PRAISE FOR PROGRESSIVE DELIVERY

"Progressive Delivery is one of those practices that seems simple on the surface but whose waters run deep.…Come for the practices, stay for the reframing of how to think about and improve your organization. *Progressive Delivery* just might be the catalyst that enables organizations to change."

—**Nathen Harvey**, DORA Lead and Developer Advocate, Google Cloud

"*Progressive Delivery* presents a working guide for people who are interested in building adaptively, responsibly, and agentically in the midst of rapid change.…This approach centers human decision-making, clarity of purpose, and collaborative goals, which have too often been lacking from out-of-the-box technology approaches."

—**Dr. Cat Hicks**, Software Research Scientist, Catharsis Consulting

"This book is written by the strategists who pioneered Progressive Delivery…The text steps between the four pillars of a delivery framework and corresponding case studies from big real-world teams. Ultimately, this is a call to action on why tech has to serve up customer happiness and not just process metrics."

—**Alexis Richardson**, CEO and Cofounder, ConfigHub

"This book builds on existing paradigms and sage wisdom to introduce the concept of Progressive Delivery. Get your highlighter ready, there's some good stuff in here!"

—**Katie McLaughlin**, Senior Developer Relations Engineer, Google Cloud

"From thought leaders in the industry, an invigorating new model for how (and why) to deliver software."

—**Rachel Chalmers**, Cofounder of Generationship.ai


PROGRESSIVE
DELIVERY

PROGRESSIVE DELIVERY

Build The Right Thing For The Right People At The Right Time

James Governor, Kim Harrison,
Heidi Waterhouse & Adam Zimman

IT Revolution
Portland, Oregon

25 NW 23rd Pl, Suite 6314
Portland, OR 97210

First Edition
Printed in the United States of America
30 29 28 27 26 25 1 2 3 4 5 6 7 8 9 10

Cover and book design by D.Smith Creative, LLC

Library of Congress Control Number: 2025012420

Paperback: 9781950508976
Ebook: 9781950508983
Audio: 9781950508990

For information about special discounts for bulk purchases or for information on booking authors for an event, please visit our website at www.ITRevolution.com.

Dedication

To our families, for putting up with us.
To our friends, for encouraging us.
To our colleagues, for inspiring us.

CONTENTS

PREFACE

Like many good stories, this one begins with rage.

***

It's a Tuesday evening at 6:17 p.m. You're making dinner. Your phone rings and you answer. Your parents are in hysterics. "We're trying to transfer money between accounts, and we can't figure out how to log in to the bank website!" After a few minutes of calming them down and trying to understand the situation, you realize their bank rolled out a new website, moved the location of the login screen, and implemented mandatory multi-factor authentication. You spend the next two hours helping them navigate the new interface and set up an authenticator app on their phone. By this point your dinner has burned and cooled into a charred mass. Technology has jerked your parents forward.

***

It's Friday at 9:52 p.m. You open the app on your phone to adjust the alarm on the "smart" speakers in your bedroom and your children's rooms. You need to ensure you're all up to make it to the airport on time the next day for your flight to Boston. When the app opens, it's different. You think, "Oh, cool, a new update. Looks nice, lots of rounded corners, etc." Then

you start looking for the alarm control settings. After twenty minutes of tapping on every section of the screen, you finally go to Google to find out where the alarm control moved to, only to learn through numerous Reddit threads that the new app removed all ability to see or change alarms in your system. The comments then inform you there is no way to revert or roll back the app version. You spend the next hour trying to set up alarm clocks in all the bedrooms without waking the kids, your partner, or the dog. Technology has jerked your family forward.

***

It's Monday at 8:27 a.m. You need to hop on a video call at 9 a.m. to prep your boss's boss for a meeting with the CEO about budget justification. You open the app for your video conference, and there is a pop-up window informing you that you need to update the app before continuing. You download the update, install it, and restart the app. You're able to get into the meeting at 9:06 a.m. and apologize for being late. You share a recap of the situation and are about to share your proposal when another pop-up window appears on your screen with the message, "For security and compliance, your computer will shut down and update in 3…2…1." By the time your laptop finishes updating, it is now 9:12 a.m. Your budget request was not approved. Why can't technology do a better job delegating control of when changes occur?

***

It's Wednesday at 9:41 a.m., and your CEO just flipped a feature flag for that cool new idea your team implemented from the main stage of your company's conference in front of a live audience of over five thousand users. Instantaneously, the user interface for hundreds of thousands of users changes. You've spent months working on this redesign, building and testing in production to ensure everything would work and had 100% feature parity. Over the next few hours, the reactions and reviews from users start to appear online. Half of the reviews are from happy users who

love the new interface and find the enhancements intuitive. But the other half are from users who are frustrated because you changed the workflow they used. Some are even having legal challenges because of contractual obligations around training timelines. Within days you hear this divide is showing up in sales meetings with customers as well. Some companies love the new vision and direction, while others are threatening to cancel contracts because of the disruption the change caused to their business. You could roll back the new user experience for everyone, but then the happy users would be angry (they really like the new design). On the other hand, keeping it on risks losing the users who were not ready for the change. You want to deliver the right product, but the readiness for something new varies across your user base.

***

It's Friday at 04:09 UTC, and you push out a routine content configuration change to 100% of your globally distributed enterprise customers. Due to a bug in your content validation system, your change passed validation despite containing problematic content data. Within hours your change has caused the crash of 8.5 million devices. The resulting economic impact from this incident is estimated at $5.4 billion. Your development practices would benefit from a more progressive approach to software delivery.

***

These are just a few stories that we have lived where the rapidly increasing rate of change has led to a technological jerk felt by users. Where some might be justified in their frustration, we want to channel the rage and start to build the right product for the right people at the right time.

INTRODUCTION

For the past thirty years, technologists have spent an immense amount of time and effort getting better at making software. We have refined how we deliver it, how we support it, how we build it, how we store it, how we run it, and even how we talk about it. The cloud is our environment, and the network is our foundational metaphor.

Technologists can create miracles and wonders with software, but without a user, none of that matters. Without users, we can't make money, change the world, or provide anyone with any value. Users are the other side of the software delivery equation, and they often get overlooked or undervalued in the process.

All of the work we've done to improve the software creation and deployment process in the last thirty years is effectively invisible to the user. On one hand, this is good. We don't want to share our struggles with our customers. But what does this evolution in the software development life cycle look like from the user's perspective? Let's flip the mirror.

What looks like deployment from our side looks like a release from the user side. This release is essentially a demand, a push, from us to them that requires a change in *their* behavior. We may ask them to update, or we may thrust updates upon them. But unlike greatness, update notifications are very persistent. The more often we want users to accept our changes, the more change we ask them to adapt to, even if it's very tiny.

Progressive Delivery takes the DevOps idea of breaking the wall between silos to its logical conclusion: We need to knock down the wall between software *users* and software *makers*.

We already have the tools we need for this demolition—automation across all layers of the stack, configuration as code, monitoring, observability, telemetry, feature flags, security built into the software development life cycle (SDLC), and, yes, data collection.

We can see how people use our software, or don't, if we only care to look. Once we understand how users truly engage with our software, we can package and parse that knowledge and use it to create software that better fits the user's needs and desires. In other words, we can understand software in the context of it actually being used, not just designed.

In our thirty years of improving software delivery, the part we've been missing is how our software affects the people who use it. Figure 0.1 condenses all the very real and important technical advances of coding, testing, and shipping as "making software" (on the left) and expands out all the ways our users can interact with what we've made (on the right). On top are user behaviors and on the bottom are the tools we share with users to make those behaviors possible.

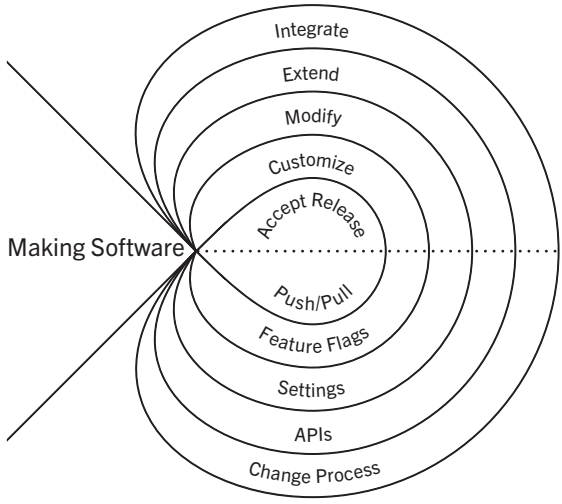

FIGURE 0.1: Software Acceptance and Use After Delivery

We, as software developers, are at the core, pushing or offering deployments and our users accepting them. Feature flags can customize the behavior of forcing or accepting changes. Settings can modify the software's environment. APIs can extend the program's data and behavior to a different format. Integration occurs when the user elects to use other software to interact with our software to meet their needs.

This *progression* reflects how the control point of software behavior shifts further away from the software creator (developer) and more toward the software consumer (user). It may seem counterintuitive that the "integrate/change process" is on the outer loop, since this seems pretty technical, but that's actually the point where our software interacts with the user's software. After all, our software is not the only tool our users are using to get their work done, so this ripple intersects with the ripples of dozens of other software products in the user's unique ecosystem (see Figure 0.2).

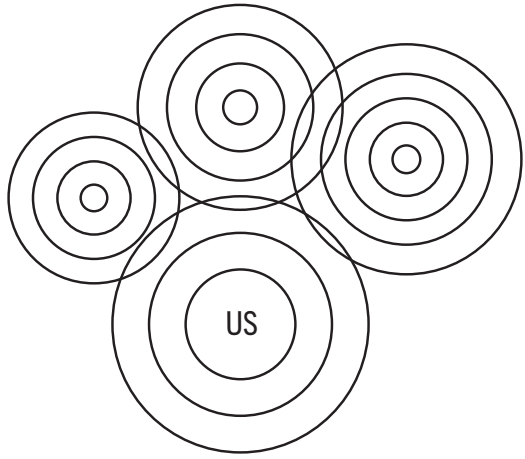

FIGURE 0.2: User's Software Ecosystem Has Many Interactions

Why We Wrote This

Any new way of describing the world requires context. It requires a community. It is a set of ideas and practices that are packaged up and it often has

a moment that helps to crystallize things. The software industry has spent the last thirty years trying to get better at writing software.

We (the developer community) have talked about continuous delivery and Agile. We have gotten much better at testing and shifted testing left. We have done many things, but we have never managed to do them for everybody. We're always promising a bright future…if only. The community working on user-centricity in the future already needs to make it a mainstream phenomenon. To make this new model accessible, we must name it and talk about it.

In 2017, James Governor had an intuition based on a conversation with Sam Guckenheimer, who worked at Microsoft. After hearing about the application routing processes Microsoft used for rolling out services, James realized that one part of the puzzle, which Microsoft called "progressive experimentation," was really about a broader phenomenon—Progressive Delivery. The impact of a basket of technologies and approaches applies to the entire SDLC. From there, our group came together—James Governor, Adam Zimman, Heidi Waterhouse, and Kimberly Harrison—and began to talk about, contextualize, and advance these ideas.

We all have a history of communicating with multiple stakeholders in the industry, helping them understand complex ideas and make them more broadly applicable. We have decades of experience and now we're bringing it together to bear on this new idea. Progressive Delivery takes all the goodness of the cloud and all the things that were not there when some of the original works in continuous integration/continuous delivery (CI/CD) were written and makes them applicable for now and into the future.

With continuous delivery and even late-stage Agile, there was the idea of the separation of deployment from release. With Progressive Delivery, though, we are adding that larger community in the context of our consumer. In Progressive Delivery, we now have deployment, release, and adoption. (See Figure 0.3.) That user cycle is representative of adoption, and that is the part we need to incorporate back into how we're thinking about our software delivery. We've gotten significantly better at shipping software, but helping people adopt that software and feel good about it… that's where we need to do a lot of work.

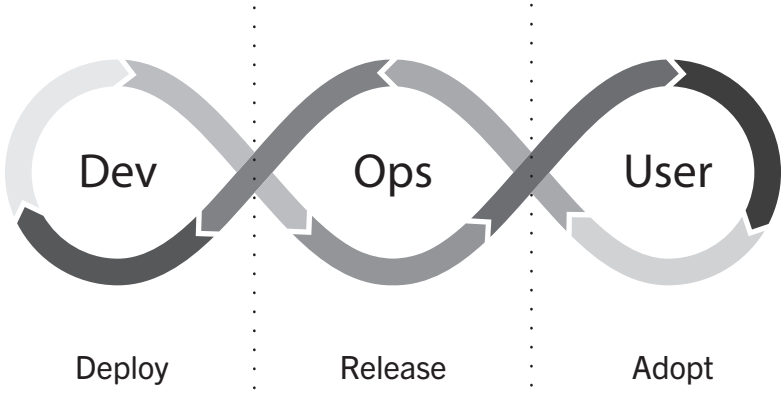

FIGURE 0.3: Deployment vs. Release and Who's Impacted

As Dr. Cat Hicks notes, "A successful software builder wants to create a successful, positive relationship to the change they're introducing."[1] We want users involved and happy. The truth is that different people are going to adopt software at different paces, and we're always going to be in an environment that is a mixture of the old and the new.

The first key breakthrough is understanding that the cloud changed everything because of its opportunities to increase autonomy, alignment, abundance, and automation. Particularly abundance. There are things we could do with the cloud that we could never do before. But the second most important breakthrough, the crucial breakthrough, is closing the third loop. It's not enough to have DevOps as two loops; we need to bring the user into the heart of what we are doing. That is the prize and the opportunity. That's what makes Progressive Delivery different.

Who Should Read This Book

Progressive Delivery is a holistic framework for an entire organization. It is intended to bring together the business, the builders, and the users in a way that honors everyone. But change really starts at the source, the builders. This book is primarily written from the perspective of enabling software

developers, who are at the front line of creating change, to create an environment where the right software gets delivered to the right people at the right time.

We say, "build the right thing for the right users at the right time." We start with "build" because until you build the software you can't deliver it. And how you build directly impacts your options at the time of delivery and your ability to observe adoption. Generative AI and vibe coding may be shifting the cost of building all variations to a nominal fee, but the cycle still starts with understanding the right thing to build. We need to start the conversation around how we are building to properly enable our teams to deliver to the right users at the right time.

Whether you are an engineering lead, a product owner, or an executive, this book is intended to expose you to the latest in software delivery thinking. Throughout the book, we also discuss how software creation and delivery affect other groups, who we call constituents (more on this below). We want to provide some useful ways to change your thinking about software delivery and some practical questions and techniques to make that delivery progressive, inclusive, and future proof.

When we talk about the collective of people who use and make our software and those who market, sell, and distribute it, we could use the traditional expression "stakeholders," but mostly, we prefer to think about that constellation of people as *constituents*.

- A **stakeholder** is a person who cares about the outcomes. In a very literal sense, stakeholders have a stake in the success of the product. This can be a developer whose job performance is tied to the product, an investor, or company management.
- A **constituent** is someone who contributes to success. This can include developers, support, marketing, users, and IT departments.

We need to treat users as participants in our work rather than as objects. We're doing something with them, not to them.

How to Read This Book

This book is a layer cake of theory and practice. The theory chapters provide explanations for what we are seeing in the industry, what you can look for in your organization, and questions to ask yourself about your alignment with Progressive Delivery. The corresponding case study chapters demonstrate a particular aspect of Progressive Delivery in action, but, of course, other elements also make their way in. Read through the book and focus on the parts that line up with your current experience. Then go through and use the questions at the end of the chapters to consider how you want to tweak the practices and behaviors in your organization.

Tools and Patterns

These tools and patterns are ways that we have seen organizations practice Progressive Delivery. Many of them flow into each other or relate to each other, but we are listing them in alphabetical order for ease of reference. We're introducing these concepts here as they'll come up throughout the book and form a foundation to engage in moving toward a Progressive Delivery approach.

Blast Radius

This is a way to describe how much effect a change will have. It is often coupled with ring deployment or canary deployments. Changes with a small blast radius limit the impact of changes since only a few people will be affected. Limiting the blast radius also provides an early feedback loop on changes from the user perspective.

Blue-Green Deployments

Blue-green deployments are often used in a "breaking change" scenario. If a software change is going to change how data is stored and communicated, the blue-green pattern helps prevent data loss. A second full system is set up that mirrors the original system, and traffic is directed to both systems

simultaneously to check that the data is all being stored properly and that the new system is robust. Only then is the older system shut down. Variations on this pattern include load migration and traffic shaping. The pattern is also related to sunsetting.

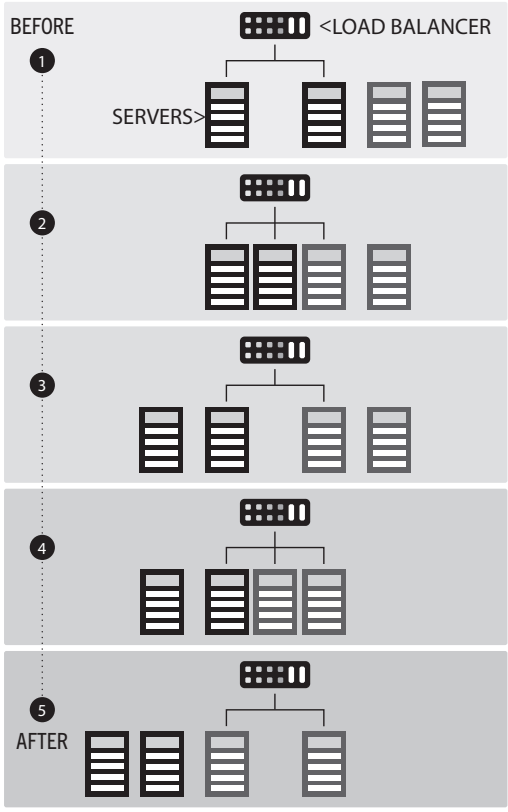

FIGURE 0.4: Progression of a Blue-Green Deployment

Canary Testing

Derived from the use of canaries in coal mines as an early warning for poor air quality, a canary test rolls out a software change to a small group of monitored users and checks their response and experience. In the coal mining story, the canary stops singing and faints if it loses oxygen. Since canaries are very small, it's a sensitive indicator. In the same way, canary

testing is a sensitive test that can indicate general safety for the group, but only if it is well-monitored. Canary tests are often administered by feature flags and may be part of a ring deployment strategy.

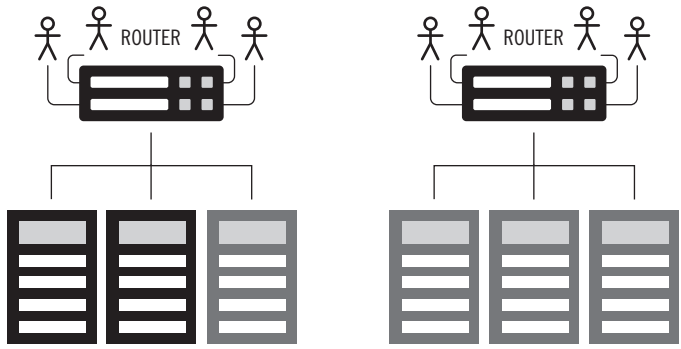

FIGURE 0.5: Canary Deployment

Used to Evaluate Viability of a Change Before Exposing to All Users

Constituents

Software is not just a set of computer instructions. It is a web of relationships between people, processes, and systems. The constituents of a Progressive Delivery system include the developers, the product team, the businesses that create and consume the software, the environment, and the users. For example, a healthcare record system is created by a product team and developers, sold by marketers and salespeople, maintained by operations and support staff, and used by insurance companies, healthcare providers, and patients. All those people are part of the constituency of the healthcare software.

Feature Flags

Feature flags are a way to change the behavior of software at runtime based on conditions that may be external to the code. Feature flags can be used to control software based on conditions such as user ID, browser language, geographic region, software version, security permission level, A/B testing cohort, and server.

Feature flags frequently fall into two categories: ephemeral flags, which are used for a finite period of time and then removed from the code base to prevent inadvertent activation, and long-lived flags, which control aspects of the software that will continue to be variable. For example, an ephemeral flag might control the phased rollout of a new feature. A long-lived flag might control software that has a paid premium tier. Feature management software helps organize, control, and distribute an organization's feature flags.

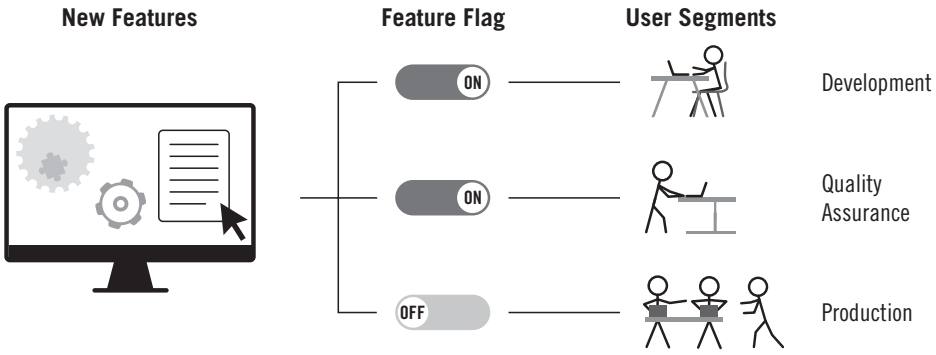

FIGURE 0.6: Feature Flag Controlling Which User Segments Have Access to a New Feature

Observability

Observability is the combination of gathering high-cardinality data about a system (including its users) and being able to ask unanticipated questions about that data.

Release Impact

Much like blast radius, release impact is a way to understand the effect of a software change. However, release impact also implies that the change may be positive. Some implementations of release impact also include a consideration of monetary effects.

Release vs. Deployment vs. Acceptance/Adoption

Deployment is the act of getting software to a place where it will be available to the users. Release is the point where users can actually use the software and are told about it. Acceptance or adoption is when users make the software a part of their workflows.

Ring Deployments

A ring deployment is the practice of deploying software to increasingly larger groups of people as part of a release strategy. For example, the first ring might be to the team, and the second ring might go to 1% of the users, then 10% of the users, etc. At each stage, the impact is evaluated.

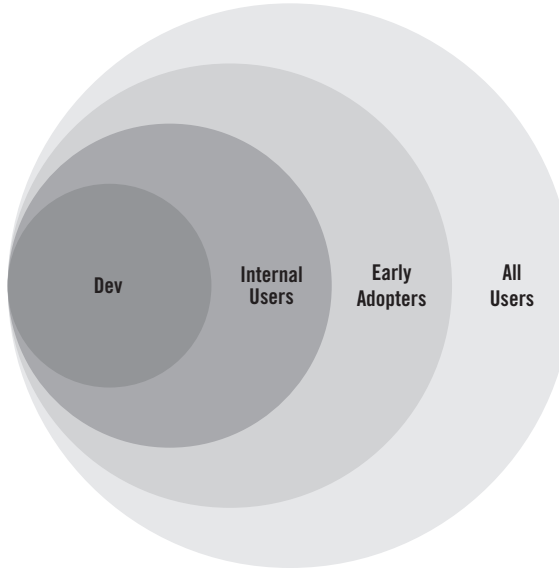

FIGURE 0.7: Ring Deployment

Rollbacks

One way to make changes less dangerous is to ensure that they can be reverted cleanly. Controlling releases with feature flags makes it faster and easier to roll back to a previous state without needing to change code, especially in an urgent situation. As Thomas Dohmke, CEO of GitHub, said in an interview with us: "The feature flag is only really useful if you can't only

progressively roll out, but you also need to be able to aggressively roll back. That's actually the key feature."[2]

Test in Production

In a complex modern software environment, it is impossible to fully test every scenario before software is released. However, production is a test environment from which we can obtain valuable information if we choose to record and integrate it.

Sunsetting

All software has a lifespan. When software needs to be retired, some users are ready to move on to the next thing, and some aren't, for business or personal reasons. Sunsetting is the act of retiring software or versions using feature flags so there is not an abrupt cutoff but a mindful wind down.

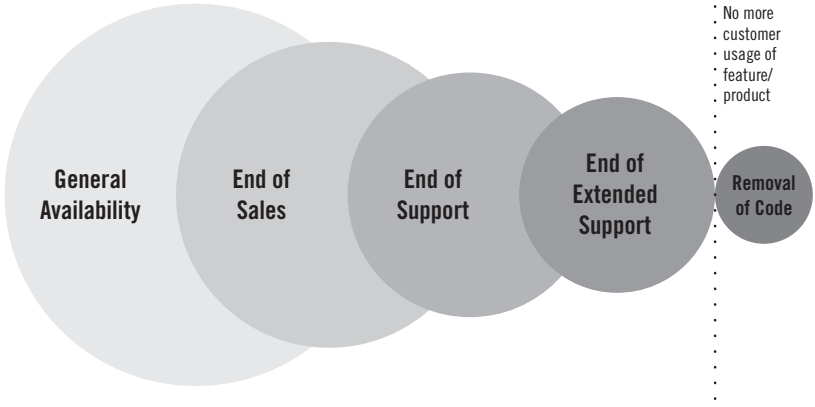

FIGURE 0.8: Software End-of-Life Diagram

Progressive Delivery Is a Mindset

Empowering the user to change their experience of software is an extension of the Agile, DevOps, and CI/CD philosophies. Our collaboration circle

grows wider as our ability to understand and incorporate data increases. From the organizational side, abundance, autonomy, alignment, and automation make it easier for organizations to create and sustain software that is flexible, responsive, and useful.

We believe that with this guide, you will be able to look at your own organization and see places where you can improve one of the four A's and thus deliver value a little sooner or more accurately or make the work with others easier.

Chapter 1
PROGRESSIVE DELIVERY

> "Well, in our country," said Alice, still panting a little, "you'd generally get to somewhere else—if you run very fast for a long time, as we've been doing."
>
> "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"
>
> —**Lewis Carroll**, *Through the Looking-Glass and What Alice Found There*

In physics, a jerk isn't just someone cutting you off in traffic—it's the rate at which acceleration changes. Technically known by physicists as the third derivative of position, it's the feeling that makes you grab for the subway pole when the train lurches or brace yourself during an elevator's sudden start. It's that moment when steady, predictable motion becomes a jolt, defying your expectations of smooth acceleration.

jerk (/jurk/): The rate of change of an object's acceleration over time.

We feel this same jerk in our digital lives, where change itself is accelerating. The history of technology has been hallmarked by an ever-increasing velocity of transformation.

As Alvin Toffler warned in 1970, change is "a concrete force that reaches deep into our personal lives, compels us to act out new roles, and confronts us with the danger of a new and powerfully upsetting psychological disease." He called this phenomenon "future shock,"[1] and nothing in our current environment suggests the pace Toffler found dizzying fifty years ago will slow down.

These technological jerks reshape our personal worlds in profound ways. For someone born in the 1940s, a telephone represents stable technology—pick it up, dial, talk. For those born in the 2000s, the "phone"

function might be the least-used app on their device. Everything from how we get our news to how we pay for coffee has become a digital experience that updates without warning, consent, or control. The global infrastructure we built in the twentieth century—networks of satellites, fiber-optic cables, and physical goods transfer—has compressed adoption timelines from decades to months. (See Figure 1.1.)

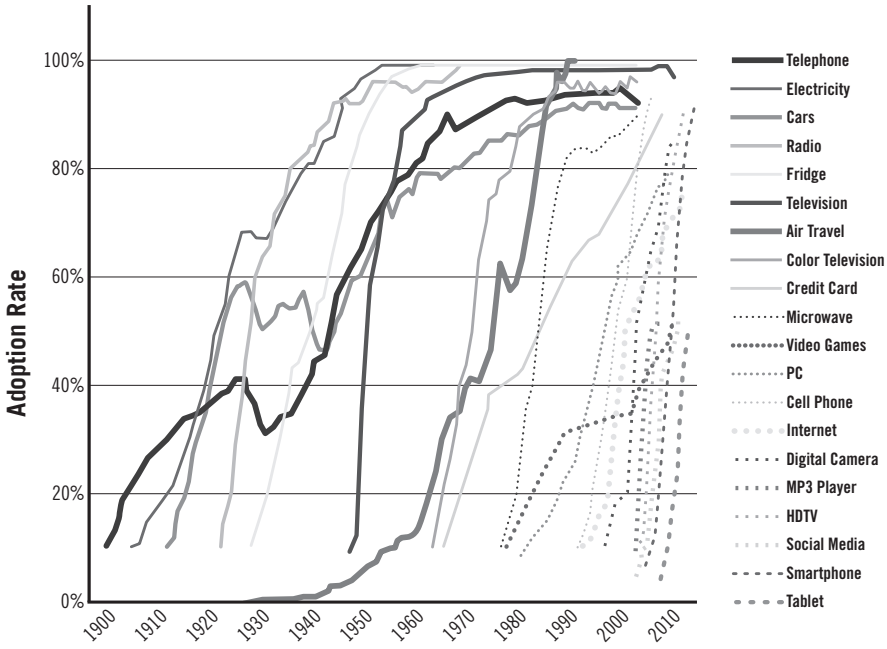

FIGURE 1.1: Adoption Rate of New Technologies from 1900 to 2012

Source: "The Topic We Should All Be Paying Attention to (in 3 Charts)," BlackRock Blog, December 11, 2015, https://web.archive.org/web/20160304140915/https://www.blackrockblog.com/2015/12/11/economic-trends-in-charts/.

In 1962, Everett Rogers captured the varied human response to this technological acceleration in *Diffusion of Innovations*, mapping out how new technologies ripple through society—from eager innovators who embrace the bleeding edge to early adopters, then the early and late majorities, and, finally, the cautious laggards who hold onto the familiar.[2] Geoffrey Moore

later expanded this insight in *Crossing the Chasm*, revealing the treacherous gap between early enthusiasm and mainstream acceptance.[3]

Yet our relationship with change isn't simple. A developer might be the earliest adopter of a new operating system on their phone but continue to use a code editor that was built in 1976.* We are all early adopters in one area but laggards in another, picking our way through an increasingly complex technological landscape.

In our professional lives, these jerks multiply. Software dashboards proliferate—one for time tracking, another for performance metrics, and yet another for project management. Each makes perfect sense to its creators, but collectively they create a dizzying acceleration. When we ask colleagues to adapt to interface changes, we're asking them to absorb another jerk in their already dynamic workflow.

Organizations feel these forces of change even more acutely. They must innovate rapidly to stay competitive—ask Sears about the cost of failing to adapt to Amazon—while managing the increased risks of outages, user frustration, and business disruption. Traditional change management systems excel at handling smooth, predictable acceleration but falter when confronting these technological jerks.

The solution isn't to slow down—it's to give people more control over their rate of change. Every time we allow users choice, whether in personal tools or workplace software, we enable them to manage their own acceleration. Some choices can be elegantly wrapped—such as advanced settings hidden behind a simplified interface—making people partners in the software experience rather than subjects of it.

This is where Progressive Delivery comes in: a methodology that recognizes different users need different rates of change. As software builders, we can release as quickly as we want while letting users choose when to incorporate changes into their lives and workflows. It's about building systems that are both dynamic and respectful, systems that recognize the human need to sometimes grab the pole and steady ourselves before the next technological jerk arrives.

---

* Both "vi" and "Emacs" were first created in 1976 and remain two of the most popular code editing applications today.

The cost of mismanaging rollouts is all around us. Microsoft found itself forced to extend Windows 10 support when organizations balked at upgrading to Windows 11.

A tiny npm package called left-pad created a cascading failure that affected thousands of projects. A security company called CrowdStrike, which tens of thousands of organizations relied on, caused a major outage by pushing a breaking misconfiguration to 100% of their audience all at once. The cost of poor software delivery practices can run into the billions. It gets kind of expensive when the entire airline industry is grounded. These cases demonstrate what happens when rollouts are not effectively managed. And, really, as an industry, we should be doing better by now.

The signs of this mismatch are clear in any organization: declining user engagement, unused new features, the proliferation of third-party workarounds, and spikes in support requests. But these symptoms also point toward solutions. By understanding how different users and organizations absorb change—from early adopters to cautious laggards—we can create systems that respect their varying needs for stability and innovation.

Over the past century, we've seen adoption rates for new technologies compress dramatically. While television, computing, and other technologies required decades to reach mass adoption, the latest software-driven innovations can become mainstream in months (see Figure 1.2). This acceleration isn't slowing down—just look at ChatGPT.

As software builders, we're both agents and victims of this acceleration. Our code is just one thread in a vast tapestry of interdependent systems, each evolving at its own pace. When we push changes too fast or too frequently, we risk creating that jarring moment—that technological jerk—for our users. The impact depends on how quickly they're already adapting to change: What feels like a gentle nudge to an early adopter might throw a late majority user off balance entirely. We are not the only ones asking our users to adapt to changes—they use more than just our software, both at work and at home.

Throughout this chapter, we'll explore how Progressive Delivery provides a framework for managing technological change that respects both the need for innovation and the human experience of adaptation. By

understanding how to deliver the right changes to the right users at the right time, we can turn the jarring experience of technological jerk into a more controlled and intentional acceleration. Let's start by examining exactly what Progressive Delivery means in practice and how it emerged as a response to these challenges.

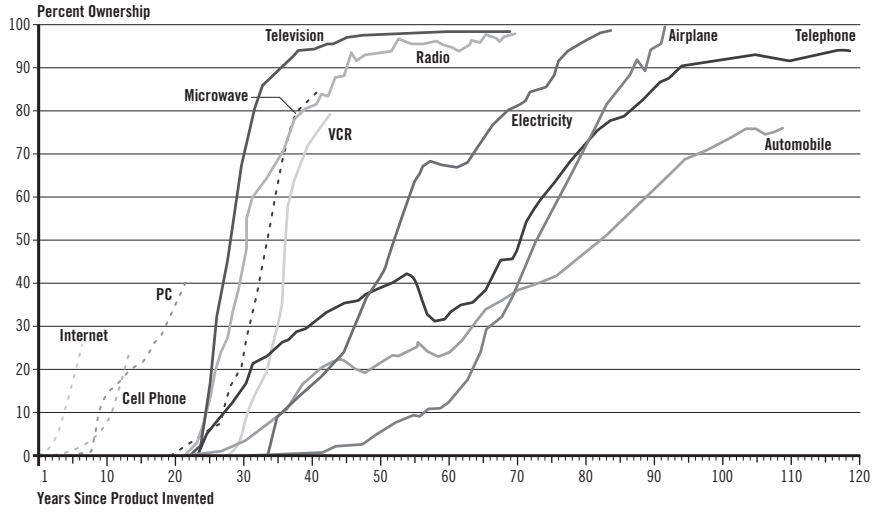

FIGURE 1.2: Years Since Technology Introduction to Reach Mass Ownership

Source: Federal Reserve Bank of Dallas, *1996 Annual Report: The Economy at Light Speed*, https://web.archive.org/web/20161224074319/https://www.dallasfed.org/~/media/documents/fed/annual/1999/ar96.pdf.

Toward a Practice of Progressive Delivery

Everywhere we look, we find new devices and services that offer replacements or enhancements to every aspect of our lives. But with these improvements come new challenges. If your device or application doesn't work, how does it get fixed? How long does it take? What if that software is running in your car? Or the locks on your house? Or the pump for your insulin? Is your software doing what you need, when you need it?

Different stakeholders want to move at different rates—factories want to run consistently all year, but consumers have times when they want to buy back-to-school clothes or holiday presents. Software developers want to be able to show delivered products before their performance reviews. Sales teams are driving toward quarterly and yearly goals. These stakeholders need a way to collaborate, not just coexist.

At its core, **Progressive Delivery is a set of software delivery practices to deliver the right software to the right users at the right time in a way that is sustainable for everyone**. Yes, everyone. This includes executives in the boardroom, leaders managing departments, engineers, designers, product teams, marketers, partners, and, most importantly, the actual product users. While this book is focused on software developers and how they can benefit from Progressive Delivery methods, Progressive Delivery is for all these stakeholders and constituents.

Progressive Delivery is not about tools or certifications. It's about what you care about and where your organization places focus. It's more of a lens than a prescription. Products are not static entities but thriving conversations where building, use, and retirement are all visible and trackable.

From a more nuanced perspective, Progressive Delivery can mean different things for different constituents:

- For the user or consumer of technology, Progressive Delivery is a user experience that minimizes technological jerk.
- For the company delivering a digital experience, Progressive Delivery is a set of practices that enable teams to move at a sustainable pace.
- For those tasked with building and delivering modern software, Progressive Delivery is a development practice that builds upon the core tenets of continuous integration and continuous delivery (CI/CD).

Progressive Delivery specifically adds two core tenets to that of CI/CD:

1. **Release progression:** progressively increasing the number of users who can see (and are impacted by) new features.
2. **Radical delegation:** progressively delegating the control of access to a feature to the owner who is closest to the outcome.

In essence, Progressive Delivery is the practice of delegating control to the user while retaining a clear vision and plan for the product. It's a way to understand what you're already doing regardless of the technology change happening in front of you, so you can do it more effectively.

Progressive Delivery asks the following key questions:

- What is "finished?" When is a product or feature truly complete, and how do we define success?
- What do we expect to happen? What are our hypotheses about how users will interact with the new features?
- What if users want a different cadence of change? How do we accommodate diverse user preferences?
- How are we stewarding the information we collect? How do we gather and analyze user feedback?
- How are we incorporating feedback? How do we use feedback to improve the product?
- Who are *all* of our constituents? We must recognize and consider the needs of all stakeholders, not just the loudest voices.

In the history of software development, Progressive Delivery represents the logical next step in a long line of improvements. According to Carlos Sanchez, who wrote the following while working at CloudBees:

> *Progressive Delivery is the next step after Continuous Delivery, where new versions are deployed to a subset of users and are evaluated in terms of correctness and performance before rolling them to the totality of the users and rolled back if not matching some key metrics.*[4]

Figure 1.3 shows the evolution of software development methods. While not comprehensive, it shows how our understanding of delivery can be additive. Specification-driven delivery (also known as waterfall) plus Agile gets us test-driven delivery (TDD). When we add operations and maintenance into the scope of TDD, we get DevOps. Adding automation to DevOps results in CI/CD. Progressive Delivery includes all the former models the way a pearl encapsulates its former layers.

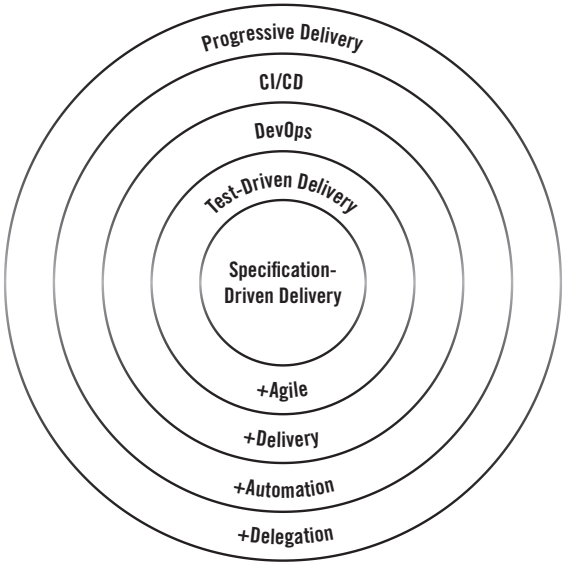

FIGURE 1.3: The Evolution of Software Development Methods

Of course, as software makers have been optimizing how to build software—through innovations in tooling and craft with continuous delivery and DevOps practices—they have exacerbated the problem of user adoption. Even if a team can *deploy* on demand, a user probably will not *adopt releases* multiple times a day.

This is the crux of why users are feeling the technological jerk now more than ever—adoption is about release cadence, not build cadence, but not all our systems are designed to separate those. The essential added ingredient in Progressive Delivery is delegation closer to the user.

This is how we continue down the path of high developer autonomy. We build systems that decouple deployment from release, and release from adoption, so users can operate at a more comfortable speed.

Once you start seeing the world in terms of Progressive Delivery, you see it everywhere—ripe mangoes in Midwest supermarkets and tap-to-pay parking meters, Calendly links, and same-day electronics delivery. User demand drives and encourages changes to delivery infrastructure. Consider Calendly: Setting up a meeting with someone used to require several steps, including figuring out availability for each person. By creating software to allow each user to independently choose a time, booking meetings has become faster and easier.

On the provider side, this coordination requires calendar rules, time zone awareness, email integration, and meeting location options. Similarly, delivering fresh tropical fruit to Minnesota in February requires a sophisticated transportation and distribution network and fruit varietals that are sturdy enough to ship and store. To the user, Progressive Delivery looks like convenience. But to a provider, Progressive Delivery takes a combination of investment, will, and effort.

The Four A's: A Framework for Progressive Delivery

The evolution of Progressive Delivery has been shaped by technological advances, much as physics has evolved to measure and manage forces of motion. Just as physicists use measurements of jerk to understand sudden changes in acceleration, there are four essential factors that help us measure and manage the technological jerks in our system: the four A's—abundance, autonomy, alignment, and automation. The rise of virtualization, containerization, and cloud computing led to the abundance of computing and storage resources. This abundance of resources led to increased developer autonomy, which was further accelerated by Git, distributed contribution, feature flags, and the architecture trend from monoliths toward microservices.

As autonomy increased, so did the need for focus and alignment. Teams began to prioritize—and value—API-first development and enhanced observability. This more loosely coupled architecture led to both the opportunity and the need for more automation and better feedback loops to manage the vast increase in the scale of systems and the opportunity to better understand user behavior and needs.

We can express this relationship as an equation:

$$Progressive\ Delivery = \frac{(Abundance \times Autonomy)}{(Alignment \times Automation)}$$

Abundance and autonomy form the foundation of the developer experience, much like the electrical grid supports our modern life. The fluctuations of power generation and conduction are smoothed out, and we get steady, reliable resources to use. We then get to choose how to apply the power streaming into our homes and businesses so abundantly. In the same way, abundance and autonomy in software development allow us to think about more difficult and interesting problems. However, just as we use everything from circuit breakers to dimmer switches to control the flow of power, the forces of abundance and autonomy also need to be well-regulated to be useful and safe.

Your "goal" for Progressive Delivery is to balance your abundance and autonomy by leveraging alignment and automation. If abundance and autonomy are too pronounced compared to alignment and automation, teams tend to build brittle systems filled with features that never get used. Conversely, if you focus too much on the user experience without addressing developer needs, you end up knowing what the users need, but you are unable to deliver it quickly enough.

In this way, abundance and autonomy are all about the developer experience, or the building side of a product, while alignment and automation are centered on the user experience, or the delivery of the product. We could simplify this as:

$$Progressive\ Delivery = \frac{Developer\ Experience}{User\ Autonomy}$$

If abundance and autonomy are the electrical grid, delivering us power and potential, then alignment and automation are the appliances that transform that energy into value. Voltage on a power line is not useful until we can convert it into light, heat, work, or video gaming minutes. Too much power and there's a risk to safety and property. Too little and we can't turn on a light or keep food cold. Alignment is what directs the current the way we want it. Automation makes our homes run without intervention and keeps us safe from mistakes or sudden surges. Without alignment and automation, we would be at risk of surprises or unwanted changes.

Let's examine each of these four pillars in detail:

Abundance

Abundance is a very large quantity of all the resources required to accomplish a task. In the context of Progressive Delivery, this centers around the developer experience. When building digital systems, this can be divided into compute resources, network bandwidth, and storage.

We can measure abundance both quantitatively (for example, how long it takes to provision a server or database for a new project) and qualitatively (for example, through developer surveys and interviews). Developer experience and abundance are interlinked. Abundance enables developers to work without friction and without waiting for permission to access resources.

Autonomy

Autonomy is the ability of an individual to act independently from others. When developing software, this independence means access to all necessary resources to complete a desired task. To have a Progressive Delivery environment, developers need to be able to innovate and build at their own pace.

To measure autonomy quantitatively, we can track how frequently developers are "blocked" or waiting for others to do their work. During some stages of growth or product expansion, the rate of blocking may naturally increase. We can also gain qualitative assessment through internal surveys.

Alignment

Alignment means focusing human and organizational resources responsible for developing software to work in the same direction. In Progressive Delivery, alignment is one of the two ways to wrangle abundance and autonomy. Both alignment and automation are centered around the user experience.

We can measure alignment through qualitative user surveys and interviews, as well as by monitoring usage rates and patterns in feature adoption and workflow completion. The exact method for gathering quantitative and qualitative data about user impact will vary with the software and the users, but it should be as broad as the team can afford, in order to capture multiple insights.

Automation

Automation is the identification and implementation of programmatic processes for repetitive tasks. For Progressive Delivery, automation is the second way to focus on abundance and autonomy. Automation supports alignment by intentionally looking for repetitive manual tasks and creating code to reduce effort while ensuring consistency. After all, one of the goals of computing, and now AI, is to make automation easier and more effective. Adoption is easier when it's automated and part of the workflow.

Measuring automation can be done quantitatively through observability tooling, which looks at the frequency of pattern repetition as users navigate a workflow. Qualitatively, user surveys can target questions about repetition and "too many steps" to accomplish frequent tasks.

Balancing Developer and User Experience

The benefit to adopting Progressive Delivery is that it is not an abrupt transformative moment but an evolution that works with what you're already doing well and gives you pointers to what could be improved. The cost of a

"transformation initiative" is often denoted in millions, and the outcome may not be at all aligned to benefit the people who are implementing the changes and those consuming the result.

Just as electrical engineers need to balance variable generation and transmission with safe, reliable, controlled delivery, Progressive Delivery works to balance the surge and ebb of developer innovation with the measured and incremental pace of user acceptance. The goal is not to eliminate change or even acceleration, but to make it as smooth and acceptable as possible. The separation between deployment and release acts as a transformer, modulating the flow down to something a household can use safely, while still retaining the capacity to serve other households.

Progressive Delivery addresses the challenge of the pace of innovation by making a hard separation between the deployment of code to the production environment and the release of features to users. This separation allows for the business to have two priorities that are loosely coupled: developer autonomy and user adoption. (See Figure 1.4.)

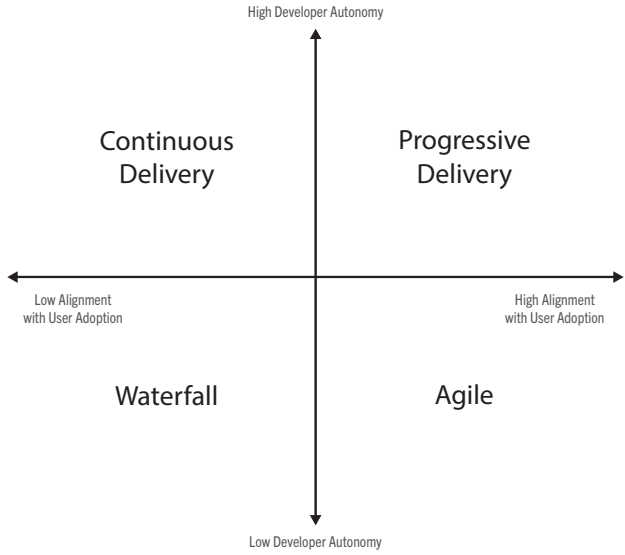

FIGURE 1.4: How Software Development Life Cycles Balance Developer Autonomy with User Adoption

Motivation and Sustainable Growth

Similarly, product teams as a whole need to know that there is a user demand for what they are building, and companies need to be able to situate themselves in an ecosystem of production and consumption. All of this alignment is much easier when the goal is something that can be communicated to everyone.

Dan Pink's *Drive* posited that humans are intrinsically motivated by autonomy, mastery, and purpose.[5] This theory fits well with what we know about burnout from Dr. Christina Maslach's work, where lack of autonomy and purpose and conflicts in moral values create a kind of moral injury.[6] Being able to connect our labor to the value that other people find in our work is a known way to stay engaged and happy.

We know that stasis is dangerous for companies—if you're not in touch with how your environment is changing, you're at a high risk of being passed by a competitor or becoming irrelevant. We also know that growth at all costs is a risky goal, especially in a post-ZIRP* world. Company growth needs to be sustainable or have sustainability on the horizon.

Finding the Middle Road

There are so many business metrics out there, and while we will give you a few more, the metric is not the goal any more than the map is the territory. If we measure people on something easily measured without repeatedly asking why they need to increase that measurement and the intended effect, then we get compliance but not cooperation.

So how do we find that middle road of making something useful, flexible, and sustainable?

- By delivering the right product to the right person at the right time.

---

* ZIRP: zero-interest-rate phenomenon. In this case, the behavior of companies when it is effectively free to borrow money. Although associated with the economic term *zero-interest-rate policy*, it is specific to how low borrowing costs affected risk estimation around investing in software and venture-backed startups.

- By avoiding overbuilding and over-optimizing.
- By working with the resources easily available.
- By making sure that we are addressing real needs our users value, not just what the loudest people are asking for.

If change is an inevitable part of our lives, both as producers and consumers, how do we make that change meaningful and useful instead of pointless motion without progress? To answer that question, we need to know what the point is—what are we trying to accomplish with what we're making, and what are the people who use it trying to accomplish? Without these purposes clearly in mind, we can never be sure that we're making the right thing.

Conclusion

Each of the four A's of Progressive Delivery reinforces and enables progress in the others. None of them is something that can be fully finished. Moore's Law continues to provide an abundance of resources. You can always automate a little more, or a realignment will reveal a way for a team to become more autonomous. Even autonomy continues to increase and expand in the face of coding assistants.

Change is a part of our lives every day. We tend to think of it as good change, like increases in capacity or learning, or bad change, like aging and decay. Change is stressful because it forces us to learn new habits and patterns and ways of doing things. The larger and faster a change is from a single point of view, the harder it is to adapt to it. Jared Spool, cofounder of Center Centre, said in the article "The Quiet Death of the Major Re-Launch,"

> *There's another way to build a new architecture with a whole new site without the risks of a re-launch.…I explained that re-launches are a thing of the past. There was a time when sites launched in cycles, living from one major redesign to the next. Each new redesign would bring a whole new look, a whole new user experience.…However, the best sites have replaced this process of revolution*

> *with a new process of subtle evolution. Entire redesigns have quietly faded away with continuous improvements taking their place.*[7]

The way we build software has evolved to make it trivial to push changes to our users. But just because it's easy to change things doesn't always mean it's the right time or situation to do so. This is where Progressive Delivery shines—by providing a framework that balances capability with responsibility, speed with sustainability.

In physics, understanding jerk helps engineers design better systems—from elevator controls to autonomous vehicles. Similarly, understanding the forces of technological change through Progressive Delivery helps us build better software systems that respect both the need for rapid innovation and users' capacity to adapt to change. Modern software delivery works because we have an abundance of software and network resources, the autonomy to find the best path to solve a problem, the alignment to work within a distributed system, and the automation to preserve our energy for novel and challenging tasks. Through Progressive Delivery, we can ensure that this malleability serves both the creators and consumers of technology, making change not just possible but purposeful.

Chapter 2
ABUNDANCE

> If quantity forms the goals of our feedback loops, if quantity is the center of our attention and language and institutions, if we motivate ourselves, rate ourselves, and reward ourselves on our ability to produce quantity, then quantity will be the result. You can look around and make up your own mind about whether quantity or quality is the outstanding characteristic of the world in which you live.
>
> —**Donella H. Meadows**, *Thinking in Systems: A Primer*

In physics, potential energy is the energy that is stored in a system. As we have explained, jerk is the sudden, unexpected change in acceleration that throws us off balance. Abundance, for developers and builders of software, is how many resources you have available—technological potential energy. This potential energy powers your innovation. When used responsibly, abundance can provide steady acceleration and help avoid the jerk caused by exposing too much change too quickly to your users.

Over the past fifty years or so, our society has moved from an environment where technology was a scarce resource to one of abundance, where technology is not only cheap but all-pervasive. This transition represents a fundamental shift in the world of software development—from a world of constrained motion to one of technological momentum.

Prices of memory, compute, and storage continue to drop as maximum densities continue to climb. We're all familiar with Moore's Law, first described by Gordon Moore, Intel's cofounder, in 1965. He predicted that the number of transistors on a single computer chip would double roughly every two years with a negligible increase in cost.[1] This exponential growth creates a form of technological inertia—a mass and velocity that, once in motion, becomes difficult to slow down or redirect. Though this initial

observation was in relation to compute, the same growth of density has been roughly equivalent for both memory and storage as well.

At the time of this writing, a 1-terabyte hard drive costs less than $30. Phones are considerably more powerful than mainframes were twenty years ago. The cloud made, and continues to make, this abundance accessible to anyone with a credit card. High-speed networking and 5G have removed bandwidth as a limitation in most regions. Software, too, is cheap. (Or even free, as in a puppy, which may have no up-front cost but a lot of maintenance expenses.) Open source has driven an abundance revolution in software. Each of these developments adds mass to the technological momentum that organizations must now harness rather than resist.

So, what does all this abundance mean in the context of Progressive Delivery? And how can we harness this momentum without creating disruptive jerks in our systems and for our users? Let's define it clearly:

a·bun·dance (/əˈbənd(ə)ns/): More than enough of all the resources required to accomplish a task.

In the context of Progressive Delivery, abundance (along with autonomy) forms the foundation of a better developer experience, much like the electrical grid supports our modern life. This translates to better product management and applications and services that users can adopt at their own pace. It is part of the foundation that absorbs the shock of rapid change.

Historical Context of Abundance

Historically, software delivery was defined by resource constraints—a world of low technological mass and high friction. Like trying to push a heavy object across a rough surface, every movement requires significant force. Waterfall methodologies were partly a response to this lack of resources. You had to get things right (in theory, at least) the first time, with specifications and infrastructure requirements defined up front.

In this constrained environment, change was expensive and jerky—each new project represented a major acceleration from a standing start.

(Another way to look at this is if you graph innovation, waterfall is a step function, Agile made the steps smaller, and continuous delivery allowed the steps to smooth out to a curve.) Teams were split into different functional groups, each with their own infrastructure—developers needed access to development servers; test and QA had their own servers, storage, and so on; and production was a separate team with its own infrastructure and tools. High availability incurred huge costs—each extra "9" of availability added an order of magnitude to system cost. There was a great deal of replication and a lot of time spent waiting for permission. A development team could wait literally months to have resources provisioned to start a new project or application, creating a stop-start motion full of technological jerks.

The Abundance Transition

Think of the abundance transition we've made since 1995, when the internet revolution kicked into gear. In the late 1990s, a growing startup would need to raise literally millions of dollars simply to operate at scale, including funding for databases, application servers, testing, storage, networking gear, and marketing. At the time, hiring and staffing weren't the major costs; infrastructure was. Even developer tools were a significant expense, costing hundreds if not thousands of dollars. The Eclipse project—a free, open-source IDE—wasn't launched until 2001. Mercury Interactive was charging customers hundreds of thousands of dollars for licenses to use its testing products for e-commerce applications. Infrastructure abundance enabled and required a change in working practices.

The Agile Manifesto was published in 2001, but the concepts introduced in that movement became widely adopted as the cloud took off.

> *Individuals and interactions over processes and tools*
> *Working software over comprehensive documentation*
> *Customer collaboration over contract negotiation*
> *Responding to change over following a plan*
> *That is, while there is value in the items on the right, we value the items on the left more.*[2]

All of this comes from an abundance mindset. When Jez Humble and David Farley introduced the concept of deployment pipelines in *Continuous Delivery*, published in 2010, the cloud was just taking off.[3] In the intervening years, abundance has supercharged those practices.

In 2005, Daniel Terhorst-North and Jez Humble introduced the idea of blue-green deployments as a response to a client having significantly different test and production environments.[4] They wanted to be able to smoke test in one environment, the "shadow," which replicated production, before moving workloads over. This approach came from an abundance mindset and is now a common practice thanks to cloud abundance and automation capabilities. The cloud doesn't just enable abundance; it does so with powerful automation built in.

Abundance and automation enable Progressive Delivery by providing new ways of working.

Virtualization, Abundance, and Cloud Computing

The mainstream availability of virtualization was another jerk for software developers. While virtualization was initially positioned for IT efficiency, driving greater resource utilization, it also underpinned a new approach to resource availability. The same server could be used for development, test, QA, or production, so separate teams were not fighting over scarce resources. Organizations also began to collapse functional silos. Plus, sophisticated automation meant environments could be treated as ephemeral rather than built for (long-term) purpose. Automation enabled abundance.

This trend accelerated and expanded with the emergence of cloud computing. While the cloud was originally composed of virtual machines, we now also use container-based architectures, enabling ever-greater granularity of compute resources.

As resources became more abundant, the ability to incorporate software into everything became more economical. The cost of developing more value and delivering it to users dropped precipitously. A single individual can build and deploy an application. That autonomy and agency are now a baseline expectation for software, not an exception. What became more

important was understanding user needs and figuring out how to meet them in a commercially viable way.

What About On-Prem?

The cloud is an exemplar of digital abundance, but sometimes costs are hidden. Many organizations are currently considering repatriating some workloads because they are concerned with performance or the spiraling costs of cloud hosting.

When we consider abundance in the context of Progressive Delivery, two critical factors emerge: First, simplified management is a form of abundance. The cloud doesn't just mean more infrastructure, but more managed services as well. If developers don't have to worry about how to manage databases, then they have more choice and capability available to focus on the way they are adding business value.

Second, the evidence indicates that cloud infrastructure provides the optimal environment for Progressive Delivery. The cloud is the key underpinning for abundance, autonomy, and automation. No other platform comes close.

Though organizations can implement Progressive Delivery patterns and practices using on-premises infrastructure, the cloud—as delivered by hyperscalers such as AWS, Microsoft Azure, and Google Cloud—is the most flexible environment for software delivery. In terms of automation capability, network routing, and the flexibility to clone and fork infrastructure, the cloud is more flexible than on-premises.

For organizations with on-premises requirements, Kubernetes and container-based infrastructures provide a viable alternative (they're called "cloud native" for a reason). While it's certainly possible to implement Progressive Delivery patterns using these technologies alongside modern automation tools like Ansible and HashiCorp Terraform, the effort is substantial. Platform engineering teams must build and maintain much of what cloud providers deliver as managed services. This additional work—creating infrastructure, managing scaling, implementing security—represents significant overhead that detracts from focusing on customer value.

The willingness to embrace cloud services stands as a strong indicator of an organization's commitment to abundance thinking. It signals a prioritization of developer productivity and innovation speed over traditional infrastructure control patterns. Even in organizations that have physical or security constraints, virtualized systems and containers allow for more flexibility outside of a capital-expenditure budget.

There will always be outliers, and if your business is operating physical infrastructure at the scale of a public cloud provider, your teams will definitely benefit from the same Progressive Delivery practices.

Key Principles and Applications

In physics, potential energy becomes useful only when it's transformed into kinetic energy, ideally with a controlled, reliable flow. Abundance transforms the physics of software delivery in much the same way that modern electrical grids transformed society. It's not merely about generating more power; it's about fundamentally changing how that power is distributed, regulated, and used.

Abundance includes tangible resources like compute, storage, and bandwidth, but its true power comes from the transformative shift in mindset from "Why do you need that?" to "Is there any reason you shouldn't have that?" This represents a profound rebalancing of forces in the system. Just as an electrical grid with sophisticated transformers and load balancers can maintain steady power through demand spikes without brownouts, organizations with abundance thinking can absorb the jerks of rapid innovation without disrupting their forward momentum.

In the resource-scarce past, restricting access made economic sense. Like power rationing during shortages, the friction of approval processes protected valuable assets. Today, however, the cost of testing often exceeds the cost of the resources themselves. When a "test machine" represented hardware worth more than a developer's weekly compensation plus dedicated internal support, careful gatekeeping was justified. Now, when the same capability might represent just $0.73 of a multi-thousand-dollar

cloud invoice, the friction of approval creates unnecessary resistance, like forcing users to file paperwork before turning on a light switch.

The following principles exemplify how abundance thinking transforms the physics of software delivery, creating a reliable power grid of innovation that delivers consistent value while smoothing out potentially disruptive technological jerks.

From Getting to Using

When resources are scarce, organizations expend enormous energy simply acquiring what they need—the "getting" phase consumes attention, budget, and time. Abundance fundamentally shifts this equation. Sufficiency lets us change our focus from *getting* to *using*—from acquiring infrastructure to creating value with it. It's the difference between struggling to generate enough electricity and being able to focus on what you can build with reliable power.

As technology matures, it changes from an end in itself to a way to get things done. We shouldn't think about infrastructure itself, but rather how to use it to build an app that delivers value to users. The existence of new abundance, like the cloud, puts this kind of thinking into stark relief.

AWS talks about avoiding undifferentiated heavy lifting as a core principle. From the *AWS Well-Architected Framework*:

> *Stop spending money on undifferentiated heavy lifting: AWS does the heavy lifting of data center operations like racking, stacking, and powering servers. It also removes the operational burden of managing operating systems and applications with managed services. This permits you to focus on your customers and business projects rather than on IT infrastructure.*[5]

We don't need to *get* information technology (IT) because IT is all around us. The question is how we use IT to get from A to B to C, how we make progress in delivering applications and services. "Getting" is undifferentiated heavy lifting. "Using" is creating new services and new value for customers.

Abundance and Alignment: Giving the User Options

In our equation of how the four A's balance, we use alignment (along with automation) to constrain abundance and autonomy from runaway growth. This is intended to help teams avoid building beyond the needs of the users and delivering features that never get used. Progressive Delivery can also include putting the user in charge of when they choose to adopt a new service. IT creates options alongside product management, but the user decides when to adopt them.

Just as modern electrical systems offer user-controlled switches rather than centrally regulated power, Progressive Delivery separates the availability of features from their activation. Imagine if the power company controlled the dimmer switch in your living room. Instead, we generate and transmit the capability, but users control when to flip the switch.

For example, software developers can use a blue-green deployment to test new services before moving all customers over to them. This capability also enables product teams to strategically roll out features to different user segments. Smart organizations today increasingly allow users to decide when they start using a service or feature.

Google introduced Gmail Labs on June 5, 2008—an option in Gmail that allowed users to test new features and provide feedback to Google. This was a fundamental step forward in cloud-based product management.

More recently, Microsoft has adopted similar approaches. Outlook, for example, now has a "Try the new Outlook" toggle in the upper-right corner of the classic Outlook window. Here, the user is firmly in charge of when and how they adopt a set of new features. This is a great example of modern Progressive Delivery practices.

With Atlassian, for some new features (like new boards and issue transitions), users can opt into the new experience (and give feedback) or stay in the classic mode for a period of time. Atlassian did a great job of transitioning users from the old issues editor to the new using Progressive Delivery patterns, including phased rollouts and extensive user feedback.

The good news? These same experimental patterns pioneered by major corporations are accessible to everyone. What once required massive engineering investments has become standardized practice, with powerful

platforms making implementation straightforward. The automation infrastructure has evolved dramatically, transforming capabilities that teams once had to build from scratch. Feature flagging systems, for instance, have gone from custom-built solutions to robust, off-the-shelf products with thriving ecosystems.

We can all take advantage of abundance.

The bad news? Today, many of us still force updates on users that annoy them at best and, at worst, break the core experience entirely. In January 2025, Sonos CEO Patrick Spence was forced to resign after an app update in 2024 broke core user workflows, such as sleep timers, adding songs to a queue, and managing WiFi connectivity.[6] Users were rightly furious, responding to a fundamental misalignment between the brand and the market.

While it's clear we should use feature flags and give users options, abundance goes even further. With abundant resources, running two service versions simultaneously becomes a real possibility. You can maintain the current version for mainstream users while offering the next version to early adopters and supporting the legacy version for those who aren't ready to migrate.

This transforms Progressive Delivery into a strategic service management approach. We gain the power to be deliberate about managing technical debt, gracefully sunsetting outdated features rather than abruptly removing them. These decisions shift from a purely technical concern to a business alignment question.

At its core, Progressive Delivery puts users in control of their experience. When users complain about forced updates, they're often expressing frustration at their lack of agency. By giving them a choice in when and how they adopt changes, we create happier users and opportunities for new business models built on flexible, user-driven rollouts.

Abundance, Elasticity, and Optionality

In a world of digital abundance, we ask questions that embrace a variety of possibilities and users. We've moved from simple motion to complex adaptive systems that can absorb and dampen technological jerks. Thus,

we can build with customization and optionality in mind, understanding that resources are not universal, and users may indeed be resource constrained.

Abundance creates technological shock absorbers—we expect spikes in usage but also work to optimize and harmonize workloads when we see that work focus has shifted. We sometimes shut things down because abundance is most effective when we can clearly distinguish between what drives us toward our goals and what no longer delivers value. This adaptability allows our systems to maintain steady momentum even when faced with external forces trying to jerk them in different directions.

In an abundance world, we don't need to choose between A and B; instead, we can test an entire range of possible outcomes or options simultaneously. In a world with elastic response to demand, you don't need to own a whole datacenter to handle the spike in traffic from the Super Bowl. Instead, you can rent that capacity from a content delivery network (CDN) as you need it, both around predictable and unpredictable events.

Elastic scale serves in both building and operating software. During development, it enables thorough testing with production-like loads—a capability previously unimaginable for most organizations. Consider the challenge of stress-testing a major system. How do you create a production replica of a major production system and then generate a huge amount of load across it? With cloud abundance, you can spin up environments on demand without massive capital investment.

An abundant software world isn't just about raw capacity and volume—it's directed capacity. Like an electrical grid that doesn't just generate power but delivers it precisely where and when needed, abundance provides both the resources and the frameworks to guide their use. Opinionated guidance and well-established patterns give teams confidence when doing novel or unusual things, ensuring they are heading in the right direction.

Recognizing Abundance Constraints

Despite this progress and abundance, not all organizations have a true abundance mindset. You can spot this by examining how decisions are

made in your environment. When teams must justify small experiments, when accessibility features are dismissed as "too expensive," or when time tracking becomes more important than user outcomes, these are symptoms of scarcity thinking, not an abundance mindset.

The most telling example is in how your organization responds to new ideas. If the immediate reaction is "Can we do that? Is it allowed?" then you're likely operating without an abundance mindset. Other signs of low abundance might include:

- a heavy process burden in requesting additional resources
- exaggerated organizational fear of (small) failure
- resistance to any change because of potential costs
- zero-sum thinking—"If that person gets more, I will get less."

If you are in an organization with abundance constraints, do what you can to create local abundance. If the organization is too resistant to creating abundance, they are probably not organizationally prepared to work toward Progressive Delivery.

Abundance Is Additive

Another aspect of software abundance is that we all stand on the shoulders of giants. As software developers, we depend on languages, packages, libraries, and other dependencies that have been written by others. It doesn't make sense to write our own stacks, from the machine code on up, when the software is available to us at our fingertips. In fact, few of us write compilers or new programming languages.

At every level of software, we are building on the work of others, and we benefit from the abundance of this work. Because we can use the work and standards of others, our software fits with other software, and we don't have to re-create it. Like the modern electrical grid, we don't generate our own power or build transformers from scratch; we simply connect to standardized outlets, focusing our energy on what we'll do with that power.

Consider a typical web application today: It might use React for the front end (created by Facebook), run on Node.js (developed by Joyent), store data in MongoDB (from MongoDB Inc.), authenticate users with Auth0 (by Okta), process payments via Stripe, and deploy automatically through GitHub Actions. Each component represents thousands of engineering hours that the development team doesn't need to replicate. Abundance means there is almost always code at hand for solved problems, which means we are free to work on unsolved problems—the unique value our application provides to users.

AI is accelerating this abundance dramatically. Generative AI (GenAI) is itself based on abundance. Large language models (LLMs) were trained on vast datasets, and now they're generating code on behalf of developers. AI is not just finding software to build solutions with; it's generating the solutions themselves. Some people flinch away from the thought of having so many dependencies on other teams and companies—and indeed LLMs—but we already exist in a mesh of dependencies

This abundance also creates challenges, such as the danger of potential vulnerabilities in third-party libraries. But Progressive Delivery allows security testing to be added to our pipelines before deployment and again before rollout. It's another layer of security. Management of dependencies is a key aspect of modern software testing, and Progressive Delivery maps to it quite well.

Benefits of Abundance

Engineering teams once had limited access to the software they needed to build and deploy new services and applications. This software is now effectively free, available on GitHub and other repositories, with marginal costs of zero. This transition is akin to moving from a system with high friction and little mass to one with low friction and increasing mass, resulting in significant momentum. Once set in motion, development becomes harder to stop than to continue.

Distributed version control systems have removed developer dependence on central repositories, again enabling abundance. The availability of managed services means we're not even constrained by the cost of managing infrastructure. Cloud resource limitations are effectively a thing of the past—unless you count cost. This abundance has enabled entirely new ways of working—a fundamental change in the physics of development that transformed jerky stop-start motion into smooth continuous delivery.

We can measure abundance both quantitatively and qualitatively. A quantitative measurement would be how long it takes to go from request to allocation for a particular resource. What resources are available to developers and developer teams? A qualitative measurement uses surveys and interviews to ask developers if they have the resources they need to accomplish their tasks.

Today, Apple sets the bar for local machine performance. A common proxy for abundance is the availability of recent model MacBook Pros. (If your developers can't use the machines they want for work, then abundance may be in question.) Of course, some developers and organizations prefer Windows, and that's totally fine. In that case, can a developer get the latest AMD or Intel processors, or even ARM-based machines, and all the RAM they desire?

Abundance within boundaries does not mean abundance without constraints. You can't build anything without understanding constraints and making trade-offs. These constraints are often expressed as costs, but the classic aphorism "Good. Cheap. Fast. Pick two." is another expression of constraints. Time is inelastic, and there are some things we can't just pay to speed up—some jerks in the system cannot be entirely eliminated, only managed.

But abundance frees software developers to do their best work. It removes the need to wait for permission. Organizations should get out of the way, where possible, and allow builders to build.

This tension between the concept of instantly responsive software and the time, effort, resources, and underlying physical systems that support it is the core of Progressive Delivery's problem. What is meaningful to

deliver? What adds value? What supports the other parts of the structure? What do we need to change and streamline to iteratively improve the act of delivering and the experience of receiving?

Abundance enabled a Cambrian explosion that has changed how we think about software and product delivery. Organizations don't have a single integrated monolithic technology stack and may not even have a central technology administration. Instead, different parts of the organization solve the problem at hand that is closest to them, without needing to ask for permission. Abundance enables autonomy and radical delegation.

Challenges and Considerations of Abundance

Abundance can come with its own problems. When something is cheap, we tend to value it less. Abundance can also lead to problems of scale in disposal and management. Abundance creates its own form of inertia—objects in motion tend to stay in motion, even when that motion is no longer serving our goals. Just as a heavy vehicle with momentum requires more sophisticated braking systems, our systems of abundance require more sophisticated governance to prevent runaway acceleration.

Abundance can lead to carelessness in how we use resources. In the boom times, when resources are cheap, we don't meter them. Then, when resources are more constrained, we don't have the systems to use them efficiently. People raised in well-watered areas do not build the habits of water conservation that people raised in drought areas do. In a ZIRP software boom, there is little incentive to cap spending on resources since the focus is on growth.

Without proper control systems, the technological momentum we've built can cause destructive jerks as competing forces pull in different directions. Abundance is great, but it comes at a cost, even if the costs seem lower. For example, what will it cost to move or transfer your data from one vendor to another? Will your abundance prove illusory in the future when your development or deployment stack changes?

An organization that doesn't track cloud spending allows users to take advantage of digital abundance but is wasting money that could be better spent. Abundance and autonomy can lead to runaway spending. The cloud, for example, which began as a phenomenon driven by individuals with credit cards, is now a trillion-dollar industry. It has even spawned a FinOps foundation to help organizations spend wisely in abundance settings. FinOps being:

> *an operational framework and cultural practice which maximizes the business value of cloud, enables timely data-driven decision making, and creates financial accountability through collaboration between engineering, finance, and business teams.*[7]

While we're not going to delve deeply into cost management here—there are many other great books on the subject—it's worth noting how the organization talks about its role.

> *FinOps is all about removing blockers; empowering engineering teams to deliver better features, apps, and migrations faster; and enabling a cross-functional conversation about where to invest and when. Sometimes a business will decide to tighten the belt; sometimes it'll decide to invest more. But now teams know why they're making those decisions.*[8]

Another way to look at abundance and runaway costs is the current debate about cloud repatriation. Some now argue that running on-premises infrastructure is cheaper than using hyperscale cloud services. This idea was expressed most pithily by venture capital firm Andreessen Horowitz in a 2022 post, "The Cost of Cloud, a Trillion Dollar Paradox," which claimed: "You're crazy if you don't start in the cloud; you're crazy if you stay on it."[9]

Whether you agree with this thesis or not, it gets to the paradox of cloud abundance. Convenience can increase direct costs, so it's important to be intentional. If abundance enables Progressive Delivery, your ability to get the right product to the right customer at the right time, then that's

worth investing in. In some cases, enterprises will decide these costs are not, in fact, worth it. For example, in late 2024, GEICO announced a significant cloud repatriation effort.[10]

Not all costs are monetary, such as environmental impact and access problems that are not role-based. Not everyone has the same access to the servers and bandwidth that technologists often take for granted. Just because it worked on your WiFi network doesn't mean it will work well, and at a reasonable cost, in all parts of the world. Progressive Delivery can enable you to understand differences in infrastructure ubiquity and work with them, testing in different regions and on different networks. Operating in a gracefully degraded state is an important way to make sure software is accessible to as many people as possible.

Abundance also often leads to data management problems. The instinct to "store everything" doesn't necessarily improve analysis quality and often increases cost.

Think of your data lake as an actual hydroelectric reservoir. When properly channeled, it generates tremendous power for your organization. But just like a real dam, sediment accumulates over time. Without proper management, your data lake fills with silt—outdated information, duplicate records, and irrelevant metrics—making your data lake shallower and less valuable. Just as reservoir managers must properly control flow and waste, data stewards must establish retention policies and quality controls.

The challenge is compounded because the inflow of data is only partially under our control, and its original quality varies widely. With abundance, the question shifts from "Can we store this?" to "Should we store this, and for how long?" Observability provides a clear example of the challenges of abundance. While unlimited data collection offers unprecedented insights, it comes with substantial costs, particularly when dealing with high-cardinality datasets.

High-cardinality fields—attributes like userIds or shoppingCartIds that might have hundreds of thousands of unique values—can dramatically increase storage requirements and processing overhead. When organizations complain about excessive charges from observability vendors, they're

often experiencing the downside of abundance thinking: collecting everything without strategic filtering. The issue isn't necessarily the vendor's pricing model, but rather the absence of thoughtful indexing strategies or the accumulation of data that provides minimal analytical value.

Abundance made modern observability possible in the first place. It enabled the collection and analysis of logs, metrics, and system traces at a scale previously unimaginable. However, this capability shift created a new pain point where observability vendors now compete primarily on cost efficiency rather than just feature sets.

This balance between data abundance and cost management is particularly critical for Progressive Delivery. Observability provides an essential feedback loop when testing services in production through feature flags or canary deployments. Without comprehensive and queryable monitoring, teams lack the confidence to implement progressive rollout strategies.

The observability industry has responded to this tension with innovative approaches. Increasingly, observability platforms are being built on efficient open-source data lakes like ClickHouse or proprietary platforms like Snowflake. These solutions enable cost-effective querying across commodity object storage instead of relying on specialized (and expensive) time-series databases—another example of abundance driving innovation in response to its own challenges.

Getting Started with Abundance

In a world where technological jerks have become the norm—where software updates can disrupt workflows without warning, and new platforms emerge seemingly overnight—abundance offers both a challenge and a solution. The same acceleration that creates jarring experiences for users can also provide the resources to smooth these transitions.

Software now permeates nearly every aspect of our lives, from morning alarms to evening entertainment. Each interaction represents a potential moment of technological jerk—an unexpected acceleration that can either

delight or disorient. Progressive Delivery helps manage these moments by leveraging abundance not just as raw computing power, but as a comprehensive approach to change management.

The abundance mindset transforms the fundamental question from "Do we have enough resources?" to "How can we best direct our virtually unlimited resources to create smooth, controlled acceleration rather than jarring jerks?" As we build software, we need to think past pure capacity and bring in the wisdom to build in ways that respect users' need for stability amid innovation—that is, how to build the right thing for the right people at the right time.

Evaluating for Abundance

As you begin to look at your software development practices through the lens of Progressive Delivery, understanding abundance is important. What *does* abundance mean in your organization?

Here are some questions to consider as you evaluate and work to better understand what abundance means for your team:

- What is the most constraining factor in your environment?
- How much time does it take to provision a resource?
- What is the cost of doing something in time, worked hours, or money? How often is this activity performed per day/week? By how many people?
- How do you handle excess capacity?
- If you had infinite capacity in one place, where would you put it?
- What do you rely on that is mission-critical?
- What is your fail-safe mode? If something goes wrong, what happens?
- What are your core dependencies? Which are homegrown versus outsourced? (What do you build versus buy?)
- Do you offer developers choice and budget in their tooling, and do you have constraints on the interoperability of their choices?
- Do you encourage developers to use AI tools, and do you provide a budget accordingly?

- What is the abundance you are building versus the abundance you're renting?
- Is there a way to prune things automatically without repeated human cognitive cost?
- How would you deliver to a mobile app that doesn't always have internet? (Abundance is not always universal.)
- Are you limited by what you can do or what it would cost to do it?

Tools and Processes That Enable Abundance

Many organizations use the following tools and patterns to help them successfully manage abundance. For those interested in furthering their practices, here are some to consider:

- cloud-native computing
- elastic scaling
- open-source software
- observability
- release progression
- testing in production
- blue-green deployments
- A/B testing and experimentation

While this list is by no means exhaustive, it is a starting place to explore as you consider what abundance means for you. Furthermore, not all of these will necessarily solve every use case. Rather, it's important to evaluate your needs using the list of questions we shared earlier. From there, you can begin to explore what tools and practices will support you in your efforts.

Conclusion

It's important to remember that abundance through the lens of Progressive Delivery is not using everything all at once. It is accurately understanding specific needs and building for those needs. Abundance is not just capacity

and volume; it's the ability for capacity and volume to be well-directed. This is alignment on the builder/developer side of the equation. We have chosen to incorporate this into abundance since this type of alignment is more about the resources used to build a company, and not about the software being delivered to users.

Just as a skilled driver harnesses the momentum of a vehicle to navigate smoothly without jarring accelerations or jerky stops, Progressive Delivery harnesses the inertia of technological abundance to deliver change in a way that users can absorb. It transforms the potentially disruptive force of rapid technological change into a smooth, controlled acceleration that propels organizations forward without throwing their users off balance. In a world where technological jerk has become commonplace, abundance, properly managed, becomes a stabilizing force that allows us to move quickly without losing our traction.

Now that we've explored the concept of abundance in Progressive Delivery, let's explore a case study that illustrates these principles in practice. We'll see how abundance, generally, and the cloud, specifically, enabled software organizations to do things that were not possible before.

Chapter 3

CASE STUDY: SUMO LOGIC

Sumo Logic is a great example of a company built as cloud-enabled abundance arrived on the market, which influenced all of their decision-making and architectural approaches. It was founded in 2010 by a team with experience in log management, big data, and security. They set out to create a cloud-first software-as-a-service (SaaS) log analytics company, built on AWS and designed to monitor events generated by cloud-based services.

Cloud-based monitoring is different from traditional on-premises approaches because you don't have direct access to hardware metrics, because, for example, the servers are running in the cloud. APIs, however, are publishing huge amounts of data about system and application performance. In this example, data became the problem rather than instrumentation, so the company focused squarely on data management. Data volumes led to an abundance mindset, which also played into the company becoming an advanced Progressive Delivery case study.

Sumo Logic was acquired by Francisco Partners in a private equity transaction valued at $1.7 billion in February 2023. By then, it had carved out a solid position as a leader in cloud-based log management.

Situation

The first AWS primitives arrived in 2006. Launched in 2010, Sumo Logic was in a position to build an architecture from scratch on top of AWS, which

was swiftly maturing. Sumo Logic used its own product to provide observability, enabling feedback loops as it built, tested, and deployed new services.

This timing is significant in the context of abundance. As discussed in Chapter 2, the software industry has been shifting from a scarcity mindset to an abundance mindset since the late 1990s. Sumo Logic emerged at a perfect moment to take full advantage of the abundance the cloud provided, without the legacy constraints that hampered established enterprises.

AWS was a capable platform, but Sumo Logic still had to build a lot of its own infrastructure. For example, it built its own feature flag system and even its own infrastructure-as-code provisioning system. So, while it was building a very sophisticated automated infrastructure for building, testing, and deploying the platform, it was also incurring a fair amount of technical debt. This represents the "getting to using" transition description in Chapter 2. Building their own tools was still necessary, but they were focusing on how to use infrastructure rather than simply acquiring it.

There were several key architectural decisions at Sumo Logic that enabled Progressive Delivery and testing in production, including:

- Adopting a service-oriented architecture (SOA) approach, with loosely coupled services that could be updated and scaled independently. This provided flexibility for progressive rollouts.
- Implementing feature flags and shadow deployments to test changes in production without impacting all customers. This allowed Sumo Logic to experiment and validate changes before full rollouts.
- Focusing on observability and cost optimization to understand the impact of changes and manage the costs of the cloud infrastructure.

These decisions directly embody the key principles and applications of abundance. The SOA approach means different services can scale and be tested independently while also allowing multiple versions of a service to run simultaneously. Sumo Logic practices canary deployment and then

looks at what sort of customers choose to use the new feature. This practice is made possible by their extensive use of feature flags.

As Bruno Kurtic, cofounder of Sumo Logic, told us: "We roll out a new service to 5% of our customers first. What sort of users choose to use this feature? We roll out the service then leverage our logs to understand the behaviors of the system and users. Logs are integral to understanding how new code is being shipped, how you do A/B testing in production. We do testing in production."

Finally, Sumo Logic's observability focus means it is constantly using feedback loops, understanding user behaviors, and adjusting system behavior accordingly. A key lesson here is that logs are integral to understanding how new code is being shipped and can underpin A/B testing in production, aligning the needs of users, developers, and product owners. Sumo Logic's approach, and focus on optionality and observability, underpinned by the abundance of system resources, enables them to align releases with actual user needs and behaviors.

Giving Developers Their Own Production Infrastructure

One of the best examples of cloud-enabled abundance at Sumo Logic was how it provided images to developers. Technical leadership wanted to avoid "But it worked on my machine." finger-pointing between operations and developers, so the development environment had to be as close to production as possible. Therefore, Sumo Logic built a minimal layer for "personal deployments" on AWS that allowed developers to easily test their code in what was effectively a production environment, including all of the microservices—a "mini-Sumo."

This approach perfectly embodies "from getting to using." In the pre-abundance era, developers would have spent significant time acquiring and configuring test environments or working in test environments that did not closely mimic production. At Sumo Logic, they could work in an environment that was as close to production as possible.

Cloud abundance represents a dramatic transformation from the historical context, where development, test, QA, and production teams were

split into functional silos, each with their own infrastructure, and developers had local machines that didn't replicate these infrastructures at all.

This streamlined development environment greatly improved the developer experience and the ability to test changes. This abundance also empowered autonomy, allowing any developer to spin up a full Sumo Logic stack, ideally just for an hour or so.

However, giving all developers their own Sumo Logic could get expensive quickly if developers didn't turn these instances off. After all, abundance needs to be ephemeral to be cost-effective. At first, it was about reminding developers to turn these mini-Sumos off, but naturally, the company soon built a set of scripts, which became an internal app, to go out and kill clusters that weren't being used. They called it Reaper.

Autonomy driven by abundance is great, but automation was needed to keep things under control, enabling alignment between the needs of the engineer, the platform owner, and the CFO.

Cloud Bursts and Feature Optionality

Another facet of the need to maintain alignment between the business and the availability of resources is consumption-based scaling. Sumo Logic was architected to scale elastically, taking advantage of cloud resources as its customers' workloads grew. The cloud allows organizations to take advantage of hyperscaler abundance, even for customers that are extremely "bursty" from a workload perspective, such as online gaming companies. Think of the growth of Pokémon GO or Fortnite. Load testing should replicate this kind of workload fluctuation, where customers might create so much extra traffic that they effectively create a distributed denial of service (DDoS) traffic pattern by accident. But even with load testing, a truly unexpected success can drive resource utilization well above expectations.

Sumo Logic built feature optionality into its core architecture. In order to handle system load, Sumo Logic can turn any feature in its platform on or off. The company can also turn any feature in its platform on and roll it out to one specific customer in a particular region for a particular use case to test it before wider deployment. Here the cloud advantage underpin-

ning Progressive Delivery is about easy access to sophisticated networking, which is a form of abundance in its own right.

Progressive Delivery for Machine Learning

Driven by infrastructure abundance, Sumo Logic can conduct shadow tests of new machine learning models in production, something that would have been unthinkable in the pre-cloud era.

For example, a customer might complain that Sumo Logic's pattern recognition wasn't working. The danger here is that if the company changes the algorithm for other customers, it might break their experience. Therefore, Sumo Logic needed to silently spin up a couple of clusters and test the algorithm's performance.

Sumo Logic does candidate testing of each service it rolls out. To do this, they have a shadow copy of Sumo Logic that is used for testing, industry regulations, and so on. The entire infrastructure is replicated—this is literally testing in production, driven by abundance.

The clone was deployed in a different datacenter with a different set of engineers, which also created some interesting management overheads. This full system replication exemplifies the technological inertia we discussed in Chapter 2—the ability to build momentum and stability through abundance. By maintaining parallel systems, Sumo Logic creates a counterbalance to technological jerk, absorbing changes rather than being disrupted by them.

Complications

Abundance led to sprawl being a key issue, alongside technical debt. Thus, Sumo Logic made extensive use of feature flags. Over time, however, there were so many feature flags that the entire system became unwieldy. What began as a mechanism enabling flexibility became an issue for engineering. The Sumo Logic team ended up rewriting the feature flag system to make it better adapted to modern software engineering practices with version con-

trol and a Git-based workflow. Today, they would likely choose a packaged third-party feature flag solution. Not all abundance arrives at once, and any startup incurs technical debt.

By 2015, it was clear Sumo Logic needed to reduce infrastructure costs overall. So, it spun up a group, which included a data scientist, tasked with reducing infrastructure costs, and called them the Prosperity Team. This effort was a dramatic success, increasing margins for cost to serve from around 30% to over 70%.

Abundance always needs to be managed. You need to be intentional, or costs get out of control. The question becomes how to maintain cost controls and avoid sprawl while allowing abundance to underpin alignment with the business goals of a fast-growing startup.

Question

It's currently commonplace to say that every company is a software company. But if that's the case, there is a whole set of practices associated with being a software company that are really tough: managing open-source infrastructure at scale, or dealing with software dependencies, or keeping current with common vulnerabilities and exposures (CVEs). We all get blamed for poor customer experiences—software companies certainly do. But in terms of managing your own estate and identifying what is actually a competitive advantage, that's a thorny set of engineering questions with no simple set of answers.

As Christian Beedgen, one of Sumo Logic's founders, put it during an interview with James Governor in February of 2025,

> *Our declarative deployment system was a competitive advantage…until it wasn't. Because it was bespoke, and we had to maintain it. Over time Sumo Logic hired new people with a different set of expectations about industry standard infrastructure, such as HashiCorp or LaunchDarkly. These folks also had skills using these platforms. So, abundance, in the case of venture capital, meant Sumo Logic could do some incredible core engineering work.*

But it is possible to over-engineer things, and managing technical debt is always hard.[1]

Beedgen's observation highlights the transition Sumo Logic made from building its own provisioning and feature flagging management tools to adopting industry standards. This illustrates both the benefits of abundance and the challenges of maintaining bespoke solutions in an ecosystem increasingly built on shared platforms.

Summary: Abundance as the Organizational Forcing Factor

Abundance is a powerful forcing factor enabling new organizational practices, working methods, and workflows in tech. Two of the main beneficiaries of digital abundance are the developer and the engineering organization because of the autonomy it gives them. Abundance removes the need to ask for permission, removes bottlenecks, and allows engineers to get on with their work—no more time waiting for infrastructure to be provisioned. Of course, greater autonomy requires new management approaches to enable alignment, which we'll explore in upcoming chapters.

Sumo Logic exemplifies this transformation. Their "mini-Sumo" environments eliminated wait times for developers. Their elastic architecture removed the permission bottlenecks for scaling. Their shadow deployment capabilities allowed for testing without traditional approval gates. Each of these innovations demonstrates how abundance transformed the physics of their software delivery, from the jerky stop-start motion of the scarcity era to the smooth Progressive Delivery enabled by abundance.

Autonomy, derived from abundance, allows organizations to move faster, ship more products, and roll out new services more quickly. It also reduces the likelihood of burnout, by increasing agency for developers and users. This radical delegation, as explained more in future chapters, is a fundamental improvement in working culture.

The Sumo Logic case study provides a concrete example of both the possibilities and challenges of abundance as a foundational pillar of Progressive Delivery. Their journey from founding in 2010 to acquisition in 2023 spans the maturation of cloud abundance, demonstrating how organizations can harness technological inertia to create momentum while developing the necessary controls to prevent the destructive jerks of unmanaged acceleration.