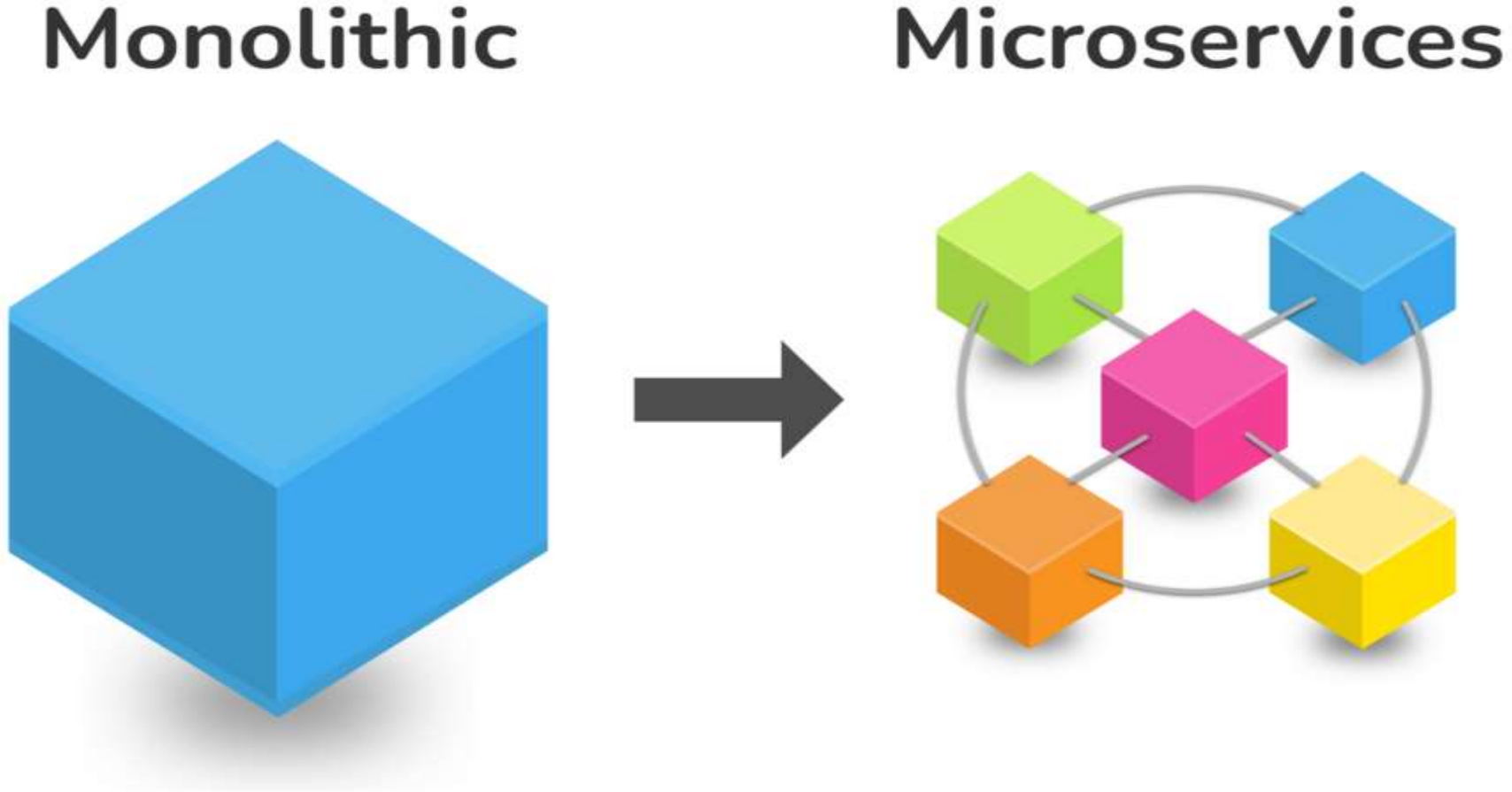
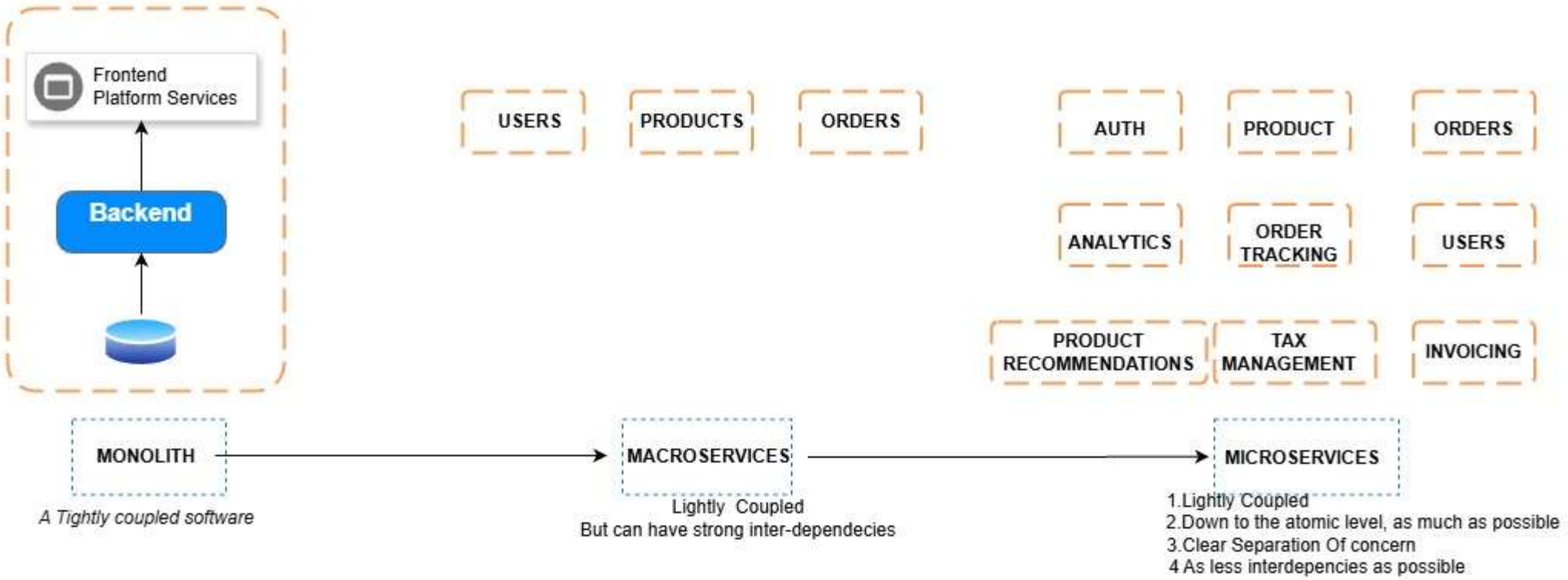


Prepared By -: Bhagwat Dhanore

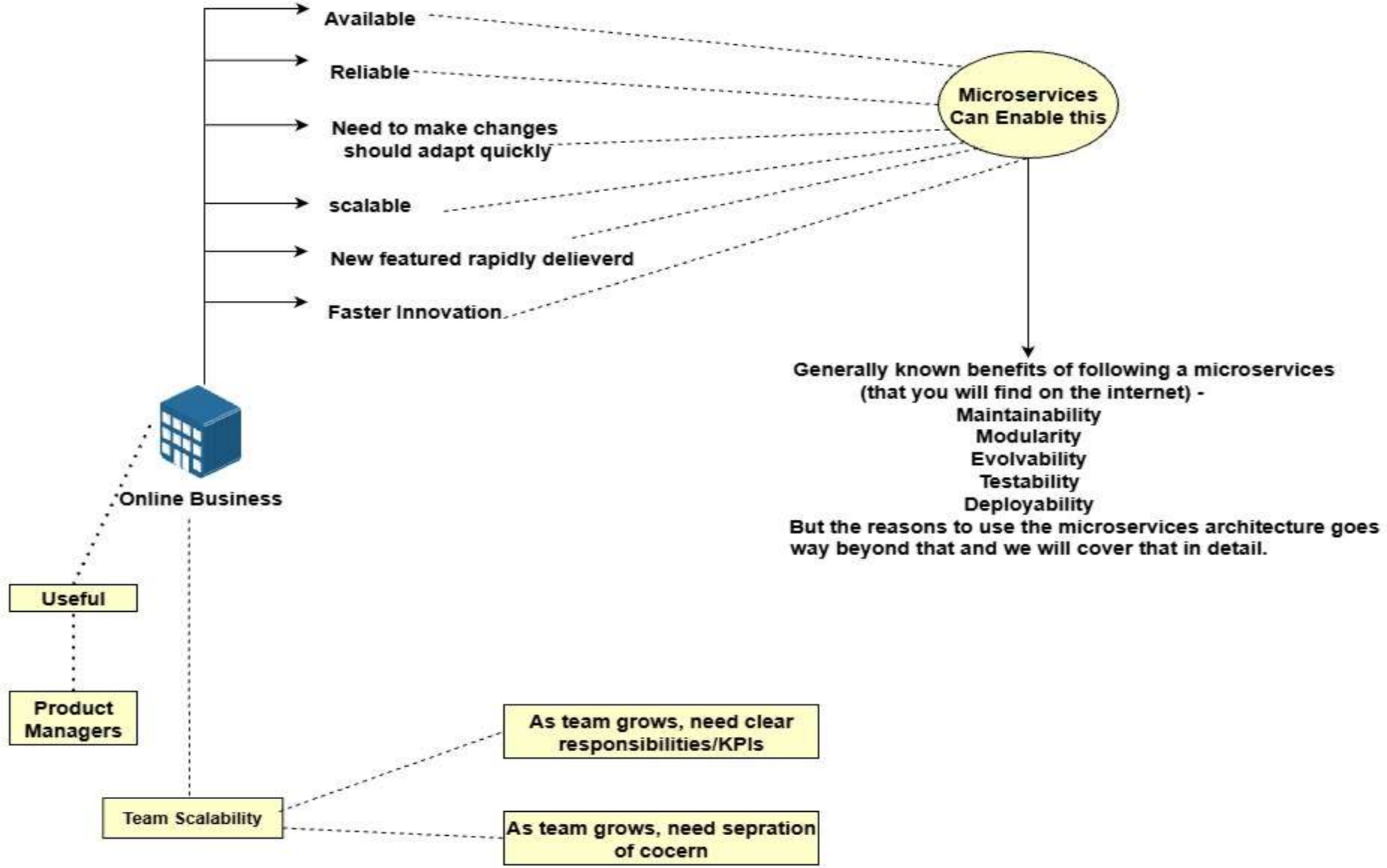
References-:

- Extracted from Akhil Sharma Youtube Channel https://www.youtube.com/@AkhilSharmaTech and Akhil Sharma Github repository github.com/AkhilSharma90
- https://www.geeksforgeeks.org/system-design/microservices/
- All the diagrams are prepared on draw.io https://app.diagrams.net/
- https://www.kandasoft.com/blog/addressing-key-scalability-challenges-in-microservices-architecture

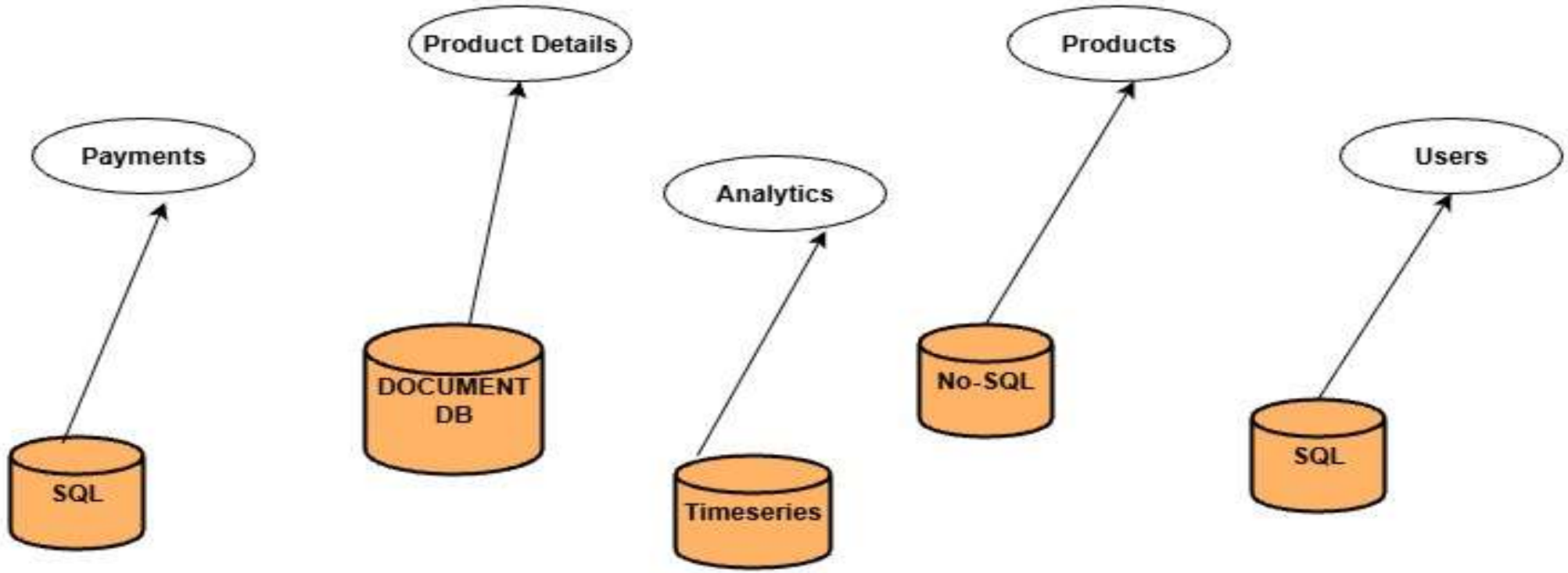
10,000 Ft Prespective



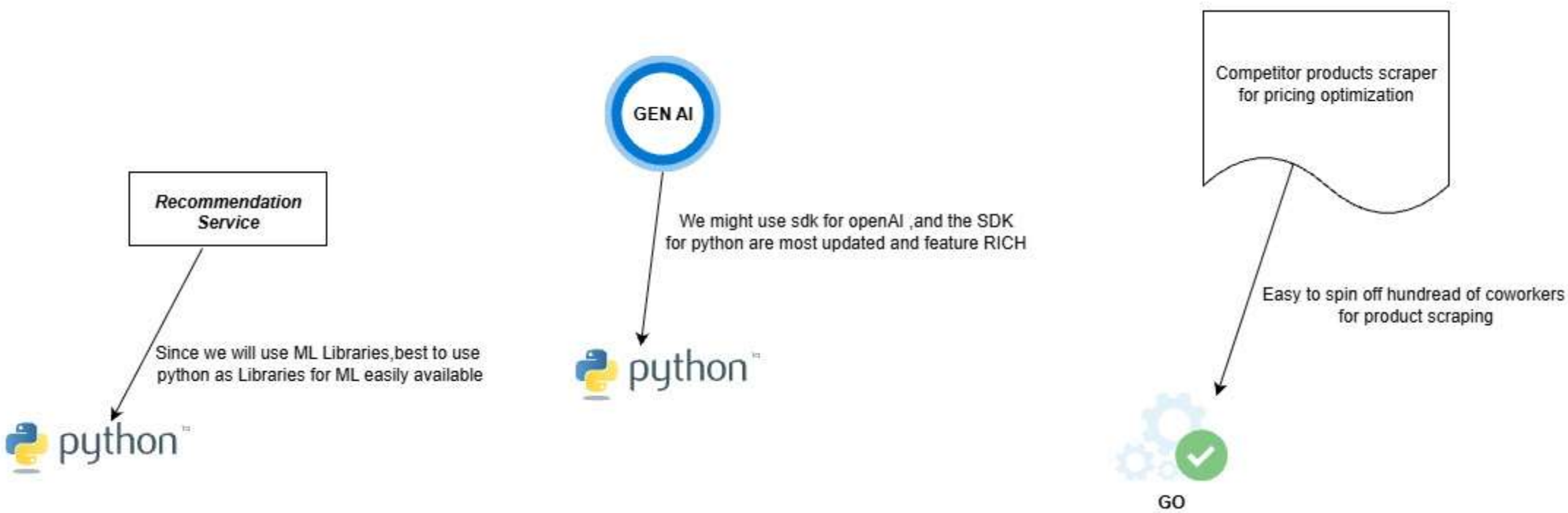
- How Online Business Looks Like



Lenses View from Tech Architect



Lenses View from Tech Architect



Multi Cloud Approach

Recommendation Service

GenAi

Public API

Vertex AI

GCP

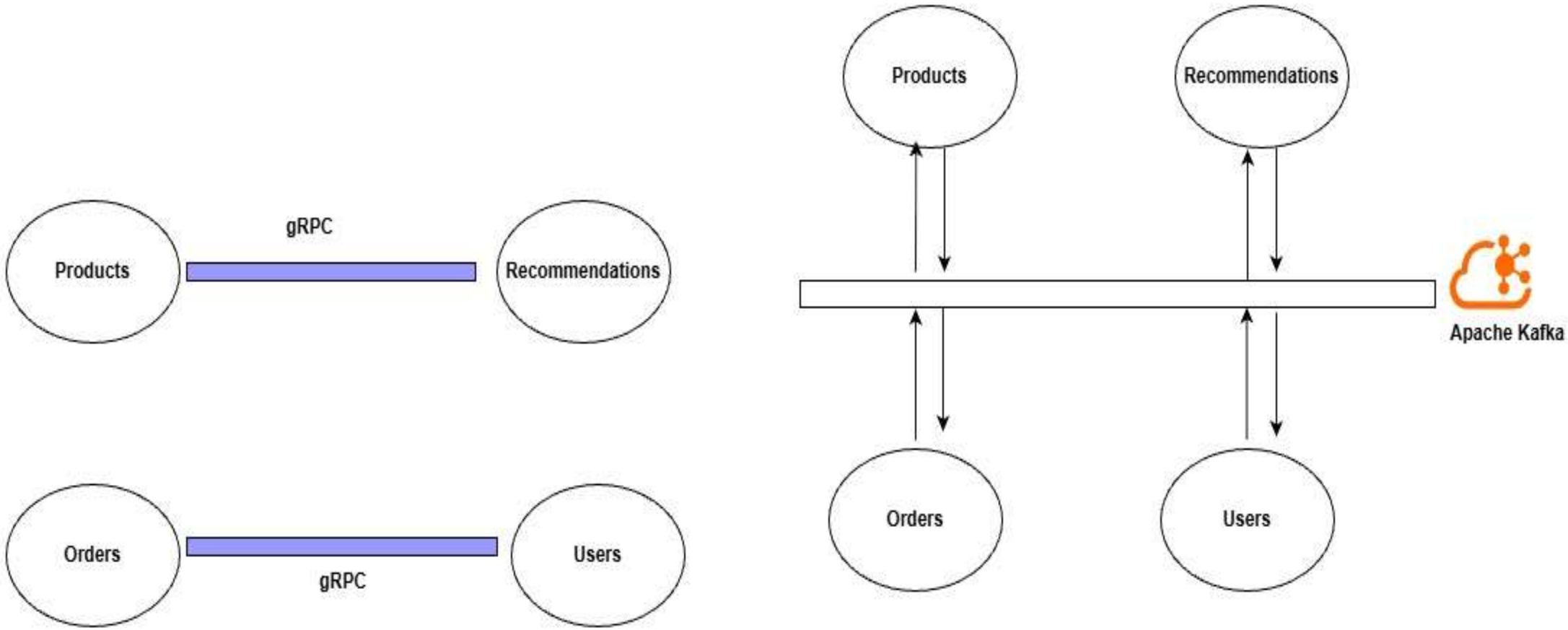
AZURE OpenAi

Products

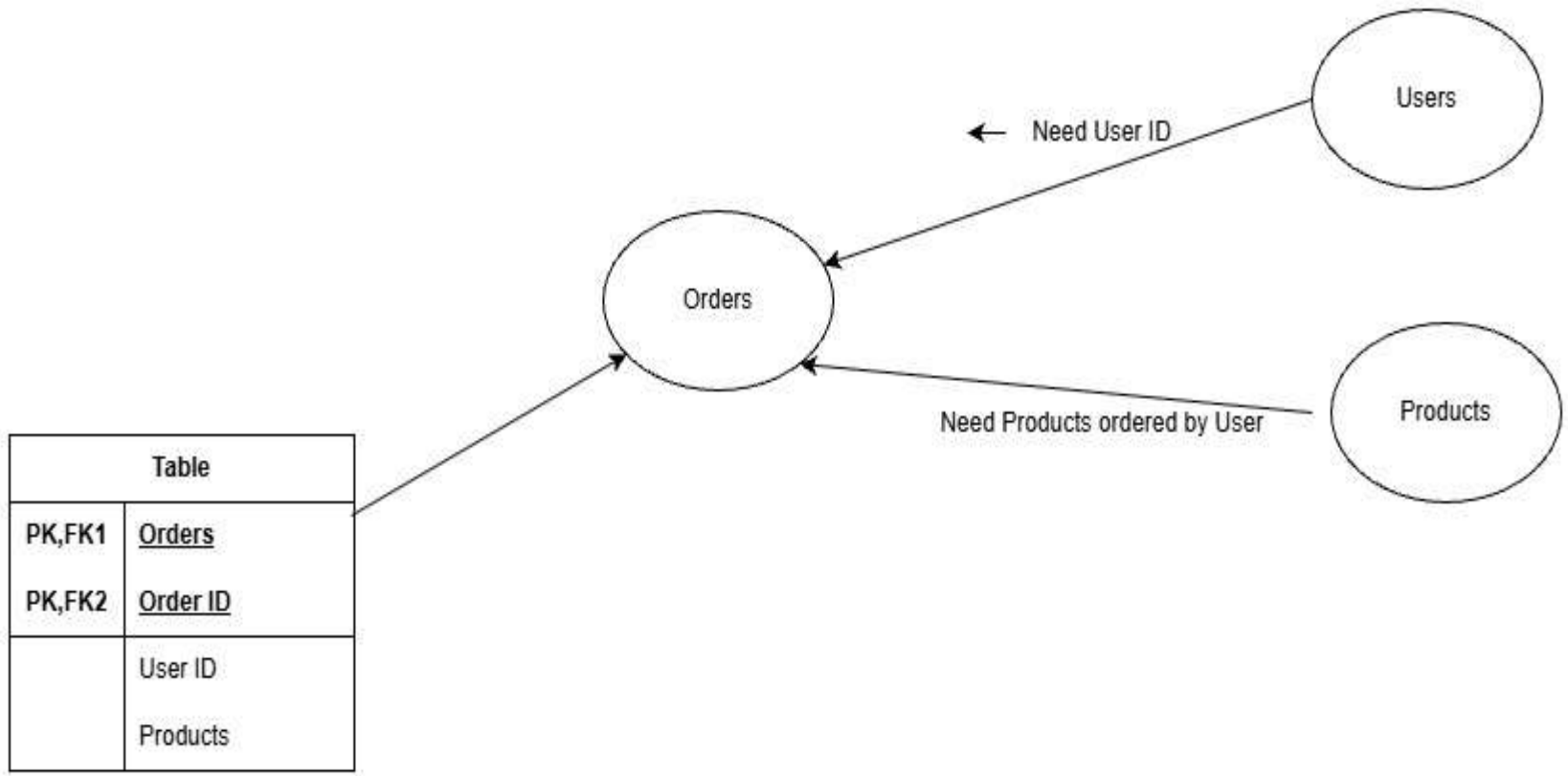
Orders

AWS API GATEWAY

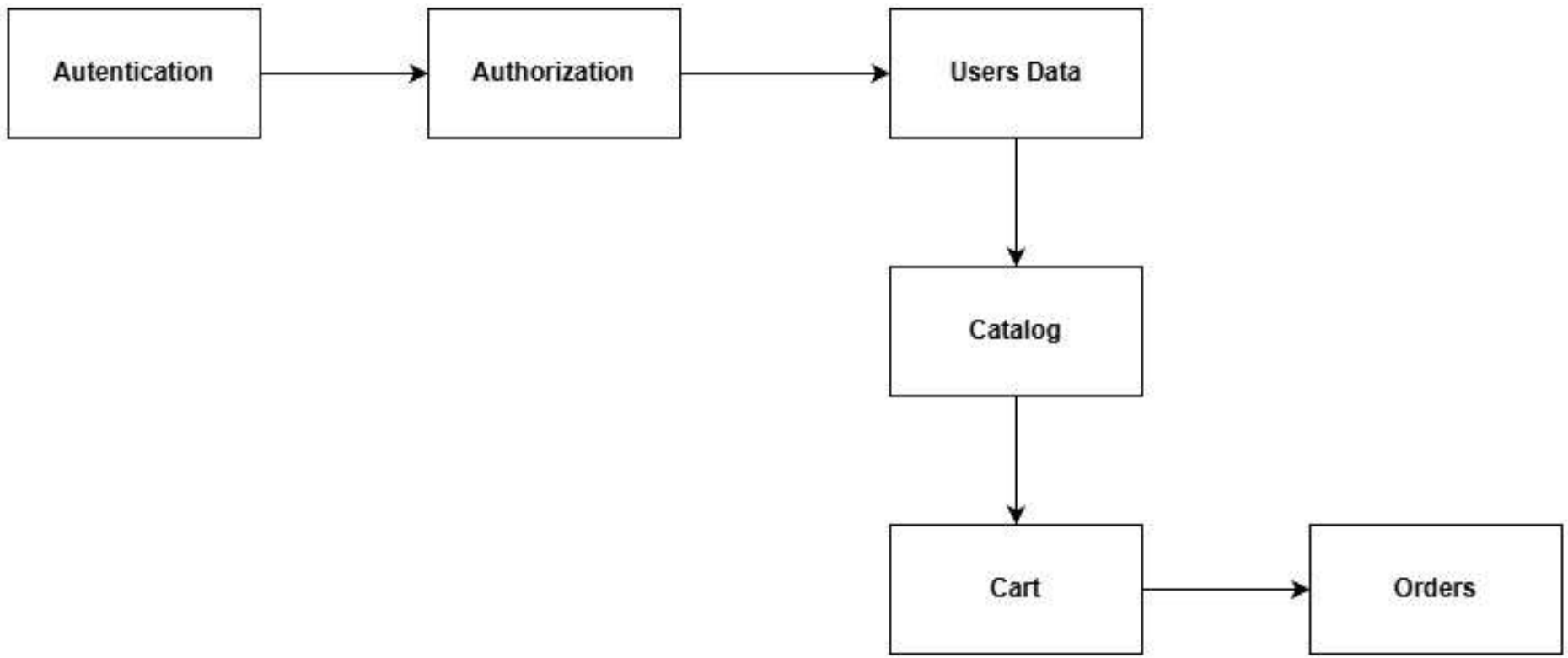
Communications



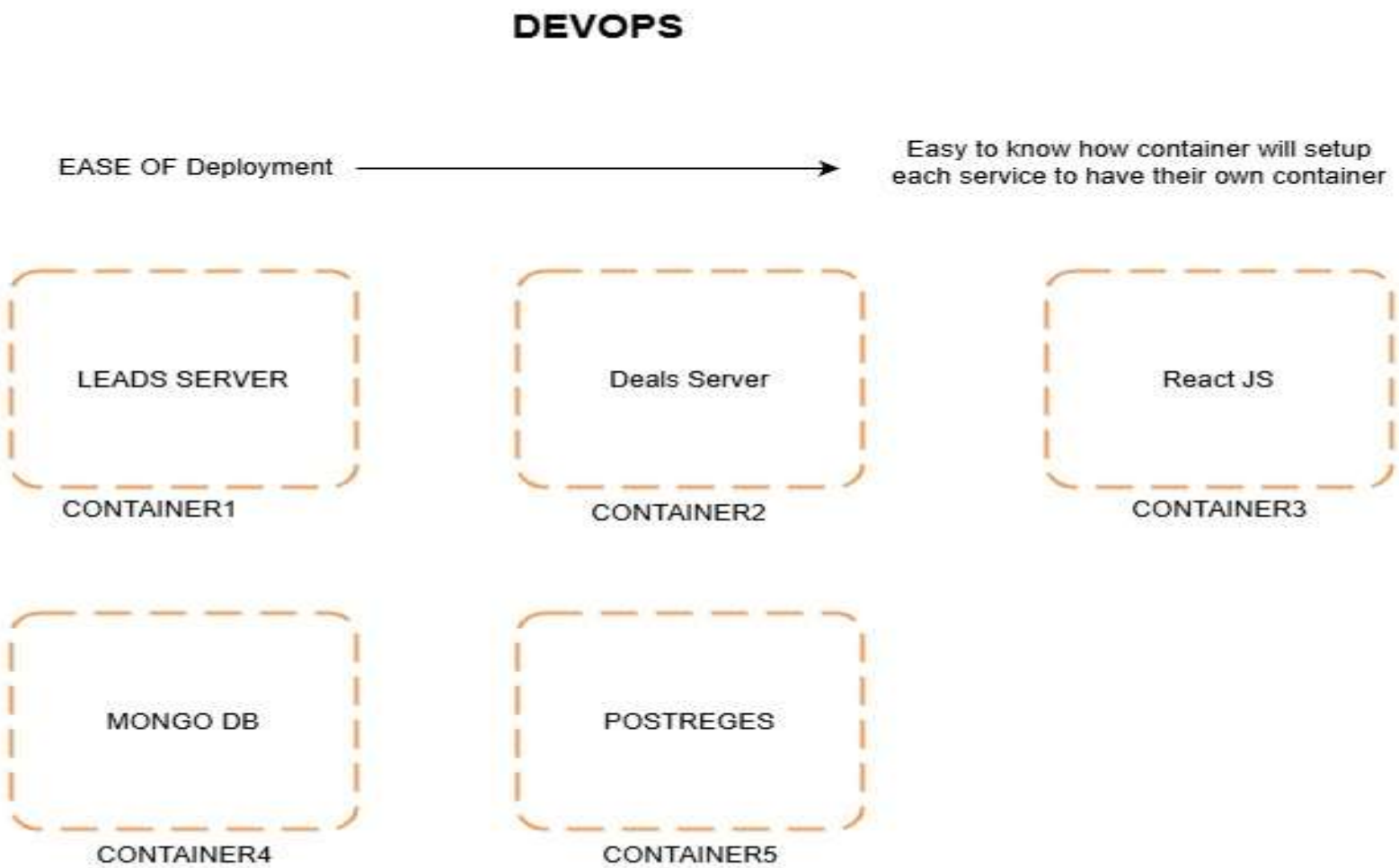
Data Sharing Pattern



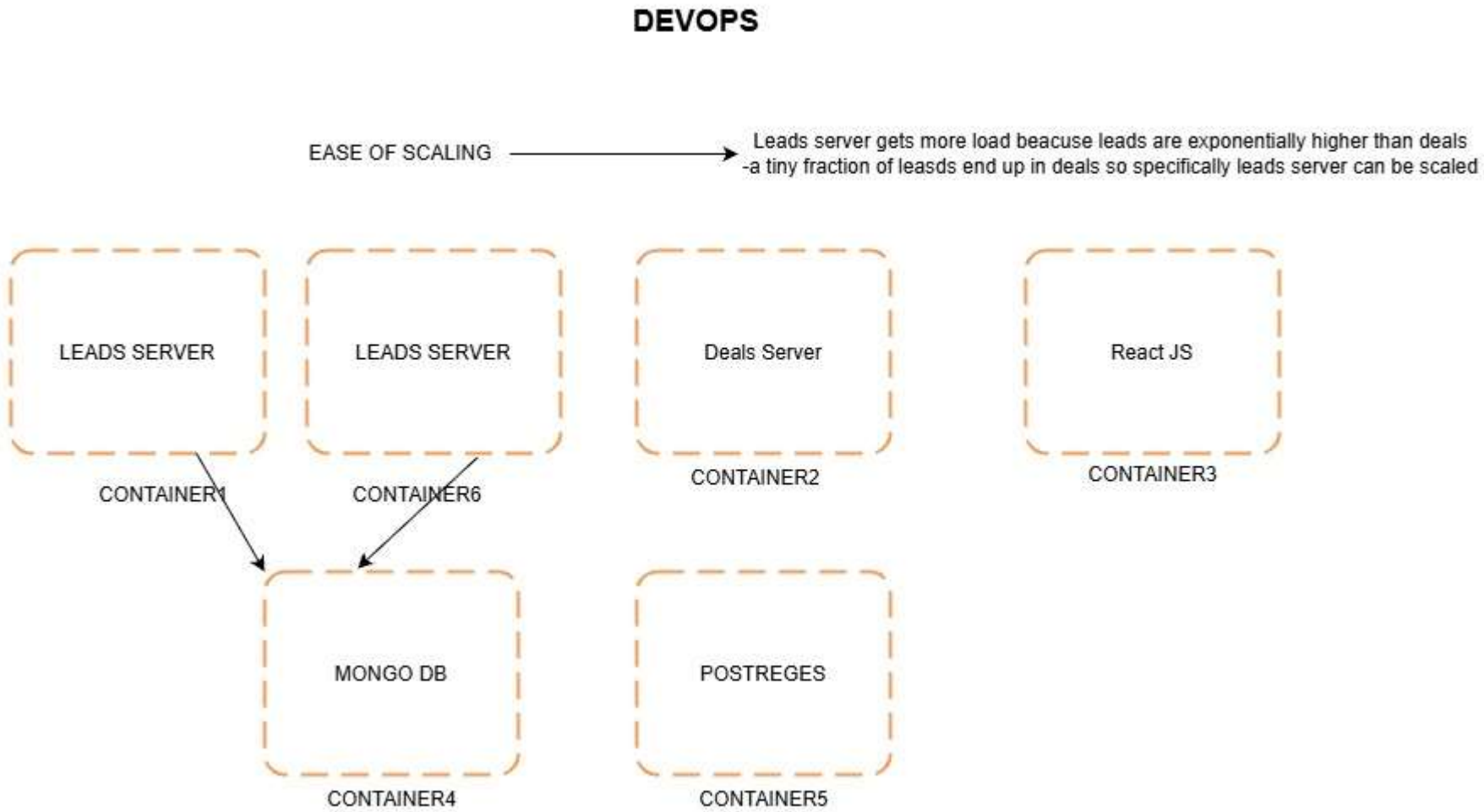
Order In Which The Information Flows



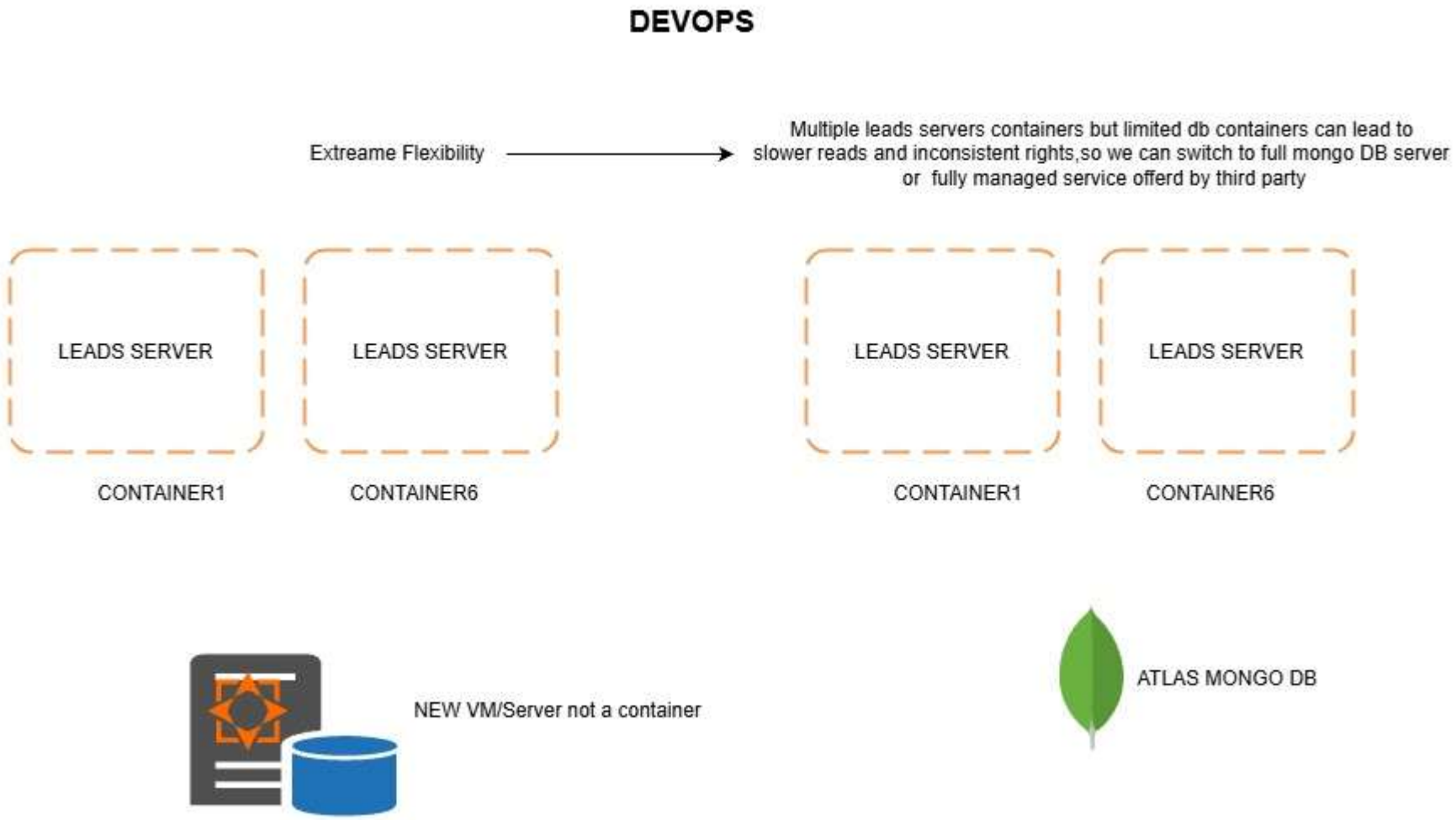
Lenses view from Devops prespective



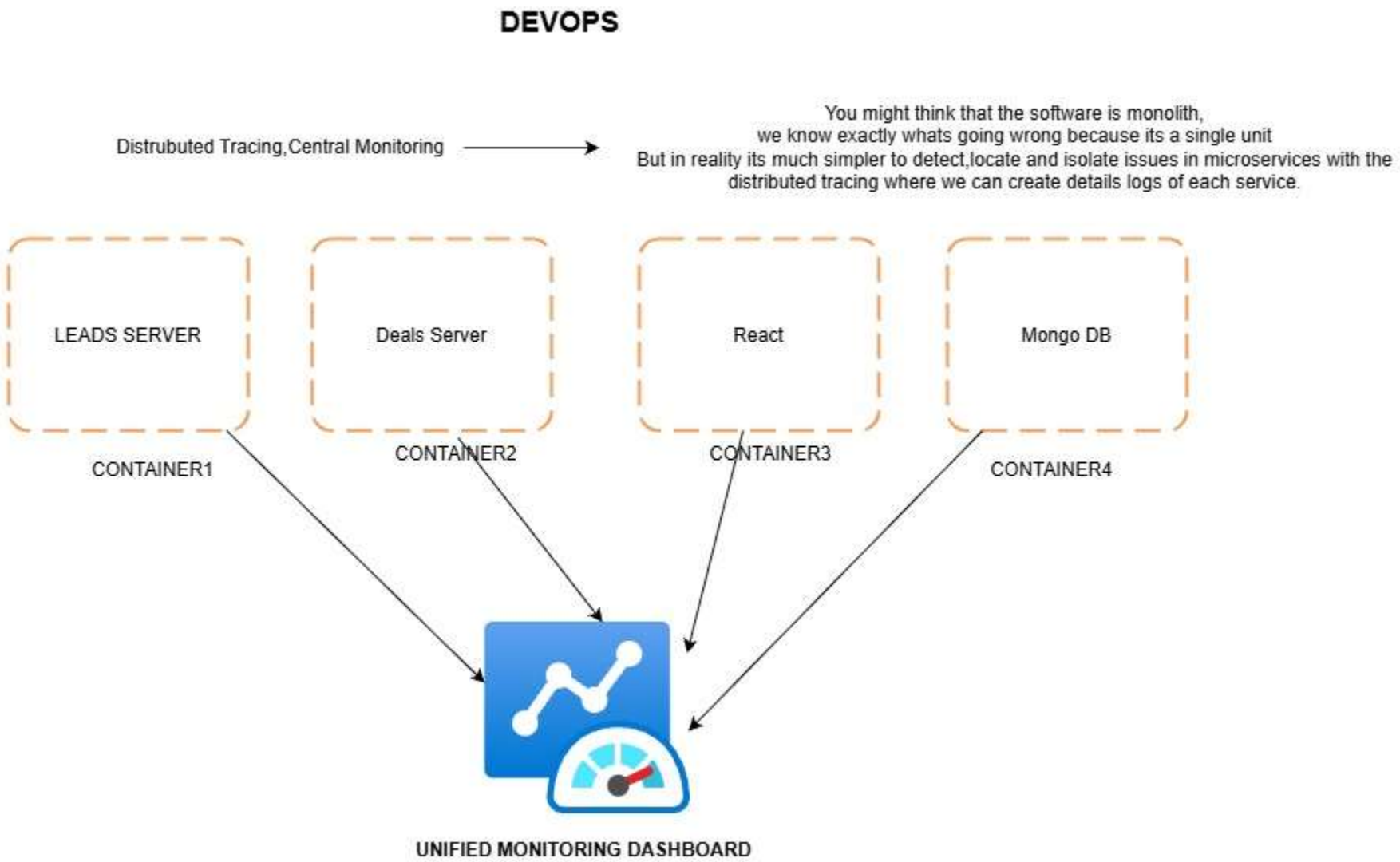
Lenses view from DeVops prespective



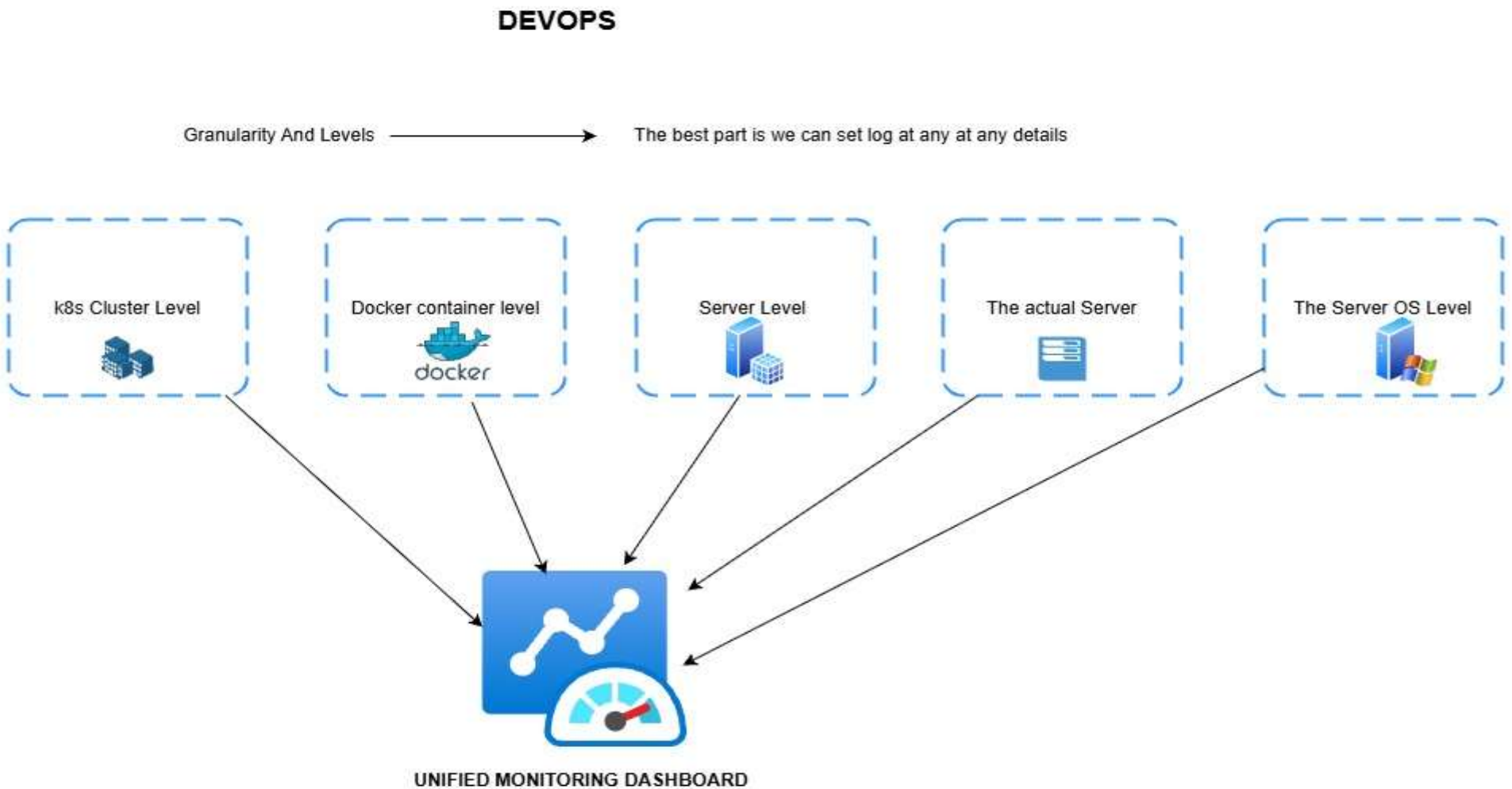
Lenses view from DevOps prespective



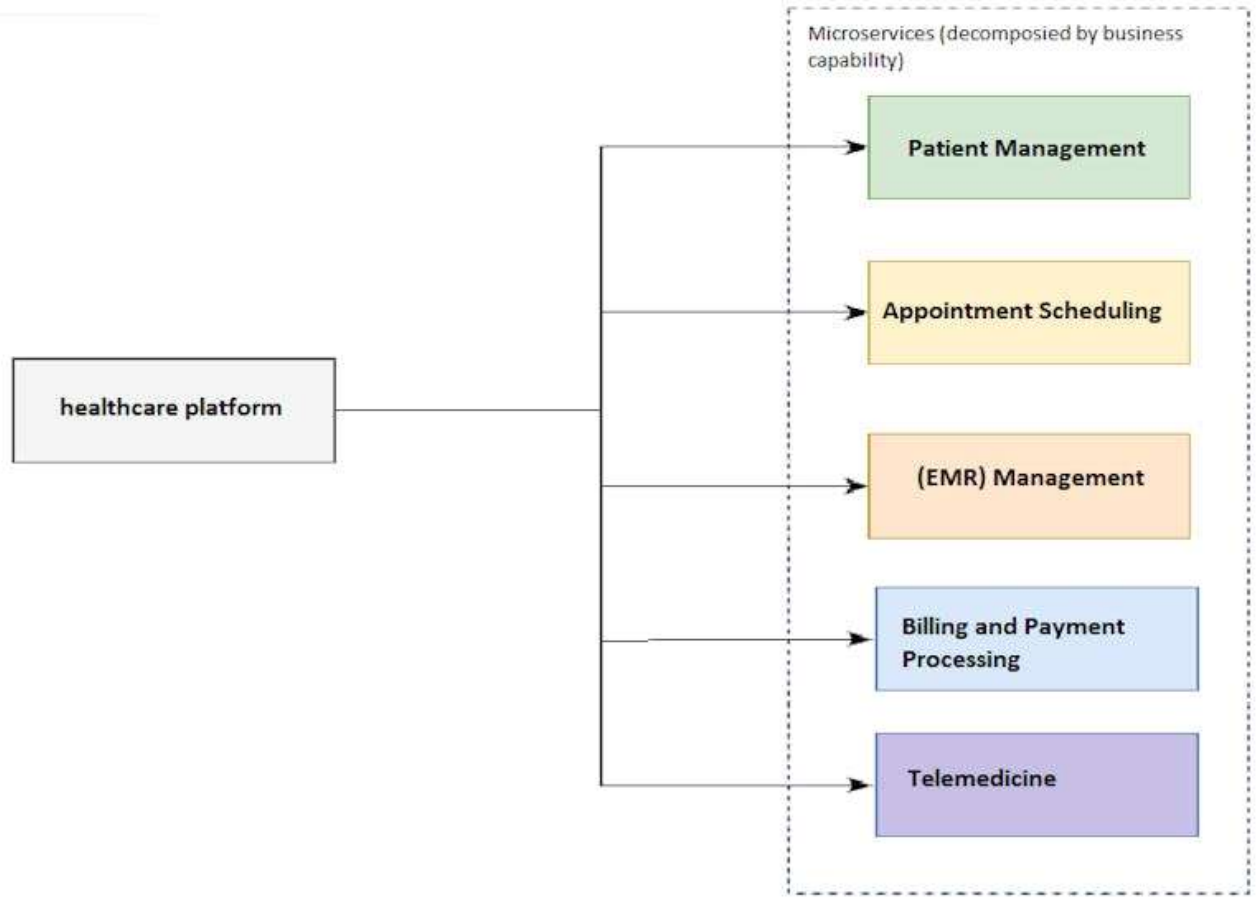
Lenses view from Devops prespective



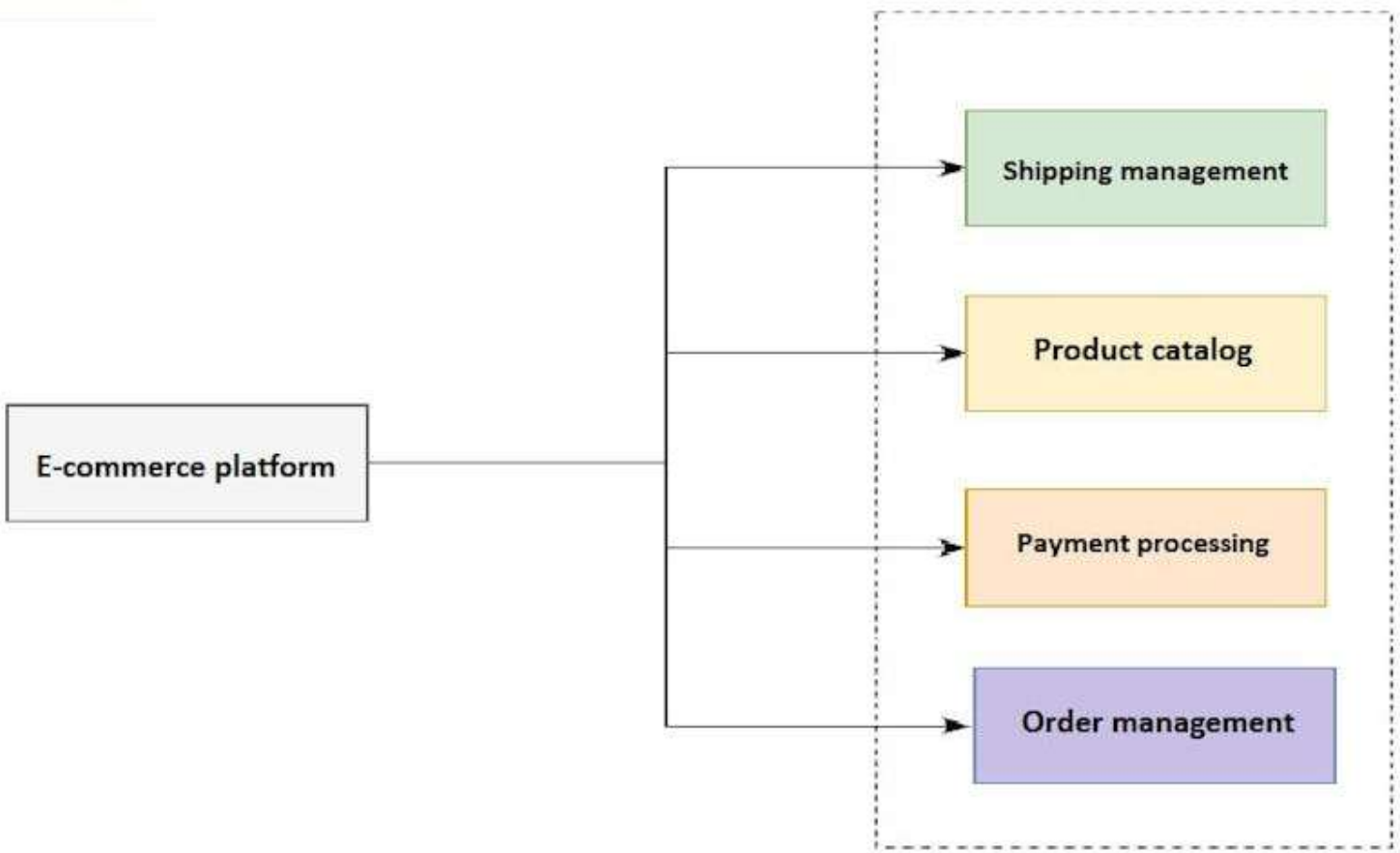
Lenses view from Devops prespective



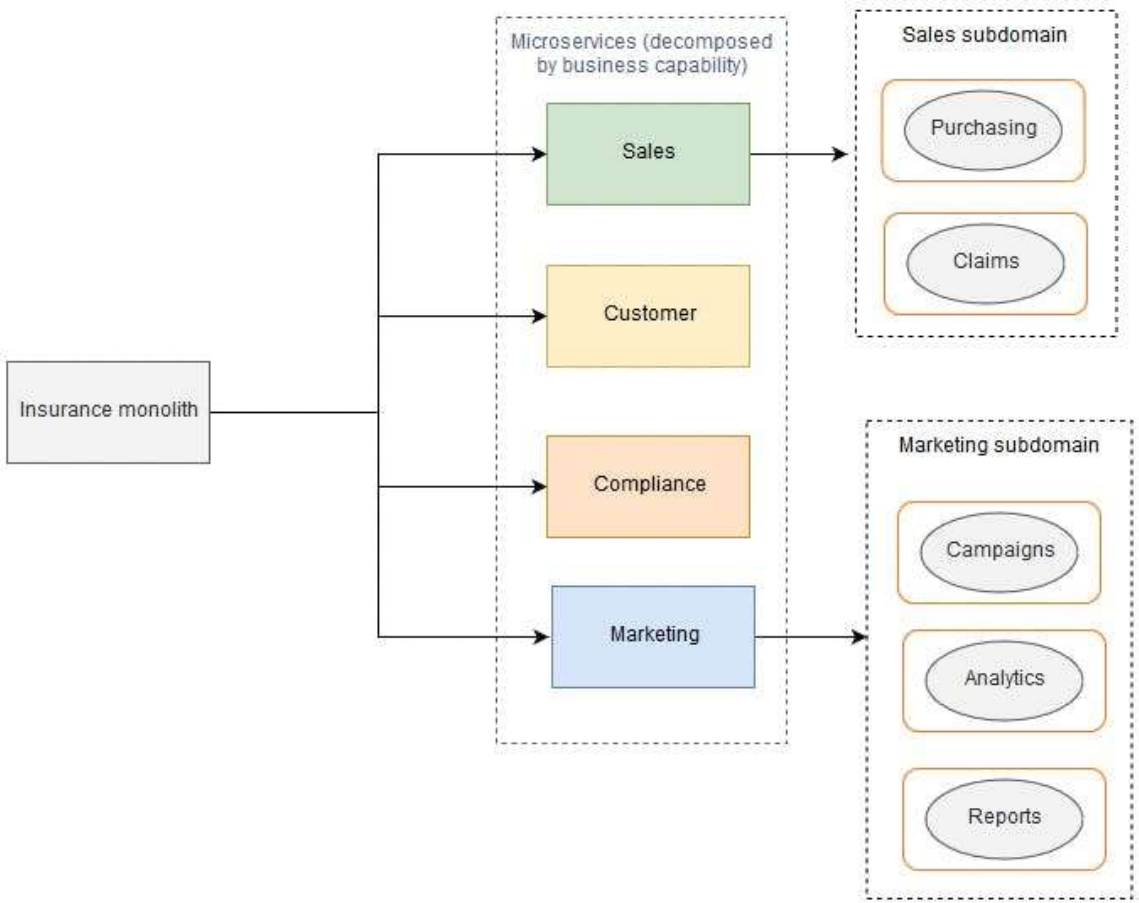
Decomposition by Buisness Capabilities



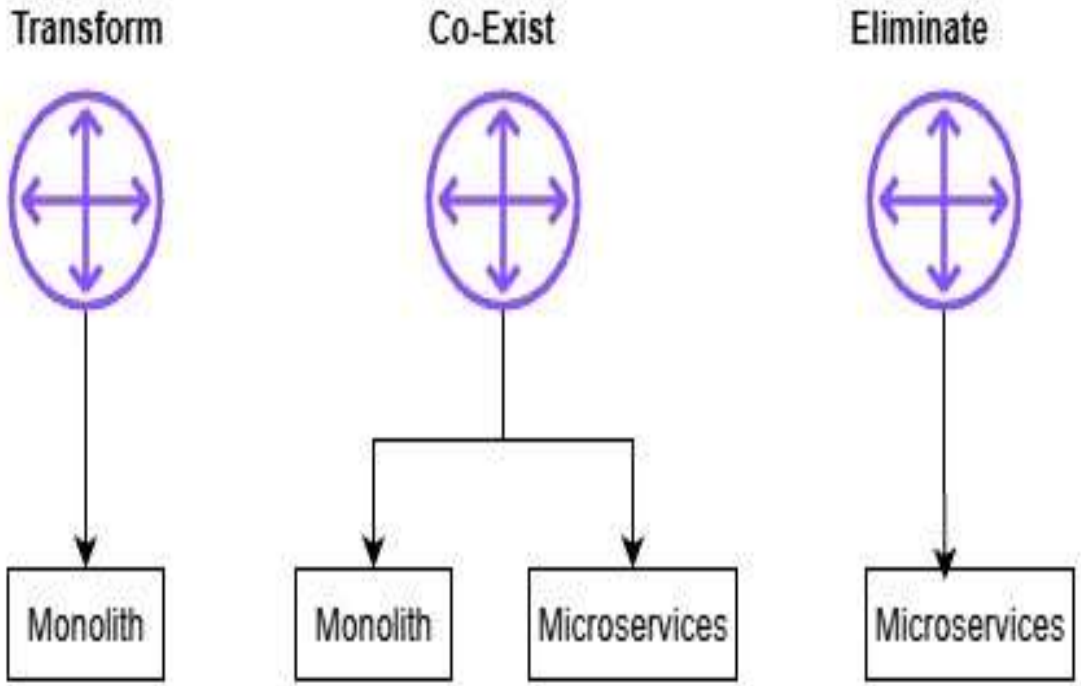
Decomposition by Transaction



Decomposition by Subdomains



Strangler Pattern



so during the decomposition phase you decided to introduce microservives
but how will they introduced into the cloud infra?
because it is risky to introduce sudden big change.

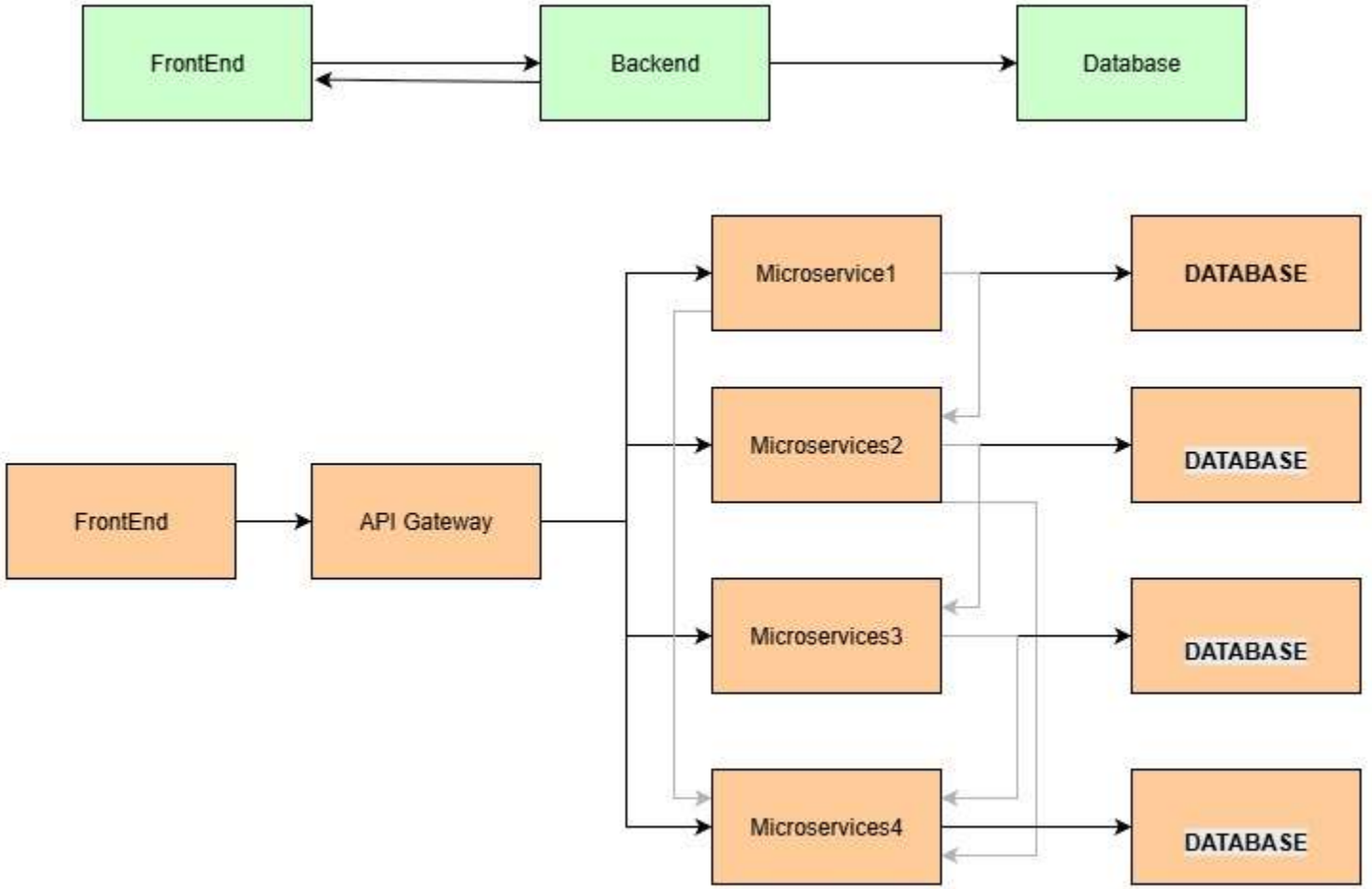
This is solved using the very famous strangler pattern where
monolith feature and microservices co exist for some time
and slowly monolith feature is removed,
this ensures no sudden jerks and shocks are introduced into the opeartions.

Ideal Microservices Characteristics

- Highly Testable
- Loosely Coupled
- Independently Deployable
- Organised around business capabilities-subdomain driven-DDD
- Owned by small team

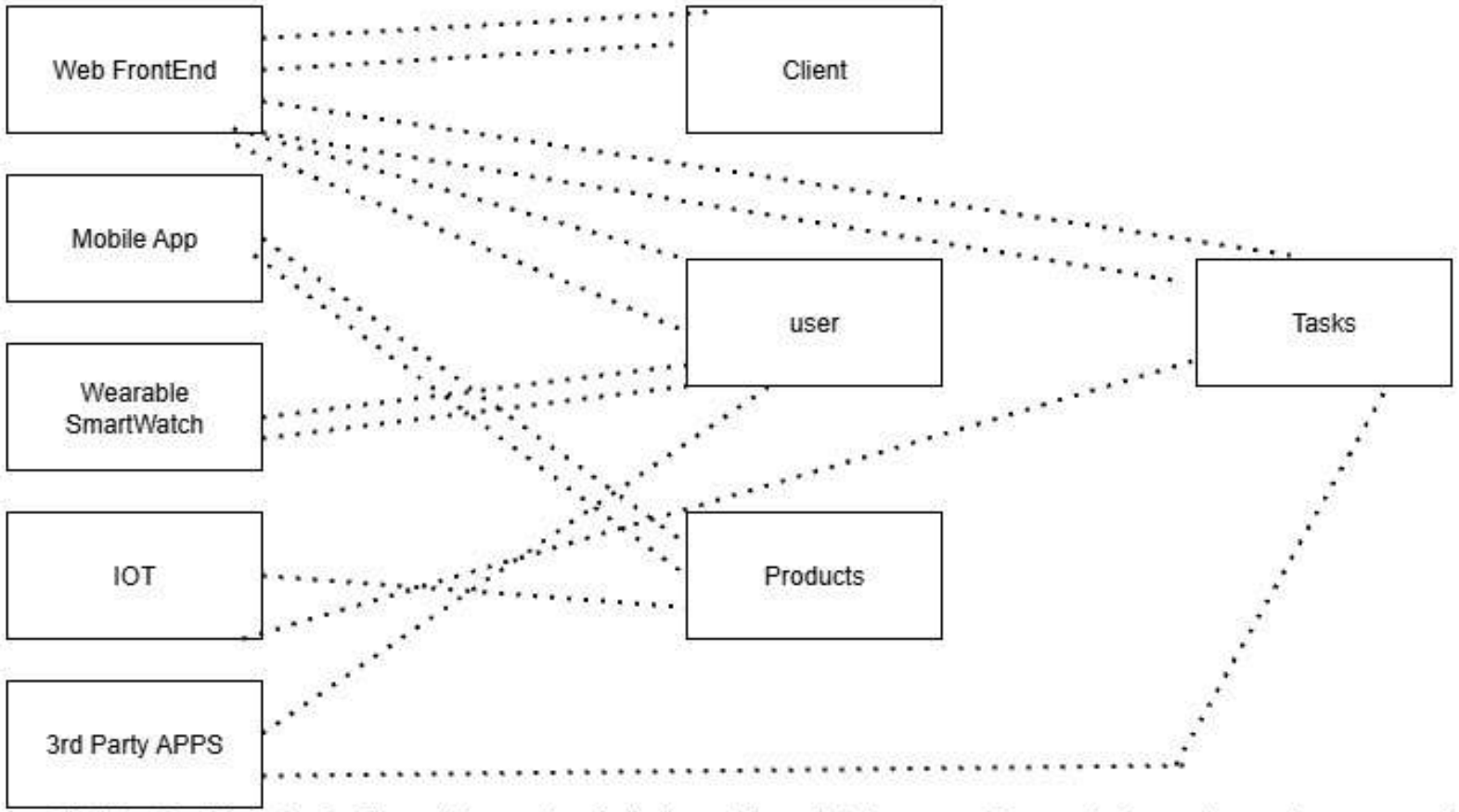
Challenges

- Ball of MUD
- LATENCY
- Service Discovery at Runtime



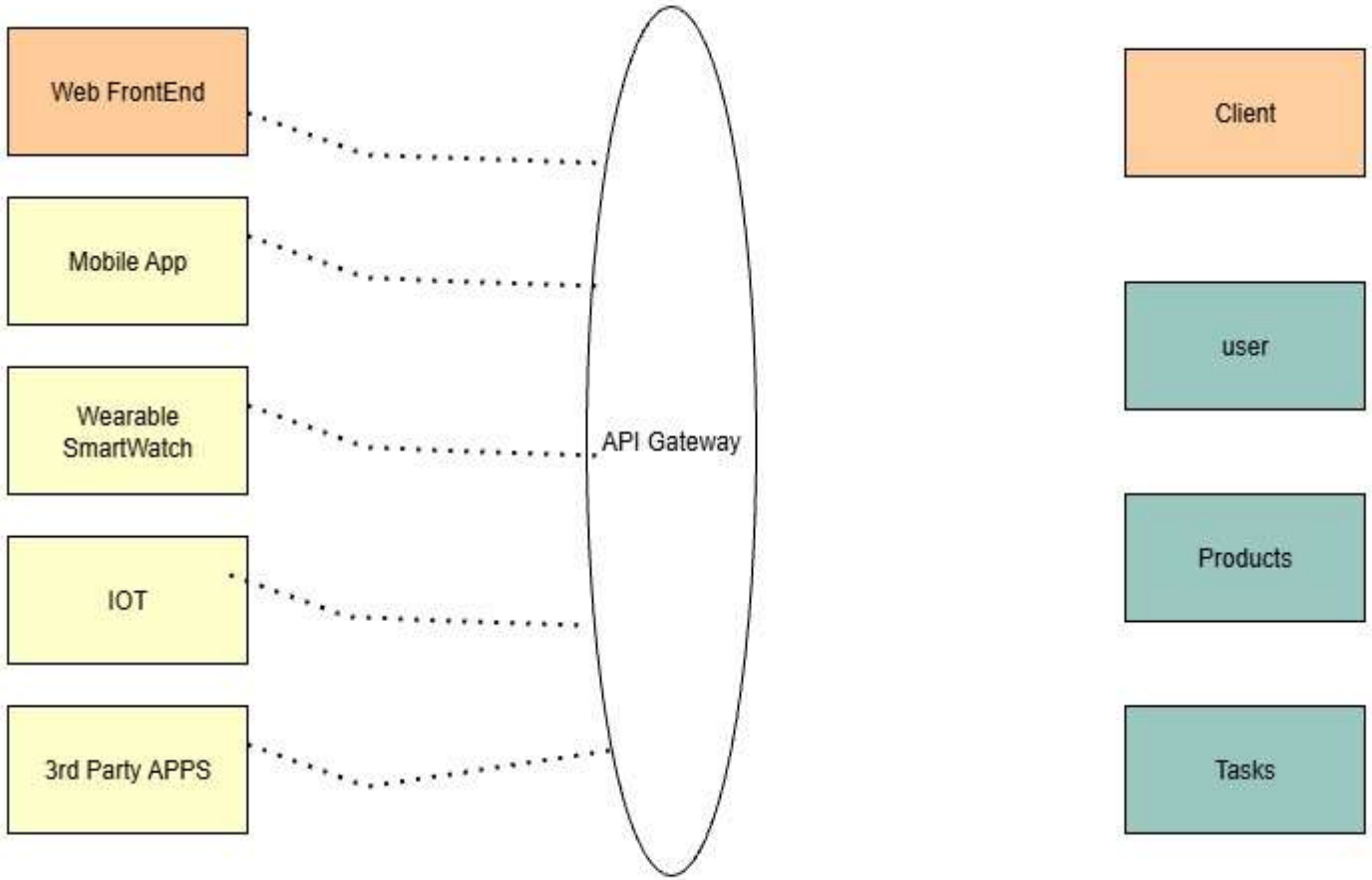
The Client Scaling Problem

The frontend has to make request to the multiple microservices to get the data if needs and at the same time there could be other client like external API'S also making request to the microservices as well as they need to integrate with each microservices- this creates a big issue
the complexity at the client side becomes exponential high and it slows down the client because excessive logic written to be able to access data effectively.



At some point of time in a lifecycle of software ,the number of clients would needs to increase .This can lead even to more issues as each client needs to interact with each microservice as it leasds no of connection and API increases and it leasds to latency.This is the point where we need to fight other customer will face very bad experience.

The Solution- Central API Gateway

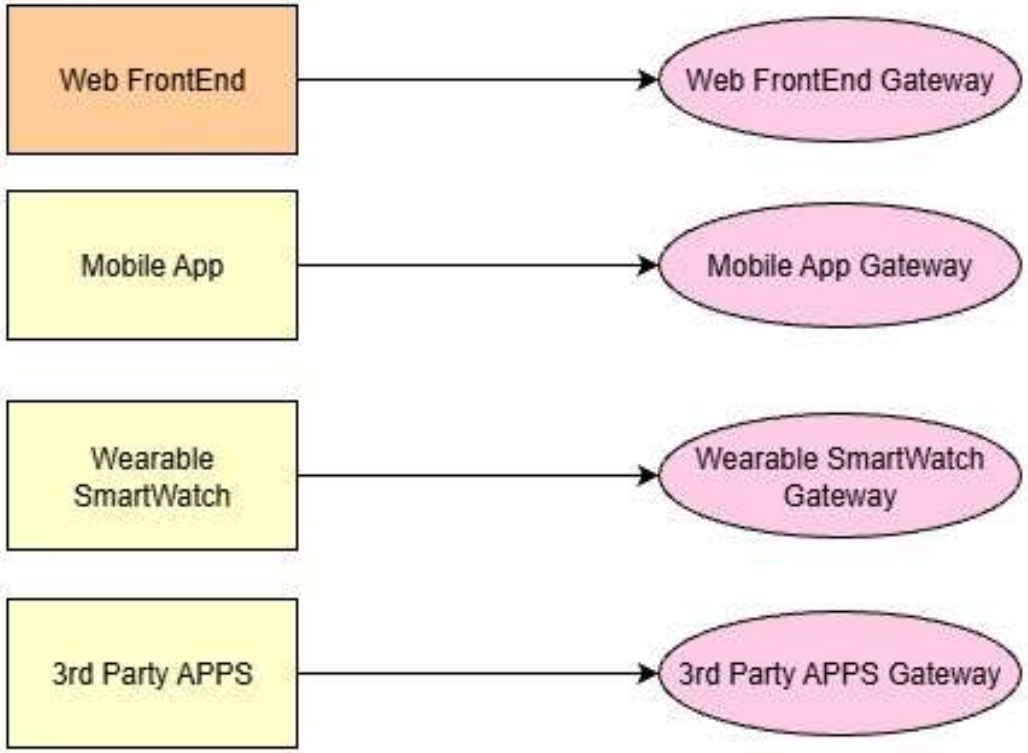


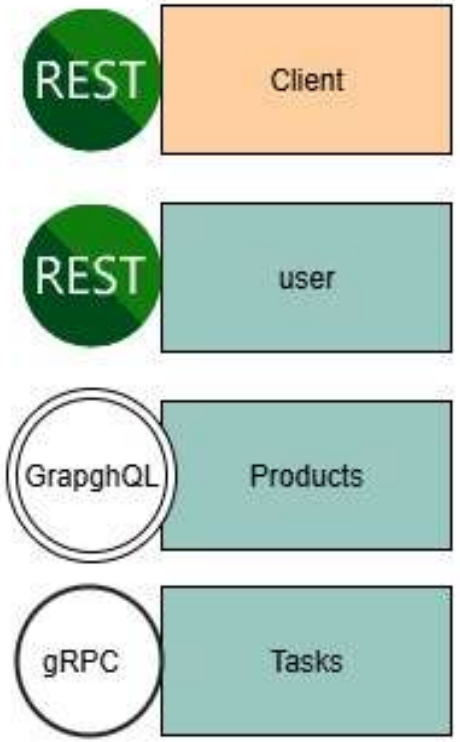
If the microservices need to increase in number and the clients also need to increase, it's wise to have a central interface between the services and the clients and the central interface, called the API gateway is responsible for interacting with with the microservices, the clients don't have direct access to them - this is quite great even from a security perspective, a compromised client doesn't bring the backend down.

The problem with a centrally available API Gateway is just that - it's centrally available. If the API gateway goes down - all clients are left hanging.

Not only that, there's another challenge - the web front end is usually more robust and can handle bigger JSON objects whereas you might want to keep the mobile app super light and send very less data in the response, this is even more true for wearable devices.

Advanced Solution- Backend For Frontend





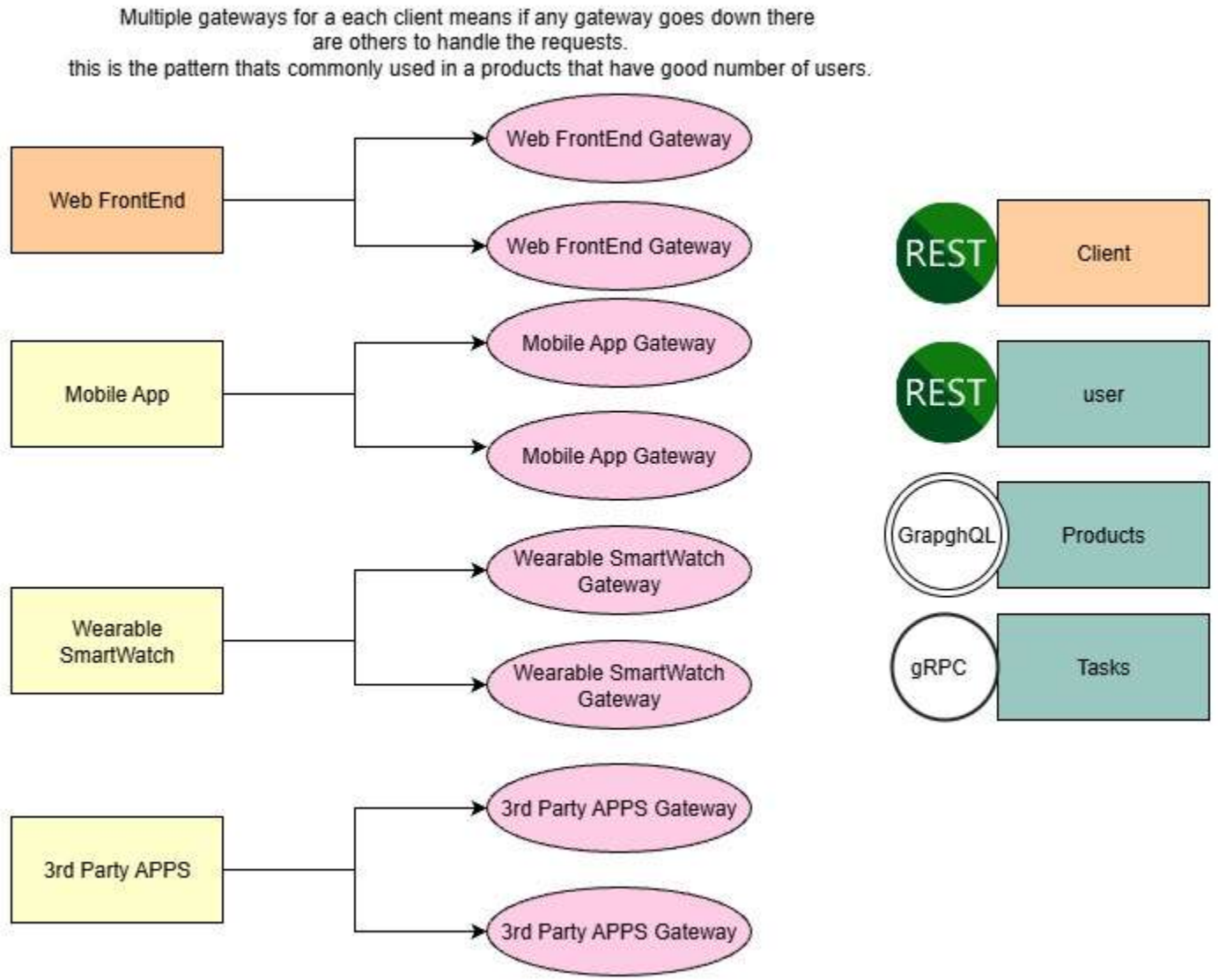
Having separate gateways for separate clients adds a whole new dimension of scalability and not only that, it brings with it flexibility and also added benefits in uptime.

Flexibility - now you can share grapght APIs with watch and mobile - this keeps the client super light and can even interact using gRPC - which is pretty awesome.

Benefits -

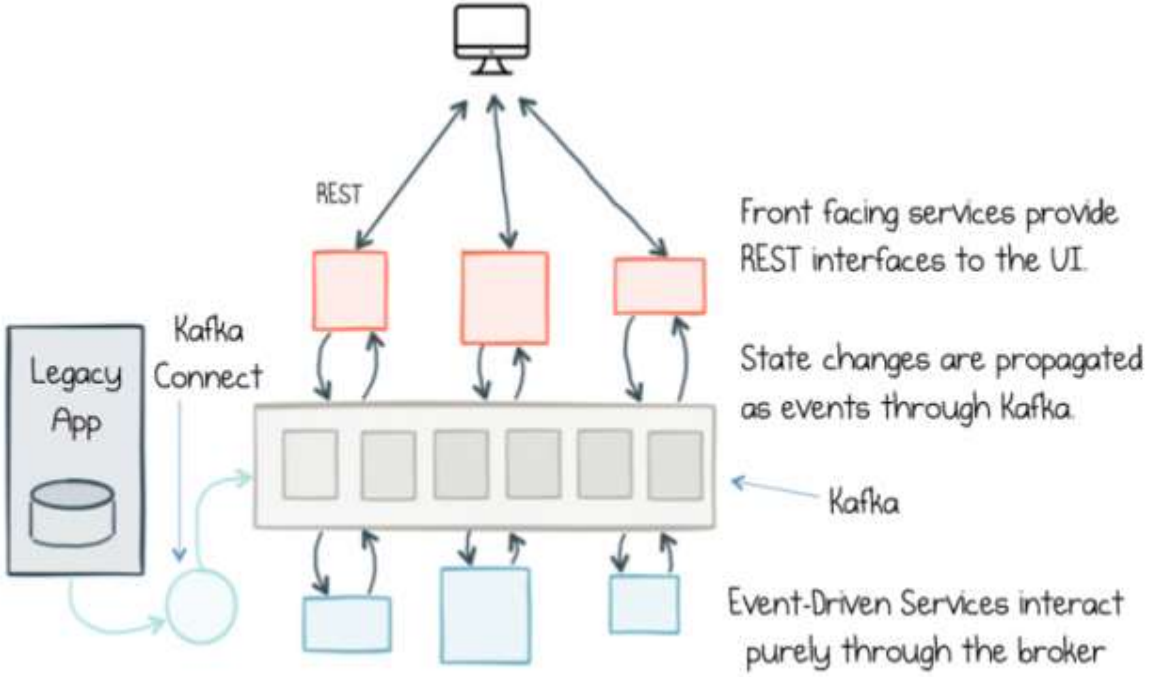
High uptime - if one galeway goes down, other clients can still keep functioning
High flexibility - use grpc, graphgl, REST - whalever you want, wherever
High scalability - number of microservices and clients can both increase without impacting the speed
Low latency - front end clients' complexity reduces signficantly, along with the number of connections, thereby leading to lower latency

Expert Solution- Distributed API Gateways



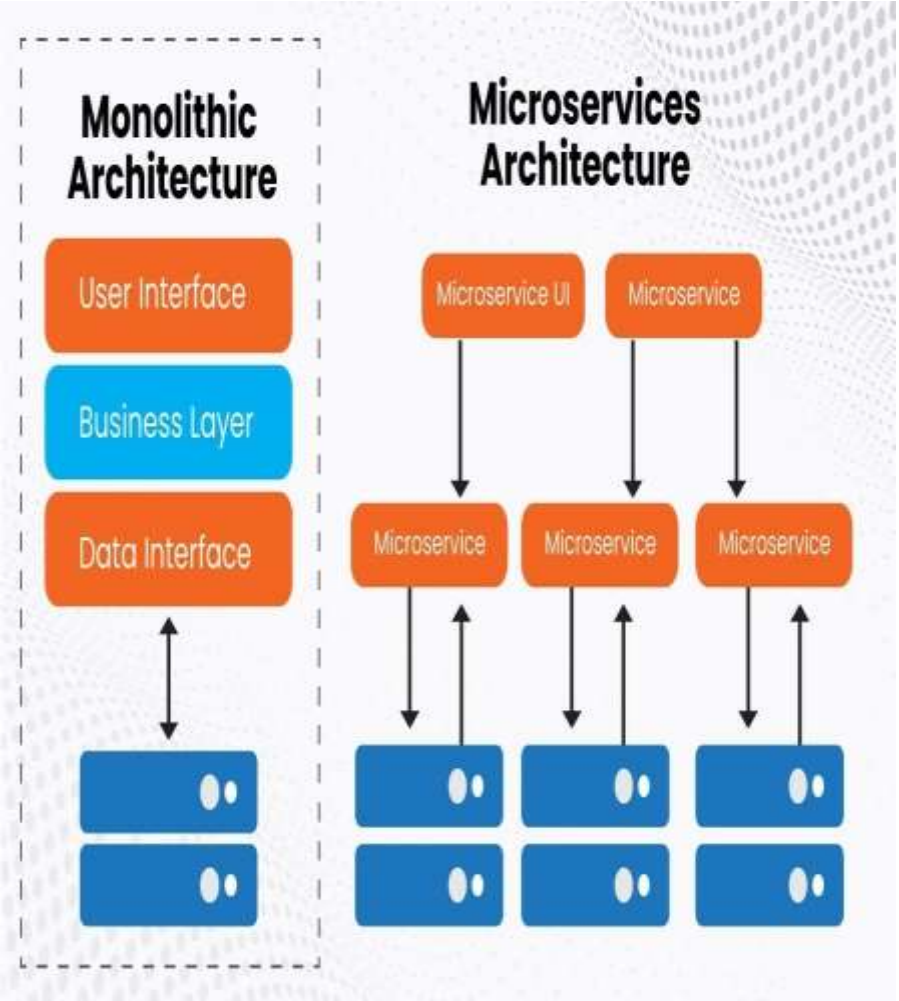
Inter Service Communication

- Things usually start off as synchronous systems with HTTP, but HTTP requests start having lag and delays.
- In traditional systems, client makes HTTP requests to server then gets response back.
- Streaming systems make it easy for everyone to be in sync without specifically making requests.
- So, systems tend to become more event driven with time as traffic and data increases and no, of requests start burgeoning.
- For a while, architects also try partially being request driven and partially event driven, and this is how it looks.
- But Eventully they may become event driven.

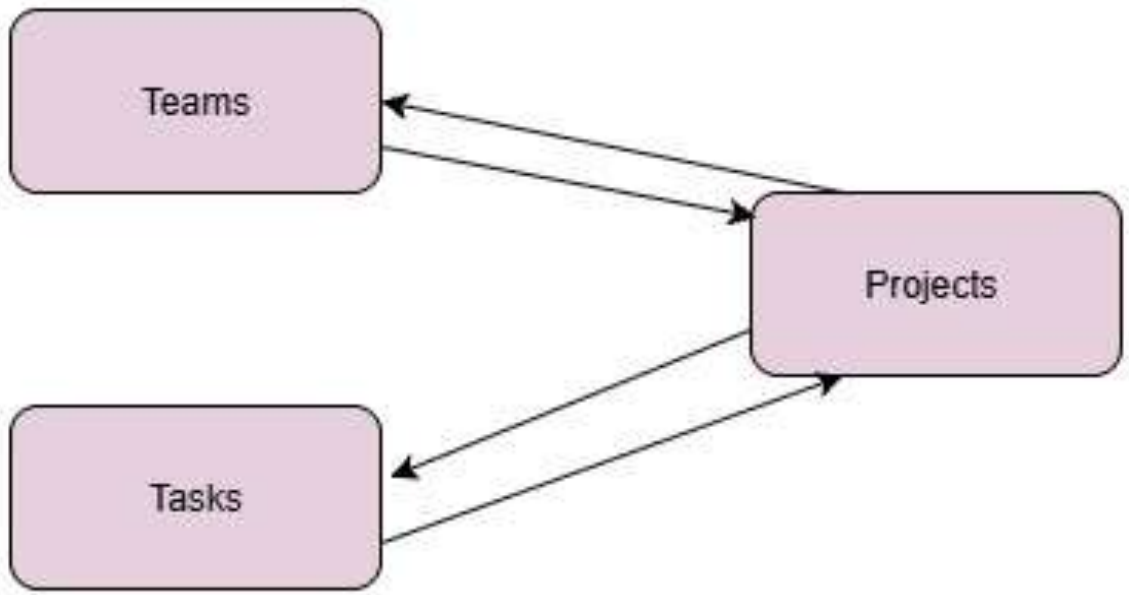


Inter Service Communication

- When teams grow and the product matures and the traffic increases.
  These are independent event but mostly happen at the same time.
  This calls for a new way of dealing with this challenge.
- Different technologies are great for different services. you may want to build services heavy with computations in GO, scraping services in python and general services in node also, you might only get few engineers of a technology and you'd want to keep your system language agnostic.
- For such evolved systems, where there's loose coupling and high separation of concern, you'd want to consider microservices.
- For such evolved systems, where there's loose coupling and high separation of concern, you'd want to consider microservices.
- Architecting REST API based microservices or synchronous microservices is easy, straightforward and at a small scale the right thing to start with and at a small scale, everything works perfectly fine.
- But soon there will be challenges with synchronous microservices - all microservices exclusively own their own data so if one service wishes to access data help by another, it must do so using an API call.
  when numerous services access the same piece of data, things get tricky. to complicate matters further, you may have microservices that utilize heterogeneous databses, i.e multiple types of DBs like SQL, NOSQL, timeseries.
- Maintaining data integrity can be an issue when the requests don't scale from Apls
- Instead of AP| calls, we want the systems to being driven by events. where events are streamed between the microservices and they pick up events that are important to them while ignogring the others.
  Based on the events, there could be triggers that take specific action.

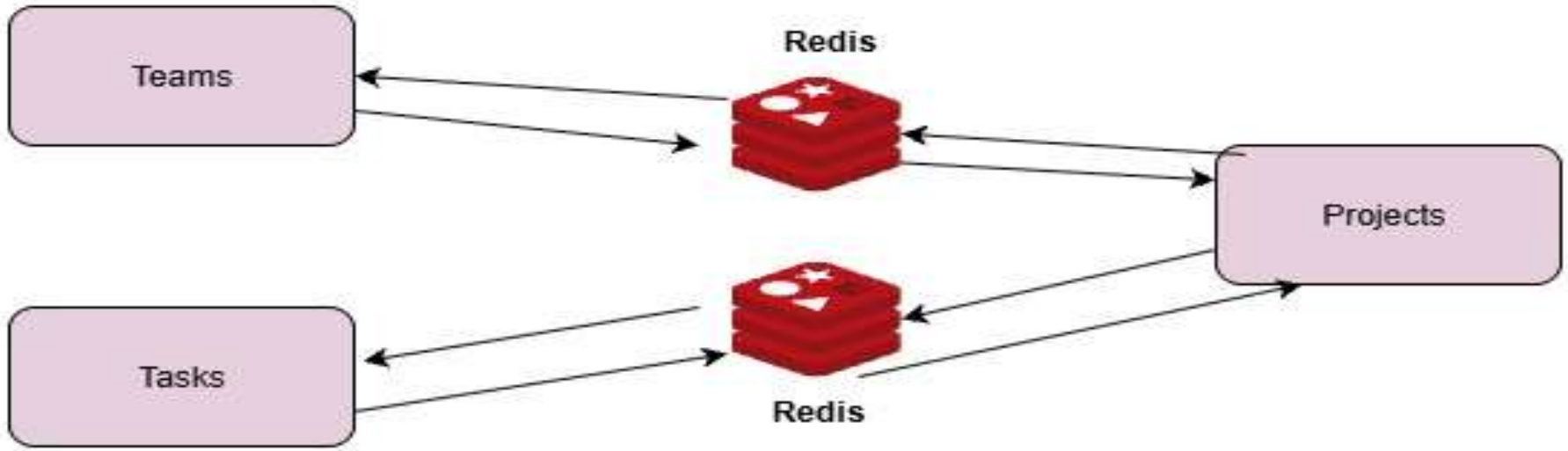


Request,Response,synchronous communication,doestnt scale and leads to ball of mud communication

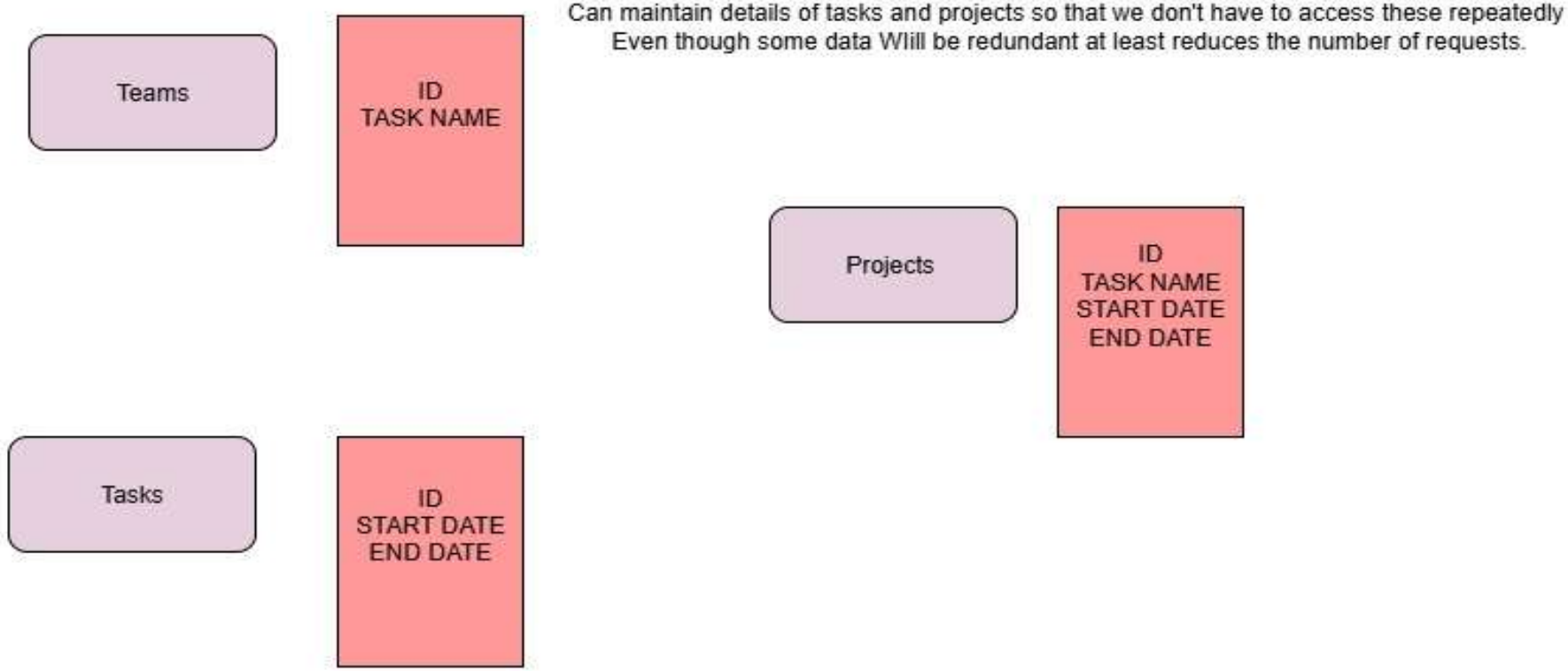


CACHING

Don't have to always hit the microservice, can use caching so that latency isn't compromised and, cache can be updated at a later time.

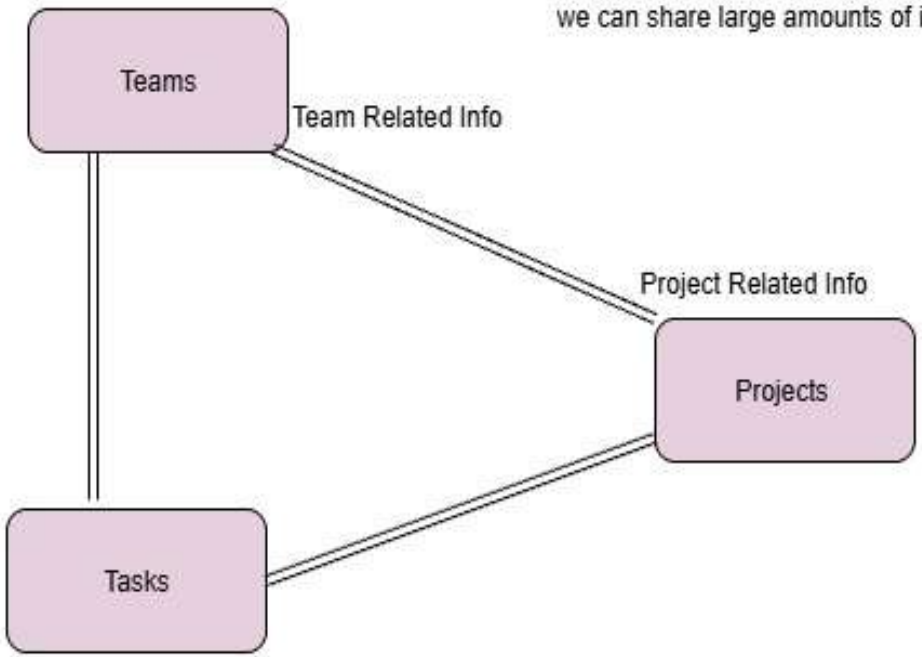


Redundant Data Storage



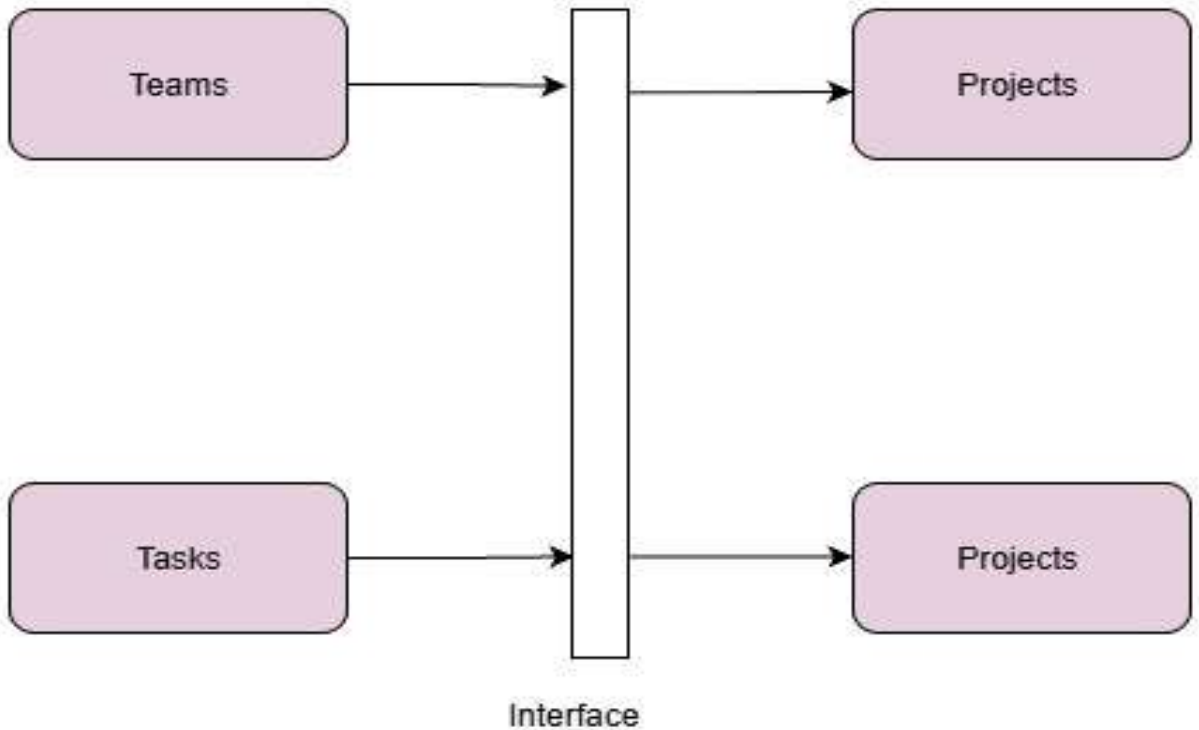
Streaming Between Services

Can start streaming information between microservices - this suadenly makes our architecture asynchronous and we can share large amounts of information but many things can go wrong here if we re not careful.



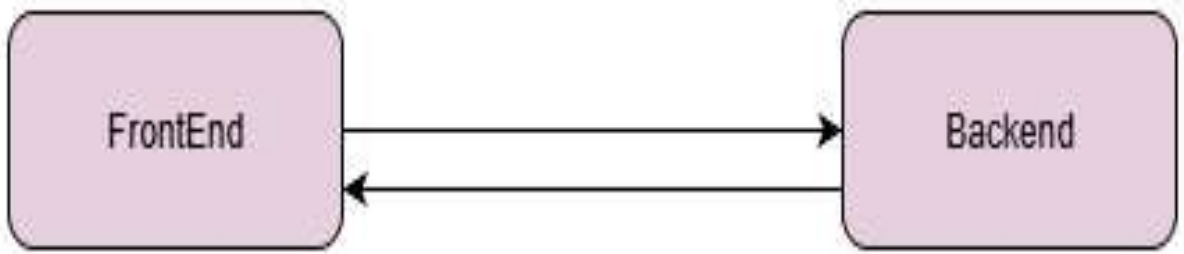
Streaming Between Services

Using an interface like Kafka / RabbitMQ (queues) we can send "events"
This type of a format is called a publisher subscriber model where each microservice can events and all microservices can subscribe to the type of events they want to listen to.

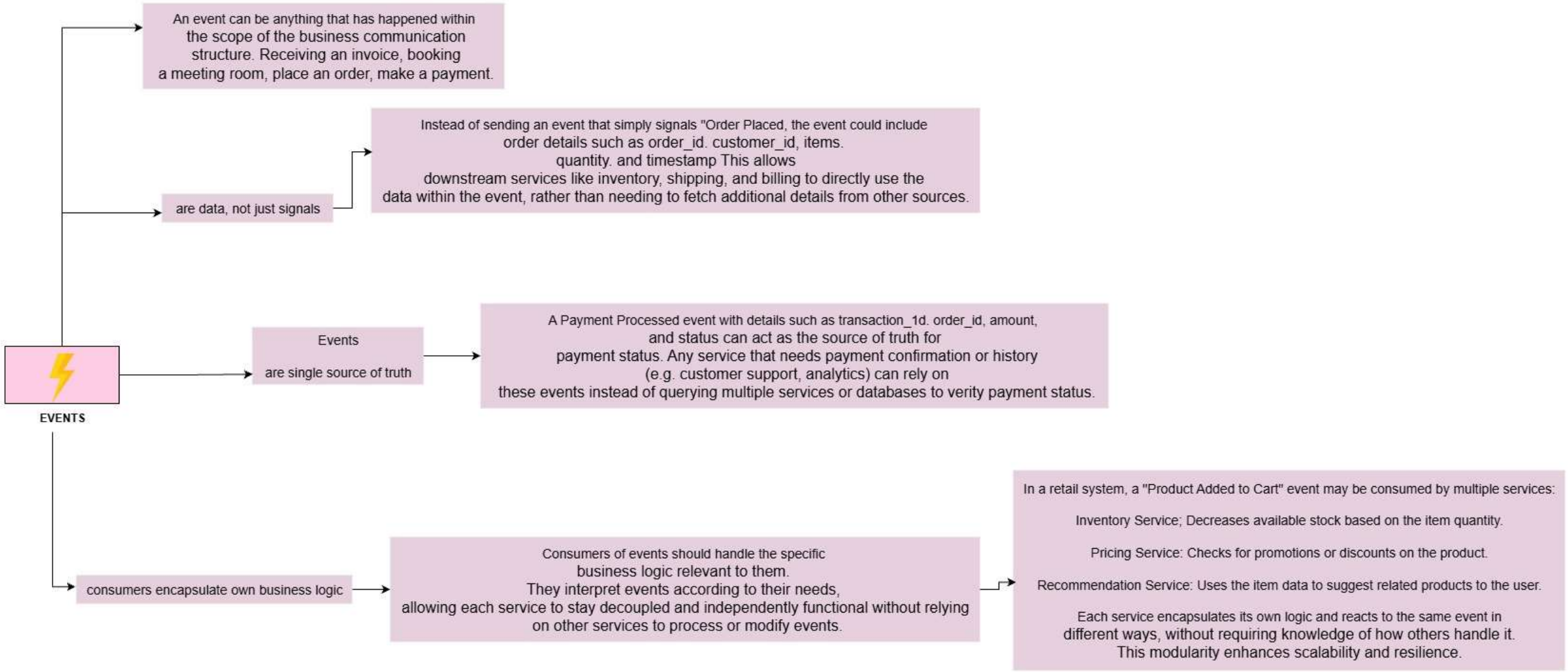


Streaming Between FrontEnd and BackEnd

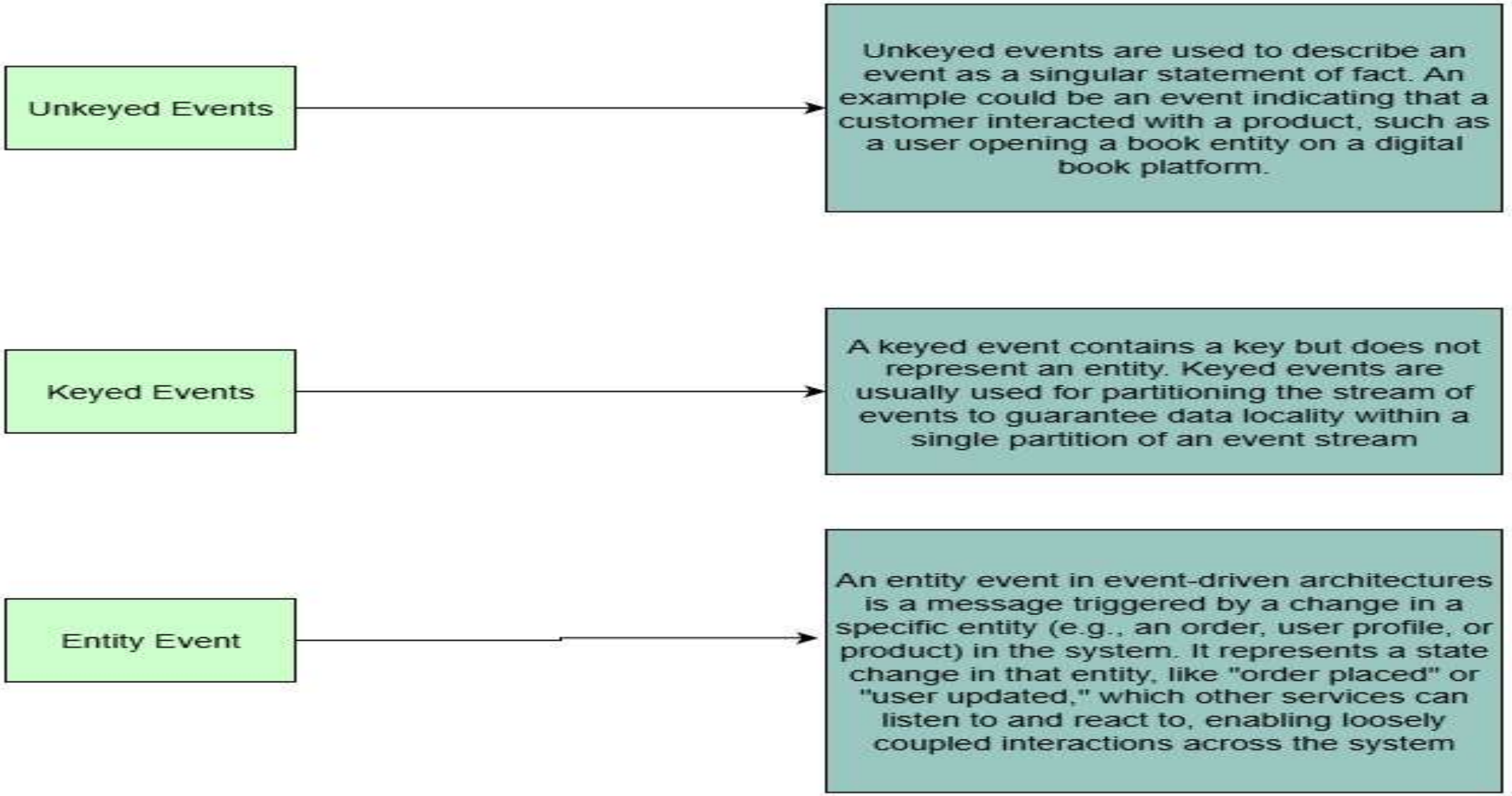
Don't have to just stream events between microservices, we can also stream events between the front end and the backend for
faster updating of the front end, which reduces the number of APi calls to the backend
Also, it's easy to set this up using sockets.



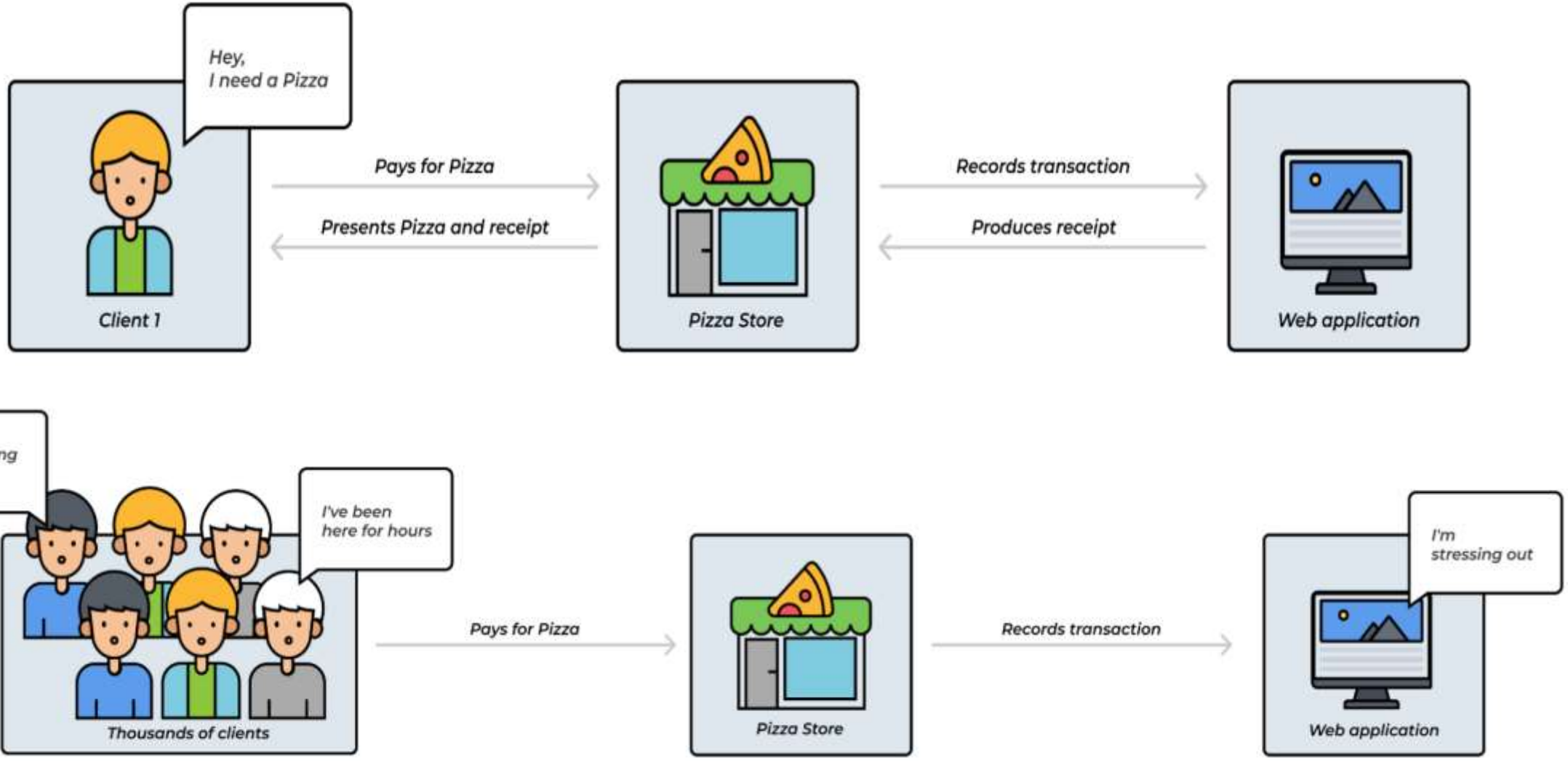
Event-Driven Architectures



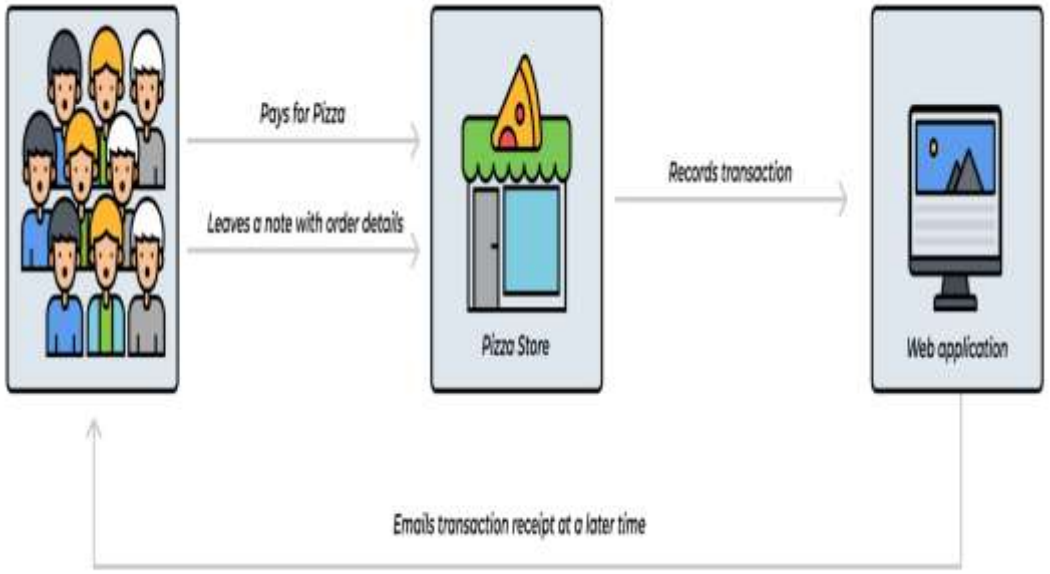
EVENT TYPES

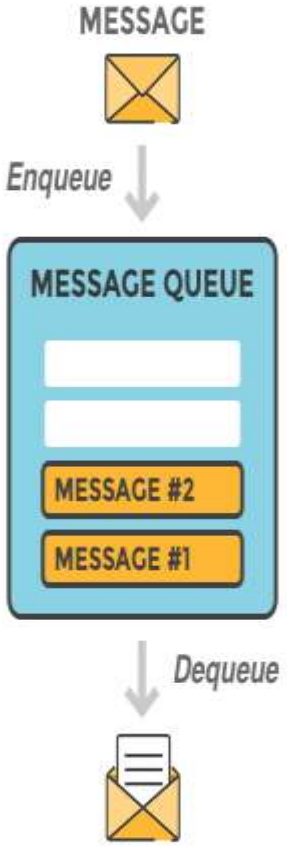

Unkeyed Events
Unkeyed events are used to describe an event as a singular statement of fact. An example could be an event indicating that a customer interacted with a product, such as a user opening a book entity on a digital book platform.
Keyed Events
A keyed event contains a key but does not represent an entity. Keyed events are usually used for partitioning the stream of events to guarantee data locality within a single partition of an event stream
Entity Event
An entity event in event-driven architectures is a message triggered by a change in a specific entity (e.g., an order, user profile, or product) in the system. It represents a state change in that entity, like "order placed" or "user updated," which other services can listen to and react to, enabling loosely coupled interactions across the system

Message Queues



Message Queues





Pub-Sub Model

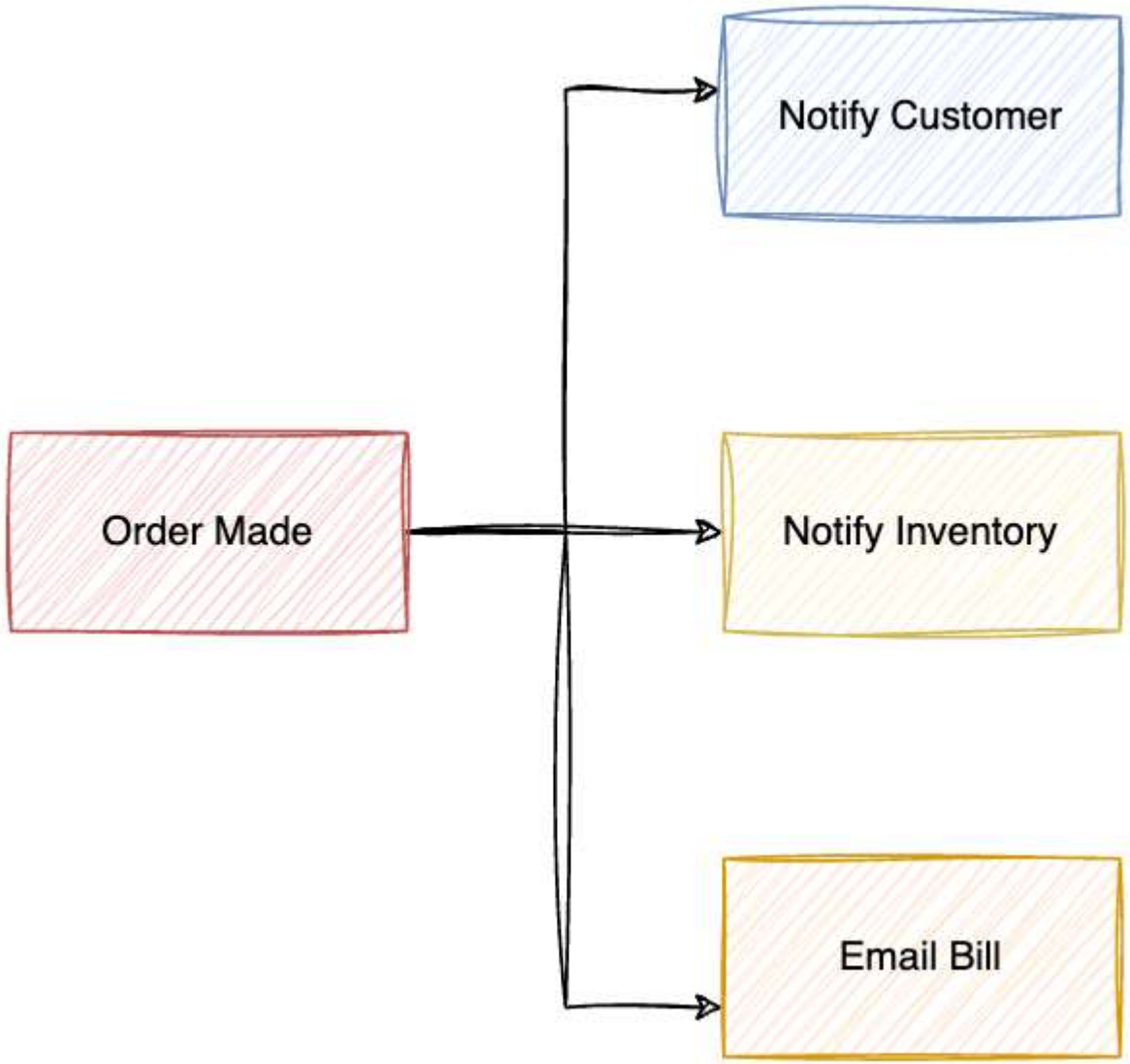


Figure: Processing an order

Pub-Sub Model

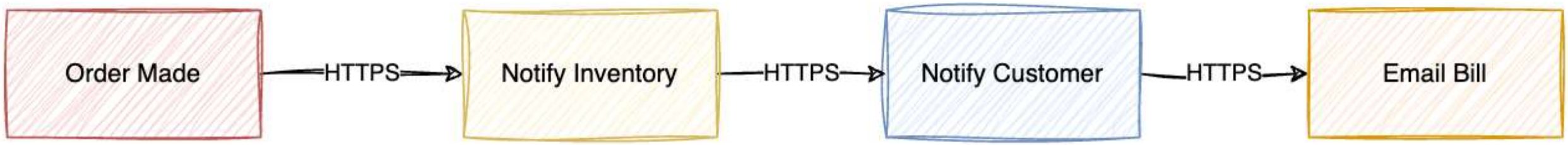
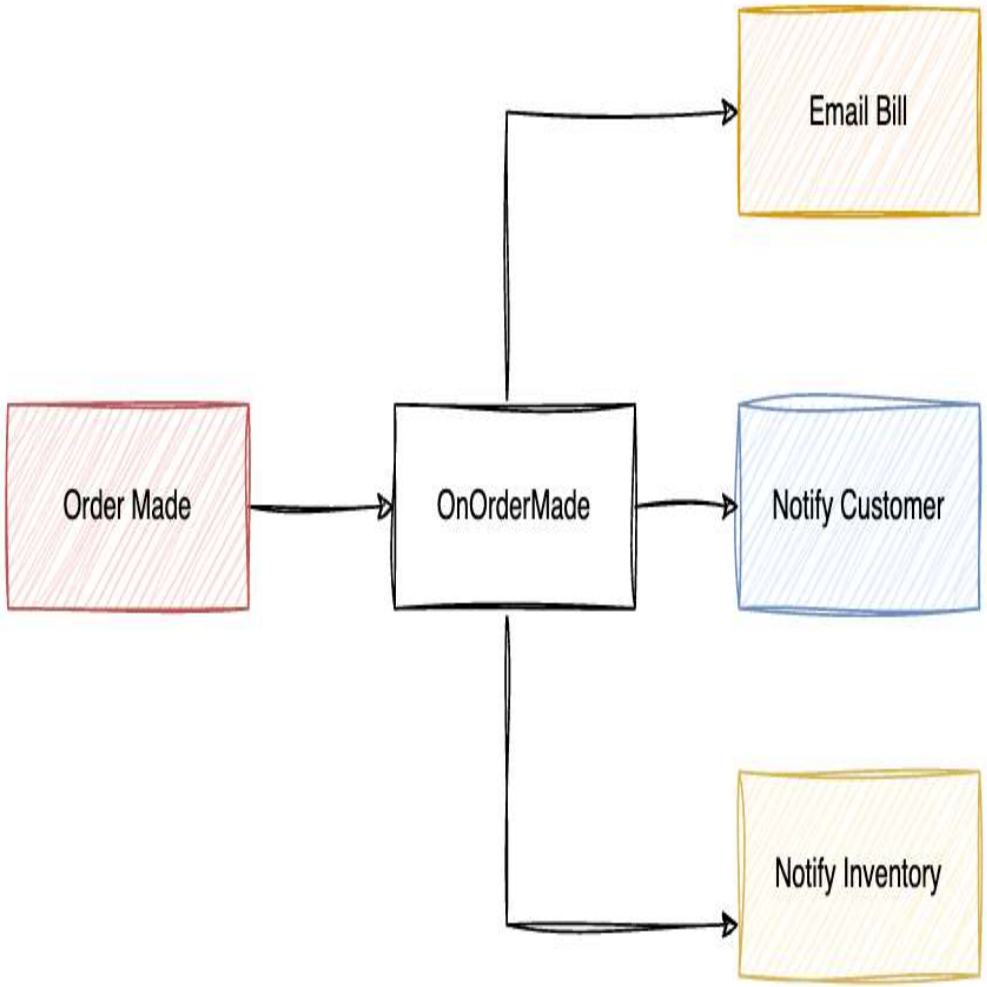


Figure: A RESTful implementation of order fulfillment

Pub-Sub Model



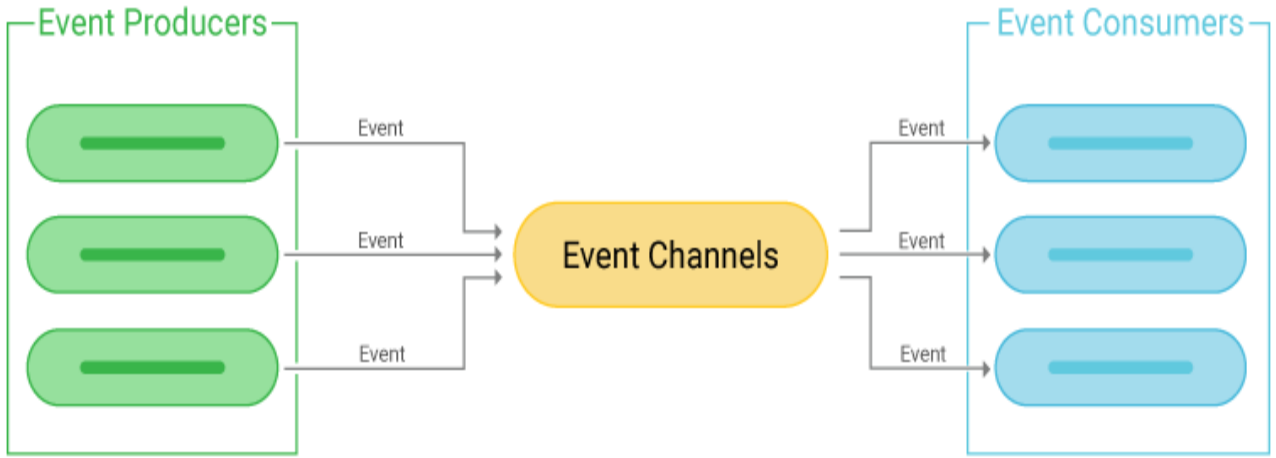
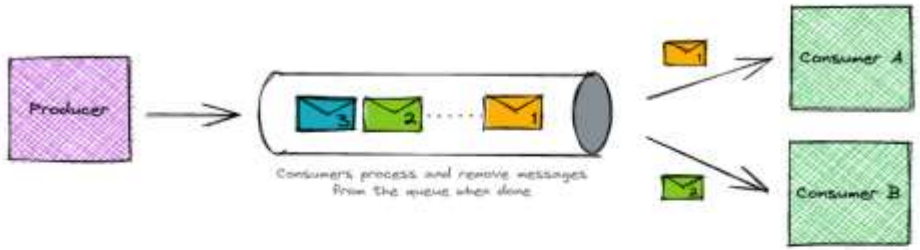
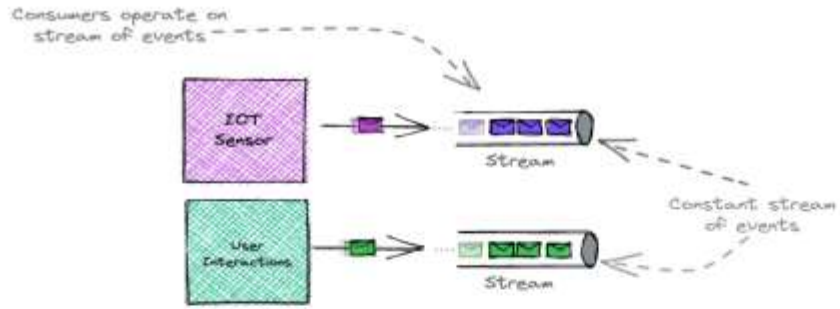


Figure: Migrating architecture toward a Pub-Sub messaging model



Queues

Consumers pull messages, process and remove



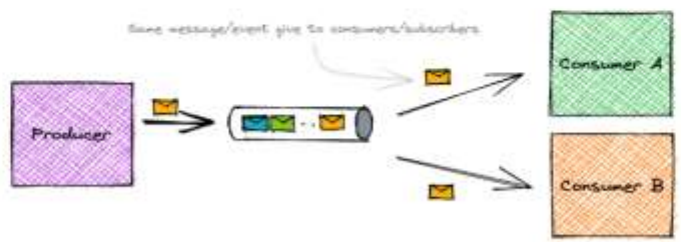
Streams

Unbounded series of events
Common examples include user click streams / IOT / transactions

Queues vs Streams vs Pub/Sub

Bite sized visual to help understand the differences

@boyney123



Pub/Sub

Publish messages/events to many subscribers
Each subscriber gets copy of event to process

Thank You !!!!!!!!!!!!!!!