CHAPTER 1

What Is a "Cloud Native" Application?

The most dangerous phrase in the language is, "We've always done it this way."[1]

—Grace Hopper, *Computerworld (January 1976)*

If you're reading this book, then you've no doubt at least heard the term *cloud native* before. More likely, you've probably seen some of the many, many articles written by vendors bubbling over with breathless adoration and dollar signs in their eyes. If this is the bulk of your experience with the term so far, then you can be forgiven for thinking the term to be ambiguous and buzzwordy, just another of a series of markety expressions that might have started as something useful but have since been taken over by people trying to sell you something. See also: Agile, DevOps.

For similar reasons, a web search for "cloud native definition" might lead you to think that all an application needs to be cloud native is to be written in the "right" language[2] or framework, or to use the "right" technology. Certainly, your choice of language can make your life significantly easier or harder, but it's neither necessary nor sufficient for making an application cloud native.

Is cloud native, then, just a matter of *where* an application runs? The term *cloud native* certainly suggests that. All you'd need to do is pour your kludgy[3] old application into a container and run it in Kubernetes, and you're cloud native now, right? Nope. All you've done is make your application harder to deploy and harder to manage.[4] A kludgy application in Kubernetes is still kludgy.

1 Surden, Esther. "Privacy Laws May Usher in Defensive DP: Hopper." *Computerworld*, 26 Jan. 1976, p. 9.

2 Which is Go. Don't get me wrong—this is still a Go book after all.

3 A "kludge" is "an awkward or inelegant solution." It's a fascinating word with a fascinating history.

4 Have you ever wondered why so many Kubernetes migrations fail?

So, what *is* a cloud native application? In this chapter, we'll answer exactly that. First, we'll examine the history of computing service paradigms up to (and especially) the present, and discuss how the relentless pressure to scale drove (and continues to drive) the development and adoption of technologies that provide high levels of dependability at often vast scales. Finally, we'll identify the specific attributes associated with such an application.

The Story So Far

The story of networked applications is the story of the pressure to scale.

The late 1950s saw the introduction of the mainframe computer. At the time, every program and piece of data was stored in a single giant machine that users could access by means of dumb terminals with no computational ability of their own. All the logic and all the data all lived together as one big happy monolith. It was a simpler time.

Everything changed in the 1980s with the arrival of inexpensive network-connected PCs. Unlike dumb terminals, PCs were able to do some computation of their own, making it possible to offload some of an application's logic onto them. This new multitiered architecture—which separated presentation logic, business logic, and data (Figure 1-1)—made it possible, for the first time, for the components of a networked application to be modified or replaced independent of the others.

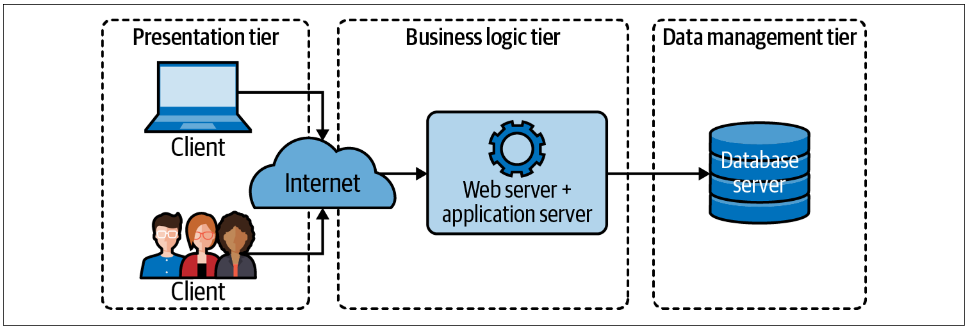


Figure 1-1. A traditional three-tiered architecture, with clearly defined presentation, business logic, and data components

In the 1990s, the popularization of the World Wide Web and the subsequent "dot-com" gold rush introduced the world to software as a service (SaaS). Entire industries were built on the SaaS model, driving the development of more complex and resource-hungry applications, which were in turn harder to develop, maintain, and deploy. Suddenly the classic multitiered architecture wasn't enough anymore. In response, business logic started to get decomposed into subcomponents that could be developed, maintained, and deployed independently, ushering in the age of microservices.

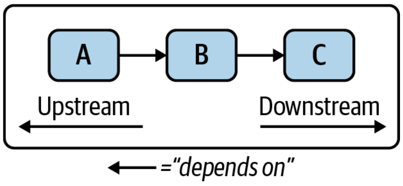
In 2006, Amazon launched Amazon Web Services (AWS), which included the Elastic Compute Cloud (EC2) service. Although AWS wasn't the *first* infrastructure as a service (IaaS) offering, it revolutionized the on-demand availability of data storage and computing resources, bringing Cloud Computing—and the ability to quickly scale—to the masses, catalyzing a massive migration of resources into "the cloud."

Unfortunately, organizations soon learned that life at scale isn't easy. Bad things happen, and when you're working with hundreds or thousands of resources (or more!), bad things happen *a lot*. Traffic will wildly spike up or down, essential hardware will fail, upstream dependencies will become suddenly and inexplicably inaccessible. Even if nothing goes wrong for a while, you still have to deploy and manage all of these resources. At this scale, it's impossible (or at least wildly impractical) for humans to keep up with all of these issues manually.

Upstream and Downstream Dependencies

In this book we'll sometimes use the terms *upstream dependency* and *downstream dependency* to describe the relative positions of two resources in a dependency relationship. There's no real consensus in the industry around the directionality of these terms, so this book will use them as follows:

Imagine that we have three services: A, B, and C, as shown in the following figure:



In this scenario, Service A makes requests to (and therefore depends on) Service B, which in turn depends on Service C.

Because Service B depends on Service C, we can say that Service C is a *downstream dependency* of Service B. By extension, because Service A depends on Service B which depends on Service C, Service C is also a *transitive downstream dependency* of Service A.

Inversely, because Service C is depended upon by Service B, we can say that Service B is an *upstream dependency* of Service C, and that Service A is a *transitive upstream dependency* of Service A.

What Is Cloud Native?

Fundamentally, a truly cloud native application incorporates everything we've learned about running networked applications at scale over the past 60 years. They are scalable in the face of wildly changing load, resilient in the face of environmental uncertainty, and manageable in the face of ever-changing requirements. In other words, a cloud native application is built for life in a cruel, uncertain universe.

But how do we *define* the term *cloud native*? Fortunately for all of us,[5] we don't have to. The Cloud Native Computing Foundation—a subfoundation of the renowned Linux Foundation, and something of an acknowledged authority on the subject—has already done it for us:

> Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds….
>
> These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil.[6]
>
> —Cloud Native Computing Foundation, *CNCF Cloud Native Definition v1.0*

By this definition, cloud native applications are more than just applications that happen to live in a cloud. They're also *scalable*, *loosely coupled*, *resilient*, *manageable*, and *observable*. Taken together, these "cloud native attributes" can be said to constitute the foundation of what it means for a system to be cloud native.

As it turns out, each of those words has a pretty specific meaning of its own, so let's take a look.

Scalability

In the context of cloud computing, *scalability* can be defined as the ability of a system to continue to behave as expected in the face of significant upward or downward changes in demand. A system can be considered to be scalable if it doesn't need to be refactored to perform its intended function during or after a steep increase in demand.

Because unscalable services can seem to function perfectly well under initial conditions, scalability isn't always a primary consideration during service design. While this might be fine in the short term, services that aren't capable of growing much beyond their original expectations also have a limited lifetime value. What's more, it's

5 Especially for me. I get to write this cool book.

6 Cloud Native Computing Foundation. "CNCF Cloud Native Definition v1.0," GitHub, 7 Dec. 2020. *https://oreil.ly/KJuTr*.

often fiendishly difficult to refactor a service for scalability, so building with it in mind can save both time and money in the long run.

There are two different ways that a service can be scaled, each with its own associated pros and cons:

Vertical scaling
: A system can be *vertically scaled* (or *scaled up*) by upsizing (or downsizing) the hardware resources that are already allocated to it. For example, by adding memory or CPU to a database that's running on a dedicated computing instance. Vertical scaling has the benefit of being technically relatively straightforward, but any given instance can only be upsized so much.

Horizontal scaling
: A system can be *horizontally scaled* (or *scaled out*) by adding (or removing) service instances. For example, this can be done by increasing the number of service nodes behind a load balancer or containers in Kubernetes, or another container orchestration system. This strategy has a number of advantages, including redundancy and freedom from the limits of available instance sizes. However, more replicas mean greater design and management complexity, and not all services can be horizontally scaled.

Given that there are two ways of scaling a service—up or out—does that mean that any service whose hardware can be upscaled (and is capable of taking advantage of increased hardware resources) is "scalable"? If you want to split hairs, then sure, to a point. But how scalable is it? Vertical scaling is inherently limited by the size of available computing resources, so a service that can only be scaled up isn't very scalable at all. If you want to be able to scale by ten times, or a hundred, or a thousand, your service really has to be horizontally scalable.

So what's the difference between a service that's horizontally scalable and one that's not? It all boils down to one thing: state. A service that doesn't maintain any application state—or which has been very carefully designed to distribute its state between service replicas—will be relatively straightforward to scale out. For any other application, it will be hard. It's that simple.

The concepts of scalability, state, and redundancy will be discussed in much more depth in Chapter 7.

Loose Coupling

Loose coupling is a system property and design strategy in which a system's components have minimal knowledge of any other components. Two systems can be said to be *loosely coupled* when changes to one component generally don't require changes to the other.