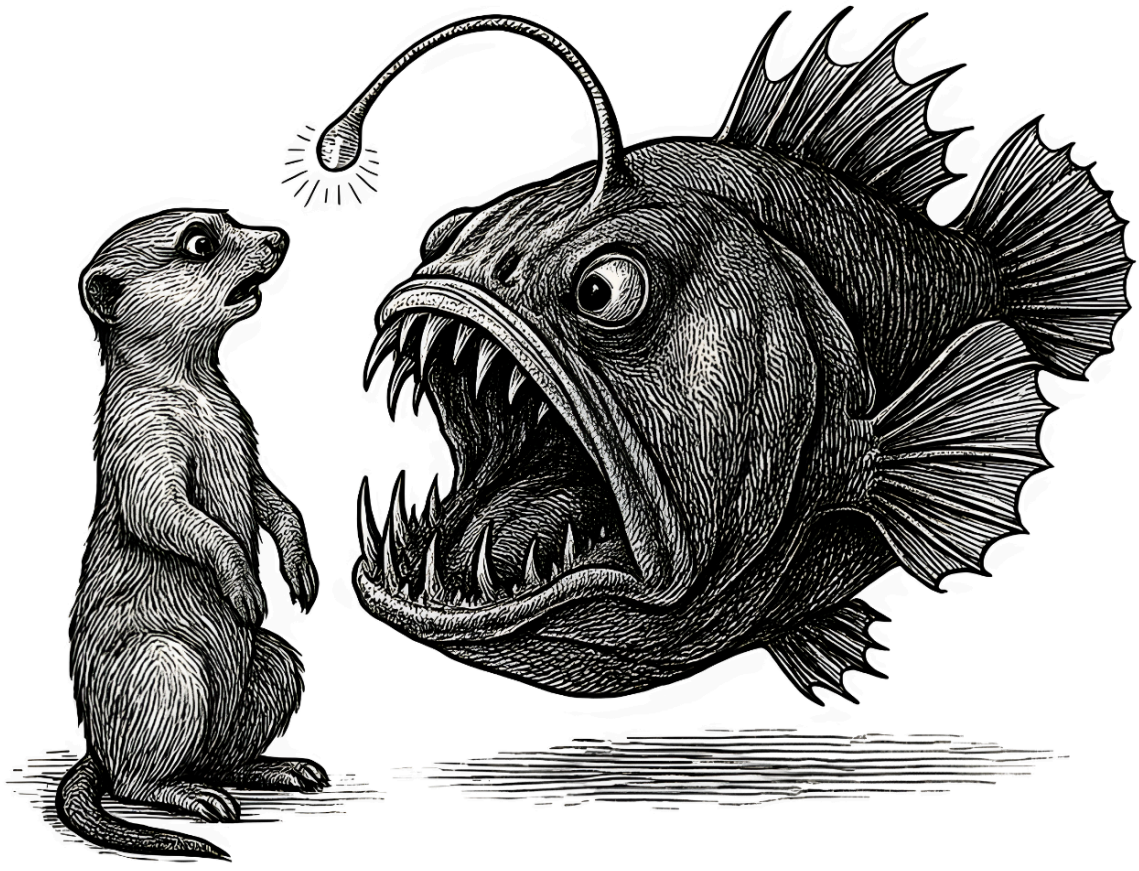
2nd Edition

On-Call In Action

Site Reliability Engineering Best Practices for Building Resilient Systems



Quan Huynh

On-Call In Action

Site Reliability Engineering Best Practices for Building Resilient Systems

Quan Huynh

Table of contents

Forewords

These days, it feels like we're always connected in an "always-on" world. From the moment we wake up to the moment we go to sleep (and sometimes even while we're asleep!), digital services are running in the background, shaping our lives.

We chat with friends across the globe, manage our finances with the phone, binge-watch our favorite shows, and depend on critical systems that need to be online 24/7. We've come to expect all of this to just work, all the time. This isn't just about convenience; it's a real demand that businesses have to meet if they want to succeed and keep their users happy. It puts a lot of pressure on organizations to keep everything running smoothly, protect their income, and build trust with their customers.

To achieve this kind of reliability, you need robust systems, careful monitoring, and, most importantly, a team of people ready to jump in when things go wrong. That's where "on-call" comes in. It means having tech experts—like Site Reliability Engineers (SREs), DevOps engineers, or operations staff—available when needed to handle problems quickly. They're the first line of defense when systems falter or something unexpected happens, and they're the ones who figure out what's wrong, halt the damage, fix the issue, or get the right people involved.

But being on-call isn't just about reacting to alerts. It also involves doing routine maintenance during shifts and, more strategically, working to make the systems better over time. The things you learn while handling incidents can help you improve the way systems are designed, make monitoring more effective, automate responses, and, ultimately, prevent the same problems from happening again.

Unfortunately, on-call practices are often put together without much thought, which can lead to messy processes, slow response times, and a lot of stress for engineers, sometimes even causing burnout. To make on-call work well and be sustainable, you need a plan.

This book, "On-Call In Action," is meant to be your guide to navigating the tricky world

of modern on-call. We'll cover the basics, explore practical ways to implement strategies and tools, with the open-source project as a key example, and dive into important concepts like Service Level Objectives (SLOs) for measuring reliability. We'll also talk about the human side of things, like scheduling, preventing burnout, and creating a supportive culture, all of which are essential for building on-call practices that are effective and sustainable. Whether you're a developer, operator, or manager dealing with systems that need on-call support, this book will give you the knowledge and best practices to create or improve an on-call system that actually works.

1

Foundations: Why On-Call Matters & SRE Principles

Welcome to the world of building reliable systems! In this chapter, we'll explore why having technical experts "on-call" is so important today and how a set of ideas called Site Reliability Engineering, or SRE, can help make this challenging job more manageable and effective.

1.1 The "Always On" World

Think about the apps and websites you use every day. Whether it's for banking, shopping, talking to friends, or getting work done, we all expect these digital services to be working perfectly, any time we need them – day or night, weekday or weekend. If a popular online store suddenly goes offline during a big sale, or if your company's internal communication tool stops working, it can cause a lot of frustration and even cost businesses money.

Because everyone expects services to be "always on," companies need a way to fix problems quickly when they happen. This is where being on-call comes in. Being on-call means that certain technical people in a company are ready to respond to issues, even if it's outside of their normal working hours. These are often engineers, perhaps with titles like Site Reliability Engineers (SREs) or DevOps Engineers. They are the first line of defense when something unexpected goes wrong with a live service. Their job is to jump in, figure out the problem, and get things back to normal as fast as possible.

1.2 Learning from Site Reliability Engineering (SRE)

Dealing with systems that need to be available all the time can be stressful.

Fortunately, there's a way of thinking about and managing technology that can make this much better. It's called **Site Reliability Engineering (SRE)**. SRE started at Google, and it's a set of practices and a mindset that treats the work of keeping systems running (often called "operations") like a software engineering challenge. This means using data, automation, and careful planning to build and maintain reliable services.

SRE offers some really useful ideas that can make on-call work more effective and less painful. Let's look at a few key ones:

It's Okay Not to Be Perfect

It might sound strange, but aiming for 100% perfect uptime for every service, all the time, is usually not a realistic or even a good goal. Why? Because trying to achieve absolute perfection is incredibly expensive and can slow down the process of adding new features or making improvements.

SRE recognizes this. Instead of chasing perfection, we define how much *unreliability* is acceptable. This "acceptable amount of downtime or errors" is called an error budget. For example, if a service aims to be available 99.9% of the time, its error budget is the remaining 0.1%. This budget gives teams a clear understanding of how much risk they can take. If the service is performing well and has plenty of error budget left, they might feel more confident releasing new features. If the error budget is used up, it's a signal to focus on stability and fix problems before making more changes. It’s all about finding a healthy balance.

Setting Clear Goals

To know if a service is reliable enough, we need clear goals. In SRE, these goals are called Service Level Objectives (SLOs). An SLO is a specific, measurable target for how well a service should perform from a user's perspective. For instance, an SLO might be "99.9% of login attempts should be successful" or "95% of search results should appear in under half a second."

SLOs are important because they:

- Tell everyone what "good enough" looks like.
- Help teams prioritize work (if an SLO is at risk, fixing it becomes a high priority).
- Provide a shared understanding between different teams (like development and

operations) about reliability expectations.

Good SLOs are based on what users actually care about.

Cutting Down on Repetitive Work

Imagine an on-call engineer having to manually restart a server every few hours because of a recurring minor issue, or spending a lot of time copying and pasting data for reports. This kind of manual, repetitive, and often boring work that doesn't add much long-term value is what SRE calls toil.

A big focus in SRE is to reduce or eliminate toil, usually by automating these tasks. If a task can be automated, it probably should be. Reducing toil is great because it:

- Frees up engineers from tedious work.
- Allows them to spend more time on projects that make the systems better and more reliable in the long run (like fixing the root cause of that recurring server issue!).
- Reduces the chance of human error in repetitive tasks.
- Makes on-call work less frustrating.

Learning Without Blame

When something goes wrong – and in complex systems, things inevitably will – it's natural to want to know why. However, it's very important *how* we approach this. SRE promotes a blameless culture. This means that when an incident happens, the focus is on understanding *what* happened in the system and *how* the processes or tools might have contributed to the problem, not on blaming individuals.

A blameless approach assumes that everyone was doing their best with the information and tools they had at the time. Why is this so important?

- It creates psychological safety: People feel safe to report problems, admit mistakes, and share information openly without fear of punishment.
- It leads to better learning: When people are honest, the team can get to the real root causes of issues and make meaningful improvements.
- It helps prevent the same problems from happening again.

Instead of asking "Who messed up?", we ask "What can we learn from this to make our systems and processes better?"

Mixing Project Work with Daily Tasks

To truly improve reliability and reduce future on-call pain, SRE teams aim to spend a significant portion of their time on engineering projects that have a long-term impact. This could be automating manual tasks, improving monitoring, redesigning parts of the system for better resilience, or developing new tools.

A common guideline in SRE (like at Google) is that engineers should spend at least 50% of their time on this kind of project work, with the other 50% (or less) on operational duties, including handling incidents. If a team is constantly firefighting and has no time for improvement projects, they get stuck in a reactive cycle, and the on-call burden rarely gets lighter. This balance is key to making on-call sustainable.

Fair Pay for On-Call Work

Being on-call is a significant responsibility. It can disrupt personal time, affect sleep, and add stress. SRE principles generally suggest that engineers who take on this extra duty should be compensated fairly for it. This acknowledges the effort and the importance of the role in maintaining service reliability.

1.3 What This Book Will Cover

Now that you have a basic understanding of why on-call is essential and how SRE principles can guide us, you might be wondering how to put all of this into practice. That's what the rest of this book is about!

We'll take you on a journey, chapter by chapter, through the key aspects of building and managing an effective and sustainable on-call system.

We will cover:

- **The Incident Lifecycle:** Understanding the stages from detection to resolution and learning (Chapter 2).
- **Effective Alerting:** Strategies for creating meaningful, actionable alerts while minimizing noise, including practical implementation using tools like Versus Incident (Chapter 3).
- **Integration and Escalation:** Connecting diverse monitoring sources and defining robust escalation policies (Chapter 4).
- **Measuring Reliability:** Defining and using SLIs, SLOs, and Error Budgets to drive decisions (Chapter 5).

- **Putting It All Together**: Practical Examples of Unified Alerting & Templating (Chapter 6).
- **Learning from Failure:** Implementing blameless postmortems for continuous improvement (Chapter 7).
- **The Human Element:** Designing fair schedules, managing burnout, and ensuring effective communication (Chapters 8 & 9).
- **The Ecosystem:** Navigating the tooling landscape and looking at future trends (Chapter 10).
- **Digital Customer Onboarding:** A case study illustrating the journey of building an on-call practice (Chapter 11).

Our goal is to give you the knowledge and confidence to build on-call practices that not only keep your services running smoothly but also support the well-being and growth of your engineering teams. Let's get started!