Cloud Computing Fundamentals
IN401 – M1S1 – 5 Credits

Course Grading

- Partial Exam
- Final Exam
- Second Session

Course Contents

- Chapter 1: Introduction to Cloud Computing
- Chapter 2: Infrastructure as a Service (IaaS)
- Chapter 3: Platform as a Service (PaaS)
- Chapter 4: Software as a Service (SaaS)
- **Chapter 5: Cloud Native Technologies & Architectures**
- Chapter 6: Cloud Security
- Chapter 7: Practical Case Studies

Chapter 5:

Cloud Native Technologies & Architectures

Content

- Introduction
- Microservices, API gateways
- Container orchestration overview (Kubernetes introductory)
- Serverless computing basics (Functions as a Service: AWS Lambda, Azure Functions)

Introduction

- What is Cloud Native?
- Problems with traditional approach (How Cloud Native address these issues).
- Architecture

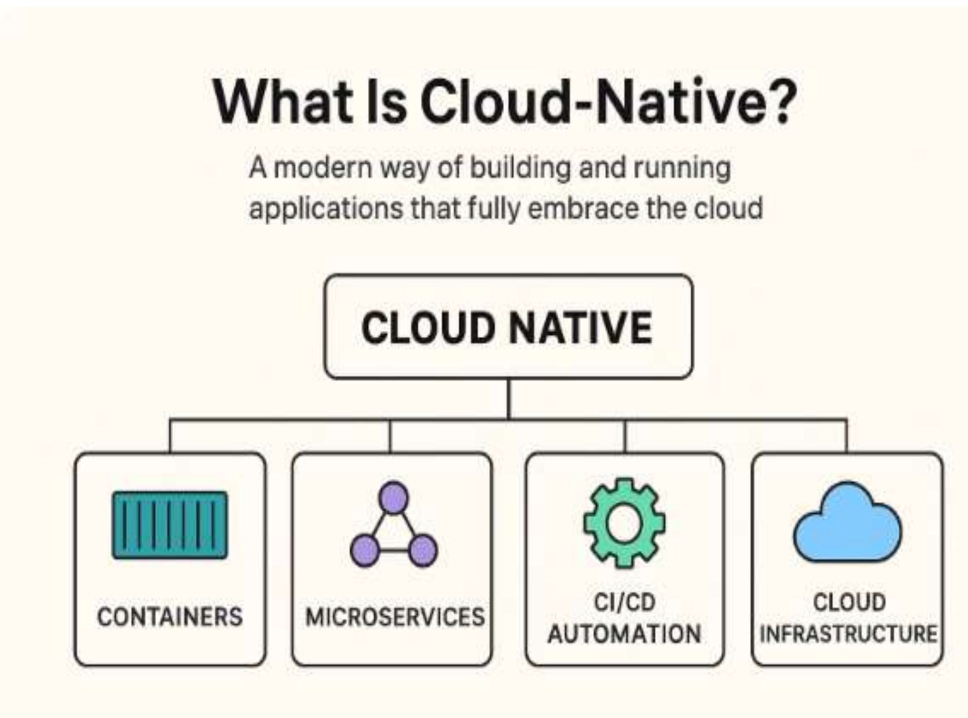
Cloud Native Definition

- **Cloud-native** refers to an approach for building and running applications that fully leverage the benefits of cloud computing.

- Instead of simply hosting traditional applications on cloud servers, cloud-native applications are **designed specifically for the cloud environment** from the start.

Cloud Native Definition

- **Cloud-native** is an architecture and development methodology that focuses on building scalable, resilient, and flexible applications using cloud technologies.



Cloud Native – Real examples

- Netflix
- Facebook / Meta
- Airbnb
- Uber
- UberEats
- Tesla
- Instagram
- ….

Problems with Traditional approach

Monolithic architecture
one big codebase

Slow deployment & release cycles
weeks or months

Limited scalability
buy physical servers or VMs

Environment inconsistency
works on my machine

Poor resilience
a single crash may cause full downtime

High costs
CapEx

Cloud-Native
Microservices modular design

CI/CD automation
Autoscaling

Containers
(Docker/Kuberneetes)

Self-healing
Infrastructure as Code (IaC)

Pay-as-you-go
IaAC/ aotomated provisioning

Pay-as-you-go
Pay-as-you-go

1. Monolithic Architecture

- Traditional apps are usually *one big codebase*.
- Any small change requires redeploying the **entire application**.
- Hard to scale—must scale the whole system, not individual components.
- Failure in one part can crash the whole app.

☞ Cloud-native uses **microservices**, so changes and failures are isolated.

2. Slow Deployment & Release Cycles

- Releases happen every **weeks or months**.
- Manual deployments increase risk and effort.
- Hard to adopt automation.

☞ Cloud-native uses **CI/CD** and **DevOps practices** for fast, automated, safe deployments.

3. Limited Scalability

- Scaling requires buying physical servers or VMs.
- Scaling is **manual**, slow, and expensive.
- Resources remain unused when traffic decreases (over-provisioning).

☞ Cloud-native apps scale automatically (horizontal/vertical autoscaling).

4. Environment Inconsistency

- “Works on my machine” problems are common.
- Different OS, libraries, and configs across dev, test, and prod.

☞ Cloud-native uses **containers** (Docker) and **orchestration** (Kubernetes), ensuring consistency everywhere.

5. Poor Resilience

- Traditional systems are not built for failure.
- A single crash may cause full downtime.
- Limited ability to restart or self-heal.

☞ Cloud-native systems have **self-healing, redundancy, and fault tolerance** built-in.

6. Manual Infrastructure Management

- Requires manual server provisioning.
- Long setup times (days or weeks).
- High operational workload for the team.

☞ Cloud-native uses **Infrastructure as Code (IaC)** and automated provisioning.

7. High Costs

- Must buy servers upfront (CapEx).
- Pay even when traffic is low (always-on resources).
- Scaling requires hardware purchases.

☞ Cloud-native is **pay-as-you-go**, optimized for cost efficiency.

Cloud Native – Core Characteristics

- **1. Microservices Architecture**

Applications are broken into small, independent services that can be deployed, scaled, and updated separately.

- **2. Containers**

Microservices are packaged inside containers for portability and consistent environments.

- **3. Dynamic Orchestration**

Platforms like **Kubernetes** automatically manage scaling, deployment, healing, and load balancing.

Cloud Native – Core Characteristics

- **4. Elastic Scalability**

Applications automatically scale up/down based on demand.

- **5. Resilience**

If one microservice fails, the whole application does not crash; self-healing and redundancy are built-in.

- **6. DevOps + Automation**

Automated pipelines for testing, deployment, monitoring, and rolling updates.

- **7. Cloud Services**

Uses managed services (databases, messaging, storage, API gateways…).

Cloud Native Technologies

- Cloud-native technologies are the **tools and platforms** that help you build applications designed specifically to run **in the cloud**—not on traditional servers.

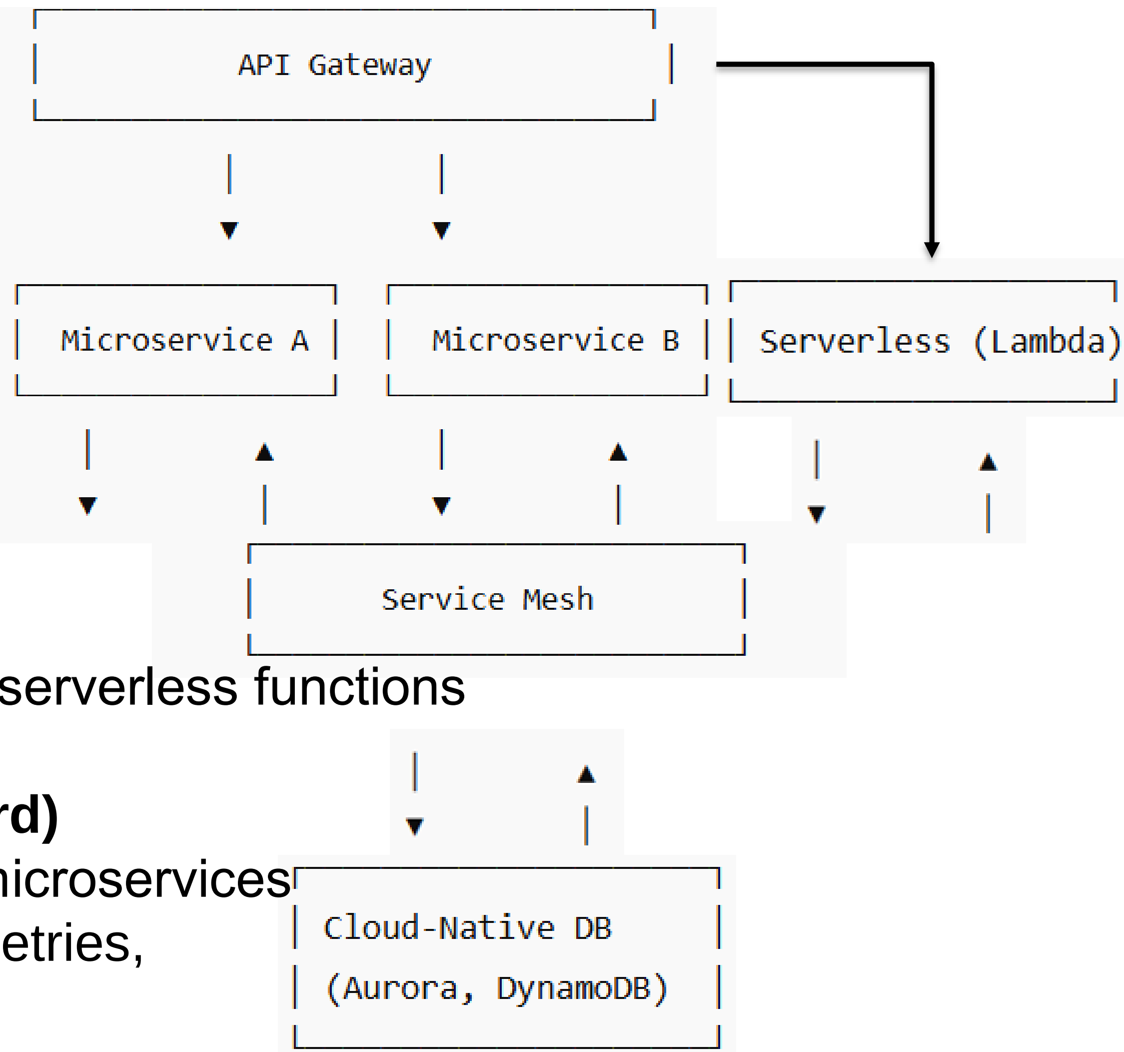
- These technologies are:

Layers:
1-Microservices
2-Serverless
3-API Gateway
4-Service Mesh
5-Cloud Native DB

Execution Environment:
6-Containers (Docker)
7-Orchestration (Kubernetes)

Platform & Automation:
8-DevOps practices
9-Infrastructure as Code (IaC)
10-Continuous Integration & Continuous Deployment (CI/CD)

Architectural Flow – Layers



Application Layer:

1-Microservices

→ main business logic

2-Serverless

→ used for event-driven or background tasks

Entry Point Layer:

3-API Gateway

→ receives client requests

→ routes to microservices or serverless functions

Communication Layer:

4-Service Mesh (Istio/Linkerd)

→ manages traffic between microservices

→ ensures security (mTLS), retries, load balancing

Data Layer:

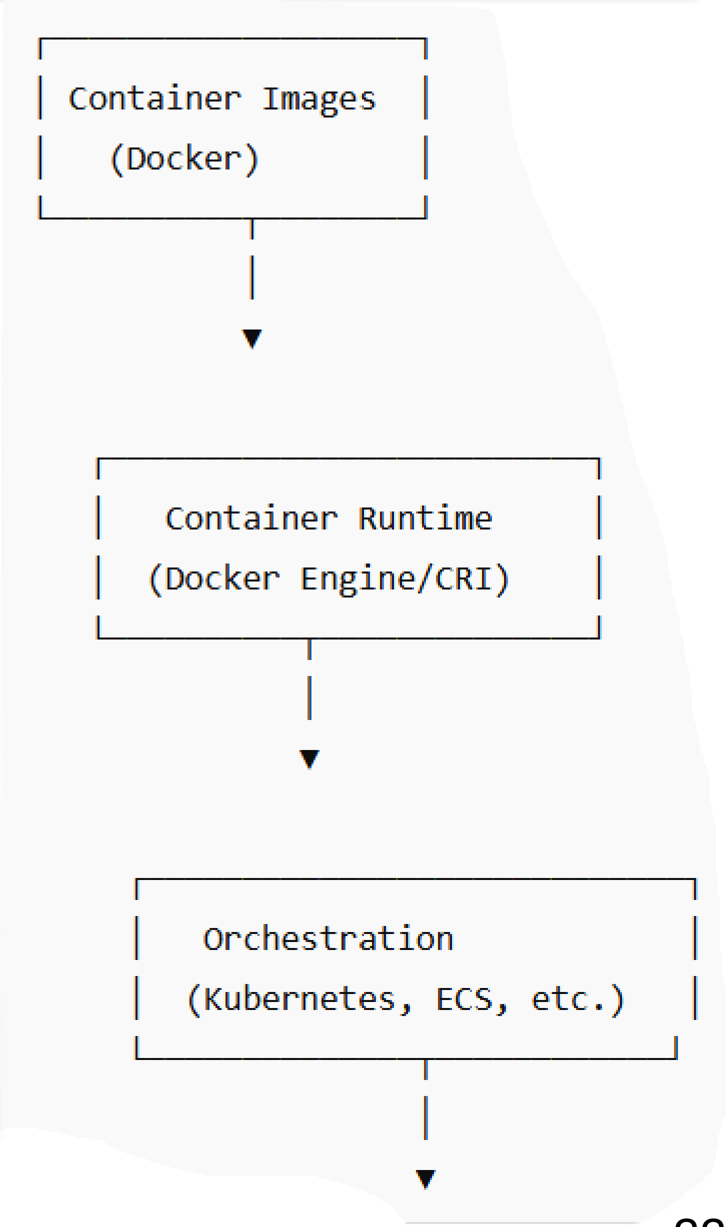
5-Cloud-Native Database

→ scalable, distributed, fully managed

Architectural Flow – Execution Environment

6-Containers (Docker)
→ Package microservices
Container Runtime
→ Runs containers on nodes

7-Orchestration (Kubernetes)
→ Deploys, scales, heals containers



Architectural Flow – Platform & Automation

8-DevOps practices

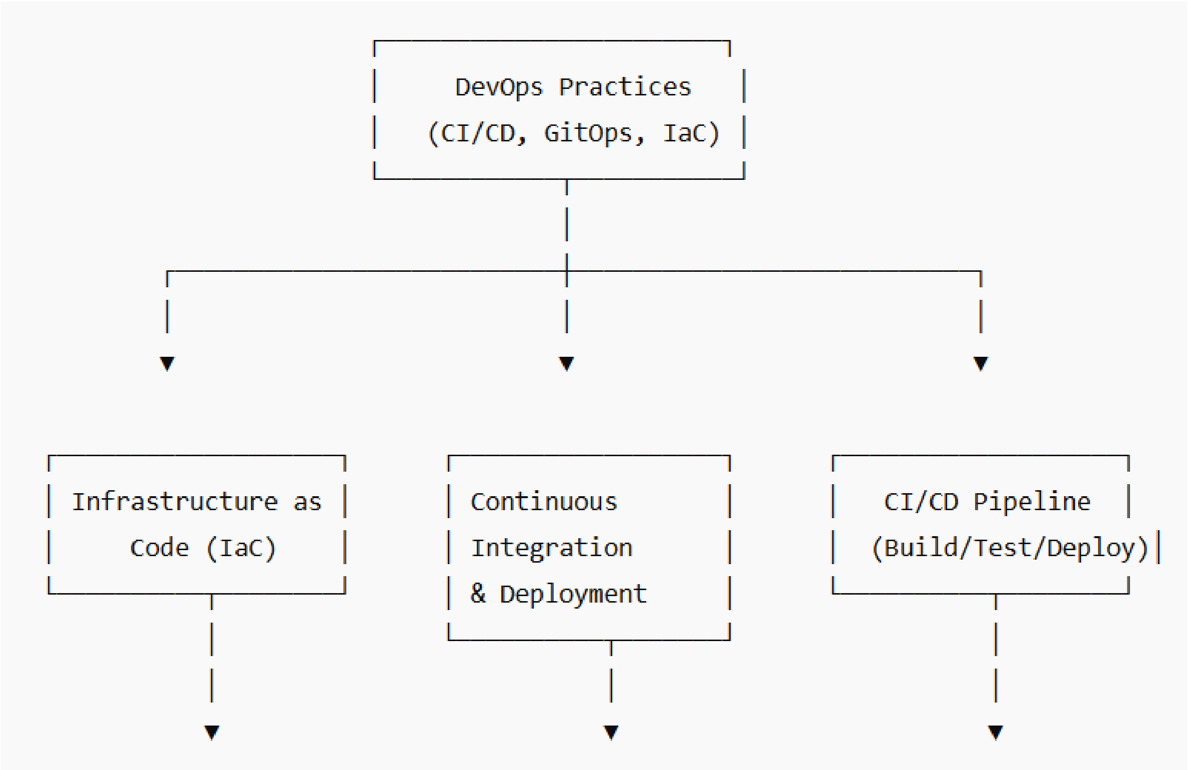
→ Overarching culture + tools enabling automation and reliability

9-Infrastructure as Code (IaC)

→ Creates the underlying infrastructure

10-Continuous Integration & Continuous Deployment CI/CD

→ Builds and deploys microservices and serverless code



1 – Microservices

What Are Microservices?

Microservices is an **architectural style** where an application is divided into many small, independent services.

Each service:

- performs a **single business function**
- can be **developed, deployed, and scaled** independently
- communicates with other services via **APIs**

It is the opposite of a **monolithic** architecture.

Monolithic Architecture

- A **monolithic architecture** is a traditional software design where the entire application is built as **one large, tightly integrated unit**.

- All components—UI, business logic, and database access—are combined into a **single codebase** and deployed together.

Disadvantages of Monolithic Architecture

- **1. Poor Scalability**

You must scale the whole application—even if only one part needs more resources.

- **2. Difficult to Maintain**

As the app grows, the codebase becomes large, complex, and hard to understand.

- **3. Tight Coupling**

A small change can affect many other modules.

Disadvantages of Monolithic Architecture

- **4. Slow Deployment**

You must redeploy the entire application for any update.

- **5. Not Fault Tolerant**

One small component failure may cause the entire system to fail.

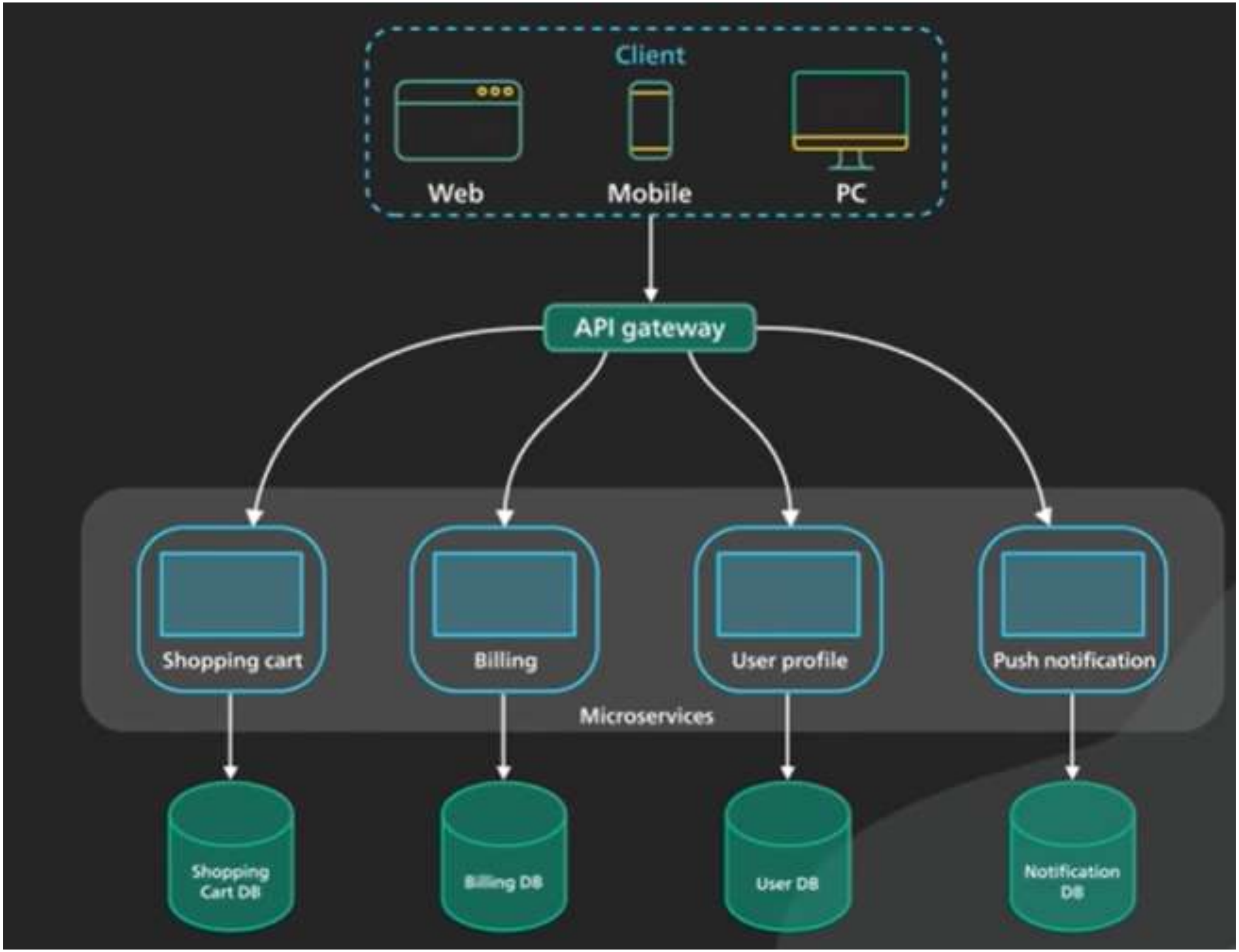
- **6. Technology Lock-In**

Since everything is one project, it's hard to adopt new tech/languages.

Microservices: Breaking down

- Split monolithic (1 codebase into many ms-i)

Microservices: Breaking down



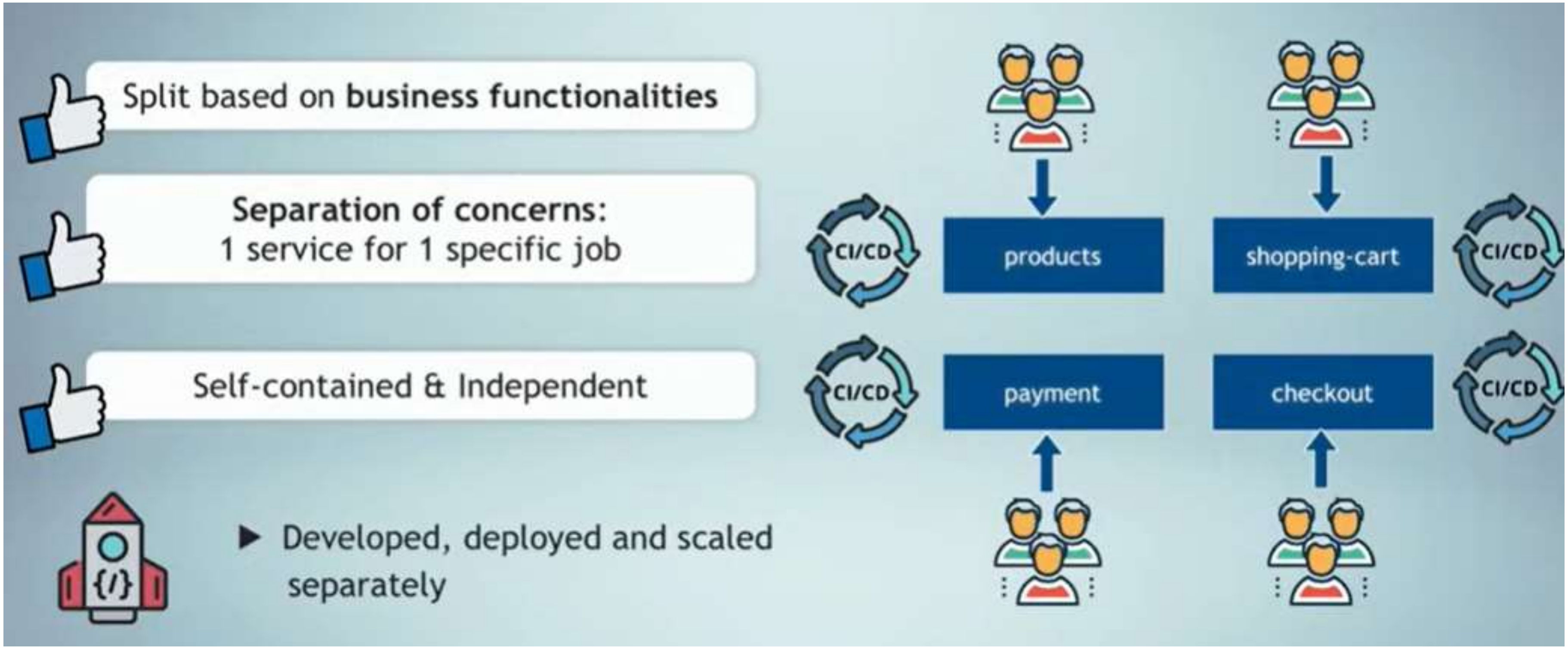
Microservices: Breaking down

Q1: How to break down the application?

Q2: How big or small they should be?

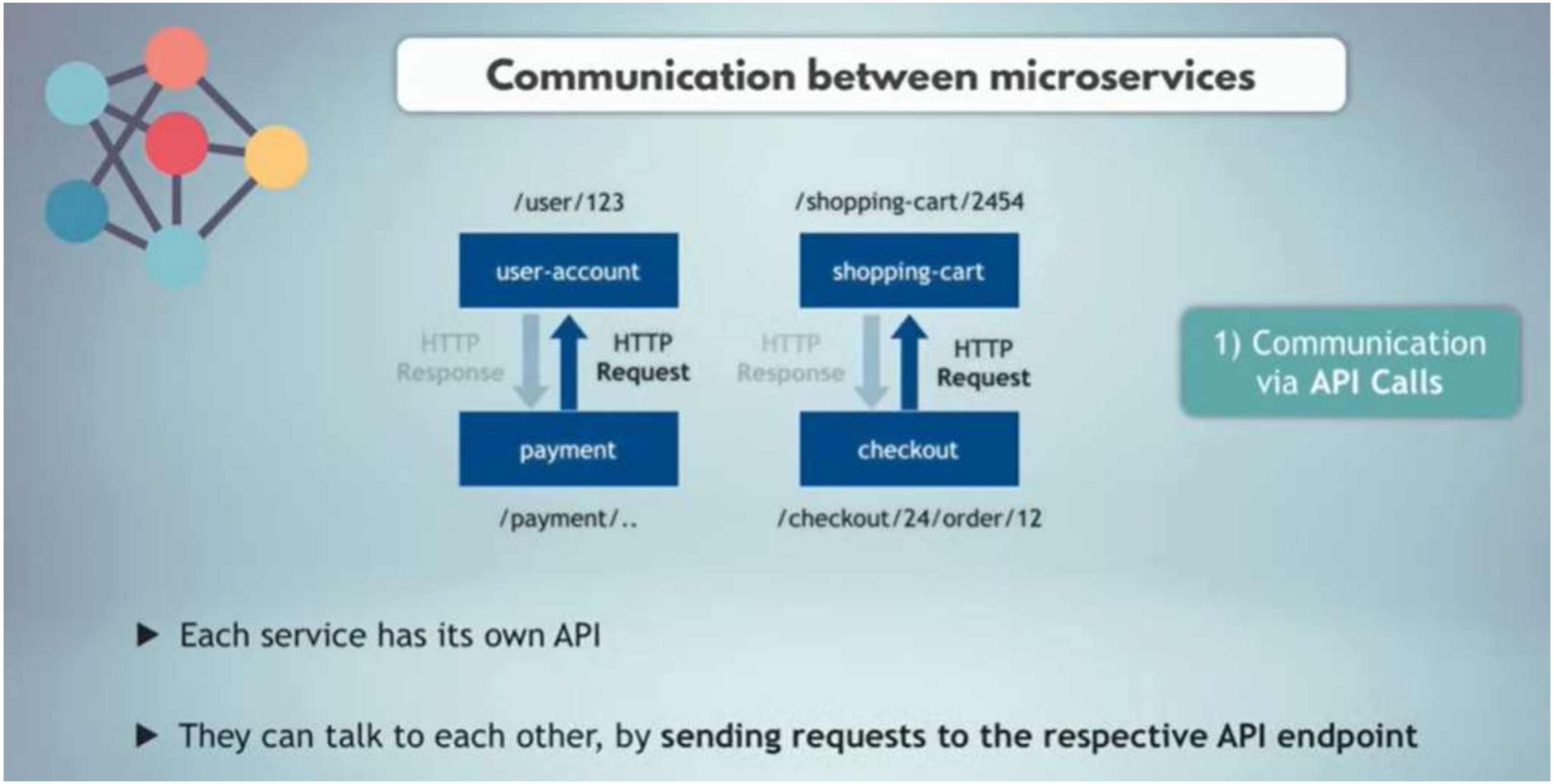
Q3: How many microservices?

Q4: What code goes where? (Loosely coupled)



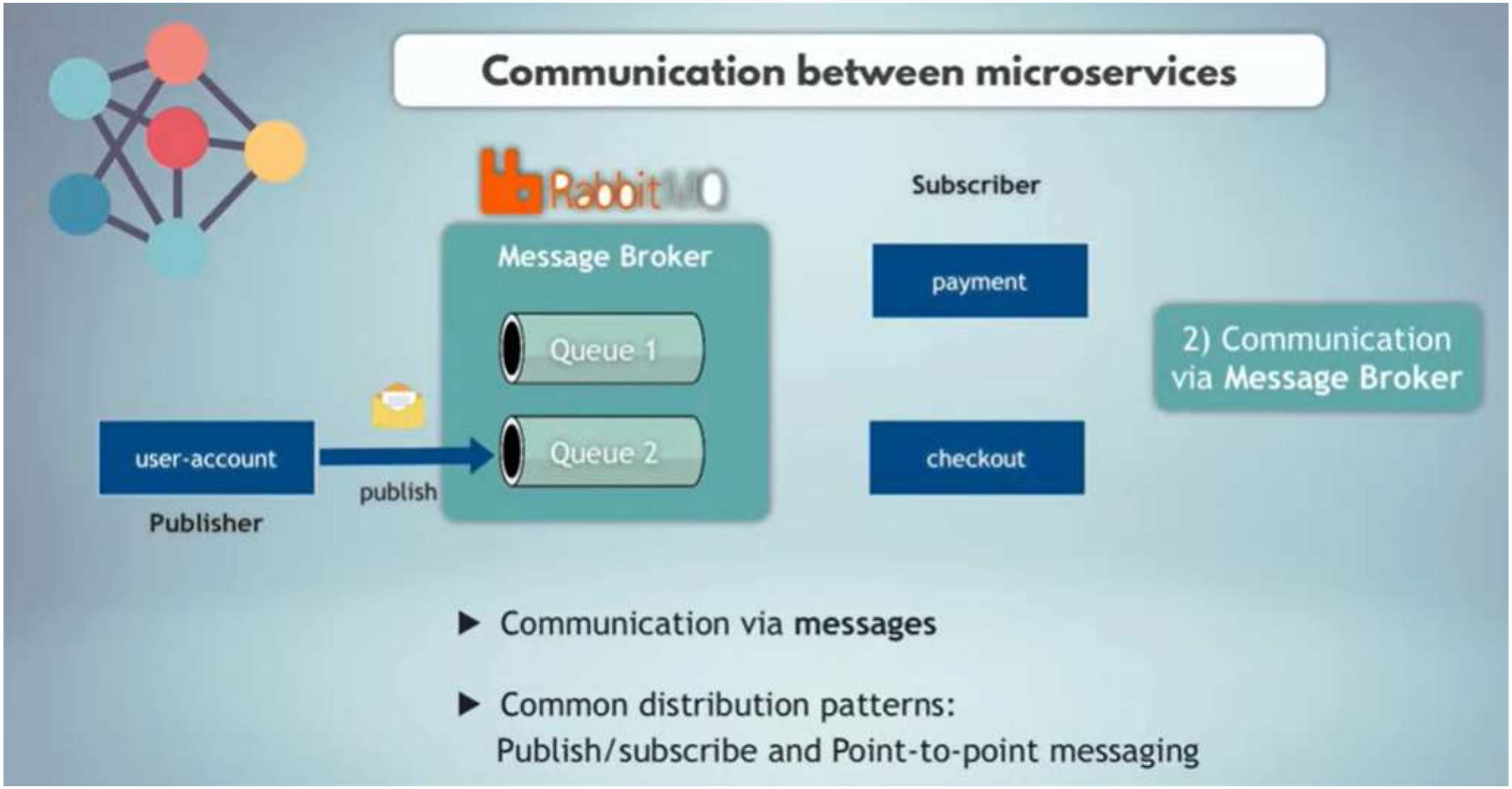
Q5: How do they communicate?

- **1) Synchronous Communication (via API Calls)**



Q5: How do they communicate?

- **1) Asynchronous Communication (via Message Broker)**



Advantaged of Microservices

- **1. Scalability**

Scale each service independently (e.g., scale Payment without scaling User).

- **2. Faster Development**

Different teams develop different services. Improves productivity.

- **3. Flexibility**

Use different languages/technologies per service.

- **4. Fault Isolation**

If Order service fails, User service continues working.

- **5. Continuous Deployment**

Small services → fast deployment, minimal risk.

Disadvantaged of Microservices

- **1. Increased Complexity**

Many small services = complex network.

- **2. Harder Debugging and Monitoring**

Distributed logs and errors.

- **3. Network Latency**

Services talk over the network → delays possible.

- **4. Data Consistency Challenges**

Each service has its own DB → needs eventual consistency.

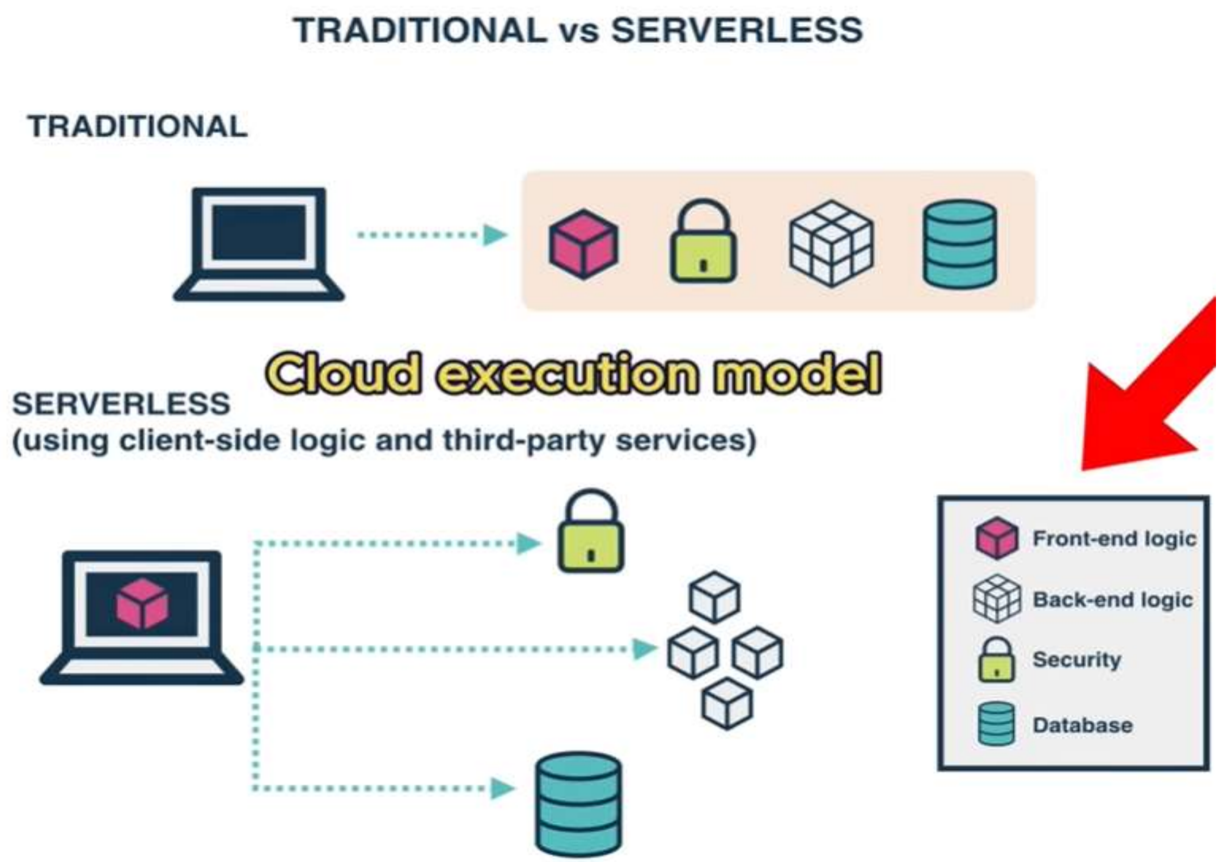
- **5. Requires DevOps Expertise**

Automation, CI/CD, container orchestration (K8s) required.

2 – Serverless (Functions as a Service: AWS Lambda, Azure Functions)

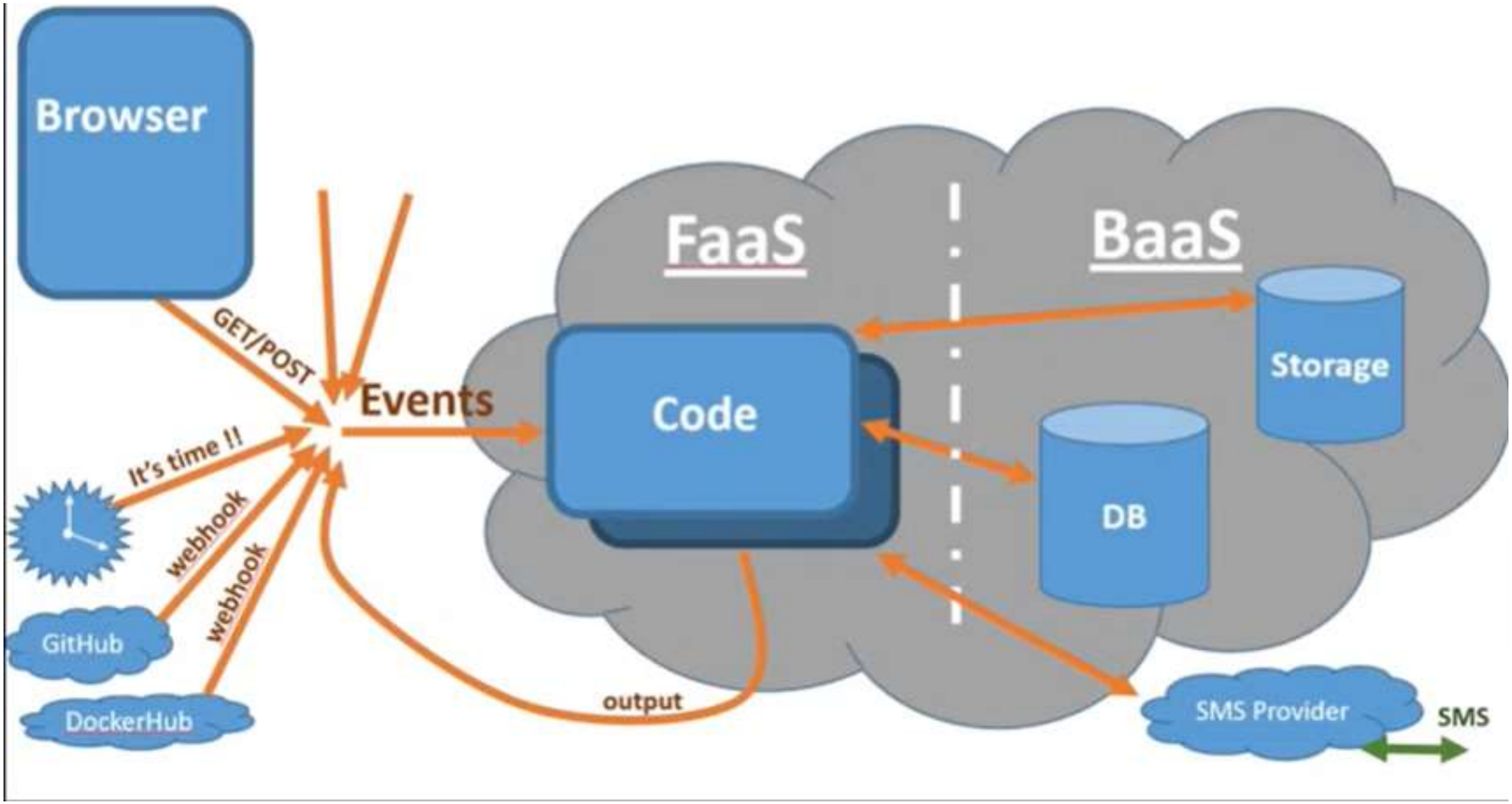
Serverless computing

- **Serverless computing** allows you to run code without provisioning or managing servers. You only upload functions; the cloud handles everything else.
- Though servers exist, they are **fully hidden** from the developer.



Serverless computing

- Serverless = Function as a Service (FaaS) + Backend as a Service (BaaS).



Serverless – FaaS, BaaS

- **FaaS** is a cloud-native compute model that lets you deploy small, single-purpose pieces of code — called ***functions*** — that execute only when triggered.
- you pay only for the time the function runs.

- **BaaS** platforms give developers **prebuilt backend** functionality, accessible via **APIs**, such as:
- Authentication & authorization
- Database & storage
- File handling
- Analytics & logging
- Hosting
- API management, …

Serverless computing – Key Features

1-No server management → you don't provision or maintain servers. No patching, no OS updates, no cluster management.

2-Automatic scaling → scales instantly based on traffic, scales up during high traffic and scales to zero when idle.

3-Pay-per-use → you pay only for execution time, not idle time. Billing is based on:

- Number of requests
- Execution time
- Memory/CPU usage

Serverless computing – Key Features

4-Event-driven → functions run in response to <u>triggers</u>:

- HTTP requests
- Database changes
- Messaging queues
- File uploads
- Scheduled events

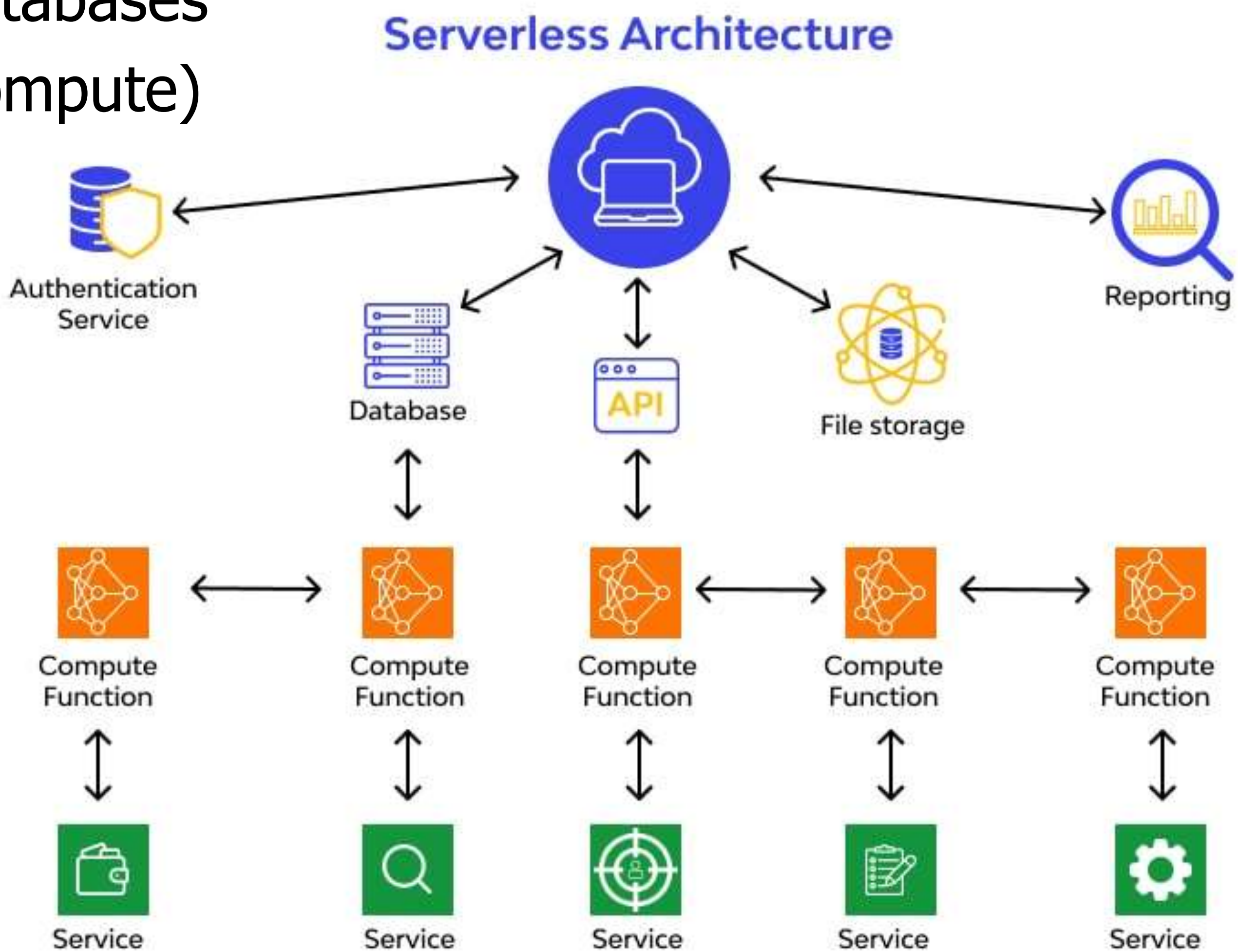
5-Stateless → functions are stateless. No stored session data on the server. State is stored in databases, caches, or storage services.

Serverless – Disadvantages

- Cold starts (delay on first request)
- Limited execution time
- Vendor lock-in
- Harder debugging for distributed functions
- Stateless by design

Serverless – Architecture Components

- Event sources (API Gateway, queues, cron, file upload triggers)
- Authentication services
- Serverless databases
- Functions (compute)
- Monitoring
- Reporting



Serverless – Services

- **Compute**
  - AWS Lambda
  - Azure Functions
  - Google Cloud Functions
  - Cloudflare Workers

- **Serverless Containers**
  - AWS Fargate
  - Google Cloud Run
  - Azure Container Apps

Serverless – Services

- **Serverless Databases**
  - DynamoDB
  - Aurora Serverless
  - Firestore
  - Cosmos DB

- **Serverless Storage**
  - Amazon S3
  - Google Cloud Storage

Serverless computing – AWS Lambda

- Runs small pieces of code in response to **events** i.e. when triggered by:
  - API Gateway request
  - File upload to S3
  - Message in queue
  - Cron-like schedules
  - Database changes

- You upload your function code, and Lambda executes it **on-demand**, scaling automatically.

Serverless computing – AWS Lambda

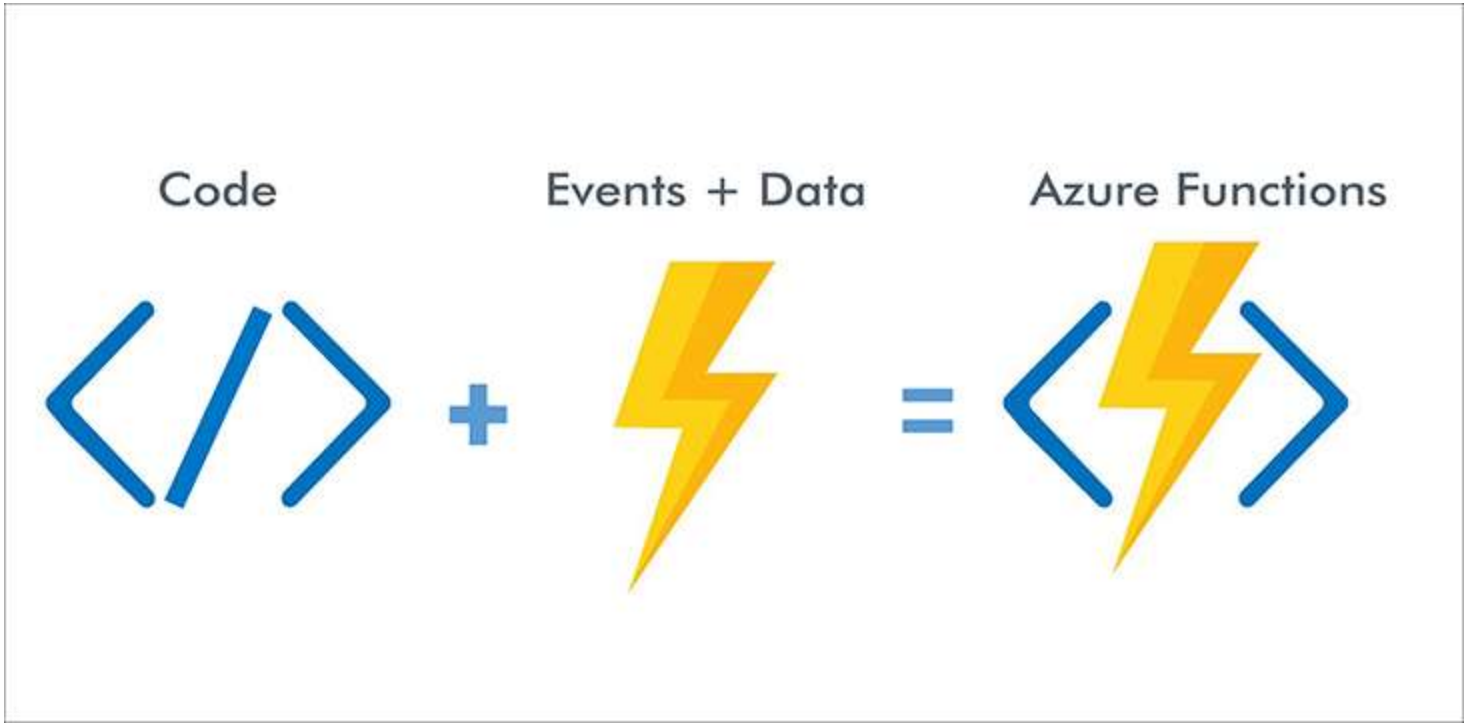
- **How AWS Lambda Works:**
- You write a function (Python, Node.js, Java, Go, .NET, Ruby, etc.).
- Deploy it to AWS Lambda.
- Configure an **event trigger**.
- Lambda runs the function only when needed.

- You only pay for the **duration** your function runs (ms-based). However, you have only 15-minute maximum execution time.

Serverless computing – Azure Functions

- **Azure Functions** is Microsoft Azure's serverless compute service that lets you run code **on-demand** without managing servers.

- It follows an **event-driven** model just like AWS Lambda.



Serverless computing – Azure Functions

- **How Azure Functions Works:**
- Write a function (C#, Python, JavaScript, Java, PowerShell, etc.)
- Deploy it to Azure Functions.
- Bind it to a Trigger (something that starts the function).
- (Optional) Use Bindings to integrate with other Azure services.
- Azure runs the function only when needed and scales automatically.

3 – API Gateways

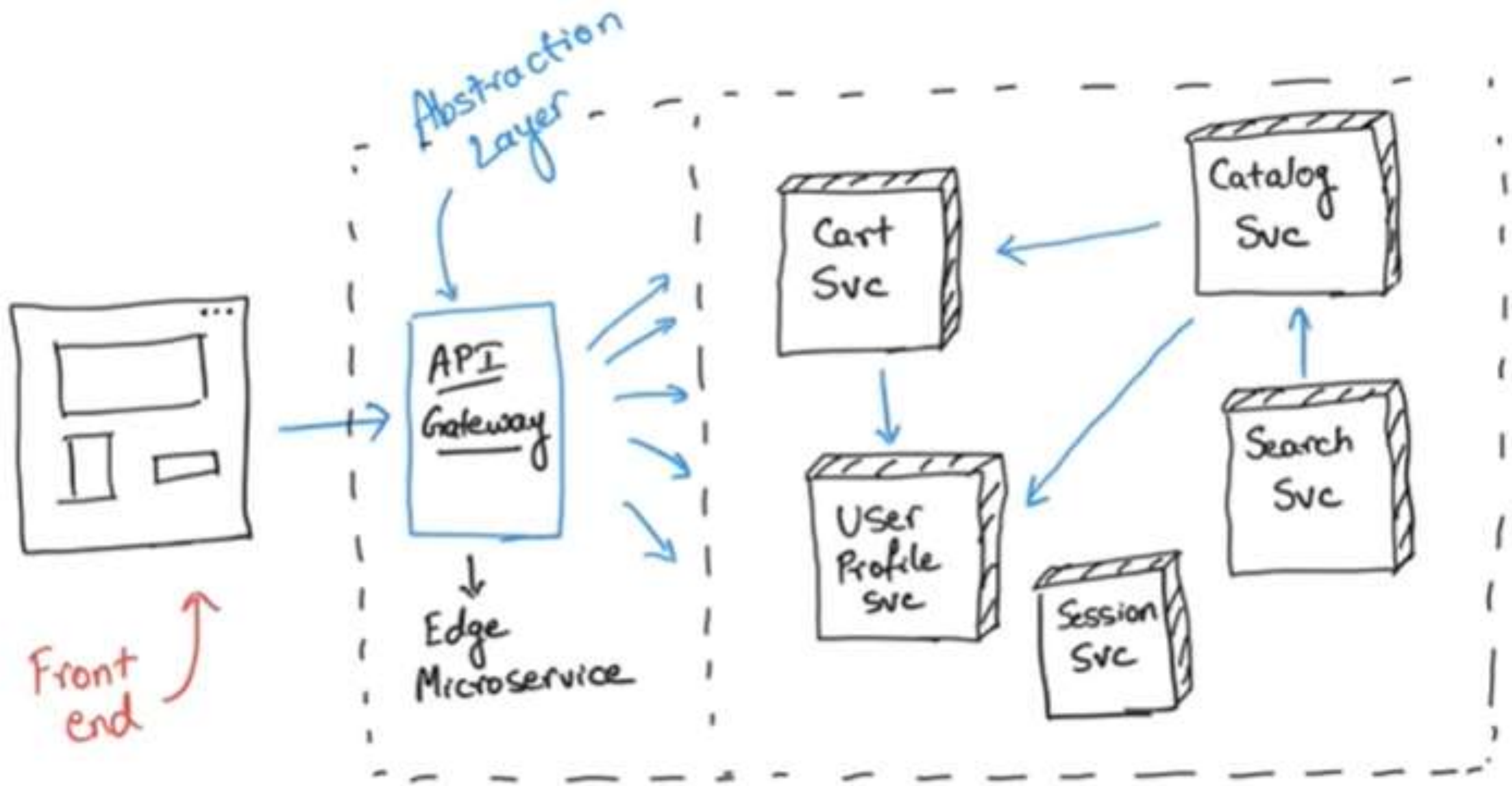
API Gateways

- An **API Gateway** is a critical component in a microservices architecture, acting as a **single-entry point** for client requests.

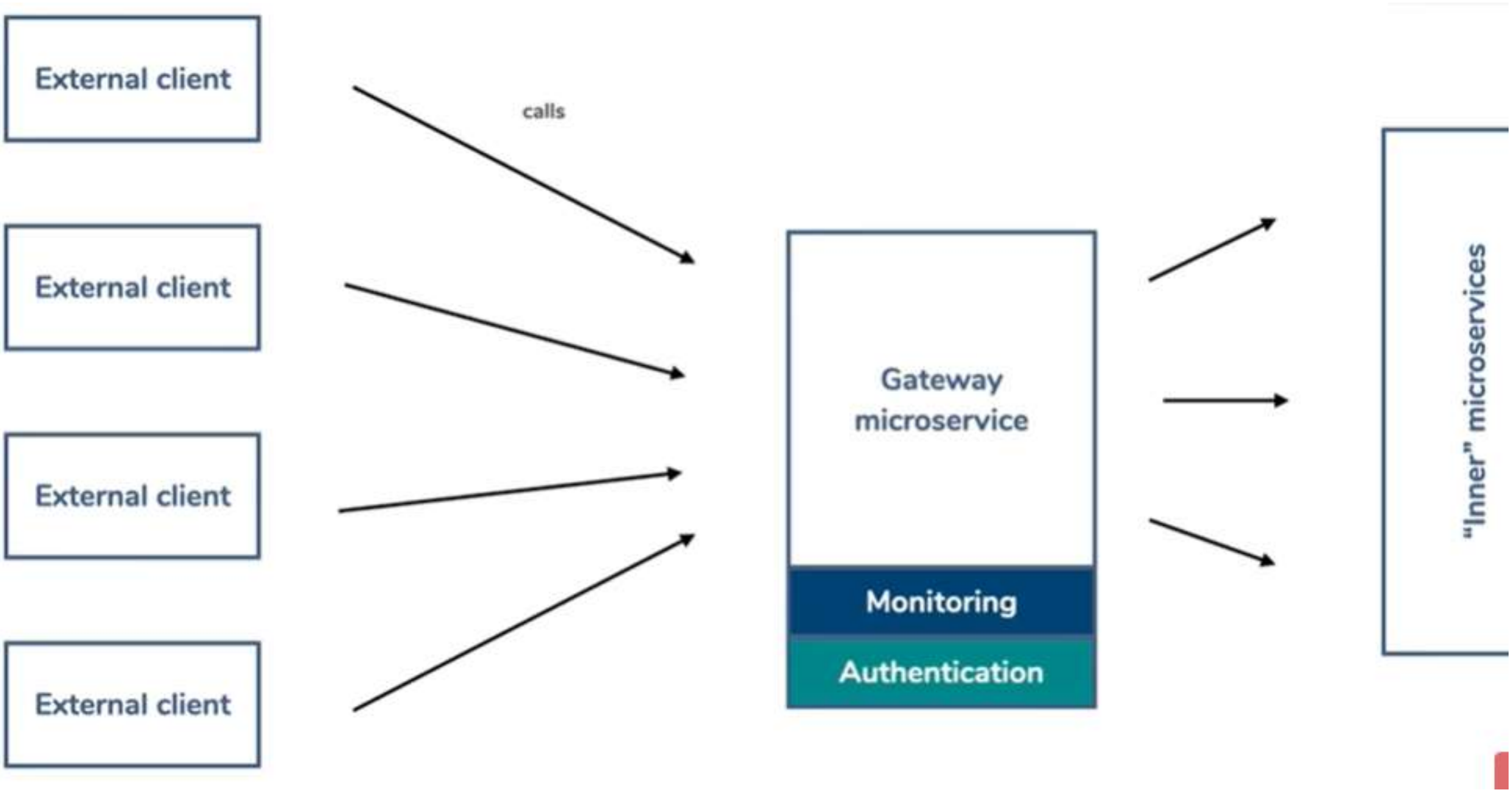
- It simplifies communication between clients and multiple microservices by managing **routing**, **security**, and **protocol** translation.

- This approach enhances scalability, security, and performance in cloud systems.

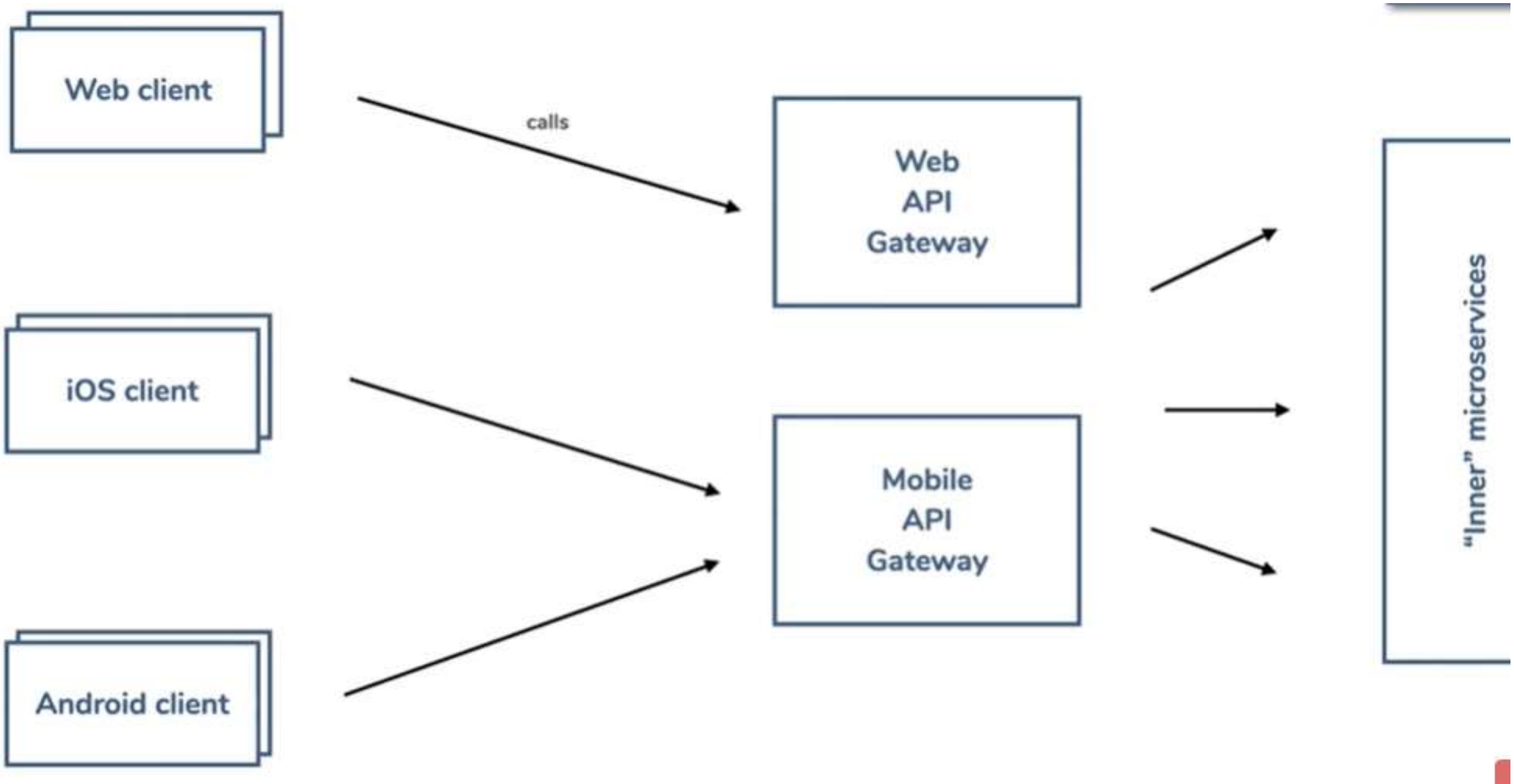
API Gateways



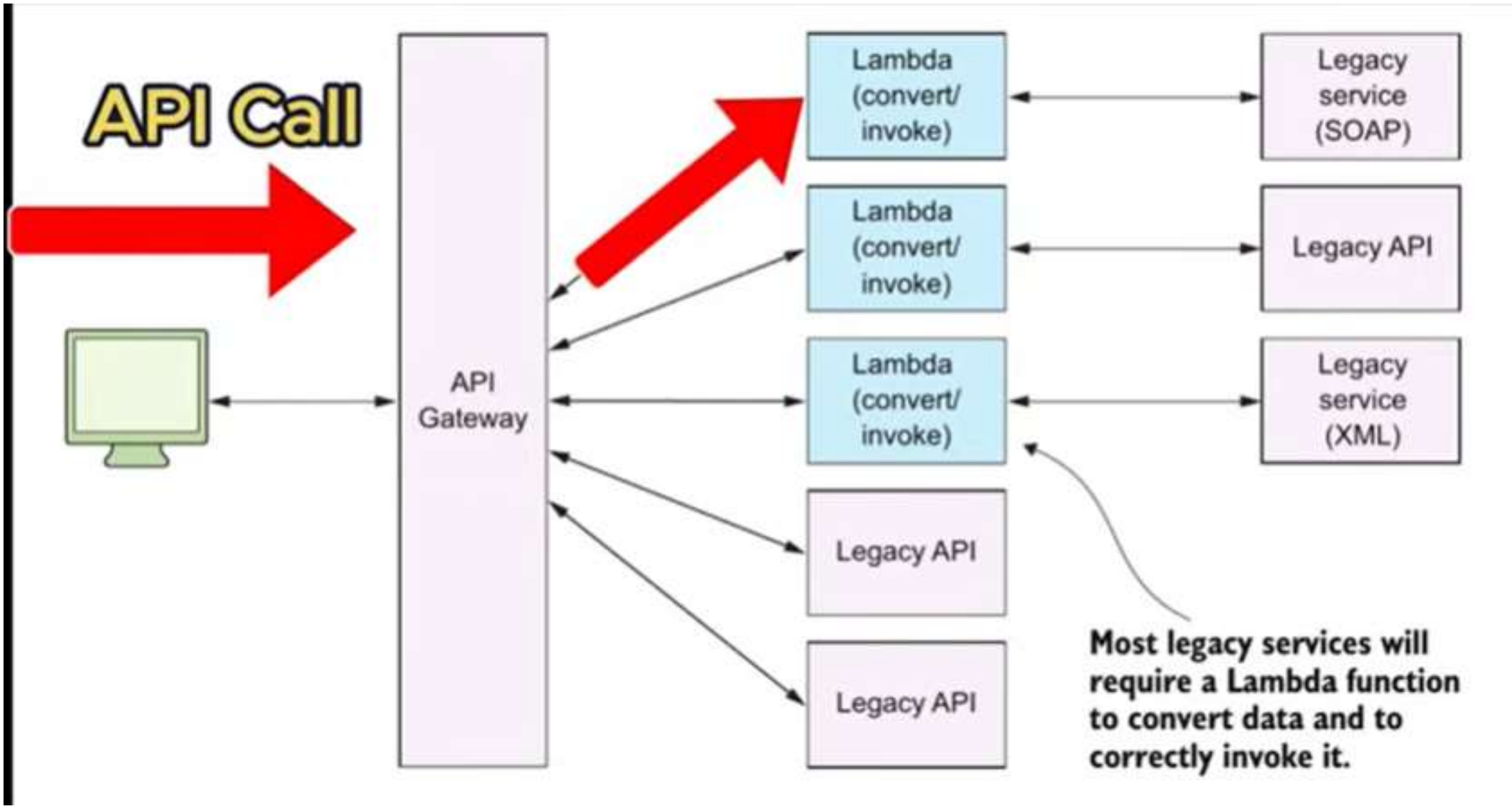
API Gateways



API Gateways



Route and invoke the correct Lambda



Key Features of an API Gateway

- **Routing and Load Balancing**: It routes client requests to the appropriate microservice based on predefined rules and balances the load across multiple service instances to ensure reliability and scalability.

- **Protocol Translation**: It translates protocols (e.g., HTTP to gRPC) and data formats to ensure compatibility between clients and backend services.

- **Request Transformation**: It modifies incoming requests or outgoing responses, such as altering headers, parameters, or payloads, to meet backend service requirements.

Key Features of an API Gateway

- **Caching**: It caches frequently requested data to reduce latency and improve response times, minimizing the load on backend services.

- **Security**: It centralizes security measures like authentication, authorization, and encryption, reducing the burden on individual microservices.

Benefits of Using an API Gateway

- **Centralized Management**: It provides a unified entry point, simplifying traffic management, security policies, and API monitoring.
- **Improved Security**: It enforces authentication, authorization, and SSL termination, protecting microservices from direct exposure.
- **Scalability**: It distributes requests across service instances, ensuring high availability and optimal resource utilization.
- **Protocol Agnosticism**: It allows clients to use their preferred protocols, enabling seamless integration with diverse systems.
- **Performance Optimization**: It reduces network overhead by aggregating responses from multiple microservices into a single response.

Common API Gateway Patterns

- **Gateway Aggregation**: Combines responses from multiple microservices into a single response, reducing client-server interactions. For example, an e-commerce platform can aggregate product, payment, and shipping APIs into one endpoint.

- **Gateway Offloading**: Offloads tasks like authentication, rate limiting, and request validation to the gateway, reducing the complexity of individual microservices.

- **Gateway Routing (Backend for Frontend (BFF))**: Directs requests to the appropriate microservice based on URL paths, headers, or metadata. Different gateways for mobile/web/IoT. For instance, requests for product details are routed to the product catalog service.

Common API Gateway Patterns

- **Gateway Transformation**: Modifies requests or responses to ensure compatibility between clients and services, such as converting JSON to XML or enriching responses with additional metadata.

- **Gateway Security (Edge Proxy Pattern)**: Gateway normally sits at cloud edge for performance/security. It implements security measures like OAuth, SSL/TLS encryption, and input validation to protect microservices from unauthorized access and attacks.

Challenges

- **Complexity**
- **Single point of failure**

- Should ensure high availability, scalability, and robust monitoring to mitigate these risks !!

4 – Service Mesh

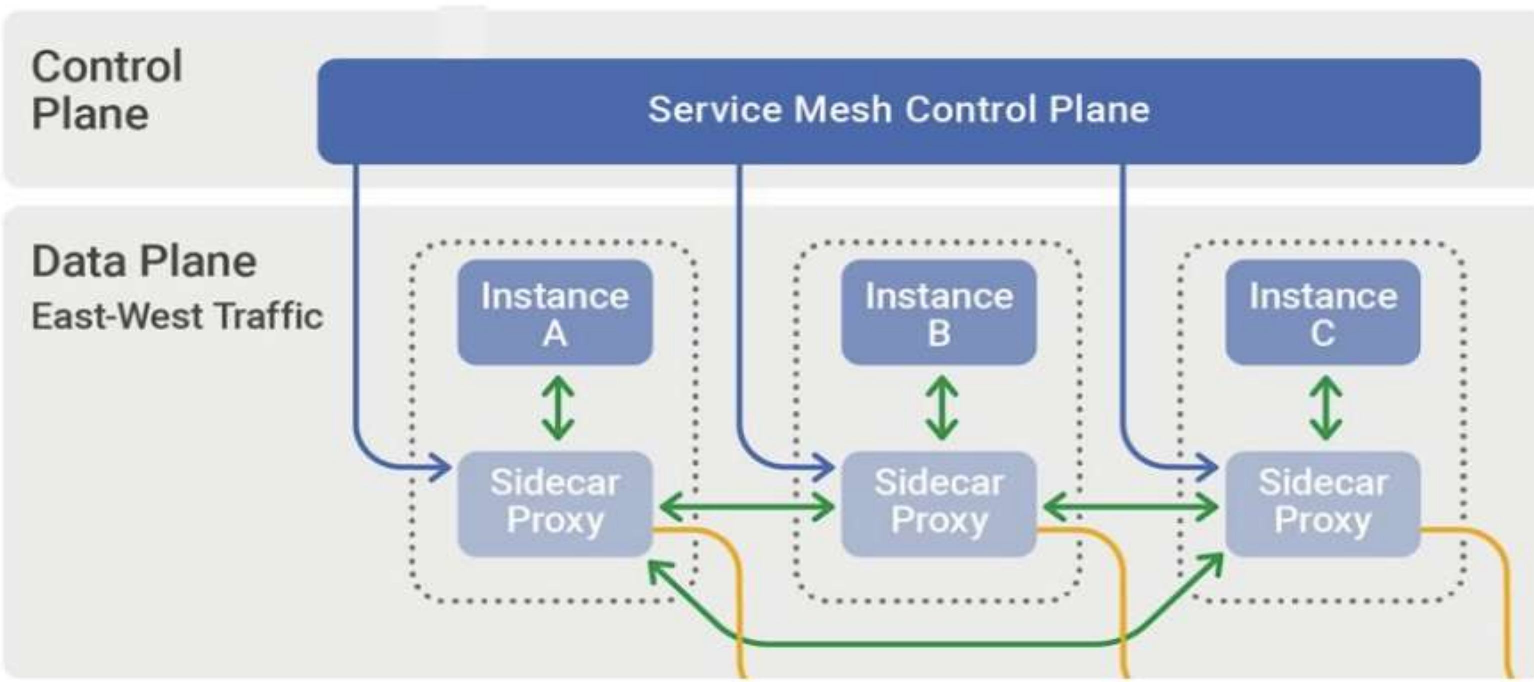
Service Mesh

A **service mesh** is a dedicated infrastructure layer that manages communication between microservices automatically due to the need of secure, reliable, and observable communication at scale.

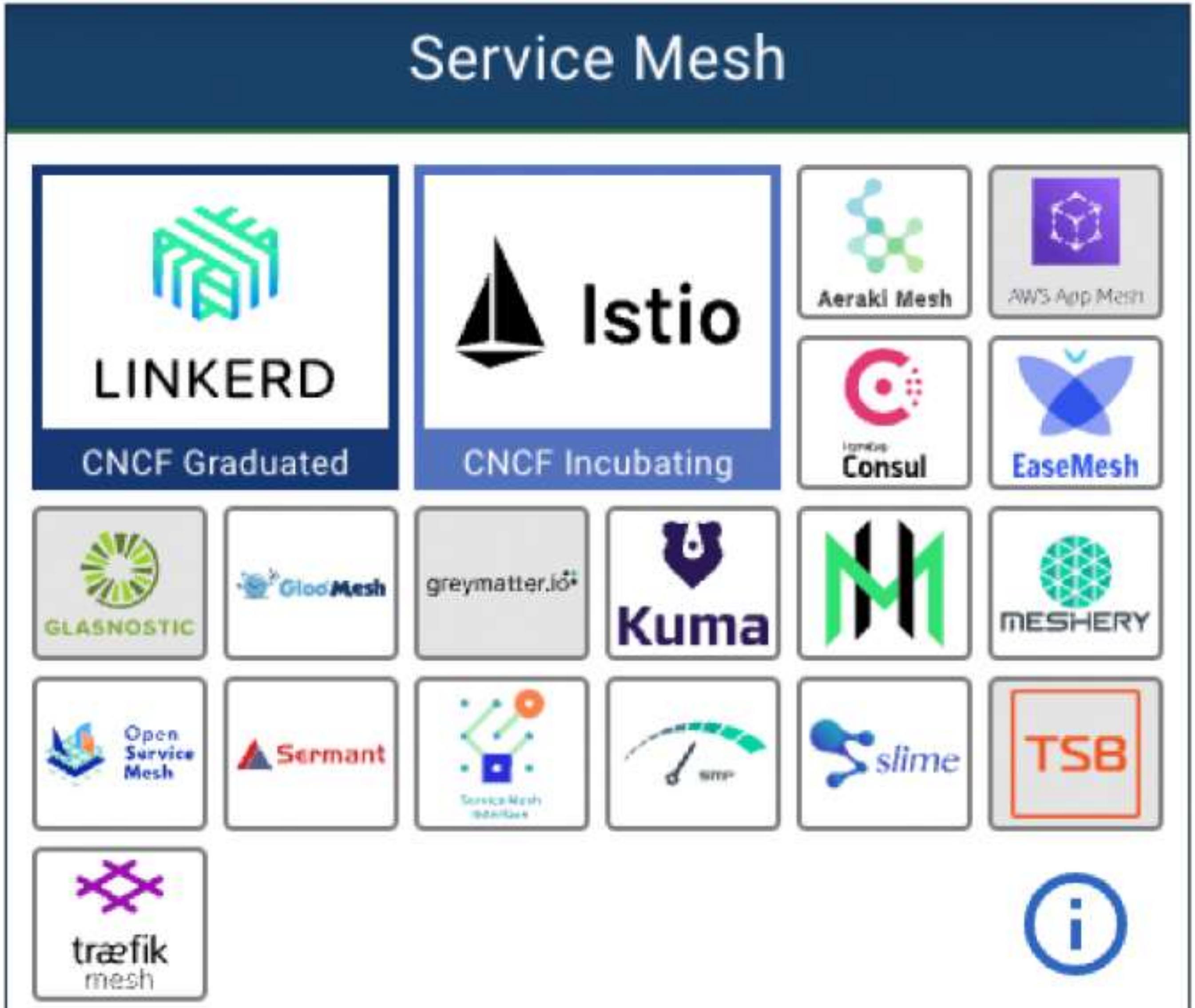
- **It Provides:**
- Secure service-to-service communication (mTLS)
- Traffic management (routing, retries, circuit breaking)
- Observability (metrics, logs, tracing)
- Policy enforcement
- Zero-trust networking
- Service discovery
- Load balancing

Service Mesh

- **It** Uses **sidecar proxies** (e.g., Envoy) that handle communication for each service:
- Each microservice instance runs with a lightweight proxy (sidecar).
- All inbound/outbound traffic goes through the proxy.
- Mesh control plane configures these proxies.



Service Mesh – Examples



5 – Cloud-Native Database

Cloud-native Database

A **cloud-native database** is designed to run optimally in cloud environments—scalable, distributed, and resilient by default.

Each microservice typically uses:

- its **own database** (Database per Service pattern)
- often independent storage engines
- accessed over APIs rather than shared direct queries

Cloud-native Database – Characteristics

- Fully managed by the cloud provider
- Auto-scaling and high availability
- Distributed architecture across nodes or regions
- Pay-as-you-go pricing
- Self-heal, Integrated backup & monitoring
- Update without downtime
- Handle massive workloads
- Integrate deeply with cloud platforms

Cloud-native Database – Types

- **Distributed SQL Databases**
  - Provide SQL + horizontal scalability.
  - Examples: Google Cloud Spanner, YugabyteDB, CockroachDB, Amazon Aurora Serverless, Azure Cosmos DB (SQL mode)

- **NoSQL Cloud Databases**
  - Designed for massive scale and high throughput.
  - Examples: DynamoDB, Cassandra (AstraDB), MongoDB Atlas, Cosmos DB (Mongo/NoSQL APIs), Bigtable

- **Serverless Databases**
  - Scale automatically to zero when idle.
  - Examples: Aurora Serverless V2, DynamoDB On-Demand, PlanetScale

6 – Containers (Docker)

Containers

A **container** is a lightweight, portable environment that packages an application with all its dependencies.

- It Works the same everywhere (no “it works on my machine” issues), Fast to start, Uses fewer resources than virtual machines, Ideal for microservices

Docker is the most commonly used containerization platform.

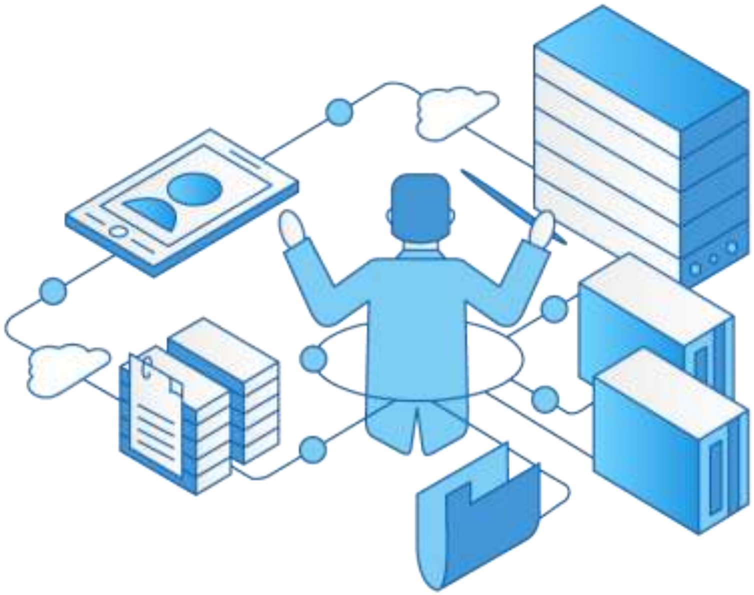
Key Concepts:

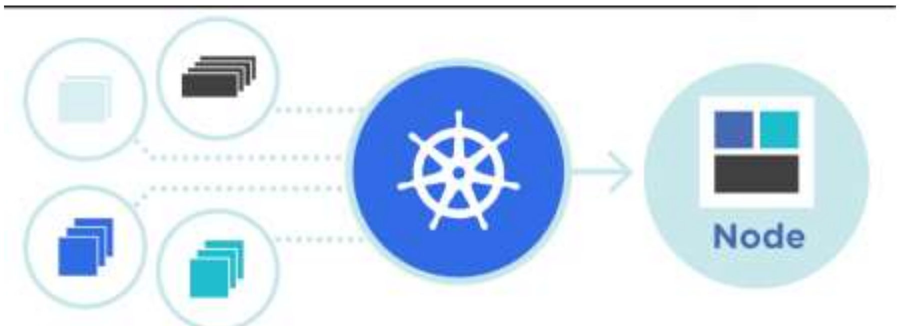
- **Image** → template
- **Container** → running instance
- **Dockerfile** → instructions to build an image
- **Docker Hub** → image repository

7 – Orchestration (Kubernetes)

Orchestration (Kubernetes)

- **Orchestration** is the automated management of containerized applications—handling deployment, scaling, networking, and lifecycle.
- **Kubernetes (K8s)** is the most widely used container orchestration platform. Used when you have many microservices running in containers, where manual management becomes impossible.





Orchestration (Kubernetes)

- **Before orchestration, teams faced challenges:**
  - Managing dozens or hundreds of containers
  - Restarting crashes manually
  - Handling traffic routing
  - Scaling services during peak load
  - Managing updates without downtime
  - Coordinating microservices communication

- **Kubernetes automates all of this.**

Application Deployment Evolution

- **2000s:** Physical servers → slow provisioning, snowflake servers
- **2010–2014:** Virtual machines → faster but still heavy
- **2013–2016:** Containers (Docker) → lightweight, portable runtime
- **Today:** Kubernetes → orchestration, automation, scalability, resilience

- **2014:** Kubernetes open-sourced by Google
- **2015:** Donated to the CNCF (Cloud Native Computing Foundation)
- **2017+:** Explosion in adoption → major cloud providers adopt it as a managed service

Core Responsibilities of Kubernetes as an Orchestrator

1. Automated Deployment & Scheduling

- Decides where containers should run based on CPU, RAM, and resource needs
- Ensures the application reaches the desired state declared by the developer

2. Scaling (Auto-Scaling)

- Horizontal Pod Autoscaler (HPA): replicate Pods based on metrics
- Vertical Pod Autoscaler (VPA): adjust Pod resource sizes
- Cluster Autoscaler: add/remove nodes based on demand

Core Responsibilities of Kubernetes as an Orchestrator

3. Self-Healing
- Automatically restarts failed containers
- Replaces unresponsive Pods
- Ensures the correct number of replicas

4. Service Discovery & Load Balancing
- Services get stable IPs
- Traffic is automatically routed to healthy Pods
- Nearest or least-loaded Pod receives traffic

5. Rolling Updates & Rollbacks
- Update applications with **zero downtime**
- Roll back if the update fails

Core Responsibilities of Kubernetes as an Orchestrator

6. Configuration & Secrets Management

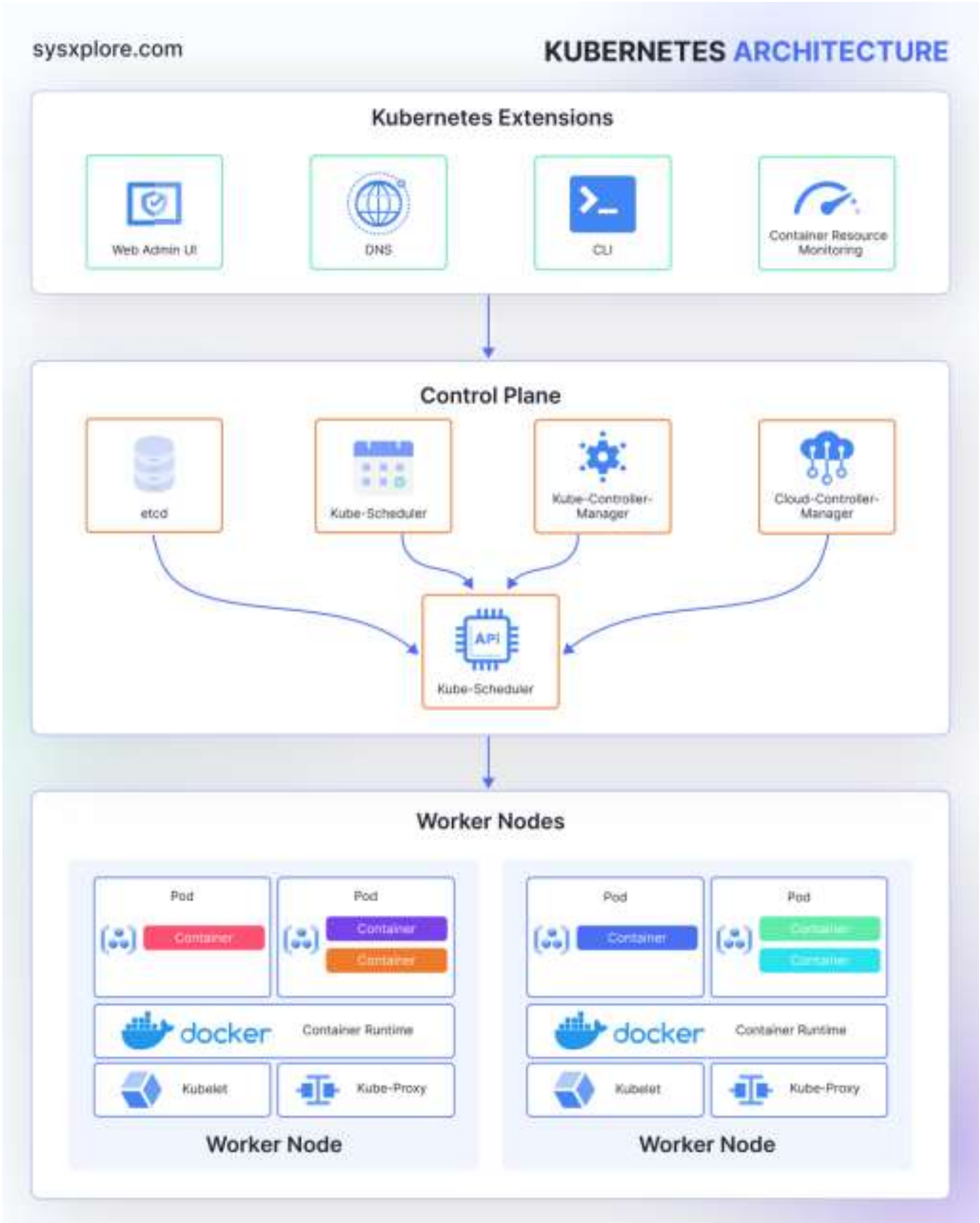
- Separates config from code using **ConfigMaps**
- Stores sensitive data in **Secrets**

7. Storage Orchestration

- Automatically mounts persistent storage volumes
- Supports cloud storage (EBS, Azure Disk, GCP Persistent Disk)

8. Multi-cloud flexibility

Kubernetes Components



Control Plane – etcd

etcd (The Key-Value Store)

- Distributed, strongly consistent (Raft-based)
- Stores **cluster state**, including:
  - Resource definitions
  - Secrets (by default base64, not encrypted unless configured)
  - Configs
  - Node state

Control Plane – Kube-API

Kube-API Server (The "Front Door")

- The only component in Kubernetes that talks to etcd.
- All external communication passes through the API server.
- Exposes REST endpoints.
- Responsible for:
  - authentication
  - authorization
  - admission control
  - validation
  - request routing

Control Plane – Kube Controller Manager

Kube Controller Manager

- Runs core controllers:
  - **Deployment Controller** — ensures correct number of replicas
  - **ReplicaSet Controller**
  - **Node Controller**
  - **Endpoint Controller**
  - **Job Controller**
  - **Service Account Controller**

- Controllers constantly compare desired state (Spec) vs actual state (Status) and take actions until they match.

Control Plane – Cloud Controller Manager

Cloud Controller Manager (CCM)

- allows the cluster to integrate with a cloud provider's infrastructure.
  It runs only when Kubernetes is deployed **on a cloud provider.**

Control Plane – Scheduler

Scheduler

- Responsible for placing Pods on Nodes.
- **Scheduling logic overview:**
  - **Filter** nodes
    - resources (CPU/Memory)
    - taints/tolerations
    - affinity rules
  - **Score** remaining nodes
  - **Bind** Pod to the selected node

Worker Nodes – Pod

Pod

- is a smallest deployable unit.
- is a wrapper around one or more containers.
- It represents a **single instance of a running application** in the cluster.

- **Pod Lifecycle**
  - **Pending** → waiting for scheduling
  - **Running** → one or more containers running
  - **Succeeded / Failed** → containers stopped
  - **CrashLoopBackOff** → container repeatedly crashes
  - **Terminating** → shutting down

Worker Nodes – Kubelet

Kubelet (The Node Agent)

- Runs on every node.
- is like a supervisor on each worker machine.
- Responsibilities:
  - Ensures containers are running
  - Watches the API server for Pod specifications
  - Starts/stops containers using Container Runtime (containerd, CRI-O, etc.)
  - Reports pod status and resource usage
  - Executes readiness/liveness/startup probes

Worker Nodes – Container Runtime

Container Runtime

- Common runtimes:
  - containerd
  - CRI-O
  - Docker (deprecated as default runtime)
  - gVisor, Kata Containers for sandboxed workloads
- Must implement the **CRI (Container Runtime Interface)**.

Worker Nodes – Kube-Proxy

Kube-Proxy

- Networking component.
- Functions:
  - Implements **Service** abstraction
  - Automatically manages iptables or IPVS rules
  - Provides cluster-wide service discovery and load balancing

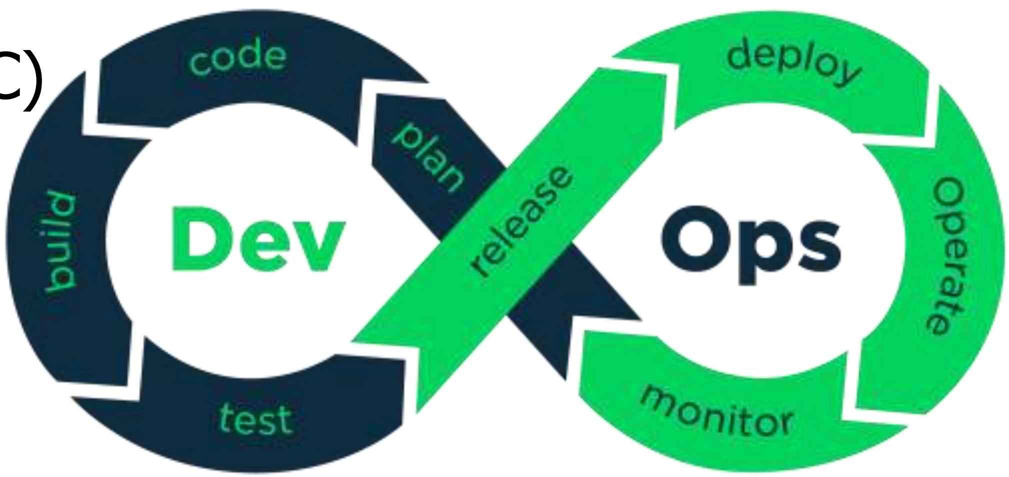
8 – DevOps practices

DevOps practices

DevOps is a set of **cultural principles + practices + tools** that improve collaboration between development and operations teams.

Core DevOps Practices:

- CI/CD automation
- Infrastructure as Code (IaC)
- Monitoring & logging
- Version control (GitOps)
- Automated testing
- Continuous feedback
- Microservices & containerization
- Collaboration and shared ownership



DevOps practices

Goals:

- Faster development cycles
- More reliable releases
- Improved collaboration
- High automation
- Scalability and stability

Popular DevOps Tools:

- Jenkins, GitLab, GitHub Actions
- Terraform, Ansible
- Kubernetes
- Prometheus, Grafana
- Docker

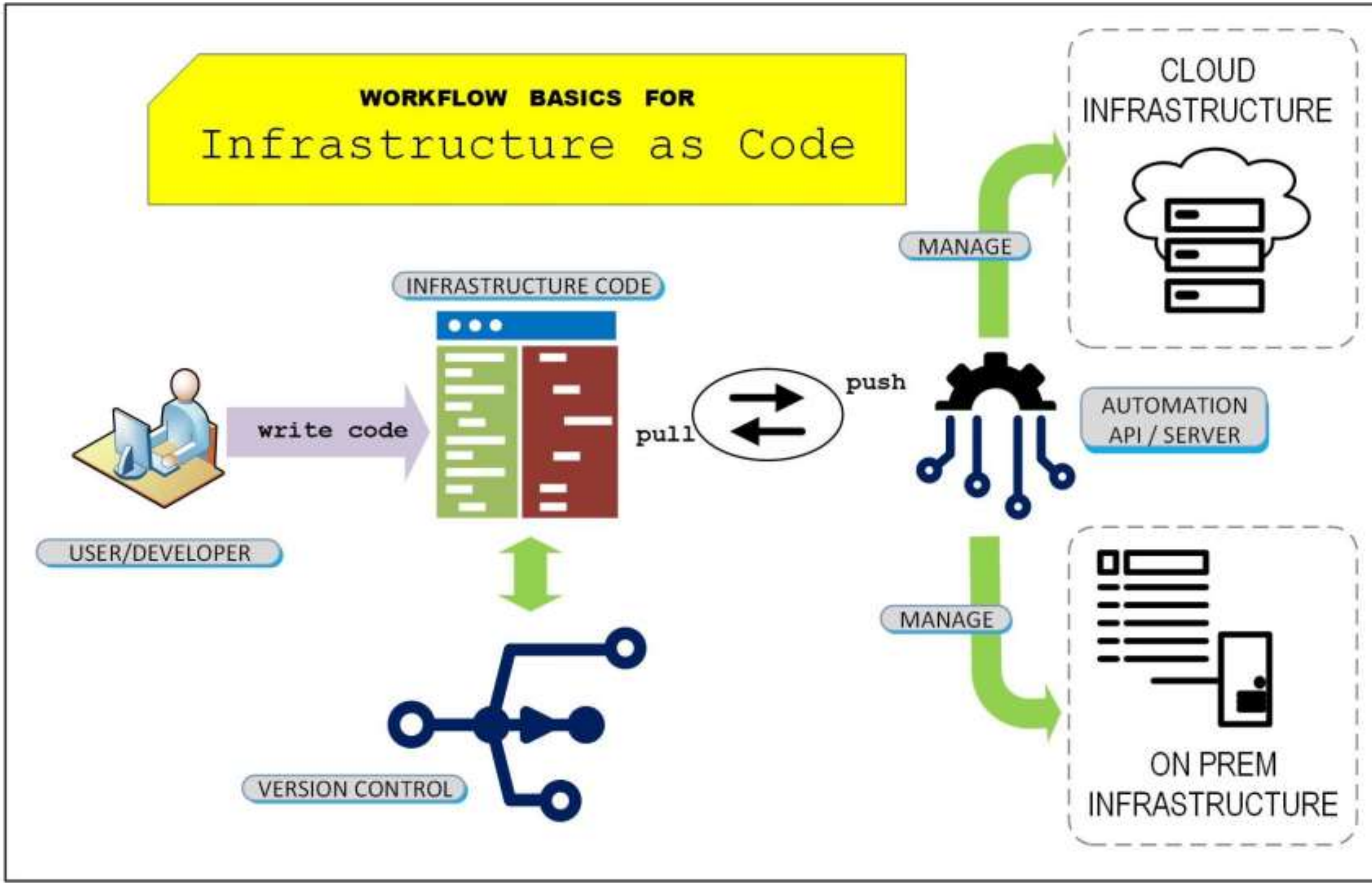
9 – Infrastructure as Code (IaC)

What is IaC ?

- **Infrastructure as Code (IaC)** is a modern approach to managing and provisioning IT infrastructure using machine-readable configuration files instead of manual processes.

- It allows you to **define, deploy, and update infrastructure (servers, networks, databases, load balancers, etc.) using code**—similar to how software applications are developed.

What is IaC ?



WORKFLOW BASICS FOR
Infrastructure as Code
INFRASTRUCTURE CODE
write code
USER/DEVELOPER
pull
push
MANAGE
CLOUD
INFRASTRUCTURE
AUTOMATION
API / SERVER
MANAGE
ON PREM
INFRASTRUCTURE
VERSION CONTROL

IaC Approaches

A. Declarative (WHAT to provision)

You specify the desired final state. The tool figures out how to reach that state.

- **Terraform**
- **AWS CloudFormation**
- **Azure ARM/Bicep**
- **Kubernetes YAML**

B. Imperative (HOW to provision)

You define step-by-step instructions.

- **Ansible**
- **Puppet**
- **Chef**

IaC Workflow

- Write configuration code
- Validate & test
- Commit to version control (Git)
- Run provisioning (Terraform apply, ansible-playbook…)
- Cloud provider creates the resources
- CI/CD pipelines handle updates and deployments

Simple Example (Terraform)

```
resource "aws_instance" "myserver" {
  ami = "ami-067c21fb1979f2d1c"
  instance_type = "t2.micro"
}
```

Benefits of IaC

1. Speed & Automation

- Infrastructure can be deployed instantly using scripts instead of manual setup.

2. Consistency & Standardization

- Every environment (dev, test, prod) is identical—reduces configuration drift.

3. Version Control

- Infrastructure definitions stored in Git → rollback, history, collaboration.

Benefits of IaC

4. Scalability

- Automates spinning up multiple servers, clusters, networks, etc.

5. Cost Efficiency

- Auto-scaling and automated teardown avoid paying for unused resources.

6. Reusability

- Modules and templates help avoid repeating configurations.

10 – Continuous Integration & Continuous Deployment (CI/CD)

CI/CD

- **CI/CD** is a DevOps practice that automates software development, testing, and deployment.

- **Continuous Integration (CI):**
- Developers frequently merge code changes into a shared repository.
- Automated tests run to ensure code quality.

- **Continuous Deployment / Delivery (CD):**
- Automatically deploy code to production (CD) or staging (Delivery) after passing all tests.

CI/CD

Benefits:

- Faster releases
- Fewer bugs
- High automation
- Reliable and consistent deployments

Examples of CI/CD Tools:

- Jenkins
- GitLab CI
- GitHub Actions
- Azure DevOps
- CircleCI

CI/CD

Typical CI/CD Pipeline Workflow:

1. Developer pushes code
2. CI triggers: build + tests
3. Security scans & code quality checks
4. Create container image
5. Push image to registry
6. Deploy to Kubernetes (or VM/server)
7. Auto tests in staging
8. Manual/automatic promotion to prod
9. Monitoring and rollback if needed