# Chapter 1. The Need for Observability

Infrastructure software is in the midst of a paradigm shift. Containers, orchestrators, microservices architectures, service meshes, immutable infrastructure, and functions-as-a-service (also known as "serverless") are incredibly promising ideas that fundamentally change the way software is built and operated. As a result of these advances, the systems being built across the board—at companies large and small—have become more distributed, and in the case of containerization, more ephemeral.

Systems are being built with different reliability targets, requirements, and guarantees. Soon enough, if not already, the network and underlying hardware failures will be robustly abstracted away from software developers. This leaves software development teams with the sole responsibility of ensuring that their applications are good enough to make capital out of the latest and greatest in networking and scheduling abstractions.

In other words, better resilience and failure tolerance from off-the-shelf components means that—assuming said off-the-shelf components have been understood and configured correctly—most failures not addressed by application layers within the callstack will arise from the complex interactions between various applications. Most organizations are at the stage of early adoption of cloud native technologies, with the failure modes of these new paradigms still remaining somewhat nebulous and not widely advertised. To successfully maneuver this brave new world, gaining visibility into the behavior of applications becomes more pressing than ever before for software development teams.

Monitoring of yore might have been the preserve of operations engineers, but observability isn't purely an operational concern. This is a book authored by a software engineer, and the target audience is primarily other software developers, not solely operations engineers or site reliability engineers (SREs). This book introduces the idea of observability, explains how it's different from traditional operations-centric monitoring and alerting, and most importantly, why it's so topical for software developers building distributed systems.

# What Is Observability?

*Observability* might mean different things to different people. For some, it's about logs, metrics, and traces. For others, it's the old wine of monitoring in a new bottle. The overarching goal of various schools of thought on observability, however, remains the same—bringing better visibility into systems.

---

**OBSERVABILITY IS NOT JUST ABOUT LOGS, METRICS, AND TRACES**

Logs, metrics, and traces are useful tools that help with testing, understanding, and debugging systems. However, it's important to note that plainly having logs, metrics, and traces does not result in observable systems.

---

In its most complete sense, observability is a property of a system that has been designed, built, tested, deployed, operated, monitored, maintained, and evolved in acknowledgment of the following facts:

- No complex system is ever fully healthy.
- Distributed systems are pathologically unpredictable.
- It's impossible to predict the myriad states of partial failure various parts of the system might end up in.
- Failure needs to be embraced at every phase, from system design to implementation, testing, deployment, and, finally, operation.
- Ease of debugging is a cornerstone for the maintenance and evolution of robust systems.

**THE MANY FACES OF OBSERVABILITY**

The focus of this report is on logs, metrics, and traces. However, these aren't the only observability signals. Exception trackers like the open source Sentry can be invaluable, since they furnish information about thread-local variables and execution stack traces in addition to grouping and de-duplicating similar errors or exceptions in the UI.

Detailed profiles (such as CPU profiles or mutex contention profiles) of a process are sometimes required for debugging. This report does not cover techniques such as SystemTap or DTrace, which are of great utility for debugging standalone programs on a single machine, since such techniques often fall short while debugging distributed systems as a whole.

Also outside the scope of this report are formal laws of performance modeling such as universal scalability law, Amdahl's law, or concepts from queuing theory such as Little's law. Kernel-level instrumentation techniques, compiler inserted instrumentation points in binaries, and so forth are also outside the scope of this report.

---

# Observability Isn't Purely an Operational Concern

An observable system isn't achieved by plainly having monitoring in place, nor is it achieved by having an SRE team carefully deploy and operate it.

Observability is a feature that needs to be enshrined into a system at the time of system design such that:

- A system can be built in a way that lends itself well to being tested in a realistic manner (which involves a certain degree of testing in production).
- A system can be tested in a manner such that any of the hard, actionable failure modes (the sort that often result in alerts once the system has been deployed) can be surfaced during the time of testing.
- A system can be deployed incrementally and in a manner such that a rollback (or roll forward) can be triggered if a key set of metrics devi-

ate from the baseline.
- And finally, post-release, a system can be able to report enough data points about its health and behavior when serving real traffic, so that the system can be understood, debugged, and evolved.

None of these concerns are orthogonal; they all segue into each other. As such, observability isn't purely an operational concern.

## Conclusion

Observability isn't the same as monitoring, but does that mean monitoring is dead? In the next chapter, we'll discuss why observability does *not* obviate the need for monitoring, as well as some best practices for monitoring.