DOI:10.1145/3570522

Article development led by acmqueue
queue.acm.org

**CPT can provide actionable and precise latency analysis.**

BY BRIAN EATON, JEFF STEWART, JON TEDESCO, AND N. CIHAN TAS

# Distributed Latency Profiling through Critical Path Tracing

FOR COMPLEX DISTRIBUTED systems that include services that constantly evolve in functionality and data, keeping overall latency to a minimum is a challenging task. Critical path tracing (CPT) is a new applied mechanism for gathering critical path latency profiles in large-scale distributed applications. It is currently enabled in hundreds of different Google services, which provides valuable day-to-day data for latency analysis.

Fast turnaround time is an essential feature for any online service. In determining the root causes of high latency in a distributed system, the goal is to answer a key optimization question: Given a distributed system and workload, which subcomponents can be optimized to reduce latency?

Low latency is an important feature for many Google applications, such as Search,[4] and latency-analysis tools play a critical role in sustaining low latency at scale. The systems evolve constantly because of code and deployment changes, as well as shifting traffic patterns. Parallel execution is essential, both across service boundaries and within individual services. Different slices of traffic have different latency characteristics.

CPT provides detailed and actionable information about which subcomponents of a distributed system are contributing to overall latency. This article presents results and experiences as observed in using CPT in a particular application: Google Search.

*Critical path* describes the ordered list of steps that directly contribute to the slowest path of request processing through a distributed system so optimizing these steps reduces overall latency. Individual services have many subcomponents, and CPT relies on software frameworks[17] to identify which subcomponents are on the critical path. When one service calls another, RPC (remote procedure call) metadata propagate critical path information from the callee back to the caller. The caller then merges critical paths from its dependencies into a unified critical path for the entire request.

The unified critical path is logged with other request metadata. Log analysis is used to select requests of interest, and then critical paths from those requests are aggregated to create critical path profiles. The tracing process is efficient, allowing large numbers of requests to be sampled. The resulting profiles give detailed and precise information about the root causes of latency in distributed systems.

**An example system.** Consider the distributed system in Figure 1, which consists of three services, each with two subcomponents. The purpose of the system is to receive a request from the user, perform some processing, and return a response. Arrows show the direction of requests where responses are sent back in the opposite direction.

The system divides work across many subcomponents. Requests arrive

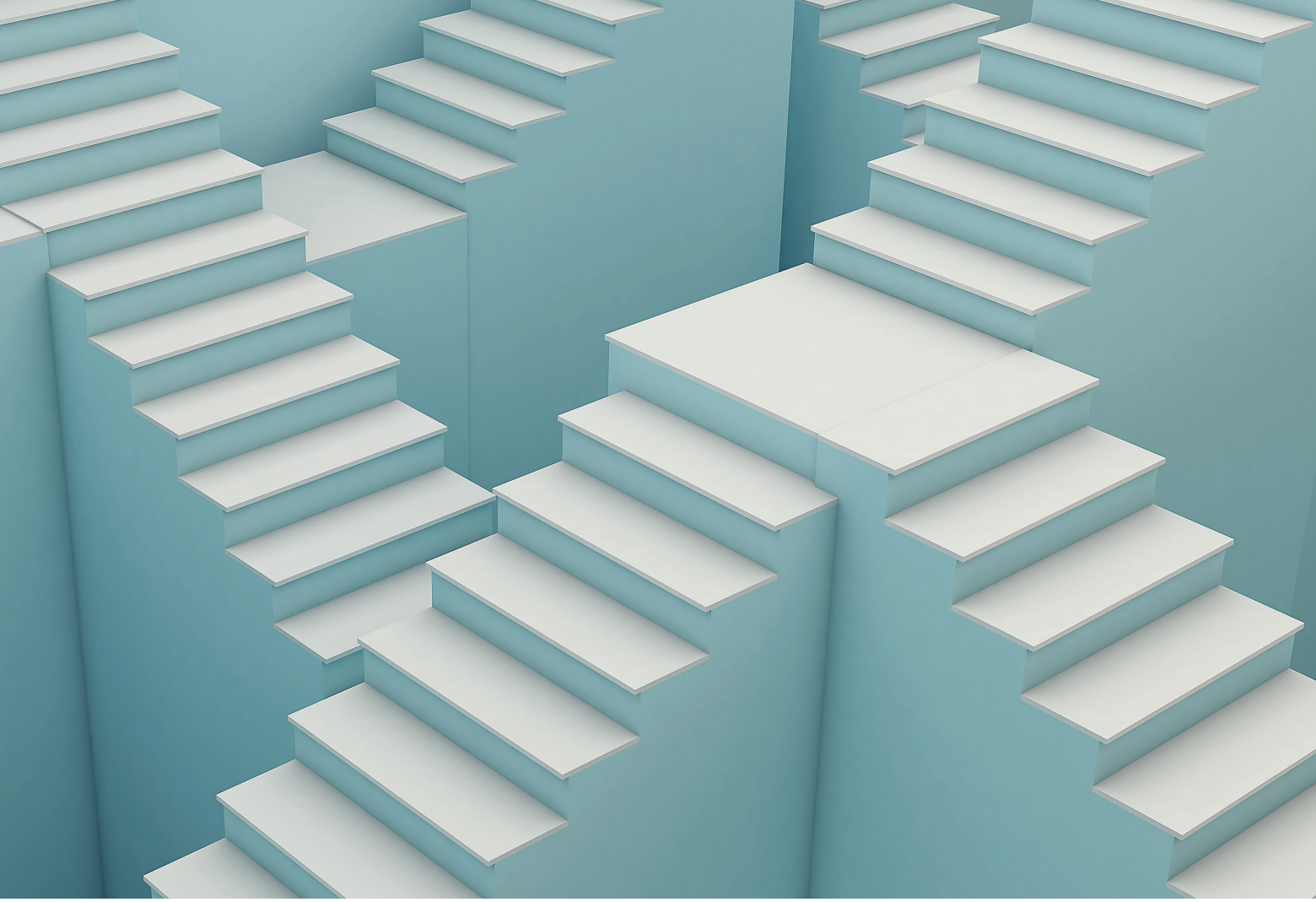


IMAGE BY OSELOTE

at Service A, which hands request processing off to subcomponent A1. A1 in turn relies on subcomponents B1, B2, and A2, which have their own dependencies. Some subcomponents can be invoked multiple times during request processing (for example, A2, B2, and C2 all call into C1).

Even though the system architecture is apparent from the figure, the actual latency characteristics of the system are hard to predict. For example, is A1 able to invoke A2 and B1 in parallel, or is there a data dependency so the call to B1 must complete before the call to A2 can proceed? How much internal processing does A2 perform before calling into B2? What about after receiving the response from B2? Are any of these requests repeated? What is the latency distribution of each processing step? And how do the answers to all of these questions change depending on the incoming request?

Without good answers to these questions, efforts to improve overall system latency will be poorly targeted and might go to waste. For example, in Figure 1, to reduce the overall latency of A1 and its downstream subcomponents, you must know which of these subcomponents actually impact the end-to-end system latency. Before deciding to optimize, you need to know whether $A1 \rightarrow A2$ actually matters.

**Analysis with RPC telemetry.** RPC telemetry is commonly used for latency analysis. Services export coarse-grain information about how many times an RPC is made and the latency characteristics of those RPCs. Monitoring services collect this information and create dashboards showing system performance. Coarse-grain slicing (for example, time range, source, and destination information) is commonly used to enhance RPC telemetry-based analysis.[1,11,16]

RPC telemetry works well when a few RPC services are always important for latency. Monitoring for those services can quickly identify which are causing problems. Service owners can be identified, and they can work to improve performance.

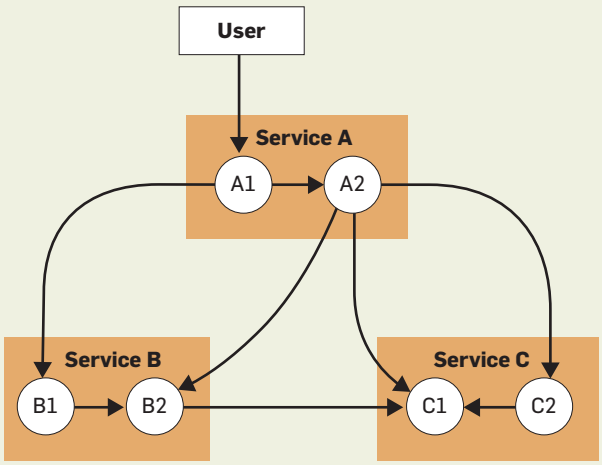
RPC telemetry struggles with parallelism, however. Referring to the figure, assume that A2 does CPU-bound work while waiting for responses from B2, C1, and C2. Improving latency for B2, C1, and C2 will not improve overall performance, because A2 is not actually blocked while waiting for their responses.

Repeated RPCs can also make monitoring confusing. Perhaps A2 is making hundreds of requests to C1. RPC telemetry can show the average latency as very fast—but a single slow RPC out of hundreds might be the root cause of slowness across the entire system.[8] RPC telemetry also cannot tell whether those hundreds of requests are happening in parallel or are serialized. This is important information for understanding how those requests impact latency.

RPC telemetry struggles with identifying important subcomponents within services. For example, both A1 and A2 (subcomponents within Service A) make requests into B2. Telemetry for Service A and Service B will typically mix these requests together, even though they might have different latency characteristics. This can make it difficult to tell which requests should be optimized.

The last major issue with RPC telemetry is the streetlight effect: RPC telemetry sheds light in one particular area of

Figure 1. A simple distributed system.




the system, so you spend time optimizing that part of the system, but meanwhile, latency problems not caused by RPCs get lost in the dark.

**Analysis with CPU profilers.** CPU profiling complements RPC telemetry well. Once RPC telemetry has identified a problematic service, CPU profiling can help figure out how to make that service faster. CPU samples with function call stacks are collected and aggregated, providing insights into expensive code paths. Profiles are typically collected for a single service at a time and might include the type of request that in-flight and hardware profile counters.[2,12,15]

CPU profiling excels at identifying specific expensive subcomponents within services. When CPU time contributes to overall latency, CPU profiling can help identify where to optimize. Many of the same issues that impact RPC telemetry, however, also cause problems for CPU profiles. Lack of information about parallelism means that you can't tell whether CPU-bound work is happening in parallel with RPCs, in parallel with other CPU-bound work, or actually blocking request progress. Heterogeneous workloads cause problems as well: Small but important slices of traffic get lost in the noise. Joining CPU profiles with information about parallelism and request metadata from distributed tracing can address these limitations, but that technology is not widely deployed.[3]

The streetlight effect also impacts CPU profiling: It makes it more likely you will focus on code that uses a lot of CPU, even if that code is not contributing to overall system latency.

**Analysis with distributed tracing.** The last common tool in the latency profiling toolkit is distributed tracing. This approach follows individual requests through a system, collecting timing points and additional data as those requests are processed. Traces are aggregated and analyzed to yield application insights.[5,18,19]

Unlike RPC telemetry and CPU profiling, distributed tracing handles parallelism and heterogeneous workloads well. Information about all cross-service requests is collected, including timing points. Visualization shows exactly when work for each service began and ended, and which services were running in parallel versus serial.

Most distributed tracing includes tracing for RPC boundaries by default but leaves out service subcomponent information. Developers can add tracing for subcomponents as needed.

Workload slicing to find traces for particularly important requests is also possible, although again, developers have to manually tag traces that are important. Distributed tracing even allows automated analysis to identify which services contribute to total latency.[5,20]

The major obstacle to using distributed tracing for detailed latency analysis is cost. In Google Search, a single service might have a few or dozens of RPC dependencies but can easily have 100 times that number of important subcomponents. Instrumenting subcomponents by default increases the size of traces by orders of magnitude.

Sample rates for distributed tracing systems are another issue. As tracing becomes more detailed, it becomes more expensive. Adaptive sampling techniques[18,19] are sometimes used to increase the sampling rate for interesting requests. If the task is to investigate the 99$^{th}$ percentile latency for a 1% experiment, finding a single relevant example requires 10,000 traced requests; 10,000 to 100,000 examples might be necessary to gain statistical confidence about where a regression occurred. Together, this means that it might be necessary to gather full traces for $10^8$ to $10^9$ requests, with each trace being 100 times larger than the default expected by the distributed tracing system.

**Critical path tracing.** CPT is designed to fill some of the gaps in these systems. The term *critical path* in project management[14] refers to the many interdependent steps that must be completed to finish a project. In this article, critical path describes the ordered list of steps that directly contribute to the slowest path of request processing through a distributed system. Aggregating these traces into *critical path profiles* can identify latency bottlenecks in the overall system.

Software frameworks are used to instrument service subcomponents automatically.[17] Support has been added to most of the commonly used frameworks in Google Search, and developers using those frameworks get fine-grain CPT without additional coding effort.

Framework instrumentation code automatically identifies the critical path for request execution. Only the critical path is retained for analysis; other trace information is discarded. This reduces tracing cost by orders of magnitude.

Critical path traces are logged alongside other request and response metadata, such as A/B experiment information and which features appear on the Search results page. This allows standard log-analysis techniques to use business criteria to find traces from requests of interest. Other distributed tracing systems log traces separately for each machine involved in a request. Reconstructing a complete request requires joining traces from hundreds of machines. Logging all critical path subcomponents together avoids the overhead of joining the traces.

Together, these cost reductions allow detailed traces with high sampling rates.

### Tracing a Request

This section describes the work needed to gather a fine-grain critical path trace for a single request.

**Critical path definition.** For latency profiling, the key input used is the critical path—the set of steps that blocked progress on producing the response. When multiple steps proceed in parallel, the slowest step is the only one on the critical path.

The execution of the request can be modeled as a directed graph of named nodes (for example, subcomponent names). Each node in the graph does some of its own computation. Each edge in the graph is a dependency where a node must wait for completion of one of its dependencies before computation can proceed. The critical path is the longest-duration path through the nodes, starting at the request entry point and finishing at the node that computes the response. The length of the critical path is the total latency for processing the request.

Consider a distributed system such as that in Figure 1 and suppose the subcomponents execute in parallel according to scenarios as summarized in Figure 2. Figure 2a shows an example of a critical path calculation where the requests to B1 and A2 happen sequentially: A1 does some computation, then blocks, waiting for B1 to complete. A1 proceeds with additional computation before waiting for A2 to complete. This example has a critical path of {A1=5ms, B1=20ms, A1=8ms, A2=2ms}, with a total critical path of 35ms.

Figure 2b shows how this changes when B1 and A2 execute in parallel. The new critical path becomes {A1=5ms, B1=20ms, A1=8ms}, for a total critical path of 33ms. A2 has been eliminated from the critical path. In this scenario, optimization efforts should be focused on A1 and B1.

Parallel execution also applies when the parent node overlaps with child nodes, as in Figure 2c. In this example, A1 sends an RPC to B1 but does not immediately block to wait for a response. Instead, A1 continues with other computations in parallel. For this case, whichever node finishes last to the critical path is assigned: {A1=3ms, B1=14ms, A1=10ms}.

**Identifying service subcomponents.** Infrastructure-first distributed tracing systems typically collect data at RPC boundaries by default and allow developers to add trace annotations for more-detailed tracing.[18,19] With CPT, a developer can take advantage of the standardization provided by software frameworks to collect more-detailed information by default.[17]

**The major obstacle to using distributed tracing for detailed latency analysis is cost.**

As an example, the Dagger framework encourages authors to write code as ProducerModules.[7] Each ProducerModule declares the inputs it requires and the outputs it produces. Dagger then coordinates the execution of ProducerModules to process a request. Referring to Figure 1, the following code snippet shows a Dagger implementation of subcomponent A1:

```
@ProducerModule
public abstract class A1ProducerModule {
  @Produces
  @A1Output
  static A1Output runA1(@A2Output a2,
@B1Output b1, @B2Output b2) {
  ... Code reads information from a2, b1,
and b2, and calculates A1Output...
  }
}
```

The collection of ProducerModules creates a graph of subcomponents that are executed by the framework to process the request. For this example, the framework knows which of A2, B1, and B2 was the last to block execution to produce A1's output. Since the framework is aware of the subcomponent dependencies, it can record the critical path.

For Google Search, subcomponent-level traces are collected from several software frameworks in multiple programming languages. Framework-level implementation is essential for scalability, since it allows relatively small teams of developers to provide detailed critical path traces for code written by thousands of other people. Since each framework instruments and reports the critical path automatically, most developers are not aware that critical path tracing is happening.

A few services that do not use frameworks have implemented service-specific CPT as well. Traces from these services are more closely grained. The system degrades gracefully when requests are made to services that do not provide any traces. The size of the blind spot in the critical path is reported correctly and in-

cludes which service is responsible for that part of the critical path.

**Propagation and merging.** Propagating child-node paths to parent nodes allows more-detailed views of the request critical path.

In Figure 2b, the initial critical path is {A1=5ms, B1=20ms, A1=8ms}. The B1 child node takes up the majority of execution time; however, B1's internal execution details are lacking. To add this detail, B1 first computes its own critical path (for example, {B1=4ms, B2=12ms}) and returns that path to A1.

A1 then merges the critical path from B1 into the overall critical path, and so on, recursively through the system. This presents potential challenges at each service boundary for *undercounting*, or A1 seeing a higher latency than that spent on B1 (for example, network routing or request/response serialization is nontrivial), as well as *overcounting*, where B1 reports a higher latency than that observed by A1 (typically, an instrumentation bug in B1). Since each service reports its own critical path, *blind spots* (for example, cases where B1 does not report any critical path) are also common instrumentation challenges in Search.

**Triggering.** Depending on the framework implementation, CPT can incur significant overhead. In practice, not every request has to be traced, so sampling is used to amortize the cost. Sampling requires cross-service coordination to avoid creating unnecessary blind spots in the traces.

Each service can make an independent sampling decision, and then it conveys that decision on outbound RPCs in request metadata. As with Facebook's Mystery Machine,[5] when downstream services see that the caller has opted in to CPT, the downstream services enable tracing as well. Even if the caller has not opted in to sampling, downstream services are free to track and log their own critical paths. Callers that have not requested sampling will ignore the resulting traces.

System operators can opt in to tracing for specific requests, instead of relying on random sampling. This is useful for cases where a human operator needs to collect traces for debugging purposes. When identifying a particularly slow type of request, the operator collects many samples for that request to get data for a profile. Since these requests are not random samples, they are excluded from the aggregated analysis by default.

**For latency profiling, the key input used is the critical path—the set of steps that blocked progress on producing the response.**

**Limitations of CPT.** The critical path as defined here is useful for addressing culprit-finding problems but has some limitations.

In general, any latency optimization efforts should focus on the subcomponents that are on the critical path. Resource contention with off-critical path subcomponents, however, can also slow down critical path execution.

Considering Figure 2b again, imagine that A2 is holding a mutex or is running intensive computation and causing CPU starvation in critical path code. The critical path will show part of the resource-starvation issue: Nodes waiting on the blocked resource will appear on the critical path. The culprit node (A2), however, will not appear on the critical path. Profilers focused on CPU and lock contention are well suited for identifying these bottlenecks.

The critical path also lacks visibility into the drag and slack[14] of the overall execution. Drag and slack are measures of the potential change in the critical path based on optimizing a single step. A single subcomponent that blocks all others has large drag: Improving that one subcomponent is likely to improve the overall execution time.

When multiple subcomponents run in parallel, they have large slack: Even if one slows down, it probably won't impact the overall time. Causal profiling[6] uses an experiment-driven approach to identify headroom by automatically injecting latency into different subcomponents to determine their drag and slack. Quartz[3] aims to identify subcomponents with high drag via CPU profiling. The Mystery Machine[5] identifies subcomponent slack by reconstructing system-dependency graphs via log analysis.

Streaming is an important technique for improving latency, but unfortunately, CPT for streaming APIs is not well defined. Services can return a stream of results, allowing work to begin on the early results before later results are ready. Bidirectional streaming, where client and server send multiple messages back and forth, is a more complex programming model but is also useful for latency in some situations. The critical path must be carefully defined for these operations: Should segments end when the first message is returned, or

the last? What if the caller of the API is also streaming? Google Search's current implementation of CPT deals with this challenge by defining the critical path segment as ending when the last message of the stream is received, but this definition is misleading in situations where earlier messages are more important for latency.

Even with these caveats, though, CPT helps focus effort on areas where it is most likely to make a difference.

**Operational costs.** Operational overhead of CPT is low enough to enable profile data collection by default in many applications, including Google Search. Search collects traces continuously on 0.1% of requests. Traced requests see a 1.7% increase in mean latency, adding .002% to overall mean request latency. Overhead remains low at the tail, with 99$^{th}$ percentile latency overhead for traced requests being 2.0%.

CPU overhead is more difficult to calculate. Frameworks were rewritten to incorporate CPT at runtime. Those changes typically incurred less than 0.1% of the overall CPU cost, but framework overhead depends on the workload. The CPU overhead was deemed acceptable when compared with the value of the latency profiles.

Network overhead is significant, largely because of an overly verbose wire format. For a critical path with $N$ elements and an average of $M$ subcomponents per element, the wire format uses $O(N * M)$ memory. A graph representation would encode the same information at lower cost. In practice, the network overhead is mitigated by both sampling only 0.1% of requests and compression.

## Aggregation and Visualization

A single critical path trace is interesting but might not present a realistic view of overall system performance. Single requests can be outliers. Merging multiple profiles creates a statistically significant view of system performance.

Individual critical paths are logged along with other metadata about the request. Standard log-analysis techniques are used to select a random sample of requests, meeting whatever criteria are of interest. For example, critical path analysis for Google Search routinely includes slicing by A/B experiment arms, filtering to select requests where certain UI elements appear on the Search results page, and from certain time ranges and geographic regions. One productive technique is simply to select only requests that were extremely slow. Analyzing these requests frequently reveals systemic issues that amount to significant optimization opportunities.

Note that unlike distributed tracing systems,[19] a single log entry contains a critical path that spans multiple services. This is important for correctness: Back-end services can record their own critical path, but they cannot have a priori knowledge of whether their work will be on the critical path of their callers.

**Aggregation algorithm.** The techniques Google Search uses for aggregating critical path information are similar to the mechanisms generally used in CPU profiling. The process creates a profile for visualization and analysis by merging critical paths from sampled requests into a single "average critical path," similar to the Mystery Machine.[5] Google Search uses pprof[10] for visualization, but CPT is flexible and can be used in conjunction with other visualization tools.

Consider Figure 1 as an example system where two requests are received. Table 1 shows how these two requests merge to create an average critical path.

**Figure 2. Different parallelized execution scenarios.**

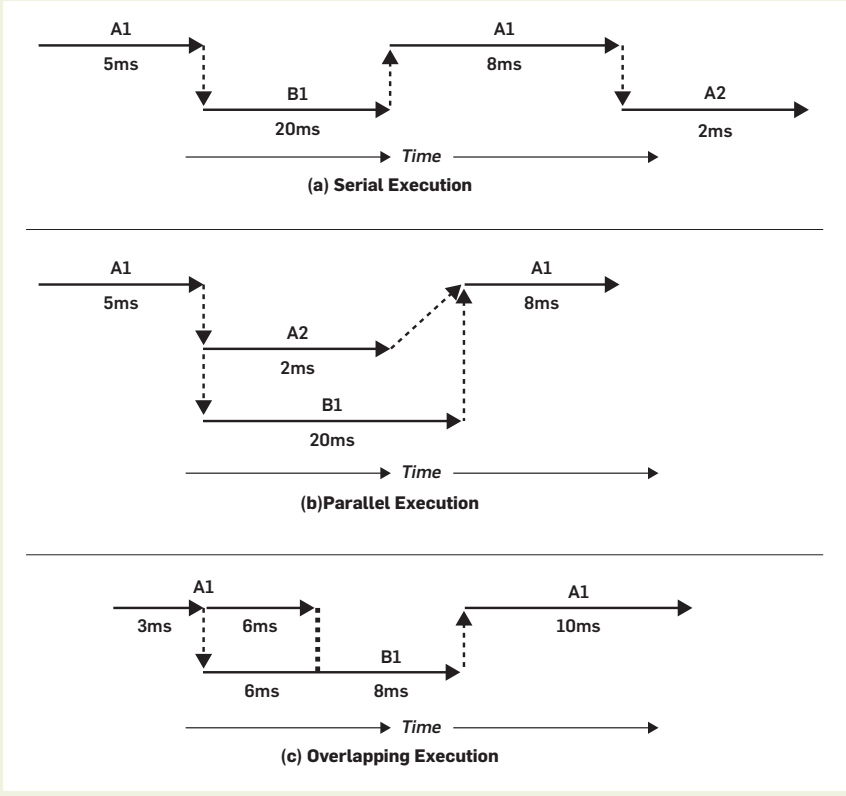



**Figure 3. An example CPT visualization as a flame graph.**

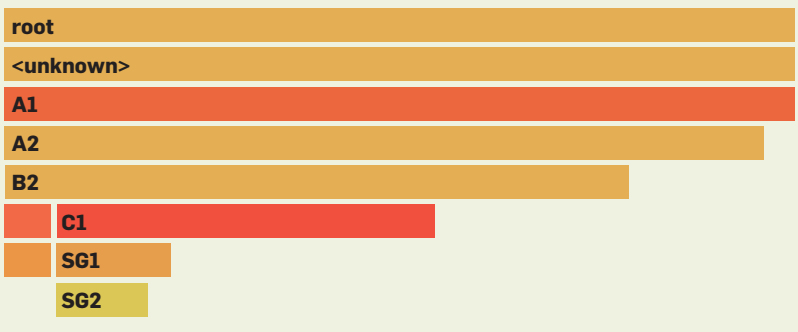


**Table 1. Critcal path aggregation.**

| Subcomponent | Request 1 | Request 2 | Aggregated |
|---|---|---|---|
| A1/self | 6ms | 4ms | 5ms |
| A1/A2 | 10ms | — | 5ms |
| A1/B1 | — | 4ms | 2ms |
| Total | 16ms | 8ms | 12ms |

**Table 2. Precision of critical path time.**

| Sample Size | 95th Percentile CI Width |
|---|---|
| 1,000 | ±19.9ms |
| 10,000 | ±6.3ms |
| 100,000 | ±2.0ms |
| 1,000,000 | ±0.7ms |

In Request 1, A1 calls A2 and B1 in parallel, and only A2 is on the critical path. In Request 2, A2 and B1 are called again in parallel, but B1 is slower and on the critical path. These are aggregated into a single profile by merging subcomponents with identical paths and then dividing by the number of samples to keep metrics relative to average request latency.

This aggregation can be thought of as the average critical path of the overall system. Interpretation of the average critical path has to bear in mind that subcomponents are not sequential, and the average might not reflect any real critical path that the system could have taken. If two subcomponents are called in parallel, it is not possible for both to be on the critical path for a single request. Both are still likely to appear on the average critical path.

Subcomponent time on the average critical path reflects both how often the subcomponent is on the critical path and how long it takes when present. The difference becomes important when considering system subcomponents that are usually fast but occasionally slow. A system subcomponent that always takes 5ms on the critical path will show up in an aggregate profile as taking 5ms. A system subcomponent that is on the critical path only 1% of the time but takes 500ms when it does appear will also show up in an aggregate profile as taking 5ms.

As with other statistical analysis, looking at the distribution of the data in the sample pool can be helpful. In addition to pprof format profiles, aggregation tools collect latency histographs for critical path segments, and this data is consulted early on when latency problems are investigated. Optimizing something that occurs rarely but is very slow is a different problem from optimizing something that occurs frequently. When the latency distribution for a particular subcomponent is highly skewed, it is helpful to have additional sampling to focus on cases where the subcomponent is slow.

**Precision of total critical path.** How many samples are needed to be confident that a critical path change of a certain size is not a result of random variation in request latency? The central limit theorem can calculate confidence intervals for various sample sizes. Table 2 shows estimates of the width of the 95th percentile confidence intervals for Google Search profiles.

In practice, profile visualization becomes quite slow, with millions of requests in a sample. The default is to look at samples of 100,000 requests as a compromise between precision and usability of profiling tools, and to increase sampling beyond 100,000 requests as needed.

**Individual subcomponent precision.** Aggregated profiles contain thousands of subcomponents. This creates a situation where even 95th percentile confidence intervals can yield a large number of false positives—cases where a subcomponent appears to have had a statistically significant change in latency, but the difference is actually caused by random variation. False positives waste engineering time since investigating the latency change will not find a root cause.

One way of measuring the false positive rate is to create two profiles using identical sampling criteria and then compare the resulting profiles using pprof's difference-base view. The difference-base view highlights large differences between profiles. When you know in advance that the profiles should be the same, any differences larger than a given threshold are false positives.

To quantify the latency-profiling false-positive rate for Google Search, we repeated the pairwise comparison experiment 100 times at various sample sizes and with various thresholds for considering a profile difference important enough to investigate. Table 3 shows the 95th percentile confidence intervals for the expected number of false positives. As an example, if the profile comparison is based on 1,000 requests per profile, you can be 95% confident that the average number of components with false positives larger than 5ms is between 0.2 and 0.4.

In Google Search, it's extremely rare for profiles to show large differences in a single subcomponent because of random variation. An additional finding from the analysis is that not all subcomponents are equally likely to show random differences. Components that have larger mean times and standard deviations also have larger false-positive rates. (The central limit theorem predicts this outcome. Even though subcomponents have non-normal latency distributions, estimates of mean latency become more precise as standard deviation shrinks and sample sizes grow.)

This finding has important implications for latency investigations in Search. Identifying the root causes of latency differences in A/B experiments is straightforward, because two profiles with identical sampling criteria can be compared. Any subcomponent-level differences above 1ms are probably caused by the experiment, not by random variation. Root-cause analysis is much more difficult when comparing profiles of different workloads. Under those circumstances, larger sample sizes do not make the profile differences clearer—any large differences between the pro-

**Table 3. Components with false positives.**

| Sample Size | False Positives > 5ms | False Positives > 1ms | False Positives > 0.1ms |
|---|---|---|---|
| 1,000 | [0.2 – 0.4] | [3.5 – 4.2] | [56.8 – 61.2] |
| 10,000 | n/a | [0.7 – 1.1] | [15.6 – 18.0] |
| 100,000 | n/a | n/a | [3.0 – 3.6] |

files are probably a result of the different sampling criteria, not random variation.

**Visualization.** Once profiles have been aggregated, a variety of visualizations can be used to help engineers understand system latency. Top-down, bottom-up, call graphs, and flame graphs[13] are all helpful. Figure 3 shows an example CPT visualization as a flame graph, with a latency- critical path represented as columns of boxes, and each box representing a subcomponent. (The color of each box is insignificant and optimized for easy viewing.) Profile comparison is essential for diagnosing regressions, since they show only subcomponents with large changes. This is particularly helpful for analyzing regressions in A/B experiments.

## Future Work

The approach outlined here is practical and scales well, but has limitations. One area for future work is increasing adoption of fine-grain tracing and reducing the size of blind spots in profiles. This is likely to require increasing adoption of frameworks-based programming models.

CPT is currently defined only where RPCs create a directed graph of calling nodes with a clear hierarchical structure of RPCs. Streaming protocols that make incremental progress or contain multiple requests and responses in one session are difficult to analyze with CPT, and can be handled in future work.

CPT focuses on server-side latency analysis. Adding support for client-side components of latency (as was done with Facebook's Mystery Machine[5]) would improve the ability to understand which contributors to end-to-end latency have the most impact on the end-user experience.

The frameworks integration that CPT uses to identify service subcomponents has profiling applications outside of latency: We have built proof-of-concept distributed cost profilers within Google Search that allow downstream systems to trace expensive traffic to specific subcomponents in callers. These cost profiles are detailed and useful, but have not been generalized for use outside of Search.

Finally, adding metadata to trace collection seems both possible and helpful. For example, frameworks could collect *slack* information for each critical path node. Where a node is blocked, frameworks could collect information about the contended resource that is causing the delay. Continued research in this area should yield additional techniques to help profile for latency.

## Conclusion

In large, real-world distributed systems, existing tools such as RPC telemetry, CPU profiling, and distributed tracing are valuable for understanding the subcomponents of the overall system but insufficient to perform end-to-end latency analyses in practice. Issues such as highly parallel execution flows, heterogeneous workloads, and complex execution paths within subsystems make latency analysis difficult. In addition, these systems and workloads change frequently.

No one team or person has a detailed understanding of the system as a whole.

CPT in such systems addresses these challenges to provide actionable, precise latency analysis. Integration at the frameworks level[17] provides detailed views of application code without requiring each developer to implement their own tracing. Common wire protocols allow consistent mechanisms for triggering, aggregation, and visualization. Efficient implementation allows a high sampling rate, giving precise profiles even for relatively rare events. Scalable and accurate fine-grain tracing has made CPT the standard approach for distributed latency analysis for many Google applications, including Google Search.

**Acknowledgments.** This article would not be possible without a number of contributors from throughout Google: particularly F. Gura, D. German, S. Procter and M. Levin. We also thank J. Klein, Ł. Milewski, A. Sheikh, P. Haahr, B. Stoler and S. Virji.

See the original article at *acmqueue*[9] (https://queue.acm.org/detail.cfm?id=3526967), which features case studies, including an example illustrating the challenges with streaming RPC APIs and another illustrating the value of focusing on the most affected subset of requests using a real example from a Search home page latency regression.

**References**

1. Amazon Web Services. Amazon CloudWatch: Observability of your AWS resources and applications on AWS and on-premises; https://aws.amazon.com/cloudwatch/.
2. Amazon Web Services. What is Amazon CodeGuru profiler?; https://go.aws/3C6P8s8.
3. Anderson, T.E., Lazowska, E.D. Quartz: a tool for tuning parallel program performance. In *Proceedings of the 1990 ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, 115–125; https://dl.acm.org/doi/10.1145/98457.98518.
4. Arapakis, I., Bai, X., Cambazoglu, B.B. Impact of response latency on user behavior in web search. In *Proceedings of the 37th Intern. ACM SIGIR Conf. Research and Development in Information Retrieval*, 2014, 103–112; https://dl.acm.org/doi/10.1145/2600428.2609627.
5. Chow, M., Meisner, D., Flinn, J., Peek, D., Wenisch, T.F. The Mystery Machine: end-to-end performance analysis of large-scale internet services. In *Proceedings of the 11th Usenix Symp. Operating Systems Design and Implementation*, 2014, 217–231; https://dl.acm.org/doi/10.5555/2685048.2685066.
6. Curtsinger, C., Berger, E.D. Coz: Finding code that counts with causal profiling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, 184–197; https://dl.acm.org/doi/10.1145/2815400.2815409.
7. Dagger. Dagger Producers; https://dagger.dev/dev-guide/producers.
8. Dean, J., Barroso, L.A. The tail at scale. *Commun. ACM 56*, 2 (Feb. 2013), 74–80; https://dl.acm.org/doi/10.1145/2408776.2408794.
9. Eaton, B., Stewart, J., Tedesco, J., Tas, N.C. Distributed latency profiling through critical path tracing. *acmqueue 20*, 1 (2022); https://queue.acm.org/detail.cfm?id=3526967.
10. GitHub. pprof; https://github.com/google/pprof.
11. Google. Cloud monitoring; https://cloud.google.com/monitoring.
12. Google. Google Cloud's operations suite (formerly Stackdriver); https://cloud.google.com/profiler.
13. Gregg, B. The flame graph. *Commun. ACM 59*, 6 (Jun. 2016), 48–57; https://dl.acm.org/doi/10.1145/2909476.
14. Kelley Jr., J.E. Critical path planning and scheduling: mathematical basis. *Operations Research 9*, 3 (1961), 296–435; https://www.jstor.org/stable/167563.
15. Microsoft. Profile production applications in Azure with Application Insights Profiler; https://docs.microsoft.com/en-us/azure/azure-monitor/app/profiler-overview.
16. Microsoft. Azure Monitor; https://azure.microsoft.com/en-au/services/monitor/.
17. Nokleberg, C., Hawkes, B. Best Practice: Application Frameworks. *acmqueue 18*, 6 (2021), 52–77; https://queue.acm.org/detail.cfm?id=3447806.
18. Pandey, M et al. Building Netflix's Distributed Tracing Infrastructure. Netflix Tech Blog; 2020; https://bit.ly/3e1C92Z.
19. Sigelman, B.H. et al. Dapper, a Large-scale Distributed Aystems Tracing Infrastructure. Google Technical Report, 2010; https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf.
20. Yang, C.-Q., Miller, B. Critical path analysis for the execution of parallel and distributed programs. In *Proceedings of the 8th Intern. Conf. Distributed Computing Systems*. IEEE Computer Society Press, 1988, 366–375.

**Brian Eaton** joined Google's Information Security team as a software engineer in 2007 and joined Web Search in 2014 to lead engineering productivity and performance teams, developing latency profiling and the first microservices deployments in Web Search. He now works on Search Quality and user trust.

**Jeff Stewart** worked at Google for more than 17 years, focusing on Web Search, leading teams to improve compute efficiency and end user latency. He led the initiative to enable https on search traffic by default and helped launch Gmail. He moved to a new employer in 2021.

**Jon Tedesco** has worked at Google in Web Search infrastructure for nine years. During that time, he has led several teams to build latency and compute analysis tools for the Search stack, and to develop new components of Search infrastructure focusing on performance analysis and optimization.

**N. Cihan Tas** is a site reliability software engineer on the Stadia team at Google, currently focusing on the optimization of datacenter networks. Before Google, he worked at Siemens Corporate Research in wireless networks, broadly ranging from vehicle-to-vehicle networks to smart-grid communication networks.