Troubleshooting Transiently-Recurring Errors in Production Systems with Blame-Proportional Logging

Liang Luo, *University of Washington;* Suman Nath, Lenin Ravindranath Sivalingam, and Madan Musuvathi, *Microsoft Research;* Luis Ceze, *University of Washington*

https://www.usenix.org/conference/atc18/presentation/luo

This paper is included in the Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC '18).

July 11–13, 2018 • Boston, MA, USA

ISBN 978-1-931971-44-7

Open access to the Proceedings of the 2018 USENIX Annual Technical Conference is sponsored by USENIX.

Troubleshooting Transiently-Recurring Problems in Production Systems with Blame-Proportional Logging

Liang Luo[*]*, Suman Nath*[†]*, Lenin Ravindranath Sivalingam*[†]*, Madan Musuvathi* [†]*, Luis Ceze*[*]
[*]*University of Washington,* [†]*Microsoft Research*

Abstract

Many problems in production systems are *transiently recurring* — they occur rarely, but when they do, they recur for a short period of time. Troubleshooting these problems is hard as they are rare enough to be missed by sampling techniques, and traditional postmortem analyses of runtime logs suffers either from low-fidelity of logging too little or from the overhead of logging too much.

This paper proposes AUDIT, a system specifically designed for troubleshooting transiently-recurring problems in cloud-based production systems. The key idea is to use *lightweight triggers* to identify the first occurrence of a problem and then to use its recurrences to perform *blame-proportional logging*. When a problem occurs, AUDIT automatically assigns a *blame rank* to methods in the application based on their likelihood of being relevant to the root-cause of the problem. Then AUDIT enables heavy-weight logging on highly-ranked methods for a short period of time. Over a period of time, logs generated by a method is proportional to how often it is blamed for various misbehaviors, allowing developers to quickly find the root-cause of the problem.

We have implemented AUDIT for cloud applications. We describe how to utilize system events to efficiently implement lightweight triggers and blame ranking algorithm, with negligible to $< 1\%$ common-case runtime overheads on real applications. We evaluate AUDIT with five mature open source and commercial applications, for which AUDIT identified previously unknown issues causing slow responses, inconsistent outputs, and application crashes. All the issues were reported to developers, who have acknowledged or fixed them.

1 Introduction

Modern cloud applications are complex. Despite tremendous efforts on pre-production testing, it is common for applications to misbehave in production. Such misbehaviors range from failing to meet throughput or latency SLAs, throwing unexpected exceptions, or even crashing. When such problems occur, developers and operators most commonly rely on various runtime logs to troubleshoot and diagnose the problems.

Unfortunately, runtime logging involves an inherent tradeoff between logging sufficient detail to root-cause problems and logging less for lower overhead (see for instance [1, 2, 3, 4, 5]). Our experiments show (§6) that even for web applications that are not compute intensive, logging parameters and return values of all methods can increase latency and decrease throughput by up to 7%. Moreover, determining what to log is made harder by the fact that modern cloud and web applications involve multiple software components owned by different software developer teams. As a result, most logs generated today are irrelevant when root-causing problems [6].

To solve this problem, we make an important observation that many misbehaviors in production systems are *transiently-recurring*. As many frequent problems are found and fixed during initial phases of testing and deployment, we expect many problems in production systems to be rare and transient (the rarity makes it challenging to troubleshoot using sampling techniques [1, 2]). However, when they occur, they recur for a short amount of time for a variety of reasons, e.g., the user retrying a problematic request or a load-balancer taking some time to route around a performance problem (§2.2).[1]

Contributions. In this paper, we utilize the recurrence of these misbehaviors and present the design and implementation of AUDIT (AUtomatic Drilldown with Dynamic Instrumentation and Triggers): a *blame-proportional logging* system for troubleshooting transiently-recurrent problems in production systems. The basic idea is as follows. AUDIT uses lightweight triggers to detect problems. When a problem occurs, AUDIT automatically assigns a *blame rank* to methods in the application based on their likelihood of being rel-

[1] A notable exception to transient-recurrence are *Heisenbugs* which occur due to thread-interleaving or timing issues. AUDIT is not designed to troubleshoot these problems.

evant to the root-cause of the problem. Then AUDIT drills-down—it dynamically instruments highly-ranked methods to start heavy-weight logging on them until a user-specified amount of logs are collected. Over a period of time, logs generated by a method is proportional to how often the method is blamed for various misbehaviors and the overall logging is temporally correlated with the occurrence of misbehaviors. Developers analyze the logs *offline*, and thus AUDIT is complementary to existing techniques that help in interactive settings [7, 8].

We demonstrate the feasibility and benefits of AUDIT with the following contributions. First, AUDIT introduces *lightweight triggers* that continuously look for target misbehaviors. Developers can declaratively specify new triggers, describing target misbehaviors, the set of metrics to collect, and the duration for which to collect. The design of the trigger language is motivated by recent studies on misbehaving issues in production systems and when/where developers wish to log [3, 9, 10].

To evaluate the triggers and to blame-rank methods, AUDIT uses continuous end-to-end request tracing. To this end, our second contribution is *a novel tracing technique* for modern cloud applications built using Task Asynchronous Pattern (TAP), an increasingly popular way to write asynchrnous programs with sequential control flow and found in many languages including .NET languages, Java, JS/Node.js, Python, Scala, etc. AUDIT leverages system events at thread and method boundaries provided by existing TAP frameworks for monitoring and debugging purposes. AUDIT correlates these readily available events for lightweight end-to-end tracing. As a result, AUDIT introduces acceptable (from negligible to $< 1\%$) overhead in latency and throughput during normal operations. Note that AUDIT can also support non-TAP applications using known techniques based on instrumentation and metadata propagation [8, 11, 12] that are shown to have acceptable overheads in production systems.

Our third contribution is a novel ranking *algorithm that assigns blame scores to methods*. After a trigger fires, AUDIT uses the algorithm to identify high-ranked methods to initiate heavy-logging on them. AUDIT's blame ranking algorithm uses lessons from recent studies on where and what developers like to log for successful troubleshooting [3, 13]. It prioritizes methods where misbehavior originates (e.g., at a root exception that later causes a generic exception), that slow down requests, and that are causally related to misbehaving requests. It addresses key limitations of existing bottleneck analysis techniques that ignore critical path [14] or methods not on critical paths [15, 16, 17].

Our final contribution is an *evaluation* of AUDIT. We used AUDIT on a Microsoft production service and 4 popular open source applications. AUDIT uncovered

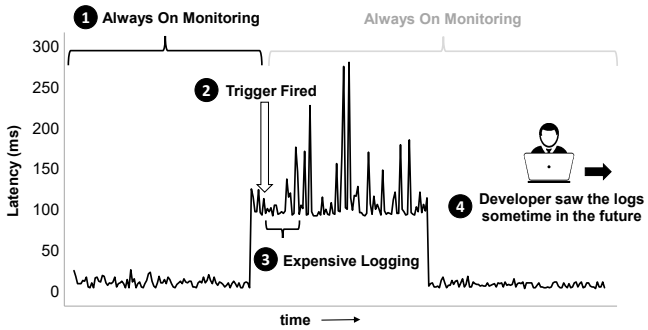


Figure 1: Timeline of AUDIT finding a performance bug in Forum.

previously-unseen issues in all the applications (§6.1). Many of the issues manifest only on production, as they are triggered based on user inputs and concurrency. All the issues have been reported to and acknowledged by developers of the applications. Some of them have already been fixed by developers with insights from logs generated by AUDIT.

2 Overview

2.1 A motivating case study

Microsoft Embedded Social. We start with an example demonstrating how AUDIT helped troubleshoot a problem in Microsoft Embedded Social (hereafter referred to as Social for brevity), a large-scale social service at Microsoft. Social is written in C#, deployed on Microsoft Azure, and is used by several applications and services in production. Social lets users add/like/search/delete posts, follow each other, and see feeds.[2]

Enabling AUDIT. AUDIT is easy to use. AUDIT works with unmodified application binaries and is enabled by simply setting a few environment variables. AUDIT comes with a set of triggers targeting common performance and exception-related problems. Social developers enabled AUDIT with these default triggers.

The problem. Figure 1 shows a performance problem that occurred in production: the latency of retrieving *global feeds* increased for a few hours. The developer was offline during the entire time and later relied on AUDIT logs to troubleshoot the issue.

AUDIT in operation. An AUDIT trigger fired shortly after the sudden spike in latency ((2) in Figure 1). For all misbehaving requests, AUDIT logged end-to-end *request trace* consisting of the request string, names and caller-callee relationship of executed methods. In addition, its blame ranking algorithm selected top-k (k = 5 by default) ranked methods and dynamically instrumented them to log their parameters and return values.

[2]Open source SDKs are available on GitHub, e.g., https://github.com/Microsoft/EmbeddedSocial-Java-API-Library

The *heavyweight logging* continues for a short time (5 minutes by default, stage (3) in Figure 1). This spatially- and temporally-selective logging helped the developer to root-cause of the problem, even long after the problem disappeared (stage (4) in Figure 1).

Troubleshooting with AUDIT logs. AUDIT's request traces showed that the misbehaving request was retrieving the global feed. The feed consists of a list of post-ids and contents; the latter is stored in a back-end store (Azure Table Storage) and is cached (in Redis) to achieve low-latency and high-throughput. Request tracing showed that the spike was due to post contents consistently missing the cache, albeit without revealing the cause of cache misses.

Among the methods AUDIT selected for heavyweight logging was a method that queries the backing store (Azure Table Storage). The logged arguments showed that the spike was caused by one particular post id, which according to logged return value didn't exist in the backing store. This inconsistency lead to the root-cause of the bug — a post id was present in the global feed but its contents were missing.

This inconsistency occurred when a user deleted a post. Social deleted its content from cache and backing store but failed to delete its id from the global feed due to a transient network failure. This inconsistency was not visible to users as missing posts are omitted in feeds, but it created a persistent performance spike. The inconsistency eventually disappeared when the problematic id was moved out of the global feed by other posts.

In addition to pinpointing the inconsistency, AUDIT also helped the developer root-cause the inconsistency to the failure in the delete operation through its detailed failure logging. We discuss the bug in more detail in §6.1.1. The developer chose to fix the problem by implementing negative caching, where Redis explicitly stores that a post is deleted from the backing store.

The case study demonstrates the value of AUDIT: it can capture *useful logs* for *relatively rare issues* that may appear in production or large-scale tests when the developer is absent. Moreover, logs are collected *only for a short period* after the issue occurs, reducing the log collection overhead and making it suitable even for expensive logging operations.

2.2 Transiently-recurring Problems

We argue that many problems in cloud-based production systems are transiently-recurring. First, if an error is frequent, it will most likely be detected and fixed during pre-production testing and staging of the application. Second, many problems are due to infrastructure problems such as transient network hardware issues [9, 18]; SLAs from cloud service providers ensure that such problems are rare and fixed within a short window of time. Therefore, cloud applications commonly use the "retry pattern" [19, 20] where the application transparently retries a failed operation to handle transient failures. Third, some problems are due to user inputs (e.g., malformed). Such errors are rare in well-tested production systems; however, once happened, they persist till the user gives up after several retries [21].

Note that AUDIT is also useful for troubleshooting errors that appear frequently—as long as they persist for a small window of time (e.g., not Heisenbugs).

3 AUDIT design

At a high level, AUDIT consists of four components: (1) declarative *triggers* for defining misbehaving conditions (§ 3.1), (2) a light-weight *always-on monitoring* component that continuously evaluates trigger conditions and collects request execution traces (§3.2 and § 4), (3) a *blame assignment algorithm* to rank methods based on their likelihood of being relevant to the root cause of a misbehavior (§ 3.3), and (4) a *selective logger* that uses dynamic instrumentation to enable and disable logging at top-blamed methods (§ 3.4).

3.1 AUDIT triggers

Trigger Language. AUDIT triggers are similar to *Event-Condition-Action* rules that are widely used in traditional databases [22] and in trigger-action programming such as IFTTT [23]. A key challenge in designing AUDIT's trigger language is to make it concise, yet expressive enough for a developer to specify interesting misbehaviors and useful logs. Before we elaborate on the rationale behind our choice, we first describe the four key components of an AUDIT trigger:

(1) `ON`**.** It specifies when (`RequestStart`, `RequestEnd`, `Exception`, or `Always`) the trigger is evaluated.

(2) `IF`**.** It describes a logical condition that is evaluated on the `ON` event. The condition consists of several useful properties of the request `r` or the exception `e` such as `r.Latency`, `e.Name`, `r.ResponseString`, `r.URL`, etc. It also supports several streaming aggregates: $\texttt{r.AvgLatency}(\texttt{now}, -\texttt{1min})$ is the average latency of request `r` in the last 1 min, $\texttt{e.Count}(\texttt{now}, -\texttt{2min})$ is the number of exception `e` in the last 2 mins, etc.

(3) `LOG`**.** It describes what to log when the `IF` condition satisfies. AUDIT supports logging `RequestActivity`[3] and `method` of a request. A key component of `LOG` is `ToLog`, which indicates target metrics to log: e.g., `args`, `retValue`, `exceptionName`, `latency`, `memoryDump`. Logs can be collected for requests matching (or not matching) the `IF` condition with a sampling probability of `MatchSamplingProb` (or `UnmatchSamplingProb`,

[3] A request activity graph (§3.3) consists of all methods invoked by the request as well as their causal relationship.

```
1 DEFINE TRIGGER T
2  ON RequestEnd R
3  IF R.URL LIKE 'http:*GetGlobalFeed*'
4     AND R.AvgLatency(-1min, now) > 2 * R.
          AvgLatency(-2min, -1min)
5  LOG RequsetActivity A, Top(5) Methods M
6     WITH M.ToLog=args, retValues
7     AND MatchSamplingProb = 1
8     AND UnmatchSamplingProb = 0.3
9  UNTIL (10 Match,10 Unmatch) OR 5 Minutes
```

Figure 2: An AUDIT trigger that fires when the latency of the global feed page in Social increases.

respectively). This enables comparing logs from "good" and "bad" requests. Finally, AUDIT supports logging all or a specified number of top performance-critical methods (with the `Top()` keyword). The later is useful when the request involves a large number of methods and instrumenting all of them would incur a high runtime overhead. Users can also define custom logging library that AUDIT can dynamically load and use.

(4) `UNTIL`**.** It describes how long or how many times the `LOG` action is performed.

Language Rationale. As mentioned, AUDIT's trigger language is motivated by prior works [3, 9, 10]. The general idea of enabling logging on specific misbehaving conditions (specified by `ON` and `IF`) and disabling it after some time (specified via `UNTIL`) addresses a key requirement highlighted in a recent survey of 54 experienced developers at Microsoft by Fu et. al [3]. The authors also analyzed two large production systems and identified three categories of *unexpected situation* logging. AUDIT's triggers support all of them: (1) exception logging, through `exceptionName` and `RequestActivity`, (2) return-value logging, via `retValue`, and (2) assertion-check logging, via `args`. The `ToLog` metrics are chosen to support common performance and reliability issues in production systems [9]. Logging both "good" and "bad" requests is inspired by statistical debugging techniques such as Holmes [10].

An Example Trigger. Figure 2 shows a trigger that can be used by Social for the scenario described in §2.1. The trigger fires when the average latency of the global feed page computed over a window of 1 minute increases significantly compared to the previous window. AUDIT starts logging all requests matching the `IF` condition and 30% of requests not matching the condition (for comparison) once the trigger fired. For each such request, AUDIT logs the *request activity*, consisting of all sync/async methods causally related to the request. Additionally, it assigns a blame rank to the methods and logs parameters and return values of 5 top-ranked methods. AUDIT continues logging for 10 matched and 10 unmatched requests, or for a maximum of 5 minutes.

Specifying Triggers. The trigger in Figure 2 may look overwhelming, with many predicates and parameters. We use this trigger for illustration purpose. In practice, a developer does not always need to specify all trigger parameters, letting AUDIT use their default values (all numerical values in Figure 2 are default values). Moreover, AUDIT comes with a set of predefined triggers that a developer can start with in order to catch exceptions and sudden spikes in latency and throughput. Over time, she can dynamically refine/remove existing triggers or install new triggers as she gains more operational insights. For example, the trigger in Figure 2 minus the predicate in Line 3 is a predefined trigger; Social developers modified its scope to global feed requests.

3.2 Always-on monitoring

AUDIT runtime continuously evaluates installed triggers. AUDIT instruments application binaries to get notified of triggering events such as exceptions, request start and end, etc. AUDIT automatically identifies instrumentation points for web and many cloud applications that have well-defined start and end methods for each request; AUDIT users can declaratively specify them for other types of applications. The handlers of the events track various request and exception properties supported by AUDIT trigger language. In addition, if needed by active triggers, AUDIT maintains lightweight streaming aggregates such as `Count`, `Sum`, and `AvgLatency` over a window of time.

In addition, AUDIT uses end-to-end causal tracing to continuously track identity and caller-callee relationships of methods executed by each request. For general applications, AUDIT uses existing tracing techniques based on instrumentation and metadata propagation [1, 8, 24, 25, 26, 27, 28]. For cloud applications using increasingly popular Task Asynchrnous Pattern (TAP), AUDIT uses a more lightweight and novel technique that we describe in §4.

AUDIT represents causal relationships of methods with a *request activity graph* (RAG), where nodes represent instances of executed methods and (synchronous or asynchrnous) edges represent caller-callee relationships of the nodes. A *call chain* to a node is the path from the root node to that node. (A call chain is analogous to a stack trace, except that it may contain methods from different threads and already completed methods.)

For multi-threaded applications, a RAG can contain two special types of nodes. A *fork node* invokes multiple asynchronous methods in parallel. A *join node* awaits and starts only after completion of the its nodes. A join node is an *all-join* node (or, *any-join* node), if it waits for all (or, any, respectively) of its parents node to complete. For each method in the RAG, AUDIT also tracks four timestamps: a (t_{start}, t_{end}) pair indicating when the

method starts and ends, and a ($t_{pwStart}$, t_{pwEnd}) pair indicating when the method's parent method starts and ends waiting for it to complete (more details in §4).

3.3 Blame assignment and ranking

After a misbehaving request fires a trigger, AUDIT uses a novel algorithm that ranks methods based on their blames for a misbehavior – the higher the blame of a method, the more likely it is responsible for the misbehavior. Thus, investigating the methods with higher blames are more likely to be helpful in troubleshooting the misbehavior.

To assign blames, AUDIT relies on RAGs and call chains of misbehaving requests, as tracked by the always-on monitoring component of AUDIT.

3.3.1 Exception-related triggers

On an exception-related trigger, AUDIT uses the call chain ending at the exception site to rank methods (on the same or different threads). Methods on the call chain are ranked based on their distance from the exception – the method that throws the exception has the highest rank and methods nearer to the exception are likely to contain more relevant information to troubleshoot root causes of the exception (as suggested by the survey in [3]).

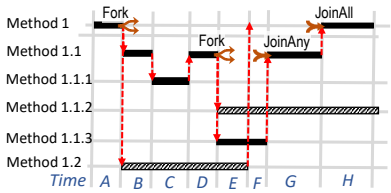
3.3.2 Performance-related triggers

On a performance-related trigger, AUDIT uses a novel bottleneck analysis technique on the RAGs of misbehaving requests. Existing critical path-based techniques (e.g., Slack [15], Logical Zeroing [16], virtual speedup [17]) fall short of our purpose because they ignore methods that should be logged but are not on a critical path or have very little exclusive run time on critical path. Techniques that ignore critical paths (e.g., NPT [14]) also miss critical methods that developers wish to log. §6 shows several real-world examples that illustrate these limitations.

Blame assignment. AUDIT addresses the above limitations with a new metric called *critical blame* that combines critical path, execution time distribution, and join-node types. Given a RAG, computation of critical blames of methods consists of two steps.

First, AUDIT identifies critical paths in the RAG. A critical path is computed recursively, starting from the last node of the RAG. Critical path to a node includes the node itself and (recursively computed) critical paths of (1) all parent non-join nodes, (2) longest parents of all-join nodes, and (3) shortest parents of any-join nodes. Each method in the critical path has the property that if its runs faster, total request latency goes down. See §5 for how these timestamps are derived.

Second, AUDIT assigns to each method on the critical path a *critical blame* score, a metric inspired by



Method	Blame
1	A+H/2
1.1	(B+D+G)/2
1.1.1	C/2
1.1.2	E/3+(F+G+H)/2
1.1.3	E/3+F/2
1.2	(B+C+D)/2+E/3

Figure 3: Critical blame assignment to methods. Solid edges represent methods on the critical path.

NPT[14]. Critical blame for a method consists of its exclusive and fair share of time on the critical path. Figure 3 illustrates how AUDIT computes critical blames of various methods in a RAG. Recall that each node in the RAG has four timestamps: a (t_{start}, t_{end}) pair and a ($t_{pwStart}$, t_{pwEnd}) pair. At a given time t, we consider a node to be *active* if t is within its t_{start} and t_{end} but not within any of its child method's $t_{pwStart}$ and t_{pwEnd}.

To compute critical blames of methods, AUDIT linearly scans the above timestamps of all methods (including the ones not in the critical path) in increasing order. Conceptually, this partitions the total request lifetime into a number of discrete *segments*, where each segment is bounded by two timestamps. In Figure 3, the segments are marked as $A, B, \ldots$ at the bottom. At each segment, AUDIT distributes the total duration of the segment to all methods active in that segment. For example, in the segment A, Method 1 is the only active method, and hence it gets the entire blame A. In segment B, methods 1.1 and 1.2 are active, and hence they both get a blame of $B/2$. Total blame of a method is the sum of all blames it gets in all segments (Method 1's total blame is $A + H/2$).

Selecting top methods. Given a target number n, AUDIT first selects the set $\mathbf{B}_1$ of n highest-blamed methods on the critical path. Let α be the lowest blame of methods in $\mathbf{B}_1$. AUDIT then compute another set $\mathbf{B}_2$ of methods not in the critical path whose execution times overlap with a method in $\mathbf{B}_1$, and whose blame scores are $\geq \alpha$. Finally, AUDIT computes $\mathbf{B} = \mathbf{B}_1 \cup \mathbf{B}_2$, and outputs all unique method names in $\mathbf{B}$. Essentially, the algorithm includes all slow critical methods and some slow non-critical methods that interfere with the critical methods.

Note that size of $\mathbf{B}$ can be larger (as it takes non-critical methods in $\mathbf{B}_2$) or smaller (as it ignores method instances) than n. If needed, AUDIT can try different sizes of $\mathbf{B}_1$ to produce a $\mathbf{B}$ whose size is close to n.

The intuition behind the above algorithm is as follow: (1) we want to blame only tasks that are actually running for the time they use; (2) we want co-running tasks to share the blame for a specific time period, assuming fixed amount of resources; (3) we want to first focus on tasks that are critical path as they affect runtime directly and (4) we want to include selective non-critical path tasks as they can be on the next longest path, may interfere

with tasks on the critical path, and not all critical path methods can be modified to run faster. §6.2 compares critical blame to other metrics quantitatively.

3.4 Enabling and disabling logging

AUDIT uses dynamic instrumentation to temporarily inject logging statements into blamed methods. The process works with unmodified applications and only requires setting few environment variables pointing to AUDIT library. Like Fay [7] and SystemTap [29], AUDIT supports instrumenting tracepoints at the entry, normal return, and exceptional exit of any methods running in the same address space as the application.

Specifically, AUDIT decorates each selected method with three callbacks. `OnBegin` is called as the first instruction of the method, with the current object and all arguments. It returns a local context that can be correlated at two other callbacks: `OnException`, called with the exception object, and `OnEnd`, called with the return value. These callbacks enable AUDIT to collect a variety of drilldown information. To log method parameters, global variables, or system parameters such as CPU usage, AUDIT uses `OnBegin`. To log return values, it uses `OnEnd`. Latency of a method is computed by taking timestamps at `OnBegin` and `OnEnd`. To collect memory dumps on exception, AUDIT uses `OnException`.

4 Optimizations for TAP applications

Task asynchronous pattern (TAP) is an increasingly popular programming pattern[4], especially in cloud applications that are typically async-heavy. Unlike traditional callback-based Asynchronous Programming Model (APM), TAP lets developer write non-blocking asynchronous programs using a syntax resembling synchronous programs. For example, TAP async functions can return values or throw exceptions to be used or caught by callers. This makes TAP intuitive and easier to debug, avoiding callback hell [30]. Major languages including .NET languages (C#, F#, VB), Java, Python, JavaScript, and Scala support TAP. In Microsoft Azure, for many services, TAP is provided as the *only* mechanism to do asynchronous I/O. Amazon AWS also provides TAP APIs for Java [31] and .NET [32].

One contribution of AUDIT is to show that for TAP applications, it is possible to construct RAG and call chains extremely efficiently, without extensive instrumentation or metadata propagation. Our techniques provide *intra-machine* RAG and call chains, where APIs of nodes may cross machine boundaries but edges are within the same machine. We focus only on such RAGs as we found them sufficient for our target cloud applications; if needed, inter-machine edges can be tracked by using the techniques used by Pivot Tracing [8].

[4]To quantify TAP's popularity, we statically analyzed all C# (total 18K), JavaScript (Node.js) (16K), and Java (Android) (15K) GitHub repositories created between 1/1/2017 and 6/30/2017. Our conservative analysis, which may miss applications using 3rd party TAP libraries, identified 52% of C#, 50% of JavaScript, and 15% of Java projects using TAP. The fractions are significantly higher than the previous year (e.g., 35% higher for C#), showing increasing popularity of TAP.

4.1 Continuous tracking of RAGs

AUDIT utilizes *async lifecycle events* provided by existing TAP frameworks for constructing RAGs. For debugging and profiling purpose, all existing TAP frameworks we know provide light-weight events or mechanisms indicating various stages of execution of async methods. Examples include ETW events in .NET [33], AsyncHooks [34] in Node.js, Decorators for Python AsyncIO [35], and RxJava Plugin [36] for Java. The events provide limited information about execution times and caller-callee relationships between some async methods, based on which AUDIT can construct RAGs. Using lifecycle events for tracing is not trivial. Depending on the platform, the lifecycle events may not directly provide all the information required to construct a RAG. We describe a concrete implementation for .NET in § 5.

4.2 On-demand construction of call chains

Even though call chain is a path in the RAG, AUDIT uses a separate mechanism to trace it for TAP applications. The advantage is that it lazily constructs a call chain on-demand, only *after* an exception-related trigger fires. Thus, the mechanism has *zero cost* during normal execution, unlike existing proactive tracking techniques [11, 12, 37]. AUDIT combines several mechanisms to achieve this.

AUDIT exception handler. AUDIT registers AUDIT event handler (AEH) to system events that are raised on all application exceptions. Examples of such events are *First Chance Exception* [38] for .NET and C++ for Windows, *UncaughtExceptionHandler* [39, 40] for Java, and *RejectionHandled* [41] for JavaScript.

AUDIT's exception tracing starts whenever the application throws an exception that satisfies a trigger condition. Consider `foo` synchronously calling `bar`, which throws an exception. This will invoke AEH with `bar` as the call site and a stacktrace at AEH will contain `foo`. This enables AUDIT to infer the RAG edge from `foo` to `bar`. If, however, `bar` runs asynchronously and in a different thread than `foo`, stacktrace won't contain `foo`. To infer the async edge from `foo` to `bar`, AUDIT relies on how existing TAP frameworks handle exceptions.

Exception propagation in TAP. Recall that TAP allows an async method to throw an exception that can be caught at its caller method. When an exception `e` is thrown in the async method `bar`, the framework first handles it and then revisits or rethrows *the same* exception object `e` when the caller method `foo` retrieves the result of

bar [42]. This action may trigger another first chance exception, calling the AEH with e.

AUDIT correlates on exception objects to discover async caller methods in a call chain and uses the order in which the AEHs are invoked in various methods to establish their order. In general, a call chain may contain a combination of synchronous and asynchronous edges. AUDIT uses stack traces to find small chains of consecutive synchronous edges, and correlates on exception objects to stitch the chains.

An application may catch one exception e1 and rethrow another exception e2. This pattern is dominant especially in middleware, where library developers hide low-level implementation details and expose higher level exceptions and error messages. The exception tracing technique described so far will produce two separate call chains, one for e1 and another for e2. However, since e1 has triggered e2, causally connecting the two chains can be useful for troubleshooting and root cause analysis [3].

Inheritable thread-local storage (ITS). AUDIT uses ITS to connect correlated exceptions. Inheritable thread-local storage allows storing thread-local contents that automatically propagate from a thread to its child threads. This is supported in Java (InheritableThreadLocal), .NET (LogicalCallContext), and Python (AsyncIO Task Local Storage[43]). Using ITS is expensive due to serialization and deserialization of data at thread boundaries. Existing causal tracing techniques use ITS all the time [27]; in contrast, AUDIT uses it only for exception tracing and on demand.

When e1 and e2 happens in the same thread, AUDIT can easily correlate them by storing a correlation id at the AEH of e1, and then using the id at the AEH of e2.

If e2, however, is thrown on a different thread than e1, the situation is more subtle. This is because e2 is thrown on the parent (or an ancestor) of e1's thread, and the correlation id stored in a thread's ITS is not copied *backward* to the parent thread's context (it is only copied forward to child threads).

To address this, AUDIT combines ITS with how TAP propagates exceptions across threads (described above). More specifically, AUDIT uses the first exception e1 as the correlation id and relies on TAP to propagate the id to the parent thread, which can correlate it to e1. The AEH for e2 stores e1 in ITS for further correlating it with other related exceptions on the same thread.

5 Implementation

We here describe our implementation of AUDIT for TAP applications written in .NET for Windows and cross-platform .NET Core.

Listening to exceptions. AUDIT listens to AppDomain.FirstChanceException to inspect all exceptions thrown by the application. First chance exception is a universal debugging concept (e.g., catch point in GDB, first chance exception in Visual Studio). A first chance exception notification is raised as soon as a runtime exception occurs, irrespective of whether it is later handled by the application.

Request tracing. For efficiently constructing the RAG of a request, AUDIT uses TplEtwProvider, an ETW-based [33] low overhead event logging infrastructure in .NET. TplEtwProvider generates events for the life-cycle of tasks in TAP.

Specifically, AUDIT uses TraceOperationBegin event to retrieve the name of a task. TaskWaitBegin is used for timestamp when a parent task transitions to suspended state and starts to wait on a child task. TraceOperationRelation is used to retrieve children tasks of a special join task (WhenAll, WhenAny), these join tasks are implemented in a special way such that they do not produce other life cycle events. At last, TraceOperationComplete, TaskWaitEnd, TaskCompleted, RunningContinuation, TaskWaitContinuationComplete are used to track the completion of a task. Many events are used because not all tasks generate the same event.

Constructing RAG based only on TPL ETW events is challenging for two key reasons, which AUDIT addresses by utilizing semantics of the events. First, ETW events are not timestamped by their source, but by the ETW framework after it receives the event. The timestamps are not accurate representation of the event generation times as the delivery from source to ETW framework can be delayed or out-of-order. To improve the quality of timestamps, for each method on the RAG, AUDIT aggregates multiple ETW events. For example, ideally, the t_{end} timestamp should come from the TaskCompleted ETW event. However, TPL generates other events immediately after a task completes. AUDIT takes the earliest of the timestamps of any and all of these events, to tolerate loss and delayed delivery of some events. AUDIT also uses the fact that in a method's lifetime, $t_{start} \geq t_{pwStart} \geq t_{end} \geq t_{pwEnd}$. Thus, if, e.g., all ETW events related to t_{start} are lost, it is set to $t_{pwStart}$.

Second, TPL does not produce any ETW events for join tasks, which are important parts of RAG. AUDIT uses reflection on the joining tasks (that produce ETW events) to identify join tasks, as well as their types (all-join or any-join). The t_{start} and t_{end} timestamps of a join task is assigned to the t_{start} and t_{end} timestamps of the shortest or the longest joining task, depending on whether the join task is any-join or all-join, respectively.

Dynamic instrumentation AUDIT uses .NET's profiling APIs to dynamically instrument target methods during runtime. The process is similar to dynamically instrumenting Java binaries [44].

6 Evaluation

We now present experimental results demonstrating:

1. AUDIT can effectively root-cause transiently recurring problems in production systems (§6.1).
2. AUDIT's blame ranking algorithm is more effective in root-causing than existing techniques (§6.2)
3. AUDIT has acceptably small runtime overhead for production systems, and its TAP-related optimizations further reduce the overhead (§6.3).

6.1 Effectiveness in root-causing bugs

We used AUDIT on five high-profile and mature .NET applications and identified root causes of several transiently recurring problems and bugs (Table 1). All the issues were previously unknown and are either acknowledged or fixed by developers. In all cases, AUDIT's ability to trigger heavyweight logging in a blame-proportional manner were essential to resolve problems.

6.1.1 Case study: Embedded Social

In § 2.1, we described one performance issue AUDIT found in Embedded Social (Social), a large-scale production social service in Microsoft. We now provide more details about Social and other issues AUDIT found in it. At the time of writing, Social had millions of users in production and beta deployments. We deployed Social in a deployment cluster. We enabled AUDIT with a generic exception trigger and a few performance triggers for latency-sensitive APIs (e.g. Figure 2).

Social 1: The persistent performance spike (Figure 1) arose because of an inconsistency caused by a failure (network timeout) during post deletion – the post id in the feed was left undeleted. Social swallowed the actual exception and produced only a high level exception for the entire delete operation. AUDIT logged the entire chain, pinpointed that post contents were deleted, but global feed deletion failed. AUDIT also logged the request URL, which identified the post id that was being deleted. The RAGs produced by the performance trigger showed the persistent store being consistently hit for one post. AUDIT's blame ranking algorithm top-ranked the persistent store query method, dynamically instrumented it, and logged arguments and return value for the next few requests to the global feed. The logged arguments showed the problematic post id causing the spike and the logged return value (`NULL`) indicated that it was deleted from the store and pointed to lack of negative caching as an issue. The post id matched the one logged during delete operation failure, which explained the bug.

Social 2: AUDIT revealed a few more transiently recurring issues related to lack of negative caching. For example, Social recommends a list of users with high follower count to follow. In the corner case of a popular user not following anyone, Social did not create an entity for the following count in the persistent store (and thus in the cache). In this case, the main page persistently missed the cache when reporting such users in the recommended list. AUDIT correctly assigned blame to the count-query method and logged both the user id (as part of parameters) and the return value of 0. Social's developers implemented negative caching to fix them.

Social 3: AUDIT's exception trigger in Social helped root-cause several transiently recurring request failures. We discuss a couple of them here. "Likes" for a post are aggregated and persisted to ATS using optimistic concurrency. When a specific post became hot, updates to ATS failed because of parallel requests. Through drill down, AUDIT pinpointed the post id (parameter) of the hot post and showed that like requests were failing only for that particular post id and succeeding for others.

Social 4: As posts are added, Social puts them in a queue and indexes the content of the posts in a backend worker. Typical to many systems, when a worker execution fails, the jobs are re-queued and retried a few times before being dead-lettered. This model perfectly fits AUDIT's triggered logging approach. After the first time a worker fails on a request, AUDIT triggers expensive parameter logging for subsequent retries. By logging their parameters, AUDIT root-caused many content-related bugs during indexing due to bad data formats.

We also found AUDIT useful in root-causing rare but recurrent problems in several open-source projects. Below we summarize the symptoms, AUDIT logs, and root cause of the problems.

6.1.2 Case study: MrCMS

MrCMS[45] is a content management system (CMS) based on the ASP.NET 5 MVC framework.

Symptoms. On a rare occasion, after an image is uploaded, the system crashed. Then the system became permanently unusable, even after restarting.

AUDIT logs. The AUDIT log from the first occurrence of the problem indicated an unhandled `PathTooLongException`. This was surprising because MrCMS checks for file name length. The methods on the call chain, however, indicated that the exception happened when MrCMS was creating thumbnail for the image. After AUDIT instrumented methods on the call chain, recurrence of the problem (i.e., recurrent crashing after restart) generated logs including method parameters. This included the actual file name for which a file system API was throwing the exception.

Root cause and fix. When image files are uploaded, MrCMS generates thumbnails with the image file name suffixed with dimensions. Thus, when an input file name is sufficiently long, the thumbnail file name can exceed the filesystem threshold which is unchecked and caused the crash. As most bugs in production systems, the fix for

Application	Issue	Root cause based on AUDIT log	Status from devs
Social 1	Performance spike when reading global feeds	Deleted operation failed to delete the post from global feeds	Fixed
Social 2	Poor performance reading user profiles with no following in "Popular users" feed	Lack of caching zero count value	Fixed
Social 3	Transient "Like" API failures	Concurrent likes on a hot post	Acknowledged, open
Social 4	Indexing failures	Bad data formats	Some of them fixed
MrCMS	Crash after image upload and subsequent restart of the application (Issue# 43)	Auto-generated thumbnail file name too long	Acknowledged, investigating
CMSFoundation	Failure to save edited image (Issue# 321)	Concurrent file edit and delete	Acknowledged, open
Massive	Slow request (Issue# 270)	Unoptimal use of Await	Fixed and closed
Nancy	Slow request (Issue# 2623)	Redundant Task method calls	Fixed and closed

Table 1: Summary of previously-unknown issues found by using AUDIT.

the bug once the root cause is known is simple: check file name lengths after adding the suffixes. The issue was acknowledged by the developer.

6.1.3 Case study: CMS-Foundation

CMS-Foundation[46] is a top-rated open source CMS with more than 60K installations worldwide.

Symptoms. When an admin saves after editing an image, they occasionally get a cryptic "Failed to get image properties: check that the image is not corrupt" message. The problem recurred as the admin retried the operation.

AUDIT logs. AUDIT log showed a crucial causality through two exception chains (as the application caught and rethrew exceptions) to the file being deleted while the admin was editing the image.

Root cause and fix. While the admin was editing the image, another admin deleted it, leading to a race condition. One way to fix this behavior is to use locking to prevent two admins from performing conflicting operations. The issue was acknowledged by the developers.

We now summarize two case studies demonstrating AUDIT's value in diagnosing performance problems.

6.1.4 Case study: Massive

Massive[47] is a dynamic MicroORM and a showcasing project for ASP .NET. Massive is popular and active on GitHub, with 1.6K stars and 330 forks.

Symptoms. Slow requests for certain inputs.

AUDIT logs. AUDIT produced RAG for the slow requests, as well as input parameters and return values of 5 top-ranked methods.

Root cause and fix. The top two methods ranked by AUDIT constituted 80% of the latency for some inputs. These methods query a backend database. Input parameters (i.e., query string) of the methods indicated that the method calls are independent (we confirmed this by looking at the code), yet Massive runs them in sequence. We modified the code to call both methods in parallel. This simple change resulted in a 1.37× speedup of the query in our deployment. We filed this potential optimization on GitHub and this issue was acknowledged and fixed.

6.1.5 Case study: Nancy

Nancy[48] is "a lightweight, low-ceremony, framework for building HTTP based services on .NET Framework/-Core and Mono". Nancy is also popular on GitHub, with 5.8K stars, 1.3K forks, and more than 250 contributors.

Symptoms. Some requests were slow.

AUDIT logs. AUDIT's log identified RAG and top-blamed method calls for the slow requests.

Root cause and fix. The top-blamed method calls, that constitued signficant part of the latency, were expensive and redundant [42]. We therefore changed the code by simply removing the redundant code, without affecting semantics of the code. This reduced average latency of the Nancy website from 1.73ms to 1.27ms with our deployment, a 1.36× improvement. We have reported this issue to Nancy developers, who have quickly acknowledged and fixed it. This, again, shows effectiveness of AUDIT's blame ranking algorithm.

6.2 Blame ranking algorithm

We compare AUDIT's blame ranking algorithm with three other algorithms: (1) NPT [14] that distributes running time evenly among concurrently running methods and ranks methods based on their total time, (2) Top critical methods (TCM), which ranks methods based on their execution time on critical path, and (3) Iterative Logical Zeroing (ILZ), an extension of Logical Zeroing [16]. ILZ first selects the method that, if finished in zero time, would have the maximum reduction in end-to-end latency. It then selects the second method *after* setting the first method's execution time to zero, and so on.

We consider four common code patterns observed in 11 different open source TAP applications and tutorials. Figure 4 shows corresponding RAGs. Two developers manually studied the applications and RAGs and identified the methods they would log to troubleshoot performance misbehaviors. Methods identified by both the developers are used as baseline.

Table 2 shows top-3 methods identified by different algorithms (and the baseline). TCM and ILZ fail to identify methods not on critical paths (e.g., Scenario 3). NPT

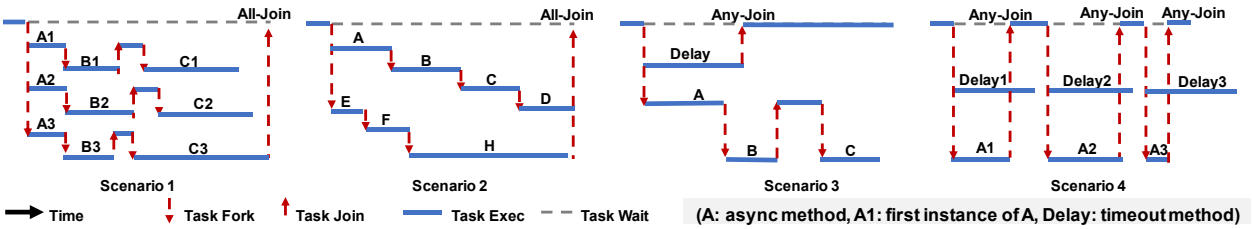


Figure 4: Common code patterns in TAP. Scenario 1 (found in Social, [49, 50]) starts parallel tasks with same code path and awaits all to finish. Scenario 2 (found in [51, 52]) starts several different tasks (which in turns fires up more children tasks) and they could finish close to each other. Scenario 3 (found in [53, 54]) starts a task and waits for a timeout. Scenario 4 (found in [20, 55, 56, 57]) retries a failed task a few times, and each trial is guarded with a timeout.

Algorithm	Scenario 1	Scenario 2	Scenario 3	Scenario 4	Total
Baseline	C_3, B_3, A_3	H, B, A	A, C, B	A_1, A_2, A_3	–
NPT	C_3, C_2, C_1	H, B, A	$A, \mathbb{D}, C$	$\mathbb{D}_3, \mathbb{D}_2, \mathbb{D}_1$	6/12
TCM	C_3, B_3, A_3	B, A, C	$\mathbb{D}$	A_2, A_1, A_3	7/12
ILZ	C_3, C_2, C_1	B, H, A	$\mathbb{D}$	A_2, A_1, A_3	7/12
AUDIT	C_3, B_3, A_3	H, B, A	$\mathbb{D}, A, B$	$A_1, A_2, \mathbb{D}_3$ $\mathbb{D}_1, \mathbb{D}_2, A_3$	11/12

Table 2: Top 3 blamed methods identified by various algorithms for scenarios in Figure 4. ($\mathbb{D}$ = *Delay*.)

fail to find important methods on the critical path (e.g., Scenario 4). Last column of the table shows how many of the developer-desired methods (baseline) are identified by different algorithms. Overall, AUDIT performs better – it identified 11 out of 12 methods marked by developers; while other algorithms identified 6-7 methods only. The only scenario where AUDIT failed to identify an important method *C* is Scenario 3, where *C* does neither fall on a critical path nor overlap or interfere with any method on the critical path.

6.3 Runtime overhead

We now evaluate runtime overhead of AUDIT running on a Windows 10 D8S_V3 instance, on Microsoft Azure. Web applications are hosted using ASP.NET 5 on IIS 10.0. SQL Server 2016 is used as database.

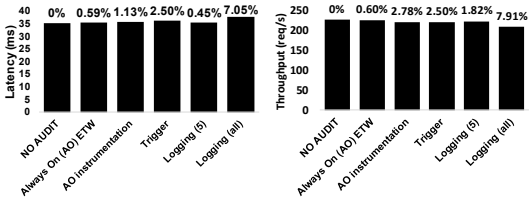
6.3.1 Micro benchmark results

Considerable design effort went in reducing the always-on overhead of AUDIT. We measure the overhead with a simple benchmark application that waits on two consecutive tasks. To measure the overhead of AUDIT's exception handling mechanism, we modified the application such that the async task throw an exception that the main task catches. Finally, to measure the *lower bound* on the cost of dynamic instrumentation, we instrumented an empty callback at the beginning and the end of each function with no parameter.

Table 3 shows AUDIT overhead numbers averaged over 100k runs. As shown, AUDIT's always-on ETW monitoring incurs small overhead – tens of μs per task.

	Without Exception	With Exception
Always-On ETW	15.56μs	112.2μs
Overhead	**+13.96μs/task**	**+19.2μs/task**
Always-On INST	91.5μs	152μs
Overhead	**+89.9μs/method**	**+59μs/method**
Trigger	29.66μs	283μs
Overhead	**+28.06μs/task**	**+190μs/task**
Logging	93.5μs	148μs
Overhead	**+90.9μs/method**	**+55μs/method**

Table 3: AUDIT overhead on benchmark application.



(a) Latency (b) Throughput

Figure 5: AUDIT overhead for Massive.

The overhead is acceptable for modern-day cloud applications that contain either compute-intensive or I/O-intensive tasks that typically run for tens of milliseconds or more. Always-on monitoring with instrumentation (Always-On INST) and metadata propagation incurs higher overhead mainly from instrumentation cost.[5] AUDIT significant lowers always-on monitoring overhead by leveraging ETW in TAP applications. The overhead is also higher immediately after a trigger fires (for constructing RAG and computing blames). This cost is acceptable as triggers are fired infrequently. Finally, logging has the highest overhead. Even an empty callback incurs hundreds of μs; serializing and logging method parameters, return values, stacktrace, etc. and writing to storage will add more overhead. This overhead clearly motivates the need for blame-proportional logging, which limits the number of logging methods and the duration of logging.

[5] Our measurement shows accessing an integer from ITS takes about 100ns and propagating an integer across thread costs 800ns, with a base cost of 700ns

6.3.2 Overheads for real applications

We measured AUDIT's overhead on Massive and Social, two TAP applications we used in our case studies. To emulate AUDIT's overhead on non-TAP applications in the always-on monitoring phase, we use AUDIT with and without its TAP optimizations (§4). We report maximum throughput and average query latency at maximum throughput over 5000 requests. We use a trigger to fire when latency is $2\times$ the average latency (over 1 minute) and to log method parameters and return values.

Figure 5 shows the results for Massive, with a reasonably complex request that comes with Massive, involving 55 method invocations. Without TAP-optimizations, AUDIT always-on monitoring increases latency by 1.1% and reduces throughput by 2.8%. The overhead is smaller for simpler requests (with fewer methods) and is acceptable in many non-TAP applications in production. The overhead is significantly smaller with TAP-optimizations: latency and throughput are affected only by $< 0.6\%$, showing effectiveness of the optimizations.

Overhead of the trigger phase is slightly larger ($+2.5\%$ latency and -2.5% throughput). Logging all methods decreases throughput by 8% and increases latency by 7%. The high overhead is mainly due to serializing method parameters and return values of 55 dynamically invoked methods. Logging at only five top-blamed methods, however, has much smaller overhead (-0.45% latency and -1.8% throughput). This again highlights the value of logging only for a short period of time, and only a small number of top methods.

For Social, we used a complex request involving 795 method invocations. With TAP optimizations, latency and throughput overheads of always-on phase is within the measurement noise ($< 0.1\%$). Without the optimizations, the overhead of always-on is 4.3%, due to instrumentation overhead of 795 method invocations. Trigger phase incurs 4.1% overhead. Logging, again is the most expensive phase, causing 5.3% overhead.

7 Related work

In previous sections, we discussed prior work related to AUDIT's triggers (§3.1), request tracing (§4), dynamic instrumentation (§3.4), and blame ranking (§3.3). We now discuss additional related work.

AUDIT triggers are in spirit similar to datacenter network-related triggers used in Trumpet [58], but are designed for logging cloud and web applications.

Collecting effective logs and reducing logging overhead have been an important topic of research. Errlog [2] proactively adds appropriate logging statements into source code and uses adaptive sampling to reduce runtime overhead. In contrast, AUDIT dynamically instruments unmodified application binary and uses triggers rather than sampling to decide when to log. Log^2 [4] enables logging within an overhead budget. Unlike AUDIT, it uses static instrumentation, continuous logging, and decides only *whether* (not *what*) to log. Several recent works investigate what should be logged for effective troubleshooting [3, 13], and AUDIT incorporates their findings in its design. Several recent proposals enhance and analyze *existing* log messages for failure diagnosis [59, 60, 61, 62], and are orthogonal to AUDIT.

Pivot Tracing [8] is closely related, but complimentary to AUDIT. It gives users, at runtime, the ability to define arbitrary metrics and aggregate them using relational operators. Unlike AUDIT, Pivot Tracing requires users to explicitly specify tracepoints to instrument and to interactively enable and disable instrumentation. Techniques from Pivot Tracing could be used to further enhance AUDIT; e.g., if implemented, happen-before join could be used as a trigger condition and baggage could be used to trace related methods across machine boundaries.

AUDIT's techniques for identifying methods related to a misbehaving request is related to end-to-end causal tracing [1, 24, 25, 26, 27, 28]. Existing solutions use instrumentation and metadata propagation; in contrast, AUDIT can also leverage cheap system events. To keep overhead acceptable in production, prior works trace coarse-grained tracepoints [1, 24], or fine-grained but a small number of carefully chosen tracepoints (which requires deep application knowledge) [26], and/or a small sample of requests [1]. In contrast, AUDIT traces all requests at method granularity, along with forks and joins of their execution.

Adaptive bug isolation [63], like AUDIT, adapts instrumentation during runtime. However, AUDIT's adaptation can be triggered by a single request (rather than statistical analysis of many requests, as in many other statistical debugging techniques [10, 64]), can work at a much finer temporal granularity (logging only for a small window of time), and has much better selectivity of logging methods due to causal tracking.

8 Conclusions

We presented AUDIT, a system for troubleshooting transiently-recurring errors in cloud-based production systems through *blame-proportional logging*, a novel mechanism with which logging information generated by a method over a period of time is proportional to how often it is blamed for various misbehaviors. AUDIT lets a developer write declarative triggers, specifying what to log and on what misbehavior, without specifying where to collect the logs. We have implemented AUDIT and evaluated it with five mature open source and commercial applications, for which AUDIT identified previously unknown issues causing slow responses and application crashes. All the issues are reported to developers, who have acknowledged or fixed them.

References

[1] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Technical report, Google, 2010.

[2] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael M. Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. Be conservative: Enhancing failure diagnosis with proactive logging. In *Presented as part of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 293–306, Hollywood, CA, 2012. USENIX.

[3] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. Where do developers log? an empirical study on logging practices in industry. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 24–33. ACM, 2014.

[4] Rui Ding, Hucheng Zhou, Jian-Guang Lou, Hongyu Zhang, Qingwei Lin, Qiang Fu, Dongmei Zhang, and Tao Xie. Log2: A cost-aware logging mechanism for performance diagnosis. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 139–150, Santa Clara, CA, 2015. USENIX Association.

[5] Gerd Zellweger, Denny Lin, and Timothy Roscoe. So many performance events, so little time. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 14:1–14:9, New York, NY, USA, 2016. ACM.

[6] Rui Ding, Qiang Fu, Jian-Guang Lou, Qingwei Lin, Dongmei Zhang, and Tao Xie. Mining historical issue repositories to heal large-scale online service systems. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2014, Atlanta, GA, USA, June 23-26, 2014*, pages 311–322, 2014.

[7] Úlfar Erlingsson, Marcus Peinado, Simon Peter, Mihai Budiu, and Gloria Mainar-Ruiz. Fay: extensible distributed tracing from kernels to clusters. *ACM Transactions on Computer Systems (TOCS)*, 30(4):13, 2012.

[8] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. Pivot tracing: Dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 378–393, New York, NY, USA, 2015. ACM.

[9] Haryadi S Gunawi, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, Thanh Do, Jeffry Adityatama, Kurnia J Eliazar, Agung Laksono, Jeffrey F Lukman, Vincentius Martin, et al. What bugs live in the cloud? a study of 3000+ issues in cloud systems. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[10] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. Holmes: Effective statistical debugging via efficient path profiling. In *Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on*, pages 34–44. IEEE, 2009.

[11] Async stacktraceex 1.0.1.1. https://www.nuget.org/packages/AsyncStackTraceEx/, 2017.

[12] Async programming async causality chain tracking. https://msdn.microsoft.com/en-us/magazine/jj891052.aspx, 2017.

[13] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Learning to log: Helping developers make informed logging decisions. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 415–425. IEEE, 2015.

[14] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. In *Acm Sigmetrics Conference on Measurement & Modeling of Computer Systems*, pages 115–125, 1990.

[15] Jeffrey K. Hollingsworth and Barton P. Miller. Slack: A new performance metric for parallel programs. *University of Wisconsin-Madison Department of Computer Sciences*, 1970.

[16] B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. S. Lim, and T. Torzewski. Ips-2: The second generation of a parallel program measurement system. *Parallel & Distributed Systems IEEE Transactions on*, 1(2):206–217, 1990.

[17] Charlie Curtsinger and Emery D. Berger. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 184–197, New York, NY, USA, 2015. ACM.

[18] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. *SIGARCH Comput. Archit. News*, 44(2):517–530, March 2016.

[19] Retry pattern. https://docs.microsoft.com/en-us/azure/architecture/patterns/retry, 2017.

[20] Cleanest way to write retry logic? https://stackoverflow.com/questions/1563191/cleanest-way-to-write-retry-logic.

[21] Matthew Merzbacher and Dan Patterson. Measuring end-user availability on the web: Practical experience. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 473–477. IEEE, 2002.

[22] Jennifer Widom and Stefano Ceri. *Active database systems: Triggers and rules for advanced database processing*. Morgan Kaufmann, 1996.

[23] Ifttt. http://ifttt.com.

[24] Rodrigo Fonseca, George Porter, Randy H Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, pages 20–20. USENIX Association, 2007.

[25] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu, and Gregory R. Ganger. Diagnosing performance changes by comparing request flows. In *Proceedings of the USENIX Conference on Networked Systems Design and Implementation*, NSDI '11, 2011.

[26] Eno Thereska, Brandon Salmon, John Strunk, Matthew Wachs, Michael Abd-El-Malek, Julio Lopez, and Gregory R Ganger. Stardust: tracking activity in a distributed storage system. In *ACM SIGMETRICS Performance Evaluation Review*, volume 34, pages 3–14. ACM, 2006.

[27] Raja R Sambasivan, Rodrigo Fonseca, Ilari Shafer, and Gregory R Ganger. So, you want to trace your distributed system? key design s from years of practical experience. Technical report, Technical Report, CMU-PDL-14, 2014.

[28] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The Good, the Bad, and the Differences: Better Network Diagnostics with Differential Provenance. In *Proceedings of ACM SIGCOMM 2016*, August 2016.

[29] Vara Prasad, William Cohen, FC Eigler, Martin Hunt, Jim Keniston, and J Chen. Locating system problems using dynamic instrumentation. In *2005 Ottawa Linux Symposium*. Citeseer, 2005.

[30] What is callback hell? https://www.quora.com/What-is-callback-hell,, 2017.

[31] Asynchronous programming with the aws sdk for java. https://aws.amazon.com/articles/5496117154196801.

[32] Amazon web services asynchronous apis for .net. http://docs.aws.amazon.com/sdk-for-net/v3/developer-guide/sdk-net-async-api.html.

[33] Etw tracing. https://msdn.microsoft.com/en-us/library/ms751538.aspx.

[34] Nodejs v8.4.0 documentation. https://nodejs.org/api/async_hooks.html, 2017.

[35] Asyncio decorator. https://gist.github.com/Integralist/77d73b2380e4645b564c28c53fae71fb#file-python-asyncio-timing-decorator-py-L28, 2017.

[36] Rxjava debug plugin. https://github.com/ReactiveX/RxJavaDebug.

[37] Long stacktraces. https://github.com/tlrobinson/long-stack-traces,, 2017.

[38] Appdomain.firstchanceexception event. https://msdn.microsoft.com/en-us/library/system.appdomain.firstchanceexception(v=vs.110).aspx, 2017.

[39] Thread.uncaughtexceptionhandler (java platform se 7) - oracle. http://docs.oracle.com/javase/7/docs/api/java/lang/Thread.UncaughtExceptionHandler.html.

[40] Detect and log all java security exceptions. https://freckles.blob.core.windows.net/sites/marketing/media/assets/partners/brixbits/securityanalyzer_datasheet_2015_02.pdf.

[41] rejectionhandled event reference — mdn. https://developer.mozilla.org/en-US/docs/Web/Events/rejectionhandled.

[42] .net reference source. https://referencesource.microsoft.com/, 2017.

[43] tasklocals 0.2. https://pypi.python.org/pypi/tasklocals/.

[44] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[45] Mrcms/mrcms. https://github.com/MrCMS/MrCMS, 2017.

[46] Orckestra/cms-foundation. https://github.com/Orckestra/CMS-Foundation, 2017.

[47] Massive. https://github.com/FransBouma/Massive,, 2017.

[48] Nancy. https://github.com/NancyFx/Nancy, 2017.

[49] Codehub issueview.cs. https://github.com/CodeHubApp/CodeHub/blob/ee9b2acacab461730cf946836f5dff149908f8ad/CodeHub.iOS/Views/Issues/IssueView.cs#L276.

[50] Blazor/src/anglesharp/extensions/documentextensions.cs. https://github.com/SteveSanderson/Blazor/blob/749bae9def3ccb57006da1b155e46ea3e4c62c0f/src/AngleSharp/Extensions/DocumentExtensions.cs#L261.

[51] eshoponcontainer. https://github.com/dotnet-architecture/eShopOnContainers/search?utf8=%E2%9C%93&q=WhenAll&type=.

[52] Massive.shared.async.cs. https://github.com/FransBouma/Massive/blob/d8135f6ed44f36418ab9ee78f9cb14e023778d30/src/Massive.Shared.Async.cs#L862.

[53] Codehub whenany example. https://github.com/CodeHubApp/CodeHub/blob/e8513b052ba34ab54b7d08e08adbc4dbd3ceeac1/CodeHub.iOS/Services/InAppPurchaseService.cs#L57.

[54] Reactivewindows whenany example. https://github.com/Microsoft/react-native-windows/search?utf8=%E2%9C%93&q=whenany&type.

[55] Polly. https://github.com/App-vNext/Polly.

[56] Implementing the retry pattern for async tasks in c#. https://alastaircrabtree.com/implementing-the-retry-pattern-for-async-tasks-in-c/.

[57] Retry a task multiple times based on user input in case of an exception in task. https://stackoverflow.com/questions/10490307/retry-a-task-multiple-times-based-on-user-input-in-case-of-an

[58] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. Trumpet: Timely and precise triggers in data centers. In *Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference*, pages 129–143. ACM, 2016.

[59] Xiao Yu, Pallavi Joshi, Jianwu Xu, Guoliang Jin, Hui Zhang, and Guofei Jiang. Cloudseer: Workflow monitoring of cloud infrastructures via interleaved logs. *ACM SIGOPS Operating Systems Review*, 50(2):489–502, 2016.

[60] Ding Yuan, Haohui Mai, Weiwei Xiong, Lin Tan, Yuanyuan Zhou, and Shankar Pasupathy. Sherlog: error diagnosis by connecting clues from run-time logs. In *ACM SIGARCH computer architecture news*, volume 38, pages 143–154. ACM, 2010.

[61] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. Improving software diagnosability via log enhancement. *ACM Transactions on Computer Systems (TOCS)*, 30(1):4, 2012.

[62] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132. ACM, 2009.

[63] Piramanayagam Arumuga Nainar and Ben Liblit. Adaptive bug isolation. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 255–264. ACM, 2010.

[64] Linhai Song and Shan Lu. Statistical debugging for real-world performance problems. *SIGPLAN Not.*, 49(10):561–578, October 2014.