# Unleashing Performance Insights with Online Probabilistic Tracing

M. Toslali* †, S. Qasim*, S. Parthasarathy†, F. A. Oliveira†, H. Huang†, G. Stringhini*, Z. Liu‡, A. K. Coskun*
*Boston University, †IBM Research,‡University of Maryland

***Abstract*—Distributed tracing has become a fundamental tool for diagnosing performance issues in the cloud by recording causally ordered, end-to-end workflows of request executions. However, tracing workloads in production can introduce significant overheads due to the extensive instrumentation needed for identifying performance variations. This paper addresses the trade-off between the cost of tracing and the utility of the "spans" within that trace through Astraea, an online probabilistic distributed tracing system. Astraea is based on our technique that combines online Bayesian learning and multi-armed bandit frameworks. This formulation enables Astraea to effectively steer tracing towards the useful instrumentation needed for accurate performance diagnosis. Astraea localizes performance variations using only 20-35% of available instrumentation, markedly reducing tracing overhead, storage, compute costs, and trace analysis time.**

***Index Terms*—Distributed Systems, Performance Diagnosis, Microservices, Cloud Computing, online Bayesian learning.**

## I. Introduction

Performance variations constitute a common challenge in the cloud, and diagnosing them can be a time-consuming process [1]–[16]. For instance, diagnosing an unexpected slowdown in a request can consume hours or even days [17]. Distributed tracing has emerged as an essential tool for diagnosing performance variations in the cloud [2], [4], [5], [9], [18]–[25]. Distributed tracing enables tracking the journey of a request within a distributed cloud application, as it moves through various services. A *trace* consists of instrumentation choices known as *spans*, which capture causal relationships and the propagation of latency from downstream to upstream operations. The end-to-end narrative of a request provided by distributed tracing reveals *what went wrong*, making it easier to pinpoint and resolve the issues [2], [4]–[6], [8], [9], [18]–[24], [26], [27].

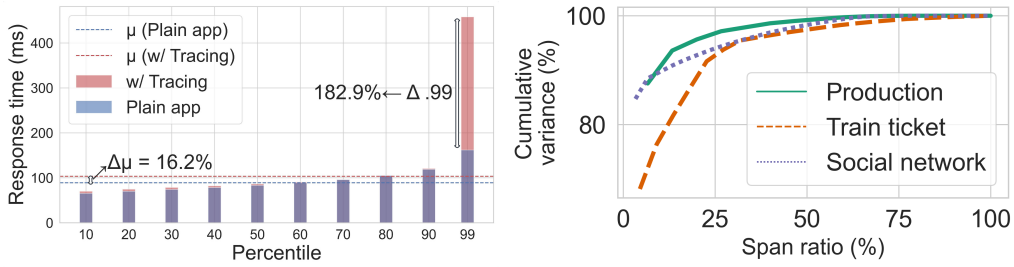
**Tracing cost vs. utility trade-off.** Predicting which code locations to instrument to diagnose potential future problems is a challenging task [19], [28]. Developers aim to trace all potential application behaviors to enable diagnosis of new problems. However, recording, processing, and storing traces from production workloads result in significant overheads in terms of storage, computation, and network usage [4], [29]. For instance, Facebook reported generating 1.16 GB/s of tracing data [4], necessitating substantial backend infrastructure and storage capacities [29]. Furthermore, the computational overhead of tracing can lead to an increase in end-to-end request latency, which is not tolerable for end users [30]. Fig. 1a shows the response time overhead of tracing (i.e., Jaeger [25]) imposed on the Train Ticket microservice application [17], revealing 180% and 16% overhead on the tail and average latency, respectively, observed in our experiments. Similarly, Google reported an average response time increase of 16.3% in their search engine due to tracing [24].

Distributed tracing often employs request-based sampling as a means to mitigate costs, allowing for tracing only a subset of requests [23], [25], [30]. While request-based sampling partially addresses the overhead problem, when it comes to performance diagnosis, the key insight is that majority of spans in a sampled request are extraneous to explain performance variation under investigation [4], [17], [19], [22], [28]. For instance, an analysis of the Alibaba traces reveals that 90% of the spans provide no useful information, with only 10% of spans being essential for performance diagnosis [10]. Fig. 1b substantiates these observations by illustrating the cumulative variance of spans from Train Ticket [17], Social Network [31], and a large-scale Internet application (denoted as Production in Fig. 1), showing that 15% of spans account for over 80% of the total variance. Identifying this "vital" set of instrumentation with the most utility is a challenge as this information is unknown *a priori* or even changes over time. For example, researchers revealed that 28K revisions in Hadoop, HBase, and ZooKeeper applications were made only to insert or modify instrumentation choices concerning cost vs. utility trade-off [32].

**Automated control of instrumentation.** To address the tracing cost vs. utility trade-off, ideally, a tracing system should adaptively and automatically control instrumentation choices to enable vital ones needed to diagnose performance variations [19], [28]. The key requirements of such a system are: (R1) correctly identifying the vital instrumentation needed to explain on-going performance variations, (R2) adapting to new sources of performance variations that might evolve over time, (R3) providing a low-overhead and scalable mechanism to efficiently control instrumentation, and (R4) addressing all requirements in a practical and developer-friendly manner.

Researchers have designed automated techniques to selectively enable the instrumentation needed (§II). VAIF [19] and Log$^2$ [28] are two automated instrumentation systems that are tailored for performance diagnosis. Log$^2$ captures all logging records in individual processes and persists ones that contribute to performance variation. However, it inherently lacks the causal context provided by distributed tracing, which captures latency propagation among distributed services (~~R1~~ and ~~R2~~). Its tail-based approach, focused on persisting logging records post-execution, primarily reduces storage overheads but neglects the computational overheads (~~R3~~).

VAIF [19], our previous work, is an automated tracing system tailored for performance diagnosis in distributed cloud applications. In an offline profiling phase, VAIF memorizes execution paths of requests through exhaustive workloads [19]. When triggered by developers at runtime, it enables spans within the region of code that currently exhibits the highest variance. VAIF is effective in




(a) Tracing overhead on mean ($\Delta\mu$) and tail ($\Delta$.99) latency

(b) Cumulative latency variance of spans

Fig. 1: The practicality of tracing is limited by (a) overhead and (b) large portion of extraneous instrumentation. (a) Tracing overheads can increase end-to-end request latency. Tracing in this experiment is conducted with a 100% sampling rate to emphasize the overhead. (b) The majority of spans in a trace are extraneous to explain variation.

localizing sources of performance variations, however, considerable developer effort is needed to run comprehensive workloads *a priori* to collect representative traces for VAIF, which is further exacerbated by the need for recurring developer-driven profiling efforts for each new code delivery (~~R4~~). Second, VAIF's span decisions are binary, offering only options of enabling or disabling spans. While this simplifies decision-making and implementation, completely disabling spans may result in a loss of accuracy as no new observations are captured for ongoing or new performance variations, as shown in §IV-C (~~R1~~ and ~~R2~~).

**Our work.** We present Astraea [1], a system backed by statistical rigor to accurately and adaptively control instrumentation to address the cost vs. utility trade-off of distributed tracing in performance diagnosis. Astraea works on top of request-based sampling to further reduce the size of individual traces utilizing span-level sampling probability. Astraea operates online from the start, eliminating the need for time-consuming offline phases. It begins with the same default level of instrumentation as current distributed tracing practices. As Astraea learns, it gradually decreases the sampling probability of unnecessary instrumentation, leading to continuous performance improvement. Astraea employs a probabilistic approach, enabling nuanced decision-making by sampling spans with varying probabilities for efficient resource utilization, accurate coverage, and adaptability to changing execution patterns or new issues.

Astraea formulates the cost vs. utility trade-off as an exploration vs. exploitation problem [33]. It continuously learns spans that explain performance variations (explore) and dynamically steers tracing toward them (exploit). This formulation enables an online learning setting tailored for automated control of instrumentation to enhance tracing utility. The Bayesian framework of Astraea enables space- and time-efficient computation by mapping large trace data to a low-dimensional Bayesian belief representation, gradually building beliefs of spans, thus eliminating the need for extensive trace data storage (§III). Based on accumulated beliefs, Astraea employs a probability matching decision strategy where the sampling probability of a span is proportional to the fraction of observations that a span emerges as a vital contributor to variation, aiding in continuously exploring and resolving uncertainty to identify vital spans [34].

[1]Astraea is accessible at https://github.com/ai4cloudops/Astraea.

Overall, Astraea enhances the accuracy and adaptability of identifying spans required to explain both current and new performance variations (addressing R1 and R2), while maintaining low overheads and scaling effectively to handle large production traces (addressing R3).Astraea eliminates the need for cumbersome offline training or retraining phases (addressing R4). The primary goal of Astraea is to elevate the practical utility of distributed tracing to new heights. It empowers developers to instrument their source code during the development phase without worrying about the costs and manual analysis associated with managing a large number of extraneous spans in response to performance variations at runtime.

**Our contributions and highlights.**

- We devise and formulate the tracing cost vs. utility trade-off as an exploration vs. exploitation problem, enabling an online learning setting. Building on our formulation, we design and implement Astraea, which dynamically steers tracing toward vital regions.
- We show that Astraea accurately localizes injected performance variations 92% of the time, using only 20-35% of instrumentation in three widely used distributed cloud applications (Social Network, Media, and Train Ticket) (§IV). In contrast, Log$^2$ and VAIF achieve 65% and 71% accuracy, enabling comparable or larger instrumentation and incurring 3-24x more latency overhead.
- We demonstrate the efficacy of Astraea in diagnosing performance variations in Social Network, Media, and Train Ticket applications deployed on two public clouds, namely CloudLab (on VMs) and New England Research Cloud's (NERC) OpenShift Container Platform. Astraea pinpoints sources of high variance due to implementation bugs, resource-related issues, network delays, deployments, and inefficient instrumentation, enabling only 20-35% of available instrumentation (§V).

## II. Background and Related Work

This section provides a background on distributed tracing and reviews prior work on tracing, instrumentation, and their automation.

### A. Distributed tracing overview

Figure 3 shows how distributed tracing frameworks operate in general [4], [25]. An instrumented service (e.g., Social Network in Fig. 3) creates a *span* when receiving a new request and attaches context information (e.g., traceId and spanId) to outgoing requests. A span represents a logical unit of work with an operation name,

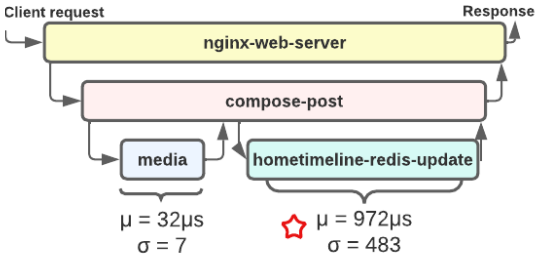



Fig. 2: A simplified trace from Social Network. Analysis on $\mu$ and $\sigma$ of span latency helps localize a performance issue.

start time, and duration. Spans are nested and ordered to model causal relationships (i.e., parent/child relationships represent caller/callee). The tracing backend collects, orders, and stitches span records with the same traceId to create end-to-end traces. A final trace is an execution path of a request through the system [19].

The causal ordering of spans enables composing a narrative of a request's execution, making it easier to pinpoint issues. The following example illustrates how distributed tracing helps diagnose a performance issue we found using Astraea (§V). Fig. 2 demonstrates a simplified trace, along with various statistics of operation durations (mean ($\mu$) and standard deviation ($\sigma$)) from the Social Network application. The root cause of the issue is an inefficient implementation that sends one request for each key to Redis instead of querying with multiple keys [35], [36]. Analysis via distributed tracing reveals that the Redis update operation shows high variance and significantly contributes to response time.

Distributed tracing helps pinpoint the code regions that initiate performance problems [5], [37]. However, there are challenges with efficient and practical use of distributed tracing in production. Developers wish to comprehensively trace the source code to easily diagnose new problems [19], [28]; however, they struggle with the cost vs. utility trade-off [32]. Reducing latency is one of the primary concerns that lead developers to use distributed tracing. However, overheads of distributed tracing also contribute to latency, as shown in Fig. 1. Understanding this impact is vital in choosing the right granularity for tracing. We briefly describe overheads with respect to Fig. 3 (marked with circled numbers). ① Spans are created on the critical path of requests. This step incurs *computational runtime overhead*. ② Tracing agents listen for spans from the application, receive, and route them to the tracing backends. This step incurs *network overheads*. ③ Tracing backends persist the trace data, where they can be later queried. This step incurs *storage and compute overheads*.

### B. Related work

**Distributed tracing & diagnostics.** Past research has shown a variety of use cases of distributed tracing in cloud [2], [4]–[6], [8], [9], [20], [21], [23], [24], [26], [37]–[41]. Zeno [39] uses traces to infer off-path root causes such as resource contentions. SAGE [20] is a machine learning-based system that leverages tracing to identify culprit services responsible for QoS violations. These state-of-the-art systems aim to facilitate rich use cases but are not geared towards resolving the cost vs. utility of the spans trade-off.

**Request-based sampling.** To help with various overheads of distributed tracing, existing techniques include head- and tail-based request sampling [23]–[25], [42]. Head-based method decides whether to trace a request uniformly at random at the beginning of a request, circumventing computational, network, and storage overheads (①, ②, and ③ in Fig. 3). On the other hand, tail-based methods capture traces for all requests and later decide whether to persist a trace, circumventing only storage overheads (③ in Fig. 3). For example, Sifter [23] biases sampling decisions towards outlier traces with respect to their frequency. While effective at reducing various overheads, these techniques are orthogonal to the cost vs. utility of the spans trade-off. That is, they don't control the instrumentation at the granularity of spans while tracing; thus, resultant traces include a large portion of extraneous spans that constitute the majority of costs. When diagnosing the current performance variation, only a few spans are useful, as a large portion of them are extraneous, constituting the majority of cost [19], [22], [28]. For example, the Fig. 1b shows that 16% of spans constitute more than 80% of the total variation.

**Dynamic instrumentation.** Some techniques allow inserting instrumentation at runtime in almost arbitrary locations in applications [22]. These techniques provide crucial flexibility during performance debugging. However, manual and iterative exploration of potential instrumentation locations in source code is a labor-intensive process, incurring prolonged diagnosis times [30]. Unlike dynamic instrumentation, Astraea does not rely on end users to manually explore data and identify problems but automates this process.

**Automated control of instrumentation.** To address cost vs. utility trade-off, researchers have proposed automated techniques that selectively enable instrumentation choices needed [19], [28], [32], [43]. However, most focus on correctness (or failure) problems, not performance variations. For example, Log20 [32] enables log points to differentiate request workflows, aiming to help identify faulty executions. However, they are not sufficient for performance diagnosis because additional instrumentation may be needed to identify where on a unique workflow a performance problem lies. In contrast, Astraea focuses on identifying areas of the code that lead to requests' overall execution to be slow. For example, inefficient implementation of a Redis update might not break functionality, however, can cause a significant slowdown (Fig. 2). The systems closest to our work are VAIF and Log$^2$, tailored for performance variations. We compare Astraea with these systems in §IV.

## III. Astraea

While an ideal tracing solution would focus effort on parts of an application that are important for performance diagnosis, this information is unknown *a priori*, hence the need for learning at runtime. Astraea is an online, probabilistic tracing system designed to combat this challenge. Fig 3 shows Astraea components that implement the logic of effectively learning the parts of an application that are needed for diagnosis (exploration) and confidently steering tracing toward the most rewarding instrumentation choices (exploitation). Astraea works in a continuous loop. As the application receives user traffic, spans are collected and stitched together by collector(Jaeger) and stored in a database. Astraea periodically queries the tracing database to build Bayesian belief distributions to estimate span sampling probabilities which represents the usefulness of a span for performance diagnosis

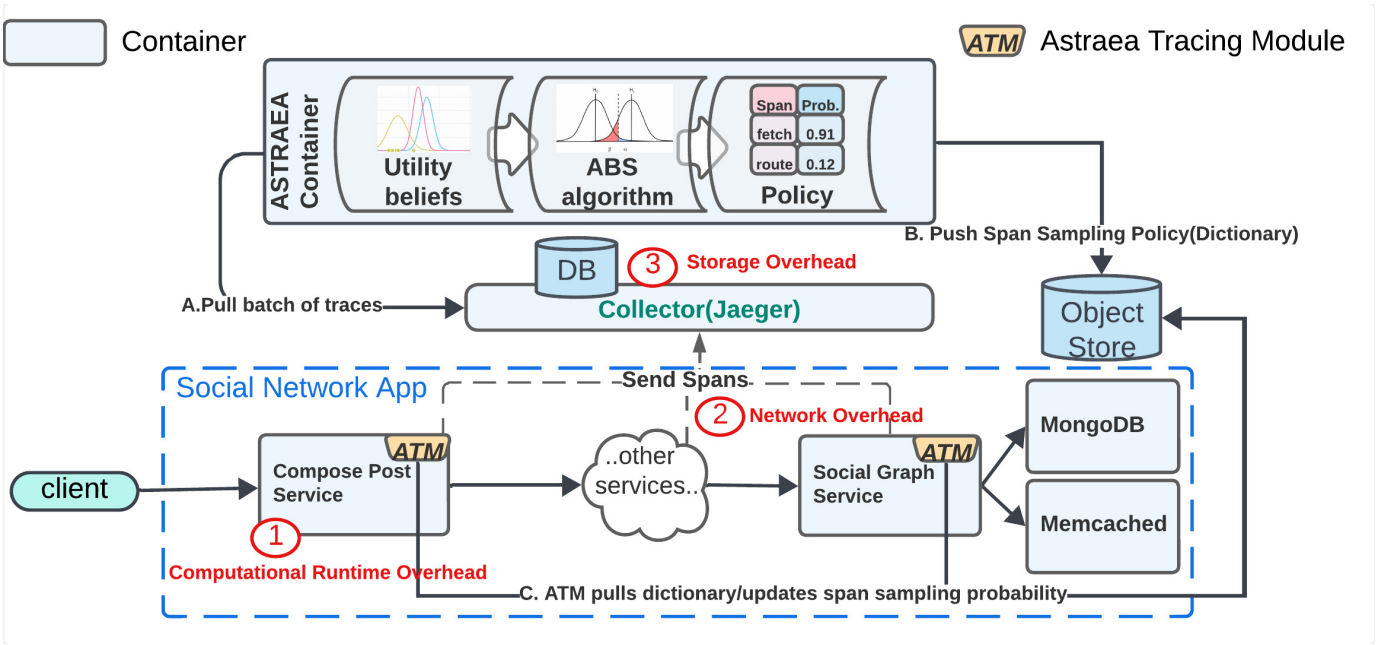



Fig. 3: An overview of the distributed tracing architecture with Astraea on a cloud deployment. The bottom displays a simplified Social Network application, with the top featuring Astraea components guiding tracing to rewarding spans. Tracing overheads are demonstrated with circled numbers.

(§III-A); as more data is available, the belief distributions converge to the true values of the utility (§III-B). It then publishes this dictionary of spans (policy) to Object store. Astraea tracing agents pull this policy periodically and updates their in memory dictionary. Initially this policy has 100% sampling rate for all spans.

Fig. 3 shows a snippet of a sample policy inside Astraea container, where *fetch* is the span and its sampling probability is 0.91. Astraea continuously builds and updates this policy to steer tracing toward vital spans, using its Approximate Bayesian Sampling algorithm considering exploration vs. exploitation trade-off (§III-B).

**Output.** Astraea is designed to help developers diagnose unanticipated performance variations by dynamically controlling the instrumentation in running systems concerning the cost vs. utility of spans trade-off. The main output from Astraea is, in fact, distributed traces; however, the data contained in the traces is the set of vital spans that are most able to explain performance variations. Second, Astraea provides users a query interface that reports the ranking of spans based on their utilities, top-k problematic spans, confidence levels (e.g., the probability that a span is vital), and significantly correlated tags (e.g., service.version) within traces to help developers interpret the results regarding localized code regions (§V).

**Design principles.** Astraea adheres to the following principles to address practicality, accuracy, and efficiency requirements (§I).

- **Statistical rigor:** While Astraea doesn't offer explicit accuracy guarantees, our method diverges from a greedy strategy of consistently opting for the highest variance span. Rather, Astraea constructs Bayesian belief distributions, guiding decisions based on statistical confidence. Consequently, it demonstrates superior accuracy in pinpointing performance issues, as evidenced by the experimental results (§IV).
- **Adaptive:** Binary decisions on enabling (or disabling) spans prevent existing tools from adapting to changing sources of variations. Astraea's probabilistic approach enables keeping pace with the changes on-the-fly (§III-B).
- **Low-overhead & scalable:** Astraea's low-dimensional Bayesian approach enables space- and time-efficient computations to decide whether to create spans based on accumulated beliefs, which helps avoid computational, network, and storage overheads (①, ②, and ③). Because Astraea maps large trace data to compact Bayesian representation, it circumvents storing large chunks of trace and can scale well with production workloads that have 1000's of spans.
- **Online:** Training the entire tracing solution is cumbersome, exacerbated by retraining due to frequent code updates. Astraea embodies an online learning framework to eliminate offline phases.

### A. *Span utility*

We now examine how Astraea decomposes latency contributions of spans and present the span utility measure, which assesses the effectiveness of spans in performance diagnosis.

**Latency decomposition.** Fig. 4 shows the latency decomposition of spans. Trace includes a root span that corresponds to the client's request to the web server (span A) that calls various operations in parallel (span B and C) and sequentially (span D) after receiving responses from preceding operations. The latency contribution of a leaf span (e.g., span D) is determined by the processing duration of itself. On the other hand, a non-leaf span with one or more children (e.g., span A) is further decomposed into segments (i.e., self_segment and child_waiting). Astraea relies on the self_segment that represents the amount of time spent by span itself.

**Utility.** Astraea defines the *utility* as something that represents how much a span contributes to performance diagnosis. The utility

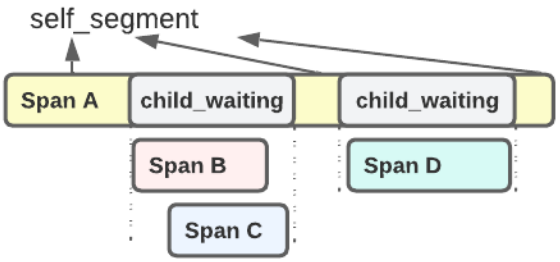



Fig. 4: The latency contribution of a leaf span (B, C, and D) corresponds to its processing duration. In the case of non-leaf spans, Astraea isolates the self_segment, representing the duration when an operation is not awaiting the completion of a child operation.

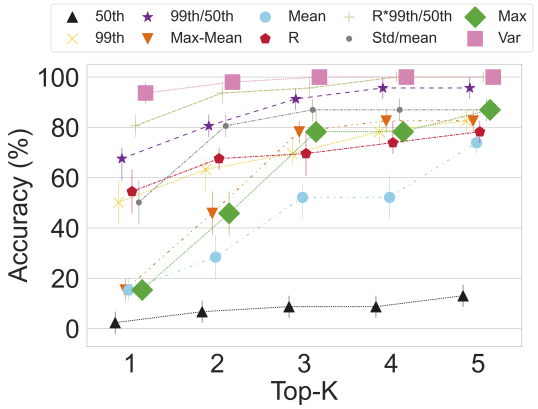



Fig. 5: Various statistical measures for span utilities. Figure evaluates accuracy in terms of whether top-k spans (with the maximum sampling probabilities in Astraea with given utility) capture the faulty spans (correspond to the location of injected problem).

serves the following two purposes. First, it is considered a *reward function* in Astraea's online learning framework to identify and focus on spans with the highest utility [33]. Second, utilities can also be used to serve insights into developers' diagnosis efforts in terms of ranking and comparison of different spans.

We study various ways to measure the utility of a span through performance anomaly injection experiments on three cloud applications (Social Network, Media, and Train Ticket) and problems include random delays and resource contentions (§IV-A). Figure 5 shows the top-K accuracy (i.e., whether the correct span is among the top-K rankings) per statistical measure. We find that the latency variance yields the most favorable outcomes, aligning with recent research [2], [19], which consequently becomes the default utility measure in Astraea. Beyond preset utility measures, Astraea allows developers to define and use their own customized measures for specific scenarios.

**Tags analysis.** Astraea records each tag pair in spans and constructs a data structure, where features represent tag values (categorical ones encoded using sklearn LabelEncoder), and the target variable is end-to-end latency. Astraea applies Pearson correlation analysis to measure the strength of relationships between tags and latency. Analyzing tags is crucial for performance diagnosis; for instance, we illustrate how the service.version tag can reveal issues with newly deployed code versions (§V).

### B. Online learning

Astraea's online learning framework encapsulates our Bayesian online learning formulation and multi-armed bandit-based sampling algorithm (ABS). The former builds belief distributions to progressively learn span utilities, and the latter adjusts span sampling probabilities based on the belief distributions.

**Belief distributions.** Astraea associates a belief distribution with every span and updates them periodically based on the new batch of tracing data. The batched update is a practical necessity, as traces are not available instantaneously but often with delays [7], [25].

We present the mathematical intuition behind belief updates using the following scenario. Suppose we have a coin with $p$ ($=0.7$) as the probability of landing a head. In Bayesian inference, $p$ is a random variable and can be modeled using beta belief distributions. Beta distribution, $Beta(\alpha,\beta)$, describes how likely $p$ can take on each value between 0 and 1. $\alpha$ and $\beta$ parameters can be thought of as the number of successes and failures, respectively. $Beta(1,1)$ is a reasonable *prior* when we have no *a priori* information about $p$ as it distributes $p$ uniformly in [0, 1]. If we toss the coin 100 times and observe 70 heads in the sample, we can use Bayes' theorem to update our belief for $p$ (i.e., $\alpha \leftarrow \alpha +$ # of heads (vice versa for $\beta$)) to obtain a *posterior distribution* for $p$, which is now $Beta(1+70,1+30)$. This is an instance of the classic beta belief update model.

Drawing from the coin example, Astraea models span utilities in the form of beta distributions. Utility measures such as variance of latency may not be 0/1 random variables; thus, Astraea normalizes the utility observations with respect to the maximum value observed in samples. Formally, let $max(\overline{h(s)})$ represents the largest value of the sample utility function $\overline{h(s)}$. The beta distribution for span $s$, $Beta(\alpha_s,\beta_s)$, captures the uncertainty about the *normalized expectation* $\frac{\mathbb{E}[h(s)]}{max(h(s))}$. This distribution is initialized to uninformative prior, $Beta(1,1)$. Astraea then updates these distributions periodically at each iteration based on the new batch of trace observations. Let $\overline{h_e(s)}$ represents the sample utility for span $s$ at epoch $e$, and the beta belief updates for each span are illustrated as follows:

$$\alpha_{e,s} \leftarrow (1-\lambda)*\alpha_{e-1,s}+\lambda*\frac{\overline{h_e(s)}}{max(\overline{h(s)})} \quad (1)$$

$$\beta_{e,s} \leftarrow (1-\lambda)*\beta_{e-1,s}+\lambda*\frac{max(\overline{h(s)})-\overline{h_e(s)}}{max(\overline{h(s)})} \quad (2)$$

where $\lambda$ represents exponentially weighted moving average (explained below). We choose the Bayesian beliefs due to its simplicity and the following considerations. First, it provides a low-dimensional data structure, enabling space- and time-efficient computation (§IV). Second, the posterior mean of the beta distribution approaches true expectation almost surely as with more data, provided by the law of large numbers [44]. Similarly, the variance of the distribution, $\frac{\alpha\beta}{(\alpha+\beta)^2(\alpha+\beta+1)}$, goes to 0 as with more samples. Third, Astraea can easily incorporate new span utility distributions (e.g., due to a code change). Initially, the belief distributions for the new spans will exhibit higher variance; however, they will converge to their true expectations as with more requests.

***Non-stationary distributions.*** It is common to find that span utility distributions drift over time (e.g., due to changing

performance problems). To address this, Astraea employs $\epsilon$-exploration strategy that prevents span sampling probabilities from reaching zero, ensuring observation of new samples with minimum probability of $\epsilon$ (default 5%). In §IV-D, we experiment with various values of $\epsilon$ to provide insights on this parameter.

**Approximate Bayesian sampling.** *Given the span observations until now, how can we quantify the extent to which a span is worth sampling?* Fig. 1b shows most of the performance variations are contributed by a small number of spans, similar to what we observed (right-skewed) in traces from benchmark applications and the production system (§IV). Astraea employs a percentile-based threshold (denoted by *P*, default $75^{th}$) to determine whether a span belongs to the *vital set* that contributes most to performance variation. We choose the percentile-based threshold for the following reasons. First, the percentile is an intuitive measure that depicts where a span stands relative to others. Astraea focuses tracing effort on spans that fall above the *P* (i.e., vital set). Second, the percentile serves as a control knob, modeling the developers' preferences. By setting *P* to higher values (e.g., $90^{th}$), developers can increase the focus on the tail portion and enjoy a higher reduction in trace sizes (§IV-D).

The probability that a span belongs to the vital set can be approximated using random sampling, referred to as Monte Carlo methods [44]. Recall that $Beta(\alpha_s, \beta_s)$ represents the posterior belief distribution for the normalized utility ($\frac{\mathbb{E}[h(s)]}{max(h(s))}$). Astraea samples multiple times from this distribution, and the collected data is used to approximate the desired quantity. Given the law of large numbers [44], larger number of random samples enable a more accurate approximation (e.g., 1e+6 in Astraea).

Given the posterior probabilities, Astraea now faces a fundamental trade-off between exploration (of span utilities) vs. exploitation (confidently steering tracing toward vital spans). Exploration-exploitation trade-off is best exemplified by the multi-armed bandit problem [7], [34]. Drawing from the bandit problem, our ABS algorithm chooses the spans with probability matching decision strategy [34]. To illustrate, suppose that span A belongs to vital set 90% of the samples and span B with 10%. A greedy strategy is to always enable A and disable B, hence can forgo an opportunity to learn about span B. In contrast, ABS samples spans A and B with probabilities equal to 0.9 and 0.1, respectively. As such, ABS explores to resolve uncertainty to help identify the optimal spans.

More formally, Astraea uses the Monte Carlo procedure to create a sample utility approximation matrix with each row containing a single utility sample for each span. Given the matrix, ABS picks the spans that exceed the given percentile *P* of utility as per row of the matrix and marks them as candidates. In the final sampling policy, the sampling probability of a span equals the fraction of rows in which the span emerged as the candidate. Since the posterior probability of suboptimal spans converges to 0 over time (§III-B), ABS progressively shifts tracing towards the optimal spans as desired. Our algorithm is inspired by the classic Bayesian algorithms designed for such explore vs. exploit problems [34]. These algorithms provide various benefits over classical multivariate hypothesis tests and have solid theoretical guarantees.

***$\epsilon$-ABS.*** Astraea employs *$\epsilon$-exploration* strategy to account for non-stationary behavior. So, final sampling policy computed by Astraea is equal to maximum of the probability that ABS derives and $\epsilon$.

### C. Implementation

Our Astraea prototype was developed in Python (2.5K LOC), utilizing popular data science libraries such as NumPy, SciPy, and Pandas. Our implementation involved several key components of Astraea, including a module that periodically utilizes Jaeger APIs to fetch batches of trace data and derives a sampling policy, and a module that applies sampling decisions to the instrumented applications.

During each cycle, Astraea fetches a batch of trace data and converts them to a low-dimensional Bayesian format. To be specific, we assign a belief distribution to each span with $\alpha_s$ and $\beta_s$ initialized to 1, ensuring that the prior beliefs are uniformly random. Based on the span utilities acquired from the batch of trace data, belief distributions are updated. The probability that a span belongs to the vital set is approximated using Monte Carlo sampling, which is the most computationally intensive task. To enable fast and efficient computations for this step, we utilize NumPy, which implements statistical sampling functionality in C. Astraea periodically applies span sampling policies to the tracing modules in the application layer. These policies map span identifiers to their corresponding sampling probabilities. We use a combination of service name, operation name, and URL to identify individual spans, addressing instances of identical operation names.

Utilizing Astraea in cloud applications is a straightforward process. These applications only need an existing distributed tracing framework to enable the required support. We developed two prototype tracing modules in Java (for Train Ticket) and C++ (for Social Network and Media) languages to utilize Astraea. Our implementation introduced a low-priority background thread that periodically receives sampling policy from Astraea and loads it into memory as a dictionary. To store the policy data produced by Astraea, we utilize Docker volumes (for VMs) or publish it to Object Store (for NERC), that are then accessed by the modified Jaeger tracing module living in containers. Our modifications were kept to a minimum, with the tracing module only fetching span policy data from the volume at periodic intervals (e.g., every 5 seconds). Our policy data structure includes per-service span identifiers and their sampling probabilities, with low memory overhead (a few kilobytes).

In Jaeger C++, we only made changes to the function responsible for starting spans. Our changes include querying the in-memory policy to acquire the sampling probability of the intended span and determining whether to create the span based on that probability. In Jaeger Java, we needed to make changes in opentracing-spring-jaeger library that provides automated instrumentation for Java Web Servlet applications. Specifically, we modified the servlet filter that is used to create spans upon receiving requests in Spring stack. Similar to C++ implementation, we implemented the sampling decisions based on the probabilities fetched periodically from Astraea. The modifications needed to deliver selective enabling or disabling of spans by Astraea were minor, totaling less than 200 lines of code. In our implementations, if a span is not sampled, the function to start spans does not create new spans but returns, which frees up overheads such as computational costs (e.g., allocating a new object, attaching a reference, reading from a performance timer, and updating the thread-local state), network costs (e.g., emitting the span), and storage costs (e.g., persisting the span record) for the disabled

spans. In contrast to the alternatives of tail-based sampling (Log$^2$) or conditional checks using filesystem (VAIF), our implementation uses an in-memory sampling policy to make sampling decisions based on accumulated beliefs before creating spans, resulting in significantly lower overhead (as shown in the evaluation).

The main output of Astraea consists of distributed traces that contain vital spans for explaining performance variations. When developers use the tracing UI to address issues, they will only see crucial spans that explain the current variations. Additionally, Astraea reports the ranking of spans based on their usefulness, the most problematic spans, confidence levels (such as the probability that a span is vital), and tags that are correlated with latency.

## IV. Experimental evaluation

We evaluate the performance of Astraea in three parts. First, we assess its accuracy and trace size reduction by testing it on two benchmark distributed cloud applications (Social Network and Train Ticket), deployed on CloudLab and NERC to demonstrate the versatility and practicality of our approach. Second, we conduct a comparative analysis between Astraea, Log$^2$, and VAIF, illustrating how Astraea outperforms them in various metrics. Third, we perform sensitivity analysis, examining how Astraea's performance varies in response to internal parameters such as epsilon and percentile cutoff.

### A. Experimental apparatus

**Benchmark applications & datasets.** We use three distributed cloud applications and a trace dataset acquired from a large-scale Internet application to evaluate Astraea. *Social Network* is a broadcast-style Social Network, implemented with 36 microservices, including databases, caches, application logic, and a frontend Nginx web server, communicating with each other via Thrift RPCs. The application is instrumented with the Jaeger-CPP client [45]. *Train Ticket* is a publicly available booking system, implemented with 41 microservices that communicate via REST APIs. The application uses Opentracing java library [46] that automatically instruments the Spring stack by recording each web request/response via Jaeger-Java client [47]. *Media* implements an end-to-end service for browsing movie information, reviewing, renting, and streaming movies. The Media application consists of 38 services, and they are instrumented with Jaeger-CPP client.

**Comparison baselines.** We compare our proposed system to two automated techniques that selectively enable the instrumentation required for performance diagnosis. *Log$^2$ [28]* is an automated logging mechanism to decide 'whether to log' in such a way that the logging overhead is constrained within the budget while maximizing logging effectiveness. *VAIF [19]* is an automated instrumentation framework that integrates distributed tracing with control logic to dynamically enable the necessary instrumentation for identifying performance issues. During an offline profiling phase, VAIF maps out all possible execution paths of requests, which are referred to as the constructed search space.

**Infrastructure setup.** We performed experiments on both VMs (CloudLab) and Containers (NERC) to demonstrate practicality of our approach. Our CloudLab cluster comprised of four nodes, all running Ubuntu 18.04 with Linux 4.15,8 CPU cores (Intel(R) Xeon(R) E5-2640), 64 GB of memory, and a 1 Gbps network connection. We deployed the aforementioned applications separately using a YAML configuration file that defines a set of services and their dependencies (Docker version 23.0.2).

On NERC, we deployed our services on OpenShift. Each service had by default 1 pod with 1 CPU and 512 MB RAM. The Social Network app spanned across 32 pods, and the Train Ticket app was deployed on 97 pods. We deployed Jaeger to collect spans, and Astraea utilized it to obtain traces and spans. Astraea published its span sampling dictionary to a bucket in NERC object store. All tracing agents in all pods synced this dictionary every 5 seconds. We used hey [48] and wrk2 [49] workload generators, to send requests to applications with various concurrent workers (between 5-25) and req/s (between 20-100).

To evaluate accuracy we introduced various types of performance degradations to randomly chosen operations or services [2], [3]. We selected a span in a service randomly to inject the delay. We sampled the delay value from a Gaussian distribution with a mean $\mu$ of (10 ms, 50 ms, etc.) and a standard deviation $\sigma$ of (1 ms, 2 ms, etc.). Controlling $\mu$ and $\sigma$ served as a control knob for the intensity of the variance injected. Additionally, we used Pumba [50], a chaos testing command-line tool for Docker containers, that uses CPU and memory stressors of stress-ng [51], resulting in delays in the upstream operations of congested containers.

Since we knew which span had the delay injected, we could measure if Astraea was able to correctly capture it.

### B. Astraea validation

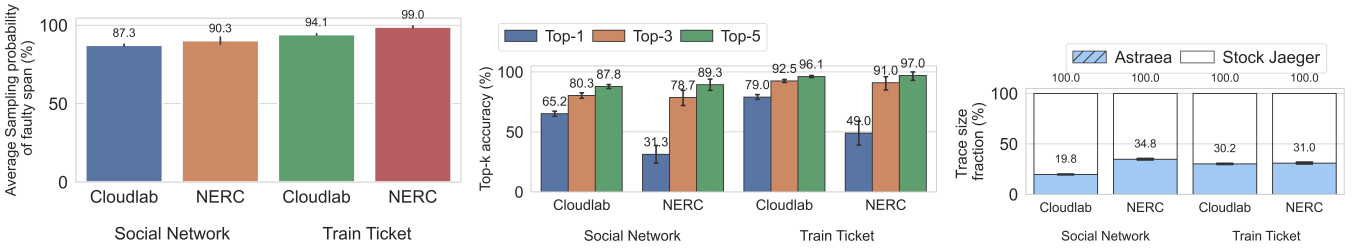
We first validate Astraea's ability to correctly localize problematic operations and steer tracing toward them. We did this experiment on CloudLab and NERC.

Fig. 6a reports the sampling probability of faulty spans, which capture problematic operations from 20 different runs per application. Astraea maximizes the sampling probability of faulty spans in all applications (above 85% on average). We define Coverage, as the average sampling probability of the faulty span for the experiment.

Fig. 6b presents whether top-k spans (with the maximum sampling probabilities in Astraea) capture the faulty spans. Astraea achieves 92% Top-5 accuracy on average. The accuracy is greater on CloudLab, as there is less overall interference, whereas, since NERC is a shared environment and has more variations inherrently, Top-1 accuracy takes a hit, but Top-3 and Top-5 accuracy are still high. Accuracy is slightly higher for Train Ticket as traces include a larger number of repeated spans and sequential executions, enabling Astraea to learn faster. Fig. 6c demonstrates the fraction of traces that Astraea uses with respect to the Stock Jaeger implementation, where all spans are enabled all the time whereas Astraea attaches a probability to each span, lowering its contribution to average trace size. We find that Astraea has around 30% as the average span sampling probability across all spans.

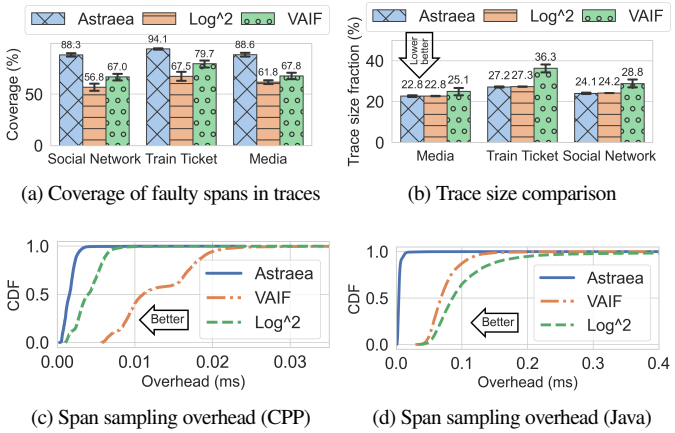
### C. Comparative analysis

We compare Astraea with VAIF and Log$^2$. We run VAIF with default parameters [19]. Log$^2$ requires a budget parameter [28], which we set equal to the average number of spans enabled by Astraea. We




(a) Sampling of faulty span (b) Accuracy of top-k spans (w/ highest sampling prob.) (c) Trace size comparison with Stock Jaeger

Fig. 6: Astraea's accuracy and reduction in trace sizes. Astraea maximizes the sampling probability of faulty spans (where problem is injected) to above 85% confidence level on average, enabling only ∼29% of all available spans cumulatively.




(a) Coverage of faulty spans in traces (b) Trace size comparison

(c) Span sampling overhead (CPP) (d) Span sampling overhead (Java)

Fig. 7: Comparison of Astraea with Log2 and VAIF in a controlled experiment. Astraea's bandit-based statistical approach outperforms baselines in terms of coverage of faulty spans while enabling fewer spans cumulatively. Astraea's sampling decisions are made before creating spans using an in-memory sampling policy data structure, incurring significantly lower latency overheads than baselines.

conducted this experiment only on CloudLab for a fair comparison, as VAIF and Log$^2$ were both originally reported on VMs.

**Accuracy.** Fig. 7a and 7b summarize the results from 20 runs per application. Astraea achieves higher coverage of faulty spans while reducing larger fraction of traces. Among the three systems, Log$^2$ is the least performing. This result stems from lacking the causal context provided in distributed tracing. Problems in child spans propagate to the parents, misleading Log$^2$. VAIF achieves higher accuracy than Log$^2$ as it leverages the causal context. Astraea surpasses VAIF by making more accurate decisions, leveraging the statistical rigor offered by the Bayesian framework. VAIF's effectiveness decreases in applications that exhibit high concurrency (Social Network and Media). It concentrates solely on critical paths, relying on the most frequent path learned offline, which hinders its ability to detect issues in less frequent paths.

**Effort.** A key difference lies in practicality. First, VAIF requires time-consuming offline phases to memorize all execution paths across the application. Astraea instead relies on live data to learn and make statistically robust decisions in an online manner. Second, Log$^2$ incurs prolonged diagnosis times. Log$^2$ falls short in ranking problematic operations as it lacks the causal context provided by tracing.

**Overhead.** We next compare Astraea's overhead with VAIF and Log$^2$. Both VAIF and Astraea selectively enable or disable spans, requiring a check embedded in tracing clients before creating new spans. VAIF's sampling decisions are conducted in-band using the filesystem. Log$^2$ works in a tail-based manner and decides whether to keep or discard the instrumentation after it is created. Figure 7d shows the measurements for Java (Train Ticket) and Figure 7c for CPP (Social Network and Media) implementations. On average, Astraea incurs 1.6 and 5.7 microseconds overhead per span decision in CPP and Java implementations, respectively. In contrast, VAIF incurs 8x and 14x, and Log$^2$ incurs 2.6x and 24x more overhead in CPP and Java, respectively. We additionally verify that Astraea's overhead on end-to-end latency in all applications is less than 1.5% on average, 6x and 7x lower than VAIF and Log$^2$.

### D. *Sensitivity analysis*

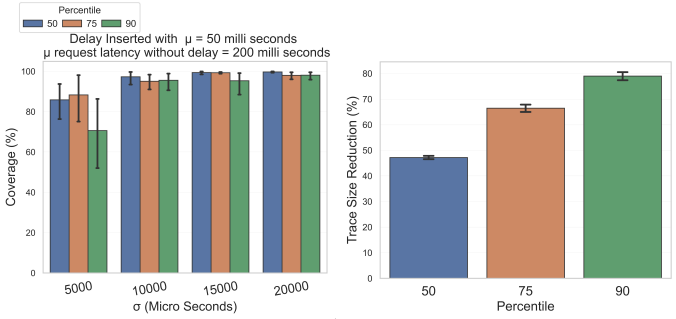
Astraea has two internal parameters (percentile threshold and epsilon exploration rate). Based on empirical evaluations, we demonstrate the impact of these parameters on Astraea's performance. We conducted these experiments for different levels of variance. We observed that as we increase the standard deviation ($\sigma$), Astraea is able to correctly identify the problematic span. However, when ($\sigma$) is very small, Astraea misses it. Yet, these minimal variations are insignificant and are not major contributors of performance variations of the application.

**Percentile-based elimination:** Astraea focuses on steering tracing toward a vital set of spans (§III-B). Percentile here determines the percentage of spans that Astraea will aim to reduce. So a 90 percentile value means the sampling probability of 90/100 spans being dropped. Fig. 8a and 8b summarize results for various percentile levels. Higher percentile levels translate into a higher reduction in trace sizes by Astraea at the expense of lower accuracy as problems with low variance might not be detected.

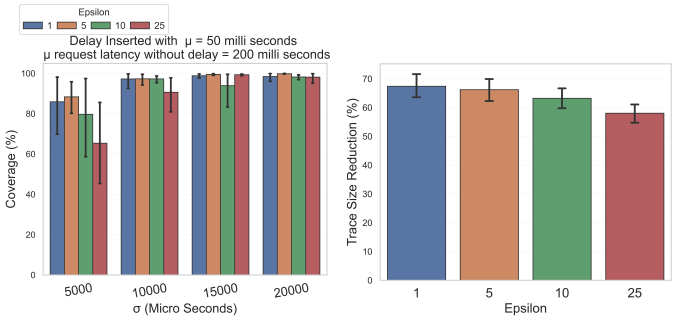
**Exploration rate:** Astraea can adapt to changing sources of variations on-the-fly using $\epsilon$-exploration (§III-B). Figure 8c shows Astraea's accuracy for various epsilon values. $\epsilon$-exploration is also the least amount sampling can get for any span, i.e., $\epsilon = 5$ means, Astraea will not lower sampling probability of any span below 5. Figure 8d presents the reduction in trace sizes. Higher epsilon values translate into lower savings. We choose the default exploration rate as 5%, and observe that Astraea can confidently steer tracing toward new problems with this rate value.

| Case | Description | App / API | Utility | Samples | Savings |
|---|---|---|---|---|---|
| 1. Implementation bug | Redis update portion of the WriteHomeTimeline function causing slow down (i.e., no batched Redis query with multiple keys) | SN / CP | ∼57% | ∼450 | ∼89% |
| 2. Network delay | NGINX web server experiences latency variation when accessing Compose service | SN / CP | ∼12% | ∼600 | |
| 3. Resource throttling | Ticketinfo query API experiences delays before calling downstream service (i.e., overloaded downstream service) | TT / QI | ∼41% | ∼550 | ∼83% |
| 4. Implementation bug | UploadRating operation in Rating service experiences delay after calling downstream services (i.e., unneccesarily waiting async operations) | Media / CR | ∼25% | ∼500 | ∼86% |
| 5. Incorrect instrumentation | MongoFindUser span within UserReview service shows high response times (i.e., span.finish() is never called) | Media / CR | ∼12% | ∼700 | |
| 6. Implementation bug | ReadUserTimeline operation shows high response time variation (i.e., Case 9 in [37]) | SN / RUT | ∼65% | ∼200 | ∼72% |
| 7. Network delay | NGINX experiences delays when accessing UserTimeline service (i.e., Case 1 in [37]) | SN / RUT | ∼25% | ∼250 | |
| 8. Deployment | Canary version of the UniqueId service shows high response times | Media / CR | ∼33% | ∼300 | ∼84% |

TABLE I: Performance variations localized by Astraea. We report the case description, utility, # of samples for 90% confidence, and savings (reduction in traces compared to Stock Jaeger tracing) measures on applications SN (Social Network), TT (Train Ticket) and APIs CP (ComposePost), QI (QueryInfo), CR (ComposeReview), and RUT (ReadUserTimeline).




(a) Accuracy for various percentiles (b) Trace sizes for various percentiles




(c) Accuracy for various $\epsilon$ (%) (d) Trace sizes for various $\epsilon$ (%)

Fig. 8: Sensitivity analysis on Train Ticket Application. (a-b) Higher percentile thresholds may allow a higher reduction in trace sizes; however, may decrease accuracy. (c-d) Increasing epsilon exploration rate lowers size reduction in traces.

## V. Case studies

This section demonstrates the efficacy of Astraea in performance diagnosis. We present eight case studies of how we used Astraea to diagnose performance variations on Social Network, Train Ticket, and Media applications. We run Astraea with all applications and observe its interface that reports sources of eight performance issues found with 90% confidence, which include implementation bugs, resource-related delays, network delays, deployment, and incorrect instrumentation (Table I). Seven cases were actually present in the applications and discovered by Astraea. Only one (Case 8) was injected to show how Astraea can effectively pinpoint performance variations due to deployments. We discuss two of the cases below.

**Case 1:** We run Astraea in the Social Network application with no injected anomaly. We find that resulting traces are 89% smaller than Stock Jaeger. Within 450 sampled requests, Astraea concludes some performance problems with 90% confidence. The most vital span corresponds to Redis update portion of the WriteHomeTimeline function. The implementation sends one update request for each key as Redis++ currently does not support a pipeline with multiple keys. This operation incurs high latency variation and corresponds to 57% of the total utility observed in traces. We verify this problem via code documentation (authors have a ToDo note on this issue [36]).

**Case 8:** We demonstrate how Astraea helps diagnose an issue caused by the deployment of a new version. We run Astraea with the Media application; 5 minutes in, we deploy a canary version of unique-id service. The canary code is modified to introduce random latency spikes, following a normal distribution, and encode the version information as a tag of the corresponding span. Astraea localizes the performance issue and also infers that the version tag in the span is significantly correlated with the end-to-end latency (r = 0.83).

## VI. Conclusion

We presented Astraea, an online, probabilistic tracing system designed to combat the cost vs. utility of the "spans" trade-off. Our key contribution was formulating this tradeoff as an exploration vs. exploitation problem. Unlike prior work, Astraea embodies a statistically rigorous, online approach to accurately and adaptively localize and trace the sources of varying performance variations. Further, we demonstrated Astraea's practicality, accuracy, and resource efficiency through extensive experiments. We showed that Astraea can localize 92 % of the performance variations using only 25% of the available instrumentation in three popular distributed cloud applications (Social Network, Media, and Train Ticket), while existing approaches, Log2 and VAIF, achieve 65% and 71% accuracy, incurring 3-24x more latency overhead. As more complex services will continue to be built in the cloud, systems like Astraea can help localize performance problems without introducing significant overheads.

## References


[1] Akamai, "Akamai online retail performance report: Milliseconds are critical.." https://www.akamai.com/uk/en/about/news/press/2017-press/akamai-releases-spring-2017-state-of-online-retail-performance-report.jsp, Apr 2017.

[2] H. Qiu, S. S. Banerjee, S. Jha, Z. T. Kalbarczyk, and R. K. Iyer, "FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 805–825, USENIX Association, Nov. 2020.

[3] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, pp. 74–80, 2013.

[4] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, and Y. J. Song, "Canopy: An end-to-end performance tracing and analysis system," in *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, (New York, NY, USA), p. 34–50, Association for Computing Machinery, 2017.

[5] 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)*, 2011.

[6] M. Toslali, S. Parthasarathy, F. Oliveira, and A. K. Coskun, "JACKPOT: Online experimentation of cloud microservices," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, USENIX Association, July 2020.

[7] M. Toslali, S. Parthasarathy, F. Oliveira, H. Huang, and A. K. Coskun, "Iter8: Online experimentation in the cloud," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC, (New York, NY, USA), p. 289–304, Association for Computing Machinery, 2021.

[8] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pp. 405–417, 2018.

[9] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, "{X-Trace}: A pervasive network tracing framework," in *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI)*, 2007.

[10] S. Luo, H. Xu, C. Lu, K. Ye, G. Xu, L. Zhang, Y. Ding, J. He, and C. Xu, "Characterizing microservice dependency and performance: Alibaba trace analysis," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC, (New York, NY, USA), p. 412–426, Association for Computing Machinery, 2021.

[11] J. Teoh, M. A. Gulzar, G. H. Xu, and M. Kim, "Perfdebug: Performance debugging of computation skew in dataflow systems," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC, (New York, NY, USA), p. 465–476, Association for Computing Machinery, 2019.

[12] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI, (New York, NY, USA), p. 77–88, Association for Computing Machinery, 2012.

[13] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating Root-Cause diagnosis of performance anomalies in production software," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Hollywood, CA), pp. 307–320, USENIX Association, Oct. 2012.

[14] D. Ardelean, A. Diwan, and C. Erdman, "Performance analysis of cloud applications," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, (Renton, WA), pp. 405–417, USENIX Association, Apr. 2018.

[15] L. Luo, S. Nath, L. R. Sivalingam, M. Musuvathi, and L. Ceze, "Troubleshooting Transiently-Recurring errors in production systems with Blame-Proportional logging," in *2018 USENIX Annual Technical Conference (USENIX ATC)*, (Boston, MA), pp. 321–334, USENIX Association, July 2018.

[16] A. Maricq, D. Duplyakin, I. Jimenez, C. Maltzahn, R. Stutsman, and R. Ricci, "Taming performance variability," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, (Carlsbad, CA), pp. 409–425, USENIX Association, Oct. 2018.

[17] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, vol. 47, no. 2, pp. 243–260, 2021.

[18] R. R. Sambasivan, I. Shafer, J. Mace, B. H. Sigelman, R. Fonseca, and G. R. Ganger, "Principled workflow-centric tracing of distributed systems," in *Proceedings of the Seventh ACM Symposium on Cloud Computing*, pp. 401–414, 2016.

[19] M. Toslali, E. Ates, A. Ellis, Z. Zhang, D. Huye, L. Liu, S. Puterman, A. K. Coskun, and R. R. Sambasivan, "Automating instrumentation choices for performance problems in distributed applications with vaif," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC, (New York, NY, USA), p. 61–75, Association for Computing Machinery, 2021.

[20] Y. Gan, M. Liang, S. Dev, D. Lo, and C. Delimitrou, "Sage: Practical and scalable ml-driven performance debugging in microservices," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, (New York, NY, USA), p. 135–151, Association for Computing Machinery, 2021.

[21] 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 *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), p. 19–33, Association for Computing Machinery, 2019.

[22] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, pp. 378–393, 2015.

[23] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, "Sifter: Scalable sampling for distributed traces, without feature engineering," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 312–324, 2019.

[24] 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," 2010.

[25] "Jaeger, open source, end-to-end distributed tracing." https://www.jaegertracing.io/.

[26] J. Mace and R. Fonseca, "Universal context propagation for distributed system instrumentation," in *Proceedings of the Thirteenth EuroSys Conference*, EuroSys, (New York, NY, USA), Association for Computing Machinery, 2018.

[27] Z. Zhang, J. Zhan, Y. Li, L. Wang, D. Meng, and B. Sang, "Precise request tracing and performance debugging for multi-tier services of black boxes," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, pp. 337–346, IEEE, 2009.

[28] R. Ding, H. Zhou, J.-G. Lou, H. Zhang, Q. Lin, Q. Fu, D. Zhang, and T. Xie, "Log2: A Cost-Aware logging mechanism for performance diagnosis," in *2015 USENIX Annual Technical Conference (USENIX ATC)*, (Santa Clara, CA), pp. 139–150, USENIX Association, July 2015.

[29] L. Zhang, V. Anand, Z. Xie, Y. Vigfusson, and J. Mace, "The benefit of hindsight: Tracing edge-cases in distributed systems," 2022.

[30] "Distributed Tracing in Practice." https://go.lightstep.com/rs/260-KGM-472/images/distributed-tracing-in-practice-lightstep.pdf.

[31] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, (New York, NY, USA), p. 3–18, Association for Computing Machinery, 2019.

[32] 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 *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP, (New York, NY, USA), p. 565–581, Association for Computing Machinery, 2017.

[33] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. The MIT Press, second ed., 2018.

[34] W. R. Thompson, "On the likelihood that one unknown probability exceeds another in view of the evidence of two samples," *Biometrika*, vol. 25, pp. 285–294, 12 1933.

[35] Redis cluster issue. https://github.com/sewenew/redis-plus-plus/issues/212, 2022.

[36] "Redis++ currently does not support pipeline with multiple hashtags in cluster mode ." https://github.com/delimitrou/DeathStarBench/blob/66eed9b6f9fd56fbdbd750f18ba3760899c566d3/socialNetwork/src/HomeTimelineService/HomeTimelineHandler.h\#L127.

[37] L. Huang and T. Zhu, "Tprof: Performance profiling via structural aggregation and automated analysis of distributed systems traces," in *Proceedings of the ACM Symposium on Cloud Computing*, SoCC, (New York, NY, USA), p. 76–91, Association for Computing Machinery, 2021.

[38] 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)*, pp. 217–231, 2014.

[39] Y. Wu, A. Chen, and L. T. X. Phan, "Zeno: Diagnosing performance problems with temporal provenance," in *16th USENIX Symposium on Networked*

*Systems Design and Implementation (NSDI)*, (Boston, MA), pp. 395–420, USENIX Association, Feb. 2019.

[40] A. Parker, D. Spoonhower, J. Mace, B. Sigelman, and R. Isaacs, *Distributed tracing in practice: Instrumenting, analyzing, and debugging microservices*. O'Reilly Media, 2020.

[41] Z. Zhang, M. K. Ramanathan, P. Raj, A. Parwal, T. Sherwood, and M. Chabbi, "CRISP: Critical path analysis of Large-Scale microservice architectures," in *2022 USENIX Annual Technical Conference (USENIX ATC)*, (Carlsbad, CA), pp. 655–672, USENIX Association, July 2022.

[42] "OpenTelemetry." https://opentelemetry.io/docs/migration/opentracing/.

[43] E. Ates, L. Sturmann, M. Toslali, O. Krieger, R. Megginson, A. K. Coskun, and R. R. Sambasivan, "An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 165–170, 2019.

[44] M. H. DeGroot and M. J. Schervish, *Probability and Statistics*. Addison-Wesley, 3 ed., 2002.

[45] "Jaeger CPP client library." https://github.com/jaegertracing/jaeger-client-cpp/.

[46] "Opentracing Java spring framework." https://github.com/opentracing-contrib/java-spring-jaeger.

[47] "Jaeger Java client library." https://github.com/jaegertracing/jaeger-client-java.

[48] Hey-workload. https://github.com/rakyll/hey, 2022.

[49] Wrk2-workload. https://github.com/giltene/wrk2, 2022.

[50] "Pumba: chaos testing tool for Docker." https://github.com/alexei-led/pumba.

[51] "Stress-ng." https://wiki.ubuntu.com/Kernel/Reference/stress-ng.