**Chapter 3. The platform for cloud-native software**

*This chapter covers*

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

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

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

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

|  |
| --- |
|  |

**Note**

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

|  |
| --- |
|  |

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

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

**3.1. The cloud(-native) platform evolution**

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

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

**3.1.1. It started with the cloud**

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

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

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

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

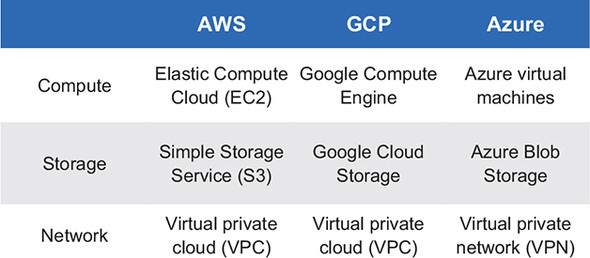
AWS effectively created a new market, and it took competitors, such as Google and Microsoft, two years to have any response. When they did, each came with unique offerings.

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

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

Google and Microsoft both eventually provided services that exposed infrastructure abstractions, as shown in [figure 3.1](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig01), and, in reverse, AWS began offering cloud services with higher-level abstractions.

**Figure 3.1. Infrastructure-as-a-service (IaaS) offerings from major cloud platform providers**



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

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

**3.1.2. Cloud-native dial tone**

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

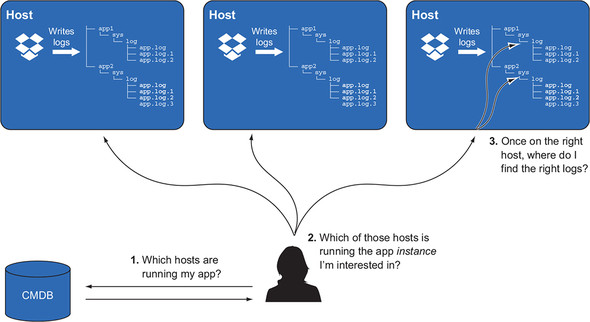
As hinted at in the cloud platform evolution you just read about, this is changing. To clearly understand the difference, let’s look at a concrete example. Say you have an application deployed. To make sure that it’s running well, or to diagnose when things go wrong, you must have access to log and metric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

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

[Figure 3.2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig02) depicts the steps:

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

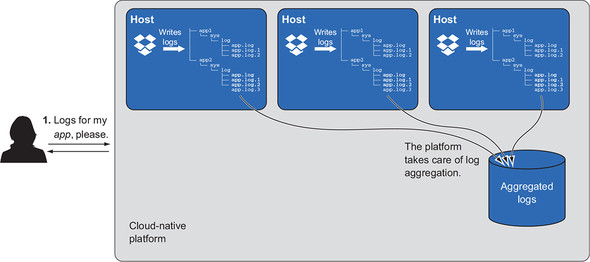
**Figure 3.2. Accessing application logs in an infrastructure-centric environment is tedious.**



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

By contrast, [figure 3.3](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig03) shows the operator experience when the apps are running on a cloud-native platform. It’s extremely simple: you ask for the logs for your application. You’re making app-centric requests.

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



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

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

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

|  |
| --- |
|  |

**Definition**

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

|  |
| --- |
|  |

|  |
| --- |
|  |

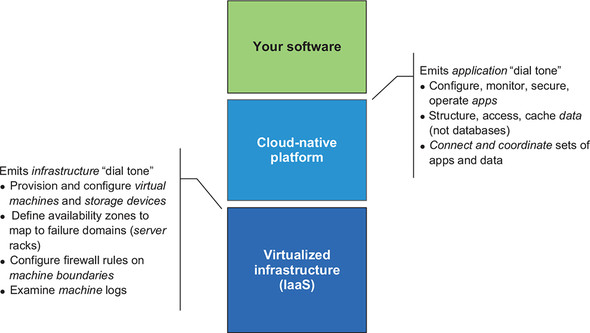
**Definition**

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

|  |
| --- |
|  |

You’ve surely seen the blocks that are stacked in [figure 3.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig04), clearly separating the three layers that ultimately come together to provide a digital solution to consumers. Virtualized infrastructure enables easier consumption of compute, storage, and network abstractions, leaving the management of the underlying hardware to the IaaS provider. The cloud-native platform brings the level of abstraction up even further, allowing a consumer to consume OS and middleware resources more easily, and leaving the management of the underlying compute, storage, and network to the infrastructure provider.

**Figure 3.4. The cloud-native platform abstracts infrastructure concerns, allowing teams to focus on their applications instead of these lower-level concerns.**



The annotations on either side of the stack in [figure 3.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig04) suggest differences in the operations performed against these abstractions. Instead of deploying an app onto one or more hosts via IaaS interfaces, on the cloud-native platform an operator deploys an application, and the platform takes care of distributing the requested instances against available resources. Instead of configuring the firewall rules to secure the boundary of the hosts that are running a particular application, the operator 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 app, the operator accesses the logs for the app. The experiential differences that a cloud-native platform offers over an IaaS platform are significant.

What I’ll talk about in this chapter, and what I’m encouraging you to build your cloud-native software to, is a cloud-native platform—the one that emits application dial tone. Several of these platforms are available today. From the big cloud providers, we have Google App Engine, AWS Elastic Beanstalk, and Azure App Service (none of which are particularly widely adopted). Cloud Foundry is an open source cloud-native platform that has had remarkable penetration into large enterprises globally. Several vendors have commercial offerings (Pivotal, IBM, and SAP, to name only a few).**[**[**1**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fn1)**]** Although the details of these platforms vary, all have a common philosophical foundation and provide an application dial tone.

***1***

*Full disclosure: at the time of this book’s publication, I work for Pivotal on its Cloud Foundry, Kubernetes, and other emerging platforms.*

**3.2. Core tenets of the cloud-native platform**

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

**3.2.1. First, let’s talk about containers**

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

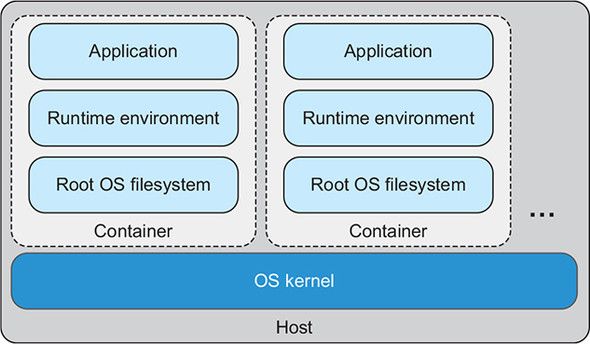
If, when I use the term “container” you immediately think “Docker,” that’s cool—close enough. But I do want to cover key elements of containers in the abstract so that you can more easily connect those capabilities to the elements of cloud-native software.

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

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

[Figure 3.5](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig05) shows the portions of the application and runtime environment that are running both on the host and inside your containers. Only the OS *kernel* is provided by the host. Inside the container you first have the OS root filesystem, including operating system functions such as openssh or apt get. The runtime needed by your application is also inside the container—the Java Runtime Environment (JRE) or the .NET Framework, for example. And then, finally, your application is also in the container, hopefully running there.

**Figure 3.5. A host usually has multiple containers running on it. These containers share the OS kernel from the host. But each container has its own root operating filesystem, runtime environment, and application code.**

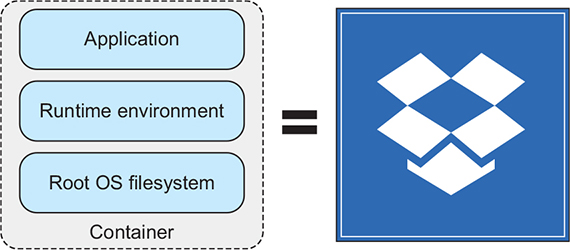


When an application instance is to be run, a container is created on a host. All the bits necessary to run your app—the OS filesystem, application runtime, and the application itself—will be installed into that container and the appropriate processes started. The cloud-native platform, using containers at the core, provides a whole lot of functionality for your software, and creation of an app instance is but one. Others include the following:

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

The key points I want to you remember about the container as you begin to study what a cloud-native platform brings to bear is that (1) your infrastructure will have multiple hosts, (2) a host has multiple containers running on it, and (3) your app uses the OS and runtime environment installed into the container for its functionality. In many of the diagrams that follow, I depict the container with the icon shown in [figure 3.6](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig06).

**Figure 3.6. When studying the capabilities of a cloud-native platform, at times we’ll think of the container as a black box within which your application runs. A bit later, we’ll drill into the details of what’s running in the container.**



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

**3.2.2. Support for “constantly changing”**

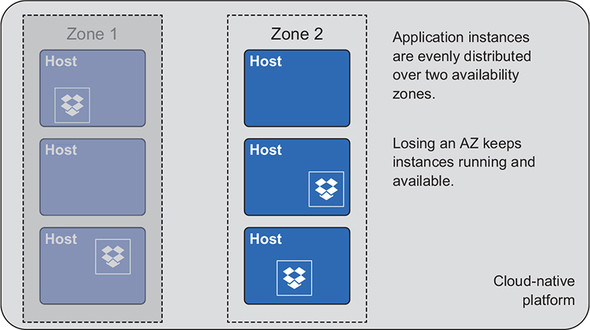
I started this book with the story of an Amazon outage that demonstrated how an application can remain stable even as the platform it’s running on is experiencing trouble. Although developers play a crucial role in achieving that resilience through the design of their software, they needn’t be responsible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

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

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

Although the AZ is an abstraction that AWS exposes to the user of its EC2 service, it needn’t be exposed to the user of the cloud-native platform. Instead, the platform can be configured to use multiple AZs, and all that orchestration of application instances across those AZs is then handled by the platform. An app team simply requests multiple instances of an app be deployed (say, four, as shown in [figure 3.7](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig07)), and the platform automatically distributes them evenly across all available zones. The platform implements all of the orchestration and management that humans would otherwise shoulder the burden for if they weren’t using a cloud-native platform. Then when change happens (an AZ goes down, for example), the application continues to work.

**Figure 3.7. Management of workloads across availability zones is handled by the cloud-native platform.**

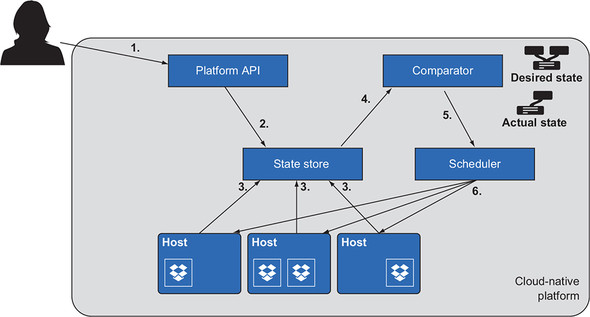


Another concept that I’ve previously mentioned is *eventual consistency*, a key pattern in the cloud, where things are constantly changing. Deployments and management tasks, which we know must be automated, are designed with the expectation that they are never done. Instead, the management of the system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

Several cloud-native platforms implement this basic pattern, including Kubernetes and Cloud Foundry. Although the implementation details differ slightly, the basic approaches are the same. [Figure 3.8](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig08) depicts the key actors and the basic flow among them:

1. The user expresses the desired state by interacting with the API for the platform. For example, the user may ask that four instances of a particular app be running.
2. The platform API continually broadcasts changes to the desired state into a fault-tolerant, distributed data store or messaging fabric.
3. Each host running workloads is responsible for broadcasting the state of what is running on them into a fault-tolerant, distributed data store or messaging fabric.
4. An actor, which I’m calling the *comparator* here, ingests information from the state store, maintains a model of both the desired state and the actual state, and compares the two.
5. If the desired and actual states don’t match, the comparator informs another component in the system of the difference.
6. This component, which I’m calling the *scheduler*, determines where new workloads should be created or which workloads should be shut down, and communicates with the hosts to make this happen.

**Figure 3.8. The state of applications running on the platform is managed by continually comparing the desired state to the actual state and then executing corrective actions when necessary.**



The complexity lies in the distributed nature of the system. Frankly, distributed systems are hard. The algorithms implemented in the platform must account for lost messages from the API or hosts, network partitions that may be brief but disrupt the flow nonetheless, and flapping state changes that are sometimes due to such flaky 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). Just as application teams needn’t concern themselves with the complexity of managing workloads across AZs, they also needn’t be burdened with implementation of eventually consistent systems; that capability is baked into the platform.

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

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

**3.2.3. Support for “highly distributed”**

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

The good news is that, as an industry, we’ve been working on solutions to these new problems for some time, and the patterns are fairly well established. When one component needs to communicate with another, it needs to know where to find that other component. When an app is horizontally scaled to hundreds of instances, you need a way to make a configuration change to all instances without requiring a massive, collective reboot. When an execution flow passes through a dozen microservices to fulfill a user request and it isn’t performing well, you need to find where in the elaborate network of apps the problem lies. You need to keep retries—a foundational pattern in cloud-native software architectures in which a client service repeats requests to a providing service when responses aren’t forthcoming—from DDoS**[**[**2**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fn2)**]**-ing your system as a whole.

***2***

*A distributed denial of service (DDoS) (*[*http://mng.bz/4OGR*](http://mng.bz/4OGR)*) isn’t always intentional or with malicious intent.*

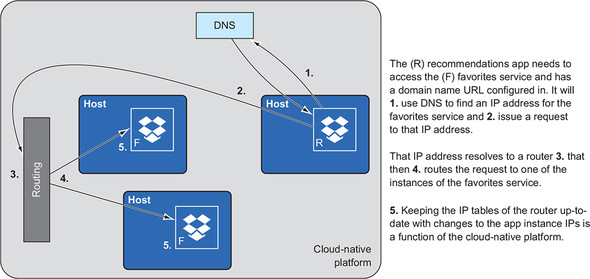
But remember, the developer isn’t responsible for implementing all the patterns required of cloud-native software; instead, the platform can give the assist. Let’s take a brief look at some of the capabilities offered by cloud-native platforms in this regard.

I want to use a concrete example to illustrate a handful of patterns; I’ll use a recipe-sharing site. One of the services it provides is a list of recommended recipes, and in order to do this, the *recommendations service* calls a *favorites service* to obtain the list of recipes that the user previously starred. These favorites are then used to calculate the recommendations. You have several apps, each with multiple instances deployed, and the functionality of those apps and the interaction among them determines the behavior of your software. You have a distributed system. What are some of the things that a platform might provide to support this distributed system?

**Service discovery**

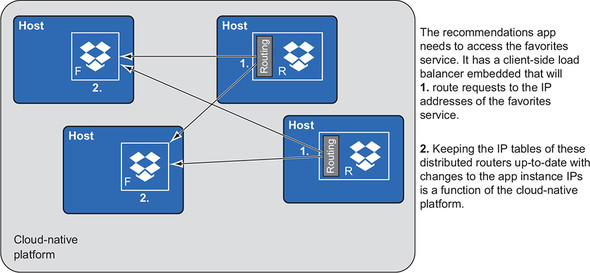
Individual services are running in separate containers and on different hosts; in order for one service to call another, it must first be able to find the other service. One of the ways that can happen is via the well-known patterns of the World Wide Web: DNS and routing. The recommendations service calls the favorites service via its URL, the URL is resolved to an IP address through DNS lookup, and that IP address points to a router that then sends on the request to one of the instances of the favorites service ([figure 3.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig09)).

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



Another way is to have the recommendations service directly access instances of the favorites service via IP address, but because there are many instances of the latter, the requests must be load-balanced as before. [Figure 3.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig10) depicts that this pulls the routing function into the calling service, thereby distributing the routing function itself.

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



Whether the routing function is logically centralized ([figure 3.9](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig09)) or highly distributed ([figure 3.10](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig10)), keeping what are effectively routing tables up-to-date is an important process. To fully automate this process, the platform implements patterns such as collecting IP address information from newly launched or recovered microservice instances, and distributing that data to the routing components, wherever they may be.

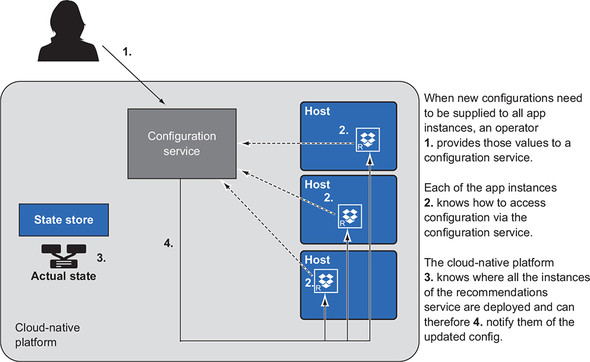
**Service configuration**

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

To provide this capability in a cloud-native setting, a configuration service is required. This service works in concert with other parts of the platform to implement what’s shown in [figure 3.11](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig11). The process depicted there is as follows:

1. An operator will supply the new configuration values to a configuration service (likely via a commit to a source code control system).
2. Service instances know how to access a configuration service from which they obtain configuration values whenever necessary. Certainly, the service instances will do this at startup time, but they must also do this when the configuration values are changed or when certain lifecycle events occur.
3. When configuration values are changed, the trick is to have each service instance refresh itself; the platform knows about all the service instances. The actual state exists in the state store.
4. The platform notifies each of the service instances that new values are available, and the instances take on those new values.

**Figure 3.11. 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, but are exemplars of the runtime support needed for the modular and highly distributed nature of the cloud-native application. Other services include the following:

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

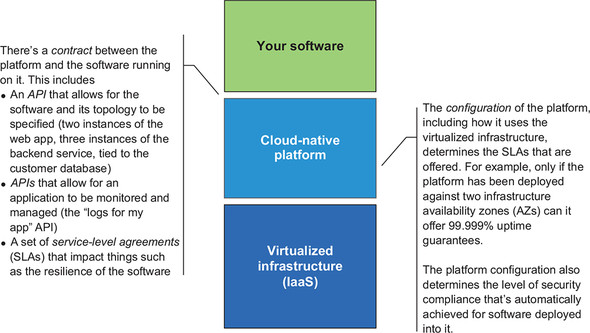
These and many more services are table stakes for a cloud-native platform and greatly reduce the burden that would otherwise be placed on the developer and operator of the modern software we’re now building. Adoption of such a platform is essential for a high-functioning IT organization.

**3.3. Who does what?**

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

[Figure 3.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig12), a variant of [figure 3.4](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig04), shows the same stack, but now I want to home in on the boundary between the cloud-native platform and your software. At that boundary is a contract that determines how the software must be provided (the platform API) and a set of service levels that provide guarantees around how well the software will run on the platform.

**Figure 3.12. The cloud-native platform presents a contract that allows consumers to deploy and manage their software without being exposed to low-level infrastructure details. Nonfunctional requirements such as performance guarantees are realized via SLAs that are achieved via specific platform configurations.**



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

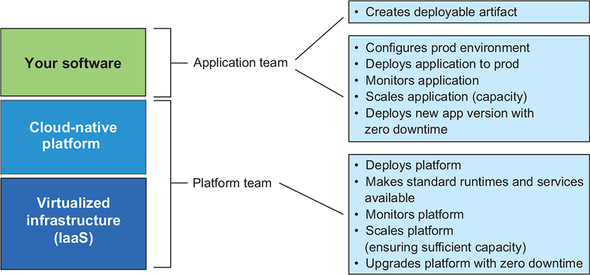
Establishing these boundaries and contracts enables something powerful: it allows you to form separate teams. One team is responsible for configuring the cloud-native platform in such a way as to provide the service levels required by the organization. Members of this platform team have a particular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

The other team, or shall I say *teams*, are the application teams whose members build and operate software for end consumers. They know how to use the platform APIs to deploy and manage the apps they’re running there. Members of these teams have skills profiles that enable them to understand cloud-native software architectures and know how to monitor and configure them for optimal performance.

[Figure 3.13](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig13) shows the part of the full stack that each team is responsible for. I want to draw your attention to two elements of this diagram:

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

**Figure 3.13. The right abstractions support the formation of autonomous platform and application teams. Each is responsible for deployment, monitoring, scaling, and upgrading their respective products.**



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

But achieving that autonomy, which is such an essential ingredient for delivering digital solutions in this era, depends not only on the definition of the contracts, but 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 by disallowing snowflakes and enabling app team autonomy, while concurrently implementing security, regulatory, and other controls. And it must provide services that lessen the burdens that are added when we create software composed of many highly distributed app components (microservices) running in a multitenant environment.

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

**3.4. More cloud-native platform capabilities**

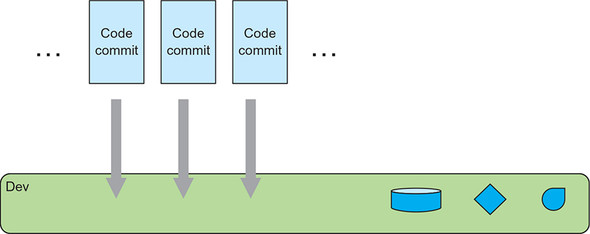
Now that you understand the basic support that platforms provide for highly distributed software running in an environment of constant change, as well as the workings of the application team and platform team, let’s look at additional factors you should understand about the cloud-native platform.

**3.4.1. The platform supports the entire SDLC**

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

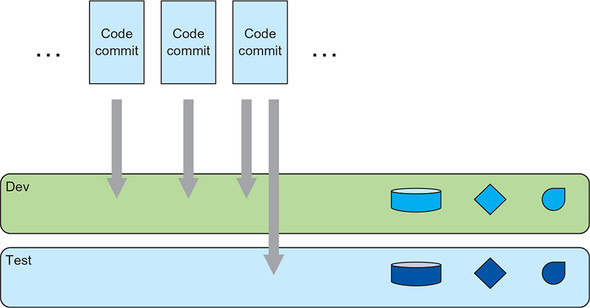
After you, as a developer, verify that the code is running on your own workstation, you check in the code. This kicks off a pipeline that builds the deployable artifact, installs it into an official dev environment, and runs the test suite. If the tests pass, you can move on to implementing the next feature, and the cycle continues. [Figure 3.14](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig14) depicts these deployments 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 3.14. Code commits generate deployable artifacts that are deployed into a dev environment that looks similar to production, but has development versions of services such as databases and message queues (depicted by the symbols on the right).**



Another, less-frequent, trigger, perhaps a time-based one that runs daily, will deploy the artifact into testing, where a more comprehensive (and likely longer-running) set of tests are executed in an environment that’s a bit closer to production. You’ll notice that in [figure 3.15](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig15), the general shape of the test environment is the same as that of the dev environment, but the two are shaded differently, indicating variances. For example, the network topology in the dev environment might be flat, with all apps being deployed into the same subnet, whereas in the test environment, the network may be partitioned to provide security boundaries.

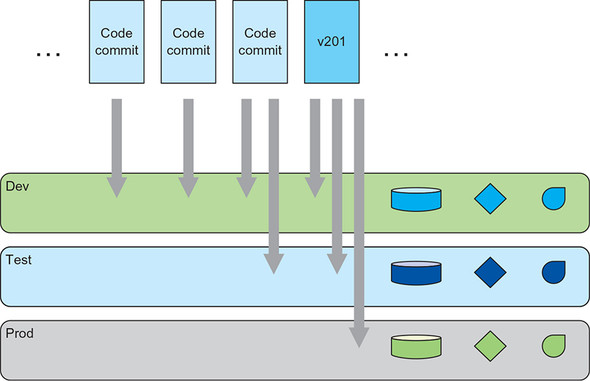
**Figure 3.15. The same deployable artifact is deployed into a staging environment, where it’s bound to services (depicted by the symbols on the right) that more closely match those that exist in production.**



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

Finally, when the business decides it would like to release the software, the artifact is tagged with a release version and deployed into production; see [figure 3.16](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig16). The production environment, including the service instances, again differs from that of testing. For example, here the app is bound to the live customer database.

**Figure 3.16. The same artifact is deployed into similar environments throughout the SDLC and must absorb the unavoidable differences across the environments.**



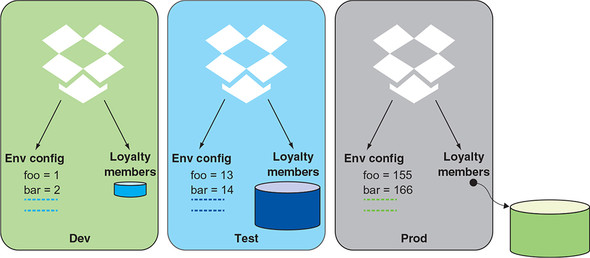
Although differences exist in the dev, test, and production environments, I hinted at and want to emphasize that important similarities exist as well. For example, the API used to deploy into any of the environments is the same; managing the automation essential for a streamlined SDLC process with varying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

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

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

[Figure 3.17](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig17) offers a visual depiction of this concept. The artifact deployed into each of the spaces is exactly the same. The contracts between the app and the environment config, and the app and services (in this case, the Loyalty Members database), are also uniform. Note that the arrows pointing from each of the deployable artifacts are exactly the same across all environments—what differs are the details behind the *env config* and *loyalty members* abstractions. Abstractions such as these are an essential part of a platform that’s designed to support the entire SDLC.

**Figure 3.17. 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.**



On occasion, I’ve had clients implement a platform only for preproduction environments or only for production. There’s no question that having a cloud-native platform that offers capabilities such as automated health management or a means of controlling standardized machine images provides value, even if available only in production. But given the need for continuous delivery of digital solutions, the platform must be applied across the entire SDLC. When the platform offers environment parity with the right abstractions, and an API that can be used to automate all interactions with it, the process of developing software and bringing it all the way through to production can be turned into a predictable, efficient machine.

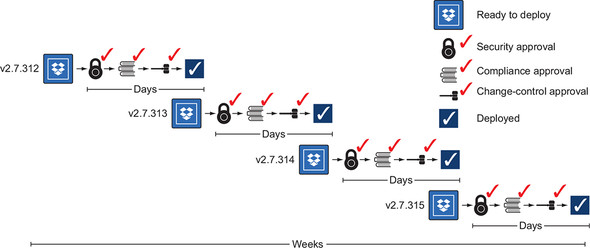
**3.4.2. Security, change-control, compliance (the control functions)**

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

The trouble with the current process isn’t the people, or even the organizations from which they come. The challenges arise because there are simply too many ways to make a mistake. A developer could, for example, specify a dependence on a particular version of the JRE that had been known to cause performance degradations for certain types of workloads and was therefore no longer permitted on production systems—so change control is needed to keep it from going to production. Do you have a control that says every user access to a particular database must be logged? The compliance office is going to verify that logging agents are correctly deployed and configured. An explicit and often manual check that the rules are being followed is sometimes the only point of control.

Those points of control are implemented in various places across the application lifecycle and all too often are pushed too late in the cycle. When a deficiency is detected only the day before a planned deployment, the schedule is then at great risk, deadlines are missed, and everyone is unhappy. The most sobering thing about this, illustrated in [figure 3.18](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig18), is that these controls apply to every deployment: every version, of every app. The time from *ready to ship* to *deployed* is, at best, counted in days, and multiple deployments in weeks.

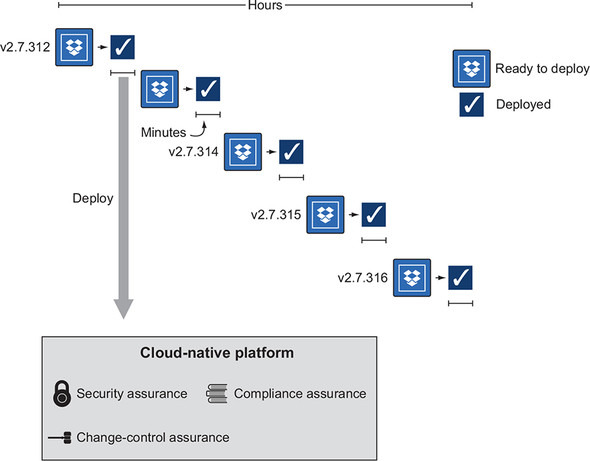
**Figure 3.18. Control functions that are on the critical path for every release of every app reduce the number of deployments that can be performed.**



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

In just a moment I’ll talk about how baking those controls into the platform works, but first let’s look at the outcome. If a deployment is guaranteed to meet the controls, you no longer have to go through a checklist before it happens. And if you no longer have a lengthy checklist, then the time from when you have your artifact ready to go to the time that it’s deployed shrinks dramatically. A deployment that once took days now takes minutes. And whereas a sequence of deployments took weeks, you can now complete several cycles within a single day ([figure 3.19](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig19)). You can try things out and get feedback far more frequently than before, and you’ve already studied the many benefits of such a practice.

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



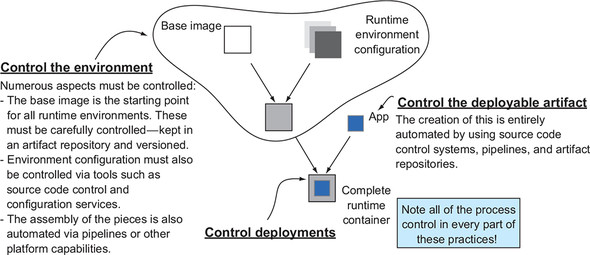
Next, let’s dig into how such controls are built into the cloud-native platform. How do you get the security, compliance, and change-control assurances that are claimed in [figure 3.19](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig19)?

**3.4.3. Controlling what goes into the container**

In [chapter 2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02), I talked about the need for repeatability and that you achieve it by controlling the runtime environment, controlling the deployable artifact, and controlling the deployment process itself. Using container technology, you have a way of baking that level of control directly into the platform and can therefore achieve the security, compliance, and change-control assurances you’re after.

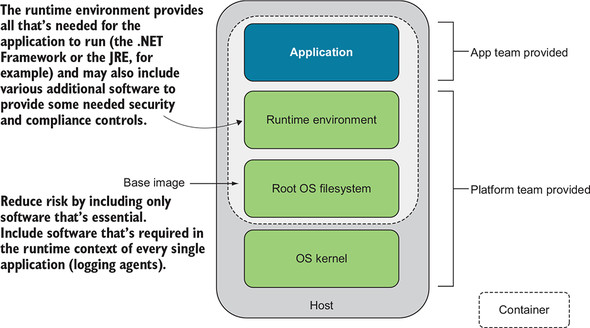
[Figure 3.20](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig20) repeats [figure 2.12](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02fig12) from [chapter 2](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_013.html#ch02), which addressed the way the various parts of a running application are combined. Now that we’ve talked about containers, we can map each of the pieces in this diagram to the very entity that will be the running application.

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



[Figure 3.21](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig21) shows a container running on a host. What I call the “base image” in [figure 3.20](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig20) is now clearly shown as the root OS filesystem of the container. The runtime environment represents additional components that are installed into the root filesystem, such as the JRE or the .NET runtime. And then, finally, the deployable artifact also comes into the container. So then, how do you control each of these pieces?

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



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

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

|  |
| --- |
|  |

**Point 1**

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

|  |
| --- |
|  |

That base image can be used as a foundation for a variety of specialized workloads. For example, some of your apps might be written in Java and therefore require the JRE, and other apps might be written in Python and therefore need the Python interpreter installed. This is the role of what is shown in [figure 3.21](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig21) as the runtime environment. Of course, parts of that runtime environment, such as the JRE or the Python interpreter, may have themselves vulnerabilities, so the security office will have specific versions that are approved for use.

|  |
| --- |
|  |

**Point 2**

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

|  |
| --- |
|  |

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

|  |
| --- |
|  |

**Point 3**

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

|  |
| --- |
|  |

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

Looking back at [figure 3.21](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig21), you can see that the platform team supplies approved base images and approved runtime environments. You can also see that the platform team is responsible for the OS kernel that’s running on the host. In short, the platform team is able to impose security and compliance controls through each of the layers you’ve just studied.

**3.4.4. Upgrading and patching vulnerabilities**

When any part of the application container depicted in [figure 3.21](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig21) needs to be updated, running instances aren’t modified. Instead, you deploy *new* containers with the new set of components. Because cloud-native apps always have multiple instances deployed, you can move from an old version to the new with zero downtime.

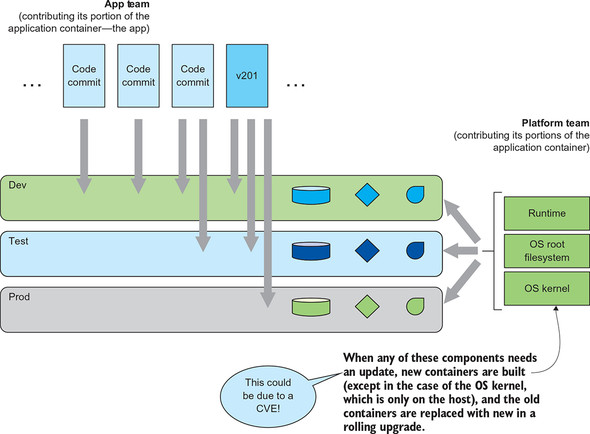
The basic pattern for such an upgrade is that (1) a subset of the app instances are shut down and disposed of, (2) an equal number of instances of the new container are launched, and (3) after they’re up and running, you move on to replacing the next batch of old instances. The cloud-native platform handles this process for you; you need only provide the new version of the app.

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

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

[Figure 3.22](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig22) extends earlier diagrams that showed the app team doing deployments into the dev, test, and prod environments. Now you understand that with each deployment from the app team, a container is assembled with pieces supplied by the platform team and pieces supplied by the app team (recall [figure 3.21](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig21)).

**Figure 3.22. The right platform enables app teams to operate independently from platform teams.**



It follows that when there are new versions of the parts of the container supplied by the platform team, a new container could also be assembled. If the app team has something new, a new container is assembled and deployed, and if the platform team has something new, a new container is assembled and deployed. This is shown in [figure 3.22](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig22): from above, the app team is creating new containers; and from the side, on their own schedule, the platform team is revving the platform elements. The platform team is updating the platform-supplied portions of the container.

This autonomy is essential for patch management in the data center. When a new vulnerability (CVE) is found,**[**[**3**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fn3)**]** patches need to be applied quickly, without complex coordination across all apps running in the data center. This type of complex coordination was part of the reason that patches were often not applied as rapidly as they should have been in prior data center configurations.

***3***

*CVE is an acronym for Common Vulnerabilities and Exposures; Wikipedia offers more details at*[*http://mng.bz/QQr6*](http://mng.bz/QQr6)*.*

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

If you think about this for a moment from the perspective of Google Cloud Platform, Amazon Web Services, Azure, or any of the other cloud platform providers, this type of autonomy between the platform team and the users of the platform is essential. With over one million active users,**[**[**4**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fn4)**]** AWS couldn’t manage its platform offering if it required coordination with individual members of that user base. You can absolutely apply the same practices in your data center, with the help of a cloud-native application platform.

***4***

*See “Amazon’s AWS Is Now a $7.3B Business as It Passes 1M Active Customers,” by Ingrid Lunden (*[*http://mng.bz/Xgm9*](http://mng.bz/Xgm9)*) for more details.*

**3.4.5. Change control**

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

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

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

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

But as you know, architectures have changed, and the smaller, individual parts that come together to form cloud-native software, coupled with the far more dynamic environments these apps are running in, have stressed the VM-based platforms. Although other attempts were previously made, container-based approaches have proven an outstanding solution. Based on the foundational concepts of control groups (cgroups), which control the use of shared resources, and namespaces, which control the visibility of shared resources, Linux containers have become the execution environments for the microservices collective that forms cloud-native software.**[**[**5**](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fn5)**]**

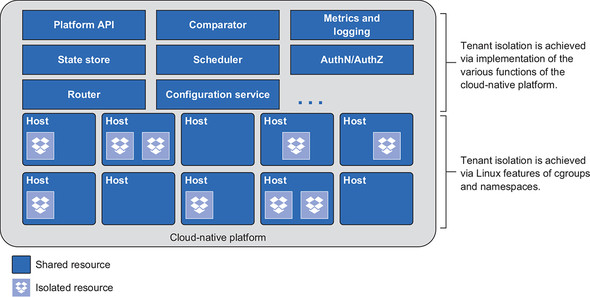
***5***

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

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

The answer is that the platform itself, which provides access control, monitoring, routing functions, and more, must be tenant aware. [Figure 3.23](https://learning.oreilly.com/library/view/cloud-native-patterns/9781617294297/kindle_split_014.html#ch03fig23) shows a set of hosts at the bottom, where Linux cgroups and namespaces are providing the compute isolation you need. In the upper part of the diagram are a whole host of other platform components that govern its use. The platform API is the place that access control is enforced. The metrics and logging system needs to group collected data into buckets for individual tenants. The scheduler, which determines where containers will be run, must be aware of relationships within a tenant and across tenants. In short, the cloud-native application platform is multitenant.

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



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

**Summary**

* A cloud-native platform takes on a great deal of the burden of satisfying the requirements on modern software.
* The cloud-native platform is used throughout the entire software development lifecycle.
* A cloud-native platform projects higher-level abstraction than that of the infrastructure-centric platforms of the last decade.
* By baking control functions into the platform, deployments can be done far more frequently and are safer than when approvals are needed for every version of every app.
* App teams and platform teams can work independently, each managing the construction, deployment, and maintenance of their respective products.
* Eventual consistency is at the core of the platform as it constantly monitors the actual state of the system, compares it to the desired state, and remediates when necessary. This applies to both the software running on the platform and to the deployment of the platform itself.
* As software becomes more modular and distributed, the services that bring the components together into a whole do so as well. The platform must bake in support for these distributed systems.
* A cloud-native platform is absolutely essential for organizations building and operating cloud-native software.