

Compliments of **Pivotal**.



Cloud Native Foundations

Selections from
Cloud Native: Designing change-tolerant software
by Cornelia Davis





Selections from
Cloud Native: Designing change-tolerant software

Selected by Cornelia Davis

Chapter 1
Chapter 2
Chapter 3

Copyright 2017 Manning Publications
To pre-order or learn more about these books go to www.manning.com

For online information and ordering of these and other Manning books, please visit www.manning.com. The publisher offers discounts on these books when ordered in quantity.

For more information, please contact

Special Sales Department
Manning Publications Co.
20 Baldwin Road
PO Box 761
Shelter Island, NY 11964
Email: orders@manning.com

©2017 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

- ⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.
20 Baldwin Road Technical
PO Box 761
Shelter Island, NY 11964

Cover designer: Leslie Haimes

ISBN: 9781617295300
Printed in the United States of America
1 2 3 4 5 6 7 8 9 10 - EBM - 21 20 19 18 17 16

brief contents

1 YOU KEEP USING THAT WORD – DEFINING CLOUD.....	1
1.1 It's Not Amazon's Fault	1
1.2 Today's Application Requirements	4
1.3 Cloud-Native Software	7
1.4 Cloud, Cloud-Native, None of the Above and Somewhere In-Between	18
1.5 Summary	21
2 RUNNING CLOUD-NATIVE APPLICATIONS IN PRODUCTION.....	22
2.1 The Obstacles	23
2.2 The Enablers	29
2.3 Summary	44
3 THE PLATFORM FOR CLOUD-NATIVE SOFTWARE	46
3.1 The Cloud(-/native) Platform Evolution	47
3.2 Cloud-Native Dial Tone	49
3.3 The Platform Supports the Entire SDLC	52
3.4 Enable Operational Excellence	56
3.5 Support Microservice-Based Software Architectures	65
3.6 Summary	71
index	73



You Keep Using that Word – Defining Cloud

1.1 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, even their core customer facing services, on AWS, 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 more than five hours (or even more, depending on how you count) resulting in even longer outages for the affected AWS customers by the time 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 a part of the deal, AWS is responsible for keeping their 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, their customers can and will find alternatives. Service providers generally provide SLAs – Amazon, for example provides a 99.95% uptime guarantee for most of their services.

What I'm asserting is that the applications you're running on some 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 was one of the many companies affected by the outage and with it being, by one measure (by the amount of internet bandwidth consumed – 36%!), the top Internet site in the United States, a Netflix outage affects a lot of people. But Netflix had this to say about the outage:

"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

They were able to quickly recover from the AWS outage, being fully functional only minutes after the incident began. Netflix, which still runs on AWS, was fully functional even when the AWS outage continued.

NOTE How were they able to do this? 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 in place. Amazon Web Services does exactly this, and makes those redundancy abstractions available to their users. In particular, AWS offers services in numerous regions; for example, at the time of writing, their Elastic Compute platform (EC2) is running and available in Ireland, Frankfurt, Tokyo, Seoul, Singapore, Sydney, Sao Paulo, and in three locations in the United States (Virginia, California, and Oregon). Furthermore, within each region, the service is further partitioned into numerous availability zones (AZ) which are configured to isolate the resources of one AZ from another, limiting 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.

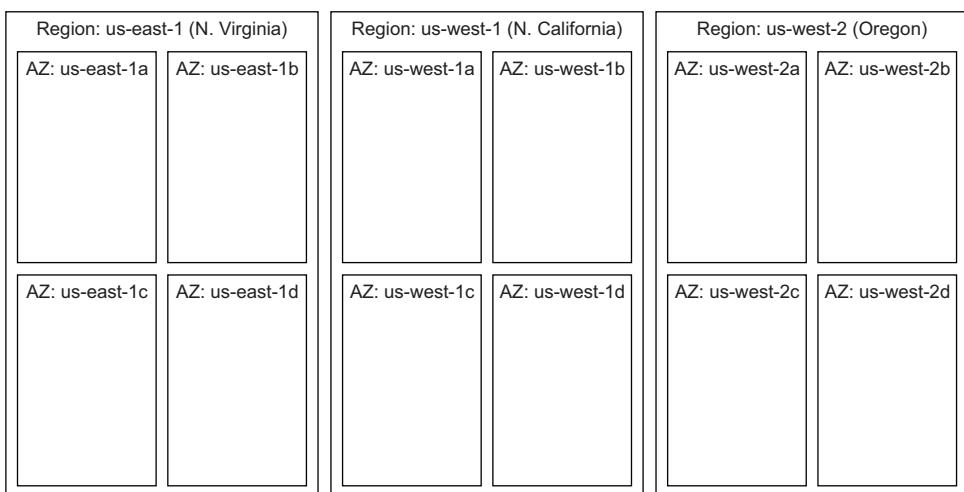


Figure 1.1 AWS partitions the services they offer into regions and availability zones. Regions map to geographic regions, and AZs provide further redundancy and isolation within a single region

¹ <http://techblog.netflix.com/2015/09/chaos-engineering-upgraded.html>

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 was one of the keys to uptime.

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

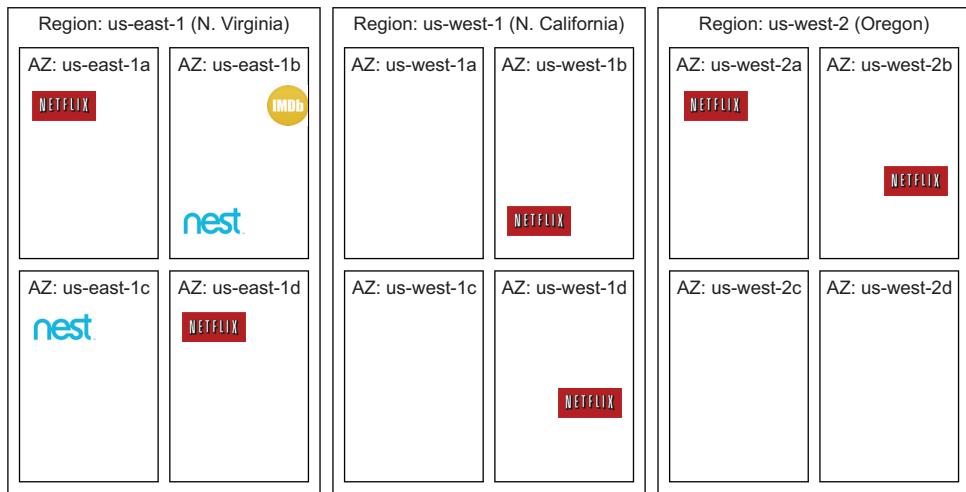


Figure 1.2 Applications deployed onto AWS may be deployed into a single AZ (IMDb), multiple AZs (Nest) but only a single region, or in multiple AZs and multiple regions (Netflix). This will provide different resiliency profiles.

Figure 1.3 depicts a single region outage, such as what happened during the AWS outage of September 2015; in that instance only us-east-1 went dark:

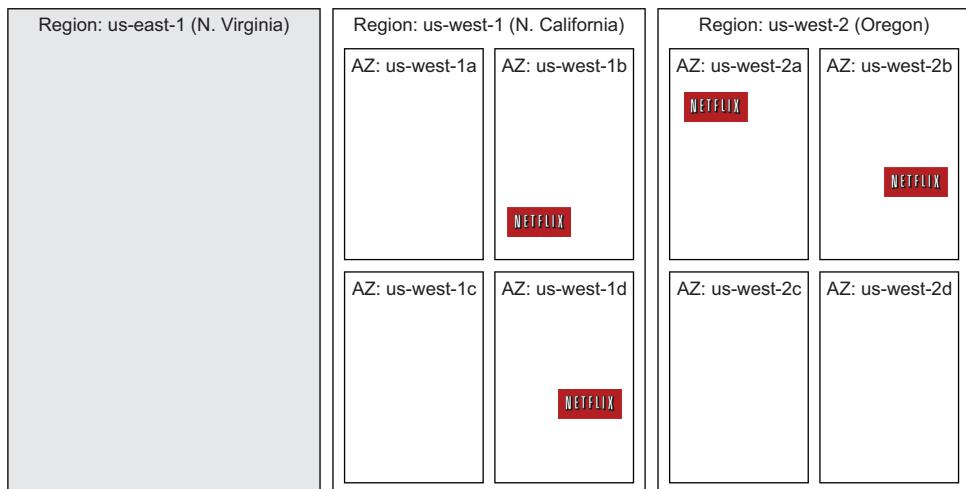


Figure 1.3 If applications are properly architected and deployed, digital solutions can survive even a broad outage such as an entire region.

In this simple graphic, we can immediately see how Netflix might have weathered the outage far better than others companies; they already had their applications running in other AWS regions and were 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, they'd anticipated (even practiced!¹) a possible outage such as this and had architected their software and designed their 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 for how to achieve the required SLAs was often limited to a small handful of "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**. 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 Because failure is the rule, not the exception.

1.2 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 in what is expected from the software we use – we want applications to always be available, be perpetually upgraded with new wiz-bang features, and provide us 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 take a brief look at the key themes and how these requirements are new or subtly evolved from what we've seen in the past.

¹ <http://www.techrepublic.com/article/aws-outage-how-netflix-weathered-the-storm-by-preparing-for-the-worst/>

1.2.1 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 where applications are unavailable are tolerated – the world is always online. And although unplanned downtime was never desirable, the impact of it has reached astounding levels. For example, in 2013 Forbes estimated that Amazon.com lost almost \$2 million during a 13-minute unplanned outage¹. Downtime, planned or not, results in significant revenue loss and customer dissatisfaction.

But maintaining uptime isn't only a problem for the operations team. The software developer or architect is responsible for creating a system design where loosely coupled components can be deployed allowing inevitable failures to be compensated for with redundancy and air-gaps that keep those failures from cascading through the entire system. She must also design the software to allow planned events, such as upgrades, to be done with zero downtime.

1.2.2 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 per day. Exciting customers is unquestionably valuable, but perhaps the biggest driver for these continuous releases is the reduction of risk. From the moment that we have an idea for a feature, we'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 performant way? As much as we try to predict the possible outcomes, reality is often different from what we 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, we can then make adjustments or even change course entirely. Frequent software releases shorten feedback loops and reduces risk.

The monolithic software systems that have dominated the last several decades can't be released often enough. Too many closely interrelated sub-systems, 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 were found late in the integration-testing phase, the long and laborious process would begin anew. New software architectures are essential to achieve the required agility in releasing software to production.

1.2.3 Mobile and Multi-Device Support

In April 2015, comScore, a leading technology-trend measurement and analytics company, released a report that for the first time mobile device Internet usage eclipsed that

¹ <http://www.forbes.com/sites/kellyclay/2013/08/19/amazon-com-goes-down-loses-66240-per-minute/#46207ddb3c2a>

of desktop computers.¹ In addition to needing 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), users increasingly expect the experience with an application to seamlessly move from one device to another as they navigate through their day. For example, they may be watching a movie on their Apple TV and then transition to viewing the program on their mobile device when they are on the train to the airport. Furthermore, the usage patterns on the mobile device are significantly different than those from the 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 front-end devices serving users and the system must adapt to expanding and contracting demands.

1.2.4 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, millions 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 2022². The connected home has sensors and remotely controlled devices such as motion detectors, cameras, smart thermostats, and even lighting systems. And this is all 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 in total. 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 devices³ 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 computer resources placed at the “edge”, closer to where the connected device lies. This difference in data volume and infrastructure architecture necessitates a new software designs and practices.

¹ <https://www.comscore.com/Insights/Blog/Mobile-Internet-Usage-Skyrockets-in-Past-4-Years-to-Overtake-Desktop-as-Most-Used-Digital-Platform>

² <https://globenewswire.com/news-release/2017/04/12/959610/0/en/Smart-Home-Market-Size-Share-will-hit-53-45-Billion-by-2022.html>

³ <http://www.gartner.com/newsroom/id/3598917>

1.2.5 Data-Driven

Considering several of the requirements that we've discussed up to this point drives us to think about data at a more holistic level. 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 is often disconnected from data centers housing such databases, and bandwidth limitations won't allow all of the data to be transmitted to the data center in the short windows where 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.

And then, 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 last decades that implemented painstakingly designed algorithms carefully tuned for anticipated usage scenarios are replaced with applications that are constantly being revised, or may even be self-adjusting their internal algorithms and configurations.

These **user** requirements – constant availability, frequent and seamless evolution, 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?

1.3 Cloud-Native Software

Our software needs to be up, twenty-four seven. We need to be able to release frequently to give our users the instant gratification they seek. The mobility and always-connected state of our users drives a need for our software to be responsive to larger and more fluctuating 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 we can run the software, have led directly to the emergence of a new architectural style for software – cloud-native software.

1.3.1 Defining Cloud-Native

What characterizes cloud-native software? Let's analyze these requirements a bit further and see where they lead – figure 1.4 takes the first few steps, listing requirements over the top and showing causal relationships going downward.

- Software that is always up must be resilient to infrastructure failures and changes, whether planned or unplanned. When properly constructed, deployed, and managed, composition of independent pieces can limit the blast radius of any failures that do occur – this drives us to a modular design. And because we know that no single entity can be guaranteed to never fail, we include redundancy throughout the design.
- Our goal is to release frequently, and monolithic software doesn't allow this; there are too many interdependent pieces that 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 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 always have on their persons. 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 dynamic scaling to continue functioning adequately.

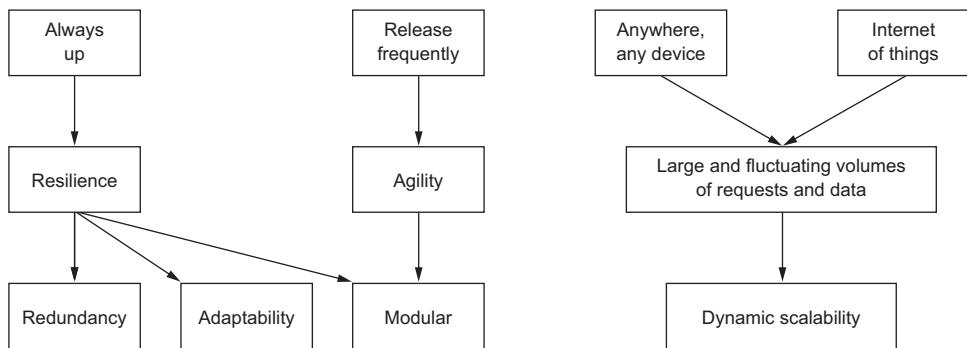


Figure 1.4 Requirements on our software drive toward the architectural and management tenets of cloud-native software.

Some of these attributes are architectural – the resultant software is made up of a composition of redundantly deployed, independent components. Other attributes address the management practices used to deliver the digital solutions – a deployment must 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 is constructed as a set of independent components, redundantly deployed in part for resilience, implies distribution – if our redundant copies were all deployed in the same failure zone, then we're at greater risk of failures having far reaching consequences. As a result, we deploy those modules in a

highly distributed manner. To make efficient use of the infrastructure resources we have, when we deploy additional instances of an app to serve increasing request volumes, we must be able to place them across a wide swath of that infrastructure, again leading us to wide distributions.

- Adaptable software is by definition “able to adjust to new conditions,” and the conditions we refer to here are those of the infrastructure. The way that software performs is partially determined by the context or conditions it’s running in, implying that as the infrastructure changes, the software changes. And because running software is part of the context, any changes in that software has rippling effects throughout the environment. In short, agility and adaptation cause constant change, driving the need for adaptability. Finally, dynamic scaling of the software topology implies constant change.

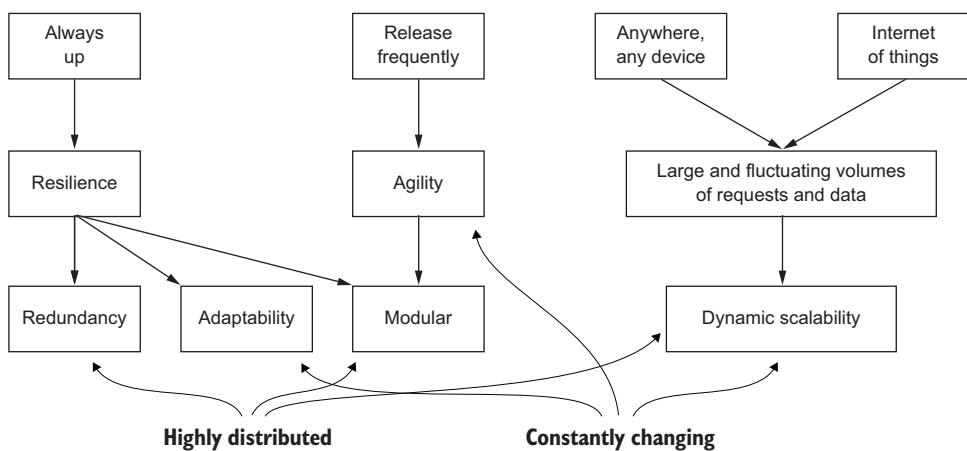


Figure 1.5 Architectural and management tenets lead to the core characteristics of cloud-native software – it's highly distributed and must operate in a constantly changing environment even as the software is constantly evolving.

To summarize, I give you the following definition:

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

Many more granular details go into the making of cloud-native software, the things that fill the pages of this volume, but ultimately, they all come back to these core characteristics. This will be our mantra as we progress through the material, and I will repeatedly draw us back to extreme distribution and constant change.

1.3.2 Cloud-Native Software – The Pieces and Parts

The key elements of cloud-native software are familiar to you. We have apps that implement key business logic, and those apps use other components, usually some

type of persistence like a relational database, as well as other services. For example, a movie-streaming site might have an app that allows a user to supply their personal information such as name and address, as well as payment information. It validates the credit card information via call to external services, and stores the user info in a database. Figure 1.6 is a depiction of this simple model.

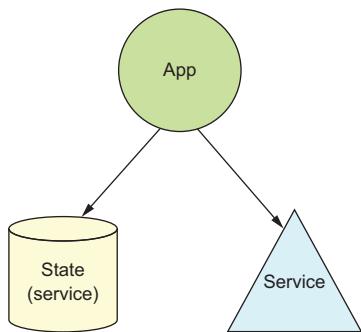


Figure 1.6 Figure 6 - Familiar elements of a basic software architecture.

When we take this familiar model and layer on the requirements for and characteristics of cloud-native software, things get interesting. Figure 1.7 shows a more complex topology.

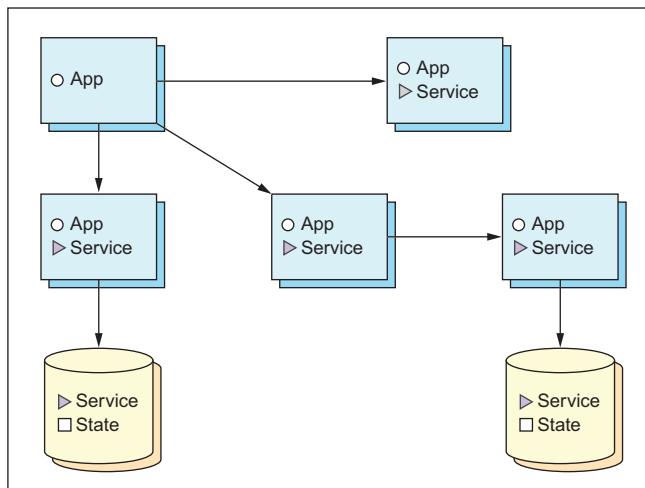


Figure 1.7 Cloud-native software takes familiar concepts and adds extreme distribution with redundancy everywhere and constant change.

The first thing you might notice in this diagram is that each of the entities is annotated with the role or roles they're playing within the cloud-native software. Yes, a component often plays a dual role. For example, most apps are services that other apps use. The credit-card validation service from the example above is an app that accepts data, verifies that the card number's valid by looking it up in a database, and invokes a

fraud-detection service to filter out bad requests. Similarly, entities responsible for managing state, like databases, are also acting as services.

Next, as we've already concluded, virtually every entity in the system has redundant "copies." For apps, this manifests itself as having multiple instances of an app deployed. For stateful services, though I depict that redundancy in the diagram with additional instances of the database, I ask you not to read this illustration too literally – data redundancy isn't necessarily realized through multiple instances of a database (though it might be). Data resilience and distribution is a complex topic that I'll cover in great depth in later chapters of the book.

Finally, it's the connections between the individual pieces that ultimately bring the independent components, whether different entities or multiple instances of a single one, into the collective whole that forms cloud-native software. Depicted in the diagram with arrows, the relationships these represent are complex, carrying with them details around interface contracts, communication protocols, even temporal considerations (synch vs. asynch, for example). And it's the connections between the components that are most markedly impacted by the constant change we must adapt to. These relationships are a first-class entity in the model we're developing for reasoning about cloud-native software, and we'll spend a lot of time studying patterns for effective use.

1.3.3 A Model for Cloud-Native Software

Adrian Cockcroft talks about the complexity of operating a car¹ – as drivers we must control the car and navigate streets, all as you make sure not to come into contact with other drivers performing the same complex tasks. We're able to do this only because we've formed a model that allows us to understand the world and control our instrument – in this case a car. Most of us use our feet to control the speed and our hands to set direction, collectively determining our velocity. In an attempt to make it more navigable, 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 us a framework in which we can reason about the journey we're taking from start to finish.

Writing cloud-native software is also complex. To aid in bringing order to the myriad of concerns in writing cloud-native software, I present a model through which we can reason about such systems. My hope is that this framework supports you in understanding the key concepts and techniques to make you a proficient designer and developer of cloud-native software. In figure 1.8 I bring you a slight modification of figure 1.7 – in some ways a simplification, but also bringing with it some nuances.

The three main parts to this model are the app, the data, and the collective formed when we put the pieces together.

- The Cloud-native app: This forms the bulk of the code that you write; it's the business logic for your software. Implementing the right patterns here allows those apps to act as good citizens in the composition. Although the composi-

¹ <http://clusterhq.com/2016/08/08/microservices-adrian-cockcroft/#transcript>

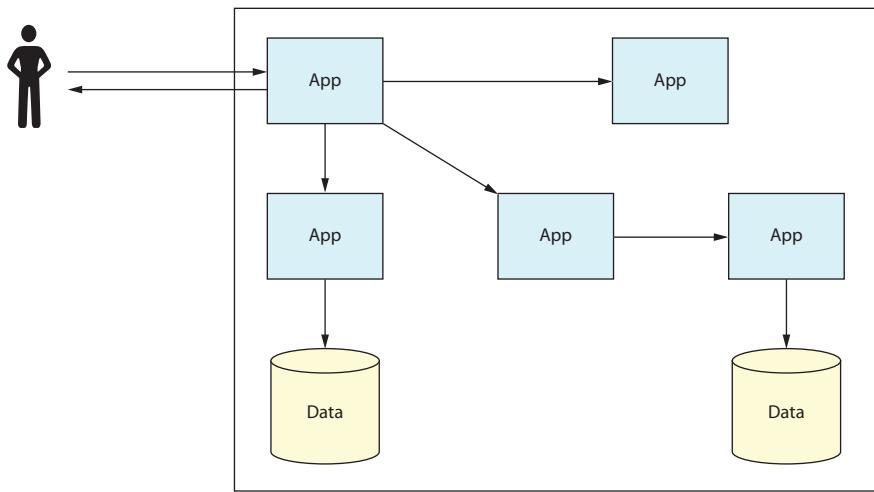


Figure 1.8 Key entities in the model for cloud-native software: apps, data, and the composition.

tion is important, I consider it a first-class concept in the model. It's essential that as you build the app, your design choices reflect the context in which the app lives.

- **Cloud-native data:** The way that data is handled in cloud-native software is both subtly and sometimes markedly different than the way it was handled in earlier system architectures. You might already notice, for example, that there are multiple data entities – what appear to be databases – present in the diagram, a departure from the shared database that has dominated the architectures of the last several decades. For cloud-native data, similar to cloud-native apps, the key is decomposition – breaking it into smaller pieces that provide the characteristics of cloud native that drive us to extreme distribution (recall figure 1.7). But the trick is dealing with the challenges that arise as a result of that componentization. For example, when a concept such as *customer* is present in numerous individual persistence services, how do we keep the values in synch across each of those? Loose coupling is harder when state is involved. We simplified the problem in the app tier by pushing state out of it, but then we must address it in our data tier.
- **The Cloud-native composition:** Once we connect a set of apps and data entities together, we have cloud-native software. But the compositions are more than connections of two entities. Any composition is likely a part of a broader collective, and brings with it transitive implications. Contracts that govern those connections must be properly formalized and managed. You might have noticed that in going from figure 1.7 to figure 1.8, I've dropped the “services” abstraction – this is because the key aspect of “serviceness” arises only once the (service) app, or data (service) is used by something else.

Managing the composition brings with it new challenges that require new tools and patterns to be effective. For example, if we begin seeing high latency for certain user requests, and such requests are being processed by a cloud-native composition, how can we find where the problem lies?

Notice that I've drawn the user into figure 1.8 because how they interact with the cloud-native software is part of the model.

Looking at these key abstractions in the context of the extreme distribution and the need to adapt to constant change is the work of this book. The following overview summarizes many of the considerations.

CLOUD-NATIVE APPS

Some of the concerns for cloud-native apps include:

- 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 we have multiple instances of an app, and even when only a single instance is being disrupted in some way, the app must be designed to be stateless.
- Configuration of the cloud-native app poses some unique challenges when many instances are deployed and the environments in which they are running are constantly changing.
- The dynamic nature of cloud-based environments necessitates changes to the way we manage the application lifecycle (not the software *delivery* lifecycle, but rather the startup and shutdown of the actual app). We must reexamine how we start, configure, reconfigure, and shut down apps in this new context.

CLOUD-NATIVE DATA

In figure 1.7, we analyzed the requirements for today's digital systems, ultimately concluding that cloud-native software is highly distributed and exists in an environment of extreme change. This has led to the employ of certain design patterns, such as making apps stateless. But handling state is an equally important part of a software solution, and the distribution and change tolerance characteristics apply equally to the data-handling portions of our systems.

That said, handling data in a cloud setting poses some unique challenges, hence it receives its own part in the holistic story. The concerns for cloud-native data include:

- We 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 different domains, and hence implemented in many different software systems, were best treated centrally as a single entity. For example, in a hospital setting the concept of a patient was relevant in many different settings including clinical/care, billing, experience surveys, and more, and we created 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. We need to create a distributed data fabric, as we created a distributed app fabric.

- The distributed data fabric is made up of independent, 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.
- 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 COMPOSITIONS

And finally, when we draw all of the pieces together, a new set of concerns surface for the cloud-native composition:

- 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, we 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 user request is served by the collective, not a single entity within it. As such, properly managing cloud-native software to ensure a good user experience is a task of managing the composition. Application metrics and logging information must be specialized for the highly distributed nature of the system.
- One of the greatest advantages of a modular system is the ability to more easily evolve a holistic system by evolving parts of it independently. Proper versioning of services and contracts that give the client and service parts of the relationship proper control over evolution are key considerations.

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

1.3.4 **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. You are also a registered user of their online banking application. After receiving only unsolicited calls on your home landline for the better part of the last year or two, you've finally

decided to discontinue it. As a result, you need to update that 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.

I want to superimpose the flow through the system over the entities depicted in figure 1.9.

- Because the main *User Profile* app has many instances deployed, requests to it, as depicted by the arrows from the stick figure to the app (recall that I explicitly included this relationship in the cloud-native collection), must be carefully orchestrated. In particular, we’ll use a router or load balancer to distribute requests over the multiple instances.
- The first request is for authentication and after providing your username and password, the *Auth API* is synchronously invoked and access is granted.
- Your next request, to submit the changes to your phone number, also comes to the software through the *User Profile* app, but because numerous instances of that app cause this second request to be served by a different instance (spoiler alert: sticky sessions are bad in cloud-native software!).
- In fact, the instance that served your first request might already have been replaced by another instance, implying the need for the router I mentioned being easily updated with software topology changes.
- That second request results in the *User API* being invoked, but the arrow between the *User Profile* app and the *User API* could represent an asynchronous invocation pattern, using something like a message bus to loosely couple the client from the service.

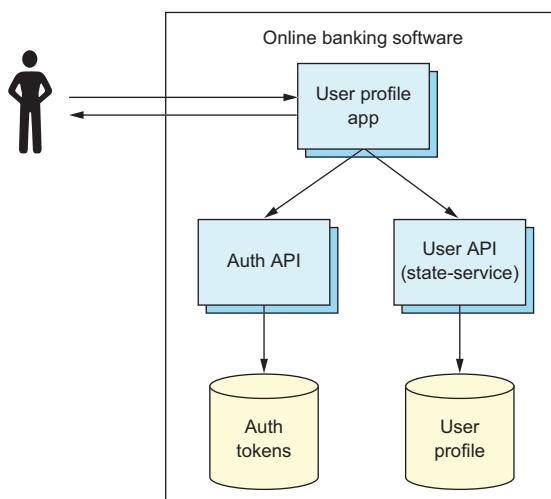


Figure 1.9 The online banking software is a composition of apps and data services.

In this simple example, we're already seeing several of the nuances of cloud-native software demonstrated: Scaled out, stateless apps bound to clients and services through protocols (synch and asynch) designed to serve the highly distributed, changing environment of the cloud. I want to carry this example a bit further to demonstrate a few more things.

I haven't explicitly stated it, but something you expect is that when you are back at the store branch and the teller verifies your current contact information, they also 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 we've identified as important for modern digital systems. Figure 1.10 shows this product suite.

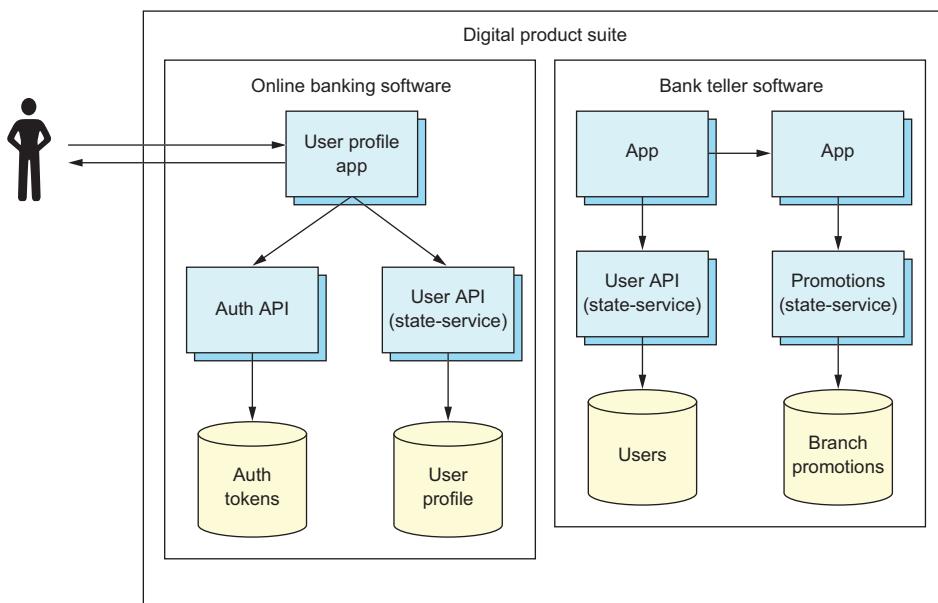


Figure 1.10 What appears to a user as a single experience with Wizard Bank is realized by independently developed and managed software assets.

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. I want to draw your attention to the fact that they both interact with a data service that manages information about the user. I alluded to the fact that in cloud-native software we lean toward loose coupling, even when we're dealing with data, and you can see two different user stores and two different User APIs (I'm calling the one in the bank teller software "User API-prime"). The question's how to reconcile the loose coupling in data management with the need for your phone number being updated in both places.

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. I use this example to illustrate part of what I summarized in the overview of cloud-native data of the previous section. For the time being, accept this as a problem statement; I promise we'll study solutions soon.

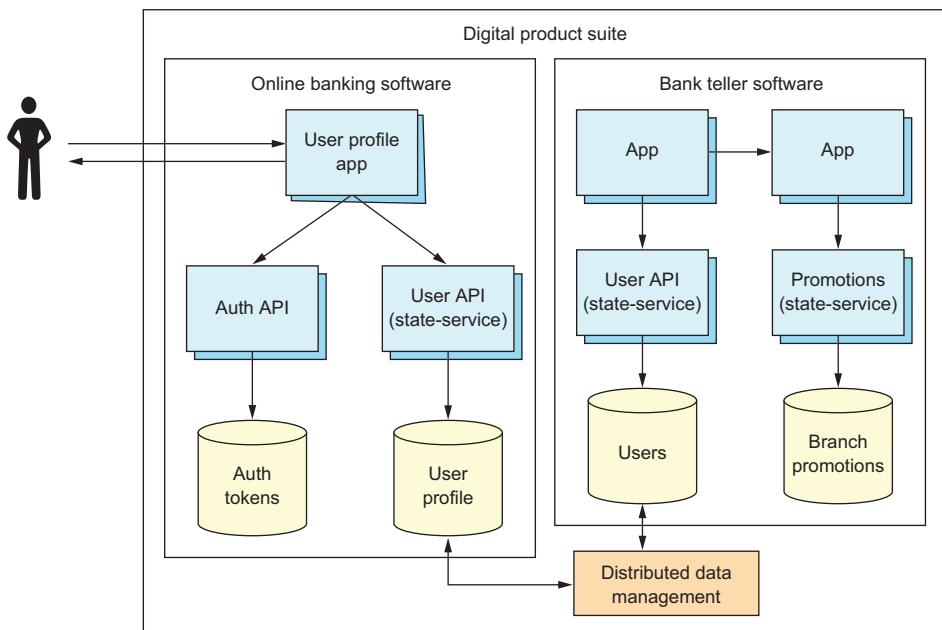


Figure 1.11 A decomposed and loosely coupled data fabric requires techniques for cohesive data management.

The key elements of the cloud-native software model, taken together with the patterns and practices we'll apply to them, give us a framework in which we can design and build our cloud-native applications to meet the requirements of contemporary digital solutions. Many people (including yours truly) are predicting that these architectures will come to dominate software designs for the next decade or more, like the client server models that have been the mainstay of the last twenty or thirty years. It's a new architecture for a new era.

Does this mean you must rewrite your existing applications, porting them over to the new model? Definitely, no. Well, maybe, yes. As with almost any real design question in computing, the real answer is that it depends. Let's dig in on that a bit.

1.4 **Cloud, Cloud-Native, None of the Above and Somewhere In-Between**

The narrative around “cloud” can be confusing. When I hear a company say “we’re moving to the cloud” they often, but not always, mean they’re moving some or maybe even all of their apps into someone else’s data center – like Amazon Web Services, Microsoft Azure, Google Cloud Platform, or any number of options. They might also be moving apps into “private clouds” which are on-premises data centers that share many of the characteristics of the commercial cloud offerings that I’ve mentioned. They’re built on a virtualized infrastructure; they serve things like virtual machines, and they operate via self-service models and APIs for provisioning resources.

Initially the cloud offered the same set of primitives that formed the foundation of their existing data centers – machines, storage, and network, albeit virtualized. When they moved to the cloud, public or private, they didn’t need to change much in their software – the apps depended on a set of primitives that were still available to them. But such a move to the cloud offered none of the benefits of resilience or agility that I’ve introduced. If I had a single instance of my large app deployed and I lost a server, or if one part of my application ran amok, my whole digital solution could go offline.

As an industry we didn’t stop there: we’ve evolved our software architectures for the requirements of the modern digital solution. And the moniker for that new architecture is “cloud native.” I want to clarify our definition of cloud native in this way:

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

With that clarification, we can consider what software is appropriate for the new cloud-native model, and, as important, what software isn’t. We also need to address how to move existing applications over to the new architecture, if we do so at all.

Let me start first with a relatively obvious case, when the software you’re building is not distributed; yes, there are still non-distributed systems. For example, code that is embedded in some physical device, 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, and if some part of the software or hardware fails, the worst that can happen is that I need to order out when my home-cooked meal is ruined.

But this doesn’t mean that all distributed apps will move over to a fully cloud-native model either. Although it’s been a subtle point until now, you might already have the sense that some of requirements of strong consistency and “cloud-native” approaches don’t lend themselves well to use cases where strong consistency is required. Although it isn’t a major problem if, due to some network problem, the movie recommendations you are served don’t immediately reflect the latest five-star rating a user supplied – providing recommendations based on stale data is preferable over not providing recommendations at all. On the other hand, 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 another branch when the two systems are momentarily disconnected. Eventual consistency is at the core of quite a number of cloud-native patterns, meaning that when strong consistency is required, those particular patterns can't be used.

But it's not all or nothing. When the mainframe that holds bank balance information is inaccessible (Figure 1.12), the bank teller application or an ATM can't dispense funds. In this case the banking customer isn't insulated from an infrastructure instability, as is one of the goals of cloud-native software; this behavior is preferable over allowing for transactions to complete in error.

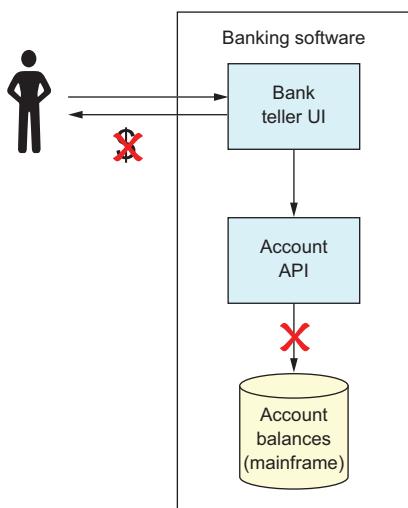


Figure 1.12 Dispensing funds without access to the source of record is ill advised.

But applying some cloud-native techniques can still be beneficial. For example, the bank teller application is ultimately connecting to the mainframe through some (micro)services. If we deploy many instances of those microservices across numerous availability zones, a network partition in one zone still allows access to mainframe through service instances deployed in other zones (figure 1.13).

The vast majority of you won't be starting from scratch. You might have been asked to move some existing software "to the cloud," or you're building a new piece of software that needs to interact with some of those existing systems. This raises the question of whether you need to rewrite those applications or leave them alone. The short answer is that it's usually somewhere in-between.

Netflix, for example, moved their entire customer-facing digital solution into the cloud, specifically into AWS. Eventually. The move took them seven years,¹ but they began refactoring **some** parts of their monolithic, client-server architecture in the process, with immediate benefits. As with the banking example above, the lesson is that even during a migration a partially cloud-native solution is valuable.

¹ <https://media.netflix.com/en/company-blog/completing-the-netflix-cloud-migration>

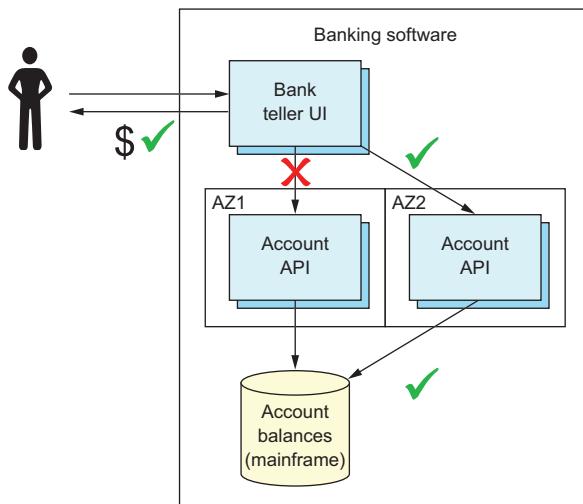


Figure 1.13 Applying some cloud-native patterns, such as redundancy, appropriately deployed, even when other patterns aren't appropriate, still brings benefit.

This isn't to say that every application should be at least partially refactored. As we still run mission critical applications on the mainframe, we'll continue running non-cloud-native software for decades to come. Migrations to a new architectural style should only be undertaken if there's clear business benefit to doing it.

Finally, if you're in the enviable position of creating a brand new digital solution that is entirely unencumbered by any legacy, I strongly encourage you to follow the cloud-native architecture. I say this with two cautions; First, be careful not to over-engineer too early. For example, let's say you're building a new site for sharing cat pictures (I'm a geek, you knew I'd have to bring that in somewhere!). Your vision for the product includes way more than sharing images; you also plan to use machine learning to drive a recommendation engine. You'll have state-of-the-art collaboration capabilities that allow distributed teams to come up with new memes, and you anticipate needing to scale capacity to handle traffic spikes when the latest shocking news comes out of Hollywood. You'll need stateless apps, distributed configuration, circuit breakers, event sourcing, CQRS, and much, much more. And yes, when you're at twenty million users, you'll absolutely need that.

But using cloud-native patterns brings with it some added burden – deployment, management, and perhaps even cognitive. Start small, get initial versions of your product to market, and when you see the uptick, then add on. All of that said, focus on the foundational patterns. Many of you are likely familiar with the extraordinary embrace of Pokémon Go in the summer of 2016, where the adoption went far beyond the creators' most optimistic predictions.¹ Had they not applied the fundamentals at the onset, they couldn't have met the demand. Remember that cloud native isn't only about meeting scaling demands; rather, the architectural tenets also provide resilience and agility in a constantly changing world.

¹ <https://gamerant.com/pokemon-go-peak-worldwide-projections/>

The second warning I offer on the thread of over-engineering is to start with the problem, not the solution. The new techniques you'll learn in this book are intellectually stimulating and genuinely fun. As engineers, we want to apply the cool new tech, but it may not always be warranted. You might be tempted, for example, to implement a reactive pattern where the two parts that you're connecting through that exchange are better run in the same process. As you look to apply the patterns, give careful consideration to the value that they bring.

1.5 **Summary**

We've entered a new era of computing. As design patterns and practices changed when the primary computing systems they ran on went from mainframes to client-server systems, they are changing now as we move our workloads to the cloud.

In particular, you've learned:

- 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 are around 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 that has three key entities:
 - The cloud-native app
 - Cloud-native data
 - The Cloud-native composite
- “Cloud” is about where software runs; “cloud native” is about how it runs.
- That 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.

Although most of this book is targeted at the application architect and developer, understanding modern Devops practices and the value they bring to the organization that produces digital products allows you to more fully appreciate the value behind applying many of the patterns I cover in this book. In the next chapter, we'll take a high-level tour of those operational concerns.



Running Cloud-Native Applications in Production

This chapter covers

- The motivation for developers to care about operations
- Obstacles to making software deployments easy and to keeping production systems running smoothly
- Techniques that eliminate those obstacles, allowing low-ceremony, frequent deployments, and easy production operations
- The central role that continuous delivery plays in well functioning IT operations
- How cloud-native architectural patterns support operational concerns

As a developer, you want nothing more than to create software that users will love and 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 to them with ease. And you want your software run well in production – always be available and responsive.

Unfortunately, for most organizations the process of getting software deployed in production is rather challenging. Processes designed to reduce risk and improve efficiencies have the inadvertent effect of doing exactly the opposite because they're slow and cumbersome to use. And once 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.

Given a body of well-written, completed software, it is:

- 1 Hard to get it deployed
- 2 Hard to 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; rather 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, quite 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 us to first understand what is causing the greatest pains in the current one. Once we've analyzed the obstacles we currently face, we can construct a new system that not only avoids the challenges, but 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 production operations are 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 there to be a fair bit of conflict between application development groups and operations teams, with each blaming the other for failures to adequately serve consumers with superior digital experiences.

But as I said, it isn't the fault of the ops team, nor of the app-dev team, rather the challenges come from a system that inadvertently erects a series of obstacles to success. Although every challenging situation is unique with a variety of detailed root causes playing a part, there are a number of themes that 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 both to trouble with initial deploys and to lack of stability once 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's highly complex with a great many tightly coupled, interrelated components. As such, there's a great risk 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 affect of limiting the frequency with which we 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. Although this philosophy was probably always suspect, particularly now with IT systems being quite complex and highly distributed, this expectation of infrastructure stability is a complete fallacy¹. As a result, any instability in the infrastructure propagates up into the running application, making it hard to keep running.
- And finally, because doing deployments into an unstable environment is usually inviting more trouble, the frequency of production deployments is limited.

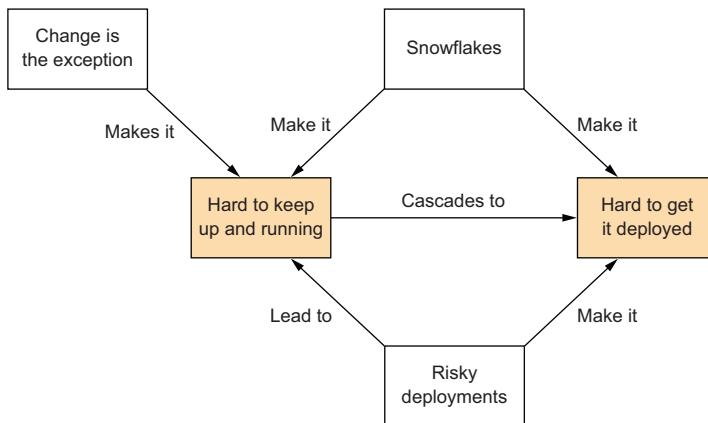


Figure 2.1 Factors that contribute to the difficulty in deploying software and keeping it running well in production.

Let's explore each of these factors a bit further.

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 they reach 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 software delivery lifecycle (SDLC). This variability occurs along two lines:

¹ https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

- 1 A difference in environments
- 2 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 is lacking or different in another one. One obvious example of this is when there are differences 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 a 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, the tests are no longer passing and the 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 actual artifact being deployed also varies through the SDLC. Even when environment-specific values aren't hard-coded into the implementation (which none of us would ever do, right?). Property files often contain configurations 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 recompile or repackaging of the deployable artifact can, and often does, end up bringing in other differences as well.

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

It's a difference in environment. Users connect to the system through sockets, socket connections require open file descriptors, and there's a configuration setting that 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 we ran in UAT didn't test for what we'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 on when a new

server is configured and the software is moved over to it, 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 we finally have everything deployed and running and now something changes in my infrastructure topology – my server name, URL, or IP address changes, or we add some servers for scale. If those environment configurations are in the property file, then I must recreate the deployable artifact and I 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 2AM 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 some downtime during an upgrade, or for a deployment to cause unexpected downtime. Downtime’s 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 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 prod. Perhaps the practice with the highest price tag in terms of person and infrastructure resources is a testing process that depends on doing trial runs on an “exact replica of production.” In principle, none of these things 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 exemplar: 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, which is complicated by a multitude of added 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 where we 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 pre-production environment. How should these two teams' activities be coordinated? One option is to test the applications one at a time, and although there's an obvious schedule impact of executing the tests in sequence, the happy path is relatively tractable.

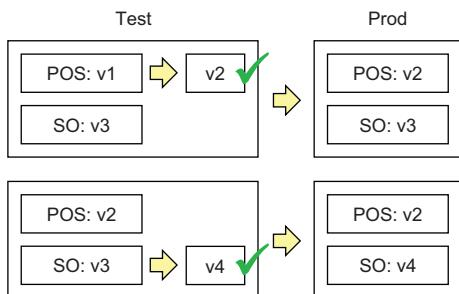


Figure 2.2 Testing two apps in sequence is straightforward when both tests pass.

Figure 2.2 shows the following two steps. First, version 2 of the PoS app is tested with version 3 (the old version) of the SO app. When it's successful, version 2 of the PoS application is deployed into production – both test and prod are now running v2 of POS, and both are still running v3 of SO. The *test* environment is a replica of *prod*. Now we can test v4 of the SO system and when all the tests pass we can promote that version into production and both application upgrades are complete.

But what happens if tests fail for the upgrade to the PoS system? Clearly, we can't deploy the new version into production and allow v1 of the POS and v3 of the SO to continue to run there, and we must revert to v1 of the POS app in the *test* environment. But then the question remains whether we can move forward with the testing of version 4 of the SO system? Did SO v4 already depend on POS v2? And when the POS team is ready for the next test, have they made any assumptions about the version of the SO software running? Figure 2.3 poses exactly this question.

My goal isn't to solve this problem here, but rather to demonstrate that even an over-simplified scenario can quickly become extraordinarily complicated. I'm sure you can imagine that when we add more applications to the mix and/or try to test new versions of multiple applications in parallel that the process becomes completely intractable. The environment that is designed to ensure 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 business have time periods in the year where new deployments into production aren't permitted. For health insurance companies it's the open-enrollment period. In ecommerce in the United States it's the month between Thanksgiving and Christmas. That Thanksgiving to Christmas timeframe 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 on some regular interval 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.

2.1.3 Change Is the Exception

Over the years I have had dozens of conversations with CIOs and their staff where they express 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 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 having developers involved in initial deployments – there's a fair bit of uncertainty during fresh rollouts and involving the team that deeply understands the implementation is essential. But at some point,

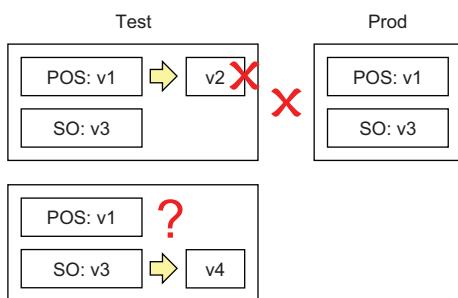


Figure 2.3 A failing test immediately complicates the process for pre-production testing.

responsibility for maintaining the system in production is completely handed over to the ops team and the information for how to keep things humming's provided to them in a "run book." The run book details possible failure scenarios and their resolutions, and although this sounds good on principle, on deeper reflection it demonstrates an assumption that the failure scenarios are known – 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 we've already established that in the cloud it will, systems will persist in experiencing instability.

2.1.4 **Production Instability**

All of the factors I've covered until now inarguably help to keep 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. Coordinating those rare moments with 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. But the insight of how these factors negatively impact operations today comes from studying well-functioning organizations – those from born-in-the-cloud companies. When we apply the practices and principles as they do, we develop a system that optimizes for 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 this using the techniques I've described. A different way exists.

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

- 1 Easy and frequent releases into production, and
- 2 Operational stability and predictability.

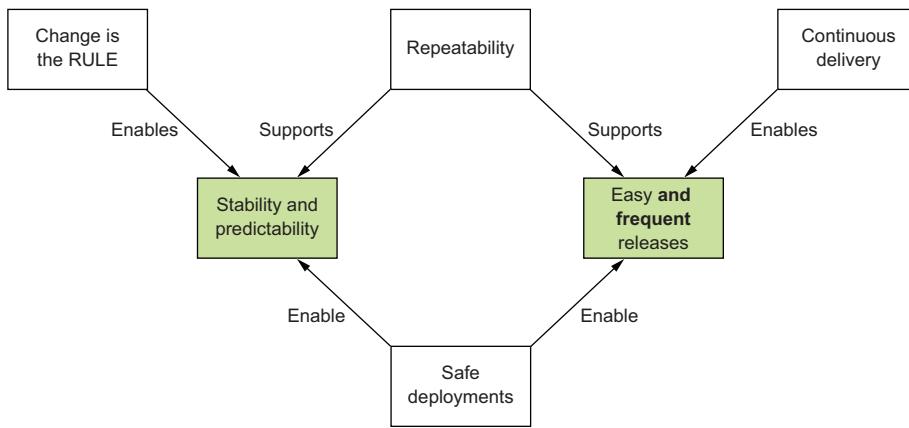


Figure 2.4 Explicit attention to these four factors develops a system of efficiency, predictability, and stability.

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

- Where “snowflakes” had previously contributed to slowness and instability, repeatability supports the opposite.
- Where risky deployments had caused both of the negatives, the ability to do deployments in a safe manner drives agility and stability.
- Replacing practices and software designs that depended on an unchanging environment with ones that expect constant change radically reduces time spent fighting fires.

But looking at this diagram 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 software development lifecycle (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'll first explain what CD is, how basic changes in the SDLC enable CD, and what the positive outcomes are. I'll return to the other three key enablers and describe their main attributes and benefits in some detail.

2.2.1 **Continuous Delivery**

Amazon may be the most extreme example of frequent releases – they're said to release code into production for Amazon.com on average every second of every day. Although you might question the need for such frequent releases in your business, and sure, you probably don't need to release software 86,000 per day, 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 is 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.

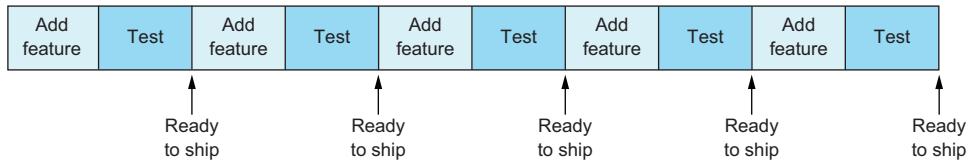


Figure 2.5 Every dev/test cycle doesn't result in a ship; rather, every cycle results in software that is ready to ship. Shipping then becomes a business decision.

Contrast this to 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 where a great many features are added to an implementation. Once a predetermined set of new capabilities have been added, an extensive test phase is completed and the software is readied for release.

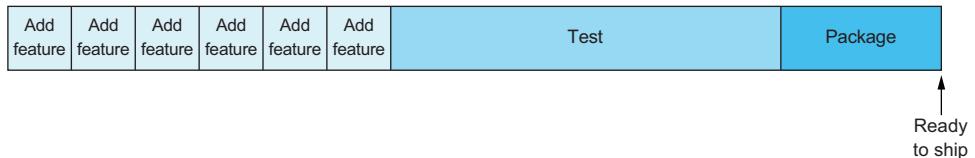


Figure 2.6 Traditional software delivery lifecycle frontloads a large amount of development work and a long testing cycle before creating the artifacts that can then be released into production.

Let's assume that the time span covered by each of figure 2.5 and 2.6 is the same, and the start of the process is on the left, and the “Ready to Ship” is on the far right of each diagram line up in time. If we look at that right-most point in time alone, we might not see much of a difference in outcome, roughly the same features will be delivered at roughly the same time. If we dig under the covers a bit we'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 complex, unpredictable software development process. For example, let's say we learn that a competitor is planning a release of a similar product to ours in two weeks, and as a result the business decides that we should make our own product immediately available. The business says, “Let's release now!” Overlaying that point in time over the previous two diagrams in figure 2.7 shows a stark contrast.

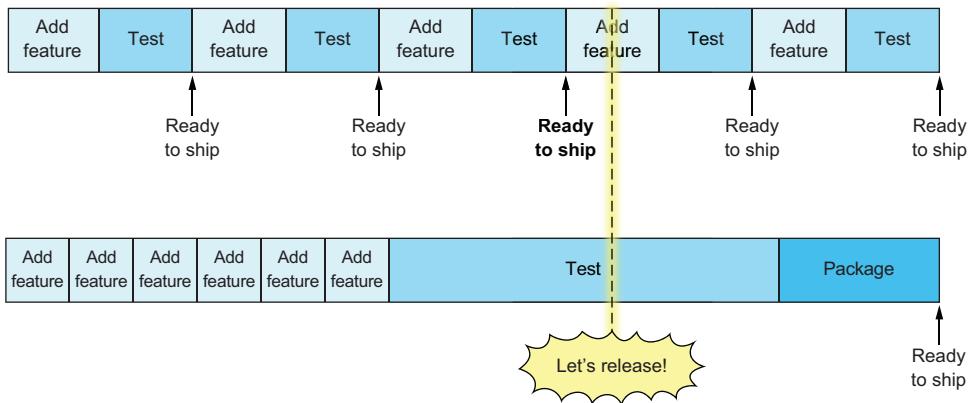


Figure 2.7 Continuous Delivery is concerned with allowing business drivers, not IT readiness, to determine when software are shipped.

Using a software development methodology that supports CD allows the “ready to ship” software of the third iteration (shown in bold) 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 diagram we 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!

Another important outcome that the iterative process affords is that when the “ready to ship” versions are frequently made available to customers, it gives us 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 it is our inherent optimism and 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, and this usually puts us behind schedule even before we begin.

Unanticipated implementation challenges are always expected – say we underestimated the effect of network latency on one part of our implementation, and instead of the simple request/response exchange that we planned for, we now need to implement a much more complex asynchronous communication protocol. Although we’re implementing this next set of features, we’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 our stretch goals fit within an already challenging time schedule.

The impact these factors have on the old-school development process is that we miss our 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 on what we 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.

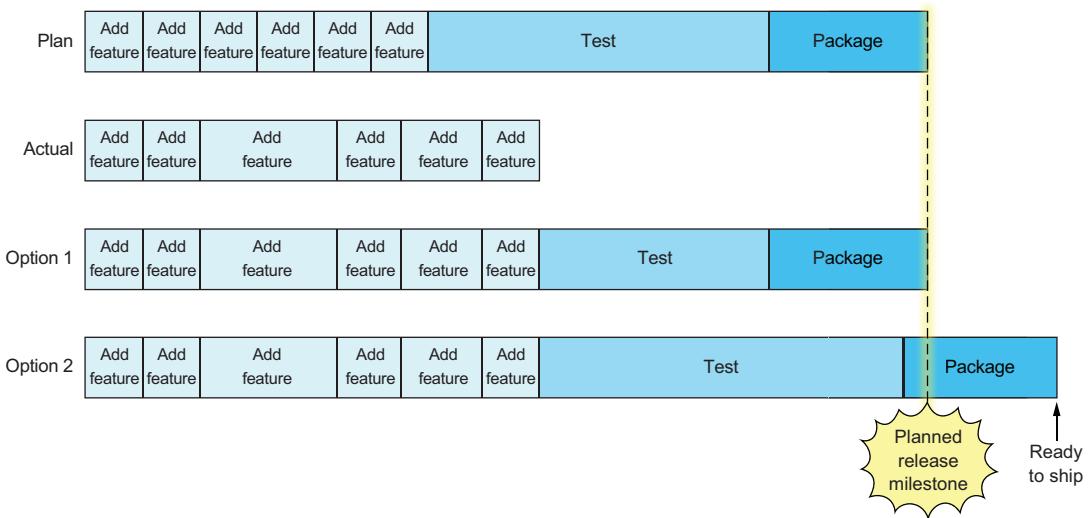


Figure 2.8 When the development schedule slips, we need to decide between two unpalatable options.

Contrast this to the effects that “unanticipated” development delays have on a process that implements many shorter iterations. Depicted in figure 2.9 we again see that our planned release milestone’s expected to come after six iterations. When the actual implementation takes longer than expected we see that we’re presented with some new options. We can either release on schedule with a more limited set of features (option 1), or we can choose a slight or a longer delay for the next release (options 2 & 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, we make deployments less risky, and therefore do them more frequently, we can do 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

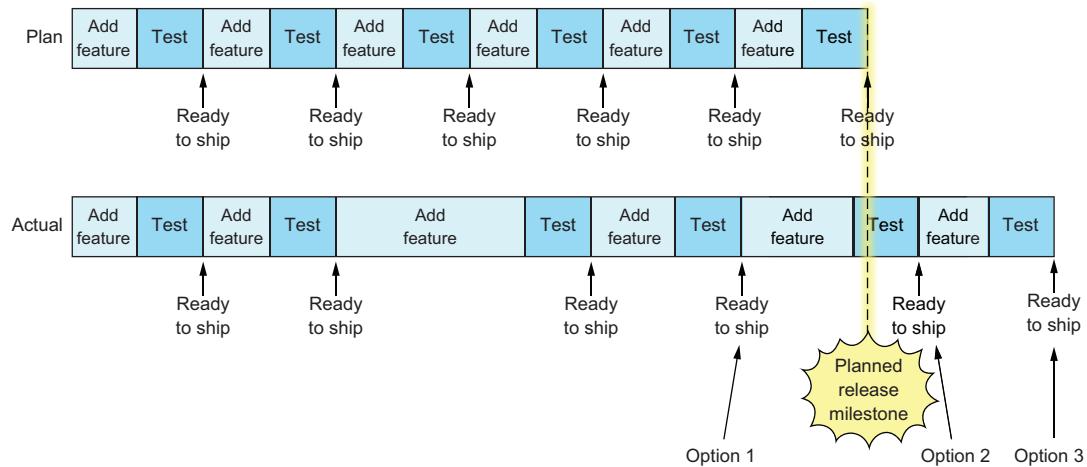


Figure 2.9 Shorter iterations designed for Continuous Delivery allow for an agile release process while maintaining software quality.

in the awkward position of trading off near-term market pressures with long-term goals of software quality and ability to evolve.

NOTE By contrast, 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 we must constantly adjust to differences in both the environments into which we're deploying, and in the variability of the artifacts we're deploying. That same inconsistency makes it extremely difficult to keep things running well once in production, because every environment and piece of software get special treatment anytime something changes. Drift from a known configuration's a constant threat to stability when we can't reliably recreate the configuration that was working before a crash.

When we turn that negative to a positive in our enabling system the key concept is repeatability. It's analogous to the steps in an assembly line – each time we attach a steering wheel to a car we 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 we 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 prod, whether we'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 we have with 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. In order to achieve the needed repeatability, we must be disciplined. In particular we need to:

- Control the environments into which we'll deploy the software.
- Control the software that we're deploying – also known as the deployable artifact.
- Control the deployment processes.

CONTROL THE ENVIRONMENT

In an assembly line we 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 $\frac{3}{4}$ -inch socket wrench each time we need it, because it's always in the same place. In software, the way that we consistently lay out the context in which the implementation runs is through two primary mechanisms.

First, we must begin with standardized machine images – in building up environments we must consistently begin with a known starting point. Second, changes applied to that base image to establish the context into which our software is deployed **must be coded**. For example, if we begin with a base Ubuntu image and our software requires the Java Development Kit (JDK), we'll script the installation of the JDK into the base image. The term often used for this latter concept is “infrastructure as code.” When we need a new instance of an environment then, we begin with the base image, apply the script, and we are 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 the control post 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 environments, 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 controlled 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 is 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 organizations 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 some differences in environments. In production, your software connects to your live customer database, found at a URL like `http://prod.example.com/customerDB`; in staging it connects to a copy of that database which has been cleansed of personally identifiable information and is found at `http://staging.example.com/cleansedDB`; and during initial development there may be a mock database which is accessed at `http://localhost/mockDB`. Obviously, credentials differ from one environment to the next. How do we account for such differences in the code we're creating?

I know we aren't hard-coding such strings directly into our code (right?) – likely we're parameterizing our code and putting these values into some type of a property file. This is a good first step, but there's often a problem in the details of how we do this: 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 environments. And therein lies the problem. When the environment specific settings are compiled in, the `.jar` file that I deploy in the test environment's different from the `.jar` file that I deploy into prod – see figure 2.10.

As soon as we build different artifacts for different stages in the SDLC, repeatability may be compromised. The discipline for controlling the variability of that software artifact, ensuring the only difference in the artifacts is the contents of the property files, must now be implanted into the build process itself. Unfortunately, because the `jar` files are different, we can no longer compare shas to verify that the artifact that I've deployed into the staging environment is exactly the same as that which I've deployed into prod. And if something changes in one of the environments and one of the property values changes, I 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. In order to make this happen, the code needs to be structured in the right way – for example, property files don't carry envi-



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.

ronment-specific values, rather they are themselves. We 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 these pieces come together in a controlled, repeatable manner. Figure 2.11 depicts the desired outcome – in all stages of the SDLC, we can reliably stamp out exact copies of as many running units as needed.

This picture has no snowflakes. The deployable artifact, the app, is exactly the same across all deployments and environments. The runtime environment has some variation across the different stages, but as indicated by the different shades of the same grey coloring, the base is the same and has only different configurations applied, like database bindings. Within a lifecycle stage all of the configurations are the same – they have exactly the same shade of grey. Those anti-snowflake 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 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 ques-

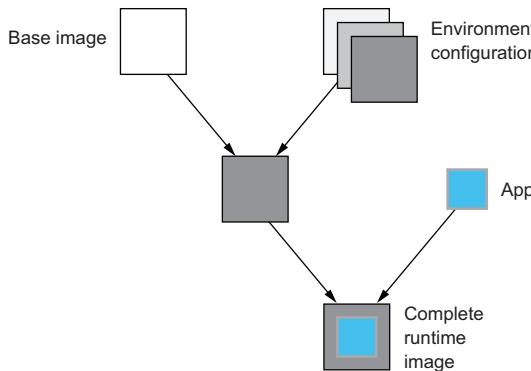


Figure 2.12 Assembly of standardized base images, controlled environment configurations and single deployable artifacts are automated.

tions 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, as in the right composite box at the bottom of the figure, is to automate.

Although the use of continuous integration tools and practices is fairly ubiquitous in the development phase of writing software, 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 A¹PI to download new "stem cells," the base images into which apps are deployed, from a software distribution site,

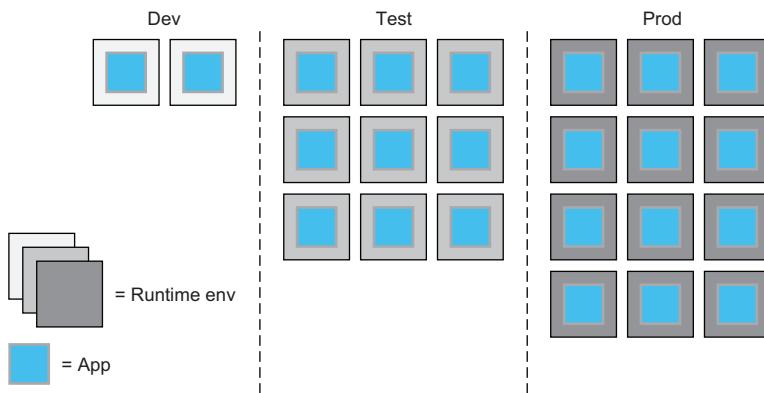


Figure 2.11 Desired outcome is to be able to consistently establish apps running in standardized environments. Note that the app is the same across all environments; runtime environment is standardized within a SDLC stage.

¹ <https://network.pivotal.io/docs/api>

and use pipelines to complete the assembly of the runtime environment and the application artifact. Another pipeline does the final deploy 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 functions) happy.

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

2.2.3 Safe Deployments

Earlier I talked about risky deployments and how 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 is no alternative – the only way to know that something works when deployed into prod is to test it first, but, I suggest that it is 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.

Egads! 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 a trapeze artist lets go of one ring, spins through the air and grasps another, they most often achieve their goal and entertain spectators. No question 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 and they perform over a safety net.

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 we can minimize the chance of failure. But eliminating it entirely isn't the goal – expecting failure, and failure happens, greatly lessens the chances of it being catastrophic. Perhaps a small handful of users will receive an “aw snap”¹ page and need to refresh, but overall the system remains up and running.

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.

¹ <http://www.urbandictionary.com/define.php?term=Aw%20Snap>

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

First, if we eliminate the complex and time-consuming testing process that I described in section 2.1.2, and instead we go straight to production following some basic integration testing, a great deal of time's 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 us 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, we're better able to address the failings of what we're currently running in production, allowing us to maintain a more stable system as a whole.

Let's then talk about 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. We'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 $n-1$. Coupled with the fact that the things we deployed were large pieces of software encompassing a wide range of capabilities, when the unexpected happened the results could be catastrophic. An entire mission critical application could experience significant down time, for example.

At the core of our safe deployment practices is the parallel deployment. Instead of completely replacing one version of running software with a new version, we keep the known working version running as we add a new version to run alongside it. We start out with only a small portion of traffic routed to the new implementation and we watch what happens. We 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 we look at data. Is the implementation running without crash? Has new latency been introduced? Have click-through rates increased or decreased?

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

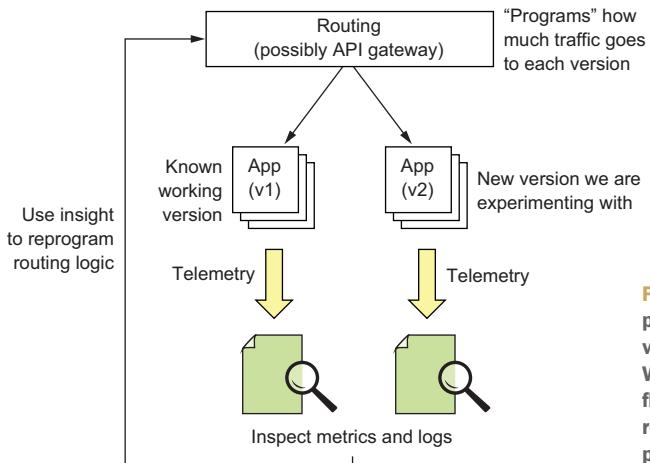


Figure 2.13 Data tells us how parallel deployments of multiple versions of our apps are operating. We use that data to program control flows to those apps, supporting safe rollouts of new software in production.

Figure 2.13 shows how the core practice works, but 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:

- Software artifacts must be versioned, and the versions must be visible to the routing mechanism to allow it to appropriately direct traffic. Further, because we'll be analyzing data to determine whether the new deployment's 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 of the 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, that the routing software "implementation" can be realized by configuring some of the components of your infrastructure. Other times we may want more sophisticated routing logic and need to write some code to realize it. For example, we may want to test some geographically localized optimizations and only want to send requests from within the same geography to the new version. Or perhaps we wish to expose a new feature only to

our 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, say the catalog, search engine, image service, recommendation engine, the shopping cart, and the 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 microservice-based architecture is directly linked to the operability of digital solutions.¹

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 three, all four of these things – 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 us to adapt when change is thrust upon us affords far greater resilience. What's subtle is 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. We respond to an outage, get everything in working order and we're done - until the next incident.

"Done."

This is the root of the problem. We believe that after we're finished with a deployment, responding to an incident, or making a firewall change that we've somehow

¹ <https://puppet.com/resources/whitepaper/state-of-devops-report>

completed our work. The idea that we're "done" inherently treats change as something that causes us to become not-done.

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 – rather 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, 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.

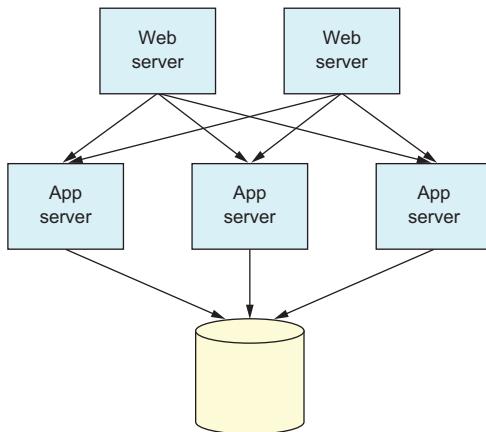


Figure 2.14 The desired state of our deployed software

We'd hope that, at some point, even most of the time, we have that system entirely established and running well, but we'll never assume that things remain as we left them immediately following a deploy. Instead we treat the *actual state*, a model of what is currently running in our system as a first-class entity, constructing and maintaining it using some of the metrics we already considered in this chapter. The eventually consistent system then constantly compares the *actual state* to the *desired state* and when there is a deviation, performs actions to bring them back into alignment.

For example, let's say that we lose an application server from the topology I laid out in the example above. 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 the app server off from other parts of the system. Figure 2.15 then 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, networked into the topology, and the application must be installed and started thereon (recall earlier discussions around repeatable deployments).

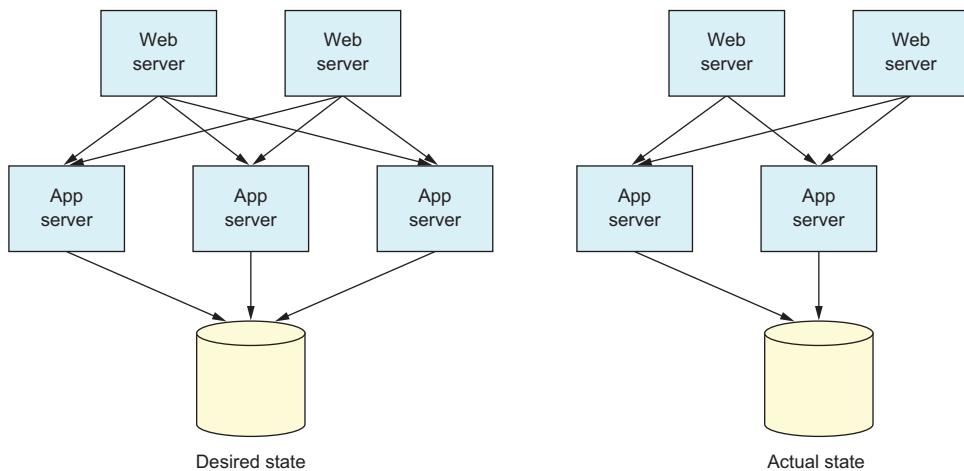


Figure 2.15 When the actual state doesn't match the desired state, the eventually consistent system initiates actions to bring them back into alignment.

For those of you who previously might not have done much with eventual consistency, this might feel a bit like rocket science. In fact, I have an expert colleague who avoids using the term because he worries that it'll invoke a fear in our customers. But systems built on this model are increasingly common and there are a great number of tools and educational materials to 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 – better to embrace it than to react to it.

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 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 and even further de-risks those deployments. Adopting a mindset that change is the rule changes the nature of the managing software in production.

2.3 **Summary**

In this chapter, you learned:

- In order for value to be realized from the code we write we 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 things shouldn't be blamed on developers or operators, rather 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.
- We can invert each of these negatives – focus on repeatability, making deployments safe, and embracing change – and create a system that supports rather than hinders the practices we desire.
- Repeatability is at the core of optimized IT operations, and automation applies not only to the software build process, but also the creation of runtime environments and the deployment of applications.
- Software design patterns as well as operational practices must be designed for the constant change in cloud-based environments.
- That the new system depends on a highly iterative SDLC that supports continuous delivery practices.
- That 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 rules in a system where change is the rule.



The Platform for Cloud-Native Software

This chapter covers

- A brief history of cloud platform evolutions.
- The use of a platform throughout the entire software development lifecycle.
- The operational benefits offered by a cloud-native platform.
- The support that such a platform offers to microservice-based software.

I work with a lot of clients to help them understand and adopt both cloud-native patterns and practices, and also a platform that is optimized to run the software they produce; in particular, I work with and on the Cloud Foundry platform. One of those clients who adopted Cloud Foundry and deployed an existing application onto it had an enlightening experience.

Although the software they deployed wasn't fully cloud native, in that it didn't employ *all* cloud-native patterns, but it did adhere to a few – the apps were stateless and were bound to backing services that held the needed state. After moving the application over 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 on Cloud Foundry, but reviewing the logs they found something quite surprising. The application was, in fact, crashing as frequently as it had been before – but they hadn't noticed it. The cloud-native application plat-

form 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.

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 of 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 digital solutions.

Now, let me be clear here – I'm not suggesting that such a platform 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 sub-par 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 earlier parts of this book, requirements for software which 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 rather than build yourself.

3.1 The Cloud(-/native) Platform Evolution

Arguably cloud platforms began in earnest with Amazon Web Services. Its first offerings, made publicly available in releases throughout 2006, included compute (Elastic Compute Cloud (EC2)), storage (Simple Storage Service (S3)) and messaging (Simple Queueing Service (SQS)) services. This was definitely a game changer in that developers and operations personnel no longer had to procure and manage their own hardware using the self-service interface, and could obtain the resources they needed in a fraction of the time it had previously taken. 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 “cloud” was only about where computing was happening.

Almost at the onset characteristics of the cloud began to put pressure on software that was built for pre-cloud infrastructures. Instead of using “enterprise-grade” servers, network devices, and storage, Amazon Web Services used commodity hardware in their 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 their software and offerings, and presented abstractions to their 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 exposing these new primitives to the user of the service, the organization which is running its software on AWS for example, places new responsibilities on the user to appropriately use those abstractions. We may not have realized it

at the time, but exposing these new abstractions in the API of the platform began influencing a new architecture for software. Software expressly designed to run well on such a platform.

Amazon Web Services effectively created a new market and it took competitors, such as Google and Microsoft, two years to have any response. When they did, they each came with unique offerings. Google's first was 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 abstraction to give the user control over server pool affinity or anti-affinity. By contrast, the Google App Engine 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, and, for example, one of the things that they entered the market with was a capability to run “Medium Trust Code.” Similar to Google's approach, the Medium Trust Code 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 infrastructure, thereby offering certain security and resilience guarantees.

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 if 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 is characterized by these higher-level abstractions, services tied to those, and the affordances they bring.

The cloud-native platform is what we'll study here. Let's start out at the level of abstraction that is surfaced in it.

	AWS	GCP	Azure
Compute	Elastic compute cloud (EC2)	Google compute engine	Azure virtual machines
Storage	Simple storage service (S3)	Google storage	Azure blob storage
Network	Virtual private cloud (VPC)	Virtual private cloud (VPC)	Virtual private network (VPC)

Figure 3.1 Infrastructure as a Services (IaaS) offerings from major cloud platform providers.

3.2 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 not only were required to configure application deployments correctly, but they were also responsible for proper configuration of the infrastructure those applications ran on. The cloud-native software platform separates the concerns of infrastructure configuration and maintenance from application configuration and maintenance. To make this clear, let's look at a specific example.

To make sure that a piece of software is running well, or to diagnose when things go wrong, personnel must have access to log and metric data for their apps. As we've already established, cloud-native apps have multiple copies deployed, both for resilience and scale. If we're running those modern apps on an infrastructure-centric platform, one that exposes traditional infrastructure entities such as hosts, storage volumes, and networks, we must navigate through traditional data center abstractions to get our 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 Once 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 his job done are CMDBs, hosts, and directories.

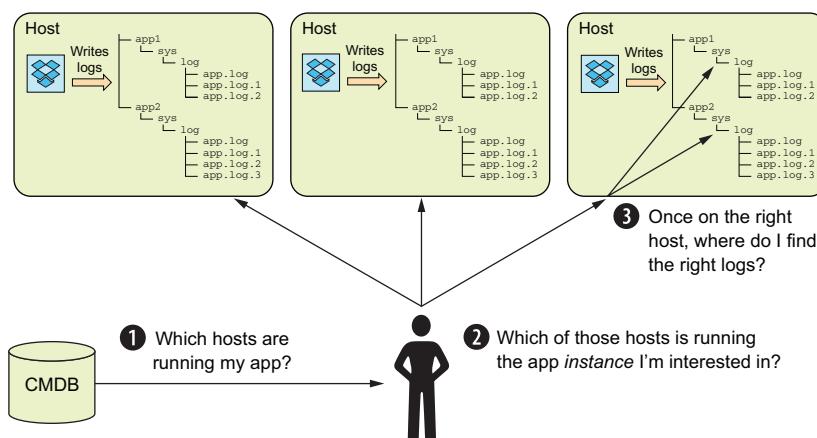


Figure 3.2 Accessing application logs in an infrastructure-centric environment is tedious.

By contrast, figure 3.3 shows the operator experience when the apps are running on a cloud-native platform. It's extremely simple: the operator asks for the logs for their application.

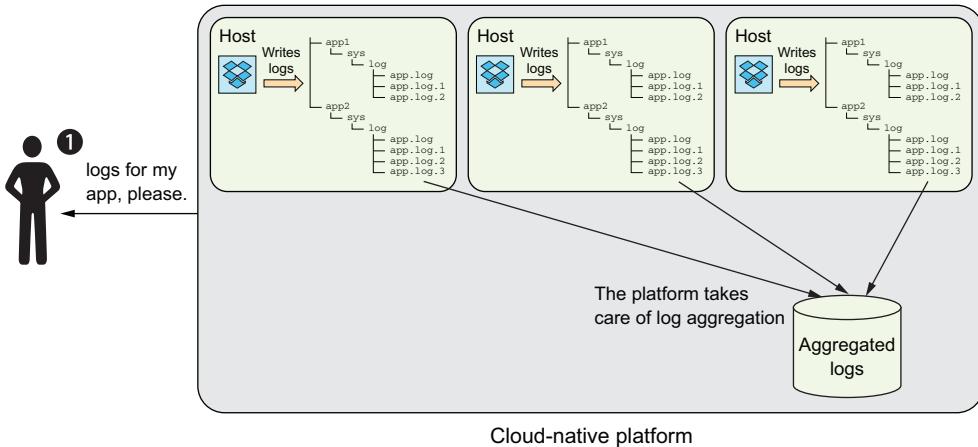


Figure 3.3 Accessing application logs in an app-centric environment is simple.

The entity that the operator is interacting with to get his job done is the app.

The cloud-native platform takes on a burden that was previously placed on the operator – it natively maintains an understanding of the application topology (that which was previously stored in the CMDB), uses it to aggregate the logs for all application instances, and provides the operator the data they need for the entity they're interested in. The key point is this: the entity that the operator is interested in is the **application**; he isn't interested in the hosts the app's running on, or the directories that hold the logs. He needs the logs for the application he's diagnosing.

The contrast that we see in this example is one of infrastructure-centricity vs. 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.” Let me define these:

INFRASTRUCTURE DIAL TONE Infrastructure as a Service (IaaS) platforms present “infrastructure dial tone”: an interface that provides access to hosts, storage, and networks – infrastructure primitives.

By contrast:

APPLICATION DIAL TONE 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 different 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.

The annotations on either side of the stack in figure 3.4 suggest differences in the operations performed against these different 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, she applies a policy to the application and the platform takes care of securing the application container. Instead of accessing hosts to get to logs for the apps, she accesses the logs for the app. The experiential differences that a cloud-native platform offers over an IaaS platform are significant.

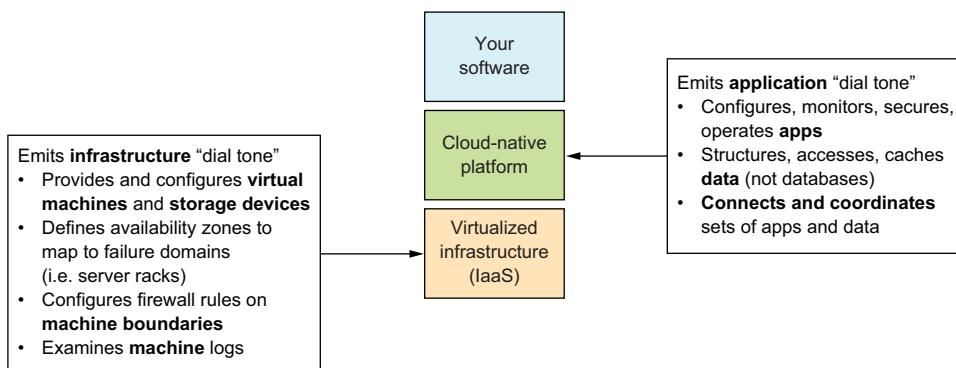


Figure 3.4 The cloud-native platform abstracts infrastructure concerns the team building business software solutions.

Establishing these boundaries and contracts enables something powerful – the exact separation of concerns we've been aiming for; it allows us to form separate teams. One team is responsible for providing the cloud-native platform that is used by application teams (and there will be many of these). Members of this platform team have a particular skills profile – they know how to work with infrastructure resources, how to set them up and keep them running well. Second, we have those application teams whose members build and operate software for end consumers. They know how to monitor those apps and optimize their performance. Each team is staffed with members that have the skills to address the needs of that particular layer. Figure 3.5 defines the platform team that consumes the infrastructure dial tone provided by the virtualized infrastructure to provide a cloud-native platform, and an application team that consumes the application dial tone to produce digital solutions.

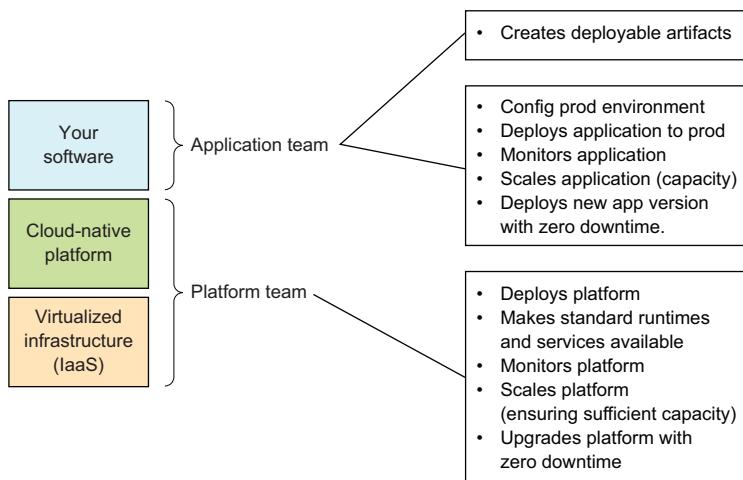


Figure 3.5 The right abstractions support the formation of autonomous platform and application teams. Each is responsible for deployment, monitoring, scaling and upgrade of their respective products.

With the right contracts in place, the application teams and the platform team are autonomous – each can execute their responsibilities without extensive coordination with the others. It's interesting to note how similar their responsibilities are – each team is responsible for deployment, configuration, monitoring, scaling, and upgrade of their 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 also on the inner workings of the cloud-native platform itself. The platform must support the continuous delivery practices that are essential to achieving the agility we require. It must enable operational excellence, disallowing snowflakes and enabling app team autonomy, concurrently maintaining security, regulatory, and other controls. And it must provide services that lessen the burdens that are added when we create software composed of a large number of highly distributed app components (microservices) running in a multi-tenant environment.

3.3 **The Platform Supports the Entire SDLC**

Continuous delivery can't be achieved by automating deploys into *production* – success begins early in the software development lifecycle. We've established that a single deployable artifact that carries through the SDLC is essential. What we 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 a developer verifies that the code is running on her own workstation, she checks in the code and 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, the

developer can move on to implementing the next feature, and the cycle continues. Figure 3.6 depicts these deploys into the dev environment. The dev environment contains lightweight versions of various services on which the app depends – databases, message queues, and so on – in the diagram, these are represented by the symbols on the right-hand side.

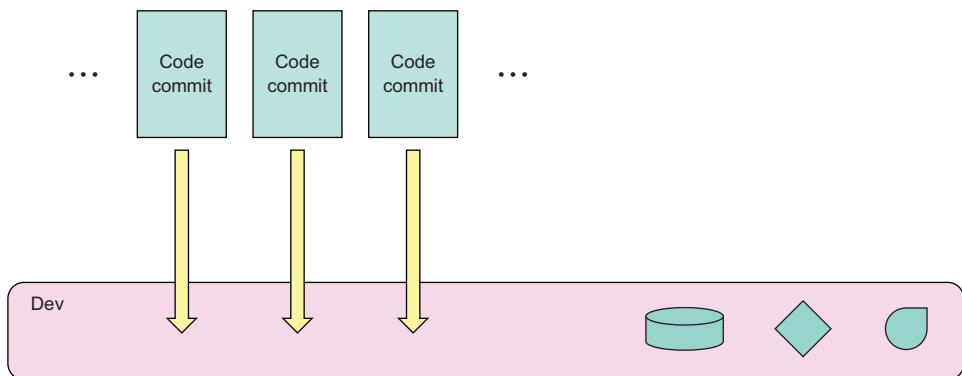


Figure 3.6 Code commits generate deployable artifacts which are deployed into a dev environment that looks similar to production but has development-versions of services such as databases and message queues (depicted with the symbols on the right).

Another less-frequent trigger, perhaps a time-based one that runs daily, will deploy the artifact into staging where a more comprehensive (and likely longer running) set of tests are executed in an environment which is a bit closer to production. You'll notice that in figure 3.7 the general shape of the staging environment is the same as that of the dev environment, but the two are shaded differently indicating some differences. For example, network topology in the dev environment might be flat with all apps being deployed into the same subnet, whereas in staging 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 in staging) but the difference in shading again signifies that they differ. For example, in staging, 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 they'd like to release, the artifact is tagged with a release version and deployed into production – figure 3.8. The production environment, including the service instances, differs from that of staging. For example, here the app is bound to the live customer database.

Although there are, as I've conceded, differences in the dev, staging, and production environments, I hinted at and want to emphasize that there are important similarities as well. For example, the API used to deploy into any of the environments is the same –

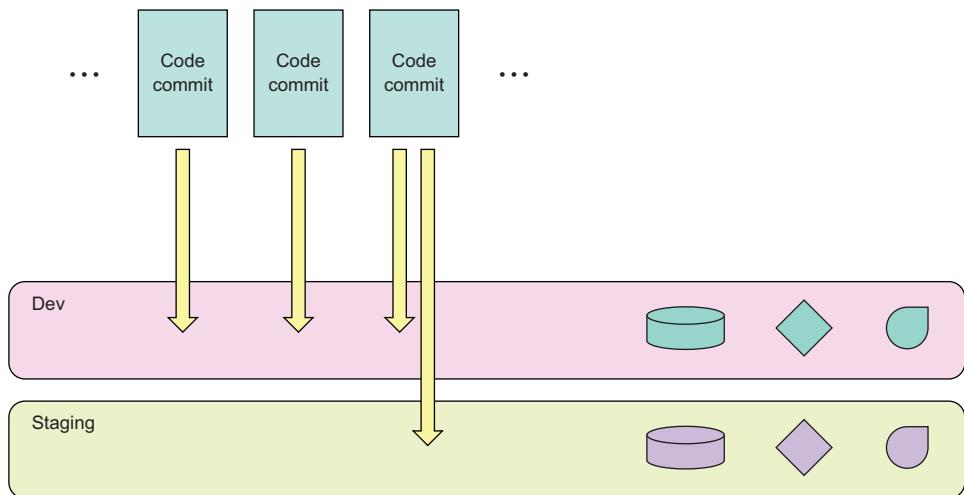


Figure 3.7 The same deployable artifact is deployed into a staging environment where it's bound to services (depicted with the symbols on the right) that more closely match those that exist in production.

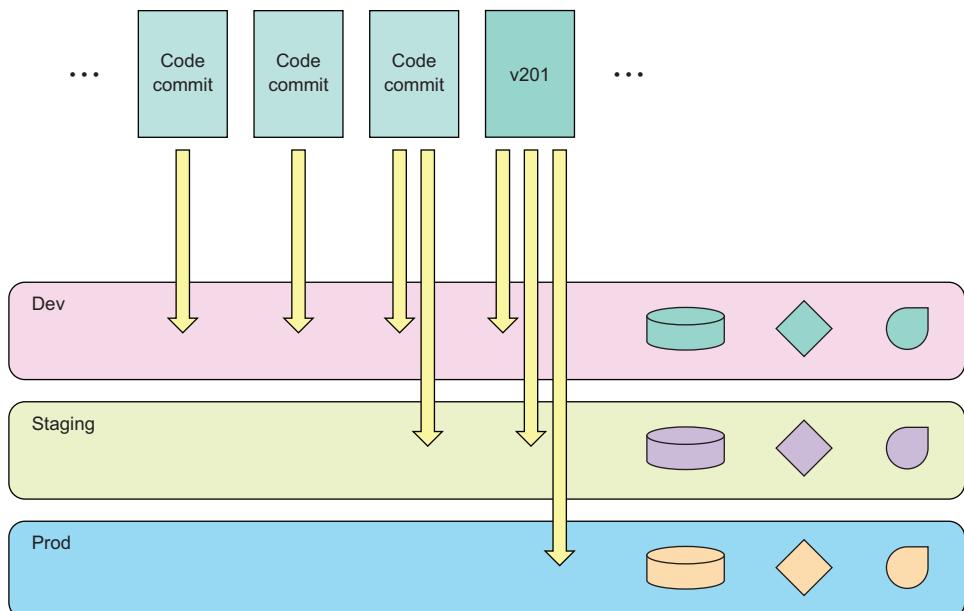


Figure 3.8 The same artifact is deployed into similar environments throughout the SDLC and must absorb the unavoidable differences across the environments.

managing the automation essential for a streamlined SDLC process with varying APIs would be an unnecessary burden. The base environment that includes things such as the operating system, language runtimes, specific I/O libraries, and more must be the same across all environments. 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 these. When the OS version in the dev environment is upgraded, it's only in lock-step with all other environments. Similarly, when any of the services are revved, a new version of RabbitMQ or Postgres is made available, it's simultaneously done in all environments.

But even more than ensuring the runtime environments match, a platform must also provide contracts that allow deployed apps to absorb the differences that exist. 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 3.9 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 is designed to support the entire SDLC.

On occasion, I've had clients implement a platform only for pre-production environments, or only for production. Although 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 only avail-

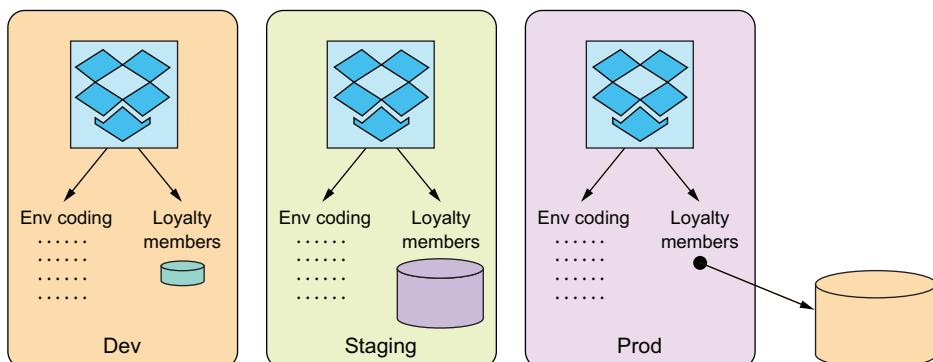


Figure 3.9 The platform must include a mechanism that allows the contract between app and runtime environment and bound services to serve the needs of the SDLC.

able in prod, 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 of the 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 **Enable Operational Excellence**

When, back in the early 2010s, I starting investigating this new technology category of Platform as a Service, or PaaS, I kept reading about the value it brought to the developer. The developer would benefit through self-service provisioning of the resources they needed. They'd no longer need to figure out what middleware components they'd build upon, source, and install. They'd no longer be constrained to particular programming frameworks because the company they worked for was “a java shop” or “a .Net shop”. PaaS was going to change all of that and make the developer's life sheer joy.

And then I joined the Cloud Foundry team at Pivotal and began working with large enterprise IT organizations and within a month a realized that PaaS was as much about the operational benefits as it was those for the developer. When a platform automatically detects crashed application instances and restarts them, or when it allows for an operating system vulnerability to be addressed within hours or days instead of weeks or months, the value clearly extends well beyond the process of creating applications. Although I can't cover all of the benefits of what we now call cloud-native platforms (instead of PaaS) in this short volume, I'd like to highlight a few.

3.4.1 **Repeatability**

Remember, change is the rule, not the exception. Application instances come and go. Environments those applications are running in are constantly disappearing and being recreated. Infrastructure topologies are incessantly metamorphosing. Yet in the face of these constant changes we need to maintain stability and order – we need the systems we're running to look the way we expect, ensuring every interaction we have with them is routine rather than a new invention.

Obviously, automation is foundational, and a fairly pervasive understanding of this drove the growth of the infrastructure automation market. Tools such as Chef, Puppet, Ansible, and more allowed the operations team to move from using a set of unmanaged scripts (at best) coupled with manual steps (at worst) to a streamlined operation. These tools provide a means for specifying infrastructure as code, sometimes through a DSL (Domain Specific Language), checking this code into source code control systems (like any other code), and executing that code against infrastructure resources (such as virtual machines).

But automation isn't for provisioning and configuring infrastructure resources; it also applies to full lifecycle management of the application. Code must be built into artifacts that are predictably deployed. When more application capacity is needed, the scaling

process must also be fully automated. New version of the app to be released? The deployment thereof and gradual cut-over from the old to the new is formalized with code.

Impossible to argue with any of this, but the question I'd pose is this: How much of this application-level automation must be done by the app team? When application capacity is scaled from five instances to ten, must the app team run a script that provisions five new application containers, deploys the code bits into them, starts the processes running, adds the new IP addresses to the router, and more?

The answer to these questions takes us back to the earlier discussion of infrastructure dial tone and application dial tone. A cloud-native platform projects abstractions that allow the app team (developers and operations) to not be responsible for this level of automation. They can provide their code and a declaration of the service level requirement, and the platform itself performs actions to satisfy that need. Automation is absolutely essential, but a cloud-native platform changes who is responsible for that automation.

Automation only gives us repeatability if the starting point of the automation is also predictable. Standardized machine images are those starting points and the control and management of those images must be a first-class concern of the platform. The platform team is responsible for maintaining a library of those images and upgrading them when needed (i.e. when something like Heartbleed¹ happens). And because the base environment, and even the automation applied on top of it, give us the context with which our mission critical and often sensitive applications run in, the platform team interfaces closely with several other IT functions that are responsible for maintaining the security, compliance, and stability postures of production IT systems.

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 – the dev wants to get his app running in production and these control functions require endless tickets to be filed and ultimately can stop a deploy from happening. On the other hand, if he sets his frustration aside for a moment, even the developer must admit the value that these organizations bring. We must protect our customer's personal data. We 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 access to the runtime environments has been too permissive. When an operator was able to ssh into a box to manually update an installed package, a vulnerability could sneak in. We had to guard a developer from specifying a dependence on a particular version of the JDK that had been known to cause performance degradations for certain types of workloads, and was therefore no longer permitted on production systems. We had to verify that appli-

¹ <https://en.wikipedia.org/wiki/Heartbleed>

cations would correctly communicate with compliance-centric, logging agents running in the environments. An explicit and often manual check that the rules were being followed was the only point of control.

Those points of control are implemented in various places across the application lifecycle and all too often are pushed 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 3.10, is that these controls apply to every deployment: every version, of every app. This means that the time from *ready to ship* to *deployed* is, at best, counted in days and multiple deployments in weeks.

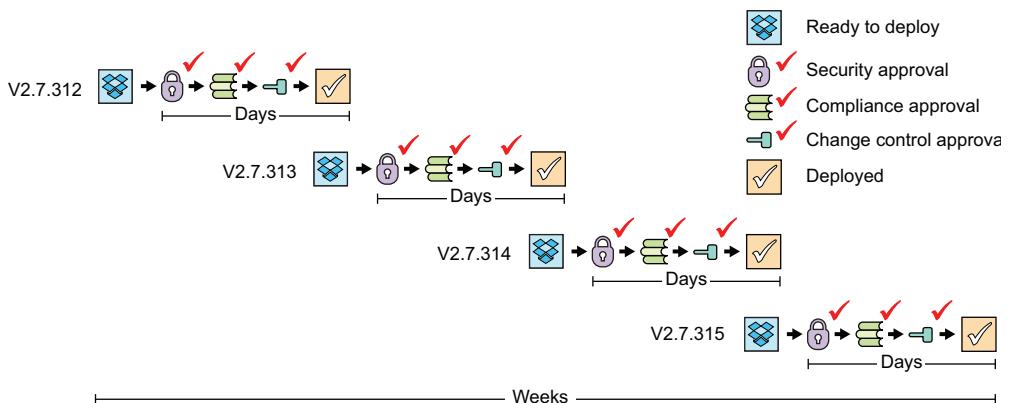


Figure 3.10 Control functions are on the critical path for every release of every app reduce the number of deployments that can be performed.

Remember when I talked about Amazon performing tens of thousands of deployments per day? They're doing something different. It isn't that they're exempt from regulatory requirements, nor are they cavalier with customer's personal data. The difference is that they're satisfying the requirements that the controls are designed for in a different manner. They've baked the solutions into their platform.

I've already talked about standardized machine images. Instead of allowing IT personnel to provision and configure virtual machines for a deployment and following that with an approval thereof, the parties responsible for ensuring the controls work with the platform team to construct and approve a single standardized image that all deployments then use. But then the use of those images must also be controlled.

NOTE It's **absolutely critical** that the platform only allow approved base images to be used and it restricts access to the runtime environments to keep their configuration in compliance.

Team autonomy's unquestionably the most critical factor in achieving the IT agility we now require. When we build a platform that addresses concerns that previously forged a dependence on not one, but several other IT functions, we shake the logjam loose. Because the control functions are implemented in the platform and deployments into the platform automatically satisfy the requirements of the control, the time from *ready to ship* to *deployed* takes minutes, and a series of deployments takes hours. (figure 3.11).

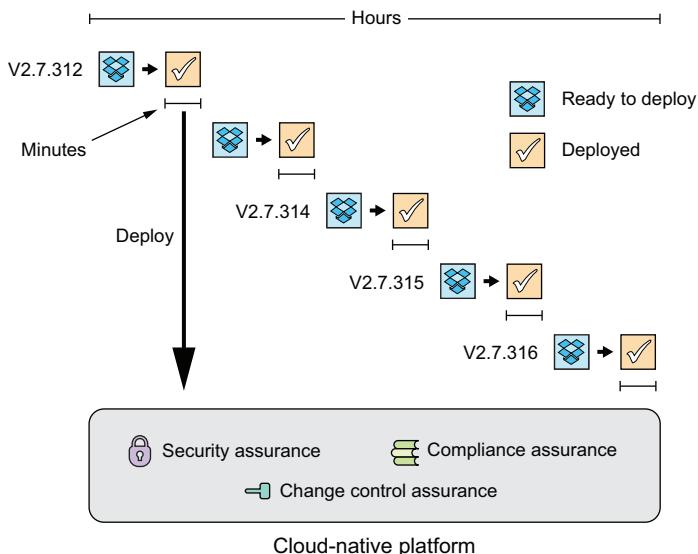


Figure 3.11 Deployments to a platform that implements controls allows for the time between having an app ready for deployment and performing that deploy is counted in minutes.

The app team is able to deploy whenever *they* feel it's necessary, and we've already studied the many positive effects of frequent deploys.

3.4.3 Autonomy

The narrative around microservice-based software architectures always lists team autonomy as one of the motivating factors. If the Amazon.com team that is responsible for the image service that generates thumbnails allows image zooming is independent from the one responsible for the recommendations engine (and many other teams), then most of the previous bottlenecks are eliminated. But independence of one app team from another app team is only part of the equation. We must also achieve independence between the app team and the platform team.

We've established that the platform team's responsible for the environments in which applications run. This includes the base operating system, language runtimes (i.e. an approved version of the JDK) and standardized services (i.e. Rabbit MQ, MySQL). This team is responsible for evolving that platform including upgrades needed to any of the components of the platform.

The app team is responsible for developing, deploying, and managing their apps. I've already talked about empowering the app team to do their own deploys, something achieved by giving controlled access to a equally controlled environment. It should be obvious that this type of self-service is needed from early on in the development process. When code is committed, the deployment into the dev environment requires no ticket be filed.

Let's then look at the question of app team and platform team independence through a concrete scenario. When there's a bug in an app (or a feature implementation, or any other change) it's up to the app team to fix it and deploy. When there's a bug in the platform, say an operating system vulnerability, it's up to the platform team to fix it and deploy. We want to do this with little to no coordination.

If we 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 is a given. With over one million active users¹, AWS couldn't manage its platform offering if it required coordination with that user base. But in the enterprise we continue to be burdened with such things – again, the difference between these two scenarios is the existence of the right platform. What we're aiming for is depicted in figure 3.12. I've let the dev environment fade away, it's still there but unimportant for making the point I'm aiming for.

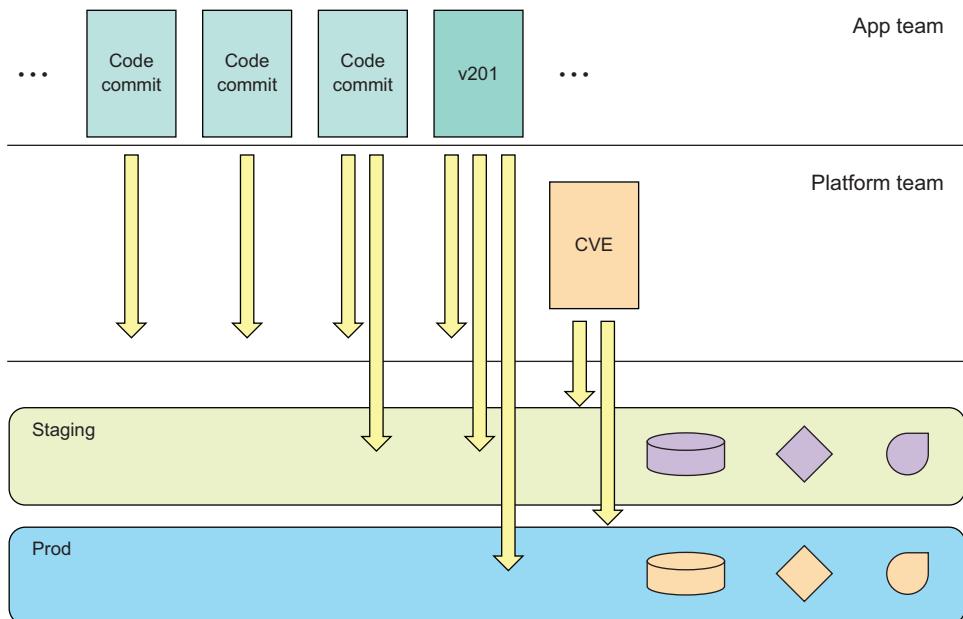


Figure 3.12 The right platform enables app teams to operate independently from platform teams.

¹ <https://techcrunch.com/2015/10/07/amazons-aws-is-now-a-7-3b-business-as-it-passes-1m-active-enterprise-customers/>

What remains are two different environments – staging and prod. At the top of the diagram I've established two swim lanes. The app team is executing the workflow that I talked about earlier, implementing features, fixing bugs, and checking code in. Pipelines drive further deployments into staging and eventually into production. Disrupting that flow slows release cadence. The platform team is responsible for addressing CVEs (Common Vulnerabilities and Exposures) in the base image (and the standard packages deployed onto it) and they must be able to do this without coordinating, or impacting the app team.

With the right platform, these two swim lanes needn't collide.

Now let's consider what happens when a CVE such as Heartbleed comes along. The platform team owns the base image that needs to be updated. What we'd like to enable is the ability for the platform team to do it. We aren't going to be reckless; we'll first deploy into the staging environment and run tests there and when they pass we move onto deployments in production. All of this is easy to draw on a diagram such as figure 3.12, but how can we do this safely?

Two of the most essential ingredients are 1) canary-style, rolling upgrades¹ and 2) a platform architecture that keeps the concerns of the app separate from the concerns of the platform, even within the container running the app. Let me start with the latter.

I haven't said much about it, but today it's a given that cloud-native workloads are running in containers. Docker is but one option for running containers, Kubernetes uses Docker within it, and Cloud Foundry embeds the same Linux container primitives in its implementation. When I talk about having an architecture that separates the concerns of the app team from the platform team, I'm talking about how the container image is structured. Figure 3.13 shows this structure.

Remember that the interface presented to the developer by the platform has app dial tone; the developer provides only their application code. The platform team, who are responsible for standardized OS images and runtime environments, provide everything

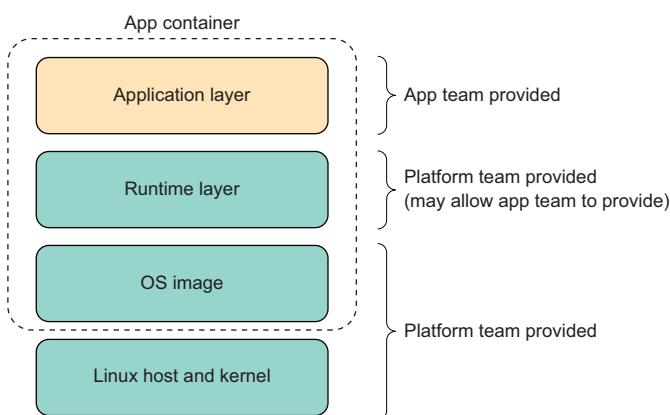


Figure 3.13 The structure of the container image clearly separates the concerns of the app team from the platform team.

¹ <https://martinfowler.com/bliki/CanaryRelease.html>

else. The cloud-native platform brings these two pieces together into the container image, which is then scheduled to run and otherwise managed on the platform itself.

Now when either the application, or the platform, provided parts of the container image are upgraded, a new container image is constructed and deployed and this is where canary-style, rolling upgrades come in. Cloud-native apps are always deployed with multiple instances. This provides a level of resilience in an ever-changing environment, and allows for exactly the incremental, zero-downtime deploys we're after. 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 some subset of the app instances running as others are being cycled. As a part of the safety net, the first replacements, called the canaries¹, can halt the deployment and revert the system to the previous state in the event that those initial deploys fail.

Whether the new platform deployment succeeds, or it's halted during a canary stage, the app team needn't even be aware that this is occurring. That autonomy is extraordinarily empowering and is an essential part of the cloud-native platform.

3.4.4 Reliability

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 the developer plays 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 availability zones (AZs), for example. To support reliability, Amazon provides their Elastic Compute Cloud (EC2) users with access to multiple AZs, giving them the option to deploy their apps into more than one to allow the app to survive an AZ failure. But when an AZ fails on AWS there are always users who lose their entire online presence. The exact reason surely varies, but in general, failing to deploy apps across AZs is due to the fact that it's non-trivial. 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 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 some other pattern. Need to move some 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 their EC2 service, it needn't be in the cloud-native platform. Instead, the platform team, understanding that multi-AZ platform configurations will help them meet the SLAs their user's demand, will configure the platform to use multiple AZs. An app team requests

¹ <https://martinfowler.com/bliki/CanaryRelease.html>

multiple instances of an app be deployed, say four as seen in figure 3.14, and the platform automatically distributes them evenly across all available zones. The platform implements all of the orchestration and management that folks would shoulder the burden for if they weren't using a cloud-native platform.

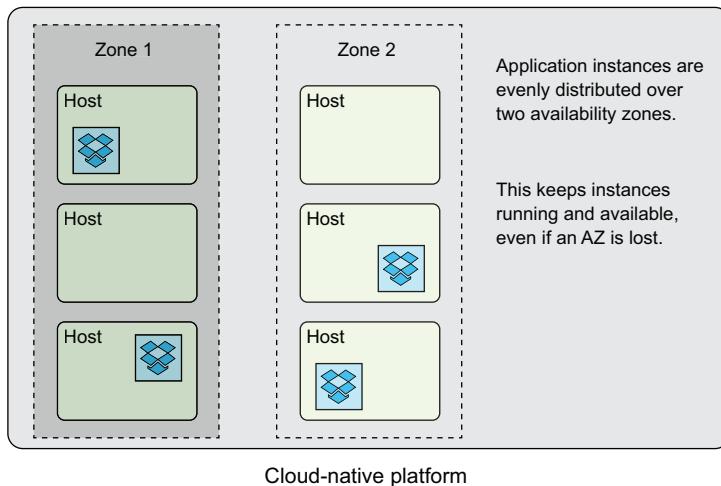


Figure 3.14 Management of workloads across availability zones is handled by the cloud-native platform.

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 executed in a manner that expects to never be done. Instead, the management of the system comes through constant monitoring of the actual (constantly changing) state of the system, comparison of it to a desired state, and remediating when necessary. This is a technique that 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, and although the implementation details differ slightly, the basic approaches are the same. Figure 3.15 depicts the key actors and the basic flow amongst them:

- 1 The user expresses the desired state by interacting with the API for the platform. For example, they 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 Hosts running workloads are each responsible for broadcasting the state of what is running on them into a fault-tolerant, distributed data store or messaging fabric.

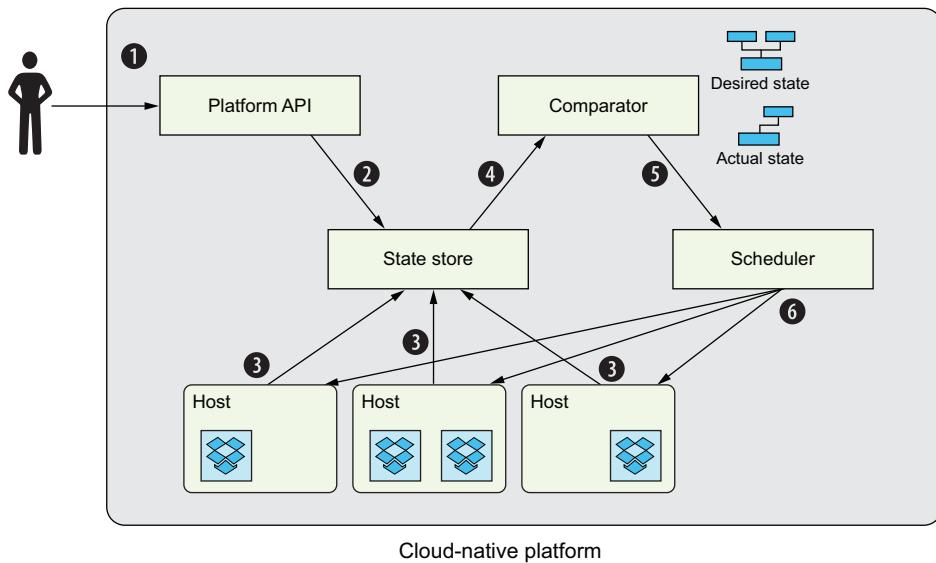


Figure 3.15 The state of applications running on the platform’s managed by continually comparing the desired state to the actual and executing corrective actions when necessary.

- 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 – quite 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, flapping state changes that are sometimes due to such flaky networks. 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). 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 auto-healing. The patterns I’ve described here for apps running on the platform are used for the management of the platform. The desired state may include one hundred hosts running application

workloads and a five-node distributed state store. If the system topology differs from that, then corrective actions will bring it back to the desired state.

What I've described throughout this section is quite sophisticated and goes well beyond the simple automation of steps that may have previously been performed manually. These are the capabilities that make up the cloud-native platform.

3.5 **Support Microservice-Based Software Architectures**

The 2017 State of DevOps Report¹ shows a clear correlation between high-performing IT teams and software architecture. When the architecture is made up of a set of loosely coupled components that can be independently developed, tested, and released, the IT processes runs far more efficiently and systems as a whole are more stable. Despite it being dangerously close to an overused buzzword (hence why I'm limiting its use throughout this text), "microservices" is a term that well describes the components and the architectural style.

The benefits of building and deploying microservices are clear, but their adoption doesn't come without some added burden. Instead of having a single monolith to manage – deploy, monitor, upgrade, and even reason about – we now have hundreds of different software components, each of which may have tens, hundreds or even thousands of instances deployed. Practices we've long used to manage traditional architectures don't work the same way and the tools of the past fall short.

The good news is that a new set of tools has emerged and when made available in a cloud-native platform, both developers and operations personnel can easily build the new muscle they need.

3.5.1 **Multi-Tenancy**

Often when considering multi-tenancy, we think of two different companies, the proverbial Coke and Pepsi, running in the same environment. We don't want these competitors to even know of the existence of each other, and our first thought turns to security – and it often stays there. Security is unquestioningly important, particularly if competing consumers are using your software – but there are other concerns.

First, let me refine the concept of a microservice – in fact, this leads us directly to the other primary concern. When our software is made up of a set of loosely coupled, independent components that are all running on the same platform, each one of these products, and I like to think of each microservice as an independent product, is a tenant of that platform. It may or may not be okay for the app team developing and operating their service to have some level of visibility to the other services running on the platform, but it's essential that one microservice not be able to starve other tenants of the resources they need to function. Fair resource sharing's a primary multi-tenancy concern.

VMware pioneered shared computing infrastructure right around the turn of the century. They surfaced the abstraction of a *virtual machine* (VM), the same entity that we interact with as a physical resource – a machine – and software-controlled doling

¹ <https://puppet.com/resources/whitepaper/state-of-devops-report>

out shares of the physical resources to multiple VMs. Arguably the main concern of flagship VMware products is shared resource utilization, and many, if not most, of the digital products running in large and small enterprises alike 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 we 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 a number of other attempts were previously made, container-based approaches, generally on Linux, have proven an outstanding solution. Based on the foundational concepts of control groups (Cgroups), which control the utilization of shared resources, and namespaces, which control the visibility of shared resources, Linux containers become the execution environments for the microservices collective that forms cloud-native software.

The first building blocks for containers have been around since the 1980s, but it wasn't until the late 2000s that the primitives that are heavily used today were readily available. But the low-level APIs for these primitives are something that the application developer and operator should be burdened with. The cloud-native platform allows the app team to provide the code they'd like to run, and the creation of containers and deployment of the app into those containers is entirely orchestrated by the platform. Containers are launched and disposed of in a tiny fraction of the time that VMs are, and because they share the host kernel, use far fewer resources than VMs do.

Container technology enables something powerful for cloud-native platforms; components that make up the platform, the API, the health management (eventual consistency engine), the logging and monitoring subsystem, the standardized images and runtime environments and much more, to be resources which are shared across tenants, all while ensuring that tenants are isolated from one another from a security and resource consumption perspective (figure 3.16). The cloud-native application platform is truly multi-tenant.

But software needs state as well, and that state is stored in services that are also available on the platform – things like relational and non-relational databases or messaging systems. The question is how to achieve multi-tenancy for those data services. Remember that multi-tenancy is about both security and fair sharing of resources, and as the shift in software architectures drove advancement in virtualization technologies in the compute tier, it also did this in the data tier. But truth be told, advancements here are lagging those in the compute tier a bit.

Many, if not most, of the stateful service products being used in the enterprise aren't built for multi-tenancy. Guides for these products often prescribe having individual instances, which is, individual deployments of the service per tenant. Even the newer, born-in-the-cloud DBs achieve multi-tenancy by launching individual database processes for each tenant. Few to no shared resources exist across tenants. The cloud-native platform can assist here, even as innovation in true database multi-tenancy progresses.

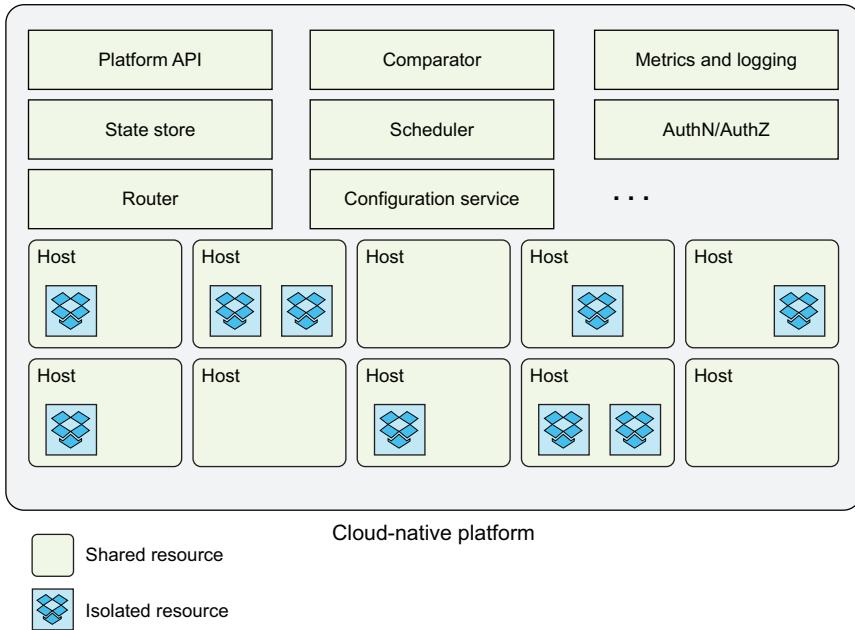


Figure 3.16 True multi-tenancy in the compute tier shares resources in the control plane, and also in the compute layer (the Linux host and kernel) while using containers to achieve resource isolation.

The platform API that the app team uses abstracts away unnecessary infrastructure details, and this is true for compute as well as for stateful services. They can request a database instance and they receive it (perhaps after a short wait). Whether this provisions new virtual machines, or new DB processes, or even creates a new schema in an already running DB process, that is all implemented within the platform. Figure 3.17 shows these options.

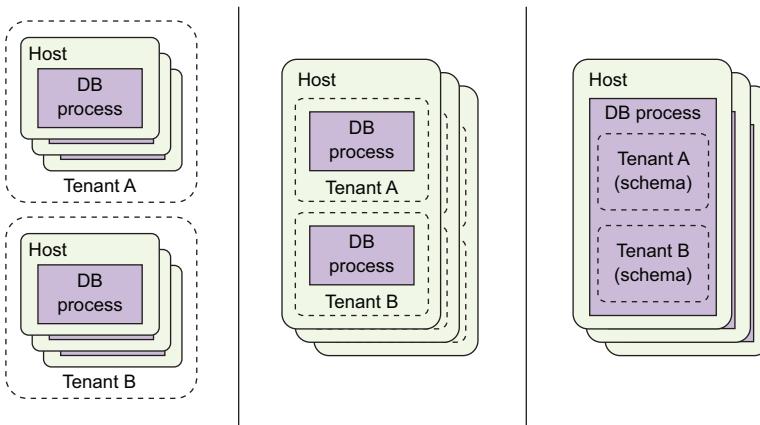


Figure 3.17 Database capacity served by the platform may implement a number of patterns from launching new machines, new processes on existing hosts or creating schema in existing DB processes (left to right).

Multi-tenancy is a concern for cloud-native software because each of the microservices that make up the whole of the digital solution must maintain autonomy. One microservice (i.e. the image service) can't be allowed to negatively impact another (i.e. the payment processing service). In general, a single microservice maps to a single tenant, and the cloud-native platform supports this paradigm.

3.5.2 **That Which Is Distributed Once Was Not**

With all of the talk about autonomy, team autonomy that empowers them to evolve and deploy their 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 to 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 intra-process, and 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, we need a way to make a configuration change to all of the instances without requiring a massive, collective reboot. When an execution flow passes through a dozen different microservices to fulfill a user request and it isn't performing well, we need to find where in the elaborate network of apps the problem lies. We need to keep retries, a foundational pattern in cloud-native software architectures where a client service repeats requests to a providing service when responses aren't forthcoming, from DDoSing our system as a whole. But remember, the developer isn't responsible for implementing all of the patterns required of cloud-native software – rather 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.

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 used to calculate the recommendations.

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, that IP address points to a router that then sends on the request to one of the instances of the *favorites service* (figure 3.18).

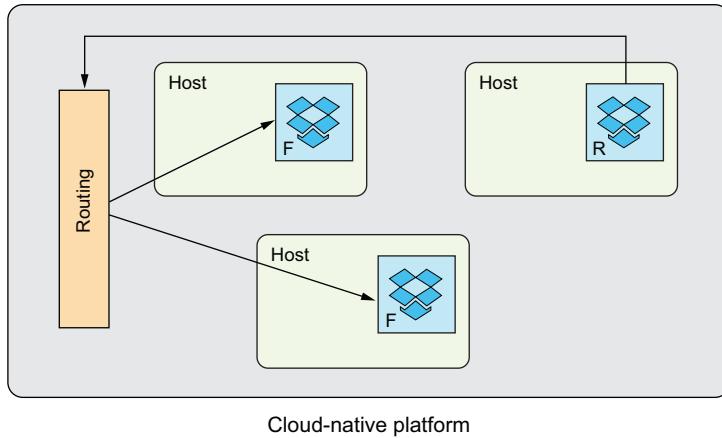


Figure 3.18 Recommendations Service finds the favorites service via DNS lookup and routing.

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 3.19 clearly depicts that this pulls the routing function into the calling service, thereby distributing the routing function itself.

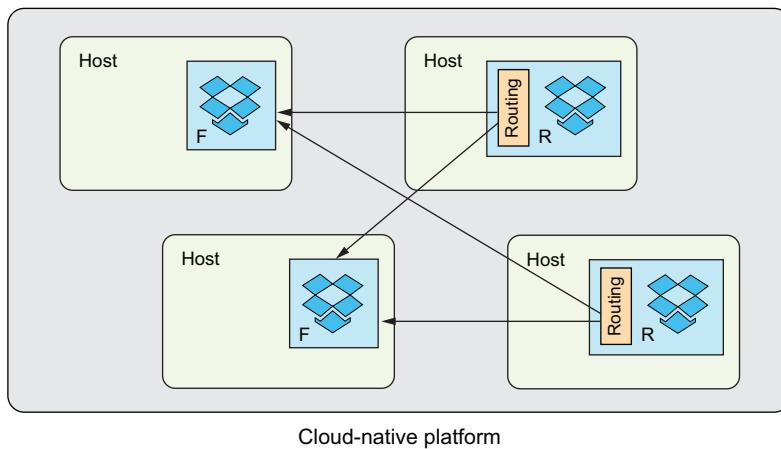


Figure 3.19 Recommendations Service directly access the favorites service via IP address; the routing function's distributed.

Whether the routing function is logically centralized (figure 3.18) or highly distributed (figure 3.19), keeping what are effectively routing tables up-to-date is an important process. In order to fully automate this process, the platform implements patterns such as collecting IP address information from newly launched or recovered microservice instances, and distribution of that data to the routing components, wherever they may be.

SERVICE CONFIGURATION

Our data scientists have done some 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 val-

ues. When the recommendation engine was deployed as a single process, we could go to that instance, supply a new configuration file, and restart the app. But now no (human) individual knows where all of 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 3.20.

- 1 Service instances know how to access a configuration service from which it obtains configuration values whenever necessary. Certainly, they'll do this at start-up time, but they must also do this when the configuration values are changed.
- 2 When configuration values are changed, the trick is to have each service instance refresh itself; the platform knows about all of the service instances – the actual state exists in the state store.
- 3 The platform notifies each of the service instances that new values are available and the instances take on those new values.

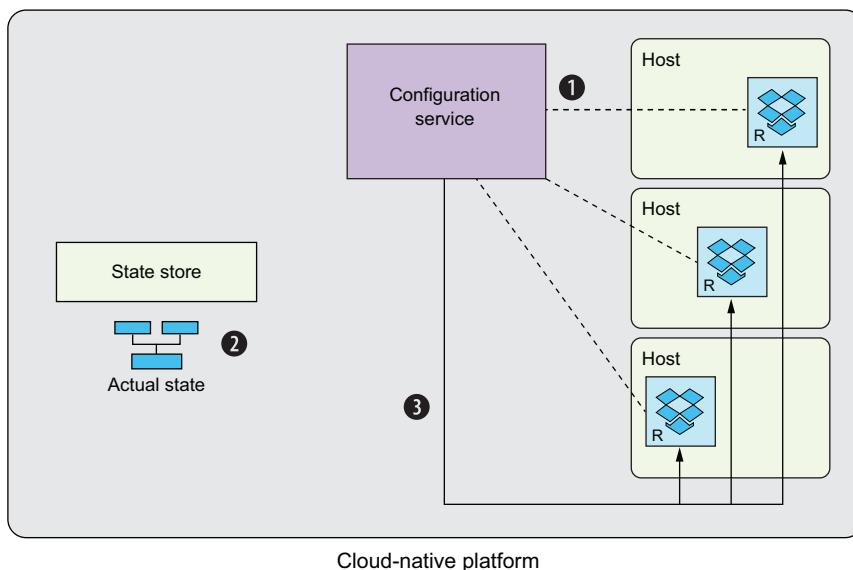


Figure 3.20 The configuration service of the cloud-native platform provides important configuration capabilities for microservice-based application deployments.

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, one designed specifically to provide the runtime

support needed for the modular and highly distributed nature of the cloud-native application. Other services include:

- Aggregated metrics and logging that draw these values together from the many instances of a particular service.
- A distributed tracing mechanism that allows us to diagnose issues 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'd 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.

3.6 **Summary**

In this chapter, you learned:

- A cloud-native platform takes on a great deal of the burden of satisfying the requirements on modern software.
- That the 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, as well as to the deployment of the platform itself.
- The different components that make up a piece of cloud-native software should be thought of as different products, and likely different tenants. The platform must provide support for multi-tenancy.
- As software becomes more modular and distributed, the services that bring the components together into a whole does 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.

index

Symbols

.Net shop 56
/proc/sys/fs/file-max file 25

A

abstractions
AWS and cloud services with higher level of 48
compute, storage and network 51
hardware resilience and 47
actual state 43
Adrian Cockcroft 11
aggregated metrics 71
agility 30
cloud-native software 20
frequent releases and driving business 30
shorter iterations 33
team autonomy 59
Amazon, extreme example of frequent releases 30
Amazon Web Services. *See* AWS
Amazon.com team 59
Ansible 56
app autonomy 68
app team 51–52
independence of 60
separation of concerns from platform team 61
application developers, cloud-native software and 4
application dial tone 61
cloud-native platform and 50
application lifecycle, points of control 58

application requirements 4–7
application.properties file 36
applications
access to log and metric data for 49
and running within availability zones (AZs) 3
as entities operators are interested in 50
crashed, automatically detected by platform 56
deployment 3
full lifecycle management of 56
key requirements for modern 5, 21
lifecycle of 13
parallel deployments of multiple versions 40
refactoring 20
stem cells 38
testing two apps in sequence 27
architectural style, migrations to new 20
artifacts
code and predictably deployed 56
deployable 52–53
and property files 25
single deployable 38
variability across SDLC and difference in 25
versioned 40
assembly line, environment control and 35
Auth API, example of cloud-native software in use 15
automation 56–57
availability zones. *See* AZs
availability, modern applications and 5
AWS (Amazon Web Services) 18
and availability zones 62
and redundant systems 2
cloud platforms and 47
downtime experience 1

perspective 60
 service partition 2
 zero downtime 5
 AZs (availability zones) 2, 19
 and stability of software running on AWS 47
 failure in deploying apps across 62
 multi-AZ platform configurations 62
 Azure
 perspective 60

B

bank teller application, cloud-native software and example of 19

C

capacity, cloud-native apps and 13
 CD (continuous delivery) 30, 45, 52–56
 and unanticipated implementation challenges 32
 defined 30
 example 31
 iterative process and gathering feedback 32
 more flexible and palatable options 33
 shorter iterations 33–34
 Change Control, control function 57
 change
 as the rule 42
 different approach in treating 4
 Chaos Kong exercises, Netflix and experience of AWS outage 2
 Chef 56
 Chief Security Office, control function 57
 circuit breakers 71
 Cloud Foundry 56
 container running 61
 eventual consistency 63
 platform 46
 cloud platform providers, perspective 60
 cloud provider, responsibility for system functioning 1
 cloud, set of primitives offered by 18
 cloud-based environments, dynamic nature of 13
 cloud-native
 approaches, requirements of strong consistency and 18
 clarification of the definition 18
 composition 12, 14
 data 12–14
 dial tone 49–52
 patterns, added burden in using 20

platform 46
 and management of workloads across AZs 62
 and the entity in the focus of the operator 50
 approved base images 58
 automation level 57
 container technology 66
 continuous delivery 52
 entire SDLC support 52–56
 evolution of 47–48
 responsibilities of platform and app teams 52
 true multi-tenancy of 66
 vs. IaaS platform 51
 software 4
 and platforms designed to support 47
 and providing functionality 47
 automatic retries 14
 complexity of 11
 defined 9
 described 7–9
 dual role of components 10
 in use 14–17
 key elements 9–11
 model for 11–14
 multi-tenancy 68
 nuances of 16
 cloud-native applications 11–13
 and releases into production 23
 easy and frequent releases 29
 stability and predictability 29
 obstacles in production operations 23–29
 things to consider when designing and building 42
 cloud-nateness, as an option 21
 code, ability to frequently release 5
 Common Vulnerabilities and Exposures. *See* CVEs
 comparator 64
 Compliance, control function 57
 components, connections between, and the impact of constant change 11
 composition, managing, and cloud-native software 13–14
 compute, storage and network primitives 48
 configuration management database (CMDB) 49
 configuration, cloud-native apps and 13
 connected devices 6
 consistency, cloud-native patterns and 19
 containers, cloud-native workloads 61
 continuous delivery. *See* CD
 control groups (Cgroups), Linux containers 66
 CVEs (Common Vulnerabilities and Exposures) 61

D

data
 fabric, distributed 14
 handling in cloud setting 13
 holistic level of approach to 7
 management, cohesive 17
 models 13
decomposition, cloud-native data and 12
deployable artifact control 36–37
design patterns, cloud-native software and 13
desired state, example of 43
dev environment 53
dev team, and adding new capabilities to
 implementation 31
developers
 and availability of good base image to 38
 and implementation of stability feature 62
 and making available and responsive
 software 22
 and well-written, completed software 23
 involvement in initial deployments 28
 motivation to care about operations 22
development schedule, delays 33
digital experience, ubiquitousness of 4
digital solutions
 autonomy in executing responsibilities 52
 developers and proper functioning of 49
 properly deployed applications and 3
discipline, achieving repeatability and 35
distributed data coordination 17
distributed systems, complexity of 64
distributed tracing mechanism 71
Docker, container running and 61
Domain Specific Language. *See* DSL
done state 42
downtime, unplanned, impact of 5
DSL (Domain Specific Language) 56

E

Elastic Compute Cloud (EC2) 47
 Amazon and reliability support 62
Elastic Compute platform (EC2), availability in
 other countries 2
enablers
 and key SDLC changes 29
 CD (continuous delivery) 30–34
 repeatability 34–38
 safe deployments 39–42
env config abstraction 55
environment control 35

environment parity 55
environments
 accessing logs in app-centric 50
 accessing logs in infrastructure-centric 49
 and artifact deployment 52
 bottleneck in software deployment 28
 differences in dev, staging and production 53
 expectation of constant change 28, 30
 pre-production 55
 example of performing test in 27
 responsibility for fragile 23
 responsibility of platform team 59
 staging and production 61
 standardized machine images 35
 variability across SDLC and difference in 25
eventual consistency 42–43, 63–64, 71
 and change as the rule 42

F

failure, different approach in treating 4
favorites service 68–69
file descriptors, difference in environment
 and 25
frequent releases 40
 cloud-native software and 7
 essentiality of iterative cycles to 35
funds, dispensing without access to the source of
 record 19

G

Google App Engine (GAE) 48
Google Cloud Platform 18
 perspective 60

H

Hearbleed. *See also* CVEs
Heartbleed 57
home automation market 6

I

infrastructure
 resources, efficient use of 9
 virtualized 18
infrastructure as code 35
infrastructure dial tone
 IaaS platforms and 50
 platform team and 51

infrastructure-centricity vs. application centricity 50
 accessing an app and 14
 cloud native apps and 62
Internet of Things. *See IoT*
Internet, devices connected to the 6
 internet-accessible data centers, cloud platform and 47
 internet-native companies, alterations in software architecture 4
IoT (Internet of Things) 6

J

jar file 36
Java Development Kit. *See JDK*
 java shop 56
JDK (Java Development Kit) 35

K

Kubernetes, eventual consistency and 63

L

localized databases, application requirements and 7
 loose coupling, cloud-native software and 12, 16
 loyalty members abstraction 55

M

Medium Trust Code 48
 microservices 8, 19
 adoption 65
Microsoft Azure 18
 mobile and multi-device support 5
 modular design, software and 8

N

Netflix
 AWS outage and 2, 4
 moving into cloud 19
 network latency 32
 non-distributed systems 18

O

obstacles, production operations and 23–29
 online banking software, cloud-native software in use 15

operational excellence
 automation 56
 predictable starting point 57
 autonomy
 app/platform team independence 61
 container image 62
 CVEs (Common Vulnerabilities and Exposures) 61
 team autonomy 59
 control functions 57
 frequent deployments 59
 points of control 58
 enabling 56
 reliability
 AZs (availability zones) 62
 eventual consistency 63
 repeatability 56
 operational instability, snowflakes and 25
 outages
 expensive, and software release. *See also* risky deployments
 responding to incident 42

P

PaaS (platform as a service), operational benefits 56
 parallel deployment 40
 routing as key enabler of 41
 patterns
 cloud-native 46
 cloud-native app and implementing the right 11
Paxos, protocol 64
 personally identifiable information (PII) 53
Pivotal Cloud Foundry 38
 planned release milestone, continuous delivery and 33
 platform
 application quality 47
 as complex distributed system 64
Platform as a Service. *See PaaS*
 platform team 51–52
 independence of 60
 separation of concerns from app team 61
 standardized machine images 57
Pokemon Go 20
 practices, cloud-native 46
 pre-cloud infrastructures, cloud platform and 47
 process control 37–38
 product vision 20

production

- differences in environments 36
- product instability 29
- safe experimentation in 39–40
- safety net during 40–41
- self-healing system in software building 44
- shorter development cycles 45
- software deployment as challenging process 22
- production operations, efficient, safe and repeatable 37
- property file 36
- providers, cloud-platform 48
- Puppet 56

R

- Raft, protocol 64
- ready to ship to deployed, time between 58–59
- recommendation algorithm 69
- recommendations service 68–69
- redundancy 2
 - AZs and providing further 2
 - cloud-native patterns, applying 20
 - cloud-native software and redundant entity copies 11
- relational database 10
- release cycles, shortened, major impediment to 7
- release date, shifting 33
- repeatability 45
 - benefits of 35
 - compromised 37
 - deployable artifact control 36–37
 - environment control 35
 - overview 34
 - process control 37–38
- See also* operational excellence
- resilience
 - abstractions and hardware 47
 - cloud-native software and 13, 16, 20
- resource sharing, fair, multi-tenancy and 65
- risk, reduction and continuous releases of code 5
- risky deployments
 - and less frequent releases 28
 - as one of the leading causes of system breakage 29
 - test environment 26
 - test failure 27
 - two applications example 27
 - unexpected downtime 26
- routing 40–41
- run book 29

runtime environments 36, 55

- access to 57
- repeatability and 45
- restricted access 58
- standardized 38

S**safe deployments**

- and generation of necessary telemetry 40
- experiment results assessment 40
- overview 39
- parallel deployment 40
- routing 40–41
- safety net for 39
- smaller units of deployment 41
- safe experimentation, safe deployments and 39
- scheduler 64
- SDLC (software delivery lifecycle)
 - traditional 31
 - variability across 24
- SDLC (Software Development Lifecycle) 23
 - continuous delivery 52
 - streamlined 55
- security, multi-tenancy 66
- separation of concerns 51
- service configuration 69–71
- service discovery 68–69
- service instances 70
- service level, earlier decades and providing correct 49
- serviceness 12
- shared databases 7
- shipping, as business decision 31
- shortened feedback cycles 5
- Simple Queuing Service (SQS) 47
- Simple Storage Service (S3) 47
- single region outage, example 3
- SLAs
 - and the knowledge needed for 4
 - service providers 1
- snowflakes 24
 - changes in infrastructure topology 26
 - contribution to slowness and instability 30
 - differences in environment 25
- UAT (User Acceptance Test) 25
- software
 - adaptable 9
 - and resilience to infrastructure failures 8
 - greater risk of failures 8
 - requirements 8

responsiveness 7
user's expectations and 4
well-written and completed 23

software architecture
app autonomy 68
basic, familiar elements of 10
container technology 66
microservice-based 59
multi-tenancy
 security 65
 stateful service 66
retries 68
State of DevOps Report (2017) 65
team autonomy 68

software deployment
and creating smaller units of deployment 41
differences in packages software depends on 25
example of desired state 43
factors contributing to difficulty of 24
inherently risky 44
obstacles 23
 and expectation of infrastructure stability 24, 28
 production instability 24
 risky deployments 24, 26
 snowflakes 23–24
production operations 23
time periods without new 28

software designs, prediction for the next decade 17

software development
connecting practices and architectural patterns 23
methodology, CD and 32

software development lifecycle. *See* SDLC

software releases
making deployments less risky and 33
reducing risks associated with 26–28

software systems, monolithic 5

source code control systems 56
ssh access, running environments 35
stability, cloud-native applications and 21
standardized base images, assembly of 38
standardized machine images 57–58
 environment control 35
State of DevOps Report (2017) 65
stateful services 66
system outage 1

T

team autonomy 68
telemetry 40
test deployment, running in exact replica of production 26
test environment 26–27

U

UAT (User Acceptance Test) 25
unexpected changes, reaction to 42
User Acceptance Test. *See* UAT
User API 15
User Profile app 15

V

variability, detrimental effect of. *See* snowflakes
virtual machine. *See* VM
Virtualized Infrastructure 51
VM (virtual machine) 65
 containers 66
VMware, shared computing infrastructure and 65

Z

zero downtime 5

Want to learn more?

Special offer from manning.com

Save 50% on the full version of *Cloud Native* by Cornelia Davis
with discount code **davisdv** at

<https://www.manning.com/books/cloudnative>

