Chapter 2. Monitoring and Observability

No discussion on observability is complete without contrasting it to monitoring. Observability isn't a substitute for monitoring, nor does it obviate the need for monitoring; they are complementary. The goals of monitoring and observability, as shown in Figure 2-1, are different.

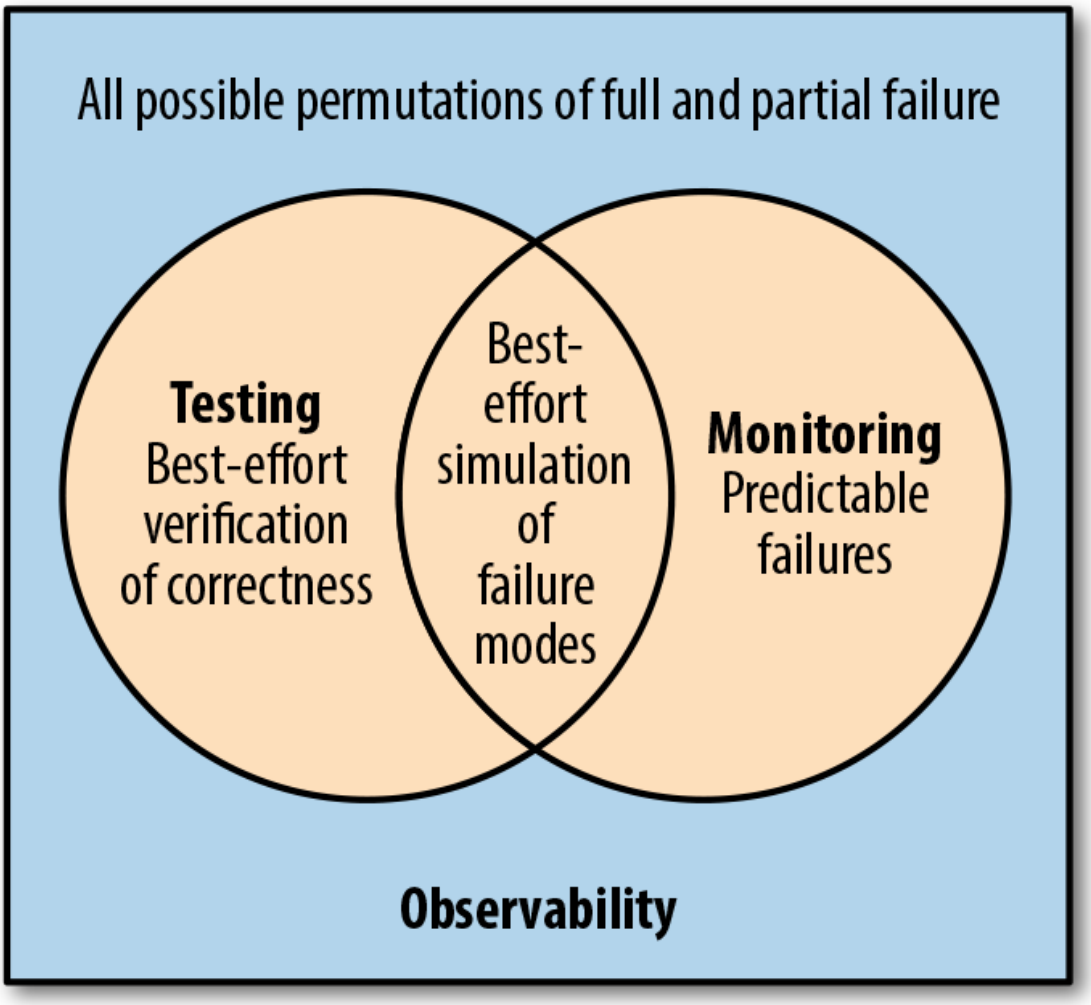

Figure 2-1. Observability is a superset of both monitoring and testing; it provides information about unpredictable failure modes that couldn't be monitored for or tested

Observability is a superset of monitoring. It provides not only high-level overviews of the system's health but also highly granular insights into the implicit failure modes of the system. In addition, an observable system furnishes ample context about its inner workings, unlocking the ability to uncover deeper, systemic issues.

Monitoring, on the other hand, is best suited to report the overall health of systems and to derive alerts.

Alerting Based on Monitoring Data

Alerting is inherently both failure- and human-centric. In the past, it made sense to "monitor" for and alert on symptoms of system failure that:

- Were of the predictable nature
- Would seriously affect users
- Required human intervention to be remedied as soon as possible

Systems becoming more distributed has led to the advent of sophisticated tooling and platforms that abstract away several of the problems that human- and failure-centric monitoring of yore helped uncover. Health-checking, load balancing, and taking failed services out of rotation are features that platforms like Kubernetes provide out of the box, freeing operators from needing to be alerted on such failures.

BLACKBOX AND WHITEBOX MONITORING

Traditionally, much of alerting was derived from blackbox monitoring. *Blackbox monitoring* refers to observing a system from the outside—think Nagios-style checks. This type of monitoring is useful in being able to identify the *symptoms* of a problem (e.g., "error rate is up" or "DNS is not resolving"), but not the triggers across various components of a distributed system that led to the symptoms.

Whitebox monitoring refers to techniques of reporting data from inside a system. For systems internal to an organization, alerts derived from blackbox monitoring techniques are slowly but surely falling out of favor, as the data reported by systems can result in far more meaningful and actionable alerts compared to alerts derived from external pings. However, blackbox monitoring still has its place, as some parts (or even all) of infrastructure are increasingly being outsourced to third-party software that can be monitored only from the outside.

However, there's a paradox: even as infrastructure management has become more automated and requires less human elbow grease, understanding the lifecycle of applications is becoming harder. The failure modes now are of the sort that can be:

- *Tolerated,* owing to relaxed consistency guarantees with mechanisms like eventual consistency or aggressive multitiered caching
- *Alleviated* with graceful degradation mechanisms like applying backpressure, retries, timeouts, circuit breaking, and rate limiting
- *Triggered* deliberately with load shedding in the event of increased load that has the potential to take down a service entirely

Building systems on top of relaxed guarantees means that such systems are, by design, not necessarily going to be operating while 100% healthy at any given time. It becomes unnecessary to try to predict every possible way in which a system might be exercised that could degrade its functionality and alert a human operator. It's now possible to design systems where only a small sliver of the overall failure domain is of the hard, urgently human-actionable sort. Which begs the question: where does that leave alerting?

Best Practices for Alerting

Alerting should still be both hard failure–centric and human-centric. The goal of using monitoring data for alerting hasn't changed, even if the scope of alerting has shrunk.

Monitoring data should at all times provide a bird's-eye view of the overall health of a distributed system by recording and exposing high-level metrics over time across all components of the system (load balancers, caches, queues, databases, and stateless services). Monitoring data accompanying an alert should provide the ability to drill down into components and units of a system as a first port of call in any incident response to diagnose the scope and coarse nature of any fault.

Additionally, in the event of a failure, monitoring data should immediately be able to provide visibility into the impact of the failure as well as the effect of any fix deployed.

Lastly, for the on-call experience to be humane and sustainable, all alerts (and monitoring signals used to derive them) need to *actionable*.

What Monitoring Signals to Use for Alerting?

A good set of metrics used for monitoring purposes are the USE metrics and the RED metrics. In the book *Site Reliability Engineering* (O'Reilly), Rob Ewaschuk proposed the four golden signals (latency, errors, traffic, and saturation) as the minimum viable signals to monitor for alerting purposes.

The *USE methodology* for analyzing system performance was coined by Brendan Gregg. The USE method calls for measuring utilization, saturation, and errors of primarily system resources, such as available free memory (utilization), CPU run queue length (saturation), or device errors (errors).

The *RED method* was proposed by Tom Wilkie, who claims it was "100% based on what I learned as a Google SRE." The RED method calls for monitoring the request rate, error rate, and duration of request (generally represented via a histogram), and is necessary for monitoring request-driven, application-level metrics.

Debugging "Unmonitorable" Failures

The key to understanding the pathologies of distributed systems that exist in a constant state of elasticity and entropy is to be able to debug armed with evidence rather than conjecture or hypothesis. The degree of a system's observability is the degree to which it can be debugged.

Debugging is often an iterative process that involves the following:

- Starting with a high-level metric
- Being able to drill down by introspecting various fine-grained, contextual observations reported by various parts of the system
- Being able to make the right deductions
- Testing whether the theory holds water

Evidence cannot be conjured out of thin air, nor can it be extrapolated from aggregates, averages, percentiles, historic patterns, or any other

forms of data primarily collected for monitoring.

An unobservable can prove to be impossible to debug when it fails in a way that one couldn't proactively monitor.

Observability Isn't a Panacea

Brian Kernighan famously wrote in the book *U*nix for Beginners in 1979:

> *The most effective debugging tool is still careful thought, coupled with judiciously placed print statements.*

When debugging a single process running on a single machine, tools like GDB helped one observe the state of the application given its inputs. When it comes to distributed systems, in the absence of a distributed debugger, observability data from the various components of the system is required to be able to effectively debug such systems.

It's important to state that observability doesn't obviate the need for careful thought. Observability data points can lead a developer to answers. However, the process of knowing what information to expose and how to examine the evidence (observations) at hand—to deduce likely answers behind a system's idiosyncrasies in production—still requires a good understanding of the system and domain, as well as a good sense of intuition.

More importantly, the dire need for higher-level abstractions (such as good visualization tooling) to make sense of the mountain of disparate data points from various sources cannot be overstated.

Conclusion

Observability isn't the same as monitoring. Observability also isn't a purely operational concern. In the next chapter, we'll explore how to incorporate observability into a system at the time of system design, coding, and testing.