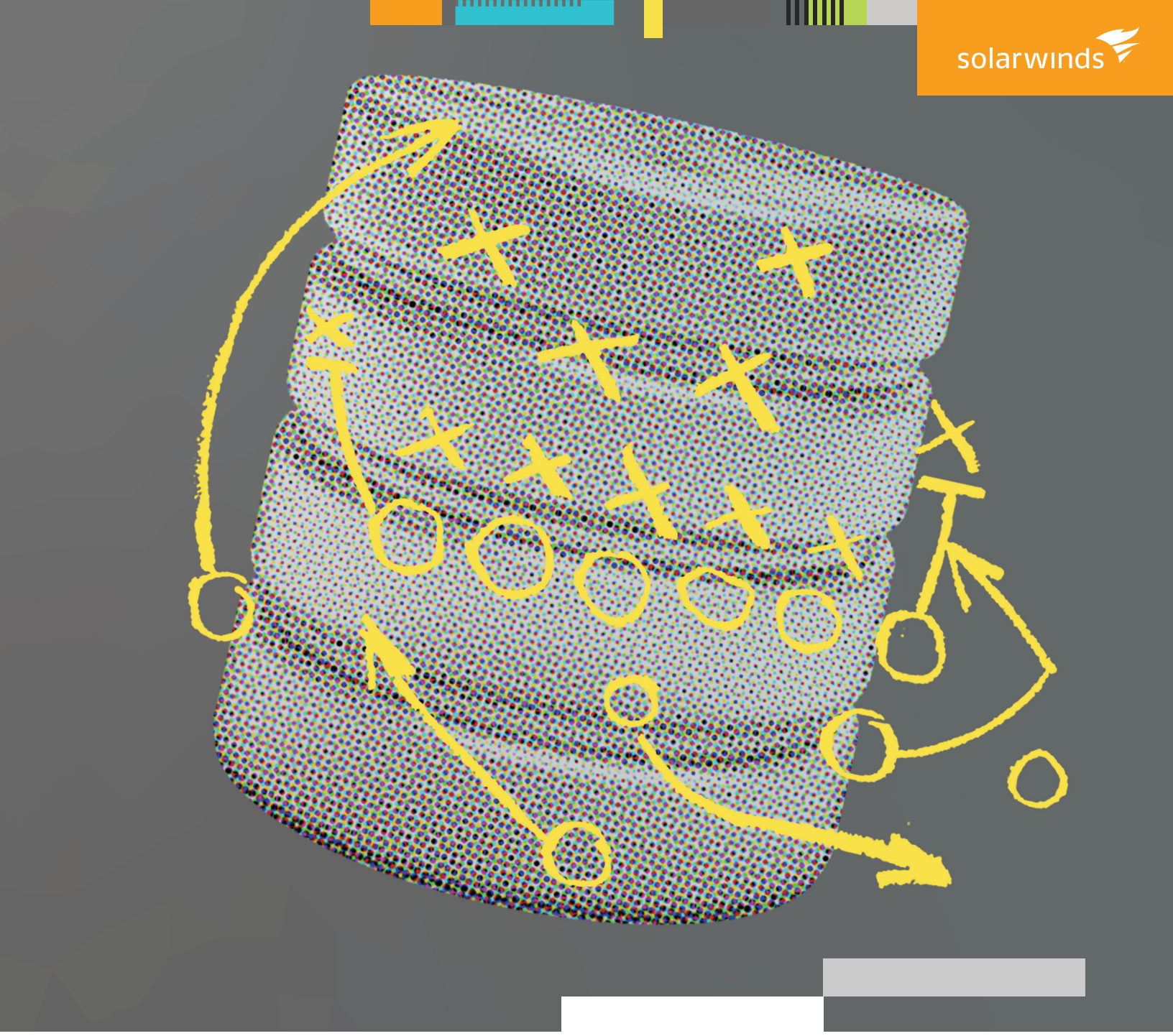
solarwinds



eBOOK

DevOps for the Database

By Baron Schwartz

Table of Contents

Meet the Author

Baron Schwartz

Baron, the founder of VividCortex, is a performance and scalability expert who participates in various database, open-source, and distributed systems communities. He has helped build and scale many large, high-traffic services for Fortune 1000 clients. He has written several books, including O'Reilly's best-selling High Performance MySQL. Baron has a Computer Science degree from the University of Virginia.

Introduction

When it comes to the database, some teams innovate faster, break things less, make more money, and have happier humans. I tried to discover what distinguishes the high and low performers. Then, I struggled to find words for it. Eventually, I realized it's DevOps. I'd been staring DevOps in the face without knowing it.

Most companies don't include their databases in their DevOps practices, which is why I wrote this book. I believe applying DevOps culture and processes is just as helpful for the database as it is for the rest of the software development and delivery pipeline. Without DevOps, I've seen the database and/or the DBA become a critical bottleneck. With DevOps, I've seen teams enjoy all the benefits DevOps brings to other parts of their systems and processes: faster, better, and cheaper, pick all three.

This book isn't just about companies adopting DevOps as a general concept, but specifically about applying it to their databases. Even among DevOps adopters, the database tends to be a stronghold of pre-DevOps culture and processes. Thus, it becomes what I contend is often the biggest bottleneck and the biggest opportunity in many software teams. So, this book is about DevOps for the database, and why it's different, hard, valuable… and doable!

I also wrote this book because we don't all think about DevOps the same way, especially when it comes to the database. There's a spectrum of opinions on what DevOps means, but I think two viewpoints are especially prominent. To oversimplify: one thinks DevOps is automating database operations; the other thinks DevOps is developers owning and operating their applications' databases. From what I've seen, many teams don't progress beyond automation, and I hope to encourage those folks to explore further. In both theory and practice, the benefits of taking the next steps are enormous.

Finally, I wrote this book because I'm a seeker myself. It's about progress, not perfection. I've never personally attained the level of DevOps skill and sophistication I've seen others develop. I wrote to clarify my own thinking, and to reflect some of the light others have cast for me. I wrote to advocate these ideas to the teams I work with—as you'll see, at SolarWinds we don't do everything I'm describing in this book. And I wrote to help our customers learn from each other. I want each of them to be able to study and learn from others, and in so doing, I hope they'll be able to apply some of that learning, and benefit from it.

You can adopt or improve DevOps practices for your databases. So can I, and so can we all. And I wrote this book to help the best way I know how. I hope it's useful.

Who This Book is for

I wrote this book with a broad, mostly technical audience in mind:

» Modern software developers or engineers.

» DBAs, SREs, and technical operations staff.

» Engineering, operations, and product roles from managers, directors, and VPs, all the way up to and including CTOs.

If you're responsible for all or part of building and running a data-intensive application, and you want to modernize and automate as much as you can for speed, stability, and efficiency—and you'd like to make progress in this regard for the database—then I wrote this book for you.

Database DevOps Stories

During the past 15 years or so, I've seen a spectrum of successes with managing the complex processes and interactions around the database. I was a developer for a while, then a DBA, then a consultant, and now in leadership, but I continue to be very curious and have lots of conversations with people who are building and running systems. Most of the pain I've personally experienced, or seen others experience, has come from change. Specifically, change fast enough to stress the teams' and databases' abilities to respond and cope: hyper-growth teams, fast-growing customer data and workload, and rapidly changing software.

Several stories from these years stand out to me as archetypes of what happens, or doesn't happen, when teams have the opportunity and need to apply DevOps practices to their databases. First, a couple of cases where things didn't go so well:

- **Firefighting**. One of the teams I worked with was growing their customer base so quickly they couldn't keep up with the demands on the database. Management's response was to grow their team of database administrators (DBAs) while, in their words, "throwing hardware at the problems" of capacity, performance, and availability. This brute-force approach resulted in a strongly siloed team of DBAs with a clearly defined role. The silos were reinforced by management mandates and incentives, so the DBA team viewed their domain and specialty as their job security. At the same time, the development teams were pushed away from the database, widening the gulf and creating more incentives for database ownership to remain exclusively the province of the DBAs. The last time I spoke with them, they were still in constant firefighting mode. From my admittedly limited outsider's perspective, it looked like they were working *in* the system, rather than working on the system.

- **A Valiant Try**. A friend of mine was the sole DBA at one of the fastest-growing startups in Silicon Valley history. "The development team is growing from 20 to 100 engineers in the next couple of months," he told me. "But the databases are on fire and there's no headcount allocated for another DBA. All these new developers are being pressured to ship software to production their first day on the job, to prove they've got what it takes to be part of the team." My friend's situation was, in my opinion, a classic case of a misalignment between responsibility and authority: he was supposed to be responsible for keeping the databases up and fast, but he lacked authority over the queries coming from the new developers' code. How could he solve this? He was thinking about requiring that code couldn't be deployed without his approval. He'd have to review all new releases, inspecting them for query and schema changes. But after he thought about it more, he decided to divest responsibility instead of taking on more: he insisted the developers should own the production performance of their own application's database queries. I think it was a good decision, but management overruled him, so he left. Not coincidentally, I believe, the company ended up outgrowing its ability to execute and serve its fast-growing customer base. It suffered a lot of problems like high turnover and low service reliability, and the investors sold it to try to cut their losses. Today the company still exists as a subsidiary of another. According to their DBAs, they have a very rigid, siloed, slow-moving legacy DBA mindset and culture, and because of this, they can't iterate rapidly on their product or infrastructure.

Those stories didn't have happy endings, but I've seen many successes too. In one case, a couple of people in technical operations in a media company purchased the Database Performance Monitor (formerly VividCortex) monitoring product to gain visibility into their database's behavior and performance.[1] We expected the two ops engineers to be our primary users, but we were surprised to see instead they disengaged and nearly two dozen others became daily active users. As it turned out, this was their intent: they didn't want to be our primary users. They encouraged their software engineers to monitor the databases themselves and be responsible for production database performance. And it worked! During the several years I worked with them, I was consistently impressed at how much the team and company outperformed. They ran a larger, higher-traffic, more complicated service than similar companies who had hundreds of engineers, they did it with lower cost and less hardware, and they shipped and evolved their product surprisingly fast.

[1]This story isn't about SolarWinds Database Performance Monitor, but it doesn't make sense if I tell it without mentioning our product, as you'll see.

I've seen similar things happen at many other companies. When I first got to know one fast-growing company in the gaming industry years ago, they had a traditional DBA team of two people. With the backing and mandate of their CTO and managers, these DBAs extricated themselves from daily operations tasks, went all-in on the cloud, and enabled each service delivery team to self-service. The resulting speed and stability improvements were dramatic. The DBAs eventually moved on and weren't replaced. Today the company has dozens of service delivery teams and zero DBAs, and their former DBAs are running critical infrastructure platform teams at other companies. In another case, I saw a single DBA in a situation similar to the "valiant try" story above. She was straining under the load, she divested herself of traditional DBA responsibilities, and she "automated herself out of a job" along with giving the engineering teams tools to self-service. Engineering was enabled to innovate the services much more quickly, while she focused on innovating the data tier into a platform to support the company's growth. Years later, she still has DBA in her title, the redefined DBA team has grown significantly under her leadership, and the company continues to grow and transform rapidly.

As I continued to witness experiences like these, I became increasingly curious about ***why***. Why did some companies remain mired in the mud of the way they always did things, overwhelmed by the sheer toil of database operations? Why did some try to adopt practices they'd witnessed their friends doing, but stop short of success or fall back to old behaviors? Why did others succeed in changing their culture, process, and ability to execute with speed and stability?

I've spent a lot of time thinking, reading, writing, and talking about this with many people over the last several years. I described it many times and got many nods of understanding and agreement in response. At first, I didn't call it DevOps, but I realized later it's what I'd been describing—it just took me a while to see. ***The outperforming companies are applying DevOps principles and practices to their databases***. More specifically, I believe some DevOps approaches are more impactful than others, and the high performers have adopted higher-value practices others have overlooked or rejected.

What Is DevOps?

If DevOps is the answer, then what exactly is DevOps for the database? What are these high-performing companies and teams doing that others aren't? To answer this, first I'll try to answer, "What is DevOps?"

DevOps communities are hesitant to define DevOps very explicitly. Instead, we usually talk about what it isn't, or describe it in inclusive terms that can be too non-specific for newcomers to get a clear idea what we're talking about. There are good reasons for this reluctance, and although I respect the reasons, to make this book as useful as possible, I need to give my own definition of DevOps, at least to some degree of clarity.

DevOps is variously described as a culture, practice, process, mindset, or situation in which software developers and IT operations either unify or work closely together across the entire software development lifecycle, at a minimum from development through testing, deployment, and production operations. This collaborative and/or shared approach contrasts with siloed roles or workflows, where each group "throws it over the wall" to the next group when they're done with "their part" of the work.

Historically, DevOps took root most quickly and strongly in IT operations, or what we in the DevOps community often call "technical operations."[2] This is what Charity Majors (@mipsytipsy) calls "the first age of DevOps." The first age consisted of ops engineers automating their work with tools like Chef and Puppet, as opposed to performing old-school manual systems administration by typing commands into terminals. What Charity calls the second age of DevOps is arriving quickly now: developers are responsible and involved in a lot of operational tasks, such as provisioning infrastructure, deploying their own applications, and monitoring how applications perform in production.

Whatever DevOps is, it obviously involves a lot of complex moving parts. When you try to unpack it, you begin to see a variety of codependent factors such as who knows what, who does what, who has which skills, and so on. And there are many different opinions:

[2]This distinction clarifies the relatively recent separation between technical operations, who runs the systems that support customer-facing applications; and IT, which historically referred to those who handle internal technology services like phones, printing, networking, and desktop support.

- Some say DevOps isn't a job title, while others see no problem with hiring a DevOps Engineer or having a DevOps team.
- Some say development and ops are *roles*, while others say those are *skills* anyone can have, and yet others don't object to either usage.
- Some believe DevOps is primarily about automation, while some believe it's primarily about culture and collaboration.
- Some say DevOps is about *working together across silos*, some say DevOps is about *removing silos*. In other words, some say it's about specialized developers and operations folks collaborating without expanding the scope of their individual roles, and some say it's about combining development and operations into a single team of generalists, where nobody is a specialist and everyone's role encompasses both dev and ops. (I've seen a lot of disagreement on this point.)

In my experience, there are many different "right" ways to do it—it's very context-specific. This is one of the reasons although individuals can often hold strong opinions, the DevOps community isn't prescriptive—or even overly descriptive—about definitions of DevOps.

There *is* broad agreement, though, about which general categories of technologies and practices—which *capabilities*—are important. For example, most people agree automation, infrastructure-as-code, continuous delivery, and monitoring are important parts of most companies' DevOps practices, among others. The book *Accelerate: Building and Scaling High Performing Technology Organizations* (Forsgren et al, 2018) lists 24 key capabilities (on page 201) their research shows contribute to high performance. This is, in my opinion, the best list currently in existence.

While writing this book, I was reminded not only are definitions of DevOps quite flexible, but *we don't always realize how much we disagree on them*. Many blog posts, surveys, podcasts, books, and so on use DevOps in quite specific ways, without stating the assumptions explicitly. I have to believe I'm subject to similar unconscious biases as everyone else. That's why I'm trying to be as explicit as I can about my viewpoint, so you don't have to try to reverse-engineer my concept of DevOps by reading between the lines. However, my attempts at clarity are just that—they're not intended to say anyone's belief is wrong or right.

Database DevOps Capabilities

What about the database? What is DevOps for the database? Just as with the big-picture definition of DevOps, there are a lot of different right ways to do it, it's context-specific, but I have my viewpoint based on what I've seen work well and not-so-well.

In this section I'll do two dangerous things: I'll introduce a list of capabilities I consider important for DevOps in the database; and I'll lay them out in a rough progression you could be tempted to use as a kind of maturity model. Why is my list, and my thematic progression, dangerous?

- The list is dangerous because it's my opinions formed anecdotally. If you want more reliable guidance, you need to rely on science, and there's no research, data, theory, or science here. The only place I think you can find reliable DevOps science is in the book *Accelerate* (and other works by the same team). Nonetheless, I think it's valuable to advance this list as a hypothesis because if I'm even partially right, the benefits are worthwhile, and my experience should count for something even though it's not science.
- The thematic progression is dangerous because it smells like a maturity model, and those are problematic, as the *Accelerate* authors detail on pages 6 – 8. In brief, maturity models fool you into seeing DevOps as a destination instead of a journey; they present the illusion of linear progress through atomic and well-bounded stages with invariant definitions in different contexts; they encourage vanity metrics instead of being outcome-focused; and they're static instead of being dynamically defined in terms of an evolving understanding. However, I'm presenting a way to organize my list of DevOps practices because I believe there are benefits to doing so, and many people have requested my opinion about where to start and how to make progress.

Caveats aside, here's how I articulate my current understanding of what separates teams who do DevOps for the database exceptionally well. I view the following capabilities as important:

1. Automated database provisioning and configuration management (infra-as-code).
2. Automated backups, recovery (restore), and continual, automatic backup testing.
3. Schema-as-code in version control, with "normal" change and review processes.
4. Migrations-as-code with version control, for both schema and data migrations.

5. Continuous delivery of database schema migrations and data migrations.
6. Forwards and backwards application/schema version compatibility (decoupling).
7. Holistic and detailed (workload- and internals-focused) database monitoring.
8. Developers have immediate visibility into queries in production.
9. Developers own the full app lifecycle, including production query performance.
10. Developers own or participate in database performance incident response.
11. Database-specific skill, knowledge, and processes are spread around the team.
12. DBAs are shielded from most incidents caused by applications or customers.
13. DBAs are focused on proactive, strategic architectural/platform/product initiatives.

I'll explore those capabilities in varying levels of detail throughout the rest of the book, but when I've discussed these in the past with people, a common request is "where do I start? What comes first?"

It's a perfectly valid question. Of course, the only universally applicable answer is "it depends," but I think there are more-or-less-applicable patterns to help people make decisions and start making progress.

Although I understand the reasons maturity models are dangerous, I have also seen the benefits of using them as one of several inputs into a planning process that uses a target state description and current-state assessment, produces a gap analysis, and breaks down the gap into a plan of action people can follow. It's concrete and it works. So, for those who want it, I'll share my opinion: I've seen teams fall along a spectrum of progression in sophistication and capability, in what I'd anecdotally group into categories. I think the following six categories tend to describe those:

1. Baseline DevOps infrastructure operations practices are applied to the database: automated provisioning, automated configuration, monitoring, and so forth. This starting point is a natural extension of the "first age of DevOps" practices, where systems administrators automated their work. Some people consider this merely DBA fundamentals, a kind of

pre-DevOps level of basic competency that doesn't qualify as DevOps, but many other people feel differently. I think this one might be especially vertical-specific; I've been told older, highly regulated industries like telecoms, insurance, and banking often are administering databases manually, and have no automation, reproducibility, or tooling.

2. Database-specific operational tasks such as backups, recovery, upgrades, and refreshing the test dataset on pre-production databases are automated. This is an extension of the first point into tasks that were historically DBA-specific but don't need to be: if a system administrator can automate their work, then so can a database administrator. Similar to item 1, however, some consider this pre-DevOps and others think it's a significant portion of what DevOps entails.

3. The database schema, code, and data models—table structures, stored procedure code, and so on—are managed similarly to application source code. This code is in version control, and "normal" change control processes are in place, whatever that means for you. For example, if pull requests, reviews, and approvals are normal in deploying application code, then ideally changes to the database are treated the same way.

4. Deployments of database changes such as schema changes (migrations) and data transformations (data migrations) are automated with tooling, not performed manually. At a more advanced level, these migrations are part of the same deployment tooling and processes that deploy applications, so there's a single path to production deployment for both application and database changes. See the section Achieving Continuous Delivery With Databases for more details on this.

5. Performance and availability aren't just DBA jobs. Developers are involved in, and ideally responsible and on-call for, the production database performance of the applications they build. At a more advanced level, I've seen these same developers own performance and availability *fully*, diagnosing and repairing outages in the databases their applications use—it's no longer primarily a DBA job at all, though DBAs are usually a great resource for the harder-to-solve problems. This is the database-specific flavor of the "second age" of DevOps: extended from code and infrastructure, to include the database within its scope of responsibilities.

6. Database administrators aren't doing repetitive tasks; they're subject matter experts consulting to, or embedded within, software engineering teams to multiply the engineers' leverage. They focus on proactive, strategic, high-value work such as architecture, performance, scalability, and capacity management. At a more advanced level, databases are sometimes provided as a service within the company, operated as a self-service platform with development teams as internal customers, and managed according to Database Reliability Engineering guidelines similar to SRE practices from Google. (I usually see this in companies that aren't adopting cloud computing and are building their own.)

I think those points are roughly in order of ease to tackle for most organizations, but I'd also re-emphasize this isn't a strictly linear model of distinct and atomic stages of development with a well-defined terminal state of maximal achievement. Organizations can be in several of these categories, and I'm unaware of any "done" organization.

Some of the points listed above are easy to achieve. In particular, the first two require little more expertise or insight than recognizing they're beneficial. It's no mystery how to accomplish them. Lots of good reference material is available for many of them, in particular the book *Database Reliability Engineering*. This book will focus mostly on points 3 – 6, with emphasis on things that might seem daunting or impossible unless you see how others do them successfully.

Additionally, there's not an exact mapping of capabilities (the first list) to the six categories. I think some of the capabilities, such as production query visibility (monitoring) are important for all the categories: they're cross-cutting concerns.

Again, even during the draft stages of writing this book, some people viewed parts of the above six categories as aspirational and hard to achieve, and others thought those same items were non-negotiable baseline requirements. I'm reminded of one of John Willis's major themes in the 2010 article *What DevOps Means to Me*: DevOps is not a judgment. In a sense, DevOps is about doing what's feasible and valuable in your situation, valuing progress over attainment.

What Kinds of Companies Can Apply DevOps to the Database?

I've heard a lot of reactions to the practices and progressions I mentioned in the previous section. People frequently have situation-specific objections to some or all these points. The reasons it's harder for some people to envision applying some of these practices to their own databases range from cost, to security, to regulatory, to risk, to resources, to team skill level, and many more.

In my experience, most or all of these things are possible for nearly every company to do and do well. I've seen it myself in a variety of industries, companies, products, technologies, and so on. In my opinion, there isn't much industry-specific, database-specific, or technology-specific about most of it. This isn't just my opinion; research corroborates it. *Accelerate*, in particular, agrees (page 221).

This isn't to say the objections are excuses: far from it. There are sometimes legitimate reasons why a practice has to be done more skillfully, or more carefully, or with more effort or cost. But most often, the biggest difficulty I've observed has to do with "the way it's always been done." That is, the hardest part of change is the change, not the desired end state after the change. I discuss this in more detail in the sections on Realigning Software Development Teams and The Importance of the Second Age of DevOps.

What do the successful companies and teams have in common, then, if they span all industries, all verticals, all technology stacks, and so on? I believe those who succeed in applying DevOps practices to their databases share a mindset. They have a vision and a culture[3] valuing the following attributes:

- Customer experience and application performance are valued because they're competitive advantages for customers, employees, and investors; and because they add directly to the financial bottom line.
- Developer productivity is valued because it's the highest-leverage investment a technology-centric company can make.

[3] This is my personal articulation of what I've seen work well, but the book *Accelerate* formalizes this by describing it in terms of the Westrum organizational structure.

- Engineering speed[4] and operational stability are valued because they're a virtuous cycle: speed begets stability, which begets speed. These companies understand that and optimize for speed and stability because the outcome is delivering customer value faster.
- DevOps principles such as transparency, empathy, and tight feedback loops are valued because they lead to higher performance, better aligned people, and processes.

The principles I mentioned not only aren't company-specific or industry-specific, but they're not even database-specific. The upshot is, *any company wanting to apply DevOps practices to the database can make progress*. If you can DevOps your code and infrastructure, you can DevOps the database too. You only need to want to, and you can find a way to *improve* what you're doing, even if *transform* seems hard right now.

[4] I'm not using a specific definition of speed here. It could be deployment lead time, engineering productivity, or any other measure that works for you. The overall point is faster progress tends to create higher stability. See the DORA report's definition of software delivery performance, which I return to in the section Why DevOps for the Database Succeeds and Fails.

Benefits of DevOps for the Database

The benefits of bringing DevOps to the database are the same as the benefits of a DevOps approach to any other part of the software development lifecycle. These are widely discussed, and at this point there's even credible research substantiating the claims.

The most prominent, and I believe the most rigorous, research supporting the claim of DevOps teams outperforming is from the DORA team: Dr. Nicole Forsgren, Jez Humble, and Gene Kim. Their book *Accelerate* and their 2018 State of DevOps Report sums up the results: "High performing DevOps teams that have transformed their software delivery processes can deploy code faster, more reliably, and with higher quality than their low-performing peers."

Notably, the research finds nothing database specific about that, and indeed the report affirms bringing DevOps practices to the database improves overall software delivery performance. Quoting from page 57, "Managing Database Changes,"

> *Database changes are often a major source of risk and delay when performing deployments… integrating database work into the software delivery process positively contributed to continuous delivery… There are a few practices that are predictive of performance outcomes. We discovered that good communication and comprehensive configuration management that includes the database matter. Teams that do well at continuous delivery store database changes as scripts in version control and manage these changes in the same way as production application changes. Furthermore, when changes to the application require database changes, these teams discuss them with the people responsible for the production database… When teams follow these practices, database changes don't slow them down, or cause problems when they perform code deployments*[5].

The authors of the State of DevOps Report have unpacked the science behind DevOps in much greater detail in their book *Accelerate: Building and Scaling High Performing Technology Organizations*. This book is such a treasure that I own several paper copies of it as well as the audiobook and Kindle versions of it. I regularly gift it, too. Get yourself a copy!

In *The Three Ways: The Principles Underpinning DevOps* by Gene Kim, you can see some of the underlying reasons why DevOps practices produce these types of outcomes, and why the practices themselves are beneficial and necessary. For example, "the outcomes of putting the First Way into practice include never passing a known defect to downstream work centers, never allowing local optimization to create global degradation, always seeking to increase flow, and always seeking to achieve profound understanding of the system (as per Deming)." Note the emphasis on increasing flow, important to the database, which otherwise becomes the bottleneck to deployment and improvement. And the emphasis on achieving profound understanding of the system, which requires observability, discussed in depth in the section Improving Database Observability.

Why Is it Hard to Apply DevOps to the Database?

Unfortunately, most companies' database management culture and processes lag behind the other areas where they've applied DevOps practices. According to a 2017 survey of 244 IT professionals by DBMaestro, only 37% of respondents have adopted continuous delivery for their databases. In a similarly focused 2018 survey of more than 1,000 people by Redgate, only 34% of respondents said the database is "part of the automated build and deploy process." In my own informal survey, unsurprisingly, even fewer people responded positively to a question about software developer ownership of database performance (see Figure 4).

DevOps adoption in the database is doable, but you'll surely encounter database-specific challenges. Many of these require database-specific fixes, too. If you don't know how to do it, this can feel daunting at first. Fortunately, as I'll share in sections such as Automating Database Migrations, there are many success stories, and they have enough in common to paint a clear pattern of success. Once you see how it's done, it's not too hard after all.

The most significant database-specific DevOps difficulties I've seen are as follows:

1. **Scar Tissue**. Some teams have seen so many database-related outages and problems they're afraid of the database's sharp edges. As a result of getting their fingers cut repeatedly, they're metaphorically scarred. This is a very real thing: just like a physical injury resulting in reduced mobility and strength, they're less able to perform well when it comes to the database. This is a cultural barrier, not a technical one. I put this item first in the list because I think it needs to be named, so it's easier to address. Once you overcome it, it's easier to make speedy progress on other items.

2. **Statefulness**. It's relatively easy to adopt modern, cloud-native, DevOps practices for disposable parts of the infrastructure, such as web servers and application instances. These are designed to be replaceable and stateless. But databases are stateful, which makes them a lot harder to treat as "cattle, not pets." There's not only a lot more to get right, but the stakes are higher—data loss is worse than service unavailability—and it's impossible to make it instantaneous. Data has inertia, so you can't just install a database and put it into production; you have to import data into it, which takes time. Likewise, you can't always roll back database changes instantly if a deploy fails, because if state was mutated, it takes time to mutate the state back again.

3. **RDBMS Paradigms**. Most mature, production-ready database technologies were designed in the days of manual system administration. They're built to be maintained by human operators and aren't automation friendly. They don't benefit from being designed in alignment with methodologies such as the twelve-factor app philosophy. It's hard to wrap automation and programmatic control

4. **Operations Tooling**. The most popular infrastructure-as-code and other operations tooling products weren't really built to be database-centric. You can use them with the database, but there are awkward edge cases you have to handle yourself. In particular, a declarative approach often has a bit of impedance mismatch with stateful services. These services are supposed to avoid data loss and may require exclusive access, synchronization points, significant amounts of time waiting for data imports, and the like. Declarative tooling, designed to "do whatever's needed to make this service's configuration match what's expected," can sometimes make unintelligent decisions when extreme delicacy is needed. It's possible to manage databases with such tooling, but you usually have to augment it yourself a little bit; there aren't complete, elegant, robust, built-in primitives specifically made for these types of tasks.

5. **Developer Tooling**. Developer tooling such as ORMs and migration tooling suffers from similar shortcomings when it comes to the database. It's often the case that it'll work fine on a laptop with a SQLite database, but won't work on a production service. The reasons are often related to scale and load, which is the next item in this list. And some open-source developer (and ops) tooling was developed inside of a big company, and it just doesn't extend well to general-purpose uses; it has too many dependencies and assumptions about how things were done at the company that produced it.

6. **Scale and Load**. A production database often slowly evolves to the point when it's running on a monstrously powerful server, handling a large dataset, and serving a high traffic workload. Attempting operations such as schema changes on large datasets under load can lead to a variety of problems and cause degradation or outages. Foremost among these are long-running operations, locking, and exhaustion of resources.

[5] This focuses on capabilities up to and including automation of schema changes in my "maturity model" categories, but not on full-lifecycle developer ownership of applications.

[6] Both the DBMaestro and Redgate surveys emphasize schema change automation as the core of DevOps. No endorsement of these surveys is given or implied by inclusion in this book.

Again, *all of these are solvable problems* with enough effort. After you identify the specific problems you have (or will likely have), specific solutions that will work well are easy to identify, even if not easy to apply (or apply fully). This is a good place to tout the contributions of an expert DBA, whose intimate knowledge of how your database works will be indispensable. I think this is a great way a DBA can make a difference, and we should celebrate our DBAs for their invaluable contributions in things like this.

For example, each database technology has different "sharp edges" that can cut us. MySQL lacks transactional schema changes and has brittle replication. PostgreSQL is susceptible to vacuum and bloat problems during long-running transactions. Nearly all databases have at least the possibility of a schema change being a locking operation that brings everything to a halt, and most databases are hard to observe and complicated to troubleshoot. These problems require approaches that figure out how to "thread the needle" between the database's limitations or characteristics, and your application and dataset's unique behavior and requirements.

I've collected specific examples from many companies to illustrate successful approaches to these and other problems. For more on these topics, see the sections titled Achieving Continuous Delivery With Databases and Automating Database Migrations.

Why DevOps for the Database Succeeds and Fails

I've had in-depth discussions with many teams about how they apply DevOps to their databases, and I've developed a theory about why it succeeds or fails in various circumstances. I think three core elements are required: people, tooling, and leadership (including management). In each of these categories, the right ingredients can lead to success, and the wrong ingredients can cause failure.

People. To modernize a legacy culture, you need the right people. This not only means you need the right combination of skills on your team, but you need people with the right interests and mindset, too. I think you need curious, driven people who are looking for personal growth and challenge, and see DevOps as an opportunity, not a threat. This is true for both DBAs and developers.

SendGrid's Silvia Botros wrote a great article on this topic: *DBAs, a Priesthood No More*. It's a clear picture of the difference between the legacy DBA role and mindset, and the modern necessity of a high-value, strategic DBA. The entire article is a great read, because Silvia herself is the archetype of the best I've seen in modern DBAs. A few excerpts:

> *In a traditional sense, the job of the DBA means she is the only person with access to the servers that host the data, the go-to person to create new database cluster for new features, the person to design new schemas, and the only person to contact when anything database related breaks in a production environment.*
>
> *I am here to argue that your job is not to perform and manage backups, create and manage databases, or optimize queries. You will do all these things in the span of your job but the primary goal is to make your business's data accessible and scalable. This is not just for the business to run the current product but also to build new features and provide value to customers.*

Silvia goes on to say the DBA role needs to be redefined as a source of knowledge and expertise for engineers, not as a "gofer" to perform database-related tasks. The modern DBA should accomplish this through a) infrastructure-as-code database configuration management; b) making database-related expertise readily accessible in runbooks referenced in pager alerts and *are designed for non-DBAs*; c) procuring and making available tools for *developers, not DBAs*, to gain immediate visibility into database state, health, and historical behavior; and d) investing proactively in ways to *shield the DBA from being interrupted by production alerts*, especially when they're symptoms of the business's success placing additional demand on infrastructure.

Personally, I think Silvia's emphasis—echoed by my italics above—is crucial to understand and internalize deeply. She focuses explicitly on getting out of interrupt-driven work and shifting database responsibilities to developers instead of DBAs. If the DBA is consumed by reactive firefighting work, then the database will become a serious software delivery bottleneck for the business.

This is why I think maturity stage 5, the second age of DevOps, is important. This is the main difference between the first two DevOps adoption stories I told previously, and the stories in which the company became a high performer.

Silvia's article also explains the difference between the traditional DBA's role as guardian and hindrance, and the modern DBA's role as one of your most valuable team members. Sometimes there's a perception of DBAs as second-class team members, keeping people away from the database, slowing things down and having to be bothered to get things done, and as passive—not actively driving progress. If that's the perception, then in my experience one or both of the following are likely contributing:

- The developers are resisting ownership of traditional DBA responsibilities.
- The DBAs are clinging to their familiar role from the past.

I think this clinging or aversion is itself a core problem and needs to be solved.

Some DBAs might fear if they automate themselves out of a job there'll be no need for them, but in my experience this fear isn't well-founded. The problem I've seen in the traditional DBA culture is they're not actually *using* those abilities. They're stuck in what SREs (Site Reliability Engineers) call toil. I think a great DBA's skill and knowledge will *always* be needed. DBAs should see DevOps as an opportunity, not a threat—as long as they're willing to change.

Sometimes, sadly, nothing changes in the team until you change the people in the team. Conway's Law is real, and dysfunctional human relationships create dysfunctional technical systems. In my personal experience, I could relate several stories of people I worked with, who prioritized firefighting serious recurring incidents—but wouldn't invest in fixing the underlying problems. Although the rest of the team worked hard to try to help them change, it eventually became clear they wouldn't or couldn't. At another time I saw two folks who couldn't get along, so they built two separate infrastructure-as-code repositories with opposing philosophies and gridlocked everything.

[7] Fear of reckless change causing outages and getting a DBA fired is valid. For more on this, see the sections on Achieving Continuous Delivery With Databases and Automating Database Migrations

Tooling. To successfully execute the change you're trying to undertake, you're asking people to do more things than they're currently doing. They're already fully busy, working as hard as they can, so the only way they can do more is by either deprioritizing something they're currently doing, or using tooling as a force-multiplier. In many cases the correct answer is to build or buy tools.

Netflix uses tooling extensively to support teams in changing their structure, workflow, and culture. For example, in *Full Cycle Developers at Netflix*, they wrote about the importance of tooling to support the desired shift in practices:

> Ownership of the full development life cycle adds significantly to what software developers are expected to do. Tooling that simplifies and automates common development needs helps to balance this out… Full cycle developers apply engineering discipline to all areas of the life cycle. They evaluate problems from a developer perspective and ask questions like "how can I automate what is needed to operate this system?" and "what self-service tool will enable my partners to answer their questions without needing me to be involved?" … Knowledge is necessary but not sufficient; easy-to-use tools for deployment pipelines (e.g., Spinnaker) and monitoring (e.g., Atlas) are also needed for effective full cycle ownership… Netflix's tools are often open source, and it may be compelling to try them as a first pass. However, other open source and SaaS solutions to these problems can meet most companies' needs.

Netflix has centralized teams dedicated to building their own tooling, but as they point out, it's a smarter decision for most companies to use open-source or SaaS, not build from scratch. If you read between the lines, I think it looks like they're cautioning that Netflix's own open-source tooling isn't always the best choice for most companies.

The exact tooling you'll need is going to be context specific, but I think it's safe to bet you need *at least* deployment, migration, and monitoring tools. And when you're bringing the DevOps approach to the database, you need *both* the right tools *and* a pragmatic approach to adopting them. Common tooling failures include:

- **Fragile Automation**. Trying to use generic, least-common-denominator tools for the database is one of the big reasons I've seen why people think DevOps for the database is too hard, as I discussed previously. General-purpose tools are usually not up to the job at hand because they trip over those sharp edges, and cause outages and "career interruptions."

- **Overly Ambitious Automation**. Tools trying to do too many things too fancily will set you up to fail. I prefer tools doing one thing really well. If a tool has dozens of features, most of them can be safely assumed to have been bolted on as afterthoughts, which means these features will have lots of risky edge case behaviors.
- **Incomplete Automation**. If you're still required to do manual toil, you're not gaining much benefit. Automation feels best when you don't have to step in and clean up leftover work the tooling couldn't do.
- **Too Many Tools**. The balance between overlaps and gaps in your tooling is tricky, because you need best-in-class tools for the database's specific needs, but you should have a toolkit, not a tool chest. I like fully managed SaaS products because you can extend them through APIs to customize them for your specific needs, and paying someone else to run them lets you focus on your core business. I'm a customer of SaaS offerings for nearly everything in the software development automation lifecycle, including source code hosting, issue tracking, testing, CI/CD, and monitoring. Buying fully managed subscription services, run by dedicated experts with a single focus, reduces cost and complexity quite a bit.
- **Tooling Burden**. If you have unrealistic expectations of the extra burden the tools will *add* to the jobs of those who use them, or you don't plan for this burden, it can cause problems. Tools require various kinds of cost and effort including maintenance (even if it's SaaS), training, time spent using the tool (hopefully offset by time gained by not doing what the tool does for you), and management or leadership coordination around things like agreeing upon processes (all tools can be used in multiple, possibly conflicting, ways). See also the classic paper "Ironies of Automation."
- **Culture Fit**. Not all tooling is equally compatible with the team's standard tooling or culture. For example, teams choose from solutions like Chef, Puppet, and Ansible because of factors such as the programming language, the users' preferred level of complexity, and so on. My first extensive Chef deployment, which I built with a consultant, was thrown away in favor of Ansible in a matter of weeks simply because the team knew and preferred Bash and not Ruby. Likewise, if the tooling is designed more for operations than developers, you'll get better ops adoption; if it's not documented, you might only get adoption among a small group of champions. Ideally you choose tools to help break silos, not reinforce them.
- **Leadership**. Changing a traditional DBA-driven, non-DevOps database environment into a high-performing DevOps team requires good leadership at all levels. Importantly, leadership doesn't come solely from higher in the org chart. Everyone involved will have an opportunity to lead.

The essential components of leadership I think everyone needs and ideally everyone contributes to, are as follows:

- **Vision**. It's difficult or impossible to align together around a shared goal unless it's vividly clear. A compelling vision of the desired destination is the simplest, most effective way to achieve this clarity. And you must align together around a shared goal, because DevOps is literally about collaborating and uniting. One goal for some people, and another for others, will not work very well. Work together to write down what you're trying to achieve. Make it as brief, simple, and clear as you can. It should sound like a "promised land" type of accomplishment. It's your North Star, your compelling what and why. Bonus: it can be helpful in hiring for long-term fit.
- **Objectives**. How will you know whether you're making progress? Your progress will be a journey, not a destination. What simple, clear objectives will you use to measure your efforts and achievements? I personally subscribe to the rule of three: as few objectives as possible, never more than three. And they should be impossible to cheat, so you can't achieve the objectives without achieving real progress towards the vision. This is simple, but hard! The 2018 State of DevOps Report contains a wealth of ideas you can use to measure outcomes that matter; for example, they quantify software delivery performance in terms of throughput (deployment frequency, lead time for changes) and stability (time to restore service, change fail rate). Pick the metrics you want and set an improvement target and date.
- **Planning**. A vision without a plan is just a dream. If you've articulated your target state clearly, then you can just assess where you are, perform a gap analysis between current state and envisioned state, and write down what needs to happen to close the gap. Now prioritize the first items, and you have a plan of action you can input into tickets and sprints.
- **Support**. You need genuine buy-in and commitment to make change happen. It doesn't have to be everyone—it's OK if some people take a wait-and-see attitude and you resolve to win them over—but you need to have at least some people who are personally invested in success. It's a huge help to have executive or management endorsement. Choosing compelling objectives—and avoiding exposing too much low-level detail to executives—can help a lot. Tying your efforts to company-level, publicly stated goals such as revenue or user growth, or aligning with big initiatives or other mandates, can help get that top-level buy-in. On the other hand, if you're trying to create progress against the active resistance of some people—for example, if developers and DBAs are siloed under different VPs, the DBAs want change but the developers fight it, and the VPs of engineering and tech ops are staying out of the fight—the chances for success are slim to none.

You also need vendors who support you with a) products, b) training and services, and c) clear stories of how others have trodden the path you're setting out to walk. Vendors can be a crucial part of your success.

- **Culture**. My favorite definition of culture is what a company incentivizes, rewards, punishes, or tolerates. If you're trying to make a change supported by people's incentives, you'll feel the wind in your sails. But if you're working against someone's promotion or paycheck, it'll be a stiff headwind. For more on this topic, please see the section Realigning Software Development Teams.

Leadership's role in all these things is to paint a clear picture of success, produce a plan of action, encourage people towards that direction, highlight the successes and learnings, and align or change incentives to make sure people are rewarded for their progress. Most of this can be done by literally anyone to some extent, from the grassroots level up.

Some of the leadership failures I've witnessed (and often done myself) include the following:

- Trying to buy DevOps. See the following section for more on this.
- Believing if the DBA's work is automated, you've "achieved DevOps."
- Continuing to incentivize the "old way" that you're trying to change away from or allowing people to cling to (or revert to) their comfortable, familiar old ways. Change requires new, unfamiliar things.
- Insisting on a single, idealistic One True Way. This creates such a narrow definition of success and progress that few teams can achieve it. This is another reason the definition of DevOps is so open and inclusive.
- Micromanaging. If you're micromanaging, you're not leading. If you feel the need to be hands-on, maybe you can do some work yourself instead of treating people like puppets. If you feel your boss is micromanaging you, ask yourself: are you failing to manage up effectively? I recommend the book *The Manager's Path* by Camille Fournier to both managers and their teams.
- Underinvesting in people, tooling, skills, or training. This is a management failure, but individual contributors also need to help their managers. Most managers are removed far enough from the actual work they have to make decisions based on weak signals, because it's all they can get. I've experienced this myself: people came to me for decisions long after I had lost track of how our own systems and processes worked. Tell your managers what you need—a clearly stated, well-justified position statement from someone with their hands in the actual systems is a boon to a manager. If you don't do this, the risk of failure is higher.

- A "bull in a china shop" heavy-handed Big DevOps Transformation Project. This all-or-nothing, burn-the-boats approach often ends in shattered team morale and disillusioned management.
- Lack of a clear plan for success. Mission, vision, strategy, and so on are common. What's really rare, but is most important, is an actual plan. What, exactly, has to happen for this to succeed? If the investment and potential downside are minor, the need for a plan is correspondingly low. But if there's a high cost for failure, a detailed plan is more important. I used to "just wing it" for everything all the time, but these days I'm doing more and more planning and preparation.
- Prioritizing speed over safety and resilience. Unrealistic goals and deadlines just set everyone up for failure and force them to cut corners, potentially sabotaging the whole effort when a preventable disaster happens. Few things in my career have felt worse than screwing up the one chance I had to succeed, because I was rushed to get things out the door before they actually worked.
- Failure to take the time to test and rehearse new things. The most important rule of experimentation is to first do no harm: it's OK to punch a hole in the hull of the boat if it helps you learn—but only if it's above the waterline. Or, as we used to drill when I was an EMT (Emergency Medical Technician): first secure the scene, then triage and stabilize the patient, and only then transport. To bring this back to the realm of DevOps and databases, you're going to introduce new tools, new automation, and new processes, but do it safely and carefully. Rehearse on non-production systems, get peer reviews, do it in dry-run mode, and make sure you have fast, well-tested recovery processes for any systems you're changing. I've seen buggy scripts doing things like removing the root volume instead of a subdirectory or iterating the EC2 API and terminating every instance instead of one.

Training is especially important. People are going to be learning a lot of new skills. They need time and help. When it comes to DevOps, remember the old story about the manager who said, "What if we invest in training our people and they leave?" and the other manager replied, "What if we don't train them, and they stay?" Vendors can be a big help with training, and any vendor in the DevOps space is likely to provide at least some education, because it's still an early-adopter market and requires a lot of professional services for their customers to cross the chasm.

It also can be helpful to make your *current* work and processes visible to the whole team. Usually, a lot of people have a limited view of how work gets done and who does it. This can make gaps more obvious to people with power, who'd care if they only knew. A lunch-and-learn talk on "how we deploy database changes" can work wonders

You Can't Buy DevOps

Although vendors can be partners in your success, you can't buy DevOps from a vendor. I've seen this attempted multiple times, and in some cases, it's had negative consequences. The tooling simply might not be adopted, as in the Chef deployment I mentioned previously. A more serious potential problem is if the vendor is endowed with too much power, they can destabilize things rapidly.

The first time I experienced something like this was early in my own career, when I was working on a team without proper version control or deployment tooling. This caused many problems, for which users such as me, not the tooling or those who resisted changing it, were blamed. I tried to advocate for change, and after the senior team members belittled me to silence me, I tried to move to a different team instead. Coincidentally, the company hired a consultant, who bluntly said we needed to use real version control and deployment tooling. I was in the room, watching but not saying anything, while the same people who'd scoffed at me were trying to save face with management by agreeing with the consultant and disparaging developers like me. I left the company soon after.

I tell this story to illustrate one of the ways placing credit, blame, or responsibility on a vendor can cause problems. There are different ways things can backfire:

- A vendor's approval, criticism, or other guidance can be weaponized by internal politics in ways the vendor and your managers can't see or prevent. In the story above, the admonishment from the consultant was turned into punishment and rerouted to the least powerful team members.
- A vendor or their sales rep can want the deal so badly they agree to be accountable for things outside the scope of what they can offer. This almost always turns into a parting of ways. An example from my own experience is the content writing firm I hired because the sales rep said they had a solid track record of achieving a specific outcome I was looking for. After I signed the deal, the delivery team told me the promise was impossible to keep. If a vendor promises their product will bring about a DevOps transformation, my advice is to call their customers and check references.
- A team member who champions bringing in a vendor risks their career if the vendor doesn't deliver on what the team or management perceives to be

the expected outcome. This is true even if the vendor didn't agree to, or even know about, the expectations. If Joe says, "Let's bring in Acme for deployment automation," and the boss says, "OK, but if I pay for Acme, I expect better site uptime," then if the site uptime doesn't improve, Joe's reputation and influence might be damaged. This can happen even if everyone agrees it wasn't Joe's "fault." Bias works subtly, and Joe might never regain the confidence or credibility he needs to be promoted, advocate for change, or even get his work recognized fairly.

- A vendor can walk right into the middle of a pitched political battle and be caught in the crossfire, damaging them and your future relationship with them at your next job. More than once I've seen a DBA ready to quit because of the misery of a non-DevOps situation involving the database, but the developers and managers didn't care. It's appalling, but some bosses and colleagues can be very OK with one team member taking all the punishment of badly designed cultures and processes. If they can hire someone to literally feed the machine with their life essence night after night, they'll continue to do so. My very first job after college was this way, and I really admired the DBA and hated the way he was treated. A DBA in this situation can be tempted to bring in a vendor as a last resort, on the slim chance their product can create a culture change. They know it's unlikely to work, but they're going to quit anyway if it fails. My advice is to quit immediately and save yourself and the vendor the heartache of seeing it fail. The company doesn't deserve you or the loyalty motivating you to such last resorts. Prioritize the relationship with the vendor, because good vendor relationships are a career asset, and take them with you into a company with the psychological safety you need to succeed and thrive.

- A vendor's products and sales processes can be designed to sell to where there's budget and ROI is easily quantifiable, rather than selling to where the opportunity is. A lot of products are sold like aspirin for a headache or sold via the psychology of fear and loss avoidance ("I've calculated your cost of downtime as a fraction of your annual revenue, and it proves you can't afford not to buy my solution"). In software sales, which is mostly stuck in pre-DevOps mindsets, traditional wisdom is developers have no budget, and only operations teams have problems worth spending large amounts of money on. This means there are a lot of firefighting tools for operations (first age of DevOps, at best) and not a lot of tools focused on developer productivity, which is also notoriously hard to quantify (do you prefer to measure lines of code, or tickets closed?). The second age of DevOps, focused on developer productivity and full-lifecycle ownership, is where the next opportunity is—yet many vendors don't believe in it, and will actively work against it in a budget land-grab for more first-age solutions. A vendor who says they'll help with DevOps, but then undermines it, is a big setback.

Another vote for "you can't buy DevOps" comes from Brian Dawson's article *6 Keys to Unlock the Value of DevOps*, in which he warns,

> Don't try to "buy DevOps." Some organizations operate on the belief that they can buy a set of tools that will make them into a DevOps organization. Don't fall for this trap. Tools are a critical part of connecting teams across silos and enabling the automation that supports DevOps, but tools alone will not solve your problem. You need to do all the hard work creating the culture and looping in all the important aspects of continuous integration and continuous delivery to ensure the tools are doing their jobs.

Achieving Continuous Delivery With Databases

In this and the following sections, I'll explore practical, tangible ways you can make progress towards a more DevOps approach to your databases. The exact steps and order will be context specific to your situation (a fancy way to say "it depends"), but I'm confident you can identify and take the next step that's right for you, create at least some improvement, and then reassess and repeat. In this section I focus on continuous delivery.

Continuous delivery is part of a confusingly named trio of related continuous practices. In brief:

- **Continuous integration** is about source code management, in which developers simultaneously merge their changes frequently and bidirectionally with a central repository and each other. This reduces the scope, magnitude, and duration developers' code can diverge from each other in incompatible ways. It's a little like a crowd of people moving: they shuffle in small steps to stay packed closely together.
- **Continuous delivery** automates the test, build, and deployment processes. Code checked into the source code repository is automatically packaged, and every step in the process of getting the working software into the hands of customers is triggered automatically, *except for the very last step*, which is still manual. (There's further nuance on this on the following few pages.)
- **Continuous deployment** automates the last step, wherein the customer gets to use the software. In continuous delivery, this final step is manual: a human decides when to "push the button." In continuous deployment, *everything* after you push code to the repository is automatic.

Continuous delivery is also identified as a key capability in the book *Accelerate*, and it's widely accepted as a baseline practice for DevOps in general. One of the major tenets of DevOps is to work in cycles, creating feedback loops to help early stages in the software delivery lifecycle benefit from information generated later. In other words, to bring insights from production operations back into development. A key to doing this effectively is to reduce cycle time, so iteration becomes cheap and fast. The automation in continuous delivery helps do this.

This is the virtuous cycle I mentioned earlier: speed leads to stability, which leads to more speed. As Adrian Cockcroft put it, "speed wins in the marketplace," but of course, speed also wins in shipping, observing, adjusting, and shipping again. If developers can make a small adjustment to an app, and quickly ship and observe the change in production, then they naturally prefer to work in small batches, which are more stable and result in less rework, wasted work, and unplanned work. They also tend to drive the fixed cost of change down, further amplifying the cycle.

The difference between continuous delivery and continuous deployment used to be clearer, but advances in how we do it have blurred the lines a little bit. To generalize, there are basically two levels of sophistication: without feature flags and canaries, and with them. This has major implications for continuous delivery and continuous deployment:

- **Without** feature flags or canaries, deploying software is tightly coupled to releasing it, so the final manual step in continuous delivery is to push the deployment button. Once the software is deployed, it's in production, receiving traffic and visible to customers. If it doesn't work, you have to do a "rollback," which usually means redeploying the previous working release.
- **With** feature flags and/or canaries, you can decouple deployment and release.[9] The new code can be dark-deployed to servers behind disabled feature flags, or on "canary in the coal mine" servers not receiving traffic, so the code is not yet released to customers. Then, the final manual step in continuous delivery to activate it is toggling the feature flag and/or tweaking proxies to route some traffic to the canaries. This activates the code, so you can observe whether it works as intended. If it doesn't work, you just toggle the flag back off, or switch off the routing to take the canaries out of service.

[9] The book *Lean Enterprise* has a chapter devoted to decoupling deployment and releasing Database Migrations

Canaries and feature flags are amazing because you can do things like enable the new code gradually; it's not just all-or-nothing.[10] For example, you can enable an experimental feature only for a small portion of the user population, or on a small number of canaries, so you can find problems before a lot of people or servers are impacted. And companies like Netflix do A/B regression tests including performance tests. With sufficient automation, new releases can be gradually and automatically rolled out if they work well and rejected if they degrade performance. Humans can be removed from the process almost entirely.

Continuous delivery is well-established practice in modern DevOps software engineering. But what about the database? Several vital parts of continuous delivery have to be designed to work well with the database, too. You need to treat schema as code; you need to automate database migrations; and you need to loosen the coupling between the application and the database schema, to enable deployment without release. In the following sections, I'll explore these in detail.

Schema as Code for Monoliths and Microservices

Your database has a *schema*, a set of data structure definitions or expectations describing what type of data is stored, what shape it is, how it's organized, and how it's related.[11] A key part of DevOps for the database is to recognize this schema is code, just like your infrastructure and configuration and all the other artifacts necessary to build and define your application. As such, it should be stored in source control, and subject to the same processes and controls as application source code. This enables not only controlling what's in production, but other benefits such as making it easy to tear down and recreate databases for development and other purposes.

Non-schema code in the database should also be kept in source control. This includes things such as stored procedures, triggers, views, functions, schemata, and roles. User accounts are better defined in your configuration-as-code tooling rather than in version control.

You can even version-control some configuration data such as definitional tables. Imagine, for example, the database needs a list of countries or states to populate a pull-down selection menu. The rows in the table should go into source control too. Some people also store other reference data in version control, such as datasets against which integration tests are run.

[10] Watch this space; the term "Progressive Delivery" seems to be emerging to describe it.

[11] This is true even if you have a "schemaless" database. The database management software might not enforce the schema, but it's still there. For more on this, see the section on Loosening the Application/Database Coupling.

In which source repository should the schema-as-code be stored? This depends on the application architecture you're working with. If you have a fine-grained microservices architecture, then each application or service will have its own dedicated backing store—usually a simple database with a few tables and little else. In this case, store the schema with the code it belongs to. If you have one source repository per microservice, put it there. If you have a monorepo, put it there.

If your application is monolithic or coarse-grained services, with a monolithic database and lots of commingled data accessed by many different applications or services, then you might want or need a separate schema repository to store all the schema-related code and data. In this case, the schema repository is associated more with the database than the application.

Either way is fine, at least to start. Beginning with a monolith is a sensible way to manage complexity. If the monolith needs to be broken up into separate services later, you can use the Strangler Pattern to refactor it incrementally and safely. The essence of the Strangler Pattern is to build a replacement microservice for some part of the monolith, direct user traffic to the new service, and remove the old code path from the monolith. In this way, the monolith can be shrunk a bit at a time. The book *Lean Enterprise* has a chapter devoted to the Strangler Pattern.

You can apply the Strangler Pattern to the database, too. You can break up and refactor a monolithic database by carving off one or two tables at a time into separate databases or separate servers. When the tables in the monolith are no longer used, you can drop them. You can choose natural boundaries to help you determine which tables to move, such as usage by an application or even transaction boundaries.

Whether you're refactoring an app or a database, however, it's important to *know* the old code or the old database table is truly no longer needed. If it's needed and you remove it, something will break.

The only way to know this is with reliable data. In the application this is easy: just emit a metric from the code path in the monolith, and if the metric goes to zero for a satisfactory length of time, you're done. It's a little harder in the database, because most databases lack built-in fine-grained observability and historical performance data. Nonetheless, there are ways. Database Performance Monitor is often used for this type of refactoring. A publicly available example is SendGrid. A table that should have been unused was changing, and the source of the queries was a mystery, so the refactoring was stalled. Database Performance Monitor made it easy to identify the source of the changes in minutes, which previously would have taken weeks.

Automating Database Migrations

Continuous delivery has some dependencies to make it work well. (None of this is all-or-nothing; it's all degrees of advancement), and automated migrations are one of the big enablers of continuous delivery. A migration is a modification or refactoring of the database, "migrating" it from one schema version to the next. A schema change is one simple example. An action like a schema change might also include a data migration, which is necessary when you're doing something like breaking up a column into two, or refactoring a 1:1 relationship into a 1:N or N:N relationship.

The topic of database migrations is large, and in this book, I'll only cover it at an intermediate depth, but I'll give you several references to reading where you can dig deeper into specific stories relevant to your needs. A few of the things you might need to consider and decide are:

- Whether to do migrations within the application code, or with an external tool.
- Whether to do declarative or patch (diff-based) migrations.
- How sophisticated your migrations need to be.
- What specific technologies and constraints will influence your choices.

Tooling. You can do migrations with separate tools designed for the purpose, or you can build your migrations into your application itself. In either case, a variety of tooling and libraries are available for you to evaluate. You don't have to build your own migration system from scratch. In fact, if you're using a popular application development framework such as Ruby on Rails or Django, reasonably sophisticated migration tools are already included.

Many techniques are useful in both approaches. A common one is schema versioning, in which the database itself can be inspected to see what version of schema it has. This is usually implemented by storing the version number into a table, or by recording patch identifiers in a table as they're applied, so you can see which patches have been applied to the database schema.

When you perform migrations by building them into the application, the application checks the database schema version at startup to make sure it's upgraded to the state it expects. If the database schema isn't up to date, the application runs a pre-boot process to upgrade the schema before it starts.

When you use an external tool for migrations, it's usually invoked from the deployment tooling, but it can also be invoked as a separate process, such as from a chatops command. It's less than ideal to have migrations exist outside of the normal deployment tooling. There are advantages to having all your

deployment go through a single set of tooling, so you don't need to coordinate between two paths from development to production. However, there's no reason you cannot start this way, and then merge your application and schema deployment tooling later.

The main limitation of including migrations within the application is it works best if the database is mostly single-purpose and your application technology stack is homogeneous. If you have a monolithic database accessed by lots and lots of different applications, and no application has exclusive ownership of the database, it becomes harder to do this. But if you have a microservices database, which essentially has only a single customer, it's easier.

No matter what tooling or techniques you use, the two most important things your migrations should achieve are automated and idempotent (and transactional, if possible) schema and data migrations . These qualities are important because you should run each migration many times. You should run the migration many times in your development environment, in your pre-production environment, and by the time it reaches production, you should feel confident it's going to work. This is in alignment with the DevOps principle that painful or risky things should be done more often, not less often.

The traditional, legacy way of doing a migration—a database administrator comparing production and development schema, and then applying changes to make production match development—is a practice you should prefer to eliminate. Automated, idempotent migrations are iterative and continuous processes, not one-time manual schema reconciliation efforts.

I know there are lots of tools for comparing schemas and generating scripts to change one into the other, but those aren't the type of migration tools I encourage you to use. That's the type of task I think a modern DBA should move away from, not invest in more deeply. The type of migrations you ideally want to be doing in DevOps practices are developer tasks engineered into the application or the CD toolchain, not DBA tasks. Making manual work more convenient leads to getting stuck in a local maximum, rather than optimizing the system. Remember the First Way of DevOps: *The First Way emphasizes the performance of the entire system, as opposed to the performance of a specific silo of work or department*.

Whether your tooling is standalone or built into your application, you need to choose between patch-based or declarative tooling. Most tooling is patch-based, with a pair of scripts for each migration between schema versions. There's a script to migrate the schema forward (or up), and one to migrate it backwards or down again if you want to roll it back.[12] However, there's also tooling that accepts a description of the desired target state, and makes changes if this isn't the state it finds when it inspects the database. The Skeema tool takes this approach and argues for the benefits of it. I have two hesitations with this philosophy:

1. You don't know exactly what operations your tooling will apply.
2. The conceptual between a declarative approach and a DBA manually using diff-and-merge tooling are confusing. I worry it might lead to arguments along the lines of, "We're already doing this, we don't need to change," when you might very much need to change. At various times in my career, I've seen similar dynamics. It's led to people resisting improvements, because they mistakenly feel what they're doing already is just as good, only different.

Migration Primitives. Schema migrations can be categorized into a set of universally applicable operations, each of which can be handled with a step-by-step process. The main migration primitives you'll need to handle are:

- Add, drop, or rename a table or column.
- Split, combine, or move a table or column.
- Change data types.
- Add or drop referential integrity constraints.
- Change stored code such as a stored procedure or function.

If you don't yet have a lot of experience with database migrations, and you simply follow your intuition and reasoning to develop your techniques from scratch, you're likely to reinvent some failure patterns already discovered by those who've gone before you. For this reason, it's well worth studying some of the existing literature on the topic and reading some of the references to articles I'll provide. Here are a few of the types of problems and solutions you'll eventually need to be familiar with:

- Some kinds of operations (i.e., dropping a table or column) are harder and more complicated than others, and depending on your application code and how well-behaved its queries are, they can make it easier to introduce breaking changes. The solution is to study the patterns of migrations others have choreographed

[12]Rollbacks are a trap in my experience. I agree with Dan McKinley, who wrote about why rollbacks aren't possible. In the rest of this book, you'll see multiple mentions of avoiding rollbacks. ORM migration tools always have rollback functionality, but in my experience, it eventually leads to trouble.

- Locking and long-running operations are often problematic. The solution is to study your database and figure out how to use nonblocking operations when possible; and to do migrations in small chunks of data instead of in one fell swoop.
- Big changes can introduce compatibility or performance problems you didn't foresee. The solution is to trickle migrations out gradually in many small steps and do gradual canary releases, instead of doing a lot of operations all at once.
- Rollbacks are essentially impossible, as per the Dan McKinley article I mentioned earlier. The solution is to design your tooling and application code to be able to deploy schema and application in separate deployments, instead of bundling them into one, and to do a separate roll-forward deploy to undo a schema migration. I'll discuss this more in the next section.
- Some database architectural patterns, such as flipping back and forth between active-passive pairs of replicated servers, have been popularized but are nonetheless very problematic. The solution is to familiarize yourself with a variety of architectural patterns and be sure you Google search or ask peers for stories of negative experiences when you're considering a technique.

Migration Case Studies. Let's talk about how some specific teams do database migrations. I'll start with a discussion of how we do it at SolarWinds. Our architecture is a mixture between monoliths and microservices, and we use MySQL and Go extensively. We've developed a standalone migration tool in-house, and we keep our database schema in a dedicated repository. Migrations are kept in separate files in this repository, with a file naming convention to help the migration tool find each migration and determine whether it's been applied to the database, by looking up a migration identifier in the database's version table.

This tool is executed separately from our automated deployment tooling, and developers need to coordinate the deployment of schema migrations with their releases of software. This isn't ideal, because it requires developers to know how to synchronize these activities. But in practice, our schema evolves relatively slowly, and we don't feel this is the most important problem for us to solve.

We've been using this relatively simple system for a long time. It has some legacy assumptions built into the tooling, such as assuming it's possible to connect directly to a database server's EC2 instance via SSH (this isn't possible with Amazon RDS, which we're starting to use). As we evolve and mature our infrastructure, we're invalidating some of those assumptions and we need to change the tool accordingly. However, this system is reasonable, and works well for our current needs

In a previous life, I worked at a company using Microsoft SQL Server, where we had a rule that all database access had to go through stored procedures. There were a very large number of databases and a mindboggling number of stored procedures at this company. The stored procedures acted as a compatibility layer to shield application code from the database schema. It wasn't managed in source control and had a lot of other problems, but the basic pattern was successful in the sense that it let us evolve the application and schema independently. So, although it was crude by modern standards, it was effective.

Several other companies have written about how they do database migrations. Here's a summary of a few of these:

- **Benchling**. As their team wrote in *Move Fast and Migrate Things: How We Automated Migrations in Postgres*, they use SQLAlchemy and Alembic with PostgreSQL. They've refined their schema migrations over the last couple of years and wrote up a fairly extensive article documenting the key things they learned during this process. The key takeaways are automation saves time and errors; iteration and speed is a winning strategy; they needed to account for locking, long-running operations, timeouts, and retries; having pre-deploy and post-deploy migrations was a problem, and it was better to remove post-deploy migrations and do them separately in another deploy; and their ORM sometimes used columns in ways they didn't expect and couldn't readily see from the source code (another testament to the importance of good observability).
- **BlackStar**. Responding to an article on continuous deployment at IMVU, Tony Bowden writes on how BlackStar does gradual deployment of schema changes by evolving the schema and application gradually, a step at a time, each step maintaining cross-version compatibility and eventually moving towards the desired target state.
- **Braintree**. The article *PostgreSQL at Scale: Database Schema Changes Without Downtime* documents several years of lessons learned about how to do zero-downtime schema changes at scale in a payment processing company. Key takeaways include making sure adjacent versions of the application and schema are forward- and backward-compatible with each other (also the topic of the next section of this book); ensuring locks are short; rollbacks don't roll back the schema; schema changes don't combine lots of modifications in a single transaction; and there are many rules and requirements for various types of schema change operations, based on what they've learned can cause problems.

- **Etsy**. Etsy is one of the early successes of DevOps and continuous deployment. However, although their Deployinator tool helped them move from slow, infrequent code deployments to fast and frequent ones, schema changes remained hard at first, and were bunched into a single day every week, which was a painful and risky process. In 2011, they were partially through this process, and spoke at Surge on *Scaling Etsy: What Went Wrong, What Went Right*. In 2012, they wrote Two Sides for Salvation, explaining how they use two-server shards to perform migrations half a shard at a time. And then in 2013, they wrote about *Schemanator: Love Child of Deployinator and Schema Changes*, elaborating on tooling they built to make schema changes faster and easier. Etsy is a rare example of stories from a while ago, when few companies were doing this. This peek into history lets you see how quickly DevOps database migration strategies have evolved over the last handful of years. I think they were among the first to tout the benefits of using feature flags to disable features until the required schema is deployed, for example. I think they'll forgive me for saying the lessons they've shared have been a major reason many companies avoid the active-passive two-sided sharding topology; my personal experience with this has been as a consultant, in which capacity I repaired many problems caused by two-sided sharding. Other lessons Etsy learned often reflect older versions of MySQL, which suffered from brittle replication and unavoidably blocking schema changes.
- **Facebook**. If you speak German, a recording of a talk given by Facebook's Martin Mazein is available, and he discusses schema changes at scale beginning around 13:15 minutes in. If, like me, you don't speak much German, the slides are in English. Key problems they faced were locking and space constraints. Their solution is an automatic online schema change toolset that a) handles the operational aspects by using triggers to change table structures in parallel while the application continues to change the data, and applying schema changes as deployments, and b) takes care of development needs such as comparing schemas, creating diffs automatically, and maintaining the diffs in version control.
- **Gilt Groupe**. Mike Bryzek has written and spoken extensively about his experiences building high-velocity, high-reliability DevOps processes and culture at the Gilt Groupe. A significant part of this was schema changes. In *Schema Evolutions at Gilt Groupe and Handling Microservices in the Real World*, he shares key lessons, such as disallowing rollback, and isolating schema changes to their own deploy instead of rolling out schema and code changes simultaneously. Implicit in this is the decoupling of schema version from application code version I've mentioned a few times already. Mike is now at Flow and continues to apply leading-edge DevOps techniques.

- **SendGrid**. In Silvia Botros' guest appearance on *Real World DevOps*, she discusses the importance of schema migration and versioning, how most migration frameworks built into ORMs and app frameworks like Ruby on Rails don't work at scale, how migration rollback in databases isn't actually possible, and the importance of enabling developers to own their schema instead of making it a DBA task.
- **Stack Exchange**. In *SQL is No Excuse to Avoid DevOps*, Thomas A. Limoncelli explores why and how schema changes can be done continuously. He advises applications to check the schema version of the database they connect to, and upgrade it on startup if it's outdated, by running scripts stored in a database table. He doesn't explicitly say so, but I believe his day job involves SQL Server—not that it makes a difference. Everything he writes about is applicable to any technology.
- **Zalando**. Zalando has spoken various times about the extensive tooling and processes they have developed to make sure schema changes can be deployed quickly and reliably at very large scale, with a great track record of reliability. They use PostgreSQL with a layer of stored procedures as a mapping between underlying schema and the application code, abstracting away the schema itself. Developers are responsible for their application's schema and SQL, supported by training, tooling, and monitoring. One of the benefits of this system is developers get familiar with the database and learn how to write good SQL; DBAs are called in to help only when specialized knowledge is needed. They have established patterns for performing common changes to applications and schema, and ways to ensure there's cross-version compatibility. Schema changes are scripted in version-controlled patches, tooling checks version tables to see what's been applied to the database, and dependencies or incompatibilities between the app and database are encoded explicitly. Big changes are processed in small chunks, which are throttled and controlled by tooling. This development culture and tooling was created in response to problems they previously experienced, such as long-running migrations causing database bloat, increased load, and locking. All of this is elaborated in PGConf.EU 2014 - *Alter Database Add Sanity* and OSCon 2015 – *Enabling Development Teams to Move Fast With PostgreSQL*.

In every case, the engineers have crafted their solutions carefully to a) work around the limitations of their database technology and b) take advantage of extra features their database offers. For example, PostgreSQL users frequently cite its sophisticated support for transactional schema changes as an advantage.

Off-The-Shelf Migration Tools. Most of these stories involve migrations at a scale and size requiring custom tooling, but most companies are *not* running at that scale or with those types of constraints. Instead, most teams will find simpler, lighter processes will work, at least as a starting point. Off-the-shelf tools and frameworks have a lot to offer. Here's a few you can study; a web search will find more like these.

- Active Record (Rails ORM) migrations
- Alembic, a migration tool for the SQLAlchemy database toolkit for Python
- DbUp, a tool originally for SQL Server, which now supports more databases
- Django ORM's migrations
- Flyway, a tool for running migration scripts
- Goose, a database migration tool for Go
- Liquibase Database Refactoring tools
- Mayflower, a simple, forward-only, database migrator for SQL Server and .NET
- pgTAP, PostgreSQL functions for unit testing
- Schema Evolution Manager, originally developed at Gilt Groupe by Mike Bryzek
- Sequel ORM's migrations
- Skeema, a pure-SQL schema management utility
- Squitch, a database change management application
- Strong_migrations, a tool for catching unsafe migration operations while coding

In addition to these, other tools for managing databases at scale have come out of big companies, such as social media websites. I was involved in the evolution of nonblocking schema change software for MySQL. Facebook was the first to release an "online schema change" software tool, but it wasn't flexible or easy enough for the general public to use, so I helped write a newer tool inspired by it for Percona Toolkit, called pt-online-schema-change. This used essentially the same techniques—creating a new table with the new data structure and migrating data to it in the background, then doing an atomic swap when it was finished. This technique is OK, but MySQL's quirks make it difficult and complicated to do correctly, and tooling can't work around all of those problems. GitHub developed what I consider the successor to that tool, gh-ost, which uses a slightly different technique and might be preferable if you need to migrate extremely large tables under heavy load with zero downtime.

Other tools often are useful examples to study but aren't readily usable in environments much different from the company in which they were developed. An example is the Jetpants toolkit by Tumblr. There was some initial interest and quite a few conference talks from its developers, but I'm not aware of anyone other than Tumblr using it, perhaps because it's so Tumblr-specific. It appears to be dormant now. Another example is the built-in schema swap functionality for long-running schema changes from Vitess, which is interesting to study but probably won't be applicable elsewhere.

Finally, there are schema migration tools from companies such as DBMaestro and Redgate. I'm not here to disparage any tools, vendors, or those who use and benefit from these tools. There's value in these tools. But I want to emphasize the best continuous delivery toolchains I've seen are built upon automation and engineering, not a more convenient way to do the work manually. So, if you're using their tooling this way, I want to gently challenge you to see if you can stop doing that work, for the sake of optimizing the system and not your individual tasks.

Loosening the Application/Database Coupling

A perennial topic in removing the database as a bottleneck to continuous delivery is the need to break the tight coupling between the application and the database. In short, you need to be able to upgrade the database schema without breaking the application, and ideally you also need to be able to upgrade the application and run it against an old schema version. This lets different people and systems operate, as Thomas A. Limoncelli says, "in coordination but not in lockstep." My own experience bears this out too, as does each of the stories I cited in the previous section.

If possible, it's nice to have at least a single versions' worth of both backwards- and forwards-compatibility between database and schema. There are three big reasons why:

- It lets you deploy the application and database schema upgrades independently.
- It enables deploying both application and schema upgrades, without activating the application upgrade—putting it behind a feature flag and gradually enabling the flag to see if anything breaks.
- It's mandatory in highly distributed applications, where there are a lot of different application servers and database servers. Instantaneous, synchronized, consistent, zero-failure deploys and upgrades of all those moving pieces are simply impossible, so by definition application and database schema versions won't match all the time.

If the application can't tolerate an older or newer schema than it was built for, deploys can either break things or require downtime. But if the application isn't coupled tightly to the database schema, zero-downtime and no-breakage deploys are possible.

This isn't too hard to do, but there are simple rules and patterns you have to follow closely to make it work. For example, don't do SELECT * queries. These will return different results when run against different schemas. Name every column explicitly and you'll get the same result set back from a new schema version with a new column. Simple, but important.[13]

[13] Avoiding SELECT * has other benefits too, such as making the app's actual usage of the data more obvious, thus avoiding some potential fear and doubt about whether changes will break things.

In simpler scenarios, with less-distributed apps and no out-of-order deploys, you can get by with a best-effort approach that isn't fully bidirectionally compatible. If the old app can work with the new schema, most things are manageable. This confers most of the benefits without all the rigorous requirements.

What about using a schemaless NoSQL database? The schema isn't contained solely in the database. The application encodes expectations about the data's shape, size, and relationships, which can be enforced by the database's schema validation rules—which is really all the database's "schema" is, validation rules. But the schema really lives in the application and the data, not the database software. The database's schema is just a double encoding of the data model. In fact, the most important knowledge of—and coupling to—the data's schema is in the application code, and NoSQL databases do little to eliminate or even ease that.

To return to a couple of the stories from the previous section and elaborate a bit more on this topic, at Zalando the basic rule is trying to read from the new structure, with a fallback to the old structure if the new doesn't exist. Then after migrating data, they read from the new, and write to both old and new. They manage multiple schema versions, stored procedure API versions, and application versions to simultaneously, to ensure dependencies don't create stalls in the flow of work. There's a new database/schema version per deployment version. This schema versioning is achieved through the PostgreSQL search_path parameter, which the application sets when it connects.

I don't have insider knowledge of the system at Zalando (you can read and watch their talks for some more details), but it sounds a lot like the systems I was involved in early in my career. We also used stored procedures[14] to insulate apps from the database's table structure. But it sounds like they've done it well, with modern tooling and good coding hygiene—and with developers bearing most of the responsibility, which I particularly like.

Putting a lot of the schema logic into stored code in the database enables things like centralizing the "read from the new, fall back to reading from the old" logic that otherwise might need to be put into the application code. Similarly, using views, triggers, and functions can help ensure things work correctly in both read and write operations while both the schema and data are being migrated simultaneously. And since they use PostgreSQL, they can rely on features such as updating schema and stored procedures in the same transaction, ensuring they work together seamlessly

[14]To be clear, I've since become a fan of not using stored procedures, but I admit they have benefits.

Zalando's is a *highly* flexible and sophisticated system, but yours doesn't have to be so involved. I'm a big fan of the simplicity of Thomas A. Limoncelli's article. It has two major parts: automated migrations (see previous section) and how to code for multiple schemas (this section). As he points out, coding for multiple schemas is about separating deployment and release.

Everyone uses mostly the same basic techniques to accomplish this. Many of the tools and articles I've linked previously have discussions of the steps needed to migrate schemas on live systems without breaking them, but Limoncelli's article is the most succinct, so I'll quote it in full:

> **The Five Phases of a Live Schema Change**
>
> 1. The running code reads and writes the old schema, selecting just the fields that it needs from the table or view. This is the original state.
> 2. Expand: The schema is modified by adding any new fields but not removing any old ones. No code changes are made. If a rollback is needed, it's painless because the new fields are not being used.
> 3. Code is modified to use the new schema fields and pushed into production. If a rollback is needed, it just reverts to phase 2. At this time any data conversion can be done while the system is live.
> 4. Contract: Code that references the old, now unused, fields is removed and pushed into production. If a rollback is needed, it just reverts to phase 3.
> 5. Old, now unused, fields are removed from the schema. In the unlikely event that a rollback is needed at this point, the database would simply revert to phase 4.

This pattern is documented as the "McHenry Technique" in the book *The Practice of Cloud System Administration*. It is also called "*Expand/Contract*" in *Release It!: Design and Deploy Production-Ready Software*, and it's related to the Strangler Pattern mentioned previously.

Here are a few other books that discuss database migration and evolution in quite a bit of depth:

- *Lean Enterprise*: *How High Performance Organizations Innovate at Scale*, by Jez Humble, Joanne Molesky, and Barry O'Reilly. The chapter titled "Decouple Deployment and Release" is good reading—as is the rest of the book. This book is one I'd recommend.
- *Database Reliability Engineering*, by Charity Majors and Laine Campbell, discusses topics such as schema changes in Chapter 8, "Release Management." This is an excellent book, and I recommend it too.
- *Refactoring Databases: Evolutionary Database Design*, by Scott J Ambler and Pramod J. Sadalage, has a wealth of good information and advice.
- *Migrating to Microservice Databases: From Relational Monolith to Distributed Data*, by Edson Yanaga, has a chapter titled "Evolving Your Relational Database." My only caution would be to note there's some nonstandard terminology (sharding is used to refer to migrating data in small chunks, rather than partitioning an application's data across many servers), and some of the techniques are discussed in slightly less depth than you'll need when you implement them. Finally, it talks about rollbacks a lot. As I've noted previously, and as many others such as Dan McKinley have corroborated, you can't rollback. You can say you do, but after the rollback, your application's overall state isn't the same as it was before.

Improving Database Observability

A core tenet of DevOps is measurement, which is necessary for the feedback loops that help us increase the rate at which we can make our systems run better. One of the big reasons it's hard to apply DevOps practices to the database is how hard it is to measure them and figure out what they're *truly* doing—both external activity on behalf of user requests, and internal state and mechanisms.

Observability Is Foundational. Observability (aka measurement) is a fundamental need for all of the stages in the progression of maturity I laid out in Database DevOps Capabilities. Without observability, it's impossible to safely automate, it's impossible to have confidence about migrations, it's impossible to employ continuous delivery, and so on. Each of the capabilities and categories of development rests on a foundation of being able to observe and monitor the database. As Google writes in the SRE book,

> Without monitoring, you have no way to tell whether the service is even working; absent a thoughtfully designed monitoring infrastructure, you're flying blind. Maybe everyone who tries to use the website gets an error, maybe not—but you want to be aware of problems before your users notice them.

The phrase "if you can't measure it, you can't manage it" is a bit of a trope and not entirely valid, but it's still one of the first things you should do—if not *the* first. When Mikey Dickerson took leave from Google to work on healthcare.gov, monitoring was one of the first and most important steps the team took to tame the situation and enable rapid forward progress. To cite the Google SRE book again, they show Dickerson's Service Reliability Hierarchy—an analogy to Maslow's Hierarchy of Needs. This diagram was developed to help communicate the importance of tackling first things first when rescuing the troubled health insurance marketplace:

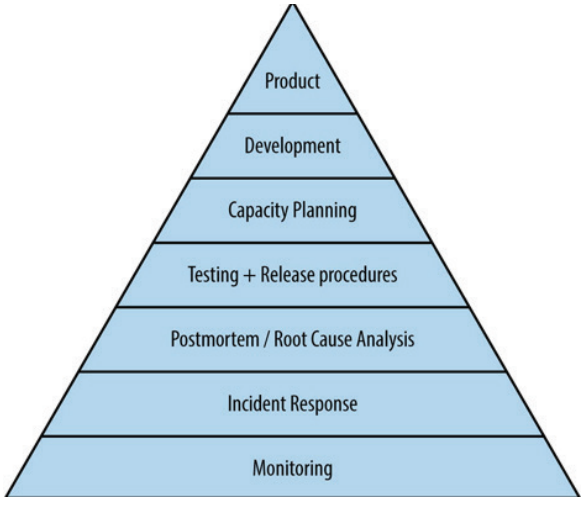


Figure 1. Dickerson's Service Reliability Hierarchy. Image: https://landing.google.com/sre/sre-book/

Observability Is Hard, But Important. Silvia Botros discusses the difficulty and importance of database observability in her appearance on the *Real World DevOps* podcast. (There's a full transcript if you want to read it instead of listening to it). Here's an excerpt:

> Silvia: Observability is difficult… someone comes up to your desk and says, "We think the database is slow." It's a very vague problem… and for a very long time, there was not a whole lot of tooling that actually made observability of a data storage layer's performance a thing that everybody can look at.
>
> Mike: Yeah, I remember when I first started my system administration career 10 plus years ago, the troubleshooting in the database performance was a really hard problem in that there was just nothing there. Like, what do you look at? Sure, I could look at CPU, memory, and all these OS metrics but that doesn't actually help because inevitably…
>
> Silvia: It doesn't tell you what's actually the thing that's running, yup.
>
> Mike: Right, inevitably it's some long running query or some unoptimized query, but how do I find that out?

There are a few reasons for this, and when you put them into perspective, unfortunately it's an even more serious situation than it looks like at first glance:

- The database is code you didn't write, so you not only don't know how it really works, but you can't always just put a printf() into it or get a metric out of it and learn. Even with tools like DTrace, figuring out what the database is doing is hard, and sometimes impossible or unsafe on production systems.
- The database is a kind of virtual machine designed to execute "declarative code" you did write (SQL statements), but in a way that's highly abstract and very difficult for most people—even smart engineers—to understand and use.
- The database has hundreds of "metrics" you can draw on to graphs, but most of them have no intrinsic meaning or usefulness. And again, they're metrics about code you didn't write, so even the potentially important ones are hard for you to interpret.
- The most important thing the database is doing is executing SQL (or equivalent), but this is literally the least-instrumented part of nearly every database. Databases expose lots of metrics about their "status," but much less about the details of their workload. More on this topic later in this section.

So that explains why it's hard. But observability is important. If you can't measure and understand what your database is doing, it becomes really hard to apply a DevOps approach to it. It *especially* becomes hard to get your team into the second age of DevOps, where *developers* are responsible for the database performance of their own application—a point Silvia and Mike alluded to in the excerpt above: making "performance a thing that *everybody* can look at." It's unreasonable to ask developers to manage systems they can't observe.

The good news is, it's possible to do, albeit not always easy. You need to measure what you should, not what's easy to measure. What the database exposes most readily to you is the easy stuff. But you need to get the valuable stuff, which requires more work.

CELT+USE: The Seven Golden Signals. How do you know what's valuable? A framework or methodology can help, and I've spent a lot of my career thinking about this. The valuable stuff is a pretty basic set of telemetry, much smaller than the overwhelming volume of distracting metrics most databases emit. First, there are two important categories of things you need to measure about a database:

- The database's customer-facing workload and QoS (quality of service). The *workload* is the entire population of user requests it handles—queries/SQL/whatever is the appropriate term for the database you're using. The *quality of service* is the user-focused definition of performance. Users want their queries to return quickly (low latency) and correctly (no error). Thus, you can characterize workload and its QoS holistically with the CELT acronym: Concurrency, Error rate, Latency, Throughput. There's a lot of performance theory substantiating this claim, but I'll skip it for brevity. Just remember the four "golden signals" of what the database is doing and how well it's doing it, from the external (customer) point of view, are CELT.
- The database's internally visible status and health. This is the sufficiency and health of each of the four key resources the database uses to execute the workload it's given: CPU, memory, storage, and network. You can characterize each of these with the USE rubric from Brendan Gregg: Utilization, Saturation, and Errors.

To understand your database's performance and behavior, you need both the internal and external perspectives, or CELT+USE. Customers only care about CELT, but as a service operator, you need to care about USE for purposes of troubleshooting *why* CELT shows a performance problem or poor customer experience, and for things like capacity planning.

CELT+USE is an amazingly complete and effective set of seven golden signals to monitor your database. In my experience, most people monitor too much about their databases, and often mostly things end up not helping much. They miss some of the CELT+USE data because it's hard to get, focusing on other low value but easy-to-get things. (It's not their fault! It's the natural outcome of trying to figure out how to monitor a system vomiting noise and lacking signal.)

In addition, you need to measure and monitor the database's "sharp edges," the technology-specific things that are extra brittle, likely to fail, or easy to overlook or mess up. These usually are things even a beginner knows about. In MySQL, it's things like replication, which seems to have a million failure modes and only one way to work well. In PostgreSQL, it's things like VACUUM, which never seems to be tuned right for the workload. In MongoDB, it's things like lack of indexes.

How to Get CELT Metrics. The *internal* view of the database's status and health is usually easy to measure and well instrumented. Ultimately, the extra work you need to do is all about observing the *external*, workload-centric point of view: the CELT metrics about the stream of event data representing user requests (queries, SQL, etc.). There are a few ways you can do this:

- **Query logs**. Built-in, but high-overhead to enable on busy systems; accessible only to privileged users; expensive to analyze; and most of the tooling is ops- and DBA-focused, not developer-friendly.
- **TCP traffic**. Golden, if you can get it: low overhead, high fidelity, and great developer-friendly tools exist. The main downside is reverse engineering a database's network protocol is a hard, technical problem. (Database Performance Monitor shines here.)
- **Built-in instrumentation**. For example, in MySQL, it's the performance schema statement digest tables; in PostgreSQL it's the pg_stat_statements extension; MongoDB doesn't have a great solution for this internally, but you can get most of the way there with the top() command; or if you can accept higher performance impact, the MongoDB profiler. These have to be enabled, and often aren't enabled by default, especially on systems like Amazon RDS, but this instrumentation is valuable and worth enabling, even if it requires a server restart. Oracle and SQL Server have lots of robust instrumentation. Most monitoring tools can use this data.

Database Observability Empowers Developers. The most important factor in which telemetry you choose isn't about the technical ramifications. Your first consideration should be *which tools enable developers to have immediate, high-resolution, self-service visibility into the database's externally focused, customer-centric workload quality of service*. I have an obvious bias, because I founded VividCortex (now part of SolarWinds), a company focused on transforming

traditionally DBA-focused monitoring tools into tools for every engineer. But biases or not, it matters because it *works*! As SendGrid's Chris McDermott said, "Granting our development teams access to VividCortex has enabled them and driven more ownership and accountability. The turnaround time for identifying a performance problem, pushing a fix, and confirming the result is dramatically lower."

That's why I'm so unapologetically passionate about database observability. This outcome is so important, because this is what's needed to get out of the "bad old days" in which nobody can figure out what the database is doing without asking the DBA, who then SSH's in and does stuff at the commandline and reports back. As Dr. Nicole Forsgren, PhD. wrote in *Unblocking the Database Bottleneck in Enterprise DevOps Deployments* (emphasis mine),

> Check your change scripts into source control, run automated builds, get notified when database changes "break the build" and *provide your engineers with the tooling they need to find and fix database problems* long before they become an issue in production.

This is the main reason database observability is so important for applying DevOps approaches to the database. If you don't provide your developers with self-service, highly developer-friendly, usable tooling, I don't see how you're ever going to get the DBA out of the critical path of software delivery. You're never going to remove legacy DBA responsibilities as a business bottleneck. You're never going to get your DBAs out of reactive firefighting mode, so they can contribute their expertise in high-value strategic ways.

Observability and Mental Models. The other reason this high-resolution database observability is crucial is because *database workload is a critical signal about the application itself*. Our mental models of our systems are incomplete. When an engineer gets access to telemetry about an aspect of their application they didn't have before, it invariably results in them finding previously unknown bugs. Literally as I write this book, a technology company is trying Database Performance Monitor and discovering giant problems "hidden in plain sight" now that they can see what the databases are doing:

> *In less than five minutes I found the most time-consuming query across all shards, inspected a sample, and realized we didn't need to do any of that work*.

It makes me happy this insight is coming from Database Performance Monitor, but dear reader, Database Performance Monitor isn't the only way to get observability and find these types of wins. Any sufficiently powerful observability solution of whatever ilk—Honeycomb, LightStep, Scalyr, etc.—could enable this engineer to find and fix these problems, if it were set up to ingest the right data.

The bigger point is none of us truly understands how our systems work. The signals we get from our databases are important information we need to form a more complete and accurate mental model. It's a critical DevOps feedback loop not only for the database, but for the application overall. I'm not the only one who thinks so; there's research backing up this claim. From page 127 of *Accelerate*:

> **Make monitoring a priority** … the visibility and transparency yielded by effective monitoring are invaluable. Proactive monitoring was strongly related to performance and job satisfaction…

Finally, in my experience, it's important to get database-focused database observability, not "database observability from the application's point of view" via an observer at a distance, like an APM product designed to instrument the application code. How many times have you experienced the following?

1. You drill down into a slow page load's stacktrace in an APM dashboard.
2. You find a single-row query taking multiple seconds for no apparent reason.
3. You try running the query manually to see why it was slow.
4. Instead of being slow, it's instantaneous, as it should be.
5. You bang your head on a convenient nearby object in frustration.

This happens constantly. What the database is doing, and what the application thinks the database is doing, are often very different. The APM product measured the application sending a query to the database, but it didn't see what else the database was doing. Perhaps it was a backup or a schema change; perhaps someone opened Tableau and ran a query for the marketing team; perhaps it was even a monitoring tool doing something unwise. Measuring the app can't explain database behavior and performance. Only measuring the database can do that.

Democratizing Database Knowledge and Skill

Many software developers don't feel confident and capable of assuming full responsibility for the database. There are several reasons for this, all of which I think are valid and understandable.

- Databases are often not taught, or taught only superficially, in engineering and computer science curricula.
- Many developers are trained only in imperative programming, not declarative or set-based logic, and the gap between most programming languages and SQL is quite significant. Thinking in sets has a steep learning curve if you've been trained to think mostly in commands.
- Databases are among the most complicated and confusing software we use, with many internal mechanisms designed to do miraculously hard things. For example, they ensure consistent behavior in the presence of concurrent access to the data, while also providing availability and safety of the data.

For these and other reasons, people who cross the threshold of knowledge with databases, and come to appreciate and enjoy it, often see an opportunity to specialize. This is how a DBA is born. This specialized knowledge is great, but it's important to distribute it widely, rather than relying on a single person for everything database-related.

Not everyone needs to become an expert in databases. But everyone can, and I think everyone should have a working knowledge of the most important skill sets required to successfully build an application reliant on the database. These include

- The fundamentals of good data modeling.
- Writing good queries using the most basic constructs like JOIN and GROUP BY.
- Indexing and other basic optimizations.
- Avoiding common mistakes like N+1 patterns.
- Familiarity with some of the database's technological and operational "sharp edges."

As Will Gallego writes in *Support Driven Engineering (SDE)*,

> *Part of DevOps is about avoiding the throwing-over-the-wall principle between teams and dismantling silos that reinforce bureaucracy and mistrust amongst engineers. A classic example of this older thinking in engineering is the following:*
>
> *"I can't work on this SQL query, that's a job for our DBAs."*
>
> *With a DevOps mindset, not only should I have a strong desire to see a project through and communicate with others, I should have a strong desire to learn about the deeper requirements beyond the surface area required by my day to day.*

This doesn't need to be a big effort. But it requires people who have the specialized skills and knowledge to use the 80/20 rule and put in some extra effort to share those basics with other people. Likewise, it requires people who haven't yet developed those specialized skills to be curious and motivated to learn. It doesn't take much to get started, but it quickly pays big dividends and develops into significant momentum towards familiarity, comfort, and competence with the database extending far beyond a database administrator. Teams who have a broadly shared database skill set can work much more independently and quickly.

A few books I recommend are *Database Reliability Engineering* by Laine Campbell and Charity Majors; *SQL Performance Explained* by Markus Winand; and *SQL Antipatterns* by Bill Karwin. I also recently became aware of *Mastering PostgreSQL in Application Development* by Dimitri Fontaine, and although I haven't read it yet, the table of contents looks great. And of course, there's my own book, *High Performance MySQL 3rd Edition*.

Here's a list of simple practices I've seen in nearly every high-performing engineering team I've gotten to know. Each of these is relatively simple but pays big dividends.

Deploy Confidence Procedures. When an engineer deploys their code into production, one of three things can happen: they can check and gain confidence quickly that all is well, then move onto other tasks; they can assume it worked and move on to other tasks; or they can agonize about whether they've just broken the systems, and spend a long time trying to figure out, without confidence they know for sure. Two of these three options are ineffective and harmful.

High performing teams have post-deploy procedures engineers can use to assess the impact of their changes quickly. This can take the form of dashboards to check; wiki pages to go through; chat commands to run; or key performance metrics to observe. This is a shared artifact many engineers have contributed to over time, building up tribal knowledge in a shared and accessible format, not locked in the head of one senior engineer who has been in the team a long time and knows how to figure out whether something is broken.

At Etsy, for example, the deploy tooling itself has links to actions an engineer should take after they run a deploy. Here is a screenshot of Deployinator, Etsy's deploy tool. Notice the links at the bottom:

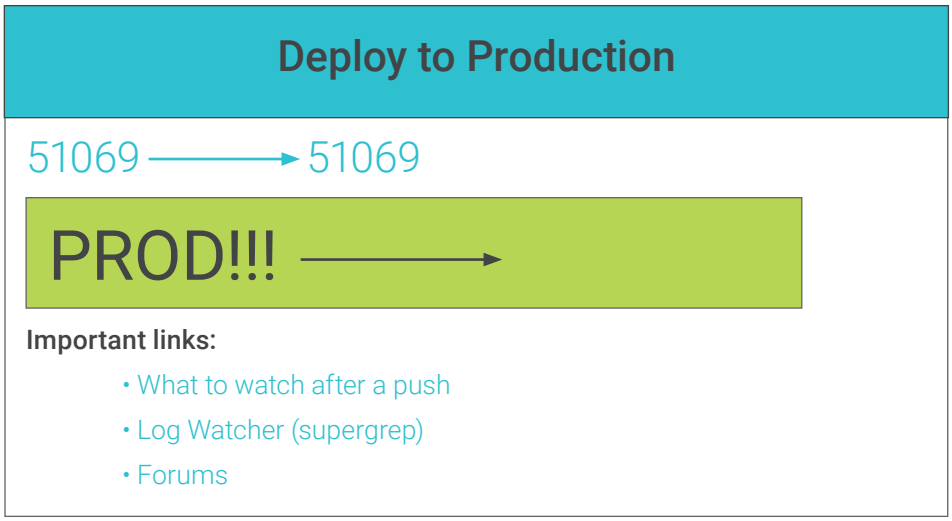


Figure 2. Etsy's Deployinator tool contains links to post-deploy processes and tooling.

Another example is GitHub, who uses an internal wiki page for their deploy confidence procedures. Engineers are expected to work through it and treat it as a deployment checklist. It contains things like chat commands to run and links to dashboards.

At SolarWinds, we use a time-versus-time comparison feature in our own product to quickly see what changed after a deploy. We deploy to staging first; our staging system is a full-fledged, production system monitoring our production systems, so it has a real workload and reveals real problems.

Runbooks. A runbook is a process for approaching a situation, such as responding to an alert when you are on-call. The runbook is similar to a post-deploy process, but in a different context: it's designed to help you remember which situation-specific things are important to check, evaluate them quickly, and try to rule them in or out as you investigate the situation. Runbooks are a key part of effective incident response, and getting them out of the DBA's head and into something like a wiki page is a step towards enabling the team to troubleshoot without relying on a single person. Runbooks should ideally be referenced from alerts relevant to them, so when you get paged at 3 a.m., the alert itself has a link leading directly to the runbook.

As Silvia Botros wrote, "There are a lot of production DB issues we may encounter as DBAs that are simple for us to debug and resolve. We tend to underestimate that muscle memory… Some simple documentation on how to do initial debugging, data collection, can go a long way in making the rest of the operations team comfortable with the database layer and more familiar with how we monitor it and debug it."

Psychologically, runbooks also play an important part in team dynamics. Writing a runbook for your colleagues[16] is a not-so-subtle way of telling them you care about them and you have their back. Engineers who know they have runbooks to rely on feel supported, instead of feeling like they've just been thrown into a situation they can't handle.

At SolarWinds, we again use our own product, which has a Notebooks feature. It's like a combination of a Markdown-based rich text wiki, and a graphing and charting tool with live, annotated data embedded. Notebooks are good for lots of purposes: postmortems, deploy confidence dashboards, documentation of SLIs, and runbooks.

Effective Incident Response. A culture of well-practiced, systematic incident response is important for DevOps overall, and supports efforts to involve the entire team in effective DevOps for the database. There is nothing database-specific about effective incident response, and any team can get better at incident response simply by making it a priority. A great resource for this is the PagerDuty incident response documentation, which they have open-sourced, so other companies can learn from it.

[16] And with them, too—you need team participation, testing, and feedback.

Performance Indicators. A simple, clear, documented way to quantify whether performance is acceptable or not is very helpful in giving engineers confidence they would know a performance problem if they saw one. Google's contributions in this area are invaluable. Their Site Reliability Engineering (SRE) discipline's concise, simple, very effective definition of performance objectives is now becoming widespread. These are service-level indicators (SLIs), service-level objectives (SLOs), and service-level agreements (SLAs). The SRE books they've published go into these topics in detail, and the Database Reliability Engineering book does too. Importantly, these metrics can be applied to any component or service, such as a database. My advice is to use the CELT metrics as SLIs to define acceptable performance (SLOs, SLAs) for the database. If there's a database performance problem, the underlying performance theory guarantees you'll see it in these metrics.

Training. None of the above will take root and flourish unless you actively support your team with training. Just because you wrote it doesn't mean anyone will know about it or use it. Effective DevOps teams share knowledge with each other repeatedly through different mechanisms. Lunch-and-learn talks can be a great way to share knowledge. Onboarding training is also important when you bring new engineers into the team. Retrospectives and after-action reviews support learning from incidents. And periodic exercises such as game days, tabletop exercises, and incident response rehearsals are valuable ways to develop and maintain muscle memory in the team

Realigning Software Development Teams

The most effective teams I've seen are structurally aligned to create the desired results. The org design I've seen work best is one in which developers are responsible for operating their own applications in production, they're on call for their own software, and they fix the service when it breaks. This goes by different names: some call it you-build-you-own (YBYO), some call it full-stack teams, some call it full lifecycle ownership. Whatever you call it, the principles and the results are essentially very similar. And I believe—and have seen repeatedly—including the database (and DBA skills/roles/teams) in this is an important step forward.

This is one of the controversial differences of opinion in DevOps. As I referenced near the beginning of this book, some feel the "full-lifecycle team" is DevOops, not DevOps. Others feel it's the best way to do DevOps, and some—like Netflix—just do it without saying whether it's DevOps or not.

Regardless of whether it's DevOps or not, I've seen full-lifecycle ownership work well. It's not only about sharing knowledge in the right places at the right time. It's not only about having the right skills in the right place. It's also about aligning incentives and connecting feedback loops to produce the desired outcomes. As Tom Smith notes in DZone's 2019 DevOps Guide, "The most common reason for DevOps failures are unwillingness to adopt a 'fail fast and iterate approach;' executive level support for the initiative; and *alignment between all of the people responsible for implementing DevOps"* (emphasis mine).

From the knowledge-sharing point of view, full-lifecycle ownership makes a lot of sense. If you want your software to run with high performance and low defect rate, then the people who run it should be the people who build it. Nobody knows better how the software works than the people who built it. In the traditional world where developers developed and operators operated, the developers had no idea how well their software was working in production, so they had no way to know they were writing bugs, and no easy way to fix them when they were told about the bugs. And operators had no idea about the inner workings of the software, so they couldn't understand why it failed or how to make it better, or even give good feedback (write good bug reports) to the developers.

This structure also makes sense from the point of view of creating behaviors through incentives. Nobody is as motivated to make the software work well as the people who get woken up when it doesn't work well. In team after team, I've seen (and been told) when developers are put on-call for their own software, it's painful at first, but within a matter of a few weeks, the late-night on-call alerting problems are solved, and people rarely get woken up anymore—because the engineers fixed the bugs causing reliability problems.

Full-lifecycle ownership also happens to be something that I believe in personally, because my overall life experience has been aligning responsibility and authority is important. When someone is responsible for something over which they have no control or authority, it's a recipe for dysfunction. Operations teams with responsibility for keeping software up and running, but with no authority over what code is put into the software or how it works, will try to assert authority in other ways, such as blocking software from being deployed. This is how you get slow-moving teams with long cycles and low productivity.

This is why Adrian Cockcroft says DevOps is a reorg, not a new team to hire. Speaking from his experience in leadership at Netflix, he emphasizes the importance of developers running their own code, developers being on call, developers having freedom, and developers having incentives to be responsible. The results he observed were less downtime, no meetings, high trust, and a disruptive level of speed and innovation in engineering.

Adrian isn't the only one who has written and spoken about how this works in practice at Netflix. Greg Burrell spoke about it at QCon in 2018, and a companion blog post dives into similar material:

> *"Operate what you build" puts the DevOps principles in action by having the team that develops a system also be responsible for operating and supporting that system. Distributing this responsibility to each development team, rather than externalizing it, creates direct feedback loops and aligns incentives. Teams that feel operational pain are empowered to remediate the pain by changing their system design or code; they are responsible and accountable for both functions. Each development team owns deployment issues, performance bugs, capacity planning, alerting gaps, partner support, and so on… By combining all of these ideas together, we arrived at a model where a development team, equipped with amazing developer productivity tools, is responsible for the full software life cycle.*

It doesn't just work at big companies like Netflix. Mike Bryzek, CTO at Flow and formerly of the Gilt Groupe, advocates for a similar approach and highlights similar successes. "This is an engineering-driven culture, where engineers are responsible full stack for product development, operations, and support post-production," he says. I've gotten a peek at their workflow, and the engineering velocity they achieve with a small team is breathtaking.

Another great case study publicly available for all of us to learn from is Slack, which moved from a centralized operations team towards service ownership during the 2017 – 2019 timeframe (and it's still ongoing). Holly Allen spoke about this, also at QCon. She explained why the "separate but collaborating dev and ops" team model worked well for a while at Slack but didn't scale. They gradually took incremental steps towards more involvement and ownership by devs, with ops still there to help but gradually also shifting away from being the first ones who were paged, towards more of an SRE model and ops availability when needed. The link leads to a page with slides, a video, and a full transcript of the talk. A lot of the problems and solutions they found at each step of the way are discussed.

New Relic also shifted their team structure as they scaled. They called it "Project Upscale," and they've blogged and spoken about it quite a bit. A good jumping-off point is this InfoQ article. When I step back and try to distill takeaways from the stories of Netflix, Slack, and New Relic, I think one of the biggest is not that you have to do this reorg as you scale. Rather, I look at it from the other perspective: when you're not yet at scale, the problems with "collaborating siloed teams" aren't yet evident, and you underestimate how serious they'll become. Only as you scale are you forced to confront them, but in each of those teams I think there's reason to believe they delayed the change past the point when it would have been helpful, because of the cost and effort of change. I know there are difficulties with service ownership in smaller teams: a big lesson I took away from a conversation with Adrian Cockcroft is in a small team, distributing the pager rotation is a bit harder. But I see teams manage it even at smallish sizes, such as Flow, and it's an area I want to learn more about.

I've even seen full-lifecycle team ownership be the reason some companies thrive while others die. Without naming names, one of our customers who has a full-lifecycle team structure, including engineers owning the database, acquired a direct competitor that refused to adopt the same approach. I got to know both companies quite well before this happened, and in my mind, there is no doubt engineering velocity was a significant contributing factor to the one company crowding the other out of the marketplace. (This was one of the success stories I mentioned at the beginning of this book.)

Team structure is so important that Matthew Skelton and Manuel Pais have specifically focused on it in their work and have written a website and a book called *DevOps Team Topologies*. It examines various team structures people have tried and tries to explain the results. It's a valuable learning resource, with many different examples to study, and I believe it corroborates what I've been advocating in this book. One of the anti-patterns is the siloed dev and DBA teams. Quoting the website:

> *Anti-Type G: Dev and DBA Silos… Because these databases are so vital for the business, a dedicated DBA team, often under the Ops umbrella, is responsible for their maintenance, performance tuning and disaster recovery. That is understandable. The problem is when this team becomes a gatekeeper for any and every database change, effectively becoming an obstacle to small and frequent deployments… the DBA team is not involved early in the application development, thus data problems (migrations, performance, etc.) are found late in the delivery cycle… the end result is constant firefighting and mounting pressure to deliver.*

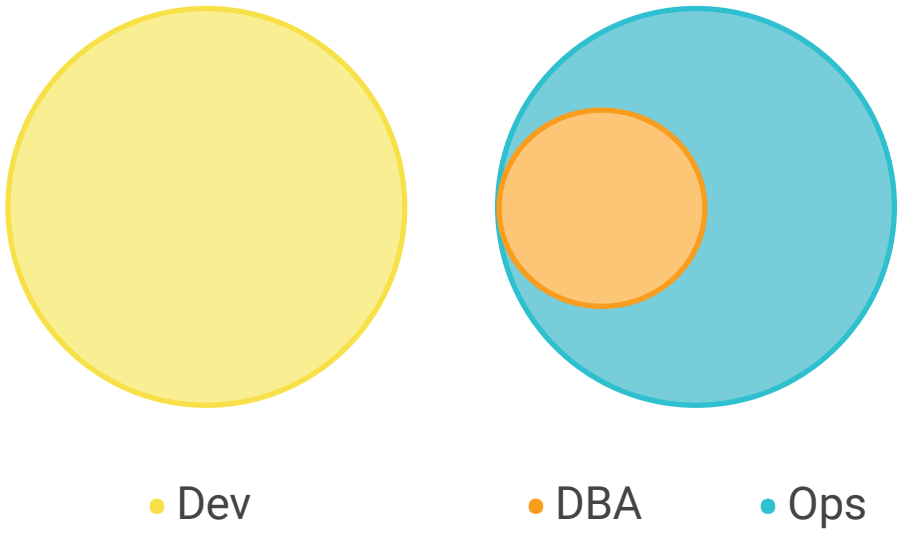


Figure 3. DevOps Anti-Type G, Dev and DBA Silos. Image: devopstopologies.com

What happens to the DBA in a DevOps team? They need to embrace a new role and redefine the value they provide to their organization. Similar to how a system administrator divests themselves of toil and accepts a higher standard of personal performance to become a Site Reliability Engineer, a DBA should evolve into more of a Database Reliability Engineer. As Dr. Nicole Forsgren, PhD. writes,

> *These practices elevate DBAs to valuable consultants, not downstream constraints. This may feel foreign at first, as the role of the DBA expands from building tools to collaborating and consulting across whole teams of people. Where they were once protective and working on technology alone or with only other DBAs, they are now open and consultative with others. Where their work with the database was once the last stop in the chain of application release automation, this work is (hopefully) triggered much earlier in the pipeline.*

And to revisit Silvia Botros's article again,

> *DBAs can no longer be gatekeepers or magicians. A DBA can and should be a source of knowledge and expertise for engineers in an organization. She should help the delivery teams not just deliver features, but to deliver products that scale and empower them to not fear the database.*

The Importance of the Second Age of DevOps

Why is it so important for developers to have responsibility and authority for database performance? Why do teams that do this outperform others so dramatically? In my experience, all this comes down to the core principles of DevOps, and the importance of moving past what Charity Majors calls the first age of DevOps into the second age.

In *The DevOps Handbook*, on the preface on page xvi, one of a long list of "DevOps Myths" they debunk is the myth that "DevOps is just 'infrastructure as code' or automation." The book quotes Christopher Little, who wrote, "DevOps isn't about automation, just as astronomy isn't about telescopes." DevOps requires *shared ownership* of the entire software development and operations lifecycle, however you define that.

I believe the reason for this all comes back to the Eli Goldratt's Theory of Constraints. *The DevOps Handbook* and its immediate predecessor, *The Phoenix Project*, were heavily inspired by Goldratt's classic book *The Goal*, which introduces the theory to explain why constraints, dependencies, and statistical variations in a process flow such as software development influence performance of the overall system so much.

Goldratt's Theory of Constraints, in turn, is based on principles from a field of operations research called queueing theory. Without getting into queueing theory itself, it's enough to say the field of performance optimization has been thoroughly analyzed and developed into a robust and reliable set of mathematical and statistical models to predict performance. And it shows constraints and dependencies in a system of work (such as a software development team) have an almost unbelievable performance cost. Literally: I have built queueing spreadsheets to analyze people's software development performance and shown it to them, and they simply refused to believe how badly their bottlenecks were hurting them.[17]

The real-world experience of teams such as Netflix and Flow, however, lend credence to the predictions of queueing theory. These teams so dramatically outperform traditional software development teams that it is almost unbelievable if you haven't experienced it. Another story I can share is Shopify, where "developers across different teams in the organization are responsible for not only writing new queries, but performance testing them as well. By giving developers access to monitoring tools, Shopify eliminates the need for complicated back-and-forths between devs and DBAs." They credit this as a major factor in their ability to deploy dozens of times daily.

[17] In hindsight, this probably wasn't a very good sales pitch.

But if you believe the predictions of queueing theory, the dramatic performance gains created by eliminating silos, specialization, and bottlenecks are not surprising—they're reasonable and expected. For more on this topic, see Donald G. Reinertsen's book *The Principles of Product Development Flow: Second Generation Lean Product Development*.

My point is this: if you don't move your software development team into the second age of DevOps, and you stop at simply automating database operations and leaving them in the hands of DBAs who act as gatekeepers, I think you're creating a constraint and dependency in your software delivery process. And in my experience, the negative effects on engineering productivity are every bit as bad as queuing theory predicts. Creating a silo around the database prevents your team from getting some of the biggest productivity gains DevOps has to offer.

This insight is not new to DevOps, nor is it solely the province of forward-thinking "hipster" teams. Oracle's Tom Kyte, one of the godfathers of relational databases, wrote about the need to eliminate walls between developers and DBAs in 2003. On page 2 in Chapter 1 of *Effective Oracle By Design*, he wrote,

> *This will be the least technical of all the chapters in this book. Nevertheless, it may be the most important one. Time after time, the mistakes I see being made are more procedural than technical in nature… The tendency to put up walls between developers and DBAs… this is all about human interaction… too often, the relationship between database development team members and the DBA staff members that support them is characterized by a heavy-duty "us versus them" mentality. In such an atmosphere, DBAs often feel a need to protect the database from the developer…*

Unfortunately, however, my experience is few companies take the leap into the second age of DevOps. Most teams stop at simply automating database operations and giving DBAs better tools. In fact, just as the authors pointed out in the preface to *The DevOps Handbook*, the myth of DevOps being just automation is a strong one and is proving hard to overcome and move beyond.

I did an informal survey (N=57) by asking people on a DevOps newsletter, and people who follow me on Twitter, to fill out a few questions about what they think about DevOps as it relates to the database. I asked how important they thought it was for DevOps teams to adopt specific practices. All questions were optional, so answers don't always sum to 57.

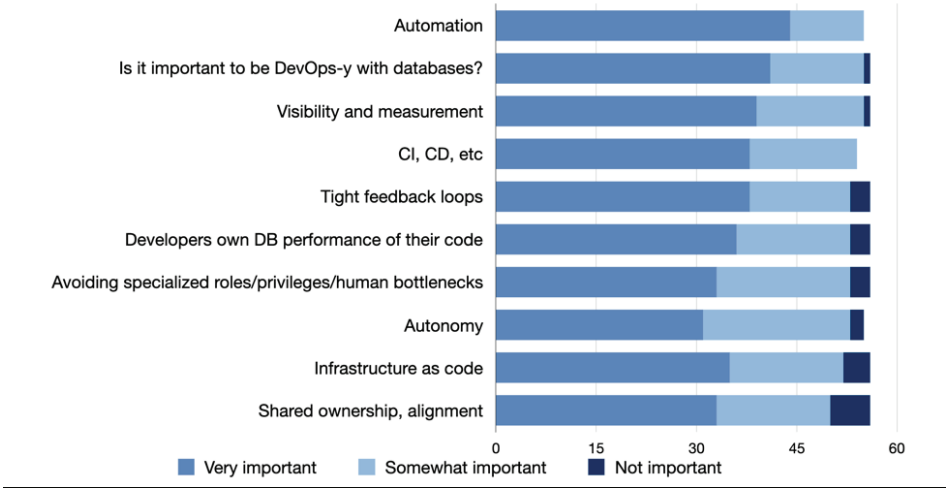


Figure 4. Survey responses to "How important are each of the following in your view of 'Database DevOps'?"

Notice how many people think bread-and-butter practices like automation and CI/CD are very important, and think DevOps is very important for the database. But they place a little less importance on the "second age of DevOps" practices: developers owning database performance of their code, avoiding specialized roles and privileges such as those a DBA typically has, and shared ownership and alignment. This isn't a scientific survey, but this tells me at least some people feel DevOps can stop at DBAs automating their work and doesn't require handing over ownership of database performance to developers. Yet my experience, and many of the dozens of books, articles, podcasts, and conference talks I've cited in this book, suggest otherwise.

I hope this book helps change this viewpoint, because it's one of the major reasons I was motivated to write it. Again, I'm unabashedly passionate about this topic. It makes such a difference in the lives of humans and the performance of businesses.

Your Journey Towards Database DevOps

If you're still reading, hopefully by this point you're sold on the concept of taking a DevOps approach to the database. You know what the destination looks like, and you're excited about the benefits of applying the methodology. But how do you actually *do* it?

I wish I could tell you there's a magic recipe, simple steps you can just follow, and success will be assured. Unfortunately, the reality is progressing from a less-DevOps culture to a more-DevOps culture can be tricky, and requires skill and care, lest you do more harm than good.

However, it's doable. Delightfully, one of the biggest factors in success is applying an iterative, incremental, easy-does-it approach towards adopting DevOps itself. It's much more of a process than a project, much more of a journey than a destination. You never arrive, but you can take steps every day by simply putting one metaphorical foot in front of the other. I cannot give you details on each step, but I can give you some guidance, more akin to a list of landmarks than a roadmap.

Getting Started. First, I think it's important to reemphasize you should seek to do no harm. Just as a doctor's oath is to care for their patients, so you should resolve to keep the welfare of your team in mind.[18]

Next, you need to pick a place to get started. Don't make the mistake of trying to change everything all at once. Start small, in a place where success will be easier, and you can achieve the credibility and buy-in you need to expand from there. As Brian Dawson advises in *6 Keys to Unlock the Value of DevOps*, "you need to make sure it works on a microscale, ensuring there is a bottom-up adoption and buy-in from key managers. Management buy-in can be difficult and time-consuming at the start of an implementation, but it will be required to successfully achieve an organization-wide transformation. It's going to affect timelines, budgets, and organization." This is true if you're trying to introduce DevOps to an organization where it's not practiced at all, and it's also true if you have a team who's deeply invested in DevOps—but doesn't want to apply it to the database.

[18] If you feel you can't do this without sacrificing your own well-being, perhaps this team is toxic and doesn't deserve you.

Where's the best place to start? It depends on where you are now.

- If you're just beginning, pick a tiny, basic thing near the beginning of my ordered list of six categories of DevOps progression early in this book. For example, get the database's configuration into your infra-as-code tool, like Chef or Puppet. Get repeatability in provisioning and configuration, in other words.
- If you're further along, try to get migrations into deployment automation. At SolarWinds, for example, I think a good next step for us would be to move from manually triggered migrations to automated and move those from SRE into engineering.
- If you have all those things down to a science, experiment with things like who's on call, or move from one org structure to another. We currently have front end, back end, security, and SRE teams at SolarWinds, mostly aligned around the layer of the architecture and the technologies each layer uses. I'd really like to experiment with full-lifecycle ownership a bit more. But only after getting the back-end CI/CD pipeline upgraded to a similar level of automation in continuous delivery our front-end web app currently has.

It's a lot easier to start with a new project, where you don't risk disrupting something that's working. Rewriting a legacy system or changing a legacy process is always riskier than starting with something greenfield the business doesn't yet depend upon. And remember my cautions from earlier: don't try to do too much too fast, don't push for velocity at the expense of stability and safety.

Agile Approach. Try to work in small, agile, lean iterations. The process will involve some extra work, both technical and otherwise, and you need to make sure it's not hidden. Make all the work visible by including it as tasks in your normal development backlog. That way, it gets prioritized and staged appropriately, and everyone sees what's happening.

Try to take a phased approach to everything, working on a small number of priorities at a time. This includes not only technical work, but also things like experimenting with your team structure or topology or changing a process. For example, perhaps for one sprint you can experiment with changing who's on-call or experiment with a different peer review and approval process for database changes.

At the end of the sprint, hold a retrospective, and share what you learned with the team. Be sure to focus on both successes and failures, so you can repeat the successes and broadcast them widely, while messaging that you have a plan to change what didn't work as well and try it again a different way.

Culture Change. DevOps is not only a reorg, it's a culture change. The difficulty with culture changes is culture is *emergent* from underlying structures, processes, and motivations. Culture is the outcome of what's easy or hard, what's rewarded or punished, and the resulting *behaviors* people exhibit. As Will Gallego wrote, "It's hard to convince people to eagerly dive into work that gets less praise, more toil, and generally underappreciated."

That's why you can't change culture directly. You can only change the *inputs* into the system, such as incentives and rewards, and observe carefully how culture—the *output*—changes in response.

One of the most important ways you can change incentives and encourage a different way of doing things is by making the right behavior easy. Create a new path of least resistance that matches the desired behaviors. For example, Netflix wrote about the "paved road":

> *Tooling and automation help to scale expertise, but no tool will solve every problem in the developer productivity and operations space. Netflix has a "paved road" set of tools and practices that are formally supported by centralized teams. We don't mandate adoption of those paved roads but encourage adoption by ensuring that development and operations using those technologies is a far better experience than not using them.*

Another key to culture change is adopting DevOps practices itself changes and improves culture. *Accelerate* has an entire chapter on measuring and changing culture. And these themes are woven throughout the book. They write on page 6 about how focusing on capability models (not maturity models) engenders good culture; on page 39 they discuss the power of changing *what people do and how they act* as an input into culture; again on pages 105 – 114; and there's a significant focus on understanding what culture really is and how leadership plays a role in it.

Speaking of changing culture by changing what people do, I've seen the power of this many times in action. What people *do* influences things more readily than most other levers at your disposal. And many experienced leaders I respect have told me variations of "working on the culture is fruitless; work on behavior and actions." Ryn Daniels writes about culture change using *designable surfaces* in *The Design of Engineering Culture*:

> *There's more to culture than values, and this becomes apparent when looking at how different organizations that espouse the same values in theory end up having different cultures in practice… A designable surface is something concrete that can be directly changed in order to impact one or more aspects of culture… Changing designable surfaces in an organization, monitoring the impacts of those changes, and iterating toward goals is how to build lasting culture change… The heart of culture design is understanding how collections of designable surfaces can be changed in concert to achieve specific outcomes. The most successful transformations tend to be those that involve multiple designable surfaces working together.*

On a more personal note, two other nonscientific things I've observed over my career are worth mentioning:

- You can't fix a bad culture by hiring good people. Putting good people into a broken culture breaks the people, it doesn't fix the culture. (I think this might be a Jez Humble quote, but I'm not sure.)
- A lot of influence comes from small behavior changes. Let it begin with you, personally. Train yourself: learn and model good behaviors. Many surprising results come from things that might not occur to you.[19] Open yourself to these possibilities by deliberately listening and using the many resources available to help you discover and address your own opportunities for personal growth. One of my favorites is https://www.jennifer.kim/inclusion.

Make Success Attractive. Getting people to change is really a sales job. It's about persuading people something is in their individual and collective interests, getting them to say yes to trying, and then helping them feel good about the results. If the results aren't a complete success, you can still help them feel good about what went right and feel hopeful about what hasn't yet.

A crash course in sales, even at a summary level, is well beyond the scope of this book. But in a nutshell, all sales is about presenting a desired target state, and convincing people not only is the target state better, it will be enough better to justify the pain of disrupting the status quo.

[19]Random example: the power of not inviting a man to speak first in a meeting composed mostly of men. Did you know about that one? If not, a thousand more surprises lie in store for you. Cultivate curiosity!

There's a bit of psychology involved in this, and even small things such as getting people to say the word "yes" can help. It should never rise to the level of manipulation, but it's good to be aware of the power of the words you choose. For example, you can ask a question, phrased in a positive way, designed to elicit a yes as a response. If something doesn't work the first time, you might ask, "We learned this hasn't worked yet, right?" By getting someone to respond with yes to a question containing the word "yet," you're helping put them into a positive, can-do, hopeful frame of mind. Another great technique when selling change is to emphasize what's not changing, to help reassure people they'll be able to anchor themselves to familiar things. Change is stressful for humans—all humans, even the ones who say they like change!

Tanya Reilly's great article *I Wouldn't Start From Here: How to Make a Big Technical Change* contains lots of pearls of wisdom along these lines. As she notes, "there's a ton of work involved in making people change their minds."

Three Steps on the Journey

Where are you right now in your journey to applying DevOps practices to your database? Most companies I meet are still in the traditional world where databases get individual care-and-feeding from DBAs, and developers aren't comfortable or familiar with the database. I've seen team after team go through the following three-phase high-level process:

1. **Put Out the Fires**. Pick the worst problem and swarm it to eliminate it. I agree with Dickerson's Hierarchy of Service Health: the lack of monitoring and observability is often the biggest problem. I've seen so many teams gain visibility into what their database is doing, which always produces good surprises, and then the team focuses on fixing those. And for the team to make any changes—even "improvements"—with confidence, they need monitoring and observability. Perhaps you prioritize something different, though: maybe you think the biggest problem is the lack of configuration-as-code, deployment automation, or schema-as-code migrations. At the risk of repeating myself too many times, I still think you need monitoring in place first, before you start making these types of changes. Whatever it is, focus enough effort on it to produce wins that you can trumpet to the rest of the team.

2. **Secure Your Wins**. After the ashes have cooled a bit, use the political capital you've earned with all those improvements. Spend it on investing in sustaining your newfound, higher level of performance. Don't move on to the next thing and let technical debt and problems creep back in; don't invite a slide back into renewed firefighting.

3. **From Reactive to Proactive**. A small team is probably responsible for the initial wins, and perhaps a slightly larger team participated in shoring up things to prevent the problems from returning. Now is your chance to take the next step: whatever you did in the first two steps, take the laurels and place them onto the heads of a larger team. From my vantage point as a spectator of observability gains, I often see a DBA team bring in observability tooling, fix performance, stability problems, and bugs, and then turn around and sell (and push—let's be honest) the broader development team to own database performance, now that they have tooling that makes it possible for them to do so. By doing this, they complete a transition from First Age DevOps to Second Age DevOps, and often transform their entire software delivery processes into full-lifecycle ownership by engineering teams.

This takes commitment to pull off, but every time I've seen it happen, the results have been more than worth the investment. Good luck, and let me know how it goes!

Acknowledgments

This book isn't really my work, and I'm not just "being humble." I literally assembled this book from watching and asking others what works and what doesn't. I read a lot of blog posts and books, watched a lot of videos from conference talks, and talked to a lot of customers. There's not much original material here; my contribution is just synthesizing other people's wisdom. A bunch of this came from people who responded to me on Twitter when I asked for material I could study. Even the realization that DevOps is a good name for this "do databases better" phenomenon I was seeing wasn't my own: I think Jessica Kerr inspired me to make that connection when she invited me to present at QCon and we had a call to discuss topics.

So, many thanks to all the people who wrote this book with me. In particular: Jessica Kerr, Silvia Botros, Charity Majors, all our amazing customers at SolarWinds, the powerhouse trio behind DORA (Dr. Nicole Forsgren, Jez Humble, Gene Kim), every author of every book I cited, and the former VividCortex, now SolarWinds team. I know I'm omitting many, but that's inevitable.

My thanks also to the generous reviewers who volunteered to read the first draft. They brought a perspective to the work that was impossible for me to see, and especially helped improve the overall tone and feeling, making this book more pleasant to read which will help it appeal to a wider audience. They also suggested a greater breadth of material, technical resources and tooling, and reference literature than I was aware of. Many of the best parts of this book are due to their insightful comments. Not all the reviewers told me they wanted to be named, but those who did are Devdas Bhagat, Silvia Botros, Steve Fenton, Dimitri Fontaine, Dave Poole, Pierre Vincent, and KimSia, Sim, as well as several of the former VividCortex, now SolarWinds team members. I'm grateful to you all, named and anonymous.

Free Electronic Version

We hope you enjoyed the first edition of DevOps for the Database and have found many actionable insights to help guide your DevOps journey.

If you borrowed this copy from a colleague and would like to get a complimentary PDF version for yourself, please use the links below. Thanks!

Download the FREE PDF ebook at:

www.solarwinds.com/resources/devops-for-the-database-ebook

ABOUT SOLARWINDS

SolarWinds (NYSE:SWI) is a leading provider of powerful and affordable IT infrastructure management software. Our products give organizations worldwide, regardless of type, size, or IT infrastructure complexity, the power to monitor and manage the performance of their IT environments, whether on-prem, in the cloud, or in hybrid models. We continuously engage with all types of technology professionals—IT operations professionals, DevOps professionals, and managed service providers (MSPs)—to understand the challenges they face maintaining high-performing and highly available IT infrastructures. The insights we gain from engaging with them, in places like our THWACK online community, allow us to build products that solve well-understood IT management challenges in ways that technology professionals want them solved. This focus on the user and commitment to excellence in end-to-end hybrid IT performance management has established SolarWinds as a worldwide leader in network management software and MSP solutions. Learn more today at www.solarwinds.com.

solarwinds

For additional information, please contact SolarWinds at 866.530.8100 or email sales@solarwinds.com.
To locate an international reseller near you, visit http://www.solarwinds.com/partners/reseller_locator.aspx