Bad Observability

Anti-pattern #1 - Forgetting the customer
Anti-pattern #2 - Environment inconsistency
Anti-pattern #3 - Not understanding your ecosystem
Anti-pattern #4 - No consistent trace ID
Anti-pattern #5 - The big dumb metric
Anti-pattern #6 - Bad sampling intervals
Anti-pattern #7 - Misunderstanding metrics
Anti-pattern #8 - Lazy synthetic transactions
Anti-pattern #9 - A plague of dashboards
Anti-pattern #10 - Unnecessary alerts
Anti-pattern #11 - Hoarding data
Anti-pattern #12 - Disconnected data
Anti-pattern #13 - Throwing tools at a problem
Anti-pattern #14 - Mandating tools
Anti-pattern #15 - The chosen few
•

Focus on the customer first. If you don't know whether your customers are able to consume your services, then go back and make that happen first before setting up low level technical monitoring.
•**It's about outcomes**. Your organization is trying to achieve something. Everything we do in technology should be working toward achieving that, and that includes observability. Use observability to help track whether your organization is succeeding in its endeavors.
•**Treat observability as a product** for your organization. Open it up for others to make use of, make sure it plays nicely with the wider organization, be thoughtful about how you build it.

Bad Observability

Anti-pattern #1 - Forgetting the customer

organizations who **monitor a *lot* of technical signals about their systems and services**, but who are not able to answer questions about the end customer. Such as...

- **What was the customer impact of that change we deployed last night?**
- **Has that performance improvement we made had an impact on our conversion rates?**

We spend a lot of time and money building services for customers. Yet rather than validating whether what we have built is having the desired effect on customers, we instead focus on tracking the health of our technology.

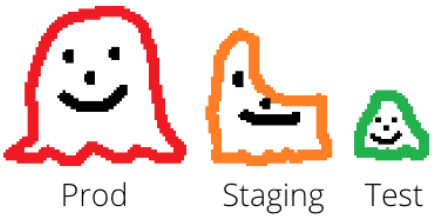
If we were cooks in a restaurant it would be like measuring our success based on how clean the dishes are rather than whether our customers are enjoying the food we made for them (or not).

Tracking customer experience and behavior should be the *first* thing we observe. It is our best measure of reliability. Anything else we look at to track reliability is one or more steps removed. It is entirely possible to have system monitoring showing an issue, yet the customer experience has not degraded at all. It is equally as possible for system monitoring to show everything is green yet the customers are suffering.

I think this is part of why SLOs exist. They put the focus on the customer when delivering features and services. They involve tracking this in production and building a feedback loop back into decision making.

Don't get me wrong...**we still need detailed infrastructure, platform, and application level monitoring to help diagnose issues and help us understand what's happening under the covers**. But if you don't know what **the customer behavior is**, and you don't know whether your services are able to be consumed by your customers or not...then closing that gap is the priority.

Anti-pattern #2 - Environment inconsistency

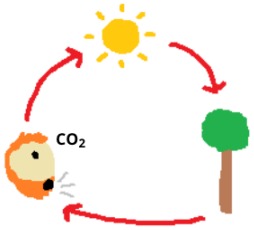


Many organizations have wildly (or subtly) different observability tooling and configuration set up in their production and pre-production environments. This lack of consistency can lead to a number of drawbacks.

Firstly, teams miss out on the opportunity to practice using their observability tooling and ways of working before features reach production. This is a huge missed opportunity to identify issues before they impact real customers. It also encourages the age old gap between delivery and operations.

Secondly, observability tools themselves can impact reliability and performance (and even cause major incidents). If they only exist in production, there is no opportunity to uncover these issues before real customers are impacted.

Anti-pattern #3 - Not understanding your ecosystem



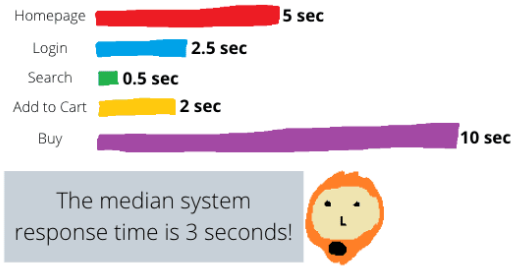
- Not being clear on the **downstream services that you depend on**. To meet your customer objectives you probably need to track the reliability of these services.
- Not understanding the **upstream customers who consume your services,** which gives you context into what is important to observe (or not).
- **Not understanding the most business critical or technically important parts of a system**. This can lead to spending too much time setting up observability for components that are unlikely to fail or impact your customers, or leaving blind spots, which end up causing major incidents later.
- Only tracking server-side metrics when there is significant work being done in the customer's browser or mobile device. This leaves you blind to the full end-to-end reliability, performance, and customer experience. It can also hide issues such as JavaScript execution issues.
- Only observing workloads of a particular type. For example, only monitoring the experience of external customers **but forgetting about your own internal staff**, who depend on your services to do their daily work.

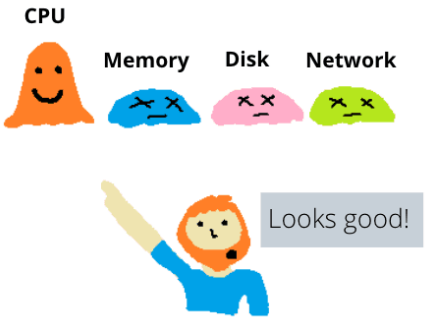
Anti-pattern #4 - No consistent trace ID

When an incident occurs or there is a performance bottleneck in a large complex distributed system, it's incredibly helpful to be able to **track a single customer interaction right through the solution**.

As the solutions we work with continually increase in complexity, this is becoming more important than ever (and more challenging).
This is a fairly straightforward problem to solve. Just make sure the top level component is generating a **unique token (trace or correlation ID) that is passed throughout the solution**. This is usually passed as an HTTP header. Check out the B3 Propagation specification for an example.

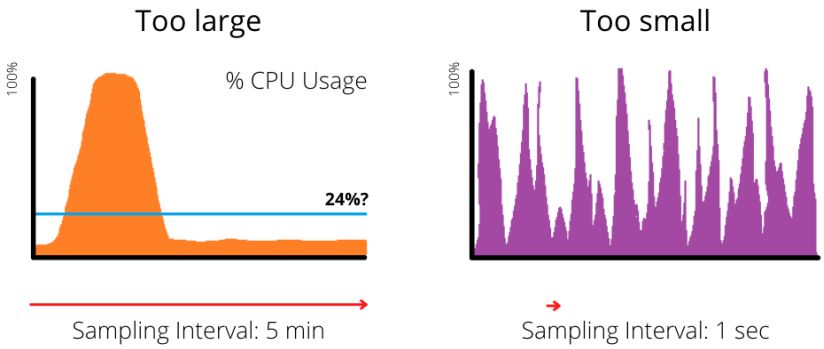
Anti-pattern #5 - The big dumb metric





"big dumb metric" happens when we monitor infrastructure. I frequently see monitoring that only looks at one metric: total % CPU Usage. **CPU is important, but it's not the only hardware resource.** The other three that need to be considered are memory, disk, and network. And even within your CPU monitoring**, you sometimes need to know which process is consuming CPU time**, which CPU cores are active at a point in time, or whether CPU is being consumed by system or user processes. Blindly tracking just the one metric is going to bite you.

Anti-pattern #6 - Bad sampling intervals



Let's use the example of % CPU usage. What if you decided to capture the average CPU usage on a server at five minute intervals? **During that time there might be a one minute period where the CPU burned at 100% and the rest of the time it was 5%. Your monitoring would report an average CPU usage of 24% over that five minute window**, which is true, but it doesn't reflect the fact that there was a period where customers were likely impacted. Seeing a CPU usage of 24% is also misleading because it makes it sound like there was fairly consistent utilization, whereas in reality it came in bursts.

On the other end of the spectrum, what if you were capturing and reporting CPU usage *every second*? Statistically, there are going to be one second periods where the CPU usage hits 100% but, depending on your context, this probably won't impact customers in any meaningful way. With CPU usage, it's about prolonged periods of high usage that lead to queuing. I worked with a team once that was fixated on "maximum CPU usage" as their key metric of utilization, which resulted in them *massively* over-provisioning their infrastructure.

Anti-pattern #7 - Misunderstanding metrics

It's important not to take a metric at face value without being clear what exactly it does and does not tell you.
One example I see all the time is tracking "available memory" on a server and claiming that "there is a memory leak" when said available memory decreases over time. Available memory is the truly "free" memory that is not allocated to any process yet. Just because "free" memory has run out does not mean that there is no available memory for processes on the server.
Processes can have memory allocated to them but that memory is actually still available for other processes or the operating system to use if required. Just because available memory is low does not necessarily mean that there is a memory problem.
Applications are often designed to take as much memory as possible for efficiency; it's normal behavior. Especially database management systems. If you really want to track memory, do it at the platform or application level. For example, tracking heap memory usage in a JVM.
Another example is when we track container CPU usage. In the context of a container, what is % CPU usage? What does it mean? In his 2020 Neotys PAC talk "Seeing is knowing, measuring CPU throttling in containerized environments" Edoardo Varani demonstrated (with proof) that % CPU usage was not a good indicator of how utilized a container is. He was able to produce situations where container CPU usage was 100% but the application performance was not degraded, or % CPU usage was 50% but the application was queuing for processor time. When you are monitoring containers, look at % Throttled Time as a more accurate measure of how utilized a container is.

Anti-pattern #8 - Lazy synthetic transactions

Let's be honest... synthetic transactions are just a fancy term for an automated test that runs on a regular basis in production. It helps you keep track of the health of your services even when there is no customer activity at all, and it checks service health in a consistent way.

Most customer facing web applications contain a flow of steps that culminate in some customer outcome. There is usually some kind of session information being carried throughout the flow. For example, for an online store a customer might:

1.Visit a landing page
2.Search for a product
3.View a product
4.Add the product to their cart
5.Purchase the product

A lazy synthetic transaction would hit the landing page...and then stop. This is really easy to implement, but it hasn't proved whether the customer outcome (to buy a product) can be achieved or not. Unless we can prove that, we're not doing our job.

An effective synthetic transaction would need to prove that all the individual services required in a product purchase are working as intended. In my experience, I've seen this implemented as an automated script that steps through the product purchase process.

This is, of course, much more work. There is data to manage and non-trivial test assets to build and maintain. I understand the reluctance to go to the effort but, on the other hand, a synthetic transaction that doesn't give us confidence about the customer experience isn't serving its purpose.

There are plenty of creative ways to make this more manageable. Maybe re-using automated test assets that have already been built. Maybe creating special test endpoints to run in production to make checking all the services easier.

Anti-pattern #10 - Unnecessary alerts

If you wouldn't wake up at 3am in the morning to handle a situation, then you shouldn't be generating alerts for it.

Every alert or page that goes out that does not need immediate action is training your engineers not to take them seriously. I'm sure you've heard of the boy who cried wolf - well, this is the monitoring platform that cried major incident.

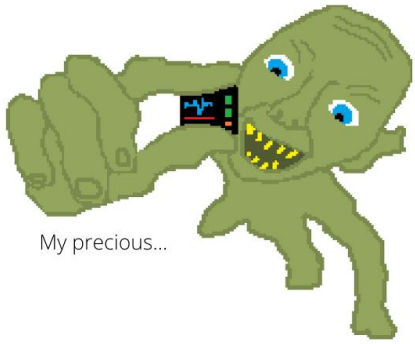
If you are currently getting a lot of pages for incidents that do not require immediate action, then it's time to adjust your alerting rules.

These false alarms will drain patience and sanity. Not to mention, you need enough free time to be proactive about reliability. If you're constantly fighting unimportant fires, that's going to be difficult.

As with many of my other points in this article, bring it back to the customer outcome. If your customers can still effectively use your services (and there's no threat to that in the immediate future) then why are you panicking or waking up in the middle of the night?

Bad Observability

Anti-pattern #11 - Hoarding data



At times I have come across teams who have their own observability platform and won't share this with the rest of the organization.

To be fair, it's something I've rarely seen, but when it does it happen it comes from a pathological culture that includes a fear of failure, a fear of change, and command and control. This kind of culture not only opens up chasms between the teams in your organization, but also strips away the psychological safety and ultimately the performance of your teams.

I believe that observability data should be freely open for anyone in your organization to see and learn from. Logs with customer information, of course, need careful attention, but most other observability data is not a significant security risk.

Bad Observability

Anti-pattern #12 - Disconnected data

Sometimes we have all the observability data we need, but it's spread all over the organization in different tools and repositories. There might be no consistent use of standards or trace IDs either.
Having multiple tools isn't a problem in and of itself. I think it's better to have a few different tools used by teams who are motivated, own their own observability, and have a sense of autonomy than mandating one tool for everyone. There's still a challenge around pulling all that data together, but that's something we can solve creatively.
The real anti-pattern here isn't having too many tools, it's where teams are treating their observability like a private resource, rather than a product for the whole organization. Because that's what it is and how it should be treated.

Bad Observability

Anti-pattern #14 - Mandating tools

When an organization mandates that a particular tool *must* be used, it strips autonomy and ownership away from the teams they intend to use it. This doesn't lead to great outcomes. Speaking frankly, the times I've seen this in the past were when a tool vendor has sold an idea to senior leadership and the decision was made without consulting the engineers who were actually supposed to use it.

Product teams and engineers need to be part of the decision making process. The decision making should be driven by the desired outcomes you want to see. What problems or opportunities is this tool supposed to help tackle? How do we need to adjust our ways of working to get the most out of it?

I would rather see many teams using many tools but with a sense of ownership of their observability, rather than one consistent tool imposed on a bunch of reluctant teams.