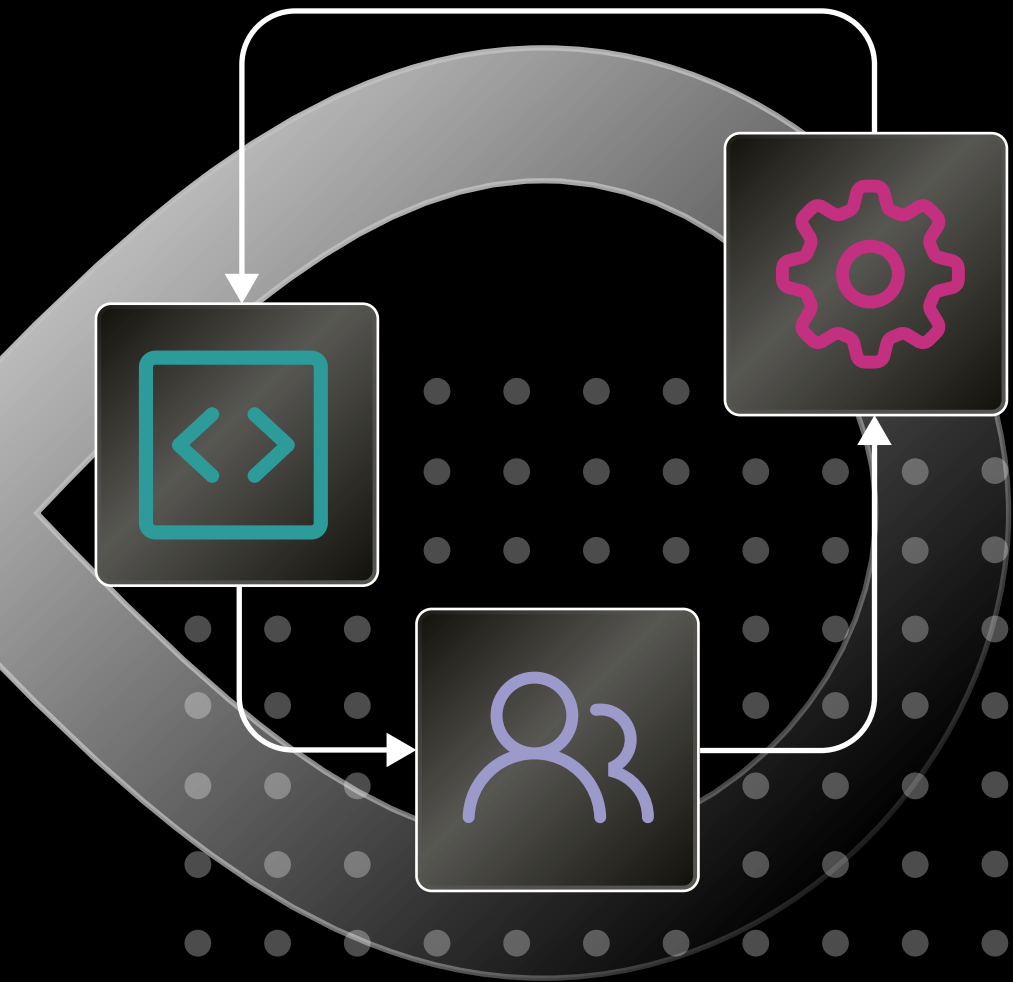
Essential DevOps

by Ian Griffiths and Brian A. Randell



Essential DevOps

COPYRIGHT

Essential DevOps

by Ian Griffiths and Brian A. Randell

Copyright © 2023 GitHub. All rights reserved.

Printed in the United States of America.
Published by GitHub, Inc. 88 Colin P Kelly Jr St.
San Francisco, CA 94107.

Layout and design by Paul Waller

Published: August 2023

Ian Griffiths (idg10)
Technical Fellow
endjin.com

Brian Randell (octobrian)
Staff Developer Advocate
GitHub

Paul Waller (PaulWaller)
Graphic Designer and Illustrator
endjin.com

Foreword **vi**
Acknowledgements **viii**
Preface **1**
01. The Psychology of DevOps **7**
The Evolution of DevOps 9
The Nature of Software Development 15
Feedback Loops 22
Some Popular DevOps Anti-patterns 25
02. Ways of Thinking about DevOps **29**
DevOps 31
DevSecOps 47
SRE 53
OtherOps 55
03. Planning and Tracking Progress **61**
Planning Horizons 64
Responsibility and Decision Making 72
Tracking Progress 80
DevOps and Popular Agile Methodologies 86
04. Security **89**
Ways of Thinking about Security 92
Security and DevOps Organizational Culture 105
Tools, Tactics, and Techniques 109
05. Ensuring Quality **127**
Quality Assurance and Organizational Culture 130
Assessment Attitudes 136
Speed and Convenience vs Realism 154
06. Managing Data and Analytics **167**
Online Application Databases 170
DevOps, Data Science, AI, and Analytically Led Processes 188
Other Store Types 199
Data Governance 201

07. Continuous Integration, Deployment, and Delivery 203
Types of Automated Process 206
Automation, Delivery, and Planning 220
Environments 224
08. Automated Infrastructure Deployment 233
The Disposable Infrastructure Mindset 236
Infrastructure as Code 238
Building Confidence 250
09. Monitoring 253
Monitoring Goals 256
Service Level Indicators, Objectives, and Agreements 263
Information Sources 265
Analytics: from Measurements to Insights 269
Response 273
10. Observability 279
Kinds of Observers 282
Observability Data 286
Information Sources 291
Balancing Privacy with Capability 298
Analysis and Response 299
11. The Last Word 301
What does success look like? 302
Feedback Loops 307
What's Next 309
About GitHub Let's build from here 310

Foreword

The past few years have made it clear: we live in a world now fully eaten by software. From the moment your alarm goes off, to your commute and your day at work, to the second you set your alarm before you sleep, you cannot escape using software. And building, running, and maintaining all this software for the world has never been more complex. Now more than ever, it is necessary that we equip developers with everything they need both to build the world's software and for the sake of their own happiness.

"Accelerate human progress"

Throughout the advance of software, the practice of DevOps has continued to grow and evolve to help developers build their best in this wave of digital transformation. DevOps served as a paradigm shift, moving us from traditional siloed roles towards a holistic, collaborative, and agile approach. It created the art of testing, deploying, and supporting software. Today, at GitHub, we envision a future where DevOps evolves into one, integrated solution powered by AI at every step. This makes this book all the more important in setting and contextualizing potential guardrails so developers can build with AI to their fullest potential.

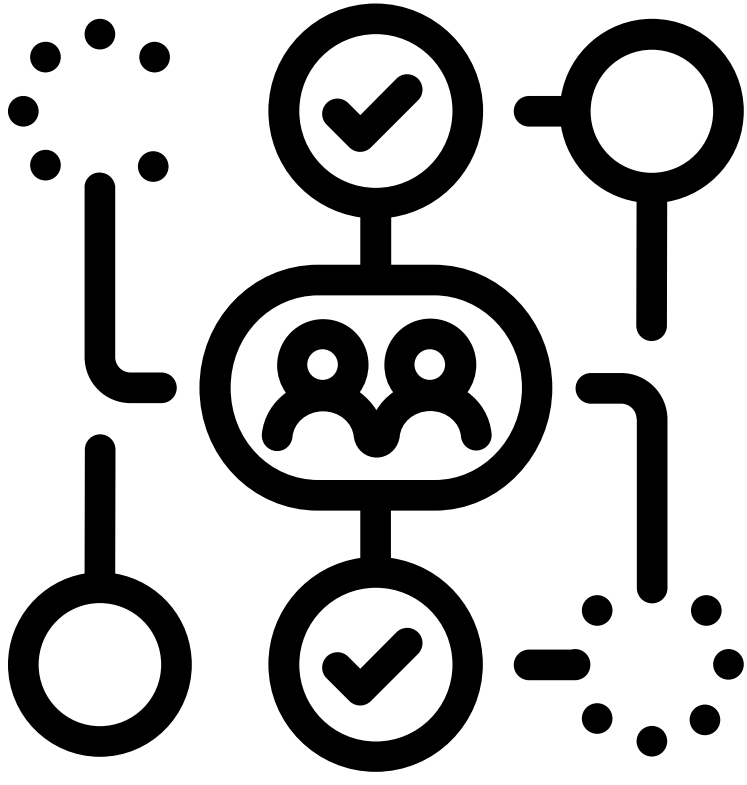
“We envision a future where DevOps evolves into one, integrated solution powered by AI at every step”

In “Essential DevOps”, industry veterans Ian Griffiths and Brian A. Randell embarked on the formidable task of boiling down a simple analysis of the complex evolution of DevOps. The authors draw on a reservoir of experience, providing insights, practical tips, and comprehensive guidance, making this book an invaluable resource for both beginners and seasoned professionals. It is critical that we learn from what has made this practice a successful staple of software development, so we can continue to evolve DevOps in a way that empowers every developer on this planet to accelerate human progress for us all.

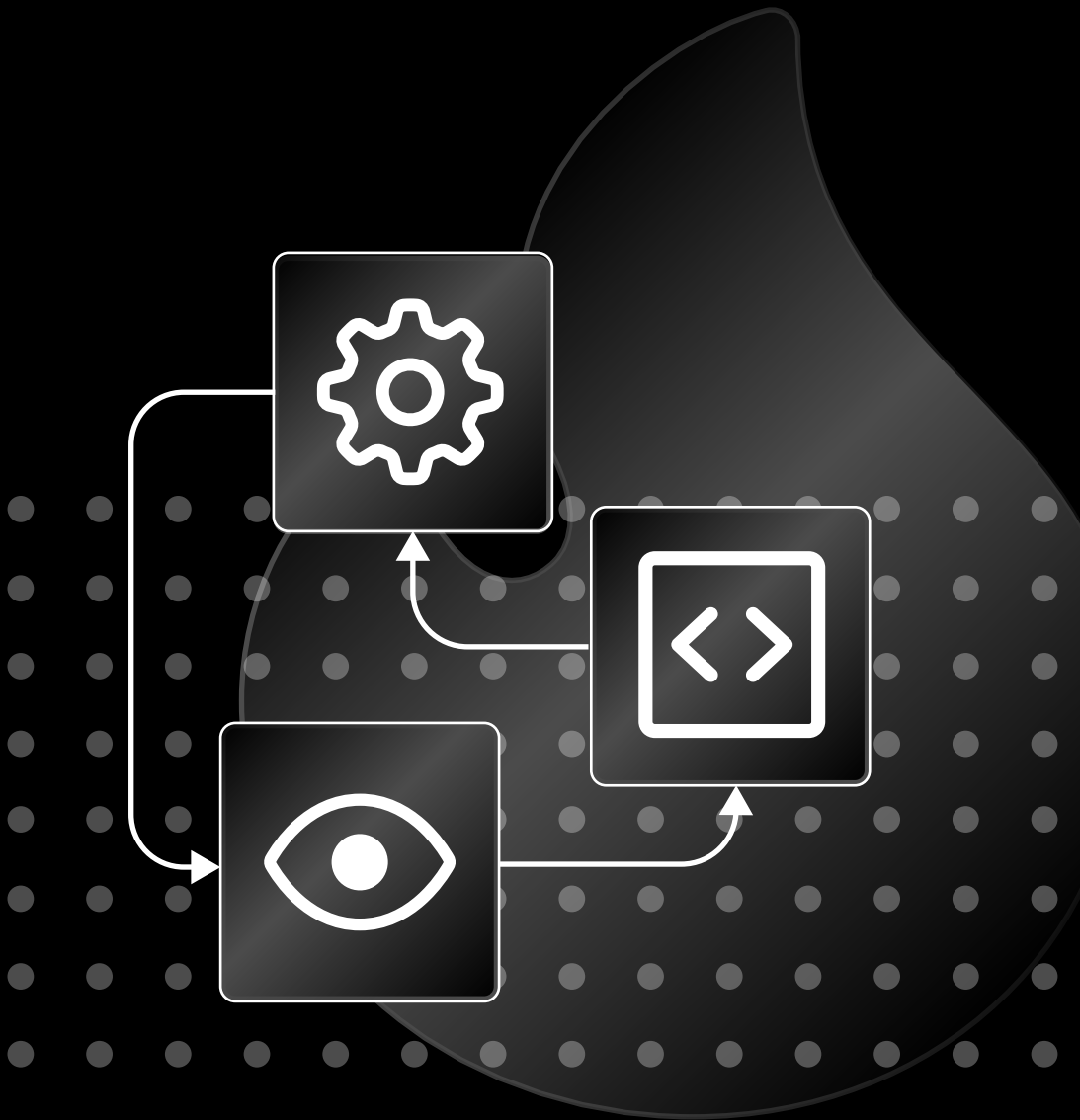
Thomas Dohmke
CEO, GitHub

Acknowledgements

This book is the product of over 100 years of collective experience in building technology and applying DevOps principles and practices. The authors would like to thank everyone at endjin who contributed to this book: Matthew Adams, James Broome, James Dawson, Ed Freeman, Elisenda Gascon, Jonathan George, Jessica Hill, Mike Larah, Liam Mooney, Barry Smart, Howard van Roojien, and Paul Waller. Many thanks also to the Hubbers whose feedback helped to shape this book: Mickey Gousset, Keith Hoodlet, Francisco Lee, Erdem Menges, Mona Lisa Octocat, Brittany O'Shea, Mark Paulsen, Jon Peck, Chris Reddington, Aaron Winston, and Martin Woodward.



Preface



“DevOps is a way of applying critical thinking to the business of delivering value through software development”

Software development is a complex endeavor. It requires a wide range of demanding technical skills, but many organizations underestimate the social challenges. Teams that develop and deliver software need a deep understanding of their stakeholders’ needs if they are to build useful systems. It’s all too common for projects to burn a lot of money building complex systems that provide little value. Even useful projects often frustrate users with problems such as slow, infrequent feature delivery, bugs, performance problems, and security issues.

There is no panacea—there is no single organizational blueprint or set of technical tools that will guarantee successful, efficient delivery of useful, high-quality work. However, in recent years, many organizations have found that by adopting a certain mindset and making corresponding changes to organizational structure and working practices, they have dramatically increased the value they get from their development work. This approach has come to be called “DevOps.”

DevOps is a way of applying critical thinking to the business of delivering value through software development. As the name suggests, DevOps means bringing together the work of developing

software, and the operational work of provisioning and supporting the systems that run that software. It also addresses cross-cutting concerns, including security. (Some people use the name DevSecOps to include that explicitly, although since security is critical, we believe that DevOps has to imply DevSecOps.) Instead of the common disjointed approach in which a development team writes a feature and then "throws it over the wall" into testing, who then pass it onto operations to deploy and support, we have cross-functional teams that own the whole process. Teams now have collective responsibility for delivering features of value to the users.

When organized in this way, technology teams can provide features to customers earlier, more often, and with higher value than is possible if Development, QA, InfoSec and IT Operations are split across separate teams working in silos. DevOps lowers friction across the value stream from development through to delivery. By minimizing the disruption required to deliver value, it improves an organization's productivity. And by lowering the time between concept and delivery, development teams learn more quickly which ideas work best in practice, enabling them to maximize the value of their work.

With DevOps, we think of work in terms of "value streams." Instead of individuals focusing on the technical artifacts they produce, they should be thinking about the results that end users will be consuming, and the processes required to get to that point. Figure 1 shows a simplified representation of what a value stream might look like. When groups are split into silos based on technical specialization, a transactional attitude can prevail: performance is evaluated in terms of a team's technical outputs

and the overall point of the work gets lost. By focusing on the value stream, we remain mindful of why we're doing the work, and how each task fits into the bigger picture.

Figure 1: A value stream (simplified)

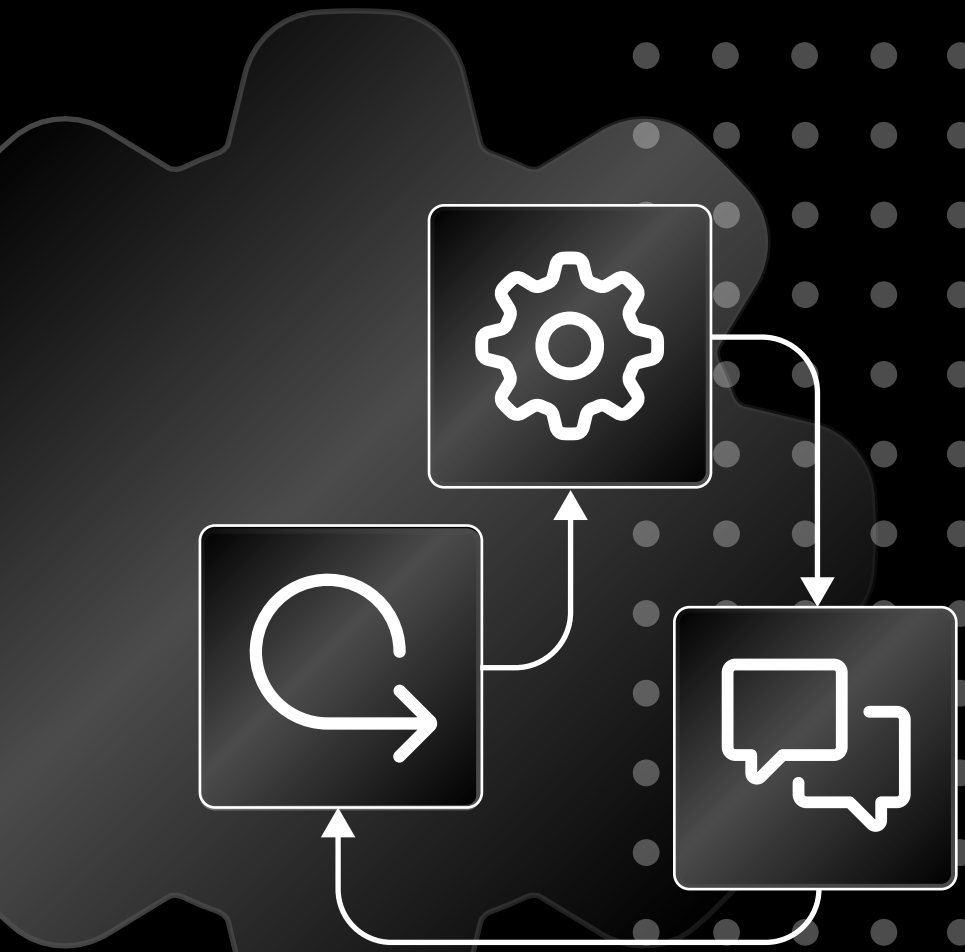
One of the most powerful features of DevOps is that it supports an experimental mindset. Feedback loops are essential for ensuring quality in any production system, but they also have a critical role to play if we want to innovate. An idea for a new software feature might be conceptually brilliant, but it is extremely

rare for the first implementation to hit the spot. The implications of design choices might only become apparent when you first try to use the resulting work, so we need to be able to try several ideas, iterating in response to user feedback. If the time between having an idea and getting it in front of a user is measured in months, or even weeks, it will take forever to arrive at the best feature design. Applying DevOps practices can lower lead times dramatically, making it possible to try out several feature iterations in a week. This provides great freedom to experiment, enabling you to narrow in rapidly on the best approach for innovative features.

As we'll describe in the following chapters, there are various important techniques that make this cross-functional, fast-iterating approach viable. Collectively, they improve the rate at which work flows through the value stream, and they also reduce fear of change. All too often, teams are afraid of deployments because they expect things to break. This leads to less frequent deployments, which increases the chances of failure, creating a vicious circle. A great deal of what makes DevOps work is a set of practices that reduce the risks and fears associated with change, as well as improving the flow. But before we get into the specific techniques, we will look at the thinking behind DevOps.

Chapter 01

The Psychology of DevOps



“DevOps emphasizes the importance of evidence and feedback”

Although there are many techniques and practices widely used by teams that have successfully adopted DevOps, it is important to understand that DevOps is not about tools or processes. It is a collection of principles, and ways of thinking about the problems of delivering value through software.

DevOps emphasizes the importance of evidence and feedback, and it considers the sociological and psychological aspects of creating and supporting software. Our exploration of DevOps therefore begins by looking at the thinking behind it.

The Evolution of DevOps

To understand DevOps, it is helpful to look at changes in the computing industry that influenced it. “The DevOps Handbook” (by Gene Kim, Jez Humble, Patrick Debois, John Willis, and Nicole Forsgren, published by IT Revolution Press) acknowledges that DevOps builds on many of the ideas that came out of the Agile movement. The rise of cloud computing helped to shape many of the practices that make DevOps work. But it’s not just the arrival of new technologies, tools, or methodologies: DevOps is also a response to a relentless increase in complexity for software projects.

Increasing Complexity

Software projects have gotten ever more ambitious. This is partly due to basic improvements in available technology: many of the kinds of systems commonly built today would have been unaffordable to most organizations 15 years ago, and impossible another 15 years before that. It is also driven by competitive pressure. Organizations invest in software development to gain a competitive advantage (or at least to avoid falling behind) leading to an unending demand that our software systems do more and more. Changes in laws and regulations can often increase the demands made on systems over time. Tools and practices advance, continually expanding the ability of any single team to exploit the latest technology, and so with both strong demand and improving capability, complexity tends to increase.

Wholesale digitalization is driving scale. Modern businesses rely on technology more than ever, with IT getting involved in almost everything that companies do. The number of digital assets in an

organization only seems to grow, and this also tends to increase overall complexity.

In the early days of personal computers, it was not unusual for lone developers to achieve heroic feats, with entire software businesses launched off the efforts of one or two people. Many organizations have started out with small development teams achieving big things—it's common for a young organization to have a single team owning everything from conceptual design through to operations. But after early success, it can all fall apart as the demands of the business start to outgrow what one team can do. Expansion follows, with multiple teams becoming inevitable, but the way in which the work is divided has proven to be critically important. One of the primary inspirations for DevOps was seeing how much damage organizational siloes can cause—separating individuals and teams by technical specialization often ends up creating internal conflict. DevOps establishes various principles which, if followed, can enable organizations to rise to the challenge of increasing complexity without falling apart.

Agile Principles

The Agile software movement created its manifesto in 2001. It identified certain principles and priorities, and it has had a strong influence on the development of DevOps. Take its first principle:

> Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

"There is a view that DevOps is what you get when Agile meets the cloud"

This is arguably the main point of DevOps. And if you read through the 12 principles at https://agilemanifesto.org/principles.html, there are several that apply directly to DevOps: welcoming change, frequent delivery, the importance of collaboration, and the development of a constructively critical reflective culture. Early advocates of DevOps have explicitly acknowledged the connections to the Agile movement, and many DevOps adopters today consider DevOps to be its natural extension—there is a view that DevOps is what you get when Agile meets the cloud.

Cloud Computing and Infrastructure as Code

One of the most significant changes to the information technology landscape over the last 20 years has been the rise of cloud platforms. If you wanted to host a web site on the internet in the pre-cloud days, you had to buy the servers, find somewhere to put them that offered reliable power supply, cooling, and network connectivity, install the operating system and all necessary supporting software, devise procedures by which such software would be kept up to date, and acquire suitable networking hardware such as switches, firewalls, and load balancers. Over time, hosting providers emerged enabling you to rent servers, and now, thanks to today's cloud platforms, you can, on a whim, provision a new fully managed environment capable of hosting a web application and be up and running in minutes.

This shift has dramatic business and technical implications. The capital investment that used to be required to set up a web server typically entailed a detailed business plan projecting the ROI that would justify the outlay, followed by a potentially lengthy procurement process. This is not conducive to experimentation—in that world, setting up a web server merely to find out whether your idea is any good was a tough sell.

The most obvious changes with the cloud are reduced risk and increased speed. The risk is lower because we don't need to spend a fortune (or convince someone else to host our workload on their very expensive server) just to try something out. Competition amongst cloud platform providers means that we can now provision new infrastructure in seconds. This is so straightforward that it's easy to take it for granted, forgetting what a remarkable technical feat this is. These changes directly support DevOps practices: an experimental approach involving repeated iterations with a feedback loop are a lot easier if the risks and lead times are low.

There is a more subtle technical shift associated with the rise of cloud computing (although to be precise, it's not absolutely necessary to be in the cloud to enjoy this). To be able to provision infrastructure in seconds, cloud providers have invested heavily in virtualization of the services they offer. When you click a button asking for a new server, nobody unboxes a new computer, installs it in a rack, plugs in all the necessary cables and performs software installation or configuration procedures. Invariably, what you actually get is some share of hardware that was already there. This is true not just for servers, but also networking: you can provision new networks, network interfaces, gateways,

firewalls and the like, but no new hardware is added when you do this. Cloud providers have devised flexible architectures that enable network topologies to be changed and virtual devices to be created without requiring any physical modifications—changes which a few years ago would have meant walking into a datacenter to install physical devices and connect them with cables.

Cloud providers had to virtualize these facilities to be able to offer competitively priced support for a wide range of workloads while remaining profitable. Dedicating specific hardware to a customer tends to be expensive, and so virtualization is a necessity. But a happy upshot of this is that it offers amazing flexibility to us as customers of the cloud. Back in the days when a network firewall was a physical device with its own power supply and network cables, provisioning a new firewall was a lengthy process for all the same reasons that provisioning a new web server once was. But if network devices are now all virtual entities, we can create them just as quickly and easily as the cloud lets us provision a server.

The practical upshot of this is that our infrastructure is now as malleable as our code. In effect, it is code, often sitting alongside the source code that defines the system behavior. The topology of our services—the particular servers we have, the networks that connect them, the gateways that present them to the outside world, and so on—is defined in one or more configuration files defining the desired state that we push to the cloud provider. Since they're just files in source control, they share an important characteristic with the other source code: a change in the

network topology now requires nothing more than a commit to source control followed by a deployment.

This is often described as *infrastructure as code* (IaC), and it is radically different from physical infrastructure, where changes entail moving actual boxes and cables. With great power, comes great responsibility—if your infrastructure can be altered by code, then we need to ensure that changes don't break functionality, and that all non-functional requirements such as security are maintained. As you will see in the chapters that follow, some common DevOps practices exploit the flexibility of infrastructure as code to great effect, while ensuring that this power is exercised responsibly.

Although DevOps has been influenced in many ways by these industry-wide changes, there are elements that address more timeless aspects of software development.

The Nature of Software Development

Respect is an essential element of DevOps. It is rooted in understanding and appreciating all the people involved in the process of delivering software, and the special skills required throughout the software delivery lifecycle. It is important to understand that software development is not just a technical activity: it has significant social aspects too. (In other words, it is a sociotechnical activity.) DevOps principles and practices take this into account.

"Respect is an essential element of DevOps"

Organizational Implications

The social dimension of software delivery motivates DevOps' organizational principles. We have cross-functional teams because it enables collective responsibility for providing value to the end user. Compare this with the common (non-DevOps) approach of grouping by technical specialization. If a team's purpose is defined in terms of "security" or "database administration" it becomes much harder for the team members to understand the connection between their work, and the value that work might create. When a team is focused on creating a particular outcome for a user, everyone understands the purpose of their work in terms of the value it will create.

Team Sizes

The nature of human interactions has implications for team sizes. The social dynamics of a group of 10 people are very different from those of a group of 100. This is not news. In 1975, Fred Brooks wrote a book called "The Mythical Man Month: Essays on Software Engineering" (published by Addison Wesley). This book's continued relevance is astonishing given how early in the history of commercially produced computer systems it was written. It has had a couple of updates (in 1982 and 1995) but it's surprising how little needed to change. For example, half a century later, its advice on team sizes still looks relevant.

There is a fundamental reason for this. As teams get larger, communication becomes more difficult because the number of lines of communication grows dramatically. Adding three people to a team of 6 more than doubles the number of relationships, as Figure 1 illustrates.

The mathematics behind this aren't changing, and nor is the innate human capacity for comprehending human interactions, which is why advice from 1975 remains relevant today. (Perhaps not in the precise details—we might question the continued relevance of Brooks' assertion that a lead programmer still needs two secretaries to handle correspondence and document filing. But certainly, on the matter of scale.) Small teams typically achieve a level of per-person productivity that is higher than that of a larger team. What is more remarkable is that Fred Brooks had found ample evidence that making a team larger might not just reduce the per-person productivity, it might reduce the team's aggregate productivity. You can hire more team members only to find that your team now produces less work.

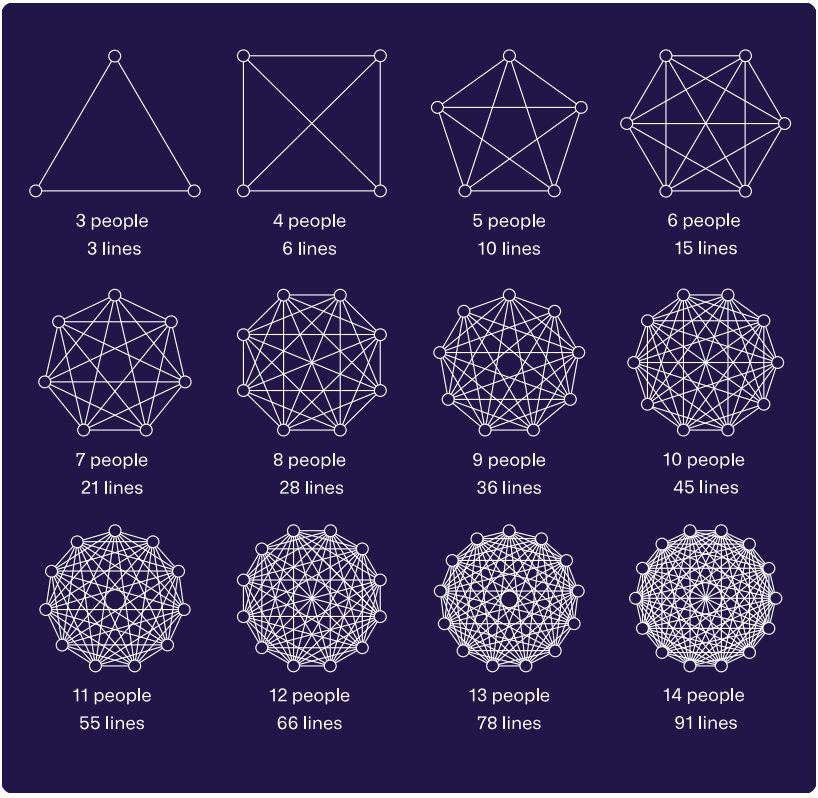


Figure 1: Team size and lines of communication

This suggests that the best approach is to have small teams, each aligned with one value stream, and optimized for flow of value. We should minimize the need for hand offs between teams and carefully manage the cognitive load placed on workers. This can all help reduce, friction, miscommunication, and mistakes. To increase capacity, it will often be better to have more teams than to enlarge existing ones. From a DevOps perspective, this will enable work to flow more quickly—smaller teams are typically able to have a lower *cycle time*, the time between having an idea and delivering an implementation of that idea. This provides shorter

feedback loops, which, as later chapters will discuss, are fundamental to improving the value that development teams deliver.

It is not always this simple. The network effects shown in Figure 1 apply to teams just as much as individuals. An organization with 10 teams will find it much harder to coordinate work than one with 5 teams. If there is poor communication between the teams involved, the quality of the results is likely to reflect that. There are no straightforward solutions to this, because as Conway's law (https://en.wikipedia.org/wiki/Conway%27s_law) tells us, companies tend to create systems or products that reflect their organizational structure. From a software development perspective, this means that the way we divide work into teams will also determine our system architecture. This may add friction in ways that counteract the benefits smaller teams have to offer. It is the job of management to hold such things in the balance.

Motivation

Psychology draws a distinction between intrinsic and extrinsic motivation—extrinsic motivation is when your only interest in an activity is some sort of external reward, whereas intrinsic motivation is when something internal comes into play, such as a feeling of fulfilment from doing a good job, or an aesthetic appreciation of some well-crafted work. Of course, a paid job always features an important extrinsic element, but if there can be some intrinsic motivation as well, then everybody wins. The person doing the work gets to enjoy the work in its own right, in addition to getting paid. Intrinsically motivated people tend to perform higher quality work, so the end user will typically get better results.

“A problem with dividing teams along lines of technical expertise is that it puts distance between work and its purpose”

A problem with dividing teams along lines of technical expertise is that it puts distance between work and its purpose. This makes it harder to see the point of your work, which tends to reduce intrinsic motivation. This is why DevOps encourages us to focus on shared responsibility for delivering useful work. It tends to make the purpose of the work more obvious than if goals are all defined in terms of arbitrary technical box ticking. Every individual on a team has the potential to act as a “change agent” and this is likely to produce much more useful results when the individuals involved can see exactly why their efforts are making things better. Moreover, a shared vision can help with the social cohesion of the team which can improve productivity and reduce the churn caused by unhappy employees leaving. There are no guarantees that DevOps will be a cure for unhappy teams, but it can be better aligned with intrinsic motivation than other common approaches.

Exploration, not Repetition

When considering the nature of software development, and what that means for how teams should be organized, it’s important to avoid a couple of common pitfalls. It is easy to think that software development is just like any IT service, but it is not. IT services tend to be focused on the reliable, predictable provision of a

particular set of well-defined services (such as ensuring that everyone has working email, or that the OS is up to date on all computers). When teams bring new services into existence, this is inherently a much more exploratory activity. The freedom to experiment rapidly is very important in software development, and this makes it quite different from typical IT service provision.

Software development is also not the same as a factory production line. It is mostly not a repetitive process, and there can be limits to how far work can be parallelized. Treating software as being analogous to physical production is an easy trap to fall into, not least because there are some useful insights from manufacturing that can be applied to software. In fact, some DevOps practices have their roots in lean manufacturing principles and the Toyota Production System and continue to share some important features. Lean manufacturing identifies 7 forms of waste (partially done work, unnecessary processes, low value features, task switching, waiting, motion, and defects) all of which are either directly applicable to or have analogs in the world of software. But production lines tend to want to optimize for producing the same thing again and again, whereas software development by its nature produces something different each time.

Executive Support

We will discuss the practical implications of the social aspects of software development later in this book, as we get into the detail of what DevOps means in practice. However, there's one implication that is especially important. These insights tell us that we must adapt organizational structures to the realities of software development. This is one reason why buy-in from senior

management is essential. You can only unlock the full benefits DevOps has to offer if you have executive support. This requires consistent, clear leadership—leadership that understands the importance, value, and priority of DevOps to the long-term success of the organization. Senior management needs to understand that DevOps is a persistent theme, not a 6-month project.

One of the themes of DevOps is the creation of a working environment that promotes continuous improvement and better understanding. Everyone involved needs to be given time to be introspective and to understand what isn't working, where waste or friction occurs or where opportunities for improvement exist. And more than just making space (and allocating budget) for this, leadership needs to establish that team members are expected to make good use of this, that improving the process is a vital part of everyone's job. Team members need to feel that they work in a safe environment to raise issues openly without fear of being shot down or perceived as being the problem rather than raising awareness of the problem. And leaders need to demonstrate that they value sustainability by giving time for this kind of thinking about continuous improvement and carving out the resources and budget to implement the improvements that people suggest. DevOps is about long-term sustainability of the endeavour not about meeting short term deadlines.

Feedback Loops

It's virtually impossible to do something well if you've got no way of knowing whether your actions are having the intended effect. Any software development process needs feedback loops—information about what we've produced that lets us know how good it is. DevOps emphasizes the importance of tightening feedback loops—minimizing the delay between doing something and learning how good a job you did.

Feedback loops take many forms. Arguably the most important is user feedback: if the people using what you've built think it's useless, then it doesn't really matter what your other signals may be telling you. This is one of the main reasons DevOps emphasizes minimizing lead times: the sooner you can get your work into the hands of a user, the more quickly you will find out if you're hitting the mark. This lead time is often referred to as *cycle time* in DevOps circles. The earlier you can discover a misstep, the less time you will spend building the wrong thing, and the less time the users will be stuck using something that doesn't suit their requirements. And even if the users do like what you've made, they might have constructive suggestions for what you could do next, and again, and you will want to feed this information back into the planning process as early as possible.

Asking the Right Question

Depending on the type of solution you're building, your customer may be a co-worker that you sometimes have lunch with; they could be your best (and possibly only) customer, or they could be something more ephemeral like "our first

customer". Each type of customer requires a different level of engagement and feedback. Are you able to interact directly with your users by walking over to their desk? Can you jump on a digital meeting to discuss and demo things? What if your customer base is too large to know each and every customer? Do you rely upon focus groups? Tweets? Support tickets? And what about the product that will define a new type of customer? For every successful product built under the guise of "If we build it, they will come" (with apologies to the movie Field of Dreams where the quote is "If you build it, he will come."), there are thousands of products that die before they ever make back their investment.

Steve Jobs is famously misinterpreted and misquoted for a couple takes on listening to customer feedback. The book "Steve Jobs" by Walter Isaacson, (published by Simon and Schuster, 2011) reports that when someone asked Jobs whether to do some market research to see what customers wanted, he replied "No, because customers don't know what they want until we've shown them." The first obvious point is very few people are Steve Jobs. Many write ups on his personality, behavior, and general opinions often portray him as a genius (something not being debated here). That said, the key to understanding these kinds of quotes (and looking deeper into them) is that Steve didn't like the way most big companies did research. The secret is simply to make sure you engage with humans and not numbers. Don't become dependent upon disconnected values like sales figures, number of downloads, bugs, etc. to tell you what to do. Ask. Observe. Inspect and adapt. Know your customer and provide value as quickly as possible and focus on continuous improvement.

Short lead times are particularly important for more experimental work. If you're just adding a new field to a form, the risks of a misunderstanding are non-zero but it is much more likely to go smoothly than introducing a wholly new area of functionality. And exploratory work is particularly likely to benefit from creative suggestions from users.

While feedback from the people the system is supposed to benefit is the most important kind of feedback, it's not the only kind. DevOps practices include the use of automated testing systems that can discover some kinds of mistakes long before your work gets into the hands of actual users. Unit test suites can discover certain kinds of mistakes within seconds of them being created. Automated integration and system test suites are not quite as fast, but can spot problems within minutes, or at worst, hours instead of weeks or months.

Continuous monitoring of production systems also provides a vital source of feedback. Wherever possible, technical staff would prefer to discover new problems before users do. You can improve your chances of doing this by monitoring the vital signs of various services—detecting when services are approaching capacity limits for example. You can also implement system-wide integrity checks, which might give you a "canary in the coal mine" warning that something somewhere has gone wrong, when you might otherwise only have noticed when disaster struck. It needs to be possible for everyone in the team to get access to monitoring data—not just those directly accountable for ensuring that services are available, but also support and development staff—because this information can provide useful feedback to all stages of development and production.

Some Popular DevOps Anti-patterns

"Having a dedicated DevOps team is a red flag, and sadly, quite a common one"

To understand DevOps, it can be useful to look at some common mistakes and misunderstandings. What anti-patterns often emerge when organizations attempt to apply DevOps, and how can we learn from them?

Having a dedicated DevOps team is a red flag, and sadly, quite a common one. This can arise from the perception that DevOps is a distinct practice, process, or set of skills. There are many practices and skills that are widely used in implementing DevOps, but it is a mistake for a single team to own these. It risks losing sight of the core DevOps principle of shared responsibility—if DevOps is now a thing that is owned by one team, that implies that for all the other teams, shared responsibility is now someone else's job, which obviously makes no sense. Moreover, DevOps is a mindset and a collection of principles that need to be applied across the board. DevOps must be something done by the whole team delivering a value stream, so partitioning it off into its own group won't work.

A perception can emerge that DevOps is just about tools and processes. For example, a team might be using tools enabling

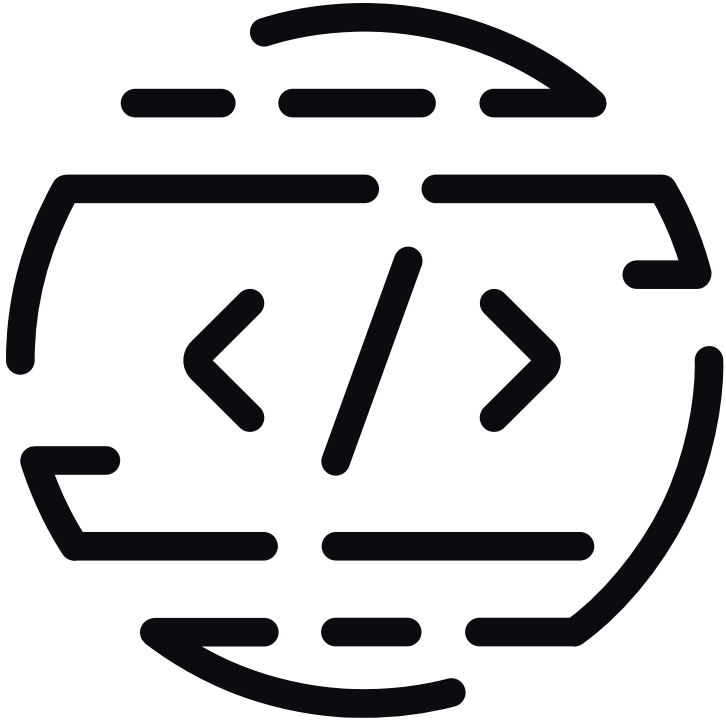
infrastructure as code techniques and employing continuous integration and deployment. These are certainly common practices in DevOps, but they are not the whole story. This point of view ignores the evidence-led aspects that enable us to determine whether we're doing the right thing. It ignores the importance of communication. The way we think about our systems, and the way we communicate both inside the team and with external stakeholders matters more than any specific tools and processes. Those technical facets have an important supportive role, but they are not the point. Fundamentally, while DevOps involves people, processes, and tools, the emphasis is always firmly on people.

Another anti-pattern that can emerge is for developers to attempt to control everything. DevOps does require developers to take an active role in some operational considerations such as deployment and monitoring, but this is not the same as saying that everything is development now. Certain aspects of operations may become a good deal less labor intensive thanks to automation, but there will still be operational responsibilities for which developers are not directly accountable. For example, developers will need to think deeply about how to ensure the quality of their code, but this does not mean that there is no role for testers.

Having a DRI

When working cross-team within an organization, it is often important to define a directly responsible individual (DRI). Various companies have used this model for years. Apple is often mentioned as one of the first "big" tech companies to

employ it. We use it at GitHub. In Scrum, the DRI for a Scrum Team is the Product Owner. Regardless of the exact job title used, the purpose is the same: the DRI is the named person who is responsible for a project. Understand this is not a dictatorial role. It is collaborative. DRI's should be chosen based on ability, not just title. They should have subject-matter expertise, an ability to focus and be able to help drive a project through communication. An empowered DRI uses a consult and decide model to make decisions.

There is a related, more general anti-pattern: a lack of clarity around responsibility and ownership. Because DevOps promotes the idea of shared responsibility, it can become unclear where the buck stops. Shared responsibility does mean the whole team pulls together to solve problems instead of saying "that's not my problem." Nonetheless, it continues to be important to determine who is ultimately responsible for ensuring that all the important things get done.

Now that we've explored the industry trends that led to the rise of DevOps, and the social and psychological backdrop that informs its approach, it's time to look at how that works. The next chapter will introduce the most important goals, concepts, and principles of DevOps.