honeycomb.io

Framework for an Observability Maturity Model

Using Observability to Advance Your Engineering & Product

Charity Majors & Liz Fong-Jones

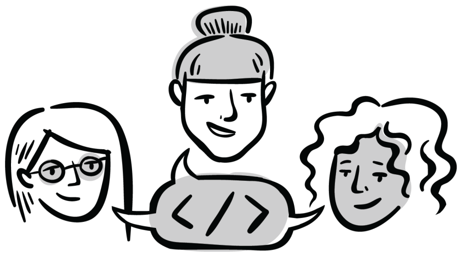
Introduction and goals

We are professionals in systems engineering and observability, having each devoted the past 15 years of our lives towards crafting successful, sustainable systems. While we have the fortune today of working full-time on observability together, these lessons are drawn from our time working with Honeycomb customers, the teams we've been on prior to our time at Honeycomb, and the larger observability community.

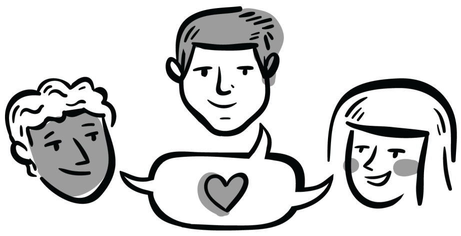
The goals of observability

We developed this model based on the following engineering organization goals:

- **Sustainable systems and engineer happiness**
  This goal may seem aspirational to some, but the reality is that engineer happiness and the sustainability of systems are closely entwined. Systems that are observable are easier to own and maintain, which means it's easier to be an engineer who owns said systems. In turn, happier engineers means less turnover and less time and money spent ramping up new engineers.



- **Meeting business needs and customer happiness**
  Ultimately, observability is about operating your business successfully. Having the visibility into your systems that observability offers means your organization can better understand what your customer base wants as well as the most efficient way to deliver it, in terms of performance, stability, and functionality.



The goals of this model

Everyone is talking about "observability", but many don't know what it is, what it's for, or what benefits it offers. With this framing of observability in terms of *goals* instead of *tools*, we hope teams will have better language for improving what their organization delivers and how they deliver it.

For more context on observability, review our e-guide "Achieving Observability."

The framework we describe here is a starting point. With it, we aim to give organizations the structure and tools to begin asking questions of themselves, and the context to interpret and describe their own situation--both where they are now, and where they could be.

The future of this model includes everyone's input

Observability is evolving as a discipline, so the endpoint of "the very best o11y" will always be shifting. We welcome feedback and input. Our observations are guided by our experience, and intuition and are not yet necessarily quantitative or statistically representative in the same way that the Accelerate State of DevOps[1] surveys are. As more people review this model and give us feedback, we'll evolve the maturity model. After all, a good practitioner of observability should always be open to understanding how new data affects their original model and hypothesis.

The Model

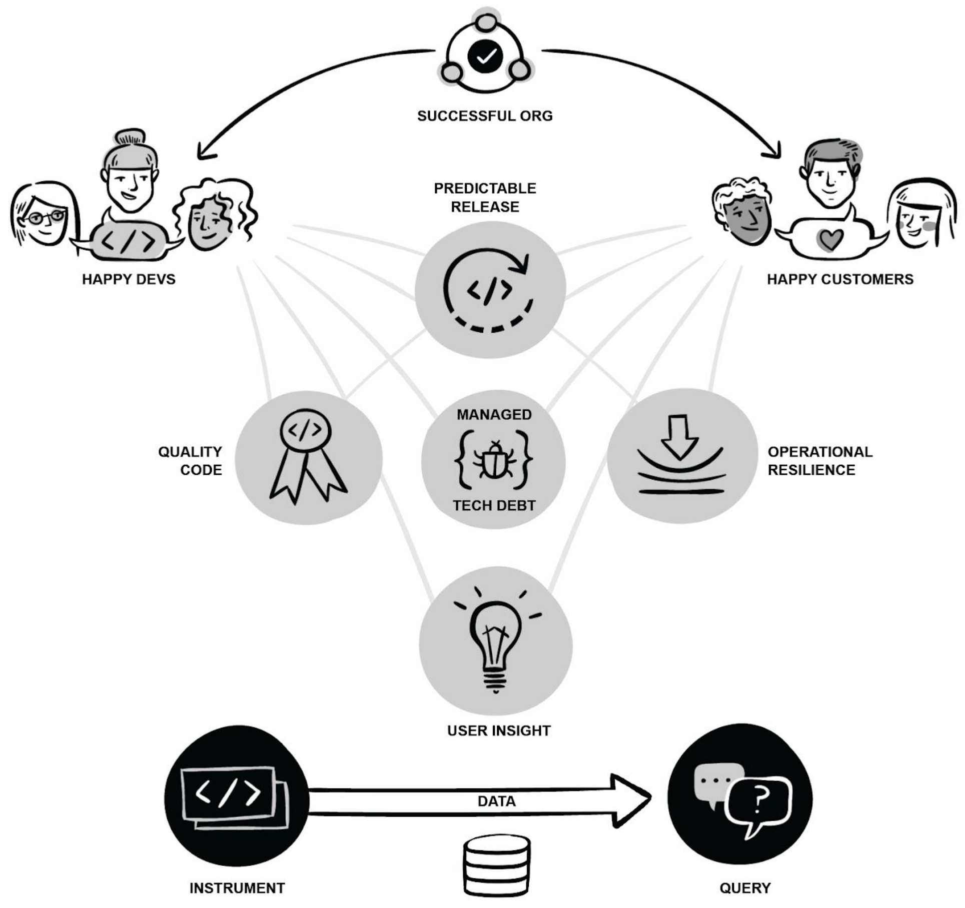
The following is a list of capabilities that are directly impacted by the quality of your observability practice. It's not an exhaustive list, but is intended to represent the breadth of potential areas of the business. For each of these capabilities, we've provided its definition, some examples of what your world looks like when you're doing that thing well, and some examples of what it looks like when you're *not* doing it well. Lastly, we've included some thoughts on how that capability fundamentally requires observability--how improving

[1] https://cloudplatformonline.com/2018-state-of-devops.html

your level of observability can help your organization achieve its business objectives.

The quality of one's observability practice depends upon both technical and social factors. Observability is not a property of the computer system alone or the people alone. Too often, discussions of observability are focused only on the technicalities of instrumentation, storage, and querying, and not upon how a system is used in practice.

If teams feel uncomfortable or unsafe applying their tooling to solve problems, then they won't be able to achieve results. Tooling quality depends upon factors such as whether it's easy enough to add instrumentation, whether it can ingest the data in sufficient granularity, and whether it can answer the questions humans pose. The same tooling need not be used to address each capability, nor does strength of tooling for one capability necessarily translate to success with all the suggested capabilities.



If you're familiar with the concept of production excellence[2], you'll notice a lot of overlap in both this list of relevant capabilities and in their business outcomes.

There is no one right order or prescriptive way of doing these things. Instead, you face an array of potential journeys. Focus at each step on what you're hoping to achieve. Make sure you will get appropriate business impact from making progress in that area right now, as opposed to doing it later. And you're never “done” with a capability unless it becomes a default, systematically supported part of your culture. We (hopefully) wouldn't think of checking in code without tests, so let's make o11y something we live and breathe.

[2] https://www.infoq.com/articles/production-excellence-sustainable-operations-complex-systems/

Respond to system failure with resilience

Definition

Resilience is the adaptive capacity of a team together with the system it supports that enables it to restore service and minimize impact to users. Resilience doesn't only refer to the capabilities of an isolated operations team, or the amount of robustness and fault tolerance in the software[3]. Therefore, we need to measure both the technical outcomes and people outcomes of your emergency response process in order to measure its maturity.

To measure technical outcomes, we might ask the question of "if your system experiences a failure, how long does it take to restore service, and how many people have to get involved?". For example, the 2018 Accelerate State of DevOps Report defines Elite performers as those whose average MTTR that is less than 1 hour and Low performers as those averaging an MTTR that is between 1 week and 1 month[4].

Emergency response is a necessary part of running a scalable, reliable service, but emergency response may have different meanings to different teams. One team might consider satisfactory emergency response to mean "power cycle the box", while another might understand it to mean "understand exactly how the automation to restore redundancy in data striped across disks broke, and mitigate it." There are three distinct goals to consider: how long does it take to detect issues, how long does it take to initially mitigate them, and how long does it take to fully understand what happened and decide what to do next?

But the more important dimension to managers of a team needs to be the set of people operating the service. Is oncall sustainable for your team so that staff remain attentive, engaged, and retained? Is there a systematic plan

[3] https://www.infoq.com/news/2019/04/allspaw-resilience-engineering/

[4] https://cloudplatformonline.com/2018-state-of-devops.html

to educate and involve everyone in production in an orderly, safe way, or is it all hands on deck in an emergency, no matter the experience level?[5] If your product requires many different people to be oncall or doing break-fix, that's time and energy that's not spent generating value. And over time, assigning too much break-fix work will impair the morale of your team.

If you're doing well	If you're doing poorly
● System uptime meets your business goals, and is improving. ● Oncall response to alerts is efficient, alerts are not ignored. ● Oncall is not excessively stressful, people volunteer to take each others' shifts ● Staff turnover is low, people don't leave due to 'burnout'.	● The organization is spending a lot of money staffing oncall rotations. ● Outages are frequent. ● Those on call get spurious alerts & suffer from alert fatigue, or don't learn about failures. ● Troubleshooters cannot easily diagnose issues. ● It takes your team a lot of time to repair issues ● Some critical members get pulled into emergencies over and over.

How observability is related

Skills are distributed across the team so all members can handle issues as they come up.

Context-rich events make it possible for alerts to be relevant, focused, and actionable, taking much of the stress and drudgery out of oncall rotations. Similarly, the ability to drill into highly-cardinal data[6] with the accompanying context supports fast resolution of issues.

[5] https://www.infoq.com/articles/production-excellence-sustainable-operations-complex-systems/

[6] https://www.honeycomb.io/blog/metrics-not-the-observability-droids-youre-looking-for/

Deliver high quality code

Definition

High quality code is code that is well-understood, well-maintained, and (obviously) has a low level of bugs. Understanding of code is typically driven by the level and quality of instrumentation. Code that is of high quality can be reliably reused or reapplied in different scenarios. It's well-structured, and can be added to easily.

If you're doing well

- Code is stable, there are fewer bugs and outages.
- The emphasis post-deployment is on customer solutions rather than support.
- Engineers find it intuitive to debug problems at any stage, from writing code to full release at scale.
- Issues that come up can be fixed without triggering cascading failures.

If you're doing poorly

- Customer support costs are high.
- A high percentage of engineering time is spent fixing bugs vs working on new functionality.
- People are often concerned about deploying new modules because of increased risk.
- It takes a long time to find an issue, construct a repro, and repair it.
- Devs have low confidence in their code once shipped.

How observability is related

Well-monitored and tracked code makes it easy to see when and how a process is failing, and easy to identify and fix vulnerable spots. High quality observability allows using the same tooling to debug code on one machine as on 10,000. A high level of relevant, context-rich telemetry means engineers can watch code in action during deploys, be alerted rapidly, and repair issues before they become user-visible. When bugs do appear, it is easy to validate that they have been fixed.

Manage complexity and technical debt

Definition

Technical debt is not necessarily bad. Engineering organizations are constantly faced with choices between short-term gain and longer-term outcomes. Sometimes the short-term win is the right decision if there is also a specific plan to address the debt, or to otherwise mitigate the negative aspects of the choice. With that in mind, code with high technical debt is code in which quick solutions have been chosen over more architecturally stable options. When unmanaged, these choices lead to longer-term costs, as maintenance becomes expensive and future revisions become dependent on costs.

If you're doing well

- Engineers spend the majority of their time making forward progress on core business goals.
- Bug fixing and reliability take up a tractable amount of the team's time.
- Engineers spend very little time disoriented or trying to find where in the code they need to make the changes or construct repros.
- Team members can answer any new question about their system without having to ship new code.

If you're doing poorly

- Engineering time is wasted rebuilding things when their scaling limits are reached or edge cases are hit.
- Teams are distracted by fixing the wrong thing or picking the wrong way to fix something.
- Engineers frequently experience uncontrollable ripple effects from a localized change.
- People are afraid to make changes to the code, aka the "haunted graveyard" effect.

How observability is related

Observability enables teams to understand the end-to-end performance of their systems and debug failures and slownesses without wasting time.

Troubleshooters can find the right breadcrumbs when exploring an unknown part of their system. Tracing behavior becomes easily possible. Engineers can identify the right part of the system to optimize rather than taking random guesses of where to look and change code when the system is slow.

Release on a predictable cadence

Definition

Releasing is the process of delivering value to users via software. It begins when a developer commits a change set to the repository, includes testing and validation and delivery, and ends when the release is deemed sufficiently stable and mature to move on. Many people think of continuous integration and deployment as the nirvana end-stage of releasing, but those tools and processes are just the basic building blocks needed to develop a robust release cycle--a predictable, stable, frequent release cadence is critical to almost every business[7].

If you're doing well

- The release cadence matches business needs and customer expectations.
- Code gets into production shortly after being written. Engineers can trigger deployment of their own code once it's been peer reviewed, satisfies controls, and is checked in.
- Code paths can be enabled or disabled instantly, without needing a deploy.
- Deploys and rollbacks are fast.

If you're doing poorly

- Releases are infrequent and require lots of human intervention.
- Lots of changes are shipped at once.
- Releases have to happen in a particular order.
- Sales has to gate promises on a particular release train.
- People avoid doing deploys on certain days or times of year.

[7] https://www.intercom.com/blog/shipping-is-your-companys-heartbeat/

How observability is related

Observability is how you understand the build pipeline as well as production. It shows you if there are any slow or chronically failing tests, patterns in build failures, if deploys succeeded or not, why they failed, if they are getting slower, and so on. Instrumentation is how you know if the build is good or not, if the feature you added is doing what you expected it to, if anything else looks weird, and lets you gather the context you need to reproduce any error.

Observability and instrumentation are also how you gain confidence in your release. If properly instrumented, you should be able to break down by old and new build ID and examine them side by side to see if your new code is having its intended impact, and if anything else looks suspicious. You can also drill down into specific events, for example to see what dimensions or values a spike of errors all have in common.

Understand user behavior

Definition

Product managers, product engineers, and systems engineers all need to understand the impact that their software has upon users. It's how we reach product-market fit as well as how we feel purpose and impact as engineers. When users have a bad experience with a product, it’s important to understand both what they were trying to do and what the outcome was.

If you’re doing well

- Instrumentation is easy to add and augment.
- Developers have easy access to KPIs for the business and system metrics and understand how to visualize them.
- Feature flagging or similar makes it possible to iterate rapidly with a small subset of users before fully launching.
- Product managers can get a useful view of customer feedback and behavior.
- Product-market fit is easier to achieve.

If you’re doing poorly

- Product managers don't have enough data to make good decisions about what to build next.
- Developers feel that their work doesn't have impact.
- Product features grow to excessive scope, are designed by committee, or don't receive customer feedback until late in the cycle.
- Product-market fit is not achieved.

How observability is related

Effective product management requires access to relevant data. Observability is about generating the necessary data, encouraging teams to ask open-ended questions, and enabling them to iterate. With the level of visibility offered by event-driven data analysis and the predictable cadence of releases both enabled by observability, product managers can investigate and iterate on feature direction with a true understanding of how well their changes are meeting business goals.

What happens next?

Now that you've read this document, you can use the information in it to review your own organization's relationship with observability. Where are you weakest? Where are you strongest? Most importantly, what capabilities most directly impact your bottom line, and how can you leverage observability to improve your performance?

You may want to do your own Wardley mapping to figure out how these capabilities relate in priority and interdependency upon each other, and what will unblock the next steps toward making your users and engineers happier.

For each capability you review, ask yourself: who's responsible for driving this capability in my org? Is it one person? Many people? Nobody? It's difficult to make progress unless there's clear accountability, responsibility, and sponsorship with money and time. And it's impossible to have a truly mature team if the majority of that team still feels uncomfortable doing critical activities on their own, no matter how advanced a few team members are. When your developers aren't spending up to 21 hours a week[8] handling fallout from code quality and complexity issues, your organization has correspondingly greater bandwidth to invest in growing the business.

Our plans for developing this framework into a full model

The acceleration of complexity in production systems means that it's not a matter of *if* your organization will need to invest in building your observability practice, but *when* and *how*. Without robust instrumentation to gather contextful data and the tooling to interpret it, the rate of unsolved issues will continue to grow, and the cost of developing, shipping, and owning your code will increase--eroding both your bottom line and the happiness of your team. Evaluating your goals and performance in the key

[8] "...the average developer spends more than 17 hours a week dealing with maintenance issues, such as debugging and refactoring. In addition, they spend approximately four hours a week on "bad code," which equates to nearly $85 billion worldwide in opportunity cost lost annually...."
https://stripe.com/reports/developer-coefficient-2018

areas of resilience, quality, complexity, release cadence, and customer insight provides a framework for ongoing review and goal-setting.

We are committed to helping teams achieve their observability goals, and to that end will be working with our users and other members of the observability community in the coming months to expand the context and usefulness of this model. We'll be hosting various forms of meetups and panels to discuss the model, and plan to conduct a more rigorous survey with the goal of generating more statistically relevant data to share with the community.

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](#)