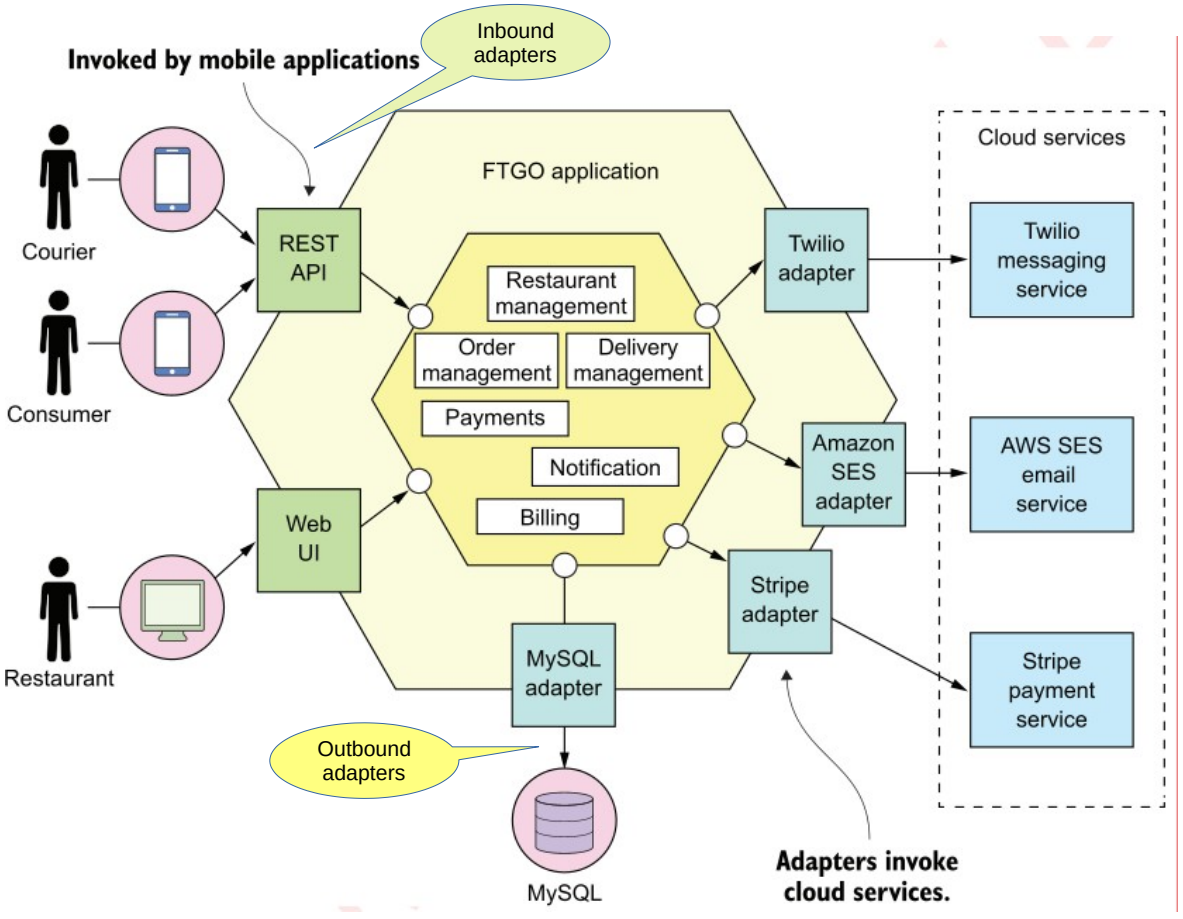


BIG Ball
Of
MUD

Application has a hexagonal architecture.

It consists of business logic surrounded by adapters that implement UIs and interface with external systems, such as mobile applications and cloud services for payments, messaging, and email.

Benefits of Monolithic Arch

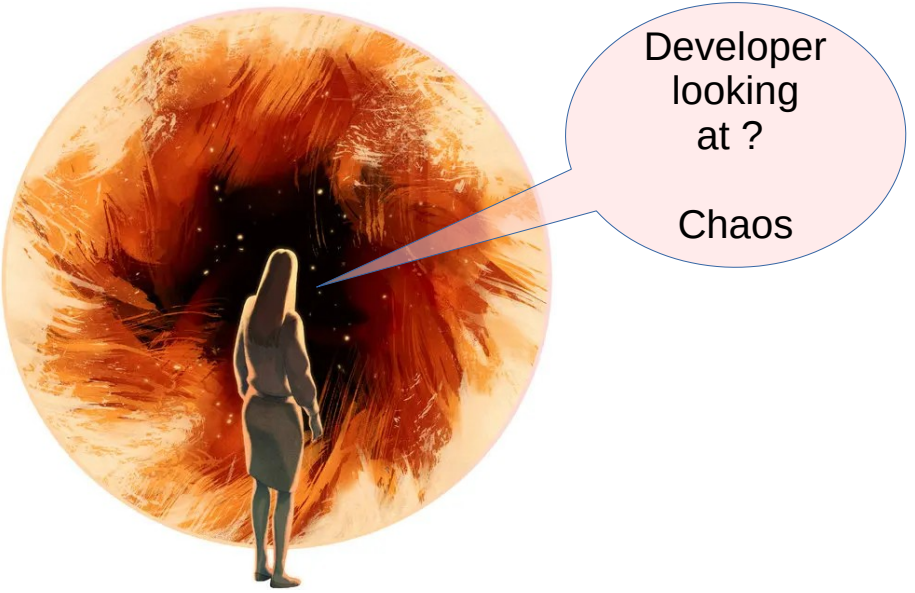
a) Simple to develop

b) Making radical changes is easy to the application

c) Testing is straightforward

d) Scaling it behind the load balancer is easy

Monolithic Hell



Complexity intimidates developers

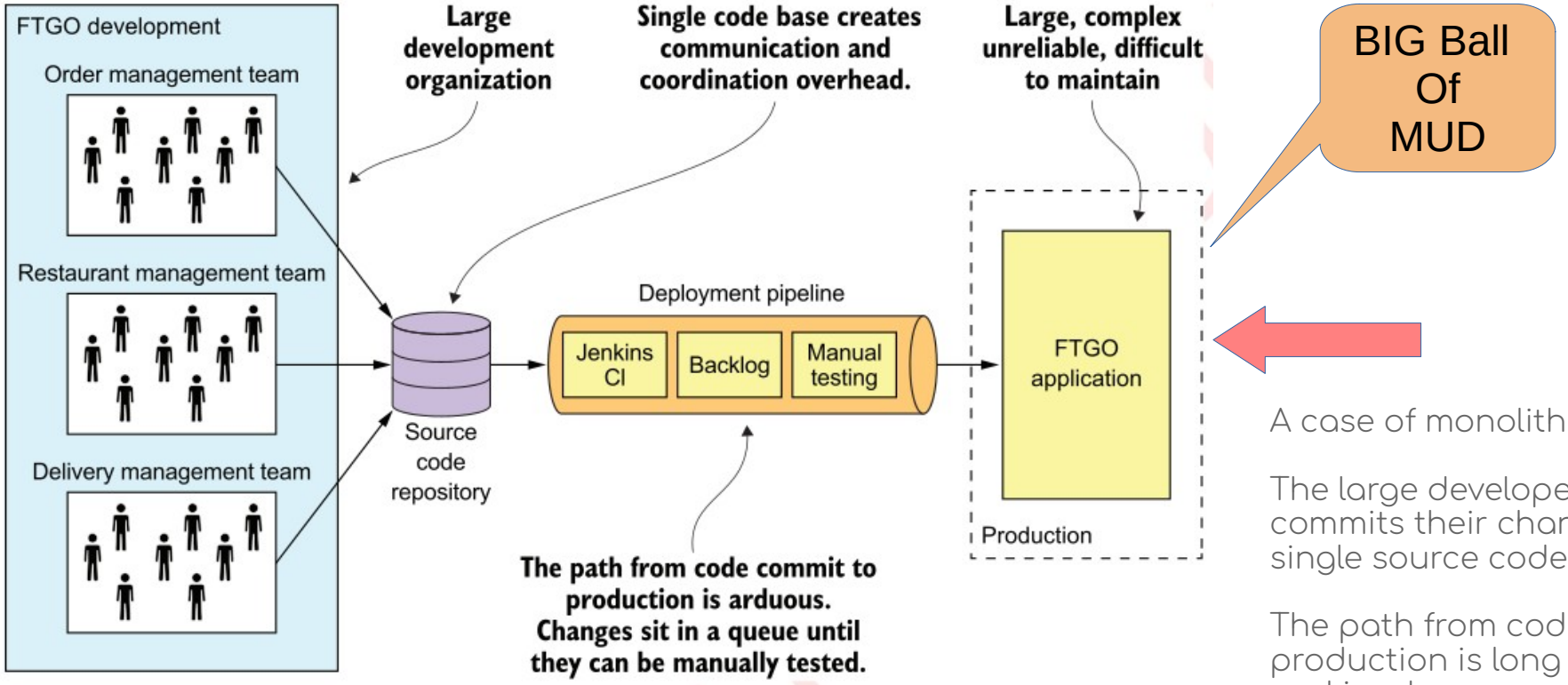
- As features are added to the app

  Team grows, code grows, -

  - increases management overhead
  - Becomes too large for any developer to handle.....
  - Developement is slow
  - Takes long time from commit to deployment – continuous deployment is great challenge
  - Testing takes lot of time, code stabilization is problem as different teams commit causing merging issues.
  - Some Manual and automated testing needs to be done as it cannot be automated fully
  - Fixing code issues takes days – continuos integration is a challenge
  - Scaling is diff – in memory, cpu intensive apps parts cannot leverage on h/w resource as they are part of same app.

- lack of reliability, testing the app thoroughly is a challenge, thereby resulting in bugs, causing crashes.
- Technology stack cannot be upgraded, living with obsolete stack makes it diff to adapt to latest trends.
- Using the new framework for app, will be to rewrite the whole app again - ????
- Optimizing the app is NONO.

The slow march toward monolithic hell



BIG Ball Of MUD

A case of monolithic hell.

The large developer team commits their changes to a single source code repository.

The path from code commit to production is long and arduous and involves manual testing.

The application is large, complex, unreliable, and difficult to maintain.

Microservices

For microservices, we need to know

- Three tier architecture
- Web application design
- How to develop business logic using object oriented design
- How to use RDBMS, SQL & ACID transactions
- How ti use IPC using message brokers & REST Api's
- Security – Auth & Authorization

Learning ?

Essential Characteristics of Microservice architecture – benefits,drawbacks.

Why and When to use microservices architecture

Distributed data management patterns

Effective strategies for testing

Deployment options

Strategies for refactoring a monolithic app into microservice architecture

Learning ?

Architect an application

Develop business logic for a service

Use SAGAS for maintaining data consistency across services

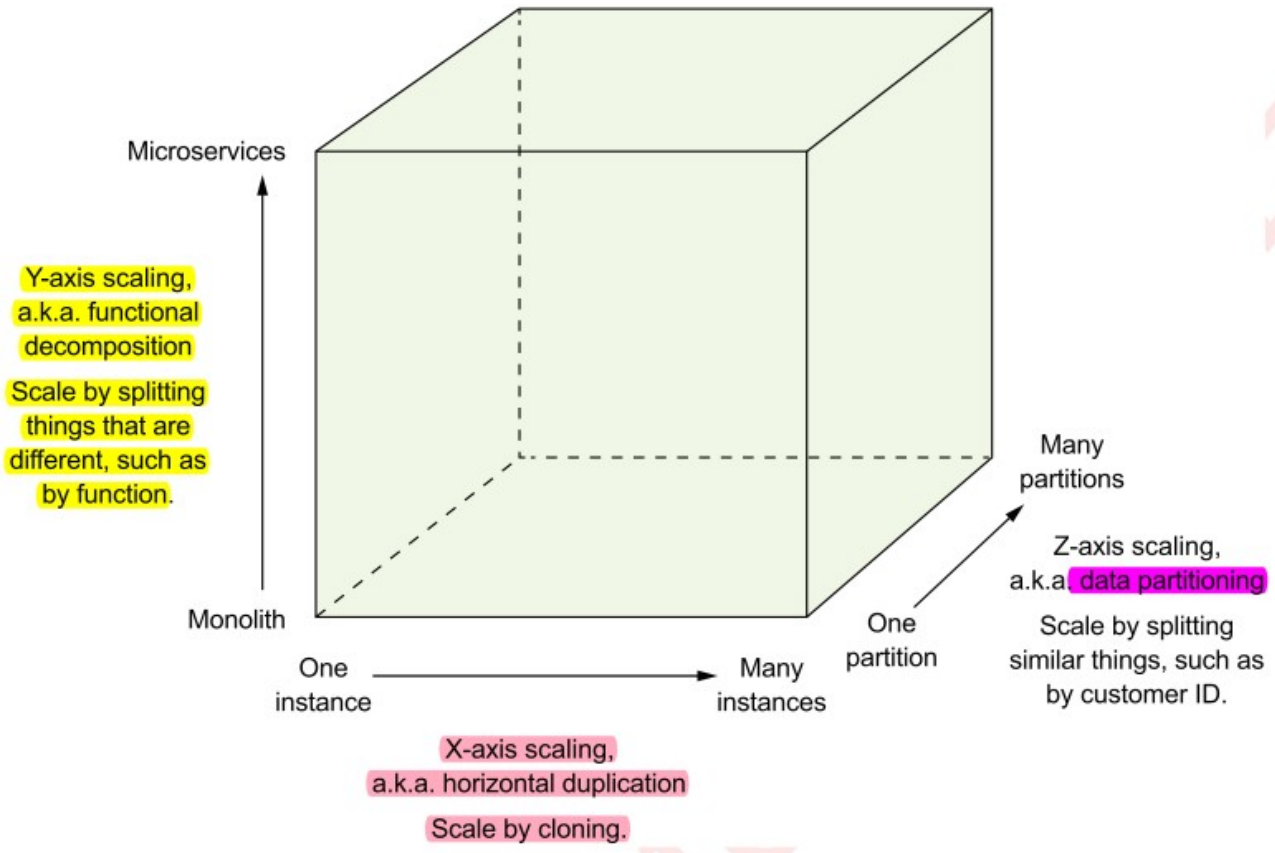
Implement queries that span services

Test microservices

Defining Microservices Architecture

Microservice architecture as a service-oriented architecture composed of *loosely coupled* elements that have *bounded contexts.*

Scale Cube & Microservices



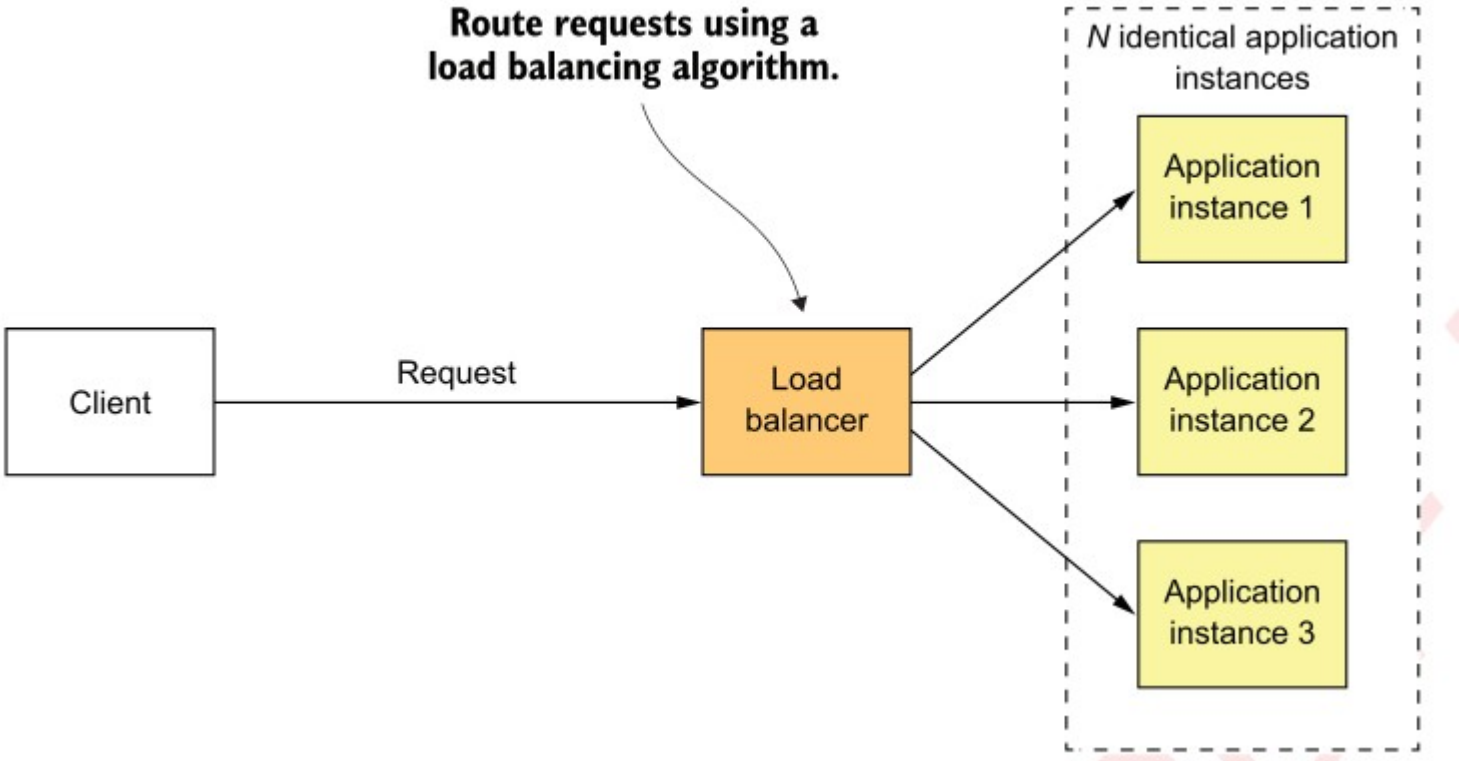
scale cube defines three separate ways to scale an application:

X-axis scaling load balances requests across multiple, identical instances;

Z-axis scaling routes requests based on an attribute of the request;

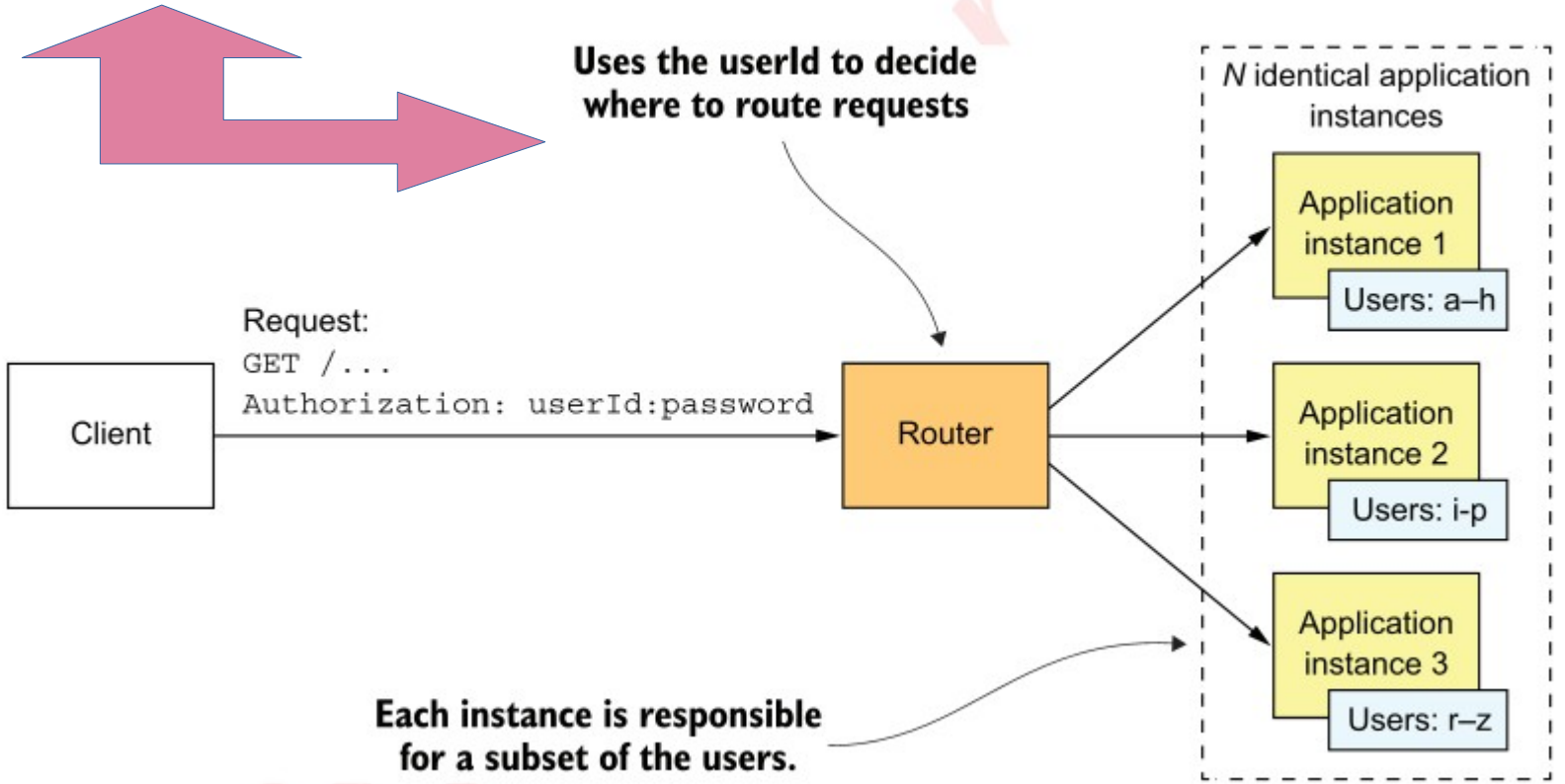
Y-axis functionally decomposes an application into services.

X-axis scaling runs multiple, identical instances of the monolithic application behind a load balancer.



Z-axis scaling runs multiple identical instances of the monolithic application behind a router, which routes based on a **request** attribute .

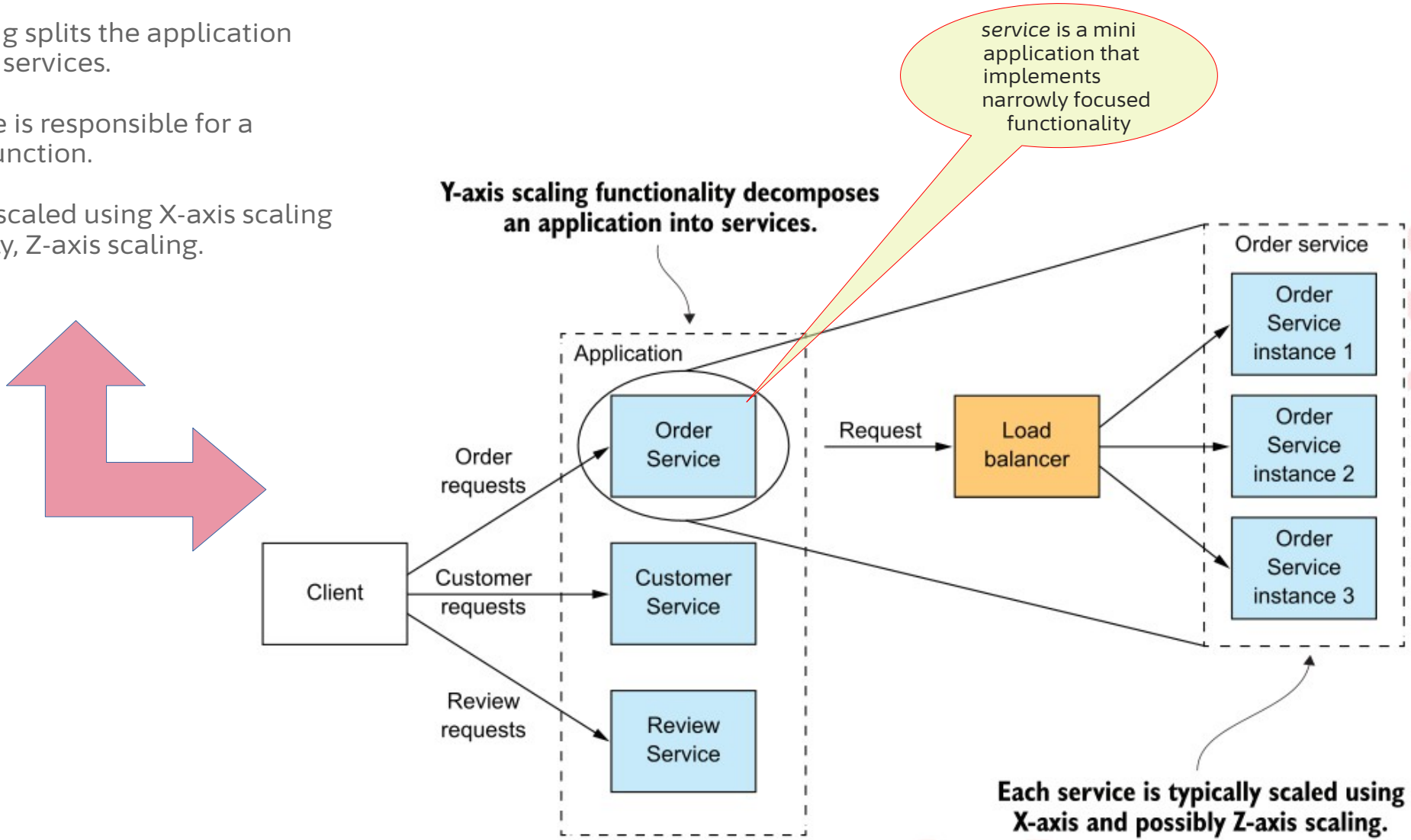
Each instance is responsible for a subset of the data.



Y-axis scaling splits the application into a set of services.

Each service is responsible for a particular function.

A service is scaled using X-axis scaling and, possibly, Z-axis scaling.



Modularity of microservice

Modularity is essential when developing large, complex applications.

Applications must be decomposed into modules that are developed and understood by different people.

The microservice architecture uses services as the unit of modularity.

A service with it's API has impermeable boundary which cannot be violated.

Services Own Database

Key characteristic of the microservice architecture is that the services are **loosely coupled** and *communicate only via APIs.*

Way to achieve loose coupling is by each ***service having its own datastore.***

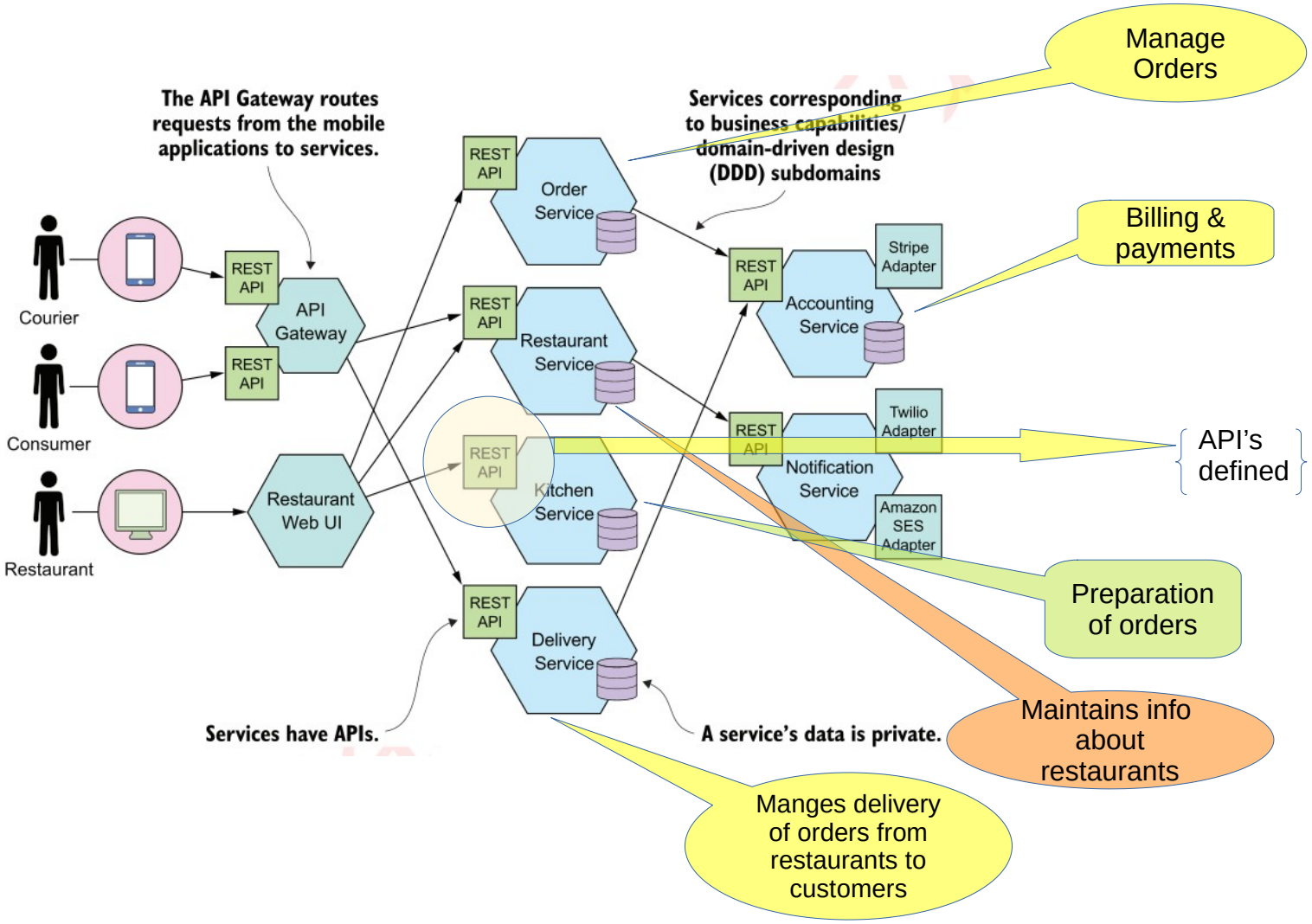
e.g

Order Services has it's own database with it's orders table

Support Services has it's own database with it's – support table etc....

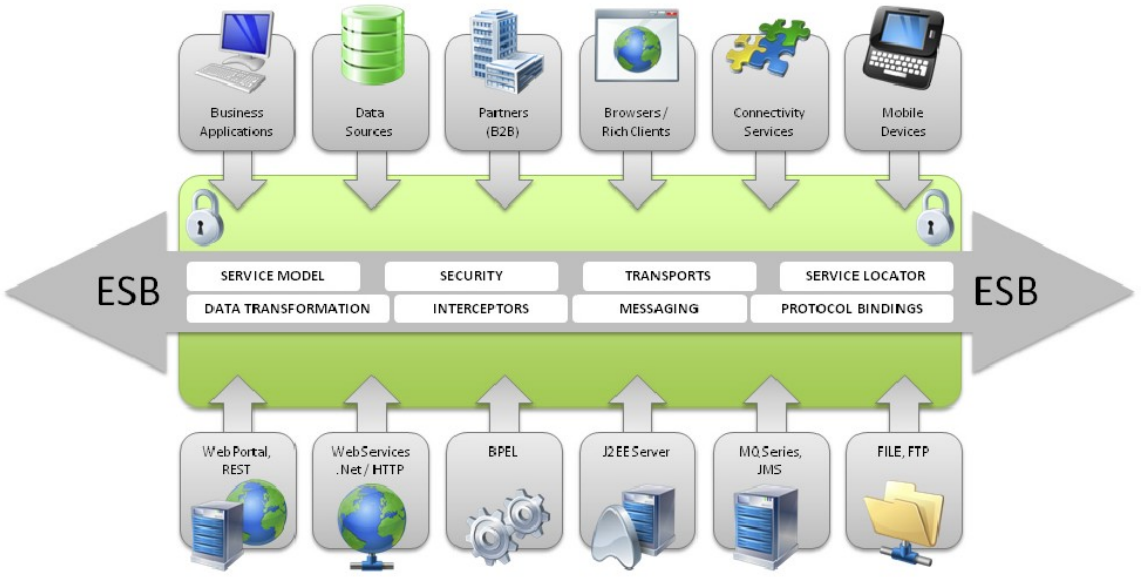
Advantage -

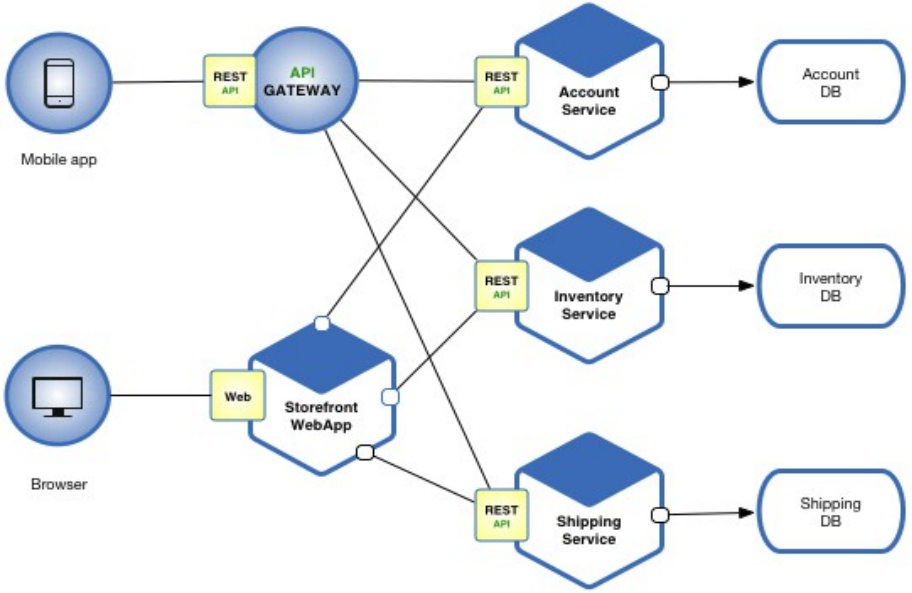
- Service's schema can be changed without having to coordinate with developers working on other services.
- During runtime services are isolated from each other, thereby never blocking another services holds database locks.


The API Gateway routes requests from the mobile applications to services.
Services corresponding to business capabilities/ domain-driven design (DDD) subdomains
Courier
Consumer
Restaurant
REST API
API Gateway
Restaurant Web UI
Order Service
Restaurant Service
Kitchen Service
Delivery Service
Accounting Service
Stripe Adapter
Notification Service
Twilio Adapter
Amazon SES Adapter
Services have APIs.
A service's data is private.
Manage Orders
Billing & payments
API's defined
Preparation of orders
Maintains info about restaurants
Manges delivery of orders from restaurants to customers

Comparing the microservice architecture and SOA

	SOA	Microservices
Inter-service communication	Smart pipes, such as Enterprise Service Bus, using heavyweight protocols, such as SOAP and the other WS* standards.	Dumb pipes, such as a message broker, or direct service-to-service communication, using lightweight protocols such as REST or gRPC
Data	Global data model and shared databases	Data model and database per service
Typical service	Larger monolithic application	Smaller service





Benefits

Enable ***continuous delivery*** and ***deployment*** of large, complex applications.

Services are small and easily maintained. Service starts faster.

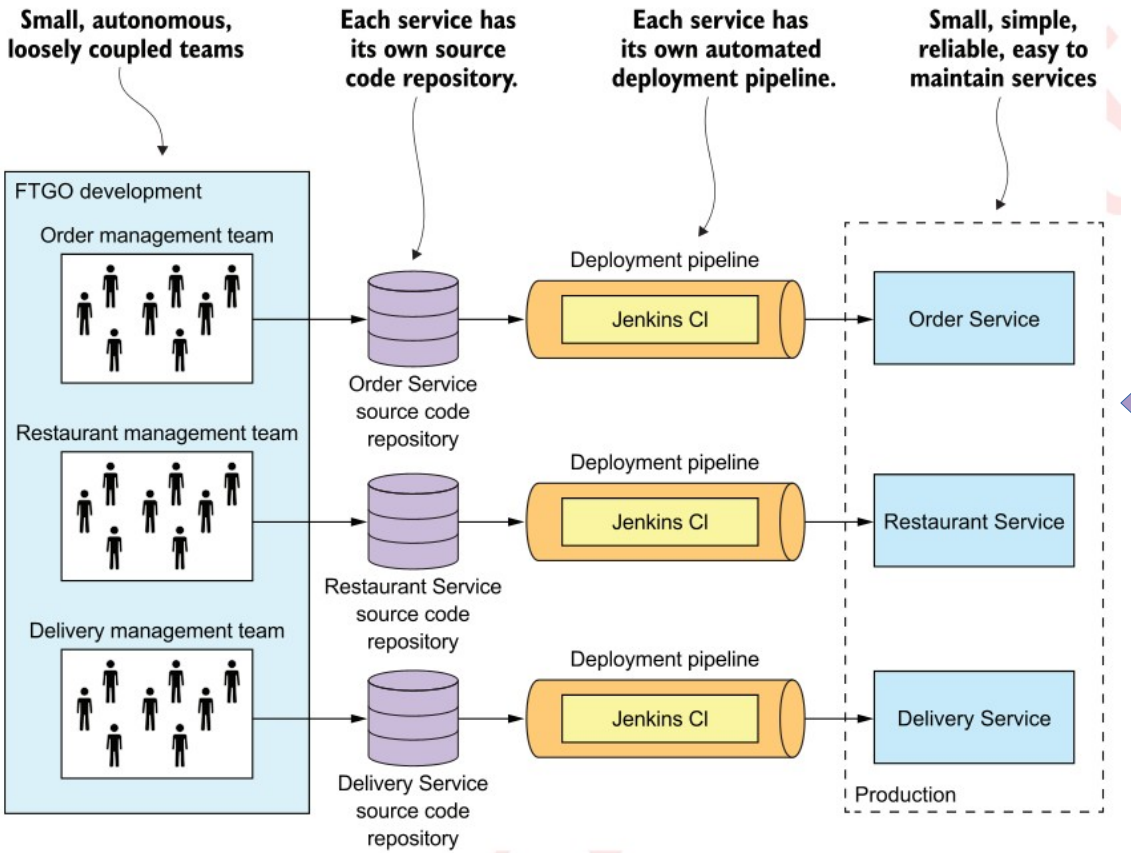
Services are **independently deployable**.

Services are independently scalable.

The microservice architecture enables teams to be autonomous.

It allows easy **experimenting** and adoption of new technologies.

It has better fault isolation.



The microservices-based application consists of a set of loosely coupled services.

Each team develops, tests, and deploys their services independently.

Drawbacks of microservices architecture.

Finding the right set of services is challenging.

Distributed systems are complex, which makes development, testing, and deployment difficult.

Note: - SAGAS – data consistency

Deploying features that span multiple services requires careful coordination.

Deciding when to adopt the microservice architecture is difficult.

Pattern is a reusable solution to a problem that occurs in a particular context.

Pattern Structure

Forces

forces (issues) that you must address when solving a problem in a given context.
e.g

Code must easy to understand as well give performance.

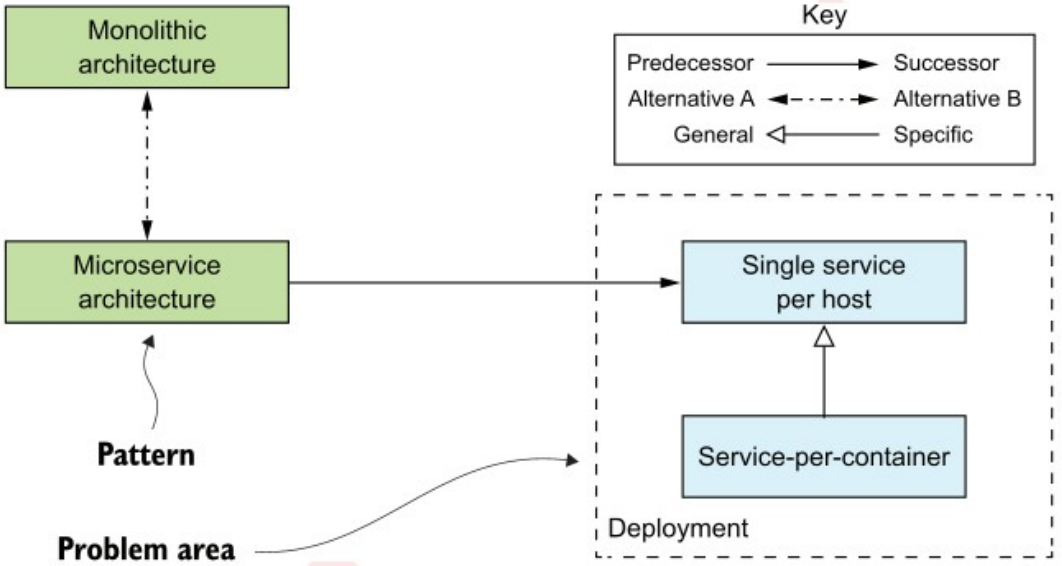
Make a list of forces is nice exercise

Resulting context

Consequences of applying the pattern
a) Benefits - issues resolved
b) Drawback – unresolved by the patten
c) Issues – New problems arising due to application of pattern

Related patterns

Different patterns applied and relation between them of 5 types
a) Predecessor pattern
b) Successor pattern
c) Alternative pattern
d) Generalization pattern
e) Specialization pattern



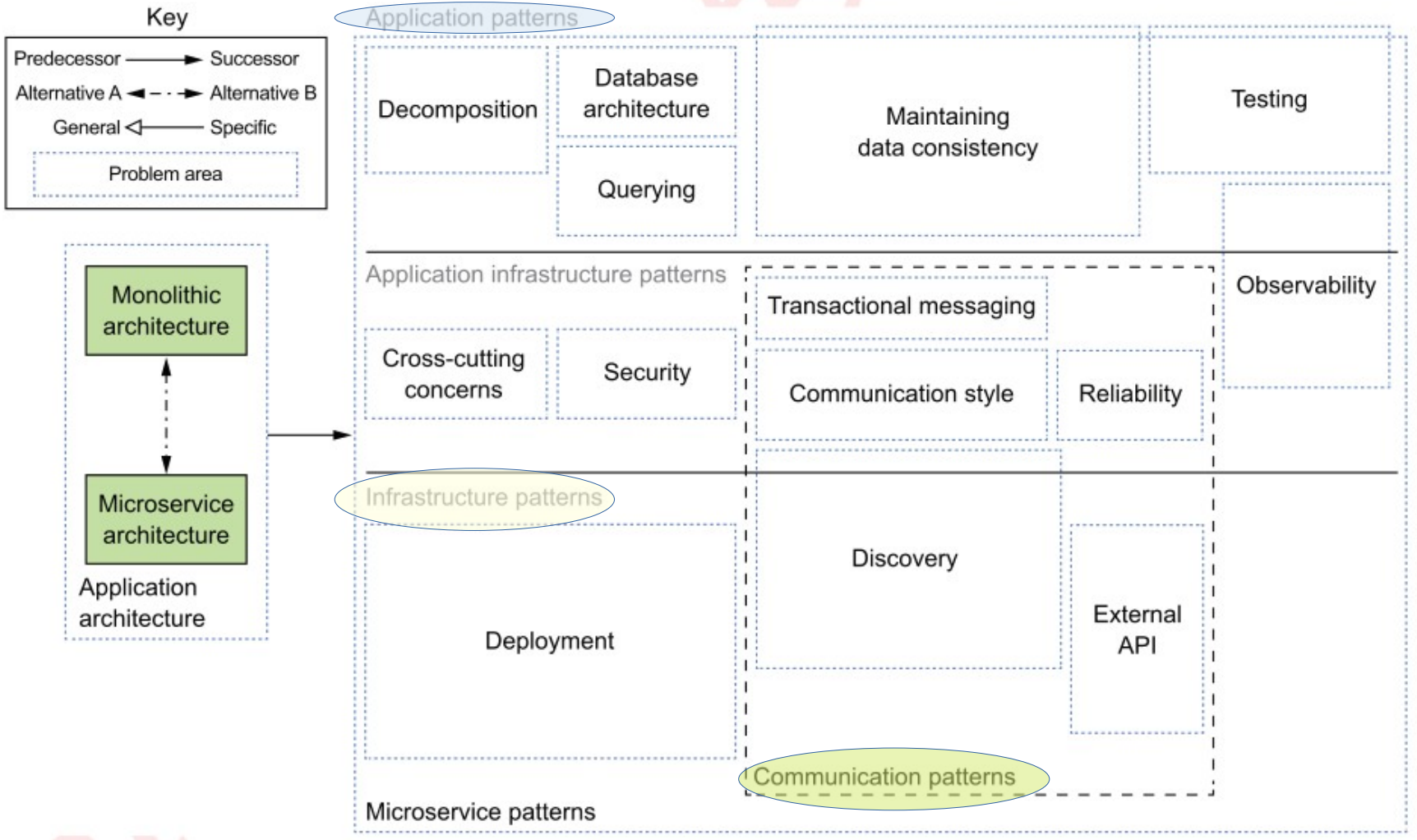
The visual representation of different types of relationships between the patterns:

Successor pattern solves a problem created by applying the *predecessor* pattern;

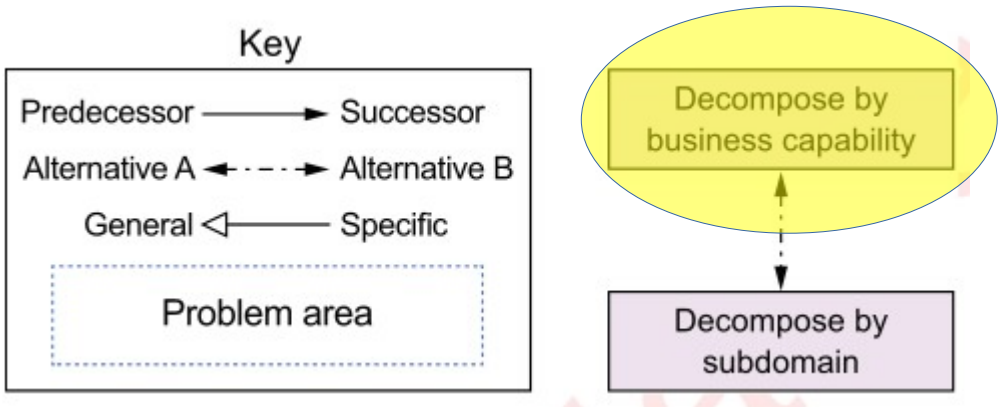
Two or more patterns can be *alternative* solutions to the same problem;

One pattern can be a *specialization* of another pattern;

Patterns that solve problems in the same area can be grouped, or *generalized*.


Key
Predecessor
Successor
Alternative A
Alternative B
General
Specific
Problem area
Application patterns
Decomposition
Database architecture
Querying
Maintaining data consistency
Testing
Application infrastructure patterns
Observability
Transactional messaging
Cross-cutting concerns
Security
Communication style
Reliability
Monolithic architecture
Microservice architecture
Application architecture
Infrastructure patterns
Discovery
Deployment
External API
Communication patterns
Microservice patterns

PATTERNS FOR DECOMPOSING AN APPLICATION INTO SERVICES

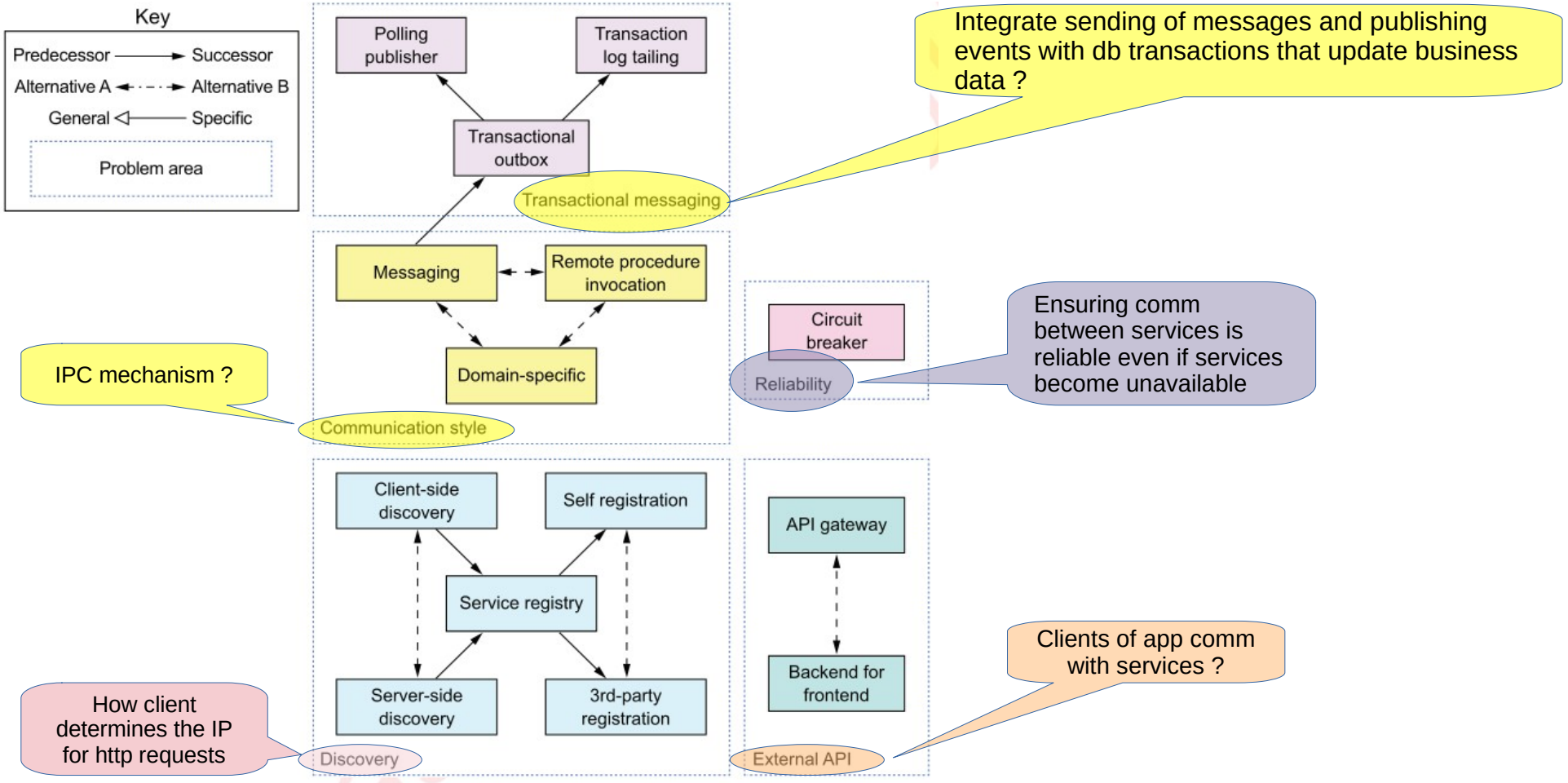


There are two decomposition patterns:

Decompose by business capability, which organizes services around business capabilities, OR

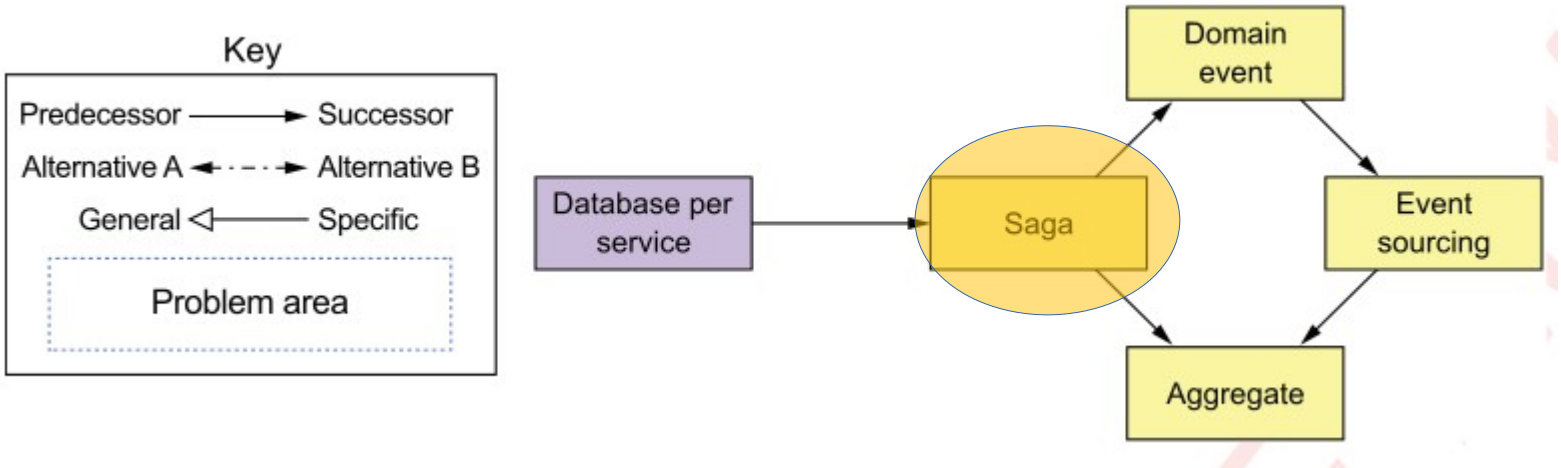
Decompose by subdomain, which organizes services around domain-driven design (DDD) subdomains.

COMMUNICATION PATTERNS



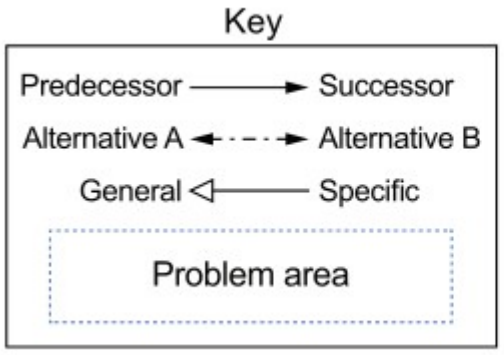
In order to ensure loose coupling, each service has its own database.

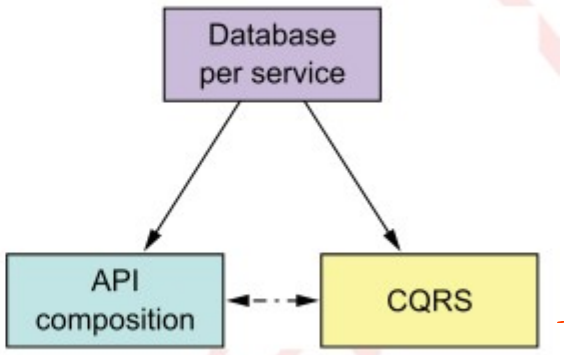
Which brings it’s own challenges



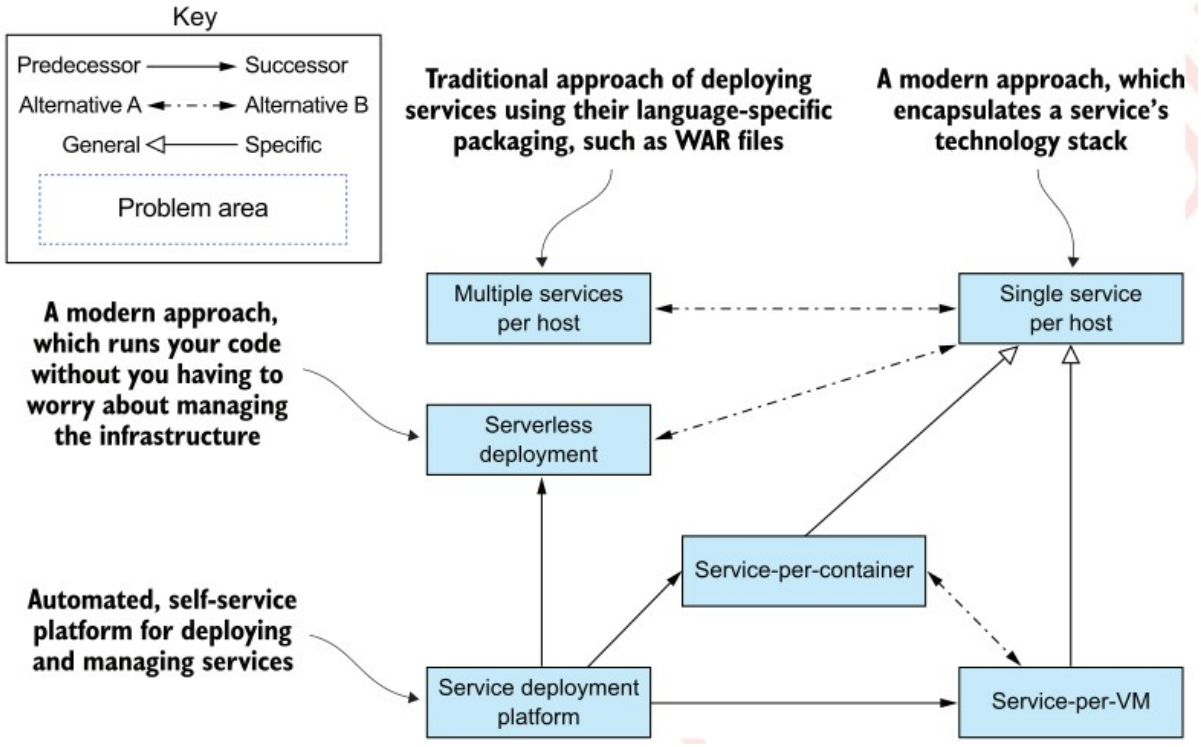
Because each service has its own database, you must use the Saga pattern to maintain data consistency across services.

Because each service has its own database, we must use one
of the querying patterns to retrieve data scattered across multiple services.





Command & Query Responsibility Segregation



Several patterns for deploying microservices.

Traditional approach is to deploy services in a language-specific packaging format.

There are two modern approaches to deploying services.

1) Deploys services as VM or containers.

2) Serverless approach.
We simply upload the service's code and the serverless platform runs it.

We should use a service deployment platform, which is an automated, self-service platform for deploying and managing services.

OBSERVABILITY PATTERNS PROVIDE INSIGHT INTO APPLICATION BEHAVIOR

Key part of operating an application is understanding its runtime behavior and trouble-shooting problems such as

a) failed requests
b) high latency.

In micorservice environment -

A request can bounce around between multiple services before a response is finally returned to a client. Consequently, there isn't one logfile to examine.

Problems with latency are more difficult to diagnose because there are multiple suspects.

For designing observable services -

a) **Health Check API** – Explore and endpoint that returns the health of the services.

b) **Log Aggregation** - Log service activity and write logs into a centralized loggingserver, which provides searching and alerting.

c) **Distributed tracing** - Assign each external request a unique ID and trace requestsas they flow between services.

d) **Exception tracking** - Report exceptions to an exception tracking service, whichdeduplicates exceptions, alerts developers, and tracks the resolution of eachexception.

e) **Application metrics** - Maintain metrics, such as counters and gauges, and exposethem to a metrics server.

f) **Audit logs** – Log user actions.

Patterns for automated testing of services

Patterns for simplifying testing by testing services in isolation:-

Consumer-driven contract test — Verify that a service meets the expectations of its clients.

Consumer-side contract test — Verify that the client of a service can communicatewith the service.

Service component test — Test a service in isolation.

Patterns for handling cross-cutting concerns

When developing the services use microservices chassis pattern.

Building services on top of a framework is the idea

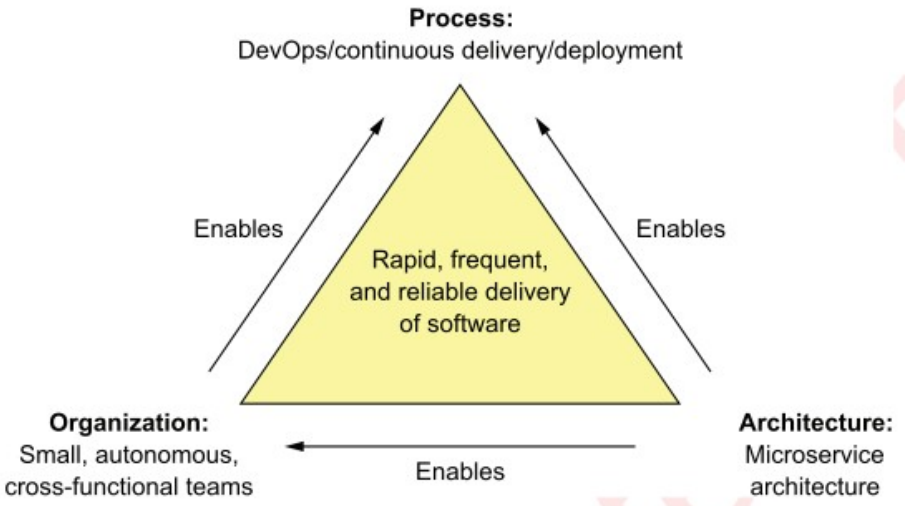
Patterns for security

In microservices users authenticated by the API gateways.

Solution is to apply the Access token pattern.

API gateway passes an access token, such as JWT (JSON Web Token), to the services, which can validate the token and obtain information about the user.

Process and organization



The **rapid, frequent**, and **reliable delivery** of large, complex applications requires a combination of **DevOps**, which includes continuous delivery/deployment,
small, autonomous teams, and the microservice architecture.

Software development and delivery process

What is essential when using microservices architecture ?

a) Agile development
b) Agile deployment

e.g scrum, kanban

c) Practice Continuous delivery/deployment - DevOps

The Monolithic architecture pattern structures the application as a single deployable unit.

The Microservice architecture pattern decomposes a system into a set of inde-pendently deployable services, each with its own database.

The monolithic architecture is a good choice for simple applications, but microservice architecture is usually a better choice for large, complex applications.

The microservice architecture accelerates the velocity of software developmentby enabling small, autonomous teams to work in parallel.

The microservice architecture isn’t a silver bullet—there are significant draw-backs, including complexity.

The Microservice architecture pattern language is a collection of patterns thathelp you architect an application using the microservice architecture. It helps us decide whether to use the microservice architecture, and if we pick the microservice architecture, the pattern language helps us apply it effectively.

We need more than just the microservice architecture to accelerate softwaredelivery. Successful software development also requires DevOps and small,autonomous teams.

Don’t forget about the human side of adopting microservices. We need to consider employees’ emotions in order to successfully transition to a microservice architecture.

Strategies for *Decomposition*

Lets understand

1) Understanding software architecture and why it's important.

2) Decomposing an application into services by applying the decomposition patterns Decompose by business capability and Decompose by subdomain

3) Using the bounded context concept from domain-driven design (DDD) to untangle data and make decomposition easier.

What is software architecture goal ? What is the software architecture ?

Goal of architecture has been **scalability**, **reliability**, and **security**.

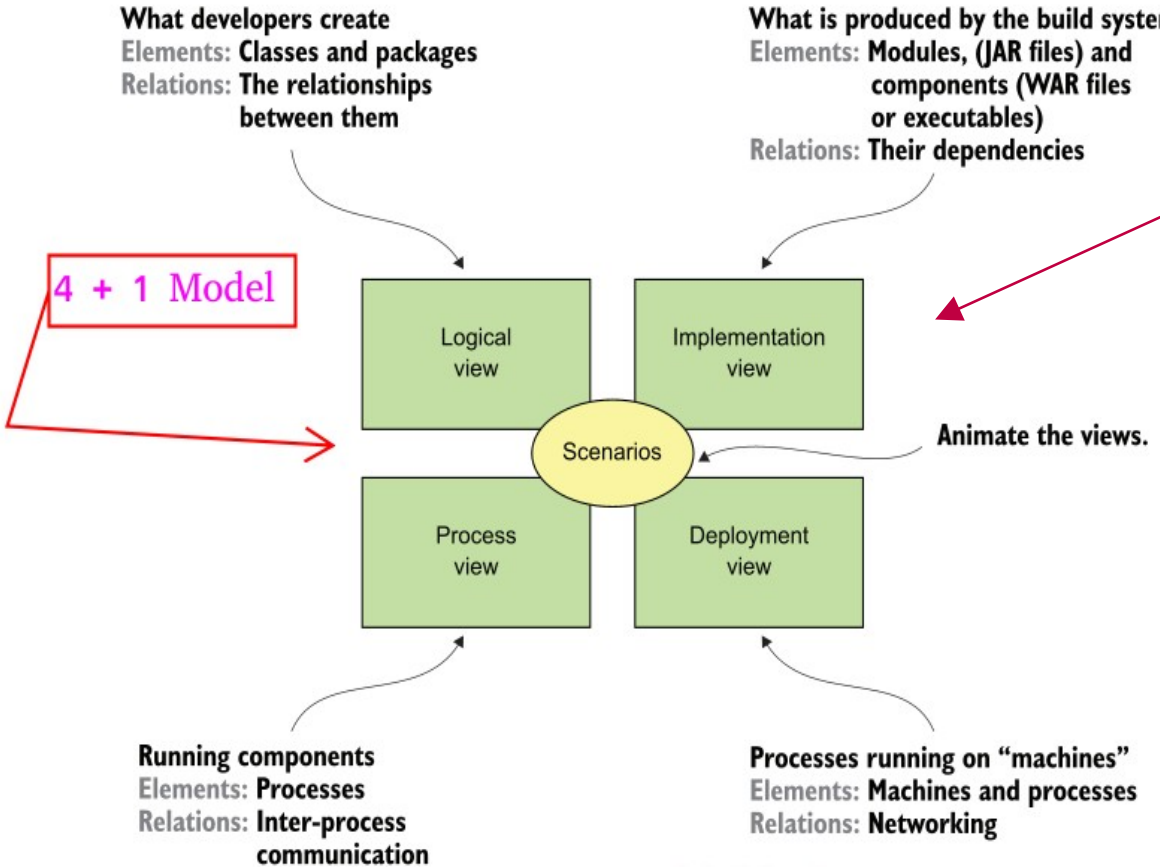
Architecture of a software application is its **high-level structure**, which consists of constituent parts and the dependencies between those parts.

Application structure is always **multidimesional.**

What is microservice architecture precisely ?

Microservice architecture style that gives an application

a) High maintainability,
b) Testability,
c) Deployability



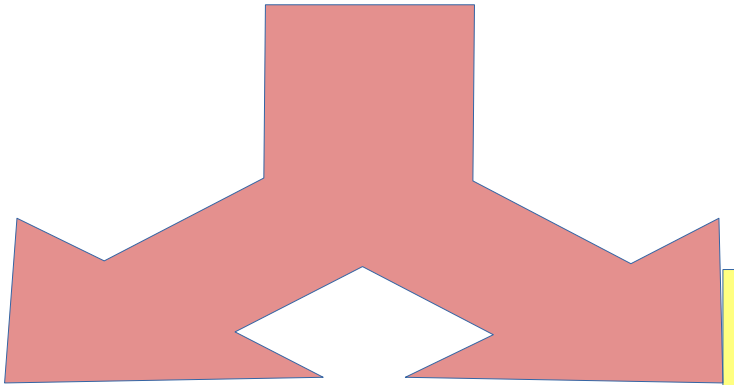
The 4+1 view model describes an application's architecture using four views, along with scenarios that show how the elements within each view collaborate to handle requests.

A scenario in the logical view, shows how the classes collaborate.

A scenario in the process view shows how the processes collaborate.

Why Architecture matters ?

Application requirements



1) Functional requirement

2) Quality of service (QoS)

Quality attributes

Architecture we choose for our application determines how well it meets these quality requirements

Define – runtime qualities -
a) Scalability
b) Reliability
c) Maintainability
d) Testability
e) Deployability

Architectural style

Defines a family of such systems in terms of a *pattern of structural organization*.

An architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.

Layered architecture Style

Layered Architecture organizes software elements into layers.

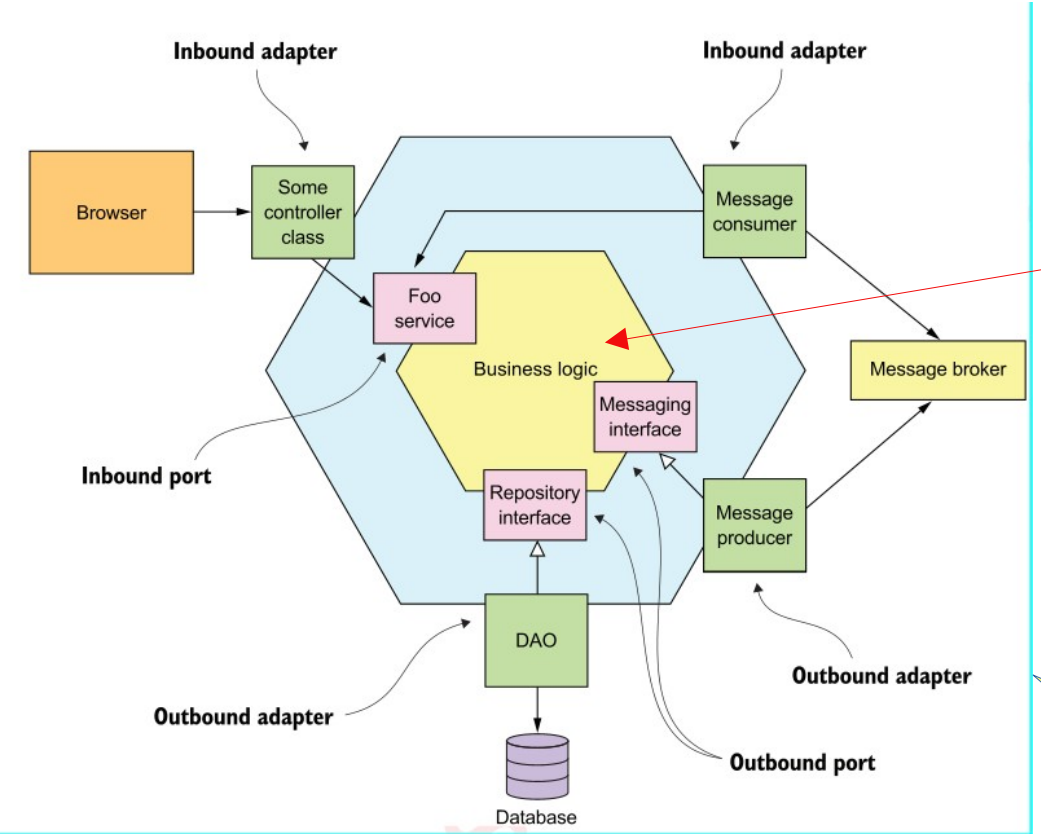
When applied to 4 views, e.g

logical view, *organizes applications classes into following:-*

Presentation layer — Contains code that implements the user interface or external APIs

Business logic layer — Contains the business logic

Persistence layer — Implements the logic of interacting with the database



Hexagonal architecture, which consists of the business logic and one or more adapters that communicate with external systems.

The *business logic* has one or more ports.

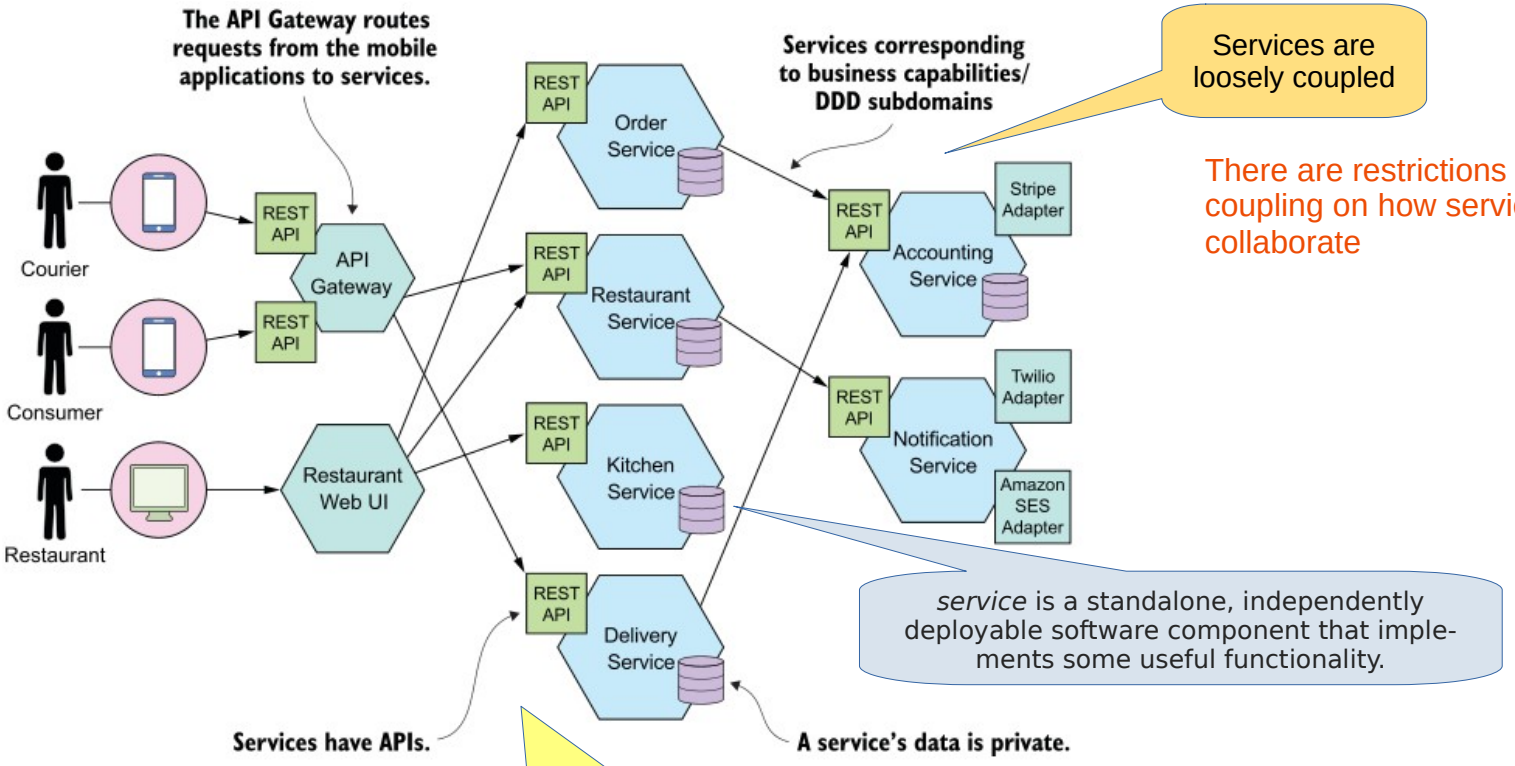
at center

Inbound adapters, which handled requests from external systems, invoke an inbound port.

An outbound adapter implements an outbound port, and invokes an external system.

Alternative to layered architecture style

Microservices Architecture



Services are loosely coupled

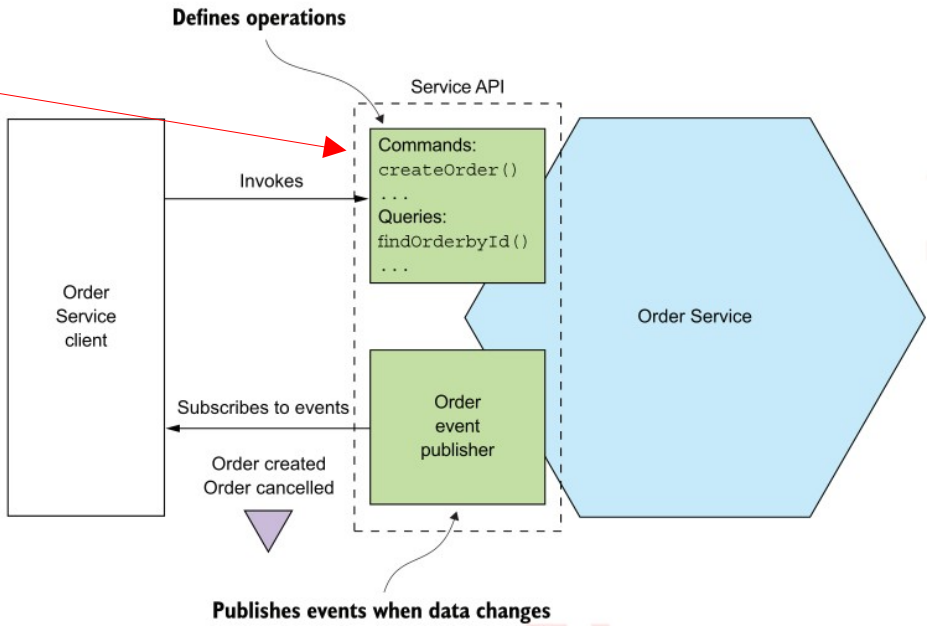
There are restrictions due to loose coupling on how services collaborate

service is a standalone, independently deployable software component that implements some useful functionality.

A possible microservice architecture for the FTGO application. It consists of numerous services.

A service is a standalone, independently deployable software component that implements some useful functionality.

Implements an API – 2 types of operations -
1) Commands
2) Queries



service has an API that encapsulates the implementation.

API defines operations, which are invoked by clients.

There are two types of operations:
commands update data, and **queries** retrieve data.

When its data changes, a service publishes events that clients can subscribe to.

Loose coupling ?

All ***interaction with a service happens via its API***, which encapsulates its implementation details.

This enables the implementation of the service to change without impacting its clients.

Loosely coupled services are key to improving an application’s development time attributes,including its maintainability and testability.

Shared libraries role ?

Developers often package functionality in a library (module) so that it can be reusedby multiple applications without duplicating code.

Good way to reduce code duplication in our services.

But it will introduce tight coupling between our services ? How ?

##

Size of the service is unimportant ?

In reality, size isn't a useful metric.

Better goal is to define a well-designed service to be a service capable ofbeing developed by a small team with minimal lead time and with minimal collaboration with other teams.

Detecting when it’s not architected properly as microservice

if a service requires a large team or takes a long time to test, it probably makes sense to split the team and the service.

Or

if you constantly need to change a service because of changes to other services.

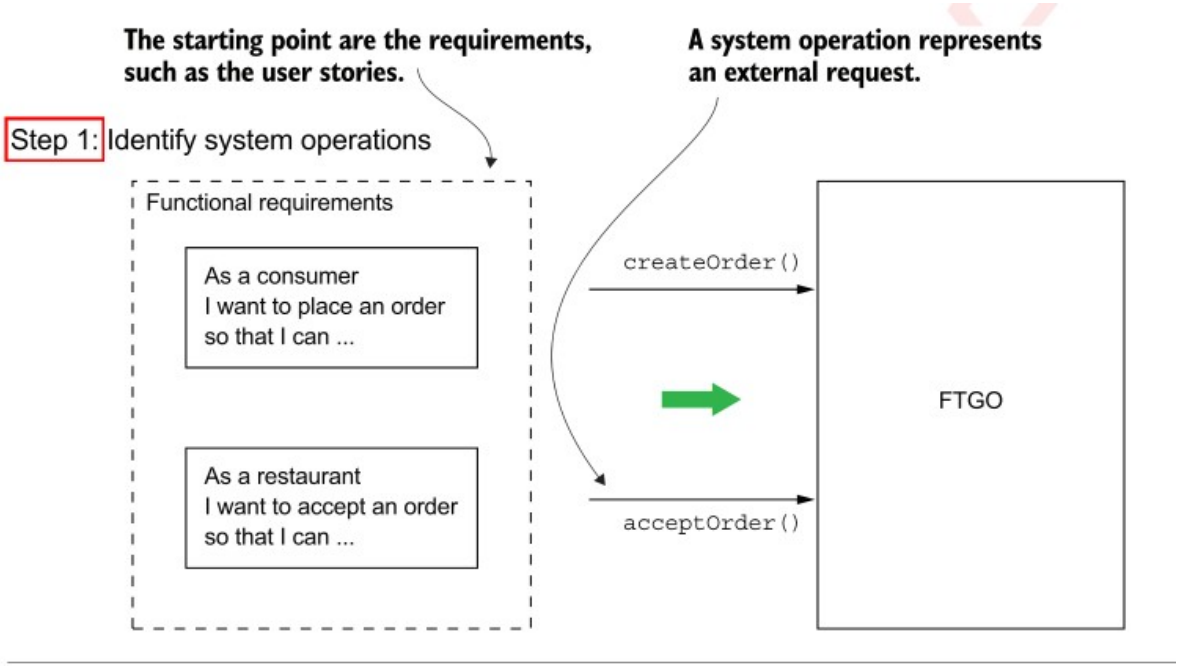
or

if it’striggering changes in other services, that’s a sign that it’s not loosely coupled.

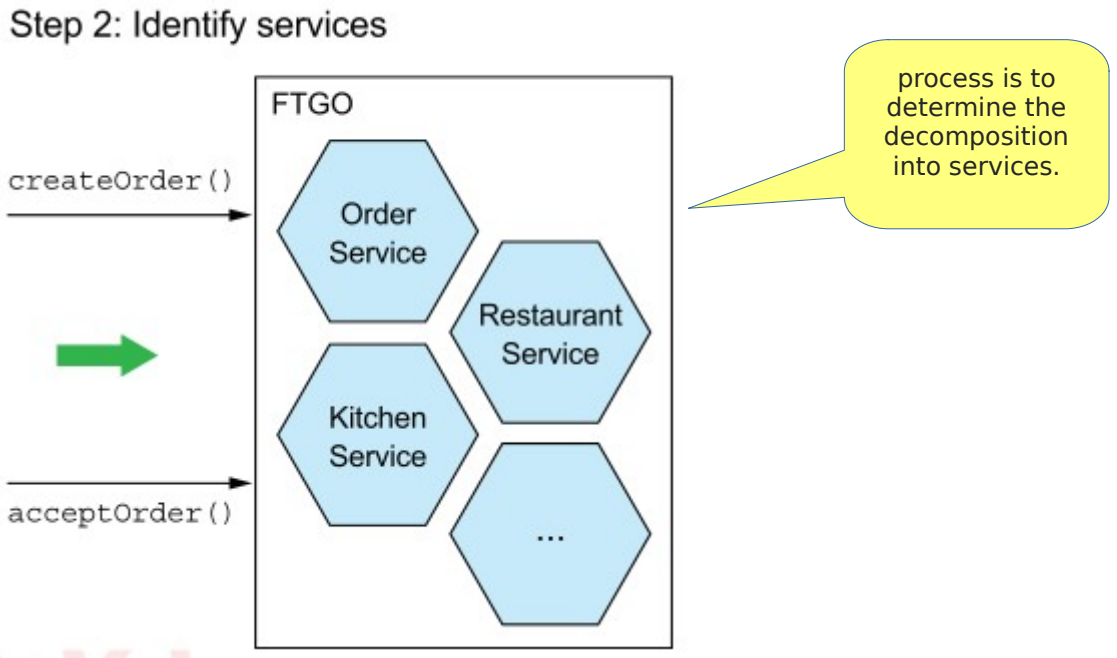
Microservice architecture structures an application as a set of

small,
loosely coupled services.

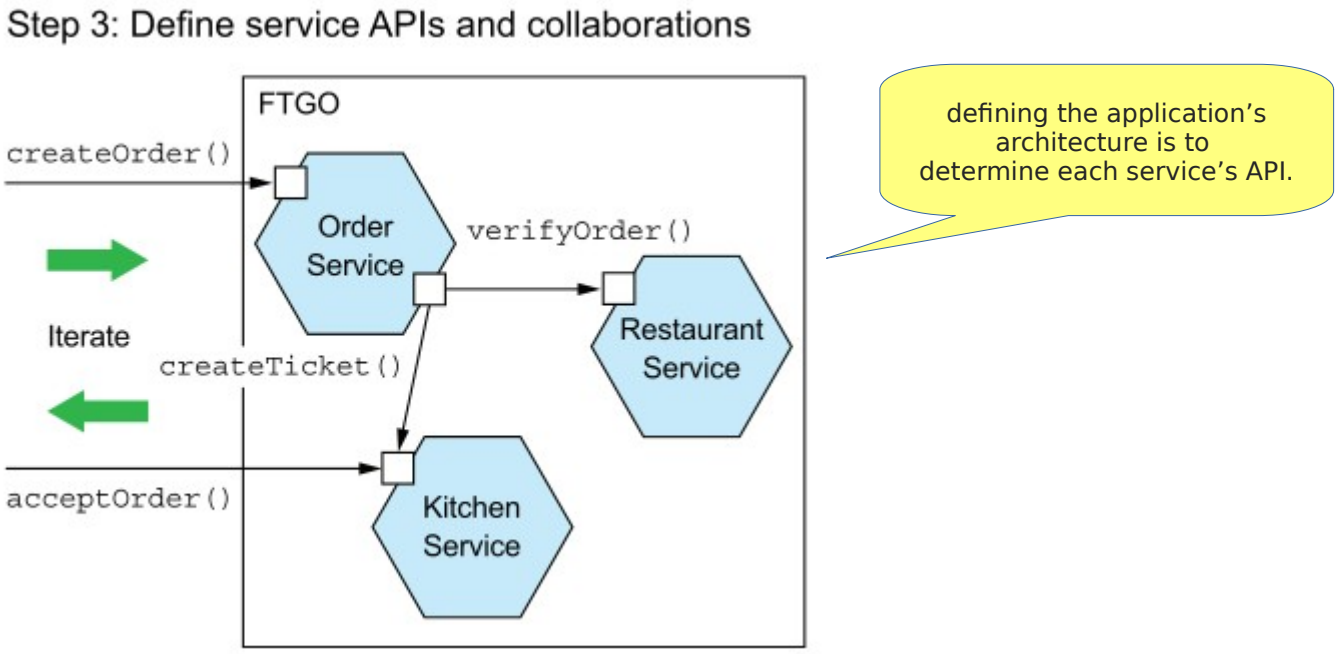
Defining an application’s microservice architecture



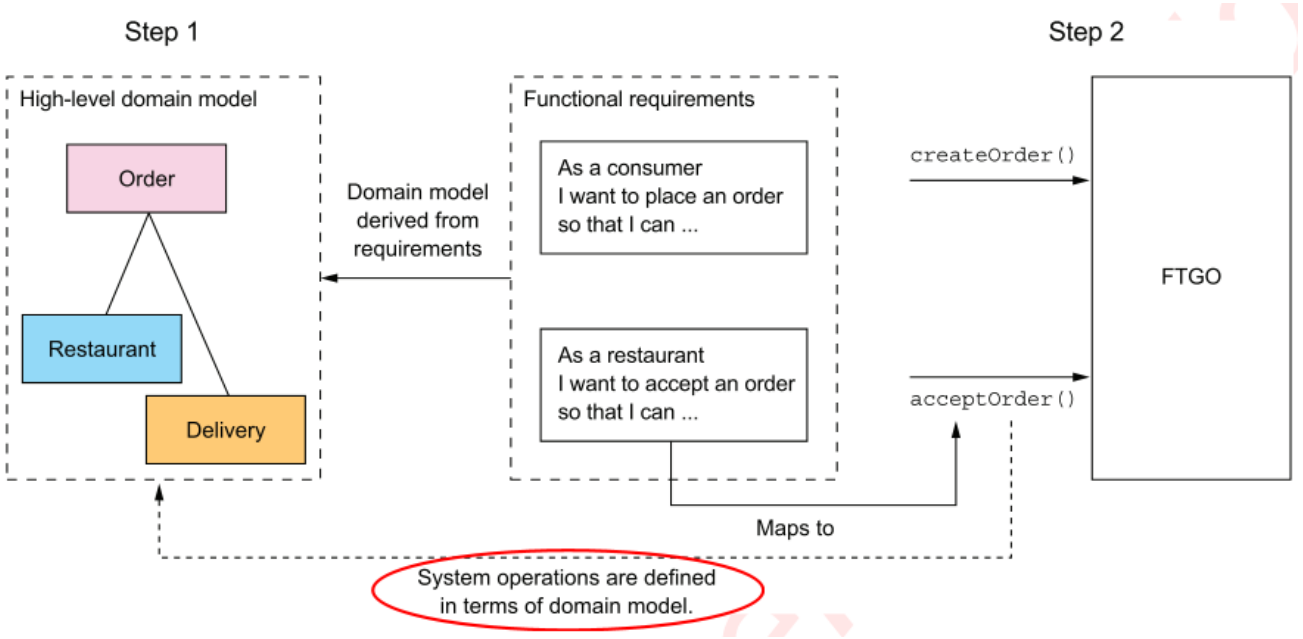
3-step process for defining an application's microservice architecture



3-step process for defining an application's microservice architecture



Identifying the system operations



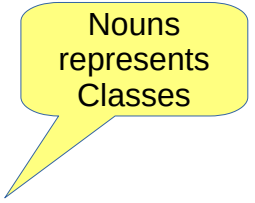
System operations are derived from the application's requirements using a two-step process.

The first step is to create a high-level domain model.

The second step is to define the system operations, which are defined in terms of the domain model.

Creating HIGH-LEVEL domain model

PLACE Order Story

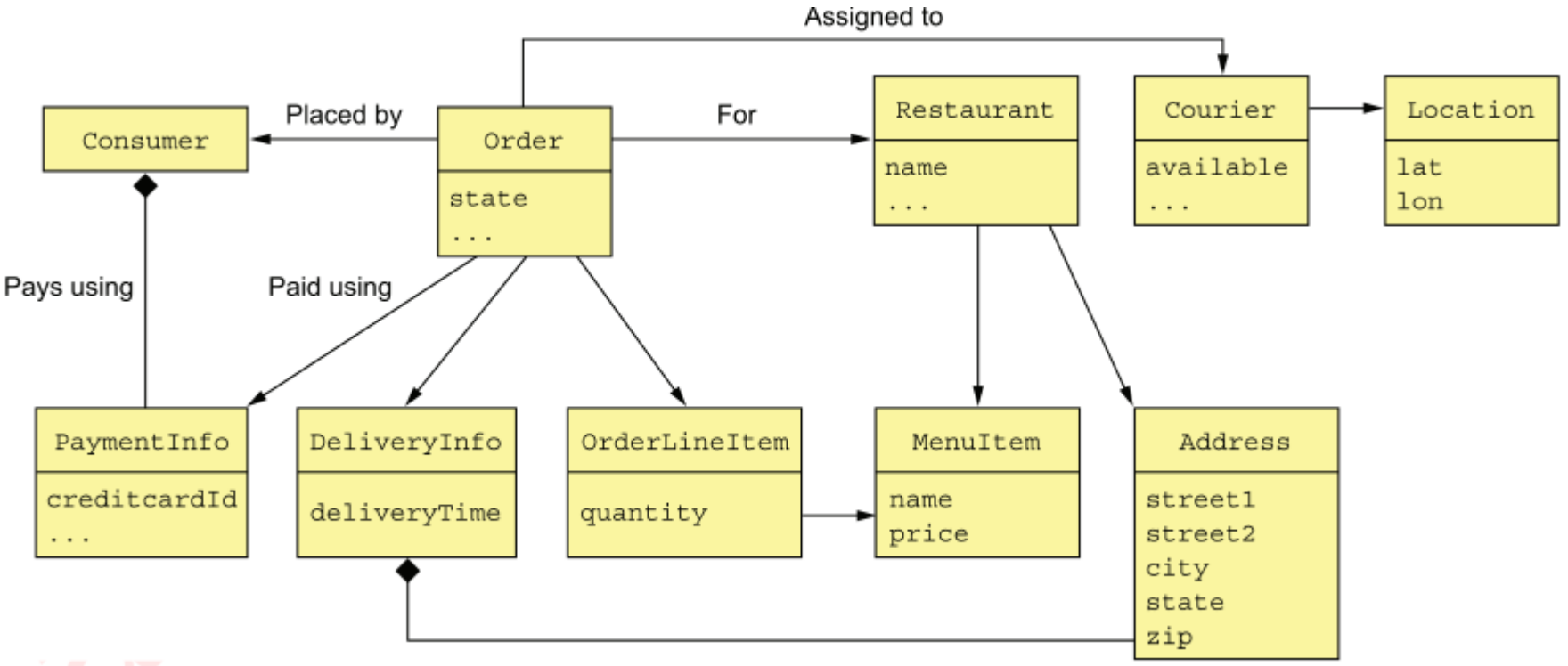



```
Given a consumer
  And a restaurant
  And a delivery address/time that can be served by that restaurant
  And an order total that meets the restaurant's order minimum
When the consumer places an order for the restaurant
Then consumer's credit card is authorized
  And an order is created in the PENDING_ACCEPTANCE state
  And the order is associated with the consumer
  And the order is associated with the restaurant
```

Accept order story

```
Given an order that is in the PENDING_ACCEPTANCE state
  and a courier that is available to deliver the order
When a restaurant accepts an order with a promise to prepare by a particular
     time
Then the state of the order is changed to ACCEPTED
  And the order's promiseByTime is updated to the promised time
  And the courier is assigned to deliver the order
```

With iteration finally landing with - key classes in the apps domain model



Responsibilities of each class

- `Consumer`—A consumer who places orders.
- `Order`—An order placed by a consumer. It describes the order and tracks its status.
- `OrderLineItem`—A line item of an `Order`.
- `DeliveryInfo`—The time and place to deliver an order.
- `Restaurant`—A restaurant that prepares orders for delivery to consumers.
- `MenuItem`—An item on the restaurant's menu.
- `Courier`—A courier who deliver orders to consumers. It tracks the availability of the courier and their current location.
- `Address`—The address of a `Consumer` or a `Restaurant`.
- `Location`—The latitude and longitude of a `Courier`.

Define System Operations

After high-level doman model – we identify requests which application must handle.

Various UI methods can be used – webapp(http), messaging.

Irrespective of the method, we categorize the operastion based

1) Commands
2) Queries

This could be REST or RPC or messaging end points

Identify key System commands

Actor	Story	Command	Description
Consumer	Create Order	`createOrder()`	Creates an order
Restaurant	Accept Order	`acceptOrder()`	Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time
Restaurant	Order Ready for Pickup	`noteOrderReadyForPickup()`	Indicates that the order is ready for pickup
Courier	Update Location	`noteUpdatedLocation()`	Updates the current location of the courier
Courier	Delivery picked up	`noteDeliveryPickedUp()`	Indicates that the courier has picked up the order
Courier	Delivery delivered	`noteDeliveryDelivered()`	Indicates that the courier has delivered the order

createOrder() specs defining it's parameters,return values, behavior in domain model class

Operation	`createOrder` (consumer id, payment method, delivery address, delivery time, restaurant id, order line items)
Returns	`orderId`, ...
Preconditions	▪ The consumer exists and can place orders. ▪ The line items correspond to the restaurant's menu items. ▪ The delivery address and time can be serviced by the restaurant.
Post-conditions	▪ The consumer's credit card was authorized for the order total. ▪ An order was created in the `PENDING_ACCEPTANCE` state.

acceptOrder()

Operation	`acceptOrder(restaurantId, orderId, readyByTime)`
Returns	—
Preconditions	▪ The `order.status` is `PENDING_ACCEPTANCE`. ▪ A courier is available to deliver the order.
Post-conditions	▪ The `order.status` was changed to `ACCEPTED`. ▪ The `order.readyByTime` was changed to the `readyByTime`. ▪ The courier was assigned to deliver the order.

Queries also matter, it would provide a UI with the information a user needs to make decision.

Let's imagine the flow when a consumer places an order:

1) User enters delivery address and time.

2) System displays available restaurants.

3) User selects restaurant.

4) System displays menu.

5) User selects item and checks out.

6) System creates order.

This user scenario suggests the following queries:

a) findAvailableRestaurants(deliveryAddress, deliveryTime)—Retrieves the restaurants that can deliver to the specified delivery address at the specified time

b) findRestaurantMenu(id)—Retrieves information about a restaurant including the menu items

Defining services by applying the Decompose by business capability pattern

business capability is some-thing that a business does in order to generate value.

The set of capabilities for a given business depends on the kind of business.

E.g

Capabilities of an insurance company typically include Underwriting, Claims management, Billing, Compliance,and so on.

Capabilities of an online store include Order management, Inventory management, Shipping, and so on.

Identifying business capability

An organization's business capabilities are identified by analyzing the organization's purpose, structure, and business processes.

Each business capability can be thought of as a service, except it's business-oriented rather than technical.

Its specification consists of various components, including inputs, outputs, and service-level agreements.

E.g

input to an Insurance underwriting capability is the consumer's application, and the outputs include approval and price.

A business capability is often focused on a particular business object.

E.g

Claim business object is the focus of the Claim management capability.

A capability can often be decomposed into sub-capabilities.

E.g

Claim management capability has several sub-capabilities, including
Claim information management,
Claimreview, and
Claim payment management.

Our application example – some of the business capabilities are ?

Supplier management
- *Courier management*—Managing courier information
- *Restaurant information management*—Managing restaurant menus and other information, including location and open hours

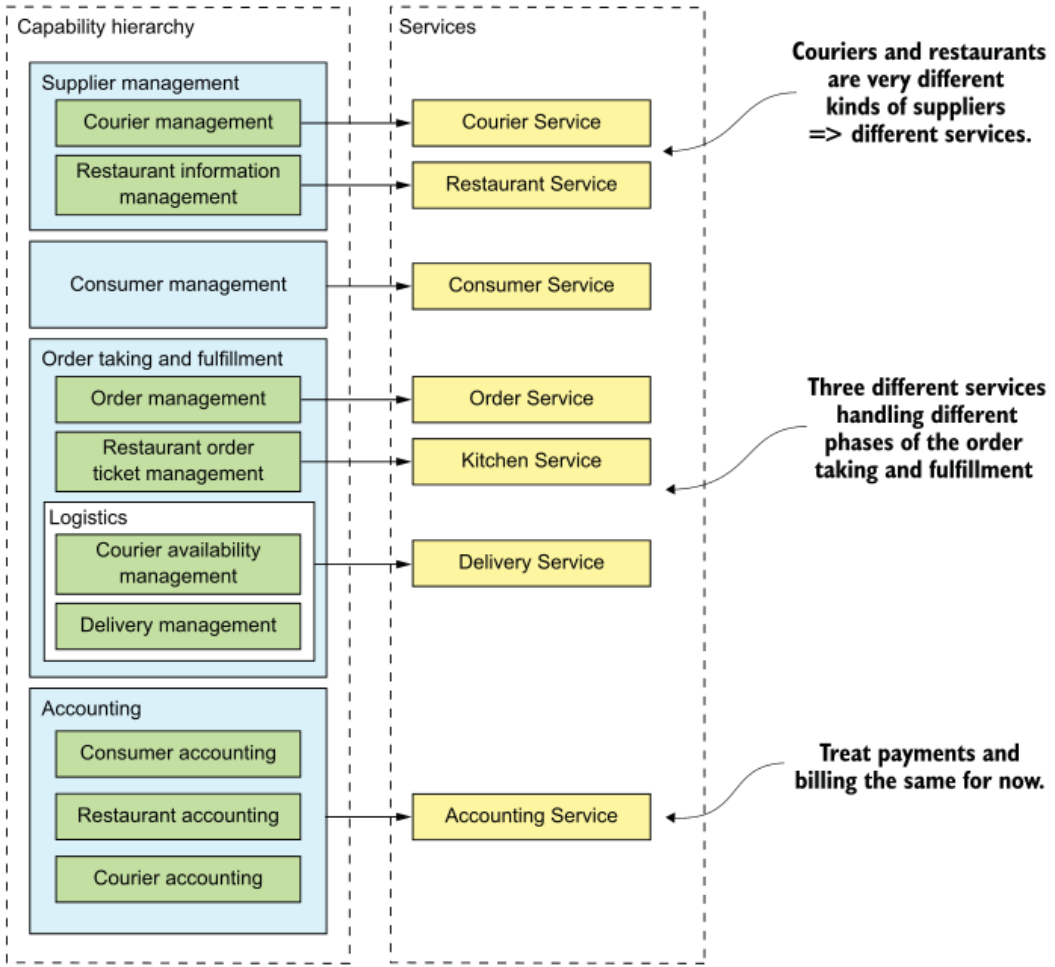
Consumer management—Managing information about consumers

Order taking and fulfillment
- *Order management*—Enabling consumers to create and manage orders
- *Restaurant order management*—Managing the preparation of orders at a restaurant
- Logistics
- *Courier availability management*—Managing the real-time availability of couriers to delivery orders
- *Delivery management*—Delivering orders to consumers

Accounting
- *Consumer accounting*—Managing billing of consumers
- *Restaurant accounting*—Managing payments to restaurants
- *Courier accounting*—Managing payments to couriers

From business capabilities to services

Once we've identified the business capabilities, we then **define a service for each capability** or group of related capabilities.



Mapping capability to service is subjective

A key benefit of organizing services around capabilities is that because they're stable,

Resulting architecture will also be relatively stable.

Individual components of the architecture may evolve as the how aspect of the business changes, but the architecture remains unchanged.

Defining services by applying the Decompose by sub-domain pattern

DDD = Domain driven model

Define services corresponding to DDD subdomains.

Domain mode captures knowledge about a domain in a form that can be used to solve problems within that domain.

DDD - defining multiple domain models, each with an explicit scope.

Terms:

1) Subdomains
2) Bounded contexts – Scope of services

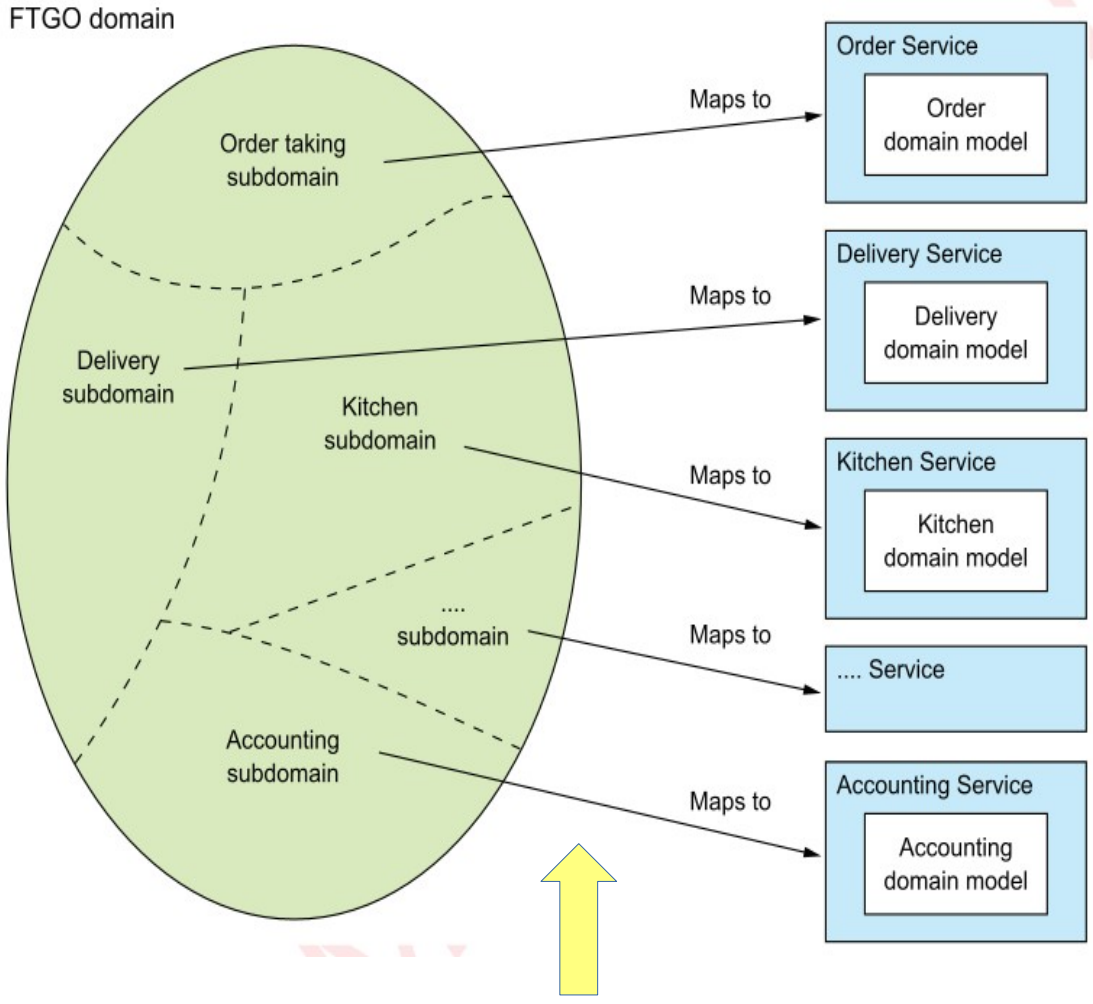
E.g of Subdomains – order taking, order management, kitchen management,Delivery and financial services.

DDD calls the scope of a domain model a *bounded context*.

Bounded context includes the code artifacts that implement the model.

When using the microservice architecture, each bounded context is a service or possibly a set of services.

We can create a microservice architecture by applying DDD and defining a service for each subdomain.



From subdomains to services:
each subdomain of the application domain is mapped to a service, which has its own domain model.

Two main patterns for defining an application's microservice architecture.

1) Decompose by subdomain
2) Decompose by business capability

Decomposition guidelines

SRP – single responsibilty Principle

We define classes that each have a single responsibility and hence a single reason for change.

We can apply SRP when defining a microservice architecture and create small,cohesive services that each have a single responsibility.

This will reduce the size of theservices and increase their stability.

A class should have only one reason to change

CCP – Common Closure Principle

Classes in a package should be closed together against the same kinds of changes.
A change that affects a package affects all the classes in that package.

Obstacles to decomposing an application into services

Network latency
Reduced availability due to synchronous communication
Maintaining data consistency across services
Obtaining a consistent view of the data
God classes preventing decomposition

DDD provides solution

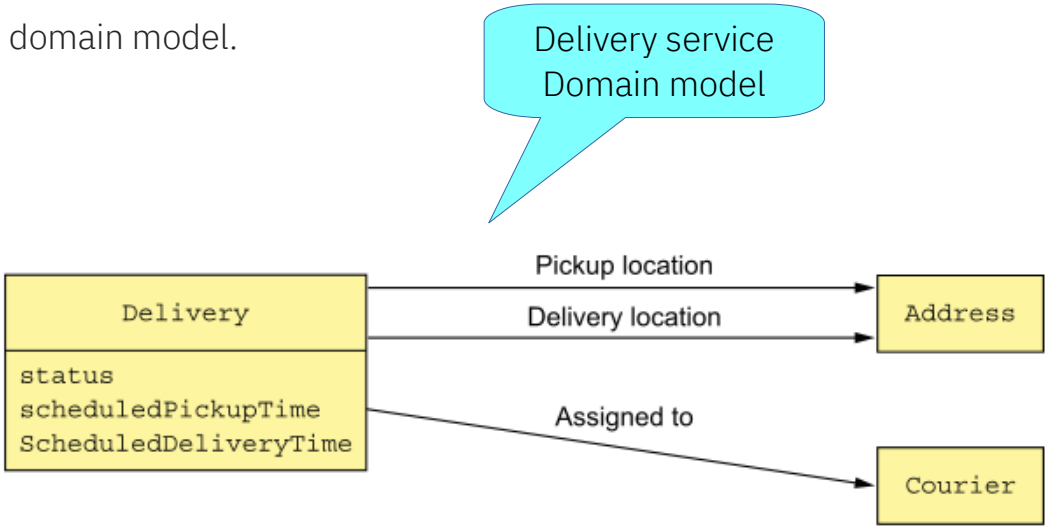
Treat each service as a separate sub-domain with its own domain model.

E.g

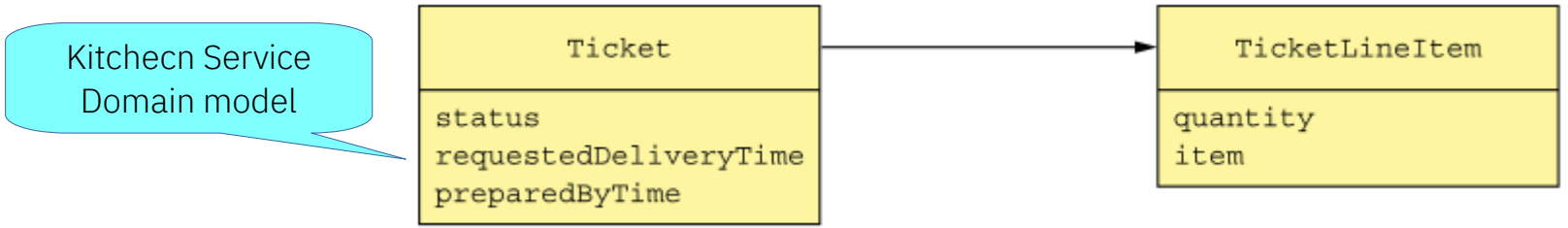
Benefit of multiple domain models is the Delivery Service.

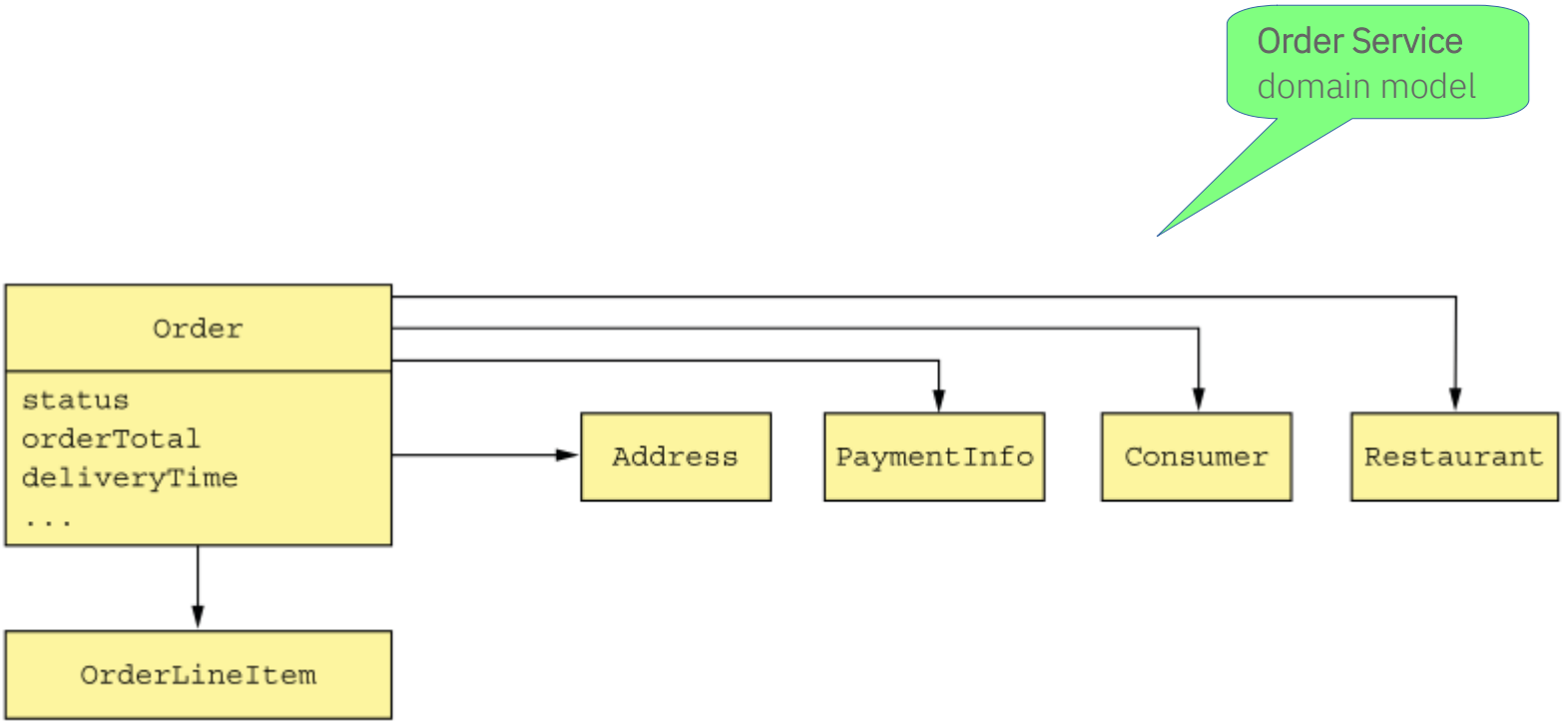
Its view of an Order, is simple:

- pickup address,
- pickup time,
- delivery address, and
- delivery time.



DeliveryService uses the more appropriate name of Delivery rather than calling it order.




Order Service
domain model
Order
status
orderTotal
deliveryTime
...
Address
PaymentInfo
Consumer
Restaurant
OrderLineItem

Defining service APIs

A service API is needed to perform systems operations

Each service will have it's API

API is invoked by external clients and also by other services

API operations are needed for collaboration between services

How to Define service API's

1) Starting point for defining the service APIs is to map each system operation toa service.

2) After that, we decide whether a service needs to collaborate with others to implement a system operation.

3) If collaboration is required, we then determine whatAPIs those other services must provide in order to support the collaboration.

ASSIGNING SYSTEM OPERATIONS TO SERVICES

Service	Operations
Consumer Service	createConsumer()
Order Service	createOrder()
Restaurant Service	findAvailableRestaurants()
Kitchen Service	▪ acceptOrder() ▪ noteOrderReadyForPickup()
Delivery Service	▪ noteUpdatedLocation() ▪ noteDeliveryPickedUp() ▪ noteDeliveryDelivered()

Mapping system operations to services in our application

DETERMINING THE APIS REQUIRED TO SUPPORT COLLABORATION BETWEEN SERVICES

Certain system operations could span across multiple services.

E.g

In order to implement the ***createOrder() operation***, the OrderService must invoke the following services in order to verify its preconditions andmake the post-conditions become true:

- ConsumerService—Verify that the consumer can place an order and obtain their payment information.
- RestaurantService—Validate the order line items, verify that the deliveryaddress/time is within the restaurant's service area, verify order minimum ismet, and obtain prices for the order line items.
- KitchenService—Create the Ticket.
- AccountingService—Authorize the consumer's credit card.

Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	▪ Consumer Service verifyConsumerDetails() ▪ Restaurant Service verifyOrderDetails() ▪ Kitchen Service createTicket() ▪ Accounting Service authorizeCard()
Restaurant Service	▪ findAvailableRestaurants() ▪ verifyOrderDetails()	—
Kitchen Service	▪ createTicket() ▪ acceptOrder() ▪ noteOrderReadyForPickup()	▪ Delivery Service scheduleDelivery()
Delivery Service	▪ scheduleDelivery() ▪ noteUpdatedLocation() ▪ noteDeliveryPickedUp() ▪ noteDeliveryDelivered()	—
Accounting Service	▪ authorizeCard()	—

services, their revised APIs, and their collaborators

Interprocess communication in a microservice architecture

Learning Outcomes

Applying the communication patterns: Remote procedure invocation, Circuit breaker, Client-side discovery, Self registration, Server-side discovery, Third party registration, Asynchronous messaging, Transactional outbox, Transaction log tailing, Polling publisher

The importance of interprocess communication in a microservice architecture

Defining and evolving APIs

The various interprocess communication options and their trade-offs

The benefits of services that communicate using asynchronous messaging

Reliably sending messages as part of a database transaction

What is IPC ?

Interprocess communication (IPC) is used for programs to communicate data to each other and to synchronize their activities. Semaphores, shared memory, and internal message queues are common methods of interprocess communication.

For microservices

We have –

Synchronous req/resp - http based REST or gRPC

Asynchronous – AMQP or STOMP

Interaction between services and it's clients

1st Dimension

a) *One-to-one*—Each client request is processed by exactly one service.
b) One-to-many—Each request is processed by multiple services.

2nd Dimension

a) *Synchronous*—The client expects a timely response from the service and might even block while it waits.
b) synchronous—The client doesn't block, and the response, if any, isn't necessar-ily sent immediately.

	one-to-one	one-to-many
Synchronous	Request/response	—
Asynchronous	Asynchronous request/response One-way notifications	Publish/subscribe Publish/async responses

One to one interactions

a) Request/response—A service client makes a request to a service and waits for a response.

The client expects the response to arrive in a timely fashion.

It might **event block** while waiting. *This is an interaction style that generally results inservices being tightly coupled*.

b) Asynchronous request/response—A service client sends a request to a service, which replies **asynchronously**.

The client **doesn't block** while waiting, because the service might not send the response for a long time.

c) One-way notifications—A service client sends a request to a service, but no reply is expected or sent.

One-to-Many interactions

Publish/subscribe—A client publishes a notification message, which is consumed by zero or more interested services.

Publish/async responses—A client publishes a request message and then waits for a certain amount of time for responses from interested services.

Defining APIs in a microservice architecture

APIs or interfaces are central to software development.

An application is comprised of modules.

Each module has an ***interface that defines the set of operations*** that module's clients can invoke.

A well designed interface **exposes useful functionality** while hiding the implementation.

It enables the implementation to change without impacting clients.

Overall – API-first design is essential

Evolving APIs

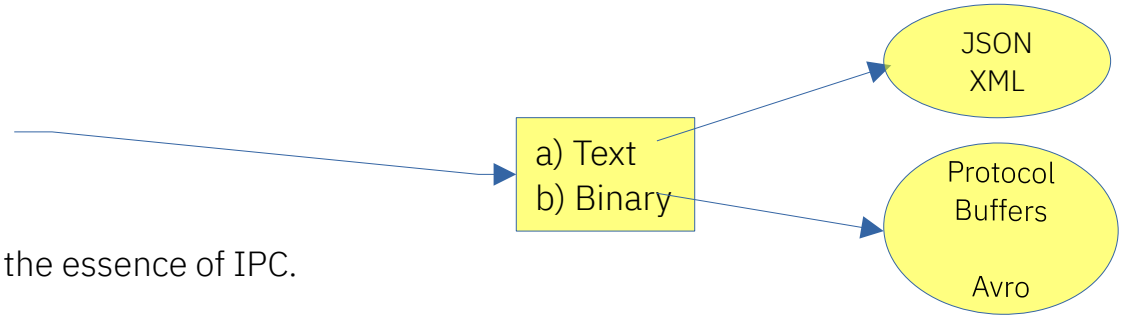
In microservices updating API's is a challenge as they developed by different teams.

So proper DevOps tools need to used to streamline the process of update with strategies like rolling updates.

Use semantic versioning - major, minor, patch for API's

Remember- backward compatibility is to be maintained uptil certain older versions

Message formats



Messages form the essence of IPC.

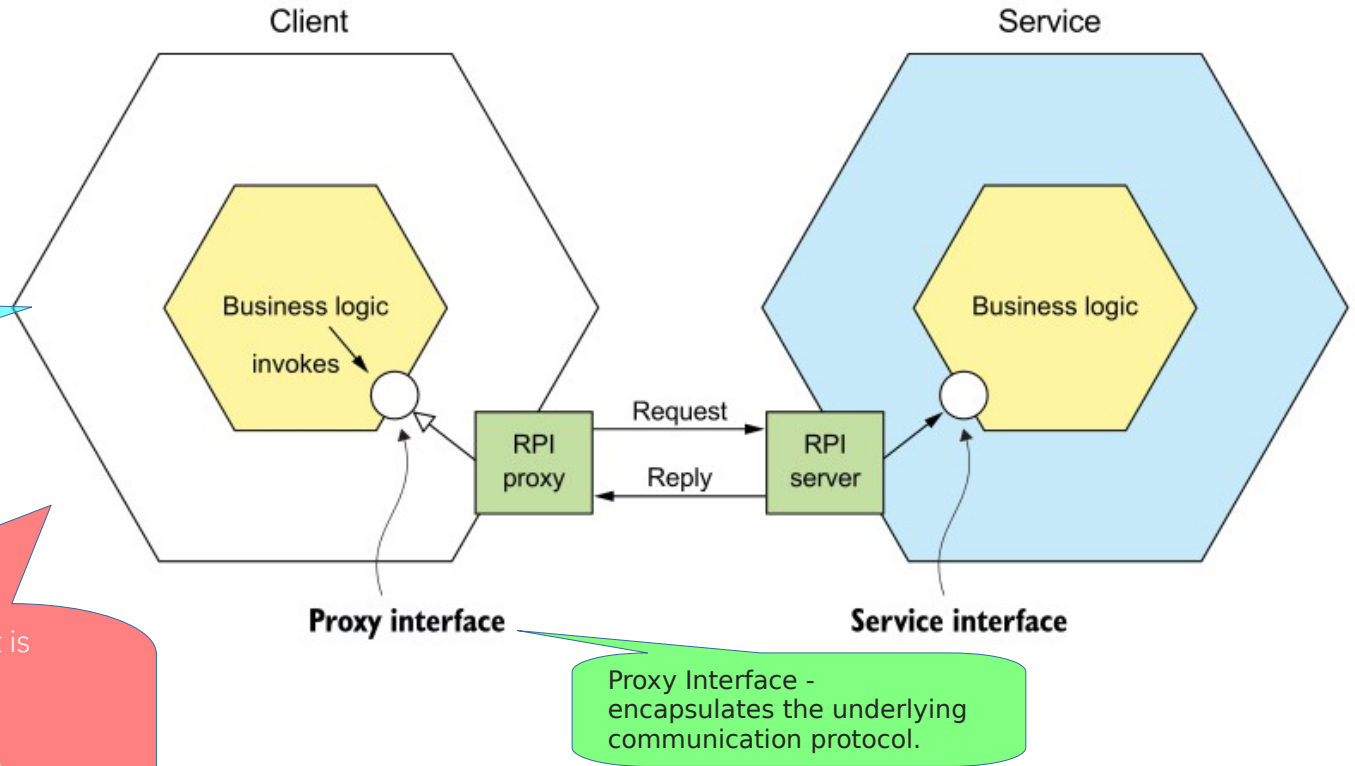
Messages = data

Communicating using the synchronous Remote procedure invocation pattern

IPC - GRPC

Communicating using the synchronous Remote procedure invocation pattern

Like REST



The client's business logic invokes an interface that is implemented by an *RPI proxy*adapter class.

The *RPI proxy class* makes a request to the service.

The *RPI server* adapter class handles the request by invoking the service's business logic.

Using REST

REST provides a set of architectural constraints that, when applied as a whole, emphasizes scalability of component interactions, generality of interfaces, independent deployment of components, and intermediary components to reduce interaction latency, enforce security, and encapsulate legacy systems.

Roy Fielding

REST is an IPC mechanism that (almost always) uses HTTP.

Resource is the key concept in REST

Resource = single business object, customer, product,
collection of business objects.

REST MATURITY MODEL

Level 0—Clients of a level 0 service invoke the service by making HTTP POST requests to its sole URL endpoint.

Each request specifies the action to perform,the target of the action (eg, the business object), and any parameters.

Level 1—A level 1 service supports the idea of resources.

To perform an actionon a resource, a client makes a POST request that specifies the action to perform and any parameters.

Level 2—A level 2 service uses HTTP verbs to perform actions:

GET to retrieve, POST to create, and PUT to update.

The request query parameters and body, if any, specify the actions' parameters.

This enables services to use web infrastructure such as caching for GET requests.

REST MATURITY MODEL

Level 3—The design of a level 3 service is based on the terribly named HATEOAS (Hypertext As The Engine Of Application State) principle.

The basic idea is that the representation of a resource returned by a GET request contains links for performing actions on that resource.

E.g

A client can cancel an order using a link in the representation returned by the GETrequest that retrieved the order.

The benefits of HATEOAS include ***no longer having to hard-wire URLs into client code*** (www.infoq.com/news/2009/04/hateoas-restful-api-advantages).

CHALLENGE OF FETCHING MULTIPLE RESOURCES IN A SINGLE REQUEST

Refer

a) **graphQL** - https://graphql.org/

b) **Falcor** - https://netflix.github.io/falcor/

Benefits and drawbacks of REST

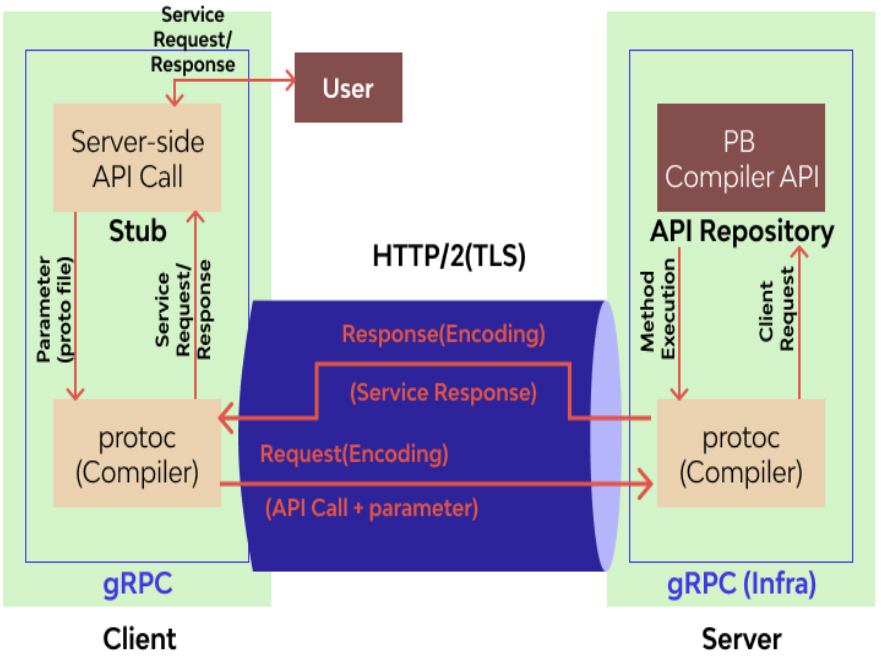
There are numerous benefits to using REST:

- It's simple and familiar.
- You can test an HTTP API from within a browser using, for example, the Postman plugin, or from the command line using curl (assuming JSON or some other text format is used).
- It directly supports request/response style communication.
- HTTP is, of course, firewall friendly.
- It doesn't require an intermediate broker, which simplifies the system's architecture.

There are some drawbacks to using REST:

- It only supports the request/response style of communication.
- Reduced availability. Because the client and service communicate directly without an intermediary to buffer messages, they must both be running for the duration of the exchange.
- Clients must know the locations (URLs) of the service instances(s). As described in section 3.2.4, this is a nontrivial problem in a modern application. Clients must use what is known as a *service discovery mechanism* to locate service instances.
- Fetching multiple resources in a single request is challenging.
- It's sometimes difficult to map multiple update operations to HTTP verbs.

Using gRPC



An IPC technologythat avoids this issue is gRPC (www.grpc.io), a framework for writing cross-languageclients and servers.

gRPC is a binary message-based protocol, we're forced to take an API-first approach to service design.

We define your gRPC APIs using a Protocol Buffers-based IDL, which is Google's language-neutral mechanism for serializing structured data.

We use the Protocol Buffer compiler to generate client-side stubs and server-side skeletons.

Compiler can generate code for a variety of languages, including Java, C#,NodeJS, and GoLang.

Clients and servers exchange binary messages in the ProtocolBuffers format using HTTP/2.

Refer https://en.wikipedia.org/wiki/Remote_procedure_call formore)

gRPC has several benefits:

- It's straightforward to design an API that has a rich set of update operations.
- It has an efficient, compact IPC mechanism, especially when exchanging large messages.
- Bidirectional streaming enables both RPI and messaging styles of communication.
- It enables interoperability between clients and services written in a wide range of languages.

gRPC also has several drawbacks:

- It takes more work for JavaScript clients to consume gRPC-based API than REST/JSON-based APIs.
- Older firewalls might not support HTTP/2.

Circuit breakers

Situation ?

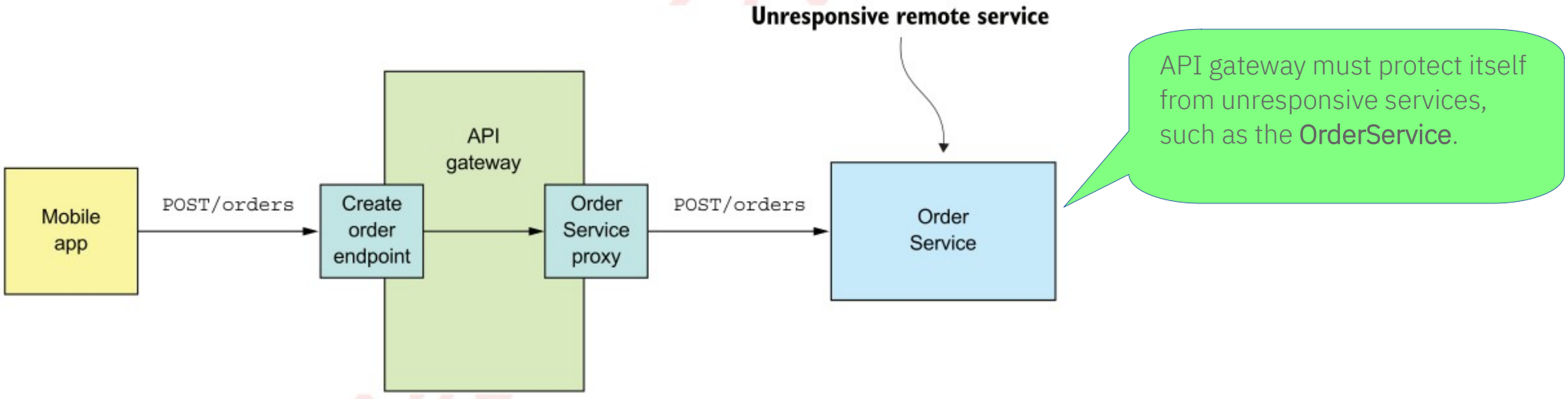
In a distributed system, whenever a service makes a synchronous request to another service, there is an ever present risk of partial failure. Why ?

Because the client and the service are separate processes, a service may not be able to respond in a timely way to aclient's request.

The service could be down because of a failure or for maintenance. Or the service might be overloaded and responding extremely slowly to requests.

Because the client is blocked waiting for a response, the danger is that the failure could cascade to the client's clients and so on and cause an outage.

An RPI proxy that immediately rejects invocations for a timeout period after the number of consecutive failures exceeds a specified threshold.



Robust RPI proxies

Whenever one service synchronously invokes another service, it should protect itself From

a) Network timeouts -Never block indefinitely and always use timeouts when waiting for a response.

Using timeouts ensures that resources are never tied upindefinitely.

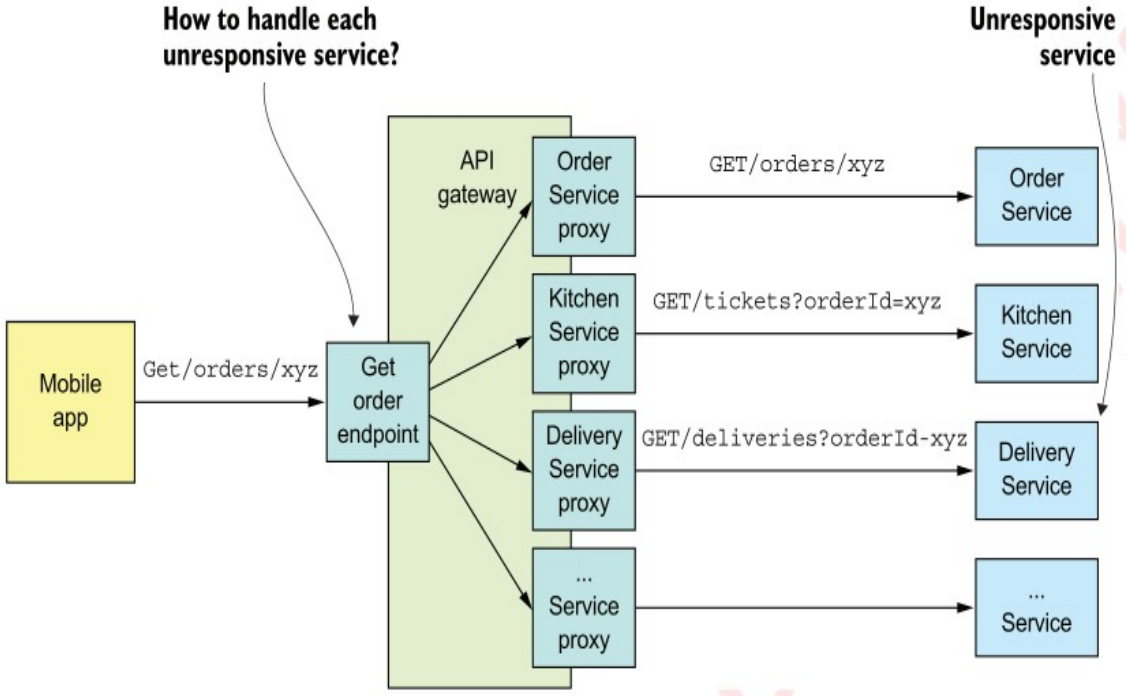
b) Limit the number of outstanding requests from a client to a service - Impose an upperbound on the number of outstanding requests that a client can make to a particular service.

If the limit has been reached, it's probably pointless to makeadditional requests, and those attempts should fail immediately.

c) Circuit breaker pattern - Track the number of successful and failed requests,and if the error rate exceeds some threshold, trip the circuit breaker so thatfurther attempts fail immediately.

A large number of requests failing suggeststhat the service is unavailable and that sending more requests is pointless.After a timeout period, the client should try again, and, if successful, close thecircuit breaker.

Refer - http://techblog.netflix.com/2012/02/fault-tolerance-in-high-volume.html



API gateway implements the **GET/orders/{orderId}** endpoint using API composition.

It calls several services, aggregates their responses, and sends a response to the mobile app.

The code that implements the endpoint must have a strategy for handling the failure of each service that it calls.

Types of failures to be handled

Partial failures

Unresponsive remote services

One service requests to another services using RPI needs to know network location of services

Service Discoveries

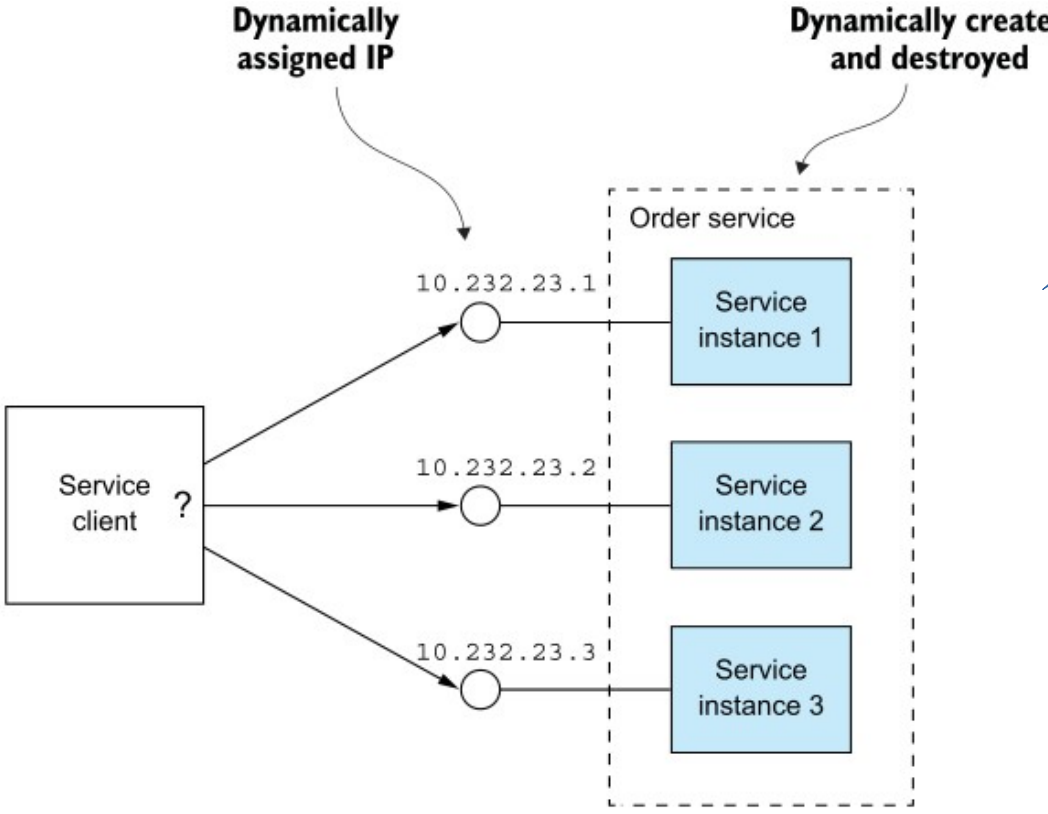
Service discovery is how applications and (micro)services locate each other on a network.

Implementations include both a central server(s) that maintain a global view of addresses and clients that connect to the central server to update and retrieve addresses.

Service instances have dynamically assigned network locations.

Set of service instances changes dynamically because of autoscaling, failures, and upgrades.

Consequently, *our client code must use a service discovery.*



Service instances have dynamically assigned IP addresses.

How does service discovery work

Dynamic service discovery mechanism.

Service discovery :

its key component is a **service registry**, which is a database of the network locations of an application's service instances.

The service discovery mechanism updates the service registry when service instances start and stop.

When a client invokes a service, the service discovery mechanism queries the service registry to obtain a list of available service instances and routes therequest to one of them.

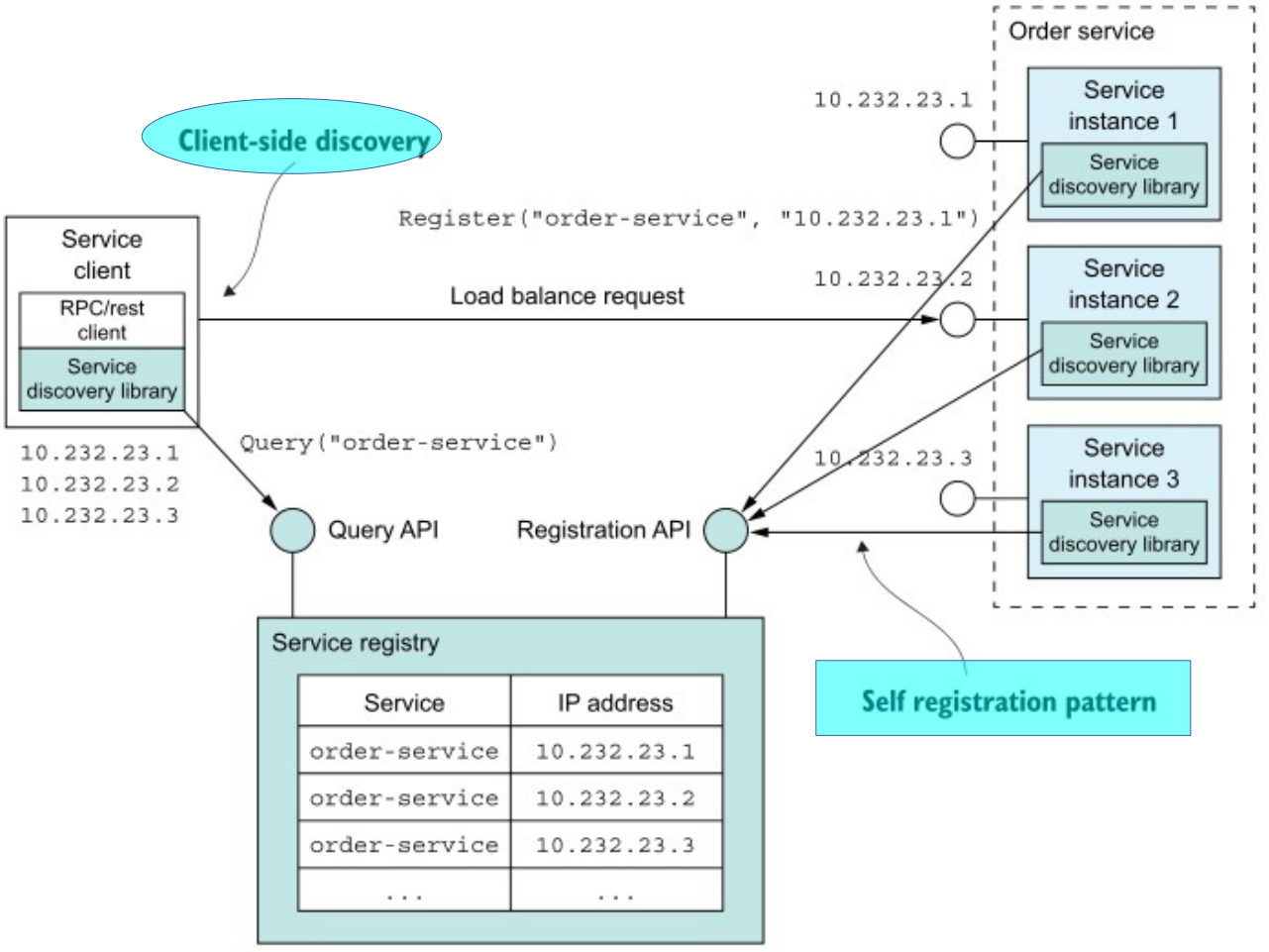
There are two main ways to implement service discovery:

a) The services and their clients interact directly with the service registry.

b) The deployment infrastructure handles service discovery.

service registry keeps track of the service instances.

Clients query the service registry to find network locations of available service instances.



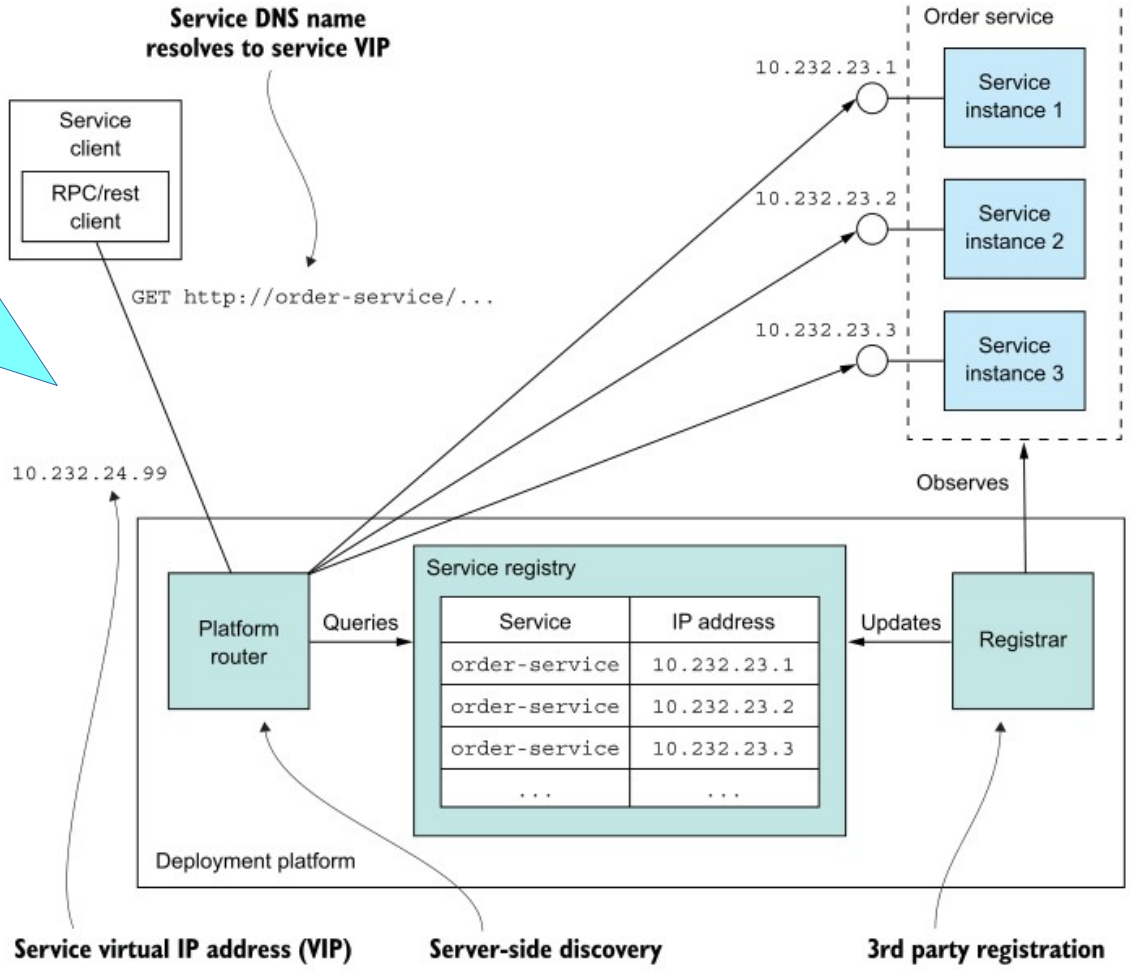
Platform is responsible for service registration, discovery, and request routing.

Service instances are registered with the service registry by the *registrar*.

Each service has a network location, a DNS name/virtual IP address.

A client makes a request to the service's network location.

The router queries the service registry and load balances requests across the available service instances.



Communicating using the Asynchronous messaging pattern

When we use messaging -

Services communicate by asynchronously exchanging messages.

A messaging based application uses a ***message broker***, which acts as an intermediary between the services.

Alternatively we can use a **brokerless architecture**, where the services communicate directly with each other.

A service client makes a request to a service by sending it a message.

If the service instance is expected to reply, it will do so by sending a separate message back to the client.

Nonblocking
Because the communication is asynchronous, the client doesn't block waiting for a reply.
Instead,the client is written assuming that the reply won't be received immediately.

What’s in message ?

Document —A generic message that contains only data.

The receiver decides howto interpret it. The reply to a command is an example of a document message.

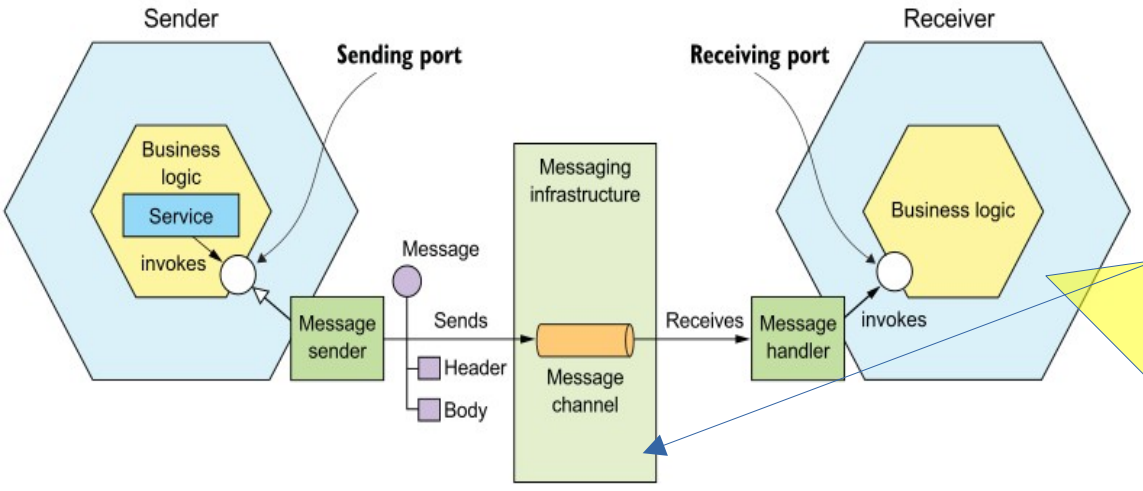
Command —A message that’s the equivalent of an RPC request.

It specifies theoperation to invoke and its parameters.

Event —A message indicating that something notable has occurred in the sender.

An event is often a domain event, which represents a state change of a domainobject such as an Order, or a Customer.

Message Channels



The business logic in the sender invokes a sending port interface, which is implemented by a *message sender adapter*.

The message sender sends a message to a receiver via a message channel.

The *message channel is an abstraction of messaging infrastructure.*

A *message handler adapter in the receiver is invoked to handle the message*. It invokes the receiving port interface implemented by the receiver's business logic.

Message channel types

1) point-to-point channel delivers a message to exactly one of the consumers that is reading from the channel.

Services use **point-to-point channels** for the one-to-one interaction styles.

E.g

A command message is often sent over a point-to-point channel.

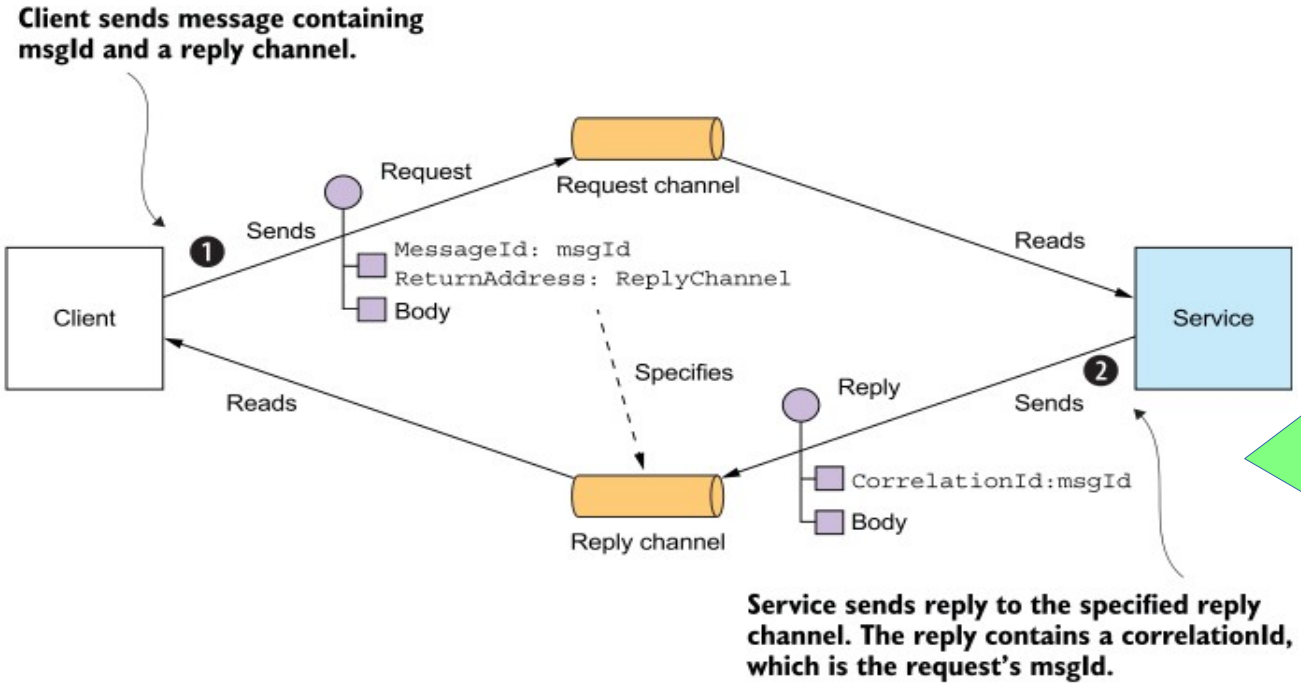
2) *publish-subscribe* channel delivers each message to all of the attached consumers.

Services use publish-subscribe channels for the **one-to-many interaction** styles.

E.g

An event message is usually sent over a publish-subscribe channel.

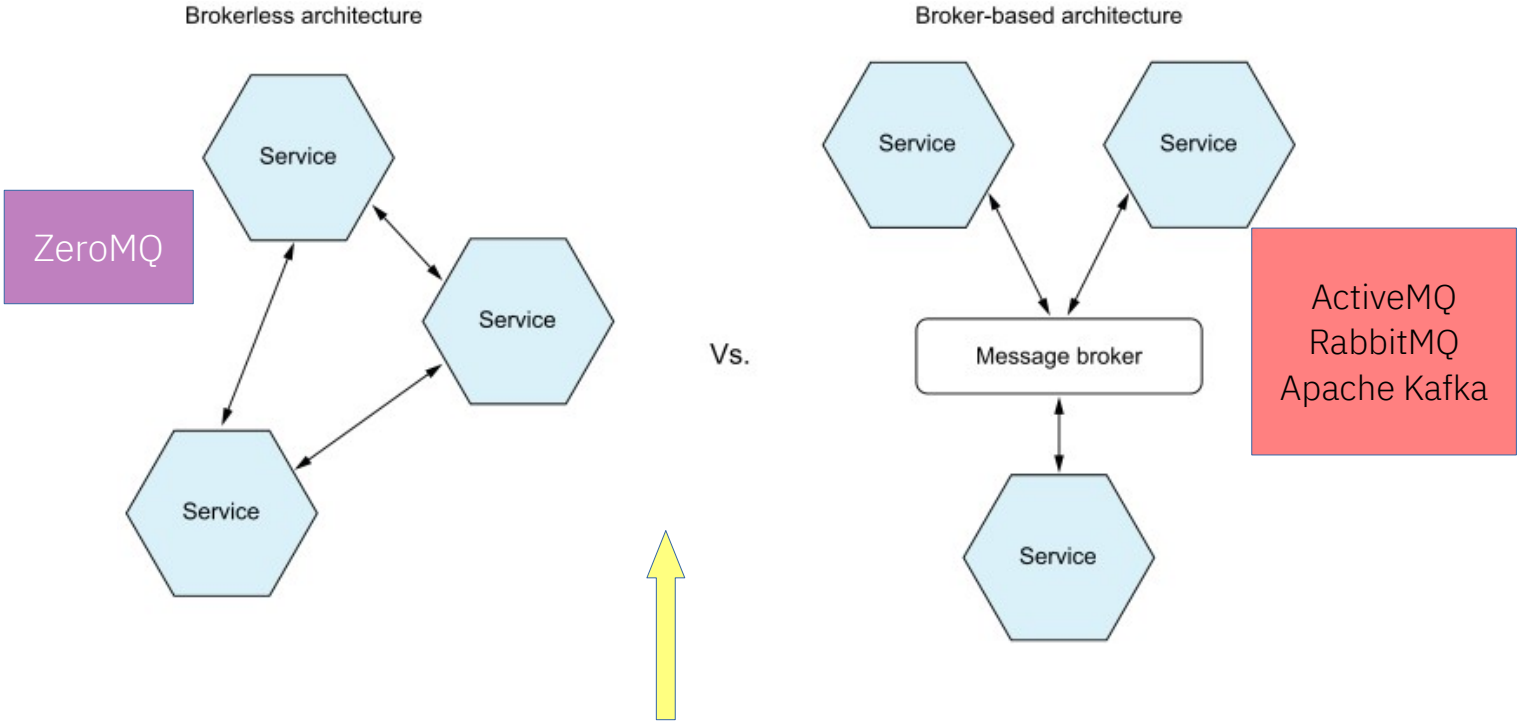
IMPLEMENTING REQUEST/RESPONSE & ASYNCHRONOUS REQUEST/RESPONSE



Implementing asynchronous request/response by including a reply channel and message identifier in the request message.

The receiver processes the message and sends the reply to the specified reply channel.

Using a message broker



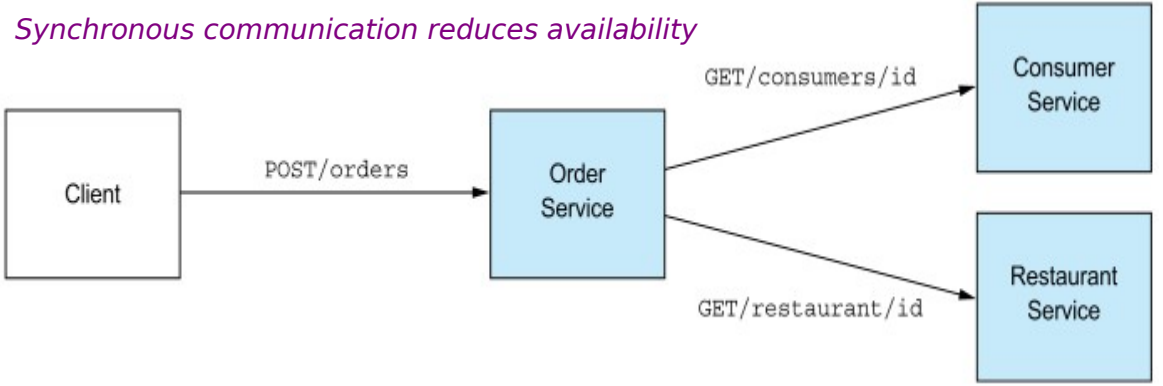
The services in brokerless architecture communicate directly.

The services in a broker-based architecture communicate via a message broker.

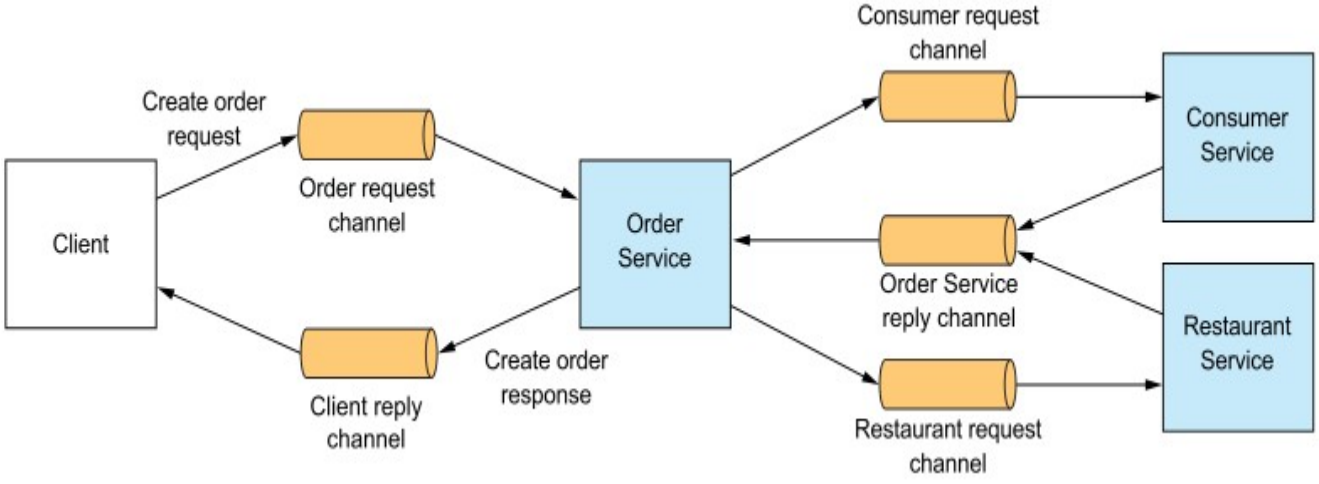
OrderService invokes other services using REST.

It's straightforward, but it requires all the services to be simultaneously available, which reduces the availability of the API.

Synchronous communication reduces availability

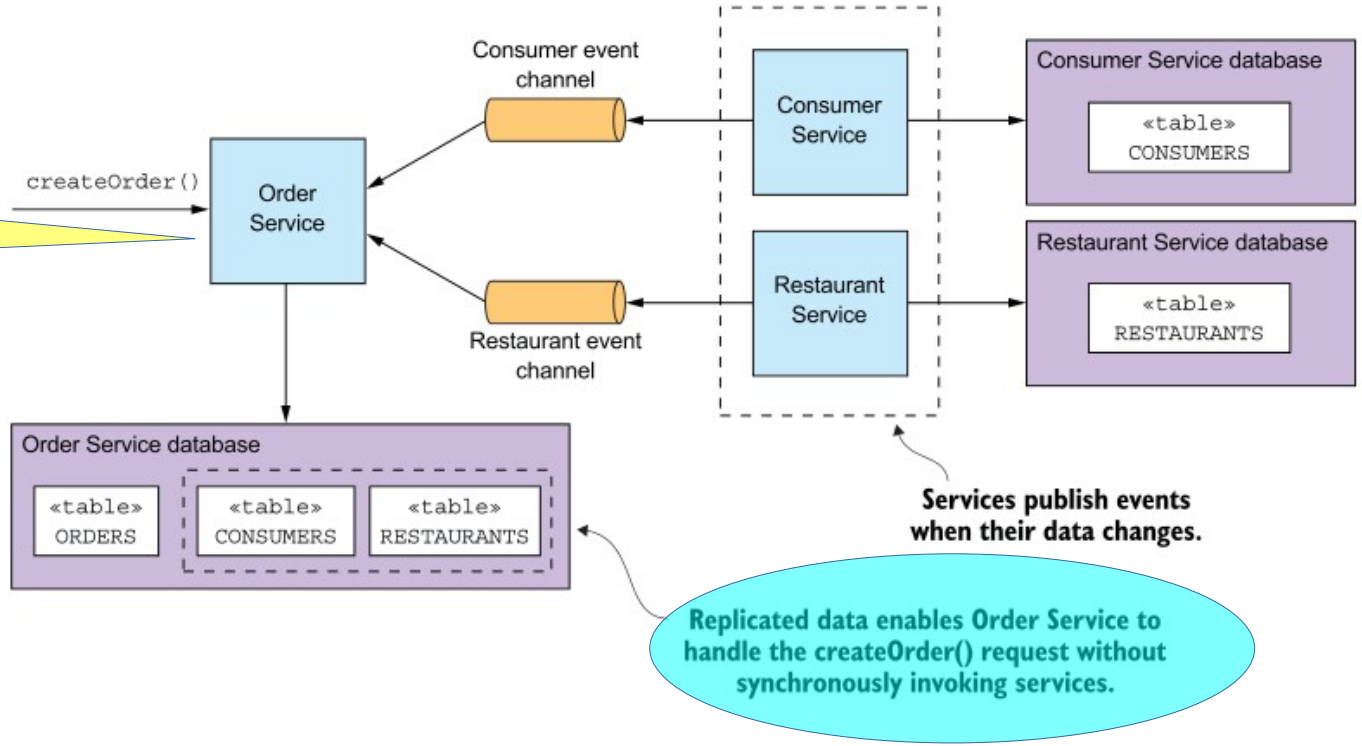


We should design your servicesto use asynchronous messaging whenever possible.



The application has higher availability if its services communicate using asynchronous messaging instead of synchronous calls.

OrderService is self-contained because it has replicas of the consumer and restaurant data



Managing transactions with sagas

Let’s dicsuss

Why distributed transactions aren’t a good fit for modern applications

Using the Saga pattern to maintain data consistency in a microservice architecture

Coordinating sagas using choreography and orchestration

Using countermeasures to deal with the lack of isolation

Transactions

Transactions are an essential ingredient of every enterprise application.

Without transactions it would be impossible to maintain data consistency.

ACID – Atomicity,Consistency, Isolation, Durability for Database system provides transaction.

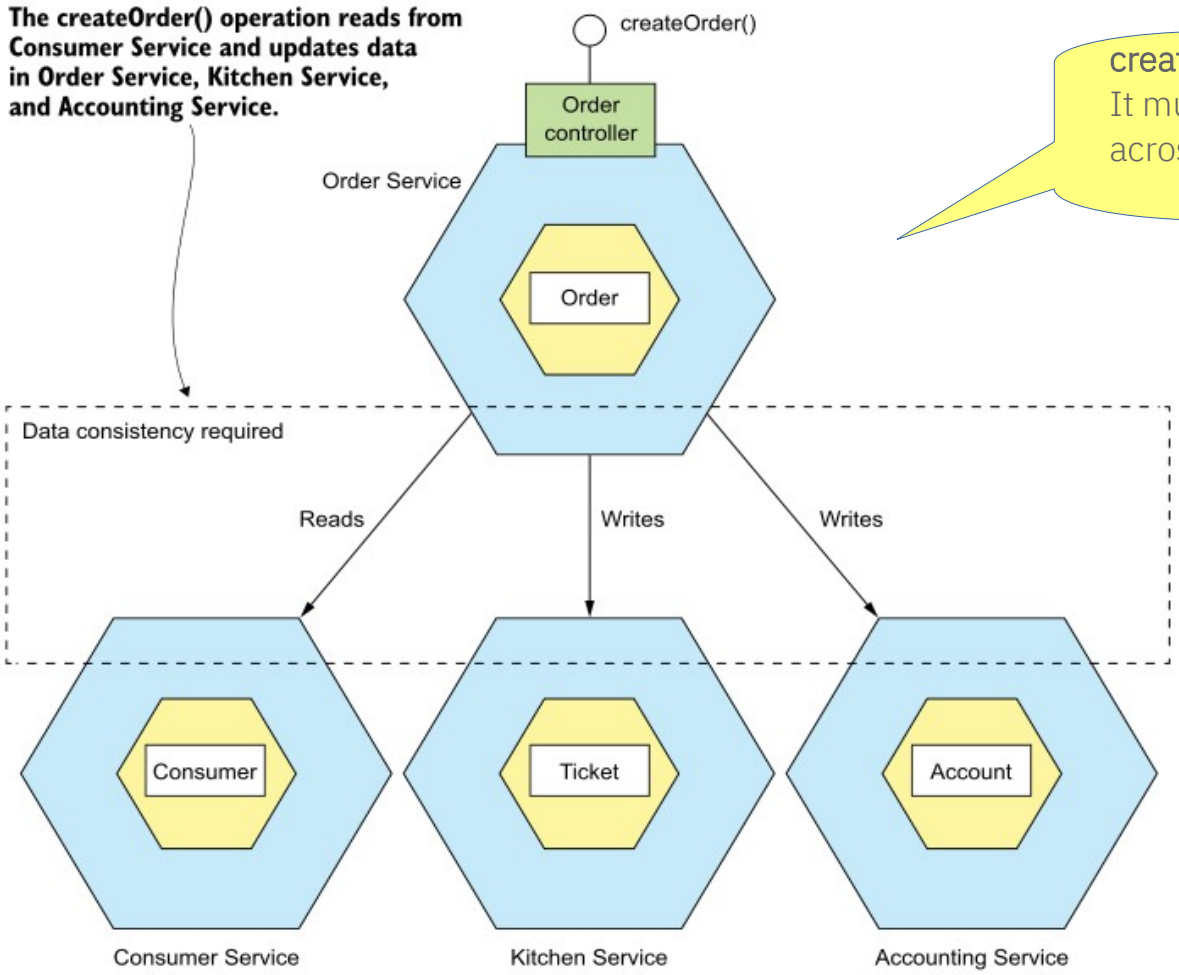
Microservices Data consistemcy challenge ?

Implementing transactions for operations that update data owned by multiple services.

E.g

createOrder() operation spans numerous services, including OrderService, KitchenService, and AccountingService.

Operations such as these need a transaction management mechanism that works across services.



createOrder() operation updates data in several services. It must use a mechanism to maintain data consistency across those services.

Using the Saga pattern to maintain data consistency

saga, a message-driven sequence of local transactions, to maintain data consistency.

Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions.

We define a saga for each system command that needs to update data in multiple services.

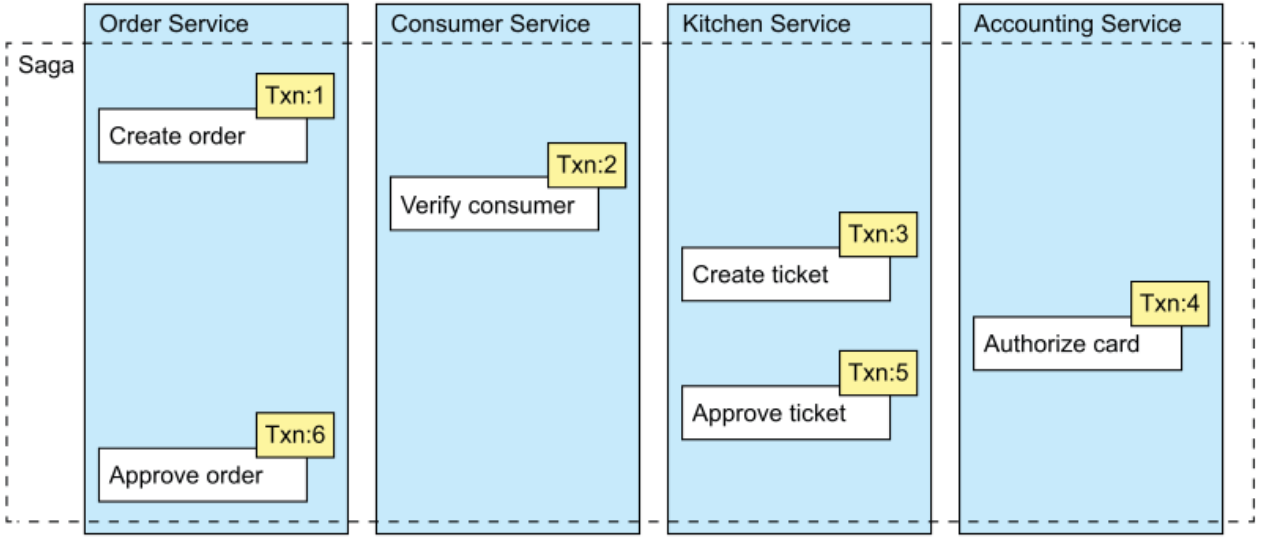
A saga is a sequence of local transactions.

Each local transaction updates data within a single service using the familiar ACID transaction frameworks and libraries.

SAGAS pattern is for loosely coupled, asynchronous services

Saga is ACD – Atomicity, Consistency, Durability, lacks isolation

CREATE ORDER SAGA Example



Creating an **Order** using a saga. The **createOrder()** operation is implemented by a saga that consists of local transactions in several services.

This saga consists of the following local transactions:

1. **OrderService** — Create an Order in an APPROVAL_PENDING state.
2. **ConsumerService** — Verify that the consumer can place an order.
3. **KitchenService** — Validate order details and create a Ticket in the CREATE_PENDING.
4. **AccountingService** — Authorize consumer's credit card.
5. **KitchenService** — Change the state of the Ticket to AWAITING_ACCEPTANCE.
6. **OrderService** — Change the state of the Order to APPROVED.