CLOUD COMPUTING
(Week-2)

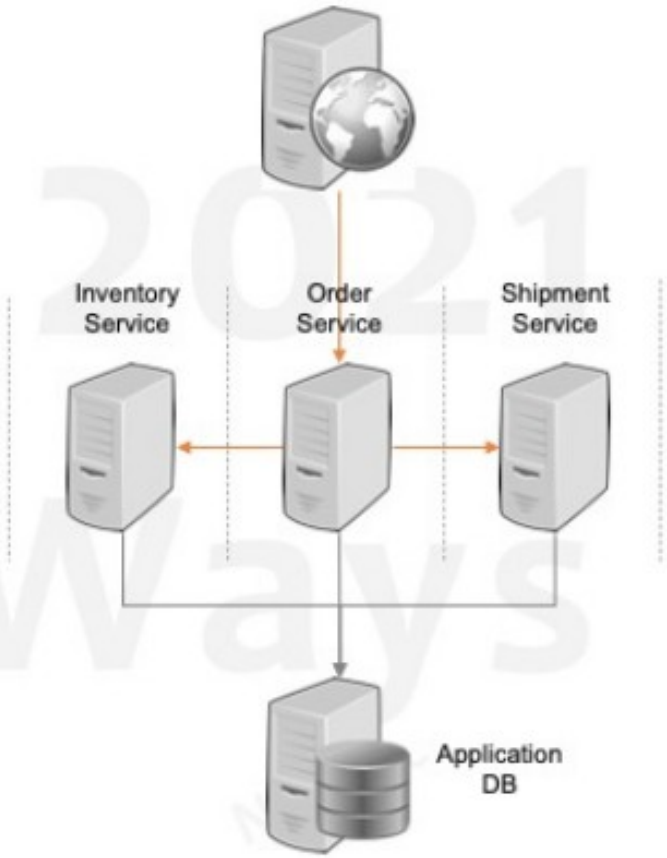
USAMA MUSHARAF

LECTURER (Department of Software Engineering)

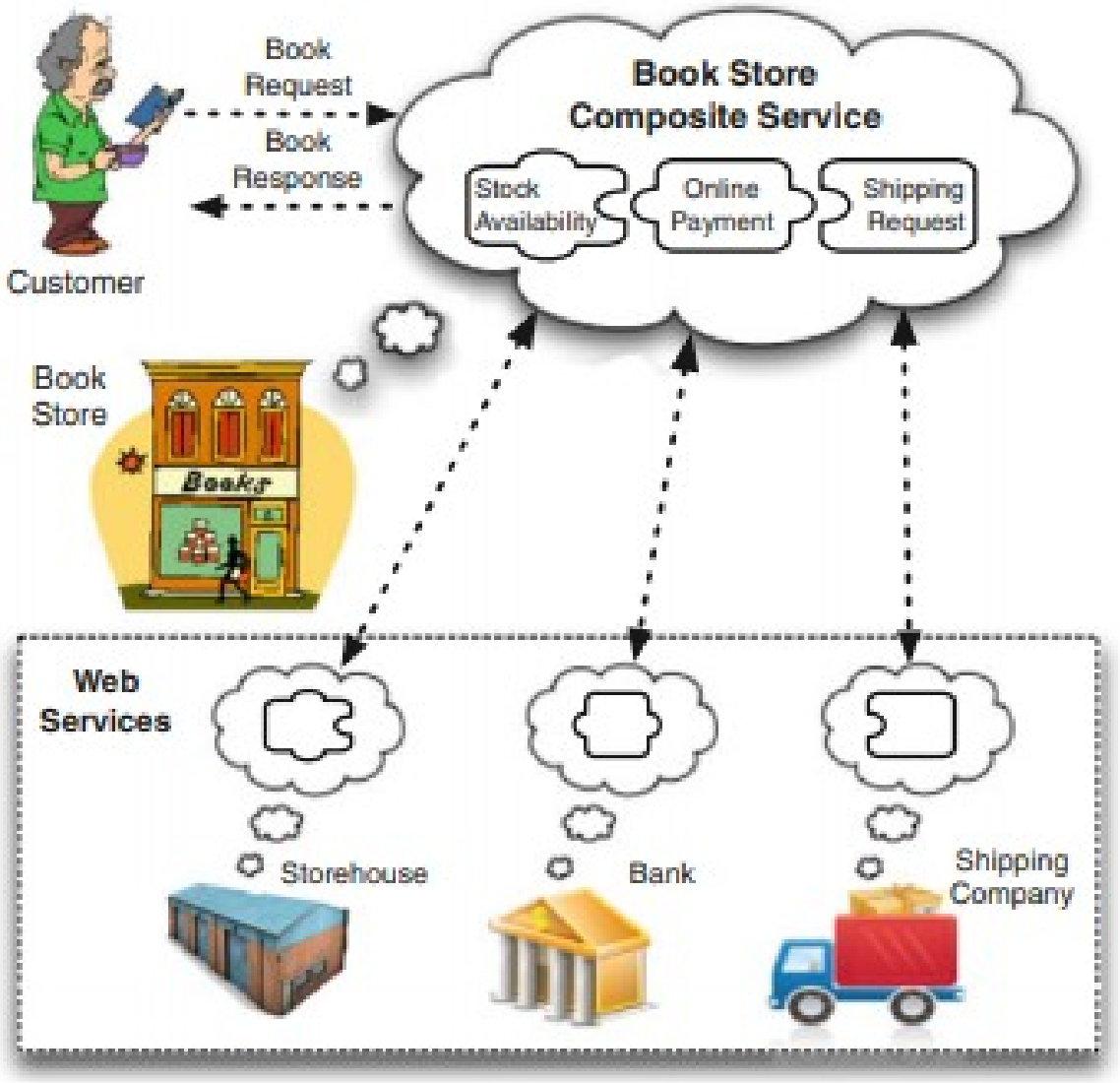
FAST-NUCES PESHAWAR

Service Oriented Architecture

- Independent
  - Each service can have its own technology stack, libraries, frameworks etc.
  - Each service can be scaled independently and differently
- Not Independent
  - Common interface schema
    - XML schema
  - Common database schema
    - RDBMS schema
- Issues
  - Service development may be independent but not deployment
  - Single database has scalability limitations



Service Oriented Architecture

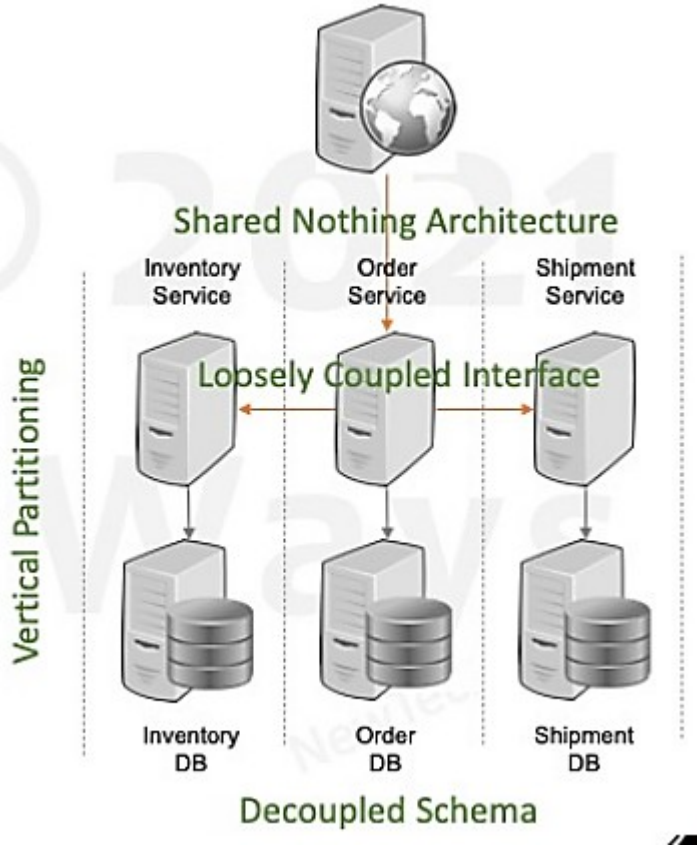


An example of Web service composition

MICROSERVICES

MICROSERVICES

- Microservices are small, individually deployable services performing different operations.
- Variant of SOA



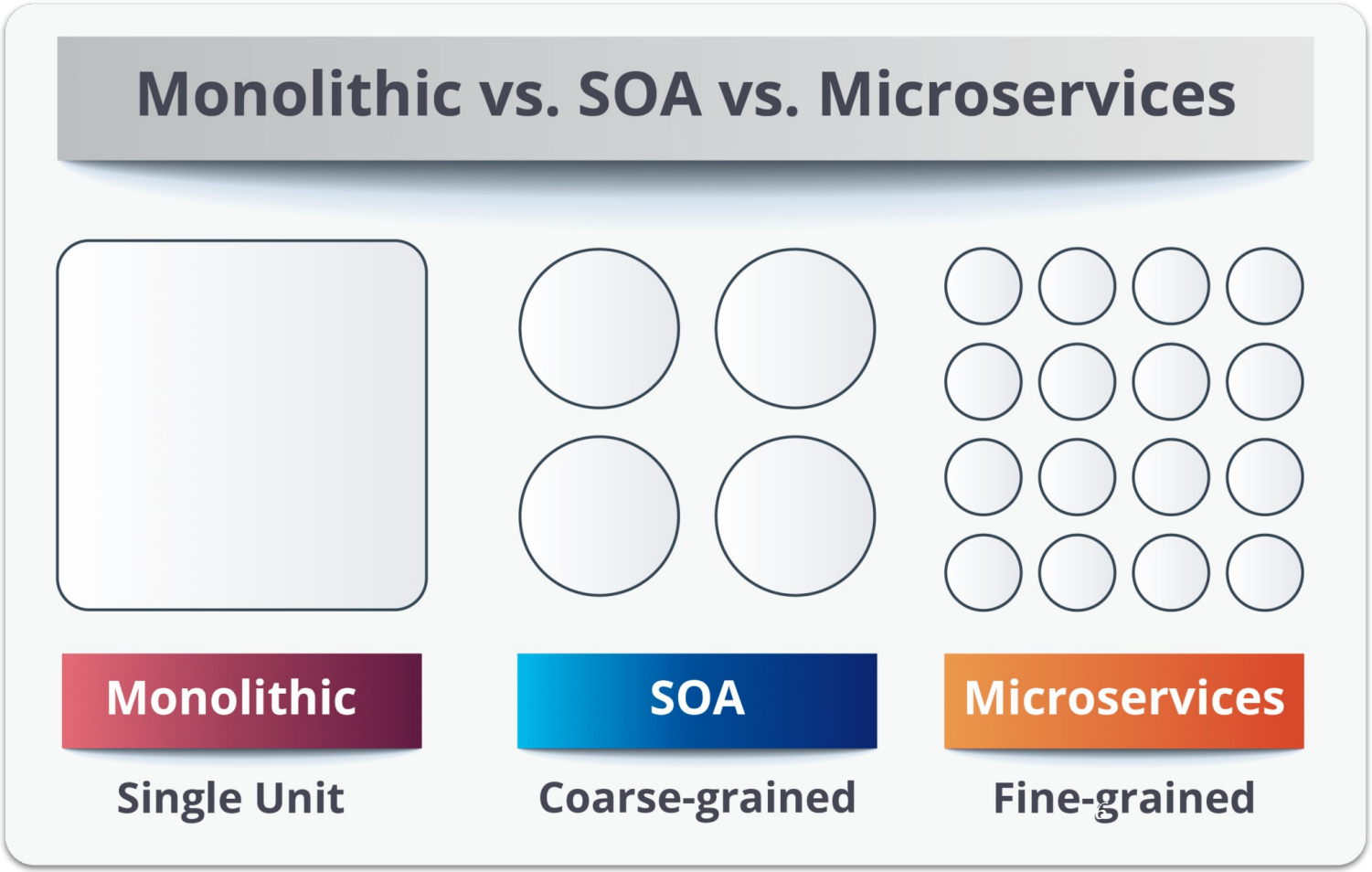
Frequent Deployment

Independent Deployment

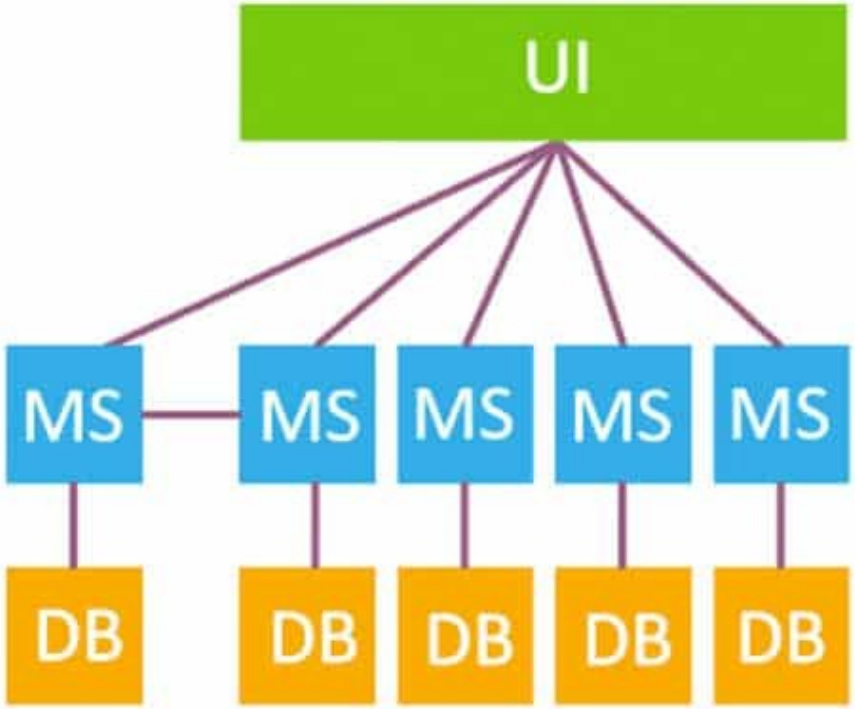
Independent Development

Independent Services

SOA VS MICROSERVICES



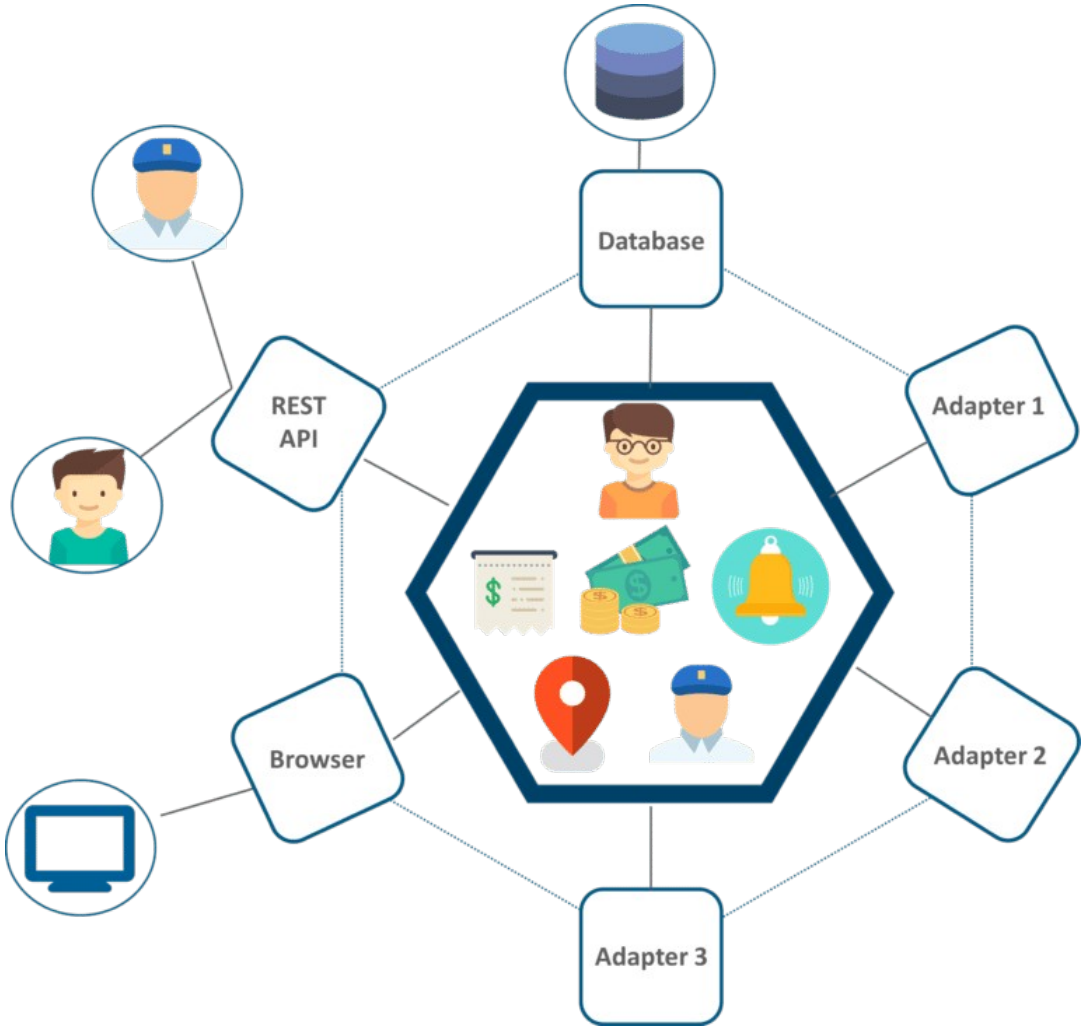
SOA VS MICROSERVICES



UBER’S PREVIOUS ARCHITECTURE

- A REST API is present with which the passenger and driver connect.
- Three different adapters are used with API within them, to perform actions such as billing, payments, sending emails/messages that we see when we book a cab.
- A MySQL database to store all their data.

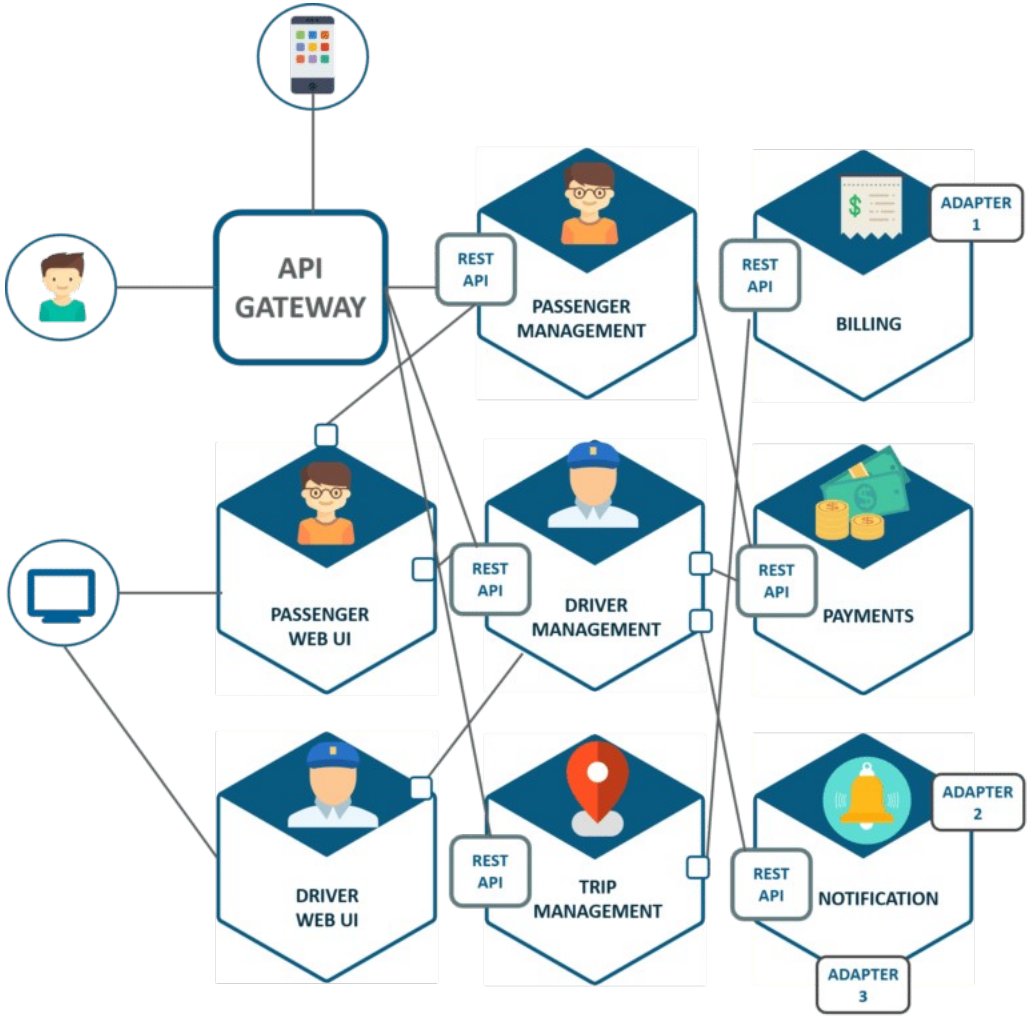
All the features such as passenger management, billing, notification features, payments, trip management, and driver management were composed within a single framework.



PROBLEM IN UBER'S ARCHITECTURE

- All the features had to be re-built, deployed and tested again and again to update a single feature.
- Fixing bugs became extremely difficult in a single repository as developers had to change the code again and again.
- Scaling the features simultaneously with the introduction of new features was quite tough to be handled together.

SOLUTION IS MICROSERVICES ARCHITECTURE



SOLUTION

- The units are individual separate deployable units performing separate functionalities.
- For Example: If you want to change anything in the billing Microservices, then you just have to deploy only billing Microservices and don’t have to deploy the others.
- All the features were now scaled individually i.e. The interdependency between each and every feature was removed.

DECOMPOSITION OF MICROSERVICES

There are some Prerequisite of decomposition of microservices.

Services must be cohesive.

- A service should implement a small set of strongly related functions.

Services must be loosely coupled

- Each service as an API that encapsulates its implementation.

MICROSERVICES WITH DOCKERS AND KUBERNETES

CHARACTERISTICS OF MICROSERVICES

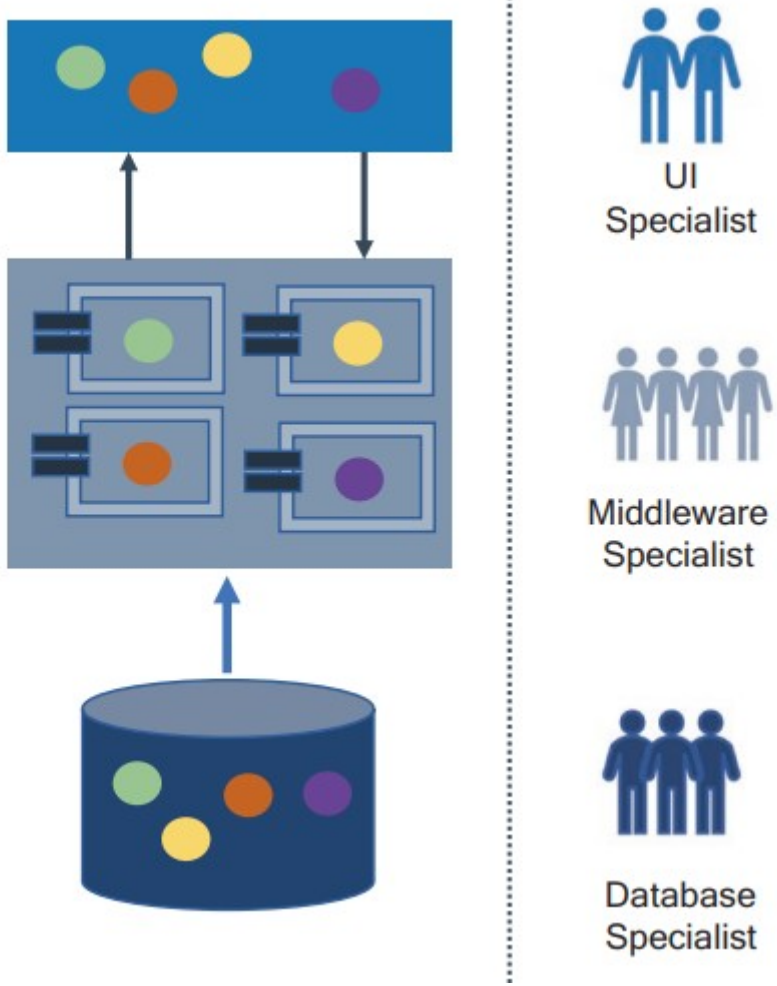
ORGANIZED AROUND BUSINESS CAPABILITIES

Before you start designing any microservices application, identifying and defining each microservice is important.

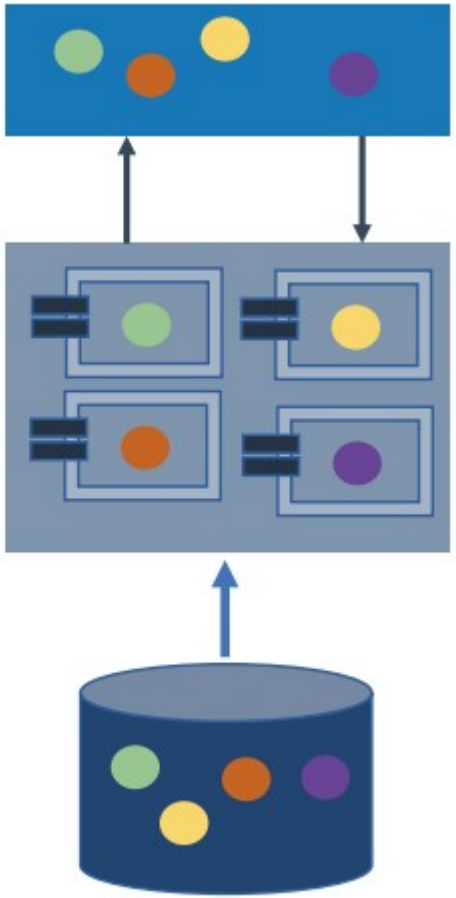
What is the boundary of the microservice?
What domain should it contain?
What are the events ?
Where will my microservices be deployed?

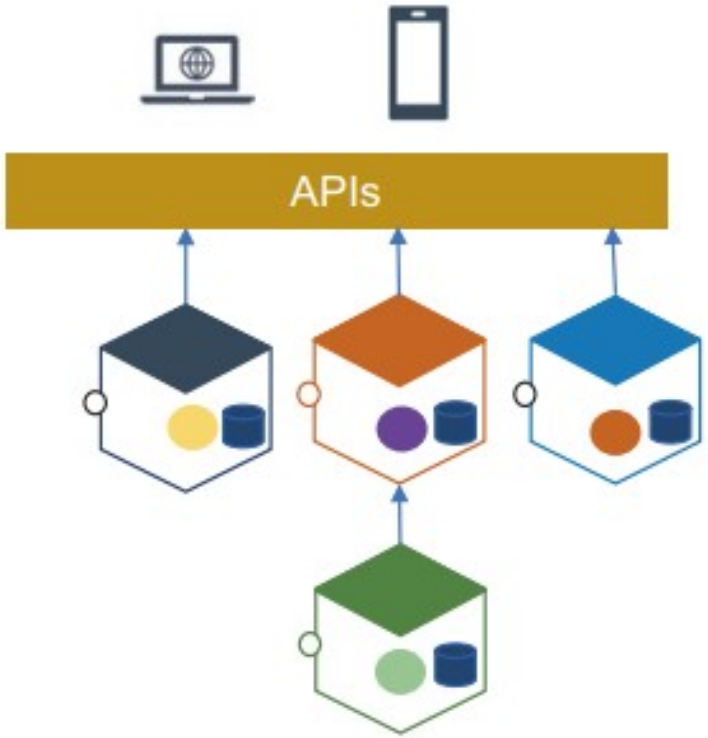
TRADITIONAL APPROACH VS ORGANIZED AROUND BUSINESS CAPABILITIES

In a traditional application development, as shown in Figure we designed around the *technological capabilities* such as the *user interface, databases, business logic,* etc., but we never had any discussion about the domain or contract or boundary.



TRADITIONAL APPROACH VS ORGANIZED AROUND BUSINESS CAPABILITIES





Organized around capabilities

AUTONOMOUS

Microservices are a self-contained unit of functionality with loosely coupled dependencies across other services.

The services are separate entities and deployed in an isolated container environment.

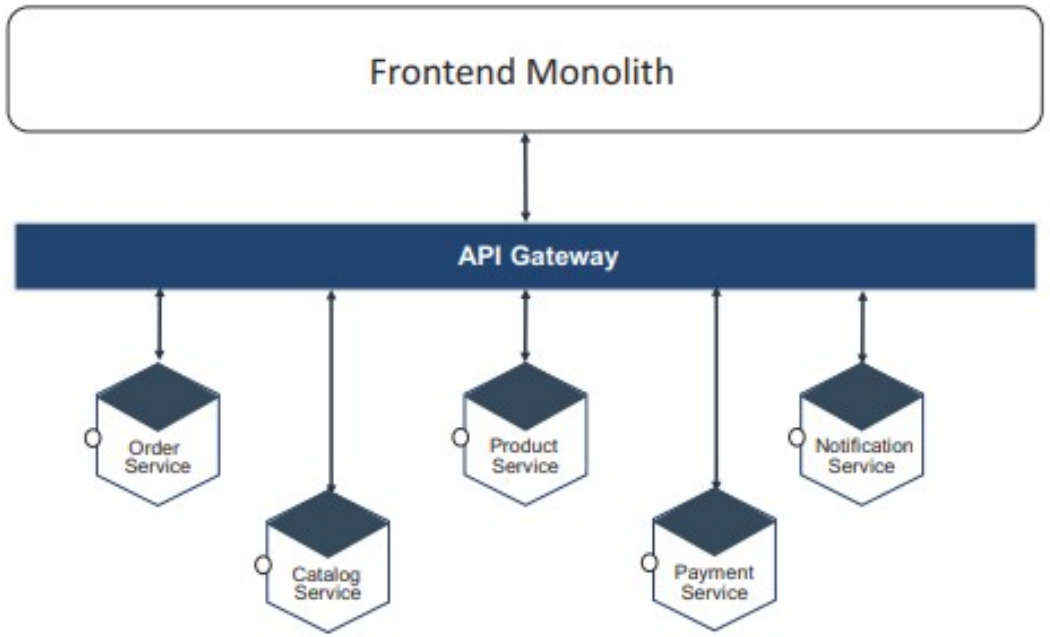
SMART END AND DUMB PIPES

The smart endpoints and dumb pipes simplify communication across microservices.

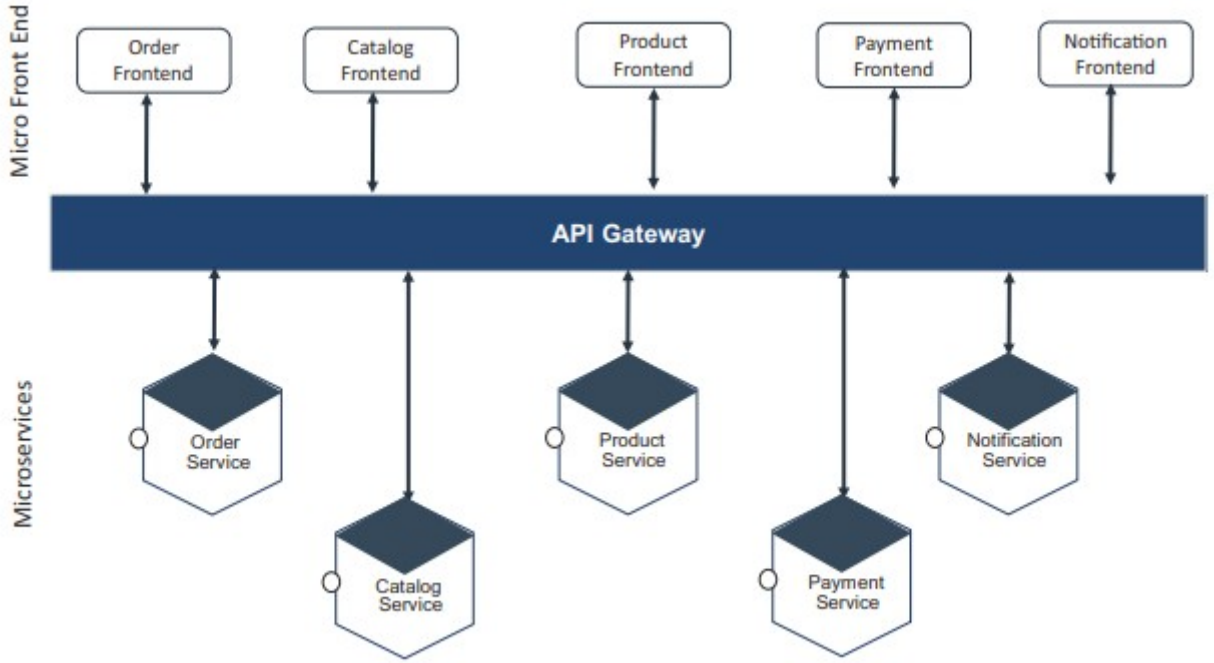
In microservices communication, we use two types of protocols: synchronous and asynchronous with request/response and publish/subscribe, respectively.

Endpoints are applications and the pipes are what connect them and allow them to communicate with each other.

MICROSERVICES AND USER INTERFACE: MICRO FRONT END



5-20. *Front-end monolithic with microservices*



EVENT DRIVEN ARCHITECTURE

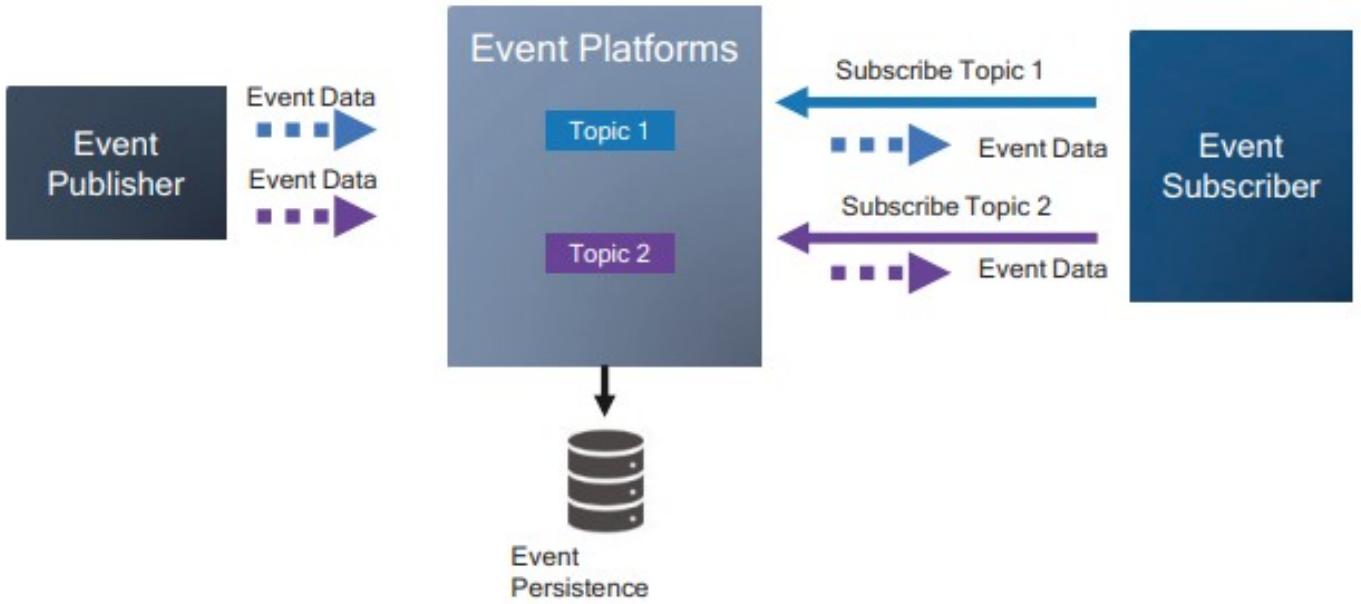
EVENT DRIVEN ARCHITECTURE

An event-driven architecture (EDA) is a distributed, asynchronous architecture that integrates applications and components through events.

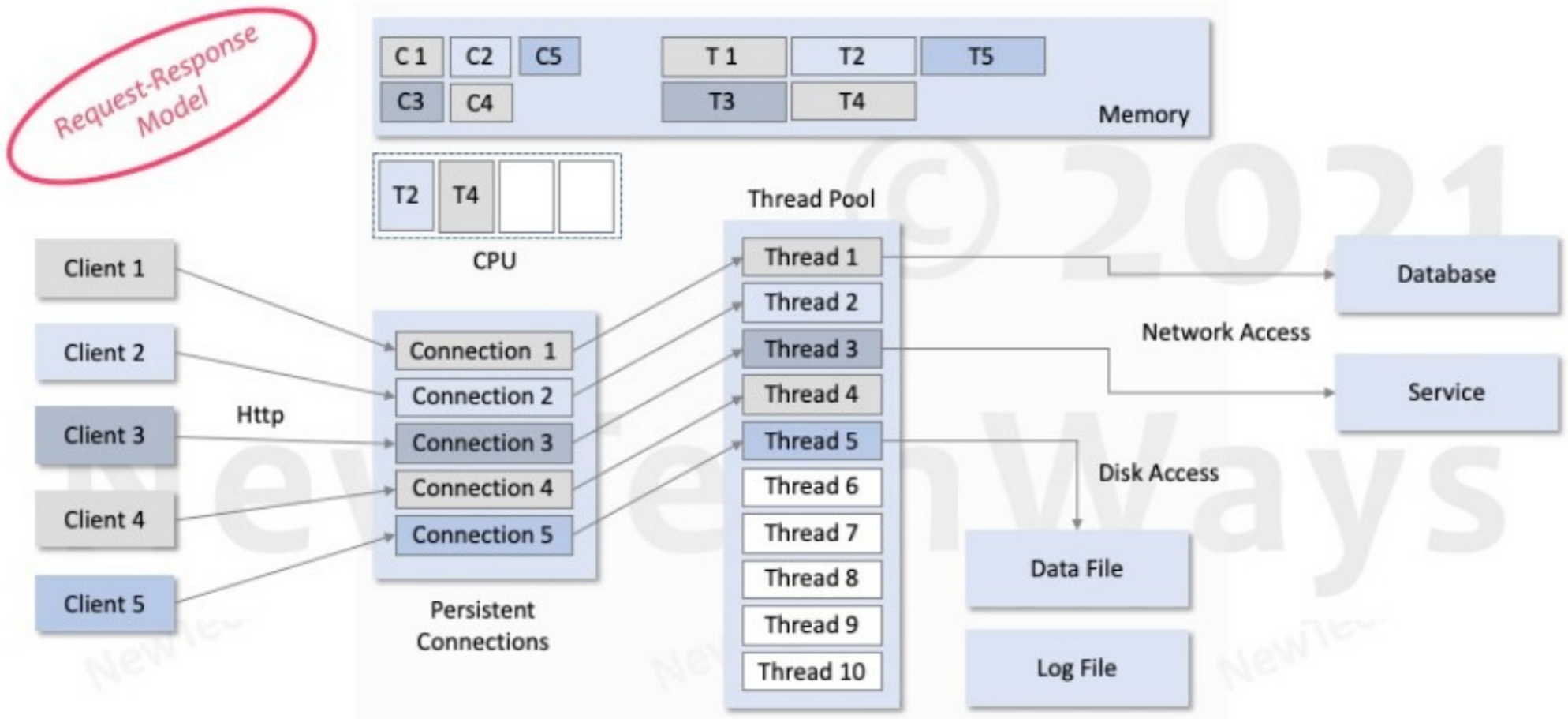
EVENT DRIVEN ARCHITECTURE

How Does Event-Driven Architecture Work?

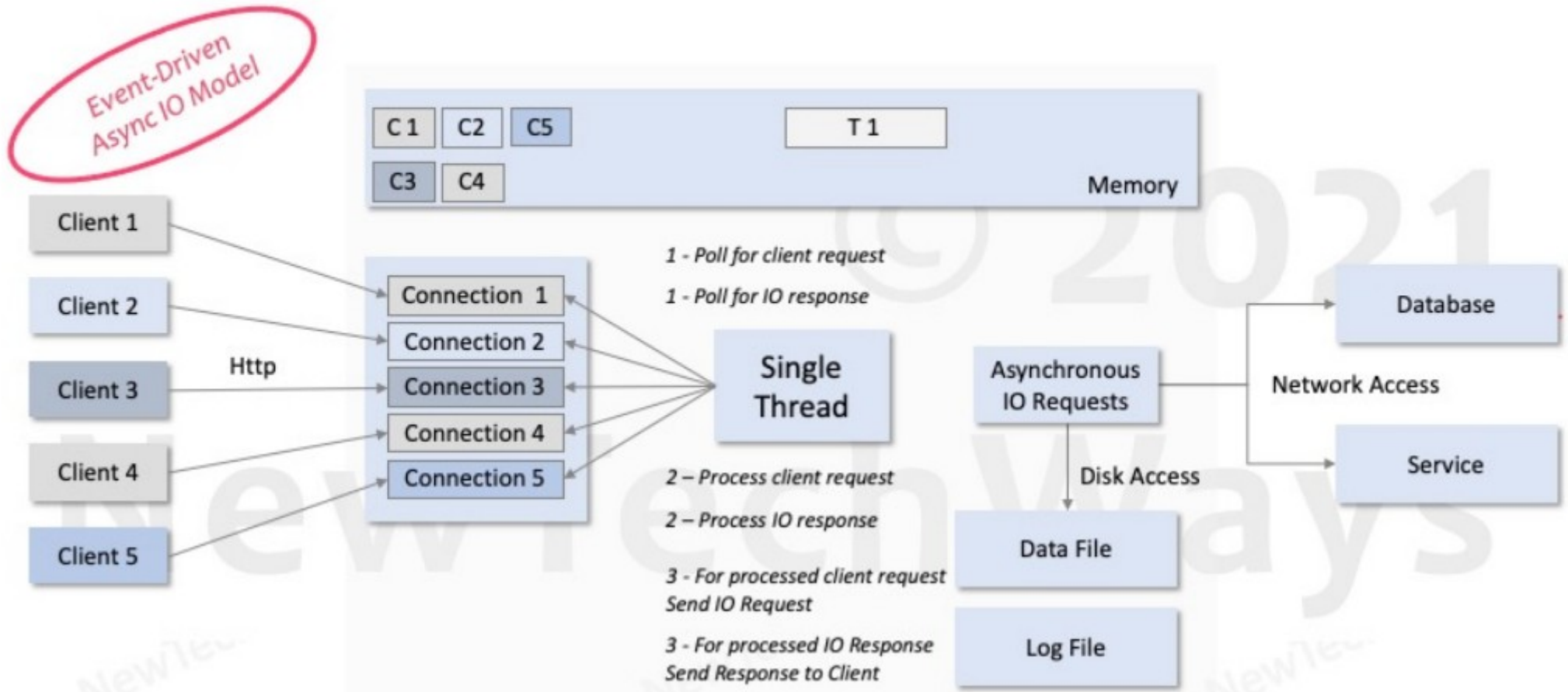
- Event producer/publisher
- Event consumer/subscriber
- Event broker or routers
- Event persistence (part o platform)



Apache Webserver Architecture



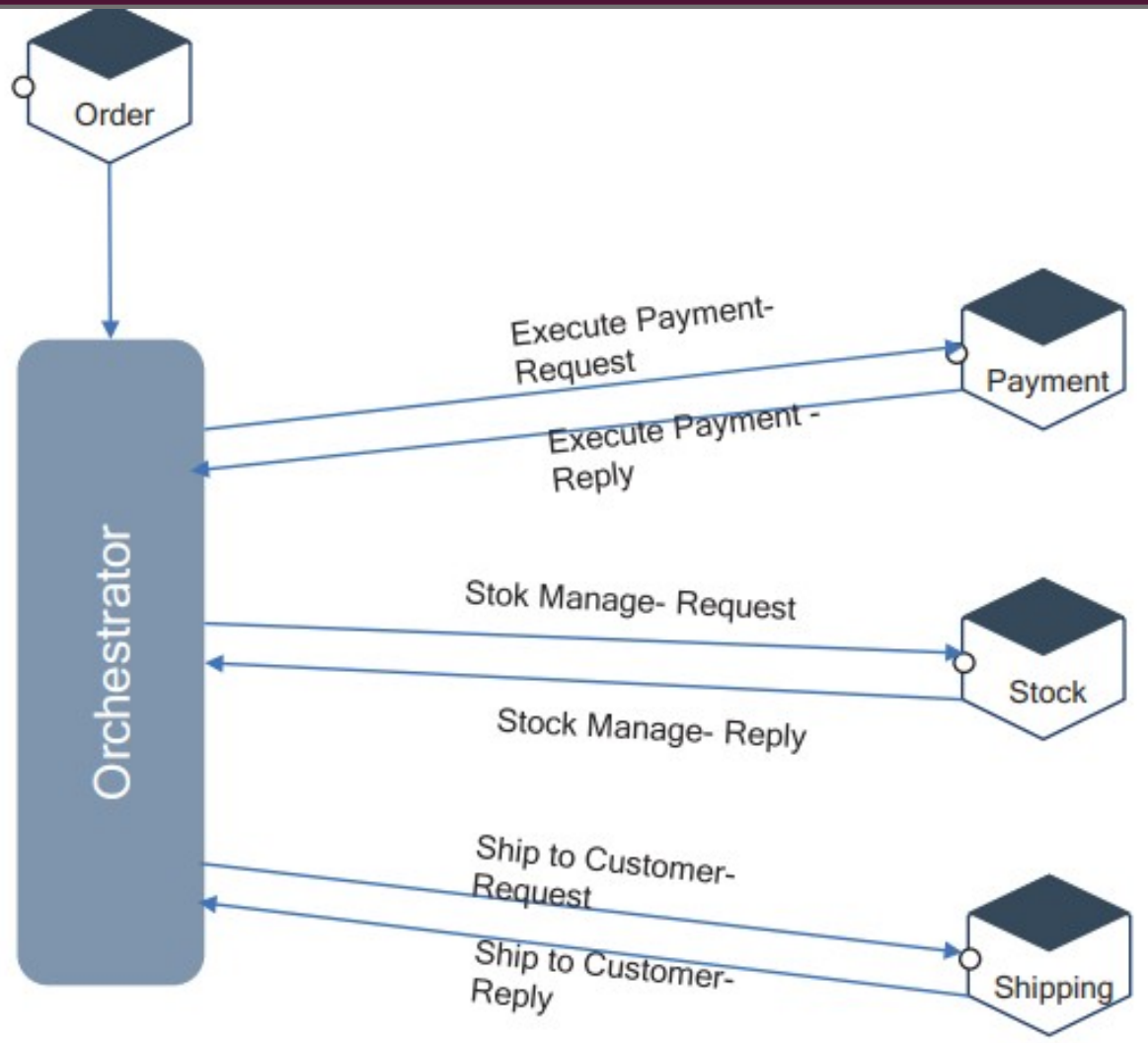
Nginx Architecture



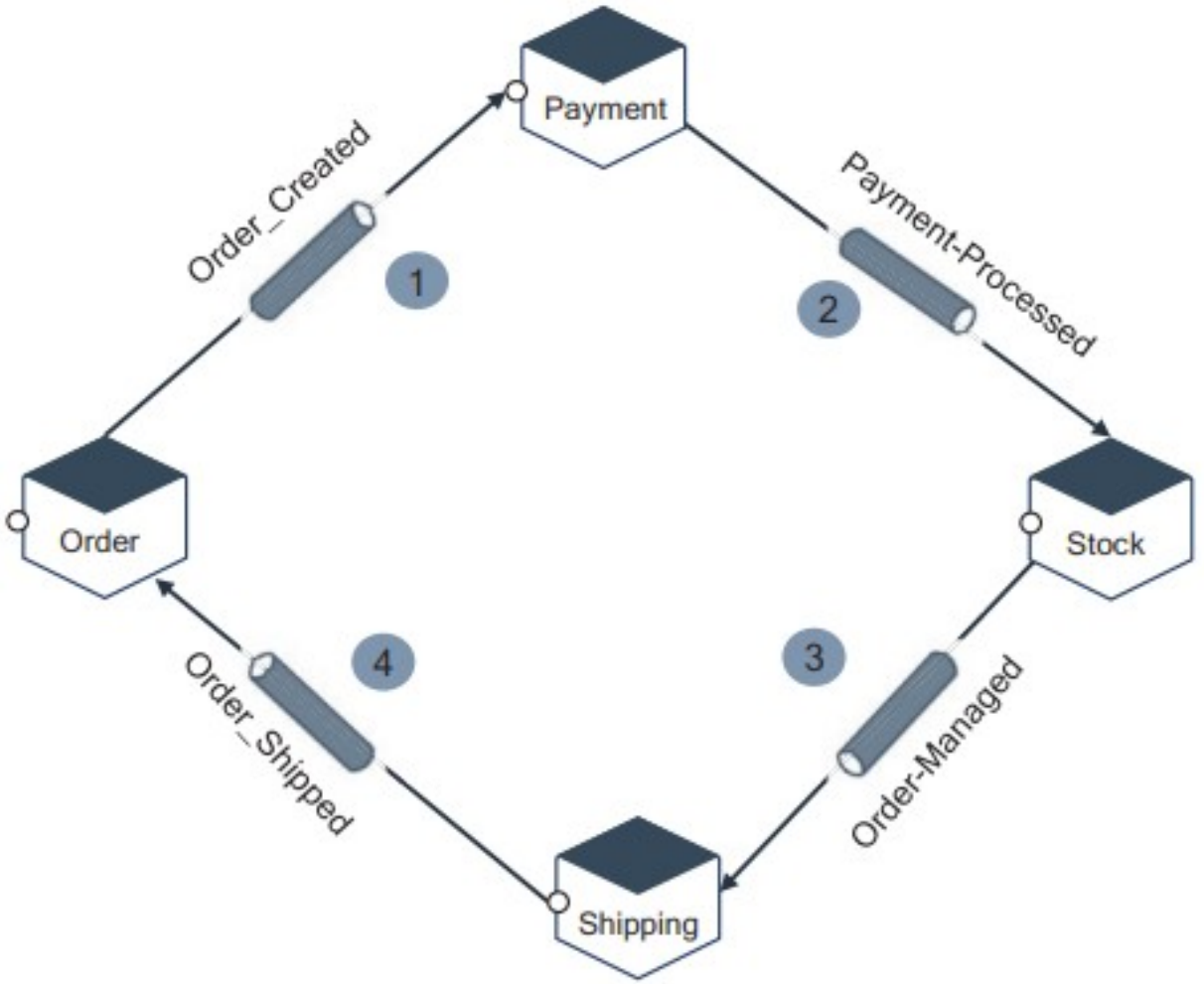
EVENT-DRIVEN MICROSERVICES INTERACTION

- Orchestration with synchronous
- Choreography with asynchronous

Orchestration with synchronous



Choreography with asynchronous



CHARACTERISTICS OF EVENT DRIVEN ARCHITECTURE

Multicast communication:

The events are generated from the publishing systems, and event-driven systems can send these events to multiple event processors.

Real-time transmission:

The events are processed in real time to the event processors. The mode of processing or transmission is real time rather than batch processing.

Asynchronous communication:

The event does not need to wait for the event processor to be available before publishing an event.

Loose Coupling / Decoupling of Producers and Consumers

In event-driven architecture, producers and consumers of events are decoupled.

SERVERLESS ARCHITECTURE

What is Serverless?

a cloud-native platform
for

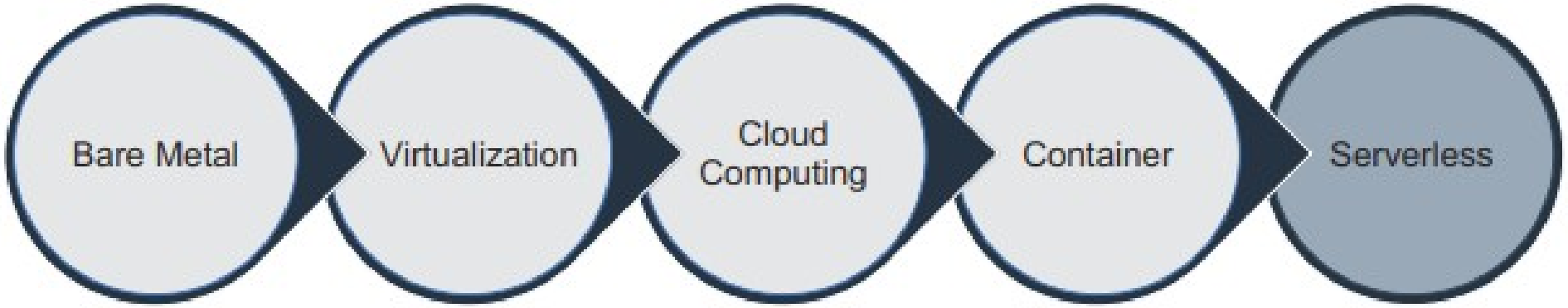
short-running, stateless computation
and

event-driven applications
which

scales up and down instantly and automatically
And

charges for actual usage at a millisecond granularity

EVOLUTION OF SERVERLESS



SERVER-LESS MEANS NO SERVERS?
OR WORRY-LESS ABOUT SERVERS?

Runs code **only** on-demand on a per-request basis

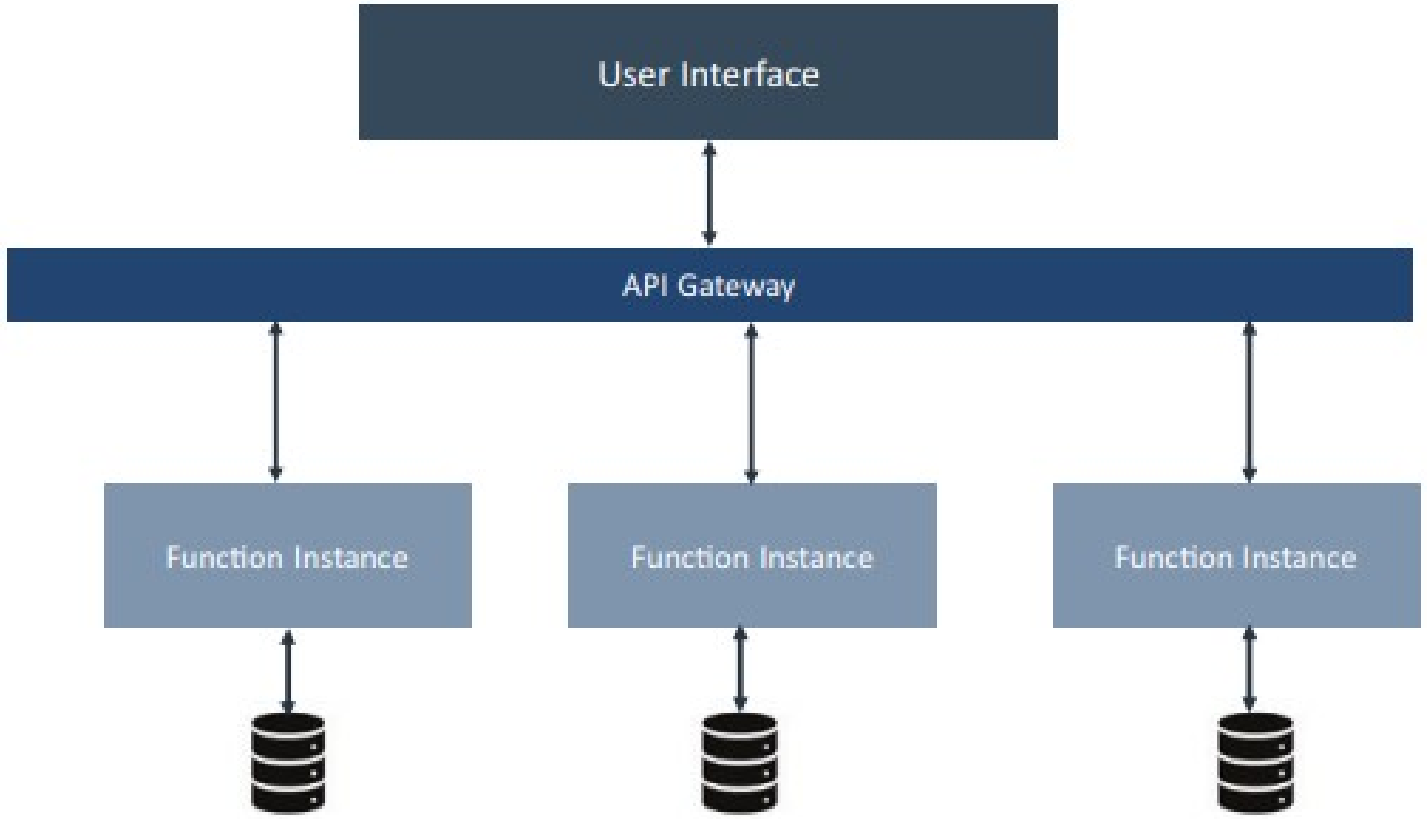
Serverless deployment & operations model

No servers

Just code

SERVERLESS COMPUTING ARCHITECTURE

Serverless almost behaves the same as microservices, but you do not need to worry about any runtime environment or deployment environment.



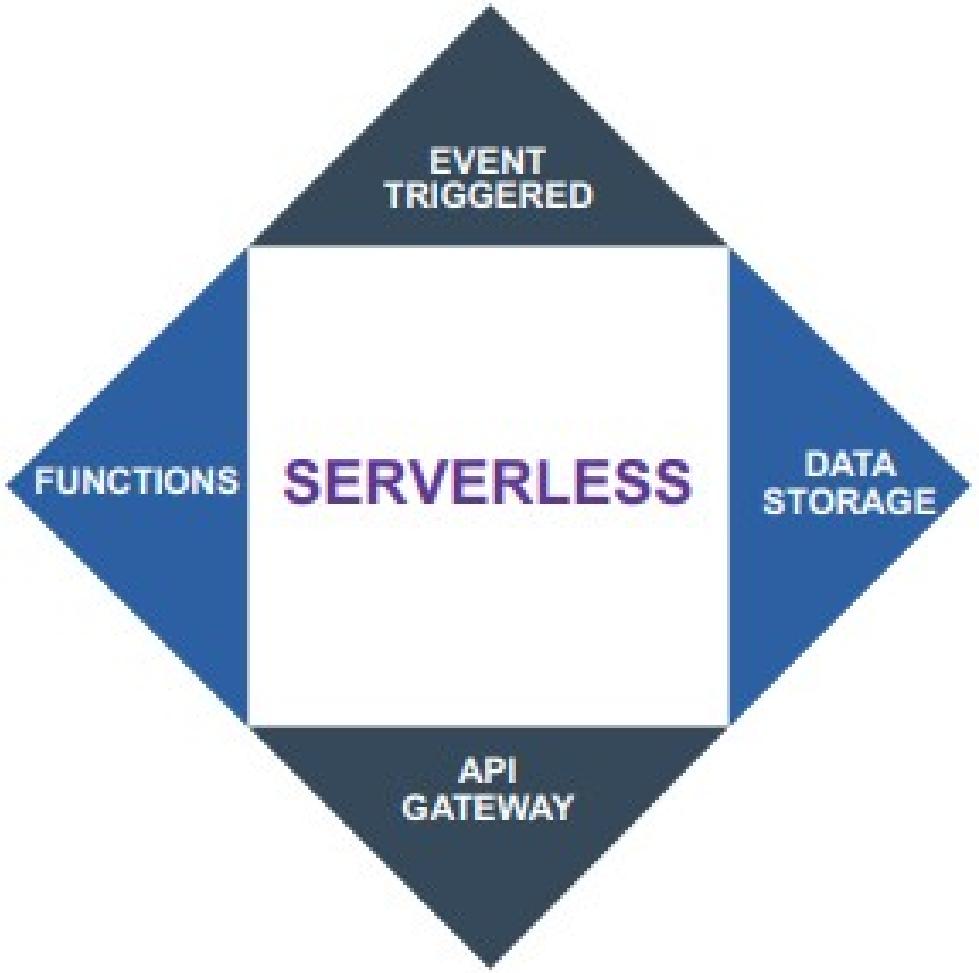
ESSENTIAL COMPONENTS OF SERVERLESS COMPUTING

Event Triggered
The applications provide a constant stream of events to which functions subscribe and act upon.

Data Storage
Storage ranges from simple disk storage to highly scalable data stores.

Functions
Functions are the actual business logic to process an event.

API Gateway
A light API gateway provided by the cloud platform may be in the form of streaming or REST calls to support endpoints for mobile devices.



TOWARDS CLOUD NATIVE ARCHITECTURE PRINCIPLES

ARCHITECTURE PRINCIPLES

A principle is a law or rule that is usually followed when making key architecture decisions.

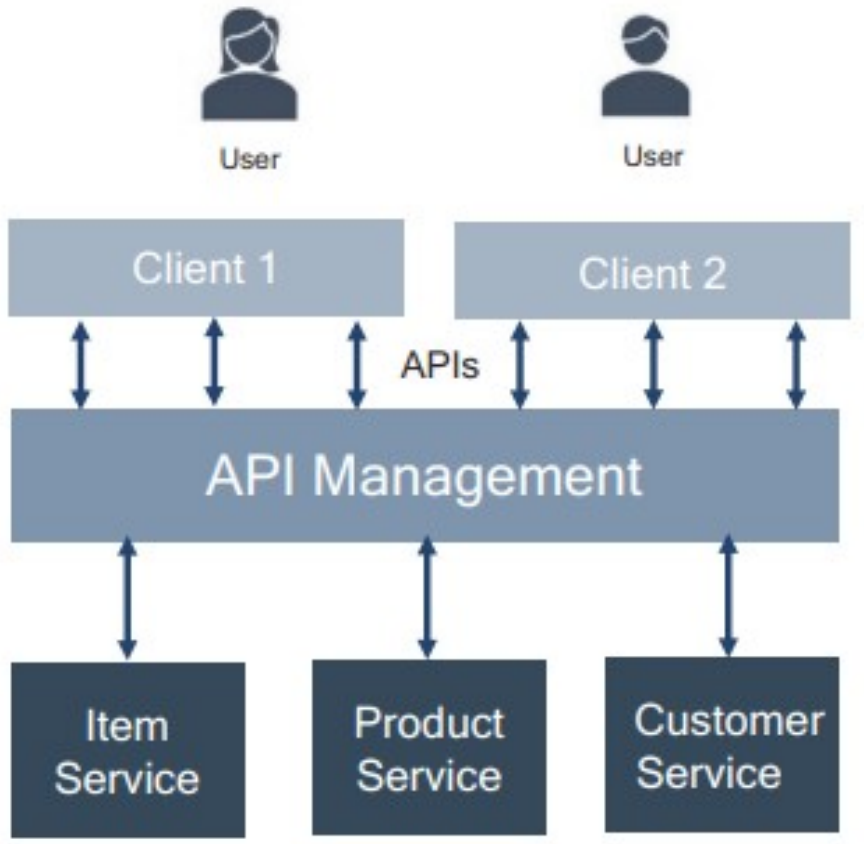
It’s important to note that principles are not commandments; exceptions are acceptable when necessary.

API FIRST PRINCIPLE

The API first principle is the de facto principle of modern architecture.

Every application is designed and developed with the API first principle.

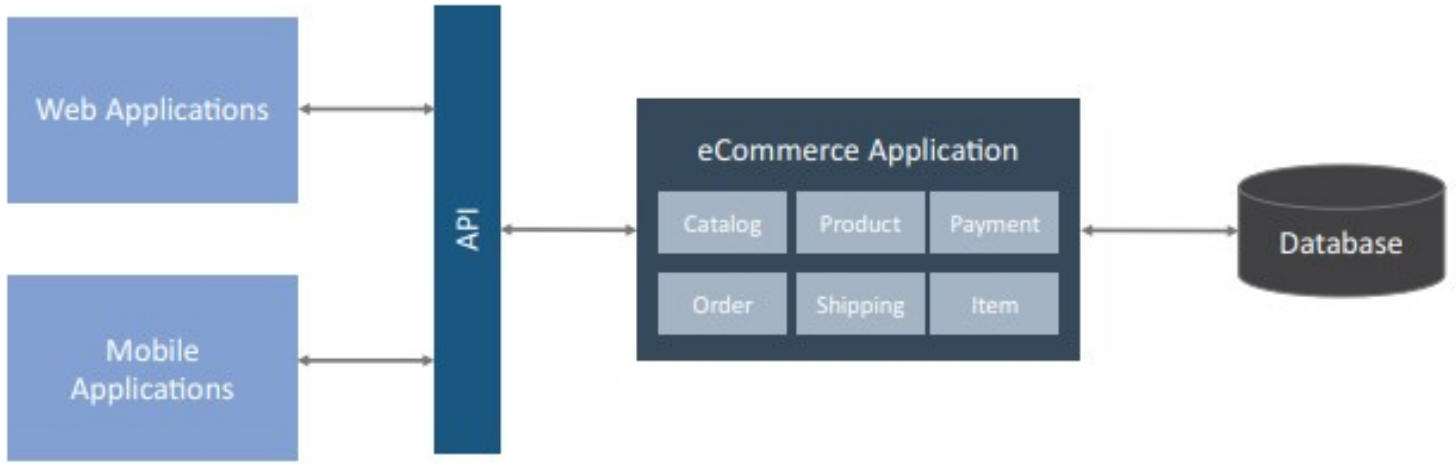
This principle allows all implementation details to be exposed through APIs to the consumers and encourages the application design and development teams to have resources accessible through REST HTTP interfaces.



MONOLITHIC ARCHITECTURE PRINCIPLE

The monolithic architecture principle (MAP) is building the architecture as a single unit with a single codebase.

The monolithic architecture can expose APIs to the client applications and also focus on desktop/laptop devices with a web browser as a client, as shown in Figure given below.

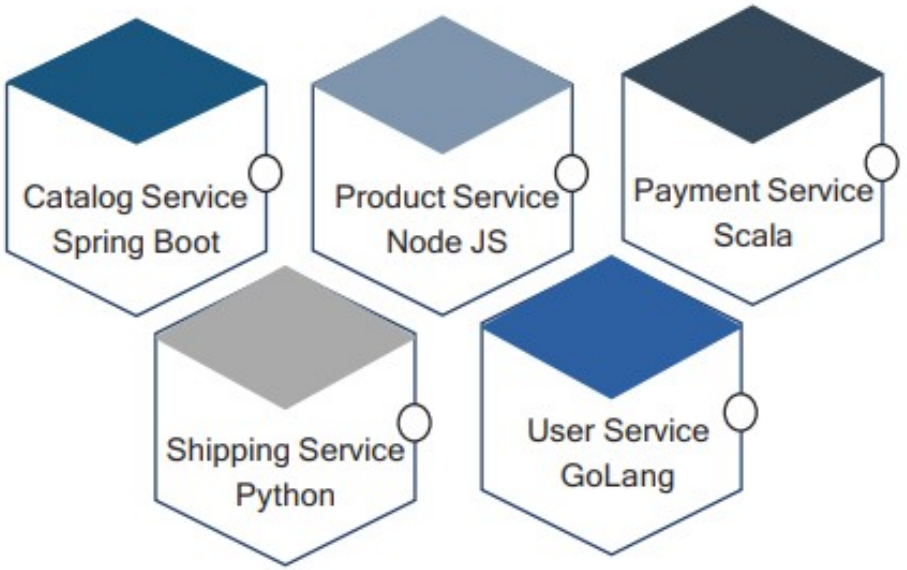


POLYLITHIC ARCHITECTURE PRINCIPLE

The polylithic architecture principle (PAP) provides a different variant of microservices.

Each microservice provides domain functionality.

These separated modules are consolidated through several programming techniques



POLYGLOT PERSISTENCE PRINCIPLE

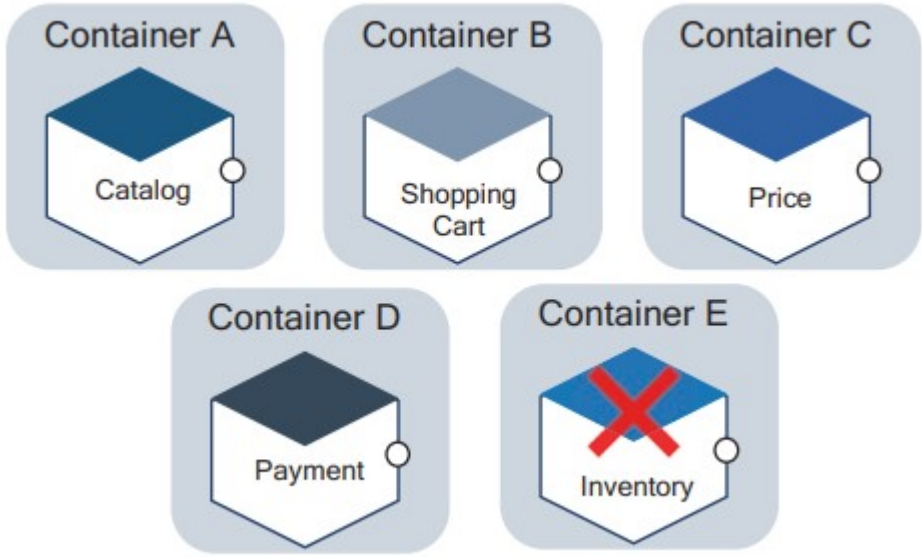
The polyglot persistence principle is about is choosing the way data is stored based on the way data is being used by individual applications. In short, you need to pick the right storage for the right kind of data.

CLOUD NATIVE RUNTIME PRINCIPLES

ISOLATE FAILURE PRINCIPLE (IFP)

Embracing a cloud native architecture doesn’t automatically make your system more stable.

Microservices are not reliable by default; therefore, you can’t assume that your microservices become more resilient or scalable by default.



DEPLOY INDEPENDENTLY PRINCIPLE

The deploy independently principle (DIP) says that every service should be deployed independently in an infrastructure as a service (IaaS) by using containers and Kubernetes.

When you bundle more services into a single machine, you limit your ability to change things independently; this is the reason why you should deploy one service per container.

What happens if one service fails?

You need to forcefully stop other services also.

BE SMART WITH STATE PRINCIPLE

Storing the state is the hardest aspect of architecting a distributed, cloud native architecture.

Therefore, architect your system as stateless wherever it is possible.

Stateless means that any state must be stored outside of a container, and this external state can be stored in various storage.

By storing data externally, you remove data from the container itself, meaning that the container can be cleanly shut down and destroyed at any time without fear of data loss.

If a new container is created to replace the old one, you just connect the new container to the same datastore or bind it to the same disk.

ADVANTAGES OF STATELESS COMPONENT

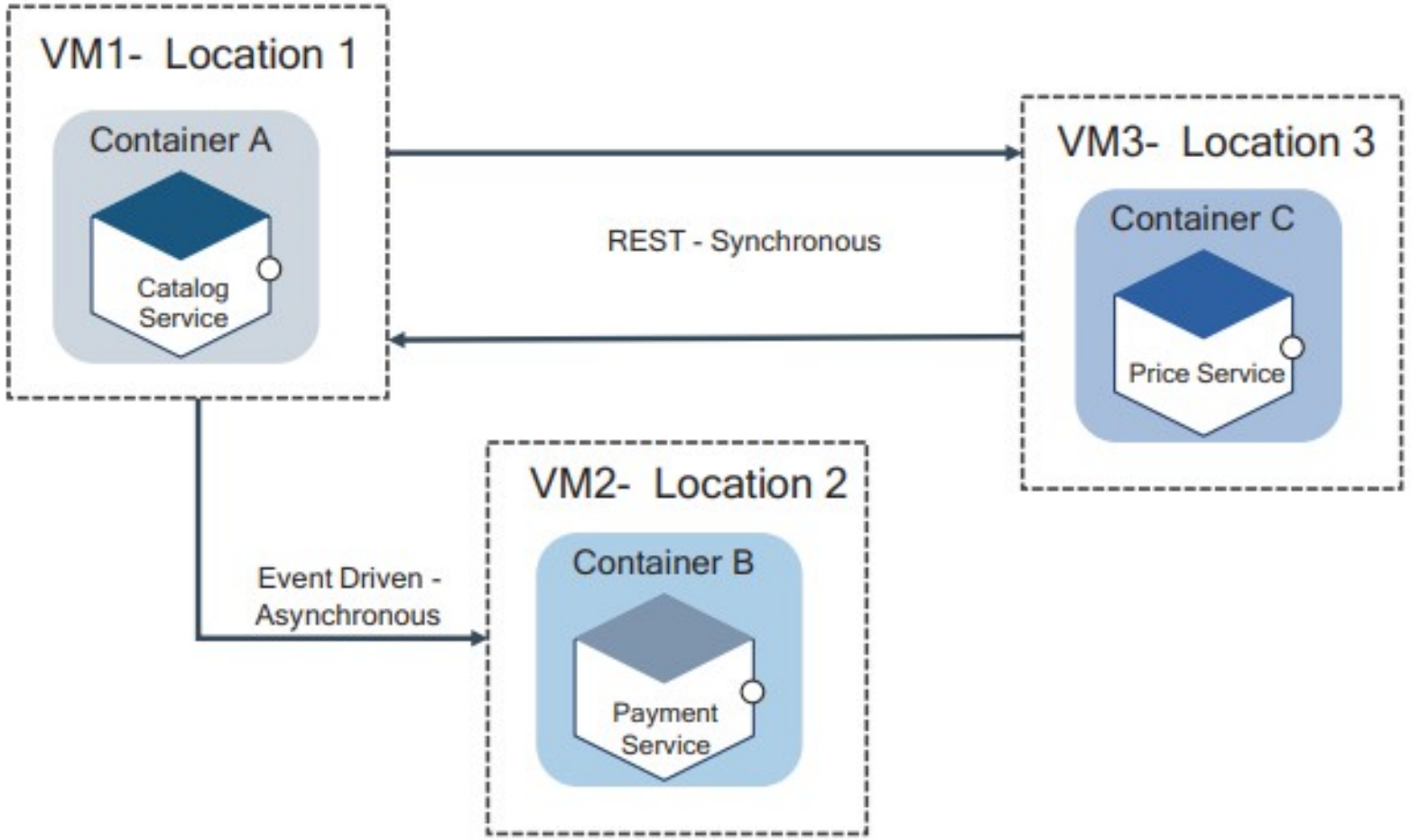
Easy to destroy and easy to create.

Easy to repair: If you want to repair failed instances in your deployment, simply terminate gracefully and spin up a replacement.

Auto scale/horizontal scale: To scale more instances, just add more copies; the orchestrator can manage the scale-up and down. This scale can be managed automatically based on load or CPU usage.

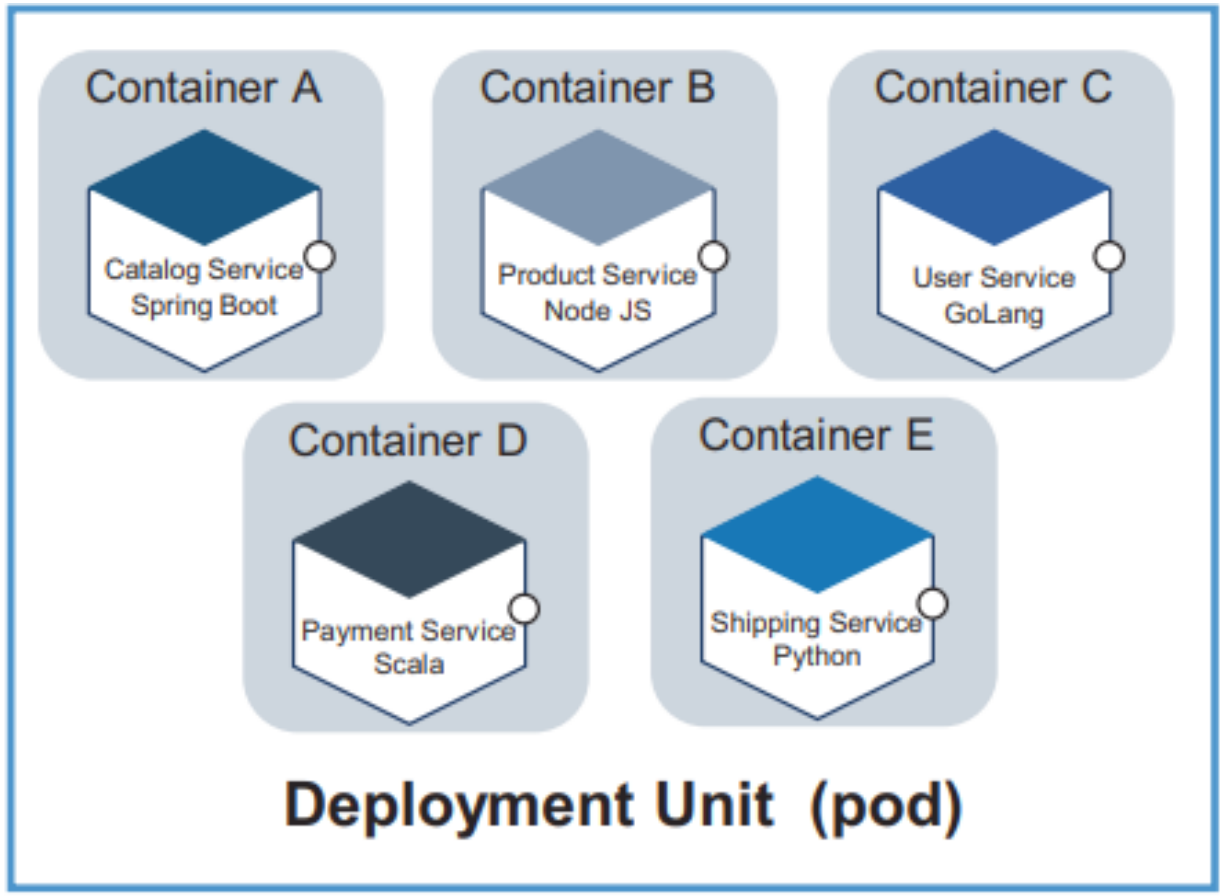
Rollback: If you have a wrong deployment, the stateless containers are much easier to replace with new ones without any human intervention.

LOCATION-INDEPENDENT PRINCIPLE

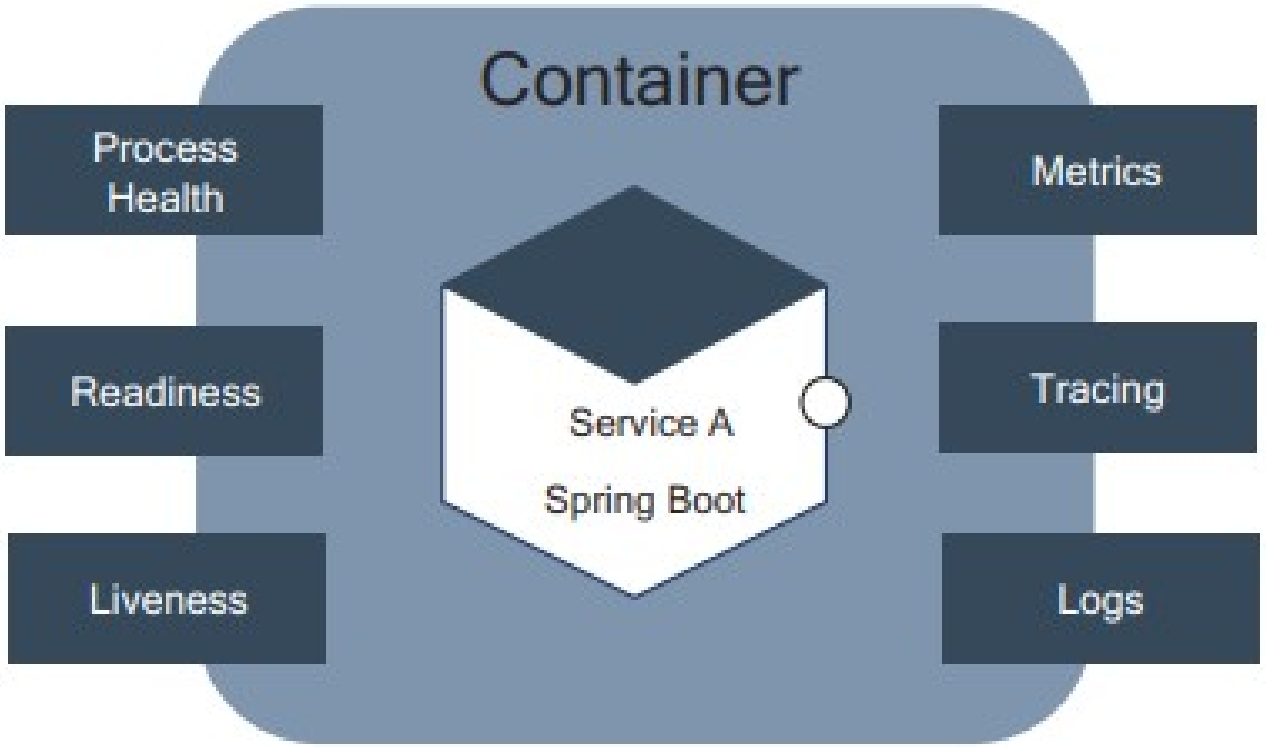


CONTAINER PRINCIPLES

SINGLE CONCERN PRINCIPLE



HIGH OBSERVABILITY PRINCIPLE



Observability in cloud native application

IMAGE IMMUTABILITY PRINCIPLE (IIP)

IIP means an image is unchangeable once it is built and requires creating a new image if changes need to be made.

For each image change, you need to build a new image and reuse it across various environments in your development lifecycle.

PROCESS DISPOSABILITY PRINCIPLE (PDP)

PDP is a container runtime principle and states applications must be ephemeral as possible and ready to be replaced with container instances at any point of time by using infrastructure as code.

SELF-CONTAINMENT PRINCIPLE (SCP)

SCP addresses the build-time concern, and the objective of this principle is that the container must contain everything that it needs at build time.

CLOUD NATIVE API MANAGEMENT PATTERNS

SERVICE DISCOVERY

The API gateway needs to know the location (IP address and port) for each microservice with which it communicates.

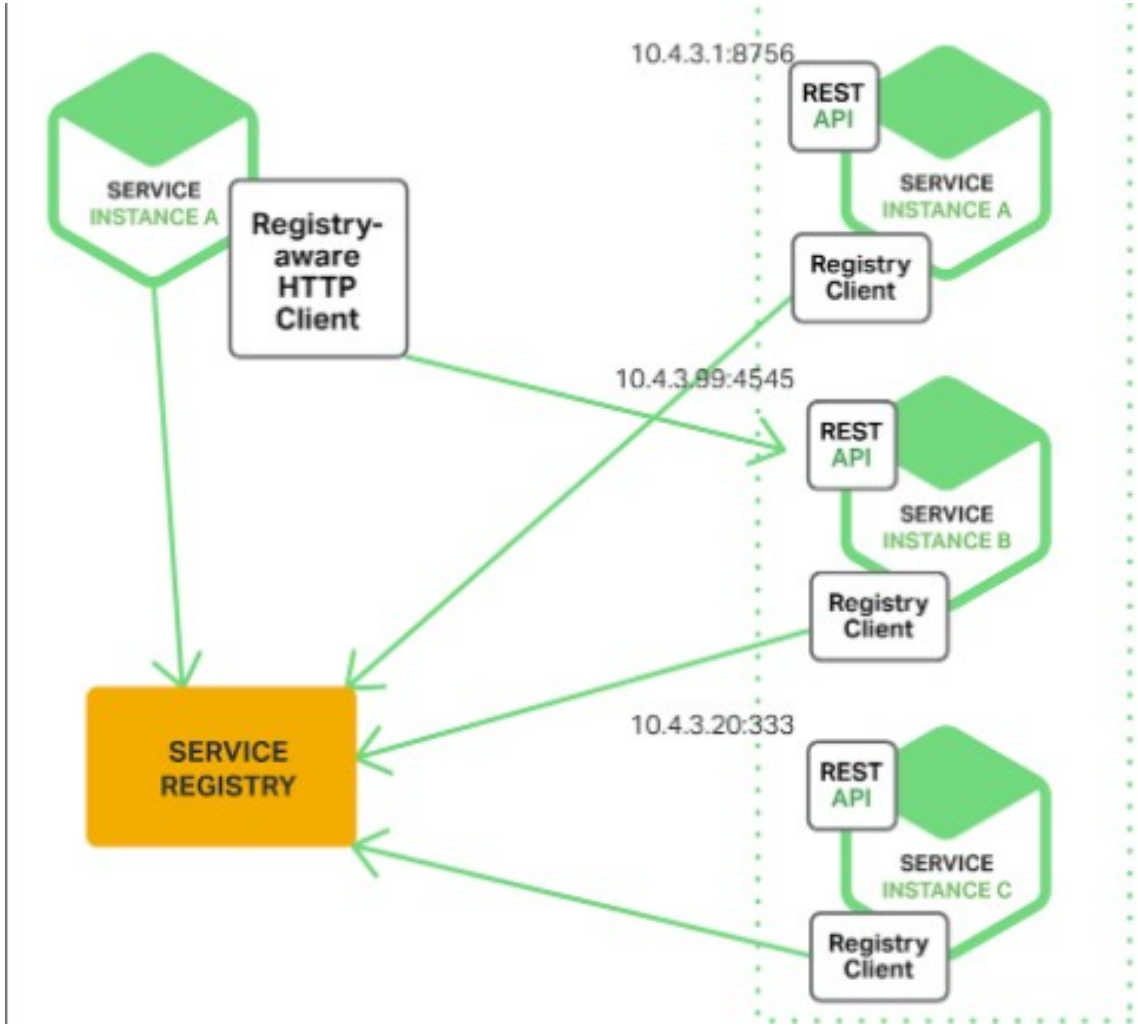
In a traditional architecture and system, you could probably hardwire the location because this application is not dynamic.

SERVICE DISCOVERY

Application services are assigned a location and set of instances of service changes dynamically because of autoscaling, container orchestration, etc.

Consequently, the API gateway needs to use the system’s service discovery mechanism either in server-side discovery or in client-side discovery.

CLIENT-SIDE DISCOVERY PATTERN



CLIENT-SIDE DISCOVERY PATTERN

The client-side registry pattern has a few benefits and drawbacks.

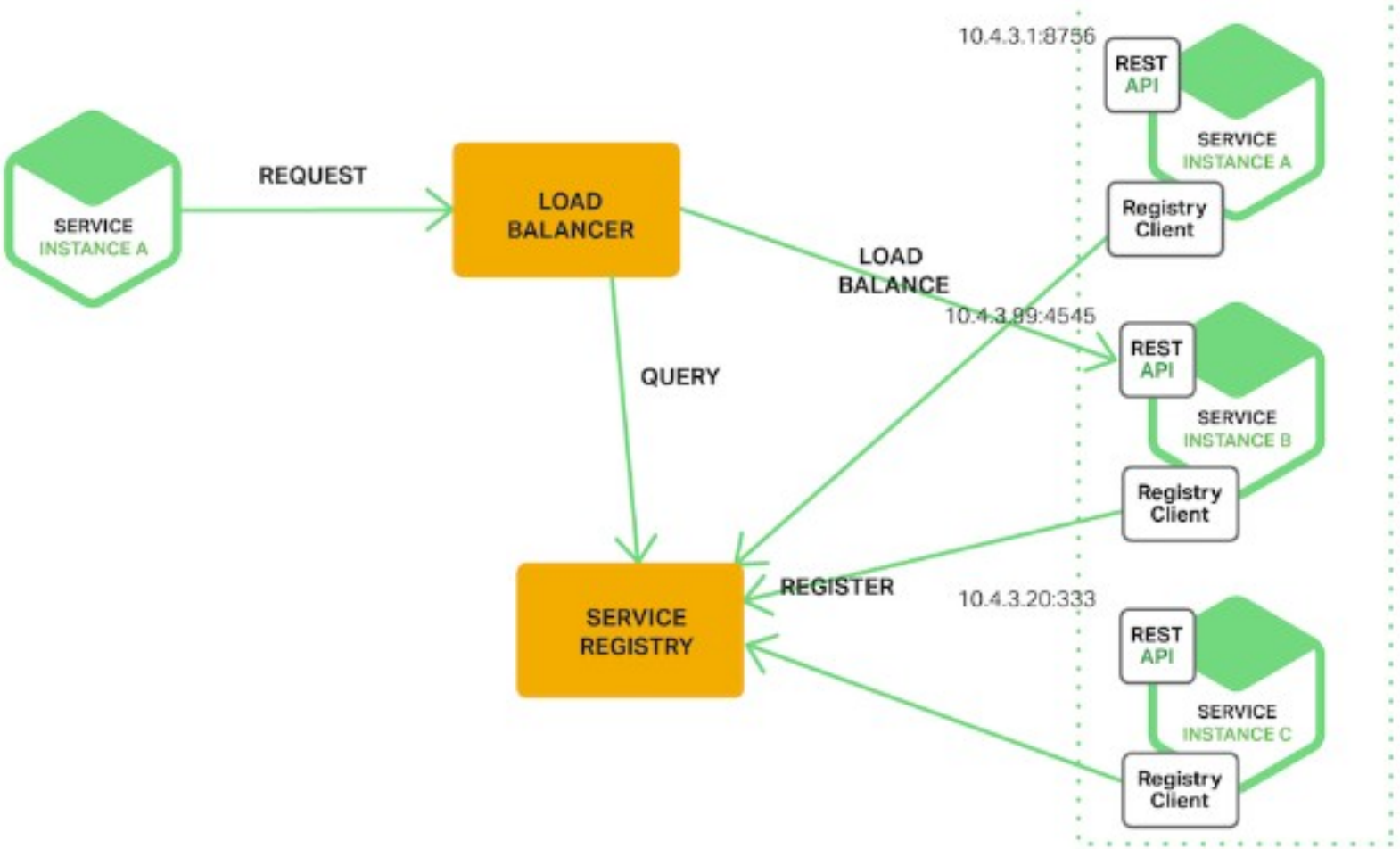
The following are the benefits:

- It is relatively simple, without additional components required except for the registry.
- The client can make intelligent, application-specific load balancing decisions.

The drawbacks are as follows:

- You must implement client-side service discovery logic for each programming language and framework used by your service clients.

SERVER-SIDE DISCOVERY PATTERN



SERVER-SIDE DISCOVERY PATTERN

The server-side pattern has several benefits and drawbacks.

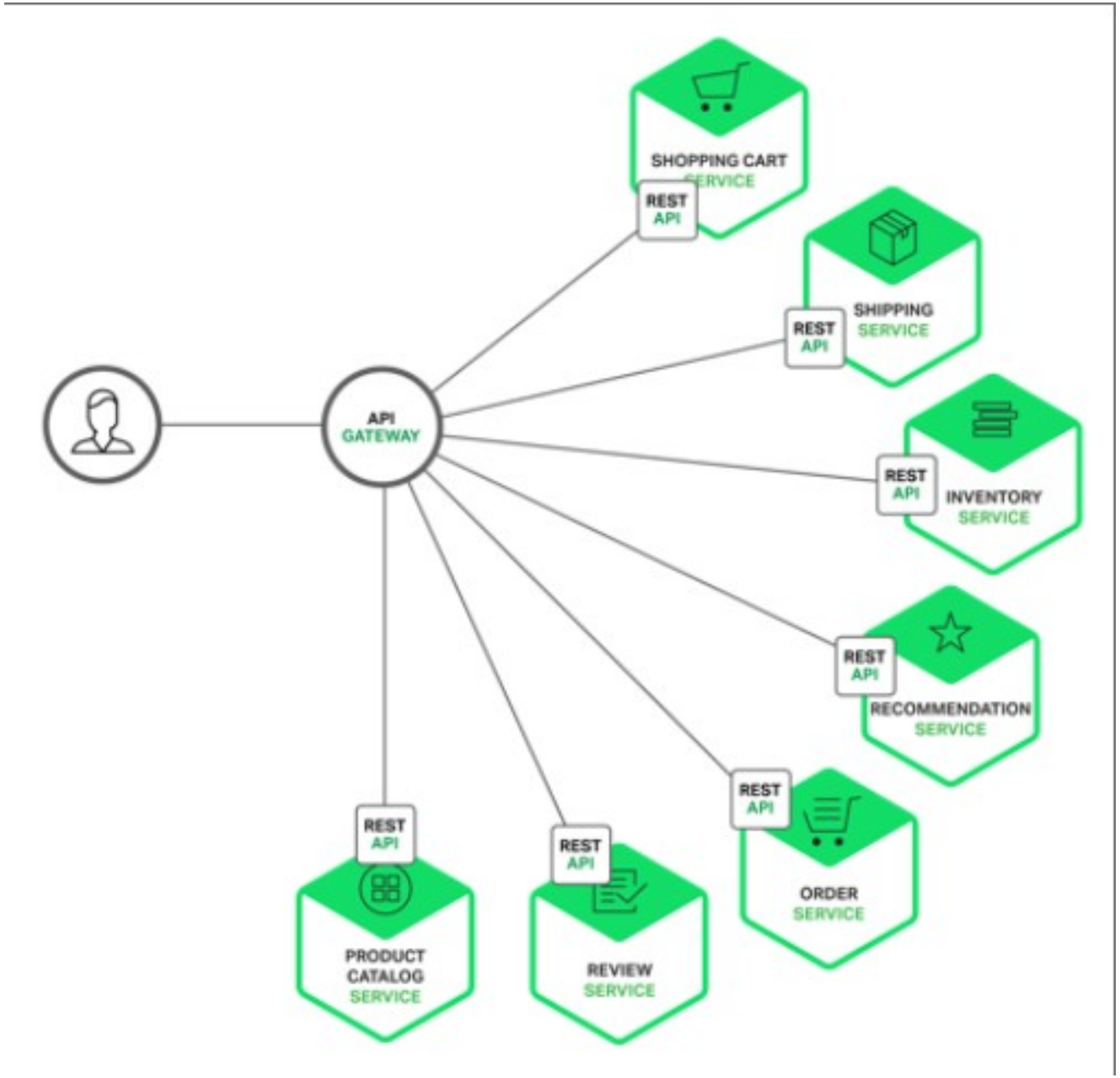
The benefits are as follows:

- Compared to client-side discovery, the client does not need to know how to deal with discovery. The discovery is abstracted away from the client. Instead, a client simply requests the load balancer (LB).
- This eliminates discovery logic for each programming language and framework used by your service consumers.
- Some cloud environments provide this functionality like cloud ELBs.

The drawbacks are as follows:

- Unless it is part of the cloud environment, the LB is another system component that must be installed and configured. It will also need to be replicated for availability and capacity.
- More network hops are required than the client-side discovery.

API GATEWAY


SHOPPING CART SERVICE
REST API
SHIPPING SERVICE
REST API
API GATEWAY
REST API
INVENTORY SERVICE
REST API
RECOMMENDATION SERVICE
REST API
ORDER SERVICE
REST API
PRODUCT CATALOG SERVICE
REST API
REVIEW SERVICE

HAVE A GOOD DAY!