

# Microservices Architecture

*Software Architecture*

Richard Thomas

April 7, 2025

## *History*

- Service Oriented Architecture (SOA)
  - Enterprise Service Bus (ESB)
  - RESTful APIs
  - Microservices
- REST – REpresentational State Transfer

# SOA

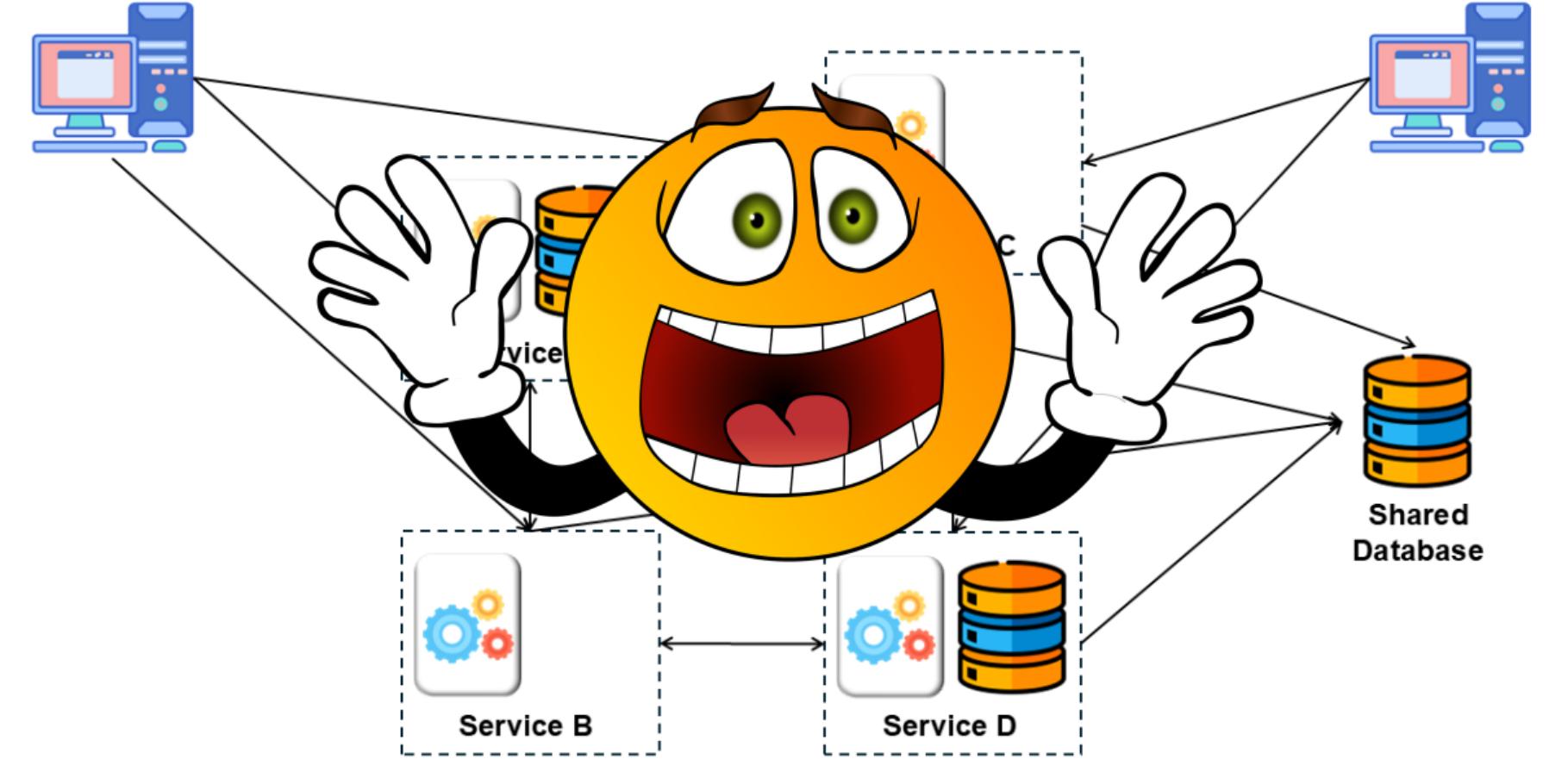


- DB Icon made by setiawanap from [www.flaticon.com](http://www.flaticon.com)
- Cogs Icon made by Design Circle from [www.flaticon.com](http://www.flaticon.com)

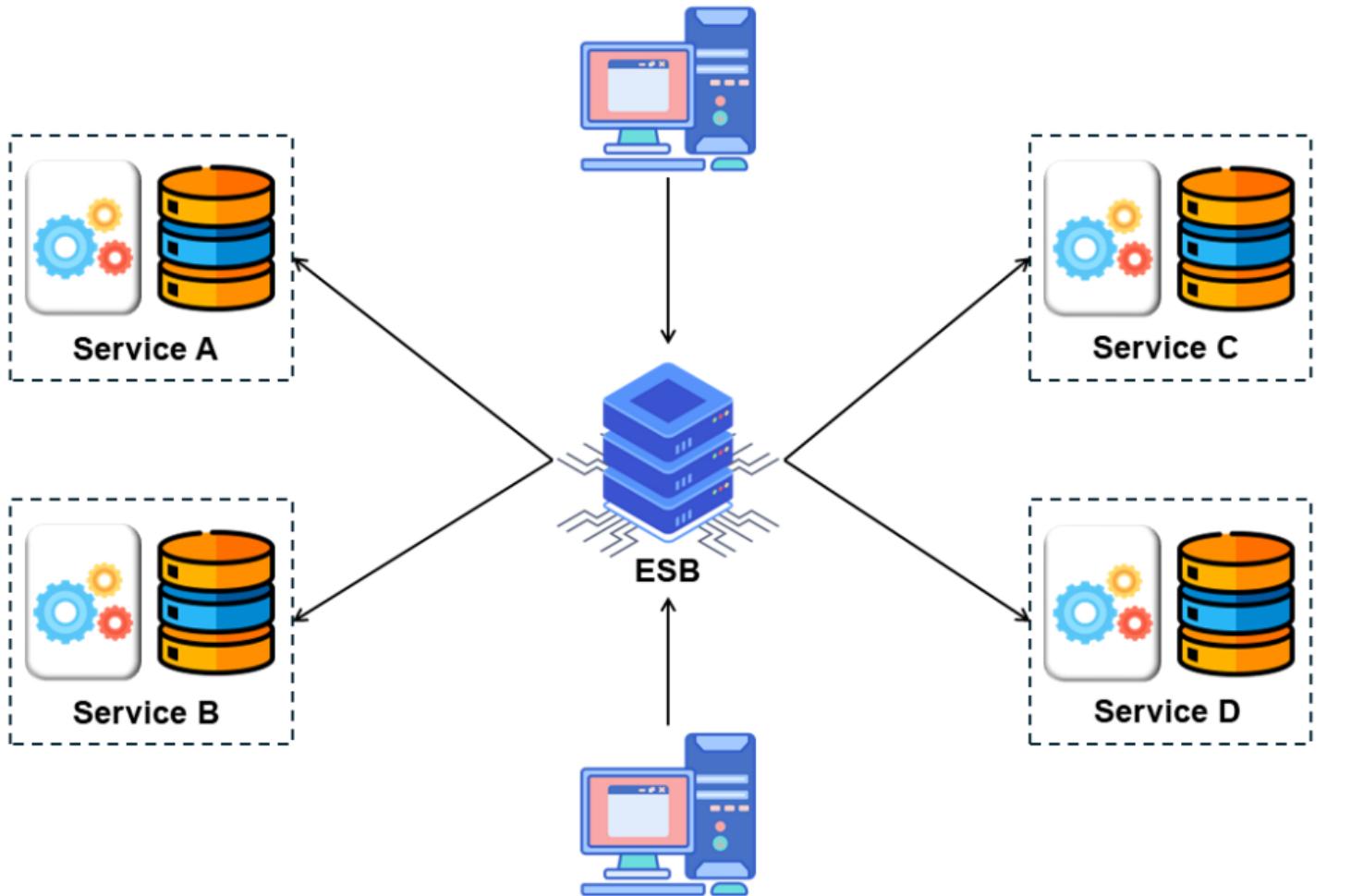
## SOA Circumvented



## SOA Circumvented



# ESB

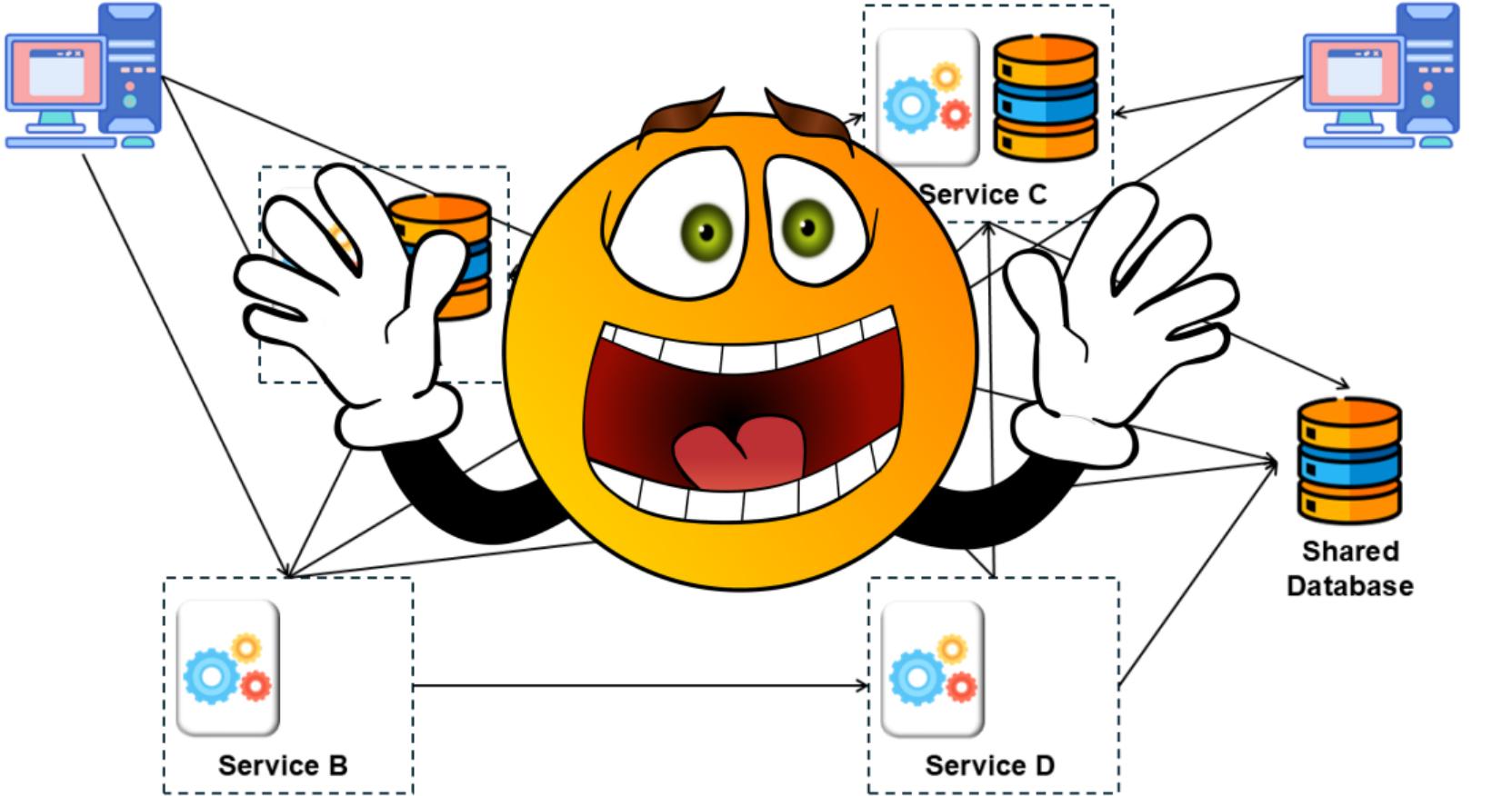


- DB Icon made by setiawanap from [www.flaticon.com](http://www.flaticon.com)
- Cogs Icon made by Design Circle from [www.flaticon.com](http://www.flaticon.com)
- ESB Icon made by vectorsmarket15 from [www.flaticon.com](http://www.flaticon.com)

## ESB Circumvented



## ESB Circumvented



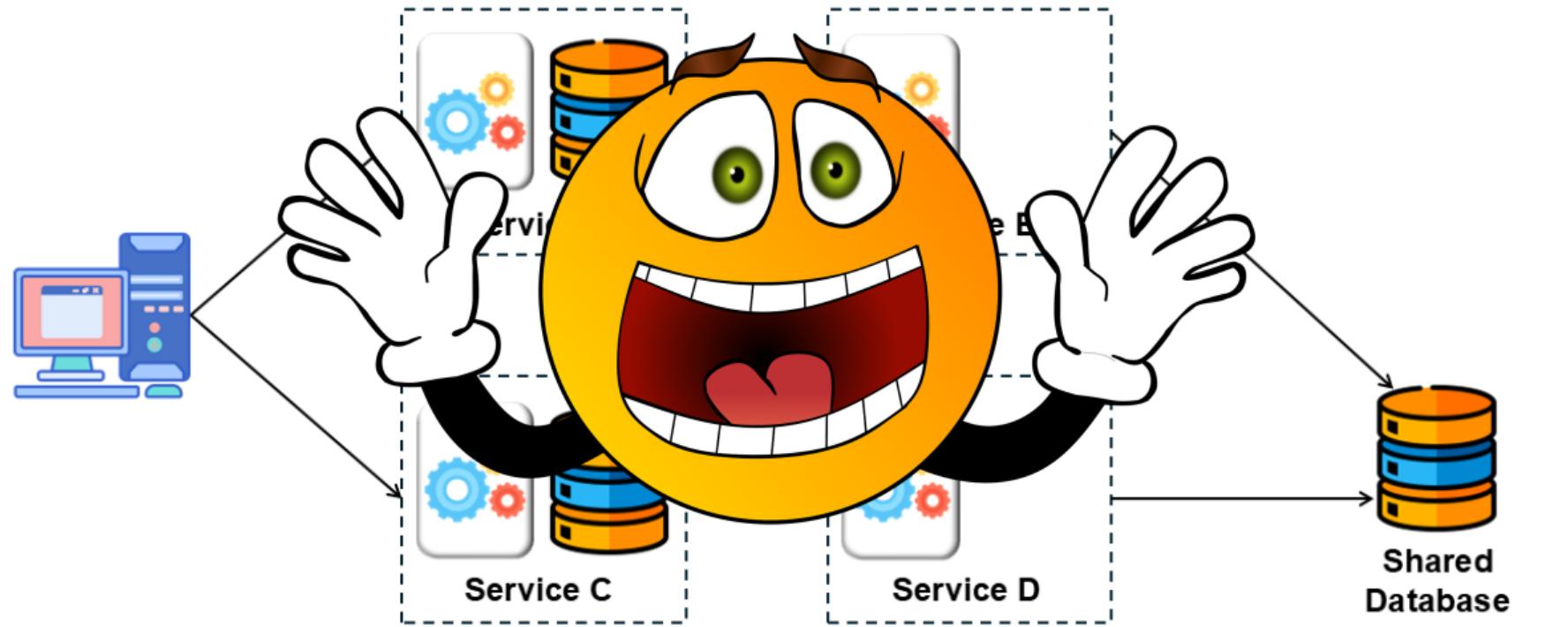
## REST-based Microservices



## REST-based Microservices Circumvented



## REST-based Microservices Circumvented



*Question*

Why does it go wrong?

*Question*

Why does it go wrong?

*Answer*

Getting caught up in the technology!



If you haven't shipped *one* service.  
How will you ship a *dozen*?

# *§ Microservices Architecture*

# Microservices General Topology



- Multiple clients demonstrates common scenario of multiple interfaces to system (e.g. mobile, web).
- Client UIs may be monolithic to provide a rich interface.

# API Layer Components



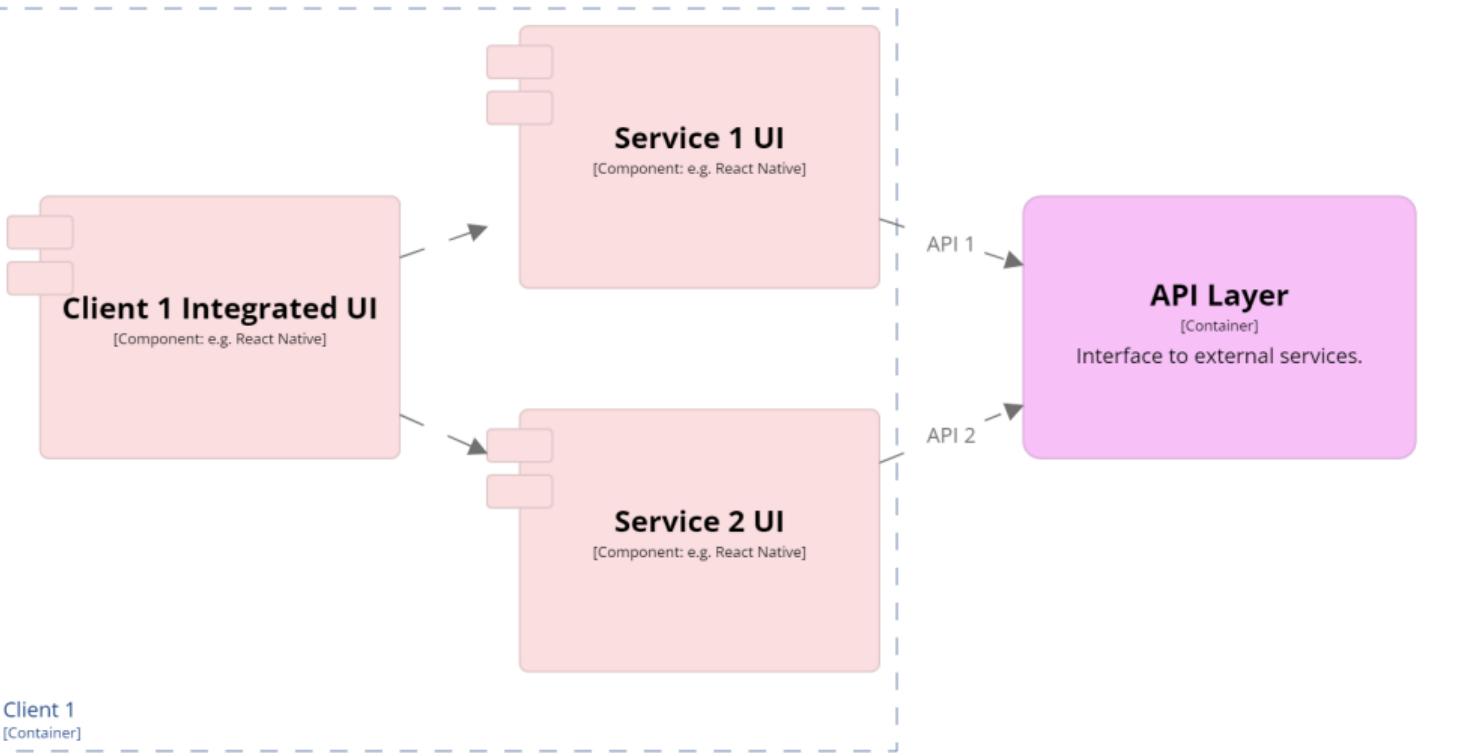
- Each client may use a *different* combination of services
- API layer provides reverse proxy or gateway services
  - See Service-Based Architecture notes & slides
- Typically, Service APIs in the API layer have a one-to-one relationship with Services and are *designed* by the Service teams
- Routing behaviour may not be required

## Service 1 Components



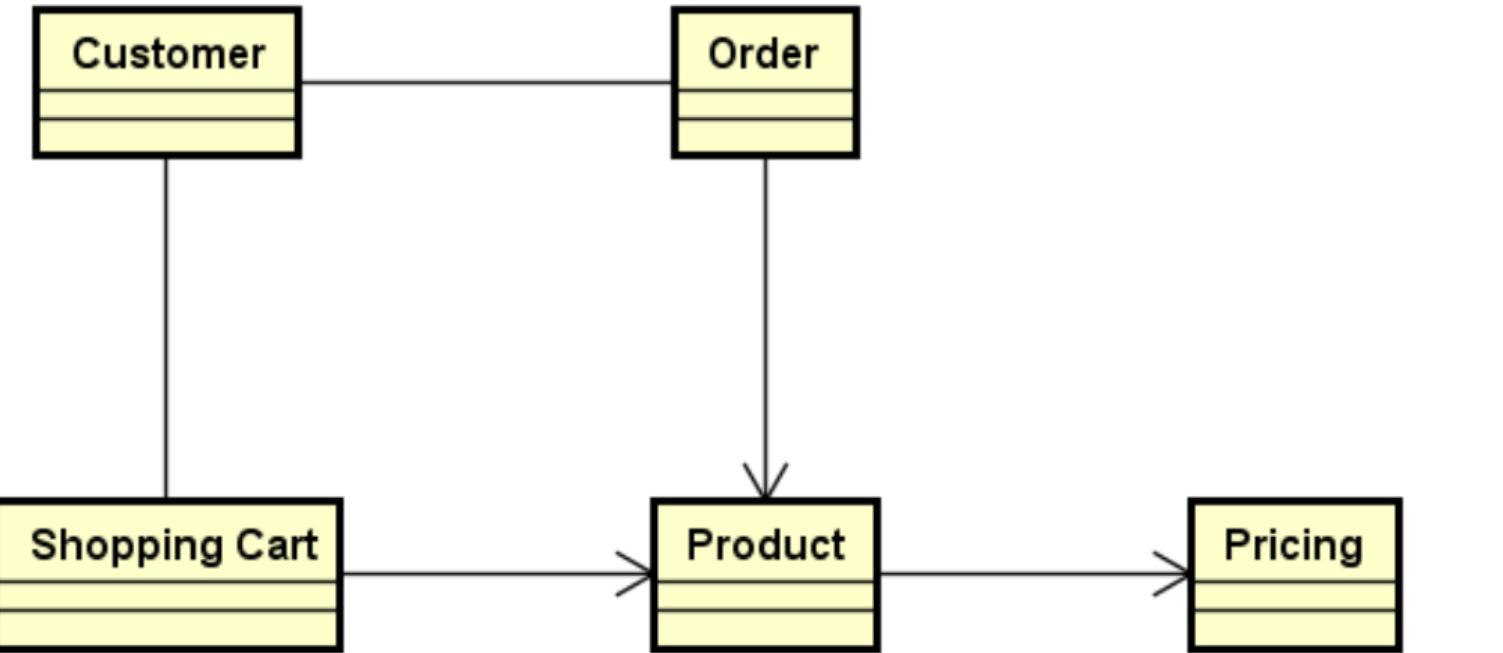
Services 2 & 3 are essentially the same.

## Client with Monolithic UI



- Purist Microservices architecture
  - Each service development team builds their service's UI(s)
- Typically needs some *coordinating activity* in the UI
  - Shown as *Client 1 Integrated UI* in model
- Can still have multiple UIs (e.g. web, mobile, ...)

## Entity Service Anti-Pattern



OO Modelling – Identify entities in the problem domain

- Tend to be stabler / longer-lived parts of the problem domain

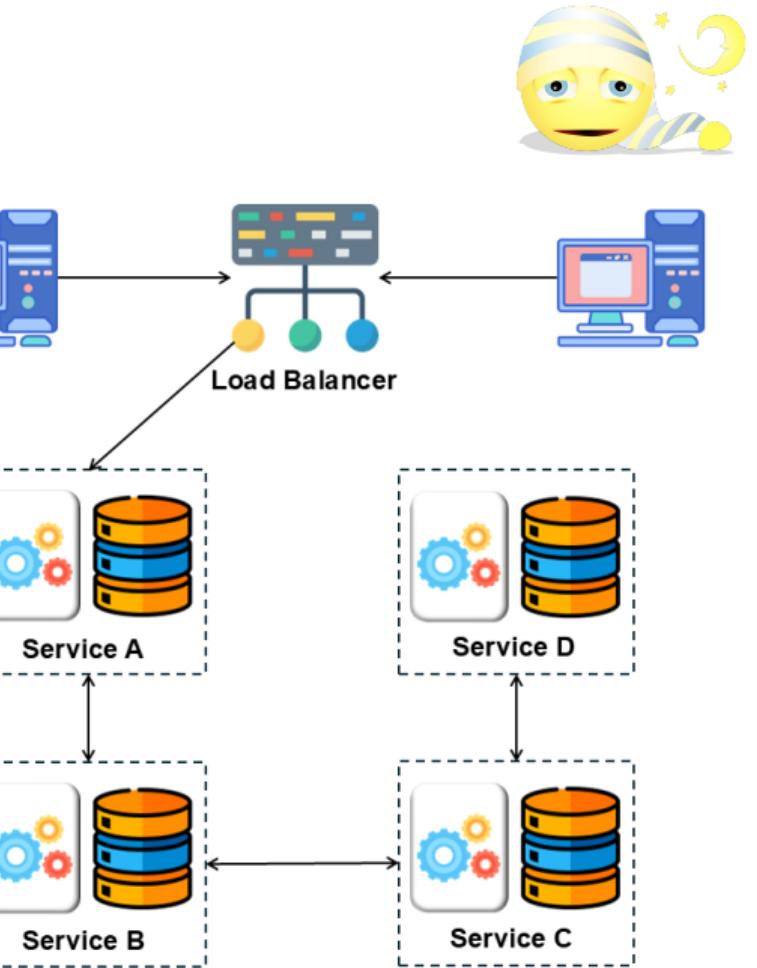
## Entity Service Anti-Pattern



- Intuitive approach – Design services around main entities
- Usually introduces *synchronous* flow of logic between services
  1. Customer places an Order
  2. Order calls Product to get details
  3. Product calls Pricing to determine price

## Problem with Synchronous Logic

- Potential chain of failure
- Failures are multiplied
- Latency is multiplied
- Deployment is hard
- Scaling is hard
- Lazy design



- DB Icon made by setiawanap from [www.flaticon.com](http://www.flaticon.com)
- Cogs Icon made by Design Circle from [www.flaticon.com](http://www.flaticon.com)
- Load Balancer Icon made by DinosoftLabs from [www.flaticon.com](http://www.flaticon.com)
- Chains of request (synchronous logical flow) don't scale.

# *§ Domain Driven Design*

### *Definition 0.* Bounded Context

Logical boundary of a domain where particular terms and rules apply consistently.



*Definition 0.* Service Cohesion Principle

Services are cohesive business processes.

### *DDD Consideration*

Services are *bounded contexts*.

Bounded contexts are not necessarily *services*.

## *Large Bounded Contexts*

Bounded context may be too large to be a single service.

Split it into services that are *independent* sub-processes.

*Definition 0.* Service Independence Principle

Services should not depend on the implementation of other services.

*Corollary 0.* Low Coupling

There should be minimal coupling between services.

*Corollary 0.* No Reuse

Avoid dependencies between services.

Do not reuse components between services.

## *Restaurant Examples*

### Bad Restaurant

- Arrive: Wait in queue to be seated
- Wait for menu: Server has to go get board with menu
- Ask about special: Server has to go ask chef
- Order items: Server takes order to kitchen
- Server returns, an item is not available
- Change order: Server takes order to kitchen
- Server waits for order – constantly asking if its ready
- Server brings everything to table together
  - Drinks, Entrée, Main, Dessert, ...
- ...

## *Restaurant Examples*

Bad Restaurant

Good Restaurant

- Describe normal experience at a restaurant

## *Lessons Learnt*

Chains of request don't *scale*.

People who have the *information* to do their job are more effective.

Service boundaries that follow these *principles* work better.

## *Human Shaped Microservices*

Look at *behaviour* rather than entities.

- Who & How
- Not the Thing

From Damian McLennan

What would people *do*?

How would they *communicate*?

## *Bounded Domains Implications*

- Duplication
  - Entities specialised for domain
    - Requires mapping of entity data between domains

## *Bounded Domains Implications*

- Duplication
  - Entities specialised for domain
    - Requires mapping of entity data between domains
  - Should everything be duplicated?

## *Bounded Domains Implications*

- Duplication
  - Entities specialised for domain
    - Requires mapping of entity data between domains
  - Should everything be duplicated?
    - What about common services (e.g. logging, ...)?

## *Bounded Domains Implications*

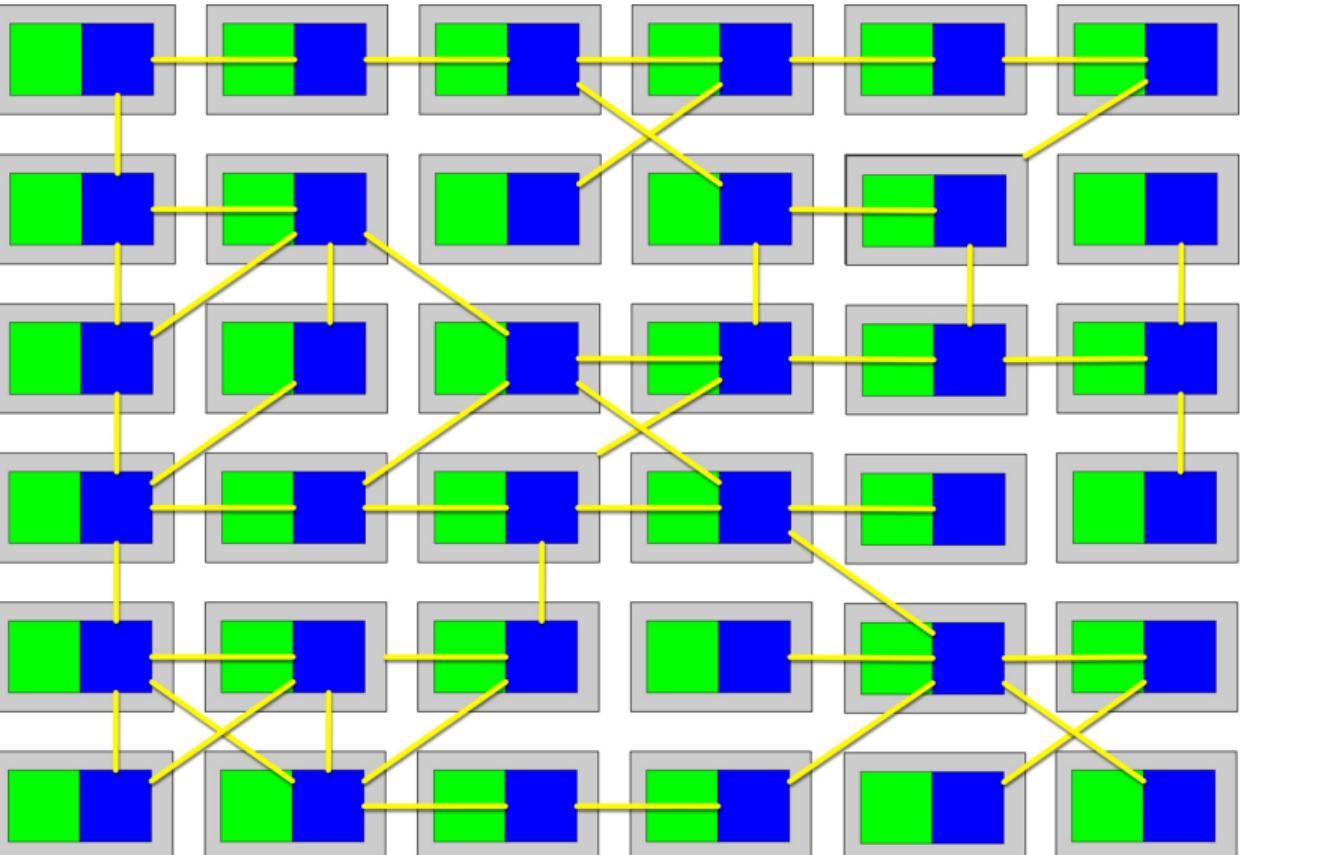
- Duplication
  - Entities specialised for domain
    - Requires mapping of entity data between domains
  - Should everything be duplicated?
    - What about common services (e.g. logging, ...)?
- Heterogeneity
  - Services can use different implementation technologies

# *§ Microservices Design Options*

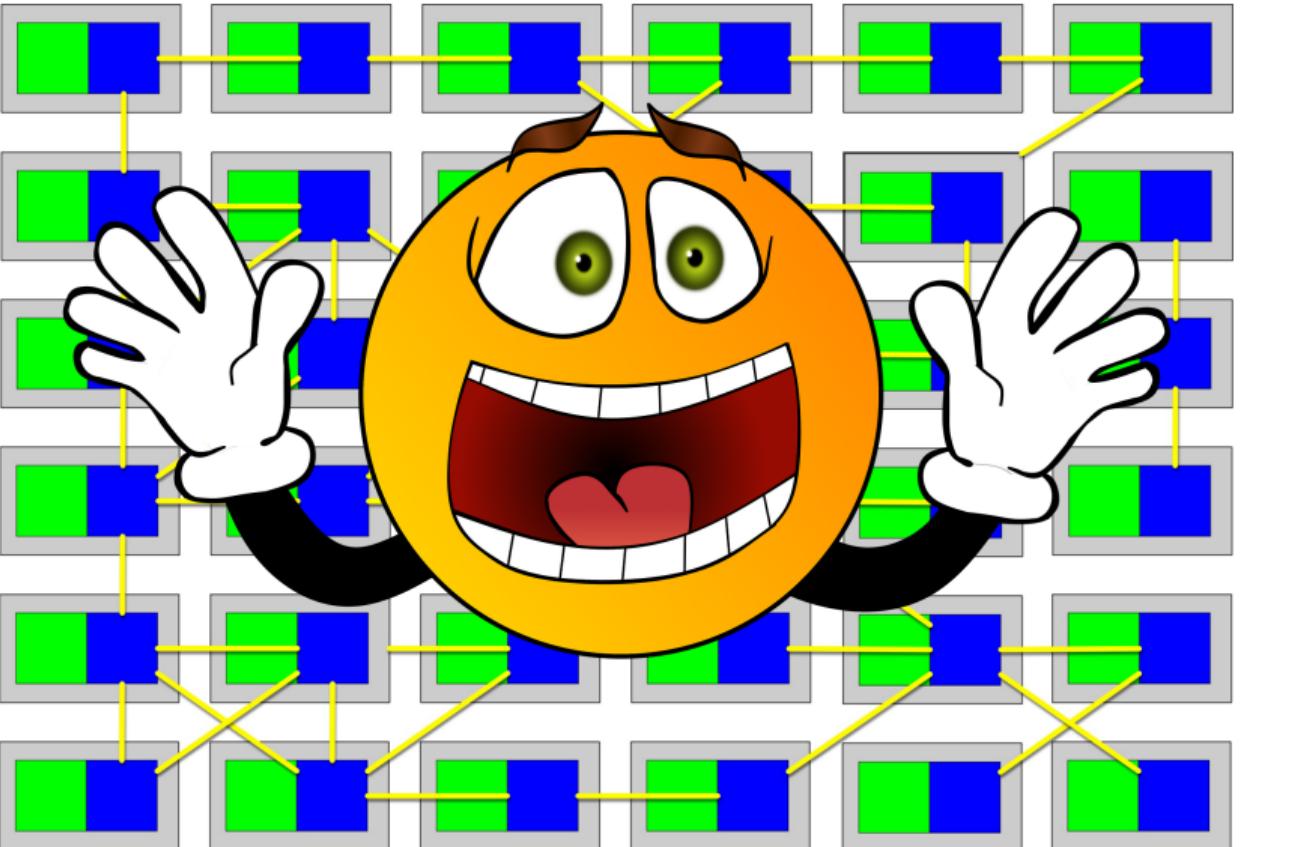
# Service Plane



## Service Mesh



## Service Mesh

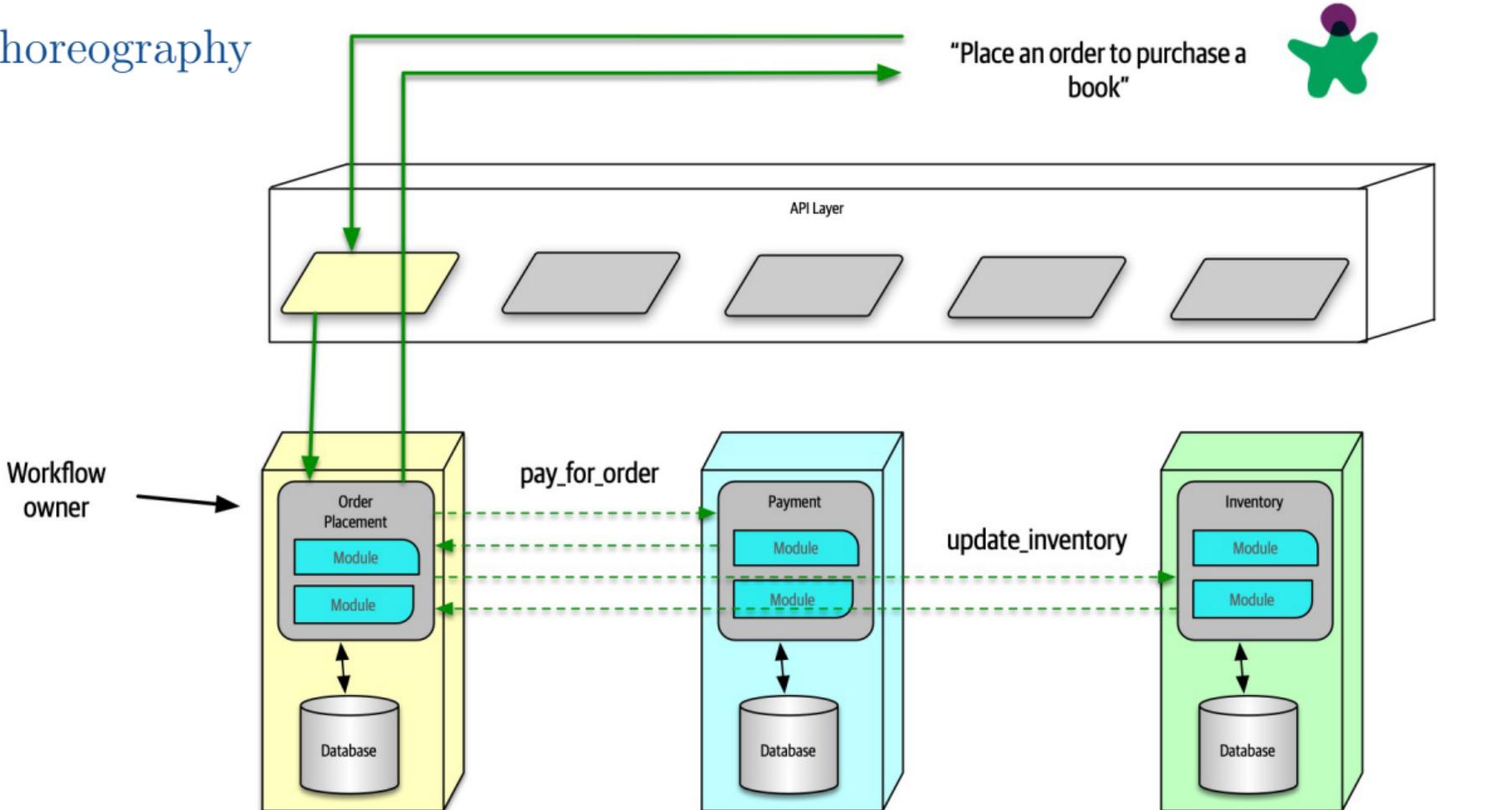


## *Choreography & Orchestration*

Choreography Similar to event-driven *broker*

Orchestration Similar to event-driven *mediator*

## Choreography

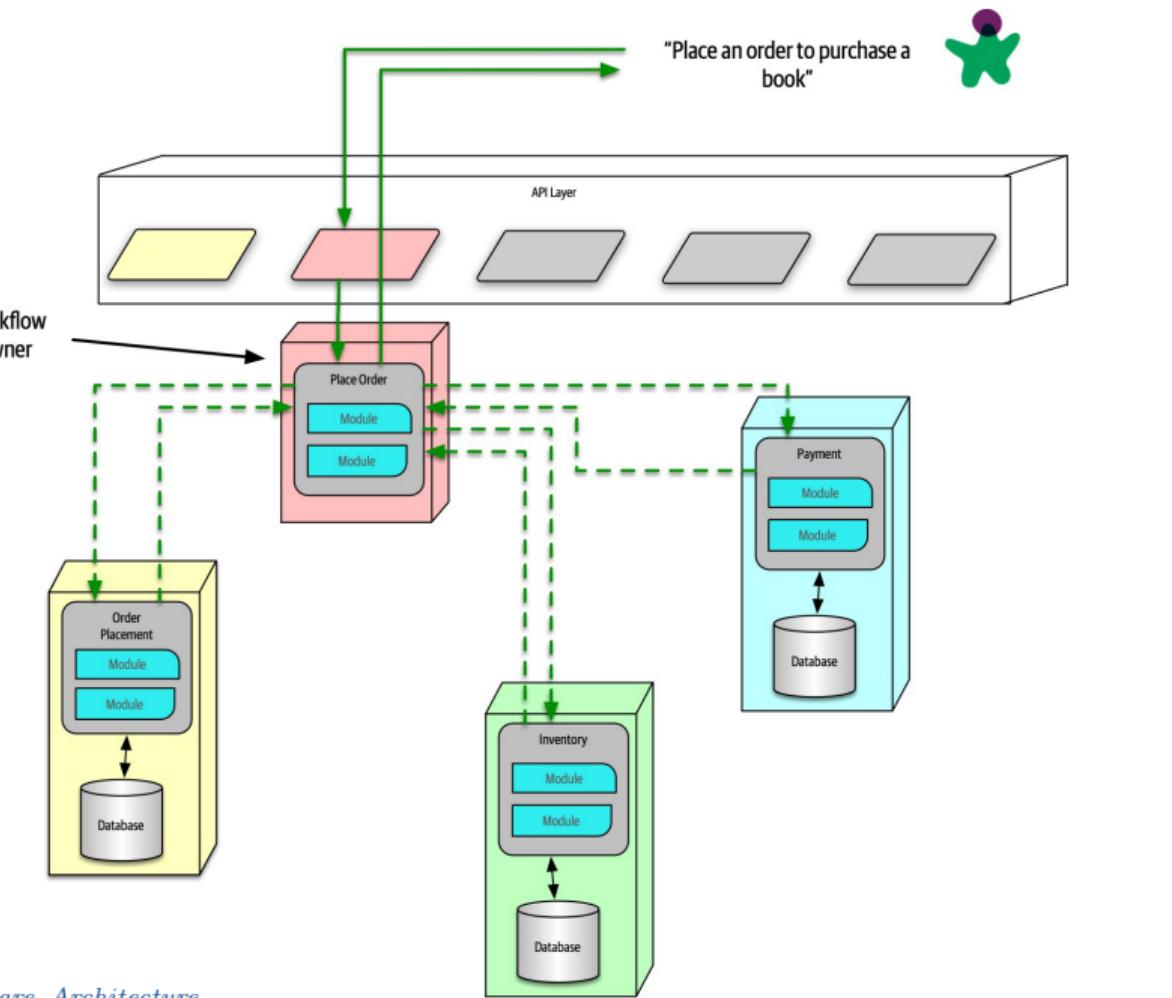




# Purchase Product Dynamic Diagram



# Orchestration



*Question*

How bad is coupling with choreography or  
orchestration?

*Question*

How bad is coupling with choreography or orchestration?

*Answer*

For a large system, *very bad*.

In 2017, Uber had over 1400 services ... consider how bad coupling would be with either approach.

*Question*

How do we scale large microservices-based systems?

*Question*

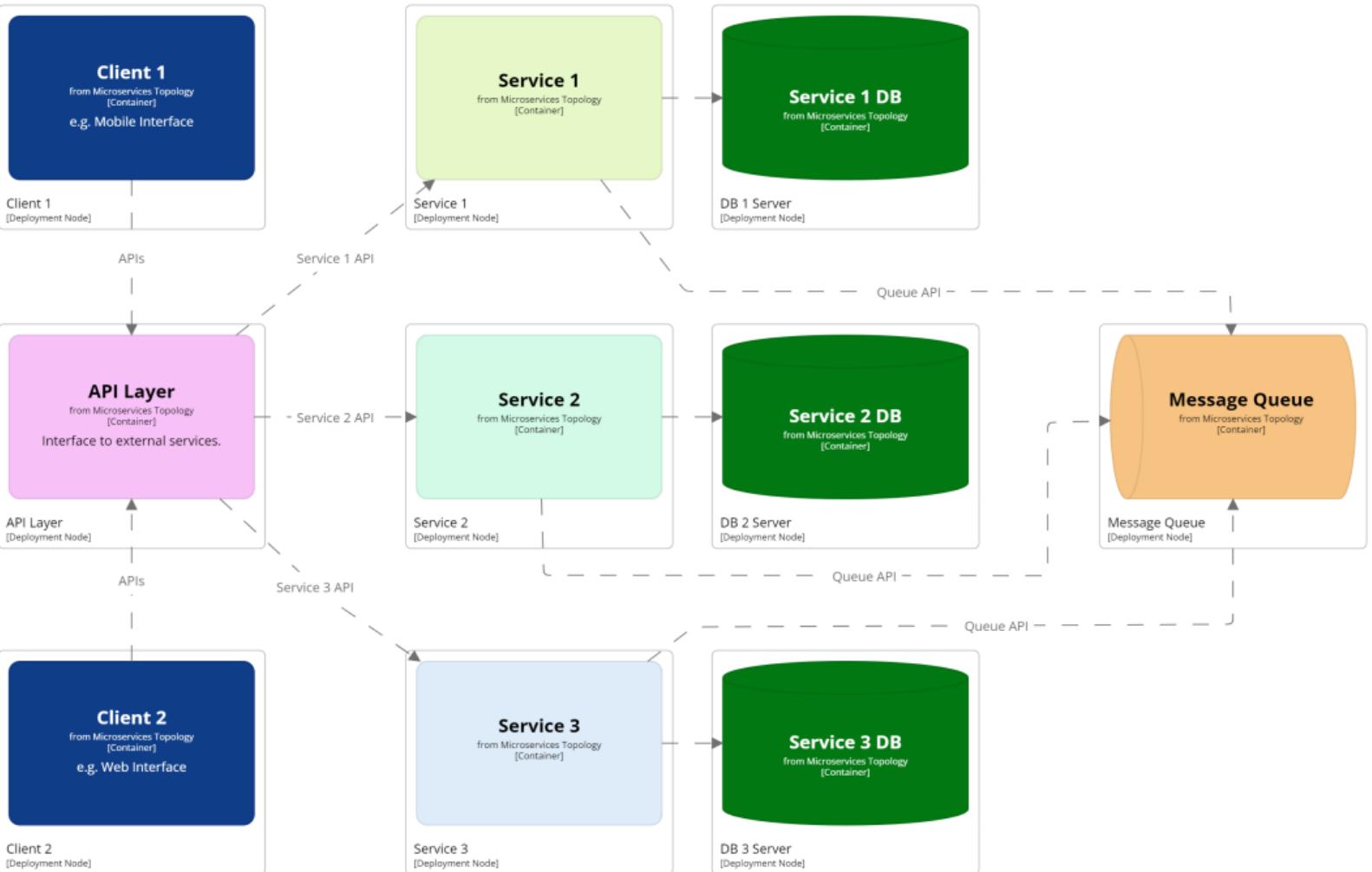
How do we scale large microservices-based systems?

*Answer*

Combine architectural patterns

- Event-Driven Microservices

# Microservices with Event Queue



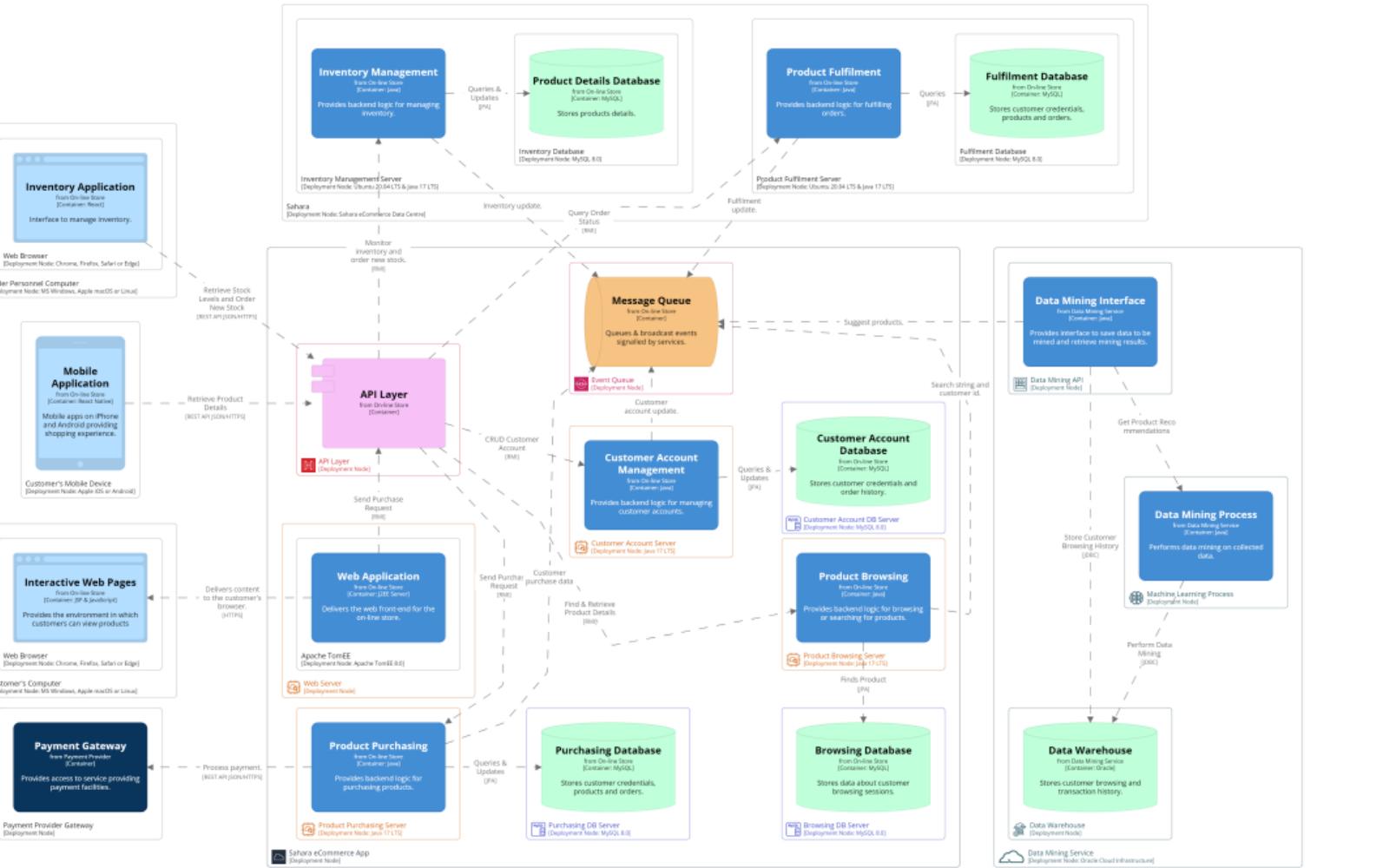
- Use the tried and true *Observer* pattern, with the *Event-Driven Architecture* pattern.
- Clients send *commands* telling services to do something.
- Services *publish events* indicating what they have been done.
- Services *listen for events* to decide how to coordinate their part of the system behaviour.

## Service 1 Components with Event Queue



Services 2 & 3 are essentially the same.

# Sahara using an Event Queue



- Sahara eCommerce system as a simple microservices architecture, using event-driven messaging between services.
- Services publish events indicating what they have been done.
- Also an example of a multi-tenanted system built across in-house servers, AWS and OCI.

*Question*

Are *browsing* and *purchasing* separate contexts?

*Question*

Are *browsing* and *purchasing* separate contexts?

*Answer*

- Are they a single business process or different processes?
- Do they share much or little data?

- Probably different business processes, but possibly the same context.
- If separate services, browse needs to send an event for every change to the shopping cart, and purchase needs to listen for these.
- Possibly merge into one service, as one context.

*Question*

- What about *inventory management* and *browse*?
- How do they maintain a consistent product database?

## *Model Behaviour*

- *Commands & Events* describe *behaviour*
- They will help you better model your *domain*
- Leading to *independent, scalable* services

## Pros & Cons

Modularity



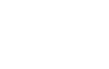
Extensibility



Reliability



Interoperability



Scalability



Security



Deployability



Testability



Simplicity

