

Cloud Integration

Benoît Charroux

Benoît Charroux

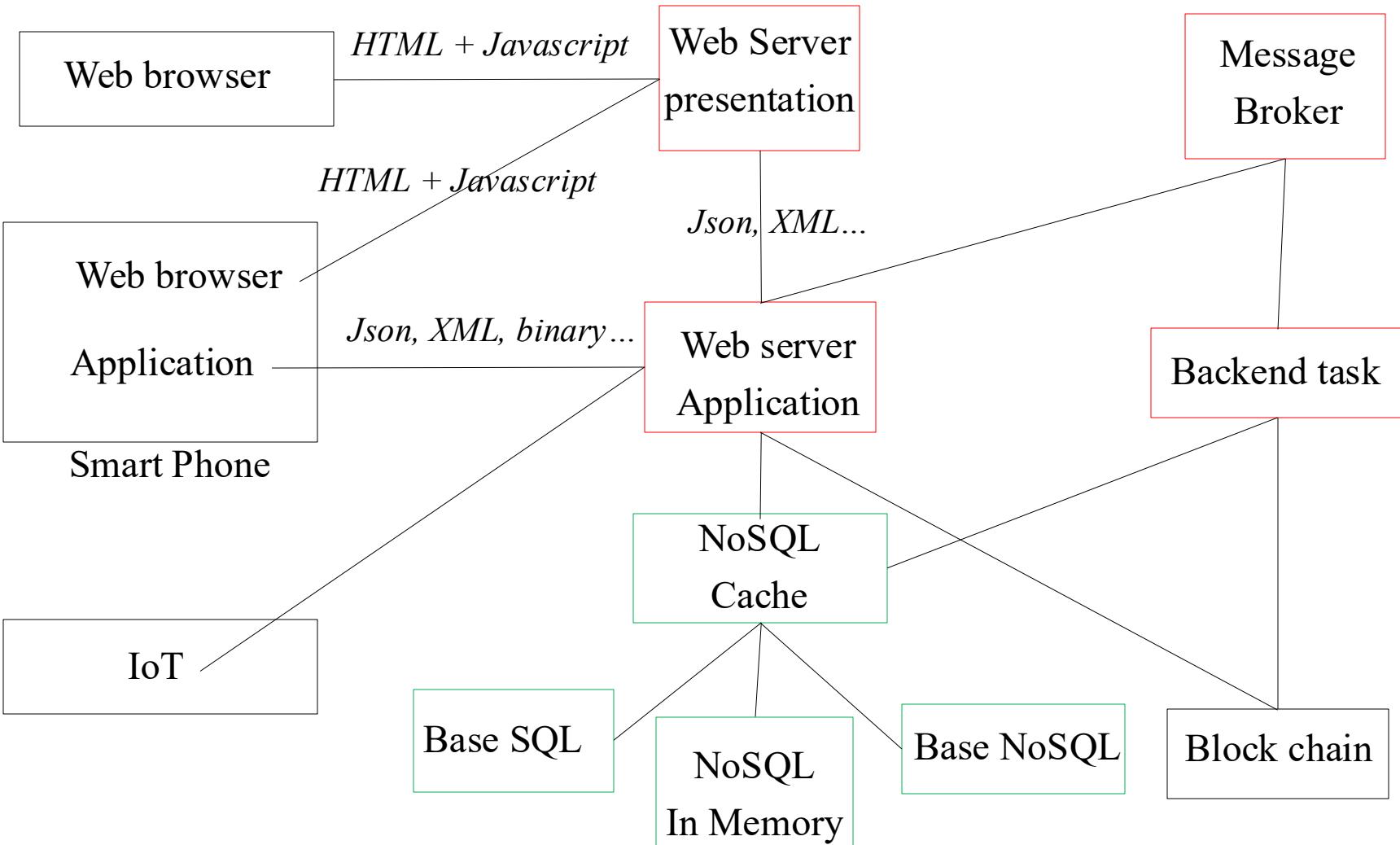
- Enseignant / Chercheur
- Domaines de recherche :
 - Intégration des blockchains dans les systèmes d'information des entreprises afin de transformer les processus métiers en Smart Contract.
 - Permettre la collaboration inter-entreprises en toute confiance.
 - Technologies :
 - Intégration
 - Kafka
 - Docker/Kubernetes

Summary

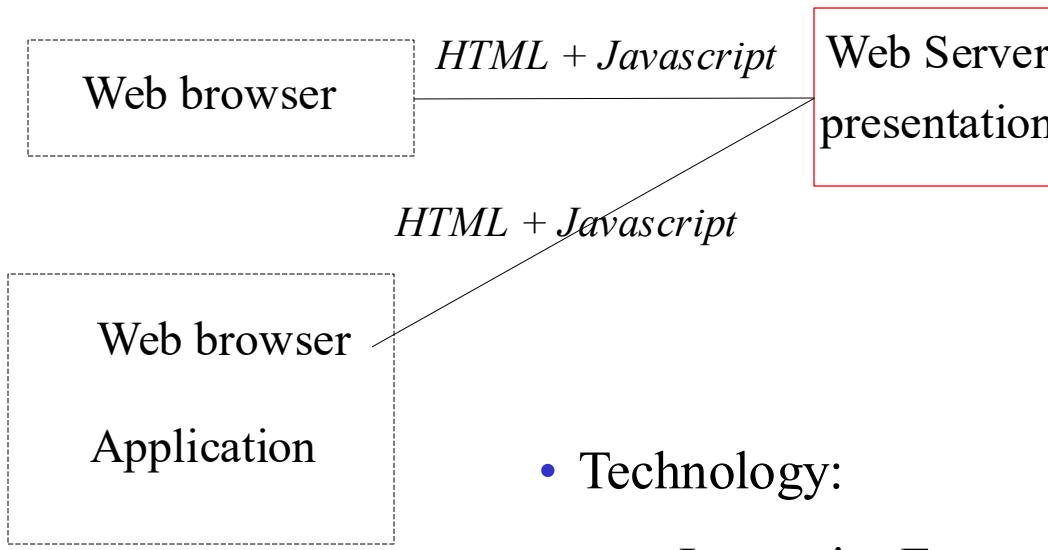
- The big picture of the web services
- Web services
 - Rest web service
 - gRPC
- Message Oriented Middleware
- Microservices
- Applications cloud native
- API gateway / API management

The big picture of the Web services

From the end user to the database



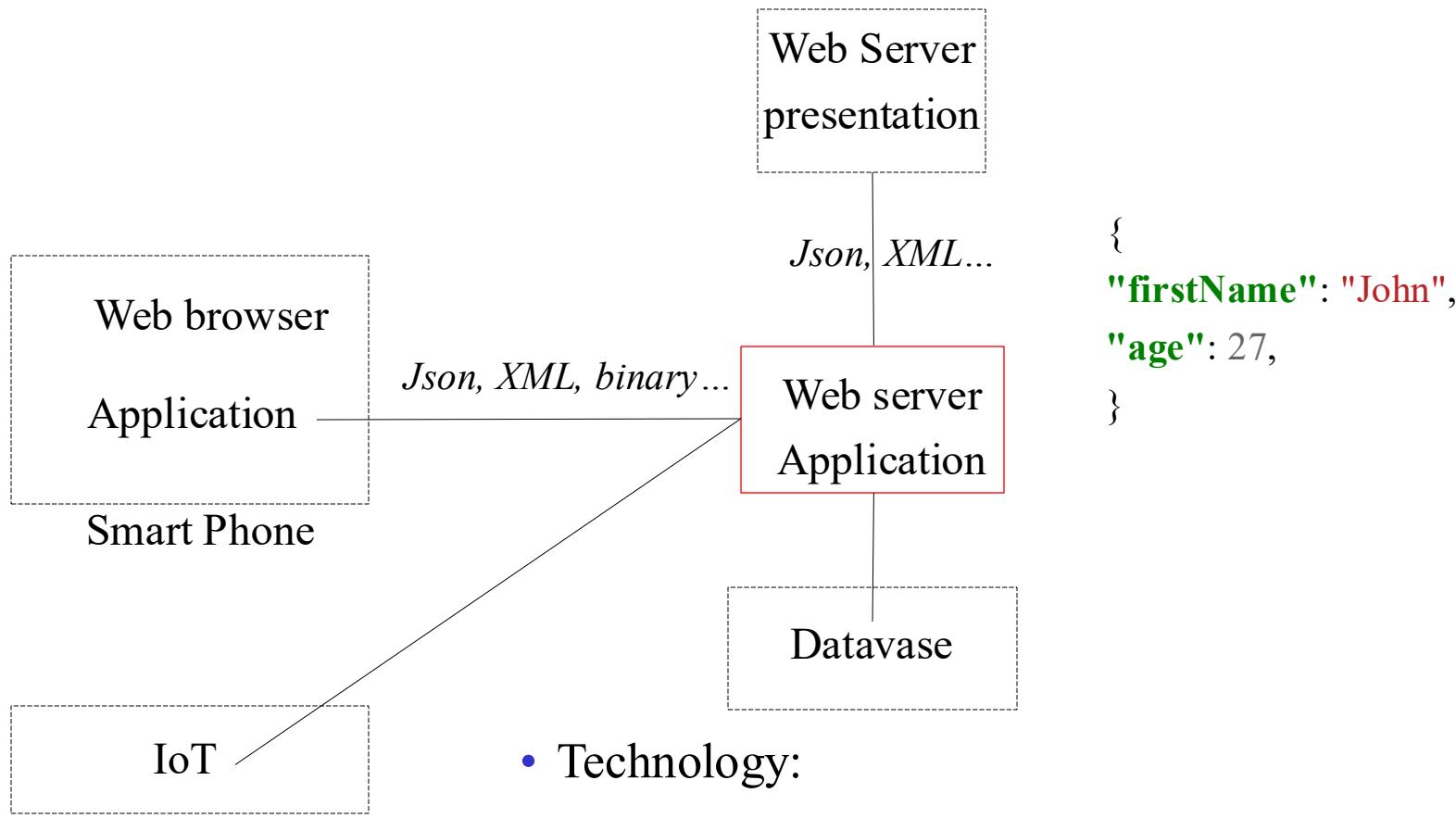
Presentation server



- Technology:
 - Javascript Framework (Angular, React, VueJS...)
 - Example angular:

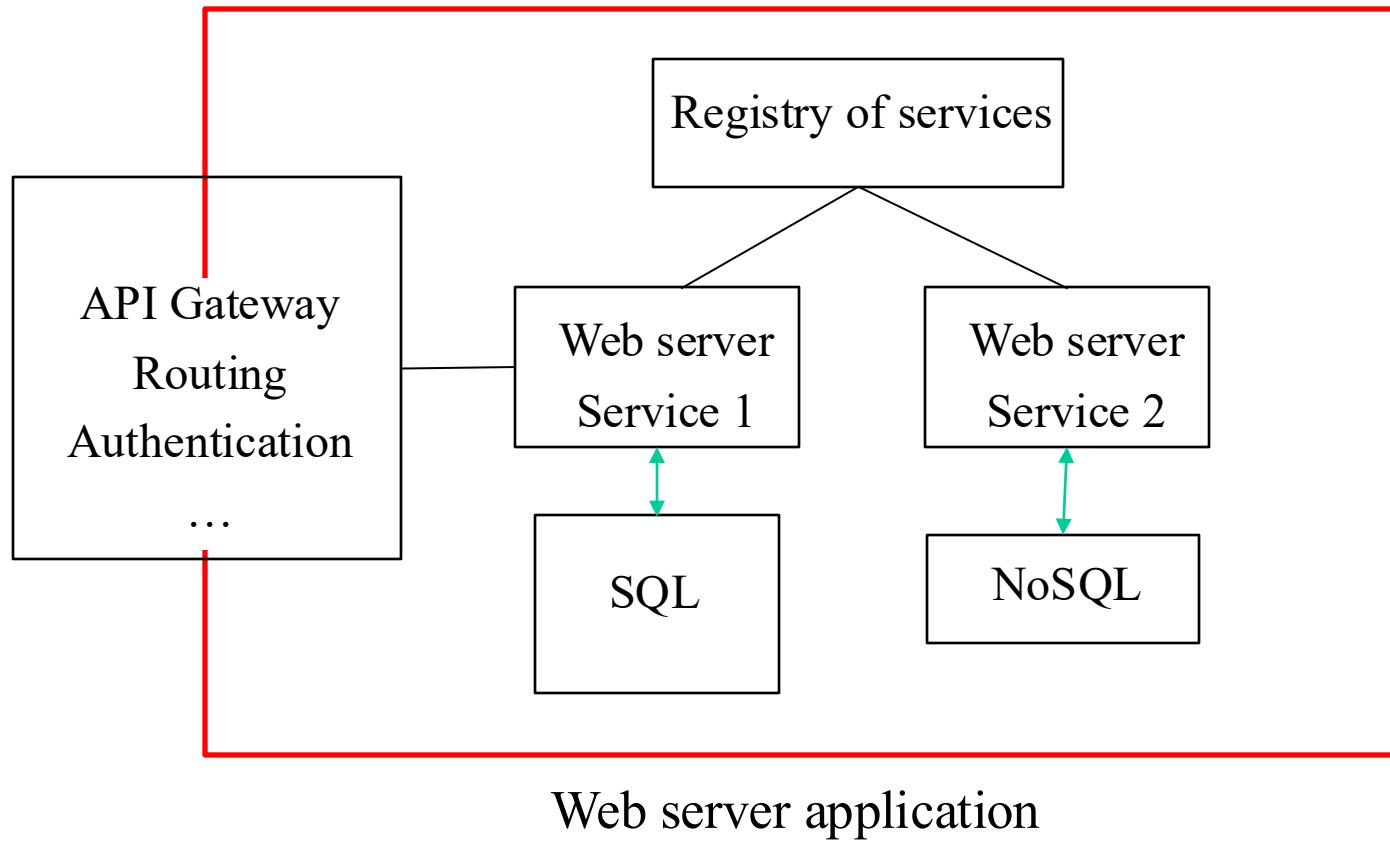
```
<div *ngFor="let item of items" [item]="item"></div>
```

Service



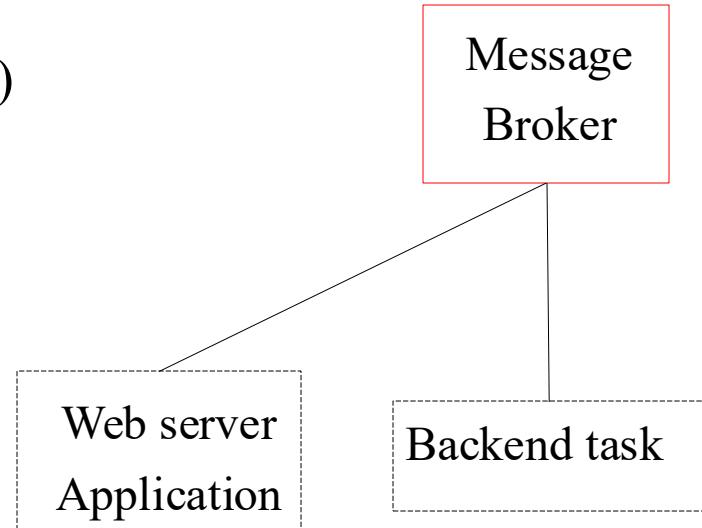
- Technology:
 - Web Service Rest
 - Remote Procedure Call
 - Multiple languages: Java, Node.js, Python...

Microservices



Asynchronous microservices

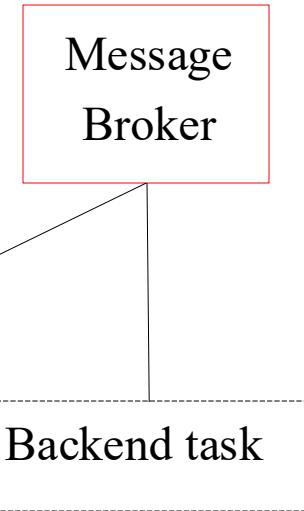
- Technology:
 - Message brokers (Kafka, RabbitMQ...)
- Properties:
 - Asynchronous
 - Message Driven, Event oriented
 - Low latency
 - High availability
 - Fault tolerance
 - Durable
 - Trillions of messages per day
 - Multi languages: Java, Python...



Event Driven Systems

- Messages contain events

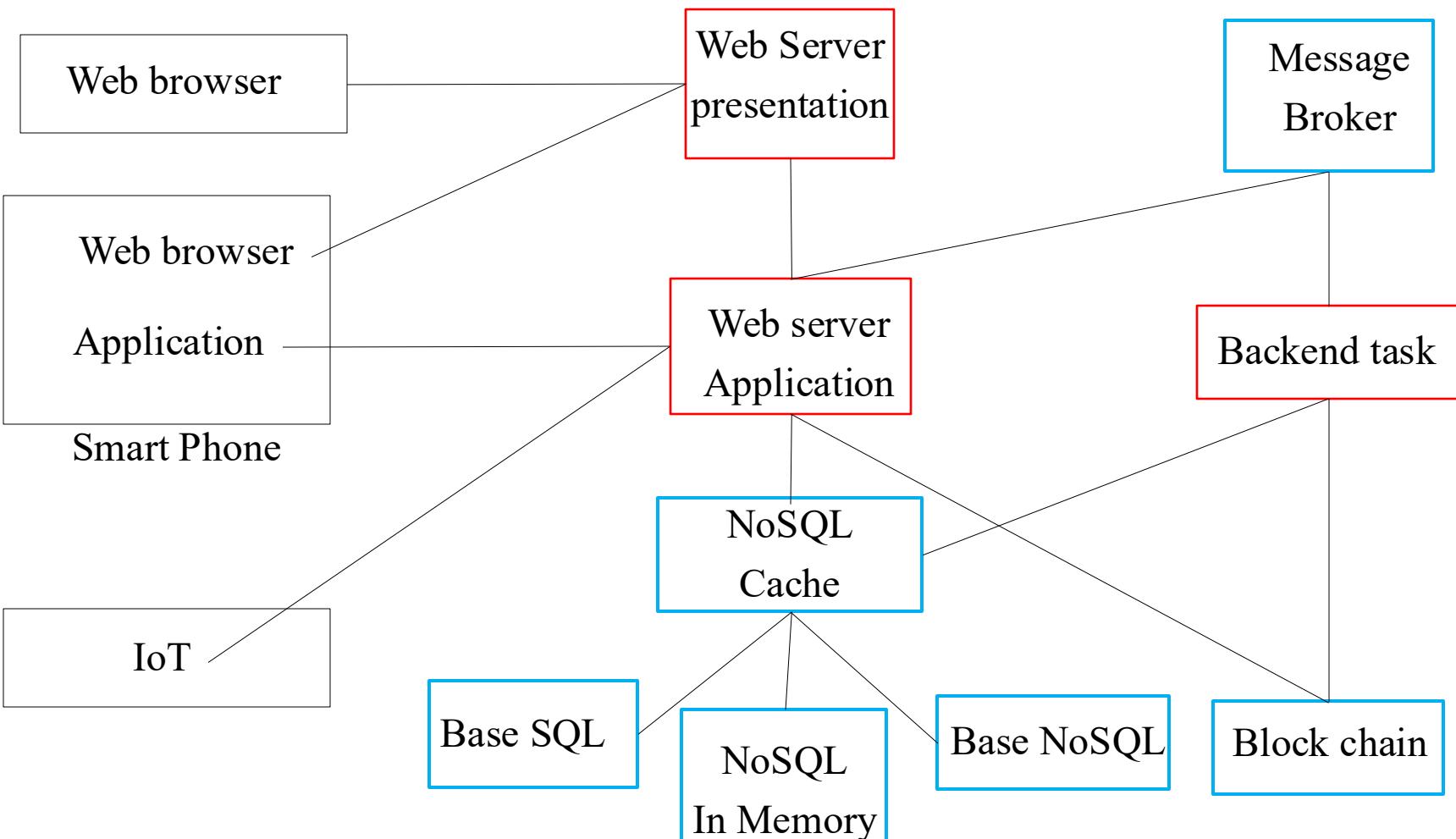
Paul	added	3 pants	12:13
Emilie	added	1 T-shirt	12:27
Paul	removed	1 pants	13:04
Emilie	added	1 hat	14:12



- Stream processing

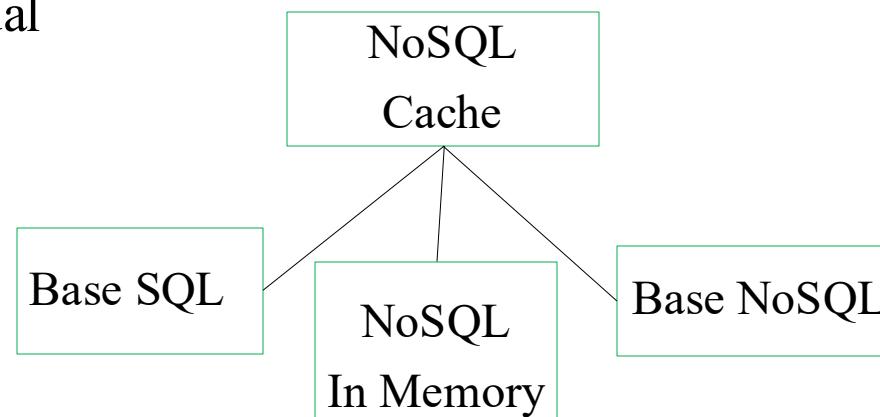
Cloud native

Containerized app



Database

- SQL:
 - Standard query language
 - Low but ACID
- NoSQL:
 - Supports heterogeneous data
 - Simpler horizontal scaling to cluster
 - Speed but eventual consistency
- NoSQL:
 - Key-value store => DynamoDB
 - Document store => MongoDB
 - Object base => CouchDB
 - Graph => Neo4J
- Cache:
 - Key-value store => Redis



Database and Cloud providers

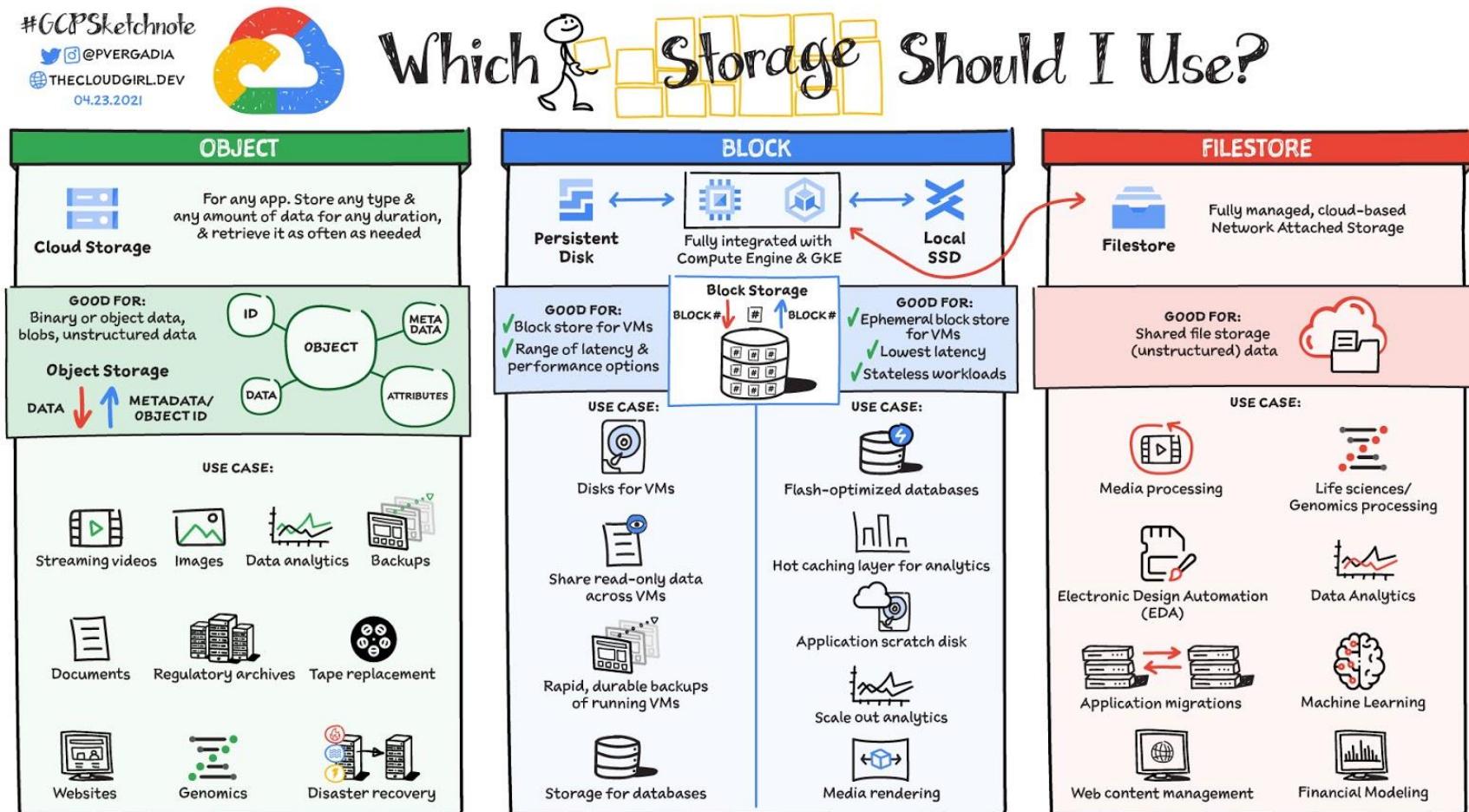
	AWS	Azure	Google Cloud Platform
Relational Database	RDS Instance de DB relationnelle (MySQL, PostgreSQL, Oracle, SQL Server, MariaDB) RDS on VMware	Azure Database (MySQL, PostGreSQL, SQL Server, MariaDB) SQL Edge	Cloud SQL (MySQL, PostGreSQL, SQL Server), BareMetal
NoSQL	DynamoDB / Keyspaces NoSQL DB / Cassandra	Table Storage / Azure Managed Instance for Apache Cassandra NoSQL DB / Cassandra	Firebase Realtime Database
Cache	Elasticache		MemoryStore
Warehouse	Redshift	Datalake gen2	BigQuery Looker
High performance database	Aurora & Aurora Serverless		Cloud Spanner
Other	Neptune (BD de graphes), Quantum Ledger (transaction sécurisée), Timestreams (time series), DocumentDB (MongoDB)	Cosmos DB (NoSQL), Azure confidential ledger (preview)	BigTable (colonne), Firestore (Document)
In memory	MemoryDB for Redis	Azure Cache for Redis	Memory Store

Storage

	File	Block	Object
Accessibility	File server via OS/protocole	Low level interface	HTTP (REST)
Examples	NFS, SMB... Hadoop FS...	SAN iSCSI, FC... Amazon Elastic Block Storage, Openstack Cinder, Ceph RADOS...	Amazon S3, Openstack Swift, Ceph Storage...

Use case for storage

#GCPSketchnote
Twitter: @PVERGADIA
TheCloudGirl.dev
04.23.2021



Service providers offers

	AWS	Azure	Google Cloud Platform
Object storage	Amazon Simple Storage Service	Blob	Cloud Storage
Block storage	Elastic Block Store	Disk	Persistent Disk Local SSD (éphémère)
File storage	Elastic File System, FSx for Lustre, FSx for Windows File Server, NetApp ONTAP, OpenZFS	File (SMB), Netapp Files	File Storage (NFS)
On-premises gateway	AWS Storage gateway	StorSimple	
Backup	AWS Backup	Azure Backup	Actifio
Cold storage	Amazon S3 Glacier (storage classes)	Archive Storage	Archive Storage
Disaster Recovery	CloudEndure	Azure Site recovery	Actifio

Summary

- The big picture of the web services
- **Web services**
 - Rest web service
 - gRPC
- Message Oriented Middleware
- Microservices
- Applications cloud native
- API gateway / API management

Web Services

Restful architecture

Remote Procedure Call

Definition

- OASIS defines a service as "a mechanism to enable access to one or more capabilities, where the access is provided using an interface and is exercised consistent with constraints and policies as specified by the service description.

Definition

- A Web Service is a program allowing communication and data exchange between applications and heterogeneous systems in a distributed environment:
 - Representational State Transfert (REST):
 - Simple and standard
 - Remote Procedure Call:
 - SOAP:
 - Complex and for legacy systems
 - gRPC:
 - HTTP/2 : asynchronous (temporal recoupling)

REST Web Services

What is REST ?

- **Representational State Transfert** is an abstraction of the architecture of the World Wide Web :
 - States are represented in Json, XML...
 - Transfert through HTTP
- Proposed by Roy Fielind in 2000 !

RESTful architecture

- The 4 rules:

- Each resource has a unique URI
- Resource are represented in Json, XML, Atom...
- Communicate using the HTTP protocol (GET, PUT, POST, DELETE):
 - GET: send back data to the client from the server
 - PUT: update data to the server
 - POST: create data on the server side
 - DELETE: delete data on the server
- Hypertext links to futures states and resources.

A car rental service

- The resources: cars
- Resources URI: <http://www.rental.com/>
- Protocole:
 - Get the list of cars to be rented:
 - URI: [http://www.rental.com / cars](http://www.rental.com/cars)
 - http: GET
 - Json response: [{"plaque": "22AA33"}, {..}]
 - Get the features of a car:
 - URI: [http://www.rental.com / cars / 22AA33](http://www.rental.com/cars/22AA33)
 - http: GET
 - Json response: {"plaque": "22AA33"}

Design a Rest application: a car rental service

- Protocole:
 - Rent a car:
 - URI: http://www.rental.com / cars / 22AA33 ? rent=true
 - http: PUT
 - Send date: { "begin" : "..." }
 - Get back the car:
 - URI: http://www.rental.com /cars / 22AA33 ? rent=false
 - http: PUT

HATEOAS

- Hypermedia as the Engine of Application State:
 - All subsequent requests are discovered inside the responses to each request.
 - Rent a car returns links to get back the car:
 - Response:

```
{  
  "links": {  
    "getBackCar": "/plateNumber?rent=false"  
  }  
}
```

Properties of REST

- Stateless:
 - each request contains all information to understand the request.
 - The session management must be:
 - At the client side or
 - In a database
 - Every server can serve any client at any time.
- Benefit: scaling
- Clear contract is optional (unlike gRPC):
 - Swagger

The lockdown problem

- User A => GET /car/AA11BB
- User B => GET /car/AA11BB
- User A => PUT /car/AA11BB
- User B => PUT /car/AA11BB
- User A => GET /car/AA11BB => incoherent state



- Pessimistic locking: User A locks the resource.
- Drawback: how long user A keep the resource?
- Optimistic locking: Any user can get the resource but only user A can update it (Mongo DB can mark a resource with a version id).

Rest web service implementation

<https://github.com/charroux/servicemesh>

gRPC

HTTP/1.1

- Textual (non-binary) protocol.
- Ordered and blocking
- Requires multiple connections for parallelism.

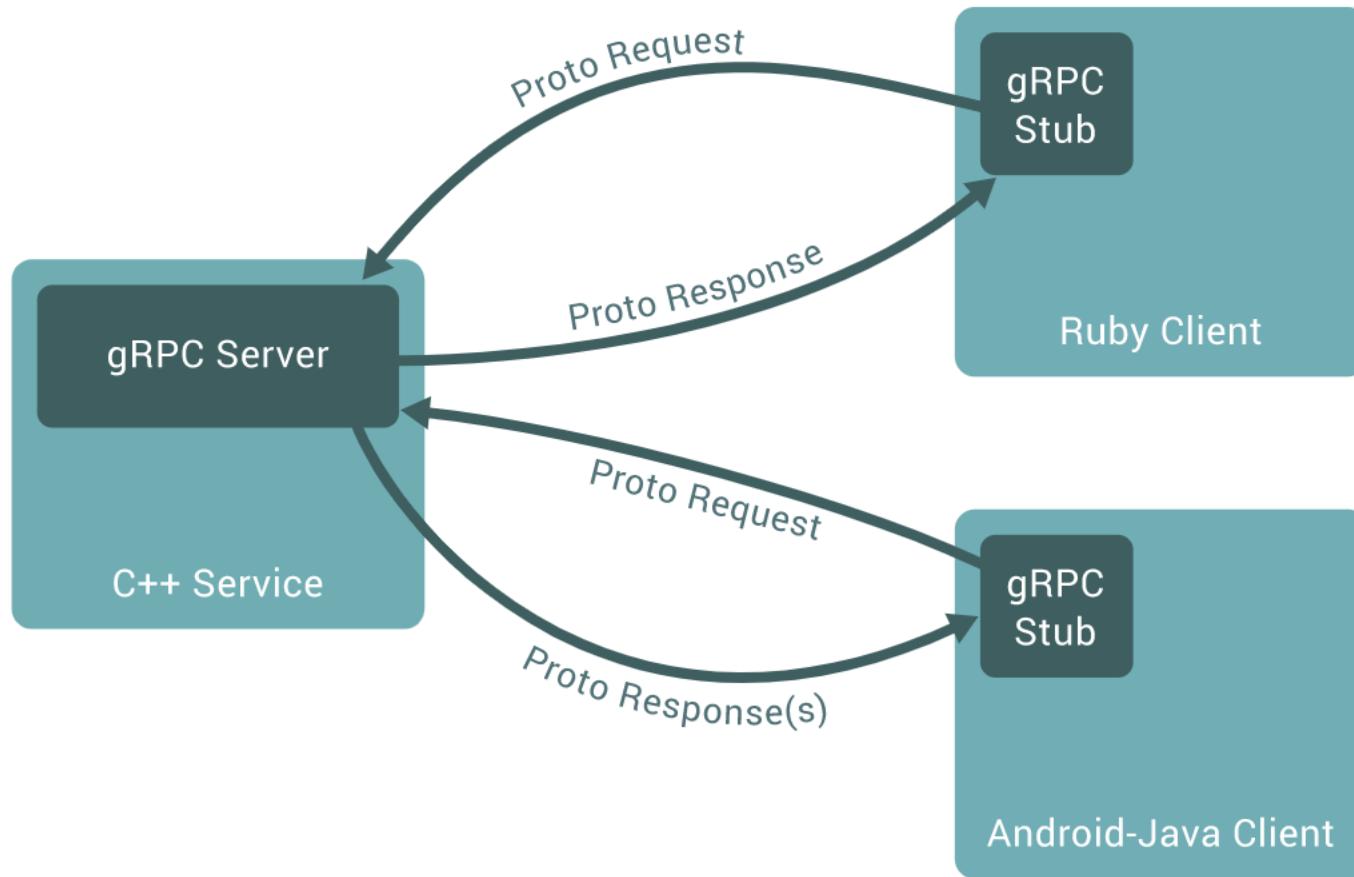
JSON

- Human readable.
- Not secure (clear text).
- Not strongly typed => errors.
- Requires manual (de)serialisation.

HTTP/2

- Single TCP connection per client-server.
- Requests Multiplexing.
- Header compression.
- Bidirectional streaming.
- Server push.

Remote Procedure Call



Protocole Buffer

Protocole Buffers

```
service CarRentalService {  
    rpc rentCars(stream Car) returns (stream Invoice) {}  
}  
  
message Car {  
    string plateNumber = 1;  
    string brand = 2;  
    uint32 price = 3;  
}  
  
message Invoice {  
    uint32 price = 1;  
}
```

Service method

- Single request => single response.
- Single request => response as a stream (sequence of messages).
- Stream => single response.
- Bidirectional streaming RPC (both sides send a sequence of messages).

Synchronous or asynchronous call

- Synchronous => calls that block until a response.
- Asynchronous => Never block.

Languages

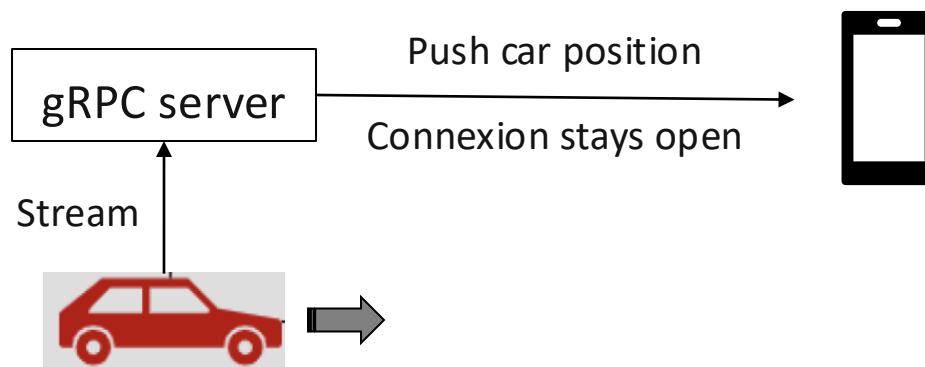
- C# / .NET
- C++
- Dart
- Go
- Java
- Kotlin
- Node
- Objective-C
- PHP
- Python
- Ruby

Platforms

- Android
- Web
- Coming soon:
 - IoS
 - Flutter: mobile, Web, desktop

Use cases

- Temporal unifies workflow activities written in several programming languages:
 - one developer code activity in PHP
 - another one code in Go.
 - Each activity sends its mission-critical data to a gRPC client that forwards it to a gRPC server.
- Lyft uses gRPC to transmit the location of a vehicle in a continuous stream of gRPC messages



PRO / CONS

Pro	Cons
Machine Readable	Not human readable
Handles (de)serialization	
Code generator for many languages	Lot of boilerplate code generator at compilation time
Based on a schema to code	
Binary data representation – Less expansive for big payload - Fast	Not browser consumable natively. Java script client is possible.
Strongly typed	Client and server need to access the generated code
Structures can be extended	

Thrift an alternative to gRPC

- Apache fundation
- Multiple serialization formats
- more flexibility in terms of message types and service definition
- wide range of transport protocols such as TCP, HTTP, ...
- Performance: gRPC is generally considered to be faster due to its use of HTTP/2 as a transport layer

Implementation web service

<https://github.com/charroux/servicemesh>

Summary

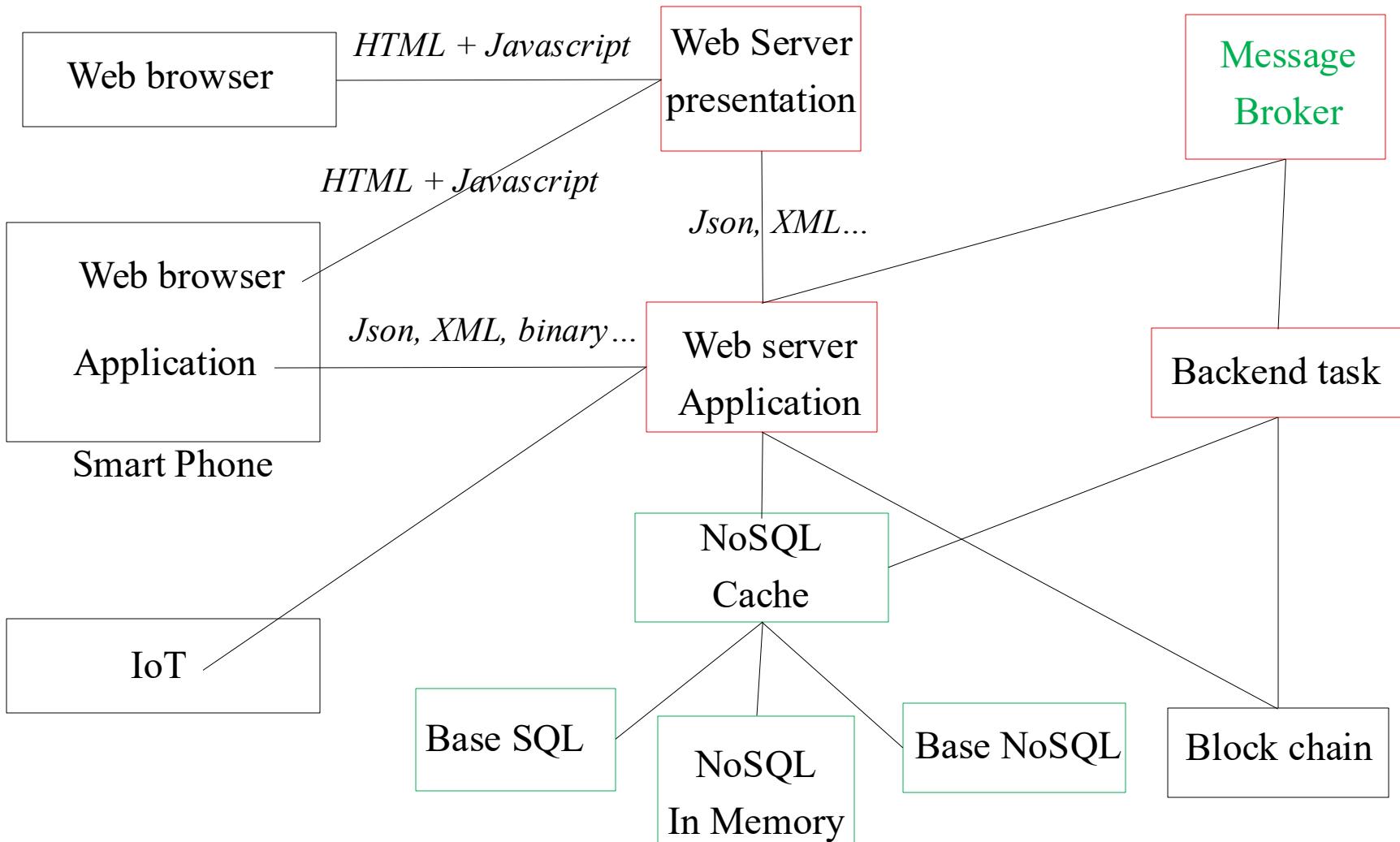
- The big picture of the web services
- Web services
 - Rest web service
 - gRPC
- **Message Oriented Middleware**
 - Microservices
 - Applications cloud native
 - API gateway / API management

Message Oriented Middleware

-

MOM

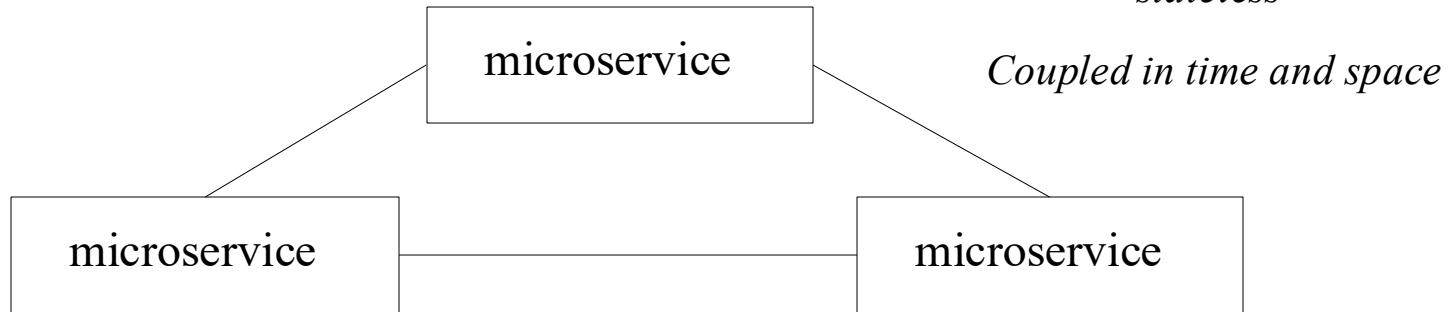
Where are message brokers



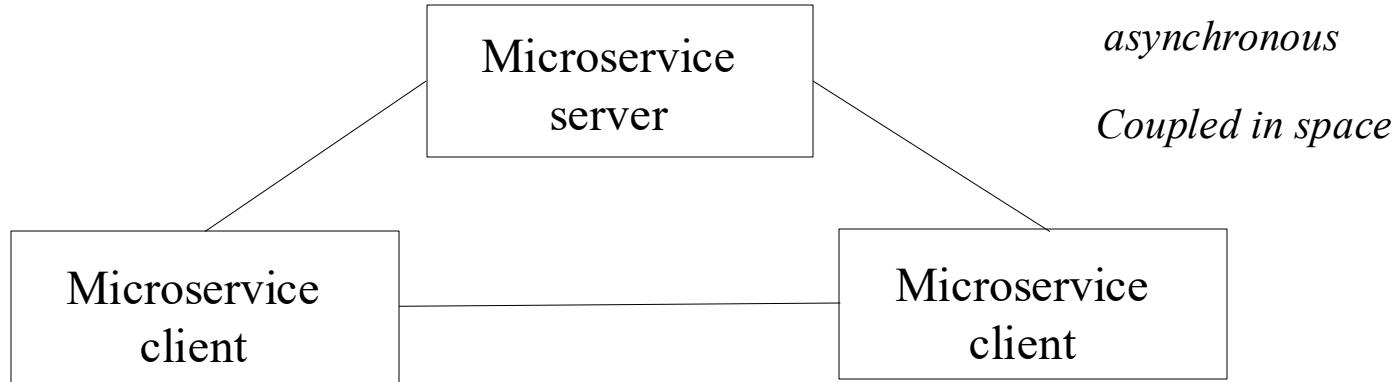
Why MOM?

How do microservices communicate?

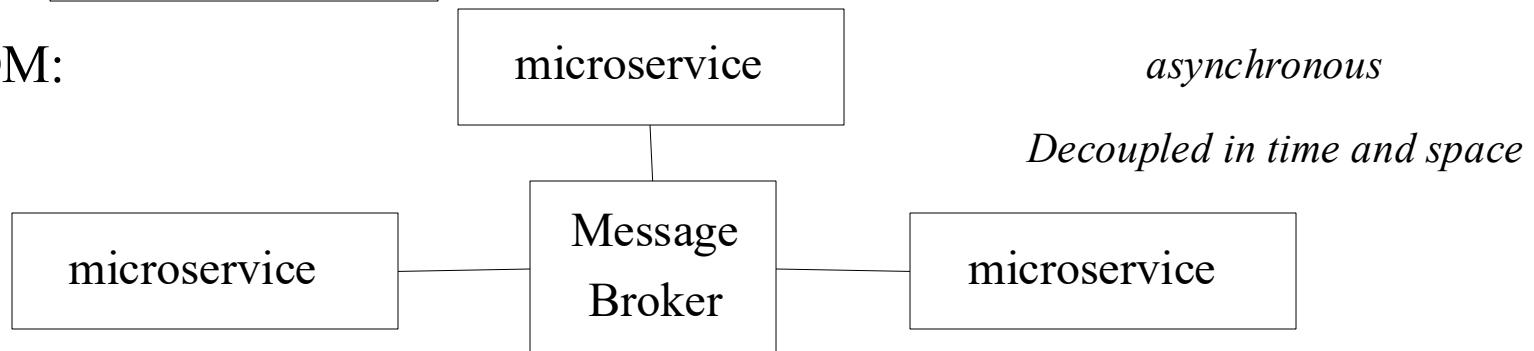
- Rest:



- RPC:



- MOM:



Message driven

- Message driven:
 - Asynchronous message passing => gRPC, message broker
 - Location transparency => Message broker
 - Delegate failure as messages => Message broker
 - Applying back-pressure if necessary => Reactive programming
 - Non-blocking communication => gRPC, message broker

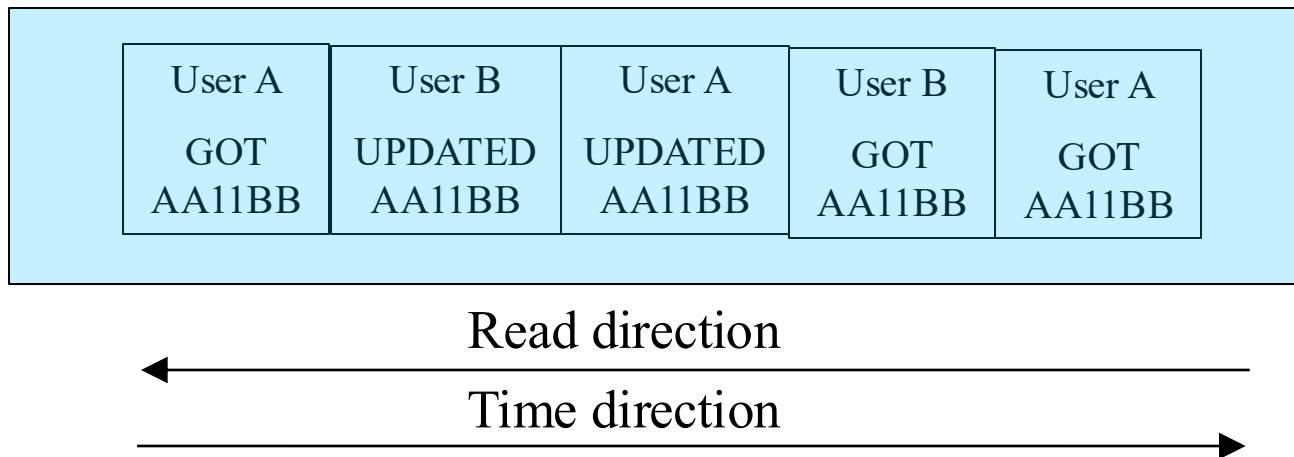
Why MOM? Integration

- Information Systems are:
 - heterogenous
 - decentralized
 - distributed
 - Need for integration => message broker

The MOM approach

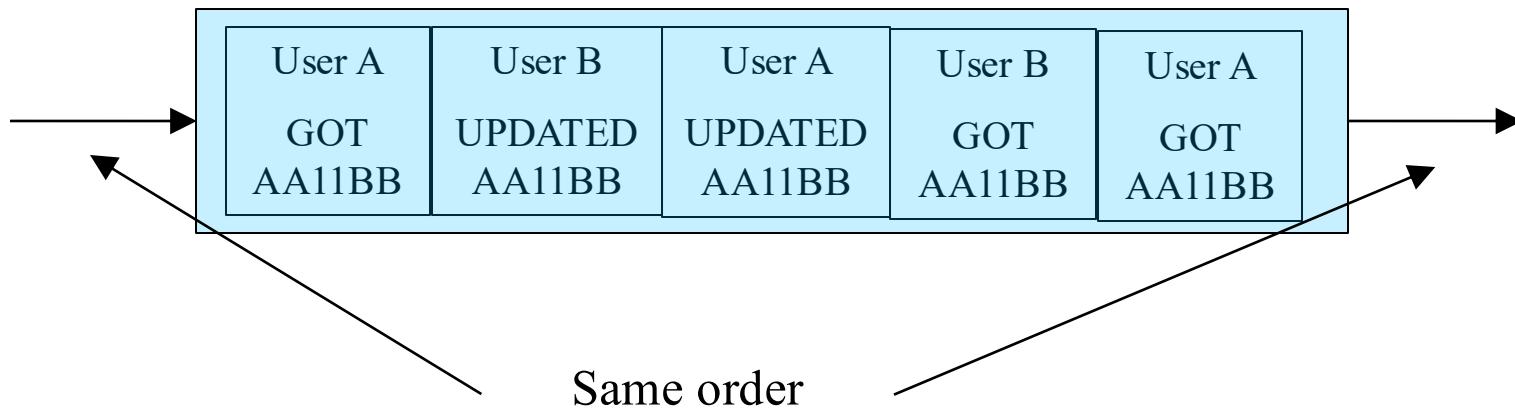
The MOM approach

- Messages are stored and delivered in order (Kafka).
- Message are stored according to a priority (RabbitMQ).
- Kafka keeps the latest version of a message and delete the older versions.

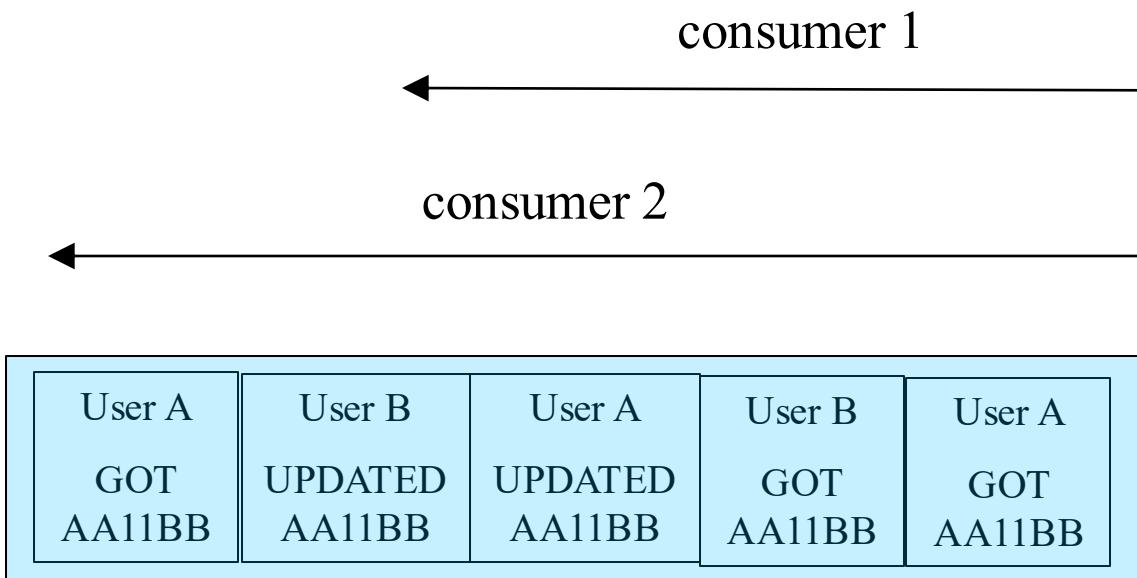


Reading strategies

Reading in order



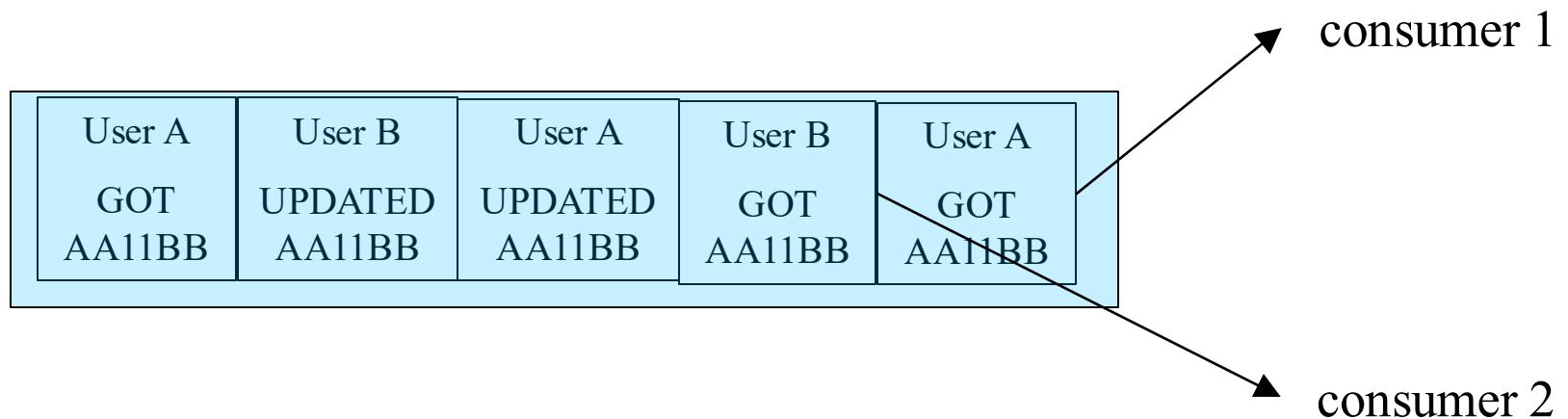
Independent reading



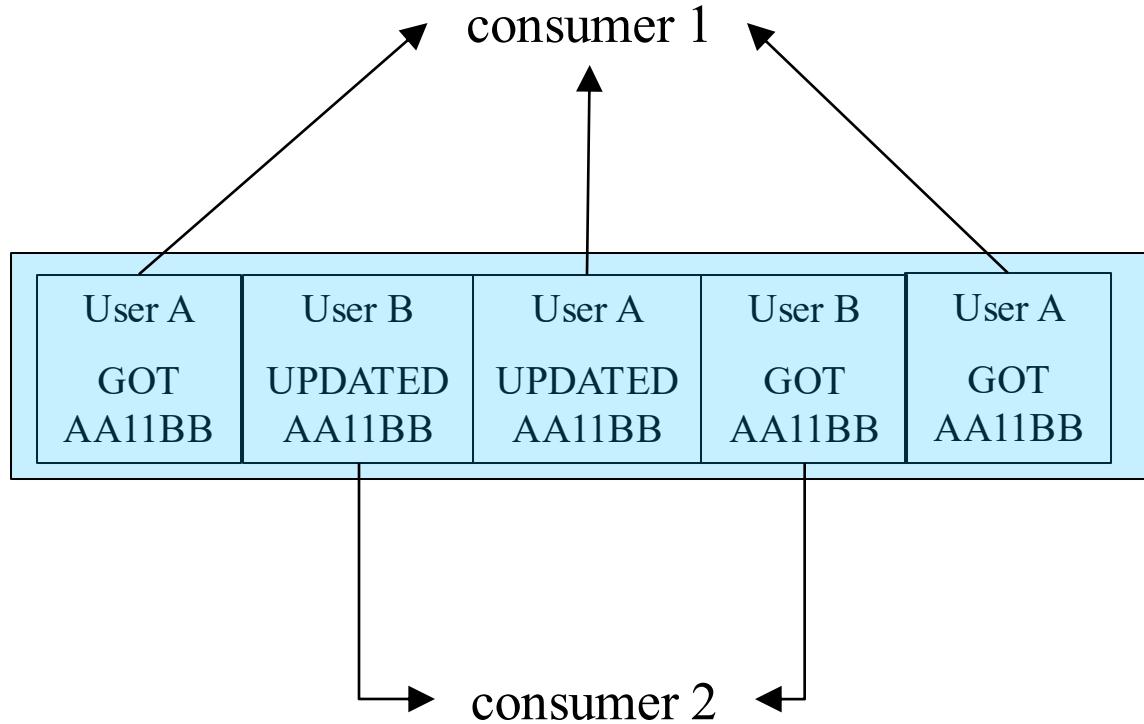
Several interpretations of the past

=> the basis of Command and Query Responsibility Segregation (CQRS)

Load balancing

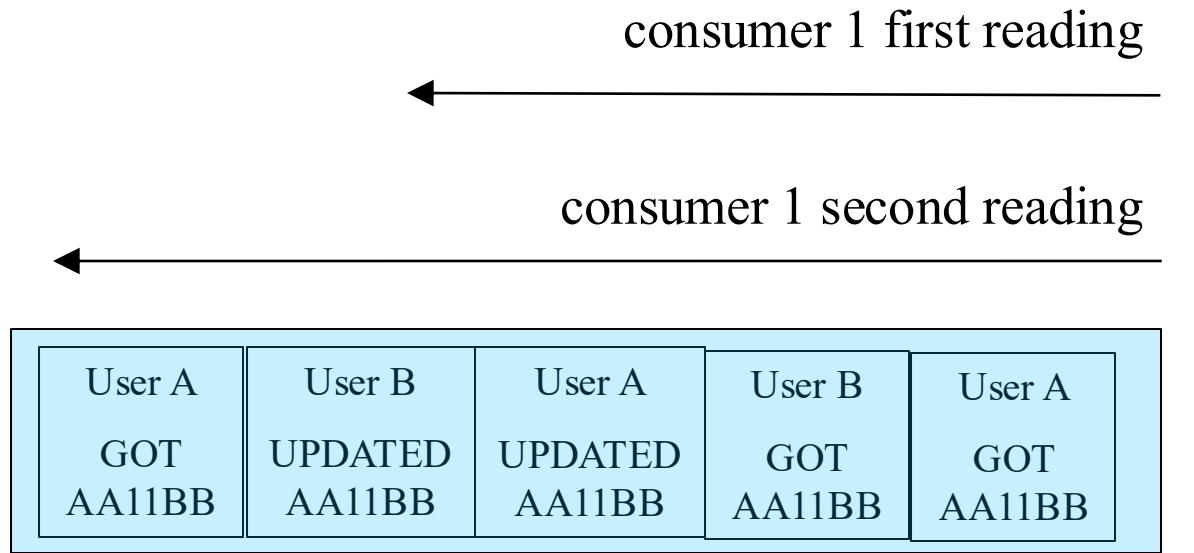


Flexible routing strategies



Replayability

- An immutable and append-only journal of events.



Kafka

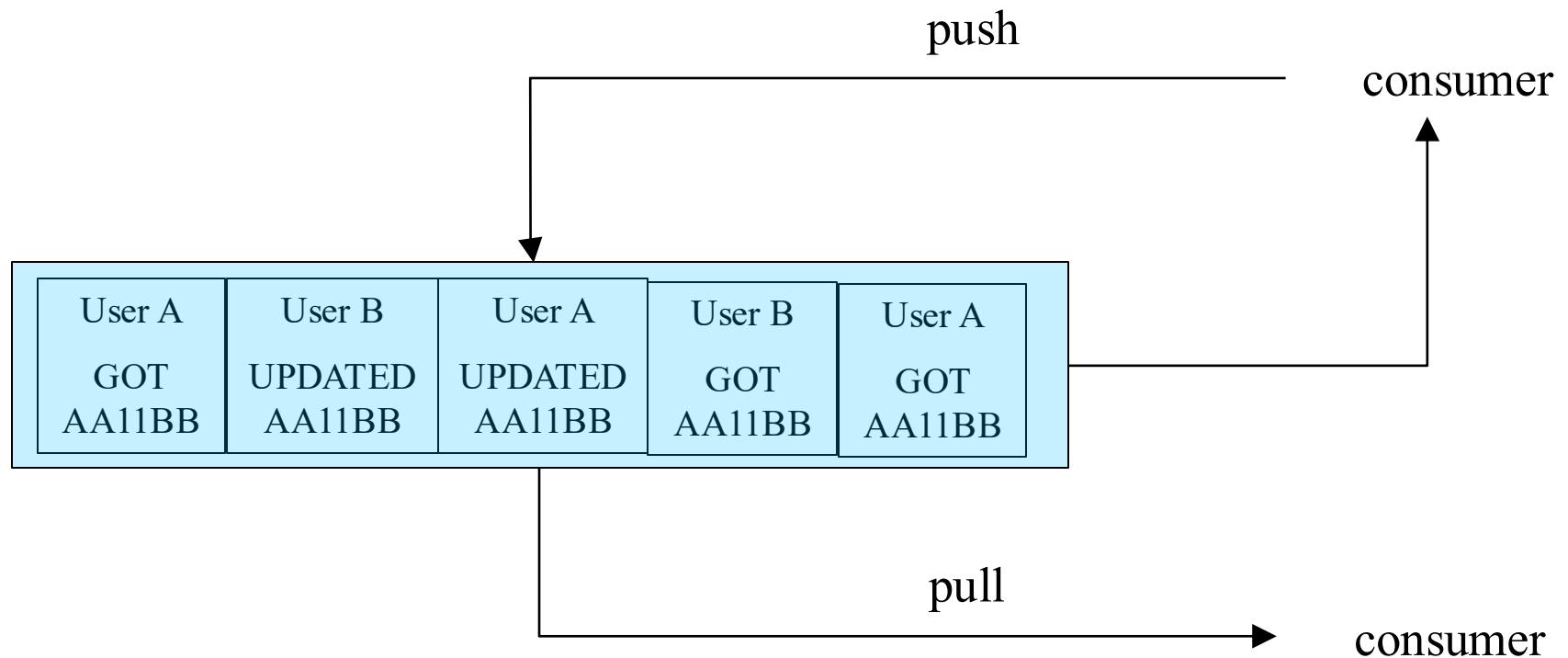
RabbitMQ streams

Several interpretations of the past

⇒ multiple views

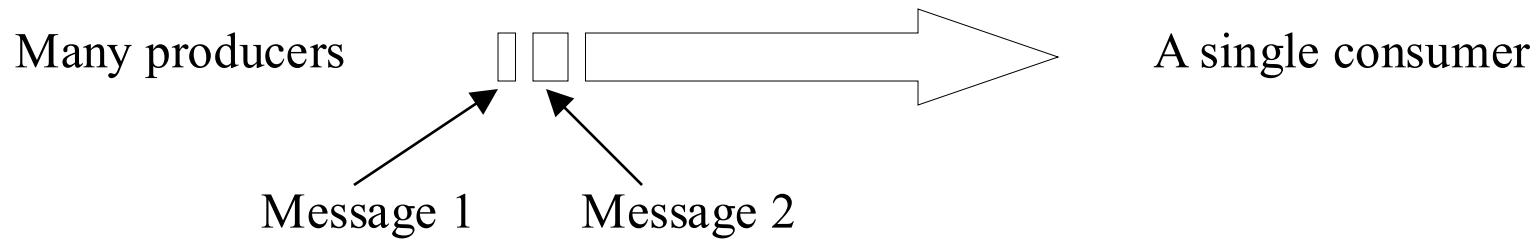
⇒ the basis of CQRS

Push or Pull

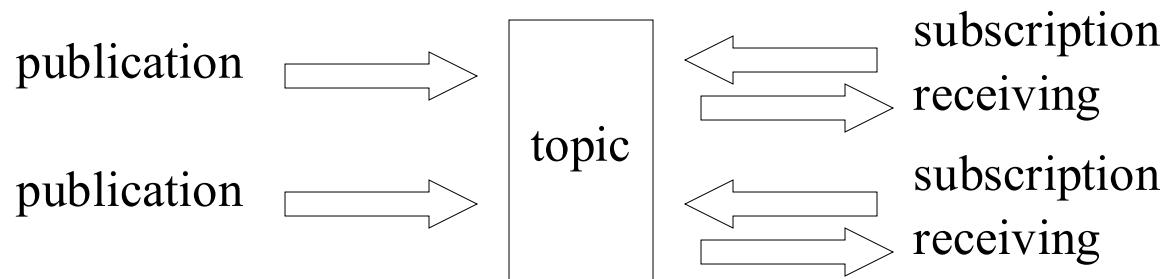


peer to peer
or
publication / subscription

Peer to peer



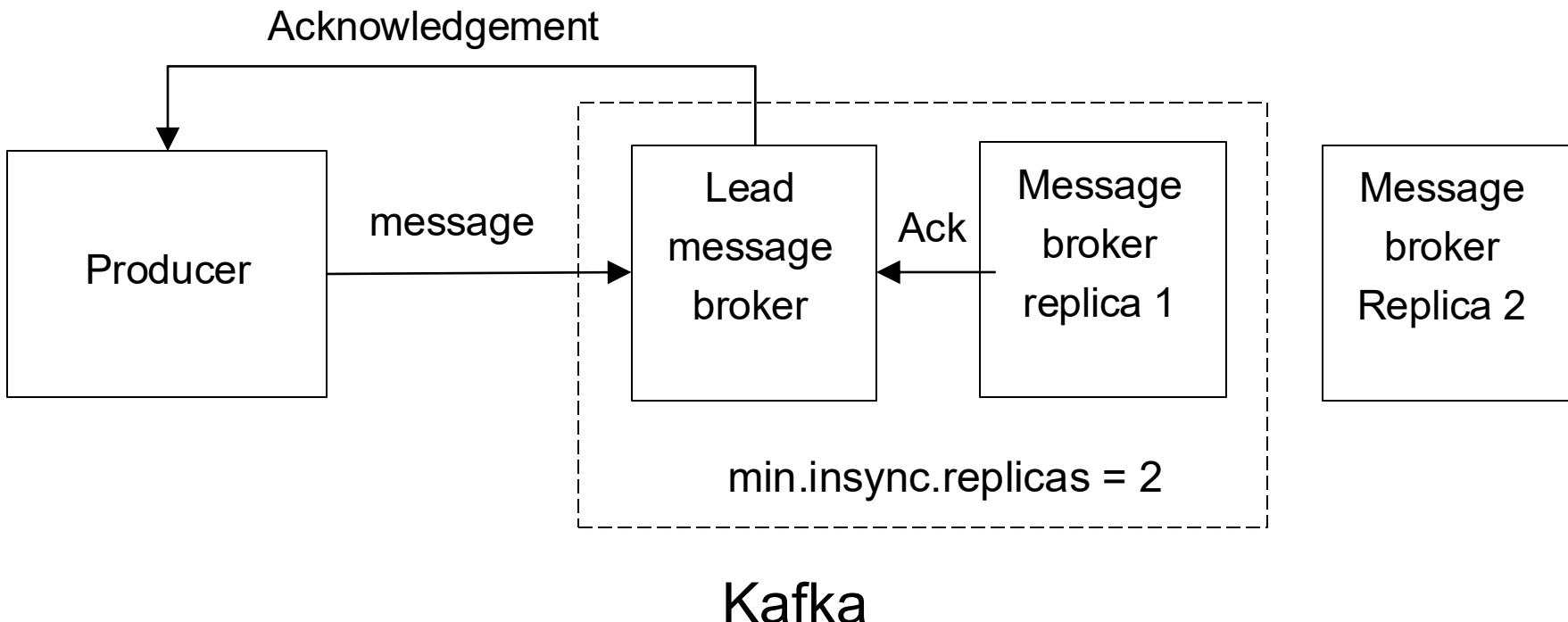
Publication/subscription



Message delivery and
durability guarantees

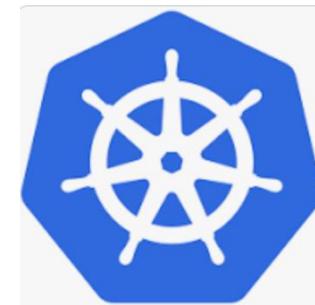
Acknowledgement while producing messages

- The producer waits for an acknowledgment:
 - from the lead broker.
 - from all the in-sync replicas.



Cloud native solutions ?

- Confluent Apache Kafka
 - Fully managed:
 - Elastic scalability
 - Infinite storage
 - Resilience
 - 70+ connectors
 - Multicloud: AWS, Azure, Google
- VMWare RabbitMQ for Kubernetes
 - CloudCloud native messaging and streaming service that you can deploy on any Kubernetes cluster
 - Interoperability with AMQP (Microsoft Azure Service Bus)



 RabbitMQ™

Communication with RabbitMQ

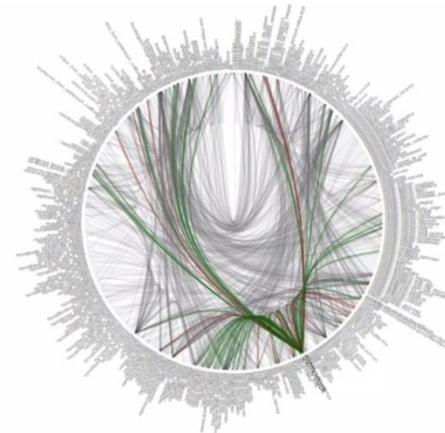
<https://github.com/charroux/eventsourcing>

<https://www.rabbitmq.com/streams.html>

Summary

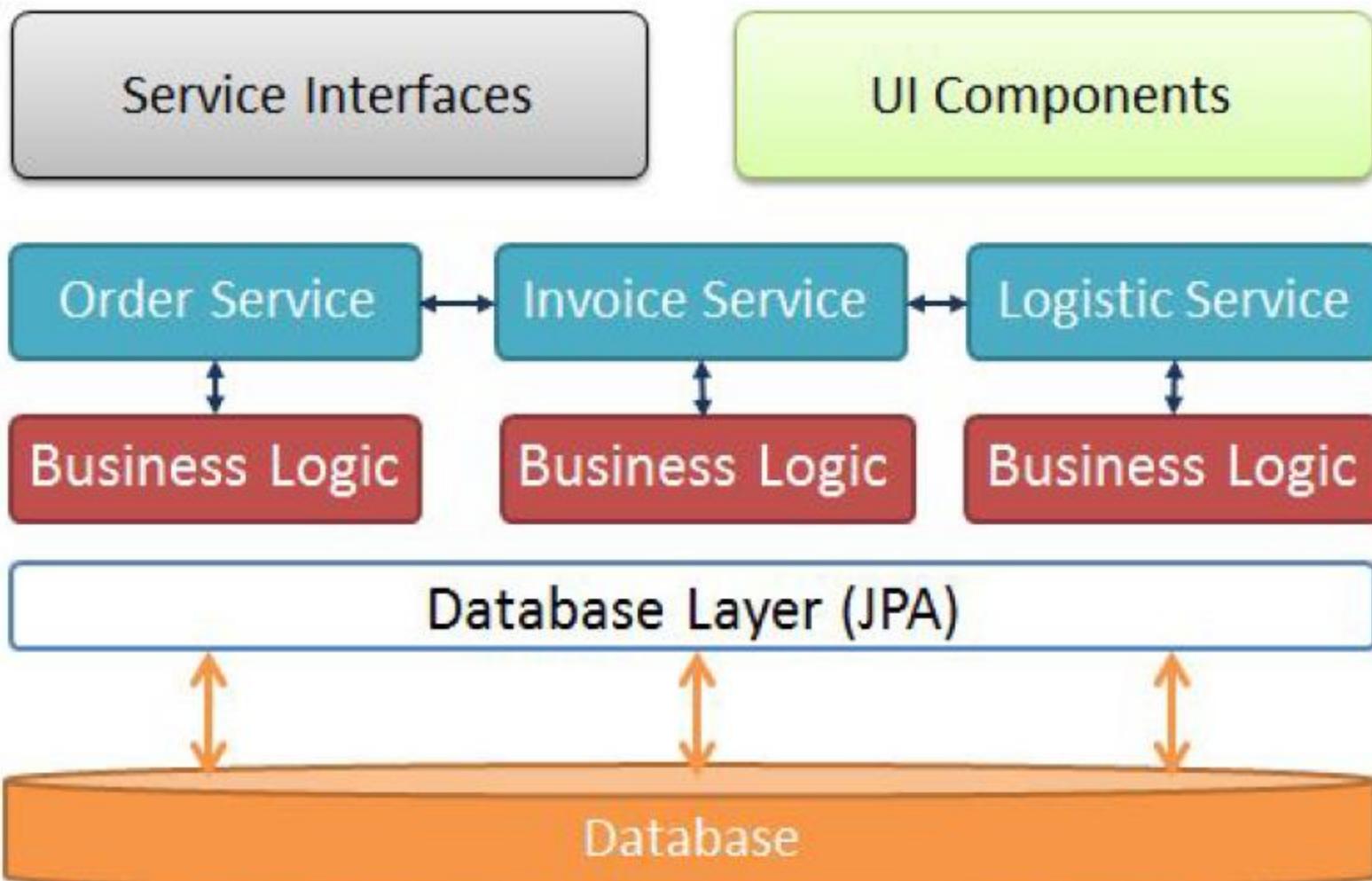
- The big picture of the web services
- Web services
 - Rest web service
 - gRPC
- Message Oriented Middleware
- **Microservices**
 - Applications cloud native
 - API gateway / API management

Microservices



The monolithic architecture

Monolithic Enterprise Application



Drawbacks

- A small change => rebuilt and deployed the monolith.
- Refactoring => hard to keep a good modular structure.
- Scaling => scaling of the entire application.

Microservice

Definition

Microservices are small building blocks, highly decoupled and focused on doing a small task, facilitating a modular approach to system-building.

Characteristics of a microservice

- Relatively small
- Managed by a small team (DevOps).
- Dev: can use his own technology
- Ops :
 - can be deployed independently
 - can be scaled independently
 - can be isolated in case of failure

Characteristics of a microservice

- Relatively small => Domain Driver Design
- Managed by a small team (DevOps)
- Dev: can use his own technology
- Ops :
 - can be deployed independently => CI/CD / Docker-Kubernetes
 - can be scaled independently => Kubernetes
 - can be isolated in case of failure => Service mesh

Drawback

- Complex inter-service communication mechanism:
 - Discovery is complex
 - Synchronism leads to bad performance
 - Distributed transactions are never ACID
- How implementing use cases that span multiple services ?
- Testing is more difficult

Solutions?

- Complex inter-service communication mechanism:
 - Discovery is complex => DNS / pub/sub
 - Synchronism leads to bad performance. => Reactive systems
 - Distributed transactions are never ACID. => Saga pattern
- How implementing use cases that span multiple services ?
=> API management, GraphQL
- Testing is more difficult => CI

Are they able to support an information system?

- Information Systems are:
 - heterogenous
 - decentralized
 - distributed
- Need for integration

Are they able to support an information system?

- Information Systems are:
 - heterogenous
 - decentralized
 - distributed
- Need for integration => Enterprise Integration Patterns, message broker
- Microservice architecture is just the most cloud compatible and fastest option
- Microservice architecture ≠ Service Oriented Architecture
- Domain Driven Design is thinking in terms of code
- API approach can bridge the gap between the users and the microservices
- Event Driven Architectures brings decoupling and asynchronous exchanges

Are microservices cloud native?

Virtualize microservices
=> Containers (CaaS)

Cloud compatible
app by design =>
15 factor app

Fast delivery of new releases =>
Continuous Integration /
Continuous Delivery

The gateway to the cloud =>
API management / API
gateway

Yes they can

Deploy from
scratch => IaC

Monitoring and
Control pane =>
Service mesh

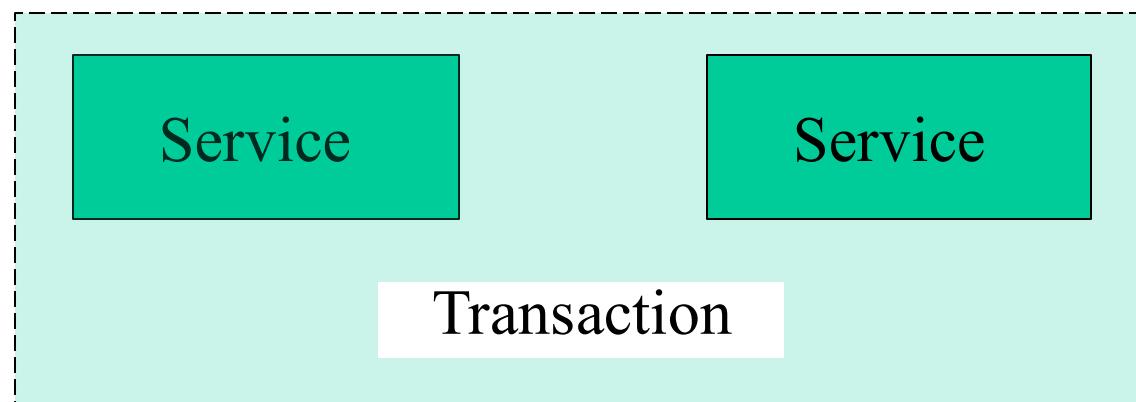
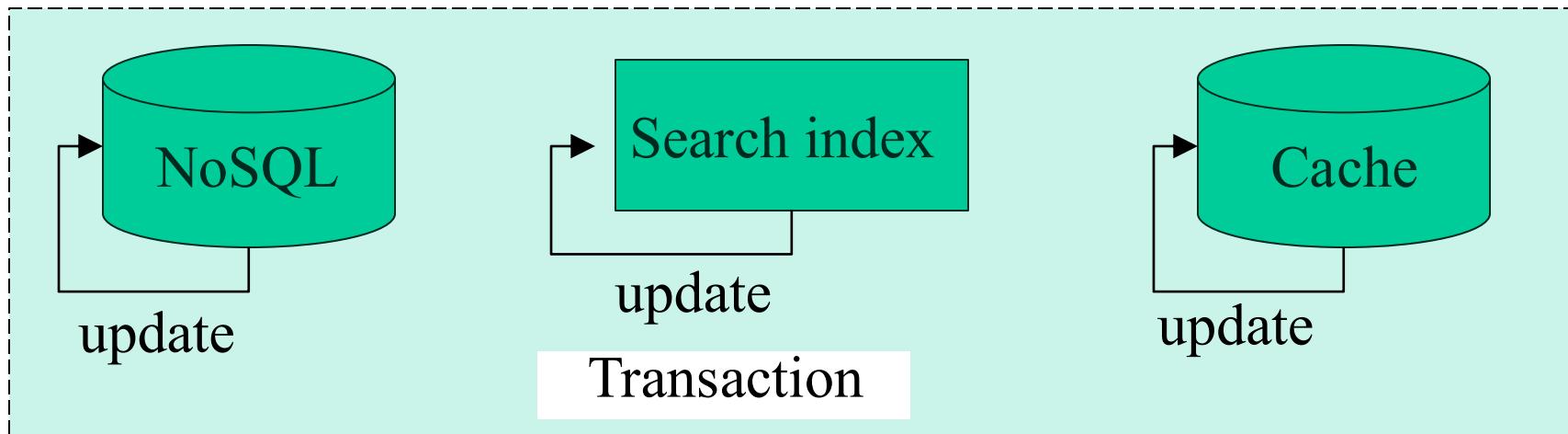
Manage containers
in a cluster =>
Kubernetes

Distributed transactions

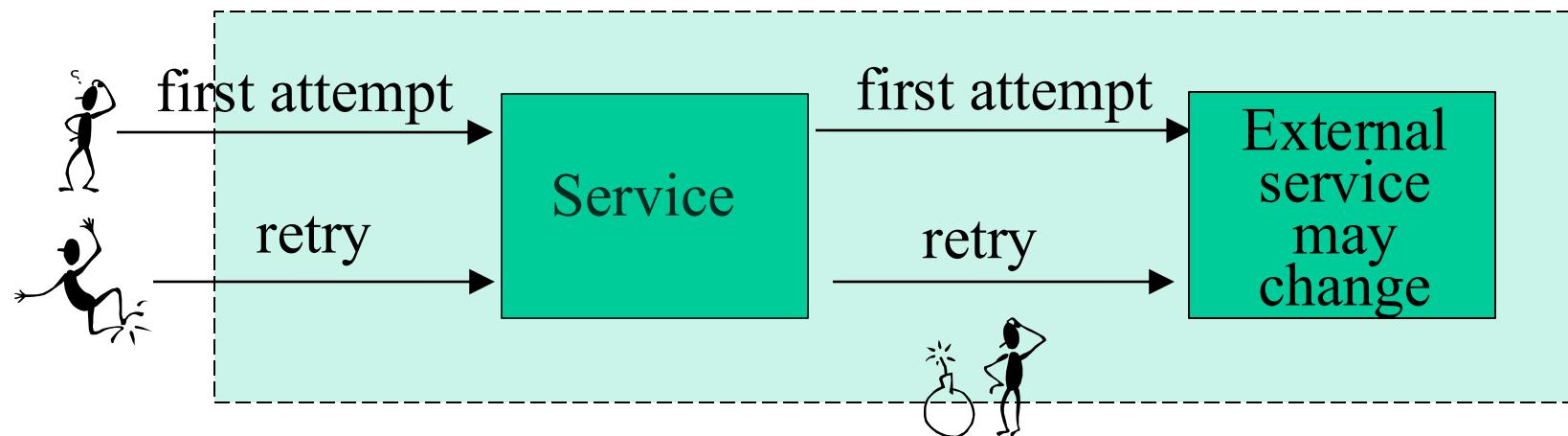
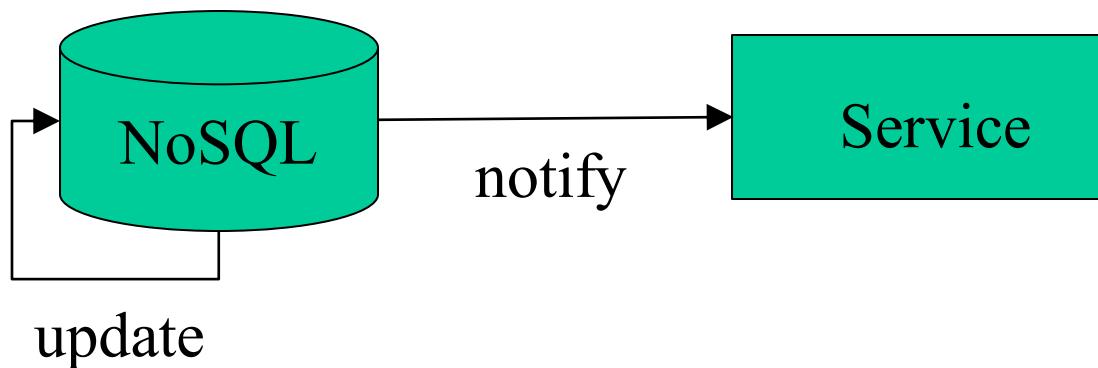
The CAP theorem

- Any distributed data store can provide only two of the following three guarantees:
 - **Consistency**: every read receives the most recent write or an error.
 - **Availability**: every request receives a (non-error) response, without the guarantee that it contains the most recent write.
 - **Partition tolerance**: the system continues to operate despite an arbitrary number of messages being dropped (or delayed).

The dual write problem

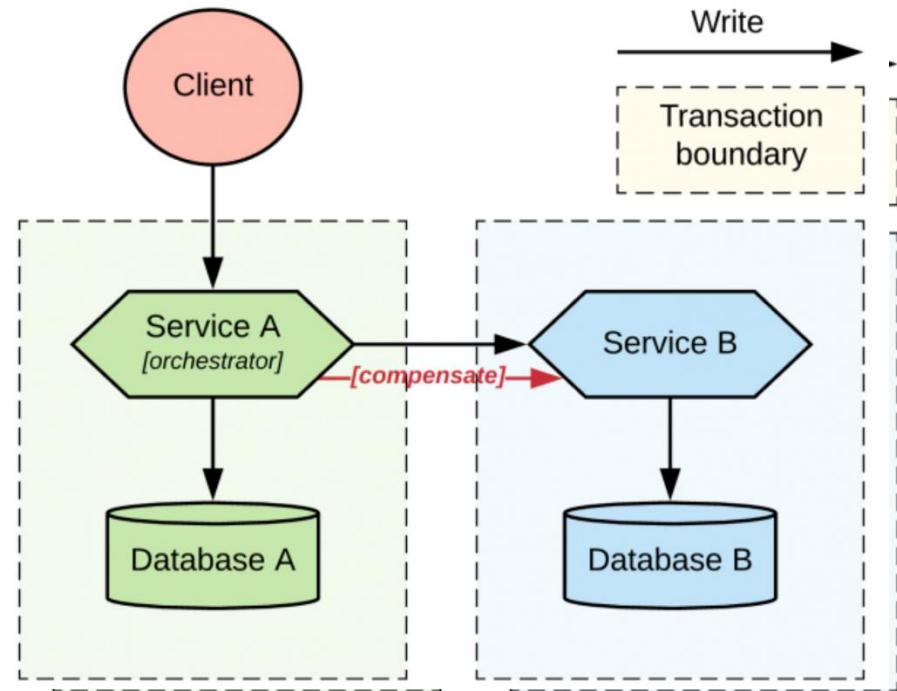


The dual write problem



Transactions with compensations

- Service A is the orchestrator.
- Service A and Service B have local transactions.



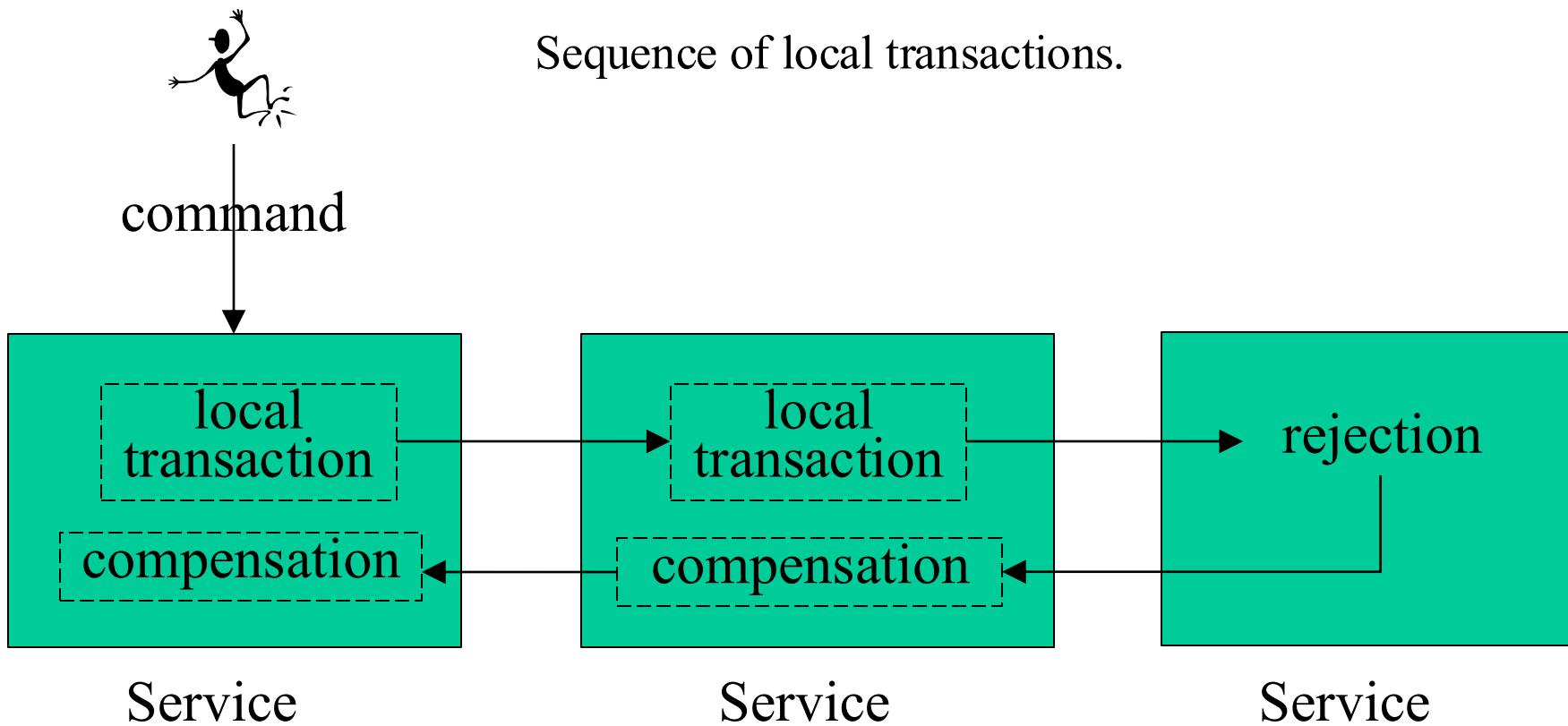
- Idempotency during compensation.
- Eventual consistency during updates.
- Recover failures during compensations?



The Saga Pattern

The saga pattern

Sequence of local transactions.



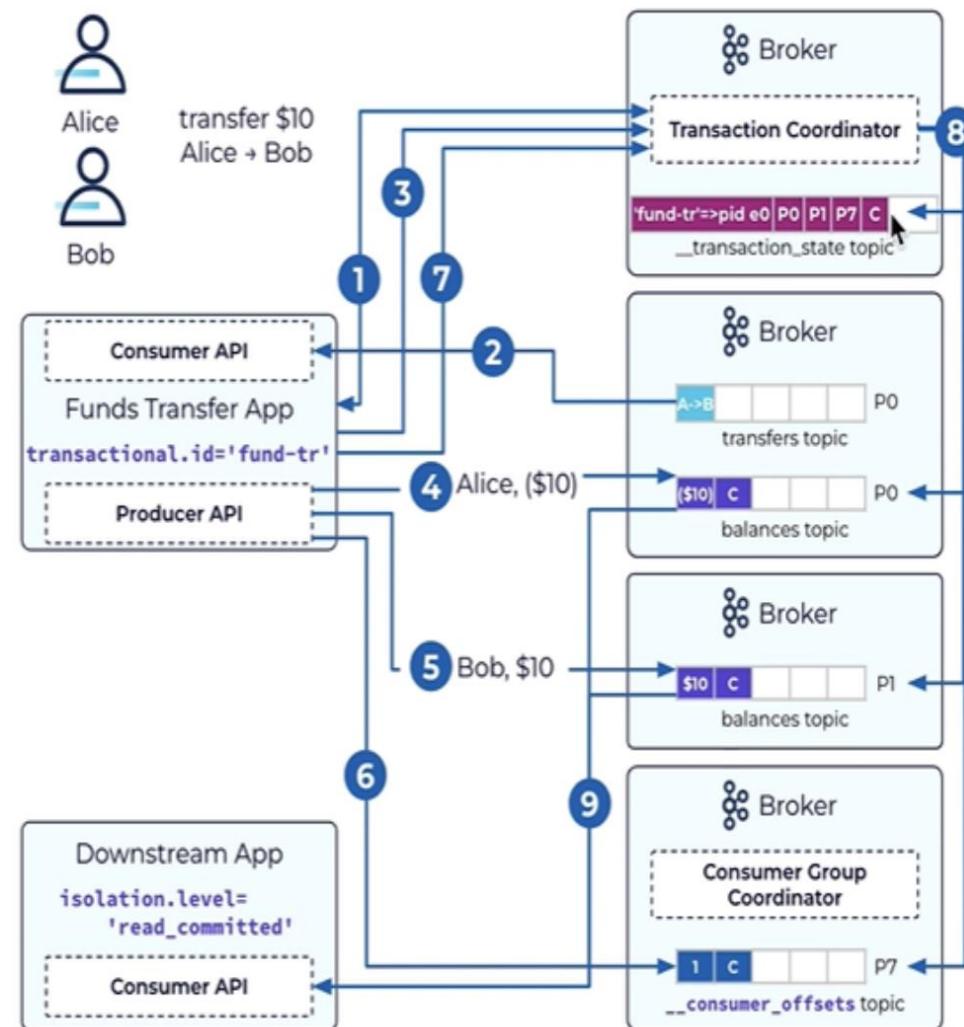
- Eventual consistency:
 - while the entry cycle continues
 - if messages fails

Saga pattern implementation

<https://github.com/charroux/servicemesh>

How transactions work in Kafka?

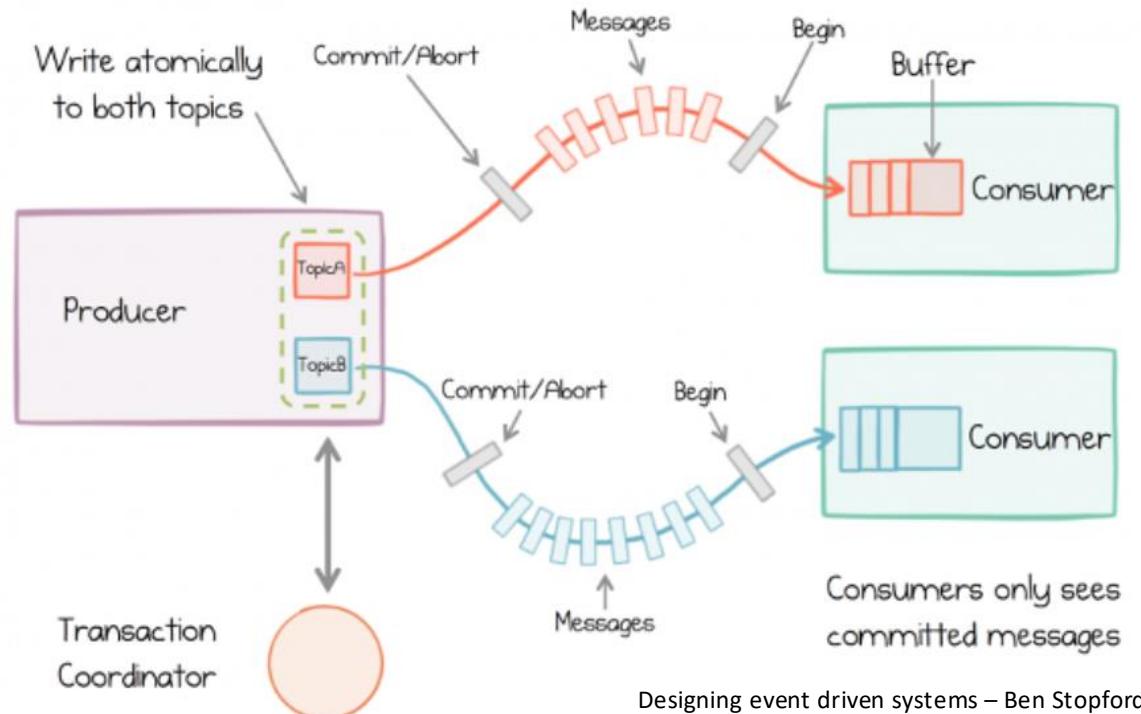
1. Requests txn ID and assigned PID and epoch
2. Event fetched by consumer
3. Notifies coordinator of partition being written to
4. Alice's account debited
5. Bob's account credited
6. Consumer offset committed
7. Notify coordinator that transaction is complete
8. Coordinator writes commit markers to p0, p1, p7
9. Downstream consumer with `read_committed` processes committed events



<https://developer.confluent.io/courses/architecture/transactions/>

How transactions work in streaming?

- Messages are buffered in each consumer so that only committed messages are visible to the consumer program:

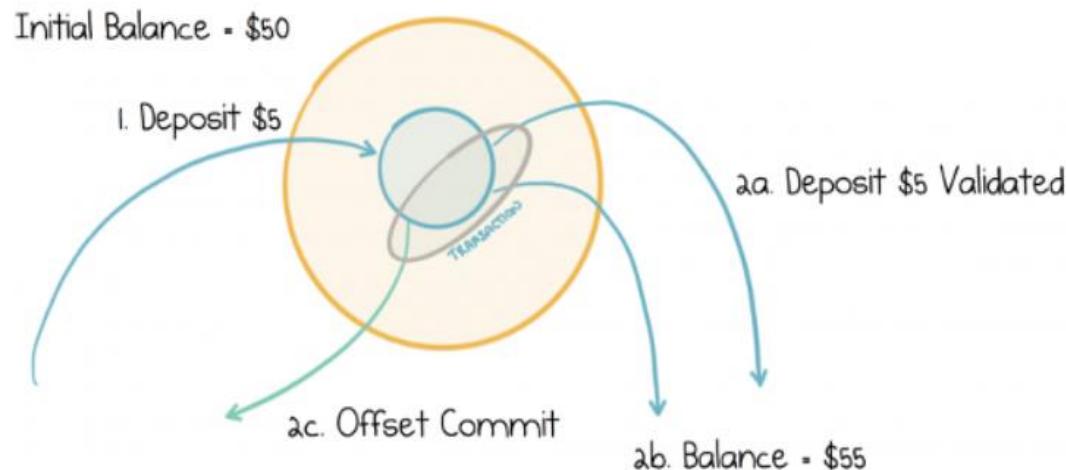


- Transactions while streaming



Transactions in Kafka

- Transactions come with guarantees:
 - Groups of messages are sent, atomically, to different topics, or none at all (the **final state** is consistent).
 - Messages sent to a single topic will never be duplicated (removed), even on failure.
 - Writes to state store and writes to output topic are atomic (included inside a transaction).



Zombie fencing

- Each transactional producer be assigned a unique identifier called the transactional.id.
- The API requires that the first operation of a transactional producer should be to explicitly register its transactional.id with the Kafka cluster. It also increments an epoch associated with the transactional.id.
- Once the epoch is bumped, any producers with same transactional.id and an older epoch are considered zombies and are fenced off, ie. future transactional writes from those producers are rejected.

Summary

- The big picture of the web services
- Web services
 - Rest web service
 - gRPC
- Message Oriented Middleware
- Microservices
- **Applications cloud native**
- API gateway / API management

Application
cloud native

Cloud native ?

The term Cloud Native is used to:

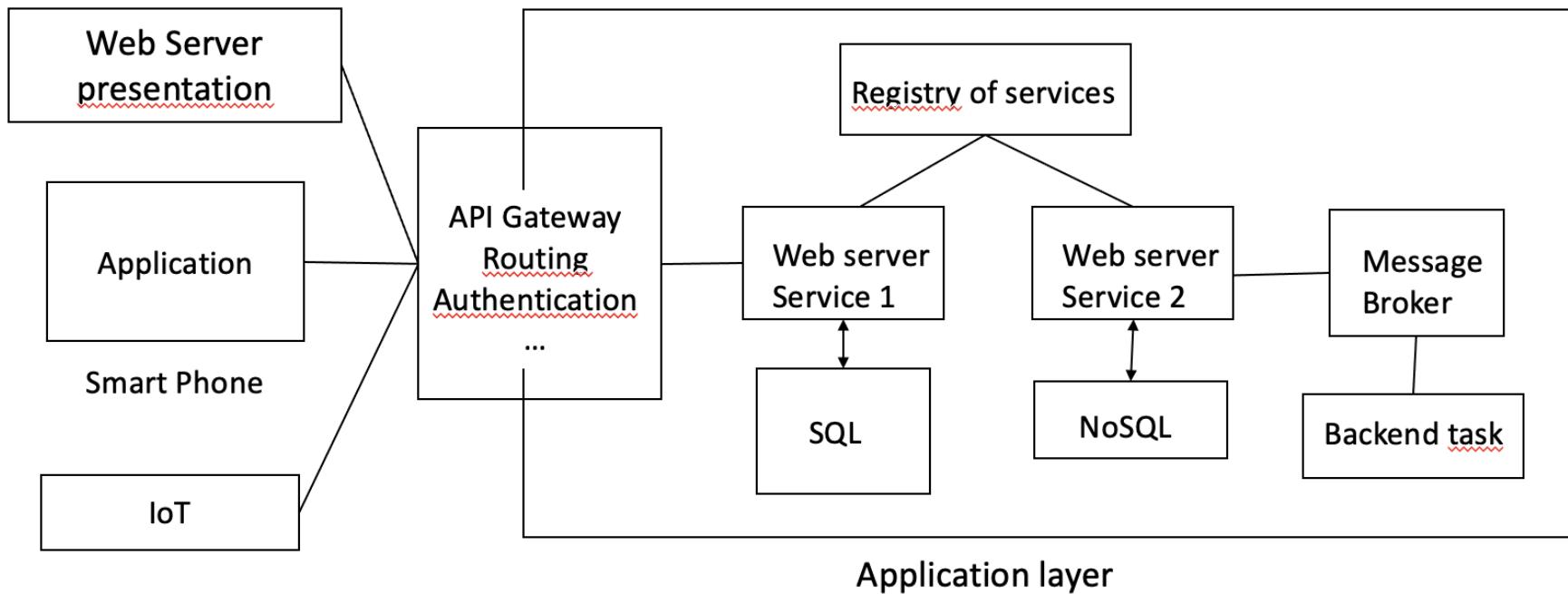
- Describe application architectures adapted to the Cloud deployment model
- Describe how applications are developed and deployed
 - The applications are distributed in container format
 - Applications are developed on a micro-services model to exploit the elasticity properties of the infrastructure
 - Applications are developed using agile methods and deployed on the infrastructure using a DevOps process
 - The DevOps process is based on CI/CD pipeline technologies
 - Applications are designed for failure: design for failure

<https://github.com/cncf/foundation/blob/main/charter.md>

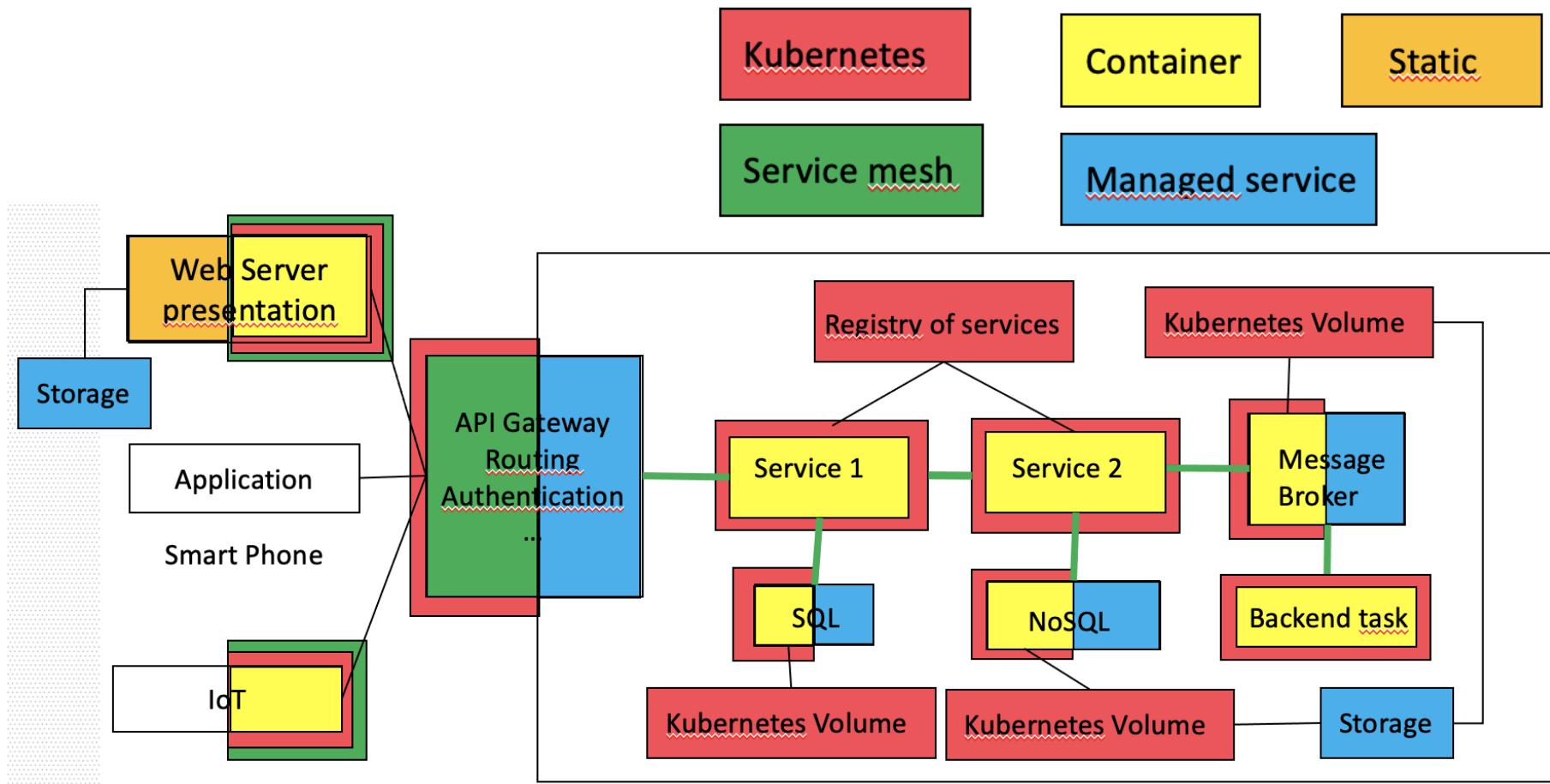


Container as a Service

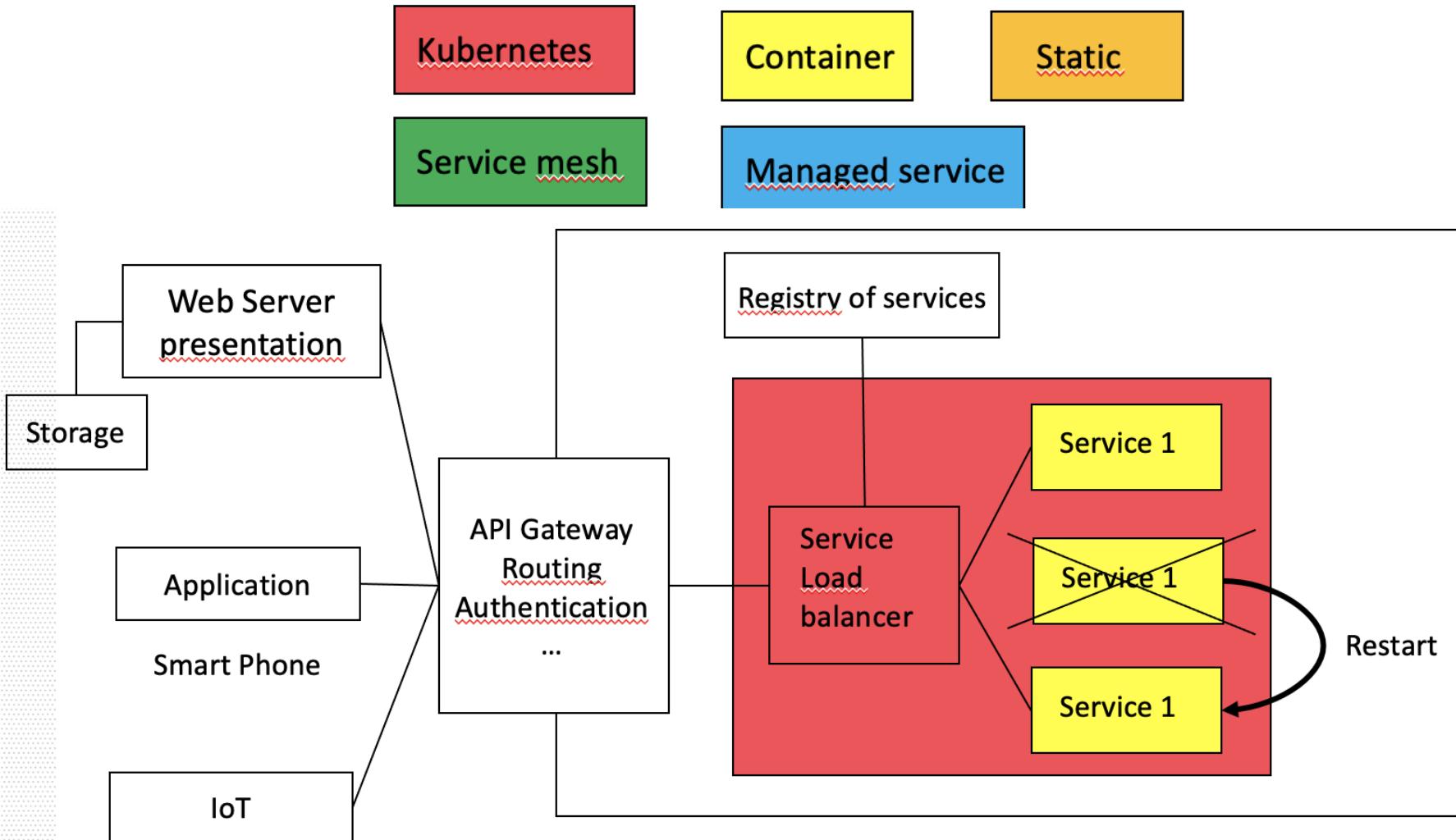
Microservices



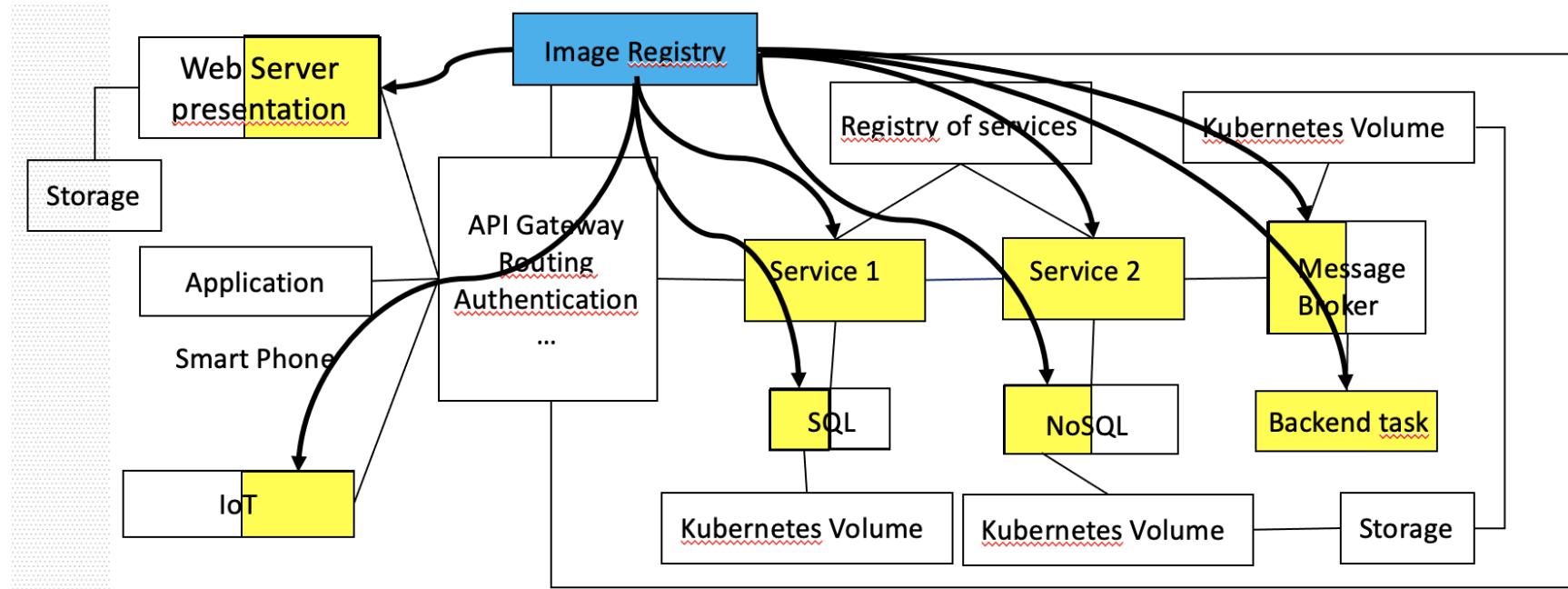
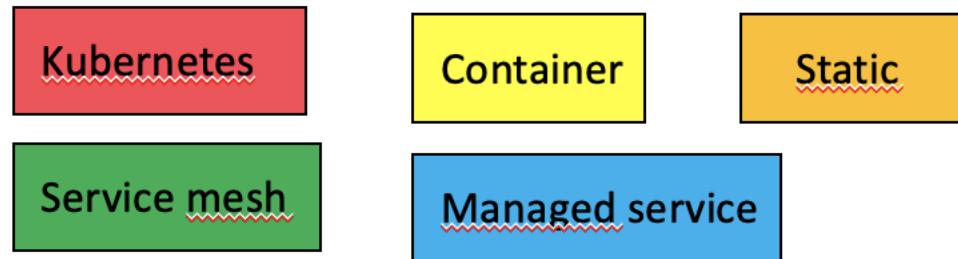
CaaS



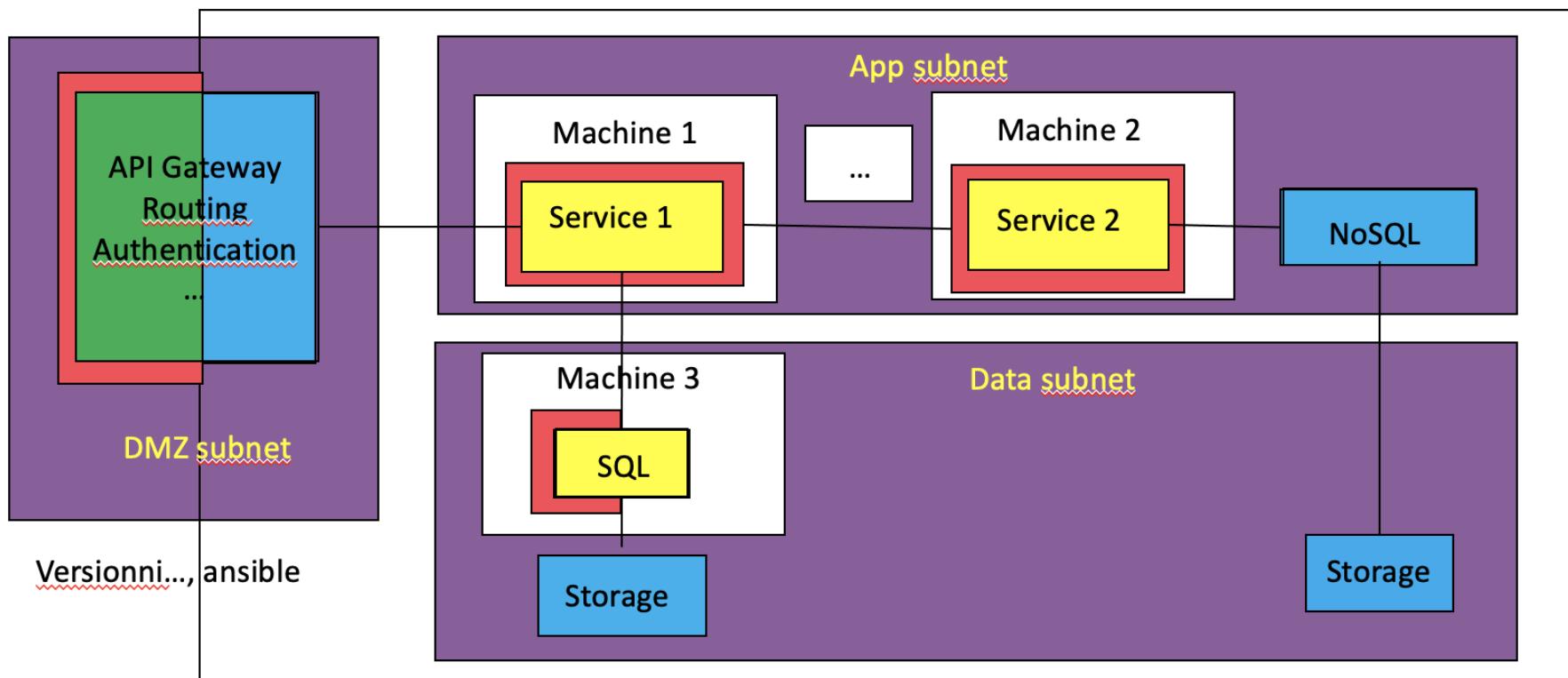
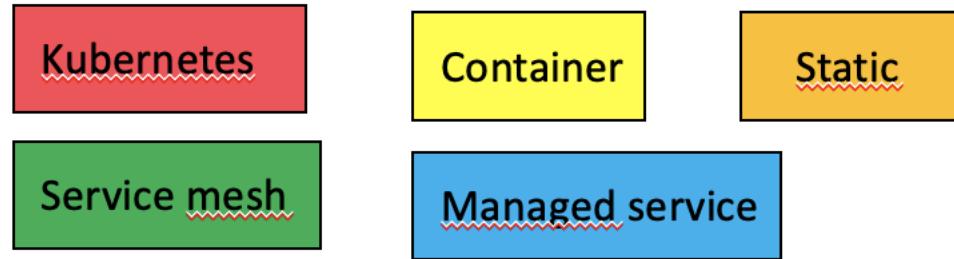
Scalling and fault tolerance



Images management



Infrastructure as Code



The twelve-factor app

The twelve-factor app

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project => Terraform, Kubernetes and Servicemesh are declaratives.
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments => Docker images
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration => Terraform, Kubernetes and Service mesh are supported by all clouds
- Minimize divergence between development and production, enabling continuous deployment for maximum agility => CI/CD pipeline, containers vs managed services
- And can scale up without significant changes to tooling, architecture, or development practices => Kubernetes

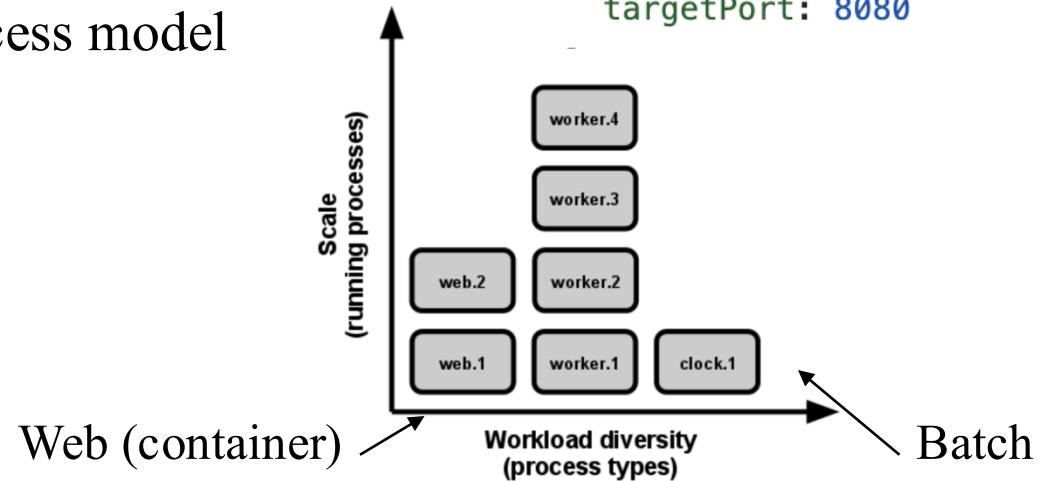
The twelve-factor app

- One codebase tracked in revision control, many deploys: **Git**
- Explicitly declare and isolate dependencies: **package manager**
- Store config in the environment (environment variables) not in the code, this includes:
 - Resource handles to the database, Memcached, ...
 - Credentials to external services such as Amazon S3 or Twitter
 - Per-deploy values such as the canonical hostname for the deploy
- => **Kubernetes**
- Treat backing services as attached resources
 - Swap out a local MySQL database or services with one managed by a third party without coding => Kubernetes volumes
- Execute the app as one or more stateless processes
 - processes are stateless and share-nothing. Any data that needs to persist must be stored in a stateful backing service, typically a database ????

The twelve-factor app

- Export services via port binding
- Scale out via the process model

```
apiVersion: v1
kind: Service
metadata:
  labels:
    run: carstat
  name: carstat
spec:
  ports:
    - name: grpc-web
      port: 8080
      targetPort: 8080
```

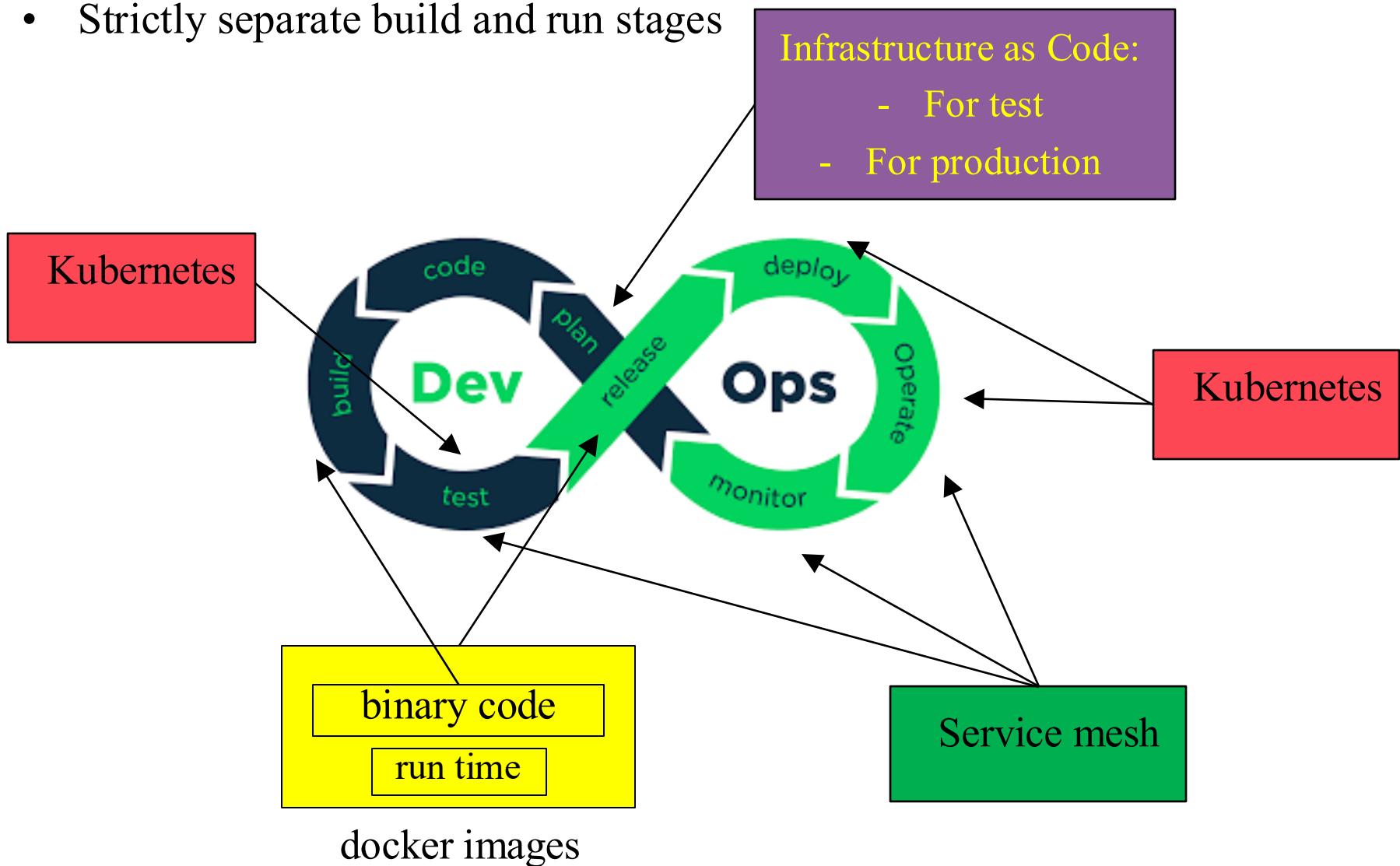


The twelve-factor app

- Maximize robustness with fast startup and graceful shutdown
 - Docker containers start instantly because they don't contain a full OS
 - Kubernetes restarts failed containers
- Keep development, staging, and production as similar as possible
 - Kubernetes can be used as virtual (Dev) or real machines (production)
 - Kubernetes claim can search for volumes at starting (local, NTFS, cloud...)
 - Kubernetes can change containerized database (for dev) to cloud managed database (for prod)
- Treat logs as event streams (Sequence of business events for indexing and analysis):
 - During local development, the developer view the logs in stdout
 - In production, each process' log is routed to a long-term archival.
- Run admin/management tasks as one-off processes (running database migrations...)
 - Admin code must ship with application code to avoid synchronization issues.
 - The same dependency isolation techniques should be used on all process types.

The twelve-factor app

- Strictly separate build and run stages

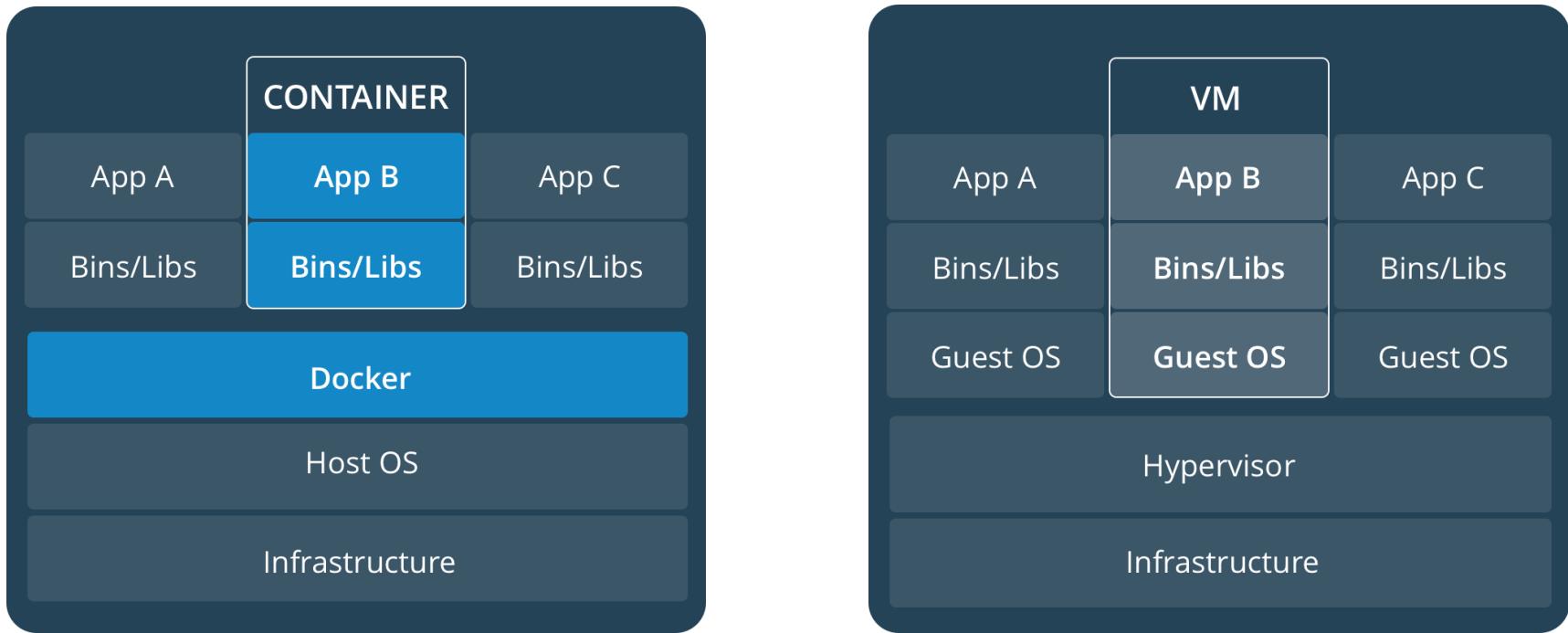


3 additional factors

- Telemetry and real-time app monitoring
 - Monitoring, tracing, logging
 - Service mesh for example
- Authentication and authorization
 - Service mesh:
 - Authentication through the gateway
 - mTLS
 - ...
 - DevSecOps
 - Secret management
- API first

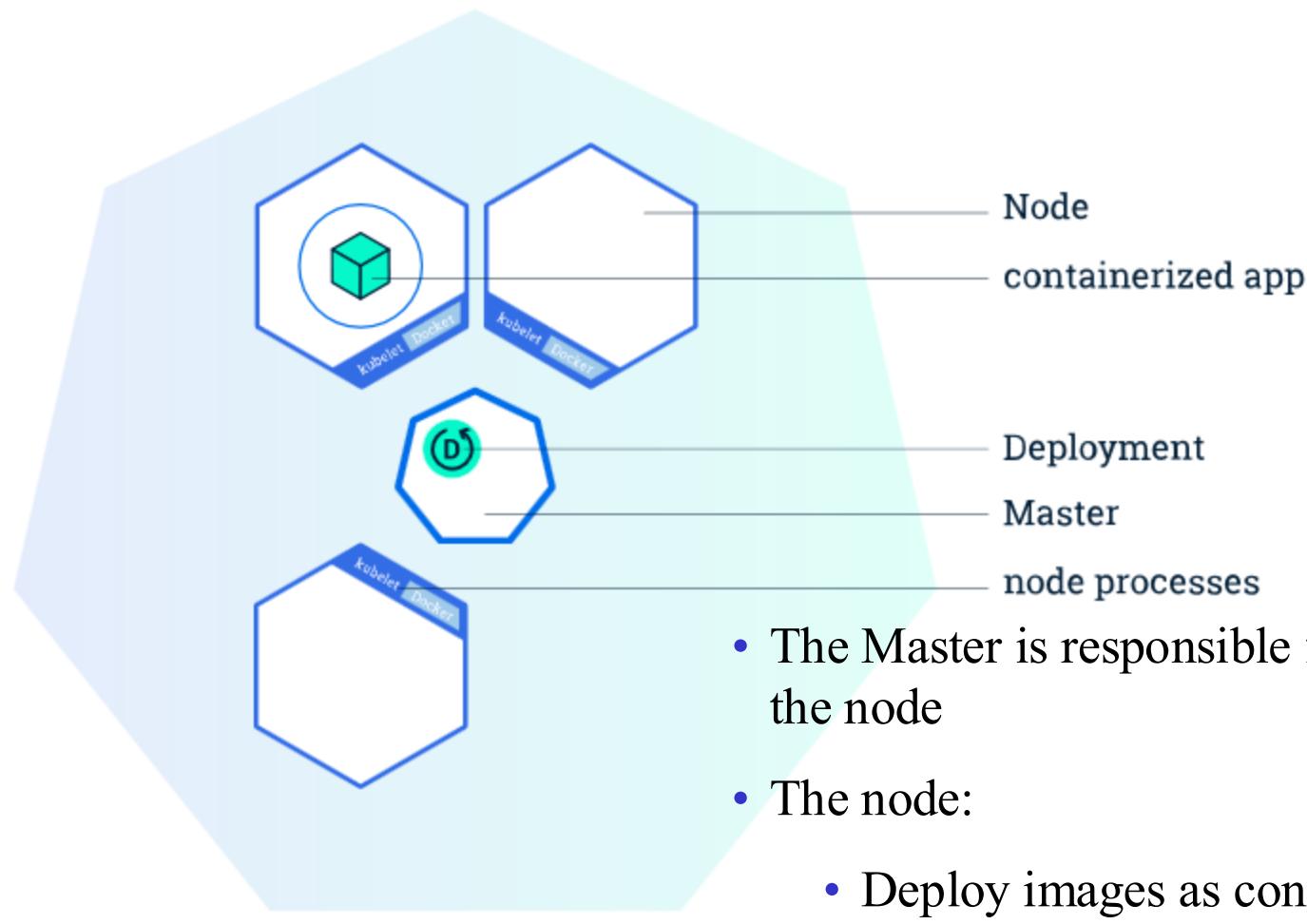
What is Kubernetes?

Docker containers vs Virtual Machines



- Multiple containers share the OS kernel.
- Containers take less space than VMs.
- Containers start almost instantly.

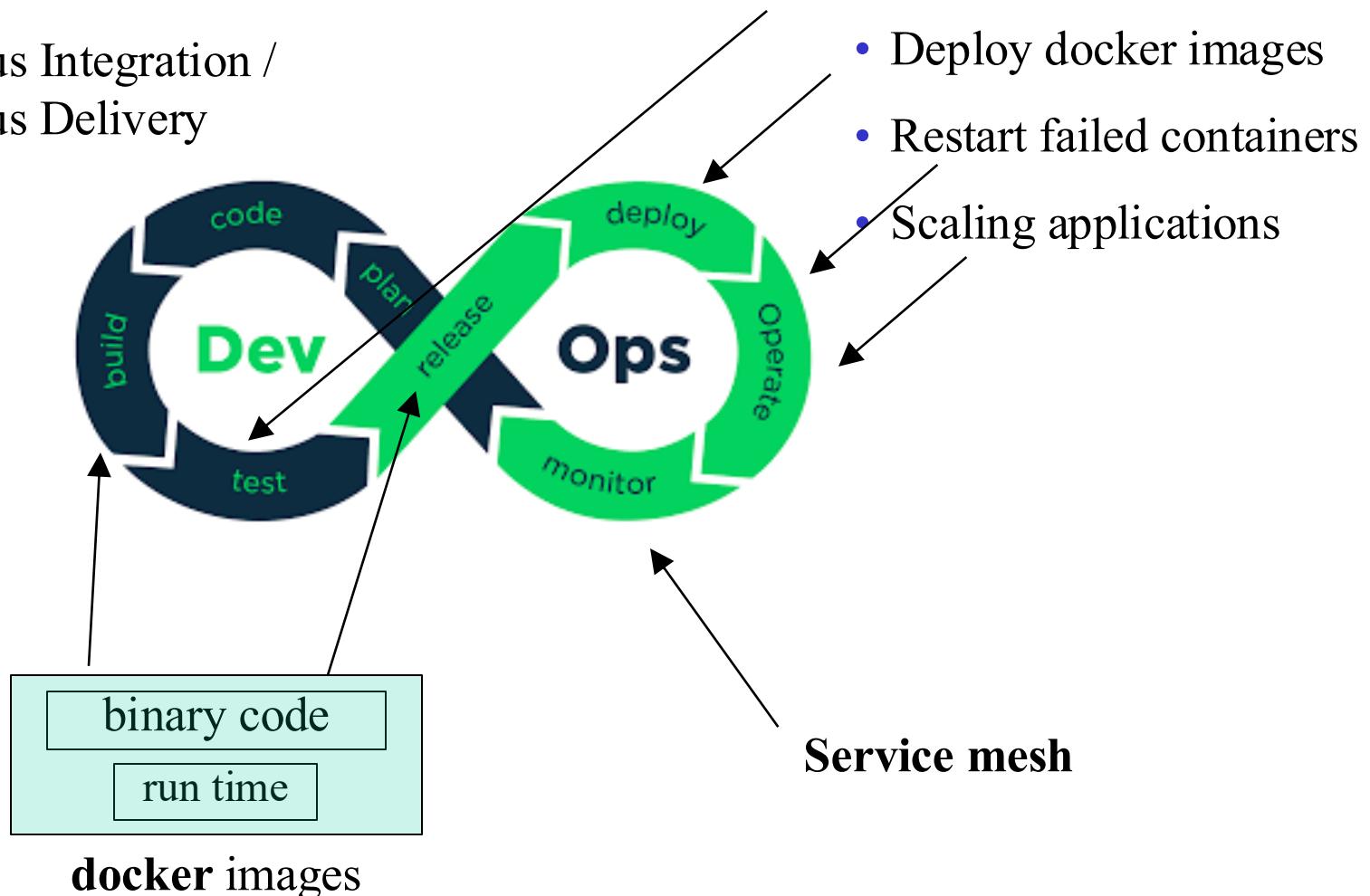
The Kubernetes cluster



- The Master is responsible for selecting the node
- The node:
 - Deploy images as containers
 - Maintain applications state
 - Scaling applications

Kubernetes and DevOps

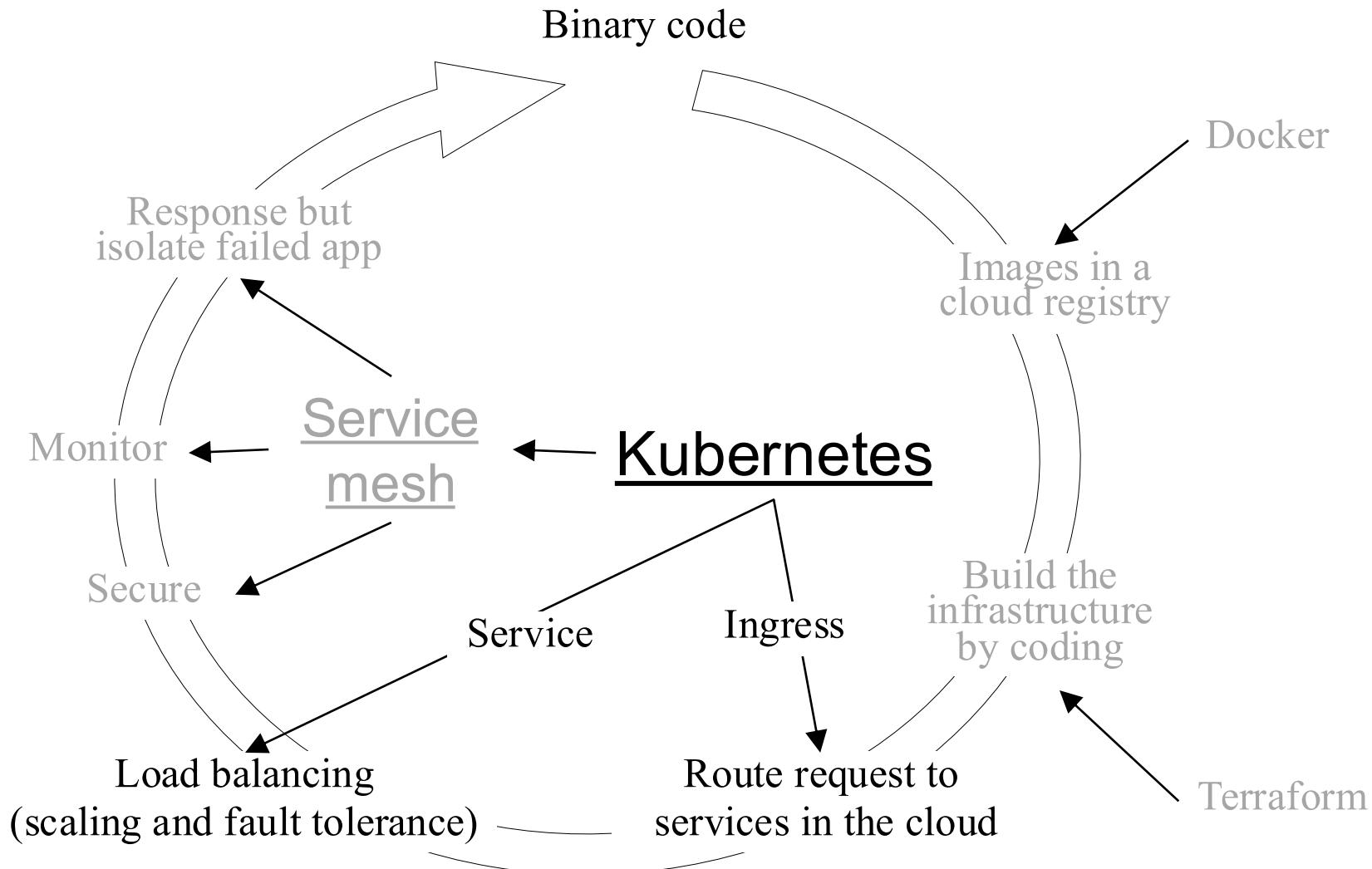
- 24/7 availability
- Continuous Integration / Continuous Delivery



Kubernetes

- Kubernetes is open source
- Versions:
 - k8s
 - k3s: lightheaded solution for edge computing, IoT, Raspberry Pi...
- Turnkey solutions (IaaS providers with a few command):
 - Google cloud
 - AWS
 - ...

Cloud native applications



Best practise for databases and containers

- Keep the current mode of operation for relational databases (HA, replication, backup, etc.)
- Integrate stateless parts of applications on your CaaS while maintaining current database environments
- Create containers with the database engine when relevant (low performance, low volume of data, non-production environment, micro-services development, etc.) and place the data on persistent volumes
- Segment the BDDs during the development in micro-service of the associated functionality
- Upgrade to Cloud Native database engines (sharding, etc.)
- Depending on the level of adoption of IaaS or public cloud technologies, evaluate DBaaS solutions

Helm



- An ideal tool for managing Kubernetes applications should:
 - Support YAML file templating:
 - Possible to customize the files by Variables => useful for Update, RollBack and Scaling
 - Support dependency management:
 - Allow the tool to install the target application, and all these dependencies simply
 - Ease of publishing applications and their dependencies
 - Offer a repeatable and consistent deployment model

Kubernetes in practice

<https://github.com/charroux/servicemesh>

Service Mesh

Why service mesh?

The more Kubernetes applications grow,
the more service to service communication is difficult.

What is service mesh?

Service mesh

- Service mesh manages network traffic between services at the platform layer instead of the application layer.



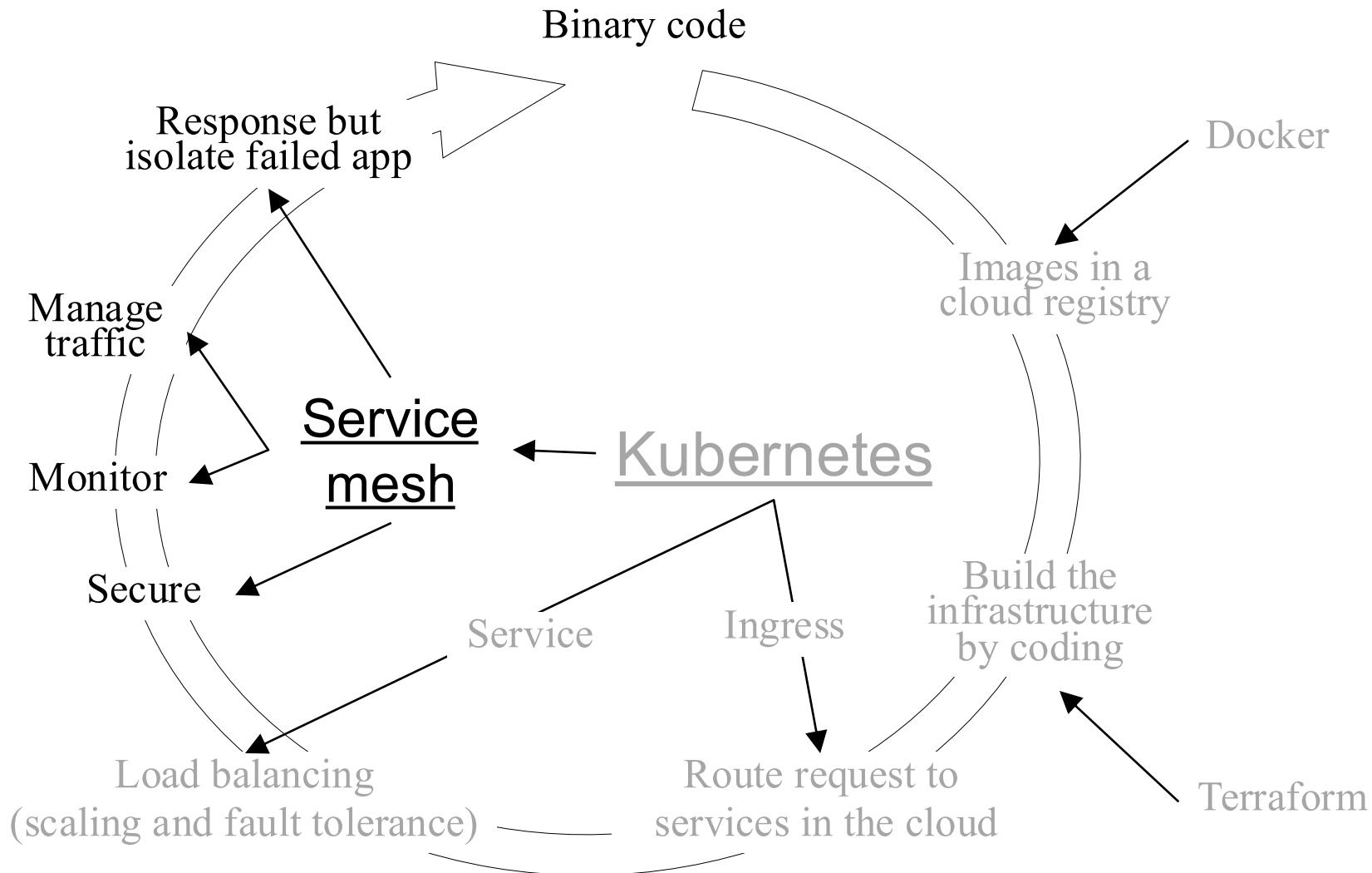
Open service mesh

AWS App Mesh

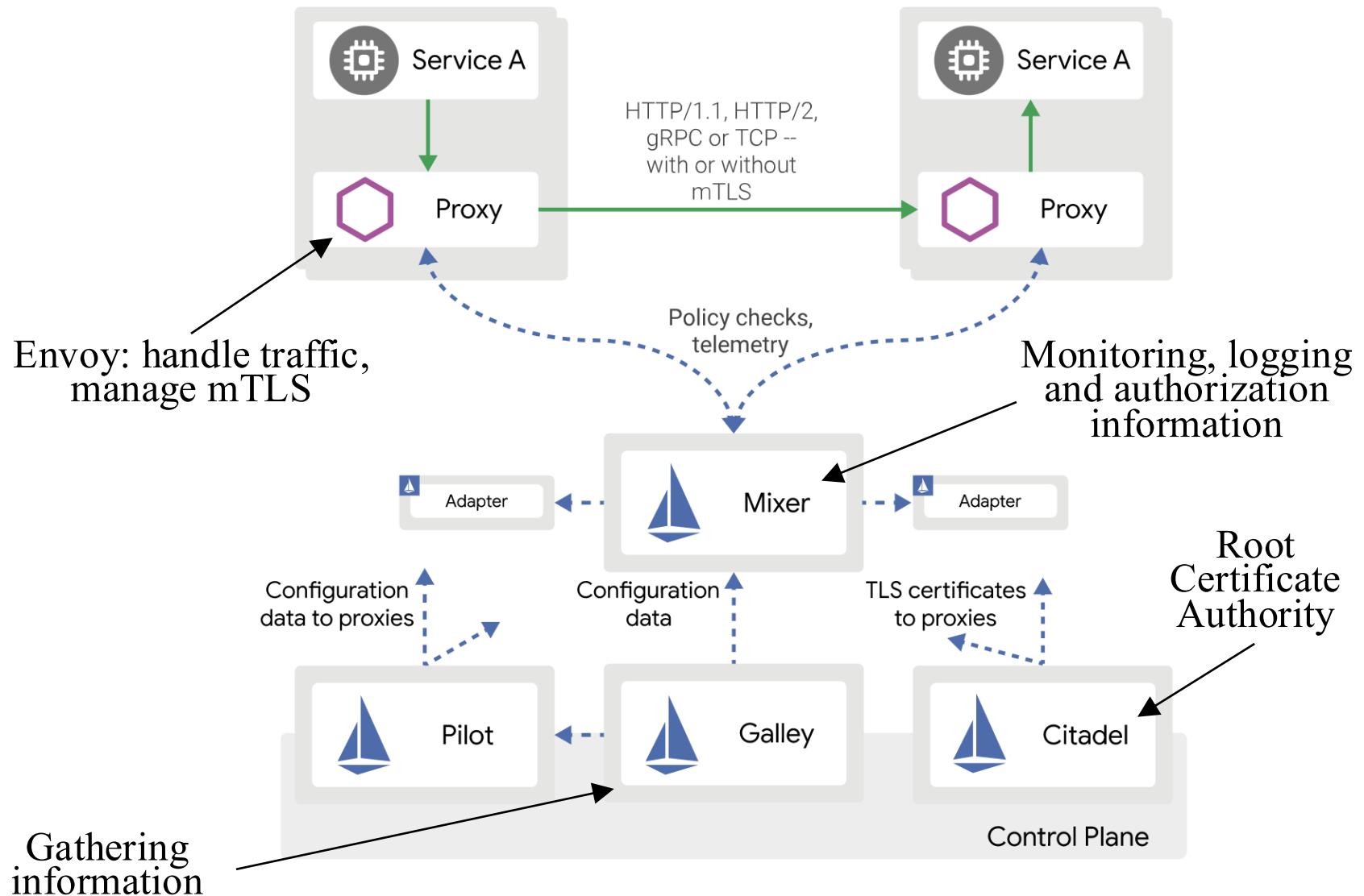
Anthos Service Mesh



Cloud native applications



The example of Istio



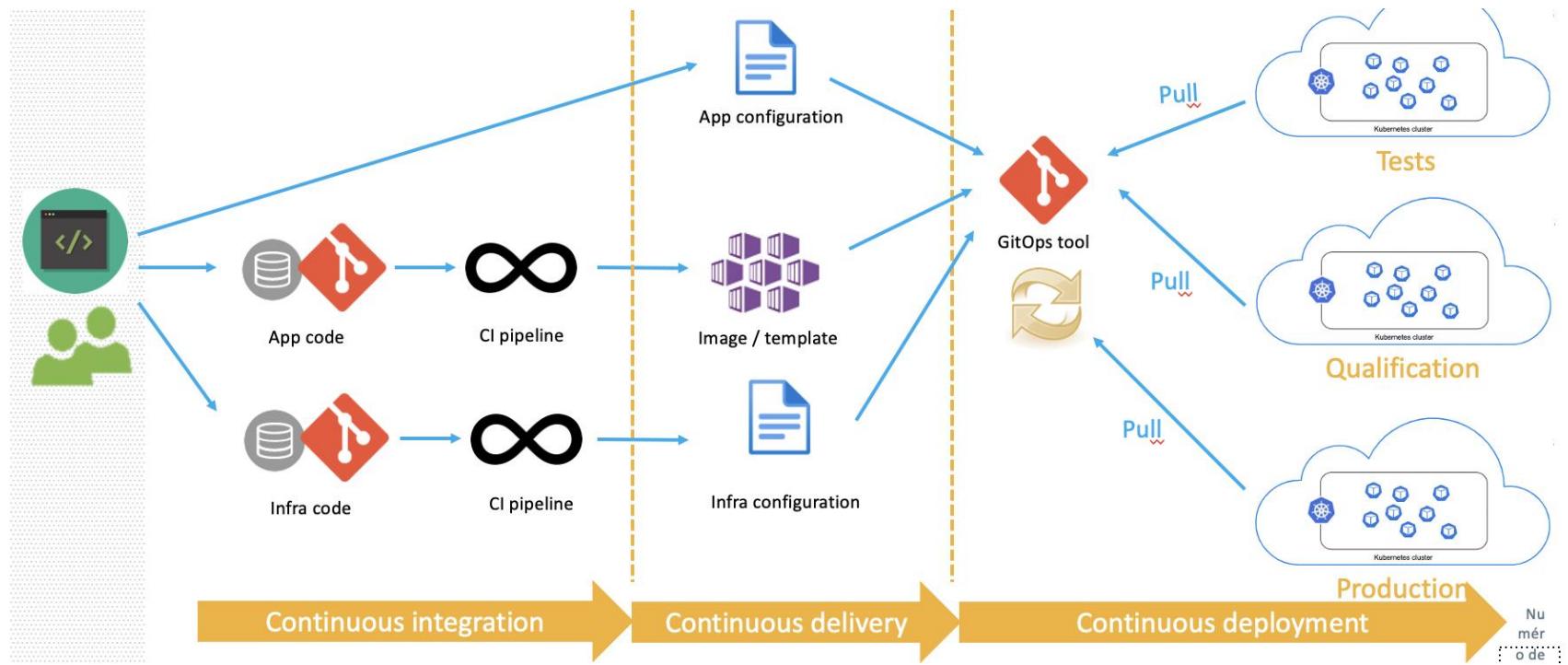
Service mesh in practice

<https://github.com/charroux/servicemesh>

Continuous Integration with Kubernetes

<https://github.com/charroux/servicemesh#continuous-integration-with-github-actions>

Continuous Delivery / Deployment with GitOps



Summary

- The big picture of the web services
- Web services
 - Rest web service
 - gRPC
- Message Oriented Middleware
- Microservices
- Applications cloud native
- **API gateway / API management**

API Gateway / API Management

Why API Management?

Microservices

- Complex inter-service communication mechanism:
 - Discovery is complex => DNS / pub/sub
 - Synchronism leads to bad performance. => Reactive systems
 - Distributed transactions are never ACID. => Saga pattern, Event Sourcing
- How implementing use cases that span multiple services ?
=> API management, GraphQL
- Testing is more difficult

API and reactive systems

- Responsive mains:
 - responds in a timely manner
 - problems are detected (and fixed quickly)

=> Service mesh
- Elastic under varying workload:
 - Scalability

=> Stateless services, Kubernetes

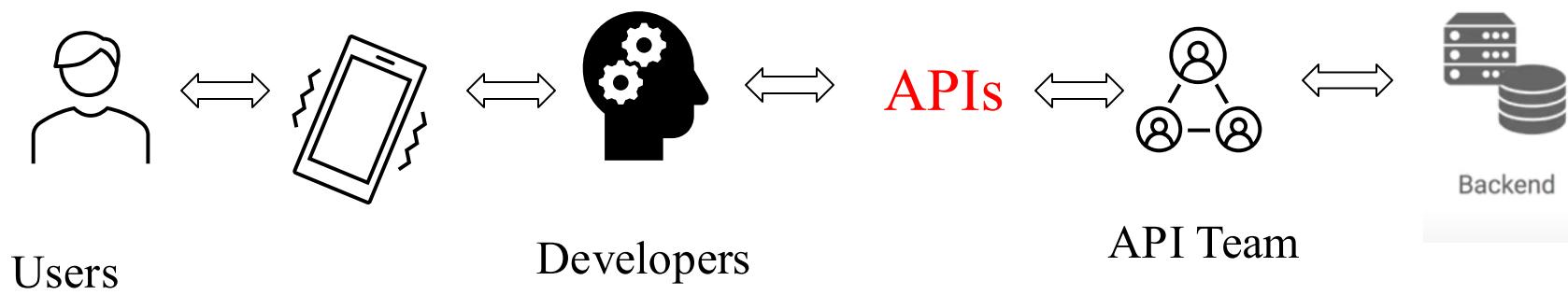
 - Live performance measure

=> API Management / Service mesh

API and reactive systems

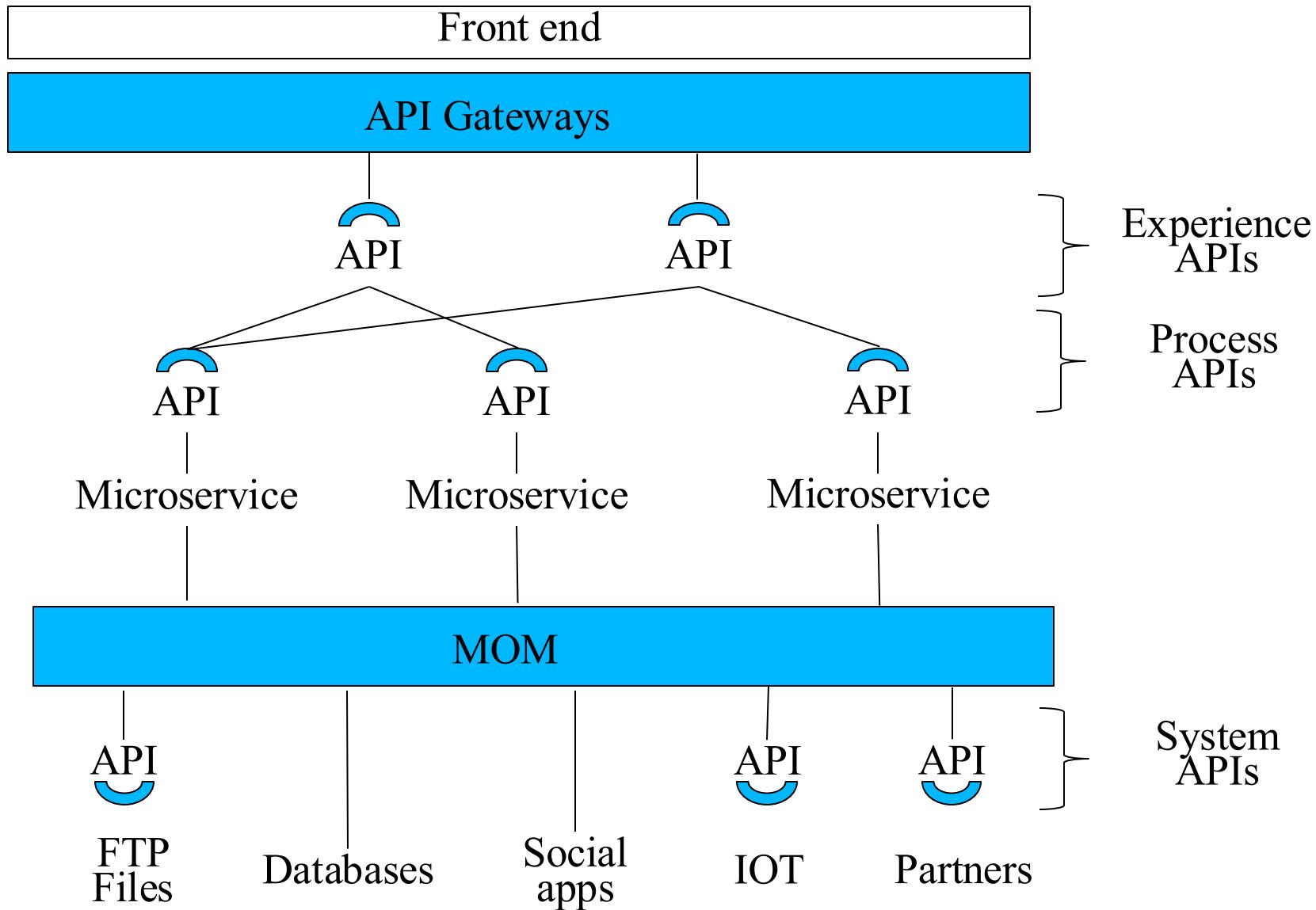
- Resilient is achieved by:
 - Replication (executing a component simultaneously in different places)
=> Routing with API gateway
 - Isolation (decoupling, both in time and space)
=> Message brokers
 - Delegation (the execution of a task will take place in another component)
 - Multitasking => Reactive programming
 - Different networks => Routing with API gateway
 - ...
 - Contained to each component => Circuit breaker

Users view



Where APIs are needed?

Asynchronous pub/sub Synchronous http rest



The API-first approach

What is API-first?

- The API-first approach prioritizes APIs at the beginning of the software development process.
- API-first organizations develop APIs before writing other code.
- This lets teams construct applications with internal and external services that are delivered through APIs.

API-first companies

- API-first companies answer yes to all the following questions:
 - Do you have APIs to operate most of your data?
 - Do you make APIs available to your customers and partners?
 - Do you know how to organize and discover your APIs?
 - Do you have standardized processes to build APIs?
 - Do your APIs meet regulatory requirements?
 - Do you know the security risk to your API perimeter?

Private, partner, and public APIs

- 58 percent of the APIs that developers work with are for internal use only.
- Partner APIs constitute 27% of organizations' APIs.
- Public APIs, are about 15% percent of organizations' APIs.
- 51% of developers say that more than half of their organizations' development effort is spent on APIs.

Postman's 2022 State of the API report

Stage 1 of API lifecycle: define

- **Team members:** who is responsible for each part of producing an API.
- **Team workspace:** Setting up a dedicated workspace for each API..
- **Public workspace:** Some APIs will benefit from establishing a public workspace where third-party consumers can learn about an API.
- **GitHub repository:** Having a dedicated Github repository for each API.

Stage 2 of API lifecycle: design

- **Determine what the API is intended to do:**
 - All stakeholders agree on the API's business use case.
 - Different architecture for different use cases (gRPC-based architecture might make the most sense for an API that connects internal microservices, while a GraphQL API would be well-suited for a service that relies on disparate data sources).
 - Stakeholders should clearly outline their goals for the API by describing—in natural language—exactly how it will meet specific needs.

Stage 2 of API lifecycle: design

- **Define the API contract with a specification**
 - which resources are required? How their data should be formatted and structured? How they should relate to one another? Determine the desired levels of abstraction and encapsulation in your API.
 - These decisions should be captured in an API definition with specifications, such as OpenAPI and AsyncAPI.
- **Validate your assumptions with mocks and tests**
 - Generate mock servers.
 - API tests within CI/CD pipelines.
- **Document the API**
 - Key details about every resource, method, parameter, and path.
 - Include examples of API requests and responses

Stage 3 of API lifecycle: develop

Stage 4 of API lifecycle: test

- **Contract testing:**
 - Contract tests can be derived from API artifacts like OpenAPI, JSON Schema, and GraphQL, and used to ensure that API conform specifications.
- **Performance testing:**
 - measuring the time each path takes for the response to be sent.

Stage 5 of API lifecycle: secure

- **Authentication**
- **Security testing:** testing for common OWASP vulnerabilities and other custom scenarios or business approaches

Stage 6 of API lifecycle: deploy

- **CI/CD pipeline**
- **API Gateway**

Stage 7 of API lifecycle: observe

- **Contract testing monitor:** ensures that each individual API contract is never breaking with consumers and each API is delivering as expected on a 24/7 schedule.
- **Performance testing monitor:** ensures that each individual API SLA is met, identifying API performance issues early on and before there is significant business impact.
- **Security testing monitor:** ensures that each individual API is regularly certified as being secured.
- **Activity:** observability into when APIs, mock servers, documentation, testing, monitors, and other critical elements of API operations are changed or configured.
- **Changelog:** detail history of the changes.

Stage 8 of API lifecycle: distribute for delivery

- **Search:** Ensuring APIs are always available.
- **Private API network:** allow organizations, groups, and teams to establish private catalogs of APIs.
- **Public API network:** allow for teams, APIs, collections, and other elements to be made available for searching and browsing by a public audience.

Modeling APIs

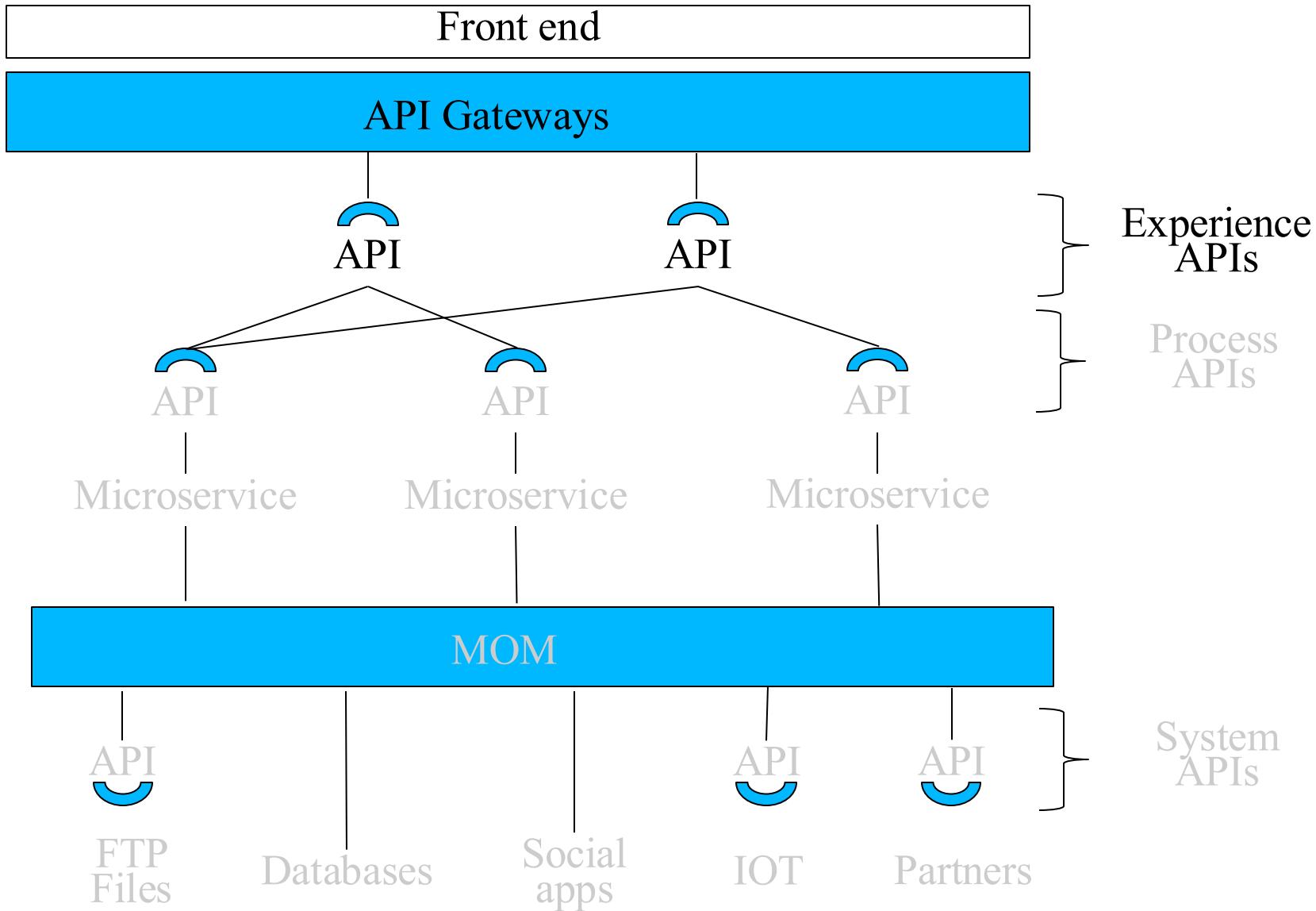
GraphQL

A query Language for an API



Where is GraphQL?

Synchronous http rest



Model the business domain as a graph: API first

```
"rentalAgreements":  
[  
    {  
        "id": "1",  
        "customer": {  
            "id": "1",  
            "name": "Tintin"  
        },  
        "cars": [  
            { "plateNumber": "11AA22", "brand": "Ferrari"  
            },  
            { "plateNumber": "22BB33", "brand": "Porsche" }  
        ]  
    }  
]
```

Query schema definition

```
"rentalAgreements":
```

```
[
```

```
{
```

```
    "id": "1",
```

```
    "customer": {
```

```
        "id": "1",
```

```
        "name": "Tintin"
```

```
    },
```

```
    "cars": [
```

```
        { "plateNumber": ...},
```

```
        { "plateNumber": ...}
```

```
    ]
```

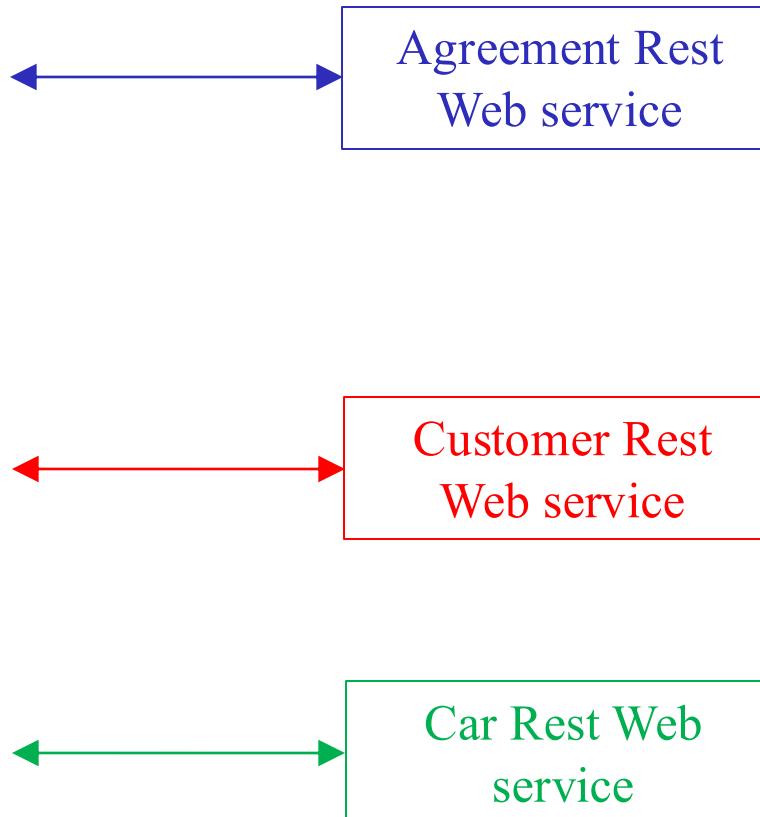
```
}
```

```
]
```

```
type Query {  
    rentalAgreements: [RentalAgreement]  
}  
type RentalAgreement {  
    id: ID  
    customer: Customer  
    cars: [Car]  
}  
type Customer {  
    id: ID  
    name: String  
}  
type Car {  
    plateNumber: ID!  
    brand: String!  
    price: Int!  
}
```

Mapping with services

```
type Query {  
    rentalAgreements: [RentalAgreement]  
}  
  
type RentalAgreement {  
    id: ID  
    customer: Customer  
    cars: [Car]  
}  
  
type Customer {  
    id: ID  
    name: String  
}  
  
type Car {  
    plateNumber: ID!  
    brand: String!  
    price: Int!  
}
```



Code (here in Java)

```
type RentalAgreement {  
    id: ID  
    customer: Customer  
    cars: [Car]  
}
```

<=>

```
public class RentalAgreement {  
  
    long id;  
    Customer customer;  
    Collection<Car> cars = new ArrayList<>();  
}
```

GraphQL

Java

Coding queries in Java (Spring boot)

```
type RentalAgreement {
    id: ID
    customer: Customer
    cars: [Car]
}

    @QueryMapping()
    public Iterable<RentalAgreement> rentalAgreements() {
        ... http request to a back service
    }

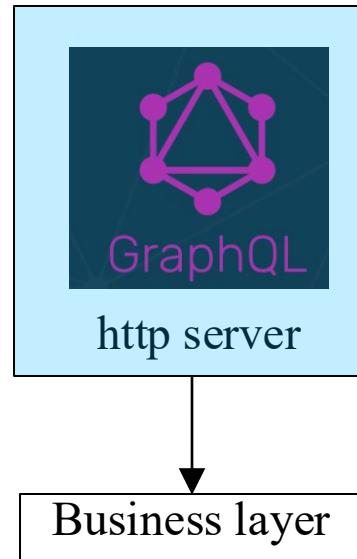
    @SchemaMapping
    public Collection<Car> cars(RentalAgreement rentalAgreement) {
        ... http request to a back service
    }

    @SchemaMapping
    public Customer customer(RentalAgreement rentalAgreement) {
        ... http request to a back service
    }
```

Serving over HTTP

- A query (always an http post) returns data and some in JSON

```
{  
    "data": { ... },  
    "errors": [ ... ]  
}
```



Load balancing necessary to
avoid congestion



Make queries and get exactly what you want

```
query allRentalAgreementsWithDetails {  
    rentalAgreements {  
        id  
        customer {  
            id  
            name  
        }  
        cars {  
            plateNumber  
            brand  
        }  
    }  
}
```

```
query allRentalAgreements {  
    rentalAgreements {  
        customer {  
            name  
        }  
        cars {  
            brand  
        }  
    }  
}
```

Limit the number of requests => only 2 queries !

```
query {  
    books {  
        id @QueryMapping  
        name public Collection<Book> books() {  
            ratings {  
                rating return bookCatalogService.getBooks();  
            }  
            comment  
        }  
    } @BatchMapping  
}  
public Map<Book, List<Rating>> ratings(List<Book> books) {  
    return ratingService.ratingsForBooks(books);  
}
```

Stay tuned at the client side (1/2)

- Define the subscription

```
type Subscription {  
    notifyNewCarPrice (plateNumber: ID): Car  
}
```

- Implement the subscription

```
@SubscriptionMapping  
public Flux<Car> notifyNewCarPrice(@Argument String plateNumber) {  
    return Flux.fromStream(  
        Stream.generate(() -> {  
            ...  
            car.setPrice(500);  
            return car;  
        }));  
}
```

Stay tuned at the client side (2/2)

- Subscribe

```
subscription {  
    notifyNewCarPrice(plateNumber: "AA11BB") {  
        brand  
        price  
    }  
}
```

Mutation

- Modify server-side data

```
extend type Mutation {  
    rentCars(request: RentCarsRequest!): RentCarsResponse
```

```
}
```

```
input RentCarsRequest {  
    customerId: ID!  
    numberOfCars: Int  
}
```

```
type RentCarsResponse {  
    customerID: ID!  
    rentalAgreementId: Int  
    state: String  
}
```

```
@MutationMapping  
public RentCarsResponse rentCars(  
    @Argument RentCarsRequest request  
)  
{  
    ... http request to backend  
}
```

GraphQL vs Rest

	GraphQL	REST
Performance	Fast	Multiple network calls
Queries	No Over-Fetching or Under-Fetching	Over-Fetching or Under-Fetching
Usability from the front office developer's point of view	Single endpoint	Multiple endpoints
Number of requests from the front	Low	High
File uploading	No	Yes
Use case	Single endpoint => Multiple microservices / API Gateway	Resource driven application

Coding languages

- JavaScript: server side and Web browser
- Go
- PHP
- Java / Kotlin
- C# / .Net
- Python
- Flutter
- ...

GraphQL in practice

<https://github.com/charroux/servicemesh>

Standard API specifications

Defining a Rest API

- OpenAPI 2.0 spec = Swagger
- An API definition includes:
 - The URL, or entry point of the backend service
 - The data format passed on a request
 - The data format returned by the service
 - The authentication mechanism

Example of an OpenAPI document

```
swagger: "2.0"
info:
  title: API_ID / optional-string /
  description: "Get the name of an airport from its three-letter IATA code."
  version: "1.0.0"
host: DNS_NAME_OF_DEPLOYED_API /
schemes:
  - "https"
paths:
  "/airportName":
    get:
      description: "Get the airport name for a given IATA code."
      operationId: "airportName"
      parameters:
        -
          name: iataCode
          in: query
          required: true
          type: string
      responses:
        200:
          description: "Success."
          schema:
            type: string
        400:
          description: "The IATA code is invalid or missing."
```

Service configuration

- Which APIs are served?
- How callers are authenticated?
- How the APIs are accessed by using HTTP REST?

```
type: google.api.Service
config_version: 3
name: calendar.googleapis.com
title: Google Calendar API
apis:
- name: google.calendar.v3.Calendar
authentication:
  providers:
    - id: google_calendar_auth
      jwks_uri: https://www.googleapis.com/oauth2/v1/certs
      issuer: https://securetoken.google.com
  rules:
    - selector: "*"
      requirements:
        provider_id: google_calendar_auth
backend:
  rules:
    - selector: "*"
      address: grpcs://my-service-98sdf8sd0-uc.a.run.app
```

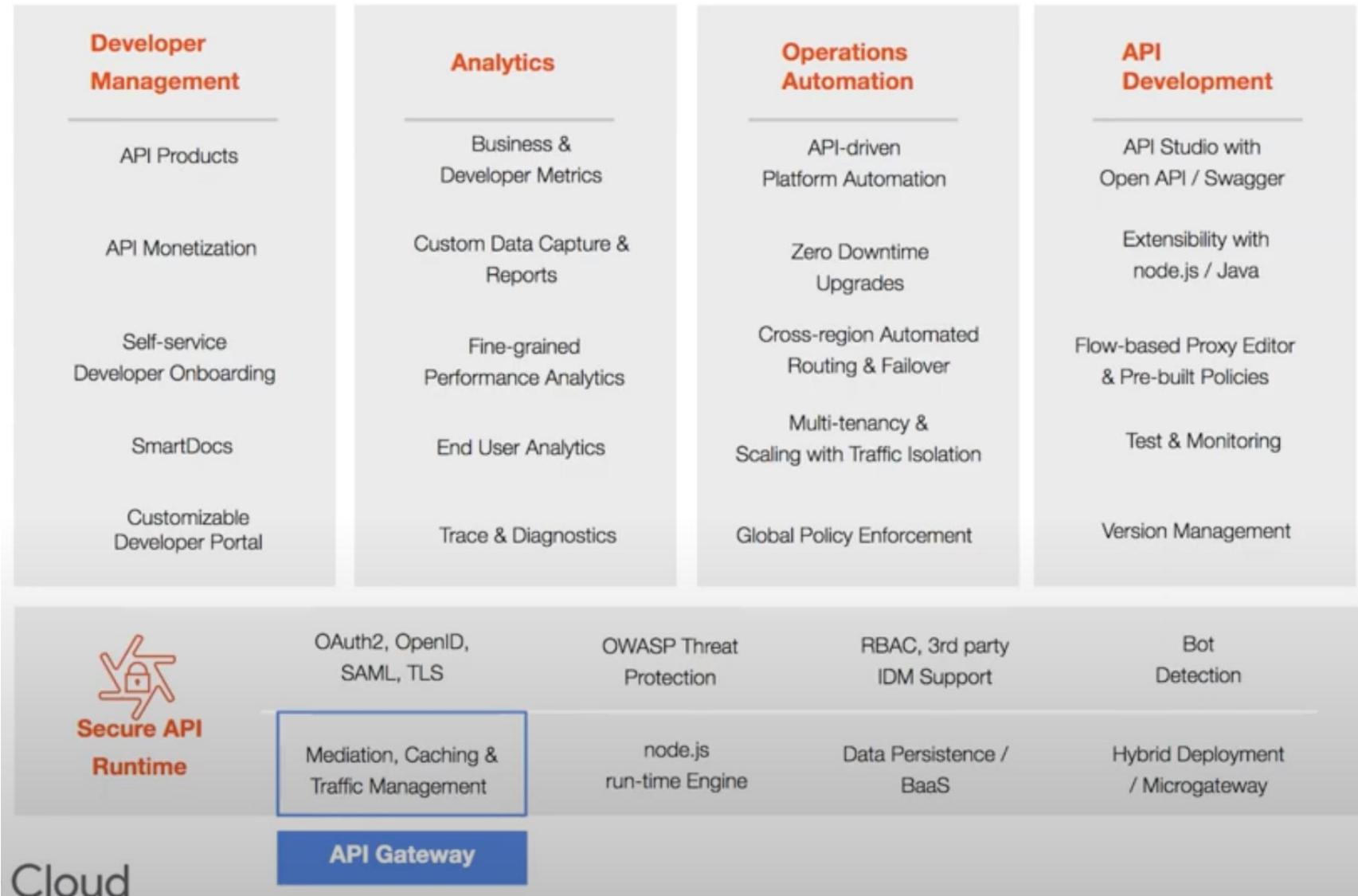
AsyncAPI Specification

- AsyncAPI is an open source initiative to promote the Event-Driven Architectures (EDA).
- The goal is to make it as easy as working with Swagger.



API Management

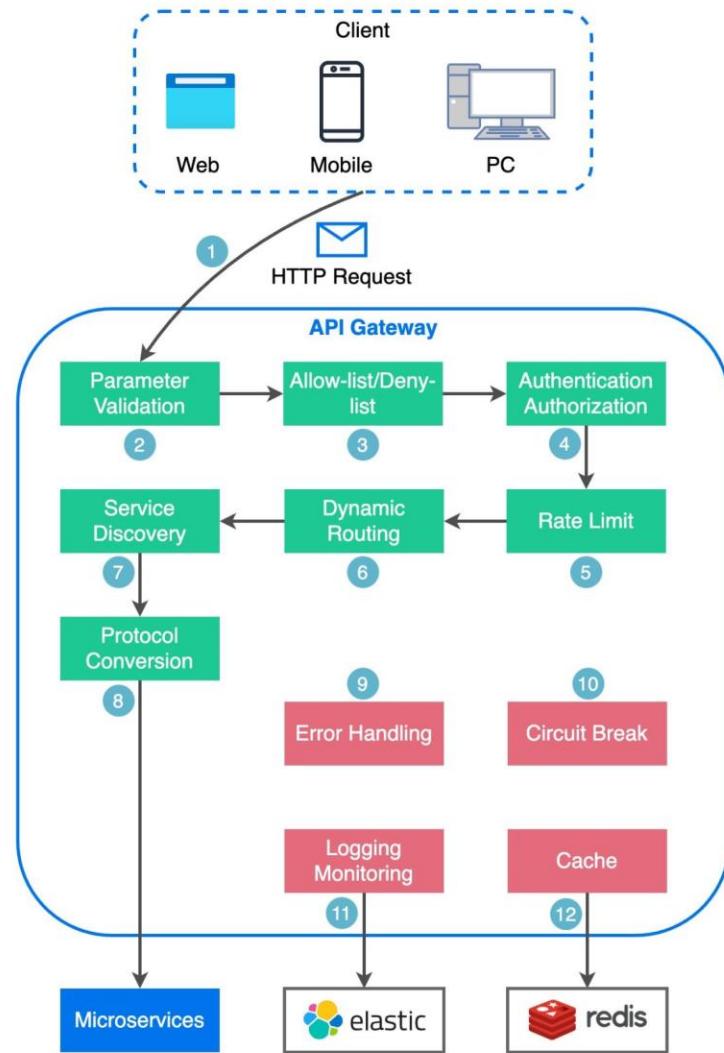
API Management / API Gateway



API Management / API Gateway

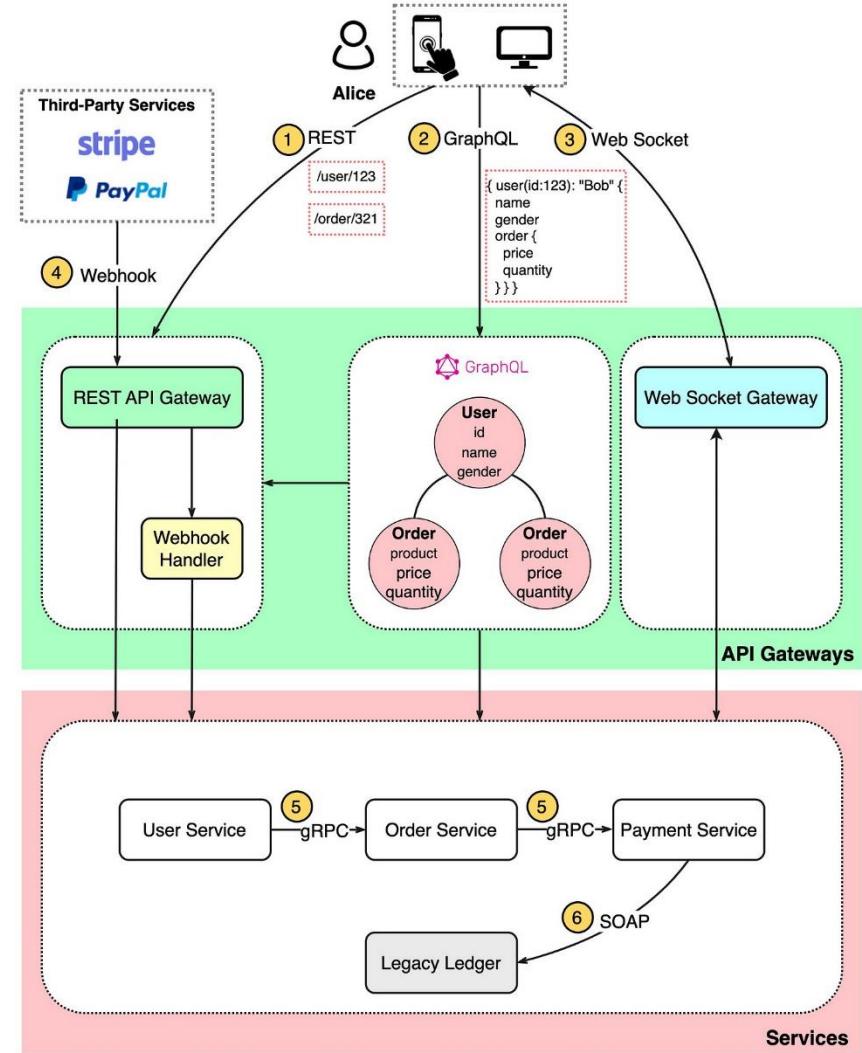
What does API Gateway do?

 blog.bytebytego.com



API Architectural Styles

 blog.bytebytego.com



API Management / API Gateway

