Design change-tolerant software

Cloud Native Patterns

Selected Chapters
Cornelia Davis

From the team behind
ORACLE Linux
contents

foreword iv

about the author vi

1 You keep using that word: Defining “cloud native” 1
   1.1 Today’s application requirements 5
   1.2 Introducing cloud native software 8
   1.3 Cloud native and world peace 19

2 Running cloud native applications in production 24
   2.1 The obstacles 25
   2.2 The enablers 33

3 The platform for cloud native software 49
   3.1 The cloud(native) platform evolution 50
   3.2 Core tenets of the cloud native platform 55
   3.3 Who does what? 63
   3.4 More cloud native platform capabilities 66

index 79
Cornelia Davis’ *Cloud Native Patterns: Designing change-tolerant software* draws you into the world of next-generation application development. Through relevant, real-world examples delivered in a practical and engaging style, it offers insightful context for an ever-changing, highly distributed, maximum-availability model, more simply called cloud native computing.

Oracle is one of the few cloud vendors to also have a long history of providing enterprise software. Wearing both software developer and cloud service provider hats, we understand the complexity of transforming on-premises applications into cloud native applications. Removing that complexity for customers is a guiding tenet at Oracle.

This ebook, *Cloud Native Patterns*, is a three-chapter excerpt of *Cloud Native Patterns* that’s intended to help you navigate key elements of the cloud native landscape, determine potential pitfalls in your environment, and to streamline your cloud adoption.

In these three chapters, you’ll learn about important cloud native concepts such as data-driven applications and shorter user feedback cycles. You’ll also learn why repeatability and continuous delivery will help you succeed.

Building cloud native applications is a challenging undertaking, especially considering the rapid evolution of cloud native computing. But it’s also very liberating and rewarding. You can develop new patterns and practices where the limitations of hardware dependent models, geography, and size no longer exist. This new approach to technology will make cloud application developers more agile and efficient, even as it reduces deployment costs and increases independence from cloud service providers.

If you find this excerpt helpful, you’ll also find value in the Davis’ unexcerpted work, which provides more specific application of the patterns. It’s well worth your time to read.

Onward to cloud native.

Robert G. Shimp,
Group Vice President,
Product Management,
Infrastructure Software,
Oracle
Oracle Cloud Native Environment is a curated set of open source software selected from cloud native projects, integrated into a unified operating environment that includes:

- A rich set of software components for cloud native application development and deployment
- Certified Kubernetes and Kata Containers
- Enterprise-grade performance, scalability, reliability, and security

Oracle Cloud Native Environment runs on-premises or in the cloud. It is tested and proven with Oracle products such as Oracle Database, Oracle WebLogic Server, MySQL and more.

To find out more see oracle.com/linux/cloudnative and oracle.com/linux

Stay connected:

- blogs.oracle.com/linux
- facebook.com/OracleLinux
- twitter.com/OracleLinux
about the author

Cornelia Davis is Chief Technology Officer at Weaveworks. A teacher at heart, she’s spent the last 25 years making good software and great software developers.

Save 50% on this book – eBook or pBook. Enter mecnp50or in the Promotional Code box when you checkout. Only at manning.com.

Cloud Native Patterns
by Cornelia Davis
ISBN 9781617294297
400 pages
$39.99
You keep using that word: Defining “cloud native”

It’s not Amazon’s fault. On Sunday, September 20, 2015, Amazon Web Services (AWS) experienced a significant outage. With an increasing number of companies running mission-critical workloads on AWS—even their core customer-facing services—an AWS outage can result in far-reaching subsequent system outages. In this instance, Netflix, Airbnb, Nest, IMDb, and more all experienced downtime, impacting their customers and ultimately their business’s bottom lines. The core outage lasted about five hours (or more, depending on how you count), resulting in even longer outages for the affected AWS customers before their systems recovered from the failure.

If you’re Nest, you’re paying AWS because you want to focus on creating value for your customers, not on infrastructure concerns. As part of the deal, AWS is responsible for keeping its systems up, and enabling you to keep yours functioning as well. If AWS experiences downtime, it’d be easy to blame Amazon for your resulting outage.

But you’d be wrong. Amazon isn’t to blame for your outage.

Wait! Don’t toss this book to the side. Please hear me out. My assertion gets right to the heart of the matter and explains the goals of this book.

First, let me clear up one thing. I’m not suggesting that Amazon and other cloud providers have no responsibility for keeping their systems functioning well; they obviously do. And if a provider doesn’t meet certain service levels, its customers can and will find alternatives. Service providers generally provide service-level
agreements (SLAs). Amazon, for example, provides a 99.95% uptime guarantee for most of its services.

What I’m asserting is that the applications you’re running on a particular infrastructure can be more stable than the infrastructure itself. How’s that possible? That, my friends, is exactly what this book will teach you.

Let’s, for a moment, turn back to the AWS outage of September 20. Netflix, one of the many companies affected by the outage, is the top internet site in the United States, when measured by the amount of internet bandwidth consumed (36%). But even though a Netflix outage affects a lot of people, the company had this to say about the AWS event:

Netflix did experience a brief availability blip in the affected Region, but we sidestepped any significant impact because Chaos Kong exercises prepare us for incidents like this. By running experiments on a regular basis that simulate a Regional outage, we were able to identify any systemic weaknesses early and fix them. When US-EAST-1 became unavailable, our system was already strong enough to handle a traffic failover.1

Netflix was able to quickly recover from the AWS outage, being fully functional only minutes after the incident began. Netflix, still running on AWS, was fully functional even while the AWS outage continued.

NOTE How was Netflix able to recover so quickly? Redundancy.

No single piece of hardware can be guaranteed to be up 100% of the time, and, as has been the practice for some time, we put redundant systems are in place. AWS does exactly this and makes those redundancy abstractions available to its users.

In particular, AWS offers services in numerous regions; for example, at the time of writing, its Elastic Compute Cloud platform (EC2) is running and available in Ireland, Frankfurt, London, Paris, Stockholm, Tokyo, Seoul, Singapore, Mumbai, Sydney, Beijing, Ningxia, Sao Paulo, Canada, and in four locations in the United States (Virginia, California, Oregon, and Ohio). And within each region, the service is further partitioned into numerous availability zones (AZs) that are configured to isolate the resources of one AZ from another. This isolation limits the effects of a failure in one AZ rippling through to services in another AZ.

Figure 1.1 depicts three regions, each of which contains four availability zones. Applications run within availability zones and—here’s the important part—may run in more than one AZ and in more than one region. Recall that a moment ago I made the assertion that redundancy is one of the keys to uptime.

In figure 1.2, let’s place logos within this diagram to hypothetically represent running applications. (I have no explicit knowledge of how Netflix, IMDb, or Nest have deployed their applications; this is purely hypothetical, but illustrative nevertheless.)

---

See “Chaos Engineering Upgraded” at the Netflix Technology blog (http://mng.bz/P8rn) for more information on Chaos Kong.
Figure 1.1. AWS partitions the services it offers into regions and availability zones. Regions map to geographic areas, and AZs provide further redundancy and isolation within a single region.

Figure 1.2. Applications deployed onto AWS may be deployed into a single AZ (IMDb), or in multiple AZs (Nest) but only a single region, or in multiple AZs and multiple regions (Netflix). This provides different resiliency profiles.

Figure 1.3 depicts a single-region outage, like the AWS outage of September 2015. In that instance, only us-east-1 went dark.

In this simple graphic, you can immediately see how Netflix might have weathered the outage far better than others companies; it already had its applications running in other AWS regions and was able to easily direct all traffic over to the healthy instances. And though it appears that the failover to the other regions wasn’t automatic, Netflix
had anticipated (even practiced!) a possible outage such as this and had architected its software and designed its operational practices to compensate.  

NOTE Cloud native software is designed to anticipate failure and remain stable even when the infrastructure it’s running on is experiencing outages or is otherwise changing.  

Application developers, as well as support and operations staff, must learn and apply new patterns and practices to create and manage cloud native software, and this book teaches those things. You might be thinking that this isn’t new, that organizations, particularly in mission-critical businesses like finance, have been running active/active systems for some time, and you’re right. But what’s new is the way in which this is being achieved.  

In the past, implementing these failover behaviors was generally a bespoke solution, bolted on to a deployment for a system that wasn’t initially designed to adapt to underlying system failures. The knowledge needed to achieve the required SLAs was often limited to a few “rock stars,” and extraordinary design, configuration, and testing mechanisms were put in place in an attempt to have systems that reacted appropriately to that failure.  

The difference between this and what Netflix does today starts with a fundamental difference in philosophy. With the former approaches, change or failure is treated as an exception. By contrast, Netflix and many other large-scale internet-native companies, such as Google, Twitter, Facebook, and Uber, treat change or failure as the rule.  

---

2 See “AWS Outage: How Netflix Weathered the Storm by Preparing for the Worst” by Nick Heath (http://mng.bz/J8RV) for more details on the company’s recovery.
Today’s application requirements

These organizations have altered their software architectures and their engineering practices to make designing for failure an integral part of the way they build, deliver, and manage software.

**NOTE**  Failure is the rule, not the exception.

### 1.1 Today’s application requirements

Digital experiences are no longer a sidecar to our lives. They play a major part in many or most of the activities that we engage in on a daily basis. This ubiquity has pushed the boundaries of what we expect from the software we use: we want applications to be always available, be perpetually upgraded with new whizbang features, and provide personalized experiences. Fulfilling these expectations is something that must be addressed right from the beginning of the idea-to-production lifecycle. You, the developer, are one of the parties responsible for meeting those needs. Let’s have a look at some key requirements.

#### 1.1.1 Zero downtime

The AWS outage of September 20, 2015, demonstrates one of the key requirements of the modern application: it must always be available. Gone are the days when even short maintenance windows during which applications are unavailable are tolerated. The world is always online. And although unplanned downtime has never been desirable, its impact has reached astounding levels. For example, in 2013 Forbes estimated that Amazon lost almost $2 million during a 13-minute unplanned outage.\(^3\) Downtime, planned or not, results in significant revenue loss and customer dissatisfaction.

But maintaining uptime isn’t a problem only for the operations team. Software developers or architects are responsible for creating a system design with loosely coupled components that can be deployed to allow redundancy to compensate for inevitable failures, and with air gaps that keep those failures from cascading through the entire system. They must also design the software to allow planned events, such as upgrades, to be done with no downtime.

#### 1.1.2 Shortened feedback cycles

Also of critical importance is the ability to release code frequently. Driven by significant competition and ever-increasing consumer expectations, application updates are being made available to customers several times a month, numerous times a week, or in some cases even several times a day. Exciting customers is unquestionably valuable, but perhaps the biggest driver for these continuous releases is the reduction of risk.

From the moment that you have an idea for a feature, you’re taking on some level of risk. Is the idea a good one? Will customers be able to use it? Can it be implemented in a better-performing way? As much as you try to predict the possible outcomes, reality

---

3 See “Amazon.com Goes Down, Loses $66,240 Per Minute” by Kelly Clay at the Forbes website for more details ([http://mng.bz/wEgP](http://mng.bz/wEgP)).
is often different from what you can anticipate. The best way to get answers to important questions such as these is to release an early version of a feature and get feedback. Using that feedback, you can then make adjustments or even change course entirely. Frequent software releases shorten feedback loops and reduce risk.

The monolithic software systems that have dominated the last several decades can’t be released often enough. Too many closely interrelated subsystems, built and tested by independent teams, needed to be tested as a whole before an often-fragile packaging process could be applied. If a defect was found late in the integration-testing phase, the long and laborious process would begin anew. New software architectures are essential to achieving the required agility in releasing software to production.

1.1.3 Mobile and multidevice support

In April 2015, Comscore, a leading technology-trend measurement and analytics company, released a report indicating that for the first time, internet usage via mobile devices eclipsed that of desktop computers. Today’s applications need to support at least two mobile device platforms, iOS and Android, as well as the desktop (which still claims a significant portion of the usage).

In addition, users increasingly expect their experience with an application to seamlessly move from one device to another as they navigate through the day. For example, users may be watching a movie on an Apple TV and then transition to viewing the program on a mobile device when they’re on the train to the airport. Furthermore, the usage patterns on a mobile device are significantly different from those of a desktop. Banks, for example, must be able to satisfy frequently repeated application refreshes from mobile device users who are awaiting their weekly payday deposit.

Designing applications the right way is essential to meeting these needs. Core services must be implemented in a manner that they can back all of the frontend devices serving users, and the system must adapt to expanding and contracting demands.

1.1.4 Connected devices—also known as the Internet of Things

The internet is no longer only for connecting humans to systems that are housed in and served from data centers. Today, billions of devices are connected to the internet, allowing them to be monitored and even controlled by other connected entities. The home-automation market alone, which represents a tiny portion of the connected devices that make up the Internet of Things (IoT), is estimated to be a $53 billion market by 2022.

The connected home has sensors and remotely controlled devices such as motion detectors, cameras, smart thermostats, and even lighting systems. And this is all

---

4 See Kate Dreyer’s April 13, 2015 blog at the Comscore site (http://mng.bz/7eKv) for a summary of the report.

5 You can read more about these findings by Zion Market Research at the GlobeNewswire site (http://mng.bz/mm6a).
extremely affordable; after a burst pipe during a –26-degree (Fahrenheit) weather spell a few years ago, I started with a modest system including an internet-connected thermostat and some temperature sensors, and spent less than $300. Other connected devices include automobiles, home appliances, farming equipment, jet engines, and the supercomputer most of us carry around in our pockets (the smartphone).

Internet-connected devices change the nature of the software we build in two fundamental ways. First, the volume of data flowing over the internet is dramatically increased. Billions of devices broadcast data many times a minute, or even many times a second. Second, in order to capture and process these massive quantities of data, the computing substrate must be significantly different from those of the past. It becomes more highly distributed with computing resources placed at the “edge,” closer to where the connected device lies. This difference in data volume and infrastructure architecture necessitates new software designs and practices.

1.1.5 Data-driven

Considering several of the requirements that I’ve presented up to this point drives you to think about data in a more holistic way. Volumes of data are increasing, sources are becoming more widely distributed, and software delivery cycles are being shortened. In combination, these three factors render the large, centralized, shared database unusable.

A jet engine with hundreds of sensors, for example, is often disconnected from data centers housing such databases, and bandwidth limitations won’t allow all the data to be transmitted to the data center during the short windows when connectivity is established. Furthermore, shared databases require a great deal of process and coordination across a multitude of applications to rationalize the various data models and interaction scenarios; this is a major impediment to shortened release cycles.

Instead of the single, shared database, these application requirements call for a network of smaller, localized databases, and software that manages data relationships across that federation of data management systems. These new approaches drive the need for software development and management agility all the way through to the data tier.

Finally, all of the newly available data is of little value if it goes unused. Today’s applications must increasingly use data to provide greater value to the customer through smarter applications. For example, mapping applications use GPS data from connected cars and mobile devices, along with roadway and terrain data to provide real-time traffic reports and routing guidance. The applications of the past decades that implemented painstakingly designed algorithms carefully tuned for anticipated usage scenarios are being replaced with applications that are constantly being revised or may even be self-adjusting their internal algorithms and configurations.

Gartner forecasts that 8.4 billion connected things will be in use worldwide in 2017; see the Gartner report at www.gartner.com/newsroom/id/3598917.
These user requirements—constant availability, constant evolution with frequent releases, easily scalable, and intelligent—can’t be met with the software design and management systems of the past. But what characterizes the software that can meet these requirements?

### 1.2 Introducing cloud native software

Your software needs to be up, 24/7. You need to be able to release frequently to give your users the instant gratification they seek. The mobility and always-connected state of your users drives a need for your software to be responsive to larger and more fluctuating request volumes than ever before. And connected devices (“things”) form a distributed data fabric of unprecedented size that requires new storage and processing approaches. These needs, along with the availability of new platforms on which you can run the software, have led directly to the emergence of a new architectural style for software: cloud native software.

### 1.2.1 Defining “cloud native”

What characterizes cloud native software? Let’s analyze the preceding requirements a bit further and see where they lead. Figure 1.4 takes the first few steps, listing requirements across the top and showing causal relationships going downward. The following list explains the details:

- Software that’s always up must be resilient to infrastructure failures and changes, whether planned or unplanned. As the context within which it runs experiences those inevitable changes, software must be able to adapt. When properly constructed, deployed, and managed, composition of independent pieces can limit the blast radius of any failures that do occur; this drives you to a modular design. And because you know that no single entity can be guaranteed to never fail, you include redundancy throughout the design.

- Your goal is to release frequently, and monolithic software doesn’t allow this; too many interdependent pieces require time-consuming and complex coordination. In recent years, it’s been soundly proven that software made up of smaller, more loosely coupled and independently deployable and releasable components (often called microservices) enables a more agile release model.

- No longer are users limited to accessing digital solutions when they sit in front of their computers. They demand access from the mobile devices they carry with them 24/7. And nonhuman entities, such as sensors and device controllers, are similarly always connected. Both of these scenarios result in a tidal wave of request and data volumes that can fluctuate wildly, and therefore require software that scales dynamically and continues to function adequately.

Some of these attributes have architectural implications: the resultant software is composed of redundantly deployed, independent components. Other attributes address the management practices used to deliver the digital solutions: a deployment must...
Introducing cloud native software

Always up
Resilience
Redundancy
Modular

Release frequently
Agility
Adaptability

Anywhere, any device
Internet of Things
Large and fluctuating volumes of requests and data

Dynamic scalability

Figure 1.4 User requirements for software drive development toward cloud native architectural and management tenets.

adapt to a changing infrastructure and to fluctuating request volumes. Taking that collection of attributes as a whole, let’s carry this analysis to its conclusion; this is depicted in figure 1.5:

- Software that’s constructed as a set of independent components, redundantly deployed, implies distribution. If your redundant copies were all deployed close to one another, you’d be at greater risk of local failures having far-reaching consequences. To make efficient use of the infrastructure resources you have, when you deploy additional instances of an app to serve increasing request volumes, you must be able to place them across a wide swath of your available infrastructure—even, perhaps, that from cloud services such as AWS, Google Cloud Platform (GCP), and Microsoft Azure. As a result, you deploy your software modules in a highly distributed manner.

- Adaptable software is by definition “able to adjust to new conditions,” and the conditions I refer to here are those of the infrastructure and the set of interrelated software modules. They’re intrinsically tied together: as the infrastructure changes, the software changes, and vice versa. Frequent releases mean frequent change, and adapting to fluctuating request volumes through scaling operations represents a constant adjustment. It’s clear that your software and the environment it runs in are constantly changing.

**DEFINITION** Cloud native software is highly distributed, must operate in a constantly changing environment, and is itself constantly changing.

Many more granular details go into the making of cloud native software (the specifics fill the pages of this volume). But, ultimately, they all come back to these core characteristics: highly distributed and constantly changing. This will be your mantra as you progress through the material, and I’ll repeatedly draw you back to extreme distribution and constant change.
1.2.2 A mental model for cloud native software

Adrian Cockcroft, who was chief architect at Netflix and is now VP of Cloud Architecture Strategy at AWS, talks about the complexity of operating a car: as a driver, you must control the car and navigate streets, all while making sure not to come into contact with other drivers performing the same complex tasks. You’re able to do this only because you’ve formed a model that allows you to understand the world and control your instrument (in this case, a car) in an ever-changing environment.

Most of us use our feet to control the speed and our hands to set direction, collectively determining our velocity. In an attempt to improve navigation, city planners put thought into street layouts (God help us all in Paris). And tools such as signs and signals, coupled with traffic rules, give you a framework in which you can reason about the journey you’re taking from start to finish.

Writing cloud native software is also complex. In this section, I present a model to help bring order to the myriad of concerns in writing cloud native software. My hope is that this framework facilitates your understanding of the key concepts and techniques that will make you a proficient designer and developer of cloud native software.

I’ll start simple, with core elements of cloud native software that are surely familiar to you, shown in figure 1.6.

An app implements key business logic. This is where you’ll be writing the bulk of the code. This is where, for example, your code will take a customer order, verify that items are available in a warehouse’s inventory, and send a notification to the billing department.

The app, of course, depends on other components that it calls to either obtain information or take an action; I call these services. Some of the services store state—the warehouse inventory, for example. Others may be apps that implement the business logic for another part of your system—customer billing, for example.

---

Figure 1.5 Architectural and management tenets lead to the core characteristics of cloud native software: it’s highly distributed and must operate in a constantly changing environment even as the software is constantly evolving.

7 Hear Adrian talk about this and other examples of complicated things at http://mng.bz/5NzO.
Introducing cloud native software

The app is the code you write—the business logic for your software.

Some of the services store and/or manage the software state.

The app will call upon other components to provide services it needs to fulfill its requirements.

Taking these simple concepts, let’s now build up a topology that represents the cloud native software you’ll build; see figure 1.7. You have a distributed set of modules, most of which have multiple instances deployed. You can see that most of the apps are also acting as services, and further, that some services are explicitly stateful. Arrows depict where one component depends on another.

Each of the components has a dual role. Virtually all components act as a service in some capacity.

The root app, in turn, has dependencies on three other services.

The user depends on the root app of the software.

Some apps depend on stateful services.

Apps will almost always have many instances deployed.

Solid arrows indicate a dependency.

Figure 1.6  Familiar elements of a basic software architecture

Figure 1.7  Cloud native software takes familiar concepts and adds extreme distribution, with redundancy everywhere, and constant change.
CHAPTER 1  You keep using that word: Defining “cloud native”

This diagram illustrates a few interesting points. First, notice that the pieces (the boxes and the database, or storage, icons) are always annotated with two designations: apps and services for the boxes, and services and state for the storage icons. I’ve come to think of the simple concepts shown in figure 1.7 as roles that various components in your software solution take on.

You’ll note that any entity that has an arrow going to it, indicating that the component is depended upon by another, is a service. That’s right—almost everything is a service. Even the app that’s the root of the topology has an arrow to it from the software consumer. Apps, of course, are where you’re writing your code. And I particularly like the combination of service and state annotations, making clear that you have some services that are devoid of state (the stateless services you’ve surely heard about, annotated here with “app”), whereas others are all about managing state.

And this brings me to defining the three parts of cloud native software, depicted in figure 1.8:

- **The cloud native app**—Again, this is where you’ll write code; it’s the business logic for your software. Implementing the right patterns here allows those apps to act as good citizens in the composition that makes up your software; a single app is rarely a complete digital solution. An app is at one or the other end of an arrow (or both) and therefore must implement certain behaviors to make it participate in that relationship. It must also be constructed in a manner that allows for cloud native operational practices such as scaling and upgrades to be performed.

- **Cloud native data**—This is where state lives in your cloud native software. Even this simple picture shows a marked deviation from the architectures of the past, which often used a centralized database to store state for a large portion of the software. For example, you might have stored user profiles, account details, reviews, order history, payment information, and more, all in the same database. Cloud native software breaks the code into many smaller modules (the apps), and the database is similarly decomposed and distributed.

- **Cloud native interactions**—Cloud native software is then a composition of cloud native apps and cloud native data, and the way those entities interact with one another ultimately determines the functioning and the qualities of the digital solution. Because of the extreme distribution and constant change that characterizes our systems, these interactions have in many cases significantly evolved from those of previous software architectures, and some interaction patterns are entirely new.

Notice that although at the start I talked about services, in the end they aren’t one of the three entities in this mental model. In large part, this is because pretty much everything is a service, both apps and data. But more so, I suggest that the interactions between services are even more interesting than a service alone. Services pervade through the entire cloud native software model.
Introducing cloud native software

Figure 1.8  Key entities in the model for cloud native software: apps, data, and interactions

With this model established, let’s come back to the modern software requirements covered in section 1.1 and consider their implications on the apps, data, and interactions of your cloud native software.

**Cloud native apps**

Concerns about cloud native apps include the following:

- Their capacity is scaled up or down by adding or removing instances. We refer to this as *scale-out/in*, and it’s far different from the scale-up models used in prior architectures. When deployed correctly, having multiple instances of an app also offers levels of resilience in an unstable environment.

- As soon as you have multiple instances of an app, and even when only a single instance is being disrupted in some way, keeping state out of the apps allows you to perform recovery actions most easily. You can simply create a new instance of an app and connect it back to any stateful services it depends on.

- Configuration of the cloud native app poses unique challenges when many instances are deployed and the environments in which they’re running are constantly changing. If you have 100 instances of an app, for example, gone are the days when you could drop a new config into a known filesystem location and restart the app. Add to that the fact that these instances could be moving all over your distributed topology. And applying such old-school practices to the instances as they are moving all over your distributed topology would be sheer madness.

- The dynamic nature of cloud-based environments necessitates changes to the way you manage the application lifecycle (not the software *delivery* lifecycle, but rather the startup and shutdown of the actual app). You must reexamine how you start, configure, reconfigure, and shut down apps in this new context.
**Cloud native data**

Okay, so your apps are stateless. But handling state is an equally important part of a software solution, and the need to solve your data-handling problems also exists in an environment of extreme distribution and constant change. Because you have data that needs to persist through these fluctuations, handling data in a cloud setting poses unique challenges. The concerns for cloud native data include the following:

- You need to break apart the data monolith. In the last several decades, organizations invested a great deal of time, energy, and technology into managing large, consolidated data models. The reasoning was that concepts that were relevant in many domains, and hence implemented in many software systems, were best treated centrally as a single entity. For example, in a hospital, the concept of a patient was relevant in many settings, including clinical/care, billing, experience surveys, and more, and developers would create a single model, and often a single database, for handling patient information. This approach doesn’t work in the context of modern software; it’s slow to evolve and brittle, and ultimately robs the seemingly loosely coupled app fabric of its agility and robustness. You need to create a distributed data fabric, as you created a distributed app fabric.

- The distributed data fabric is made up of independent, fit-for-purpose databases (supporting polyglot persistence), as well as some that may be acting only as materialized views of data, where the source of truth lies elsewhere. Caching is a key pattern and technology in cloud native software.

- When you have entities that exist in multiple databases, such as the “patient” I mentioned previously, you have to address how to keep the information that’s common across the different instances in sync.

- Ultimately, treating state as an outcome of a series of events forms the core of the distributed data fabric. Event-sourcing patterns capture state-change events, and the unified log collects these state-change events and makes them available to members of this data distribution.

**Cloud native interactions**

And finally, when you draw all the pieces together, a new set of concerns surface for the cloud native interactions:

- Accessing an app when it has multiple instances requires some type of routing system. Synchronous request/response, as well as asynchronous event-driven patterns, must be addressed.

- In a highly distributed, constantly changing environment, you must account for access attempts that fail. Automatic retries are an essential pattern in cloud native software, yet their use can wreak havoc on a system if not governed properly. Circuit breakers are essential when automated retries are in place.

- Because cloud native software is a composite, a single user request is served through invocation of a multitude of related services. Properly managing
cloud native software to ensure a good user experience is a task of managing a composition—each of the services and the interactions between them. Application metrics and logging, things we’ve been producing for decades, must be specialized for this new setting.

- One of the greatest advantages of a modular system is the ability to more easily evolve parts of it independently. But because those independent pieces ultimately come together into a greater whole, the protocols underlying the interactions among them must be suitable for the cloud native context; for example, a routing system that supports parallel deploys.

This book covers new and evolved patterns and practices to address these needs.

Let’s make all of this a bit more concrete by looking at a specific example. This will give you a better sense of the concerns I’m only briefly mentioning here, and will give you a good idea of where I’m headed with the content of this text.

1.2.3 Cloud native software in action

Let’s start with a familiar scenario. You have an account with Wizard’s Bank. Part of the time you engage with the bank by visiting the local branch (if you’re a millennial, just pretend with me for a moment 😊). You’re also a registered user of the bank’s online banking application. After receiving only unsolicited calls on your home landline (again, pretend 😊) for the better part of the last year or two, you’ve finally decided to disconnect it. As a result, you need to update your phone number with your bank (and many other institutions).

The online banking application allows you to edit your user profile, which includes your primary and any backup phone numbers. After logging into the site, you navigate to the Profile page, enter your new phone number, and click the Submit button. You receive confirmation that your update has been saved, and your user experience ends there.

Let’s see what this could look like if that online banking application were architected in a cloud native manner. Figure 1.9 depicts these key elements:

- Because you aren’t yet logged in, when you access the User Profile app, it will redirect you to the Authentication app. Notice that each of these apps has multiple instances deployed and that the user requests are sent to one of the instances by a router.
- As a part of logging in, the Auth app will create and store a new auth token in a stateful service.
- The user is then redirected back to the User Profile app, with the new auth token. This time, the router will send the user request to a different instance of the User Profile app. (Spoiler alert: sticky sessions are bad in cloud native software!)
- The User Profile app will validate the auth token by making a call to an Auth API service. Again, there are multiple instances, and the request is sent to one of them by the router. Recall that valid tokens are stored in the stateful Auth Token service, which is accessible from not only the Auth app, but also any instances of the Auth API service.
Because the instances of any of these apps (the User Profile or Auth apps) can change for any number of reasons, a protocol must exist for continuously updating the router with new IP addresses.

The User Profile app then makes a downstream request to the User API service to obtain the current user’s profile data, including phone number. The User Profile app, in turn, makes a request to the user’s stateful service.

After the user has updated their phone number and clicked the Submit button, the User Profile app sends the new data to an event log.

Eventually, one of the instances of the User API service will pick up and process this change event and send a write request to the Users database.

Figure 1.9 The online banking software is a composition of apps and data services. Many types of interaction protocols are in play.
Introducing cloud native software

Yes, this is already a lot, but I want to add even more.

I haven’t explicitly stated it, but when you’re back at the bank branch and the teller verifies your current contact information, you’ll expect the teller to have your updated phone number. But the online banking software and the teller’s software are two different systems. This is by design; it serves agility, resilience, and many of the other requirements that I’ve identified as important for modern digital systems. Figure 1.10 shows this product suite.

The structure of the bank teller software isn’t markedly different from that of the online banking software; it’s a composition of cloud native apps and data. But, as you can imagine, each digital solution deals with and even stores user data, or shall I say, customer data. In cloud native software, you lean toward loose coupling, even when you’re dealing with data. This is reflected with the Users stateful service in the online banking software and the Customers stateful service in the bank teller’s software.

The question, then, is how to reconcile common data values across these disparate stores. How will your new phone number be reflected in the bank teller software?

Figure 1.10 What appears to a user as a single experience with Wizard’s Bank is realized by independently developed and managed software assets.
CHAPTER 1  You keep using that word: Defining “cloud native”

Each of the two independent pieces of software deals with customer information. On the left, the customer is referred to as a user, and on the right, as a customer. Cloud native data is highly distributed. Cloud native software must address the way data is synchronized across separate systems.

Figure 1.11  A decomposed and loosely coupled data fabric requires techniques for cohesive data management.

In figure 1.11, I’ve added one more concept to our model—something I’ve labeled “Distributed data coordination.” The depiction here doesn’t imply any implementation specifics. I’m not suggesting a normalized data model, hub-and-spoke master data management techniques, or any other solution. For the time being, please accept this as a problem statement; I promise we’ll study solutions soon.

That’s a lot! Figures 1.9, 1.10, and 1.11 are busy, and I don’t expect you to understand in any great detail all that’s going on. What I do hope you take from this comes back to the key theme for cloud native software:

- The software solution comprises quite a distribution of a great many components.
- Protocols exist to specifically deal with the change that’s inflicted upon the system.

We’ll get into all the details, and more, throughout the following chapters.
1.3 **Cloud native and world peace**

I’ve been practicing in this industry long enough that I’ve seen several technological evolutions promise to solve *all* problems. When object-oriented programming emerged in the late 1980s, for example, some people acted as if this style of software would essentially write itself. And although such bullish predictions wouldn’t come to pass, many of the hyped technologies, without question, brought improvements to many elements of software—ease of construction and management, robustness, and more.

Cloud native software architectures, often referred to as microservices,\(^8\) are all the rage today—but spoiler alert, they also won’t lead to world peace. And even if they did come to dominate (and I believe they will), they don’t apply to everything. Let’s look at this in more detail in a moment, but first, let’s talk about that word, *cloud*.

1.3.1 **Cloud and cloud native**

The narrative around the term *cloud* can be confusing. When I hear a company owner say, “We’re moving to the cloud,” they often mean they’re moving some or maybe even all of their apps into someone else’s data center—such as AWS, Azure, or GCP. These clouds offer the same set of primitives that are available in the on-premises data center (machines, storage, and network), so such a “move to the cloud” could be done with little change to the software and practices currently being used on premises.

But this approach won’t bring much improved software resilience, better management practices, or more agility to the software delivery processes. In fact, because the SLAs for the cloud services are almost always different from those offered in on-prem data centers, degradation is likely in many respects. In short, moving to the cloud doesn’t mean your software is cloud native or will demonstrate the values of cloud native software.

As I reasoned through earlier in the chapter, new expectations from consumers and new computing contexts—the very ones of the cloud—force a change in the way software is constructed. When you embrace the new architectural patterns and operational practices, you produce digital solutions that work well in the cloud. You might say that this software feels quite at home in the cloud. It’s a native of that land.

**NOTE**  *Cloud* is about *where* we’re computing. *Cloud native* is about *how*.

If cloud native is about how, does that mean you can implement cloud native solutions on premises? You bet! Most of the enterprises I work with on their cloud native journey first do so in their own data centers. This means that their on-premise computing infrastructure needs to support cloud native software and practices. I talk about this infrastructure in chapter 3.

As great as it is (and I hope that by the time you finish this book, you’ll think so too), cloud native isn’t for everything.

---

\(^8\) Although I use the term microservice to refer to the cloud native architecture, I don’t feel that the term encompasses the other two equally important entities of cloud native software: data and interactions.
**1.3.2 What isn’t cloud native**

I’m certain it doesn’t surprise you to hear that not all software should be cloud native. As you learn the patterns, you’ll see that some of the new approaches require effort that otherwise might not be necessary. If a dependent service is always at a known location that never changes, you won’t need to implement a service discovery protocol. And some approaches create new problems, even as they bring significant value; debugging program flow through a bunch of distributed components can be hard. Three of the most common reasons for not going cloud native in your software architecture are described next.

First, sometimes the software and computing infrastructure don’t call for cloud native. For example, if the software isn’t distributed and is rarely changing, you can likely depend on a level of stability that you should never assume for modern web or mobile applications running at scale. For example, code that’s embedded in an increasing number of physical devices such as a washing machine may not even have the computing and storage resources to support the redundancy so key to these modern architectures. My Zojirushi rice cooker’s software that adjusts the cooking time and temperature based on the conditions reported by on-board sensors needn’t have parts of the application running in different processes. If some part of the software or hardware fails, the worst that will happen is that I’ll need to order out when my home-cooked meal is ruined.

Second, sometimes common characteristics of cloud native software aren’t appropriate for the problem at hand. You’ll see, for example, that many of the new patterns give you systems that are eventually consistent; in your distributed software, data updated in one part of the system might not be immediately reflected in all parts of the system. Eventually, everything will match, but it might take a few seconds or even minutes for everything to become consistent. Sometimes this is okay; for example, it isn’t a major problem if, because of a network blip, the movie recommendations you’re served don’t immediately reflect the latest five-star rating another user supplied. But sometimes it’s not okay: a banking system can’t allow a user to withdraw all funds and close their bank account in one branch office, and then allow additional withdrawals from an ATM because the two systems are momentarily disconnected. Eventual consistency is at the core of many cloud native patterns, meaning that when strong consistency is required, those particular patterns can’t be used.

And, finally, sometimes you have existing software that isn’t cloud native, and there’s no immediate value in rewriting it. Most organizations that are more than a couple of decades old have parts of their IT portfolio running on a mainframe, and believe it or not, they may keep running that mainframe code for another couple of decades. But it’s not just mainframe code. A lot of software is running on a myriad of existing IT infrastructures that reflect design approaches that predate the cloud. You should rewrite code only when there’s business value in doing so, and even when there is, you’re likely to have to prioritize such efforts, updating various offerings in your portfolio over several years.
1.3.3 **Cloud native plays nice**

But it’s not all or nothing. Most of you are writing software in a setting filled with existing solutions. Even if you’re in the enviable position of producing a brand-new application, it will likely need to interface with one of those existing systems, and as I just pointed out, a good bit of the software already running is unlikely to be fully cloud native. The brilliant thing about cloud native is that it’s ultimately a composition of many distinct components, and if some of those components don’t embody the most modern patterns, the fully cloud native components can still interact with them.

Applying cloud native patterns where you can, even while other parts of your software employ older design approaches, can bring immediate value. In figure 1.12, for example, you see that we have a few application components. A bank teller accesses account information via a user interface, which then interfaces with an API that fronts a mainframe application. With this simple deployment topology, if the network between that Account API service and the mainframe application is disrupted, the customer will be unable to receive their cash.

![Diagram of banking software](https://example.com/diagram.png)

**Figure 1.12** Dispensing funds without access to the source of record is ill-advised.

But now let’s apply a few cloud native patterns to parts of this system. For example, if you deploy many instances of each microservice across numerous availability zones, a network partition in one zone still allows access to mainframe data through service instances deployed in other zones (figure 1.13).

It’s also worth noting that when you do have legacy code that you wish to refactor, it needn’t be done in one fell swoop. Netflix, for example, refactored its entire
Deploying multiple instances of an app across different failure boundaries allows cloud native patterns to provide benefit in a hybrid (cloud native and non-cloud native) software architecture.

Figure 1.13 Applying some cloud native patterns, such as redundancy and properly distributed deployments, brings benefit even in software that isn’t fully cloud native.

customer-facing digital solution to a cloud native architecture as a part of its move to the cloud. Eventually. The move took seven years, but Netflix began refactoring some parts of its monolithic, client-server architecture in the process, with immediate benefits. As with the preceding banking example, the lesson is that even during a migration, a partially cloud native solution is valuable.

Whether you’re building a new application that’s born and bred in and for the cloud, where you apply all of the newfangled patterns, or you’re extracting and making cloud native portions of an existing monolith, you can expect to realize significant value. Although we weren’t using the term cloud native then, the industry began experimenting with microservices-centric architectures in the early 2010s, and many of the patterns have been refined over several years. This “new” trend is well enough understood that its embrace is becoming significantly widespread. We’ve seen the value that these approaches bring.

I believe that this architectural style will be the dominant one for a decade or two to come. What distinguishes it from other fads with less staying power is that it came as a result of a foundational shift in the computing substrate. The client-server models that dominated the last 20 to 30 years first emerged when the computing infrastructure moved from the mainframe to one where many smaller computers became available,

---

9 For more details, see “Completing the Netflix Cloud Migration” by Yury Izrailevsky (http://mng.bz/6j0e).
and we wrote software to take advantage of that computing environment. Cloud native has similarly emerged as a new computing substrate—one offering *software-defined* compute, storage, and networking abstractions that are highly distributed and constantly changing.

**Summary**

- Cloud native applications can remain stable, even when the infrastructure they’re running on is constantly changing or even experiencing difficulties.
- The key requirements for modern applications call for enabling rapid iteration and frequent releases, zero downtime, and a massive increase in the volume and variety of the devices connected to it.
- A model for the cloud native application has three key entities:
  - The cloud native app
  - Cloud native data
  - Cloud native interactions
- *Cloud* is about where software runs; *cloud native* is about how it runs.
- Cloud nativeness isn’t all or nothing. Some of the software running in your organization may follow many cloud native architectural patterns, other software will live on with its older architecture, and still others will be hybrids (a combination of new and old approaches).
As a developer, you want nothing more than to create software that users will love and that will provide them value. When users want more, or you have an idea for something you’d like to bring to them, you’d like to build it and deliver it with ease. And you want your software to run well in production, to always be available and responsive.

Unfortunately, for most organizations, the process of getting software deployed in production is challenging. Processes designed to reduce risk and improve efficiency have the inadvertent effect of doing exactly the opposite, because they're
The obstacles

slow and cumbersome to use. And after the software is deployed, keeping it up and running is equally difficult. The resulting instability causes production-support personnel to be in a perpetual state of firefighting.

Even given a body of well-written, completed software, it’s still difficult to

- Get that software deployed
- Keep it up and running

As a developer, you might think that this is someone else’s problem. Your job is to produce that well-written piece of code; it’s someone else’s job to get it deployed and to support it in production. But responsibility for today’s fragile production environment doesn’t lie with any particular group or individual; instead, the “blame” rests with a system that has emerged from a set of organizational and operational practices that are all but ubiquitous across the industry. The way that teams are defined and assigned responsibility, the way that individual teams communicate, and even the way that software is architected all contribute to a system that, frankly, is failing the industry.

The solution is to design a new system that doesn’t treat production operations as an independent entity, but rather connects software development practices and architectural patterns to the activities of deploying and managing software in production.

In designing a new system, it behooves you to first understand what is causing the greatest pains in the current one. After you’ve analyzed the obstacles you currently face, you can construct a new system that not only avoids the challenges, but also thrives by capitalizing on new capabilities offered in the cloud. This is a discussion that addresses the processes and practices of the entire software delivery lifecycle, from development through production. As a software developer, you play an important role in making it easier to deploy and manage software in production.

2.1 The obstacles

No question—handling production operations is a difficult and often thankless job. Working hours usually include late nights and weekends, either when software releases are scheduled or when unexpected outages happen. It isn’t unusual for a fair bit of conflict to arise between application development groups and operations teams, with each blaming the other for failure to adequately serve consumers with superior digital experiences.

But as I said, that isn’t the fault of the ops team nor of the app-dev team. The challenges come from a system that inadvertently erects a series of barriers to success. Although every challenging situation is unique, with a variety of detailed root causes playing a part, several themes are common across almost all organizations. They’re shown in figure 2.1 and are summarized as follows:

- **Snowflakes**—Variability across the software development lifecycle (SDLC) contributes to trouble with initial deployments as well as to a lack of stability after the apps are running. Inconsistencies in both the software artifacts being deployed and the environments being deployed to are the problem.
- **Risky deployments**—The landscape in which software is deployed today is highly complex, with many tightly coupled, interrelated components. As such, a great risk exists that a deployment bringing a change in one part of that complex network will cause rippling effects in any number of other parts of the system. And fear of the consequences of a deployment has the downstream effect of limiting the frequency with which you can deploy.

- **Change is the exception**—Over the last several decades, we generally wrote and operated software with the expectation that the system where it ran would be stable. This philosophy was probably always suspect. But now, with IT systems being complex and highly distributed, this expectation of infrastructure stability is a complete fallacy. As a result, any instability in the infrastructure propagates up into the running application, making it hard to keep running.

- **Production instability**—And finally, because deploying into an unstable environment is usually inviting more trouble, the frequency of production deployments is limited.

Let’s explore each of these factors further.

---

**Figure 2.1** Factors that contribute to the difficulty in deploying software and keeping it running well in production

---

2.1.1 Snowflakes

“It works on my machine” is a common refrain when the ops team is struggling to stand up an application in production and reaches out to the development team for help. I’ve spoken with professionals at dozens of large enterprises who’ve told of six-, eight-, or even ten-week delays between the time that software is ready for release and the time it’s available to the user. One of the primary reasons for this delay is variability across the SDLC. This variability occurs along two lines:

- A difference in environments
- A difference in the artifacts being deployed

Without a mechanism for providing exactly the same environment from development through testing, staging, and production, it’s easy for software running in one environment to inadvertently depend on something that’s lacking or different in another one. One obvious example of this occurs when differences exist in the packages that the deployed software depends on. A developer might be strict about constantly updating all versions of the Spring Framework, for example, even to the point of automating installs as part of their build scripts. The servers in the production environment are far more controlled, and updates to the Spring Framework occur quarterly and only after a thorough audit. When the new software lands on that system, tests no longer pass, and resolution likely requires going all the way back to the developer to have them use the production-approved dependencies.

But it isn’t only differences in environment that slow deployments. All too often the artifact being deployed also varies through the SDLC—even when environment-specific values aren’t hardcoded into the implementation (which none of us would ever do, right?). Property files often contain configurations that are directly compiled into the deployable artifact. For example, the JAR file for your Java application includes an application.properties file and, if certain configuration settings are made directly in that file—ones that vary across dev, test, and prod—the JAR files must be different for dev, test, and prod too. In theory, the only differences between each of those JAR files are the contents of the property files, but any recompiling or repackaging of the deployable artifact can, and often does, end up inadvertently bringing in other differences as well.

These snowflakes don’t only have a negative impact on the timeline for the initial deployment; they also contribute greatly to operational instability. For example, let’s say you have an app that has been running in production with roughly 50,000 concurrent users. Although that number doesn’t generally fluctuate too much, you want room for growth. In the user acceptance testing (UAT) phase, you exercise a load with twice that volume, and all tests pass. You deploy the app into production, and all is well for some time. Then, on Saturday morning at 2 A.M., you see a spike in traffic. You suddenly have more than 75,000 users, and the system is failing. But, wait, in UAT you tested up to 100,000 concurrent users, so what’s going on?

It’s a difference in environment. Users connect to the system through sockets, socket connections require open file descriptors, and a configuration setting limits
the number of file descriptors. In the UAT environment, the value found in /proc/sys/fs/file-max is 200,000, but on the production server it’s 65,535. The tests you ran in UAT didn’t test for what you’d see in production, because of the differences between the UAT and production environments.

It gets worse. After diagnosing the problem and increasing the value in the /proc/sys/fs/file-max file, all of the operations staff’s best intentions for documenting this requirement are trumped by an emergency; and later, when a new server is configured, it has the file-max value set to 65,535 again. The software is installed on that server, and the same problem will eventually once again rear its ugly head.

Remember a moment ago when I talked about needing to change property files between dev, test, staging, and production, and the impact that can have on deployments? Well, let’s say you finally have everything deployed and running, and now something changes in your infrastructure topology. Your server name, URL, or IP address changes, or you add servers for scale. If those environment configurations are in the property file, then you must re-create the deployable artifact, and you risk having additional differences creep in.

Although this might sound extreme, and I do hope that most organizations have reigned in the chaos to some degree, elements of snowflake generation persist in all but the most advanced IT departments. The bespoke environments and deployment packages clearly introduce uncertainty into the system, but accepting that deployments are going to be risky is itself a first-class problem.

### 2.1.2 Risky deployments

When are software releases scheduled at your company? Are they done during “off hours,” perhaps at 2 A.M. on Saturday morning? This practice is commonplace because of one simple fact: deployments are usually fraught with peril. It isn’t unusual for a deployment to either require downtime during an upgrade, or cause unexpected downtime. Downtime is expensive. If your customers can’t order their pizza online, they’ll likely turn to a competitor, resulting in direct revenue loss.

In response to expensive outages, organizations have implemented a host of tools and processes designed to reduce the risks associated with releasing software. At the heart of most of these efforts is the idea that we’ll do a whole bunch of up-front work to minimize the chance of failure. Months before a scheduled deployment, we begin weekly meetings to plan the “promotion into upper environments,” and change-control approvals act as the last defense to keep unforeseen things from happening in production. Perhaps the practice with the highest price tag in terms of personnel and infrastructure resources is a testing process that depends on doing trial runs on an “exact replica of production.” In principle, none of these ideas sound crazy, but in practice, these exercises ultimately place significant burdens on the deployment process itself. Let’s look at one of these practices in more detail as an example: running test deployments in an exact replica of production.

A great deal of cost is associated with establishing such a test environment. For starters, twice the amount of hardware is needed; add to that double the software, and
capital costs alone grow twofold. Then there are the labor costs of keeping the test environment in alignment with production, complicated by a multitude of requirements such as the need to cleanse production data of personally identifiable information when generating testing data.

Once established, access to the test environment must be carefully orchestrated across dozens or hundreds of teams that wish to test their software prior to a production release. On the surface, it may seem like it’s a matter of scheduling, but the number of combinations of different teams and systems quickly makes it an intractable problem.

Consider a simple case in which you have two applications: a point-of-sale (PoS) system that takes payments, and a special order (SO) application that allows a customer to place an order and pay for it by using the PoS application. Each team is ready to release a new version of their application, and they must perform a test in the preproduction environment. How should these two teams’ activities be coordinated?

One option is to test the applications one at a time, and although executing the tests in sequence would extend the schedule, the process is relatively tractable if all goes well with each of the tests.

Figure 2.2 shows the following two steps. First, version 4 of the SO app is tested with version 1 (the old version) of the PoS app. When it’s successful, version 4 of the

![Diagram showing testing two apps in sequence](image-url)

**Figure 2.2** Testing two apps in sequence is straightforward when all tests pass.
SO application is deployed into production. Both test and production are now running v4 of SO, and both are still running v1 of PoS. The test environment is a replica of prod. Now you can test v2 of the PoS system, and when all the tests pass, you can promote that version into production. Both application upgrades are complete, with the test and prod environments matching.

But what happens if tests fail for the upgrade to the SO system? Clearly, you can’t deploy the new version into production. But now what do you do in the test environment? Do you revert to version 3 of SO (which takes time), even if PoS doesn’t depend on it? Was this a sequencing problem, with SO expecting PoS to already be on version 2 before it began its test? How long before SO can get back into the queue for using the test environment?

Figure 2.3 shows a couple of alternatives, which get complicated quickly, even in this toy scenario. In a real setting, this becomes intractable.

My goal isn’t to solve this problem here, but rather to demonstrate that even an oversimplified scenario can quickly become extraordinarily complicated. I’m sure you can imagine that when you add more applications to the mix and/or try to test new versions of multiple applications in parallel, the process becomes completely intractable. The environment that’s designed to ensure that things go well when software is deployed in production becomes a substantial bottleneck, and teams are caught between the need to get finished software out to the consumer as quickly as possible and doing it with complete confidence. In the end, it’s impossible to test exactly the scenarios that will present themselves in the production environment, and deployments remain risky business.

Risky enough, in fact, that most businesses have time periods in the year when new deployments into production aren’t permitted. For health insurance companies, it’s the open-enrollment period. In e-commerce in the United States, it’s the month between Thanksgiving and Christmas. That Thanksgiving-to-Christmas time frame is also sacred for the airline industry. The risks that persist despite efforts to minimize them make it difficult to get software deployed.

And because of this difficulty, the software running in production right now is likely to stay there for some time. We might be well aware of bugs or vulnerabilities in the apps and on the systems that are driving our customer experiences and business needs, but we must limp along until we can orchestrate the next release. For example, if an app has a known memory leak, causing intermittent crashes, we might preemptively reboot that app at regular intervals to avoid an emergency. But an increased workload against that application could cause the out-of-memory exception earlier than anticipated, and an unexpected crash causes the next emergency.

Finally, less-frequent releases lead to larger batch sizes; a deployment brings with it many changes, with equally many relationships to other parts of the system. It has been well established, and it makes intuitive sense, that a deployment that touches many other systems is more likely to cause something unexpected. Risky deployments have a direct impact on operational stability.
First up in the schedule for using the test environment is the SO app. You deploy version 4 into test, but there’s a problem. You don’t promote v4 into prod.

At this point, the test env has version 4 of the SO app deployed; prod has version 3. The environments don’t match.

Do you:
- Revert the version of SO in the test environment.
- Now let PoS in for testing.
- Then SO can try again.

Or:
- Leave version 4 of the SO app deployed (prod has version 3).
- Allow PoS into the test v2 environment. (It doesn’t have a dependence on SO anyway.) Notice that test no longer is an “exact replica” of production.
- Then schedule SO back in to try again. This time all goes well, and both new versions are now available in test and production.

Figure 2.3 A failing test immediately complicates the process for preproduction testing.
2.1.3 Change is the exception

Over the years, I’ve had dozens of conversations with CIOs and their staff who have expressed a desire to create systems that provide differentiated value to their business and their customers, but instead they’re constantly facing emergencies that draw their attention away from these innovation activities. I believe the cause of staff being in constant firefighting mode is the prevailing mindset of these long-established IT organizations: change is an exception.

Most organizations have realized the value of involving developers in initial deployments. A fair bit of uncertainty exists during fresh rollouts, and involving the team that deeply understands the implementation is essential. But at some point, responsibility for maintaining the system in production is completely handed over to the ops team, and the information for how to keep things humming is provided to them in a runbook. The runbook details possible failure scenarios and their resolutions, and although this sounds good in principle, on deeper reflection it demonstrates an assumption that the failure scenarios are known. But most aren’t!

The development team disengaging from ongoing operations when a newly deployed application has been stable for a predetermined period of time subtly hints at a philosophy that some point in time marks the end of change—that things will be stable from here on out. When something unexpected occurs, everyone is left scrambling. When the proverbial constant change persists, and I’ve already established that in the cloud it will, systems will persist in experiencing instability.

2.1.4 Production instability

All the factors I’ve covered until now inarguably hinder software from running well, but production instability itself further contributes to making deployments hard. Deployments into an already unstable environment are ill-advised; in most organizations, risky deployments remain one of the leading causes of system breakage. A reasonably stable environment is a prerequisite to new deployments.

But when the majority of time in IT is spent fighting fires, we’re left with few opportunities for deployments. Aligning those rare moments where production systems are stable with the timing of completing the complex testing cycles I talked about earlier, and the windows of opportunity shrink even further. It’s a vicious cycle.

As you can see, writing the software is only the beginning of bringing digital experiences to your customers. Curating snowflakes, allowing deployments to be risky, and treating change as an exception come together to make the job of running that software in production hard. Further insight about how these factors negatively impact operations today comes from studying well-functioning organizations—those from born-in-the-cloud companies. When you apply the practices and principles as they do, you develop a system that optimizes the entire software delivery lifecycle, from development to smooth-running operations.
2.2 The enablers

A new breed of companies, those that came of age after the turn of the century, have figured out how to do things better. Google has been a great innovator, and along with some of the other internet giants, has developed new ways of running IT. With its estimated two million servers running in worldwide data centers, there’s no way that Google could’ve managed using the techniques I just described. A different way exists.

Figure 2.4 presents a sketch of a system that’s almost an inverse of the bad system I described in the previous section. The goals are as follows:

- Easy and frequent releases into production
- Operational stability and predictability

![Diagram showing relationships between factors]

Figure 2.4 Explicit attention to these four factors develops a system of efficiency, predictability, and stability.

You’re already familiar with the inverses of some of the factors:

- Whereas snowflakes had previously contributed to slowness and instability, repeatability supports the opposite.
- Whereas risky deployments contributed to both production instability and challenging deployments, the ability to deploy safely drives agility and stability.
- Replacing practices and software designs that depend on an unchanging environment with ones that expect constant change radically reduces time spent fighting fires.

But looking at figure 2.4, you’ll notice a new entity labeled “Continuous delivery” (CD). The companies that have been most successful with the new IT operations model have redesigned their entire SDLC processes with CD as the primary driver.
This has a marked effect on the ease with which deployments can happen, and the benefits ripple through the entire system.

In this section, I first explain what CD is, how basic changes in the SDLC enable CD, and the positive outcomes. I then return to the other three key enablers and describe their main attributes and benefits in detail.

### 2.2.1 Continuous delivery

Amazon may be the most extreme example of frequent releases. It’s said to release code into production for www.amazon.com on average every second of every day. You might question the need for such frequent releases in your business, and sure, you probably don’t need to release software 86,000 times per day. But frequent releases drive business agility and enablement—both indicators of a strong organization.

Let me define *continuous delivery* by first pointing out what it isn’t. Continuous delivery doesn’t mean that every code change is deployed into production. Rather, it means that an as-new-as-possible version of the software is *deployable* at any time. The development team is constantly adding new capabilities to the implementation, but with each and every addition, they ensure that the software is ready to ship by running a full (automated!) test cycle and packaging the code for release.

Figure 2.5 depicts this cycle. Notice that there’s no “packaging” step following the “test” phase in each cycle. Instead, the machinery for packaging and deployment is built right into the development-and-test process.

Contrast this with the more traditional software development practice depicted in figure 2.6. A far longer single cycle is front-loaded with a large amount of software development that adds a great many features to an implementation. After a predetermined set of new capabilities has been added, an extensive test phase is completed and the software is readied for release.
Let’s assume that the time spans covered by figures 2.5 and 2.6 are the same, and that the start of each process is on the left, and the Ready to Ship point is on the far right. If you look at that rightmost point in time alone, you might not see much of a difference in outcome; roughly the same features will be delivered at roughly the same time. But if you dig under the covers, you’ll see significant differences.

First, with the former approach, the decision of when the next software release happens can be driven by the business rather than being at the mercy of a complex, unpredictable, software development process. For example, let’s say you learn that a competitor is planning a release of a product similar to yours in two weeks, and as a result, the business decides that you should make your own product immediately available. The business says, “Let’s release now!” In figure 2.7, overlaying that point in time over the previous two diagrams shows a stark contrast.

Using a software development methodology that supports CD allows the Ready to Ship software of the third iteration (shown in italics) to be immediately released. True, the application doesn’t yet have all of the planned features, but the competitive advantage of being first to market with a product that has some of the features may be significant. Looking at the lower half of the figure, you see that the business is out of luck. The IT process is a blocker rather than an enabler, and the competitor’s product will hit the market first!

The iterative process also affords another important outcome. When the Ready to Ship versions are frequently made available to customers, it gives you an opportunity to gather feedback used to better the subsequent versions of the product. You must be deliberate about using the feedback gathered after earlier iterations to correct false assumptions or even change course entirely in subsequent iterations. I’ve seen many Scrum projects fail because they strictly adhere to plans defined at the beginning of a project, not allowing results from earlier iterations to alter those plans.

Finally, let’s admit it: we aren’t good at estimating the time it takes to build software. Part of the reason is our inherent optimism. We usually plan for the happy path,
where the code works as expected immediately after the first write. (Yeah, when put like that, we see the absurdity of it right away, huh?) We also make the assumption that we’ll be fully focused on the task at hand; we’ll be cutting code all day, every day, until we get things done. And we’re probably getting pressured into agreeing to aggressive time schedules driven by market needs or other factors, usually putting us behind schedule even before we begin.

Unanticipated implementation challenges always come. Say you underestimate the effect of network latency on one part of your implementation, and instead of the simple request/response exchange that you planned for, you now need to implement a much more complex asynchronous communication protocol. And while you’re implementing this next set of features, you’re also getting pulled away from the new work to support escalations on already released versions of the software. And it’s almost never the case that your stretched goals fit within an already challenging time schedule.

The impact these factors have on the old-school development process is that you miss your planned release milestone. Figure 2.8 depicts the idealized software release plan in the first row. The second row shows the actual amount of time spent on development (longer than planned for), and the final two rows show alternatives for what you can do. One option is to stick with the planned release milestone, by compressing the testing phase, surely at the expense of software quality (the packaging phase usually can’t be shortened). A second option is to maintain the quality standards and move the release date. Neither of these options is pleasant.

<table>
<thead>
<tr>
<th>Plan:</th>
<th>Add feature</th>
<th>Add feature</th>
<th>Add feature</th>
<th>Add feature</th>
<th>Add feature</th>
<th>Test</th>
<th>Package</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual:</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Test</td>
<td>Package</td>
</tr>
<tr>
<td>Option 1:</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Test</td>
<td>Package</td>
</tr>
<tr>
<td>Option 2:</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Add feature</td>
<td>Test</td>
<td>Package</td>
</tr>
</tbody>
</table>

**Figure 2.8** When the development schedule slips, you need to decide between two unpalatable options.

Contrast this to the effects that “unanticipated” development delays have on a process that implements many shorter iterations. As depicted in figure 2.9, you again see that your planned release milestone is expected to come after six iterations. When the
actual implementation takes longer than expected, you see that you’re presented with some new options. You can either release on schedule with a more limited set of features (option 1), or you can choose a slight or longer delay for the next release (options 2 and 3). The key is that the business is presented with a far more flexible and palatable set of options. And when, through the system I’m presenting in this section, you make deployments less risky and therefore deploy more frequently, you can complete those two releases in rapid succession.

To net it all out, lengthy release cycles introduce a great deal of risk into the process of bringing digital products to consumers. The business lacks the ability to control when products are released to the market, and the organization as a whole is often in the awkward position of trading off near-term market pressures with long-term goals of software quality and ability to evolve.

**NOTE** Short iterations release a great deal of tension from the system. Continuous delivery allows business drivers to determine how and when products are brought to market.

I’ve talked about continuous delivery first, and at relative length, because it truly is at the core of a new, functional system of software development and operations. If your organization isn’t yet embracing practices such as these, this is where your initial efforts should be placed. Your ability to change the way that you bring software to market is hindered without such changes. And even the structure of the software you build, which is what this book is about, is linked to these practices in both subtle and direct ways. Software architecture is what this book is about, and we’ll cover that in depth throughout.
Now let’s go back to figure 2.4 and study the other factors that support our operational goals of easy, frequent releases and software stability.

2.2.2 Repeatability

In the previous section, I talked about the detrimental effect of variability, or as we often call them, snowflakes, on the workings of IT. They make things hard to deploy because you must constantly adjust to differences in both the environments into which you’re deploying, and in the variability of the artifacts you’re deploying. That same inconsistency makes it extremely difficult to keep things running well once in production, because every environment and piece of software gets special treatment anytime something changes. Drift from a known configuration is a constant threat to stability when you can’t reliably re-create the configuration that was working before a crash.

When you turn that negative into a positive in your enabling system, the key concept is repeatability. It’s analogous to the steps in an assembly line: each time you attach a steering wheel to a car, you repeat the same process. If the conditions are the same within some parameters (I’ll elaborate on this more in a moment), and the same process is executed, the outcome is predictable.

The benefits of repeatability on our two goals—getting things deployed and maintaining stability—are great. As you saw in the previous section, iterative cycles are essential to frequent releases, and by removing the variability from the dev/test process that happens with each turn of the crank, the time to deliver a new capability within the iteration is compressed. And once running in production, whether you’re responding to a failure or increasing capacity to handle greater volumes, the ability to stamp out deployments with complete predictability relieves tremendous stress from the system.

How do we then achieve this sought-after repeatability? One of the advantages of software is that it’s easy to change, and that malleability can be done quickly. But this is also exactly what has invited us to create snowflakes in the past. To achieve the needed repeatability, you must be disciplined. In particular, you need to do the following:

- Control the environments into which you’ll deploy the software
- Control the software that you’re deploying—also known as the deployable artifact
- Control the deployment processes

CONTROL THE ENVIRONMENT

In an assembly line, you control the environment by laying out the parts being assembled and the tools used for assembly in exactly the same way—no need to search for the three-quarter-inch socket wrench each time you need it, because it’s always in the same place. In software, you use two primary mechanisms to consistently lay out the context in which the implementation runs.

First, you must begin with standardized machine images. In building up environments, you must consistently begin with a known starting point. Second, changes applied to that base image to establish the context into which your software is deployed...
must be coded. For example, if you begin with a base Ubuntu image and your software
requires the Java Development Kit (JDK), you’ll script the installation of the JDK into
the base image. The term often used for this latter concept is infrastructure as code.
When you need a new instance of an environment, you begin with the base image and
apply the script, and you’re guaranteed to have the same environment each time.

Once established, any changes to an environment must also be equally controlled.
If operations staff routinely ssh into machines and make configuration changes, the
rigor you’ve applied to setting up the systems is for naught. Numerous techniques can
be used to ensure control after initial deployment. You may not allow SSH access into
running environments, or if you do, automatically take a machine offline as soon as
someone has ssh’d in. The latter is a useful pattern in that it allows someone to go
into a box to investigate a problem, but doesn’t allow for any potential changes to
color the running environment. If a change needs to be made to running environ-
ments, the only way for this to happen is by updating the standard machine image as
well as the code that applies the runtime environment to it—both of which are con-
trolled in a source code control system or something equivalent.

Who is responsible for the creation of the standardized machine images and the
infrastructure-as-code varies, but as an application developer, it’s essential that you use
such a system. Practices that you apply (or don’t) early in the software development
lifecycle have a marked effect on the organization’s ability to efficiently deploy and
manage that software in production.

CONTROL THE DEPLOYABLE ARTIFACT
Let’s take a moment to acknowledge the obvious: there are always differences in envi-
ronments. In production, your software connects to your live customer database,
found at a URL such as http://prod.example.com/customerDB; in staging, it connects
to a copy of that database that has been cleansed of personally identifiable informa-
tion and is found at http://staging.example.com/cleansedDB; and during initial
development, there may be a mock database that’s accessed at http://localhost/mockDB. Obviously, credentials differ from one environment to the next. How
do you account for such differences in the code you’re creating?

I know you aren’t hardcoding such strings directly into your code (right?). Likely,
you’re parameterizing your code and putting these values into some type of a prop-
erty file. This is a good first step, but often a problem remains: the property files, and
hence the parameter values for the different environments, are often compiled into
the deployable artifact. For example, in a Java setting, the application.properties file is
often included in the JAR or WAR file, which is then deployed into one of the environ-
ments. And therein lies the problem. When the environment-specific settings are
compiled in, the JAR file that you deploy in the test environment is different from the
JAR file that you deploy into production; see figure 2.10.

As soon as you build different artifacts for different stages in the SDLC, repeatabil-
ity may be compromised. The discipline for controlling the variability of that software
artifact, ensuring that the only difference in the artifacts is the contents of the prop-
Figure 2.10 Even when environment-specific values are organized into property files, including property files in the deployable artifact, you’ll have different artifacts throughout the SDLC.

Property files, must now be implanted into the build process itself. Unfortunately, because the JAR files are different, you can no longer compare file hashes to verify that the artifact that you’ve deployed into the staging environment is exactly the same as that which you’ve deployed into production. And if something changes in one of the environments, and one of the property values changes, you must update the property file, which means a new deployable artifact and a new deployment.

For efficient, safe, and repeatable production operations, it’s essential that a single deployable artifact is used through the entire SDLC. The JAR you build and run through regression tests during development is the exact JAR file deployed into the test, staging, and production environments. To make this happen, the code needs to be structured in the right way. For example, property files don’t carry environment-specific values, but instead define a set of parameters for which values may later be injected. You can then bind values to these parameters at the appropriate time, drawing values from the right sources. It’s up to you as the developer to create implementations that properly abstract the environmental variability. Doing this allows you to create a single deployable artifact that can be carried through the entire SDLC, bringing with it agility and reliability.

**Control the process**

Having established environment consistency, and the discipline of creating a single deployable artifact to carry through the entire software development lifecycle, what’s left is ensuring that these pieces come together in a controlled, repeatable manner. Figure 2.11 depicts the desired outcome: in all stages of the SDLC, you can reliably stamp out exact copies of as many running units as needed.
The enablers

Dev

Test

Prod

= Runtime
Env

= App

Figure 2.11 The desired outcome is to be able to consistently establish apps running in standardized environments. Note that the app is the same across all environments; the runtime environment is standardized within an SDLC stage.

This figure has no snowflakes. The deployable artifact, the app, is exactly the same across all deployments and environments. The runtime environment has variation across the different stages, but (as indicated by the different shades of the same gray coloring) the base is the same and has only different configurations applied, such as database bindings. Within a lifecycle stage, all the configurations are the same; they have exactly the same shade of gray. Those antisnowflake boxes are assembled from the two controlled entities I’ve been talking about: standardized runtime environments and single deployable artifacts, as seen in figure 2.12.

A whole lot is under the surface of this simple picture. What makes a good base image, and how is it made available to developers and operators? What is the source

Control the environment
Numerous things must be controlled:
- The base image is the starting point for all runtime environments. These must be carefully controlled—kept in an artifact repository and versioned.
- Environment configuration must also be controlled via tools such as source code control and configuration services.
- The assembly of the pieces is also automated via pipelines or other platform capabilities.

Control the deployable artifact
The creation of this is entirely automated using source code control systems, pipelines, and artifact repositories.

Control deployments
Note all of the process control in every part of these practices!

Figure 2.12 The assembly of standardized base images, controlled environment configurations, and single deployable artifacts is automated.
of the environment configuration, and when is it brought into the application context? Exactly when is the app “installed” into the runtime context? I’ll answer these questions and many more throughout the book, but at this juncture my main point is this: the only way to draw the pieces together in a manner that ensures consistency is to automate.

Although the use of continuous integration tools and practices is fairly ubiquitous in the development phase of writing software (for example, a build pipeline compiles checked-in code and runs some tests), its use in driving the entire SDLC isn’t as widely adopted. But the automation must carry all the way from code check-in, through deployments, into test and production environments.

And when I say it’s all automated, I mean everything. Even when you aren’t responsible for the creation of the various bits and pieces, the assembly must be controlled in this manner. For example, users of Pivotal Cloud Foundry, a popular cloud native platform, use an API to download new “stem cells,”11 the base images into which apps are deployed, from a software distribution site, and use pipelines to complete the assembly of the runtime environment and the application artifact. Another pipeline does the final deployment into production. In fact, when deployments into production also happen via pipelines, servers aren’t touched directly by humans, something that’ll make your chief security officer (and other control-related personnel) happy.

But if you’ve totally automated things all the way to deployment, how do you ensure that these deployments are safe? This is another area that requires a new philosophy.

### 2.2.3 Safe deployments

Earlier I talked about risky deployments and that the most common mechanism that organizations use as an attempt to control the risk is to put in place expansive and expensive testing environments with complex and slow processes to govern their use. Initially, you might think that there’s no alternative, because the only way to know that something works when deployed into production is to test it first. But I suggest that it’s more a symptom of what Grace Hopper said was the most dangerous phrase: “We’ve always done it this way.”

The born-in-the-cloud-era software companies have shown us a new way: they experiment in production. Egad! What am I talking about?! Let me add one word: they safely experiment in production.

Let’s first look at what I mean by safe experimentation and then look at the impact it has on our goals of easy deployments and production stability.

When trapeze artists let go of one ring, spin through the air, and grasp another, they most often achieve their goal and entertain spectators. No question about it, their success depends on the right training and tooling, and a whole load of practice. But acrobats aren’t fools; they know that things sometimes go wrong, so they perform over a safety net.

---

11 See Pivotal’s API Documentation page at [https://network.pivotal.io/docs/api](https://network.pivotal.io/docs/api) for more information.
When you experiment in production, you do it with the right safety nets in place. Both operational practices and software design patterns come together to weave that net. Add in solid software-engineering practices such as test-driven development, and you can minimize the chance of failure. But eliminating it entirely isn’t the goal. Expecting failure (and failure will happen) greatly lessens the chances of it being catastrophic. Perhaps a small handful of users will receive an error message and need to refresh, but overall the system remains up and running.

**TIP** Here’s the key: everything about the software design and the operational practices allows you to easily and quickly pull back the experiment and return to a known working state (or advance to the next one) when necessary.

This is the fundamental difference between the old and the new mindset. In the former, you tested extensively before going to production, believing you’d worked out all the kinks. When that delusion proved incorrect, you were left scrambling. With the new, you plan for failure, intentionally creating a retreat path to make failures a non-event. This is empowering! And the impact on your goals, easier and faster deployments, and stability after you’re up and running is obvious and immediate.

First, if you eliminate the complex and time-consuming testing process that I described in section 2.1.2, and instead go straight to production following basic integration testing, a great deal of time is cut from the cycle and, clearly, releases can occur more frequently. The release process is intentionally designed to encourage its use and involves little ceremony to begin. And having the right safety nets in place allows you to not only avert disaster, but to quickly return to a fully functional system in a matter of seconds.

When deployments come without ceremony and with greater frequency, you’re better able to address the failings of what you’re currently running in production, allowing you to maintain a more stable system as a whole.

Let’s talk a bit more about what that safety net looks like, and in particular, the role that the developer, architect, and application operators play in constructing it. You’ll look at three inextricably linked patterns:

- Parallel deployments and versioned services
- Generation of necessary telemetry
- Flexible routing

In the past, a deployment of version $n$ of some software was almost always a replacement of version $n - 1$. In addition, the things we deployed were large pieces of software encompassing a wide range of capabilities, so when the unexpected happened, the results could be catastrophic. An entire mission-critical application could experience significant downtime, for example.

At the core of your safe deployment practices is parallel deployment. Instead of completely replacing one version of running software with a new version, you keep the known working version running as you add a new version to run alongside it. You
CHAPTER 2  Running cloud native applications in production

start out with only a small portion of traffic routed to the new implementation, and you watch what happens. You can control which traffic is routed to the new implementation based on a variety of available criteria, such as where the requests are coming from (either geographically or what the referring page is, for example) or who the user is.

To assess whether the experiment is yielding positive results, you look at data. Is the implementation running without crashing? Has new latency been introduced? Have click-through rates increased or decreased?

If things are going well, you can continue to increase the load directed at the new implementation. If at any time things aren’t happy, you can shift all the traffic back to the previous version. This is the retreat path that allows you to experiment in production.

Figure 2.13 shows how the core practice works.

None of this can be done if proper software engineering disciplines are ignored, or applications don’t embody the right architectural patterns. Some of the keys to enable this form of A/B testing are as follows:

- Software artifacts must be versioned, and the versions must be visible to the routing mechanism to allow it to appropriately direct traffic. Further, because you’ll be analyzing data to determine whether the new deployment is stable and achieving the desired outcomes, all data must be associated with the appropriate version of the software in order to make the proper comparisons.
• The data used to analyze how the new version is functioning takes a variety of forms. Some metrics are completely independent of any details of the implementation; for example, the latency between a request and response. Other metrics begin to peer into the running processes, reporting on things such as the number of threads or memory being consumed. And finally, domain-specific metrics, such as the average total purchase amount of an online transaction, may also be used to drive deployment decisions. Although some of the data may automatically be provided by the environment in which the implementation is running, you won’t have to write code to produce it. The availability of data metrics is a first-class concern. I want you to think about producing data that supports experimentation in production.

• Clearly, routing is a key enabler of parallel deployments, and the routing algorithms are pieces of software. Sometimes the algorithm is simple, such as sending a percentage of all the traffic to the new version, and the routing software “implementation” can be realized by configuring some of the components of your infrastructure. Other times you may want more-sophisticated routing logic and need to write code to realize it. For example, you may want to test some geographically localized optimizations and want to send requests only from within the same geography to the new version. Or perhaps you wish to expose a new feature only to your premium customers. Whether the responsibility for implementing the routing logic falls to the developer or is achieved via configuration of the execution environment, routing is a first-class concern for the developer.

• Finally, something I’ve already hinted at is creating smaller units of deployment. Rather than a deployment encompassing a huge portion of your e-commerce system—for example, the catalog, search engine, image service, recommendation engine, shopping cart, and payment-processing module all in one—deployments should have a far smaller scope. You can easily imagine that a new release of the image service poses far less risk to the business than something that involves payment processing. Proper componentization of your applications—or as many would call it today, a microservices-based architecture—is directly linked to the operability of digital solutions.12

Although the platform your applications run on provide some of the necessary support for safe deployments (and I’ll talk more about this in chapter 3), all four of these factors—versioning, metrics, routing, and componentization—are things that you, as a developer, must consider when you design and build your cloud native application. There’s more to cloud native software than these things (for example, designing bulkheads into your architecture to keep failures from cascading through the entire system), but these are some of the key enablers of safe deployments.

2.2.4 Change is the rule

Over the last several decades, we’ve seen ample evidence that an operational model predicated on a belief that our environment changes only when we intentionally and knowingly initiate such changes doesn’t work. Reacting to unexpected changes dominates the time spent by IT, and even traditional SDLC processes that depend on estimates and predictions have proven problematic.

As we’re doing with the new SDLC processes I’ve been describing throughout this chapter, building muscle that allows you to adapt when change is thrust upon you affords far greater resilience. What’s subtle is identifying what those muscles are when it comes to stability and predictability for production systems. This concept is a bit tricky, a bit “meta” if you will; please bear with me a moment.

The trick isn’t to get better at predicting the unexpected or allocating more time for troubleshooting. For example, allocating half of a development team’s time to responding to incidents does nothing to address the underlying cause of the firefighting. You respond to an outage, get everything in working order, and you’re done—until the next incident.

“Done.”

This is the root of the problem. You believe that after you’re finished with a deployment, responding to an incident, or making a firewall change, you’ve somehow completed your work. The idea that you’re “done” inherently treats change as something that causes you to become not done.

Tip You need to let go of the notion of ever being done.

Let’s talk about eventual consistency. Rather than creating a set of instructions that brings a system into a “done” state, an eventually consistent system never expects to be done. Instead, the system is perpetually working to achieve equilibrium. The key abstractions of such a system are the desired state and the actual state.

The desired state of a system is what you want it to look like. For example, say you want a single server running a relational database, three application servers running RESTful web services, and two web servers delivering rich web applications to the users. These six servers are properly networked, and firewall rules are appropriately set. This topology, as shown in figure 2.14, is an expression of the desired state of the system.

You’d hope that, at some point, even most of the time, you have that system entirely estab-
lished and running well, but you’ll never assume that things remain as you left them immediately following a deployment. Instead, you treat the actual state, a model of what’s currently running in your system, as a first-class entity, constructing and maintaining it by using some of the metrics you already considered in this chapter.

The eventually consistent system then constantly compares the actual state to the desired state, and when there’s a deviation, performs actions to bring them back into alignment. For instance, let’s say that you lose an application server from the topology laid out in figure 2.14. This could happen for any number of reasons—a hardware failure, an out-of-memory exception coming from the app itself, or a network partition that cuts off the app server from other parts of the system.

Figure 2.15 depicts both the desired state and the actual state. The actual state and desired state clearly don’t match. To bring them back into alignment, another application server must be spun up and networked into the topology, and the application must be installed and started thereon (recall earlier discussions around repeatable deployments).

For those of you who previously might not have done much with eventual consistency, this might feel a bit like rocket science. An expert colleague avoids using the term eventual consistency because he worries that it’ll invoke fear in our customers. But systems built on this model are increasingly common, and many tools and educational materials can assist in bringing such solutions to fruition.

And I’ll tell you this: it’s absolutely, totally, completely essential to running applications on the cloud. I’ve said it before: things are always changing, so better to embrace that change than to react to it. You shouldn’t fear eventual consistency. You should embrace it.

Figure 2.15 When the actual state doesn’t match the desired state, the eventually consistent system initiates actions to bring them back into alignment.
Let me clarify something. Although the system I’m referring to here isn’t necessarily entirely automated, having a platform that implements the core portions of the paradigm is required (I’ll say more about the role of the platform in the next chapter). What I want you to do is design and build your software in a manner that allows a self-healing system to adapt to the constant change inflicted upon it. Teaching you how to do this is the aim of this book.

Software designed to remain functional in the face of constant change is the Holy Grail, and the impact on system stability and reliability is obvious. A self-healing system maintains higher uptime than one that requires human intervention each time something goes wrong. And treating a deployment as an expression of a new desired state greatly simplifies it and reduces risk. Adopting a mindset that change is the rule fundamentally alters the nature of managing software in production.

**Summary**

- In order for value to be realized from the code you write, you need to be able to do two things: get it deployed easily and frequently, and keep it running well in production.
- Missing the mark on either of these tasks shouldn’t be blamed on developers or operators. Instead, the “blame” rests with a failing system.
- The system fails because it allows bespoke solutions, which are hard to maintain; creates an environment that makes the act of deploying software inherently risky; and treats changes in the software and environment as an exception.
- When deployments are risky, they’re performed less frequently, which only serves to make them even riskier.
- You can invert each of these negatives—focusing on repeatability, making deployments safe, and embracing change—and create a system that supports rather than hinders the practices you desire.
- Repeatability is at the core of optimized IT operations, and automation applies not only to the software build process, but also to the creation of runtime environments and the deployment of applications.
- Software design patterns as well as operational practices expect the constant change in cloud-based environments.
- The new system depends on a highly iterative SDLC that supports continuous delivery practices.
- Continuous delivery is what a responsive business needs to compete in today’s markets.
- Finer granularity throughout the system is key. Shorter development cycles and smaller application components (microservices) account for significant gains in agility and resilience.
- Eventual consistency reigns supreme in a system where change is the rule.
This chapter covers

- A brief history of cloud platform evolution
- Foundational elements of the cloud native platform
- The basics of containers
- Use of a platform throughout the entire SDLC
- Security, compliance, and change control

I work with a lot of clients to help them understand and adopt cloud native patterns and practices, as well as a platform that’s optimized to run the software they produce. In particular, I work with and on the Cloud Foundry platform. I want to share an experience of one of my clients who adopted Cloud Foundry and deployed an existing application onto it.

Although that deployed software adhered to only a few of the cloud native patterns covered in this book (the apps were stateless and were bound to backing services that held the needed state), my client realized immediate benefits from moving to a modern platform. After deploying onto Cloud Foundry, they found
that the software was more stable than it had ever been. Initially, they attributed this to inadvertently improving quality during the light refactoring done for Cloud Foundry deployment.

But in reviewing the application logs, they found something surprising: the application was crashing just as frequently as it had before. They just hadn’t noticed it. The cloud native application platform was monitoring the health of the application, and when it failed, the platform automatically launched a replacement app. Under-the-covers problems remained, but the operator’s, and more importantly the user’s, experience was far better.

**NOTE** The moral of the story is this: although cloud native software prescribes many new patterns and practices, neither the developer nor the operator is responsible for providing all the functionality. Cloud native platforms, those designed to support cloud native software, provide a wealth of capabilities that support the development and operation of these modern digital solutions.

Now, let me be clear here: I’m not suggesting that such a platform should allow application quality to suffer. If a bug is causing a crash, it should be found and fixed. But such a crash needn’t necessarily wake an operator in the middle of the night or leave the user with a subpar experience until the problem is fixed. The new platform provides a set of services designed to deliver on the requirements that I’ve described in the preceding chapters, requirements for software that are continuously deployed, extremely distributed, and running in a constantly changing environment.

In this chapter, I’ll cover the key elements of cloud native platforms to explain what capabilities you can look to them for. Having a solid understanding of these capabilities will not only help you focus on your business needs rather than on the plumbing to support it, but will also allow you to optimize your implementation for cloud native deployment.

### 3.1 The cloud(native) platform evolution

Using platforms to support the development and operation of software isn’t new. The massively adopted Java 2 Platform, Enterprise Edition (J2EE) was first released nearly 20 years ago and has had seven major releases since then. JBoss, WebSphere, and WebLogic are commercial offerings of this open source technology that have generated billions in revenue for RedHat, IBM, and Oracle, respectively. Many other proprietary platforms such those from TIBCO Software or Microsoft have been equally successful—and have brought benefit to their users.

But just as new architectures are needed to meet modern demands on software, new platforms are needed to support the new implementations and operational practices around them. Let’s take a quick look at how we got to where we are today.
3.1.1 It started with the cloud

Arguably, cloud platforms began in earnest with Amazon Web Services (AWS). Its first offerings, made publicly available in releases throughout 2006, included compute (Elastic Compute Cloud, or EC2), storage (Simple Storage Service, or S3) and messaging (Simple Queue Service, or SQS) services. This was definitely a game changer in that developers and operations personnel no longer had to procure and manage their own hardware, but could instead obtain the resources they needed in a fraction of the time by using self-service provisioning interfaces.

Initially, this new platform represented the transference of existing client-server models into internet-accessible data centers. Software architectures didn’t change dramatically, nor did the development and operational practices around them. In these early days, the cloud was more about where computing was happening.

Almost immediately, characteristics of the cloud began to put pressure on software that was built for precloud infrastructures. Instead of using “enterprise-grade” servers, network devices, and storage, AWS used commodity hardware in its data centers. Using less-expensive hardware was key to offering cloud services at a palatable price, but with that came a higher rate of failure. AWS compensated for the reduced level of hardware resilience within its software and offerings, and presented abstractions to its users, such as availability zones (AZs), that would allow for software running on AWS to remain stable even while the infrastructure wasn’t.

What’s significant here is that by exposing these new primitives, such as AZs or regions, to the user of the service, that user takes on a new responsibility of using those primitives appropriately. We may not have realized it at the time, but exposing these new abstractions in the application program interface (API) of the platform began influencing a new architecture for software. People began writing software that was designed to run well on such a platform.

AWS effectively created a new market, and it took competitors, such as Google and Microsoft, two years to have any response. When they did, each came with unique offerings.

Google first came to market with Google App Engine (GAE), a platform designed expressly for running web applications. The abstractions it exposed, the first-class entities in the API, were markedly different from those of AWS. The latter predominantly exposed compute, storage, and network primitives; AZs, for example, generally map to sets of servers, allowing the abstraction to give the user control over server pool affinity or anti-affinity. By contrast, the GAE interface didn’t, and still doesn’t, provide any access to the raw compute resources that are running those web apps; it doesn’t expose infrastructure assets directly.

Microsoft came with its own flavor of cloud platform, including the capability to run medium trust code, for example. Similar to Google’s approach, the Medium Trust offering provided little direct access to the compute, storage, and network resources, and instead took the onus to create the infrastructure in which the user’s code would run. This allowed the platform to limit what the user’s code could do in the infrastruc-
ture, thereby offering certain security and resilience guarantees. Looking back now, I see these offerings from Google and Microsoft as two of the earliest forays from cloud into cloud native.

Google and Microsoft both eventually provided services that exposed infrastructure abstractions, as shown in figure 3.1, and, in reverse, AWS began offering cloud services with higher-level abstractions.

The different courses that these three vendors took in the latter half of the 2000s were hinting at the significant change that was coming in software architectures. As an industry, we were experimenting, seeing whether there might be ways of consuming and interacting with data center resources that would give us advantages in areas of productivity, agility, and resilience. These experiments eventually led to the formation of a new class of platform—the cloud native platform—that’s characterized by these higher-level abstractions, services tied to those, and the affordances they bring.

The cloud native platform is what you’ll study in this chapter. Let’s start by talking more about the higher-level abstractions that the cloud native platform provides.

### 3.1.2 Cloud native dial tone

Developers and application operators care about whether the digital solutions they’re running for their users function properly. In decades past, in order to provide the right service levels, they were required to correctly configure not only application deployments but also the infrastructure those applications ran on. This is because the primitives they had available to them were the same compute, storage, and network components they had always worked with.

As hinted at in the cloud platform evolution you just read about, this is changing. To clearly understand the difference, let’s look at a concrete example. Say you have an application deployed. To make sure that it’s running well, or to diagnose when things go wrong, you must have access to log and metric data.

As I’ve already established, cloud native apps have multiple copies deployed, both for resilience and scale. If you’re running those modern apps on an infrastructure-centric platform, one that exposes traditional infrastructure entities such as hosts, storage volumes, and networks, you must navigate through traditional data center abstractions to get or access those logs.
Figure 3.2 depicts the steps:

1. Determine which hosts are running the instances of your app; this is typically stored in a configuration management database (CMDB).
2. Determine which of those hosts are running the app instance you’re trying to diagnose the behavior for. This sometimes comes down to checking one host at a time until the right one is found.
3. After you’ve found the right host, you must navigate to a specific directory to find the logs you seek.

The entities that the operator is interacting with to get the job done are CMDBs, hosts, and filesystem directories.

By contrast, figure 3.3 shows the operator experience when the apps are running on a cloud native platform. It’s extremely simple: you ask for the logs for your application. You’re making app-centric requests.

The cloud native platform takes on a burden that was previously placed on the operator. This platform natively maintains an understanding of the application topology (previously stored in the CMDB), uses it to aggregate the logs for all application instances, and provides the operator the data needed for the entity they’re interested in.

The key point is this: the entity that the operator is interested in is the application—not the hosts the app is running on, or the directories that hold the logs. The operator needs the logs for the application they’re diagnosing.
1. Logs for my app, please.

**Figure 3.3** Accessing application logs in an app-centric environment is simple.

The contrast that you see in this example is one of infrastructure centricity versus application centricity. The difference in the application operator’s experience is due to the difference in the abstractions they’re working with. I like to call this a difference in *dial tone*.

**DEFINITION** Infrastructure-as-a-service (IaaS) platforms present *infrastructure dial tone*: an interface that provides access to hosts, storage, and networks—infrastructure primitives.

**DEFINITION** The cloud native platform presents *application dial tone*: an interface that makes the application the first-class entity that the developer or operator interacts with.

You’ve surely seen the blocks that are stacked in figure 3.4, clearly separating the three layers that ultimately come together to provide a digital solution to consumers. Virtualized infrastructure enables easier consumption of compute, storage, and network abstractions, leaving the management of the underlying hardware to the IaaS provider. The cloud native platform brings the level of abstraction up even further, allowing a consumer to consume OS and middleware resources more easily, and leaving the management of the underlying compute, storage, and network to the infrastructure provider.

The annotations on either side of the stack in figure 3.4 suggest differences in the operations performed against these abstractions. Instead of deploying an app onto one or more hosts via IaaS interfaces, on the cloud native platform an operator deploys an application, and the platform takes care of distributing the requested instances against available resources. Instead of configuring the firewall rules to secure the boundary of the hosts that are running a particular application, the opera-
3.2 Core tenets of the cloud native platform

Before I go deeper into some of the capabilities of and resultant benefits from adopting a cloud native platform, it’s important that you understand the philosophical underpinnings and the foundational patterns everything else is built on. It shouldn’t surprise you that this foundation is really all about providing support for highly distributed apps that live in an environment that’s constantly changing. But before I
present those two elements in more detail, let’s talk about the technology that’s essential to such platforms.

### 3.2.1 First, let’s talk about containers

As it happens, containers are a great enabler of cloud native software. Okay, that relationship isn’t quite the coincidence that my somewhat flippant remark suggests, but it’s a chicken-and-egg situation: the popularity of containers was without question driven by the need to support cloud native applications, and the availability of containers has equally driven advances in cloud native software.

If, when I use the term “container” you immediately think “Docker,” that’s cool—close enough. But I do want to cover key elements of containers in the abstract so that you can more easily connect those capabilities to the elements of cloud native software.

Starting at the most basic level, a container is a computing context that uses functionality from a host that it’s running on; for example, the base operating system. Generally, multiple containers are running on a single host, the latter of which is a server, either physical or virtual. These multiple containers are isolated from one another. At the highest level, they’re a bit like virtual machines (VMs), an isolated computing environment running on a shared resource. Containers, however, are lighter weight than VMs, allowing them to be created in orders of magnitude less time, and they consume fewer resources.

I already mentioned that multiple containers running on a single host share the host’s operating system, but that’s all. The rest of the runtime environment needed by your app (and yes, your app will be running in a container) runs within the container.

Figure 3.5 shows the portions of the application and runtime environment that are running both on the host and inside your containers. Only the OS kernel is provided by the host. Inside the container you first have the OS root filesystem, including operating system functions such as openssh or apt get. The runtime needed by your application is also inside the container—the Java Runtime Environment (JRE) or the .NET Framework, for example. And then, finally, your application is also in the container, hopefully running there.
When an application instance is to be run, a container is created on a host. All the bits necessary to run your app—the OS filesystem, application runtime, and the application itself—will be installed into that container and the appropriate processes started. The cloud native platform, using containers at the core, provides a whole lot of functionality for your software, and creation of an app instance is but one. Others include the following:

- Monitoring the health of the application
- Appropriate distribution of app instances across the infrastructure
- Assignment of IP addresses to the containers
- Dynamic routing to app instances
- Injection of configuration
- And much more

The key points I want you to remember about the container as you begin to study what a cloud native platform brings to bear is that (1) your infrastructure will have multiple hosts, (2) a host has multiple containers running on it, and (3) your app uses the OS and runtime environment installed into the container for its functionality. In many of the diagrams that follow, I depict the container with the icon shown in figure 3.6.

With this basic understanding of the container, let’s now look at the key tenets of the cloud native platform.

### 3.2.2 Support for “constantly changing”

I started this book with the story of an Amazon outage that demonstrated how an application can remain stable even as the platform it’s running on is experiencing trouble. Although developers play a crucial role in achieving that resilience through the design of their software, they needn’t be responsible for implementing every stability feature directly. The cloud native platform provides a significant part of that service.

Take AZs, for example. To support reliability, Amazon provides its EC2 users with access to multiple AZs, giving them the option to deploy their apps into more than one so that the apps may survive an AZ failure. But when an AZ fails on AWS, some users still lose their entire online presence.

The exact reason surely varies, but in general, failing to deploy apps across AZs occurs because doing so is nontrivial. You must keep track of the AZs you use, launch
machine instances into each AZ, configure networks across the AZs, and decide how to deploy app instances (containers) across the VMs that you have in each AZ. When you do any type of maintenance (an OS upgrade, for example), you must decide whether you’ll do this one AZ at a time or via another pattern. Need to move workloads because AWS is decommissioning the host you’re running on? You must think about your whole topology to see where, including which AZ, that workload should be moved to. It’s definitely complicated.

Although the AZ is an abstraction that AWS exposes to the user of its EC2 service, it needn’t be exposed to the user of the cloud native platform. Instead, the platform can be configured to use multiple AZs, and all that orchestration of application instances across those AZs is then handled by the platform. An app team simply requests multiple instances of an app be deployed (say, four, as shown in figure 3.7), and the platform automatically distributes them evenly across all available zones. The platform implements all of the orchestration and management that humans would otherwise shoulder the burden for if they weren’t using a cloud native platform. Then when change happens (an AZ goes down, for example), the application continues to work.

![Figure 3.7](image-url) - Management of workloads across availability zones is handled by the cloud native platform.

Another concept that I’ve previously mentioned is *eventual consistency*, a key pattern in the cloud, where things are constantly changing. Deployments and management tasks, which we know must be automated, are designed with the expectation that they are never done. Instead, the management of the system comes through constant monitoring of the actual (constantly changing) state of the system, comparing it to a desired state, and remediating when necessary. This technique is easy to describe but difficult to implement, and realizing the capability through a cloud native platform is essential.

Several cloud native platforms implement this basic pattern, including Kubernetes and Cloud Foundry. Although the implementation details differ slightly, the basic
Figure 3.8  The state of applications running on the platform is managed by continually comparing the desired state to the actual state and then executing corrective actions when necessary.

approaches are the same. Figure 3.8 depicts the key actors and the basic flow among them:

1. The user expresses the desired state by interacting with the API for the platform. For example, the user may ask that four instances of a particular app be running.

2. The platform API continually broadcasts changes to the desired state into a fault-tolerant, distributed data store or messaging fabric.

3. Each host running workloads is responsible for broadcasting the state of what is running on them into a fault-tolerant, distributed data store or messaging fabric.

4. An actor, which I’m calling the comparator here, ingests information from the state store, maintains a model of both the desired state and the actual state, and compares the two.

5. If the desired and actual states don’t match, the comparator informs another component in the system of the difference.

6. This component, which I’m calling the scheduler, determines where new workloads should be created or which workloads should be shut down, and communicates with the hosts to make this happen.

The complexity lies in the distributed nature of the system. Frankly, distributed systems are hard. The algorithms implemented in the platform must account for lost messages from the API or hosts, network partitions that may be brief but disrupt the flow nonetheless, and flapping state changes that are sometimes due to such flaky net-
works. Components such as the state store must have ways of maintaining state when inputs to it are in conflict (Paxos and Raft protocols are two of the most widely used at the moment). Just as application teams needn’t concern themselves with the complexity of managing workloads across AZs, they also needn’t be burdened with implementation of eventually consistent systems; that capability is baked into the platform.

The platform is a complex distributed system, and it needs to be as resilient as distributed apps are. If the comparator goes down, either due to failure or even something planned such as an upgrade, the platform must be self-healing. The patterns I’ve described here for apps running on the platform are also used for the management of the platform. The desired state may include 100 hosts running application workloads and a five-node distributed state store. If the system topology differs from that, corrective actions will bring it back to the desired state.

What I’ve described throughout this section is sophisticated and goes well beyond the simple automation of steps that may have previously been performed manually. These are the capabilities of the cloud native platform that support constant change.

### 3.2.3 Support for “highly distributed”

With all the talk about autonomy—team autonomy, which empowers the team to evolve and deploy its apps without high ceremony and heavily coordinated efforts, and app autonomy itself, which has individual microservices running within their own environment to both support independent development and reduce the risk of cascading failures—it feels like many problems are solved. And they are, but (yes, there’s a “but”) what comes from this approach is a system made up of distributed components that in prior architectures might have been singleton components or housed intraprocess; with that comes complexity where there once was none (or at least less).

The good news is that, as an industry, we’ve been working on solutions to these new problems for some time, and the patterns are fairly well established. When one component needs to communicate with another, it needs to know where to find that other component. When an app is horizontally scaled to hundreds of instances, you need a way to make a configuration change to all instances without requiring a massive, collective reboot. When an execution flow passes through a dozen microservices to fulfill a user request and it isn’t performing well, you need to find where in the elaborate network of apps the problem lies. You need to keep retries—a foundational pattern in cloud native software architectures in which a client service repeats requests to a providing service when responses aren’t forthcoming—from DDoS14-ing your system as a whole.

But remember, the developer isn’t responsible for implementing all the patterns required of cloud native software; instead, the platform can give the assist. Let’s take a brief look at some of the capabilities offered by cloud native platforms in this regard.

---

14 A distributed denial of service (DDoS) ([http://mng.bz/4OGR](http://mng.bz/4OGR)) isn’t always intentional or with malicious intent.
I want to use a concrete example to illustrate a handful of patterns; I’ll use a recipe-sharing site. One of the services it provides is a list of recommended recipes, and in order to do this, the recommendations service calls a favorites service to obtain the list of recipes that the user previously starred. These favorites are then used to calculate the recommendations. You have several apps, each with multiple instances deployed, and the functionality of those apps and the interaction among them determines the behavior of your software. You have a distributed system. What are some of the things that a platform might provide to support this distributed system?

**SERVICE DISCOVERY**

Individual services are running in separate containers and on different hosts; in order for one service to call another, it must first be able to find the other service. One of the ways that can happen is via the well-known patterns of the World Wide Web: DNS and routing. The recommendations service calls the favorites service via its URL, the URL is resolved to an IP address through DNS lookup, and that IP address points to a router that then sends on the request to one of the instances of the favorites service (figure 3.9).

![Figure 3.9](image_url)  

**Figure 3.9** The recommendations service finds the favorites service via DNS lookup and routing.

Another way is to have the recommendations service directly access instances of the favorites service via IP address, but because there are many instances of the latter, the requests must be load-balanced as before. Figure 3.10 depicts that this pulls the routing function into the calling service, thereby distributing the routing function itself.

Whether the routing function is logically centralized (figure 3.9) or highly distributed (figure 3.10), keeping what are effectively routing tables up-to-date is an important process. To fully automate this process, the platform implements patterns such as
The recommendations app needs to access the favorites service. It has a client-side load balancer embedded that will route requests to the IP addresses of the favorites service.

2. Keeping the IP tables of these distributed routers up-to-date with changes to the app instance IPs is a function of the cloud native platform.

Figure 3.10 The recommendations service directly accesses the favorites service via IP address; the routing function is distributed.

collecting IP address information from newly launched or recovered microservice instances, and distributing that data to the routing components, wherever they may be.

**SERVICE CONFIGURATION**

Our data scientists have done additional analysis and as a result would like to change some parameters for the recommendation algorithm. The recommendation service has hundreds of instances deployed, each of which must receive the new values. When the recommendation engine was deployed as a single process, you could go to that instance, supply a new configuration file, and restart the app. But now, with your highly distributed software architecture, no (human) individual knows where all the instances are running at any given time. But the cloud native platform does.

To provide this capability in a cloud native setting, a configuration service is required. This service works in concert with other parts of the platform to implement what’s shown in figure 3.11. The process depicted there is as follows:

1. An operator will supply the new configuration values to a configuration service (likely via a commit to a source code control system).
2. Service instances know how to access a configuration service from which they obtain configuration values whenever necessary. Certainly, the service instances will do this at startup time, but they must also do this when the configuration values are changed or when certain lifecycle events occur.
3. When configuration values are changed, the trick is to have each service instance refresh itself; the platform knows about all the service instances. The actual state exists in the state store.
4. The platform notifies each of the service instances that new values are available, and the instances take on those new values.
Again, neither the developer nor the app operator is responsible for implementing this protocol; rather, it’s automatically provided to apps deployed into the cloud native platform.

Service discovery and service configuration are but two of the many capabilities offered by the cloud native platform, but are exemplars of the runtime support needed for the modular and highly distributed nature of the cloud native application. Other services include the following:

- A distributed tracing mechanism that allows you to diagnose issue requests that flow through many microservices by automatically embedding tracers into those requests
- Circuit breakers that prevent inadvertent, internal DDoS attacks when something like a network disruption produces a retry storm

These and many more services are table stakes for a cloud native platform and greatly reduce the burden that would otherwise be placed on the developer and operator of the modern software we’re now building. Adoption of such a platform is essential for a high-functioning IT organization.

### 3.3 Who does what?

The cloud native platform can help with many more tasks—security and compliance, change control, multitenancy, and controlling the deployment process that I talked about in the previous chapter. But in order for you to fully appreciate the value, I first
need to talk about humans. In particular, I want to map responsibilities against the structure of the cloud native platform and data center.

Figure 3.12, a variant of figure 3.4, shows the same stack, but now I want to home in on the boundary between the cloud native platform and your software. At that boundary is a contract that determines how the software must be provided (the platform API) and a set of service levels that provide guarantees around how well the software will run on the platform.

For example, to run a simple web application, you may supply, via the platform API, the JAR files and HTML files for a web app and some backend services as well as the deployment topology. You might want two instances of your web app, and three instances of the backend service, which are connecting to the customer database. In terms of service levels, the contract may provide a guarantee that the application will have five nines (99.999%) of availability and will have all application logs persisted in your Splunk instance.

Establishing these boundaries and contracts enables something powerful: it allows you to form separate teams. One team is responsible for configuring the cloud native platform in such a way as to provide the service levels required by the organization. Members of this platform team have a particular skills profile; they know how to work with infrastructure resources, and they understand the inner workings of the cloud native platform and the primitives that allow them to fine-tune the behavior of the platform (that application logs are sent to Splunk).

**Figure 3.12** The cloud native platform presents a contract that allows consumers to deploy and manage their software without being exposed to low-level infrastructure details. Nonfunctional requirements such as performance guarantees are realized via SLAs that are achieved via specific platform configurations.
The other team, or shall I say teams, are the application teams whose members build and operate software for end consumers. They know how to use the platform APIs to deploy and manage the apps they’re running there. Members of these teams have skills profiles that enable them to understand cloud native software architectures and know how to monitor and configure them for optimal performance.

Figure 3.13 shows the part of the full stack that each team is responsible for. I want to draw your attention to two elements of this diagram:

- The parts of the stack that each team is responsible for don’t overlap. This is extraordinarily empowering and one of the main reasons that application deployments can happen far more frequently when using such a platform. This lack of overlap, however, is achieved only if the contract at the boundary between layers is designed correctly.
- Each team “owns” a product that it’s responsible for, and owns the entire lifecycle. The app team is responsible for building and operating the software; the platform gives those team members the contract that they need to do this. And the platform team is responsible for building (orconfiguring) and operating the product—the platform. The customers for this product are the app team members.

With the right contracts in place, the application team and the platform team are autonomous. Each can execute its responsibilities without extensive coordination with the others. Once again, it’s interesting to note how similar their responsibilities are. Each team is responsible for deployment, configuration, monitoring, scaling, and upgrading its respective products. What differs are the products they’re responsible for and the tools they use to perform those duties.

But achieving that autonomy, which is such an essential ingredient for delivering digital solutions in this era, depends not only on the definition of the contracts, but

Figure 3.13 The right abstractions support the formation of autonomous platform and application teams. Each is responsible for deployment, monitoring, scaling, and upgrading their respective products.
also on the inner workings of the cloud native platform itself. The platform must sup-
port the continuous delivery practices that are essential to achieving the agility we 
require. It must enable operational excellence by disallowing snowflakes and enabling 
app team autonomy, while concurrently implementing security, regulatory, and other 
controls. And it must provide services that lessen the burdens that are added when we 
create software composed of many highly distributed app components (microser-
vices) running in a multitenant environment.

I already touched on some of these topics when talking about the core tenets of 
cloud native platforms. Let’s now dig a bit deeper.

3.4 More cloud native platform capabilities

Now that you understand the basic support that platforms provide for highly distrib-
uted software running in an environment of constant change, as well as the workings 
of the application team and platform team, let’s look at additional factors you should 
understand about the cloud native platform.

3.4.1 The platform supports the entire SDLC

Continuous delivery can’t be achieved by only automating deployments into produc-
tion. Success begins early in the software development lifecycle. I’ve established that a 
single deployable artifact that carries through the SDLC is essential. What you need 
now are the environments into which that artifact will be deployed, and a way to have 
that artifact take on the appropriate configurations of those environments.

After you, as a developer, verify that the code is running on your own workstation, 
you check in the code. This kicks off a pipeline that builds the deployable artifact, 
installs it into an official dev environment, and runs the test suite. If the tests pass, you 
can move on to implementing the next feature, and the cycle continues. Figure 3.14 
depicts these deployments into the dev environment. The dev environment contains 
lightweight versions of various services on which the app depends—databases, mes-
 sage queues, and so on. In the diagram, these are represented by the symbols on the 
right-hand side.

Figure 3.14 Code commits generate deployable artifacts that are deployed into a dev 
environment that looks similar to production, but has development versions of services 
such as databases and message queues (depicted by the symbols on the right).
Another, less-frequent, trigger, perhaps a time-based one that runs daily, will deploy the artifact into testing, where a more comprehensive (and likely longer-running) set of tests are executed in an environment that’s a bit closer to production. You’ll notice that in figure 3.15, the general shape of the test environment is the same as that of the dev environment, but the two are shaded differently, indicating variances. For example, the network topology in the dev environment might be flat, with all apps being deployed into the same subnet, whereas in the test environment, the network may be partitioned to provide security boundaries.

The instances of the services available in each environment also differ. Their general shapes are the same (if it’s a relational database in dev, then it’s relational in test), but the difference in shading again signifies that they differ. For example, in the test environment, the customer database to which the app is bound may be a version of the entire production customer database, cleansed of personally identifiable information (PII), whereas in the development environment, it’s a small instance with some sample data.

Finally, when the business decides it would like to release the software, the artifact is tagged with a release version and deployed into production; see figure 3.16. The production environment, including the service instances, again differs from that of testing. For example, here the app is bound to the live customer database.

Although differences exist in the dev, test, and production environments, I hinted at and want to emphasize that important similarities exist as well. For example, the API used to deploy into any of the environments is the same; managing the automation essential for a streamlined SDLC process with varying APIs would be an unnecessary burden and a barrier to efficiency. The base environment that includes elements such as the operating system, language runtimes, specific I/O libraries, and more.
must be the same across all environments (I’ll come back to this when we talk about controlling the process in the next section). The contracts that govern the communication between the app and any bound services are also consistent across all environments. In short, having environment parity is absolutely essential to the continuous delivery process that begins all the way back in dev.

Managing those environments is a first-class concern of the IT organization, and a cloud native platform is the place to define and manage them. When the OS version in the dev environment is upgraded, it’s only in lockstep with all other environments. Similarly, when any of the services are revved (a new version of RabbitMQ or Postgres is made available, for example), it’s simultaneously done in all environments.

But even more than ensuring that the runtime environments match, a platform must also provide contracts that allow deployed apps to absorb the differences that exist from one stage to the next. For example, environment variables, which are a ubiquitous way of supplying values needed by an app, must be served to the app the same way all through the SDLC. And the manner in which services are bound to apps, thereby supplying connection arguments, must also be uniform.

Figure 3.17 offers a visual depiction of this concept. The artifact deployed into each of the spaces is exactly the same. The contracts between the app and the environment config, and the app and services (in this case, the Loyalty Members database), are also uniform. Note that the arrows pointing from each of the deployable artifacts are exactly the same across all environments—what differs are the details behind the env config and loyalty members abstractions. Abstractions such as these are an essential part of a platform that’s designed to support the entire SDLC.
On occasion, I’ve had clients implement a platform only for preproduction environments or only for production. There’s no question that having a cloud native platform that offers capabilities such as automated health management or a means of controlling standardized machine images provides value, even if available only in production. But given the need for continuous delivery of digital solutions, the platform must be applied across the entire SDLC. When the platform offers environment parity with the right abstractions, and an API that can be used to automate all interactions with it, the process of developing software and bringing it all the way through to production can be turned into a predictable, efficient machine.

### 3.4.2 Security, change-control, compliance (the control functions)

I’ve found that many, if not most, developers aren’t terribly fond of the chief security office, compliance, or change control. On the one hand, who can blame them? Developers want to get their app running in production, and these control functions require endless tickets to be filed and ultimately can stop a deployment from happening. On the other hand, if developers set their frustration aside for a moment, even they must admit the value that these organizations bring. We must protect our customer’s personal data. We must keep changes from taking down critical business applications. We have to appreciate the safeguards in place to keep an oversight from turning into a full-blown production incident.

The trouble with the current process isn’t the people, or even the organizations from which they come. The challenges arise because there are simply too many ways to make a mistake. A developer could, for example, specify a dependence on a particular version of the JRE that had been known to cause performance degradations for certain types of workloads and was therefore no longer permitted on production systems—so change control is needed to keep it from going to production. Do you have a control that says every user access to a particular database must be logged? The com-
CHAPTER 3  The platform for cloud native software

The compliance office is going to verify that logging agents are correctly deployed and configured. An explicit and often manual check that the rules are being followed is sometimes the only point of control.

Those points of control are implemented in various places across the application lifecycle and all too often are pushed too late in the cycle. When a deficiency is detected only the day before a planned deployment, the schedule is then at great risk, deadlines are missed, and everyone is unhappy. The most sobering thing about this, illustrated in figure 3.18, is that these controls apply to every deployment: every version, of every app. The time from ready to ship to deployed is, at best, counted in days, and multiple deployments in weeks.

![Figure 3.18 Control functions that are on the critical path for every release of every app reduce the number of deployments that can be performed.](image)

Remember when I talked about Amazon performing tens of thousands of deployments per day? It’s doing something fundamentally different. It isn’t that Amazon is exempt from regulatory requirements, nor is it cavalier with customers’ personal data. Instead, Amazon is satisfying the requirements that the controls are designed for in a different manner. It bakes the controls directly into its platform so that anything deployed there is guaranteed to meet the security and regulatory requirements.

In just a moment I’ll talk about how baking those controls into the platform works, but first let’s look at the outcome. If a deployment is guaranteed to meet the controls, you no longer have to go through a checklist before it happens. And if you no longer have a lengthy checklist, then the time from when you have your artifact ready to go to the time that it’s deployed shrinks dramatically. A deployment that once took days now takes minutes. And whereas a sequence of deployments took weeks, you can now complete several cycles within a single day (figure 3.19). You can try things out and get feedback far more frequently than before, and you’ve already studied the many benefits of such a practice.
More cloud native platform capabilities

Figure 3.19 By deploying to a platform that implements controls, you can reduce to mere minutes the time between having an app ready for deployment and performing that deployment.

Next, let’s dig into how such controls are built into the cloud native platform. How do you get the security, compliance, and change-control assurances that are claimed in figure 3.19?

3.4.3 Controlling what goes into the container

In chapter 2, I talked about the need for repeatability and that you achieve it by controlling the runtime environment, controlling the deployable artifact, and controlling the deployment process itself. Using container technology, you have a way of baking that level of control directly into the platform and can therefore achieve the security, compliance, and change-control assurances you’re after.

Figure 3.20 repeats figure 2.12 from chapter 2, which addressed the way the various parts of a running application are combined. Now that we’ve talked about containers, we can map each of the pieces in this diagram to the very entity that will be the running application.

Figure 3.21 shows a container running on a host. What I call the “base image” in figure 3.20 is now clearly shown as the root OS filesystem of the container. The runtime environment represents additional components that are installed into the root filesystem, such as the JRE or the .NET runtime. And then, finally, the deployable artifact also comes into the container. So then, how do you control each of these pieces?
CHAPTER 3  The platform for cloud native software

Control the environment
Numerous aspects must be controlled:
- The base image is the starting point for all runtime environments. These must be carefully controlled—kept in an artifact repository and versioned.
- Environment configuration must also be controlled via tools such as source code control and configuration services.
- The assembly of the pieces is also automated via pipelines or other platform capabilities.

Control the deployable artifact
The creation of this is entirely automated by using source code control systems, pipelines, and artifact repositories.

Complete runtime container

Control deployments

The runtime environment provides all that’s needed for the application to run (the .NET Framework or the JRE, for example) and may also include various additional software to provide some needed security and compliance controls.

Reduce risk by including only software that’s essential. Include software that’s required in the runtime context of every single application (logging agents).

Figure 3.20  The assembly of standardized base images, controlled environment configurations, and single deployable artifacts is automated.

Base image  Complete runtime container  App

Figure 3.21  The structure of the container image clearly separates the concerns of the app team from those of the platform team.

First, let’s talk about the base image. Recall that the operating system kernel comes from that which is running on the host (and I’ll come back to this in just a moment). In the root filesystem inside the container are additional things that are added to that kernel; you can think of them as the software packages installed into the OS. Because any software deployed into the operating system could bring with it a vulnerability, the best practice is to keep that base image as minimal as possible. For example, if you don’t want to allow SSH access into the container (restricting SSH access is a really...
good idea), you wouldn’t include OpenSSH in the base image. If you then control the set of base images, you have significant control over many security characteristics of your workloads.

Making the base image as small as possible is indeed a best practice, and making an attack surface smaller makes a system more secure. But security and compliance also come through ensuring that certain processes are guaranteed to run (logging agents, for example). Software packages that are required to run in every container should be included in the base image.

**POINT 1** The platform should allow only approved base images to be used.

That base image can be used as a foundation for a variety of specialized workloads. For example, some of your apps might be written in Java and therefore require the JRE, and other apps might be written in Python and therefore need the Python interpreter installed. This is the role of what is shown in figure 3.21 as the runtime environment. Of course, parts of that runtime environment, such as the JRE or the Python interpreter, may have themselves vulnerabilities, so the security office will have specific versions that are approved for use.

**POINT 2** The platform should control all runtime environments that may be included in a container.

Finally, the last piece that’s in the container is the application itself, and practices for the careful creation of this deployable artifact are well understood.

**POINT 3** Build pipelines coupled with code scans to provide the automation to repeatably and safely create the artifact.

Let’s now turn to the points of control. I’ve talked about having an architecture that separates the concerns of the app team from those of the platform team. The app team is responsible for delivering digital offerings that support the business, and the platform team is responsible for meeting security and compliance needs for the enterprise. The app team supplies only its app, and the platform team provides everything else.

Looking back at figure 3.21, you can see that the platform team supplies approved base images and approved runtime environments. You can also see that the platform team is responsible for the OS kernel that’s running on the host. In short, the platform team is able to impose security and compliance controls through each of the layers you’ve just studied.

### 3.4.4 Upgrading and patching vulnerabilities

When any part of the application container depicted in figure 3.21 needs to be updated, running instances aren’t modified. Instead, you deploy new containers with the new set of components. Because cloud native apps always have multiple instances deployed, you can move from an old version to the new with zero downtime.
The basic pattern for such an upgrade is that (1) a subset of the app instances are shut down and disposed of, (2) an equal number of instances of the new container are launched, and (3) after they’re up and running, you move on to replacing the next batch of old instances. The cloud native platform handles this process for you; you need only provide the new version of the app.

Look at the first few words of this section’s first paragraph: “When any part of the application container” needs to be updated—sometimes it’s the app that’s changing, and sometimes it’s all the pieces supplied by the platform. That’s right—the rolling upgrade is performed when you have a new version of your app, or whenever you have new versions of the operating system (kernel or root filesystem) or anything else in the runtime environment.

And it gets even better. The cloud native platform, designed to serve both the platform and the app teams’ needs, allows these teams to operate independently. This is extraordinarily powerful!

Figure 3.22 extends earlier diagrams that showed the app team doing deployments into the dev, test, and prod environments. Now you understand that with each deployment from the app team, a container is assembled with pieces supplied by the platform team and pieces supplied by the app team (recall figure 3.21).

It follows that when there are new versions of the parts of the container supplied by the platform team, a new container could also be assembled. If the app team has something new, a new container is assembled and deployed, and if the platform team has something new, a new container is assembled and deployed. This is shown in figure 3.22: from above, the app team is creating new containers; and from the side, on their own schedule, the platform team is reworking the platform elements. The platform team is updating the platform-supplied portions of the container.

This autonomy is essential for patch management in the data center. When a new vulnerability (CVE) is found,15 patches need to be applied quickly, without complex coordination across all apps running in the data center. This type of complex coordination was part of the reason that patches were often not applied as rapidly as they should have been in prior data center configurations.

Now, with the cloud native platform—you guessed it—when the platform team rolls out a fix for the latest vulnerability, the platform automatically creates the new container image and then replaces the running instances in batches, always leaving a subset of the app instances running as others are being cycled. This is the rolling upgrade. Of course, you aren’t going to be reckless; you’ll first deploy the patch into the staging environment and run tests there, and only after those pass will you move on to deployments in production.

If you think about this for a moment from the perspective of Google Cloud Platform, Amazon Web Services, Azure, or any of the other cloud platform providers, this type of autonomy between the platform team and the users of the platform is essential. With

---

15 CVE is an acronym for Common Vulnerabilities and Exposures; Wikipedia offers more details at http://mng.bz/QQr6.
More cloud native platform capabilities

over one million active users,\(^{16}\) AWS couldn’t manage its platform offering if it required coordination with individual members of that user base. You can absolutely apply the same practices in your data center, with the help of a cloud native application platform.

3.4.5 Change control

The change-control function is the last defense against a change (an upgrade or a new app being deployed, for example) causing something bad to happen in production. This is quite a responsibility that’s usually addressed by carefully looking at all the details of the planned deployment and evaluating how it might impact other systems running in the same environment. Impacts can include contention for computing resources, a broadening or restriction of access to various system components, or a dramatic increase in network traffic. What makes this job hard is that many things are

---

\(^{16}\) See “Amazon’s AWS Is Now a $7.3B Business as It Passes 1M Active Customers,” by Ingrid Lunden (http://mng.bz/Xgm9) for more details.
used and deployed into the same IT environment, so a change in one area can have rippling effects in many others.

The cloud native platform allows for a fundamentally different way of addressing the concerns of change control. It provides the means of insulating components from one another, so that problems in one part of the data center will be kept from impacting others.

It’s helpful to have a name you can use to refer to the entities that need to be isolated from one another; the term I use is tenant. When I use that term in this context, I don’t mean the proverbial Coke and Pepsi, two organizations that might be using the same environment but need to be so isolated that they don’t even know of each other. I’m more concerned with tenants that have a level of isolation that keeps them from inadvertently affecting each other. Our conversation then becomes one of multi-tenancy: you have many tenants that are all using a shared IT environment.

VMware pioneered shared computing infrastructure right around the turn of the century. It created a VM abstraction, the same entity that you interact with as a physical resource—a machine—and software controlled doling out shares of the physical resources to multiple VMs. Arguably, the main concern addressed with such virtualization technologies is shared resource use, and many, if not most, of the digital products running in large and small enterprises are now running in virtual machines. Independent software deployments are tenants on a shared computing infrastructure, and this worked extraordinarily well for software that was architected to be run on machines.

But as you know, architectures have changed, and the smaller, individual parts that come together to form cloud native software, coupled with the far more dynamic environments these apps are running in, have stressed the VM-based platforms. Although other attempts were previously made, container-based approaches have proven an outstanding solution. Based on the foundational concepts of control groups (cgroups), which control the use of shared resources, and namespaces, which control the visibility of shared resources, Linux containers have become the execution environments for the microservices collective that forms cloud native software.17

Containers provide part of the isolation that stands to satisfy the concerns of the change-control office; an app that gobbles up all the memory or CPU available in one container won’t affect other containers running on the same host. But, as I pointed out, other concerns remain. Who is allowed to deploy containers? How can you be sure to have monitoring data sufficient to assess whether an app is running amok? How can you allow routing changes for one app without allowing routing changes to be inadvertently made to another?

The answer is that the platform itself, which provides access control, monitoring, routing functions, and more, must be tenant aware. Figure 3.23 shows a set of hosts at the bottom, where Linux cgroups and namespaces are providing the compute isola-

---

17 Container technology was initially innovated and used on Linux, and the majority of container-centric systems still run on this operating system. More recently, Windows has added container support, yet its embrace remains far behind that of Linux.
More cloud native platform capabilities

Tenant isolation is achieved via implementation of the various functions of the cloud native platform.

Tenant isolation is achieved via Linux features of cgroups and namespaces.

Figure 3.23 True multitenancy in the compute tier shares resources in the control plane, as well as in the compute layer (the Linux host and kernel) while using containers to achieve resource isolation.

tion you need. In the upper part of the diagram are a whole host of other platform components that govern its use. The platform API is the place that access control is enforced. The metrics and logging system needs to group collected data into buckets for individual tenants. The scheduler, which determines where containers will be run, must be aware of relationships within a tenant and across tenants. In short, the cloud native application platform is multitenant.

And this multitenancy is what relieves the tension from the change-control function. Because deployment, upgrade, and configuration changes applied to one app/tenant are isolated from other apps/tenants, application teams are empowered to manage software on their own.

Summary

- A cloud native platform takes on a great deal of the burden of satisfying the requirements on modern software.
- The cloud native platform is used throughout the entire software development lifecycle.
- A cloud native platform projects higher-level abstraction than that of the infrastructure-centric platforms of the last decade.
- By baking control functions into the platform, deployments can be done far more frequently and are safer than when approvals are needed for every version of every app.
- App teams and platform teams can work independently, each managing the construction, deployment, and maintenance of their respective products.

- Eventual consistency is at the core of the platform as it constantly monitors the actual state of the system, compares it to the desired state, and remediates when necessary. This applies to both the software running on the platform and to the deployment of the platform itself.

- As software becomes more modular and distributed, the services that bring the components together into a whole do so as well. The platform must bake in support for these distributed systems.

- A cloud native platform is absolutely essential for organizations building and operating cloud native software.
Cloud Architecture Strategy, AWS 10
cloud native data 14
cloud native interactions 14–15
cloud native platform 52
cloud native software 1, 8–23
application requirements 5–8
data-driven 7–8
Internet of Things (IoT) 6–7
mobile support 6
multidevice support 6
shortened feedback cycles 5–6
zero downtime 5
cloud versus 19
defined 8–9
example of 15–18
interfacing with existing solutions 21–23
model for 10–15
cloud native apps 13
cloud native data 14
cloud native interactions 14–15
reasons for not going cloud native 20
cloud native software delivery lifecycle 48
enablers 33–48
change is the rule 46–48
continuous delivery 34–38
repeatability 38–42
safe deployments 42–45
obstacles 25–32
change is exception 32
production instability 32
risky deployments 28–30
snowflakes 27–28
cloud native software platforms
capabilities of 66–78
change control 75–78
containers, controlling what goes into 71–73
control functions 69–71
software development lifecycle (SDLC), support for entire 66–69
upgrading and vulnerability patching 73–75
core tenets of 55–63
containers 56–57
support for 57–63
evolution of 50–55
cloud 51–52
infrastructure-centric versus app-centric environment 52–55
map of responsibilities 63–66
CMDB (configuration management database) 53
Cockcroft, Adrian 10
Common Vulnerabilities and Exposures (CVE) 74
compliance 69–71
configuration management database (CMDB) 53
control-related functions 42
containers
controlling what goes into 71–73
overview 56–57
continuous delivery (CD) 33–38
control functions 69–71, 77
data-driven, as application requirement 7–8
deployable artifacts 38, 66
deployment 42–45
risky 28–30
design patterns 48
desired state 46–47
distribution, support for 60–63
DNS (Domain Name System) 61
EC2 (Elastic Compute Cloud platform) 2, 51
Amazon and reliability support 57
availability in other countries 2
event log 16
event-sourcing patterns 14
eventual consistency 47, 58
failures 43
favorites service 61
feedback cycles, shortened 5–6
front-loaded cycle 34
G
GAE (Google App Engine) 51
GCP (Google Cloud Platform) 9, 19, 74
hardcoding strings 39
I
IaaS (infrastructure-as-a-service) 54
infrastructure as code 39
infrastructure-as-a-service (IaaS) 54
IoT (Internet of Things) 6–7
IP addresses 16, 57
J
J2EE (Java 2 Platform, Enterprise Edition) 50
JAR files 39
Java 2 Platform, Enterprise Edition (J2EE) 50
Java Development Kit (JDK) 39
Java Runtime Environment (JRE) 56
JBoss 50
JDK (Java Development Kit) 39
JRE (Java Runtime Environment) 56
loose coupling 17
medium trust code 51
M
microservices 8, 19, 21
Microsoft Azure 9
mobile devices, support for 6
N
Netflix 22
O
OpenSSH 73
openssh function 56
**OS kernel** 56, 73  
**OS root filesystem** 56

**P**

PII (personally identifiable information) 67  
pipelines 73  
production instability 26, 32  
property files 39

**R**

Ready to Ship software 35  
recommendations service 61  
redundancy 11  
regions 51  
repeatability 38–42  
control of deployable artifact 39–40  
control of environment 38–39  
control of process 40–42  
resilience 13, 17  
risky deployments 26  
routing 61  
runtime book 32

**S**

S3 (Simple Storage Service) 51  
scale-out/in 13  
SDLC (Software Development Lifecycle)  
continuous delivery 66  
streamlined 67  
traditional 34  
variability across 27  
security 69–71  
SLAs (service-level agreements) 2, 4  
snowflakes 25, 27–28, 32, 38, 41  
SO (special order) application 29  
software artifacts 44  
software delivery lifecycle 13  
Software Development Lifecycle (SDLC) 25, 27, 66–69  
special order (SO) application 29  
Splunk 64  
SQS (Simple Queue Service) 51  
ssh 39  
SSH access 39, 72  
state-change events 14  
stem cells 42

**U**

UAT (user acceptance testing) 27  
unified log 14  
upgrading 73–75  
user acceptance testing (UAT) 27  
User Profile app 15–16

**V**

VMs (virtual machines) 56, 76  
vulnerability patching 73–75

**W**

WebLogic 50  
WebSphere 50

**Z**

zero downtime, as application requirement 5