Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering

Pedro Las-Casas Giorgi Papakerashvili Vaastav Anand Jonathan Mace
Max Planck Institute for Software Systems

Abstract

Distributed tracing is a core component of cloud and datacenter systems, and provides visibility into their end-to-end runtime behavior. To reduce computational and storage overheads, most tracing frameworks do not keep all traces, but sample them *uniformly at random*. While effective at reducing overheads, uniform random sampling inevitably captures redundant, common-case execution traces, which are less useful for analysis and troubleshooting tasks. In this work we present Sifter, a general-purpose framework for *biased* trace sampling. Sifter captures qualitatively more diverse traces, by weighting sampling decisions towards edge-case code paths, infrequent request types, and anomalous events. Sifter does so by using the incoming stream of traces to build an *unbiased* low-dimensional model that approximates the system's common-case behavior. Sifter then biases sampling decisions towards traces that are poorly captured by this model. We have implemented Sifter, integrated it with several open-source tracing systems, and evaluate with traces from a range of open-source and production distributed systems. Our evaluation shows that Sifter effectively biases towards anomalous and outlier executions, is robust to noisy and heterogeneous traces, is efficient and scalable, and adapts to changes in workloads over time.

CCS Concepts

• **Computer systems organization → Cloud computing**; • **Software and its engineering → Software performance**.

ACM Reference Format:
Pedro Las-Casas Giorgi Papakerashvili Vaastav Anand Jonathan Mace. 2019. Sifter: Scalable Sampling for Distributed Traces, without Feature Engineering. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3357223.3362736

1 Introduction

Over the past decade, distributed tracing has emerged as a fundamental component of cloud and datacenter applications. Distributed tracing is widely deployed in all major internet companies [14, 31], and there are several popular open-source variants such as OpenTracing [27], Jaeger [11], and Zipkin [34].

Capturing, processing, and storing traces is computationally expensive – especially for production workloads. To handle this most tracing systems today do not trace everything; instead they sample traces *uniformly at random*, which proportionally reduces the computational overheads. Sampling is done at the granularity of requests – either the end-to-end request trace is kept, or discarded.

While uniform random sampling is effective at reducing overheads, it fails to take into account the *utility* of the traces it samples. Uniform random sampling will naturally capture more traces of common-case executions than it will of edge-case or anomalous executions, since 'normal' executions are far more prevalent in the workload. On the other hand, the most valuable traces for analysis and troubleshooting are traces of edge-case and anomalous executions.

In this paper we investigate *biased* trace sampling, where traces are sampled proportional to how 'interesting' their content is. Biased trace sampling increases the overall utility of traces that are sampled, by reducing the prevalence of redundant common-case traces.

However, biased trace sampling is not straightforward and we face several challenges. First, we cannot rely on manually engineered features; this is brittle and time consuming for developers, and does not automatically adapt to new or unexpected cases. On the other hand, using generic trace features (*e.g.* enumerating all happened-before relationships) is difficult due to extremely high dimensionality. Second, sampling decisions must be robust to noise and heterogeneity in the trace data: qualitatively similar traces often differ in subtle ways, due to transient issues like timing, and the detail and verbosity of traces varies because they combine information across many, independently-instrumented components. Lastly, sampling techniques face strict operational requirements to be useful in practice: sampling decisions must be fast, techniques must scale to a large volume of traces; and they must operate online over a continuous stream of traces. Approaches that are useful in an offline batch setting often don't extend to an online setting in an obvious way, or are impractical due to significant computational costs.

To address these challenges we present Sifter, a general-purpose trace sampler that automatically biases sampling decisions towards outlier and anomalous traces. Sifter operates on a continuous stream of traces, and its computational cost is fixed with respect to both workload volume and sampling rate. The intuition behind Sifter is to approximate the distributed system's common-case behavior, and to sample new traces based on how well represented they are.

To do this, Sifter maintains a low-dimensional probabilistic model of execution paths in the distributed system. By constraining the amount of internal state and minimizing the model's approximation error, Sifter is forced to approximate only the execution paths most prevalent across all traces. Sifter's approximation is inspired by recent advances in the area of neural language modeling [3], which deal with similar challenges of high dimensionality and noise when modeling words, sentences, and documents.

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362736

With every sampling decision, Sifter also updates its internal model to incorporate the new trace. Sifter does this for *every* trace, not just those that are sampled. Consequently, Sifter models common-case behaviors at the frequency they occur, resulting in an *unbiased model*, which enables it to then identify edge-cases when they arise.

In addition to biased sampling, Sifter adapts to changes in workload distributions over time and is robust to new kinds of traces (*e.g.*, in environments where code is continuously deployed). Sifter satisfies the operational requirements of sampling: decisions are fast, and both computational costs and storage costs are independent of both the number of traces previously sampled and the workload volume.

We have implemented Sifter in Python using TensorFlow, and support sampling for X-Trace [8], Jaeger [11], and Zipkin [34] traces. To evaluate Sifter's ability to bias sampling decisions, we use a number of trace datasets from several different distributed systems, including open-source systems HDFS [30], YARN [35], and Spark [40]; the DeathStar social network benchmark [9]; and production traces from a large internet company. In comparison to alternative techniques based on pairwise trace comparison [17], Sifter samples a more qualitatively interesting set of traces; is robust to noisy and heterogeneous traces; makes faster sampling decisions; maintains significantly less internal state; and adapts to variations in workload over time.

In summary, the main contributions of this paper are:

- We demonstrate that distributed traces can be used to build an unbiased probabilistic model of system behavior.
- We identify prediction error (loss) as a strong signal that indicates when a trace is an edge-case or outlier execution.
- We describe the design of Sifter, a framework that simultaneously models a system and makes sampling decisions for traces of that system.
- We have implemented Sifter and evaluate it on trace datasets from open-source and production distributed systems.

2 Motivation

2.1 Distributed Tracing Background

Distributed tracing is a valuable tool for providing end-to-end visibility of distributed systems. Tracing is useful for diagnosing a range of correctness and performance problems in production systems (cf. §9), as traces capture both the events that occur during request execution, as well as the causal ordering and concurrency of these events, and combine this information across all distributed components traversed by the request. Today, distributed tracing tools are deployed at all major internet companies [11, 14, 31]. Open-source distributed tracing tools are widely deployed, with notable examples including OpenTracing [27], Jaeger [11], and Zipkin [34].

A key design goal for distributed tracing tools is to trace production workloads – that is, to produce, collect, and store traces at large request volume, with reasonable overheads. In the extreme, tracing tools can capture an end-to-end execution trace for every request in the entire production workload. Orthogonally, for just a single request, its trace can contain a significant amount of data about that request, capturing all of the events that happened during its execution across all processes and machines; the order, concurrency, and relationships between events; logging messages from developers; timing information; and more.

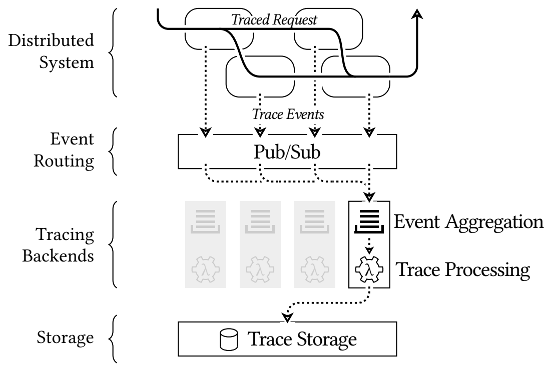


Fig. 1: Tracing pipeline described in §2.2.

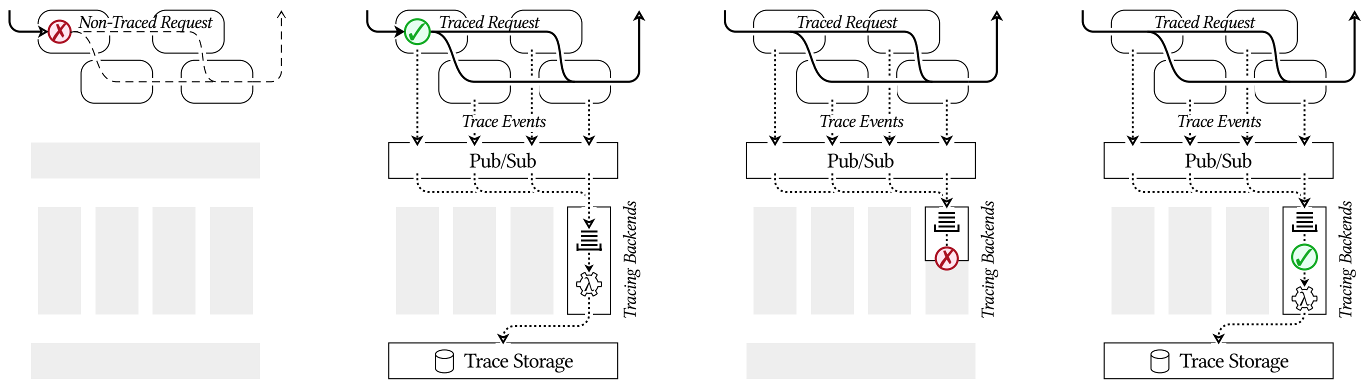
2.2 Tracing Pipeline and Overheads

Distributed tracing tools operate in live production environments, so computational overheads are a key concern. We briefly describe these overheads with respect to Figure 1, which illustrates the typical tracing pipeline. First, traces are generated on the critical path of requests in the distributed system. This incurs runtime overhead to generate trace data, and for propagating trace contexts alongside requests. For a single request, trace data will be spread across all components traversed by the request. Second, a pub-sub system such as Kafka [1] or Scribe [13] will route trace data to tracing backends, incurring network overheads. Third, tracing backends receive trace data and aggregate it in memory for a short period of time before then processing it. Tracing backends are shared-nothing and can easily be sharded using the request ID contained in the trace data. Thus, despite originating from many disparate locations, trace data for any given request will consistently arrive at only a single backend instance. Processing a trace incurs computational costs for constructing its abstract representation and applying feature-extraction functions. There is no need for communication between backend instances – each trace is processed in isolation. Lastly, the trace data and the extracted features are forwarded to long-term storage, where they can later be queried and analyzed.

2.3 Reducing Overheads by Sampling

The prevailing approach to reduce tracing overheads is sampling. Instead of tracing every request, the distributed tracing tools only capture and persist traces for a subset of requests to the system [31]. To ensure the captured data is useful, sampling decisions are coherent per request – a trace is either sampled in its entirety, capturing the full end-to-end execution, or not at all. Sampling is effective at reducing computational overheads; these overheads are only paid if a trace is sampled, so they can be easily reduced by reducing the sampling probability. In practice, sampling rates can be as low as 0.1% [31].

Head-Based Sampling Early tracing systems such as Google's Dapper [31], and later Facebook's Canopy [14], make sampling decisions immediately when a request enters the system. This approach – called *head-based* sampling – avoids the runtime costs to generate trace data. However, since head-based sampling occurs prior to request execution, the sampling decision is uniformly at random, and the resulting data is simply a random subset of requests. Inevitably, the set of sampled traces contains mostly common-case execution paths, with a lot of overlap and redundant information. Conversely,



(a) Head-based, not sampled. **(b) Head-based, sampled.** **(c) Tail-based, not sampled.** **(d) Tail-based, sampled.**

Fig. 2: Trace sampling (✓✗) can occur before (a,b) or after (c,d) request execution. For non-sampled requests (✗), head-based sampling avoids all overheads (a), while tail-based sampling avoids only backend processing and storage overheads (c).

uncommon edge-case executions may be missing entirely, despite containing useful information about anomalous behaviors and infrequently exercised code paths. This impacts the utility of the collected traces in subsequent analyses and investigations.

Tail-Based Sampling Although the runtime overheads of distributed tracing are non-zero, they are not dominant compared to the subsequent trace processing, querying, and storage costs. *Tail-based* sampling is an alternative to head-based sampling: it captures traces for *all* requests, and only decides whether to keep a trace *after* the trace has been generated and sent to tracing backends. Tail-based sampling schemes pay the runtime costs of generating trace data, but in return the tracing data itself can be factored into the sampling decision. Tail-based sampling introduces the possibility of *biased* sampling schemes, where we only persist the most useful traces, and discard traces that carry no useful information (*e.g.*, redundant traces of common-case execution paths). Using tail-based sampling, we can sample qualitatively better traces for the same storage budget; or alternatively, to achieve the same utility of our sampled traces we need sample fewer traces overall.

Figure 2 illustrates the pipeline for recording, aggregating, processing, and storing distributed traces, showing where in the pipeline different sampling decisions can be made.

3 Limitations of Existing Approaches

With tail-based sampling, the contents of a trace can be factored into the decision of whether to keep it. However, it remains an open question how to determine what constitutes a useful trace, and how to make sampling decisions at scale. Preliminary work on tail-based sampling has raised these questions, but there remain obstacles: the need for feature engineering, and a lack of scalability.

3.1 Feature Engineering

The most immediate question for tail-based samplers is *what constitutes an 'interesting' trace*? The approach taken in prior work is manual feature engineering. For each trace, a handful of high-level features are calculated from traces. For example, request latency can be derived from traces as the timestamp delta between the request begin and the response being sent [14]. The sampler then uses these features to make its sampling decision. For example, if our use case is tail latency investigations, we might weight the sampling probability proportional to the measured request latency.

Manual feature engineering is suitable when we already have a good sense of the features that will correlate with traces being interesting. Latency is the clearest example of such a feature, as many anomalous executions or edge-case behaviors result in increased request latency. However, feature engineering is naturally limited to only those features that developers can predict will be interesting a priori, and those features that can be easily written as a feature extraction functions. For instance, at Facebook manually-derived features are a useful starting point for many investigations, but the authors acknowledge that manual features leave a large quantity of data unused from each trace [14].

Ideally, a tail-based sampler should not require developers to explicitly specify features on which to bias the sampling decision. We argue that it would be impossible to predict all possible useful features a priori, and since traces are richly structured, the space of possible features is extremely large. More pragmatically, the burden on developers to identify these features and write feature extraction functions is undesirable. Most importantly, it is undesirable for features to be identified and incorporated only in *response* to a problem occurring, which is commonly how new features are added to trace processing pipelines [14].

To address this, we propose that the sampling decision should be made directly on the underlying trace data, and features should be learned, rather than engineered. However, each trace is a richly annotated, arbitrarily sized, directed acyclic graph, and there are a number of additional subtleties that make it difficult to work with this data. We outline these challenges in more detail in §4.

3.2 Operational Requirements

Trace sampling is inherently *online*: samplers operate in a production setting, and they must make sampling decisions over a continuous stream of incoming traces. Sampling decisions must be fast, and the sampler should scale to large production workloads. Satisfying these constraints while avoiding feature engineering is difficult and has not been solved by prior work.

A common approach to trace sampling, and to related problems like trace clustering, is to adapt general-purpose machine learning and graph mining algorithms. However, while many techniques have a good conceptual fit, most algorithms are designed for an offline setting and have computational costs far in excess of acceptable production limits. For example, some initial work on trace sampling used graph kernels as a pairwise trace distance function, and applied off-the-shelf clustering algorithms [21]. While this approach yielded interesting offline clusters, the authors acknowledged that significant improvements were needed to be reasonable in an online setting.

Adapting offline approaches to an online setting is non-trivial, however. For example, recent work proposed using hierarchical clustering for trace sampling [17]; it was initially developed in an offline setting, and to adapt it to the online setting, it required a new special-purpose sliding window algorithm.

Techniques for trace clustering face similar challenges. For example, to satisfy operational requirements, Magpie uses an approximate string-edit-distance comparison metric, and a simple "nearest centroid" clustering algorithm [2]. Simplicity comes at a loss of utility, with the authors acknowledging that the approach is likely to deteriorate in larger systems with more complex request structures.

In all prior approaches, scalability remains a challenges. In all cases, the internal state grows in proportion to the number of traces seen or previously sampled. This is problematic if the computational cost of a sampling decision is proportional to the internal state, especially if that state is unbounded. For example, most prior approaches to both sampling and clustering compare incoming traces with the set of previously sampled traces [2, 17, 21]. In terms of state, this requires persisting in memory the set of previously sampled traces; and in terms of computational cost, each successive sampling decision is proportional the number of previously sampled traces. Although approaches such as hierarchical clustering can reduce the computational cost to logarithmic in the number of previously sampled traces, they still quickly deteriorate as workload volume increases [17].

To be feasible for a production setting, trace sampling must be fast and scalable. Ideally, the computational cost of each sampling decision should be constant, and the sampler should have a known and fixed memory overhead over time. These costs should be independent of workload volume, the number of traces previously seen, and the number of traces previously sampled.

3.3 Goals

The goal of this work is to design a tail-based sampler that satisfies the operational requirements of production distributed systems. Similar to the representative trace sampling problem [17], the sampler biases sampling decisions towards edge-case traces. Given a target sampling rate of α, we seek a sampling function S_α that maintains an average rate of α and for each trace T calculates a biased sampling probability ρ. In addition, S_α maintains state Θ that is updated with each new trace.

$$S_\alpha(T, \Theta) \rightarrow \rho,\ \Theta'$$

For scalability, S_α must satisfy the following two constraints: that its computational complexity be $O(1)$ with respect to workload volume and previous sampling decisions; and that the memory requirement of Θ is less than some constant B. Lastly, we desire an S_α that does not depend on explicitly engineered trace features.

4 Challenges

The goal of this work is to design a trace sampler that is general-purpose and makes sampling decisions directly on the underlying trace data, rather than on manually engineered features. Simultaneously, the sampler must satisfy the strict operational requirements described in §3.2. Balancing these objectives is challenging for several reasons that we outline in this section. We begin with a brief overview of what kind of data is captured in traces.

4.1 A Primer on Traces

Events Events are the core building block of a trace. Events occur during a request's execution, they are instantaneous in time, and they can be likened to logging statements. A request trace comprises all events generated during the request's execution, from all processes and machines where the request executed. When events are generated, they are sent to the tracing framework's backend; the backend receives all of the events and stitches them into a trace.

Events convey a variety of information. They carry useful diagnostic messages, indicate important control flow points in a program, capture timing information, log concurrent communication, report performance counters, log exceptions, and more. Events often carry auxiliary information such as timestamps, hostnames, process IDs, thread IDs, and so on. Traces can be arbitrarily large, depending on the amount and detail of instrumentation, and length of execution. For example, traces at Facebook contain several thousand events [14]; traces of Hadoop and Spark in our experiments contain hundreds of thousands.

Relationships Events alone tell some of the story, but the most important information in a trace is the causal ordering *between* events, defined by Lamport's happened-before relation [16]. Causal ordering tells us the timing, concurrency, and relationship between events, and is useful for determining which events potentially affected each other, and which did not. For example, within a process it can delineate a request's path through request queues and thread pools; between processes it ties together sending and receiving of packets and messages. Causal ordering is useful for diagnosing problems that cross address space boundaries, because it lets users work backwards from symptoms to potential root causes, even if those root causes were in a different process, machine, or application tier [23]. Causal ordering is a first-class concept for all distributed tracing frameworks, and distinguishes distributed tracing from other types of monitoring data, such as logs and performance counters, which do not explicitly capture the causal ordering of events.

Traces A trace combines events and their causal ordering into a directed, acyclic graph (DAG); events are vertices, and edges are happened-before relationships. Since traces record occurrences of events over time, edges always point forward in time. Alternative to DAGs, another popular representation for traces is as a tree of spans – this representation was proposed by Google's Dapper [31] and it is used by most open-source frameworks. Although spans are a more common representation, DAGs are more general [19], and there is a straightforward mapping from spans into DAGs.

4.2 Heterogeneous Event Annotations

Distributed tracing is an application-level concern. Events are generated explicitly in application code, and tracing metadata is maintained and propagated directly by applications or application-level frameworks. Consequently, the origin and content of an event is arbitrary, as events are defined by the developers of the software being traced. Many events are simply human-readable text, to aid in manual human-driven analysis and troubleshooting [7, 14, 26, 38].

For automated tasks such as trace sampling, we cannot easily incorporate event annotations provided by developers, as interpreting messages to extract meaning is difficult; this is similar to challenges faced in automated log analysis [42, 43]. However, we observe that events almost always do identify their origin. For example, in our experiments all events have source file and line number annotations, even for different client libraries in different languages. We use this origin information to assign each event a *label* that uniquely identifies the event's origin. Events from the same origin share the same label (*i.e.*, multiple invocations of the same point in code), so there may be multiple events with the same label both within and across traces.

4.3 Incorporating Structure

In some cases event labels may be enough to distinguish edge-case and anomalous traces from common cases. For example, events that are generated in exception handlers will only be present in traces where those exceptions occurred. A simple representation for a trace may then be a vector of label counts, and similar approaches have been taken in prior work [10, 17]. However, this approach naturally fails to account for executions that differ in event ordering, which is something we must account for in our sampler.

Of course, incorporating structural information is not straightforward. Explicitly encoding every path in a trace as a feature is intractable due to a combinatorial explosion. Traces are arbitrary in size, both in the number of events and the edges between events, and the number of happened-before relationships grows exponentially in the number of events. On the other hand, using only partial information inevitably loses useful context, *e.g.* using only neighboring events omits useful non-local relationships.

This challenge also appears in work on execution clustering and comparison, where a range of approximations have been explored, including linearized versions of traces [2, 29], probabilistic context-free grammars [6], pairwise graph kernels [21], summary features [14, 17], and in the extreme, graph isomorphism [24, 29]. In all cases, these sacrifice parallelism and concurrency information to deal with the state space explosion, and most do not scale to large datasets. Outside of distributed tracing, a similar challenge occurs in language modeling, an important topic of natural language processing research. Language modeling suffers a "curse of dimensionality" because of the exponentially many ways in which words can be composed to form sentences (many of which may be valid, but never seen twice) [3].

Our intuition to address this challenge is twofold. First, while the total number of paths in a trace is intractably large, the number of 'high signal' paths is likely to be manageable. For example, we often see the same subsequences of events occurring in exactly the same order, *e.g.* because there are no branch points between successive events. While we cannot predict which paths might be useful, we can take advantage of efficient dimensionality reduction techniques to perform this reduction for us. Second, we believe that useful path-based features are more likely to be short than long.

4.4 Approximate Matching

In practice, approaches based on exact matching of paths [6] or graphs [24, 29] are ineffective because they are not robust to *noise* in traces: when two traces are very similar but not identical because of transient, non-deterministic runtime effects. Noise arises from subtle timing differences during execution, which might reorder events or result in different happened-before orderings. Noise also arises when requests generate repeated events, *e.g.* for actions on a periodic timer, or best-effort IO calls. Noise makes exact matching ineffective, because we might have significant overlap between traces, yet never see exactly the same trace twice. For example, our HDFS trace dataset (cf. §7) contains 70,966 traces of 7 different types, yet no two trace DAGs are isomorphic.

In general we seek techniques that are robust to noise and do not rely on exact matching, as minor permutations of common-case executions are not as interesting, even if they have not been seen before. A possible approach to combat noise would be to sanitize traces (*i.e.*, to identify and remove noise), but this is not a realistic solution as it either requires manual effort in postprocessing (aka feature engineering), or imposes unrealistic constraints on the developers doing instrumentation (constraints that are easily violated [22]).

4.5 Cross-Component Tracing

A final challenge is the cross-component nature of tracing. Traces combine events across multiple components, where developers make different and possibly inconsistent decisions about what to trace, and the level of detail of tracing.

Varying trace detail Different system components may incorporate tracing with varying levels of detail. One service may be intricately instrumented and fully capture its internal concurrency, while another might only log entry and exit points. An artifact of detailed instrumentation is that we often find repeated sequences of events that always appear in the same order, *i.e.* due to a lack of intermediate branching. Simple techniques such as node counting unduly bias towards services with more detailed instrumentation; our ideal techniques should be robust to this.

Composition Popular system designs like microservice architectures are compositional in nature. There are often complex dependencies between services and system components; the same service might be invoked in many different places. As a result, two traces might be qualitatively different at a high level, yet internally have some commonality because they invoked the same libraries or downstream services. This commonality makes them less interesting from a trace sampling standpoint, as there some amount of internal redundancy.

Change over time Workload mixes change over time, and a sampler should adapt to these changes. Similarly, code can change over time, introducing new events, more detailed instrumentation, or changing the relationships between events between version updates [14].

5 Design

This section describes the design of Sifter, a tail-based sampler that address the challenges described in §4. Sifter is an online sampler that operates over a continuous stream of traces coming from production distributed systems. Unlike prior work, Sifter does not rely on pairwise trace comparisons, and is efficient enough to be suitable for production environments.

Our starting point is the observation that collectively, the stream of traces in aggregate reflect the overall system and workload behavior, capturing the commonly-traversed paths, corner case executions, and distributions over paths and timings. Based on this observation, our intuition for Sifter is to use the incoming stream of traces to construct and maintain an *unbiased* low-dimensional approximation of the distributed system's common-case behavior. Then to sample traces, it is a matter of comparing each trace to the low-dimensional approximation, and biasing sampling decisions towards traces that are *not* well approximated.

5.1 Dimensionality Reduction

Sifter constructs and maintains a low-dimensional approximation of the distributed system's common-case behavior. The purpose of this low-dimensional approximation is to capture the probability that events occur in a particular order. As outlined in §4 our approximation must be robust to noise, and should condense highly correlated events and trace substructures, so explicitly enumerating paths and probabilities is an unsuitable approach.

Instead, to achieve our goal, we take inspiration from the area of neural language modeling [3], which we find shares many of the challenges described in this work. Of particular interest are techniques for constructing low-dimensional word and document embeddings from large corpuses of example text [18, 25]. These techniques share a similar goal of learning a probabilistic model for words appearing together in sentences, and also deal with challenges of noise and high dimensionality. The model used by Sifter is an adaptation of the paragraph vector embedding [18].

Concretely, Sifter uses a neural network that models the conditional probability of a label occurring given its immediate causal predecessors and successors. To incorporate structure, we decompose a trace into sequences of events following happened-before relationships. For each trace, we first extract all N-length paths, where a path is a sequence of events that follow happened-before relationships. We then map each event in the path to its corresponding label. The model then operates on these N-length label paths. For each path, we predict the middle label $l_{\frac{N}{2}}$ given the surrounding labels $l_0 \ldots l_N$ (excluding $l_{\frac{N}{2}}$) as input.

We implement the model using the same architecture as the distributed memory paragraph vector model, using concatenation [18]. The architecture is a 2-layer neural network illustrated in Figure 3. Inputs are one-hot encodings of $l_0 \ldots l_N$ (excluding $l_{\frac{N}{2}}$), and the output is a one-hot encoding of $l_{\frac{N}{2}}$. The first layer of the network transforms each input label to a P-dimensional vector. The second layer concatenates them then predicts the output label using a softmax classifier.

Key to the design of this architecture is the intermediate layer, where the choice of P is typically small. This forces the model to find approximate representations for its inputs, because it cannot represent the full joint probability distribution of inputs and outputs. Conceptually, this layer is similar to the hidden layer used by autoencoders. When used by language modeling, the weights of the intermediate layer can be used as low-dimensional representations of the corresponding words, and have a range of interesting properties [25]. In our case, P forces the model to approximate – specifically, the model maximizes the average log probability of a label given its surrounding labels, across all walks in all traces. By this definition, the model makes better predictions for paths that are seen more frequently in the input dataset (*i.e.*, in the traces we train on).

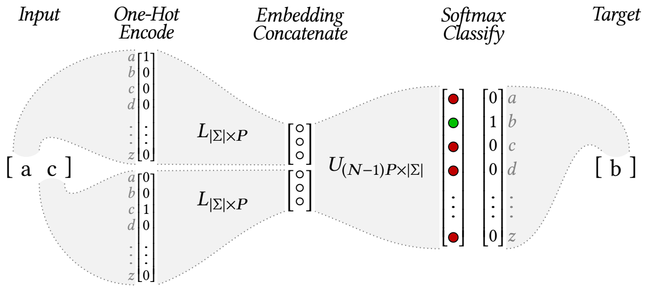


Fig. 3: For a path of labels a→b→c, Sifter predicts b given [a c]. Sifter uses a modified version of the concatenation-based distributed memory paragraph vector model [18]. P **is embedding dimension,** N **is path length, and** Σ **is the alphabet of labels.**

5.2 Sampling Using Prediction Error

To make a sampling decision for a trace, Sifter extracts all N-length paths from the trace, batches them together, and performs a single forward-pass of the model. Since traces are directed and acyclic, small values of N do not result in an intractable number of paths; Table 1 outlines statistics for datasets used in our experiments.

The output of the forward pass is both a prediction of events, and the prediction's *loss*, *i.e.* the error between predicted labels and actual labels. A high loss means that the model was unable to predict the execution paths contained in the trace. Since the model is biased towards the common-case execution paths, a high loss implies that the trace contains edge-case execution paths that are not commonly exercised. This implies that the trace is more interesting, since it captures execution paths that are outside of the norm.

Sifter uses this signal to determine whether to sample the trace – traces with higher loss are more interesting; we could not predict their execution paths well, so we want to sample them with higher probability.

5.3 Calculating Sampling Probability

After calculating the prediction loss for a trace, the next step is to map this loss to a sampling probability. To begin, Sifter is parameterized with a target sampling rate α, *e.g.* to sample 1 out of every 100 traces, we set $\alpha = 0.01$. Translating prediction loss to a sampling probability is not straightforward, because it depends on the prediction loss of previously seen traces; *i.e.* if other traces seen recently had higher prediction loss, then this trace is qualitatively less interesting.

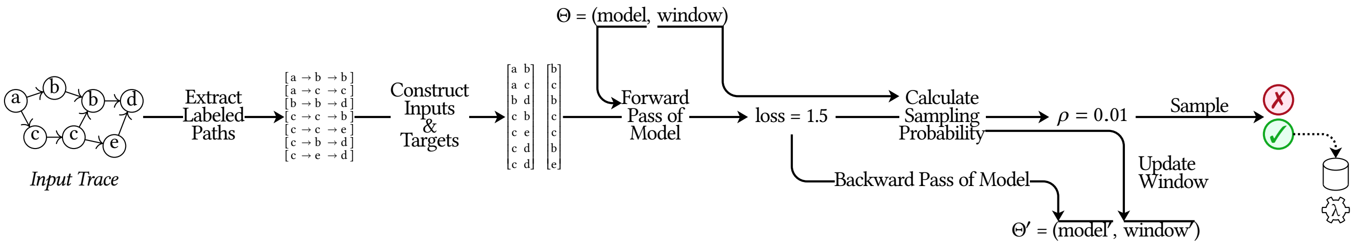


Fig. 4: Sifter's internal pipeline that makes sampling decisions based on an internal probabilistic model of paths in the system. Each sampling decision updates the model.

To address this Sifter tracks the loss of the k most recently seen traces, and weights the sampling probability of the next trace based on the distribution of these prior k losses. For the k most recently seen traces and for the new trace, we calculate weights w_i where:

$$w_\mathrm{i} = \mathrm{loss}_i - \min_{1 \le j \le k+1} \mathrm{loss}_j$$

If all weights are equal (*i.e.* all losses are the same) then we sample the next trace uniformly at random with probability α. Otherwise, we sample the next trace with probability:

$$\rho = \frac{w_{k+1}}{\sum_{j=1}^{k+1} w_j} \times (k+1) \times \alpha$$

The effect of this sampling scheme is that traces with the lowest loss have sampling probability of zero; traces with the highest loss have highest sampling probability; and for traces in between, the sampling probability increases linearly with the error. If zero sampling probability is undesirable, a constant term can be added to ρ, representing a minimum sampling probability.

5.4 Sifter Workflow

Figure 4 outlines Sifter's sampling workflow. Sifter integrates with a typical tracing backend. Typical tracing backends aggregate events in memory for a short period of time, before forwarding them for trace processing and storage [14]. When a trace is ready to be forwarded, it is sent to Sifter to make a sampling decision. Sifter begins by extracting all N-length paths from the trace and does a forward-pass of the model, as described in §5.2. From this forward pass, we calculate a loss value for the trace.

Next, Sifter performs a gradient descent backpropagation pass on the model to update the model weights to incorporate the trace that was just seen. Sifter then calculates a sampling probability from the loss, as described in §5.3. After calculating the sampling probability, Sifter inserts the trace's loss to the window of k most recent losses and drops the oldest value.

Finally, Sifter makes the sampling decision using the calculated sampling probability. If sampled, then the trace continues in the trace processing pipeline. If not sampled, the trace is now dropped.

5.5 Scalability

Part of Sifter's workflow is to perform a backpropagation pass on the model to update the model weights for every trace that is seen. This enables the model to capture common-case execution behavior, even if that behavior is then less likely to be sampled. This approach would fail if we only built the model using traces that are sampled, because it's a *lack* of bias in the model that enables us to determine common cases in the incoming workload. As a consequence, the model weights change over time if the underlying distribution of traces changes (*e.g.*, if the proportion of execution paths change; if new execution types are added; if the vocabulary of labels changes due to code updates, etc.).

Sifter only keeps internal state for the model's weights, and to track the k most recently seen losses. The model comprises exactly $|\Sigma| \times P$ weights for the first layer, and $(N-1) \times P \times |\Sigma|$ weights for the second layer, where P is the size of the low-dimensional intermediate representation, N is the path size we wish to use, and Σ is the label vocabulary. The model does not maintain paragraph vectors, since each trace is seen once and only once. Thus the total size of Θ (cf. §3.3) is constant with respect to the workload volume and number of sampled traces. In terms of computation, performing forward and backwards passes of the models constitute a fixed number of floating point multiplications, which has constant-time complexity.

6 Implementation

We have implemented Sifter in Python using TensorFlow for the internall model. To use Sifter, an instance is first initialized using input parameters N, P, k, and α described in §5. Subsequently, sampling decisions are made through a call to `sample(T)`, which takes as input a trace T and gives as output a boolean sampling decision. The input trace T is simply a directed, acyclic graph where each node has a label l. For convenience, we implemented functions to convert X-Trace [8], Zipkin [34], and Jaeger [11] traces into this format.

Each call to `sample` performs the steps described in §5.4, including backpropagation to update the internal model. Sifter does not require a priori knowledge of the label vocabulary Σ; it extends the model weights as needed when it encounters previously unseen labels. By default, Sifter uses paths of length 5 ($N = 5$), embedding dimension 10 ($P = 10$), a window of 50 ($k = 50$), and a learning rate of 0.01 for the model backpropagation pass.

7 Evaluation

In this section we evaluate Sifter's ability to bias sampling decisions towards edge-case and anomalous traces. We use traces from a variety of distributed systems, and consider a mix of synthetic and real-world workloads. Our evaluation shows that Sifter addresses the challenges described in §4 across all datasets, regardless of trace size, detail, or noise. In particular, we demonstrate that:

- Sifter satisfies production sampling requirements: it makes sampling decisions with low computational overhead, maintains constant-size intermediate state, and can operate online over a continuous stream of traces.

- Sifter is able to clearly distinguish anomalies and infrequent executions from common-case executions. For mixed workloads, Sifter biases sampling decisions towards under-represented execution types.
- Sifter requires no bootstrapping or pretraining – it begins discriminating after only a few examples of traces.
- Sifter reacts to changing workload distributions, both when common executions become less common, and vice versa.
- Sifter's successful sampling decisions are only possible by preserving happened-before relationships.

To evaluate the quality of sampling decisions made by Sifter, we compare Sifter to the hierarchical clustering approach of prior work [17], which uses label counting to perform pairwise trace comparison.

7.1 Datasets

Our evaluation of Sifter focuses on the following three trace datasets:

Hadoop Distributed File System We have instrumented the Hadoop Distributed File System (HDFS) [30] with the X-Trace framework and deployed it on a 9-node cluster. To generate detailed traces, we override HDFS's logging calls to emit X-Trace events. As a result, the generated traces can be large, and contain lots of noise and minor permutations of events. Our dataset comprises traces of the following requests: (1) 1MB file writes to HDFS; (2) reads of 1kB, 10kB, 100kB, 1MB, 10MB, and 100MB files from HDFS; (3) random reads of between 1kB and 100kB. The dataset comprises 70,966 traces in total.

DeathStar Social Network Benchmark We instrumented the DeathStar social network microservices benchmark [9] with X-Trace. We duplicated all logging calls to emit X-Trace events, and logged events at the start and end of all services. We deployed the benchmark on one machine and captured traces of 7 different API types (Register user, Follow user, Unfollow user, Compose post, Write timeline, Read timeline, Read user timeline). Internally, the benchmark comprises 36 microservices; each high-level API call invokes an overlapping subset of the services. In addition to datasets of regular workloads, we also captured traces of two classes of anomaly: one where we manually triggered exceptions in the internal microservices; and one arising accidentally from a configuration error in our deployment, causing docker containers to intermittently restart and services to be temporarily unavailable. The dataset comprises 15,148 traces in total.

Production Traces This dataset comprises 676 traces from a large internet company. The traces capture spans from a microservice architecture, and can be grouped into five different API types.

Table 1 summarizes statistics for traces in these datasets.

Dataset	Traces	Avg. Nodes	Avg. Labels	Avg. Walks
HDFS	70966	1428	38	2547
DeathStar	15148	127	82	155
Production	676	71	56	130

Table 1: Statistics for datasets used in experiments. Nodes, labels, and walks are averages for the traces in each dataset.

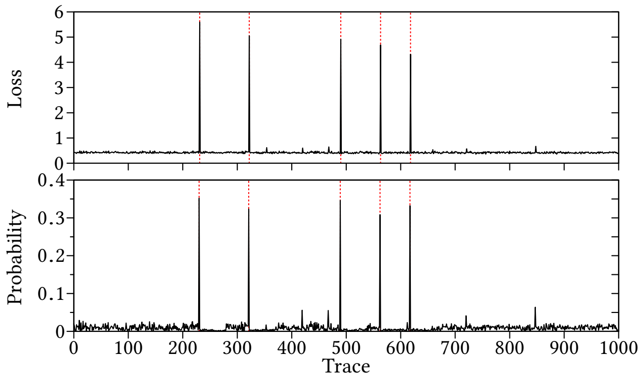


Fig. 5: Loss and sampling probability for a workload of 1000 HDFS `read` traces. We intermittently introduce 5 `write` traces, corresponding to the five spikes with high sampling probability.

7.2 Anomalies and Outliers

Our first set of experiments evaluates Sifter's ability to identify and sample anomalous and outlier executions.

HDFS Reads In our first experiment we replay traces of HDFS `read` API calls, which randomly read between 1kB and 100kB of data from HDFS. We record the loss and sampling probability for a sequence of 1000 traces, and at five separate instances we insert a trace of an HDFS `write` API call. Since Sifter primarily sees examples of `read` calls, `writes` are outliers, representing only 0.5% of the workload. Qualitatively, there is little overlap between `read` and `write` traces; some RPC events are shared, but most of the data pipelining events are mutually exclusive.

Figure 5 plots Sifter's loss and sampling probability, where our target sampling rate is $\alpha = 0.01$. The figure clearly shows five spikes for each instance of a `write`. The average sampling probability for `reads` is 0.0084, with minor fluctuations from trace to trace owing to their internal variability. For the `write` traces, the sampling probability is significantly higher, averaging 0.3325.

Social Network Compose Post We perform a similar experiment using `ComposePost` traces from the DeathStar social network benchmark. We record the loss and sampling probability for a sequence of 1000 traces, and at five separate instances we insert an anomalous trace. The anomalous traces are also calls to the `ComposePost` API, but internally we randomly trigger an exception in one of the internal services. These anomalies represent only 0.5% of the workload, but unlike the previous experiment, there is qualitatively more overlap between normal and anomalous executions, since they are similar up to the point of the exception.

Figure 6 plots Sifter's loss and sampling probability, with a target sampling rate of $\alpha = 0.01$. The five anomalous traces have higher sampling probabilities between 0.12 and 0.28, while the regular traces fluctuate around 0.01. Unlike the HDFS traces, the regular `ComposePost` executions have more internal diversity, resulting in more fluctuations; on the other hand, the anomalies are closer to regular traces. Nonetheless, Sifter clearly identifies the anomalies and samples them with higher probability.

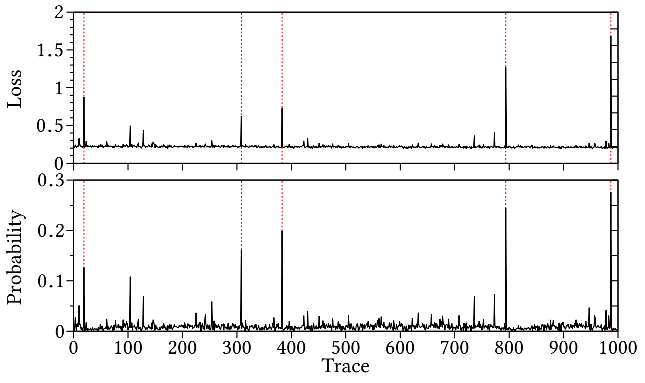


Fig. 6: Loss and sampling probability for `ComposePost` API calls. Five exception traces are introduced, corresponding to five spikes with high sampling probability.

Social Network ReadUserTimeline We perform a final experiment using `ReadUserTimeline` traces from the DeathStar social network benchmark. In this experiment, we compare normal traces to anomalous traces that we inadvertently encountered due to a Docker misconfiguration that intermittently restarted some of the microservices. We replay traces with a 90%-10% distribution of regular to failure traces, corresponding to the error ratio we experienced. We replay 1000 traces and record the loss and sampling probability.

Figure 7 plots Sifter's loss and sampling probability with a target sampling rate $\alpha = 0.01$. We plot the loss and sampling probability for regular traces separately from the anomalous traces (which have 90% fewer datapoints). The figure illustrates that sampling probability for anomalous traces is higher than regular traces; regular traces have average sampling probability 0.008, while failures have 0.029.

7.3 Representative Sampling

Our next set of experiments evaluate's Sifter's ability to generally bias towards underrepresented execution paths and request types for workloads in aggregate. We compare Sifter to random sampling [31] and hierarchical clustering [17]. In practice, hierarchical clustering is not a suitable choice for production systems due to its computational costs. The purpose of this comparison is to assess whether Sifter, with its lower, bounded computational costs, can nonetheless sample traces with similar or better quality.

Production Traces We apply Sifter to a workload of 676 traces captured from a large internet company's production system. These traces comprise 5 different high-level API calls; internally, the requests share commonality in some of the services they call. To introduce more variability, we adjust the representation of each API type to 2%, 8%, 15%, 25%, and 50% respectively.

Figure 8a compares the distribution of traces sampled by Sifter, and compares this to the original workload, as well as random sampling and hierarchical clustering. Sifter increases the representation of low-frequency APIs, while decreasing the representation of common-case APIs. By comparison, hierarchical clustering only slightly increases the underrepresented APIs. To quantify this, Table 2 lists the mean-squared error compared to an equal 20% division between APIs; Sifter has significantly lower error than both random and hierarchical clustering.

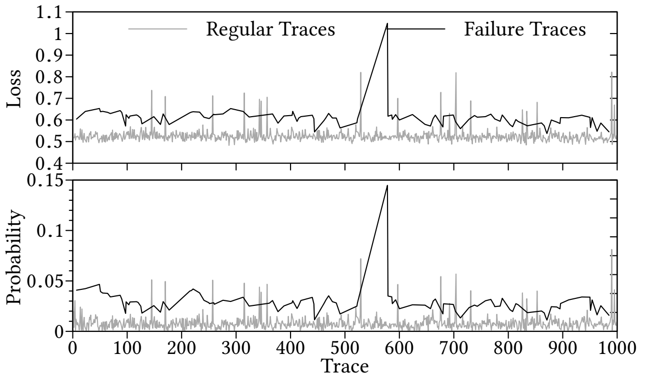


Fig. 7: Loss and probability for `ReadUserTimeline` requests. 10% of the executions failed because of a Docker misconfiguration.

DeathStar Benchmark We perform the same experiment for the 7 DeathStarBench API types totaling 14,000 traces, with the following distribution: 0.5% `Register`; 2% `Unfollow`; 2% `Follow`; 2.75% `WriteHomeTimeline`; 2.75% `ComposePost`; 45% `ReadHomeTimeline`; 45% `ReadUserTimeline`.

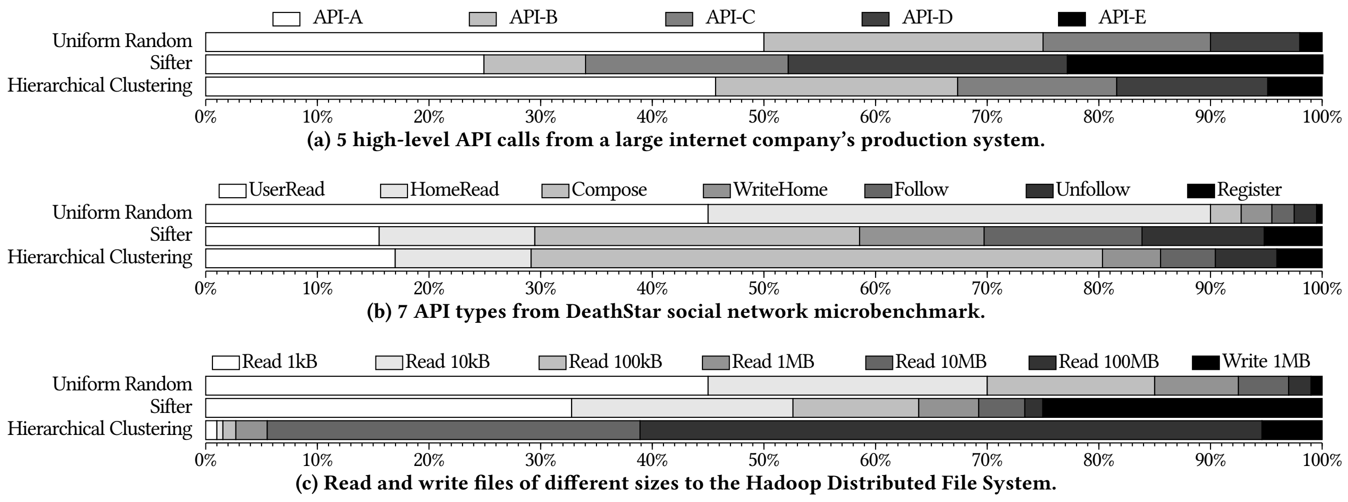
Figure 8b shows the distribution of traces sampled by Sifter. Sifter increases the representation of low-frequency APIs, while decreasing that of common-case calls. `ComposePost` has the highest representation, as it is internally the most complex API with the highest internal execution diversity. Table 2 lists the mean-squared error of the clusters compared to an equal 14.3% division between the APIs.

HDFS We lastly perform a similar experiment for HDFS. We include the following traces: 1MB write API calls; and 1kB, 10kB, 100kB, 1MB, 10MB, and 100MB read calls. The read calls have significant internal similarity, as increasing the read size primarily duplicates events recording data chunk transfers. We include 1% writes, and 45%, 25%, 15%, 7.5%, 4.5%, and 2% of the reads respectively.

Figure 8c shows the distribution of traces sampled by Sifter. Sifter significantly increases the proportion of `write` traces that are sampled, to 25%. Fewer `read` API calls are sampled, but they are sampled approximately in proportion to their representation in the original dataset. This occurs because the traces comprise the same *types* of events (*i.e.*, the same set of labels); just some events occur more frequently in the large `read` calls. Consequently, the large reads are interesting, but not too interesting, as the additional data transfer events are treated as noise. Despite the uneven clustering, this is a desirable result. The figure also shows the distribution of traces sampled using hierarchical clustering [17]; this approach simply counts the co-occurrence of labels in each trace, and inevitably weights towards traces with higher node counts, which are the larger traces.

	Sifter	Hierarchical	Random
Production	35.95	193.12	283.60
DeathStar	46.31	246.26	377.72
HDFS	119.91	404.22	218.28

Table 2: Mean-squared error of sampled clusters (cf. §7.3) for Sifter, random sampling, and hierarchical clustering.



(a) 5 high-level API calls from a large internet company's production system.

(b) 7 API types from DeathStar social network microbenchmark.

(c) Read and write files of different sizes to the Hadoop Distributed File System.

Fig. 8: Proportion of request types sampled by three different schemes, for three different workloads. Note that uniform random sampling reflects the original workload distribution.

7.4 Change over Time

We next evaluate how Sifter adapts to change over time. We focus on two scenarios: when workload distributions change; and boostrapping Sifter from scratch.

Changing distributions This experiment uses 1MB HDFS `write` traces and 1MB HDFS `read` traces. Iterations of the experiment alternate between 90% `write` 10% `read`, followed by 10% `write` 90% `read`. We expect Sifter's sampling probability to adapt to these workload changes; increasing the probability for the 10% class, and decreasing the probability for the 90% class. Figure 9 plots the average sampling probability for 9 iterations, with a target sampling probability of 0.01. The figure illustrates how the sampling probability alternates between low and high for the classes as they alternate between over- and under-represented.

Bootstrapping Sifter is designed to operate in real-time, and does not require pre-training or manual configuration before operation. Initially, Sifter makes essentially random sampling decisions, as its internal model is randomly initialized. As soon as it starts receiving samples, it begins learning its model of system behavior. In this experiment we select two API types from the production traces, and use 5% of one type and 95% of the other.

Figure 10 plots Sifter's loss and sampling probability, starting randomly initialized. Sifter quickly begins to differentiate common cases from edge cases, and the loss for common cases decreases faster than the loss for the edge-case traces. We repeated this experiment 100 times, varying the choice of API type and randomizing the trace order, and saw the same behavior each time.

7.5 Importance of Structure

In this experiment we compare Sifter to a similar scheme that fails to take into account trace structure. The goal of this experiment is to evaluate Sifter's effectiveness when using a linearized version of traces that discards internal concurrency information [2, 29]. We compare to Lifter, a modified version of Sifter in which we collapse a trace's events into a single timestamp-ordered sequence. We then apply Sifter to this sequential data, where paths are simply a sliding window over this sequence.

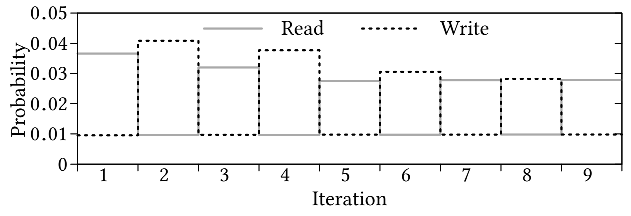


Fig. 9: Each iteration alternates the workload proportion 90%-10% and vice versa. Sifter adapts to these changes, increasing the sampling probability of the underrepresented class each iteration.

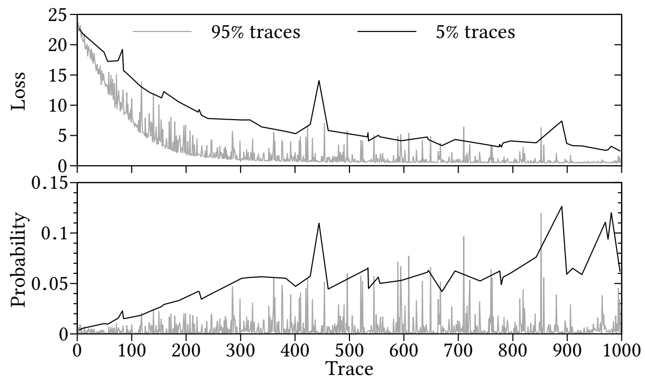
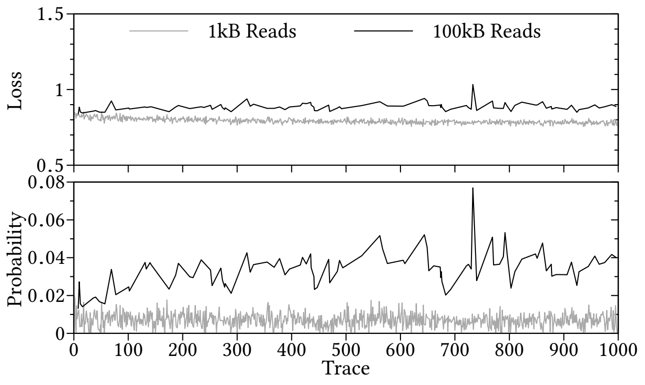
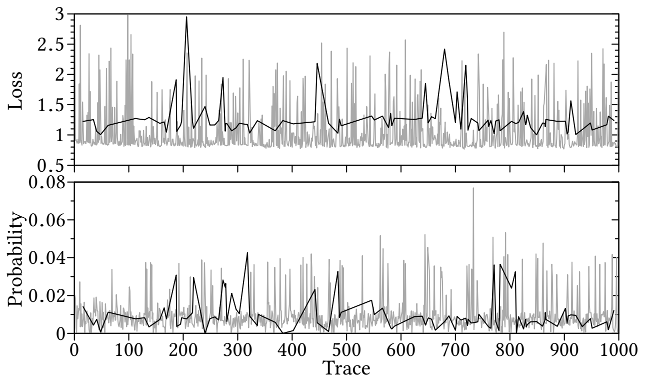


Fig. 10: Sifter requires no bootstrapping or pre-training.

We apply Sifter and Lifter to a workload of 1,000 traces comprising 90% 1kB HDFS reads and 10% 100kB HDFS reads. Figure 11a compares the loss and sampling probability for Sifter and Lifter. We observe that the loss for Lifter has significantly more fluctuations than Sifter, owing the fact that concurrent paths are superimposed on one another. The average sampling probabilities for 1kb and 100kb traces respectively are 0.010 and 0.010 for Lifter, compared to 0.007 and 0.034 for Sifter.



(a) Sifter samples underrepresented traces with higher probability.



(b) Lifter fails due to lack of trace structure.

Fig. 11: Trace structure is key to Sifter's ability to distinguish underrepresented traces. Lifter, a modified version that works on linearized traces, fails to distinguish subtle differences in structure.

7.6 Overheads

To evaluate the sampling overhead of Sifter, Figure 13 plots the distribution of sampling latencies for traces from our experiments. Sampling latencies vary between 3ms and 20ms. The factors that contribute to increased sampling latency are the size of the trace, and the number of unique labels in the trace. However, sampling latency is independent of the workload volume, and the number of previously sampled traces. By comparison, Figure 12 plots the average overhead for a hierarchical clustering approach [17]. The key limitation of this prior approach is that sampling overhead grows proportional to the number of sampled traces. By contrast, Sifter's internal state Θ is constant-sized with respect to the number of sampled traces.

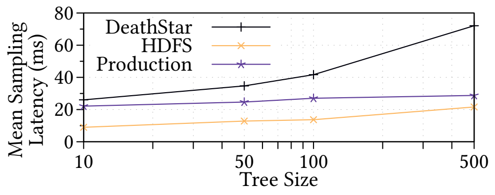


Fig. 12: Sampling latency for Hierarchical Clustering grows proportional to the number of sampled traces.

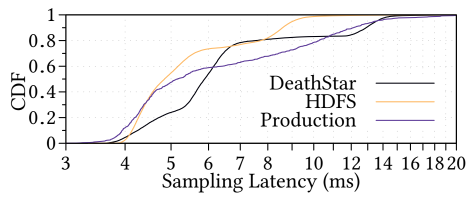


Fig. 13: Distribution of Sifter sampling latency for datasets.

8 Discussion

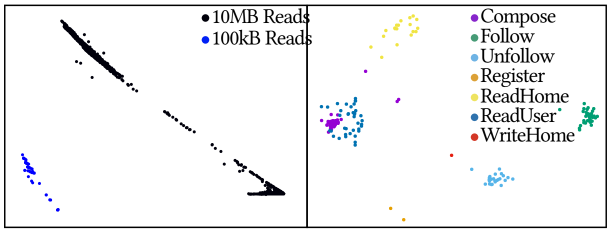
In this work, we were interested in sampling traces based on the structure of the traces themselves. The main advantage of this approach over manual feature engineering (cf. §3.1) is that it no longer requires explicit features. We believe that differences in high-level metrics can always be explained by differences in the timing and ordering of events in the underlying traces, so it is here that sampling should be done. Nonetheless, Sifter is only part of the story. Useful features often *are* available, such as end-to-end request latency. By definition, these features exist because an engineer previously found them useful for some particular problem, and outlier values offer a strong signal that a trace should be sampled. When this is a case, simple and computationally efficient statistical techniques will do a good job of sampling. Sifter is not intended to replace this use case, but to handle the case when engineered features do not capture differences between traces.

In practice, instead of a single global sampling policy, it is often desirable to specify separate sampling policies for different stratified populations. Sifter can extend to this scenario in a straightforward way: subpopulations share the same model, but maintain a separate window of losses, each parameterized with their own α (cf. §5.3).

Sifter makes use of a popular neural network architecture for language modeling [18]. We did not exhaustively explore either the hyperparameters of the model, or possible refinements to model architecture. We avoided prematurely designing custom models, as our goal was to explore the feasibility of our proposed model-based sampling approach. Nonetheless, we believe that Sifter may benefit in the future from more specialized models, such as structural autoencoders and embeddings [39].

Sifter's internal state is independent of workload volume, but does grow proportional to the number of unique labels ($|\Sigma|$). We envision two possible approaches to deal with vocabulary size. First, we can evict old labels using a least-recently-used cache; this would garbage collect old labels when they become irrelevant after a software version update. Second, we can overload labels (*e.g.* by hashing labels), since the model is robust to labels with multiple meanings.

Sifter only considers happened-before relationships and event labels, as we were primarily interested in trace structure. A natural next step would be to incorporate event timing, which many trace events report. Beyond this, it is an open question whether arbitrary event annotations can be incorporated in an efficient and general-purpose way.



(a) HDFS **(b) DeathStar benchmark**

Fig. 14: PCA projection shows clusters of different trace types.

Lastly, an interesting side-effect of model-based sampling is the model itself. While Sifter uses loss to make sampling decisions, an alternative approach would be to use the intermediate embeddings produced by Sifter. Figure 14 plots a PCA projection of traces from HDFS and the DeathStar social network benchmark. The figure shows clear separation for traces in different clusters. For HDFS traces, there is also clear separation despite both request types being read requests. A possible sampling algorithm might try to uniformly sample traces across different regions of the embedding space. In practice we found that loss provided a much stronger signal for sampling, especially in cases of never-before-seen traces, for which loss will be very high, but the position in the embedding space may be arbitrary. Nevertheless, while we did not use intermediate embeddings produced by Sifter, they may be useful for other analysis tasks such as clustering or trace comparison, especially since they are a compact and efficient representation of a trace.

9 Related Work

Distributed tracing supports a range of troubleshooting and diagnosis tasks, including diagnosing anomalous requests based on structure or timing [31, 37]; diagnosing steady-state problems [6, 8, 28, 29]; identifying slow components [5, 24]; modeling workloads and resource usage [24, 32]; and identifying system bottlenecks [10, 37].

Several prior works sought to extract value from end-to-end traces in aggregate, through clustering, classification, or anomaly detection, and similar techniques could be considered for the trace sampling problem. Pairwise graph comparison often appears as a recurring primitive [2, 6, 17, 21, 24, 29]. However, many of the approaches only consider approximated versions of traces, such as string-edit-distance of linearized graphs [2, 29], event counting [17], or grammars [6].

Alternatives Sifter could have considered include frequent subgraph mining [12]; maximal common subgraphs [4], and graph kernels [36]. Many of these techniques do not approximate well, or rely on pairwise comparison, and while they have potential for other trace analysis tasks, we do not believe they are suitable for low-overhead sampling. Furthermore, as described in §3.2 it can be difficult to extend offline approaches to the online setting, and even then they can have prohibitive computational costs.

Sifter's approach takes advantage of recent results in the area of neural language modeling [3]. In particular, approaches from this research area excel at tackling high dimensionality, inferring relationships between proximate words, and handling variable length structured input [18, 25]. We are not the first to identify an analogy between natural language and traces. Pinpoint generates a probabilistic context-free grammar from paths [6], which is an intuitive way to model how the system generates events.

Sifter's approach to trace sampling is a form of *online* machine learning. Many online learning algorithms try to mitigate the fact that models bias towards more recently seen training samples. For our use case, we explicitly want to adapt to changes in workload; but it remains open how to exactly quantify this rate of change. In general, the phenomenon of temporally varying data is called concept drift [33], and advances in this area may yield solutions that can apply to Sifter. Sifter takes advantage of a concept called class imbalance, which affects many machine learning models which give better predictions for more common training examples. Common techniques to mitigate class imbalance include subsampling frequently seen samples [25]; we did not use or evaluate these techniques.

Outside of distributed tracing, prior work has also found structure useful for the metrics emitted by a system [15, 23], and for logs [20]. Complementary research also looks into placing logging statements in a way that maximizes differentiating code paths [41], and extracting structure and meaning from logging messages [7, 26].

Not discussed in this paper is trace *retention*, a similar task to sampling that can employ similar techniques. Most tracing systems simply discard traces older than a particular threshold; for example, Stardust [32] and Dapper [31] discard traces after two weeks; Canopy after a month [14]. A better approach might be to progressively subsample.

10 Conclusion

In this paper we presented Sifter, a general-purpose trace sampler that builds a low-dimensional *unbiased* model of common-case system behaviors. Using the model of common-case behaviors, Sifter exploits prediction error as a signal for identifying edge-case and anomalous traces, and biases its sampling decision towards these traces. In our evaluation, we showed that Sifter can identify and bias its sampling decisions towards edge case and underrepresented trace types. Sifter has low memory requirements, and only requires a few milliseconds per sampling decision.

References

[1] Apache. Kafka: A Distributed Streaming Platform. Retrieved June 2019 from https://kafka.apache.org/. (§2.2).

[2] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*. (§3.2, 4.3, 7.5, and 9).

[3] Y. Bengio, R. Ducharme, P. Vincent, and C. Jauvin. A Neural Probabilistic Language Model. *Journal of Machine Learning Research*, 3(Feb):1137–1155, 2003. (§1, 4.3, 5.1, and 9).

[4] H. Bunke. On a relation between graph edit distance and maximum common subgraph. *Pattern Recognition Letters*, 1997. (§9).

[5] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional Profiling for Multi-Tier Applications. In *2nd ACM European Conference on Computer Systems (EuroSys '07)*. (§9).

[6] M. Y. Chen, A. Accardi, E. Kiciman, D. A. Patterson, A. Fox, and E. A. Brewer. Path-Based Failure and Evolution Management. In *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI '04)*. (§4.3, 4.4, and 9).

[7] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. (§4.2 and 9).

[8] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems*

Design and Implementation (NSDI '07). (§1, 6, and 9).

[9] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, et al. An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud and Edge Systems. In *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. (§1 and 7.1).

[10] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '19)*. (§4.3 and 9).

[11] Jaeger. Jaeger: Open Source, End-to-End Distributed Tracing. Retrieved June 2019 from https://www.jaegertracing.io/. (§1, 2.1, and 6).

[12] C. Jiang, F. Coenen, and M. Zito. A survey of frequent subgraph mining algorithms. *The Knowledge Engineering Review*, 2013. (§9).

[13] R. Johnson. Facebook's Scribe technology now open source. Retrieved August 2017 from https://www.facebook.com/note.php?note_id=32008268919. (§2.2).

[14] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Vekataraman, K. Veeraraghavan, and Y. J. Song. Canopy: An End-to-End Performance Tracing And Analysis System. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*. (§1, 2.1, 2.3, 3.1, 4.1, 4.2, 4.3, 4.5, 5.4, and 9).

[15] M. P. Kasick, J. Tan, R. Gandhi, and P. Narasimhan. Black-Box Problem Diagnosis in Parallel File Systems. In *8th USENIX Conference on File and Storage Technologies (FAST '10)*. (§9).

[16] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, 1978. (§4.1).

[17] P. Las-Casas, J. Mace, D. Guedes, and R. Fonseca. Weighted Sampling of Execution Traces: Capturing More Needles and Less Hay. In *10th ACM Symposium on Cloud Computing (SOCC '18)*. (§1, 3.2, 3.3, 4.3, 7, 7.3, 7.3, 7.6, and 9).

[18] Q. Le and T. Mikolov. Distributed Representations of Sentences and Documents. In *31st International Conference on Machine Learning (ICML '14)*. (§5.1, 3, 8, and 9).

[19] J. Leavitt. End-to-End Tracing Models: Analysis and Unification. B.Sc. Thesis, Brown University, 2014. (§4.1).

[20] J.-G. Lou, Q. Fu, S. Yang, Y. Xu, and J. Li. Mining Invariants from Console Logs for System Problem Detection. In *USENIX Annual Technical Conference (ATC '10)*. (§9).

[21] J. Mace. Revisiting End-to-End Trace Comparison with Graph Kernels. M.Sc. Project, Brown University, 2013. (§3.2, 4.3, and 9).

[22] J. Mace and R. Fonseca. Universal Context Propagation for Distributed System Instrumentation. In *13th ACM European Conference on Computer Systems (EuroSys '18)*. (§4.4).

[23] J. Mace, R. Roelke, and R. Fonseca. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *25th ACM Symposium on Operating Systems Principles (SOSP '15)*. (§4.1 and 9).

[24] G. Mann, M. Sandler, D. Krushevskaja, S. Guha, and E. Even-Dar. Modeling the Parallel Execution of Black-Box Services. In *3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '11)*. (§4.3, 4.4, and 9).

[25] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed Representations of Words and Phrases and their Compositionality. In *27th Conference on Neural Information Processing Systems (NIPS '13)*. (§5.1 and 9).

[26] K. Nagaraj, C. Killian, and J. Neville. Structured Comparative Analysis of System Logs to Diagnose Performance Problems. In *9th USENIX Conference on Networked Systems Design and Implementation (NSDI '12)*. (§4.2 and 9).

[27] OpenTracing. OpenTracing. Retrieved January 2017 from http://opentracing.io/. (§1 and 2.1).

[28] K. Ostrowski, G. Mann, and M. Sandler. Diagnosing Latency in Multi-Tier Black-Box Services. In *5th Workshop on Large Scale Distributed Systems and Middleware (LADIS '11)*. (§9).

[29] R. R. Sambasivan, A. X. Zheng, M. De Rosa, E. Krevat, S. Whitman, M. Stroucken, W. Wang, L. Xu, and G. R. Ganger. Diagnosing Performance Changes by Comparing Request Flows. In *8th USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*. (§4.3, 4.4, 7.5, and 9).

[30] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. (§1 and 7.1).

[31] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a Large-Scale Distributed Systems Tracing Infrastructure. Technical Report, Google, 2010. (§1, 2.1, 2.3, 4.1, 7.3, and 9).

[32] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: Tracking Activity in a Distributed Storage System. In *2006 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '06)*. (§9).

[33] H. Tian, M. Yu, and W. Wang. Continuum: A Platform for Cost-Aware, Low-Latency Continual Learning. In *10th ACM Symposium on Cloud Computing (SOCC '18)*. (§9).

[34] Twitter. Zipkin. Retrieved July 2017 from http://zipkin.io/. (§1, 2.1, and 6).

[35] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache Hadoop YARN: Yet Another Resource Negotiator. In *4th ACM Symposium on Cloud Computing (SoCC '13)*. (§1).

[36] S. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *JMLR*, 99:1201–1242, 2010. (§9).

[37] Y. Wu, A. Chen, and L. T. X. Phan. Zeno: Diagnosing Performance Problems with Temporal Provenance. In *16th USENIX Conference on Networked Systems Design and Implementation (NSDI '19)*. (§9).

[38] W. Xu, L. Huang, A. Fox, D. Patterson, and M. I. Jordan. Detecting Large-Scale System Problems by Mining Console Logs. In *22nd ACM Symposium on Operating Systems Principles (SOSP '09)*. (§4.2).

[39] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *24th ACM Conference on Computer and Communications Security (CCS '17)*. (§8).

[40] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI '12)*. (§1).

[41] X. Zhao, K. Rodrigues, Y. Luo, M. Stumm, D. Yuan, and Y. Zhou. Log20: Fully Automated Optimal Placement of Log Printing Statements under Specified Overhead Threshold. In *26th ACM Symposium on Operating Systems Principles (SOSP '17)*. (§9).

[42] X. Zhao, K. Rodrigues, Y. Luo, D. Yuan, and M. Stumm. Non-Instrusive Performance Profiling for Entire Software Stacks based on the Flow Reconstruction Principle. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*. (§4.2).

[43] X. Zhao, Y. Zhang, D. Lion, M. Faizan, Y. Luo, D. Yuan, and M. Stumm. lprof: A Non-intrusive Request Flow Profiler for Distributed Systems. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI '14)*. (§4.2).