

Introduction to Microservices

Kannan Manoharan B.Tech., M.E., (Ph.D.),
VMware Certified Instructor & Microsoft Certified Trainer
5x VCP (DCV, NV, AM, CMA, DTM)
Microsoft Certified Azure Administrator

Course Contents

- **Chapter 1: Overview of Microservices Architecture**
 - Explore the ideal software development practice
 - Understandings of microservices and differences between monolithic to microservices
 - Design principles of microservices with developers and admin in mind
- **Chapter 2: Microservices Technologies and Languages**
 - Understandings of microservices architecture
 - Overview of familiar Technologies and languages
 - Learn about popular technologies that enable the development, deployment, and support of microservices
 - **Lab 1(Optional): Microservices App Development in Spring Framework**
- **Chapter 3: Containerization using Docker**
 - Overview about the containerization
 - Learn about the Docker Container Runtime Engine
 - **Lab 2: Building and testing the container images**

Course Contents

- **Chapter 4: Kubernetes Cluster and its objects**
 - Learn about the architecture and benefits of Kubernetes Orchestration
 - Learn about the cluster formation of Kubernetes
 - Deep dive on the Kubernetes objects
 - **Lab 3: Setting up the Kubernetes Cluster**
 - **Lab 4: Running Kubectl Commands**
- **Chapter 5: Integration and Decomposition Patterns**
 - Learn about the integration and inter-communication patterns
 - Learn about the decomposition patterns from monolith to microservices
 - **Lab 5: Exploring microservices using Kubernetes objects and provide communications**
- **Chapter 6: Data Management, Transactional and Deployment Patterns**
 - Understanding about the database patterns
 - Learn about how to deploy microservices and its patterns
 - Review some of the distributed transactional patterns for microservices
 - **Lab 6: Deployment management**

Course Contents

- **Chapter 7: GIT**
 - Introducing GIT and how to get started
 - Learn about the GIT Devops Methodology
 - **Lab 7: Installing Git**
 - **Lab 8: Working Git operations**
- **Chapter 8: Jenkins**
 - Introducing the Jenkins and Jenkins CI/CD Pipeline
 - Learn how to install, manage and secure the Jenkins
 - Learn about diagnosing the errors and troubleshooting
 - **Lab 9: Installing Jenkins**
 - **Lab 10: Implementation of CI/CD pipeline using Jenkins**
- **Chapter 9: Istio Fundamentals**
 - Introducing service mesh with its architecture and features
 - Learn about the installation procedure of Istio for microservices
 - Learn how to reviewing the service requirements and adding services to the mesh
 - **Lab 11: Installing Istio Service Mesh**

Course Contents

- **Chapter 10: Securing Communication within Istio and controlling traffic**
 - Learn about the Istio Security
 - Review how to enable the mTLS communication between services and secure inbound traffic
 - Learn about Canary Testing, resiliency and chaos testing
- **Chapter 11: Monitoring Patterns**
 - Learn about the monitoring and observability patterns of microservices
 - Understand what is Prometheus and Grafana.
 - Learn how to work with Prometheus and Grafana for monitoring and alerting
 - **Lab 12: Installing Prometheus and Grafana and monitoring microservices**

Chapter 1

The Emergence of Microservice Architecture

Objectives

Chapter 1: Overview of Microservices Architecture

- Explore the ideal software development practice
- Understandings of microservices and differences between monolithic to microservices
- Design principles of microservices with developers and admin in mind

Monolithic - Overview

- The term *monolithic* applies to tightly integrated applications where it is hard to change one function without also recoding other parts of the application.
- Components in a monolithic application might be distributed among many machines, but they remain highly dependent on one another.
- Not only does the addition of a new feature have ripple effects throughout the code, but deploying the change requires retesting and redeploying the entire application.
- These upgrades can be labor-intensive and hazardous, particularly when an application has hundreds of thousands or even millions of users.

Monolithic Application Architecture

- The term "monolithic" comes from the Greek terms *monos* and *lithos*, together meaning a large stone block. The meaning in the context of IT for monolithic software architecture characterizes the uniformity, rigidity, and massiveness of the software architecture.
- A monolithic code base framework is often written using a single programming language, and all business logic is contained in a single repository.

Monolithic Application Architecture

- Typical monolithic applications consist of a single shared database, which can be accessed by different components. The various modules are used to solve each piece of business logic. But all business logic is wrapped up in a single API and is exposed to the frontend. The **user interface (UI)** of an application is used to access and preview backend data to the user. Here's a visual representation of the flow:



Monolithic Application Architecture

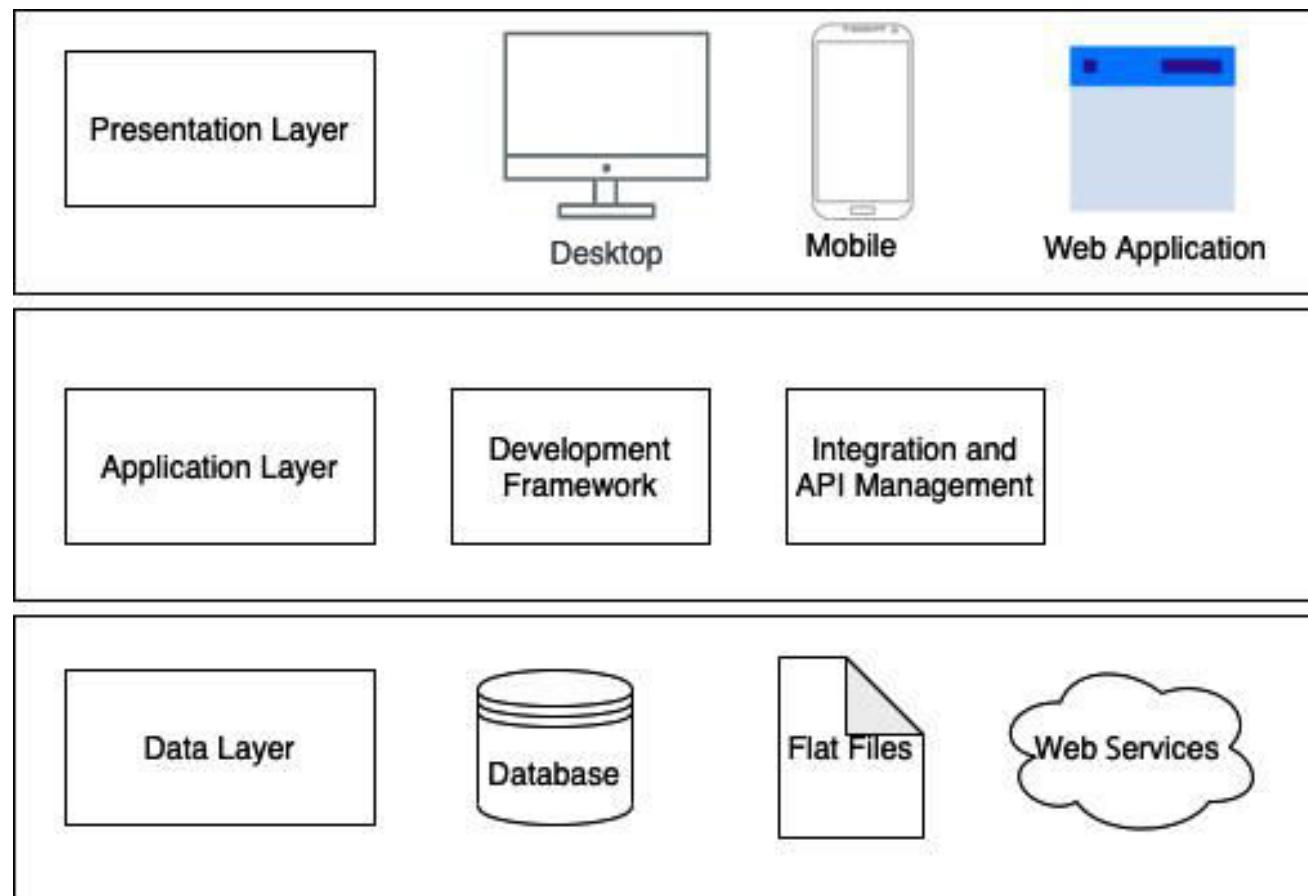
In this architecture, all the processes are tightly interconnected and run in a single service. This means that if you make a change in a single process, you may need to update the entire system. This approach is simple to develop and deploy and is excellent for small, simple applications.

An example of this might be a traditional eCommerce web application like an initial version of Amazon, where all functionalities like user interface, product catalog management, shopping cart, payment, and customer management are part of a single application. If a change is needed in one area, such as the payment system, the entire application must be tested and redeployed.

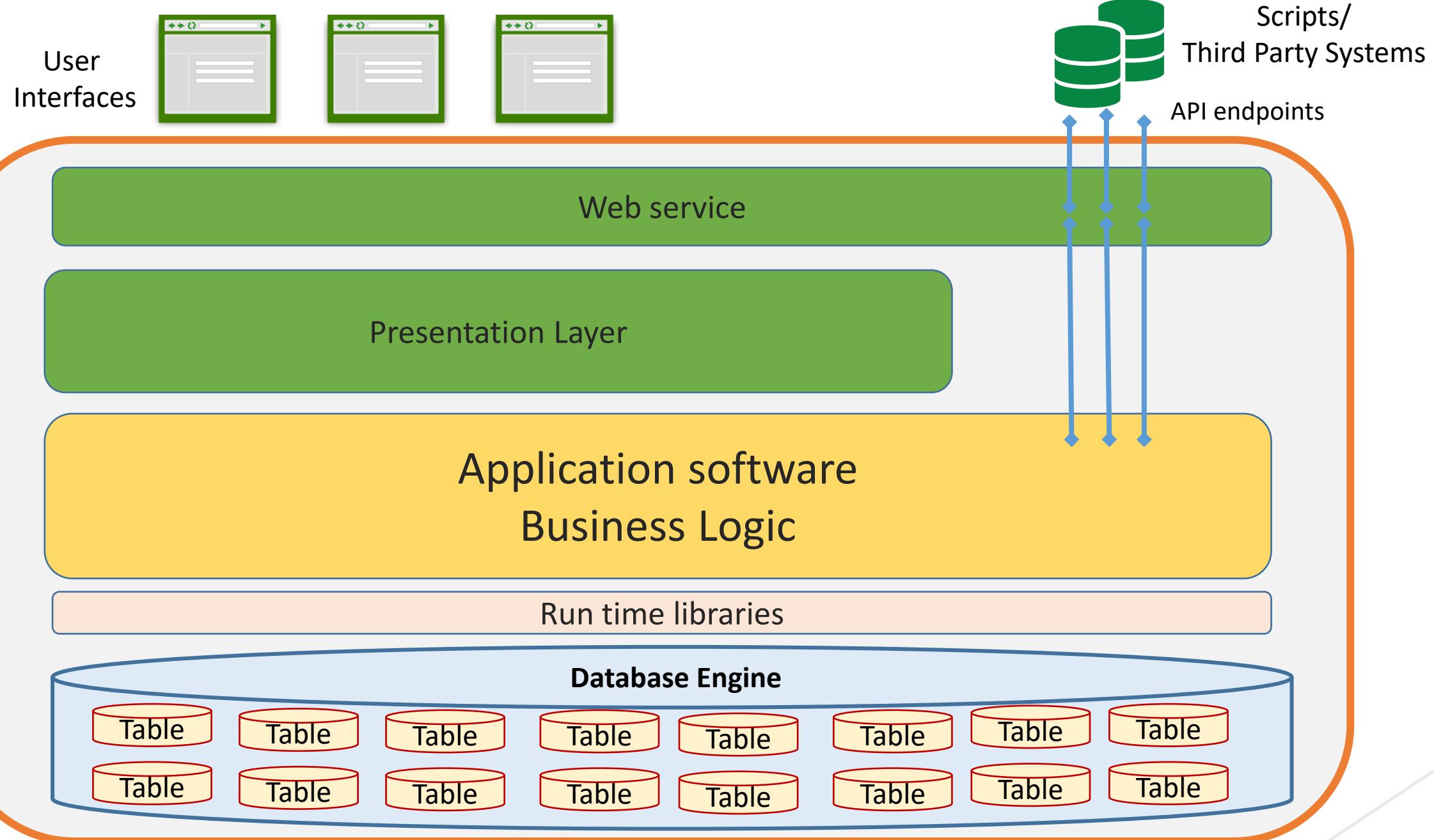
N-tier architecture in Monolithic Apps

- The N-tier architecture allows developers to build applications on several levels. The simplest type of N-tier architecture is the one-tier architecture.
- In this type of architecture, all programming logic, interfaces, and databases reside in a single computer.
- As soon as developers understood the value of decoupling databases from an application, they invented the two-tier architecture, where databases were stored on a separate server.
- This allowed developers to build applications that allow multiple clients to use a single database and provide distributed services over a network.

N-tier architecture in Monolithic Apps



Monolithic Application Conceptual Model



Service Oriented Architecture

- Service-oriented architecture was largely created as a response to traditional, monolithic approaches to building applications.
- SOA breaks up the components required for applications into separate service modules that communicate with one another to meet specific business objectives.
- Each module is considerably smaller than a monolithic application, and can be deployed to serve different purposes in an enterprise. Additionally, SOA is delivered via the cloud and can include services for infrastructure, platforms, and applications.

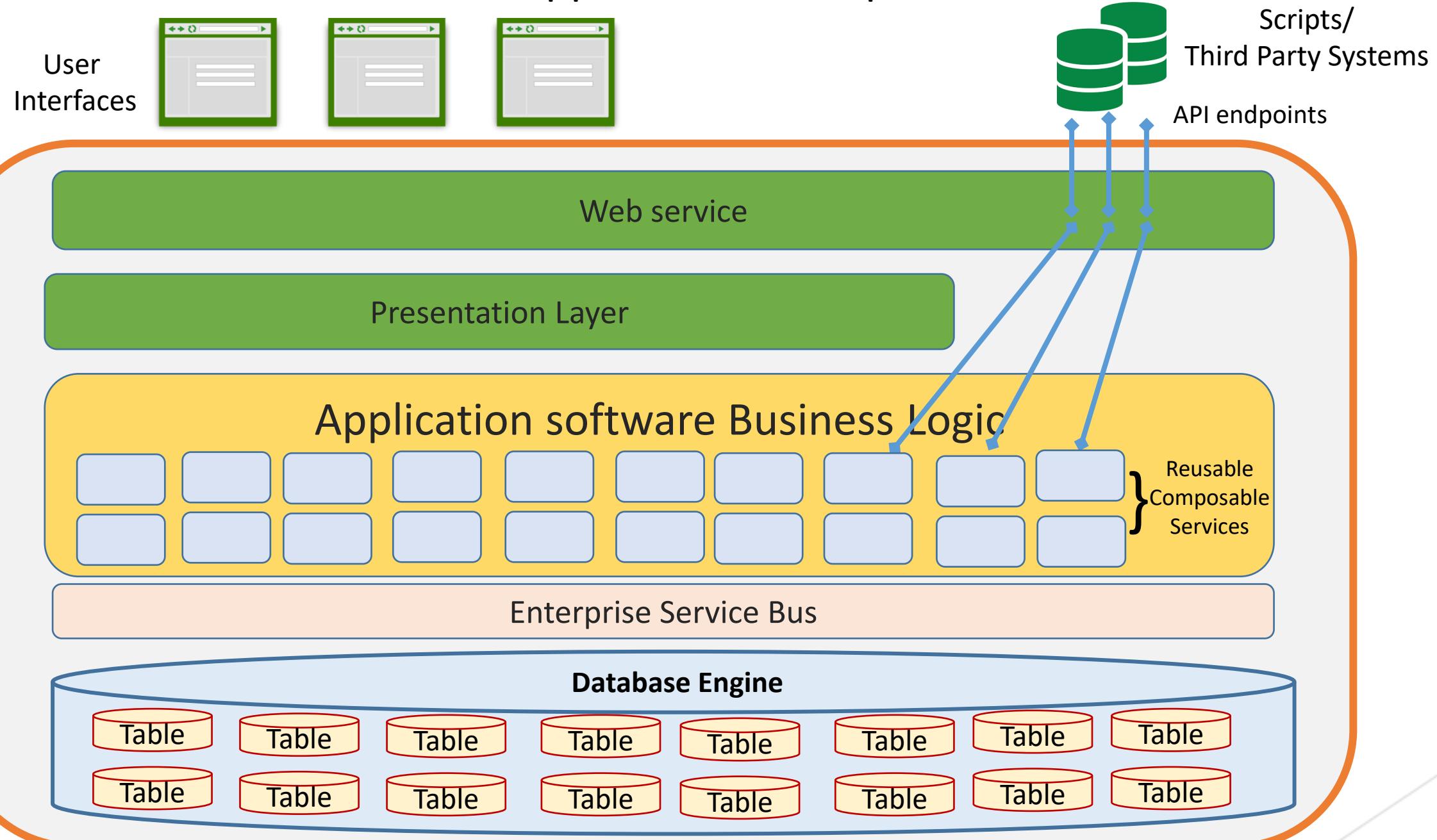
Service Oriented Architecture

- Each service in an SOA embodies the code and *data* required to execute a complete, discrete business function (e.g. checking a customer's credit, calculating a monthly loan payment, or processing a mortgage application).
- The service interfaces provide loose coupling, meaning they can be called with little or no knowledge of how the *service* is implemented underneath, reducing the dependencies between applications.

Service Oriented Architecture

- SOA's two main roles are as a service provider and a service consumer. Its service provider layer includes the different services involved in SOA, while the consumer layer operates as the user interface.
- SOA delivers four different types of services:
 - **Functional services** are used for business operations
 - **Enterprise services** implement the functionality
 - **Application services** are specific for developing and deploying apps
 - **Infrastructure services** are for non-functional processes such as security and authentication

Monolithic Application: Enterprise SOA Model



Enterprise Service Bus (ESB)

- It is possible to implement an SOA without an ESB, but this would be equivalent to just having a bunch of services.
- Each application owner would need to directly connect to any service it needs and perform the necessary data transformations to meet each of the service interfaces.
- This is a lot of work (even if the interfaces are reusable) and creates a significant maintenance challenges in the future as each connection is point to point.

Microservices

- Microservices is an architecture style, in which large complex software applications are composed of one or more services.
- Microservice can be deployed independently of one another and are loosely coupled.
- Each of these microservices focuses on completing one task only and does that one task really well. In all cases, that one task represents a small business capability.

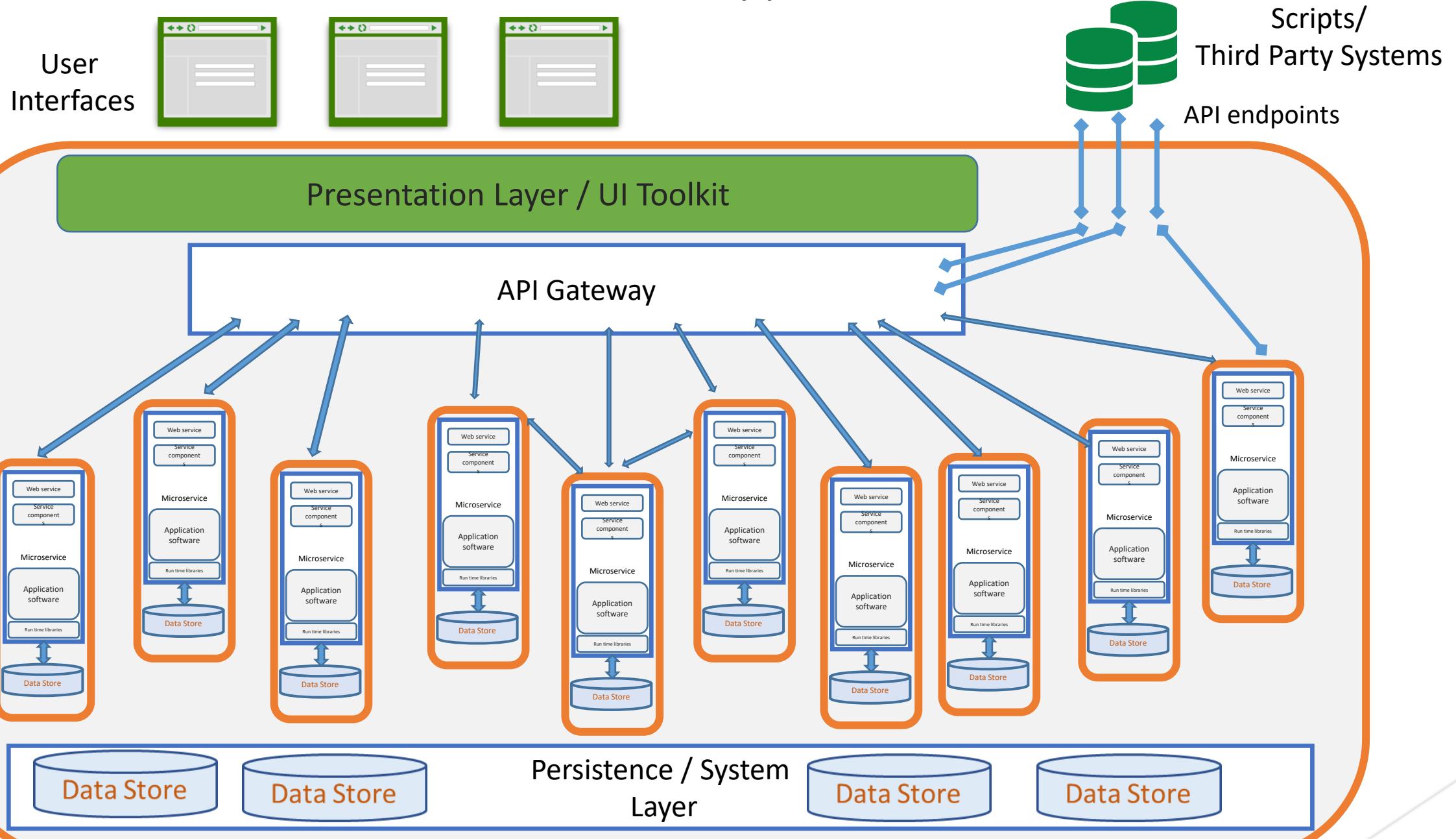
Microservices

- Microservices can be developed in any programming language. They communicate with each other using language-neutral application programming interfaces (APIs) such as Representational State Transfer (REST).
- Microservices also have a bounded context. They don't need to know anything about underlying implementation or architecture of other microservices.

Microservices Architecture

- Microservice architecture (MSA) is an approach to building software systems that decomposes business domain models into smaller, consistent, bounded-contexts implemented by services.
- These services are isolated and autonomous yet communicate to provide some piece of business functionality.
- Microservices are typically implemented and operated by small teams with enough autonomy that each team and service can change its internal implementation details (including replacing it outright!) with minimal impact across the rest of the system.

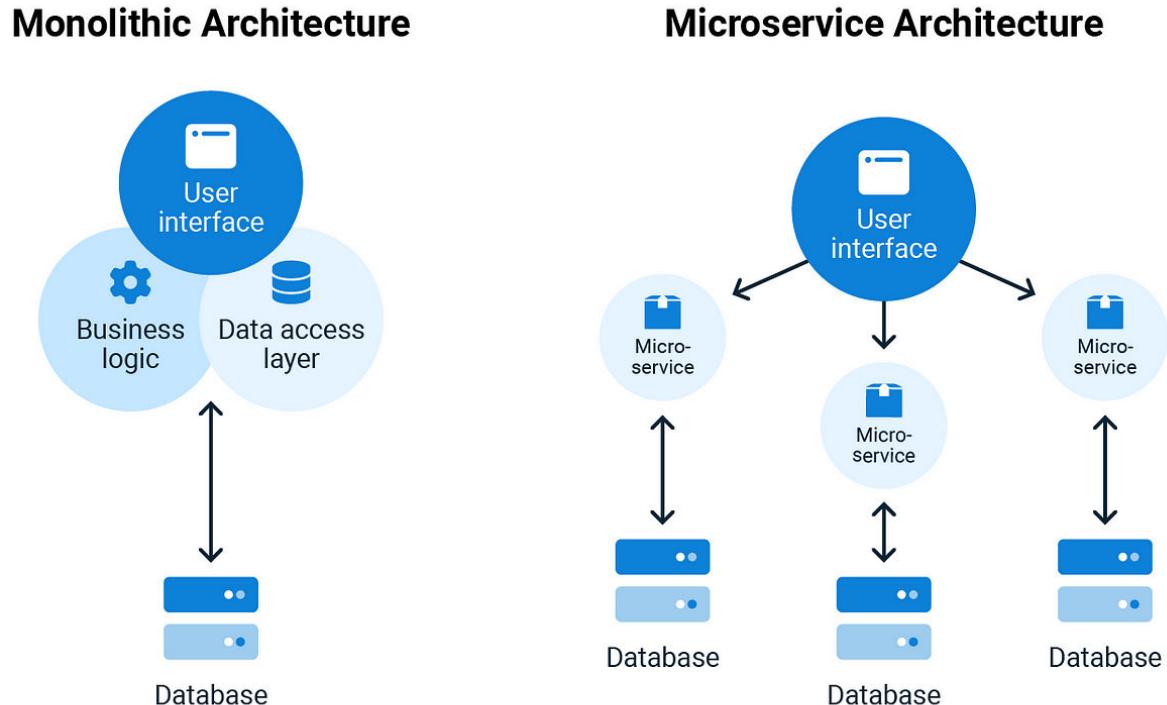
Microservices-based Application



Benefits of transition from Monolithic to Microservices

The transition from monolithic applications to microservices can bring many benefits, including:

- 1.Improved scalability
- 2.Increased flexibility
- 3.Faster delivery time
- 4.Better resource utilization
- 5.Technology diversity



Monolithic vs Microservices

Category	Monolithic architecture	Microservices architecture
Code	A single code base for the entire application.	Multiple code bases. Each microservice has its own code base.
Understandability	Often confusing and hard to maintain.	Much better readability and much easier to maintain.
Deployment	Complex deployments with maintenance windows and scheduled downtimes.	Simple deployment as each microservice can be deployed individually, with minimal if not zero downtime.
Language	Typically entirely developed in one programming language.	Each microservice can be developed in a different programming language.
Scaling	Requires you to scale the entire application even though bottlenecks are localized.	Enables you to scale bottle-necked services without scaling the entire application.

Microservices Architectures

- **Single-Service Microservices:** Each microservice is responsible for a single, very specific task. This results in a large number of very small microservices.
- **Domain-Driven Microservices:** The services are designed based on business capabilities and priorities. This approach emphasizes understanding the business's needs and modeling the services accordingly.
- **Serverless Microservices:** This is a new type of microservice where the service is hosted on a third-party server. This approach helps businesses to scale automatically and pay only for the resources they use.

5 Design Principles of MOA – Admin in mind

- The benefits of an MOA can be significant, but they come with a price. You need to know a thing or two about microservice design to implement an MOA effectively—you can't make it up as you go along. A microservice application must follow these five principles:
 - Single concern
 - Discrete
 - Transportable
 - Carries its own data
 - Inherently ephemeral

1. Single concern microservice

- Having a single concern means that a microservice should do one thing and one thing only. For example, if the microservice is intended to support authentication, it should do authentication only.
- This means that its interface should expose only access points that are relevant to authentication. And internally, the microservice should have authentication behavior only.
- For example, there should be no side behavior such as providing employee contact information in the authentication response.

2. Discrete microservice boundaries

- A microservice must have clear boundaries separating it from its environment. Another way to think about this principle is that a microservice must be well-encapsulated.
- This means that all logic and data relevant to a microservice's single concern must be encapsulated into a single deployment unit. Examples of units for discrete deployment are a Linux container, a WebAssembly binary, a .NET DLL, a Node.js package, and a Java JAR file, to name a few.
- Also, a discrete microservice is hosted in a distinct source control repository and is subject to its own CI/CD (continuous integration/continuous delivery) process.

3. Transportable microservice

- A transportable microservice can be moved from one runtime environment to another with little effort. Perhaps currently, the optimal form of a transportable microservice is a Linux container image.
- Usually, a Linux container image is hosted in an image repository such as Red Hat Quay.io. The container image can be targeted to any destination from that image repository, so a variety of applications can use the image. This is all possible because the microservice is encapsulated into a discrete deployment unit that can be transported to any destination. The encapsulation removes from developers all tasks except configuration and deployment.

4. Carry-its-own-data microservice

- A microservice should have its own data storage mechanism that is isolated from all other microservices. The only way data can be shared with other microservice is by way of a public interface that the microservice publishes.
- This principle imposes some discipline on data sharing: For instance, the particular data schema used by each microservice has to be well-documented. The design rules out behind-the-scenes hanky-panky that makes data hard to access or understand.
- The principle that a microservice carries its own data is hard for many developers to accept. The common argument against a microservice carrying its own data is that the principle leads to a proliferation of data redundancy.

5. Inherently ephemeral

- The principle that a microservice is ephemeral means that it can be created, destroyed, and replenished on-demand on a target easily, quickly, and with no side effects. The standard operating expectation is that microservices come and go all the time; sometimes due to system failure or scaling demands.
- This scenario is common in a Kubernetes environment that uses the Horizontal Pod Autoscaler (HPA) to accommodate scaling demands. The HPA creates and destroys containers according to momentary demands. Each time a container is created, an IP address is assigned dynamically. There are even situations where port numbers will be assigned dynamically too. Such are the impacts of ephemeral computing.

10 Design Principles of MOA – Developers in Mind

- For developing an optimal microservice architecture, you need to follow some design principles.
 - Independent and Autonomous/Self-governing services
 - API aggregation
 - Flexibility
 - Scalability
 - Constant monitoring
 - Failure Isolation/ Failure resilience
 - Realtime Load balancing
 - Inclusion of DevOps
 - Versioning
 - Availability

1. Independent and Autonomous/Self-Governing Services

- Every microservice should be self-contained and should operate independently of all the other services in an application.
- Each service should have its resources to run including separate databases and business logic without relying on other services to perform any of its functions.
- The main reason behind this principle is to ensure that each service can be **developed, tested, as well as deployed independently** without affecting the other part of the system.

2. API Aggregation

- A developer should be able to design microservices in such a manner that they can communicate throughout the system with other microservices without having a programming language barrier.
- Most of the time different microservices belonging to the same system may be written in different programming languages but as the intercommunication among the microservices is important this is achieved with the help of **API aggregation** using different architectural approaches such as REST.
- This principle emphasizes the use of well-defined and consistent APIs to enable communication between different microservices in the system.

3. Flexibility

- The ability of a microservice to change or be changed easily according to the circumstances or the need of the time is referred to as the flexibility of a microservice.
- Assuming you added a feature that was criticized and was not successful, now that it needed to be taken out of the service, this can be done easily when you have made a flexible microservice. This principle makes your project more adaptable to possible future changes.

4. Scalability

- It is a design principle that enables your application to get modified according to the increasing or decreasing traffic, data, and complexity without affecting the performance of the system.
- There are several ways to achieve scalabilities such as Service partitioning, Load balancing, Horizontal scaling, and Caching.

5. Constant monitoring

- When numerous microservices are working in synchronization to successfully run an application there is a high possibility of faults occurring throughout the system.
- As the architecture becomes more complex, it becomes a tedious task to find and resolve these issues. To avoid this, constant monitoring is required so that the faults can be resolved at the earliest and your application can run smoothly.
- Constant monitoring can be achieved by using these methods-
 - Logging and metrics
 - Distributed tracing
 - Health checks
 - Alerting and notifications

6. Failure Isolation/ Failure Resilience

- As a developer, you should always be prepared to encounter failures in your application, and to minimize these failures you should include fault-tolerant approaches during the designing of microservices.
- This principle helps to minimize the impact of failures on the whole system due to the failure of a particular system.
- There are several techniques to make your application fault tolerant:
 - Include Redundancy in order to have multiple instances present at the time of failure.
 - Separation of Services in order to be isolated in case of failures.
 - The microservices should Gracefully Degrade instead of crashing the whole service.
 - Each microservice should have Circuit Breakers in order to detect and isolate the failures.

7. Real-time Load Balancing

- Sometimes it may happen when a client sends the request to the server then the database has to retrieve the data from multiple microservices at the same time.
- At that point, the Load balancer comes into the picture and it defines how much amount of the Central Processing Unit or GPU is to be used for a particular service to fetch the data and finally how the client request should be passed.

8. Inclusion of DevOps

- The inclusion of DevOps in a project promotes communication and collaboration between different microservices that in the end improves the efficiency and effectiveness of the software development process. DevOps helps microservices to achieve greater speed, high flexibility, and agility.
- DevOps inclusion can be done using several services such as Automated Testing, Continuous Delivery, and Development as well as Continuous Monitoring that leads to faster time-to-market and best-quality software.
- Docker and Kubernetes are two common DevOps tools used for managing microservices.

9. Versioning

- Versioning is a term used for updating to the latest versions of the services according to the latest tech stacks that are being used in present.
- Versioning helps to manage changes in the services over time and update them to the latest ones. This results in minimizing the disruption to the existing clients using the services.
- The versioning process includes updating version numbers, service interface, request, and response through URL versioning and the same in Header Versioning.

10. Availability

- When businesses are serving worldwide then there are no chances that can be taken with downtime.
- Every minute can be proved to serve big losses if the services are down, thus **availability is a very important design principle** which states that all the services should be developed in such a manner that the services are up and working 24x7 or if not possible, they can at least be available for the maximum amount of time.
- Availability straightway means the presence of microservices **24x7**.

Chapter 2

Microservice Technologies and Languages

Objectives

Chapter 2: Microservice Technologies & Languages

- Understandings of microservices architecture
- Overview of familiar Technologies and languages
- Learn about popular technologies that enable the development, deployment, and support of microservices

Microservice Languages & Technologies

- The microservices architecture is a development methodology wherein you can fragment a single application into a series of smaller services, each executing in its own process and interacting with lightweight mechanisms.
- The microservices are developed around business capabilities, which are independently deployable with automated deployment mechanism.
- The microservices architecture needs a bare minimum of management of these services, built in different programming languages and employs different data storage technologies.

Criteria for Choosing a Technology for Microservices

- With microservices, you can build a reliable platform to extend the business while taking advantage of diversity in languages. Of course, you can use different technologies or languages for different services, but it does not mean it is effective.
- The microservices architecture comes with lots of operational overhead; hence adding a diverse programming language on top of that can exponentially raise that performance overhead.
- To reduce that, you should standardize your microservices technology stack by choosing the programming language based on your business needs. Here are the criteria to evaluate the programming language for microservices development:
 - Highly observable
 - Support for automation
 - Consumer-first approach
 - Independent deployment
 - Modelled around business domain
 - Decentralization of components
 - Support for continuous integration

Best Languages for Microservices

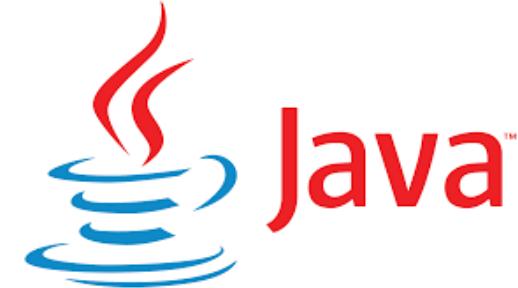
- Microservices can be implemented with a horde of frameworks, versions, and tools. Java, Python, C++, Node JS, and .NET are few of them. Let us explore the languages that support microservices development in detail through upcoming slides.
- The natural question that comes to your mind is, Which is the most favorable language to be used while developing microservices for your next project?

1. Java



- Annotation syntax, which is easy to read, is the key factor that makes Java a great programming language for developing microservices. This feature makes Java Microservices much easier to develop when powered by Microservices frameworks.
- It offers more value in readability, particularly while working with complex systems.
- Java includes many opinions to support developing and deploying Java Microservices.
- It offers a user Interface, model components as well as connectivity to back-end resources, everything within the boundaries of a single, isolated, and independently deployed apps.

1. Java



- There are several Frameworks for developing Microservices architecture. Some of the Java Microservices Frameworks are as follows:
 - **Spring Boot** – This framework works on top of various languages for Aspect-Oriented programming, Inversion of Control and others
 - **Dropwizard** – This Java microservices framework assembles stable and mature libraries of Java into a simple and light-weight package
 - **Restlet** – It supports developers to build better web APIs, which trail the REST architecture model
 - **Spark** – One of the best Java Microservices frameworks, supports creating web apps in Java 8 and Kotlin with less effort

1. Java



- In addition, many of Java EE standards are well suited for microservices applications like:
 - JAX-RS for APIs
 - JPA for data handling
 - CDI for dependency injection & lifecycle management
- In addition, service discovery solutions like Consul, Netflix Eureka or Amalgam8 are effortless in connecting with Java Microservices.



Golang

2. GoLang

- If you want to enhance your existing project, the Golang can be a good choice for microservices development. Golang, also known as Go is popular for its concurrency and API support in terms of microservices architecture.
- With the Golang's concurrency possibility, you can expect increased productivity of various machines and cores. It includes a powerful standard for developing web services. It is exclusively designed for creating large and complex applications. Go provides two impressive frameworks for microservices development:
 - **GoMicro** – It is an RPC framework, which comes with the advantages like Load balancing, server packages, PRC client, and message encoding.
 - **Go Kit** – The key difference of Go Kit from GoMirco is it needs to be imported into a binary package. Moreover, it is advanced for explicit dependencies, Domain-driven design, and declarative aspect compositions.



3. Python

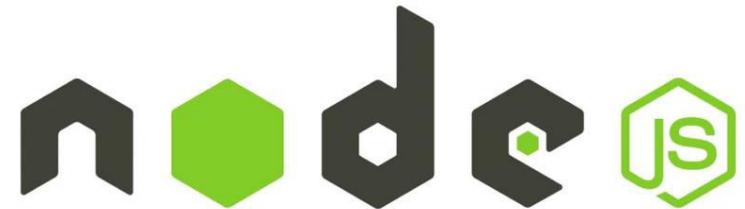
- Python is a high-level programming language that offers active support for integration with various technologies.
- Prototyping in Python is faster and easier when compared to other frameworks and languages.
- It includes powerful substitutes for heavy implementations like Django.
- Microservices Python ensures compatibility with legacy languages like ASP and PHP, which allows you to create web service front-ends to host Microservices.



3. Python

- With all these benefits, Microservices Python is considered to have an edge over other languages.
- Developers who implement Microservices Python use a RESTful API approach - a comprehensive way of utilizing web protocols & software to remotely manipulate objects.
- There is a broad range of Python microservices frameworks to choose from for your web application development. Some of them are as follows:
 - **Flask** – Most popular Python Micro framework based on Jinja2 and Werkzeug
 - **Falcom** – Create smart proxies, cloud APIs and app back-ends
 - **Bottle** – Simple, lightweight and fast WSGI micro framework
 - **Nameko** – Best among the Python Microservices frameworks that allows developers to concentrate on application logic
 - **CherryPy** – Mature, Python object-oriented web framework

4. Node.js



- Node JS became the go-to platform in the past few years for enterprises and startups who want to embrace microservices.
- Node JS is built with the V8 runtime; hence, microservices Node JS is going to be super-fast for Input-Output (IO) – bound tasks. Normally, Microservices Node JS is developed either using CPU-bound or IO-bound code.
- CPU-bound program demands many intensive calculations. Every time you run an IO call, Node JS doesn't block the main-thread but submits the tasks to be executed by the internal IO daemon threads.
- Hence, Microservices Node JS gains popularity in terms of IO-bound tasks.



5. .NET Framework

- ASP.Net, the .NET framework for web development makes it simple to build the APIs that becomes the microservices.
- It includes built-in support for building and deploying microservices using Docker containers. .NET comes with APIs that can simply consume microservices from any application you developed including desktop, mobile, web, gaming and more.
- If you have an application, you can start adopting .NET microservices without entirely revamping that application.
- The initial setup for .NET Docker images has already been done and available on Docker Hub, helping you to concentrate only on building your microservices.

5. .NET Framework



- The .NET microservices architecture allows a compilation of technologies between each service; as such, you can use .Net for a certain part of your app without implementing it everywhere.
- It can mix the .NET microservices with applications written in Java, Node JS, or any other languages. This allows a gradual migration to .NET core technology for new microservices that function in combination with other microservices and with services built with other technologies. Similarly, the .NET microservices can run on all leading cloud platforms.

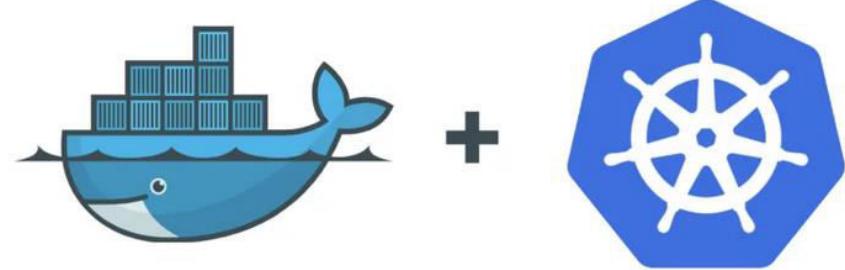
Microservices Technologies

- The main target of microservices is to simplify the deployment process and build cloud-ready, high-end software programs quickly.
- By fragmenting a single app into several smaller programs, you make them simpler to change, scale, and deploy. The microservices architecture employs different data storage technologies. It requires some in-build programming languages also.
- With the growing popularity of microservices, many new tools and technologies have developed. Let's take your look at the most effective technologies for developing microservices architecture!

Best Technologies for Microservices Architecture Development

- Microservices can be executed using different tools, versions, and frameworks. Let's learn more about the technologies that facilitate microservices development!
 - Docker and Kubernetes
 - REST
 - Consul
 - Prometheus
 - Redis
 - RabbitMQ

1. Docker and Kubernetes



- Docker is a containerization technology that helps develop, test, and run software systems as self-contained packages in a container.
- Kubernetes is a system made for automating the manual tasks engaged in deploying and handling containerized applications. It helps with container scheduling, scaling, and so forth.
- Using the combination of these technologies will help build a responsive setup for microservices architecture. It seamlessly scale up or down according to your needs.

1. Docker and Kubernetes



- Benefits of Docker
 - Docker helps you deploy your software easily as you just need to deploy Docker images utilizing Dockerfiles.
 - It supports many operating systems, several available plugins that improve its features, and integration with demanding tools and services.
 - Docker offers facilities for constant distribution and integration suitable for DevOps culture.
 - It's a lightweight technology to create microservices infrastructure.
 - Docker Compose helps coordinate many containers to create the whole system of microservices with containers.



2. REST

- REST (Representational State Transfer) is a tool that helps microservices to communicate with other microservices.
- This architectural design pattern enables microservices to communicate through HTTP directly. It deals with responses and requests in standard formats, such as JSON, HTML, or XML.
- Benefits of REST:
 - An amazing tool to create a scalable microservice.
 - Allows the server and the client to be applied separately without informing the other entity.
 - At every side, codes can be changed without impacting one another.

3. Consul



- Consul technology helps microservices in communicating with one another.
- Due to its exclusive features, it stands out from the rest service discovery technologies. Because of the Consul Template and DNS interface, you can use Consul with other technologies also.
- Using Consul to set up microservices architecture is helpful for the synchronous system. After all, its infrastructure fulfills all the basic challenges of synchronous microservices.

3. Consul



- Benefits of Consul
 - One of the best service discovery technologies.
 - Helps configure microservices.
 - Features an HTTP REST API.
 - Performs health checks.
 - Completely transparent and you can use it with no code dependency.

4. Prometheus



Prometheus

- Prometheus is an alerting tool & a full-service monitoring system made for multiple complex app topologies containing many nodes.
- This tool uses key-value labels to execute multidimensional data and provides datastore and scrapes. Prometheus is an easy and rapid tool that filters data depending on their labels.

4. Prometheus



Prometheus

- **Benefits of Prometheus**

- Supports multi-dimensional data collection and querying.
- Provides an extensible data model that enables to add of arbitrary key-values dimensions to a time series.
- Simple capacity and design to build minimalist apps.
- Perfect for simple microservices software systems.
- Ideal for cloud-native deployed environments.

5. Redis



- Being a versatile technology, Redis provides many features that make it strong enough for several applications.
- It's a high-speed, flawless NoSQL database that facilitates elastic data structures and quick responses made to fix complicated coding issues using easy commands.
- Apart from a core data model, Redis also aids Streams, Hashes, and other types of values.
- It covers data persistence, seamless data sharing between microservices, and secure temporary data storage.

5. Redis



- Redis is a high-performing platform. It can manage millions of operations in no time and read operations in a millisecond. Its distributed caching engine is another perk to consider.
- Benefits of Redis
 - Gives users access to a scalable datastore that provides many servers, apps, and processes.
 - Helps optimize the data layer, everything from syncing across geographies to inter-service communication via data management.
 - Helps reduce client workload and accelerate the app's function as a cache manager.
 - Multipurpose usage for different data handlings.

6.RabbitMQ



- This trustworthy message technology facilitates both pub/sub messaging patterns and message queuing and different consistent protocols.
- Most developers consider RabbitMQ as a unique message broker. To use SSL, you can configure RabbitMQ, which helps make an additional security layer.
- Benefits of RabbitMQ
 - Helps with complicated routing communication on a difficult microservice-based app.
 - Useful for long-term jobs if you need to run authentic background tasks.
 - Features publisher confirms and persistence to help you establish functionality with reliability.

Chapter 3

Integration and Decomposition

Microservices

Objectives

Chapter 3: Integration and Decomposition Patterns

- Learn about the integration and inter-communication patterns
- Learn about the decomposition patterns from monolith to microservices

Design Patterns for Microservices

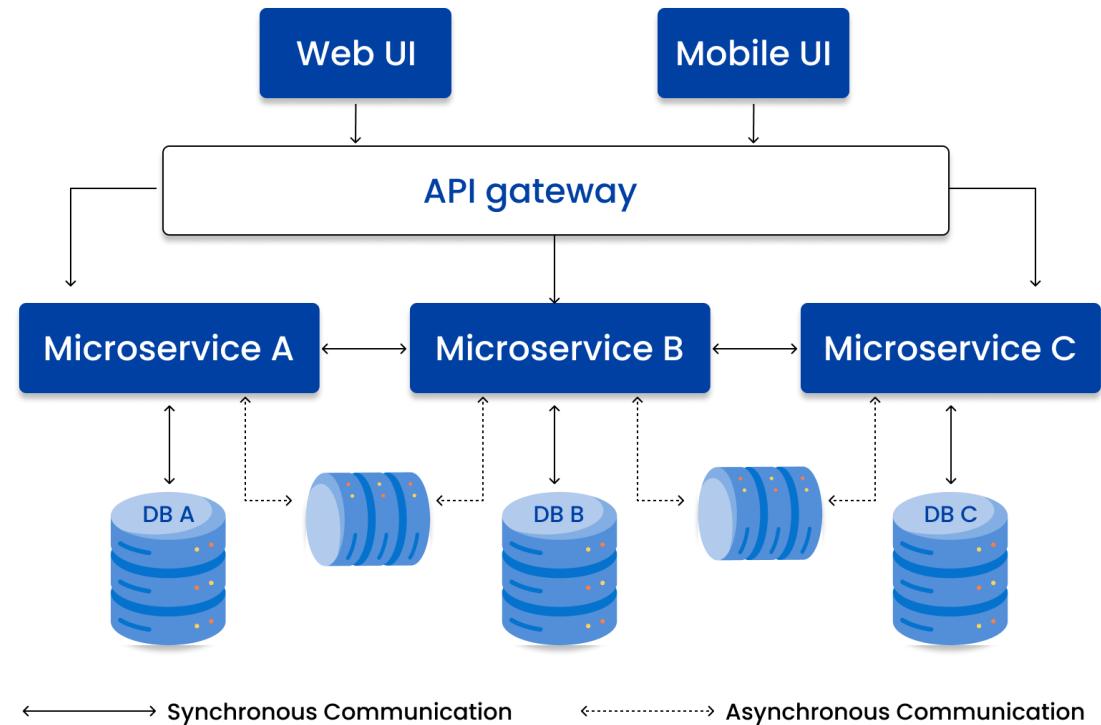
- Microservices design patterns are fundamental to creating robust, scalable, and easily maintainable microservices-based applications.
- The implementation of these patterns streamlines development and significantly improves the quality and maintainability of the resulting applications.
- Recognizing and applying these patterns effectively can often be the difference between the success and failure of a microservices-based project. However, it's important to remember that each pattern comes with its own benefits and drawbacks.

Integration patterns

- Integration patterns are a key aspect of a microservices architecture. They provide a roadmap to enable multiple microservices, possibly employing different protocols like REST or AMQP, to function in harmony.
- These patterns aim to provide an efficient way for clients to interact with individual microservices without having to handle the intricacies of various protocols.
 - API Gateway Pattern
 - Aggregator Pattern
 - Proxy Pattern
 - Gateway Routing Pattern
 - Chained Microservice Pattern
 - Branch Pattern
 - Client-Side UI Composition Pattern

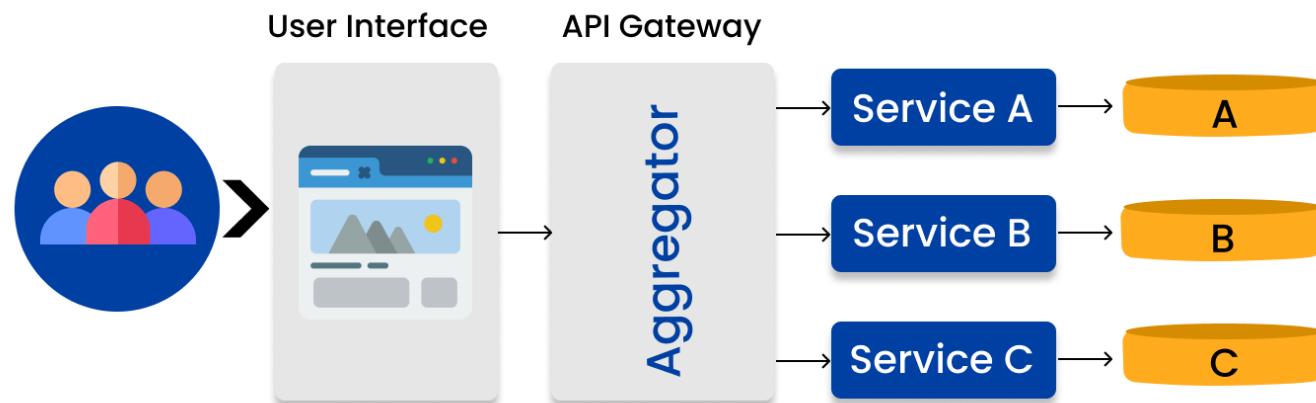
1. API Gateway Pattern

- In this pattern, an API gateway serves as a reverse proxy that routes client requests to the relevant microservice, thereby minimizing the need for the client to interact with multiple microservices directly.
- The API gateway also consolidates the results from different microservices, enhancing security by being the single point of contact for users.



2. Aggregator Pattern

- It is essential to consider how to coordinate the data generated by each service when decomposing business functionality into smaller logical units of code. This cannot be held accountable to the consumer.
- The Aggregator pattern is a useful remedy for this problem. This demonstrates the potential for integrating data from multiple services to provide a comprehensive response to the customer.

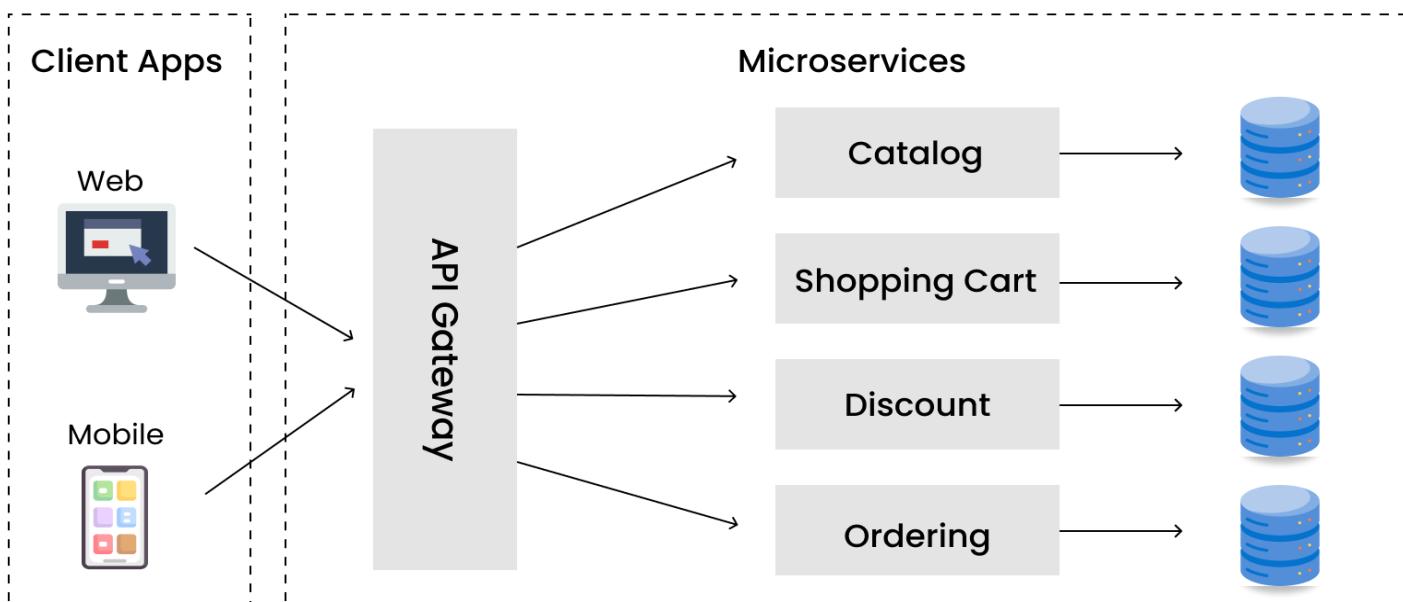


3. Proxy Pattern

- This pattern involves creating a proxy microservice that invokes other services based on business requirements, eliminating the need for an aggregator on the consumer end.

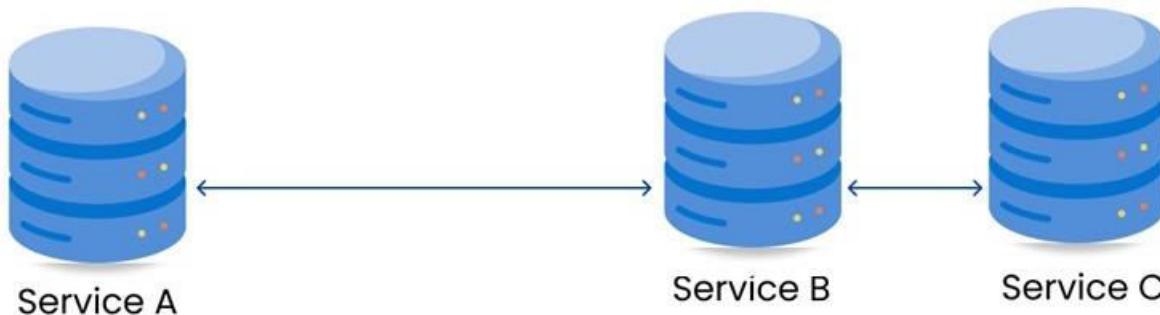
4. Gateway Routing Pattern

- The Gateway Routing pattern exposes multiple services through a single endpoint, subsequently routing the requests to the relevant backend microservices.
- For instance, an e-commerce application can use this pattern to offer a range of services like customer search, shopping cart, discounts, and order history.



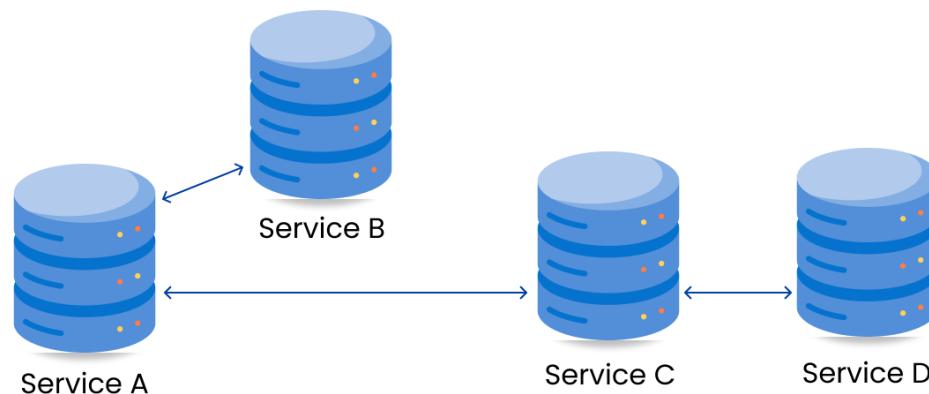
5. Chained Microservice Pattern

- In cases where a single service has multiple dependencies, the Chained Microservice pattern can be used.
- It enables a sequence of synchronous calls between microservices, ensuring a unified result in response to a given request.



6. Branch Pattern

- The Branch microservice pattern is a combination of the Aggregator and Chain design patterns.
- It allows simultaneous processing of requests and responses from multiple microservices, making it an excellent fit for microservices that need to pull data from multiple sources.



7. Client-Side UI Composition Pattern

- This pattern becomes essential when user experience services need to fetch data from various microservices.
- Unlike monolithic architecture, where a single call from the user interface fetches data from a backend service, this pattern requires a user interface design segmented into different sections, with each section retrieving data from a distinct backend microservice.

Inter-Service Communication

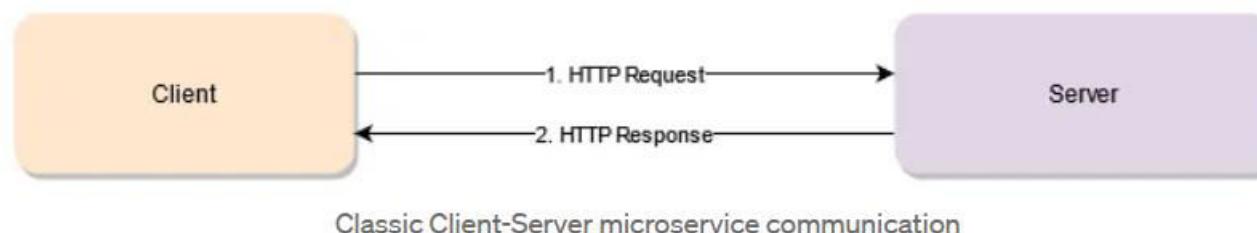
- Inter-service communication is a pivotal concept in the microservices architecture. In a microservices architecture, an application is broken down into loosely coupled, independently deployable services.
- These services need to communicate with each other to perform their tasks.
- For Example: In the Spring framework, there are several ways to handle inter-service communication, ensuring seamless operation and interaction between microservices.

Types of Inter-Service Communication

- Client and services can communicate through many different types of communication, each one targeting a different scenario and goals.
Initially, those types of communications can be classified in two axes.
 - Synchronous protocol
 - Asynchronous protocol

HTTP APIs

- An HTTP API essentially means having your services send information back and forth like you would through the browser or through a desktop client like Postman.
- It uses a client-server approach, which means the communication can only be started by the client. It is also a synchronous type of communication, meaning that once the communication has been initiated by the client, it won't end until the server sends back the response.

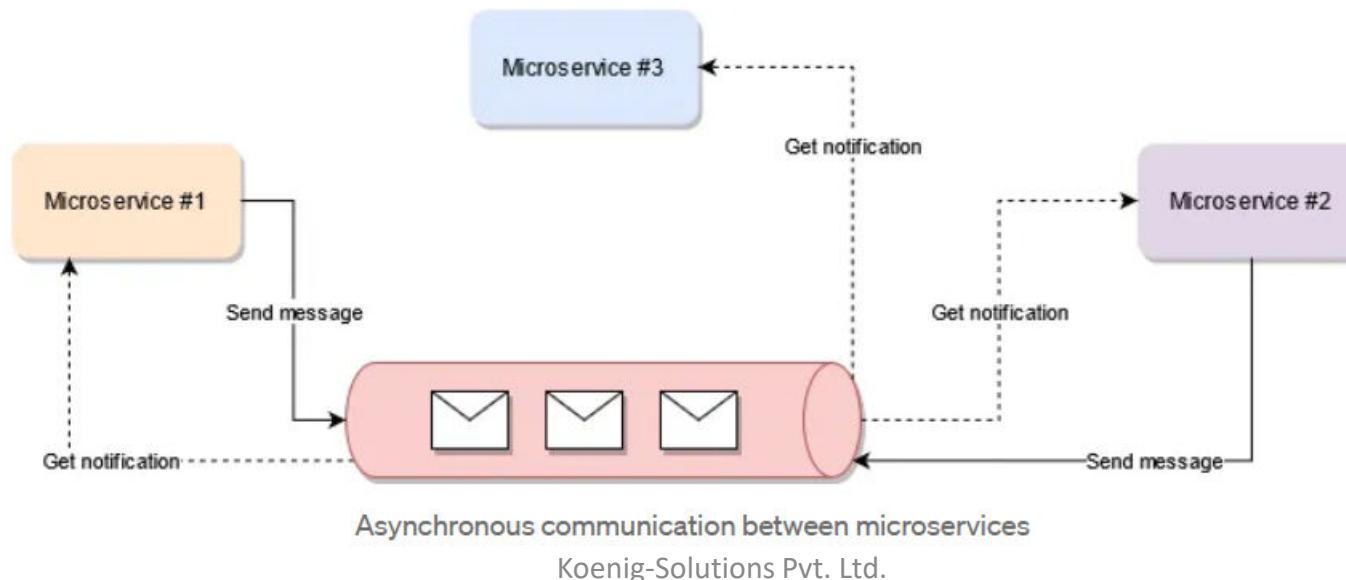


HTTP APIs

- Pros
 - Easy to implement
 - Quite Standard
 - Technology agnostic
- Cons
 - Channel adds delay to business logic
 - Timeouts
 - Failures aren't easy to solve

Asynchronous Messaging

- This pattern consists of having a message broker between the producer of the message and the receiving end.
- This is definitely one of my favorite ways of communicating multiple services with each other, especially when there is a real need to horizontally scale the processing power of the platform.



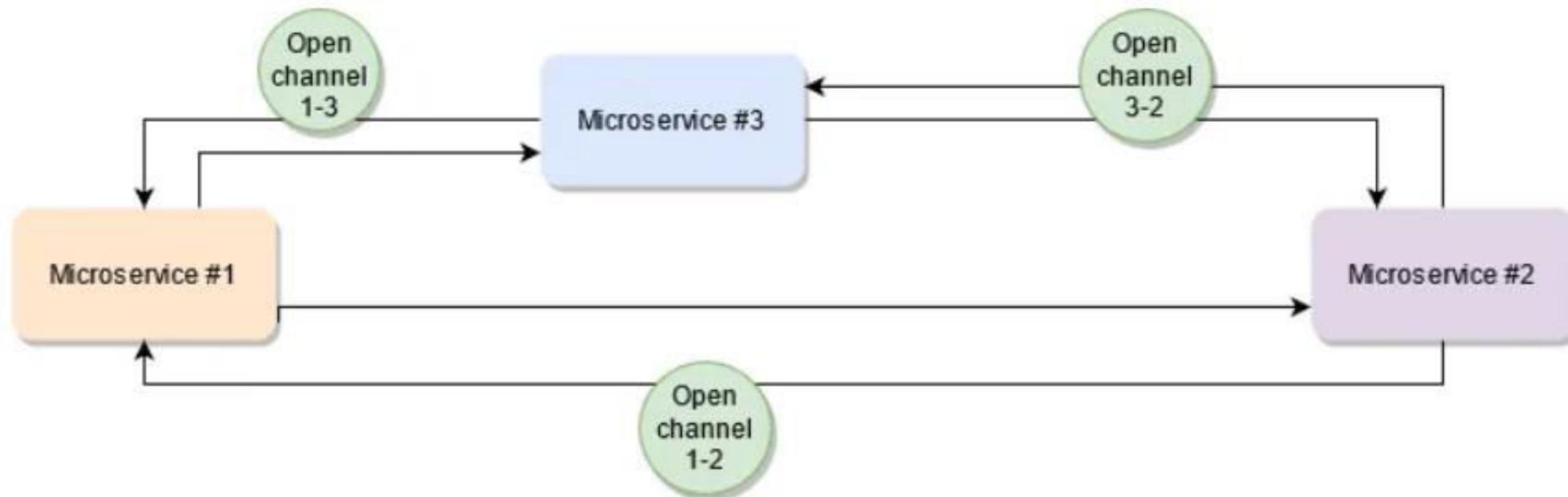
Asynchronous Messaging

- Pros
 - Easy to scale
 - Easy to add new services
 - Easier retry mechanics
 - Event-driven
- Cons
 - Debugging gets a bit harder
 - There is no clear direct response
 - The broker becomes a single point of failure

Direct Socket Communication

- At a first glance, the socket-based communication looks a lot like the client-server pattern implemented in HTTP, however, if you look closely there are some differences:
 - For starters, the protocol is a lot simpler, which means a lot faster as well. Granted, if you want it to be reliable you need to code a lot more on your part to make it so, however, the inherent latency added by HTTP is gone here.
 - The communication can be started by any actor, not only the client. Through sockets, once you have your channel open, it'll stay that way until you close it. Think about it as an ongoing phone call, anyone can start the conversation, not only the caller.

Direct Socket Communication



Open channels with sockets for microservice communication

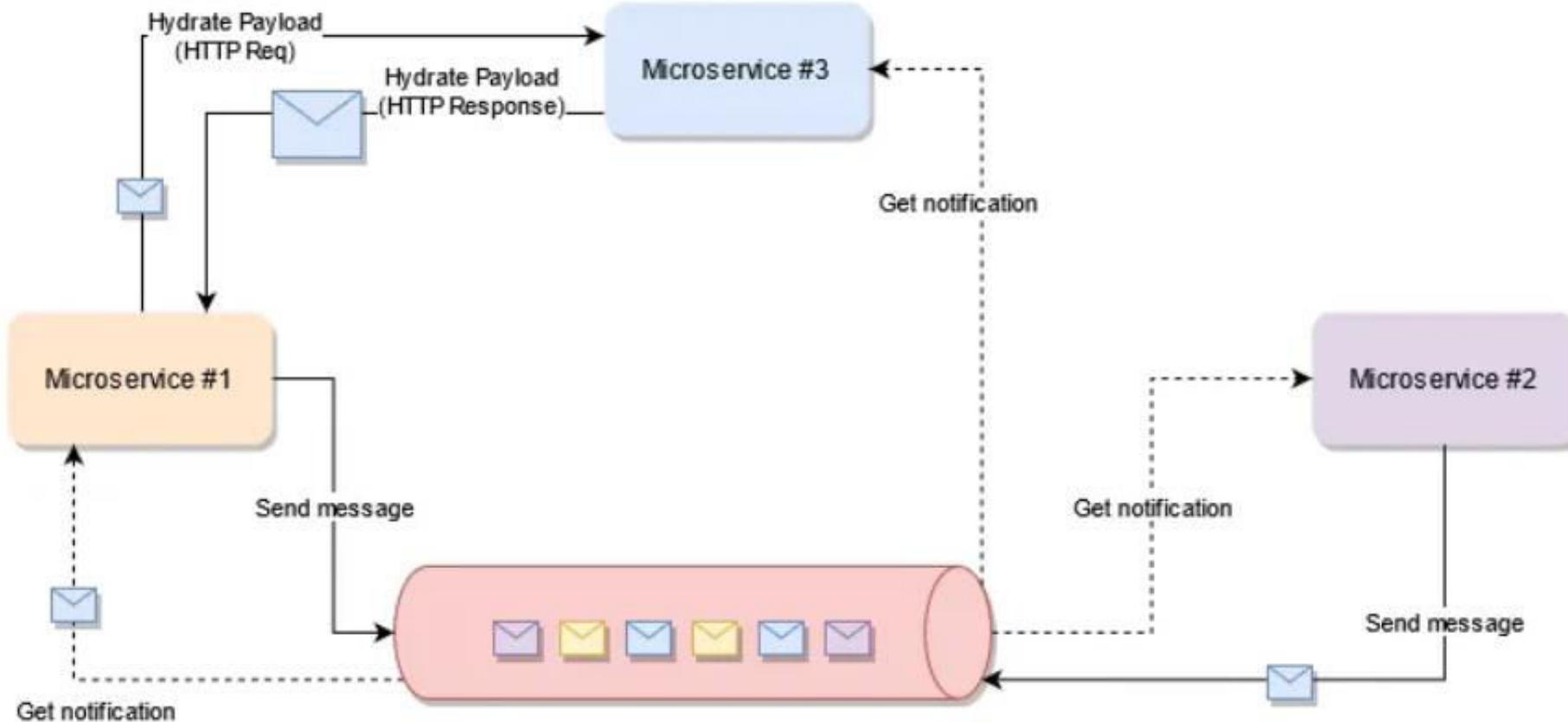
Direct Socket Communication

- Pros
 - It's very lightweight
 - Allows for a very optimized communication process
- Cons
 - No real standards in place
 - Easy to overload the receiving end

Lightweight Events

- This pattern mixes the first two on this list. On one side, it provides a way of having multiple services communicate with each other through a message bus, thus allowing for asynchronous communication.
- And on the other side, since it only sends very lightweight payloads through that channel, it requires services to hydrate that payload with extra information through a REST call to the corresponding service.

Lightweight Events



Lightweight events & hydration during microservice communication

Lightweight Events

- Pros
 - The best of both worlds
 - Focused on optimizing the most common scenario
 - Basic buffer
- Cons
 - You might end up with too many API requests
 - Double communication interface

Decomposing Monolith into Microservices

- The process of decomposition entails the partitioning of a monolithic application into microservices that are organized according to functional boundaries.
- The objective of this pattern is to enhance maintainability and resilience by enabling each microservice to operate autonomously.

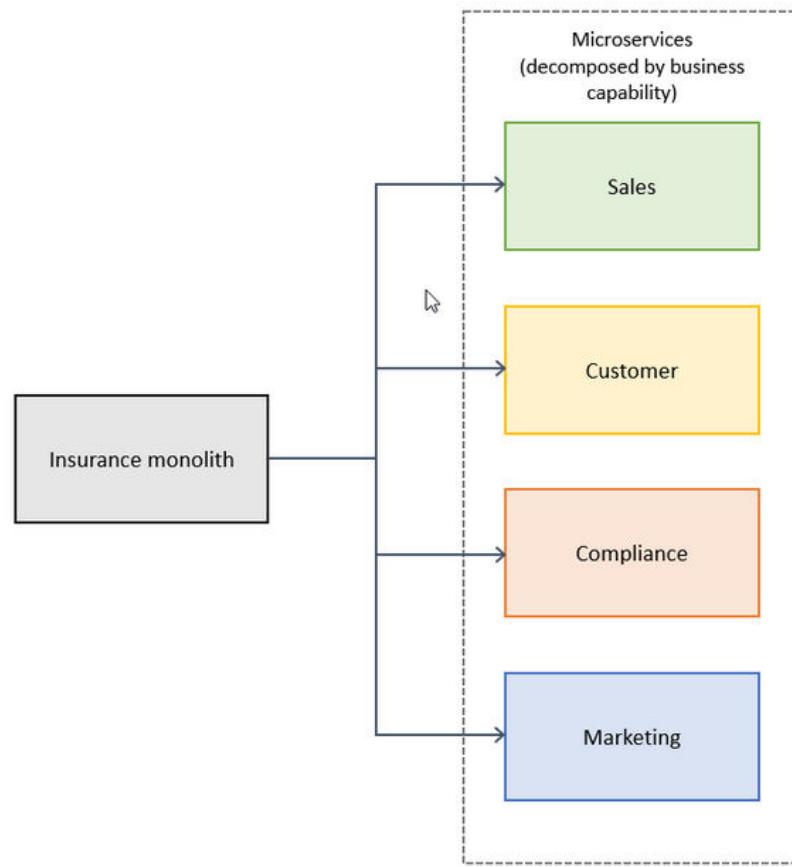
Decomposing Monolith into Microservices

- There are some patterns for decomposing the monolith to microservices:
 - Decompose by Business Capability
 - Decompose by Domain-Driven Design
 - Decompose by Transactions
 - Strangler fig Pattern
 - Service per team pattern
 - Branch by abstraction pattern

Decompose by Business Capability

- The term "business capability" is a fundamental concept utilized in business architecture modeling.
- A business capability is what a business does to generate value (for example, sales, customer service, or marketing). Typically, an organization has multiple business capabilities and these vary by sector or industry. Use this pattern if your team has enough insight into your organization's business units and you have subject matter experts (SMEs) for each business unit.
- Value generation is a fundamental objective of business operations. A business capability typically aligns with a business object.

Decompose by Business Capability



Decompose by Business Capability

Advantages	Disadvantages
<ul style="list-style-type: none">• Generates a stable microservices architecture if the business capabilities are relatively stable.• Development teams are cross-functional and organized around delivering business value instead of technical features.• Services are loosely coupled.	<ul style="list-style-type: none">• Application design is tightly coupled with the business model.• Requires an in-depth understanding of the overall business, because it can be difficult to identify business capabilities and services.

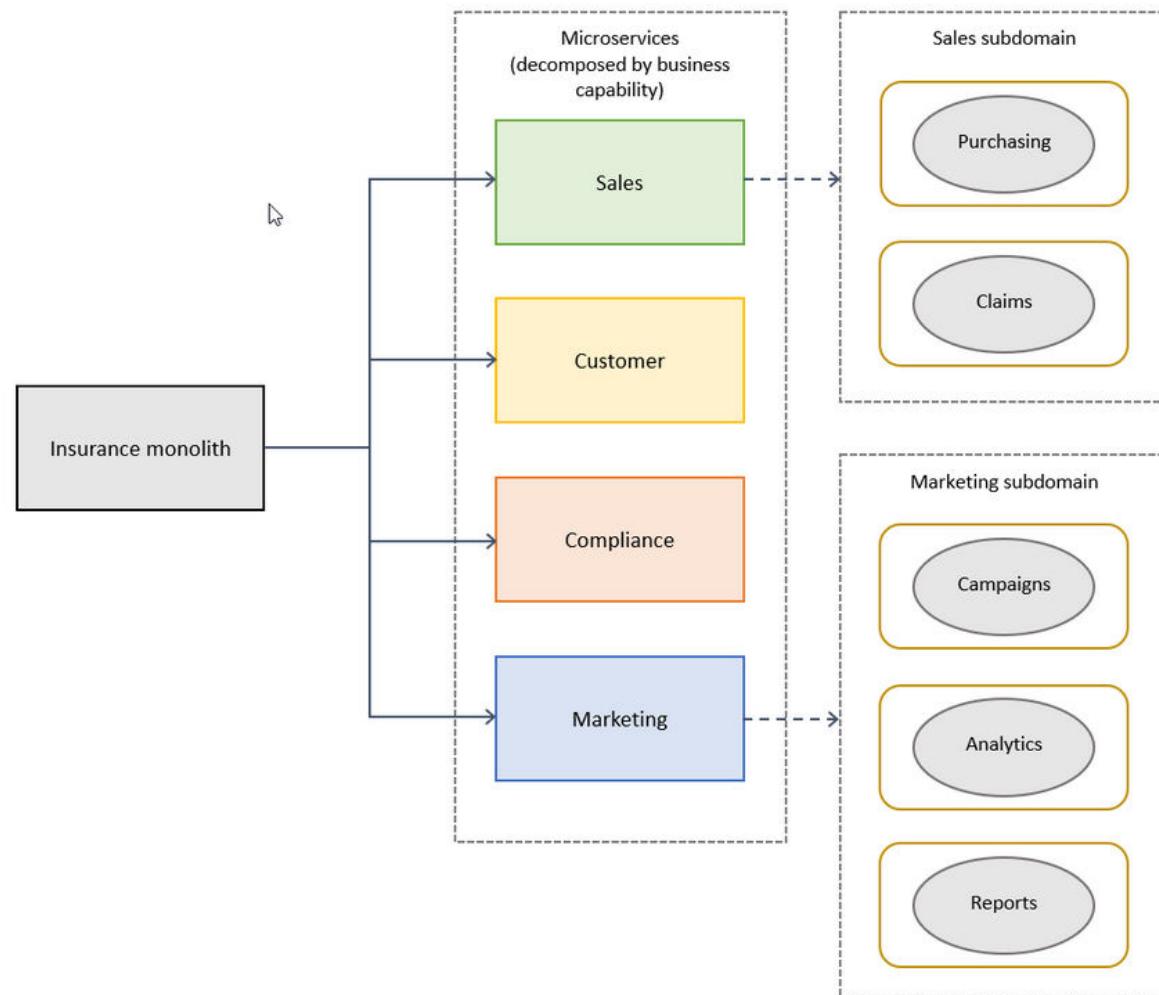
Decompose by Domain-Driven Design

- This task involves defining services that align with the subdomains of Domain-Driven Design (DDD).
- Domain-Driven Design establishes the application's domain or problem space. Domains have subdomains. Each subdomain is associated with a distinct segment of the business. Subdomain classifications include:
 - Core - The enterprise's core is the software's most valuable part.
 - Supporting- "Supporting" activities or features are related to core business operations but do not provide a competitive advantage. These solutions can be internal or outsourced.
 - Generic- Generic solutions use readily available software and are not tailored to a specific organization.

Decompose by Domain-Driven Design

- This pattern uses a domain-driven design (DDD) subdomain to decompose monoliths. This approach breaks down the organization's domain model into separate subdomains that are labeled as *core* (a key differentiator for the business), *supporting* (possibly related to business but not a differentiator), or *generic* (not business-specific).
- This pattern is appropriate for existing monolithic systems that have well-defined boundaries between subdomain-related modules. This means that you can decompose the monolith by repackaging existing modules as microservices but without significantly rewriting existing code.
- Each subdomain has a model, and the scope of that model is called a *bounded context*. Microservices are developed around this bounded context.

Decompose by Domain-Driven Design



Decompose by Domain-Driven Design

Advantages	Disadvantages
<ul style="list-style-type: none">• Loosely coupled architecture provides scalability, resilience, maintainability, extensibility, location transparency, protocol independence, and time independence.• Systems become more scalable and predictable.	<ul style="list-style-type: none">• Can create too many microservices, which makes service discovery and integration difficult.• Business subdomains are difficult to identify because they require an in-depth understanding of the overall business.

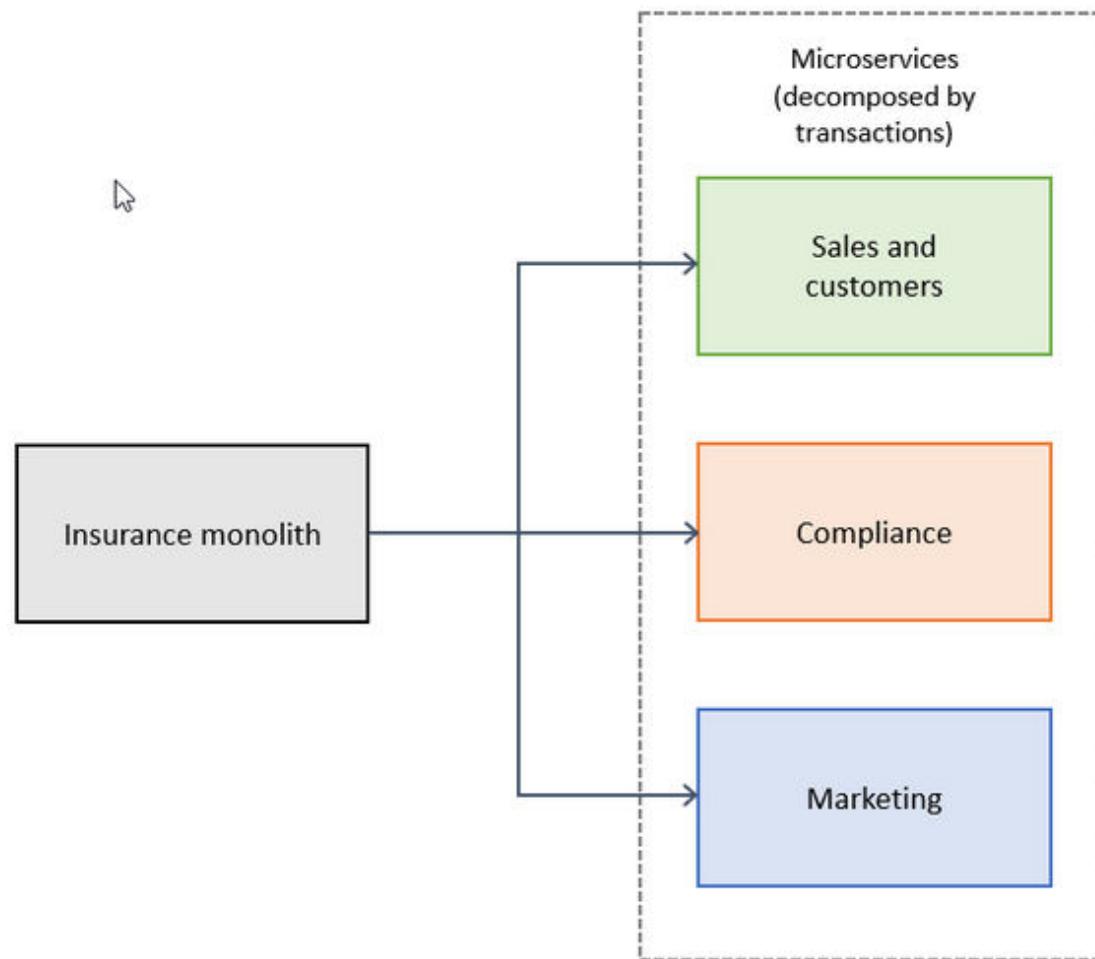
Decompose by Transactions

- In a distributed system, an application typically has to call multiple microservices to complete one business transaction.
- To avoid latency issues or two-phase commit problems, you can group your microservices based on transactions.
- This pattern is appropriate if you consider response times important and your different modules do not create a monolith after you package them.

Decompose by Transactions

- Services can be decomposed based on transactions. A distributed transaction involves two critical steps:
 - Prepare Phase: In this step, all parties involved in the transaction commit and inform the coordinator about their readiness for closure.
 - Commit or Rollback Phase: The transaction coordinator instructs all participants to either commit or rollback.
- It's important to note that the 2PC protocol tends to be slower than single microservice operations, making it less suitable for high-load scenarios.

Decompose by Transactions



Decompose by Transactions

Advantages	Disadvantages
<ul style="list-style-type: none">• Faster response times.• You don't need to worry about data consistency.• Improved availability.	<ul style="list-style-type: none">• Multiple modules can be packaged together, and this can create a monolith.• Multiple functionalities are implemented in a single microservice instead of separate microservices, which increases cost and complexity.• Transaction-oriented microservices can grow if the number of business domains and dependencies among them is high.• Inconsistent versions might be deployed at the same time for the same business domain.

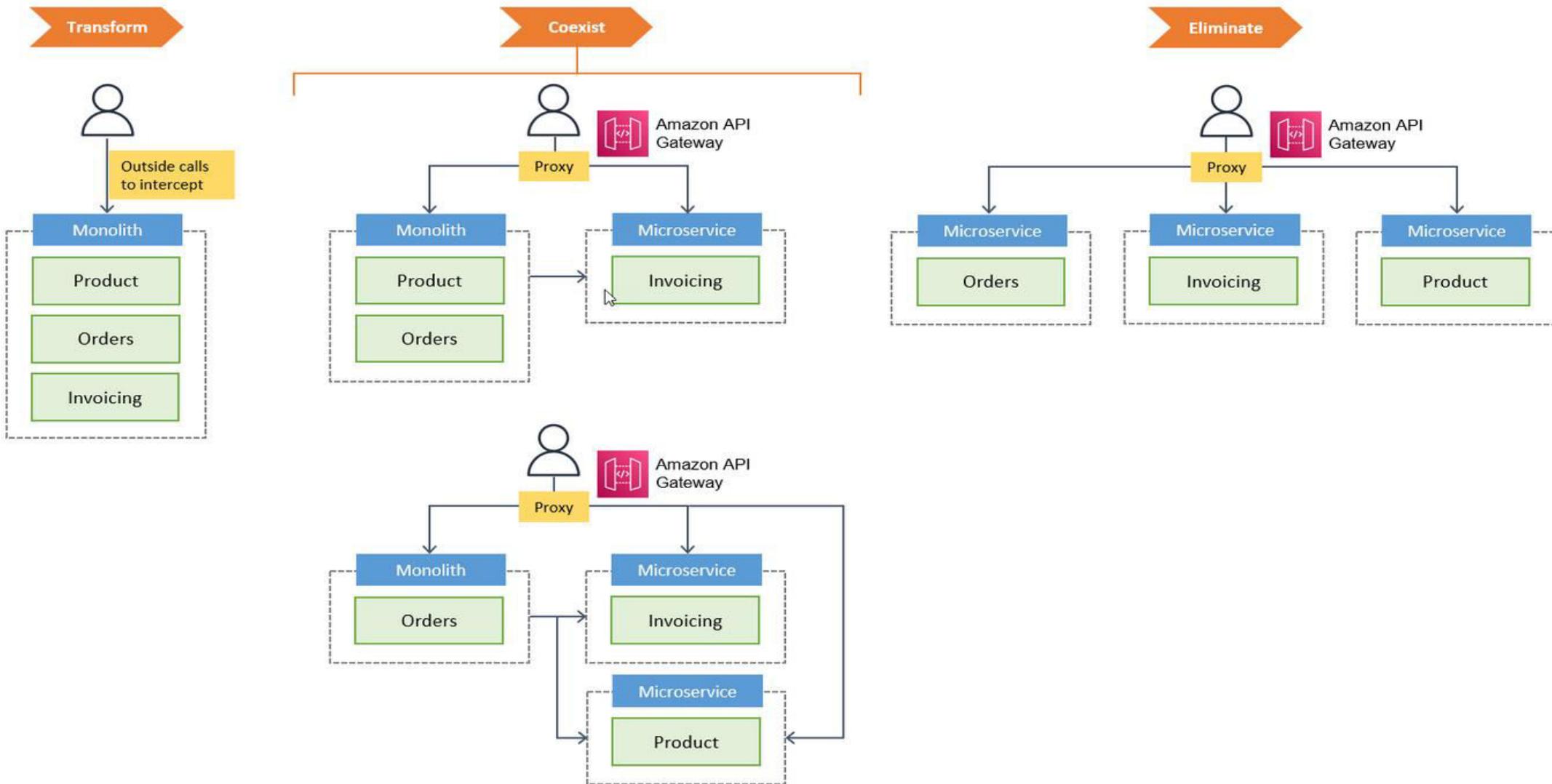
Strangler fig Pattern

- The design patterns discussed so far in this guide apply to decomposing applications for greenfield projects.
- What about brownfield projects that involve big, monolithic applications? Applying the previous design patterns to them will be difficult, because breaking them into smaller pieces while they're being used actively is a big task.
- The strangler fig pattern is a popular design pattern that was introduced by Martin Fowler, who was inspired by a certain type of fig that seeds itself in the upper branches of trees. The existing tree initially becomes a support structure for the new fig. The fig then sends its roots to the ground, gradually enveloping the original tree and leaving only the new, self-supporting fig in its place.

Strangler fig Pattern

- The process to transition from a monolithic application to microservices by implementing the strangler fig pattern consists of three steps: transform, coexist, and eliminate:
 - **Transform** – Identify and create modernized components either by porting or rewriting them in parallel with the legacy application.
 - **Coexist** – Keep the monolith application for rollback. Intercept outside system calls by incorporating an HTTP proxy (for example, Amazon API Gateway) at the perimeter of your monolith and redirect the traffic to the modernized version. This helps you implement functionality incrementally.
 - **Eliminate** – Retire the old functionality from the monolith as traffic is redirected away from the legacy monolith to the modernized service.

Strangler fig Pattern



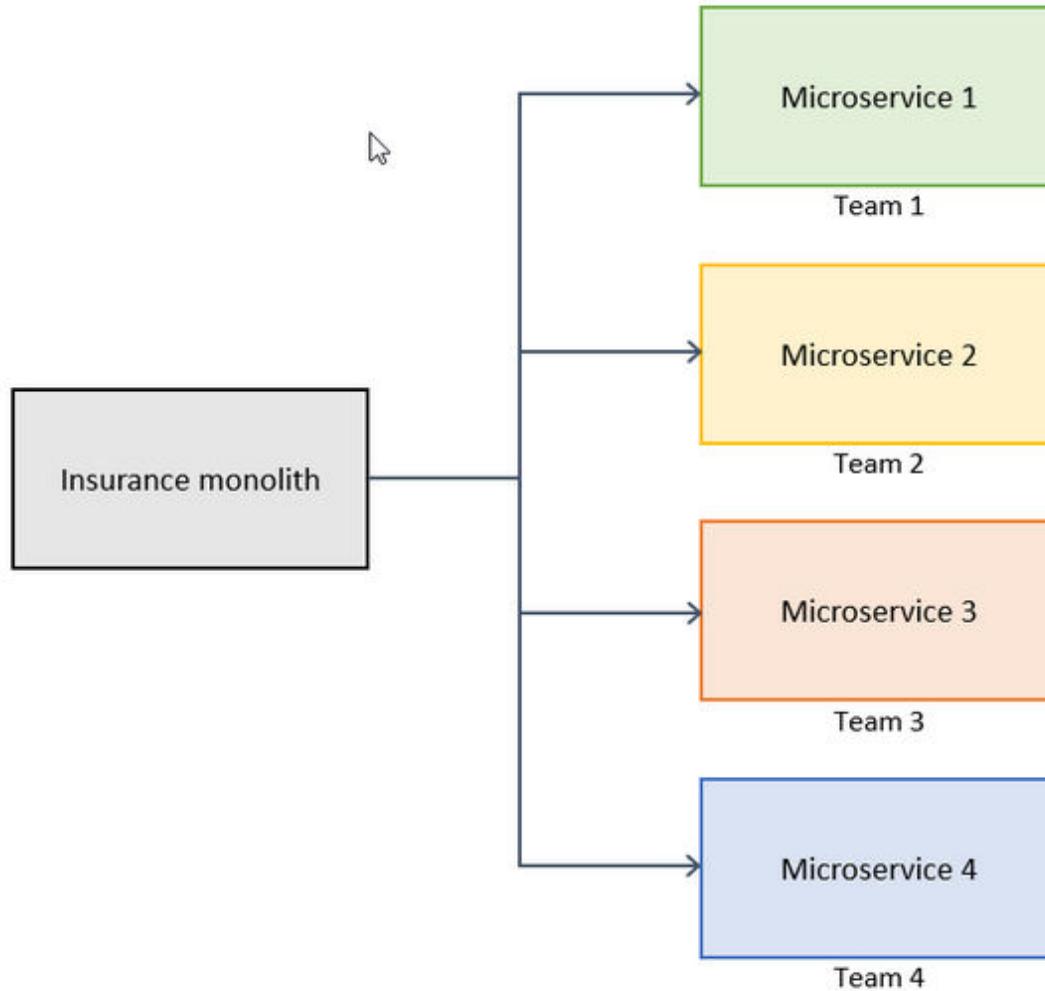
Strangler fig Pattern

Advantages	Disadvantages
<ul style="list-style-type: none">• Allows for graceful migration from a service to one or more replacement services.• Keeps old services in play while refactoring to updated versions.• Provides the ability to add new services and functionalities while refactoring older services.• The pattern can be used for versioning of APIs.	<ul style="list-style-type: none">• Isn't suitable for small systems where the complexity is low and the size is small.• Cannot be used in systems where requests to the backend system cannot be intercepted and routed.• The proxy or facade layer can become a single point of failure or a performance bottleneck if it isn't designed properly.

Service per Team Pattern

- Instead of decomposing monoliths by business capabilities or services, the service per team pattern breaks them down into microservices that are managed by individual teams.
- Each team is responsible for a business capability and owns the capability's code base.
- The team independently develops, tests, deploys, or scales its services, and primarily interacts with other teams to negotiate APIs.
- We recommend that you assign each microservice to a single team. However, if the team is large enough, multiple subteams could own separate microservices within the same team structure.

Service per Team Pattern



Service per Team Pattern

Advantages	Disadvantages
<ul style="list-style-type: none">• Teams act independently with minimal coordination.• Code bases and microservices are not shared by multiple teams.• Teams can quickly innovate and iterate on product features.• Different teams can use different technologies, frameworks, or programming languages..	<ul style="list-style-type: none">• It can be difficult to align teams to end-user functionality or business capabilities.• Additional effort is required to deliver larger, coordinated application increments, especially if there are circular dependencies between teams.

Branch by abstraction pattern

- The strangler fig pattern works well when you can intercept the calls at the perimeter of the monolith.
- However, if you want to modernize components that exist deeper in the legacy application stack and have upstream dependencies, we recommend the branch by abstraction pattern.
- This pattern enables you to make changes to the existing code base to allow the modernized version to safely coexist alongside the legacy version without causing disruption.

Branch by abstraction pattern

- To use the branch by abstraction pattern successfully, need to follow this process:
 - Identify monolith components that have upstream dependencies.
 - Create an abstraction layer that represents the interactions between the code to be modernized and its clients.
 - When the abstraction is in place, change the existing clients to use the new abstraction.
 - Create a new implementation of the abstraction with the reworked functionality outside the monolith.
 - Switch the abstraction to the new implementation when ready.
 - When the new implementation provides all necessary functionality to users and the monolith is no longer in use, clean up the older implementation.

Branch by abstraction pattern

Advantages	Disadvantages
<ul style="list-style-type: none">• Allows for incremental changes that are reversible in case anything goes wrong (backward compatible).• Lets you extract functionality that's deep inside the monolith when you can't intercept the calls to it at the edge of the monolith.• Allows multiple implementations to coexist in the software system.	<ul style="list-style-type: none">• Isn't suitable if data consistency is involved.• Requires changes to the existing system.• Might add more overhead to the development process, especially if the code base is poorly structured.

Chapter 4

Data Management, Transactional and Deployment Patterns

Objectives

Chapter 4: Data Management, Transactional and Deployment Patterns

- Understanding about the database patterns
- Learn about how to deploy microservices and its patterns
- Review some of the distributed transactional patterns for microservices

Database Patterns for Data Management

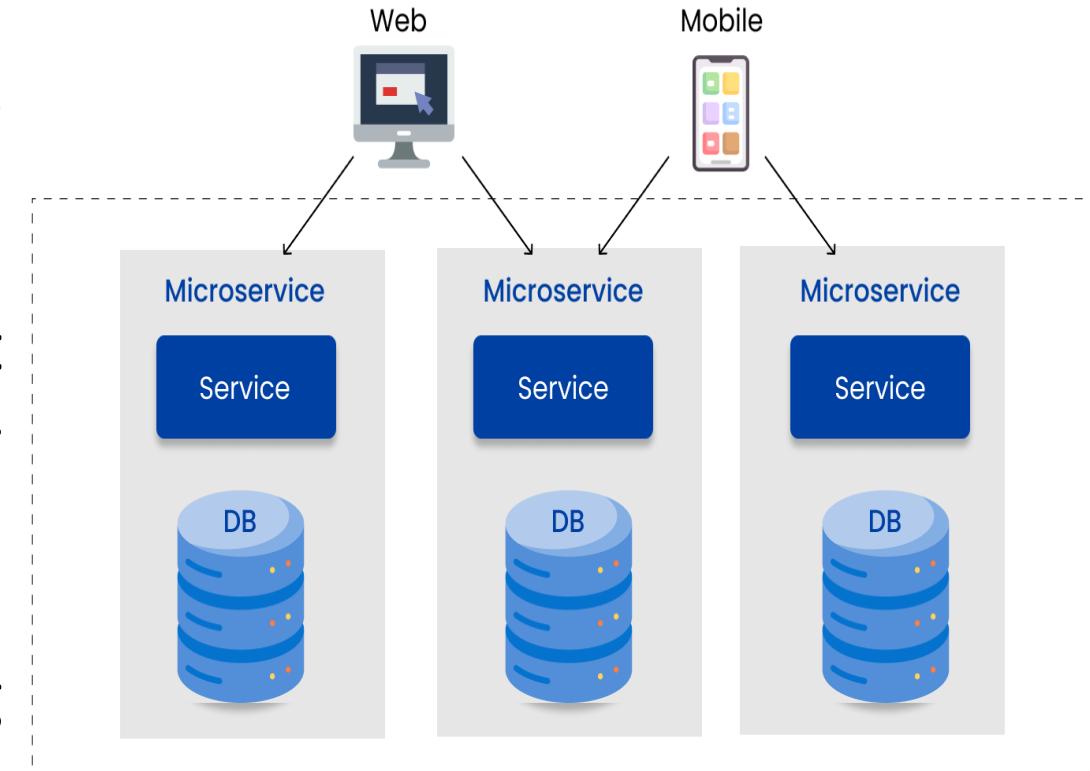
- Database design patterns play a pivotal role in shaping the microservices database architecture.
- These patterns provide the framework for whether each service should have its own dedicated database or share a common one.
- Microservices architecture breaks an application into loosely coupled services, which can be independently developed and deployed.

Database Patterns for Data Management

- Let's explore some of the key database patterns:
 - Database per service
 - Shared Database per service
 - Command Query Responsibility Segregation
 - Event Sourcing
 - Saga Pattern
 - Choreography
 - Orchestration

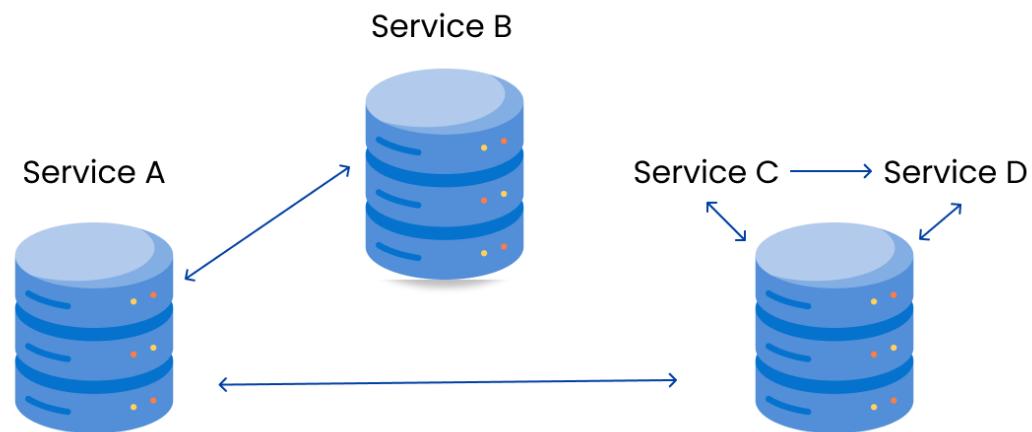
1. Database per service

- In this pattern, each microservice manages its own database. Communication between the databases of different microservices occurs via well-defined APIs.
- The database per service pattern promotes scalability, loose coupling between databases, and simplified impact analysis.



2. Shared Database per service

- Here, microservices share a common database, with each service accessing it using local ACID transactions.
- However, sharing a database can counteract some of the core benefits of microservices such as loose coupling and service independence.
- A single point of failure can crash the entire system if the shared database fails.

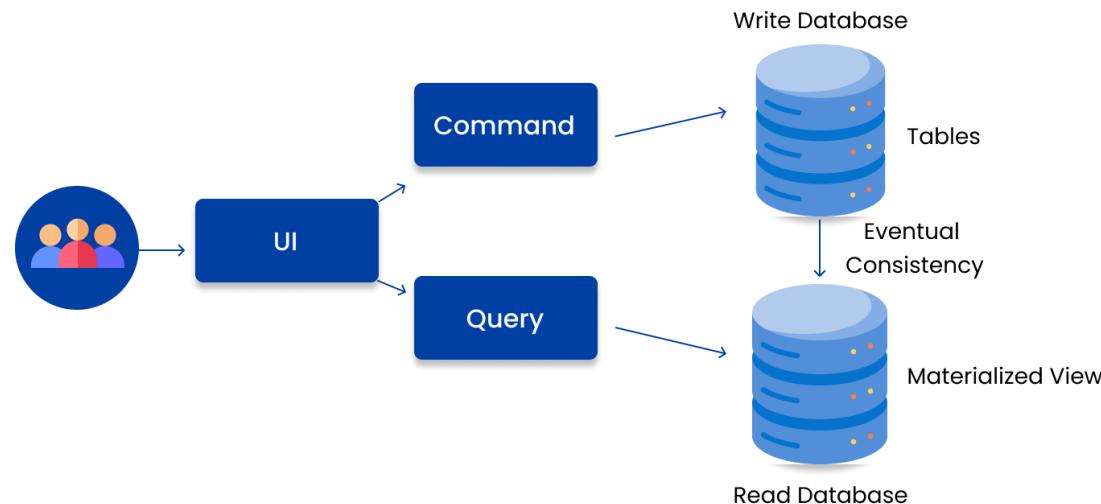


3. Command Query Responsibility Segregation (CQRS)

- The implementation of database-per-service necessitates querying that involves retrieving data from multiple services. This requires the combination of data from different services.
- The CQRS pattern suggests splitting the application into two components: the command side and the query side.
 - The command side processes Create, Update, and Delete requests.
 - The query side utilizes materialized views to handle the query component.

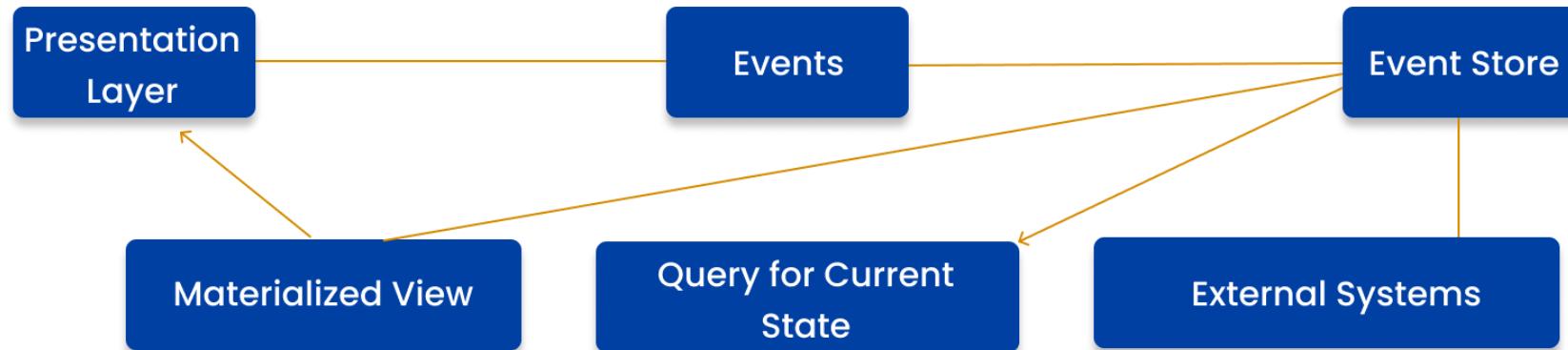
3. Command Query Responsibility Segregation (CQRS)

- The utilization of the event sourcing pattern is common practice in conjunction with it, in order to generate events for every alteration in data.
- Materialized views are maintained in a current state by subscribing to a continuous stream of events.



4. Event Sourcing

- This pattern stores the aggregate data as a series of state-altering events. Any changes or additions to the data generate a new event.
- These events can be stored in a list and replayed at a later time, thereby allowing services to subscribe to and retrieve events via APIs.

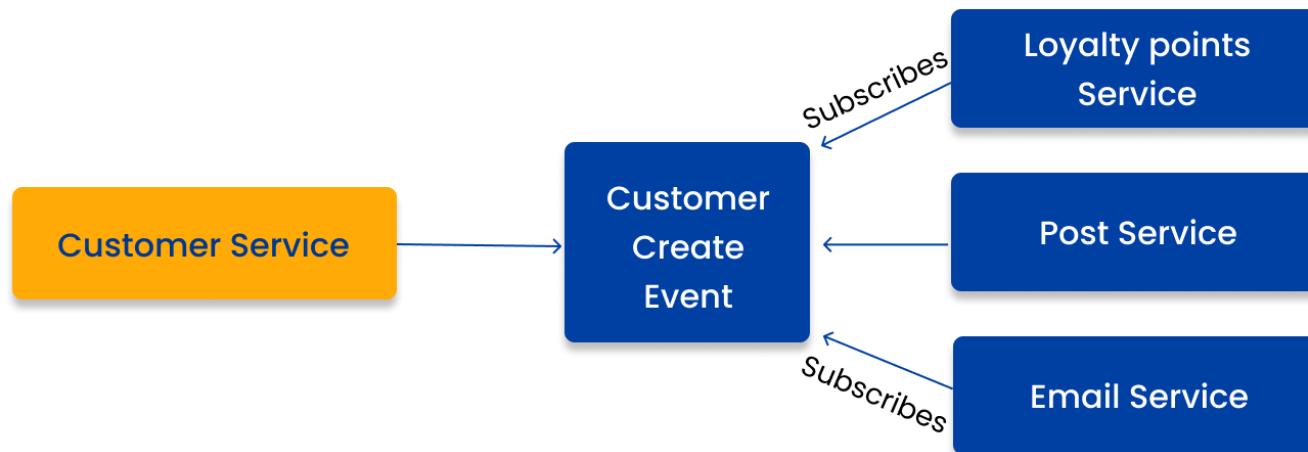


5. Saga Pattern

- To maintain data consistency across services, it is crucial to address the challenge of having separate databases for each service while dealing with business transactions that involve multiple services.
- Compensating requests are triggered by failed requests.
- There are two possible implementation methods:
 - Choreography
 - Orchestration

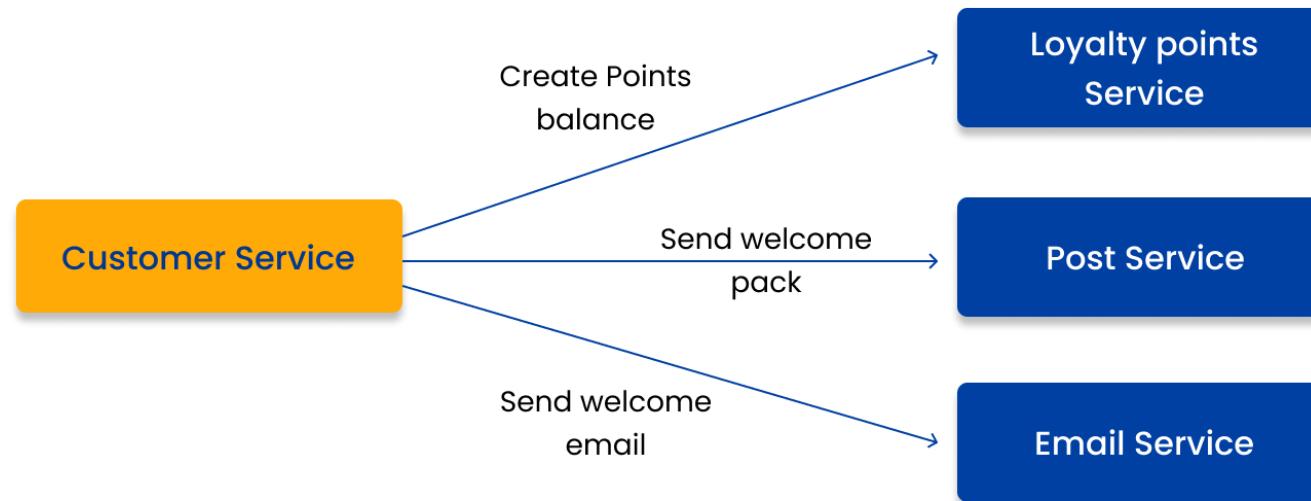
5. Saga Pattern (Choreography)

- Without central coordination, each service produces and monitors events from other services and decides whether any action is needed. Choreography facilitates information and value sharing.



5. Saga Pattern (Orchestration)

- An orchestrator sequences the business logic and makes decisions in a saga. Orchestration is used when all participants in a process are under a single control domain, such as within a single organization.

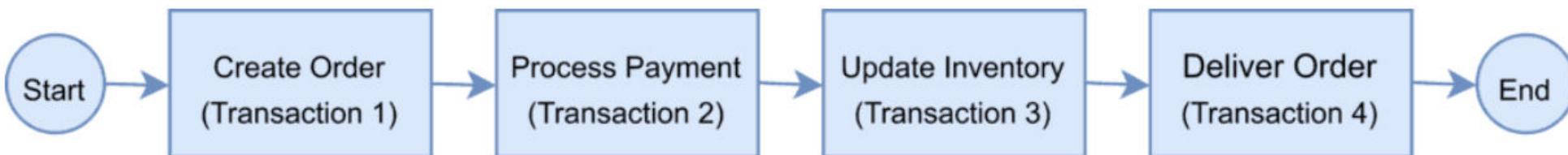


Transactional Patterns

- Distributed transactions are complex and costly because they involve coordinating multiple services, networks, and databases, which can introduce latency, failures, and inconsistencies.
- Traditional approaches, such as using a two-phase commit protocol or a distributed lock manager, are not suitable for microservices, as they create tight coupling, reduce availability, and increase the risk of deadlock and data corruption.
- Therefore, you need to adopt a different mindset and strategy for dealing with distributed transactions in microservices.

Transactional Patterns

- To demonstrate the use of distributed transactions, we'll take an example of an e-commerce application that processes online orders and is implemented with microservice architecture.
- There is a microservice to create the orders, one that processes the payment, another that updates the inventory and the last one that delivers the order.
- Each of these microservices performs a local transaction to implement the individual functionalities

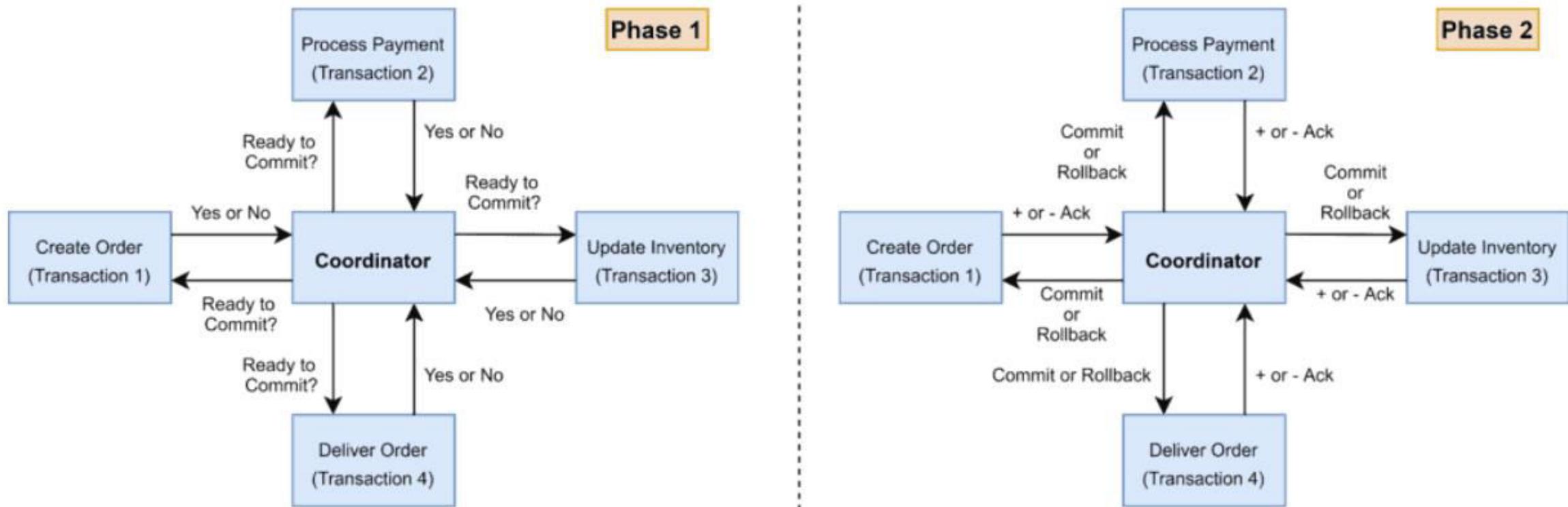


- To ensure a successful order processing service, all four microservices must complete the individual local transactions. If any of the microservices fail to complete its local transaction, all the completed preceding transactions should roll back to ensure data integrity.

Understanding Two-Phase Commit Protocol

- The Two-Phase Commit protocol (2PC) is a **widely used pattern to implement distributed transactions**. It is to use this pattern in a microservice architecture to implement distributed transactions.
- In a two-phase commit protocol, there is a coordinator component that is responsible for controlling the transaction and contains the logic to manage the transaction.
- As the name indicates, the two-phase commit protocol runs a distributed transaction in two phases:
 - **Prepare Phase** – The coordinator asks the participating nodes whether they are ready to commit the transaction. The participants returned with a *yes* or *no*.
 - **Commit Phase** – If all the participating nodes respond affirmatively in phase 1, the coordinator asks all of them to commit. If at least one node returns negative, the coordinator asks all participants to roll back their local transactions.

Understanding Two-Phase Commit Protocol



Problems in 2Pc Protocol

- Although 2PC is useful to implement a distributed transaction, it has the following shortcomings:
- The **onus of the transaction is on the coordinator node**, and it can become the single point of failure.
- All other services need to wait until the slowest service finishes its confirmation. So, the overall performance of the transaction is bound by the slowest service.
- The two-phase commit protocol is **slow by design due to the chattiness and dependency on the coordinator**. So, it can lead to scalability and performance issues in a microservice-based architecture involving multiple services.
- Two-phase commit protocol is **not supported in NoSQL databases**. Therefore, in a microservice architecture where one or more services use NoSQL databases, we can't apply a two-phase commit.

Transactional Outbox Pattern

- A common challenge that arises when handling distributed transactions in microservices is how to reliably publish events to other services without losing or duplicating them.
- One design pattern that can address this challenge is the outbox pattern.
- The outbox pattern is based on the idea of using a database table as a temporary buffer for storing events that need to be published, and using a separate process to read and send them to a message broker.
- This way, you can ensure that the events are published atomically with the local transactions, and avoid inconsistencies and failures due to network issues or broker outages, but you also need to deal with the potential for duplicate or out-of-order events.

SAGA Pattern

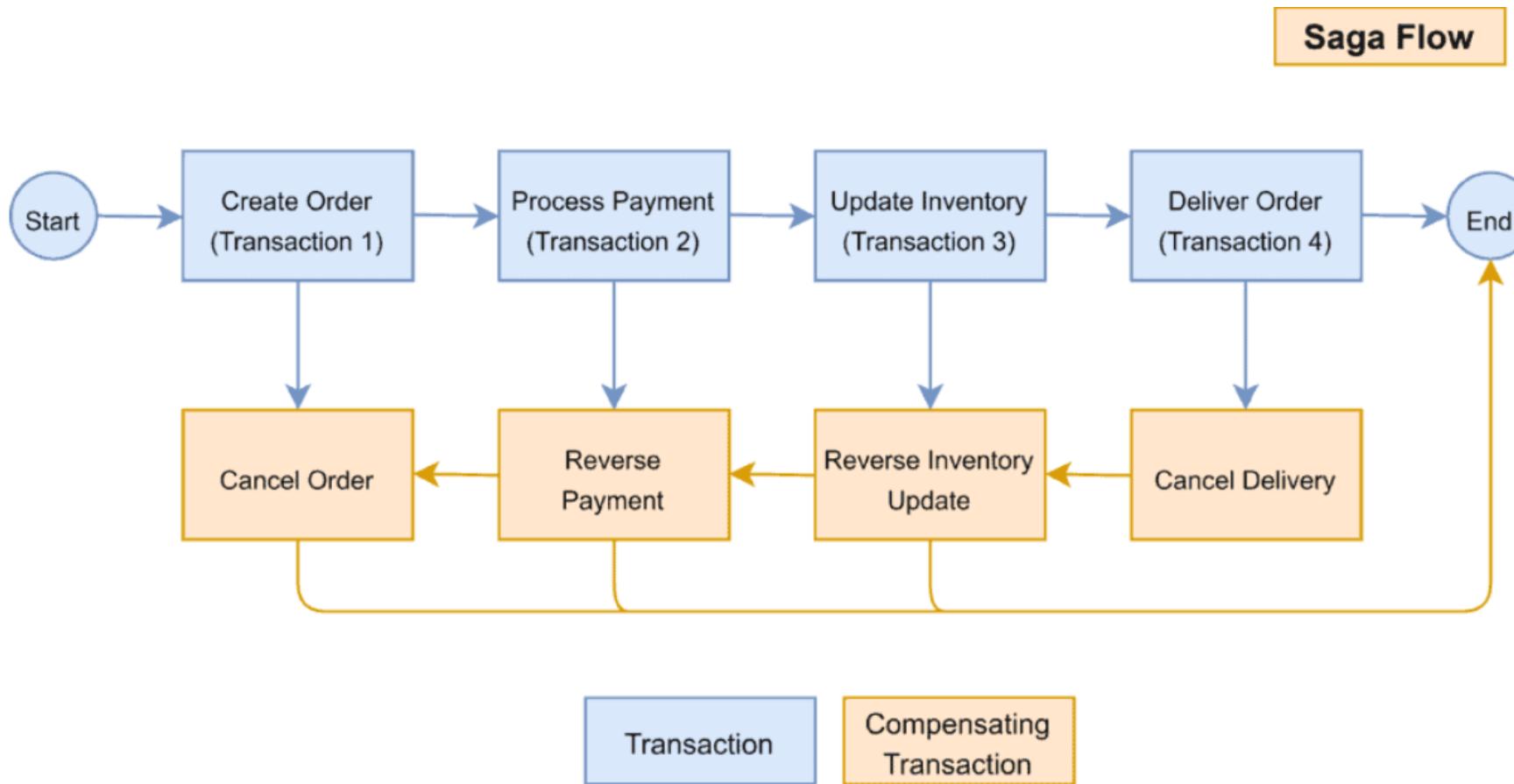
- The saga pattern is based on the idea of breaking a long-running transaction into a series of local transactions, each executed by a different service, and compensating for any failures along the way.
- A saga can be orchestrated by a central service that controls the workflow and triggers the compensation logic, or choreographed by each service that publishes and subscribes to events.
- The saga pattern allows you to achieve eventual consistency and avoid blocking and locking resources, but it also requires careful design and testing of the compensation logic and the event handling.

SAGA Pattern

- The Saga architecture pattern **provides transaction management using a sequence of local transactions.**
- A local transaction is the unit of work performed by a Saga participant. Every operation that is part of the Saga can be rolled back by a compensating transaction.
- Further, the Saga pattern guarantees that either all operations complete successfully or the corresponding compensation transactions are run to undo the work previously completed.

SAGA Pattern

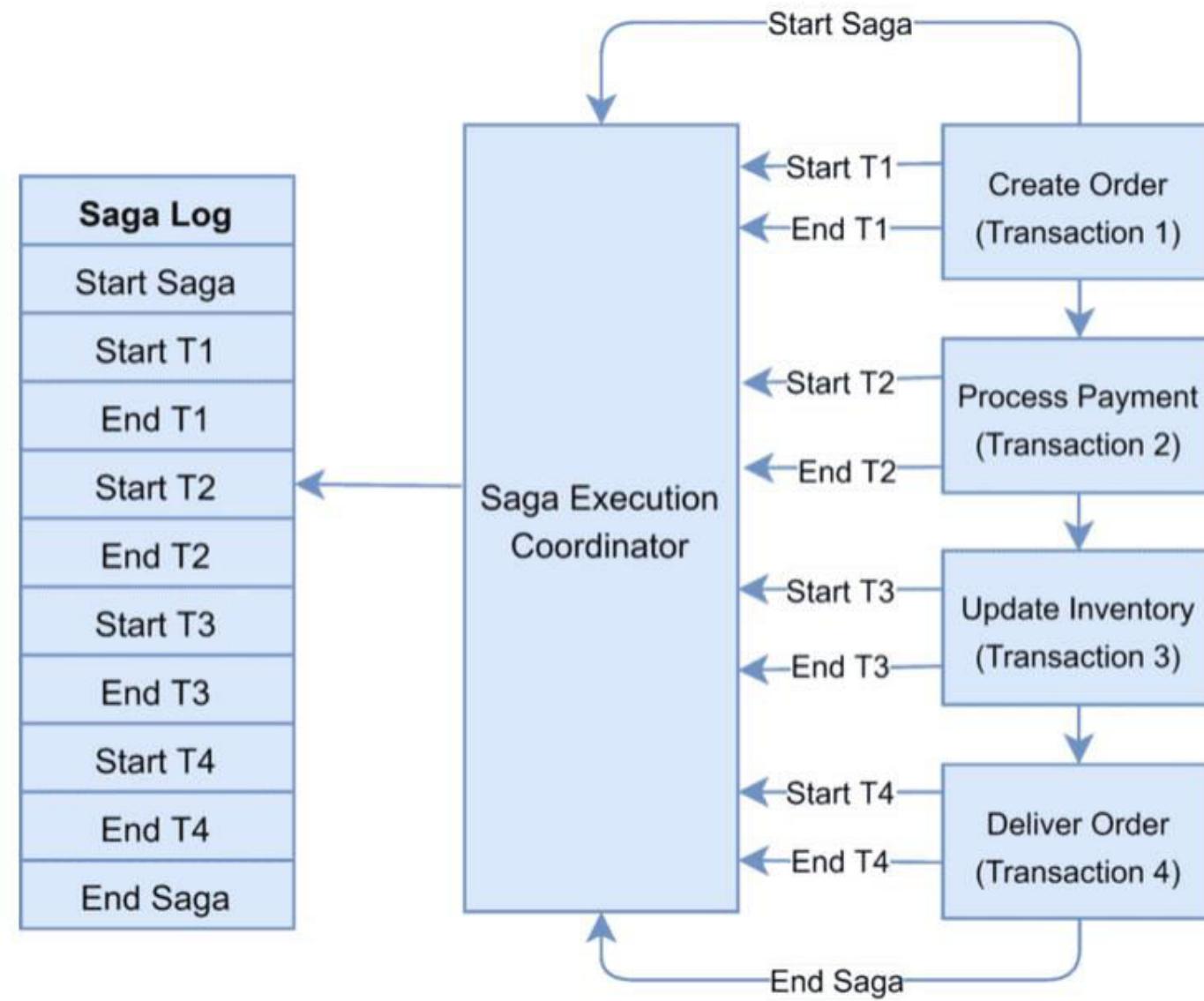
- The Saga Execution Coordinator (SEC) guarantees these principles:



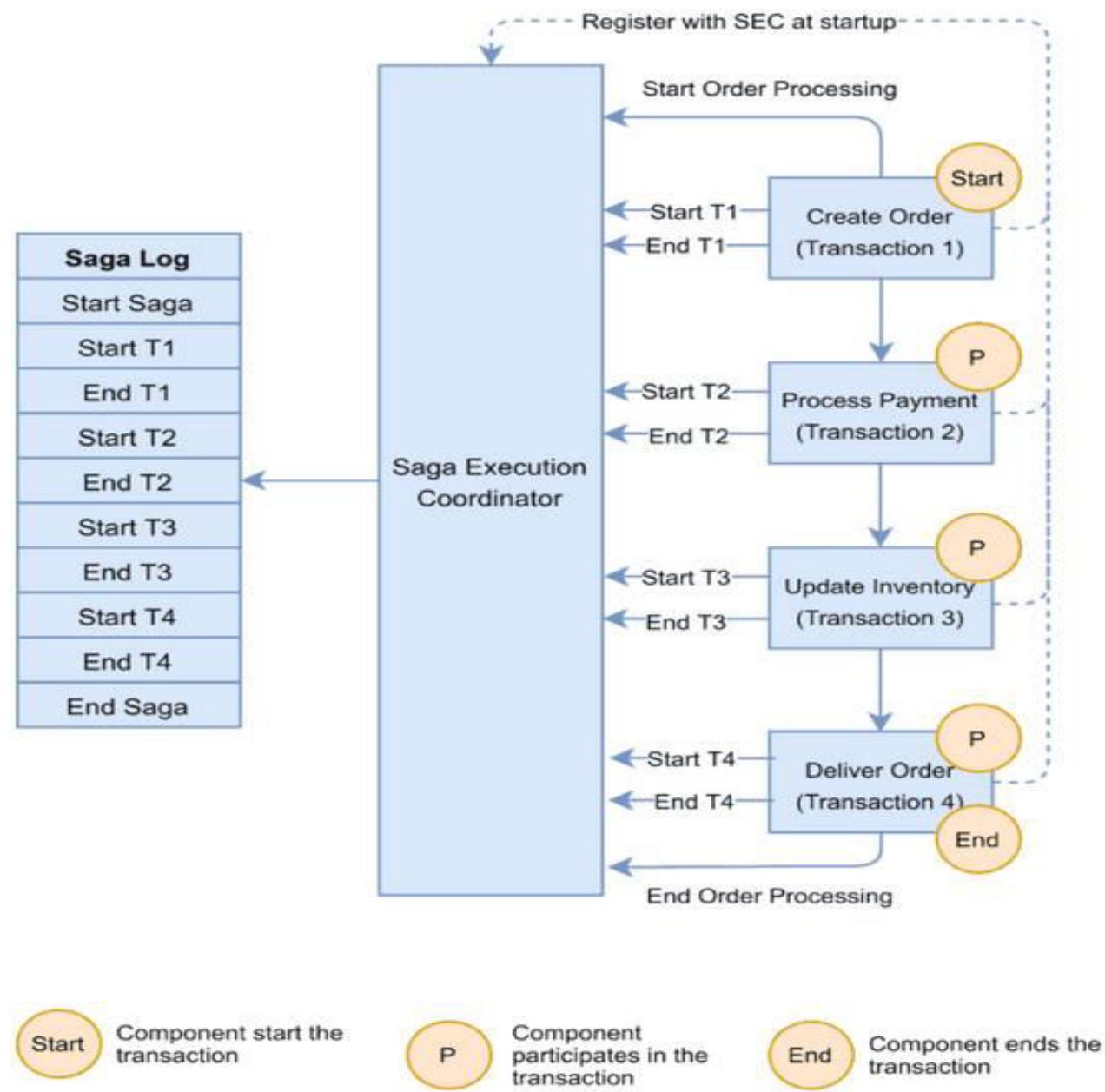
SAGA Pattern

- The **Saga Execution Coordinator** is the central component to implement a **Saga flow**. It contains a Saga log that captures the sequence of events of a distributed transaction.
- For any failure, the SEC component inspects the Saga log to identify the impacted components and the sequence in which the compensating transactions should run.
- For any failure in the SEC component, it can read the Saga log once it's coming back up.

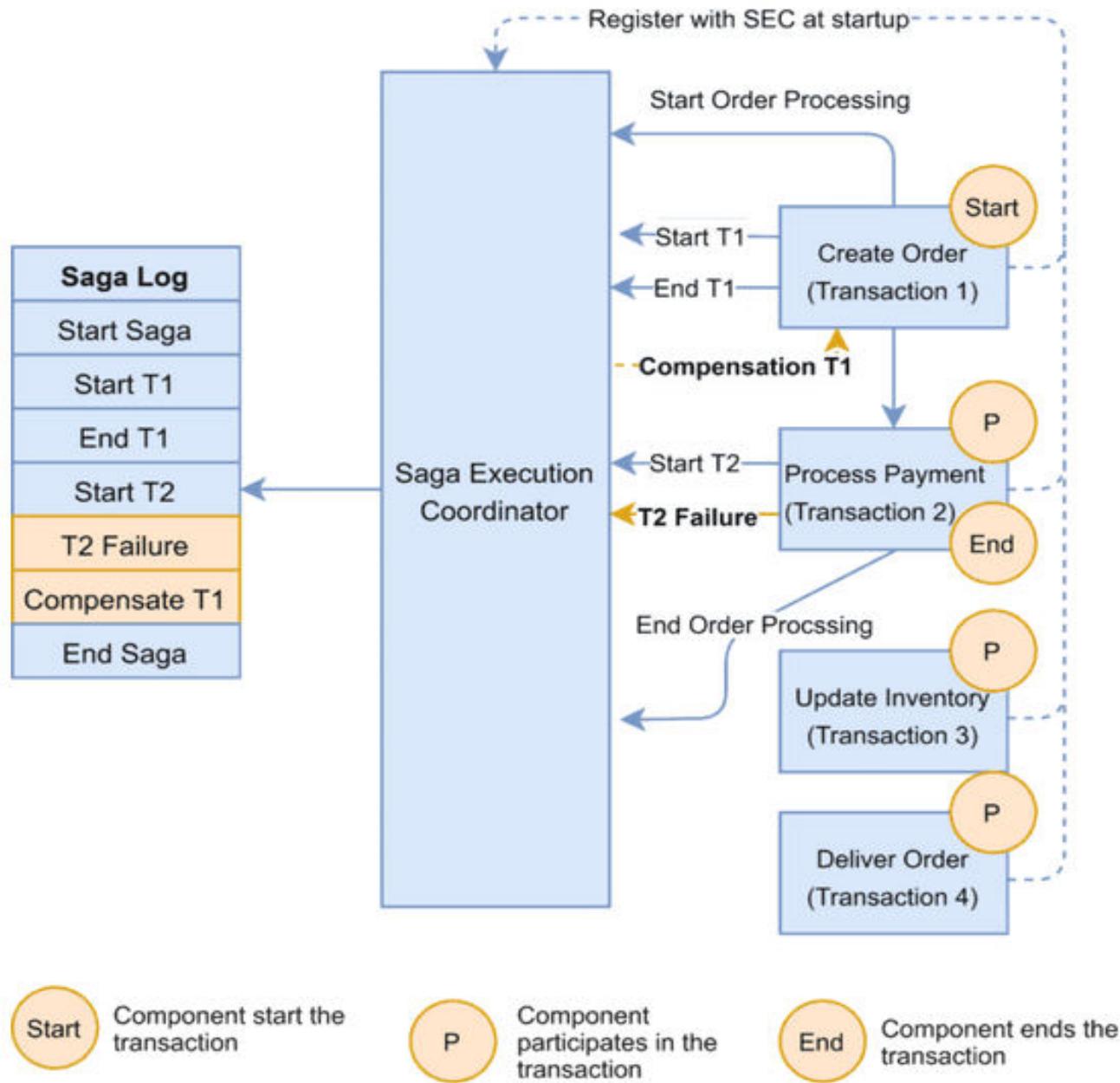
SAGA Pattern



SAGA Pattern



SAGA Pattern



Deployment Pattern

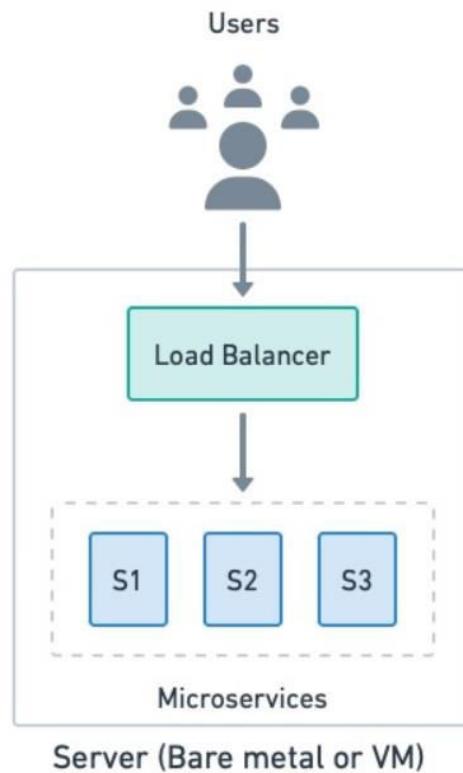
- The microservices deployment pattern is a technique for updating and modifying software components.
- A microservices deployment pattern or strategy enables easy deployments and allows you to modify microservices.
- Microservices are the most scalable way of developing software. But that means nothing unless we choose the right way to deploy microservices: processes or containers? Run on my servers or use the cloud? Do I need Kubernetes? When it comes to the microservice architecture, there is such an abundance of options and it is hard to know which is best.

The 5 ways to deploy microservices

- Microservice applications can run in many ways, each with different tradeoffs and cost structures. What works for small applications spanning a few services will likely not suffice for large-scale systems.
- From simple to complex, here are the **five ways of running microservices**:
 - **Single machine, multiple processes**: buy or rent a server and run the microservices as processes.
 - **Multiple machines, multiple processes**: the obvious next step is adding more servers and distributing the load, offering more scalability and availability.
 - **Containers**: packaging the microservices inside a container makes it easier to deploy and run along with other services. It's also the first step towards Kubernetes.
 - **Orchestrator**: orchestrators such as Kubernetes or Nomad are complete platforms designed to run thousands of containers simultaneously.
 - **Serverless**: serverless allows us to forget about processes, containers, and servers, and run code directly in the cloud.

Option 1: Single machine, multiple processes

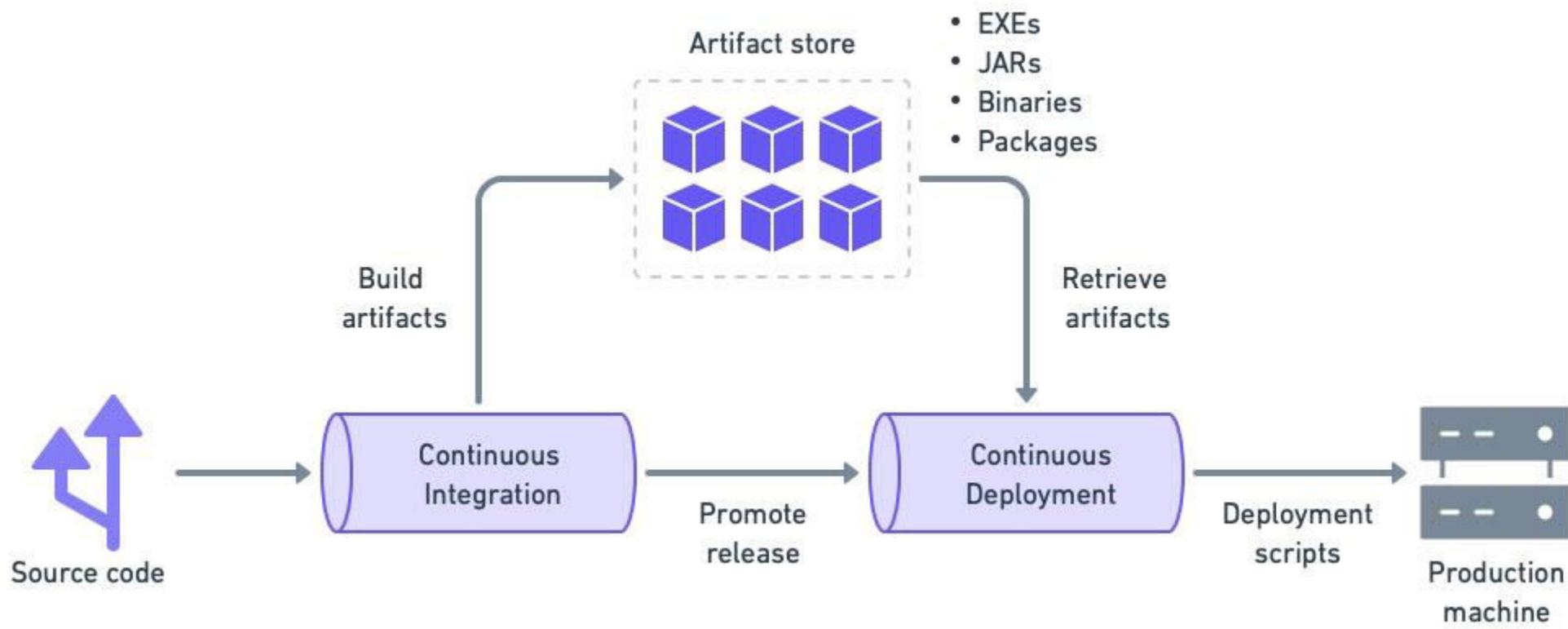
- At the most basic level, we can run a microservice application as multiple processes on a single machine. Each service listens to a different port and communicates over a loopback interface.



Option 1: Single machine, multiple processes

- **The benefits of this approach are:**
 - **Lightweight:** there is no overhead as it's just processes running on a server.
 - **Convenience:** it's a great way to experience microservices without the learning curve that other tools have.
 - **Easy troubleshooting:** everything is in the same place, so finding a problem or reverting to a working configuration in case of trouble is very straightforward, if you have continuous delivery in place.
 - **Fixed billing:** we know how much we'll have to pay each month.

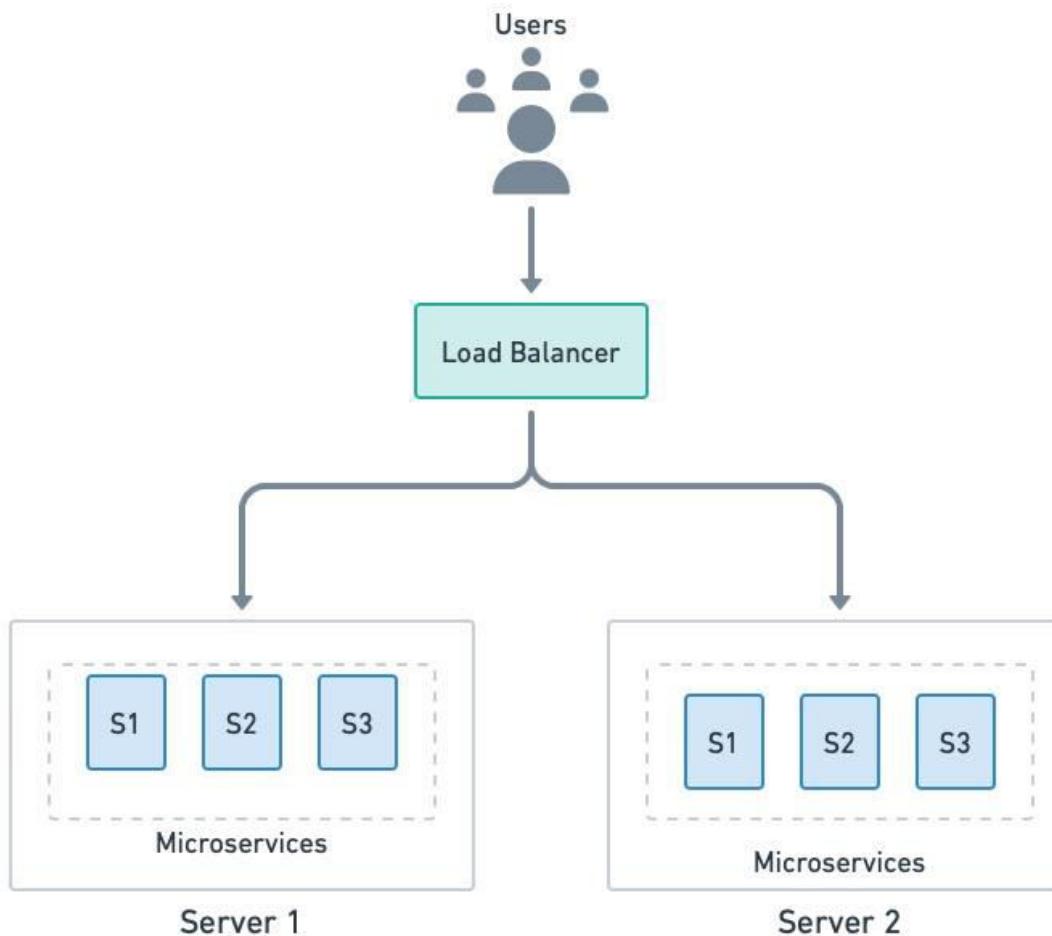
Option 1: Single machine, multiple processes



Option 2: Multiple machines and processes

- This option is essentially an upgrade of option 1. When the application exceeds the capacity of a server, we can scale up (upgrade the server) or scale sideways (add more servers).
- In the case of microservices, horizontally scaling into two or more machines makes more sense since we get improved availability as a bonus. And, once we have a distributed setup, we can always scale up by upgrading servers.

Option 2: Multiple machines and processes



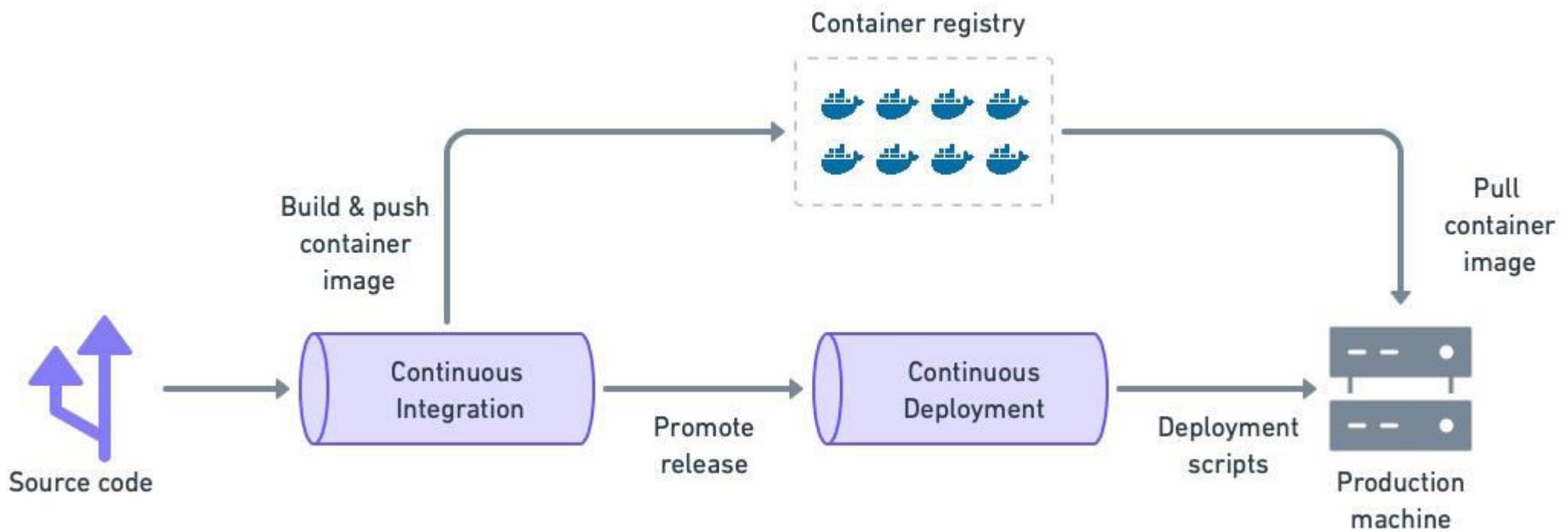
Option 3: Deploy microservices with containers

- While running microservices directly as processes is very efficient, it comes at a cost.
- The server must be meticulously maintained with the necessary dependencies and tools.
- A runaway process can consume all the memory or CPU.
- Deploying and monitoring the microservices is a brittle process.
- All these shortcomings can be mitigated with containers. Containers are packages that contain everything a program needs to run.
- A container image is a self-contained unit that can run on any server without having to install any dependencies or tools first (other than the container runtime itself).

Option 3: Deploy microservices with containers

- The benefits of this approach:
 - **Isolation:** contained processes are isolated from one another and the OS. Each container has a private filesystem, so dependency conflicts are impossible (as long as you are not abusing volumes).
 - **Concurrency:** we can run multiple instances of the same container image without conflicts.
 - **Less overhead:** since there is no need to boot an entire OS, containers are much more lightweight than VMs.
 - **No-install deployments:** installing a container is just a matter of downloading and running the image. There is no installation step required.
 - **Resource control:** we can put CPU and memory limits on containers so they don't destabilize the server.

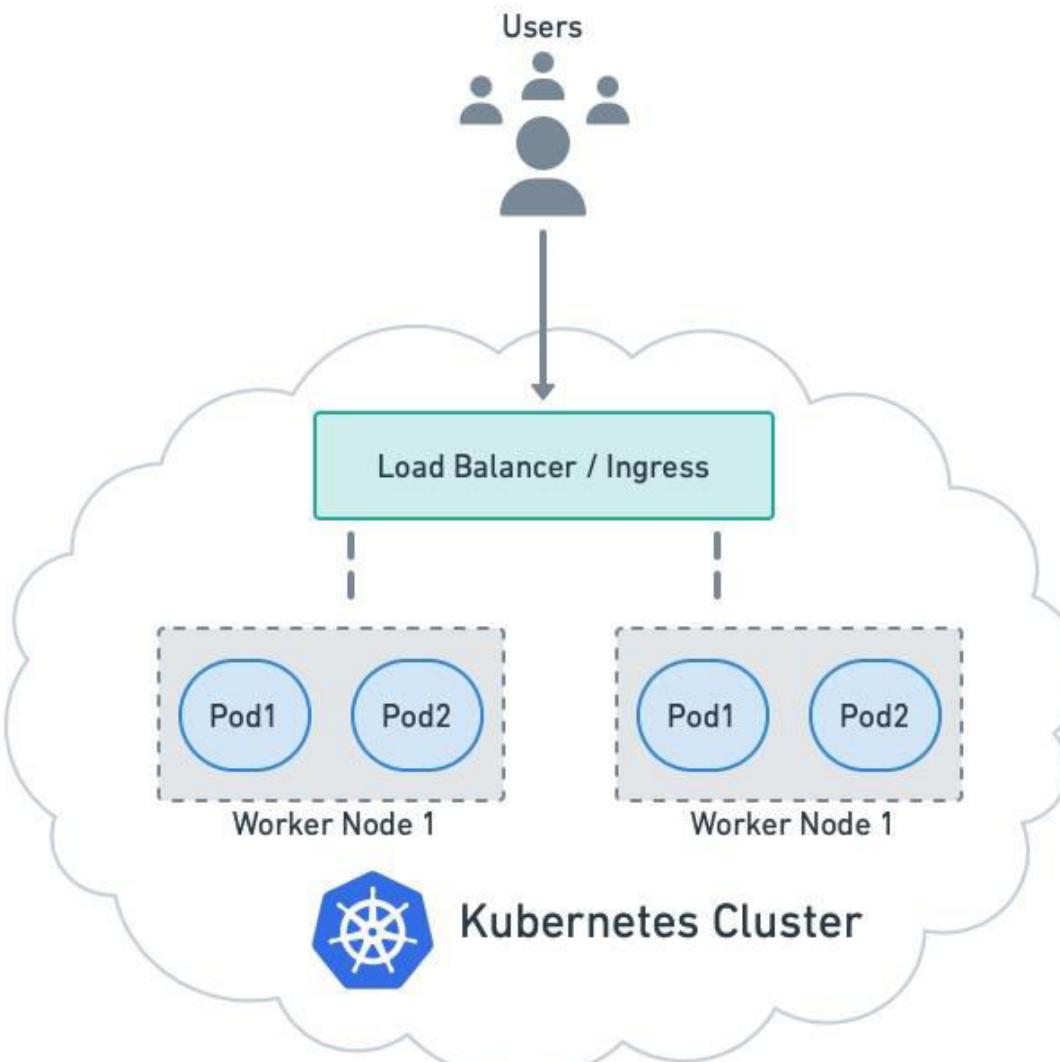
Option 3: Deploy microservices with containers



Option 4: Orchestrators

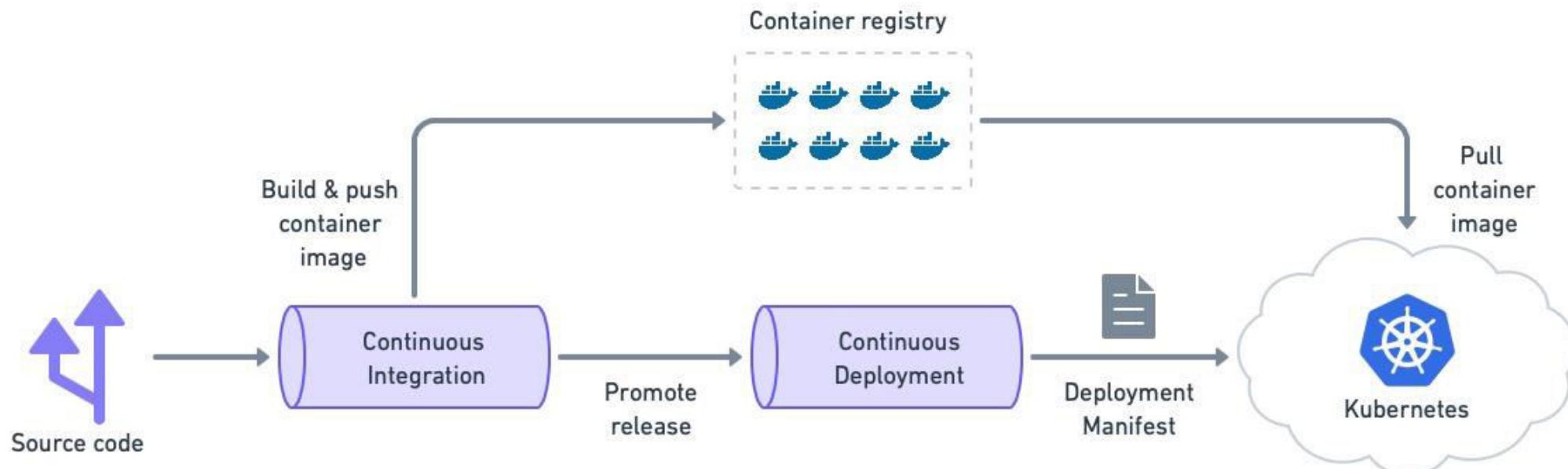
- Orchestrators are platforms specialized in distributing container workloads over a group of servers.
- The most well-known orchestrator is Kubernetes, a Google-created open-source project maintained by the Cloud Native Computing Foundation.
- Orchestrators provide, in addition to container management, extensive network features like routing, security, load balancing, and centralized logs — everything you may need to run a microservice application.

Option 4: Orchestrators



Option 4: Orchestrators

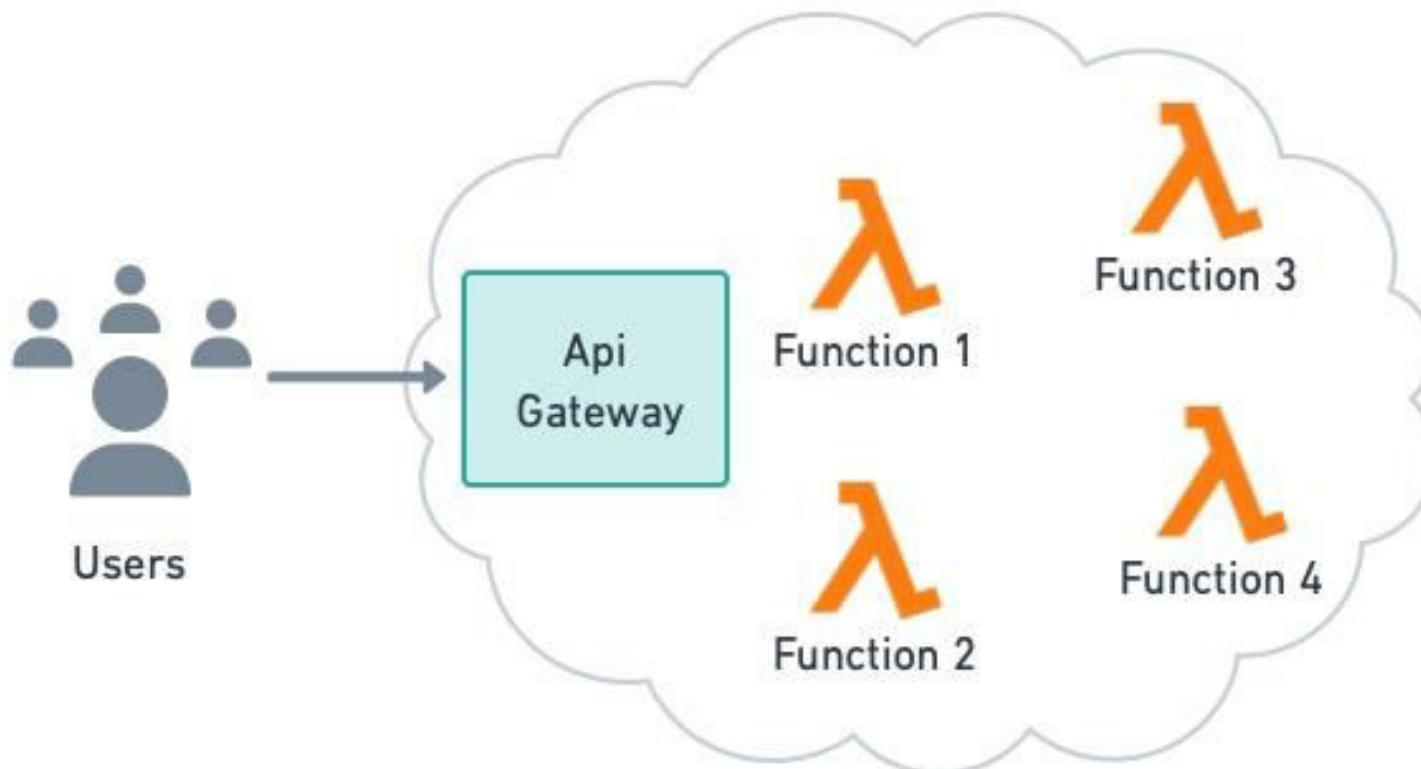
- Kubernetes is supported by all cloud providers and is the *de facto* platform for microservice deployment. As such, you might think this is the absolute best way to run microservices.



Option 5: Deploy microservices as serverless functions

- Serverless functions deviate from everything else we've discussed so far. Instead of servers, processes, or containers, we use the cloud to simply run code on demand.
- Serverless offerings like AWS Lambda and Google Cloud Functions handle all the infrastructure details required for scalable and highly-available services, leaving us free to focus on coding.

Option 5: Deploy microservices as serverless functions



Deployment Patterns

- If the application is mature and spans many services, you will require something more robust such as managed containers or serverless, and perhaps Kubernetes later on as your application grows.
- Nothing prevents you from mixing and matching different options. In fact, most companies use a mix of bare-metal servers, VMs, and Kubernetes.
- A combination of solutions like running the core services on Kubernetes, a few legacy services in a VM, and reserving serverless for a few strategic functions could be the best way of taking advantage of the cloud at every turn.

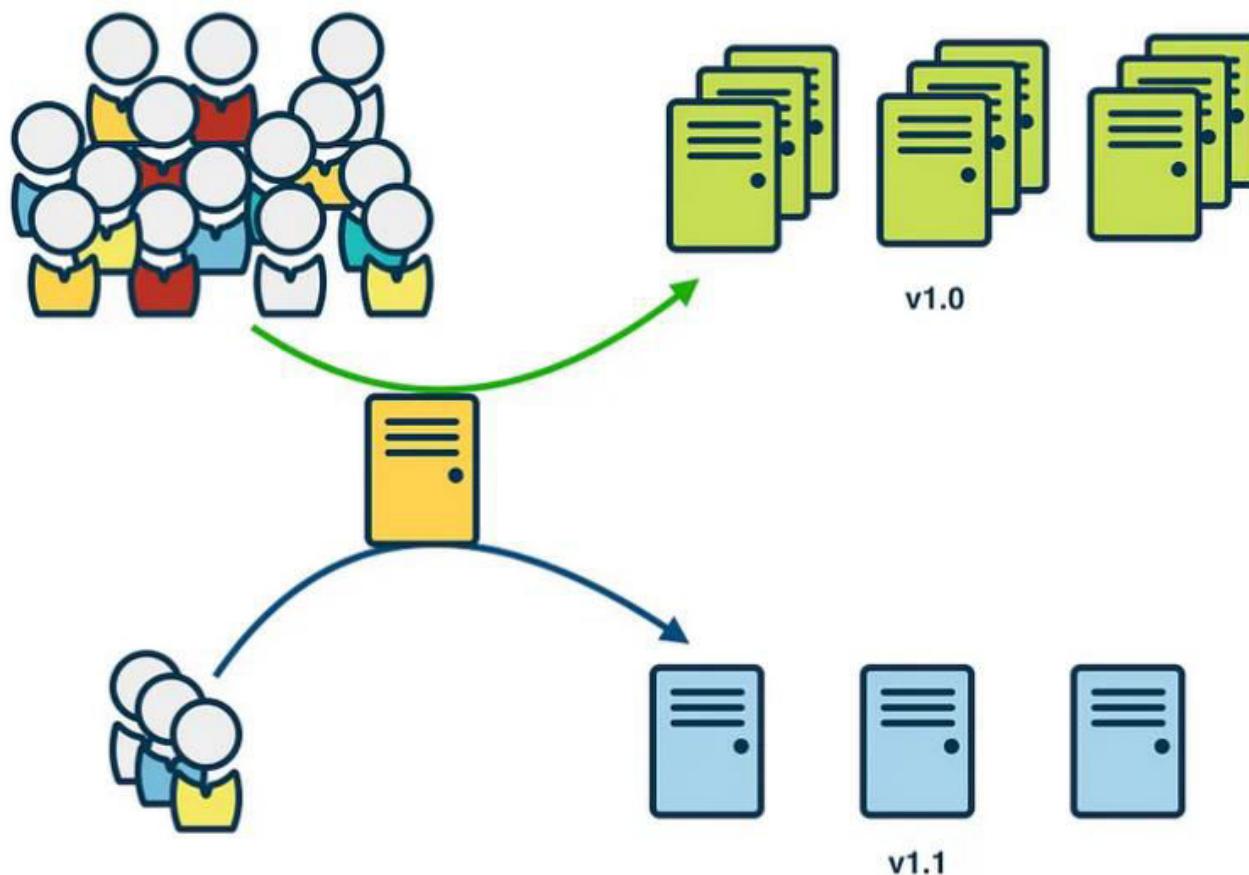
Deployment Patterns

- The microservices deployment pattern is a technique for updating and modifying software components. A microservices deployment pattern or strategy enables easy deployments and allows you to modify microservices.
- The following subsections list the microservice deployment patterns that help improve microservice availability:
 - Canary Deployment
 - Blue/Green Deployment
 - Dark Launching
 - A/B Testing

1. Canary Deployment

- A “canary deployment” is a method of spotting possible issues before they affect all consumers.
- Before making a new feature available to everyone, the plan is to only show it to a select group of users.
- In a canary release, we keep an eye on what transpires after the feature is made available. If there are issues with the release, we fix them.
- We transfer the canary release to the actual production environment once its stability has been established.

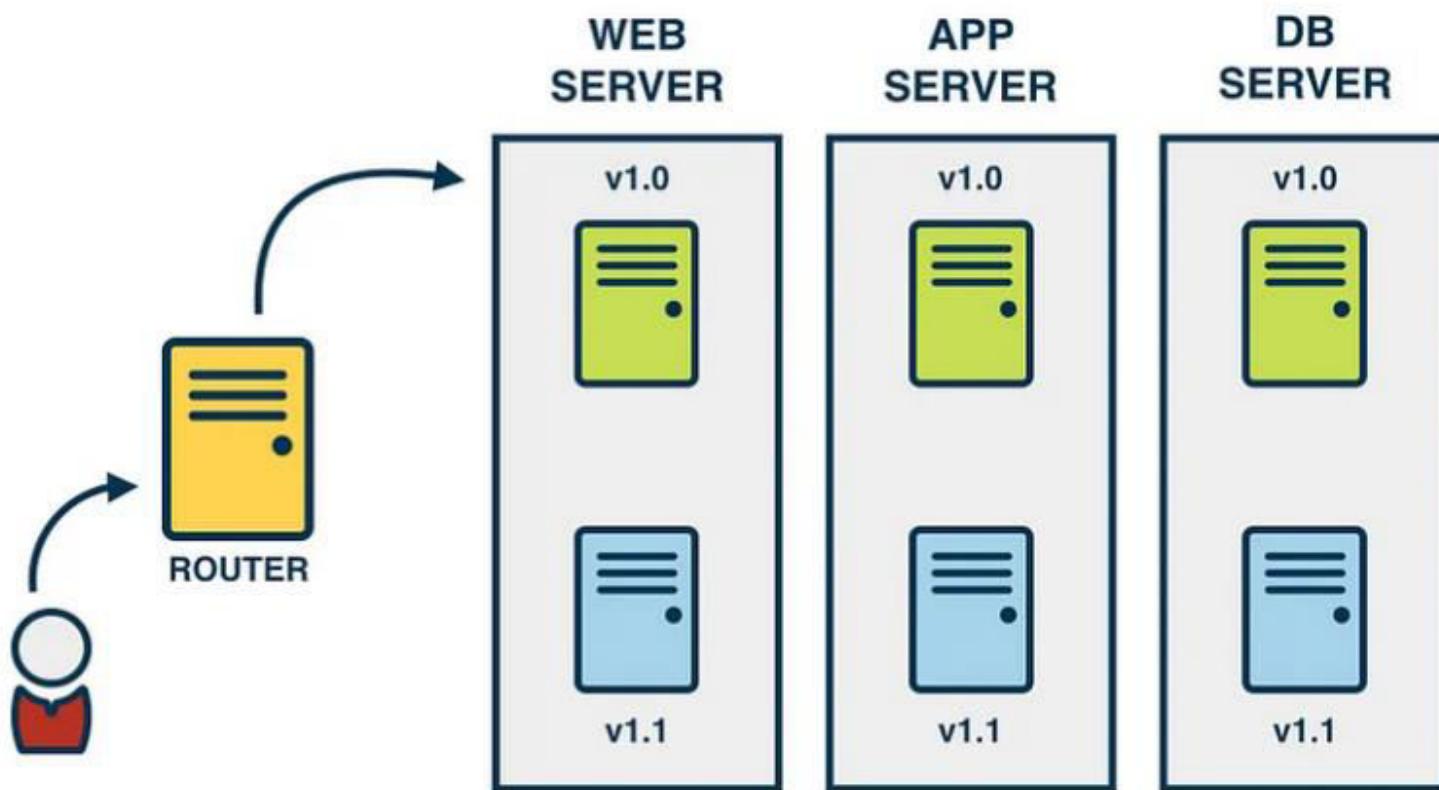
1. Canary Deployment



2. Blue/Green Deployment

- Blue/green is a technique for deployments where the existing running deployment is left in place. A new version of the application is installed in parallel with the existing version.
- When the new version is ready, cut over to the new version by changing the load balancer configuration.
- This makes rollback really simple and gives time to make sure that the new version works as expected before putting it live.

2. Blue/Green Deployment

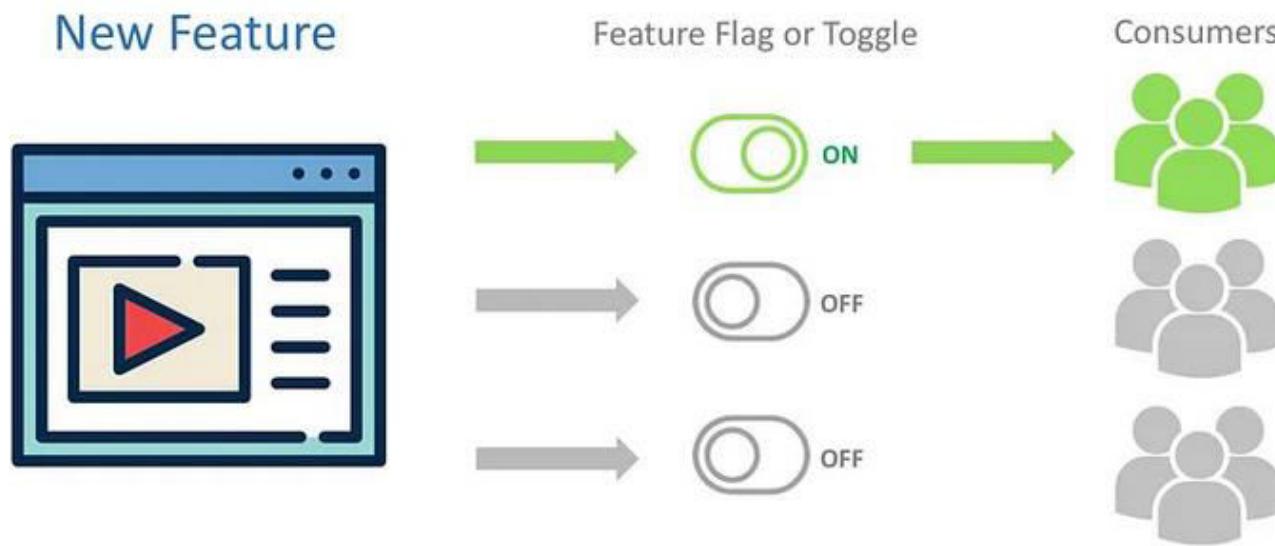


3. Dark Launching

- A dark launch is a technique that deploys updates to microservices catering to a small percentage of the user base.
- It does not affect the entire system. When you dark launch a new feature, you'll initially hide it from most end users.
- *For example, a new feature is being added to a social media platform. Instead of announcing the feature to all users, the developers may do a dark launch, where the feature is made available to a small group of users without any notification. This allows the developers to gather feedback and test the effectiveness of the feature without any potential bias from users knowing that they are participating in a test.*

3. Dark Launching

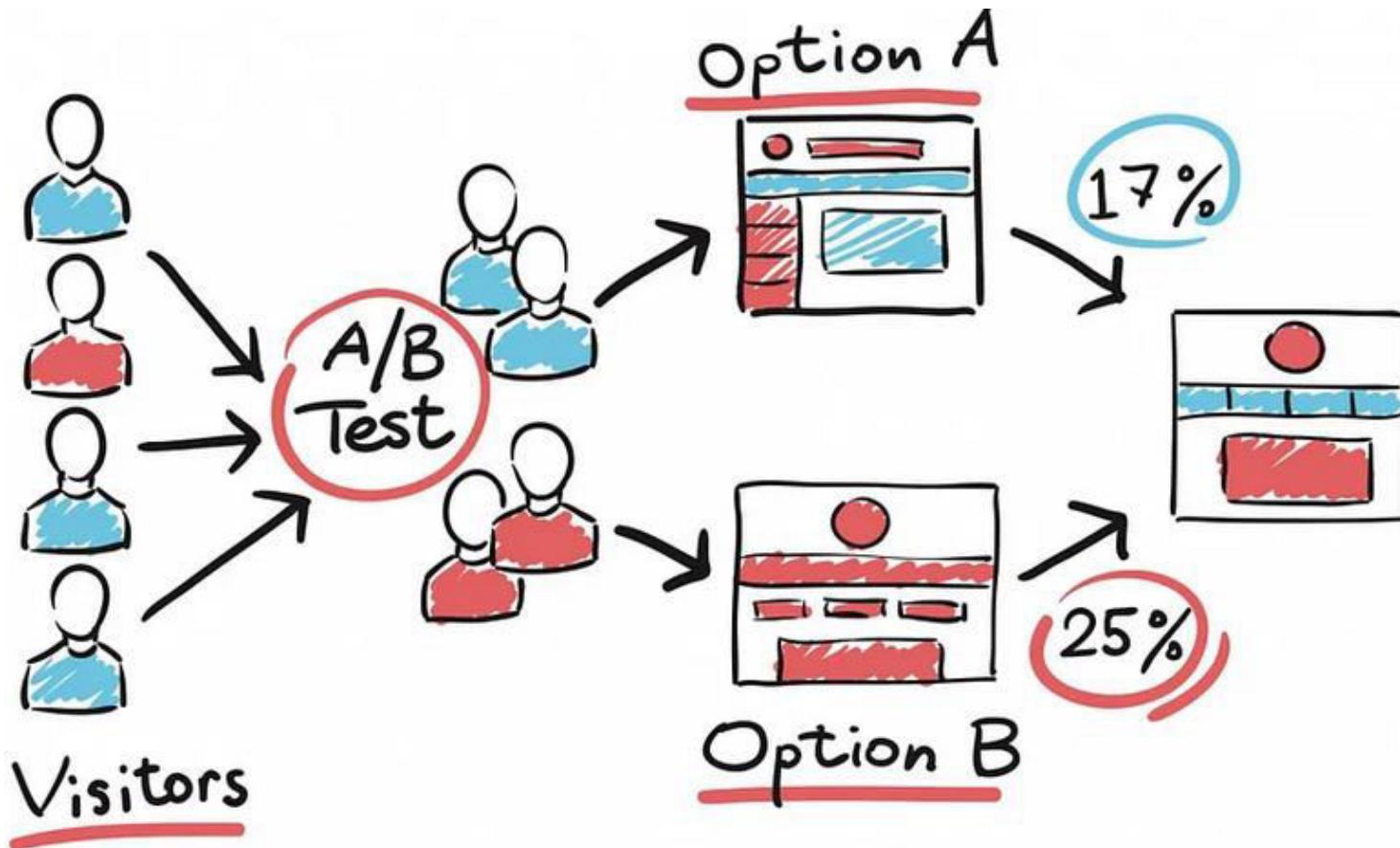
- Feature toggles — also known as feature flags — allow you to further decouple the deployment of different software versions from the release of features to users. You can deploy new versions of an application as often as needed, with certain features disabled: releasing a feature to users is simply a matter of toggling it “on.”



4. A/B Testing

- Two versions of an app are compared using A/B testing to see which one performs better.
- An experiment is like A/B testing. In A/B testing, we present users with two or more page versions at random.
- Then, we use statistical analysis to determine which variant is more effective in achieving our objectives.

4. A/B Testing



Chapter 5

Containerization using Docker

Objectives

Chapter 5: Containerization using Docker

Overview about the containerization

- Learn about the Docker Container Runtime Engine
- Learn about the architecture and benefits of Kubernetes Orchestration

Cloud-Native Principles

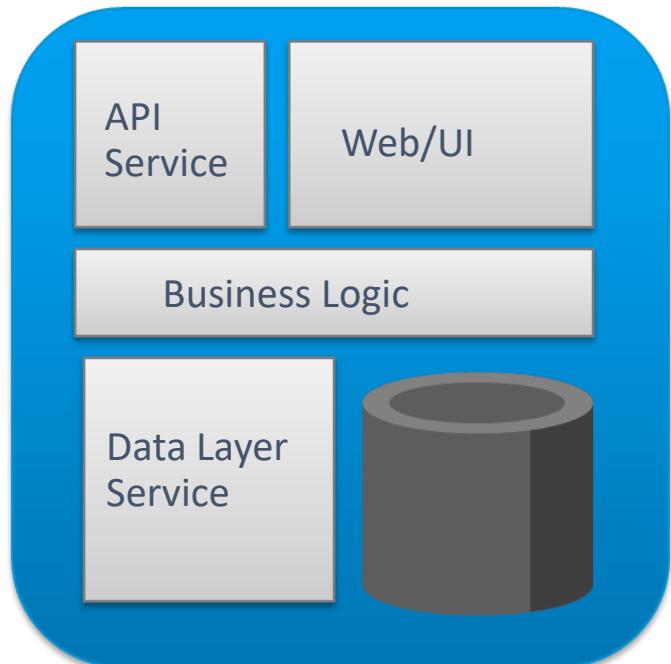
- Cloud-native principles apply to containers:
 - Isolated unit of work that does not require OS dependencies
 - Actively scheduled and managed by an orchestration process
 - Loosely coupled from any dependencies
 - Use microservices

Container Benefits

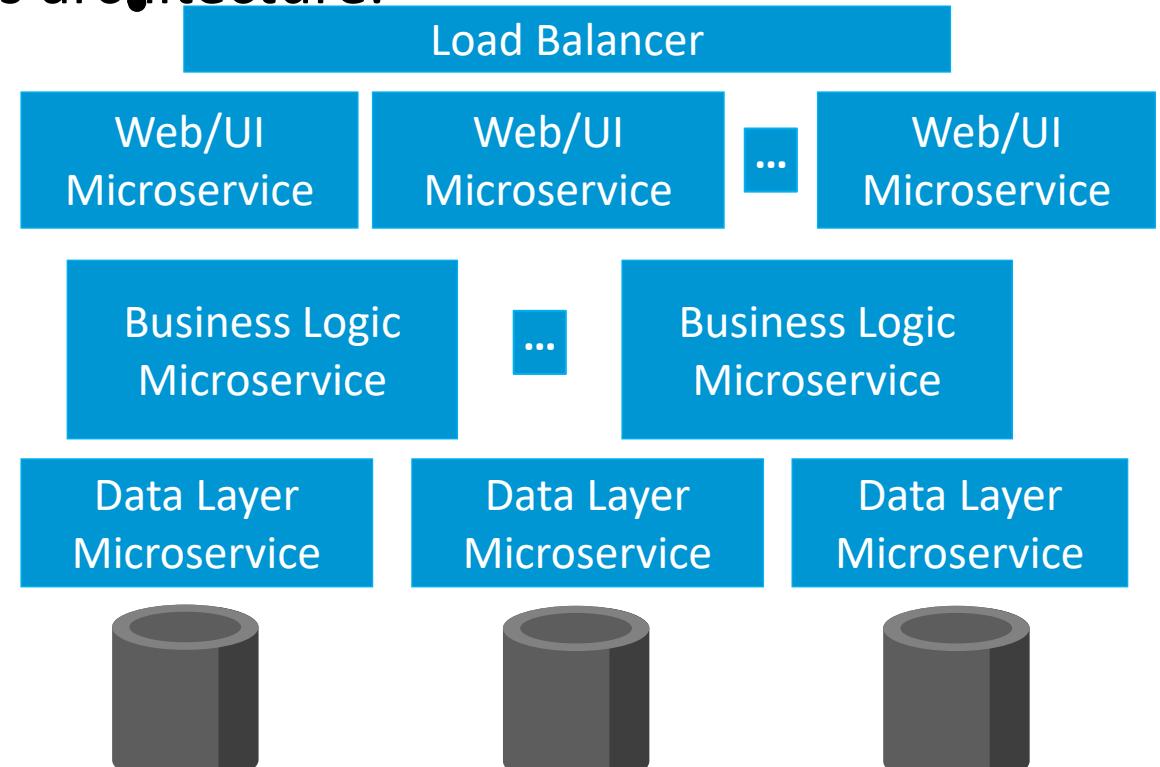
- Containers provide the following benefits:
 - Velocity
 - Portability
 - Reliability
 - Efficiency
 - Self-service
 - Isolation

Containers and Microservices

The characteristics of containers (lightweight, easily packaged, can run anywhere) align with the goals of the microservices architecture.



Monolithic Application

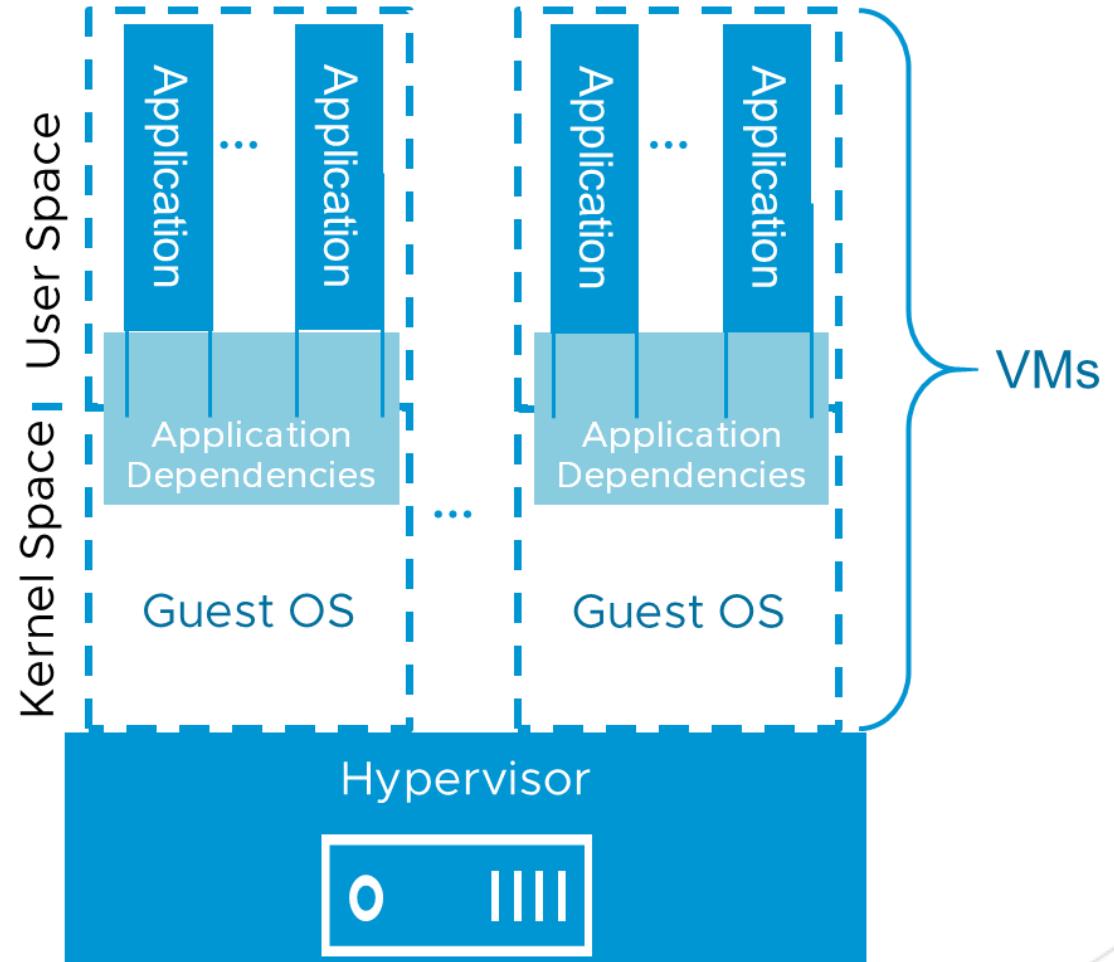


Application as Microservices

Applications in Virtual Machines at Runtime

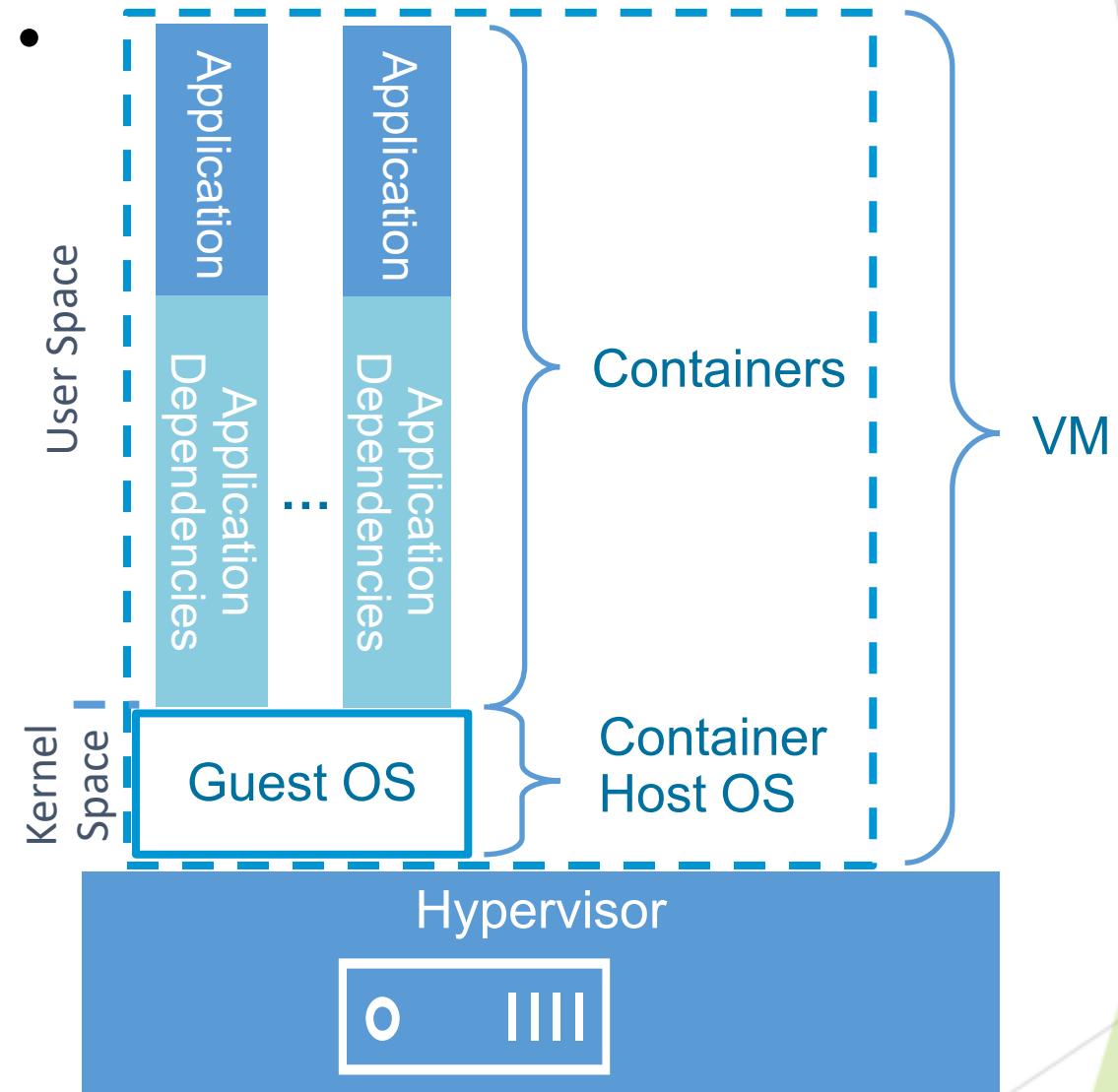
Virtual machines encapsulate a full OS with the following components:

- Running processes (such as applications)
- Memory management
- Device drivers
- Daemons
- Libraries



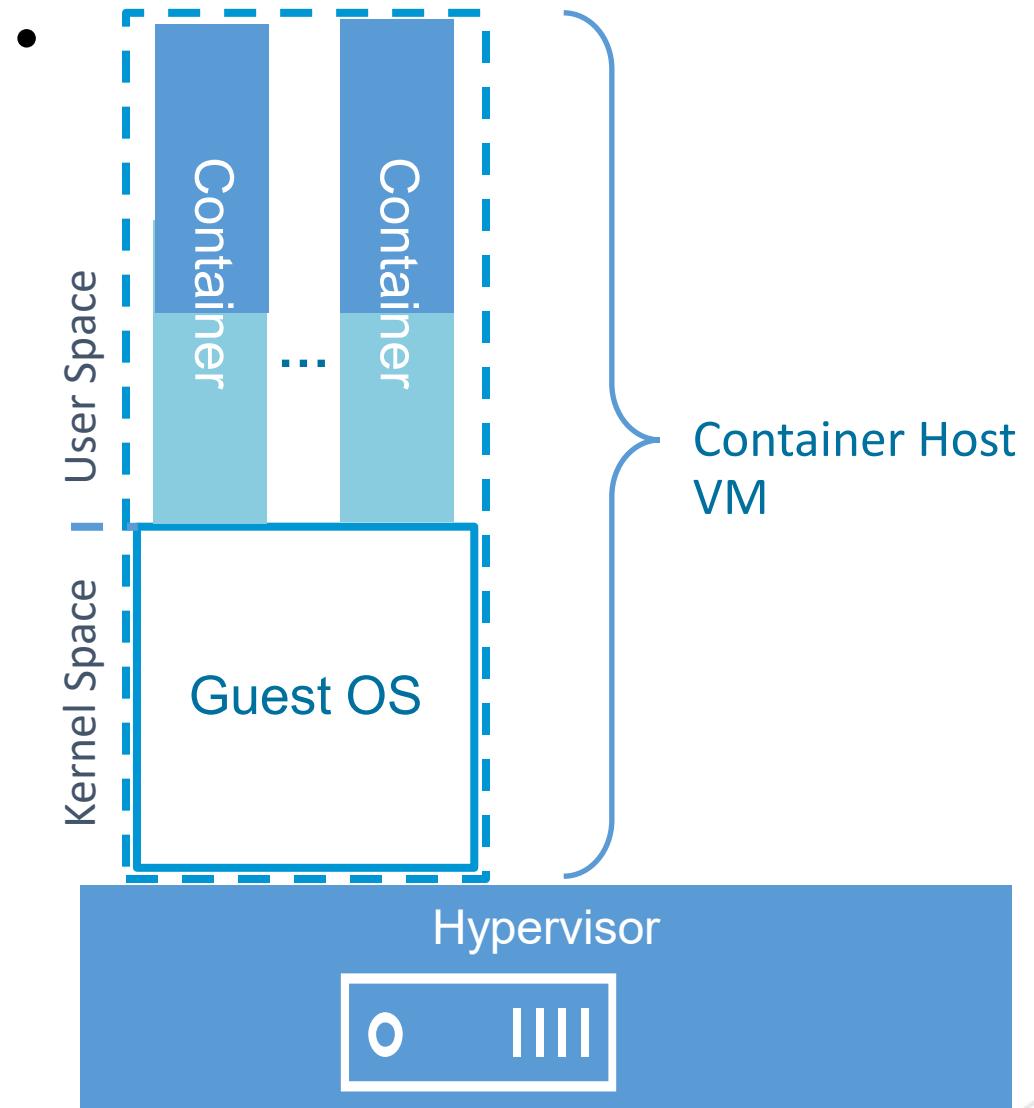
Applications in Containers at Runtime

- Containers are the encapsulation of an application process with the application dependencies.
- Containers are ultraportable. A container can run on any container host with the same operating system kernel that is specified by that container.
- The word Docker is often used as a synonym for many aspects of container technologies.



Container Hosts

- The container host runs the operating system on which the containers run.
- Using virtual machines as container hosts has many advantages:
 - Flexibility
 - Scalability
 - Security



About Container Hosts

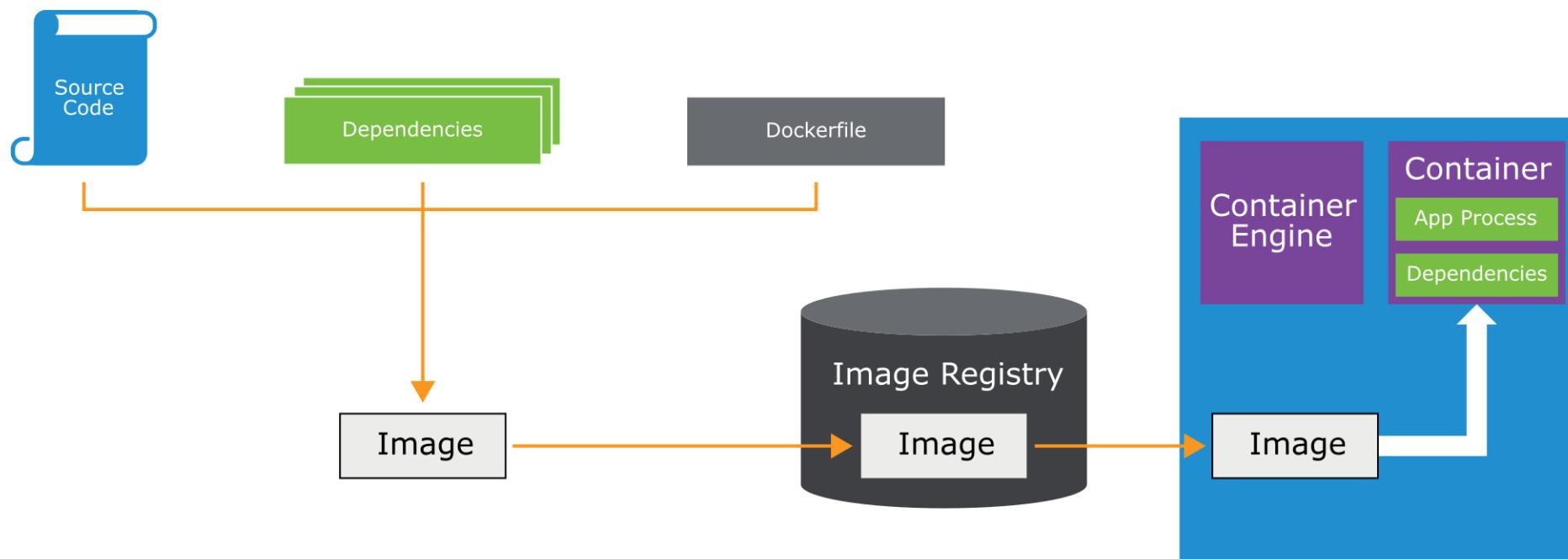
Container hosts can be of the following types:

- Standard OS with a container engine installed:
 - Ubuntu with Docker or another container engine
- OS that is developed specifically with containers in mind:
 - Photon
 - CoreOS
- Container hosts can be a virtual machine or a physical machine (bare metal):
 - Using VMs has many benefits, such as easy management and scalability.

Typical Container Workflow

A container workflow includes these steps:

1. Build an image from the source code and dependencies.
2. Send the image to the image registry.
3. Take the image from the image registry.
4. Run the image as a container.

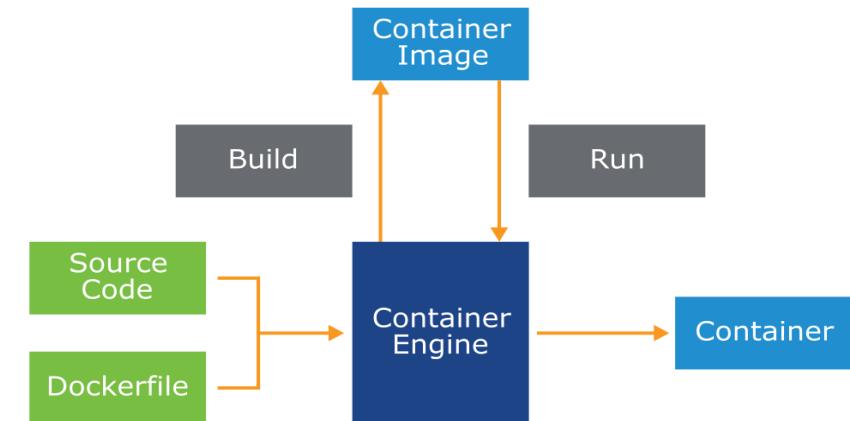


About Container Engines

A container engine is a control plane installed on each container host. It manages the containers on that host.

Container engines work in the following ways:

- Build container images from the source code (for example, Dockerfile).
- Alternatively, start container images from a repository.
- Create running containers based on a container image.
- Commit a running container to an image.
- Save an image and push it to a repository.
- Stop and remove containers.
- Suspend and restart containers.
- Report container status.



The Dockerfile and Programmatic Construction

- VM creation is not guaranteed to be scripted or reproducible.
- Dockerfile is a plaintext file that clearly defines each stage that is required to build a container image.
- Dockerfile produces a consistent and reproducible container image
- You run the docker build command to create an image from Dockerfile.
- You run the docker run command to create and start a container.

```
FROM ubuntu:18.04
RUN apt-get update && apt-get install -y php=8.0.8 nginx=1.20.1
COPY ./index.php /var/www/html/index.php
EXPOSE 80
CMD ["/var/www/html/index.php", "-D", "BACKGROUND"]
```

Container Images

- An image is effectively a stopped container that is loaded on the container host. An image is like a VM snapshot.
- Images are stored to and retrieved from an image repository server.
- They contain application code and application dependencies:

- Do not contain kernel or device drivers.
- Are built using a layered file system, for example:
 - Layer 1: Base OS layer, including typically low-level OS libraries.

Each subsequent layer stores additional components as defined in the Dockerfile.

- Layer 2: Application-specific dependencies, such as NGINX web server and PHP runtime.
- Layer 3: Application code, such as index.php.

Layer 3 - Application code
/var/www/html/index.php

Layer 2 - Application dependencies
/usr/bin/php
/usr/sbin/nginx
/usr/lib/php/20170718/pdo.so
/usr/lib/php/20170718/json.so
/etc/php/7.2/fpm/php-fpm.conf
/etc/php/7.2/fpm/php.ini
/etc/nginx/mime.types
/etc/nginx/nginx.conf
/var/log/nginx/access.log
/var/log/nginx/error.log

Layer 1 - Base OS

/bin/bash
/bin/cat
/bin/chmod
/bin/chown
/bin/cp
/bin/echo
/bin/grep
/bin/tar
/etc/hosts
/etc/fstab
/etc/passwd
/etc/group
/etc/hostname
/etc/networks
/etc/nsswitch.conf
/etc/resolv.conf
/etc/security/limits.conf
/etc/sysctl.conf
/lib/x86_64-linux-gnu/ld-2.27.so
/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2
/lib/x86_64-linux-gnu/libc-2.27.so
/lib/x86_64-linux-gnu/libc.so.6

Images and Containers

- An image is the result of a build.
- A container is a running instance from an image.

```
$ docker images
php              latest        8c811b4aec35        2 weeks ago      1.15 MB
mysql            latest        a8a59477268d        4 weeks ago      445 MB
foo.com/mysql    latest        a8a59477268d        4 weeks ago      445 MB

$ docker ps
CONTAINER ID   IMAGE     COMMAND       CREATED          STATUS          PORTS          NAMES
Acc5c8f58392   mysql     "mysqld"     About a minute ago   Up About a minute  3306/tcp       myWebsiteDB
```

Starting and Stopping Containers

- You run the following commands to start and stop containers:
- `docker run`: Starts a new container from an image
- `docker stop`: Stops a running container
- `docker rm`:
 - Deletes a container (must be stopped)
 - Add `-f` to both stop and remove a container

Managing Images

- The following commands help to manage images:
- docker images: Displays a list of images on the machine
- docker rmi: Deletes an image
- docker build: Builds an image from a Dockerfile
- docker tag: Adds tags to an image
- docker pull and docker push: Pulls and pushes images to and from a registry

Additional Docker Commands

- You might run these additional commands when working with images:
- docker ps:
 - Retrieves a list of running containers
 - Add `-a` to include non-running containers
- docker logs: Displays a container's log output
- docker exec:
 - Runs a command within a container
 - Can also start a shell within a container (if available)
- docker network: Creates a network

Container Registry

- A container registry provides a central location for container image storage.
- Types of registries include hosted and self-hosted:
 - Hosted: Docker Hub, Google Container Registry
 - Self-hosted: Artifactory, Harbor, Quay
- Container registries can be public or private.
- Images are built on a build host and pushed to a registry.

Project Harbor

- Harbor is an open-source enterprise-class registry server. It is integrated into many VMware products.

- Harbor includes the following features:

- Identity integration and role-based access control (RBAC)
- Security vulnerability scanning (Clair or Trivy)
- Content trust and image signing (Notary)
- Policy-based image replication
- Helm chart management



Additional References

Title	Location
Container 101 for the vSphere Admin	https://www.youtube.com/watch?v=NeJ20lbzv0c
Google: 'EVERYTHING at Google runs in a container'	https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/
What is a Container?	https://www.youtube.com/watch?v=EnJ7qX9fkcU
The History of Container Technology	https://linuxacademy.com/blog/containers/history-of-container-technology/
A Brief History of Containers: From the 1970s to 2017	https://blog.aquasec.com/a-brief-history-of-containers-from-1970s-chroot-to-docker-2016
Container vs. Process	https://sites.google.com/site/mytechnicalcollection/cloud-computing/docker/container-vs-process
Container vs. VM	https://sites.google.com/site/mytechnicalcollection/cloud-computing/docker/container-vs-vm

Chapter 6

Kubernetes Cluster and its objects

Objectives

Chapter 6: Kubernetes Cluster and its objects

- Learn about the architecture and benefits of Kubernetes Orchestration
- Learn about the cluster formation of Kubernetes
- Deep dive on the Kubernetes objects

Cloud-Native Principles

Cloud-native principles apply to containers:

- Isolated unit of work that does not require OS dependencies
- Actively scheduled and managed by an orchestration process
- Loosely coupled from any dependencies
- Use microservices

Problems Solved by Kubernetes

With Docker, containers are managed on a single container host. Managing multiple containers across multiple container hosts creates many problems:

- Managing large numbers of containers
- Restarting failed containers
- Scaling containers to meet capacity
- Networking and load balancing

Kubernetes provides an orchestration layer to solve these issues.

About Kubernetes

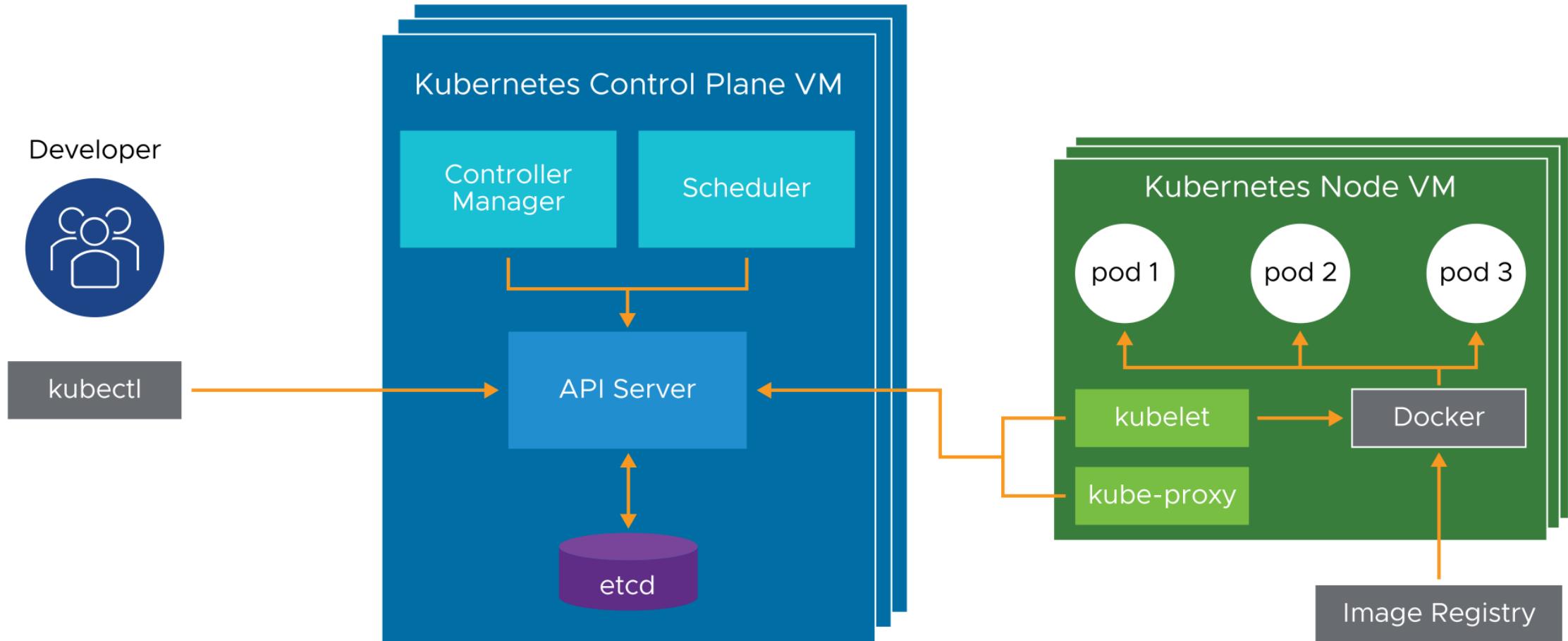
- Kubernetes is an open-source cluster management tool:
- Automates the deployment, management, and scaling of applications
- Managed by Cloud Native Computing Foundation (CNCF)
- Actively developed by a well-supported community, including former Borg engineers
- Lessons learned from Borg in production for more than a decade
- Can run on bare metal, hypervisors, or on various cloud providers

Benefits of Kubernetes

- Kubernetes provides a framework for managing containers at scale. It abstracts the underlying infrastructure and provides a consistent set of APIs and tools for deploying, scaling and managing containerized applications.
- Some key features of Kubernetes include:
 - Container Orchestration
 - Scalability
 - Service Discovery and Load Balancing
 - Self-Healing
 - Rolling Updates and Rollbacks
 - Storage Orchestration

Kubernetes Architecture

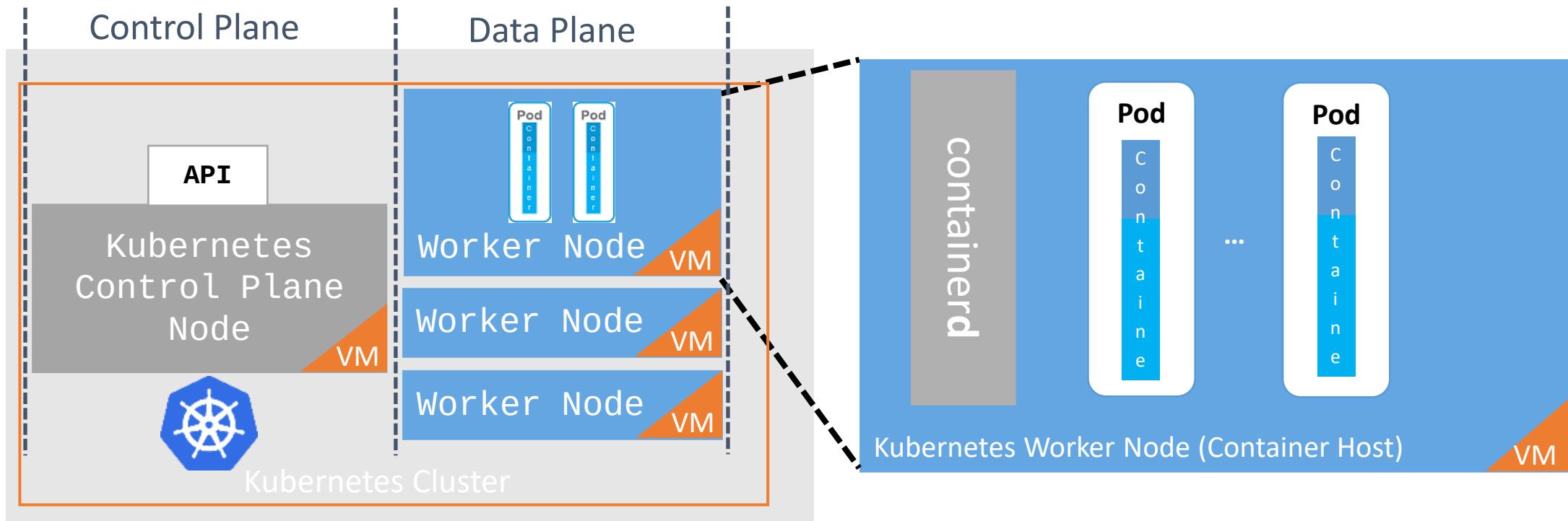
The Kubernetes architecture has several components.



Kubernetes Building Blocks

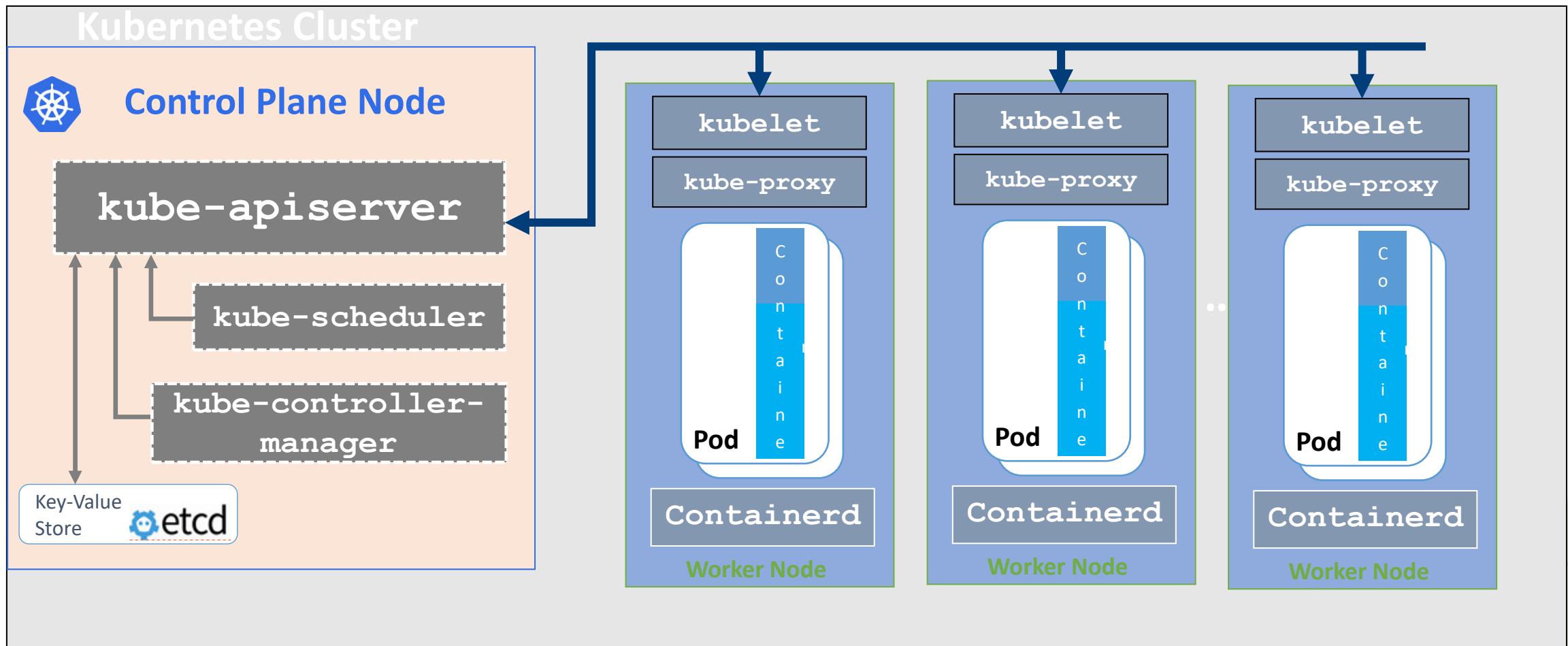
The building blocks of Kubernetes are pods, nodes, and clusters:

- Pod: Containers are encapsulated in pods.
- Node: A container host, for example, a virtual machine running Docker engine.
- Cluster: A set of worker nodes (data plane) that are managed by a control plane node.



Kubernetes Cluster Architecture

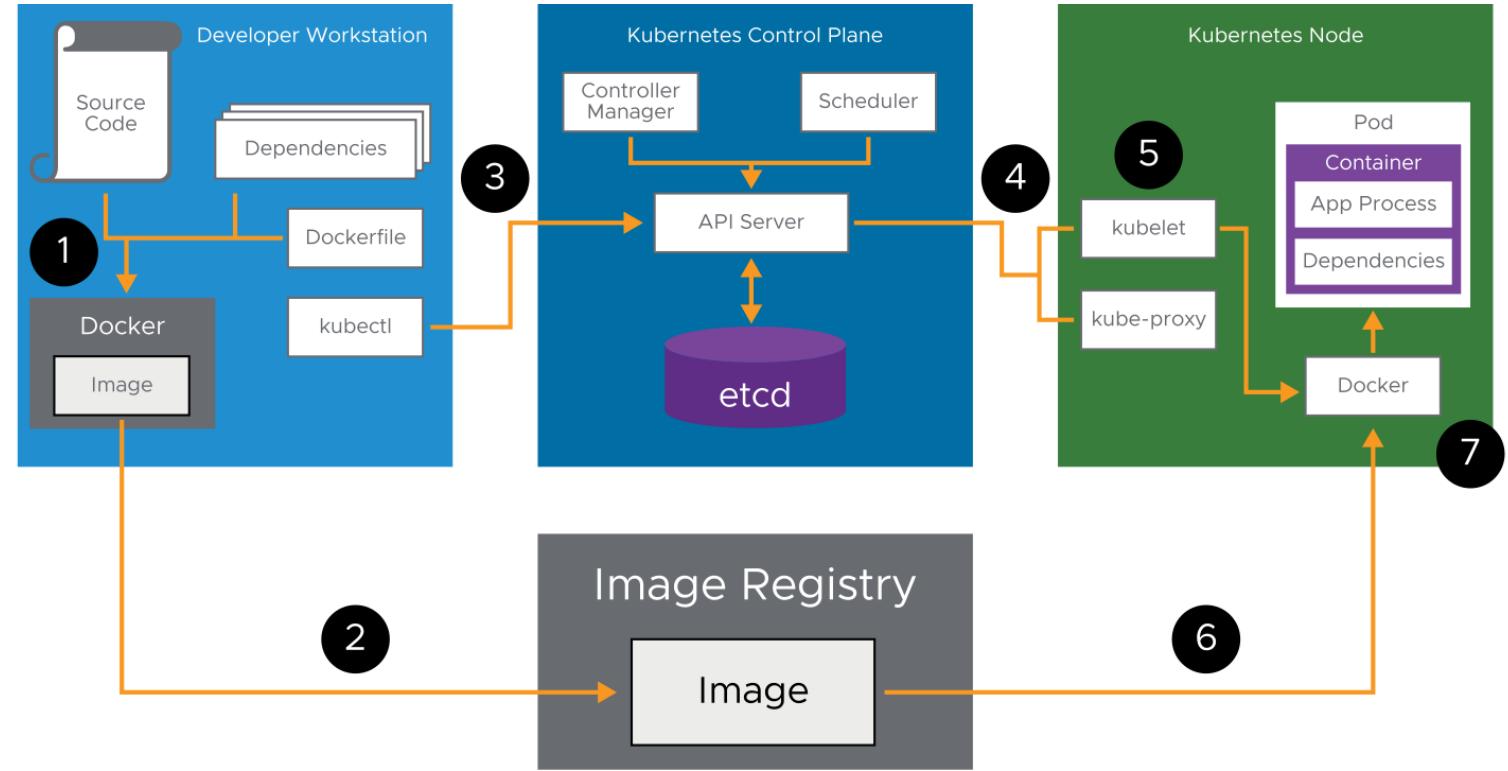
A Kubernetes cluster is a set of worker nodes managed by a control plane node. Typically, a Kubernetes node is a virtual machine.



Kubernetes Workflow Example

Example workflow steps:

1. Build an image from source code and dependencies.
2. Send the image to the image registry.
3. Tell Kubernetes to use the image to run a pod.
4. Scheduler assigns the pod to a node.
5. Kubelet accepts the pod.
6. Docker takes the image from the image registry.
7. Docker starts the container process inside a pod.



About kubeadm

- Bootstrap minimum viable cluster:
- Control plane creation and setup
- Joining nodes
- Certificate creation and management
- Initial cluster-admin account setup



About Manifests

In Kubernetes, manifest files declare the desired state of objects.

Manifests have the following properties:

- YAML format
- Declarative configuration
- Desired API primitives

Kubernetes manages the creation of the requested primitives.

About Pods (1)

A pod is a set of one or more tightly coupled containers.

It is the smallest unit of work in Kubernetes.

Containers in a pod start and stop together.

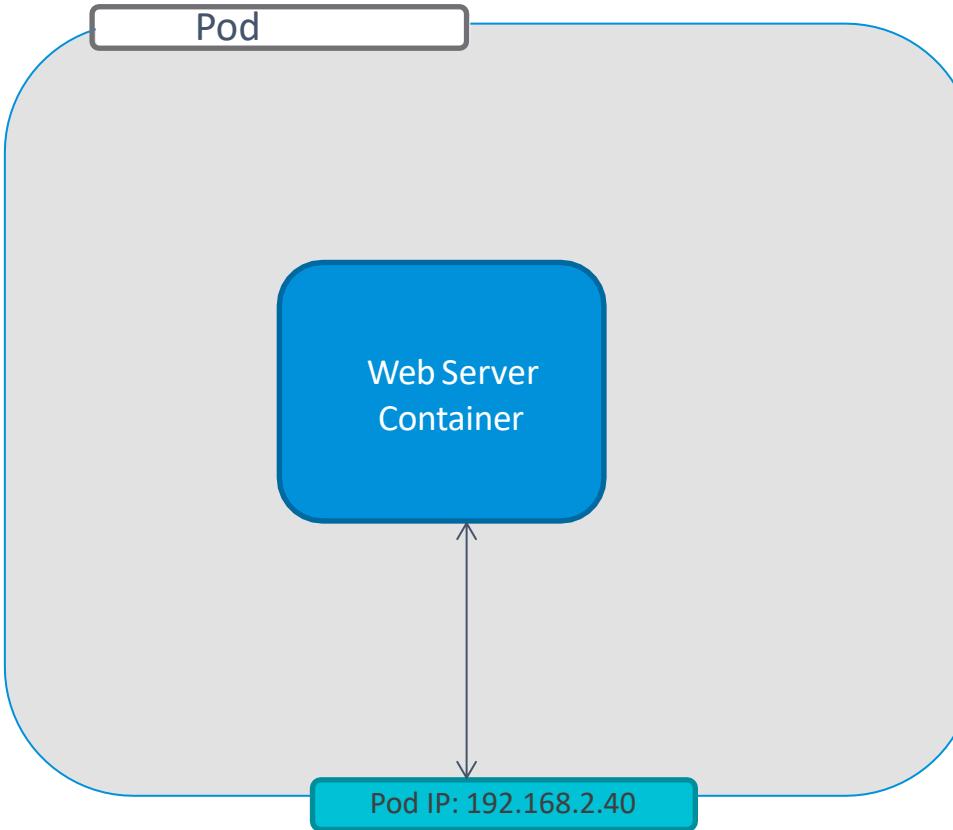
Example: Run a pod.

- The pod name is nginx-demo.
- The pod is assigned a label.
- It runs image nginx with version 1.7.9.
- It should listen on port 80.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx-demo
  labels:
    app: nginx
spec:
  containers:
  - name: nginx
    image: nginx: 1.7.9
    ports:
    - containerPort: 80
```

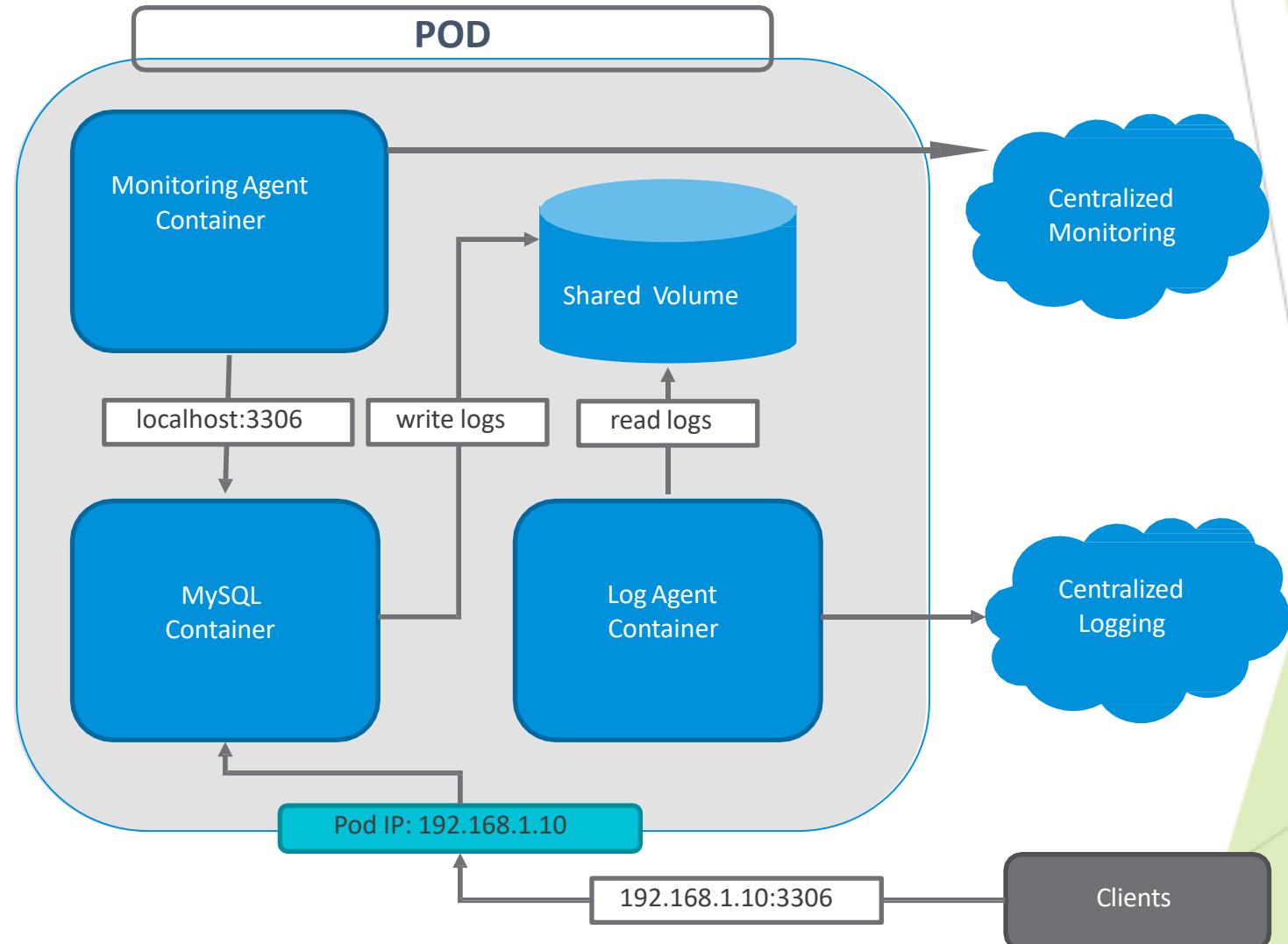
Pod-to-Container Relationship

- The Kubernetes pod is created to run the container.
- The container is the reason that a pod exists.



Multiple Containers in a Pod

- A pod typically exists for one container (1:1 relationship)
- One or more containers can be included in a pod.
- All containers in a pod run on the same node.
- Containers in a pod can talk to each other over the localhost.
- Containers can share volume resources.

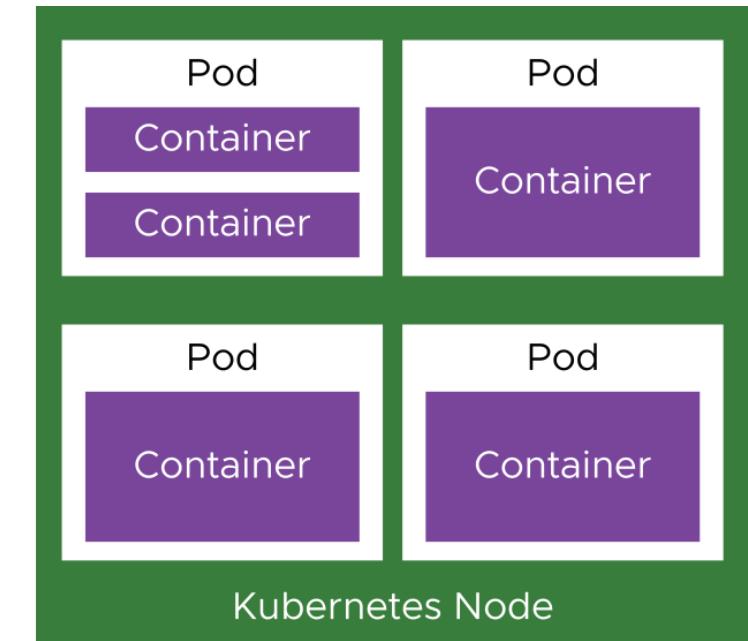
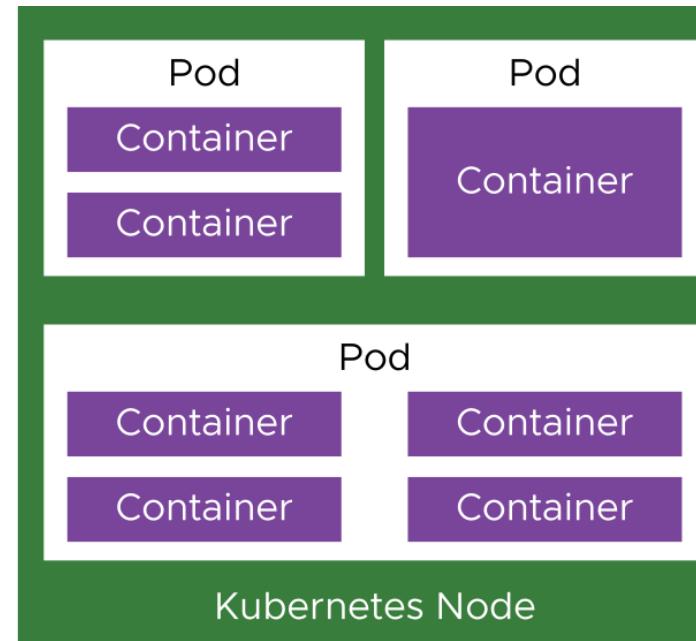
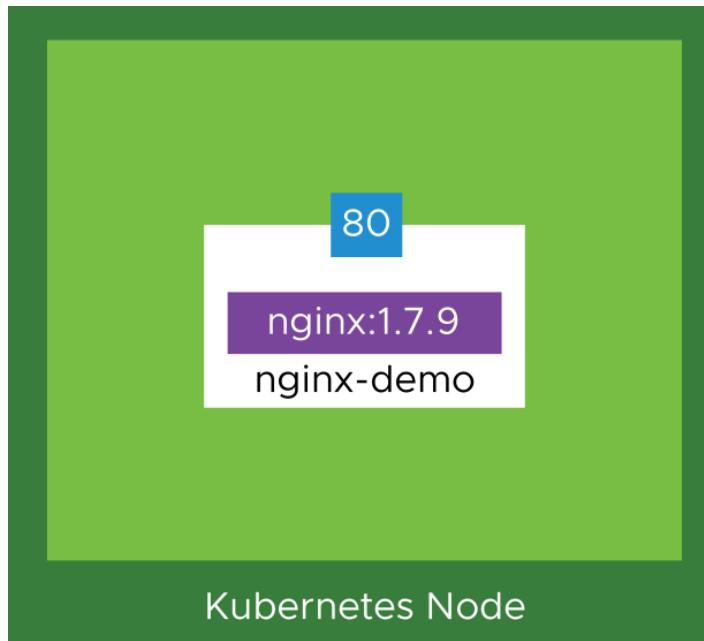


Kubernetes Pod Phases

- Pending:
 - The pod was accepted by the system, but one or more of the container images are not created. Includes the time before being scheduled and the time spent downloading images over the network.
- Running:
 - The pod is bound to a node, and all containers are created.
 - At least one container is still running or is in the process of starting or restarting.
- Succeeded:
 - All containers in the pod have terminated in success and are not restarted.
- Failed:
 - All containers in the pod have terminated. At least one container has terminated in failure (exited with nonzero exit status or was terminated by the system).
- Unknown:
 - The state of the pod cannot be obtained, typically because of an error in communicating with the host of the pod.

About Pods (2)

Pods run inside Kubernetes worker nodes and can run one or more container processes.



About ReplicaSets (1)

A ReplicaSet declares how the functionality of a pod is made scalable and resilient through redundancy.

The ReplicaSet ensures that a specified number of pods is kept running.

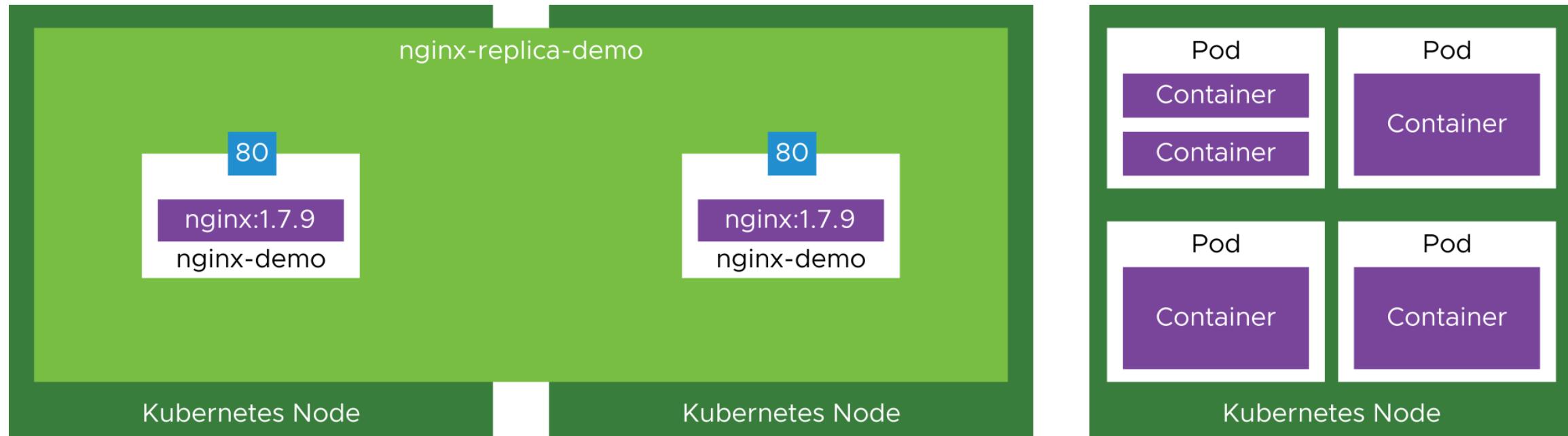
Example: Deploy a ReplicaSet.

- The ReplicaSet name is nginx-replica-demo.
- Two replicas are expected to be running.
- The ReplicaSet applies to pods with label nginx.

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: nginx-replica-demo
spec:
  replicas: 2
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
        - name: nginx
          image: nginx: 1.7.9
          ports:
            - containerPort: 80
```

About ReplicaSets (2)

ReplicaSets create and maintain copies of a pod. Replica pods can run on the same Kubernetes node or across nodes.



About Deployments (1)

A deployment is the most commonly used primitive:

- Declares whether pods can be upgraded and whether they can be patched without disrupting services
- Provides rolling updates by creating ReplicaSets and destroying old ReplicaSets
- Enables a new version of an image to be deployed without downtime

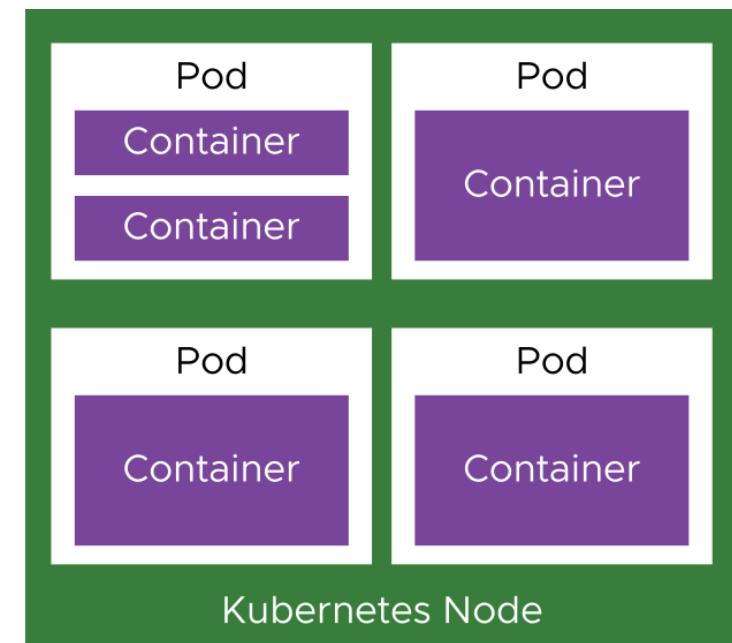
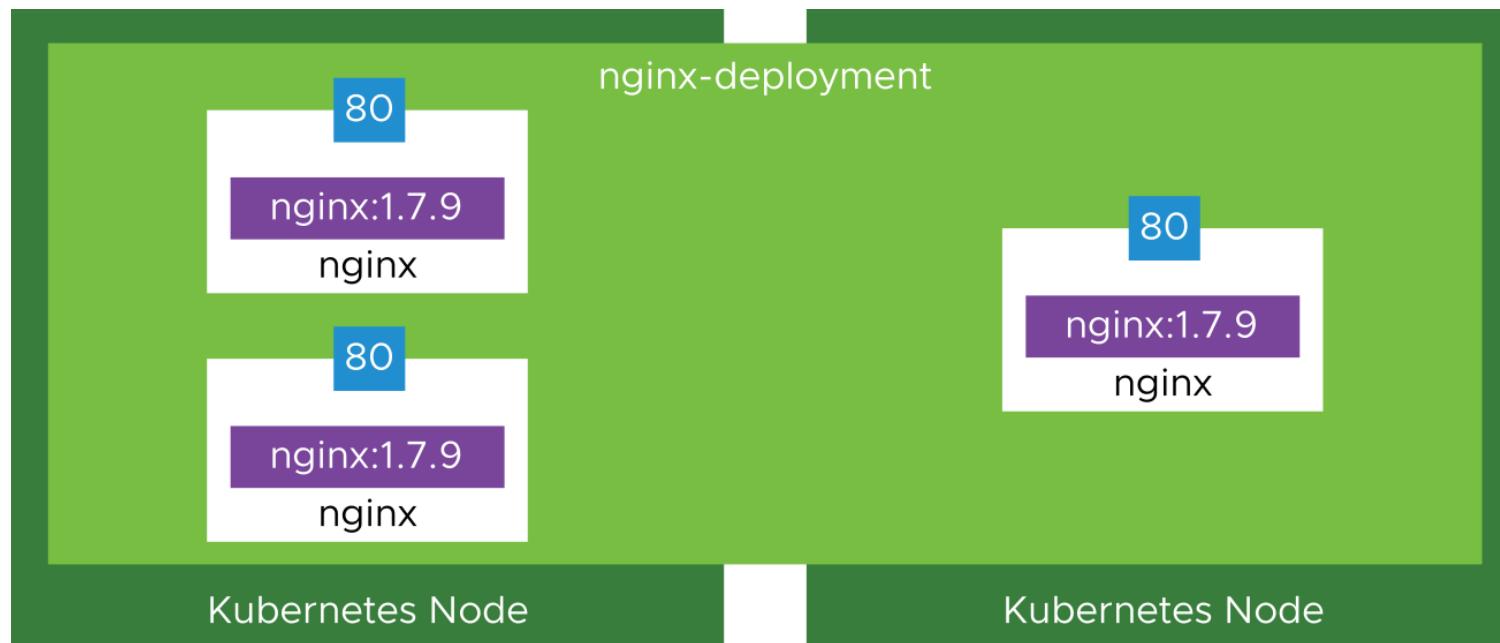
Example: Deploy a Deployment.

- Deployment name is nginx.
- Three replicas are expected to be running.
- The deployment runs image nginx with version 1.7.9.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 3
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx: 1.7.9
      ports:
      - containerPort: 80
```

About Deployments (2)

Deployments are commonly used to combine pod and ReplicaSet declarations in a single manifest.



About Services (1)

A service describes how pods discover and communicate with each other and external networks.

A service exposes pods as a single IP address.

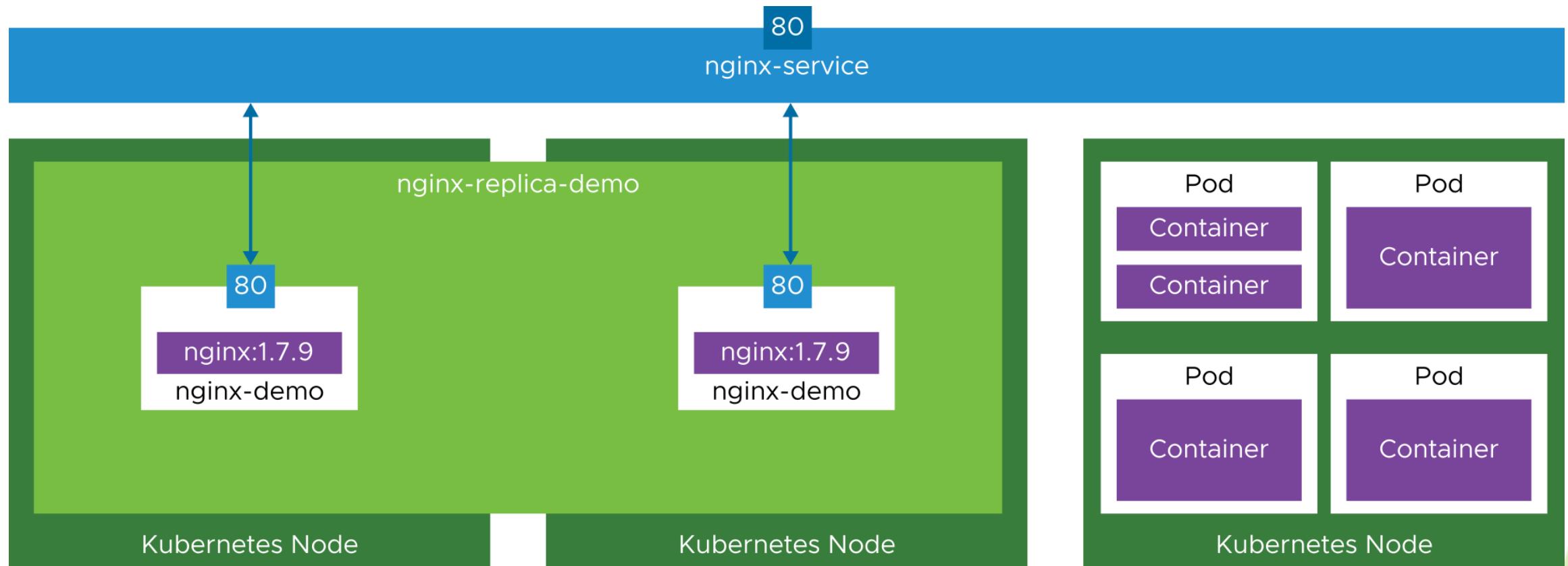
Example: Deploy a Service.

- Service name is nginx-service.
- It is a LoadBalancer service.
- It should listen on port 80.

```
apiVersion: v1
kind: Service
metadata:
  name: nginx-service
spec:
  type: LoadBalancer
  selector:
    app: nginx
  ports:
  - port: 80
    protocol: TCP
    targetPort: 80
```

About Services (2)

Services, like load balancer services, can expose pods with a static external IP address.



About Network Policies

A network policy is a specification of how groups of pods are allowed to communicate with each other and other network endpoints.

Network policies declare rules for ingress and egress traffic.

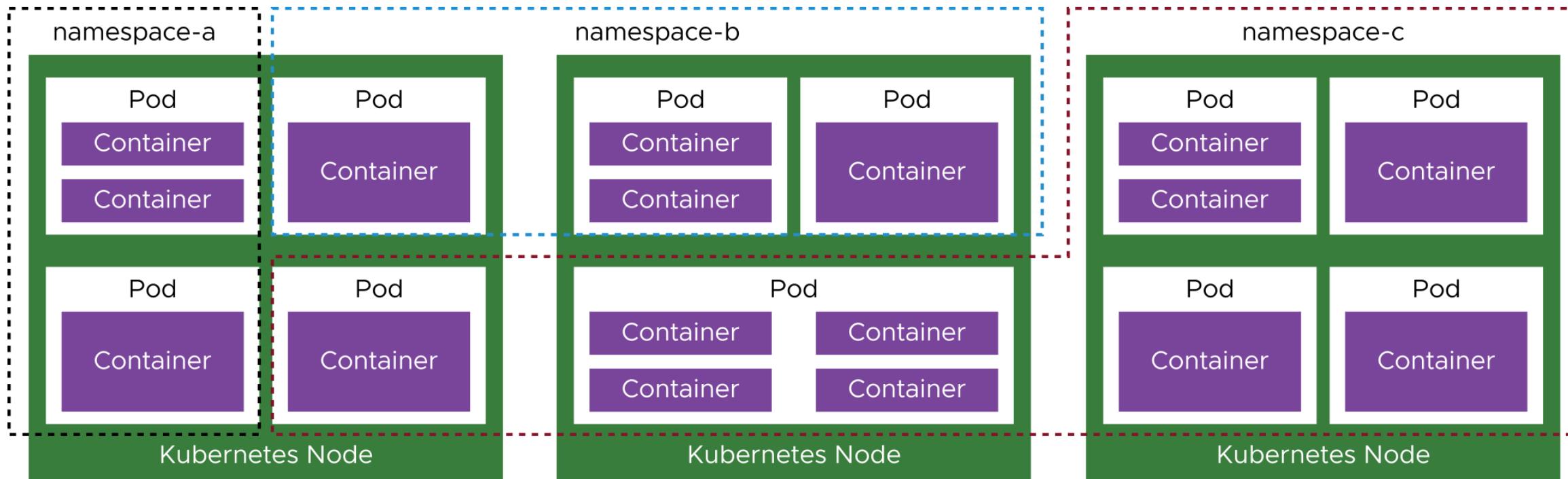
Example: Deploy a Network Policy.

- The policy name is network-policy.
- The policy allows all ingress traffic from 172.20.10.0/24.
- It allows all egress traffic to 10.0.0.0/8.
- It applies to pods with the app label nginx.

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: network-policy
  namespace: namespace-01
spec:
  podSelector:
    matchLabels:
      app: nginx
  policyTypes:
    - Ingress
    - Egress
  ingress:
    - from:
        - ipBlock:
            cidr: 172.20.10.0/24
    - podSelector:
        matchLabels:
          app: nginx
  ports:
    - protocol: TCP
      port: 80
  egress:
    - to:
        - ipBlock:
            cidr: 10.0.0.0/8
  ports:
    - protocol: TCP
      port: 80
```

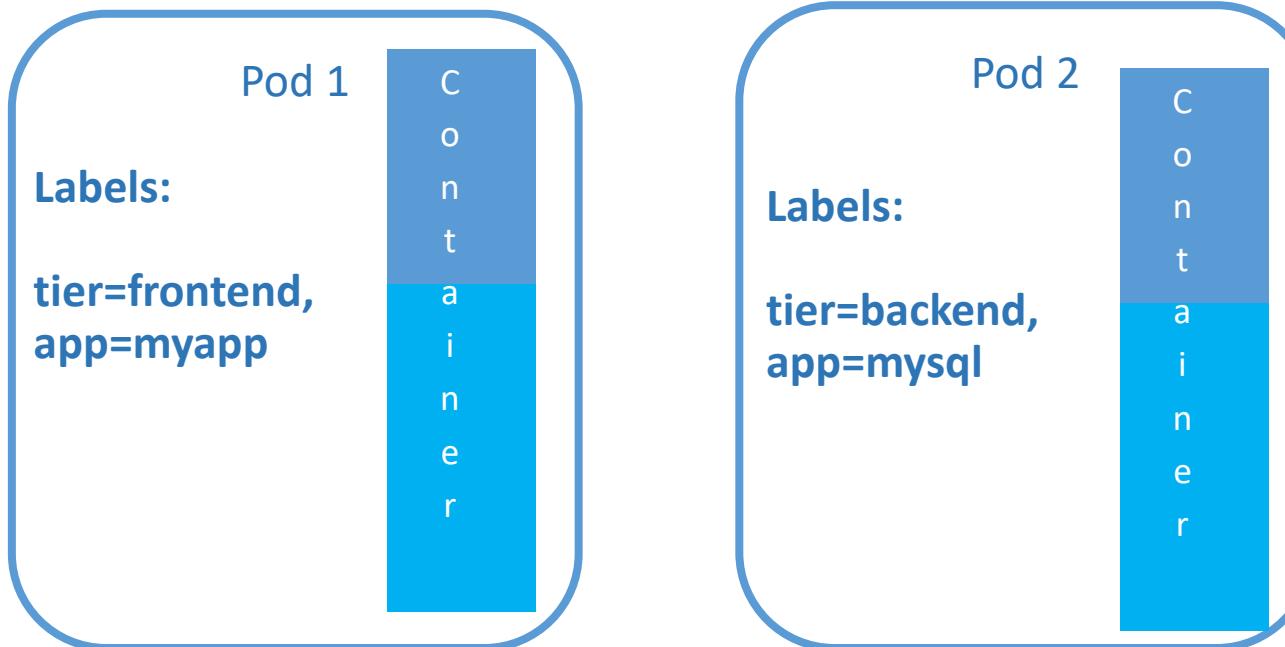
About Namespaces

Namespaces provide a resource and authorization boundary for Kubernetes objects.
Namespaces can span across Kubernetes nodes.



About Labels

- A label is a key-value pair attached to pods that conveys user-defined attributes. You aim to standardize key-value pairs across a cluster.
- You can use label selectors to select pods with specific labels and apply services or replication controllers to them (they can be functional or organizational).
- Labels can be attached to objects during creation and subsequently added and modified at any time.
- Labels are indexed, searchable, and best used consistently as a key-value pair across the cluster.



Chapter 7

ISTIO Fundamentals

Service Mesh



Objectives

Chapter 7: Istio Fundamentals

- Introducing service mesh with its architecture and features
- Learn about the installation procedure of Istio for microservices
- Learn how to reviewing the service requirements and adding services to the mesh

What is Service Mesh?

- Modern applications are typically architected as distributed collections of microservices, with each collection of microservices performing some discrete business function.
- A service mesh is a dedicated infrastructure layer that you can add to your applications.
- It allows you to transparently add capabilities like observability, traffic management, and security, without adding them to your own code.
- The term “service mesh” describes both the type of software you use to implement this pattern, and the security or network domain that is created when you use that software.

What is Service Mesh?

- As the deployment of distributed services, such as in a Kubernetes-based system, grows in size and complexity, it can become harder to understand and manage.
- Its requirements can include discovery, load balancing, failure recovery, metrics, and monitoring.
- A service mesh also often addresses more complex operational requirements, like A/B testing, canary deployments, rate limiting, access control, encryption, and end-to-end authentication.

What is Service Mesh?

- Service-to-service communication is what makes a distributed application possible.
- Routing this communication, both within and across application clusters, becomes increasingly complex as the number of services grow.
- Istio helps reduce this complexity while easing the strain on development teams.

Istio Service Mesh

- Istio is an open source service mesh that layers transparently onto existing distributed applications.
- The Istio project is an open source project cofounded by IBM, Google, and Lyft in 2017.
- Istio makes it possible to connect, secure, and observe your microservices while being language agnostic.
- Istio has grown to include contributions from companies beyond its original cofounders, companies such as VMware, Cisco, and Huawei, among others.
- Istio's powerful features provide a uniform and more efficient way to secure, connect, and monitor services.
- Istio is the path to load balancing, service-to-service authentication, and monitoring – with few or no service code changes.

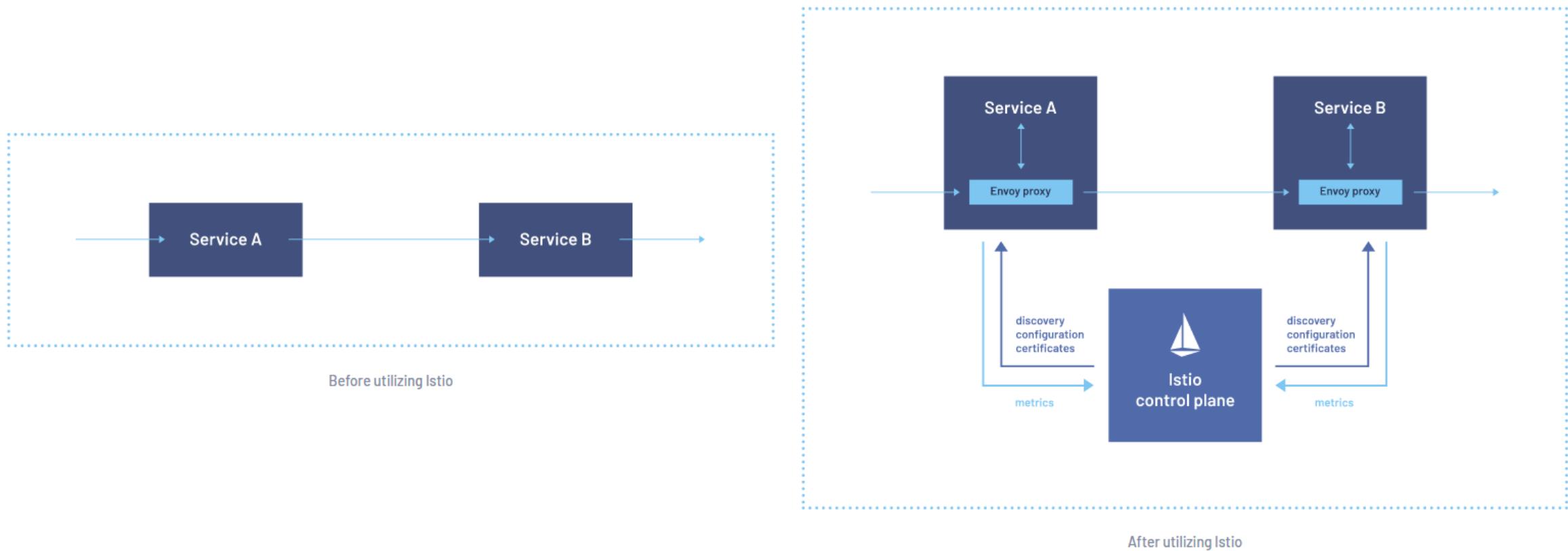
Istio Service Mesh

- Istio is designed for extensibility and can handle a diverse range of deployment needs.
- Istio's control plane runs on Kubernetes, and you can add applications deployed in that cluster to your mesh, extend the mesh to other clusters, or even connect VMs or other endpoints running outside of Kubernetes.

Istio Service Mesh – How it works?

- Istio has two components: the data plane and the control plane.
- The **data plane** is the communication between services. Without a service mesh, the network doesn't understand the traffic being sent over, and can't make any decisions based on what type of traffic it is, or who it is from or to.
- The **control plane** takes your desired configuration, and its view of the services, and dynamically programs the proxy servers, updating them as the rules or the environment changes.
- Service mesh uses a proxy to intercept all your network traffic, allowing a broad set of application-aware features based on configuration you set.
- An Envoy proxy is deployed along with each service that you start in your cluster, or runs alongside services running on VMs.

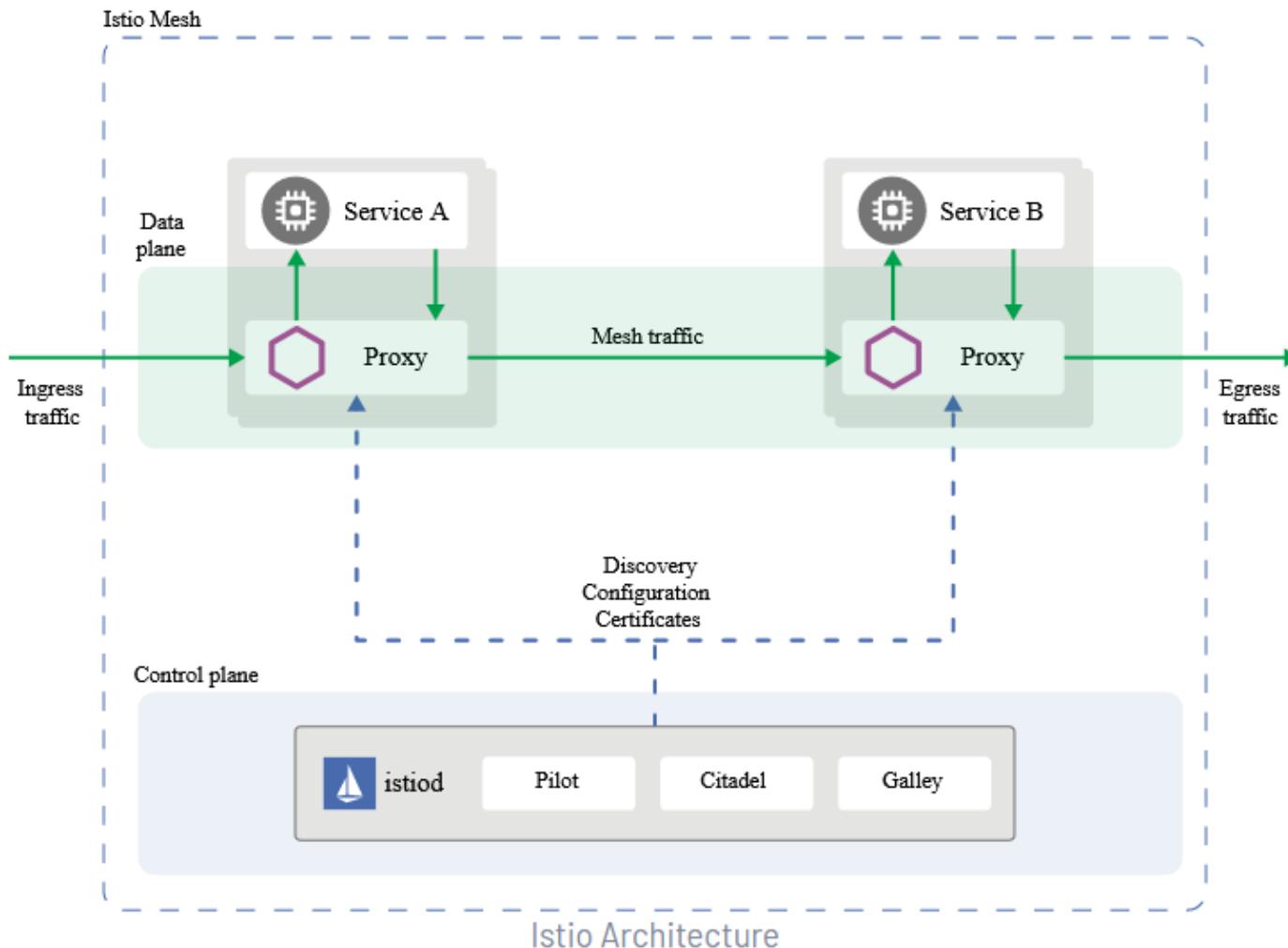
Istio Service Mesh – How it works?



Istio Service Mesh Architecture

- An Istio service mesh is logically split into a data plane and a control plane.
- The data plane is composed of a set of intelligent proxies (Envoy) deployed as sidecars. These proxies mediate and control all network communication between microservices. They also collect and report telemetry on all mesh traffic.
- The control plane manages and configures the proxies to route traffic

Istio Service Mesh Architecture



Istio Service Mesh Core Components

- **Envoy**
 - Istio uses an extended version of the Envoy proxy. Envoy is a high-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh. Envoy proxies are the only Istio components that interact with data plane traffic.
- **Istiod**
 - Istiod provides service discovery, configuration and certificate management.

Istio Service Mesh Core Components : Envoy

- Envoy proxies are deployed as sidecars to services, logically augmenting the services with Envoy's many built-in features, for example:
 - Dynamic service discovery
 - Load balancing
 - TLS termination
 - HTTP/2 and gRPC proxies
 - Circuit breakers
 - Health checks
 - Staged rollouts with %-based traffic split
 - Fault injection
 - Rich metrics
- This sidecar deployment allows Istio to enforce policy decisions and extract rich telemetry which can be sent to monitoring systems to provide information about the behavior of the entire mesh.

Istio Service Mesh Core Components : Envoy

- Some of the Istio features and tasks enabled by Envoy proxies include:
 - Traffic control features: enforce fine-grained traffic control with rich routing rules for HTTP, gRPC, WebSocket, and TCP traffic.
 - Network resiliency features: setup retries, failovers, circuit breakers, and fault injection.
 - Security and authentication features: enforce security policies and enforce access control and rate limiting defined through the configuration API.
 - Pluggable extensions model based on WebAssembly that allows for custom policy enforcement and telemetry generation for mesh traffic.

Istio Service Mesh Core Components : Istiod

- Istiod provides service discovery, configuration and certificate management.
- Istiod converts high level routing rules that control traffic behavior into Envoy-specific configurations, and propagates them to the sidecars at runtime.
- Pilot abstracts platform-specific service discovery mechanisms and synthesizes them into a standard format that any sidecar conforming with the Envoy API can consume.
- Istio can support discovery for multiple environments such as Kubernetes or VMs.

Istio Service Mesh Core Components : Istiod

- Istiod security enables strong service-to-service and end-user authentication with built-in identity and credential management.
- You can use Istio to upgrade unencrypted traffic in the service mesh. Using Istio, operators can enforce policies based on service identity rather than on relatively unstable layer 3 or layer 4 network identifiers.
- Additionally, you can use Istio's authorization feature to control who can access your services.
- Istiod acts as a Certificate Authority (CA) and generates certificates to allow secure mTLS communication in the data plane.

Istio Service Mesh Features

- Istio solves the challenges of managing microservices by using a core set of features that allow you to observe, connect, and secure your services.
- These features can be broken down into three main categories:
 - Observability
 - Traffic Management
 - Security.

Istio Service Mesh Features

- Observability:
- As services grow in complexity, it becomes challenging to understand behavior and performance. Istio generates detailed telemetry for all communications within a service mesh.
- This telemetry provides observability of service behavior, empowering operators to troubleshoot, maintain, and optimize their applications.
- Even better, you get almost all of this instrumentation without requiring application changes.
- Through Istio, operators gain a thorough understanding of how monitored services are interacting.

Istio Service Mesh Features

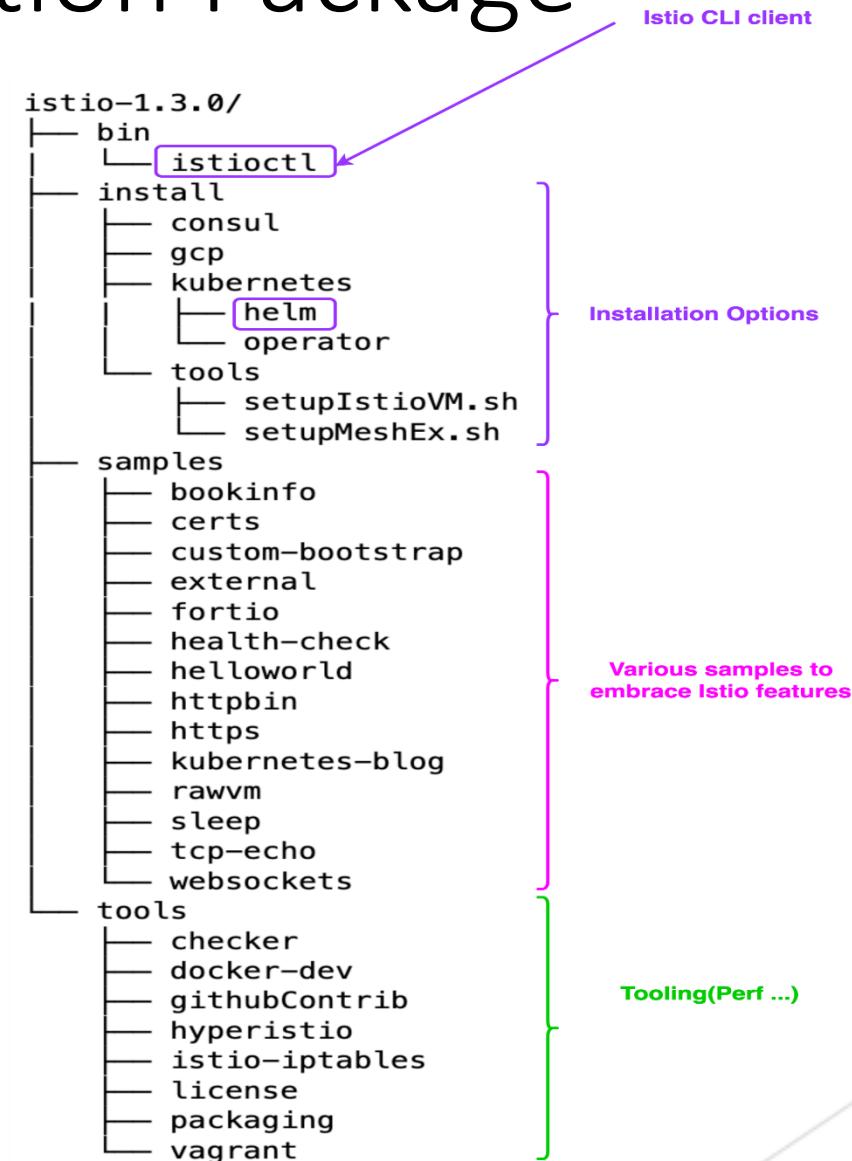
- Traffic Management
 - Routing traffic, both within a single cluster and across clusters, affects performance and enables better deployment strategy.
 - Istio's traffic routing rules let you easily control the flow of traffic and API calls between services.
 - Istio simplifies configuration of service-level properties like circuit breakers, timeouts, and retries, and makes it easy to set up important tasks like A/B testing, canary deployments, and staged rollouts with percentage-based traffic splits.

Istio Service Mesh Features

- Security
 - One of the most difficult features to enable in a distributed cloud application is secure communication between services where you have data encryption and authentication between the services.
 - As with all of the features in Istio, managing security between services is also declarative using the APIs available in the Istio control plane.
 - Enabling secure communication within the mesh is not an all-or-nothing setting.
 - Istio has settings that allow permissive secure channels between services. Selective permissive channels make it convenient for you to incrementally add services to the mesh without causing failures.

Istio Service Mesh Installation Package

- We can categorize Istio release package into 4 sections:
 - **bin/istioctl**: is the CLI for the Istio control plane similar to kubectl.
 - **install**: contains various installation options for the different platform. For Kubernetes, official support for Helm Chart and Operator Framework.
 - **sample**: contains various samples to get started with Istio and embrace its features.
 - **tools**: contains tooling for perf testing, etc ...



Istio Service Mesh Installation:

- There are several different ways to install Istio including a Helm chart, Kubernetes YAML files, and, soon to be available, an Istio Operator.
- The several steps over the installation are
 - Prepare the Kubernetes Environment
 - Download Istio
 - `$ curl -L https://git.io/getLatestIstio | ISTIO_VERSION=1.3.0 sh -`
 - Install Helm
 - Install Istio
 - `$ for i in install/kubernetes/helm/istio-init/files/crd*yaml; \`
 - `do kubectl apply -f $i; done`
 - `$ kubectl apply -f install/kubernetes/istio-demo.yaml`
 - Verify Istio
 - `$ istioctl verify-install -f install/kubernetes/istio-demo.yaml`

Reviewing Service Requirements

- Before you add Kubernetes services to the mesh, you need to be aware of the pod and services requirements to ensure that your Kubernetes services meet the minimum requirements.
 - Service descriptors:
 - Each service port name must start with the protocol name, for example, name: http.
 - Deployment descriptors:
 - The pods must be associated with a Kubernetes service.
 - The pods must not run as a user with UID 1337.
 - It is recommended that app and version labels are added to provide contextual information for metrics and telemetry.

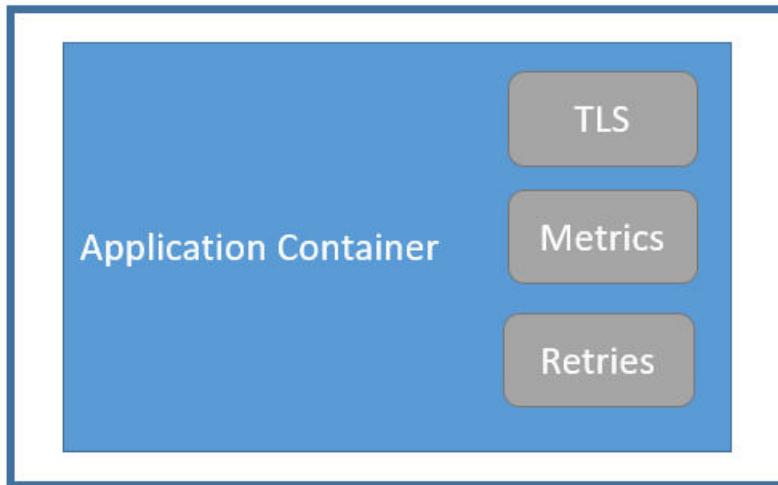
Sidecar Injection

- Adding services to the mesh requires that the client-side proxies be associated with the service components and registered with the control plane.
- With Istio, you have two methods to inject the Envoy proxy sidecar into the microservice Kubernetes pods:
 - Automatic Sidecar Injection
 - Manual Sidecar Injection

Sidecar Injection

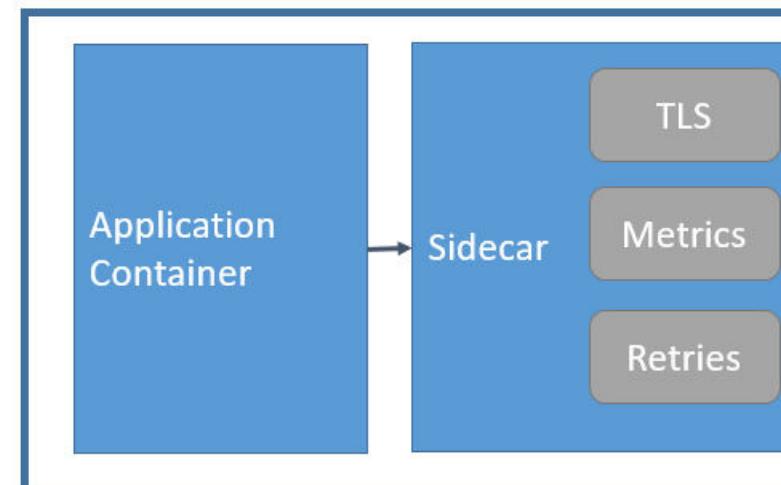
- The term sidecar is commonly used to refer to logging or metric pods that are used to forward information into another system such as logging or monitoring.

Before



Kubernetes Pod

After



Kubernetes Pod

Automatic Sidecar Injection

- Automatic sidecar injection is available within a Kubernetes environment.
- The automatic sidecar injection uses a Kubernetes mutating admission controller that is invoked during pod scheduling to determine whether it should add the Istio proxy to the deployment specification.
- The istio-injected label on the namespace of the pod is used by the Istio mutating admission controller to determine whether the Istio proxy should be injected into the pod deployment specification.

Manual Sidecar Injection

- Manual sidecar injection, on the other hand, allows you to examine the injected deployment YAML prior to deployment.
- Using the manual sidecar-injection approach gives you fine-grained control over which services are added to the mesh.
- A benefit of separating services by namespace, it is straightforward to use the automatic sidecar-injection approach configured for all services within a namespace.

Chapter 8

Securing Communication within Istio and controlling traffic

Objectives

Chapter 8: Securing Communication within Istio and controlling traffic

- Learn about the Istio Security
- Review how to enable the mTLS communication between services and secure inbound traffic
- Learn about Canary Testing, resiliency and chaos testing

Istio Security

- A key requirement for many cloud native applications is the ability to provide secure communication paths between services.
- It is always imperative to secure communication to your application by ensuring that only trusted identities can call your services.
- In traditional applications, we often see that communication to services is secured at the edge of the application, or, to be more explicit, a network gateway (appliance or software) is configured on the network in which the application is deployed.

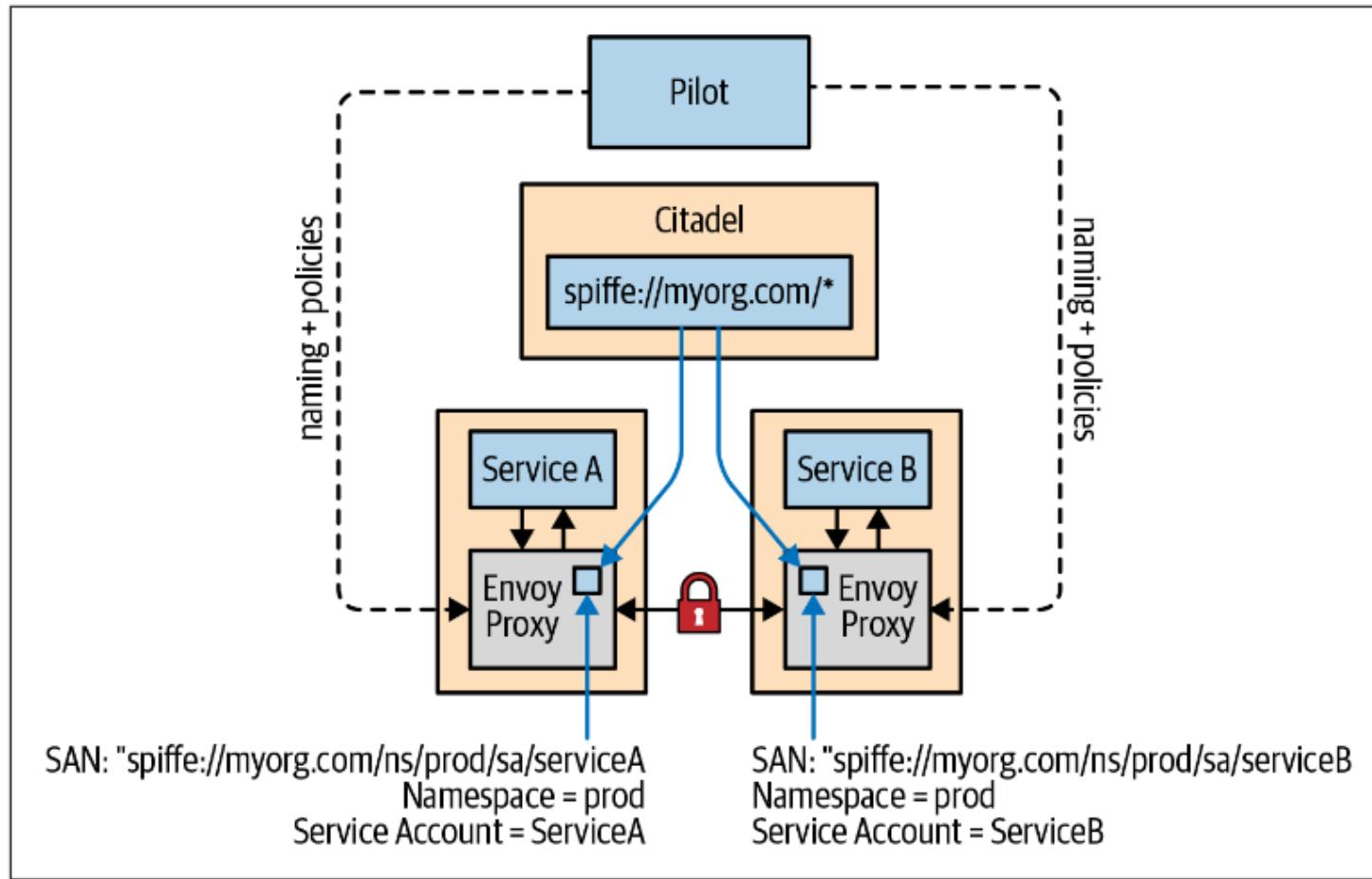
Istio Security

- Istio aims to provide security in depth to ensure that an application can be secured even on an untrusted network.
- Security at depth .places security controls at every endpoint of the mesh and not simply at the edge.
- Placing security controls at each endpoint ensures defense against man-in-the-middle attacks by enabling mTLS, encrypted traffic flow with secure service identities.

Istio Security

- The key Istio components involved in providing mTLS communication between services in the mesh, including the following:
 - Citadel
 - Manages keys and certificates including generation and rotation.
 - Istio (Envoy) Proxy
 - Implements secure communication between clients and servers.
 - Pilot
 - Distributes secure naming, mapping, and authentication policies to the proxies.

Istio Security Architecture



Istio Identities

- A critical aspect of being able to secure communication between services requires a consistent approach to defining the service identities.
- For mutual authentication between two services, the services must exchange credentials encoded with their identity.
- In Kubernetes, service accounts are used to provide service identities.
- Istio uses secure naming information on the client side of a service invocation to determine whether the client is allowed to call the server-side service.
- On the server side, the server is able to determine how the client can access and what information can be accessed on the service using authorization policies.

Istio Identities - Citadel

- To provide secure communication between services it is necessary to have a public key infrastructure (PKI). Istio's PKI is built on top of a component named Citadel.
- A PKI is responsible for securing communication between a client and a server by using public and private cryptographic keys.
- The PKI creates, distributes, and revokes digital certificates as well as manages public key encryption.
- Thus, Citadel is responsible for managing keys and certificates across the mesh.

Secure mTLS Communication

- One of the biggest reasons users adopt service mesh is to enable secure communication among applications using mutual TLS (mTLS) based on cryptographically verifiable identities.
- In this chapter, we'll discuss the requirements of secure communication among applications, how mTLS enables and meets all those requirements, along with simple steps to get you started with enabling mTLS among your applications using Istio.

Secure mTLS Communication

- **What do you need to secure the communications among your applications?**
- Modern cloud native applications are frequently distributed across multiple Kubernetes clusters or virtual machines.
- New versions are being staged frequently and they can rapidly scale up and down based on user requests.
- As modern applications gain resource utilization efficiency by not being dependent on co-location, it is paramount to be able to apply access policy to and secure the communications among these distributed applications due to increased multiple entry points resulting in a larger attack surface.
- To ignore this is to invite massive business risk from data loss, data theft, forged data, or simple mishandling.

Secure mTLS Communication

- The following are the common key requirements for secure communications between applications:
 - **Identities**
 - **Confidentiality**
 - **Integrity**
 - **Access Policy Enforcement**
 - **FIPS Compliance**

Secure mTLS Communication - Identities

- Identity is a fundamental component of any security architecture. Before your applications can send their data securely, **identities** must be established for the applications.
- This *establishing an identity* process is called **identity validation** - it involves some well-known, trusted **authority** performing one or more checks on the application workload to establish that it is what it claims to be.
- Once the authority is satisfied, it grants the workload an identity.

Secure mTLS Communication - Identities

- Your system may have identities derived from network properties such as IP addresses with distributed identity caches that track the mapping between identities and these network properties.
- These identities don't have strong guarantees as cryptographically verifiable identities because IP addresses could be re-allocated to different workloads and identity caches may not always be updated to the latest.
- Using cryptographically verifiable identities for your applications is desired, because exchanging cryptographically verifiable identities for applications during connection establishment is inherently more reliable and secure than systems dependent on mapping IP addresses to identities.

Secure mTLS Communication - Confidentiality

- Encrypting the data transmitted among applications is critical - because in a world where breaches are common, costly, and effectively trivial, relying entirely on *secure* internal environments or other security perimeters has long since ceased to be adequate.
- To prevent a man-in-the-middle attack, you require a unique encryption channel for a source-destination pair because you want a strong identity uniqueness guarantee to avoid confused deputy problems.
- In other words, it is not enough to simply encrypt the channel - it must be encrypted using unique keys directly derived from the unique source and destination identities so that only the source and destination can decrypt the data.
- Further, you may need to customize the encryption, e.g. by choosing specific ciphers, in accordance with what your security team requires.

Secure mTLS Communication - Integrity

- The encrypted data sent over the network from source to destination can't be modified by any identities other than the source and destination once it is sent.
- In other words, data received is the same as data sent.
- If you don't have data integrity, someone in the middle could modify some bits or the entire content of the data during the communication between the source and destination.

Secure mTLS Communication – Access Policy Enforcement

- Application owners need to apply access policies to their applications and have them enforced properly, consistently, and unambiguously. In order to apply policy for both ends of a communication channel, we need an application identity for each end.
- Once we have a cryptographically verifiable identity with an unambiguous provenance chain for both ends of a potential communication channel, we can begin to apply policies about who can communicate with what.
- Standard TLS, the widely used cryptographic protocol that secures communication between clients (e.g., web browsers) and servers (e.g., web servers), only really verifies and mandates an identity for one side - the server.
- But for comprehensive end-to-end policy enforcement, it is critical to have a reliable, verifiable, unambiguous identity for both sides - client and server.

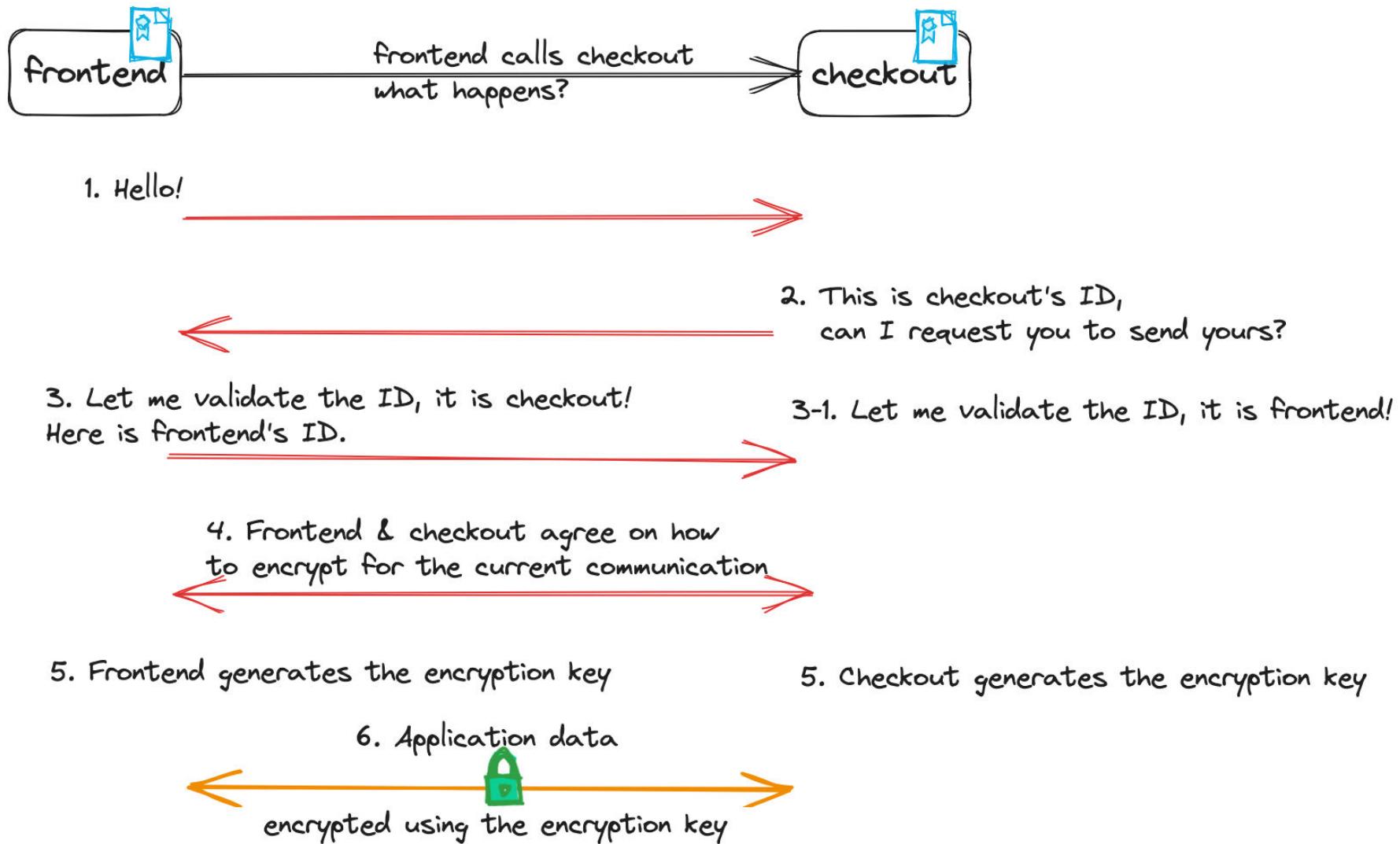
Secure mTLS Communication – Access Policy Enforcement

- This is a common requirement for internal applications - imagine for example a scenario where only a frontend application should call the GET method for a backend checkout application, but should not be allowed to call the POST or DELETE method.
- Or a scenario where only applications that have a JWT token issued by a particular JWT issuer can call the GET method for a checkout application.
- By leveraging cryptographic identities on both ends, we can ensure powerful access policies are enforced correctly, securely, and reliably, with a validatable audit trail.

Secure mTLS Communication – FIPS Compliance

- Federal Information Processing Standards (FIPS) are standards and guidelines for federal computer systems that are developed by National Institute of Standards and Technology (NIST).
- Not everyone requires FIPS compliance, but FIPS compliance means meeting all the necessary security requirements established by the U.S. government for protecting sensitive information. It is required when working with the federal government.
- To follow the guidelines developed by the U.S. government relating to cybersecurity, many in the private sector voluntarily use these FIPS standards.

Secure mTLS Communication – FIPS Compliance



Secure mTLS Communication

- **How does mTLS satisfy the above requirements?**
- TLS 1.3 (the most recent TLS version at the time of writing) specification's primary goal is to provide a secure channel between two communicating peers. The TLS secure channel has the following properties:
 - Authentication: the server side of the channel is always authenticated, the client side is optionally authenticated. When the client is also authenticated, the secure channel becomes a mutual TLS channel.
 - Confidentiality: Data is encrypted and only visible to the client and server. Data must be encrypted using keys that are unambiguously cryptographically bound to the source and destination identity documents in order to reliably protect the application-layer traffic.
 - Integrity: data sent over the channel can't be modified without detection. This is guaranteed by the fact that only source and destination have the key to encrypt and decrypt the data for a given session.

mTLS in Istio – Enable mTLS

- Enabling mTLS in Istio for intra-mesh applications is very simple.
- All you need is to add your applications to the mesh, which can be done by labeling your namespace for either sidecar injection or ambient.
- In the case of sidecar, a rollout restart would be required for sidecar to be injected to your application pods.

mTLS in Istio – Enforce strict mTLS

- The default mTLS behavior is mTLS whenever possible but not strictly enforced.
- To strictly enforce your application to accept only mTLS traffic, you can use Istio's PeerAuthentication policy, mesh-wide or per namespace or workload.
- In addition, you can also apply Istio's AuthorizationPolicy to control access for your workloads.

mTLS in Istio – Enforce strict mTLS

```
$ kubectl apply -n istio-system -f - <<EOF
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
spec:
  mtls:
    mode: STRICT
EOF|
```

Secure Inbound Traffic

- The configuration of secure mTLS communication can secure traffic within the mesh, we want to turn your attention to securing communication into the mesh from a client outside the mesh. For example, clients using a web browser want to access a service from the public internet.
- Inbound and outbound traffic for the mesh is controlled with Istio gateways.
- These gateways are implemented as Envoy proxies, which allow or block traffic from entering or leaving the mesh.
- A mesh can have multiple gateway configurations; for example, you may want one set of gateways for public internet inbound and outbound traffic, while having a separate set of gateways for private network traffic.

Secure Inbound Traffic

- Istio Gateways are primarily used to provide secure inbound access to the mesh, but the Istio egress (outbound) gateway also provides critical control over outbound traffic.
- For example, you can configure an Istio egress gateway with policies to restrict which destinations can be reached by specific services within the mesh.
- This level of traffic control is quite difficult with Kubernetes itself. Let's start by ensuring there is secure inbound communication by configuring an Istio ingress (inbound) gateway for secure TLS communication to the trade service within the mesh.

Secure Inbound Traffic

- We can see that a default Istio ingress gateway has already been deployed into our Kubernetes cluster during installation.
- Using the following command, you can see that the ingress gateway is deployed as a LoadBalancer service with an external public IP address:

```
$ kubectl get svc -n istio-system -l app=istio-ingressgateway
```

```
$ kubectl get svc -n istio-system -l app=istio-ingressgateway
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP
istio-ingressgateway	LoadBalancer	172.21.39.61	169.63.159.157
PORT(S)			
15020:31382/TCP,80:31380/TCP,443:31390/TCP,31400:31400/TCP,15029:31133/TCP,15030:30832/TCP,15031:31732/TCP,15032:32263/TCP,15443:31348/TCP			
AGE			
15d			

Istio – Control Traffic

- In a Kubernetes environment, there is simple round-robin load balancing between service endpoints.
- While Kubernetes does support deployment strategies such as a rolling deployment, it is quite coarse grained and is limited to moving to a new version of the service.
- You may find it necessary to have more than one version of the service running and perform a dark launch or a canary test.
- A service mesh enables these types of traffic management patterns by controlling requests and resiliency between services and controlling the traffic entering and leaving the cluster.

Istio – Control Traffic

- This chapter explores many of these types of features to control the traffic between services including increasing the resiliency between the services.
 - Dark Launch
 - Canary Testing
 - Resiliency and Chaos Testing

Dark Launch

- Dark launch allows you to deploy a service or a new version of a service while minimizing the impact to users; in other words, you can keep the service in the dark.
- It is imperative that you can develop and deliver new versions of your application with agility and low risk.
- Using a dark-launch approach enables you to deliver new functions rapidly with reduced risk.
- Since Istio allows you to precisely control how new versions of services are rolled out and accessed by clients, you can use a dark-launch approach for delivering changes.

Canary Testing

- A canary test is when you deploy a new version (the canary) along with the previous version and route a percentage of requests to the new version to determine whether there are problems before routing all traffic to the new version.
- After satisfactorily testing a new feature with a selective set of requests, a canary test is often performed to ensure that the new version of the service not only functions properly, but also doesn't cause a degradation in performance or reliability.
- You may even place higher load on the canary deployment monitoring the effects over time.
- If there are no observed ill effects on the environment, you would adjust the routing rules to direct all of the traffic to the canary deployment.

Canary without Istio

- Container orchestration platforms such as Kubernetes use instance scaling to manage traffic routing between deployments as well as the number of replicas to control the weight between the deployment endpoints.
- With Istio, you can have multiple versions of the trader service deployed at the same time and allow them to scale up and down independently, without affecting the traffic distribution between them.
- As a result, you can scale up or down either version of the trader service without worrying about causing an impact to the traffic distribution among the versions of the service.
- Istio allows you to decouple deployments from traffic routing.

Resiliency and Chaos Testing

- When you build a distributed application designed for the cloud, it is critical to ensure that the services within your application are resilient to failures in the underlying platform as well as failures due to dependent services.
- Kubernetes provides capabilities that allow you to increase the resiliency of your container-based components against failures in the underlying infrastructure; however, it's up to you to ensure that your service implementations are resilient to failures from other services.
- To increase the resiliency of your services, it is common to set up retries, timeouts, and circuit breakers when calling dependent services.
- Luckily, we do not need to modify your existing code to have logic for retries and timeouts. Service meshes generally have constructs that allow you to program resiliency between your services.
- Istio has support for retries, timeouts, and circuit breakers, and even has capabilities that you can use to inject faults into your service calls to help test and tune your timeouts.

Summary

- Above and beyond everything else, it is important to have a service mesh strategy when using microservices in the cloud in order to get control over the complexities introduced by the highly distributed nature of microservices and the cloud. Beyond this key point, you should have a better understanding of the following points:
- A service mesh allows you to observe, secure, and connect microservices.
- Istio is a mature, multivendor open source service mesh implementation that has an architecture that provides you with the most complete and feature-rich service mesh implementation available.

Summary

- We have seen firsthand how Istio provides rich metrics for observability and secure mTLS communication between your services with few configurations required.
- We can add services to the mesh using Istio's automatic sidecar injection support per Kubernetes namespace, making it easier to incrementally adopt a service mesh, which is important for brownfield applications.

Chapter 9

Jenkins

Objectives

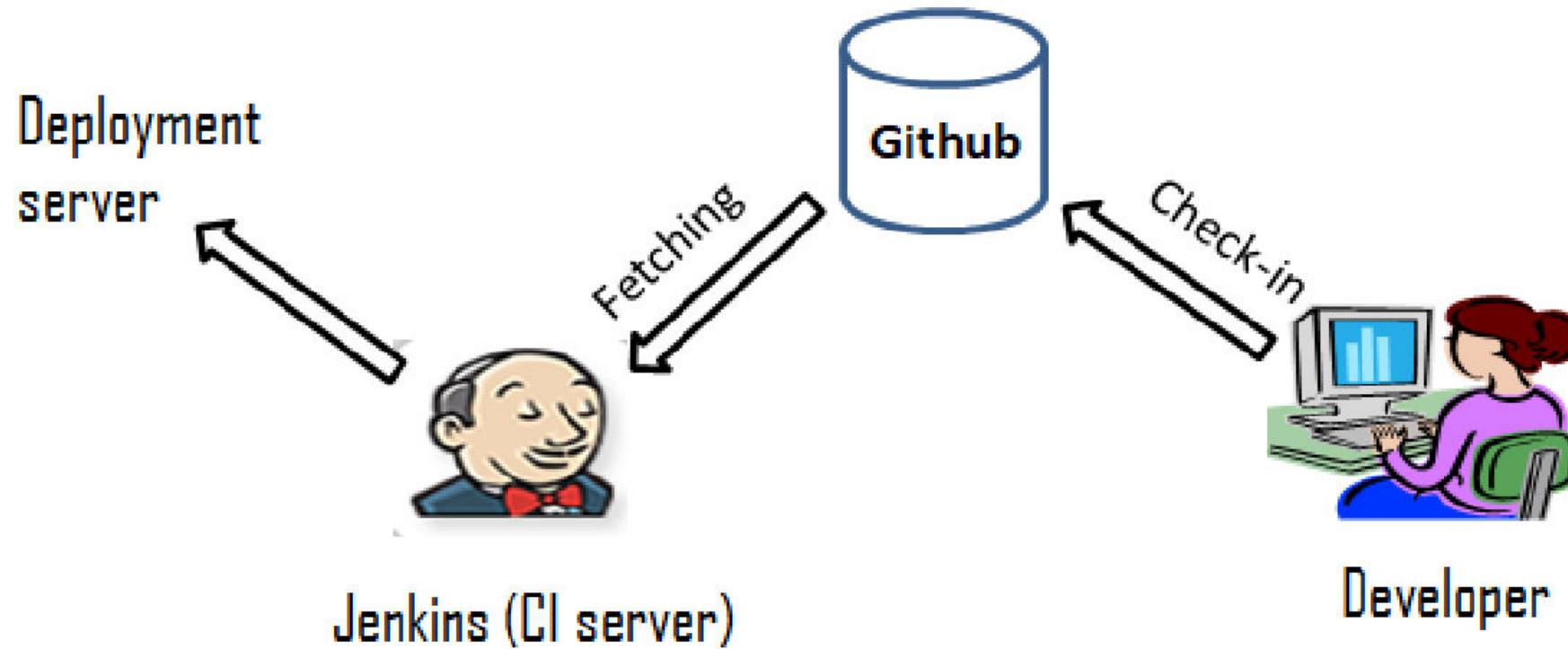
Chapter 9: Jenkins

- Introducing the Jenkins and Jenkins CI/CD Pipeline
- Learn how to install, manage and secure the Jenkins
- Learn about diagnosing the errors and troubleshooting

Jenkins-Overview

- Jenkins is an open source continuous integration (CI) server.
- It manages and controls several stages of the software delivery process, including build, documentation, automated testing, packaging, and static code analysis.
- Jenkins is a highly popular DevOps tool used by thousands of development teams.
- Jenkins is an open source automation tool written in Java programming language that allows continuous integration.
- Jenkins builds and tests our software projects, which continuously making it easier for developers to integrate changes to the project, and making it easier for users to obtain a fresh build.

Jenkins-Overview



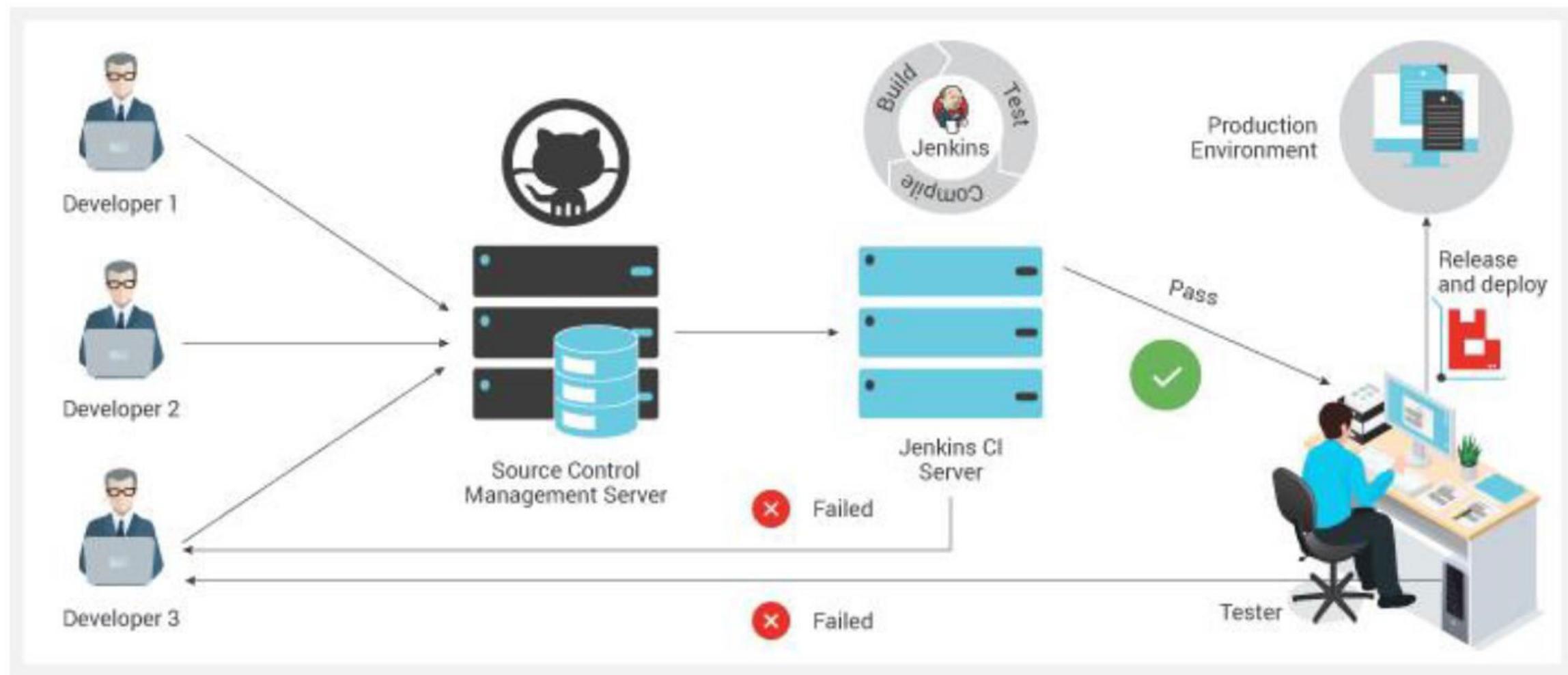
Jenkins-Overview

- Jenkins offers a straightforward way to set up a continuous integration or continuous delivery environment for almost any combination of languages and source code repositories using pipelines, as well as automating other routine development tasks.
- With the help of Jenkins, organizations can speed up the software development process through automation. Jenkins adds development life-cycle processes of all kinds, including build, document, test, package, stage, deploy static analysis and much more.
- Jenkins achieves CI (Continuous Integration) with the help of plugins. Plugins is used to allow the integration of various DevOps stages. If you want to integrate a particular tool, you have to install the plugins for that tool. For example: Maven 2 Project, Git, HTML Publisher, Amazon EC2, etc.

What is Continuous Integration?

- Continuous Integration (*CI*) is a development practice in which the developers are needs to commit changes to the source code in a shared repository at regular intervals.
- Every commit made in the repository is then built. This allows the development teams to detect the problems early.
- Continuous integration requires the developers to have regular builds. The general practice is that whenever a code commit occurs, a build should be triggered.

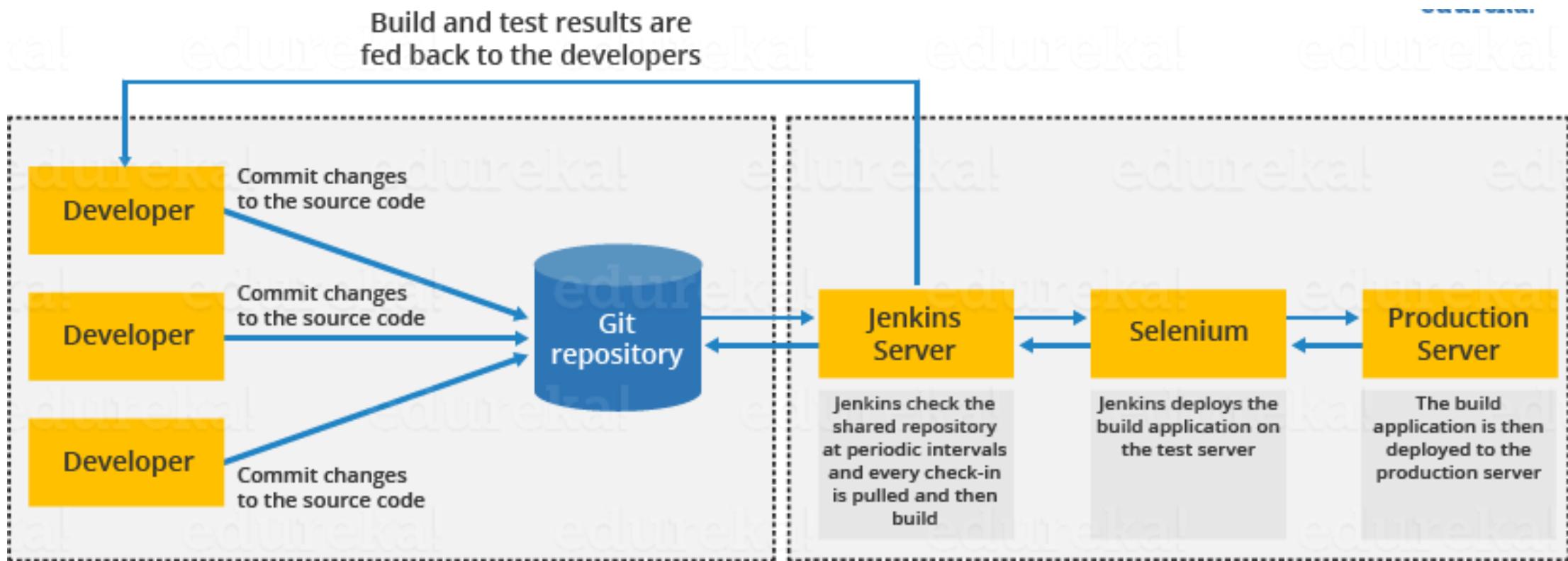
Jenkins CI Process



Continuous Integration with Jenkins

- Let's consider a scenario where the complete source code of the application was built and then deployed on test server for testing. It sounds like a perfect way to *develop software*, but this process has many problems.
 - Developer teams have to wait till the complete software is developed for the test results.
 - There is a high prospect that the test results might show multiple bugs. It was tough for developers to locate those bugs because they have to check the entire source code of the application.
 - It slows the software delivery process.
 - Continuous feedback pertaining to things like architectural or coding issues, build failures, test status and file release uploads was missing due to which the quality of software can go down.
 - The whole process was manual which increases the threat of frequent failure.

Continuous Integration with Jenkins



Jenkins- Terminology

- Agent

A machine which connects to a Jenkins master and executes tasks when directed by the master.
- Job

A deprecated term, synonymous with project.
- Master

The central, coordinating machine which stores configuration, loads plugins, and renders the various user interfaces for Jenkins.
- Node

A machine which is part of the Jenkins environment and capable of executing pipelines or projects. Both the master and agents are considered to be nodes.

Jenkins- Terminology

- Pipeline
 - A user-defined model of a continuous delivery pipeline.
- Plugin
 - An extension that provides additional functionality that is not provided by standard Jenkins.
- Project
 - A user-configured description of work which Jenkins should perform, such as building a piece of software.

Jenkins – Product Release Types

- Jenkins supports two types of releases based on the organization needs:
 - **Long term support release (LTS):** Long-term support releases (LTS) are available every 12 weeks. They are stable and are widely used and tested. Basically, this release is intended for end users.
 - **Weekly release:** This release is available in every week by fixing bugs in its earlier version. These releases are intended towards plugin developers.

Jenkins- Installing on Windows

Basic Requirements

JDK

We need either Java Development (JDK) or Java Runtime Environment (JRE)

Operating System

Jenkins can be installed on Windows, Mac OS X, Ubuntu/Debian, Red Hat/Fedora/CentOS, openSUSE, FreeBSD, OpenBSD, Gentoo.

Java Container

The WAR (Web Application Resource) file can be run in any container that supports Servlet 2.4/JSP 2.0 or later. (For example Tomcat 5).

Jenkins – Installing on Windows

- Install Java 8 or later
- Downloads Jenkins War file
 - The official website for Jenkins is <https://jenkins.io/>
- Starting Jenkins
 - Open the command prompt and go to the directory where the Jenkins.war file is located. And then run the following command:
C:/Java -jar Jenkins.war
- Accessing Jenkins
 - Now you can access the Jenkins. Open your browser and type the following url on your browser:
http://localhost:8080

Jenkins – Installing on Ubuntu Linux

- Installing Jenkins on Ubuntu

```
$ sudo apt update
```

- Install Java 8 or later

```
$ sudo apt install openjdk-8-jdk
```

- Add the Jenkins Debian Repository

- Import the GPG (GnuPG - GNU Privacy Guard) keys of the Jenkins repository using the following **wget** command:

```
$ wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
```

- The output of the above command should OK which means that the key has been successfully imported and packages from this repository will be considered trusted.
 - Now, add the Jenkins repository to the system with the following command:

```
sudo sh -c 'echo deb http://pkg.jenkins.io/debian  
stable binary/ > /etc/apt/sources.list.d/jenkins.list'
```

Jenkins – Installing on Ubuntu Linux

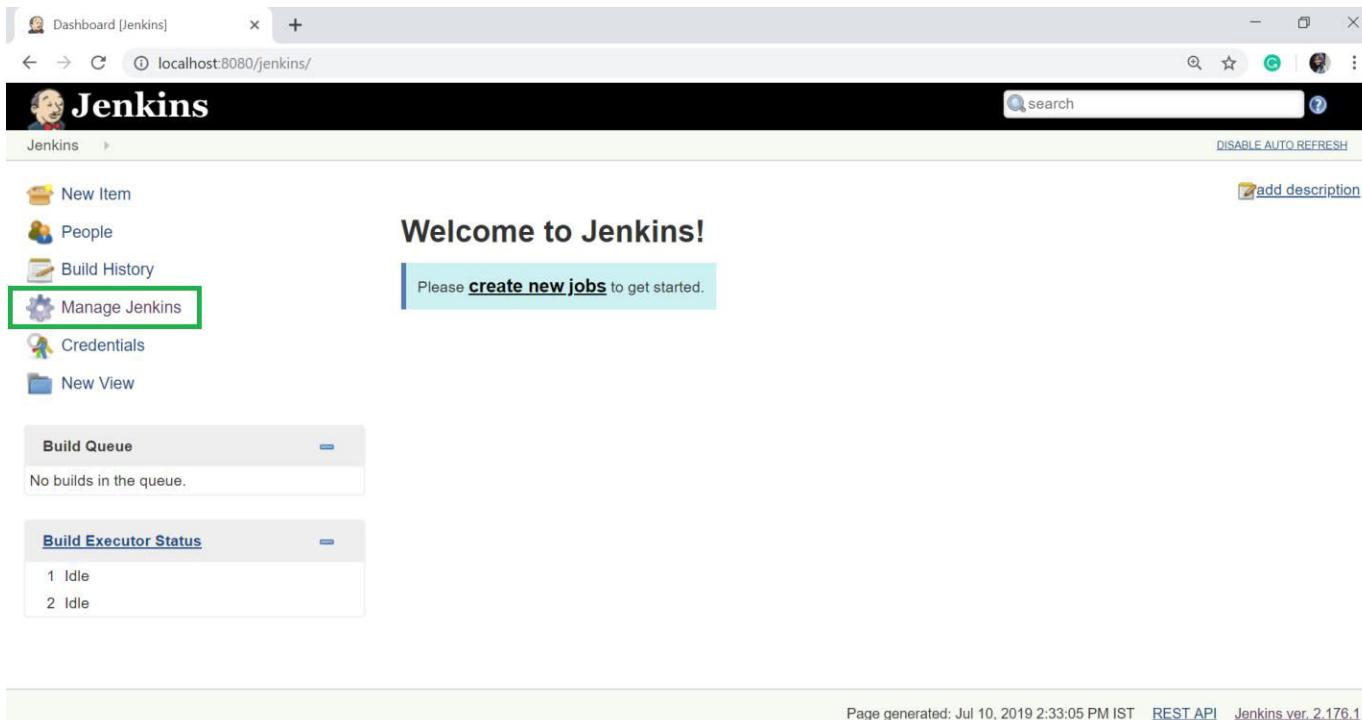
- Install Jenkins
 - Now, install the latest version of Jenkins by using the following command:
`$ sudo apt install jenkins`
 - Verify installation
`$ systemctl status Jenkins`
- Adjusting Firewall
 - If you are installing Jenkins on a remote server of Ubuntu that is protected by a firewall, you will need to open port **8080**. Consider that you are using **UFW** to manage your firewall; you can open the port with the following command:
`$ sudo ufw allow 8080`
- Setting Up Jenkins
 - To set up the new Jenkins installation, open the browser, type the domain or IP address followed by port 8080, `http://your_ip_or_domain:8080`, and screen (unlock Jenkins screen) using administrator credentials. To see the administrator password:
`$ sudo cat /var/lib/jenkins/secrets/initialAdminPassword`

GitHub Setup for Jenkins

- Github is fast becoming one of the most popular source code management systems.
- It is a web based repository of code which plays a major role in DevOps. GitHub provides a common platform for many developers working on the same code or project to upload and retrieve updated code, thereby facilitating continuous integration.
- Jenkins works with Git through the Git plugin.
- Connecting a GitHub private repository to a private instance of Jenkins can be tricky.

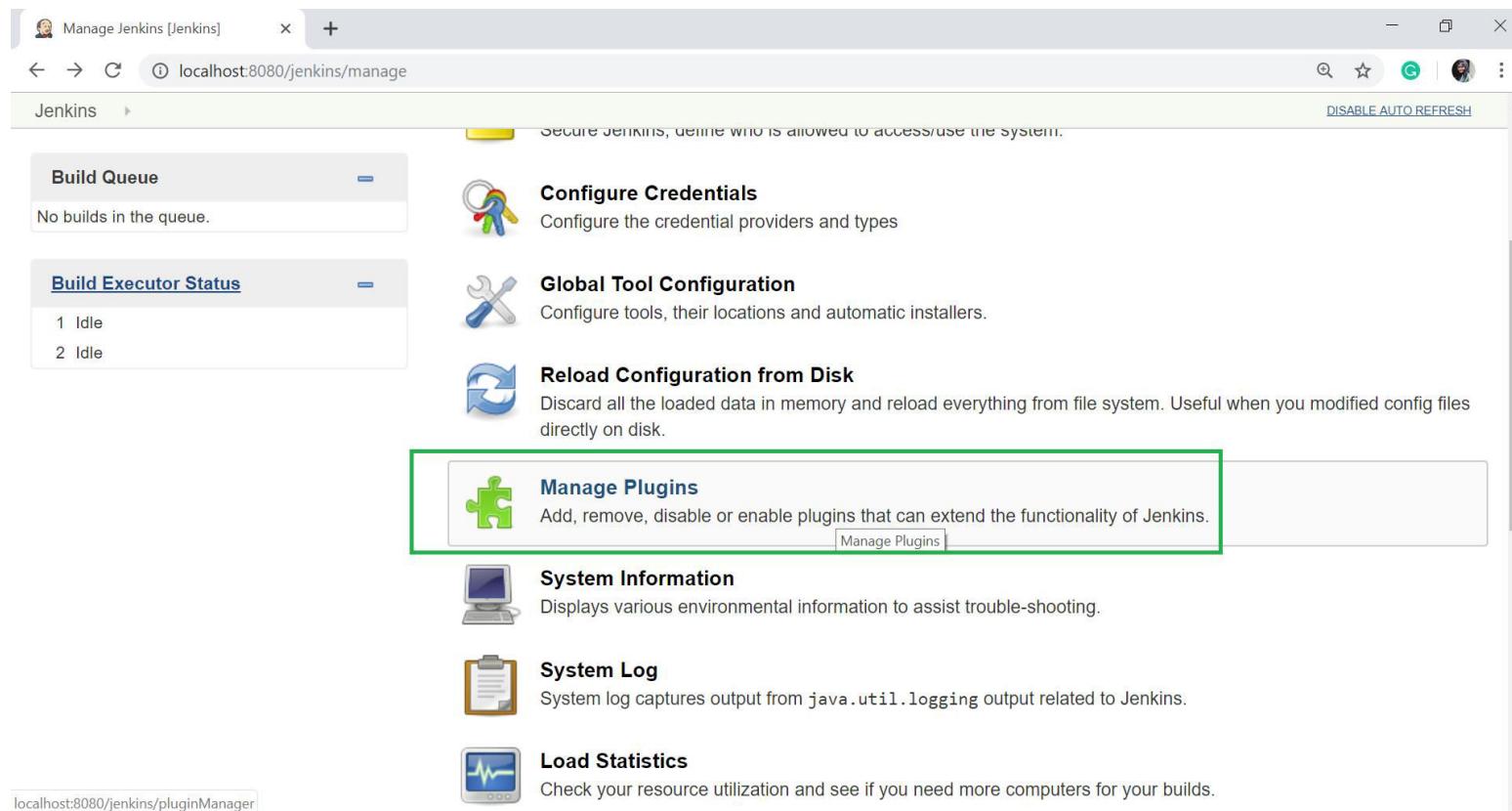
GitHub Setup for Jenkins

- In the Home screen of the Jenkins (Jenkins Dashboard), click on the **Manage Jenkins** option on the left hand side of the screen.



GitHub Setup for Jenkins

- Now, click on the **Manage Plugins** option



GitHub Setup for Jenkins

- In the next page, click on the "Available tab".

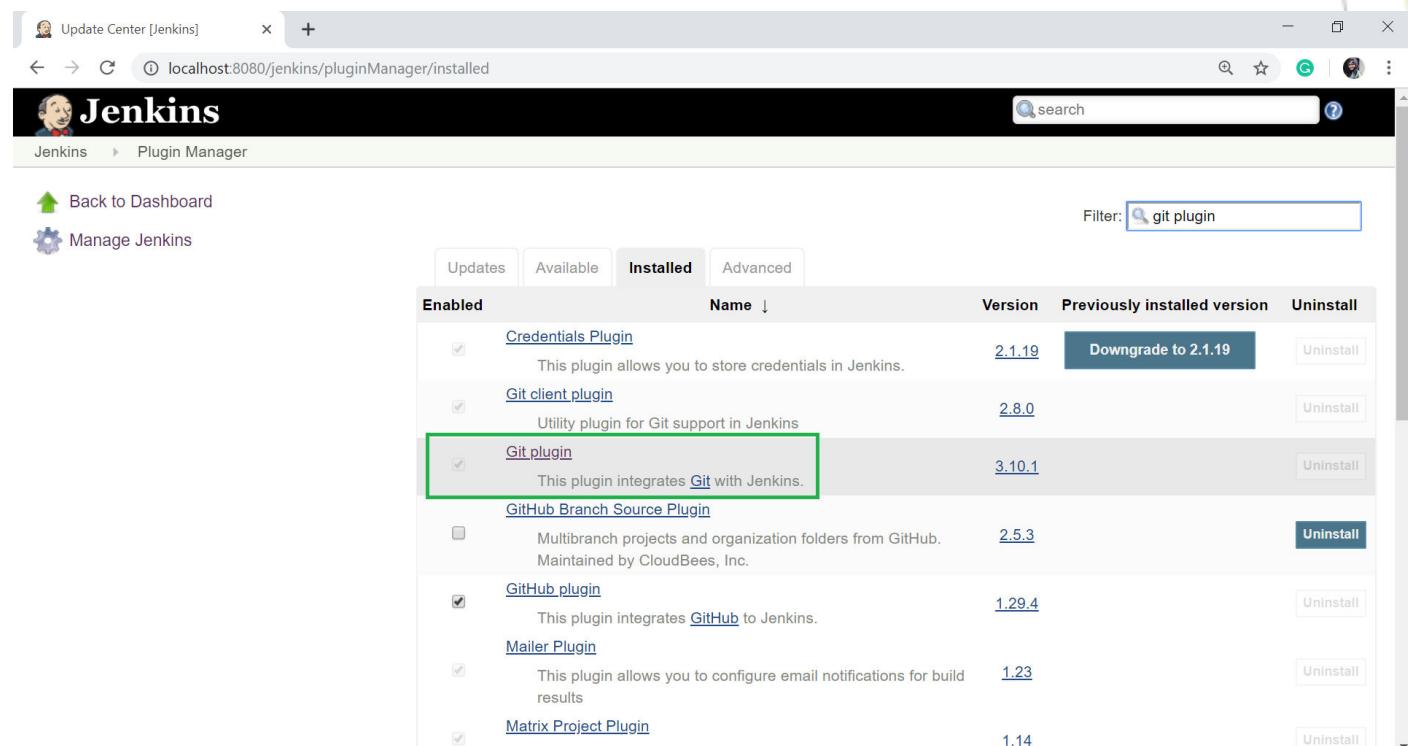
The screenshot shows the Jenkins Plugin Manager interface. At the top, there is a navigation bar with links for 'Back to Dashboard' and 'Manage Jenkins'. Below the navigation bar, there is a search bar and a filter input field. A green box highlights the 'Available' tab in the top navigation bar, which is currently active. The main content area displays a table with columns: 'Install', 'Name ↓', 'Version', and 'Installed'. A message 'No updates' is shown below the table. At the bottom of the page, there is a status message 'Update information obtained: 16 hr ago' and a 'Check now' button. The URL in the browser's address bar is 'localhost:8080/jenkins/pluginManager/available'.

GitHub Setup for Jenkins

- The "Available" tab gives a list of plugins which are available for downloading. In the Filter tab type, type the "Git Plugin".
- Select the Git Plugin.
- Click on the "**install without restart**". The plugin will take some time to finish downloading depending on your internet connection, and will be installed automatically.
- You can also click on "**Download now and install after restart**" button in which the git plugin is installed after restart.
- If you already have the Git plugin installed then go to "Installed" tab and in filter option type Git plugin.

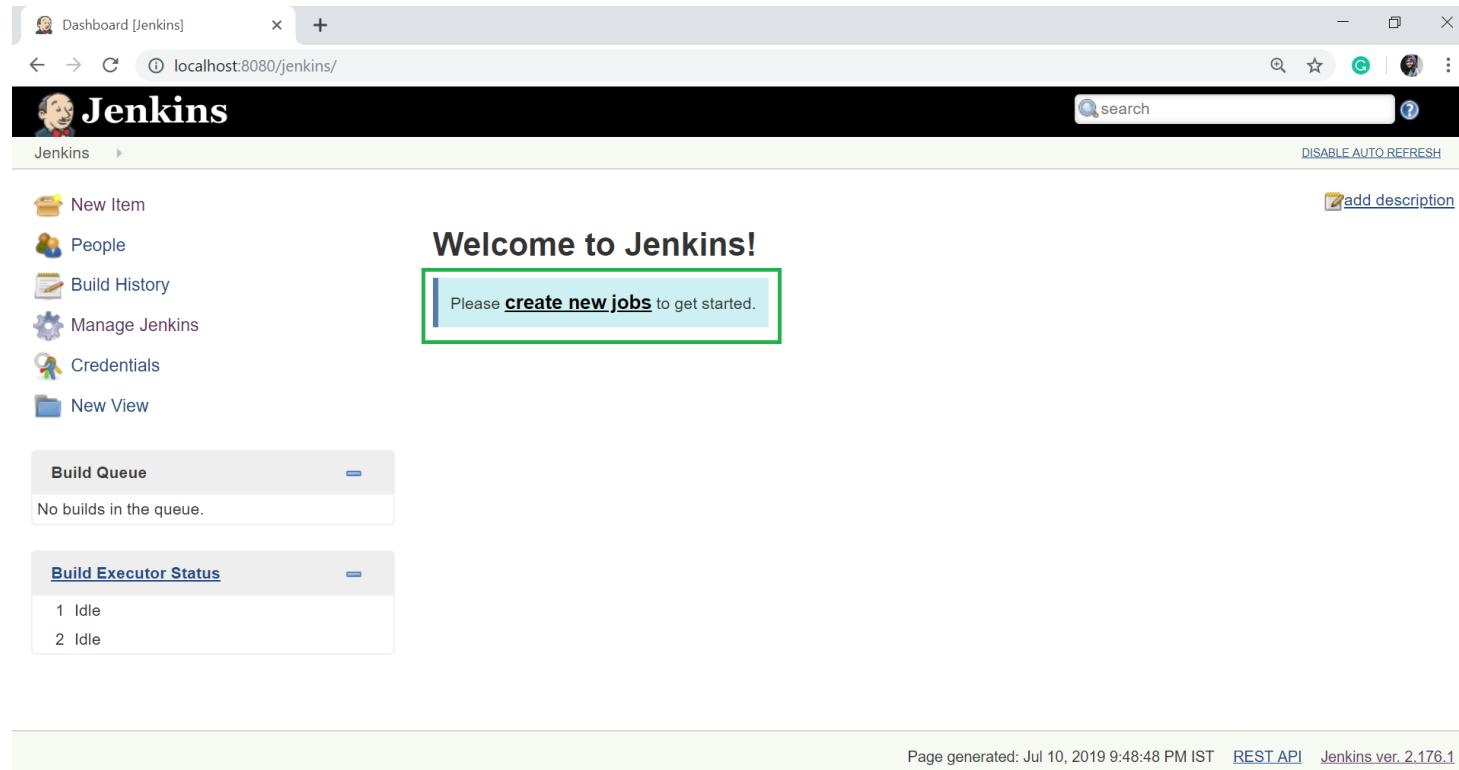
GitHub Setup for Jenkins

- Once all the installations are completed, restart Jenkins by giving the following command in the browser.
`http://localhost:8080/jenkins/restart`
- After Jenkins is restarted, Git will available as an option while configuring jobs



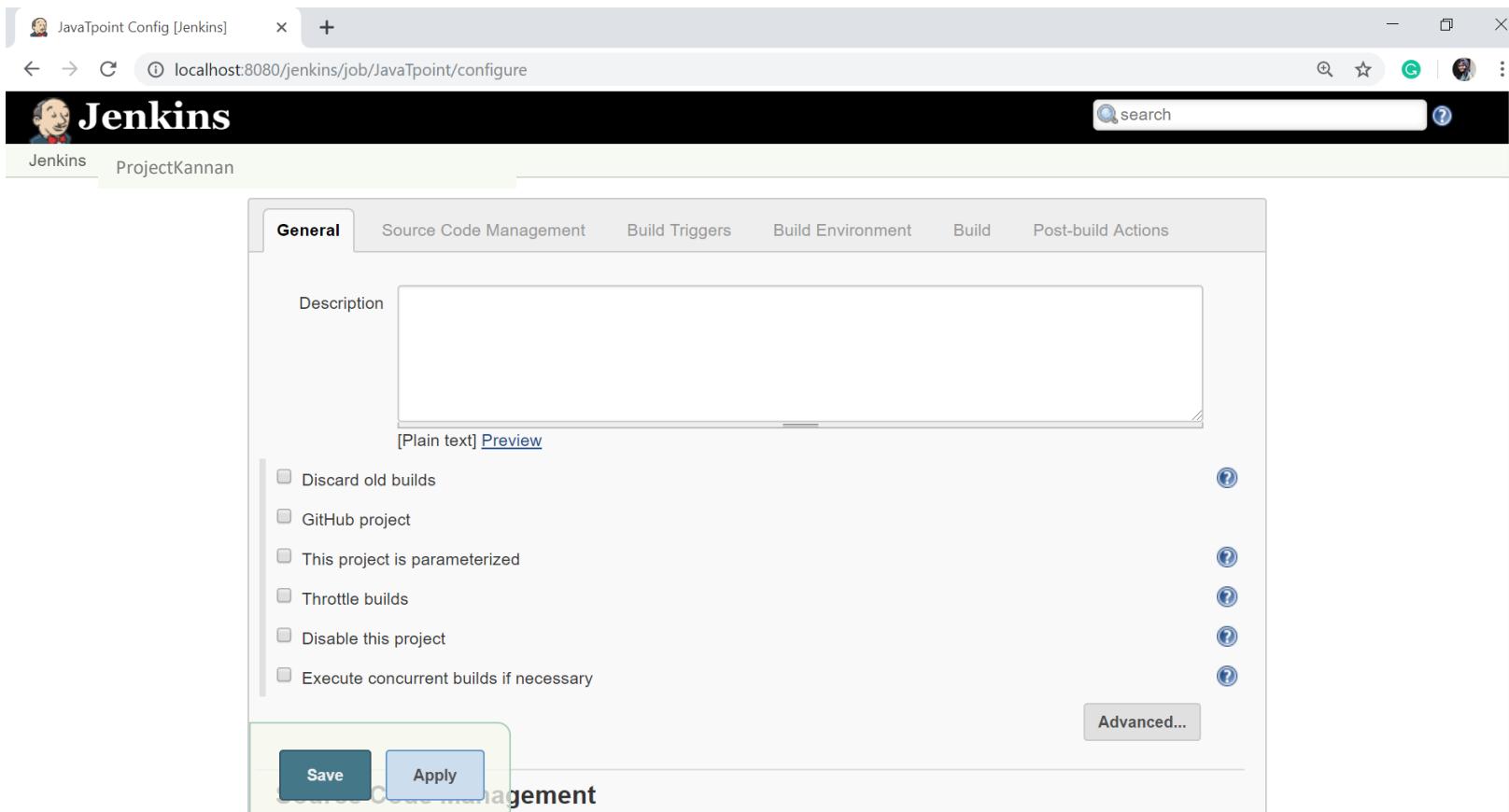
Integrating Jenkins with Git

- First create a new job in Jenkins, open the Jenkins Dashboard and click on "create new jobs".



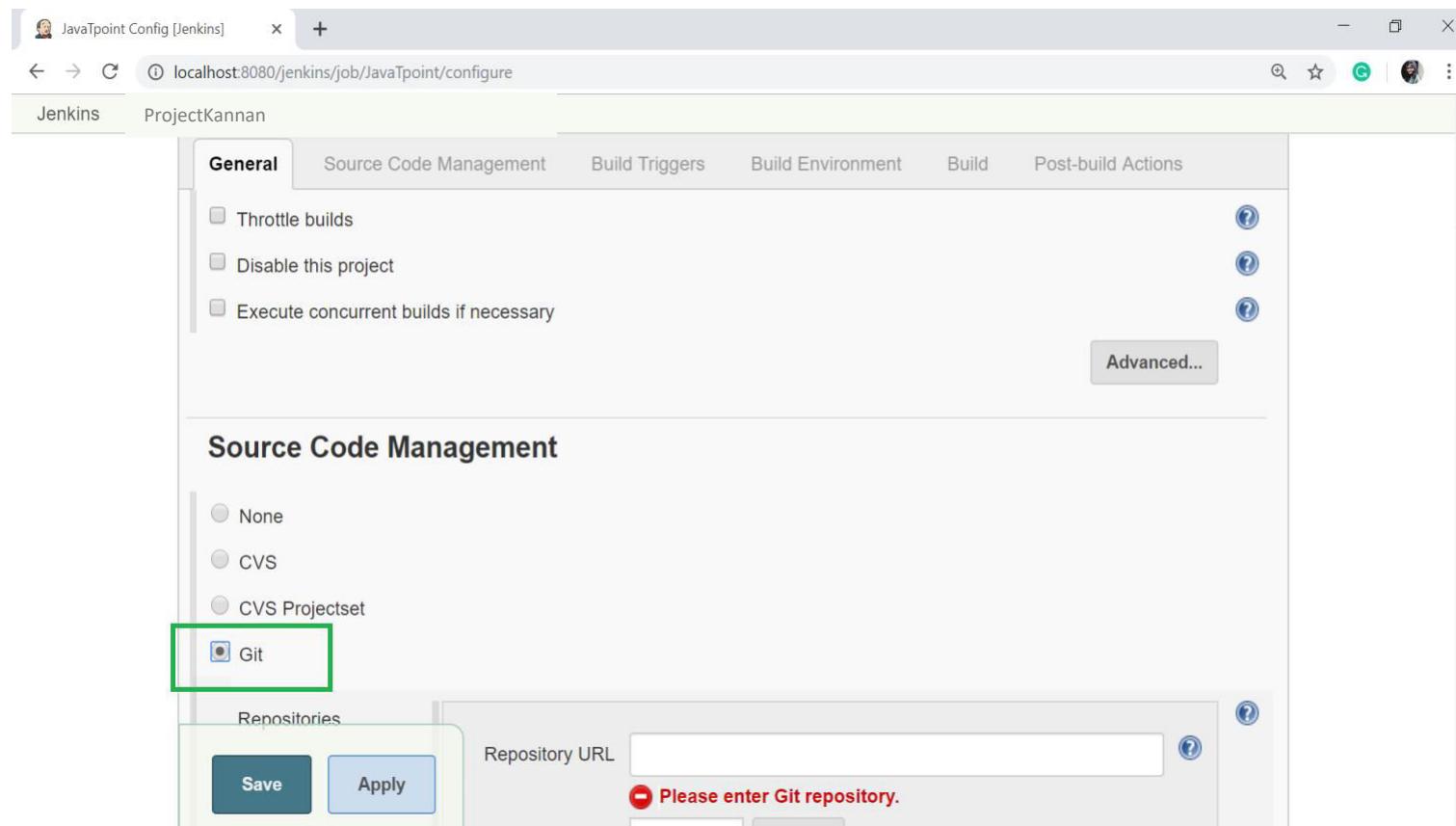
Integrating Jenkins with Git

- Once you click OK, the page will be redirected to its project configuration. Enter the project information



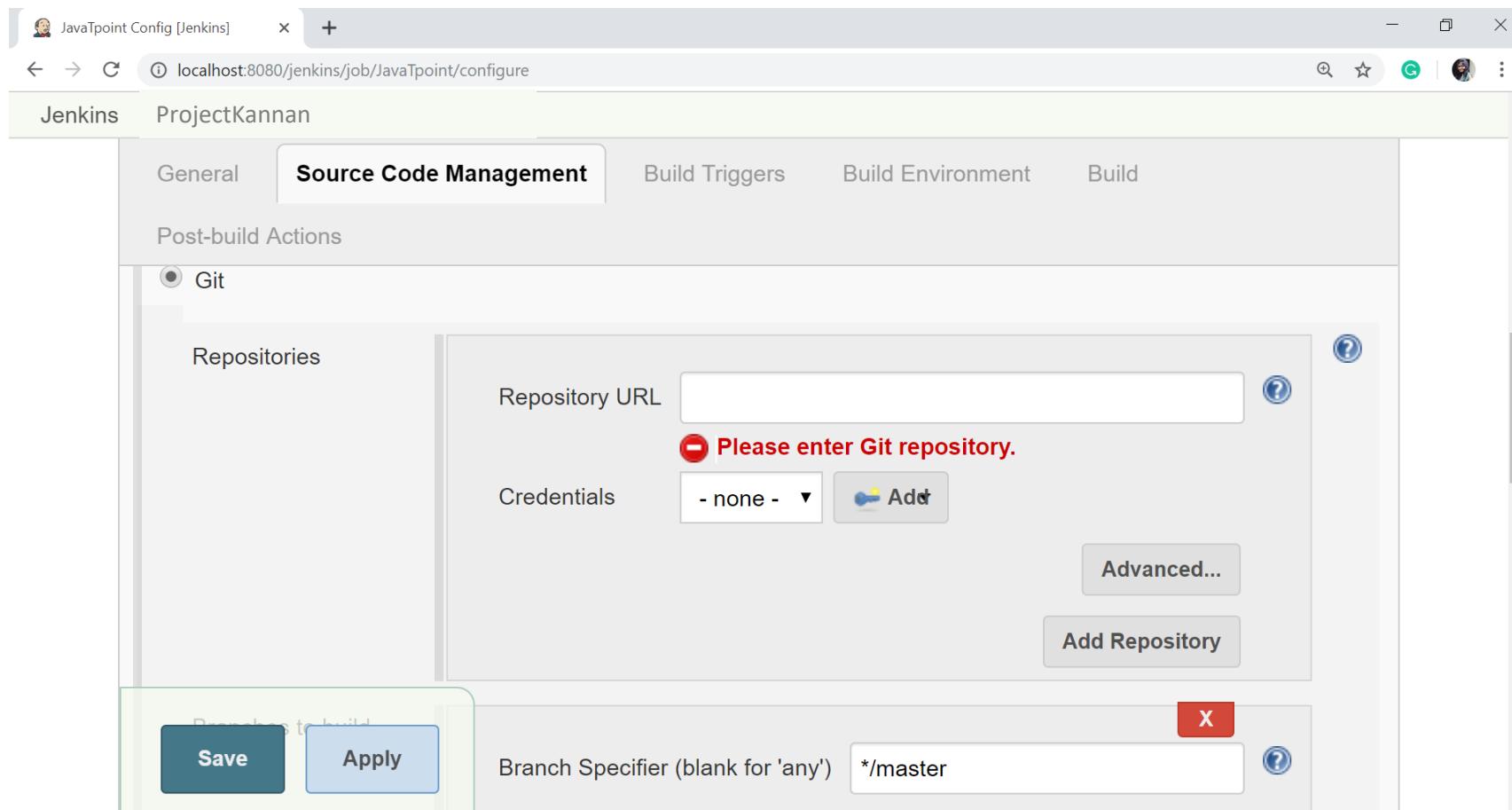
Integrating Jenkins with Git

- Now, under the "Source Code Management" you will see the Git option, if your **Git** plugin has been installed in Jenkins:



Integrating Jenkins with Git

- Enter the Git repository URL on the "Repository URL" option to pull the code from GitHub.

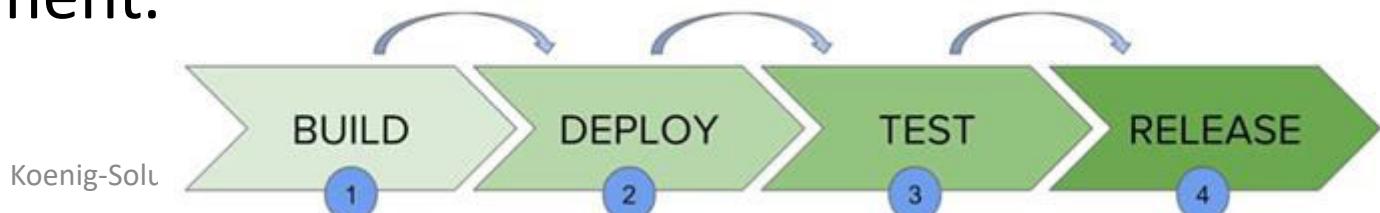


Jenkins Pipeline

- In Jenkins, a pipeline is a collection of events or jobs which are interlinked with one another in a sequence.
- It is a combination of plugins that support the integration and implementation of **continuous delivery pipelines** using Jenkins.
- In other words, a Jenkins Pipeline is a collection of jobs or events that brings the software from version control into the hands of the end users by using automation tools. It is used to incorporate continuous delivery in our software development workflow.
- A pipeline has an extensible automation server for creating simple or even complex delivery pipelines "as code", via DSL (Domain-specific language).

What is Continuous Delivery Pipeline?

- It contains a collection of states such as build, deploy, test and release. These jobs or events are interlinked with each other. Every state has its jobs, which work in a sequence called a continuous delivery pipeline.
- A continuous delivery pipeline is an automated expression to show your process for getting software for version control. Thus, every change made in your software goes through a number of complex processes on its manner to being released.
- It also involves developing the software in a repeatable and reliable manner, and progression of the built software through multiple stages of testing and deployment.



JenkinsFile

- Jenkins Pipeline can be defined by a text file called JenkinsFile.
- We can implement pipeline as code using JenkinsFile, and this can be defined by using a DSL (Domain Specific Language).
- With the help of JenkinsFile, we can write the steps required for running a Jenkins Pipeline.

JenkinsFile

- The benefits of using JenkinsFile are:
 - You can make pipelines automatically for all branches and can execute pull requests with just one JenkinsFile.
 - You can review your code on the pipeline.
 - You can review your Jenkins pipeline.
 - This is the singular source for your pipeline and can be customized by multiple users.

Jenkins Pipeline Syntax

- Two types of syntax are used for defining your JenkinsFile.
 - Declarative
 - Scripted
- **Declarative:**

Declarative pipeline syntax offers a simple way to create pipelines. It consists of a predefined hierarchy to create Jenkins pipelines. It provides you the ability to control all aspects of a pipeline execution in a simple, straightforward manner.
- **Scripted:**

Scripted Jenkins pipeline syntax runs on the Jenkins master with the help of a lightweight executor. It uses very few resources to convert the pipeline into atomic commands.
- Both scripted and declarative syntax are different from each other and are defined totally differently.

Why Use Jenkins Pipeline?

- You can create several automation jobs with the help of use cases, and run them as a Jenkins pipeline.
- Here are the reasons why you should use Jenkins pipeline:
 - Jenkins pipeline is implemented as a code which allows several users to edit and execute the pipeline process.
 - Pipelines are robust. So if your server undergoes an unpredicted restart, the pipeline will be automatically resumed.
 - You can pause the pipeline process and make it wait to continue until there is an input from the user.
 - Jenkins Pipelines support big projects. You can run many jobs, and even use pipelines in a loop.

Jenkin Pipeline Concepts

- **Pipeline:** This is the user-defined block, which contains all the processes such as build, test, deploy, etc. it is a group of all the stages in a JenkinsFile. All the stages and steps are defined in this block. It is used in declarative pipeline syntax.

```
pipeline{  
}
```

Jenkin Pipeline Concepts

- **Pipeline:** This is the user-defined block, which contains all the processes such as build, test, deploy, etc. it is a group of all the stages in a JenkinsFile. All the stages and steps are defined in this block. It is used in declarative pipeline syntax.

```
pipeline{  
}
```

- **Node:** The node is a machine on which Jenkins runs is called a node. A node block is used in scripted pipeline syntax.

```
node{  
}
```

Jenkin Pipeline Concepts

- **Stage:** This block contains a series of steps in a pipeline. i.e., build, test, and deploy processes all come together in a stage. Generally, a stage block visualizes the Jenkins pipeline process.
- Let's see an example for multiple stages, where each stage performs a specific task.

```
pipeline {  
    agent any  
    stages {  
        stage ('Build') {  
            ...  
        }  
        stage ('Test') {  
            ...  
        }  
        stage ('QA') {  
            ...  
        }  
        stage ('Deploy') {  
            ...  
        }  
        stage ('Monitor') {  
            ...  
        }  
    }  
}
```

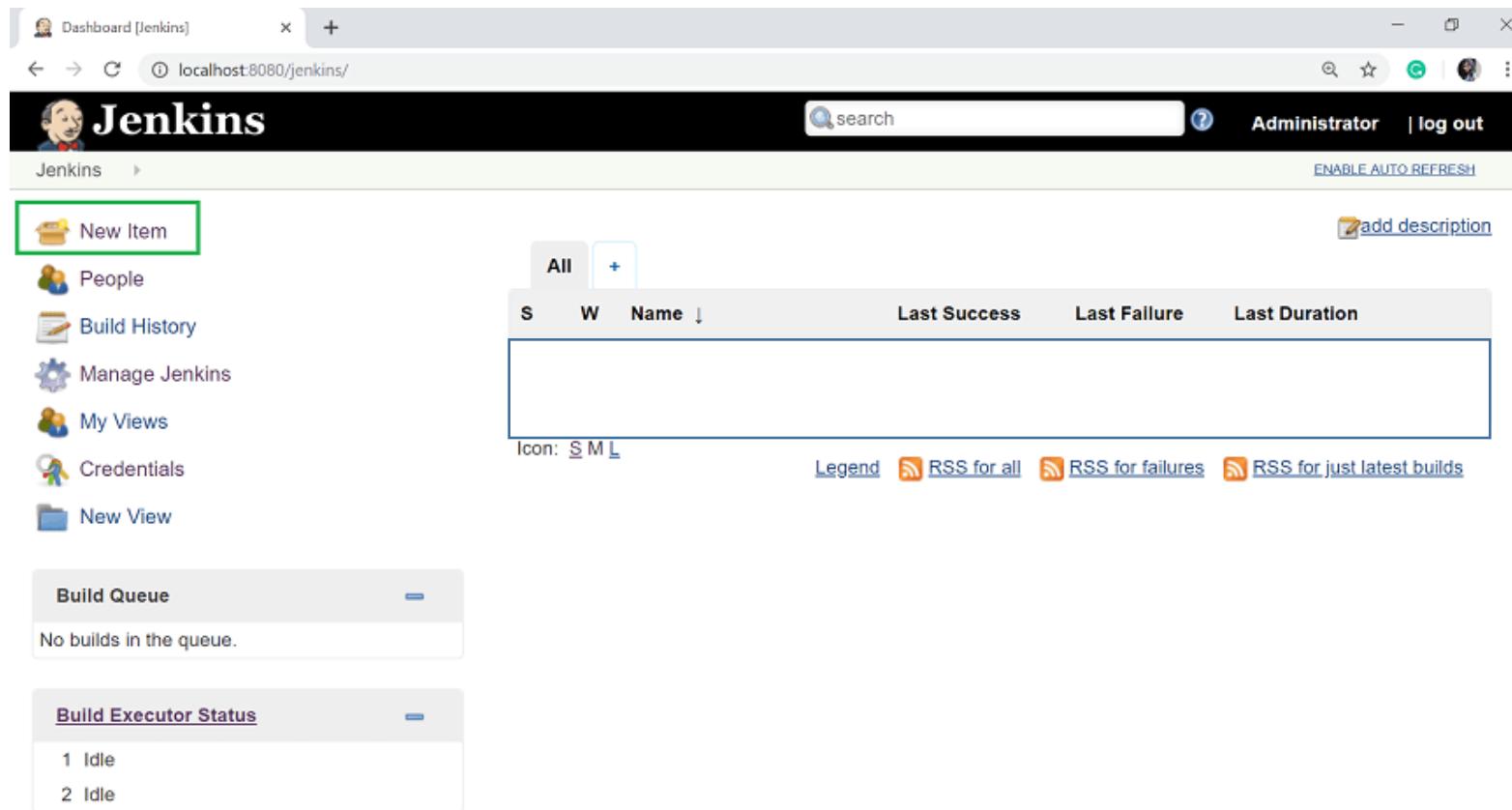
Jenkin Pipeline Concepts

- **Step:** A step is a single task that executes a specific process at a defined time. A pipeline involves a series of steps defined within a stage block.

```
pipeline {  
    agent any  
    stages {  
        stage ('Build') {  
            steps {  
                echo 'Running build phase...'  
            }  
        }  
    }  
}
```

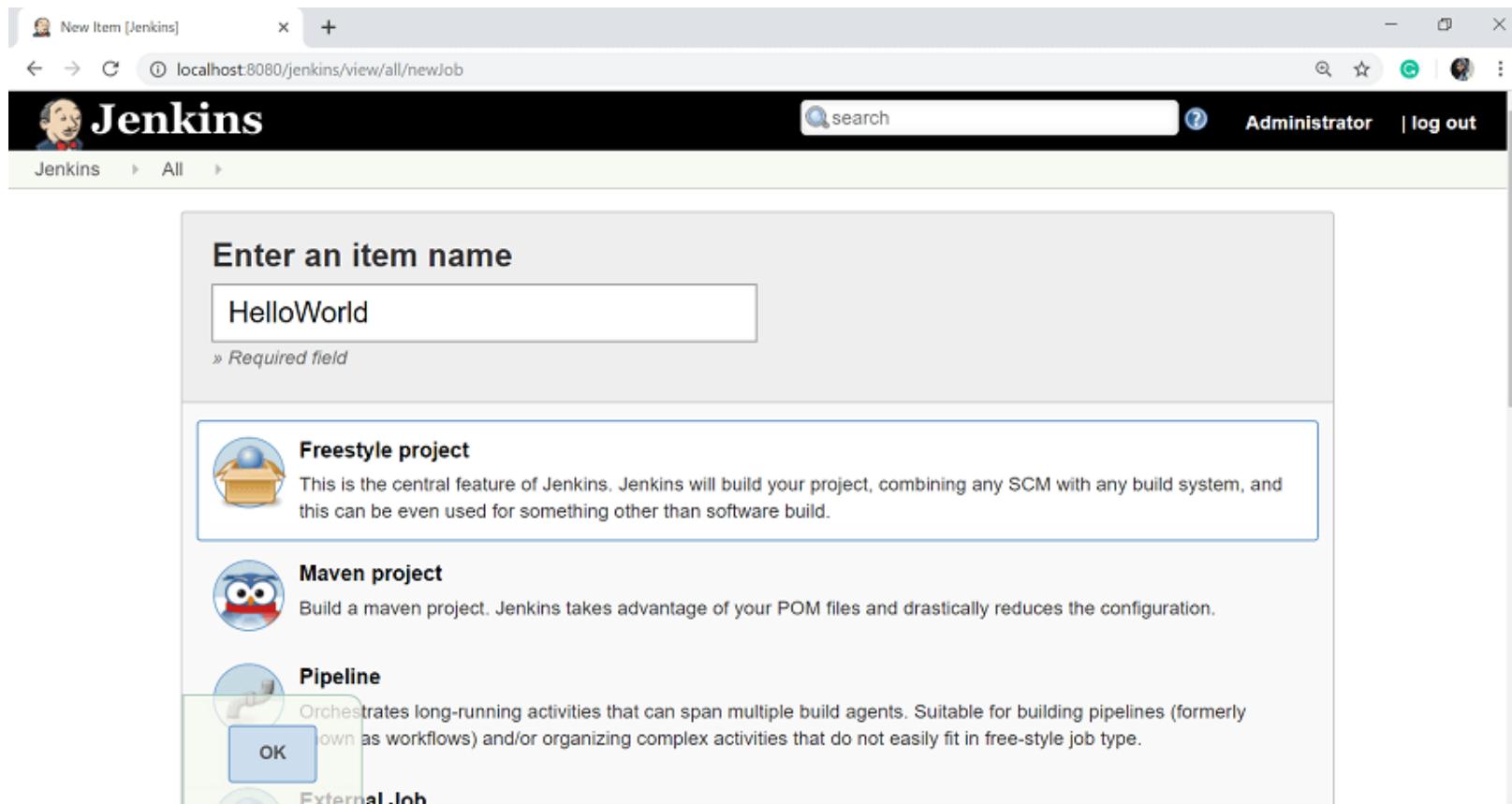
Jenkins – Build an “Hello world” Application

- Step 1: Go to the Jenkins dashboard and click on the New Item.



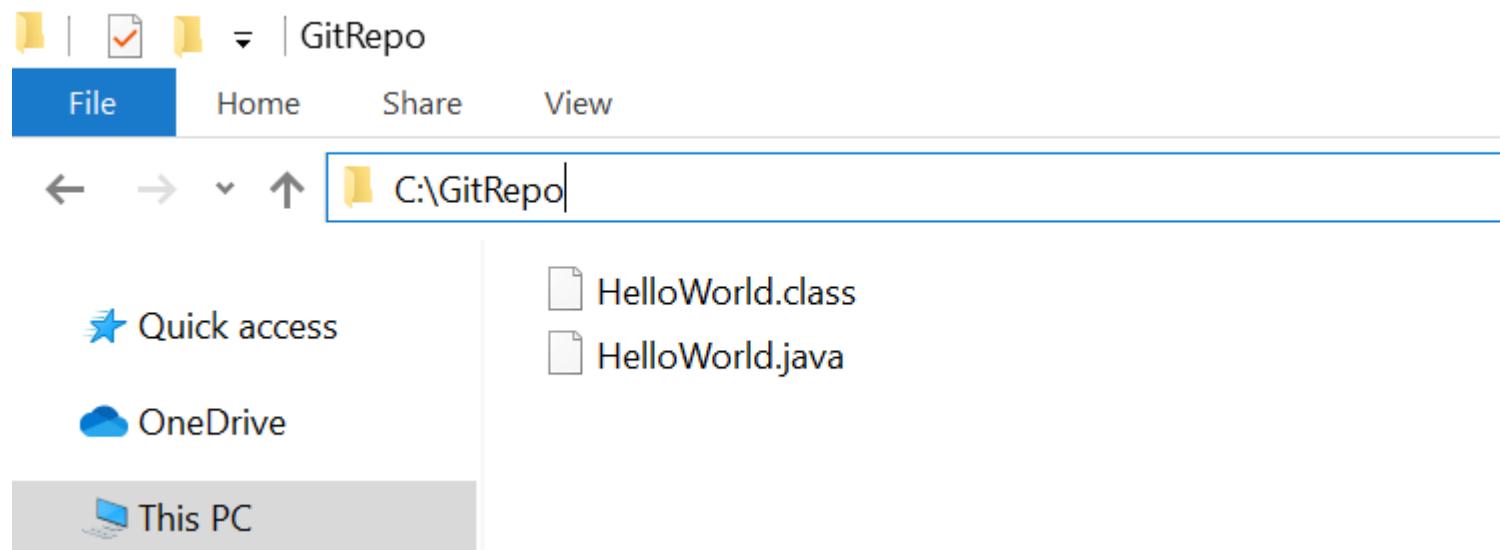
Jenkins – Build an “Hello world” Application

- **Step 2:** In the next page, enter the item name, and select the 'Freestyle project' option. And click OK. Here, my item name is HelloWorld.



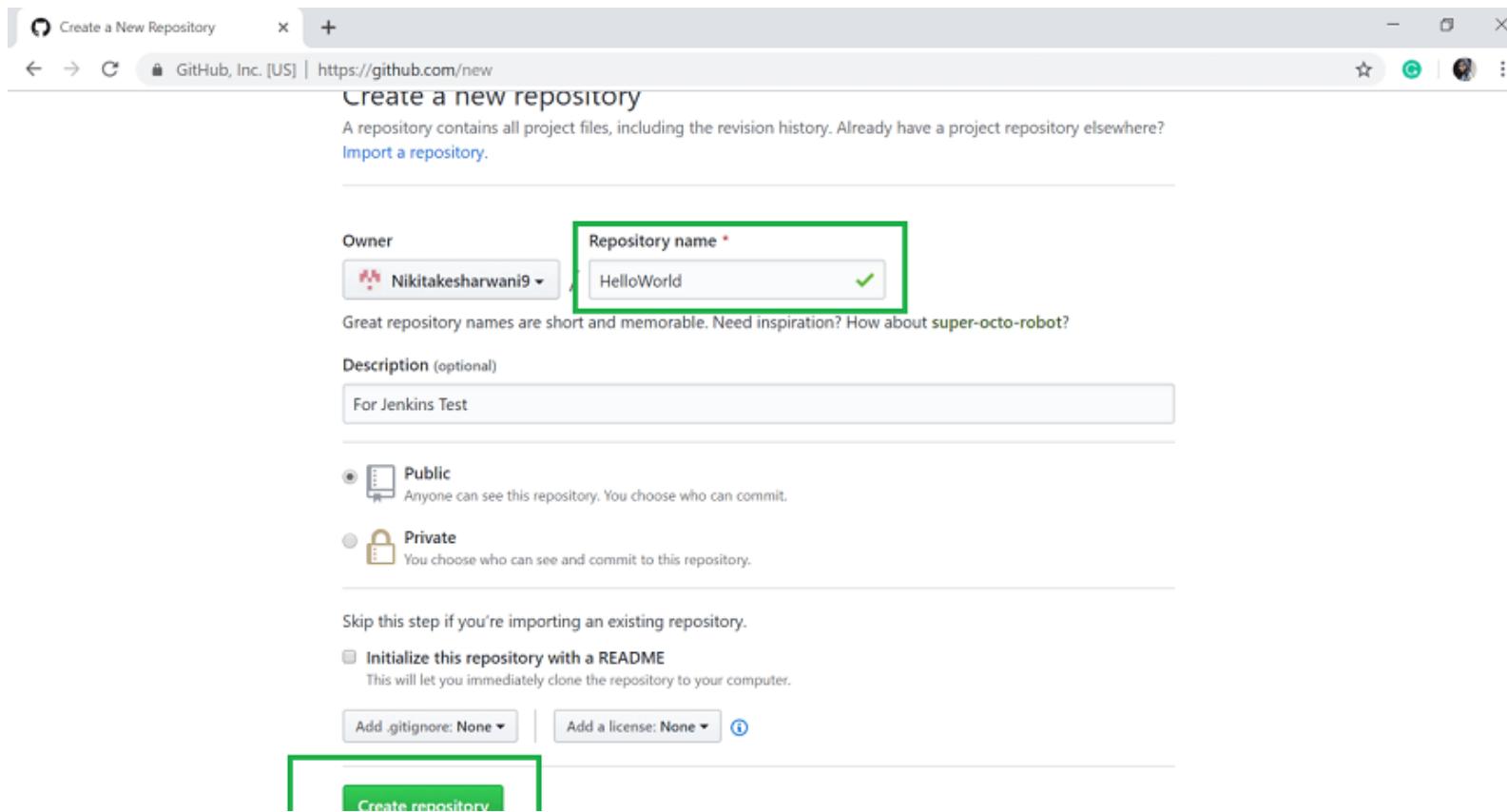
Jenkins – Build an “Hello world” Application

- **Step 4:** On the Source Code Management section, select the **Git** option, and specify the Repository URL.
- First, you have to create a project in java. Here, I created a simple **HelloWorld** program and saved it to one folder i.e. C:\GitRepo. Compile the HelloWorld.java file.



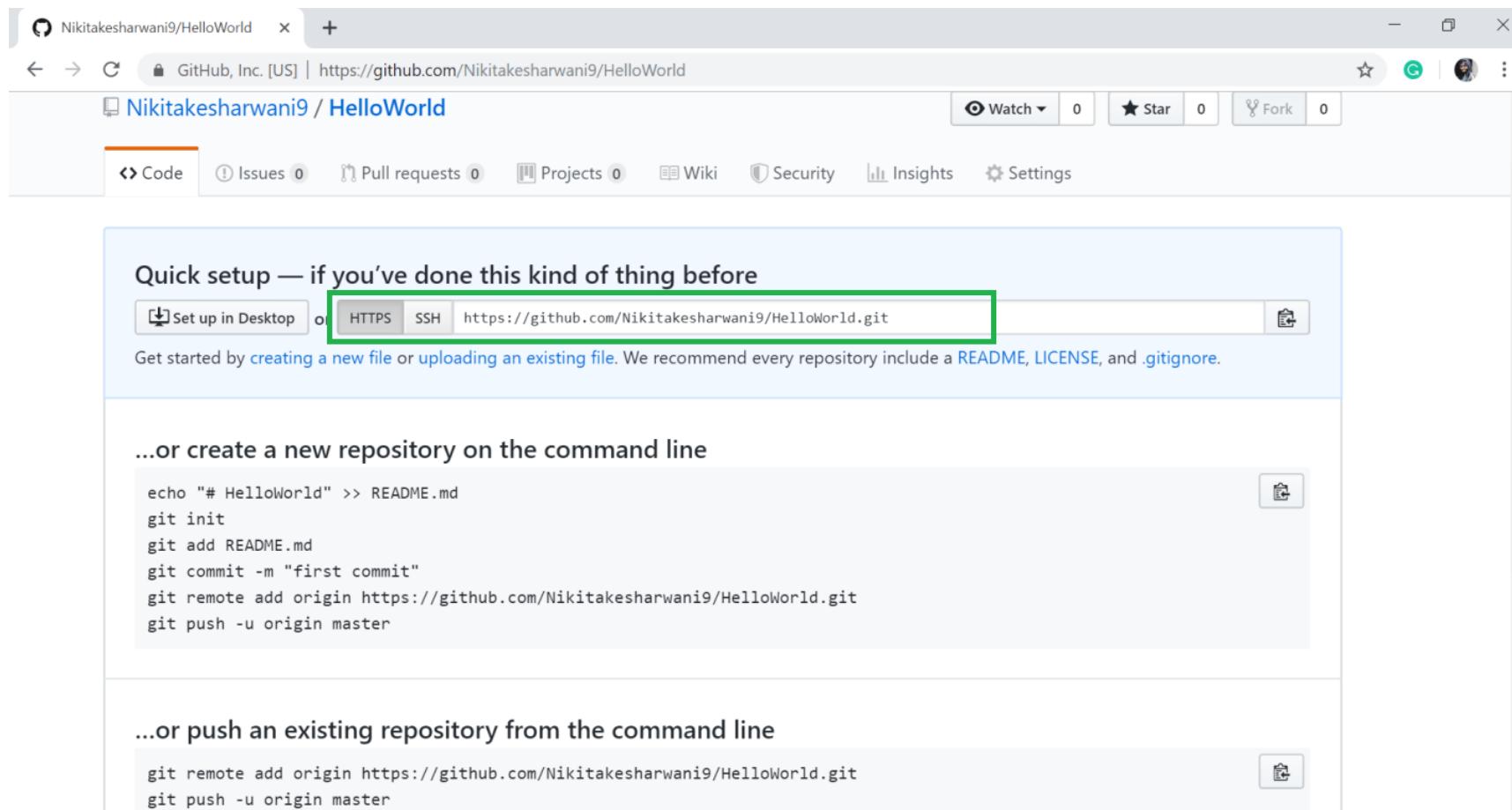
Jenkins – Build an “Hello world” Application

- Now create a project in your GitHub account and give the Repository name. Here my repository name is HelloWorld.



Jenkins – Build an “Hello world” Application

- Click on **Create repository**.



Jenkins – Build an “Hello world” Application

- Your repository is created. Copy the repository URL. My repository URL is: <https://github.com/xxxxxxxxxxxxxx>HelloWorld.git>
- Open the command prompt in your Windows and go to the path where your java file is created.
- Then run the following command.
 - git init
 - git status
 - git add .
 - git status

Jenkins – Build an “Hello world” Application

```
Command Prompt
c:\GitRepo>git init
Reinitialized existing Git repository in c:/GitRepo/.git/
c:\GitRepo>git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    HelloWorld.class
    HelloWorld.java

nothing added to commit but untracked files present (use "git add" to track)

c:\GitRepo>git add .

c:\GitRepo>git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

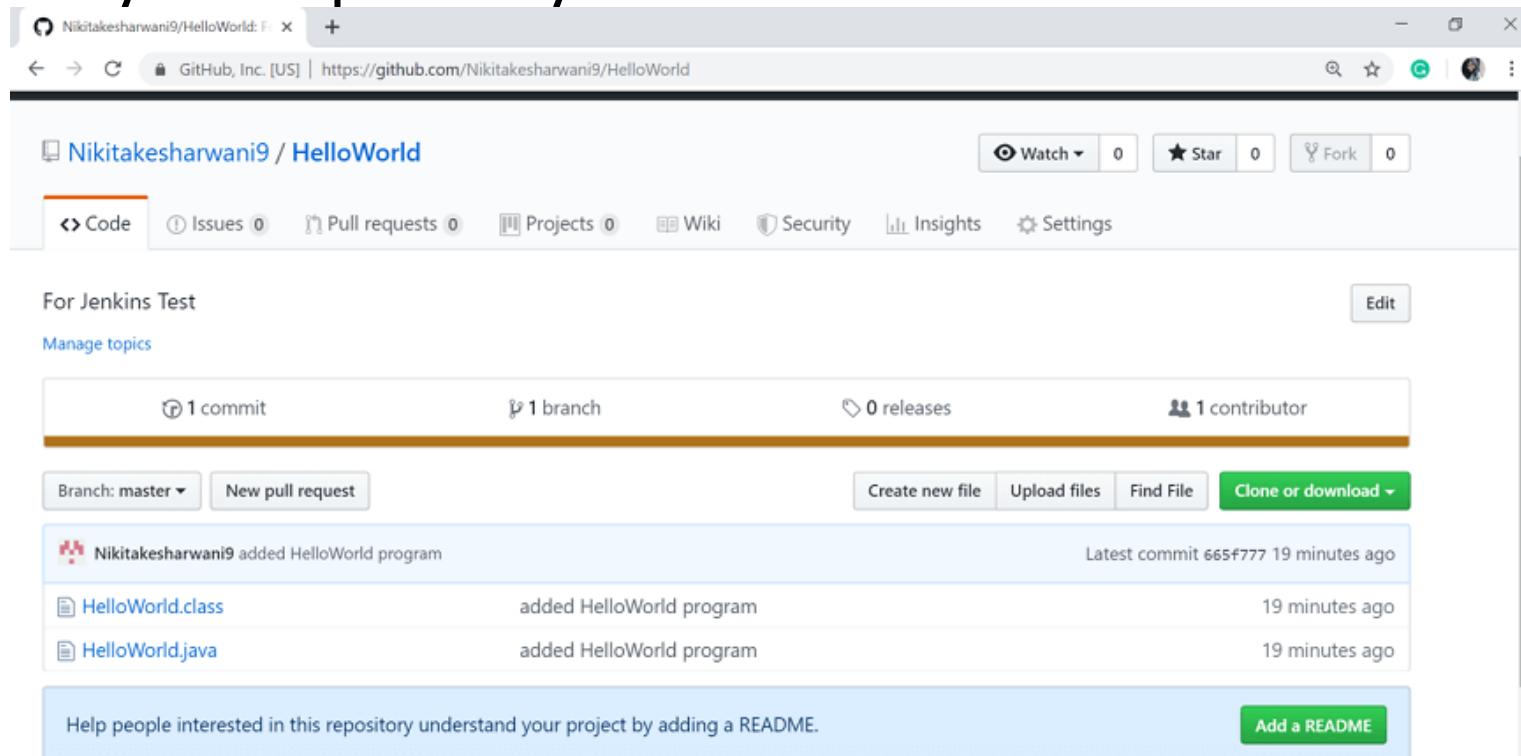
    new file:   HelloWorld.class
    new file:   HelloWorld.java
```

Jenkins – Build an “Hello world” Application

- Configure your GitHub account in your system.
 - `git config --global user.email "your@email"`
 - `git config --global user.name "username"`
- Commit it and add the repository URL.
 - `git commit -m "added HelloWorld program"`
 - `git remote add origin https://github.com/xxxxxxxxxxxxxx>HelloWorld.git`
 - `git push -u origin master`

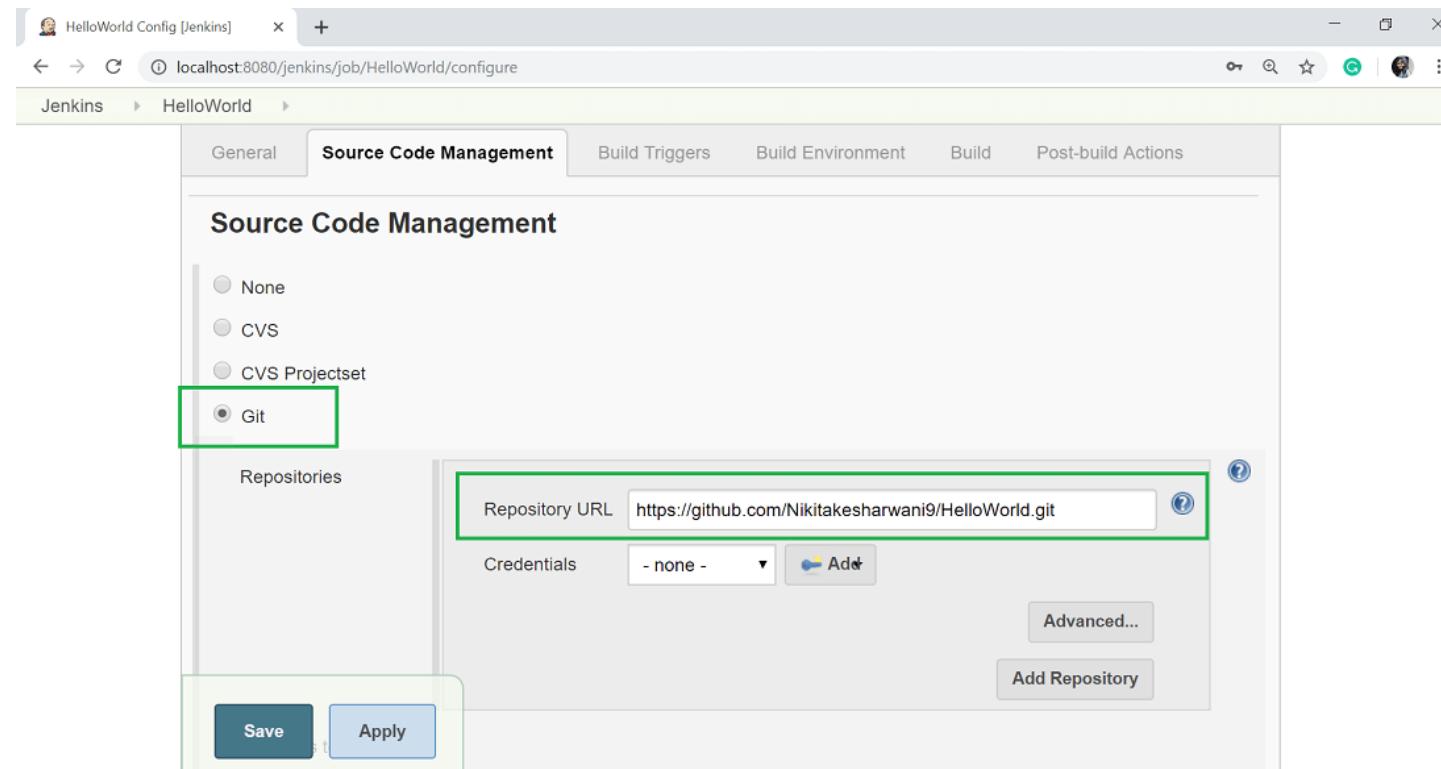
Jenkins – Build an “Hello world” Application

- Now, when you refresh your GitHub account, the helloWorld file will be added in your repository.



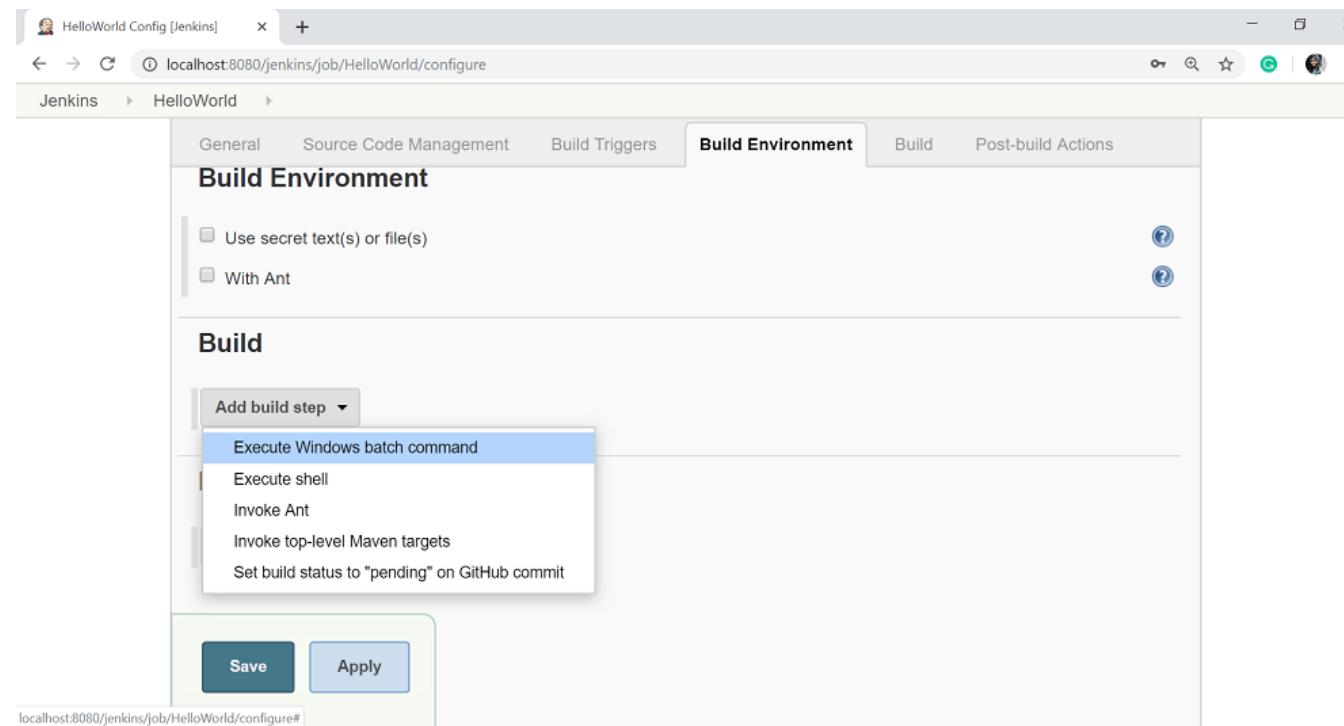
Jenkins – Build an “Hello world” Application

- **Step 5:** Add the Repository URL in the **Source Code Management** section.



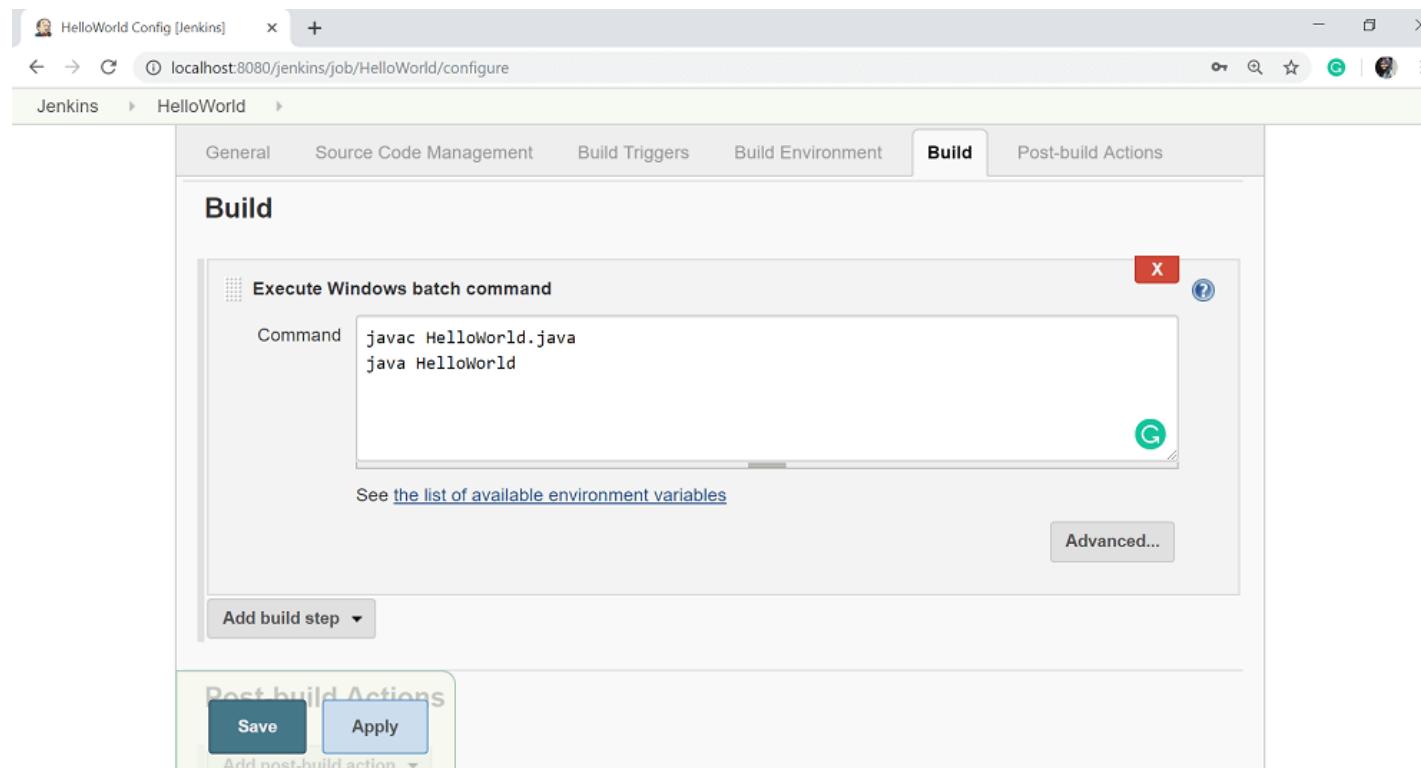
Jenkins – Build an “Hello world” Application

- **Step 6:** Now, it is time to build the code. Click on "Add build step" and select the "**Execute Windows batch command**".



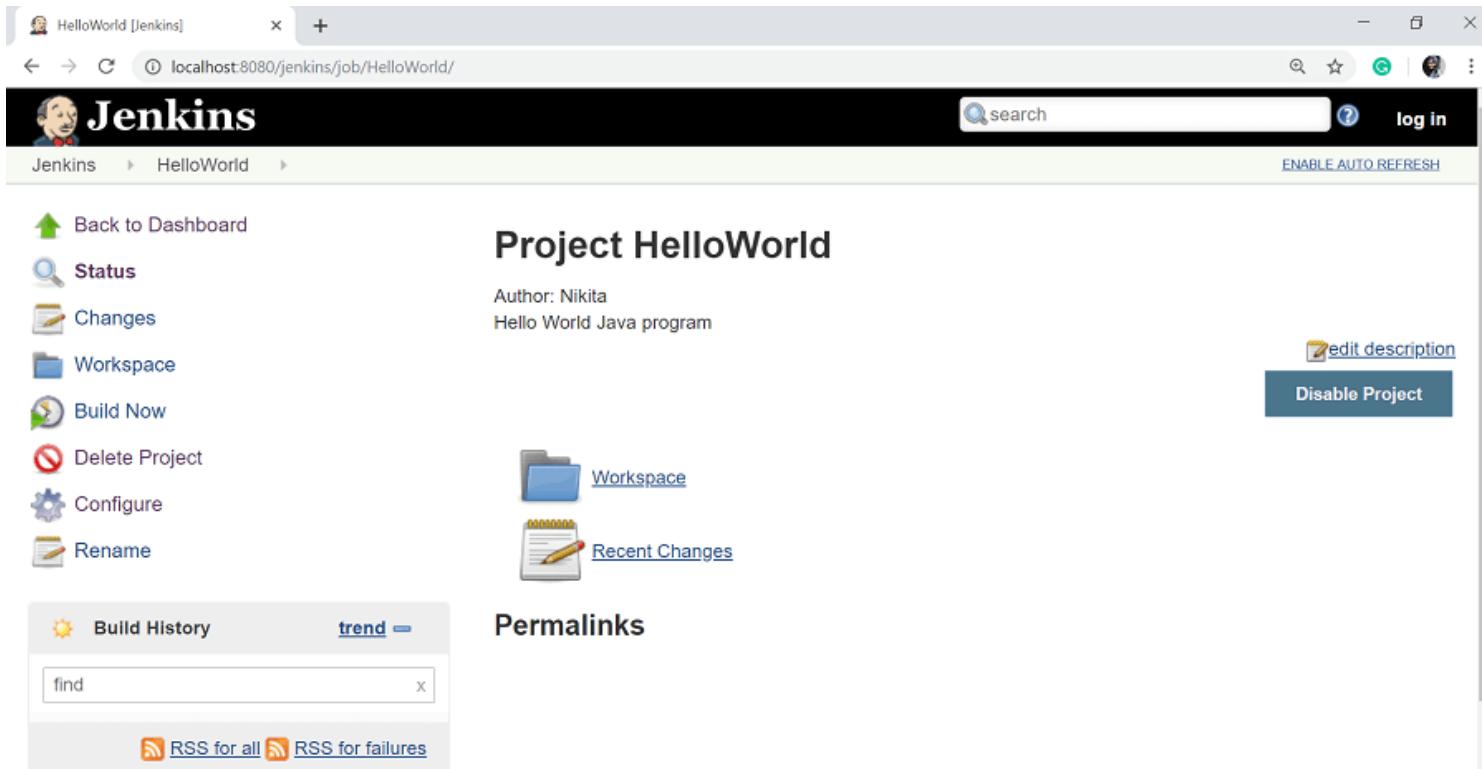
Jenkins – Build an “Hello world” Application

- **Step 7:** Enter the following command to compile the java code.
- **Step 8:** Click Apply and then Save button.



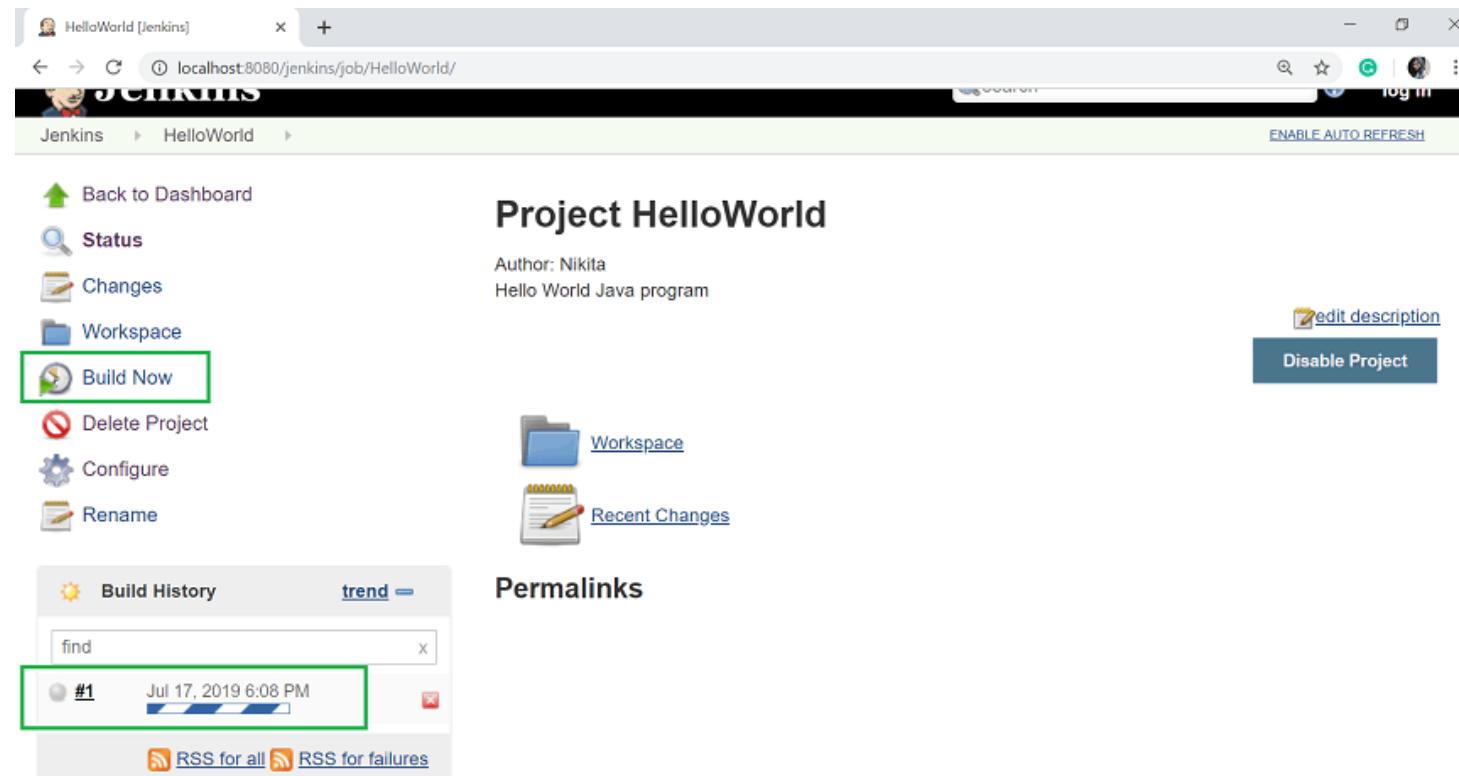
Jenkins – Build an “Hello world” Application

- **Step 9:** Once you saved the configuration, then now can click on **Build Now** option.



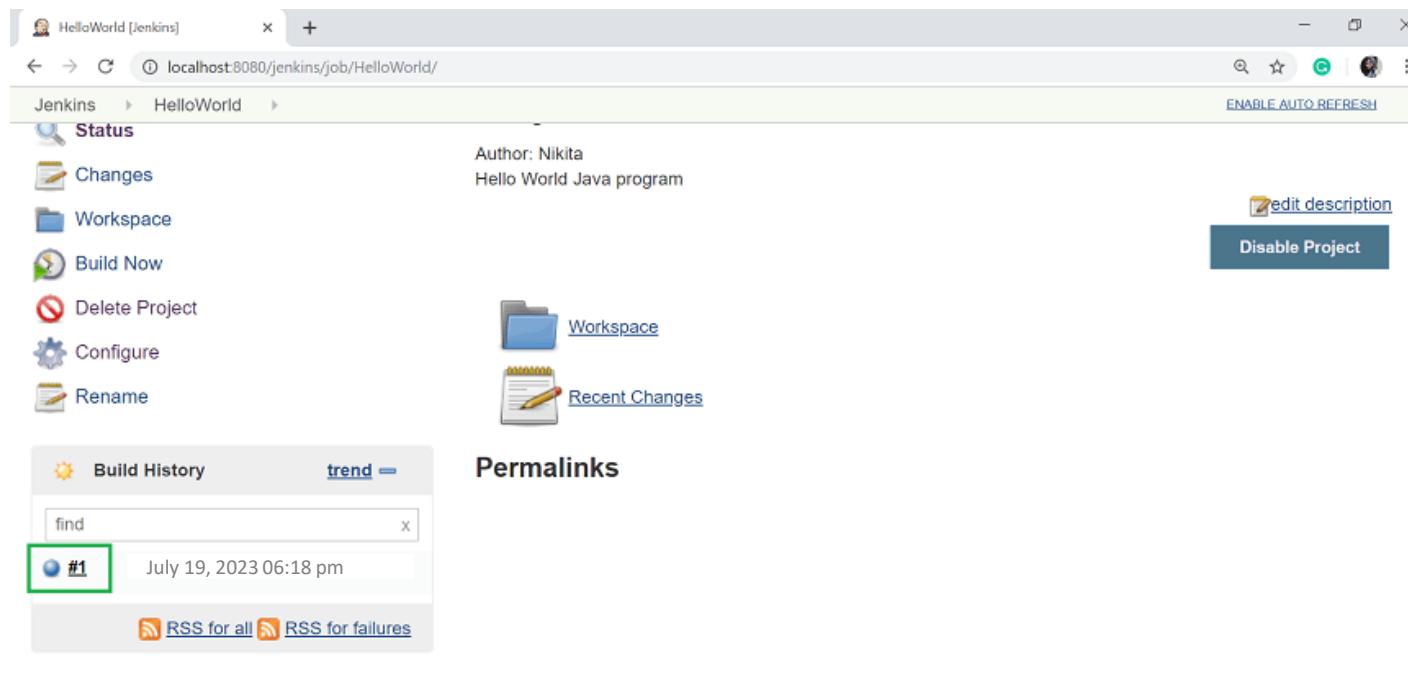
Jenkins – Build an “Hello world” Application

- **Step 10:** After clicking on **Build Now**, you can see the status of the build on the Build History section.



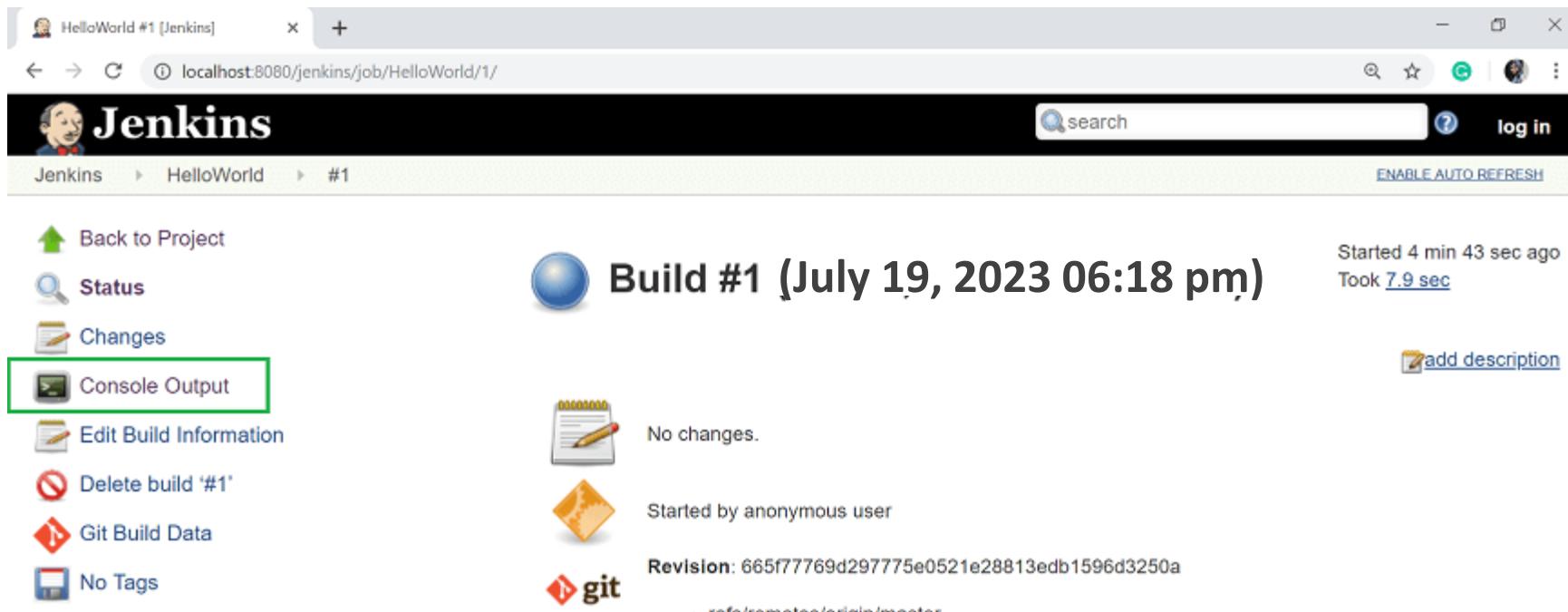
Jenkins – Build an “Hello world” Application

- Once the build is completed, a status of the build will show if the build was successful or not. If the build is failed then it will show in red color. Blue symbol is for success.



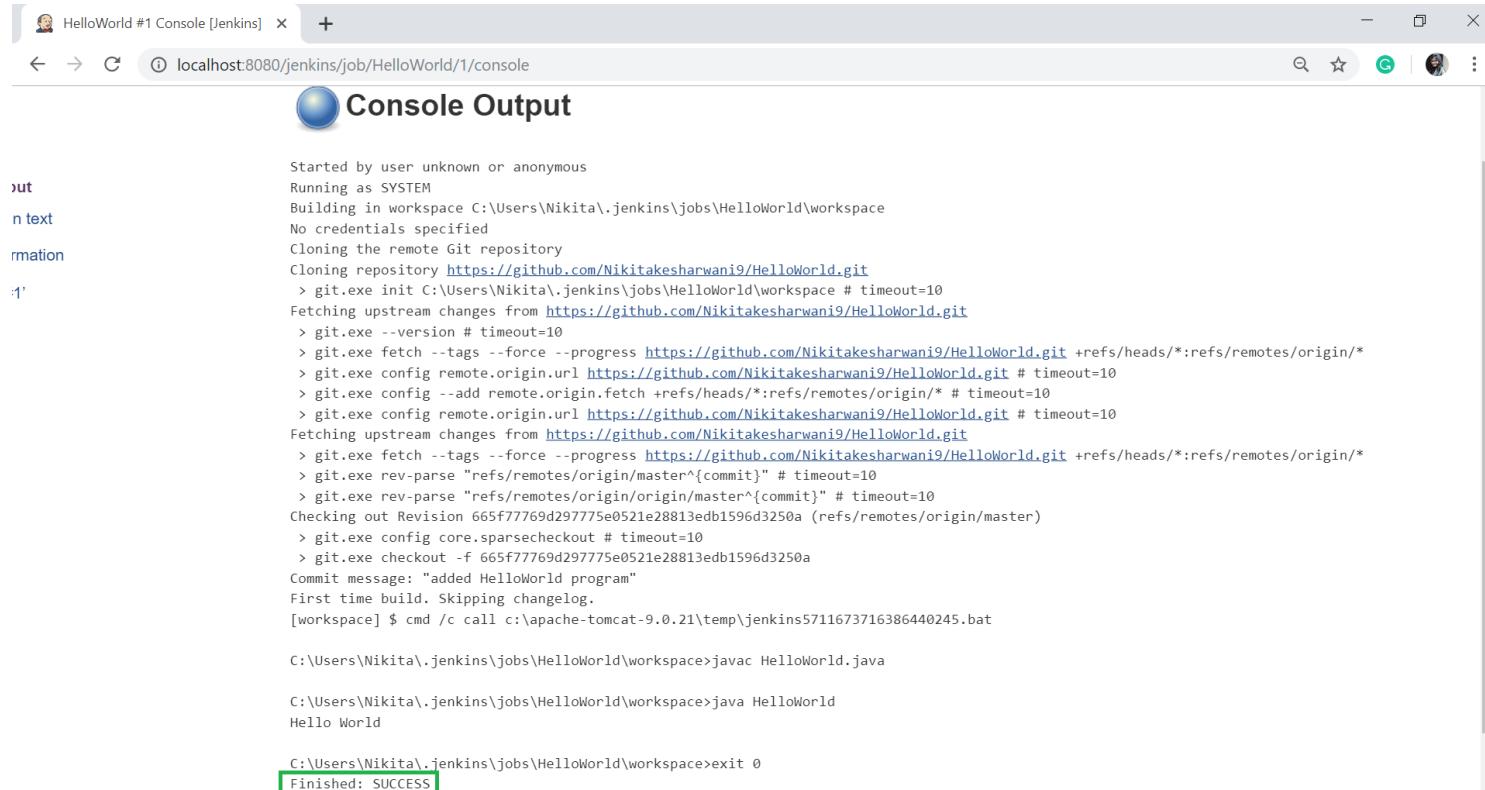
Jenkins – Build an “Hello world” Application

- Click on the build number **#1** in the **Build History section** to see the details of the build.



Jenkins – Build an “Hello world” Application

- **Step 11:** Click on **Console Output** from the left side of the screen to see the status of the build you run. It should show the success message.



The screenshot shows a browser window titled "HelloWorld #1 Console [Jenkins]". The main content is the "Console Output" tab, which displays the following log:

```
Started by user unknown or anonymous
Running as SYSTEM
Building in workspace C:\Users\Nikita\.jenkins\jobs\HelloWorld\workspace
No credentials specified
Cloning the remote Git repository
Cloning repository https://github.com/Nikitakesharwani9>HelloWorld.git
> git.exe init C:\Users\Nikita\.jenkins\jobs\HelloWorld\workspace # timeout=10
Fetching upstream changes from https://github.com/Nikitakesharwani9>HelloWorld.git
> git.exe --version # timeout=10
> git.exe fetch --tags --force --progress https://github.com/Nikitakesharwani9>HelloWorld.git +refs/heads/*:refs/remotes/origin/*
> git.exe config remote.origin.url https://github.com/Nikitakesharwani9>HelloWorld.git # timeout=10
> git.exe config --add remote.origin.fetch +refs/heads/*:refs/remotes/origin/* # timeout=10
> git.exe config remote.origin.url https://github.com/Nikitakesharwani9>HelloWorld.git # timeout=10
Fetching upstream changes from https://github.com/Nikitakesharwani9>HelloWorld.git
> git.exe fetch --tags --force --progress https://github.com/Nikitakesharwani9>HelloWorld.git +refs/heads/*:refs/remotes/origin/*
> git.exe rev-parse "refs/remotes/origin/master^{commit}" # timeout=10
> git.exe rev-parse "refs/remotes/origin/origin/master^{commit}" # timeout=10
Checking out Revision 665f77769d297775e0521e28813edb1596d3250a (refs/remotes/origin/master)
> git.exe config core.sparsecheckout # timeout=10
> git.exe checkout -f 665f77769d297775e0521e28813edb1596d3250a
Commit message: "added HelloWorld program"
First time build. Skipping changelog.
[workspace] $ cmd /c call c:\apache-tomcat-9.0.21\temp\jenkins5711673716386440245.bat

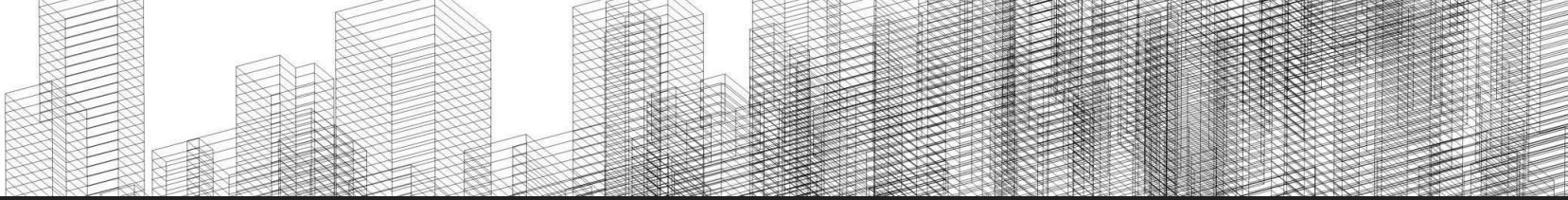
C:\Users\Nikita\.jenkins\jobs\HelloWorld\workspace>javac HelloWorld.java

C:\Users\Nikita\.jenkins\jobs\HelloWorld\workspace>java HelloWorld
Hello World

C:\Users\Nikita\.jenkins\jobs\HelloWorld\workspace>exit 0
Finished: SUCCESS
```

Chapter 10

GIT



Introduction to Version Control Systems

About Version Control

- Records changes to a file or set of files over time
- Recall specific versions later
- You can version control nearly any type of file on a computer
- It allows you to :
 - Revert selected files back to a previous state
 - Revert the entire project back to a previous state
 - Compare changes over time
 - See who last modified something that might be causing a problem
 - Who introduced an issue and when
- You get all this for very little overhead

Local Version Control Systems

- People's version-control method of choice is to copy files into another directory
- It is simple but error prone
- Accidentally one can copy wrong files or overwrite files
- To deal with it Programmer created VCS(Version Control System) long ago
- One of the more popular VCS tools was a system called RCS
- Developers can review project history to find out:
 - Which changes were made?
 - Who made the changes?
 - When were the changes made?
 - Why were changes needed?

A major issue with VCS that people encountered is that they need to collaborate with developers on other systems.

Centralized Version Control Systems

- Common examples of such CVCSs like CVS, Subversion and Perforce
- Single server contains all versioned files
- Number of clients can checkout files from central place
- Administrators have fine-grained control over who can do what

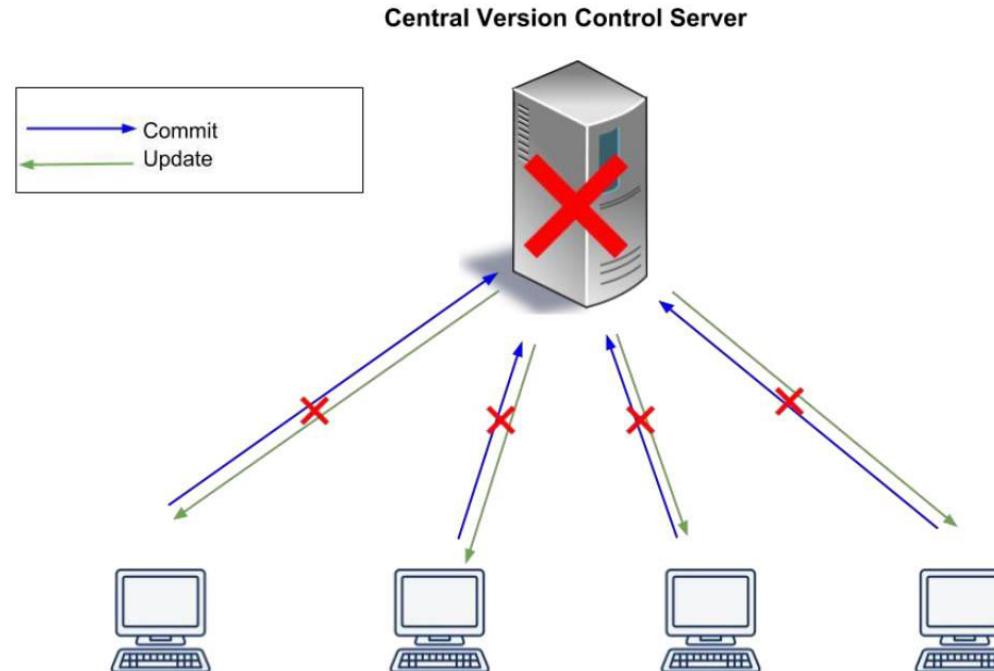
Has downside too !

Single point of Failure

If central server goes down,
no one can collaborate

Disk corruption on central
server may result into losing
everything

Local VCS systems suffer
from this same problem



Distributed Version Control Systems

- Allows clients to mirror full repository (including full history)
- In case of server failure files can be copied back up to the server
- Every clone is a full backup of data
- Collaborating becomes flexible and versatile by having several remote repositories

Examples: Mercurial, Git, Bazaar and Darcs

Short history of Git

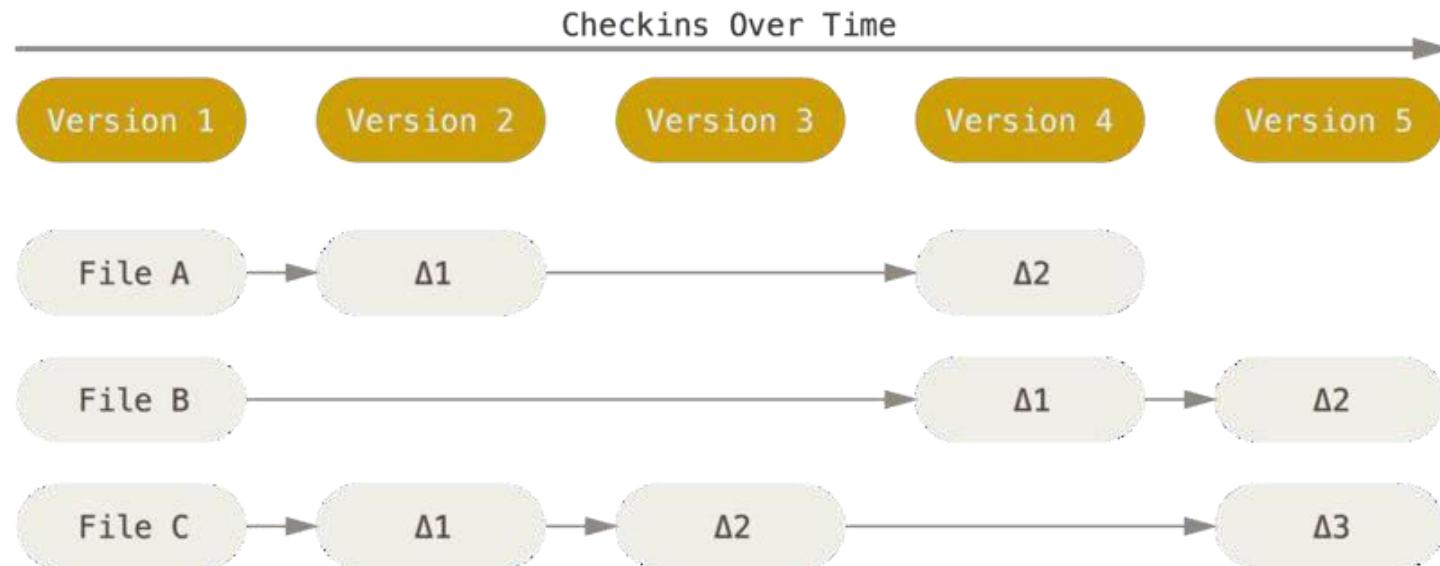
- Linux kernel is an open source software project
- Changes to this software were passed around as patches and archived files
- In 2002, they started using a proprietary DVCS called BitKeeper
- In 2005, company that developed Bitkeeper revoked the free-of-charge status
- So, Linus Torvalds and Linux community built their own tool

Goals of new tools :

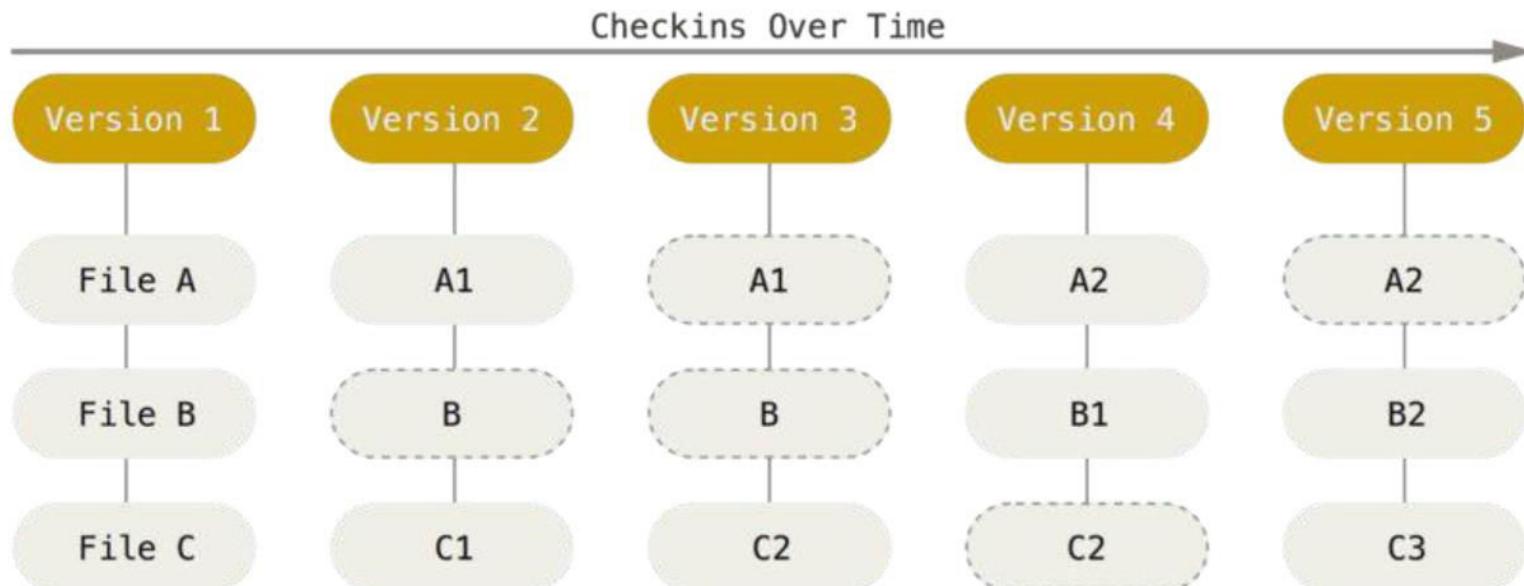
- Speed
- Simple design
- Strong support for non-linear development (thousands of parallel branches)
- Fully distributed
- Able to handle large projects like the Linux kernel efficiently (speed and data size)

Git Basics

- Snapshots, Not Differences
 - Other VCSs like Subversion etc., stores changes made to file over time
 - Often called as Delta VCS



- Git process data in a very different way
- Process data like series of snapshots
- Git basically takes a picture of what all your files look like at that moment and stores a reference to that snapshot
- Git doesn't store the file again, if no changes made to files



Things to remember

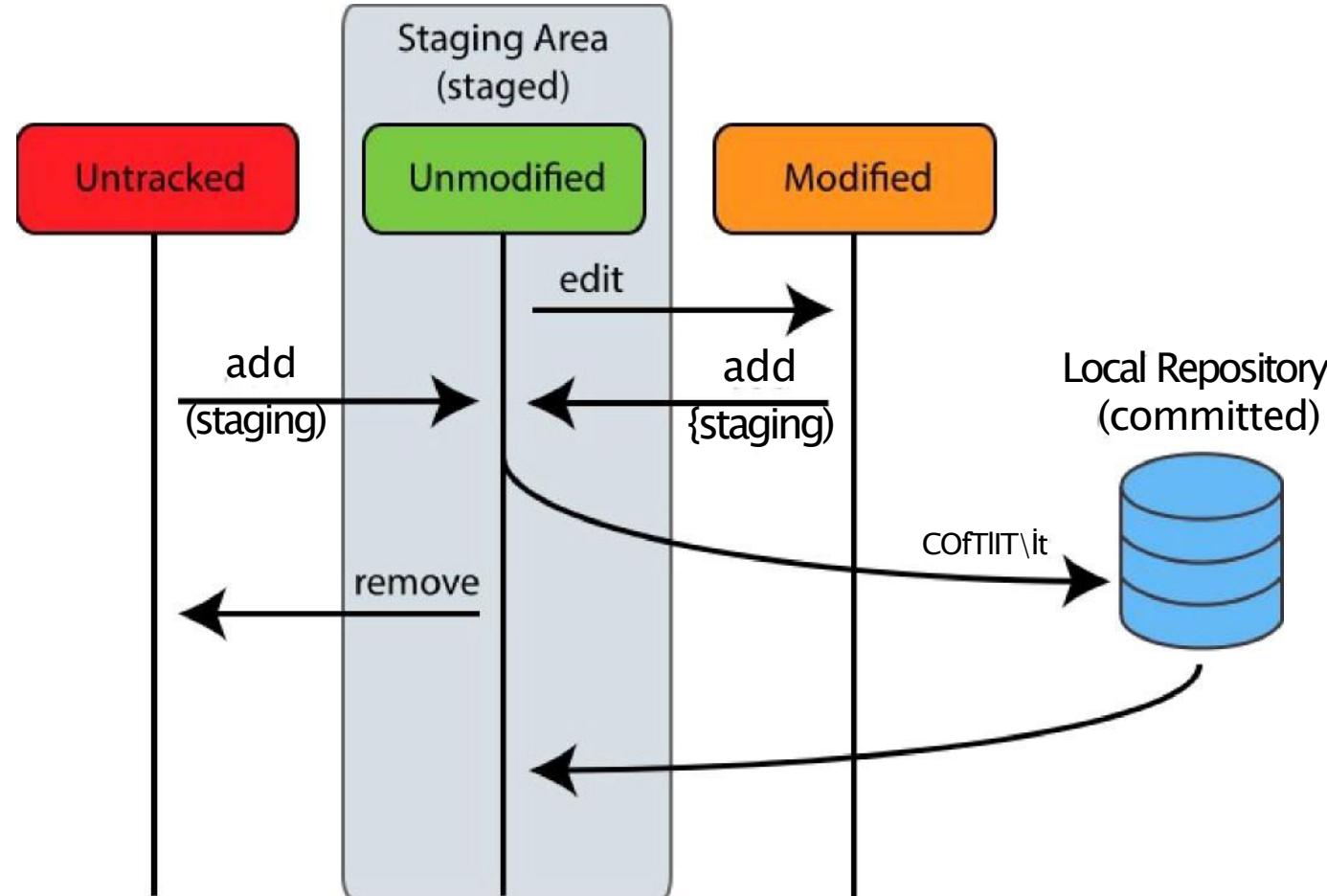
- Git requires only local files and resources to perform most operations
- Entire history of the project is stored locally, most operations seem almost instantaneous
- You can work on an Aeroplane as well
- Everything in Git is checksummed before it is stored
- Git uses SHA-1 mechanism for checksumming
 - Its 40 characters string
- Git stores everything in its database not by file name but by the hash value of its contents

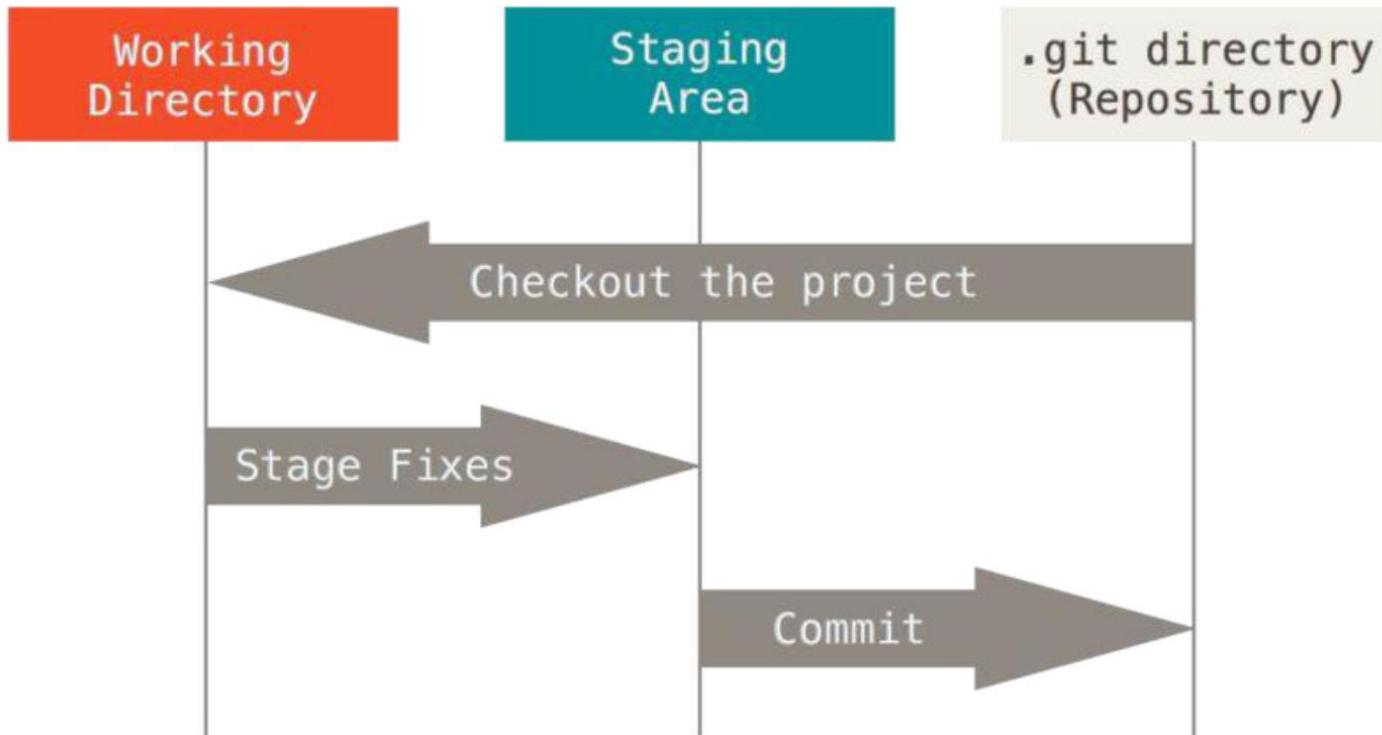
Three states in Git

“This is important to remember”

Git has three main states: **committed**, **modified**, and **staged**

- **Committed** means that the data is safely stored in your local database
- **Modified** means that you have changed the file but have not committed it to your database yet
- **Staged** means that you have marked a modified file in its current version to go into your next commit snapshot





**Let's explore the power
of Git !!**



What is Git?

- Git is a popular version control system. It was created by Linus Torvalds in 2005, and has been maintained by Junio Hamano since then.
- It is used for:
 - Tracking code changes
 - Tracking who made changes
 - Coding collaboration

What does Git do?

- Manage projects with **Repositories**
- **Clone** a project to work on a local copy
- Control and track changes with **Staging** and **Committing**
- **Branch** and **Merge** to allow for work on different parts and versions of a project
- **Pull** the latest version of the project to a local copy
- **Push** local updates to the main project

Working with Git

- Initialize Git on a folder, making it a **Repository**
- Git now creates a hidden folder to keep track of changes in that folder
- When a file is changed, added or deleted, it is considered **modified**
- You select the modified files you want to **Stage**
- The **Staged** files are **Committed**, which prompts Git to store a **permanent** snapshot of the files
- Git allows you to see the full history of every commit.
- You can revert back to any previous commit.
- Git does not store a separate copy of every file in every commit, but keeps track of changes made in each commit!

Why Git?

- Over 70% of developers use Git!
- Developers can work together from anywhere in the world.
- Developers can see the full history of the project.
- Developers can revert to earlier versions of a project.

Installing Git is fairly easy

- Git can be downloaded from <https://git-scm.com/download/>
- Git can run on Linux, Windows, Mac
- You can also look for Git-for-Windows project
 - This is separate from than Git
- Automated installation on windows can be done by community maintained “Git Chocolatey”



git

--local-branching-on-the-cheap



Search entire site...

About

Documentation

Downloads

GUI Clients

Logos

Community

The entire [Pro Git book](#) written by Scott Chacon and Ben Straub is available to [read online for free](#). Dead tree versions are available on [Amazon.com](#).

Downloads



Mac OS X



Windows



Linux/Unix

Older releases are available and the [Git source repository](#) is on GitHub.

GUI Clients

Git comes with built-in GUI tools (`git-gui`, `gitk`), but there are several third-party tools for users looking for a platform-specific experience.

[View GUI Clients →](#)

Latest source Release

2.20.1

[Release Notes \(2018-12-15\)](#)

[Downloads for Linux](#)



Logos

Various Git logos in PNG (bitmap) and EPS (vector) formats are available for use in online and print projects.

[View Logos →](#)

```
[root@workstation ~]# uname -r
3.10.0-693.el7.x86_64
[root@workstation ~]# cat /etc/centos-release
CentOS Linux release 7.4.1708 (Core)
[root@workstation ~]#
[root@workstation ~]# yum install -y git
git19-build.x86_64          git19-scl-devel.x86_64          gitg.x86_64
git19-emacs-git-el.noarch   git19.x86_64                  git-hg.noarch
git19-emacs-git.noarch      git2cl.noarch                github2fedmsg.noarch
git19-git-all.noarch        git-all.noarch                git-instaweb.noarch
git19-git-bzr.noarch        git-annex.x86_64            gitk.noarch
git19-git-cvs.noarch        git-bzr.noarch                git-lfs.x86_64
git19-git-daemon.x86_64     git-cal.noarch               git-merge-changelog.x86_64
git19-git-email.noarch      git-cola.noarch              gitolite3.noarch
git19-git-gui.noarch        git-cvs.noarch               git-p4.noarch
git19-git-hg.noarch         git-daemon.x86_64            git-publish.noarch
git19-gitk.noarch           git-email.noarch              git-review.noarch
git19-git-svn.x86_64         git-extras.noarch             gitstats.noarch
git19-gitweb.noarch         gitflow.noarch               git-svn.x86_64
git19-git.x86_64             gitg-devel.x86_64              git-tools.noarch
git19-perl-Git.noarch       gitg-libs.x86_64              gitweb.noarch
git19-perl-Git-SVN.noarch   git-gnome-keyring.x86_64        git.x86_64
git19-runtime.x86_64          git-gui.noarch              git-xcleaner.noarch
```

```
[root@workstation ~]# git version
git version 1.8.3.1
[root@workstation ~]# █
```

First-Time Git setup

- **git config** - this tool allows you to set configuration variables
- Using this you can configure how git looks and operates
- **git config --system**
 - Reads from /etc/gitconfig file
- **git config --global**
 - Reads from ~/.gitconfig or ~/.config/git/config file
- **git config --local**
 - Reads from .git/config
- Remember each level is and override

Git Identity

```
git config --global user.name "codeburster"
```

```
git config --global user.email "codeburster@gmail.com"
```

Default Editor for Git

```
git config --global core.editor emacs
```

On Windows

```
git config --global core.editor " 'C:/Program Files/Notepad++/notepad++.exe'  
-multiInst -nosession"
```

Commands to check Git config

git config --list

git config user.name

cat ~/.gitconfig or ~/.config/git/config

```
[root@workstation ~]# git config --global user.name "codeburster"
[root@workstation ~]# git config --global user.email "codeburster@gmail.com"
[root@workstation ~]# cat ~/.gitconfig
[user]
    name = codeburster
    email = codeburster@gmail.com
[root@workstation ~]#
[root@workstation ~]# git config --global core.editor vim
[root@workstation ~]# git config --list
user.name=codeburster
user.email=codeburster@gmail.com
core.editor=vim
```

What is a Repository ?

- Entire collection of files and folders associated with a project, along with each file's revision history
- a git repository also allows for:
 - interaction with the history
 - Cloning
 - Creating branches
 - Committing
 - Merging
 - Comparing changes across versions of code
 - and more

Two ways to get a Repository

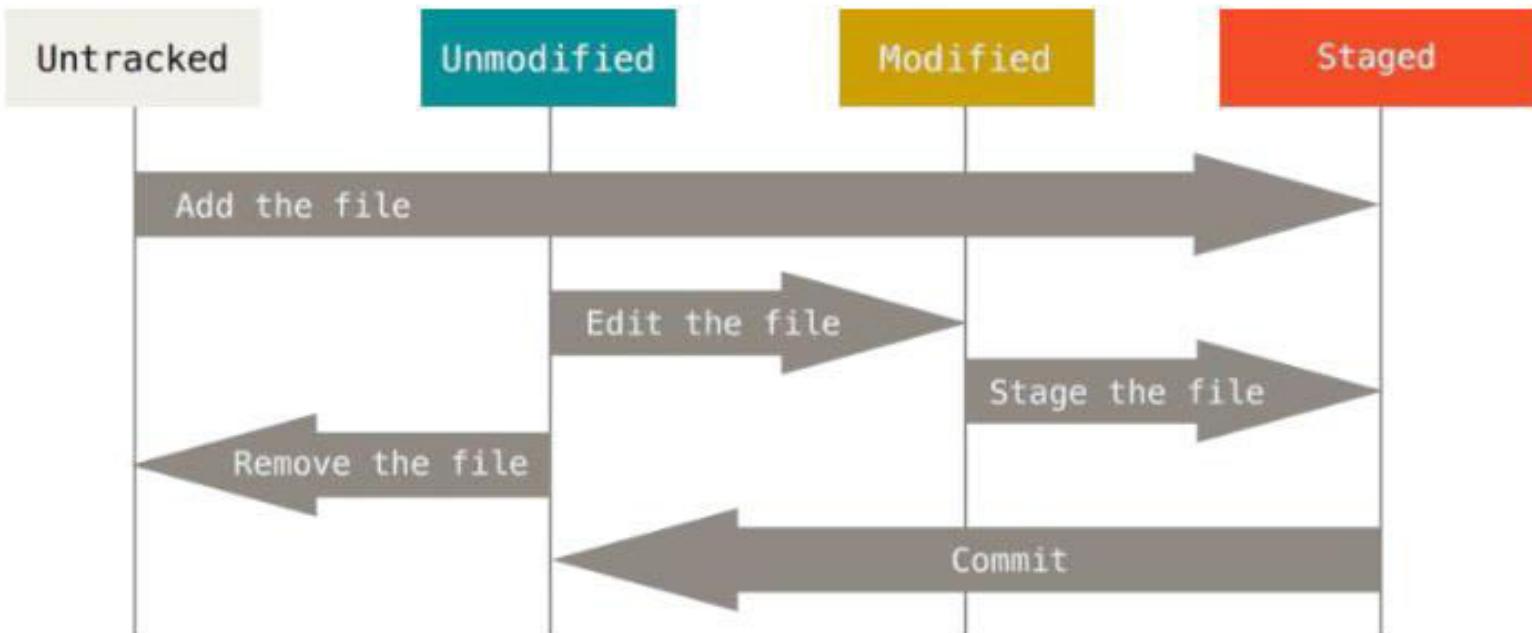
1. You can take a local directory that is currently not under version control, and turn it into a Git repository
2. You can clone an existing Git repository from elsewhere

Basic Git commands

1. **git init** - initializes a brand new Git repository and begins tracking an existing directory
2. **git clone** - creates a local copy of a project that already exists remotely
3. **git add** - stages a change. Git tracks changes to a developer's codebase, but it's necessary to stage and take a snapshot of the changes to include them in the project's history
4. **git commit** - saves the snapshot to the project history and completes the change-tracking process
5. **git status** - shows the status of changes as untracked, modified, or staged

Status of your files

1. file in your working directory can be in one of two states
 - a. Tracked
 - b. Untracked
2. Tracked files are files that Git knows about

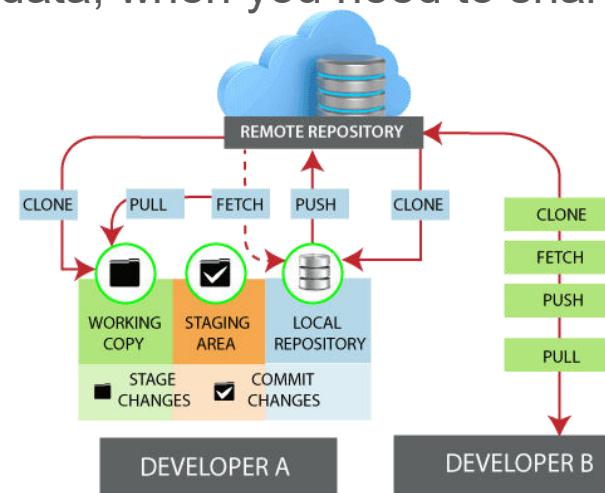


Some more git commands

1. **git status -s** → Short Status
2. **.gitignore** → specify file names that you want to ignore during commit
3. **git diff** → to view staged and unstaged changes
4. **git rm** → to remove files from repository
5. **git mv** → to move files within git repository structure or for renaming
6. **git log** → to view commit history (–stat , -p/–patch options can be used)
7. **git checkout -- <file_name>** → to unmodify the modified file

Working with Remotes

- Remote repositories are versions of your project that are hosted on the Internet or network somewhere
- Can be read-only or read/write
- Allows pushing and pulling data, when you need to share work



Showing Your Remotes

- `git remote` → To see which remote servers you have configured
- `origin` → is the default name Git gives to the server you cloned from
- `git remote -v` → shows URLs
- `git remote add cb https://github.com/gauravkumar9130/testrepo`
- `git fetch cb`
- `git push <remote> <branch>`
- `git remote show origin`
- `git remote rename <remote_name> <remote_new_name>`
- `git remote remove <remote_name>`

Tagging

- Git has the ability to tag specific points in history
- Can be used to mark release points (v1.0 and so on)
- `git tag` → to list available tags
- `git tag -l "v1.2.5"` → to list tags that match this pattern

Types of tags

- Lightweight
 - Much like a branch
 - Just a pointer to a specific commit
- Annotated
 - Stored in Git database as full objects
 - Checksummed
 - Contains tagger name, email, and date
 - Can be signed and verified with GPG

How do we do it ?

- Git tag -a v1.4 -m “version 1.4 stable”
 - -m → to specify tagging message
 - -a → to create and annotated tag

Tagging commits you forgot

```
[root@workstation demo]# git log --pretty=oneline
01cedb3ef37586d7f05c21a932814bda91788324 Third Commit
bbf8ealea47b402f33d7ab66b49dea3b7cdf378c second commit
018c07b757f6175fce169d7382a62f2ae44f5fc8 Initial Commit
```

Sharing Tags

- By default “git push” doesn’t transfer tags to remote servers
- `git push <remote> <tagname>`
 - Example: `git push origin v1.6`
- `git push --tags` → to push all tags at once
- `git tag -d v1.6` → to delete a tag
- `git push origin :refs/tags/v1.6` → to delete tag from remote as well
- `git checkout v1.4` → to view versions of file tag is pointing to

Git branching

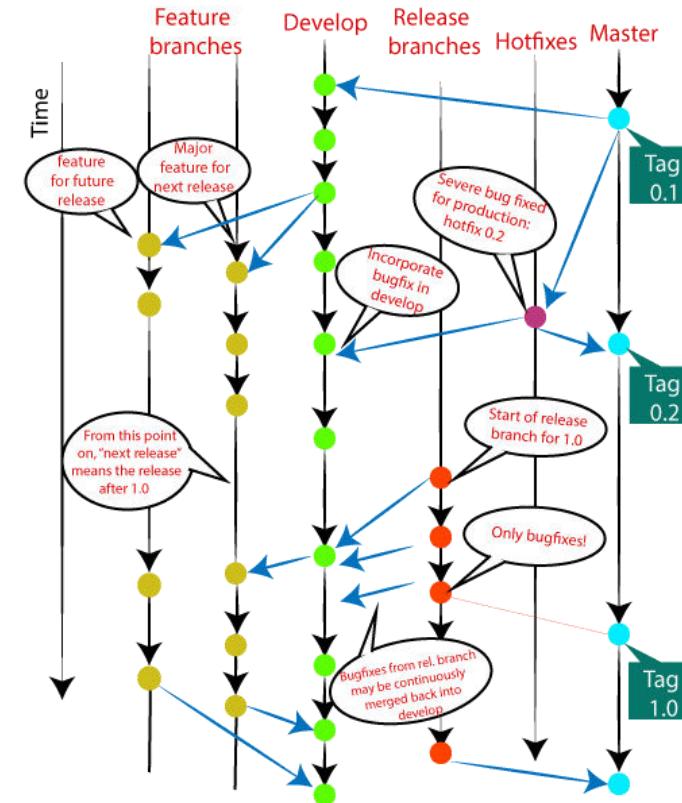
- Branching allows developer to diverge from main line of development
- This allows to continue to work without messing with main line
- Git branches is lightweight
- Almost instantaneous
- Switch back and forth between branches quickly

Git Flow/ Branching Model

- Git flow is the set of guidelines that developers can follow when using Git. We cannot say these guidelines as rules.
- These are not the rules; it is a standard for an ideal project. So that a developer would easily understand the things.
- It is referred to as **Branching Model** by the developers and works as a central repository for a project. Developers work and push their work to different branches of the main repository.

Git Flow/ Branching Model

- There are different types of branches in a project. According to the standard branching strategy and release management, there can be following types of branches:
 - Master
 - Develop
 - Hotfixes
 - Release branches
 - Feature branches



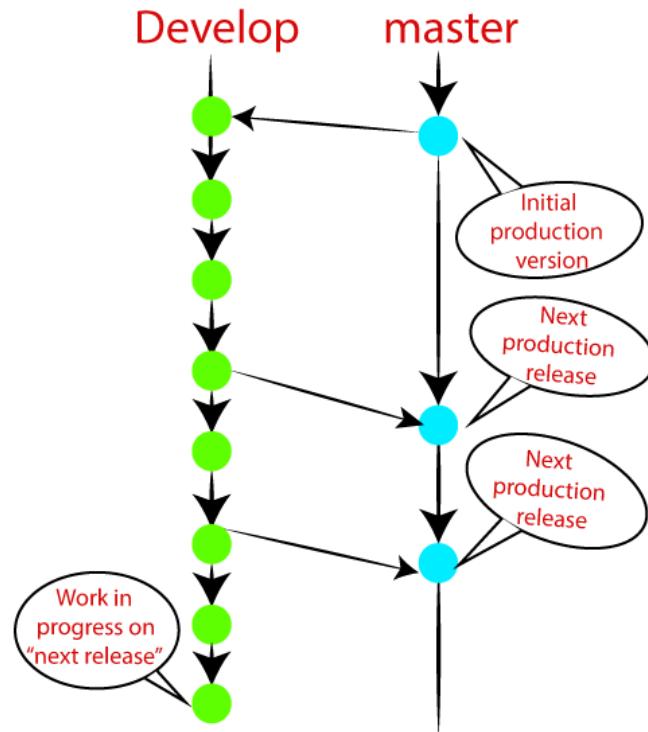
Git Branching

- **Master Branch**

- The master branch is the main branch of the project that contains all the history of final changes. Every developer must be used to the master branch. The master branch contains the source code of HEAD that always reflects a final version of the project.

- **Develop Branch**

- It is parallel to the master branch. It is also considered as the main branch of the project. This branch contains the latest delivered development changes for the next release. It has the final source code for the release. It is also called as a "integration branch."



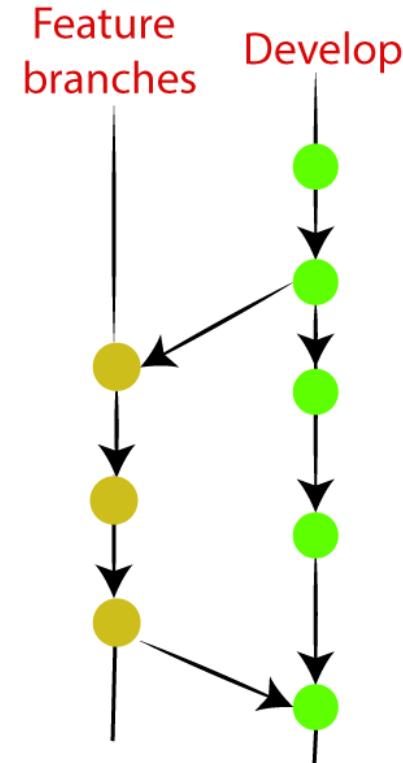
Git Branching

- **Feature Branches**

- Feature branches can be considered as topic branches. It is used to develop a new feature for the next version of the project. The existence of this branch is limited; it is deleted after its feature has been merged with develop branch.

- **Release Branches**

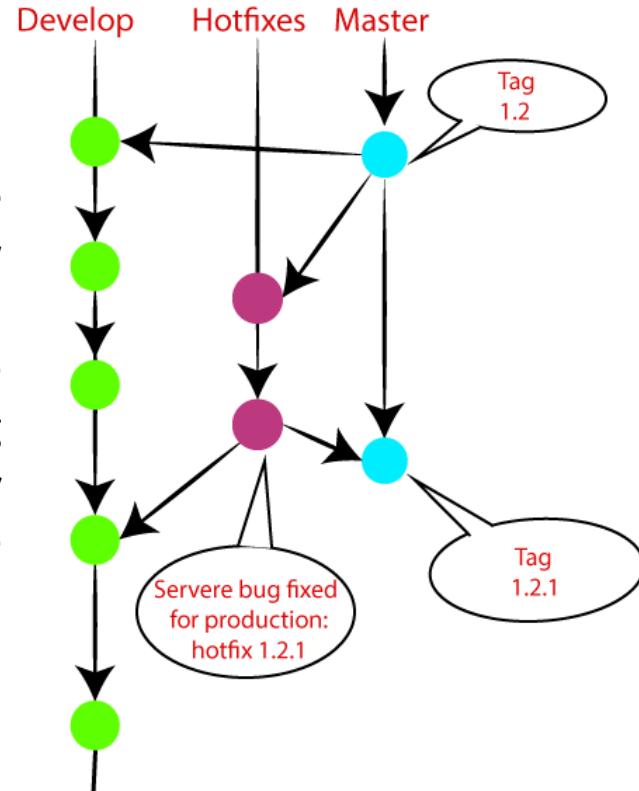
- The release branch is created for the support of a new version release. Senior developers will create a release branch. The release branch will contain the predetermined amount of the feature branch. The release branch should be deployed to a staging server for testing.



Git Branching

- Hotfix Branch

- Hotfix branches are similar to Release branches; both are created for a new production release.
- The hotfix branches arise due to immediate action on the project. In case of a critical bug in a production version, a hotfix branch may branch off in your project. After fixing the bug, this branch can be merged with the master branch with a tag.



Keep Learning more at → <https://git-scm.com/docs>



TFS Vs Git

Now, let's summarize the key differences between TFS and Git.

Factor	TFS	Git
Type	Centralized	Distributed
Branching and Merging	Uses a branch-based approach	Uses a branch-based approach
Workflow	Follows a check-in/check-out workflow	Follows a commit-based workflow
Performance	Sometimes have slower performance	Excellent performance
Integration	Tightly integrated with other Microsoft development tools and services	Has widespread adoption and a vast ecosystem of tools and integrations, making it compatible with various IDEs, build systems, and hosting platforms.
Offline Work	Limiting offline work capabilities.	Work offline and commit changes locally without requiring a network connection.

Chapter 11

Monitoring Patterns

Objectives

- **Chapter 11: Monitoring Patterns**
 - Learn about the monitoring and observability patterns of microservices
 - Understand what is Prometheus and Grafana.
 - Learn how to work with Prometheus and Grafana for monitoring and alerting

Observability patterns

- Observability in microservices is crucial as it provides development teams with necessary data to identify problems and detect failures.
- This data is collected using various observability patterns:
 - Log Aggregation
 - Performance metrics
 - Distributed Tracing
 - Health Check

Log Aggregation

- In applications comprising multiple services, requests may span across several service instances.
- Each service instance generates a standardized log file.
- A centralized logging service aggregates these logs, allowing users to search and analyze log data.
- Alerts can be set up based on specific log messages to notify users of particular events.
- For example, the Pivotal Cloud Foundry (PCF) platform includes a log aggregator that collects logs from all its components as well as from applications.

Performance Metrics

- As a microservice architecture expands, monitoring and alerting become crucial for observing transaction patterns and detecting issues in a timely manner. A metrics service collects statistics on individual operations, and these metrics should be aggregated for reporting and alerting. There are two metric aggregation models:
 - Push - the service sends metrics to a metrics service like NewRelic or AppDynamics.
 - Pull - the metrics service retrieves metrics from a service, like Prometheus.

Distributed Tracing

- In microservices architecture, requests often span across multiple services.
- For effective troubleshooting, it's useful to have end-to-end request tracing.
- This can be achieved by generating a unique identifier for each incoming request (called a trace ID or transaction ID), which is then propagated across all services and incorporated into all log messages.

Health Check

- In a microservice architecture, a service may be operational but unable to process transactions.
- Each service should therefore have a health endpoint (like /health) to verify its status.
- This health check API should assess the status of the host, connectivity to other services/infrastructure, and specific logic.

Application Performance Monitoring

- The monolith has been decomposed now into its component parts, refactored those parts as containerized microservices, and used an API manager and API gateway for handle the API interactions between the microservices that comprise the system.
- We have also used a container orchestration tool to manage deployment/scaling/resource distribution for this large cluster of services.
- However, we need another piece of technology – called “application performance monitoring (APM)” – to complete this multi-faceted puzzle.

Application Performance Monitoring

- APM helps you monitor and fine-tune the performance of the system you have built. It provides developers with the data they need to ensure:
 - The highest standards of application performance for the end-user;
 - Efficient utilization of system resources; and
 - A high availability system with minimum chances of downtime.

Types of APM

- Generally, it can be divided into two separate types:
 - (1) Application front-end performance monitoring,
 - (2) Application back-end performance monitoring, i.e., infrastructure monitoring.

Application Front End Performance Monitoring

- Application front-end performance monitoring relates to tracking the metrics and data points related to the user experience and what the user sees.
- This includes information on service availability, memory utilization, and overall app performance.
- Developers use front-end APM information to maximize app performance, improve the overall user experience, and boost user retention numbers.

Application Front End Performance Monitoring

- What it tracks?
 - Errors in web apps and websites, how often they occur, their effects on performance, and the circumstances under which they occur (i.e., under which user actions and use conditions).
 - HTTP requests/responses and any associated network request failures related to external services.
 - Web framework issues related to the functionality of libraries such as Springboot, React and Angular.
 - Activity history, click history, and page view history. This includes aggregated data, data for specific categories of users, and data for individual users.

Application Backend Performance Monitoring

- Application back-end performance monitoring (infrastructure monitoring) relates to tracking information pertaining to the infrastructure and server resources that support the app or architecture.
- This gives you key metrics and data related to an application's vital signs – allowing you to optimize performance and more efficiently use the resources that support the system.

Application Backend Performance Monitoring

- What to tracks?
 - App infrastructure metrics relating to the server, network, storage, etc.
 - The tools/systems that run the app such as Docker, Kubernetes, Zuul, Github, etc.
 - Key Performance Indicators (KPI) related to the efficiency of the app, how profitable it is to run, and which parts of the application generate the most income.
 - The APM tools themselves and their current performance and status.

APM Tools

- **Prometheus and Graphite:** These are two of the most popular APM solutions, which you can configure for both back-end and front-end monitoring.
- Prometheus and Graphite are free and open-source, but setting them up requires advanced skills and takes time (all of which cost money).
- Also, these solutions need to be paired with an open-source dashboarding tool like Grafana or Kibana for viewing metrics.
- The benefits of using Prometheus and Graphite are the flexible configuration options and no vendor lock-in.

APM Tools

- **DataDog and SolarWinds:** DataDog and SolarWinds are well-known, proprietary APM solutions for back-end and front-end monitoring.
- These fully-hosted solutions are fast and easy to set up, and they have beautiful dash-boarding options.
- However, they are priced for large enterprises and usually too expensive for small-to-medium-sized firms. Plus, these proprietary solutions subject you to vendor lock-in.

APM Tools

- **MetricFire:** MetricFire is more geared toward back-end APM.
- As a fully-hosted platform that offers Prometheus and Graphite “as-a-service,” MetricFire presents Prometheus or Graphite as if they were easy-to-set-up, proprietary monitoring solutions (without the vendor lock-in and lack of flexibility of DataDog or SolarWinds).
- While MetricFire offers front-end monitoring too, it really excels in the back-end APM space – especially when tracking time-series metrics for various systems that support an application.
- MetricFire also uses Grafana dashboards for beautiful metrics displays.

APM Tools

- **Scout APM** is more geared toward application front-end performance monitoring.
- Scout APM uses tracing logic to match performance bottlenecks with the source code that causes them – making it easier to pinpoint and resolve problems, so you can provide a better user experience and boost user retention.



Prometheus

- Prometheus is an open-source systems monitoring and alerting toolkit originally built at SoundCloud.
- Since its inception in 2012, many companies and organizations have adopted Prometheus, and the project has a very active developer and user community.
- It is now a standalone open source project and maintained independently of any company.
- To emphasize this, and to clarify the project's governance structure, Prometheus joined the Cloud Native Computing Foundation in 2016 as the second hosted project, after Kubernetes.



Prometheus

- Prometheus collects and stores its metrics as time series data, i.e. metrics information is stored with the timestamp at which it was recorded, alongside optional key-value pairs called labels.
- Prometheus's main features are:
 - A multi-dimensional data model with time series data identified by metric name and key/value pairs
 - PromQL, a flexible query language to leverage this dimensionality
 - No reliance on distributed storage; single server nodes are autonomous
 - Time series collection happens via a pull model over HTTP
 - Pushing time series is supported via an intermediary gateway
 - Targets are discovered via service discovery or static configuration
 - Multiple modes of graphing and dash-boarding support



Prometheus

- Metrics are numerical measurements in layperson terms. The term time series refers to the recording of changes over time.
- What users want to measure differs from application to application. For a web server, it could be request times; for a database, it could be the number of active connections or active queries, and so on.
- Metrics play an important role in understanding why your application is working in a certain way.
- Let's assume you are running a web application and discover that it is slow. To learn what is happening with your application, you will need some information. For example, when the number of requests is high, the application may become slow

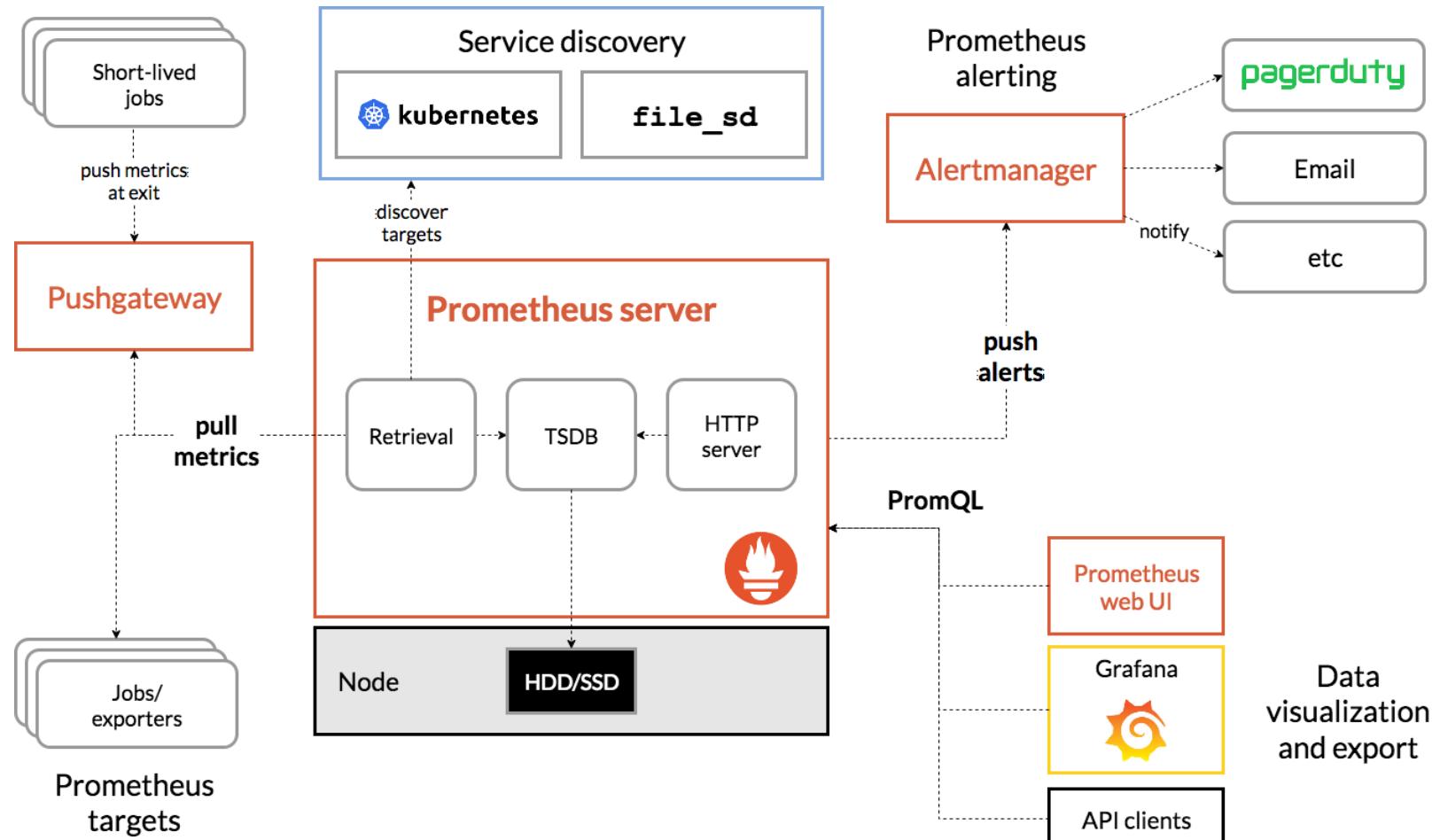


Prometheus

- The Prometheus ecosystem consists of multiple components, many of which are optional:
 - The main Prometheus server which scrapes and stores time series data
 - Client libraries for instrumenting application code
 - A push gateway for supporting short-lived jobs
 - Special-purpose exporters for services like HAProxy, StatsD, Graphite, etc.
 - An alertmanager to handle alerts
 - Various support tools



Prometheus Architecture





Prometheus Architecture

- Prometheus scrapes metrics from instrumented jobs, either directly or via an intermediary push gateway for short-lived jobs.
- It stores all scraped samples locally and runs rules over this data to either aggregate and record new time series from existing data or generate alerts.
- Grafana or other API consumers can be used to visualize the collected data.



Prometheus

- **When does it fit?**
- Prometheus works well for recording any purely numeric time series.
- It fits both machine-centric monitoring as well as monitoring of highly dynamic service-oriented architectures.
- In a world of microservices, its support for multi-dimensional data collection and querying is a particular strength.
- Prometheus is designed for reliability, to be the system you go to during an outage to allow you to quickly diagnose problems.
- Each Prometheus server is standalone, not depending on network storage or other remote services.



Prometheus

- **When does it not fit?**
- Prometheus values reliability. It can always view what statistics are available about your system, even under failure conditions.
- If you need 100% accuracy, such as for per-request billing, Prometheus is not a good choice as the collected data will likely not be detailed and complete enough.
- In such a case you would be best off using some other system to collect and analyze the data for billing, and Prometheus for the rest of your monitoring.

Grafana

- Grafana supports querying Prometheus. The Grafana data source for Prometheus is included since Grafana 2.5.0 (2015-10-28).
- Query, visualize, alert on, and understand your data no matter where it's stored.
- With Grafana you can create, explore, and share all of your data through beautiful, flexible dashboards.

Grafana – Why?

- **Unify your data, not your database**
 - Grafana doesn't require you to ingest data to a backend store or vendor database.
- **Data everyone can see**
 - Grafana was built on the principle that data should be accessible to everyone in your organization, not just the single Ops person.
- **Dashboards that anyone can use**
 - Not only do Grafana dashboards give insightful meaning to data collected from numerous sources, but you can also share the dashboards you create with other team members, allowing you to explore the data together.
- **Flexibility and versatility**
 - Translate and transform any of your data into flexible and versatile dashboards. Unlike other tools, Grafana allows you to build dashboards specifically for you and your team.

Grafana

- You can download Grafana from official website.
 - <https://grafana.com/grafana/download>

Version: 10.3.3

Edition: Enterprise ▾

The Enterprise Edition is the default and recommended edition. It includes all the features of the OSS Edition, can be used for free and can be upgraded to the full Enterprise feature set, including support for Enterprise plugins.

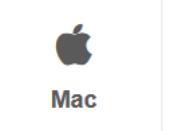
License: Grafana Labs License

Release Date: February 14, 2024

Release Info: [What's New In Grafana 10.3.3](#)

 Linux

 Windows

 Mac

 Docker

 Linux on ARM64

Grafana

- By default, Grafana will be listening on <http://localhost:3000>.
- The default login is "admin" / "admin"



Grafana- Creating a Prometheus data source

To create a Prometheus data source in Grafana:

1. Click on the "cogwheel" in the sidebar to open the Configuration menu.
2. Click on "Data Sources".
3. Click on "Add data source".
4. Select "Prometheus" as the type.
5. Set the appropriate Prometheus server URL (for example, <http://localhost:9090/>)
6. Adjust other data source settings as desired (for example, choosing the right Access method).
7. Click "Save & Test" to save the new data source.

