

# Intro to DevOps and Beyond

Ravindu Nirmal Fernando

# About Me



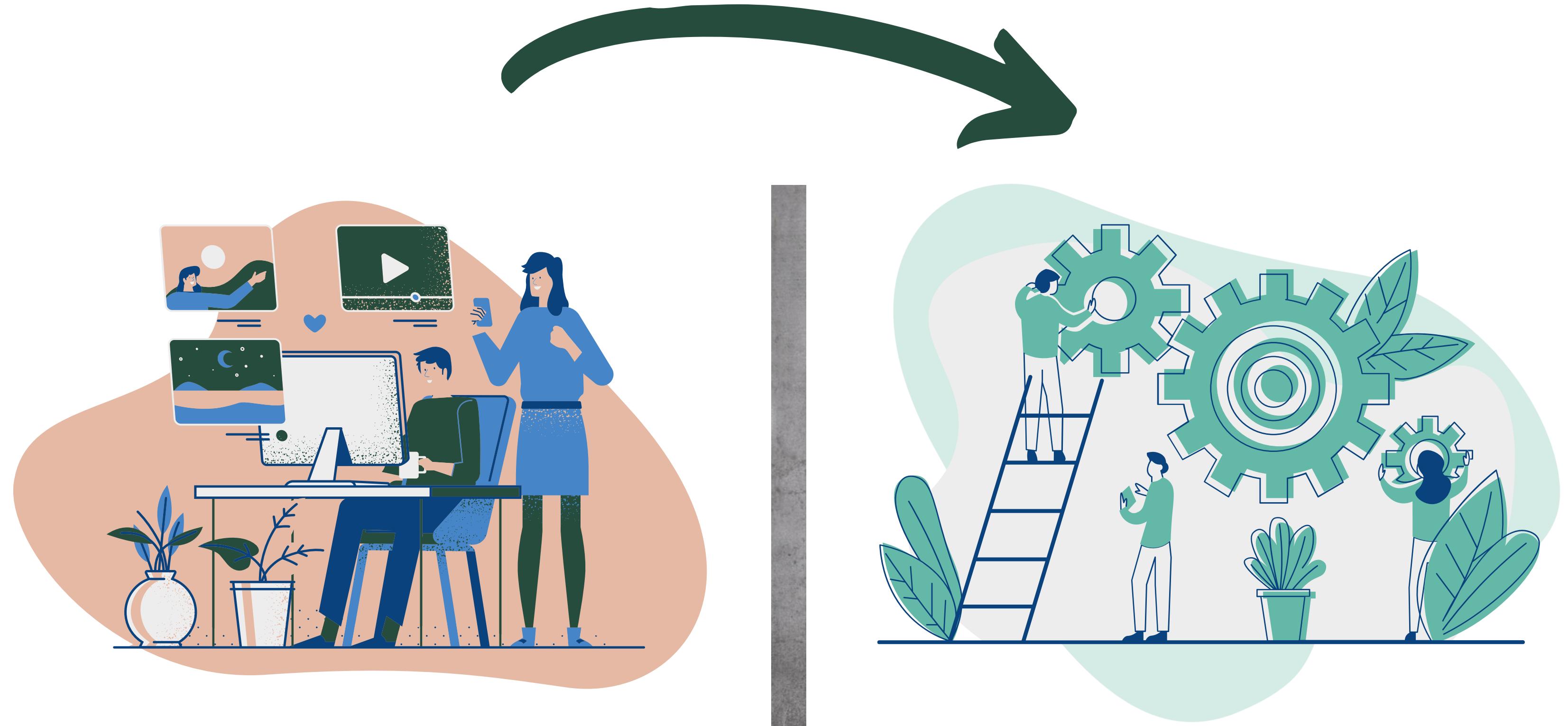
**Ravindu Nirmal Fernando**

<https://ravindunfernando.com>

- STL - DevOps @ Sysco LABS - Sri Lanka
- MSc in Computer Science specialized in Cloud Computing (UOM)
- AWS Certified Solutions Architect - Professional
- Certified Kubernetes Administrator (CKA)
- AWS Community Builder



# The Era before DevOps



**Developers**  
**Focused on Agility**

**Operators**  
**Focused on Stability**

# "Destructive downward spiral in IT" - Gene Kim



## Act 01 – Operations teams maintaining large fragile applications

Doesn't have any visibility on the application, whether or not its working as expected



## Act 02 – The product managers

Larger, unrealistic commitments made to the outside world (client/investors) without understanding the complexities behind development and operations



## Act 03 – The Developers

Developers taking shortcuts and putting more and more fragile code on top of existing ones



## Act 04 – Dev and Ops at war

"It worked on my machine" phenomenon



**How can we  
overcome  
these issues?**

“DevOps is the combination of cultural philosophies, practices, and tools that increases an organization’s ability to deliver applications and services at high velocity”

- What is DevOps? [AWS] -

“A compound of development (Dev) and operations (Ops), DevOps is the union of people, process, and technology to continually provide value to customers.”

- What is DevOps? [Azure] -

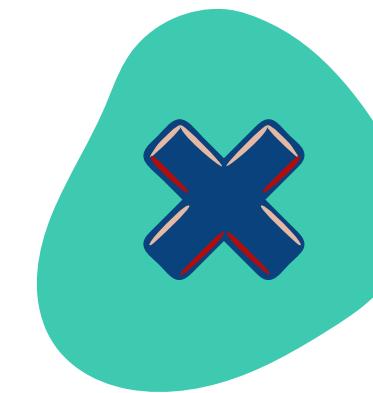
**DevOps allows evolving and improving products at a faster pace than businesses using traditional software development and infrastructure management processes. This speed allows businesses to serve their customers better and compete effectively.**

# Key Areas in DevOps



## Reduce Organizational Silos

Everyone shares the ownership of production and information is shared among everyone



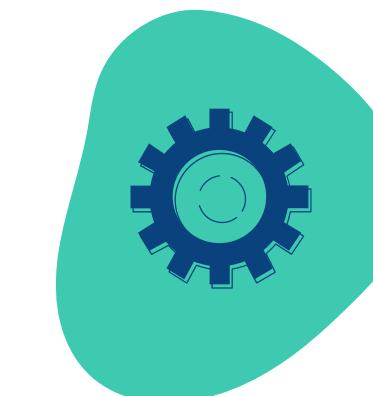
## Accept Failure as Normal

Blameless PMs/ RCA. Risk taking mindset.



## Implement Gradual Changes

Frequent deployments, frequent deterministic releases in small chunks which can be rolled back



## Leverage Tooling and Automation

Automate and reduce manual work as much as possible



## Measure Everything

Application, systems monitoring and metrics etc...



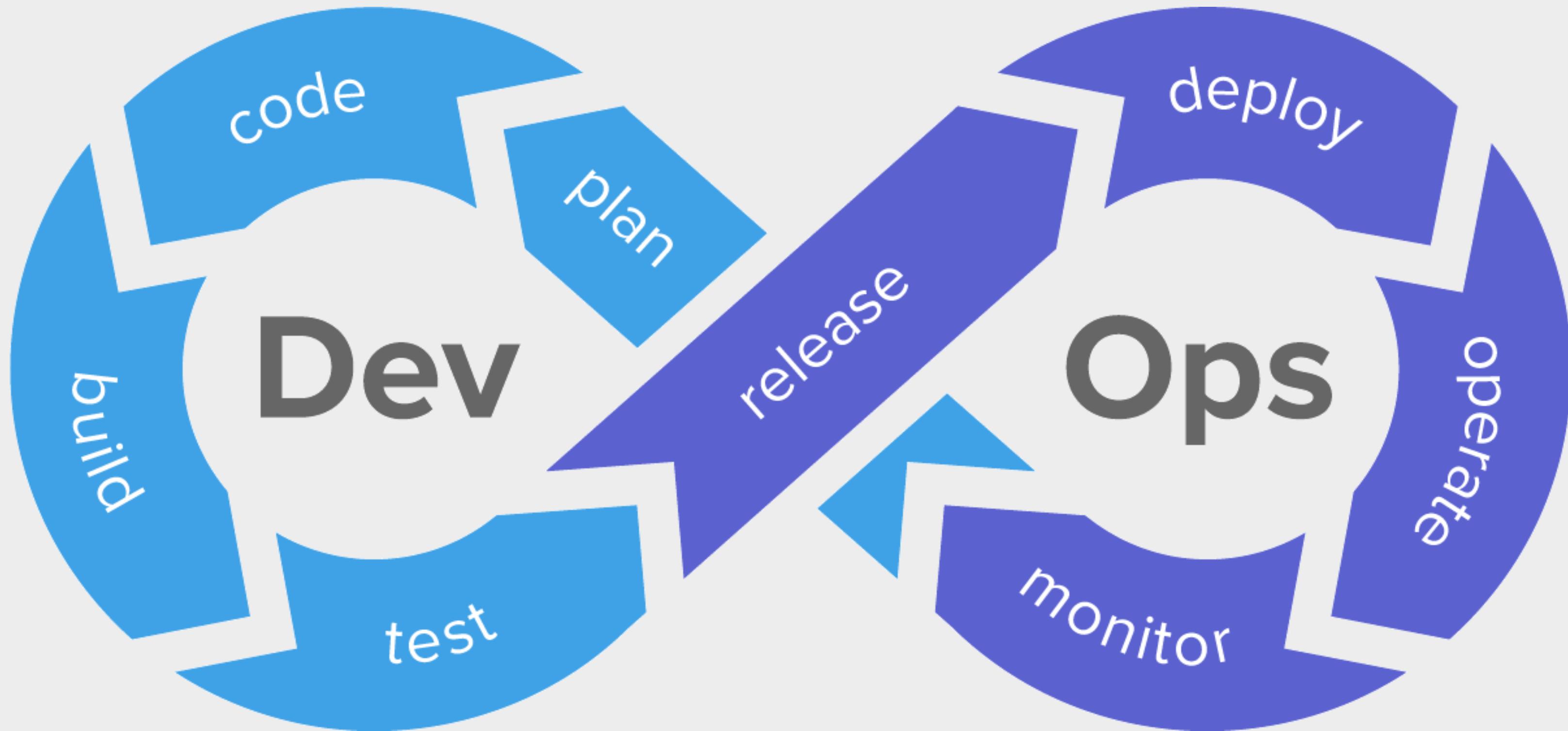
# DevOps Practices

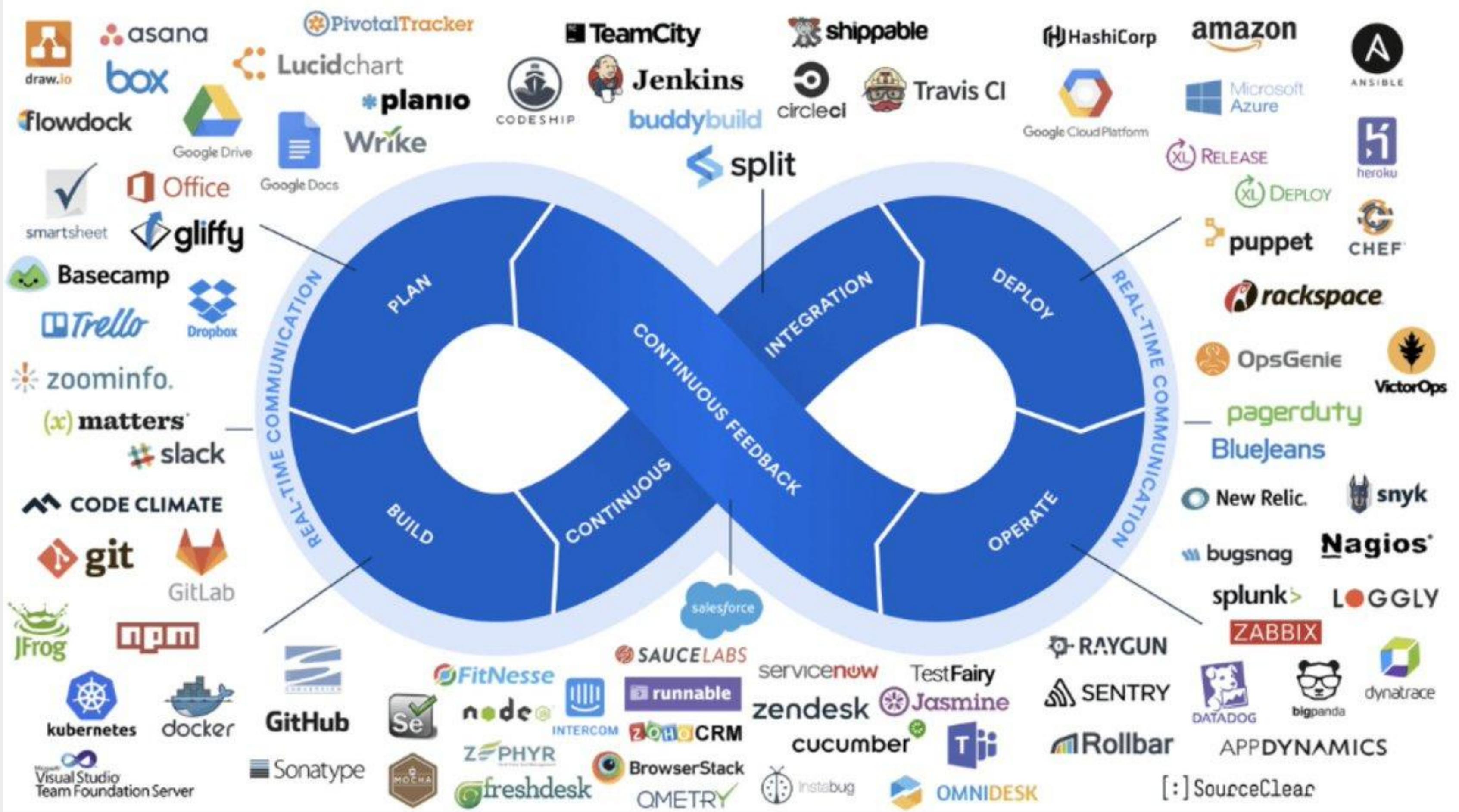
- Continuous Integration (CI) - Software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run.
- Continuous Delivery (CD) - Software development practice where code changes are automatically built, tested, and prepared for a release to production (automated code change deployment to staging/ pre-production system).
- Continuous Deployment (CD) - Every change that passes all stages of the pipeline will be deployed into production (released to customers). This practice fully automates the whole release flow without human intervention and only a failed test will prevent a new change being deployed.
- Microservices - The microservices architecture is a design approach to build a single application as a set of small services with each focusing on SRP. Each service can be created, deployed and run independently.

- Infrastructure as Code - A practice in which infrastructure is provisioned and managed using code and software development techniques, such as version control and continuous integration.
  - Configuration Management
  - Policy as Code
- GitOps - builds on the concept of IaC, incorporating the functionality of Git repositories, merge requests (MRs) and CI/CD to further unify software development and infrastructure operations. GitOps incorporates managing both infrastructure and applications as code.
- Cloud Infrastructure - Cloud provides more flexibility, scalability and toolsets for organizations to implement DevOps culture and practices. Serverless architecture in cloud brings down the efforts of DevOps teams as it eliminates server management operations.
- Continuous Monitoring, Logging and Alerting - Organizations monitor metrics and logs to see how application and infrastructure performance impacts the experience of their product's end user. Combined with real time alerts organizations can do a real time analysis on the application status.



# DevOps Tools and Technologies



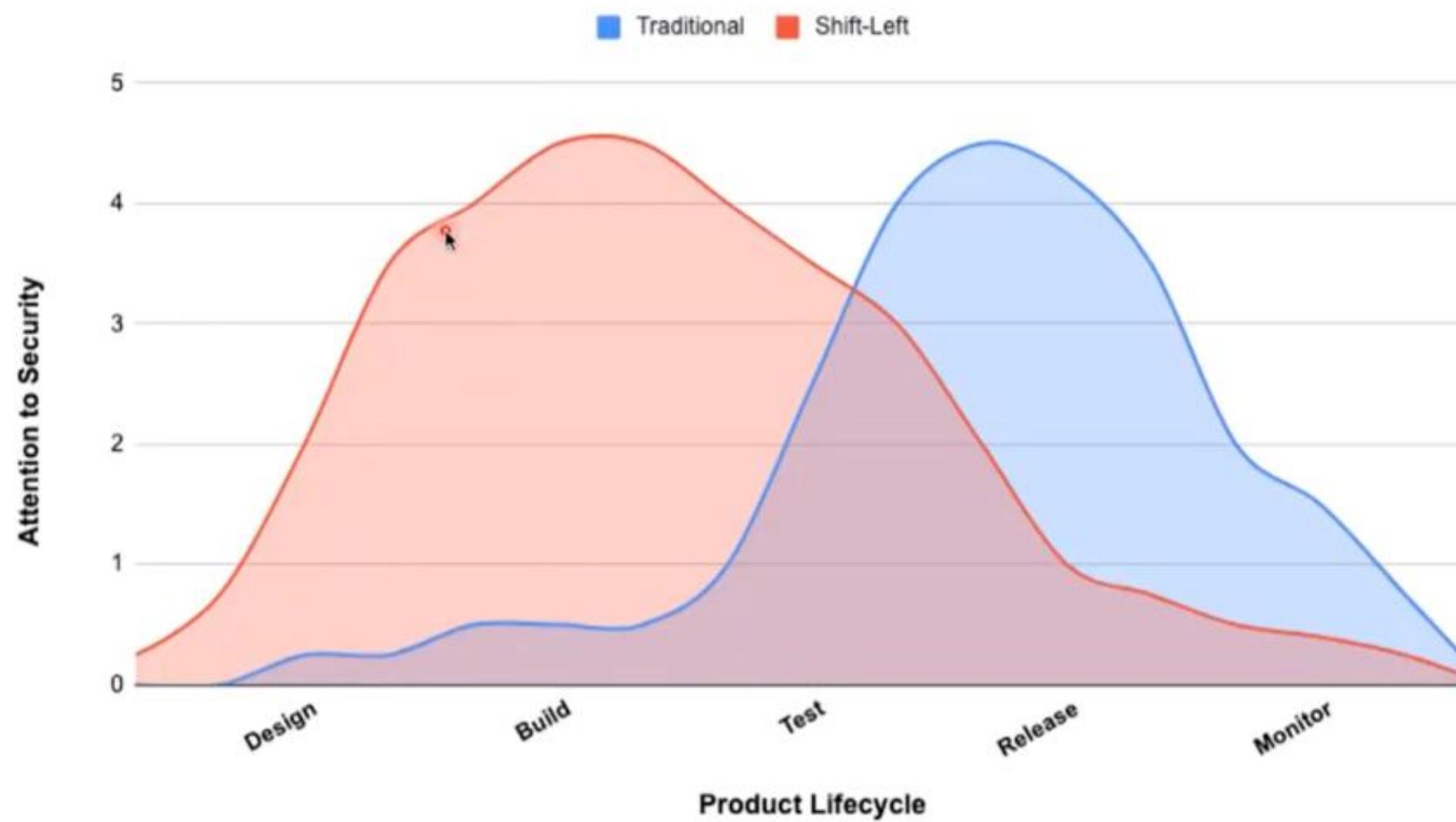




# Beyond DevOps

# DevSecOps

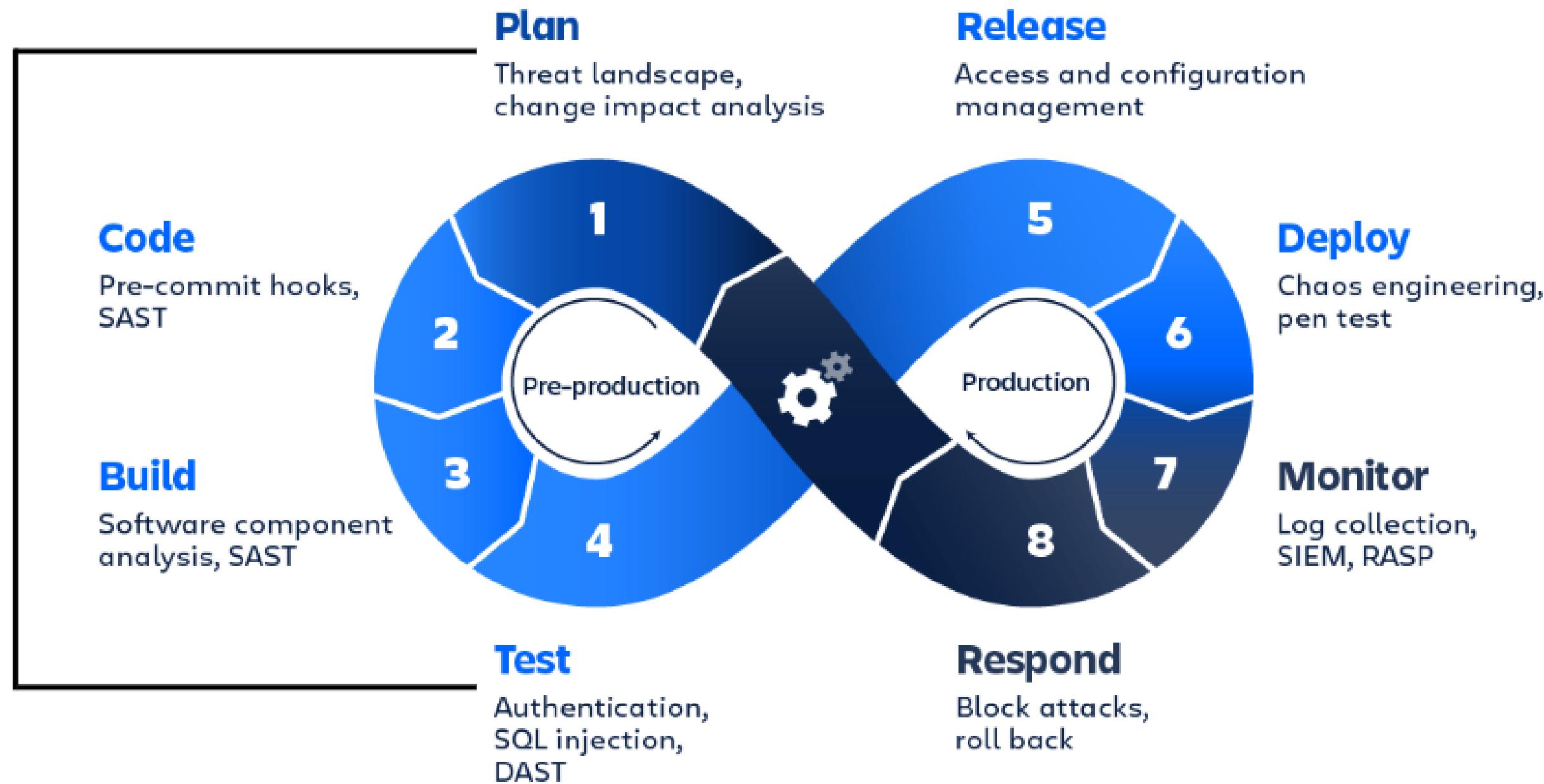
Traditional vs Shift-Left Security Model



Idea of moving Security in the early stages of the SDLC pipeline

"the practice of integrating security into a continuous integration, continuous delivery, and continuous deployment pipeline"

# DevSecOps



# SRE (Site Reliability Engineering)

- Not competing with DevOps
- Think that Class SRE implements Interface DevOps
- SRE is a part of the DevOps umbrella

## **SRE Practices**

- Identify and measure **SLIs**, define **SLOs** and agree/ commit to **SLA** for product and service
- Chaos Engineering
- Removing toil
- System designing (DR, Multi-Region, Mult-Cloud)
- Postmortems/ Root Cause Analysis
- Observability

# Platform Engineering

Before jumping to definition, let's understand the problem...

Configuration  
management

Cloud provider / runtime  
environment

Security

Database  
anonymization

**Application**

Secret management

Alerts

Infrastructure as Code

Cost insights

Security scanning

Artifact management

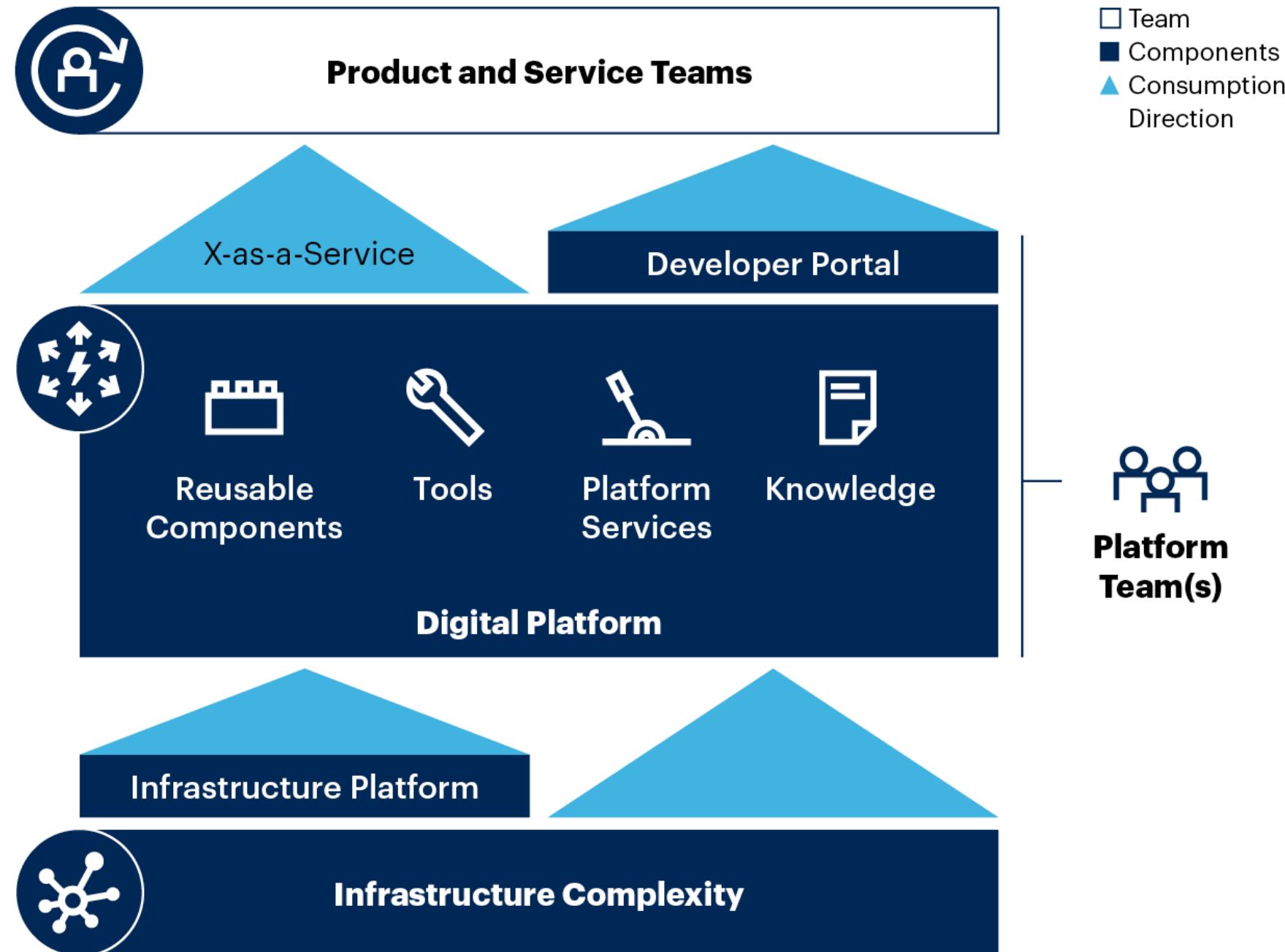
Monitoring, Logging,  
Metrics

CI / CD

Database migrations

“The composition and integration of a set of processes, tools and automation (components) to build a coherent platform with the goal of empowering developers to be able to easily build, maintain and deploy their business logic”

# Diagram of Platform Engineering



[gartner.com](https://gartner.com)

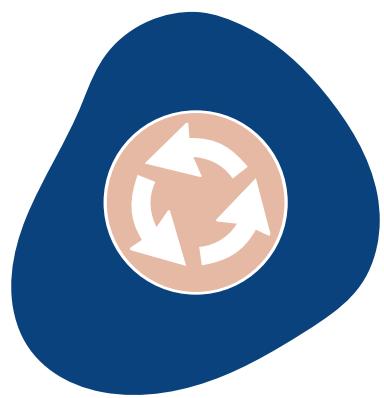
Source: Gartner  
© 2023 Gartner, Inc. and/or its affiliates. All rights reserved. CM\_GTS\_2479487

Gartner®

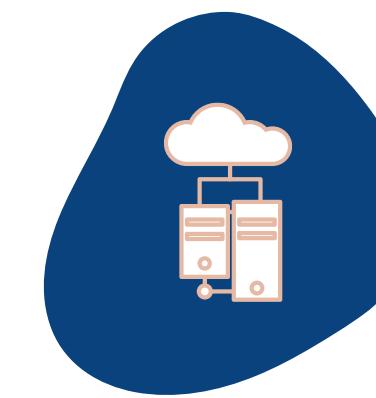


# Carrier as a DevOps Engineer

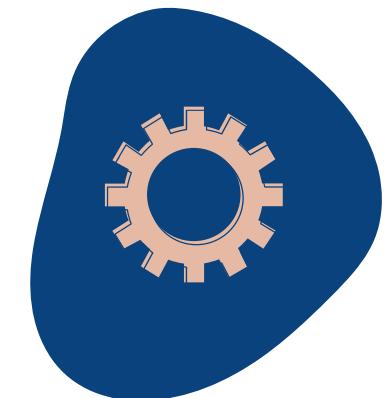
# DevOps Engineer Role



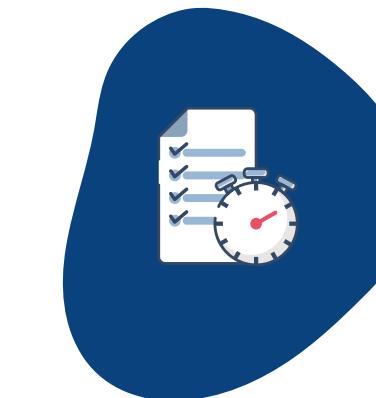
CI/ CD Management & Automation



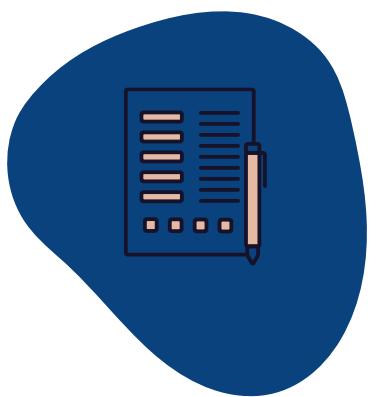
Cloud Deployment and Management



Infrastructure Management



Performance Assessment and Monitoring



Writing Specifications and Documentation



Assisting with DevOps culture adoption

## References

- <https://sre.google/sre-book/table-of-contents/>
- <https://www.gartner.com/en/articles/what-is-platform-engineering>
- [https://youtu.be/uTEL8Ff1Zvk?si=5QT\\_LrzedX-BMezt](https://youtu.be/uTEL8Ff1Zvk?si=5QT_LrzedX-BMezt)



 SCAN ME

**LinkedIn**

<https://www.linkedin.com/in/ravindufernando/>

**x (Twitter)**

@ravindunf

**Thank You!**

# Agenda

- Brief History – Infrastructure shifts over the decades
- VMs vs Containers
- What are containers and what problem does it solve?
- What is Docker?
- Deep dive into Docker Internals
- Demo

# Brief History – Infrastructure shifts over the decades

## Let's Recap

### MAJOR INFRASTRUCTURE SHIFTS



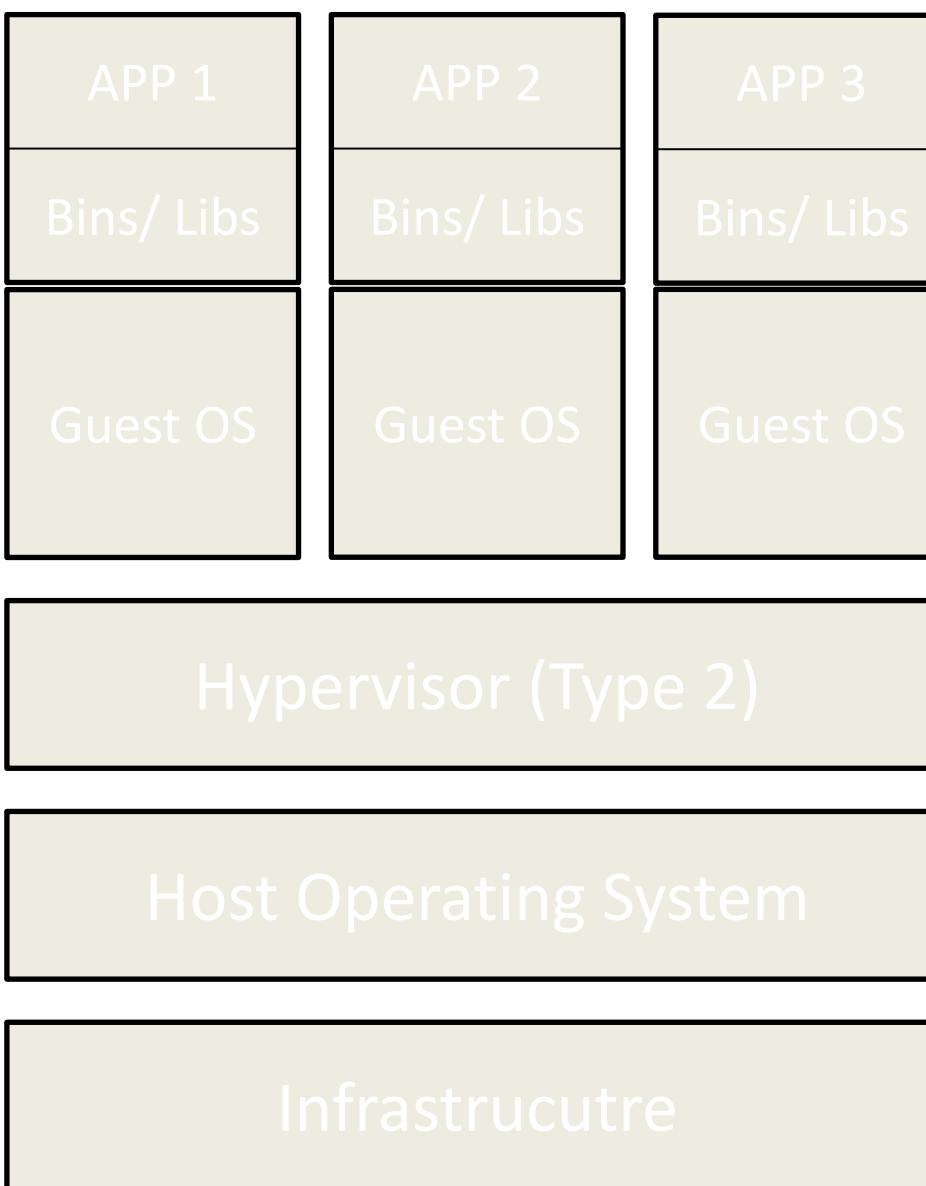
Mainframe to PC  
90's

Baremetal to Virtual  
00's

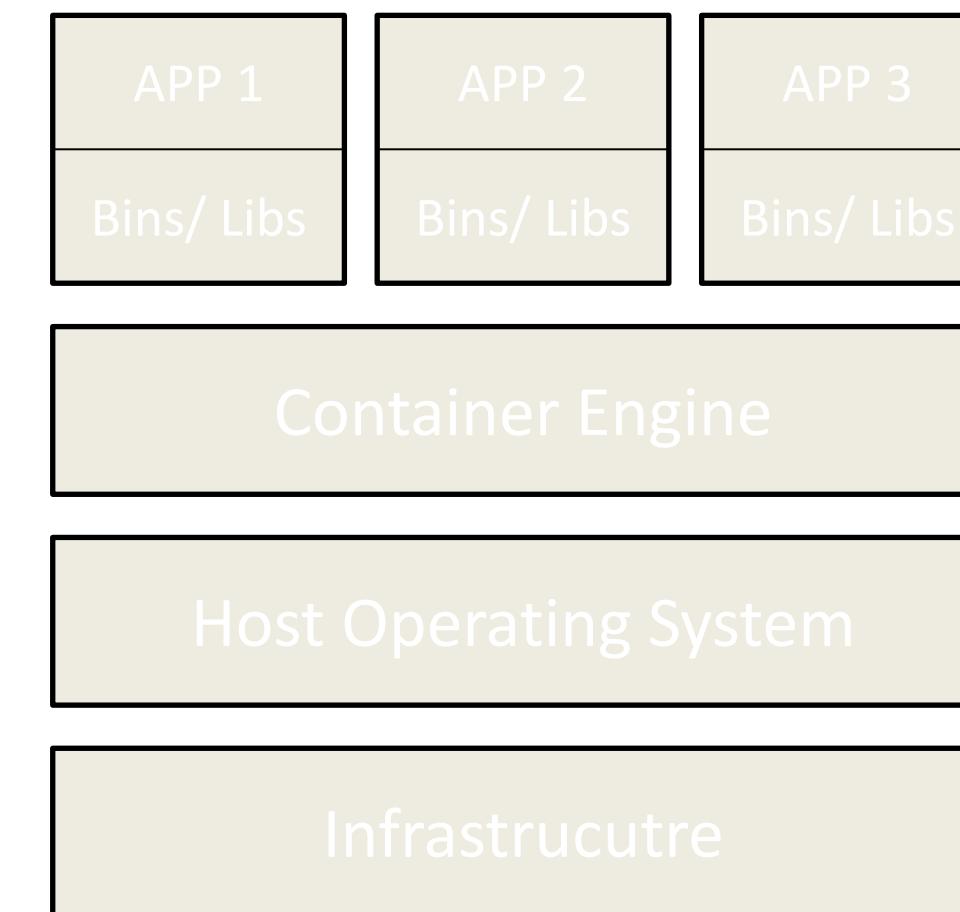
Datacenter to Cloud  
10's

Host to Container (Serverless)  
Now

# VMs vs CONtainers



-Virtual Machines-



-Containers-

What are containers and what problems does it solve?

# Matrix from hell increases the complexity

Static Website	?	?	?	?	?	?	?	?
Web Frontend	?	?	?	?	?	?	?	?
Background Workers	?	?	?	?	?	?	?	?
User DB	?	?	?	?	?	?	?	?
Analytics DB	?	?	?	?	?	?	?	?
Queue	?	?	?	?	?	?	?	?
	Desktop 	Test/QA Cluster 	Production Cluster 	Public Cloud 	Data Center 	Mainframe 	Windows Server 	Edge Device 

# Containers reduces the complexity



In summary a container is,

- Just an isolated process running on the host machine. And a restricted process.
- Will share OS and, where appropriate, bins/ libraries and limited to what resources it can access.
- It exits when the process stops.

“Containers are the next once-in-a-decade shift in IT infrastructure and process”

What is docker?

- So what's Docker? – In 2024, Docker means lot's of things, let's just clear things out.
  - Docker as a “Company”
  - Docker as a “Product”
  - Docker as a “Platform”
  - Docker as a “CLI tool”
  - Docker as a “Computer Program”

# What is Docker?

- Docker provides the ability to package and run applications within a loosely isolated environment which is a container. Simply it's a container engine (runtime + tool for managing containers and images).
- It provides tooling and a platform to manage lifecycle of your containers,
  - Develop your apps and supporting components using containers
  - Distribute and test your apps as a container
  - Ability to deploy your app as a container or an orchestrated service, in whatever environment which supports Docker installation
- It shares the same OS kernel
- It works on all major Linux Distributions and containers native to Windows Server (specific versions)

# Underlying technology in docker

- Docker is an extension of LXC's (Linux Containers) capabilities and packaged it in a way which is more developer friendly.
- It was developed in Go language and utilizes LXC, namespaces, cgroups and the linux kernel itself. Docker uses namespaces to provide the isolated workspace called a container. Each aspect of a container runs in a separate namespace and its access is limited to this namespace.

Static Website							
Web Frontend							
Background Workers							
User DB							
Analytics DB							
Queue							
	Desktop 	Test/QA Cluster 	Production Cluster 	Public Cloud 	Data Center 	Mainframe 	Windows Server 
							Edge Device 

# BASIC DOCKER COMMANDS

- Docker CLI structure,
  - Old (Still works as expected) docker <command> options
  - New – docker <command> <sub-command> (options)
- Pulling Docker Image
  - docker pull nginx
- Running a Docker Container
  - docker run -p 80:80 --name web-server nginx
- Stopping the Container
  - docker stop web-server (or container id)

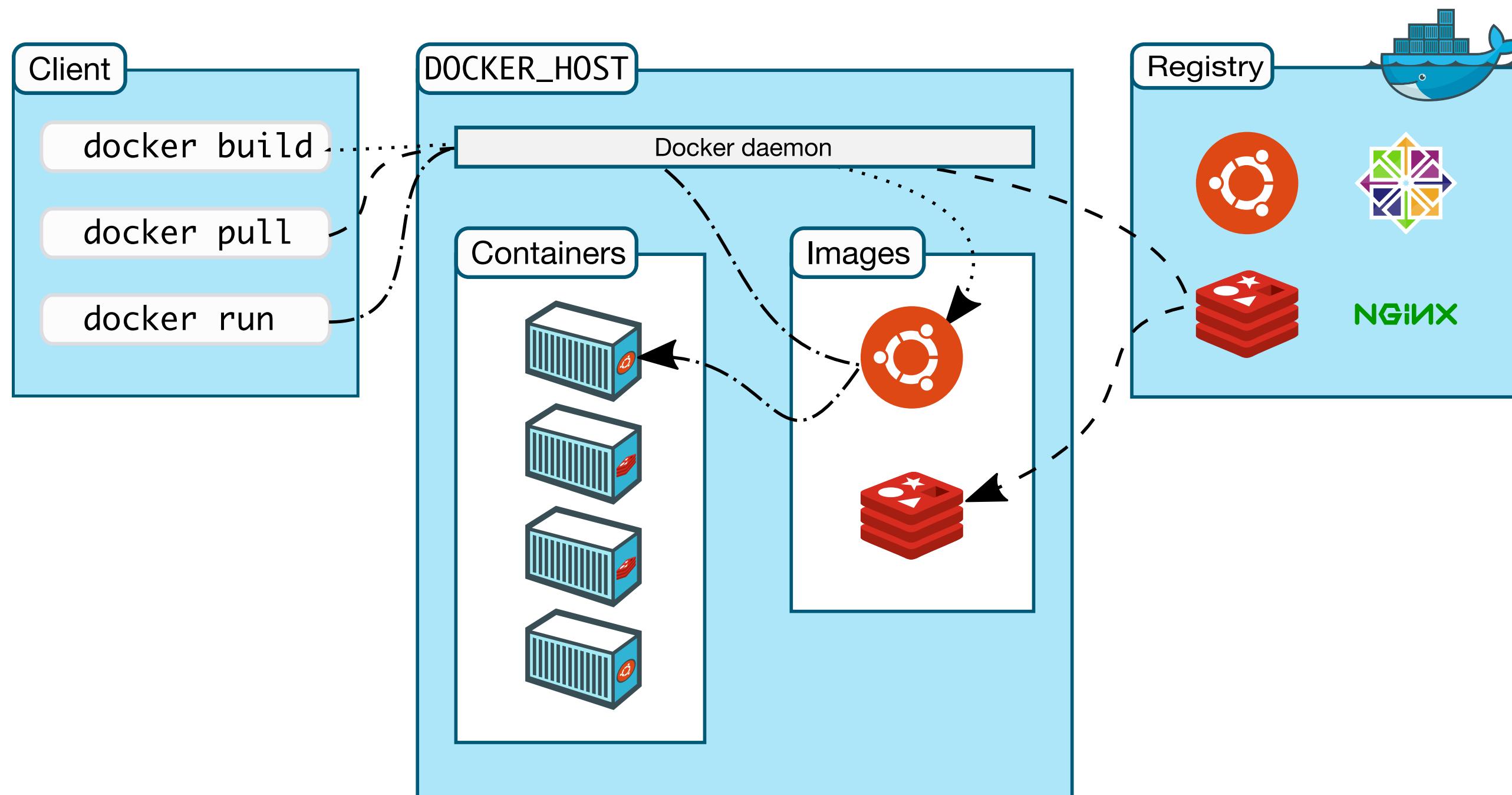
- Check what's happening in a containers,
  - docker container top web-server – Process list in 1 container
  - docker container inspect web-server – Details of one container config
  - docker container stats – Performance stats for all containers
- Getting a shell inside containers,
  - docker container run –it – Start a new container interactively
  - docker container exec –it <container\_id\_or\_name> echo “I’m inside the container” – Run additional commands in the container
- Listing, removing containers and images
  - docker images
  - docker container ls | docker ps
  - docker <object> rm <id\_or\_name>

# Are there solutions other than docker?

- Docker – Container runtime + Tool for managing containers and images
- Containerd – Container runtime only
- Podman – Tool for managing containers and images

# Deep dive into docker internals

# Docker architecture



# What happens when you run a container?

- ***docker run -p 80:80 nginx | docker container run -p 80:80 nginx***
  1. Looks for that particular image locally in image cache, if its not found pulls it from the configured registry (image repository). Downloads the latest version by default (nginx:latest)
  2. Creates a new container based on that image and prepares to start
  3. Docker allocates read write filesystem to the container, as its final layer. This allows running container to modify files and directories in its local filesystem.
  4. Gives it a virtual IP on a private network inside docker engine
  5. Opens up port 80 on host and forwards to port 80 in container.
  6. Starts container by using the CMD in the image Dockerfile.

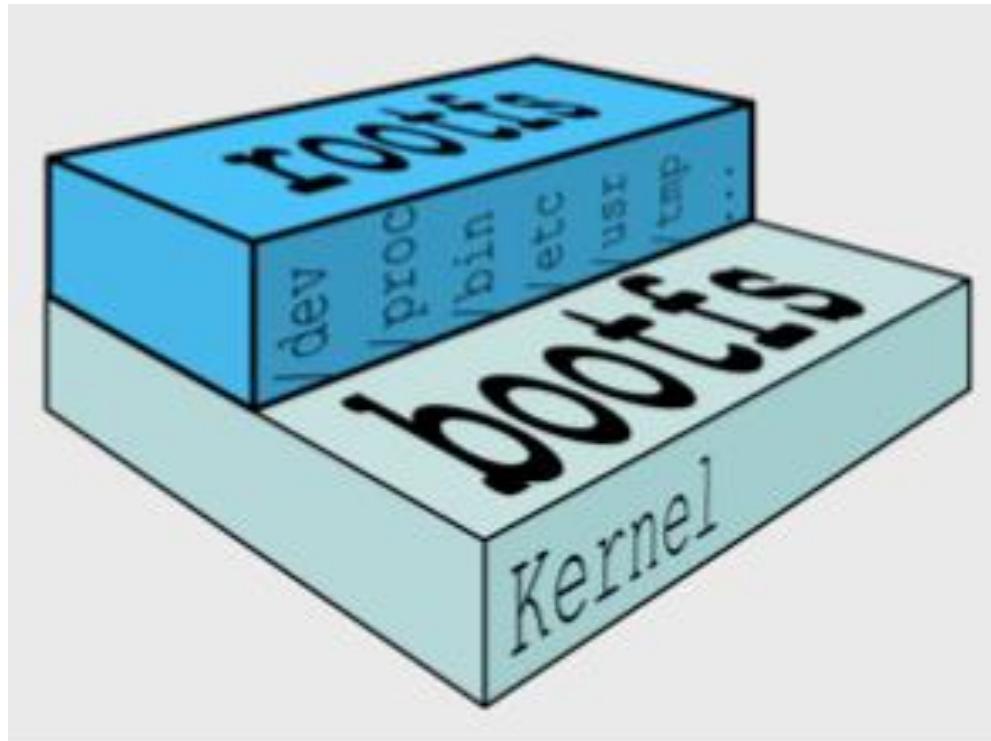
# Docker Objects

- Docker Images
  - A read-only template with instructions/ metadata for creating a Docker container.
  - Can create your own image or use images created and published in a registry by others.
  - Dockerfile can be used to define steps required to create and run the image.
  - Each instruction in Dockerfile creates a layer in the image, only those layers which changes each time are rebuilt – What makes images so lightweight, small and fast.

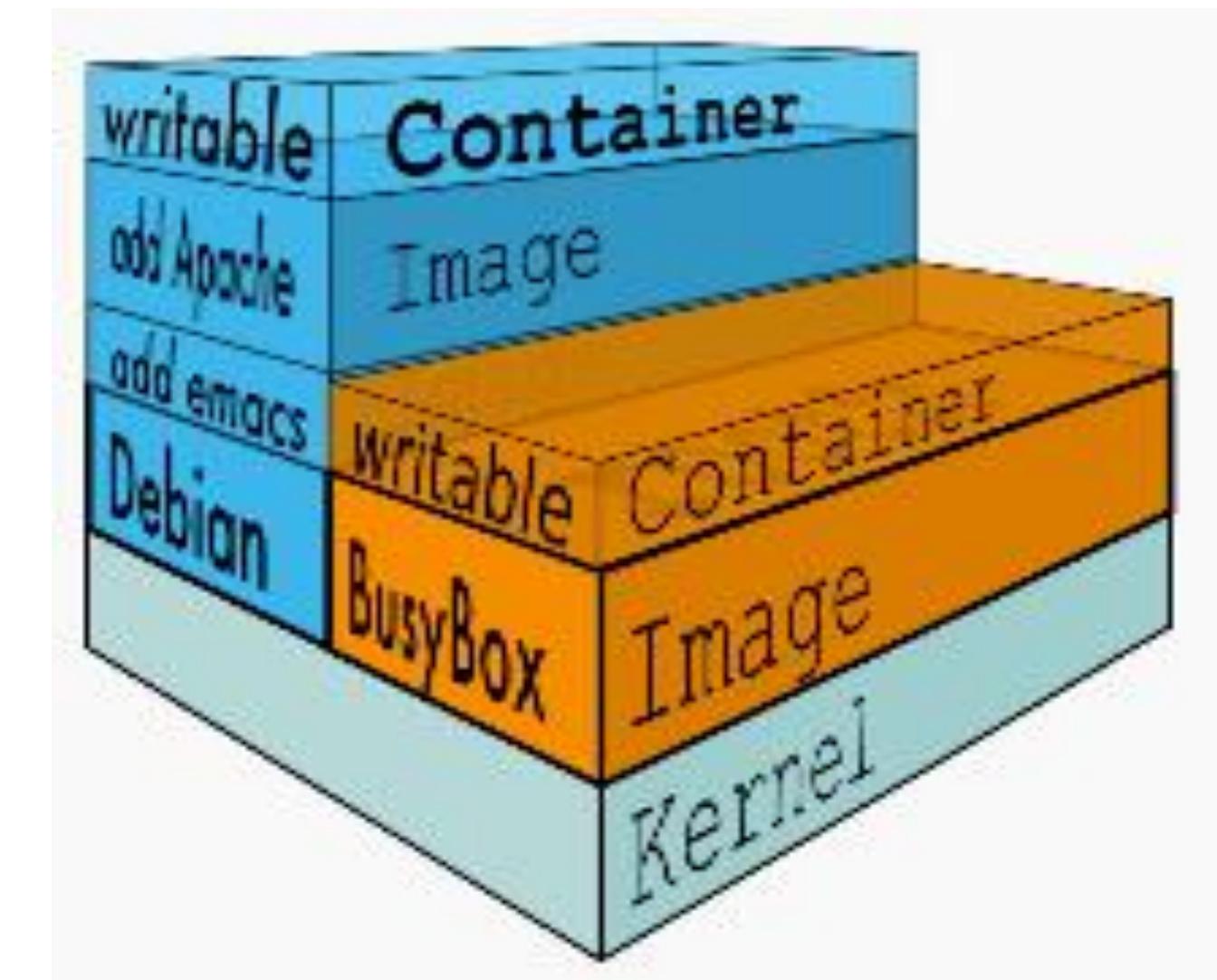
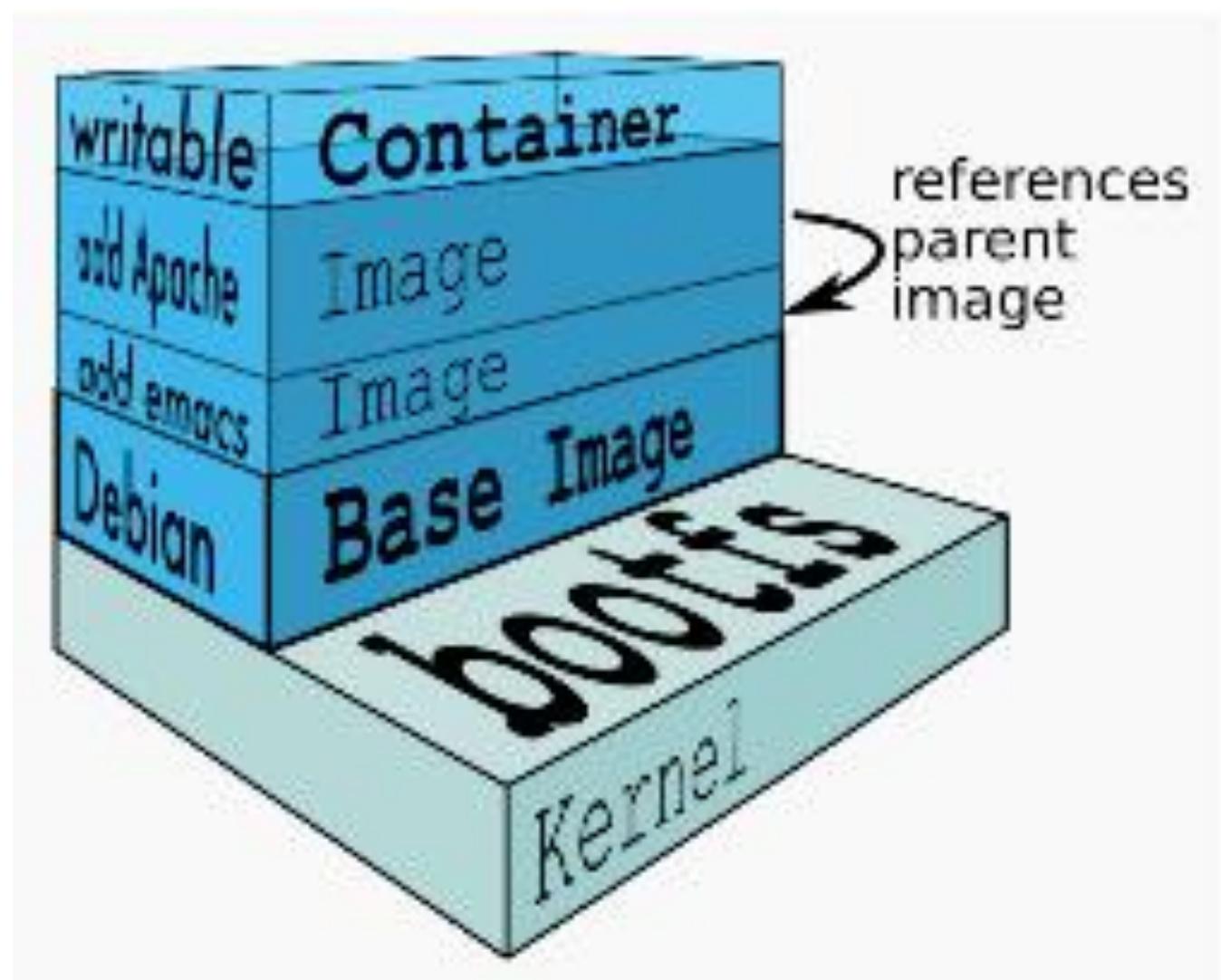
- Docker Containers
  - Runnable instance of an image.
  - Can create, start, stop, move, or delete a container using the Docker API or CLI.
  - Can connect it to one or more networks, attach storage to it, or even create a new image based on its current status.
  - A container is defined by its image as well as any config options provided to it when you create or start it. Note that when the container is removed any data associated with it will be deleted unless those are not stored in a persistent storage.

# Understanding Docker images/ containers internals

- Docker Filesystem
  - Boot file system (bootfs) – Contains the bootloader and the kernel. User never touches this.
  - Root file system (rootfs) – Includes the typical directory structure we associate with Unix-like OS.

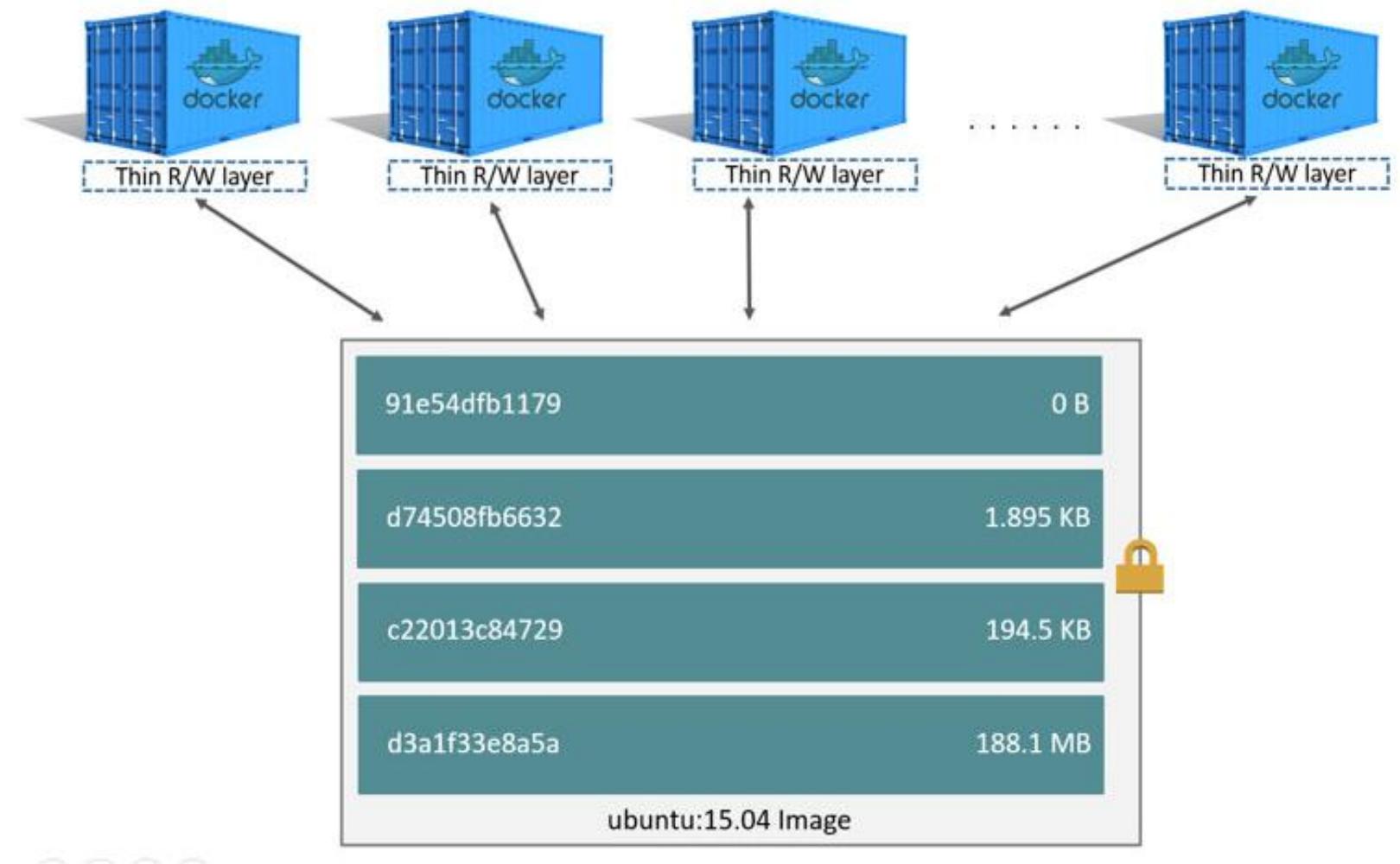


- In traditional Linux boot, kernel first mounts the rootfs as read-only, checks its integrity, and then switches the rootfs volume to read-write mode.
- Docker mounts the rootfs and **instead of changing the file system to read-write mode, it then takes advantage of union mounts service to add a read-write filesystem over the read-only file system.**
- In Docker terminology, a read-only layer is called an image. An image never changes and is fixed.
- Each image depend on one more image which creates the layer beneath it. The lower image is the parent of the upper image. Image without a parent is a base image.
- When you run a container, Docker fetches the image and its Parent Image, and repeats the process until it reaches the Base Image. Then the Union File System adds a read-write layer on top.
- That read-write layer, plus the information about its Parent Image and some additional information like its unique id, networking configuration, and resource limits is called a **container**



- A container can have two states, it may be running or exited.
- When a container is exited the state of the file system and its exit value is saved.
- You can start, stop, and restart a container. The processes of restarting a container from scratch will preserve its file system just as it was when the container was stopped. But the memory state of the container is not preserved.
- You can also remove the container permanently.
- A container can also be promoted directly into an image using the docker commit command. Once a container is committed as an image, you can use it to create other images on top of it.
  - `docker commit <container-id> <image-name:tag>`

- Based from the UFS, Docker uses a strategy called Copy on Write to improve the efficiency by minimizing I/O and the size of each subsequent layers,
  - If a file or directory exists in a lower layer within the image, and another layer (including the writable layer) needs read access to it, it just uses the existing file.
  - The first time another layer needs to modify the file (when building the image or running the container), the file is copied into that layer and modified.



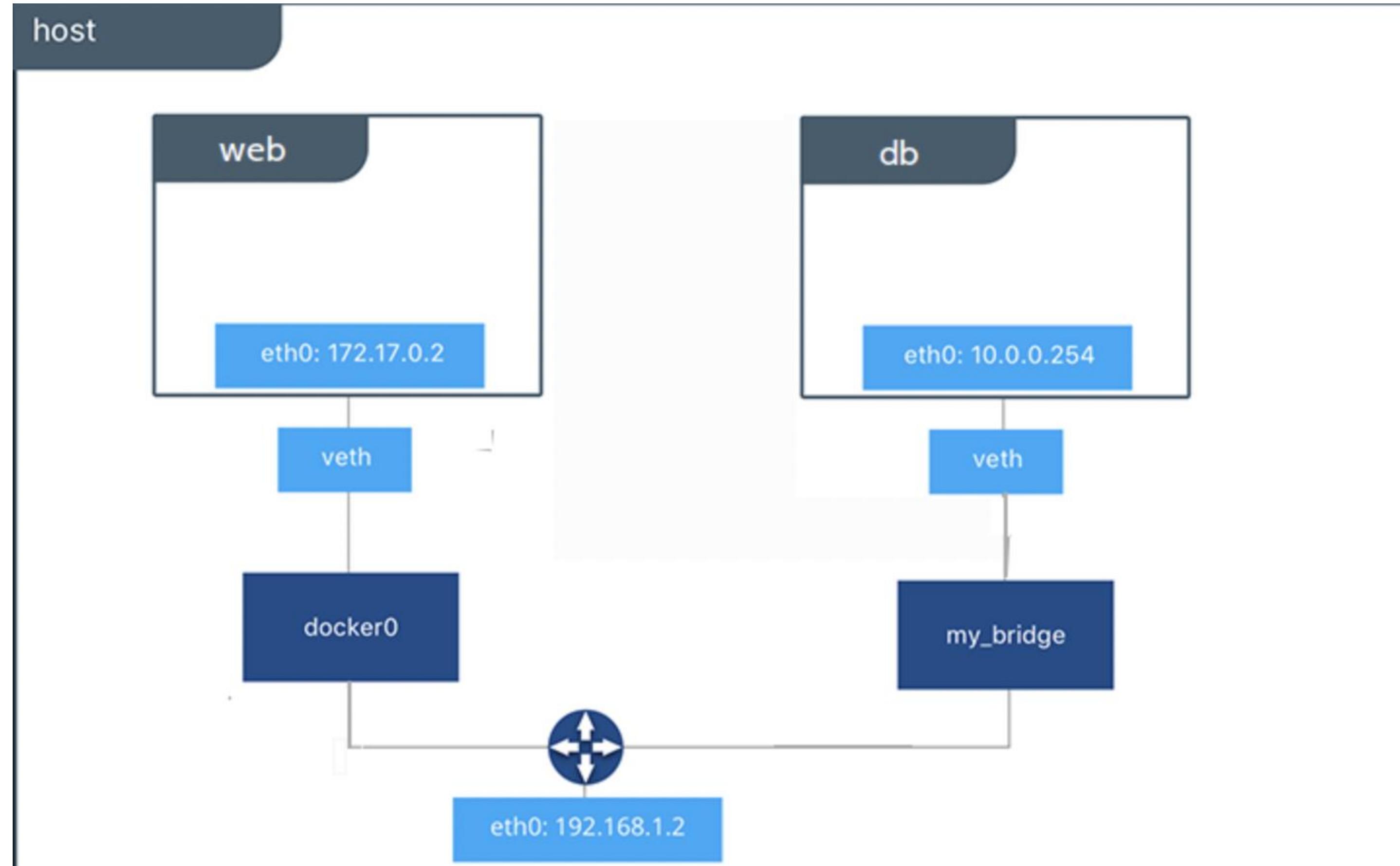
## –Docker Image Creation and Storage

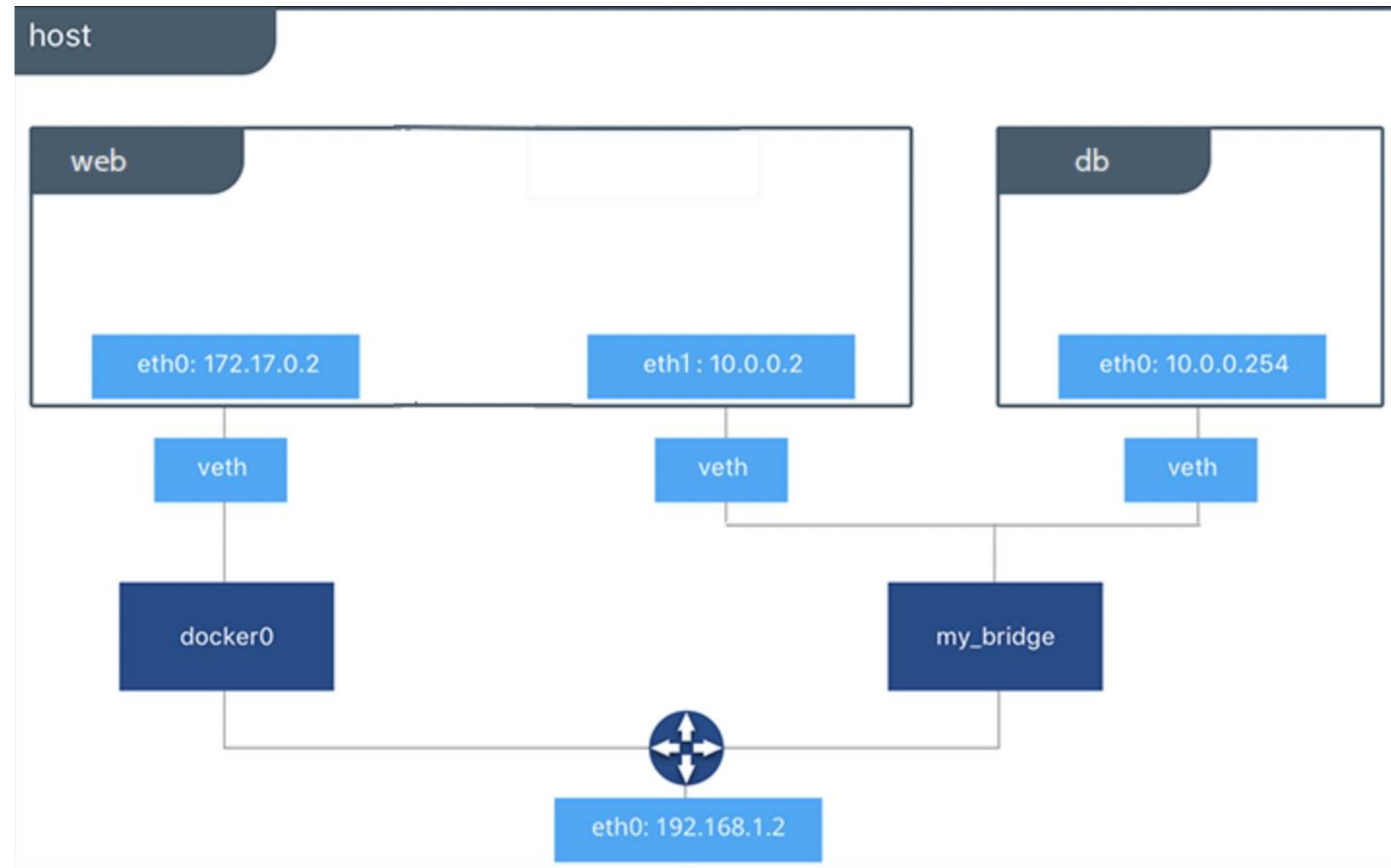
- You can create an image using a Dockerfile or by committing a container's changes back to an image.
- Once you create an image, it will be stored in the Docker host's local image cache.
- In order to move images in/out of the local image cache,
  - Export/ Import it as a tarball
  - Push/ pull to a remote image registry (ex - DockerHub)

# Docker Objects cont...

- Docker Networks
  - Each container is connected to a private virtual network called “bridge”.
  - Each virtual network routes through the NAT firewall on the host IP.
  - All containers on a virtual network can talk to each other without exposing ports.
  - Best practice is to create a new virtual network for each app.

- Docker enables to:
  - Create new virtual networks.
  - Attach container to more than one virtual network (or none)
  - Skip virtual networks and use host IP (--net=host)
  - Use different Docker network drivers to gain new abilities.
    - Docker Engine provides support for different network drivers – bridge (default), overlay and macvian etc.. . You can even write your own network driver plugin to create your own one.
- Docker Networking – DNS
  - Docker deamon has a built in DNS, which consider container name as equivalent hostname of the container.





# persistence data

- If we want to use persistence data as in like databases or unique data in containers, Docker enables that using two ways,
  - Volumes – Make a location outside of container UFS.
  - Bind Mounts - Link host path to the container path.

# Docker compose

- Another Docker client, that lets you work with apps consisting of a set of containers.
  - This saves docker container run settings in easy to read file, which can be committed to VCS.
  - Can use this to create one-line development environments
- Consists of two components
  - YAML formatted file that describes – Images, Containers, Networks, Volumes etc...
  - A CLI tool docker-compose used to automate/manage those YAML files

How can we run containers at scale?

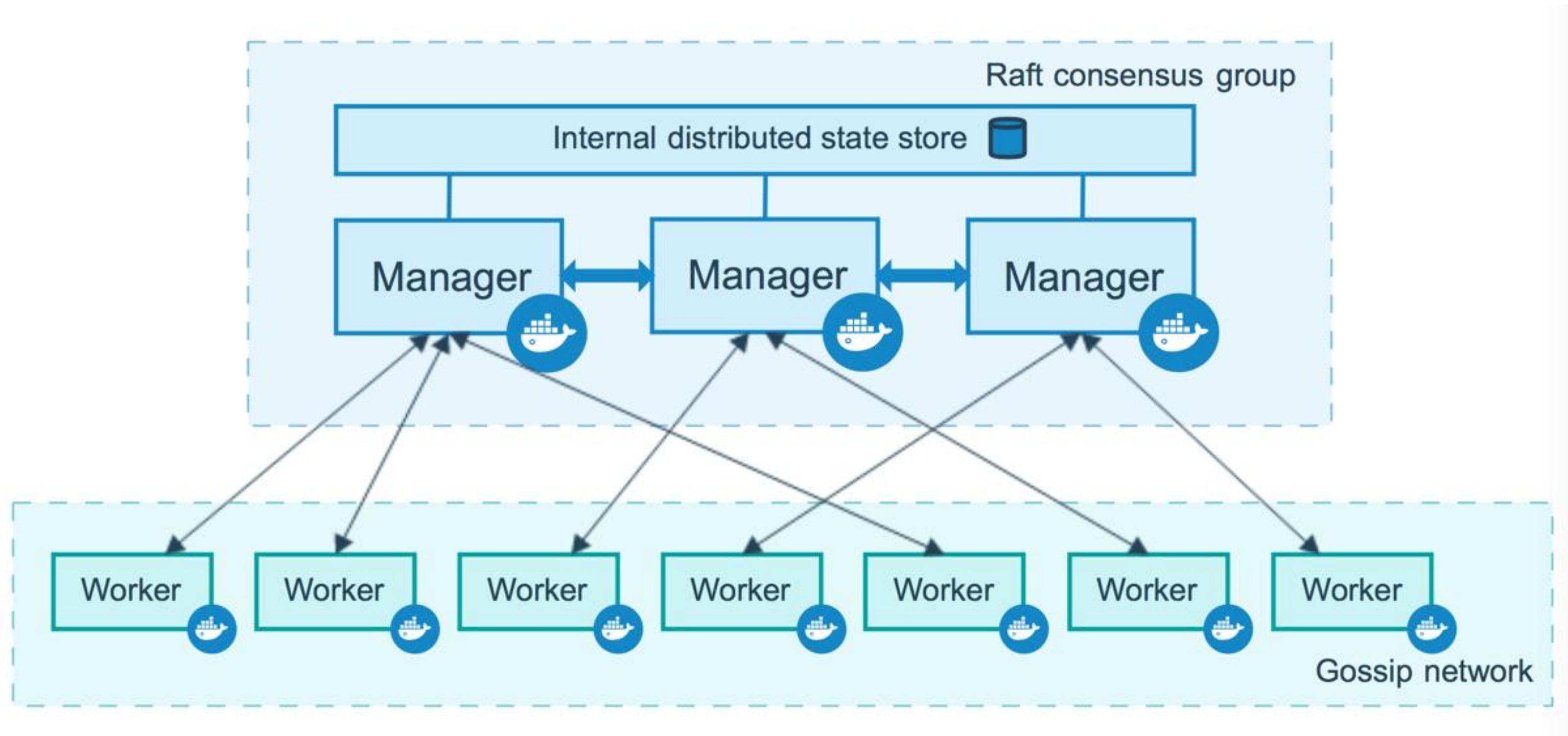
# Container orchestration

- Container orchestration automates the deployment, management, scaling, and networking of containers.
- Container orchestration can be used in any environment where you use containers. It can help you to deploy the same application across different environments without needing to redesign it.

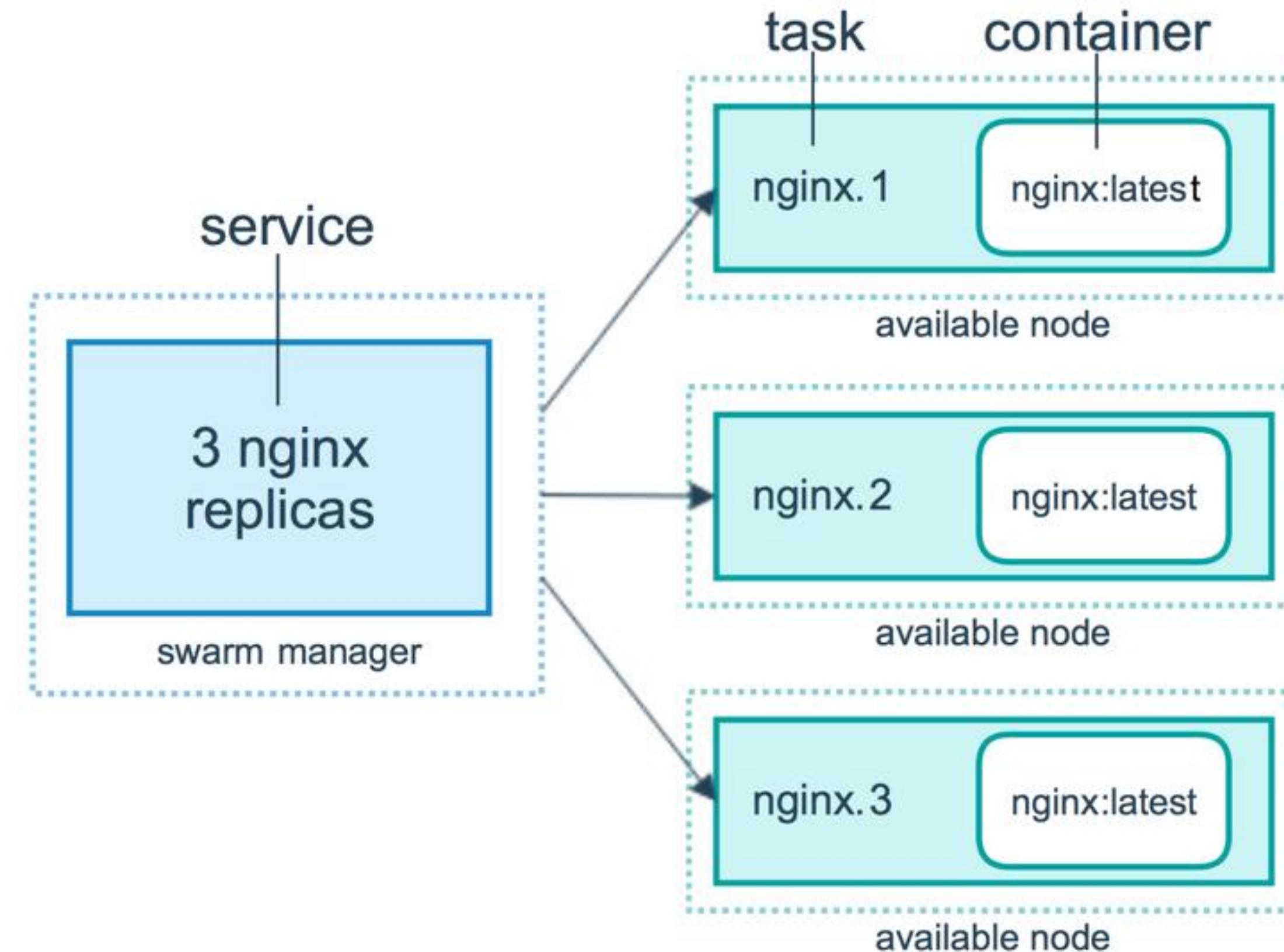
# Introduction to docker swarm

# Docker swarm key concepts

- Docker Swarm provides the cluster management and orchestration features of Docker Engine 1.12
- Nodes - A **node** is an instance of the Docker engine participating in the swarm. You can also think of this as a Docker node.
  - Manager Node - To deploy your application to a swarm, you submit a service definition to a **manager node**. These nodes are also responsible for performing the orchestration and cluster management functions required to maintain the desired state of the swarm.
  - Worker Nodes - Receive and execute tasks dispatched from manager nodes. An agent which is running within the worker nodes report the current status of the tasks assigned to it which allows manager node to keep the desired state for each worker node.



- Task - It is the atomic scheduling unit of swarm. Manager nodes assign tasks to worker nodes according to the number of replicas set in the service scale.
- Service - the definition of the tasks to execute on the manager or worker nodes. Here is where you specify which container image to use and which commands to execute inside running containers.
  - Replicated services model - the swarm manager distributes a specific number of replica tasks among the nodes based upon the scale you set in the desired state.
  - Global services model - the swarm runs one task for the service on every available node in the cluster.



- *docker swarm init --advertise-addr <MANAGER-IP>*
- Join a worker node - *docker swarm join --token SWMTKN-1  
49nj1cmql0jkz5s954yi3oex3nedyz0fb0xx14ie39trti4wxv-8vxv8rssmk743ojnwacrr2e7c  
192.168.99.100:2377*
- *docker service create --replicas 1 --name helloworld alpine ping docker.com*
- *docker service scale <SERVICE-ID>=<NUMBER-OF-TASKS>*

# Introduction to kubernetes

# What is kubernetes?

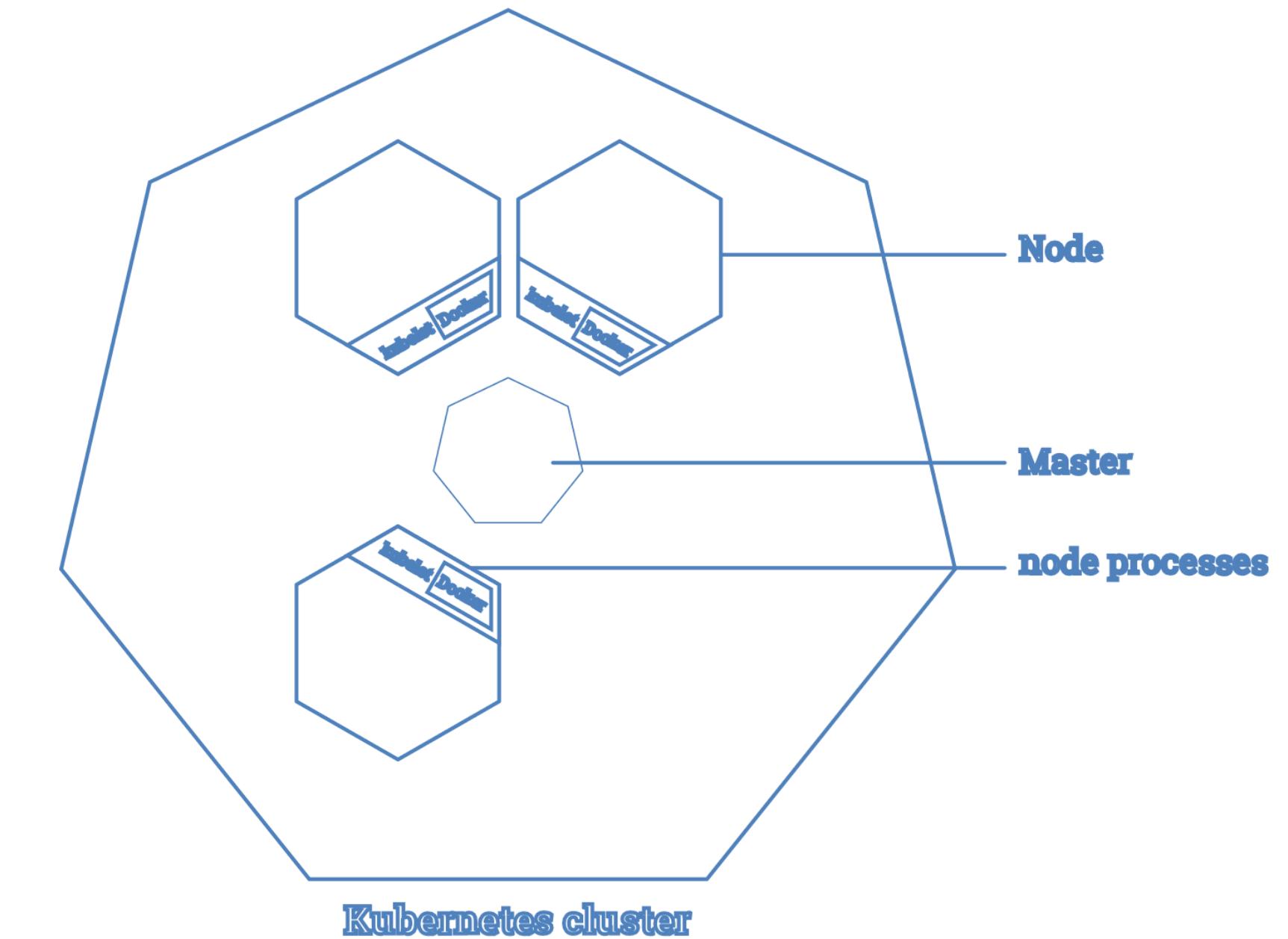
- “Kubernetes (k8s) is an open source platform for automating deployment, scaling and management of containers at scale”
- Project that was created by Google as an open source container orchestration platform. Born from the lessons learned and experiences in running projects like Borg and Omega @ Google
- It was donated to CNCF (Cloud Native Computing Foundation) who now manages the Kubernetes project
- Current Kubernetes stable version – 1.29

# It's capable of...

- Horizontal scaling
- Load distribution
- Service discovery
- Health monitoring
- Deploying new versions, rollbacks
- Handling hardware failures

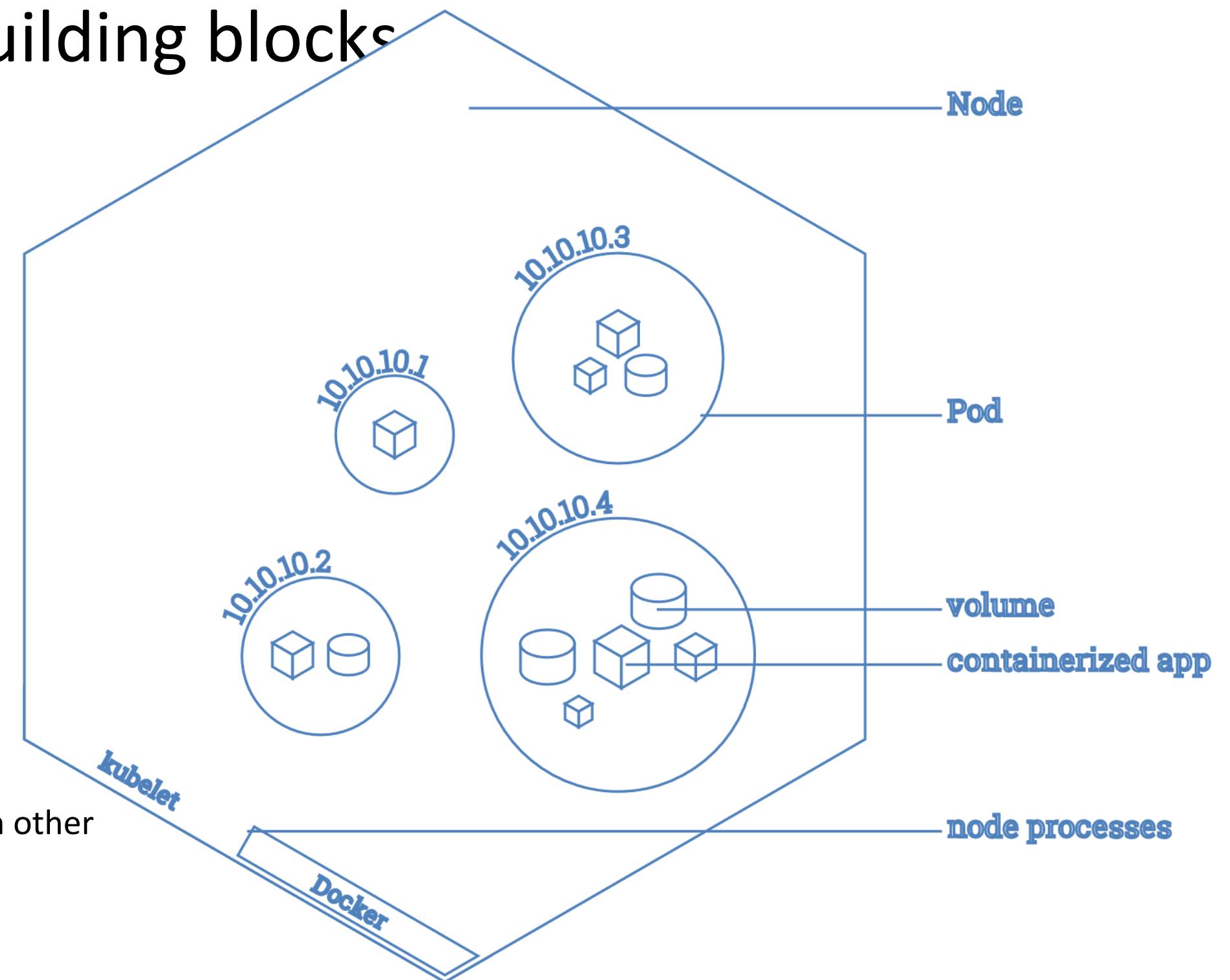
# High level Architecture

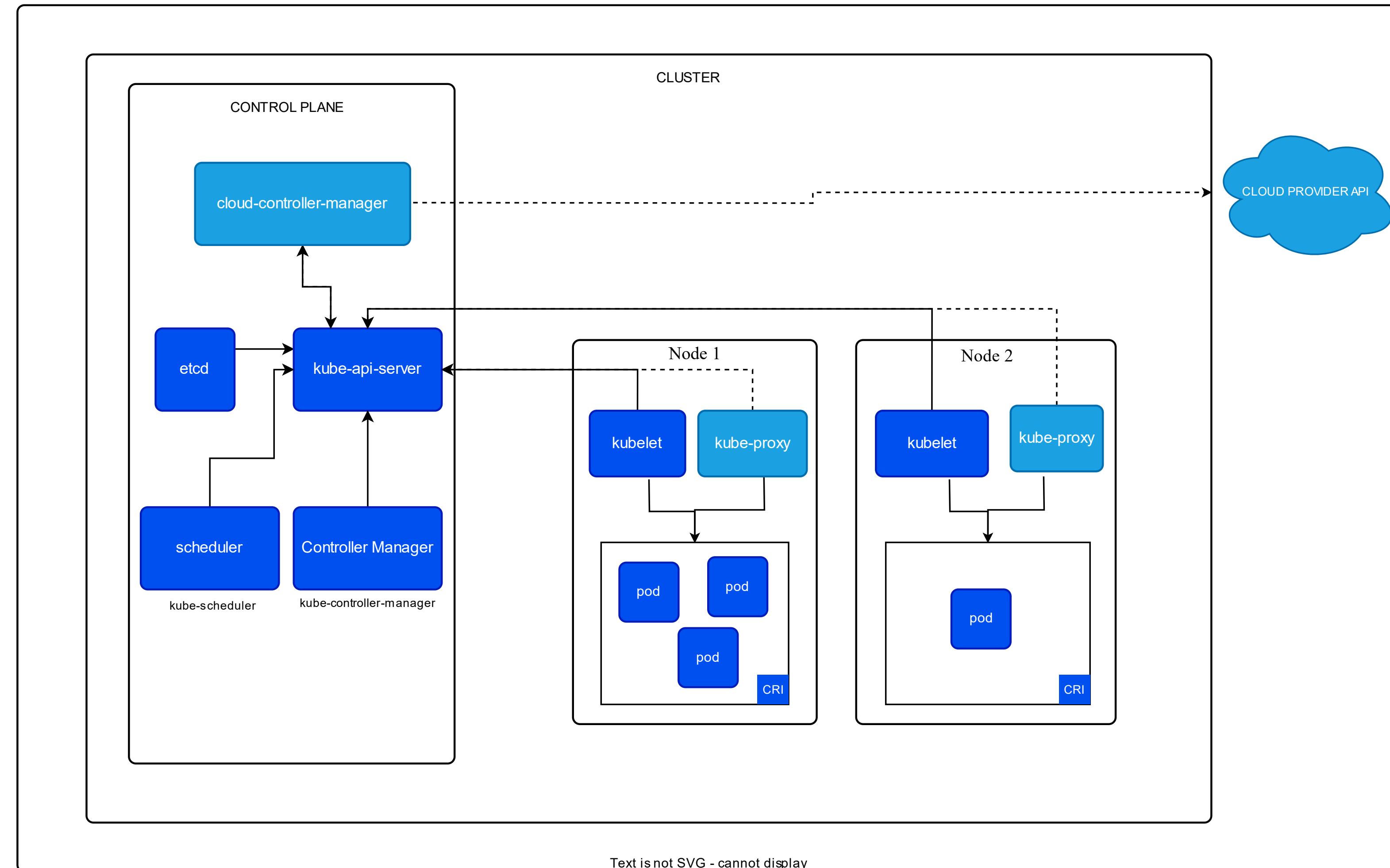
- Master [Control Plane]
  - coordinates all activities in the cluster
- Nodes:
  - virtual or physical machines
  - actual workers
  - runs processes:
    - kubelet
    - kube-proxy
    - container runtime



# Basic building blocks

- Containers
  - Define single running process\*
  - Eg docker container
- Pods
  - the way of running containers in kubernetes
  - basic deployable and scaling unit
  - defines one or more containers
  - containers are co-located on a node
  - flat network structure
- Nodes:
  - physical worker machines
  - can run multiple pods
  - pods running within single node don't know about each other





# Running things locally

- Minikube:
  - single node cluster
  - running in a VM
  - supports linux, windows and macOS
  - mature project
- kind:
  - Requires you to have Docker or Podman installed in your local computer
- Docker Desktop with built-in kubernetes:
  - single node cluster
  - running in a VM
  - windows and macOS
  - drag & drop installation
  - bound to specific kubernetes version

<https://kubernetes.io/docs/tasks/tools/>

# Managing cluster resources

- **Create resource from file** - *kubectl create -f resource\_file.yml*
- **Change existing (or create) resource based on file** - *kubectl apply -f resource\_file.yml*
- **Delete existing resource** - *kubectl delete resource\_type resource\_name*
- **List resources of type** - *kubectl get resource\_type*
- **Edit resource on the server** - *kubectl edit resource\_type resource\_name*

# Debugging cluster resources

- **Execute command on the container** - *kubectl exec [-it] pod\_name process\_to\_run*
- **Get container logs** - *kubectl logs pod\_name [-c container\_name]*
- **Forward port from a pod** - *kubectl port-forward pod\_name local\_port:remote\_port*
- **Print detailed description of a resource** - *kubectl describe resource\_type resource\_name*

# Few resource objects in k8s

- **Replica Sets** - Ensures desired number of pods exist by: scaling up or down and running new pods when nodes fail
- ***Deployment***
  - 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.
- ***Service***
  - A method for exposing a network application that is running as one or more Pods in your cluster.
    - Cluster IP/ NodePort/ Load Balancer/ Ingress

Feature	ClusterIP	NodePort	LoadBalancer
<b>Exposition</b>	Exposes the Service on an internal IP in the cluster.	Exposing services to external clients	Exposing services to external clients
<b>Cluster</b>	This type makes the Service only reachable from within the cluster	A NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service.	A LoadBalancer service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on
<b>Accessibility</b>	It is <b>default</b> service and Internal clients send requests to a stable internal IP address.	The service is accessible at the internal cluster IP-port, and also through a dedicated port on all nodes.	Clients connect to the service through the load balancer's IP.
<b>Yaml Config</b>	<code>type: ClusterIP</code>	<code>type: NodePort</code>	<code>type: LoadBalancer</code>
<b>Port Range</b>	Any public ip form Cluster	30000 - 32767	Any public ip form Cluster

demo

- *Defining a Pod*
- *Creating a ReplicaSet*
- *Creating a Deployment*
- *Creating a Service and exposing it*

Q/A

# references

- <https://docs.docker.com/get-started/overview/>
- <https://www.docker.com/blog/containers-and-vms-together/>
- <https://www.redhat.com/en/topics/containers/containers-vs-vms>
- Docker Storage Drivers - <https://docs.docker.com/storage/storagedriver/>
- <https://docs.docker.com/storage/storagedriver/select-storage-driver/>
- <https://www.youtube.com/watch?v=cjXI-yxqGTI>
- Docker Buildx - <https://docs.docker.com/buildx/working-with-buildx/>
- Jiang Huan BuildKit timings - <https://medium.com/titansoft-engineering/docker-build-cache-sharing-on-multi-hosts-with-buildkit-and-buildx-eb8f7005918e>
- What is Docker BuildKit - <https://brianchristner.io/what-is-docker-buildkit/>

Thank you!

LinkedIn - <https://lk.linkedin.com/in/ravindufernando>



Introduction → Deep Dive

Ravindu Nirmal Fernando | SLIIT | February 2025

# KUBERNETES (K8S)



How can we run containers at scale?

# Container orchestration

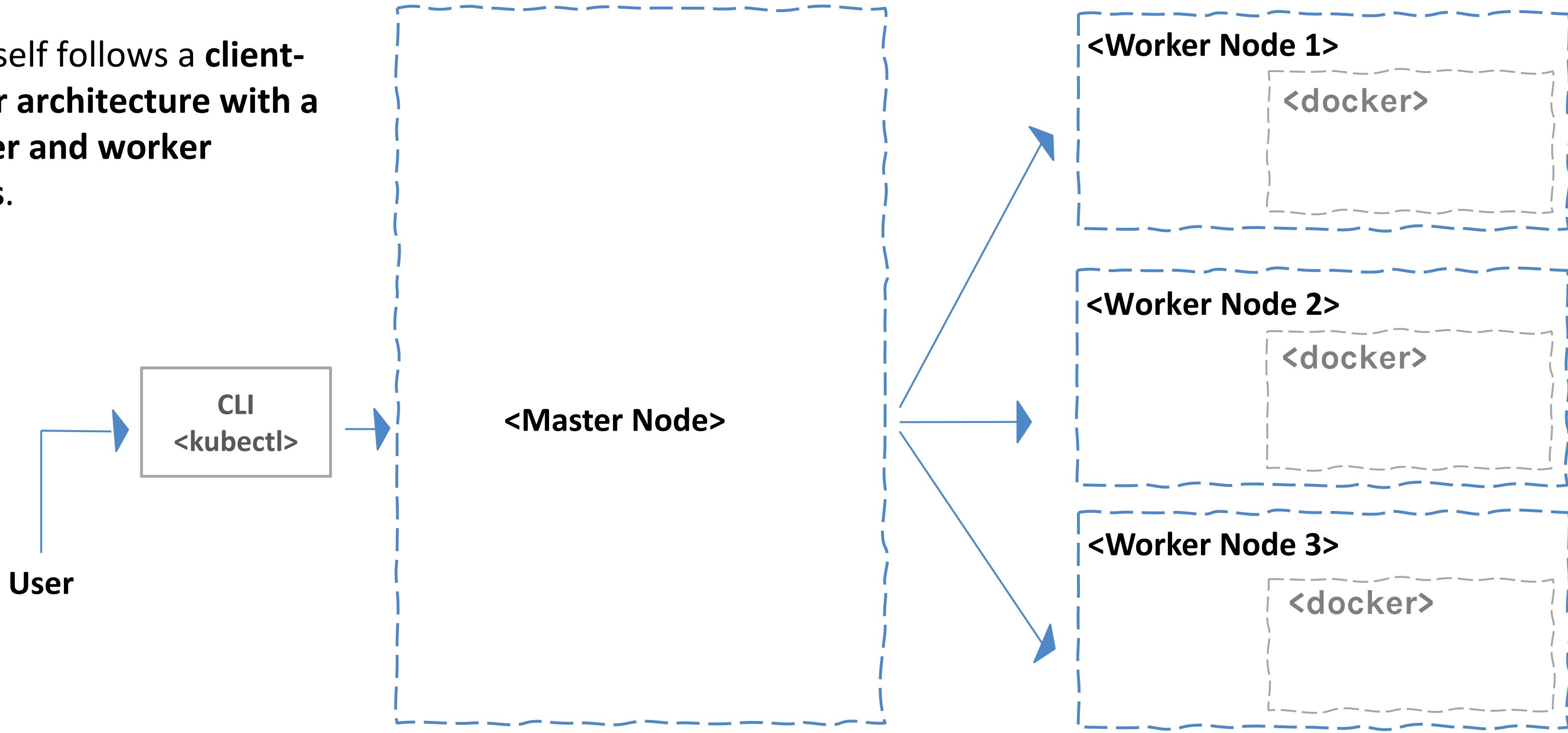
- Container orchestration automates the deployment, management, scaling, and networking of containers.
- Container orchestration can be used in any environment where you use containers. It can help you to deploy the same application across different environments without needing to redesign it.

# What is Kubernetes?

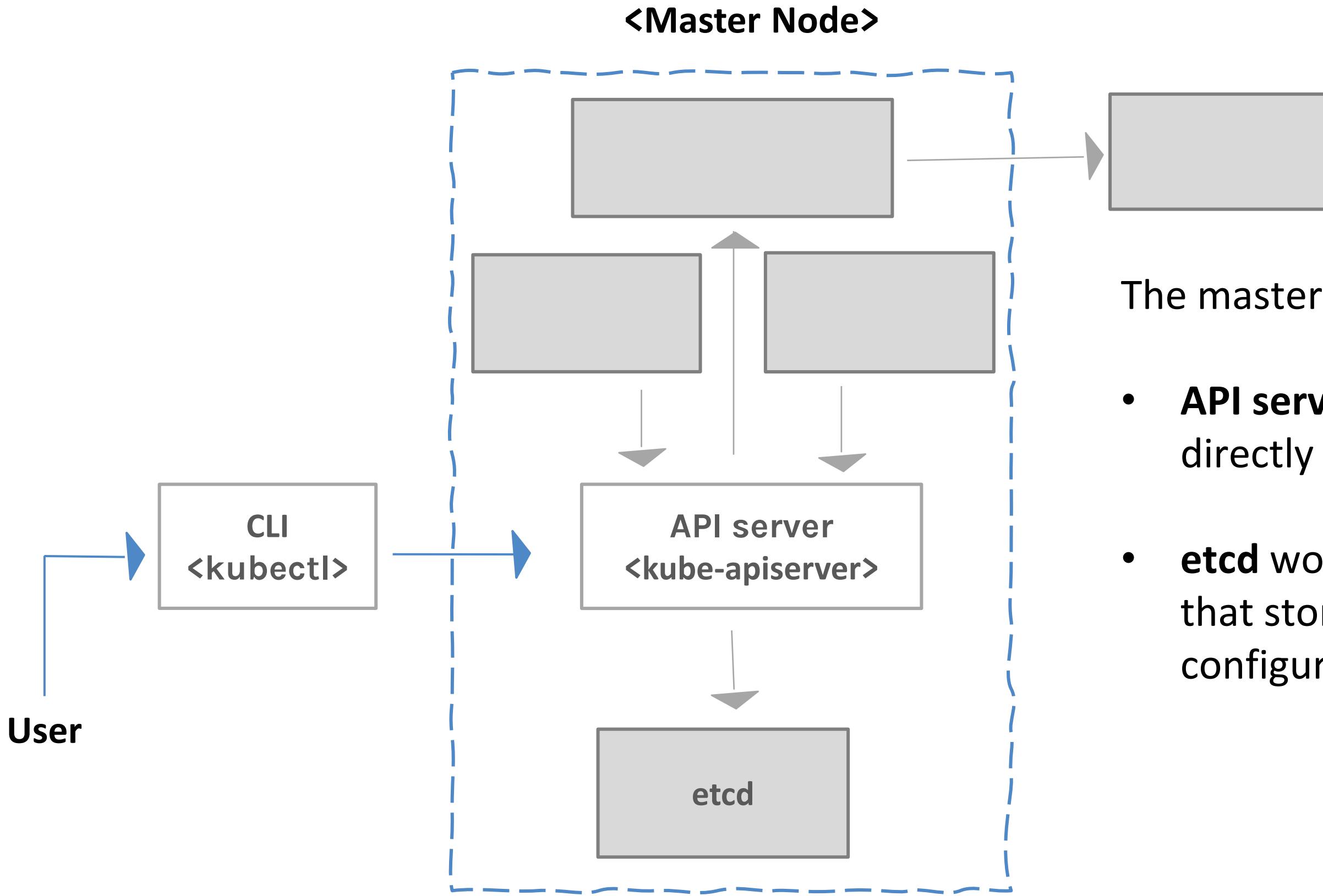
- “Kubernetes (k8s) is an open source platform for automating deployment, scaling and management of containers at scale”
- Project that was created by Google as an open source container orchestration platform. Born from the lessons learned and experiences in running projects like Borg and Omega @ Google
- It was donated to CNCF (Cloud Native Computing Foundation) who now manages the Kubernetes project
- Current Kubernetes stable version – 1.32

# K8s Components & Architecture

K8s itself follows a **client-server architecture with a master and worker nodes.**



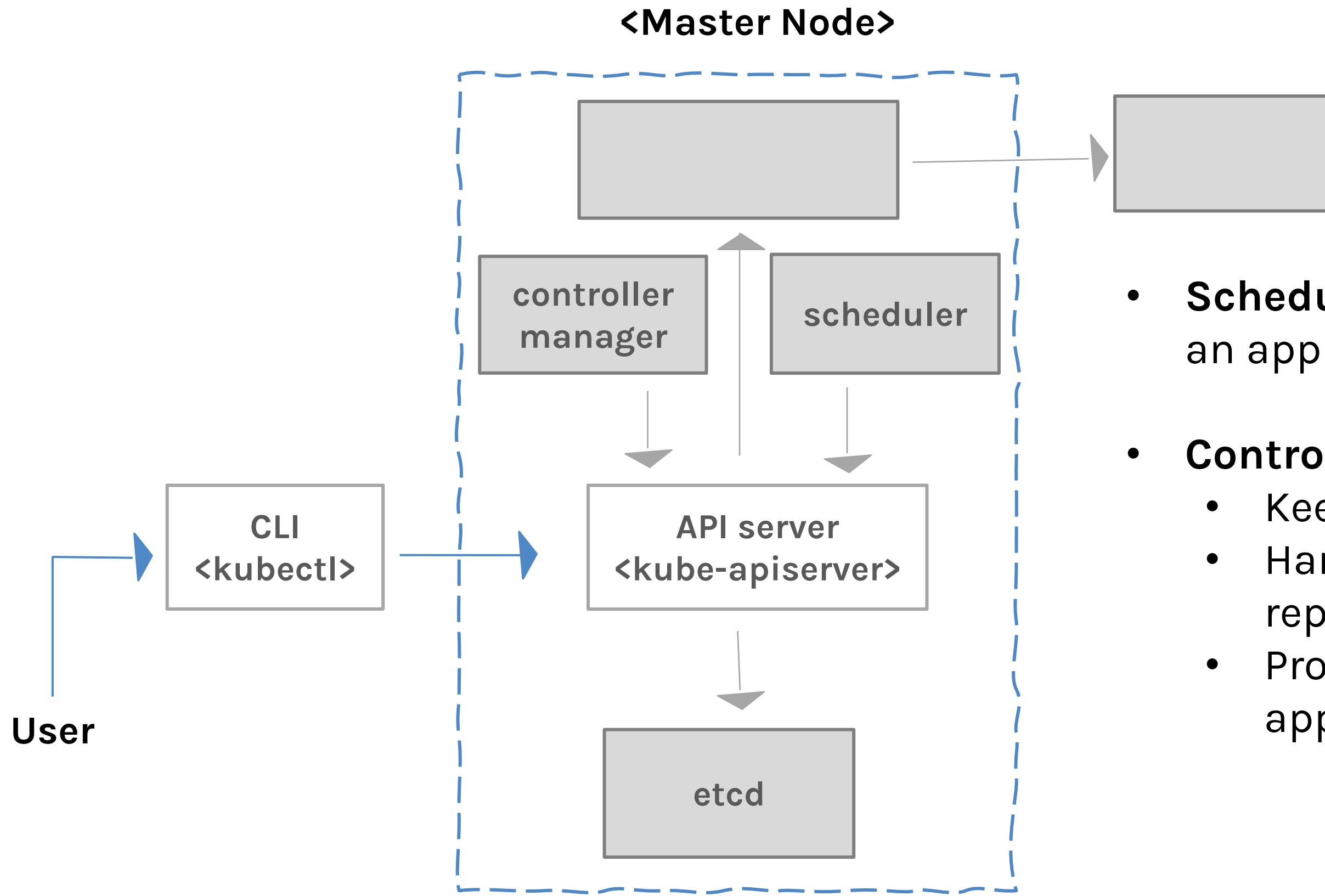
# K8s Components & Architecture <cont>



The master node has:

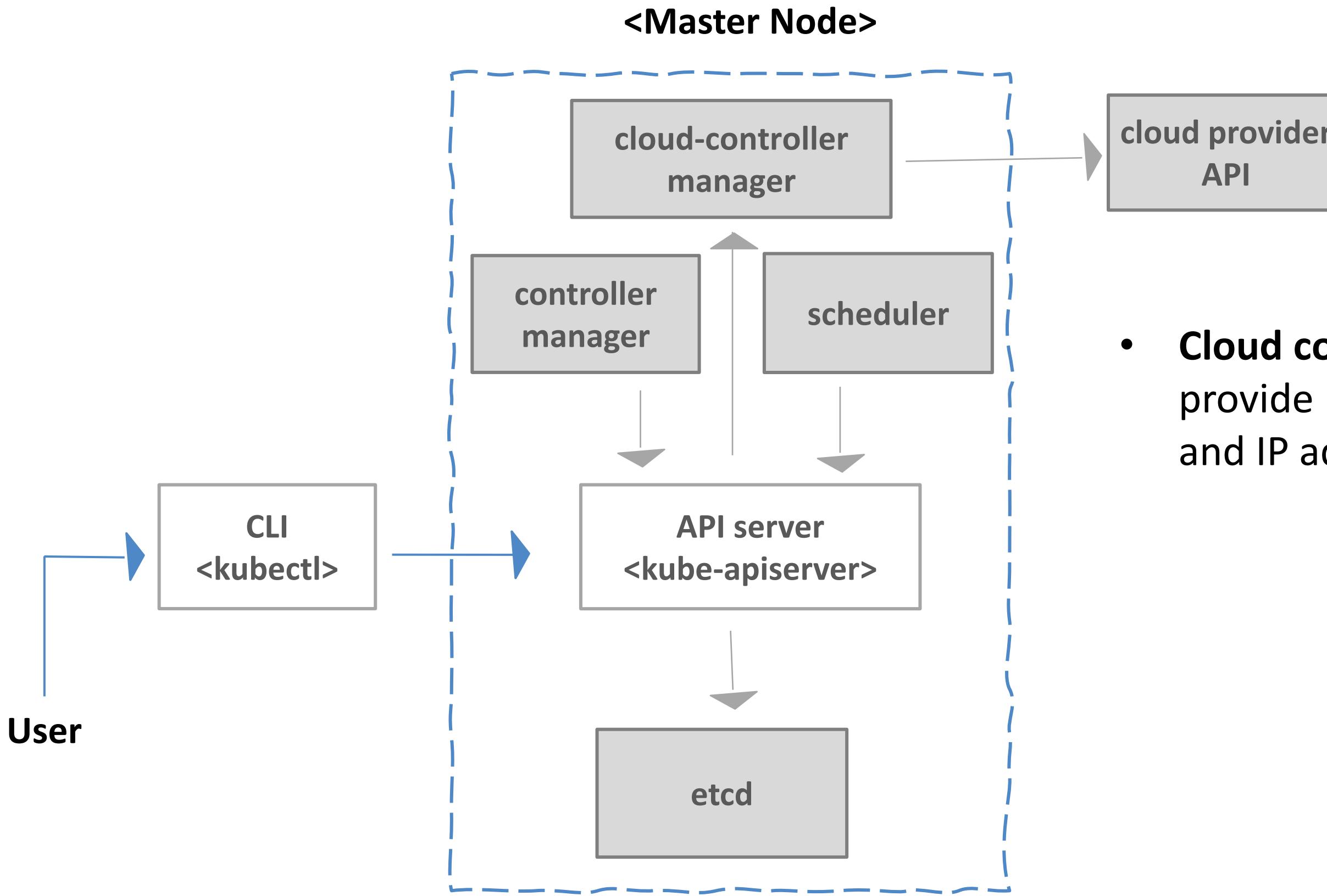
- **API server** contains various methods to directly access the Kubernetes
- **etcd** works as backend for service discovery that stores the cluster's state and its configuration

# K8s Components & Architecture <cont>



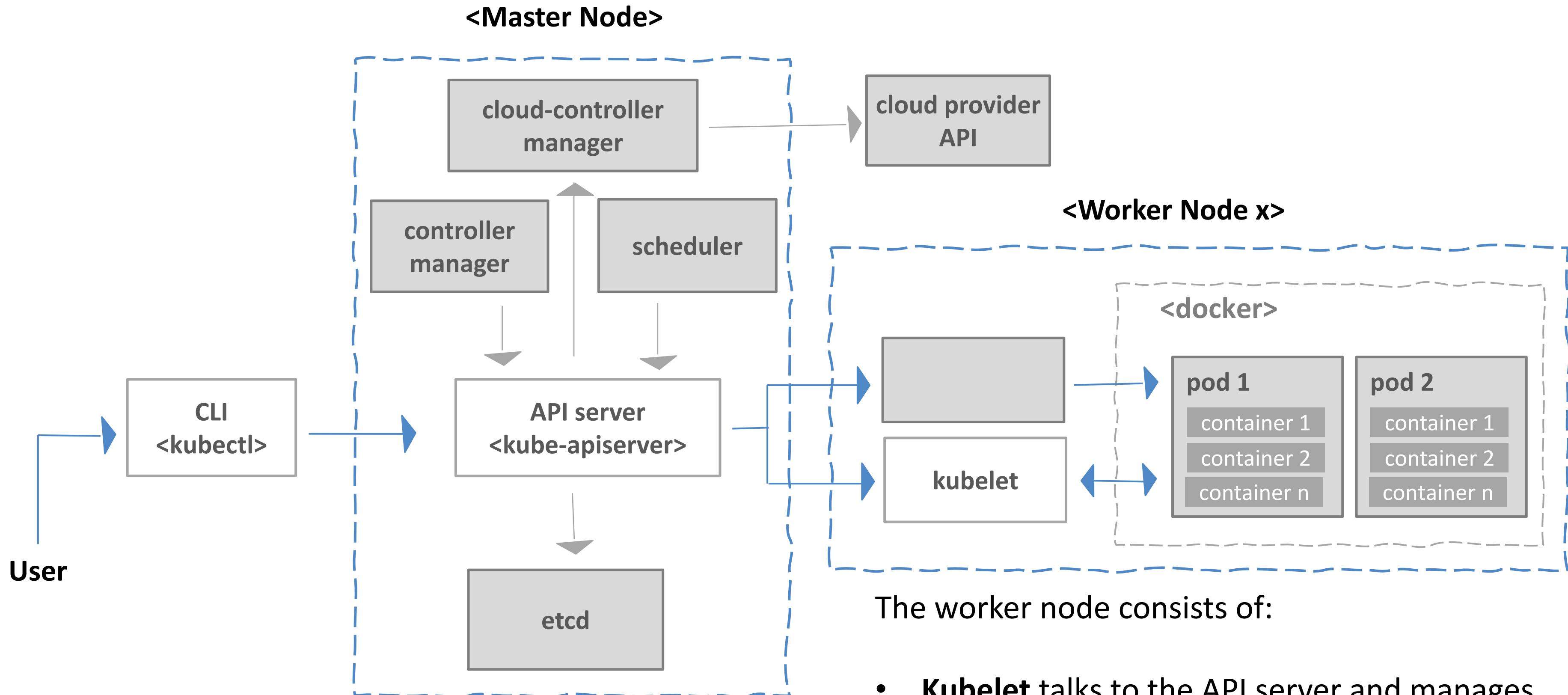
- **Scheduler** assigns to each worker node an application
- **Controller manager:**
  - Keeps track of worker nodes
  - Handles node failures and replicates if needed
  - Provide endpoints to access the application from the outside world

# K8s Components & Architecture <cont>

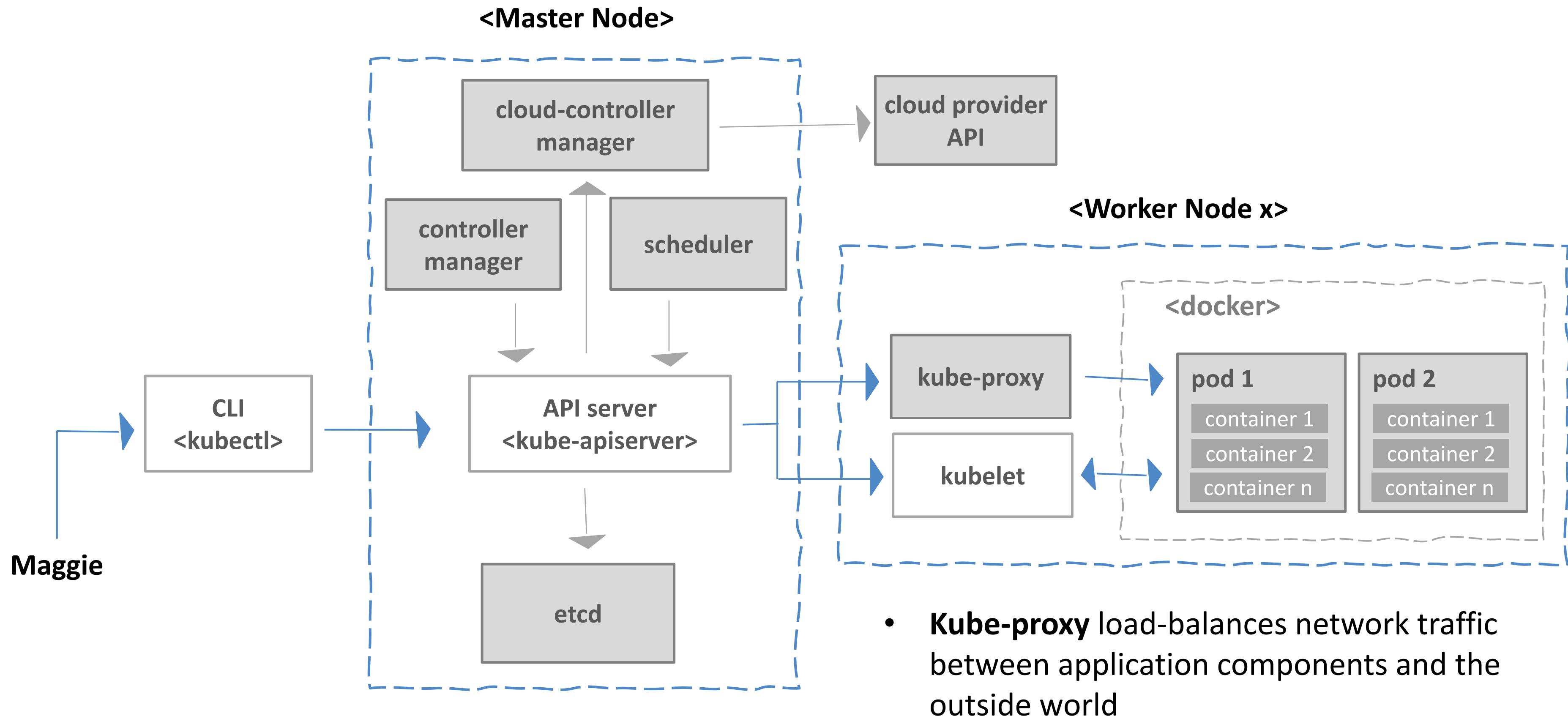


- **Cloud controller** communicates with cloud provider regarding resources such as nodes and IP addresses

# K8s Components & Architecture <cont>

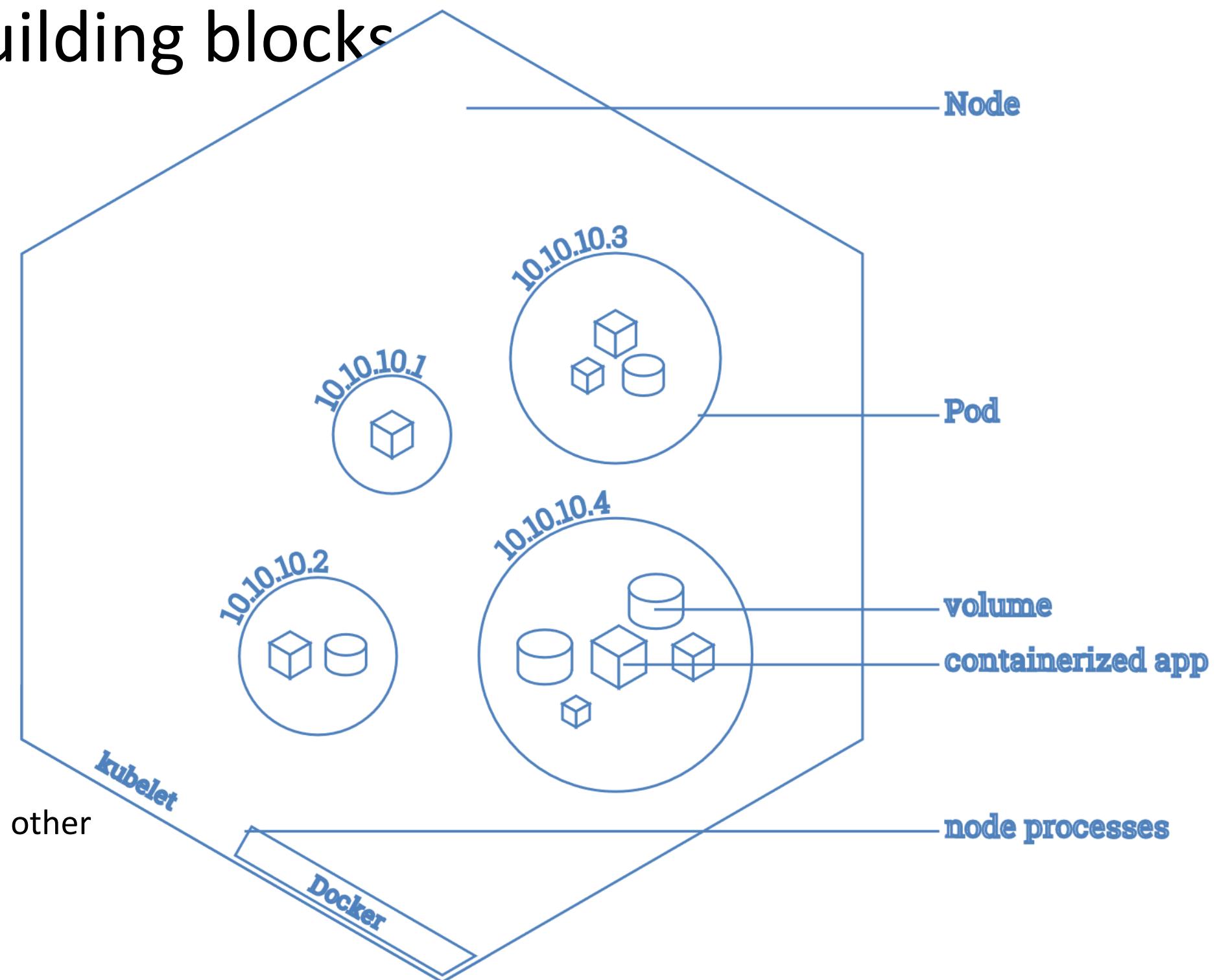


# K8s Components & Architecture <cont>



# Basic building blocks

- Containers
  - Define single running process\*
  - E.g. docker container
- Pods
  - the way of running containers in Kubernetes
  - basic deployable and scaling unit
  - defines one or more containers
  - containers are co-located on a node
  - flat network structure
- Nodes:
  - physical worker machines
  - can run multiple pods
  - pods running within single node don't know about each other



# Running things locally

- Minikube:
  - single node cluster
  - running in a VM
  - supports linux, windows and macOS
  - mature project
- kind:
  - Requires you to have Docker or Podman installed in your local computer
- Docker Desktop with built-in kubernetes:
  - single node cluster
  - running in a VM
  - windows and macOS
  - drag & drop installation
  - bound to specific kubernetes version

<https://kubernetes.io/docs/tasks/tools/>

# Managing cluster resources

- **Create resource from file** - *kubectl create -f resource\_file.yml*
- **Change existing (or create) resource based on file** - *kubectl apply -f resource\_file.yml*
- **Delete existing resource** - *kubectl delete resource\_type resource\_name*
- **List resources of type** - *kubectl get resource\_type*
- **Edit resource on the server** - *kubectl edit resource\_type resource\_name*

# Debugging cluster resources

- **Execute command on the container** - *kubectl exec [-it] pod\_name process\_to\_run*
- **Get container logs** - *kubectl logs pod\_name [-c container\_name]*
- **Forward port from a pod** - *kubectl port-forward pod\_name local\_port:remote\_port*
- **Print detailed description of a resource** - *kubectl describe resource\_type resource\_name*

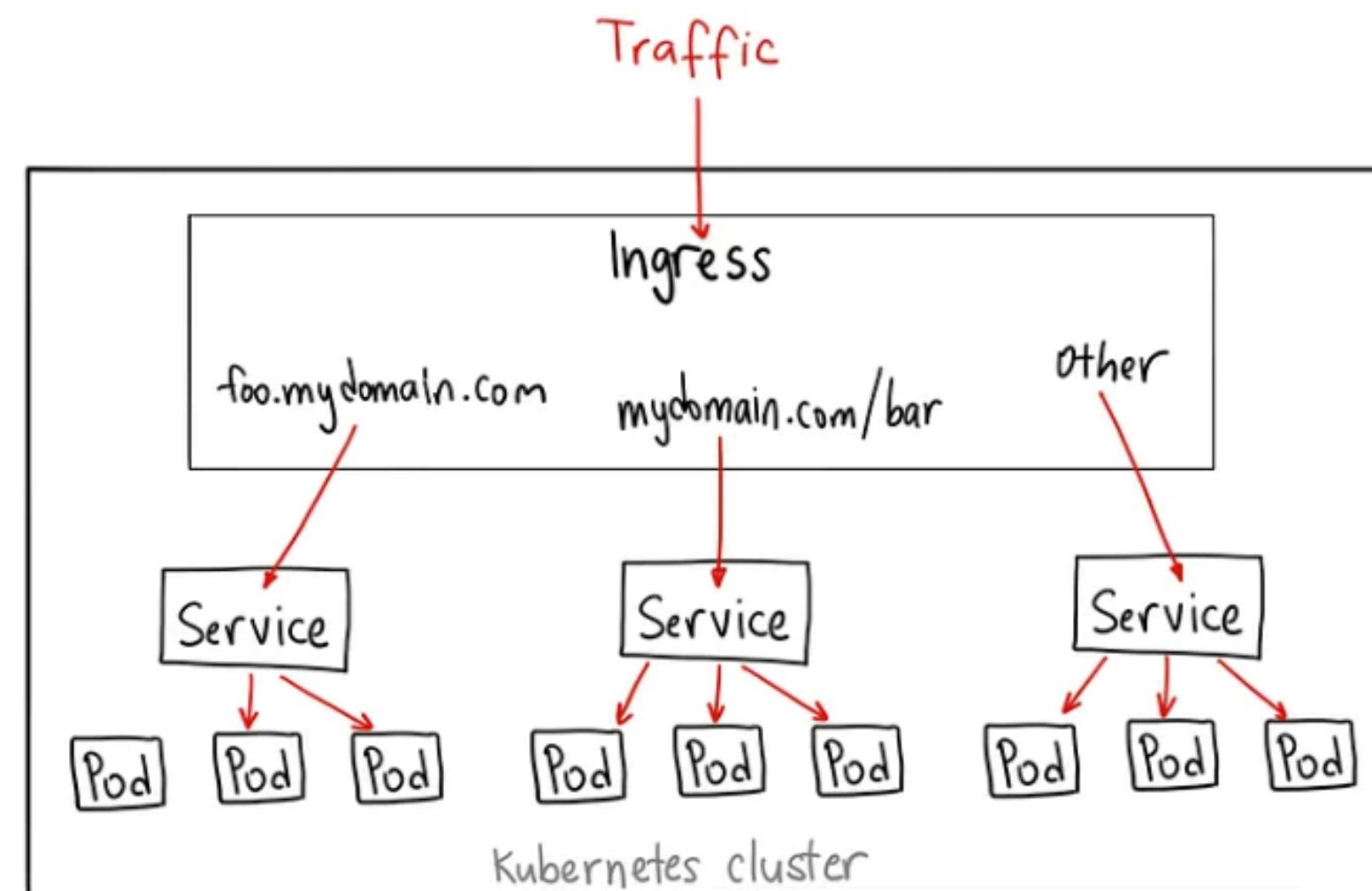
# Few resource objects in K8s

- **Replica Sets** - Ensures desired number of pods exist by: scaling up or down and running new pods when nodes fail
- ***Deployment***
  - 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.
- ***Service***
  - A method for exposing a network application that is running as one or more Pods in your cluster.
    - Cluster IP/ NodePort/ Load Balancer

Feature	ClusterIP	NodePort	LoadBalancer
<b>Exposition</b>	Exposes the Service on an internal IP in the cluster.	Exposing services to external clients	Exposing services to external clients
<b>Cluster</b>	This type makes the Service only reachable from within the cluster	A NodePort service, each cluster node opens a port on the node itself (hence the name) and redirects traffic received on that port to the underlying service.	A LoadBalancer service accessible through a dedicated load balancer, provisioned from the cloud infrastructure Kubernetes is running on
<b>Accessibility</b>	It is <b>default</b> service and Internal clients send requests to a stable internal IP address.	The service is accessible at the internal cluster IP-port, and also through a dedicated port on all nodes.	Clients connect to the service through the load balancer's IP.
<b>Yaml Config</b>	<code>type: ClusterIP</code>	<code>type: NodePort</code>	<code>type: LoadBalancer</code>
<b>Port Range</b>	Any public ip form Cluster	30000 - 32767	Any public ip form Cluster

# Few resource objects in K8s

- **Ingress** - An Ingress is a Kubernetes object that sits in front of multiple services and acts as an intelligent router. It defines how external traffic can reach the cluster services, and it configures a set of rules to allow inbound connections to reach the services on the cluster.



Thanks to [Ahmet Alp Balkan](#) for the diagrams

# Demo

- *Defining a Pod*
- *Creating a ReplicaSet*
- *Creating a Deployment*
- *Creating a Service and exposing it*

I want to be able to  
deploy and share my  
app everywhere  
consistently, and  
manage it as a single  
entity regardless of the  
different parts.

# Deploying an App – *kubectl* Way

- Let's see what it takes to deploy an app on a running Kubernetes cluster
  - There will be lot's of YAML Kubernetes manifest files
    - Ex:-
      - Application deployment and service configuration
      - Redis master deployment and service configuration
      - Redis slaves deployment and service configuration
  - Using the Kubernetes client, *kubectl*
    - Create Deployment
    - Manage Deployment

Refer - <https://github.com/IBM/guestbook/tree/master/v1>

# Deploying an App – *kubectl* Way – Pain Points

## –CI/CD pipeline

- *kubectl* deployments are not easy to configure, update and rollback
  - Deploying app to dev/test/production may require different configuration
    - » Update deployment e.g. update with a new image
    - » Change the configuration based on certain conditions
    - » A different serviceType is needed in different environments (e.g. NodePort/LoadBalancer)
    - » Need for rollback
    - » Need of having multiple deployments (e.g. multiple Redis deployments)
- Requires to track your deployment and modify YAML files (can be error prone)
- Does not allow multiple deployments without updating metadata in manifest files

## –Share your deployment configurations with your friend, team or customer?

- You need to share many files and related dependencies
- Your users are required to have knowledge of deployment configuration

# Here Comes Helm

- Deploying an app – Helm Way
  - No expertise of Kubernetes deployment needed as Helm hides Kubernetes domain complexities
  - Helm packages all dependencies
  - Helm tracks deployment making it easy to update and rollback
  - Same workload can be deployed multiple times
    - Helm allows assigning workload release names at runtime
  - Easy to share

# What is Helm?

- Helm is a tool that streamlines installation and management of Kubernetes applications
  - A tool or package manager for the Kubernetes, for deployment and management of applications into a Kubernetes cluster
  - Helm became a CNCF project in mid 2018
- It uses a packaging format called **charts**
  - A chart is a collection of files that describe Kubernetes resources
  - Think of Helm like apt/yum/homebrew for Kubernetes
- Helm is available for various operating systems like OSX, Linux and Windows
- Run Helm anywhere e.g. laptop, CI/CD etc.





## What Helm is NOT

- A fully fledged system package manager
- A configuration management tool like Chef, puppet etc.
- A Kubernetes resource lifecycle controller



A chart is organized as a collection of files inside of a directory. The directory name is the name of the chart (without versioning information). Thus, a chart describing WordPress would be stored in a `wordpress/` directory.

Inside of this directory, Helm will expect a structure that matches this:

```
wordpress/
  Chart.yaml          # A YAML file containing information about the chart
  LICENSE            # OPTIONAL: A plain text file containing the license for the chart
  README.md          # OPTIONAL: A human-readable README file
  values.yaml        # The default configuration values for this chart
  values.schema.json # OPTIONAL: A JSON Schema for imposing a structure on the values.yaml file
  charts/             # A directory containing any charts upon which this chart depends.
  crds/              # Custom Resource Definitions
  templates/          # A directory of templates that, when combined with values,
                      # will generate valid Kubernetes manifest files.
  templates/NOTES.txt # OPTIONAL: A plain text file containing short usage notes
```

Helm reserves use of the `charts/`, `crds/`, and `templates/` directories, and of the listed file names. Other files will be left as they are.

# Demo – Guestbook Chart Deployment

- Check existing installation of Helm chart
  - *helm ls*
- Check what repo do you have
  - *helm repo list*
- Add repo
  - *helm repo add helm101 <https://ibm.github.io/helm101/>*
- Verify that helm101/guestbook is now in your repo
  - *helm repo list*
  - *helm search helm101*
- Install
  - ***helm install helm101/guestbook --name myguestbook --set service.type=NodePort*** – follow the output instructions to see your guestbook application
- Verify that your guestbook chart is installed
  - *helm ls*
- Check chart release history
  - *helm history myguestbook*

# Demo – Guestbook Upgrades and Rollback

- First let's see what we have
  - ***helm history myguestbook***
- Upgrade
  - ***helm upgrade myguestbook helm101/guestbook***
  - ***helm history myguestbook***
- Rollback
  - ***helm rollback myguestbook 1***
  - ***helm history myguestbook***

# Demo – Clean Up

- Remove repo
  - **helm repo remove helm101**
- Remove chart completely
  - **helm delete --purge myguestbook**
    - Delete all Kubernetes resources generated when the chart was instantiated

Another Demo!!!

<https://github.com/rav94/devops-in-practice>

# References

- <https://kubernetes.io/docs/concepts/overview/components/>
- <https://helm.sh/docs/intro/quickstart/>



Introduction

Ravindu Nirmal Fernando | SLIIT | February 2025

# **GITOPS**

# GitOps Principles

v0.1.0

## 1 The principle of declarative desired state

A system managed by GitOps must have its Desired State expressed declaratively as data in a format writable and readable by both humans and machines.

## 2 The principle of immutable desired state versions

Desired State is stored in a way that supports versioning, immutability of versions, and retains a complete version history.

## 3 The principle of continuous state reconciliation

Software agents continuously, and automatically, compare a system's Actual State to its Desired State. If the actual and desired states differ for any reason, automated actions to reconcile them are initiated.

## 4 The principle of operations through declaration

The only mechanism through which the system is intentionally operated on is through these principles.

# GitOps in K8s

In the case of Kubernetes, GitOps deployments happen in the following manner:

A GitOps agent is deployed on the cluster.

- The GitOps agent is monitoring one or more Git repositories that define applications and contain Kubernetes manifests (or Helm charts or Kustomize files).
- Once a Git commit happens the GitOps agent is instructing the cluster to reach the same state as what is described in Git.
- Developers, operators, and other stakeholders perform all changes via Git operations and never directly touch the cluster (or perform manual kubectl commands).

# Traditional deployment without GitOps:

- 1 - A developer commits source code for the application.
- 2 - A CI system builds the application and may also perform additional actions such as unit tests, security scans, static checks, etc.
- 3 - The container image is stored in a Container registry.
- 4 - The CI platform (or other external system) with direct access to the Kubernetes cluster creates a deployment using a variation of the “kubectl apply” command.
- 5 - The application is deployed on the cluster.



- The cluster state is manually decided by `kubectl` commands or other API access.
- The platform that deploys to the cluster is having full access to the Kubernetes cluster from an external point.

# Modifying the process with GitOps

The first steps are the same.

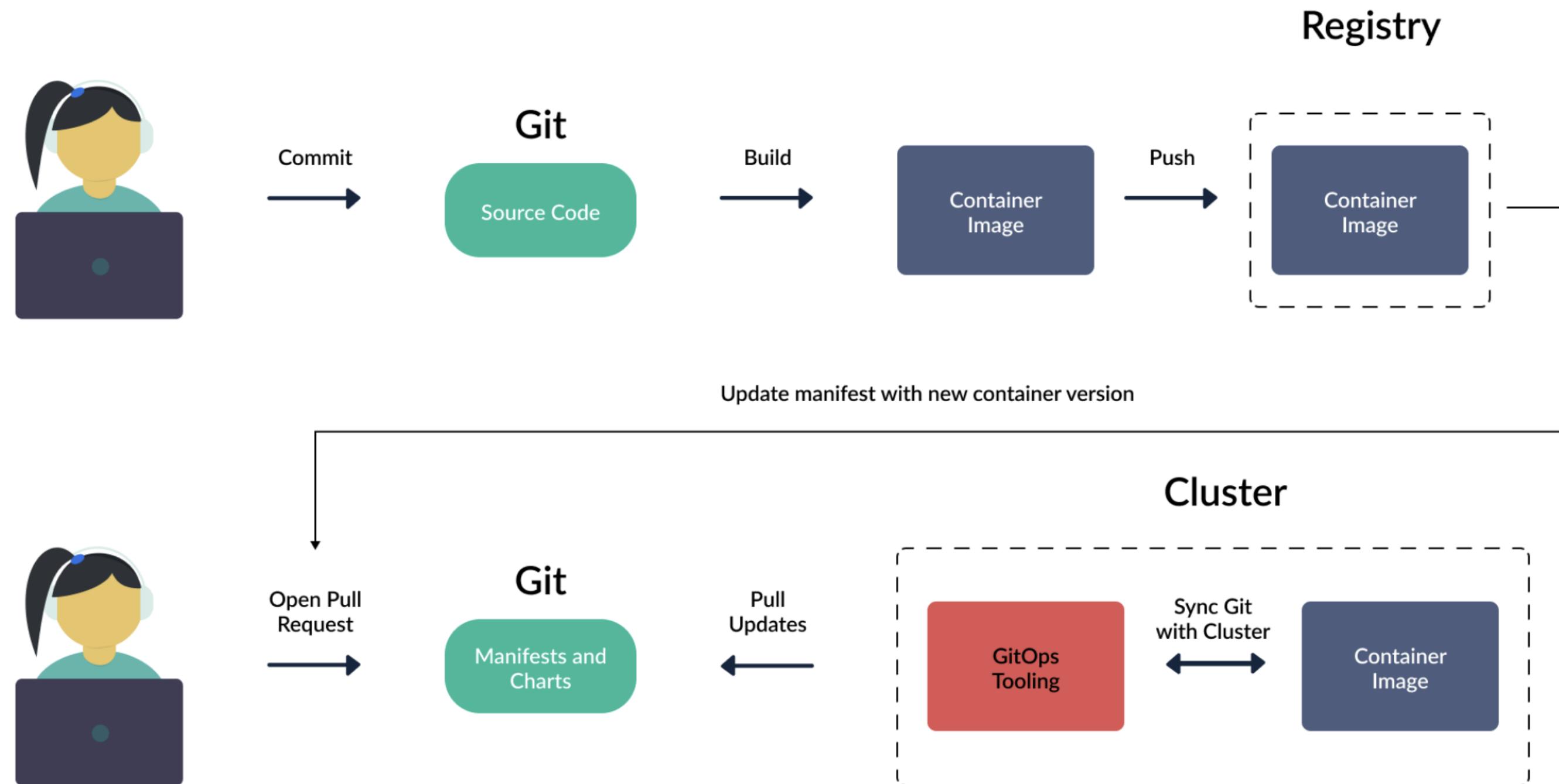
1 - A developer commits source code for the application and the CI system creates a container image that is pushed to a registry.

2 - Nobody has direct access to the Kubernetes cluster.

3 - There is a second Git repository that has all manifests that define the application.

4 - Another human or an automated system changes the manifests in this second Git repository.

5 - A GitOps controller that is running inside the cluster is monitoring the Git repository and as soon as a change is made, it changes the cluster state to match what is described in Git.



The key points here are:

- The state of the cluster is always described in Git. Git holds everything for the application and not just the source code.
- There is no external deployment/CI system with full access to the cluster. The cluster itself is pulling changes and deployment information.
- The GitOps controller is running in a constant loop and always matches the Git state with the cluster state.

# **Introduction to the AWS Cloud Platform**

Ravindu Nirmal Fernando

2x AWS Community Builder | STL @ Sysco LABS

# Agenda

- Introduction to AWS cloud platform and its benefits
- AWS Global Infrastructure
- Accessing AWS Services
- Interacting with AWS Services
- Best Practices for managing AWS Accounts
- Common AWS services
- Demo

# What is AWS Cloud?

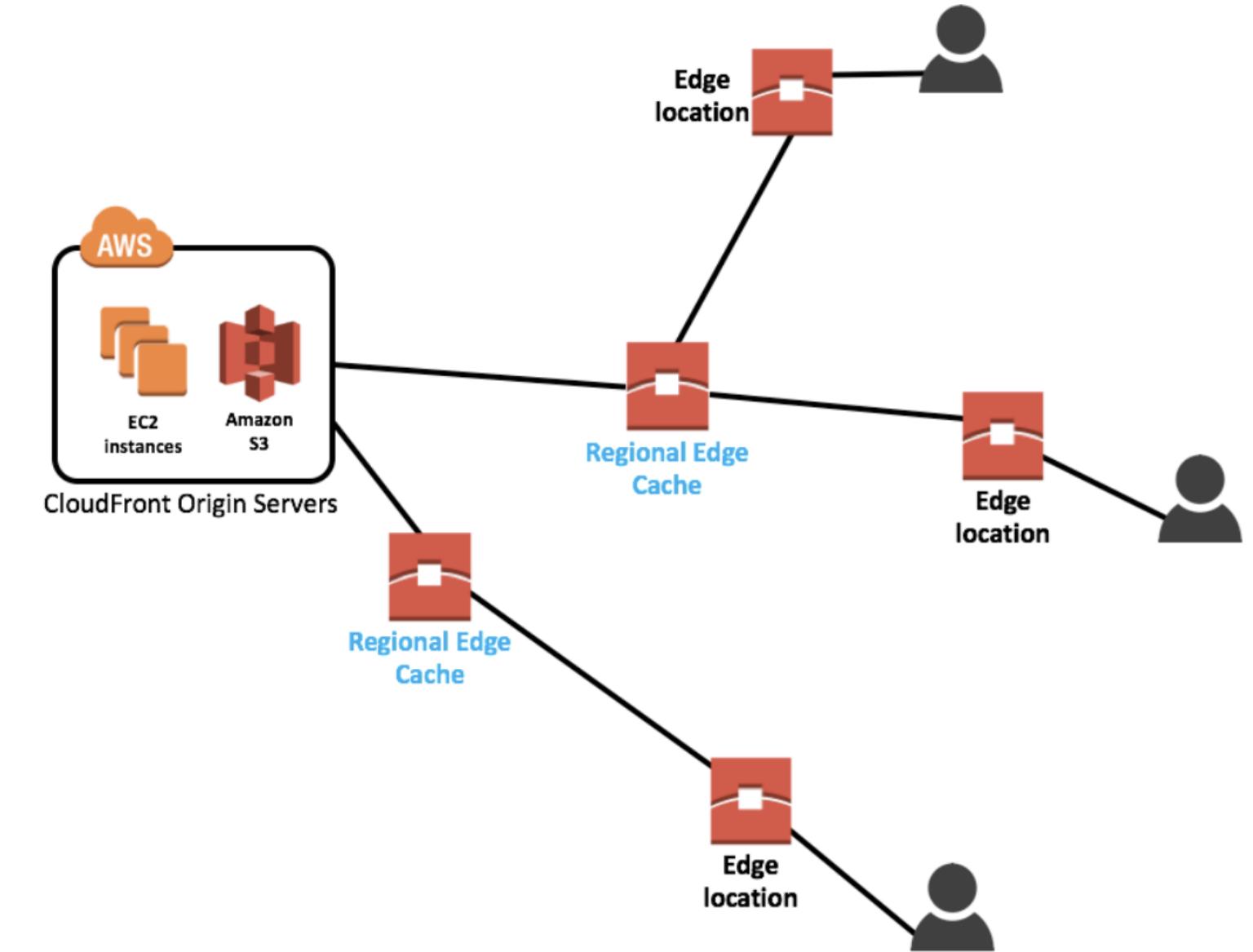
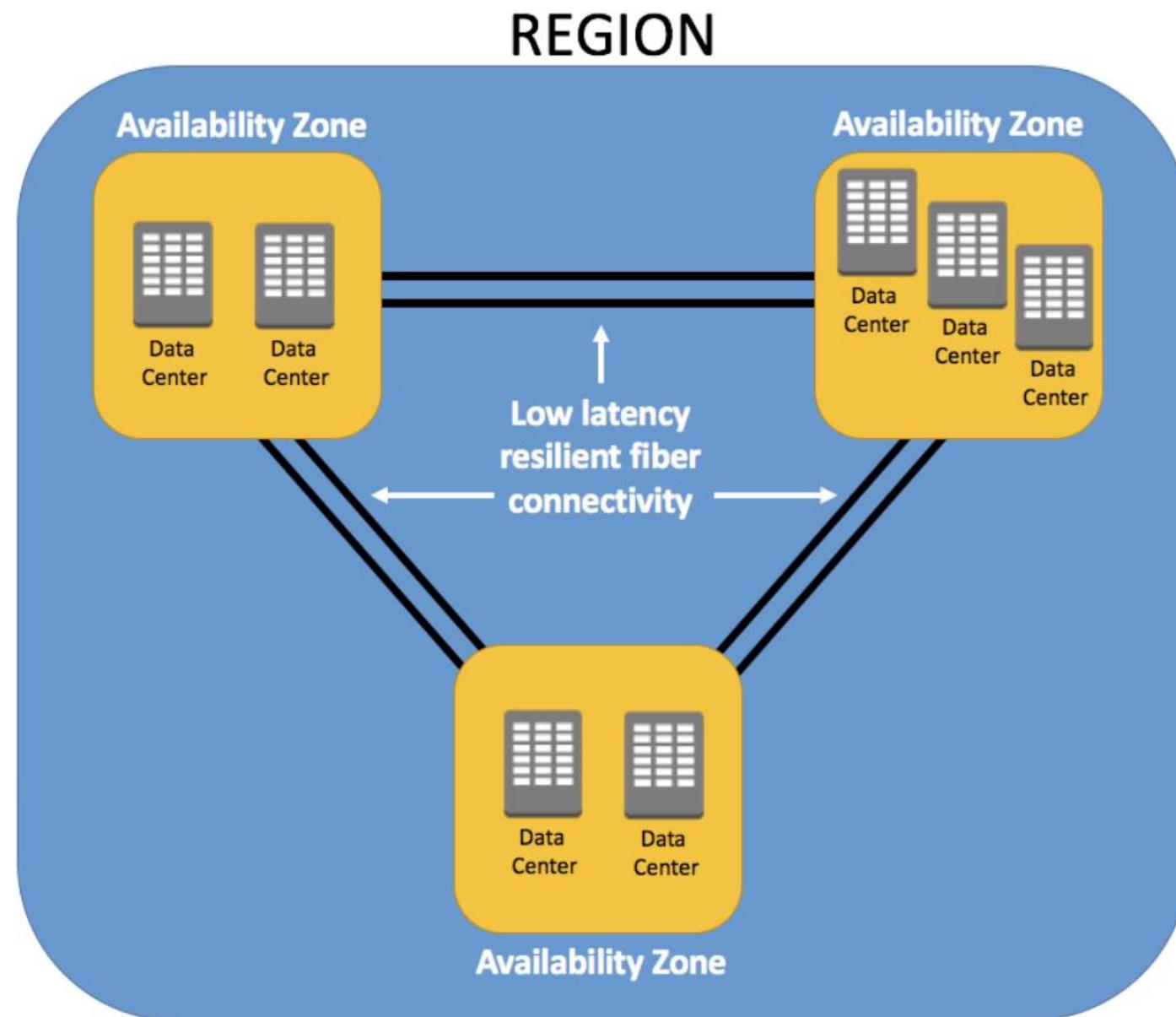
- AWS Cloud is a **cloud computing platform** that provides a wide range of services, including **compute, storage, databases, security, networking, analytics, machine learning, and DevOps etc...**
- AWS Cloud is a **highly scalable and reliable** platform that can be used to **build and deploy** applications of all sizes and complexity.
- AWS Cloud is also a **cost-effective platform**, as you **only pay for the resources that you use**.

# Benefits of using AWS Cloud?

- **Scalability** - Easily add/ remove resources as required
- **Reliability** - Backed by reliable AWS network with proven track record of uptime and performance
- **Cost-effectiveness** - Pay only for what you use
- **Security** - Wide range of security features and services to protect your data
- **Innovation** - 200+ fully featured services for a wide range of technologies, industries, and use cases

# AWS Global Infrastructure

- . AWS Global infrastructure consists of a **network of data centers located around the world**. These data centers are organized into **Regions** and **Availability Zones**.
- . A **Region** is a **geographical area that contains multiple Availability Zones**. An Availability Zone is a logically isolated section of a Region.
- . **AWS Edge Locations** are **locations around the world where AWS content is cached**. This allows users to access AWS content with lower latency and improved performance.
- . **Regional Edge Caches** are **caches of frequently accessed AWS content that are located in close proximity to AWS customers**. This allows users to access AWS content with even lower latency and improved performance.



The AWS Cloud spans 102 Availability Zones within 32 geographic regions around the world, with announced plans for 12 more Availability Zones and 4 more AWS Regions in Canada, Malaysia, New Zealand, and Thailand.



# Accessing AWS Services

- AWS IAM (Identity and Access Management) - service that allows you to manage user access to your AWS resources.
  - IAM allows you to create **Users** and **Groups**, and assign them permissions policies to specific AWS resources. Users have long term credentials.
  - IAM **Roles** - Very similar to a user, in that it is an identity with permission policies that determine what the identity can and cannot do in AWS. But no credentials.
  - IAM Policies - Documents that specify the permissions that are granted to users, groups, or roles. Used to determine what actions a user, role, or member of a user group can perform, on which AWS resources, and **under what conditions**. determine what actions a user, role, or member of a user group can perform, on which AWS resources, and under what conditions.

# Interacting with AWS Services

- AWS Management Console
- AWS Command Line Interface
- Software Development Kits

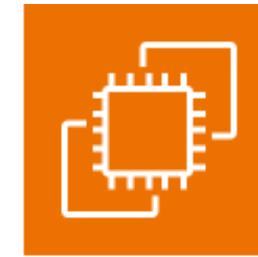
mine what actions a user, role, or member of a user group can perform, on which AWS resources, and under what conditions.

# **Best Practices for managing AWS Accounts**

- Use strong passwords, enable password policy and enable multi-factor authentication.
- Create IAM users and roles and assign them permissions to specific AWS resources.
- Use security groups, Network Access Controls and VPCs to protect your resources.
- Implement monitoring and logging to track your AWS usage and identify potential problems.

# Common AWS Services

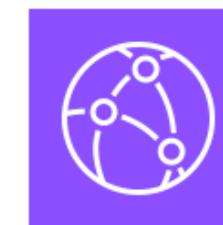
- Compute
  - Amazon Elastic Compute Cloud (EC2)
  - Amazon Elastic Container Service (ECS)
  - AWS Lambda
- Storage
  - Amazon Simple Storage Service (S3)
  - Amazon Elastic Block Store (EBS)
  - Amazon Elastic File System (EFS)



mine what actions a user, role, or member of a user group can perform, on which AWS resources, and under what conditions. This is called an IAM policy.

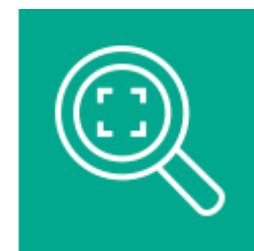
# Common AWS Services

- Databases
  - Amazon Relational Database Service (RDS)
  - Amazon DynamoDB
  - Amazon Aurora
- Networking and Content Delivery
  - Amazon Virtual Private Cloud (VPC)
  - Amazon Route 53
  - Amazon CloudFront



# Common AWS Services

- Analytics
  - Amazon Redshift
  - Amazon Athena
  - Amazon Kinesis
- Machine Learning
  - Amazon SageMaker
  - Amazon Rekognition
  - Amazon Comprehend
  - Amazon BedRock



mine what actions a user, role, or member of a user group can perform, on which AWS resources, and under what conditions.

# Common AWS Services

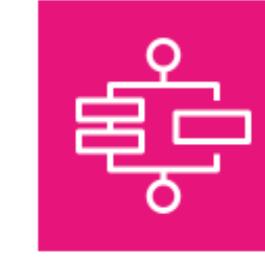
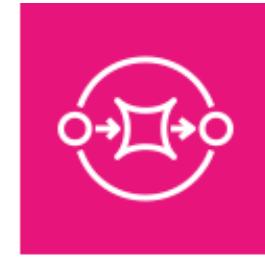
- DevOps
  - AWS CodePipeline
  - AWS CodeDeploy
- Management & Governance
  - AWS CloudFormation
  - Amazon CloudWatch
  - Amazon CloudTrail



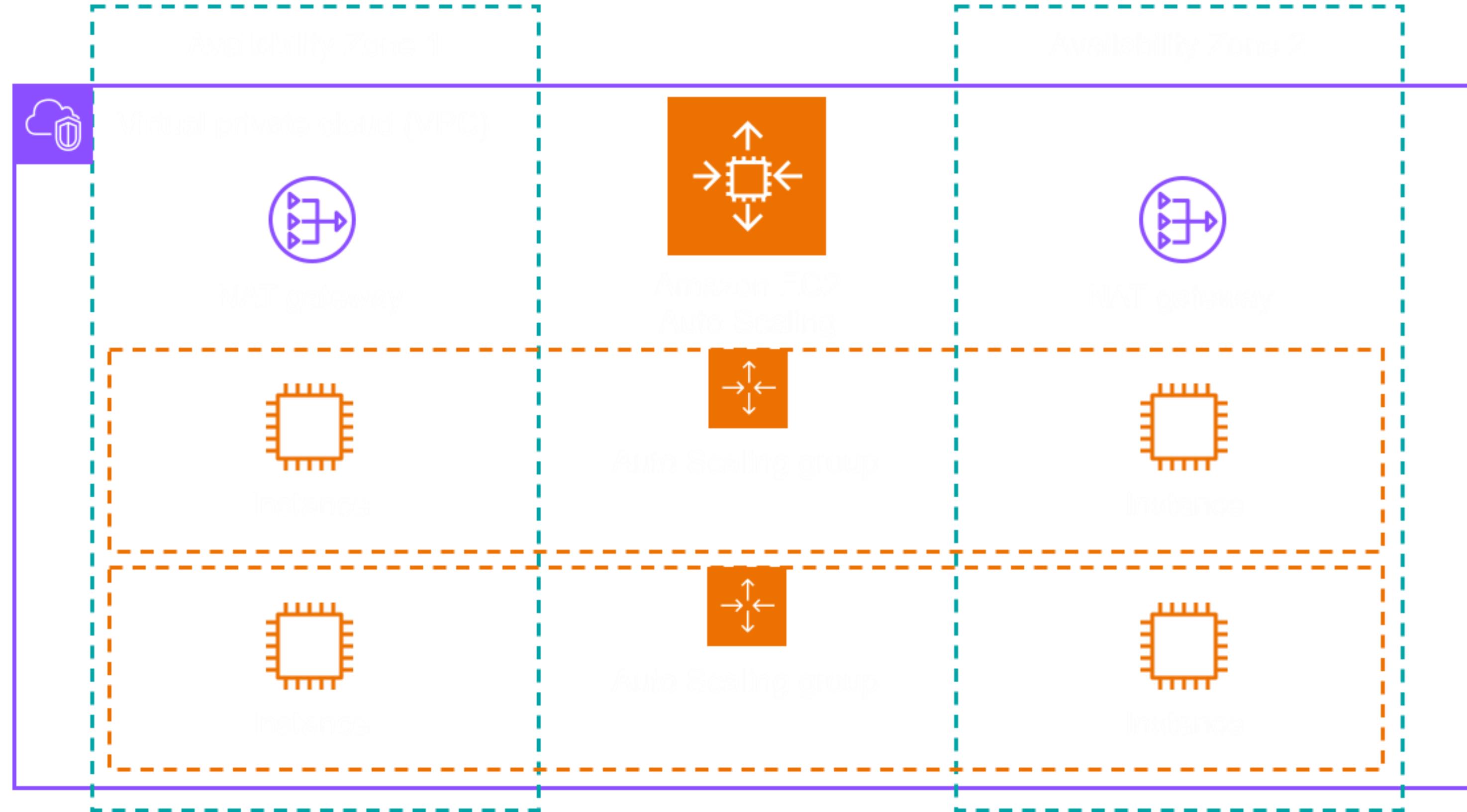
mine what actions a user, role, or member of a user group can perform, on which AWS resources, and under what conditions.

# Common AWS Services

- Application Integration
  - Amazon SNS
  - Amazon SQS
  - Amazon EventBridge
  - AWS Step Functions

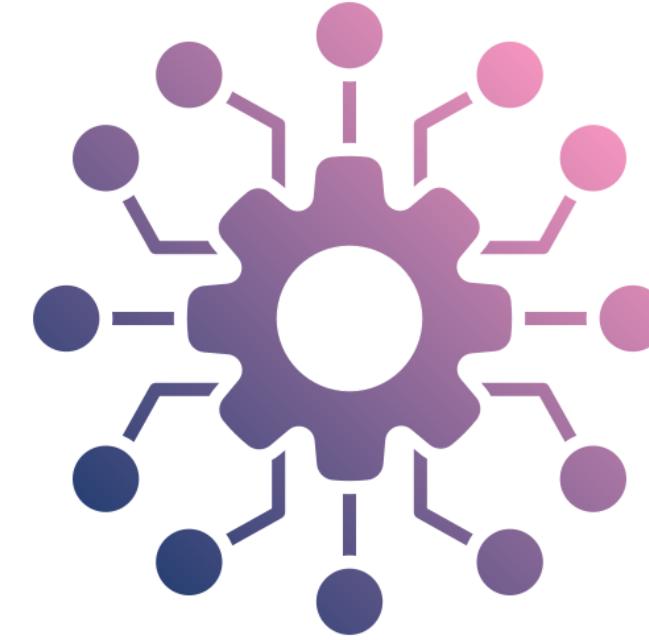


mine what actions a user, role, or member of a user group can perform, on which AWS resources, and under what conditions.



**DEMO TIME...**

**Thank You!!!**



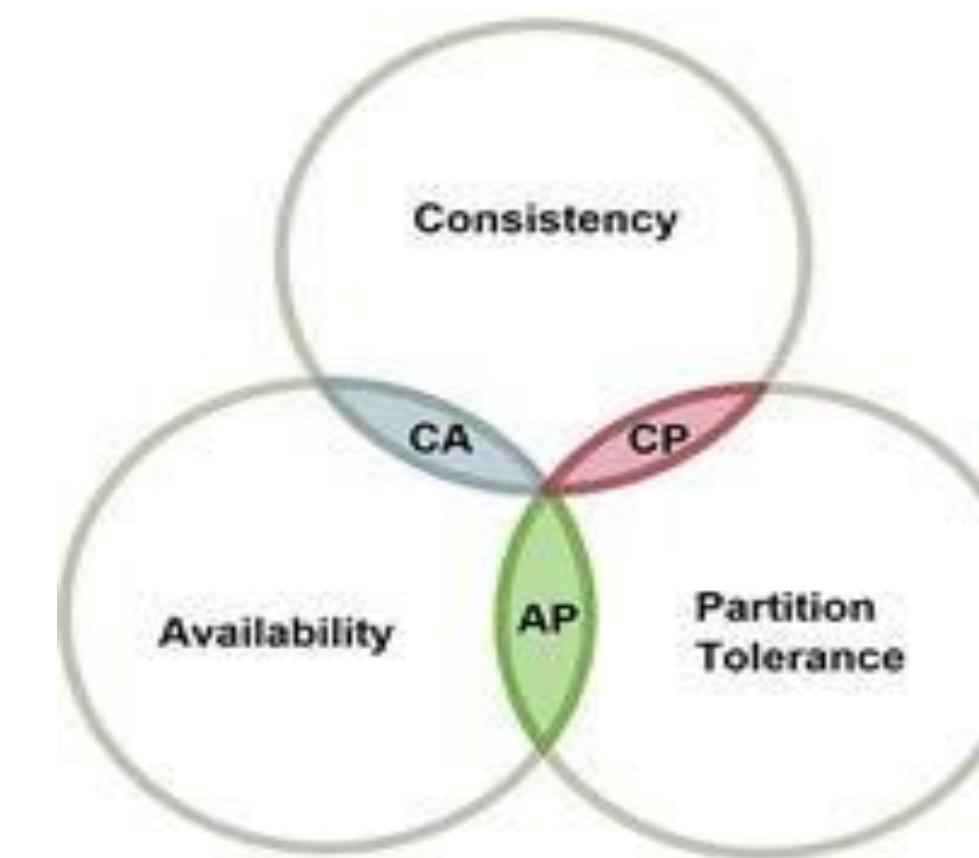
# Introduction to CAP Theorem

Ravindu Nirmal Fernando

SLIIT | March 2025

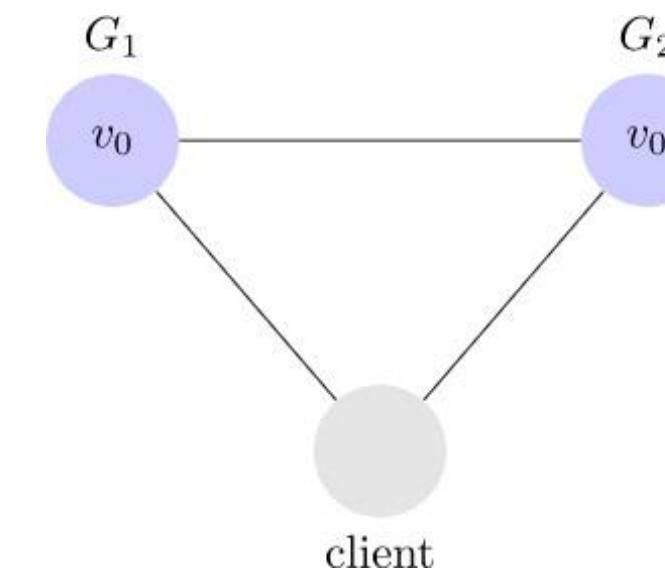
# CAP THEOREM

- A fundamental theorem in distributed systems.
- Can have at most two of the following three properties,
  - Consistency
  - Availability
  - Partition Tolerance



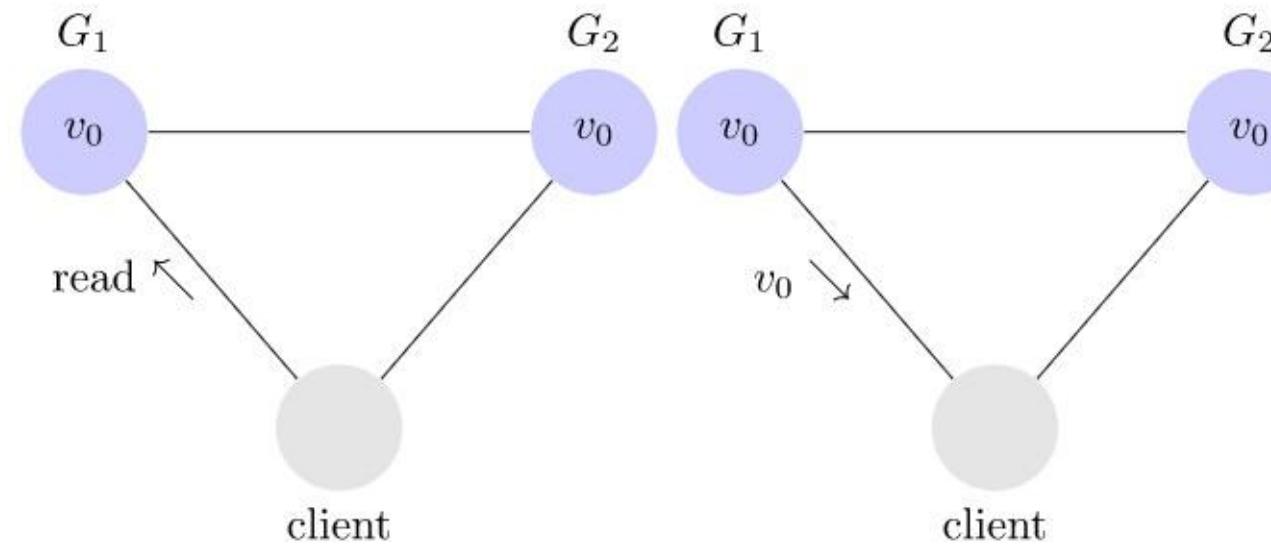
# DISTRIBUTED SYSTEM

- ▶ Consider a simple distributed system with two servers,  $G_1$  and  $G_2$ 
  - ▶ The servers can communicate with each other and connect to remote clients

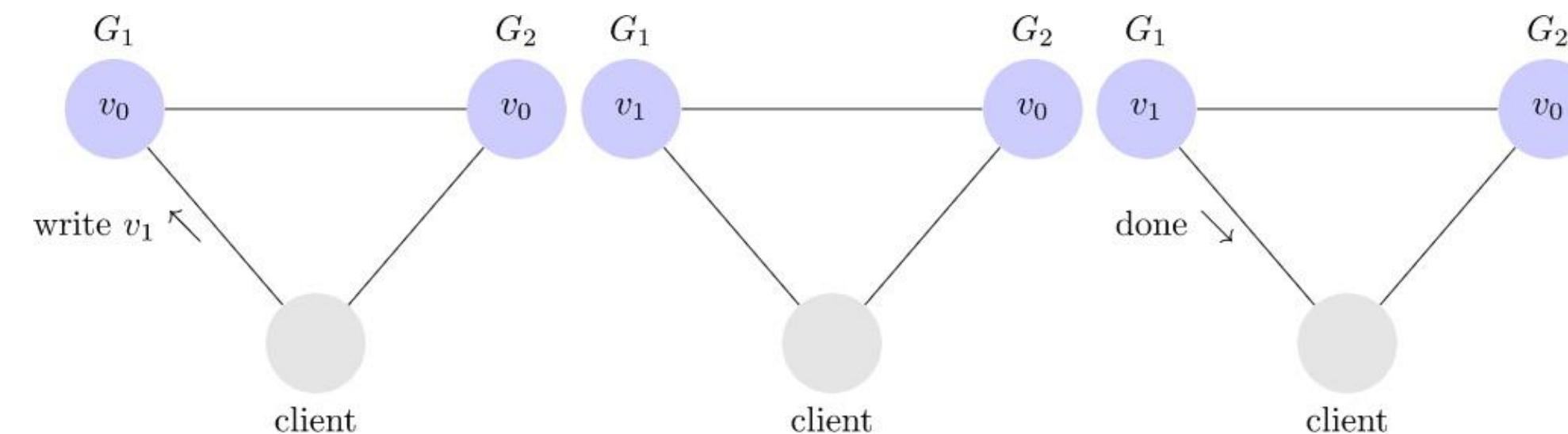


# DISTRIBUTED SYSTEM

## ▶ Read example



## ▶ Write example



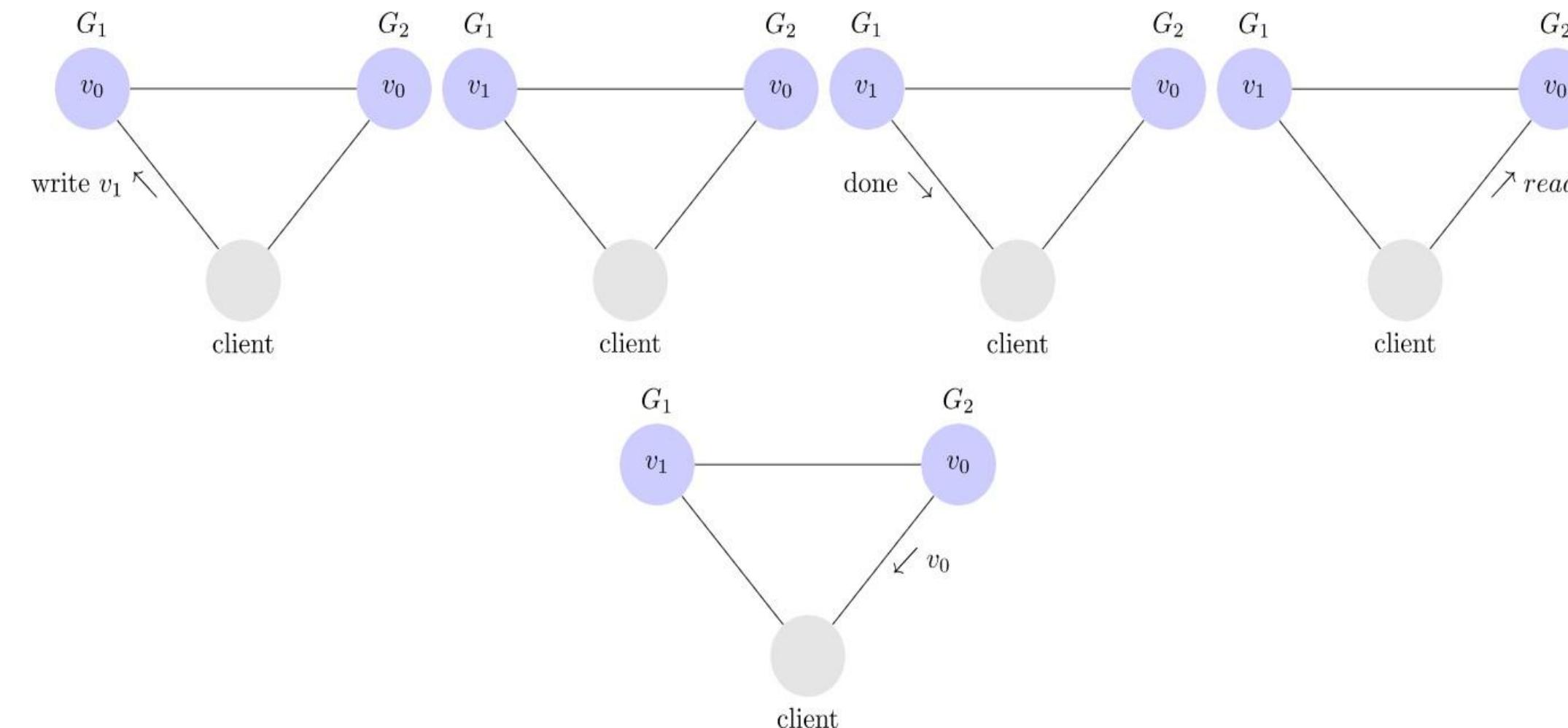
# CONSISTENCY

Consistency means that all clients see the same data at the same time, no matter which node they connect to.

For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed ‘successful.’

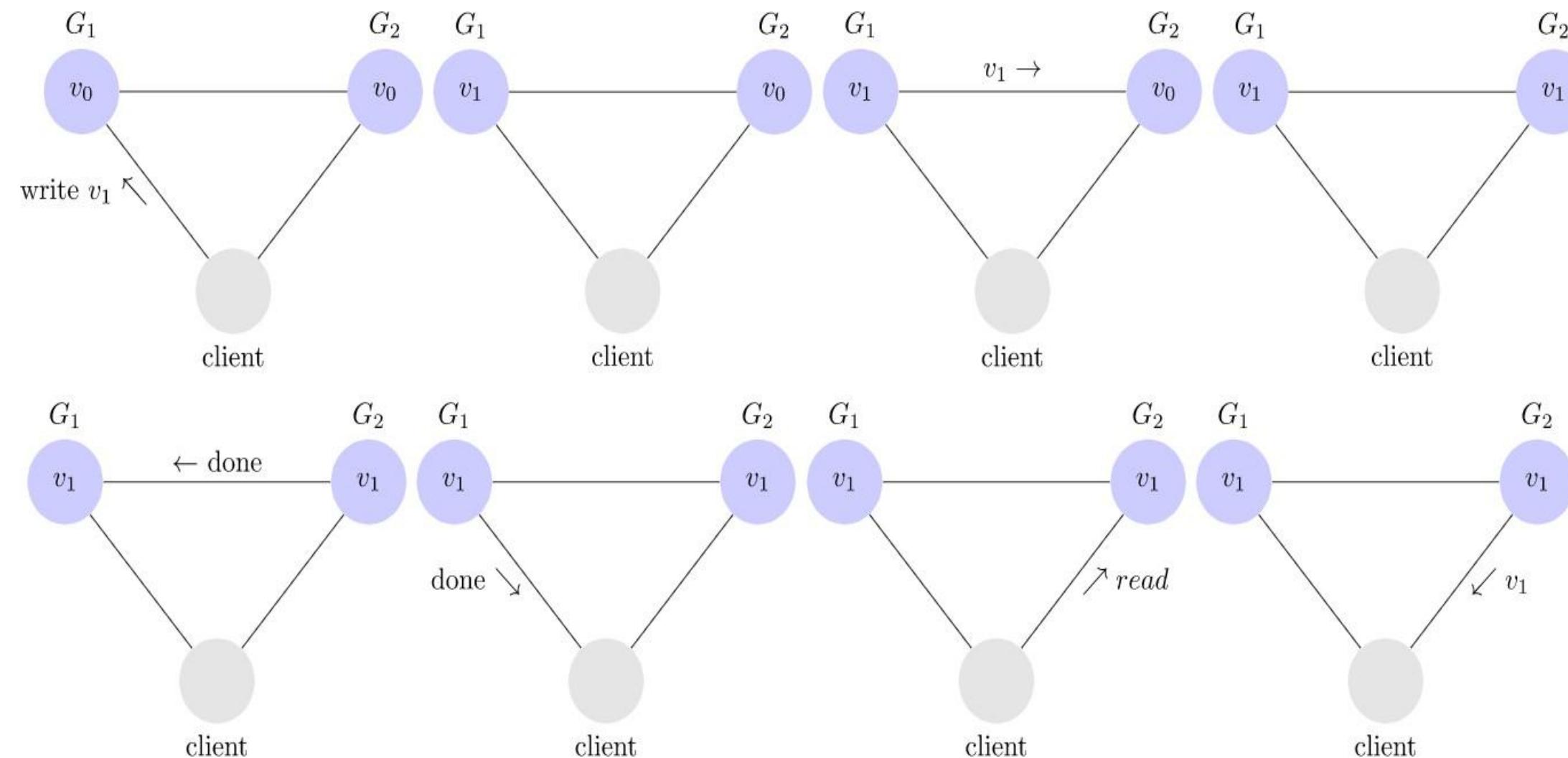
# CONSISTENCY

## Inconsistent system



# CONSISTENCY

## Consistent system



# AVAILABILITY

Availability means that any client making a request for data gets a response, even if one or more nodes are down.

Another way to state this—all working nodes in the distributed system return a valid response for any request, without exception.

# PARTITION TOLERANCE

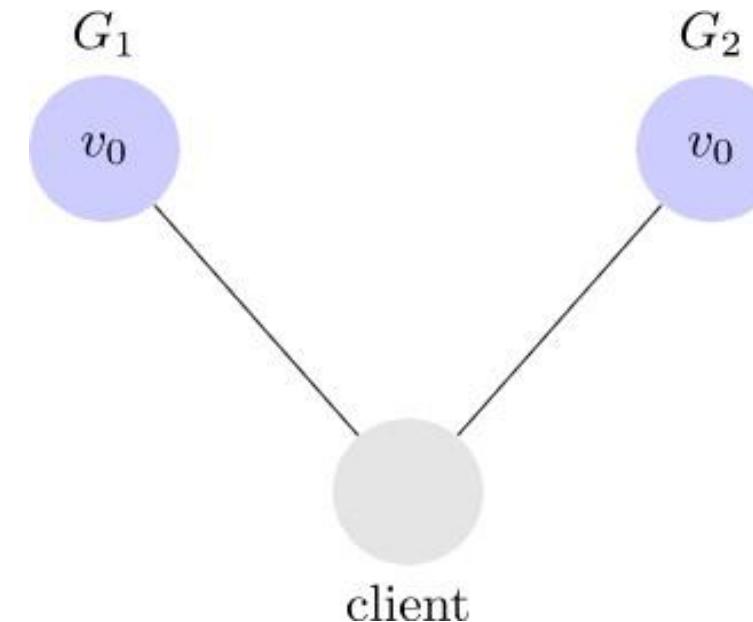
A **partition** is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes.

**Partition tolerance** means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.

# PARTITION TOLERANCE

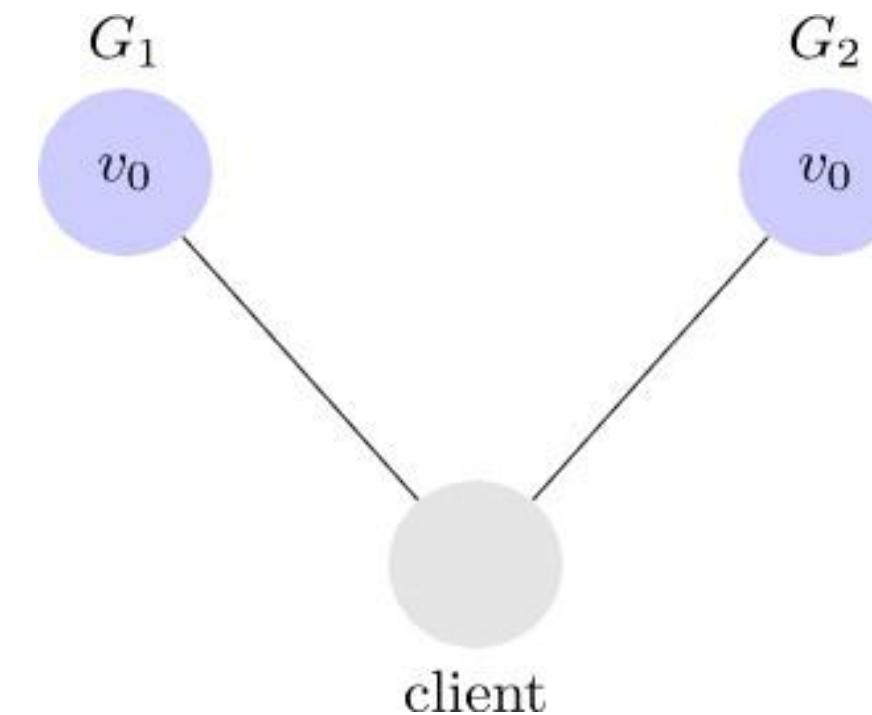
The system continues to operate despite network partitions

- Communication among the servers is not reliable
- Servers may be partitioned into multiple groups that cannot communicate with each other
- Messages may be delayed or lost forever



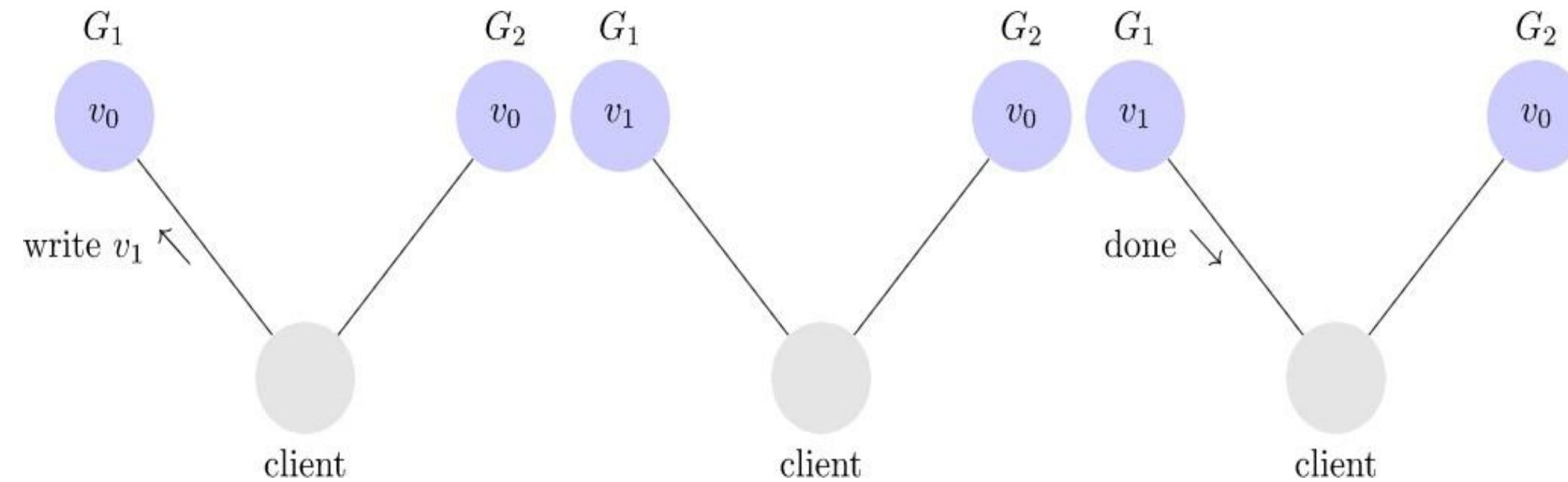
# PROOF

Consider partitioned system,



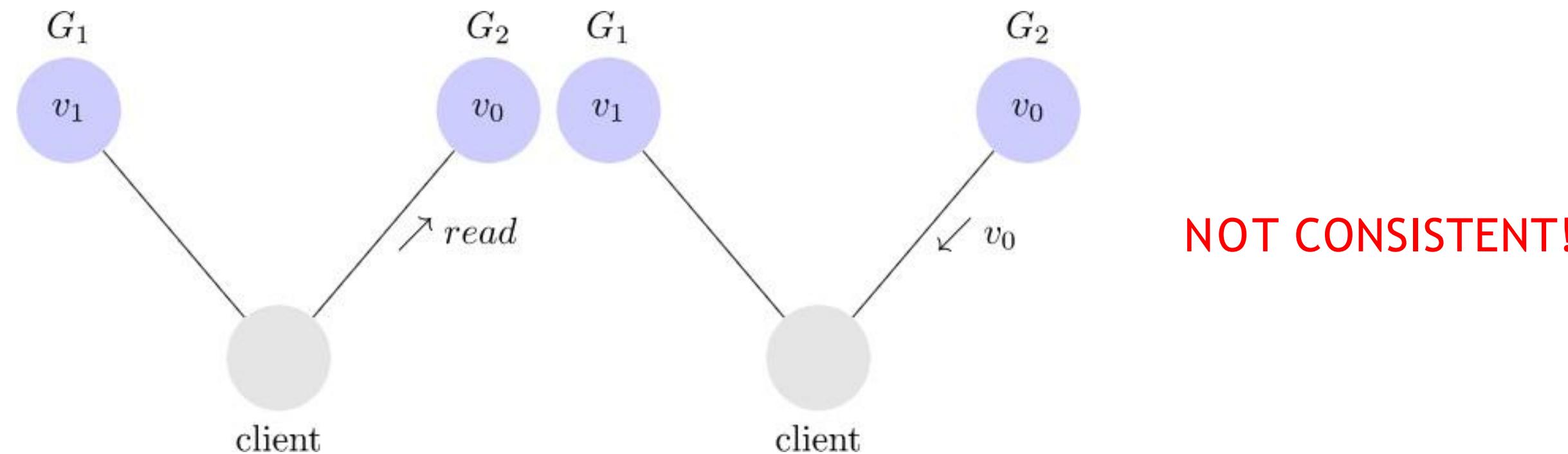
# PROOF

Client writes to  $G_1$



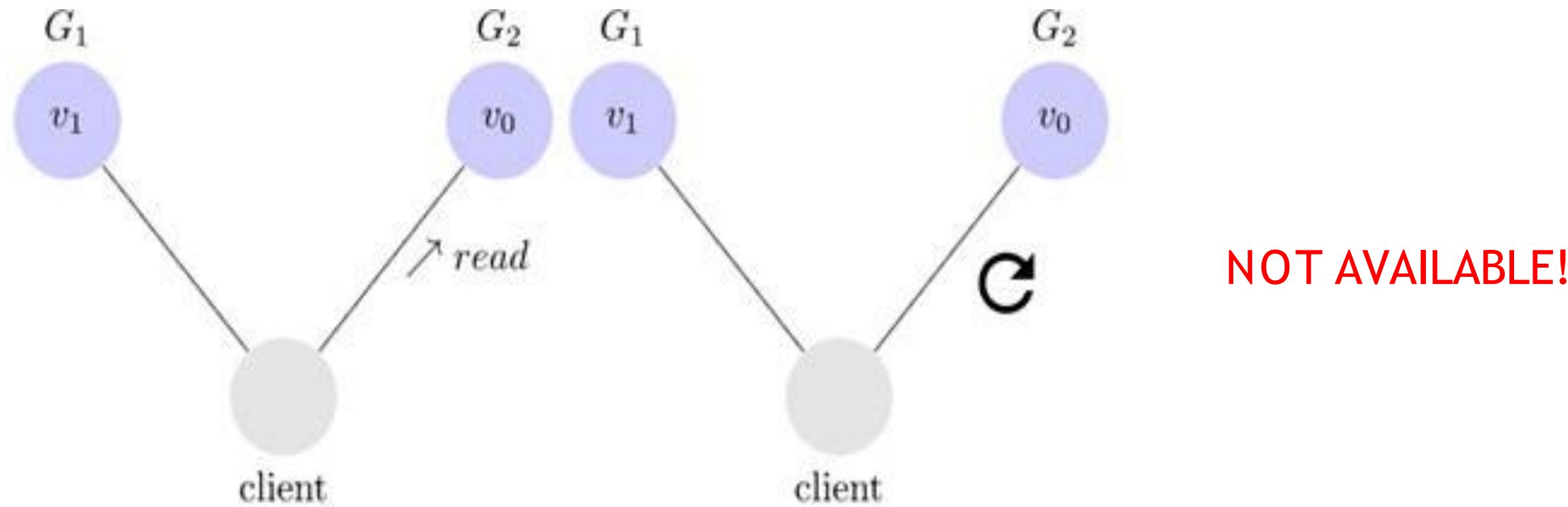
# PROOF

Client reads from  $G_2$



# PROOF

Client reads from  $G_2$



# SYSTEM DESIGN

This suggests there are three kinds of distributed systems

- CP
- CA
- AP

Why is it important?

- Future of databases is distributed (Big Data Trend)
- CAP theorem describes the trade-offs involved in distributed systems
- Proper understanding of CAP theorem is essential to making decisions about distributed database/system design
- Misunderstanding can lead to erroneous or inappropriate design choices

# CONSISTENCY OR AVAILABILITY

- Consistency & Availability is not “binary” decision
- AP systems relax consistency in favor of availability - but are not inconsistent
- CP systems sacrifice availability for consistency- but are not unavailable
- This suggests both AP & CP systems can offer a degree of consistency, & availability, as well as partition tolerance

# CONSISTENCY MODELS

## Strong Consistency

- After the update completes, any subsequent access will return the same updated value.

## Weak Consistency

- It is not guaranteed that subsequent accesses will return the updated value.

## Eventual Consistency

- Specific form of weak consistency
- It is guaranteed that if no new updates are made to object, eventually all accesses will return the last updated value (e.g., propagate updates to replicas in a lazy fashion)

## EVENTUAL CONSISTENCY – FACEBOOK EXAMPLE

- Bob finds an interesting story and shares with Alice by posting on her Facebook wall
- Bob asks Alice to check it out
- Alice logs in her account, checks her Facebook wall but
  - Nothing is there!



## EVENTUAL CONSISTENCY – FACEBOOK EXAMPLE

- Bob tells Alice to wait a bit and check out later
- Alice waits for a minute or so and checks back - **Finds the wall post!**



## EVENTUAL CONSISTENCY – FACEBOOK EXAMPLE

- Why would Facebook choose an eventual consistent model over the strong consistent one?
  - Facebook has billions of active users
  - It is non-trivial to efficiently and reliably store the huge amount of data generated at any given time
  - Eventual consistent model offers the option to reduce the load and improve availability

## DYNAMIC TRADEOFF BETWEEN C AND A

An airline reservation system:

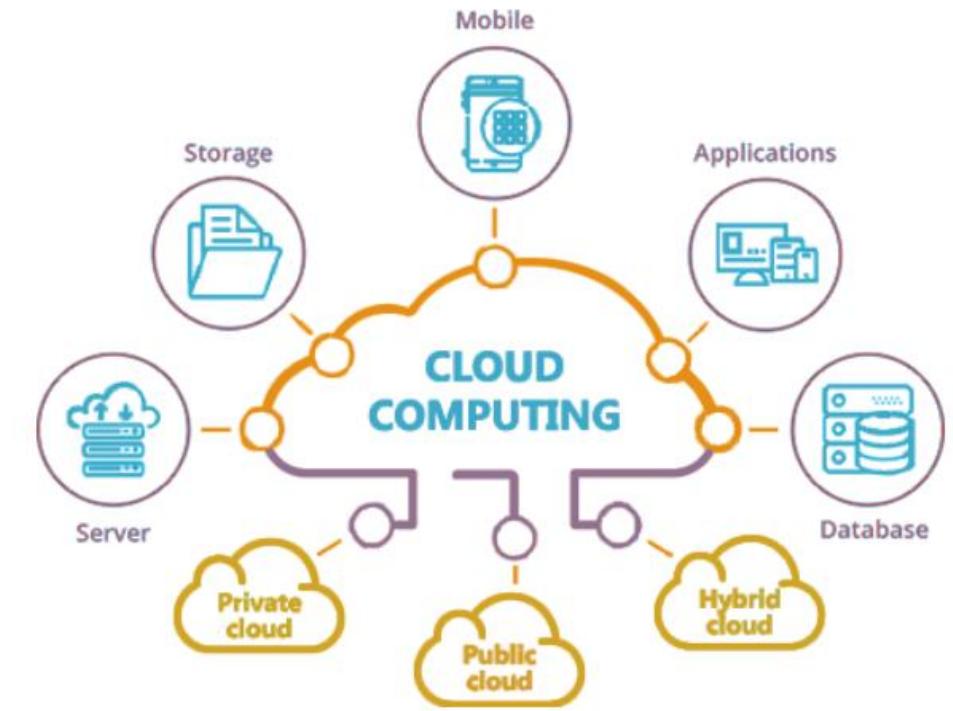
- When most of seats are available: it is ok to rely on somewhat out-of-date data, availability is more critical
- When the plane is close to be filled: it needs more accurate data to ensure the plane is not overbooked, consistency is more critical

Neither strong consistency nor guaranteed availability, but it may significantly increase the tolerance of network disruption

## REFERENCES

- Gilbert, Seth, and Nancy Lynch. "Perspectives on the CAP Theorem." Computer 45.2 (2012):30-36
- [https://mwhittaker.github.io/blog/an illustrated proof of the cap theorem/](https://mwhittaker.github.io/blog/an_illustrated_proof_of_the_cap_theorem/)
- ---

[theorem # :~:text=The% 20CAP% 20theorem% 20says% 20that,'P'% 20in% 20CAP](#)



# Key Essentials for Building Apps in Cloud

Ravindu Nirmal Fernando  
SLIIT | March 2025

# Shared Responsibility Model in Public Cloud

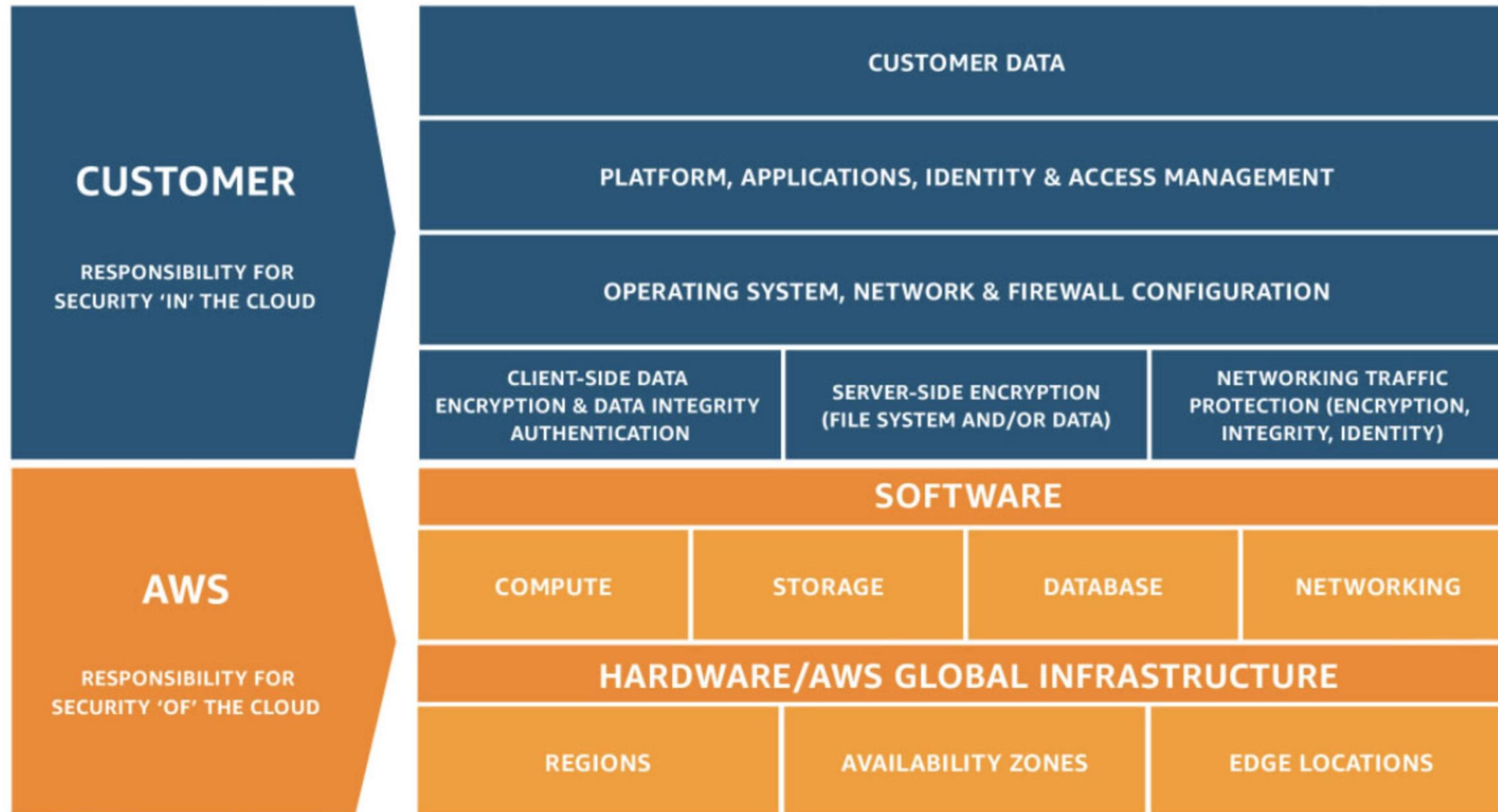
- A framework that outlines how security responsibilities are divided between the cloud service provider and the cloud user.
- **Security and compliance** is a shared responsibility between AWS and the customer. Cloud service provider manages the infrastructure, while customers are responsible for managing their data and applications.
- Cloud Service Provider (CSP) Responsibility: Known as "**Security of the Cloud.**" CSP is in charge of the infrastructure, including hardware, software, networking, and physical security.
- Customer Responsibility: Termed "**Security in the Cloud.**" Customers handle the guest operating system, application software, and AWS-provided firewall configuration.

# Key aspects of Shared Responsibility Model

- **Service/ Delivery Models:** Responsibilities vary depending on whether the service is IaaS (like EC2), PaaS, or SaaS.
- **IT Controls:** Shared management of IT controls between CSP and customers. CSP manages physical infrastructure controls, while customers handle specific application-level controls.

# Control types in Shared Responsibility Model

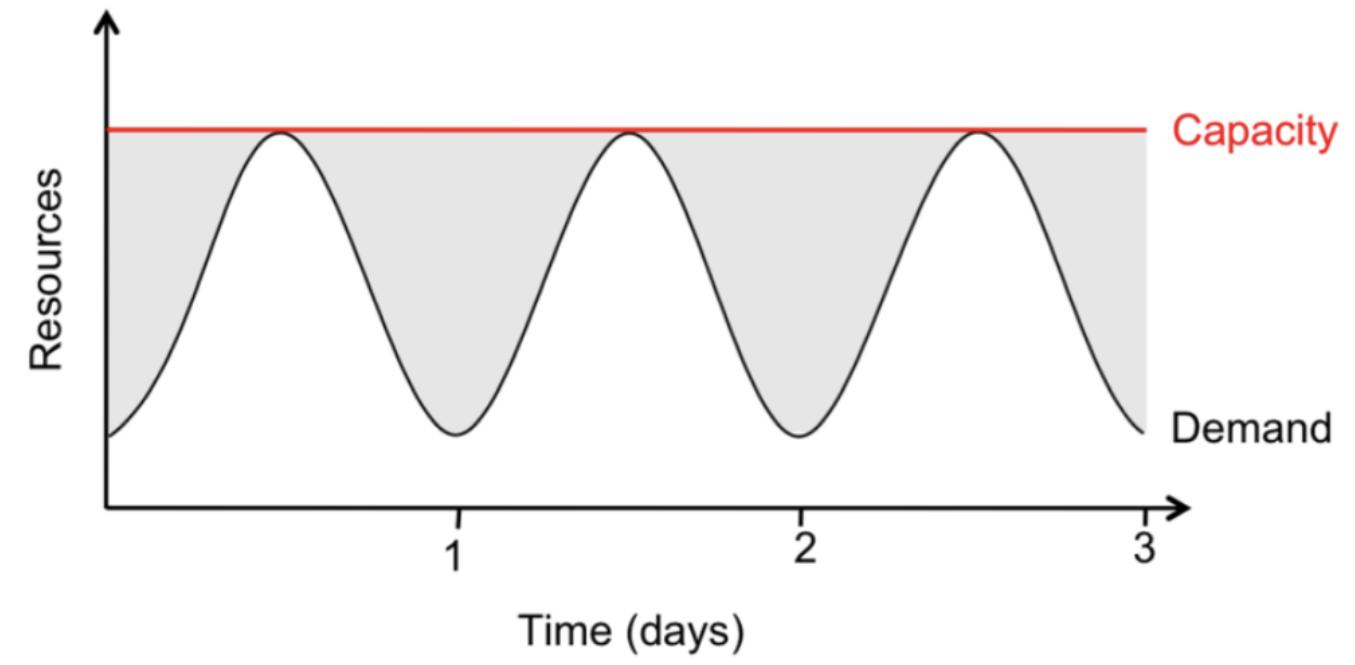
- **Inherited Controls:** Controls fully managed by CSP (e.g., physical and environmental controls).
- **Shared Controls:** Controls that apply to both CSP and customers but in different contexts (e.g., patch management, configuration management).
  - Patch Management – CSP is responsible for patching and fixing flaws within the infrastructure, but customers are responsible for patching their guest OS and applications.
  - Configuration Management – CSP maintains the configuration of its infrastructure devices, but a customer is responsible for configuring their own guest operating systems, databases, and applications.
  - Awareness & Training - CSP trains CSP's employees, but a customer must train their own employees.
- **Customer Specific Controls:** Controls solely managed by the customer, depending on their applications and use of CSP services.



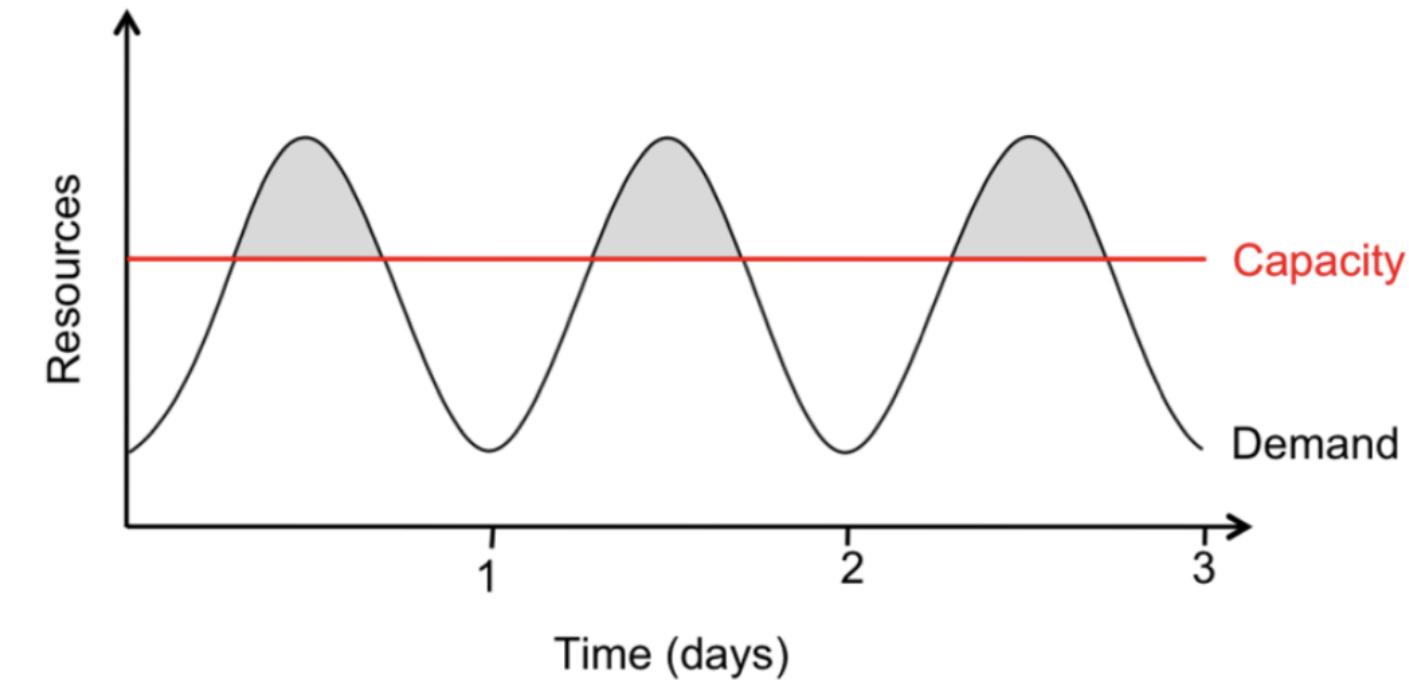
# How to use Shared Responsibility Model practically...

- Understanding the Model: Customers need to comprehend the CSP Shared Responsibility Model and its general application in cloud operations.
- Application to Use Case: Determine the model's relevance to their specific use case.
- Variability in Responsibility: Customer responsibility changes based on:
  - The choice of CSP services and geographical locations. (e.g: AWS EC2 in specified AWS region)
  - How these services integrate into their IT environment.
- Legal and Regulatory Considerations: Consideration of laws and regulations that apply to their organization and workload.

# Resource Provisioning



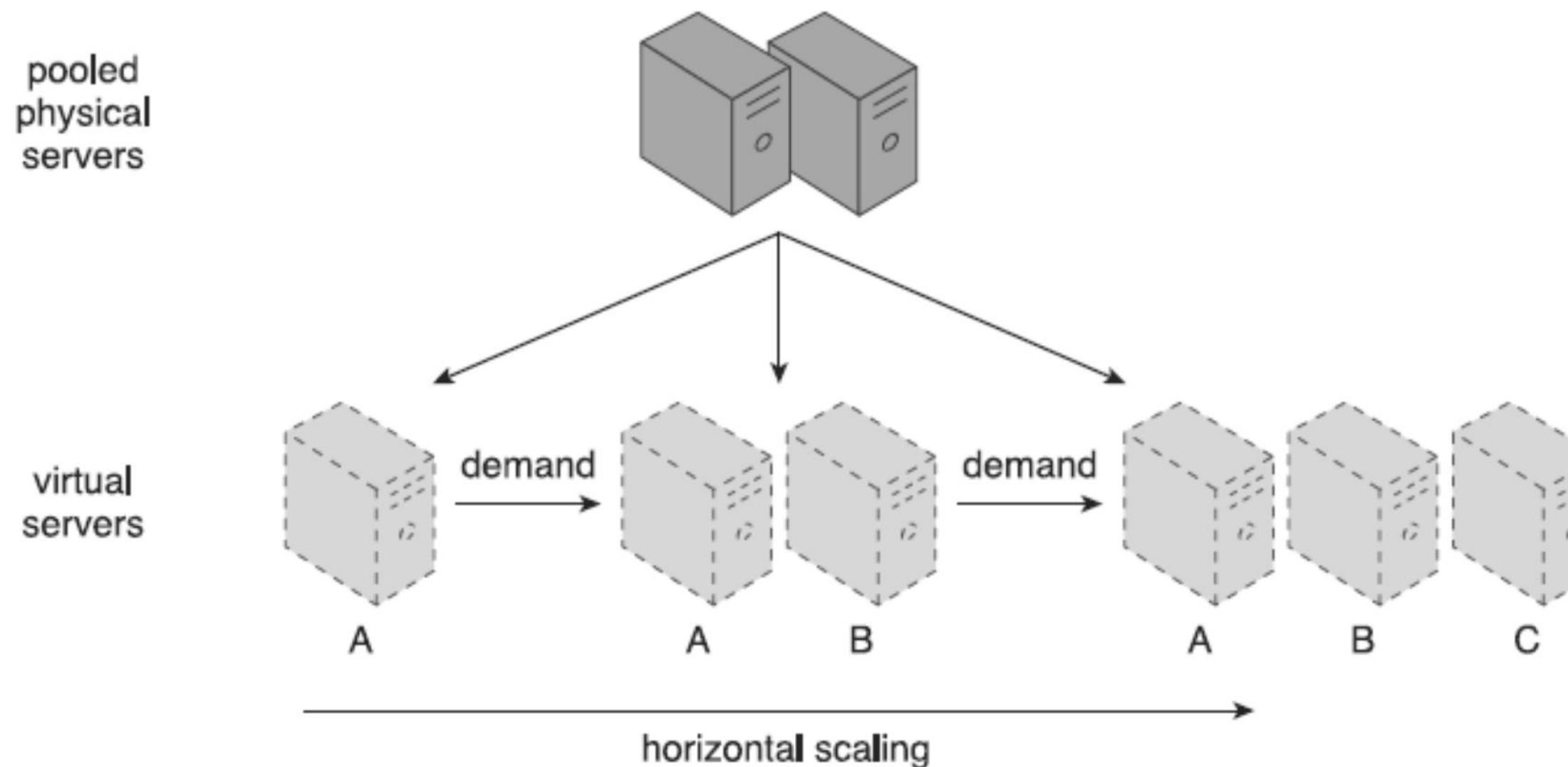
**Provisioning for Peak Load**



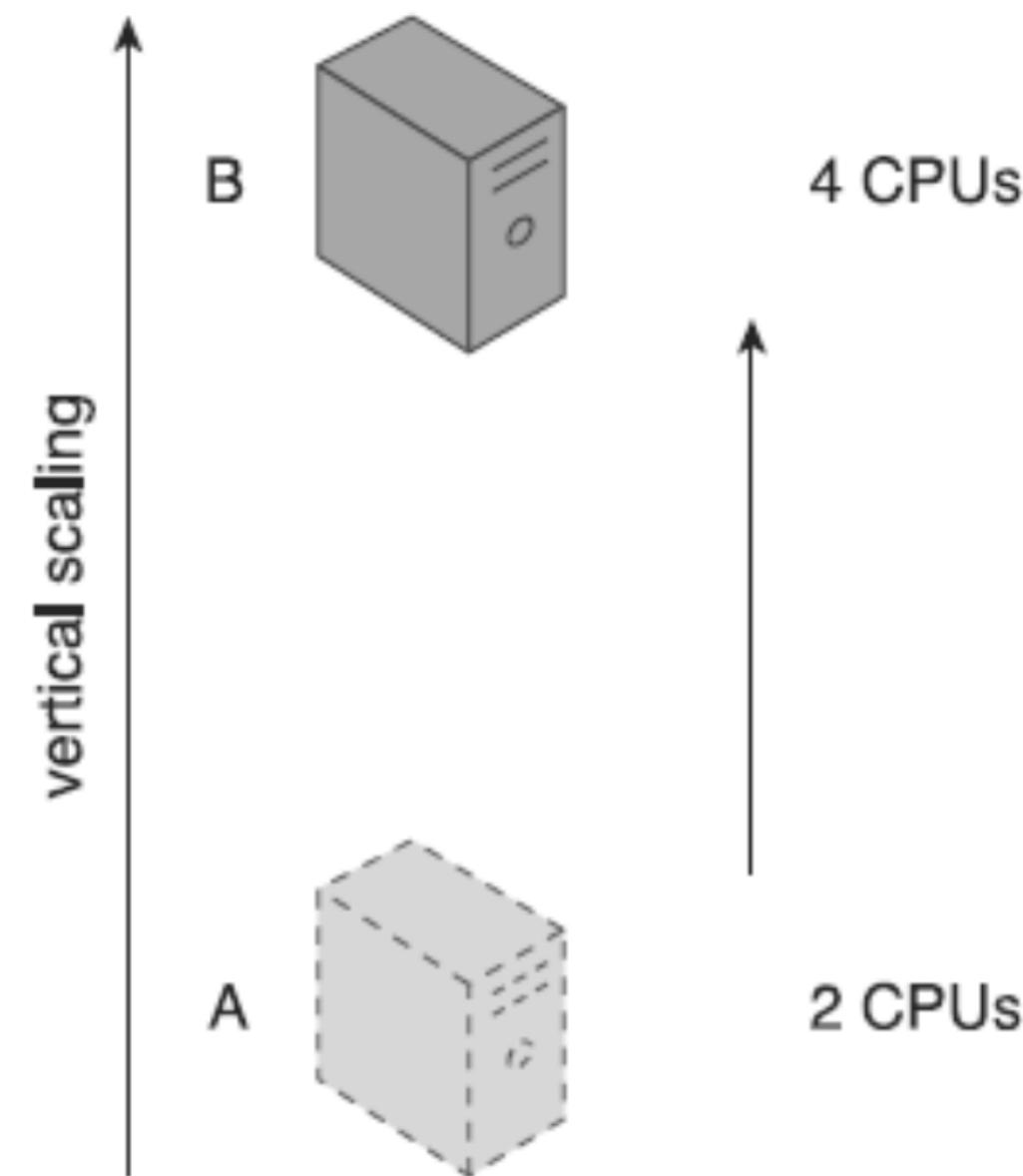
**Under-Provisioning**

# Scalability in Cloud

- Horizontal scaling
  - The allocating or releasing of IT resources that are of the same type
  - Scaling in and out



- Vertical scaling
  - Existing IT resource is replaced by another with higher or lower capacity
  - Scaling up & down



# Horizontal vs Vertical Scaling

Aspect	Horizontal Scaling (Scaling Out/In)	Vertical Scaling (Scaling Up/Down)
Definition	Adding or removing servers to adjust capacity.	Increasing or decreasing the capacity of a server.
Cost	Can be more cost-effective with pay-as-you-go models.	May involve higher costs due to high-end hardware.
Downtime	Often allows scaling with no downtime.	May require downtime for hardware upgrades.
Resource Limits	Limited by the number of servers you can add.	Limited by the maximum capacity of a single server.
Complexity	Can increase architectural complexity.	Simpler, as it involves a single resource.
Availability	Improved, as load is distributed across multiple servers.	Risk of a single point of failure.
Use Case	Ideal for distributed systems and microservices.	Suited for applications with fixed or known peaks.

- Reactive scaling
  - Once something (e.g., workload) happen
- Proactive scaling
  - Based on predictions (e.g., workload)

Based on

- Rules
  - Spawn a new VM if ave. CPU util. > 80%
- Models based on QoS/SLA targets
  - No of VMs to maintain latency < 300 ms

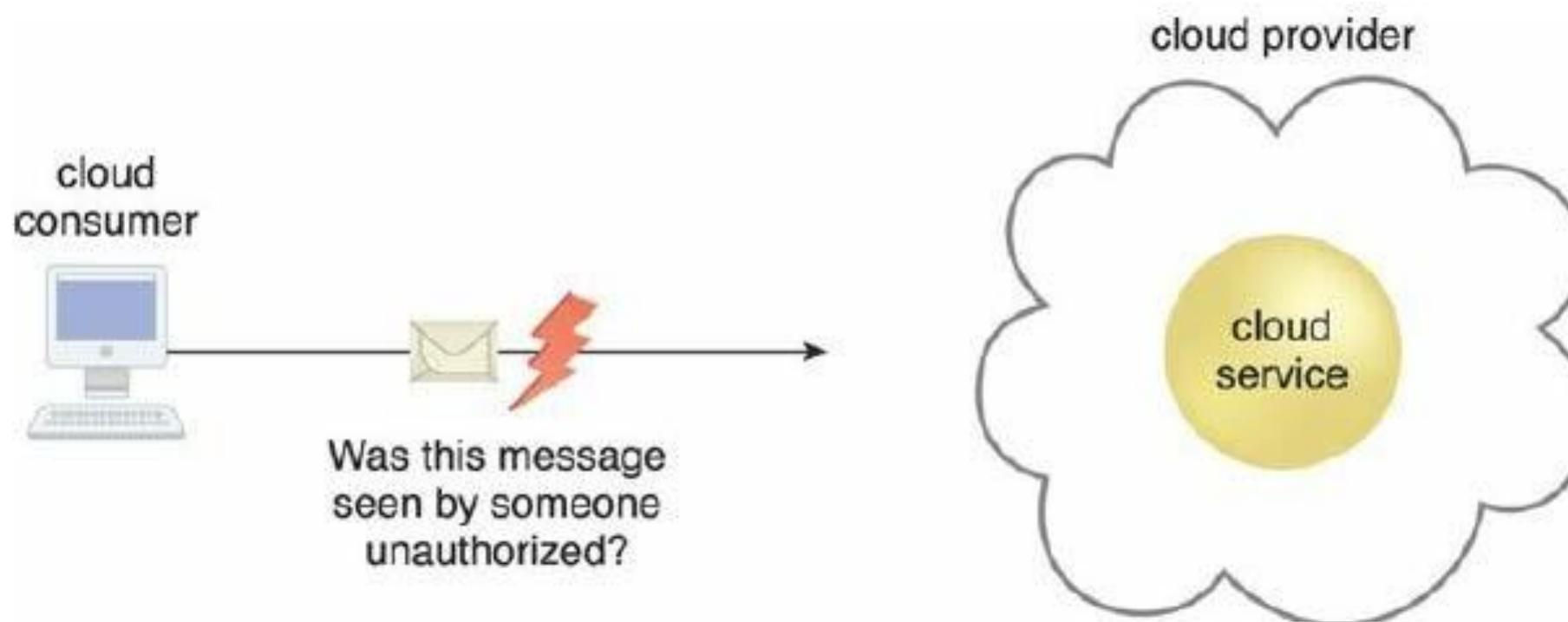
# Cloud Security Basics and Countermeasures

Based on Cloud Computing: Concepts, Technology Architecture, Thomas Erl, et al., Prentice-Hall, 2013,

# Concepts

## Confidentiality

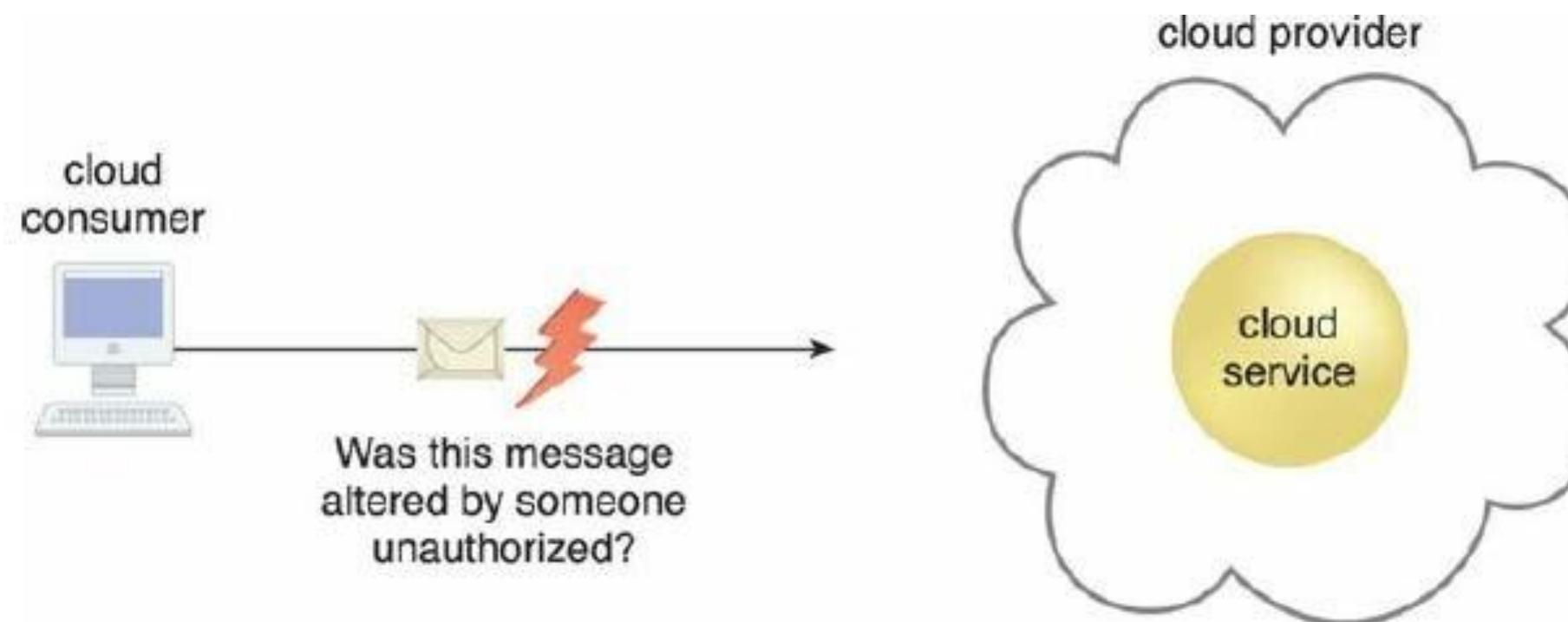
- Accessible only to authorized parties
- Within cloud environments, confidentiality targets to restricting access to data in transit and storage.



# Concepts

## Integrity

- Not having been altered by an unauthorized party
- Can cloud consumer be guaranteed transmitted data to matches the data received.
- Extends to how data is stored, processed, and retrieved.



# Concepts

## Authenticity

- Ensuring something has been provided by an authorized source.
- Can cloud consumer guarantee the authentication of an interaction and no other party can deny or challenge that.

## Availability

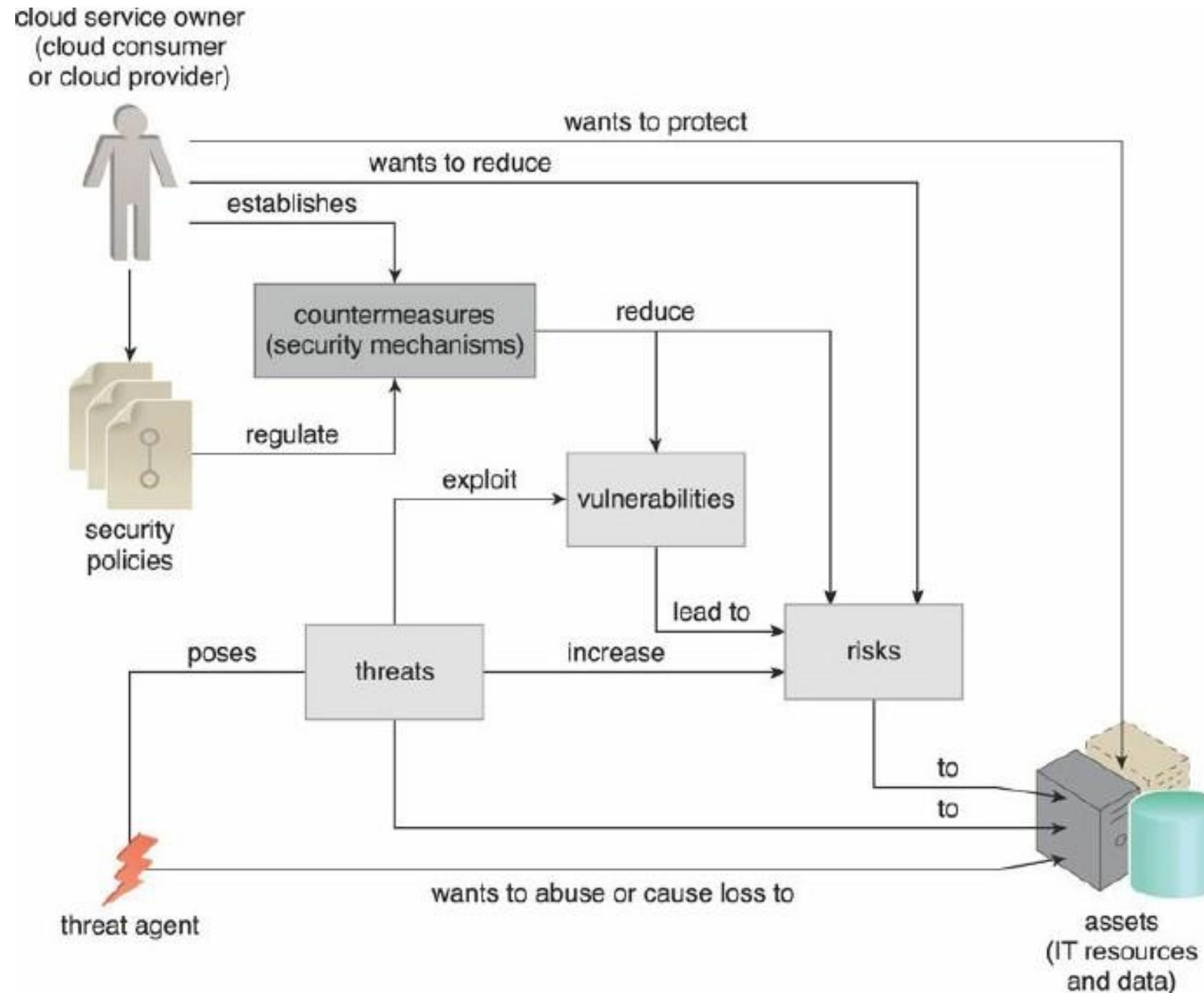
- Being accessible, available and usable within defined time period.
- In cloud the availability of cloud services can be a responsibility that is shared by the cloud provider and the cloud carrier. The availability of a cloud-based solution that extends to cloud service consumers is further shared by the cloud consumer.

# Threat Agents

An entity that poses a threat because it is capable of carrying out an attack.

- Can originate either internally or externally.
- Human or Software.

# Threat Agents



# Threat Agents

## Anonymous Attacker

- Non-trusted cloud service consumer without permissions in the cloud. Attempts attacks from outside cloud permission boundary, mostly using public networks.

## Malicious Service Agent

- Able to intercept and forward the network traffic that flows within a cloud. Then to maliciously use and augment the data.

## Trusted Attacker

- Shares IT resources in the same cloud environment as the cloud consumer and attempts to exploit legitimate credentials to target cloud providers and the cloud tenants.

## Malicious Insider

- Human threat agents acting on behalf of or in relation to the cloud provider. Typically current or former employees or third parties with access to the cloud provider's premises.

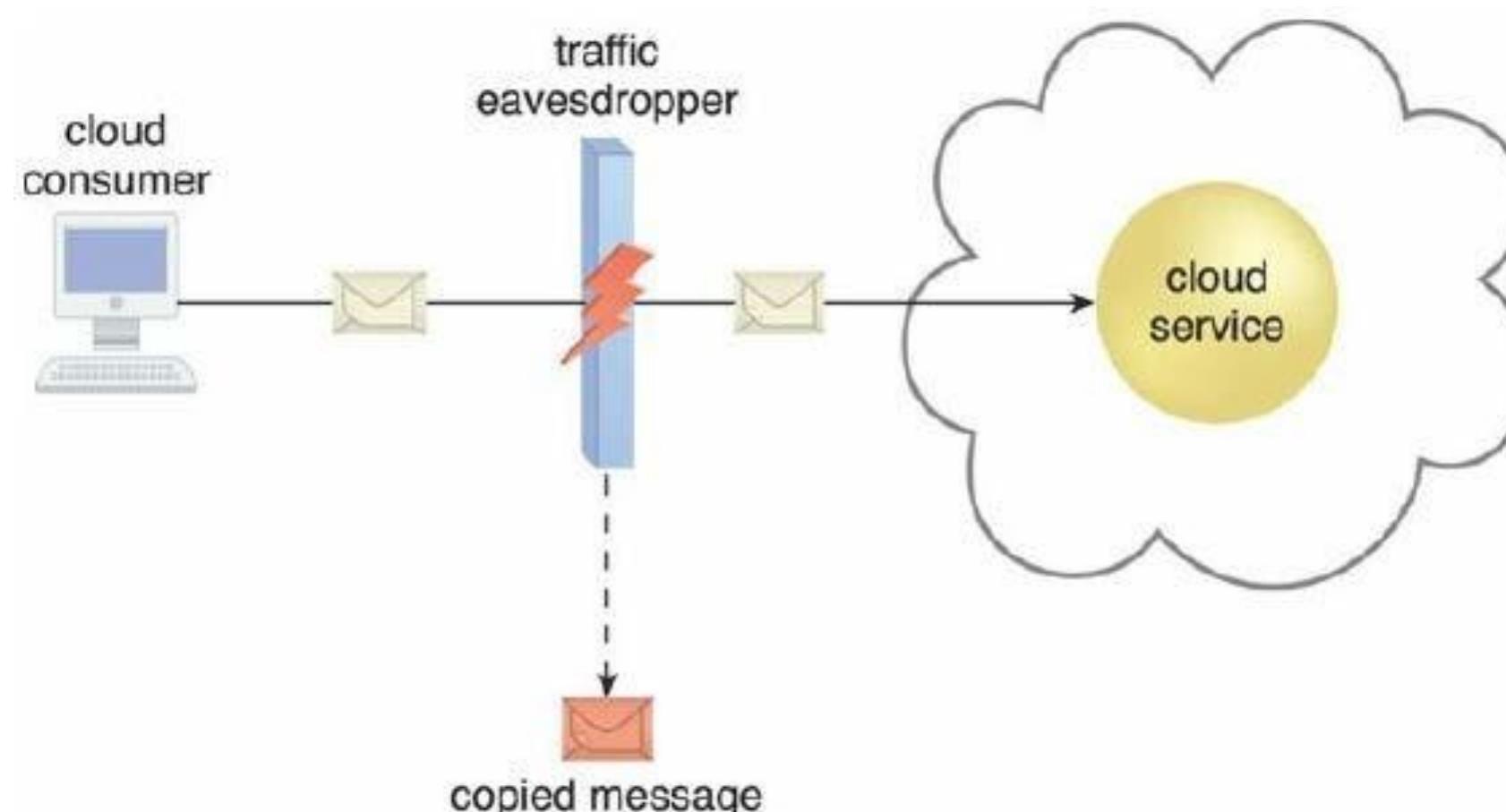
# Cloud Security Threats

- Traffic Eavesdropping
- Malicious Intermediary
- Denial of Service
- Insufficient Authorization
- Virtualization Attack
- Overlapping Trust Boundaries

# Traffic Eavesdropping

Data transferred to or within a cloud is passively intercepted by a malicious service agent for information gathering purposes.

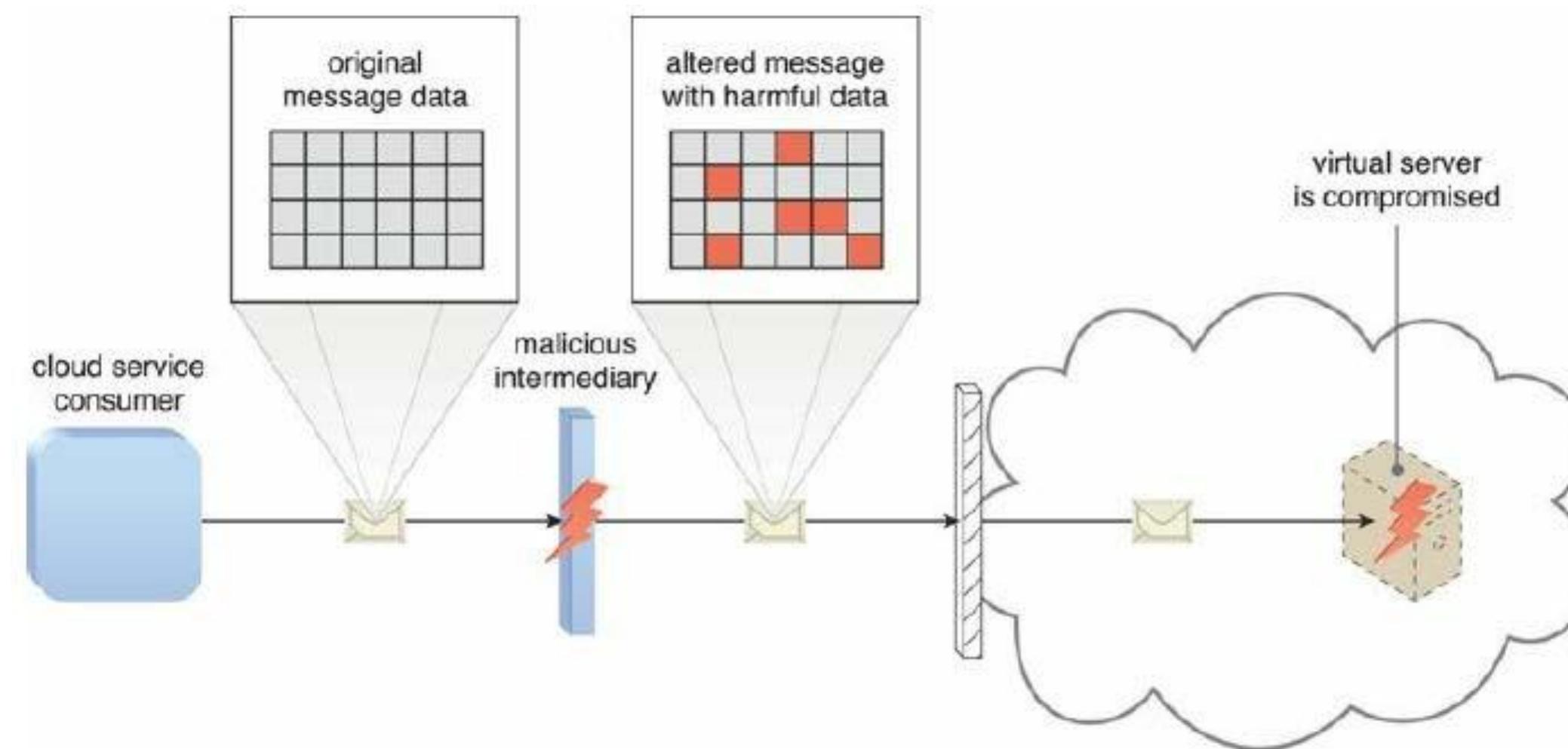
- Aim to compromise the confidentiality of the data.
- Due to passive nature of the attack, it can take place undetected for extended periods of time.



# Malicious Intermediary

Messages are intercepted and altered by a malicious service agent.

- Potentially compromising the message's confidentiality and/or integrity.
- May insert harmful data into the message.

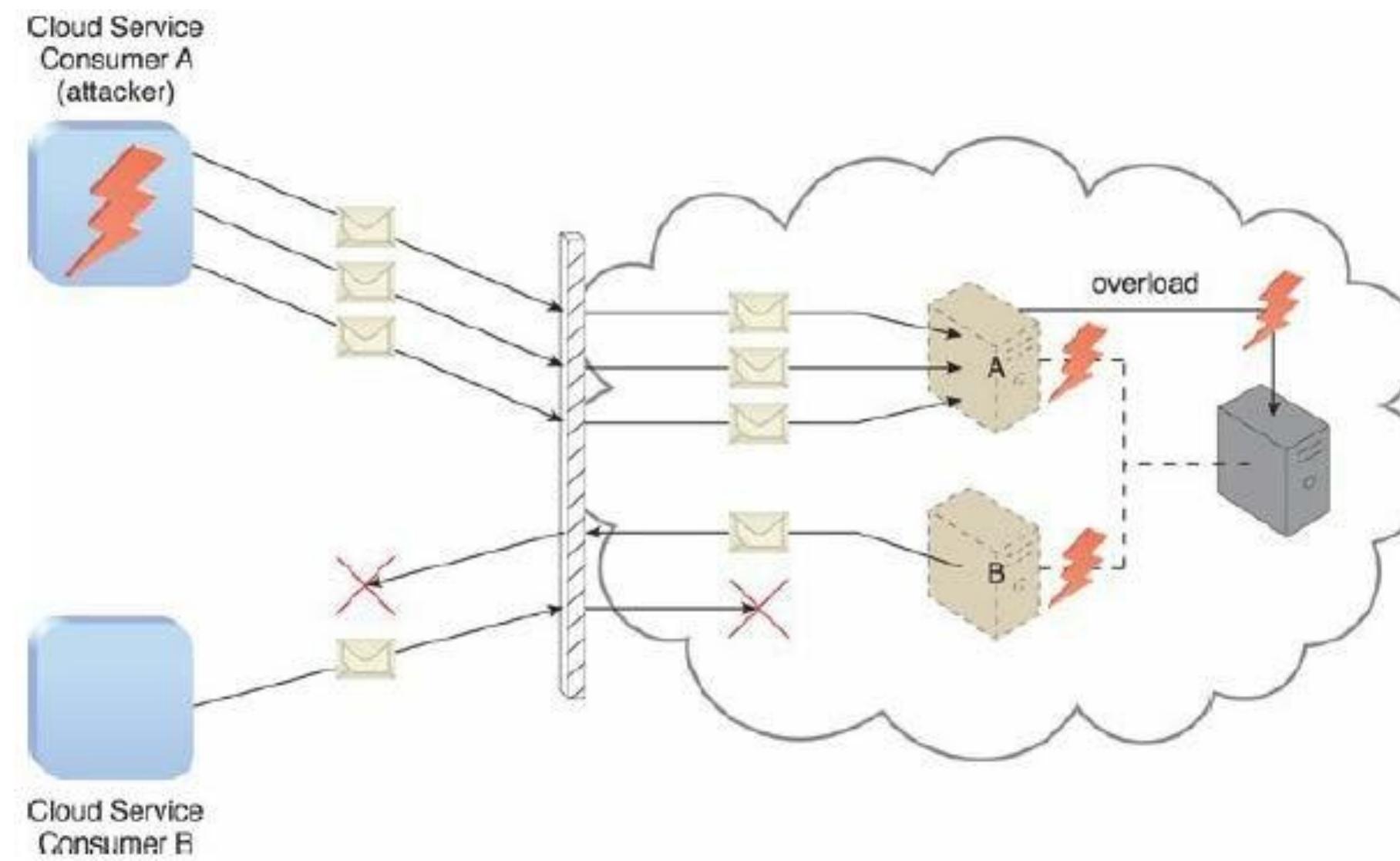


## Denial of Service

Overload IT resources to the point where they cannot function properly.

- The workload on cloud services is artificially increased with imitation messages or repeated communication requests.
- The network is overloaded with traffic to reduce its responsiveness and cripple its performance.
- Multiple cloud service requests are sent, each of which is designed to consume excessive memory and processing resources.

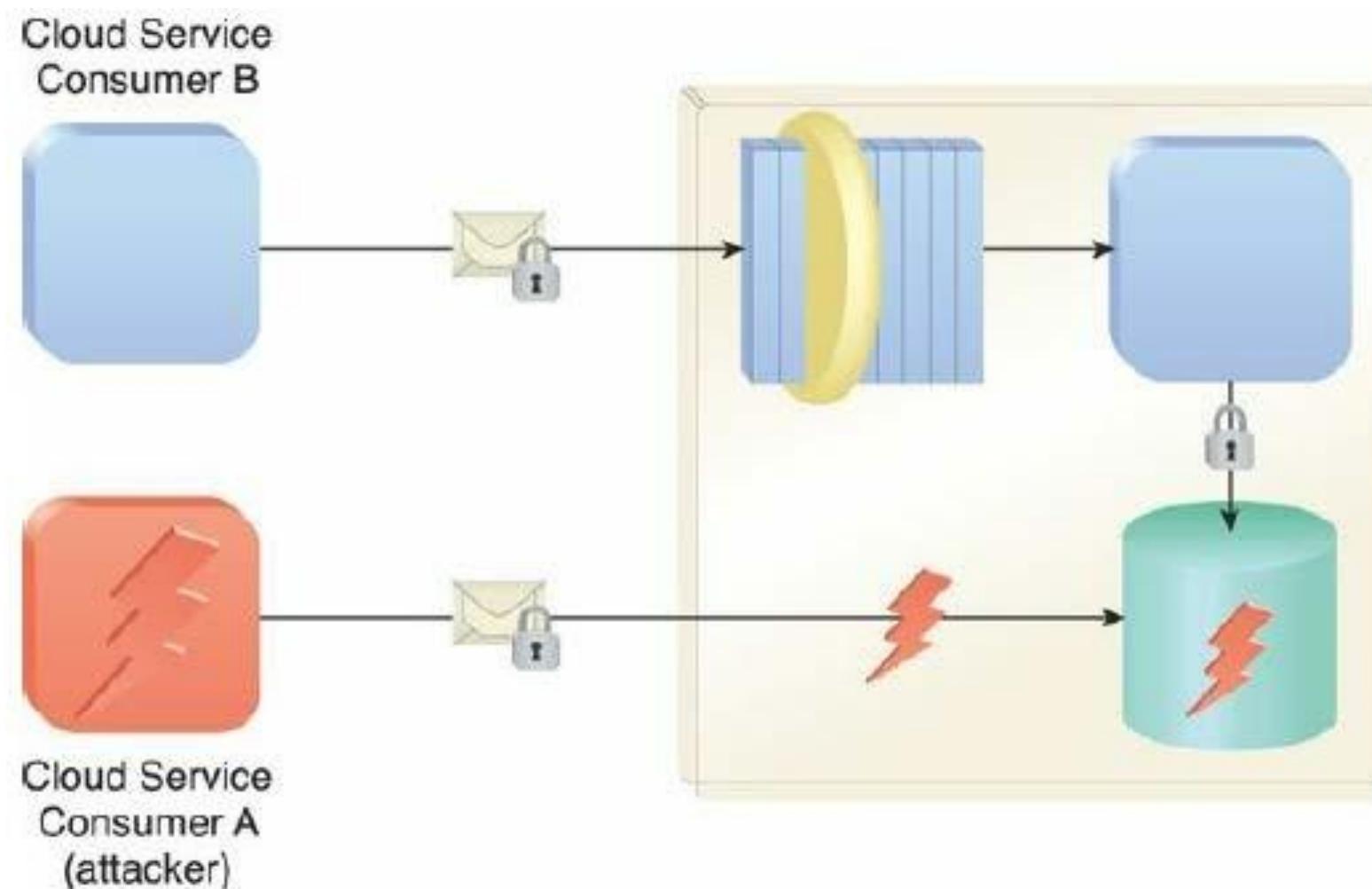
# Denial of Service



# Insufficient Authorization

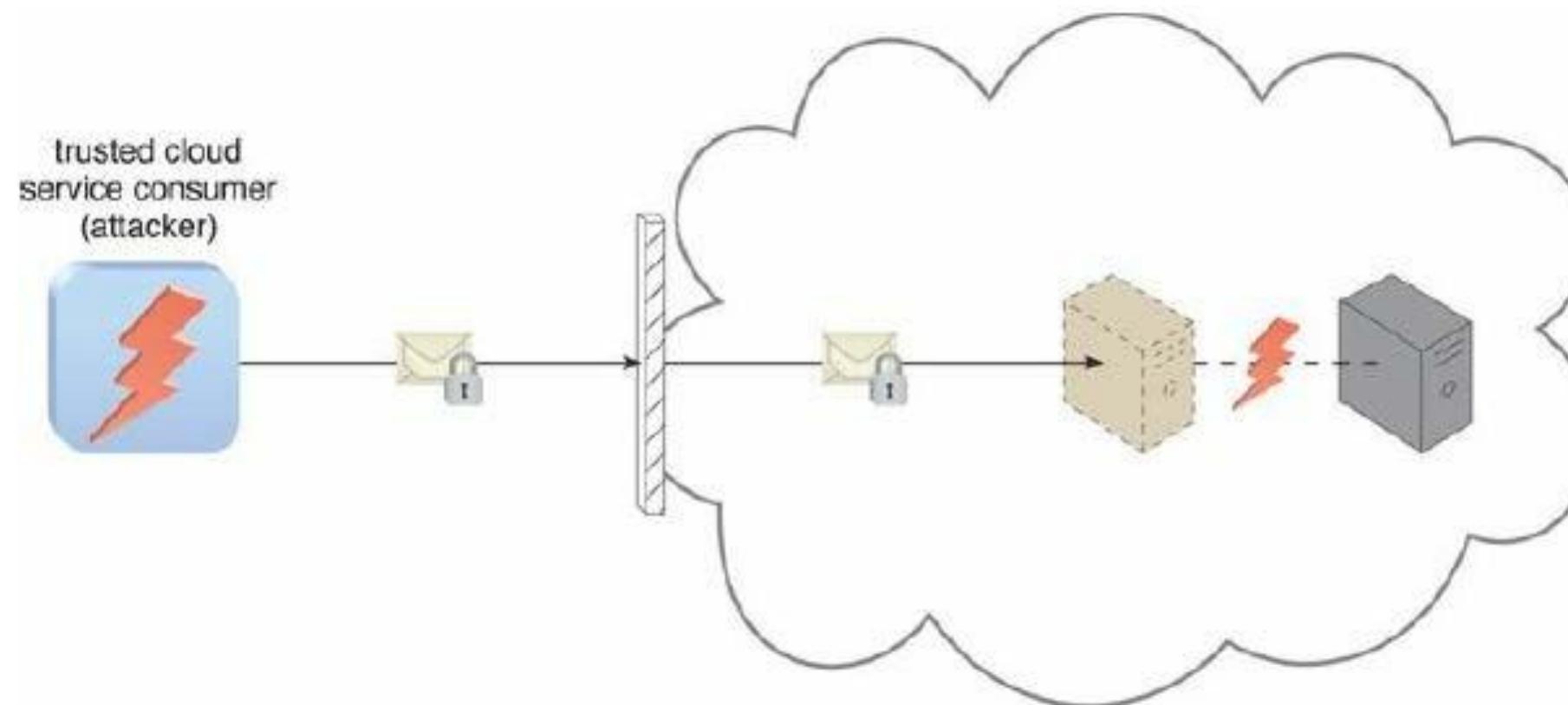
Occurs when access is granted to an attacker erroneously or too broadly to IT resources that are normally protected.

- Result of the attacker gaining direct access to IT resources that were implemented to be accessed by trusted consumer programs.



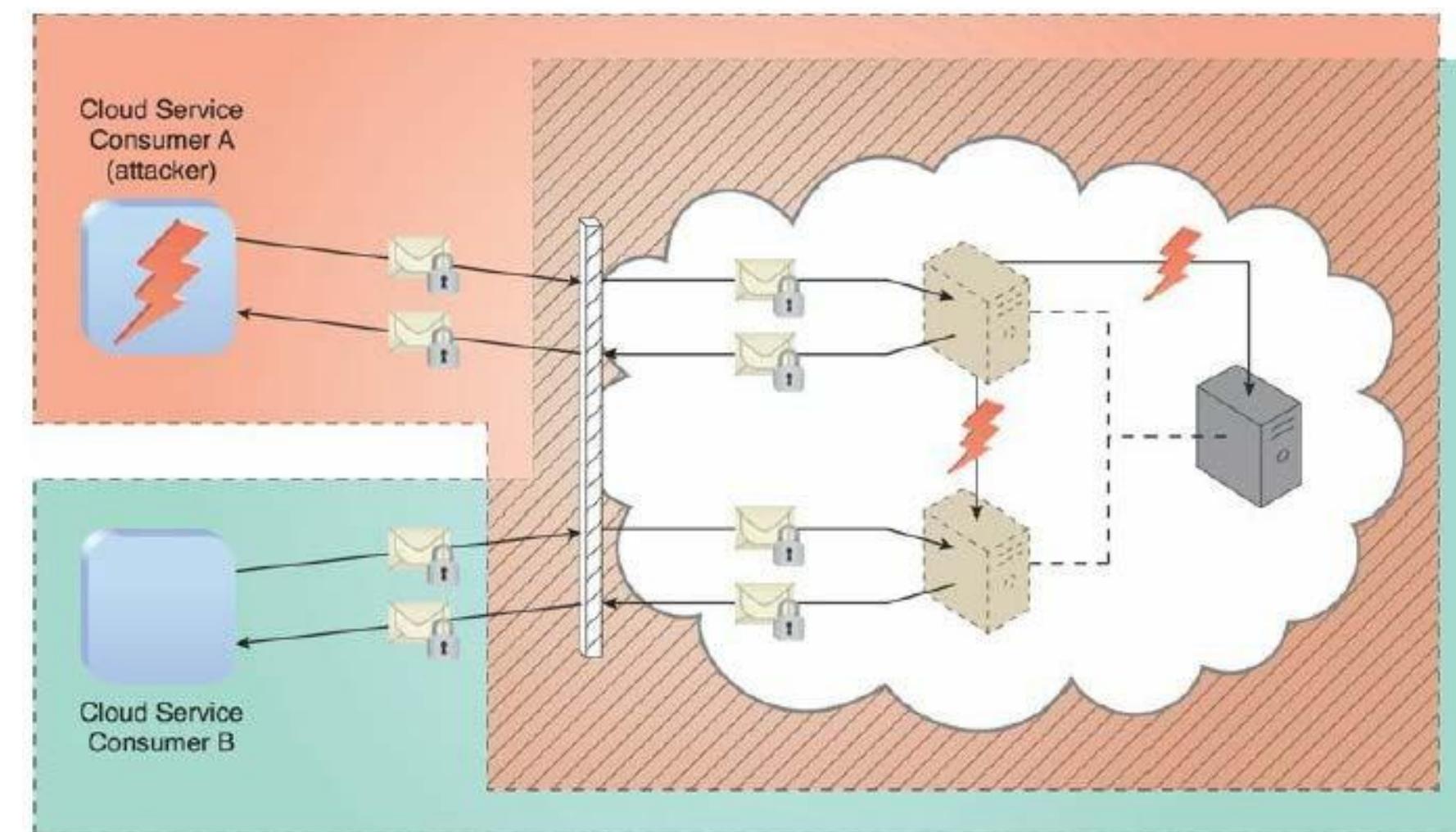
# Virtualization Attack

Exploits vulnerabilities in the virtualization platform to jeopardize its confidentiality, integrity, and/or availability.



# Overlapping Trust Boundaries

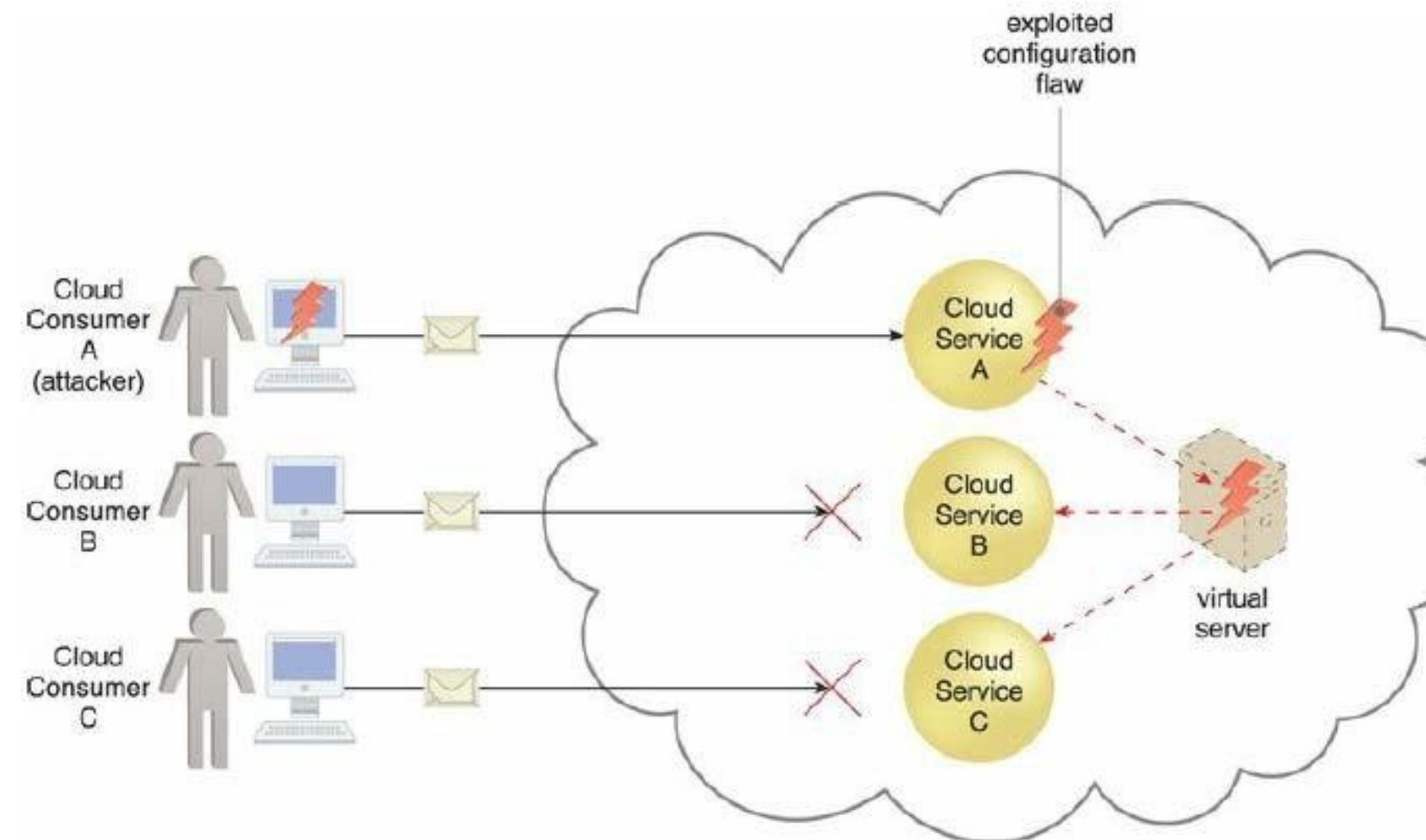
Target shared IT resources with the intention of compromising cloud consumers or other IT resources that share the same trust boundary



# Checklist on Cloud Security

## Substandard design, implementation or configuration

- Security issues may arise beyond runtime exception or system failures
- Attackers may exploit these vulnerabilities



# Checklist on Cloud Security (Cont)

## Security policy checks

- Check for any disparity as majority of the IT resources are now managed by cloud service providers

## Contracts

- Examine contracts and SLAs put forth by cloud providers to ensure that security policies, and other relevant guarantees, are satisfactory when it comes to security

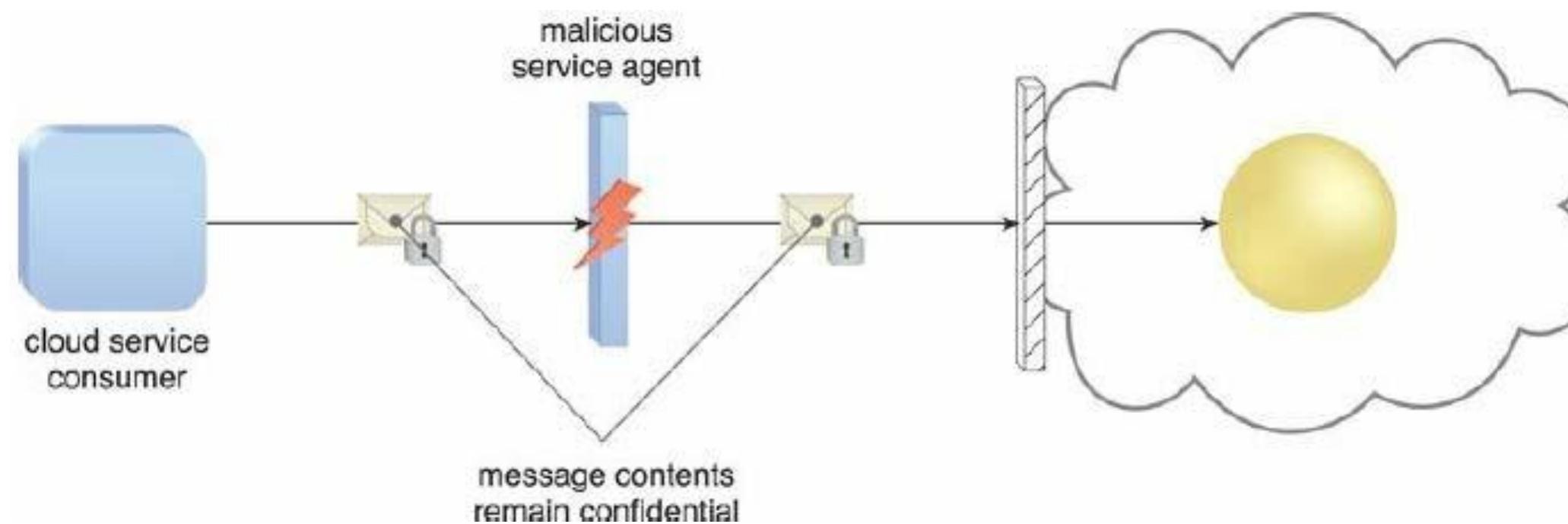
## Risk Management

- Continuous risk assessment as part of risk management strategy

# Cloud Security Countermeasures

## Encryption

Secret key based encryption mechanisms to counter traffic eavesdropping, malicious intermediary, insufficient authorization, and overlapping trust boundaries security threats.



# Encryption

## Symmetric Encryption

- Secret key cryptography, uses the same key for both encryption and decryption
- Does not have the characteristic of non-repudiation, cannot determine which party performed the message encryption or decryption

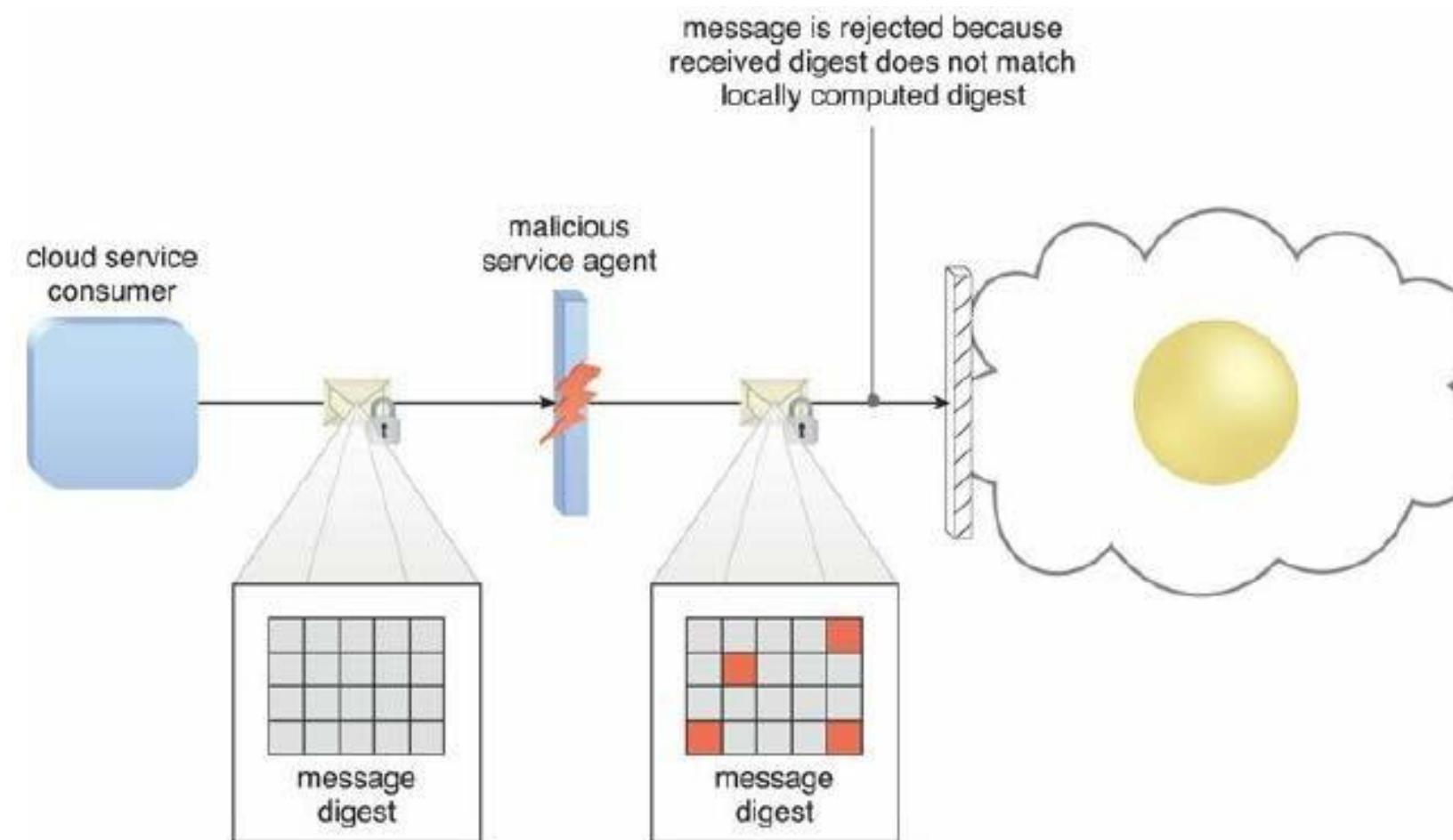
## Asymmetric Encryption

- Relies on the use of two different keys, private key and a public key
- private key is known only to its owner while the public key is commonly available
- Doesn't provide message integrity or authenticity protection due to the communal nature of the public key

# Hashing

When non-reversible form of data protection is required

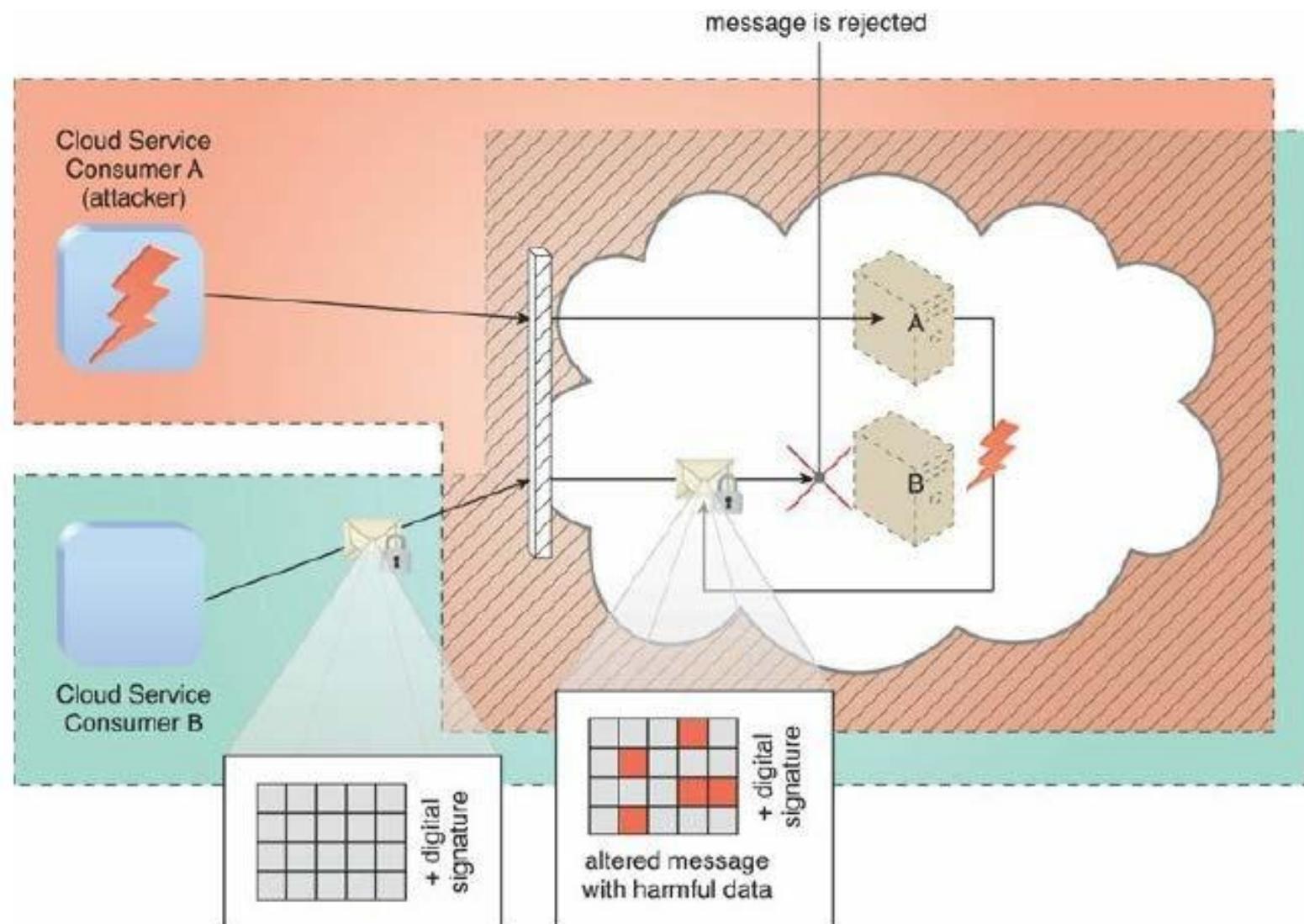
- Derive a hashing code or *message digest* from a message
- Sender attach message digest to the message
- Recipient applies the same hash function to the message to verify that the produced message digest is identical to the one that accompanied the message



# Digital Signature

Messages are assigned a digital signature prior to transmission, which is then rendered invalid if the message experiences any unauthorized modifications

- Both hashing and asymmetrical encryption are involved in the creation of a digital signature



# Public Key Infrastructure (PKI)

- Used to associate public keys with their corresponding key owners
- Rely on the use of digital certificates, which are digitally signed data structures that bind public keys to certificate owner identities with a validity time period
- Digital certificates are usually digitally signed by a third- party certificate authority

## DECRYPTION

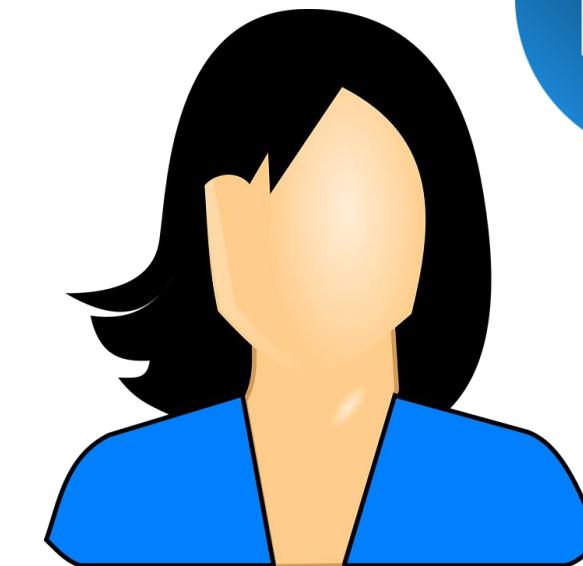


Private  
Key

Request



Sends her public key to encrypt the message



Encrypts and sends the Secure message



Public  
Key



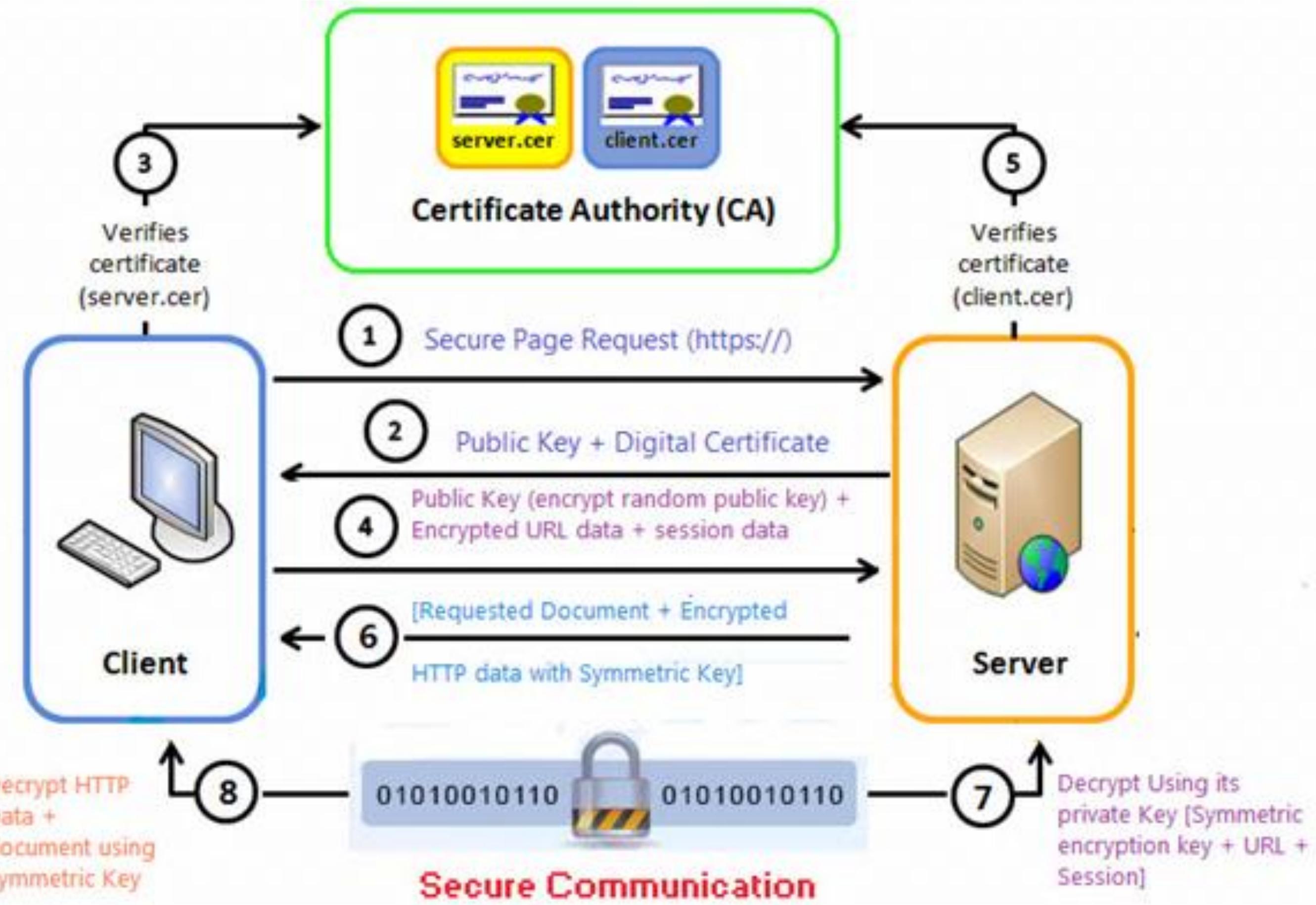
Public  
Key



Private  
Key

Can be  
distributed to  
anyone

Will be always  
private



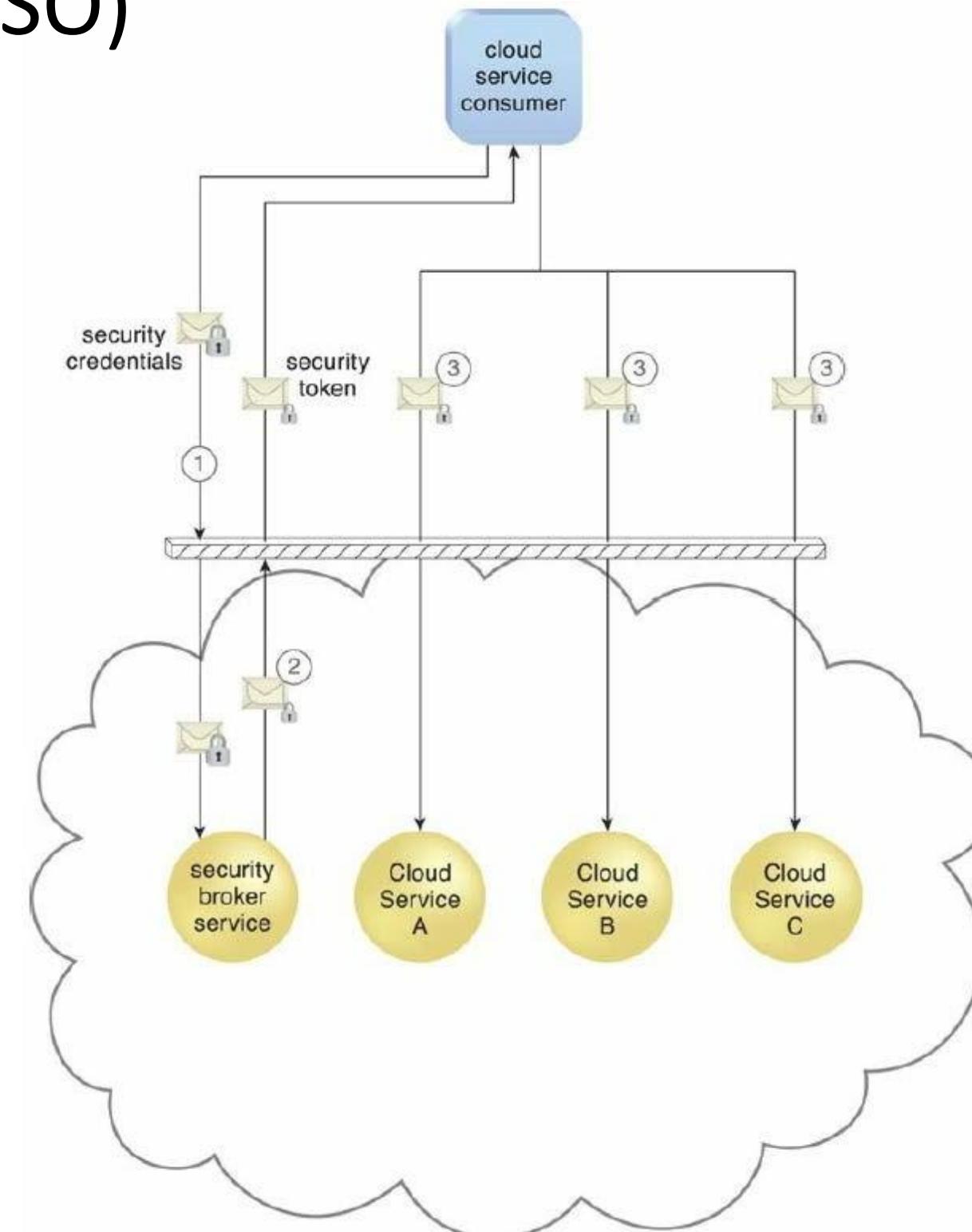
# Identity Access Management (IAM)

- IAM encompasses controlling and tracking user identities and access in IT environments.
  - **Authentication:** Manages credentials like usernames, passwords, digital signatures, biometric data, and binds accounts to specific hardware or software identifiers.
  - **Authorization:** Defines access control levels and manages relationships between user identities, access rights, and resource availability.
  - **User Management:** Involves administrative tasks like creating user accounts, resetting passwords, setting password policies, and managing privileges.
  - **Credential Management:** Establishes and manages access rules for user accounts to prevent unauthorized access.

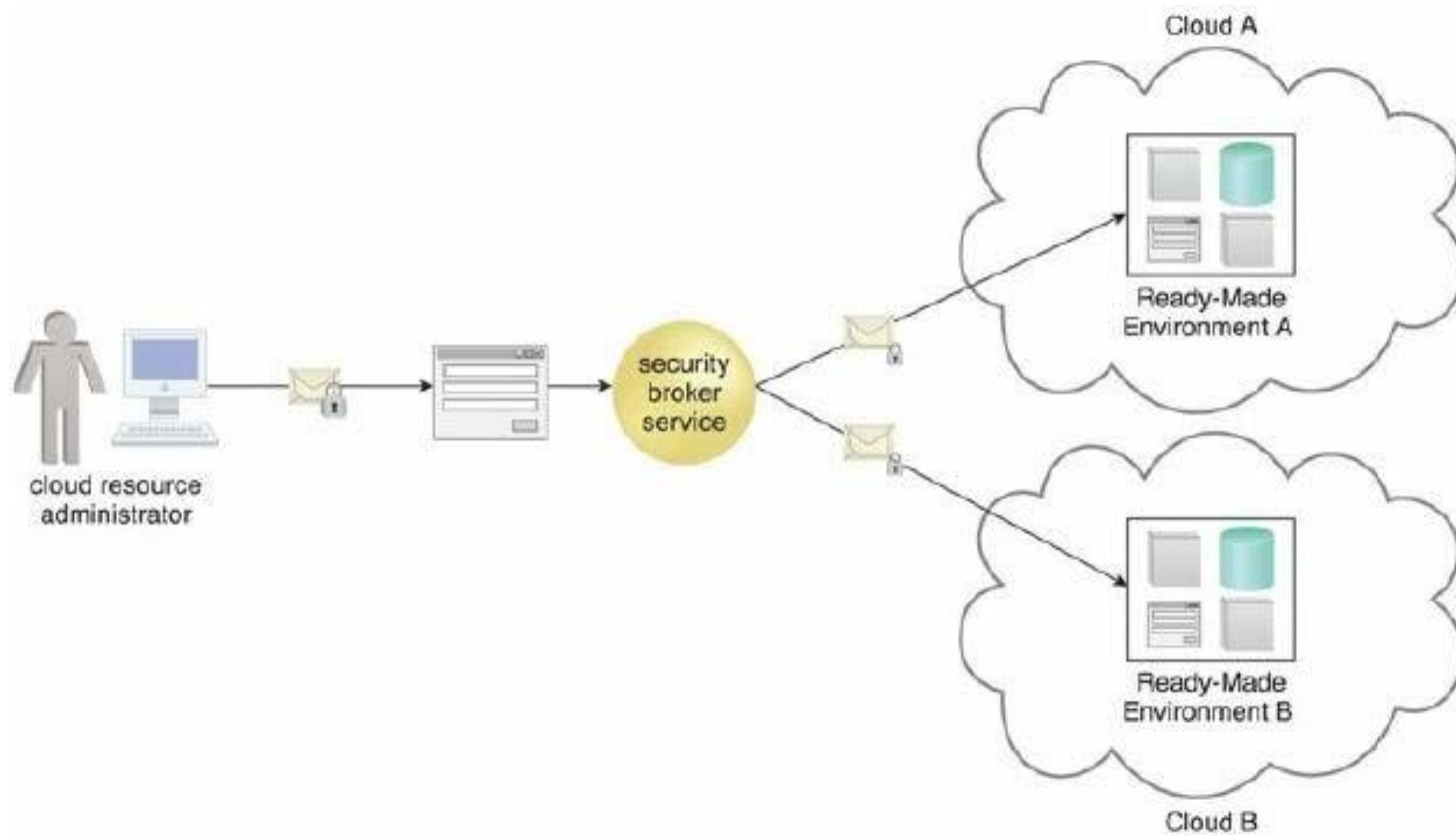
# Single Sign-On (SSO)

Propagating the authentication information across multiple cloud services can be a challenging

- Enables authentication by a security broker that establish a persisted security context during consumer accesses to cloud services

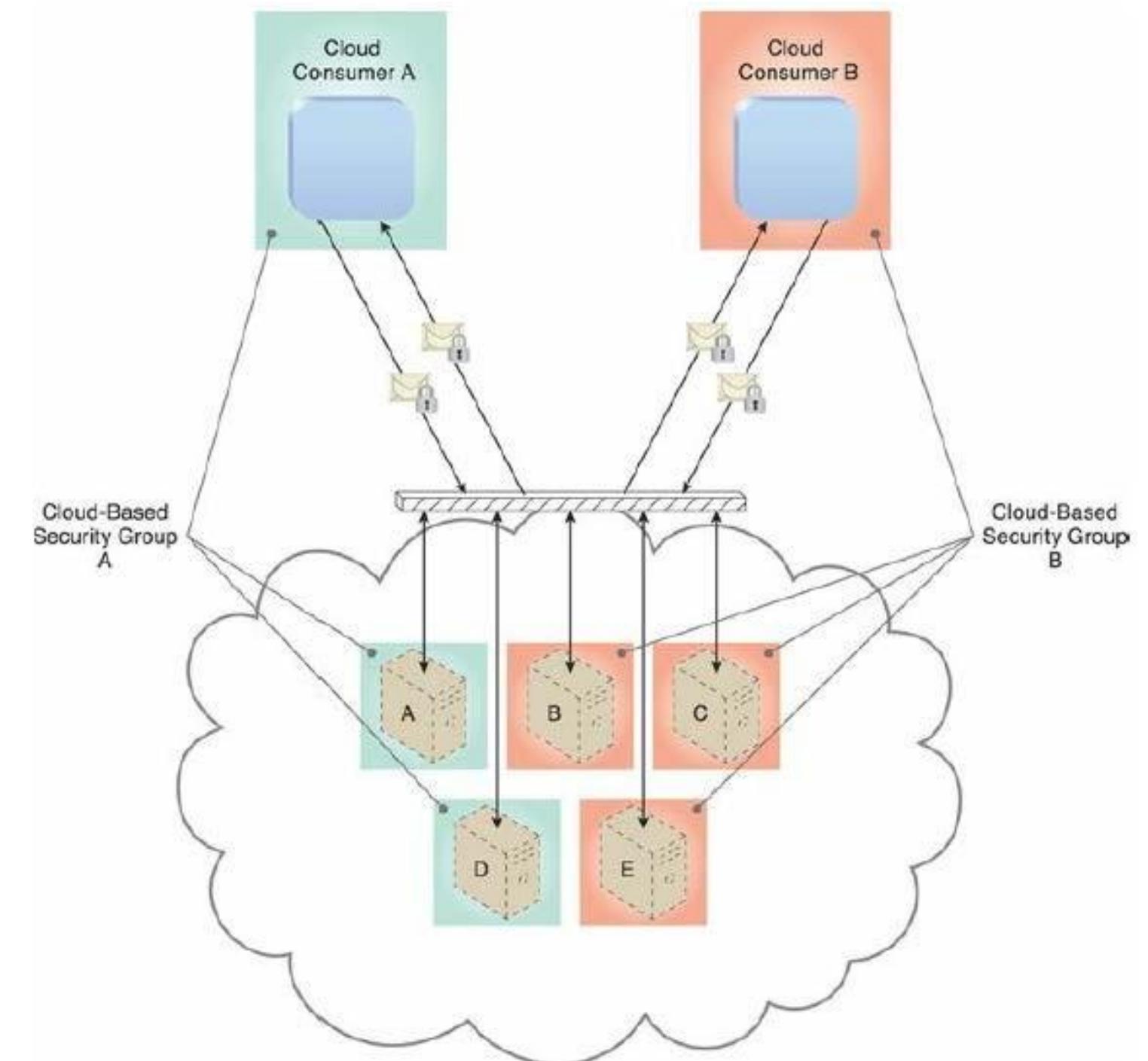


# Single Sign-On (SSO)



# Cloud-Based Security Groups

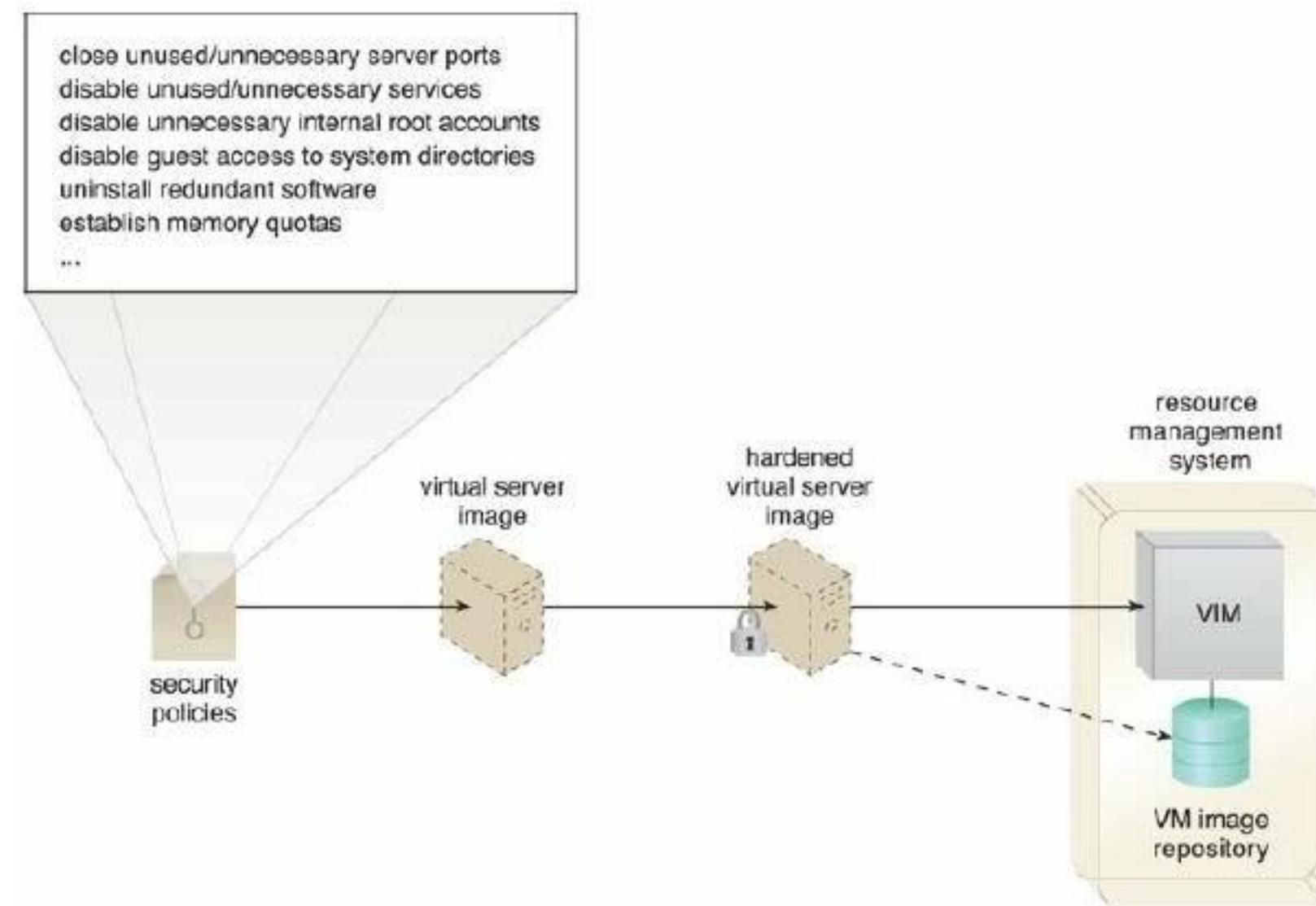
- Cloud resource segmentation is a process by which separate physical and virtual IT environments are created for different users and groups



# Hardened Virtual Server Images

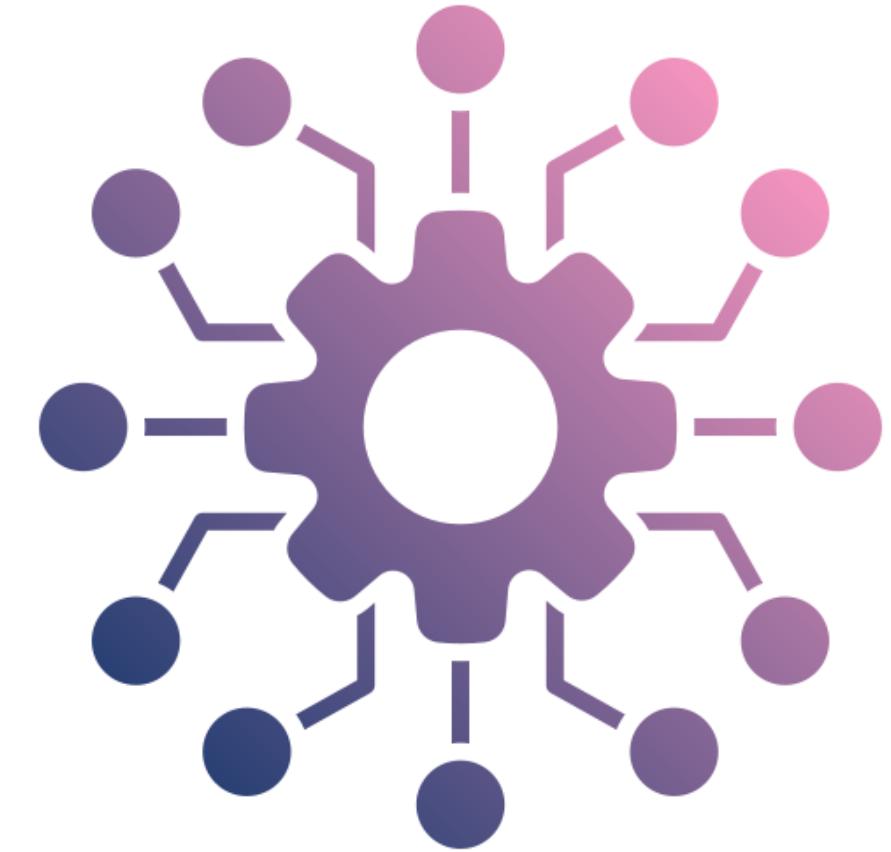
Process of stripping unnecessary software to limit potential vulnerabilities that can be exploited by attackers

- Removing redundant programs, closing unnecessary server ports, and disabling unused services, internal root accounts, and guest access



# References

- <https://aws.amazon.com/compliance/shared-responsibility-model/>
- Chapter 6 and 10, Cloud Computing: Concepts, Technology & Architecture, Thomas Erl, et al., Prentice- Hall, 2013



# Introduction to Microservices

Ravindu Nirmal Fernando

SLIIT | March 2025

# Foundations of Modern Software Architecture: Paving the Way for Microservices

- Influential Concepts and Technologies
  - **Domain-Driven Design:** Emphasizing the importance of reflecting real-world complexities in our code for better system modeling.
  - **Continuous Delivery:** Revolutionizing software deployment, making every code check-in a potential release candidate.
  - **Web Communication Advancements:** Enhancing how machines interact, leading to more efficient and robust systems.
- Architectural Shifts
  - **From Layered to Hexagonal:** Moving away from traditional layered architectures to avoid hidden complexities in business logic.
  - **Embracing Virtualization:** Utilizing on-demand provisioning and resizing of resources for greater flexibility with cloud computing.
- Organizational Practices
  - **Small Autonomous Teams:** Inspired by tech giants like Amazon and Google, promoting ownership and lifecycle management of services.
  - **Learning from Netflix:** Building resilient, scalable systems that can withstand and adapt to change.

# Microservices: A Natural Progression

- Emergence from Real-World Use:  
Microservices weren't pre-planned but evolved as a response to practical needs in software development.
- Responding to Change: Offering the agility and flexibility to adapt to new technologies and market demands.

# Monolithic Applications

- **Basic Structure**

- Single-Tiered Structure: Built as a single, unified unit.
- Combined Modules: Functional modules like UI, server logic, and database interactions are combined.

- **Design and Construction**

- Modular Architecture: Follows a modular structure within a single unit, aligning with object-oriented principles.
- Programming Constructs: Defined using language-specific constructs (e.g., Java packages).
- Build Artifacts: Built as a single artifact, such as a Java JAR file.

- **Characteristics**

- Inter-module Dependencies: Modules are tightly coupled and interdependent.
- Unified Deployment: Deployed as a single entity.

- **Scalability**

- Scalability Approach: Scaling involves replicating the entire application, not individual components.

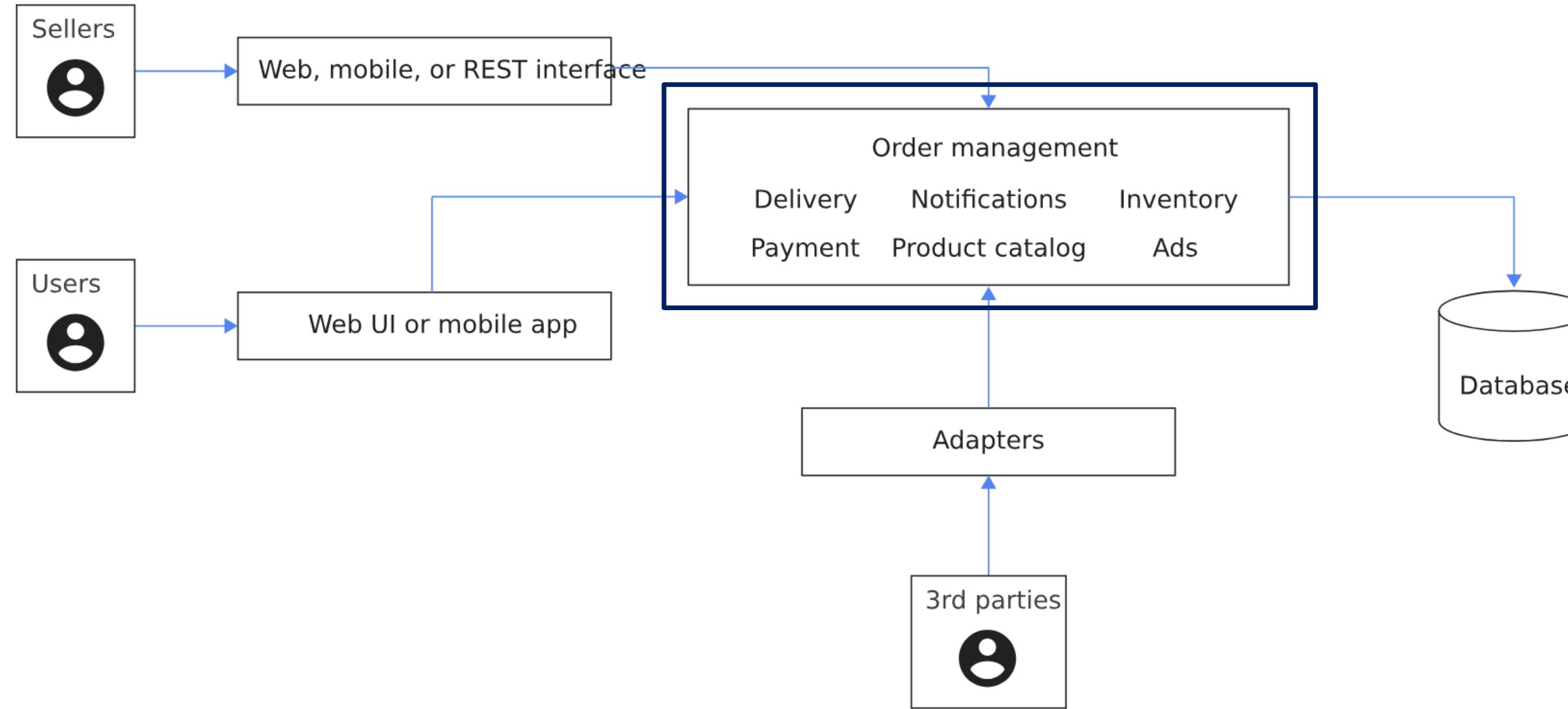


Diagram of a monolithic ecommerce application with several modules using a combination of programming language constructs. (<https://cloud.google.com/architecture/microservices-architecture-introduction>)

- **Benefits of Monolithic Architecture**

- **Simplified Testing:** Tools like Selenium enable end-to-end testing of the entire application.
- **Ease of Deployment:** Deployment involves simply copying the packaged application to a server.
- **Resource Sharing:** All modules share memory, space, and resources, streamlining cross-cutting concerns like logging, caching, and security.
- **Intra-Process Communication:** Direct module-to-module calls can offer performance advantages over network-dependent microservices.

- **Challenges of Monolithic Architecture**

- **Scalability Issues:** Difficulty in scaling when different modules have conflicting resource requirements.
- **Complexity in Maintenance and Updates:** As the application grows, implementing changes becomes more complicated due to tightly coupled modules.
- **CI/CD Complications:** Continuous integration and deployment become challenging as any update requires redeploying the entire application.
- **Vulnerability to System Failures:** A bug in any module, like a memory leak, can crash the entire system.
- **Technological Rigidity:** Adopting new frameworks or languages is costly and time-consuming, as it often requires rewriting the entire application.

# Understanding Microservices

- **Core Characteristics**

- **Small and Focused:** Aimed at doing one thing well, avoiding sprawling codebases.
- **Cohesion and Single Responsibility:** Adhering to the principle of grouping related code and separating unrelated functionalities.

- **Size and Scope**

- **No Fixed Size:** Size varies based on language expressiveness and domain complexity.
- **Team Alignment:** Ideally sized to be managed by a small team.
- **Balance in Size:** Smaller services maximize benefits but increase complexity.

- **Autonomy**
  - **Independent Entities:** Deployed separately, can be different technologies, possibly as isolated services on a PAAS or as individual operating system processes.
  - **Network Communication:** Services communicate via network calls, ensuring separation and reducing tight coupling.
- **Deployment and Change Management**
  - **Independent Deployment:** Services can be deployed independently without impacting others.
  - **API-Centric Interaction:** Services expose APIs for interaction, emphasizing decoupled, technology-agnostic interfaces.
- **Decoupling**
  - **Key to Microservices:** Essential for maintaining independence and achieving the benefits of microservices architecture.
  - **Change and Deployment:** Ability to change and deploy a service independently is crucial.

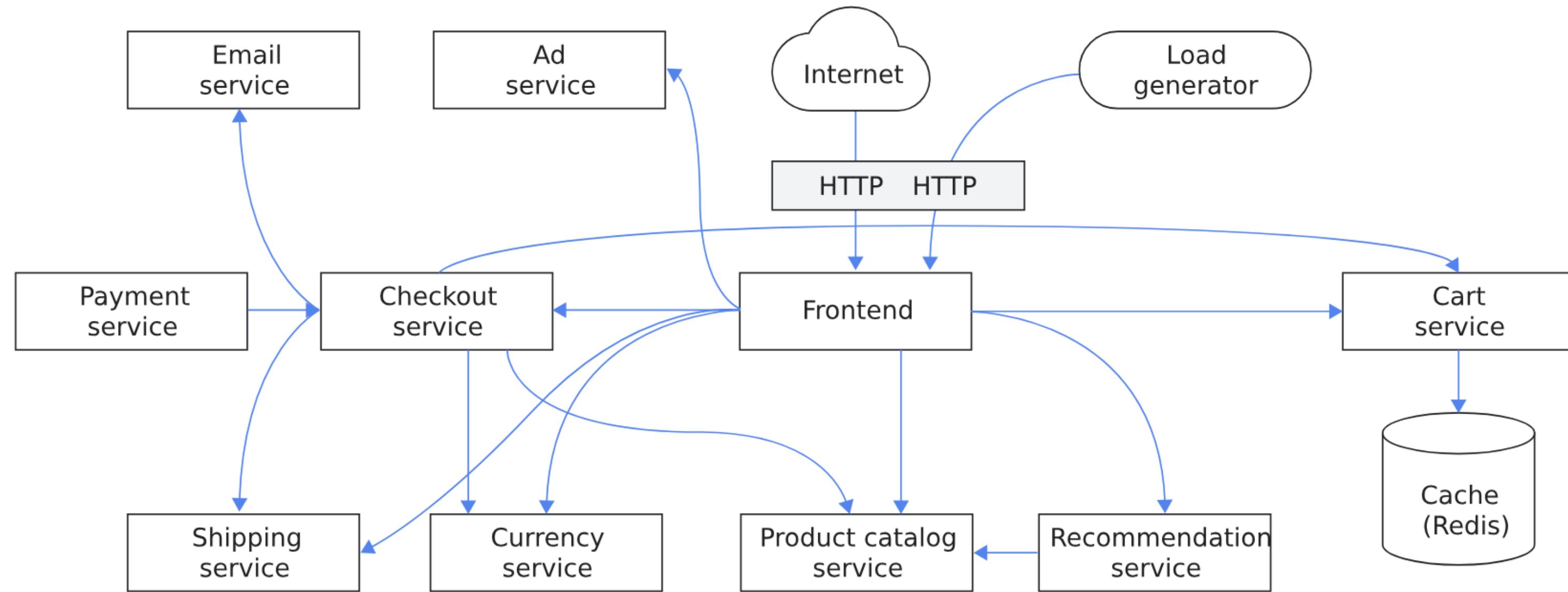
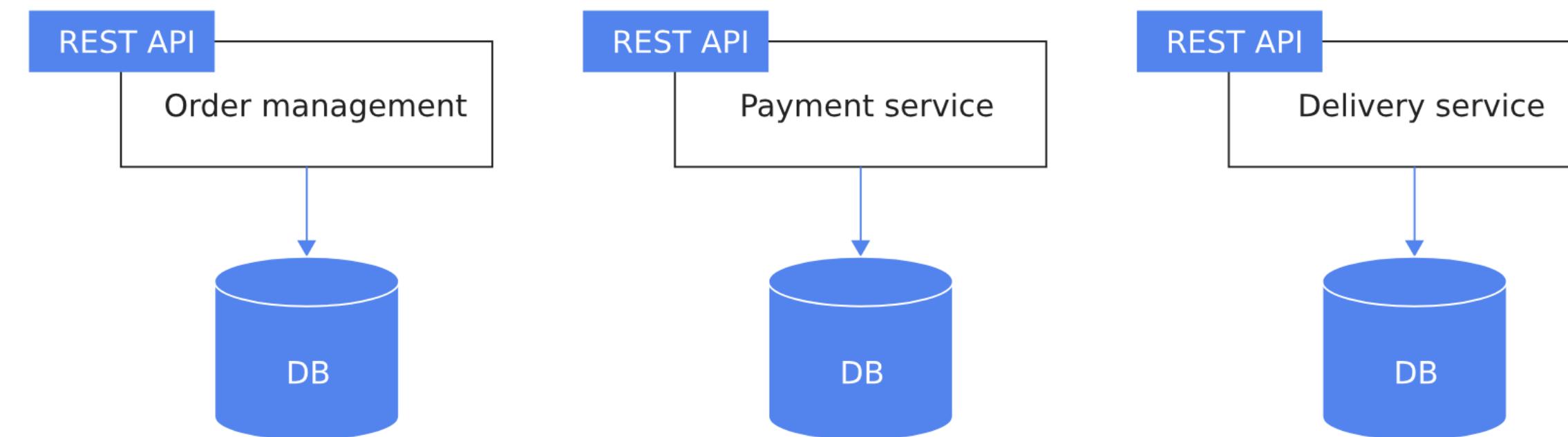


Diagram of an ecommerce application with functional areas implemented by microservices.

(<https://cloud.google.com/architecture/microservices-architecture-introduction>)

- **Database Relationship**
  - **Service-Specific Databases:** Each microservice has its own database tailored to its requirements.
  - **Loose Coupling:** This approach ensures loose coupling by routing data requests through service APIs instead of a shared database.
  - **Independent Data Management:** Each service manages its data independently, enhancing autonomy and reducing interdependencies.



# Benefits of Microservices Architecture

Aspect	Benefit Details
<b>Enhanced Development and Maintenance</b>	<ul style="list-style-type: none"><li>- Breaks application into smaller, manageable chunks.</li><li>- Clear boundaries with defined APIs.</li><li>- Quicker development, easier understanding and maintenance.</li></ul>
<b>Team Autonomy and Efficiency</b>	<ul style="list-style-type: none"><li>- Independent development of services by teams.</li><li>- Full lifecycle ownership of services.</li><li>- Flexibility to use different programming languages (Polyglot Development).</li></ul>
<b>Improved Scalability and Market Responsiveness</b>	<ul style="list-style-type: none"><li>- Independent scaling based on service needs.</li><li>- Hardware optimization for resource requirements.</li><li>- Faster product delivery and improved time to market.</li></ul>

# Challenges of Microservices Architecture

Challenge Category	Challenge Details
<b>Complexity in Distributed Systems</b>	<ul style="list-style-type: none"><li>- Necessity of choosing and implementing inter-service communication mechanisms.</li><li>- Managing partial failures and service unavailability.</li></ul>
<b>Transaction Management Across Services</b>	<ul style="list-style-type: none"><li>- Handling atomic operations across multiple microservices (Distributed Transactions).</li><li>- Maintaining data consistency during failures (Consistency Issues).</li></ul>
<b>Testing and Deployment Complexities</b>	<ul style="list-style-type: none"><li>- Requirement for comprehensive testing across multiple services.</li><li>- Complexities in managing multiple service deployments and service discovery.</li></ul>
<b>Operational Overhead</b>	<ul style="list-style-type: none"><li>- Increased need for monitoring and alerting across more services.</li><li>- Higher risk of failure due to more points of service-to-service communication.</li><li>- Challenges in productionizing and maintaining robust operations infrastructure.</li></ul>
<b>Performance and Suitability Considerations</b>	<ul style="list-style-type: none"><li>- Potential latency issues due to network calls between services.</li><li>- Not suitable for all types of applications, especially those requiring real-time data processing.</li><li>- Importance of clear communication and service boundary planning.</li></ul>

# Migrating from Monolithic to Microservices: Key Considerations

Consideration Category	Details
Assessing the Need for Migration	<ul style="list-style-type: none"><li>- Evaluate if microservices align with business goals and pain points.</li><li>- Consider simpler alternatives like autoscaling or enhanced testing.</li></ul>
Starting the Migration Process	<ul style="list-style-type: none"><li>- Begin with extracting and deploying one service independently.</li><li>- Adopt an iterative approach, learning and adapting with each service migration.</li></ul>
Strategic Implementation	<ul style="list-style-type: none"><li>- Recognize varying approaches to microservice size and quantity among teams.</li><li>- Emphasize continuous learning and strategy refinement.</li></ul>
Future Learning and Strategies	<ul style="list-style-type: none"><li>- Explore strategies for detailed refactoring from monolithic to microservices.</li><li>- Plan for ongoing education and adaptation of methods.</li></ul>

# References

- <https://cloud.google.com/architecture/microservices-architecture-introduction>
- Building Microservices, Sam Newman



# Microservice Design Patterns

Ravindu Nirmal Fernando

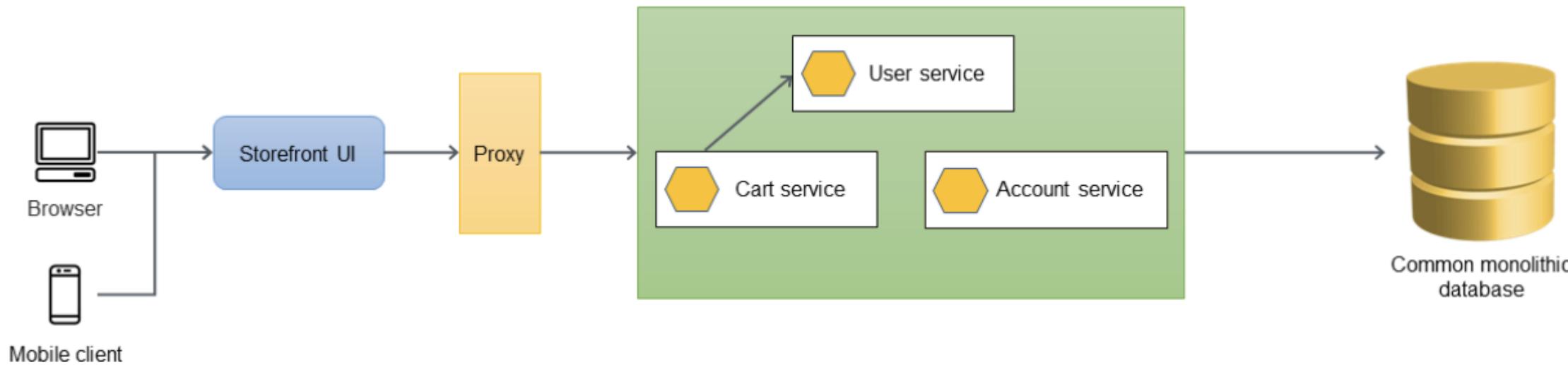
SLIIT | March 2025

# Design Patterns for migrating Monoliths to Microservices

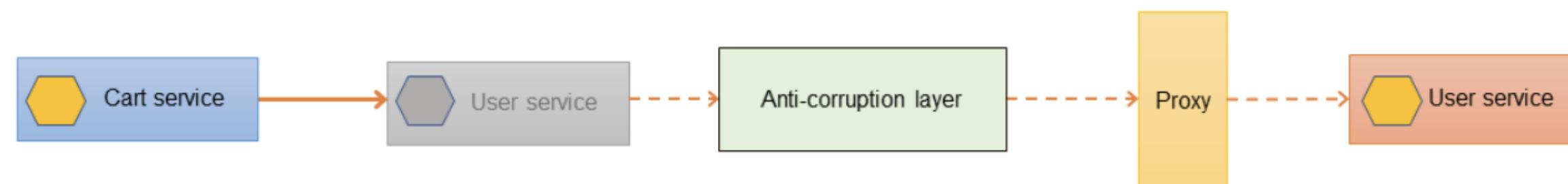
- Anti-Corruption Layer (ACL) pattern
- Strangler fig pattern

# Anti-Corruption Layer (ACL) pattern

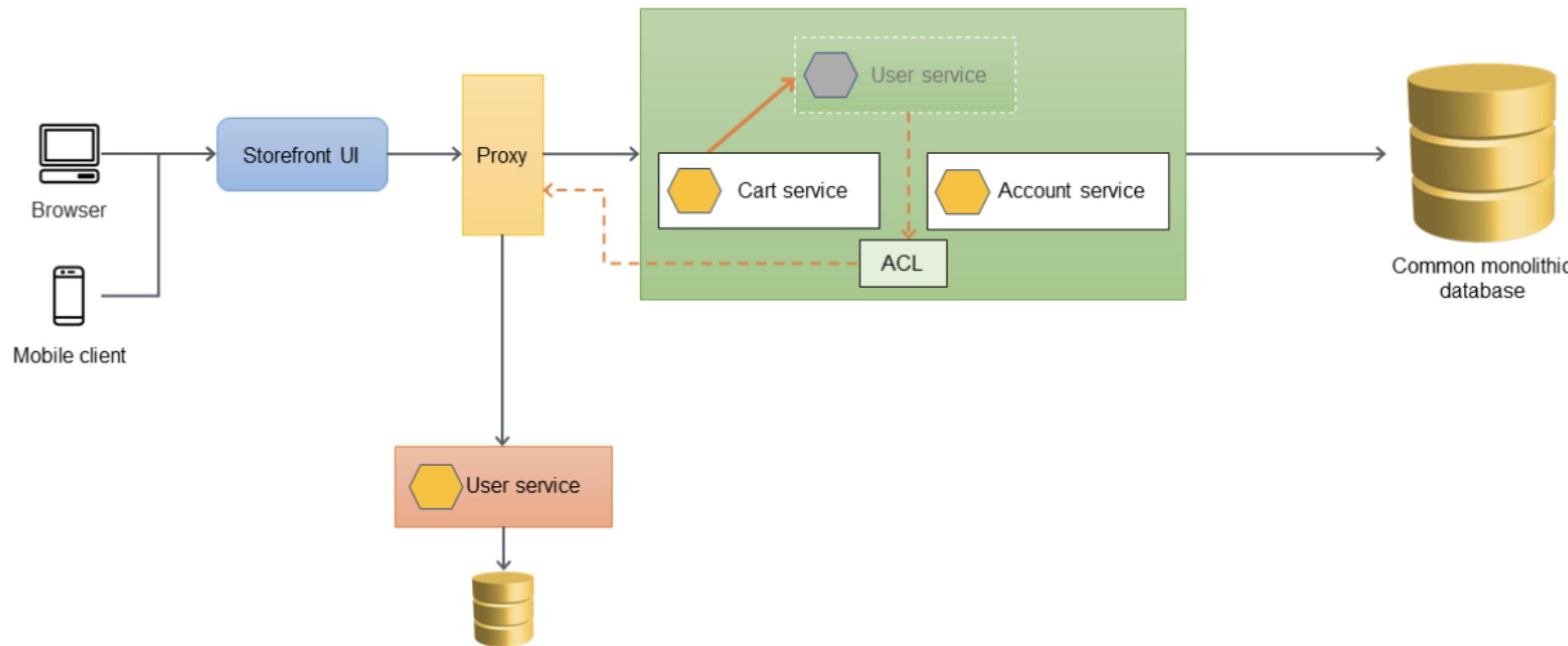
- Allows gradual translation from monoliths to microservices architecture.
- Allows legacy systems to communicate with modern services without internal changes and with minimal impact.
- Goal of ACLs is to minimize changes to existing functionality in monoliths and reduce business disruptions.
- Acts as an adapter or facade, converting calls to the new interface.
- ACL is also consumed as a part of Strangler-Fig pattern which we are going to talk next...



Existing Monolithic App



Introduction of ACL layer when User service is migrated

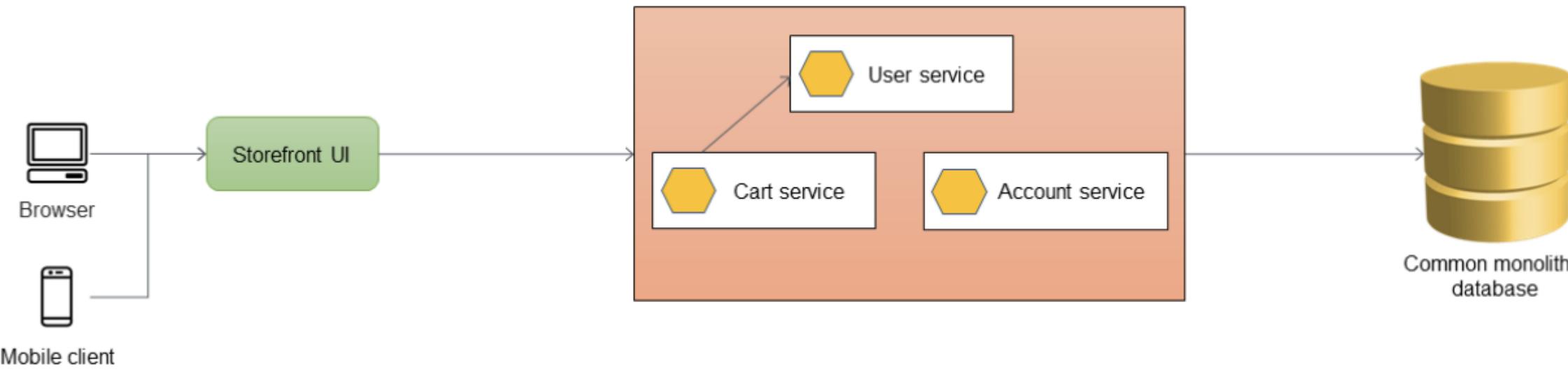


# Strangler fig Pattern

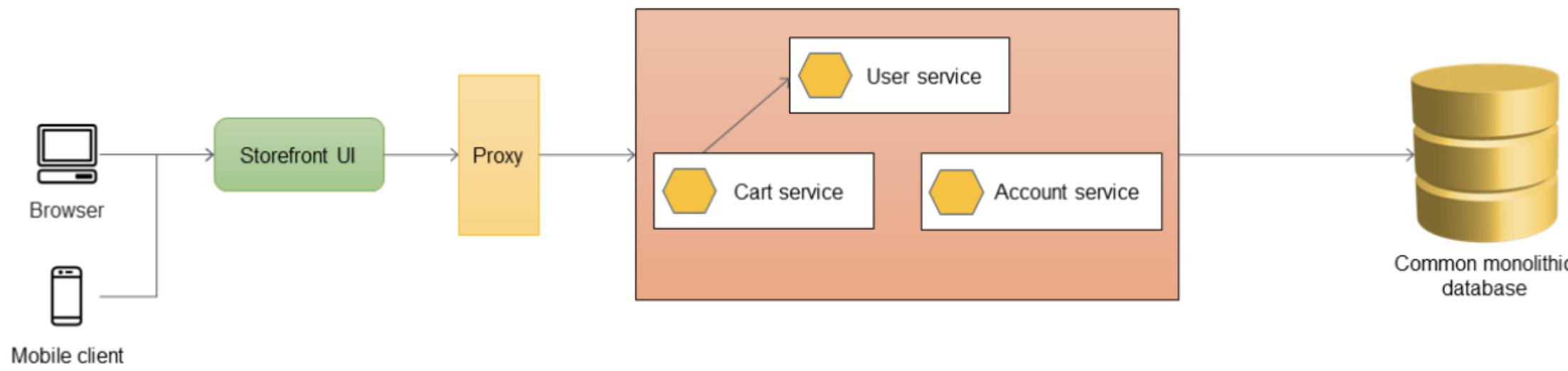
- Helps migrating a monolithic application to a microservices architecture incrementally, with reduced transformation risk and business disruption.
- Migrating a monolithic application to microservices based one requires rewriting and refactoring the code base and doing it once will be a huge risk.
- Hence Strangler fig patterns allows teams to focus on doing this migration incrementally and gradually while allowing app users to use the newly migrated features progressively.

# Process Involved in Strangler fig

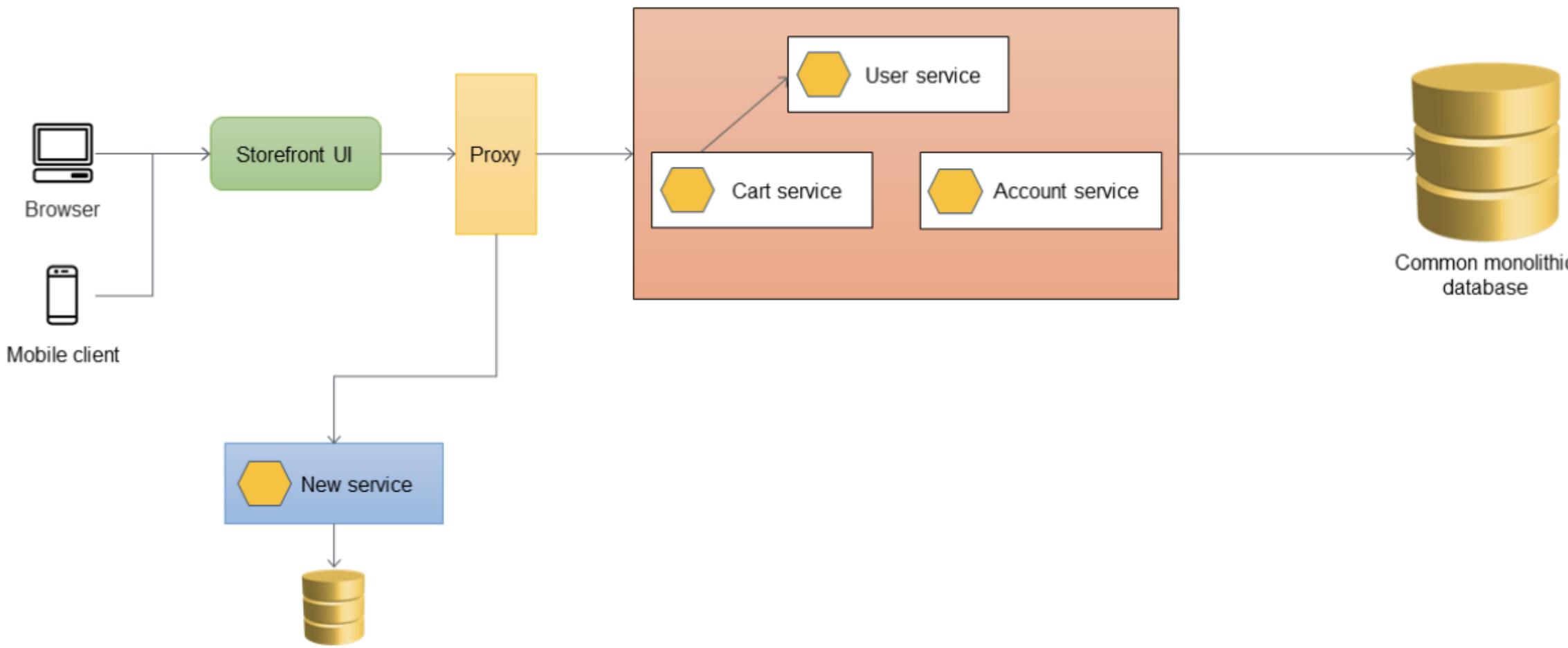
- **Identify Replaceable Components:** Start with parts of the system that are easiest to replace or most in need of an upgrade.
- **Build New Features as Services:** Develop new functionalities as separate services outside the legacy system.
- **Reroute Traffic:** Gradually reroute user traffic from the old system to the new services.



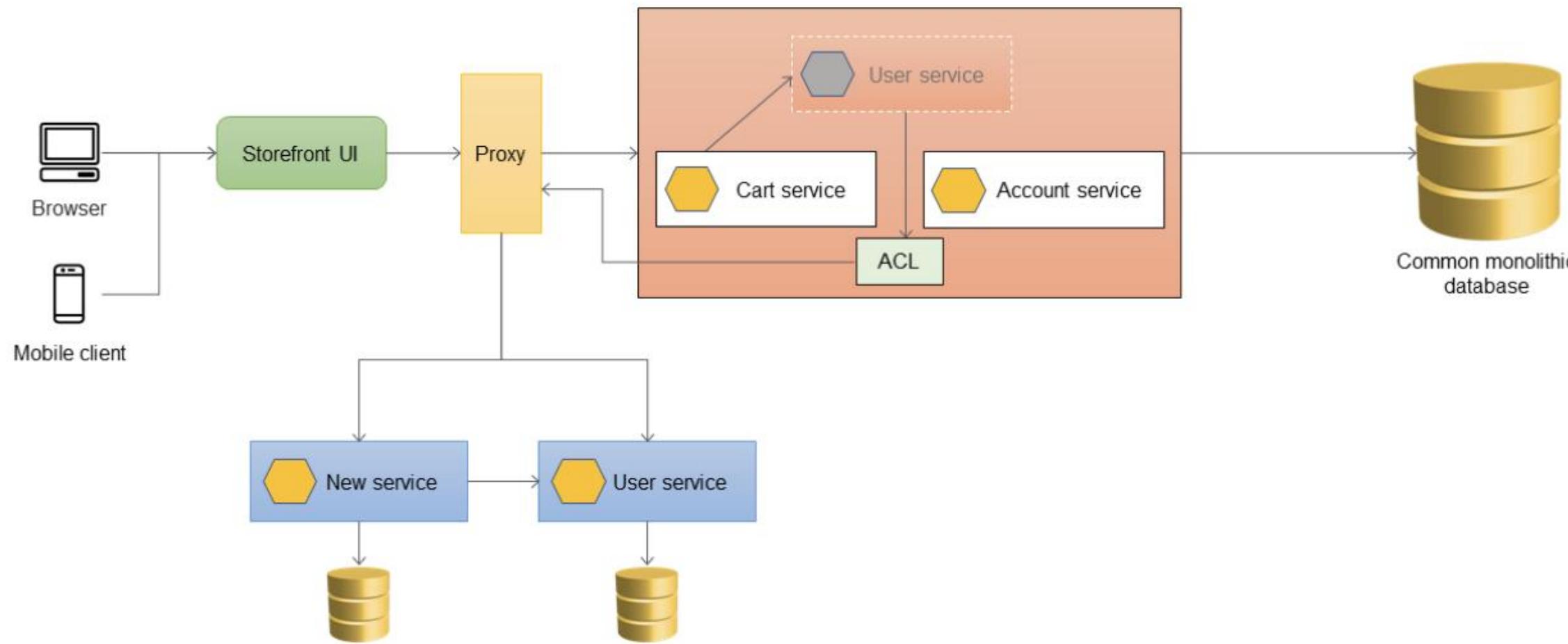
A monolithic application has three services: user service, cart service, and account service. The cart service depends on the user service, and the application uses a monolithic relational database.



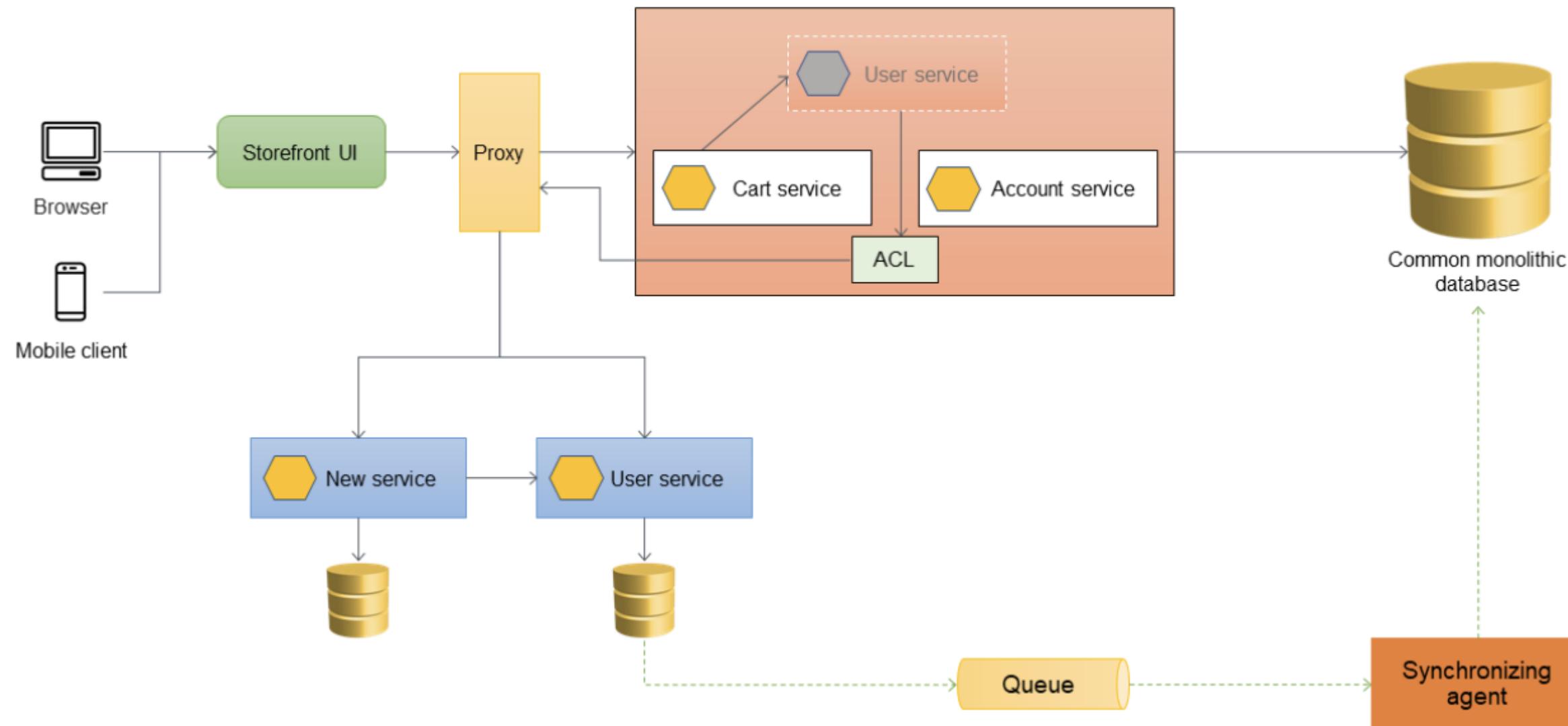
First step is to add a proxy layer between the storefront UI and the monolithic application. At the start, the proxy routes all traffic to the monolithic application.



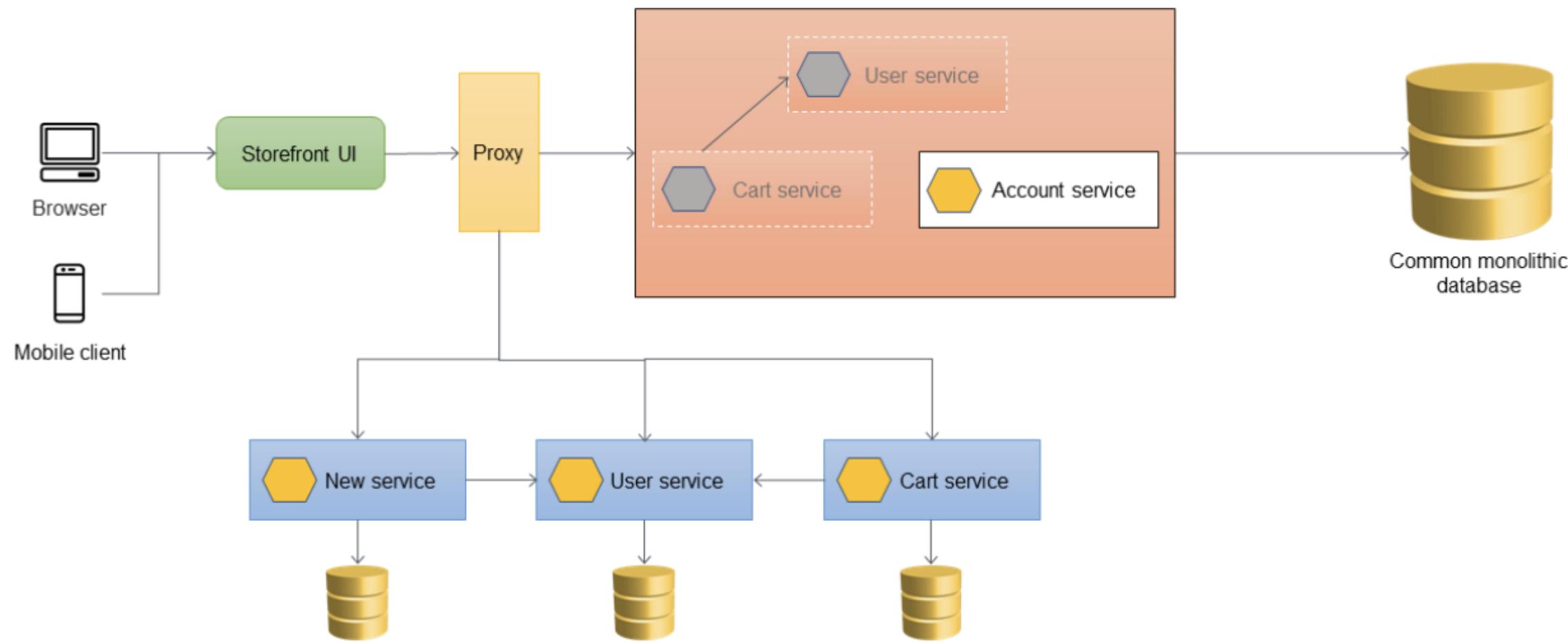
New services are implemented as microservices instead of adding features to the existing monolith. However, you continue to fix bugs in the monolith to ensure application stability. The proxy layer routes the calls to the monolith or to the new microservice based on the API URL.



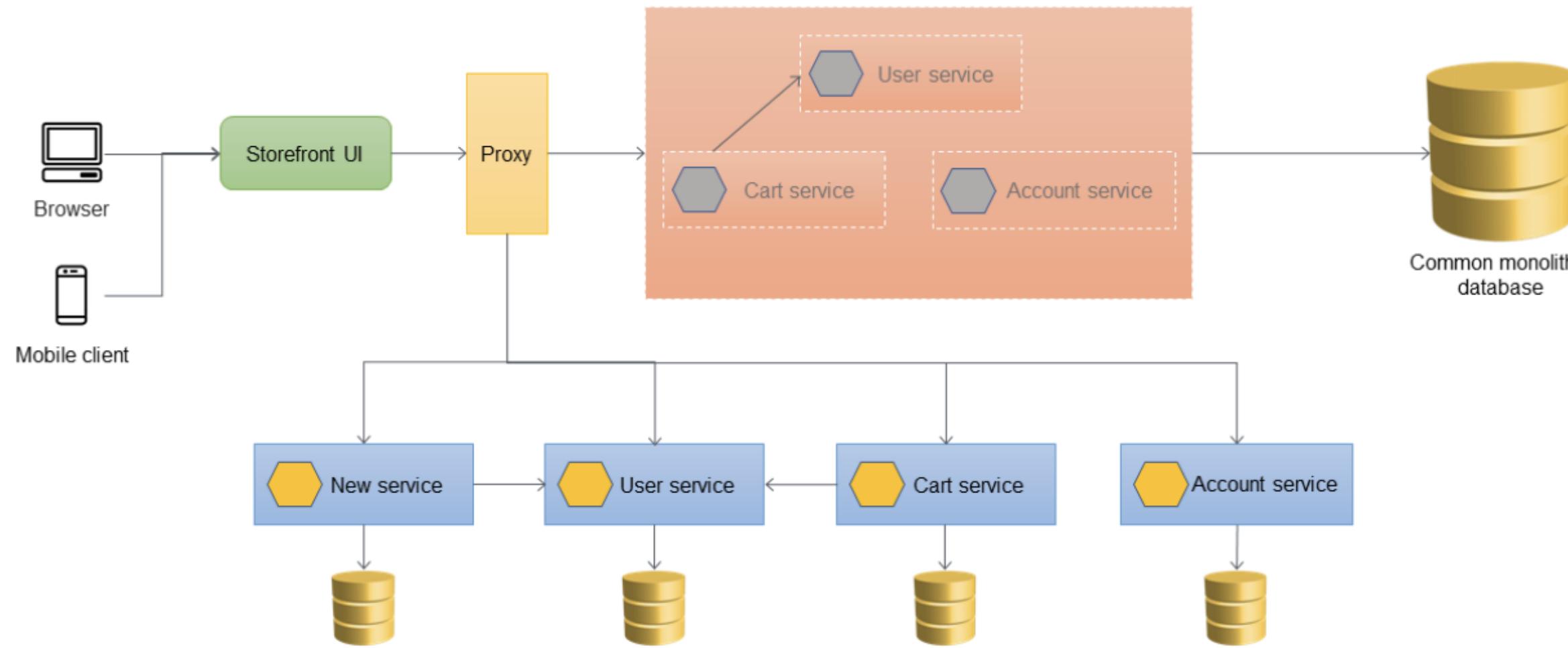
During the migration process, when the features within the monolith need to call the features that were migrated as microservices, the ACL converts the calls to the new interface and routes them to the appropriate microservice.



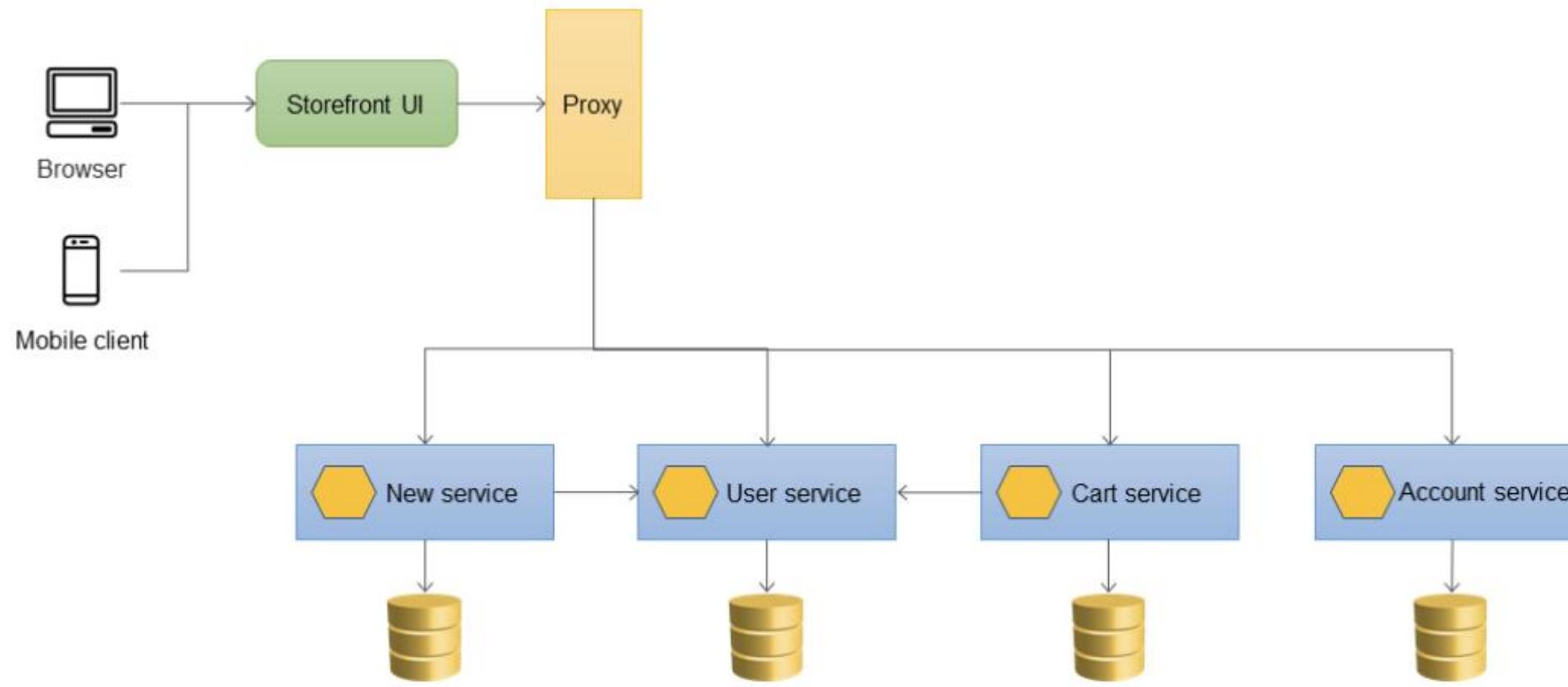
Data synchronization is crucial when downstream services, rely on a monolithic architecture, use data from a microservice. In this scenario, a User microservice, which has its own data layer, requires synchronization with the monolith. To facilitate this, a synchronization agent can be introduced. Update events from the microservice's database are sent to a queue. The agent then reads these events from the queue and synchronizes them with the monolithic database, ensuring eventual data consistency between the microservice and the monolith.



Once all interdependent components have been fully migrated to microservices, it becomes feasible to refactor the code to eliminate the Anti-Corruption Layer (ACL) components.



The final strangled state where all services have been migrated out of the monolith and only the skeleton of the monolith remains. Historical data can be migrated to data stores owned by individual services. The ACL can be removed, and the monolith is ready to be decommissioned at this stage.



The final architecture after the monolithic application has been decommissioned.

# Microservice Design Patterns

In a distributed transaction, multiple services can be called before a transaction is completed. When the services store data in different data stores, it can be challenging to maintain data consistency across these data stores.

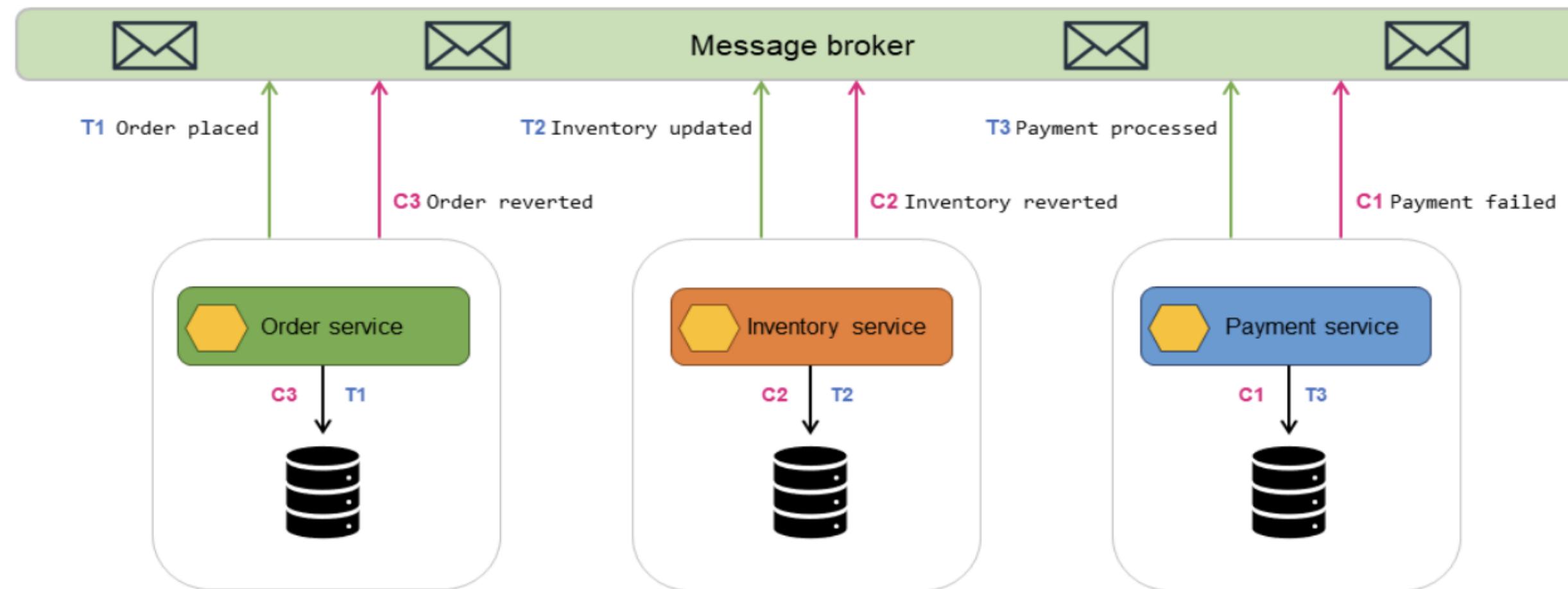
**Saga Pattern** - A *saga* consists of a sequence of local transactions. Each local transaction in a saga updates the database and triggers the next local transaction. If a transaction fails, the saga runs compensating transactions to revert the database changes made by the previous transactions.

- Manages Distributed Transactions Across Multiple Microservices and Databases.
- Breaks Down a Transaction into a Series of Local Transactions for Each Service.
- Addresses the challenge of maintaining atomicity, consistency, isolation, and durability in microservices.
- Utilizes Compensating Transactions or Actions in Case of Local Transaction Failures.
- Avoids Using Two-Phase Commit Protocols for Transaction Management.
- Maintains Overall System Consistency Through Compensatory Mechanisms.

# Saga Choreography

- Ensures data integrity in distributed transactions across multiple services using event subscriptions.
- Depends on the events published by the microservices.
- The saga participants (microservices) subscribe to the events and act based on the event triggers.



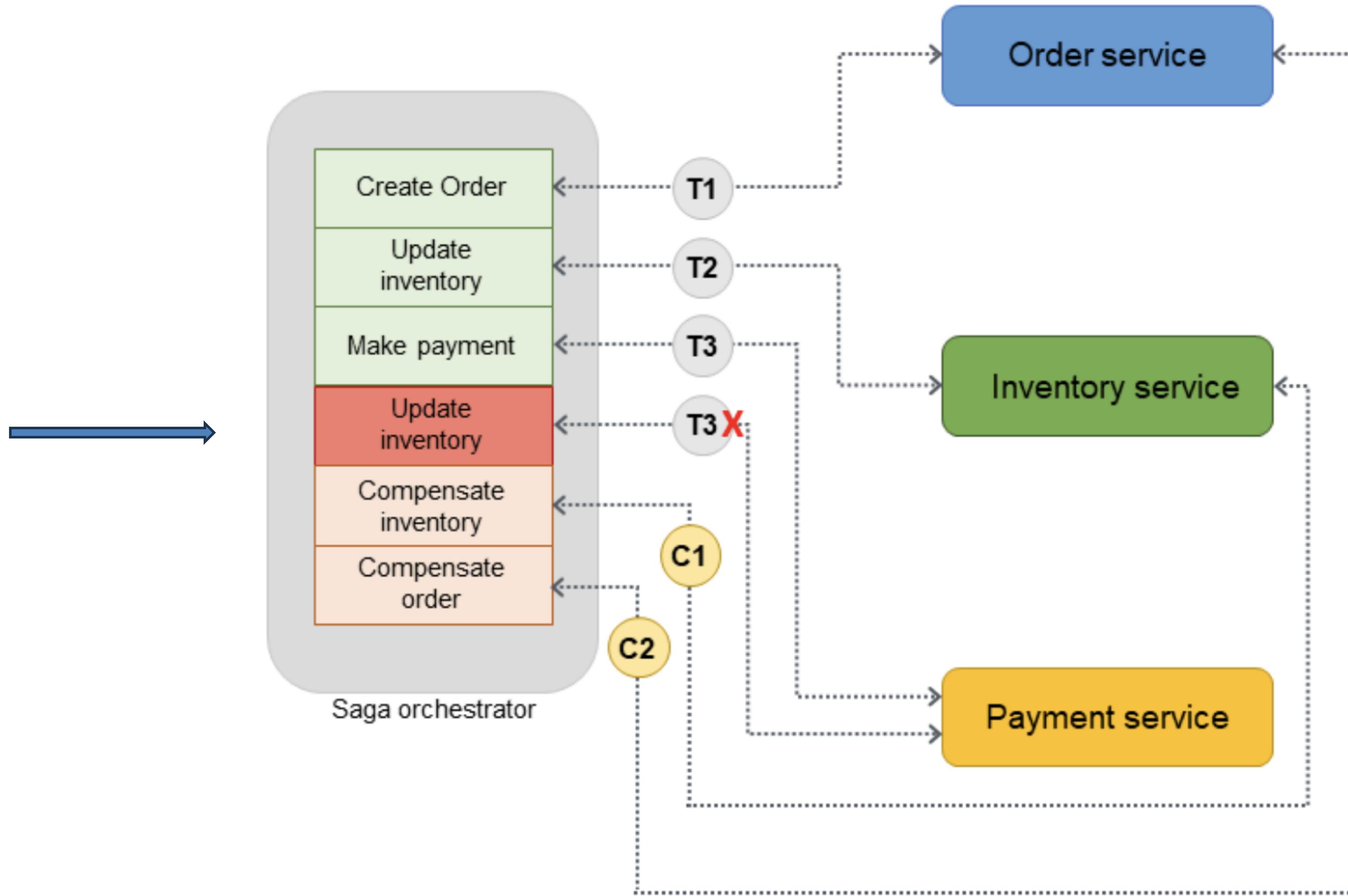


## **Use the saga choreography pattern when:**

- Your system requires data integrity and consistency in distributed transactions that span multiple data stores.
- The data store (for example, a NoSQL database) doesn't provide 2PC to provide ACID transactions, you need to update multiple tables within a single transaction, and implementing 2PC within the application boundaries would be a complex task.
- A central controlling process that manages the participant transactions might become a single point of failure.
- When there are small number of participants (microservices) involved.
- The saga participants are independent services and need to be loosely coupled.
- There is communication between bounded contexts in a business domain.

# Saga Orchestrator

- Uses a central coordinator (*orchestrator*) to help preserve data integrity in distributed transactions that span multiple services.
- Utilizes the orchestrator to manage the transaction lifecycle. Orchestrator is aware of all the steps required for transaction.
- Orchestrator sends messages to participant microservices to initiate operations. These participant microservices report back to the orchestrator.
- Orchestrator acts as the decision maker and determines the next microservice to engage based on received messages.



Use the saga orchestration pattern when:

- Your system requires data integrity and consistency in distributed transactions that span multiple data stores.
- The data store doesn't provide 2PC to provide ACID transactions, and implementing 2PC within the application boundaries is a complex task.
- You have NoSQL databases, which do not provide ACID transactions, and you need to update multiple tables within a single transaction.
- Your system has many saga participants (microservices) involved and loose coupling between participants are required.

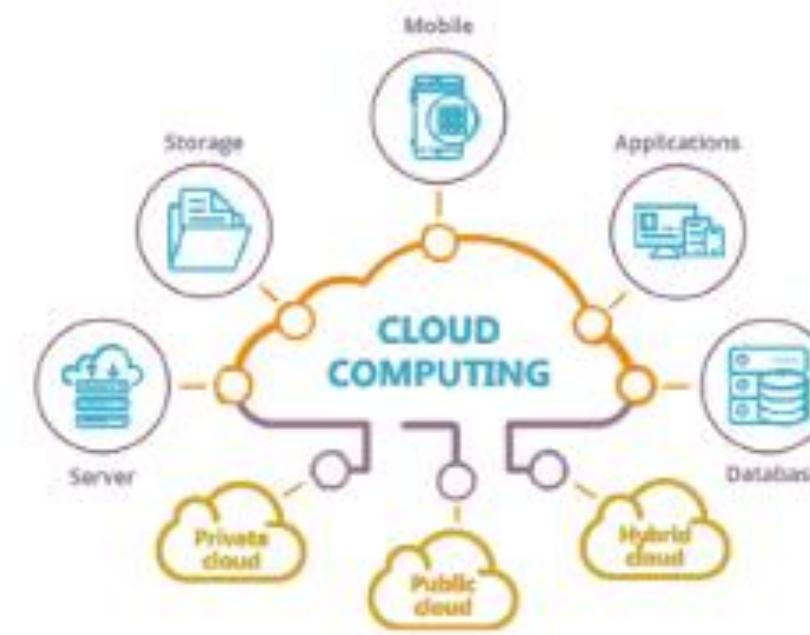
# Developing and deploying Microservices with K8s

- **Traffic Management with Ingress:**
  - Using Kubernetes Ingress for efficient HTTP/HTTPS routing to services.
  - Ingress acts as a reverse proxy, simplifying routing and providing SSL/TLS termination and load balancing.
- **Scaling Microservices:**
  - Leveraging tools like Horizontal Pod Autoscaler (HPA) for automatic scaling based on CPU usage or custom metrics.
  - Kubernetes also supports manual scaling for handling varying loads effectively.
- **Using Namespaces for Organization:**
  - Utilizing Kubernetes namespaces to divide cluster resources among multiple users or teams.
  - Grouping related services in the same namespace simplifies management and applies policies at the namespace level.
- **Implementing Health Checks:**
  - Essential for monitoring the status of services using readiness and liveness probes.
  - Health checks allow Kubernetes to replace non-functioning pods automatically.
- **Service Mesh for Advanced Traffic Management:**
  - Implementing a service mesh for handling service-to-service communication.
  - Provides traffic management, service discovery, load balancing, and failure recovery.

- **Single Responsibility Principle for Microservices:**
  - Design each microservice with a single responsibility for easier scaling, monitoring, and management.
  - Tailor scaling policies, resource quotas, and security configurations to the specific needs of each service.
- **Continuous Delivery/Deployment (CD):**
  - Utilizing Kubernetes Deployment objects for a declarative management of microservices.
  - Implement rolling updates for gradual change rollout and use tools like Argo Rollouts for more reliable rollback and progressive deployment strategies.
- **Monitoring and Debugging:**
  - Collect and visualize metrics using tools like Prometheus and Grafana.
  - Use application performance monitoring (APM) tools for detailed performance data of microservices.

# References

- <https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns/introduction.html>
- <https://learn.microsoft.com/en-us/azure/architecture/patterns/>



# Cloud Design Patterns

Ravindu Nirmal Fernando

SLIIT | March 2025

# Design Patterns

---

A generally reusable solution to a recurring problem

- ❑ A template to solve the problem
- ❑ Best practices in approaching the problem
- ❑ Improve developer communication

# Cloud Application Development Issues

---

## Availability

- The guaranteed proportion of time that the system is functional

## SLA – Service Level Agreement

Availability (%)	Downtime per year
99	3.7 days
99.9	9 hours
99.95	4.4 hours
99.99	1 hour
99.999	5 minutes

# Cloud Application Development Issues

---

## Data Management

- Typically hosted in different locations and across multiple servers for performance, scalability and availability
- Maintaining consistency and synchronizing

## Design and Implementation

- Consistent and coherent component design
- Improves ease of deployment and maintenance
- Reusability of components

# Cloud Application Development Issues

---

## ?

### Messaging

- ❑ Messaging infrastructure to connect distributed components and services
- ❑ Asynchronous messaging

## ?

### Design and Implementation

- ❑ Consistent and coherent component design
- ❑ Improves ease of deployment and maintenance
- ❑ Reusability of components

# Cloud Application Development Issues

---

## ?

### Management and Monitoring

- ?
- Cloud applications run in in a remote servers with limited control

## ?

### Performance and Scalability

- ?
- Responsiveness of a system to execute any action within a given time interval
- Handle increases in load without impact on performance
- ?
- How to handle variable workloads?

# Cloud Application Development Issues

---

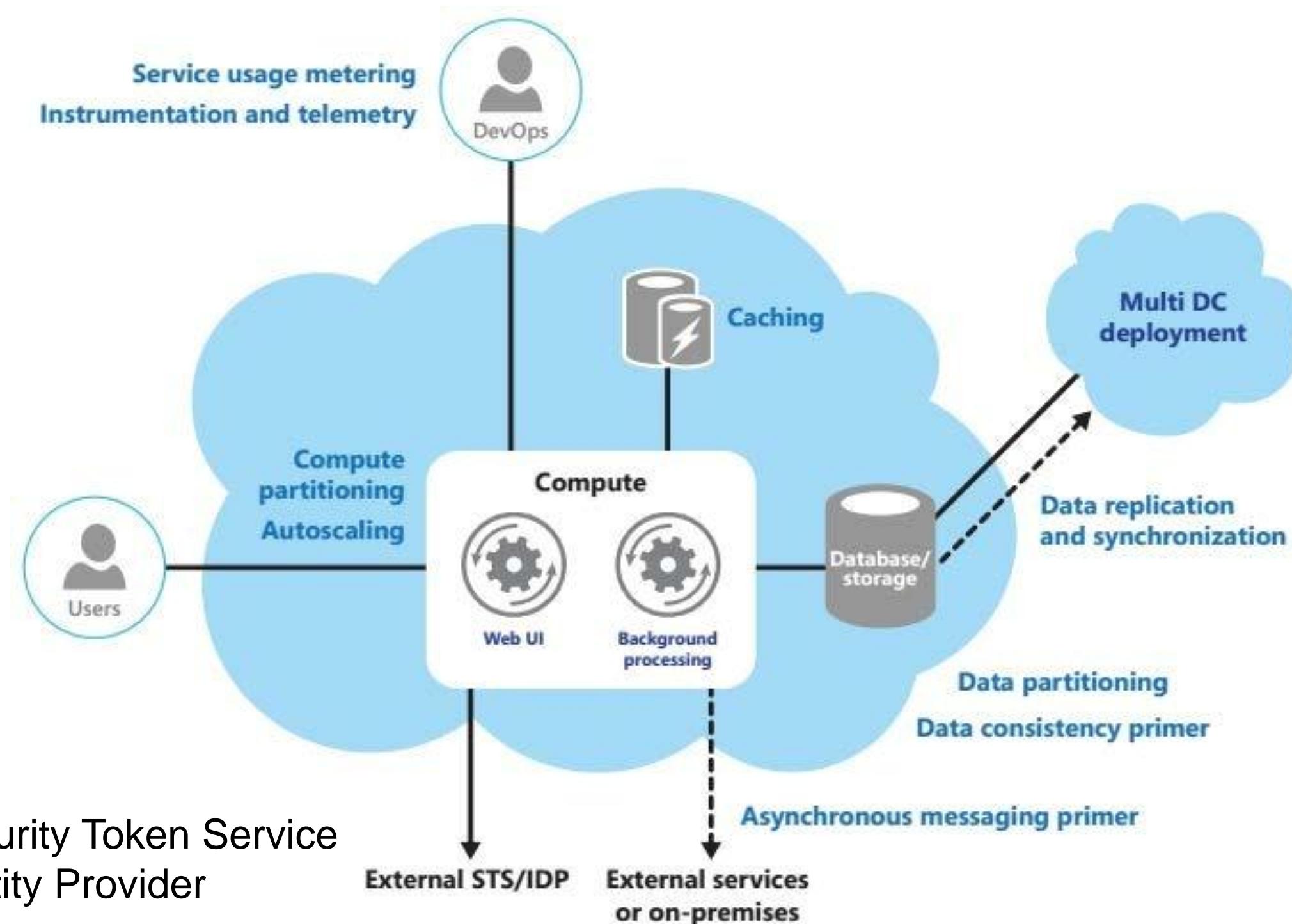
## ■ Resiliency

- Ability of the application to gracefully handle and recover from failures
- Applications are more prone to failure in cloud environments

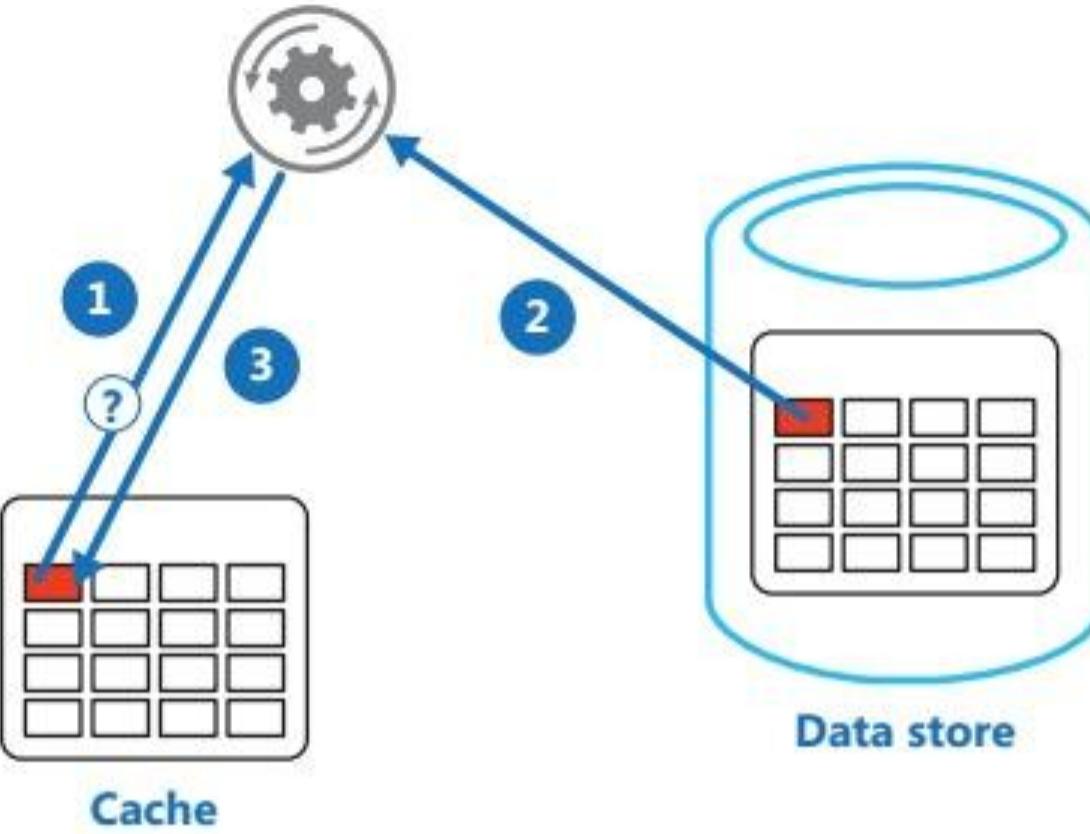
## ■ Security

- Prevent malicious or accidental actions outside of the designed usage
- Prevent disclosure or loss of information

# High-Level Model



# Cache-Aside Pattern



- 1: Determine whether the item is currently held in the cache.
- 2: If the item is not currently in the cache, read the item from the data store.
- 3: Store a copy of the item in the cache.

- ?] Load on demand data into a cache from a data store
- ?] Pros
  - Increased performance
- ?] Cons
  - Maintaining consistency between data in cache & data in underlying data store
- ?] Solutions
  - Azure Cache AWS ElastiCache
  - Google App Engine memcache
  - Redis Cache

# Cache-Aside Pattern (Cont.)

---

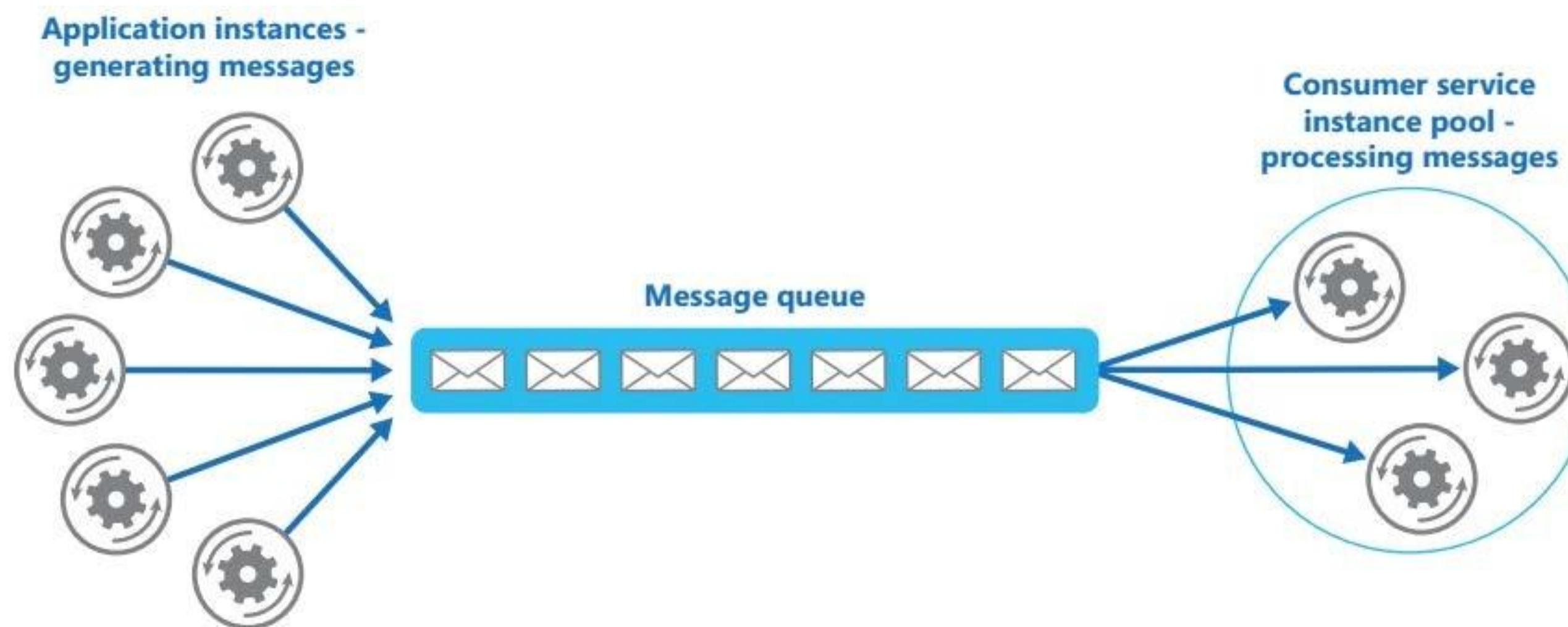
## When

- Read/write performance

## Parameters

- What to cache
- Lifetime of cached data
- Cache size
- Evicting data In Memory
- Caching

# Competing Consumers Pattern



- Multiple concurrent consumers to process messages received on same channel
- Goals
  - Optimize throughput, improve scalability & availability, load balancing

# Competing Consumers Pattern (Cont.)

---

## When

- Independent tasks that can be processed parallel
- Volume of work is highly variable
- High availability

# Competing Consumers Pattern (Cont.)

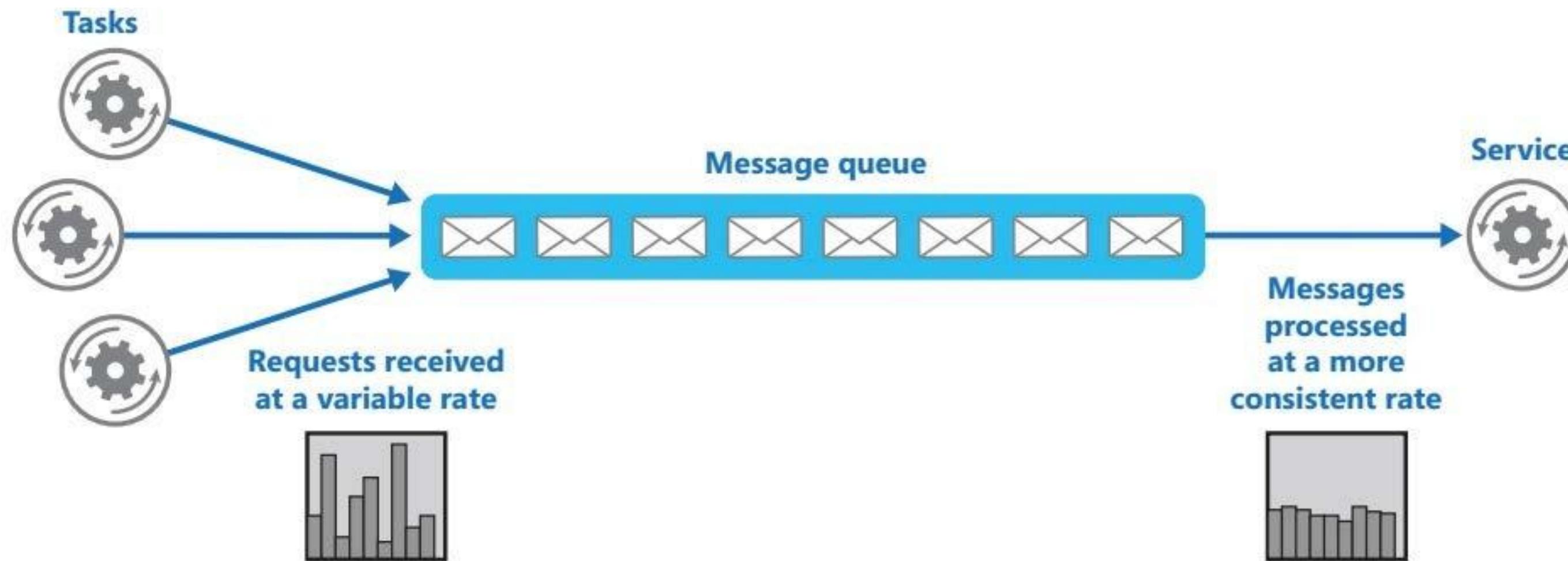
---

## Parameters

- Queue size
- Scaling
- Not loosing messages
- Preserving message ordering
- Resiliency
- Poison/malformed messages
- Returning results

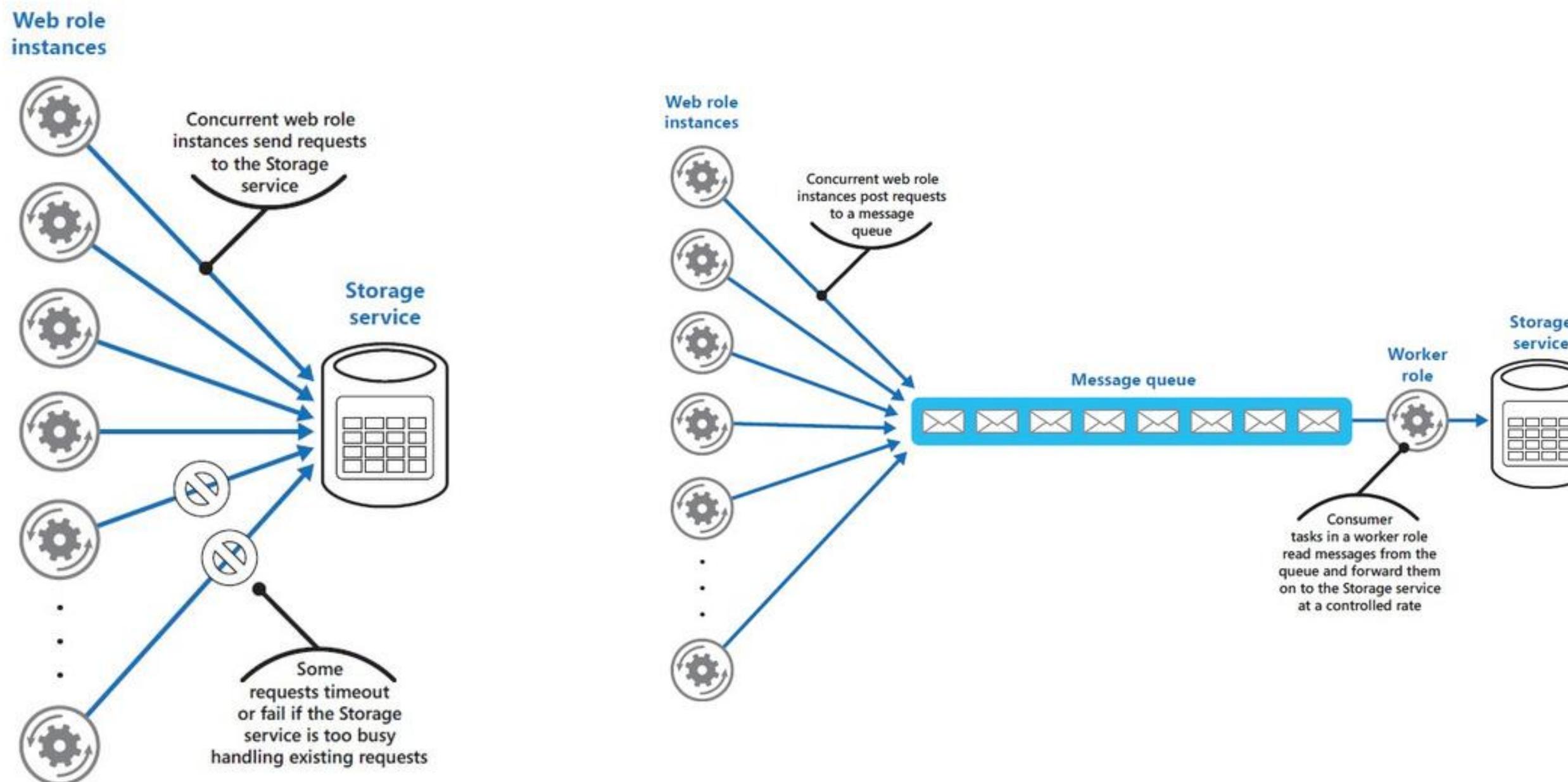
# Queue-Based Load Leveling Pattern

---



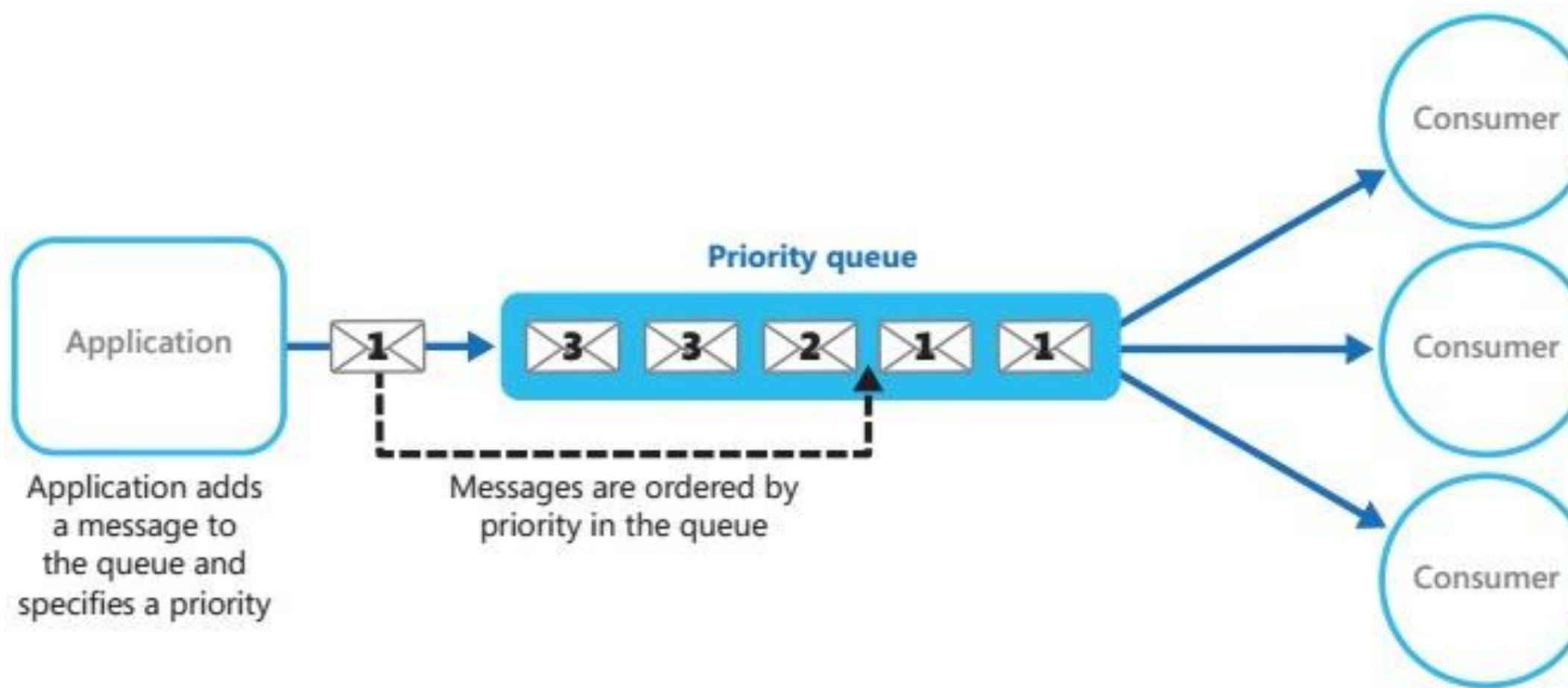
- To smooth intermittent heavy loads that may otherwise cause the service to fail or the task to time out

# Queue-Based Load Leveling Pattern



# Priority Queue Pattern

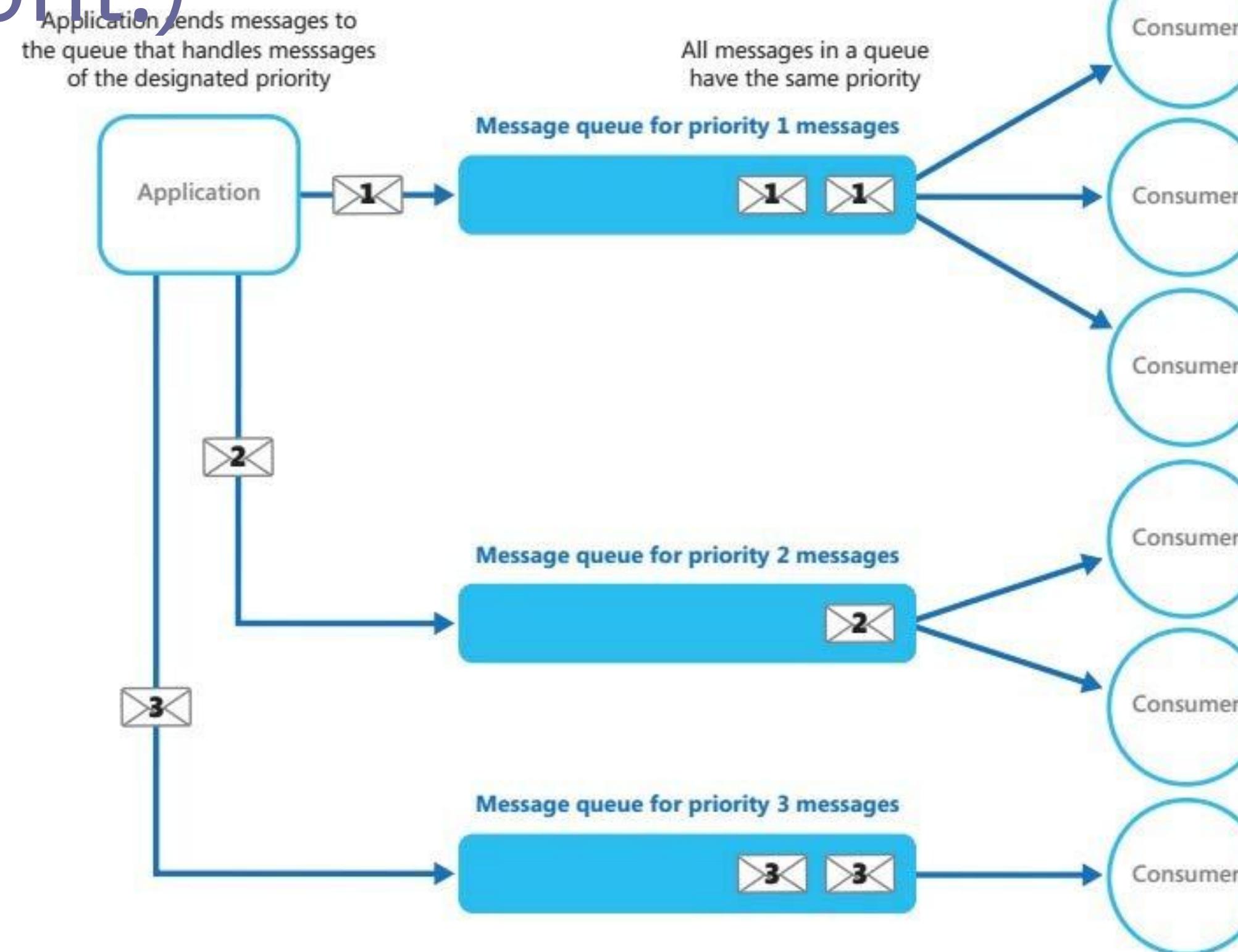
---



- Prioritize requests sent to services so that requests with a higher priority are received & processed quickly

# Priority Queue Pattern

(Cont.)



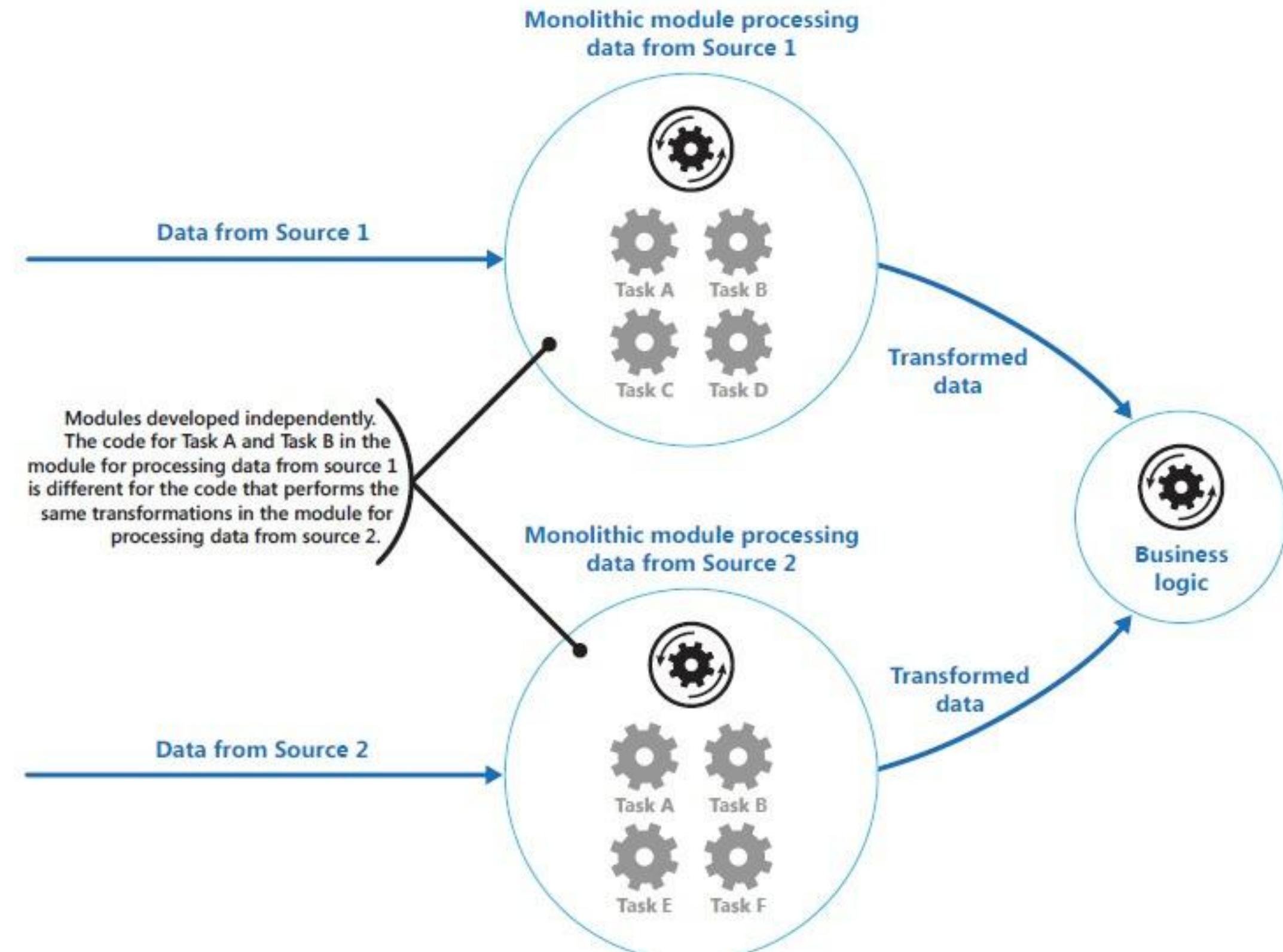
# Priority Queue Pattern (Cont.)

---

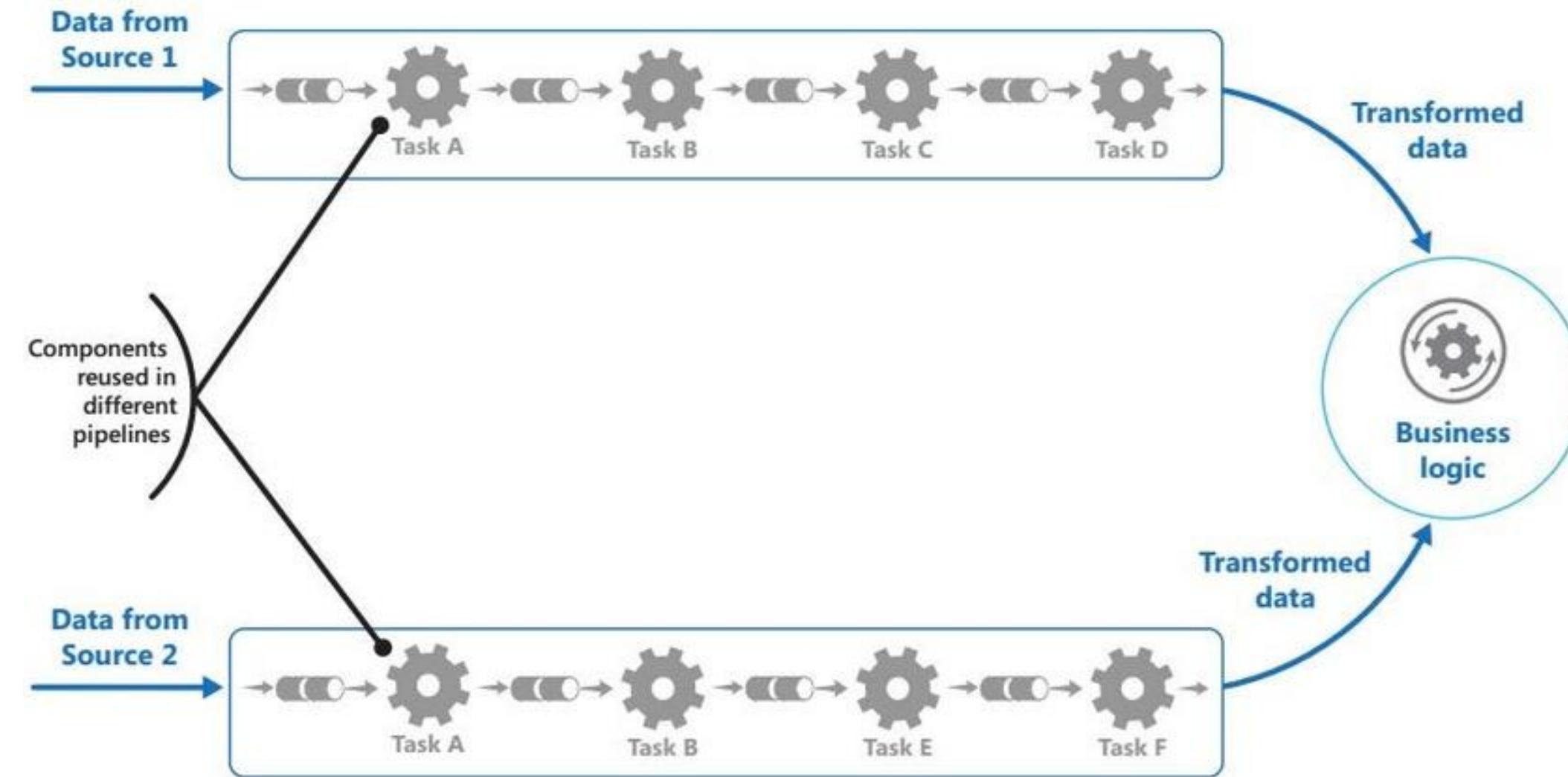
When,

- The system handles multiple tasks that have different priorities
- Different users should be served with different priorities

# Pipes & Filters Pattern



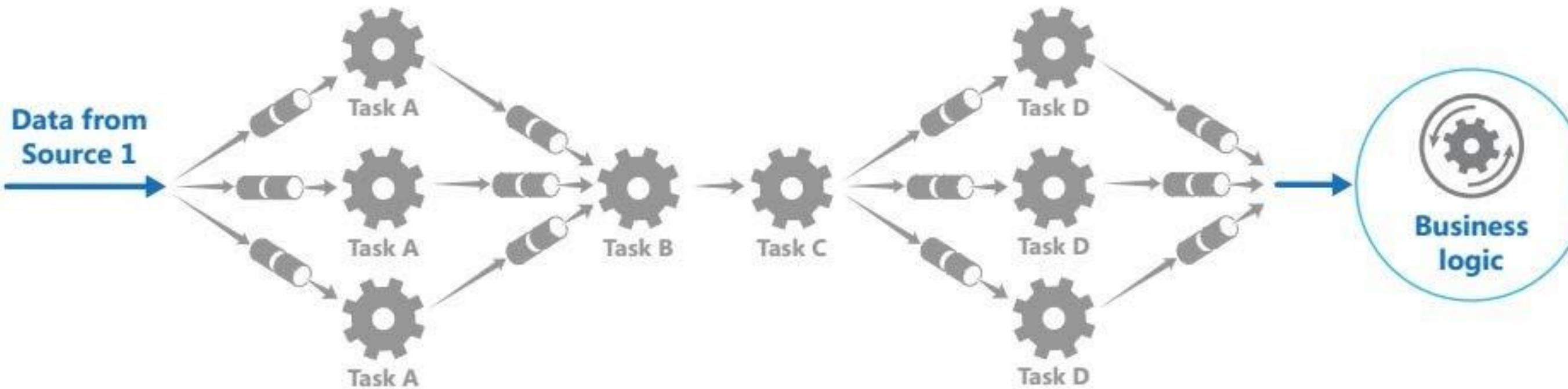
# Pipes & Filters Pattern (Cont.)



- 💡 Decompose a task that performs complex processing into a series of discrete elements that can be reused

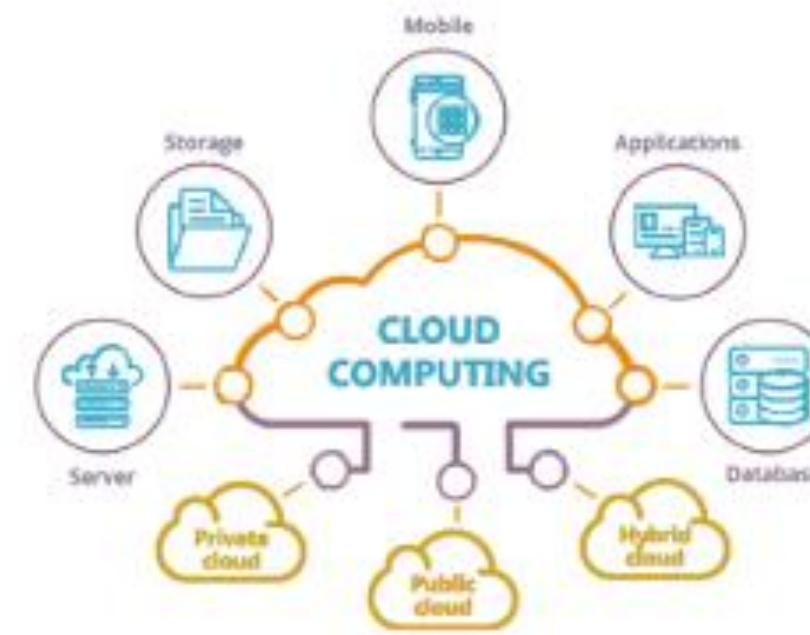
# Pipes & Filters Pattern – With Load Balancing

---



□ When,

- ❑ Application can be decomposed to steps
- ❑ Steps have different scalability requirements
- ❑ Flexibility of processing
- ❑ Need distributed processing



# Cloud Design Patterns

Ravindu Nirmal Fernando

SLIIT | March 2025

# Load Balancing

---

- Improves the distribution of workloads across multiple computing resources
  - Some resources will be busy while others are idle
  
- Aims to
  - Optimize resource use
  - Maximize throughput
  
  - Minimize response time
  - Avoid overload of any single resource

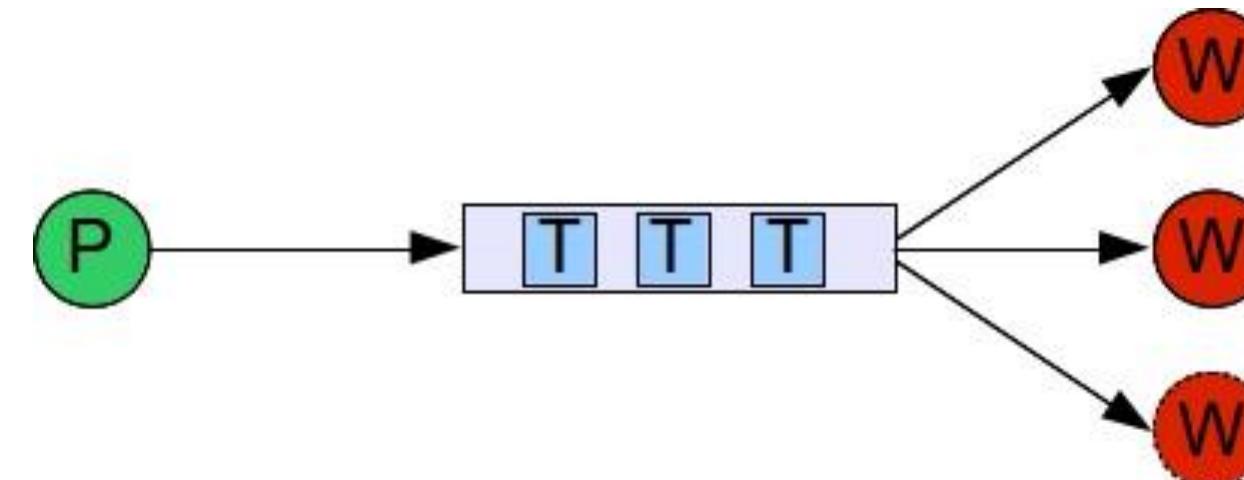
# Load Balancing

---

- ▣ Counter by distributing load equally
  - ▢ When cost of problem is well understood (e.g., matrix multiplication, known tree walk) this is possible
- ▣ Some other problems are not that simple
  - Hard to predict how workload will be distributed
  - dynamic load balancing used
  - But require communication between tasks
- ▣ 2 methods for dynamic load balancing
  - ▢ Task queues vs. work stealing

# Task Queues

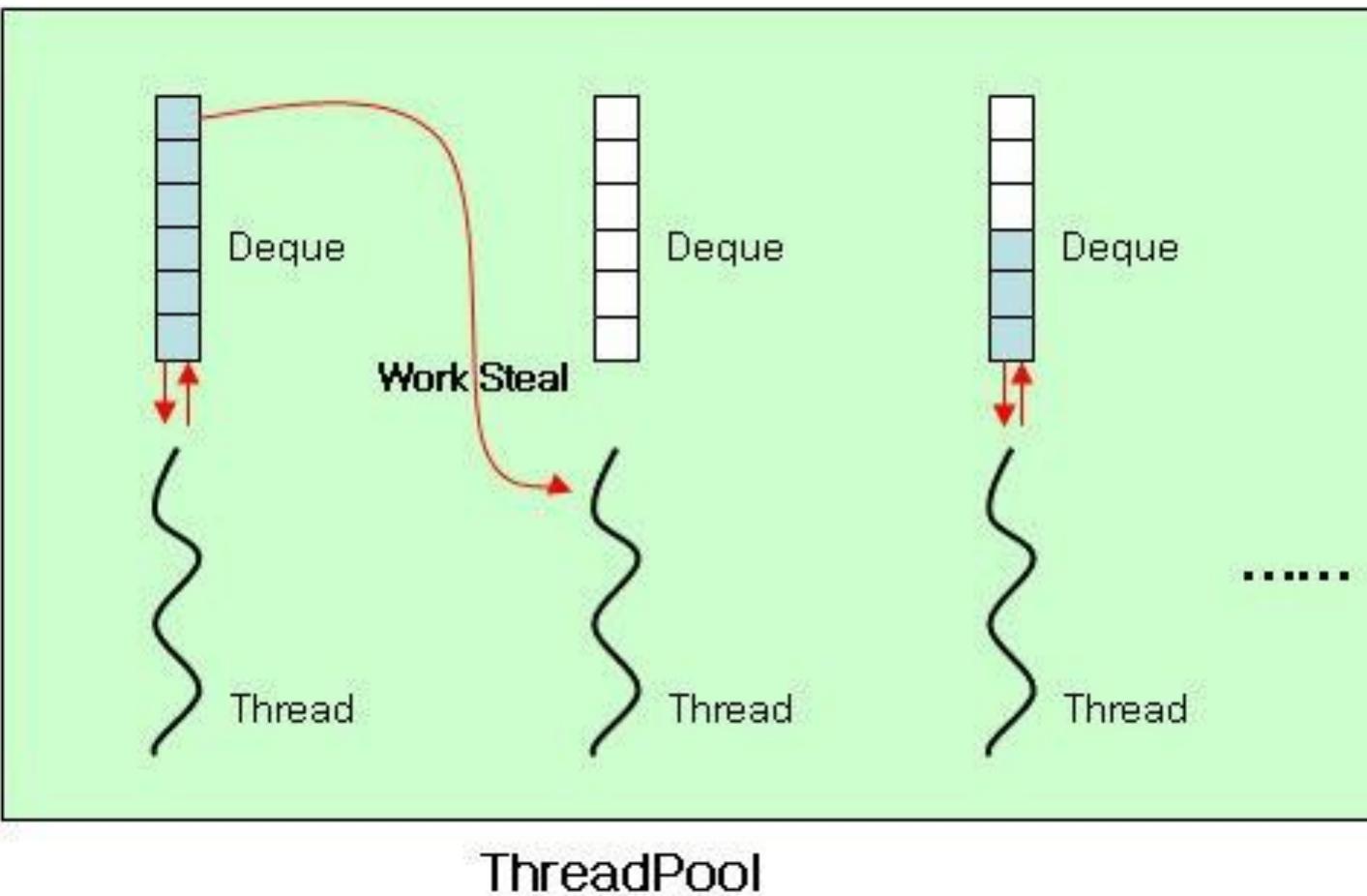
- Multiple instance of task queues (producer consumer)
- Threads comes to the task queue after finishing a task & grab next task
- Typically run with a pool of workers



# Work Stealing

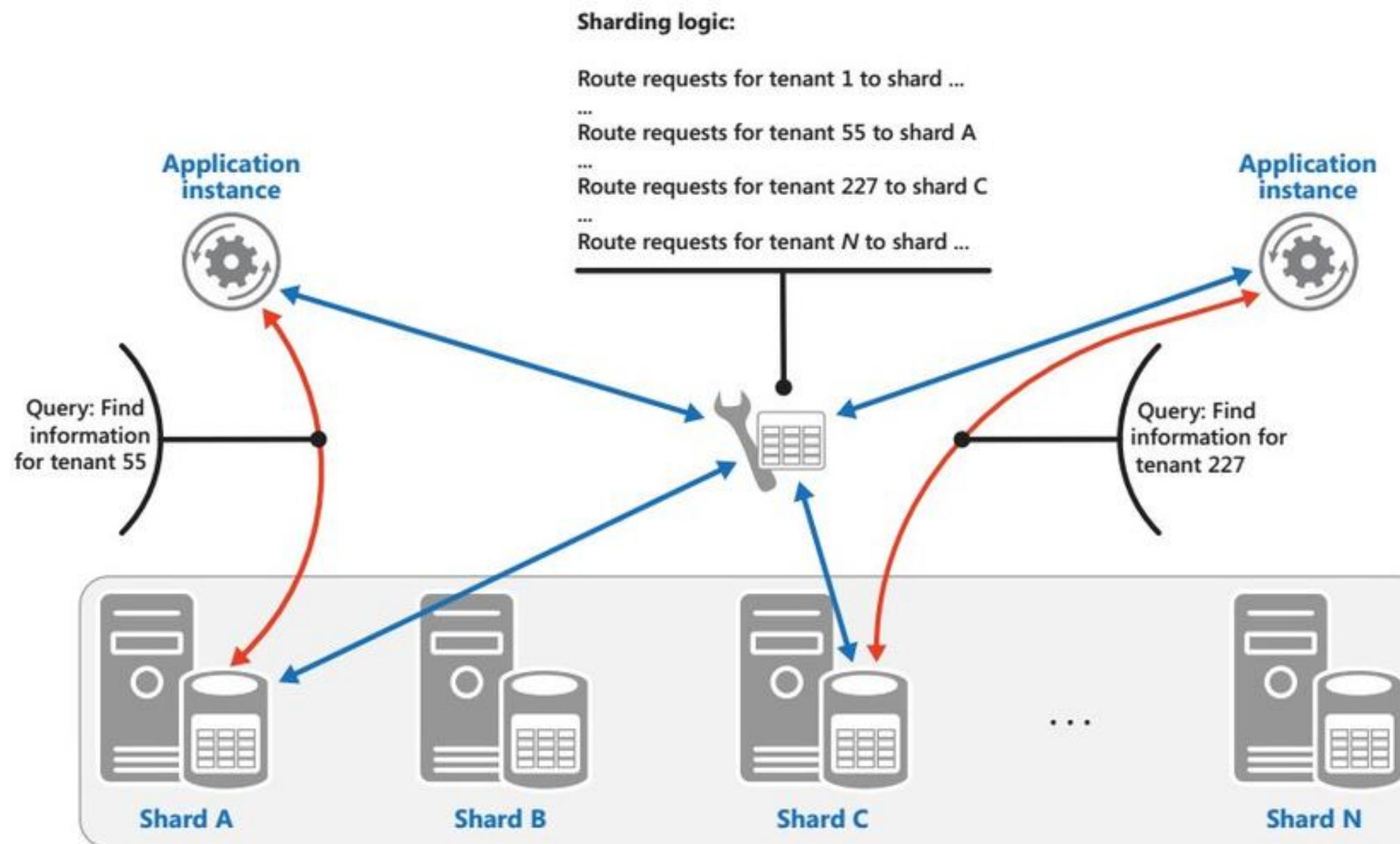
---

- ❑ Every worker has a task queue
- ❑ When 1 worker runs out of work, it goes to other worker's queue & “steal” the work



Source: <http://karlsenchoi.blogspot.com>

# Sharding Pattern



- Divide a data store into a set of horizontal partitions or shards to improve scalability

# Sharding Pattern (Cont)

---

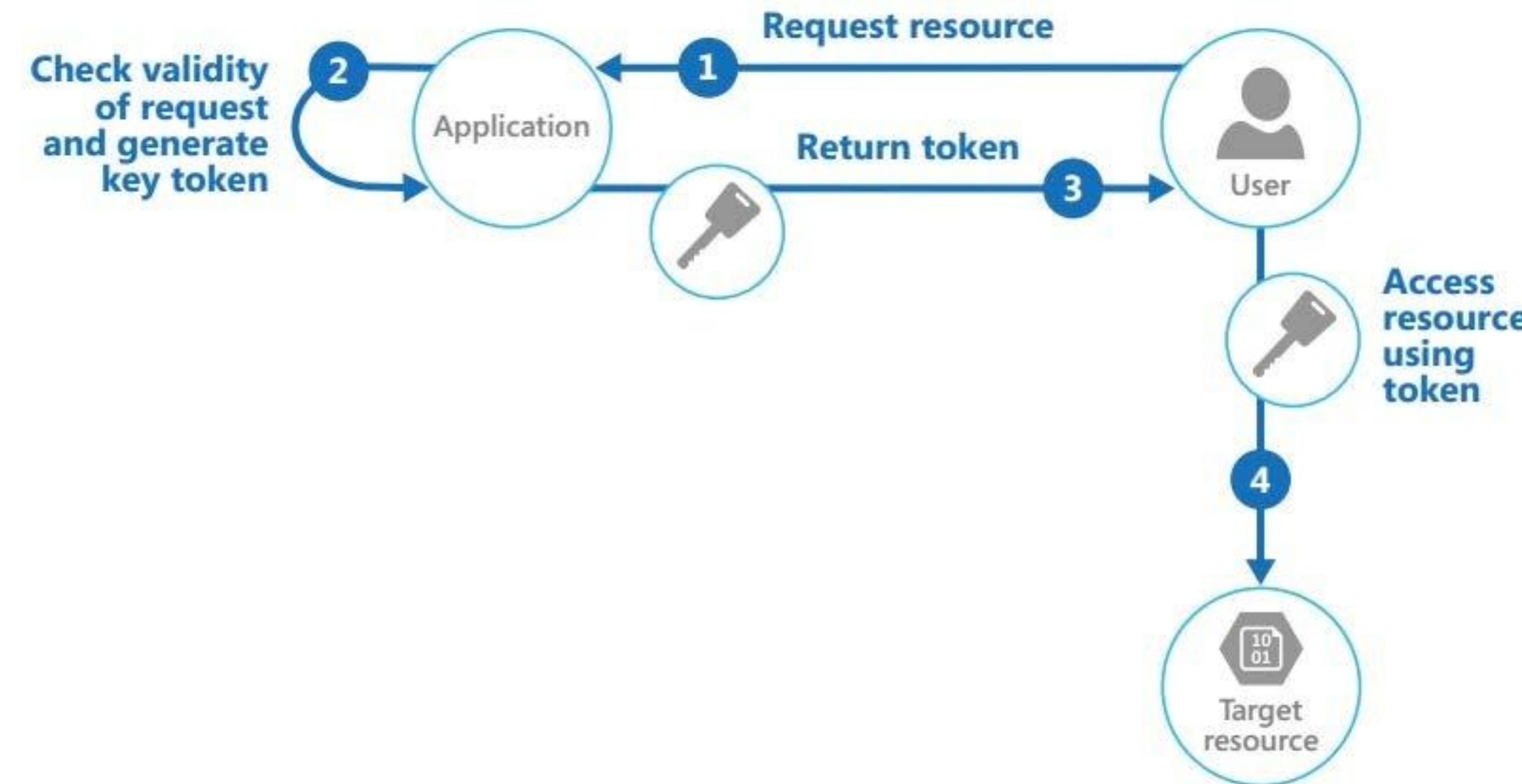
## □ When

- ❑ Limited storage space
- ❑ Large computation requirement
- ❑ Network bandwidth
- ❑ Geographical constraints

## ❑ Sharding Strategies

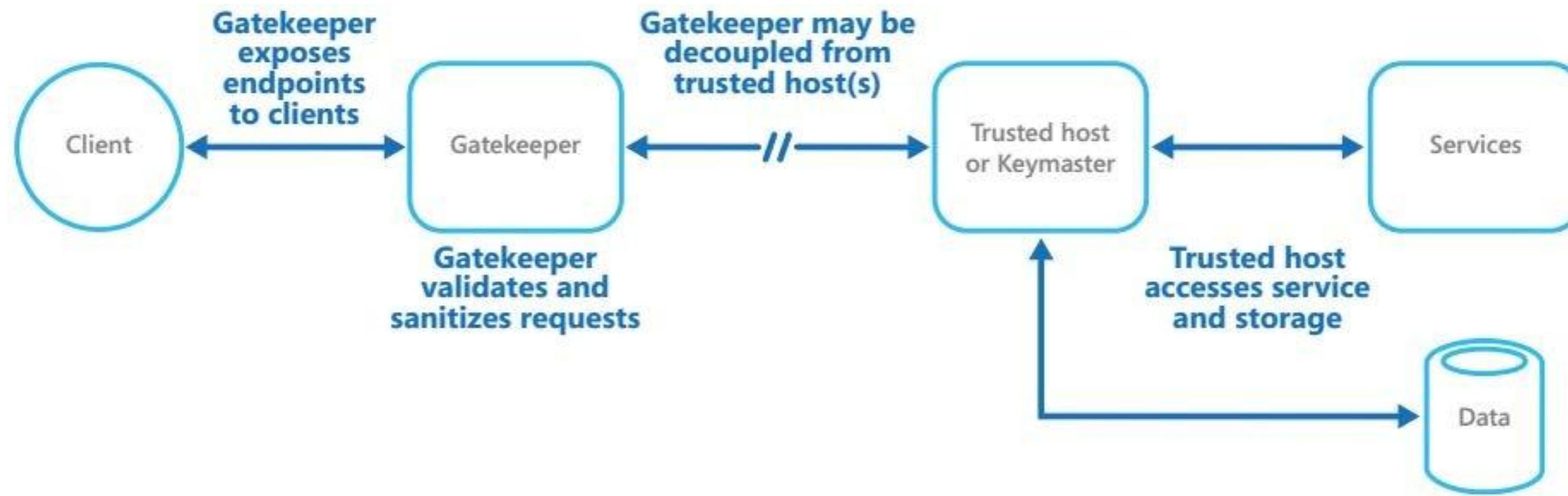
- ❑ Lookup Strategy – map request using a shard key
- ❑ Range Strategy – groups related items together in the same shard
- ❑ Hash Strategy – shard decided based on hashing data attributes

# Valet Key Pattern



- Use a token or key that provides client with restricted direct access to a specific resource or service

# Gatekeeper Pattern



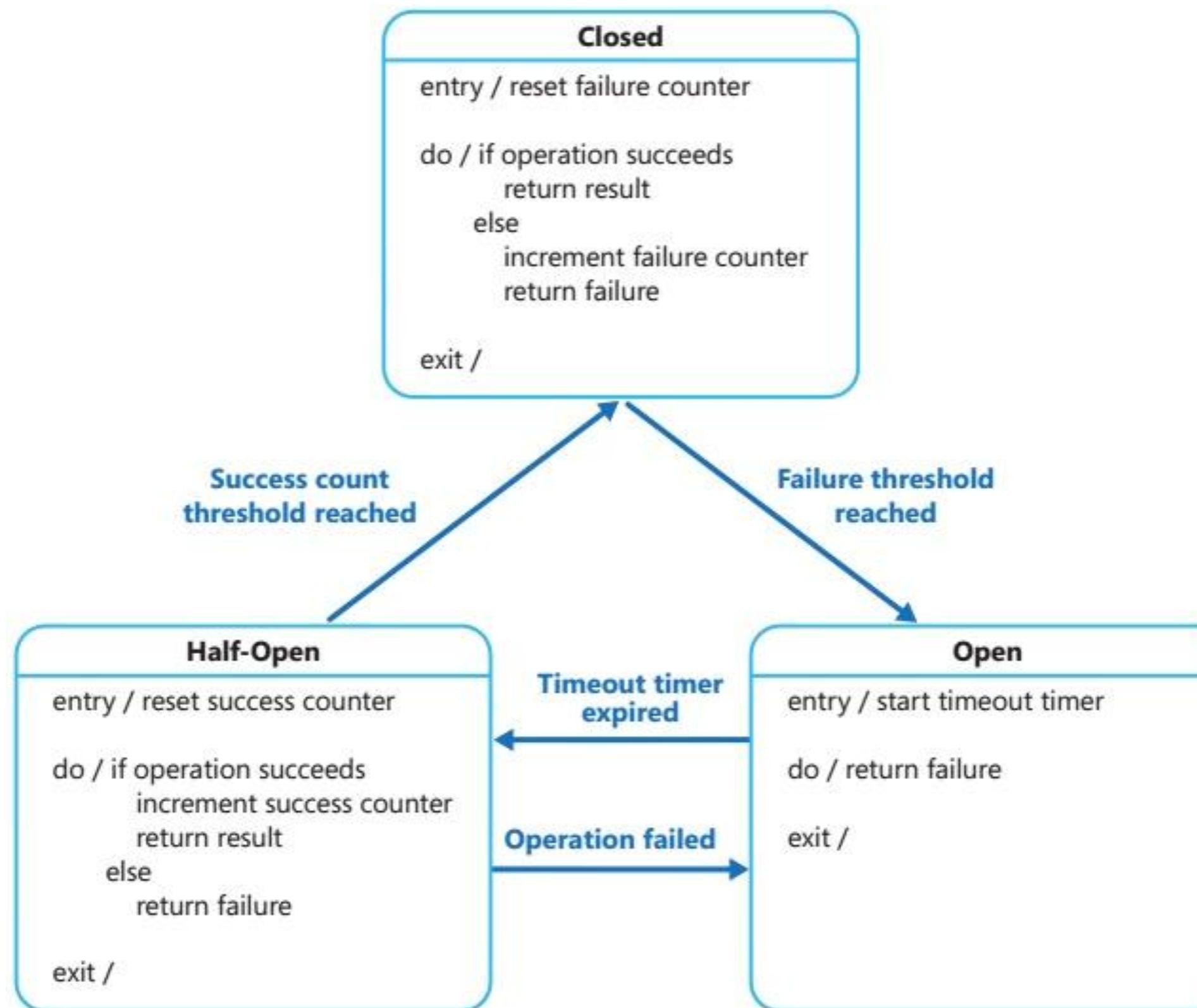
- Protect applications & services using a dedicated host instance that acts as a broker
  - Validates & sanitizes requests
  - Passes requests & data between them

# Gatekeeper Pattern (Cont.)

---

- ❑ Only function is to validate & sanitize requests
- ❑ Should use secure communication between gatekeeper & trusted hosts
- Internal end point must connect only to gatekeeper
- ❑ Gatekeeper must run in limited privilege mode
- ❑ May use multiple gatekeepers for availability

# Circuit Breaker Pattern



# Circuit Breaker Pattern (Cont.)

---

## □ When

- ❑ Handle faults that may take a variable amount of time to rectify when connecting to a remote service/resource
- ❑ When a simple retry will not work
- ❑ Prevent application from getting tied-up due to retry

## □ Half-Open State

- ❑ Allow checking whether service is responding by issuing a limited set of requests
- ❑ Prevent repeated system failures due to rapid load/volume

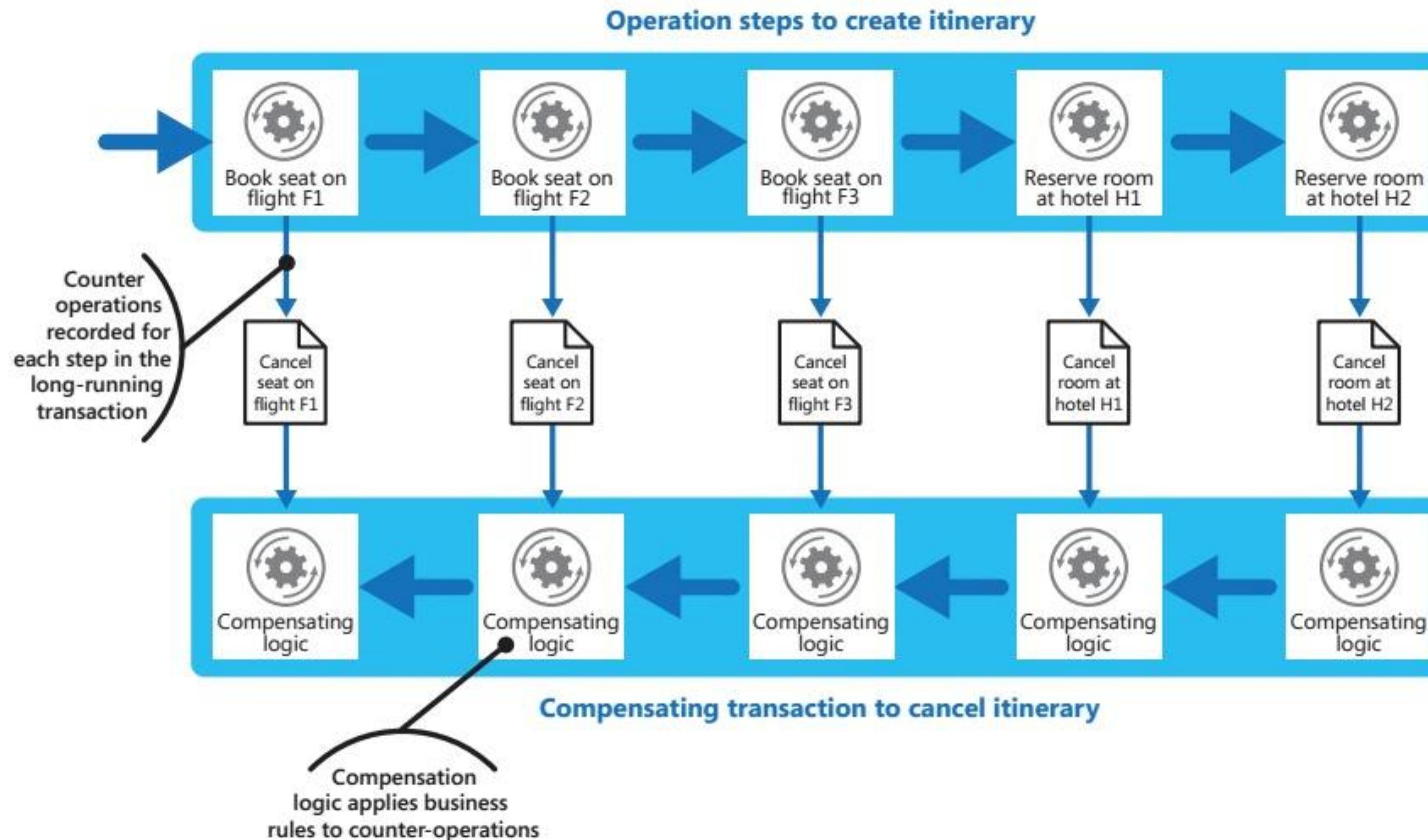
# Circuit Breaker Pattern (Cont.)

---

## Parameters

- ❑ Types of exceptions
- ❑ Handling exceptions
- ❑ Logging & replay
- ❑ Testing failed operations
- ❑ Manual reset

# Compensating Transaction Pattern



# Compensating Transaction Pattern (Cont.)

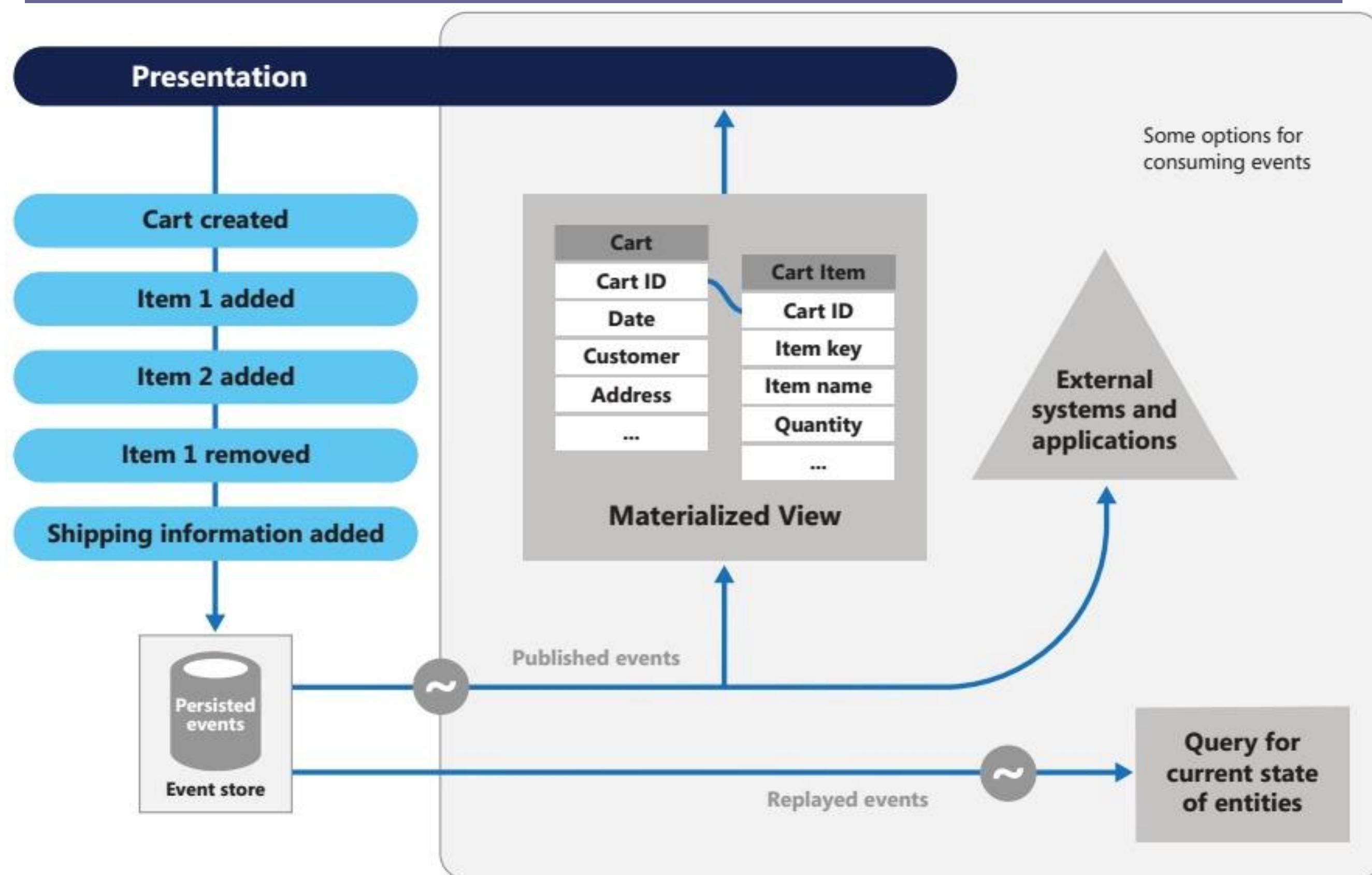
□ Undo work performed by a series of steps, which

together define an eventually consistent operation

□ Implement a workflow

- As operation proceeds, system records information about each step & how the work by that step can be undone
- If operation fails at any point, workflow rewinds back through steps it has completed while performing work that reverses each step

# Event Sourcing Pattern



# Event Sourcing Pattern (Cont.)

---

Record full series of events than current state

Pros

- Avoid requirement to synchronize data

  - Traditional Create, Read, Update, & Delete (CRUD) model too slow

  - Improve performance with eventual consistency

- Scalability

- Responsiveness

- Provide consistency for transactional data

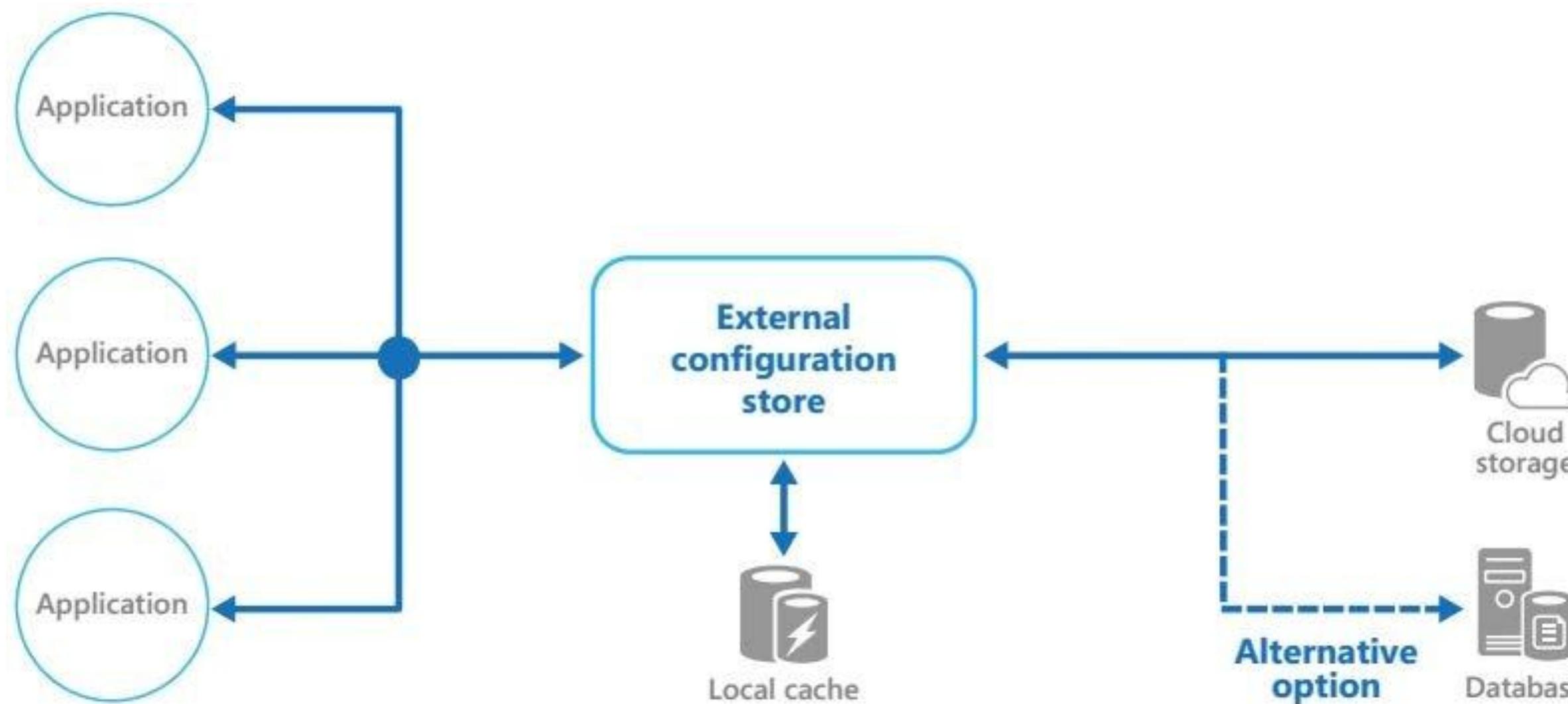
- Full audit trails

Cons

- Consistency relaxed

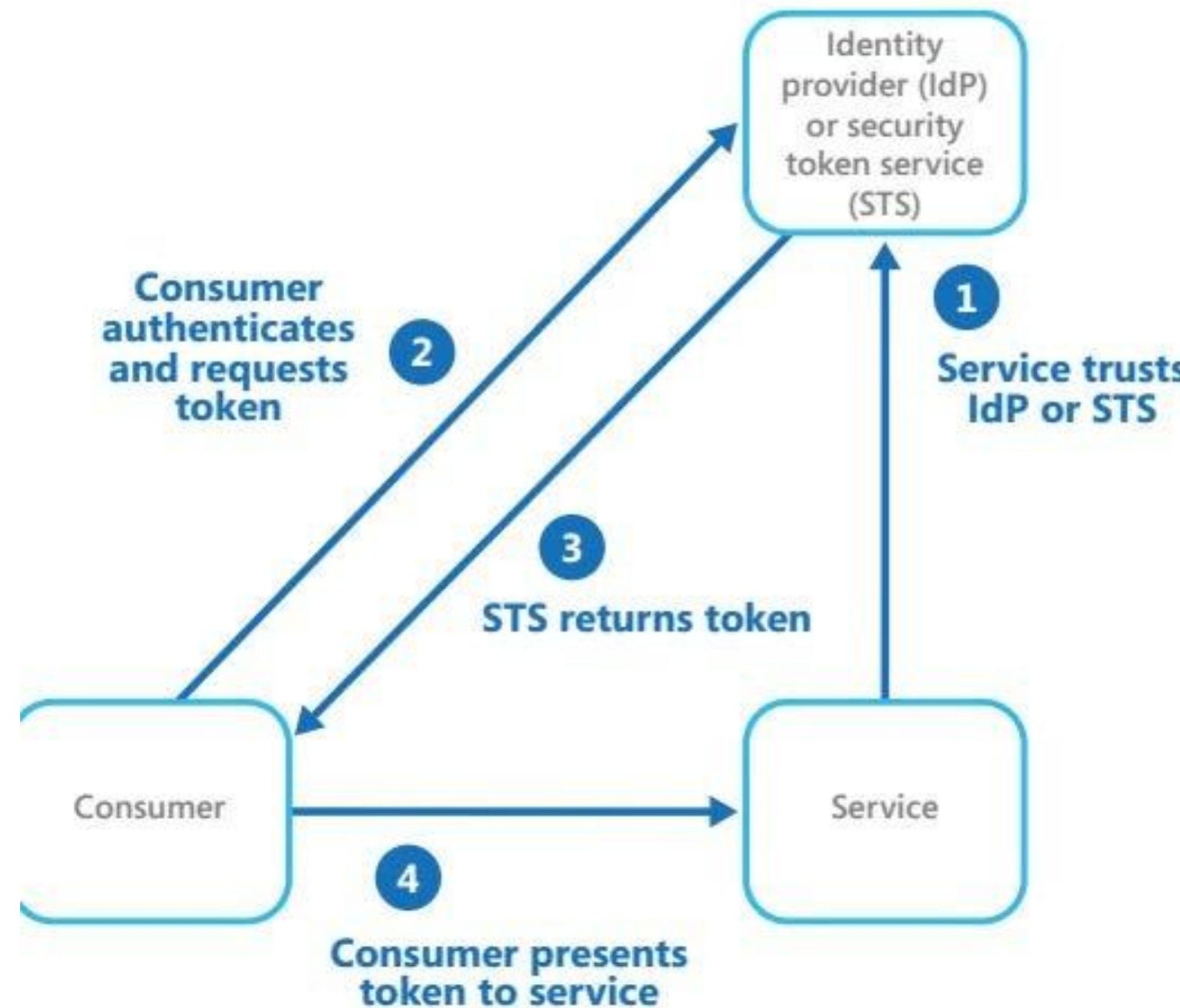
# External Configuration Store Pattern

---



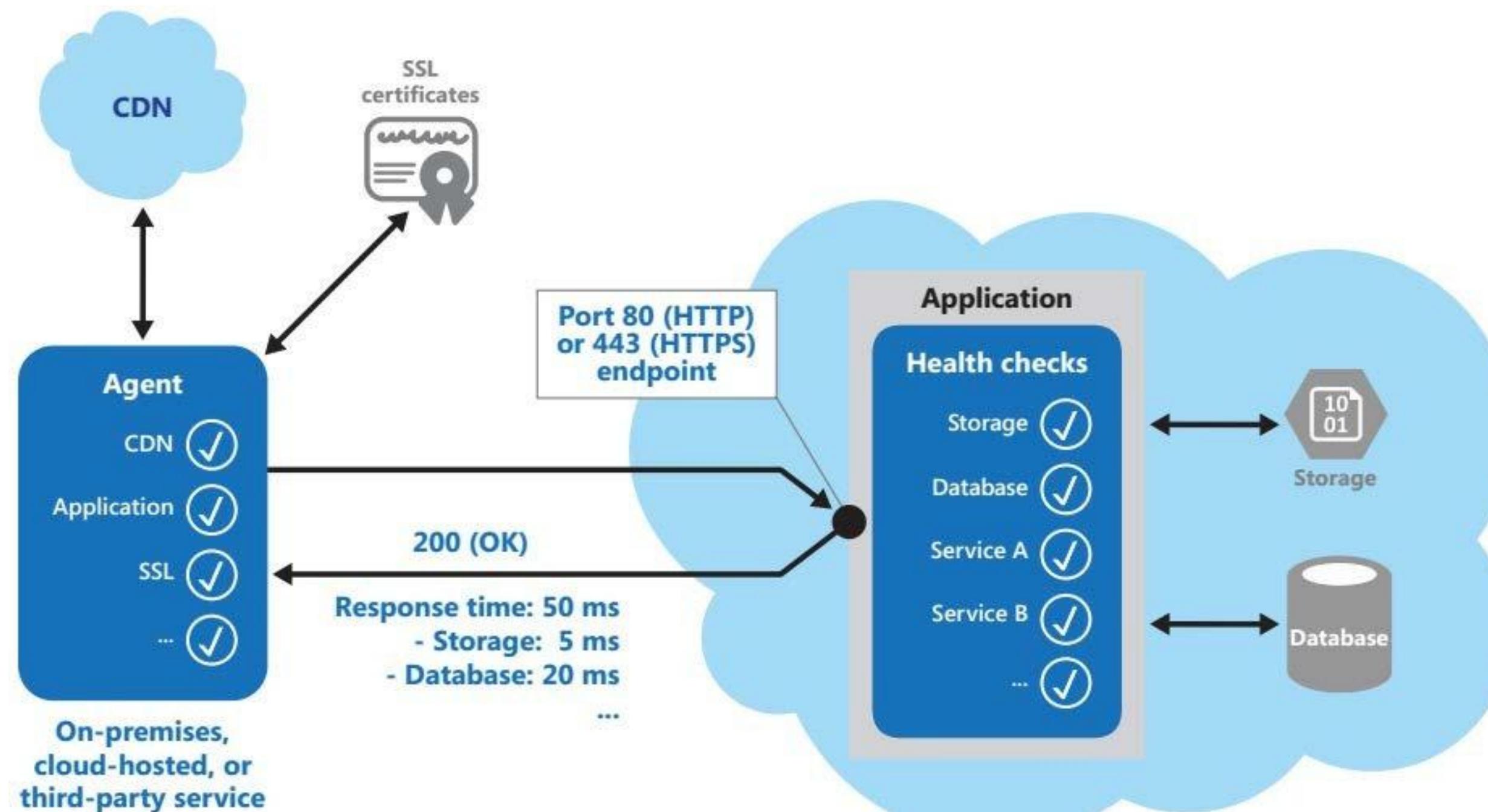
- Move configuration information out of application deployment package to a central location

# Federated Identity Pattern



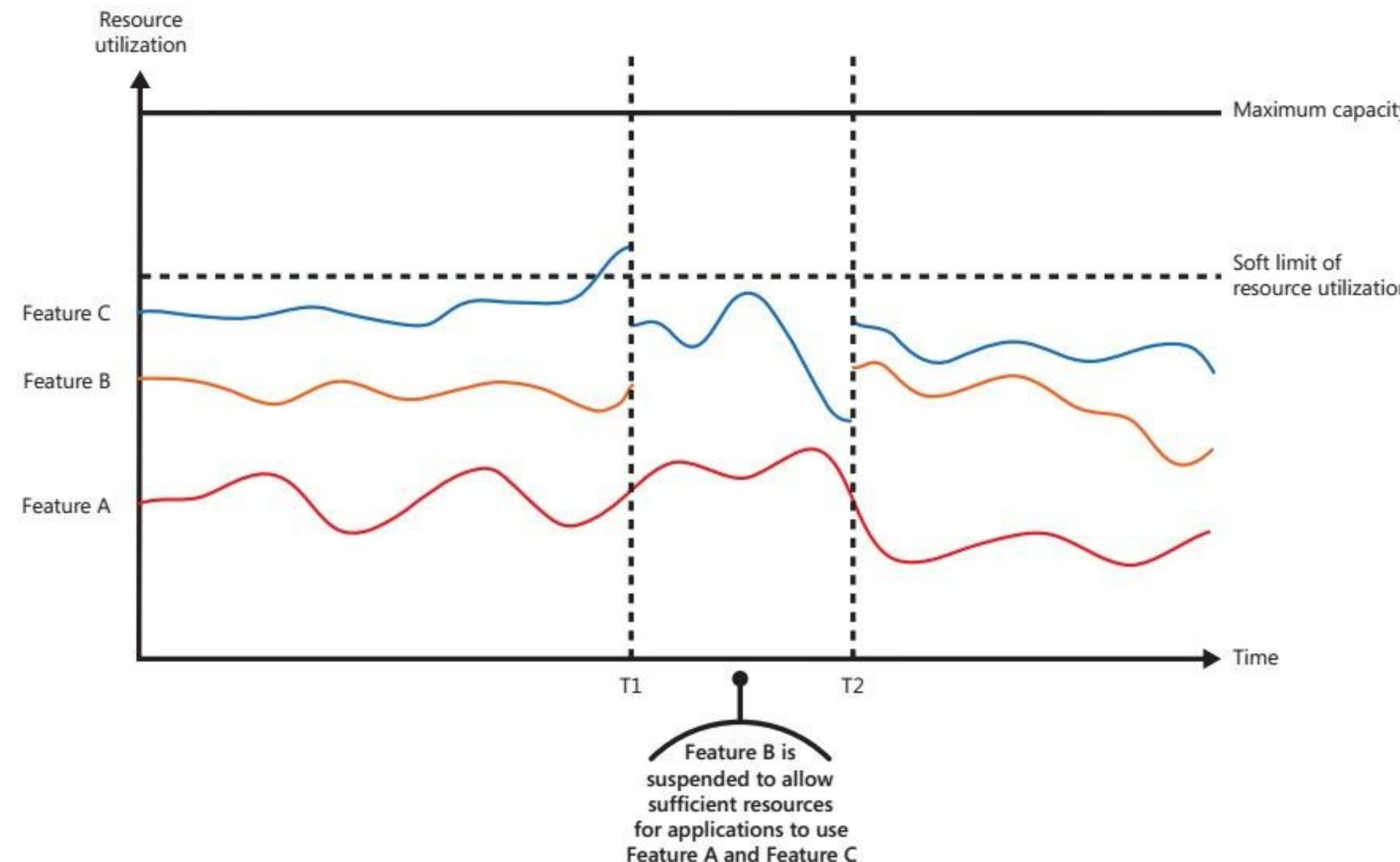
- Delegate authentication to an external identity provider

# Health Endpoint Monitoring Pattern



- 💡 Functional checks within an application that external tools can access through exposed endpoints at regular intervals

# Throttling Pattern



- Control consumption of resources used by an instance
- Allow system to continue to function & meet SLA even when an increase in demand places an extreme load on resources

# Introduction Machine Learn

By Jeevaka Perera



# Lesson Objectives

- At the end of the lesson students should be able to explain
  - What is Machine Learning?
  - Why Choose Machine Learning?
  - Different Machine Learning Algorithms and their applications

# What is Optimization?

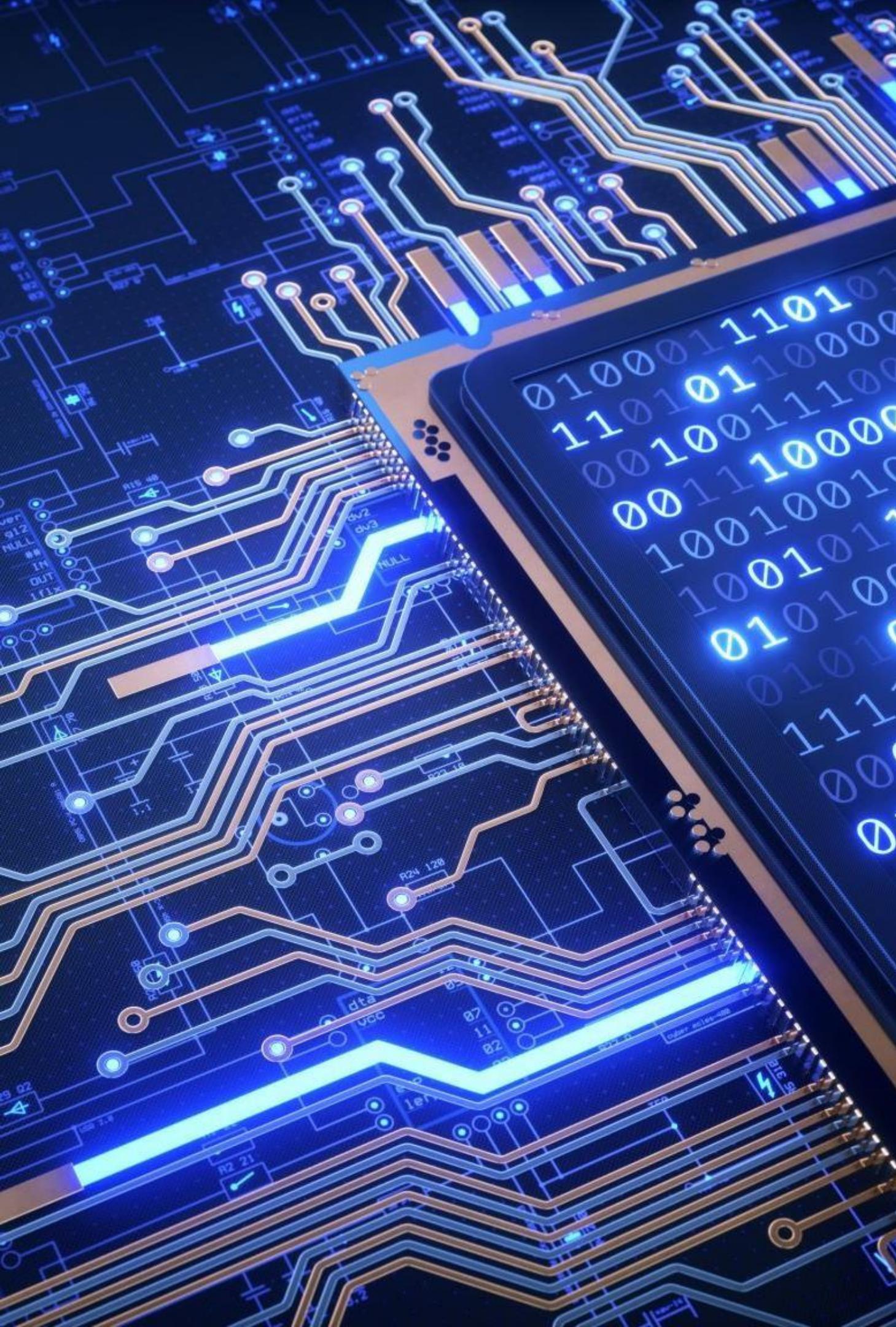
---

- **Optimization** is the mathematical discipline which is concerned with finding the maxima and minima of functions, possibly subject to constraints.



# What is Artificial Intelligence

- "It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable." by [John McCarthy](#)



# What is Machine Learning

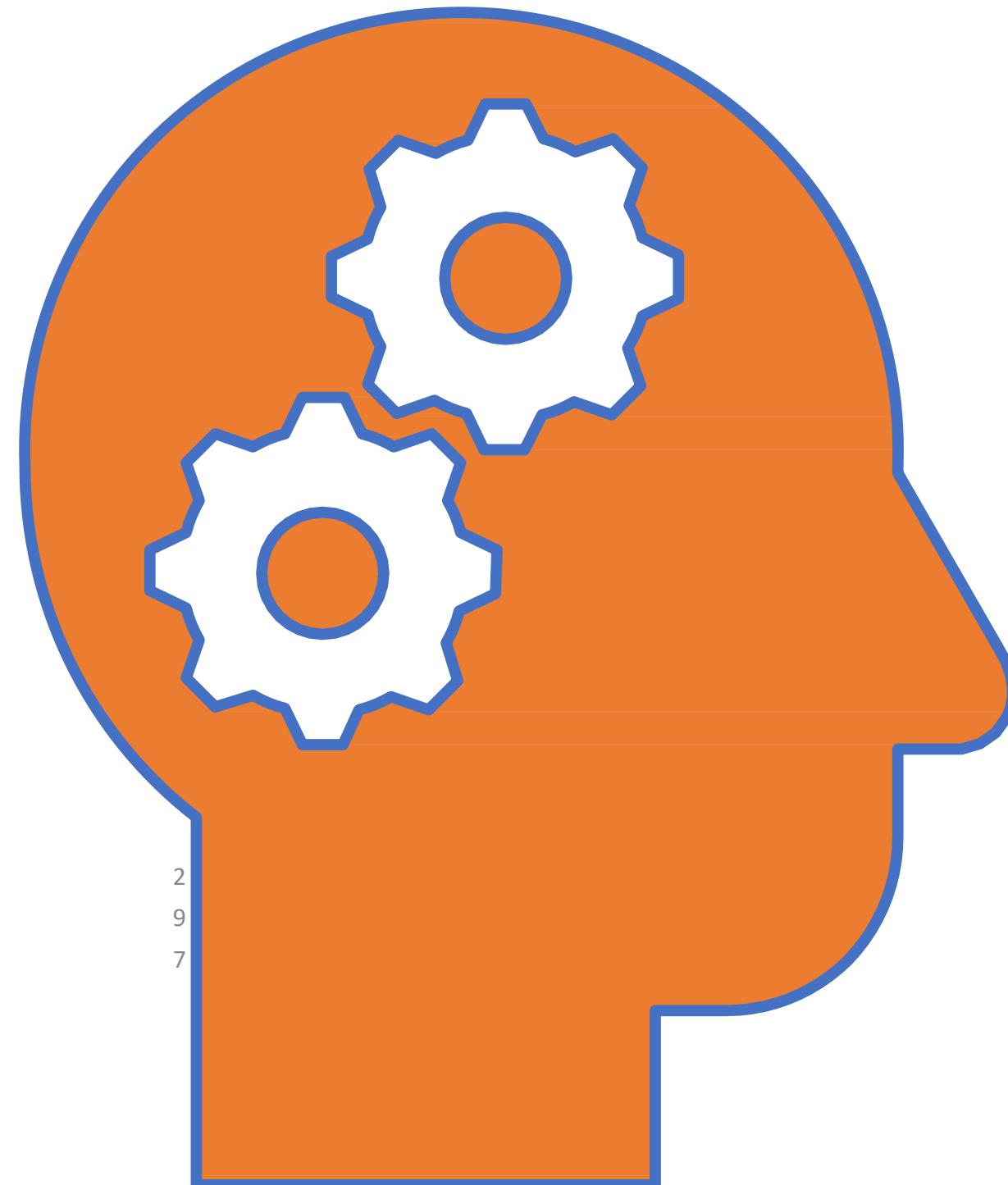
---

- “field of study that gives computers the ability to learn without explicitly being programmed.” by [Arthur Samuel](#)
- Machine learning is a subfield of artificial intelligence, which is broadly defined as the capability of a machine to imitate intelligent human behavior.

# Learning in a Machine

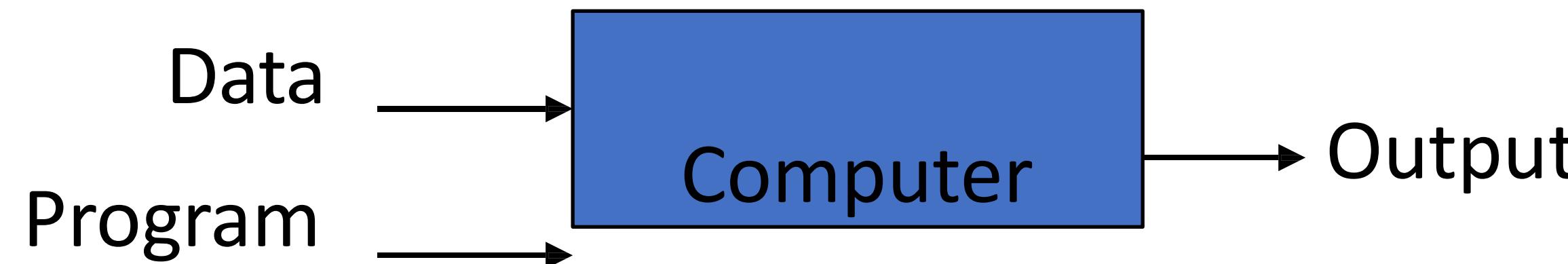
---

- “A computer program is said to learn from experience (E) with some class of tasks (T) and a performance measure (P) if its performance at tasks in T as measured by P improves with E”



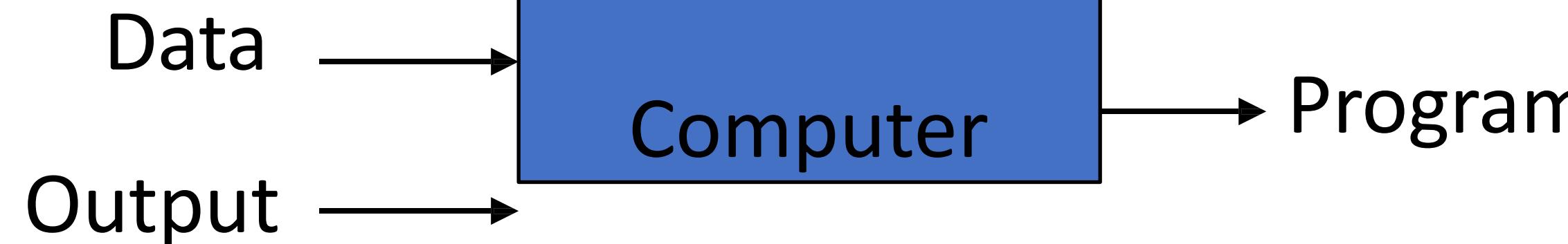
# ML vs Traditional programming

## Traditional Programming



## Machine Learning

Learning Algorithm



# Types of Artificial Intelligence Algorithms

---

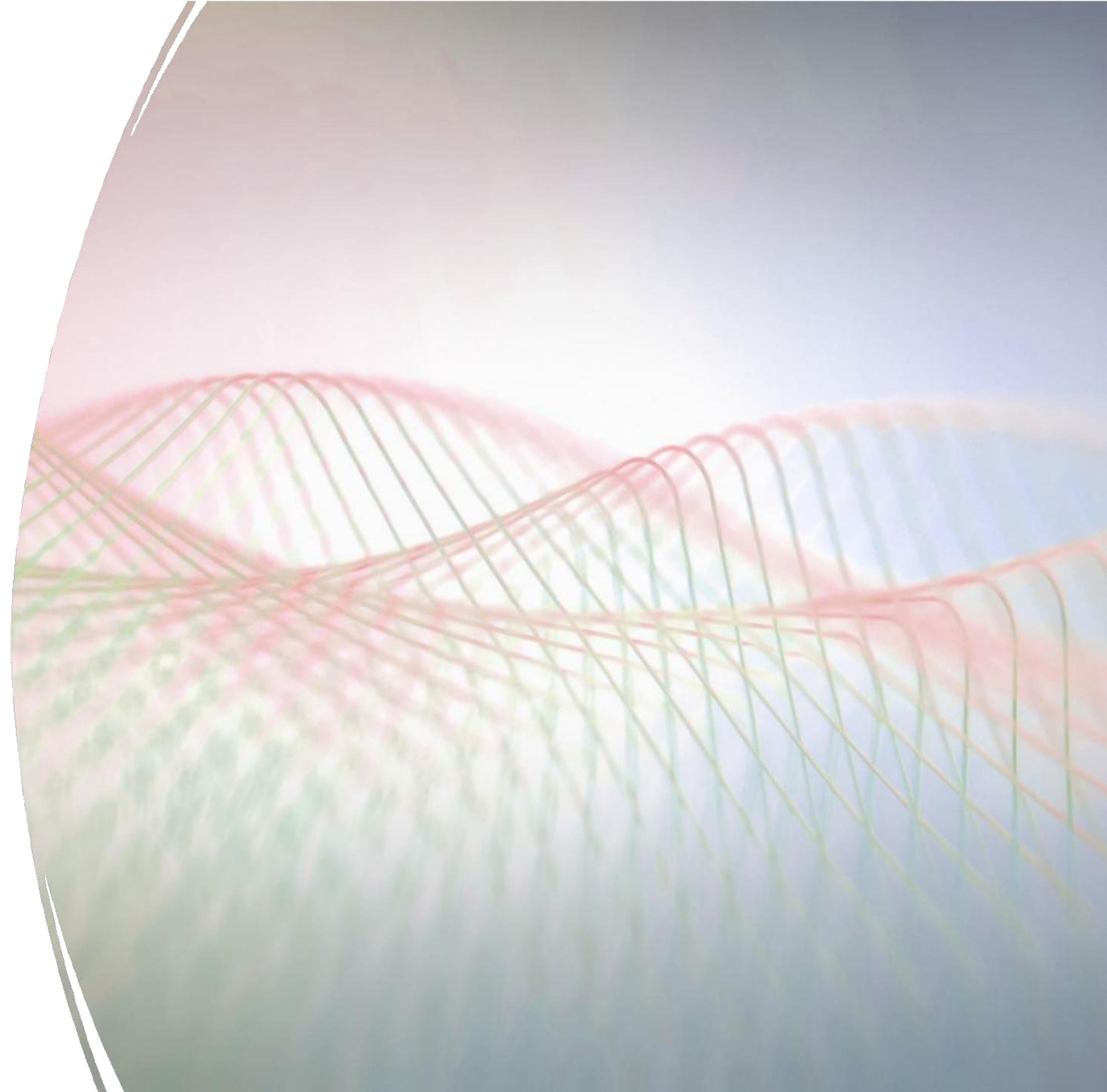
Rule-based expert

Systems

Search Algorithms

Evolutionary Algorithms and Swarm  
Intelligence Algorithms

Machine Learning Algorithms



# Rule Based Systems

- Expert Systems
  - PROSPECTOR
  - MYCIN
- Based on pre-defined Rules
- Rules defined based on domain knowledge
- Designed to mimic the decision-making process of human experts



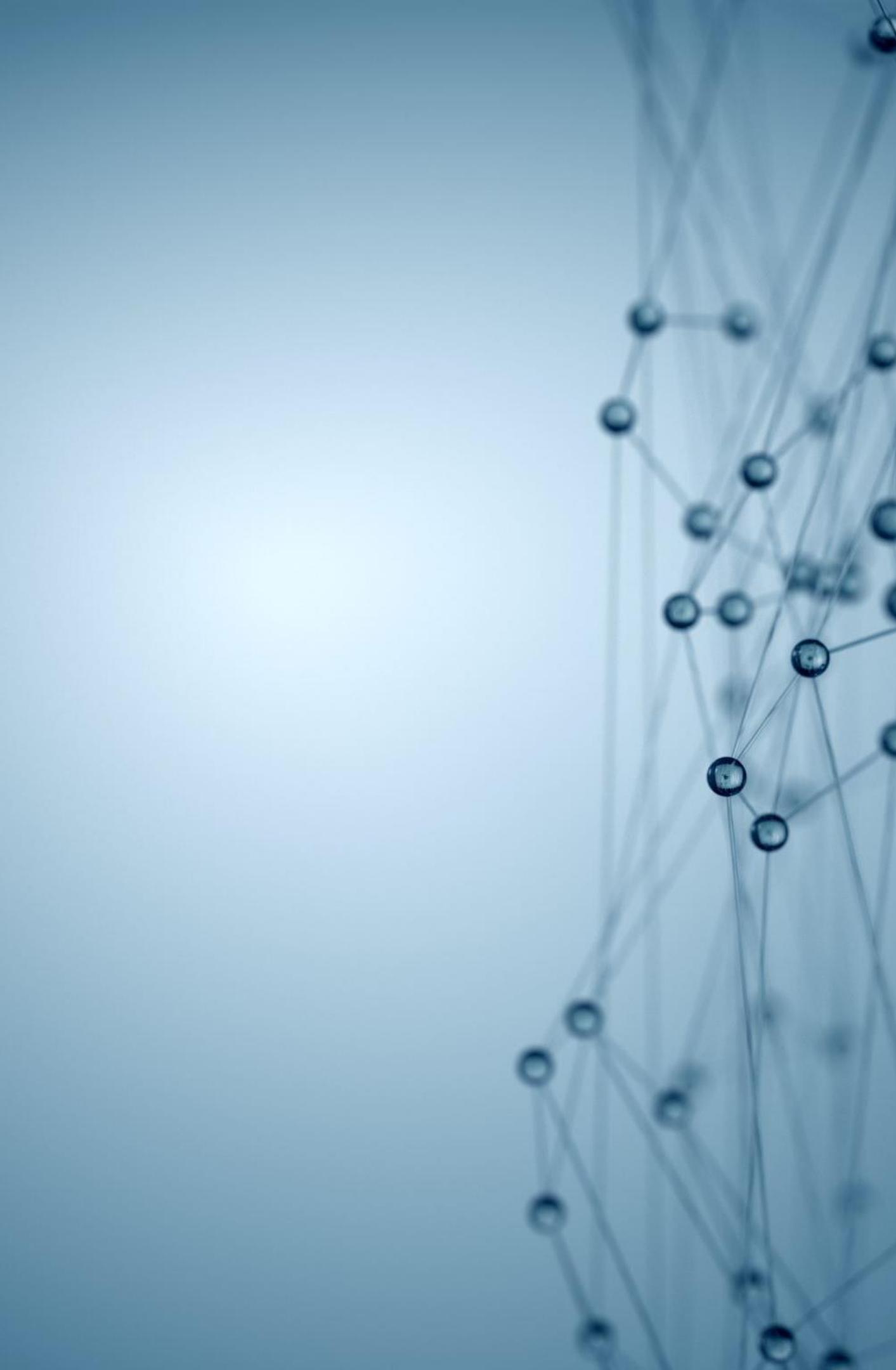
# Search Algorithms

---

- Breadth-First Search
- Depth First Search
- Iterative Deepening Search
- Uniform Cost Search
- Dijkstra's algorithm
- A\* Search

30

1



# Evolutionary Algorithms

---

- Evolutionary Algorithms are a family of nature-inspired optimization algorithms that mimic biological evolution—natural selection, mutation, recombination, and survival of the fittest.
  - Genetic Algorithms
  - Particle Swarm Optimization
  - Cultural Algorithms
  - HCA KCA
  - SI Algorithms(ACO, FA)





SCHEDULING AIRLINE  
CREWS



TUNING  
HYPERPARAMETERS IN ML



DESIGNING ANTENNAS  
(NASA EXAMPLE!)



PORTFOLIO  
OPTIMIZATION

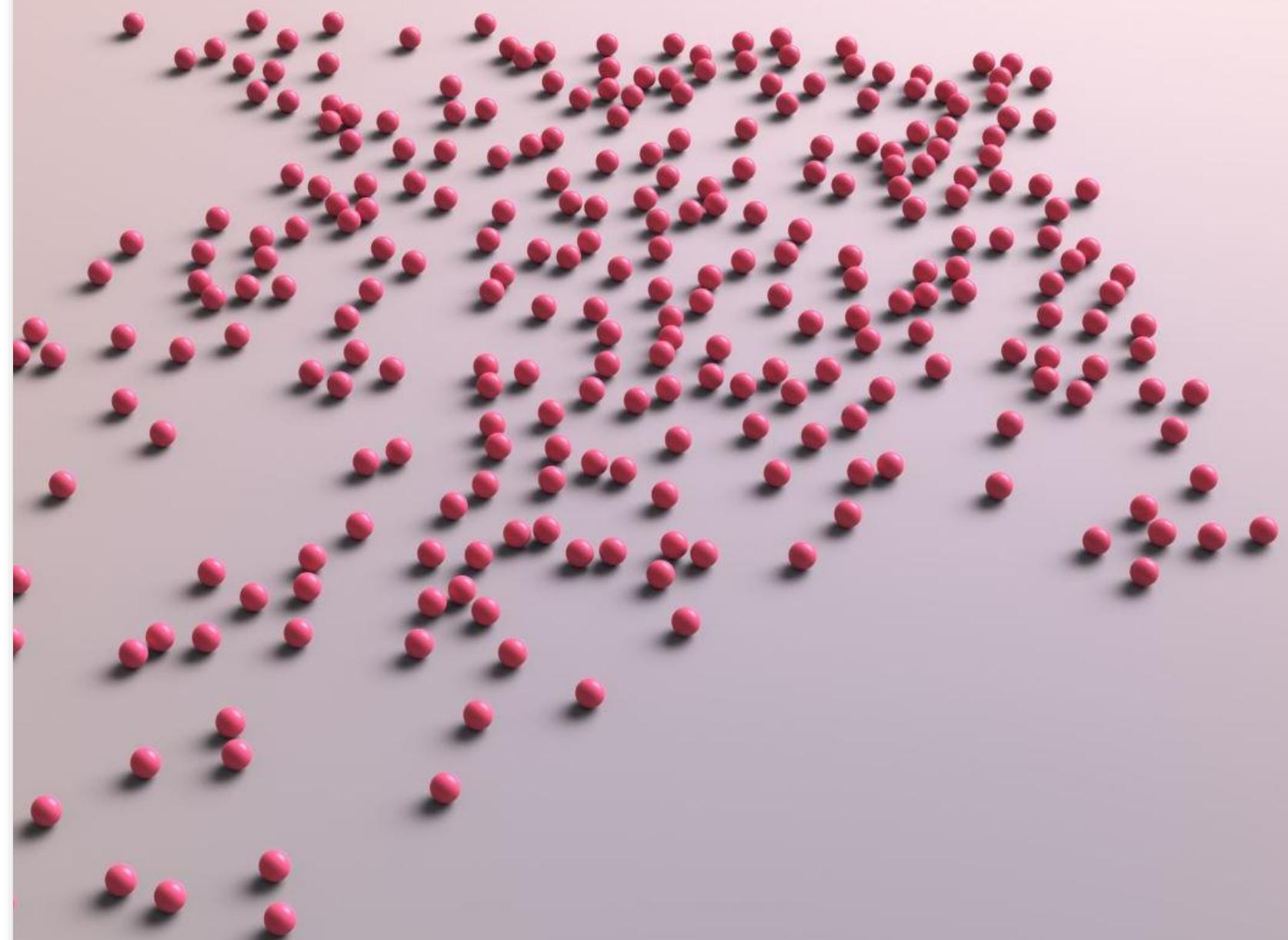
# Genetic Algorithms

- A Genetic Algorithm (GA) is a search and optimization method inspired by how living things evolve over time through natural selection.
- We represent potential solutions as chromosomes.
- Better solutions (those with higher fitness) are selected
- New solutions are created through crossover and mutation
- Over generations, solutions get better!

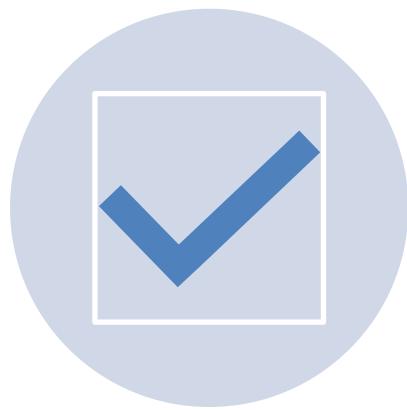


# Swarm Intelligence Algorithms

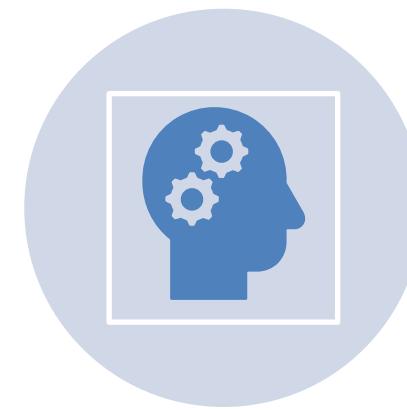
- **Inspired by:** Collective behavior of decentralized, self-organized systems (e.g., birds, ants, fish)
- **Ant Colony Optimization (ACO)**
  - Inspired by ants finding shortest paths using **pheromone trails**.
  - Good for **discrete path-based problems** like the **Traveling Salesman Problem (TSP)**.
- **Firefly Algorithm (FA)**
  - Fireflies are attracted to brighter (better) solutions.
  - Uses light intensity and distance for movement.



# Types of Machine Learning Algorithms



SUPERVISED LEARNING



UNSUPERVISED  
LEARNING



SEMI-SUPERVISED  
LEARNING



REINFORCEMENT  
LEARNING

# Supervised Learning Algorithms

- Learned under supervision.
  - Supervision of what?
  - Humans?
- Supervised by the Labeled data
  - Require labeled data. (Inputs, output)
  - This is the most difficult part of supervised learning.

# Types of Supervised Learning Models

- Regression
  - Predicting a Linear value
  - Linear Regression
  - SVR
  - DT
- Classification
  - Predicting a class/label
  - Logistic Regression
  - SVM
  - DT
  - NB

Artificial Neural Network Models such as MLP, CNN, RNNs are Considered as supervised Learning Models

# Supervised learning examples



A Bank may have borrower details (age, income, gender, etc.) of the past (**features**)



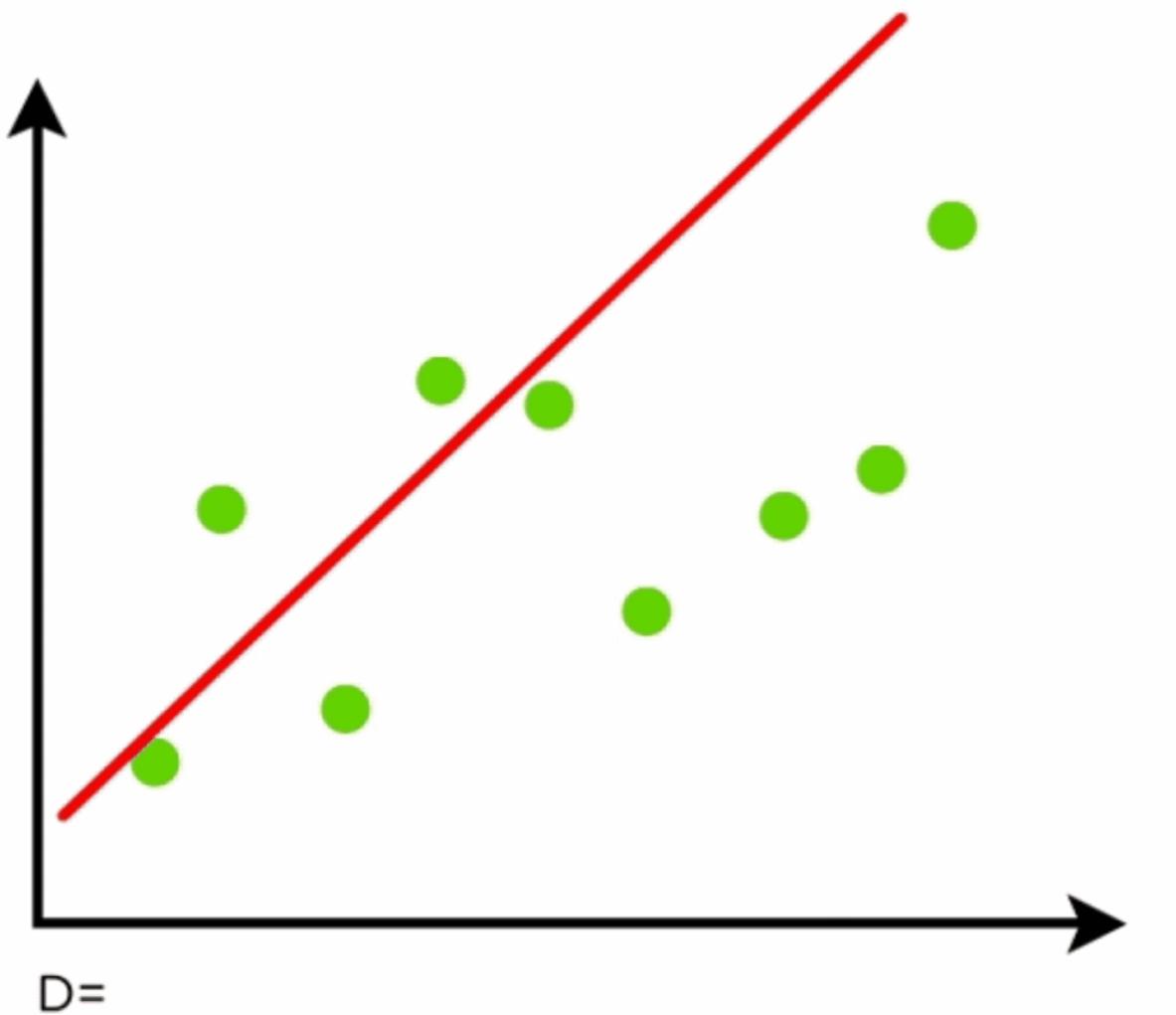
Also it may have details of the borrowers who defaulted in the past (**labels**)



Based on the above, can train a classifier to learn the patterns of borrowers who are likely to default on their payments

- **Model Explanation:**  
Draws a straight line that best fits the data.
- **Key Concept:**  
Learns the relationship as a **linear equation**:
  - $y = mx + c$ .
- **What is learnt through training:**  
Finds the best slope ( $m$ ) and intercept ( $c$ ) to minimize error.
- **Example Use Cases:**
  - Predicting house prices
  - Salary estimation
  - Sales forecasting
- **Limitations:**
  - Only works well when the relationship is **linear**
  - Not suitable for complex, non-linear patterns
  - Sensitive to **outliers**

# Linear regression



**“Predictor”:**

Evaluate line:

$$r = \theta_0 + \theta_1 x_1$$

return r



# Types of Unsupervised Learning Algorithms

---

- Clustering Algorithms
  - K Means
  - DBSCAN
- Dimensionality Reduction Algorithms
  - PCA
  - MDS (Multidimensional Scaling)<sup>31</sup>
  - LDA (Linear Discriminant Analysis)<sup>3</sup>
- Graph Based Models can be considered as Unsupervised Learning

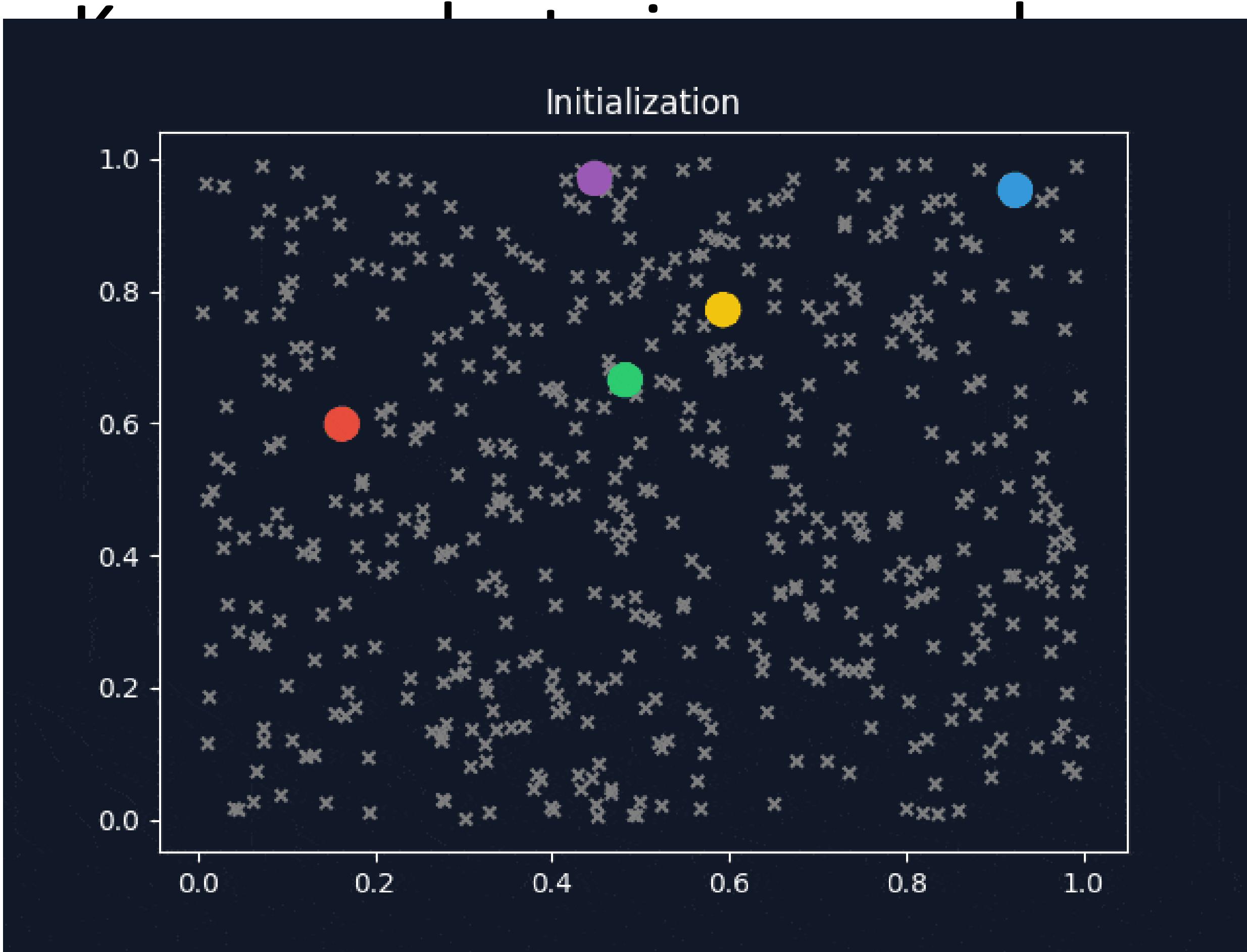
A Supermarket may store each buyer's

basket content details (**features**)

There are **NO** grouping (**labels**)

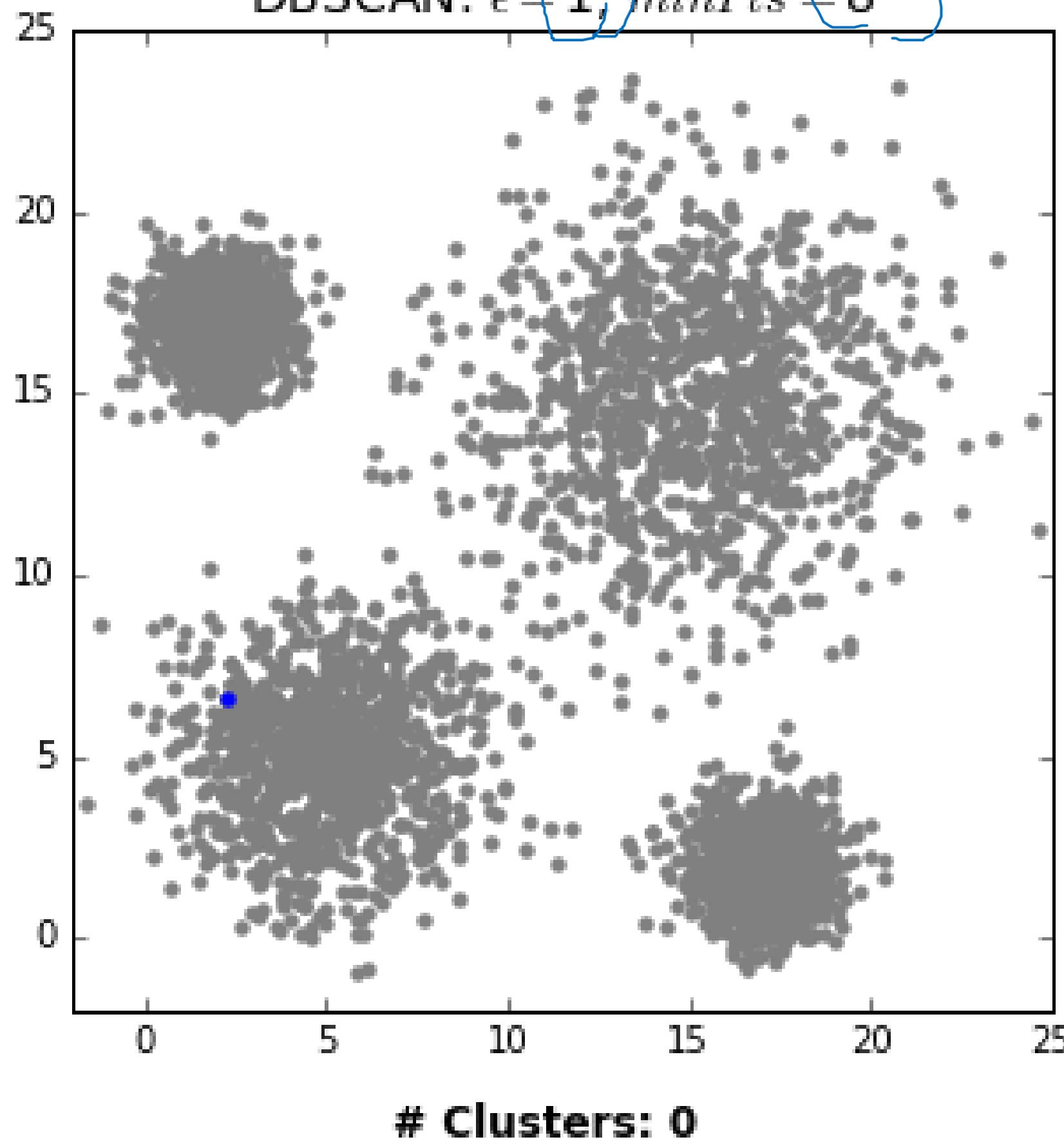
Need to group the buyers based on their buying patterns  
in order to best use the shelf space (recommendation)

- **Model Explanation:**
  - Groups data into **K clusters** based on similarity.
- **Key Concept:**
  - Finds **cluster centers** and assigns each point to the nearest one.
- **What is learnt through training:**
  - Learns the **position of cluster centers**.
- **Example Use Cases:**
  - Customer segmentation
  - Grouping articles by topic
  - Organizing images
- **Limitations:**
  - You must choose **K (number of clusters)** beforehand
  - Assumes **spherical-shaped clusters**
  - Struggles with **uneven or noisy data**

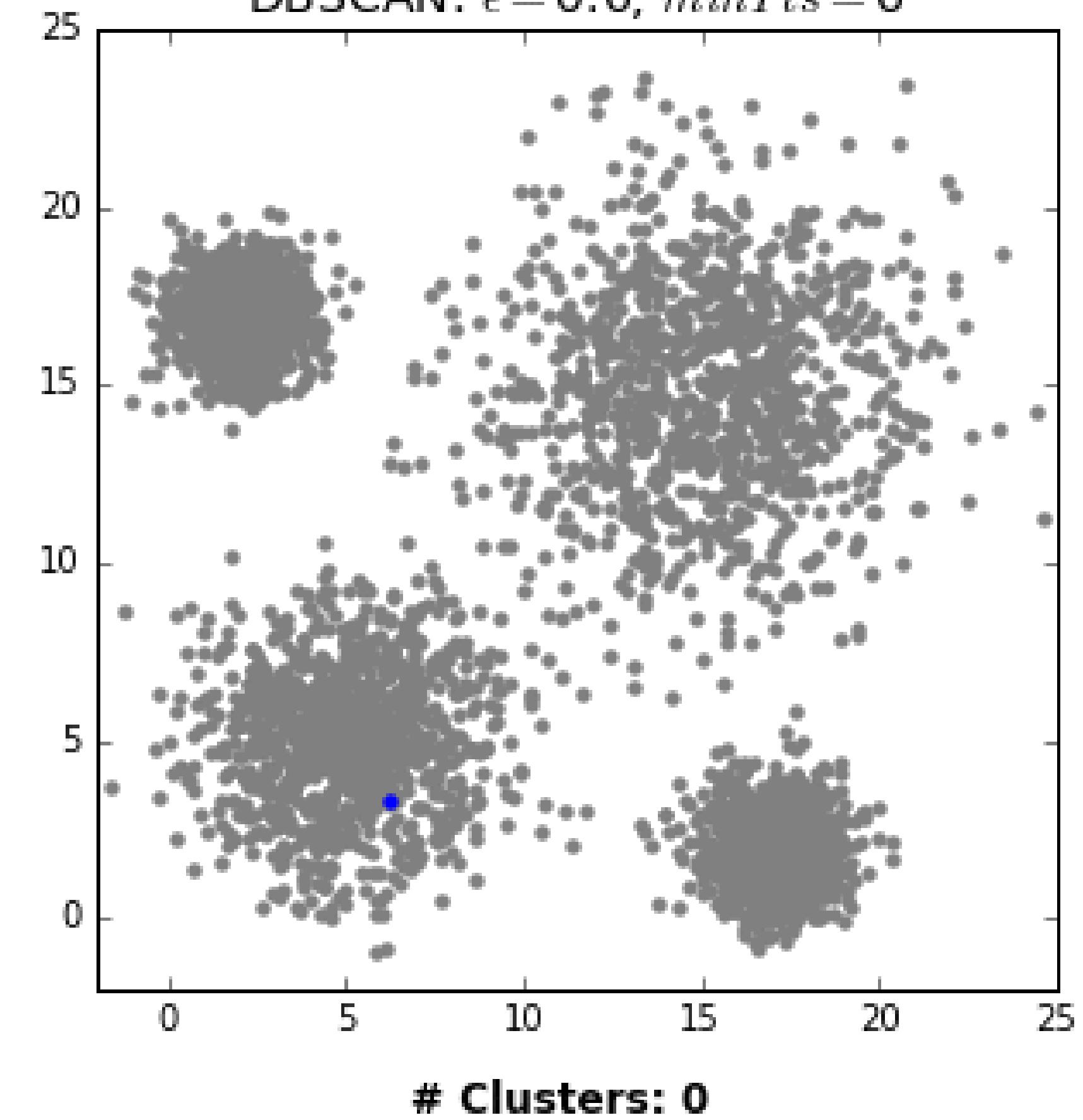


- **Model Explanation:**
  - Groups together dense areas; labels sparse points as outliers.
- **Key Concept:**
  - Clusters are formed based on **density**, not shape or number.
- **What is learnt through training:**
  - Learns which points belong to **dense clusters** and which are **noise**.
- **Example Use Cases:**
  - Fraud detection
  - Identifying event hotspots
  - Anomaly detection in GPS or sensor data
- **Limitations:**
  - Struggles with **varying densities**
  - Requires tuning of **eps** and **min points**
  - Not ideal for **high-dimensional data**

DBSCAN:  $\epsilon = 1$ ;  $minPts = 8$

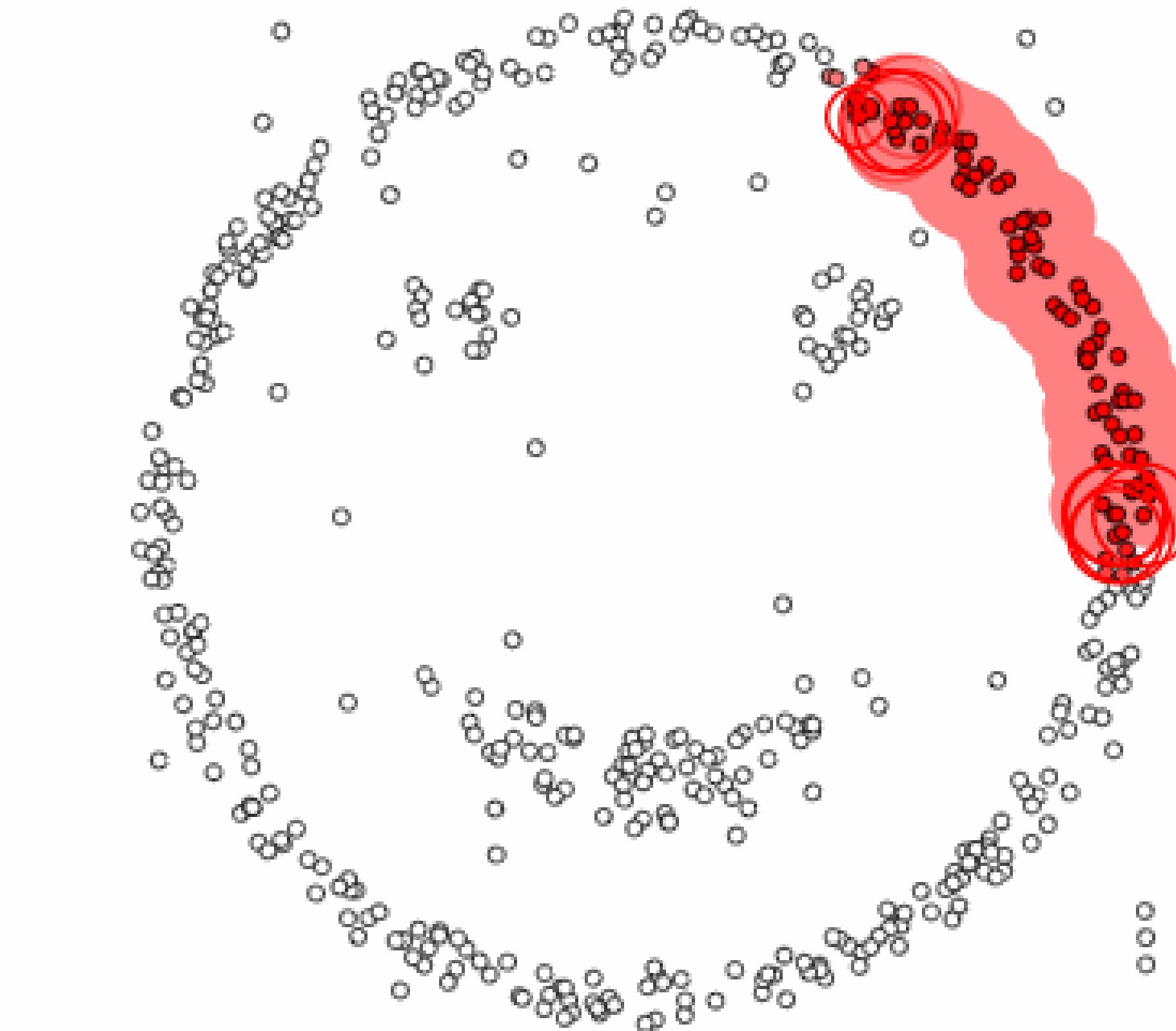


DBSCAN:  $\epsilon = 0.6$ ;  $minPts = 6$



$\text{epsilon} = 1.00$   
 $\text{minPoints} = 4$

**Restart**



**Pause**

Labeled data is expensive/difficult to get

Unlabeled data is cheap/easier to get

The idea is to use smaller amount of labelled data with larger amount of unlabeled data to creating the training/testing datasets

Algorithms - Self Training, Generative models

- Semi-Supervised Support Vector Machines, etc.



# Semi-Supervised Learning Algorithms

---

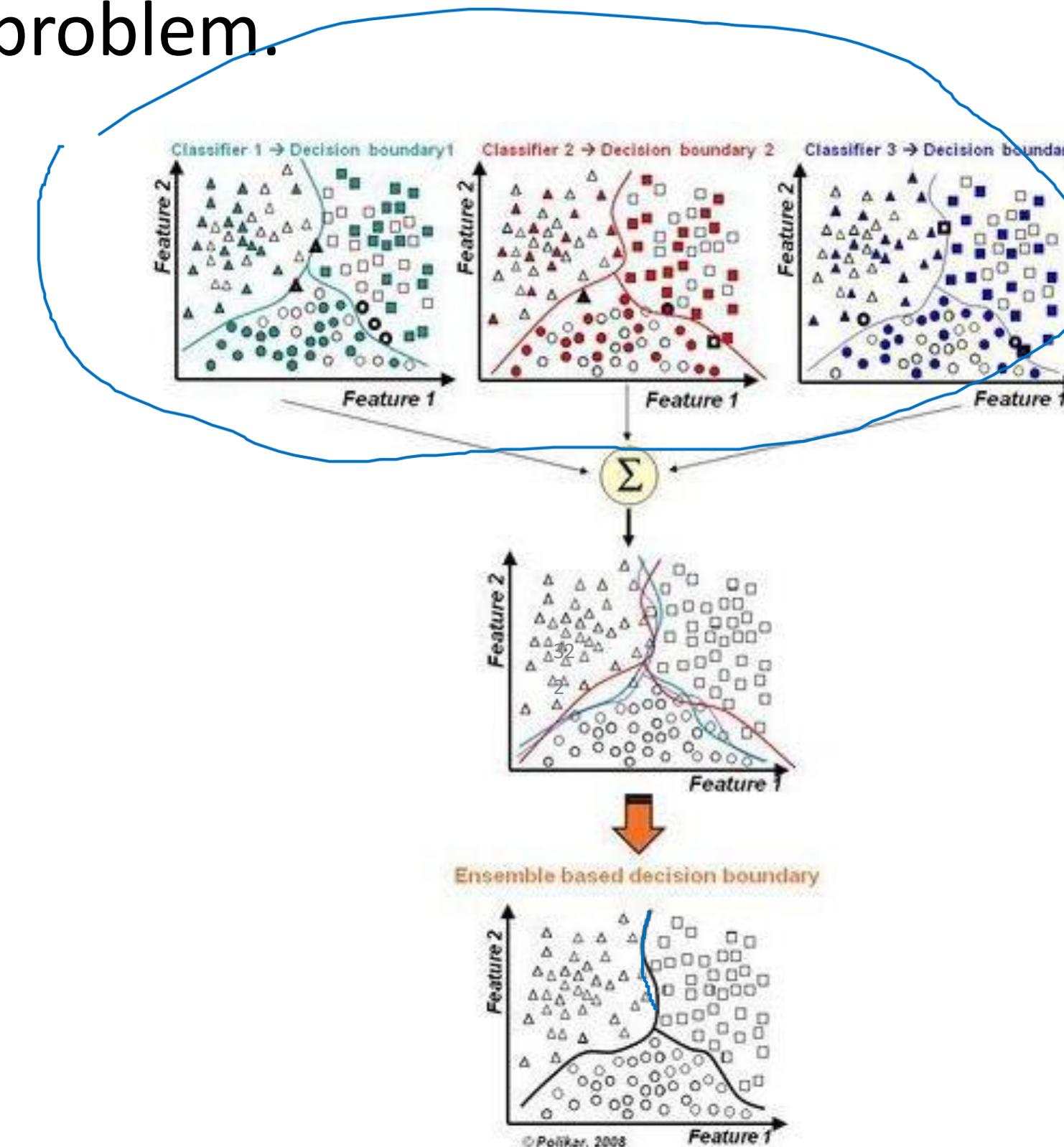
- Generative Adversarial Networks
- Auto-encoders
- Variational Auto-encoders

32

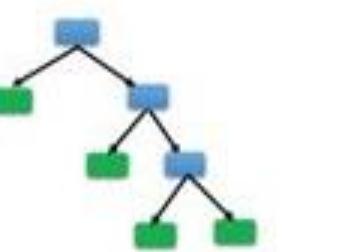
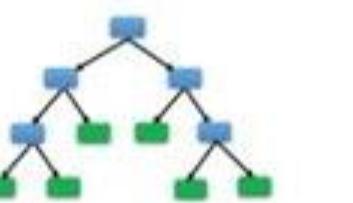
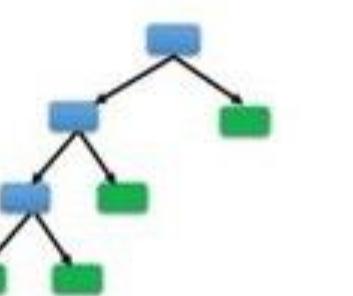
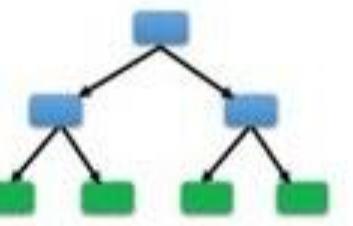
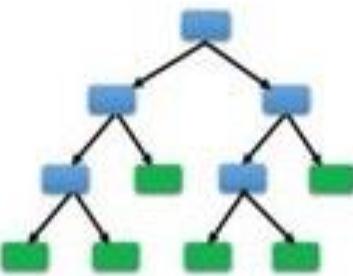
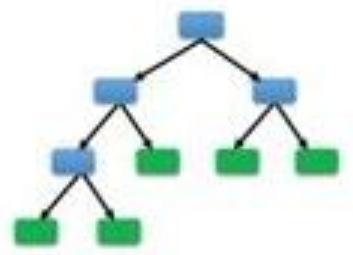
1

# Ensemble Learning

- Often, multiple classifiers need to be combined to solve a real-world problem.



- **Model Explanation:**
  - Combines predictions from many decision trees.
- **Key Concept:**
  - Uses **voting** or **averaging** to make decisions.
- **What is learnt through training:**
  - Learns different rules from subsets of data to make **robust predictions**.
- **Example Use Cases:**
  - Spam detection
  - Credit risk analysis
  - Disease classification
- **Limitations:**
  - Can be **slow** with large datasets
  - Less interpretable than a single decision tree
  - Needs tuning (like number of trees)



Random Forest in Action!!!

- **Model Explanation:**
  - Learns by interacting with an environment and receiving feedback (rewards or penalties).
- **Key Concept:**
  - Uses **trial-and-error** and **reward maximization** to improve decision-making over time.
- **What is learnt through training:**
  - Learns an **optimal policy** or **action strategy** that maximizes long-term rewards.
- **Example Use Cases:**
  - Game playing (e.g., AlphaGo)
  - Robotics (e.g., navigation or motor control)
  - Dynamic pricing or recommendation systems
- **Limitations:**
  - Requires **many interactions** with the environment (sample inefficiency)
  - Can be **unstable** or hard to converge
  - Needs **careful reward design** to avoid unintended behaviors

# Reinforcement learning examples

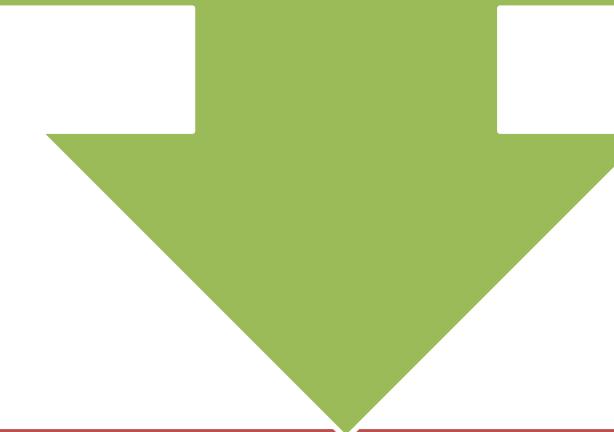
A group of robots have been deployed in an unknown territory

The objective is for them to collaboratively find the navigation path to reach a particular destination/goal

Can use reinforcement learning where achieving the goal/getting closer to the goal gives a positive reward. Negative reward otherwise

Can share the information among robots (multi-agent system)

Not all models are created equal – and neither are the ways we evaluate them.



## Key Questions to Ask:

Is it a  
**classification,**  
**regression**, or  
**clustering** task?

Do we care more  
about **correctness**,  
**fairness**, or  
**interpretability**?

What are the **costs**  
**of wrong**  
**predictions**?

# Common Evaluation Matrices

Task Type	Metrics Used
Classification	Accuracy, Precision, Recall, F1 Score, ROC-AUC
Regression	MSE, MAE, RMSE, R <sup>2</sup> Score
Clustering	Silhouette Score, Davies-Bouldin Index, Inertia
Ranking/Recommendation	MAP, NDCG, Hit Rate

# Classification Matrices Explained

Metric	Use When...	Notes
Accuracy	Classes are balanced, all errors matter	Can be misleading with imbalance
Precision	False positives are costly (e.g., spam)	$TP / (TP + FP)$
Recall	False negatives are costly (e.g., cancer)	$TP / (TP + FN)$
F1 Score	Balance between precision and recall	Harmonic mean
ROC-AUC	Need to evaluate ranking ability	Works for probabilistic models

# Regression Model

Metric	Use When...	Notes
MSE	Large errors are very bad	Penalizes large errors more
MAE	Equal penalty for all errors	More robust to outliers
RMSE	Like MSE but in original units	Square root of MSE
R <sup>2</sup> Score	Want to explain variability in output	1 = perfect, 0 = no

# Clustering Algorithms

Metric	Use When...	Notes
Silhouette Score	Want to measure how distinct clusters are	1 = well-clustered, -1 = wrong
Davies-Bouldin Index	Lower is better (compact & separated)	Good for comparing k-values
Inertia (within-cluster SSE)	Used in K-Means	Lower is better, but not scaled

# Practical Evaluation Factors

Factor	Why It Matters
Interpretability	Do we understand how/why it makes predictions?
Training Time	Important for real-time or big data
Fairness	Does it treat all groups equally?
Generalization	Does it perform well on new data?
Explainability	Can we explain decisions to stakeholders?

# Things to consider in Selecting a ML Algorithm

- If there's an algorithmic way instead of ML, use it!!! (ML is messy)
- Refer the literature!!!
- Try different ML algorithms (no single algorithm is the best)
- Check the dataset against the usage/strength of each algorithm (e.g. RNNs, ARIMA is good in time-series predictions)
- Be mindful of 'external factors' (e.g. seasonal effects, RL if you don't have data, Clustering if you have unlabeled data, etc.)  
33  
3
- Test your algorithm(s) with test data and select the best performing one for production (include the test results in your thesis/publications)
- No algorithm will be perfect! (There will be an error. The objective is to keep the error at an acceptable rate)

# Popular Frameworks/Tools

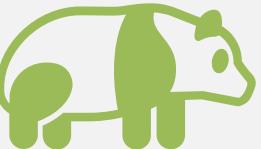
- Scikit-learn - Python (Anaconda Python Distribution)
- R (R studio)
- Matlab/Octave (can export DLLs)
- Weka (Java based)
- Java OpenNLP/Python NLTK (Natural language processing + ML)
- Apache Spark (part of the Apache Hadoop platform)
- Google Tensorflow (Python library for Deep neural networks)
- Apache Keras (Python library of neural networks)
- Theano (Python library for Multicore processing of DNNs)
- Amazon AWS Services/Microsoft Azure ML (Cloud based ML)

# Commonly used python libraries



NumPy

Matrix algebra



Pandas

Data Frames, Series



Matplotlib

Visualization

# Summary

---

- AI is a vast discipline with many varying branches.
- AI attempts to give machine the ability to mimic human decision making/learning capabilities



# Thank You

Jeewaka Perera

[Jeewaka.p@sliit.lk](mailto:Jeewaka.p@sliit.lk)

Tuesday, February 2, 20XX

