Every company is a technology company,

regardless of what business they think they

are in. A bank is just and IT company with a

banking license.

*Cristopher Little*

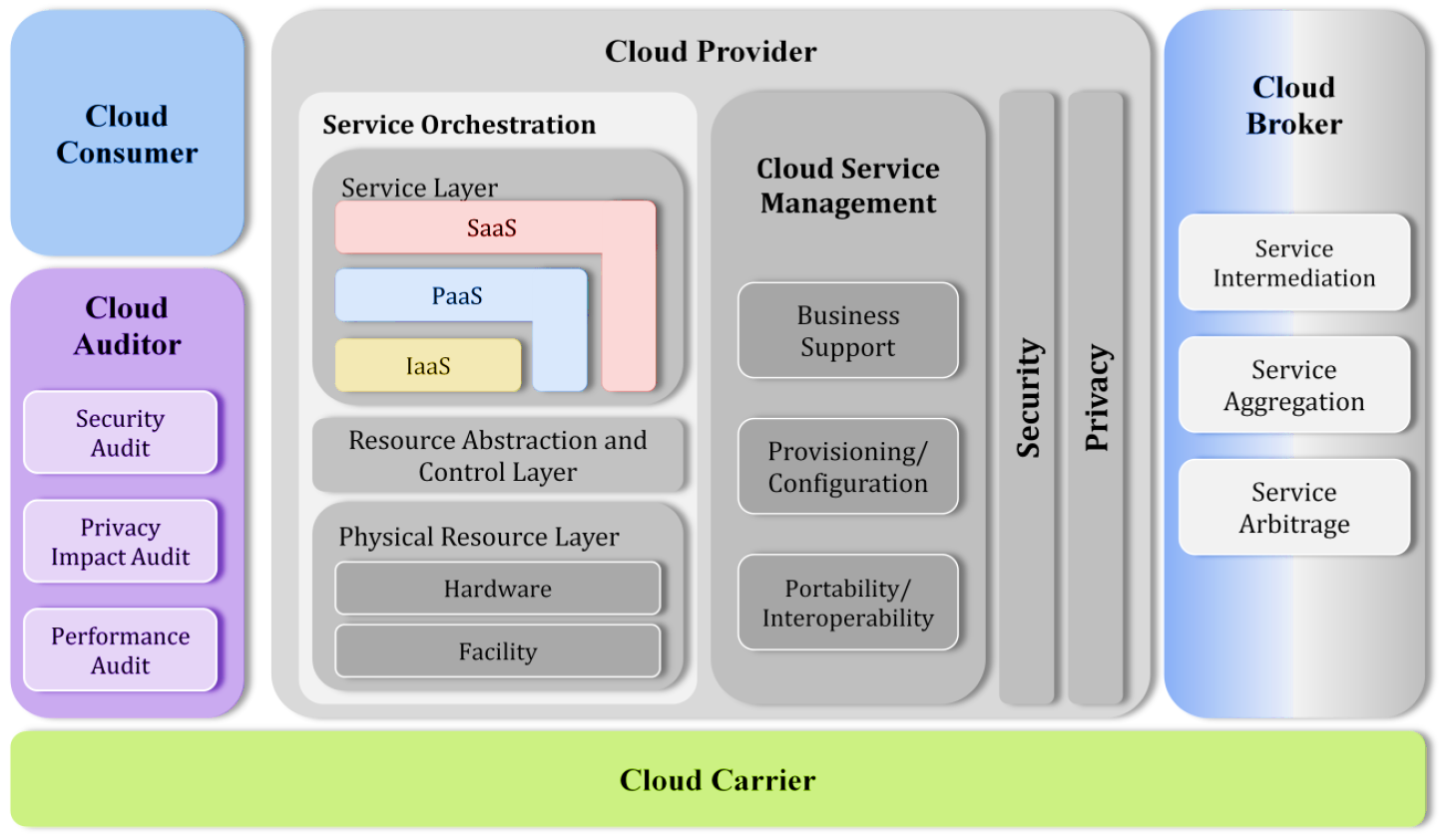
# Cloud Computing

Cloud Computing enables companies to consume a compute resource, such as a virtual machine (VM), storage or an application, as a utility -- just like electricity -- rather than having to build and maintain computing infrastructures in house.

The main Cloud Computing characteristics are:

* On-Demand Self Services;
* Broad network access;
* Resource Pooling;
* Rapid Elasticity;
* Measured Services.

The main architecture of a cloud computing platform is described in NIST Special Publication 500-292 published in 2011:

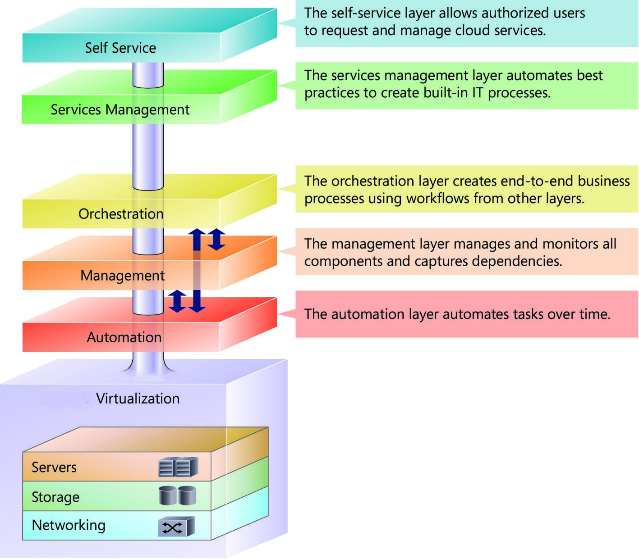


The Cloud Computing Reference Architecture defines the following main Architectural Components:

1. **Service Orchestration** refers to the composition of system components to support the Cloud Providers activities in arrangement, coordination and management of computing resources in order to provide cloud services to Cloud Consumers.
2. **Resource abstraction and control layer** contains the system components that Cloud Providers use to provide and manage access to the physical computing resources through software abstraction.
3. **Cloud Service Management** includes all the service-related functions that are necessary for the management and operation of those services required by or proposed to cloud consumers.
4. **Business Support** entails the set of business-related services dealing with clients and supporting processes.
5. **Cloud Security** is crosscutting aspect of the architecture that spans across all layers of the reference model, ranging from physical security to application security

The following main actors is defined by Cloud Computing Reference Architecture:

1. **Cloud Consumer** – is a person or organization that maintains a business relationship with, and uses service from Cloud providers;
2. **Cloud Provider** – is a person, organization, or entity responsible for making a service available to interested parties;
3. **Cloud Auditor** – is a party that can conduct independent assessment of cloud services, information system operations, performance and security of the cloud implementation;
4. **Cloud Manager** – is a party that manages the delivery of services and infrastructure;
5. **Cloud Broker** – is an entity that manages the use, performance and delivery of cloud services, and negotiates relationships between Cloud Providers and Cloud Consumers
6. **Cloud Carrier** – is an intermediary that provides connectivity and transport of cloud services from Cloud Providers to Cloud Consumers.

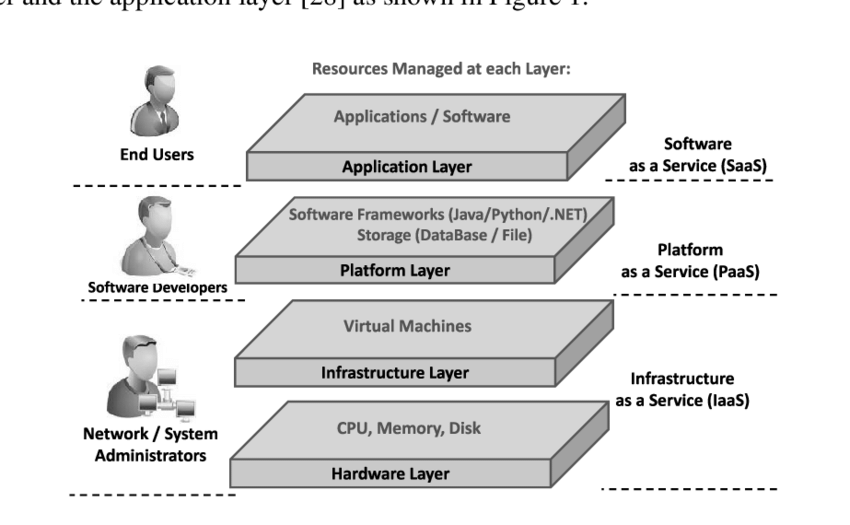
From implementation point of view a Cloud Computing Infrastructure could be splitted in three main components:

1. Hardware Infrastructure;
2. Virtualization Infrastructure;
3. Cloud Management Infrastructure.

## Cloud Services

The following are three main services which usually are provided by a Cloud Platform:

**Infrastructure as a Service (IaaS)** - The capability provided to the consumer is to provision processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. *The consumer does not manage or control the underlying cloud infrastructure but has control over operating systems, storage, deployed applications, and possibly limited control of select networking components (e.g., host firewalls)*.

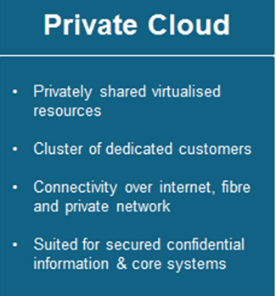


**Platform as a Service (PaaS)** - The capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications created using programming languages and tools supported by the provider. *The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, or storage, but has control over the deployed applications and possibly application hosting environment configurations*.

**Software as a Service (SaaS)** - The capability provided to the consumer is to use the providers applications running on a cloud infrastructure. The applications are accessible from various client

devices through a thin client interface such as a web browser (e.g., web-based email). *The consumer does not manage or control the underlying cloud infrastructure including network, servers, operating systems, storage, or even individual application capabilities, with the possible exception of limited user-specific application configuration settings.*

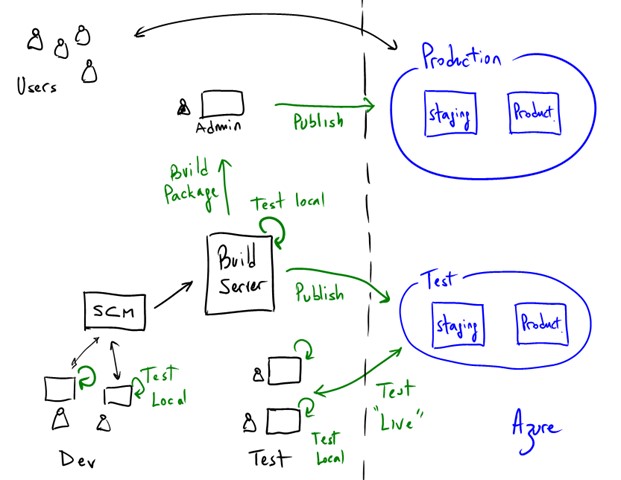
## Cloud Deployment Models

There are 4 Deployment Models, from which first 3 are the most popular:

1. **Private Cloud** – used by a single organization with exclusive access to and usage of the infrastructure and computational resources. It may be managed either by the organization or by a third party, and may be hosted on the organization’s premises or outsourced to a hosting company.
2. **Public Cloud -** the cloud infrastructure and computing resources are made available to the general public over a public network. A public cloud is owned by an organization selling cloud services, and serves to the general public.
3. **Hybrid Cloud** - is a composition of two or more clouds (usually private and public) that remain as distinct entities but are connected together by standardized or proprietary technology that offer the benefits of multiple application/data deployment model.

### Hybrid Cloud usage use case

**Use Case**. Production environment and Test environment are deployed in Public Cloud, but DevOps platform on premises.



**Question 1**: What Cloud Services do you think will be used in Public Cloud (IaaS, PaaS, SaaS) to deploy production and test infrastructure?

**Question 2**: Could you define other hybrid cloud use cases in development and deployment of application? What Public and Private Cloud Service (IaaS, PaaS, SaaS) is used in each use case? Please draw a simple diagram which represent the use cases.

## Further Reading

1. [NIST Special Publication 500-292: Cloud Computing Reference Architecture](https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication500-292.pdf)

# Cloud Native Applications: 12 Factors methodology

Typically, cloud-native applications are developed as loosely coupled. These applications anticipate failure, and they run and scale reliably even when their underlying infrastructure is experiencing outages. To offer such capabilities, cloud-native platforms impose a set of contracts and constraints on the applications running on them. These contracts ensure that the applications conform to certain constraints and allow the platforms to automate the management of the containerized applications.

As result the design of modern, cloud-native apps requires a change in how we think about software engineering, configuration, and deployment, when compared to designing apps for non-cloud environments. This section provide an introduction in 12 factor methodology which help to build a cloud native application.

12 Factors methodology is a set of principals, described below, regarding how to design and build a Cloud Native Application

## Codebase - one application <=> one codebase

*One codebase tracked in revision control, many deploys.*

The app code should be tracked in a version control system. Storing the code in a version control system enables the project team to work together by providing an audit trail of changes to the code, a systematic way of resolving merge conflicts, and the ability to roll back the code to a previous version. It also provides a place from which to do continuous integration (CI) and continuous deployment (CD).

The code in the repository is used to produce a single build, which is combined with environment-specific configuration to produce an immutable release (a release where no changes can be made, including to the configuration) that can then be deployed to a specific environments:

* development
* staging
* production

## Dependencies

*Application's dependencies must be declared and isolated*

Depending of the framework, the application must declare all dependencies, completely and exactly, via a dependency declaration manifest. Example: Spring Boot application Maven build file, pom.xml and isolation with building one fat jar file: mvn clean install.

The application and its dependencies should be isolate by packaging them into a container. Containers allow isolating an app and its dependencies from its environment and ensuring that the application works uniformly despite any differences between development and staging environments.

The Container Registry is a single place for the project team to manage container images and perform vulnerability analysis. It also lets you decide who can access what, using fine-grained access control to the container images

## Configuration

*Configuration should be stored in the environment variables.*

These configurations usually include service account credentials and resource handles to backing services such as databases.

The configurations for each environment should be external to the code and should not be checked into version control. The deployment environment determines which configuration to use. This enables one version of the binary to be deployed to each environment, where the only difference is the runtime configuration. An easy way to check whether the configuration has been externalized correctly is to see if the code can be made public without revealing any credentials.

One way of externalizing configuration is to create configuration files. However, configuration files are usually specific to a language or development framework.

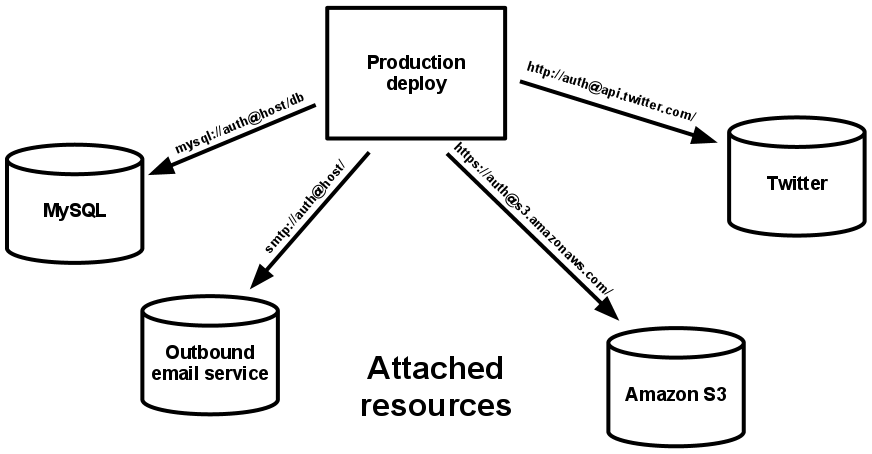
A better approach is to store configuration in environment variables. These are easy to change for each environment at runtime, are not likely to be checked into version control, and are independent of programming language and development framework.

## Backing/External services

*Treat backing services as attached resources.*

A backing/external service is any service the app consumes over the network as part of its normal operation. The application should access it  as a service and externalized in the configuration. Examples include datastores (such as MySQL or CouchDB), messaging/queueing systems (such as RabbitMQ or Beanstalkd), SMTP services for outbound email (such as Postfix), and caching systems (such as Memcached).

You should think of these backing services as abstractions for the underlying resource. This ensures the application is loosely coupled with the services so it can easily switch provider or instance if needed. What this effectively means is that it shouldn't require any code change to swap a compatible backing service. The only change should be in configurations.



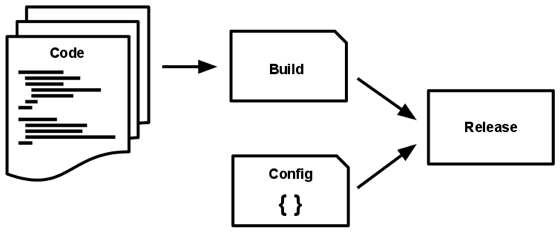
## Build, release, run

*Build / Release and Run phases must be kept separated.*

It's important to separate the software deployment process into three distinct stages: build, release, and run. Each stage should result in an artifact that's uniquely identifiable. Every deployment should be linked to a specific release that's a result of combining an environment's configuration with a build. This allows easy rollbacks and a visible audit trail of the history of every production deployment.

The build stage is usually triggered automatically when you commit code that has passed all of the required tests. The build stage takes the code, fetches the required libraries and assets, and packages these into a self-contained binary or container. The result of the build stage is a build artifact.

When the build stage is complete, the release stage combines the build artifact with the configuration of a specific environment. This produces a release. The release can be automatically deployed into the environment by a continuous deployment app. Or you can trigger the release through the same continuous deployment app.



Finally, the run stage launches the release and starts it.

## Processes

*Execute the app as one or more stateless processes.*

The Cloud Native Apps is running in the environment as one or more processes. These processes should be stateless and should not share data with each other. This allows the apps to scale up through replication of their processes. Creating stateless apps also make processes portable across the computing infrastructure.

If the app is using the concept of "sticky" sessions, this requires a change in how to handle and persisting data. Because processes can go away at any time, you can't rely on the contents of local storage being available, or that any subsequent request will be handled by the same process. Therefore, you must explicitly persist any data that needs to be reused in an external backing service such as a database.

## Port binding

Export services via port binding.

This point is related to the exposition of the application to the outside.

Cloud Native Apps can use specialized dependencies (such as http server, ...) and exposes its service through a port.

The host has the responsibility to route the request to the correct application through port mapping.

With Kubernetes execution environments the ports will be exposed by respective pods with necessary deployment configuration.

## Concurrency

*Scale out via the process model*

The app can be seen as a set of processes of different types

* web server
* worker
* cron

Each process needs to be able to scale horizontally, it can have its own internal multiplexing. Necessary Kubernetes objects must be applied.

## Disposability

*Maximize robustness with fast startup and graceful shutdown.*

Each process of an application must be disposable.

* it must have a quick startup
  + ease the horizontal scalability
* it must ensure a clean shutdown
  + stop listening on the port
  + finish to handle the current request
  + usage of a queueing system for long lasting (worker type) process

In summary, the application should expose idempotent services. This is another very desirable attribute of a service destined for cloud deployments. This gives the flexibility to stop, move, or spin new services at any time without any other considerations.

When designing and implementing the Cloud Native Apps the following recommendation should be used:

* Decouple functionality such as state management and storage of transactional data using backing services.
* Manage environment variables outside of the app so that they can be used at runtime.
* Make sure that the startup time is minimal. This means that you must decide how much layering to build into images when you use virtual machines, such as public versus custom images. This decision is specific to each app and should be based on the tasks that startup scripts perform. For example, if you're downloading several packages or binaries and initializing them during startup, a sizeable portion of your start-up time will be dedicated to completing these tasks.
* Use the SIGTERM signal (when it's available) to initiate a clean shutdown.

## Dev/prod parity

*Keep development, staging, and production as similar as possible.*

Cloud Native Apps app must **keep the gap between development and production environment as minimal as possible.** These gaps can result from long development cycles, different teams involved, or different technology stack in use. Using containers and Kubernetes as orchestrators make this happen very easily.

Containers and Kubernetes standardize how you deliver your application and its running dependencies, meaning that you’re able to deploy everything the same way everywhere.

Moreover, the right continuous integration and delivery with GitLab CI must be implemented to facilitate bridging this gap further.

## Logs

*Treat logs as event streams.*

Logs provide you with awareness of the health of your apps. It's important to decouple the collection, processing, and analysis of logs from the core logic of the apps. Decoupling logging is particularly useful when your apps require dynamic scaling and are running on cloud platform, because it eliminates the overhead of managing the storage location for logs and the aggregation from distributed (and often ephemeral) VMs.

## 12. Admin processes

*Run admin/management tasks as one-off processes*

Instead of running administrative and management processes from the application you are running, the best practice is to define a set of one-off processes that should run on-demand and execute them in same way as application processes (dependency isolation).

Usually used for maintenance task, though a REPL, admin process must be executed on the same release (codebase + configuration) than the application.

## Further Reading

[12 Factor App](https://12factor.net/)

1. Microservices and Cloud Native Applications

## What are Microservices?

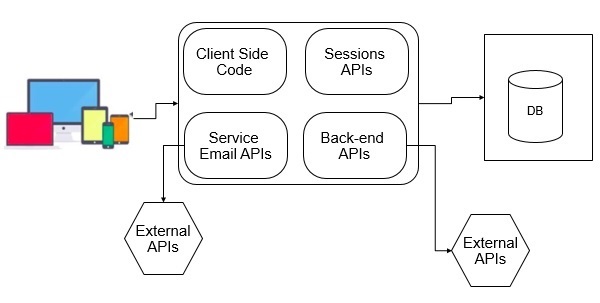
Microservices is a system used to develop an application, built on a selection of individual services working together to ensure seamless and highly responsive performance.

As opposed to monolithic development, where everything is merged and, therefore, dependent on one another, the microservice architecture consists of multiple modules of autonomous components. It is a type of architecture relying on the system of loose coupling.

## How do Microservices Talk to Each Other?

Microservices communicate with each other through **APIs**. **Application Programming Interface** (**API**) is an access point of each modular functionality that allows modules to interact. A microservice uses this interface to send and respond to requests from other modules.

Each microservice needs to have a clearly defined **API endpoint** (one end of a communication channel) to ensure it is always accessible for inquiries. In most cases, these are stateless **REST APIs**. Without distinctly defined endpoints, the system doesn’t know where to seek the information it needs.

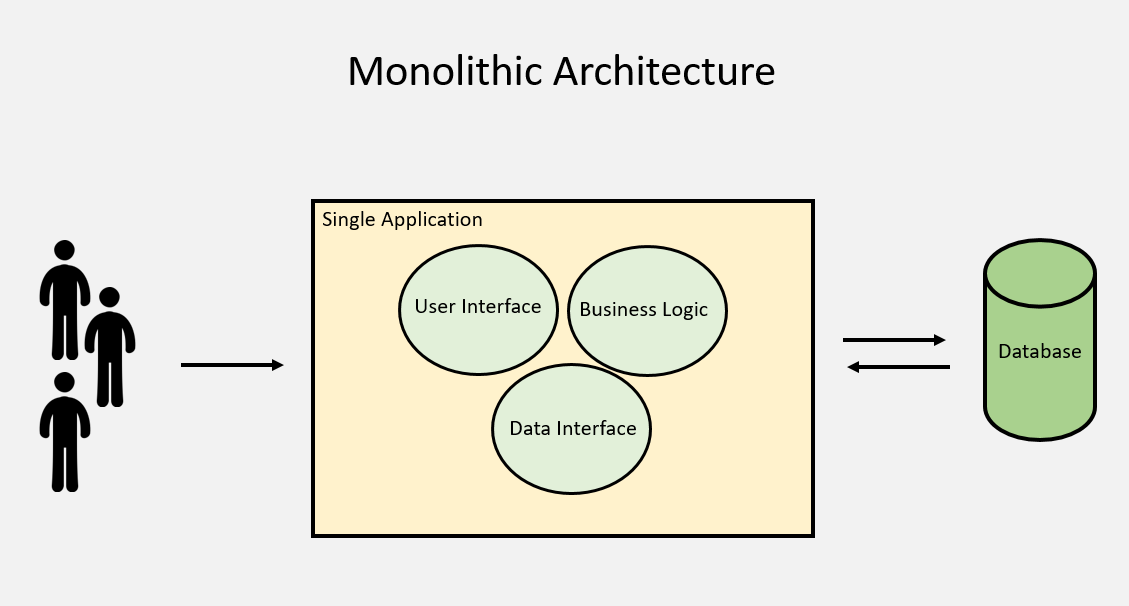


During the Development process the official standards and patterns to share endpoints among themselves should be used. It is responsibility of Development Team to establish the REST API through collaboration and clear communication.

## The Difference Between Traditional Architecture and Microservices Architecture

### Monolithic Applications

Monolithic architecture worked fine for traditional server-side systems. These are systems based on a single application. All the functionalities reside in one structure, use the same file system, communicate with the same server, and are eventually deployed on the same machine.



This kind of architecture allows developers to create applications faster, as they only need to build the essential features to which they would later add-on other functionalities. Also, the performance was faster as the process doesn’t involve APIs.

### Monolithic benefits

Monolithic architecture is a conventional solution for building applications. The following are some advantages of adopting a monolithic design for your application:

* To deploy a monolithic application, simply copy the packaged application to a server.
* All modules in a monolithic application share memory, space, and resources, so you can use a single solution to address cross-cutting concerns such as logging, caching, and security.
* The monolithic approach can provide performance advantages, because modules can call each other directly. By contrast, microservices typically require a network call to communicate with each other.

### Monolithic challenges

Complex monolithic apps often become progressively harder to build, debug, and deploy. At some point, the problems exceed the benefits:

* Applications typically grow over time. It can become complicated to implement changes in a large and complex application that has tightly coupled modules. Because any code change affects the whole system, you have to thoroughly coordinate changes. Coordinating changes makes the overall development and testing process much longer compared to microservice applications.
* It can be complicated to achieve continuous integration and deployment (CI/CD) with a large monolith. This complexity is because you must redeploy the entire application in order to update any one part of it. Also, it's likely that you have to do extensive manual testing of the entire application to check for regressions.
* Monolithic applications can be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU‑intensive image-processing logic. Another module might be an in‑memory database. Because these modules are deployed together, you have to compromise on the choice of hardware.
* Because all modules run within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire system.
* Monolithic applications add complexity when you want to adopt new frameworks and languages. For example, it is expensive (in both time and money) to rewrite an entire application to use a new framework, even if that framework is considerably better.

### Microservices-based applications

A microservice typically implements a set of distinct features or functionality. Each microservice is a mini‑application that has its own architecture and business logic. For example, some microservices expose an API that's consumed by other microservices or by the application's clients, such as third-party integrations with payment gateways and logistics.



### Microservices benefits

A microservices architecture provides the following benefits:

* Autonomous teams can independently develop individual services. You can organize microservices around business boundaries, not the technical capabilities of a product. You organize your teams for a single, independent responsibility for the entire lifecycle of their assigned piece of software from development to testing to deployment to maintenance and monitoring.
* Independent microservice development process also lets developers write each microservice in a different programming language, creating a polyglot application.
* When you decouple capabilities out of a monolith, you can have the independent teams release their microservice independently. Independent release cycles can help improve your teams' velocity and product time to market.
* Microservices architecture also lets you scale each service independently. You can deploy the number of instances of each service that satisfy its capacity and availability constraints. You can also use the hardware that best matches a service's resource requirements. When you scale services independently, you help increase the availability and the reliability of the entire system.

### Microservices challenges

Microservices have some challenges when compared to monoliths, including the following:

* A major challenge of microservices is the complexity that's caused because the application is a distributed system. Developers need to choose and implement an inter‑services communication mechanism. The services must also handle partial failures and unavailability of upstream services.
* Another challenge with microservices is that you need to manage transactions across different microservices. Business operations that update multiple business entities are fairly common, and they are usually applied in an atomic manner in which either all operations are applied or everything fails. When you wrap multiple operations in a single database transaction, you ensure atomicity.
* In a microservices‑based application, business operations might be spread across different microservices, so you need to update multiple databases that different services own. If there is a failure, it's non-trivial to track the failure or success of calls to the different microservices and roll back state. The worst case scenario can result in inconsistent data between services when the rollback of state due to failures didn't happen correctly.
* Comprehensive testing of microservices-based applications is more complex than testing a monolithic application. For example, to test the functionality of processing an order in a monolithic ecommerce service, you select items, add them to a cart, and then check out. To test the same flow in a microservices-based architecture, multiple services - such as frontend, order, and payment - call each other to complete the test run.
* Deploying a microservices‑based application is more complex than deploying a monolithic application. A microservice application typically consists of many services, each of which has multiple runtime instances.
* A microservices architecture adds operations overhead because there are more services to monitor and alert on. Microservice architecture also has more points of failure due to the increased points of service-to-service communication. A monolithic application might be deployed to a small application server cluster. A microservices-based application might have tens of separate services to build, test, deploy and run, potentially in multiple languages and environments. All of these services need to be clustered for failover and resilience. Productionizing a microservices application requires high-quality monitoring and operations infrastructure.
* The division of services in a microservice architecture allows the application to perform more functions at the same time. However, because the modules run as isolated services, latency is introduced in the response time due to network calls between services.
* Not all applications are large enough to break down into microservices. Also, some applications require tight integration between components—for example, applications that must process rapid streams of real-time data. Any added layers of communication between services may slow real-time processing down. Thinking about the communication between services beforehand can provide helpful insights in clearly marking the service boundaries.

### Reasons for Building Microservices

To understand why we would venture down this path, consider the typical challenges inherent in monolithic applications:

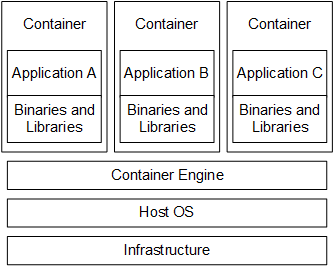
* For every change, the entire application needs to be rebuilt and redeployed, irrespective of how large or small the change is. 15–30-minute build times are not uncommon for large applications.
* A small change in one part of an application has the potential to break the entire system.
* As the application grows in size, its parts tend to become more intertwined and the codebase becomes difficult to understand and maintain.
* Large applications have a correspondingly large resource footprint — they typically consume more memory and require more computing power. As a result, they must be hosted on large servers with sufficient resource capacity. This also limits their ability to scale.
* They also tend to have a slow startup time, which is not ideal, given that even the tiniest changes tend to require a complete redeployment. They are less suited for the Cloud and cannot easily take advantage of ephemeral computing, such as spot instances.
* A single technology is used to implement the entire application, often a compromise between generality and the needs of specific areas of the application. Java and .Net are likely candidates for monoliths because they are among the best “all-rounder” languages, not because they are amazingly good at any particular task.
* Team scalability is naturally constrained by a large codebase. The more complex the application (in terms of internal dependencies), the more difficult it is to comfortably accommodate large teams of developers, without people stepping on each other’s toes.

## Further Reading

1. Microservice for Dummies (it is free, but need some registration process): <https://redis.com/docs/redis-microservices-for-dummies/>
2. Containers: moving to Microservice-based architecture

## What are Containers?

Containers are a form of operating system virtualization. A single container might be used to run anything from a small microservice or software process to a larger application. Inside a container are all the necessary executables, binary code, libraries, and configuration files.

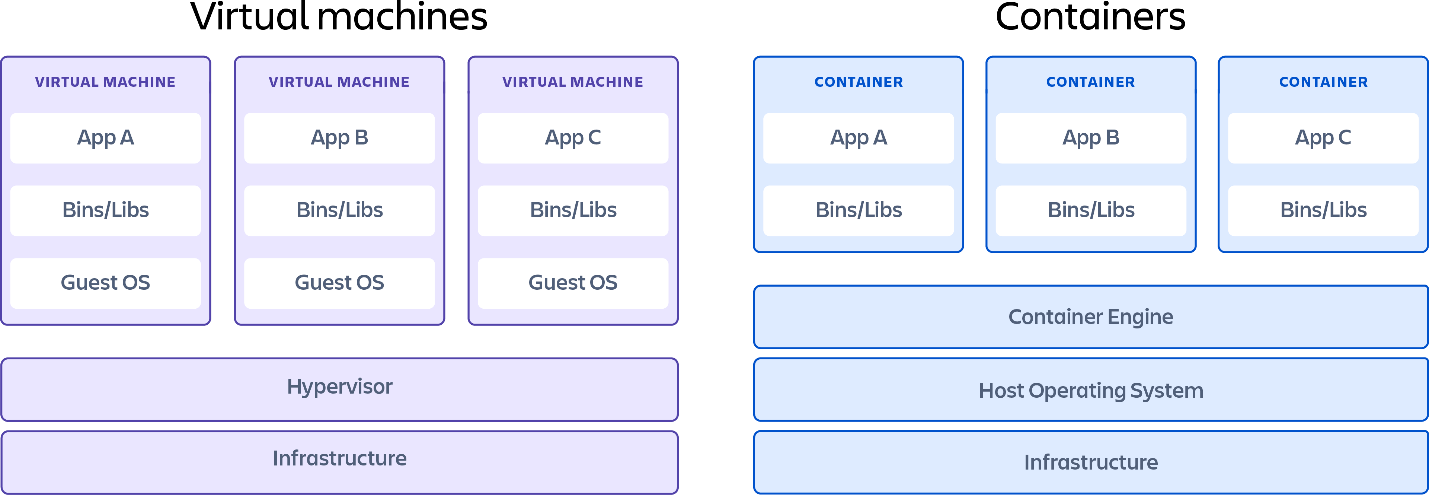


As can be seen from this diagram, a container includes an application plus any binaries or libraries that the application requires in order for it to run. The container runs under the control of the container engine (such as Docker or CRI-O), which in turn runs on top of the operating system (which can be Windows 10, Windows Server 2016, or Linux depending on the container engine being used). Micro Focus recommends that you always follow your operating system vendor's best practices when working with containers.

## Containers vs. Traditional Virtualization

Containers and VMs are very similar resource virtualization technologies. Virtualization is the process in which a system singular resource like RAM, CPU, Disk, or Networking can be ‘virtualized’ and represented as multiple resources. The key differentiator between containers and VMs is that VMs virtualize an entire machine down to the hardware layers and containers only virtualize software layers above the operating system level.

Unlike containers which are lightweight software packages that contain all the dependencies required to execute the contained software application, the VMs are heavy software packages that provide complete emulation of low-level hardware devices like CPU, disk and networking devices.



### VMs: Pros and Cons

### Pros

* **Full isolation security**  
  VMs run in isolation as  a fully standalone system. This means that VM's are immune to any exploits or interference from other VMs on a shared host. An individual VM can still be hijacked by an exploit but the exploited VM will be isolated and unable to contaminate any other neighboring VMs.
* **Interactive Development**  
  Once the basic hardware definition is specified for a VM the VM can then be treated as a bare bones computer. Software can manually be installed to the VM and the VM can be snapshotted to capture the current configuration state. The VM snapshots can be used to restore the VM to that point in time or spin up additional VM's with that configuration.

### Cons

* **Iteration speed**  
  VMs are time consuming to build and regenerate because they encompass a full stack system. Any modifications to a VM snapshot can take significant time to regenerate and validate they behave as expected.
* **Storage size cost**  
  VMs can take up a lot of storage space. They can quickly grow to several Gigabytes in size. This can lead to disk space shortage issues on the VMs host machine

### Containers: Pros and Cons

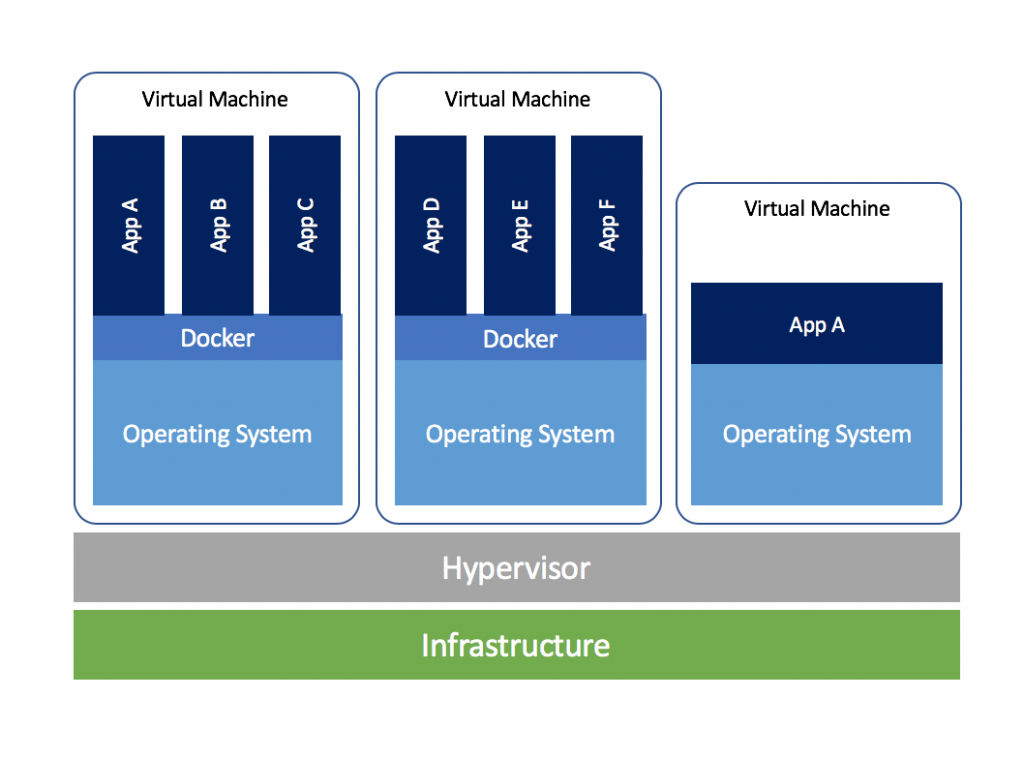
### Pros

* **Iteration speed**  
  Because containers are lightweight and only include high level software, they are very fast to modify and iterate on.
* **Robust Ecosystem**  
  Most container runtime systems offer a hosted public repository of pre-made containers. These container repositories contain many popular software applications like databases or messaging systems and can be instantly downloaded and executed, saving time for development teams

### Cons

1. **Shared host exploits**  
   Containers all share the same underlying hardware system below the operating system layer, it is possible that an exploit in one container could break out of the container and affect the shared hardware. Most popular container run-times have public repositories of pre-built containers. There is a security risk in using one of these public images as they may contain exploits or may be vulnerable to being hijacked by nefarious actors

### Container and VM tougher usage

It is entirely possible to use Containers and VM in unison although the practical use-cases may be limited. A VM can be created that emulates a unique hardware configuration. An operating system can then be installed within this VM's hardware. Once the VM is functional and boots the operating system, a container runtime can be installed on the operating system. At this point we have a functional computational system with emulated hardware that we can install Containers on.

## Containers Building Blocks (Docker Containers)

There are a number of different components to Docker that need to understand:

**The Docker Engine**

The Docker Engine is the workhorse of the solution and consists of three main elements:

* **Docker host**: This is the server that runs the Docker engine as an operating system (OS) daemon or process that lives forever.
* **API:** The Docker engine’s REST API dictates how the program can interact with the daemon.
* **Command line interface (CLI):** It’s a command line that allows an administrator to interact with the API, which interacts with the server

**Dockerfile**

Everything starts with a Dockerfile. The Dockerfile:

* is a text file which contains a series of commands or instructions.
* these instructions are executed in the order in which they are written.
* execution of these instructions takes place on a base image.
* on building the Dockerfile, the successive actions form a new image from the base parent image.

**Images**

A Docker image is the result of running the docker build command with the Dockerfile as the input. An image is just a static (read-only), nonrunning representation of all the various components that define the workload. Think of an image as a workload template that sits on storage, awaiting the day when it’s put into service.  
  
**Containers**

When you start up an image on a Docker host, you have a running container. You can have a lot of running containers that are based on a single image. But a running container is more than just a running image. It’s an image to which a read/write layer has been attached atop all the read-only information that’s present in the container.

**The Registry**

A registry is a centralized repository that holds Docker images. You can use public docker container Registry such as Docker Hub or private ones. If you want to fully control where your images are stored or tightly integrate images into your local development processes, consider deploying a private registry, such as GitLab Container Registry.

## Further Reading and Practices

1. Reading:

Containers for Dummies: <https://www.qcmtech.com/wp-content/uploads/2017/09/HPE-pub-10010-Containers-for-Dummies.pdf>

1. Practices:

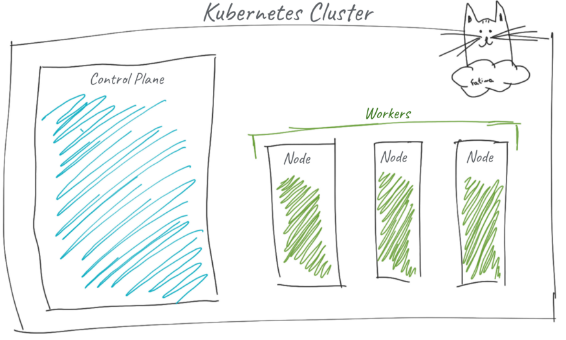
Getting started with Docker: <https://docs.docker.com/get-started/>

# Kubernetes: management of containers

## What is Kubernetes?

Kubernetes is derived from the Greek word κυβερνήτης (kubernḗtēs), which means pilot. The Kubernetes logo of a ship's steering wheel further enforces the idea of piloting or managing, which is exactly what Kubernetes does with Docker containers. Kubernetes manages Docker containers in a variety of ways so it does not have to be done manually.

Manually managing numerous containers can create similar issues to managing virtual machines. However, managing containers is especially important because cloud companies bill for things like computing time and storage. You don’t want to have many running containers doing nothing for this reason. In addition, you also don’t want one container taking a network load it cannot handle by itself. Kubernetes is designed to solve problems like these.

Kubernetes is often referred to as K8s for simplicity because of the 8 letters in between “K” and “s”.

## Kubernetes Architecture

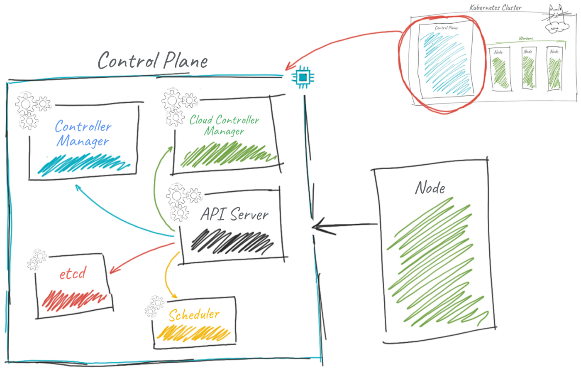
Kubernetes connects multiple servers/VMs into a cluster. From high-level view the K8s clusters as composed from control plane nodes (servers/VMs) and worker nodes. For high availability, the control plane runs on multiple nodes.

### The Control Plane

The cluster’s brain is called the control plane, and it runs all the tasks required for Kubernetes to do its job: scheduling containers, managing Services, serving API requests, etc.

The members of the cluster which run the control plane components are called master nodes.

The control plane is actually made up of several components:

* **API Server** - This is the frontend server for the control plane, handling API requests.
* **etcd** - the database where Kubernetes stores all its information: what nodes exist, what resources exist on the cluster, and so on.
* **Scheduler** - decides where to run newly created Pods.
* **Controller Manager -** responsible for running resource controllers, such as Deployments.
* **Cloud Controller Manager -**This interacts with the cloud provider (in cloud-based clusters), managing
* resources such as load balancers and disk volumes.

### Worker Nodes

Cluster members that run user workloads are called worker nodes. Each worker node in a Kubernetes cluster runs these components:

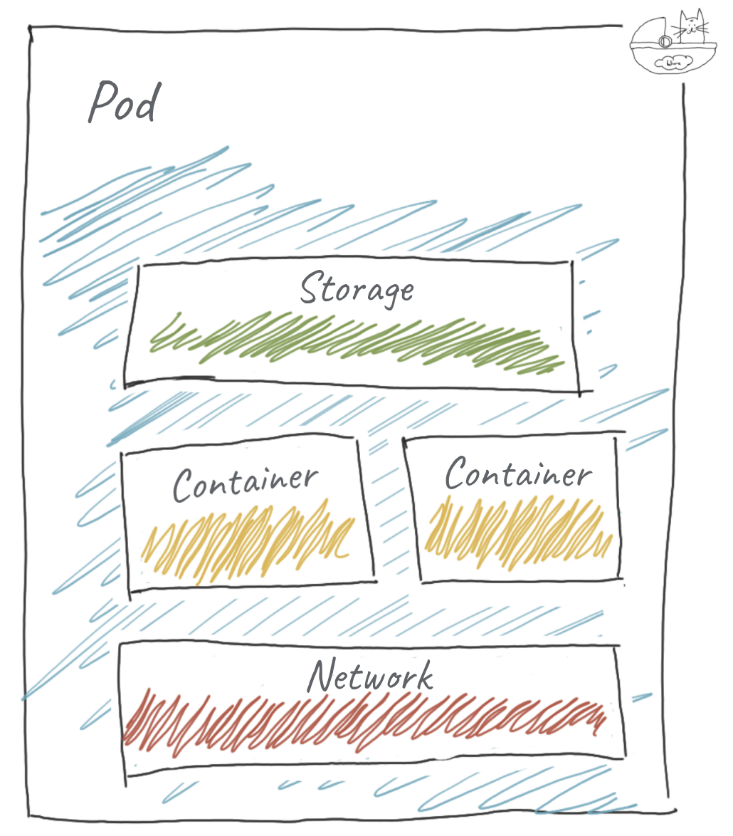
* **Kubelet** - responsible for driving the container runtime to start workloads that are scheduled on the node, and monitoring their status.
* **kube-proxy** - does the networking magic that routes requests between Pods on different nodes, and between Pods and the internet.
* **Container runtime** - starts and stops containers and handles their communications. Usually Docker, but Kubernetes supports other container runtimes, such as rkt and CRI-O.



Other than running different software components, there’s no intrinsic difference between master nodes and worker nodes. Master nodes don’t usually run user workloads, though, except in very small clusters (like Docker Desktop or Minikube).

## Kubernetes Basic Building Block: PODs

Pods are one of the most fundamental building blocks within Kubernetes. They are the smallest deployable object. Their goal is to provide a runtime environment for your containers.



The Pod provides a running environment for the containers, including networking and storage. A Pod can have multiple containers if necessary. Containers in the same Pod are tightly coupled and share resources (networking and storage, security rules, etc.).

There is only one IP assigned to a Pod, regardless of the number of containers it has. Meaning, inside a Pod, the containers share the same network namespace (IP and Ports). The consequence is that containers within a Pod can communicate through localhost (127.0.0.1).

Optionally a Pod can also specify one or more storage methods, referred to as Volumes. All containers in the Pod can access mounted volumes.

Example of Pod object (pod.yalm):

apiVersion: v1

kind: Pod

metadata:

name: nginx

spec:

containers:

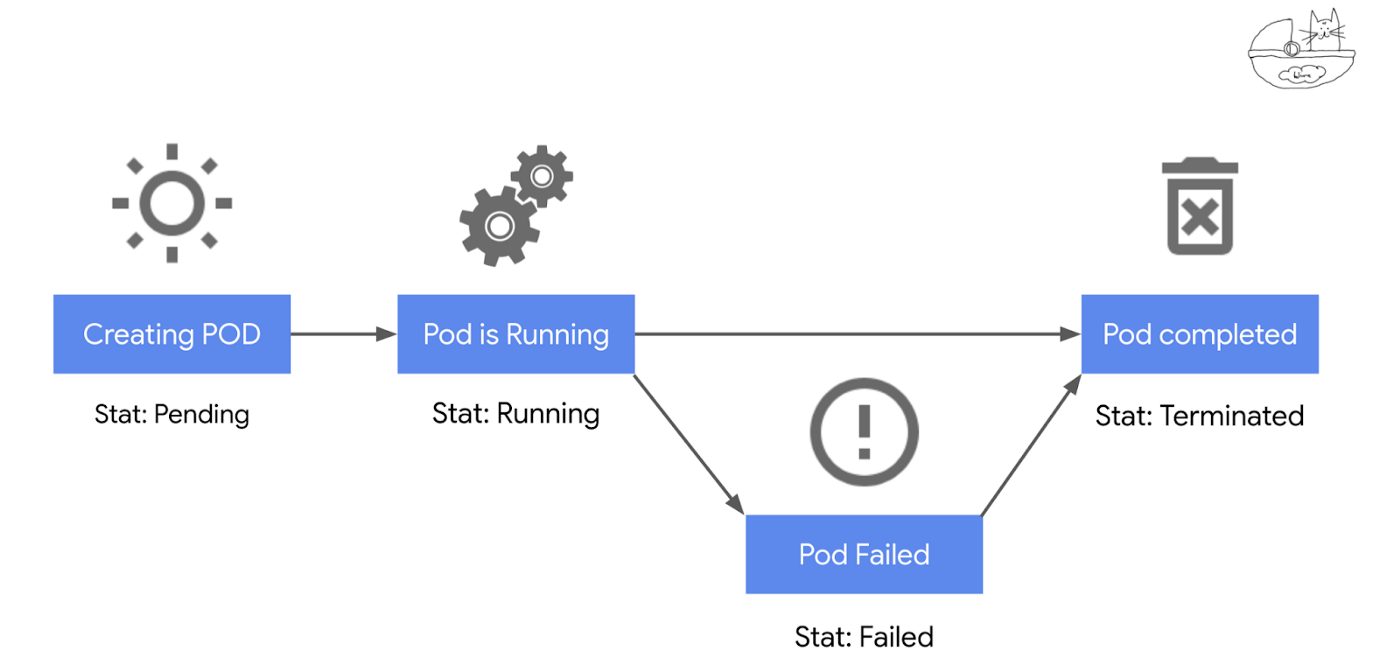
- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

### Life of a Pod



First, we create the Pod. In this stage, Kubernetes fetches the container from the registry. Then k8s ask a node to run the Pod; which causes the creation of necessary constructs around networking and storage. Then it runs the container in that space. If fetching the container fails, the Pod will assume the “UNKNOWN” state.

The process in the container will be in the foreground until completion or until an external command terminates it. This is the Running state. If the Pod fails the state is set to FAILED. If it runs successfully, it assumes the status of TERMINATED.

Let see how to create a POD. For this the following comment need to be run: *kubectl create -f pod.yaml*

When we ran that command the following is happen:

* kubectl makes a call to the API Server to create the object
* API Server saves the Object in etcd
* Scheduler detects a new Pod was created and is not assigned to a node
* The scheduler then updates the Pod object with the chosen node. It takes into account several constraints to define which node gets the Pod. These are labels and taints that define affinity (run groups of pods together) or anti-affinity (avoid scheduling pods together) rules.
* Controller Manager then detects that the Pod desired state doesn’t match the current state and makes the necessary calls to the API Server (it’s a bi-directional flow); at this point, the correct controller will respond to do these changes
* Which in turn triggers an API call from the API Server to kubelet, in the selected node, to create the actual Pod
* As part of setting up the Pod, networking kube-proxy does the required changes to the OS via the package filtering layer

## Kubernetes Deployment and Service Objects

### Deployment

Deployment adds several important things to Pods. In essence, deployments allow k8s to have **desired state rules** on a Pod or group of Pods.

Kubernetes Deployment also makes use of other k8s objects to further enhance functionality. One super common construct is the “ReplicaSet”, which defines how many copies of the same Pod should be created by this deployment. A copy of a Pod is called a replica. If we say a deployment contains 3 replicas it means it will be running 3 copies of the same Pod. This means that, at any given time, k8s will ensure that the desired amount of Pods are running as part of this deployment. If a pod dies, k8s will start a new one to replace it and match our desired state.

Here is how a declaration of a Deployment object look likes for Nginx Pod (deployment.yaml):

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Let see how to create a Deployment. For this the following comment need to be run: *kubectl create -f deployment.yaml*

### Service

Deployment adds several important things to Pods. In essence, deployments allow k8s to have **desired state rules** on a Pod or group of Pods.

Kubernetes Deployment also makes use of other k8s objects to further enhance functionality. One super common construct is the “ReplicaSet”, which defines how many copies of the same Pod should be created by this deployment. A copy of a Pod is called a replica. If we say a deployment contains 3 replicas it means it will be running 3 copies of the same Pod. This means that, at any given time, k8s will ensure that the desired amount of Pods are running as part of this deployment! If a pod dies, k8s will start a new one to replace it and match our desired state.

Here is how a declaration of a Deployment object look likes for Nginx Pod (deployment.yaml):

apiVersion: apps/v1

kind: Deployment

metadata:

name: nginx-deployment

labels:

app: nginx

spec:

replicas: 3

selector:

matchLabels:

app: nginx

template:

metadata:

labels:

app: nginx

spec:

containers:

- name: nginx

image: nginx:stable-alpine

ports:

- containerPort: 80

Let see how to create a Deployment. For this the following comment need to be run: *kubectl create -f deployment.yaml*

## Further Reading and Practices

1. Reading:

Kubernetes for Dummies: <https://medium.com/@mfsilv/kubernetes-a-gentle-introduction-9d23de7f00e0>

1. Practices:

Getting started with Kubernetes: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

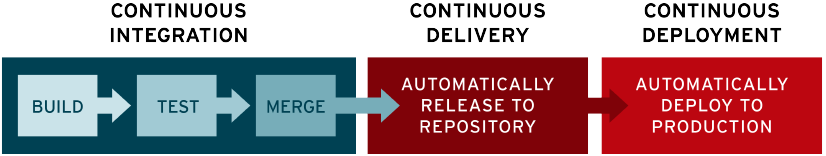
# Cloud Native Application Delivery Automation: CI/CD approach

CI/CD is a method to frequently deliver applications to customers by introducing automation into the stages of applications development. The main concepts attributed to CI/CD are continuous integration, continuous delivery, and continuous deployment.

CI/CD process introduces ongoing automation and continuous monitoring throughout the lifecycle of application, from integration and testing phases to delivery and deployment. Taken together, these connected practices are often referred to as a "CI/CD pipeline" and are supported by development and operations teams working together in an agile way.

## What's the difference between CI and CD?

The acronym CI/CD has a few different meanings.



The "**CI**" in CI/CD always refers to continuous integration, which is an automation process for developers.

In modern application development, the goal is to have multiple developers working simultaneously on different features of the same app. However, if an organization is set up to merge all branching source code together on one day (known as “merge day”), the resulting work can be tedious, manual, and time-intensive. That’s because when a developer working in isolation makes a change to an application, there’s a chance it will conflict with different changes being simultaneously made by other developers. This problem can be further compounded if each developer has customized their own local integrated development environment (IDE.

CI helps developers merge their code changes back to a shared branch, or “trunk,” more frequently—sometimes even daily. Once a developer’s changes to an application are merged, those changes are validated by automatically building the application and running different levels of automated testing, typically unit and integration tests, to ensure the changes haven’t broken the app. This means testing everything from classes and function to the different modules that comprise the entire app. If automated testing discovers a conflict between new and existing code, CI makes it easier to fix those bugs quickly and often.

The "**CD**" in CI/CD refers to continuous delivery and/or continuous deployment, which are related concepts that sometimes get used interchangeably.

Following the automation of builds and unit and integration testing in CI, continuous delivery automates the release of that validated code to a repository. So, in order to have an effective continuous delivery process, it’s important that CI is already built into your development pipeline. The goal of continuous delivery is to have a codebase that is always ready for deployment to a production environment.

In continuous delivery, every stage—from the merger of code changes to the delivery of production-ready builds—involves test automation and code release automation. At the end of that process, the operations team is able to deploy an app to production quickly and easily.

The final stage of a mature CI/CD pipeline is continuous deployment. As an extension of continuous delivery, which automates the release of a production-ready build to a code repository, continuous deployment automates releasing an app to production. Because there is no manual gate at the stage of the pipeline before production, continuous deployment relies heavily on well-designed test automation.

In practice, continuous deployment means that a developer’s change to a cloud application could go live within minutes of writing it (assuming it passes automated testing). This makes it much easier to continuously receive and incorporate user feedback. Taken together, all of these connected CI/CD practices make deployment of an application less risky, whereby it’s easier to release changes to apps in small pieces, rather than all at once. There’s also a lot of upfront investment, though, since automated tests will need to be written to accommodate a variety of testing and release stages in the CI/CD pipeline.

## What is a CI/CD pipeline?

A CI/CD pipeline is a series of steps that must be performed in order to deliver a new version of software. It is possible to manually execute each of the steps of a CI/CD pipeline, the true value of CI/CD pipelines is realized through automation.

### Elements of a CI/CD pipeline

The steps that form a CI/CD pipeline are distinct subsets of tasks grouped into what is known as a pipeline stage. Generally speaking, a CI/CD pipeline should be composed of one or several of the following stages:

#### The Trigger

**The best pipelines are triggered automatically when new code is committed to the repository.**

The CI/CD tool can be configured to poll for changes to a source code repository, or can be set up a Webhook to notify the CI/CD tool whenever a developer makes a *push*.

#### Code checkout

In this first stage, the CI server will check out the code from the source code repository. The CI/CD tool usually receives information from a poll, or a webhook, which says which specific commit triggered the pipeline. The pipeline then checks out the source code at a given commit point, and starts the process.

#### Compile the code

If you’re developing in a compiled language, the first thing needs to do is to compile your program.

This means that the CI tool needs to have access to whatever build tools needed to compile the application. For example, for Java something like Maven or Gradle will be used.

Ideally this stage should run in a clean, fresh environment. This is one of the great [use cases for Docker containers](https://www.tutorialworks.com/why-use-containers/) – being able to create fresh build environments easily and repeatably.

#### Run unit tests

The next key element of your CI/CD pipeline is unit testing. This is the stage where the CI/CD tool is configured to execute the tests that are in the codebase.

The aim at this point is not only to verify that all the unit tests pass, but that the tests are being maintained and enhanced as the code base grows.

If the application is growing rapidly, but the number of tests stays the same, this isn’t great, because it could mean that there are large parts of the code base which are untested.

#### Package the code

Once all of the tests are passing, next is to move on to packaging the code. Exactly how to package the application depends on the programming language and target environment.

In case of Docker containers, the Docker image will be built.

Whatever packaging format is chosen, a good practice is that to build the binary only once. Don’t build a different binary for each environment, because this will cause the pipeline to become very complex.

#### Run acceptance tests

Now comes the time to perform acceptance testing on the application. Acceptance tests are a way of ensuring that your software does what it is meant to do, and that it meets the original requirements.

#### Delivery or Deployment

Finally, when the application has been tested, it can move into the delivery or deployment stage.

At this stage, it is an artifact ready to be deployed (continuous delivery) which can continue automatically deploy of the application (continuous deployment).

#### Example of CI/CD pipeline:

1. **compile** the code and **package** it into an executable or intermediate format
2. perform all required **tests** and code **analysis**:
   * unit testing
   * code quality audits
   * Static Application Security Testing ([SAST](https://en.wikipedia.org/wiki/Static_program_analysis))
   * dependencies check
3. **package** the compiled code into an executable format (Docker image)
4. **create** the test environment
5. **deploy** the code into a test environment
6. perform all required **acceptance tests** on the deployed application
   * functional testing (using an automated browser or a tool to test the APIs)
   * performance testing
   * Dynamic Application Security Testing ([DAST](https://en.wikipedia.org/wiki/Dynamic_application_security_testing))
7. **publish** the validated code and/or package to the Repository
8. **deploy to production environment**

## CI/CD Tools

The description of the most used CI/CD Tools can be found here:

<https://www.guru99.com/top-20-continuous-integration-tools.html>

## Further Reading and Practices

1. CI/CD Tutorial : <https://www.youtube.com/watch?v=h9K1NnqwUvE>