Microservices architecture consists of collections of light-weight, loosely-coupled services.

Implements single business capabilities --> cohesive enough to deveop, test, release, deploy, scale, integrate and maintain independently

Benefits of micro service --> the scale cube

X-axis --> scale by cloning or horizontal duplication

Y-axiz --> functional decomposition

Z-axis --> data partitioning

Advantages:

1. Independant scaling

2. Independant releases and deployment

3. Independent development

4. Graceful degradation

5. Decentralized governance

Significant investments are required like

Service replication

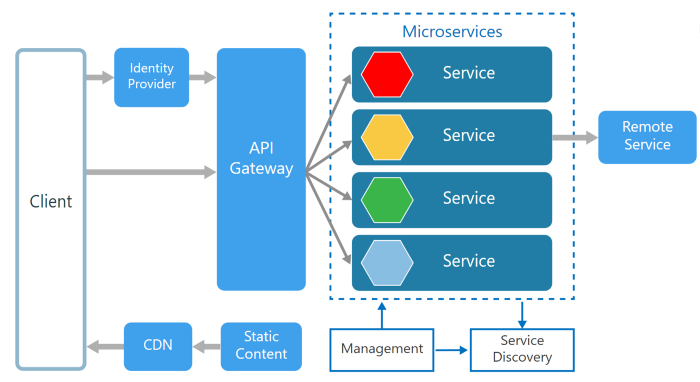
service registration and discovery

Service monitoring and logging

Resiliency

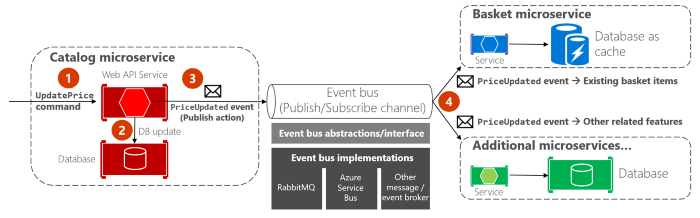
DevOps

API gateway



API Gateway acts as a **single entry point for all clients** as well as an **edge service for exposing microservices to the outside world as managed APIs**

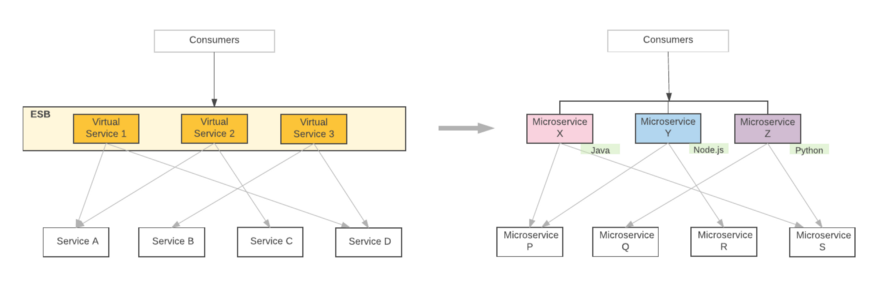
**simple load-balancing, authentication & authorization, failure handling, auditing, protocol translations,** and **routing**.



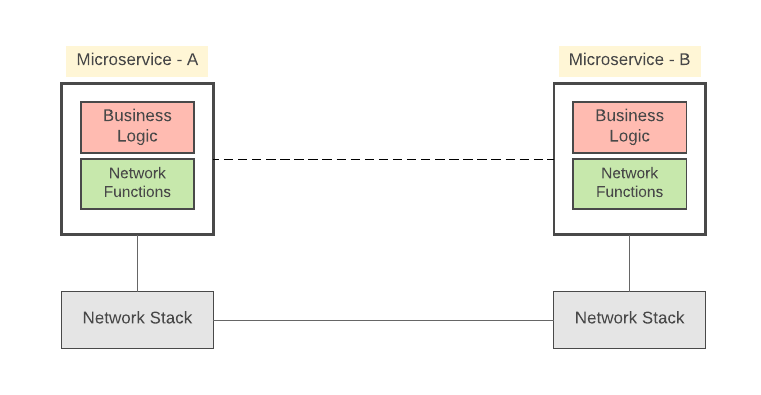
Service Mesh for Microservices

For synchronous request/reply communication between microservices

Application when dealing with composite microservices, with ESB architecture, the resilience is ensured by the functionalities such as circuit breakers, timeouts and service discovery etc..

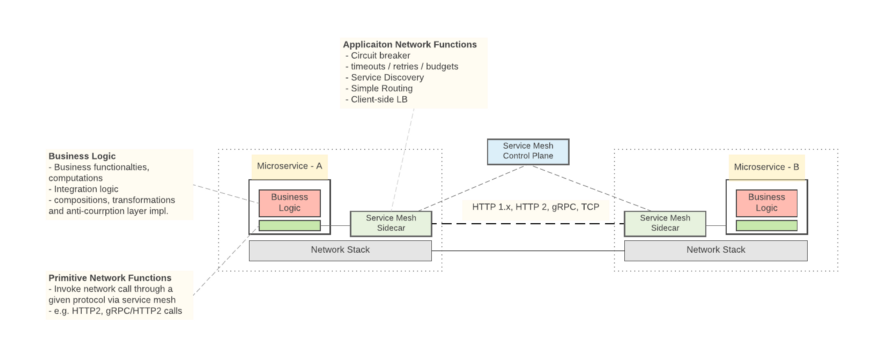


you have to implement all these functionalities at the microservices level.

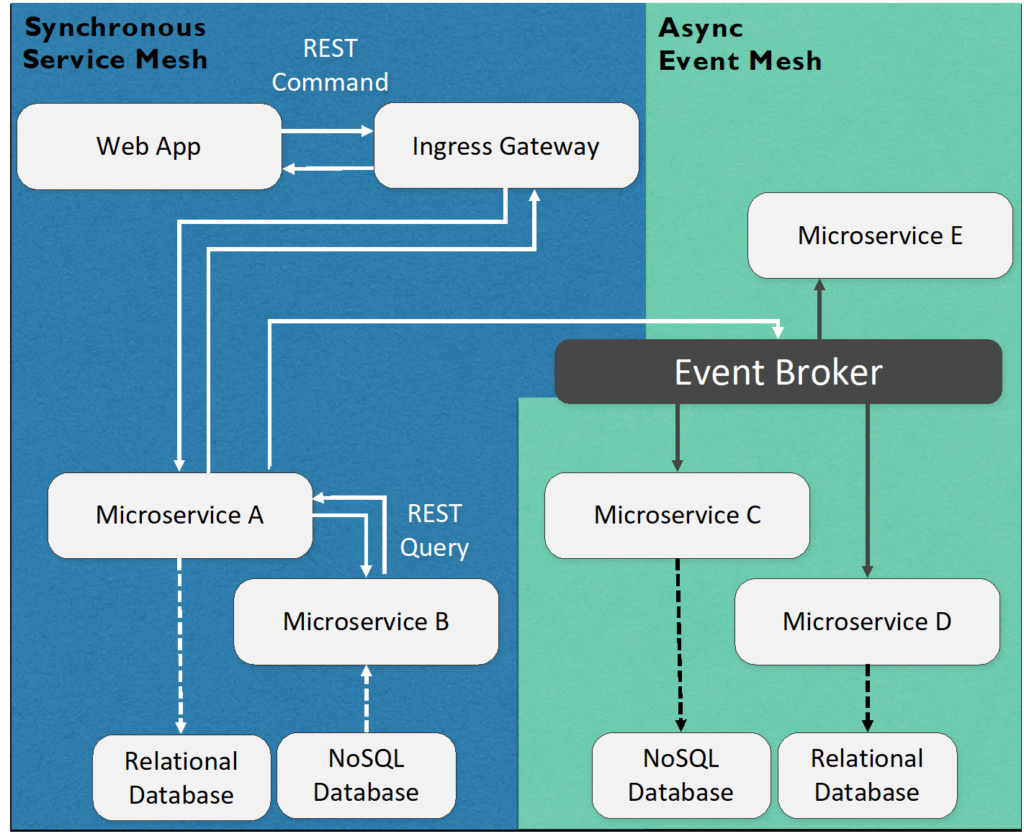


**Network Functions** that take care of the inter-service communication mechanisms (basic service invocation through a given protocol, apply resiliency and stability patterns, service discovery etc.) These network functions are built on top of the underlying OS level network stack.

A Service Mesh is an inter-service communication infrastructure, provides functions such as resiliency, service discovery etc

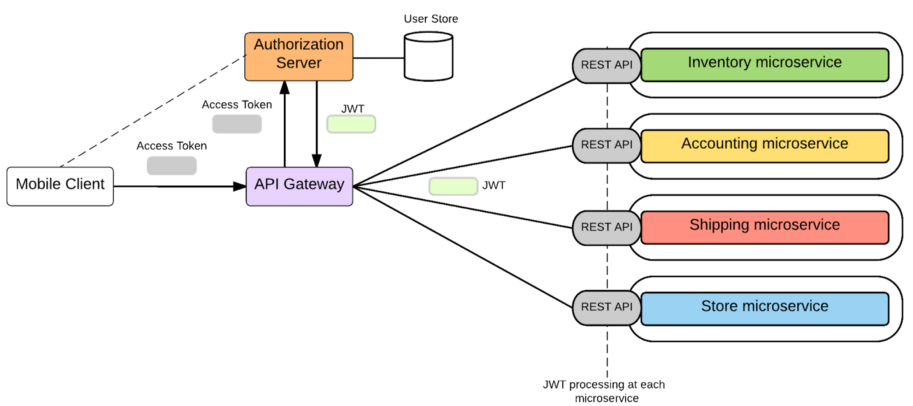


service mesh provides connection-level routing and traffic management for synchronous request/reply communications through sidecar injection into Kubernetes Pods

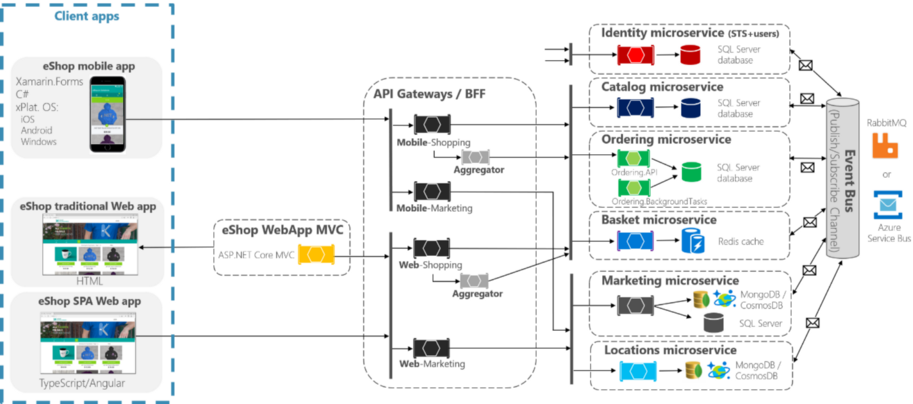


Backend for Frontend 🡪 different APIs for different frontends

Best practices

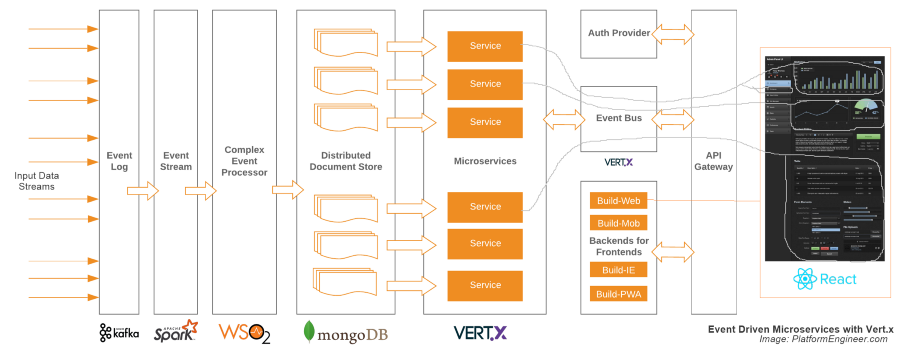
1. Doman Driven design 🡪 Model services around the business domain
2. Decentralised data management
3. Smart endpoints and dumb pipes – each service owns a well-defined API for external communication
4. Asynchronous communication
5. Avoid coupling between services
6. Decentralize development
7. Keep domain knowledge out of the gateway
8. Token-based authentication 🡪 authentication at API Gateway level
9. 
10. Event Driven nature
11. Eventual consistency
12. Fault tolerance 🡪 circuit breaking, bulkhead, retries, timeouts, fail fast, fail over caching, rate limiters, load shedders

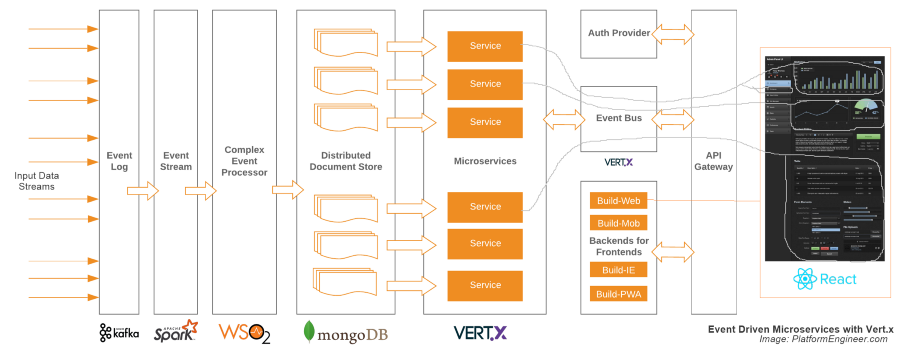
Applications suggestions



Microservices architecture for an online shopping application (Image: [microsoft.com](https://docs.microsoft.com/en-us/dotnet/standard/microservices-architecture/multi-container-microservice-net-applications/implement-api-gateways-with-ocelot)) — This architecture is proposed by Microsoft developers using Microsoft technologies. Here, the API Gateway has been tailored to treat web and mobile users differently. For data layer, data store technologies are carefully selected according to the business capabilities (relational databases for structured data, Redis for temporary data caching, MongoDB and CosmosDB for unstructured data). Interservice communication handled by the event bus. Keeping technologies aside, this is the **most common integration pattern** used in microservices-based applications.

For real-time large data processing applications





Microservices architecture for an application which displays realtime updates to end users using large amounts of input data streams coming from various event sources (e.g. traffic data, weather readings, stock market feeds, social media posts, sensor outputs). These input data streams are initially collected by an event log implemented using Kafka. It persists data on disk and thus can be used for batched consumption purposes (analytics, reporting, data science, backup, auditing) or sent for real time consumption purposes (operational analytics, CEP, admin dashboards, alert apps). However, according to this diagram, the continuous incoming stream is divided into micro-batches with specified intervals using Spark and fed into the WSO2 Siddhi CEP engine. Then it identifies the events and persists them in unstructured forms using MongoDB data stores. Microservices consume these data and display to the end users. If you carefully look at the design, since Vert.x event bus has the ability to create connections with frontend UI components, that feature has been used to efficiently update only the relevant parts in the UI. Keeping technologies aside, this is a great architecture for event-driven non-blocking microservices-based application.

NetFlix designing:

Netflix used AWS cloud

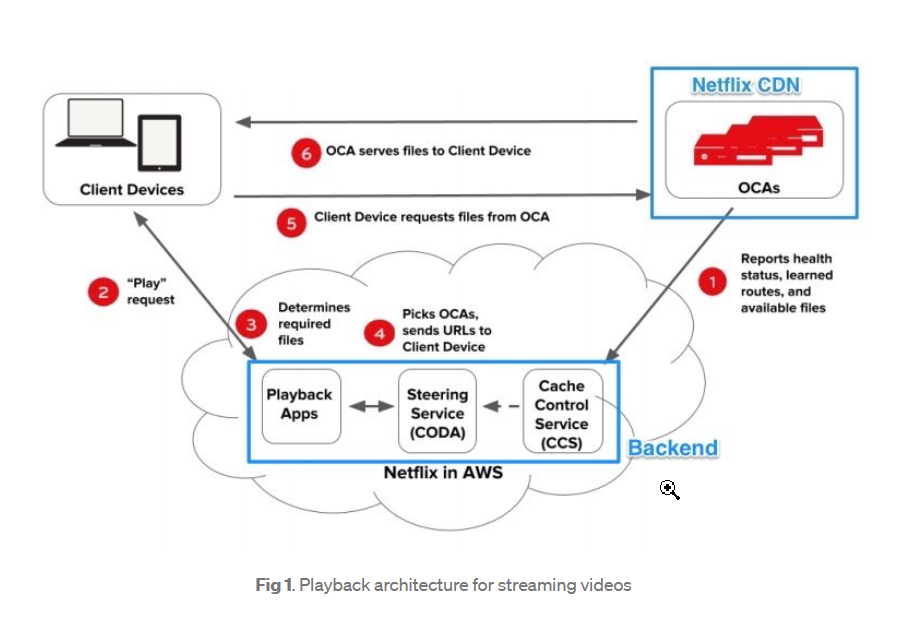
Did not do the undifferentiated heavy lifting work of building data centers

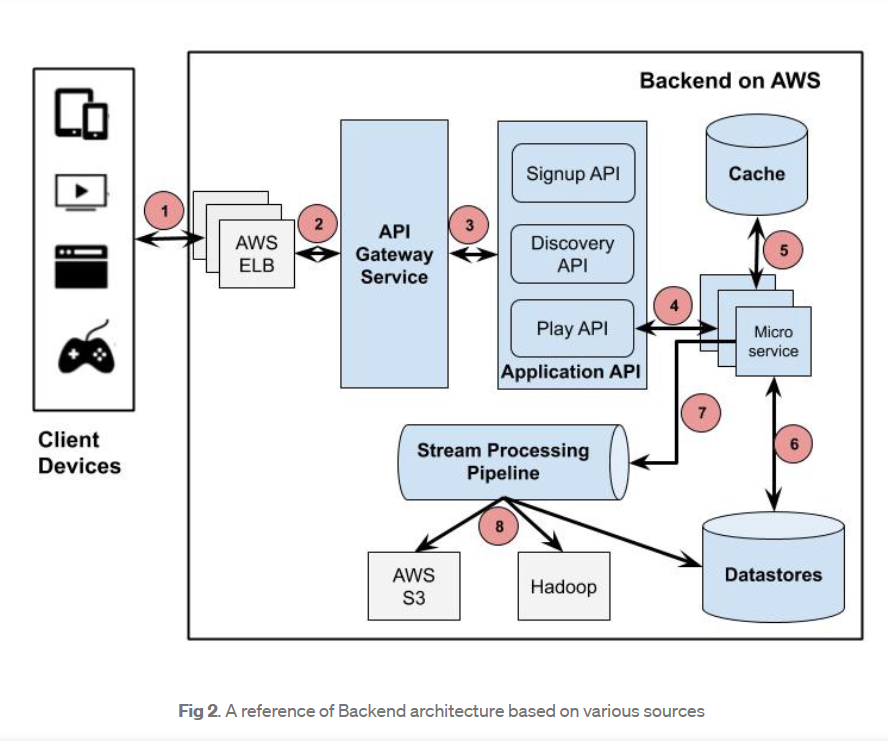
Separation of concerns and data encapsulation

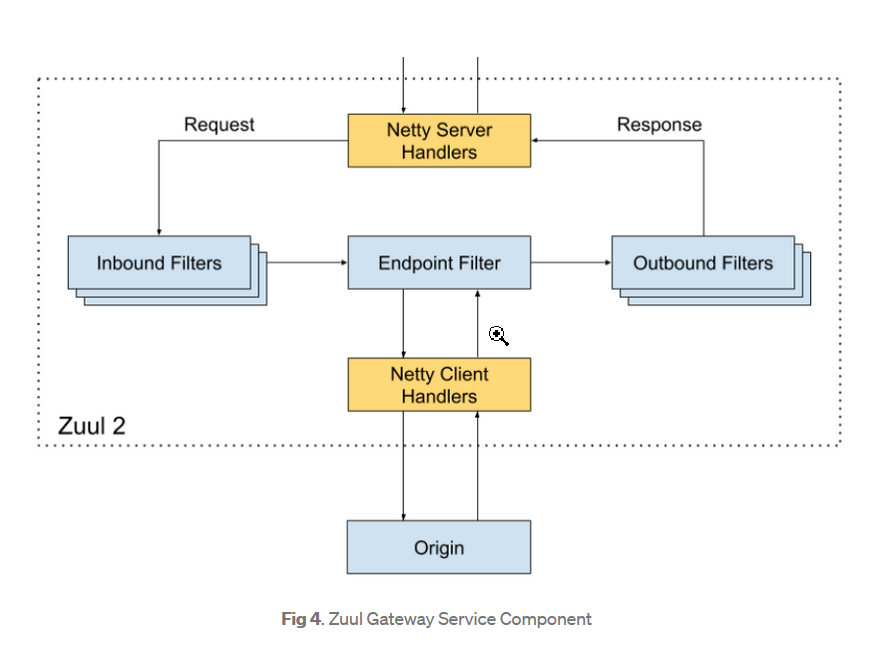
AWS handles:

* Scalable computing instances
* Scalable storage
* Business logic microservices
* Scalable distributed databases
* Big data processing and analytics jobs
* Video processing and transcoding

Open Connect 🡪 CDN







 Inbound Filters can be used for authentication, routing and decorating the request.

 Endpoint Filter can be used to return static resources or route the request to appropriate Origin or Application API for further processing.

 Outbound Filters can be used for tracking metrics, decorating the response to the user or adding custom headers.

Event Sourcing:

* An event source pattern is an approach to a sequence of events.
* Event sourcing stores the state of a database object as a sequence of events
* The system of record is a sequential log of all events that have occurred during a system’s lifetime.

Principles of an event sourcing pattern:

* The events are immutable.
* There can be as many events for the given entity as needed – that is, there is no limit on the number of events per object.
* Every event name represents the event’s meaning – for example, *NewUserCreationEvent*.
* To use the entity in the application (for example, to display the name of a user in the UI), you create a flat representation of the entity.  Each subsequent use of the entity recalculates its current state via the sequence of state-changing events.

Benefits of Event Sourcing

* Restore complete data and also the history of the data
* Greater fault tolerance 🡪 as there will be more than one event for a data
* Facilitate R and D process
* Improve operation flexibility 🡪 fixes a miscalculation with the previous event from memory
* Ensure compliance via full traceability
* Support an open architecture
* Reduce processing time 🡪 as multiple event can be processed parallel

CQRS – Command Query Responsibility Segregation – promotes separation of commands and queries – in practice, the read and write functions.

* One is intended for the update and delete – the writing model.
* With the other segment you do the read – the read model
* There are two databases, rather than the one used in the traditional CRUD approach, and each side is optimized for quick and effective operation, whether reading or writing.
* How does CQRS work together with event sourcing? The part of the application that updates data then appends the event sequence.
* For example, in an Apache Kafka topic, the “write” segment of the application adds new events to the queue. Another segment of the application (called an event handler) is subscribed to the Kafka topic; it reads the events, transforms data accordingly, and writes the final state of the data into the “read” database

