Chapter 1. Introduction to Cloud Native

What are cloud native applications? What makes them so appealing that the cloud native model is now considered not only for the cloud, but also for the edge? And, finally, how do you design and develop cloud native applications? These are all questions that will be answered throughout this book. But before we dive into the details on the what, why, and how, we want to provide a brief introduction to the cloud native world and some of the fundamental concepts and assumptions that are building the foundation for modern cloud native applications and environments.

Distributed Systems

One of the biggest hurdles that developers face when they build cloud native applications for the first time is that they must deal with services that are not on the same machine, and they need to deal with patterns that consider a network between the machines. Without even knowing it, they have entered the world of distributed systems. A *distributed system* is a system in which individual computers are connected through a network and appear as a single computer. Being able to distribute computing power across a bunch of machines is a great way to accomplish scalability, reliability, and better economics. For example, most cloud providers are using cheaper commodity hardware and solving common problems such as high availability and reliability through software-based solutions.

Fallacies of Distributed Systems

There are couple of incorrect or unfounded assumptions most developers and architects make when they enter the world of distributed systems. Peter Deutsch, a Fellow at Sun Microsystems, was identifying fallacies of distributed computing back in 1994, at a time when nobody thought about cloud computing. Because cloud native applications are, at their core, distributed systems, these fallacies still have validity today. Following is the list of the fallacies that Deutsch described, with their meanings applied to cloud native applications:

The network is reliable

Even in the cloud you cannot assume that the network is reliable. Because services are typically placed on different machines, you need to develop your software in a way that it accounts for potential network failures, which we discuss later in this book.

Latency is zero

Latency and bandwidth are often confused, but it is important to understand the difference. Latency is how much time goes by until data is received, whereas bandwidth indicates how much data can be transferred in a given window of time. Because latency has a big impact on user experience and performance, you should take care to do the following:

- Avoid frequent network calls and introducing chattiness to the network.
- Design your cloud native application in a way that the data is closest to your client by using caching, content delivery networks (CDNs), and multiregion deployments.
- Use publication/subscription (pub/sub) mechanisms to be notified that there is new data and store it locally to be immediately available. Chapter 3 covers messaging patterns such as pub/sub in more detail.

There is infinite bandwidth

Nowadays, network bandwidth does not seem to be a big issue, but new technologies and areas such as edge computing open up new scenarios that demand far more bandwidth. For example, it is predicted that a self-driving car will produce around 50 terabytes (TB) of data per day. This volume of data requires you to design your cloudnative application with bandwidth usage in mind. Domain-Driven Design (DDD) and data patterns such as Command Query Responsibility Segregation (CQRS) are very useful under such bandwidth-demanding circumstances. Chapter 4 and Chapter 6 cover how to work with data in cloud native applications in more detail.

The network is secure

Two things are often an afterthought for developers: diagnostics and security. The assumption that networks are secure can be fatal. As a developer or architect, you need to make security a priority of

your design; for example, by embracing a defense-in-depth approach.

The topology does not change

Pets versus cattle is a meme that gained popularity with the advent of containers. It means that you do not treat any machine as a known entity (pet) with its own set of properties, such as static IPs and so on. Instead, you treat machines as a member of a herd that has no special attributes. This concept is very important with cloud native applications. Because cloud environments are meant to provide elasticity, machines can be added and removed based on criteria such as resource consumption or requests per second.

There is one administrator

In traditional software development, it was quite common to have one person responsible for the environment, installing and upgrading the application, and so forth. Modern cloud architectures and DevOps methods have shifted the way software is built. A modern cloud native application is a composite of many services that need to work together in concert and that are developed by different teams. This makes it practically impossible for a single person to know and understand the application in its entirety, not to mention trying to fix a problem. Thus, you need to ensure that you have governance in place that makes it easy to troubleshoot issues. Throughout this book, we introduce you to important concepts such as *release management*, *decoupling*, and *logging and monitoring*. Chapter 5 provides a detailed look at common DevOps practices for cloud native applications.

Transport cost is zero

From a cloud native perspective, there are two ways to look at this one. First, transport happens over a network and network costs are not free with most cloud providers. Most cloud providers, for example, do not charge for data ingress, but do charge for data egress. The second way to look at this fallacy is that the cost for translating any payload into objects is not free. For example, serialization and deserialization are usually fairly expensive operations that you need to consider in addition to the latency of network calls.

The network is homogeneous

> This is almost not worth listing given that pretty much every developer and architect understands that there are different protocols that they must consider when building their applications.

As mentioned before, although these fallacies were documented a long time ago, they are still a good reminder of the incorrect assumptions people make when entering the cloud native world. Throughout this book, we teach you patterns and best practices that take all of the fallacies of distributed computing into account.

CAP Theorem

The CAP theorem is often mentioned in combination with distributed systems. The CAP theorem states that any networked shared-data system can have at most two of the following three desirable properties:

- Consistency (C) equivalent to having a single up-to-date copy of the data
- High availability (A) of that data (for updates)
- Tolerance to network partitions (P)

The reality is that you will always have network partitions (remember, "the network is reliable" is one of the fallacies of distributed computing). That leaves you with only two choices—you can optimize either for consistency or high availability. Many NoSQL databases such as Cassandra optimize for availability, whereas SQL-based systems that adhere to the principles of ACID (atomicity, consistency, isolation, and durability) optimize for consistency.

The Twelve-Factor App

In the early days of Infrastructure as a Service (IaaS) and Platform as a Service (PaaS), it quickly became obvious that the cloud required a new way of developing applications. For example, on-premises scaling was often done by scaling vertically, meaning adding more resources to a machine. Scaling in the cloud, on the other hand, is usually done horizontally, meaning adding more machines to distribute the load. This type of scaling requires stateless applications, and this is one of the

factors described by the *Twelve-Factor App manifesto*. The Twelve-Factor App methodology can be considered the foundation for cloud native applications and was first introduced by engineers at Heroku, derived from best practices for application development in the cloud. Cloud development has evolved since the introduction of the Twelve-Factor manifesto, but the principles still apply. Following are the 12 factors and their meaning for cloud native applications:

1. Codebase
   One codebase tracked in revision control; many deploys.

   There is only one codebase per application, but it can be deployed into many environments such as Dev, Test, and Prod. In cloud native architecture, this translates directly into one codebase per service or function, each having its own Continuous Integration/Continuous Deployment (CI/CD) flow.

2. Dependencies
   Explicitly declare and isolate dependencies.

   Declaring and isolating dependencies is an important aspect of cloud native development. Many issues arise due to missing dependencies or version mismatch of dependencies, which stem from environmental differences between the on-premises and cloud environments. In general, you should always use dependency managers for languages such as Maven or npm. Containers have drastically reduced dependency-based issues because all dependencies are packaged inside a container, and as such should be declared in the Dockerfile. Chef, Puppet, Ansible, and Terraform are great tools to manage and install system dependencies.

3. Configuration
   Store configuration in the environment.

   Configuration should be strictly separated from code. This allows you to easily apply configurations per environment. For example, you can have a test configuration file that stores all the connection strings and other information used in a test environment. If you want to deploy the same application to a production environment, you need only to replace the configuration. Many modern platforms support external configuration, whether it is

configuration maps with Kubernetes or managed configuration services in cloud environments.

4. Backing Services
   Treat backing services as attached resources.

   A backing service is defined as “any service the app consumed over the network as part of its normal operation.” In the case of cloud native applications, this might be a managed caching service or a Database as a Service (DbaaS) implementation. The recommendation here is to access those services through configuration settings stored in external configuration systems, which allows loose coupling, one of the principles that is also valid for cloud native applications.

5. Build, Release, Run
   Strictly separate build and run stages.
   As you will see in Chapter 5 on DevOps, it is recommended to aim for fully automated build and release stages using CI/CD practices.

6. Processes
   Execute the app in one or more stateless processes.

   As mentioned earlier, compute in the cloud should be stateless, meaning that data should only be saved outside the processes. This enables elasticity, which is one of the promises of cloud computing.

7. Data Isolation
   Each service manages its own data.

   This is one of the key tenets of microservices architectures, which is a common pattern in cloud native applications. Each service manages its own data, which can be accessed only through APIs, meaning that other services that are part of the application are not allowed to directly access the data of another service.

8. Concurrency
   Scale out via the process model.

   Improved scale and resource usage are two of the key benefits of cloud native applications, meaning that you can scale each service

or function independently and horizontally; thus, you'll achieve better resource usage.

9. Disposability
   Maximize robustness with fast startup and graceful shutdown.

   Containers and functions already satisfy this factor given that both provide fast startup times. One thing that is often neglected is to design for a crash or scale in scenario, meaning that the instance count of a function or a container is decreased, which is also captured in this factor.

10. Dev/Prod Parity
    Keep development, staging, and production as similar as possible. Containers allow you to package all of the dependencies of your service, which limits the issues with environment inconsistencies. There are scenarios that are a bit trickier, especially when you use managed services that are not available on-premises in your Dev environment. Chapter 5 looks at methods and techniques to keep your environments as consistent as possible.

11. Logs
    Treat logs as event streams.

    Logging is one of the most important tasks in a distributed system. There are so many moving parts and without a good logging strategy, you would be "flying blind" when the application is not behaving as expected. The Twelve-Factor manifesto states that you should treat logs as streams, routed to external systems.

12. Admin Processes
    Run admin and management tasks as one-off processes.

    This basically means that you should execute administrative and management tasks as short-lived processes. Both functions and containers are great tools for that.

Throughout the book you will recognize many of these factors because they are still very relevant for cloud native applications.

Availability and Service-Level Agreements

Most of the time, cloud native applications are composite applications that use compute, such as containers and functions, but also managed cloud services such as DbaaS, caching services, and/or identity services. What is not obvious is that your compound Service-Level Agreement (SLA) will never be as high as the highest availability of an individual service. SLAs are typically measured in uptime in a year, more commonly referred to as "number of nines." Table 1-1 shows a list of common availability percentages for cloud services and their corresponding downtimes.

Availability %	Downtime per year	Downtime per month
99%	3.65 days	7.20 hours
99.9%	8.76 hours	43.2 minutes
99.99%	52.56 minutes	4.32 minutes
99.999%	5.26 minutes	25.9 seconds
99.9999%	31.5 seconds	2.59 seconds

Table 1-1. Uptime percentages and service downtim

Following is an example of a compound SLA:

- Service 1 (99.95%) + Service 2 (99.90%): 0.9995 × 0.9990 = 0.9985005

The compound SLA is 99.85%.

Summary

Many developers struggle when starting to develop for the cloud. In a nutshell, developers are facing three major challenges: first, they need to understand distributed systems; second, they need to understand new technologies such as containers and functions; and third, they need to understand what patterns to use when building cloud native applications. Having some familiarity with the fundamentals, such as the fallacies of

distributed systems, the Twelve-Factor manifesto, and compound SLAs, will make the transition easier. This chapter introduced some of the fundamental concepts of cloud native, which enables you to better understand some of the architectural considerations and patterns discussed throughout the book.