# QProf: Fleetwide Transitive Cost Profiling of Warehouse-Scale Services

Sam (Likun) Xi
xyzsam@google.com

Alexey Alexandrov
aalexand@google.com

Ali Sheikh
ofrobots@google.com

Ning Wang
nwang@google.com

Vance Lankhaar
vlankhaar@google.com

Tipp Moseley
tipp@google.com

Parthasarathy Ranganathan
parthas@google.com

## Abstract

Warehouse-scale services form the backbone of major cloud services relied on by billions of users every day. As these systems grow increasingly complex, understanding them and finding opportunities for optimization becomes more and more difficult. In this paper, we present QProf, a distributed systems profiler built upon RPC tracing. Going well beyond prior work that focuses on fleetwide profiling of single programs, QProf focuses on *cost* profiling of *entire services*. By treating a distributed program as if it were a single process, QProf can produce call-graph profiles of entire systems, so it can measure the transitive cost of services through the entire stack of backend dependencies. QProf is backed by a novel tracing mechanism called *skeletal tracing* which is exceedingly low-overhead and invisible to applications, ensuring that tracing minimally perturbs application behavior. An *ambient per-query CPU profiler* ensures high profiling coverage across thousands of microservices without manual instrumentation, and new *fan-in APIs* provide support for representing batched requests in a tracing data model that is tree-centric. QProf processes billions of traces per day to produce aggregated datasets which can be queried by engineers. Despite the sampled nature of traces, QProf is highly accurate in measuring QPS and CPU across thousands of microservices in an extremely diverse datacenter fleet. QProf has been deployed for every job in Google's production fleet for several years. It has a myriad of use cases, and we present several case studies showing how it has been used to find the "room-at-the-top" in complex systems and optimize them across many dimensions.

***CCS Concepts:*** • **Computer systems organization** → **Cloud computing**; **Client-server architectures**; • **General and reference** → **Performance**.



*SOSP '26, Prague, Czech Republic*



ACM ISBN 979-8-4007-2585-2/26/09
https://doi.org/10.1145/3830418.3843854

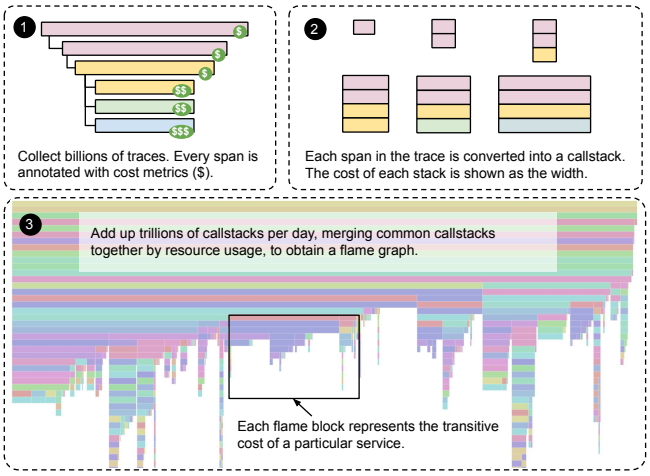


**Figure 1.** High-level view of the QProf data pipeline: 1) billions of distributed traces are collected over the fleet, 2) converted into a set of callstacks with resource usage metrics, and 3) added together to form aggregate views (like flame graphs) to show the source of cost in the system.

***Keywords:*** QProf, transitive cost, distributed tracing, datacenter, profiler, context, room-at-the-top

## 1 Introduction

In recent years, the importance and public awareness of datacenters has grown to huge proportions. All cloud services today are powered by warehouse-scale computers [9], running on thousands of datacenters around the world. As Patterson and Hennessy said, "the data center is the computer" [24, 34], and their programs are no longer single binaries, but a web of microservices communicating over the network using remote procedure calls (RPCs). Designing high-performance and reliable warehouse-scale systems is among the most important challenges today facing systems architects.

Due to the complexity of distributed systems, being able to understand their behavior and performance is critical. Engineers rely on real-time monitoring, logs, tracing, and profiling, spawning an entire industry around cloud-scale

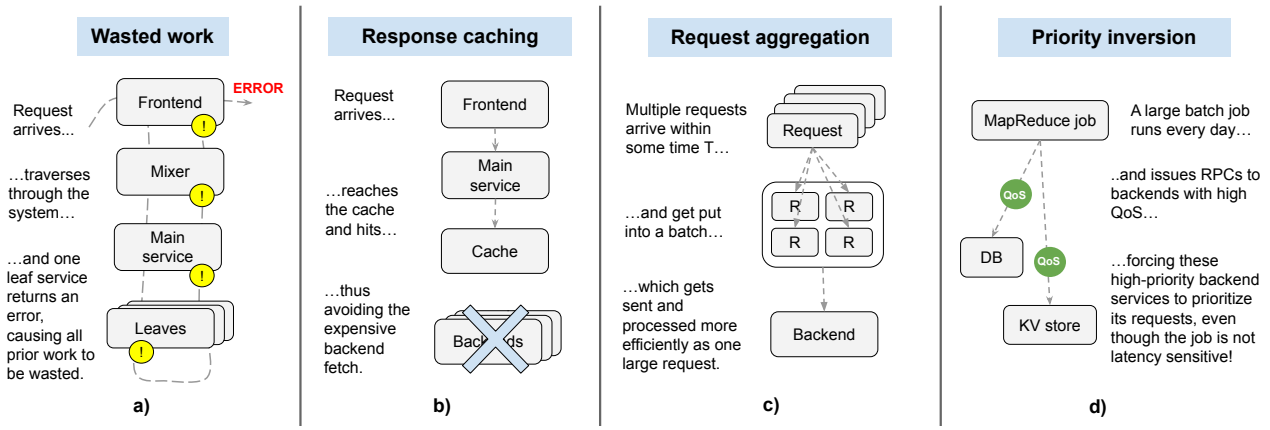


**Figure 2.** Examples of distributed system behaviors that QProf can answer. a) How much work is being wasted when RPCs return errors? b) How many resources can be saved by adding a cache in the middle? c) How many resources can be saved by processing requests as a batch? d) How many batch jobs are issuing RPCs with low latency QoS requested, even though those jobs aren't latency-sensitive, and thus are competing with real high-priority jobs for shared resources?

telemetry [1–4, 12, 19, 23]. From an optimization perspective, most of these tools are used to make systems do the same work faster. CPU profiling can identify and optimize hot functions, while distributed traces can identify the causes of slow requests. In contrast, much less attention has been paid to the problem of finding *less* work to do - or better yet, entirely skip. The fastest `memcpy` is the one you don't do; the cheapest search query is the one whose results were cached. This idea of performance optimization by doing less work for the same result (instead of just making the same work faster) was coined the "room-at-the-top" [29]. In complex distributed systems, there are plenty of such opportunities, but there are no standard techniques for finding them.

In this paper, we present QProf, a holistic profiler which can illuminate many types of "room-at-the-top"-style opportunities in warehouse-scale systems. QProf is built on top of distributed Dapper tracing [37], which works by sampling a small fraction of requests to capture additional metadata like RPC latency, developer-inserted annotations, and more. Traces are typically used for debugging latency, but in this paper, we adopt them as the basis of a profiler for cost metrics like CPUs, TPUs, disk, and more. Cost metrics are fundamentally different from latency because they can simply be summed up, whereas latency must account for parallelism. By attaching cost-per-query data into traces, QProf is able to measure *transitive* query cost - the cost of a query and any subsequent queries sent to its backends (and their backends) - at any layer of a distributed system. Just like many CPU profilers can show flame graphs for individual services [1–4, 23], QProf provides RPC callgraphs for distributed systems to visualize resource usage (Figure 1).

The key idea of QProf is measuring per-RPC cost, attaching it to the corresponding trace span, and summing the total cost of the request when assembling the whole trace. However, there are many problems with this simple framing:

- Cost profilers must have high coverage in order to be broadly useful, but achieving high coverage across thousands of diverse microservices without manual instrumentation is difficult.
- Estimating absolute resource usage from traces requires dependable sampling semantics, which can be thwarted by non-uniform sampling.
- Tracing overheads will skew the data unpredictably.
- Applications can introduce tracing-dependent behavior, making any data derived from those traces completely unrepresentative.

QProf solves all of these problems by radically re-thinking how distributed tracing interacts with applications. **First**, we built an entirely new type of tracing called *skeletal tracing*. This is an ultra-low-overhead tracing mechanism specifically designed to enable accurate profiling of distributed systems at scale. As the name implies, skeletal tracing is very barebones: each trace contains only the minimal amount of data needed to support cost profiling and nothing else. Skeletal tracing is invisible to applications, preventing code from behaving differently when traced vs not, and this ensures that the traces capture representative behavior. **Second**, we integrated an ambient per-query CPU profiler with tracing for *all* C++ and Java services (which comprise nearly all CPU usage) to automatically capture CPU usage data. **Third**, we designed new APIs to model request batching behaviors in traces. These APIs provide well-defined sampling semantics which allow us to reliably measure the cost of batches in the fleet. **Fourth**, we built an offline pipeline to process trillions of spans from billions of traces every day, creating a huge database of fleetwide profiling data for developers to query.

QProf has been deployed to every single job in Google's planet-wide datacenter fleet, and over the past several years, developers have found a myriad of use cases. Figure 2 illustrates some interesting questions we will explore, ranging from wasted work to priority inversions. While some of these capabilities may have been possible to achieve with existing tools, many of them require bespoke instrumentation and are generally system-specific. In contrast, QProf requires no production changes by service owners and works for every service in our fleet. To the best of our knowledge, this is the first work to present the design, deployment, and validation of a tracing-based profiler for warehouse-scale systems.

In short, our contributions are:

1. A discussion of all the subtle reasons why building a cost profiler with distributed tracing is complex.
2. Skeletal tracing: a lightweight, invisible tracing mode which provides reliable, accurate cost data.
3. An integration between tracing and an ambient profiler to accurately measure CPU cost per query without any manual instrumentation.
4. The careful design of fan-in APIs to support cost profiling of batched requests within the tracing data model.
5. Deployment and validation of a trace-based profiler in warehouse-scale systems.
6. A set of case studies showing how QProf has been used to find and quantify novel optimizations in distributed systems, saving millions of CPU cores, thousands of TPUs, and many petabytes of disk storage.

## 2 Distributed tracing overview

In this section, we provide a brief overview of distributed tracing, focusing on the concepts which are important for building a distributed systems profiler.

### 2.1 Trace structure

A trace represents a distributed computation. For example, a search query would produce a trace containing information about the behavior of every service that participated in fulfilling the query. A trace is uniquely identified by a 64-bit trace ID, which is assigned to a top-level request and propagated across threads and RPCs.

Traces are composed of *spans*. A span is a logical unit of work in the trace. Every span is assigned a 64-bit span ID (unique within the trace). Each span also carries its trace ID and its parent span ID to support trace assembly, as well as vital debugging information like the RPC name, latency of the query, job attributes, timestamped annotations, and tracing metadata. Dapper traces are trees[1] of spans.

### 2.2 Sampling of traces

Since tracing every RPC on every service would be prohibitively expensive, we sample requests for tracing. Traces are used for a wide variety of purposes and scenarios, so there are a variety of sampling mechanisms. In this paper, the following two categories are most relevant:

- *Uniform*: a query will be sampled with probability 1/N. This allows one to make strong statistical statements about average population behavior.
- *Forced*: this allows code to manually force a query to be sampled based on arbitrary business logic. This is particularly useful when developers want to trace rare but specific behaviors which are unlikely to be uniformly sampled.

Sampling decisions are generally made at the *root* of a request and propagated downwards, so that requests which weren't already sampled will remain so. However, services in the middle can override this decision. For instance, a root service might be configured to sample 1/100k requests, but a mixer service wants to trace 1/1000 requests, regardless of whether they were already sampled. This can create partial traces which start in the middle of the request's processing. As shown in Figure 3, partial traces include the middleware service and all its dependencies, but none of its ancestors. The existence of such traces is important because as we will discuss, they impact the kind of data that is available and usable for fleetwide profiling.

### 2.3 Instrumentation API

All of the tracing metadata described above – trace/span IDs, sampling decisions and probabilities, etc – are stored in a structure called the TraceContext. A TraceContext is the code representation of a span. When a service receives a request, it will deserialize all of the accompanying tracing metadata, create the TraceContext, and install it into the current thread by copying it to thread-local storage (TLS). Through the TraceContext, applications can check whether a request is traced, record tracing annotations, and create child spans. Asynchronous callbacks can *capture* (create a copy of) the TraceContext and store it alongside the `std::function` to be executed. When that functor is eventually executed, it first installs the TraceContext into TLS.

The lifetime of a TraceContext determines the latency of the span it represents, but callbacks can make copies of a TraceContext, and it's unclear which copy will be the last one, as async work can be delayed indefinitely, in principle. Therefore, TraceContexts are reference-counted to track the number of copies created. The span latency is recorded as the delta between the destruction (of the last copy) and construction (of the first copy) of the trace context. All tracing data associated with the TraceContext is sent when the TraceContext is destroyed.

Finally, users can create custom spans to indicate a smaller chunk of work that they want to track using the LocalTraceSpan API. This is a scoped object which creates a corresponding TraceContext and installs it in the current thread. When

[1]Traces can be forests, but most well-formed traces have a single root.

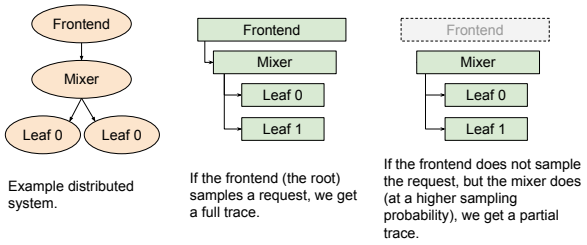


**Figure 3.** Root vs non-root sampling for a hypothetical service. A root-sampled trace produces a full request tree, while a non-root-sampled trace produces a partial trace. A partial trace is missing cost coverage for some services, making it unsuitable for fleetwide cost profiling.

```
Status s = SendRPC();
while (s != OK) {
  LocalTraceSpanOptions options("Retry");
  options.force_trace = true;
  LocalTraceSpan span(options);
  s = SendRPC();
}
```

**Figure 4.** Simplified example of real tracing code in the codebase. If retries account for a large fraction of total traffic, nearly every request will be forced-traced, resulting in no uniform traces for cost analysis.

it goes out of scope, the TraceContext is uninstalled, but the user is free to capture it if necessary as described above. This is also how users can change sampling parameters, e.g. force-trace a request:

```
LocalTraceSpanOptions options(
    "EmailService.FetchMessage");
options.force_trace = true;
LocalTraceSpan span(options);
/* Now a TraceContext has been created and
  installed. Do work. */
```

## 3 Challenges with Profiling Traces

Distributed traces are generally designed to debug latency problems, which is like finding the needle in the haystack. In contrast, cost profilers care about the *haystack* for answering questions like "where is this application spending its CPU cycles". This fundamental mismatch creates several fundamental problems for building a cost profiler on traces:

1. Users have a lot of control over what is or isn't traced, which skews traces away from an unbiased sample.
2. Tracing overhead is highly unpredictable, making CPU measurements of traced queries unreliable.
3. Applications can change their behavior in significant ways when tracing is active, rendering the traces unrepresentative of general behavior.

We will discuss why these issues are important to solve.

### 3.1 User control of tracing decisions

As mentioned in Section 2.2, we offer many mechanisms to initiate tracing. The two main categories are uniform and forced, and either of these can be triggered either at the root of the query or at any point further down in the query execution. However, to build a cost profiler from traces, we assert that the traces must be uniformly sampled and root-initiated:

- Uniform sampling produces an unbiased data population in which every sample has a well-defined statistical weight ($1/sampling\_probability$).
- Root-initiated traces contain the entire distributed computation, while non-root initiated traces will only contain partial data, thus over-representing lower-level systems in the data.

To estimate the total cost represented by a set of traces, we need to apply inverse probability weighting using the following weighted sum (called the Horvitz-Thompson estimator [25]), a process we'll refer to in this paper as "unsampling":

$$total = \sum_{n}^{spans} cost_n * \frac{1.0}{sampling\_probability_n} \quad (1)$$

User control of tracing disrupts this process. For one, a force-traced request doesn't have a well-defined sampling probability. It depends on what fraction of all requests happen to meet the criteria set forth in its custom business logic. This probability is impossible to determine at runtime and infeasible to estimate after the fact by any reasonable means. Furthermore, requests can be sampled by middleware services, rather than at the actual root of the request (like an HTTP server), thus creating a partial trace. For both of these reasons, forced traces must be excluded from the data set.

In the extreme case, this can result in no usable data at all. Consider the real example in Figure 4 we found in the codebase. The code's intent is clear: it wants to ensure traces are available for debugging if the first RPC fails. But if the retry rate is high, then most of the traces produced by this application are force-traced, which cannot be used for statistical analysis. This renders most of the cost incurred by this code invisible to a tracing-based cost profiler.

In summary, we need a population of complete and uniformly sampled traces for cost analysis, but the existing tracing APIs allow users to influence tracing decisions in ways which render traces unusable.

### 3.2 Tracing instrumentation overhead

All tracing systems introduce overhead. For every annotation, CPU must be spent to generate it, memory needs to be allocated to store it, and that data is held in memory until it can be sent to tracing backends. To reduce this overhead, most annotations are only generated if the current request is traced. This means that traced requests are inherently more expensive than non-traced requests (i.e. $cost_n$ in Equation 1 is higher), so the aggregated cost data derived from tracing will be larger than the true value. This isn't a huge issue if the overhead is small and predictable, but tracing overhead depends on how many annotations a particular query added and how expensive those annotations were to generate. This is not possible to correct after the fact at scale, making the resulting data unreliable across systems.

### 3.3 General observer effects

An observer effect is when the act of observation changes the thing being observed. Adding annotations only for traced requests is one example, but there are much more insidious possibilities. Because applications can detect when they're being traced, they can do essentially anything – including making RPCs – but only when traced. This means the trace may contain spans that otherwise would not exist if the request weren't traced! As a concrete example, many applications will log additional metadata to other systems about a query when the query is traced, for later debugging, as shown below:

```
void LogIfTraced(TraceContext ctx) {
  if (ctx.is_traced()) {
     Metadata md = /* create metadata */;
     // This sends an RPC to a service
     // named Logger.Log.
     WriteToQueryLogsViaRPC(md);
  }
}
```

As a result, every trace will contain an RPC to Logger.Log. If we treat the traces as representative of overall behavior, we'd conclude that all queries will call Logger.Log, when in reality, it's only the tiny fraction of traced requests. If the sampling probability of these requests is 1%, then each trace will have a weight of 100, which means when we estimate total cost based on traces, the cost of Logger.Log would be overestimated by 100x! This is not merely a theoretical problem: it actually exists in our services and originally caused our data to be orders of magnitude too high in these cases.

### 3.4 Lack of support for batching

Dapper traces represent distributed computations as a tree, in which work from upper-level services is fanned out to multiple lower level services. This behavior is still the most common in large distributed systems (likely because it is easy to understand, implement, and monitor). However, many services will merge multiple requests and process them together as a single request. This is called batching, and it creates a "fan-in" trace structure (Figure 5). This poses two challenges for cost profiling: 1) how to "split" the batch's cost across the individual requests inside, and 2) how to define precise sampling semantics.

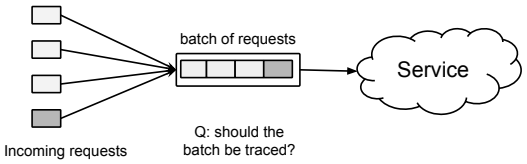


**Figure 5.** A group of four requests, one of which is traced (in dark gray), gets handled as a single batched request in the service. Should that batched request be traced, and if so, what would the sampling probability be?

Figure 5 illustrates the problem of sampling. If a set of independent requests are batched together and sent as one new batched request to a server, should that new batched request be traced? When we surveyed applications that used batching, most would simply trace the batch if any of the requests were traced. In principle, we could compute the probability of tracing the batch like so:

$$p_{batch} = 1 - \prod_{i}^{N} (1 - p_i) \tag{2}$$

where $p_i$ is the probability of tracing the $i$-th request in the batch. But determining $p_i$ in practice is impossible for two reasons: 1) if a request wasn't traced, there's nothing that stores $p_i$ so we don't know its value, and 2) requests which are non-uniformly sampled (e.g. force-traced) don't have a well-defined probability. Thus, in order to unsample the cost of batched requests, we must have well-defined sampling semantics which don't depend on whether the requests inside were sampled or not.

## 4 QProf: A Trace-Based Cost Profiler

QProf is a profiler for distributed queries. The fundamental idea is simple: a trace represents a single distributed computation, so if we can measure the cost of each span within the trace, we can simply record that cost to the trace and sum them up across the billions of other traces that are recorded on a daily basis.

To overcome the fundamental problems elucidated in Section 3, we built *skeletal tracing*, an invisible, ultra-lightweight tracing mode which keeps the active tracing overhead as low as possible and prevents applications from knowing whether they're being traced. In addition, we also built two new features: native per-query CPU profiling and fan-in APIs that allow traces to model batching and queuing systems. Finally, we built an offline postprocessing pipeline which processes

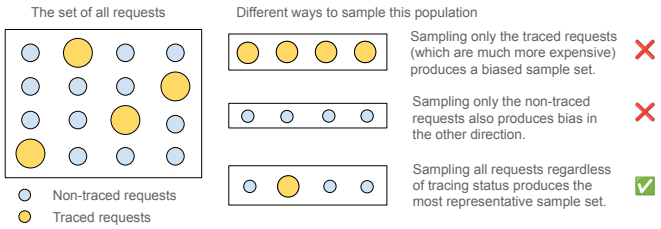


**Figure 6.** An example population of requests. Yellow requests are traced and more expensive due to overhead. The most representative way to sample them for accurate cost estimations is to ignore their tracing status. The other two methods are unrepresentative and result in either overestimation or underestimation of cost.

trillions of spans every day, turning them into a set of aggregated, "RPC-stacks" which can be sliced and diced to produce interesting cost profiles of distributed queries.

### 4.1 Skeletal tracing

Skeletal tracing is an invisible tracing mechanism which produces a "bare-bones" trace, containing only the trace structure, a small set of critical annotations, and nothing else. In aggregate, skeletal traces define a statistically unbiased subset of all query behavior, enabling reliable aggregate measurements of cost. Skeletal tracing supports C++, Java, Python, and Go, which together comprise essentially all services in the fleet. It is characterized by several notable properties: independent sampling, invisibility, minimal annotations, and limited APIs.

**4.1.1 Independent random sampling.** Consider the population of queries shown in Figure 6. If we want to uniformly sample four queries which represent the overall population, we should select three untraced requests and one traced request. Any other sampling outcome - like selecting only the traced requests or only the untraced requests - will lead to a skewed sample. This is the sample of requests that are selected by skeletal tracing. In order to produce this result, skeletal tracing is an *independent* uniform sampling mechanism, unrelated to any other tracing decision.

Therefore, skeletal tracing defines a population of requests that can be reliably used for statistical analysis. Note that it's entirely possible and intentional for some requests to be both "normally" traced *and* "skeletally" traced. These are simply "normal" traces (with all annotations) that also happen to be included in the skeletal population. These sampling decisions are all recorded inside the TraceContext.

Finally, all skeletal traces are sampled at the root of the request tree. Users cannot request a skeletal trace to start in the middle. This guarantees that every skeletal trace represents the entire distributed computation.

**4.1.2 Invisibility.** Compared to "normal" tracing, the status of a skeletally traced request is invisible to the application. The API `TraceContext::is_traced` will always return false, and all public annotation APIs are no-ops. This means that the resulting trace consists of mostly the spans and nothing else. All the other work associated with tracing is elided, reducing the runtime overhead to just span creation (which is very cheap).

**4.1.3 Critical annotations.** Of course, we need to write *some* information (notably cost-per-query), or the traces wouldn't be very useful. We implemented restricted APIs using the passkey idiom, allowing only certain code (like RPC frameworks and tracing libraries) to add critical annotations like CPU/TPU/memory/etc. cost. This does limit the utility of skeletal traces to cost profiling and high-level latency debugging. Skeletal traces do not replace normal traces.

**4.1.4 Limited user control.** To prevent applications from introducing observer effects, clients cannot alter skeletal tracing decisions via any public APIs. All the usual tracing options (like force-tracing) do not apply to skeletal tracing. There is one exception: applications can set the global skeletal tracing rate for requests that they initiate, so that low-QPS services can elect to (uniformly) trace more requests.

### 4.2 Per-query CPU profiling

CPU usage is one of the most important resource metrics that profilers should support. To measure the cost of a single RPC, a naive solution might be to instrument the code, like so:

```
void RunQuery(const Request& r) {
   // Records the current trace_id.
   ScopedCpuMonitor monitor(
       TraceContext::Current());
   // Process the request.
   // When the function exits, the monitor
   // will write the value to the trace
   // before being destroyed.
}
```

However, achieving 100% cost coverage via manual instrumentation at scale is virtually impossible. A single query might traverse through dozens of microservices owned and maintained by different teams around the world, and transitive cost is only meaningful if *all* the downstream dependencies are covered. And even if most services use shared RPC handling frameworks, any asynchronous work which gets run outside of the primary RPC handler would need to be separately instrumented. Over millions of lines of code, it is infeasible to manually instrument them all.

Instead, we solve this problem by integrating tracing with an always-on ambient CPU profiler. This CPU profiler – which has existed for over a decade – runs continuously

in the background of every process. It registers a userspace per-thread POSIX timer which fires an interrupt every 10 ms of CPU time and captures a stack trace. It can measure the *self* (not transitive) costs of every RPC by "tagging" CPU usage data with the currently active RPC method name. Also, it does not care if RPCs are traced or not, so it is unaffected by the tracing-induced problems described in this paper.

Our key insight was to add a simple atomic counter to the TraceContext (which is a per-query data structure). The profiler's interrupt handler will increment this counter for whichever TraceContext was installed in the current thread. This only happens for traced requests, but over time, expensive RPCs will accumulate more interrupt ticks than cheaper RPCs. Since our RPC frameworks and scheduling libraries ensure that the TraceContext is correctly propagated and installed, and all of our services *must* use these libraries, we know that all CPU usage spent on query processing can be attributed to a particular TraceContext.

Unlike other solutions in the industry [19, 23], our integration between tracing and CPU profiling writes the CPU data directly to traces, rather than tagging CPU profiling samples with high cardinality `trace_id`s and `span_id`s. At warehouse-scale, tagging CPU samples like that would balloon the total size of the profiling database, because many identical CPU samples can no longer be aggregated. It is also wasteful: > 99.9999% of collected traces are never individually viewed, and so preserving that much detail in CPU samples is unnecessary. In addition, joining two large tables based on *trillions* of unique trace/span IDs at query time would be extremely expensive.

At the end of a traced query, this counter is written as a critical cost annotation on the trace. This happens when the TraceContext's refcount drops to zero, indicating that all work (asynchronous or not) for this request has been completed. Note that the end is *not* indicated by when the response was sent. Deferring work to be run asynchronously after the response is a common latency optimization.

Since sampling decisions are propagated across RPC boundaries, every span in a trace will have CPU cost information included[2]. Since the sampling frequency is coarse, the measurements of individual spans are very noisy. In fact, most spans will have a CPU cost of zero because the profiler did not fire during the span's lifetime (a limitation which exists in other solutions [23]). But over trillions of spans collected every day, we can build an accurate statistical picture of how much CPU is consumed by a particular RPC method.

In practice, any cost measurement can be recorded to a trace, not just CPU time (this just happens to be the hardest to implement). Many services will report application-specific cost data to a central monitoring system for every query (prominent examples include HDD and SSD ops, TPU usage, and network traffic), but the monitoring system only stores aggregated data. To preserve per-query cost granularity, we instrumented these cost reporting APIs so that all cost measurements are automatically logged to traces.

[2]For C++ and Java services only. The background CPU profiler has very limited support for Go and Python, so these services will report zero CPU cost, but they account for a negligible fraction of CPU in our fleet.

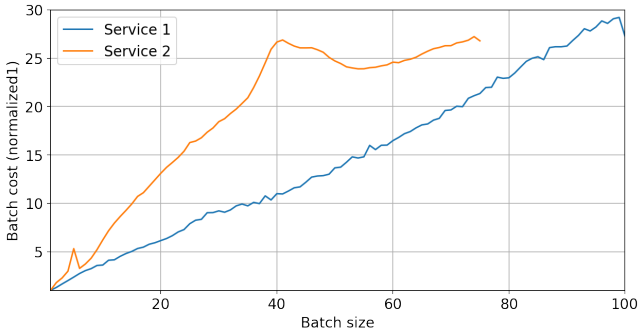


**Figure 7.** For the two largest users of batching, the average cost per batch is linearly proportional to the number of requests in the batch. Some services may experience non-linear behavior at larger batch sizes; however, such large batches are rarely observed in practice, as we will show later.

### 4.3 Batching APIs

To solve the batching problem, we designed the BatchedTraceContext API. A BatchedTraceContext creates and stores a new TraceContext which represents a batched operation. As shown in Figure 8, this new TraceContext is a root context (it has no parent). When it's created, we first flip a uniform sampling coin with a pre-defined probability (e.g. 1/10k). If the batch is sampled, then this TraceContext can be used for cost profiling, because it's the result of a fair coin flip. We then start adding requests to the batch. This does two things. First, it adds a logical *link* between two distinct requests (which have different trace_ids). Second, if the batch wasn't already sampled, but one or more of the requests were, then we retroactively make the batch *non-uniformly* sampled. This preserves the existing behavior expected by applications – which is that a batched request is traced if any of the requests inside are traced – while preventing the batch from polluting aggregated data.

To distribute costs of a batch back to the parents, we make a key simplifying assumption: the cost of a batch is linearly proportional to the batch size (i.e., cost of RPCs inside a batch is equal on average). Figure 7 for the two largest services using batching shows that this is generally true up to a point. Batch sizes (in number of requests) are driven by traffic volume, so we can use a QPS-based process to perform batch cost attribution. During the trace transformations portion of the postprocessing pipeline (see next section), we randomly sample one of the skeletally traced parents that are linked to the batch operation (assuming one exists), weighted by the inverse sampling probability. This process produces a fair

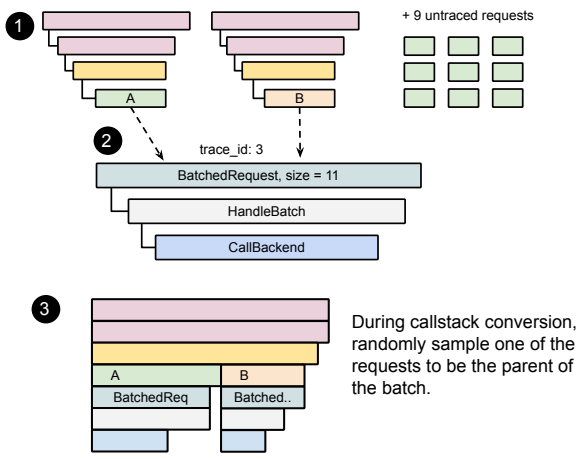


**Figure 8.** When a system uses batching, 1) QProf collects the initial requests (A+B), 2) creates a new trace span representing the batch and links A and B to the batch span, and 3) randomly samples one of the traced requests to be the batch's parent during callstack conversion. This allows cost to flow from the roots of A and B to the leaf services.

sampling over the parent trace's QPS distribution. Then we "stitch" the batched trace into the parent trace, merging two distinct traces into one (see Figure 8, step ❸). When these callstacks are aggregated, we can statistically distribute the total batch cost over all the parents, without needing to split the cost of any one batch.

The BatchedTraceContext API is designed primarily for batching a relatively small number of RPCs which all look alike. This is not always the case. In large systems, background work like garbage collection or database compaction often happens periodically. In a sense, these are all batched operations, but precise client attribution may be infeasible. For example, compactions are driven by write-heavy clients, but compactions happen relatively rarely, with thousands of writes having been committed in between. Using BatchedTraceContext to hold thousands of TraceContexts in memory could incur substantial memory overhead. In these cases, this API is not suitable, and systems will need to design their own background cost attribution methods. Some will track total QPS per *client user* (instead of per request) and perform chargeback proportional to traffic volume. Others simply pay for these background costs themselves without trying to charge clients at all.

### 4.4 Postprocessing pipeline

The pipeline transforms the trillions of spans collected daily into a smaller dataset that can be interactively queried in just a few seconds. There are three phases:

**4.4.1 Callstack expansion.** The data arrives in the form of one row per span (with columns like trace_id, span_id, parent_span_id, cpu_time, etc). This is transformed into a set of RPC callstacks, going from every span to the root as shown in step 2 of Figure 1. If the average depth of a span was M, and there are N spans, then callstack expansion will produce O(N*M) rows of data. This is a huge expansion of the total data volume, which we will shrink later on.

**4.4.2 Unsampling weights.** For each sample, we compute a statistical weight, indicating how many queries from the total population are represented by this sample. We care mostly about the *rate* of resource usage – e.g. we think about CPUs rather than CPU-seconds – so we convert the weight into a rate by dividing by seconds-per-day:

$$\text{weight} = \frac{1/\text{sampling probability}}{60 * 60 * 24} \tag{3}$$

The total average daily cost of a set of spans is thus the following weighted sum:

$$\text{total average cost} = \sum_{i}^{\text{N spans}} \text{weight}_i * \text{cost}_i \tag{4}$$

Similarly, the QPS for any span can be computed simply by summing up the weights. This is a very powerful property, because standard monitoring libraries only report QPS for RPCs, but QProf can report QPS for any operation in a trace.

**4.4.3 Trace transformations.** The pipeline runs the batched operation stitching procedure, as described in the previous section. In addition, it also does a variety of other trace transformations to handle some special cases, which we won't discuss for brevity.

**4.4.4 Callstack aggregation.** Thus far, the data volume is much larger than it was initially due to callstack expansion. To reduce this down to a manageable size, we note that 1) most spans report a cost of zero and thus don't contribute to the weighted sum, and 2) many traces/spans are nearly identical in structure. Thus, we split the data into two parts: one for spans which contain zero cost, and one for all the others. The costly spans are preserved as-is. The zero-cost spans are first downsampled (e.g. at a rate of 10%). For each of the remaining spans, we compute a callstack signature, which contains the most important and discriminating information (like span name and job metadata), while dropping high cardinality fields like build commit hashes and averaging certain non-cost metrics like latency. Downsampling these spans (as opposed to dropping them altogether) preserves the ability to compute QPS (which is just the sum of weights). We group all these samples by callstack signature, sum up their weights, and scale up by the downsampling factor. This process reduces overall storage by 10-20x with no impact on cost metric accuracy.

## 5 Results

QProf has been deployed at scale to every job running in our datacenters around the world. It has been operating for many years, serving a myriad of use cases. In this section, we evaluate the accuracy of QProf in measuring QPS and CPU usage and the improvements in accuracy metrics provided by skeletal tracing. Comparison of QPS accuracy is useful because QPS is not affected by tracing overhead, allowing us to isolate how much improvement comes from observer effects and independent sampling.

Our source of ground truth is the background CPU profiler described in Section 4.2, which provides accurate self-cost data. It is impossible to validate transitive CPU cost at scale (if such a profiler existed, then we wouldn't need QProf). Our evaluation is based on the assumption that if self cost is accurate and trace structures are unbiased, then transitive costs are also accurate.

Since QProf is RPC-centric, we compare numbers against the actual QPS and CPU usage for every unique RPC profile. An "RPC profile" is represented by the tuple (RPC name, job user). The RPC name generally looks like `Service.Method`, and the job user is the role that is running the server which hosts the RPC method. RPC names are not unique, but the combination of RPC name and job user is. There are O(10k) such RPC profiles in the data. For each RPC profile, we compute the error between QProf and ground truth (from the background profiler described in Section 4.2). We use the symmetric error function given in Equation 5:

$$error = \begin{cases} \frac{QProf}{Actual} - 1, & QProf >= Actual \\ -(\frac{Actual}{QProf} - 1), & Actual > QProf \end{cases} \quad (5)$$

This produces an intuitive result: if QProf is 10% higher than reality, it will report 0.1, and conversely, if QProf is 10% lower than reality, it will return -0.1.

Figure 9 shows the distribution of QPS measurement error over all RPC profiles. If QProf were perfect, we'd expect a single vertical line at error = zero (100% of RPCs have zero error). Of course, QProf is not perfect, but it is still very accurate at measuring QPS. With normal tracing, 65% of RPC profiles are measured within 10%, but this rises to 73% of RPC profiles with skeletal tracing. Both distributions have peaks at zero error, but with normal tracing, this is merely 40% of RPC profiles, vs 52% for skeletal tracing. These improvements are driven by eliminating observer effects and independent sampling, and even if they may seem modest on the surface, their effects compound for CPU accuracy. Importantly, these improvements are not simply because skeletal tracing produces more samples. In fact, there are actually *more* normal traces than skeletal traces, because users can freely change their normal tracing rates. The effective

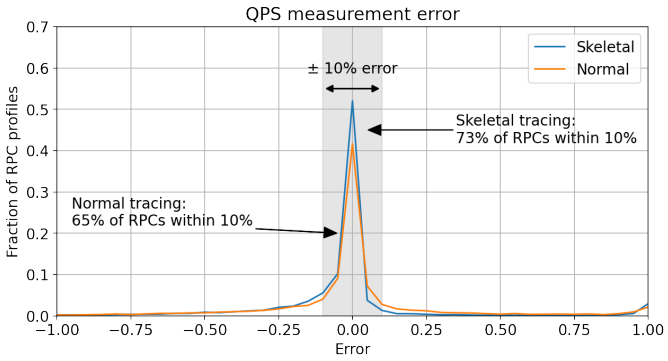


**Figure 9.** QProf is very accurate at measuring QPS. Skeletal tracing shows a tighter error distribution due to the elimination of subtle observer effects. The gray band shows the error range of ±10% error.

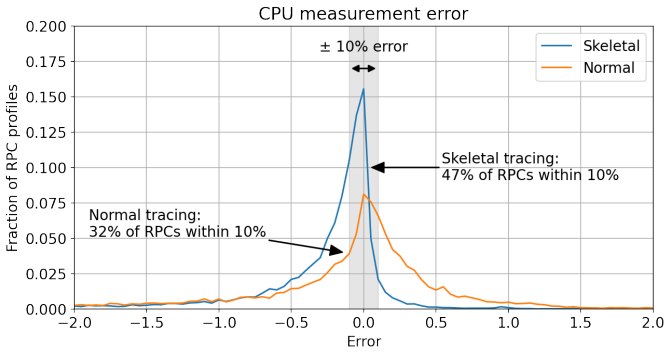


**Figure 10.** Skeletal tracing is much more accurate at measuring CPU usage than normal tracing, due to the significantly reduced tracing overhead and elimination of observer effects. It is not as accurate as QPS, since this measurement is the product of *two* random variables rather than just one. The gray band shows the error range of ±10% error.

fleetwide normal tracing rate is ≈ 1/4k, which is 2.5× the skeletal tracing rate of 1/10k.

When error is negative, QProf's measurement is too low. This can be caused by a confluence of possible reasons:

1. Throttling: services can force-trace all of their requests, flooding the tracing collectors with traffic. Collectors respond to overload by dropping data from abusive users. Although skeletal traces generally cannot cause traffic overloads, the data can still sometimes be lost in the crossfire.
2. Tracing API misuse: In normal tracing, users can force-trace their requests and/or do any number of things which prevent us from using that data in QProf (see Section 3.1). This is not a problem for skeletal tracing.

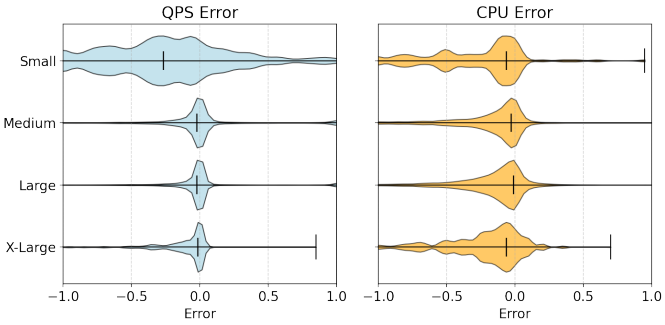


**Figure 11.** With skeletal tracing, most of the underestimation comes from the "small" services. The combination of low traffic volumes and spiky/unpredictable traffic patterns leads to higher sampling error. However, small services account for only 2% of the total RPC profiles, so their effect on overall error is minimal. Medium and large services make up over 90% of total distinct RPC profiles.

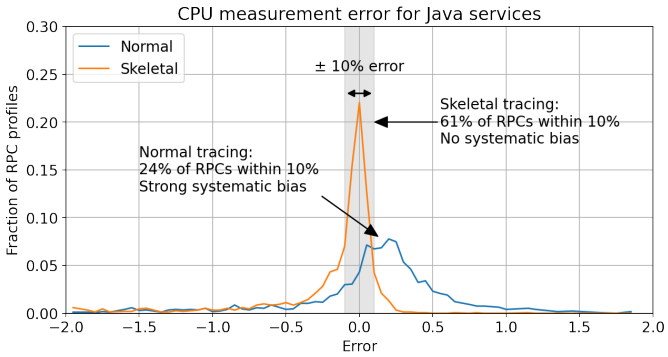


**Figure 12.** Due to reduced tracing overhead from skeletal tracing, data quality for Java services improved significantly.

3. Huge traces: we attempt to assemble an entire trace in-memory on a single machine[3] to run its analysis. This naturally imposes an upper limit on trace size. Huge traces are usually the result of user error. A few systems routinely generate traces which are much larger than this. In these cases, we can insert additional instrumentation to statistically prune part of the trace, preserving the ability to unsample cost while keeping overall trace sizes manageable.

When error is positive, QProf's measurement is too high. The most common reasons are observer effects and tracing API misuse. Services which do work conditionally based on whether a request is traced or not are more likely to generate tracing-induced RPC traffic, thus increasing the perceived QPS. This does not significantly impact skeletal tracing, due to its intentional API designs.

Figure 10 shows the error distribution for CPU usage measurements. In normal tracing, only 32% of RPC profiles are measured to within 10% of ground truth, vs 47% for skeletal tracing (a relative increase of 50%). The gap in accuracy between QPS and CPU measurements is largely caused by services which consume very little CPU. In these services, it's much rarer for the CPU profiler to sample on a traced request, leading to wider margins of error. The improvement in accuracy stats from skeletal tracing may not seem impressive, but it doesn't capture the huge improvement in usability for several very important RPCs (like core storage and database systems) whose data used to be 100× too high due to observer effects.

[3]Sharding trace assembly across machines is possible, but the benefits of supporting huge traces are unclear.

Also, note how in normal tracing, the error distribution is generally symmetric, with a small bias towards overestimation. In comparison, skeletal tracing has essentially eliminated any systematic overestimation error. Ironically, this makes the underestimation problem in skeletal traces look *worse* than normal traces. In reality, the normal traces simply have more sources of error that sometimes cancel out.

We also analyze how QProf's error varies by the size of a service. We've classified all RPC profiles into four buckets based on the global QPS handled by the RPC. "Small" services have low traffic volumes, which produce very few samples per day. Many of them exhibit unpredictable and variable traffic patterns, compounding the sampling problem. This is reflected in Figure 11 with the much wider and "lumpier" error distributions. But medium, large, and extra-large services have largely similar error profiles, because services at these scales will typically use global load balancers to evenly spread traffic across jobs and regions, resulting in smooth, predictable traffic patterns which are much easier to sample.

The effectiveness of skeletal tracing is particularly notable for Java services, as shown in Figure 12. With normal tracing, QProf's data is heavily skewed too high. The global mean error was about 15%, and the spread is very wide since applications incur different levels of tracing overhead. But with skeletal tracing, tracing overhead is reduced greatly, which results in an error distribution tightly centered around zero. 61% of RPC profiles now have less than 10% error, compared to just 24% before.

We also note that there are other important metrics like latency and lock contention. We focused our evaluation on QPS and CPU primarily because in our experience, they are by far the most widely used metrics. Furthermore, QProf is a cost profiler, but latency metrics are not directly summable. We are currently implementing support for critical path latency and lock contention metrics in QProf, but there is no universally accepted definition of lock contention to serve as a ground truth.

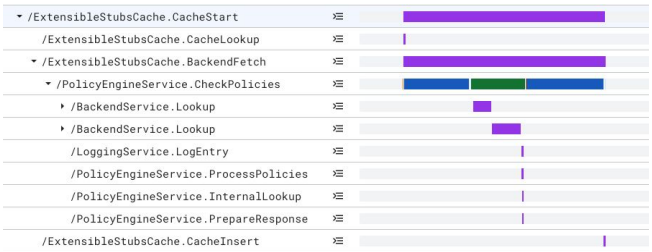


**Figure 13.** We designed a trace structure which groups major caching operations under specific spans, making it easy to generically measure the costs of lookup, miss, and insert. In this example of a real caching trace, the initial cache lookup misses and triggers a backend fetch.

These results demonstrate that tracing overhead and observer effects are significant contributors to cost measurement accuracy through traces, but the level of impact varies widely across applications. We argue that this justifies our "bottom-up" approach of reducing tracing overhead directly at the source, rather than correcting the error after the fact by trying to estimate and subtract the overhead.

## 6 Case Studies

Over the past several years, engineers have used QProf in a myriad of scenarios. The most common use case is individual teams using QProf to estimate additional resources required for new features by measuring average transitive query cost, then scaling that by the expected traffic increase. Another is to measure the resources consumed by a feature or behavior before deciding whether to pursue optimization in the first place. As a fleetwide profiler, QProf can also identify and quantify horizontal optimizations which save significant resources for many teams. In this section, we highlight some particularly interesting use cases described in Figure 2, which exemplify the "room-at-the-top" style of optimizations.

### 6.1 Caching and batching savings

Our RPC framework has support for intercepting output RPCs, at which point customizable modules can be installed and triggered. Two notable such modules are response caching and request aggregation. Both of these modules have existed for some time, but before QProf was available, there was no way to broadly measure their efficiency impact. In this section, we discuss how we used QProf to measure their ROI and expand their impact.

**6.1.1 Response caching.** In the simplest form, the caching module maintains a small cache (either in-memory or remote), keyed by an application-defined request fingerprint. When a new outgoing RPC is made, the caching module will compute the request fingerprint and look it up in the cache. A hit will immediately terminate the RPC and return the cached response, skipping the entire tree of RPCs that would otherwise be sent; a miss will trigger a cache fill operation by making the RPC and populating the cache with its response. For brevity, we omit details about how fingerprints are computed, which requests are eligible for caching, etc.

To measure resource savings, we need to estimate what the cost *would* have been without caching. It's impossible to truly measure the cost of something that didn't happen, but we can make good estimates because if a cache's hit rate is high, then that cache is receiving a lot of identical requests which should cost roughly the same amount of compute. Thus, the average cost of a cache hit and miss is sufficient. We also need to measure total cache lookups and the cache hit rate. To do this, we instrumented the caching code to produce the trace structure shown in Figure 13. This structure groups all the cost associated with a cache miss under a single umbrella span (`BackendFetch`), making the miss cost easy to measure.

Now, the cost of all quantities can be computed using QProf, both per RPC and for the full job. Computing per-job cache hit rates is very important since traffic routing is generally not uniformly distributed (e.g. requests in North America vs Asia tend to differ quite a bit).

1. The QPS of cache lookups is the QPS of the span `Cacher.Start`.
2. The cache hit rate is the ratio of QPS for `Cacher.Hit` to `Cacher.Start`, on a per-job basis.
3. The cost of a cache miss is the transitive cost of the span `Cacher.BackendFetch`.
4. The cost of a cache hit is the cost of the span `Cacher.Hit`.

Then the total per-job savings from caching is given by:

$$\sum_{j}^{\text{jobs}} QPS_j * hit\_rate_j * (miss\_cost_j - hit\_cost_j) \qquad (6)$$

We validated these numbers through careful data analysis of the major caching users. Across the entire fleet, response caching is currently saving millions of cores + HDD ops and thousands of TPUs.

**6.1.2 Request aggregation.** Request aggregation is an optimization for certain workloads which are more efficient when processing a batch of requests at once. On the client side, RPC overheads can be amortized across a group of requests. The primary tradeoff is increased RPC latency, as the client needs to wait for enough requests to form a batch (up to some deadline). Batch-oriented jobs like logs processing and analytics are often good candidates. The aggregator module handles the logic of forming batches and splitting up the batch of responses.

Just like caching, we need to estimate the cost that *would* have been incurred if aggregation wasn't enabled. We estimate this using the average cost of a single unbatched request (aka batch size = 1). However, we cannot rely on every service to occasionally generate a batch of size 1 for

this baseline, because this can bias the baseline cost towards traffic patterns or requests which are less likely to batch. Therefore, the aggregator module will, from time to time, select an already uniformly traced request and flush it in a batch of its own, continuously over time. Using these requests, we can establish an unbiased baseline cost-per-query. With the BatchedTraceContext API, we can measure the size and cost of the batch, allowing us to compute the per-job resource savings:

$$\begin{aligned}\text{Savings} &= \text{Cost with no batching} - \text{Cost with batching} \\ &= \sum_{batch}^{\text{All batches}} (batch\_size * cost_{\text{baseline}} - cost_{\text{batch}}) \quad (7)\end{aligned}$$

Request aggregation is currently saving O(100k) cores. Figure 14 shows that the batch size distribution is quite broad. Interestingly, the most common batch size is actually 1, representing 44% of all requests. This indicates that no other requests were received before the batching timeout expired[4]. Naturally, no resource savings are achieved here. But for the remaining 56%, the average batch size is 23, reflecting the fact that batch sizes need to be relatively large for the savings to be significant.

### 6.2 Finding and removing wasted work

Wasted work in distributed systems has a long history of investigation across many different contexts, ranging from MapReduce stragglers [41] to request hedging [17] and more. In this section, we examine a category of wasted work defined by RPCs which either return errors or empty results. Figure 2a shows how an error in a leaf service can cause all the work done up to that point to be wasted.

**6.2.1 Search results which are never shown.** In web search, a user query may trigger thousands of features to be executed. These features return results which are globally ranked, and only the top N are rendered on the search results page. Given the number of features and limited space, it is common for features to have relatively low rendering-to-triggering ratios while still being profitable, so this ratio alone is not sufficient to mark a feature as "wasteful".

But QProf is able to associate these ratios with their transitive processing cost, allowing us to make statements like "this feature costs 10 CPUs per RPC and often returns an error". Through this lens, we found many features which often returned `not_found` errors, in which the accompanying error messages indicated that no results were being returned at all. We also found many RPCs which returned suspiciously small response sizes, also a good indicator of emptiness. QProf measured the total transitive cost of these features to be O(10k) cores, meaning that all these cores were being spent to simply generate empty responses! By working with the feature teams, we turned down these features altogether to reclaim these resources. This is a novel example of finding and eliminating redundant work which was not simply created from request hedging [17] or retries.

[4]The single-request batches used to measure the baseline cost-per-query are not included in this analysis, since that behavior is tracing dependent and thus would introduce a bias.

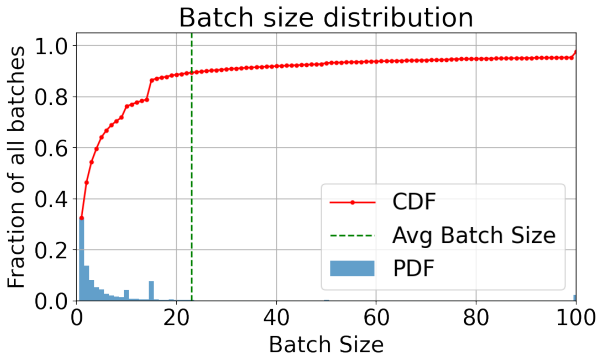


**Figure 14.** Distribution of batch sizes. The most common is size 1 (44%), and the average batch size (for size > 1) is 23. There is a very long tail of batches above size 100.

**6.2.2 Misconfigured RPC deadlines.** We found that a system was wasting TPUs due to misconfigured RPC deadlines. This system receives work from a central queue, and the work items make long-running LLM calls which can take many minutes. The central queue will automatically retry a work item if it does not complete within a deadline, but the initial calls are not canceled. QProf showed that the retry path consumed more TPUs than the initial calls themselves, indicating that the current deadlines were far too short. By doubling the deadlines, we reduced timeouts to nearly zero, saving thousands of TPUs by eliminating redundant work.

**6.2.3 Empty Ads candidates.** A team in Ads was spending O(100k) cores computing sets of promotional ad candidates to present to users. They wanted to understand how many resources are spent in RPCs which don't find any candidates at all, so they instrumented their service with additional local spans to report a `NOT_FOUND` error when this happens. QProf showed that the total cost of computing the empty set was nearly 30% of their service's entire CPU footprint. This was a trivial code change which shed light on a significant optimization opportunity.

### 6.3 Priority inversion

It is standard practice for datacenters to offer different tiers of compute resources with varying SLOs and prices [11, 30, 39, 42]. For example, jobs which serve user-facing traffic are highly latency-sensitive and generally need their CPUs to always be available, while batch processing jobs are typically throughput-oriented and can afford to wait for spare CPUs

to become available. Jobs are assigned priorities, where high-priority jobs can evict low-priority jobs from a machine. High priority resources accordingly cost more than low-priority resources. Any resource – not just CPUs, but RAM, TPUs, disk I/Os, etc – can be allocated and serviced by priority. In distributed systems, *priority inversion* happens when low-priority jobs induce demand on high-priority jobs, thus consuming expensive resources in service of low-priority work. In some cases, this is working as designed – e.g. all MapReduce jobs need to read data from highly reliable low-level storage systems – but there are many ways to tune the usage to reduce the inherent inefficiencies. With QProf, we can easily look for cases of priority inversion.

**6.3.1 High-priority disk for batch jobs.** One common anti-pattern is when batch jobs read a lot of data from distributed storage using high-priority disk requests. High-priority disk requests have lower latency guarantees since they are serviced before all other lower priority requests, but throughput-oriented jobs can generally tolerate slower reads. To find these cases, we instrumented our central disk service (using the restricted APIs) so that the cost of disk operations was annotated in traces. Then, we searched QProf for "high-priority disk usage in which the RPC stack contains a low-priority job". This search found dozens of batch jobs issuing high-priority disk requests. By working with the respective teams, we were able to reduce the requested disk priority for many jobs without impacting any of their performance metrics. While the total disk operations remained the same, this work reduced the overall disk costs charged to those teams and freed up additional capacity of high-priority disk for other applications.

**6.3.2 Mysterious CPU usage surges.** A team running a high-priority video processing service noticed that their CPU costs would suddenly increase for a few days, return to normal, then rise again a few days later. None of their existing telemetry indicated where the cost was coming from. QProf identified the origin as a batch job performing several one-off backfills. This is also a priority inversion problem: a low-priority job consumed significant resources for a high-priority service that was not provisioned to handle it. While standard monitoring telemetry can break down traffic by request originator, in this case the originator pointed to a centralized work queuing system, which obscured the true source of traffic. QProf was able to diagnose this because services run by the originator were present in the *middle* of the RPC callstack. This shows the value of having the full RPC-level callstacks.

# 7 Related Work

Distributed traces have been extensively studied for decades, but the vast majority of prior work focuses on latency optimization [8, 14, 20, 21, 27, 31, 36, 37]. However, using traces for *cost* analysis has received much less attention.

Prior to QProf, there were two ways to measure transitive cost at Google. The primary way is by attaching *tags* to RPCs describing a feature (e.g. "maps_ui=v2"). This gets propagated down the RPC chain, and the background CPU profiler can automatically measure total cost for this tag across all services. While this is easy, every feature needs its own separate tags which creates engineering toil to build and maintain. Engineers must specify data retention policies for the metrics and tags they want to monitor. Tag propagation can be expensive for downstream services, and hierarchical cost breakdowns cannot be provided with tags. The other way is to manually instrument the code to record query cost at every hop and either propagate that data back up the RPC chain or send it to another dedicated cost tracking service for analysis. As discussed in Section 4.2, this is prone to coverage gaps and requires opt-in from all downstream services.

Transitive Resource Accounting (TRA) [7] is conceptually similar to QProf. It also attaches cost-per-query to Canopy traces [27] and aggregates the data offline to produce profiles which can be used for capacity planning, performance optimization, and even power analysis [22]. However, TRA's implementation differs substantially from QProf. Most importantly, TRA does not use lightweight tracing to reduce overhead; instead, it builds models based on historical data for each service to estimate the tracing overhead, which it subtracts from the measured cost. In our experience, such post-hoc corrections may work on a small number of services, but they are very challenging to deploy to the entire fleet given the huge diversity of behavior. In addition, this does not account for observer effects (Section 3.3). TRA does not discuss how its CPU data is captured, but contemporaneous work on transitive *power* analysis suggests that TRA measures CPU by instrumenting request handlers within each application [22]. As we discussed in Section 4.2, any approach which requires extensive manual instrumentation or adoption of a new API is very difficult to enforce at scale. In contrast, QProf uses skeletal tracing to eliminate most sources of overhead/observer effects and ambient per-query profiling to support all applications.

Many tracing systems can be combined with background CPU profilers to enhance debuggability. Grafana, Elastic, Datadog, and Polar all offer the ability to join traces with per-query CPU profiles [12, 16, 19, 23]. Most such integrations work by tagging CPU profiles with trace_ids to enable a separate lookup for profiles for a specific request. As discussed earlier, this tends to balloon the size of the database by defeating callstack compression, and it requires a large join between two huge datasets at query time. This is why QProf

directly writes CPU usage data into the traces instead. These integrations are also generally geared towards interactive debugging of exemplar slow requests, rather than providing an aggregated view of cost for every service in a datacenter. These existing solutions also do not consider observer effects or tracing-induced overheads.

Canopy [27] has extensive support for custom trace processing and different execution models, but to the best of our knowledge, it does not support a trace-wide "verbosity control" that could reduce the data collected to lower runtime overhead. In contrast, QProf relies on skeletal tracing to automatically acquire accurate cost profiles for all C++ and Java services (comprising nearly all CPU cost), while addressing essentially all of the major observer effects that tracing can induce.

The problem of how to represent DAGs in tracing has been studied extensively as well. Systems like X-Trace and Canopy support DAGs by recording events instead of spans, and events can naturally have multiple causal parents [20, 27]. Pivot Tracing introduced the concept of a "happens-before join" relational operator, in which users can query a distributed trace based on causal relationships [31]. The Mystery Machine builds a model of request execution by formulating and rejecting a large number of hypotheses about causal relationships based on a massive set of event logs [14]. OpenTelemetry supports multi-parent spans by logically linking spans from different traces together [15], with batch operations being represented as new root spans linked to all the traces of their requests. But this past work focuses only on how to *represent* a batch in a trace. To the best of our knowledge, we are the first to explore how to properly sample batched operations for cost profiles, as well as propose a solution for attributing cost of batches to the batched requests in aggregate.

Wasted work goes hand-in-hand with building reliable, scalable distributed systems, and there is a rich history of research into mitigating waste in many forms. In MapReduce systems, many papers have tackled the problem of straggler mitigation through backup tasks which trade off duplicated work for lower latency [5, 6, 18, 41]. There has been much practical and theoretical work exploring tradeoffs between redundant work and latency in large-scale systems [17, 32, 33, 38, 40]. Others have analyzed the costs of metastability in datacenters like retry storms, cascading failures, and overload [10, 13, 26]. We build on this history of systems research by going beyond modeling techniques and building a general-purpose profiler which can quantify the real cost of wasted/redundant work. QProf is a tool which realizes the vision of finding the "room-at-the-top" [29].

OpenTelemetry [15] standardized the concept of "baggage", which is essentially metadata that travels with an RPC. This baggage can be used to store information like the request initiator, chargeback user, or application-defined tags like the name of a feature being executed. Coupled with a user-space CPU profiler, this also supports measurement of transitive costs. However, this approach has two primary limitations: 1) all metadata must be propagated on the wire, increasing network costs for all downstream RPCs, and 2) the cardinality of the metadata is often limited because each service needs to keep track of resource usage for every combination of metadata values in memory. QProf supports similar tagging concepts, but since it writes data to traces instead of propagating data down the wire, it has no such cardinality limits and it does not increase on-the-wire costs.

Google-Wide Profiling (GWP) [35] is a profiling system deployed at Google which collects performance counter data from a sample of all machines in the fleet, allowing powerful analysis of warehouse-scale applications [28]. However, it is fundamentally focused on profiling a single process at a time, rather than following the chain of RPCs down a distributed system. QProf can be thought of as a "GWP-for-RPCs", in which entire cloud services are treated as a single "program".

## 8 Conclusion

QProf is a tracing-based cost profiler for warehouse-scale services, providing visibility into transitive cost of complex cloud services. Through skeletal tracing, an ambient per-query CPU profiler, and batching APIs, QProf is able to provide accurate data for thousands of microservices without any manual instrumentation required. Its design solves the many subtle problems of using traces for cost profiling. QProf realizes the vision of a systems profiler which can find the "room-at-the-top", which are optimizations that aim to do less work for the same result. To the best of our knowledge, this is the first work to discuss the design considerations, deployment, validation, and usage of such a profiler in warehouse-scale systems. We have deployed QProf globally in every service in Google's fleet and used it for a myriad of use cases, including cost optimization, capacity planning, regression detection, and much more. Through its insights, we've been able to save millions of CPUs, thousands of TPUs, countless disk operations, and more.

## References

[1] 2024. Datadog: Cloud Monitoring, Security and Analytics. https://www.datadoghq.com/ Accessed: Jan. 2, 2026.

[2] 2024. Dynatrace: Unified Observability and Security. https://www.dynatrace.com/ Accessed: Jan. 2, 2026.

[3] 2024. New Relic: The All-in-One Observability Platform. https://newrelic.com/ Accessed: Jan. 2, 2026.

[4] 2024. Splunk: The Key to Enterprise Resilience. https://www.splunk.com/ Accessed: Jan. 2, 2026.

[5] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. 2013. Effective straggler mitigation: attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation* (Lombard, IL) *(nsdi'13)*. USENIX Association, USA, 185–198.

[6] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward Harris. 2010. Reining in the outliers in map-reduce clusters using Mantri. In *Proceedings of the*

*9th USENIX Conference on Operating Systems Design and Implementation* (Vancouver, BC, Canada) *(OSDI'10)*. USENIX Association, USA, 265–278.

[7] Akanksha Bansal and Richard Cornew. 2022. End-2-End Resource Accounting Leveraging Distributed Tracing. AtScale Conference (Systems and Networking). https://atscaleconference.com/videos/end-2-end-resource-accounting-leveraging-distributed-tracing/ Accessed: Jan. 2, 2026.

[8] Paul Barham, Rebecca Isaacs, Richard Mortier, and Dushyanth Narayanan. 2003. Magpie: online modelling and performance-aware systems. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9* (Lihue, Hawaii) *(HOTOS'03)*. USENIX Association, USA, 15.

[9] Luis Andre Barroso, Urs Hölzle, and Parthasarathy Ranganathan. 2025. *The Data Center as a Computer: Designing Warehouse-Scale Machines*. Morgan & Claypool. https://books.google.com/books?id=cfGkEQAAQBAJ

[10] Betsy Beyer, Chris Jones, Jennifer Petoff, and Niall Richard Murphy. 2016. *Site Reliability Engineering: How Google Runs Production Systems*. http://landing.google.com/sre/book.html

[11] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. 2014. Apollo: scalable and coordinated scheduling for cloud-scale computing. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation* (Broomfield, CO) *(OSDI'14)*. USENIX Association, USA, 285–300.

[12] Frederic Branczyk. 2024. Correlating Tracing with Profiling using eBPF. https://www.polarsignals.com/blog/posts/2024/03/05/correlating-tracing-with-profiling-using-ebpf. Accessed: 2026-03-25.

[13] Nathan Bronson, Abutalib Aghayev, Aleksey Charapko, and Timothy Zhu. 2021. Metastable failures in distributed systems. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (Ann Arbor, Michigan) *(HotOS '21)*. Association for Computing Machinery, New York, NY, USA, 221–227. doi:10.1145/3458336.3465286

[14] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. 2014. The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. USENIX Association, Broomfield, CO.

[15] Cloud Native Computing Foundation. 2024. OpenTelemetry: High-quality, ubiquitous, and portable telemetry to enable effective observability. https://opentelemetry.io/ Accessed: Jan. 2, 2026.

[16] Datadog. [n. d.]. Investigate Slow Traces or Endpoints. https://docs.datadoghq.com/profiler/connect_traces_and_profiles. Accessed: 2026-03-24.

[17] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. doi:10.1145/2408776.2408794

[18] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (Jan. 2008), 107–113. doi:10.1145/1327452.1327492

[19] Elastic. 2026. What is Continuous Profiling? | A Comprehensive Continuous Profiling Guide. https://www.elastic.co/what-is/continuous-profiling. Accessed: 2026-03-24.

[20] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: A Pervasive Network Tracing Framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[21] Yu Gan, Yanqi Zhang, Kelvin Hu, Dailun Cheng, Yuan He, Meghna Pancholi, and Christina Delimitrou. 2019. Seer: Leveraging Big Data to Navigate the Complexity of Performance Debugging in Cloud Microservices. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 19–33. doi:10.1145/3297858.3304004

[22] Alexander Gilgur, Brian Coutinho, Iyswarya Narayanan, and Parth Malani. 2021. Transitive Power Modeling for Improving Resource Efficiency in a Hyperscale Datacenter. In *Companion Proceedings of the Web Conference 2021 (WWW '21 Companion)*. ACM, Ljubljana, Slovenia. New York, NY, USA.

[23] Grafana Labs. [n. d.]. Use Traces to profiles. https://grafana.com/docs/grafana-cloud/monitor-applications/profiles/traces-to-profiles/. Accessed: 2026-03-24.

[24] John L Hennessy and David A Patterson. 2012. *Computer architecture: a quantitative approach* (5 ed.).

[25] D. G. Horvitz and D. J. Thompson. 1952. A Generalization of Sampling Without Replacement from a Finite Universe. *J. Amer. Statist. Assoc.* 47, 260 (1952), 663–685. doi:10.1080/01621459.1952.10483446

[26] Lexiang Huang, Matthew Magnusson, Abishek Bangalore Muralikrishna, Salman Estyak, Rebecca Isaacs, Abutalib Aghayev, Timothy Zhu, and Aleksey Charapko. 2022. Metastable Failures in the Wild. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 73–90. https://www.usenix.org/conference/osdi22/presentation/huang-lexiang

[27] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, Vinod Venkataraman, Kaushik Veeraraghavan, and Yee Jiun Song. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA.

[28] Svilen Kanev, Juan Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2014. Profiling a warehouse-scale computer. In *ISCA '15 Proceedings of the 42nd Annual International Symposium on Computer Architecture*. 158–169.

[29] Charles E. Leiserson, Neil C. Thompson, Joel S. Emer, Bradley C. Kuszmaul, Butler W. Lampson, Daniel Sanchez, and Tao B. Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744. doi:10.1126/science.aam9744

[30] Shaohong Li, Xi Wang, Xiao Zhang, Vasileios Kontorinis, Sreekumar Kodakara, David Lo, and Parthasarathy Ranganathan. 2020. Thunderbolt: throughput-optimized, quality-of-service-aware power capping at scale. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 70, 15 pages.

[31] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA.

[32] Michael Mitzenmacher. 2001. The Power of Two Choices in Randomized Load Balancing. *IEEE Trans. Parallel Distrib. Syst.* 12, 10 (Oct. 2001), 1094–1104. doi:10.1109/71.963420

[33] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. Association for Computing Machinery, New York, NY, USA, 69–84. doi:10.1145/2517349.2522716

[34] David A. Patterson. 2008. Technical perspective: the data center is the computer. *Commun. ACM* 51, 1 (2008), 105.

[35] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* (2010), 65–79.

[36] Raja R. Sambasivan, Alice X. Zheng, Michael De Rosa, Elie Krevat, Spencer Whitman, Michael Stroucken, William Wang, Lianghong Xu,

and Gregory R. Ganger. 2011. Diagnosing performance changes by comparing request flows. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation* (Boston, MA) *(NSDI'11)*. USENIX Association, USA, 43–56.

[37] Benjamin H. Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, et al. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical Report. Google, Inc.

[38] Lalith Suresh, Marco Canini, Stefan Schmid, and Anja Feldmann. 2015. C3: cutting tail latency in cloud data stores via adaptive replica selection. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation* (Oakland, CA) *(NSDI'15)*. USENIX Association, USA, 513–527.

[39] Abhishek Verma, Luis Pedrosa, Madhukar R. Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-scale cluster management at Google with Borg. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. Bordeaux, France.

[40] Ashish Vulimiri, Oliver Michel, P. Brighten Godfrey, and Scott Shenker. 2012. More is less: reducing latency via redundancy. In *Proceedings of the 11th ACM Workshop on Hot Topics in Networks* (Redmond, Washington) *(HotNets-XI)*. Association for Computing Machinery, New York, NY, USA, 13–18. doi:10.1145/2390231.2390234

[41] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. 2008. Improving MapReduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, USA, 29–42.

[42] Zhuo Zhang, Chao Li, Yangyu Tao, Renyu Yang, Hong Tang, and Jie Xu. 2014. Fuxi: a fault-tolerant resource management and job scheduling system at internet scale. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1393–1404. doi:10.14778/2733004.2733012