honeycomb.io

Developing a Culture of Observability

Observability gives engineers insight into how their systems function and how users experience the resulting services; it is a language you can use to communicate with your system, allowing you to answer questions that you didn't anticipate having to ask.

Even then, asking questions is only half of the battle. Having the tools to explore and investigate issues in your code doesn't necessarily mean you have a culture that supports and drives the greater benefits of observability: happier engineers, happier customers, and a stronger business overall.

The 2019 DORA State of DevOps Report surveyed 31,000 working professionals and found the following:

> *"The best strategies for scaling DevOps in organizations focus on structural solutions that build community. High performers favor strategies that create community structures at both low and high levels in the organization... likely making them more sustainable and resilient to reorgs and product changes."*
>
> *"To support productivity, organizations can foster a culture of psychological safety and make smart investments in tooling, information search, and reducing technical debt through flexible, extensible, and viewable systems."*

A culture of observability supports this kind of community and allows you to leverage its benefits.

What is a culture of observability?

At its core, a culture of observability is one that supports the behaviors and processes that drive observability: instrumentation that reflects the experience of the end user, tooling that supports rapid and collaborative debugging, extensive knowledge sharing, and shared ownership of production. When your organization develops a culture of observability, it means that the information needed to support a healthy production environment is available to everyone responsible for that environment.

Observability culture means that you might have subject matter experts but you don't have gatekeeping. It means all members of your team are comfortable going on call because they have a clear understanding of what to do and who to call in most situations, and a good set of tools and processes to figure out those things in all other situations.

Why develop a culture of observability?

A culture of observability (and the actual observability that comes along with it) doesn't just make engineers happier, it also drives a stronger business, one that is more aware of and responsive to the market.

When engineers feel responsibility for production and are supported in that endeavor, it makes sense for them to want to expand their ability to solve issues by increasing the context and content of their instrumentation based on the premises of observability--they collect more context about the goals of the software they ship rather than just trying to interpret its impact on the underlying infrastructure.

Engineers are more likely to think about the impact of the code if they 'own' it and are on call for it, as well as likely to be more interested in what other engineers are shipping. The more context an engineer has, the better they can understand how their code will interact with that of others.

This empowerment leads engineers to feel more of a stake in the team. Happy, invested engineers also attract higher-quality candidates, and strengthens your offers to those candidates when it's time to grow the team.

With greater commitment across the entire team, stronger teams can form. When asking questions and exploration becomes the norm, learning spreads across the team more effectively, and new and junior engineers onboard and level up more quickly.

Put simply, a culture of observability encourages team behaviors and production outcomes that strengthen your business and improve the experience of your customers.

Stepping back from hero culture

If observability could be said to be the antithesis of any culture commonly reflected within the ranks of folks responsible for production environments, it would be "hero" culture. The days of the lone sysadmin, cranky and put-upon, who would emerge to save the day when some issue felled the servers should be one of the past. Hero culture is a blocker for observability culture.

If significantly less than the majority of your team holds all the knowledge needed to troubleshoot your production environment, you're taking a business risk as well as a cultural one. It's not just a matter of resentment and burnout--modern distributed systems are exponentially more complex than those of just a few years ago. Relying on the capacity of one person or a small number of people to continue to understand and debug those systems as they scale and you add new features is implausible at best. Distributing that understanding, and access to the tools to use and expand it, prevents your organization from taking a huge hit to the brain trust when someone leaves.

Beyond the “hit by a bus” concern, having isolated individuals hoarding all the understanding of your systems is problematic because it means there’s less for the rest of the team to care about in terms of ownership--if one person takes care of all the major fires, no one else is really motivated to learn how to do it.

What software ownership means

Instead of relying on the resentment-, and isolation-driven hero culture described above, observability culture encourages software ownership. In turn, software ownership requires two things to be true:

1) Engineers are responsible for the code they ship to production and how it behaves.

2) Engineers are empowered to solve issues and improve the overall quality of the system and associated processes.

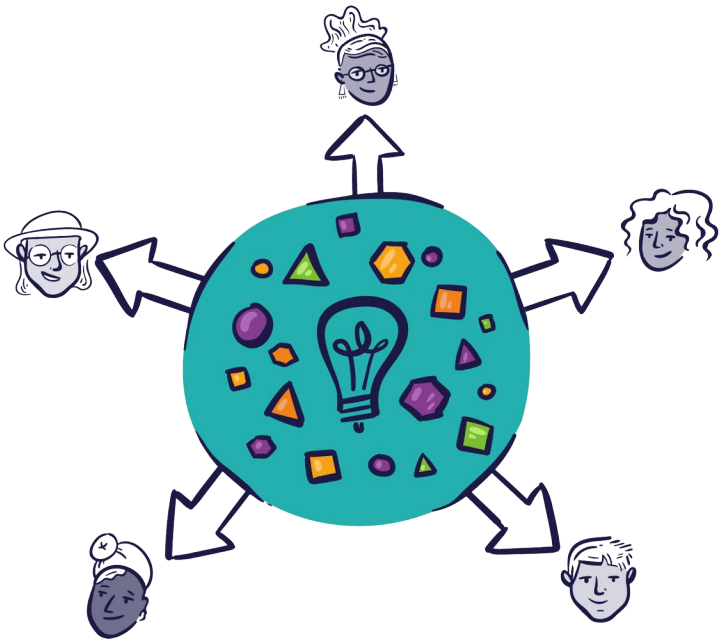
When engineers hear that they’re going to be responsible for their code in production, and perhaps join an on-call rotation, they typically assume the former (all the responsibility) will be true without the latter (the power to make things better), and dig in their heels--rightfully so, if that is in fact the case. If you want your engineers to own their software from development through deployment and into production, you must give them the tools and backing they need to effectively improve their world. Getting battered by the consequences of what’s been built but having no power to fix it, or only writing code and never experiencing the consequences of what you’ve made--either situation alone is a recipe for unhappy developers and ultimately, unhappy users as well. To solve this, you need ownership. As Liz Fong-Jones states in InfoQ:

> *“Under production ownership, operations become everyone's responsibility rather than “someone else's problem”. Every team member needs to have a basic fluency in operations and production excellence even if it's not their full-time focus. And teams need support when cultivating those skills and need to feel rewarded for them.”*

Observability is inherently democratizing. When the understanding needed to achieve production excellence is available to everyone instead of locked up in the heads of a few people, more people are empowered to solve problems, improve processes, serve and grow the business. One way to do this is to move the source of truth into a shared resource--the primary observability tool you're using. If the tooling your team is using doesn't support collaboration and learning from the experiences of your teammates, you lack the ability to persist what people have learned and benefit from it. More from Liz Fong-Jones:

> *"Production ownership, putting teams who develop components or services on call for those services, is a best practice for ensuring high-quality software and tightening development feedback loops. Done well, ownership can provide engineers with the power and autonomy that feed our natural desire for agency and high-impact work and can deliver a better experience for users."*

The craft of building software and the craft of owning it over its lifetime are effectively the same thing. The core of the craft of software design and engineering is maintainability. This should continue to matter to an engineer after their code is in production. When the knowledge to solve problems and the power to improve the quality of the system is equally available to everyone, everyone gets better at their jobs, better at sharing, better at learning from each other, and better at making working systems that we all understand--and can truly feel proud to own.



Behaviors and processes that encourage a culture of observability

The hardest thing about a discussion of the behaviors and processes that encourage a culture of observability is choosing an order in which to list recommendations. Each of these recommended approaches inform and complement each other, with combinations of several tactics being many times more effective together. So where to start? Good instrumentation is critical to the cause of observability overall--so we'll start there.

Encourage and reward useful instrumentation

If someone is going to operate and maintain a production system in the modern era, they need that system to communicate information that makes sense in terms of what the user is experiencing. They need access to context that aggregated log output won't provide. Work with your team to identify the information someone would need to debug the kinds of issues you're seeing or expecting to see, and begin an iterative, ongoing process of improving the instrumentation of the codebase.

Unsure how to start, or how to think about this process? Refer to the section titled "What should an event contain?" in Observability for Developers, our previous e-Guide, for some specific guidance on what should be included in your instrumentation.

Good instrumentation can help spread the culture of observability more broadly in your organization, as well as lighten the load of your immediate team. If the instrumentation provides enough context, the Support team might be able to address more issues without actually paging someone on your team.

Use failure as an opportunity to learn, not punish

Bugs happen. Outages happen. Nowadays, almost every team is engaging in some kind of incident review process, but just having a meeting where you talk about what went wrong isn't enough. The goal of your post-incident review should be to identify weaknesses in your systems and processes, and to encourage those responsible for your production environment to bring forward what they know about how to make it more resilient next time. Unfortunately, many organizations end up focusing on whose fault a given outage was, which leads to reticence on the part of those who have the most to contribute to future improvements.

As cited in the 2019 DORA Report, Kripa Krishnan, a director at Google previously in charge of disaster recovery testing (DiRT) exercises, makes the following observation:

> *"For DiRT-style events to be successful, an organization first needs to accept system and process failures as a means of learning. Things will go wrong. When they do, the focus needs to be on fixing the error instead of reprimanding an individual or team for a failure of complex systems."*

A culture of observability is a blameless culture. This doesn't mean no one takes responsibility for their actions--it means everyone does. Everyone is on the hook to help solve a problem, and that job is a lot easier when everyone's experience is brought to bear. When sharing what you know is a part of your culture and engineers have internalized that **what matters is improving the outcome**, your business reaps the benefit. When they're not worried about being blamed, on-call responders act more quickly during incident response, and are more likely to feel comfortable describing their part in what occurred. A culture of observability rewards people who are willing to take action and collaborate, and improves the reliability of the services you create.

No one brings production down on purpose. Use your incident review process to learn more about the frailties of your systems, and to inform how your team will handle this kind of problem in the future, not to highlight and punish those who were present when the mistake happened.

Use real data to prioritize work and make decisions

When decisions are made based on someone's ability to persuade, or be the loudest voice in the room, the voices of the rest of the team are diminished. This leads to feelings of disenfranchisement in those team members who are unheard. Beyond that, this approach increases the potential that your organization misses critical dissenting opinions before the wrong choices impact your business negatively.

With the clearer picture of both customer experience and service stability that observability delivers, you can make better decisions about when to invest in new features vs. infrastructure work. When an organization makes decisions with data rather than by fiat or force of personality, it supports greater participation from those with valuable context to provide. Team members feel more invested in the overall outcomes and their ability to effect change for the better.

Use SLOs to develop and enforce expectations

If you're making decisions using real data and insights from your observability practice, you're ready to define what success means and establish agreements with your internal and external partners and customers to reflect this understanding. Having such agreements in place offers your team the ability to make better decisions in the moment when incidents occur because they have a clear picture of the remaining error budget and can act accordingly. This level of understanding also supports less stressful decisionmaking when prioritizing work as described in the previous section -- the team knows decisions to increase product surface area vs. stabilize production will be made in the context of the SLO and allow them the capacity to support new functionality. A culture of

observability helps drive confidence in the decisionmaking process, and investment from those responsible for supporting the outcome.

Encourage and reward documentation efforts

A strong appreciation for, and enforcement of, ongoing documentation improvements is critical to establishing a culture of observability. Second only to context-rich instrumentation, thorough, up-to-date documentation about processes, escalation paths, past incidents, and anything else that provides useful context makes the job of supporting production much less stressful for everyone involved. To a very near degree, your instrumentation is documentation as well.

Documentation doesn't have to be runbooks or long lists of procedures and tables of configurations--what matters is that it's useful and accessible, as well as easy to contribute to. Documentation that increases observability culture can be things like notes and breadcrumbs on dashboards and the ability to search and re-run past queries by other teammates who may have experienced a given issue before. Incident reports in particular should be written with an eye to educate and support future folks on call, made searchable, and include links to past queries and graphs.

Many organizations designate some kind of documentation platform (typically a wiki) and start using it to describe systems and/or processes with the expectation that it will somehow magically keep itself updated as the service and architecture evolves. As most folks with any experience in the professional world have experienced, this rarely pans out. Even if lipservice is given to the idea of keeping things up to date, without enforcement or reward, documentation is usually the first thing to fall by the wayside when production incidents spike--and with it, a significant opportunity to keep that from happening in the future.

To be successful, documentation efforts must be treated as a first class citizen among the priorities of running a complex distributed production system--it should be someone's job to keep it organized, and it should be everyone's job to contribute what they have learned.

Establish more humane on-call processes

If engineers are to truly 'own' their code in production, then it follows that they should bear some of the responsibility for tending to it after it ships. It's not an absolute requirement that software engineers be on-call for a team to have a culture of observability; the thing that matters about code is how it behaves for users, in production—and by extension, for the people who are on-call for that code when it is running in production.

Why are there on-call rotations? So everyone can learn, and so the responsibility and ownership of production systems is distributed among the entire team instead of falling on the shoulders of a few folks. On-call rotations exist for the same reason that you round-robin DNS: it's about resiliency, and ultimately, sustainability. On-call rotations are the primary mechanism by which most teams address the question of sustainability.

In a culture of observability, having responsibility for production systems via being on call is only half of the equation--the other half is having the tools and processes and documentation in place to solve the problems you encounter and make the environment better for the next engineer on call. Everyone has heard or experienced on-call horror stories of the drudgery and sleepless nights, the inability to solve one issue before the next arises, the pressure from "management" to ship new features vs resolve stability issues, and so on--and when engineers are not empowered or encouraged to make things better, you cannot expect a culture of observability to develop under those circumstances.

> *"Nothing is sustainable if it is hateful to everyone who participates in it. Sustainable processes are low impact and use renewable resources. Burning people out is not sustainable. Making engineers dread being oncall is not sustainable. We have to make it suck less."* – Charity Majors

Being on call will always suck sometimes, but beyond the greater investment you should make in instrumentation and tools that support collaboration and fast debugging, consider auditing your on-call rotation processes and making changes that reflect your goals to develop an observability culture. As you develop your observability culture by investing in other areas such as

instrumentation, data-driven decisionmaking, and blameless incident reviews, reticence to join the on-call rotation will recede.

Define alert severities based on actual impact to the business

Organizations responsible for a production environment have historically defined alerts based on the information available to them about their systems. In the time before the cloud, this data was often limited to information about the state of the system, like disk space (remember that?) and performance, memory available, etc. As services have evolved into the complex distributed systems of today, commensurately more data is available to draw on to create alerts--meaning you can alert based on things that actually matter to your business rather than just trying to figure out what's going on based solely on its impact on your infrastructure.

In particular, if your instrumentation contains the kind of context that supports observability, it likely contains the data you need to develop alerts based on the actual experience of your users. With that level of understanding of the impact to the business of any given incident, it's possible to establish an alerting system that only wakes people up if it's mission-critical.

When you know you're not going to be paged unless it's actually important, it goes a long way toward feeling less put-upon when it does happen. Making sure alerts actually matter improves the quality of life for those on call, and is a clear signal that your organization values the time and expertise of your team. And a team that feels valued is happier and more invested in the goals of the organization that values them.

Review alerts frequently and remove them as often as possible

For many teams, the reality is that alerts defined in the monitoring system are almost never removed, only added to. There are sets of alerts that haven't fired in ages, other sets of alerts that just mean 'restart this service', and still further sets of alerts that fire and no one pays attention to because if they ignore it, it'll stop on its own. Alert exhaustion is common in environments like this. People burn out and leave, and others stay and become bitter. No one is benefiting from what is learned when an incident occurs or using that information to improve the outcome next time. This is the antithesis of an observability-driven environment.

Commit to a thorough review of the current set of defined alerts in your monitoring system—do the things you're monitoring actually relate to the end-user experience? Be ruthless in stripping out what has been hoarded. As discussed previously, only actually page someone if an end-user will notice, and optimally if there's something tangible the on-call person can do to resolve the problem. Add new triggers when you're seeing a new issue, and when you've identified and debugged the issue, **take them back out**.

Empower your team to remove or reconfigure alerts to minimize the impact on them while remaining focused on the customer experience. Consider making the pruning and improving of alerts a key task for whoever is currently oncall. As a side-benefit, on-boarding new people to the oncall rotation is a lot easier when someone doesn't have to explain which alerts to ignore. A culture of observability empowers the team to make the correct decisions for the business while maintaining a healthy environment for themselves.

Developing a culture of observability is an escalating, self-reinforcing process

Once a team has access to the context and power that rich instrumentation combined with observability tooling gives them, the behaviors and associated outcomes described here begin to augment and reinforce each other, much in

the way that bad habits and processes can. If your team feels ownership of a system and responsibility for outcomes AND also feels empowered to improve those outcomes, their motivation to improve all aspects of your observability practice increases. Better instrumentation, documentation, collaboration, and tooling leads to greater ease in debugging production issues. Easier debugging means cleaner and more maintainable/supportable code. Engineers are happier and more productive when code is cleaner and easier to maintain and support. Clean, easier-to-maintain code results in a more stable product and a more pleasant experience for folks on call and ultimately, for your customers. A culture of observability is good for business outcomes.

Where will you start?

About Honeycomb

Honeycomb provides next-gen APM for modern dev teams to better understand and debug production systems. With Honeycomb teams achieve system observability and find unknown problems in a fraction of the time it takes other approaches and tools. More time is spent innovating and life on-call doesn't suck. Developers love it, operators rely on it and the business can’t live without it.

Follow Honeycomb on Twitter LinkedIn
Visit us at Honeycomb.io