Micro Services Architecture

Apparao G

Monolithic Architecture Pattern:

The monolithic architecture is an architectural style that structures the application as a single executable component.

Our aim is to develop server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications.

The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker.

The application handles requests (HTTP requests and messages) by executing business logic, accessing a database; exchanging messages with other systems and returning a HTML/JSON/XML response.

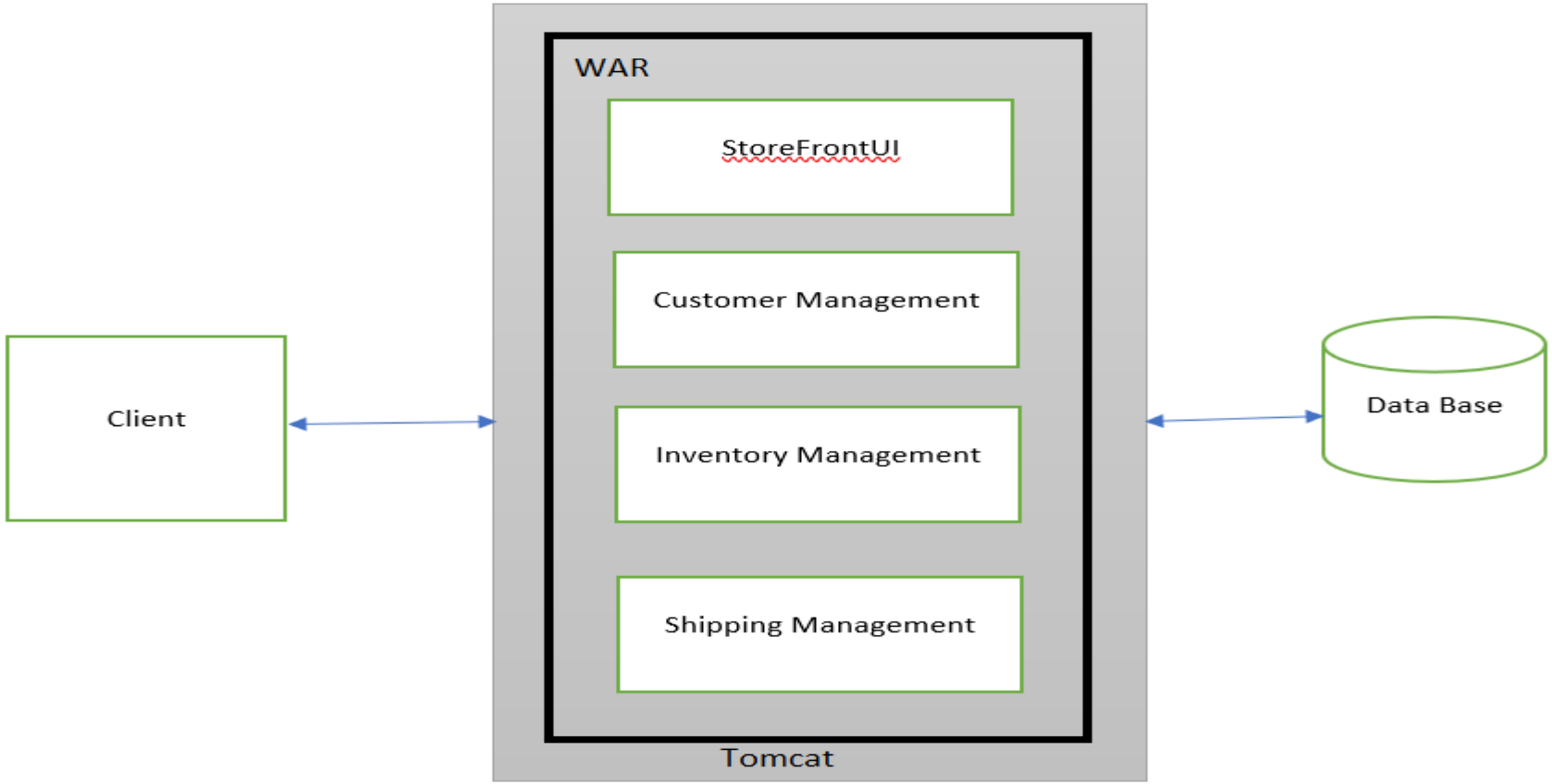
There are logical components corresponding to different functional areas of the application.

Solution:
Build an application with a monolithic architecture.

Example 1: Design the e-commerce application

You are building an e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them. The application consists of several components including the Store FrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders.

The application is deployed as a single monolithic application. For example, a Java web application consists of a single WAR file that runs on a web container such as Tomcat.



Example 2: Design Food order application

The Food application business logic consists of modules, each of which is a collection of domain objects.

Examples of the modules include Order Management, Delivery Management, Billing, and Payments. There are several adapters that interface with the external systems.

Some are inbound adapters, which handle requests by invoking the business logic, including the REST API and Web UI adapters. Others are outbound adapters,which enable the business logic to access the MySQL database and invoke cloud services such as Twilio and Stripe.

It is packaged as a single WAR file.

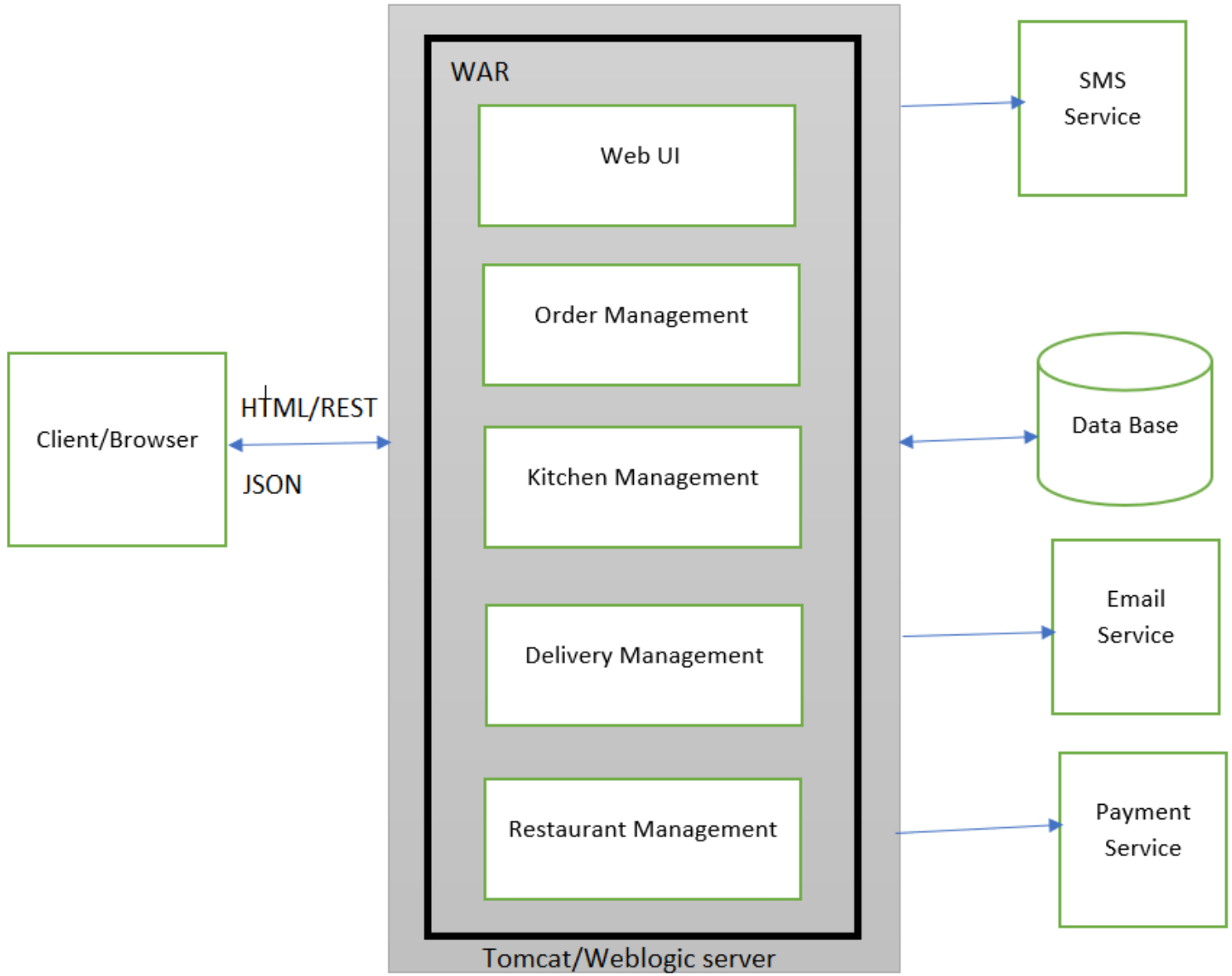
Order module : Manages orders

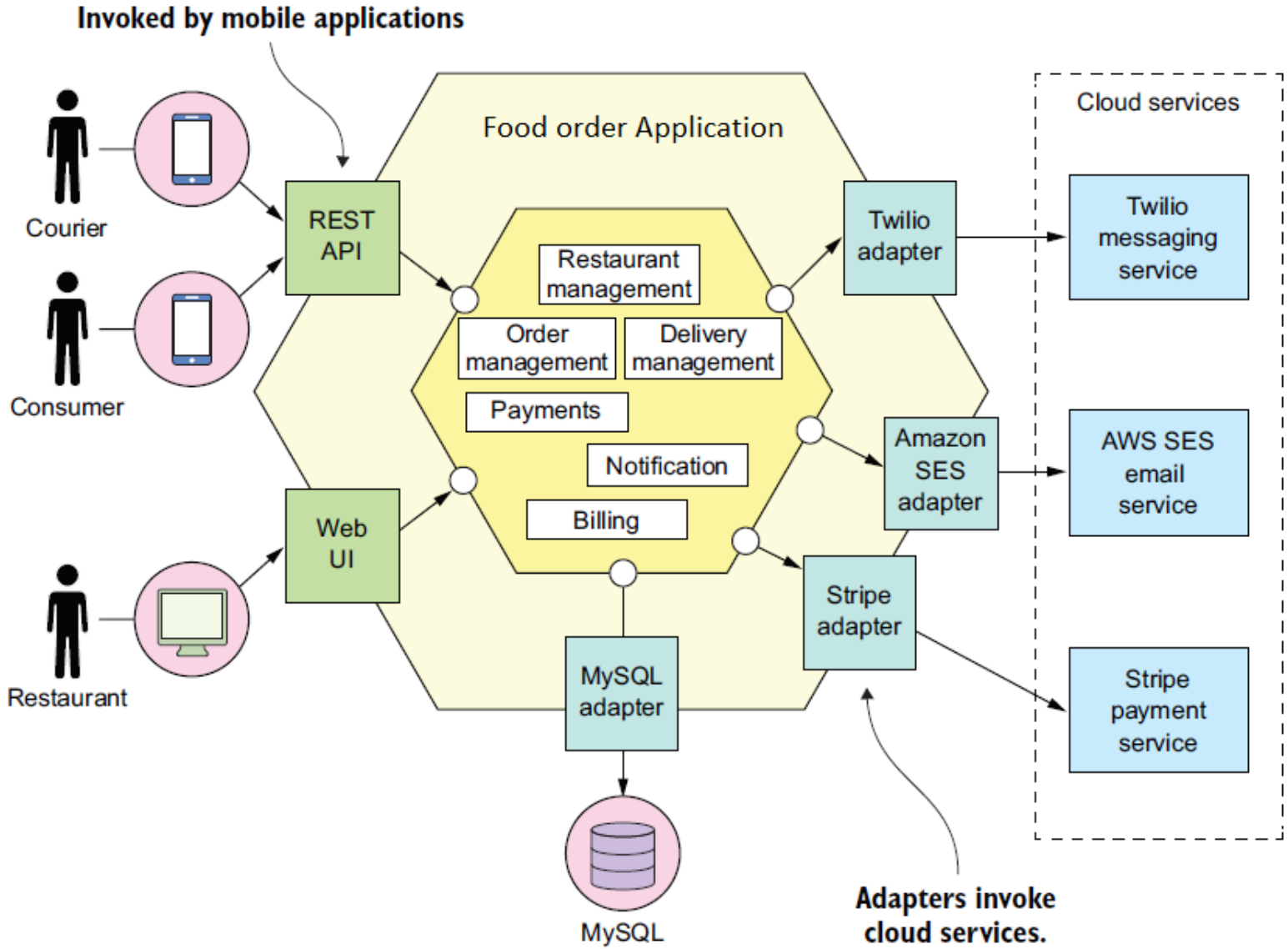
Delivery module : Manages delivery of orders from restaurants to consumers

Restaurant module : Maintains information about restaurants

Kitchen module : Manages the preparation of orders

Accounting module : Handles billing and payments


WAR
Web UI
Order Management
Kitchen Management
Delivery Management
Restaurant Management
Tomcat/Weblogic server
Client/Browser
HTML/REST
JSON
SMS
Service
Data Base
Email
Service
Payment
Service


Invoked by mobile applications
Food order Application
Cloud services
Courier
Consumer
Restaurant
REST API
Web UI
Restaurant management
Order management
Delivery management
Payments
Notification
Billing
MySQL adapter
MySQL
Twilio adapter
Amazon SES adapter
Stripe adapter
Twilio messaging service
AWS SES email service
Stripe payment service
Adapters invoke cloud services.

Benefits of the monolithic architecture:

- Simple to develop: IDEs and other developer tools are focused on building a single application.
- Easy to make radical changes to the application: You can change the code and the database schema, build, and deploy.
- Simple to test: The developers wrote end-to-end tests that launched the application, invoked the REST API, and tested the UI with Selenium.
- Simple to deploy: All a developer had to do was copy the WAR file to a server that had Tomcat installed.
- Easy to scale: Run multiple instances of the application behind a load balancer.

Disadvantages of monolithic architecture:

- Longer development times
- Fixed technology stack
- High levels of coupling between modules and between services
- Failure could affect whole system down because it is running single server
- Minor change could result in complete rebuild
- The large monolithic code base intimidates developers, especially ones who are new to the team. The application can be difficult to understand and modify. As a result, development typically slows down. Moreover, because it can be difficult to understand how to correctly implement a change the quality of the code declines over time.
- Overloaded IDE : the larger the code base the slower the IDE and the less productive developers are.
- Overloaded web container : the larger the application the longer it takes to start up. This had have a huge impact on developer productivity because of time wasted waiting for the container to start. It also impacts deployment too.
- Continuous deployment is difficult : a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems.
- Scaling the application can be difficult : a monolithic architecture is that it can only scale in one dimension. On the one hand, it can scale with an increasing transaction volume by running more copies of the application.
- Different application components have different resource requirements : one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently

Micro Service architecture:

Micro service : Small independently deployable services that work together, modelled around a business domain.

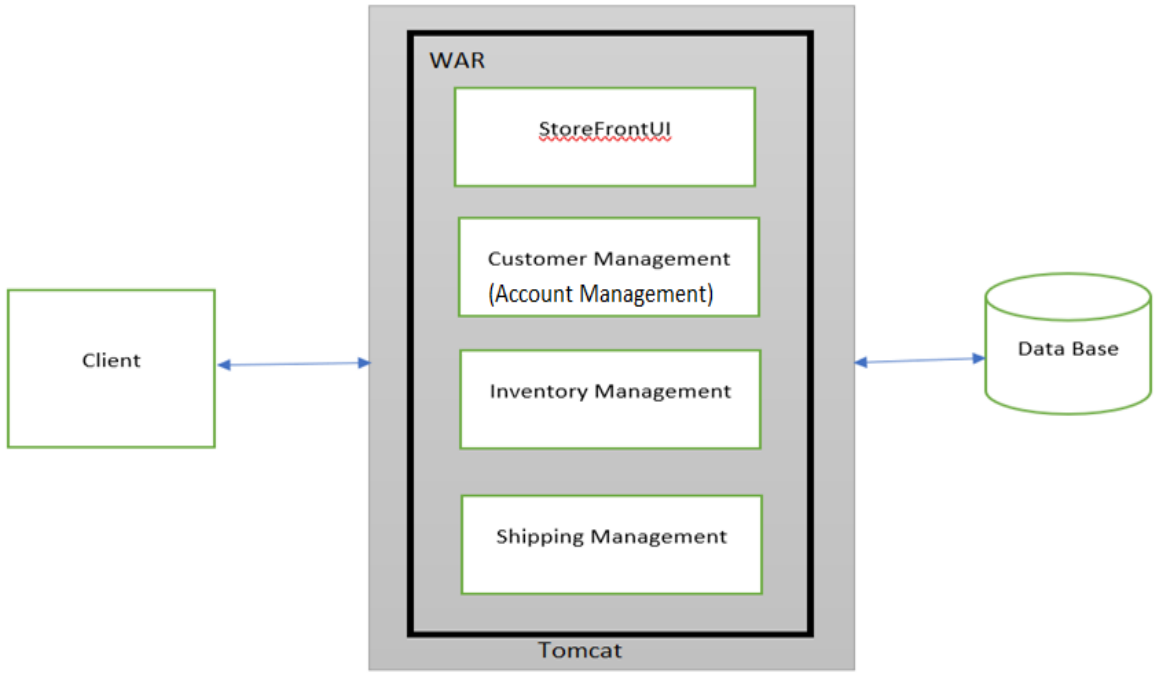
The micro service architecture is an architectural style that structures an application as a set of loosely coupled, services organized around business capabilities.

Example1: Design e-commerce application.

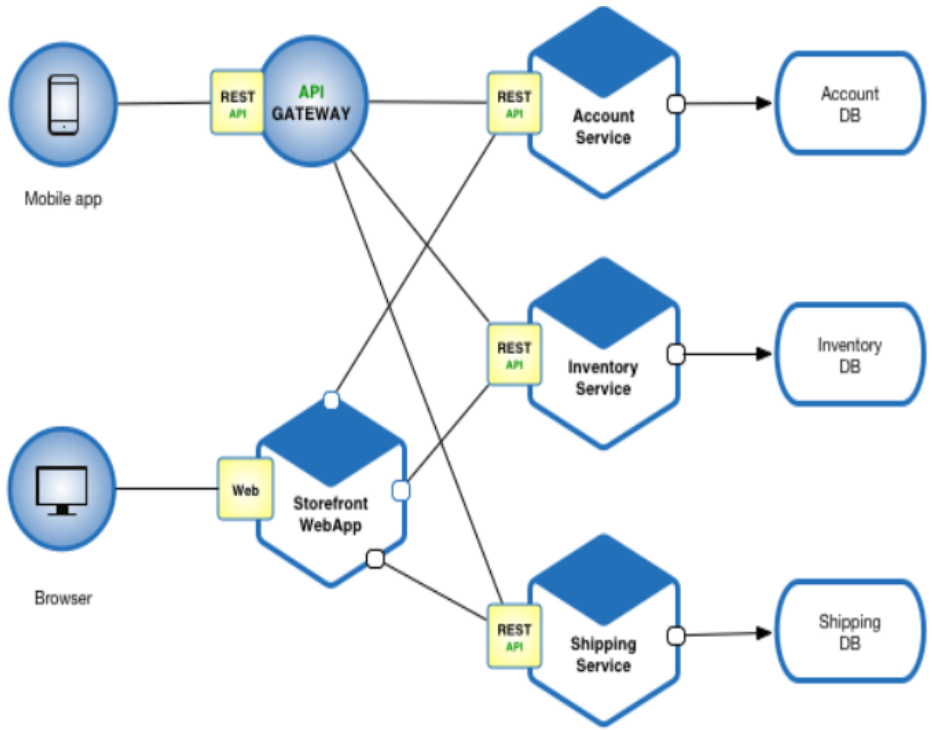
e-commerce application that takes orders from customers, verifies inventory and available credit, and ships them.

The application consists of several components including the StoreFrontUI, which implements the user interface, along with some backend services for checking credit, maintaining inventory and shipping orders.

Define an architecture that structures the application as a set of loosely coupled, collaborating services.
The application consists of a set of services. Below are the some of the services
Account service
Inventory service
Shipping service
Store Front UI



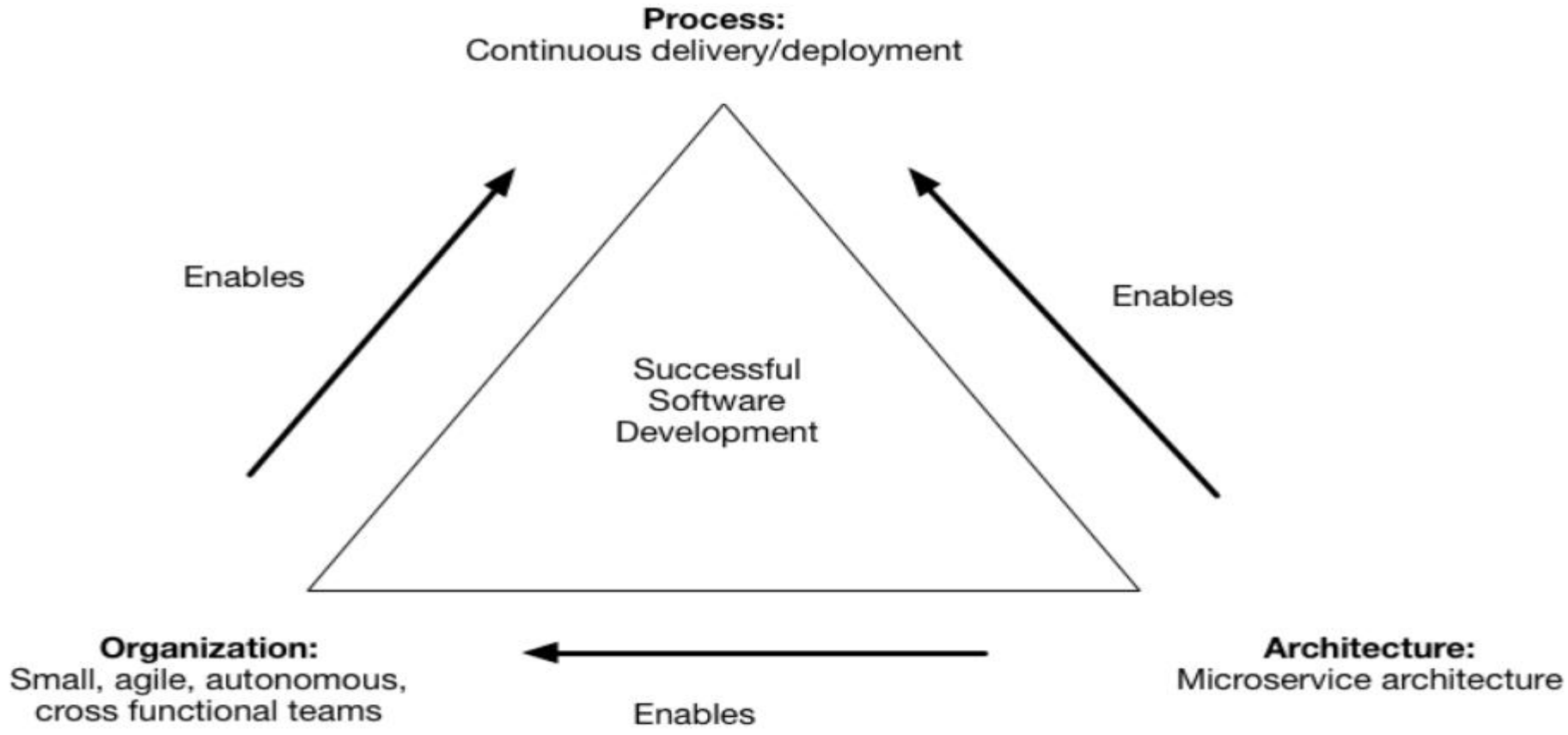
Monolithic Architecture



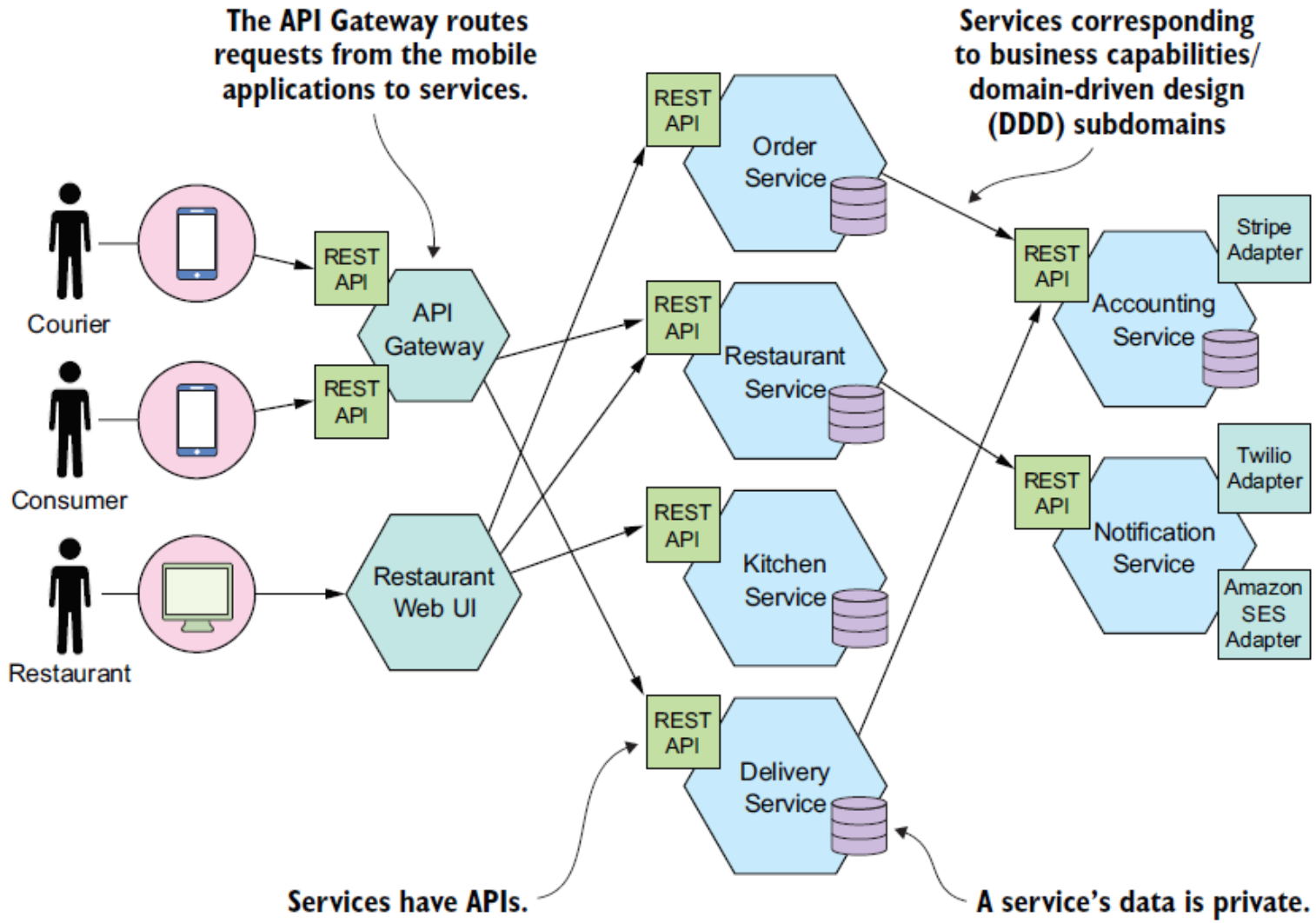
Micro Service architecture

How to decompose an application into services?

1. Decompose by business capability
2. Decompose by subdomain



Example 2: Design Food Order Application



Microservice architecture benefits:

- Enables frequent updates
- Decouple the changeable parts
- Fast issue resolution
- It enables the continuous delivery and deployment of large, complex applications.
- Services are small and easily maintained.
- Services are independently deployable.
- Services are independently scalable.
- The microservice architecture enables teams to be autonomous. (Better ownership and knowledge)
- It allows easy experimenting and adoption of new technologies.
- It has better fault isolation.
- Highly scalable and better performance

Microservice architecture drawbacks:

- Finding the right set of services is challenging.
- Integration testing and end to end testing is difficult .
- Partial failures
- Distributed systems are complex, which makes development, deployment difficult.
- Deploying features that span multiple services requires careful coordination.
- Deciding when to adopt the microservice architecture is difficult.
- Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes.

Microservices Design Principles:

High Cohesion

Autonomous

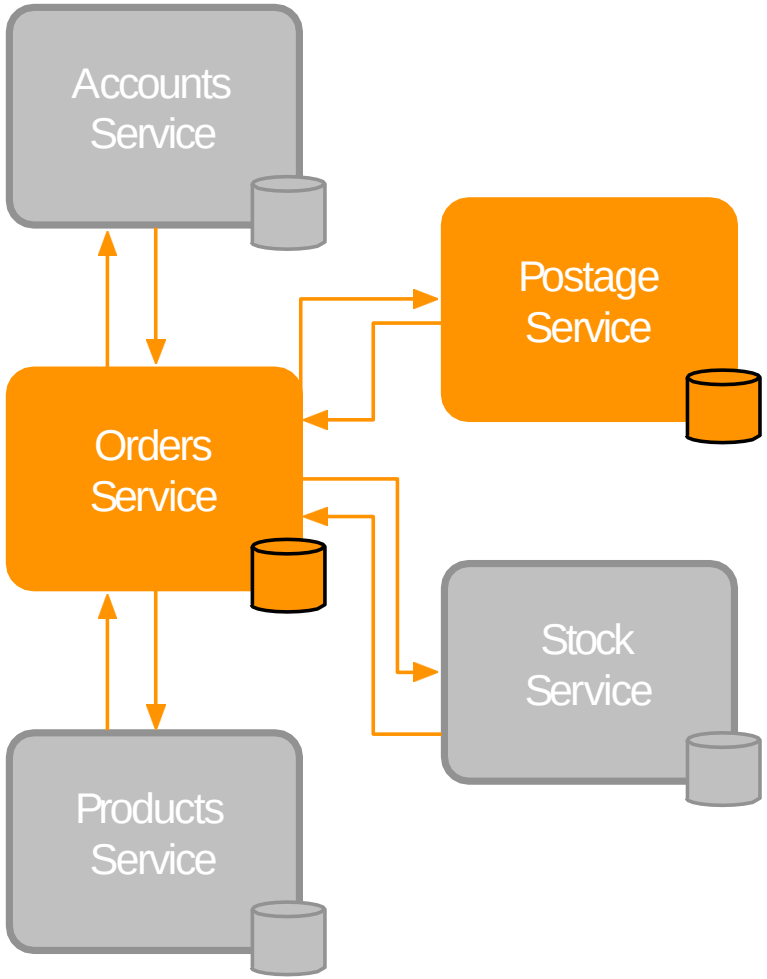
Business Domain Centric

Resilience

Observable

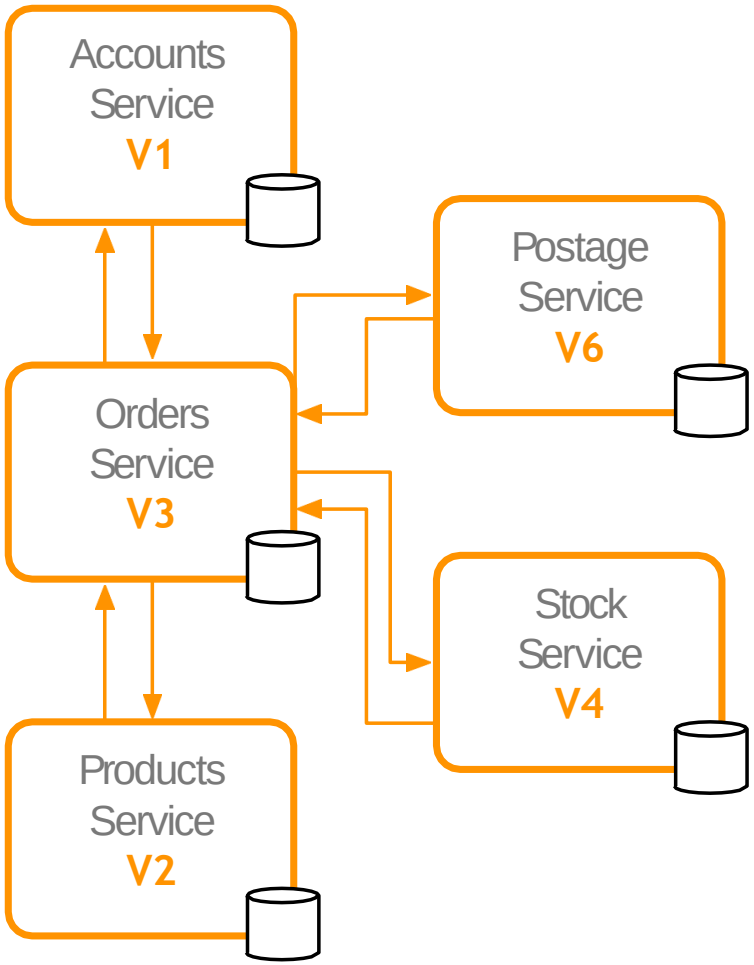
Automation

Microservices Design Principles: High Cohesion



- Single focus
- Single responsibility
  - SOLID principle
  - Only change for one reason
- Reason represents
  - A business function
  - A business domain
- Encapsulation principle
  - OOP principle
- Easily rewritable code
- Why
  - Scalability
  - Flexibility
  - Reliability

Microservices Design Principles: Autonomous



Loose coupling

Honor contracts and interfaces

Stateless

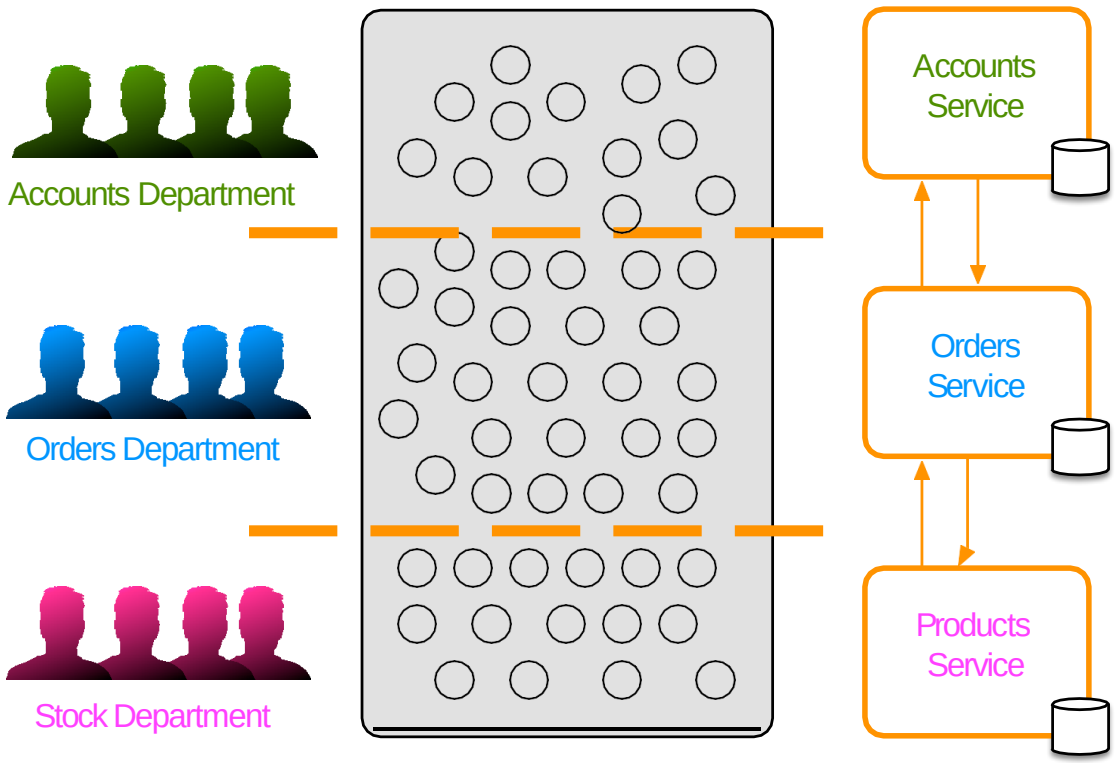
Independently changeable

Independently deployable

Backwards compatible

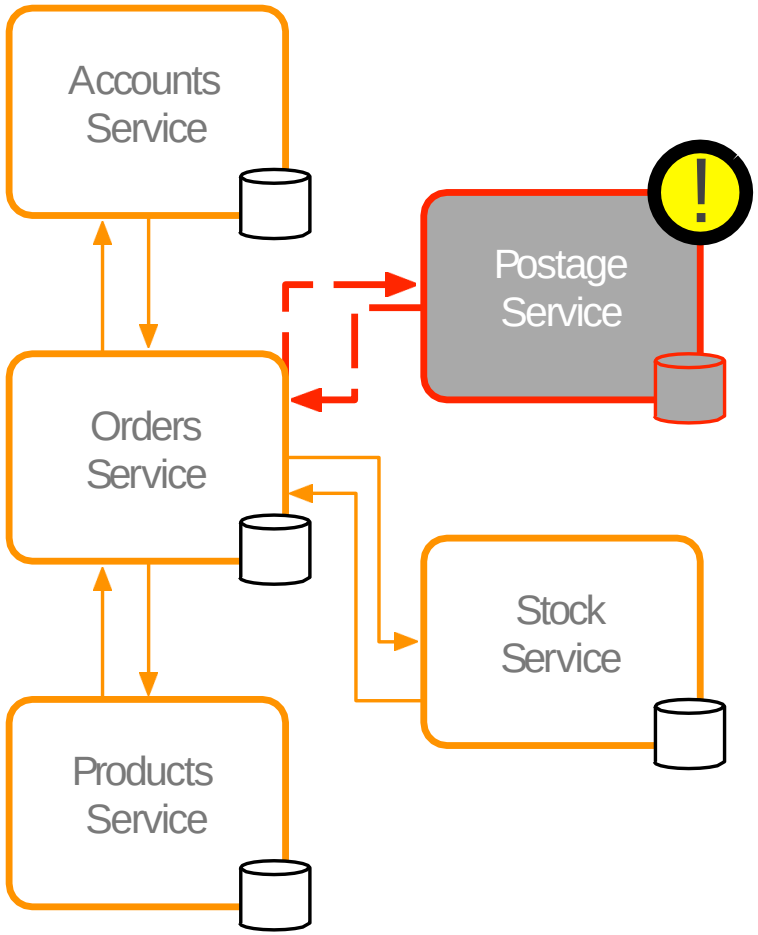
Concurrent development

Design Principles: Business Domain Centric



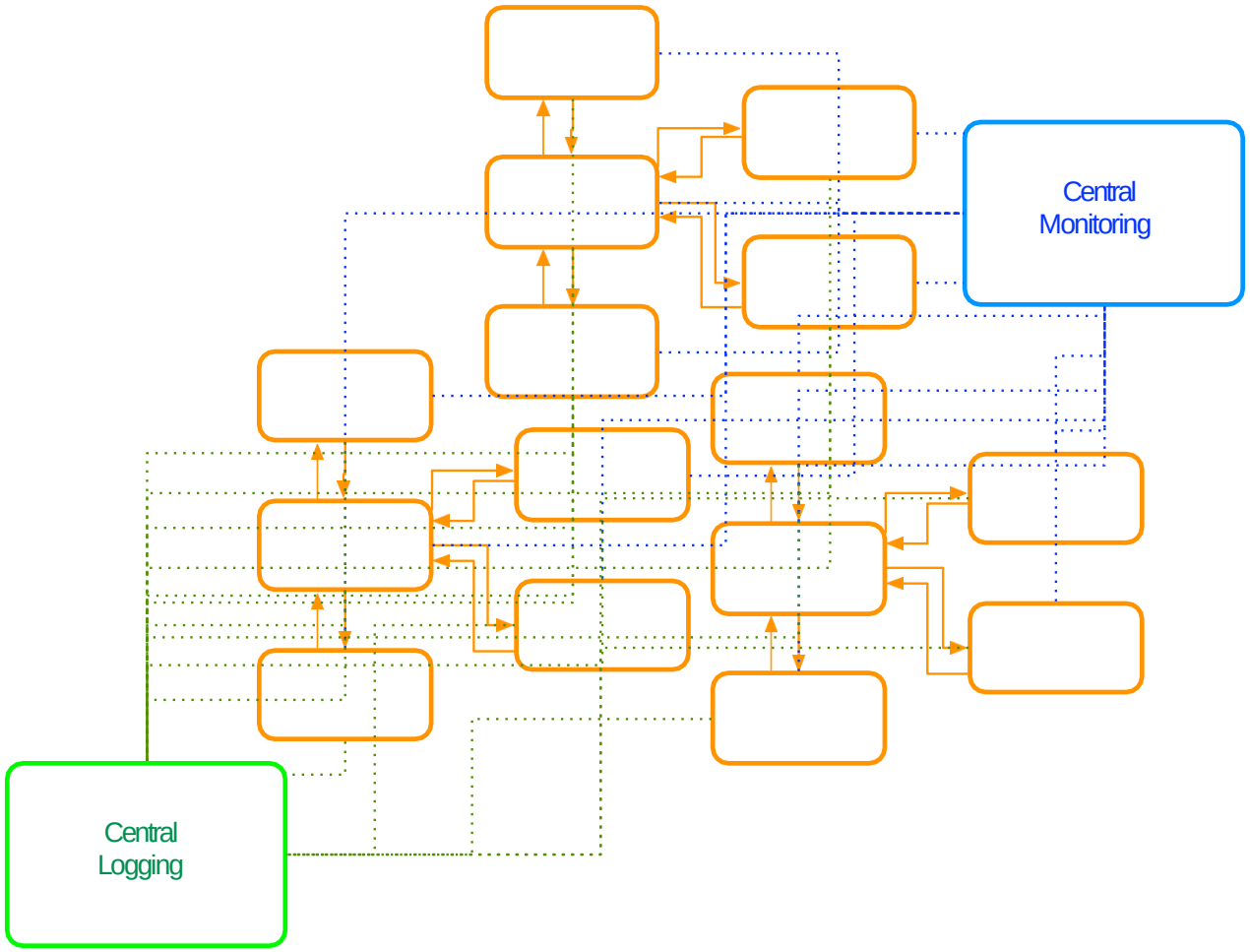
- Service represents business function
  - Accounts Department
  - Postage calculator
- Scope of service
- Bounded context from DDD
- Identify boundaries\seams
- Shuffle code if required
  - Group related code into aservice
  - Aim for high cohesion
- Responsive to business change

Microservices Design Principles: Resilience



- Embrace failure
  - Another service
  - Specific connection
  - Third-party system
- Degrade functionality
- Default functionality
- Multiple instances
  - Register on startup
  - Deregister on failure
- Types of failure
  - Exceptions\Errors
  - Delays
  - Unavailability
- Network issues
  - Delay
  - Unavailability
- Validate input
  - Service to service
  - Client to service

Microservices Design Principles: Observable



System Health

- Status
- Logs
- Errors

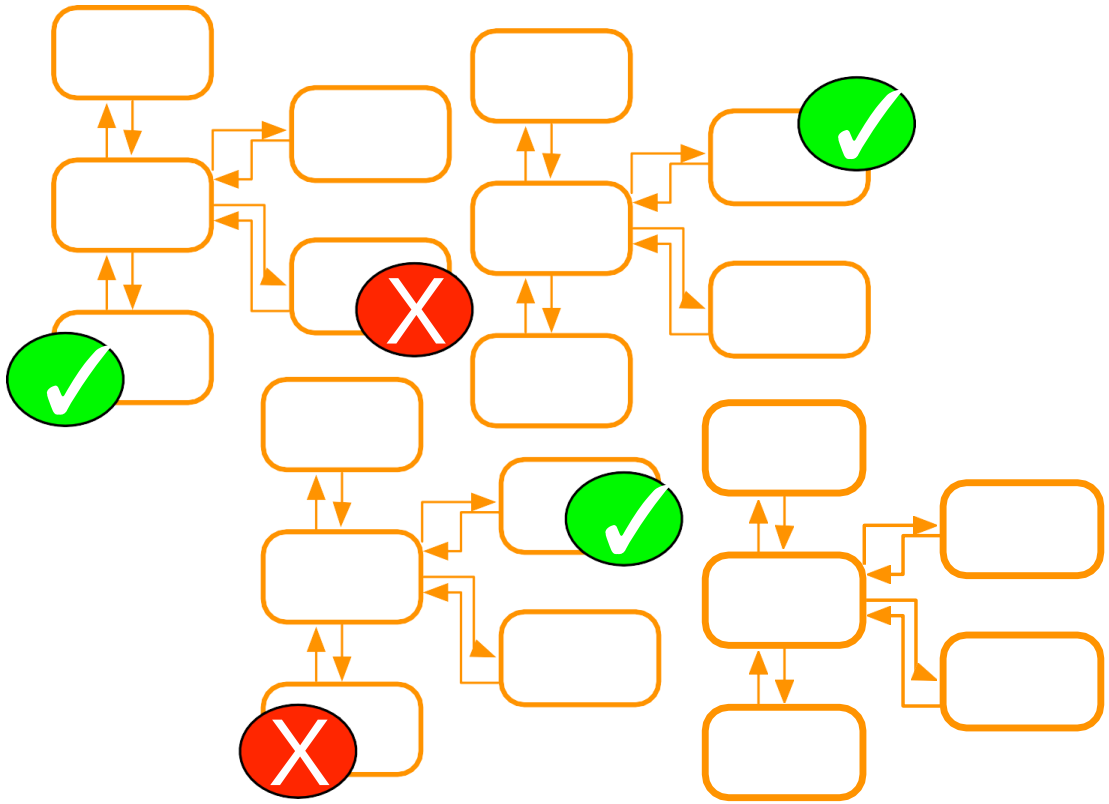
Centralized monitoring

Centralized logging

Why

- Distributed transactions
- Quick problem solving
- Quick deployment requires feedback
- Data used for capacity planning
- Data used for scaling
- Whats actually used
- Monitor business data

Microservices Design Principles: Automation



Tools to reduce testing

- Manual regression testing
- Time taken on testing integration
- Environment setup for testing

Tools to provide quick feedback

- Integration feedback on check in
- Continous Integration

Tools to provide quick deployment

- Pipeline to deployment
- Deployment ready status
- Automated deployment

Why

- Reliable deployment
- Continuous Deployment

- Distributed system
- Multiple instances of services
- Manual integration testing too time consuming
- Manual deployment time consuming and unreliable