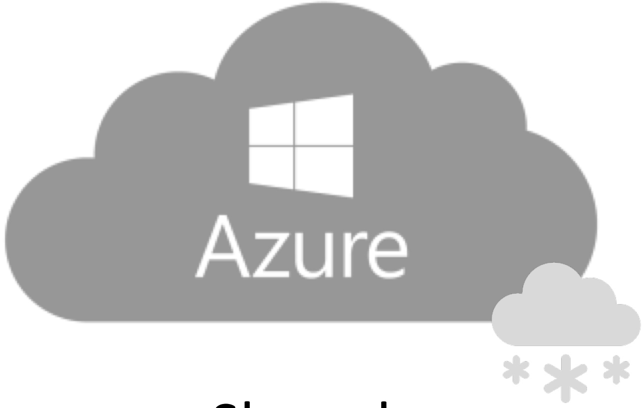
AUDIT: Troubleshooting Transiently-Recurring Problems in Production Systems with Blame-Proportional Logging

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

Cloud applications are **complex**

Many layers

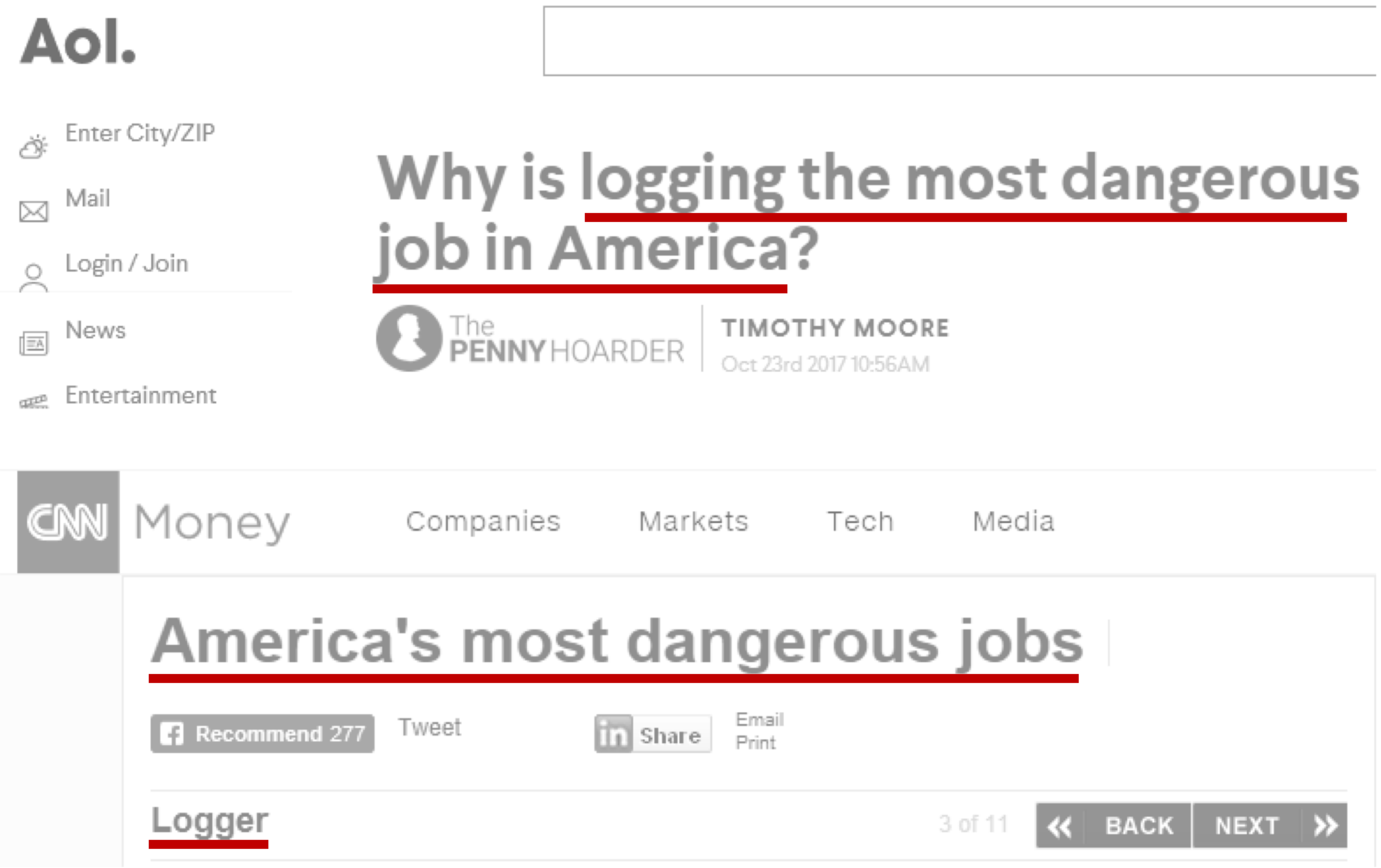
3rd-party components

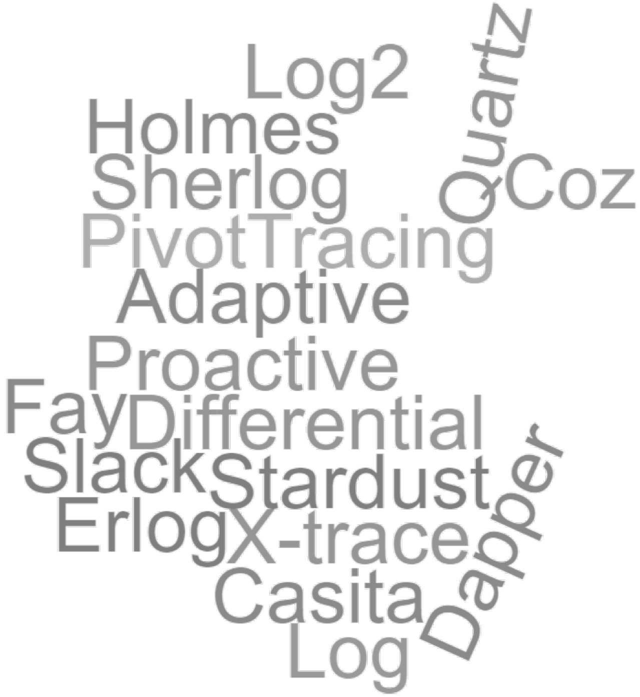


Shared environments

Despite tremendous effort of testing, SLO violations, exceptions, and crashes

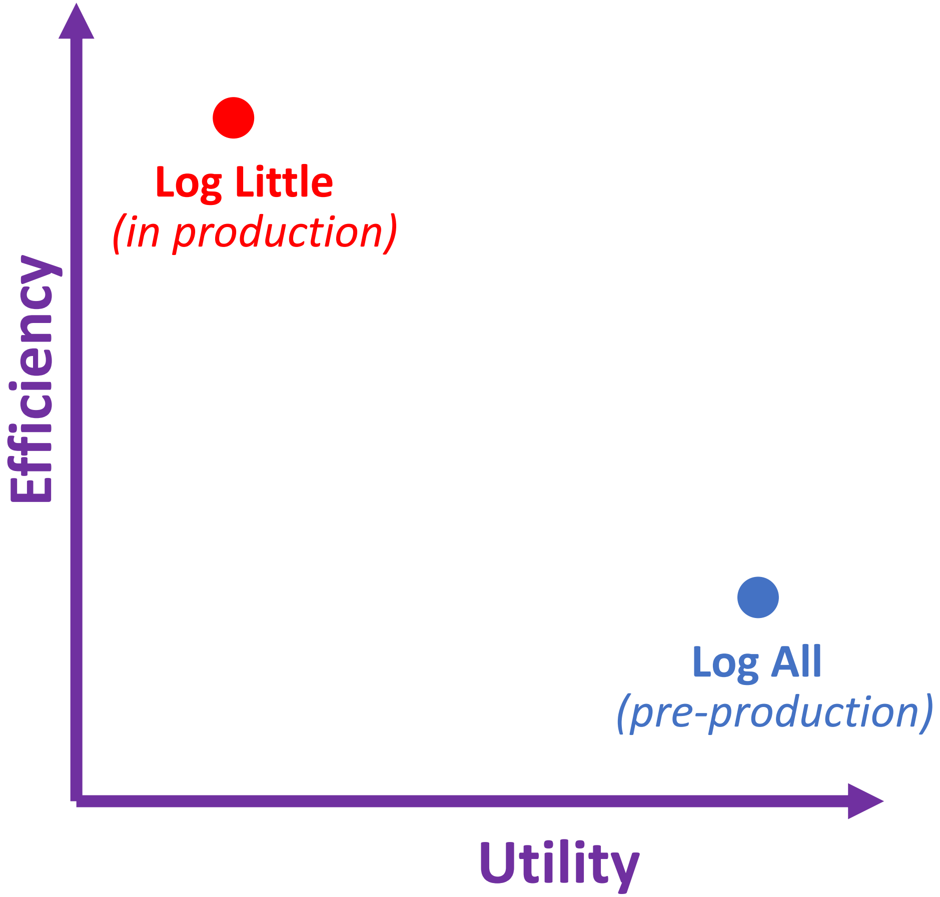
DevOps usually turn to logging for help when problems occur





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:

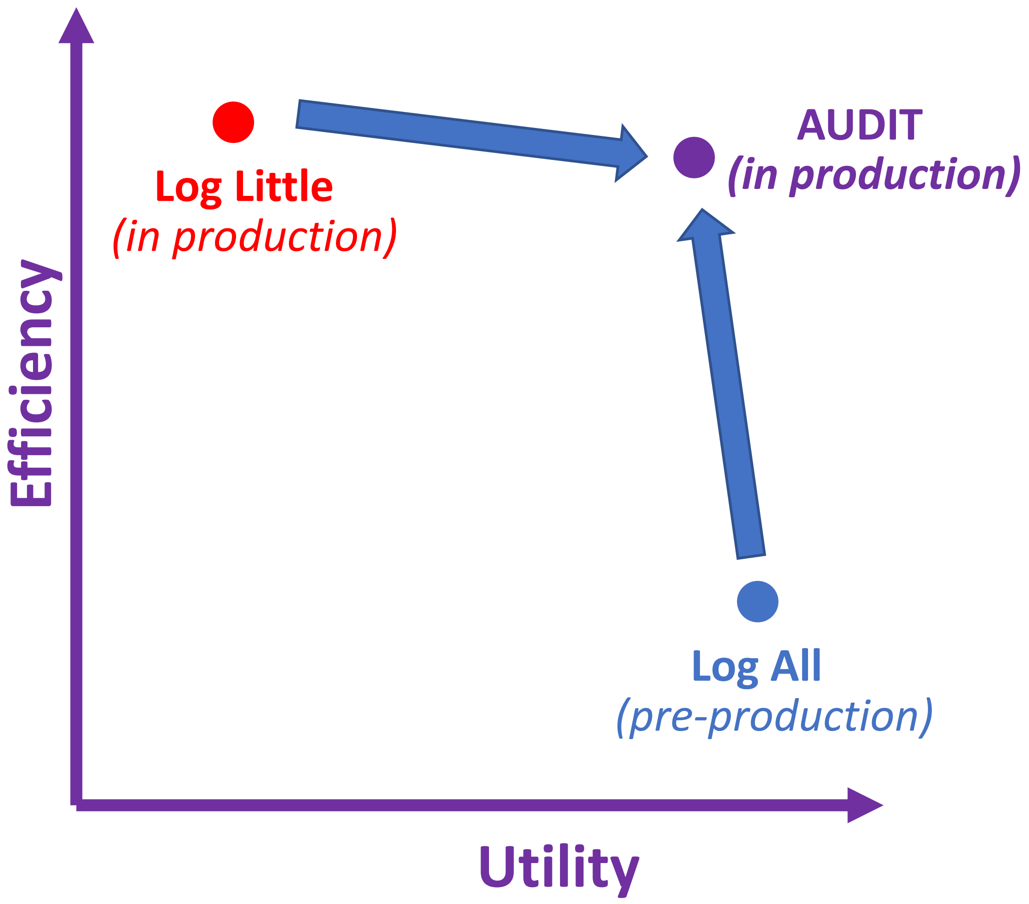


Two extremes in production and testing:

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.

AUDIT Challenges

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

AUDIT Challenges

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.

AUDIT Challenges

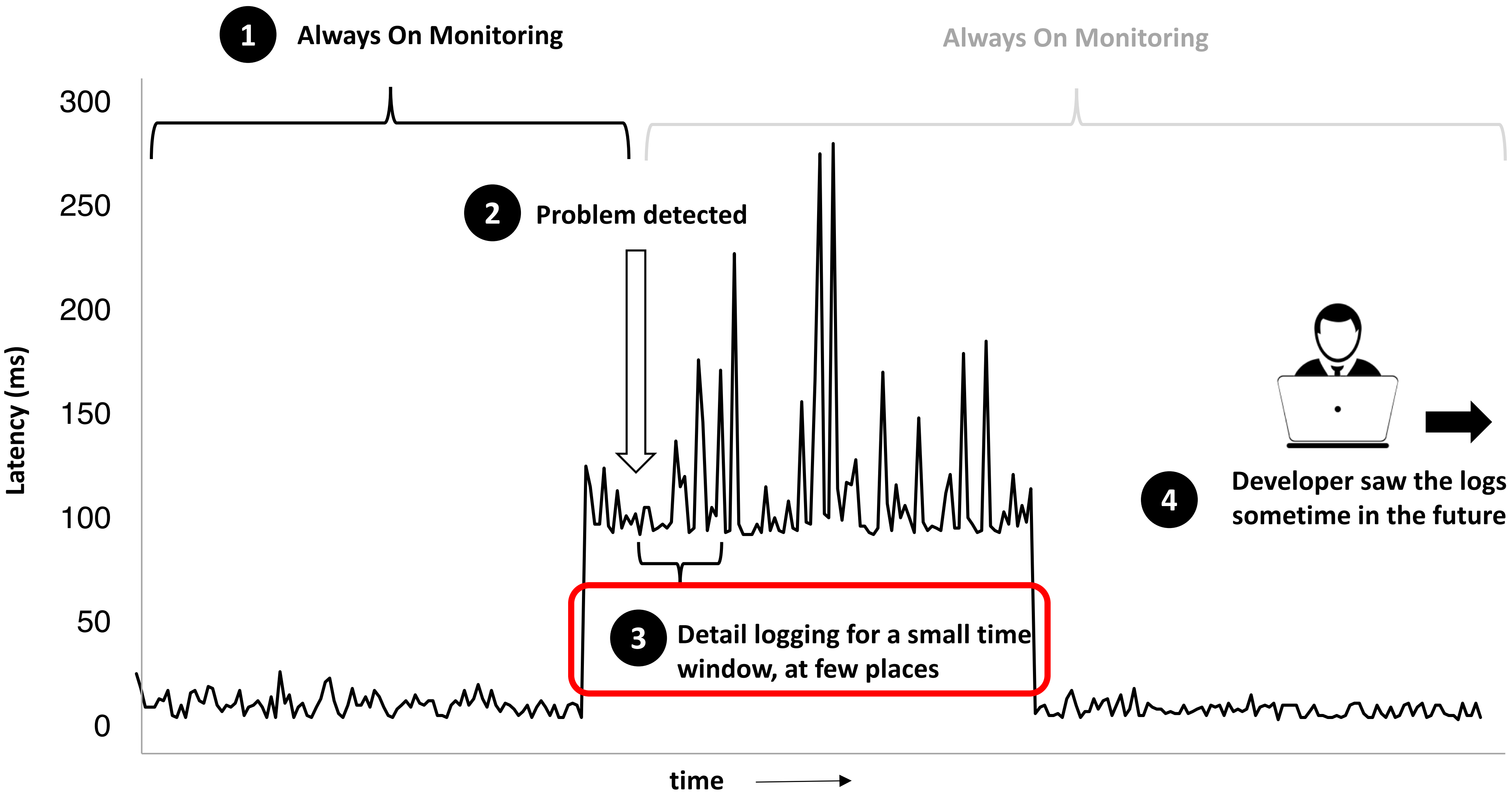
When to log?

Where to Log?

Dynamically Turning Logging On/Off?

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

AUDIT in action



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

AUDIT triggers

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?

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

AUDIT triggers highlights

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

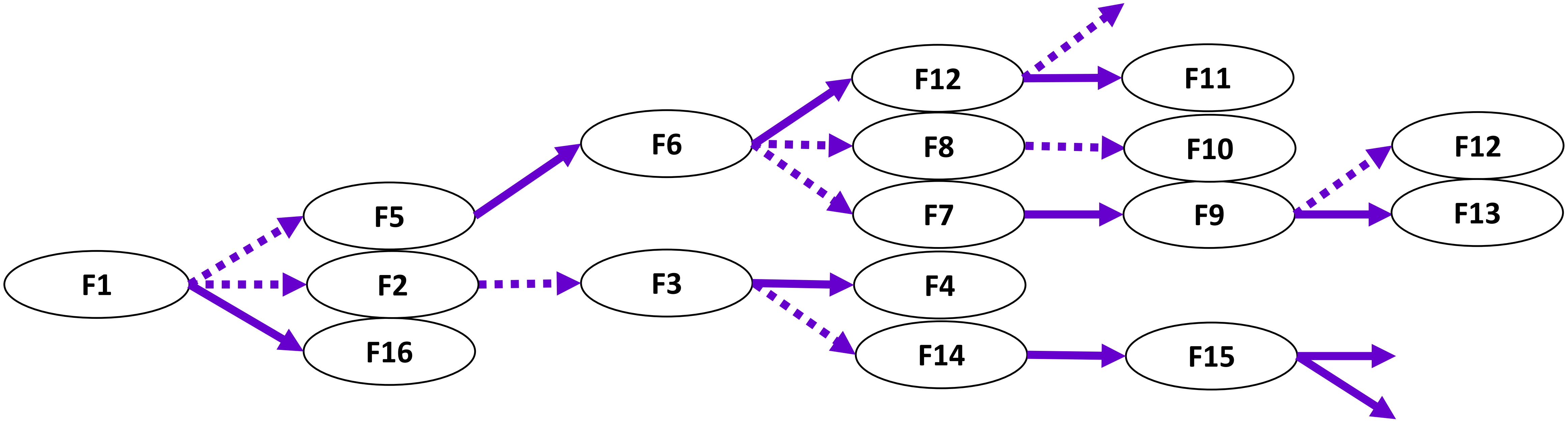
Where to Log?

- Causal tracing (Only methods related to misbehaving requests)
- Blame ranking

Evaluation

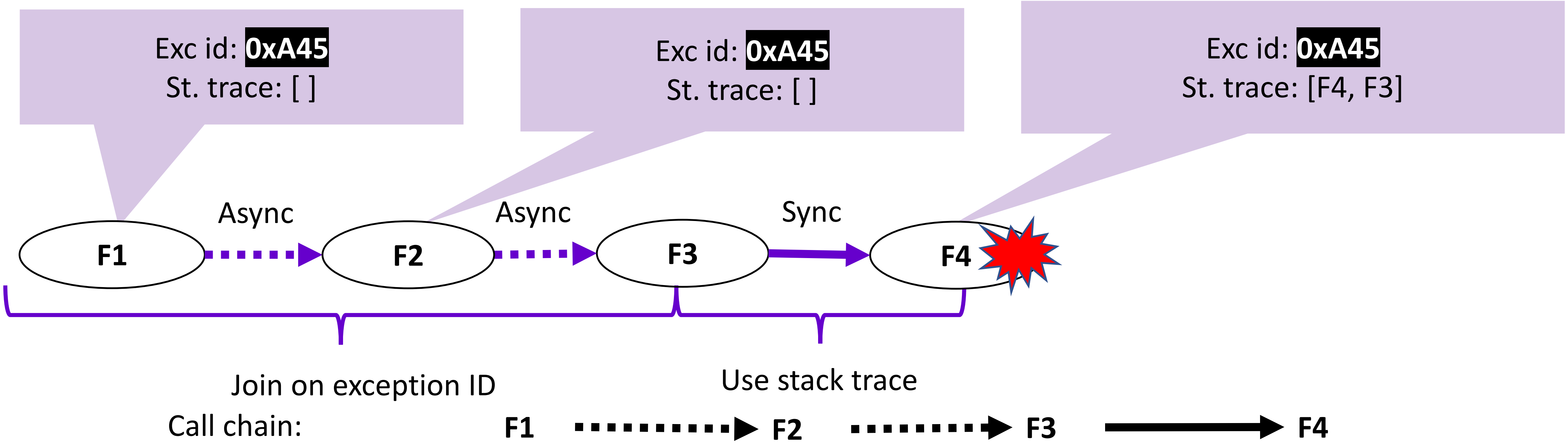
- Case studies
- Overhead

Continuous Causal Tracing: generating request activity graph



AUDIT reconstructs request activity graph (RAG), all methods that are casually related to a request. AUDIT assumes requests are independent of each other.

Continuous Causal Tracing: generating asynchronous exception chain



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.

Continuous Causal Tracing: tracing RAG and AEC

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!

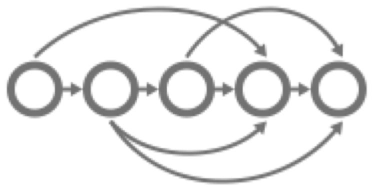
Continuous Causal Tracing: optimization for TAP applications

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.

Continuous Causal Tracing: optimization for TAP applications



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)

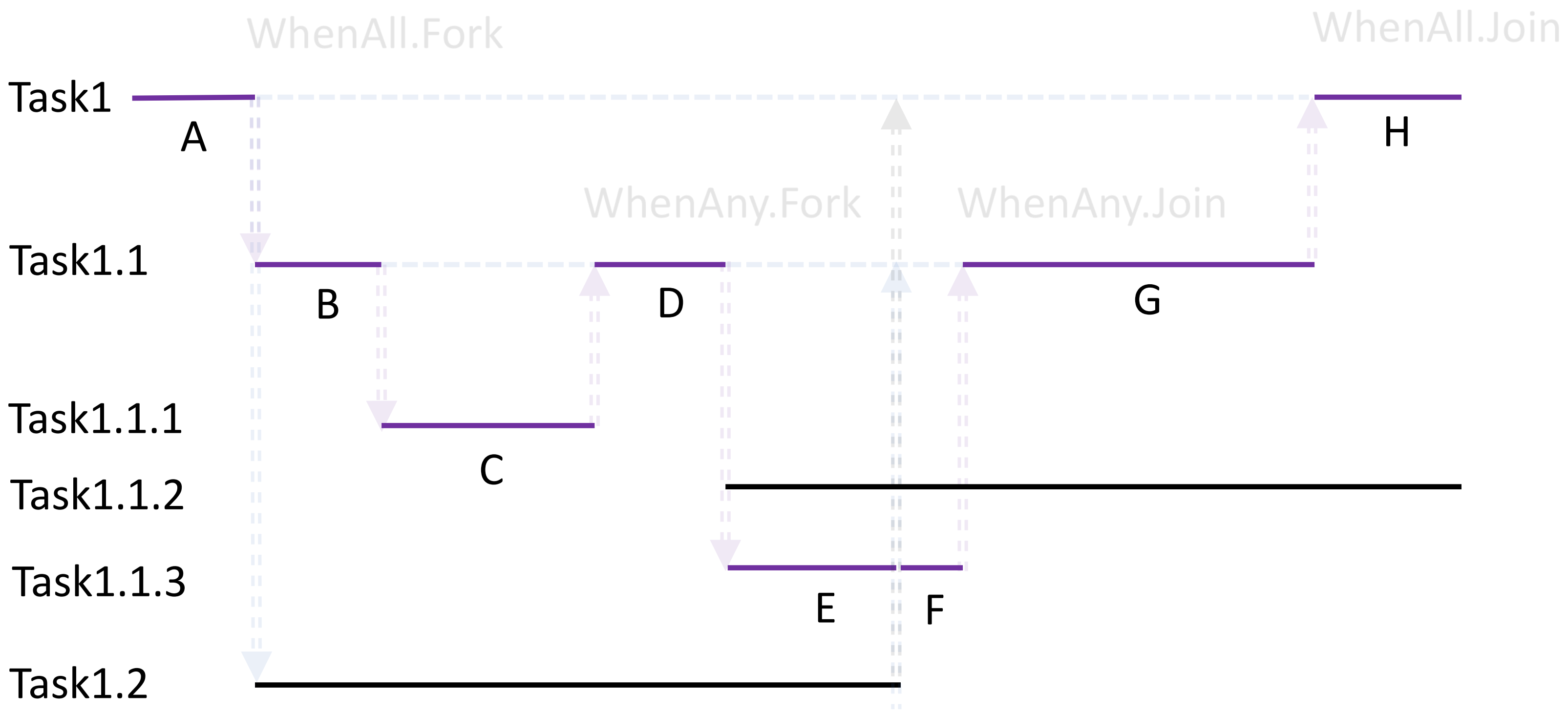
Evaluation

- Case studies
- Overhead

Critical Blame: ranking methods for exception-related trigger

Methods that are closer to the exception are more likely related to the root cause.

Critical Blame: ranking methods for performance-related trigger



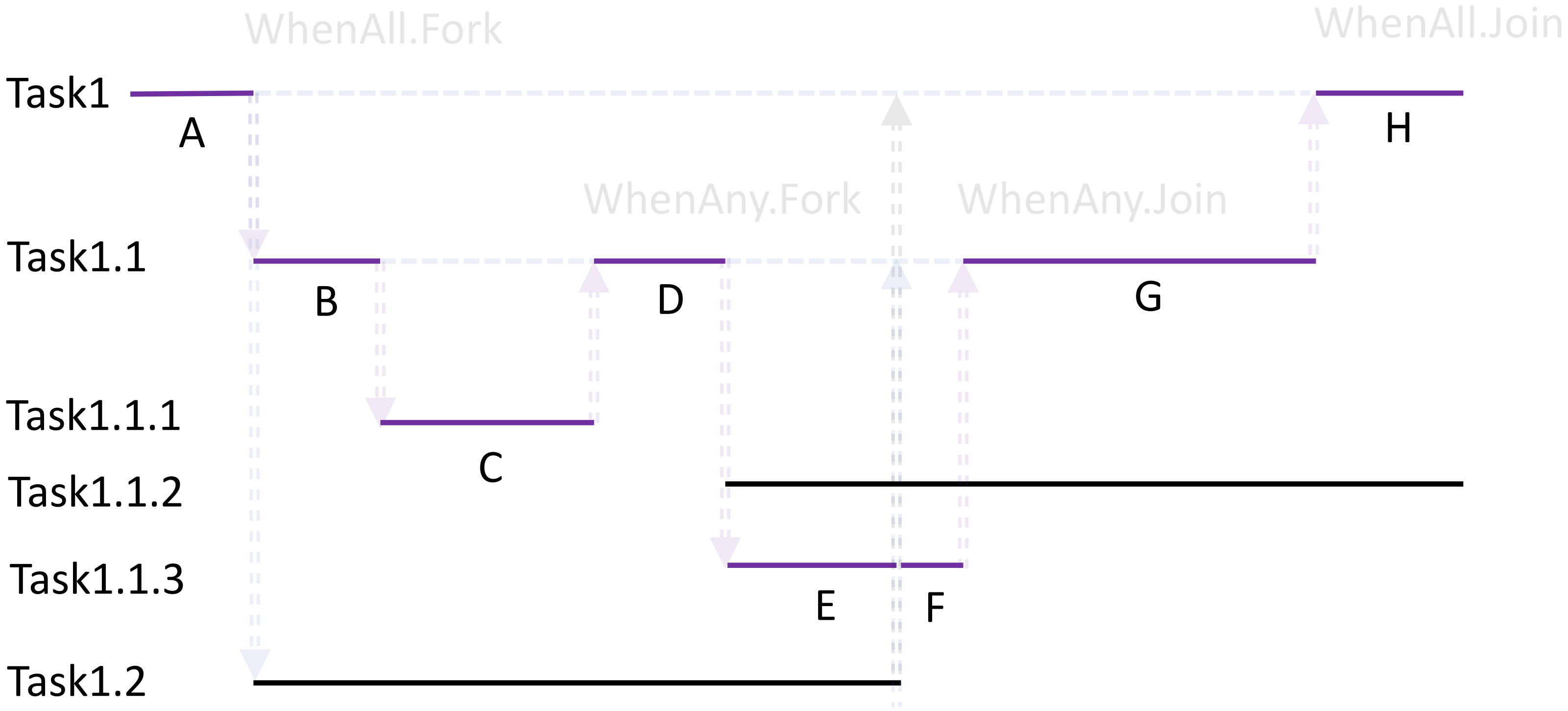
Task	Blame
Task1	A+H/2
Task1.1	(B+D+G)/2
Task1.1.1	C/2
Task1.1.2	E/3+(F+G+H)/2
Task1.1.3	(E+F)/2
Task1.2	(B+C+D)/2+E/3

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

Task	Blame
Task1.1	(B+D+G)/2
Task1	A+H/2
Task1.1.2	E/3+(F+G+H)/2
Task1.2	(B+C+D)/2+E/3
Task1.1.3	(E+F)/2
Task1.1.1	C/2



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

AUDIT Effectiveness: root-causing problems

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

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, investi-gating
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

AUDIT can pinpoint performance issues, such as

conten[illegible]rrency

AUDIT Effectiveness: root-causing problems

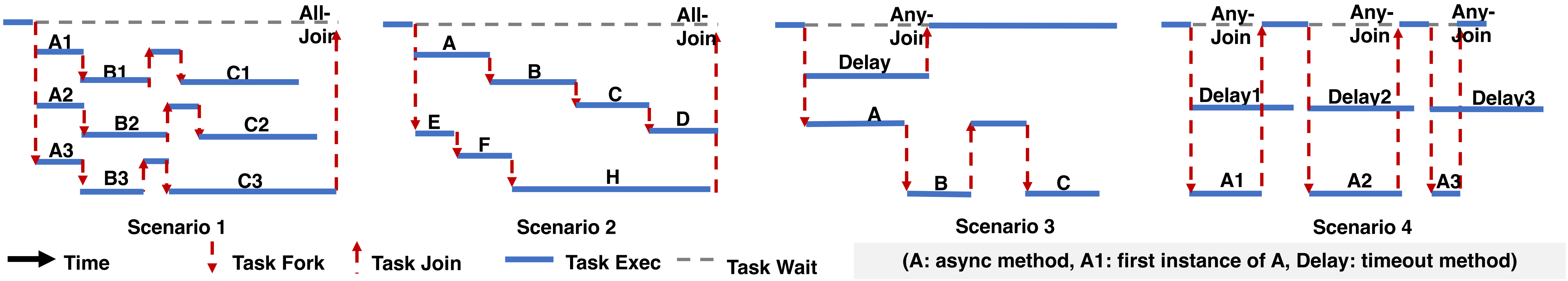
We implemented AUDIT for .NET and applied it to 1 production system at Microsoft and 4 high-profile, open source libraries in GitHub.

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

AUDIT can pinpoint exception issues, such as

file name length
concurrent edits

AUDIT Effectiveness: critical blame ranking



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.

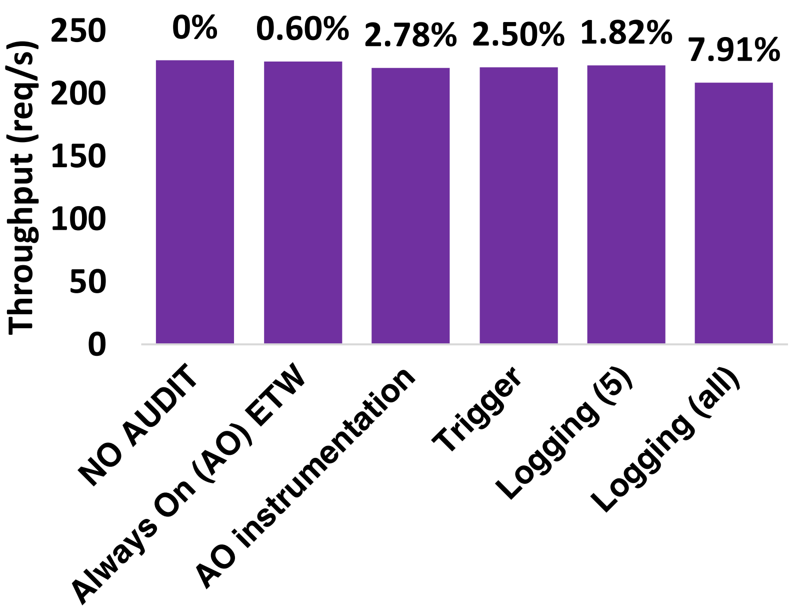
AUDIT is more sensitive than Normalized Processor Time, Top Critical Methods, and Iterative Logical Zeroing in locating bottlenecks.

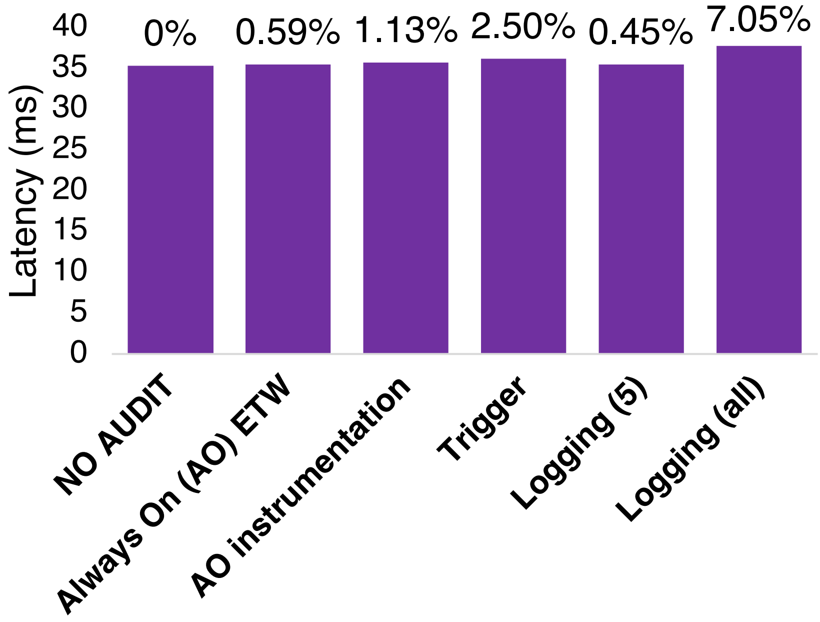
AUDIT Overhead: negligible for real applications

Raw Overhead

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

Real Application (Massive) Overhead





AUDIT

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