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

Liang Luo, Suman Nath, Lenin Ravindranath Sivalingam, Madan Musuvathi and Luis Ceze
University of Washington, Microsoft Research
Cloud applications are complex

- Many layers
- 3rd-party components
- Shared environments

Despite tremendous effort of testing, SLO violations, exceptions, and crashes
DevOps usually turn to logging for help when problems occur.

Logging is the most commonly used technique for troubleshooting, but logging itself is very hard to done right!
Logging has an inherent trade off between utility and overhead:

Two extremes in production and testing:

**Production**: collects little data for higher runtime performance; however log utility is low as most logs generated are irrelevant when root-causing problems.

**Testing**: collects everything for maximum utility and ignores runtime overhead.
AUDIT: AUtomatic Drilldown with Dynamic Instrumentation and Trigger

AUDIT Strategy: Get the best of both world with Dynamic Logging.
When to log?
Right after a problem occurs. Problems are detected using developer specified triggers.

Key insight: Many problems in cloud applications are transiently recurring -- they occur rarely, but when they do, they recur for a short period of time.

Can start logging when they show up, and when they recur, detailed info can be collected.

Examples: network hardware issues, malformed user inputs, load balancer taking time to tick in, neck and back pain
When to log?

Where to Log?
Highly blamed methods that are causally related to the misbehaving request.

Requires: AUDIT uses Continuous Causal Tracing to track methods that are causally related to misbehaving request.

Requires: AUDIT uses novel Critical Blame metric to select highly-blamed methods to log, as root cause usually involves a small set of methods.
When to log?

Where to Log?

**Dynamically Turning Logging On/Off?**

*Use dynamic instrumentation to log only what is specified in triggers.*
AUDIT in action

1. Always On Monitoring

2. Problem detected

3. Detail logging for a small time window, at few places

4. Developer saw the logs sometime in the future
AUDIT Key Mechanisms

When to log?

Where to Log?

Dynamically Turning Logging On/Off?

End product:
A “push button” tool: no knowledge of or changes to code, activate only by setting environment variable.

Efficient: <1% overhead during normal operation.

Effective: found 8 unforeseen bugs in 5 production systems.
Outline

When to log?
   AUDIT Triggers

Where to Log?
   Causal tracing
   Blame ranking

Evaluation
   Case studies
   Micro benchmark results
Defining what it means for application to “go wrong”. Triggers contains 4 components.

**ON:** when is the trigger evaluated?

**IF:** on what condition is the trigger activated?

**LOG:** what to do when the trigger is activated?

**UNTIL:** when is the logging deactivated?

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** RequestActivity A, Top(5) Methods M
6. \[ WITH M.ToLog=\text{args}, \text{retValues} \]
7. \[ AND \text{MatchSamplingProb} = 1 \]
8. \[ AND \text{UnmatchSamplingProb} = 0.3 \]
9. **UNTIL** (10 Match,10 Unmatch) OR 5 Minutes
Trigger language is motivated by recent surveys on how developers log and what logging is useful. (See paper for details)
- Logging for **both bad and good requests** help differential analysis
- Provides **streaming aggregates** of performance metrics to be used with triggers

Uses **dynamic instrumentation** to flexibly collect data required for trigger evaluation (more on this later)
Outline

When to log?
   AUDIT Triggers

Where to Log?
   Causal tracing (Only methods related to misbehaving requests)
   Blame ranking

Evaluation
   Case studies
   Overhead
AUDIT reconstructs request activity graph (RAG), all methods that are casually related to a request. AUDIT assumes requests are independent of each other.
AUDIT reconstructs asynchronous exception chain (AEC). A call chain consists of all methods from root to the exception site. A chain differs from stacktrace such that it can contain already finished methods.
AUDIT can use existing causal tracing techniques for reconstructing RAG and AEC: Dynamic Instrumentation, Thread Local Storage, and Metadata Propagation.

High runtime overhead: 8% for just continuous causal tracing, which is required for trigger evaluation (when trigger fires we need to know what methods lead to it).

Needs optimizations!
Task Asynchronous Pattern is an emerging pattern that allows writing asynchronous code in a synchronous way, using the idea of continuation.

TAP is supported in many platforms natively or via libraries.

TAP is supported in all major cloud providers.
**RAG**: AUDIT utilizes async lifecycle events by existing TAP frameworks to piece together a RAG without dynamic Instrumentation.

**AEC**: AUDIT utilizes first chance exception, global exception handler, inheritable thread-local storage to passively reconstruct the exception call chain. AUDIT’s AEC construction incurs zero overhead during normal execution.
Outline

When to log?
AUDIT Triggers

Where to Log?
Causal tracing (Only methods related to misbehaving requests)
Blame ranking (Select top-blamed methods as RAGs and AECs can be big)

Evaluation
Case studies
Overhead
Methods that are closer to the exception are more likely related to the root cause.
Critical blame combines critical path analysis and normalized processor time.

- Blame only tasks that are running (versus waiting)
- Co-running tasks share the blame for the time period
- Focus on task on the critical path
- Include selective non-critical path as they may interfere with critical path methods
Critical Blame: selecting top N=2 methods

<table>
<thead>
<tr>
<th>Task</th>
<th>Blame</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task1.1</td>
<td>(B+D+G)/2</td>
</tr>
<tr>
<td>Task1</td>
<td>A+H/2</td>
</tr>
<tr>
<td>Task1.2</td>
<td>E/3+(F+G+H)/2</td>
</tr>
<tr>
<td>Task1.2</td>
<td>(B+C+D)/2+E/3</td>
</tr>
<tr>
<td>Task1.3</td>
<td>(E+F)/2</td>
</tr>
<tr>
<td>Task1.1</td>
<td>C/2</td>
</tr>
<tr>
<td>Task1.2</td>
<td></td>
</tr>
<tr>
<td>Task1.3</td>
<td></td>
</tr>
</tbody>
</table>

Diagram:

- Task1
  - Task1.1: (B+D+G)/2
  - Task1.2: E/3+(F+G+H)/2
  - Task1.3: (E+F)/2
- Task1.1: (B+D+G)/2
- Task1.2: (B+C+D)/2+E/3
Outline

When to log?
  AUDIT Triggers

Where to Log?
  Causal tracing (Only methods related to misbehaving requests)
  Blame ranking (Select top-blamed methods)

Evaluation
  Case studies
  Overhead
We implemented AUDIT for .NET and applied it to 1 production system at Microsoft and 4 high-profile, open source libraries in GitHub.

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

AUDIT can pinpoint performance issues, such as:
- lack of negative caching
- bad content format
- missed parallel opportunity
- redundant method calls
- contention with optimistic concurrency
We implemented AUDIT for .NET and applied it to 1 production system at Microsoft and 4 high-profile, open source libraries in GitHub.

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

AUDIT can pinpoint exception issues, such as...
4 typical code patterns we found in various cloud app projects, including issuing multiple tasks that shares the same path, concurrent parallel tasks, timeout-ed task, and retry tasks.

AUDIT is more sensitive than Normalized Processor Time, Top Critical Methods, and Iterative Logical Zeroing in locating bottlenecks.
AUDIT Overhead: negligible for real applications

<table>
<thead>
<tr>
<th>Raw Overhead</th>
<th>Without Exception</th>
<th>With Exception</th>
</tr>
</thead>
<tbody>
<tr>
<td>Always-On ETW Overhead</td>
<td>15.56μs +13.96μs/task</td>
<td>112.2μs +19.2μs/task</td>
</tr>
<tr>
<td>Always-On INST Overhead</td>
<td>91.5μs +89.9μs/method</td>
<td>152μs +59μs/method</td>
</tr>
<tr>
<td>Trigger Overhead</td>
<td>29.66μs +28.06μs/task</td>
<td>283μs +190μs/task</td>
</tr>
<tr>
<td>Logging Overhead</td>
<td>93.5μs +90.9μs/method</td>
<td>148μs +55μs/method</td>
</tr>
</tbody>
</table>

Real Application (Massive) Overhead

Throughput (req/s)

Latency (ms)
- Troubleshooting transiently-recurring errors
- Blame-proportional logging
- Provide declarative trigger language
- Negligible overhead
- Found 8 new unforeseen bugs