

PART I

Going Cloud Native

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 market 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” language² 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 kludgy³ 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.

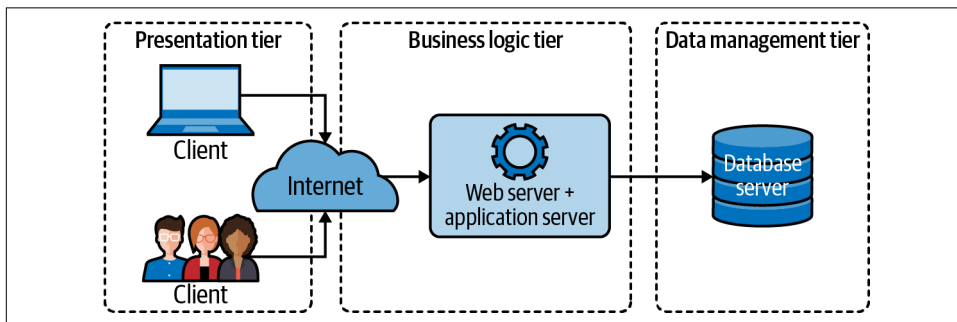


Figure 1-1. A traditional three-tiered architecture, with clearly defined presentation, business logic, and data components

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.

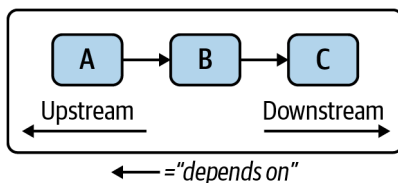
In 2006, Amazon launched Amazon Web Services (AWS), which included the Elastic Compute Cloud (EC2) service. Although AWS wasn't the *first* infrastructure as a service (IaaS) offering, it revolutionized the on-demand availability of data storage and computing resources, bringing Cloud Computing—and the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

For example, web servers and web browsers can be considered to be loosely coupled: servers can be updated or even completely replaced without affecting our browsers at all. In their case, this is possible because standard web servers have agreed that they would communicate using a set of standard protocols.⁷ In other words, they provide a *service contract*. Imagine the chaos if all the world’s web browsers had to be updated each time NGINX or httpd had a new version!⁸

It could be said that “loose coupling” is just a restatement of the whole point of microservice architectures: to partition components so that changes in one don’t necessarily affect another. This might even be true. However, this principle is often neglected, and bears repeating. The benefits of loose coupling—and the consequences if it’s neglected—cannot be understated. It’s very easy to create a “worst of all worlds” system that pairs the management and complexity overhead of having multiple services with the dependencies and entanglements of a monolithic system: the dreaded *distributed monolith*.

Unfortunately, there’s no magic technology or protocol that can keep your services from being tightly coupled. Any data exchange format can be misused. There are, however, several that help, and—when applied with practices like declarative APIs and good versioning practices—can be used to create services that are both loosely-coupled *and* modifiable.

These technologies and practices will be discussed and demonstrated in detail in [Chapter 8](#).

Resilience

Resilience (roughly synonymous with *fault tolerance*) is a measure of how well a system withstands and recovers from errors and faults. A system can be considered *resilient* if it can continue operating correctly—possibly at a reduced level—rather than failing completely when some part of the system fails.

When we discuss resilience (and the other the other “cloud native attributes” as well, but especially when we discuss resilience) we use the word “system” quite a lot. A *system*, depending on how it’s used, can refer to anything from a complex web of interconnected services (such as an entire distributed application), to a collection of closely related components (such as the replicas of a single function or service instance), or a single process running on a single machine. Every system is composed of several subsystems, which in turn are composed of sub-subsystems, which are themselves composed of sub-sub-subsystems. It’s turtles all the way down.

7 Those of us who remember the Browser Wars of the 1990s will recall that this wasn’t always strictly true.

8 Or if every website required a different browser. That would stink, *wouldn’t it*?

In the language of systems engineering, any system can contain defects, or *faults*, which we lovingly refer to as *bugs* in the software world. As we all know too well, under certain conditions, any fault can give rise to an *error*, which is the name we give to any discrepancy between a system's intended behavior and its actual behavior. Errors have the potential to cause a system to fail to perform its required function: a *failure*. It doesn't stop there though: a failure in a subsystem or component becomes a fault in the larger system; any fault that isn't properly contained has the potential to cascade upwards until it causes a total system failure.

In an ideal world, every system would be carefully designed to prevent faults from ever occurring, but this is an unrealistic goal. You can't prevent every possible fault, and it's wasteful and unproductive to try. However, by assuming that all of a system's components are certain to fail—which they are—and designing them to respond to potential faults and limit the effects of failures, you can produce a system that's functionally healthy even when some of its components are not.

There are many ways of designing a system for resiliency. Deploying redundant components is perhaps the most common approach, but that also assumes that a fault won't affect all components of the same type. Circuit breakers and retry logic can be included to prevent failures from propagating between components. Faulty components can even be reaped—or can intentionally fail—to benefit the larger system.

We'll discuss all of these approaches (and more) in much more depth in [Chapter 9](#).

Resilience Is Not Reliability

The terms *resilience* and *reliability* describe closely related concepts, and are often confused. But, as we'll discuss in [Chapter 9](#), they aren't quite the same thing:⁹

- The resilience of a system is the degree to which it can continue to operate correctly in the face of errors and faults. Resilience, along with the other four cloud native properties, is just one factor that contributes to reliability.
- The reliability of a system is its ability to behave as expected for a given time interval. Reliability, in conjunction with attributes like availability and maintainability, contributes to a system's overall dependability.

⁹ If you're interested in a complete academic treatment, I highly recommend *Reliability and Availability Engineering* by Kishor S. Trivedi and Andrea Bobbio.

Manageability

A system's *manageability* is the ease (or lack thereof) with which its behavior can be modified to keep it secure, running smoothly, and compliant with changing requirements. A system can be considered *manageable* if it's possible to sufficiently alter its behavior without having to alter its code.

As a system property, manageability gets a lot less attention than some of the more attention-grabbing attributes like scalability or observability. It's every bit as critical, though, particularly in complex, distributed systems.

For example, imagine a hypothetical system that includes a service and a database, and that the service refers to the database by a URL. What if you needed to update that service to refer to another database? If the URL was hardcoded you might have to update the code and redeploy, which, depending on the system, might be awkward for its own reasons. Of course, you could update the DNS record to point to the new location, but what if you needed to redeploy a development version of the service, with its own development database?

A manageable system might, for example, represent this value as an easily modified environment variable; if the service that uses it is deployed in Kubernetes, adjustments to its behavior might be a matter of updating a value in a ConfigMap. A more complex system might even provide a declarative API that a developer can use to tell the system what behavior she expects. There's no single right answer.¹⁰

Manageability isn't limited to configuration changes. It encompasses all possible dimensions of a system's behavior, be it the ability to activate feature flags, or rotate credentials or TLS certificates, or even (and perhaps especially) deploy or upgrade (or downgrade) system components.

Manageable systems are designed for adaptability, and can be readily adjusted to accommodate changing functional, environmental, or security requirements. Unmanageable systems, on the other hand, tend to be far more brittle, frequently requiring ad hoc—often manual—changes. The overhead involved in managing such systems places fundamental limits on their scalability, availability, and reliability.

The concept of manageability—and some preferred practices for implementing them in Go—will be discussed in much more depth in [Chapter 10](#).

¹⁰ There are some wrong ones though.

Manageability Is Not Maintainability

It can be said that manageability and maintainability have some “mission overlap” in that they’re both concerned with the ease with which a system can be modified,¹¹ but they’re actually quite different:

- Manageability describes the ease with which changes can be made to the behavior of a running system, up to and including deploying (and redeploying) components of that system. It’s how easy it is to make changes *from the outside*.
- Maintainability describes the ease with which changes can be made to a system’s underlying functionality, most often its code. It’s how easy it is to make changes *from the inside*.

Observability

The *observability* of a system is a measure of how well its internal states can be inferred from knowledge of its external outputs. A system can be considered *observable* when it’s possible to quickly and consistently ask novel questions about it with minimal prior knowledge, and without having to reinstrument or build new code.

On its face, this might sound simple enough: just sprinkle in some logging and slap up a couple of dashboards, and your system is observable, right? Almost certainly not. Not with modern, complex systems in which almost any problem is the manifestation of a web of multiple things going wrong simultaneously. The Age of the LAMP Stack is over; things are harder now.

This isn’t to say that metrics, logging, and tracing aren’t important. On the contrary: they represent the building blocks of observability. But their mere existence is not enough: data is not information. They need to be used the right way. They need to be rich. Together, they need to be able to answer questions that you’ve never even thought to ask before.

The ability to detect and debug problems is a fundamental requirement for the maintenance and evolution of a robust system. But in a distributed system it’s often hard enough just figuring out *where* a problem is. Complex systems are just too...*complex*. The number of possible failure states for any given system is proportional to the product of the number of possible partial and complete failure states of each of its components, and it’s impossible to predict all of them. The traditional approach of focusing attention on the things we expect to fail simply isn’t enough.

¹¹ Plus, they both start with *M*. Super confusing.

Emerging practices in observability can be seen as the evolution of monitoring. Years of experience with designing, building, and maintaining complex systems have taught us that traditional methods of instrumentation—including but not limited to dashboards, unstructured logs, or alerting on various “known unknowns”—just aren’t up to the challenges presented by modern distributed systems.

Observability is a complex and subtle subject, but, fundamentally, it comes down to this: instrument your systems richly enough and under real enough scenarios so that, in the future, you can answer questions that you haven’t thought to ask yet.

The concept of observability—and some suggestions for implementing it—will be discussed in much more depth in [Chapter 11](#).

Why Is Cloud Native a Thing?

The move towards “cloud native” is an example of architectural and technical adaptation, driven by environmental pressure and selection. It’s evolution—survival of the fittest. Bear with me here; I’m a biologist by training.

Eons ago, in the Dawn of Time,¹² applications would be built and deployed (generally by hand) to one or a small number of servers, where they were carefully maintained and nurtured. If they got sick, they were lovingly nursed back to health. If a service went down, you could often fix it with a restart. Observability was shelling into a server to run `top` and review logs. It was a simpler time.

In 1997, only 11% of people in industrialized countries, and 2% worldwide, were regular internet users. The subsequent years saw exponential growth in internet access and adoption, however, and by 2017 that number had exploded to 81% in industrialized countries and 48% worldwide¹³—and continues to grow.

All of those users—and their money—applied stress to services, generating significant incentive to scale. What’s more, as user sophistication and dependency on web services grew, so did expectations that their favorite web applications would be both feature-rich and always available.

The result was, and is, a significant evolutionary pressure towards scale, complexity, and dependability. These three attributes don’t play well together, though, and the traditional approaches simply couldn’t, and can’t, keep up. New techniques and practices had to be invented.

¹² That time was the 1990s.

¹³ International Telecommunication Union (ITU). “Internet users per 100 inhabitants 1997 to 2007” and “Internet users per 100 inhabitants 2005 to 2017.” *ICT Data and Statistics (IDS)*.

Fortunately, the introduction of public clouds and IaaS made it relatively straightforward to scale infrastructure out. Shortcomings with dependability could often be compensated for with sheer numbers. But that introduced new problems. How do you maintain a hundred servers? A thousand? Ten thousand? How do you install your application onto them, or upgrade it? How do you debug it when it misbehaves? How do you even know it's healthy? Problems that are merely annoying at small scale tend to become very hard at large scale.

Cloud native is a thing because scale is the cause of (and solution to) all our problems. It's not magic. It's not special. All fancy language aside, cloud native techniques and technologies exist for no other reasons than to make it possible to leverage the benefits of a "cloud" (quantity) while compensating for its downsides (lack of dependability).

Summary

In this chapter, we talked a fair amount about the history of computing, and how what we now call "cloud native" isn't a new phenomenon so much as the inevitable outcome of a virtuous cycle of technological demand driving innovation driving more demand.

Ultimately, though, all of those fancy words distill down to a single point: today's applications have to dependably serve a lot of people. The techniques and technologies that we call "cloud native" represent the best current practices for building a service that's scalable, adaptable, and resilient enough to do that.

But what does all of this do with Go? As it turns out, cloud native infrastructure requires cloud native tools. In [Chapter 2](#), we'll start to talk about what that means, exactly.

Why Go Rules the Cloud Native World

Any intelligent fool can make things bigger, more complex, and more violent. It takes a touch of genius—and a lot of courage—to move in the opposite direction.¹

—E.F. Schumacher, *Small Is Beautiful* (August 1973)

The Motivation Behind Go

The idea of Go emerged in September of 2007 at Google, the inevitable outcome of putting a bunch of smart people in a room and frustrating the heck out of them.

The people in question were Robert Griesemer, Rob Pike, and Ken Thompson; all already highly regarded for their individual work in designing other languages. The source of their collective ire was nothing less than the entire set of programming languages that were available at the time, which they were finding just weren't well-suited to the task of describing the kinds of distributed, scalable, resilient services that Google was building.²

Essentially, the common languages of the day had been developed in a different era, one before multiple processors were commonplace, and networks were quite so ubiquitous. Their support for multicore processing and networking—essential building blocks of modern “cloud native” services³—was often limited or required extraordinary efforts to utilize. Simply put, programming languages weren't keeping up with the needs of modern software development.

1 Schumacher, E.F. “Small Is Beautiful.” *The Radical Humanist*, August 1973, p. 22.

2 These were “cloud native” services before the term “cloud native” was coined.

3 Of course, they weren't called “cloud native” at the time; to Google they were just “services.”

Features for a Cloud Native World

Their frustrations were many, but all of them amounted to one thing: the undue complexity of the languages they were working with was making it harder to build server software. These included, but weren't limited to:⁴

Low program comprehensibility

Code had become too hard to read. Unnecessary bookkeeping and repetition was compounded by functionally overlapping features that often encouraged cleverness over clarity.

Slow builds

Language construction and years of feature creep resulted in build times that ran for minutes or hours, even on large build clusters.

Inefficiency

Many programmers responded to the aforementioned problems by adopting more fluid, dynamic languages, effectively trading efficiency and type safety for expressiveness.

High cost of updates

Incompatibilities between even minor versions of a language, as well as any dependencies it may have (and its transitive dependencies!) often made updating an exercise in frustration.

Over the years, multiple—often quite clever—solutions have been presented to address some of these issues in various ways, usually introducing additional complexity in the process. Clearly, they couldn't be fixed with a new API or language feature. So, Go's designers envisioned a modern language, the first language built for the cloud native era, supporting modern networked and multicore computing, expressive yet comprehensible, and allowing its users to focus on solving their problems instead of struggling with their language.

The result, the Go language, is notable as much for the features it explicitly *doesn't have* as it is for the ones it does. Some of those features (and nonfeatures) and the motivation behind them are discussed in the following sections.

Composition and Structural Typing

Object-oriented programming, which is based on the concept of “objects” of various “types” possessing various attributes, has existed since the 1960s, but it truly came into vogue in the early to mid-1990s with the release of Java and the addition of

⁴ Pike, Rob. “Go at Google: Language Design in the Service of Software Engineering.” Google, Inc., 2012.
<https://oreil.ly/6V9T1>.

object-oriented features to C++. Since then, it has emerged as the dominant programming paradigm, and remains so even today.

The promise of object-oriented programming is seductive, and the theory behind it even makes a certain kind of intuitive sense. Data and behaviors can be associated with *types* of things, which can be inherited by *subtypes* of those things. *Instances* of those types can be conceptualized as tangible objects with properties and behaviors—components of a larger system modeling concrete, real-world concepts.

In practice however, objected-oriented programming using inheritance often requires that relationships between types be carefully considered and painstakingly designed, and that particular design patterns and practices be faithfully observed. As such, as illustrated in **Figure 2-1**, the tendency in object-oriented programming is for the focus to shift away from developing algorithms, and towards developing and maintaining taxonomies and ontologies.

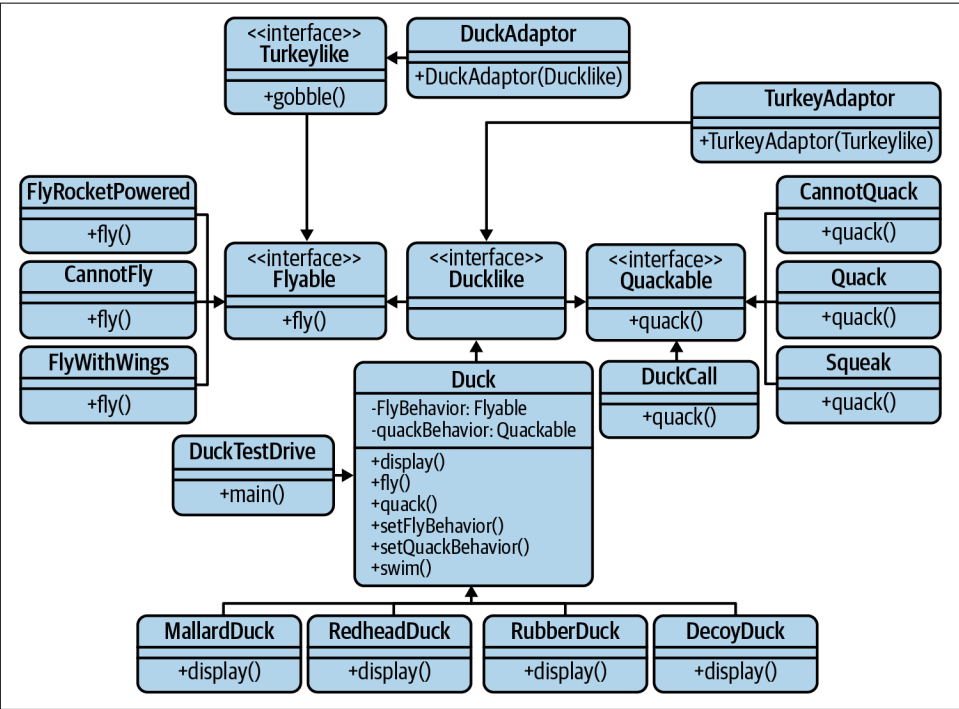


Figure 2-1. Over time, objected-oriented programming trends towards taxonomy

That's not to say that Go doesn't have object-oriented features that allow polymorphic behavior and code reuse. It, too, has a type-like concept in the form of *structs*, which can have properties and behaviors. What it rejects is inheritance and the elaborate relationships that come with it, opting instead to assemble more complex types by *embedding* simpler ones within them: an approach known as *composition*.

Specifically, where inheritance revolves around extending “is a” relationships between classes (i.e., a car “is a” motored vehicle), composition allows types to be constructed using “has a” relationships to define what they can do (i.e., a car “has a” motor). In practice, this permits greater design flexibility while allowing the creation of business domains that are less susceptible to disruption by the quirks of “family members.”

By extension, while Go uses interfaces to describe behavioral contracts, it has no “is a” concept, so equivalency is determined by inspecting a type’s definition, not its lineage. For example, given a Shape interface that defines an Area method, any type with an Area method will implicitly satisfy the Shape interface, without having to explicitly declare itself as a Shape:

```
type Shape interface {           // Any Shape must have an Area
    Area() float64
}

type Rectangle struct {          // Rectangle doesn't explicitly
    width, height float64        // declare itself to be a Shape
}

func (Rectangle r) Area() float64 { // Rectangle has an Area method; it
    return r.width * r.height       // satisfies the Shape interface
}
```

This *structural typing* mechanism, which has been described as *duck typing*⁵ at compile time, largely sheds the burdensome maintenance of tedious taxonomies that saddle more traditional object-oriented languages like Java and C++, freeing programmers to focus on data structures and algorithms.

Comprehensibility

Service languages like C++ and Java are often criticized for being clumsy, awkward to use, and unnecessarily verbose. They require lots of repetition and careful bookkeeping, saddling projects with superfluous boilerplate that gets in the way of programmers who have to divert their attention to things other than the problem they’re trying to solve, and limiting projects’ scalability under the weight of all the resulting complexity.

⁵ In languages that use duck typing, the type of an object is less important than the methods it defines. In other words, “if it walks like a duck and it quacks like a duck, then it must be a duck.”

Go was designed with large projects with lots of contributors in mind. Its minimalist design (just 25 keywords and 1 loop type), and the strong opinions of its compiler, strongly favor clarity over cleverness.⁶ This in turn encourages simplicity and productivity over clutter and complexity. The resulting code is relatively easy to ingest, review, and maintain, and harbors far fewer “gotchas.”

CSP-Style Concurrency

Most mainstream languages provide some means of running multiple processes concurrently, allowing a program to be composed of independently executed processes. Used correctly, concurrency can be incredibly useful, but it also introduces a number of challenges, particularly around ordering events, communication between processes, and coordination of access to shared resources.

Traditionally, a programmer will confront these challenges by allowing processes to share some piece of memory, which is then wrapped in locks or mutexes to restrict access to one process at a time. But even when well-implemented, this strategy can generate a fair amount of bookkeeping overhead. It’s also easy to forget to lock or unlock shared memory, potentially introducing race conditions, deadlocks, or concurrent modifications. This class of errors can be fiendishly difficult to debug.

Go, on the other hand, favors another strategy, based on a formal language called Communicating Sequential Processes (CSP), first described in Tony Hoare’s influential paper of the same name⁷ that describes patterns of interaction in concurrent systems in terms of message passing via channels.

The resulting concurrency model, implemented in Go with language primitives like `goroutines` and `channels`, makes Go uniquely⁸ capable of elegantly structuring concurrent software without depending entirely on locking. It encourages developers to limit sharing memory, and to instead allow processes to interact with one another *entirely* by passing messages. This idea is often summarized by the Go proverb:

Do not communicate by sharing memory. Instead, share memory by communicating.

—Go Proverb

6 Cheney, Dave. “Clear Is Better than Clever.” *The Acme of Foolishness*, 19 July 2019. <https://oreil.ly/vJs0X>.

7 Hoare, C.A.R. “Communicating Sequential Processes.” *Communications of the ACM*, vol. 21, no. 8, Aug. 1978, pp. 666–77. <https://oreil.ly/CHiLt>.

8 At least among the “mainstream” languages, whatever that means.

Concurrency Is Not Parallelism

Computational concurrency and parallelism are often confused, which is understandable given that both concepts describe the state of having multiple processes executing during the same period of time. However, they are most definitely not the same thing:⁹

- *Parallelism* describes the simultaneous execution of multiple independent processes.
- *Concurrency* describes the composition of independently executing processes; it says nothing about when processes will execute.

Fast Builds

One of the primary motivations for the Go language was the maddeningly long build times for certain languages of the time,¹⁰ which even on Google's large compilation clusters often require minutes, or even hours, to complete. This eats away at development time and grinds down developer productivity. Given Go's primary purpose of enhancing rather than hindering developer productivity, long build times had to go.

The specifics of the Go compiler are beyond the scope of this book (and beyond my own expertise). Briefly, however, the Go language was designed to provide a model of software construction free of complex relationships, greatly simplifying dependency analysis and eliminating the need for C-style include files and libraries and the overhead that comes with them. As a result, most Go builds complete in seconds, or occasionally minutes, even on relatively humble hardware. For example, building all 1.8 million lines¹¹ of Go in Kubernetes v1.20.2 on a MacBook Pro with a 2.4 GHz 8-Core Intel i9 processor and 32 GB of RAM required about 45 seconds of real time:

```
mtitmus:~/workspace/kubernetes[MASTER]$ time make
```

```
real    0m45.309s
user    1m39.609s
sys     0m43.559s
```

⁹ Gerrand, Andrew. "Concurrency Is Not Parallelism." *The Go Blog*, 16 Jan. 2016. <https://oreil.ly/WXf4g>.

¹⁰ C++. We're talking about C++.

¹¹ Not counting comments; Openhub.net. "Kubernetes." *Open Hub*, Black Duck Software, Inc., 18 Jan. 2021. <https://oreil.ly/y5Rty>.

Not that this doesn't come without compromises. Any proposed change to the Go language is weighed in part against its likely effect on build times; some otherwise promising proposals have been rejected on the grounds that they would increase it.

Linguistic Stability

Go 1 was released in March of 2012, defining both the specification of the language and the specification of a set of core APIs. The natural consequence of this is an explicit promise, from the Go design team to the Go users, that programs written in Go 1 will continue to compile and run correctly, unchanged, for the lifetime of the Go 1 specification. That is, Go programs that work today can be expected to continue to work even under future “point” releases of Go 1 (Go 1.1, Go 1.2, etc.).¹²

This stands in stark contrast to many other languages, which sometimes add new features enthusiastically, gradually increasing the complexity of the language—and anything written in it—in time, leading to a once elegant language becoming a sprawling featurescape that's often exceedingly difficult to master.¹³

The Go Team considers this exceptional level of linguistic stability to be a vital feature of Go; it allows users to trust Go and to build on it. It allows libraries to be consumed and built upon with minimal hassle, and dramatically lowers the cost of updates, particularly for large projects and organizations. Importantly, it also allows the Go community to use Go and to learn from it; to spend time writing with the language rather than writing the language.

This is not to say that Go won't grow: both the APIs and the core language certainly *can* acquire new packages and features,¹⁴ and there are many proposals for exactly that,¹⁵ but not in a way that breaks existing Go 1 code.

That being said, it's quite possible¹⁶ that there will actually *never* be a Go 2. More likely, Go 1 will continue to be compatible indefinitely; and in the unlikely event that a breaking change is introduced, Go will provide a conversion utility, like the `go fix` command that was used during the move to Go 1.

12 The Go Team. “Go 1 and the Future of Go Programs.” *The Go Documentation*. <https://oreil.ly/Mqn0I>.

13 Anybody remember Java 1.1? I remember Java 1.1. Sure, we didn't have generics or autoboxing or enhanced for loops back then, but we were happy. Happy, I tell you.

14 I'm on team generics. Go, fightin' Parametric Polymorphics!

15 The Go Team. “Proposing Changes to Go.” *GitHub*, 7 Aug. 2019. <https://oreil.ly/foIYF>.

16 Pike, Rob. “Sydney Golang Meetup—Rob Pike—Go 2 Draft Specifications” (video). *YouTube*, 13 Nov. 2018. <https://oreil.ly/YmMAd>.

Memory Safety

The designers of Go have taken great pains to ensure that the language is free of the various bugs and security vulnerabilities—not to mention tedious bookkeeping—associated with direct memory access. Pointers are strictly typed and are always initialized to some value (even if that value is `nil`), and pointer arithmetic is explicitly disallowed. Built-in reference types like maps and channels, which are represented internally as pointers to mutable structures, are initialized by the `make` function. Simply put, Go neither needs nor allows the kind of manual memory management and manipulation that lower-level languages like C and C++ allow and require, and the subsequent gains with respect to complexity and memory safety can't be overstated.

For the programmer, the fact that Go is a garbage-collected language obviates the need to carefully track and free up memory for every allocated byte, eliminating a considerable bookkeeping burden from the programmer's shoulders. Life without `malloc` is liberating.

What's more, by eliminating manual memory management and manipulation—even pointer arithmetic—Go's designers have made it effectively immune to an entire class of memory errors and the security holes they can introduce. No memory leaks, no buffer overruns, no address space layout randomization. Nothing.

Of course, this simplicity and ease of development comes with some tradeoffs, and while Go's garbage collector is incredibly sophisticated, it does introduce some overhead. As such, Go can't compete with languages like C++ and Rust in pure raw execution speed. That said, as we see in the next section, Go still does pretty well for itself in that arena.

Performance

Confronted with the slow builds and tedious bookkeeping of the statically typed, compiled languages like C++ and Java, many programmers moved towards more dynamic, fluid languages like Python. While these languages are excellent for many things, they're also very inefficient relative to compiled languages like Go, C++, and Java.

Some of this is made quite clear in the benchmarks of [Table 2-1](#). Of course, benchmarks in general should be taken with a grain of salt, but some results are particularly striking.

Table 2-1. Relative benchmarks for common service languages (seconds)^a

	C++	Go	Java	NodeJS	Python3	Ruby	Rust
Fannkuch-Redux	8.08	8.28	11.00	11.89	367.49	1255.50	7.28
FASTA	0.78	1.20	1.20	2.02	39.10	31.29	0.74
K-Nucleotide	1.95	8.29	5.00	15.48	46.37	72.19	2.76
Mandlebrot	0.84	3.75	4.11	4.03	172.58	259.25	0.93
N-Body	4.09	6.38	6.75	8.36	586.17	253.50	3.31
Spectral norm	0.72	1.43	4.09	1.84	118.40	113.92	0.71

^a Gouy, Isaac. The Computer Language Benchmarks Game. 18 Jan. 2021. <https://oreil.ly/bQFjc>.

On inspection, it seems that the results can be clustered into three categories corresponding with the types of languages used to generate them:

- Compiled, strictly typed languages with manual memory management (C++, Rust)
- Compiled, strictly typed languages with garbage collection (Go, Java)
- Interpreted, dynamically typed languages (Python, Ruby)

These results suggest that, while the garbage-collected languages are generally slightly less performant than the ones with manual memory management, the differences don't appear to be great enough to matter except under the most demanding requirements.

The differences between the interpreted and compiled languages, however, is striking. At least in these examples, Python, the archetypical dynamic language, benchmarks about *ten to one hundred times slower* than most compiled languages. Of course, it can be argued that this is still perfectly adequate for many—if not most—purposes, but this is less true for cloud native applications, which often have to endure significant spikes in demand, ideally without having to rely on potentially costly upscaling.

Static Linking

By default, Go programs are compiled directly into native, statically linked executable binaries into which all necessary Go libraries and the Go runtime are copied. This produces slightly larger files (on the order of about 2MB for a “hello world”), but the resulting binary has no external language runtime to install,¹⁷ or external library

¹⁷ Take that, Java.

dependencies to upgrade or conflict,¹⁸ and can be easily distributed to users or deployed to a host without fear of suffering dependency or environmental conflicts.

This ability is particularly useful when you’re working with containers. Because Go binaries don’t require an external language runtime or even a distribution, they can be built into “scratch” images that don’t have parent images. The result is a very small (single digit MB) image with minimal deployment latency and data transfer overhead. These are very useful traits in an orchestration system like Kubernetes that may need to pull the image with some regularity.

Static Typing

Back in the early days of Go’s design, its authors had to make a choice: would it be *statically typed*, like C++ or Java, requiring variables to be explicitly defined before use, or *dynamically typed*, like Python, allowing programmers to assign values to variables without defining them and therefore generally faster to code? It wasn’t a particularly hard decision; it didn’t take very long. Static typing was the obvious choice, but it wasn’t arbitrary, or based on personal preference.¹⁹

First, type correctness for statically typed languages can be evaluated at compile time, making them far more performant (see [Table 2-1](#)).

The designers of Go understood that the time spent in development is only a fraction of a project’s total lifecycle, and that any gains in coding velocity with dynamically typed languages is more than made up for by the increased difficulty in debugging and maintaining such code. After all, what Python programmer hasn’t had their code crash because they tried to use a string as an integer?

Take the following Python code snippet, for example:

```
my_variable = 0

while my_variable < 10:
    my_variable = my_variable + 1    # Typo! Infinite loop!
```

See it yet? Keep trying if you don’t. It can take a second.

Any programmer can make this kind of subtle misspelling error, which just so happens to also produce perfectly valid, executable Python. These are just two trivial examples of an entire class of errors that Go will catch at compile time rather than (heaven forbid) in production, and generally closer in the code to the location where

¹⁸ Take that, Python.

¹⁹ Few arguments in programming generate as many snarky comments as Static versus Dynamic typing, except perhaps the Great Tabs versus Spaces Debate, on which Go’s unofficial position is “shut up, who cares?”

they are introduced. After all, it's well-understood that the earlier in the development cycle you catch a bug, the easier (read cheaper) it is to fix it.

Finally, I'll even assert something somewhat controversial: typed languages are more readable. Python is often lauded as especially readable with its forgiving nature and somewhat English-like syntax,²⁰ but what would you do if presented with the following Python function signature?

```
def send(message, recipient):
```

Is `message` a string? Is `recipient` an instance of some class described elsewhere? Yes, this could be improved with some documentation and a couple of reasonable defaults, but many of us have had to maintain enough code to know that that's a pretty distant star to wish on. Explicitly defined types can guide development and ease the mental burden of writing code by automatically tracking information the programmer would otherwise have to track mentally by serving as documentation for both the programmer and everybody who has to maintain their code.

Summary

If [Chapter 1](#) focused on what makes a *system* cloud native, then this chapter can be said to have focused on what makes a *language*, specifically Go, a good fit for building cloud native services.

However, while a cloud native system needs to be scalable, loosely coupled, resilient, manageable, and observable, a language for the cloud native era has to be able to do more than just build systems with those attributes. After all, with a bit of effort, pretty much any language can, technically, be used to build such systems. So what makes Go so special?

It can be argued that all of the features presented in this chapter directly or indirectly contribute to the cloud native attributes from the previous chapter. Concurrency and memory safety can be said to contribute to service scalability, and structural typing to allow loose coupling, for example. But while Go is the only mainstream language I know that puts all of these features in one place, are they *really* so novel?

Perhaps most conspicuous of Go's features are its baked-in—not bolted-on—concurrency features, which allow a programmer to fully and more safely utilize modern networking and multicore hardware. Goroutines and channels are wondrous, of course, and make it far easier to build resilient, highly concurrent networked services, but they're technically not unique if you consider some less common languages like Clojure or Crystal.

²⁰ I, too, have been lauded for my forgiving nature and somewhat English-like syntax.

I would assert that where Go really shines is in its faithful adherence to the principle of clarity over cleverness, which extends from an understanding that source code is written by humans for other humans.²¹ That it compiles into machine code is almost immaterial.

Go is designed to support the way people actually work together: in teams, which sometimes change membership, whose members also work on other things. In this environment, code clarity, the minimization of “tribal knowledge,” and the ability to rapidly iterate are critical. Go’s simplicity is often misunderstood and unappreciated, but it lets programmers focus on solving problems instead of struggling with the language.

In **Chapter 3**, we’ll review many of the specific features of the Go language, where we’ll get to see that simplicity up close.

²¹ Or for the same human after a few months of thinking about other things.