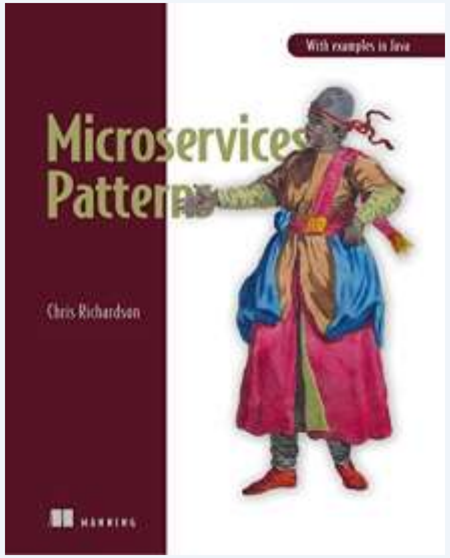
Service Decomposition Patterns

Objectives

- Service Decomposition
- Defining independent, loosely coupled services.
- System operations to assist in decomposition
- Service Decomposition Patterns

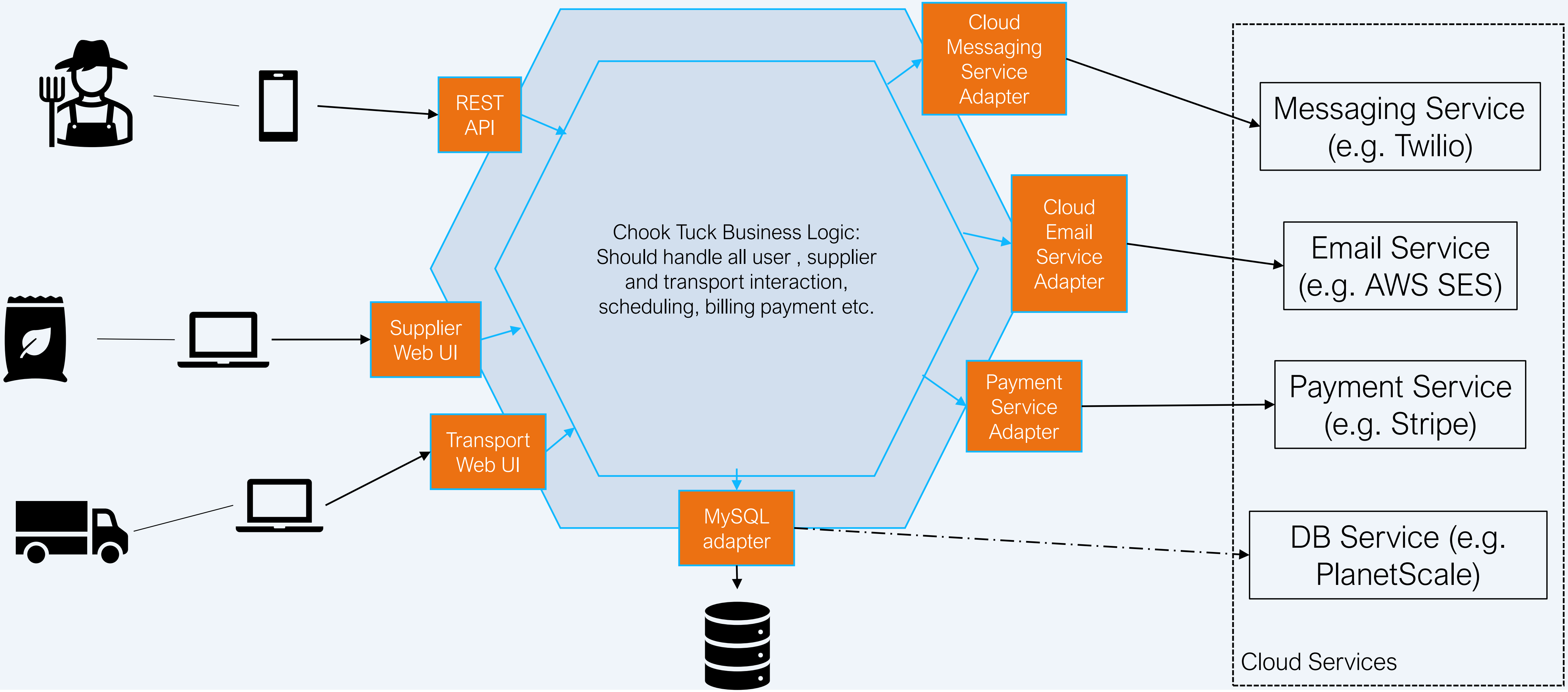


Decomposition Patterns

A Fictional Service – Chook Tuck

- Chook Tuck is a fictional company that acts as a broker between poultry farmers and chicken feed suppliers.
- Suppliers and transport companies are able to register with the company, and work is offered to them as orders become available.
- Farmers put in an order for one of three types of feed (starter, grower, finisher) in 50kg bags. The farmer can optionally nominate a supplier.
- Chook Tuck then checks the farmer’s credit, and the availability of stock and transport and returns a delivery proposal.
    - An internal process attempts to load multiple deliveries onto a single truck to save costs.
- Once accepted the invoice is made up and the order is placed.
- Payment is taken upon delivery of the goods.

Chook Tuck Application


REST API
Supplier Web UI
Transport Web UI
Chook Tuck Business Logic: Should handle all user , supplier and transport interaction, scheduling, billing payment etc.
Cloud Messaging Service Adapter
Cloud Email Service Adapter
Payment Service Adapter
MySQL adapter
Messaging Service (e.g. Twilio)
Email Service (e.g. AWS SES)
Payment Service (e.g. Stripe)
DB Service (e.g. PlanetScale)
Cloud Services

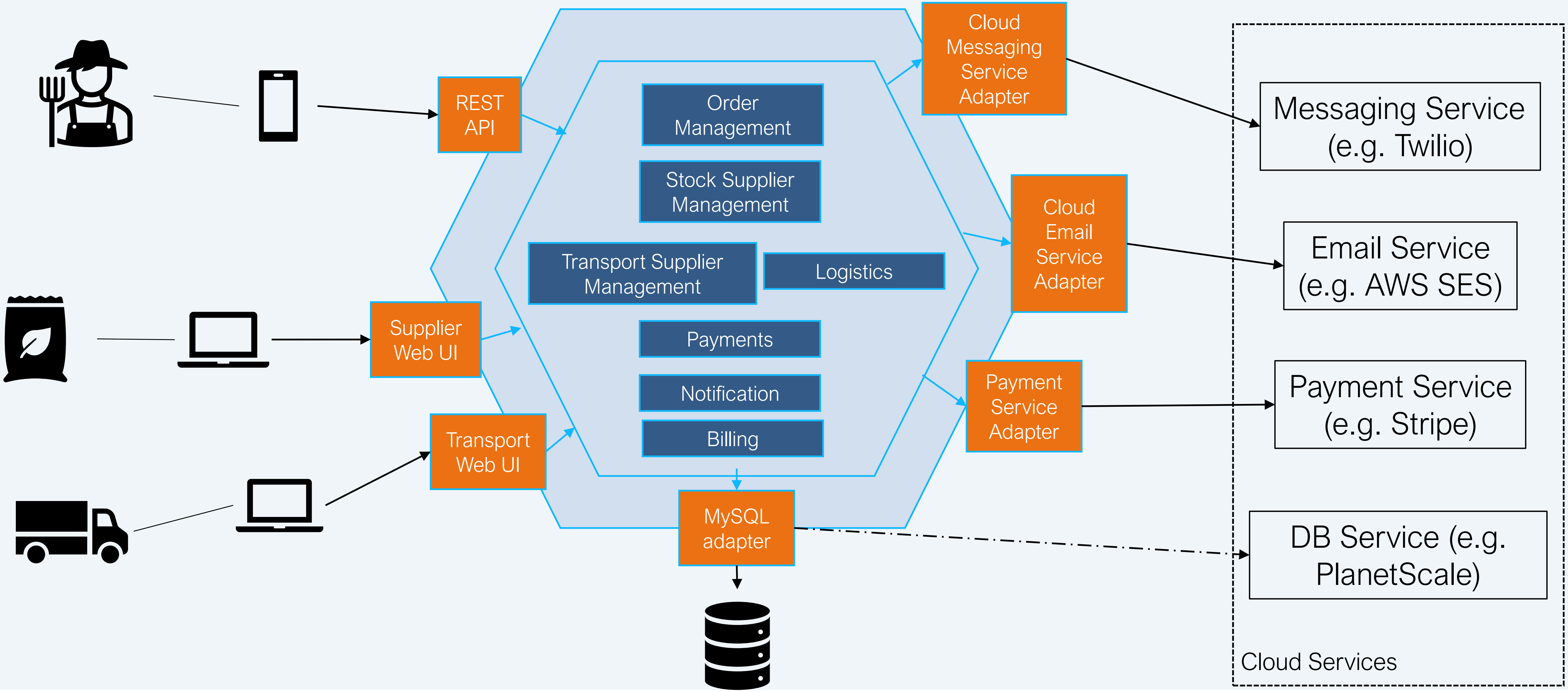
Owerya RESOURCING

Group Exercise

Break Chook Tuck into sensible microservices:

- Loosely coupled
- Each with their own databases

Chook Tuck Application with Modules


REST API
Supplier Web UI
Transport Web UI
Order Management
Stock Supplier Management
Transport Supplier Management
Logistics
Payments
Notification
Billing
MySQL adapter
Cloud Messaging Service Adapter
Cloud Email Service Adapter
Payment Service Adapter
Messaging Service (e.g. Twilio)
Email Service (e.g. AWS SES)
Payment Service (e.g. Stripe)
DB Service (e.g. PlanetScale)
Cloud Services

Defining Independent Loosely Coupled Services

- The question of where to set your service boundaries is not automatic
  - Should you have a separate service for communicating with transport and suppliers?

- What does loose coupling mean?
  - Collaborate only via APIs – you can't share a database

- What about shared libraries?
  - A poorly implemented shared library can accidently introduce coupling between services.
  - Only use libraries for functionality that is very unlikely to change
    - For example, a generic `Money` class – no point in implementing this in every service.

- How big?
  - Should be manageable by a small, independent team.

Defining a Microservices Architecture

- Same as defining any architecture, you (hopefully) have
  - Written requirements
  - Domain experts
  - Possibly an existing application
- We can define a 3 step process to define the architecture
  - It’s not mechanical
  - It is iterative
  - It requires a lot of creativity!

Three Steps to Defining a Microservices Architecture

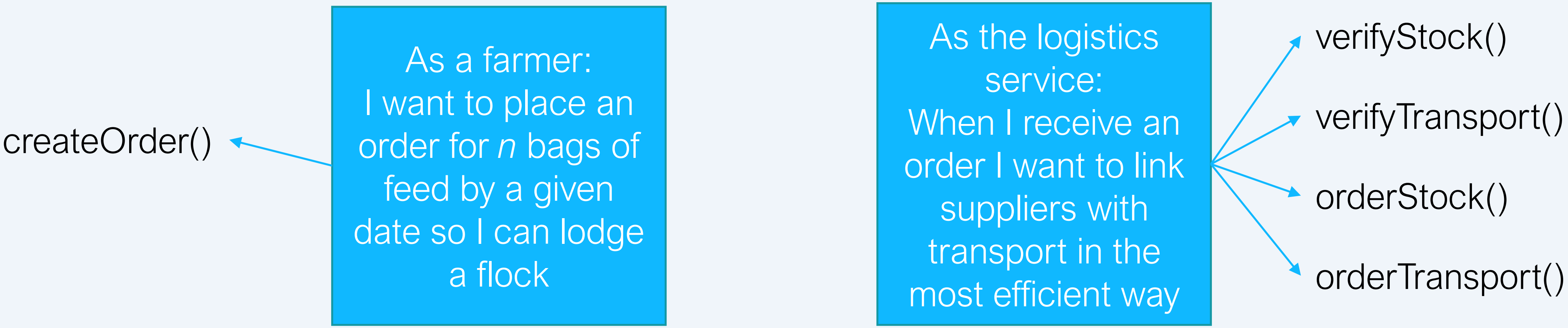
1. Identify System Operations
   - Distil the application's requirements into key requests.
   - It is a good idea to keep this abstract - queries and commands.
2. Identify/Define Services
   - This is step at which the actual microservices are shaped
   - We will look at two strategies for this
3. Define Service APIs and Collaborations
   - Assign each system operation (Step 1) to service(s) (Step 2).
   - A system operation may require only a single service, or a collaboration between services.
   - This also includes deciding on inter-process communication mechanisms.

Decomposition Obstacles

- Latency
  - A particular decomposition might involve too many round-trips between services
- Reduced Availability
  - Synchronous comms can induce tight coupling and cascading failures.
  - Decomposition might be fine, but need asynchronous comms.
- Lack of Data Consistency
  - De-normalization of databases can mean individual DBs in inconsistent state a points in time.
  - This is addressed using Sagas.
- God classes
  - Your original design might have classes that touch the whole application

Step 1: System Operations

- System operations are request transactions into the system or between the services. Can be either
    - Command – this creates, deletes or updates data
    - Query – retrieves data
- Standard user stories help define what users and suppliers expect from the system.
- Service stories help to define your system operations



createOrder()

As a farmer:
I want to place an order for *n* bags of feed by a given date so I can lodge a flock

As the logistics service:
When I receive an order I want to link suppliers with transport in the most efficient way

verifyStock()

verifyTransport()

orderStock()

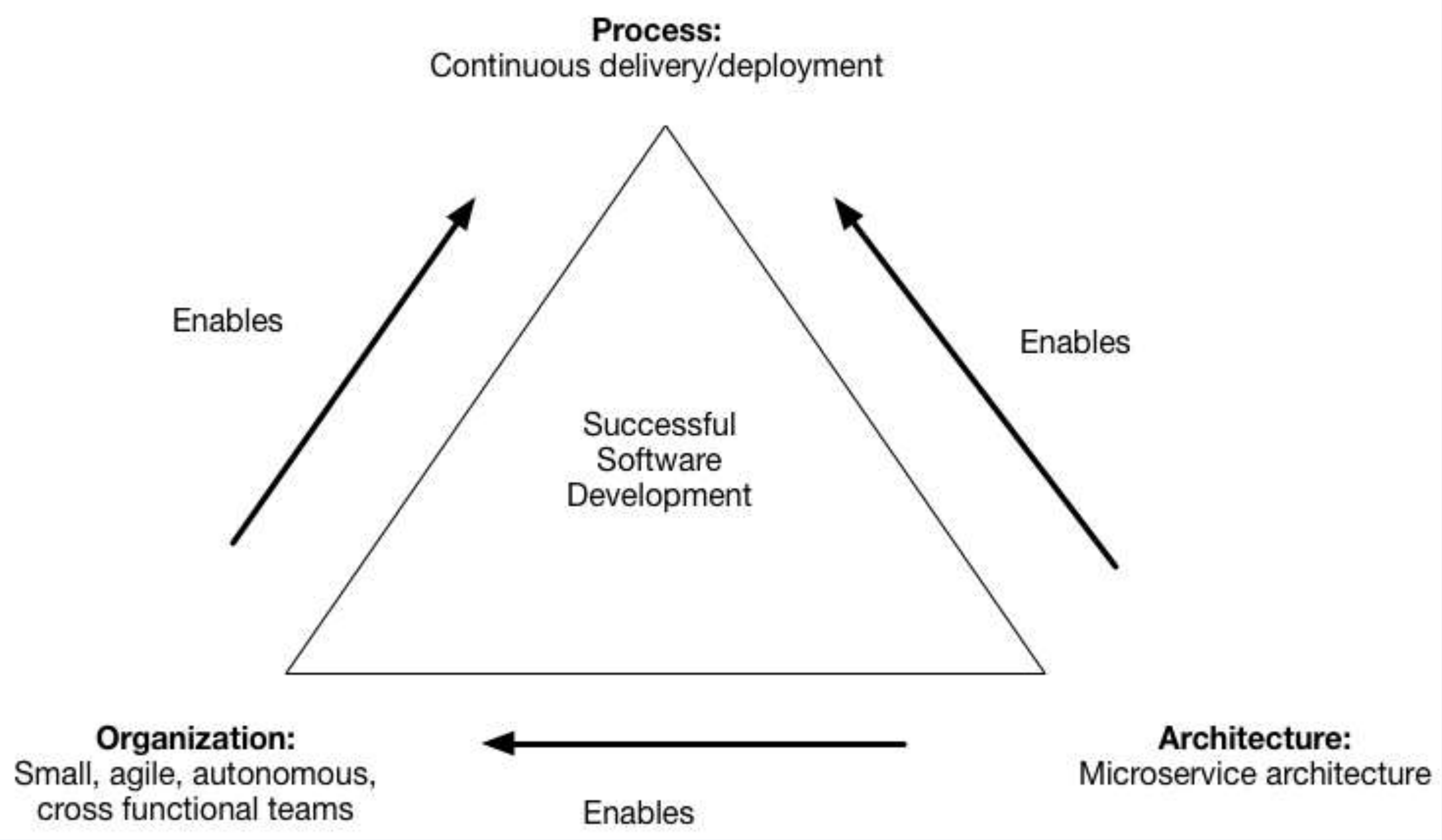
orderTransport()

Step 2: Define Services

- There are a couple of patterns that help us with this.
- First of all
  - Context
  - Problem
  - Forces

Context & Problem

Context:


Process:
Continuous delivery/deployment
Enables
Enables
Successful Software Development
Organization:
Small, agile, autonomous, cross functional teams
Enables
Architecture:
Microservice architecture

Problem: How to decompose into Microservices?

https://microservices.io/patterns/decomposition/decompose-by-subdomain.html

Question

What are the forces present when trying to define/decouple services?

Forces

- The architecture must be stable
- Services must be cohesive.
  - A service should implement a small set of strongly related functions.
- Services must conform to the Common Closure Principle
  - things that change together are packaged together → changes affect only one service
- Services must be loosely coupled
  - each service as an API that encapsulates its implementation.
- A service should be testable
- Each service be small enough to be developed by a team of 6-10 people
- Each team that owns one or more services must be autonomous
  - Able to develop and deploy their services with minimal collaboration with other teams.

Solutions: These patterns often give similar decompositions

Pattern 1: Decomposition by Business Capability

- Services are “something a business does to generate value”
- E.g. *Order management* is responsible for orders

Pattern 2: Decomposition by Subdomain (DDD)

- Services correspond to Domain-Driven Design subdomains.
- Each subdomain corresponds to a different part of the business
- Subdomains are classified as
  - Core – key differentiator for the business
  - Supporting – business related but not a differentiator; could be outsourced.
  - Generic – not business specific. Ideally off-the-shelf software.

Decomposition by Business Capability Pattern

- Business capabilities capture *what* an organization's business is
  - Not *how* it is done.
  - They're generally stable, even though the "how" might change dramatically
  - E.g. for banking the "Deposit check" business capability has remained stable, while how checks are deposited has changed dramatically.
- Identified by analysing an organisation's purpose, structure and business processes.
- A capability is a "business-oriented service"
  - Think about is as the answer to "does your business do …?"
- Should be done by people very familiar with how the business works

Mapping Capabilities to Services

- This is rather subjective!
- Sometimes a capability will map to multiple services
  - E.g. for Chook Tuck there might be a “supplier management” capability that handles both transport and produce suppliers.
  - It is probably best to have separate services for each of these
- Sometimes a capability will map directly to a service

Decomposition by Subdomain

- This pattern follows Domain-driven Design
- Based on concept that business entities (customer, order, etc) differ depending on the subdomain within the business
  - Customer in the order taking subdomain is different to customer in the supplier subdomain.
- Each (sub)domain maps onto a bounded context which is it's scope.
  - E.g. There is a bounded context for the order domain model.
  - A bounded context is a microservice or a set of microservices
- The subdomains are determined like business capabilities and often will be very similar.
- The concept of a subdomain with its own domain model/bounded context helps to eliminate god classes.

Decomposition Guidelines (from OOP)

- Single Responsibility Principle

  Each class (service) should only ever have one reason to change

  - This means we strive to have small, cohesive services that each have a single responsibility.

- Common Closure Principle

  The classes (components) in a package (service) should be closed together against the same kinds of changes. A change that affects a package (service) affects all the classes (components) in that package (service).

  - This means that we collect all the pieces that change for the same reason into the same service.
  - This minimizes the number of services that need to be changed when a requirement changes.

From: Designing Object Oriented C++ Applications Using the Booch Method, Robert C Martin, Prentice Hall 1995

Resulting Context

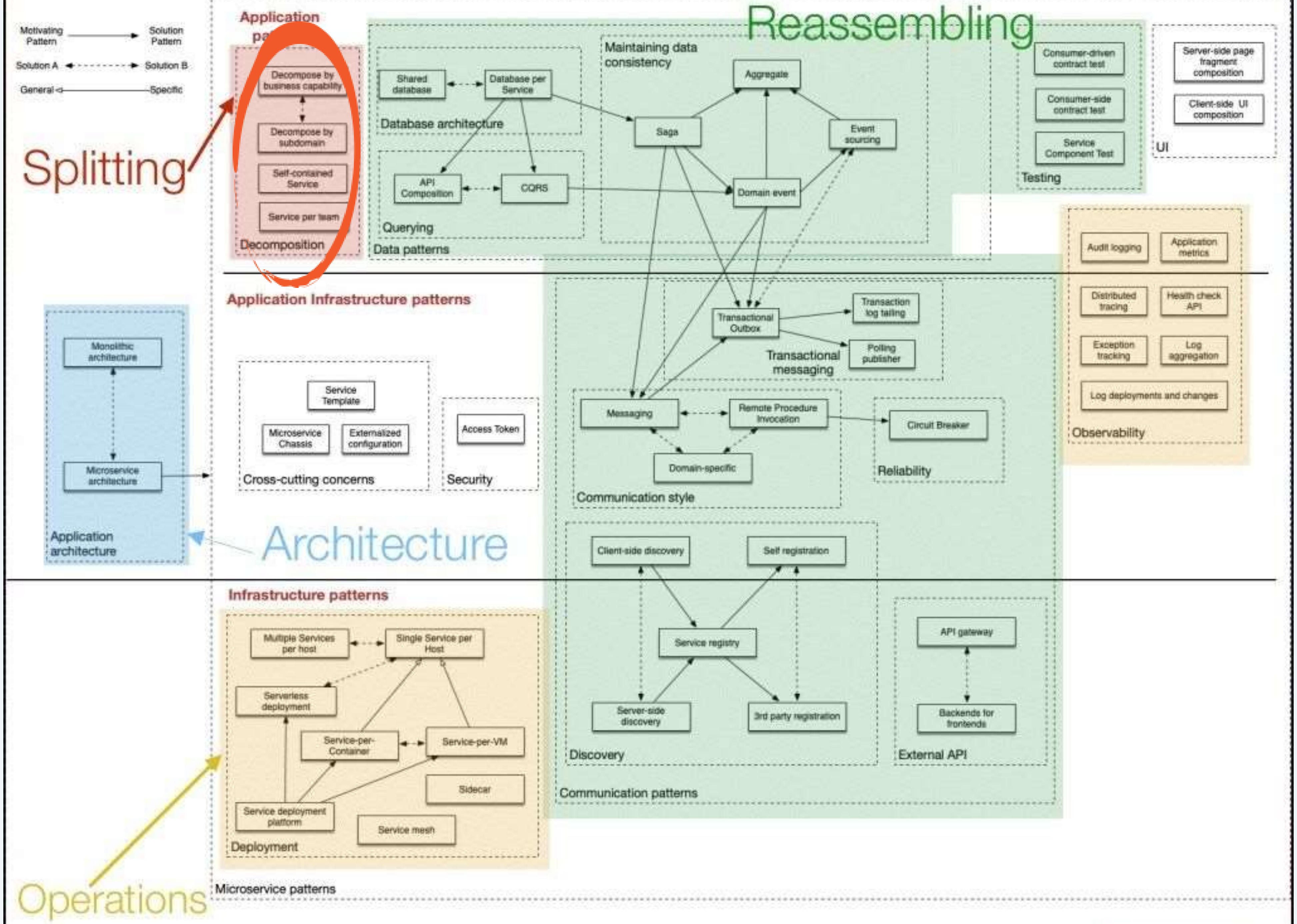
Stable architecture since the capabilities/subdomains are relatively stable

Development teams are cross-functional, autonomous, and organized around delivering business value rather than technical features

Services are cohesive and loosely coupled

Issues

- Identifying the business capabilities/subdomains is not trivial
  - Requires a good understanding of the business.
  - The current organizational structure might give a good starting point
  - Think in particular about teams!



Learn-Build-Assess Microservices http://adopt.microservices.io

Summary

- Service Decomposition
- Defining independent, loosely coupled services.
- System operations to assist in decomposition
- Service Decomposition Patterns

Questions or Comments?