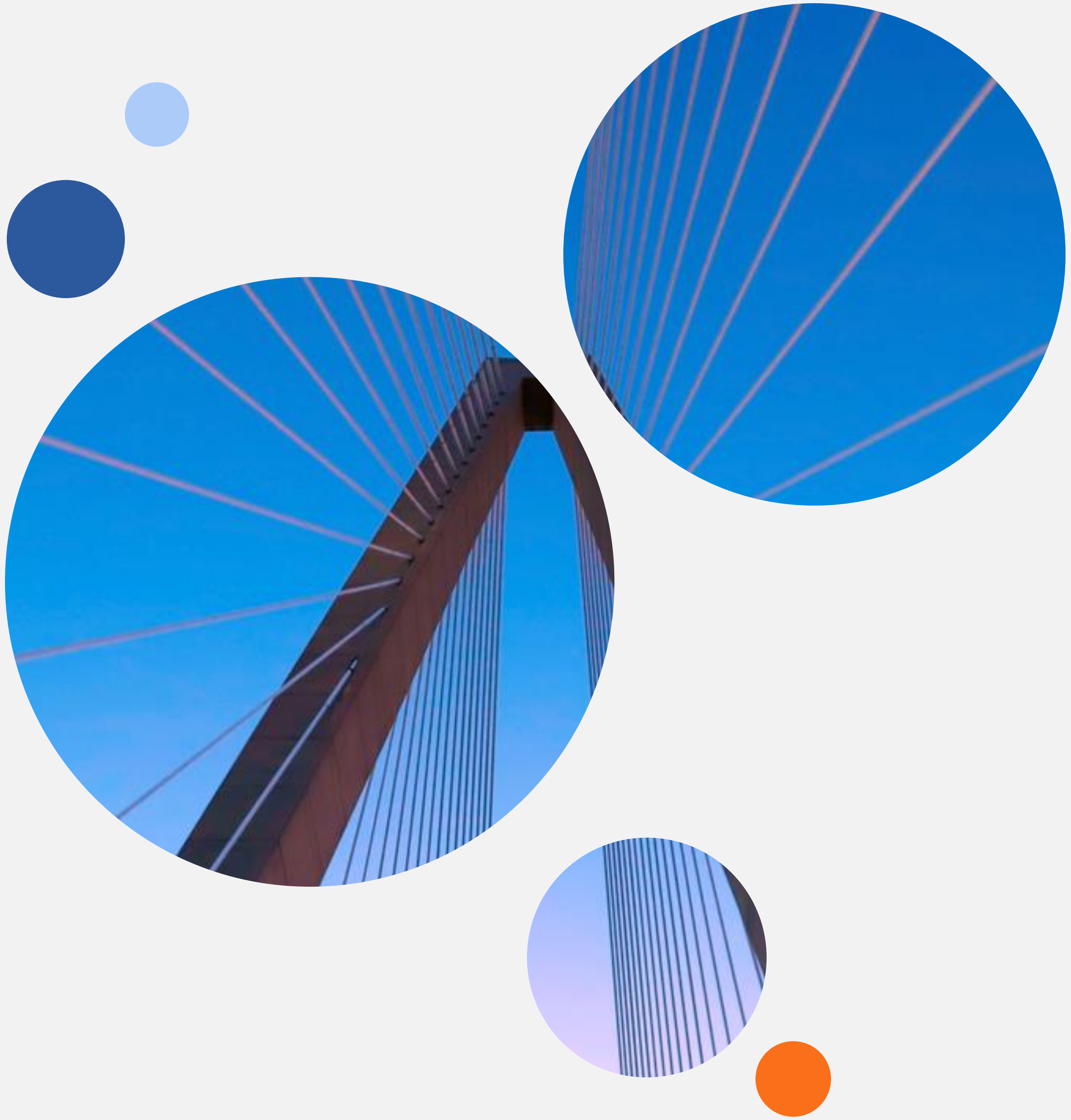
Microservices Architecture

By
Ahmed Abouzeid

Agenda

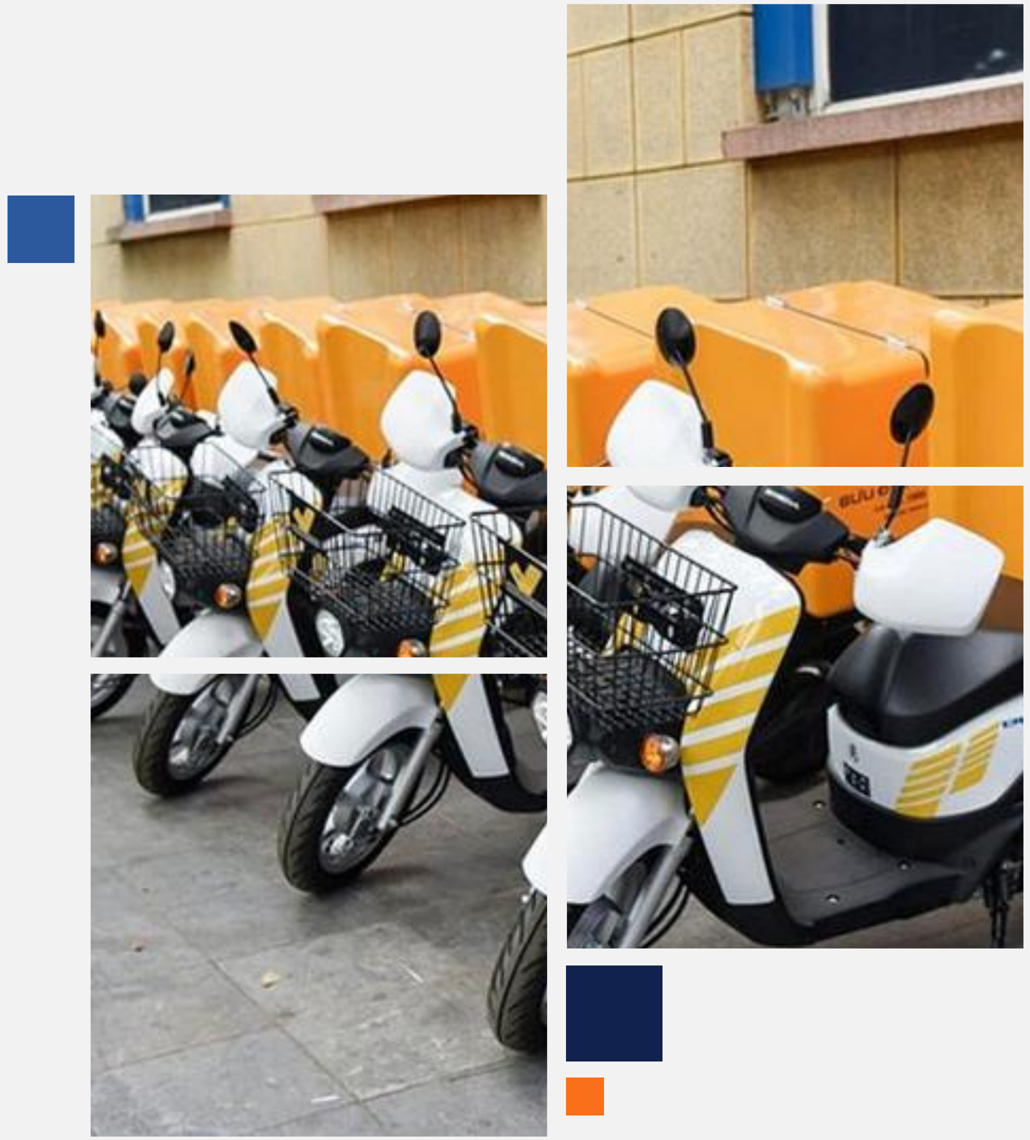
- Business Case
- Tackling the business case
- What is Microservices architecture?
- Benefits and Challenges
- Domain Modeling for Microservices
- Compute options
- Interservice communication
- API Gateway
- Data Management



Business Case

Delivery Service

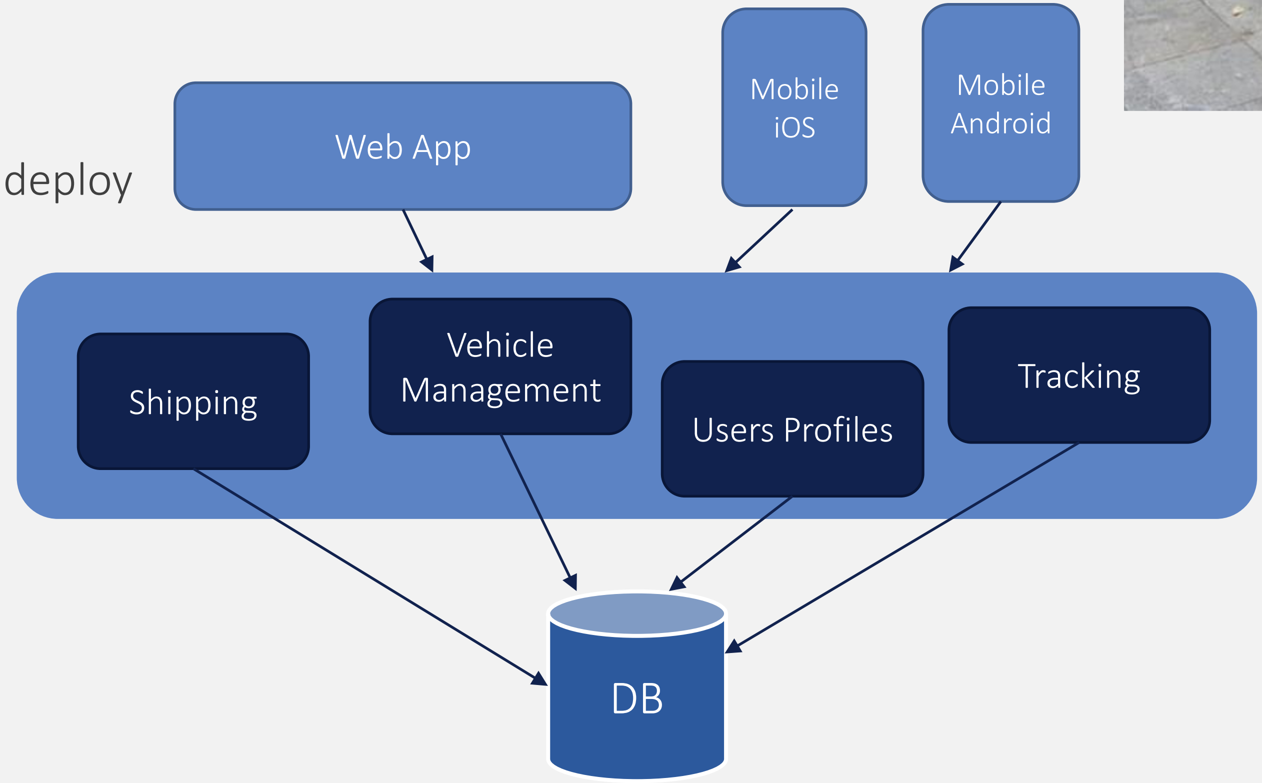
- The company manages a fleet of vehicles
- Businesses register with the service, and users can request a vehicle to pick up goods for delivery
- When a customer schedules a pickup, a backend system assigns a vehicle and notifies the user with an estimated delivery time
- The customer can track the location of the vehicle, with a continuously updated ETA

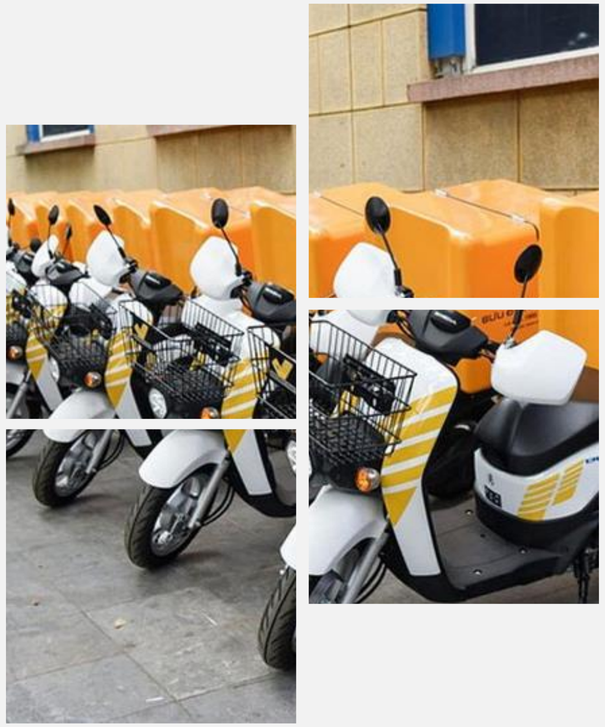


Tackling the business case

Monolithic Architecture

- The traditional unified model for the design of a software program.
- Multiple components are combined into one large application.
- Motivations:
  - Well Known
  - Easy to understand
  - Easy to develop, test and deploy



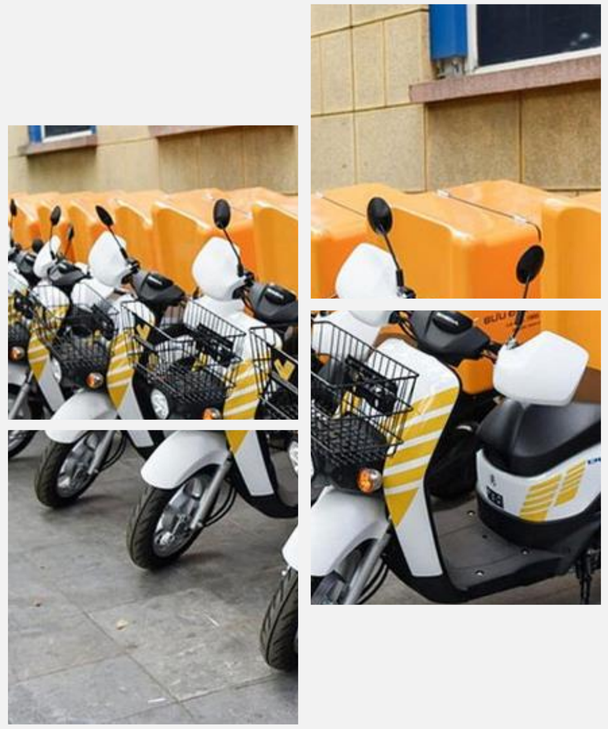


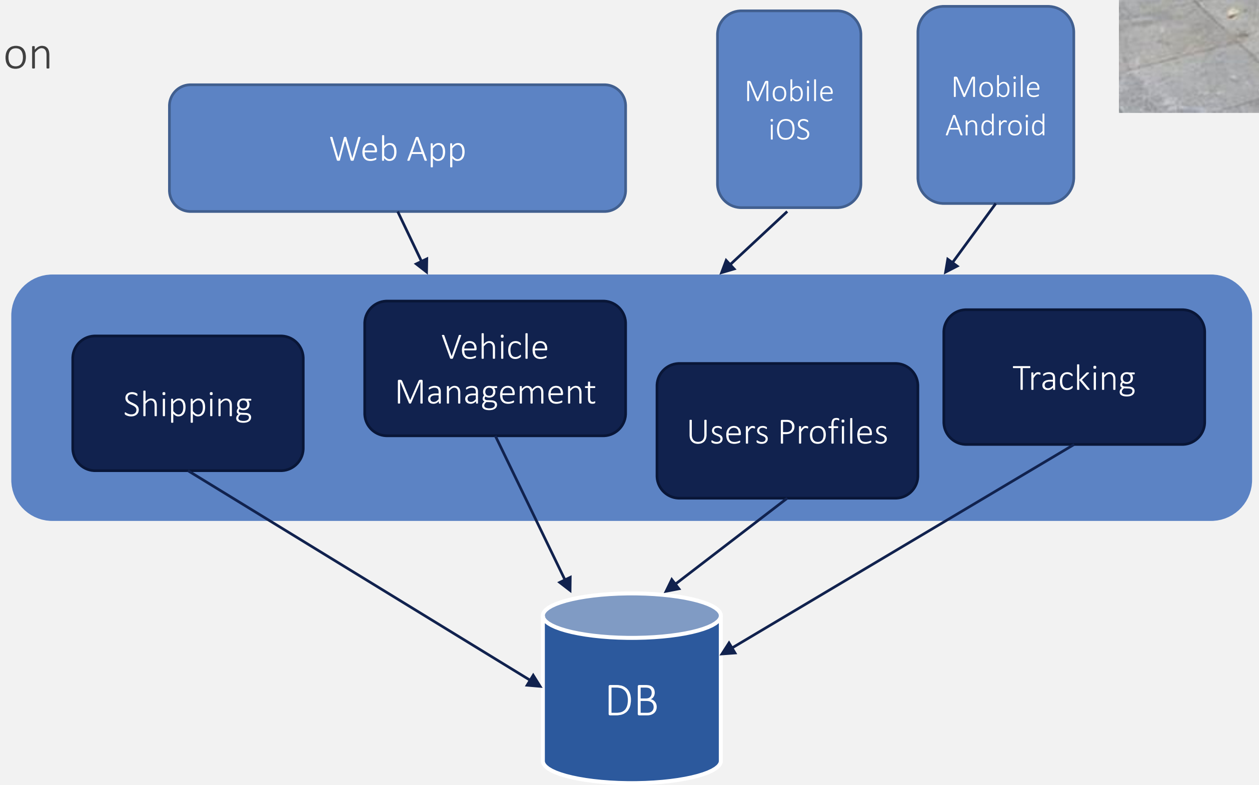
Tackling the business case

Monolithic Architecture

Downsides

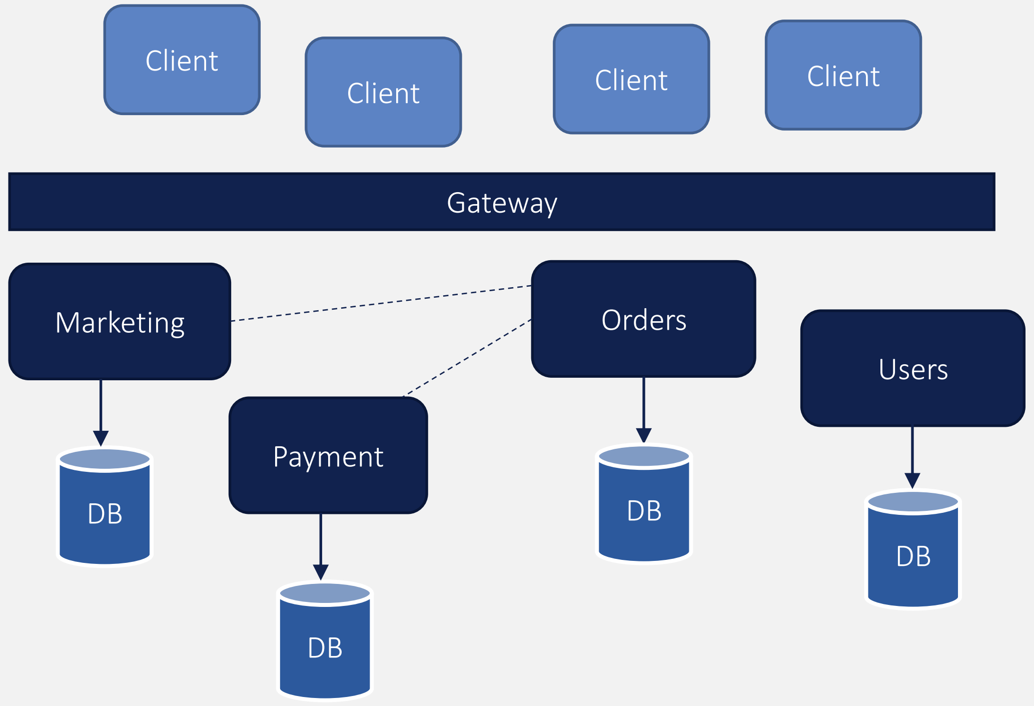
- Hard to maintain the modularity
- Bigger == more complex
- Scale out the whole application
- Fault tolerance
- Update to new technology
- Multiple teams





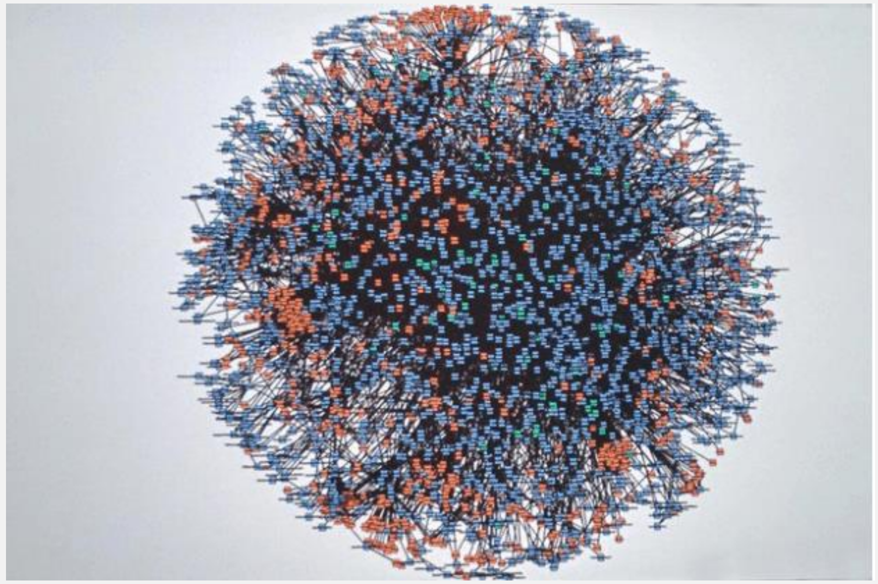
What is Microservices architecture?

- Microservices are small, independent, and loosely coupled
- Each service is a separate codebase, which can be managed by a small development team
- Services can be deployed independently
- Services are responsible for persisting their own data or external state
- Services communicate with each other by using well-defined APIs
- Services communicate with each other by using well-defined APIs



Microservices Examples

- Amazon

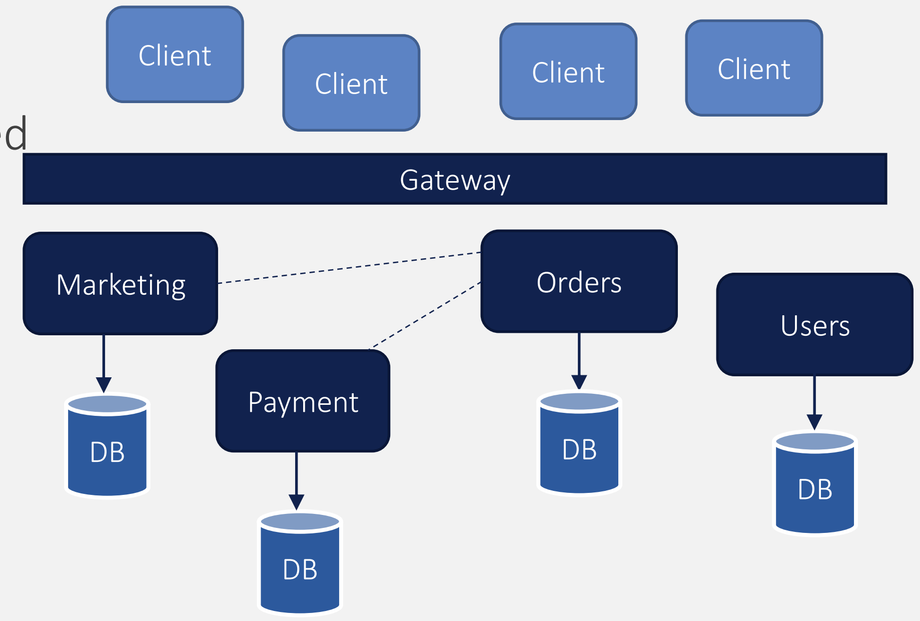

This is a 2008 graphic of Amazon’s microservices infrastructure

- **Netflix**: 110 million subscribers, 1 billion hours of video each week, 5 million new subscribers per quarter. Uses over 700 loosely coupled microservices
- **Uber**: 1300 microservices

Benefits and Challenges

Benefits

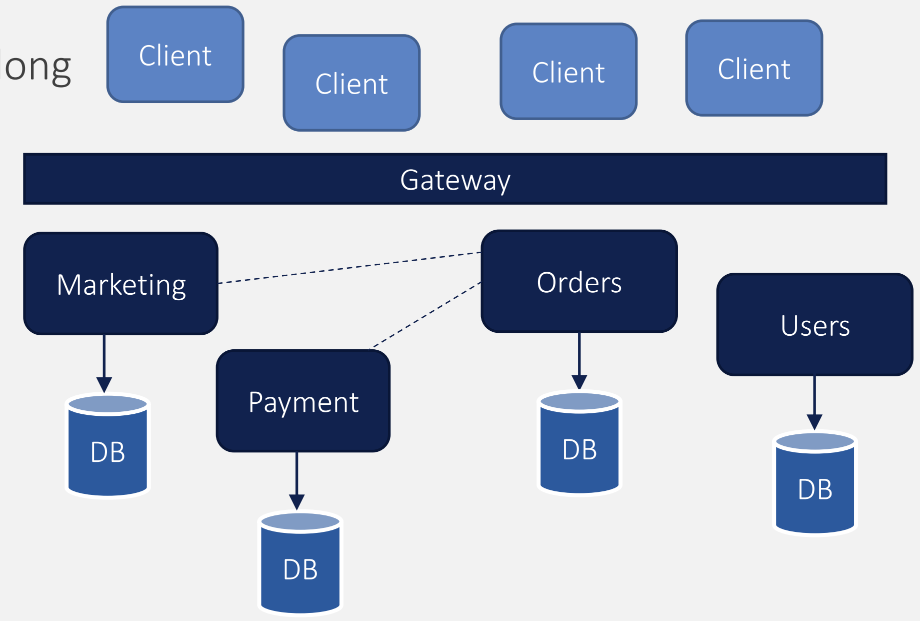
- **Small, focused teams**: A microservice should be small enough that a single feature team can build, test, and deploy it.
- **Agility**: easier to manage bug fixes and feature releases.
- **Small code base**
- **Mix of technologies**
- **Fault isolation:** as long as any upstream microservices are designed to handle faults correctly
- **Scalability:** scale out subsystems that require more resources, without scaling out the entire application
- **Data isolation:** easier to perform schema updates



Benefits and Challenges

Challenges

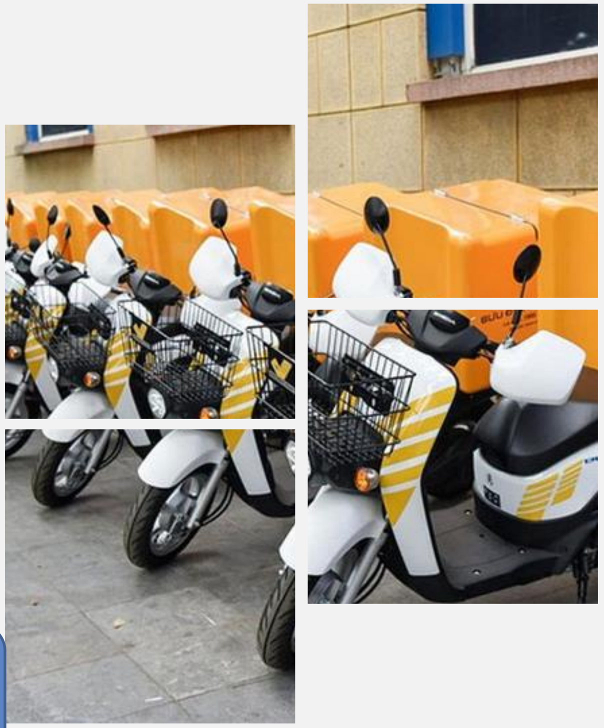
- **Complexity**: Each service is simpler, but the entire system as a whole is more complex.
- **Development and testing**: rely on other dependent services requires a different approach, refactoring across services, integration tests
- **Lack of governance:** many different languages and frameworks. It may be useful to put some project-wide standards
- **Network congestion and latency:** chain of service dependencies gets too long (service A calls B, which calls C...), avoid overly chatty APIs
- **Data integrity:** data consistency can be a challenge
- **Management:** requires a mature DevOps culture
  Correlated logging across services
- **Versioning:** Updates to a service must not break services
- **Skill set**

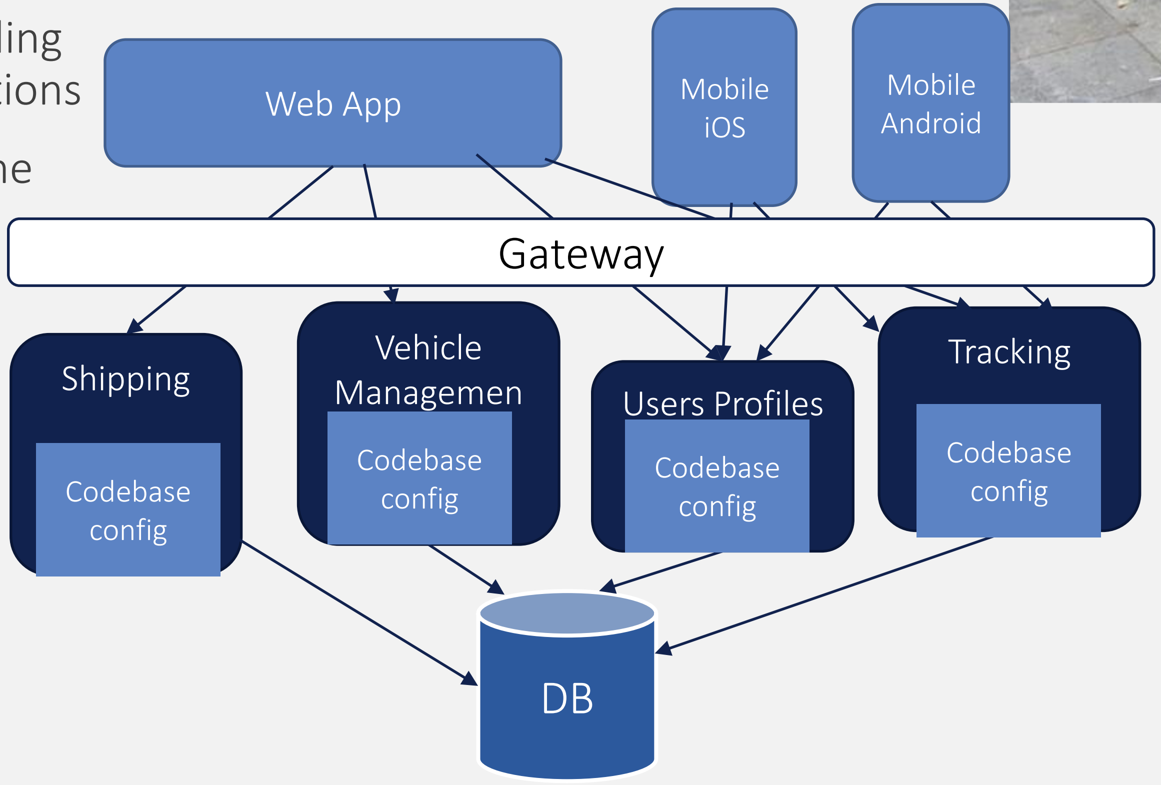


Tackling the business case (cont.)

Distributed Monolithic Architecture

- Independent services
- Shared database
- Must be deployed and released at once
- Bundled into single package including codebase , libraries and configurations
- Introduce API Gateway to overcome clients' maintenance



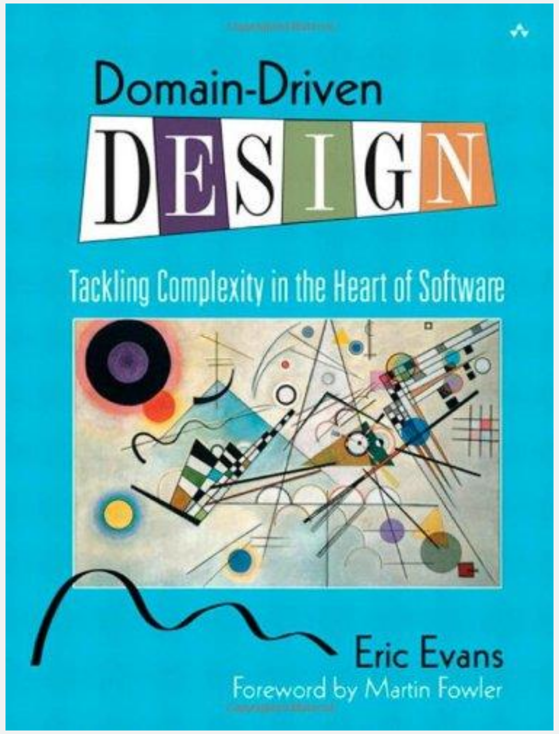


Domain Modeling for Microservices

Software Complexity Types

- Technical complexity
- Amount of data
- Performance
- Business logic complexity $\Rightarrow$ DDD

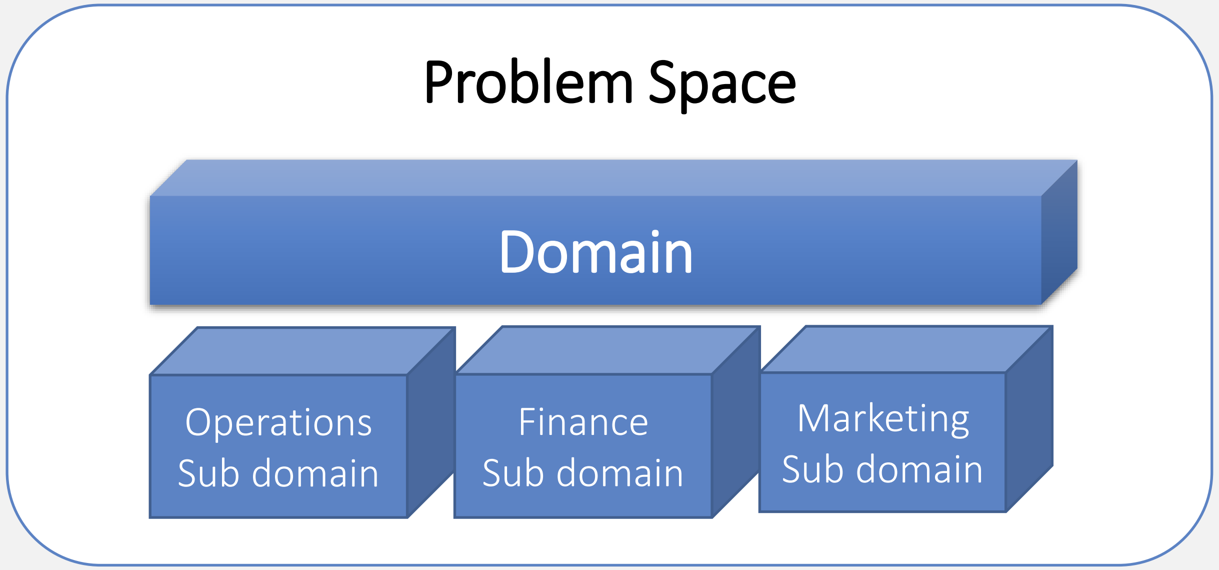
Domain Driven Design can handle complex business logic complexity in such away that it would be possible to extend, maintained and keep it simple


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler

Domain Modeling for Microservices

Domain

Field of study that defines a set of common requirements, terminology, and functionality for any software program



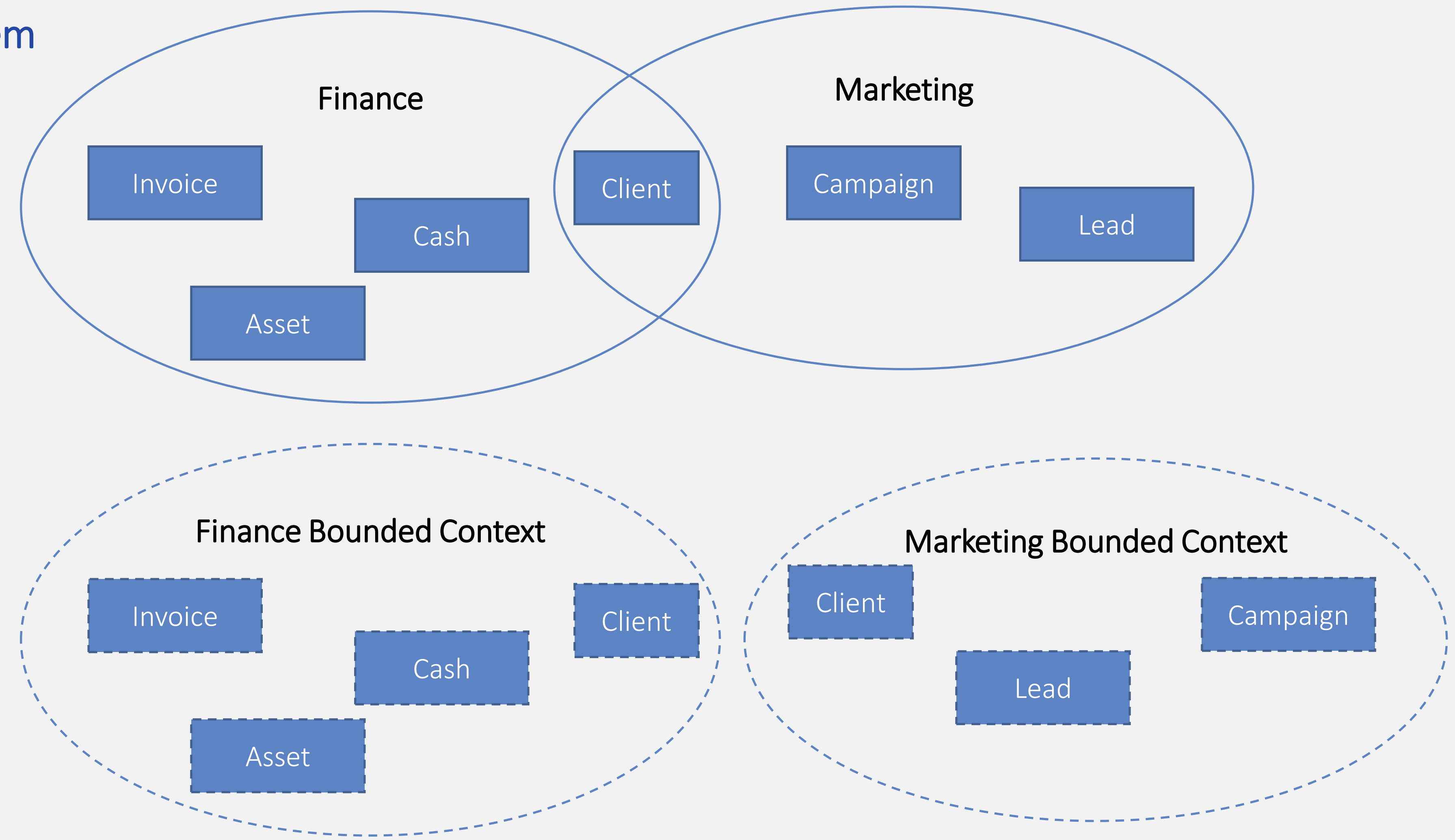
Ubiquitous Language

Domain Modeling for Microservices

Modeling the Problem

Domain

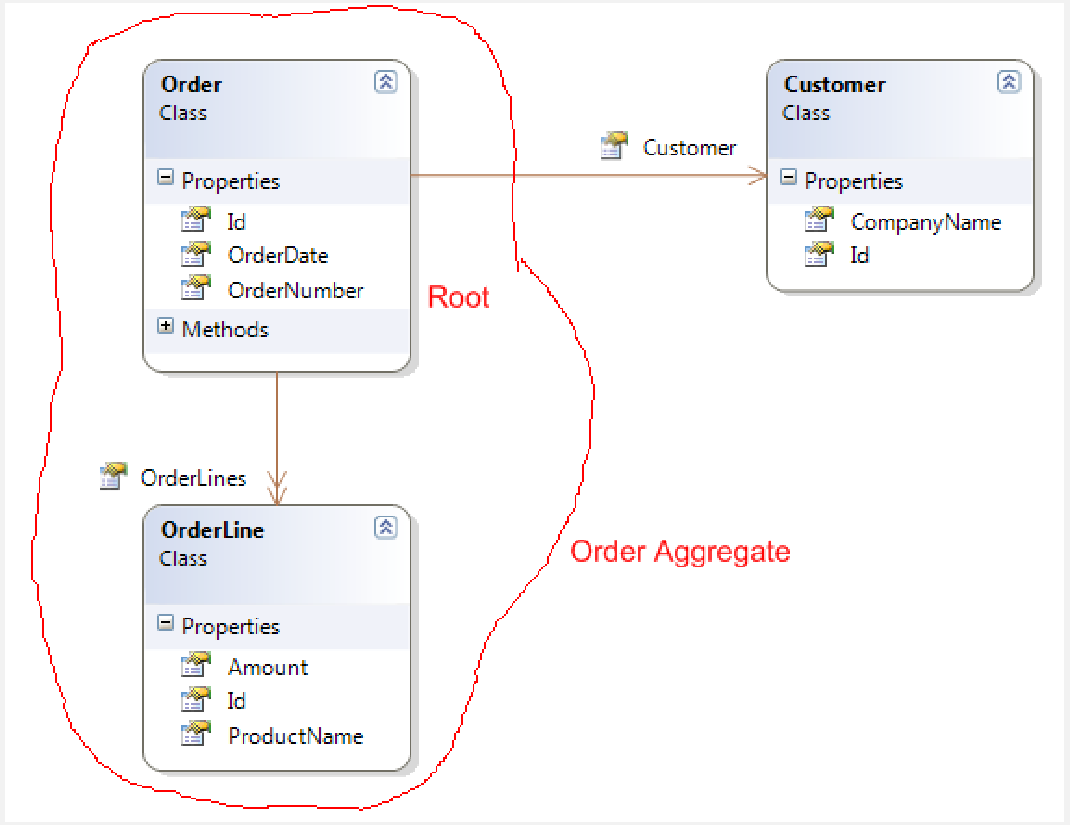
Bounded Context



Domain Modeling for Microservices

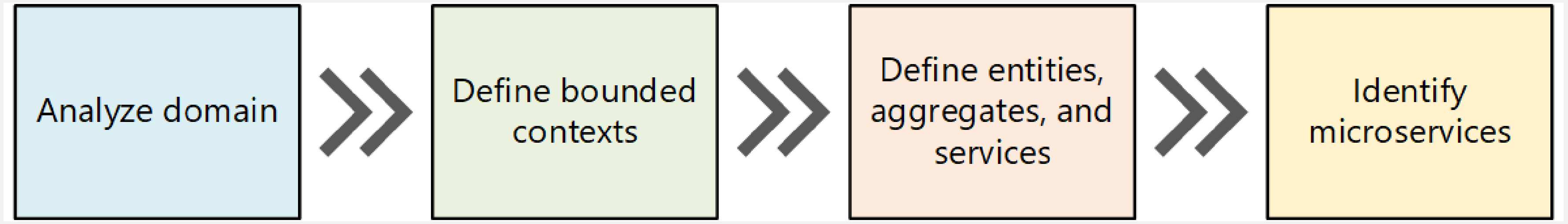
Aggregate

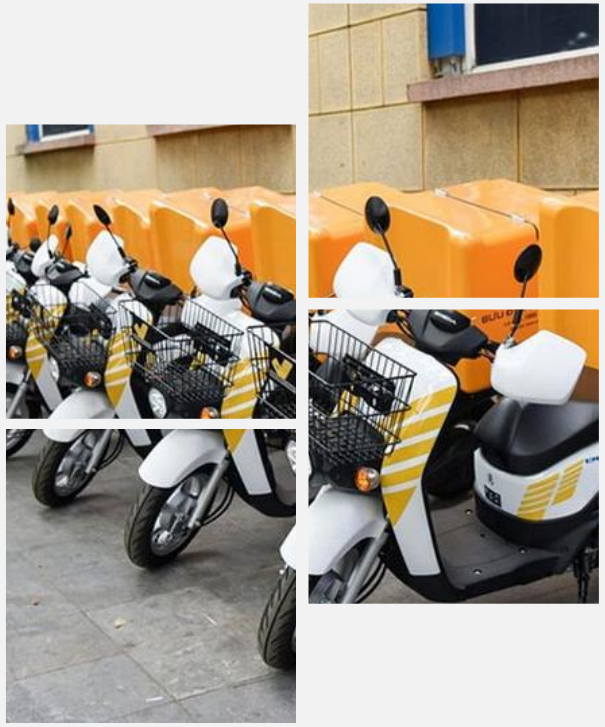
A DDD aggregate is a cluster of domain objects that can be treated as a single unit



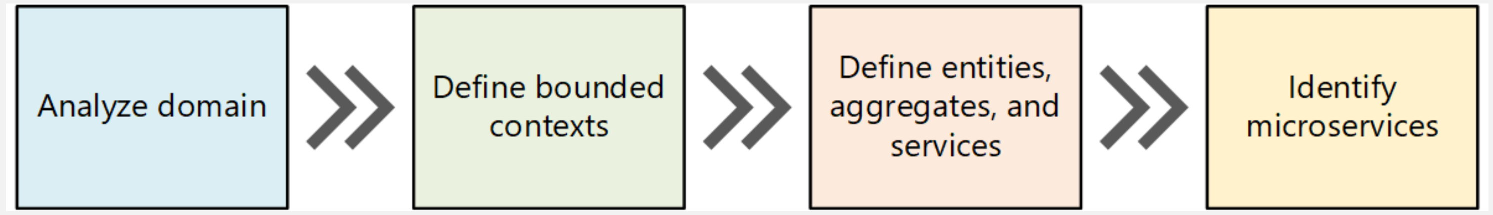
Domain Modeling for Microservices

Domain-driven design (DDD) provides a framework that can get you most of the way to a set of well-designed microservices

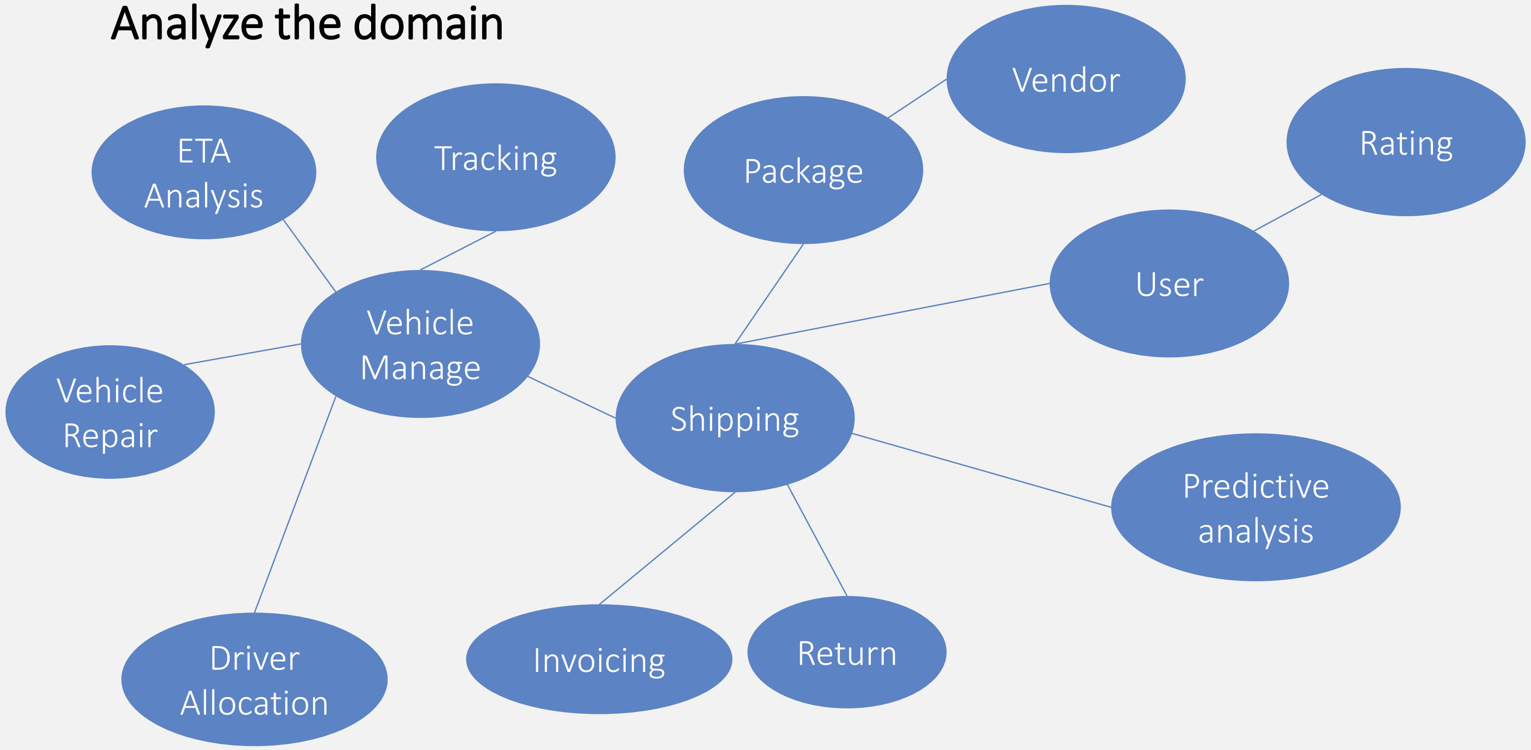


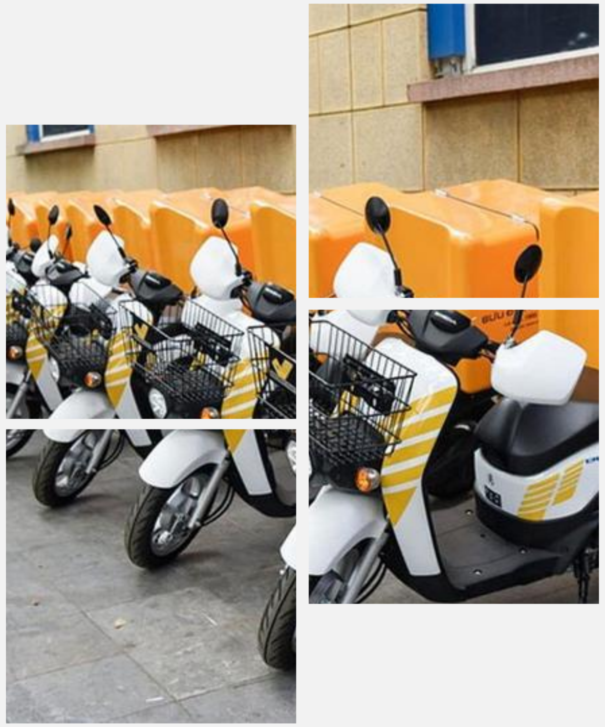


Domain Modeling for Microservices

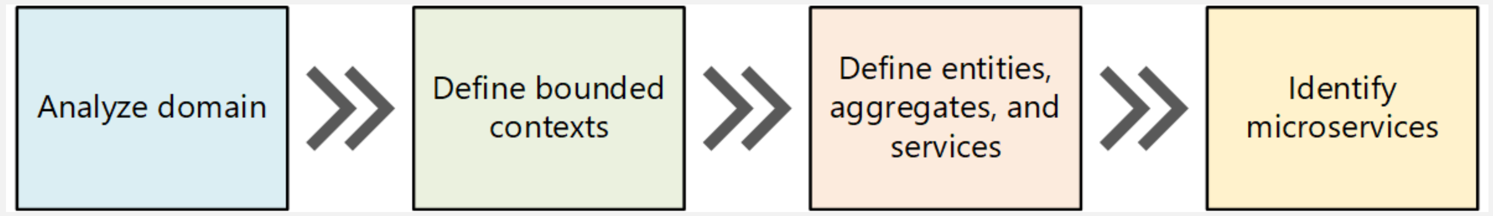


Analyze the domain

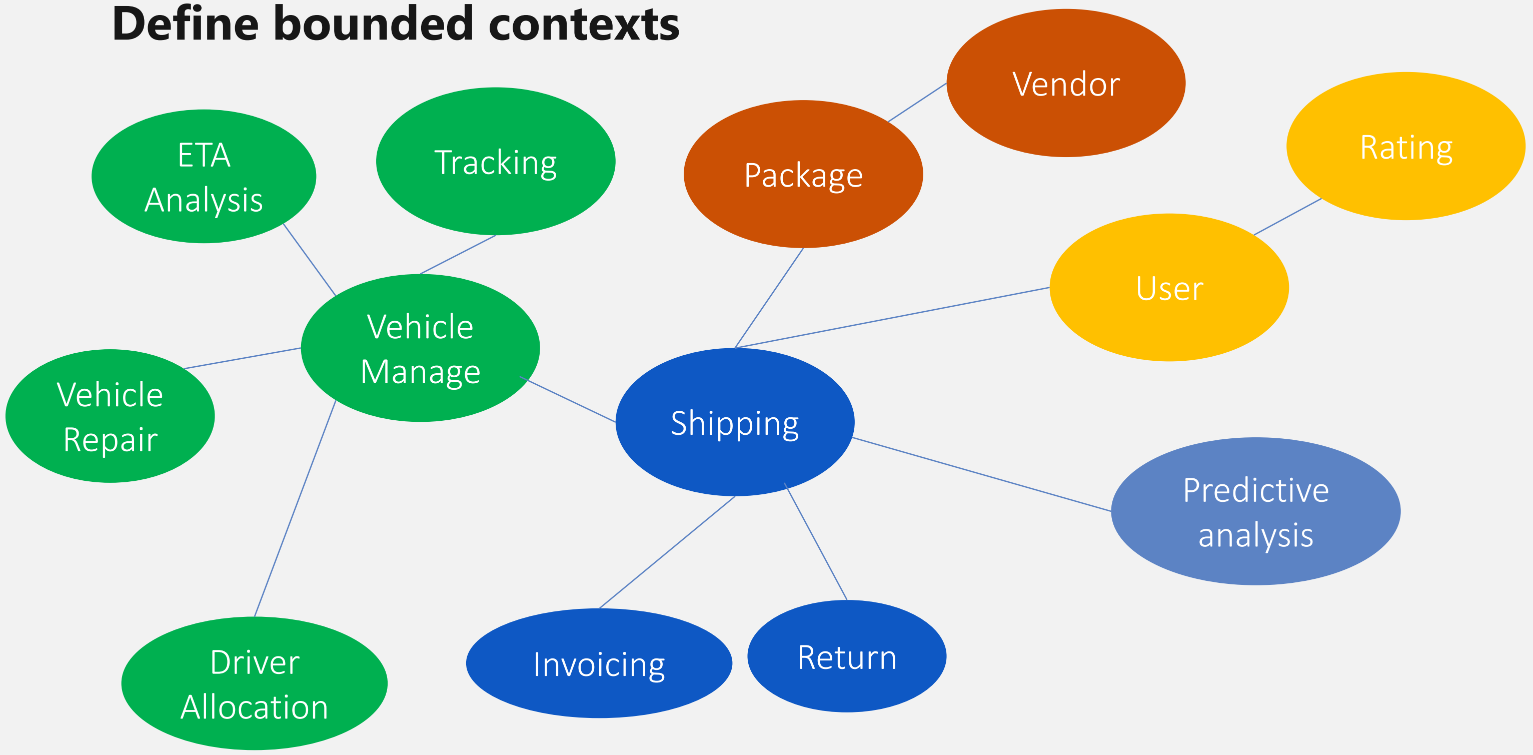


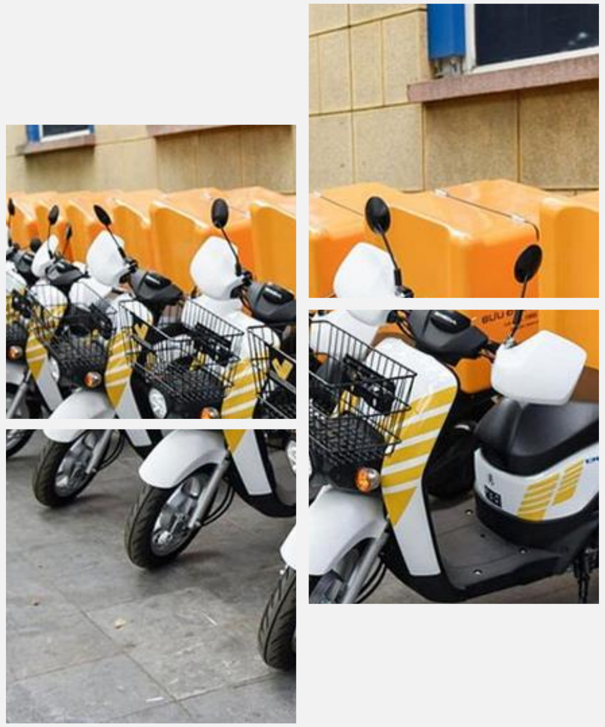


Domain Modeling for Microservices

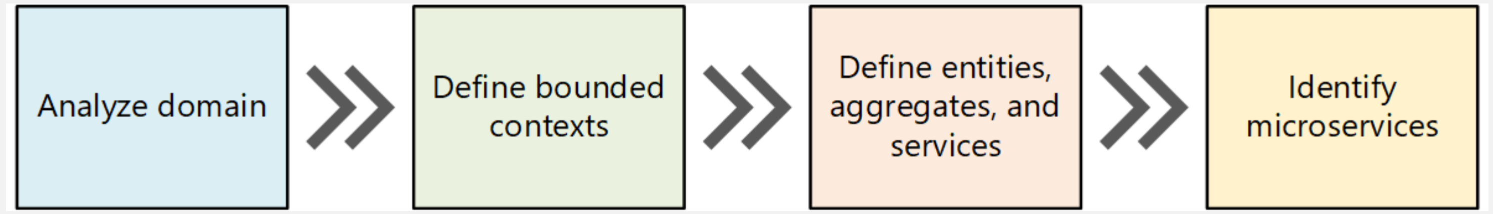


Define bounded contexts



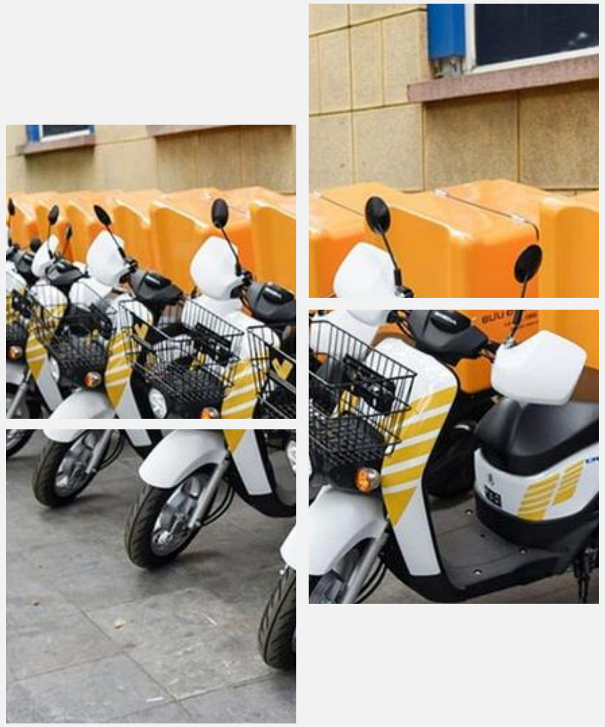


Domain Modeling for Microservices

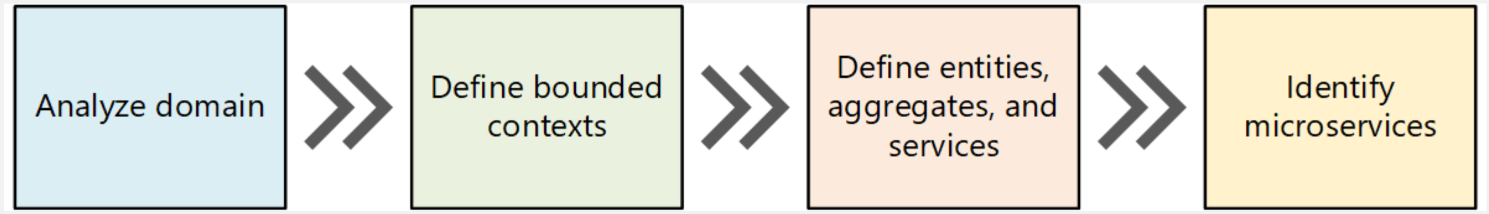


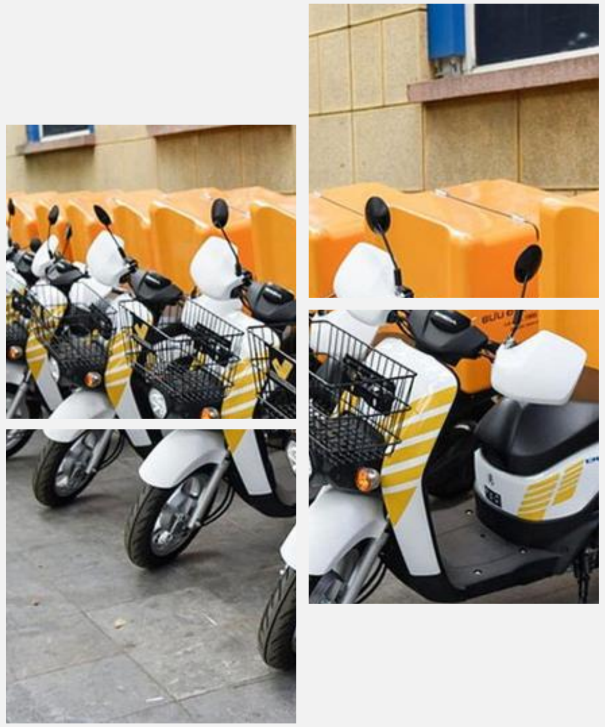
Define entities, aggregates, and services

- A customer can request a vehicle to pick up goods.
- The sender generates a tag (i.e. barcode) to put on the package.
- Pick up and deliver a package from the source location to the destination location
- The customer is notified when the delivery is completed.
- The sender can request delivery confirmation from the customer, in the form of a signature or finger print.



Domain Modeling for Microservices

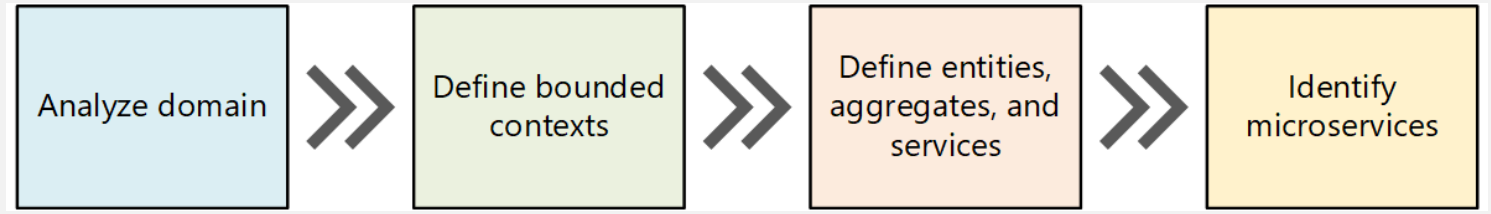




Define entities, aggregates, and services

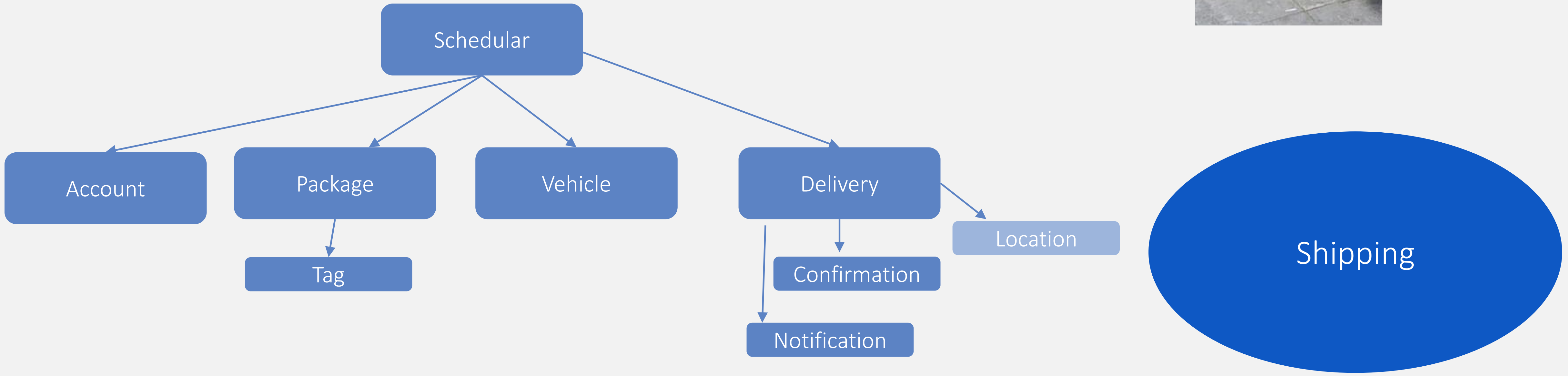
- Entities:
  - Delivery
  - Package
  - Vehicle
  - Account
  - Confirmation
  - Notification
  - Tag
- Aggregates: Delivery, Package, Vehicle, and Account

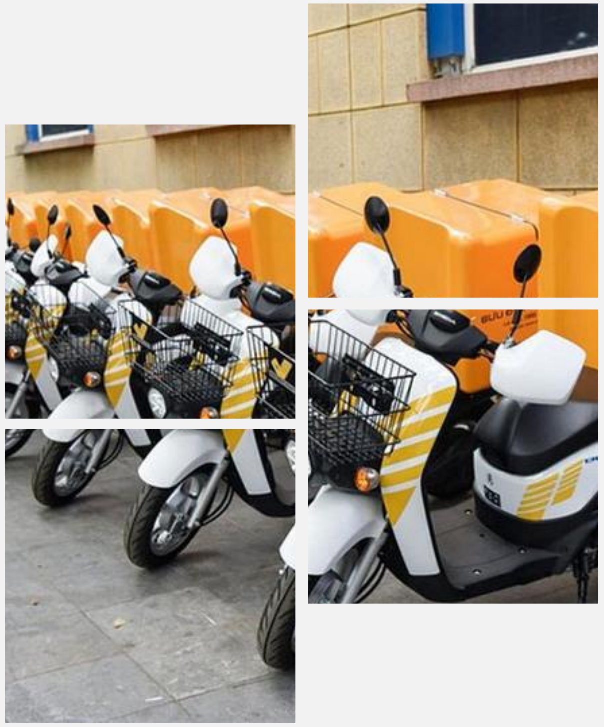
Domain Modeling for Microservices



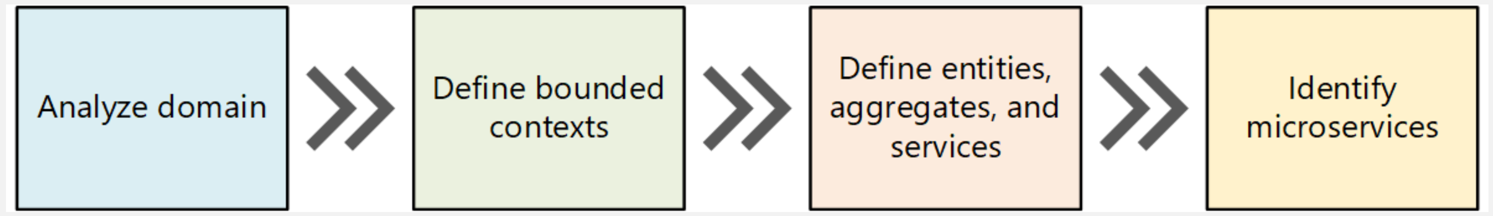
Define entities, aggregates, and services

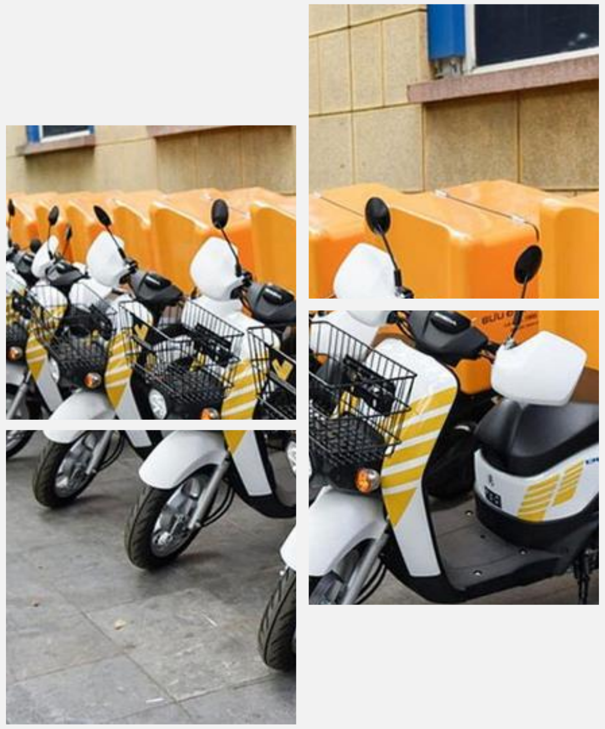
- **Domain Service**: Schedular





Domain Modeling for Microservices

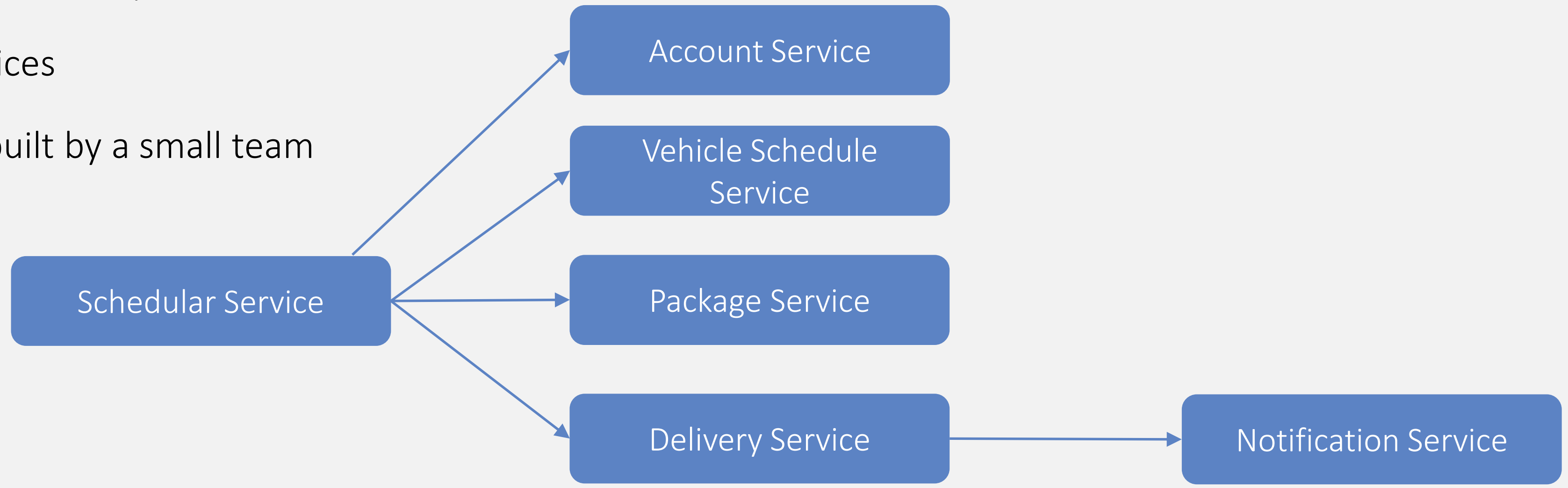




Identify microservices

"not too big and not too small"

- Each service has a single responsibility.
- No chatty calls between services
- Small enough that it can be built by a small team



Compute Options

- The hosting model for the computing resources that your application runs on. Two popular approaches:

  1. **Service orchestrator**: manages services running on dedicated nodes (VMs)
     - Placing services on nodes
     - Monitoring the health of services
     - Restarting unhealthy services
     - Load balancing network traffic across service instances
     - Service discovery
     - Service scaling
     - Applying configuration updates
  - Azure options:
     - **Azure Kubernetes Service (AKS):** Kubernetes APIs as a service
     - **Azure Container Apps:** built on Kubernetes that abstracts the complexities of orchestration
     - **Service Fabric**
     - **Others**: DC/OS, Docker Enterprise Edition and Docker Swarm

Compute Options

2. Serverless (Functions as a Service):

- **Azure Functions:** a serverless solution that allows you to write less code, maintain less infrastructure, and save on costs.
  With the Consumption plan, you only pay while your functions are running

Which one ?

- **Manageability:**
  serverless, is easy to manage
  orchestrator, need to think about issues such as load balancing, CPU and memory usage, and networking
- **Portability**:
  Kubernetes, Docker, and Service Fabric can run on-premises or in multiple public clouds.
- **Cost**:
  serverless, you pay only for the actual compute resources consumed
  orchestrator, you pay for the VMs that are running in the cluster.
- **Scalability:**
  Azure Functions scales automatically to meet demand
  orchestrator, can increase the number of service instances in the cluster or adding additional VMs to the cluster

Compute Options

Demo

Working with Service Fabric

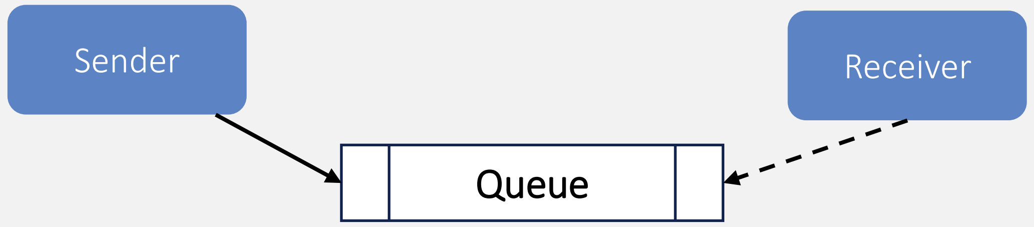
https://learn.microsoft.com/en-us/azure/service-fabric/service-fabric-get-started

Interservice Communication

Two messaging patterns:

- Synchronous communication
  - The caller waits for a response from the receiver
  - HTTP / gRPC

- Asynchronous message passing
  - A service sends message without waiting for a response
  - One or more services process the message asynchronously
  - Azure Storage Queues, Azure Service Bus



Benefits	Challenges
Reduce coupling	Coupling with the messaging infrastructure
Multiple subscription	Latency
Failure isolation	Complexity
Responsiveness	Throughput: could be a bottleneck
Load leveling	Cost

Interservice Communication

Demo

Working with Azure Service Bus

Interservice Communication (Cont.)

Challenges

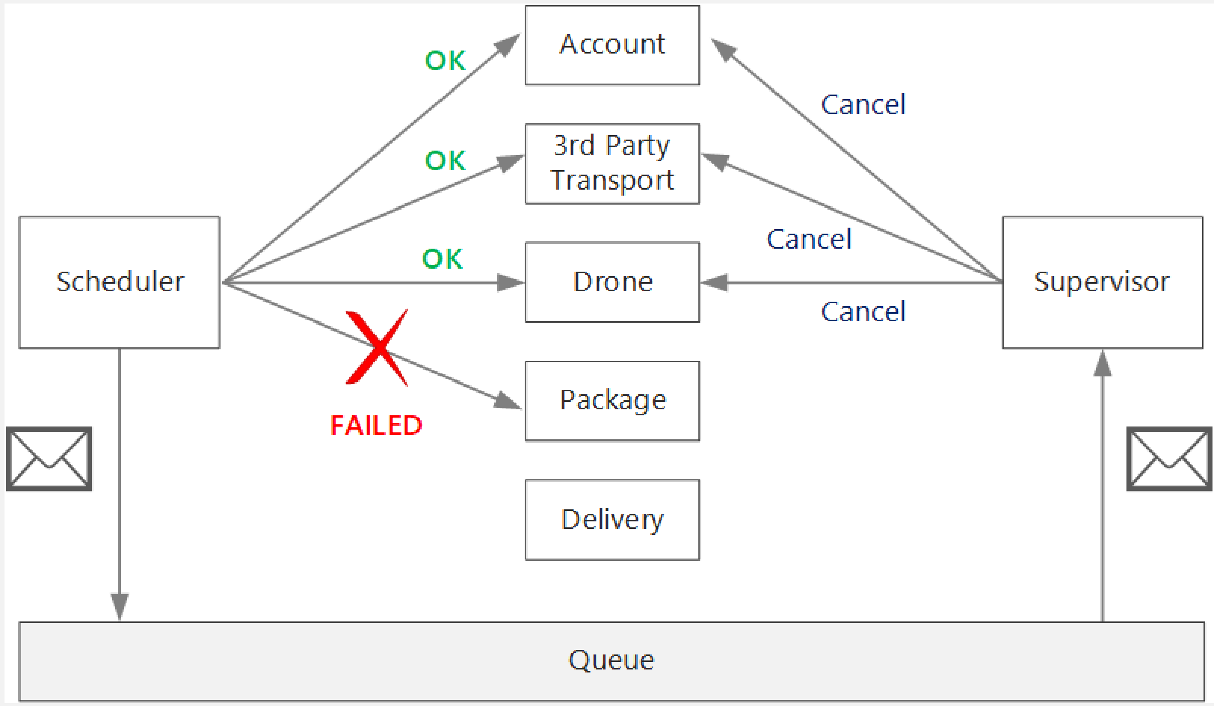
- **Resiliency**: An instance can fail for any number of reasons. Two design pattern that can help:
  - Retry: Mitigate the transient failure. Caller should retry in certain number of times or elapsed time-out.
  - Circuit Breaker: Too many failed requests can cause a bottleneck which can might hold critical system resources such as memory, threads, database connections.

    It has three stages: Closed, Half-Open and Open
- **Load balancing**: the request must reach a running instance of service.
- **Distributed tracing**: request correlation is a must
- **Service versioning**: Avoid breaking changes

Interservice Communication (Cont.)

Distributed transactions

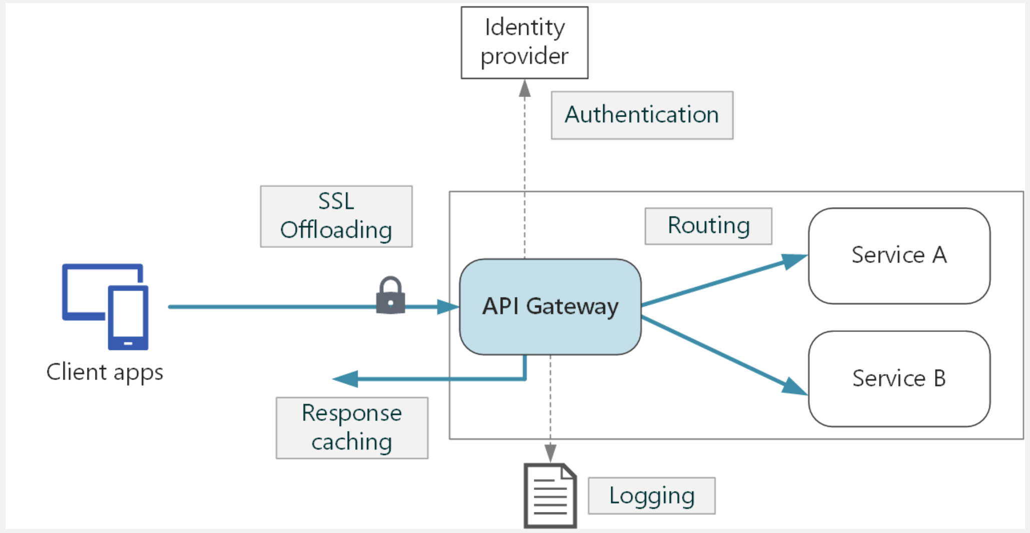
- A nontransient failure is any failure that's unlikely to go away by itself
- The entire business transaction must be marked as a failure.
- the application needs to undo the steps that succeeded, by using a **Compensating Transaction**
- Using Scheduler Agent Supervisor pattern.
- Another approach:
  save a checkpoint to a durable store after each step
- Idempotent opterations



API Gateway

Some potential problems with exposing services directly to clients:

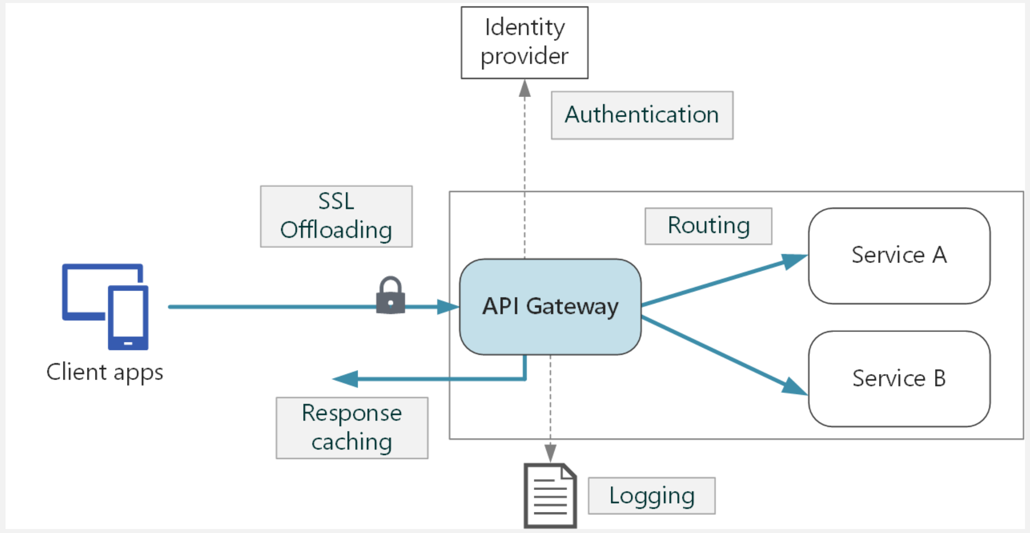
- It can result in complex client code
- It creates coupling between the client and the backend
- A single operation might require calls to multiple services
- Each public-facing service must handle concerns such as authentication, SSL, and client rate limiting
- Services must expose a client-friendly protocol such as HTTP or WebSocket
- Services with public endpoints are a potential attack surface



API Gateway (cont.)

What's API Gateway ?

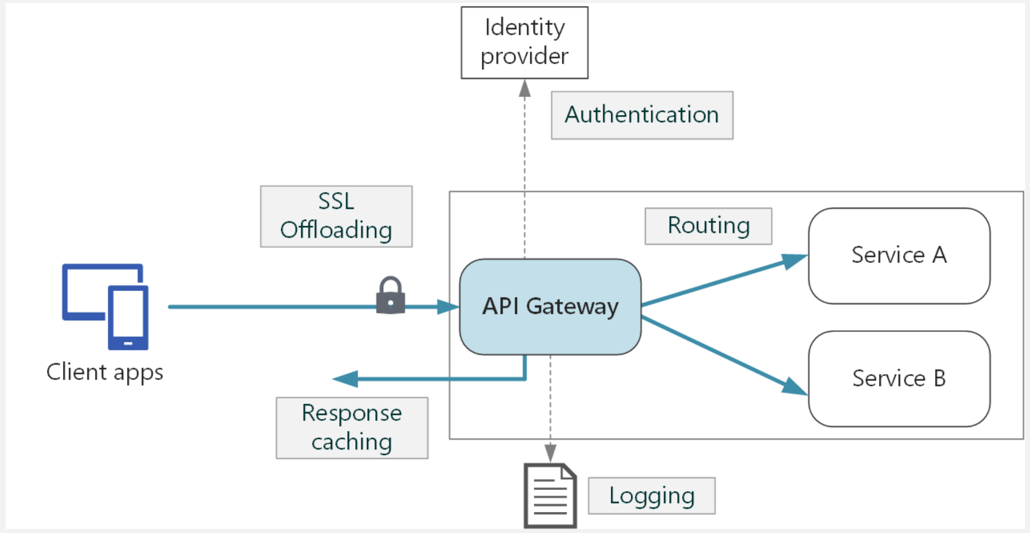
- An API gateway sits between clients and services. It acts as a reverse proxy, routing requests from clients to services.
- **Gateway Routing**: Use the gateway as a reverse proxy to route requests to one or more backend services.
- **Gateway Aggregation:** Use the gateway to aggregate multiple individual requests into a single request.
- **Gateway Offloading:** Use the gateway to offload functionality from individual services to the gateway like:
  - SSL termination
  - Authentication
  - IP allow/block list
  - Client rate limiting (throttling)
  - Logging and monitoring
  - Response caching
  - Web application firewall
  - GZIP compression
  - Servicing static content



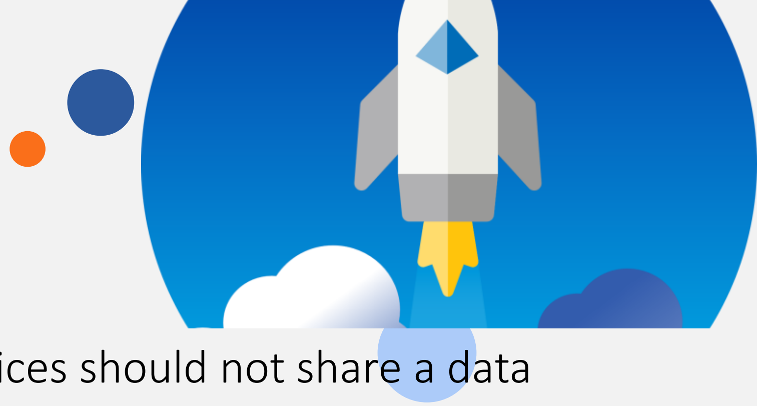
API Gateway (cont.)

Available Options

- **Reverse proxy server:**
  - YARP: Yet Another Reverse Proxy
  - Nginx
- **Azure Application Gateway:** a managed load balancing service, provides a web application firewall
- **Azure API Management**: managing a public-facing API
  - Rate limiting
  - IP restrictions
  - Authentication using Azure Active Directory or other identity providers
  - Doesn't perform any load balancing



Data Management



A basic principle of microservices is that each service manages its own data. Two services should not share a data store.

Approaches for managing data:

- Embrace **eventual consistency** where possible. Understand the places where you need strong consistency and the places where you need eventual consistency
- When you need strong consistency guarantees, one service may represent the **source of truth for a given entity**, which is exposed through an API.
- For transactions, use patterns such as **Scheduler Agent Supervisor and Compensating Transaction** to keep data consistent across several services
- Store **only** the data that a service needs
- Use an **event driven** architecture style

Thanks

ahmed.abouzeid@outlook.com