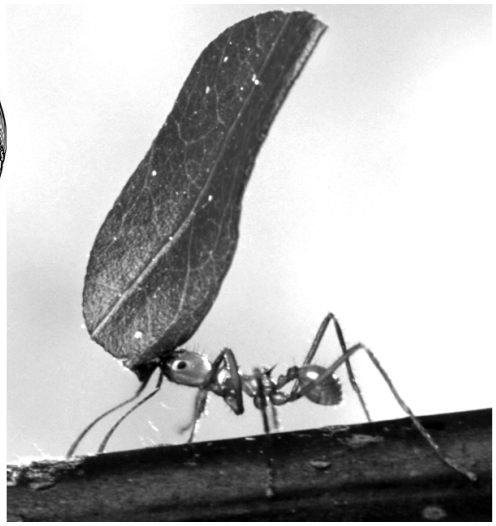



How to Think About Instrumentation Overhead

Splunk Observability
Jason Plumb

2023-12-04

For many novice observability (o11y) practitioners, instrumentation might seem magical. With just a few configuration steps, they can glean deep visibility into application operation and performance. In most cases, the services continue humming along happily while trace waterfalls and dashboards spring to life from the new data.

Even when some manual setup and configuration is required, the value of instrumentation, including auto-instrumentation, easily outweighs the costs of installation and operation. However, the addition of instrumentation into some runtime scenarios could result in unexpected side effects. Even though it is not common, some instrumentation users experience notable increases in response time, the dreaded OOM-kill, bandwidth spikes, or other similarly problematic outcomes.

Why might performance change with instrumentation? When does it happen? And what can we do about it? Can we realistically predict the impact of our observability tools?

Defining Overhead

In the physical world, the formal definition of work is a force applied over a distance, or W = F * s. That is, force is required to achieve any measurable work, and the creation of force requires

the use of energy. Similarly, in the computing world, every functional operation performed by a computer requires computing resources. We usually think of this in terms of utilization of the central processing unit (CPU), but it is often other resources as well. When we introduce instrumentation to an existing, non-observable (opaque) service, we are always adding computational work. This computational work consumes some measure of available computing resources.

Put simply: In order to generate telemetry, the computer has to do additional work, and this work comes with an associated computational cost. That cost is what we call “overhead”.

A Multitude of Perspectives

Overhead is the term used to describe additional resource usage caused by instrumentation. In the observability field, *instrumentation overhead* is simply referred to as *overhead*. For the purposes of this article, we do not consider external out-of-process resources, such as an o11y vendor or other smart pipeline, to be part of overhead, nor do we consider monetary costs when addressing overhead.

We primarily consider the following aspects of overhead:

1. Response time - How much does the instrumentation add to externally measured latency?
2. CPU - How much does CPU usage increase after instrumentation is added? This can impact horizontal scaling.
3. Memory - Does instrumentation cause a notable increase in garbage collection? Can OOMkills be attributed to instrumentation? This also impacts horizontal scaling.

Secondary concerns typically include:

1. Startup time - Usually interesting in FaaS (function-as-a-service) deployments or lots of short-lived workloads.
2. Bandwidth - If you pay for bandwidth or have limited bandwidth availability, you might be concerned about network utilization caused by instrumentation.
3. Disk/storage - Is telemetry buffered on disk before egress? Does the instrumentation itself take up some extra disk space?

One thing is certain – when it comes to overhead, one size does not fit all.

Operators of web services sometimes think about overhead as merely the increase in response time, but this simplification does not tell the whole story! Not all workloads are primarily concerned with latency, and not every web service works the same way. Overhead is *perceived* differently by different users based on their available resources and business concerns.

For example, a service running on an embedded device might have very limited memory where every byte matters. Instrumentation that uses too much of this scarce memory could cause problems for the embedded system. On the other hand, a high speed financial trading system is greatly concerned with latency, where any CPU cycle borrowed by instrumentation could be impactful. Similarly, a maritime deployment might be network limited, so instrumentation's increase in bandwidth consumption might be a challenge.

What Contributes to Overhead?

Overhead is influenced by dozens, if not hundreds of intersectional factors. Stated another way: Overhead is workload dependent.

We break these factors into two main categories when thinking about overhead:

- The quantity of telemetry data created per unit time
- The resources consumed by instrumentation itself

Telemetry Volume

We first consider the volume of telemetry data created in a given time interval. Every piece of telemetry data comes with an associated resource cost. Broadly, when more data is created, this indicates that more work has been performed by instrumentation and implies a higher overhead (see below). Larger data volumes also cause increased memory usage and exporter bandwidth.

Every byte of telemetry uses memory. Every byte processed uses CPU. Every byte exported uses bandwidth. More data often means greater insight, but comes with an increased overhead cost. A method that creates 20 OTLP log records is likely to use more memory and have higher processing overhead than a method that produces just 1. Similarly, a single log record with a 1MB body probably has higher overhead than a short log with just a few bytes, but not necessarily! The attributes of a log record must also be considered. Furthermore, log records often contain external/user input – and as a result, log instrumentation overhead can be influenced by external factors.

Trace size (the number of spans contributing to an overall trace) and trace depth (how many levels deep is the parent/child call ancestry) are both factors of telemetry volume. Some instrumentation will give many highly granular child spans, which results in higher volumes and higher overhead than simple/shallow traces.

Metric instruments are observed at a given rate. A higher rate will yield a higher fidelity signal, but at the cost of telemetry volume. For the same instrument, a lower observation rate will result in lower overhead and decreased fidelity.

This demonstrates just a few of the many complex factors that can influence the size of telemetry, and how the telemetry size influences instrumentation overhead.

Instrumentation Work

We have already established that instrumentation comes with a cost, but not every instrumentation has the *same* cost. The amount of work performed by a given instrumentation is typically governed by the complexity of the thing being observed. Complex pieces of software generally require more work from their associated instrumentation.

To demonstrate this, consider a metric instrumentation that simply adjusts a counter with each observation. There is relatively little work involved in this action – typically just updating or adding two numbers. Compare this with something like JDBC database tracing instrumentation from Java, where each resulting span contains a parsed and cleansed version of the SQL statement. The work performed when parsing the SQL statement is considerably larger than the work performed by the counter. Alternatively, one could compare a simple fixed-bucket histogram of 10 buckets with a complex histogram with thousands of buckets. The latter will surely use more memory and, dependent on the workload, likely more CPU to populate and export.

Tracing instrumentation must also keep track of span scope across thread boundaries so that context may be propagated. This mechanism is required to properly nest child spans. The work required to keep track of span scope is not computationally free, and applications with high volumes of thread context switching, such as those using async frameworks, might have slightly higher instrumentation overhead than simple single-threaded applications.

Processors and Exporters

Although not always considered part of instrumentation itself, the OpenTelemetry APIs allow for configurable telemetry processors and exporters. As expected, not all processors or all exporters are created alike. Some have higher overhead than others, depending on their task, and there is not a single guideline that can indicate which is best for your environment.

For standardization and performance reasons, OpenTelemetry generally recommends use of the OTLP (OpenTelmetry Protocol) exporters for most deployments.

Latency Percentages Are Lies

Sometimes users start leveraging instrumentation for the first time and are shocked by the results. “After adding this instrumentation, my HTTP response latency went up by 75%!

Therefore, I conclude that this instrumentation is awful, terribly inefficient, and I could never justify deploying this into production!”.

This doesn’t tell the full story, and the conclusion is drawn prematurely without understanding the real situation. You have to learn how to think about overhead.

Less is (Relatively) More

First, you might have mistakenly chosen to instrument an overly simplistic test application such as “hello world” or a similar CRUD application. If the test application is too simple or does too little work itself, it won’t reflect the real performance overhead in a production application.

Secondly, as the work performed by an application goes down, the ratio to the typically fixed cost of creating telemetry goes up. Smaller application workloads will have a higher overhead percentage increase than larger application workloads.

Let’s name externally measured latency *without* instrumentation as M_n and externally measured latency *with* instrumentation as M_w. We can think of external response latency *with* instrumentation (M_w) as a combination of both built-in application latency (L_a) and the latency caused by instrumentation overhead (L_i). From this, we can make the following equation:

$$M_w = L_a + L_i$$

And without any instrumentation overhead, latency *without* instrumentation is just

$$M_n = L_a$$

The percentage change in latency (P) between M_n and M_w is computed with the following equation:

$$P = 100 * (M_w - M_n) / M_n$$

And substituting the definitions of M_n and M_w:

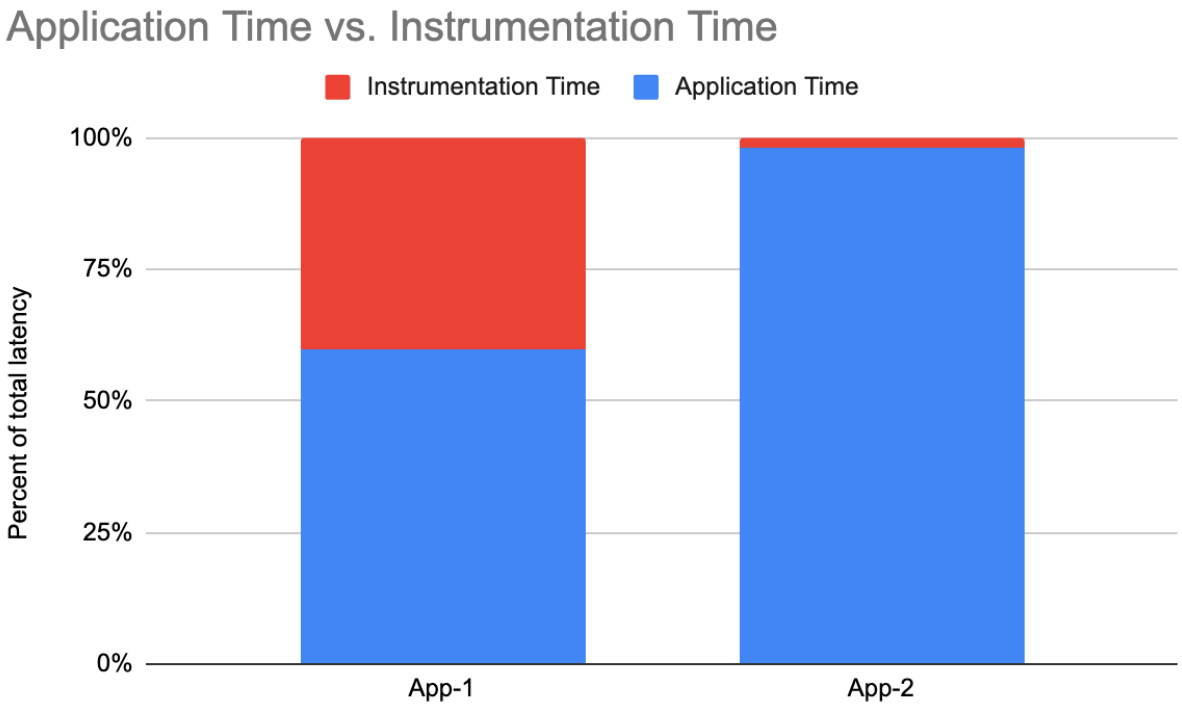
$$P = 100 * ((L_a + L_i) - L_a) / L_a$$

or simply

$$P = 100 * L_i / L_a$$

For a given value of L_i, we can now more clearly see that smaller values of L_a will cause the percentage increase to go up. In other words, for a fixed instrumentation overhead, a lower application latency will cause the *percentage change to increase.*

Let's consider a contrived example using two different applications. App-1 calculates the prime factors of the number 12, and App-2 computes the prime factors of 90085147514322, a much larger number. Without instrumentation, we measure App-1's average response latency at 3ms, and we measure App-2's average response latency at 100ms. We decide we want to instrument these applications, so we apply a fictional "factorization" instrumentation which creates a single trace with a single span for a factorization operation. When we perform our measurements again with instrumentation in place, we observe App-1's latency increased to 5ms and App-2's latency increased to 102ms.



For App-1, this amounts to a 66% increase in response latency, but for App-2 this only amounts to a 2% increase in response latency. It seems that the "factorization" instrumentation causes response times to increase a meager 2ms in both cases, which suggests a mostly fixed cost per span for this instrumentation. However, when interpreted as a percentage change in latency, App-2 *appears* to have taken a worse hit, even though that's not really representative.

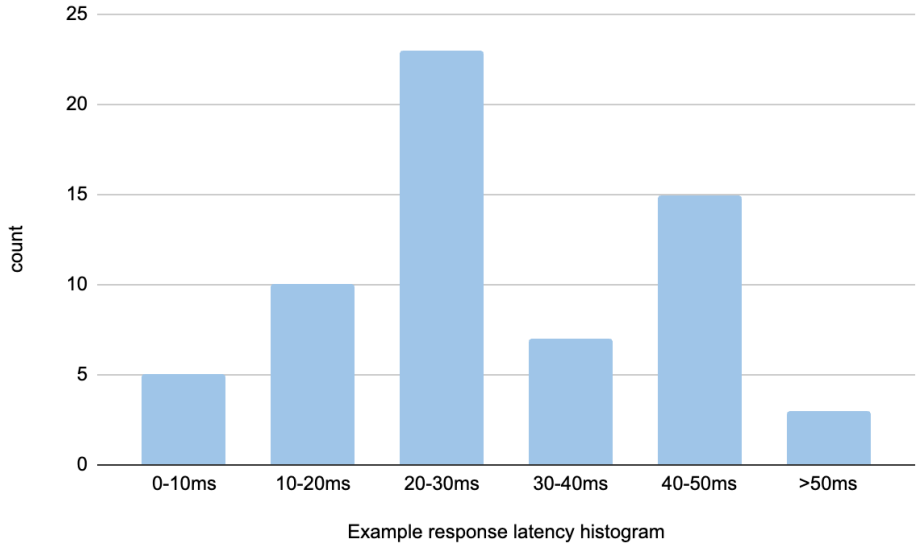
Don't make the mistake of simplifying instrumentation cost to percent-changes in response latency!

There Isn't One Latency Number

Distilling latency down to a single average (mean) is also misleading. Because modern computers are extraordinarily complicated and are influenced by environmental factors, every

execution of an application or service method will take slightly different time. This is true even when the inputs are fixed. Some users will experience 20ms response times and some will experience 400ms response times.

Most real-world production services are processing external and/or user input, and this input will often vary widely. The content of user input to a service contributes to its latency and may also influence instrumentation overhead. One way that o11y practitioners mitigate the oversimplification of mean averages is by leveraging the power of percentiles and histograms. These help to paint a more accurate picture of latency (and other overhead considerations) by quantizing the continuum of values into several ranged buckets instead of a single averaged value. These buckets show how latencies are distributed across all measured values.



Example response latency histogram

Considering the previous prime-factors app example, we could imagine another app (App-3) that, instead of being constrained to factoring a single value, factors a number passed into it by the user. For the single-span “factorization” instrumentation example this is fine: There is a fixed overhead cost for all inputs. However, factorization is often recursive, meaning that it can call itself to solve a problem. If another instrumentation (let’s call it “tree-factorization”) creates a span for every node (recursion step) in the factorization tree, the computational overhead of instrumentation is no longer fixed. Instead, the cost is directly related to the user-submitted input, which determines the number of prime factors in the result. As a result, both application latency L_a and instrumentation latency L_i are dependent on user input.

With the new “tree-factorization” instrumentation, the factorization tree for input=12 would generate only 2 spans, but the tree for input=90085147514322 would generate 6. As a result, users submitting the larger input would experience 3 times the instrumentation overhead of other users submitting input=12. And, naturally, there are pathological inputs that require considerably more steps and incur more overhead.

The takeaway here is this: Real-world latency and overhead are both influenced by largely unpredictable external factors, therefore a single averaged number is misleading. You can use histograms and percentiles to better understand overhead in this dynamic context.

TIY - Test It Yourself

Because every combination of application, deployment environment, and usage pattern is uniquely different, instrumentation overhead is extraordinarily challenging to predict accurately. The best means of confronting this is by performing tests and conducting measurements of your own applications in your own environment. The very task of measuring certain performance characteristics with and without instrumentation may also prove challenging. After all, one of the reasons to add instrumentation in the first place is to collect telemetry to quantify performance details.

Several common aspects of performance can be measured externally. These include:

- Request latency - measured by the calling client or at a proxy sitting in front of the service
- Memory - measured by the host on which the service is running
- CPU - measured by the host on which the service is running

Some language runtimes also have their own instrumentation and/or profiling tools that can be leveraged. These tools should be used consistently with and without the instrumentation being assessed.

When testing, best practices suggest that you should tightly control the environment and limit variability between test runs. In general, you should:

1. Perform several tests without instrumentation. These tests should attempt to closely mimic the production environment. In other words, make the tests realistic but consistent.
    a. Tests should use a warmup phase to reduce the stochastic impact of factors like class loading, connection setup costs, and just-in-time (JIT) compilation. Requests done during the warming phase should not be included in measurements.
2. Record the performance test results, and set these aside.
3. Thoroughly reset the environment. This step is hugely important! Without a complete reset, your subsequent tests may be impacted by hidden environmental changes such as cache warming, database pre-population, or even filesystem caches. The goal is to make a fair comparison between test configurations.
4. Add instrumentation to your application, taking care to note the specific configuration. This includes the version being used and which features are enabled or disabled.
5. Perform the same tests again as #1, and compare the results with the results from the prior tests. As before, be sure to include a fair warmup phase.

When interpreting the results, consideration should also be given to how the test environment may differ from a production environment. A pure reset (like in #3 above) could lead to pathological edge cases in some scenarios. If that happens, the test may need to be run for a longer time and with more realistic inputs. You should also remember that user inputs often vary by time of day, and that noisy neighbors can impact performance in shared cloud environments. If possible, consider testing in production with a carefully controlled canary instance.

It is also critical that your application has enough memory and CPU headroom to complete a test reliably. When testing to determine overhead, you do not want your application to be memory constrained or to run out of CPU cycles. Applications will behave inconsistently when resource starved, especially in dynamic runtimes like Java, python, or .NET. This is not the time to push the application to its maximum capacity.

After the results from the with/without test passes have been gathered and analyzed, the process should be repeated several times to ensure consistency. If the results vary too much between test runs, additional care should be given to limiting environmental variability.

Room For Tuning

OpenTelemetry developers are quite sensitive to performance concerns, and extra effort is frequently given to reduce the impact of instrumentation. In spite of this, there is almost always room for some configuration tweaking for overhead tuning.

Before starting on an instrumentation tuning exercise, however, users should always ask themselves:

> *Is it fine the way it is? Is it already just good enough?*

Typically, the value of out-of-the-box instrumentation vastly outweighs the additional overhead costs. The iterative process of tuning instrumentation to minimize overhead without diminishing the utility of telemetry is both time consuming and challenging.

Most users will not need to perform any configuration tuning.

Disable Unnecessary Signals

Users who really do need to squeeze out some additional performance should first take a look at the kinds of the telemetry being generated by their instrumentation. This is typically traces, metrics, and logs. The volume and quality of each telemetry signal should be scrutinized to decide if it is truly needed. If your telemetry consumers aren't using the logs output, for example, it should be disabled. Similarly, if you only want to see distributed traces and are not concerned with metrics, you can choose to turn those off as well. Allowing yourself to turn off entire telemetry signal types in your application can result in a massive reduction in overhead.

Upgrade Your Instrumentation

Because OpenTelemetry is a highly active project, it makes sense to keep your instrumentation libraries up to date. As mentioned above, OTel contributors are keen on making frequent performance improvements, so a newer version of a library might readily decrease overhead with little end-user effort.

Leverage Sampling

If the volume of your tracing data is vast, you may be able to leverage sampling to reduce the overhead caused by instrumentation. Sampling is a mechanism for instrumentation to decide when some data can be discarded, and this reduces the amount of telemetry generated. A probabilistic sampler is a reasonable first choice to experiment with, and the probability can be fine-tuned over time. OpenTelemetry provides more information about sampling and available samplers here: https://opentelemetry.io/docs/concepts/sampling/.

Turn Off Unnecessary Instrumentation

Sometimes, the number of libraries automatically instrumented by an agent is greater than needed. It is possible that many redundant or unhelpful spans are created at each level of the application software stack by different instrumentation libraries. Similarly, some metric instruments could perform expensive, high-frequency measurements that the observability practitioner doesn't use. Most agents have a means of disabling unwanted instrumentation libraries.

You can determine which instrumentations are creating unwanted telemetry by performing an analysis of the `otel.scope.name` (formerly `otel.library.name`) attributes on the OpenTelemetry InstrumentationScope. Disabling instrumentation libraries that are currently creating unwanted telemetry will further decrease overhead.

Database instrumentation is a frequently expensive contributor to overhead. This instrumentation often contains large strings that need to be normalized and scrubbed, both of which are costly. Many naive applications still suffer from the "N+1" database problem, wherein a result set is fetched and then a subsequent followup query is issued for each row in the result. This problem is not always obvious until instrumentation is applied and the impact is amplified. Disabling database instrumentation is often a quick way to reduce overhead, albeit at the expense of losing details about database queries.

Be Careful With Manual Instrumentation

In an effort to gain visibility into otherwise opaque portions of application code, developers will sometimes leverage manual instrumentation. In extreme circumstances, developers may sprinkle manual instrumentation excessively around the code “just in case”. The runtime impact of the added instrumentation can be troublesome if care is not taken.

To reduce overhead, developers should ensure that unnecessary manual instrumentation is removed. Manual instrumentation that is in the hot path should be scrutinized and applied carefully.

Prune Attributes

OpenTelemetry spans and logs are able to carry collections of key/value pairs called Attributes. These attributes are a fundamental and important component of these signals, and they help to provide additional context and detail about the observed data. If some data has a large number of attributes or has very large attribute payloads, this will increase overhead. A careful audit of the attributes and their content can highlight which ones are candidates for removal. Unfortunately, attribute filtering with instrumentation SDKs is currently nontrivial.

Resource attributes are another characteristic of OpenTelmetry data that helps to identify the source of telemetry. Some resource attribute data may be verbose or otherwise unnecessary. If the instrumentation library provides a means of altering or filtering unwanted resource attributes, this should be leveraged to reduce overhead.

Conclusion

Observability tools offer enchanting, almost magical insight into the behavioral and performance characteristics of complex software systems. At the heart of this is instrumentation. Instrumentation is additional software that runs within or alongside existing application code in order to make it more observable. The operational costs of instrumentation are nonzero, but its benefits nearly always outweigh these costs.

Observability practitioners are often obsessed with performance optimization and squeezing every last cycle out of their CPUs. As such, an addition of instrumentation can initiate excessive scrutiny and inflated concerns about overhead. Marginal increases in latency or CPU usage have been misconstrued as detrimental by users who haven’t yet learned how to think about overhead.

Instrumentation overhead is a necessary cost, paid in order to gain deeper visibility into the runtime operation of your systems.

Software is increasingly complex, and there are numerous external and internal factors that contribute to the quantity, form, and handling of telemetry and the resulting overhead. As such,

predicting overhead caused by instrumentation is extraordinarily difficult. Tests can help to quantify the overhead, but the results must be interpreted against the application itself and weighed against the measured outcomes.

When overhead reduction is deemed necessary, a process of measure, tune/reconfigure, evaluate, and repeat can be iteratively applied.

As observability continues its rapid development and expansion, the factors contributing to overhead will also grow and morph over time. Hopefully his paper has opened some new perspectives and enhanced the way you think about instrumentation overhead.

Credits

- Thanks to Fabrizio Ferri-Benedetti for editorial help!
- Atlas image source: https://preview.redd.it/i-am-wondering-why-is-atlas-commonly-known-for-holding-the-v0-vtzrlzot2h3b1.jpg?width=358&format=pjpg&auto=webp&s=3127b8a51c0861acf59f96ea608f3b5a8f7117d9
- Ant image source: https://upload.wikimedia.org/wikipedia/commons/0/0f/Worker_ant_carrying_leaf.jpg

---