Introduction and Learning Objectives

Introduction

Most new companies today run their business processes in the cloud. Newer startups and enterprises which realized early enough the direction technology was headed developed their applications for the cloud.

Not all companies were so fortunate. Some built their success decades ago on top of legacy technologies - monolithic applications with all components tightly coupled and almost impossible to separate, a nightmare to manage and deployed on super expensive hardware.

If working for an organization which refers to their main business application "the black box", where nobody knows what happens inside and most logic was never documented, leaving everyone clueless as to what and how things happen from the moment a request enters the application until a response comes out, and you are tasked to convert this business application into a cloud-ready set of applications, then you may be in for a very long and bumpy ride.

Learning Objectives

By the end of this chapter, you should be able to:

- Explain what a monolith is.
- Discuss the monolith's challenges in the cloud.
- Explain the concept of microservices.
- Discuss microservices advantages in the cloud.
- Describe the transformation path from a monolith to microservices.

From Monolith to Microservices

The Legacy Monolith

Although most enterprises believe that the cloud will be the new home for legacy apps, not all legacy apps are a fit for the cloud, at least not yet.

Moving an application to the cloud should be as easy as walking on the beach and collecting pebbles in a bucket and easily carry them wherever needed. A 1000-ton boulder, on the other hand, is not easy to carry at all. This boulder represents the monolith application - sedimented layers of features and redundant logic translated into thousands of lines of code, written in a single, not so modern programming language, based on outdated software architecture patterns and principles.

In time, the new features and improvements added to code complexity, making development more challenging - loading, compiling, and building times increase with every new update. However, there is some ease in administration as the application is running on a single server, ideally a Virtual Machine or a Mainframe.

A **monolith** has a rather expensive taste in hardware. Being a large, single piece of software which continuously grows, it has to run on a single system which has to satisfy its compute, memory, storage, and networking requirements. The hardware of such capacity is both complex and pricey.

Since the entire monolith application runs as a single process, the scaling of individual features of the monolith is almost impossible. It internally supports a hardcoded number of connections and operations. However, scaling the entire application means to manually deploy a new instance of the monolith on another server, typically behind a load balancing appliance - another pricey solution.

During upgrades, patches or migrations of the monolith application - downtimes occur and maintenance windows have to be planned as disruptions in service are expected to impact clients. While there are solutions to minimize downtimes to customers by setting up monolith applications in a highly available active/passive configuration, it may still be challenging for system engineers to keep all systems at the same patch level.

The Modern Microservice

Pebbles, as opposed to the 1000-ton boulder, are much easier to handle. They are carved out of the monolith, separated from one another, becoming distributed components each described by a set of specific characteristics. Once weighed all together, the pebbles make up the weight of the entire boulder. These pebbles represent loosely coupled microservices, each performing a specific business function. All the functions grouped together form the overall functionality of the original monolithic application. Pebbles are easy to select and group together based on color, size, shape, and require minimal effort to relocate when needed. Try relocating the 1000-ton boulder, effortlessly.

Microservices can be deployed individually on separate servers provisioned with fewer resources - only what is required by each service and the host system itself.

Microservices-based architecture is aligned with Event-driven Architecture and Service-Oriented Architecture (SOA) principles, where complex applications are composed of small independent processes which communicate with each other through APIs over a network. APIs allow access by other internal services of the same application or external, third-party services and applications.

Each microservice is developed and written in a modern programming language, selected to be the best suitable for the type of service and its business function. This offers a great deal of flexibility when matching microservices with specific hardware when required, allowing deployments on inexpensive commodity hardware.

Although the distributed nature of microservices adds complexity to the architecture, one of the greatest benefits of microservices is scalability. With the overall application becoming modular, each microservice can be scaled individually, either manually or automated through demand-based autoscaling.

Seamless upgrades and patching processes are other benefits of microservices architecture. There is virtually no downtime and no service disruption to clients because upgrades are rolled out seamlessly - one service at a time, rather than having to re-compile, re-build and re-start an entire monolithic application. As a result, businesses are able to develop and roll-out new features and updates a lot faster, in an agile approach, having separate teams focusing on separate features, thus being more productive and cost-effective.

Refactoring

Newer, more modern enterprises possess the knowledge and technology to build cloud-native applications that power their business.

Unfortunately, that is not the case for established enterprises running on legacy monolithic applications. Some have tried to run monoliths as microservices, and as one would expect, it did not work very well. The lessons learned were that a monolithic size multi-process application cannot run as a microservice and that other options had to be explored. The next natural step in the path of the monolith to microservices transition was refactoring. However, migrating a decades-old application to the cloud through refactoring poses serious challenges and the enterprise faces the refactoring approach dilemma: a "Big-bang" approach or an incremental refactoring.

A so-called "Big-bang" approach focuses all efforts with the refactoring of the monolith, postponing the development and implementation of any new features - essentially delaying progress and possibly, in the process, even breaking the core of the business, the monolith.

An incremental refactoring approach guarantees that new features are developed and implemented as modern microservices which are able to communicate with the monolith through APIs, without appending to the monolith's code. In the meantime, features are refactored out of the monolith which slowly fades away while all, or most its functionality is modernized into microservices. This incremental approach offers a gradual transition from a legacy monolith to modern microservices architecture and allows for phased migration of application features into the cloud.

Once an enterprise chose the refactoring path, there are other considerations in the process. Which business components to separate from the monolith to become distributed microservices, how to decouple the databases from the application to separate data complexity from application logic, and how to test the new microservices and their dependencies, are just a few of the decisions an enterprise is faced with during refactoring.

The refactoring phase slowly transforms the monolith into a cloud-native application which takes full advantage of cloud features, by coding in new programming languages and applying modern architectural patterns. Through refactoring, a legacy monolith application receives a second chance at life - to live on as a modular system adapted to fully integrate with today's fast-paced cloud automation tools and services.

Challenges

The refactoring path from a monolith to microservices is not smooth and without challenges. Not all monoliths are perfect candidates for refactoring, while some may not even "survive" such a modernization phase. When deciding whether a monolith is a possible candidate for refactoring, there are many possible issues to consider.

When considering a legacy Mainframe based system, written in older programming languages - Cobol or Assembler, it may be more economical to just re-build it from the ground up as a cloud-native application. A poorly designed legacy application should be re-designed and re-built from scratch following modern architectural patterns for microservices and even containers. Applications tightly coupled with data stores are also poor candidates for refactoring.

Once the monolith survived the refactoring phase, the next challenge is to design mechanisms or find suitable tools to keep alive all the decoupled modules to ensure application resiliency as a whole.

Choosing runtimes may be another challenge. If deploying many modules on a single physical or virtual server, chances are that different libraries and runtime environment may conflict with one another causing errors and failures. This forces deployments of single modules per servers in order to separate their dependencies - not an economical way of resource management, and no real segregation of libraries and runtimes, as each server also has an underlying Operating System running with its libraries, thus consuming server resources - at times the OS consuming more resources than the application module itself.

Ultimately application containers came along, providing encapsulated lightweight runtime environments for application modules. Containers promised consistent software environments for developers, testers, all the way from Development to Production. Wide support of containers ensured application portability from physical bare-metal to Virtual Machines, but this time with multiple applications deployed on the very same server, each running in their own execution environments isolated from one another, thus avoiding conflicts, errors, and failures. Other features of containerized application environments are higher server utilization, individual module scalability, flexibility, interoperability and easy integration with automation tools.

Success Stories

Although a challenging process, moving from monoliths to microservices is a rewarding journey especially once a business starts to see growth and success delivered by a refactored application system. Below we are listing only a handful of the success stories of companies which rose to the challenge to modernize their monolith business applications. A detailed list of success stories is available at the Kubernetes website: Kubernetes User Case Studies.

- AppDirect - an end-to-end commerce platform provider, started from a complex monolith application and through refactoring was able to retain limited functionality monoliths receiving very few commits, but all new features implemented as containerized microservices.
- box - a cloud storage solutions provider, started from a complex monolith architecture and through refactoring was able to decompose it into microservices.
- Crowdfire - a content management solutions provider, successfully broke down their initial monolith into microservices.
- GolfNow - a technology and services provider, decided to break their monoliths apart into containerized microservices.
- Pinterest - a social media services provider, started the refactoring process by first migrating their monolith API.

Introduction and Learning Objectives

Introduction

With container images, we confine the application code, its runtime, and all of its dependencies in a pre-defined format. And, with container runtimes like **runC, containerd,** or **rkt** we can use those pre-packaged images, to create one or more containers. All of these runtimes are good at running containers on a single host. But, in practice, we would like to have a fault-tolerant and scalable solution, which can be achieved by creating a single **controller/management** unit, after connecting multiple nodes together. This controller/management unit is generally referred to as a **container orchestrator.**

In this chapter, we will explore why we should use container orchestrators, different implementations of container orchestrators, and where to deploy them.

Learning Objectives

By the end of this chapter, you should be able to:

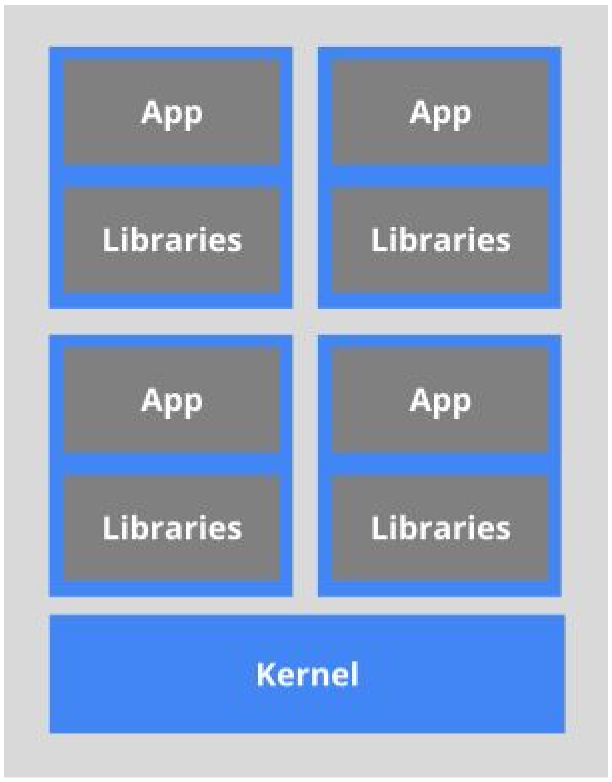
- Define the concept of container orchestration.
- Explain the reasons for doing container orchestration.
- Discuss different container orchestration options.
- Discuss different container orchestration deployment options.

Container Orchestration

What Are Containers?

Before we dive into container orchestration, let's review first what containers are.

Containers are application-centric methods to deliver high-performing, scalable applications on any infrastructure of your choice. Containers are best suited to deliver microservices by providing portable, isolated virtual environments for applications to run without interference from other running applications.



Containers

Microservices are lightweight applications written in various modern programming languages, with specific dependencies, libraries and environmental requirements. To ensure that an application has everything it needs to run successfully it is packaged together with its dependencies.

Containers encapsulate microservices and their dependencies but do not run them directly. Containers run container images.

A **container image** bundles the application along with its runtime and dependencies, and a container is deployed from the container image offering an isolated executable environment for the application. Containers can be deployed from a specific image on many platforms, such as workstations, Virtual Machines, public cloud, etc.

What Is Container Orchestration?

In Development (Dev) environments, running containers on a single host for development and testing of applications may be an option. However, when migrating to Quality Assurance (QA) and Production (Prod) environments, that is no longer a viable option because the applications and services need to meet specific requirements:

- Fault-tolerance
- On-demand scalability
- Optimal resource usage
- Auto-discovery to automatically discover and communicate with each other
- Accessibility from the outside world
- Seamless updates/rollbacks without any downtime.

Container orchestrators are tools which group systems together to form clusters where containers' deployment and management is automated at scale while meeting the requirements mentioned above.

Container Orchestrators

With enterprises containerizing their applications and moving them to the cloud, there is a growing demand for container orchestration solutions. While there are many solutions available, some are mere re-distributions of well-established container orchestration tools, enriched with features and, sometimes, with certain limitations in flexibility.

Although not exhaustive, the list below provides a few different container orchestration tools and services available today:

- **Amazon Elastic Container Service**
  Amazon Elastic Container Service (ECS) is a hosted service provided by Amazon Web Services (AWS) to run Docker containers at scale on its infrastructure.

- **Azure Container Instances**
  Azure Container Instance (ACI) is a basic container orchestration service provided by Microsoft Azure.

- **Azure Service Fabric**
  Azure Service Fabric is an open source container orchestrator provided by Microsoft Azure.

- **Kubernetes**
  Kubernetes is an open source orchestration tool, started by Google, part of the Cloud Native Computing Foundation (CNCF) project.

- **Marathon**
  Marathon is a framework to run containers at scale on Apache Mesos.

- **Nomad**
  Nomad is the container orchestrator provided by HashiCorp.

- **Docker Swarm**
  Docker Swarm is a container orchestrator provided by Docker, Inc. It is part of Docker Engine.

We have explored different container orchestrators in another edX MOOC, *Introduction to Cloud Infrastructure Technologies (LFS151x).* We highly recommend that you take *LFS151x.*

Why Use Container Orchestrators?

Although we can manually maintain a couple of containers or write scripts for dozens of containers, orchestrators make things much easier for operators especially when it comes to managing hundreds and thousands of containers running on a global infrastructure.

Most container orchestrators can:

- Group hosts together while creating a cluster
- Schedule containers to run on hosts in the cluster based on resources availability
- Enable containers in a cluster to communicate with each other regardless of the host they are deployed to in the cluster
- Bind containers and storage resources

- Group sets of similar containers and bind them to load-balancing constructs to simplify access to containerized applications by creating a level of abstraction between the containers and the user
- Manage and optimize resource usage
- Allow for implementation of policies to secure access to applications running inside containers.

With all these configurable yet flexible features, container orchestrators are an obvious choice when it comes to managing containerized applications at scale. In this course, we will explore **Kubernetes**, one of the most in-demand container orchestration tools available today.

Where to Deploy Container Orchestrators?

Most container orchestrators can be deployed on the infrastructure of our choice - on bare metal, Virtual Machines, on-premise, or the public cloud. Kubernetes, for example, can be deployed on a workstation, with or without a local hypervisor such as Oracle VirtualBox, inside a company's data center, in the cloud on AWS Elastic Compute Cloud (EC2) instances, Google Compute Engine (GCE) VMs, DigitalOcean Droplets, OpenStack, etc.

There are turnkey solutions which allow Kubernetes clusters to be installed, with only a few commands, on top of cloud Infrastructures-as-a-Service, such as GCE, AWS EC2, Docker Enterprise, IBM Cloud, Rancher, VMware, Pivotal, and multi-cloud solutions through IBM Cloud Private and StackPointCloud.

Last but not least, there is the managed container orchestration as-a-Service, more specifically the managed Kubernetes as-a-Service solution, offered and hosted by the major cloud providers, such as Google Kubernetes Engine (GKE), Amazon Elastic Container Service for Kubernetes (Amazon EKS), Azure Kubernetes Service (AKS), IBM Cloud Kubernetes Service, DigitalOcean Kubernetes, Oracle Container Engine for Kubernetes, etc. These shall be explored in one of the later chapters.

Introduction and Learning Objectives

Introduction

In this chapter, we will explain what **Kubernetes** is, its features, and the reasons why you should use it. We will explore the evolution of Kubernetes from **Borg**, which is a cluster manager created by Google.

We will also talk about the **Cloud Native Computing Foundation (CNCF)**, which currently hosts the Kubernetes project, along with other cloud-native projects, like Prometheus, Fluentd, rkt, containerd, etc.

Learning Objectives

By the end of this chapter, you should be able to:

- Define Kubernetes.
- Explain the reasons for using Kubernetes.
- Discuss the features of Kubernetes.
- Discuss the evolution of Kubernetes from Borg.
- Explain what the Cloud Native Computing Foundation does.

Kubernetes

What Is Kubernetes?

According to the Kubernetes website,

"Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications."

Kubernetes comes from the Greek word **κυβερνήτης**, which means *helmsman or ship pilot.* With this analogy in mind, we can think of Kubernetes as the pilot on a ship of containers.

Kubernetes is also referred to as **k8s**, as there are 8 characters between k and s.

Kubernetes is highly inspired by the Google Borg system, a container orchestrator for its global operations for more than a decade. It is an open source project written in the Go language and licensed under the Apache License, Version 2.0.

Kubernetes was started by Google and, with its v1.0 release in July 2015, Google donated it to the Cloud Native Computing Foundation (CNCF). We will talk more about CNCF later in this chapter.

New Kubernetes versions are released in 3 months cycles. The current stable version is 1.14 (as of May 2019).

From Borg to Kubernetes

According to the abstract of Google's Borg paper, published in 2015,

"Google's Borg system is a cluster manager that runs hundreds of thousands of jobs, from many thousands of different applications, across a number of clusters each with up to tens of thousands of machines".

For more than a decade, Borg has been Google's secret, running its worldwide containerized workloads in production. Services we use from Google, such as Gmail, Drive, Maps, Docs, etc., they are all serviced using Borg.

Some of the initial authors of Kubernetes were Google employees who have used Borg and developed it in the past. They poured in their valuable knowledge and experience while designing Kubernetes. Some of the features/objects of Kubernetes that can be traced back to Borg, or to lessons learned from it, are:

- API servers
- Pods
- IP-per-Pod
- Services
- Labels.

We will explore all of them, and more, in this course.

Kubernetes Features I

Kubernetes offers a very rich set of features for container orchestration. Some of its fully supported features are:

- **Automatic bin packing**
  Kubernetes automatically schedules containers based on resource needs and constraints, to maximize utilization without sacrificing availability.

- **Self-healing**
  Kubernetes automatically replaces and reschedules containers from failed nodes. It kills and restarts containers unresponsive to health checks, based on existing rules/policy. It also prevents traffic from being routed to unresponsive containers.

- **Horizontal scaling**
  With Kubernetes applications are scaled manually or automatically based on CPU or custom metrics utilization.

- **Service discovery and Load balancing**
  Containers receive their own IP addresses from Kubernetes, while it assigns a single Domain Name System (DNS) name to a set of containers to aid in load-balancing requests across the containers of the set.

Kubernetes Features II

Some other fully supported Kubernetes features are:

- **Automated rollouts and rollbacks**
  Kubernetes seamlessly rolls out and rolls back application updates and configuration changes, constantly monitoring the application's health to prevent any downtime.

- **Secret and configuration management**
  Kubernetes manages secrets and configuration details for an application separately from the container image, in order to avoid a re-build of the respective image. Secrets consist of confidential information passed to the application without revealing the sensitive content to the stack configuration, like on GitHub.

- **Storage orchestration**
  Kubernetes automatically mounts software-defined storage (SDS) solutions to containers from local storage, external cloud providers, or network storage systems.

- **Batch execution**
  Kubernetes supports batch execution, long-running jobs, and replaces failed containers.

There are many other features besides the ones we just mentioned, and they are currently in alpha/beta phase. They will add great value to any Kubernetes deployment once they become stable features. For example, support for role-based access control (RBAC) is stable as of the Kubernetes 1.8 release.

Why Use Kubernetes?

In addition to its fully-supported features, Kubernetes is also portable and extensible. It can be deployed in many environments such as local or remote Virtual Machines, bare metal, or in public/private/hybrid/multi-cloud setups. It supports and it is supported by many 3rd party open source tools which enhance Kubernetes' capabilities and provide a feature-rich experience to its users.

Kubernetes' architecture is modular and pluggable. Not only that it orchestrates modular, decoupled microservices type applications, but also its architecture follows decoupled microservices patterns. Kubernetes' functionality can be extended by writing custom resources, operators, custom APIs, scheduling rules or plugins.

For a successful open source project, the community is as important as having great code. Kubernetes is supported by a thriving community across the world. It has more than 2,000 contributors, who, over time, have pushed over 77,000 commits. There are meet-up groups in different cities and countries which meet regularly to discuss Kubernetes and its ecosystem. There are *Special Interest Groups* (SIGs), which focus on special topics, such as scaling, bare metal, networking, etc. We will talk more about them in our last chapter, *Kubernetes Communities.*

Kubernetes Users

With just a few years since its debut, many enterprises of various sizes run their workloads using Kubernetes. It is a solution for workload management in banking, education, finance and investments, gaming, information technology, media and streaming, online retail, ridesharing, telecommunications, and many other industries. There are numerous user case studies and success stories on the Kubernetes website:

- BlaBlaCar
- BlackRock
- Box
- eBay
- Haufe Group
- Huawei
- IBM
- ING
- Nokia
- Pearson
- Wikimedia
- And many more.

Cloud Native Computing Foundation (CNCF)

The Cloud Native Computing Foundation (CNCF) is one of the projects hosted by the Linux Foundation. CNCF aims to accelerate the adoption of containers, microservices, and cloud-native applications.

CNCF hosts a multitude of projects, with more to be added in the future. CNCF provides resources to each of the projects, but, at the same time, each project continues to operate independently under its pre-existing governance structure and with its existing maintainers. Projects within CNCF are categorized based on achieved status: Sandbox, Incubating, and Graduated. At the time this course was created, there were six projects with Graduated status and many more still Incubating and in the Sandbox.

Graduated projects:

- Kubernetes for container orchestration
- Prometheus for monitoring
- Envoy for service mesh
- CoreDNS for service discovery
- containerd for container runtime
- Fluentd for logging

Incubating projects:

- rkt and CRI-O for container runtime
- Linkerd for service mesh
- etcd for key/value store
- gRPC for remote procedure call (RPC)
- CNI for networking API
- Harbor for registry
- Helm for package management
- Rook and Vitess for cloud-native storage
- Notary for security
- TUF for software updates
- NATS for messaging
- Jaeger and OpenTracing for distributed tracing
- Open Policy Agent for policy.

There are many projects in the CNCF Sandbox geared towards metrics, monitoring, identity, scripting, serverless, nodeless, edge, etc.

As we can see, the CNCF projects cover the entire lifecycle of a cloud-native application, from its execution using container runtimes, to its monitoring and logging. This is very important to meet the CNCF goal.

CNCF and Kubernetes

For Kubernetes, the Cloud Native Computing Foundation:

- Provides a neutral home for the Kubernetes trademark and enforces proper usage
- Provides license scanning of core and vendor code
- Offers legal guidance on patent and copyright issues
- Creates open source learning curriculum, training, and certification for both Kubernetes administrators and application developers
- Manages a software conformance working group
- Actively markets Kubernetes
- Supports ad hoc activities
- Funds conferences and meetup events.