What developers need to know about observability

by Chris Engelbert
Instana Developer Advocate

Contents

01 Introduction

As developers, we can remember the time when Nagios was state-of-the-art technology. Ever since, the term “monitoring” has had some bad implications associated with it. Looking at dashboards loaded with different graphs and numbers didn’t help us solve issues, and yet we were still asked to look at them. Sure enough, those dashboards told us that something was wrong, but our job wasn’t running the system, was it? We ended up having metrics fatigue after only a short period of time.

Why didn’t Nagios work better for us developers? Apart from performance metrics gathered directly from counters built into the applications code flow, it was all just black box monitoring, mostly consisting of pings to hosts and connections to services—commonly accompanied by some log regex parsing.

To make matters worse, in the old days of service operations, the development and operations worlds were strictly separated. We had database administrators, keeping everyone else miles away from running even a select statement on their own.

The operations team—operating the system in production, handling system failures and relaying information about problems to engineering—made a career out of keeping developers away from production systems. In the best case, they let only a very limited number of developers access them.

Finally, there was the engineering department. It was our job to build features, get them “tested,” and that was it. The next time we were involved was bug fixing. For us, source code was an art in and of itself. Sure enough, we kept any non-developer away from our stuff, too.

Operating and monitoring the system simply wasn’t part of our job. Except for one specific part, adding monitoring bits and pieces into our beautiful source code—something akin to putting aluminum siding on the Sistine Chapel. We hated it for all the valid reasons. It spread like wildfire.

The responsibilities were split and, for a long time, software engineers wrote code and went home. Operations took over from there and had to deal with anything that went wrong.

Times have changed, though. Today,systems look different, deployments work differently and the borders between teams are blurred, if not completely removed. That said, as developers, we’re closer to operations than ever and an indirect part of operating the systems.

With the complexity of modern systems running microservices, “meaningful monitoring” becomes an important part of our lives. To prevent the metric overload of the past, we need to look into the benefits of observability.

This ebook is an examination of the new world. We’ll leave all the bad feelings about monitoring behind and take our first steps into the world of observability and its ever-growing importance for developers.

02 Observability terminology

First, let's get the basics of observability terminology out of the way. To dig deeper into the world of observability, it's important to understand the differences between monitoring and observability along with how data is represented to the user.

Monitoring
Traditionally, monitoring has focused on time series metrics. The process was always the same: collect a bunch of metrics, put those metrics on charts on dashboards, figure out which metrics to set alerts for, and choose some thresholds for alerting. This approach, while better than nothing, was far from ideal because it resulted in either too many or too few alerts, a false sense of security and significant time spent in the problem troubleshooting process.

Even if we expand monitoring to include the health and performance of services, traditional monitoring takes a symptom-based alerting approach. The problems are the symptoms it reacts to. If an external service becomes unreachable, that's a symptom. Instead of responding to symptoms, there should be a focus on collecting relevant data that has context embedded throughout.

Observability
Even though observability is nothing new, it's the perfect complement to monitoring. It may be seen as a superset.

The term observability implies that we observe something. The important part here is to observe at a more granular level than monitoring. While still including all the numbers and graphs from monitoring, observability adds the knowledge of what's meaningful to be monitored to all the different, previously separate teams.

Nobody needs hundreds of graphs and tables, especially if they never help solve issues. On the other hand, it's possible that the necessary data required to help solve a puzzling issue wasn't even collected.

On top of that issue, observability adds distributed tracing, basically a microservices stack trace. We'll discuss what that means. And don't forget: meaningful log analytics reach far beyond the simple error search based on regular expression (regex).

The last two elements are designed to bring monitoring from the inside of applications and services. In contrast to plain monitoring, we gather information directly from the inside of the service and bring it together with everything else we know about the system.

Observability is designed with the knowledge of known failure domains of our system, for example, a service connecting to another service via HTTP may fail to connect for various reasons—that's a known failure domain. Failure domains may depend on the way people think about the service or system as a whole. For that reason, it's important to take different perspectives into account when designing the failure domains and context associated with it, for example, for easy debugging, depending on the audience.

When looking at observability, we see three pillars to bring insight and understanding into our issue: health and performance metrics, distributed traces and logs.

Enterprise observability

In the enterprise, observability must be highly scalable to handle the complexities and scalability of transient systems running short-lived microservices or serverless applications. Systems designed for enterprise observability automatically keep up with the constantly changing infrastructure or service landscape. Part of this process is to discover new or shutdown instances or services without manual intervention. They normally provide wrapper libraries for nonintrusive, minimal changes to source code, sometimes even going as far as fully automatic code instrumentation to add measurements and health probes into an already running application.

Code instrumentation provides a way to add the necessary starting points and endpoints of operations into the running code base without manually adding these into the source code. That said, developers are mostly freed from the tedious job of adding monitoring and tracing elements into their code base, leaving more time to actually work on the business's use cases.

Furthermore, always-on but low-overhead code profiling of development, staging and production systems brings greater insight into the performance of services under live conditions—something developers were missing for a long time. Systems always behaved differently from our expectations in production environments.

After collecting all that data though, it's important to not just display it the old-fashioned way with numbers, tables and diagrams. The bigger picture is that the context of the data is what's important for solving problems, especially to developers.

To make the massive amounts of data useful, enterprise observability solutions must provide automatic correlation, which eases understanding and creates actionable information. They also often help with root cause analysis by providing the necessary context around the correlated events and information.

03 Distributed tracing

Distributed traces are to distributed systems as stack traces are to applications and exceptions or panics. It's a technique to capture and time service handlers and internal calls while a request makes its way through a systems landscape to generate its response. For that reason, it's also sometimes referred to as distributed request tracing.

Distributed tracing helps developers analyze request flows and pinpoint the root cause of issues or performance bottlenecks.

Imagine a user calls a user service asking for the user's own account details. The service itself calls a few services down the stack to retrieve different kinds of information. Figure 1 shows the basic flow of the call. Although information, such as who calls whom, is visible, important data such as timing information is missing from the diagram.

This scenario shows where distributed tracing comes into play. Figures 2 and 3 show the call flow in two different ways, as a timing bar and as a stack trace-like tree view.

Those two views deliver the necessary information for developers to find slow calls, long-running operations, failing services and where those failures happen.

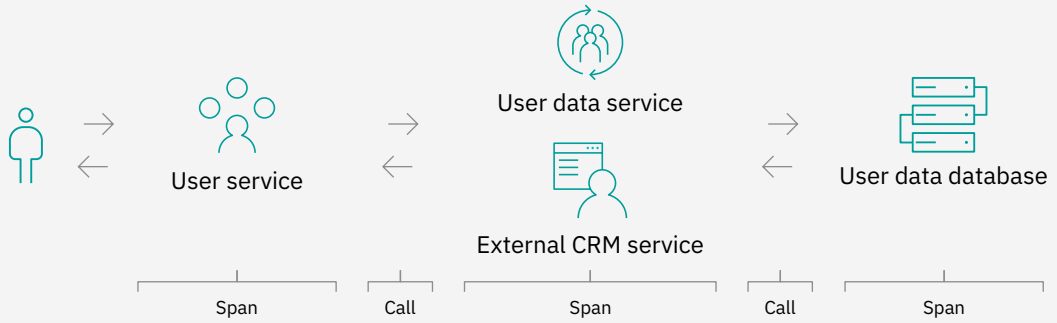


Figure 1. A request to a user service moving along the internal services

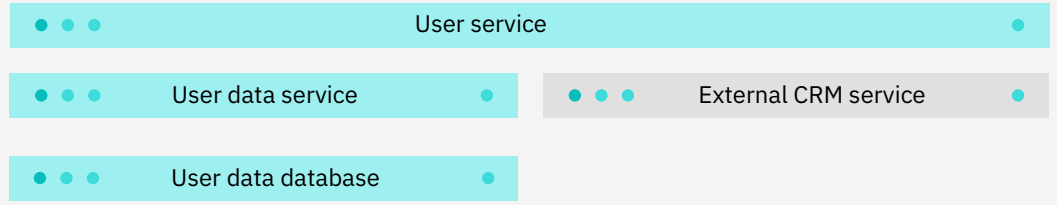


Figure 2. The call flow as a timing and hierarchy diagram

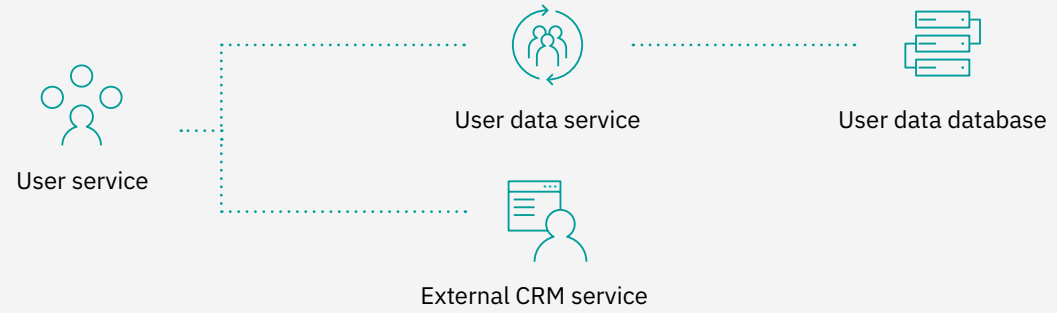


Figure 3. Hierarchy diagram of the code flow

To represent that information, the trace is broken down in elements, such as spans and calls.

A span, sometimes called a timespan, represents the runtime of a single operation inside the flow of the distributed trace. An operation in this context is a code execution with a start time and end time as well as possible parent and child spans.

Additionally, a span contains metadata to provide context around the actual span itself. To connect multiple spans, identifiers (span-id and trace-id) are used to build the parent/child hierarchies.

Communication between two spans is represented by calls. Calls contain information about what type of connection was used, header information and response data, such as HTTP status codes if a call was erroneous. Custom context information can be added, too.

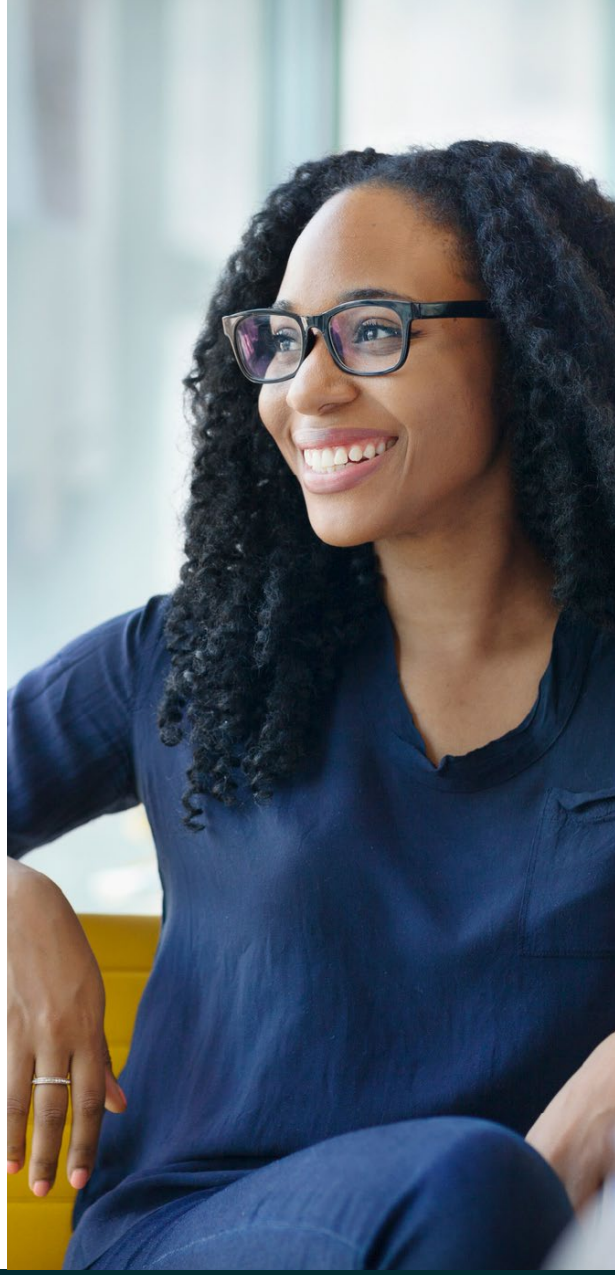
The importance of observability

When working with distributed systems, challenges are vastly different from the monolithic applications we built in the past. Though we love to think of our world in simple terms, the reality is very different—sometimes without us knowing it.

Many different systems and services are running independently and concurrently. The majority of services still have one or more sub-calls to other services or databases. An understanding of the interplay between those services is one of the most important elements to a developer when trying to fix a bug or mitigate a performance problem.

It's also important to not have too many metrics charts on dashboards because we go back to the state we were in years ago. What we are looking for is relevant information that leads us to the cause of a problem as quickly as possible. We need the system to tell us what metrics are important to the failure domain of each specific service. Focusing our attention on only what matters is important here.

And that focus makes sense because there's no guarantee that we'll end up with a failure-free system—no matter how many metrics we add. It's more the other way around; failures in our system can't be avoided. I wrote about this issue in a blog post titled Building Resilient Applications—Embrace the Failure, which goes into detail about that topic. The best we can do is prepare for this case by understanding the aforementioned failure domain.



In the end, the important element for us as developers is to get back to our creativity and motivation. We want to work on the pieces that bring the company forward, our business use cases. We don't want to clutter our source code with millions of metrics or tracing points. And most important, we don't want the constant firefighting or bug fixing.

On the last point though, a clean, helpful and insight-providing observability system can help with all those tasks. By providing slim-wrapper libraries or, even better, automatic instrumentation, we can keep our source base clean, true to the business use case. It also gives us a lot of time back—time we needed to analyze complex issues in distributed systems, time we used to add metrics points, time we spent on understanding interaction and communication between systems, and, last but not least, time we wasted on finding bottlenecks. Today, a distributed trace can help us understand shortcomings and evolutions of the current system before we hit bottlenecks or scalability issues.

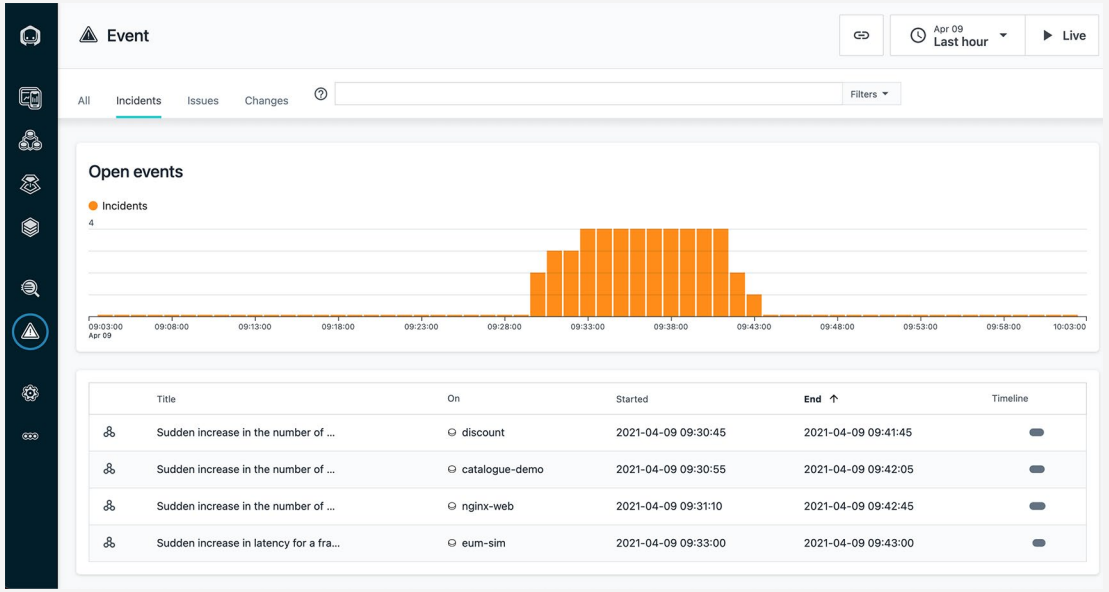


Figure 4. Quick insight into the health of a service with release markers. Source: Instana.

Time to resolution
When a deployment fails or our new version behaves erratically, we have to be fast. Either roll back to an older version or analyze the issue, find a fix, implement it and move on.

Sometimes, the issue is just a little oversight, locally reproducible and quickly fixable. It becomes more complex, though, when we can't directly reproduce it locally. In this case, observability tools can be of great help. The reality is that there's no way to predict how complex the issues are that will arise in our production environments. Because of that lack of predictability, we need to collect the required data from production as it's running.

By providing a simple way to look at the request flow using the distributed trace, it's easy to quickly gather information about where the request failed. We can also see if the error bubbled up the stack or was handled somewhere in between, and if a user was impacted.

Distributed traces also provide information about callers and callees along with their respective headers and timing or retry information.

This information helps as a quick first step into the root cause analysis and to potentially get colleagues of the responsible area to help on the issue. All of it can be accomplished without randomly pointing fingers but with conclusions based on data.

This top-down analysis approach isn't just much faster but also provides the chance to quickly find the actual root cause of the problem. We get an overview of all aspects of the issue and identify those we need to include in the investigation based on the actual upstream or downstream dependencies.

In short, when searching for the issue, observability delivers the insight we need to act quickly and, in the best case, with minimal dependency on other teams or external partners such as hosting companies. The latter can be especially slow to respond when investigating a time-critical issue.

Deployment gone bad
Historically, as a developer, I could skip this section. Deployments weren't my responsibility.

Today, though, developers write Dockerfile configurations to build Docker images and often also provide deployment configurations in the form of Kubernetes (K8s) YAML descriptors, describing the containers, services and more—or at least parts of them—for good reasons.

It's not a sole DevOps responsibility to provide those descriptors, since a full deployment is most commonly a combination of K8s deployment descriptors of engineering and DevOps.

Anyway, our last deployment failed, and there's no obvious issue with the deployment process itself, but the service dies right after the deployment. Now it's our time to shine.

The first step is to figure out. Was it really us? Looking at a distributed trace can often quickly answer that question. Is it our service that results in an error, for example, HTTP status 500, or is the problem coming from further down the stack? Is the downstream service dependency returning the failure? Maybe we're sending the wrong data. Again, a quick look may answer that question, too.

Observability solutions provide the necessary evidence, originally reported by the systems and based on our failure domain analysis, as precise and contextual observations. Distributed traces are the prime witness when searching for a way to quickly gain an understanding of yet unknown situations. Mean time to repair and mean time to restore service are the key metrics here.

Remember the "Don't deploy on Friday" rule? Me too, and I bet everyone else does as well. These days I'd claim—with good observability and the chance to immediately see small changes in behavior, latency or error rate—we can and should actually deploy on Fridays, although maybe early in the morning. In this case, we have quite a few hours left to get an issue fixed or to roll back if something happens.

If we can't fix an issue right away because we went down the rabbit hole too far, our only choice is to roll back to an older build. With an automated deployment flow, this process is easy enough. Kick off our continuous integration and continuous delivery (CI/CD) pipeline with a different tag or release version and off we go. We just bought ourselves a lot of time to investigate deeper, fix with more care and remove the additional stress to be as fast as possible.

04
High-performing engineering

After much talk about speeding up engineering and spending time where it provides the most value, we need to talk about engineering itself and how it changes or has already changed in the years prior.

Page 18 of the Accelerate State of DevOps 2019 report brought up 4 points necessary for a high-performing engineering team in the days to come. The bullets point to the fact that engineering, operations and database administrators aren't in those wholly separated spaces anymore.

But what does this statement really mean? Are we going to be operations? Certainly not. I'd never think of myself as an operations person even though I'm doing quite a bit of that work on a daily basis. The difference, however, still exists. By no means am I an expert in deployment engineering, although others are. And that statement is true for every other subject, even in engineering. Not everyone is a performance engineer. It's still necessary to have those highly specialized people, but getting the big picture is important for everyone—more important than ever before.

The 4 questions we want to answer are:
– What's our deployment frequency?
– What's our lead time for changes?
– What's our time to restore service?
– What's our change failure rate?

The answers to these questions tell us how fast we can iterate. Our answers provide a good guideline, but be honest. The same goes for our code base; we can only optimize if we understand the problem.

How often do we deploy?
Do we deploy once a year, once a month, once a day or multiple times a day? Deploying more often means we need to have better, faster insight into how new versions behave.

How long does it take for code to go live?
How much time do we actually need for a deployment? Is it a major operation, possibly a release train between all teams? Maybe we can deploy independently. Or can we deploy automatically after all tests are green.

How quickly can we recover from an outage?
If something goes wrong, who can analyze the issue? My favorite question is, "Who can solve the issue on my service if I'm on vacation, without a phone or internet connection, and how long will it take the person to understand the problem in the first place?"
How many deployments fail?

Last but not least, the most common question, the one most people can probably answer without too much thinking: are deployments failing and, if yes, how many and why?

This list is an excellent starting point for finding and understanding the speed of iterations in our company, team, and service or application.

I'd go one step further, though, and say one major question is missing—a question about the differences between releases. It's great to have a low number of failing deployments, and we may already deploy multiple times a day. However, there's still one unasked question: *What's the difference in errors and performance between releases?*

The best deployments and fastest iteration aren't worth a single thought if every release performs worse than the one before or increases the error rate. In the end, we don't want to go back to firefighting, do we?

Development is changing
The way we develop applications is obviously changing. The last few years have brought about the necessity to scale out systems greater than ever before. Setting aside Internet of Things (IoT) solutions, the number of users, customers or people we want to see happily engaging with what we've built is ever increasing.

Many companies are still in the adoption phase, but it's almost certain that new systems are not built in the monolithic way anymore. Scalability has become too important, and scaling vertically is too expensive and, in some cases, unachievable.

Microservices, while not the silver bullet many people proclaim, are definitely here to stay. Slicing larger chunks of work into bite-size pieces and deploying them independently sounds great and is amazing for scalability. The feeling of being able to scale parts of the system independently according to needs is incredible. But on the downside, this scalability comes at a cost by adding lots of network operations in between services. Operations can time out, fail or return garbage. We're fighting a whole new kind of enemy.

Furthermore, I remember the time when data was stored in relational databases. No questions asked; there were no alternatives. Although relational databases are here to stay, we have many different systems to choose from these days, from simple key-value or document stores over graph or time-series databases all the way to column stores for extremely fast aggregations.

All those systems are different, and all behave differently. Nobody can be an expert in all of them. However, the beauty as a developer is that we have the chance to choose the best tool for the job while learning new things.

Last but not least, we deploy systems and services differently today than a few years ago. Deploying to dedicated systems has become the minority of new installations. Almost everybody deploys into virtual machines now. Many already deploy in the cloud or self-hosted environments with Kubernetes or Cloud Foundry. Docker and other container runtimes such as CRI-O are a common sight on developer machines.

All those technologies are beautiful, bringing development on my machine closer to the environment they'll eventually run in. I always disliked the sentence "It works on my machine," even though it was true. The reason I disliked it is simple. It meant something was going on—something I couldn't immediately understand or explain.

Development pipelines

If we want to iterate faster, we need to have good support in place. Although unit tests are hopefully nothing new and basic integration tests are commonly employed, everything after them is often still disregarded as either too complex, too expensive or not "developer-y" enough.

However, with short iteration cycles and distributed systems, integration tests have become more important than ever. Continuous integration on a system closer to staging and production is a must-have for fast feedback cycles during development. They're also a good stage to gather first performance and error rate information. Finding potential issues early in the development of a feature can prevent long rebuilding cycles and bring fast validation of expectations—or prove them wrong.

Another piece to the puzzle is that regular load tests are best automated whenever possible. The reasons are the same as before: short feedback loops and early validation of models, expectations and performance-related questions.

The last step in optimizing the high-performing engineering team is continuous delivery, which requires all the previously mentioned elements in place. It also requires the courage to fail. With the knowledge that failure understanding, quick root cause analysis and bug fix deployment aren't only possible but are supported by all team members and tools throughout the production process, courage is much higher, automatically. We feel much safer going forward.

That said, failure situations must be recognized quickly. Context-enriched observability tools have to support the problem analysis process and help find the root cause in the shortest time possible. Also, all services should be as resilient to failing dependencies as possible.

After the fix is produced and committed, the pipeline kicks off and builds, tests and deploys the new version in production. Small, independent services are easier to handle with such a process than large monoliths.

Finally, just to stress it again, all the steps should be fully automated to minimize the impact of a failed deployment or a broken version.

And we're live
Right after a deployment went live, the most important question beyond "Is it working?" was "How does it perform?"

As mentioned before, a fast turnaround cycle is only meaningful when new releases aren't generally worse in performance than old versions. Exceptions to this rule exist, but it should only be an upfront known and expected performance hit, something already calculated into possible infrastructure scaling.

Important key metrics to keep an eye on immediately following deployment are any kind of unexpected changes in average latency, error rate or downstream calls, for example, the number of database calls or similar. Great observability tools offer direct comparisons of before and after the deployment for easy accessibility.

Anyway, our deployment went well, immediate numbers look good, and the system behaves in the expected ranges. Is this the end of the story? Does our deployment live happily ever after?

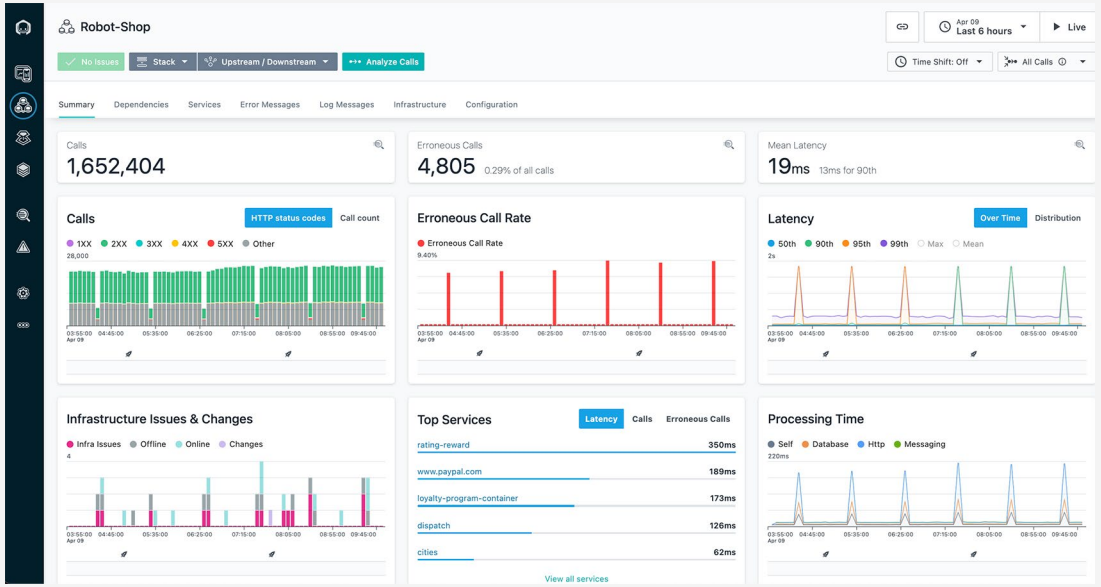


Figure 5. Release markers in the services dashboard for immediate insight after and between release, source: Instana

Unfortunately, we know this outcome is rarely the case for production environments. Something is about to happen all the time. Be prepared. Even though the times of fire-fighting have been shrunk by order of magnitude, they'll never really be gone altogether. Remember, building an unfailing system is simply impossible.

It is true that at this stage of the process, there's less direct involvement of engineering when something goes wrong. To just stress it again, however, when we're on call and being asked to chime in, an intelligent overview of all involved components, networks, machines, applications and their interconnection helps us dig in quickly. It's a task that only becomes more complicated with every bit of abstraction, or architectural complexity, we add to the system. What simplifies and eases our lives during development, increases complexity when partly analyzing unknown systems.

There's only one metric upon which we should judge ourselves, especially for on-call situations: the MTTGBTB—Mean Time to Get Back to Bed. Thanks to Karthik Kumar for this very important metric.

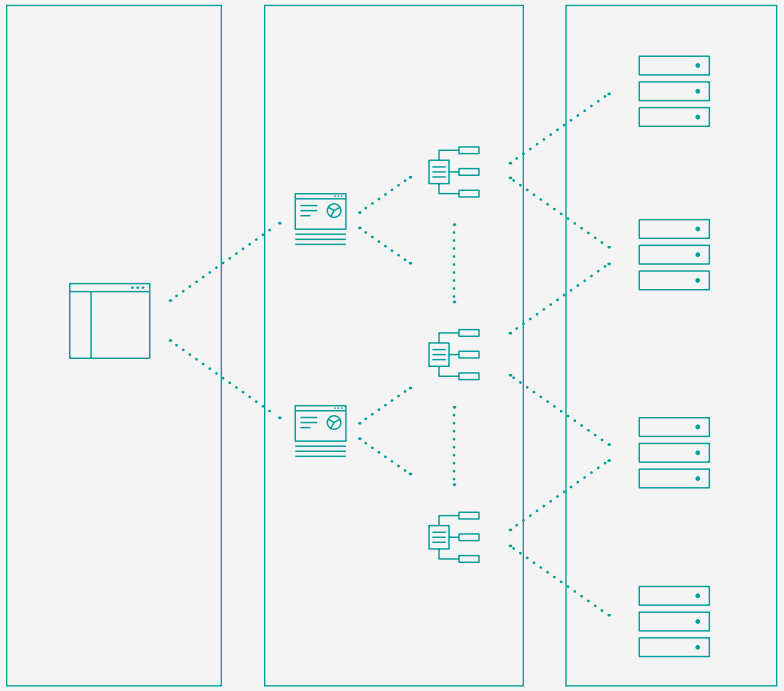


Figure 6. Architectural complexity sometimes prevents quick reasoning

05 Observability-driven development

Now that we understand the importance of observability, let's look at the value of observability-driven development (ODD). Let's quickly summarize the most important elements that are described as ODD.

ODD is a direct extension of behavior-driven development. It extends testing to include the behavior of a component with expectations around performance and health. Tests should be implemented early and run throughout the development cycle. These tests provide insight into performance before, during and after the feature development. They help compare early expectations with reality and provide information if the production system requires scaling before the final feature deployment.

Build the development cycle around the idea of short feedback loops. The real key is to make observability a central part of the development process. Make it a proactive thought and work in conjunction with the other teams. Don't build silos.

During those interactions, come up with health metrics and start to implement them early in the feature's implementation process.

Remember to keep the number of metrics in mind. The optimal selection includes 3–5, at maximum 10 metrics.

If the observability solution requires manual integration using wrappers, add them right from the start, too. If we can have our service instrumented automatically, even better. Our code—our "art"—stays intact.

For optimizations, don't guess. To select optimization points, take into account all data provided by the observability solution, metrics, distributed traces and infrastructure information. Find the biggest contributors to latency and error rates and fix those first. Low-hanging fruit gives quick wins and motivation.

Also, take a look into code profiling data. An always-on production-grade profiler will increase our understanding to a level that was almost impossible to gather beforehand. Until recently, I never had the chance to get a profile on my computer that was anywhere close to what it looked like in production. But please remember, don't just use any profiler. Production-ready profilers are tailor-made for their respective use case, with extremely low overhead.

The last tip will probably hurt just reading it. Test... wait for it... in production. Yes, you read that right. Test in production, but with feature flags. Enable the new behavior, the new feature, based on a set of user IDs, special parameters given—and be creative.

The reasoning behind that step—and the total turn away from the old rule of never testing in production—is the uniqueness of every single environment. The infrastructure state, dependencies, date and time, deployment, environment in itself, and the moon phase... everything may affect our code.

After all the development discussion and why observability makes the developer's life easier, how can we apply it? First steps first, ask the DevOps team if they already have a solution that's used by another team.

06
To be or not to be… open source

As with almost everything, the open-source community did a great job supplying the world with various options. Options are good, aren't they?

Yes and no. One major hurdle with most solutions is that they either do metric monitoring or distributed tracing, but not both. The basic problem is that we lose the correlation between the information. Open-source software (OSS) tools have independent dashboards, independent data silos and, most commonly, don't have a way to jump between metrics, traces and infrastructure for a single request.

Creating observability manually

To create observability, it's common to use two separate tools—as mentioned, one for the metrics and monitoring part, often Prometheus, and one for distributed tracing, often Jaeger or Zipkin.

These open-source solutions are available "for free"—I'll explain the reason for the quotes—and can be integrated with an application service. Integration in this case means that we need to add the tracer and metrics collector to our source code. Wrappers for the most common libraries in many programming languages are available and can be used directly.

A simple example for a Prometheus metric, measuring the number of requests per second, would look similar to the following snippet, using a counter instance to count the number of requests. See Figure 7.

```
public class FooHandler {
   Counter counter = Counter
       .build()
       .namespace("my-app")
       .name("foo-handler")
       .help("number of requests")
       .register();
public ResponseEntity handler(Request
request) {
   counter.inc(1);
   // do some business thing here
   }
}
```

Figure 7. Example for a Prometheus metric

Not too bad for direct usage. Thankfully, many frameworks provide those kinds of metrics using Prometheus out of the box.

The other part is to use Prometheus to measure usage of resources such as database connections or query runtimes. Here, we have plenty of integrations from the Prometheus community. In the Java world, the famous Hibernate object-relational mapping (ORM) solution can be used with just a bit of code. See Figure 8.

```
new HibernateStatisticsCollector()
   .add(sessionFactory, "my-app")
   .register();
```

Figure 8. Example of Hibernate with Prometheus

The Prometheus integration will handle all the dirty details of the implementation for us.

On the distributed tracing side, we have Zipkin or Jaeger. With both solutions, we can add distributed tracing to our application and, again, the integration isn't too complicated, as shown by the following snippet, based on our existing FooHandler.
See Figure 9.

We see that it's still not really complicated, but it somehow feels like we're at the level of "back in the old days." The worst thing is that we cluttered our code with probably more ceremony for monitoring and tracing than actual business logic.

Making sense of the data
Together with log data stored in something like Logstash or Splunk, there's now a lot of data to dig through when a problem happens. There is, however, yet another issue—the observability stack itself.

With 3 independent systems that don't share any data or data correlations, making sense of the different data sources is complicated and time-consuming. Matching timestamps, finding correlations, making the connection and eventually building the context to solve the riddle in question are all up to the user.

```
public class FooHandler {
   Counter counter = Counter
                                .build()
                                .namespace("my-app")
                                .name("foo-handler")
                                .help("number of requests")
                                .register();
private JaegerTracer tracer;

public FooHandler() {
   Configuration.SamplerConfiguration samplerConfig =
      Configuration.SamplerConfiguration
               .fromEnv()
               .withType("const")
               .withParam(1);
   Configuration.ReporterConfiguration reporterConfig =
      Configuration.ReporterConfiguration
               .fromEnv()
               .withLogSpans(true);
   Configuration config = new Configuration()
      .withSampler(samplerConfig)
      .withReporter(reporterConfig);
      this.tracer = config.getTracer();
   }
public ResponseEntity handler(Request request) {
      Span span = tracer.buildSpan("foo handler").start()
      try {
               counter.inc(1);
                        // do some business thing here
      }        catch (Exception e) {
                        span.setTag("http.status_code", 500);
      }        finally {
                        span.finish();
      }
   }
}
```

Figure 9. Snippet of Java tracing with Jaeger

Keep in mind that making sense of it all is the biggest issue with open-source observability tools. We now have more individual systems and more data without the necessary aggregated context itself.

The actual cost of open source
Remember when I mentioned "for free" earlier that we'd talk about it in a bit? Now's the time.

Contrary to common belief, open source isn't free. True, there's no license cost for actually being able to use it. The major cost of OSS observability is measured in time—time where companies pay for people to understand how to operate the observability stack, how to use it and how to make sense of the data. This time, often the thing we have the least of is taken away from delivering business functionality.

It may be necessary to pay additional engineers to adjust OSS observability solutions, too. Let's also not forget the time required to integrate monitoring and distributed tracing.

These tasks are all additional costs on top of the time it takes to implement OSS observability into the source base.

Keeping those integrations in line with our previously set goals can easily add up to 5%–10% of the development time—time we actually tried to avoid when we left the era of Zabbix and Nagios.

When looking at the DevOps and operations teams, this number just increases because now we're operating multiple systems, which is yet another story. We're only looking at our development story.

Last but not least, there's also a cost for the operation of the stack, storage space necessary to keep all the data, and the computation time of trying to reimplement even the very basic automatic correlations.

Suddenly, this "free" open-source tooling just became very expensive and still comes with gaps in observability and context.

The vendor solution
Commercial solutions, on the other hand, provide many, if not all, the above features in a single solution. If certain parts aren't directly provided, they integrate external services to an extent that feels like it's internally stored. Data correlation of internal and external systems is included.

Data correlation between all parts of our systems is the most important element, though. Infrastructure, applications, services, network, logging… all data is preprocessed in a way that the necessary information is easy to gather and make sense of.

Some vendors go one step further and correlate issues found in different parts of the system to create more actionable and targeted alerts. Correlating all issues or events that belong together provides even faster insight into the extent and cause of an incident, the related services and if there's impact to the user.

Furthermore, some vendors provide immediate insight into changes before and after releases, as seen in Figure 5. To achieve that insight, integration with common CI/CD pipelines, DevOps services and development tools is required.

Beyond open source

As a developer, we're looking for the development work, though. Instead of setting up everything manually, some commercial solutions provide fully automatic code instrumentation.

Remember our code example from earlier, and wonder what it would look like when using fully automated code instrumentation?

That's right, no code changes are needed to add monitoring, distributed tracing or performance gathering. Automation picks up the services, no matter if they started directly, in a Docker container or inside Kubernetes, the Red Hat® OpenShift® Platform or similar environments. After automatically discovering it, the service is instrumented on the fly, dashboards are generated based on known best practices for the given service or framework, and we're ready to go. No manual intervention is necessary. What more can we ask for?

The automatic correlation benefit

As mentioned before, the major benefit of an enterprise observability solution is the ability to correlate machine, infrastructure, and application and services metrics and traces. Distributed traces deliver the understanding of the request's flow, while metrics provide the necessary performance points.

Correlating manually, however, is quite cumbersome. The main reason people dislike having multiple dashboards for different services is that it's almost impossible to match any timespans and gather the overall context of the issue.

The biggest benefit of automatic correlation, as described before, is the immediate insight. When looking at an issue or incident, the vendor solution already did all the detective work to provide the important pieces of information as contextual evidence and lead us right to the interesting spot.

```
public class FooHandler {
   public ResponseEntity handler(Request
request) {
      // do some business thing here
   }
}
```

Figure 10. Code using fully automated instrumentation with Instana AutoTrace

07 Simplify our lives

The cost of vendor solutions
As opposed to open-source tools, vendor solutions don't claim to be free.

With open source, we have to operate the system and pay for data storage and computation power. Conversely, vendor solutions offer all the necessary tools to provide full enterprise observability as a hosted software-as-a-service (SaaS) environment or on premises in cases where it's a requirement.

With the removed cost of integration of monitoring and tracing into services and infrastructure, vendor solutions, in the end, are often more cost-effective, with fewer operational and development costs. And let's not forget the storage cost necessary to store all the information and data.

Making an educated decision
Making a choice between open source or a commercial solution is a question of how much time we want to put into development of nonbusiness-related functionality, not a matter of cost, as neither solution is free.

So far, we've been focusing on developers who dislike adding monitoring and tracing into their code, and certainly, that's the biggest group. There are, however, developers who love to tinker. I, for one, consider myself somebody with the tinkering gene. Still, I prefer to play with new technologies, not some performance metric.

Apart from that, my experience tells me that we forget the most important data collections when doing them manually, and not just once. Or we fall back into the old habit of just adding everything possible instead of focusing on our previously defined failure domains.

Vendor solutions provide an ever-growing set of best practices for automatically collected metrics and understand programming languages, frameworks and service integrations. These best practices remove the majority of work when defining the possible failure domains for new or unknown systems. With vendor solutions, it's rare that we'll ever find ourselves missing important information again.

Building software today greatly differs from the traditional methods we've used for the previous decades. Applications are being broken down into small microservices, while deployments are built on top of Docker, Kubernetes and automatic CI/CD pipelines.

Keeping up with all those changes during operations and analyzing issues increasingly becomes more complicated. New ways to gather insight into what's happening is important, especially when on call and trying to understand a problem in an unknown component.

Enterprise observability is the key.
Choose the right fight; don't fight your observability solution.

IBM Corporation
New Orchard Road
Armonk, NY 10504

Produced in the United States of America
April 2021