CLOUD NATIVE AND KUBERNETES

[Image Courtesy: https://medium.com/swlh/what-is-cloud-native-computing-d7514622f6cb]

AGENDA

- Application architectures (Monolithic vs Microservice)
- Cloud Native Applications
- Containerisation
- Container Orchestration
- Basics of Docker
- What is Kubernetes?
- Kubernetes Architecture
- Introduction to YAML
- Pods
- Replica Sets
- Deployment
- Services
- State Persistence

PART I

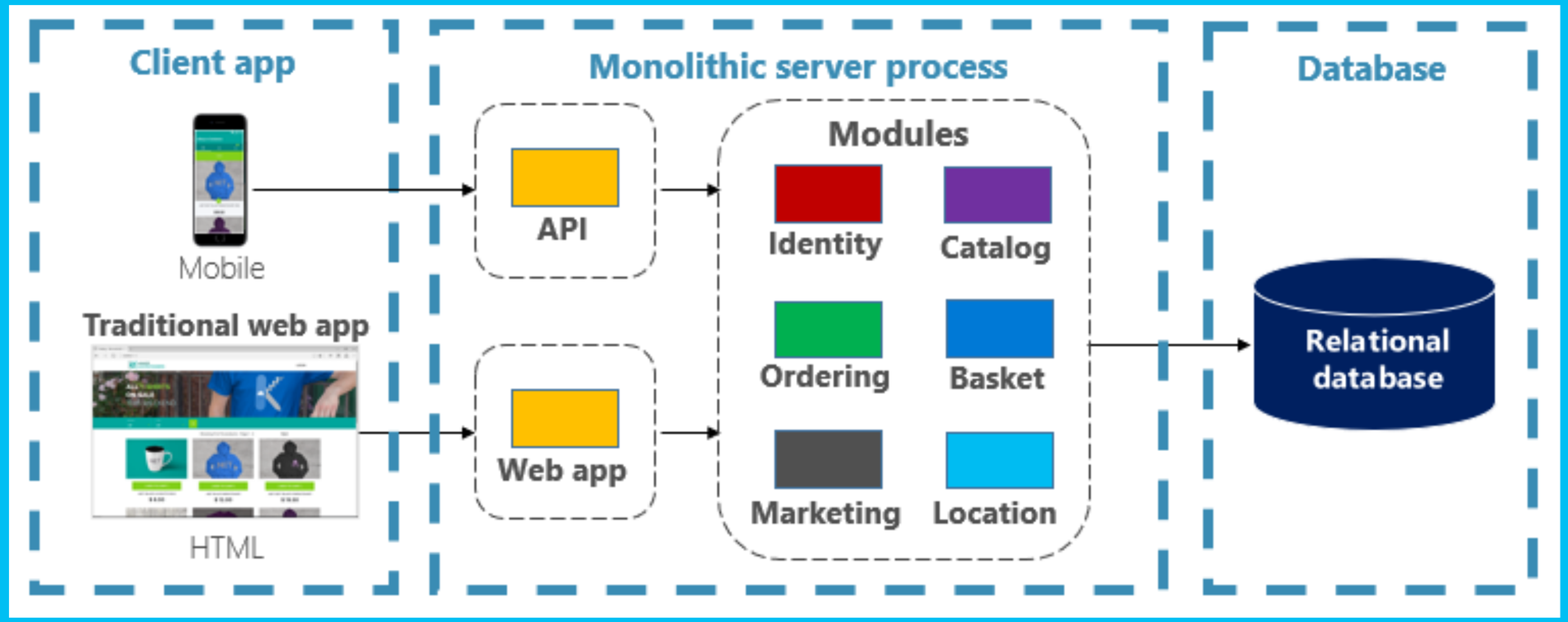
- Application architectures (Monolithic vs Microservice)
- Cloud Native Applications
- Containerisation
- Container Orchestration

LETS BUILD AN E-COMMERCE APPLICATION!

HMM...HOW DO WE BUILD THIS?

SOME QUESTIONS TO CONSIDER

- How do you want your customer to access your application? [Web, Mobile, Desktop, all of the above]
- What features do you need?
- What DB to use?
- What Stack to use?
- Where to host?
- How many customers are you expecting?
- So on....



ONE POSSIBLE SOLUTION!

References:

https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduction

https://www.ibm.com/docs/en/contentclassificatio/8.8?topic=architecture-single-server-configuration

MONOLITHIC APPLICATION

- Build
- Test
- Deploy
- Troubleshoot
- Vertically scale

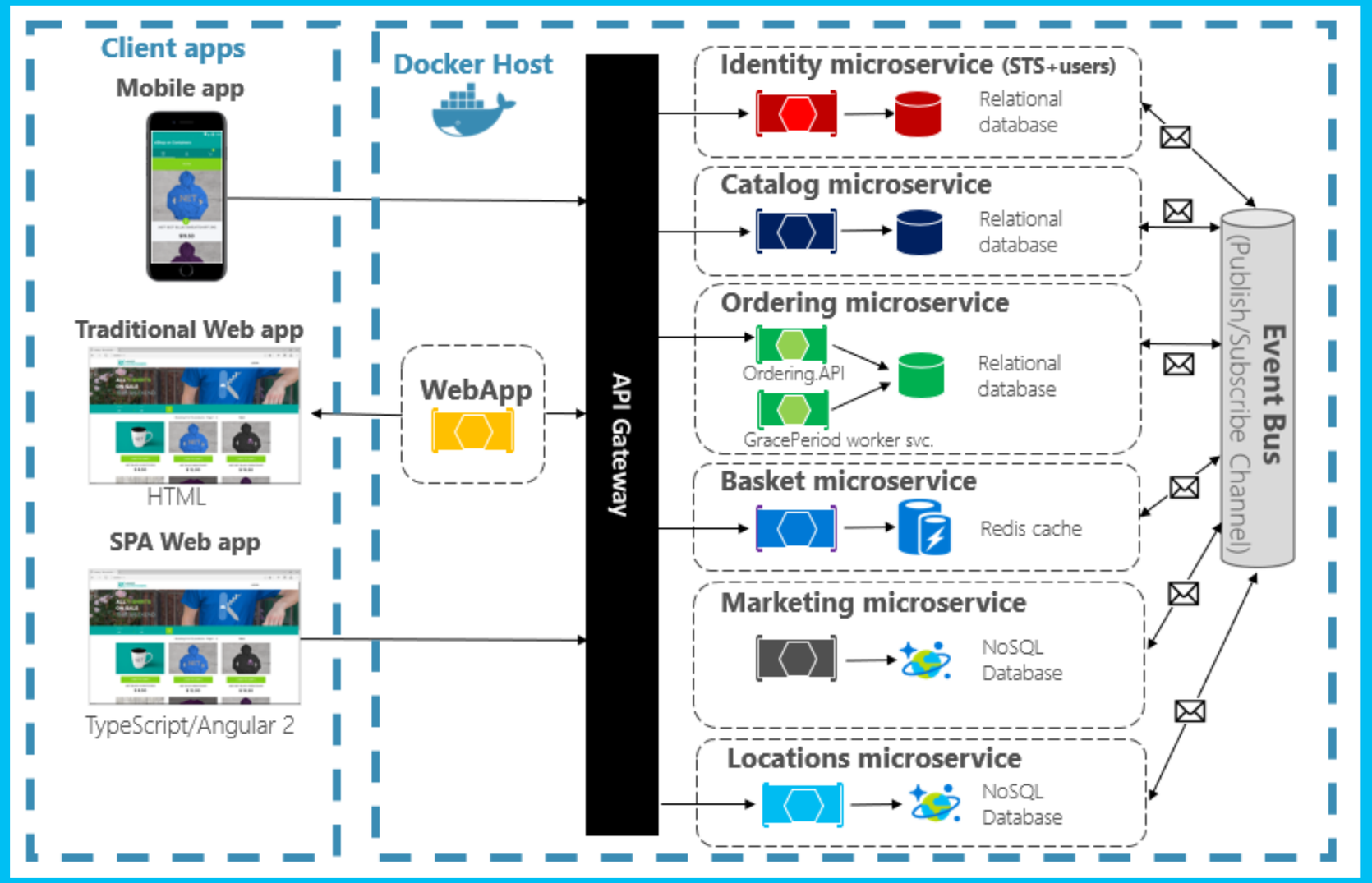
[Image Courtesy: https://blog.devgenius.io/the-connection-between-programming-memes-and-happiness-d768ab85b83d, https://ifunny.co/picture/rule-of-programming-if-it-works-don-t-touch-it-gajsw3tK9]

FEAR CYCLE

- Too Big....No single person understands it.
- Making Changes have severe side effects.
- New features/fixes become tricky, time-consuming, and expensive to implement.
- Each release becomes as small as possible and requires a full deployment of the entire application.
- One unstable component can crash the entire system.
- New technologies and frameworks aren't an option.
- It's difficult to implement agile delivery methodologies.
- Architectural erosion sets in as the code base deteriorates with never-ending "quick fixes."
- Finally, the *consultants* come in and tell you to rewrite it.

[References: https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduction]

IS THERE ANY WAY OUT??



THE ALTERNATIVE!

References:

https://docs.microsoft.com/en-us/dotnet/architecture/cloud-native/introduction

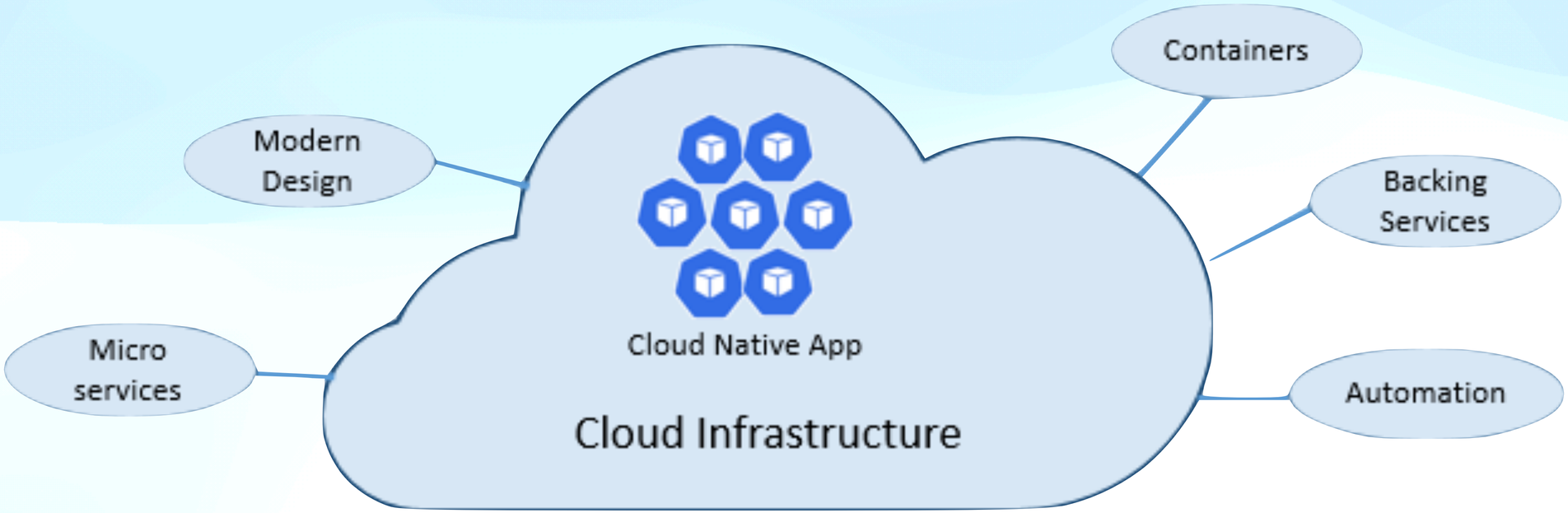
Cloud-native technologies empower organizations to build and run **scalable applications** in **modern, dynamic environments** such as **public, private, and hybrid clouds**. **Containers, service meshes, microservices, immutable infrastructure, and declarative APIs** exemplify this approach.

These techniques enable **loosely coupled** systems that are resilient, manageable, and observable. Combined with **robust automation**, they allow engineers to make high-impact changes **frequently and predictably** with **minimal toil**.

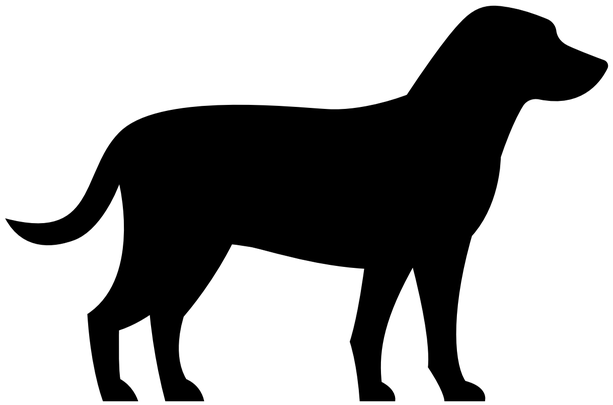
SOME NUMBERS!

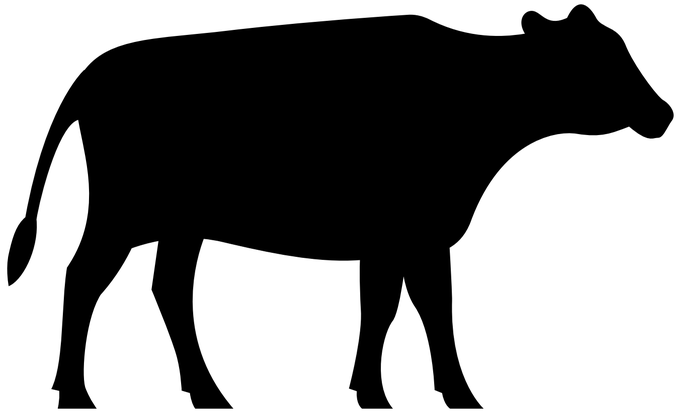
Company	Experience
Netflix	Has 600+ services in production. Deploys 100 times per day.
Uber	Has 1,000+ services in production. Deploys several thousand times each week.
WeChat	Has 3,000+ services in production. Deploys 1,000 times a day.

PILLARS OF CLOUD NATIVE



THE CLOUD

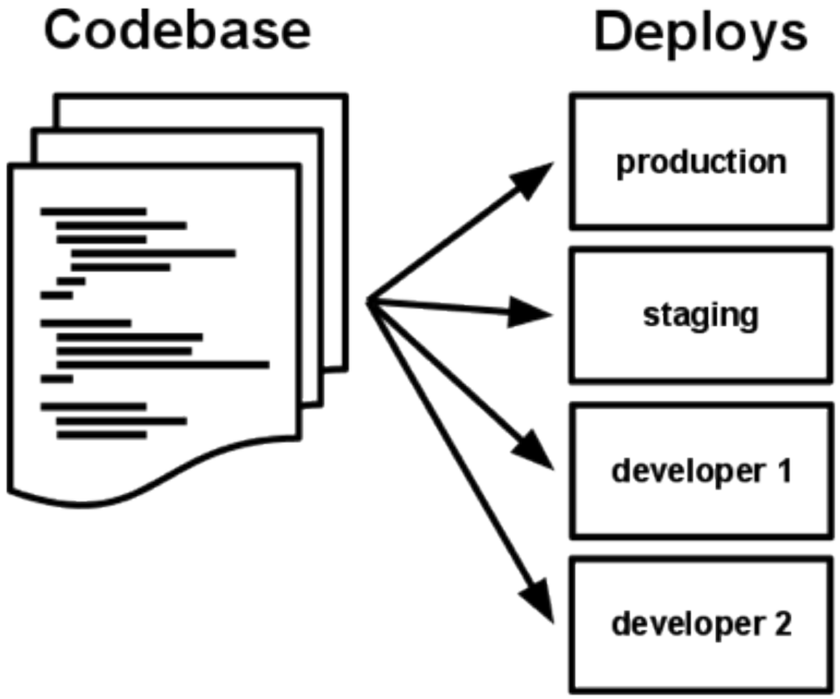


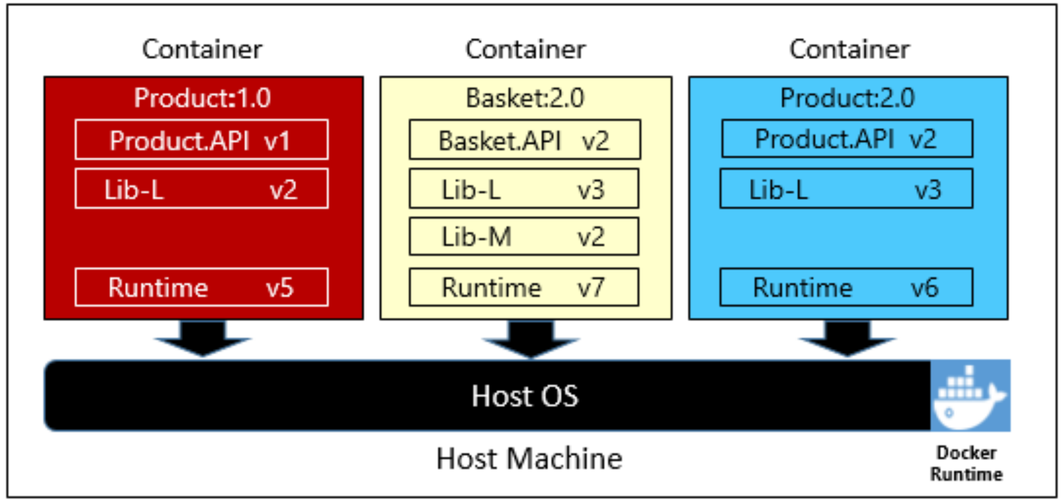


Pet vs Cattle model

MODERN DESIGN

- Twelve factor application [https://12factor.net]





Codebase

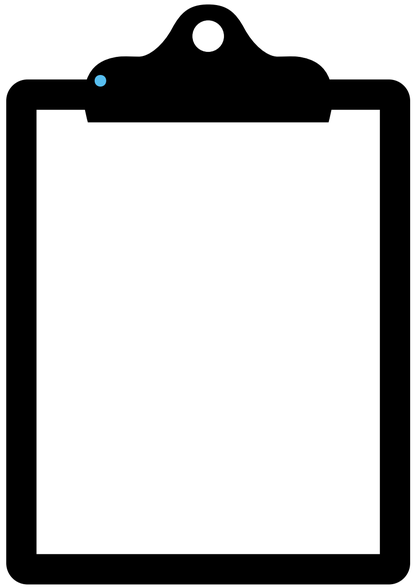
One codebase tracked in revision control, many deploys

Dependencies

Explicitly declare and isolate dependencies

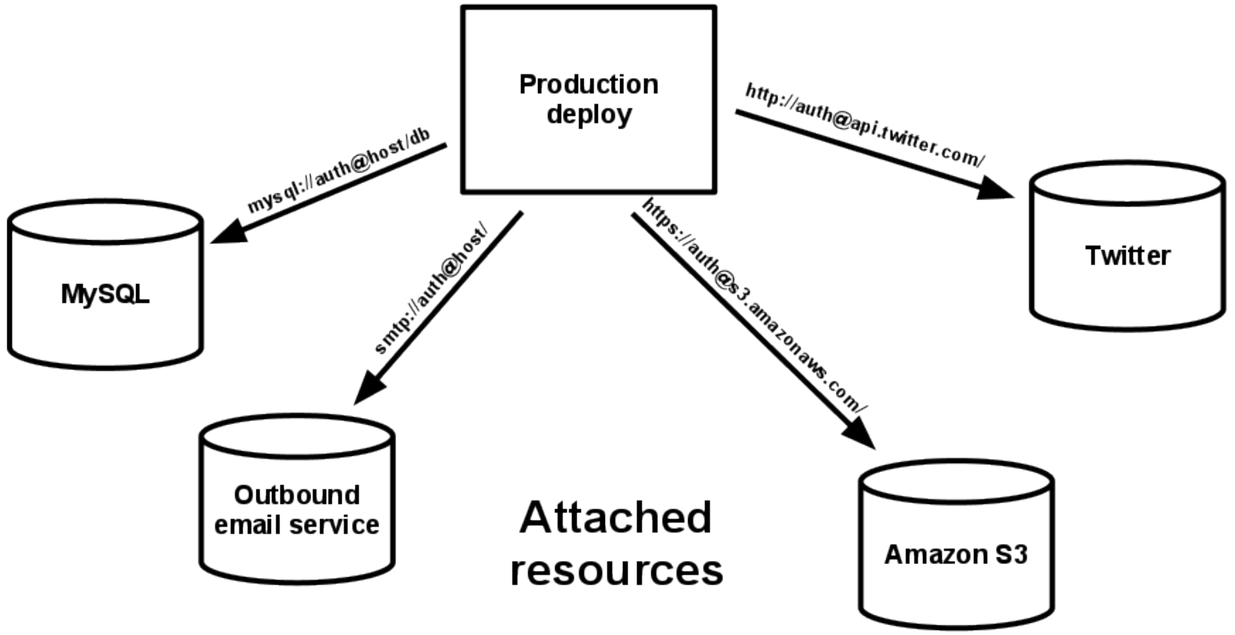
MODERN DESIGN

- Twelve factor application



Config

Store config in the environment

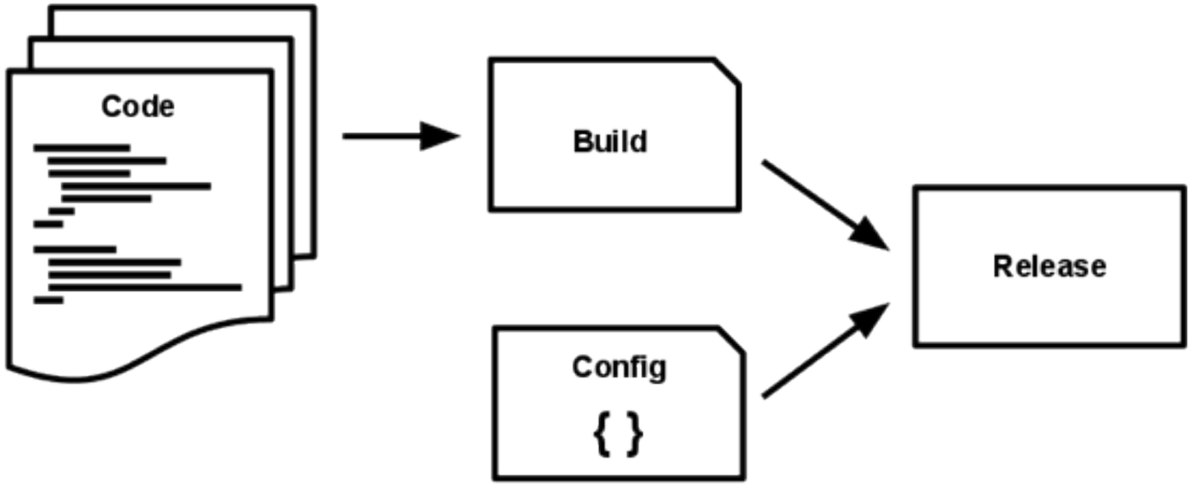


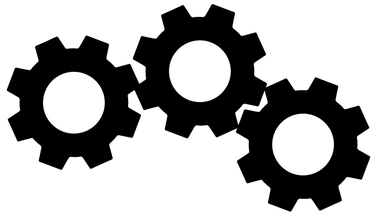
Backing services

Treat backing services as attached resources

MODERN DESIGN

- Twelve factor application





Build, release, run

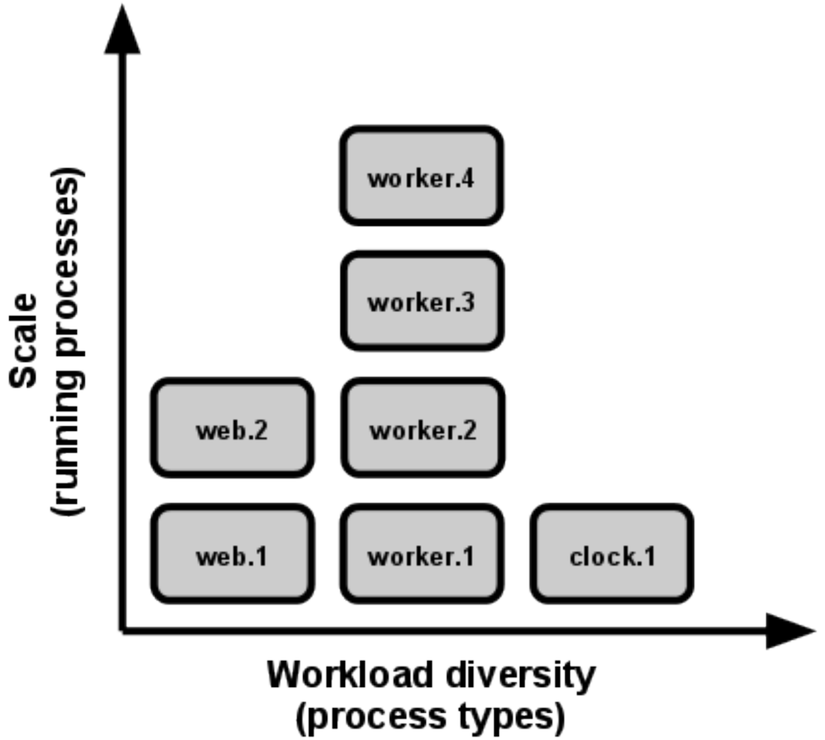
Strictly separate build and run stages

Processes

Execute the app as one or more stateless processes

MODERN DESIGN

- Twelve factor application



Port binding

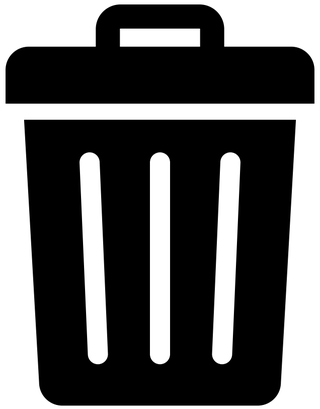
Export services via port binding

Concurrency

Scale out via the process model

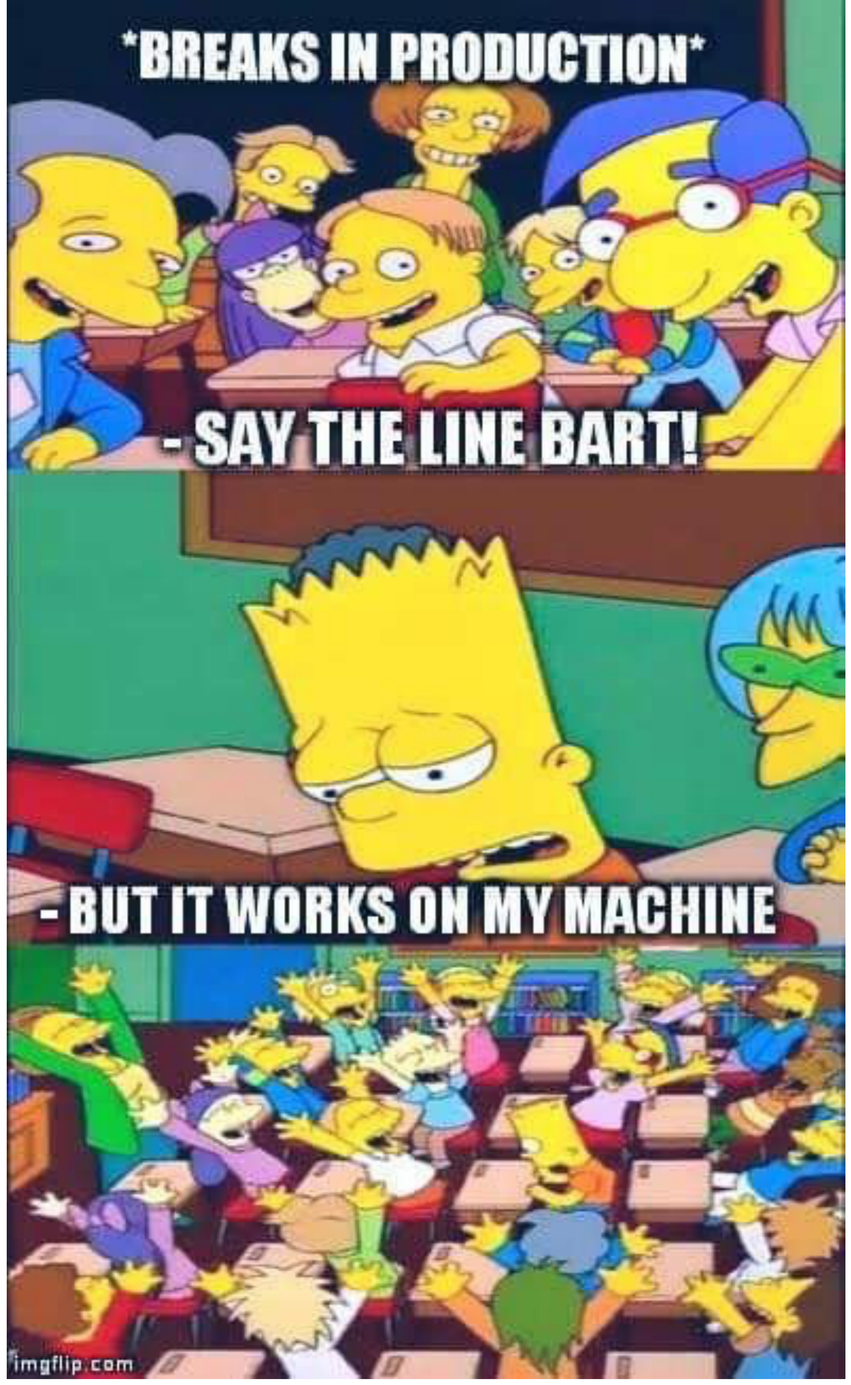
MODERN DESIGN

- Twelve factor application



Disposability

Maximize robustness with fast startup and graceful shutdown

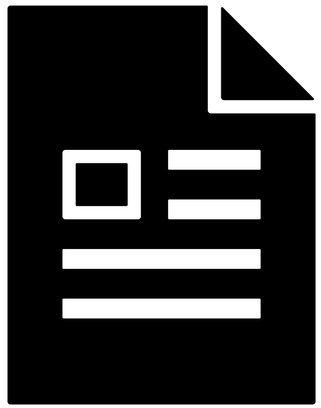


Dev/prod parity

Keep development, staging, and production as similar as possible

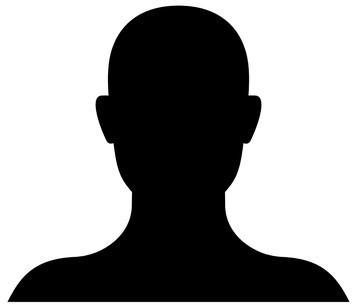
MODERN DESIGN

- Twelve factor application



Logs

Treat logs as event streams



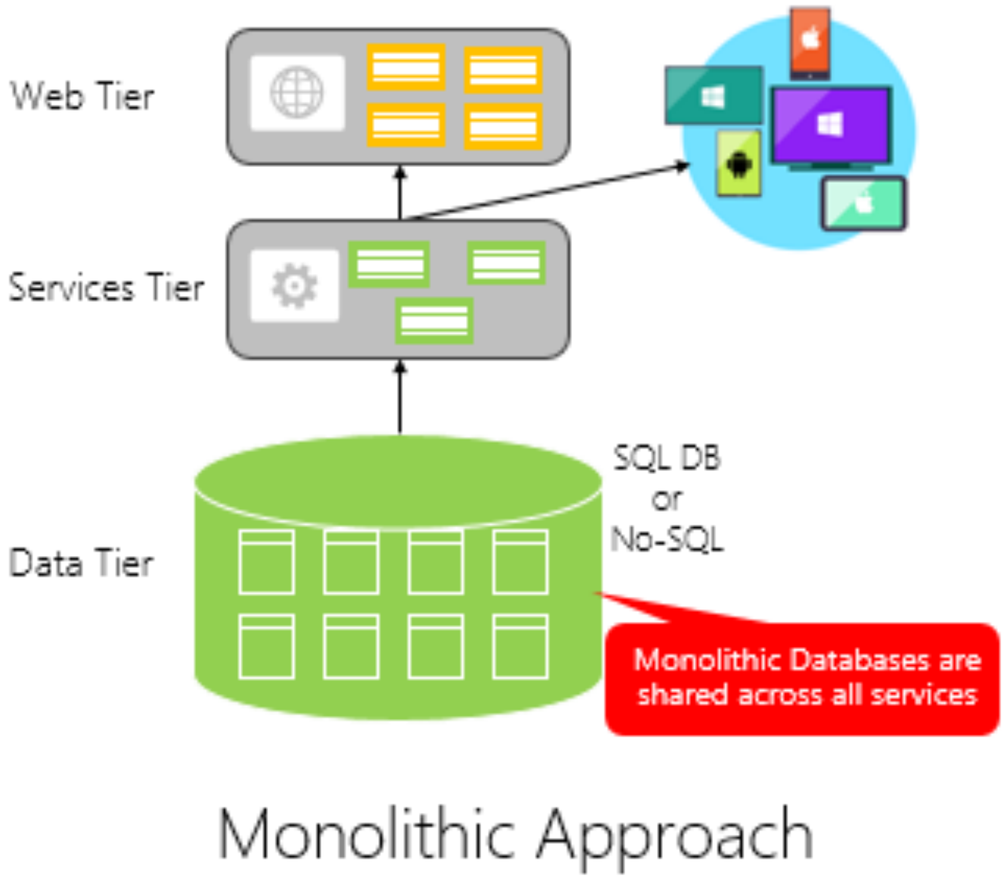
Admin processes

Run admin/management tasks as one-off processes

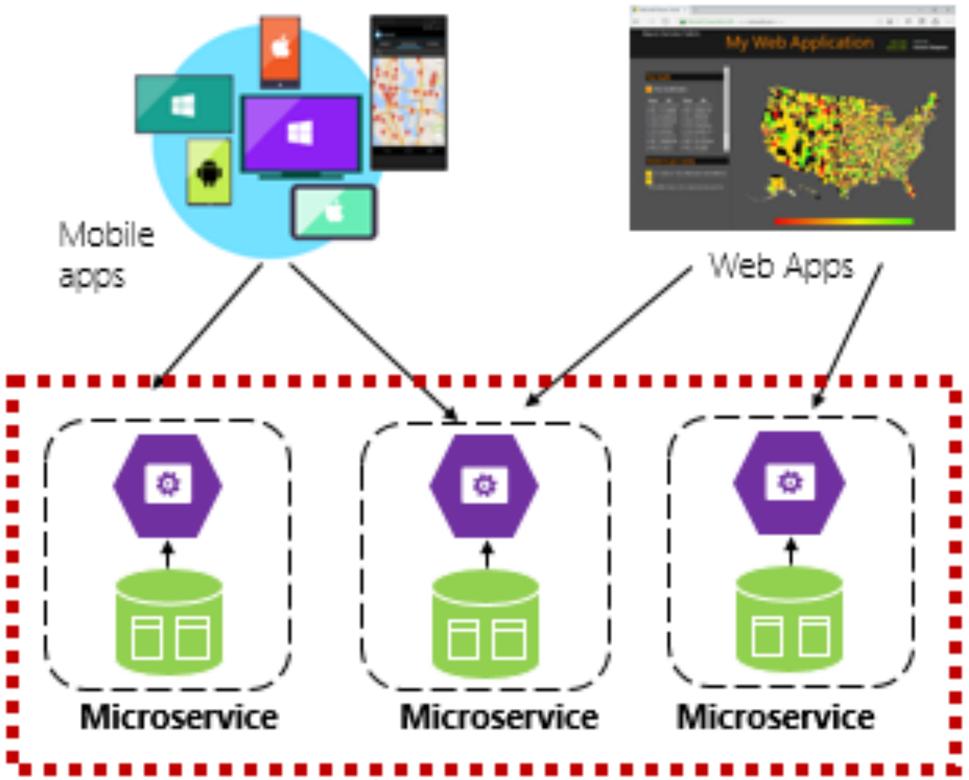
MODERN DESIGN

- **API First:** Make everything a service. Assume your code will be consumed by a front-end client, gateway, or another service.
- **Telemetry:** On a workstation, you have deep visibility into your application and its behavior. In the cloud, you don't. Make sure your design includes the collection of monitoring, domain-specific, and health/system data.
- **Authentication/ Authorization:** Implement identity from the start. Consider RBAC (role-based access control) features available in public clouds.

MICROSERVICE



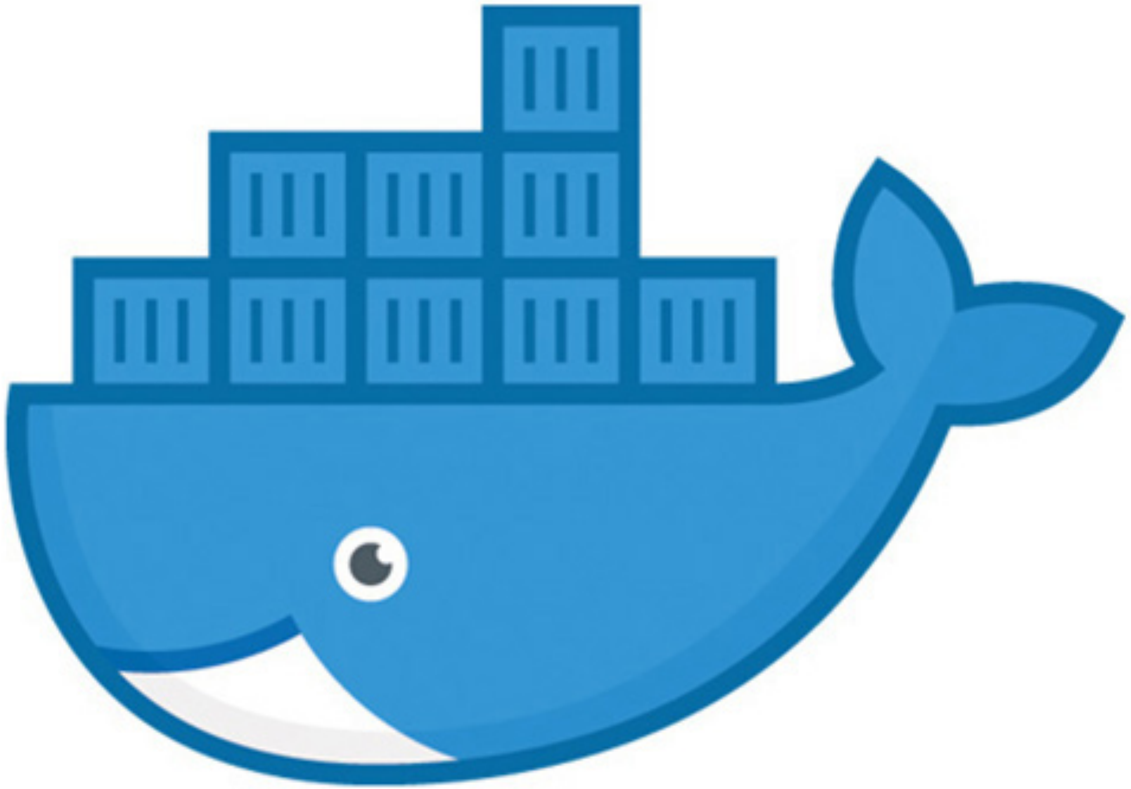
Monolithic Approach



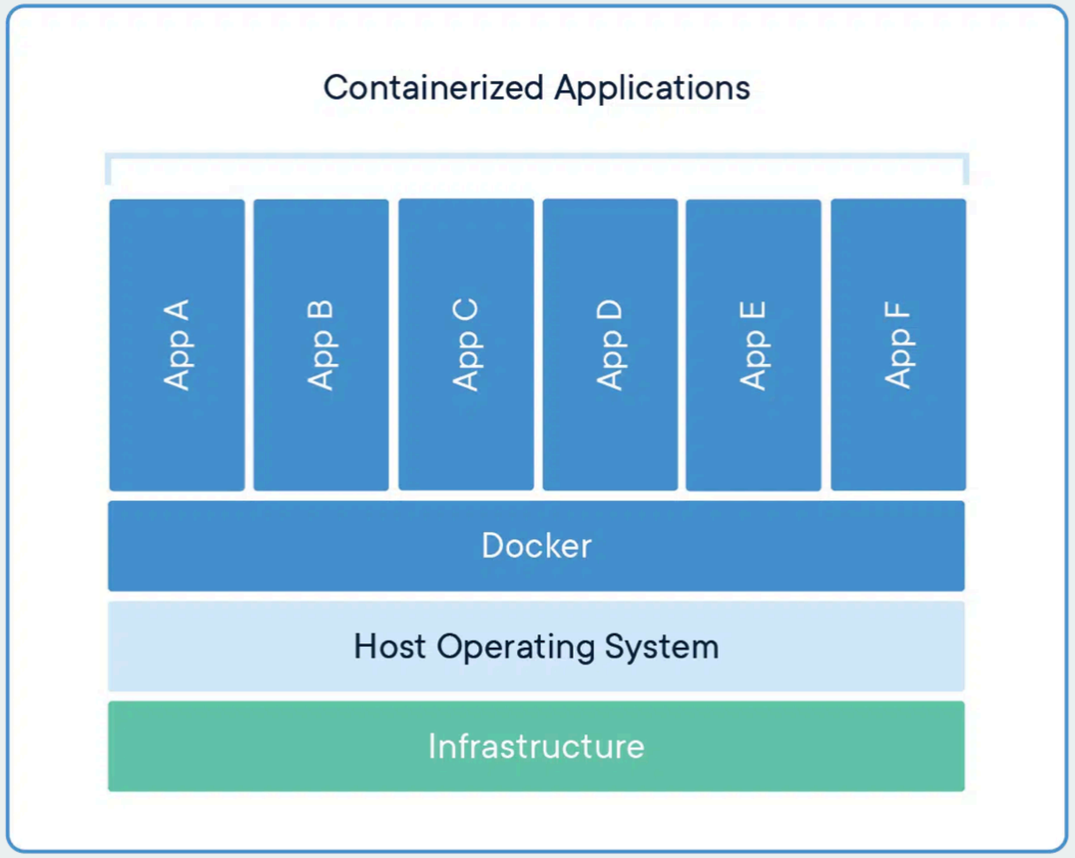
Microservices Approach

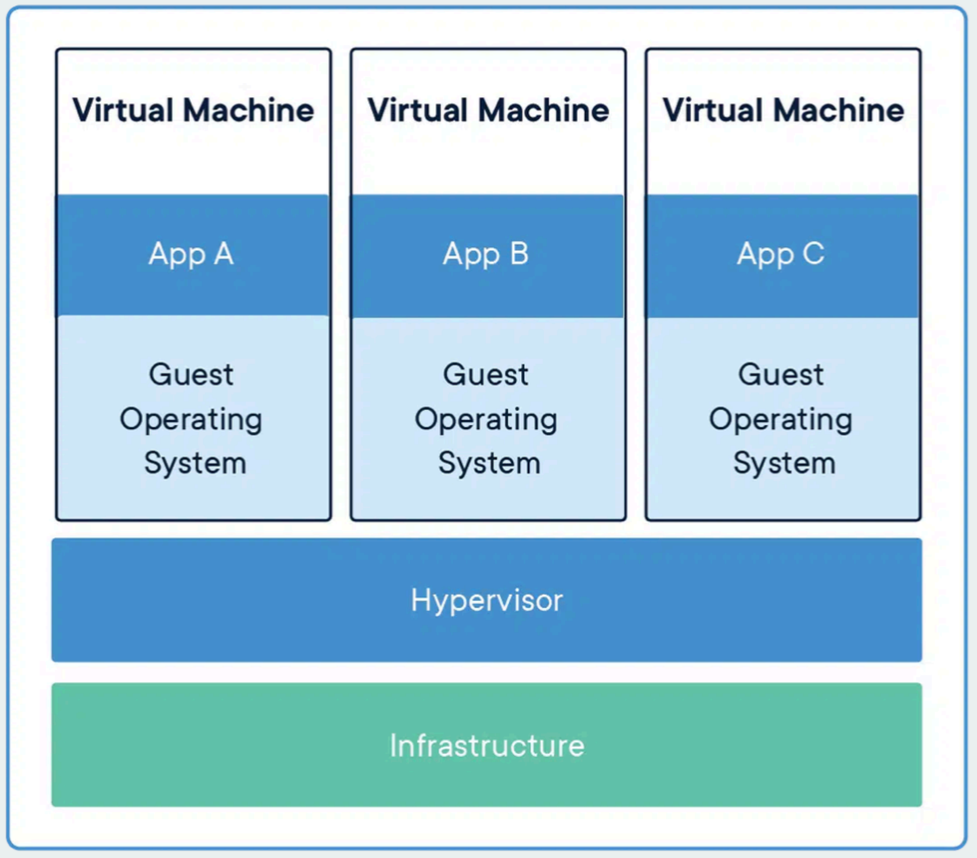
MICROSERVICE

- Communication
- Resiliency
- Distributed Data
- Secrets



CONTAINERS





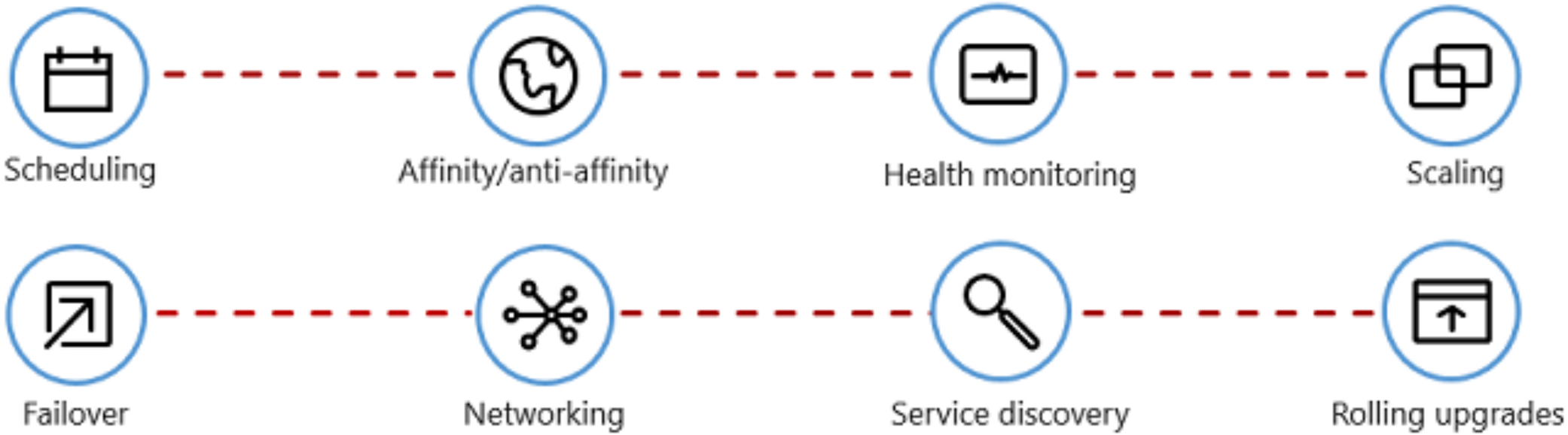
CONTAINERS

Containers are an abstraction at the app layer that packages code and dependencies together. Multiple containers can run on the same machine and share the OS kernel with other containers, each running as isolated processes in user space. Containers take up less space than VMs (container images are typically tens of MBs in size), can handle more applications and require fewer VMs and Operating systems.

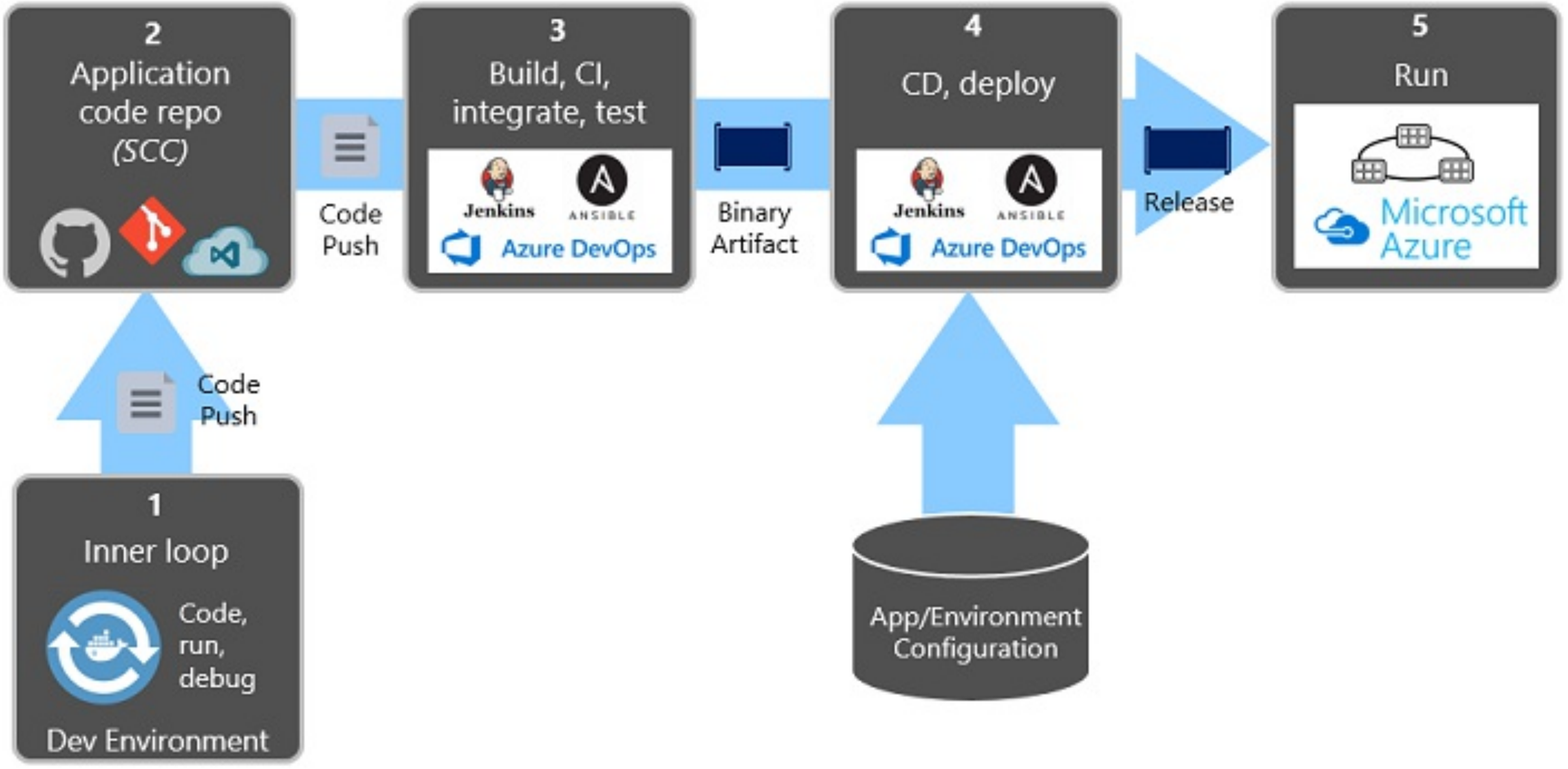
VIRTUAL MACHINES

Virtual machines (VMs) are an abstraction of physical hardware turning one server into many servers. The hypervisor allows multiple VMs to run on a single machine. Each VM includes a full copy of an operating system, the application, necessary binaries and libraries – taking up tens of GBs. VMs can also be slow to boot.

CONTAINER ORCHESTRATION



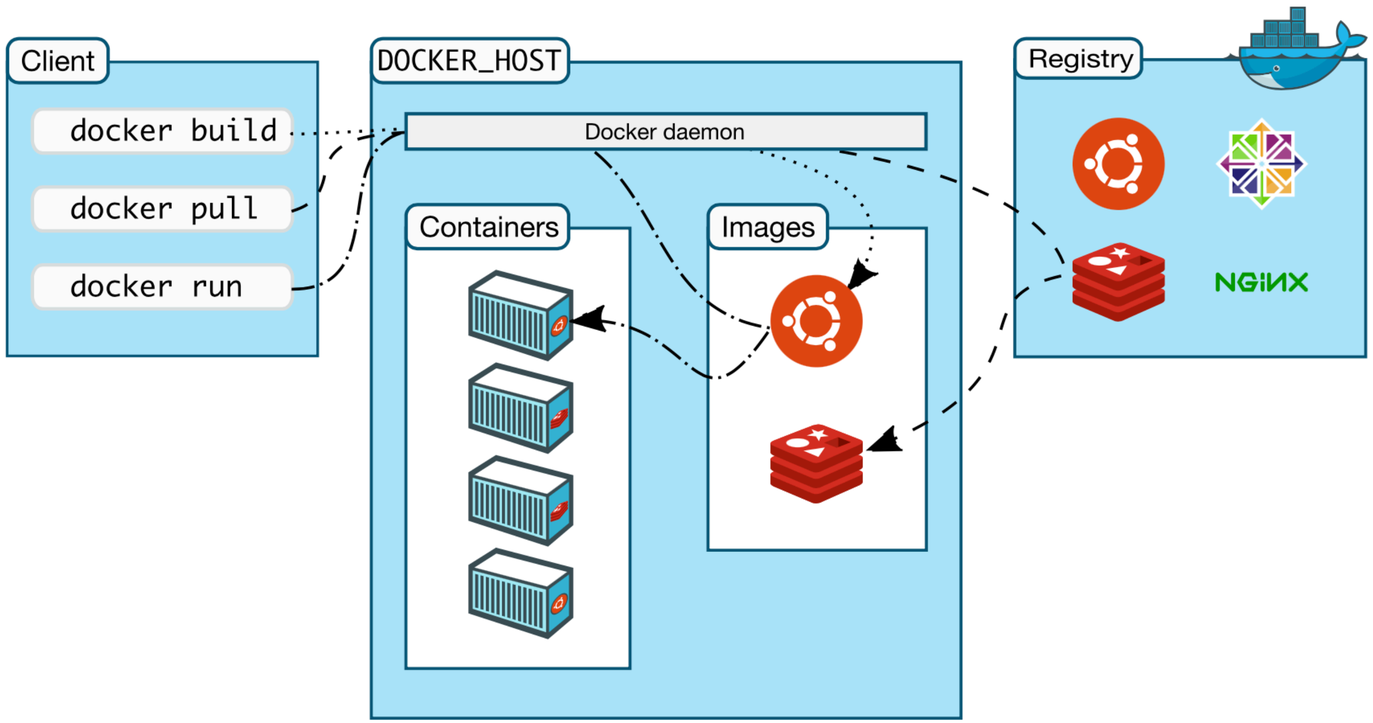
“Teams who implement IaC can deliver stable environments rapidly and at scale. They avoid manual configuration of environments and enforce consistency by representing the desired state of their environments via code. Infrastructure deployments with IaC are repeatable and prevent runtime issues caused by configuration drift or missing dependencies. DevOps teams can work together with a unified set of practices and tools to deliver applications and their supporting infrastructure rapidly, reliably, and at scale."


2
Application code repo (SCC)
Code Push
3
Build, CI, integrate, test
Jenkins
ANSIBLE
Azure DevOps
Binary Artifact
4
CD, deploy
Jenkins
ANSIBLE
Azure DevOps
Release
5
Run
Microsoft Azure
Code Push
1
Inner loop
Code, run, debug
Dev Environment
App/Environment Configuration

PART II

- Basics of Docker
- What is Kubernetes?
- Kubernetes Architecture
- Introduction to YAML

DOCKER



The Docker daemon

The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client

The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.

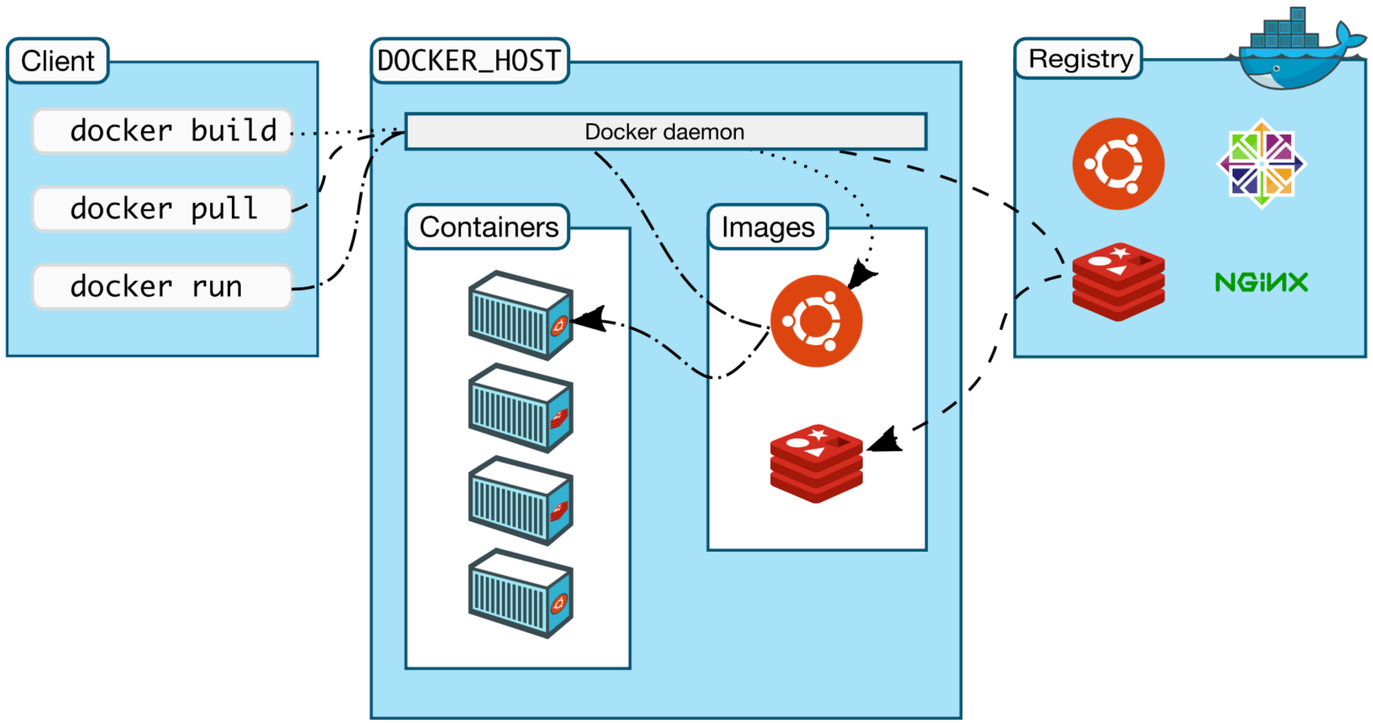
Docker Desktop

Docker Desktop is an easy-to-install application for your Mac, Windows or Linux environment that enables you to build and share containerized applications and microservices. Docker Desktop includes the Docker daemon (dockerd), the Docker client (docker), Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop.

Docker registries

A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

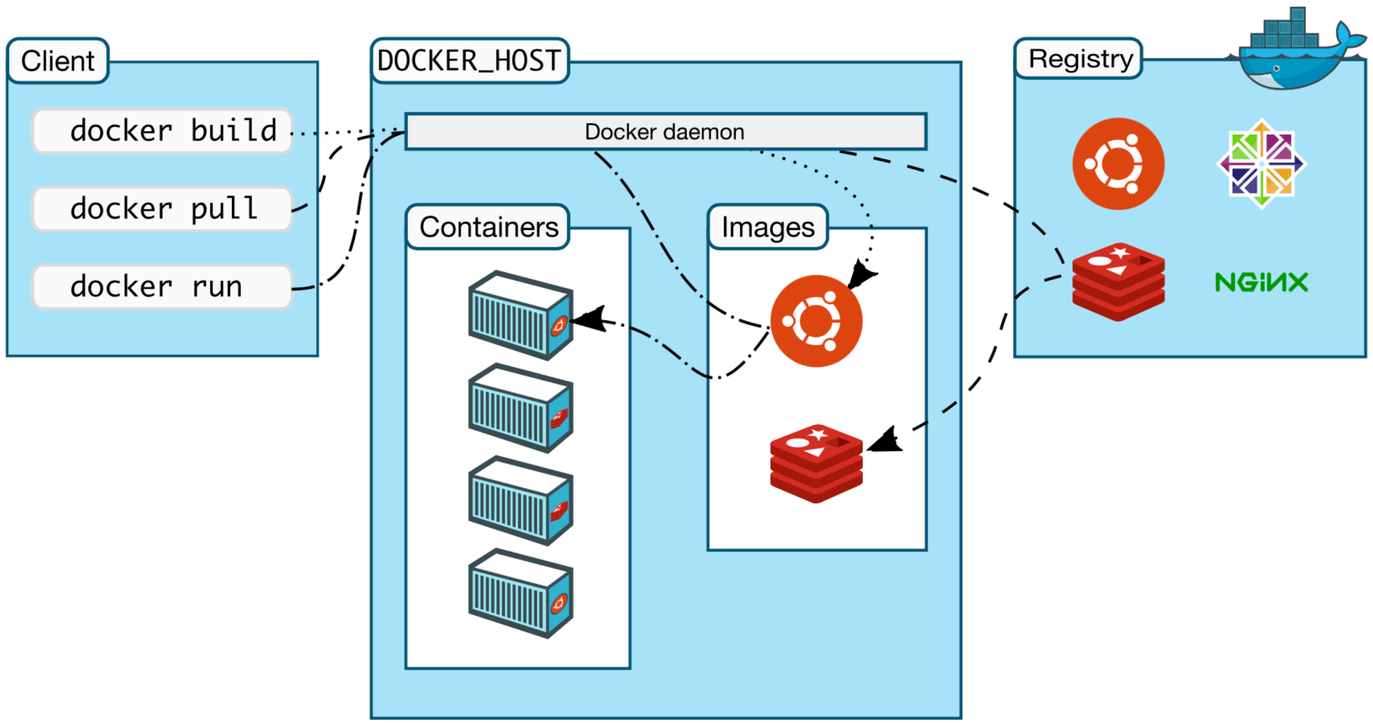
When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.



DOCKER OBJECTS

Images

- An image is a **read-only template** with **instructions** for **creating a Docker container**.
- Often, an image is **based on** another image, with some additional customization.
- You might **create your own images** or you might only use those **created by others and published in a registry**.
- To build your own image, you create a **Dockerfile** with a simple syntax for defining the steps needed to create the image and run it.
- Each instruction in a **Dockerfile creates a layer in the image**. When you change the Dockerfile and rebuild the image, only those layers which have **changed are rebuilt**.
- This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.



DOCKER OBJECTS

Containers

- A container is a **runnable instance of an image**.
- You can create, start, stop, move, or delete a container using the **Docker API or CLI**.
- You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- By default, a container is r**elatively well isolated from other containers and its host machine**.
- You can control how isolated a container's network, storage, or other underlying subsystems are from other containers or from the host machine.
- A **container is defined by its image as well as any configuration options you provide** to it when you create or start it.
- When a container is removed, **any changes to its state that are not stored in persistent storage disappear.**

ANATOMY OF A DOCKER RUN COMMAND!

$ docker run -i -t ubuntu /bin/bash

When you run this command, the following happens (assuming you are using the default registry configuration):

1. If you do not have the ubuntu image locally, Docker pulls it from your configured registry, as though you had run docker pull ubuntu manually.
2. Docker creates a new container, as though you had run a docker container create command manually.
3. Docker allocates a read-write filesystem to the container, as its final layer. This allows a running container to create or modify files and directories in its local filesystem.
4. Docker creates a network interface to connect the container to the default network, since you did not specify any networking options. This includes assigning an IP address to the container. By default, containers can connect to external networks using the host machine's network connection.
5. Docker starts the container and executes /bin/bash. Because the container is running interactively and attached to your terminal (due to the -i and -t flags), you can provide input using your keyboard while the output is logged to your terminal.
6. When you type exit to terminate the /bin/bash command, the container stops but is not removed. You can start it again or remove it.

DOCKERFILE COMMANDS

- **FROM -** Defines a base image, it can be pulled from docker hub (for example- if we want to create a javascript application with node as backend then we need to have node as a base image, so it can run node application.)
- **RUN -** Executes command in a new image layer (we can have multiple run commands)
- **CMD -** Command to be executed when running a container(It is asked to have one CMD command, If a Dockerfile has multiple CMDs, it only applies the instructions from the last one.
- **EXPOSE -** Documents which ports are exposed (It is only used for documentation)
- **ENV -** Sets environment variables inside the image
- **COPY -** It is used to copy your local files/directories to Docker Container.
- **ADD -** It is more feature-rich version of the COPY instruction. COPY is preferred over ADD. Major difference b/w ADD and COPY is that ADD allows you to copy from URL that is the source can be URL but in COPY it can only have local ones.
- **ENTRYPOINT -** Define a container's executable (You cannot override and ENTRYPOINT when starting a container unless you add the --entrypoint flag.)
- **VOLUME -** It defines which directory in an image should be treated as a volume. The volume will be given a random name which can be found using docker inspect command.
- **WORKDIR -** Defines the working directory for subsequent instructions in the Dockerfile (Important point to remember that it doesn't create a new intermediate layer in Image)

DOCKERFILE COMMANDS

All three instructions (RUN, CMD and ENTRYPOINT) can be specified in **shell form or exec form.**

Shell form:

RUN apt-get install python3
CMD echo "Hello world"
ENTRYPOINT echo "Hello world"

Exec form:

This is the preferred form for CMD and ENTRYPOINT instructions. ["executable", "param1", "param2", ...]

DOCKERFILE COMMANDS

RUN - RUN instruction allows you to install your application and packages required for it. It executes any commands on top of the current image and creates a new layer by committing the results. Often you will find multiple RUN instructions in a Dockerfile.

RUN apt-get install python

CMD - CMD instruction allows you to set a default command, which will be executed only when you run container without specifying a command. If Docker container runs with a command, the default command will be ignored. If Dockerfile has more than one CMD instruction, all but last CMD instructions are ignored.

CMD "echo" "Hello World!"

ENTRYPOINT - ENTRYPOINT instruction allows you to configure a container that will run as an executable. It looks similar to CMD, because it also allows you to specify a command with parameters. The difference is **ENTRYPOINT command and parameters are not ignored when Docker container runs with command line parameters**.

Prefer ENTRYPOINT to CMD when building **executable Docker image and you need a command always to be executed**. Additionally use **CMD if you need to provide extra default arguments that could be overwritten from command line when docker container runs**.

https://dev.to/aws-builders/understanding-the-dockerfile-format-3cc6

Docker Cheat Sheet

Build

Build an image from the Dockerfile in the current directory and tag the image
```
docker build -t myimage:1.0 .
```

List all images that are locally stored with the Docker Engine
```
docker image ls
```

Delete an image from the local image store
```
docker image rm alpine:3.4
```

Share

Pull an image from a registry
```
docker pull myimage:1.0
```

Retag a local image with a new image name and tag
```
docker tag myimage:1.0 myrepo/myimage:2.0
```

Push an image to a registry
```
docker push myrepo/myimage:2.0
```

Run

Run a container from the Alpine version 3.9 image, name the running container "web" and expose port 5000 externally, mapped to port 80 inside the container.
```
docker container run --name web -p 5000:80 alpine:3.9
```

Stop a running container through SIGTERM
```
docker container stop web
```

Stop a running container through SIGKILL
```
docker container kill web
```

List the networks
```
docker network ls
```

List the running containers (add `--all` to include stopped containers)
```
docker container ls
```

Delete all running and stopped containers
```
docker container rm -f $(docker ps -aq)
```

Print the last 100 lines of a container's logs
```
docker container logs --tail 100 web
```

Docker Management

All commands below are called as options to the base **`docker`** command. **`Run docker <command> --help`** for more information on a particular command.

`app*`	***Docker Application***
`assemble*`	***Framework-aware builds (Docker Enterprise)***
`builder`	***Manage builds***
`cluster`	***Manage Docker clusters (Docker Enterprise)***
`config`	***Manage Docker configs***
`context`	***Manage contexts***
`engine`	***Manage the docker Engine***
`image`	***Manage images***
`network`	***Manage networks***
`node`	***Manage Swarm nodes***
`plugin`	***Manage plugins***
`registry*`	***Manage Docker registries***
`secret`	***Manage Docker secrets***
`service`	***Manage services***
`stack`	***Manage Docker stacks***
`swarm`	***Manage swarm***
`system`	***Manage Docker***
`template*`	***Quickly scaffold services (Docker Enterprise)***
`trust`	***Manage trust on Docker images***
`volume`	***Manage volumes***

*Experimental in Docker Enterprise 3.0.

www.docker.com

HANDS ON!

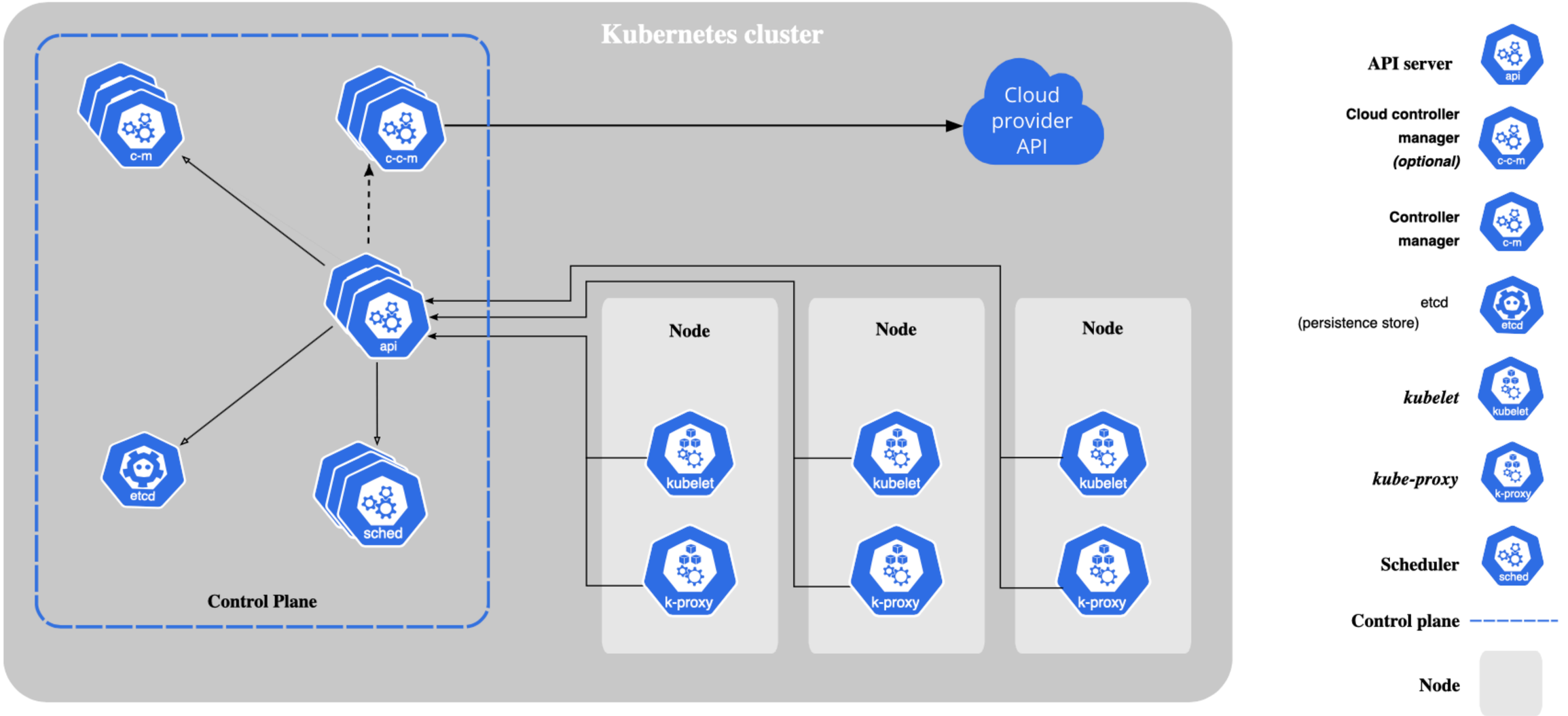
WHAT IS KUBERNETES?

- So we learned about containers and we now have our application packaged into a docker container. But what next? How do you run it in production? What if your application relies on other containers such as database or messaging services or other backend services? What if the number of users increase and you need to scale your application? You would also like to scale down when the load decreases.
- Enter Kubernetes!
- Kubernetes also known as K8s was built by Google based on their experience running containers in production. It is now an open-source project and is arguably one of the best and most popular container orchestration technologies out there.
- **The whole process of automatically deploying and managing containers is known as Container Orchestration.**

THE KUBERNETES ADVANTAGE!

There are various advantages of container orchestration.

- Your application is now highly available as hardware failures do not bring your application down because you have multiple instances of yourapplication running on different nodes.
- The user traffic is load balanced across the various containers. When demand increases, deploy more instances of the application seamlessly and within a matter of second and we have the ability to do that at a service level. When we run out of hardware resources, scale the number of nodes up/down without having to take down the application.
- And do all of these easily with a set of declarative object configuration files.



KUBERNETES ARCHITECTURE

KUBERNETES ARCHITECTURE

- When you deploy Kubernetes, you **get a cluster.**

- A Kubernetes cluster consists of a set of **worker machines, called nodes**, that run containerized applications. **Every cluster has at least one worker node.**The worker node(s) host the Pods that are the components of the application workload.

- The **control plane manages the worker nodes and the Pods in the cluster**. **In production environments**, the **control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability**.

KUBERNETES ARCHITECTURE - CONTROL PLANE

The control plane's components **make global decisions about the cluster** (for example, scheduling), as well as **detecting and responding to cluster events** (for example, starting up a new pod when a deployment's replicas field is unsatisfied).

Control plane components **can be run on any machine in the cluster**. However, for simplicity, set up scripts typically start all control plane components on the same machine, and do not run user containers on this machine.

- Kube-apiserver
- Etcd
- Kube-scheduler
- Kube-controller-manager
- Cloud-controller-manager

KUBERNETES ARCHITECTURE - CONTROL PLANE - KUBE-APISERVER

The API server is a component of the Kubernetes control plane that **exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane.**

The main implementation of a Kubernetes API server is kube-apiserver. kube-apiserver is designed to scale horizontally—that is, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

KUBERNETES ARCHITECTURE - CONTROL PLANE - ETCD

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data.

KUBERNETES ARCHITECTURE - CONTROL PLANE - KUBE-SCHEDULER

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

Factors taken into account for scheduling decisions include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.

KUBERNETES ARCHITECTURE - CONTROL PLANE - KUBE-CONTROLLER-MANAGER

Control plane component that runs controller processes.

Logically, each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.

Some types of these controllers are:

- **Node controller:** Responsible for noticing and responding when nodes go down.
- **Job controller:** Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
- **Endpoints controller:** Populates the Endpoints object (that is, joins Services & Pods).
- **Service Account & Token controllers:** Create default accounts and API access tokens for new namespaces.

KUBERNETES ARCHITECTURE - CONTROL PLANE - CLOUD-CONTROLLER-MANAGER

A Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster.

The cloud-controller-manager only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.

As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that you run as a single process. You can scale horizontally (run more than one copy) to improve performance or to help tolerate failures.

The following controllers can have cloud provider dependencies:

- **Node controller:** For checking the cloud provider to determine if a node has been deleted in the cloud after it stops responding
- **Route controller:** For setting up routes in the underlying cloud infrastructure
- **Service controller:** For creating, updating and deleting cloud provider load balancers

KUBERNETES ARCHITECTURE - NODES

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

- Kubelet
- Kube-proxy
- Container Runtime

KUBERNETES ARCHITECTURE - NODES

kubelet

An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

kube-proxy

kube-proxy is a network proxy that runs on each node in your cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your cluster. kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

Container runtime

The container runtime is the software that is responsible for running containers.Kubernetes supports container runtimes such as containerd, CRI-O, and any other implementation of the Kubernetes CRI (Container Runtime Interface).

INTRODUCTION TO YAML

- https://developer.ibm.com/tutorials/yaml-basics-and-usage-in-kubernetes/

PART III

- Pods
- Replica Sets
- Deployment
- Services
- State Persistence

PODS

Pods are **the smallest deployable units** of computing that you can create and manage in Kubernetes.

A *Pod* (as in a pod of whales or pea pod) is a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers. A Pod's contents are always co-located and co-scheduled, and run in a shared context. A Pod models an application-specific "logical host": it contains one or more application containers which are relatively tightly coupled. In non-cloud contexts, applications executed on the same physical or virtual machine are analogous to cloud applications executed on the same logical host.

As well as application containers, a Pod can contain **init containers** that run during Pod startup.

WHAT IS A POD?

The shared context of a Pod is a set of Linux namespaces, cgroups, and potentially other facets of isolation - the same things that isolate a Docker container. Within a Pod's context, the individual applications may have further sub-isolations applied.

In terms of Docker concepts, a Pod is similar to a group of Docker containers with shared namespaces and shared filesystem volumes.

https://kubernetes.io/docs/concepts/workloads/pods/

REPLICA SETS

A ReplicaSet's purpose is to maintain a stable set of replica Pods running at any given time. As such, it is often used to guarantee the availability of a specified number of identical Pods.

https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/

DEPLOYMENTS

A *Deployment* provides declarative updates for Pods and ReplicaSets.

You describe a *desired state* in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. You can define Deployments to create new ReplicaSets, or to remove existing Deployments and adopt all their resources with new Deployments.

https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

SERVICES

- An abstract way to expose an application running on a set of Pods as a network service.
- With Kubernetes you don't need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.
- Kubernetes Pods are created and destroyed to match the desired state of your cluster. Pods are nonpermanent resources. If you use a Deployment to run your app, it can create and destroy Pods dynamically.
- Each Pod gets its own IP address, however in a Deployment, the set of Pods running in one moment in time could be different from the set of Pods running that application a moment later.
- This leads to a problem: if some set of Pods (call them "backends") provides functionality to other Pods (call them "frontends") inside your cluster, how do the frontends find out and keep track of which IP address to connect to, so that the frontend can use the backend part of the workload?

https://kubernetes.io/docs/concepts/services-networking/service/

STATE PERSISTENCE

Managing storage is a distinct problem from managing compute instances. The PersistentVolume subsystem provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this, we introduce two new API resources: PersistentVolume and PersistentVolumeClaim.

A ***PersistentVolume*** **(PV)** is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes. It is a resource in the cluster just like a node is a cluster resource. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV. This API object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.

A ***PersistentVolumeClaim*** **(PVC)** is a request for storage by a user. It is similar to a Pod. Pods consume node resources and PVCs consume PV resources. Pods can request specific levels of resources (CPU and Memory). Claims can request specific size and access modes (e.g., they can be mounted ReadWriteOnce, ReadOnlyMany or ReadWriteMany, see AccessModes).

https://kubernetes.io/docs/concepts/storage/persistent-volumes/

THANK YOU :)