



# WHAT DEVELOPERS NEED TO KNOW ABOUT OBSERVABILITY

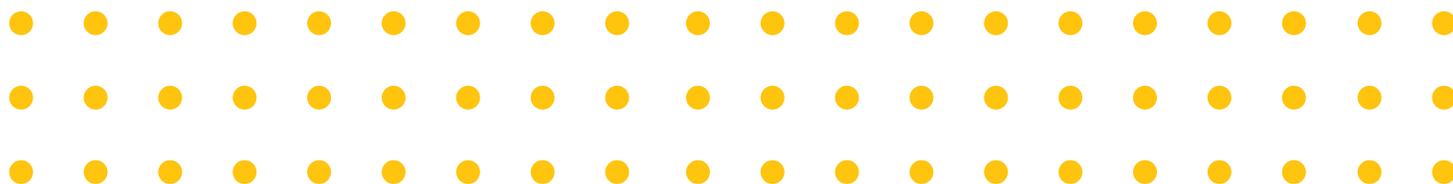
**AUTHOR**

**CHRIS ENGELBERT**

**DEVELOPER ADVOCATE**



# TABLE OF CONTENTS



## **3** INTRODUCTION

## **17** TO BE OPEN SOURCE OR NOT TO BE

- 18 Creating Observability The Manual Way
- 20 Making Sense of The Data
- 20 The Cost of Open Source

## **4** OBSERVABILITY TERMINOLOGY

## **21** THE VENDOR SOLUTION

- 5 Monitoring
- 5 Observability
- 6 Enterprise Observability
- 6 Distributed Tracing

- 22 Beyond Open Source
- 22 The Automatic Correlation Benefit
- 22 The Cost of Vendor Solutions

## **8** THE IMPORTANCE OF OBSERVABILITY

## **23** MAKING AN EDUCATED DECISION

- 10 High-Performing Engineering
- 11 Development is Changing
- 11 Development Pipelines
- 12 Time to Resolution
- 13 Deployment Gone Bad
- 14 And We're Live

## **15** OBSERVABILITY-DRIVEN DEVELOPMENT

## **24** SIMPLIFY OUR LIVES

# 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, they 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 with some log regex parsing.

To make matters worse, in the old days of service operations, the Dev and Ops 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 and 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.

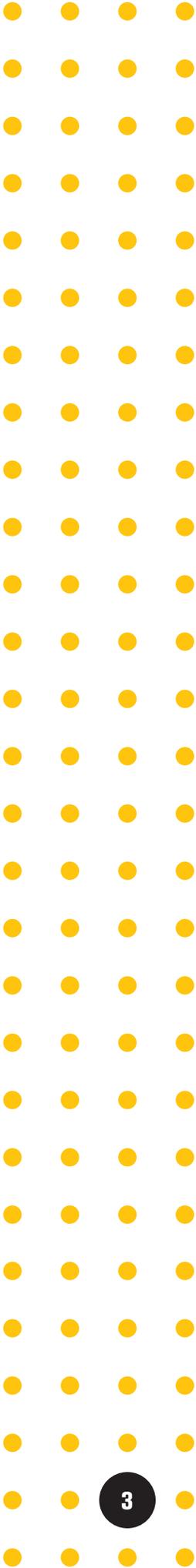
Operating and monitoring the system simply wasn’t part of our job. Except for one specific part - adding monitoring bits and pieces into our beautiful source code (something akin to putting aluminum siding on the Sistine Chapel) . We hated it, for all the valid reasons! It spread like cancer.

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.

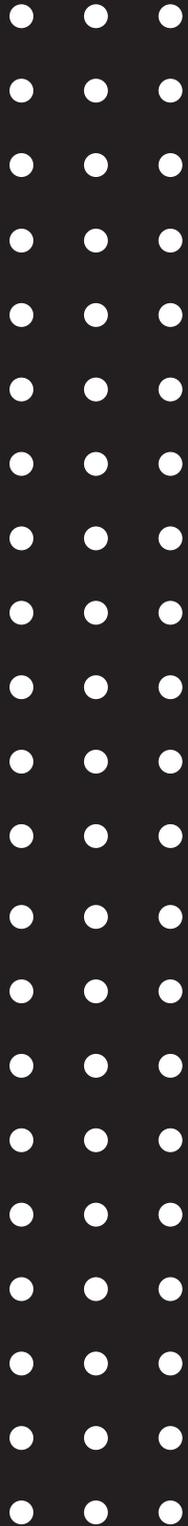
Times have changed though. Today, systems look different, deployments work differently, and the borders between teams are blurred, if not completely removed. That said, as developers, we are closer to operations than ever, and are an indirect part of operating the systems.

With the complexity of modern systems, running microservices “meaningful monitoring” becomes an important part of our life. 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, taking our first steps into the world of Observability and its ever-growing importance for developers.



# OBSERVABILITY TERMINOLOGY



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

## A MONITORING DEFINITION

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 as 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 being reacting to. If an external service becomes unreachable, that is a symptom. Instead of responding to symptoms, there should be a focus on collecting relevant data that has context embedded throughout.

## DEFINING OBSERVABILITY

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

The term Observability implies that we observe something. The important bit here is to observe at a more granular level than monitoring. While still including all the numbers and graphs from monitoring, it adds the knowledge of what is 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 is possible that the necessary data needed to help solve a puzzling issue wasn't even collected.

On top of that, Observability adds Distributed Tracing, basically a microservices stack trace. What that means will be discussed in a few minutes. And not to forget, meaningful log analytics, far beyond the simple regex based error search.

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 of the system.

Observability is designed with the knowledge of known failure domains of our system, e.g. 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 is important to take different perspectives into account when designing the failure domains and context associated with it, e.g. 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 non-intrusive, 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 start- 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' 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 in production environments from our expectations.

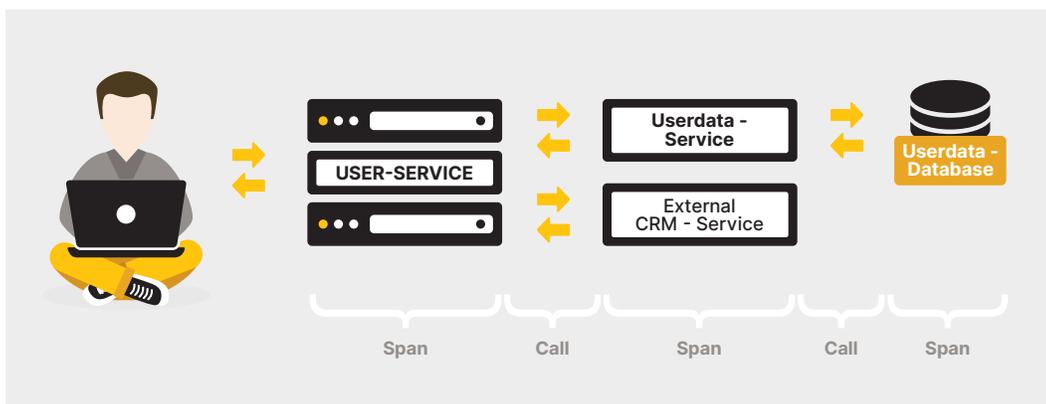
After collecting all that data though, it is important to not just display them the old fashioned way with numbers, tables and diagrams. Especially to developers, the bigger picture, the context of the data is what's important for solving problems.

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.

# DISTRIBUTED TRACING

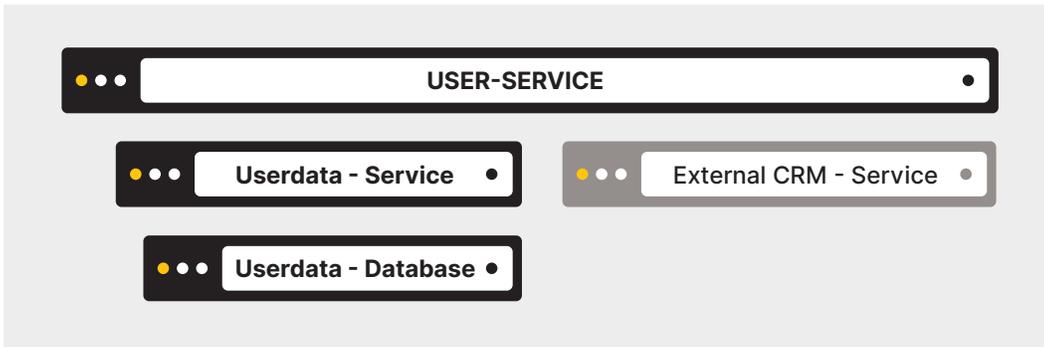
Distributed Traces are to distributed systems as stack traces are to applications and exceptions or panics. It's a technique to capture and time service handlers and internal calls while a request makes its way through a systems landscape to generate its response. For that reason, it is also referred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

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



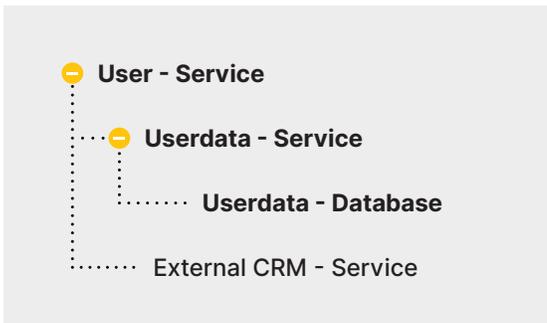
Picture 1: A request to a user service moving along the internal services

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



Picture 2: The call flow as a timing and hierarchy diagram

This is where Distributed Tracing comes into play. Picture 2 and picture 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.

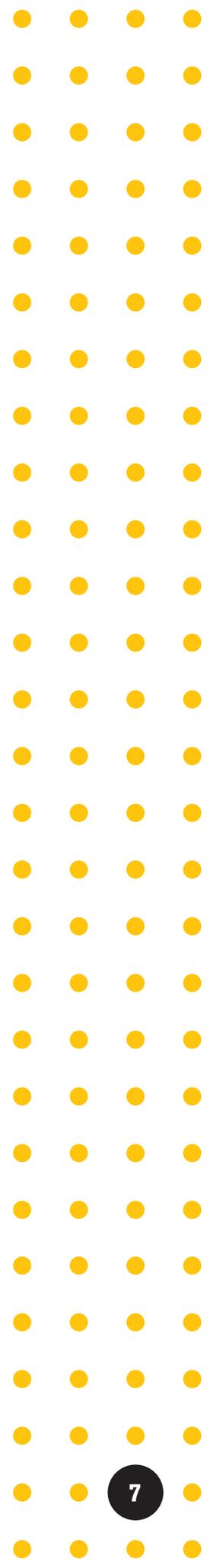
To represent that information, the trace is broken down in elements such as spans and calls.

Picture 3: Hierarchy diagram of the code flow

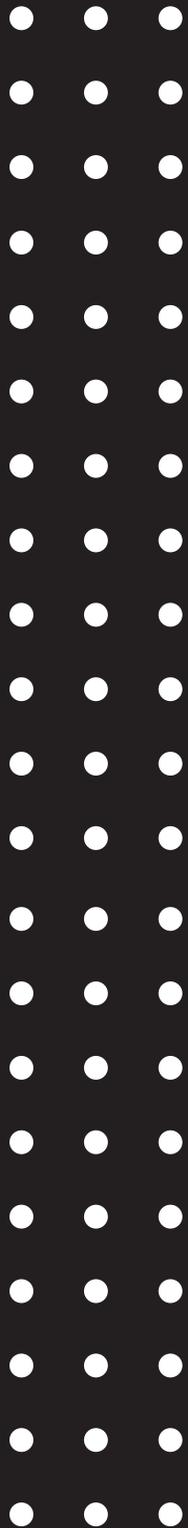
A span, sometimes also called timespan, represents the runtime of a single operation inside the flow of the Distributed Trace. An operation in this context is a code execution with start and end time, as well as possible parent and child spans.

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

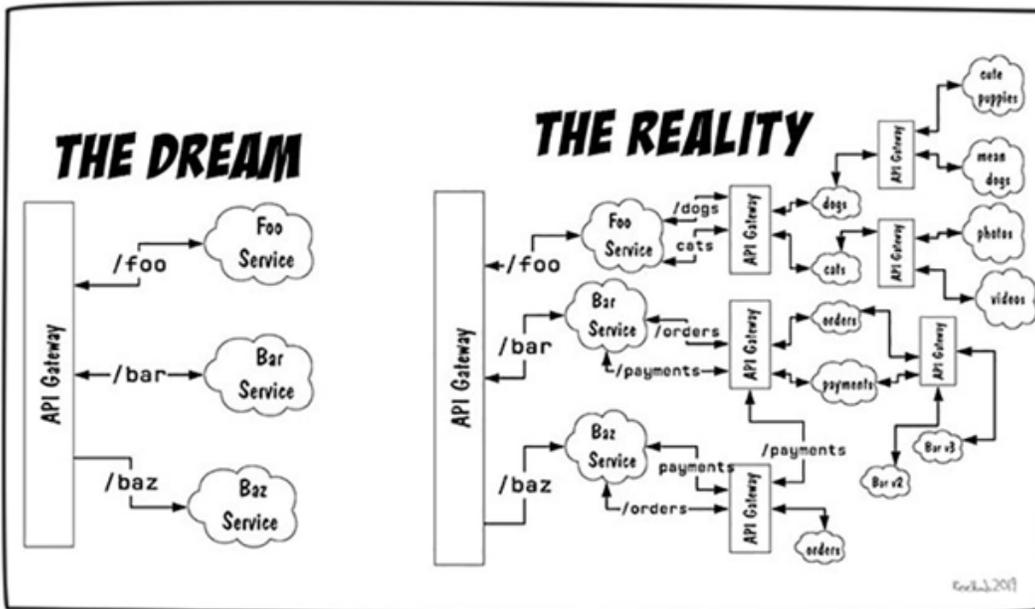
Communication between two spans are represented by calls. Calls contain information about what type of connection was used, header information, and response data, such as HTTP status codes if a call was erroneous. Custom context information can be added too.



# THE IMPORTANCE OF OBSERVABILITY



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



Picture 4: The dream vs the reality of modern architectures © [ROELBOB](#)

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 bits to a developer when trying to fix a bug or mitigate a performance problem.

It is also important to not have too many metrics on charts on dashboards, because otherwise we go back to the state we were 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 that are important to the failure domain of each specific service. Focussing our attention on only what matters is important here.

And that makes sense because there is no guarantee that, no matter how many metrics we add, we'll end up with a failure-free system. It is more the other way around, it can't be avoided to have failures in our system. I wrote about this in a blog post titled [Embrace the Failure](#), which goes into details on that topic. The best we can do is to prepare for this case, by understanding the already mentioned failure domain.

In the end, the important bit 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 fire fighting 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.

# 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 already has changed in the years prior.

The State of DevOps report ([page 18](#)) brought up 4 points necessary for a high-performing engineering team going into the days to come. 4 bullets pointing towards the fact that engineering, operations and database administrators aren't in those holy separated spaces anymore.

But what does that really mean, are we going to be ops? 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. I, by no means, am an expert in deployment engineering. Others are though. And that is true for every other subject, even in engineering. Not everyone is a performance engineer. It is still necessary to have those highly specialized people, but getting the big picture is important for everyone, more important than ever before.

Anyway, the 4 questions we want to answer;

- What is our Deployment Frequency?
- What is our Lead Time for Changes?
- What is our Time to Restore Service?
- What is 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. 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 a 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 probably most people can answer without too much thinking; are deployments failing, if yes how many and why.

This list is an amazing 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 is great to have a low number of failing deployments, and we may already deploy multiple times a day. There is one unasked question still;

## What is the difference in errors / performance between releases?

The best deployments and fastest iteration is not 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 fire fighting, 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 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 is 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 the needs is incredible. But on the downside, this adds lots of network operations in between services. Operations can timeout, can fail, can return garbage. We're fighting a whole new kind of enemy.

Furthermore, I remember the times when data was stored in relational databases. No questions asked, what else would we use? While 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. The beauty as a developer, however, we have the chance to choose the best tool for the job, 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 today. Many already deploy into the Cloud or self-hosted environments with Kubernetes or Cloud Foundry. Docker and other container runtimes like 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 hated the sentence "it works on my machine", even though it was true. The reason I hated it is simple. It meant something was going on, something I couldn't immediately understand or explain.

## DEVELOPMENT PIPELINES

If we want to iterate fast(er), we need to have good support in place. While Unit Tests are (hopefully) nothing new, and basic integration tests are commonly employed, everything after that 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 are 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, regular load tests, are also 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 has to have 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 bugfix deployment are not only possible, but supported by all team members and tools throughout the production process, courage is much higher, automatically. We feel much safer going forward.

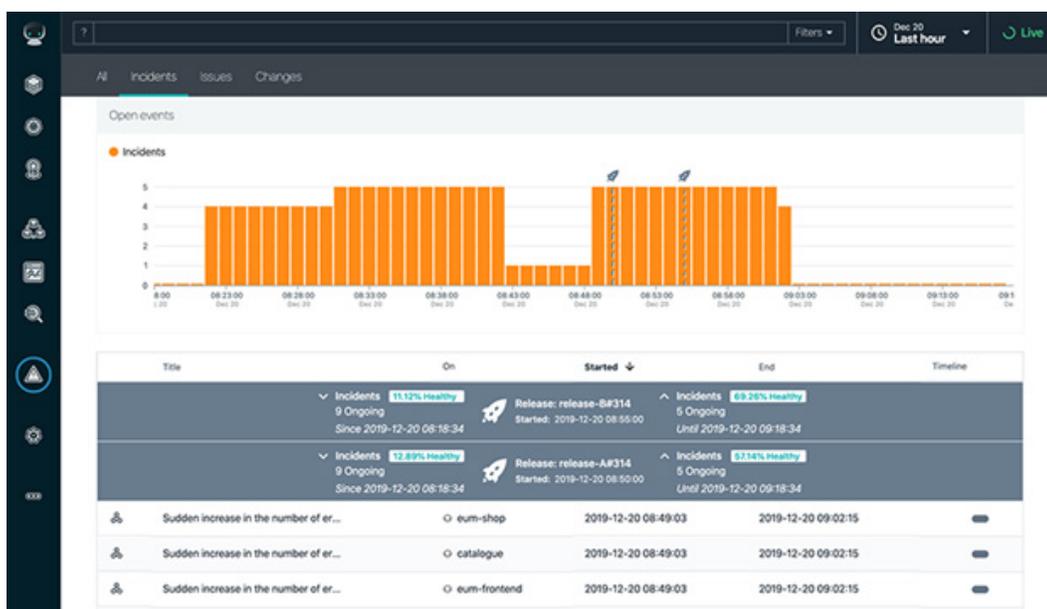
That said, failure situations need to be recognized quickly. Context enriched observability tools have to support the problem analysis process and help find the root cause in the shortest time possible. Also, all services should be as resilient to failing dependencies as possible.

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 of the steps should be fully automated to minimize the impact of a failed deployment or a broken version.

## TIME TO RESOLUTION

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



Picture 5: Quick insight into the health of a service with release markers - source Instana

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

With providing a simple way to look at the request flow using the Distributed Trace, it is easy to quickly gather information about where the request failed and 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 callee, as well as their respective headers and timing or retry information.

This helps as a quick first step into the root-cause analysis and potentially get colleagues of the responsible area to help on the issue. All of that, without randomly pointing fingers, but conclusions based on data.

This top-down analysis approach is not only 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 who we need to include in the investigation, based on the actual up- 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, like hosting companies. Especially the latter can be slow to respond when investigating a time critical issue.

## DEPLOYMENT GONE BAD

As a developer, historically, I could skip that section. Deployments were not my responsibility.

Today, though, developers write Dockerfile configurations to build Docker images and often also provide deployment configurations in form of Kubernetes (k8s) YAML descriptors, describing the containers, services and more (or at least parts of it), for good reasons.

It's not a sole devops responsibility to provide those, since a full deployment is most commonly a combination of k8s deployment descriptors of engineering and devops.

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

First step is to figure out, was it really me? Looking at a distributed trace can often quickly answer that question. Is it my service that results in an error, e.g. HTTP status 500, or is the problem coming from further down the stack. Is the downstream service dependency returning the failure? Maybe I'm sending the wrong data. Again a quick look may answer that question too.

Observability solutions provide the necessary evidence, originally reported by the systems, 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. Time to Repair and Time to Restore Service are the key metrics here.

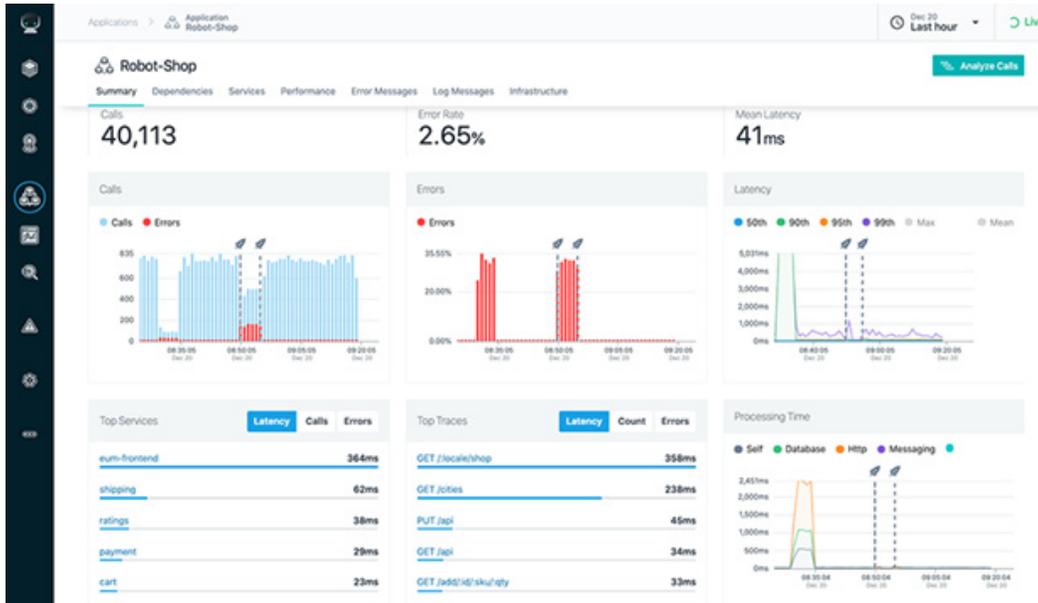
Remember the "don't deploy on Friday" rule? Me too, and I bet everyone else does as well. These days I'd claim, with good observability and the chance to immediately see small changes in behavior, latency or error rate, we can and should actually deploy on Fridays. Although maybe early in the morning ;-). 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, since 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 is easy enough. Kick off our 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".

# 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 from this rule exist, but it should only be an upfront known, and expected performance hit. Something already calculated into possible infrastructure scaling.



Picture 6: Release markers in the services dashboard for immediate insight after and between release - source Instana

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, e.g. 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?

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

It is true that at this stage of the process, there is 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 to dig in quickly. A task that only becomes more complicated

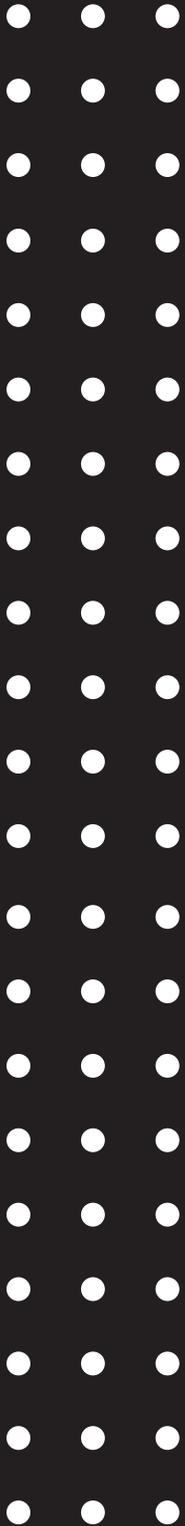
with every bit of abstraction we add to the system (Architectural Complexity). What simplifies and eases our lives during development, increases complexity when analysing (partly) unknown systems.

Especially for on-call situations, there is only one metric to judge ourselves on, the MTTGBTB (**Mean Time to Get Back to Bed**). Thanks to Karthik Kumar for this very important metric!



Picture 7: Architectural Complexity sometimes prevents quick reasoning

# OBSERVABILITY-DRIVEN DEVELOPMENT



Now that we understand the importance of Observability, let's look at the value of Observability-driven development (ODD). Building upon what's already been discussed, let's quickly summarize the most important bits 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 to 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 key 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 those early in the feature's implementation process. Remember to keep the number of metrics in mind. The optimal selection includes 3-5, at max 10 metrics.

If the observability solution requires manual integration using wrappers, add those right from the start, too. If we are able to have our service instrumented automatically, even better. Our code - our "art" - stays intact.

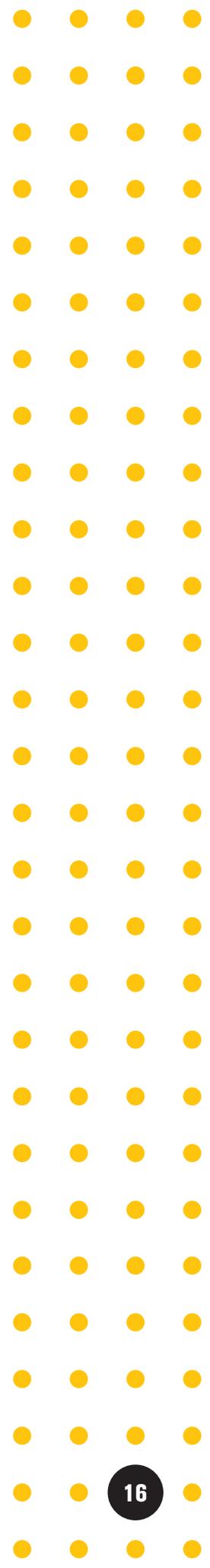
For optimizations; don't guess. To select optimization points, take into account all data provided by the observability solution, metrics, distributed traces, and infrastructure information. Find the biggest contributors to latency and error rates, fix those first. Low hanging fruit gives quick wins 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, do not just use any profiler, production-ready profilers are tailor-made for their respective use case, with extremely low overhead.

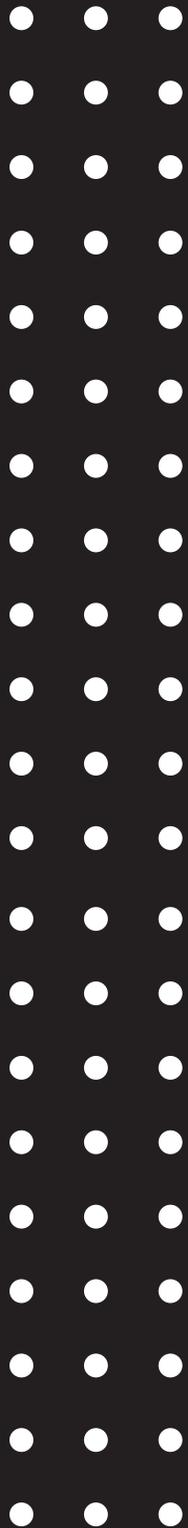
The last tip will probably hurt just reading it. Test ... wait for it ... in production. Yes, you read that right. Test in production, but with Feature Flags. Enable the new behavior, the new feature based on a set of user ids, special parameters given, be creative.

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, the dependencies, the date and time, the deployment, the 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 which is used by another team.



# TO BE OR NOT TO BE OPEN SOURCE



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

Yes and no. One major hurdle with most solutions is, they either do metric monitoring or distributed tracing, but not both. The basic problem is that we lose the correlation between the information. 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 THE MANUAL WAY

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

These open source solutions are available "for free" (the reason for the quotes will be explained in a few minutes), and can just be integrated with an application service. Integration in this case means, 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 directly be used.

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:

```
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
    }
}
```

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, like database connections or query runtimes. Here, we have plenty of integrations from the Prometheus community. In the Java world the famous Hibernate ORM solution can be used with just a bit of code, as:

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

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 is not too complicated, as the following snippet based on our existing FooHandler shows.

```
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();
        }
    }
}
```

We see, it is still not really complicated, but it somehow feels like we're at the level of "back in the old days". The worst thing is, 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 is 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 do not 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 is all up to the user.

That is the biggest issue with open source observability tools to keep in mind. We now have more individual systems, and more data without providing the necessary aggregated context itself.

## THE ACTUAL COST OF OPEN SOURCE

Remember when I mentioned “for free” earlier and said, we’re going to talk about it in a bit? Now’s the time.

Contrary to common belief, open source is not free. True, there is 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. Time, often the thing we have the least of, that 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 are all additional costs on top of the time it takes to implement OSS Observability into the source base.

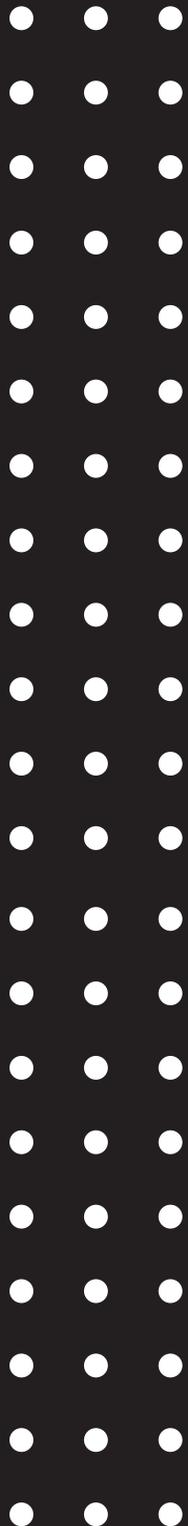
Keeping those integrations in line with our previously set goals can easily add up to 5-10% of the development time. Time we actually tried to avoid, when we left the era of Zabbix and Nagios.

When looking at the DevOps and Operations teams, this number just increases, since now we’re operating multiple systems. That, though, is yet another story. We’re only looking at our development story.

Last but not least, there is also a cost for the operation of the stack, storage space necessary to keep all of 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, of the above features in a single solution. If certain parts are not directly provided, they integrate external services to an extent that it feels like internally stored. Data correlation of internal and external systems is included.

That is the most important bit though, data correlation between all parts of our systems. 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 is impact to the end user.

Furthermore, some vendors provide immediate insight into changes before and after releases (as seen in picture 6). To achieve that, integration with common CI/CD pipelines, DevOps services and development tools is required.

## BEYOND OPEN SOURCE

As a developer we are 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 above, and wonder what it would look like when using fully automated code instrumentation?

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

That's right, no code changes to add monitoring, distributed tracing, or performance gathering is needed. Automation picks up the services, no matter if started directly, in a Docker container, or inside Kubernetes, OpenShift 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 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 / 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 hate 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.

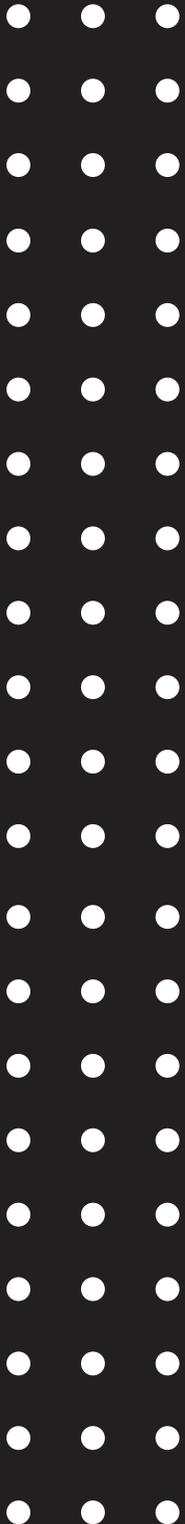
## THE COST OF VENDOR SOLUTIONS

As opposed to open source tools, vendor solutions do not claim to be free.

Whereas with open source we have to operate the system, pay for data storage and computation power, vendor solutions offer all the necessity to provide the full Enterprise Observability as a hosted SaaS (Software as a Service) environment or also as on-premises in cases where that is a requirement.

With the removed cost of integration of monitoring and tracing into services and infrastructure, vendor solutions, in the end, often end up being more cost effective, with less operational, and development costs. Not to forget the storage cost necessary to store all the information and data.

# MAKING AN EDUCATED DECISION



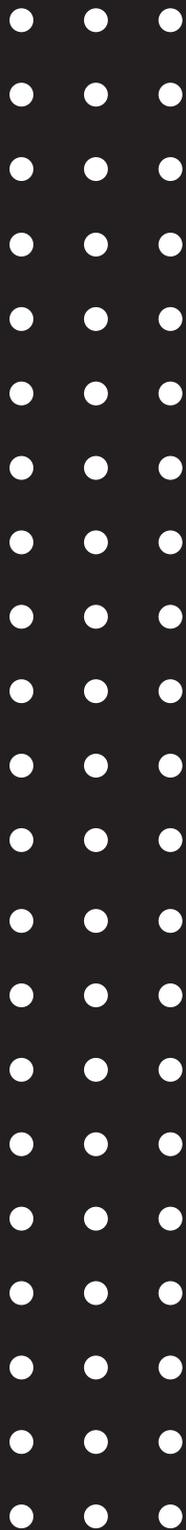
Making a choice between open source, or a commercial solution, is a question of how much time we want to put into development of non-business related functionality, not a matter of cost, as neither solution is free.

So far we've been talking about the developer, who hates 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. Either that, or we fall back into the old habit of just adding everything possible instead of focussing on our previously defined failure domains.

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

# SIMPLIFY OUR LIVES

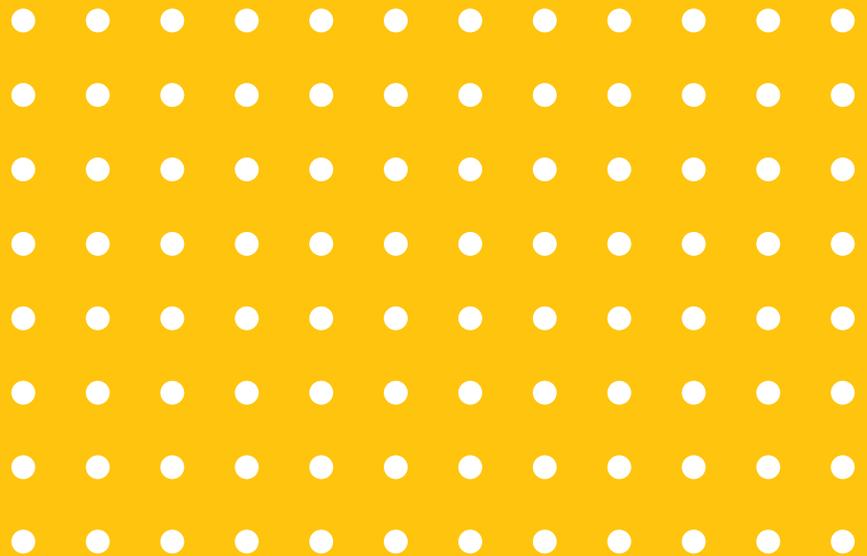


Building software today greatly differs from the traditional methods we've used for the last decades. Applications are being broken down into small microservices while deployments are built on top of Docker, Kubernetes, and automatic CI/CD pipelines.

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 being 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.

# THANK YOU FOR READING



**CHRIS**ENGELBERT

DEVELOPER ADVOCATE

