97 Things Every SRE Should Know

Collective Wisdom from the Experts

Emil Stolarsky and Jaime Woo

Copyrighted material

97 Things Every SRE Should Know

by Emil Stolarsky and Jaime Woo

Copyright © 2021 Incident Labs, Inc. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: John Devins

Developmental Editor: Corbin Collins

Production Editor: Beth Kelly

Copyeditor: nSight, Inc.

Proofreader: Shannon Turlington

Indexer: nSight, Inc.

Interior Designer: David Futato

Cover Designer: Randy Comer

Illustrator: Kate Dullea

November 2020: First Edition

Revision History for the First Edition

- 2020-11-18: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492081494* for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *97 Things Every SRE Should Know*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

The views expressed in this work are those of the authors, and do not represent the

Copyrighted material

publisher's views. While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

978-1-492-08149-4

[LSI]

Copyrighted material

Preface

If there is one defining trait of an SRE, it would be curiosity. There's something about trying to understand how a system works, bringing it back from failure, or generally improving it that tickles the parts of our brains where curiosity lives. This trait is probably common through most, if not all, engineering practices. There's a story we both love that seems to encompass this trait perfectly.

On November 14, 1969, as Apollo 12 was lifting off from its launchpad in Cape Canaveral, Florida, it was struck by lightning. Twice. First at 36.5 seconds after liftoff and then again at 52 seconds. Later the incident reports would show that the lightning had caused a power surge and inadvertently disconnected the fuel cells, leading to a voltage drop.

In the moment though, there was anything but clarity.

In an instant, every alarm in the Apollo 12 command capsule went off. Telemetry readings in Houston were complete gibberish. For an organization that thinks through everything, they never thought to ask what to do when lightning strikes. What were the chances?

Even worse, the stakes couldn't be higher. If the mission is aborted, NASA loses a $1.2 billion rocket. If not, and the safety of the astronauts is compromised, you end up broadcasting a catastrophe to the whole world. When listening back to a recording of mission control, you can feel the tension and stress.

There's a moment of silence on the audio loop before someone cuts in: "try SCE to Aux." This wasn't something ever tried before. So much so, someone radios back "what the hell is that?" With no better options, the command is relayed to the astronauts. And it worked. After searching for the switch, they flip it, and everything immediately returns back to normal.

The NASA engineer John Aaron gave the obscure suggestion. A year earlier he'd been working in an Apollo capsule simulator and ended up with a similar mess of telemetry readings. Rather than reset the simulator, he decided to play around and try fixing the problem. He'd discover that by shifting the signal conditioning electronics, or SCE, system to its auxiliary setting, it could operate in low-voltage conditions, restoring telemetry. SCE to Aux.

The lightning strike was a black swan event, something NASA had never simulated before. What inspired John Aaron to dig around to uncover the cause of that specific data signature? In an oral history with NASA, he credits a "natural curiosity with why things work and how they work."

Curiosity is a trait found in many SREs. We were reminded of a conversation with an SRE friend in Dublin who shared how she was the type to keep asking why about the systems she worked with. That echoes John Aaron talking about how he always wanted

Copyrighted material

to know how things around him worked, and not stopping until he had a deep understanding.

That willingness to learn makes sense for SREs, given the need to work with complex systems. The systems change constantly, and the role requires someone wanting to ask questions about how they work. The inquisitivity means rather than seeing one specific part of the system as their domain, SREs instead wonder about all the parts of the system, and how they function together.

But it's not just the technical system. SREs need to be curious about people too, the socio- part of the sociotechnical system. Without that, you couldn't bring different teams together to create meaningful SLOs. You couldn't navigate personality types to properly respond to incidents. You'd be satisfied with just the five whys and miss out on uncovering the lessons to be learned post-incident.

We want this book to give you an opportunity to explore, play, and satisfy your curiosity. Here, we've laid out essays to do so. (You may notice there are actually 98 essays! We figured everyone likes a little something extra on the house.) They're written by experts from across the industry, guiding you through a range of topics from the fundamentals of SRE to the bleeding edge. This book was written and edited during the pandemic, and we are deeply grateful for everyone who contributed during such a trying time.

We believe that SRE needs to be filled with many voices, and that new voices should always be welcome. New ideas from different points of view and a wide range of experiences will help evolve this field that is, honestly, remarkably still in its early days. Our dream is that as you read these essays, they spark your curiosity, and move you forward in your SRE journey, no matter where you're currently at.

We're beyond curious to read what a batch of essays on SRE will look like in 5 or 10 years.

How We Structured the Book

SRE, although it deals with complex technical systems, is ultimately a cultural practice. Culture is the product of people, and that inspired us to organize this book into sections based on the number of SREs you have in your organization—what you specifically tackle and how your day looks like depends on how many SREs there are. We've broken the book's essays into "New to SRE," 0-1 SRE, 1-10 SREs, 10-100 SREs, and the "Future of SRE."

Readers looking for guidance on where to start first can jump right to the section that applies most to them; however, you will still find value in reading essays from sections that don't currently apply to your day-to-day.

At 0 to 1 SRE, no one has been designated an SRE yet, or you have found your very first one, a role that can seem almost lonely.

At 1 to 10 SREs, you are forming a team, and there is sharing of knowledge and the

Copyrighted material

ability to divvy up work.

At 10 to 100 SREs, you have become an organization, and you need to think not just about the systems you're working on, but also about how you organize that many SREs.

"New to SRE" covers foundational topics (although not exhaustively!) and is helpful both for those just starting their SRE journeys as well as a refresher for even the most seasoned SRE. "Future of SRE" contains essays that look into where SRE is potentially headed, or are (for the moment) sitting on the zeitgeist.

There's no need to read the book in any particular order. You can read it from cover to cover. Or, if you are curious about a particular topic, flip to the index where you can find all the essays on that topic. Use this as a reference guide, or a source of inspiration—one that can provide a jolt as needed. Or, maybe create a reading club, where once a week you pick an essay to discuss with your coworkers. This is the beauty of a collection of essays. We hope you enjoy reading them as much as we did.

O'Reilly Online Learning

For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *http://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (in the United States or Canada)

707-829-0515 (international or local)

707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at *https://oreil.ly/97-SRE*.

Copyrighted material

Email *bookquestions@oreilly.com* to comment or ask technical questions about this book.

Visit *http://oreilly.com* for news and information about our courses and books.

Find us on Facebook: *http://facebook.com/oreilly*

Follow us on Twitter: *http://twitter.com/oreillymedia*

Watch us on YouTube: *http://youtube.com/oreillymedia*

Acknowledgments

Writing a book makes you acutely aware of the passage of time—even more so during a pandemic. During tumultuous times, you realize how important the people in your lives are. Some we can't hug until this pandemic is over. Others we won't be able to hug even after then. We hold dear thoughts of our loved ones here and beyond.

Copyrighted material

Part I. New to SRE

Copyrighted material

Site Reliability Engineering in Six Words

Alex Hidalgo

Nobl9

When someone I've just met asks me what I do for a living, I generally fall back to something along the lines of, "I'm a site reliability engineer. We keep large-scale computer services reliable." For many people, this is sufficiently boring and our general pleasantries continue. Occasionally, though, I run into people who are a bit more curious than that: "Oh, that sounds interesting! How do you do that?"

That's a difficult question to answer! What do SREs actually *do*? For many years, I'd rely on just listing an assortment of things—some of which have made their way into essays in this very book. Although an answer like that wasn't exactly *wrong*, it also never felt truly satisfying. There had to be a more cohesive answer, and when I reflect on my decade of performing this job, I think I've finally figured it out. Virtually everything SREs do relies on our ability to do six things: measure, analyze, decide, act, reflect, and repeat.

Measuring does not just mean collecting data. To *measure* something, you have some sort of goal in mind. You don't *collect* flour to bake a cake, you *measure* the flour; otherwise, things will end up a mess. SREs need to measure things because pure data isn't enough. Our data needs to be meaningful. We need to be able to answer the question, "Is this service doing what its users need it to be doing?"

Once you have measurements, the next step is to analyze them. This is when some basic statistics and probability analysis can be helpful. Learn as much as you can from the things you are measuring by using the centuries of study and knowledge mathematicians have made available to us.

Now you've done your best at measuring and analyzing how a certain thing is behaving. Use this analysis to make a decision about how best to move into the future!

Then you must act. You actually need to do the thing you decided to do. It could be that this action is actually to take no action at all!

Finally, reflect on what you did once you've done it. Place a critical—but blameless—eye squarely on whatever you've done. You can generally learn much more from this process than you can from your initial measurement analysis.

Now you start over. Something has either changed about the world due to your

Copyrighted material

decision or it hasn't, and you need to keep measuring to see what the real impact of this action, or inaction, actually was. Keep measuring and then analyze, decide, act, reflect, and repeat again and again. It's the SRE way. Incremental progress is the only reliable way to reliability.

Site reliability engineering is a broad discipline. We are often called on to be software engineers, system administrators, network engineers, systems architects, and even educators or consultants, but one paradigm that flows through all of those roles is that SRE is data-driven. Measure the things you need to measure, analyze the data you collect, decide what to do with this analysis, act on your findings, reflect on your decision, and then do it all over, again and again and again.

Measure, analyze, decide, act, reflect and repeat: that's site reliability engineering in six words.

Copyrighted material

Do We Know Why We Really Want Reliability?

Niall Murphy

Microsoft

Do we really understand reliability, or why we would want it?

This may seem like a strange question. It is an article of faith in this community that unreachable online services have no value. But even a moment's thought will show you that's simply not true. You yourself encounter intermittent computer failure almost every day. Some contexts even seem to expect it; with web services, users are highly accustomed to hitting refresh or (for more difficult problems) clearing cookies, restarting a browser, or restarting a machine. Even services themselves have retry protocols.

A certain amount of fudge is baked into every human–computer interaction. Even for longer outages, people almost always come back if you're down for a few minutes, and have even more patience, depending on the uniqueness of the service provided.

It's anecdotal, but suggestive: I had a conversation with a very well-known company a couple of years ago when they said they didn't put any money into reliability because their particular customer base had nowhere else to go. Therefore, time they spent on reliability would be time they wouldn't spend on capturing revenue; it wasn't worth it.

I gasped inwardly at the time, but I've thought about it often since, and I turn the question toward us, as a community, now: do we have any *real* argument against that statement, as a community and a profession? Can we put any numbers around it? Understand what the trade-offs are? Make anything other than emotive claims about brand image? Come up with a *real* explanation of why companies previously lambasted for their unreliability are worth tens of billions today, never mind companies where the inability to access the site costs real money, outages frequently last hours, yet usage, revenue, and profits keep going up?

I don't like it, but I think it's true; in a rising market, if a company could choose to acquire new customers or retain existing ones, every economic incentive is toward customer acquisition, since each customer lost would be replaced by many more gained. Of course, a systematically unreliable platform would *eventually* lose you as many customers as you acquired, but you have time to fix that, and customers are often reluctant to change, even given poor service.

Product developers know this, and this is why our conversations are so fraught. Yet we

Copyrighted material

don't have a fully satisfactory way to talk about these trade-offs today; the true value of reliability, particularly for markets that are not rising, non-web contexts, or other areas where SREs are not commonly found, is hard to articulate. The SLO model, which is meant to be able to articulate the nuances of precisely how much unreliability a given customer base can tolerate in the aggregate, is not actually sufficient; as typically used, it cannot distinguish between (say) 20 minutes of almost complete unavailability or two hours of intermittent unavailability. These situations are actually very different from the customer experience point of view and, potentially, also from the revenue generation point of view.

We have sparse data points that tenuously suggest the outlines of an approach that would enable us to understand, and argue successfully for why to spend time on reliability in the face of limited time and resources—or even worse, in a rising market—but we are very far from understanding it all.

This is therefore, depending on your point of view, quite worrying or a wonderful opportunity to stop spending a lot of time and money.

Copyrighted material

Building Self-Regulating Processes

Denise Yu

GitHub

In Camille Fournier's excellent book, *The Manager's Path* (O'Reilly, 2017), she advises readers to look for "self-regulating processes," which caught my eye. My undergraduate degree is in economics, and I jump at any opportunity to apply economic thinking to practical problem-solving. Self-regulating processes are tiny cycles of checks and balances, and it's cool to find them in human systems.

In my tech network, I often hear about process experiments succeeding or failing by the emotional or political bandwidth of the person who initiated the experiment. For example, when introducing pair-programming to a new group of engineers, it often takes a confident, charismatic person to coax reluctant teammates to start pairing for the first time.

In fact, they might not even call it pairing to begin with—they'll say, "Hey, do you wanna come over here and have a look at this with me?" But when that person leaves a company, pairing might fall by the wayside, because it was something driven by the strength of a personality. These short-lived process innovations are valuable, but they don't last; so in that context, we never learn how to adjust them, measure them, and scale them.

Self-regulating processes, on the other hand, don't depend on strong personalities to persist. The way that they work is by aligning incentives (both the positive and negative kind) in such a way that no one person is stuck with the unpleasant task of hassling other people to do their parts. Micromanagement represents exactly the opposite outcome of a self-regulating process.

To understand how to align incentives, let's talk about what incentives *are*. *Positive incentives* represent net gains for an individual if they behave in a certain manner. Think carrots, not sticks. They come in many flavors: financial (e.g., wages, stock awards), social (e.g., peer recognition), or intrinsic (e.g., mastery of a particular skill), to name a few.

Most people are driven by the positive incentive of wanting to earn more money, and perhaps wanting a better title. To facilitate that, most people, given that the organization exhibits more of a generative culture, would agree that receiving honest and constructive feedback from their peers is a good way to improve their performance.

Copyrighted material

Negative incentives are the opposite: net losses. Similarly, most people react to a set of negative incentives, such as wanting to avoid negative social repercussions and unnecessarily spending social capital. Consider that at companies with unlimited vacation policies, people end up taking fewer vacation days than their peers who accrue fixed vacation throughout the year. This is because a financial incentive structure became replaced by a social incentive structure, and the social anti-incentives feel more costly, in part because they're really hard to quantify, and we're wired to dislike uncertainty.

A self-regulating process sets up the right combination of positive incentives and negative incentives, so that people are intrinsically motivated to follow the process, and no external encouragement or facilitation is necessarily required once things get underway. Balancing positive with negative incentives is important: too much negativity and people will start to feel fearful; too much positivity and you bank on the assumption that everyone feels equally motivated by the same carrots. (That often is not true.)

In software engineering companies, and probably in other companies as well, I believe that you can design self-regulating processes if you stop and think about what incentives are in play.

Copyrighted material

Four Engineers of an SRE Seder

Jacob Scott

Stripe

During Seder, families recite a passage addressing the questions one might ask about the Passover holiday. The questions, presented from the points of view of four children, help pass the importance of the holiday down the generations. Here I present four software engineers asking about the importance of reliability.

The selfish engineer asks, “Why is your reliability so poor?” By using the word *your* and not *our*, the selfish engineer disclaims responsibility for reliability. Life is certainly easier when reliability is *your* job, not *our* job—but reliability is more and more frequently a collective responsibility.

To him, we must explain the importance, both to himself and to his team, of owning his code in production. As he decides what sort of observability to add to his features, which queries to make to data stores, or whether to push back on a resource-intensive feature request, this engineer—like every other—affects the behavior and reliability of production. None of us can avoid this power over production, and if we avoid responsibility for it, we implicitly place that burden on others. Given the importance and inevitability of this responsibility, we ask him to consider whether he might find more career growth and success in embracing responsibility than shirking it.

The junior engineer asks, “It works on my machine. Why isn’t that enough?” If only success in development environments implied success in production! To him, we sketch the vast difference between development and production. We might compare the scale and complexity of data in production to the limited, curated snapshot optimized for development. Or, we might contrast the sophisticated networking topology configured in production with the local and stubbed services in development that help him test and iterate quickly.

We suggest this engineer review a few of the spiciest or most mind-melting incident reports in our archive. Among the contributing factors whose confluence spawned these incidents, a few would certainly never show up (let alone reproducibly!) in a development environment.

The wise engineer, having responded to many incidents and read widely, asks, “How can error budgets prevent my next serious incident?” The oh-so-unfortunate truth is that error budgets are retrospective and cannot predict—let alone prevent—incidents.

To her, we note that although error budgets can’t predict or prevent incidents, they

Copyrighted material

provide a foundation for *preparing* for incidents. The process of defining error budgets creates alignment, transparency, and common ground about what reliability means, not just to engineers and users but also to executives, sales and marketing, front-line support, and the organization writ large.

We ask her to be curious about her error budgets and to reflect on what she learns about our users' desires for our system. Does she find that error budgets help elicit an active and ongoing discussion about the behavior of production? Over the long haul, this helps reduce the likelihood and impact of incidents.

Finally, the engineer who isn't sure how to frame their question asks, "Why is reliability important? Why should we be curious and passionate about it?" To them, we state that reliability is about systems behaving as expected, and users want software to be reliable! Availability—responding quickly and correctly to requests or, colloquially, not failing—is one common example. Users also want software to change and improve, often in the form of new features, better performance, or reduction in cost.

These desires are frequently in tension with each other, and he should reflect on SRE as an approach to quantifying reliability to help our entire organization understand the trade-offs involved.

Copyrighted material

The Reliability Stack

Alex Hidalgo

Nobl9

Think about your favorite digital media streaming service. You've settled down on the couch to watch a movie and you click a button on your remote. Most of the time, the movie buffers for a few seconds and then starts playing.

But what if the movie takes a full 20 seconds to buffer? You'd probably be a little annoyed in the moment, but ultimately, the rest of the movie streams just fine. Even with this little bit of failure, this service has still acted reliably for you, since the majority of the time it doesn't take anywhere near 20 seconds.

What happens if it takes 20 seconds to buffer every single time? Now things go from momentarily annoying to fully unreliable. With the plethora of digital media streaming services available, you might choose to abandon this service and switch to a different one.

Nothing is ever perfect and nothing can ever be 100% reliable. This is not only the way of the world, it also turns out that people are totally fine with this! No one actually expects computer systems to run perfectly all the time; we just need them to be reliable enough often enough.

How do we figure out the right level of reliability? This is where the reliability stack comes into play. It's made up of three components: SLIs (service level indicators), SLOs (service level objectives), and error budgets.

At the base of the reliability stack are SLIs, which are measurements of your service from your users' point of view. Why users? Because that's who your system has to perform well for. Your users determine whether you're being reliable. No user cares whether things look good from your end if their movies take 20 seconds to buffer every single time. An example SLI might be, "Movies buffer for 5 seconds or less."

Next are SLOs themselves. SLOs are fueled by SLIs. If SLIs are measurements about how your service is operating, SLOs are targets for how often you want them to be operating well enough. Using our example, you might now want to say something like, "Movies buffer for 5 seconds or less 99% of the time." If buffer times exceed 5 seconds only once in 100 times, people will probably be okay with this.

Nothing is ever perfect, so don't aim for it. Ensure instead that you're aiming to be reliable just enough of the time. You'll spend an infinite number of resources—both financial and human—trying to aim for perfection.

Copyrighted material

Finally, at the top of the reliability stack are error budgets, which are informed by SLOs and are simply a measurement of how you've performed against your target over a period of time. It's much more useful to know how you've performed from your users' perspective over a week, a month, or a quarter than simply knowing how you're performing right now. An error budget lets you say things like, "We cannot buffer reliably for 7 hours, 18 minutes, and 17 seconds every 30 days." You can use error budgets to think more holistically about the reliability of your service. Use this data to have better discussions and make better decisions about addressing reliability concerns.

You can't be perfect, and it turns out no one expects you to be perfect anyway. Use the reliability stack to ensure that you're being reliable *enough*.

Copyrighted material

Infrastructure: It's Where the Power Is

Charity Majors

Honeycomb.io

"Why infrastructure, why ops?" a coworker asked me, years ago. It was a software engineer, after a particularly gnarly on-call rotation, and the subtext was crystal clear: was I tricked into making this career choice—the sacrifice of being tethered to a pager, the pressure of being the debugger of last resort? Who would ever choose this life?

Without missing a beat, I answered: "Because that's where the power is." Then I stopped in surprise, hearing what I had said. We aren't used to thinking of infra as a powerful role. CS (computer science) departments, the media, and the popular imagination all revolve around algorithms and data structures, the heroic writer of code and shipper of features.

To business people, operations is a cost center, an unfortunate necessity. This is a historical artifact; operations should be seen as yin to development's yang, united and inseparable, never "someone else's job." Biz is the why, dev is the what, and ops is the how. Whether your company has one person or one thousand.

Code is ephemeral. Features come and go. Crafting a product in a modern development environment feels to me like erecting cloud castles in the sky: abstractions atop other abstractions, building up this rich mental world in your mind.

Software engineers are modern magicians, crafting unthinkably complex spells and incantations that spin gold from straw, generating immense real value practically out of thin air. But what happens when those spells go wrong?

A couple of years into my first job as a sysadmin, I started to notice a pattern when very senior engineers would come to me and the other ops people. They understood their code far better than I did, but when it stopped working in production, they would panic. Why didn't it work like it did yesterday? What changed? It was as though production were a foreign land, and they needed me to accompany them as a translator.

I always had crushes on the people who could turn "it's slow" into "the query planner is doing multiple full-table scans because it is using the wrong compound index." Any of us could see that it was slow; explaining *why* was next-level interesting.

Software can seem as mysterious and arcane as any ritual of the occult, but infrastructure engineers have a grimoire of tools to inspect the ritual relentlessly from every possible angle. Trace the library calls, scan the ports, step through the system

Copyrighted material

calls, dump the packets.

Infrastructure tools remind us that software operates according to the laws of scientific realism. Every mystery will yield an answer if pursued with enough persistence. To do so requires a world-weary fearlessness when things go wrong. The harder and more subtle the bug, the more interested and energized they become. Infra engineers have never seen an abstraction we trust to work as designed. The grander the claim, the more pessimistic we become.

We aren't so much cynical as we are grimly certain that everything will fail, and it will fall to us to save the world with nothing but a paper clip and a soldering iron. We compulsively peek under the lid to see what horrifying things are being done in the name of monkey patching.

When we get together with other infrastructure engineers over a pint, we boast about the outages we have seen, the bugs we have found, and the you-won't-believe-what-happened-last-holiday stories.

There is power in knowing how to be self-sufficient, in having the tools and the fearlessness, to track the answer down through layer after layer of abstractions. At the base of every technical pile sits the speed of light, which cannot be messed with or mocked up.

Copyrighted material

Thinking About Resilience

Justin Li

Shopify

In resilient systems, important variables stay in their desired state even when other variables leave their normal state. For example, many animals are able to avoid dying from minor cuts. When skin is cut, unprotected blood-carrying tissue is exposed, yet blood loss quickly trends back to zero as a clot forms. Improving a system's resilience makes dependent variables describing that system more independent.

Networked systems are often required to respond quickly, expressed as a state like this: *99th percentile latency below one second.* Ideally, this is held true all the way to the required limits of the system, for instance, *1s peak request rate of 100000 per second*. We want to ensure that the *latency* variable isn't too dependent on the *request rate* variable.

Here are ways we improve resilience:

Load reduction
: Throttling, load shedding/prioritization, queuing, load balancing

Latency reduction
: Caching, regional replication

Load adaptation
: Autoscaling, overprovisioning

Resilience (specifically)
: Timeouts, circuit breakers, bulkheads, retries, failovers, fallbacks

Meta-techniques
: Improving tooling, perhaps to scale up or fail over faster; especially impactful in cases when slow humans are in a system's critical path

Some of these tools are not usually associated with resilience (they are general optimization techniques), but all influence the dependence of critical variables. Sometimes they interact in useful ways. For example, retries can correct for transient downtime caused by a failover.

These tools also recur at multiple layers. TCP retransmission works against packet loss, but application-level retries are also used, because TCP can't retry an entire stream (among other reasons).

Let's continue the latency example. In practice, the relationship between request rate and latency is not linear but usually follows some rational function. Until a certain load

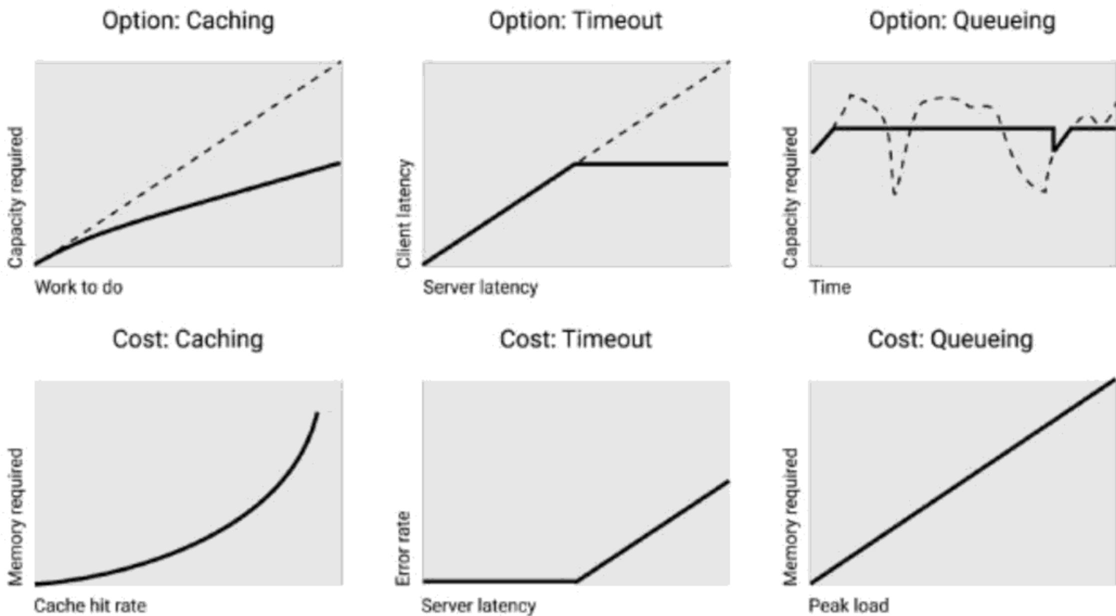
Copyrighted material

is reached, the system is unsaturated and can respond quickly, but when load approaches capacity, queues quickly fill up and latency grows accordingly.

We can scale the system by adding servers, which stretches the function horizontally, allowing more requests to be served before violating the latency objective. This costs money. If we don't like that, we can look at other options, such as load shedding: drop work (limit *request rate*) when the system is overloaded (*latency* reaches its limit).

Errors have a monetary impact too, but it might be less than paying for more servers if this condition is rare enough. The cost can be reduced further by dropping unimportant work first. Most important, the load-shedding approach entirely prevents unbounded latency growth, avoiding potential cascading failure.

You can think about every resilience tool as illustrated below:



By building in resilience, we can help increase reliability so the system bounces back and continues to function under adverse conditions.

Copyrighted material

Observability in the Development Cycle

Charity Majors and Liz Fong-Jones

Honeycomb.io

Catching bugs cleanly, resolving them swiftly, and preventing them from becoming a backlog of technical debt that weighs down the development process relies on a team's ability to find those bugs quickly. Yet software development teams often hinder their ability to do so for a variety of reasons.

Consider organizations where software engineers aren't responsible for operating their software in production. Engineers merge their code into master, cross their fingers that this change won't be one that breaks prod, and wait to get paged if a problem occurs. Sometimes they get paged soon after deployment. The deployment is then rolled back and the triggering changes can be examined for bugs. More likely, problems wouldn't be detected for hours, days, weeks, or months after that code had been merged. By that time, it's extremely difficult to pick out the origin of the bug, remember the context, or decipher the original intent behind why that code was written or why it shipped.

Resolving bugs quickly depends critically on being able to examine the problem *while the original intent is still fresh in the original author's head*. It will never again be as easy to debug a problem as it was right after it was written and shipped. It only gets harder from there; speed is key. At first glance, the links between observability and writing better software may not be clear, but it is this need for debugging quickly that deeply intertwines the two.

Newcomers to observability often make the mistake of thinking that observability is a way to debug your code, similar to using highly verbose logging. Although it's possible to debug your code, using observability tools, that is not the primary purpose of observability. Observability operates on the order of *systems*, not on the order of *functions*. Emitting enough detail at the lines level to debug code reliably would emit so much output that it would swamp most observability systems with an obscene amount of storage and scale. It would simply be impractical to pay for a system capable of doing that because it would likely cost somewhere in the ballpark of 1X–10X as much as your system itself.

Observability is not for debugging your code logic. *Observability is for figuring out where in your systems to find the code you need to debug*. Observability tools help you narrow

Copyrighted material

down swiftly where problems may be occurring. From which component did an error originate? Where is latency being introduced? Where was a piece of this data munged? Which hop is taking up the most processing time? Is that wait time evenly distributed across all users, or is it only experienced by a subset thereof? Observability helps your investigation of problems pinpoint likely sources.

Often, observability will also give you a good idea of what might be happening in or around an affected component or what the bug might be, or even provide hints to where the bug is actually happening—your code, the platform's code, or a higher-level architectural object.

Once you've identified where the bug lives and some qualities about how it arises, observability's job is done. If you want to dive deeper into the code itself, the tool you want is a debugger (for example, gdb). Once you suspect how to reproduce the problem, you can spin up a local instance of the code, copy over the full context from the service, and continue your investigation. Though related, the difference between an observability tool and a debugger is an order of scale; like a telescope and a microscope, they are primarily designed for different things.

Adapted from the upcoming book Observability Engineering, expected in 2021 from O'Reilly.

Copyrighted material

There Is No Magic

Bouke van der Bijl

When working with computers it's easy to get overwhelmed with the complexity of it all. You write some code, run it through a compiler, and execute it on your machine. It seems like magic.

But when issues occur and things break down, it's important to remember that there is no magic. These systems we work with are designed and built by humans like you, and that means that they can also be understood by humans like you. At every step, from the interface on the screen to the atoms your processor is built out of, someone considered how things should work.

I tend to work on two layers at the same time: the code I'm writing and the lower-level code I'm using. I switch back and forth between my work in progress and the source code of the Ruby gem, the Go compiler, or even a disassembly if the source is not available. This gives me context about my dependency: are there comments explaining weird behavior? Should I be using a different function mentioned in the code? Maybe an argument that wasn't immediately clear from the docs, or even a glaring bug?

I find this context switching to be a sort of superpower: X-ray goggles for the software developer. You can look deeper many times: from your code, to the virtual machine running it, to the C language it's written in, to the assembly that finally runs. Even then, you can read the Intel x86 manual to try to figure out what happens in the machine and how the various instructions are encoded. Software systems are fractal in nature—every component a world in itself.

Of course, just because all these things are created by people like us doesn't mean that it's possible for one person to understand it all. We stand on the shoulders of thousands of giants, and millennia of hours have been put into the systems to get where we are today.

It would take many lifetimes to know deeply every single step from atoms to GUIs, and that can be intimidating, but it doesn't mean we shouldn't try.

When you assume that the components we build our software from are mysterious scriptures that you can't understand or change, you will make uninformed decisions that don't account for the actual situation. Instead, you need to be more clear-eyed. You need to work with the quirks of the underlying system and use them to your advantage instead of paving over them.

Copyrighted material

So next time a library you use does something unexpected, take that extra step and pop open the hood. Poke and prod at the internals, look around, and make some changes. You will end up pleasantly surprised finding whole new worlds to explore and improve.

Copyrighted material

How Wikipedia Is Served to You

Effie Mouzeli

Wikimedia Foundation

According to Wikipedia, "Wikipedia is a multilingual, web-based, free-content encyclopedia project supported by the Wikimedia Foundation and based on a model of openly editable content." Serving billions of page views per month, Wikipedia is one of highest-traffic websites in the world. Let me explain what happens when you are visiting Wikipedia to read about Saint Helena or llamas.

First, these are the three most important building blocks of our infrastructure:

- The CDN (content delivery network), which is our caching layer
- The application layer
- Open-source software

When you request a page, the magic of our geographic DNS and internet routing sends this request to the nearest Wikimedia data center, based on your location, while with the wizardry of TLS, ATS (Apache Traffic Server) encrypts your connection. Each data center has two caching layers: in-memory (Varnish) and on disk (ATS). Most requests terminate here, because the hottest URLs are always cached. In case of cache misses, the request will be forwarded to the application layer, which might be very near if this is a primary data center, or a bit farther away if this is a caching point.

Our application layer has MediaWiki at its core, supported by a number of microservices and databases. MediaWiki is an Apache, PHP, MySQL open-source application, developed for Wikipedia. MediaWiki will look for a rendered version of the article initially on Memcached and, if not found, then on a MariaDB database cluster called Parser Cache.

If MediaWiki gets misses from Memcached and Parser Cache, it will pull the article's Wikitext and render it. Articles are stored in two database clusters: the Wikitext cluster, where Wikitext is stored in blobs, and the metadata cluster, which tells MediaWiki where an article is located in the Wikitext cluster. After an article is rendered, it is stored in turn in all aforementioned caches and, of course, is served back to you.

Things are slightly simpler when the request is a media file rather than a page. On a cache miss in the caching layer, ATS will directly fetch the file from Swift, a scalable object storage system by OpenStack.

Copyrighted material

As you can see, MediaWiki is surrounded by a very thick caching layer, and the reason is simple: rendering pages is costly. Furthermore, when a page is edited, it needs to be invalidated from all these caches and then populated again. When very famous people die, our infrastructure experiences a phenomenon called celebrity death spikes (or the Michael Jackson effect [1]). During this event, everyone links to Wikipedia to read about them while editors are spiking the edit rate by constantly updating the person's article. Eventually, this could cause noticeable load as heavy read traffic focuses on an article that's constantly being invalidated from caches.

The final building block is our use of open-source software. Everything we run in our infrastructure is open source, including in-house developed applications and tools. The community around the Wikimedia movement is not only limited to caring for the content in the various projects, its contribution extends to the software and systems serving it. Open source made it possible for members of the community to contribute; it is an integral part of Wikipedia and one of the driving forces behind our technical choices. Wikipedia obeys Conway's law in a way: a website that promotes access to free knowledge runs on free software.

It might sound surprising that one of the most popular websites is run using only open-source software and without an army of engineers—but this is Wikipedia; openness is part of its existence.

[1] Thomas Steiner, Seth Hooland, and Ed Summers. (2013). MJ no more: Using concurrent Wikipedia edit spikes with social network plausibility checks for breaking news detection, 791–794. 10.1145/2487788.2488049.

Copyrighted material

Why You Should Understand (a Little) About TCP

Julia Evans

Wizard Zines

I'd like to convince you that understanding a little bit about TCP (like how packets work and what an ACK is) is important, even if you only have systems that are making regular boring HTTP requests. Let's start with a mystery I ran into at work: the case of the extra 40 milliseconds.

One day, someone mentioned in Slack, "Hey, I'm publishing messages to NSQ and it's taking 40 ms each time." A little background: NSQ is a queue. The way you publish a message is to make an HTTP request on localhost. It really should not take 40 *milliseconds* to send an HTTP request to localhost. Something was terribly wrong. The NSQ daemon wasn't under high CPU load, it wasn't using a lot of memory, it didn't seem to be in a garbage collection pause. Help!

Then I remembered an article I'd read a week before, called, "In Search of Performance: How We Shaved 200 ms Off Every POST Request." That article described how the combination of two TCP features (delayed ACKs and Nagle's algorithm) conspired to add a lot of extra time to every POST request.

Here's how delayed ACKs plus Nagle's algorithm can make your HTTP requests slow. I'll tell you what was happening in the blog post I read. First, some background about their setup:

- They had an application making requests to HAProxy.
- Their HTTP library (Ruby's Net::HTTP) was sending POST requests in two small packets (one for the headers and one for the body).

Here's what the TCP exchange looked like:

1. client: hi! here's packet 1.
2. server: <silence>. ("I'll ACK eventually but let's just wait for the second packet.")
3. client: <silence>. ("I have more data to send but let's wait for the ACK.")
4. server: ok i'm bored. here's an ACK.

Copyrighted material

5. client: great here's the second packet!!!

That period while the client and server are both passive-aggressively waiting for the other to send information? That's the extra 200 ms! The client was waiting because of Nagle's algorithm, and the server was waiting because of delayed ACKs.

Delayed ACKs and Nagle's algorithm are both enabled by default on Linux, so this isn't that unusual. If you send your data in more than one TCP packet, it can happen to you.

The solution is TCP_NODELAY. When I read this article, I thought, "That can't be my problem, can it? Can it? The problem can't be with TCP!" But I'd read that you could fix this by enabling TCP_NODELAY on the client, a socket option that disables Nagle's algorithm, and that seemed easy to test, so I committed a change, turning on TCP_NODELAY for our application, and BOOM. All of the 40 ms delays instantly disappeared. Everything was fixed. I was a wizard.

You can't fix TCP problems without understanding TCP. I used to think that TCP was really low-level and that I did not need to understand it—which is mostly true! But sometimes in real life, you have a bug, and that bug is because of something in the TCP algorithm. I've found that in operations work, a surprising number of these bugs are caused by a low-level component of my system that I previously thought was obscure and suddenly have to learn a *lot* more about very quickly.

The reason I was able to understand and fix this bug is that, two years earlier, I'd spent a week writing a toy TCP stack in Python to learn how TCP works. Having a basic understanding of the TCP protocol and how packets are ACKed really helped me work through this problem.

Copyrighted material

The Importance of a Management Interface

Salim Virji

Google

During an outage, you care more about being able to control the system than about the system answering all user-facing requests. By adapting the concept of a *control plane* from networking hardware, engineers can separate responsibility for data transmission from control messages. The control plane provides a uniform point of entry for administrative and operational tasks, distinct from sending user data itself. For reliability purposes, this separation provides a way for operators to manage a system even when it is not functioning as expected. Let's look at why this is important and how you know when to separate these parts of a system.

In an early version of the GFS (Google File System), a single designated node was responsible for all data lookups: each of the thousands of clients began their request for data by asking this single node for the canonical location. This single node was also responsible for responding to administrative requests such as, "How many data requests are in the queue right now?" The same process was responsible for these two sets of requests—one user-facing and critical and the other strictly internal and also critical—and the process served responses to both from the same thread pool. This meant that when the server was overloaded and unable to process incoming requests, the SREs responsible for the system were unable to send administrative requests to lighten the load!

Previous versions of GFS had never been overloaded in this way due to client demand, which was why the request contention had not been apparent. In the next version, we separated the resources responsible for operations in the critical path from resources for administrative action using a control plane, and GFS production quality was able to take a significant step forward.

By extending this notion across multiple services, the benefits of a single administrative programming interface become apparent: software for automation can send an "update to new version" instruction to a heterogeneous group of servers, and they can interpret it and act accordingly. By dropping the networking nomenclature, we separate our requests into a *management* layer and a *data* layer and see the importance of separating the two for any service in the critical path. By drawing a boundary between user-facing operations, we can also have more confidence in the instrumentation we apply to the data measurements; operations in the data layer will

Copyrighted material

use and measure resources in that layer and not mingle with operations in the management layer. This in turn leads to a more successful approach to measuring user-facing operations, a useful metric for service level objectives.

How do you know when you have properly isolated administrative requests from user requests? Tools such as OpenTracing might show the full path of a management call as well as a user request, possibly exposing unintended interactions. Indeed, your systems will likely have connection points such as where the management interface actually influences user paths. Although the separation is not total and absolute, an SRE should be able to identify the boundaries between these parts of the systems they build and operate.

To implement this separation for software that's already built, such as third-party applications, you may need to add a separate service that, like a sidecar, attaches to the core software and, through a common interface such as an HTTP server, provides an endpoint for the administrative API. This glued-on management layer may be the precursor to eventual integration with the core software, or it might be a long-term solution. This approach to system design separates the request paths servicing user-facing requests from the requests providing management responsibility.

Copyrighted material

When It Comes to Storage, Think Distributed

Salim Virji

Google

Almost every application, whether on a smartphone or running in a web browser, generates and stores data. As SREs, we often have the responsibility for managing the masses of information that guide and inform decisions for applications as wide-ranging as thermostats to traffic-routing to sharing cat pictures. Distributed storage systems have gained popularity because they provide fault-tolerant and reliable approaches to data management and offer a scalable approach to data storage and retrieval.

Distributed storage is distinct from storage appliances, storage arrays, and storage physically attached to the computer using it; distributed storage systems decouple data producers from the physical media that store this data. By spreading the risk of data storage across different physical media, the system provides *speed* and *reliability,* two features that are fundamental to providing a good user experience, whether your user is a human excitedly sharing photographs with family and friends around the world or another computer processing data in a pipeline.

Distributed storage enables concurrent client access: as the system writes the data to multiple locations, there's no single disk head to block all read operations. Additional copies of the data can be made asynchronously to support increased read demand from clients if the data becomes really hot, such as a popular video. This is an example of the horizontal scaling made possible by a distributed system; although RAID (redundant array of independent disks) systems keep multiple copies of the data, they are not available for concurrent client reads in this same way.

As an additional benefit, building applications on top of distributed storage systems means that organizations don't have to post "Our service will be unavailable tonight from 3–4 for scheduled maintenance" while operators apply a kernel patch or other critical upgrade to the storage device. There is no single storage device; there is a storage *system*, with replication and redundancy.

The promise of modern web services, a *globally consistent* view of data, whether for a single user or for a large organization, would be almost impossible to implement without distributed storage systems. Previously, this required expensive device-to-device synchronization, essentially copying disks or directory trees from one specific computer to another; each was a single point of failure (SPOF).

Copyrighted material

Fault tolerance forms a key part of reliability; by sharing risk across different devices, distributed storage systems tolerate faults that storage appliances cannot. Although storage appliances might have multiple local power modules, distributed storage systems have similar power redundancy plus rack-level power diversity. This further dilutes risk and, when the distributed storage system uses this diversity to refine data placement, will result in data storage resilience to many levels of power failure.

SREs responsible for distributed storage systems need to pay attention to different metrics than they do for a single network-attached storage device. For example, they will monitor the computed recoverability of discrete chunks of data. This involves understanding the system's implementation: how does the storage system lay out the data, and where does it replicate the constituent data parts? How often does the system need to recopy data to maintain *risk diversity*, an indicator of how accurately it will be able to retrieve data? How often does the system metadata have a cache miss, causing longer data-retrieval times?

As distributed storage systems enable applications used around the globe and with massive quantities of data, they present observability opportunities for SRE. The rewards of these systems include more durable and available storage.

Copyrighted material

The Role of Cardinality

Charity Majors and Liz Fong-Jones

Honeycomb.io

In the context of databases, cardinality refers to the uniqueness of data values contained in a set. Low cardinality means that a column has a lot of duplicate values in its set. High cardinality means that the column contains a large percentage of completely unique values. A column containing a single value will always be the lowest possible cardinality. A column containing unique IDs will always be the highest possible cardinality.

For example, if you had a collection of a hundred million user records, you can assume that userID numbers will have the highest possible cardinality. First name and last name will be high cardinality, though lower than userID because some names repeat. A field like gender would be fairly low cardinality, given the nonbinary but finite choices it could have. A field like species would be the lowest possible cardinality, presuming all of your users are humans.

Cardinality matters for observability, because high-cardinality information is the most useful data for debugging or understanding a system. Consider the usefulness of sorting by fields like user IDs, shopping cart IDs, request IDs, or myriad other IDs such as instances, container, build number, spans, and so forth. Being able to query against unique IDs is the best way to pinpoint individual needles in any given haystack.

Unfortunately, metrics-based tooling systems can only deal with low-cardinality dimensions at any reasonable scale. Even if you only have merely hundreds of hosts to compare, with metrics-based systems, you can't use hostname as an identifying tag without hitting the limits of your cardinality key space. These inherent limitations place unintended restrictions on the ways that data can be interrogated. When debugging with metrics, for every question you may want to ask of your data, you have to decide—in advance, before a bug occurs—what you need to inquire about so that its value can be recorded when that metric is written.

That inherent limitation has two big implications. First, if during the course of investigation you decide that an additional question must be asked to discover the source of a potential problem, that cannot be done after the fact. You must first set up the metrics that might answer that question and wait for the problem to happen again. Second, because it requires another set of metrics to answer that additional question, most metrics-based tooling vendors will charge you for recording that data. Your cost

Copyrighted material

increases linearly with every new way you decide to interrogate your data to find hidden issues you could not have possibly predicted in advance.

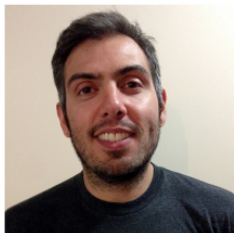
Conversely, observability tools encourage developers to gather rich telemetry for every possible event that could occur, passing along the full context of any given request and storing it for possible use at some point down the line. Observability tools are specifically designed to query against high cardinality data. What that means for debugging is that you can interrogate your event data in any number of arbitrary ways. You can ask new questions that you did not need to predict in advance and find answers to those questions, or clues that will lead you to ask the next question. You repeat that pattern again and again, until you find the needle that you're looking for in the proverbial haystack.

Adapted from the upcoming book Observability Engineering, expected in 2021 from O'Reilly.

Copyrighted material

Security Is like an Onion

Lucas Fontes



Auth0

Your company is living the dream. You've found product–market fit, sales are growing, and the idea of an IPO or acquisition steadily inches closer to reality. One day, the leadership team brings in external help to navigate the IPO process, and the conversation goes like this:

> *Consultant: Everything is looking great! So tell us, how's your security story?*
>
> *Leadership: Well, we haven't been hacked so I would say it is pretty good!*
>
> *Consultant: How do you know you haven't been hacked? What is your exposure?*
>
> *Leadership: (stares into the abyss) I will get back to you on that.*

As an SRE, one of your goals is to guide security controls and confidently answer questions related to risk management; but where should you start? I like the NIST's CyberSecurity framework of Identify, Protect, Detect, Respond, and Recover. Use it as is or as a foundation for your own security journey.

Identify what is crucial to business continuity in terms of systems, data, and assets. Once identified, evaluate the risk associated with each concern and any changes required to achieve the desired state by asking questions such as: What is preventing someone from interacting with our servers at our colocation data center? How do we deal with misplaced laptops or phones?

To get started here, you'll want to familiarize yourself with device encryption and basic mobile device management (MDM), because it can improve your security without jeopardizing usability.

Unpleasant cybersecurity events are a fact of life. The *protect* function is about limiting or containing the impact when one occurs. The keys are training, continuity, and supply chain management. Ensure that everyone goes through training related to identity management, privileged data manipulation, and remote access. Document and exercise controls for business continuity and disaster recovery. Finally, implement protective measures for the code supply chain, such as code scanning and use of third-party licenses.

A good *detection* system should have layers, raising an alarm each time one layer fails. The most important property of a detection system is its mean time to detection, which dictates how quickly you can react to a cybersecurity incident. The goal is for

Copyrighted material

Use Your Words

Tanya Reilly

Squarespace

When it comes to reliability, we're used to discussing new advances in the field, but one of the most powerful forces for reliability is also one of the oldest: the ancient art of writing things down. A culture of documenting our ideas helps us design, build, and maintain reliable systems. It lets us uncover misunderstandings before they lead to mistakes, and it can take critical minutes off outage resolution.

Code is a precise form of communication. A pull-request reviewer can mentally step through a change and evaluate exactly what it does. What they can't say, though, is whether it *should* do that thing. That's why thorough *PR descriptions* are so important. To evaluate whether a change is really safe, a reviewer needs to understand what the code author is trying to achieve. Our words need to be precise too.

Words give us a shared reality. They force us to be honest with ourselves. A system design that felt quite reasonable during whiteboard discussions might have glaring holes once the author is confronted with describing an actual migration or deployment plan or admitting their security strategy is "hope nobody notices us." An *RFC* or *design document* spells out our assumptions. They let us read each other's minds.

A culture of writing things down reduces ambiguity and helps us make better decisions. For example, an availability SLO of 99.9% only tells you anything if you know what the service's owners consider "available" to mean. If there's an accompanying *SLO definition document* that explains that a one-second response is considered a success, and you were hoping for 10-millisecond latencies, you'll reevaluate whether this back end is the one for you.

Once decisions are made, *lightweight architectural decision records* leave a trail to explain the context in which the decision was made, what trade-offs the team considered, and why they chose the path they did. Without these records, future maintainers of systems may be confronted with a Chesterton's gate: a mysterious component that seems unnecessary but that could be critical to reliability.

Writing shortens incidents too. During an outage, written *playbooks*—documentation optimized for reading by a stressed-out person who was just paged—can remind an on-caller how a system works, where its code lives, what it depends on, and who should be contacted, saving brain cycles and valuable minutes for debugging.

For long incidents, *incident-state documents* can record who's involved, which avenues

Copyrighted material

Index

A

a student should be able to (ASSBAT), Jennifer Petoff-Jennifer Petoff

abstractions, Murali Suriar

accidental complexity, Laura Nolan

ACKs, Julia Evans-Julia Evans

allostatic load, Kurt Andersen

Apache Traffic Server (ATS), Effie Mouzeli-Effie Mouzeli

API, Salim Virji, Michelle Brush, Arshia Mufti

application layer, Effie Mouzeli

application programming interface (see API)

architectural analysis, Blake Bisset-Blake Bisset

ASSBAT (a student should be able to), Jennifer Petoff-Jennifer Petoff

ATS (Apache Traffic Server), Effie Mouzeli-Effie Mouzeli

audits, Joan O'Callaghan

automation software, Salim Virji, Michelle Brush-Michelle Brush, Daniella Niyonkuru-Daniella Niyonkuru

autoscaling, Justin Li

availability, Jacob Scott, Heidy Khlaaf-Heidy Khlaaf

B

bandwidth, human, Denise Yu

black swan event, Alex Hidalgo, Blake Bisset

Blank-Edelman, David N., Björn "Beorn" Rabenstein

blogs, Anita Clarke-Anita Clarke

brag document, Julia Evans and Karla Burnett-Julia Evans and Karla Burnett

Brooks, Fred, John Looney, Laura Nolan

bugs, Julia Evans, Michelle Brush, Jake Pittis, Hillel Wayne

    (see also debugging)

Copyrighted material

bulkheads, Justin Li

Burgess, Mark, J. Paul Reed

burnout, Denise Yu, John Looney, Joan O’Callaghan, Daniella Niyonkuru-Daniella Niyonkuru, Kurt Andersen, Spike Lindsey, Caitie McCaffrey, Lorin Hochstein

C

caching, Justin Li-Justin Li

caching layer, Effie Mouzeli-Effie Mouzeli

capacity, Joan O’Callaghan

cardinality, Charity Majors and Liz Fong-Jones-Charity Majors and Liz Fong-Jones

cascading failures, Rita Lu-Rita Lu

CDN (content delivery network), Effie Mouzeli

Challenger Space Shuttle, Matthew Huxtable

changes, Joan O’Callaghan-Joan O’Callaghan, Vanessa Yiu-Vanessa Yiu, Johnny Boursiquot, Johnny Boursiquot, Arshia Mufti-Arshia Mufti

(see also cultural changes)

chatbot, Daniella Niyonkuru

(see also ChatOps)

ChatOps, Daniella Niyonkuru-Daniella Niyonkuru

Chesterton's gale, Tanya Reilly-Tanya Reilly

CI/CD (continuous integration/continuous delivery), Ingrid Epure

circuit breakers, Justin Li

client satisfaction, Matthew Huxtable

cloud-native technologies, Björn “Beorn” Rabenstein

code, Tanya Reilly-Tanya Reilly

code compliance, Heidy Khlaaf-Heidy Khlaaf

(see also cyclomatic complexity, path complexity)

code ownership, Jacob Scott

CodeSonar, Heidy Khlaaf

collaboration, Anita Clarke, Lorin Hochstein, Spike Lindsey, Avleen Vig, Nicole Forsgren, Daria Barteneva, Johnny Boursiquot

Common Vulnerabilities and Exposures (CVEs), Felix Glaser

Copyrighted material

communication, Avleen Vig, Nicole Forsgren, Daria Barteneva

Community of Practice, Nicole Forsgren

complex system, Laura Nolan

(see also complexity)

complexity, Jake Pittis-Jake Pittis, Laura Nolan-Laura Nolan, Niall Murphy

(see also accidental complexity, essential complexity)

content delivery network (CDN), Effie Mouzeli

context switching, Bouke van der Bijl, Lorin Hochstein

continuous integration/continuous delivery (CI/CD), Ingrid Epure

control plane, Salim Virji

Conway's law, Effie Mouzeli, Avleen Vig

Conway, Melvin, Avleen Vig

creativity, Kurt Andersen-Kurt Andersen

crisis, Laura Nolan-Lorin Hochstein

crisis-complacency cycle, Laura Nolan

critical user interactions, Kristine Chen and Bart Ponurkiewicz

cultural changes, Vinessa Wan, Vanessa Yiu-Vanessa Yiu

culture, Todd Palino, Suhail Patel, Miles Bryant, and Chris Evans, Spike Lindsey, Anita Clarke

(see also hero culture, team culture)

customer acquisition, Niall Murphy

customer experience, Narayan Desai

customer satisfaction, Tamara Miner, Vanessa Yiu, Brian Murphy

customer service, John Looney, Dawn Parzych

CVEs (Common Vulnerabilities and Exposures), Felix Glaser

cybersecurity, Lucas Fontes-Lucas Fontes

cyclomatic complexity, Heidy Khlaaf

D

data layers, Salim Virji

data-driven analysis, Blake Bisset-Blake Bisset

Copyrighted material

Datadog, Daniella Niyonkuru

DDoS (distributed denial of service), Felix Glaser

debuggable design (see formal specification)

debugger, Charity Majors and Liz Fong-Jones

debugging, Charity Majors and Liz Fong-Jones-Charity Majors and Liz Fong-Jones, Charity Majors and Liz Fong-Jones-Charity Majors and Liz Fong-Jones, Tanya Reilly, Avishai Ish-Shalom and Nati Cohen-Avishai Ish-Shalom and Nati Cohen, Ingrid Epure

DeGrandis, Dominica, Kurt Andersen

dependency, Bouke van der Bijl, Kristine Chen and Bart Ponurkiewicz

deployment cadence, Michelle Brush-Michelle Brush

design documents, Tanya Reilly

device encryption, Lucas Fontes

DevOps culture, Vinessa Wan

diagrams, Murali Suriar-Murali Suriar

Dijkstra, Edsger, Niall Murphy

disaster plans, Tanya Reilly-Tanya Reilly

disaster recovery (DR), Tanya Reilly

distributed denial of service (DDoS) , Felix Glaser

distributed storage systems, Salim Virji-Salim Virji

distributed teams, Avleen Vig

DNS, Effie Mouzeli

documentation, Tanya Reilly-Tanya Reilly, Joan O'Callaghan, Ashley Poole, Avishai Ish-Shalom and Nati Cohen, Julia Evans and Karla Burnett-Julia Evans and Karla Burnett, Spike Lindsey, Daria Barteneva and Eva Parish-Daria Barteneva and Eva Parish

(see also playbooks, runbooks)

DR (disaster recovery), Tanya Reilly

drain script, John Looney

E

Eisenhower matrix, Laura Nolan

Eisenhower, Dwight D., Laura Nolan, Laura Nolan

embedded model, Johnny Boursiquot

Copyrighted material