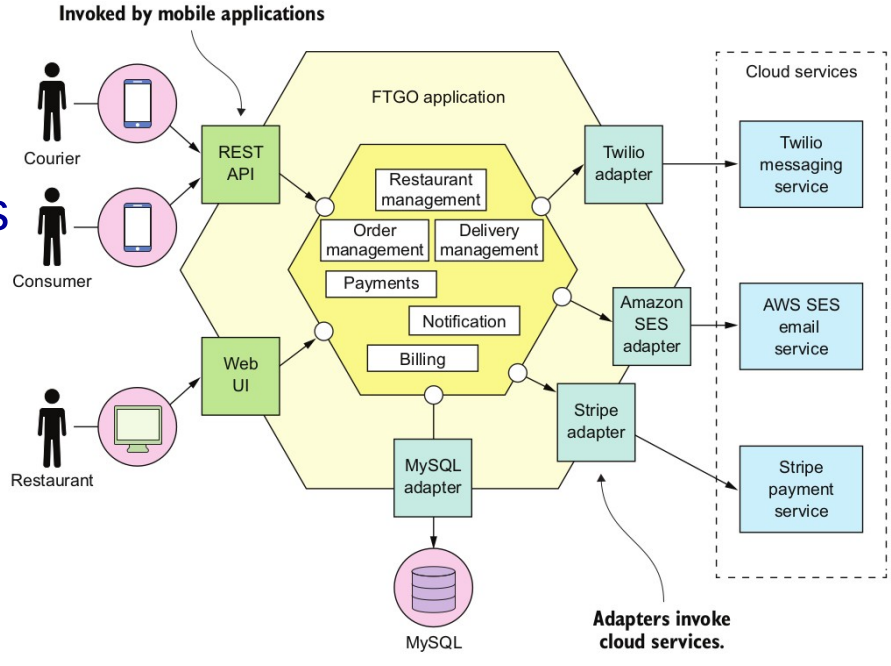
The microservice architecture style

"The software architecture of a computing system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both."

Documenting software architecture from Bass et al.

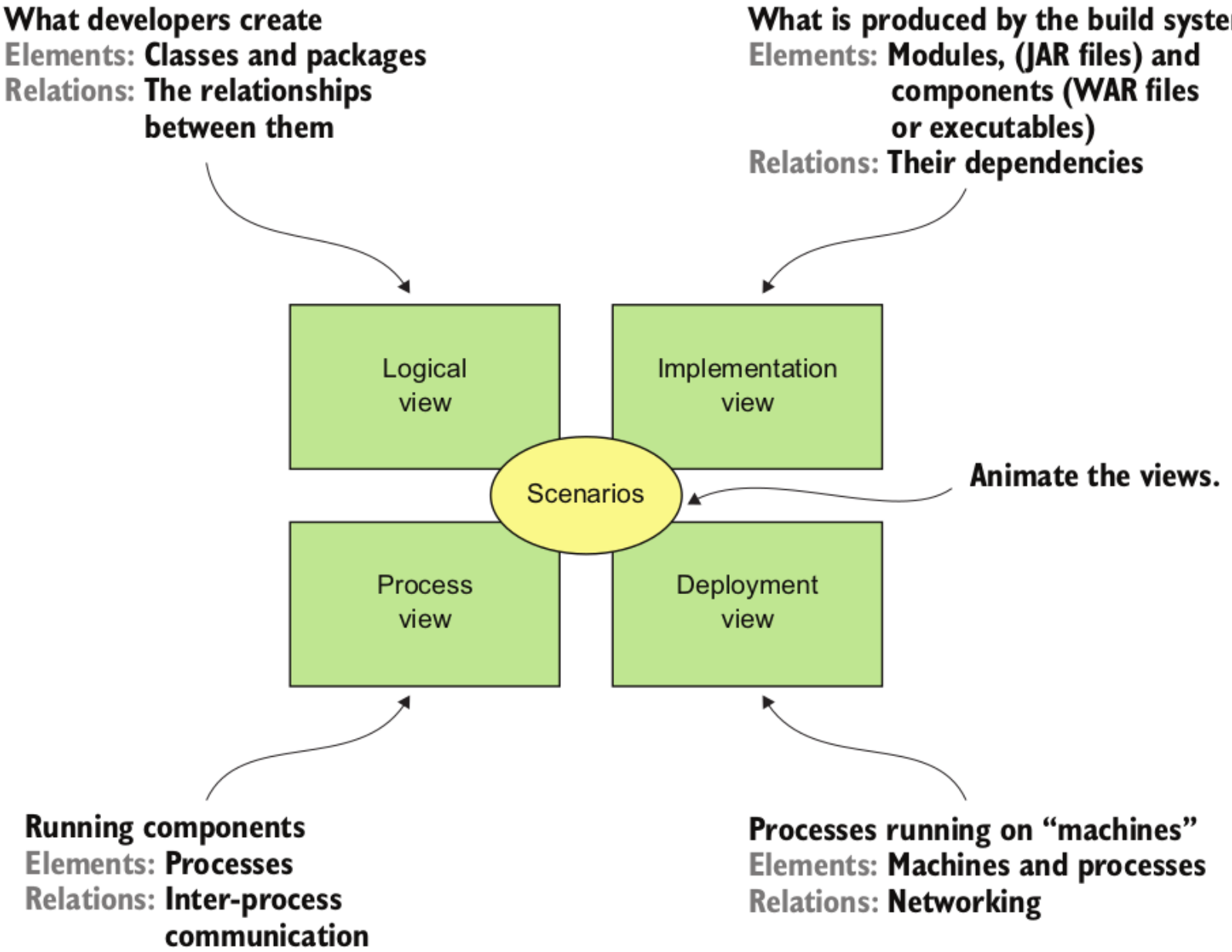
A running example

- FTGO (Food-to-go) application
  - Consumers use the FTGO web-site or mobile application to place food orders at local restaurants
- FTGO coordinates a network of
  - couriers who deliver the orders
  - restaurants that accept orders
- FTGO is responsible for
  - paying couriers and restaurants
- Restaurants use the FTGO website to
  - edit menus
  - manage orders
- The application uses various web services
  - payments
  - messaging
  - email

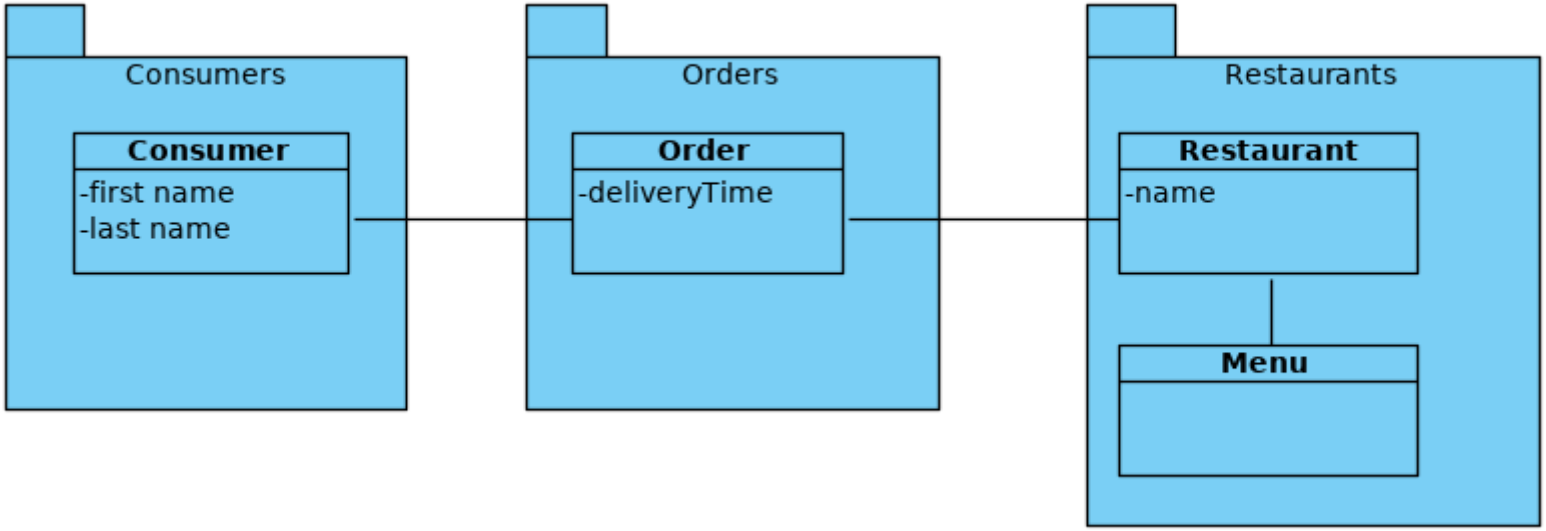


- Architecture = (elements, relations, properties)
- Architecture is multi-dimensional
  - Eg: Structural, electrical, …
- Described by multiple views
  - Focus on different (elements, relations, properties)

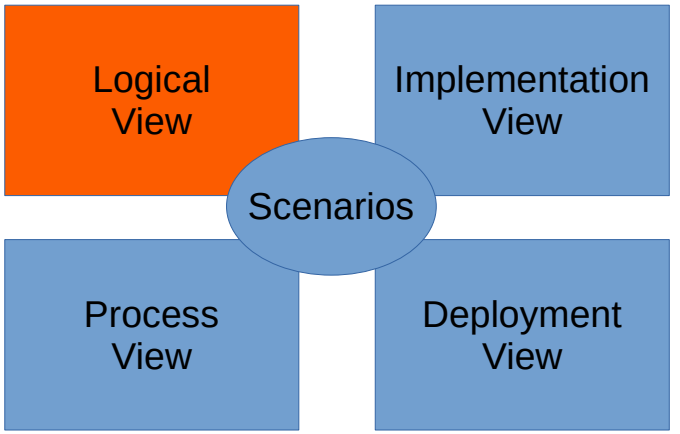
4+1 view model



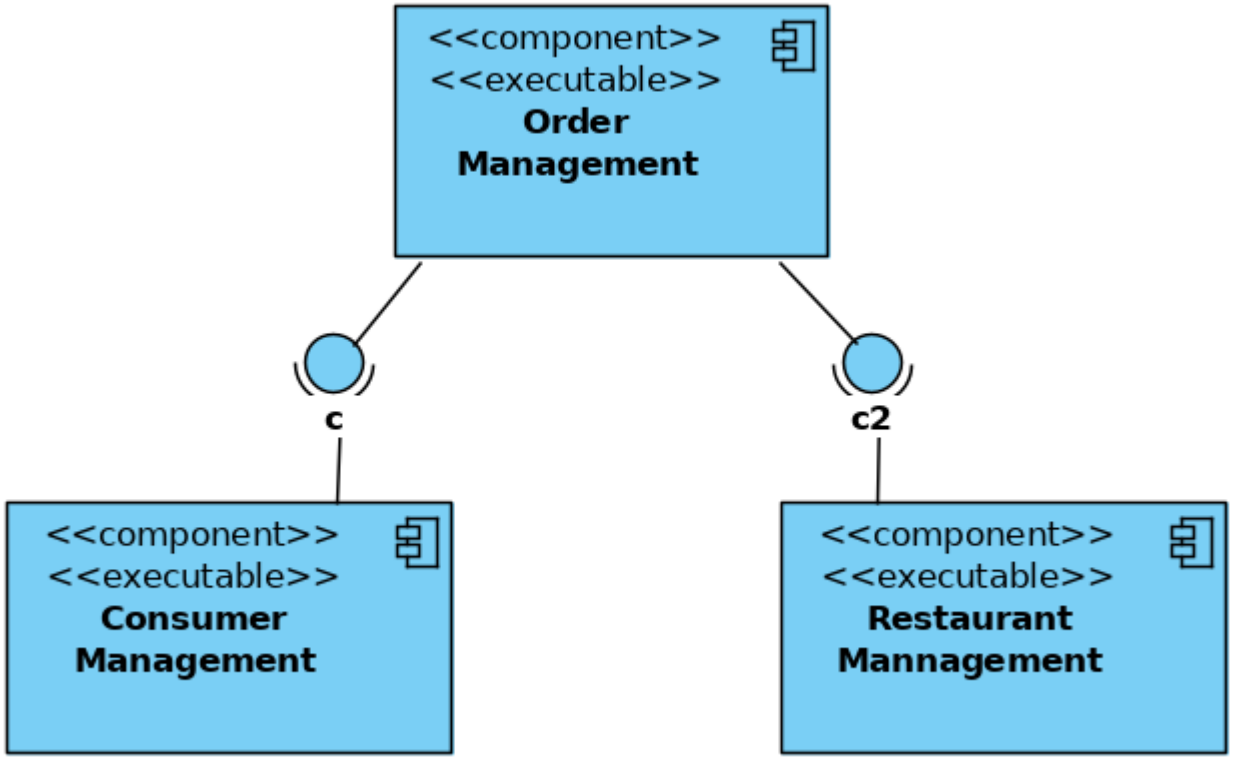
Logical view



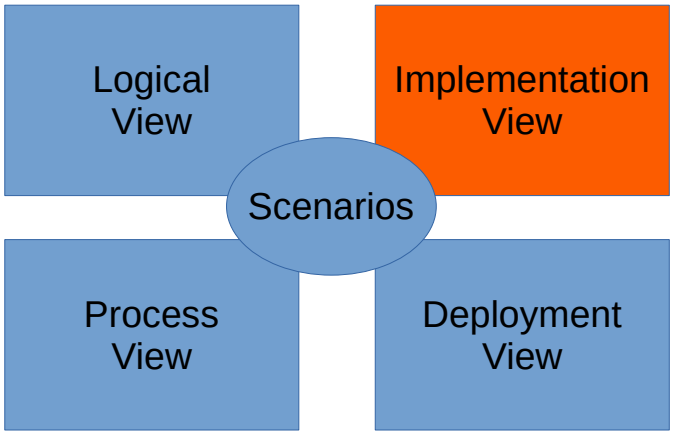
Elements: class, packages
Relations: inheritance, associations



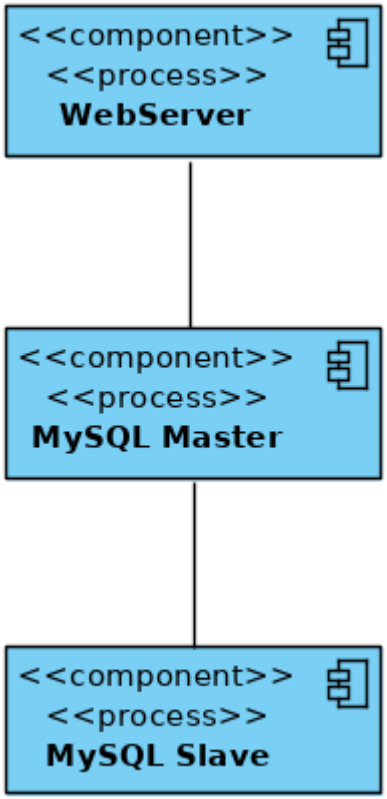
Implementation view



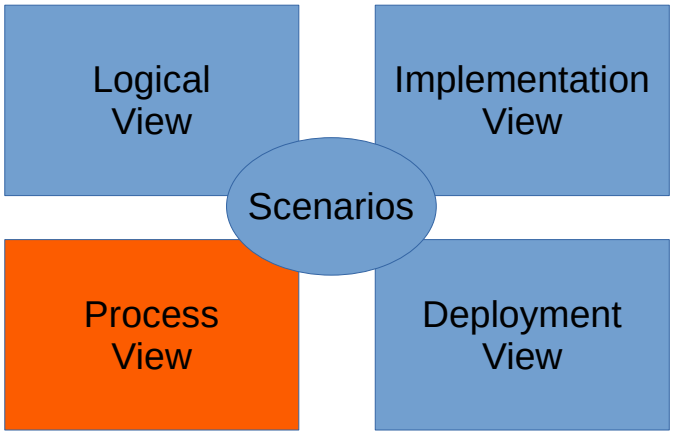
Elements: modules, components
Relations: dependencies



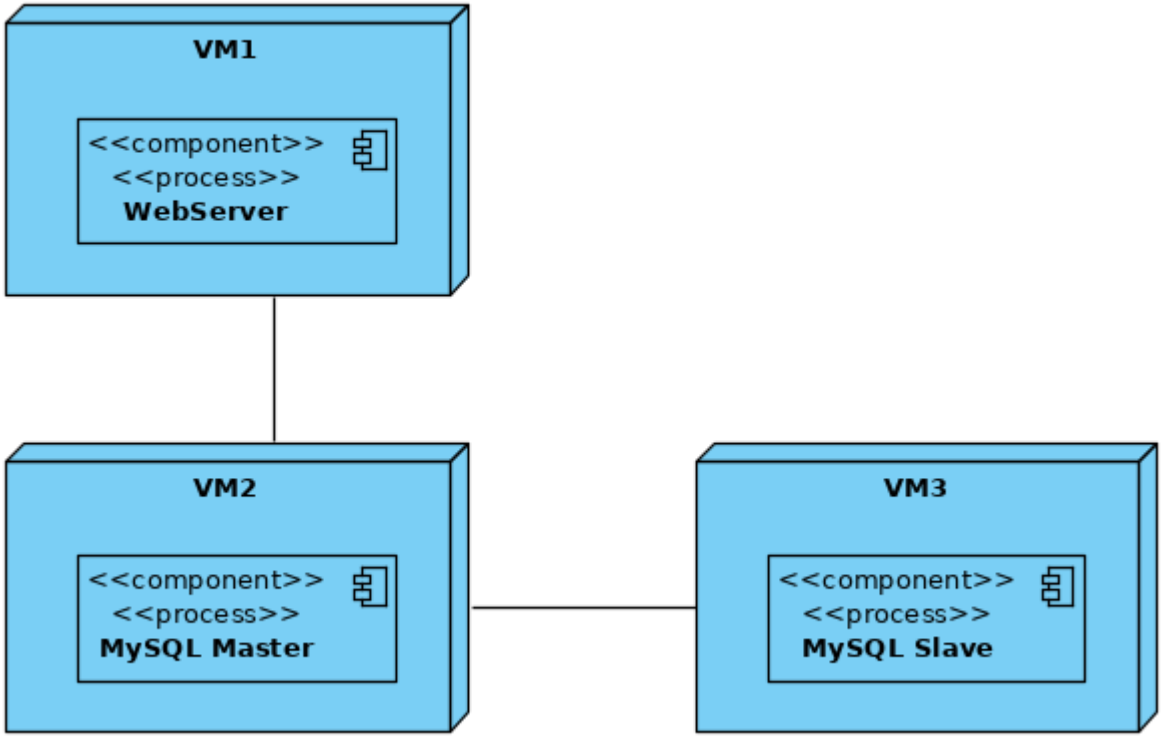
Process view



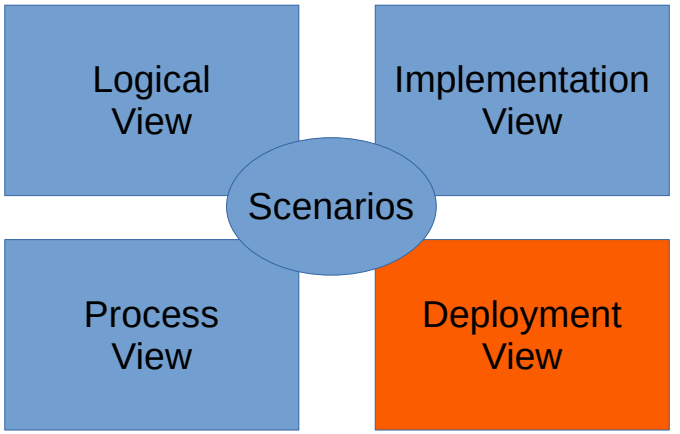
Elements: processes
Relations: IPC



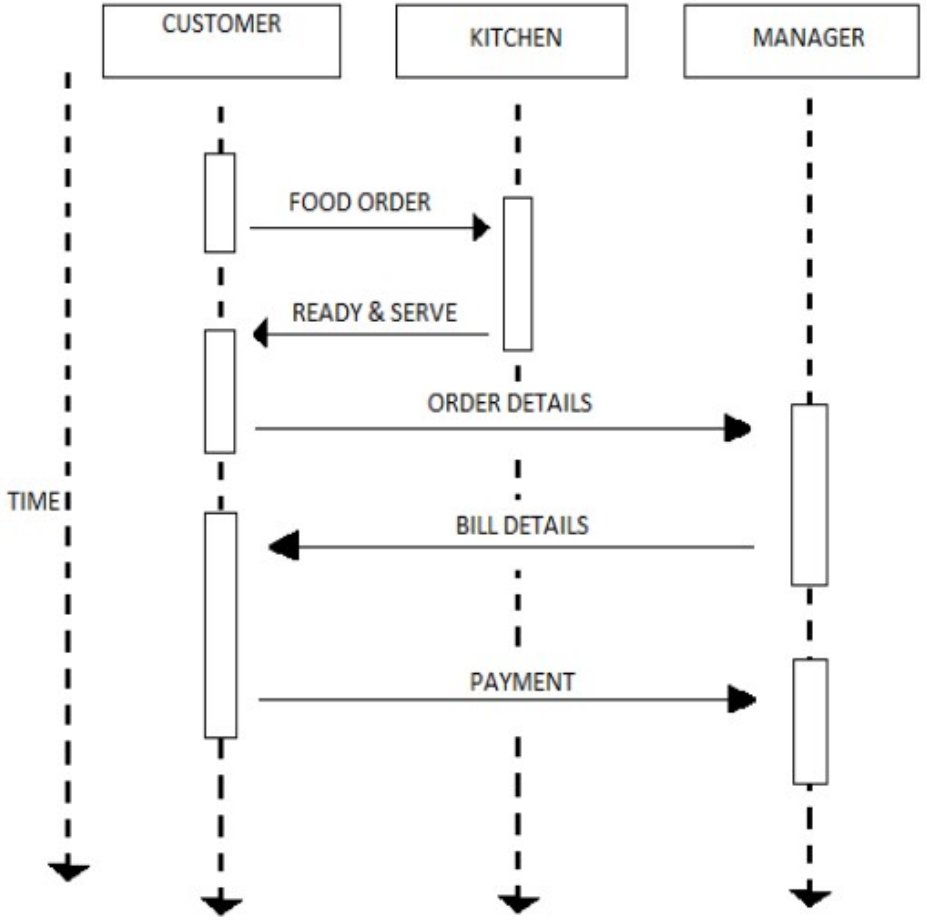
Deployment view



Elements: nodes, “machines”
Relations: networking

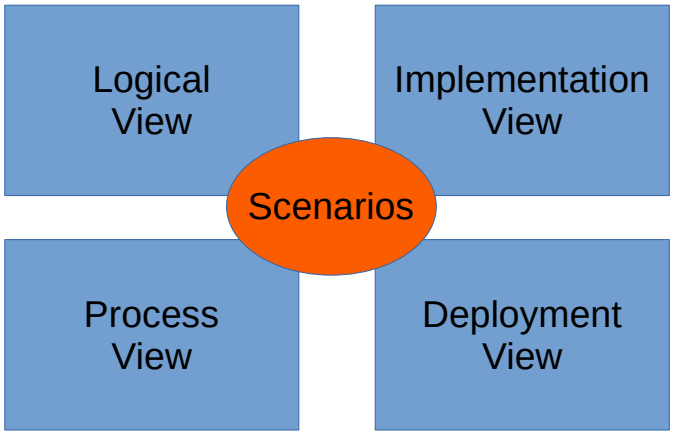


Scenarios



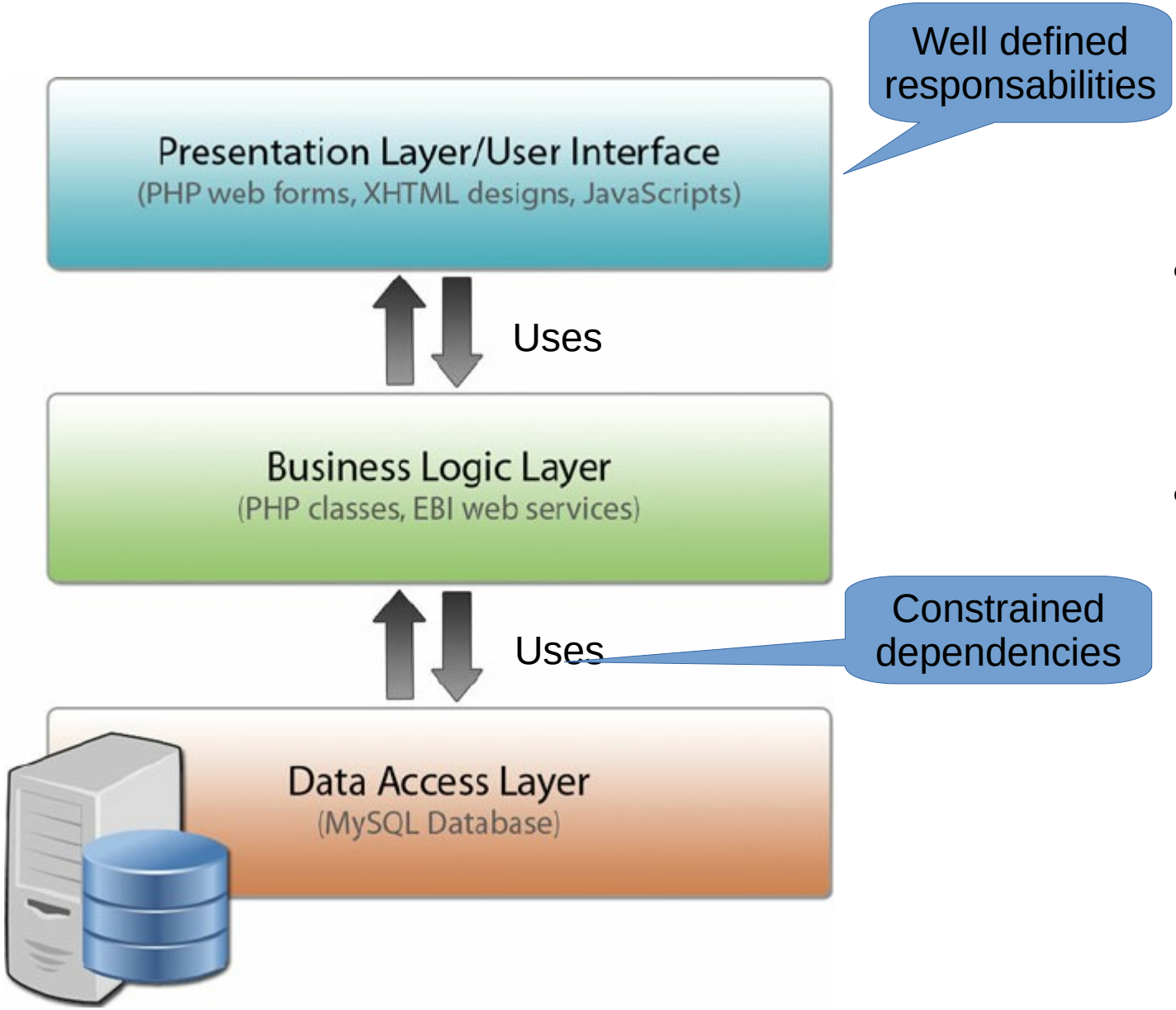
Derived from use cases/stories

Animate the views



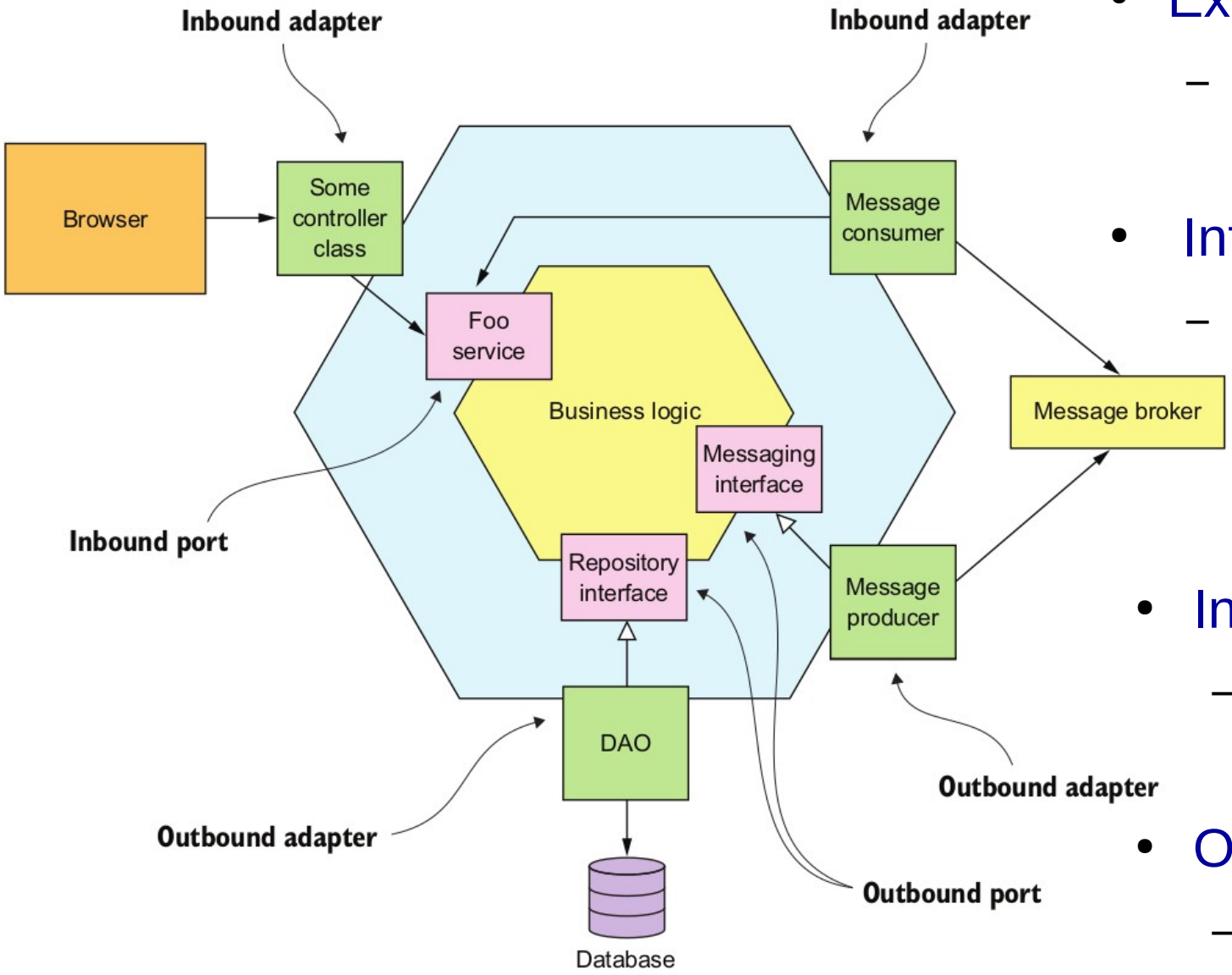
- Decomposition is important because:
  - facilitates the division of labor and knowledge enabling multiple people/teams to work together
  - defines how the software elements interact.
  - It's the decomposition into parts and their relationships between those parts that determine the application's -ilities
    - Scalability, maintainability, availability

Layered/tiered architecture style



- Layer
  - Logical view
- Tier
  - Deployment view

HEXAGONAL ARCHITECTURE STYLE



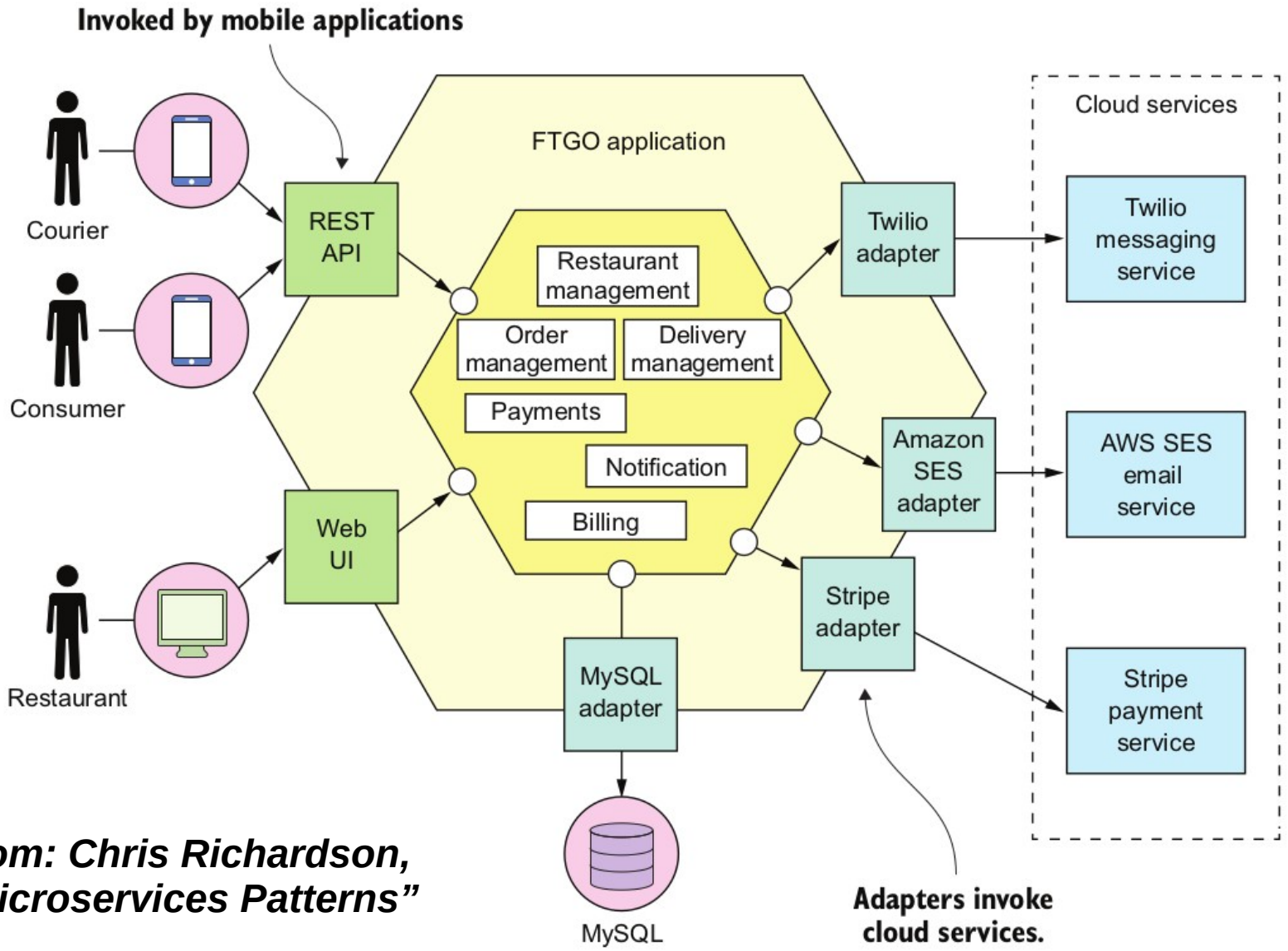
- External hexagon
  - Interface with other entities
- Internal hexagon
  - Business logic

- Inbound ports
  - Exposed API by businnes logic
- Outbounds ports
  - To call external services

From: Chris Richardson, "Microservices Patterns"

A monolithic example

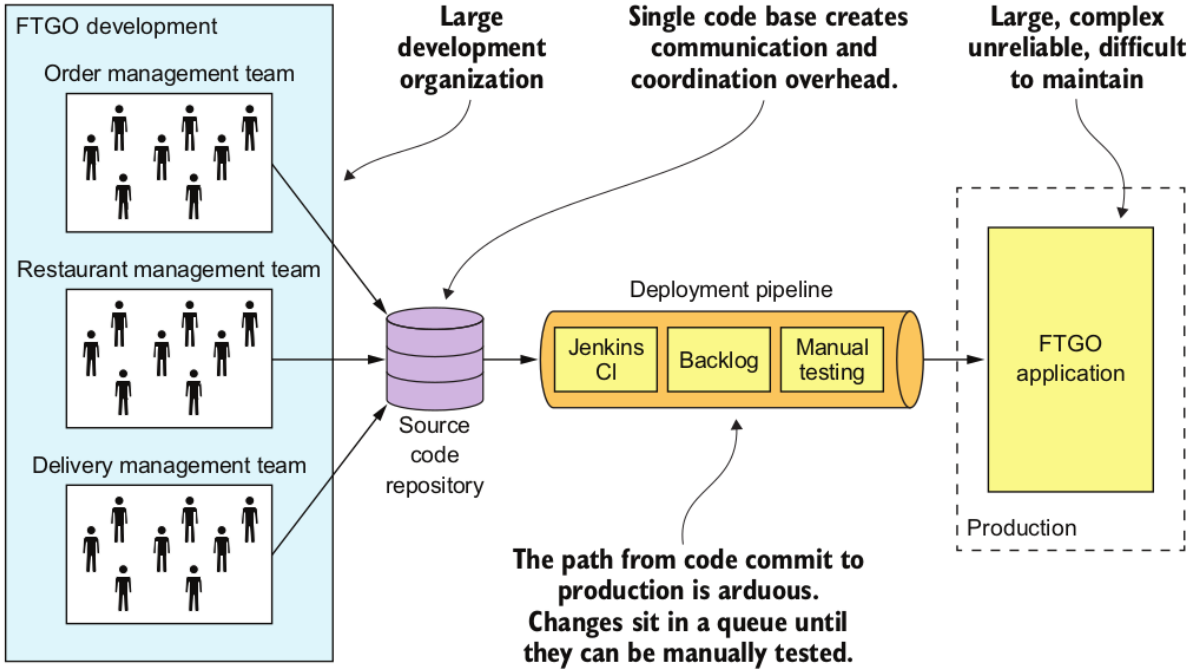
- Whole application in a single executable artifact
  - EXE, JAR, WAR, ...



From: Chris Richardson, "Microservices Patterns"

A monolithic example: drawbacks

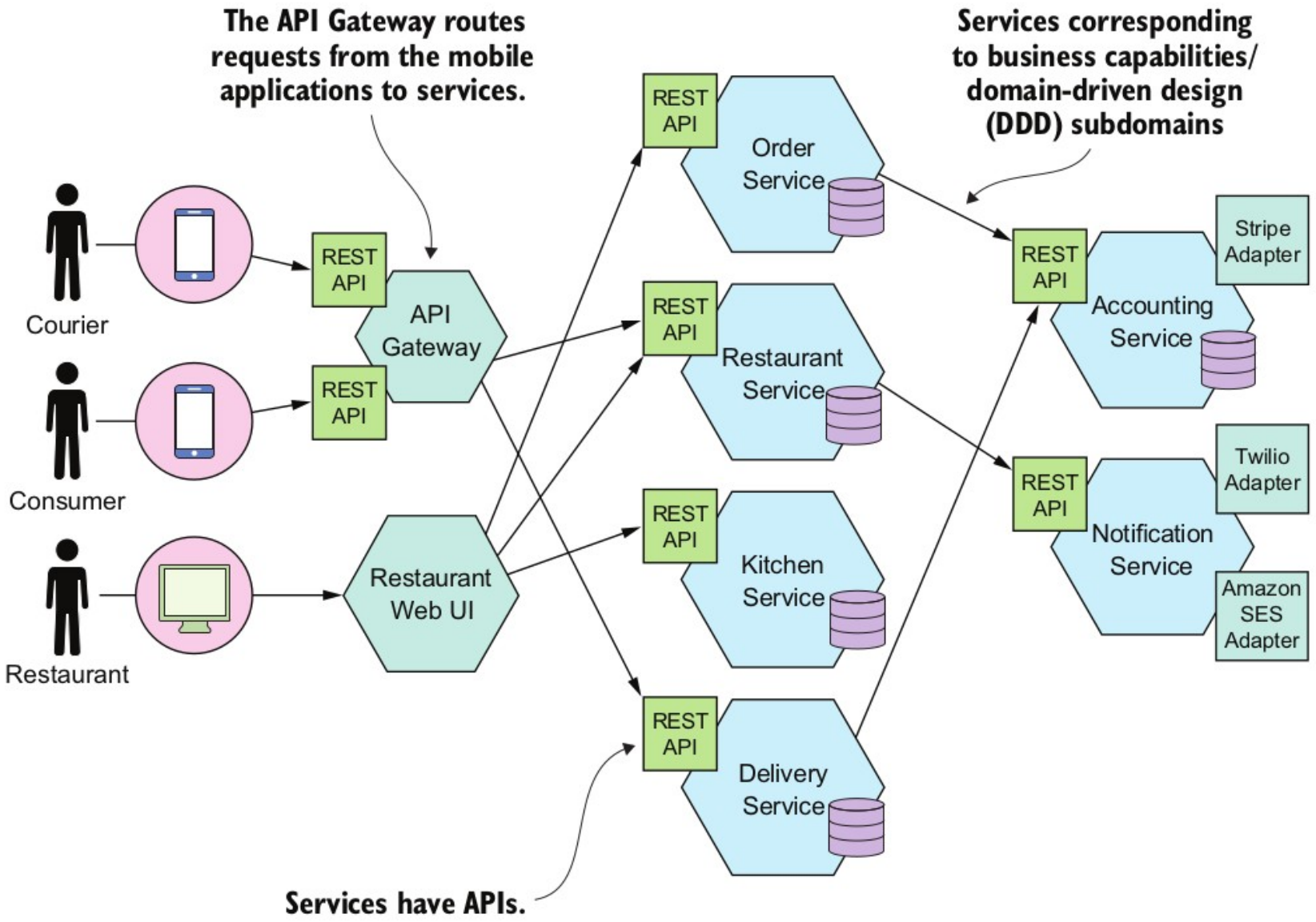
- Several problems in monolithic architecture:
  - When system and code size increase, decrease code maintainability
  - Difficult to scale
  - Number of versions to maintain rapidly increases with updates
  - Long time to develop new releases



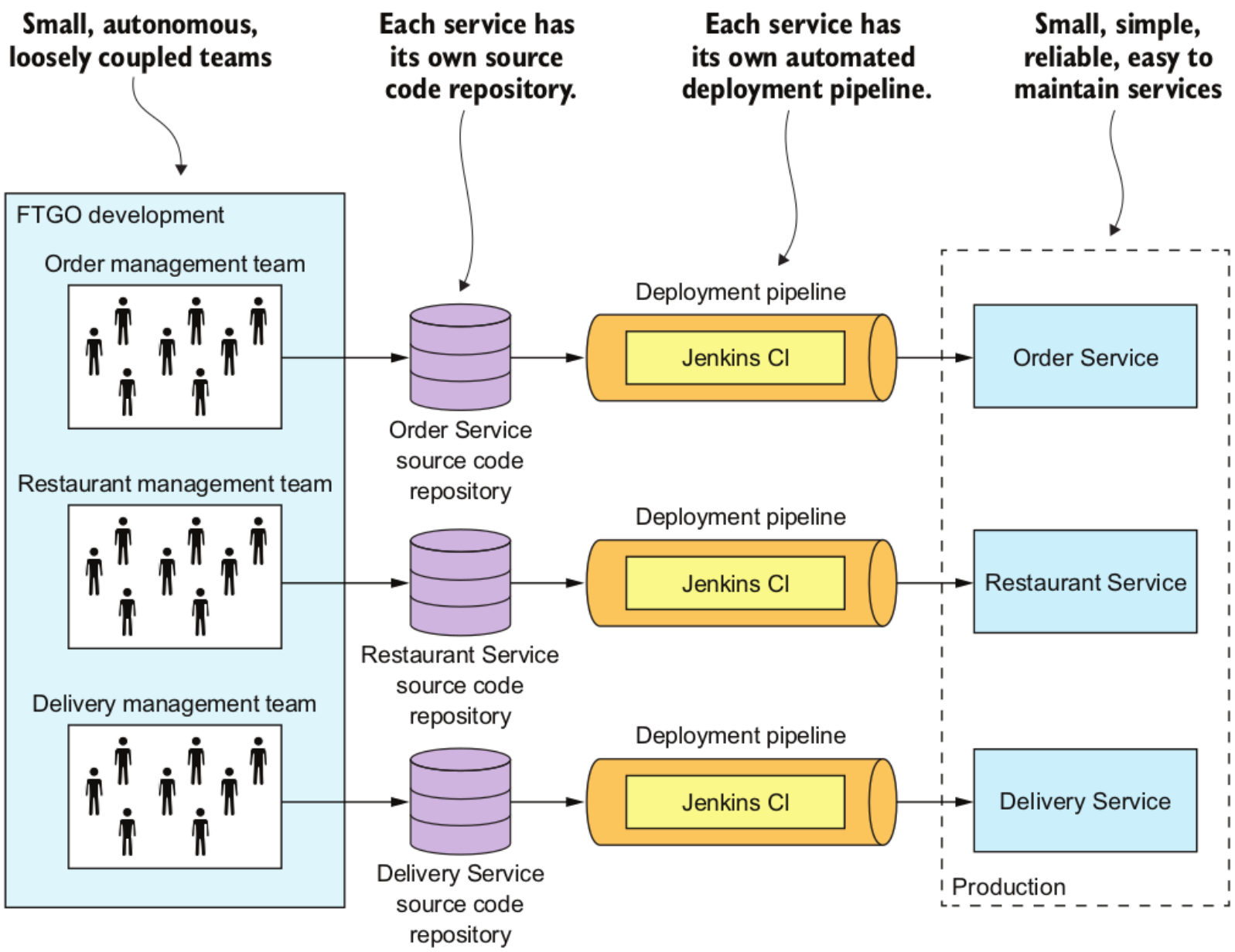
From: Chris Richardson, "Microservices Patterns"

The microservice translation

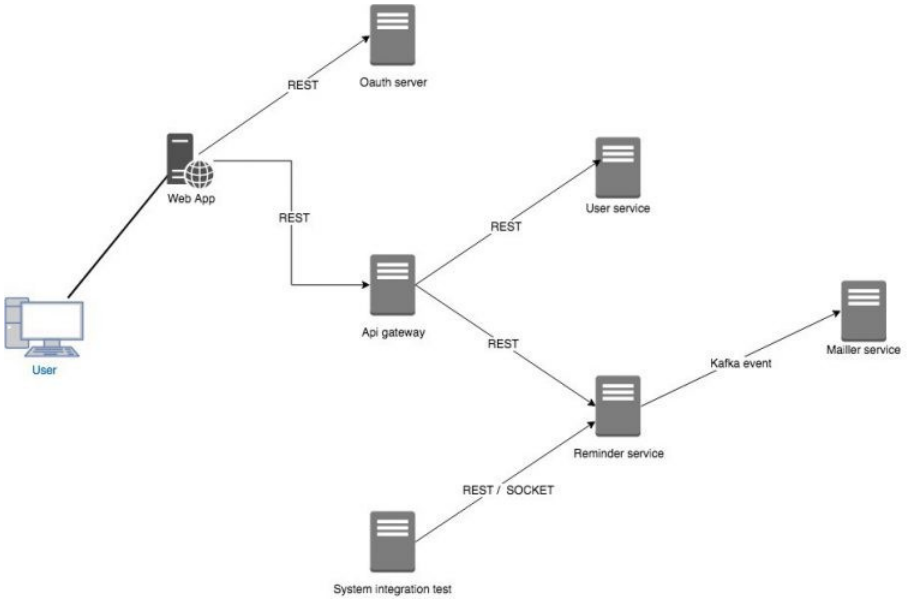
- Each service run as a different executable
  - Interacting each other by IPCs

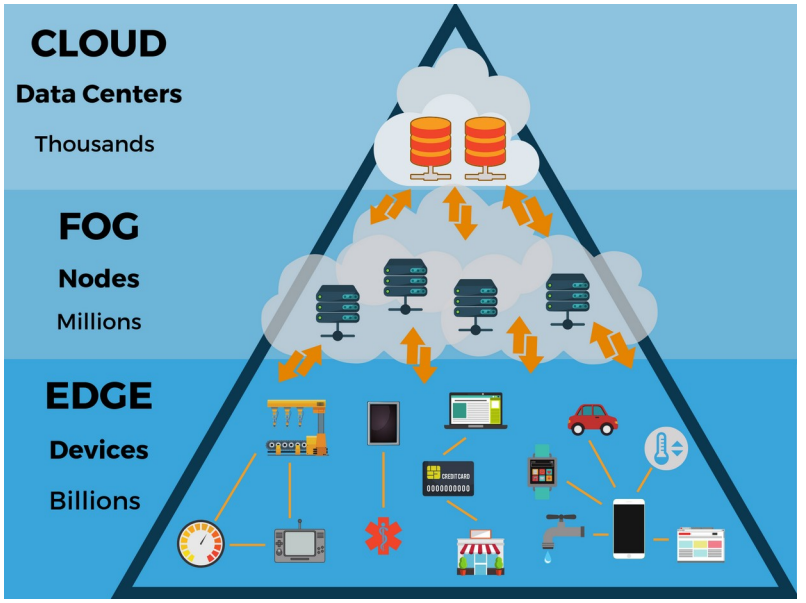


Microservice example: workflow



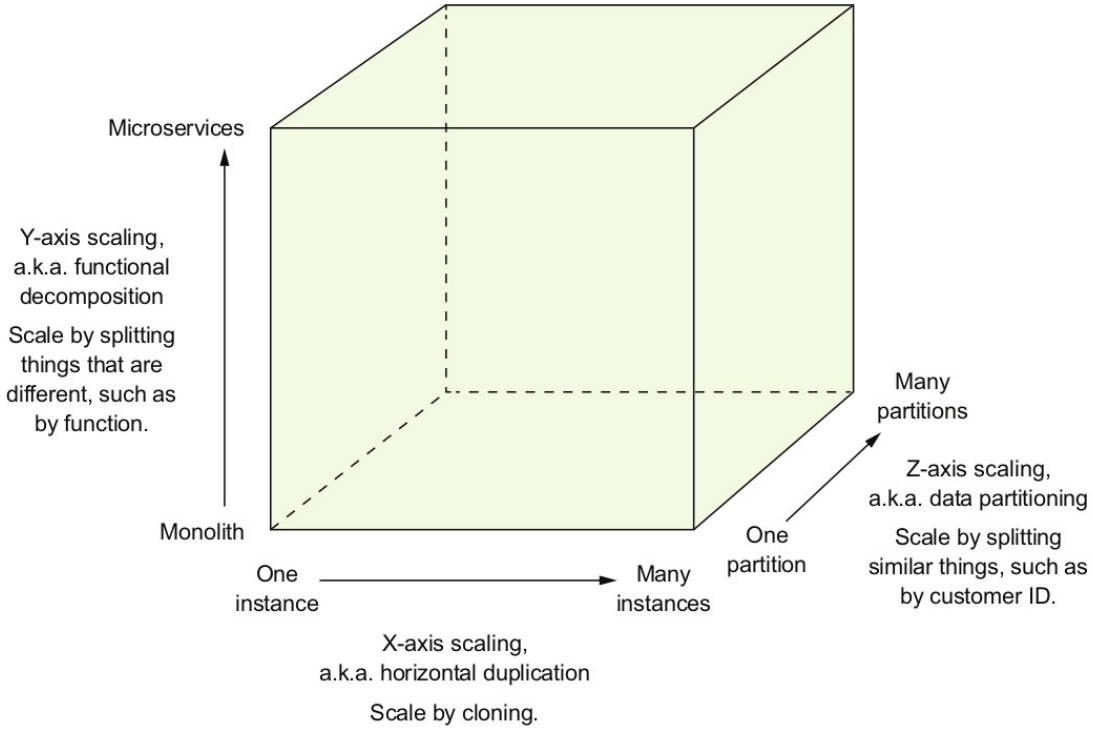
Scalability: Distribution





- No need to have one very powerful server to run your application.
- You could use several relatively weaker servers and run different microservices on each of the m.
- The system will have better IO performance since the tasks are distributed and can be done simultaneously.
-

Scalability: decomposition



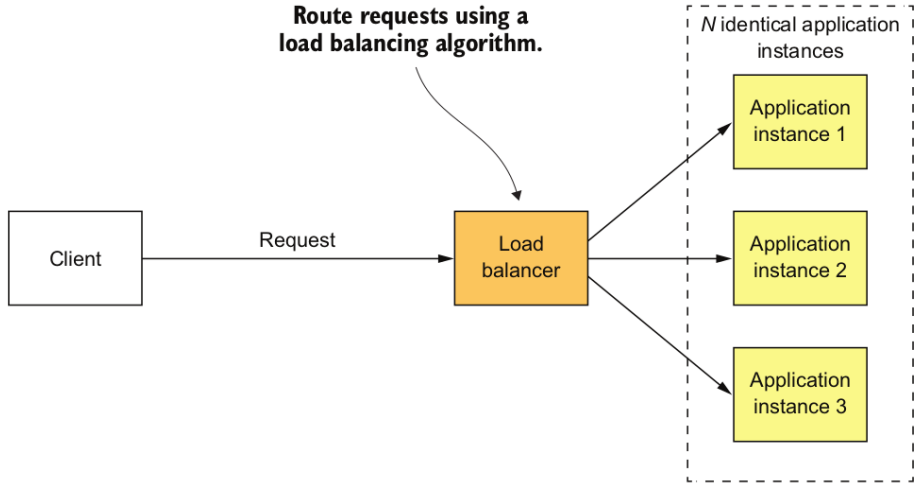


Figure 1.4 X-axis scaling runs multiple, identical instances of the monolithic application behind a load balancer.

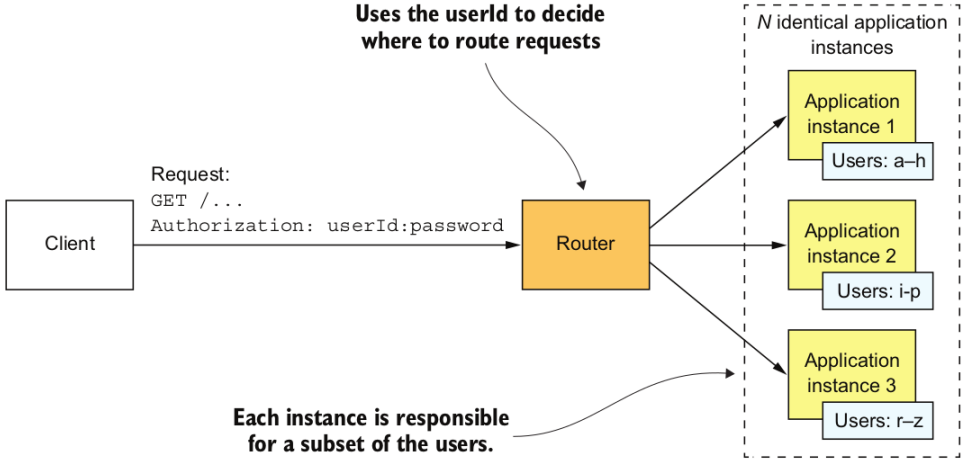
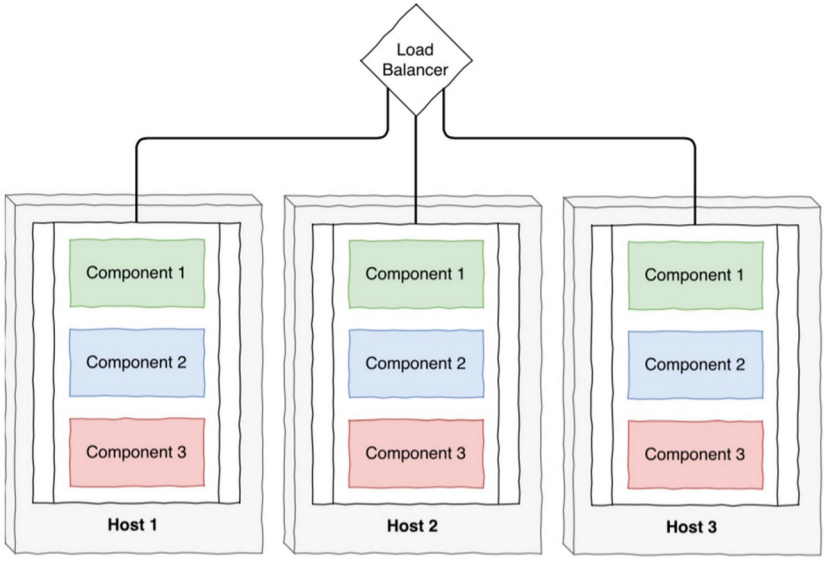


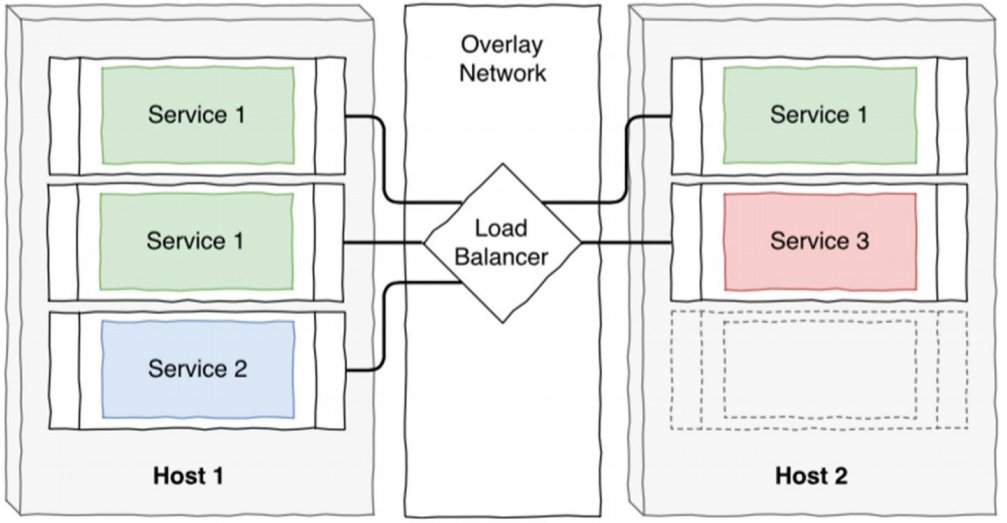
Figure 1.5 Z-axis scaling runs multiple identical instances of the monolithic application behind a router, which routes based on a `request` attribute . Each instance is responsible for a subset of the data.

Scalability: Non-uniform Scaling

Scalability —— Non-uniform Scaling



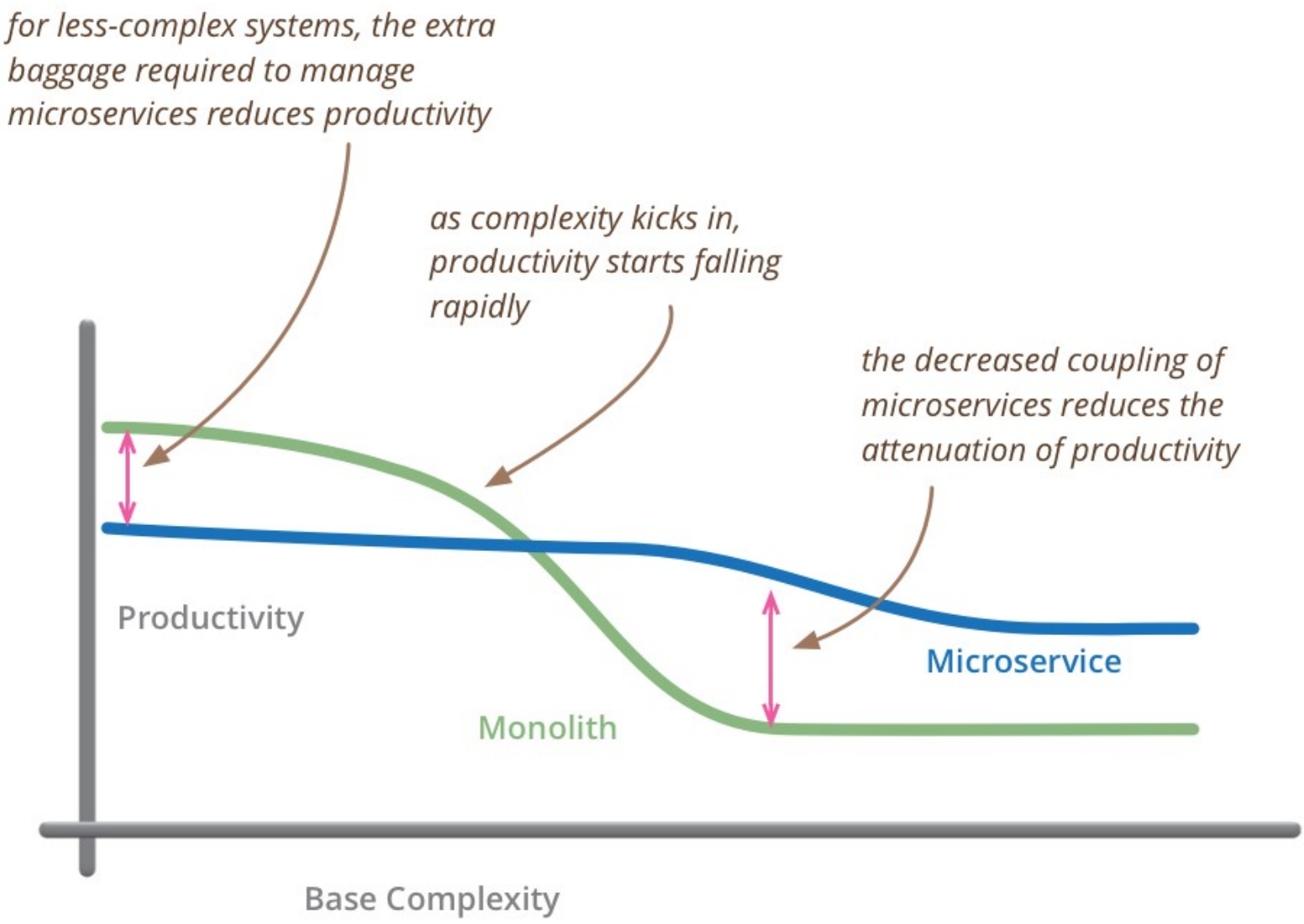
Monolithic Architecture



Microservice Architecture

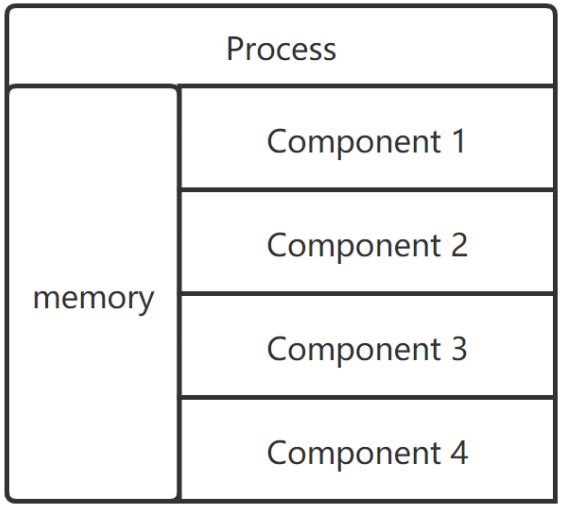
- The bottleneck may only exist in some part of the application
- No sense to replicate the whole application
- In Microservice we can do non-uniform scaling
  - deploying multiple instance of particular service so
  - we can get higher utilization with less server resource.

Where should we use Microservice

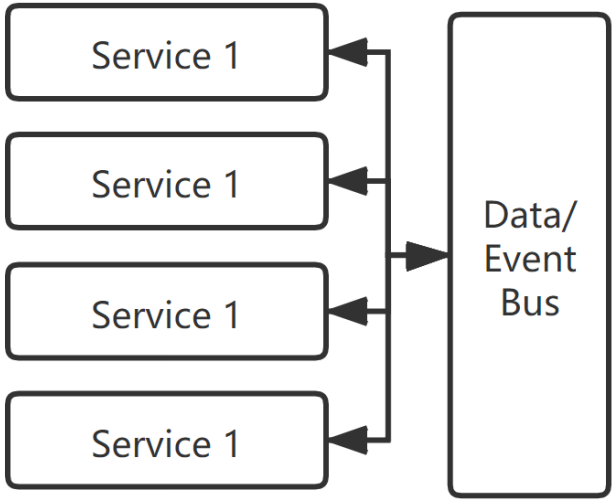


but remember the skill of the team will outweigh any monolith/microservice choice

Where should we use Microservice



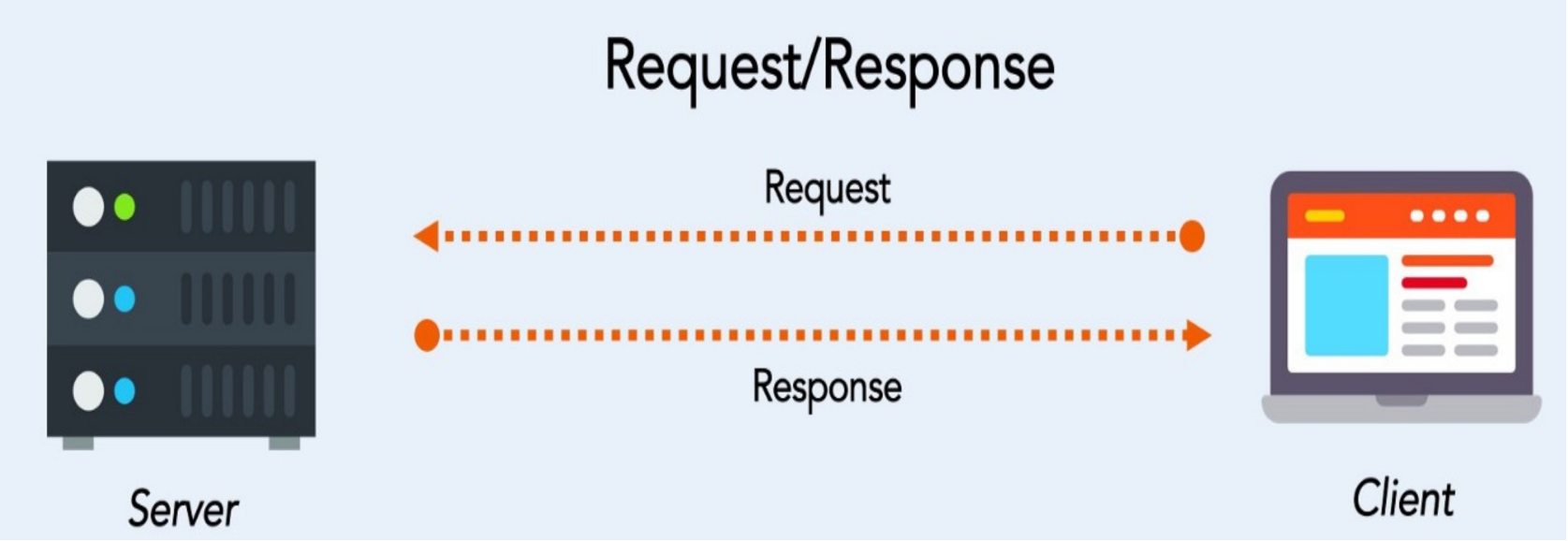
Monolithic Architecture

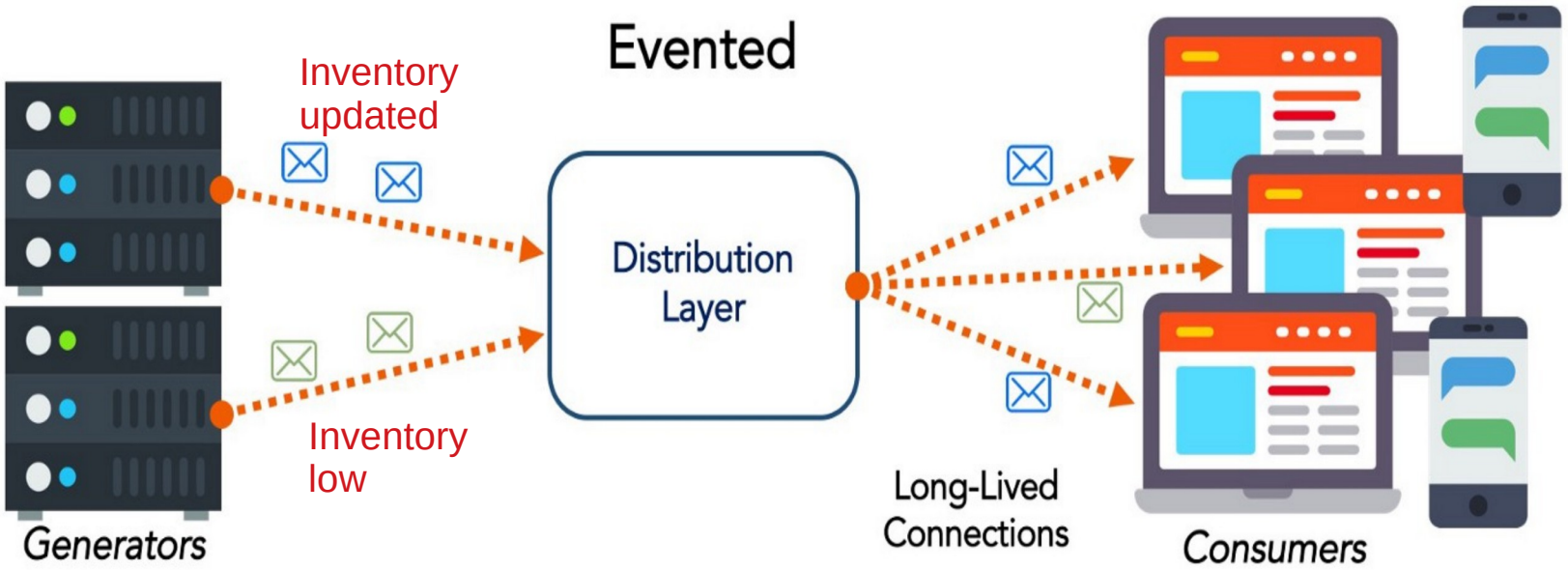


Microservice Architecture

- In monolithic system, you can take advantage of in-memory state like storing session in memory
- In microservice, because of the distribution, services can not share memory data
- If you heavily rely on in-memory state or can not accept the latency caused by the data sharing in the distributed microservice architecture, you shouldn't use microservice here.

From request driven to event driven





Events

- Anything happened (or didn’t happen).
- A change in the state.
- An event is always named in the past tense and is immutable
- A condition that triggers a notification

CustomerAddressChanged

InventoryUpdated

SalesOrderCreated

PurschaseOrderCreated

- “Real-time” events as they happen at the producer
- Push notifications
- One-way “fire-and-forget”
- Immediate action at the consumers

Benefits of event driven

- Supports the business demands for better service (no batch, less waiting)
- No point-to-point integrations (fire & forget)
- Fault tolerance, scalability, versatility, and other benefits of loose coupling
- Greater operational efficiencies

Discussion: positive

- Very strong scalability
- Distribution
- Non-uniform Scaling
- Portability
- Availability
- Programming language doesn’t matter, easy to find a developer to build microservices
- For a complex system, microservice architecture has higher productivity
- Agile friendly
- Microservices arch
  - Adopted by Netflix, eBay, Amazon, and many others

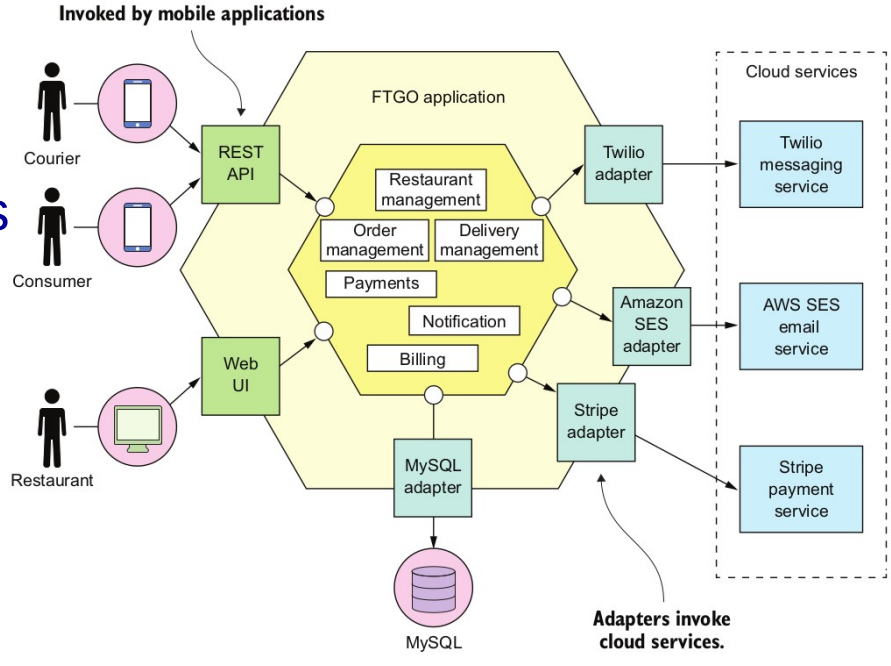
Discussion: negative

- The granularity of a microservice architecture is hard to decide
- Microservice need more precise documentation
- Microservice should be integrated with Continuous Integration and Continuous Delivery, otherwise it will cost too much human resource in maintenance
- NO SILVER BULLET !

Defining microservice architecture

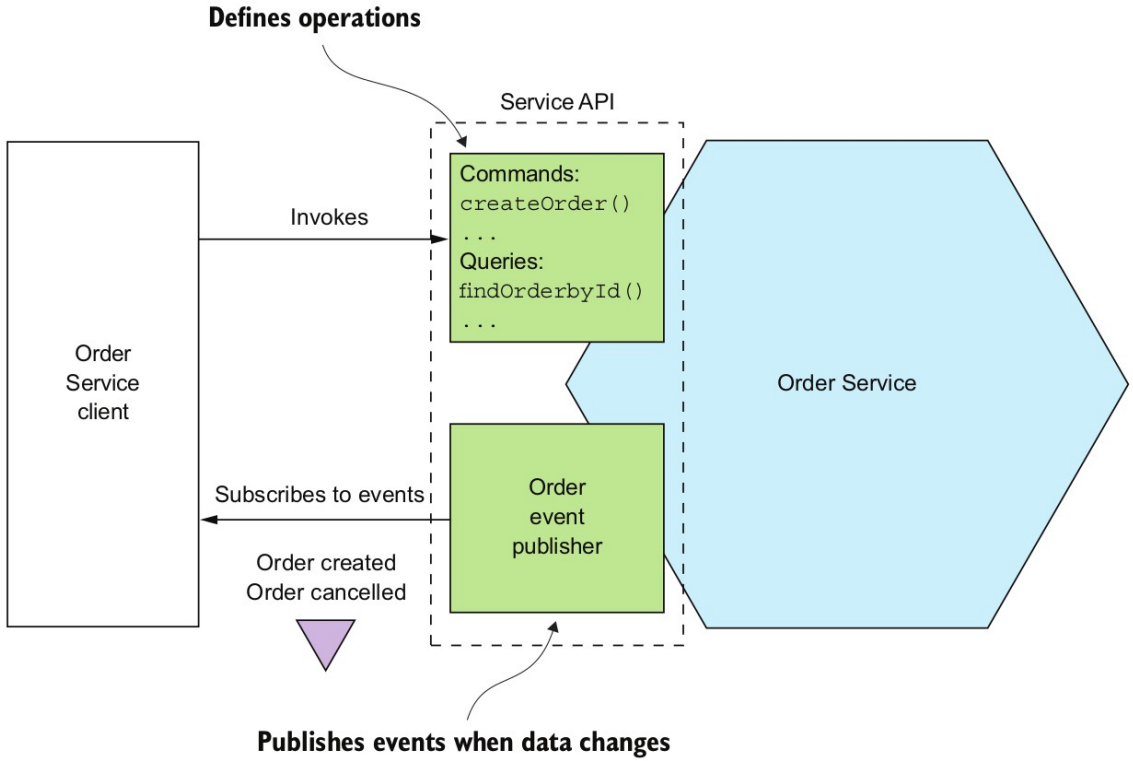
A running example

- FTGO (Food-to-go) application
  - Consumers use the FTGO web-site or mobile application to place food orders at local restaurants
- FTGO coordinates a network of
  - couriers who deliver the orders
  - restaurants that accept orders
- FTGO is responsible for
  - paying couriers and restaurants
- Restaurants use the FTGO website to
  - edit menus
  - manage orders
- The application uses various web services
  - payments
  - messaging
  - email



What is a service?

- A service is
  - standalone
  - independently deployable software component
- It implements some useful functionality
- It has an API that provide access to its functionality
- It has two operations:
  - Command, query
- It can publish events
- It has its logical view architecture

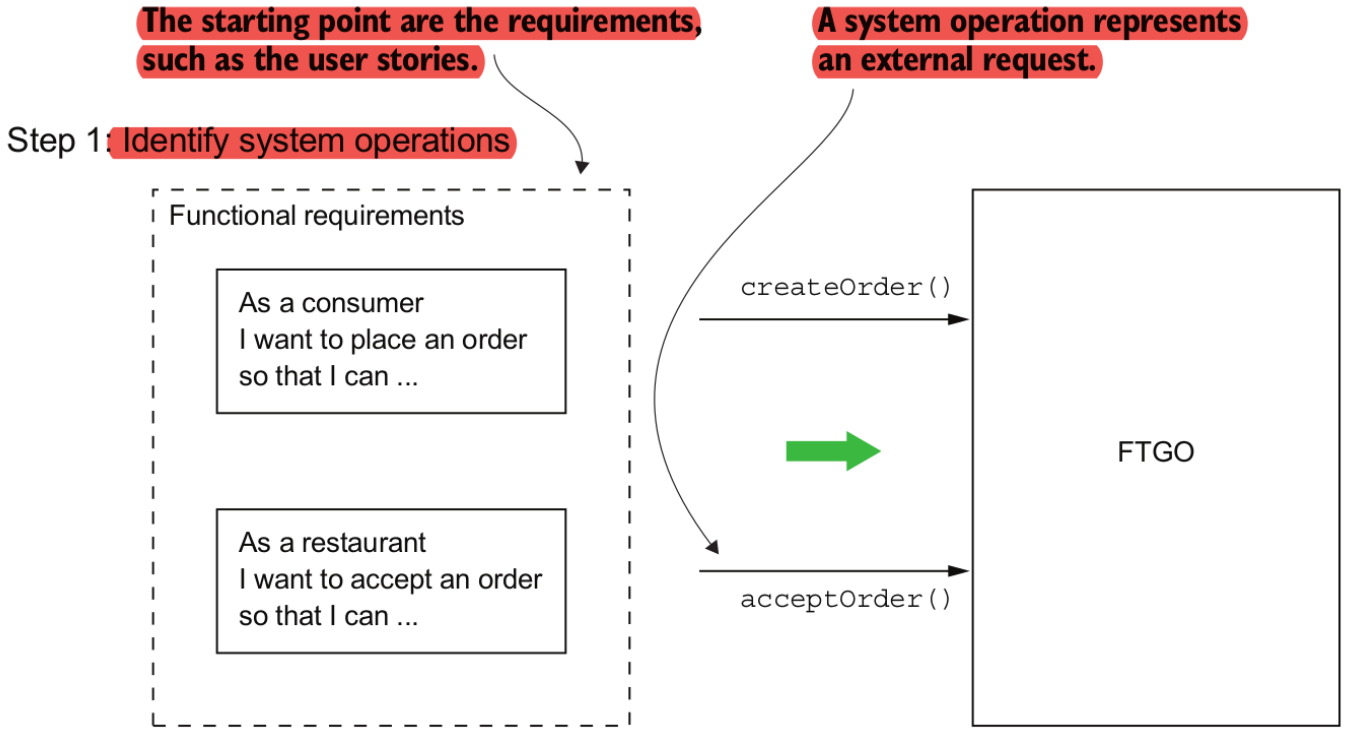


- Starting points are the written requirements
  - Interaction with domain experts
  - an existing application

- A three step process
  - Step 1: Identify system operations
  - Step 2: Identify services
  - Step 3: Define service APIs and collaborations

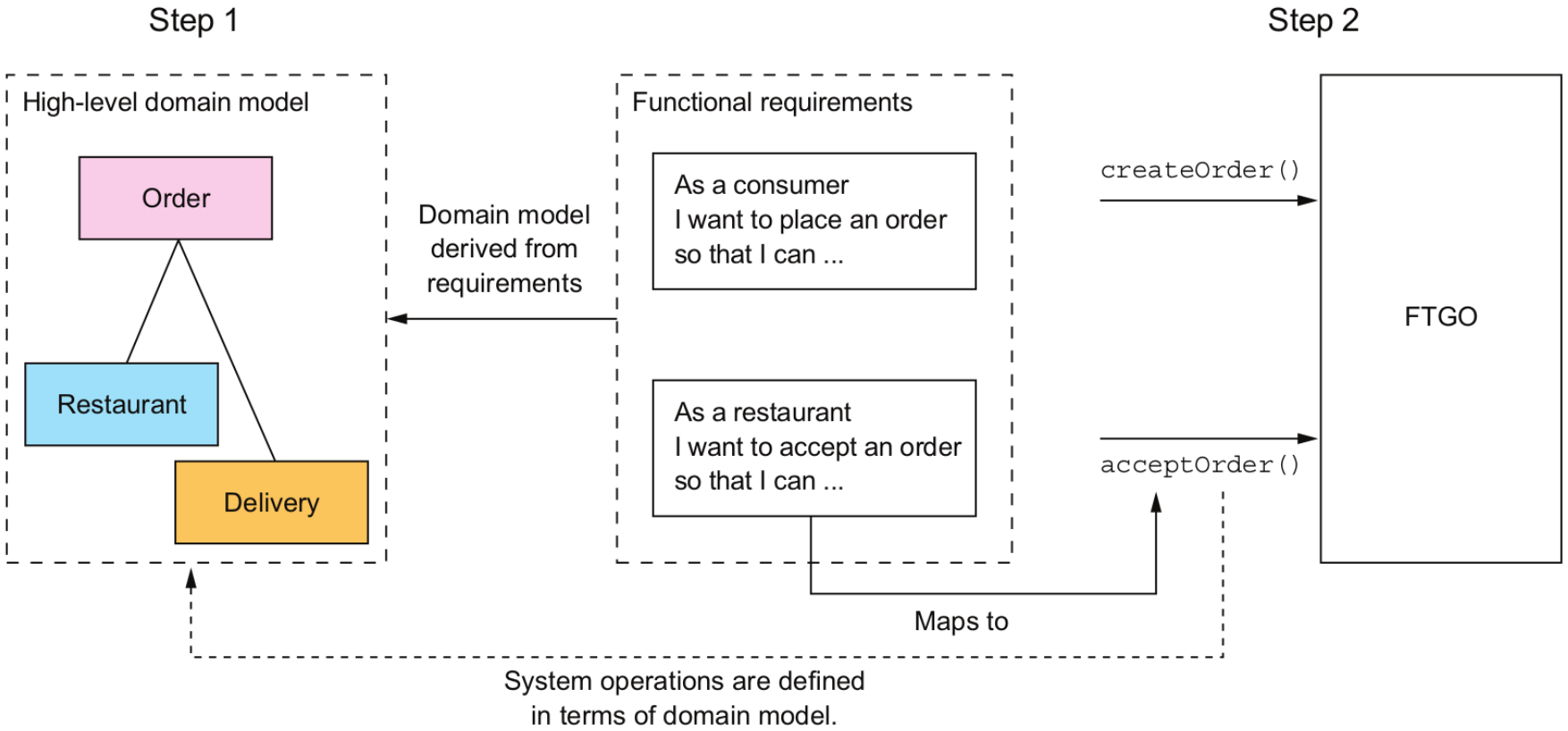
Identify system operations(1)

- Applications exist to handle requests
- System operations are abstraction of a request
  - Command: update data
  - Query: retrieve data
  - Their behavior is defined according to domain model



Identify system operations(2)

- Derive domain model
  - Classes as a vocabulary to describe the system operations
- Identify system operation



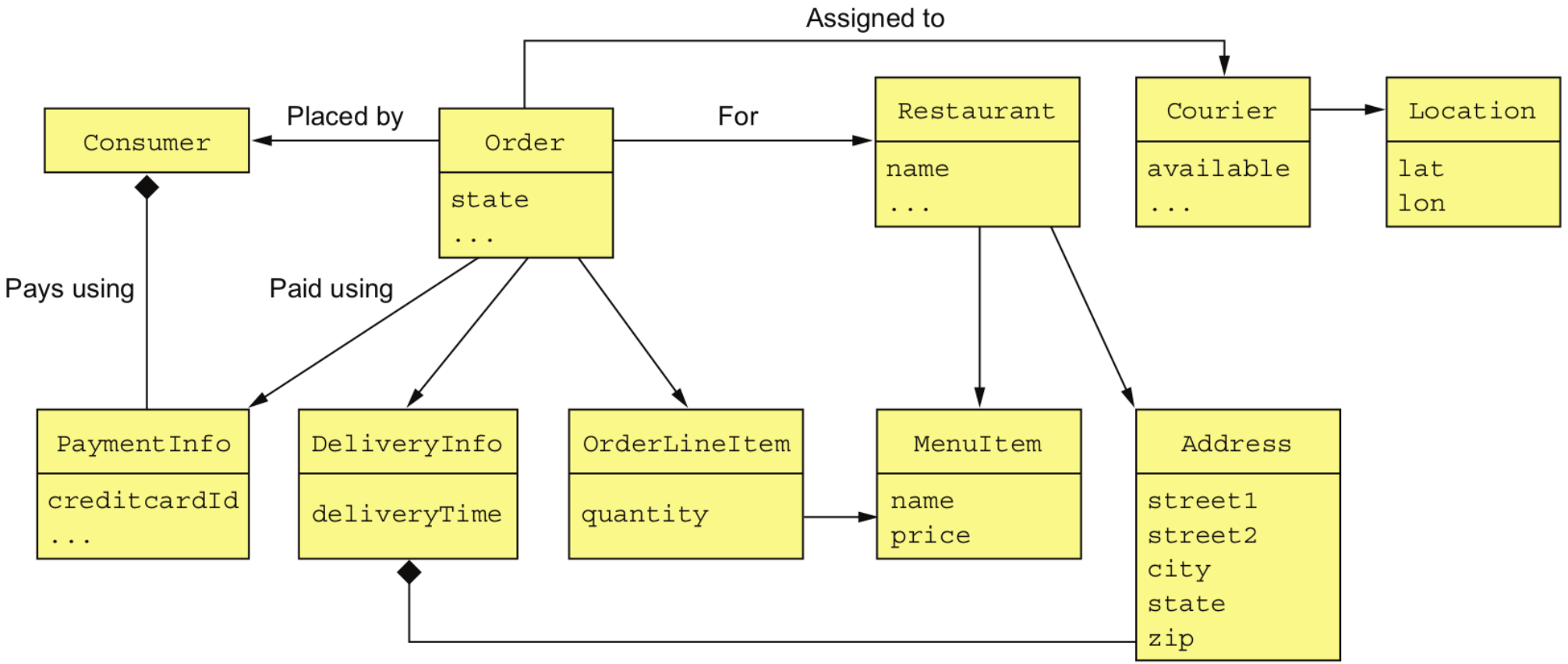
User scenario: example 1

- 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 order is created in the PENDING_ACCEPTANCE state
  - And the order is associated with the consumer
  - And the order is associated with the restaurant

User scenario: example 2

- 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

FTGO domain model



Class responsibilities

- Class responsibilities: what a class knows or does
- The responsibilities of each class are:
  - *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*

Defining system operations (1)

- Analyze the verbs in user stories and scenarios

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

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.

Defining system operations (2)

- Analyze the verbs in user stories and scenarios

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

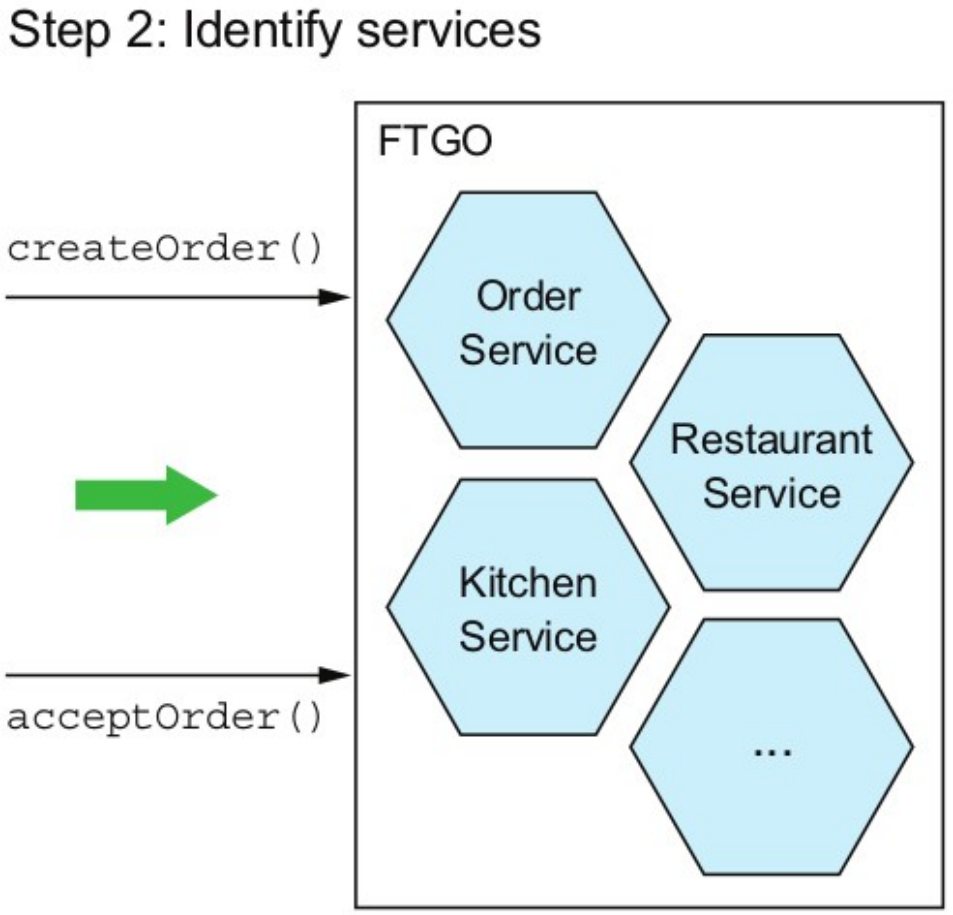
Operation	`acceptOrder(restaurantId, orderId, readyByTime)`
Returns	—
Preconditions	▪ The `order.status` is `PENDING_ACCEPTANCE`. ▪ A courier is available to deliver the order.
Post-conditions	▪ The `order.status` was changed to `ACCEPTED`. ▪ The `order.readyByTime` was changed to the `readyByTime`.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

Defining queries

- 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
- *findAvailableRestaurants(deliveryAddress, deliveryTime)*
  - Retrieves the restaurants that can deliver to the specified delivery address at the specified time
- findRestaurantMenu(id)
  - Retrieves information about a restaurant including the menu items

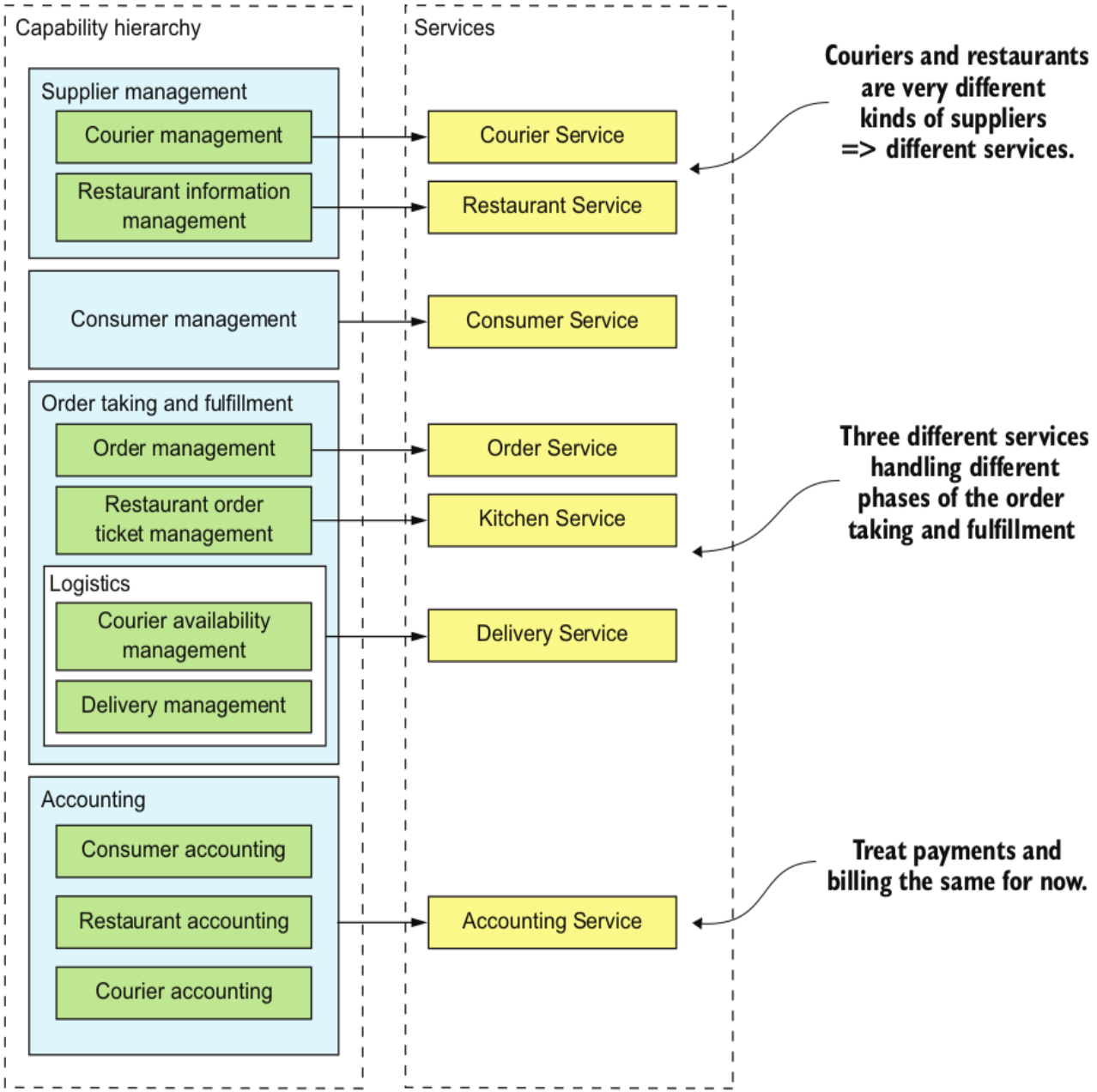
Decompose by business capability

- Identify services by business capabilities
- Business capability is
  - something that a business does to generate value
  - Order management, Inventory management…
- What organization's business is
  - not how is done
- It depends on the kind of business
  - Order management, Inventory management, Shipping, ...



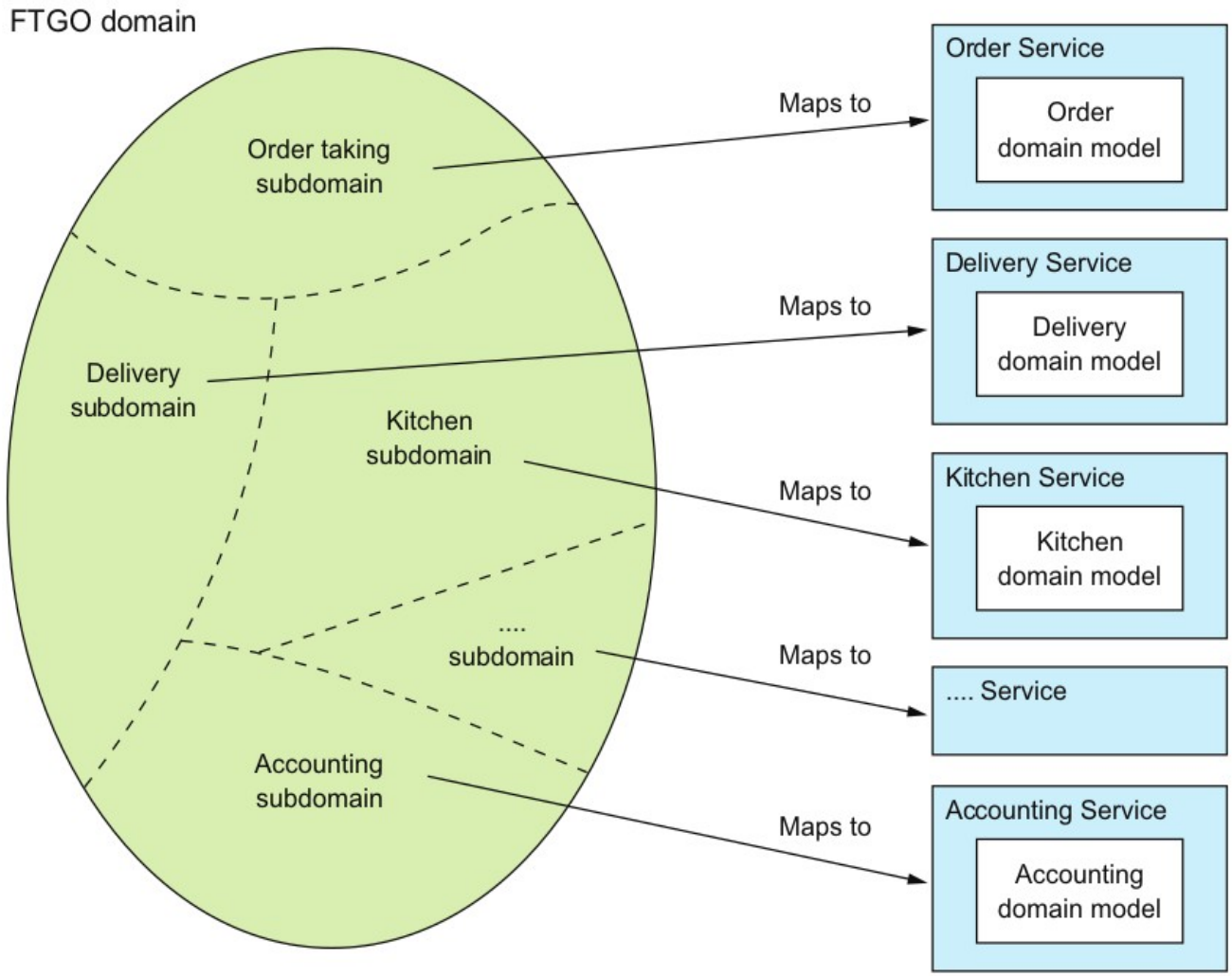
From businnes capabilities to services

- Define service for each (sub)capability or capabilities group



Decompose by sub-domain (DDD)

- A *domain model*
  - Knowledge about domain
- Subdomains are
  - different areas of expertise
  - Order taking, Order management, Kitchen management, Delivery...
- Very similar to businnes capabilities



- Single Responsibility Principle
  - *A class should have only one reason to change*

- Common Closure Principle
  - *The 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*

Operations-services mapping

- Map abstract system operations to services
  - Following the principles also for services

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

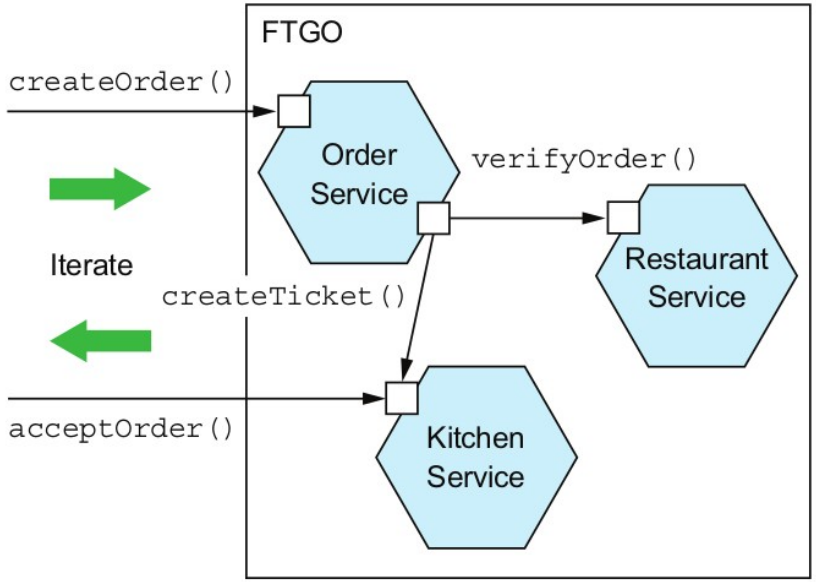
Multiple services collaboration

- System operations:
  - are handled by single service
  - span multiple services
- Example: *createOrder()* in *Order Service* depends on:
  - Consumer Service:
    - verify consumer can place an order and pay
  - Restaurant Service:
    - validate the order line items
    - verify that the delivery address/time
    - obtain price for the order line items.
  - Kitchen Service
    - create the Ticket .
  - Accounting Service
    - authorize the consumer’s credit card

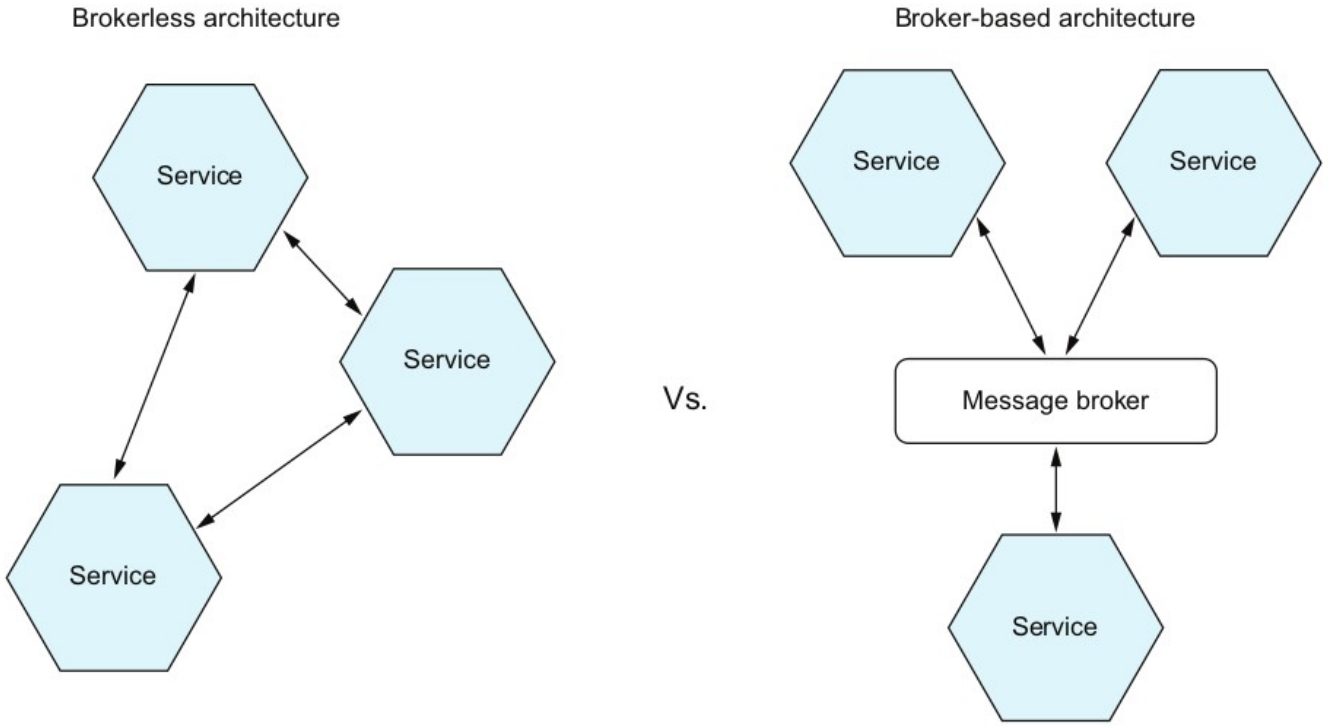
Identify collaborations

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()	—

- Identify service collaborations
- Define APIs between services
- Revise service bounds, if required



Choose communication paradigm



+ Lighter network traffic and better latency
+ No single point of failure
- Service discovery
- Reduced availability

+ Loose coupling
+ Message buffering
- Performance bottleneck
- Single point of failure
- Additional complexity