**Chapter 1. You keep using that word: Defining “cloud-native”**

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

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

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

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

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

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

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

*Netflix did experience a brief availability blip in the affected Region, but we sidestepped any significant impact because Chaos Kong exercises prepare us for incidents like this. By running experiments on a regular basis that simulate a Regional outage, we were able to identify any systemic weaknesses early and fix them. When US-EAST-1 became unavailable, our system was already strong enough to handle a traffic failover.****[***[***1***](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn1)***]***

***1***

*See “Chaos Engineering Upgraded” at the Netflix Technology blog (*[*http://mng.bz/P8rn*](http://mng.bz/P8rn)*) for more information on Chaos Kong.*

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

|  |
| --- |
|  |

**Note**

How was Netflix able to recover so quickly? Redundancy.

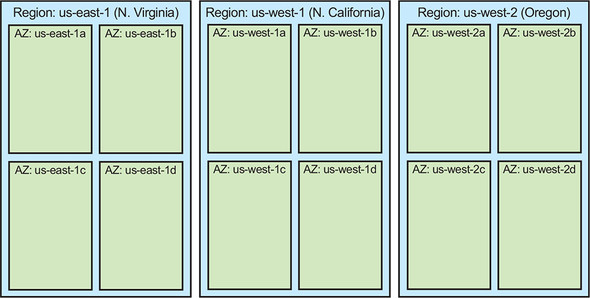
|  |
| --- |
|  |

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

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

[Figure 1.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig01) depicts three regions, each of which contains four availability zones.

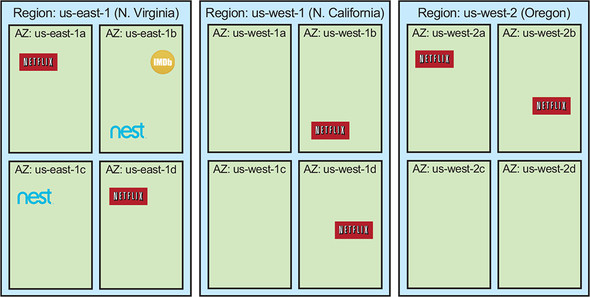
**Figure 1.1. AWS partitions the services it offers into regions and availability zones. Regions map to geographic areas, and AZs provide further redundancy and isolation within a single region.**



Applications run within availability zones and—here’s the important part—may run in more than one AZ and in more than one region. Recall that a moment ago I made the assertion that redundancy is one of the keys to uptime.

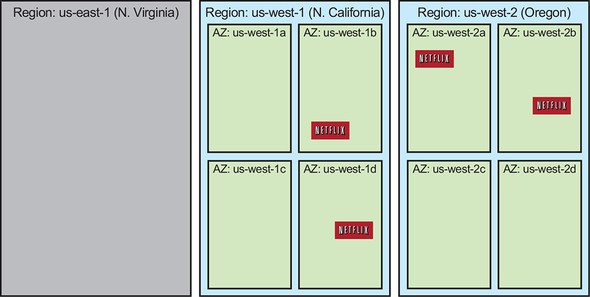
In [figure 1.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig02), let’s place logos within this diagram to hypothetically represent running applications. (I have no explicit knowledge of how Netflix, IMDb, or Nest have deployed their applications; this is purely hypothetical, but illustrative nevertheless.)

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



[Figure 1.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig03) depicts a single-region outage, like 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 of an entire region.**



In this simple graphic, you can immediately see how Netflix might have weathered the outage far better than others companies; it already had its applications running in other AWS regions and was able to easily direct all traffic over to the healthy instances. And though it appears that the failover to the other regions wasn’t automatic, Netflix had anticipated (even practiced!) a possible outage such as this and had architected its software and designed its operational practices to compensate.**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn2)**]**

***2***

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

|  |
| --- |
|  |

**Note**

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

|  |
| --- |
|  |

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

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

The difference between this and what Netflix does today starts with a fundamental difference in philosophy. With the former approaches, change or failure is treated as an exception. By contrast, Netflix and many other large-scale internet-native companies, such as Google, Twitter, Facebook, and Uber, *treat change or failure as the rule*. These organizations have altered their software architectures and their engineering practices to make designing for failure an integral part of the way they build, deliver, and manage software.

|  |
| --- |
|  |

**Note**

Failure is the rule, not the exception.

|  |
| --- |
|  |

**1.1. Today’s application requirements**

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

**1.1.1. Zero downtime**

The AWS outage of September 20, 2015, demonstrates one of the key requirements of the modern application: it must always be available. Gone are the days when even short maintenance windows during which applications are unavailable are tolerated. The world is always online. And although unplanned downtime has never been desirable, its impact has reached astounding levels. For example, in 2013 *Forbes* estimated that Amazon lost almost $2 million during a 13-minute unplanned outage.**[**[**3**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn3)**]** Downtime, planned or not, results in significant revenue loss and customer dissatisfaction.

***3***

*See “Amazon.com Goes Down, Loses $66,240 Per Minute” by Kelly Clay at the Forbes website for more details (*[*http://mng.bz/wEgP*](http://mng.bz/wEgP)*).*

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

**1.1.2. Shortened feedback cycles**

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

From the moment that you have an idea for a feature, you’re taking on some level of risk. Is the idea a good one? Will customers be able to use it? Can it be implemented in a better-performing way? As much as you try to predict the possible outcomes, reality is often different from what you can anticipate. The best way to get answers to important questions such as these is to release an early version of a feature and get feedback. Using that feedback, you can then make adjustments or even change course entirely. Frequent software releases shorten feedback loops and reduce risk.

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

**1.1.3. Mobile and multidevice support**

In April 2015, Comscore, a leading technology-trend measurement and analytics company, released a report indicating that for the first time, internet usage via mobile devices eclipsed that of desktop computers.**[**[**4**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn4)**]** Today’s applications need to support at least two mobile device platforms, iOS and Android, as well as the desktop (which still claims a significant portion of the usage).

***4***

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

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

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

**1.1.4. Connected devices—also known as the Internet of Things**

The internet is no longer only for connecting humans to systems that are housed in and served from data centers. Today, billions of devices are connected to the internet, allowing them to be monitored and even controlled by other connected entities. The home-automation market alone, which represents a tiny portion of the connected devices that make up the Internet of Things (IoT), is estimated to be a $53 billion market by 2022.**[**[**5**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn5)**]**

***5***

*You can read more about these findings by Zion Market Research at the GlobeNewswire site (*[*http://mng.bz/mm6a*](http://mng.bz/mm6a)*).*

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. Other connected devices include automobiles, home appliances, farming equipment, jet engines, and the supercomputer most of us carry around in our pockets (the smartphone).

Internet-connected devices change the nature of the software we build in two fundamental ways. First, the volume of data flowing over the internet is dramatically increased. Billions of devices broadcast data many times a minute, or even many times a second.**[**[**6**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn6)**]** Second, in order to capture and process these massive quantities of data, the computing substrate must be significantly different from those of the past. It becomes more highly distributed with computing resources placed at the “edge,” closer to where the connected device lies. This difference in data volume and infrastructure architecture necessitates new software designs and practices.

***6***

*Gartner forecasts that 8.4 billion connected things will be in use worldwide in 2017; see the Gartner report at*[*www.gartner.com/newsroom/id/3598917*](http://www.gartner.com/newsroom/id/3598917)*.*

**1.1.5. Data-driven**

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

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

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

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

These *user* requirements—constant availability, constant evolution with frequent releases, easily scalable, and intelligent—can’t be met with the software design and management systems of the past. But what characterizes the software that can meet these requirements?

**1.2. Introducing cloud-native software**

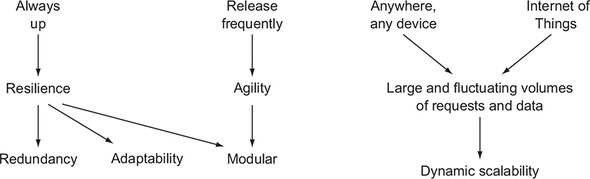
Your software needs to be up, 24/7. You need to be able to release frequently to give your users the instant gratification they seek. The mobility and always-connected state of your users drives a need for your software to be responsive to larger and more fluctuating request volumes than ever before.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

**1.2.1. Defining “cloud-native”**

What characterizes *cloud-native software*? Let’s analyze the preceding requirements a bit further and see where they lead. [Figure 1.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig04) takes the first few steps, listing requirements across the top and showing causal relationships going downward. The following list explains the details:

* Software that’s always up must be resilient to infrastructure failures and changes, whether planned or unplanned. As the context within which it runs experiences those inevitable changes, software must be able to adapt. When properly constructed, deployed, and managed, composition of independent pieces can limit the blast radius of any failures that do occur; this drives you to a modular design. And because you know that no single entity can be guaranteed to never fail, you include redundancy throughout the design.
* Your goal is to release frequently, and monolithic software doesn’t allow this; too many interdependent pieces require time-consuming and complex coordination. In recent years, it’s been soundly proven that software made up of smaller, more loosely coupled and independently deployable and releasable components (often called *microservices*) enables a more agile release model.
* No longer are users limited to accessing digital solutions when they sit in front of their computers. They demand access from the mobile devices they carry with them 24/7. And nonhuman entities, such as sensors and device controllers, are similarly always connected. Both of these scenarios result in a tidal wave of request and data volumes that can fluctuate wildly, and therefore require software that scales dynamically and continues to function adequately.

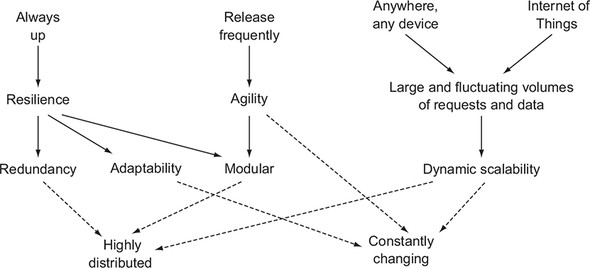
**Figure 1.4. User requirements for software drive development toward cloud-native architectural and management tenets.**



Some of these attributes have architectural implications: the resultant software is composed of redundantly deployed, independent components. Other attributes address the management practices used to deliver the digital solutions: a deployment must 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](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig05):

* Software that’s constructed as a set of independent components, redundantly deployed, implies distribution. If your redundant copies were all deployed close to one another, you’d be at greater risk of local failures having far-reaching consequences. To make efficient use of the infrastructure resources you have, when you deploy additional instances of an app to serve increasing request volumes, you must be able to place them across a wide swath of your available infrastructure—even, perhaps, that from cloud services such as AWS, Google Cloud Platform (GCP), and Microsoft Azure. As a result, you deploy your software modules in a highly distributed manner.
* Adaptable software is by definition “able to adjust to new conditions,” and the conditions I refer to here are those of the infrastructure and the set of interrelated software modules. They’re intrinsically tied together: as the infrastructure changes, the software changes, and vice versa. Frequent releases mean frequent change, and adapting to fluctuating request volumes through scaling operations represents a constant adjustment. It’s clear that your software and the environment it runs in are constantly changing.

**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.**



|  |
| --- |
|  |

**Definition**

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

|  |
| --- |
|  |

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

**1.2.2. A mental model for cloud-native software**

Adrian Cockcroft, who was chief architect at Netflix and is now VP of Cloud Architecture Strategy at AWS, talks about the complexity of operating a car: as a driver, you must control the car and navigate streets, all while making sure not to come into contact with other drivers performing the same complex tasks.**[**[**7**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn7)**]** You’re able to do this only because you’ve formed a model that allows you to understand the world and control your instrument (in this case, a car) in an ever-changing environment.

***7***

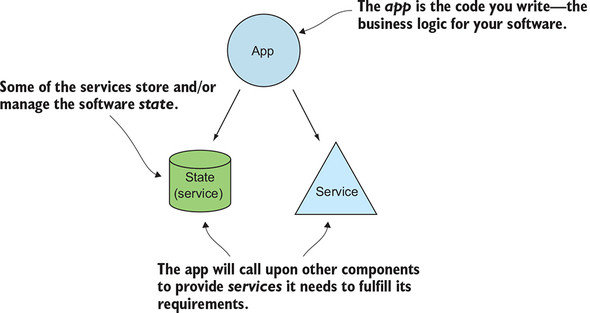
*Hear Adrian talk about this and other examples of complicated things at*[*http://mng.bz/5NzO*](http://mng.bz/5NzO)*.*

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

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

I’ll start simple, with core elements of cloud-native software that are surely familiar to you, shown in [figure 1.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig06).

**Figure 1.6. Familiar elements of a basic software architecture**

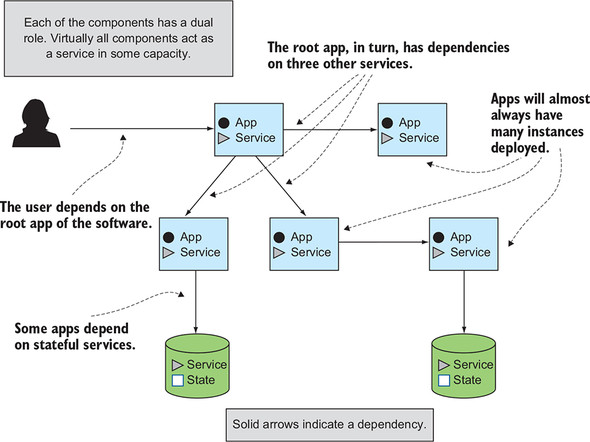


An *app* implements key business logic. This is where you’ll be writing the bulk of the code. This is where, for example, your code will take a customer order, verify that items are available in a warehouse’s inventory, and send a notification to the billing department.

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

Taking these simple concepts, let’s now build up a topology that represents the cloud-native software you’ll build; see [figure 1.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig07). You have a distributed set of modules, most of which have multiple instances deployed. You can see that most of the apps are also acting as services, and further, that some services are explicitly stateful. Arrows depict where one component depends on another.

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



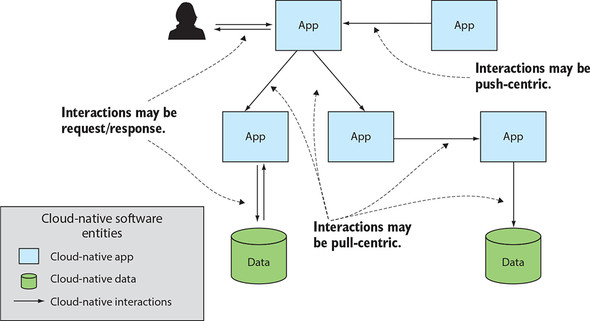
This diagram illustrates a few interesting points. First, notice that the pieces (the boxes and the database, or storage, icons) are always annotated with two designations: apps and services for the boxes, and services and state for the storage icons. I’ve come to think of the simple concepts shown in [figure 1.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig07) as roles that various components in your software solution take on.

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

And this brings me to defining the three parts of cloud-native software, depicted in [figure 1.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig08):

* ***The cloud-native app—***Again, this is where you’ll write code; it’s the business logic for your software. Implementing the right patterns here allows those apps to act as good citizens in the composition that makes up your software; a single app is rarely a complete digital solution. An app is at one or the other end of an arrow (or both) and therefore must implement certain behaviors to make it participate in that relationship. It must also be constructed in a manner that allows for cloud-native operational practices such as scaling and upgrades to be performed.
* ***Cloud-native data—***This is where state lives in your cloud-native software. Even this simple picture shows a marked deviation from the architectures of the past, which often used a centralized database to store state for a large portion of the software. For example, you might have stored user profiles, account details, reviews, order history, payment information, and more, all in the same database. Cloud-native software breaks the code into many smaller modules (the apps), and the database is similarly decomposed and distributed.
* ***Cloud-native interactions—***Cloud-native software is then a composition of cloud-native apps and cloud-native data, and the way those entities interact with one another ultimately determines the functioning and the qualities of the digital solution. Because of the extreme distribution and constant change that characterizes our systems, these interactions have in many cases significantly evolved from those of previous software architectures, and some interaction patterns are entirely new.

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



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

With this model established, let’s come back to the modern software requirements covered in [section 1.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01lev1sec1) and consider their implications on the apps, data, and interactions of your cloud-native software.

**Cloud-native apps**

Concerns about cloud-native apps include the following:

* Their capacity is scaled up or down by adding or removing instances. We refer to this as *scale-out/in*, and it’s far different from the scale-up models used in prior architectures. When deployed correctly, having multiple instances of an app also offers levels of resilience in an unstable environment.
* As soon as you have multiple instances of an app, and even when only a single instance is being disrupted in some way, keeping state out of the apps allows you to perform recovery actions most easily. You can simply create a new instance of an app and connect it back to any stateful services it depends on.
* Configuration of the cloud-native app poses unique challenges when many instances are deployed and the environments in which they’re running are constantly changing. If you have 100 instances of an app, for example, gone are the days when you could drop a new config into a known filesystem location and restart the app. Add to that the fact that these instances could be moving all over your distributed topology. And applying such old-school practices to the instances as they are moving all over your distributed topology would be sheer madness.
* The dynamic nature of cloud-based environments necessitates changes to the way you manage the application lifecycle (not the software *delivery* lifecycle, but rather the startup and shutdown of the actual app). You must reexamine how you start, configure, reconfigure, and shut down apps in this new context.

**Cloud-native data**

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

* You need to break apart the data monolith. In the last several decades, organizations invested a great deal of time, energy, and technology into managing large, consolidated data models. The reasoning was that concepts that were relevant in many domains, and hence implemented in many software systems, were best treated centrally as a single entity. For example, in a hospital, the concept of a patient was relevant in many settings, including clinical/care, billing, experience surveys, and more, and developers would create a single model, and often a single database, for handling patient information. This approach doesn’t work in the context of modern software; it’s slow to evolve and brittle, and ultimately robs the seemingly loosely coupled app fabric of its agility and robustness. You need to create a distributed data fabric, as you created a distributed app fabric.
* The distributed data fabric is made up of independent, fit-for-purpose databases (supporting polyglot persistence), as well as some that may be acting only as materialized views of data, where the source of truth lies elsewhere. Caching is a key pattern and technology in cloud-native software.
* When you have entities that exist in multiple databases, such as the “patient” I mentioned previously, you have to address how to keep the information that’s common across the different instances in sync.
* Ultimately, treating state as an outcome of a series of events forms the core of the distributed data fabric. Event-sourcing patterns capture state-change events, and the unified log collects these state-change events and makes them available to members of this data distribution.

**Cloud-native interactions**

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

* Accessing an app when it has multiple instances requires some type of routing system. Synchronous request/response, as well as asynchronous event-driven patterns, must be addressed.
* In a highly distributed, constantly changing environment, you must account for access attempts that fail. Automatic retries are an essential pattern in cloud-native software, yet their use can wreak havoc on a system if not governed properly. Circuit breakers are essential when automated retries are in place.
* Because cloud-native software is a composite, a single user request is served through invocation of a multitude of related services. Properly managing cloud-native software to ensure a good user experience is a task of managing a composition—each of the services and the interactions between them. Application metrics and logging, things we’ve been producing for decades, must be specialized for this new setting.
* One of the greatest advantages of a modular system is the ability to more easily evolve parts of it independently. But because those independent pieces ultimately come together into a greater whole, the protocols underlying the interactions among them must be suitable for the cloud-native context; for example, a routing system that supports parallel deploys.

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

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

**1.2.3. Cloud-native software in action**

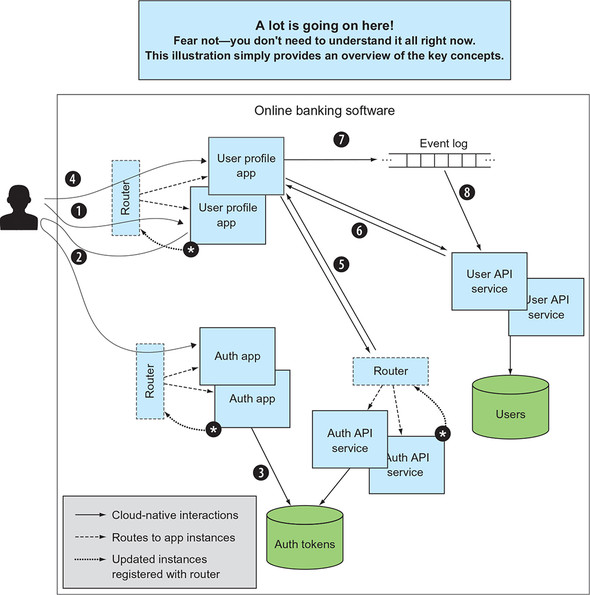
Let’s start with a familiar scenario. You have an account with Wizard’s Bank. Part of the time you engage with the bank by visiting the local branch (if you’re a millennial, just pretend with me for a moment ;-)). You’re also a registered user of the bank’s online banking application. After receiving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

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

Let’s see what this could look like if that online banking application were architected in a cloud-native manner. [Figure 1.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig09) depicts these key elements:

* Because you aren’t yet logged in, when you access the *User Profile* app, ***1*** it will redirect you to the *Authentication* app. ***2*** Notice that each of these apps has multiple instances deployed and that the user requests are sent to one of the instances by a router.
* As a part of logging in, the Auth app will create and store a new *auth token* in a stateful service. ***3***
* The user is then redirected back to the User Profile app, with the new auth token. This time, the router will send the user request to a different instance of the User Profile app. ***4*** (Spoiler alert: sticky sessions are bad in cloud-native software!)
* The User Profile app will validate the auth token by making a call to an *Auth API service.* ***5*** Again, there are multiple instances, and the request is sent to one of them by the router. Recall that valid tokens are stored in the stateful Auth Token service, which is accessible from not only the Auth app, but also any instances of the Auth API service.
* Because the instances of any of these apps (the User Profile or Auth apps) can change for any number of reasons, a protocol must exist for continuously updating the router with new IP addresses.***\****
* The User Profile app then makes a downstream request to the *User API service* ***6*** to obtain the current user’s profile data, including phone number. The User Profile app, in turn, makes a request to the user’s stateful service.
* After the user has updated their phone number and clicked the Submit button, the User Profile app sends the new data to an *event log* ***7***.
* Eventually, one of the instances of the User API service will pick up and process this change event ***8*** and send a write request to the Users database.

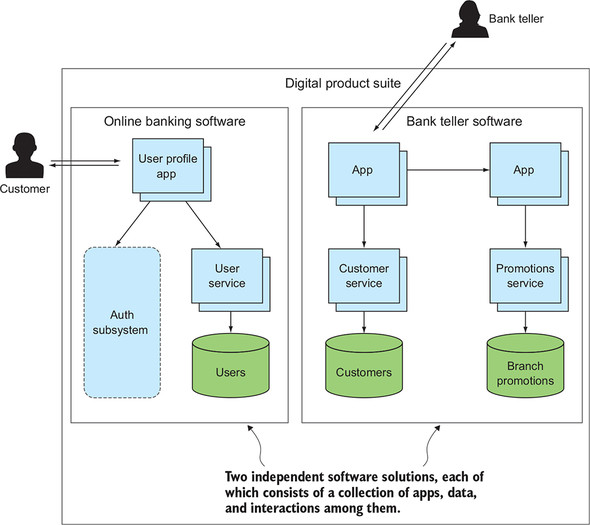
**Figure 1.9. The online banking software is a composition of apps and data services. Many types of interaction protocols are in play.**



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

I haven’t explicitly stated it, but when you’re back at the bank branch and the teller verifies your current contact information, you’ll expect the teller to have your updated phone number. But the online banking software and the teller’s software are two different systems. This is by design; it serves agility, resilience, and many of the other requirements that I’ve identified as important for modern digital systems. [Figure 1.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig10) shows this product suite.

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

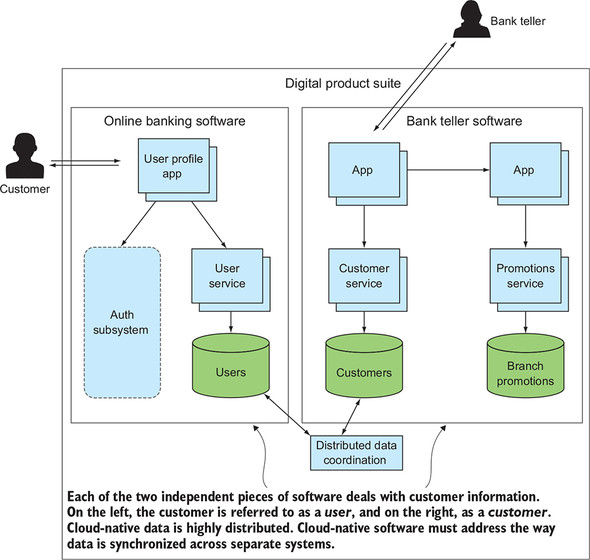


The structure of the bank teller software isn’t markedly different from that of the online banking software; it’s a composition of cloud-native apps and data. But, as you can imagine, each digital solution deals with and even stores user data, or shall I say, *customer* data. In cloud-native software,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

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

In [figure 1.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig11), I’ve added one more concept to our model—something I’ve labeled “Distributed data coordination.” The depiction here doesn’t imply any implementation specifics. I’m not suggesting a normalized data model, hub-and-spoke master data management techniques, or any other solution. For the time being, please accept this as a problem statement; I promise we’ll study solutions soon.

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



That’s a lot! [Figures 1.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig09), [1.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig10), and [1.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig11) are busy, and I don’t expect you to understand in any great detail all that’s going on. What I do hope you take from this comes back to the key theme for cloud-native software:

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

We’ll get into all the details, and more, throughout the following chapters.

**1.3. Cloud-native and world peace**

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

Cloud-native software architectures, often referred to as microservices,**[**[**8**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn8)**]** are all the rage today—but spoiler alert, they also won’t lead to world peace. And even if they did come to dominate (and I believe they will), they don’t apply to everything. Let’s look at this in more detail in a moment, but first, let’s talk about that word, *cloud*.

***8***

*Although I use the term microservice to refer to the cloud-native architecture, I don’t feel that the term encompasses the other two equally important entities of cloud-native software: data and interactions.*

**1.3.1. Cloud and cloud-native**

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

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

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

|  |
| --- |
|  |

**Note**

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

|  |
| --- |
|  |

If cloud-native is about how, does that mean you can implement cloud-native solutions on premises? You bet! Most of the enterprises I work with on their cloud-native journey first do so in their own data centers. This means that their on-premise computing infrastructure needs to support cloud-native software and practices. I talk about this infrastructure in [chapter 3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03).

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

**1.3.2. What isn’t cloud-native**

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

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

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

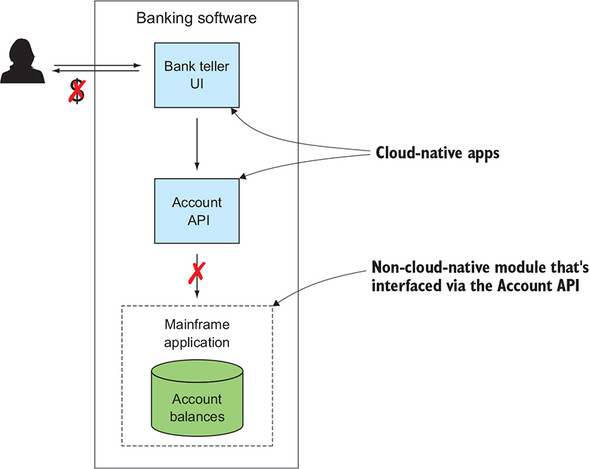
And, finally, sometimes you have existing software that isn’t cloud-native, and there’s no immediate value in rewriting it. Most organizations that are more than a couple of decades old have parts of their IT portfolio running on a mainframe, and believe it or not, they may keep running that mainframe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

**1.3.3. Cloud-native plays nice**

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

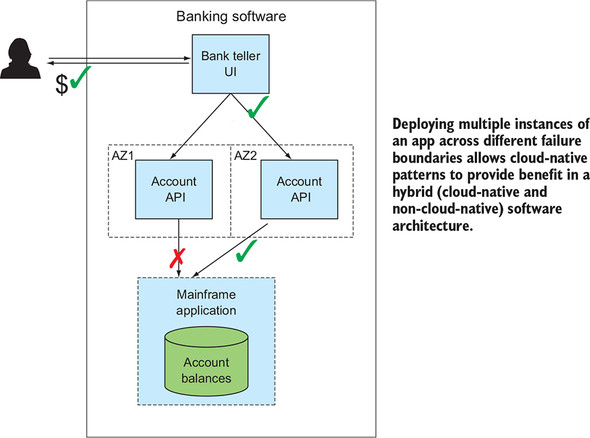
Applying cloud-native patterns where you can, even while other parts of your software employ older design approaches, can bring immediate value. In [figure 1.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig12), for example, you see that we have a few application components. A bank teller accesses account information via a user interface, which then interfaces with an API that fronts a mainframe application. With this simple deployment topology, if the network between that Account API service and the mainframe application is disrupted, the customer will be unable to receive their cash.

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



But now let’s apply a few cloud-native patterns to parts of this system. For example, if you deploy many instances of each microservice across numerous availability zones, a network partition in one zone still allows access to mainframe data through service instances deployed in other zones ([figure 1.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fig13)).

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



It’s also worth noting that when you do have legacy code that you wish to refactor, it needn’t be done in one fell swoop. Netflix, for example, refactored its entire customer-facing digital solution to a cloud-native architecture as a part of its move to the cloud. Eventually. The move took seven years, but Netflix began refactoring *some* parts of its monolithic, client-server architecture in the process, with immediate benefits.**[**[**9**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_012.html#ch01fn9)**]** As with the preceding banking example, the lesson is that even during a migration, a partially cloud-native solution is valuable.

***9***

*For more details, see “Completing the Netflix Cloud Migration” by Yury Izrailevsky (*[*http://mng.bz/6j0e*](http://mng.bz/6j0e)*).*

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

I believe that this architectural style will be the dominant one for a decade or two to come. What distinguishes it from other fads with less staying power is that it came as a result of a foundational shift in the computing substrate. The client-server models that dominated the last 20 to 30 years first emerged when the computing infrastructure moved from the mainframe to one where many smaller computers became available, and we wrote software to take advantage of that computing environment. Cloud-native has similarly emerged as a new computing substrate—one offering *software-defined* compute, storage, and networking abstractions that are highly distributed and constantly changing.

**Summary**

* Cloud-native applications can remain stable, even when the infrastructure they’re running on is constantly changing or even experiencing difficulties.
* The key requirements for modern applications call for enabling rapid iteration and frequent releases, zero downtime, and a massive increase in the volume and variety of the devices connected to it.
* A model for the cloud-native application has three key entities:
  + The cloud-native app
  + Cloud-native data
  + Cloud-native interactions
* *Cloud* is about where software runs; *cloud-native* is about how it runs.
* Cloud-nativeness isn’t all or nothing. Some of the software running in your organization may follow many cloud-native architectural patterns, other software will live on with its older architecture, and still others will be hybrids (a combination of new and old approaches).