for a single trace; to deal with this, we added triggers for blacklisting erroneous traces.
- We only support opportunistic tracing (cf. §4.1) in a subset of components, because it requires updating instrumentation to propagate sampling metadata in addition to the TraceID. It is an open question whether context propagation can generalize, *i.e.* to propagate arbitrary metadata without requiring instrumentation changes.

7 Related Work

In §2.2 we discuss the challenges Canopy addresses that are shared with many other tracing tools. We complement the discussion on related work here.

Generating Performance Data Existing tools for troubleshooting distributed systems make use of a range of different input data. They include tools that ingest existing per-process and

per-component logs [29, 37, 66]; end-to-end tracing tools that generate causally-related event data [12, 24, 39, 50, 54, 60]; state-monitoring systems that track system-level metrics and performance counters [36]; and aggregation systems to collect and summarize application-level monitoring data [30, 34, 61]. Wang *et al.* provide a comprehensive overview of datacenter troubleshooting tools in [63]. Canopy is designed to provide a common entry point for this breadth of data, since all of these input types are useful for different kinds of analysis.

Like Canopy, many tools record causality between events explicitly by propagating identifiers along the request execution path [12, 24, 39, 50, 54, 60]. Prior systems alleviate the instrumentation burden (cf. §2.2) by propagating context in shared underlying communication and concurrency layers [24, 49, 54, 56]; recent work has also used source and binary rewriting techniques to automatically instrument common execution models [26, 32, 34, 48].

An alternative approach to instrumenting systems is to infer correlation or causality from existing outputs such as logs. Approaches include combining identifiers – *i.e.* call ID, IP address, etc.– present across multiple logging statements [16, 22, 29, 58, 66, 67]; inferring causality using machine learning and statistical techniques [18, 35, 38, 66]; and augmenting models with static source code analysis [66–68]. In practice it is challenging to scale black-box analysis because inferring causal relationships is expensive; for example, computing a Facebook model from 1.3M traces took 2 hours for the Mystery Machine [22]. Canopy scales well beyond this, to 1.3 billion traces per day currently, because of its sharded backend; we require no communication between either traces or shards (cf. §4.3).

Analyzing Data Prior work in troubleshooting distributed systems has presented a variety of automated analysis techniques: semi-automatically honing in on root causes of performance anomalies [64]; identifying statistical anomalies [29]; online statistical outlier monitoring [15]; analyzing critical path dependencies, slack, and speedup [22, 47]; and explaining structural and statistical differences between 'before' and 'after' traces [35, 51]. Manual analysis has covered an even broader range of issues, such as requests whose structure or timing deviate from the norm [2, 17, 20, 21, 46, 51]; identifying slow components and functions [19, 35, 54]; and modelling workloads and resource usage [16, 17, 35, 58]. Use cases for tracing tools deployed in production at 26 companies range from debugging individual anomalous requests, to capturing performance, resource, and latency metrics, to correlating failures and behavioral clustering [33].

Canopy does not obviate any of these techniques, but instead provides the data extraction pipeline that generates the input to many of them. For example, Spectroscope [51] diagnoses regressions between before- and after- sets of traces, by comparing trace structures and performing ANOVA tests. Similarly, Facebook developers have made extensive use of statistical comparison techniques to find correlations between features and compare distributions, *e.g.* between application versions. However, operator-driven exploration is a prerequisite to most automated approaches in order to identify salient features [37]. In a recent analysis of Splunk usage, Alspaugh *et al.* noted that the use of statistical and machine learning inference techniques is *"relatively rare"* and that human inference is the key driver of analyses [3]. Canopy's primary use case is to support ad hoc high-level exploratory analysis, because problems arising in practice are difficult to anticipate and there are a wide range of potential features to explore [34].

Several prior systems support ad hoc exploration similar to the case studies we present. G^2 [25] is a system for writing queries over execution traces at the level of raw traces, so faces the scalability and abstraction challenges of §2.2. For example, evaluating a G^2 query over 130 million events takes 100 seconds; by comparison, interactive Canopy queries derive from trillions of events. Stratified sampling can improve query time by returning approximate results [62]; similarly, for historical data older than 30 days, Scuba subsamples and provides confidence bounds (cf. §4.5). Pivot Tracing [34] is a dynamic instrumentation framework that enables operators to iteratively install new queries during the diagnosis process; however it does not apply to historical analysis, which is an important Facebook use case that necessitates capturing and storing full traces.

8 Conclusion

In this paper we presented Canopy, which emphasizes a decoupled, customizable approach to instrumentation and analysis, allowing each to evolve independently. Canopy addresses several challenges in generating, collecting, and processing trace data, and introduces feature extraction as an important intermediary step for aggregate analysis. Canopy has been used for the past 2 years to solve numerous production performance problems across the Facebook stack, over both end-user client code as well as server-side code.

9 Acknowledgements

We thank our shepherd Rebecca Isaacs and the anonymous SOSP reviewers for their invaluable feedback and suggestions, and Canopy team members and collaborators past and present, including Philippe Ajoux, David Chou, Jim Hunt, Delyan Kratunov, Jérémie Marguerie, Ben Maurer, and Dawid Pustulka.

References

[1] Abraham, L., Allen, J., Barykin, O., Borkar, V., Chopra, B., Gerea, C., Merl, D., Metzler, J., Reiss, D., Subramanian, S., Wiener, J. L., and Zed, O. Scuba: Diving into Data at Facebook. In *39th International Conference on Very Large Data Bases (VLDB '13)*. (§3.1, 4.2, and 4.5).

[2] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance Debugging for Distributed Systems of Black Boxes. In *19th ACM Symposium on Operating Systems Principles (SOSP '03)*. (§7).

[3] Alspaugh, S., Di Chen, B., Lin, J., Ganapathi, A., Hearst, M. A., and Katz, R. H. Analyzing Log Analysis: An Empirical Study of User Log Mining. In *28th USENIX Large Installation System Administration Conference (LISA '14)*. (§2.2 and 7).

[4] Apache. ACCUMULO-3741: Reduce incompatibilities with htrace 3.2.0-incubating. Retrieved January 2017 from https://issues.apache.org/jira/browse/ACCUMULO-3741. (§2.2).

[5] Apache. ACCUMULO-4171: Update to htrace-core4. https://issues.apache.org/jira/browse/ACCUMULO-4171. [Online; accessed January 2017]. (§2.2).

[6] Apache. CASSANDRA-10392: Allow Cassandra to trace to custom tracing implementations. Retrieved January 2017 from https://issues.apache.org/jira/browse/CASSANDRA-10392. (§3.2).

[7] Apache. HBASE-12938: Upgrade HTrace to a recent supportable incubating version. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-12938. (§2.2).

[8] Apache. HBASE-9121: Update HTrace to 2.00 and add new example usage. Retrieved January 2017 from https://issues.apache.org/jira/browse/HBASE-9121. (§2.2).

[9] Apache. HDFS-11622 TraceId hardcoded to 0 in DataStreamer, correlation between multiple spans is lost. Retrieved April 2017 from https://issues.apache.org/jira/browse/HDFS-11622. (§2.2).

[10] Apache. HDFS-7054: Make DFSOutputStream tracing more fine-grained. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-7054. (§2.2).

[11] Apache. HDFS-9080: update htrace version to 4.0.1. Retrieved January 2017 from https://issues.apache.org/jira/browse/HDFS-9080. (§2.2).

[12] Apache. HTrace. Retrieved January 2017 from http://htrace.incubator.apache.org/. (§2.2, 3.2, 5.3, and 7).

[13] Apache. HTRACE-118: support setting the parents of a span after the span is created. Retrieved January 2017 from https://issues.apache.org/jira/browse/HTRACE-118. (§2.2).

[14] Apache. HTRACE-209: Make span ID 128 bit to avoid collisions. Retrieved January 2017 from https://issues.apache.org/jira/browse/HTRACE-209. (§2.2).

[15] Bailis, P., Gan, E., Madden, S., Narayanan, D., Rong, K., and Suri, S. MacroBase: Analytic Monitoring for the Internet of Things. *arXiv preprint arXiv:1603.00567* (2016). (§7).

[16] Barham, P., Donnelly, A., Isaacs, R., and Mortier, R. Using Magpie for Request Extraction and Workload Modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*. (§7).

[17] Barham, P., Isaacs, R., Mortier, R., and Narayanan, D. Magpie: Online Modelling and Performance-Aware Systems. In *9th USENIX Workshop on Hot Topics in Operating Systems (HotOS '03)*. (§7).

[18] Beschastnikh, I., Brun, Y., Ernst, M. D., and Krishnamurthy, A. Inferring Models of Concurrent Systems from Logs of Their Behavior with CSight. In *36th ACM International Conference on Software Engineering (ICSE '14)*. (§7).

[19] Chanda, A., Cox, A. L., and Zwaenepoel, W. Whodunit: Transactional Profiling for Multi-Tier Applications. In *2nd ACM European Conference on Computer Systems (EuroSys '07)*. (§7).

[20] Chen, M. Y., Accardi, A., Kiciman, E., Patterson, D. A., Fox, A., and Brewer, E. A. Path-Based Failure and Evolution Management. In *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. (§7).

[21] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*. (§7).

[22] Chow, M., Meisner, D., Flinn, J., Peek, D., and Wenisch, T. F. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. (§2.2 and 7).

[23] Fonseca, R., Freedman, M. J., and Porter, G. Experiences with Tracing Causality in Networked Services. In *2010 USENIX Internet Network Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN '10)*. (§2.2).

[24] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design and Implementation (NSDI '07)*. (§1, 2.2, and 7).

[25] Guo, Z., Zhou, D., Lin, H., Yang, M., Long, F., Deng, C., Liu, C., and Zhou, L. G^2: A Graph Processing System for Diagnosing Distributed Systems. In *2011 USENIX Annual Technical Conference (ATC)*. (§7).

[26] Jiang, Y., Ravindranath, L., Nath, S., and Govindan, R. WebPerf: Evaluating "What-If" Scenarios for Cloud-hosted Web Applications. In *2016 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. (§7).

[27] Johnson, R. Facebook's Scribe technology now open source. (October 2008). Retrieved August 2017 from https://www.facebook.com/note.php?note_id=32008268919. (§4.1 and 4.2).

[28] Karumuri, S. PinTrace: Distributed Tracing at Pinterest. (August 2016). Retrieved July 2017 from https://www.slideshare.net/mansu/pintrace-advanced-aws-meetup. (§2.2).

[29] Kavulya, S. P., Daniels, S., Joshi, K., Hiltunen, M., Gandhi, R., and Narasimhan, P. Draco: Statistical Diagnosis of Chronic Problems in Large Distributed Systems. In *42nd IEEE/IFIP Conference on Dependable Systems and Networks (DSN '12)*. (§7).

[30] Ko, S. Y., Yalagandula, P., Gupta, I., Talwar, V., Milojicic, D., and Iyer, S. Moara: Flexible and Scalable Group-Based Querying System. In *9th ACM/IFIP/USENIX International Conference on Middleware (Middleware '08)*. (§7).

[31] Leavitt, J. End-to-End Tracing Models: Analysis and Unification. B.Sc. Thesis, Brown University, 2014. (§2.2 and 3.3).

[32] Li, D., Mickens, J., Nath, S., and Ravindranath, L. Domino: Understanding Wide-Area, Asynchronous Event Causality in Web Applications. In *6th ACM Symposium on Cloud Computing (SoCC '15)*. (§7).

[33] Mace, J. End-to-End Tracing: Adoption and Use Cases. Survey, Brown University, 2017. http://cs.brown.edu/people/jcmace/papers/mace2017survey.pdf. (§7).

[34] Mace, J., Roelke, R., and Fonseca, R. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. (§7).

[35] Mann, G., Sandler, M., Krushevskaja, D., Guha, S., and Even-Dar, E. Modeling the Parallel Execution of Black-Box Services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '11)*. (§2.2, 3.4, and 7).

[36] Massie, M. L., Chun, B. N., and Culler, D. E. The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. *Parallel Computing 30*, 7 (2004), 817–840. (§7).

[37] Nagaraj, K., Killian, C. E., and Neville, J. Structured Comparative Analysis of Systems Logs to Diagnose Performance Problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. (§7).

[38] Oliner, A. J., Kulkarni, A. V., and Aiken, A. Using Correlated Surprise to Infer Shared Influence. In *40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*. (§7).

[39] OpenTracing. OpenTracing. Retrieved January 2017 from http://opentracing.io/. (§7).

[40] OpenTracing. Specification 5: Non-RPC Spans and Mapping to Multiple Parents. Retrieved February 2017 from https://github.com/opentracing/specification/issues/5. (§2.2).

[41] OpenZipkin. Zipkin 1189: Representing an asynchronous span in Zipkin. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1189. (§2.2).

[42] OpenZipkin. Zipkin 1243: Support async spans. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1243. (§2.2).

[43] OpenZipkin. Zipkin 1244: Multiple parents aka Linked traces. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/1244. (§2.2).

[44] OpenZipkin. Zipkin 925: How to track async spans? Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/925. (§2.2).

[45] OpenZipkin. Zipkin 939: Zipkin v2 span model. Retrieved January 2017 from https://github.com/openzipkin/zipkin/issues/939. (§2.2).

[46] Ostrowski, K., Mann, G., and Sandler, M. Diagnosing Latency in Multi-Tier Black-Box Services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*. (§2.2, 3.4, and 7).

[47] Ousterhout, K., Rasti, R., Ratnasamy, S., Shenker, S., and Chun, B.-G. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*. (§7).

[48] Ravindranath, L., Padhye, J., Mahajan, R., and Balakrishnan, H. Timecard: Controlling User-Perceived Delays in Server-Based Mobile Applications. In *24th ACM Symposium on Operating Systems Principles (SOSP '13)*. (§7).

[49] Reynolds, P., Killian, C. E., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the Unexpected in Distributed Systems. In *3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI '06)*. (§7).

[50] Sambasivan, R. R., Shafer, I., Mace, J., Sigelman, B. H., Fonseca, R., and Ganger, G. R. Principled Workflow-Centric Tracing of Distributed Systems. In *7th ACM Symposium on Cloud Computing (SOCC '16)*. (§2.2 and 7).

[51] Sambasivan, R. R., Zheng, A. X., De Rosa, M., Krevat, E., Whitman, S., Stroucken, M., Wang, W., Xu, L., and Ganger, G. R. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. (§7).

[52] Shreedhar, M., and Varghese, G. Efficient Fair Queuing Using Deficit Round Robin. In *1995 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM)*. (§4.2).

[53] Sigelman, B. H. Towards Turnkey Distributed Tracing. (June 2016). Retrieved January 2017 from https://medium.com/opentracing/towards-turnkey-distributed-tracing-5f4297d1736. (§2.2).

[54] Sigelman, B. H., Barroso, L. A., Burrows, M., Stephenson, P., Plakal, M., Beaver, D., Jaspan, S., and Shanbhag, C. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report, Google, 2010. (§1, 2.2, 5.3, and 7).

[55] Slee, M., Agarwal, A., and Kwiatkowski, M. Thrift: Scalable Cross-Language Services Implementation. Technical Report, Facebook, 2007. (§3).

[56] Spring. Spring Cloud Sleuth. Retrieved January 2017 from http://cloud.spring.io/spring-cloud-sleuth/. (§7).

[57] Tang, C., Kooburat, T., Venkatachalam, P., Chander, A., Wen, Z., Narayanan, A., Dowell, P., and Karl, R. Holistic Configuration Management at Facebook. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. (§4.1 and 4.1).

[58] Thereska, E., Salmon, B., Strunk, J., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS)*. (§7).

[59] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Antony, S., Liu, H., and Murthy, R. Hive - A Petabyte Scale Data Warehouse Using Hadoop. In *26th IEEE International Conference on Data Engineering (ICDE '10)*. (§4.2).

[60] Twitter. Zipkin. Retrieved July 2017 from http://zipkin.io/. (§2.2 and 7).

[61] Van Renesse, R., Birman, K. P., and Vogels, W. Astrolabe: A Robust and Scalable Technology For Distributed System Monitoring, Management, and Data Mining. *ACM Transactions on Computer Systems 21*, 2 (2003), 164–206. (§7).

[62] Wagner, T., Schkufza, E., and Wieder, U. A Sampling-Based Approach to Accelerating Queries in Log Management Systems. In *Poster presented at: 7th ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH '16)*. (§7).

[63] Wang, C., Kavulya, S. P., Tan, J., Hu, L., Kutare, M., Kasick, M., Schwan, K., Narasimhan, P., and Gandhi, R. Performance Troubleshooting in Data Centers: An Annotated Bibliography. *ACM SIGOPS Operating Systems Review 47*, 3 (2013), 50–62. (§7).

[64] Wang, C., Rayan, I. A., Eisenhauer, G., Schwan, K., Talwar, V., Wolf, M., and Huneycutt, C. VScope: Middleware for Troubleshooting Time-Sensitive Data Center Applications. In *13th ACM/IFIP/USENIX International Middleware Conference (Middleware '12)*. (§7).

[65] Workgroup, D. T. Tracing Workshop. (February 2017). Retrieved February 2017 from https://goo.gl/2WKjhR. (§2.2).

[66] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. I. Detecting Large-Scale System Problems by Mining Console Logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. (§7).

[67] Zhao, X., Rodrigues, K., Luo, Y., Yuan, D., and Stumm, M. Non-Intrusive Performance Profiling for Entire Software Stacks Based on the Flow Reconstruction Principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. (§7).

[68] Zhao, X., Zhang, Y., Lion, D., Faizan, M., Luo, Y., Yuan, D., and Stumm, M. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. (§7).