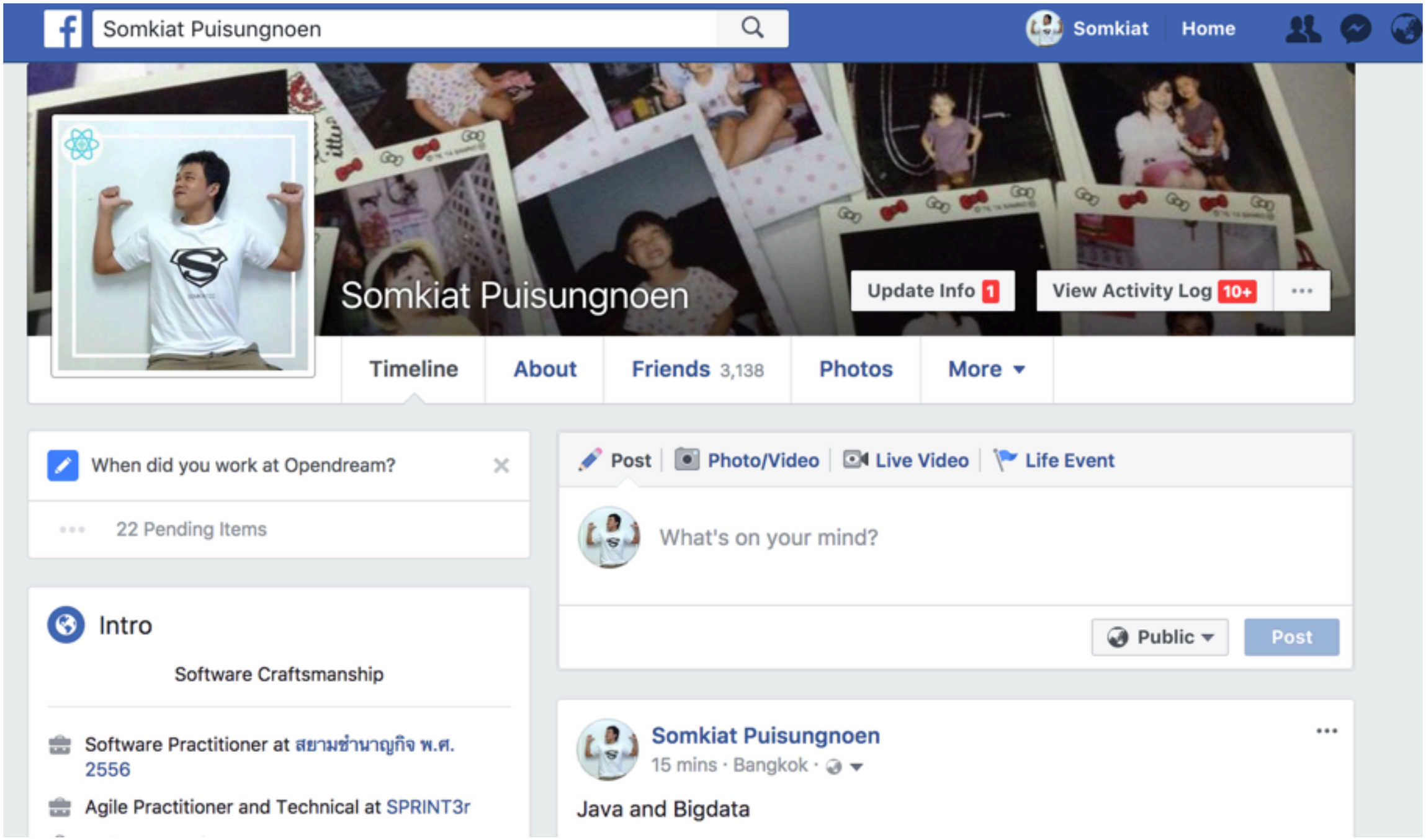
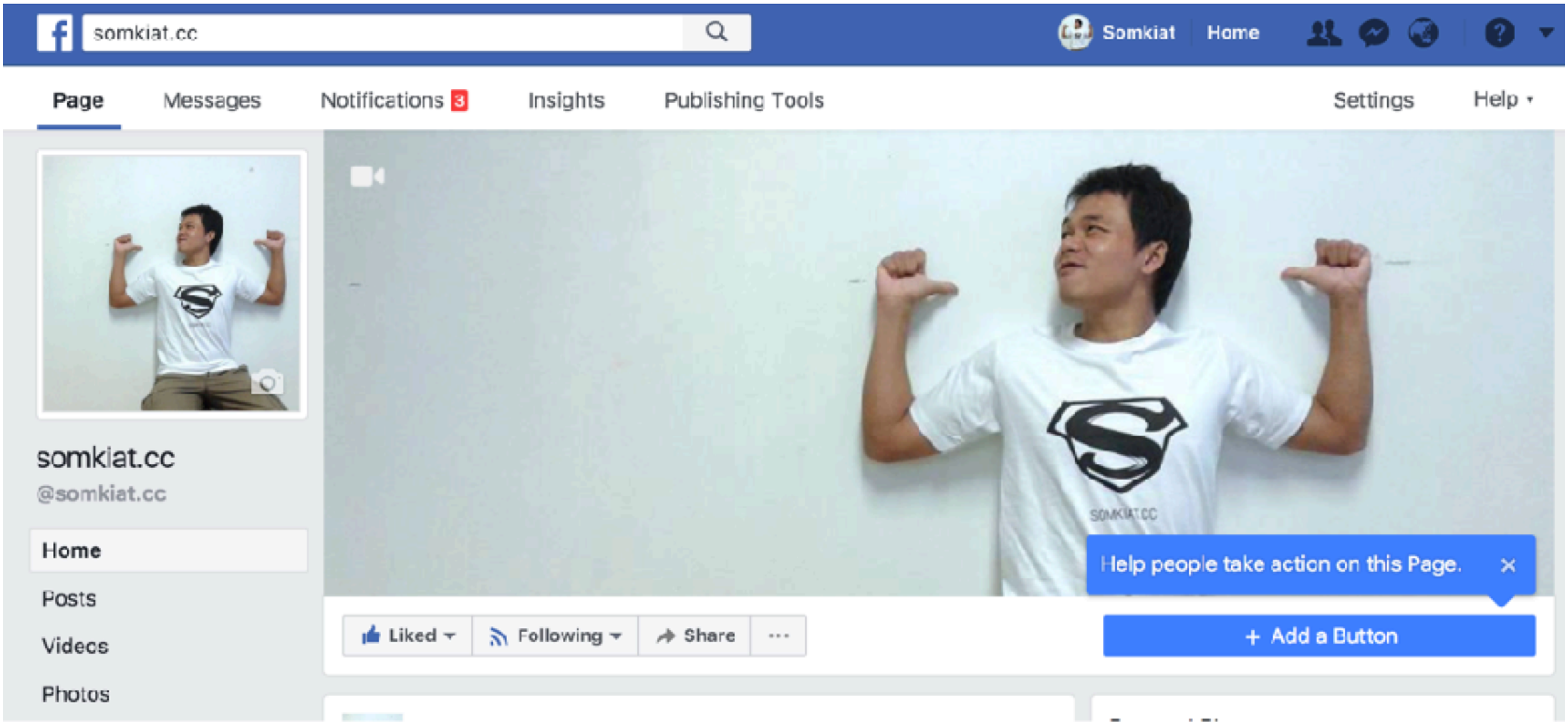
สยามชำนาญกิจ
SIAM CHAMNAN KIT

Microservices Workshop





Microservice

Module 1 : Design

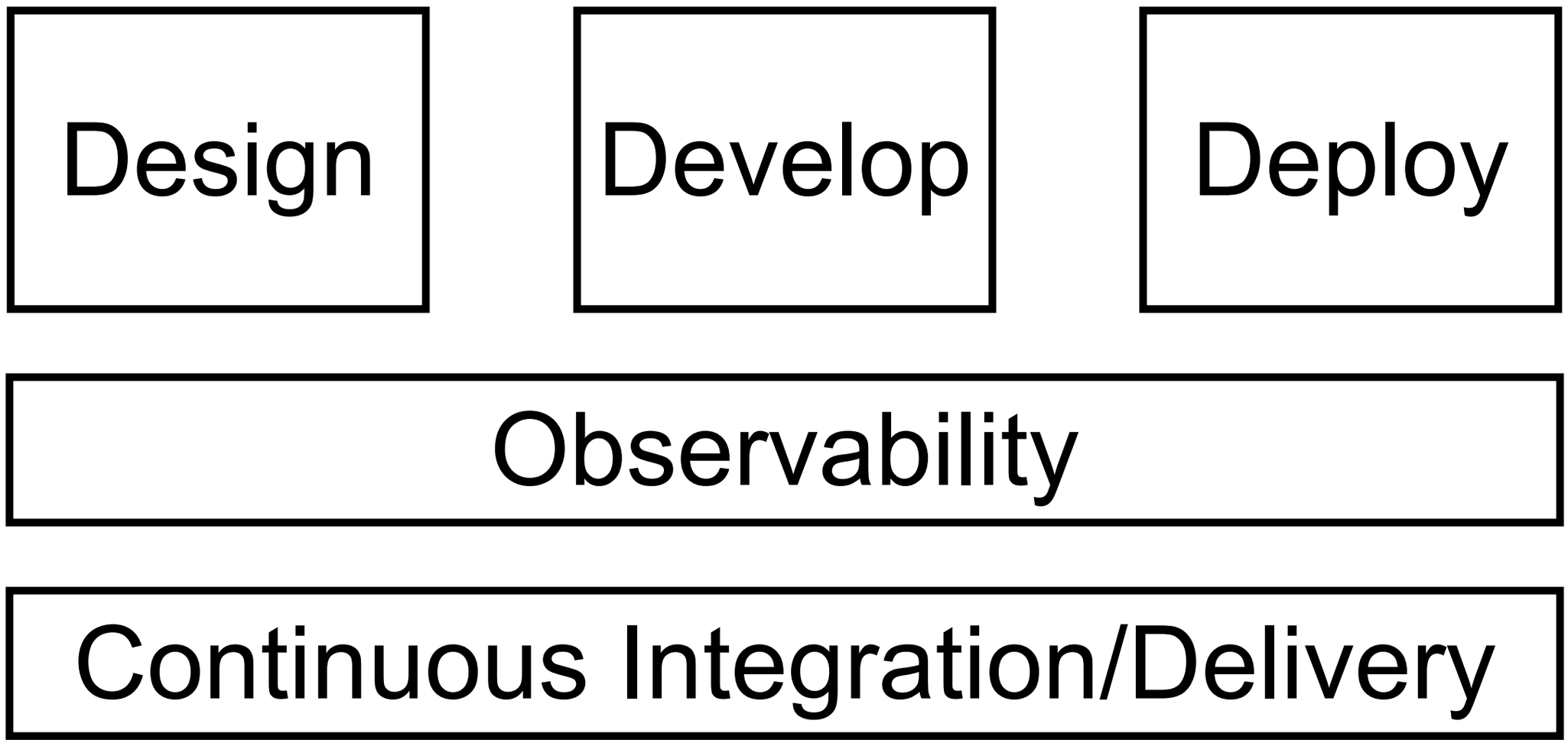
Cloud Native Application
Evolution of architecture
Microservice architecture
How to decompose app to Microservice
Communication between service
Data consistency
Workshop

Module 2 : Develop + Testing

Recap Microservice
Properties of Microservice
Microservice 1.0 - 4.0
How to develop Microservice ?
How to test Microservice ?
12-factors app
Observable services

Module 3 : Deploy

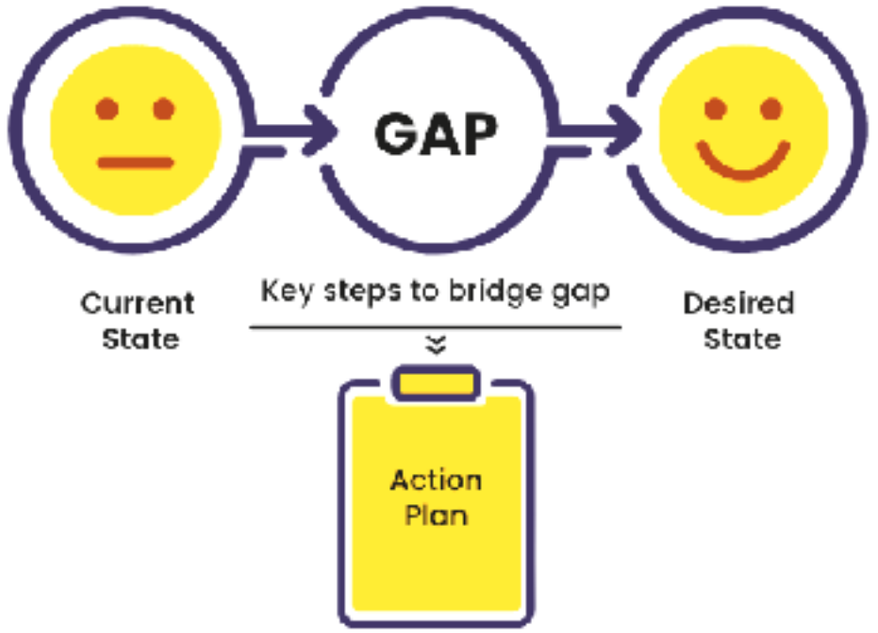
How to deploy Microservice ?
Continuous Integration and Delivery
Practices of Continuous Integration
Deployment strategies
Working with containerization (Docker)
Workshop



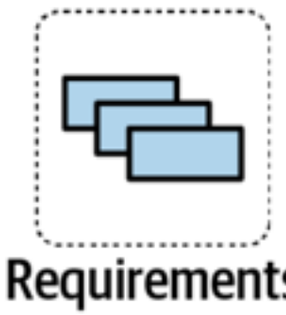
“ Don’t use Microservice ”

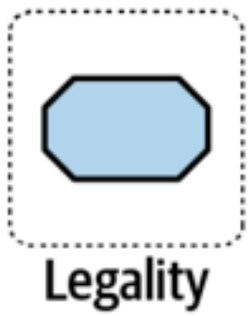
Software **Architecture**

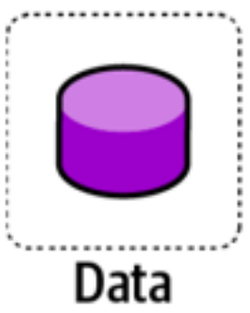
Pain Points ?

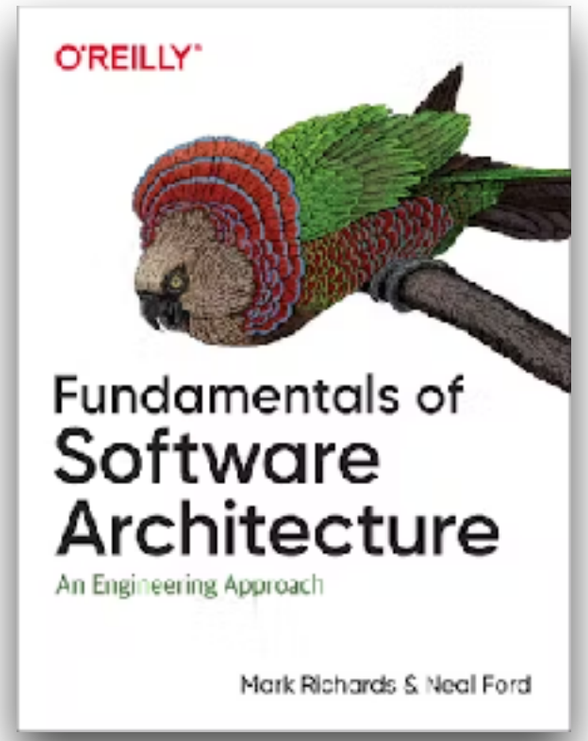


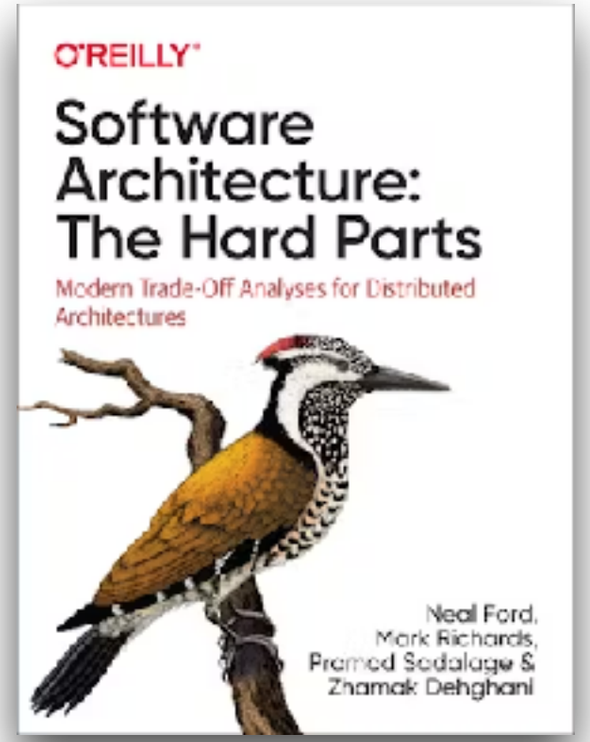
Software Architecture

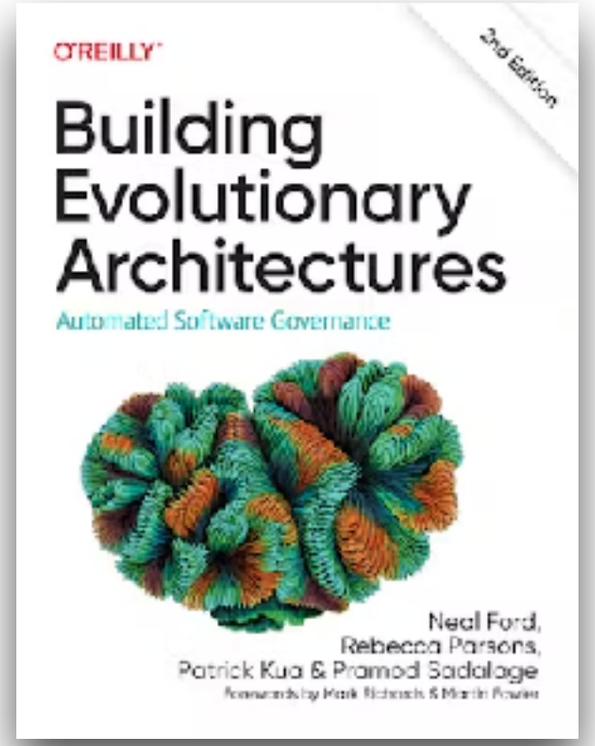


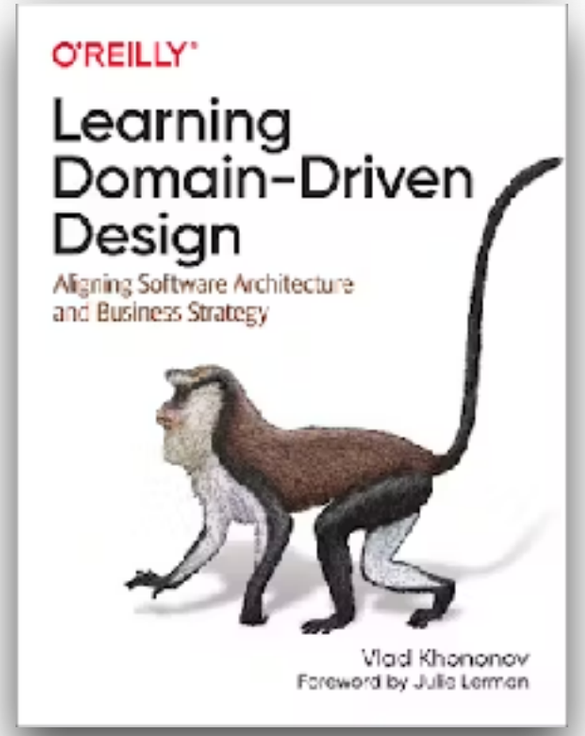


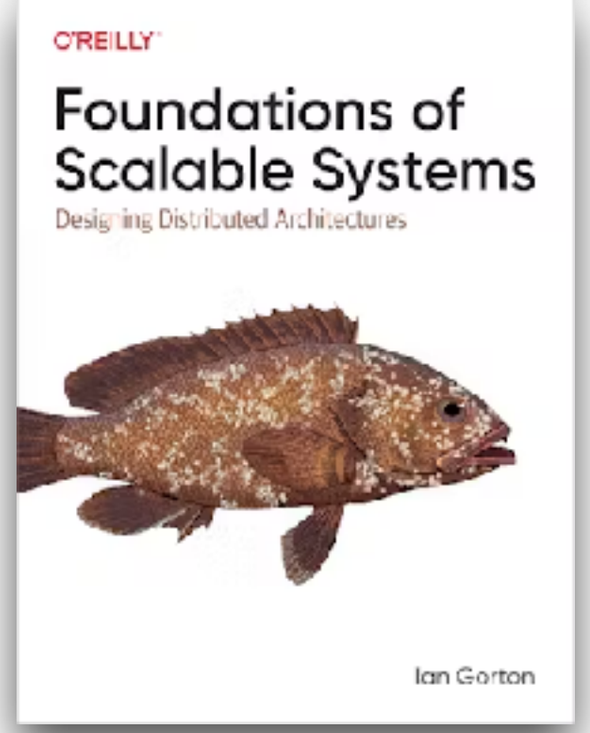


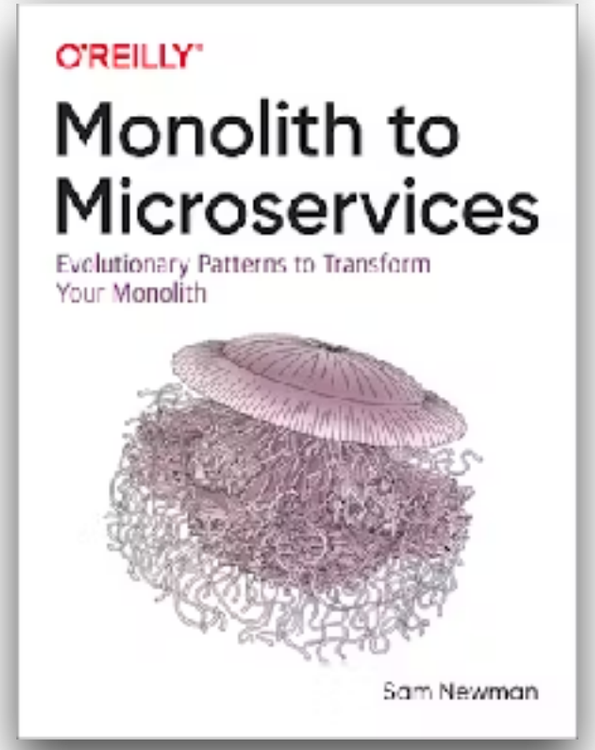




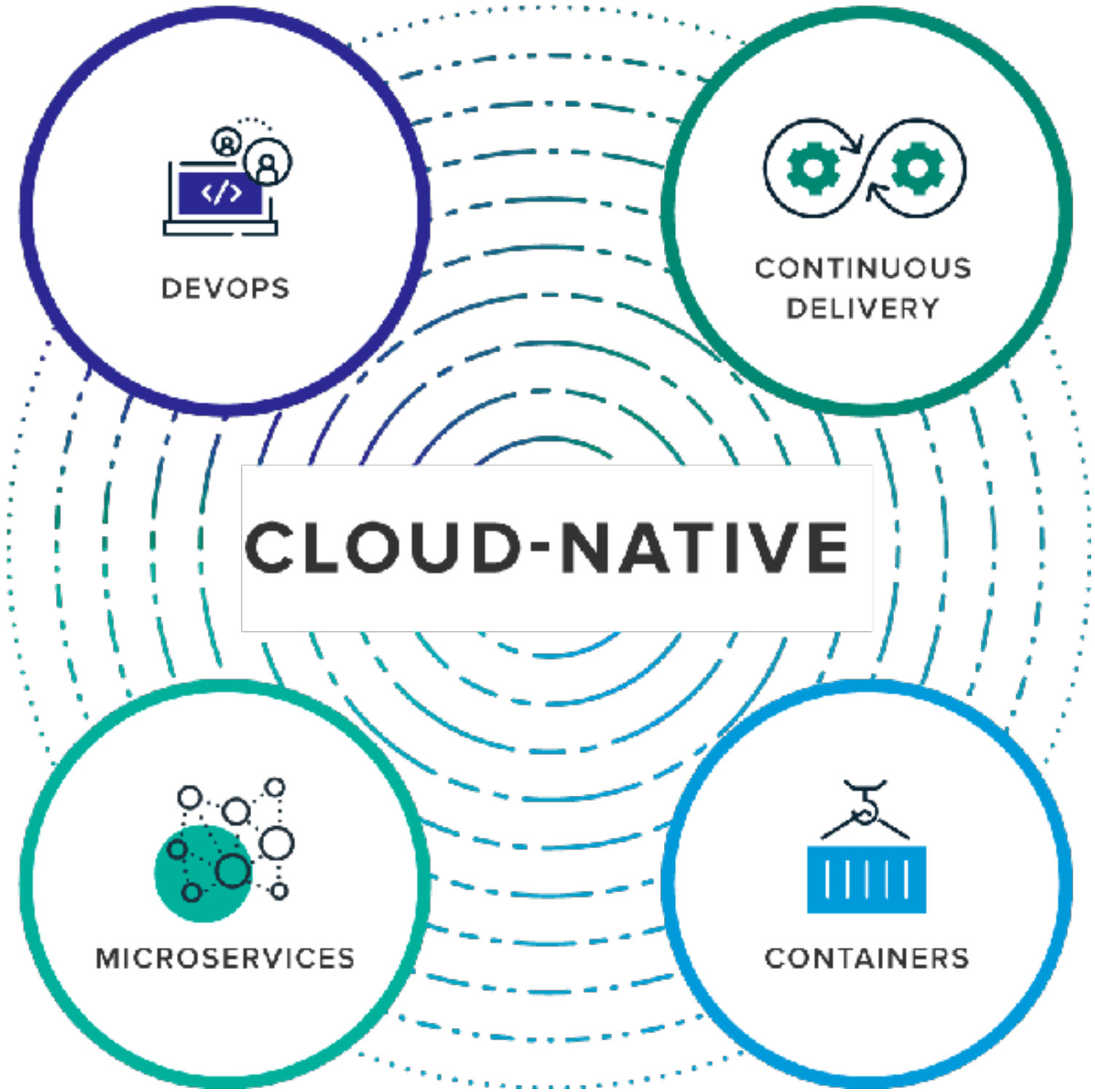








Module 1 : Design



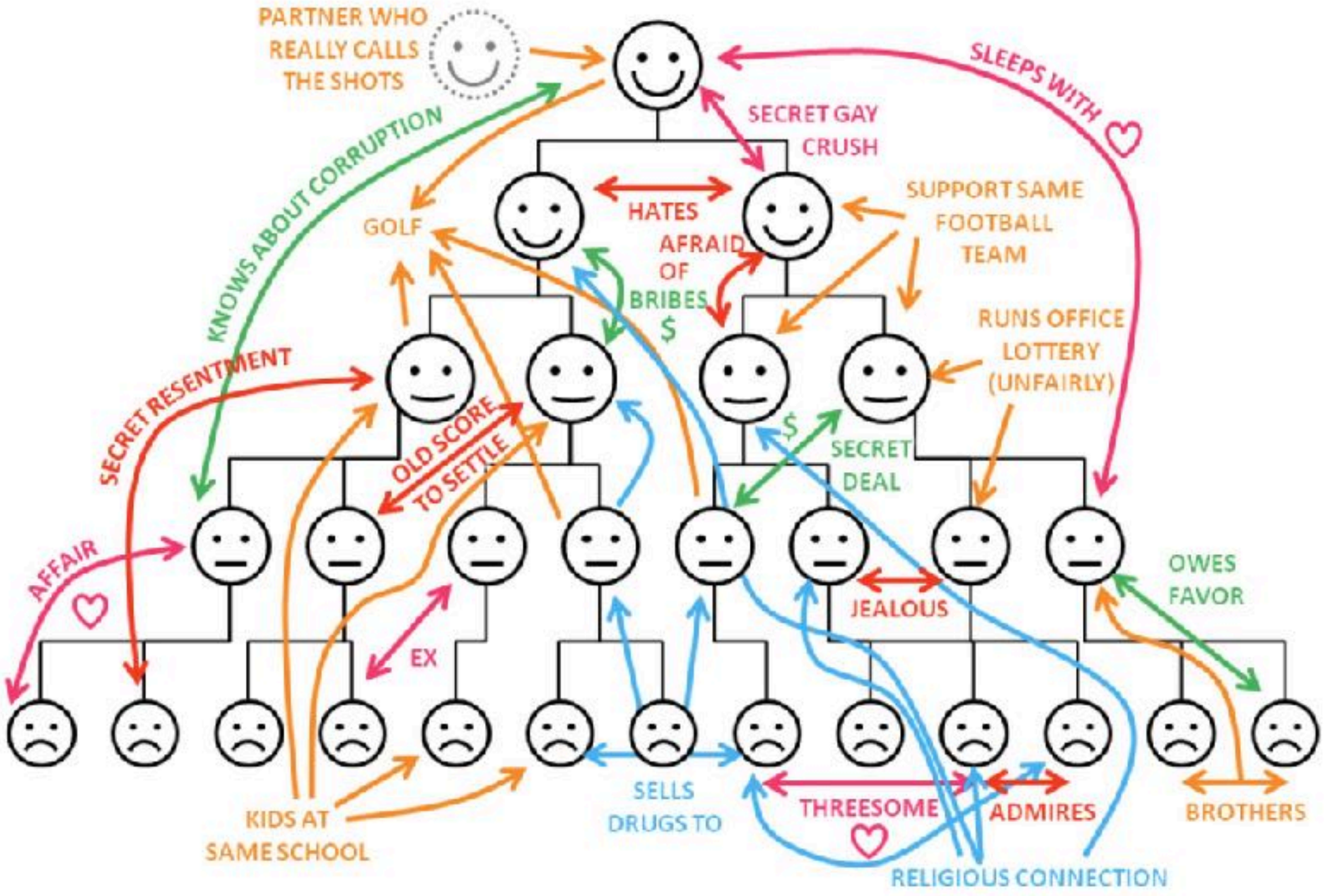
https://pivotal.io/cloud-native

Evolution of Architecture

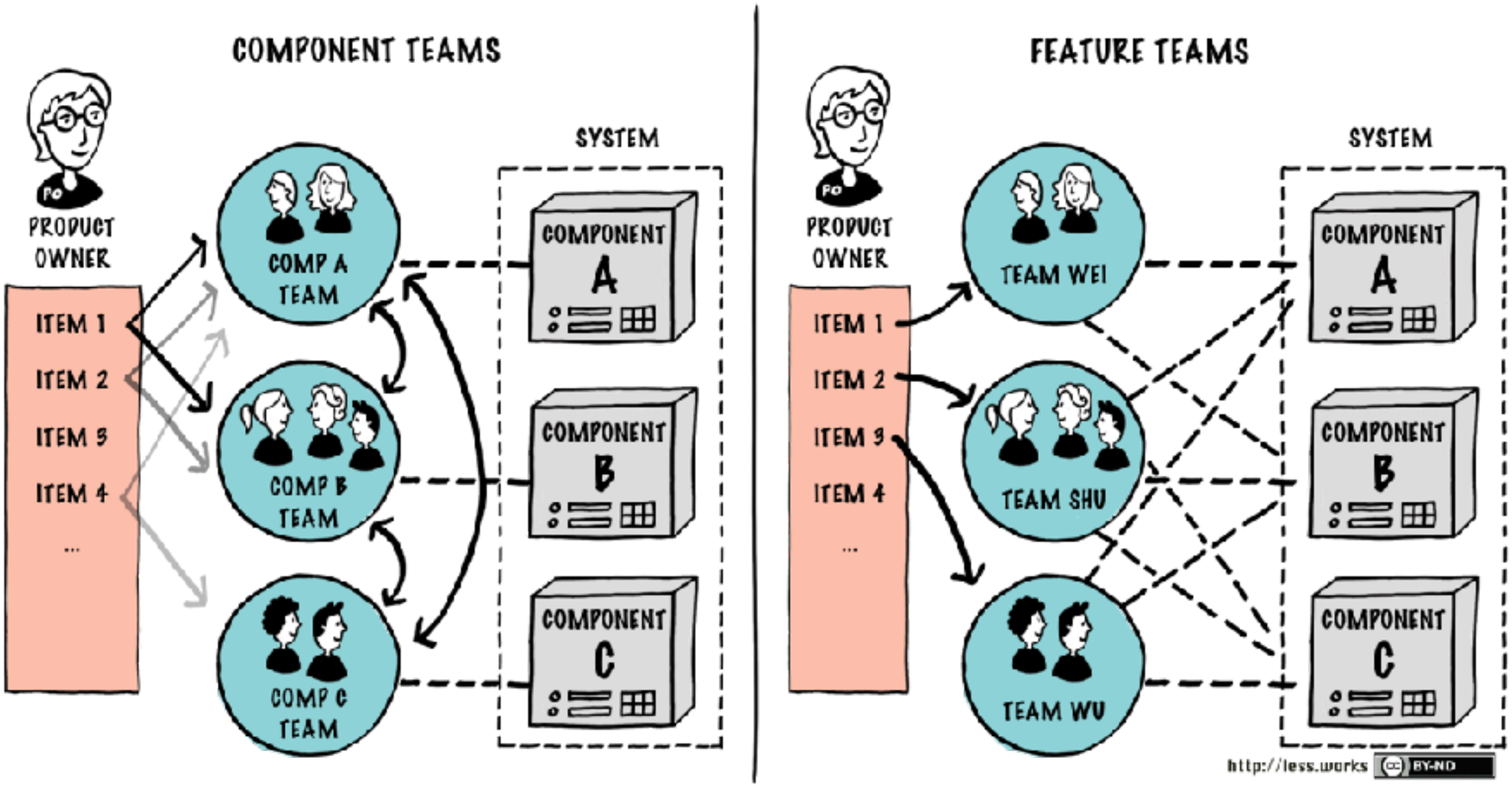
Any organization that designs a system will produce a design whose structure is a copy of the organization's communication structure.

-- Melvin Conway

Organization Structure



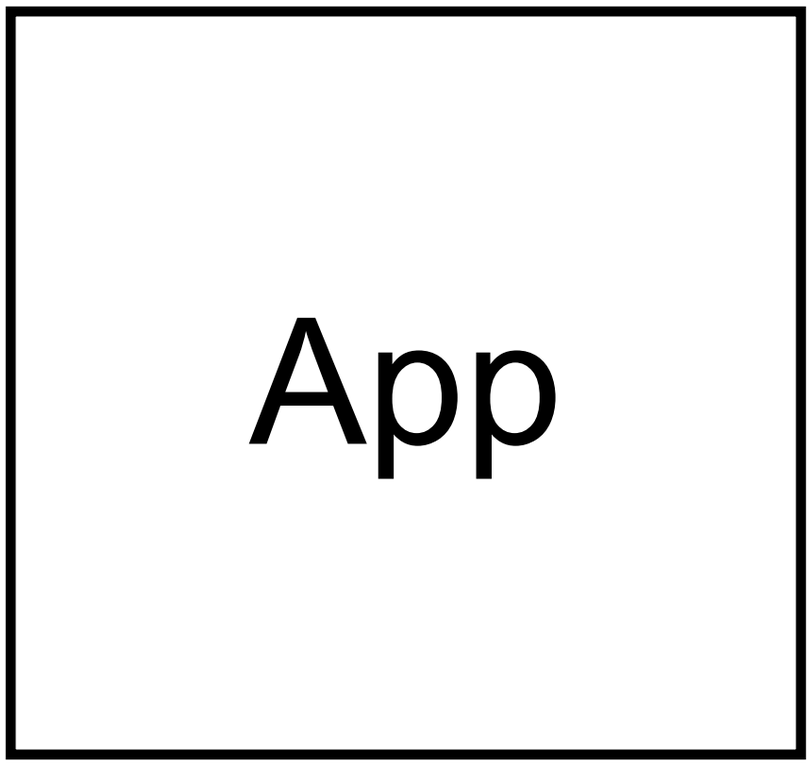
Component vs Feature teams



https://less.works/less/structure/feature-teams

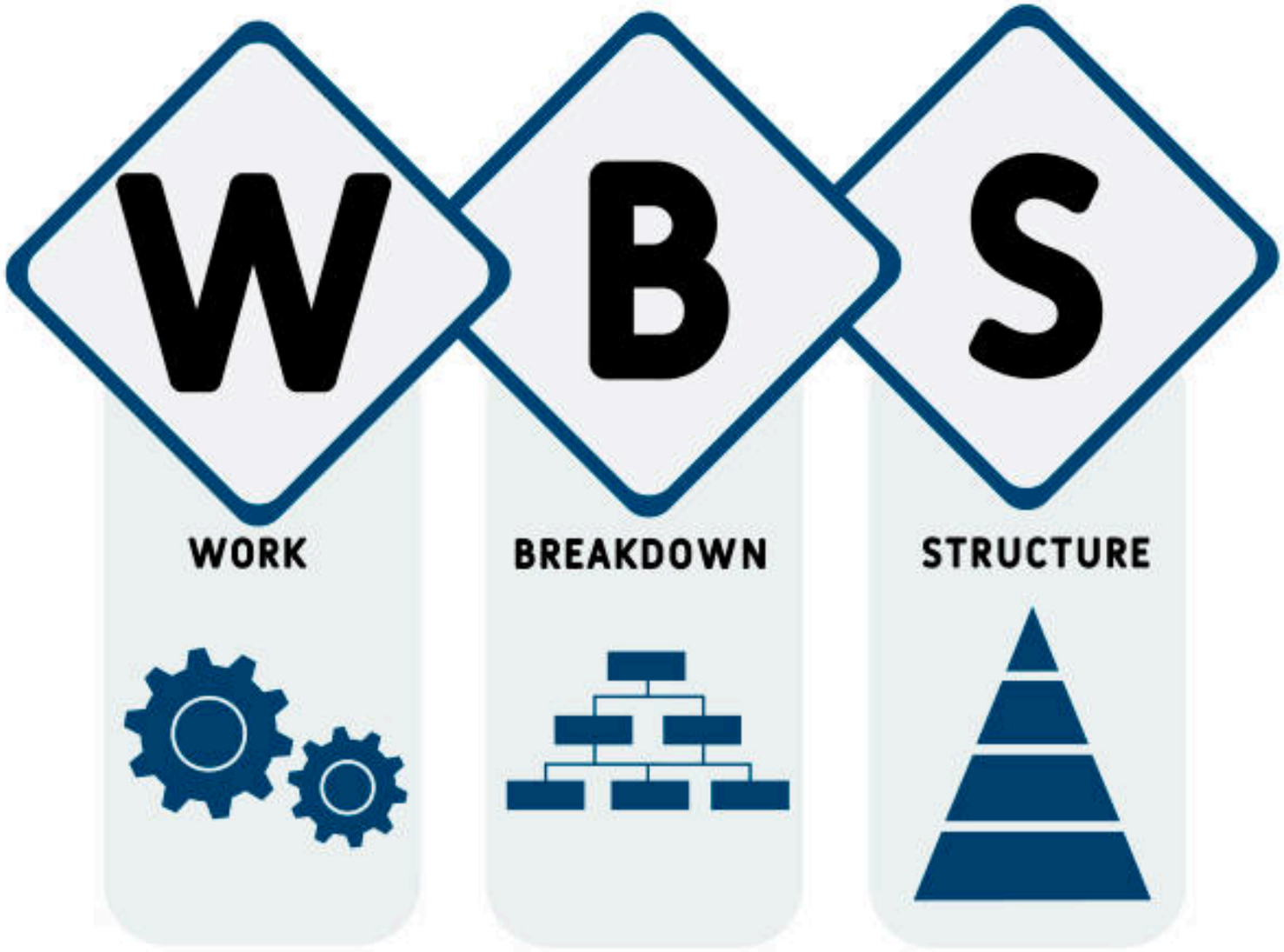
Let’s Start

Monolith

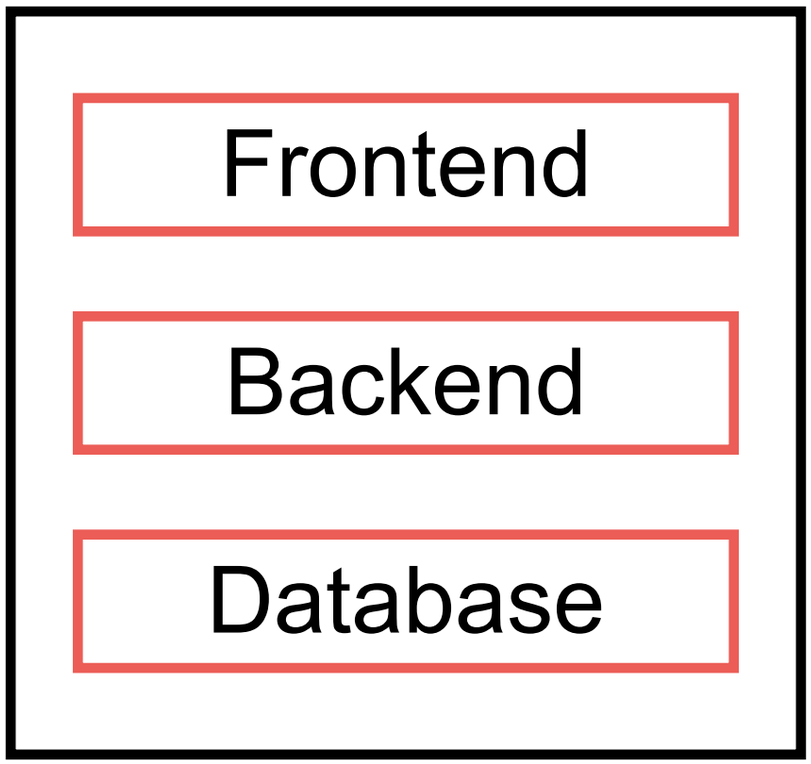


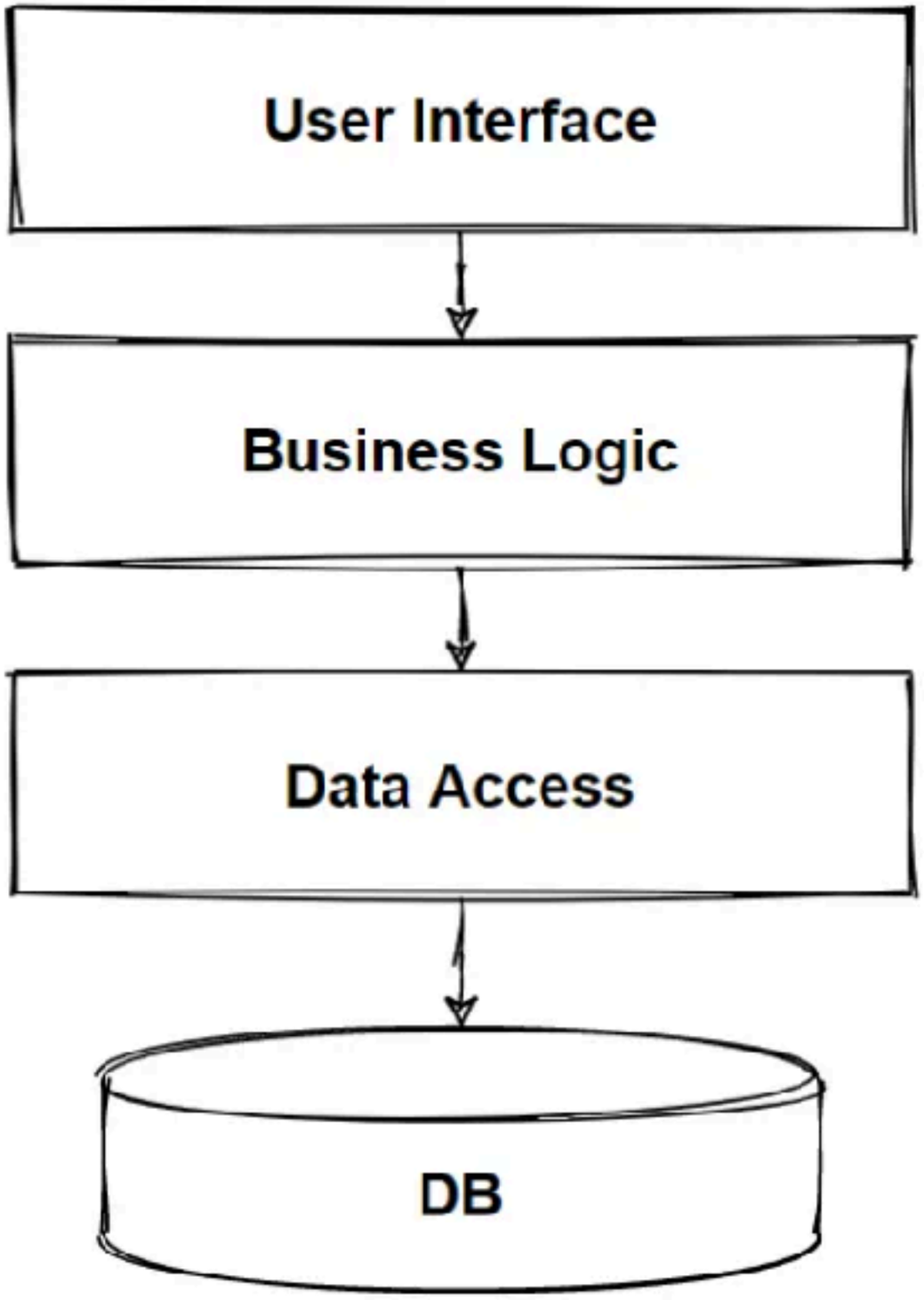
Work Breakdown Structure

Work Breakdown Structure

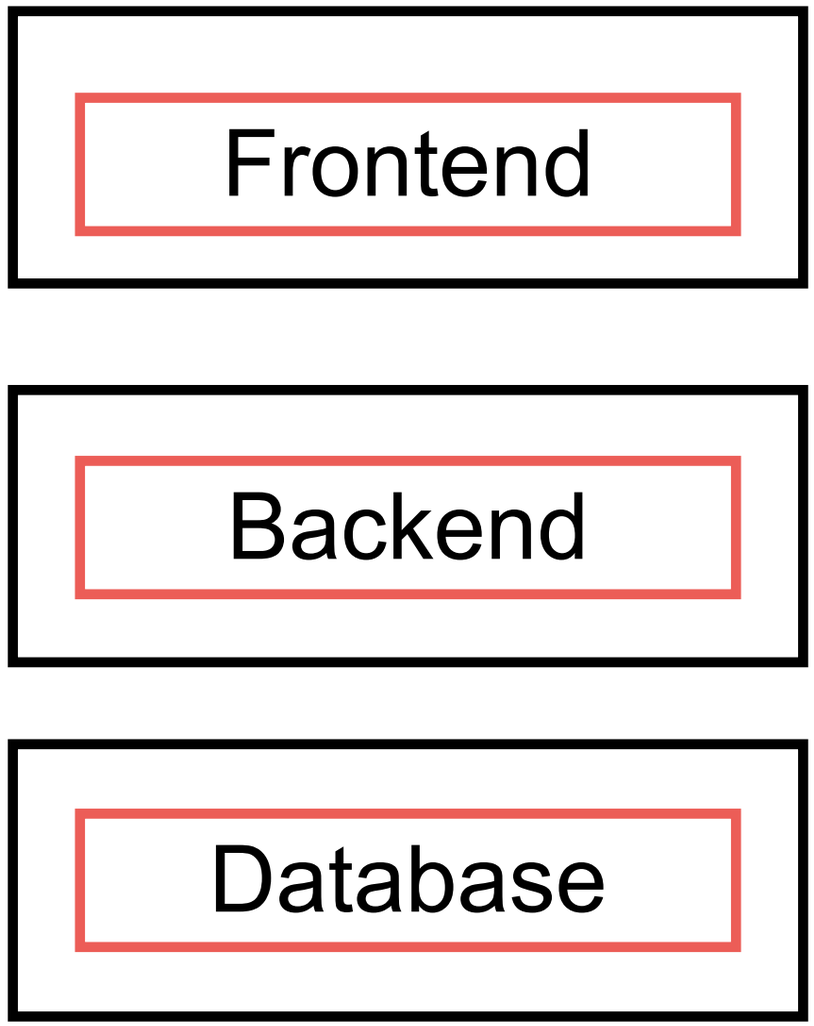


Layers (inter-process)

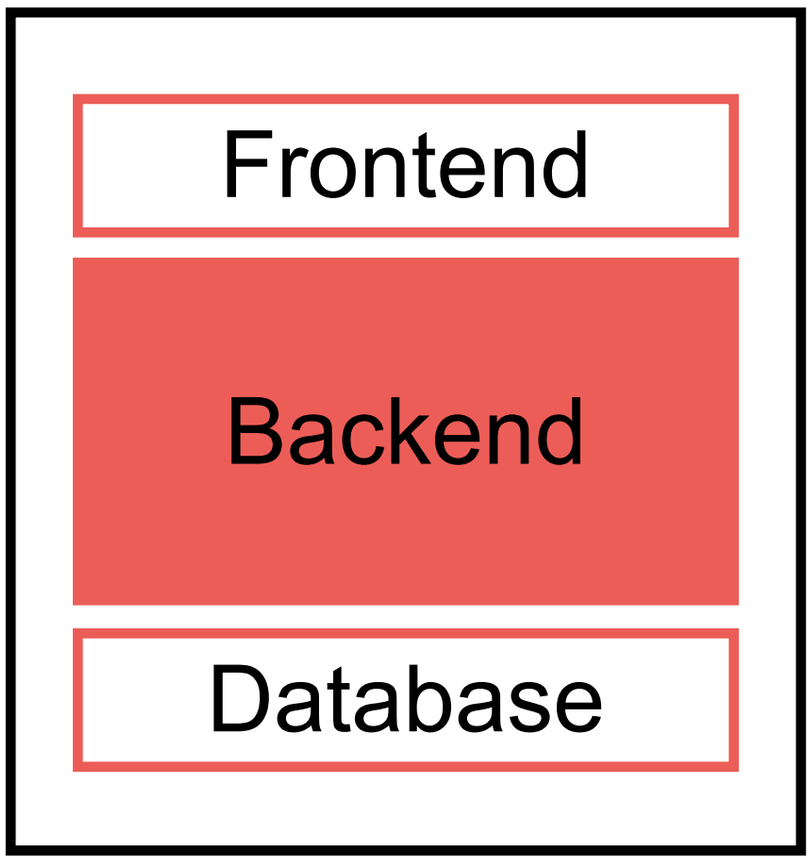




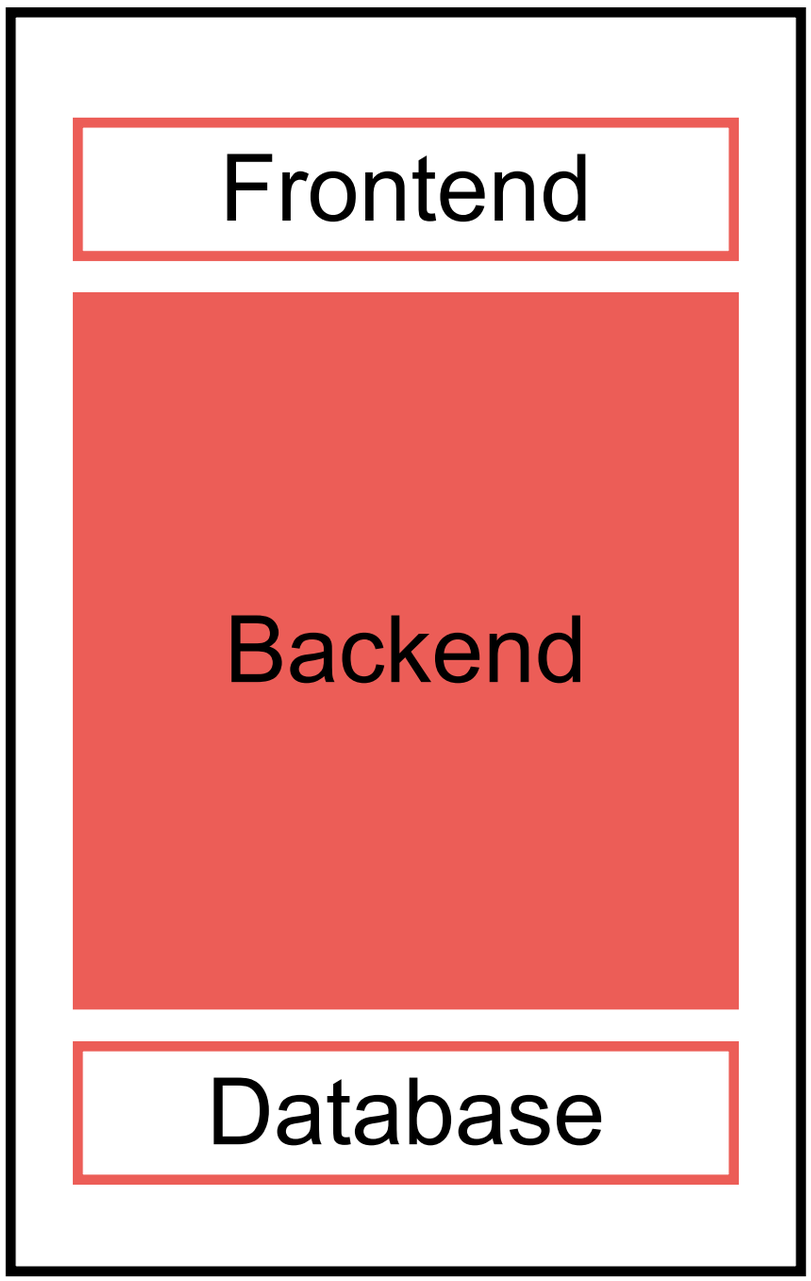
Tiers (out-of-process)



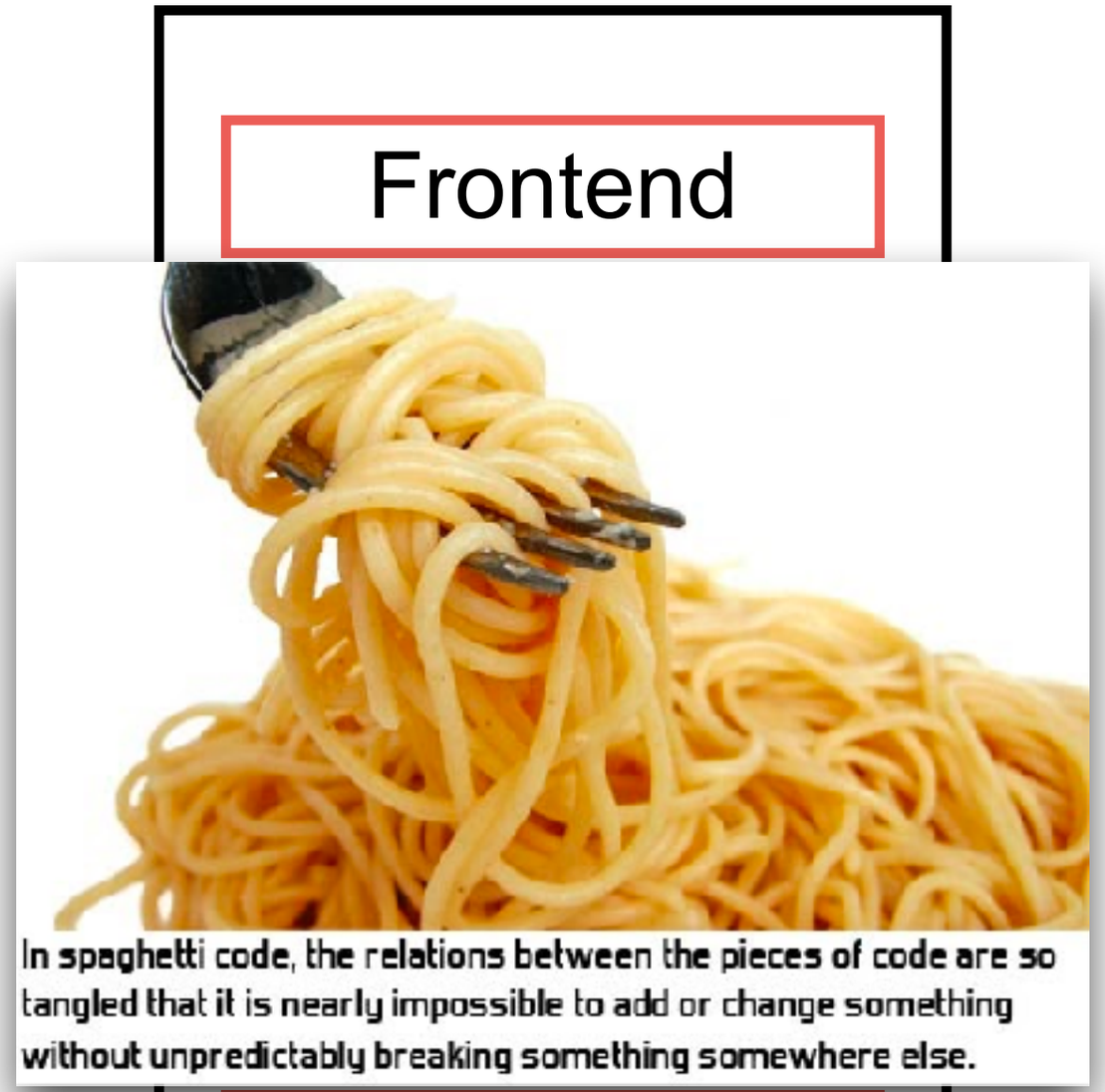
More features …



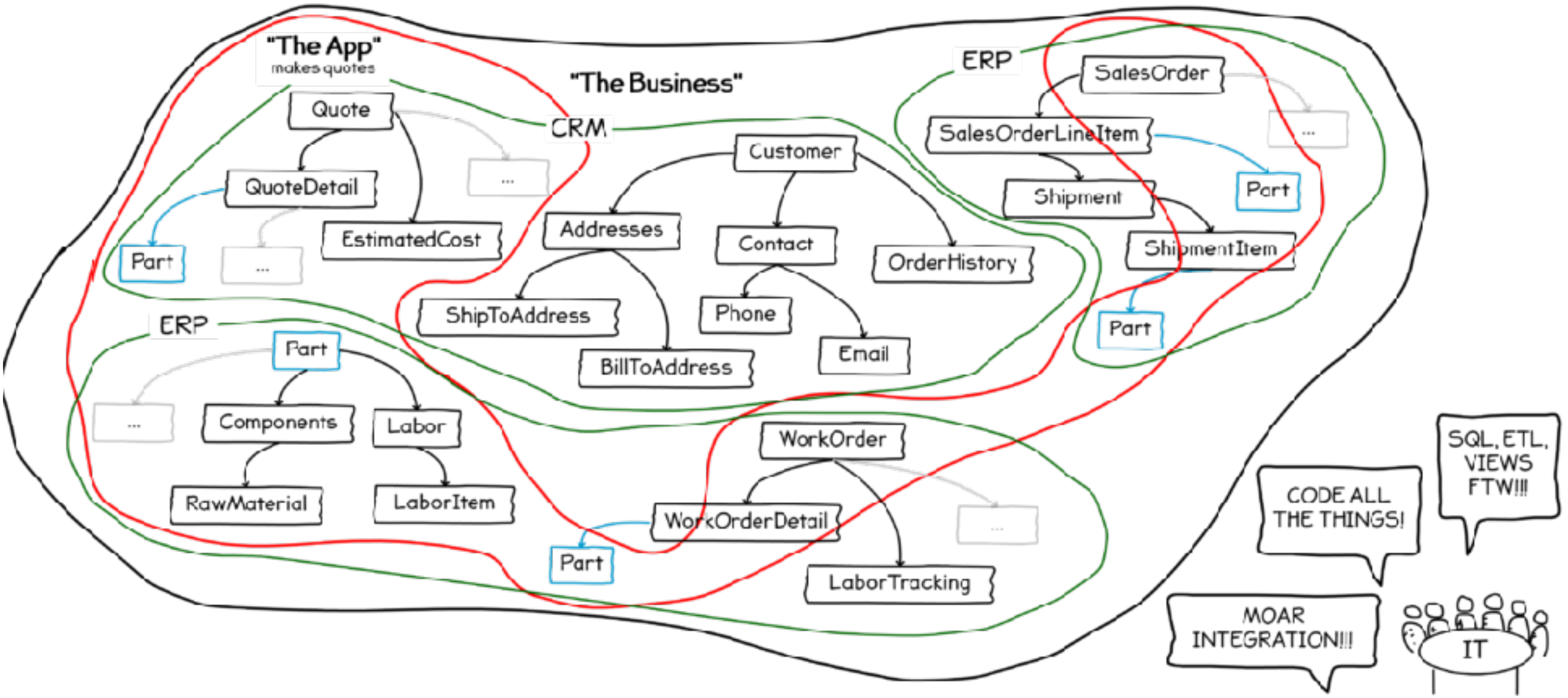
More features ...

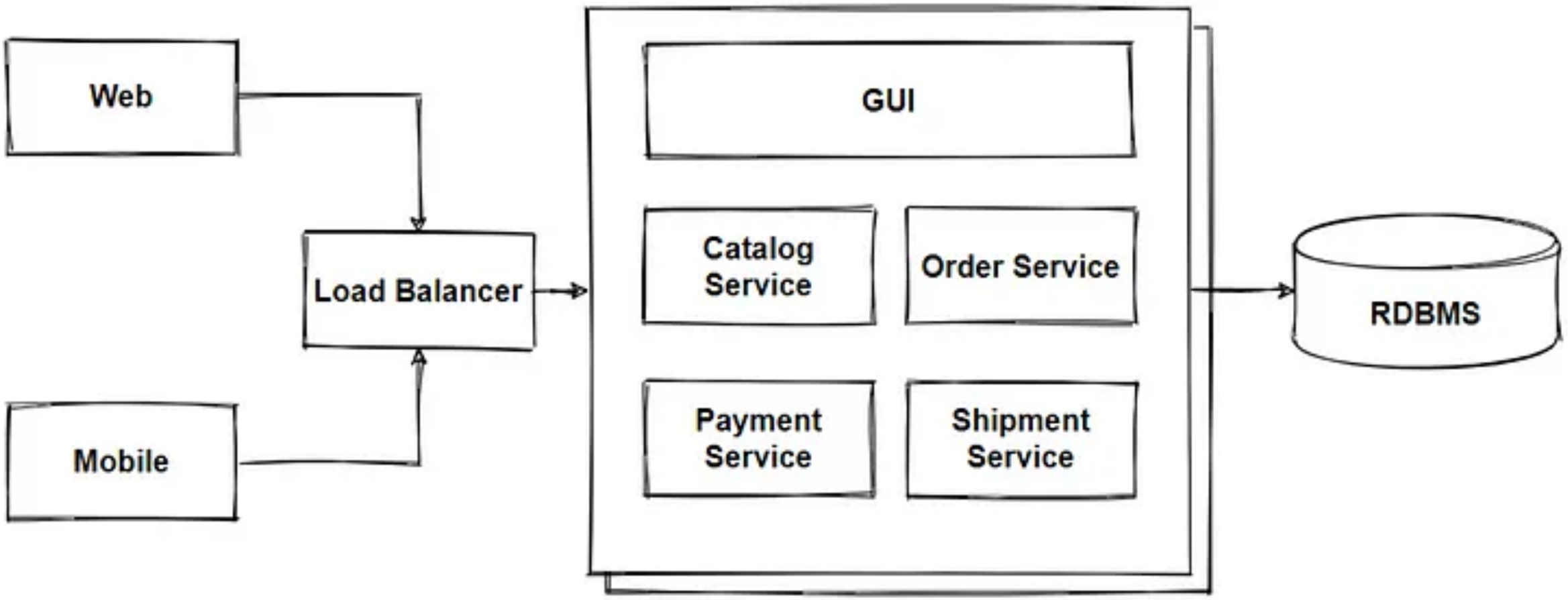


More features …

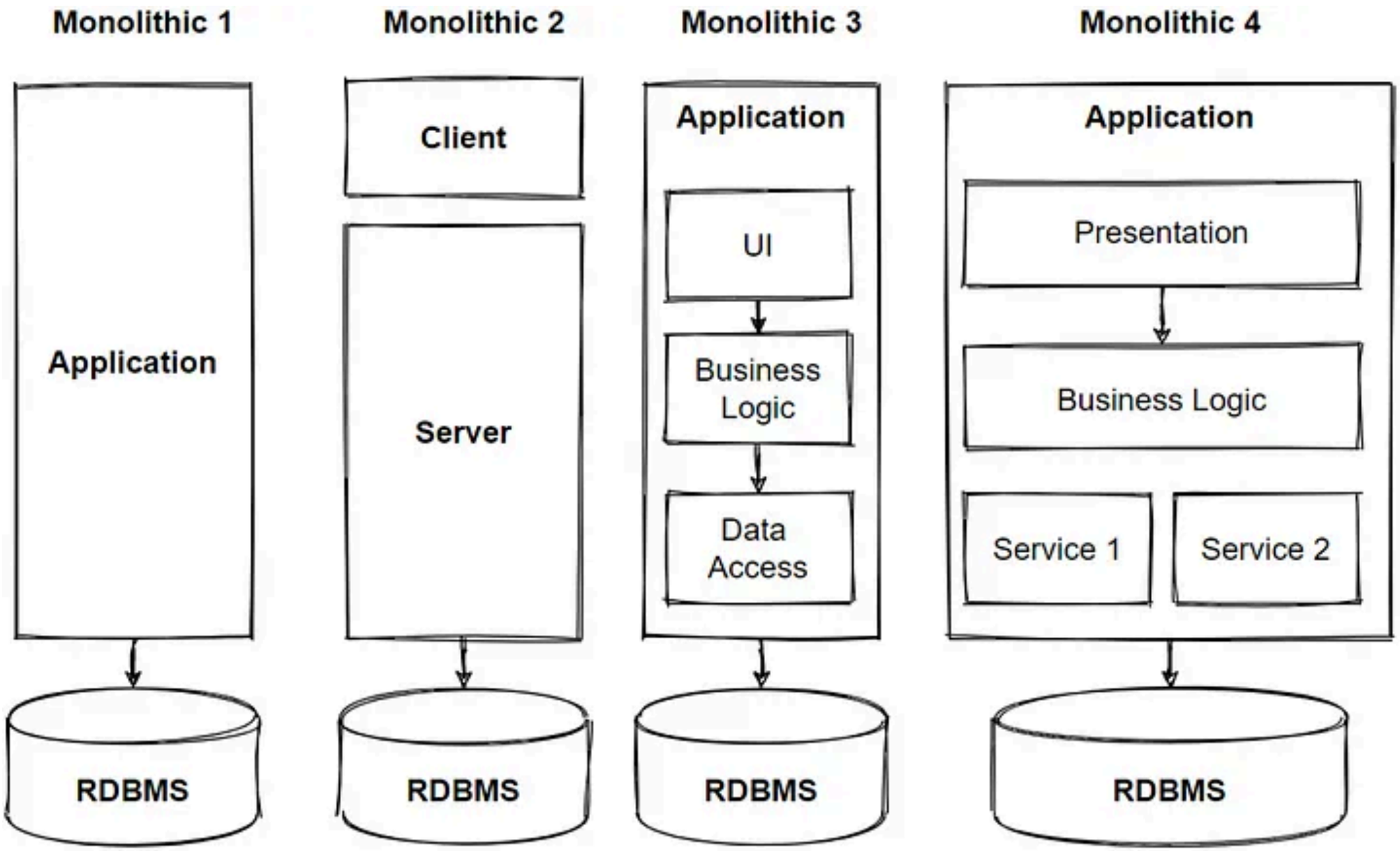


Monolith Hell



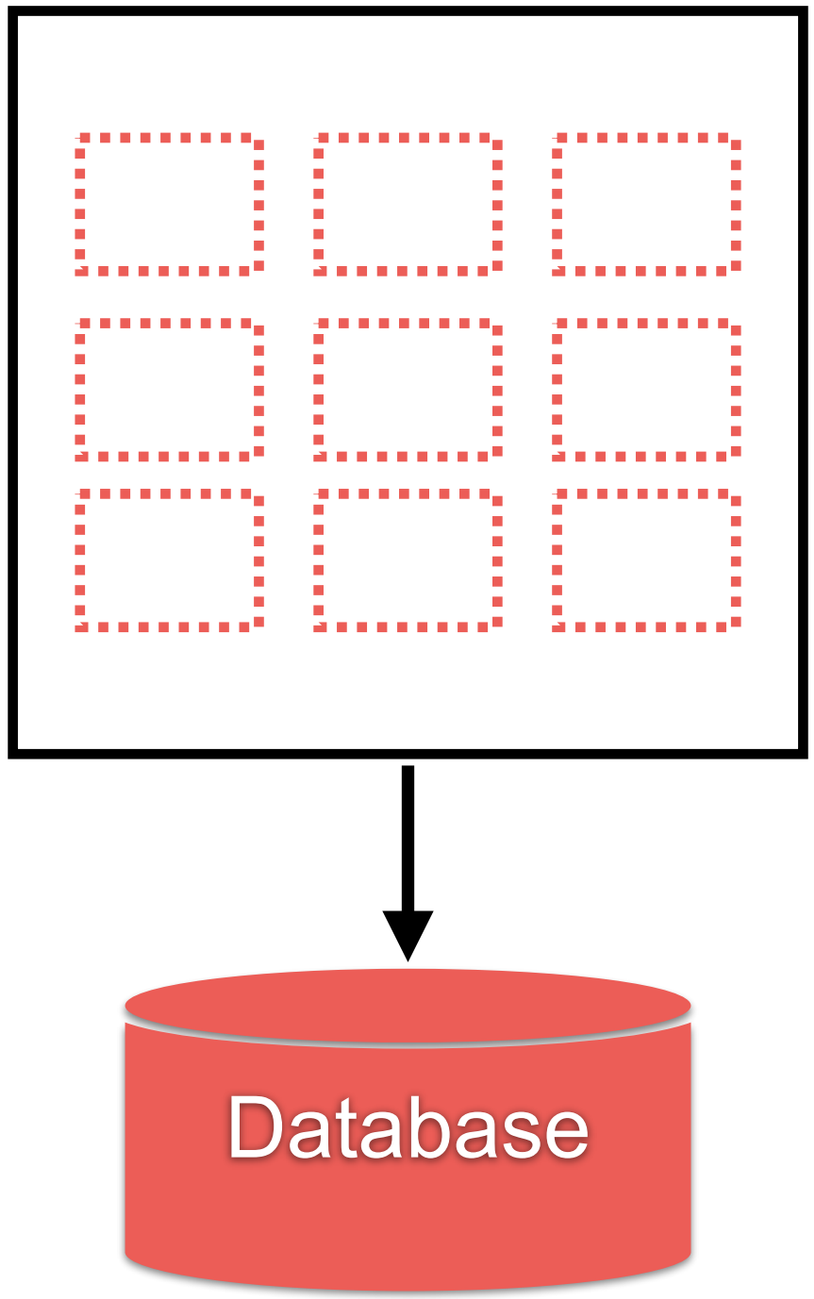


Evolution of Monolith …



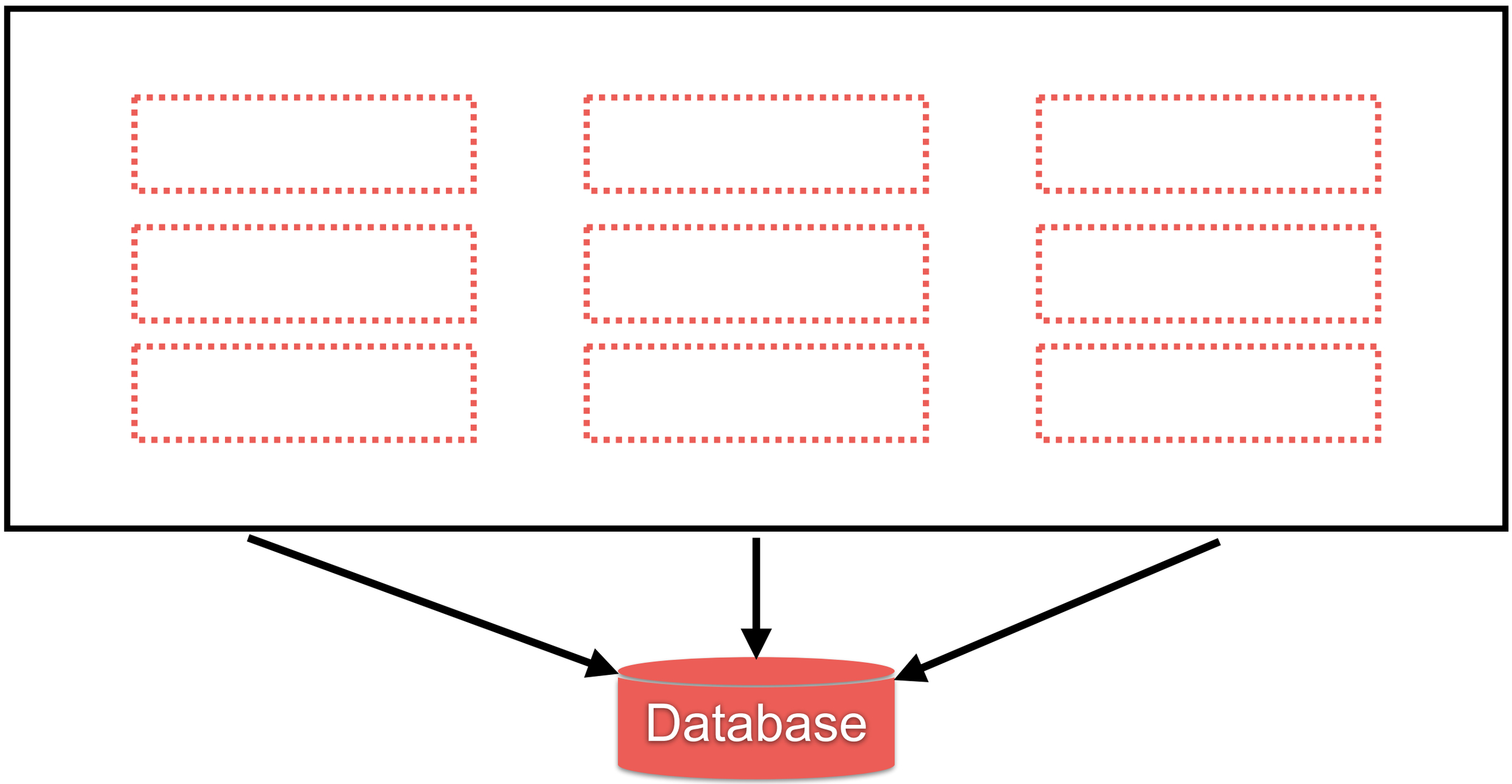
How to scale ?

Service with Modular

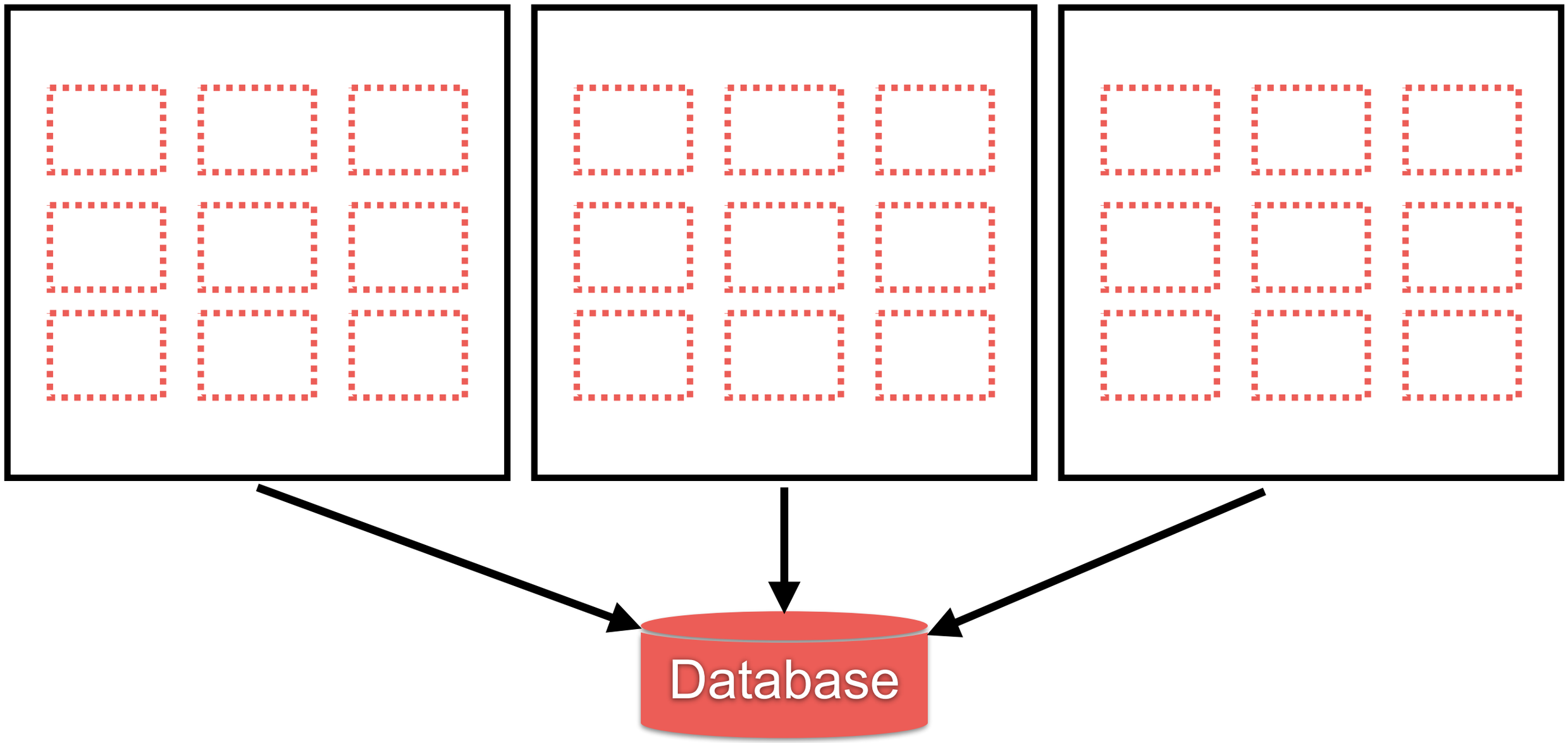


How to scale ?

Scale Up (Vertical)

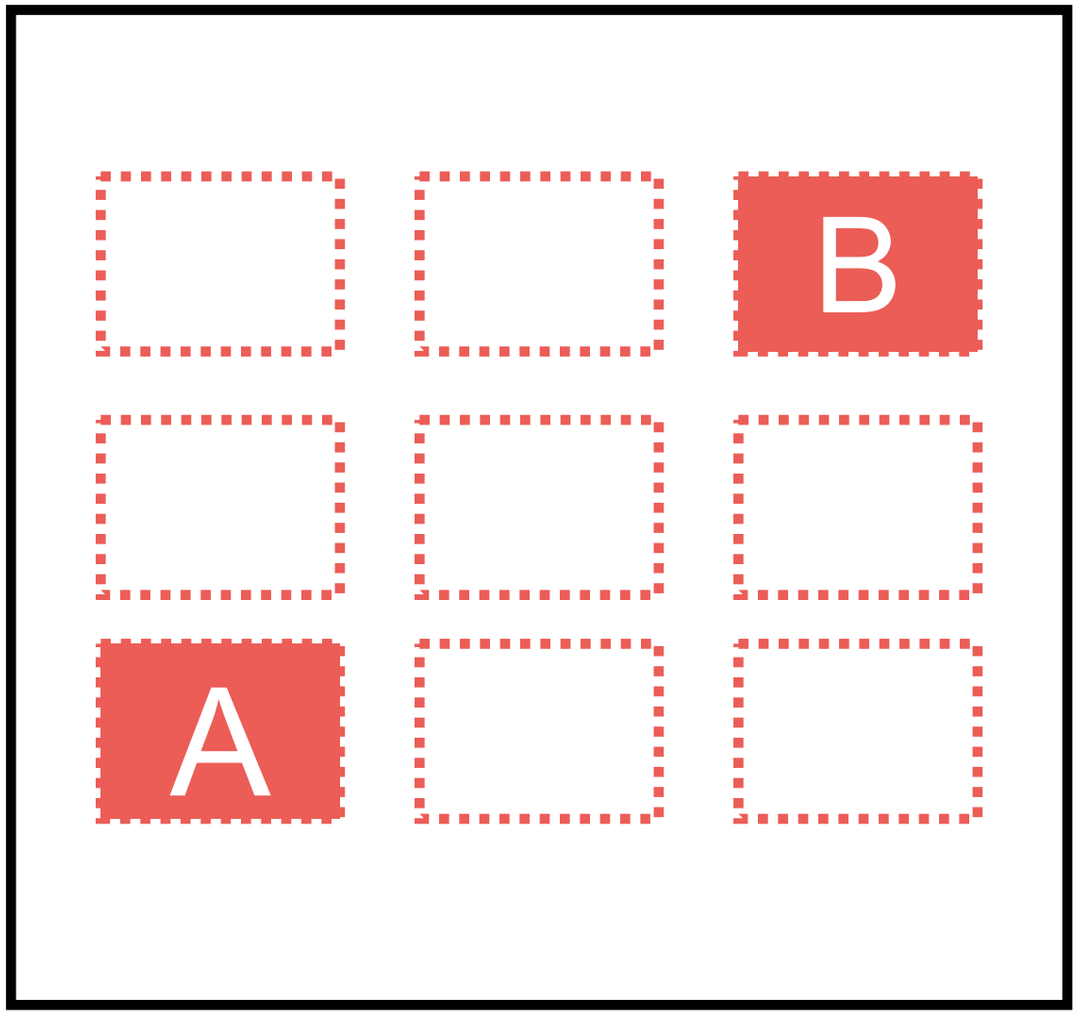


Scale Out (Horizontal)



What happens when we need more servers ?

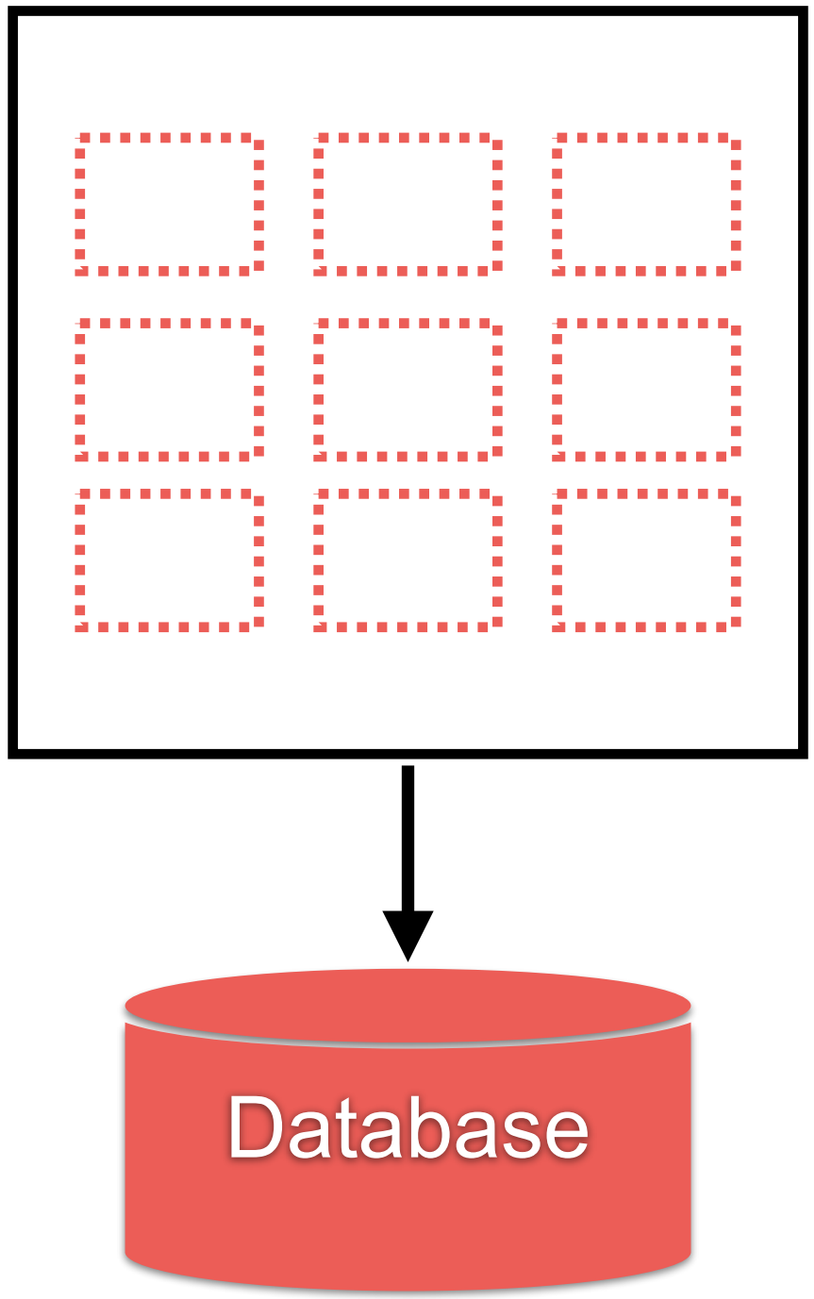
What if we don’t use all modules equally ?



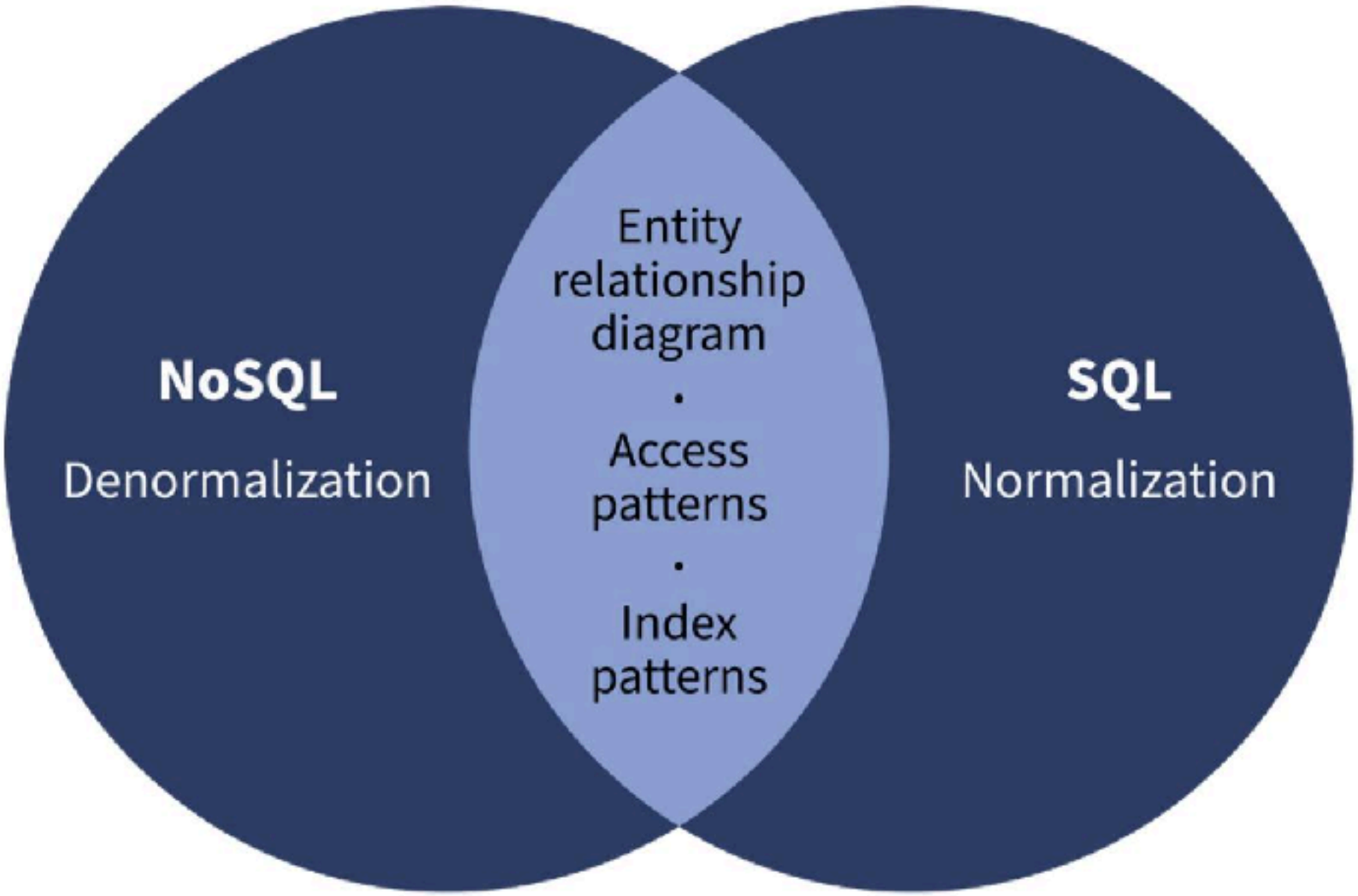
How can we update individual models/database ?

Do all modules need to use the same Database, Language and Runtime … ?

Database ?



Database ?



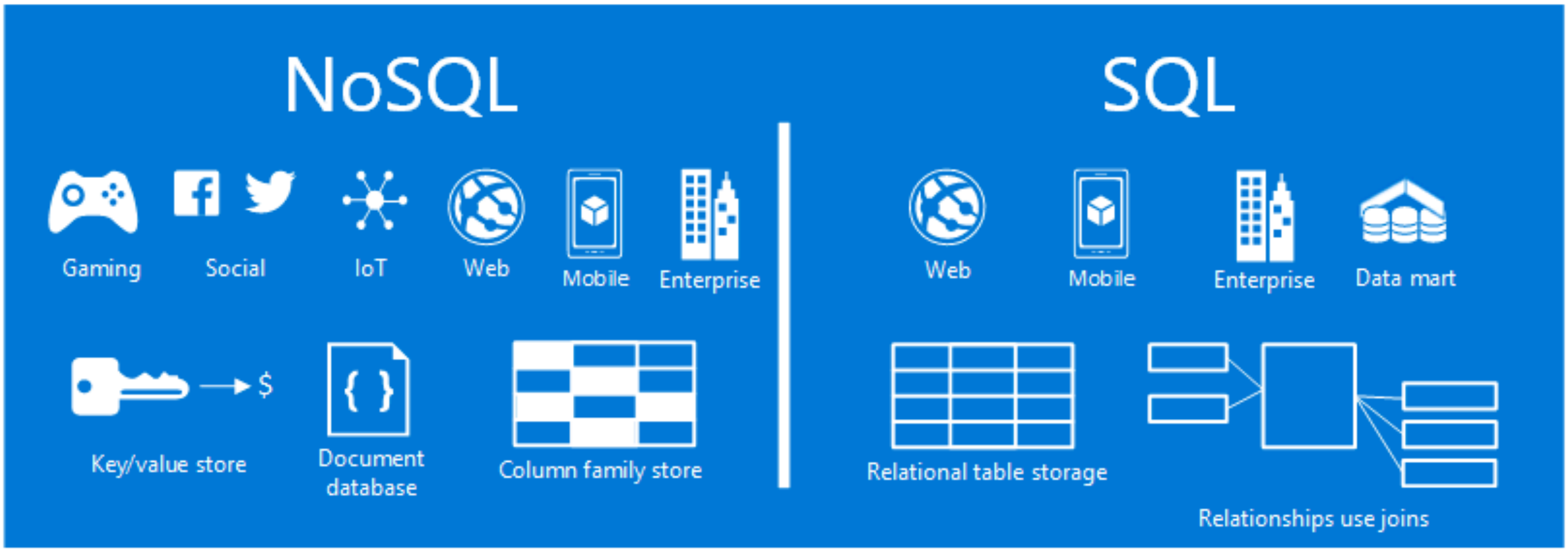
Database Models

422 systems in ranking, September 2023

Rank			DBMS	Database Model	Score		
Sep 2023	Aug 2023	Sep 2022			Sep 2023	Aug 2023	Sep 2022
1.	1.	1.	Oracle	Relational, Multi-model	1240.88	-1.22	+2.62
2.	2.	2.	MySQL	Relational, Multi-model	1111.49	-18.97	-100.98
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	902.22	-18.60	-24.08
4.	4.	4.	PostgreSQL	Relational, Multi-model	620.75	+0.37	+0.29
5.	5.	5.	MongoDB	Document, Multi-model	439.42	+4.93	-50.21
6.	6.	6.	Redis	Key-value, Multi-model	163.68	+0.72	-17.79
7.	7.	7.	Elasticsearch	Search engine, Multi-model	138.98	-0.94	-12.46
8.	8.	8.	IBM Db2	Relational, Multi-model	136.72	-2.52	-14.67
9.	↑ 10.	↑ 10.	SQLite	Relational	129.20	-0.72	-9.62
10.	↓ 9.	↓ 9.	Microsoft Access	Relational	128.56	-1.78	-11.47
11.	11.	↑ 13.	Snowflake	Relational	120.89	+0.27	+17.39
12.	12.	↓ 11.	Cassandra	Wide column, Multi-model	110.06	+2.67	-9.06
13.	13.	↓ 12.	MariaDB	Relational, Multi-model	100.45	+1.80	-9.70

https://db-engines.com/en/ranking

Database Models



We need to improve

We need to get better

SOA

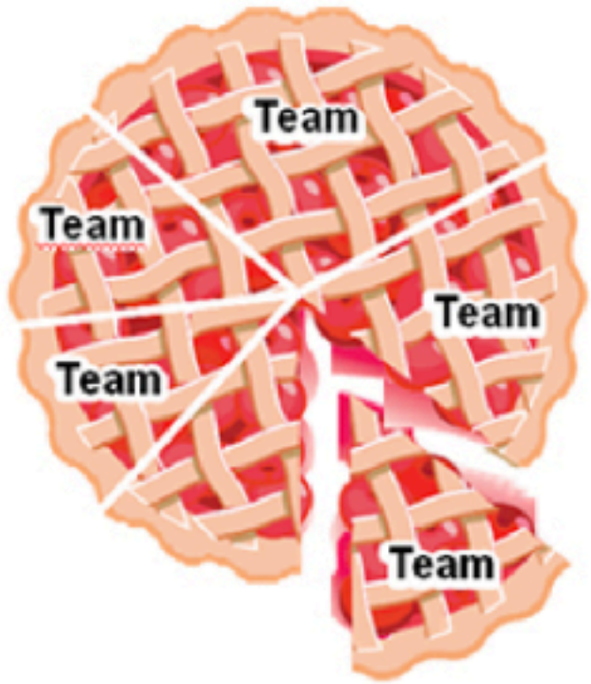
1990s and earlier

Pre-SOA (monolithic)
Tight coupling

For a monolith to change, all must agree on each change. Each change has unanticipated effects requiring careful testing beforehand.

2000s

Traditional SOA
Looser coupling



Elements in SOA are developed more autonomously but must be coordinated with others to fit into the overall design.

2010s

Microservices
Decoupled

Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.

Service-Oriented Architecture

What is service ?

Standalone and loosely couple
Independently deployable software component
Implement some useful functionality

Service-Oriented Architecture

SOA Manifesto

Service orientation is a paradigm that frames what you do.
Service-oriented architecture (SOA) is a type of architecture
that results from applying service orientation.

We have been applying service orientation to help organizations consistently deliver sustainable business value, with increased agility and cost effectiveness, in line with changing business needs.

http://www.soa-manifesto.org/

Service-Oriented Architecture

Through our work we have come to prioritize:

Business value over technical strategy

Strategic goals over project-specific benefits

Intrinsic interoperability over custom integration

Shared services over specific-purpose implementations

Flexibility over optimization

Evolutionary refinement over pursuit of initial perfection

http://www.soa-manifesto.org/

Service-Oriented Architecture

Through our work we have come to prioritize:

Business value over technical strategy

Strategic goals over project-specific benefits

Intrinsic interoperability over custom integration

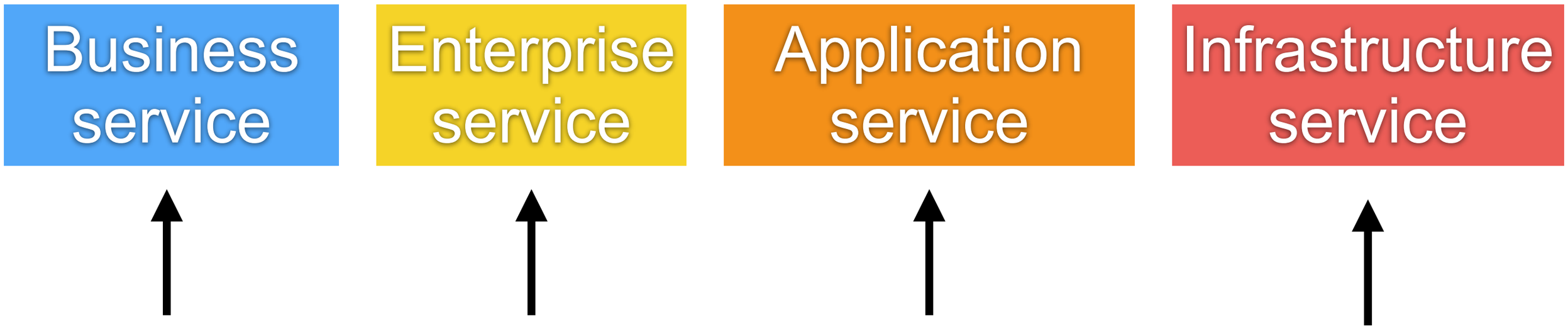
Shared services over specific-purpose implementations

Flexibility over optimization

Evolutionary refinement over pursuit of initial perfection

http://www.soa-manifesto.org/

Service-Oriented Architecture



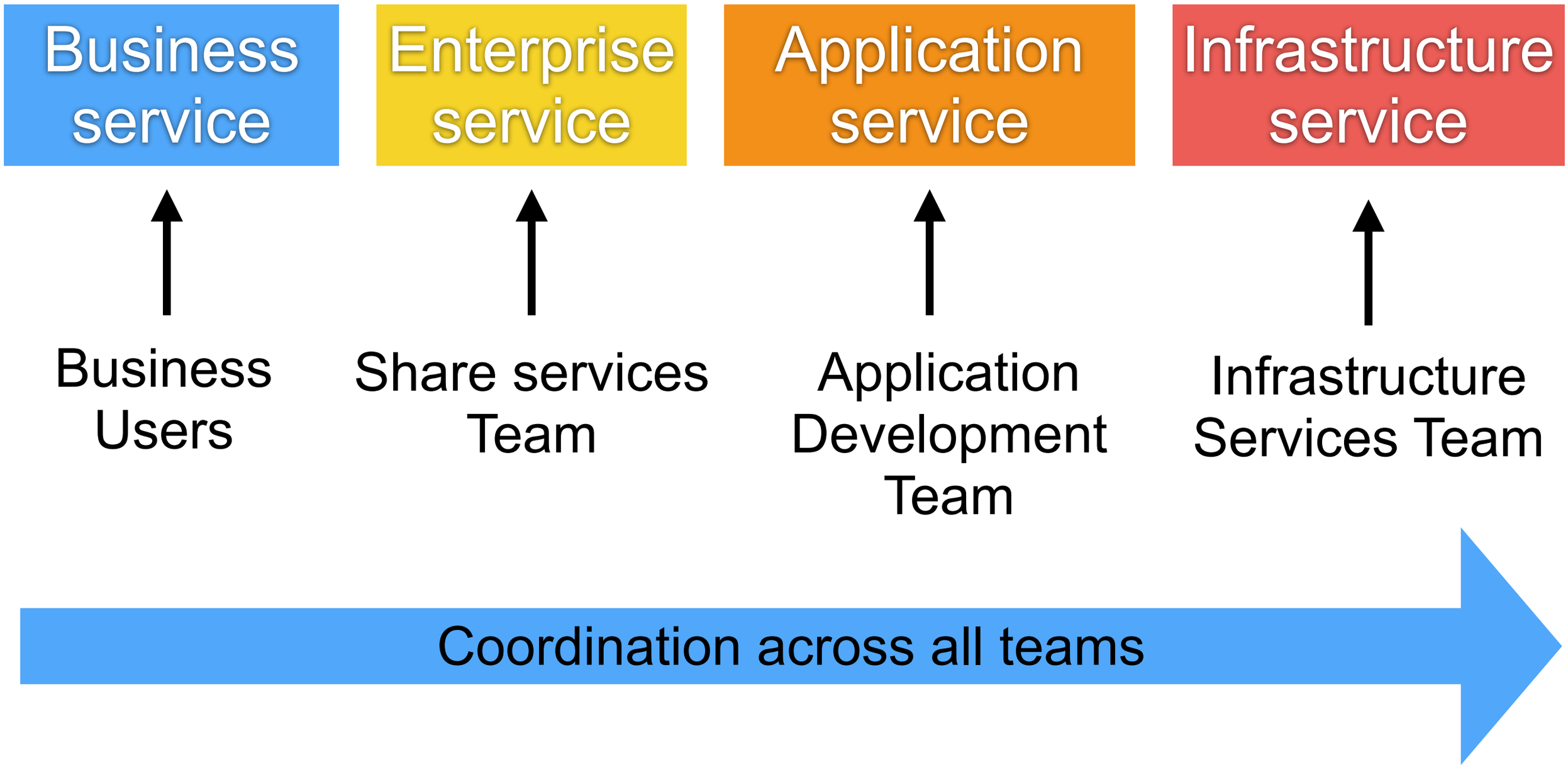
Business Users

Share services Team

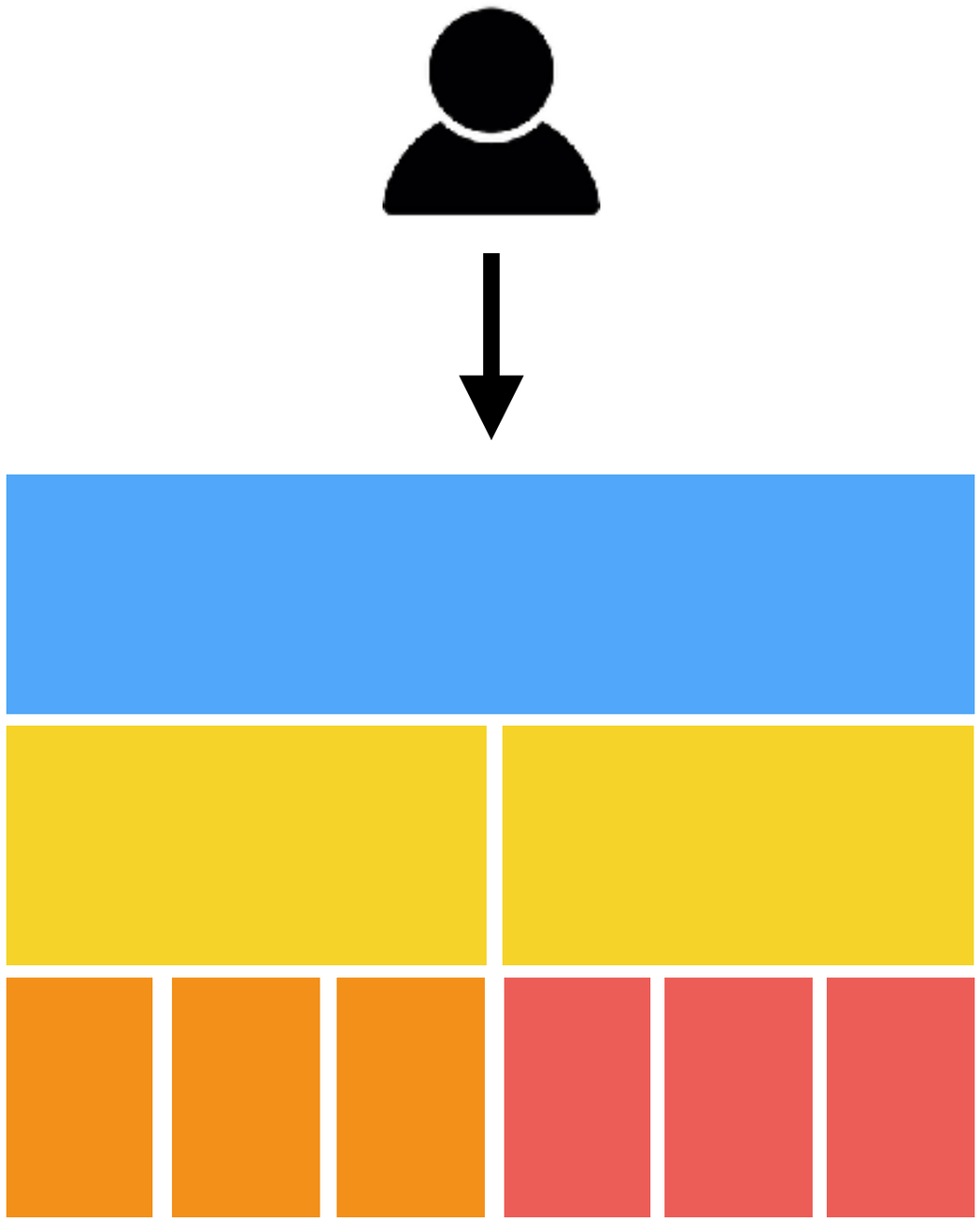
Application Development Team

Infrastructure Services Team

Service-Oriented Architecture



Service-Oriented Architecture

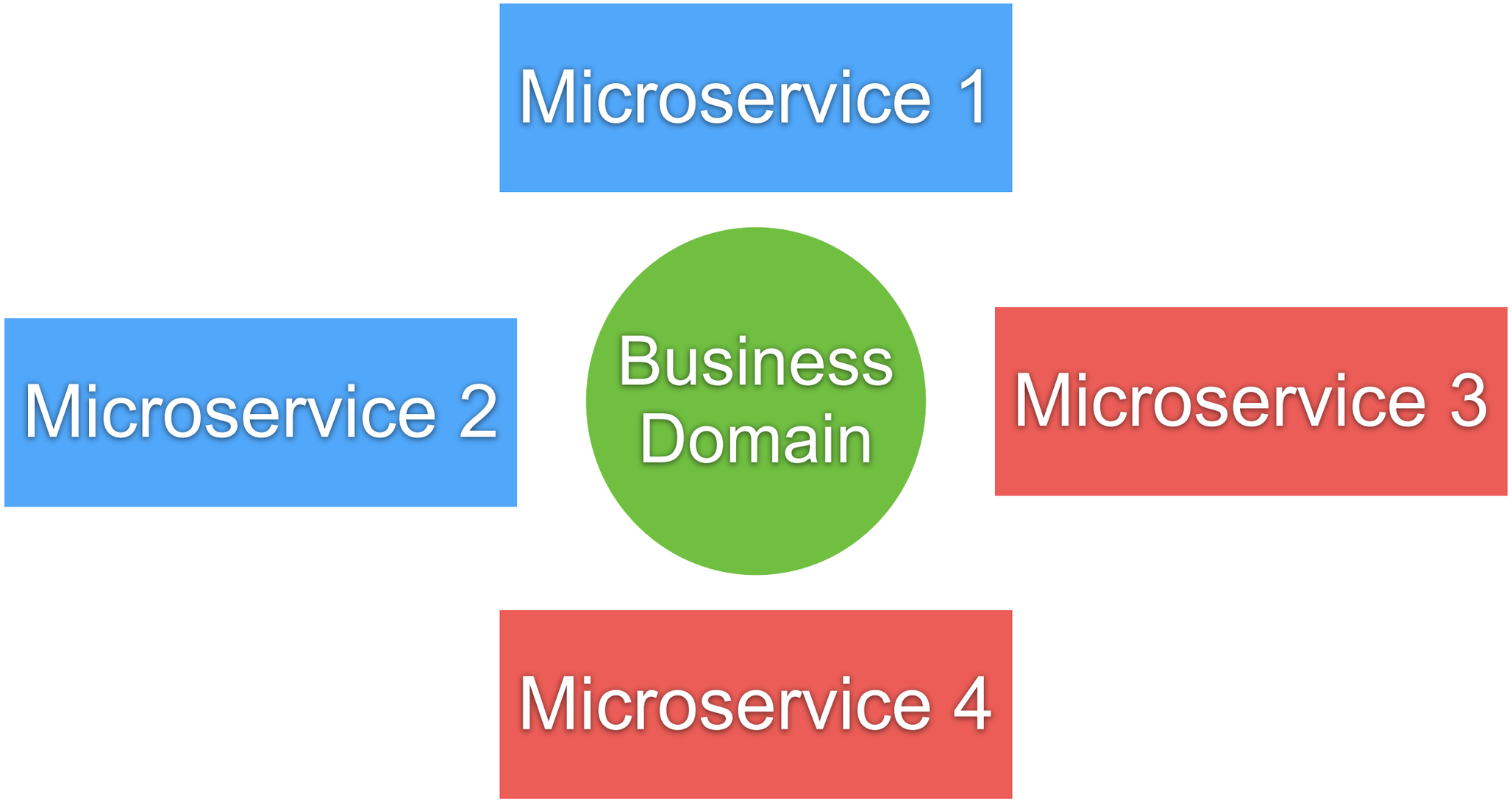


We need to improve

We need to get better

Microservice

Microservice

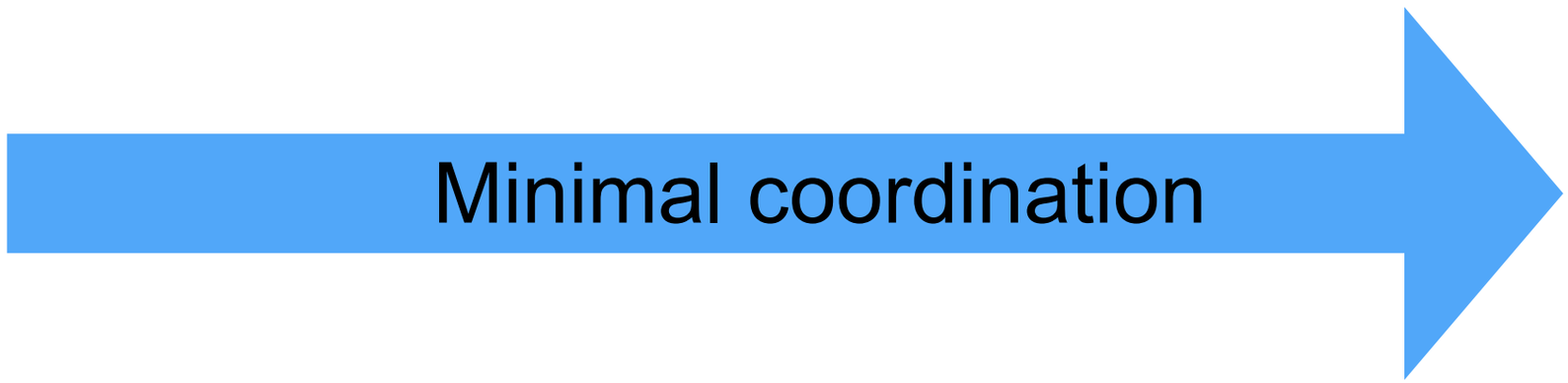


Microservice

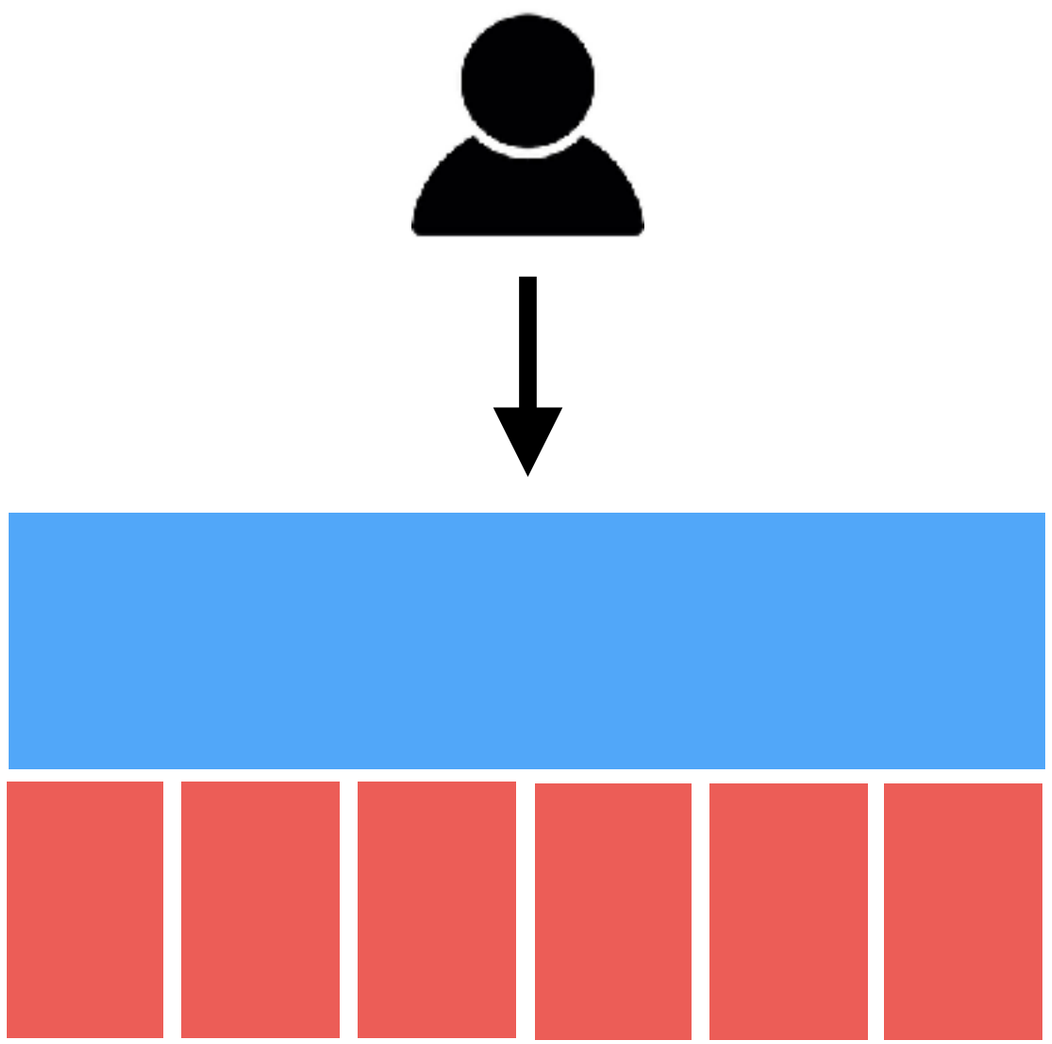
Business service

Self-service

Application Development Team



Microservice



Delivery Problems

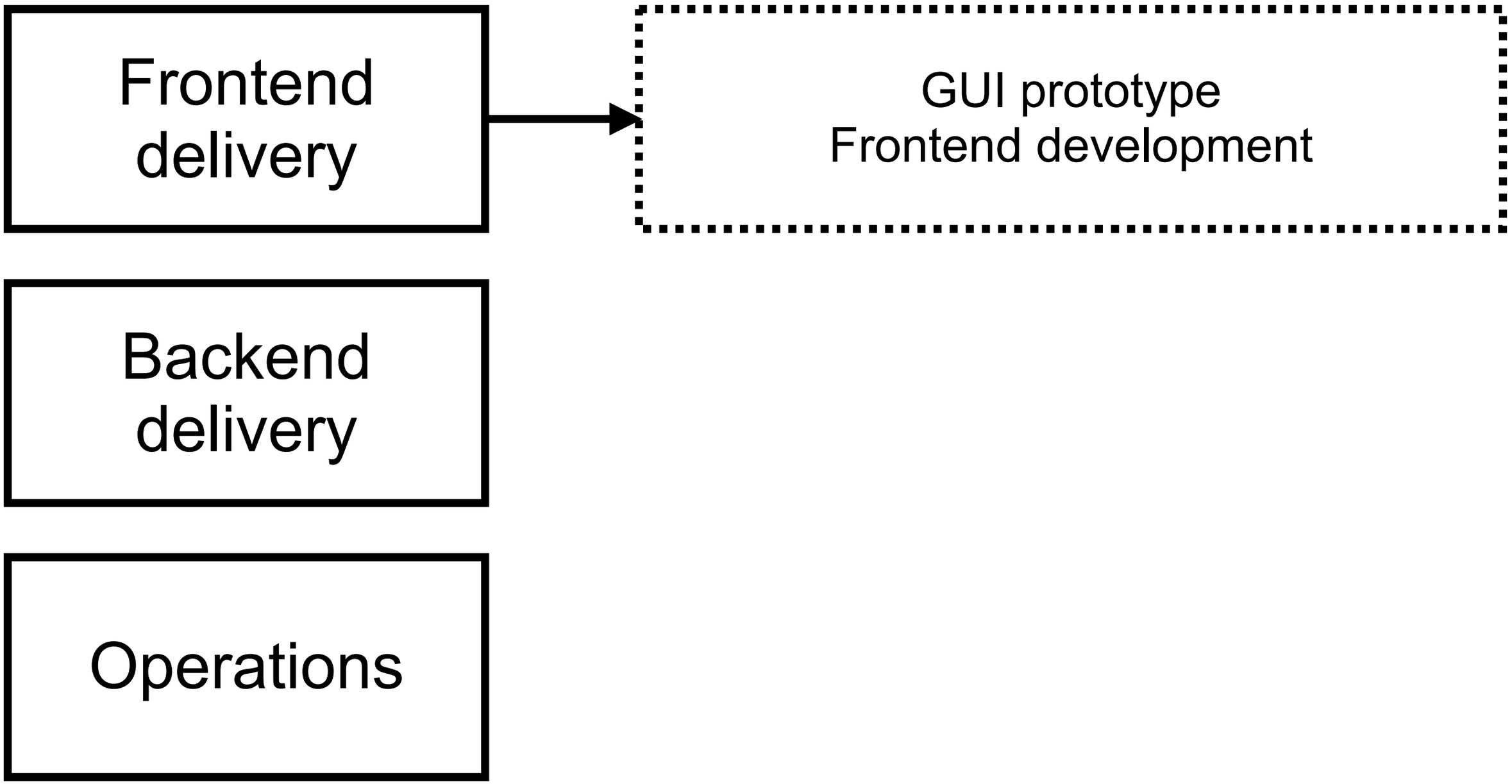
Delivery Responsibilities

Frontend delivery

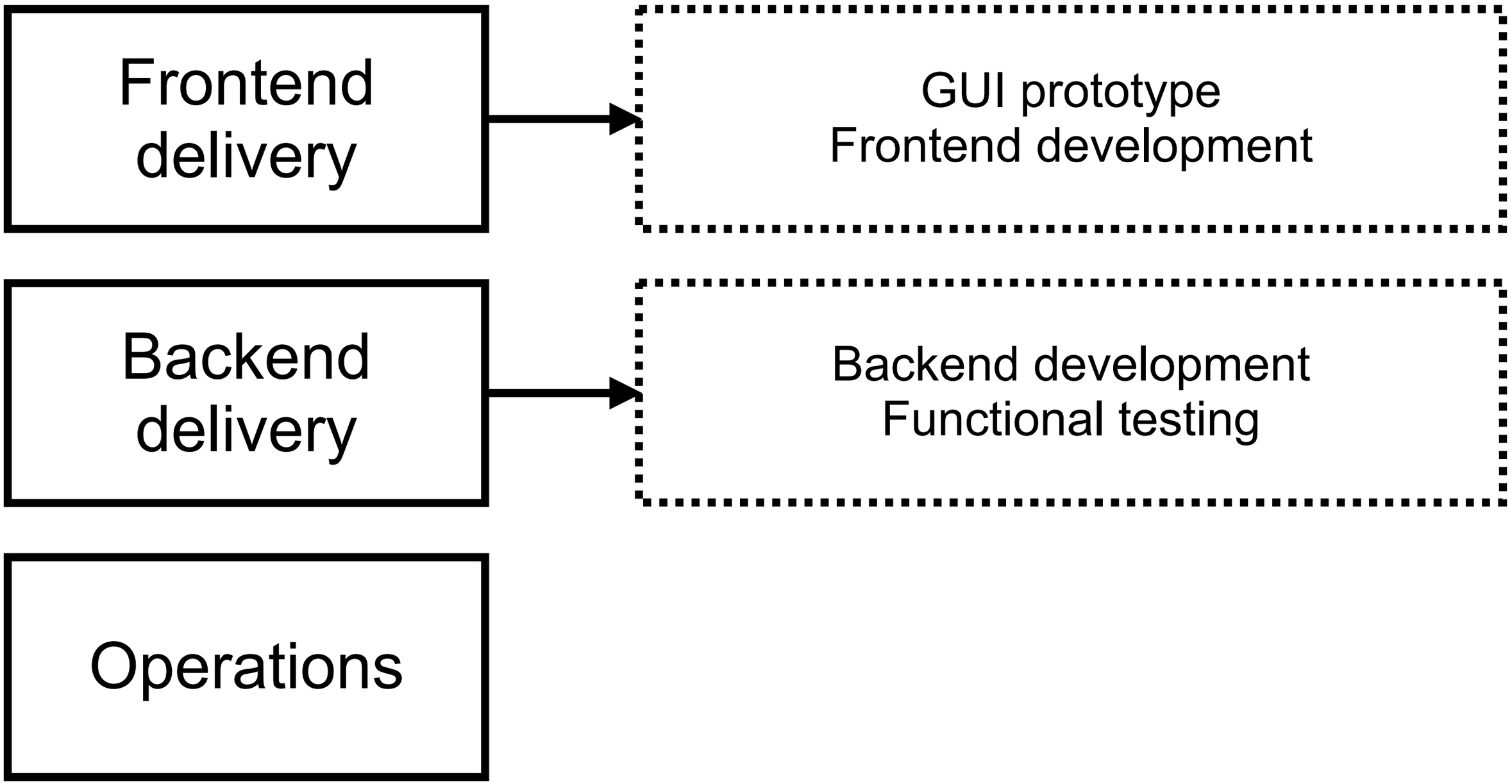
Backend delivery

Operations

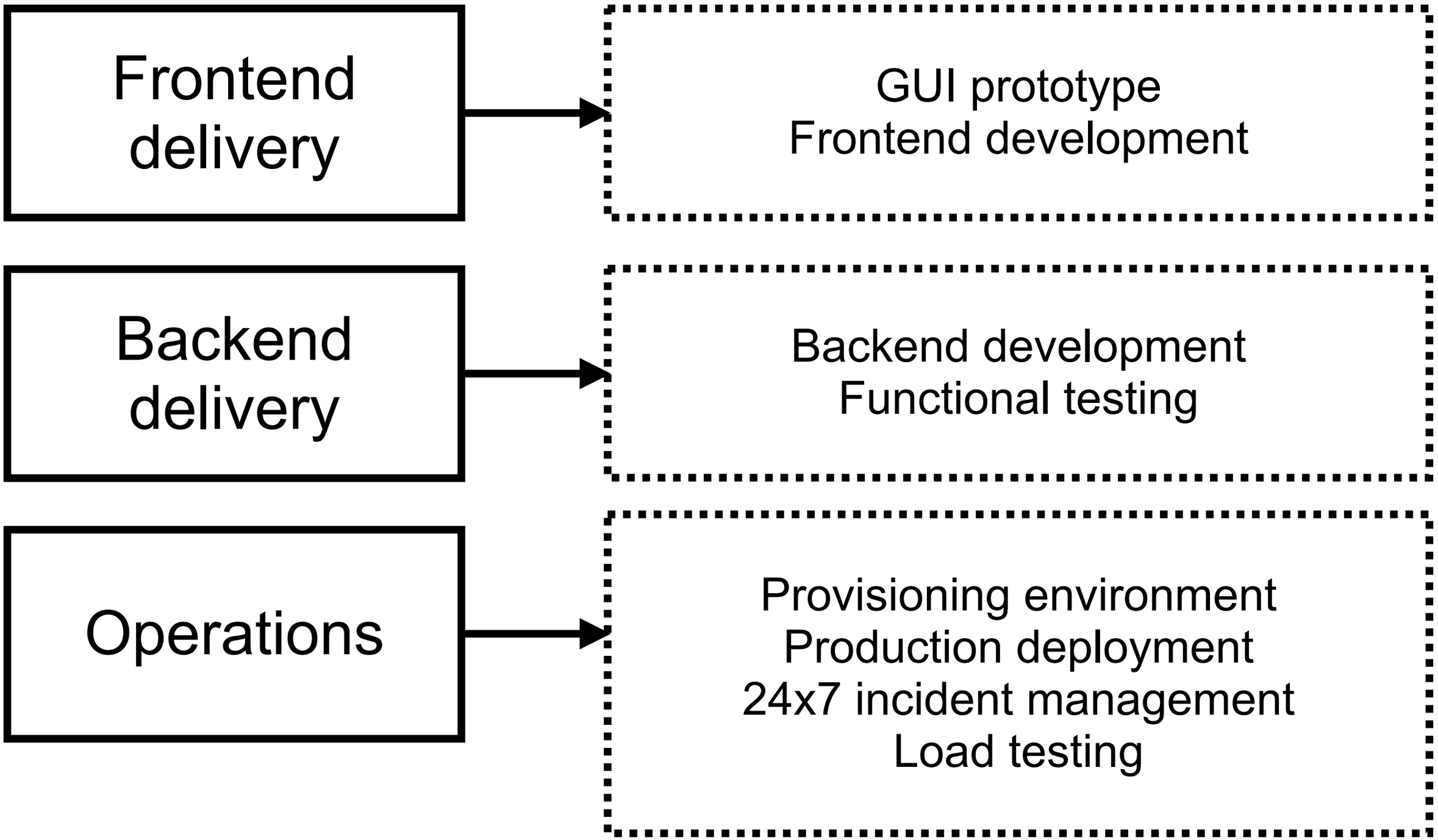
Delivery Responsibilities



Delivery Responsibilities

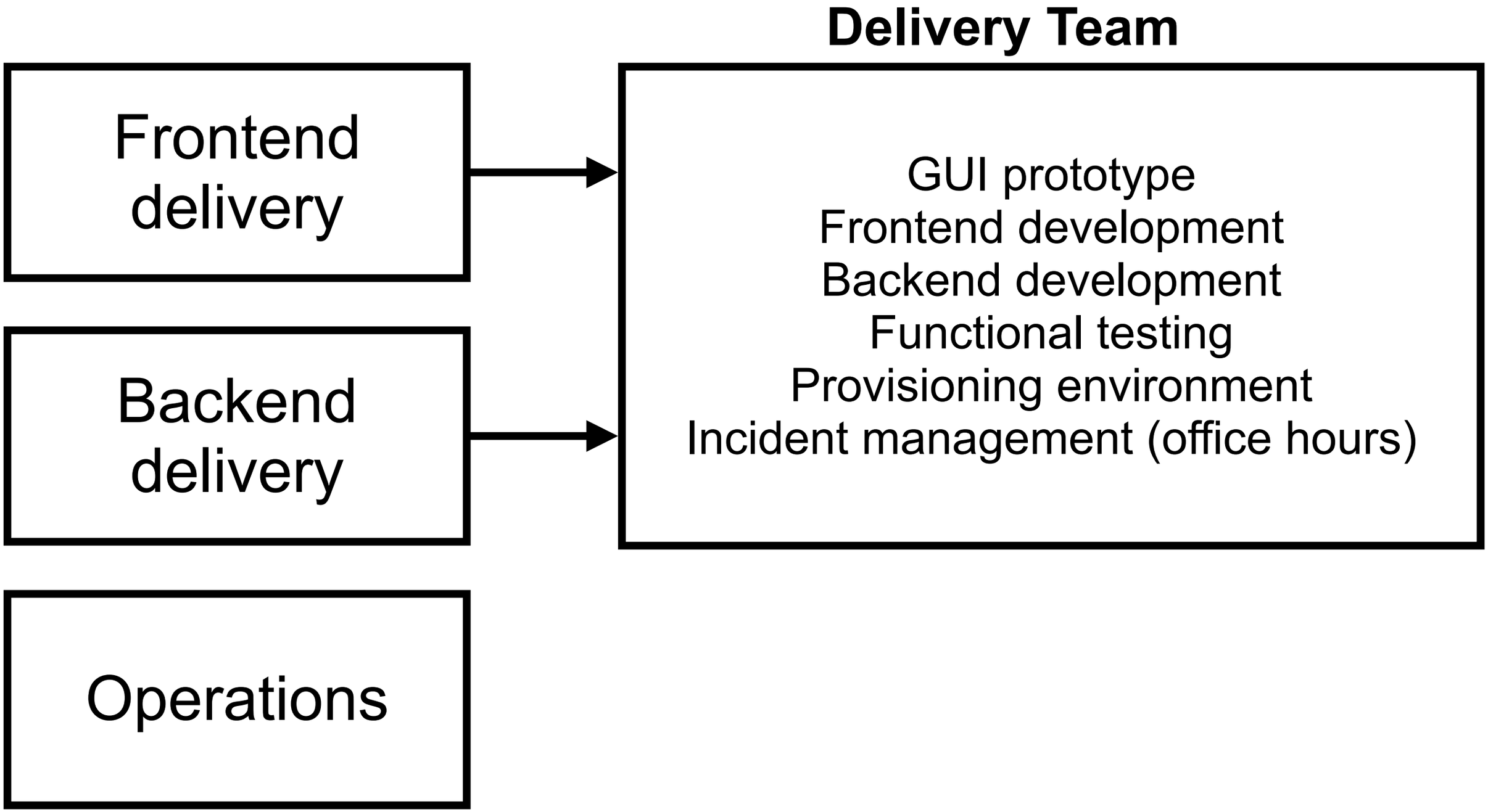


Delivery Responsibilities

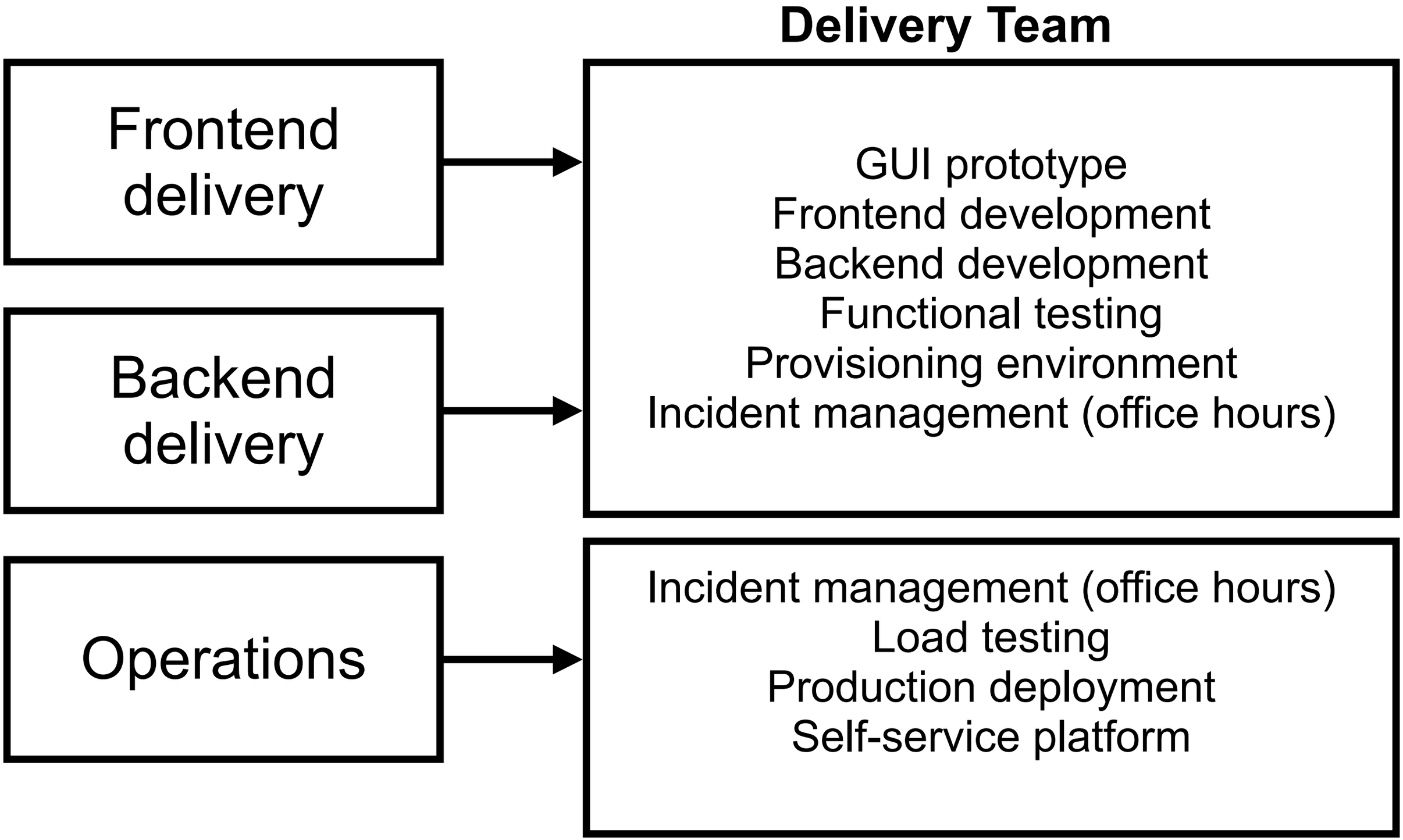


Delivery Improvement ?

Better ?



Better ?



Microservice History

Microservice History

1990	2001	2006	2009	2010	2012	2013	2014	2017
Continuous **I**ntegration	Agile Manifesto	Amazon EC2	Java EE 6 DevOps	**Netflix + AWS**	ZUUL Ribbon Hystrix Eureka **Microservices**	Container **Docker**	Spring Boot **Adopt Microservices**	Kubernetes Istio

Microservice

Small, Do one thing (Single Responsibility)
Modular
Easy to **understand**
Easy to **develop**
Easy to **deploy**
Easy to **maintain**
Scale independently

Goals

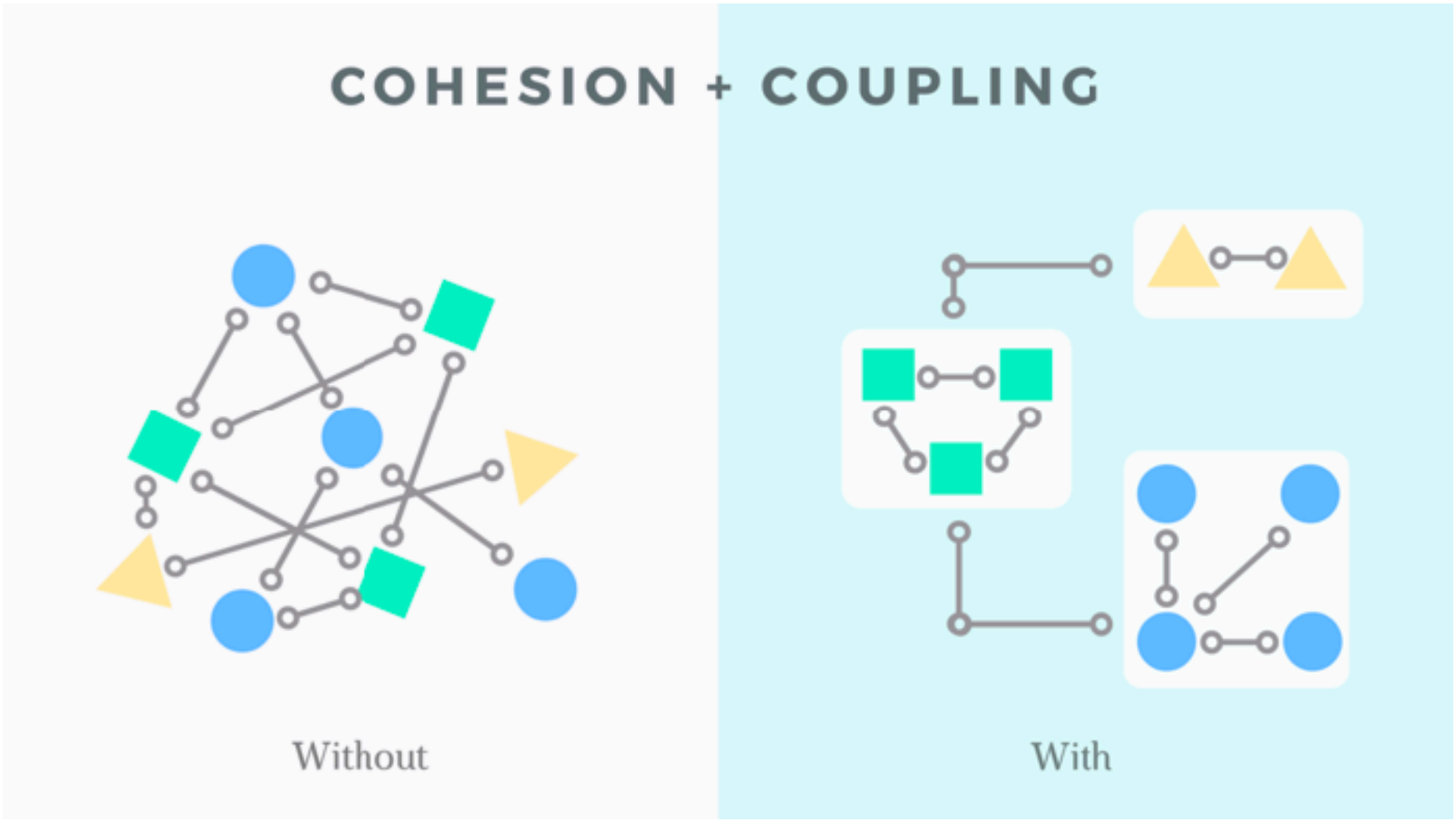
Increase the **velocity** of application release
By **decomposing** app to **small** services
Autonomous services
Deploy **independently**

Independent

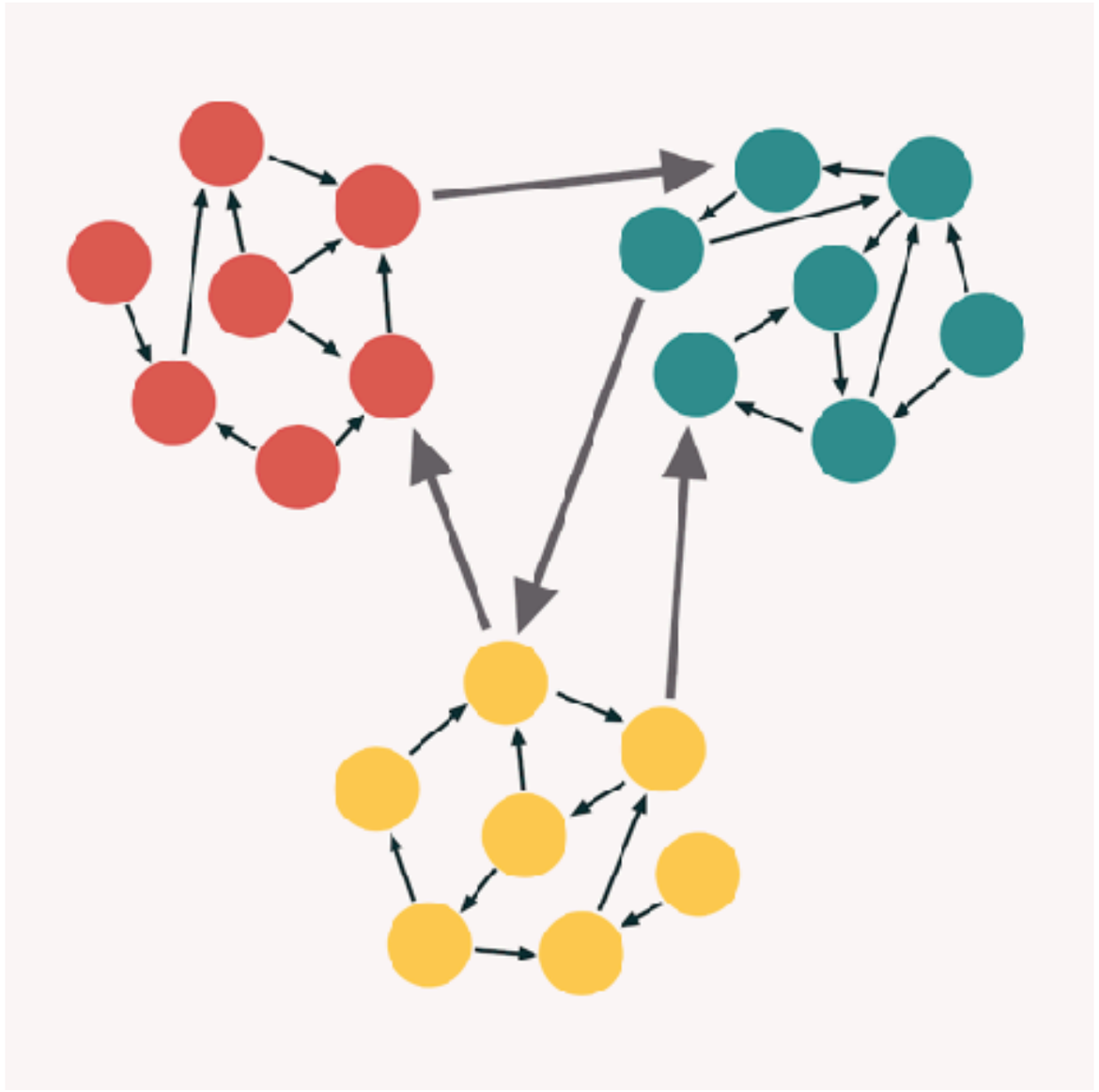
Replicable
Updatable
Scalable
Deployable

Better Architecture

Loose coupling
High cohesion

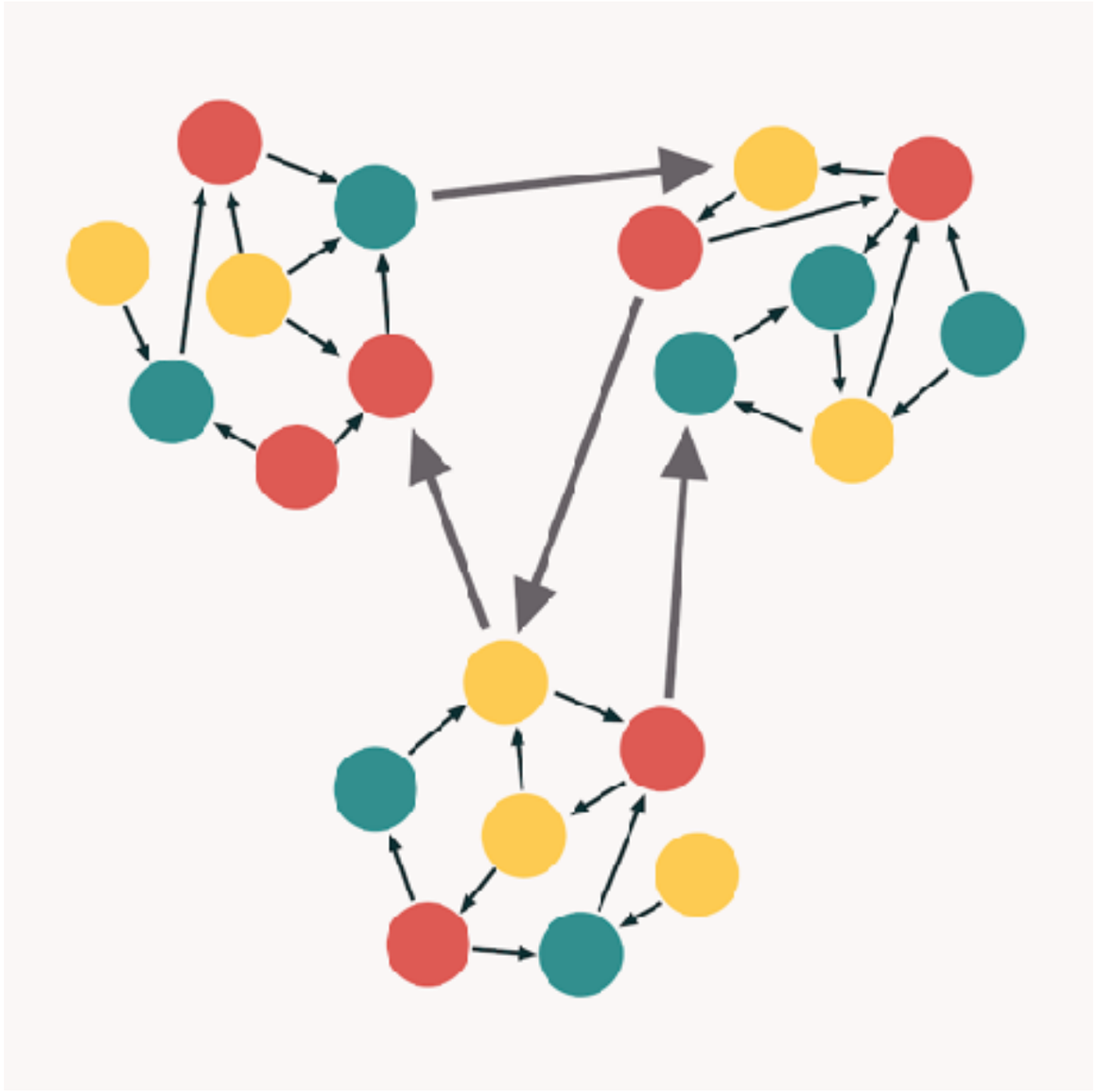


Coupling (Idea)

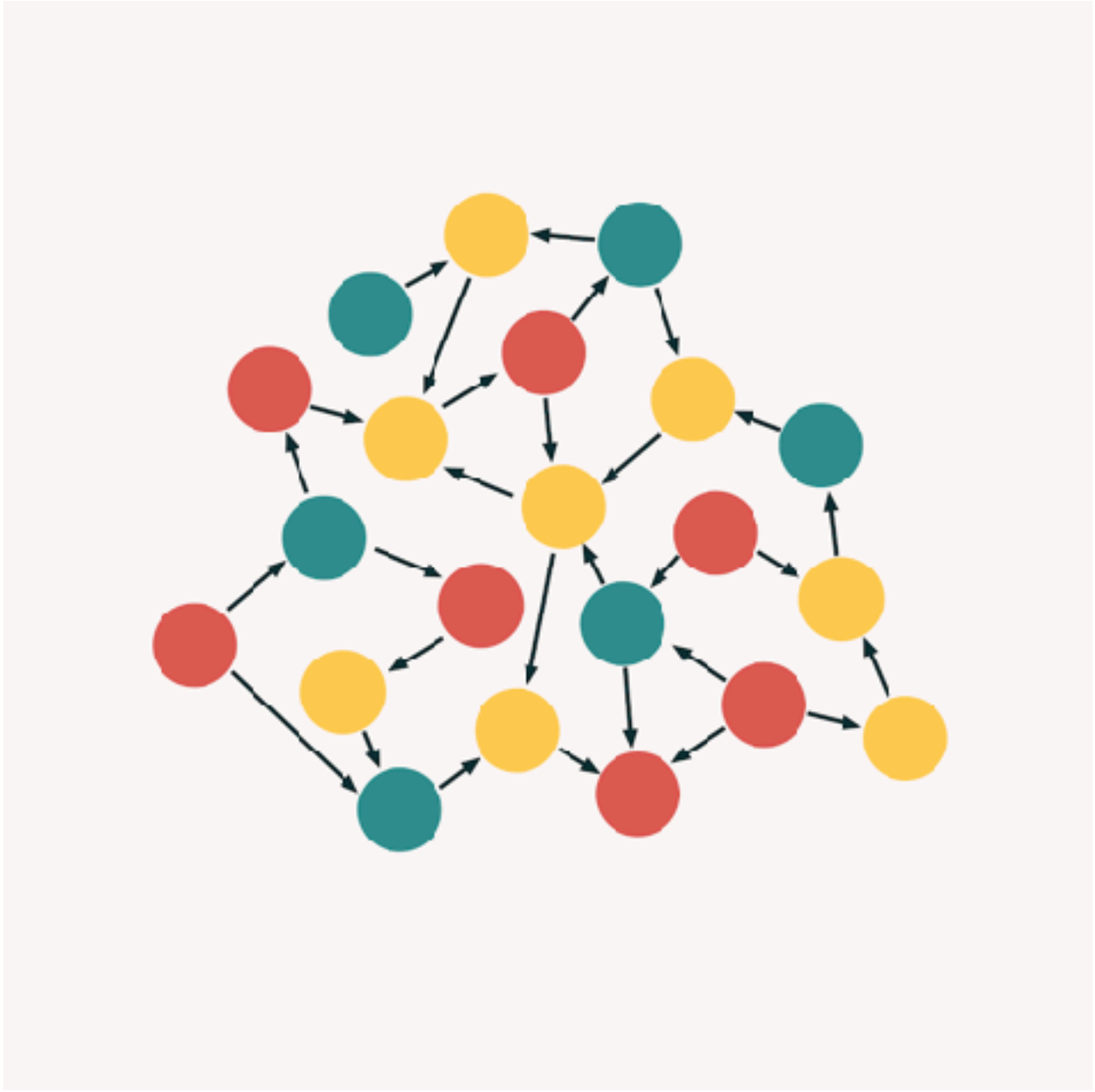


https://enterprisecraftsmanship.com/posts/cohesion-coupling-difference/

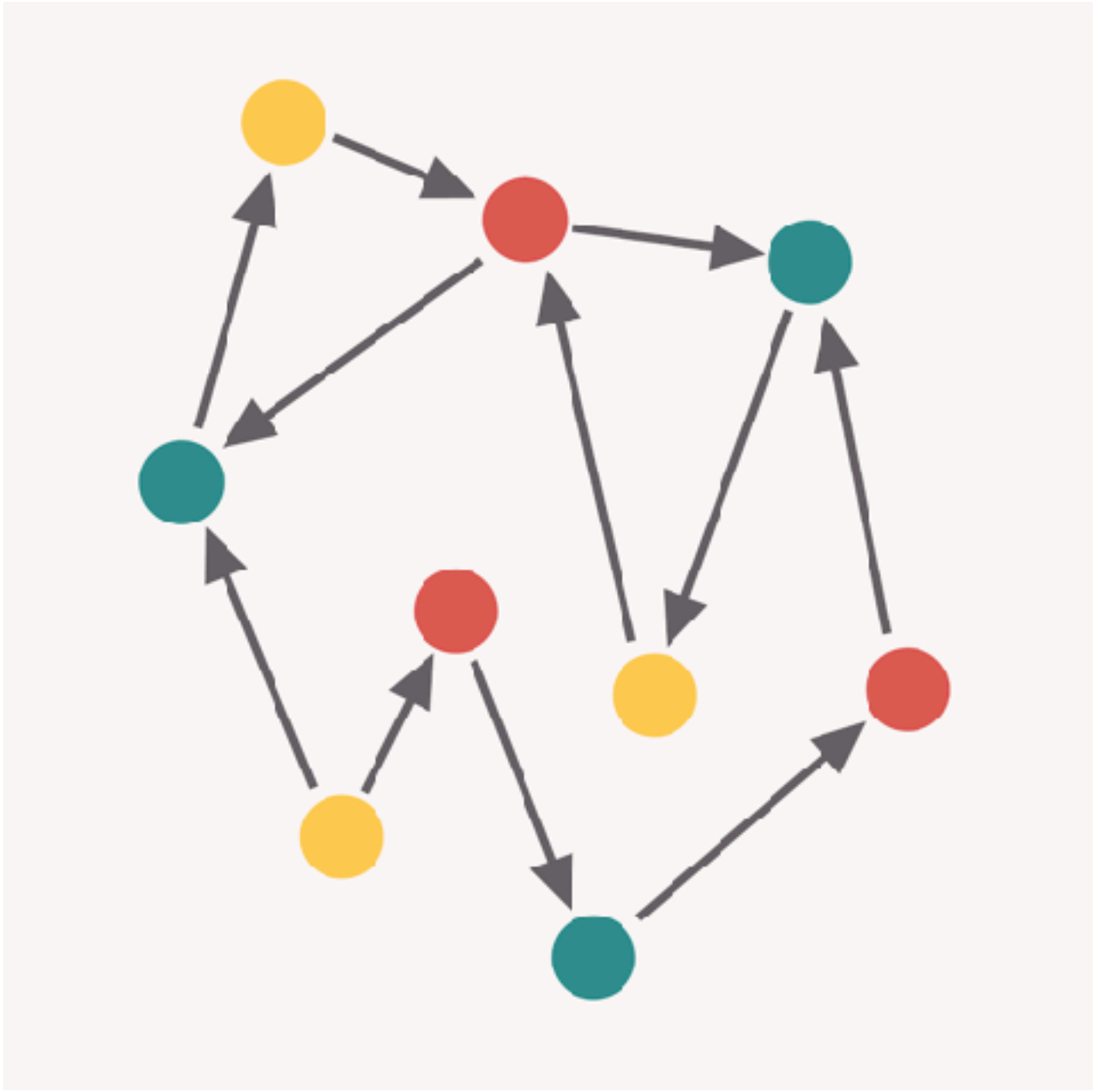
Coupling (Poorly boundary)



Coupling (God)

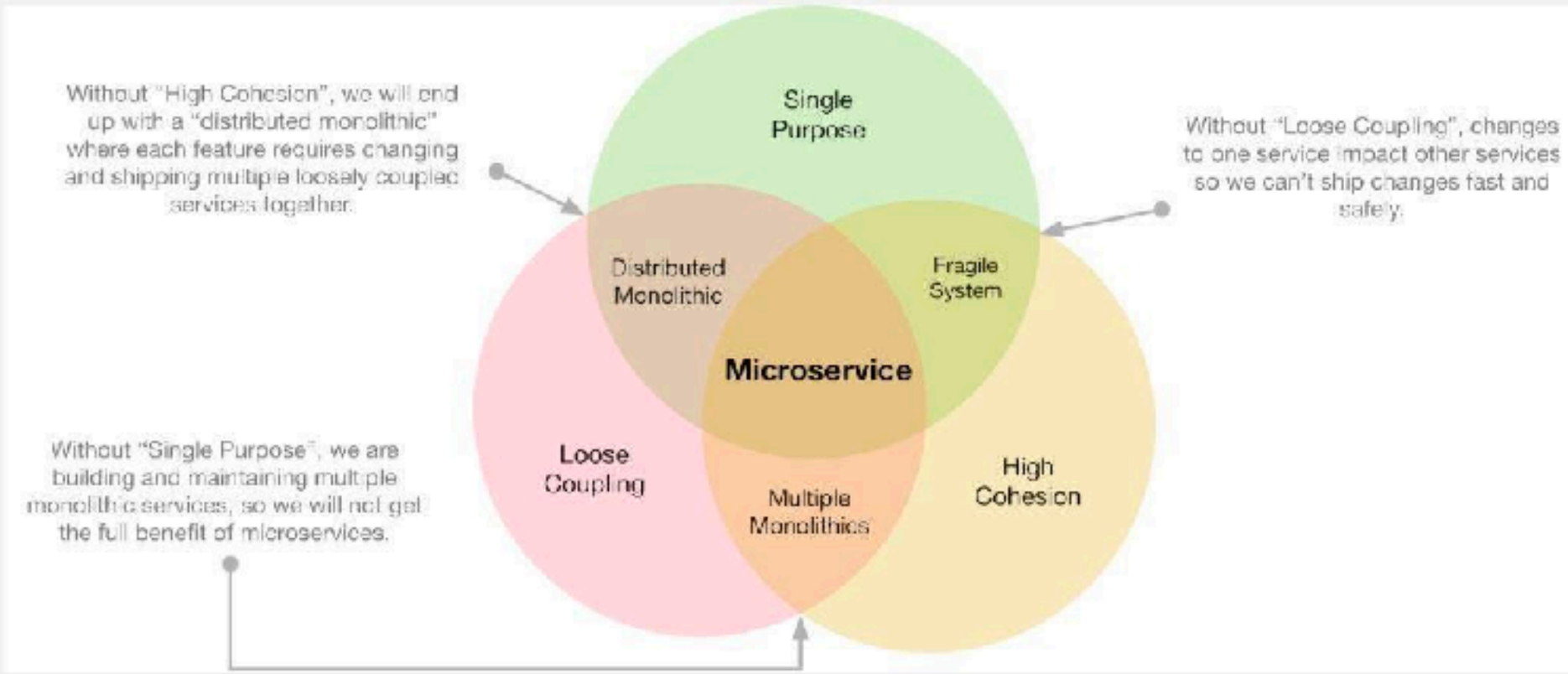


Coupling (Destructive decouple)



Better Architecture

Three principles for well-architected microservices. Source: Ma 2018.*



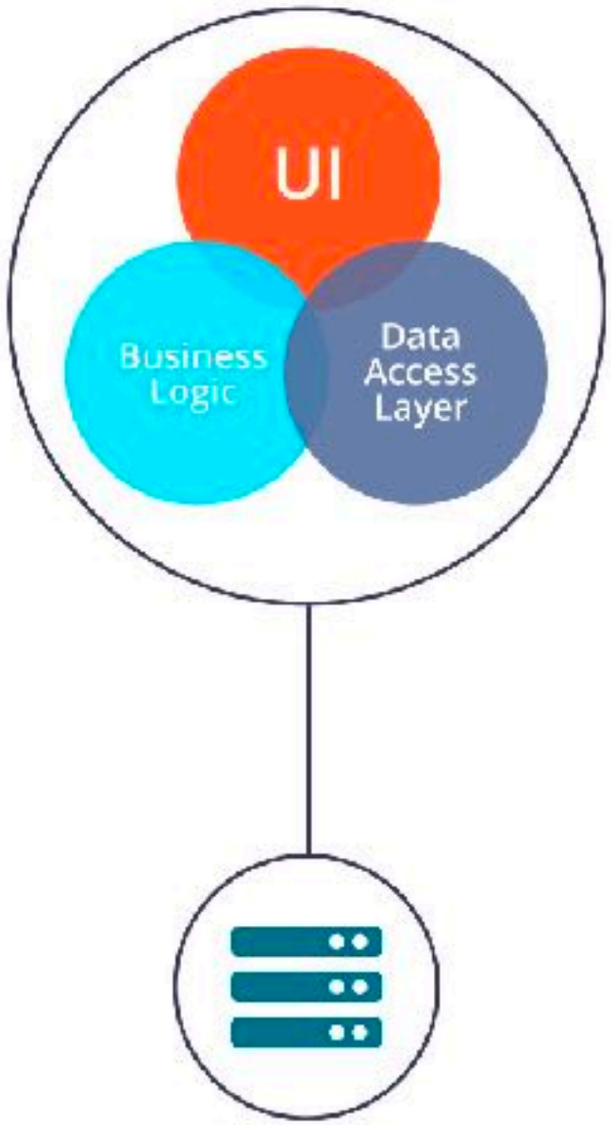
WANT TO KNOW THE BEST SOFTWARE ARCHITECTURE?

Pangaea
300M years ago

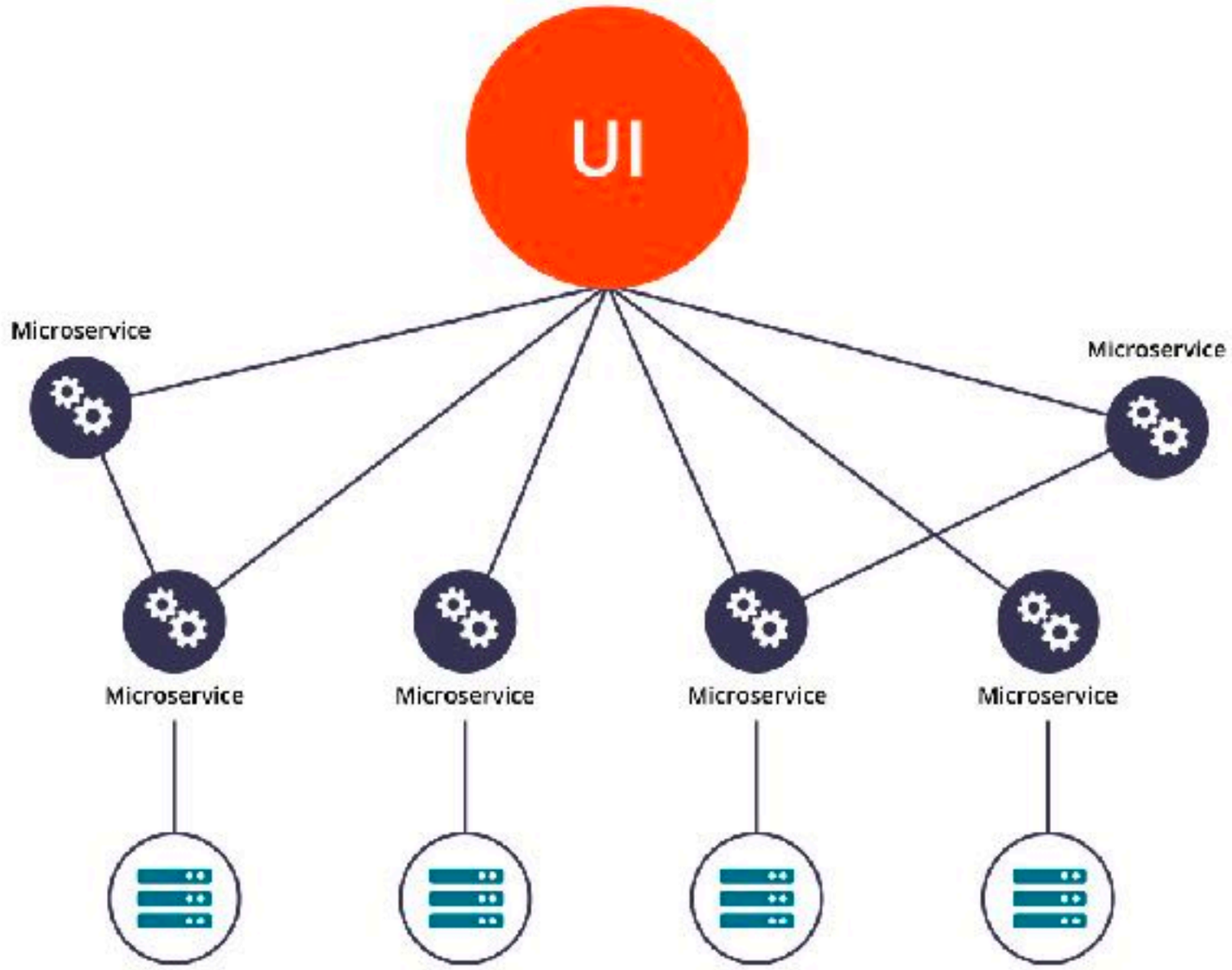
LOOK AT THE EVOLUTION OF THE EARTH.

nowadays

Daniel Stori {turnoff.us}



Monolithic Architecture



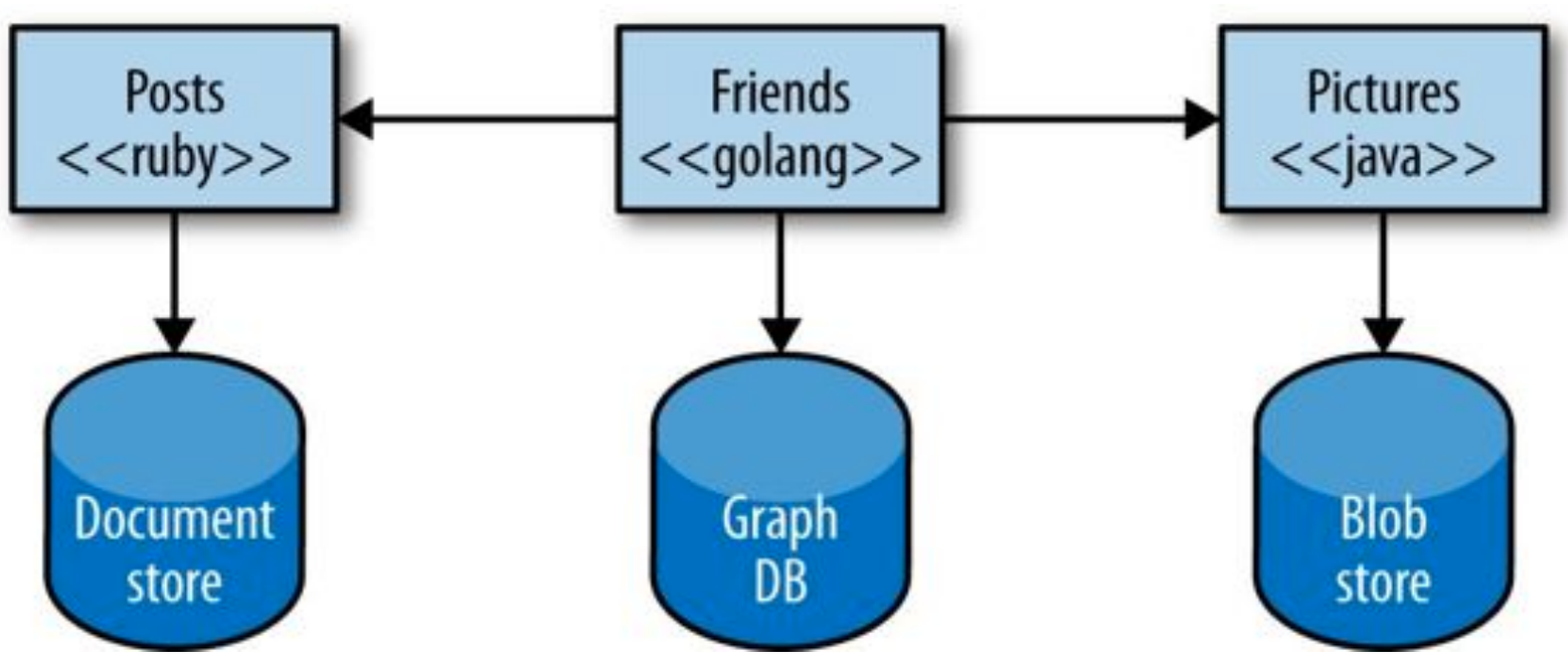
Microservice Architecture

Key Benefits

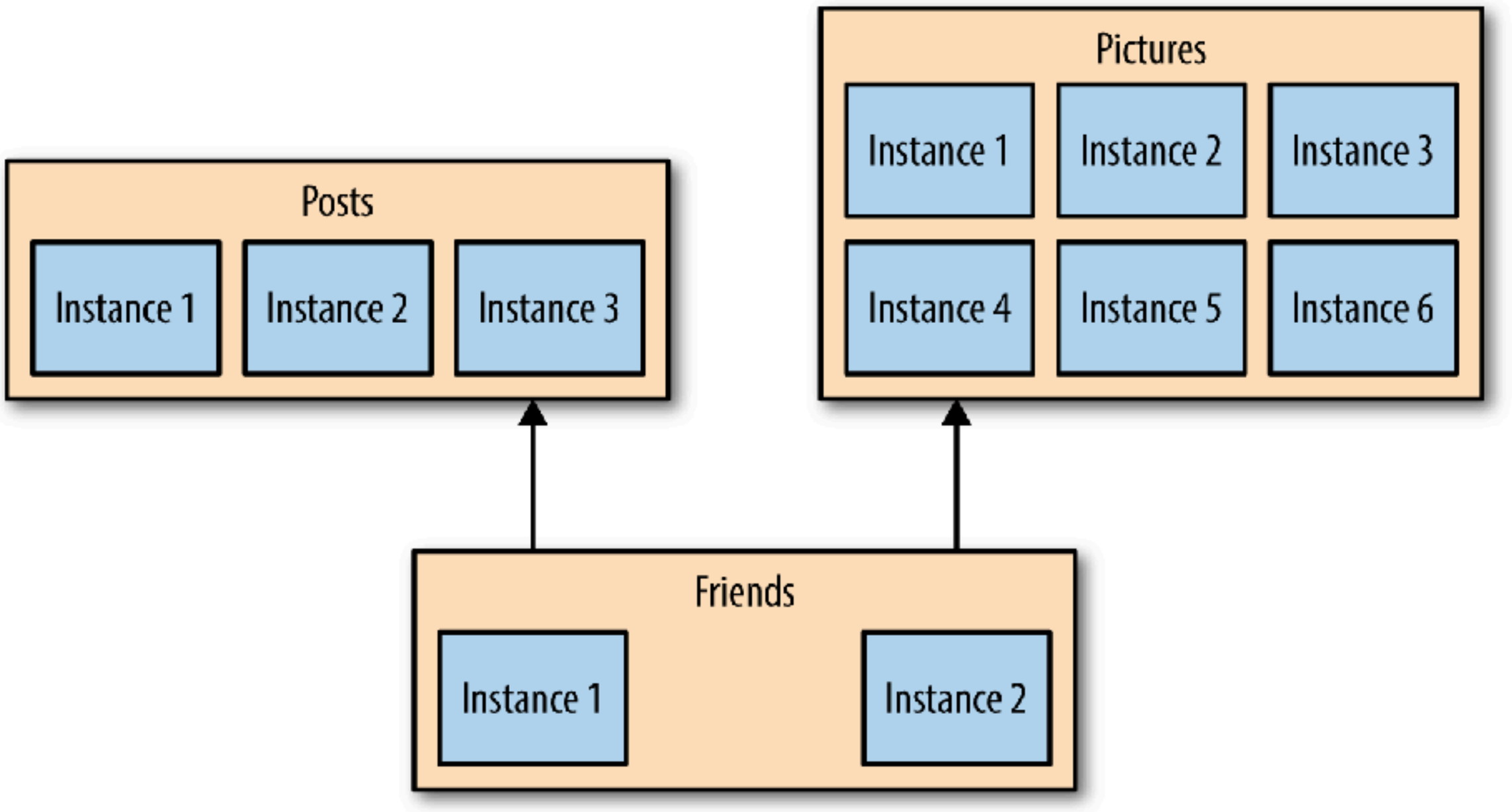
1. Technology heterogeneous

The right tool for each job

Allow easy experimenting and adoption of new technology

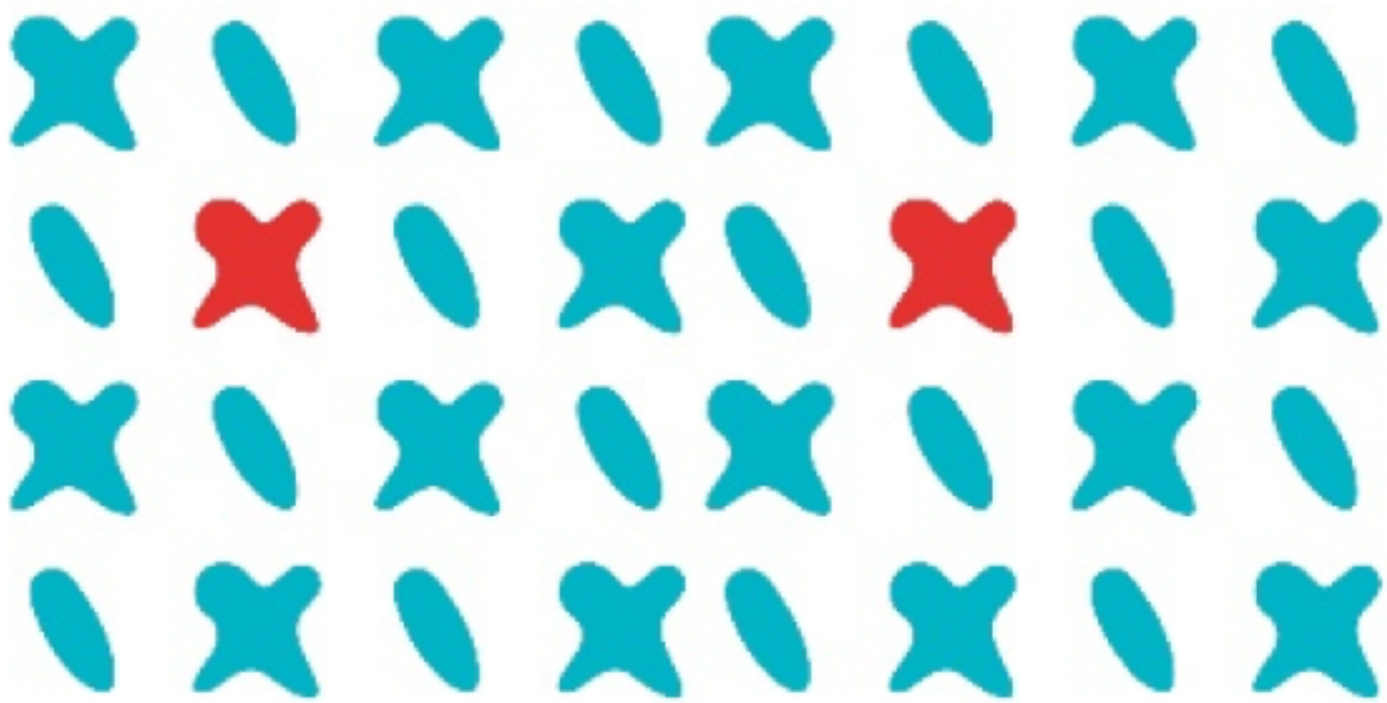


2. Scaling

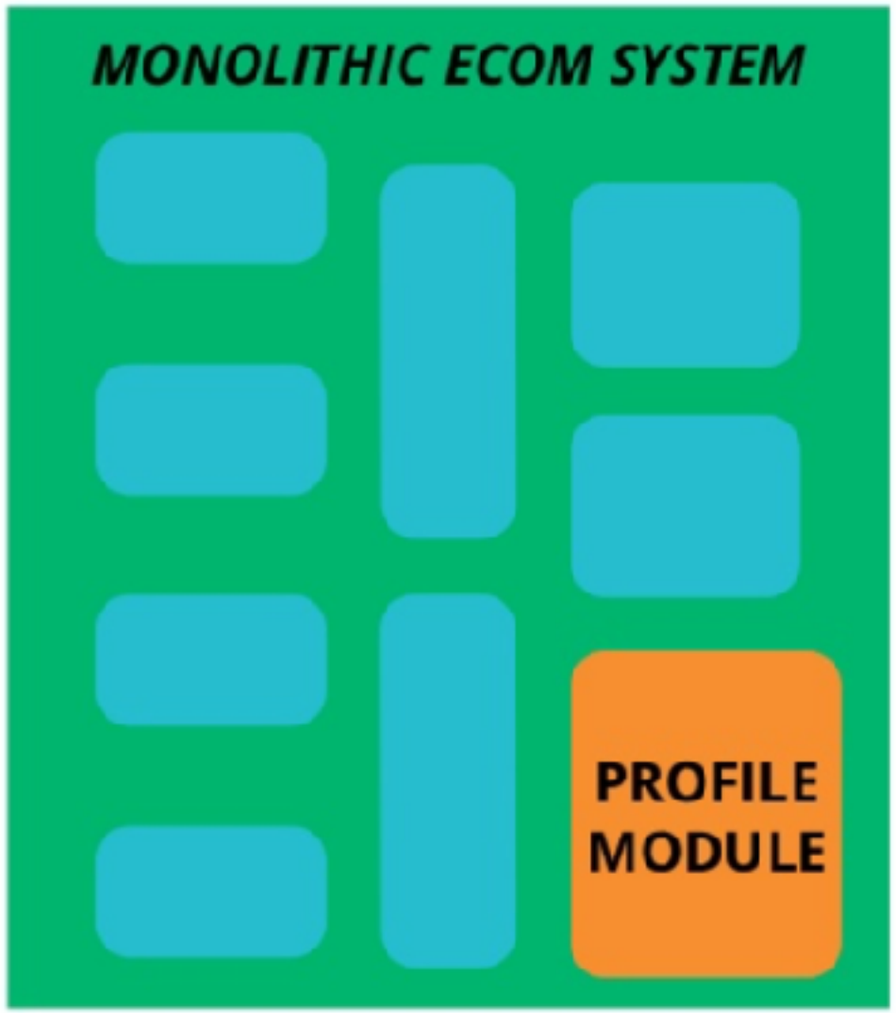


3. Ease of deployment

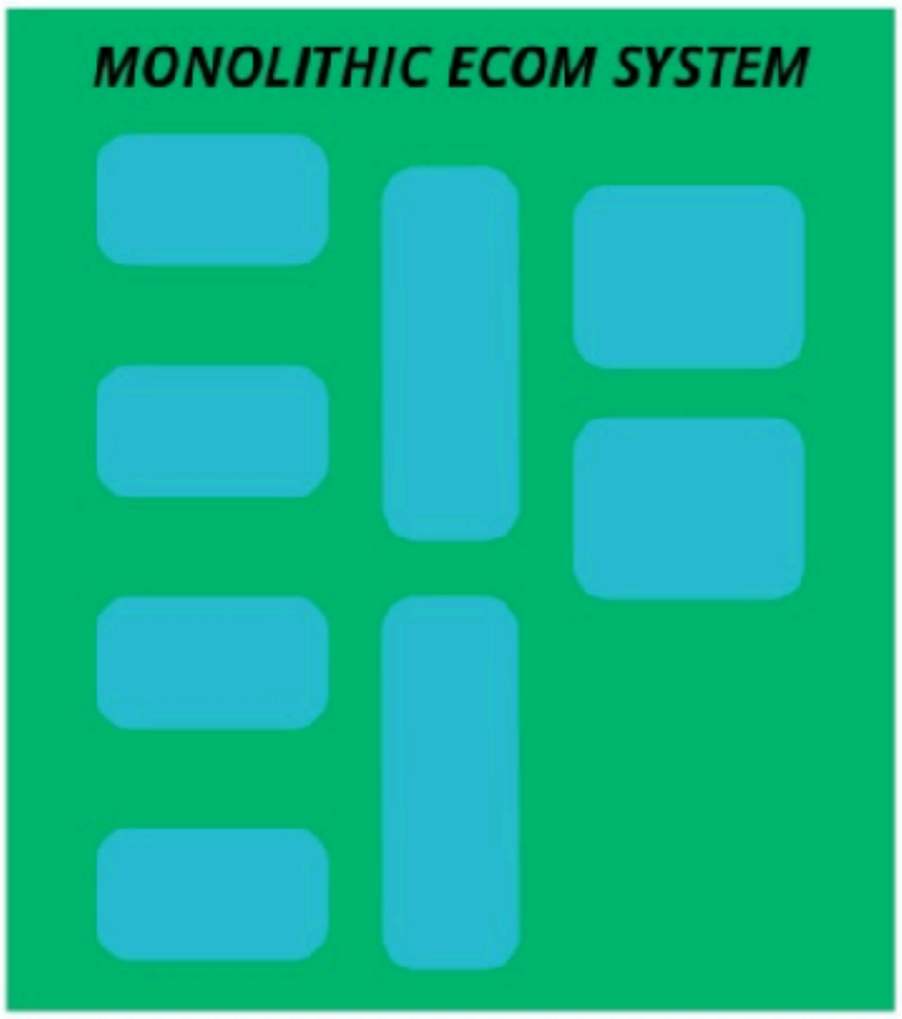
Deploys are faster, independent and problems can be isolated more easily



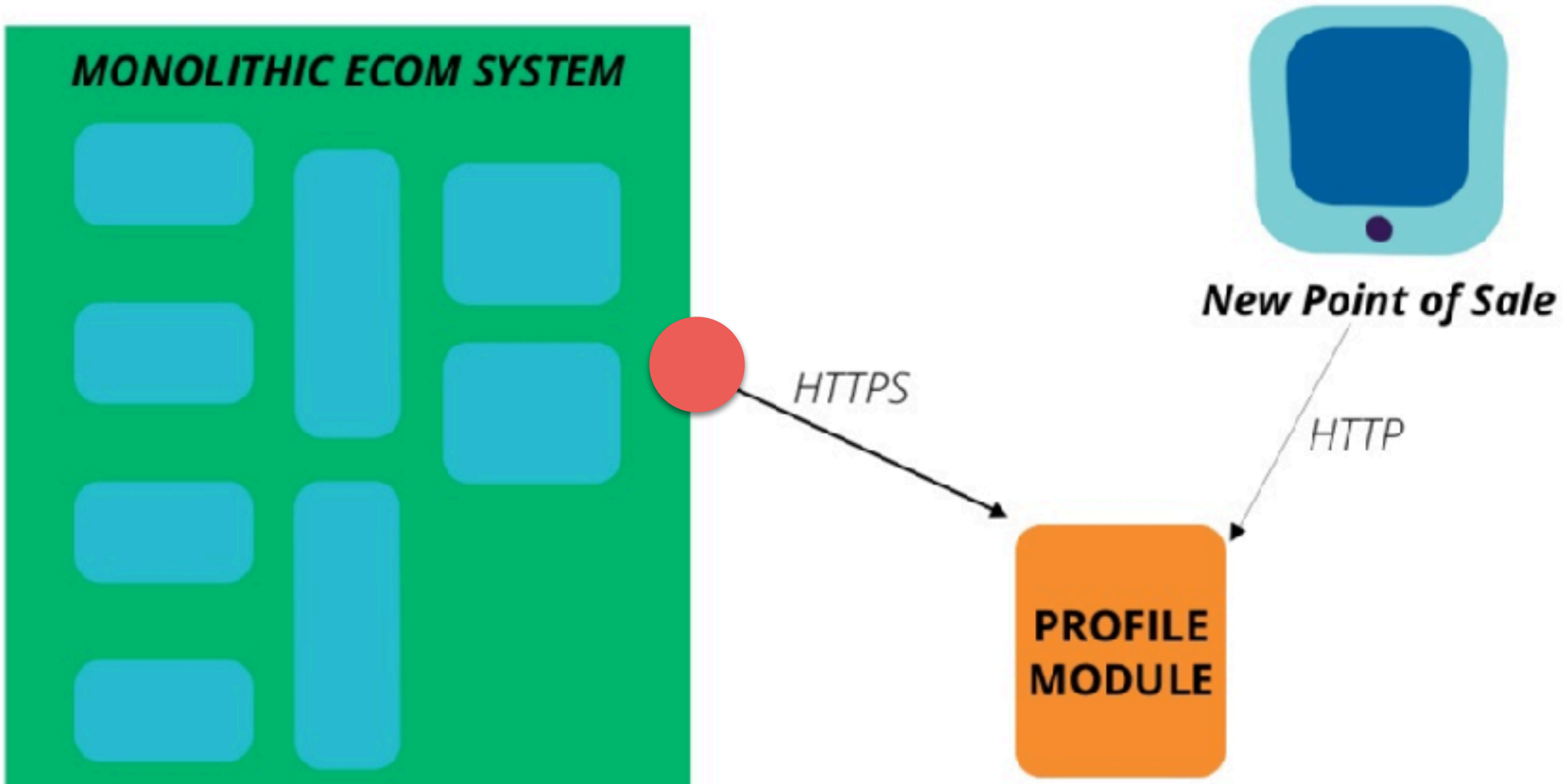
4. Composability and replaceability



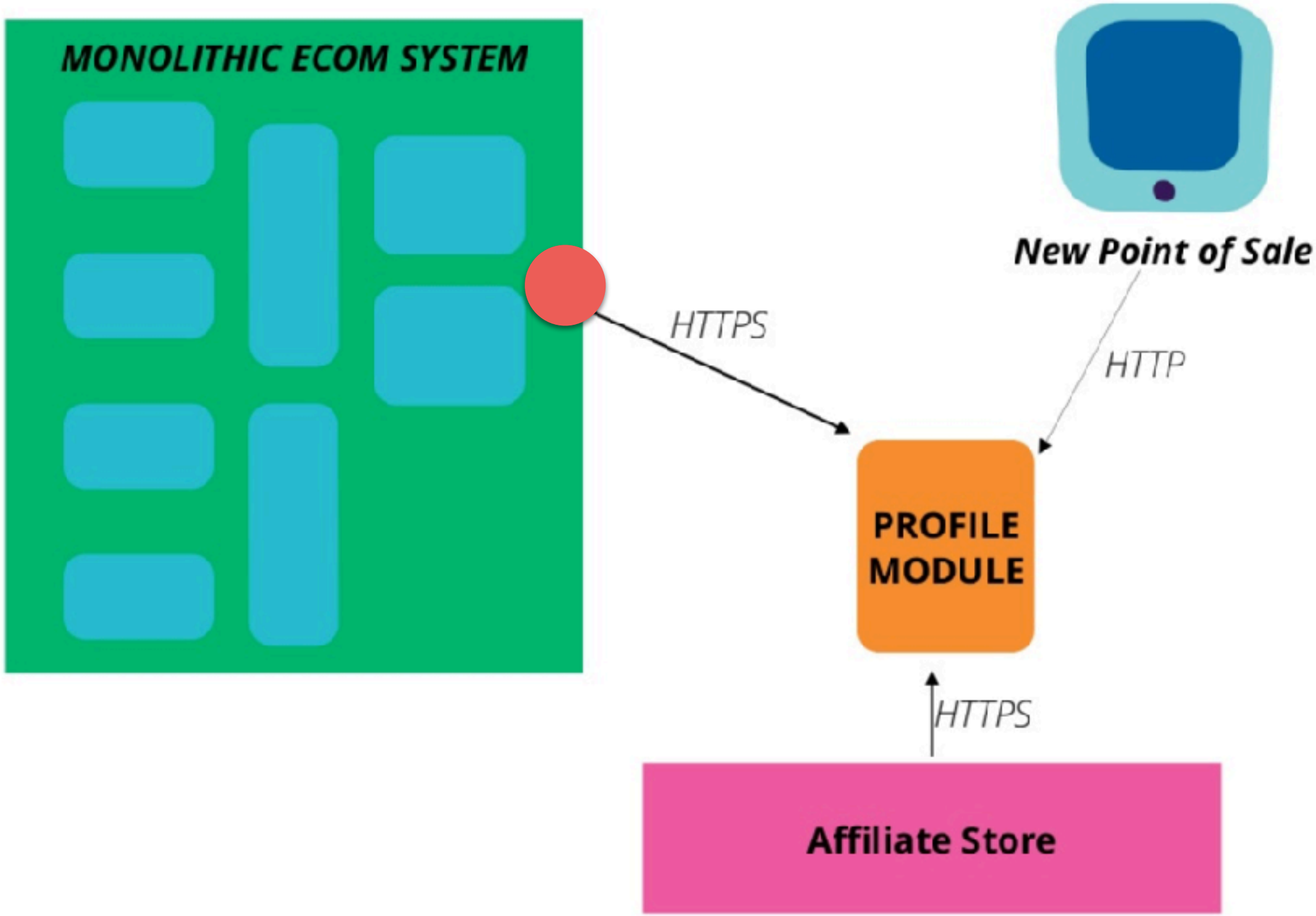
4. Composability and replaceability



4. Composability and replaceability



4. Composability and replaceability

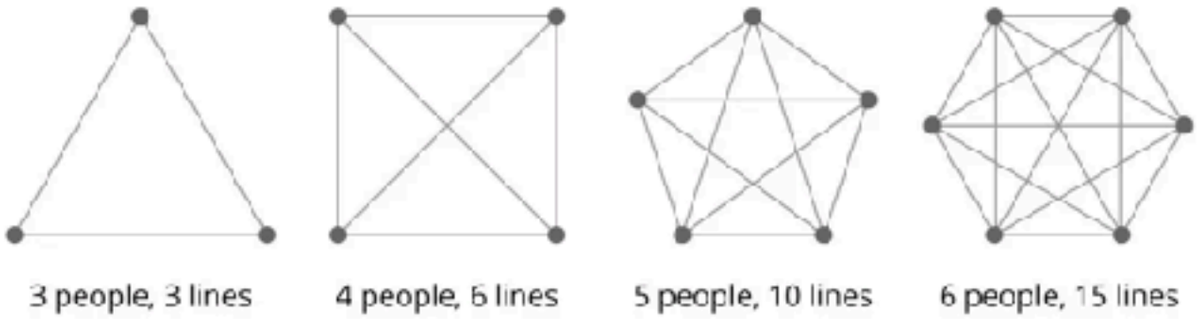


5. Organization alignment

Small teams and smaller codebases

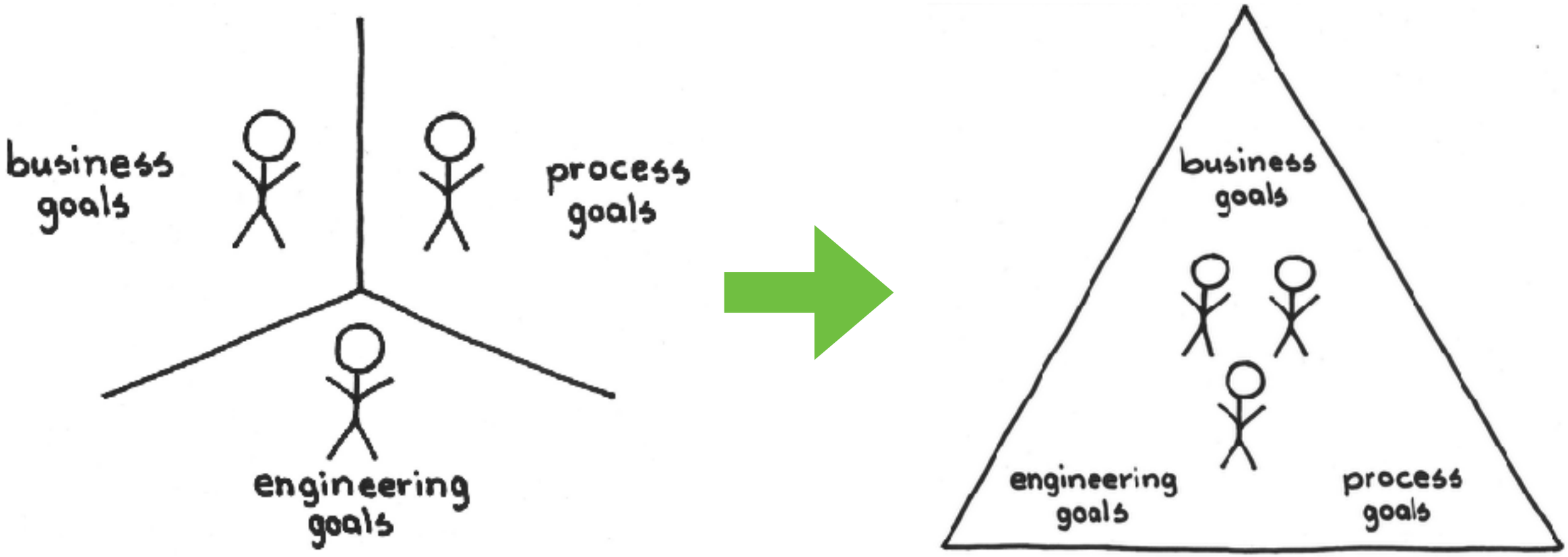
Small team !!

Line of communication and team size

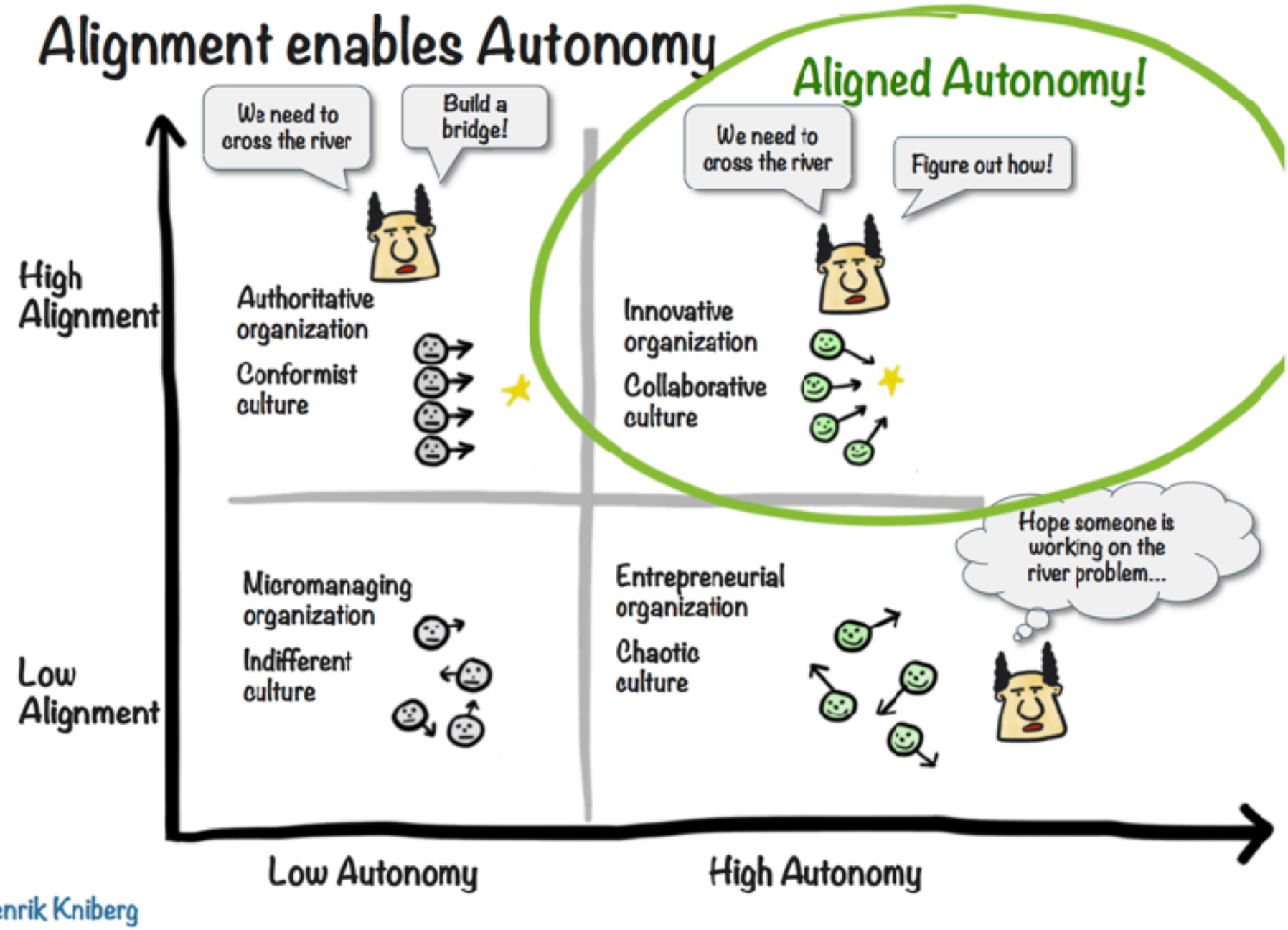


https://www.leadingagile.com/2018/02/lines-of-communication-team-size-applying-brooks-law

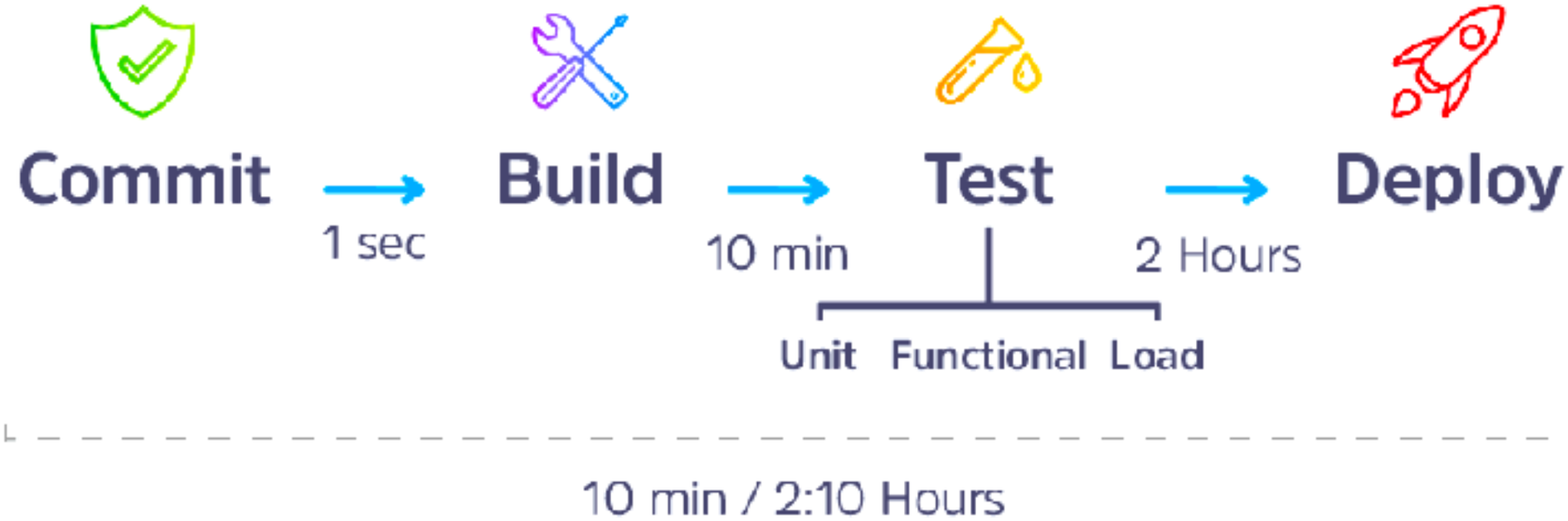
Autonomous team



Autonomous team



Autonomous team

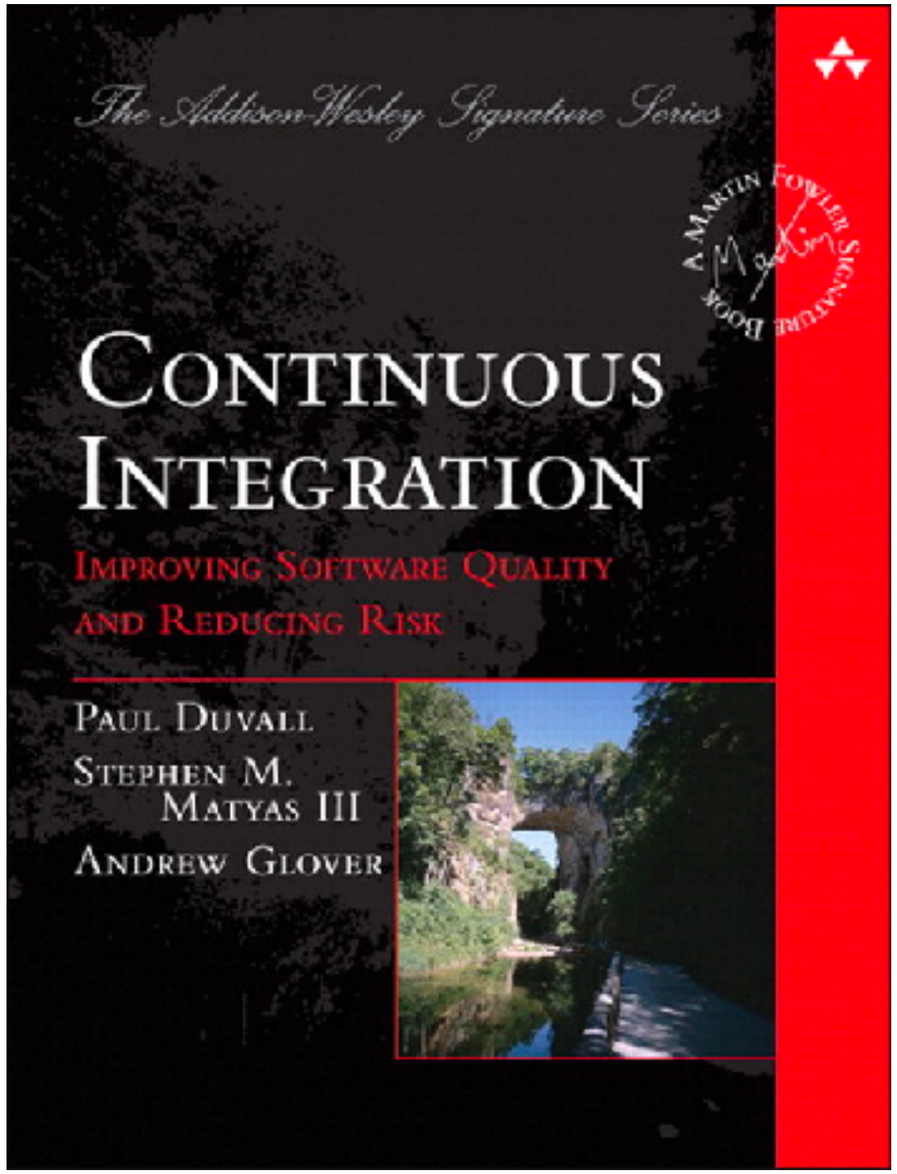


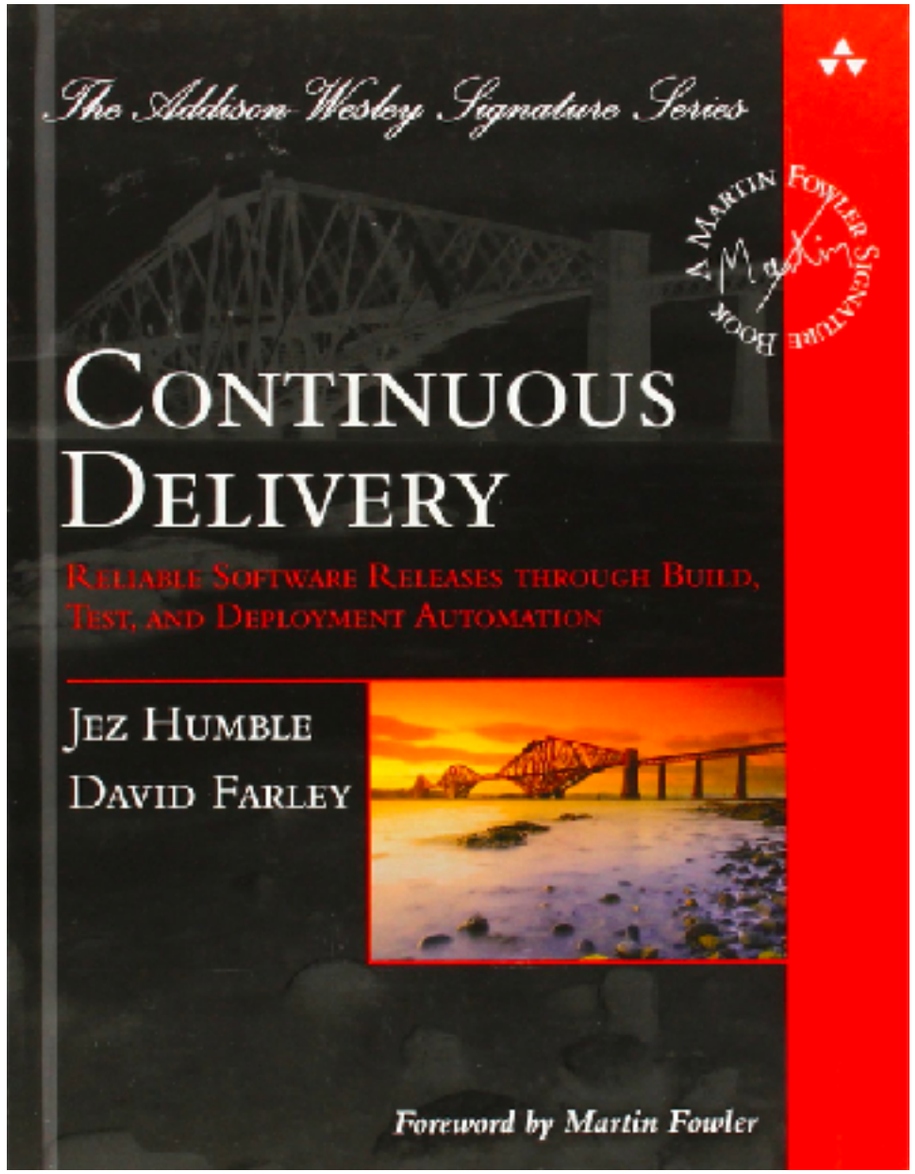
6. Enable Continuous Delivery

A part of **DevOps**

Set of practices for the **rapid**, **frequent** and **reliable delivery** software

Continuous Delivery/Deployment





Continuous Delivery/Deployment

Testability
Deployability
Autonomous team and loose coupling

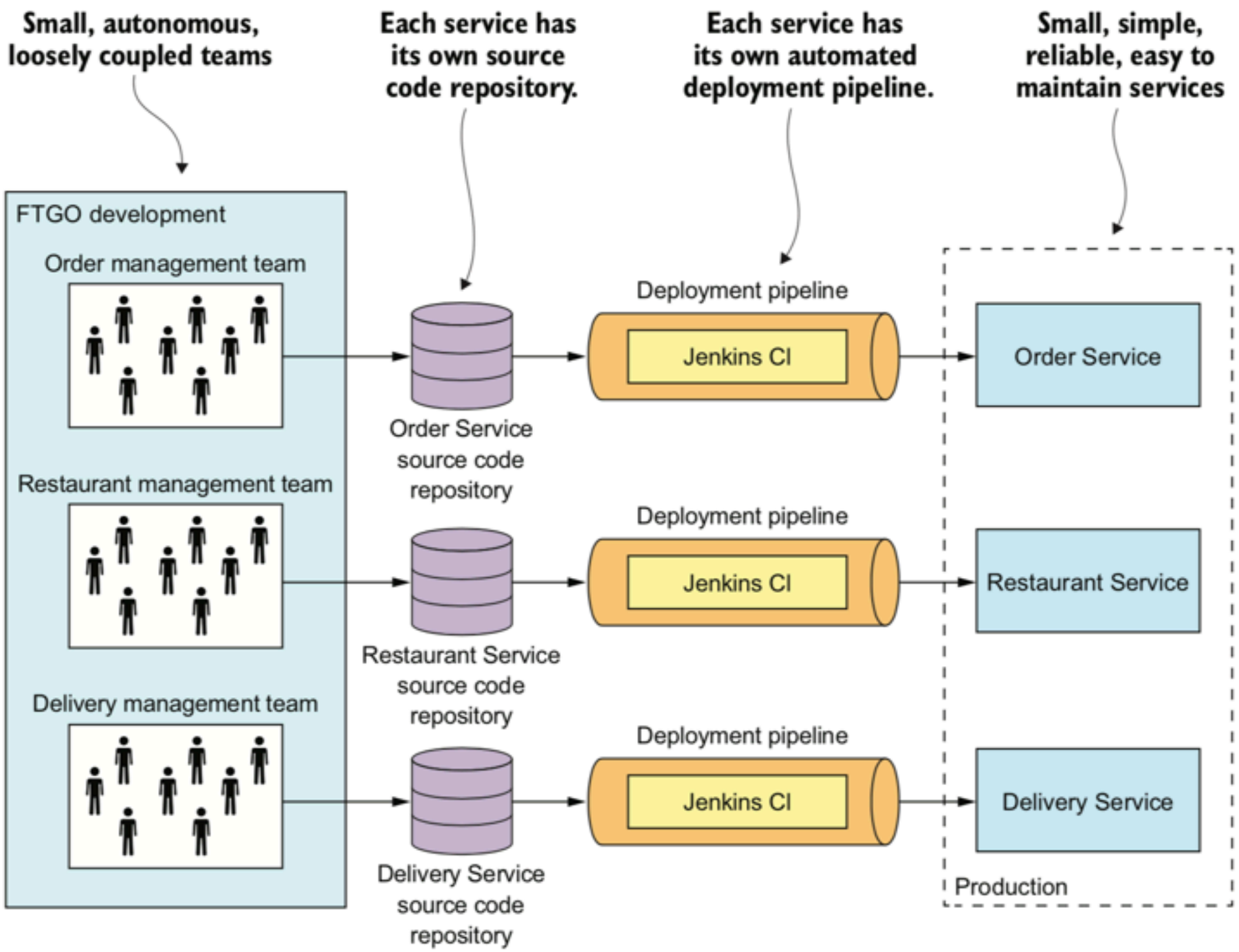
Benefits of Continuous Delivery

Reduce time to market

Enable **business** to provide reliable service

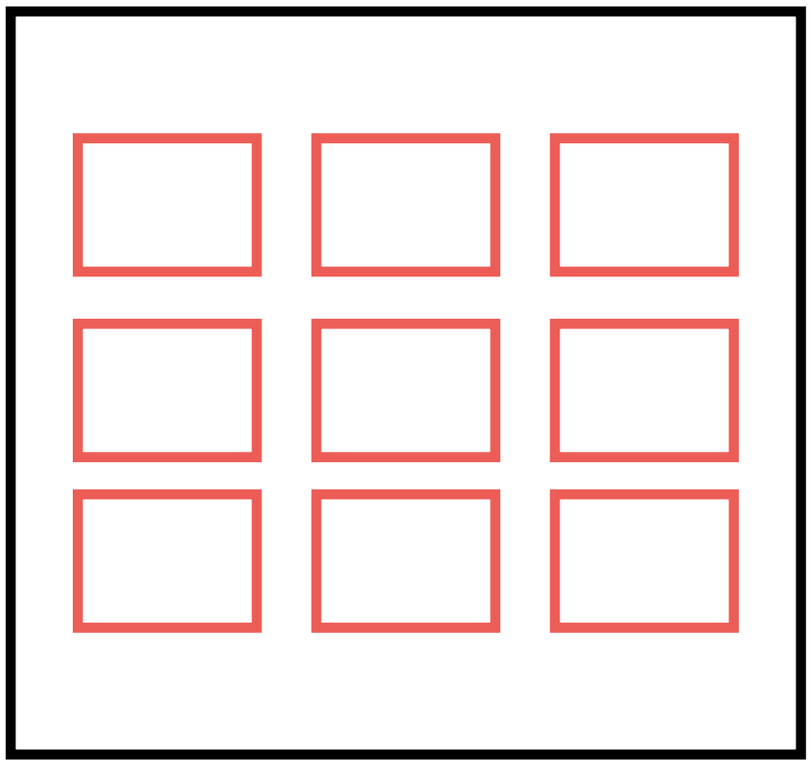
Employee satisfaction is higher

Continuous Delivery for service

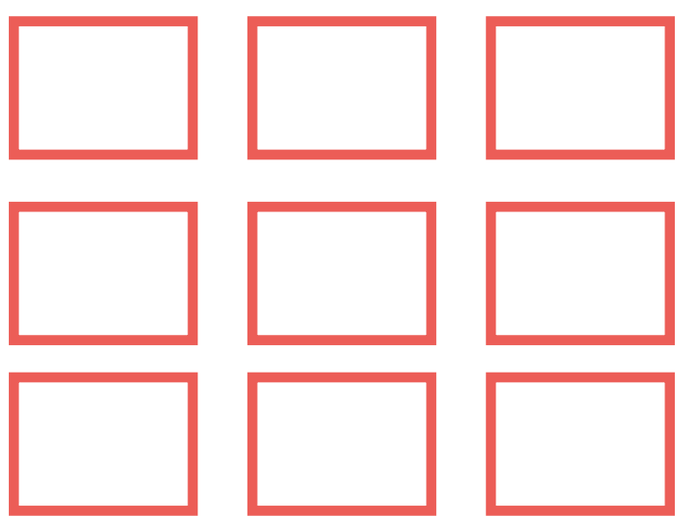


Drawbacks of Microservice

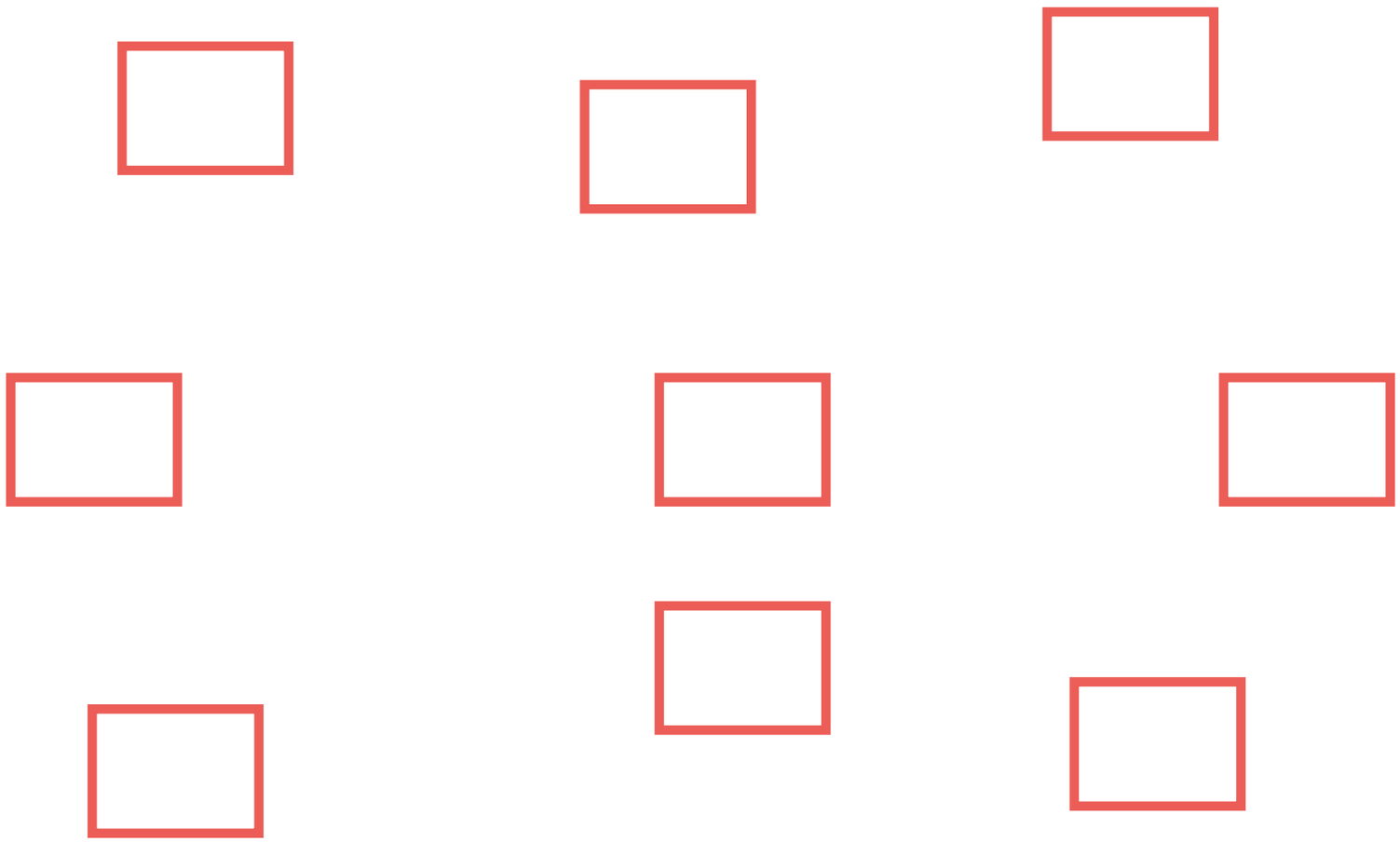
Microservice



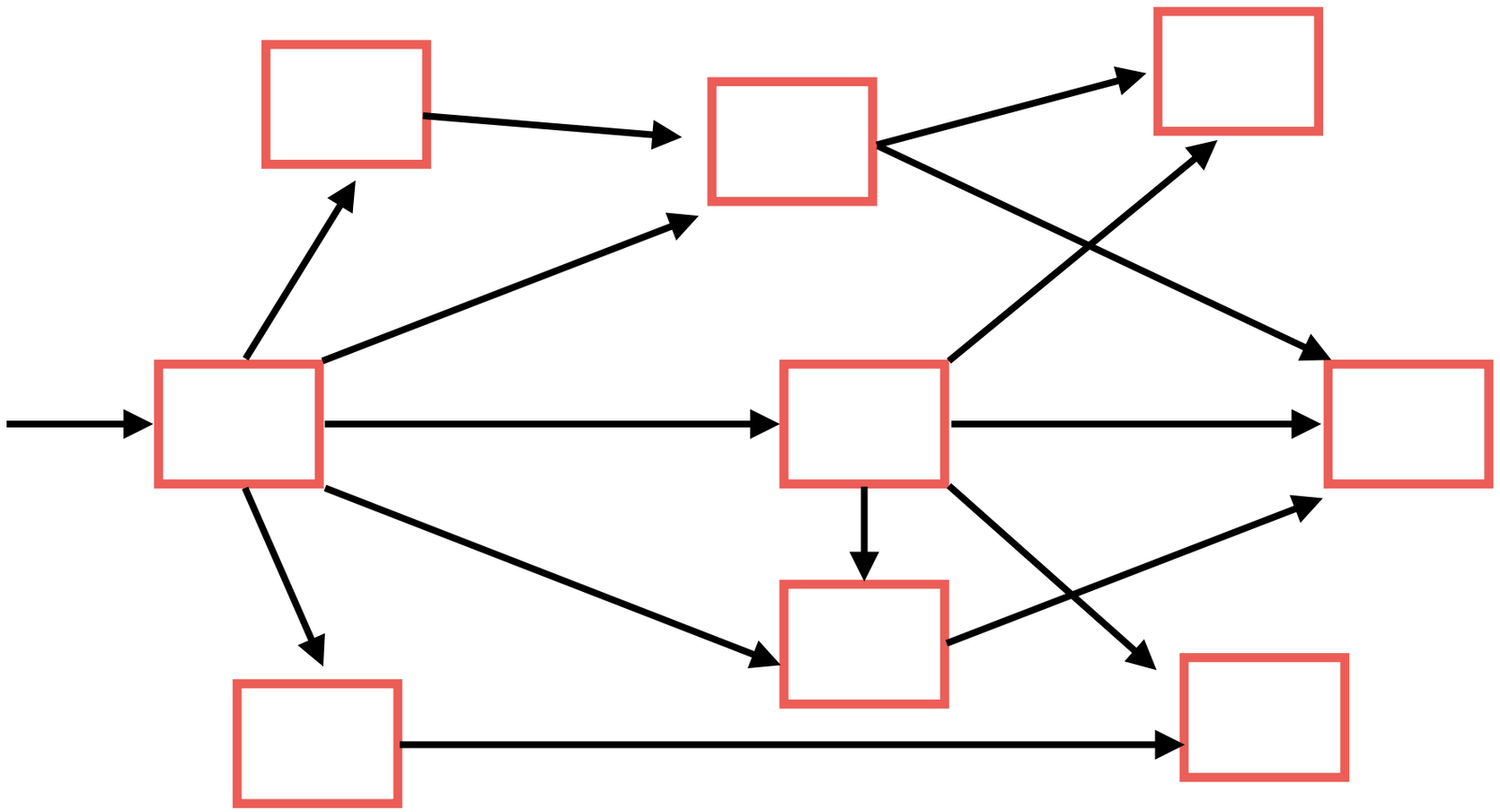
Microservice



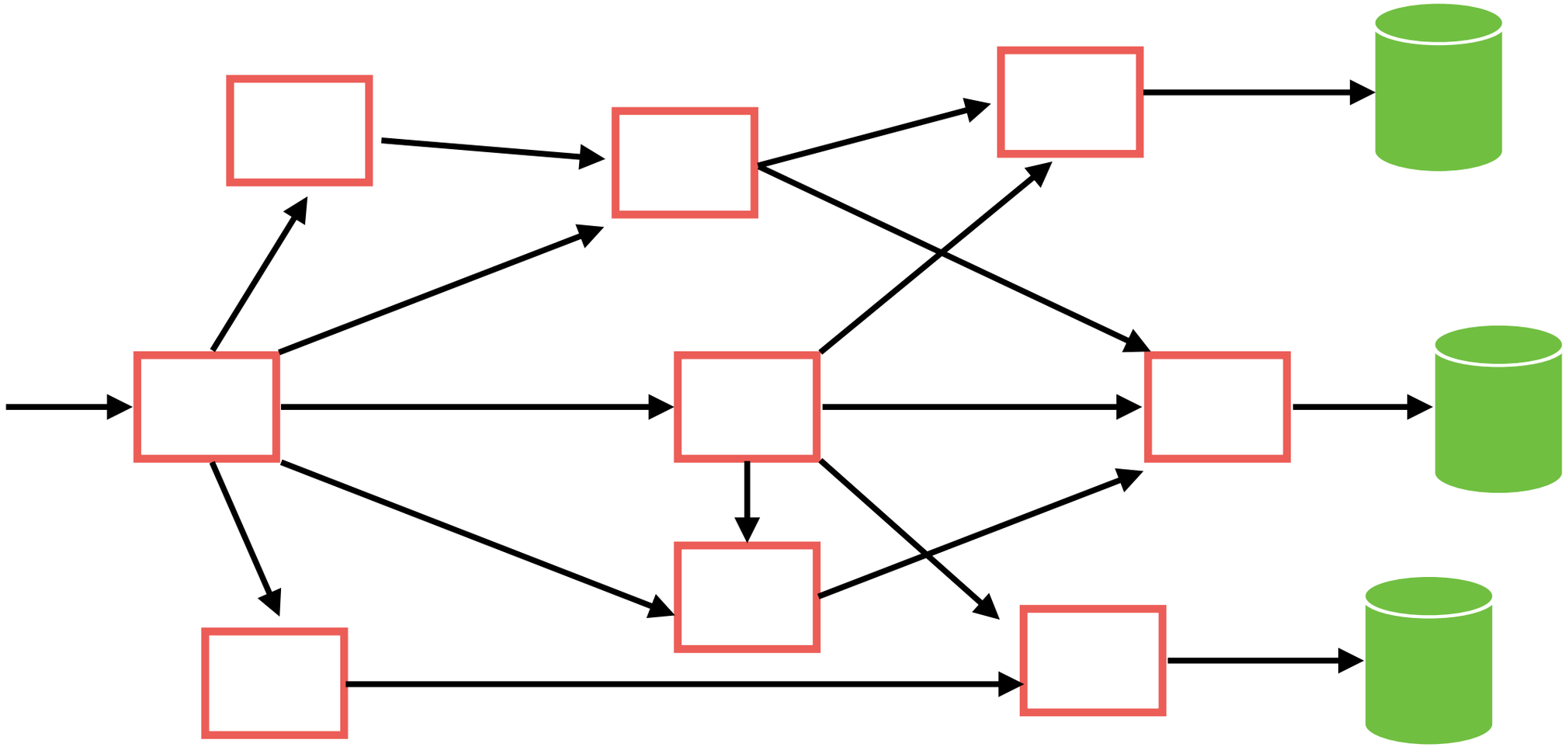
Microservice



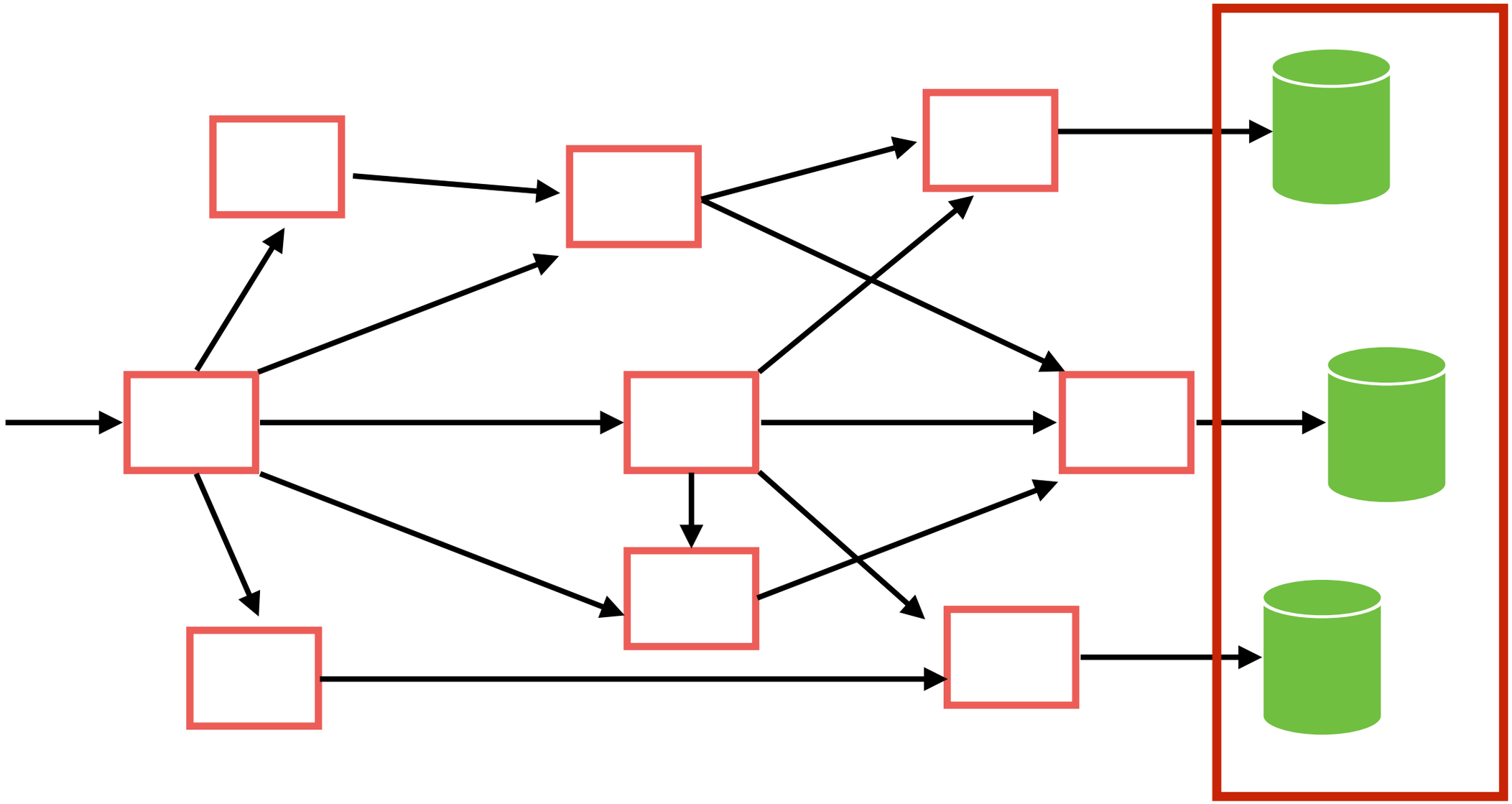
Microservice == Distributed



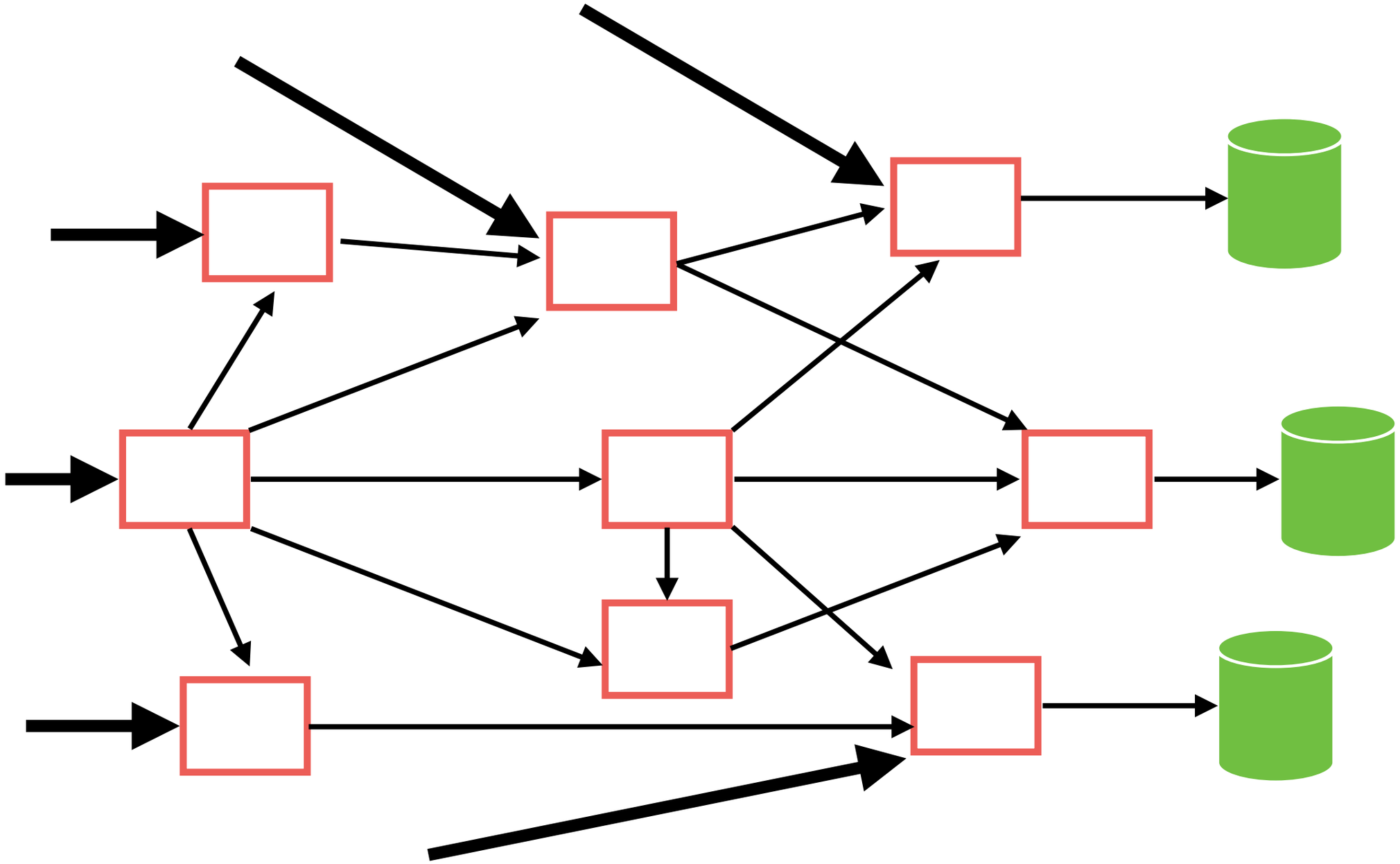
Own data



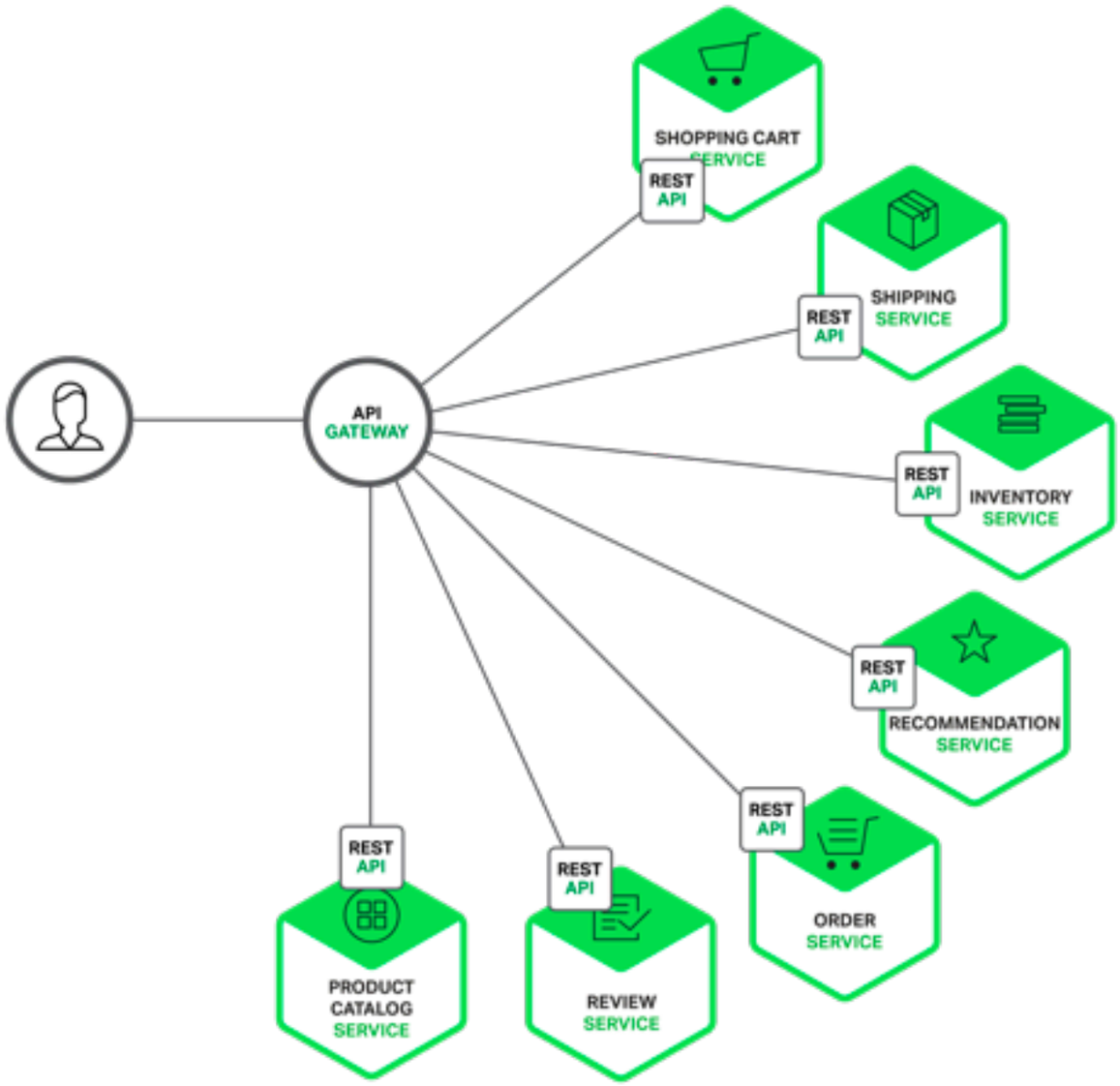
Data consistency !!



Multiple entry points

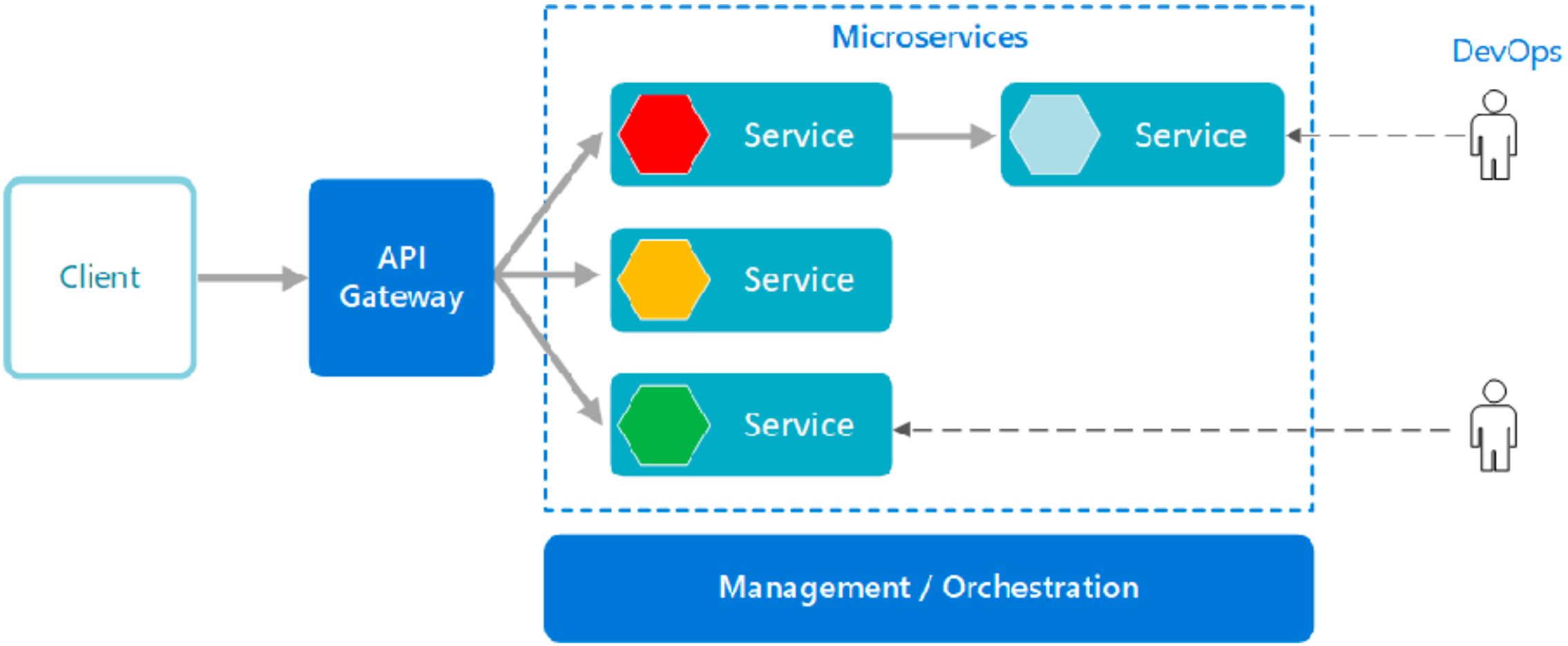


Working with API gateway



https://www.nginx.com

Working with API gateway



https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

Functions of API gateway

Authentication
Authorization
Rate limiting
Caching
Metrics collection
Request logging
Load balancing
Circuit breaking

Benefits

Encapsulation interface structure
Client talk to service via gateway
Reduce round trip between service

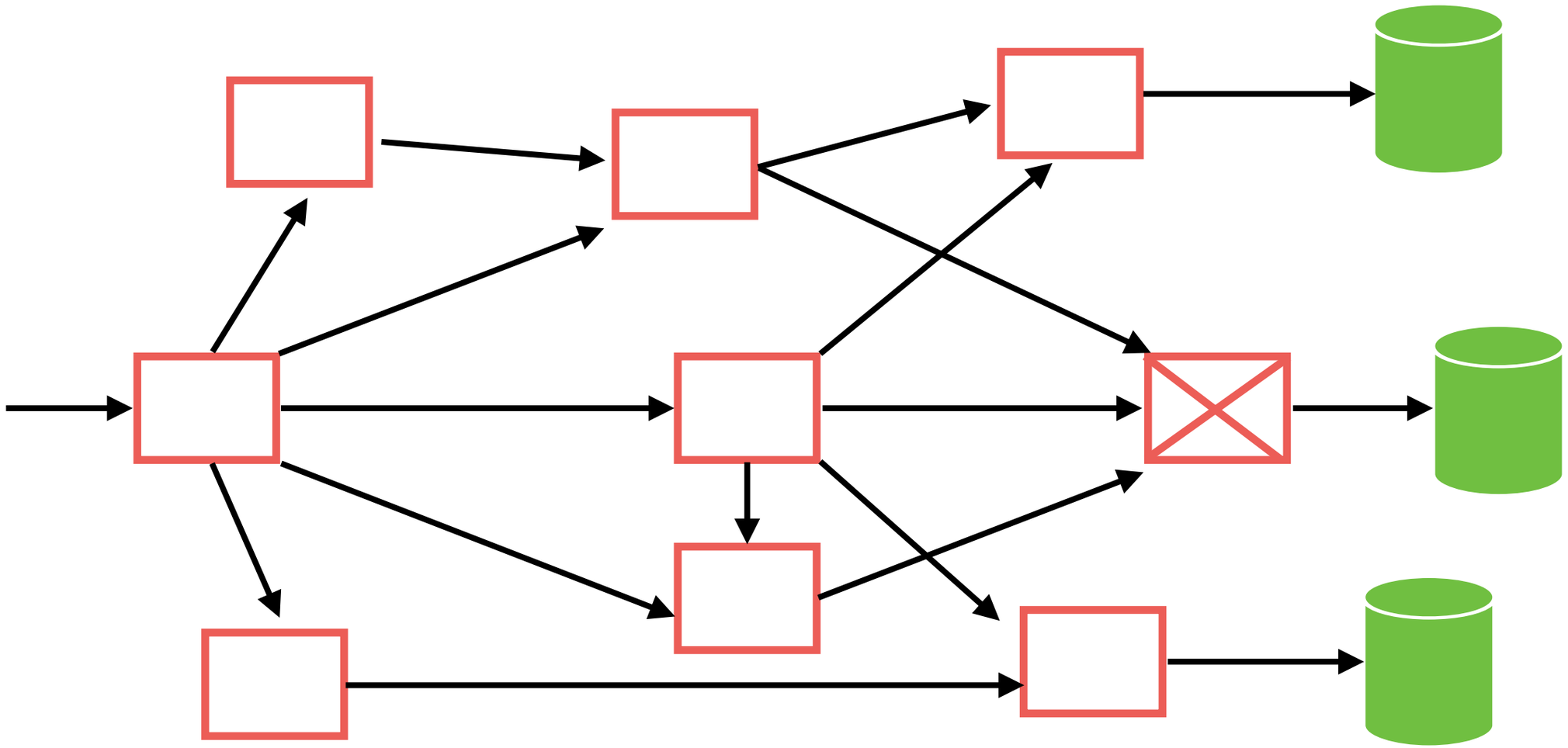
Drawbacks

Development/Performance bottleneck

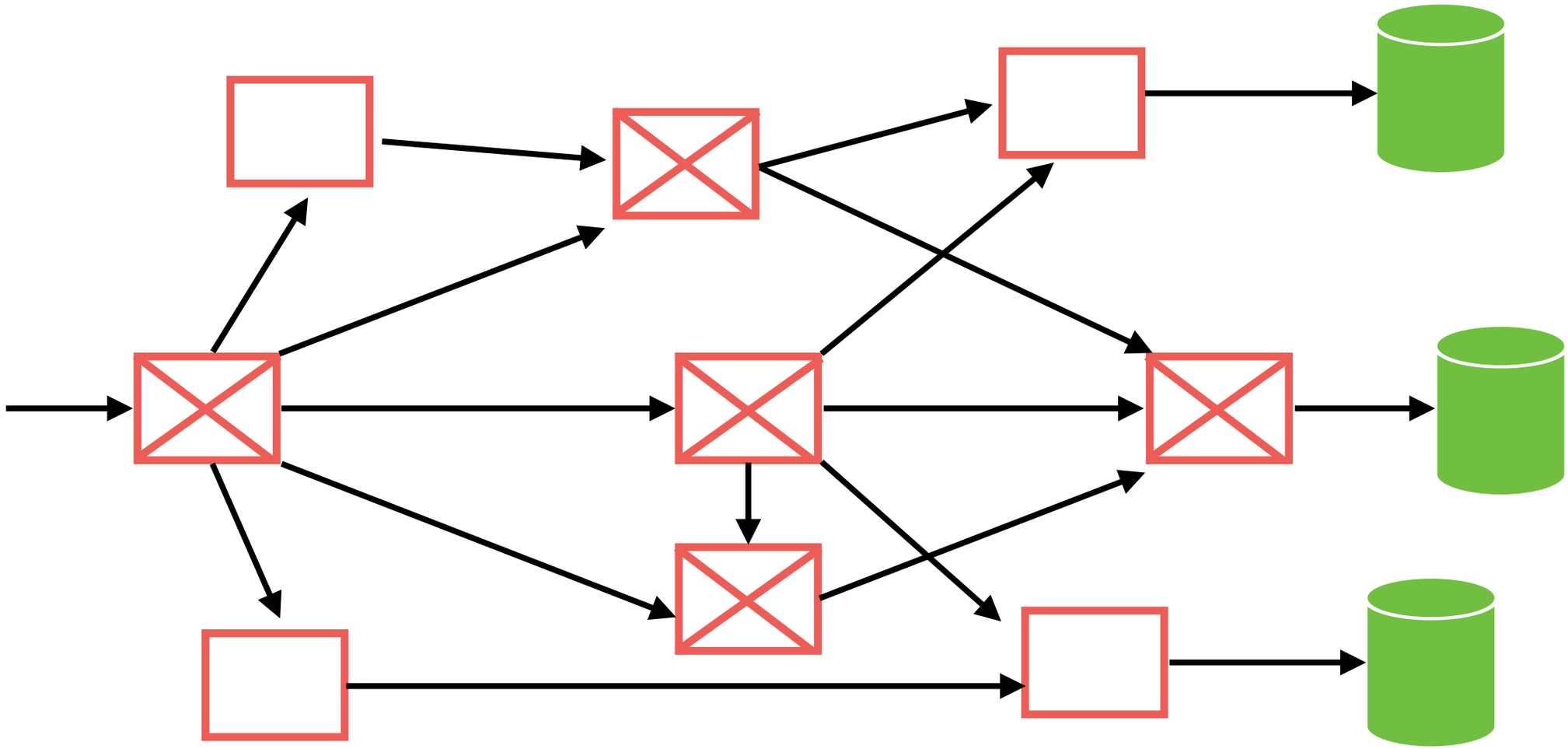
Single point of failure

Waiting for update data in gateway

Failure !!



Cascading Failure !!



Solutions ?

Solutions for Reliability

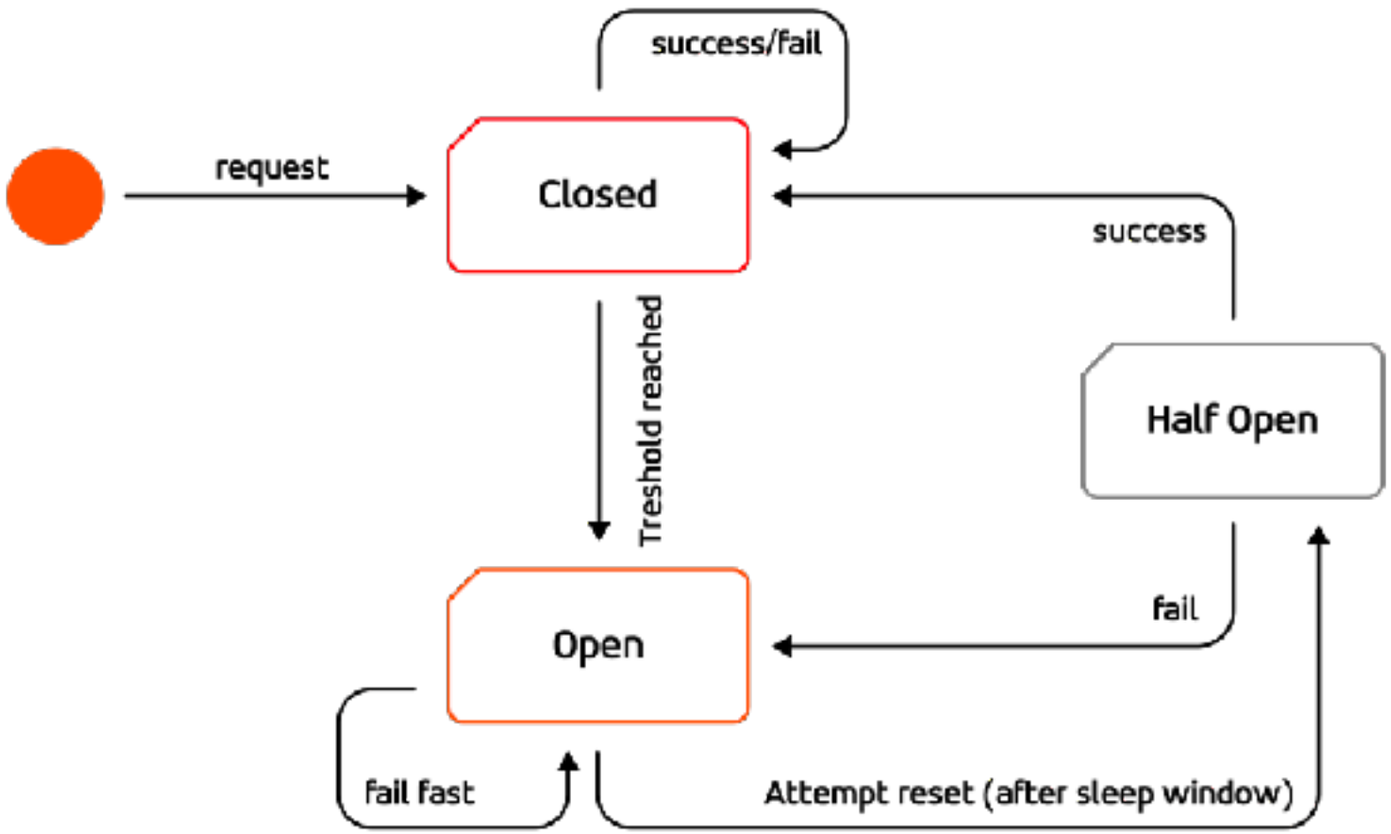
Timeout
Retry
Rate Limit
Queue
Circuit breaker
Bulkhead
Auto-scale

https://learn.microsoft.com/en-us/azure/well-architected/resiliency/reliability-patterns

Circuit Breaker pattern

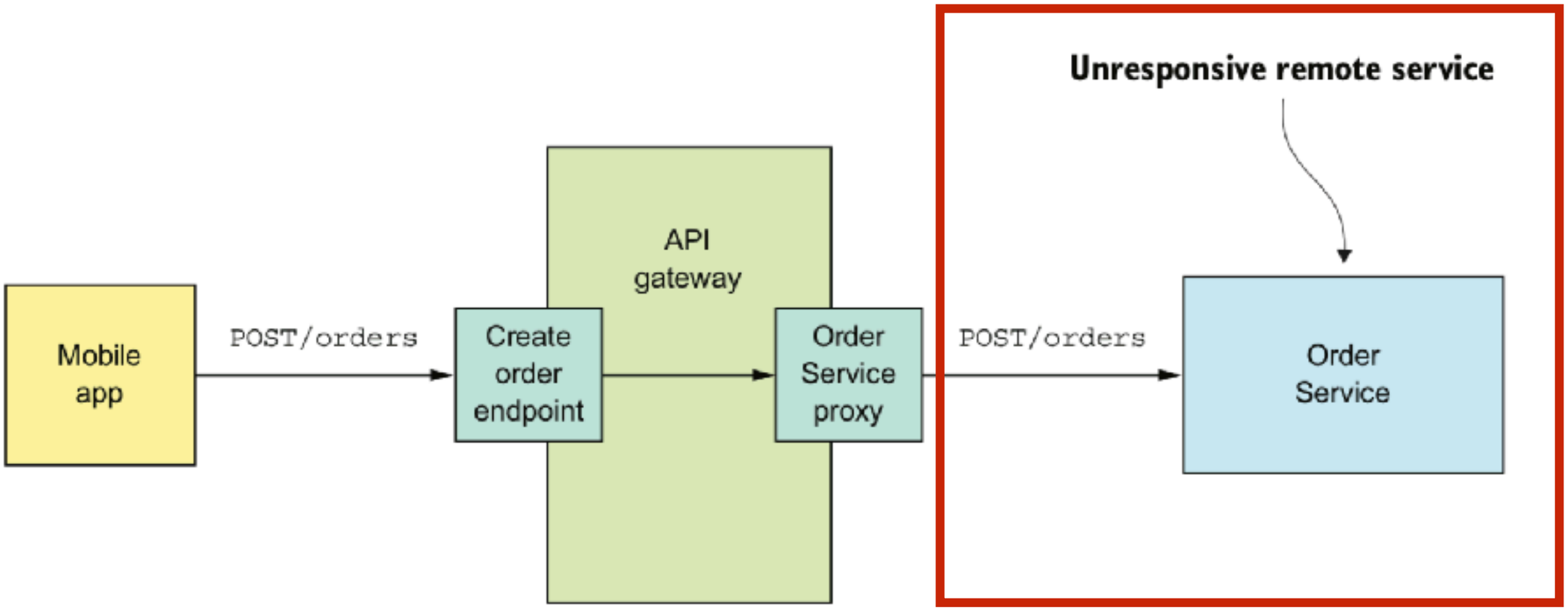
Track the number of success and failure

If error rate exceed some threshold then enable circuits breaker



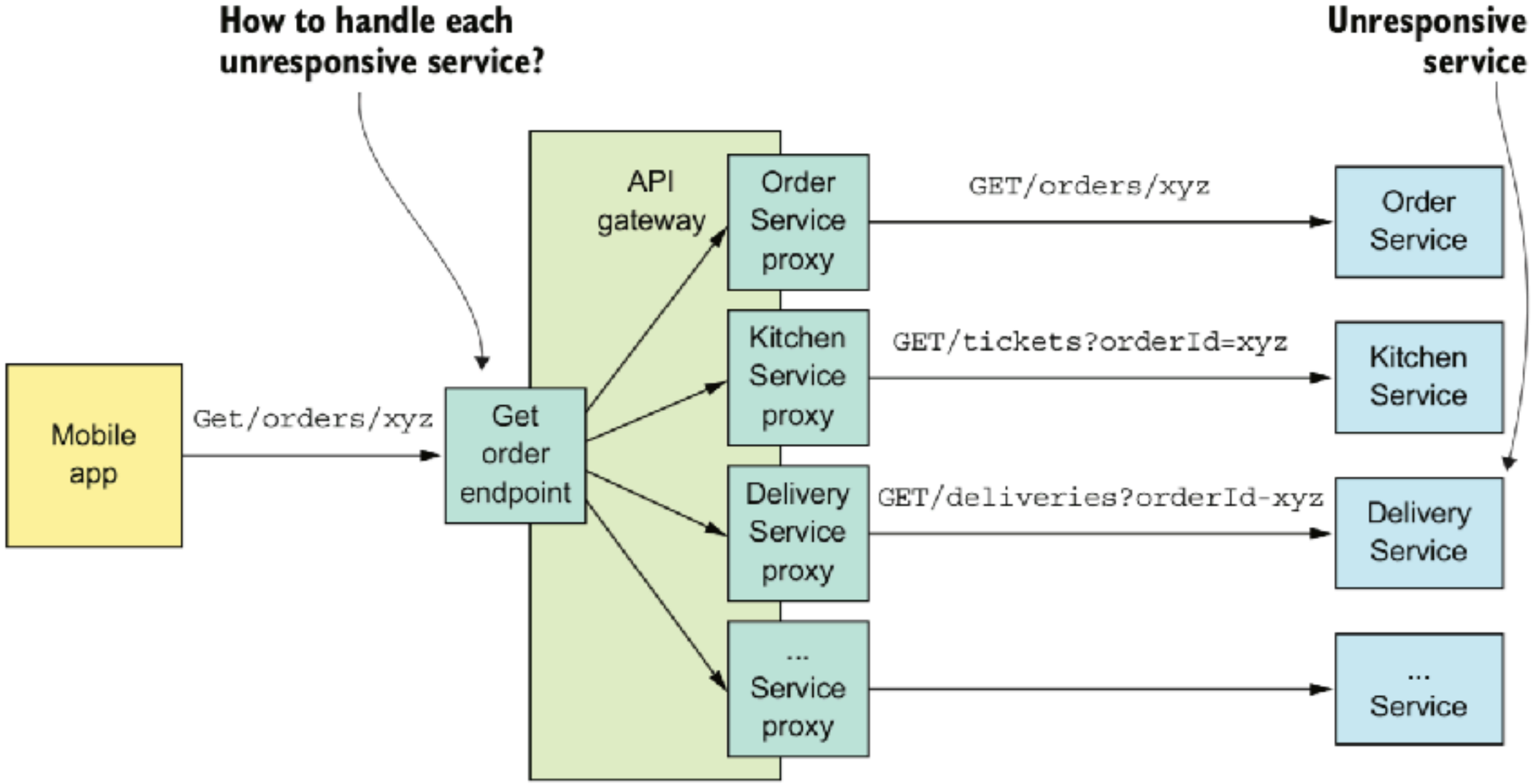
Circuit Breaker pattern

How to handling partial failure ?



Circuit Breaker pattern

How to handling partial failure ?



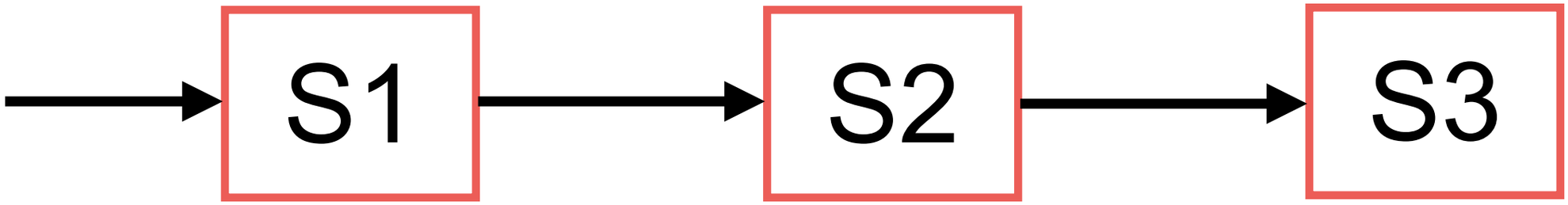
Circuit Breaker pattern

Drawbacks of Microservice

Find the right set of services
Distributed systems are complex
How to develop, testing and deploy ?

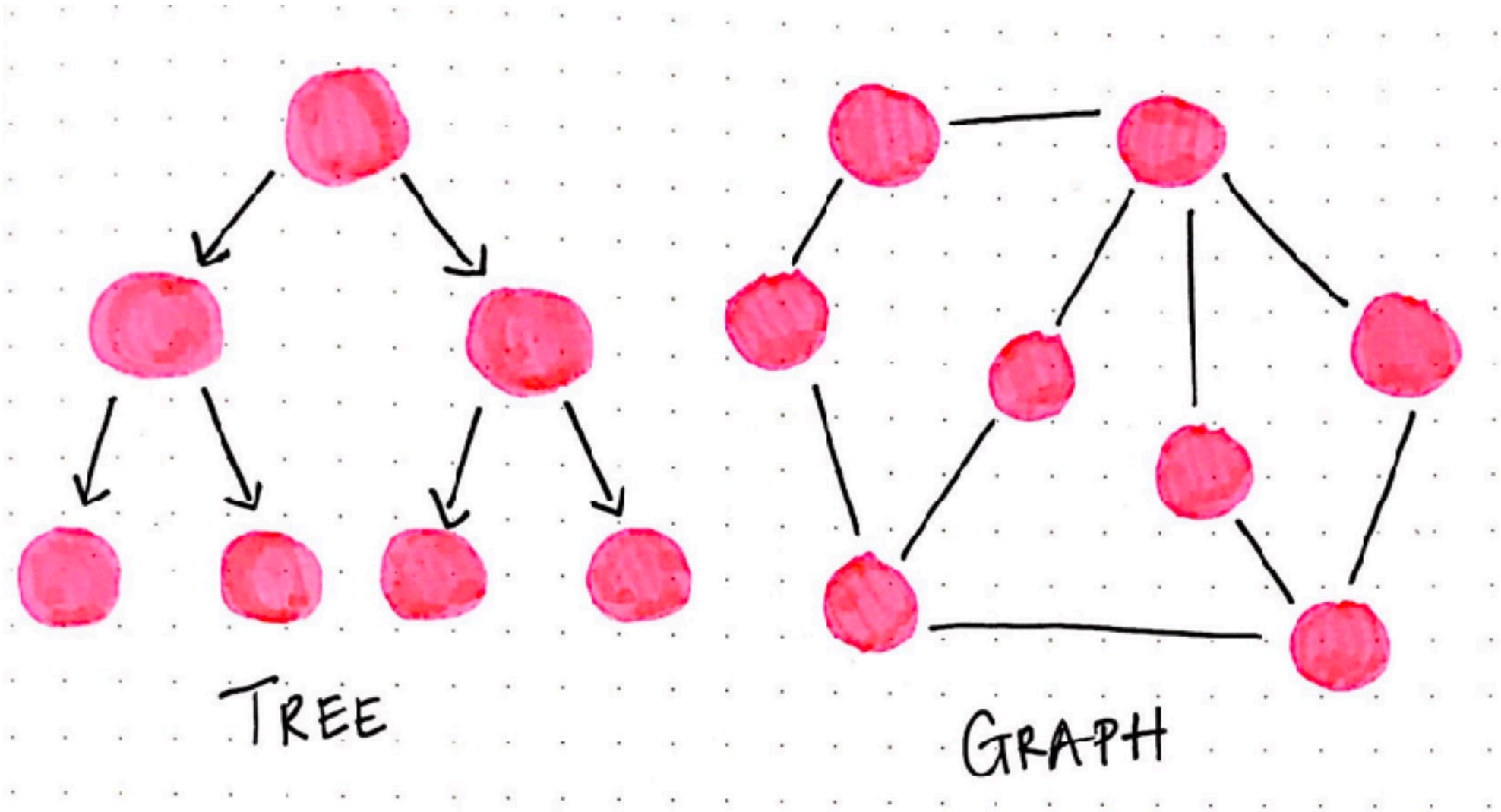
Drawbacks of Microservice

Deploy feature that required multiple service ?



Service Principles

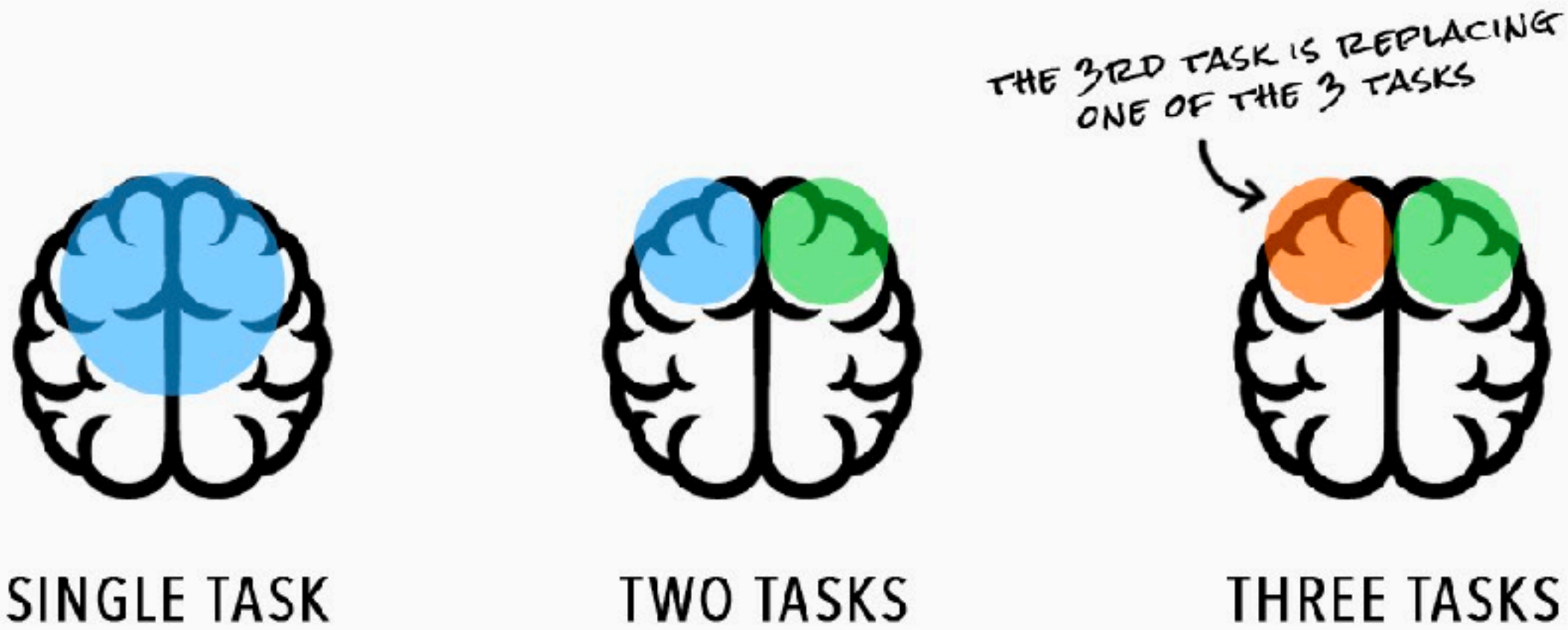
Minimize the depth of the service call-graph



https://github.com/Yelp/service-principles

Service Principles

Minimize the number of services owned by your team



https://github.com/Yelp/service-principles

Service Principles

Creation

Interface

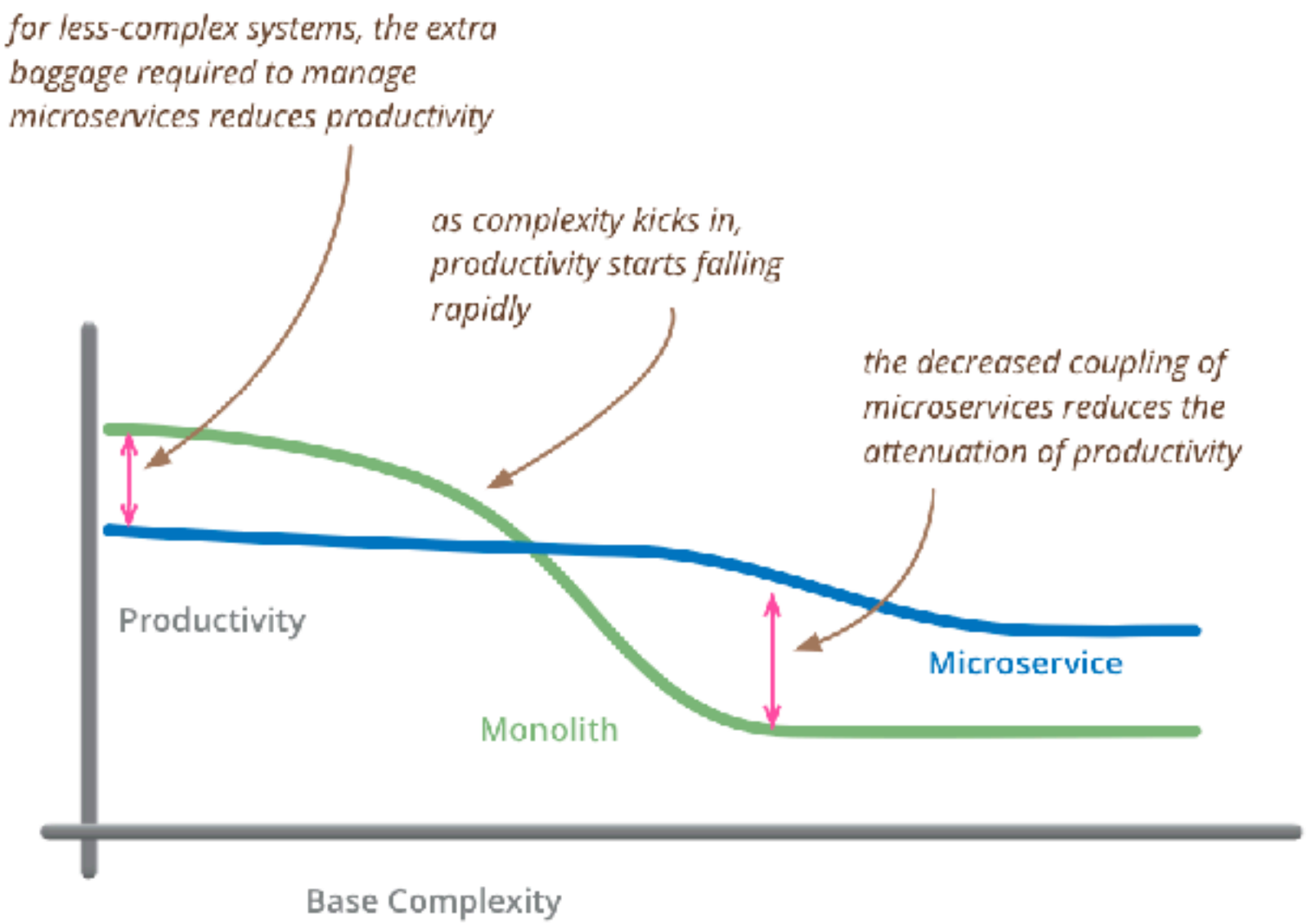
Testing

Operation

https://github.com/Yelp/service-principles

Drawbacks of Microservice

Decide to use when adopt is difficult !!

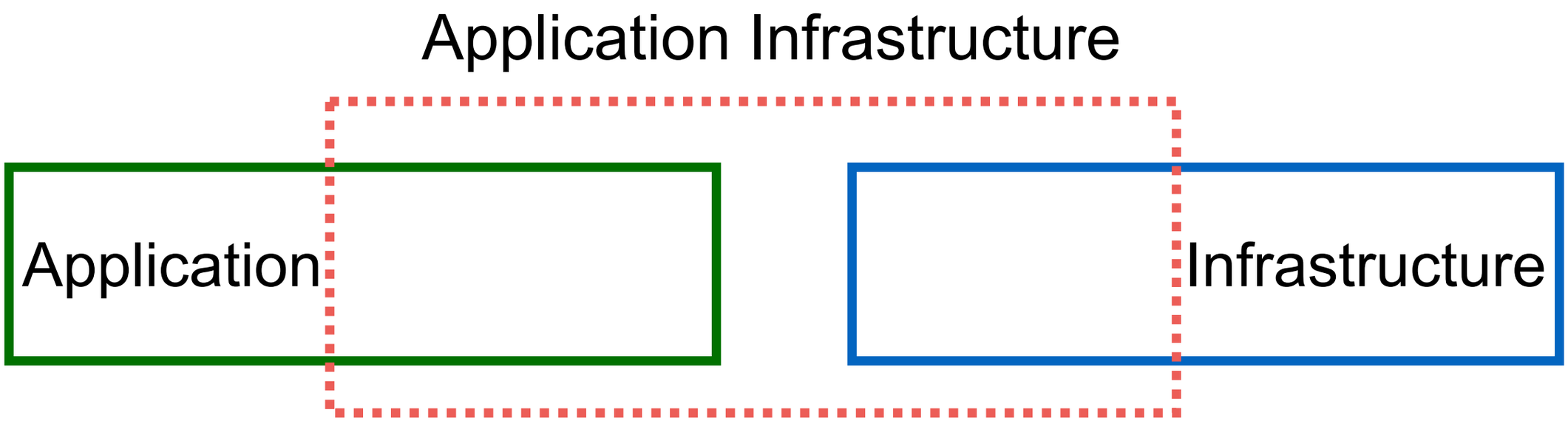


but remember the skill of the team will outweigh any monolith/microservice choice

Microservice architecture patterns

3 Layers of service patterns

Application patterns
Application infrastructure pattern
Infrastructure pattern



https://microservices.io/

Decompose application into small services !!

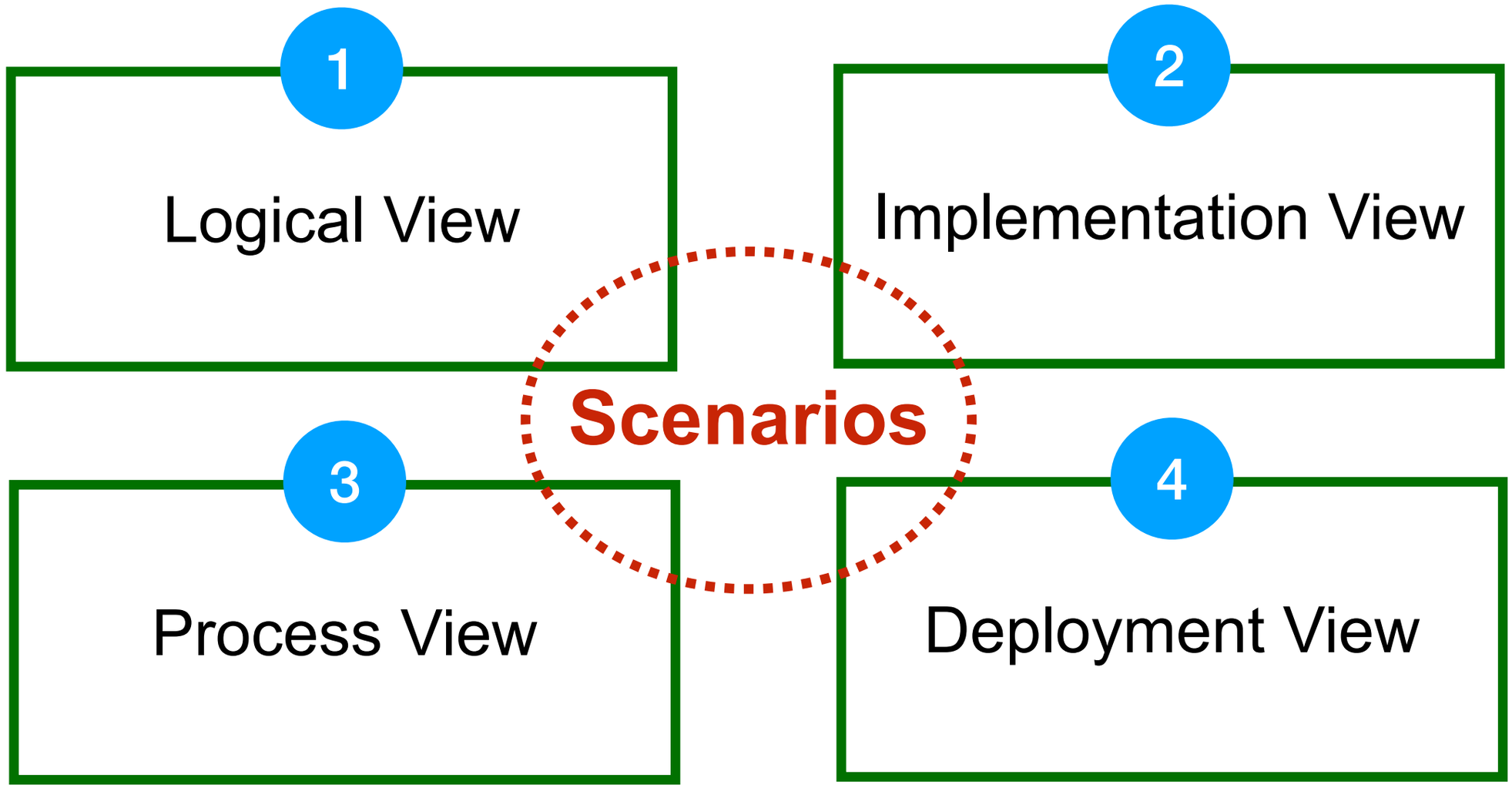
Premature splitting is the root of all **evil**.

Every time you make the decision to split out a new microservice, there's a **risk** of ending up with a bloated app.

4 View model of Software Architecture

1. Logical View
2. Implementation View
3. Process View
4. Deployment View

4 View model of Software Architecture



Decompose application into services

By business capability ?
By subdomain/technical ?
By team ?

Step to define services

1. Identify system operations
2. Identify services
3. Define service APIs and collaboration

1. Identify system operations

Start with functional requirements

User story

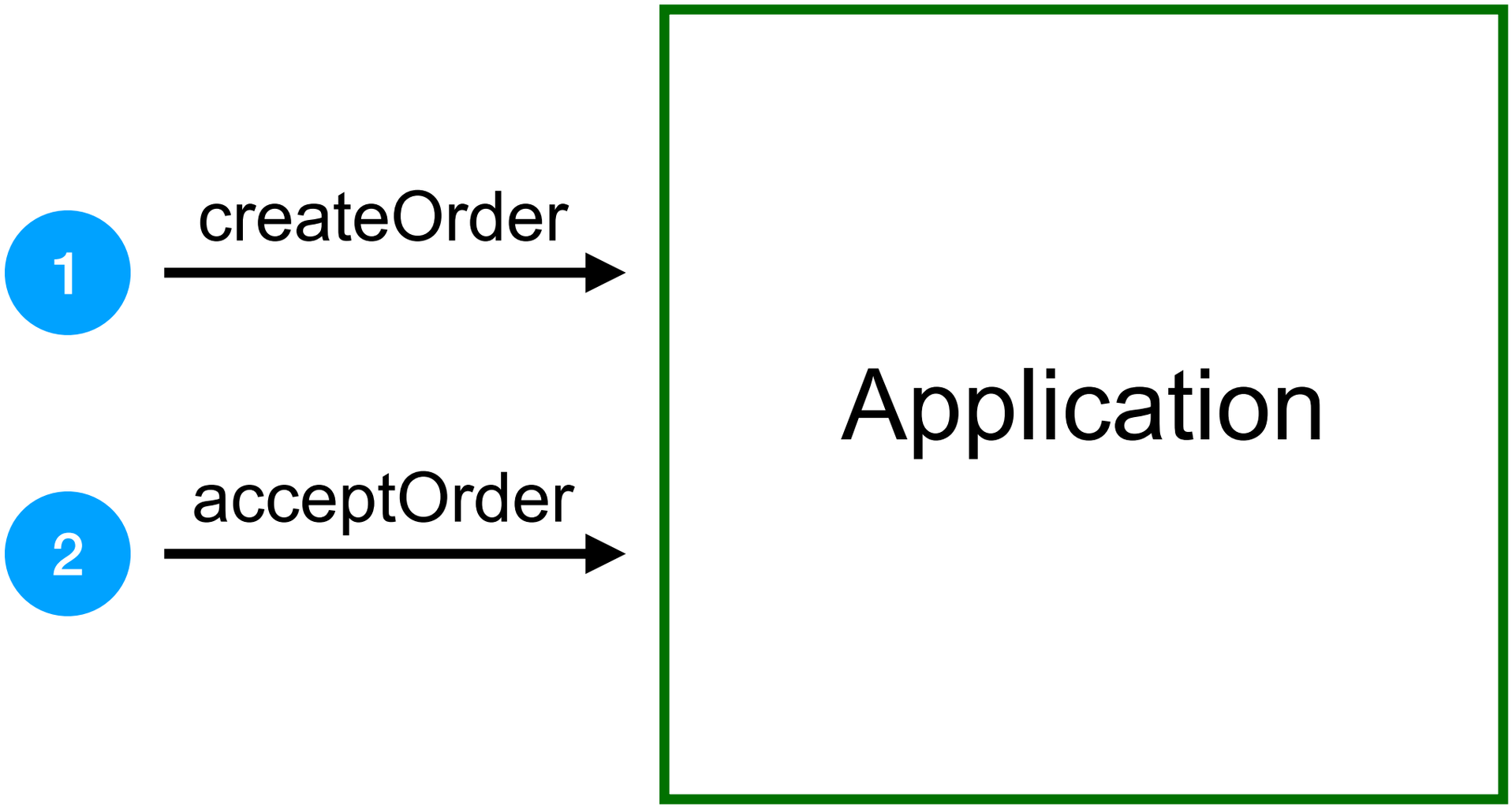
1

ลูกค้าต้องการสั่งซื้ออาหาร

2

ร้านอาหารรับคำสั่งซื้อ

1. Identify system operations



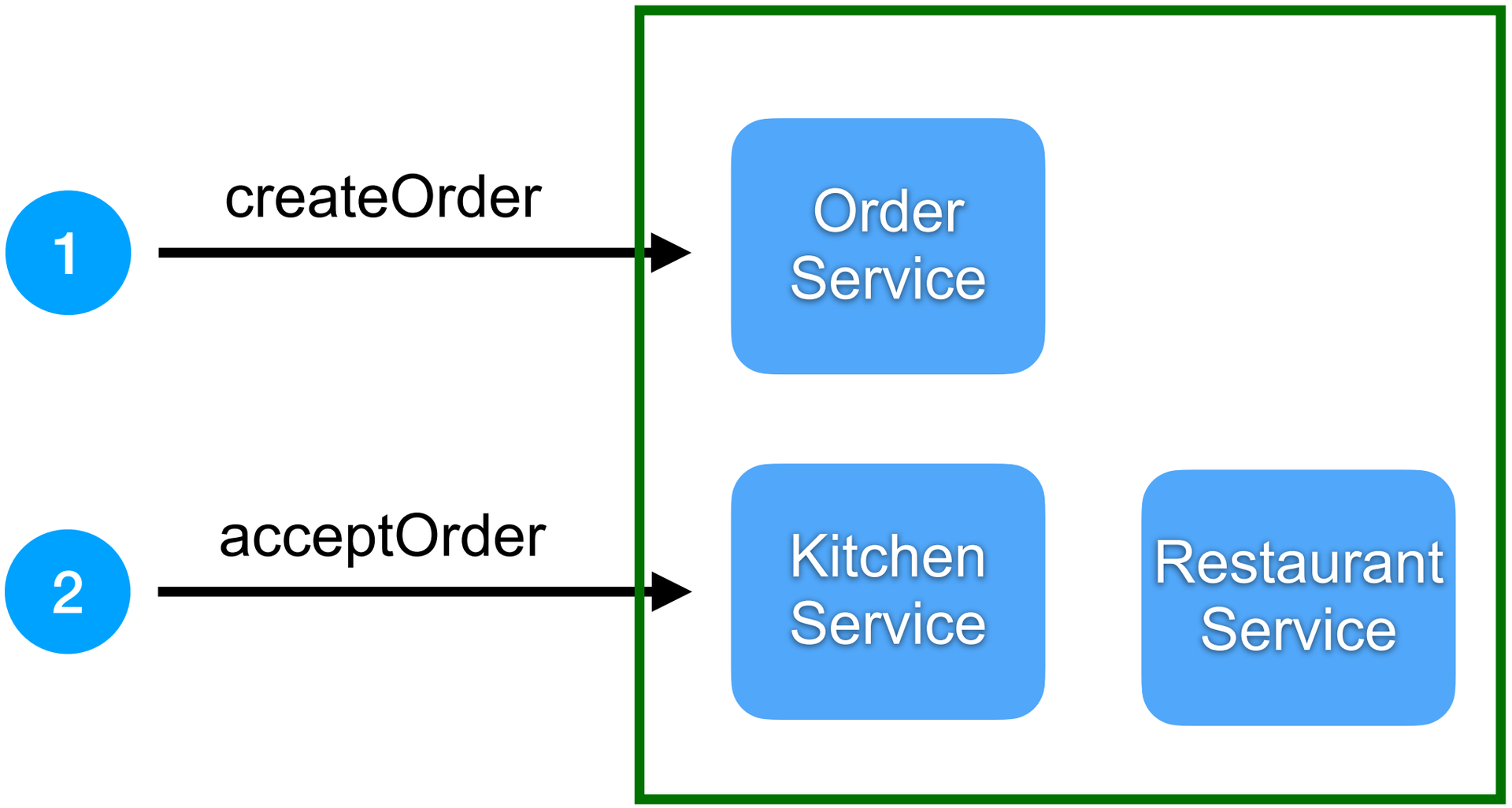
2. Identify services

Try to decomposition into services

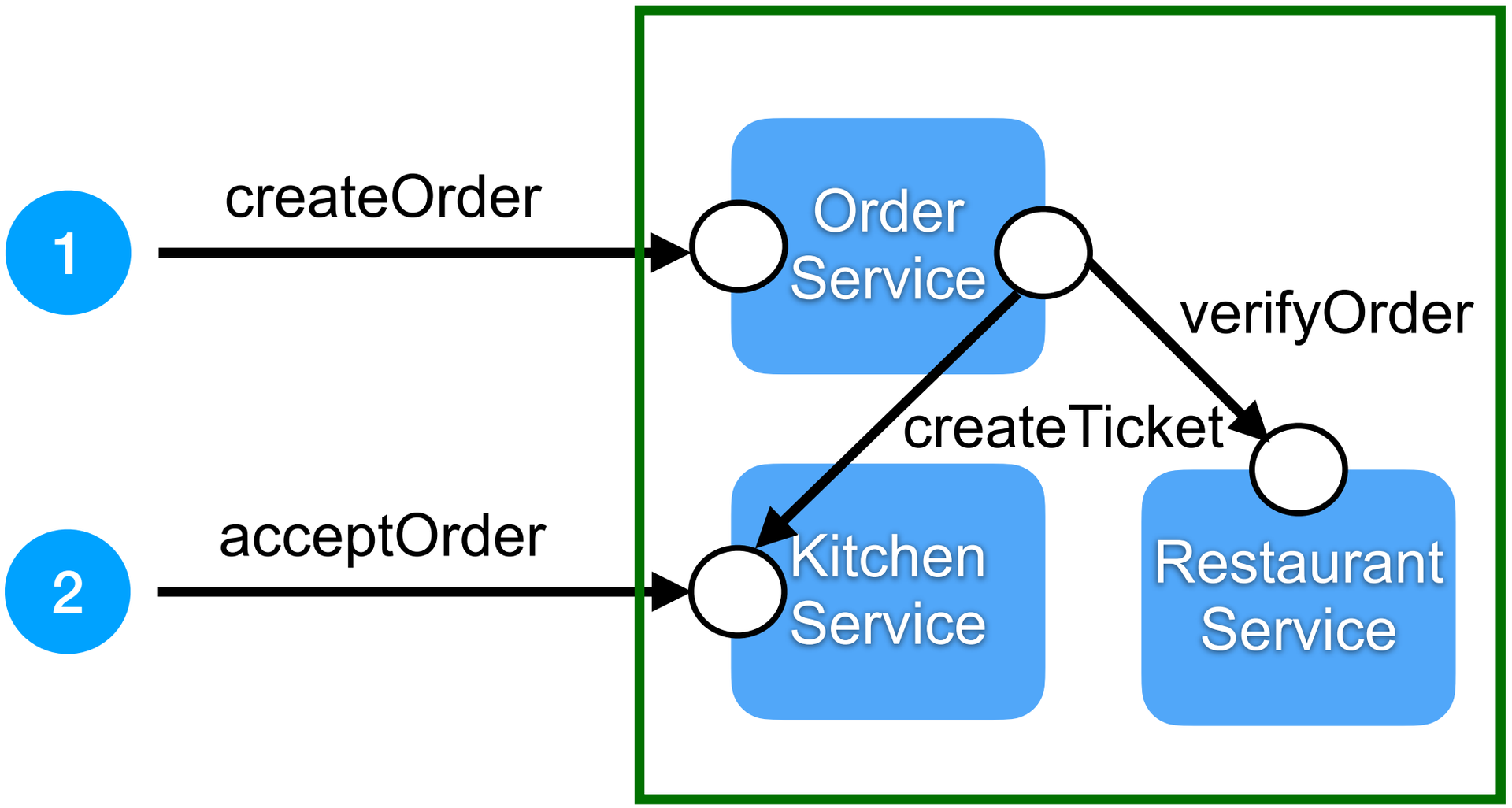
Business > Technical concept

What > How

2. Identify services



3. Define service APIs and collaborations

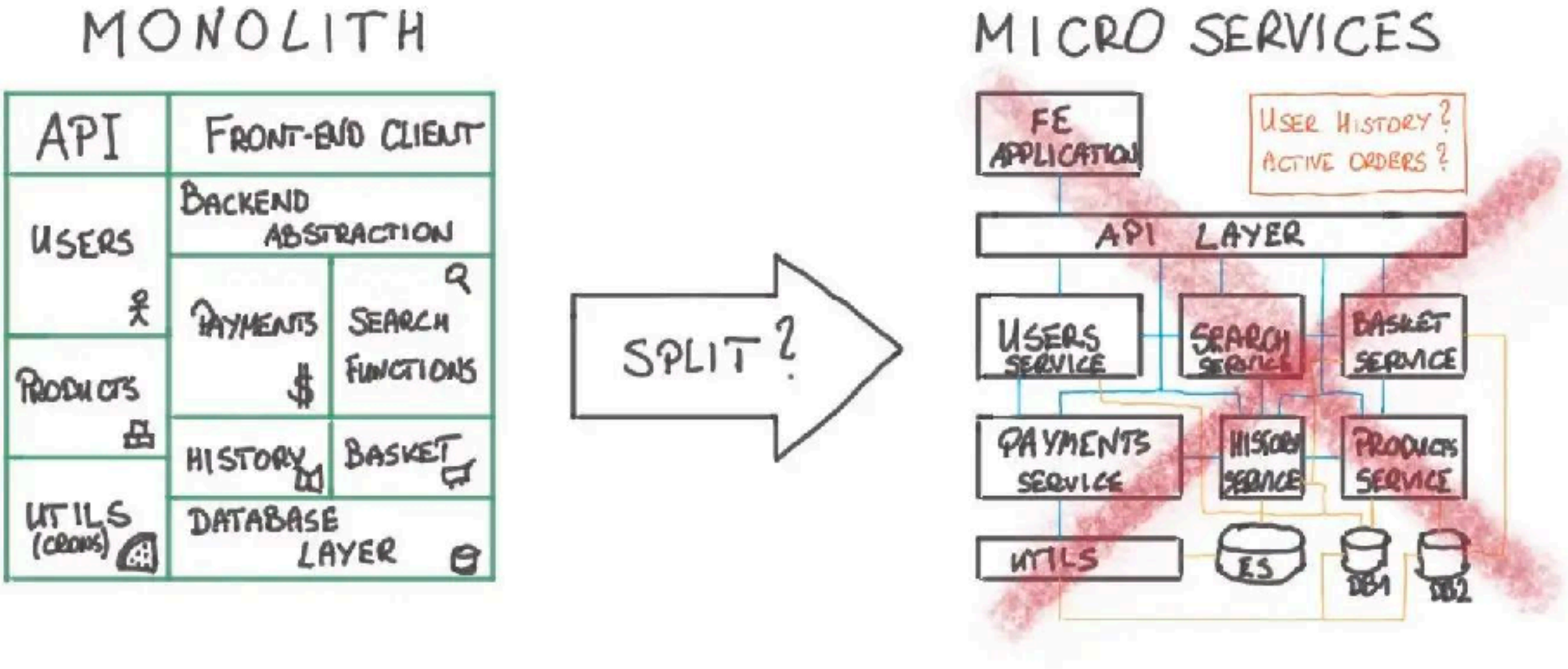


Problems when decompose services ?

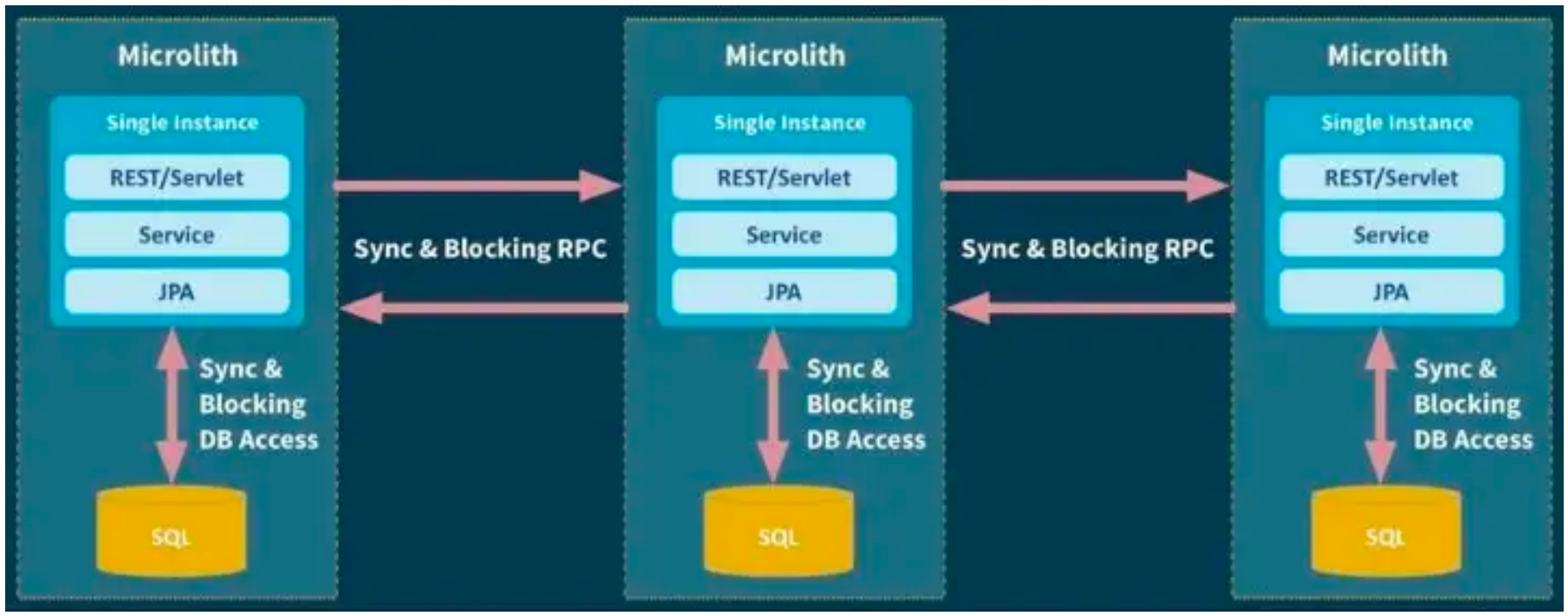
Network latency
Reliability of communication
Maintain data consistency

God services !!
Microlith !!

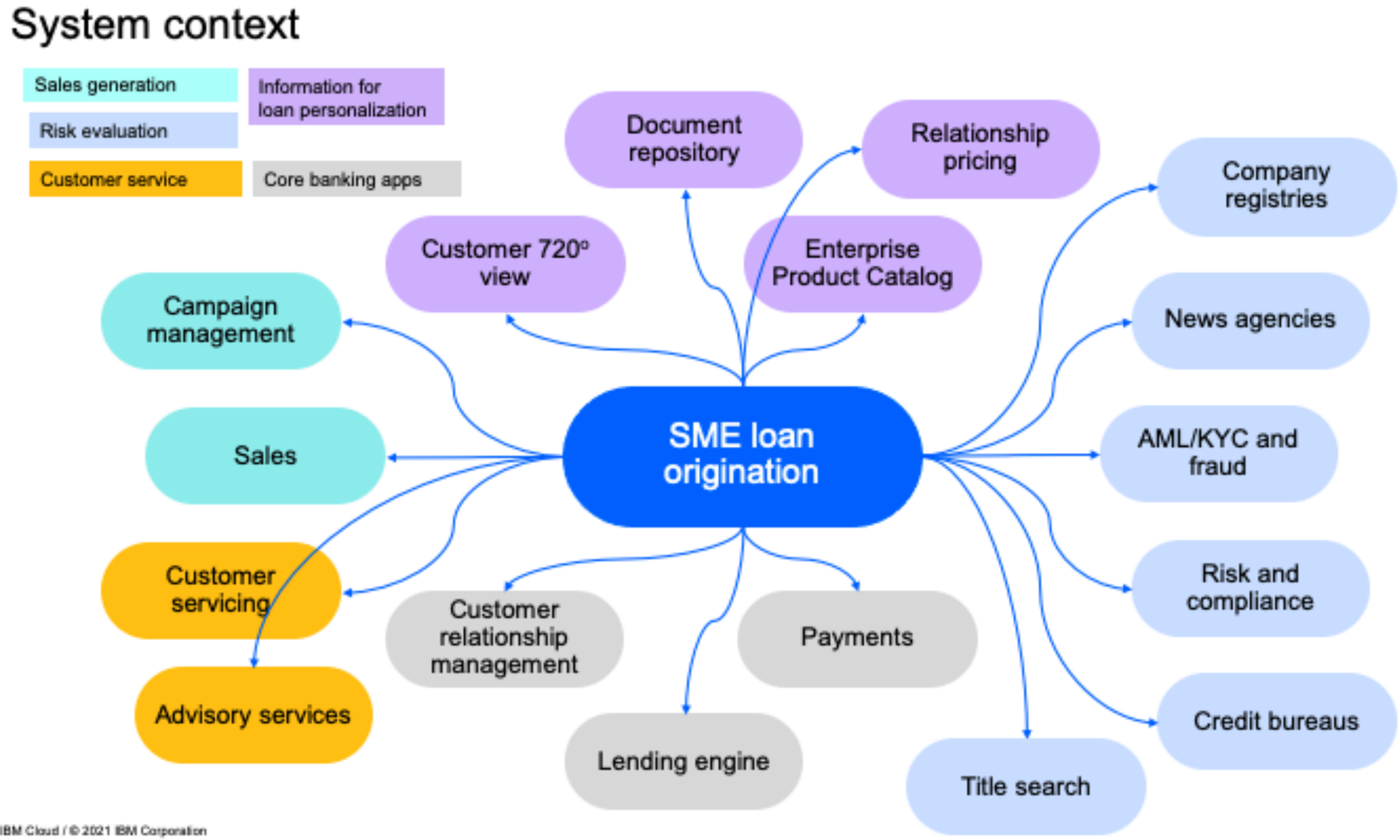
Microservices !!



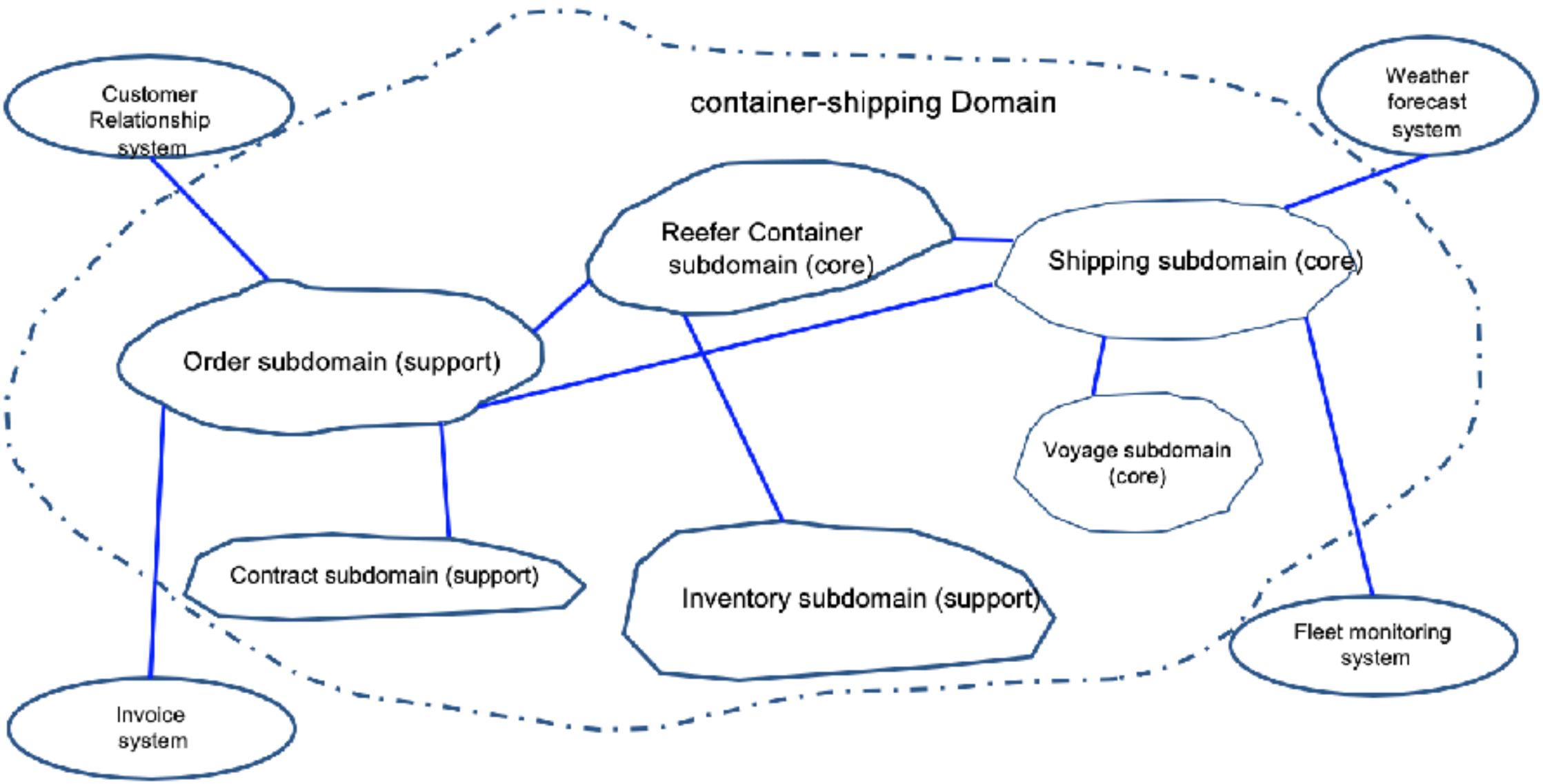
Microlith !!



Example of System Context

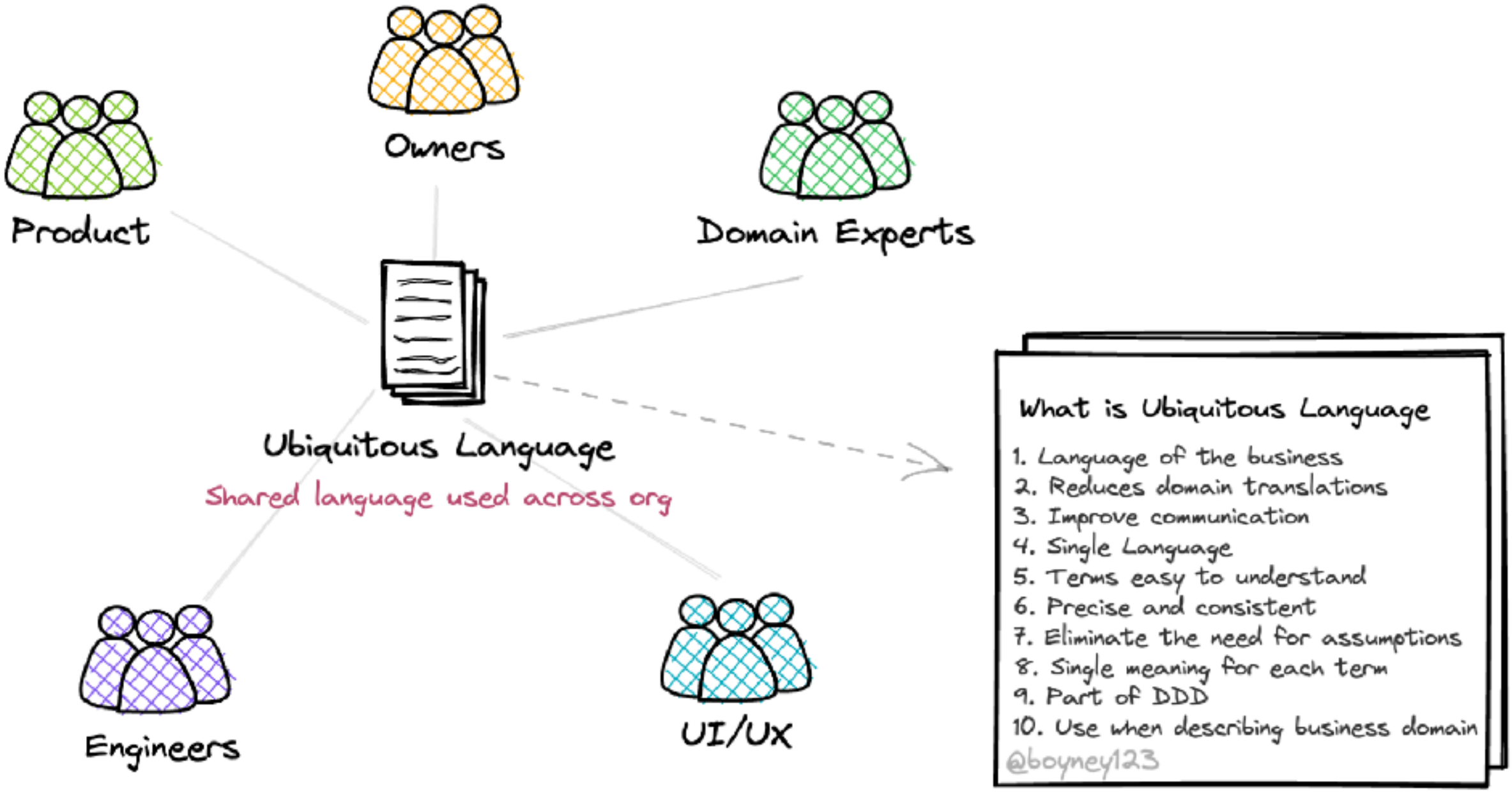


Assess domains and sub-domains



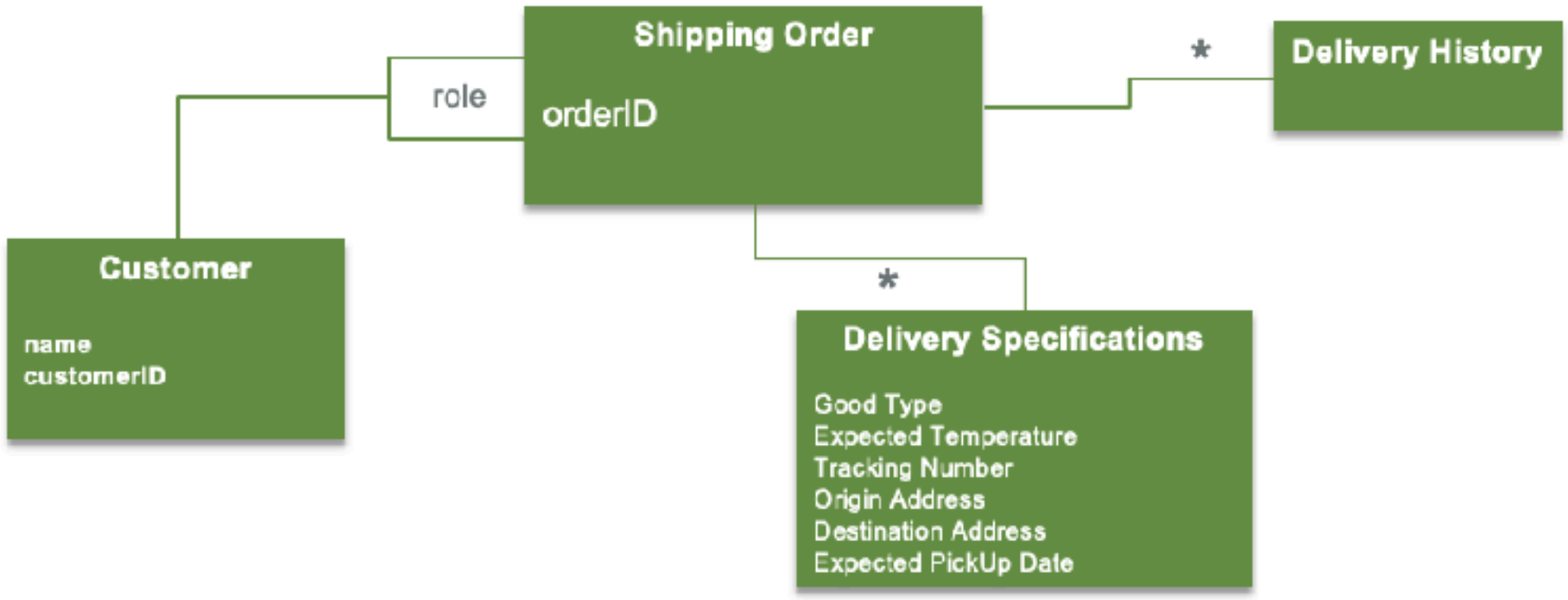
Define the ubiquitous language

Define the ubiquitous language

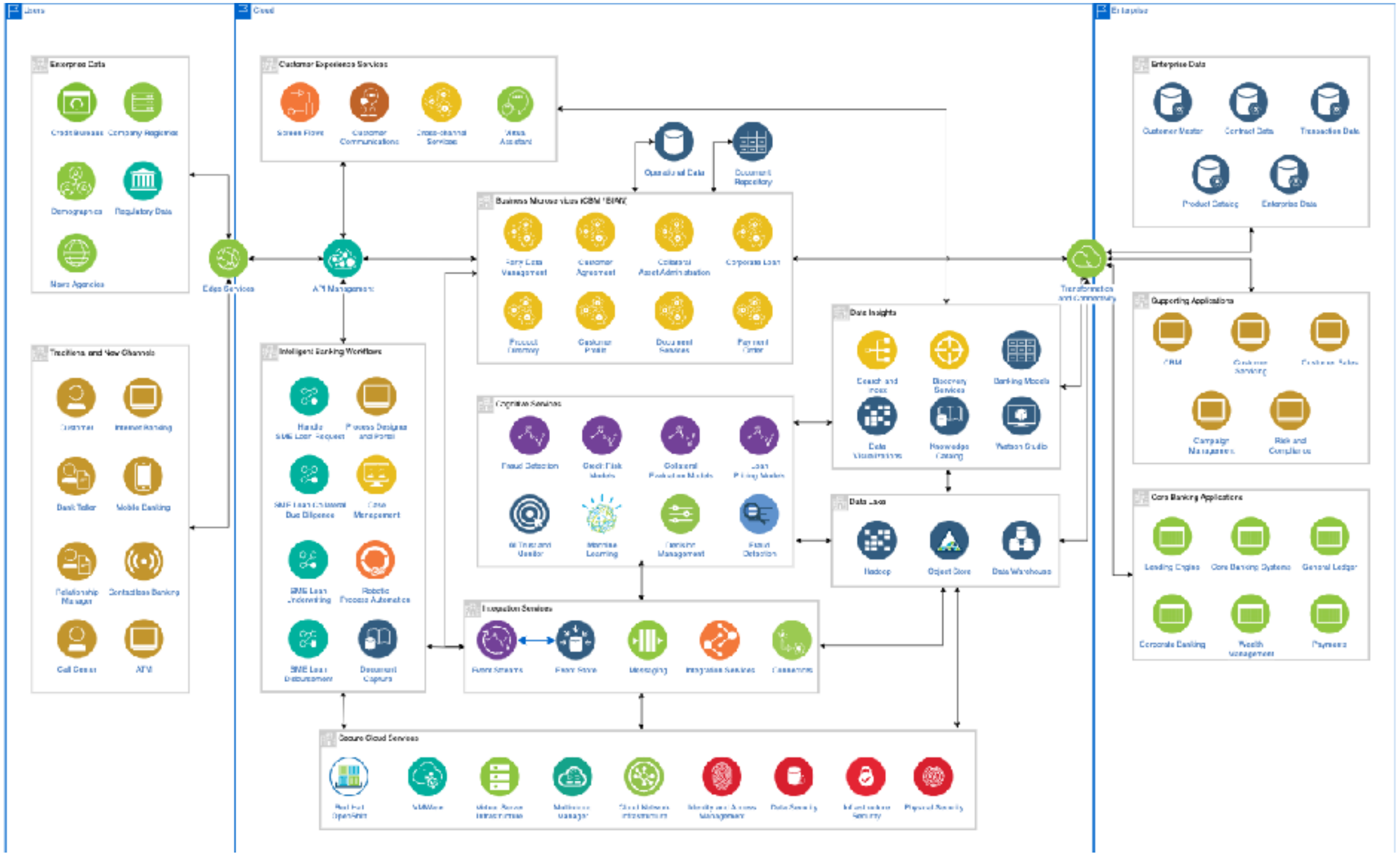


https://serverlessland.com/event-driven-architecture/visuals/ubiquitous-language

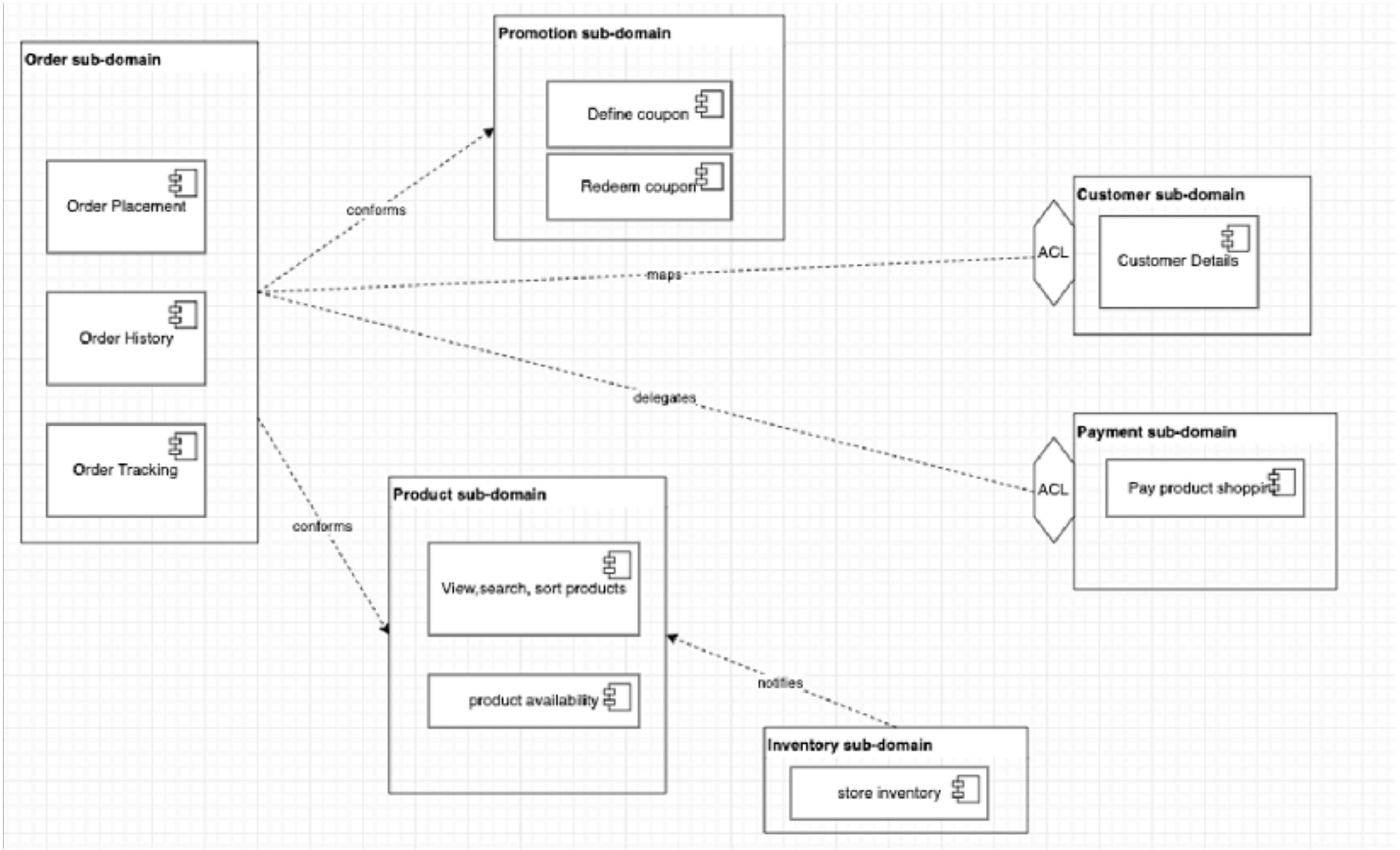
Structure of Data



Grouping services



Communication types



Communication between services

Interaction Styles

	One-to-one	**One-to-many**
Synchronous	Request/response	-
Asynchronous	Async request/response	Publish/subscribe
	Notification	Publish/async response

Synchronous Communication

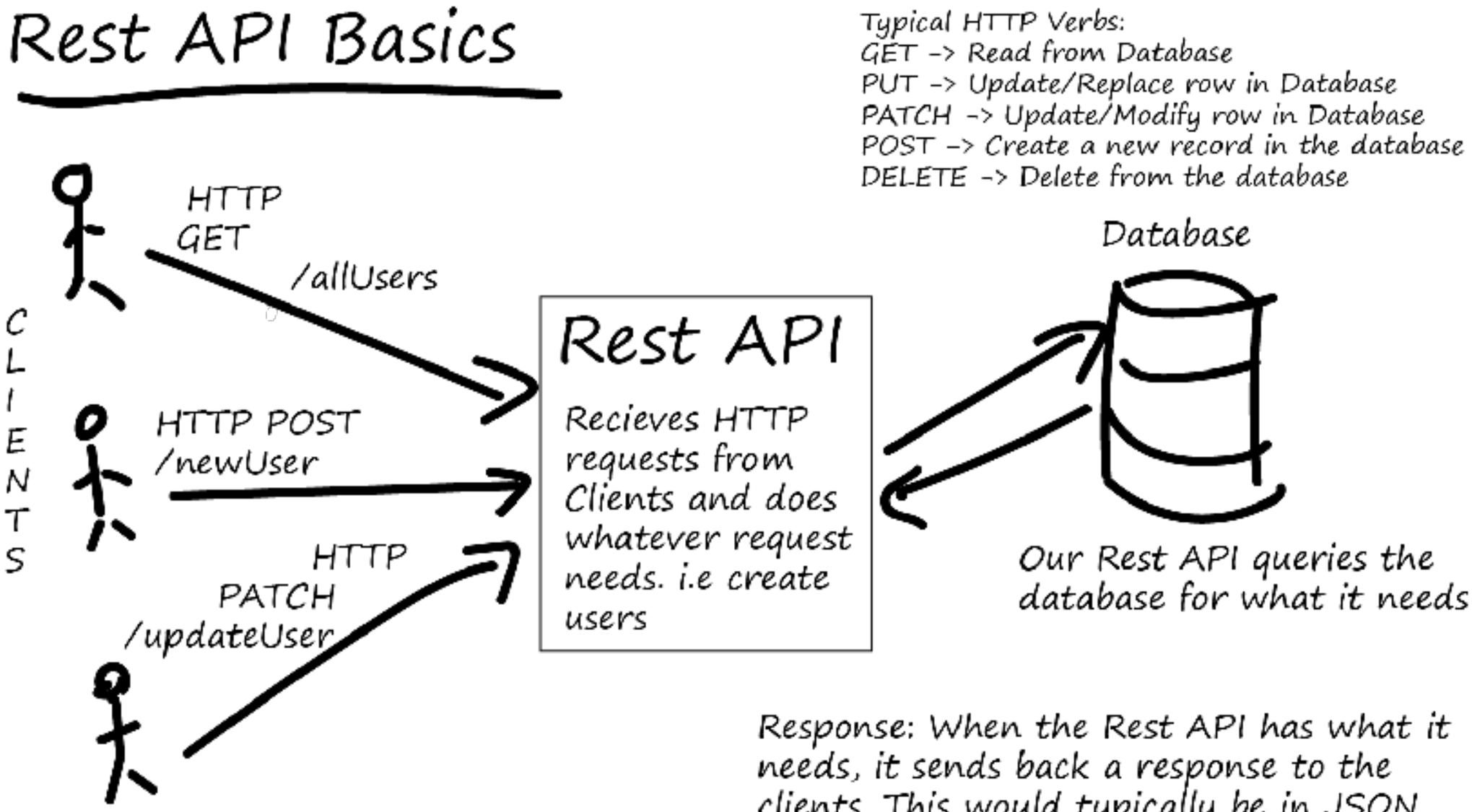
REST

gRPC

Handling failure with **Circuit breaker**

Service discovery

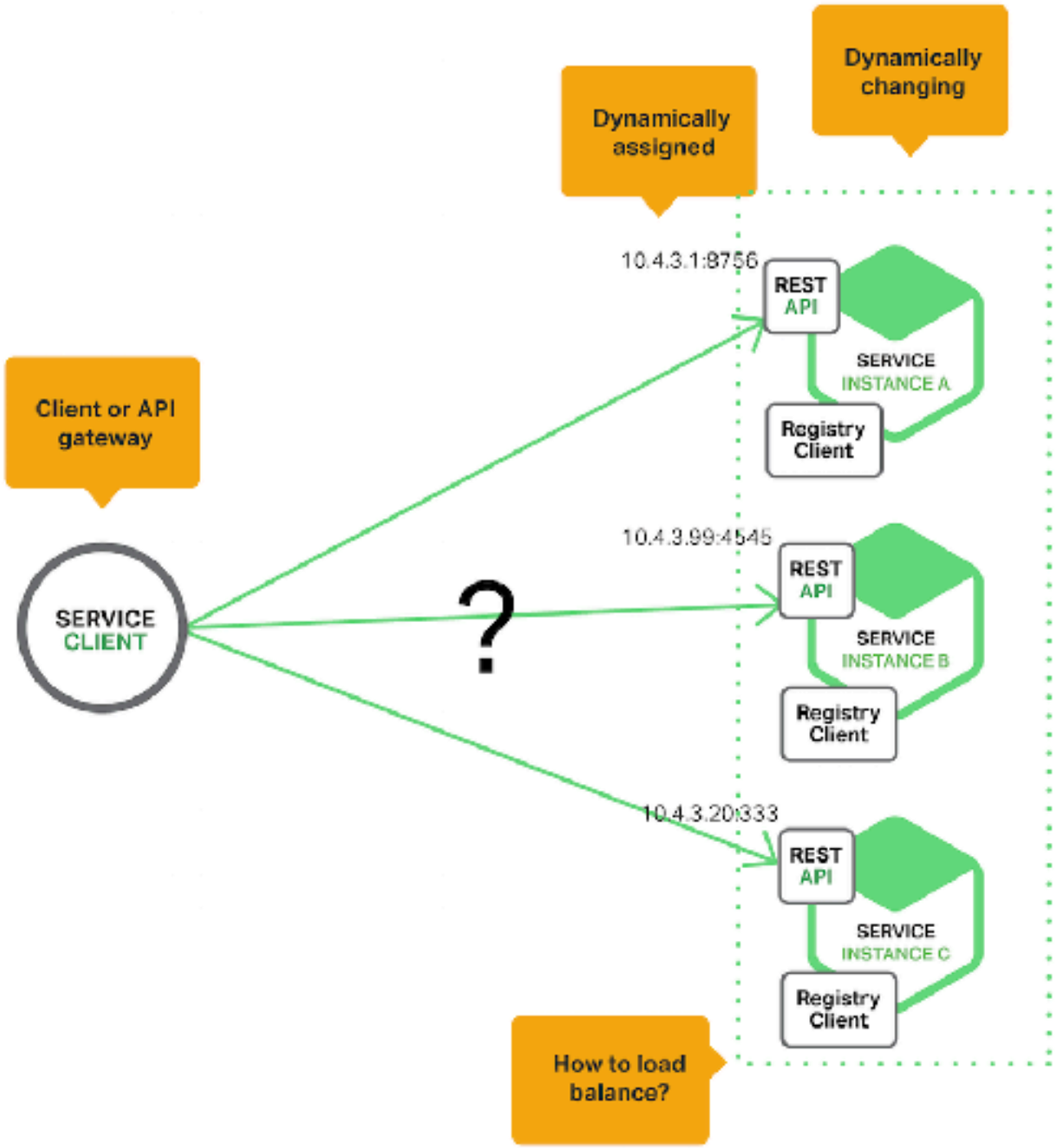
REpresentational **S**tate **T**ransfer



REST :: mapping with HTTP verbs

Activity	HTTP Method
Retrieve data	GET
Create new data	POST
Update data	PUT
Delete data	DELETE

Service Discovery ?

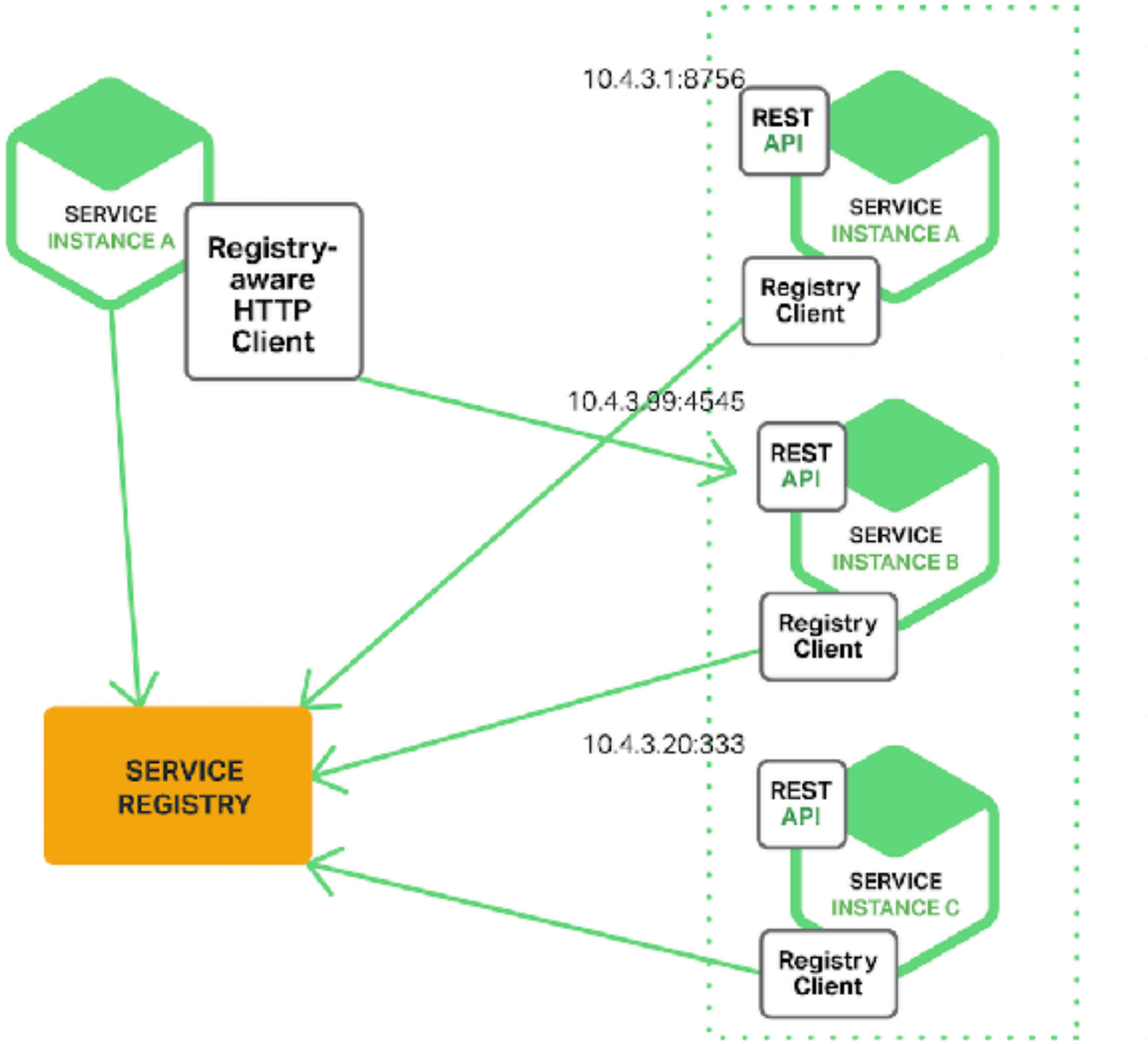


https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/

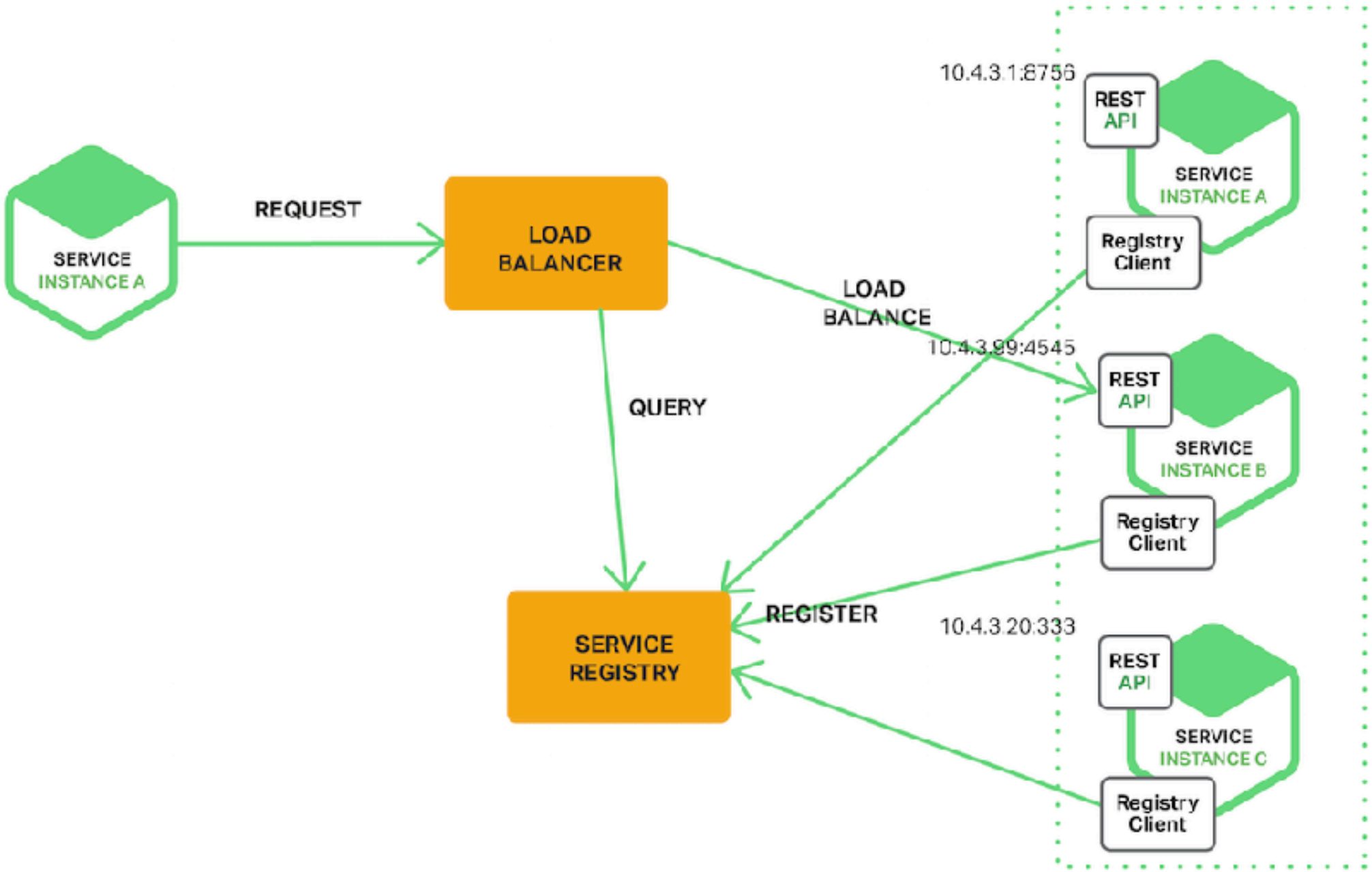
Service Discovery ?

Client-side
Server-side

Client-side Discovery

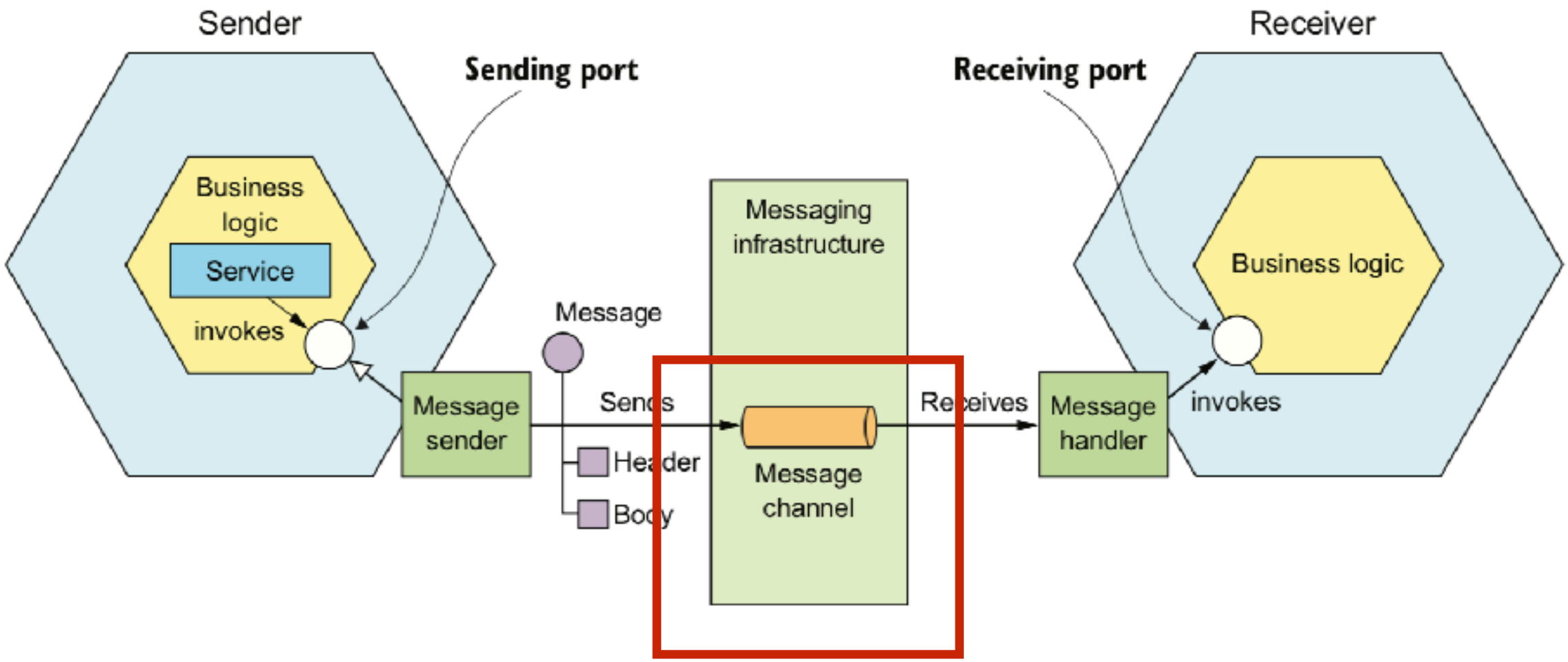


Server-side Discovery



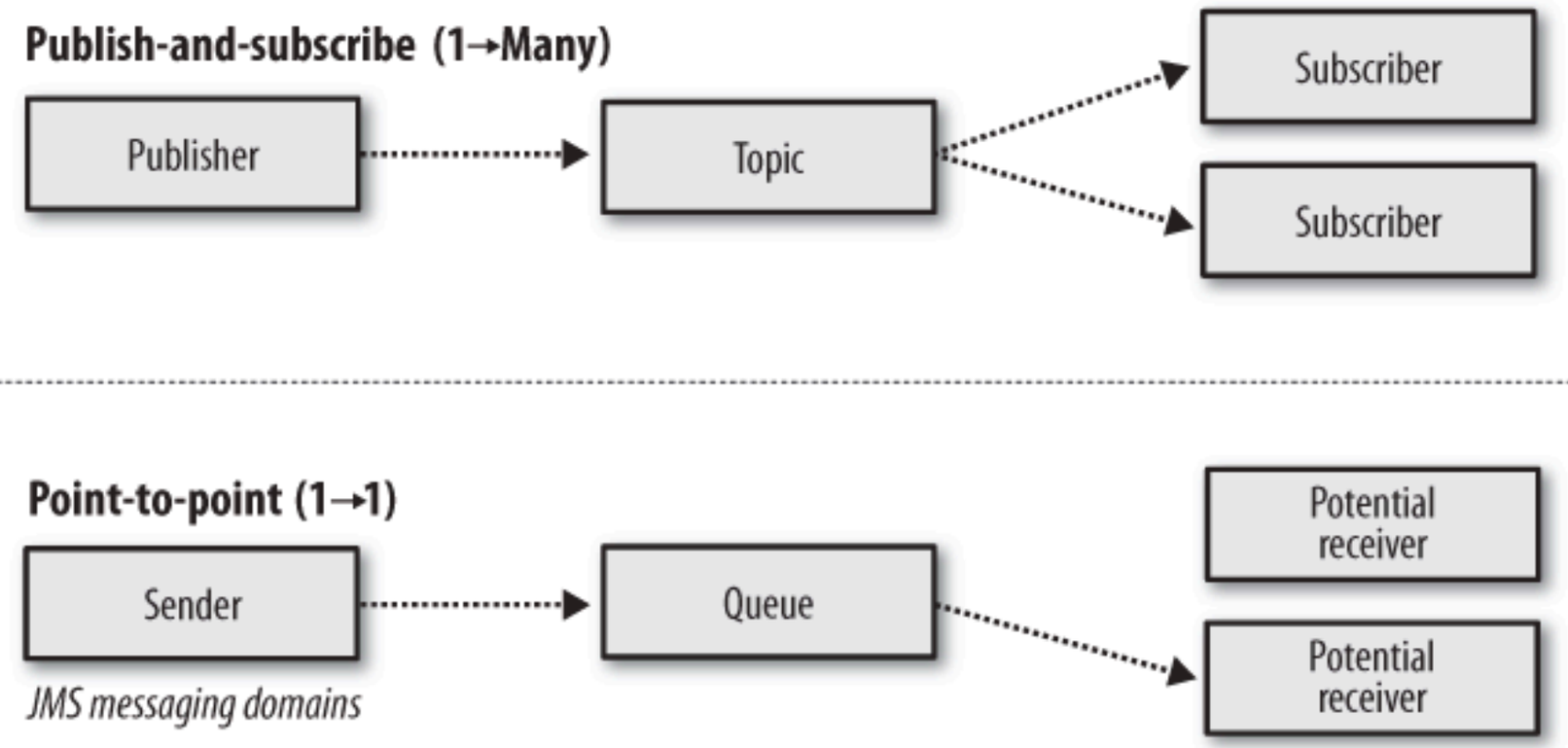
Asynchronous communication

Asynchronous messaging pattern

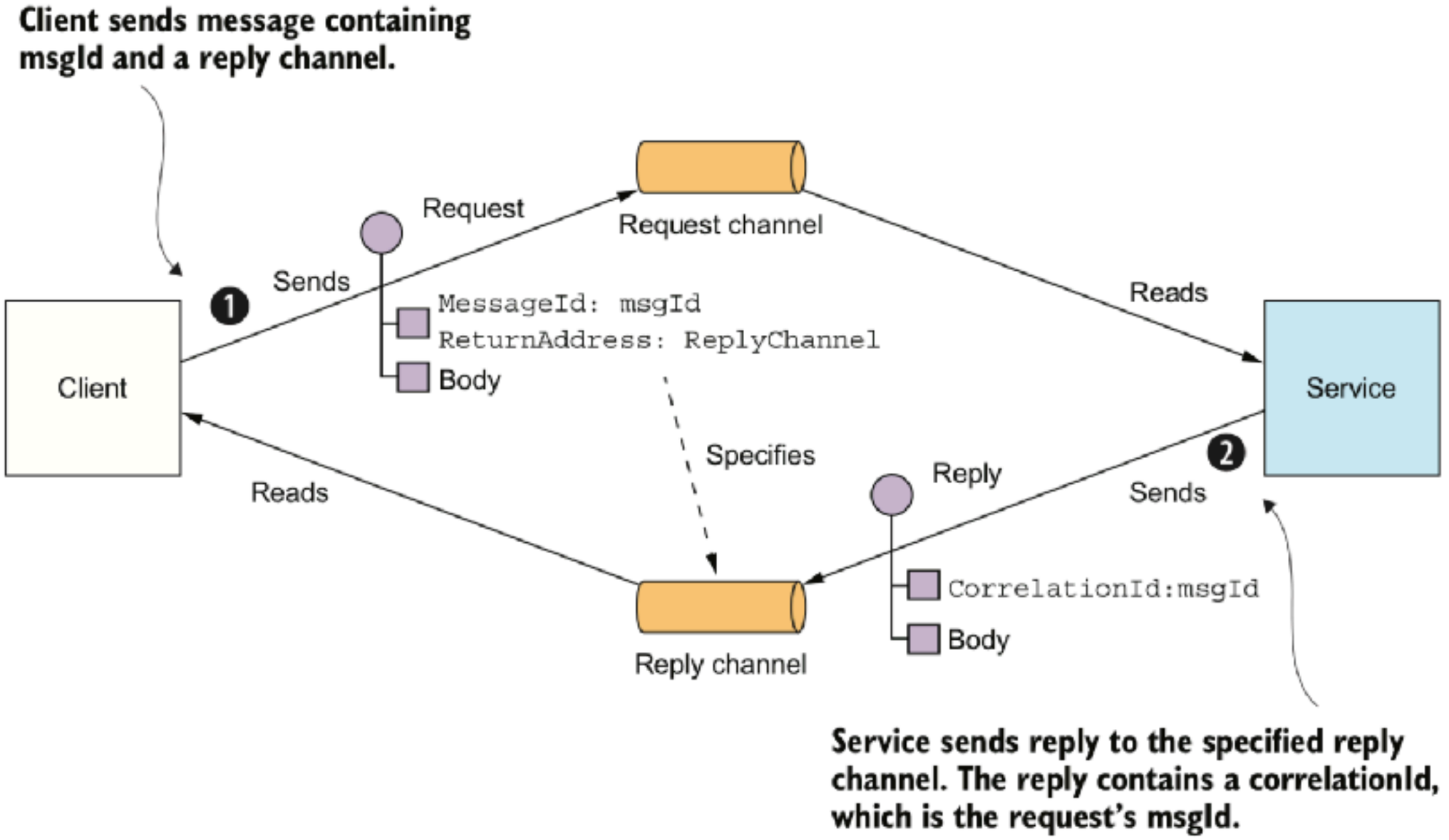


Messaging channels

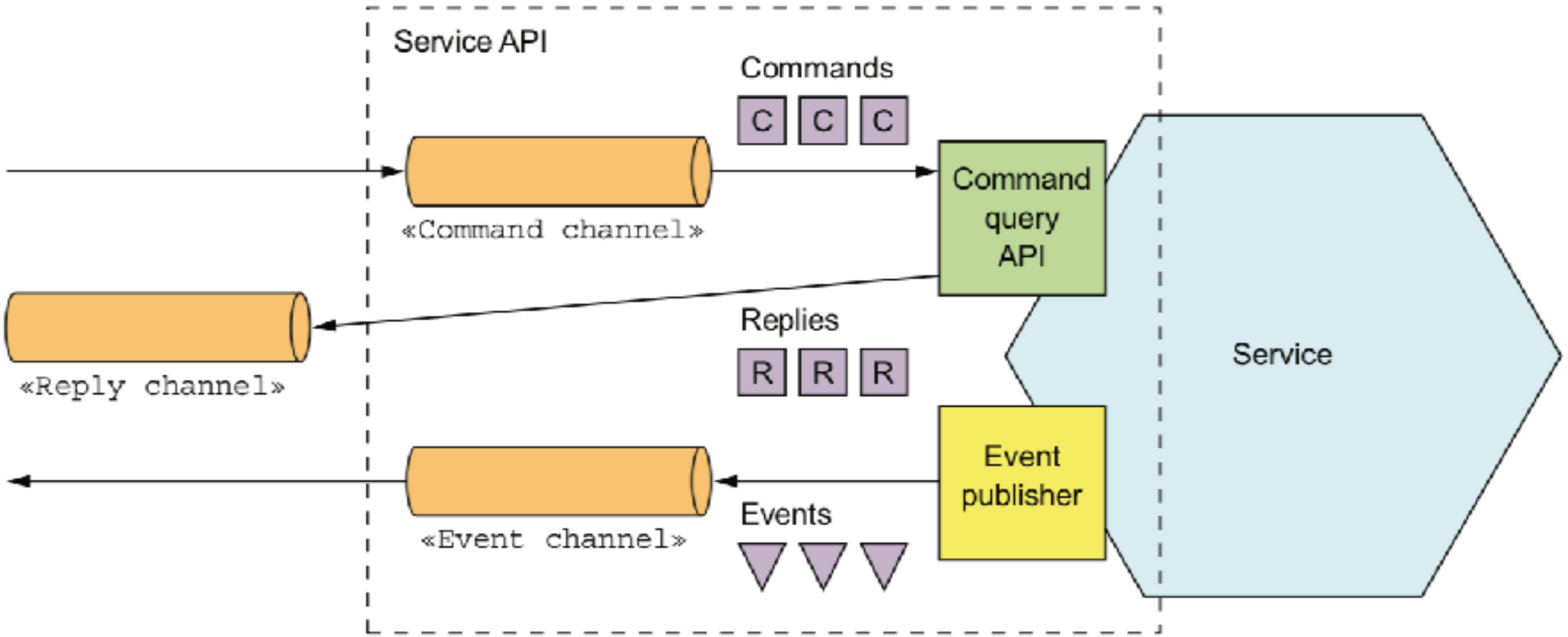
Publish-subscribe
Point-to-point



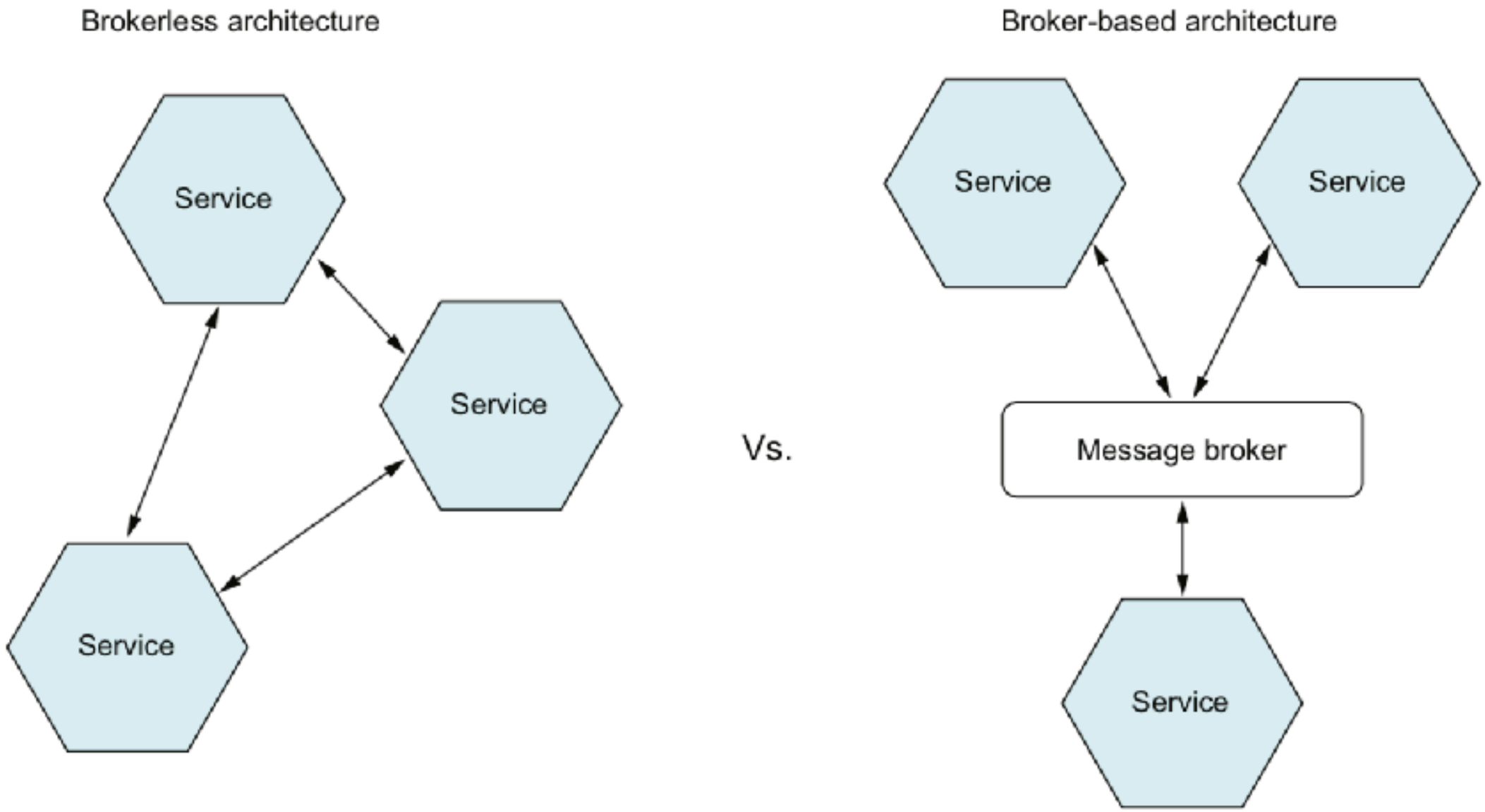
Async request/response (1)



Async request/response (2)



Messaging-based use message broker



Message brokers

Benefits

Loose-coupling
Message buffer
Flexible communication

Drawbacks

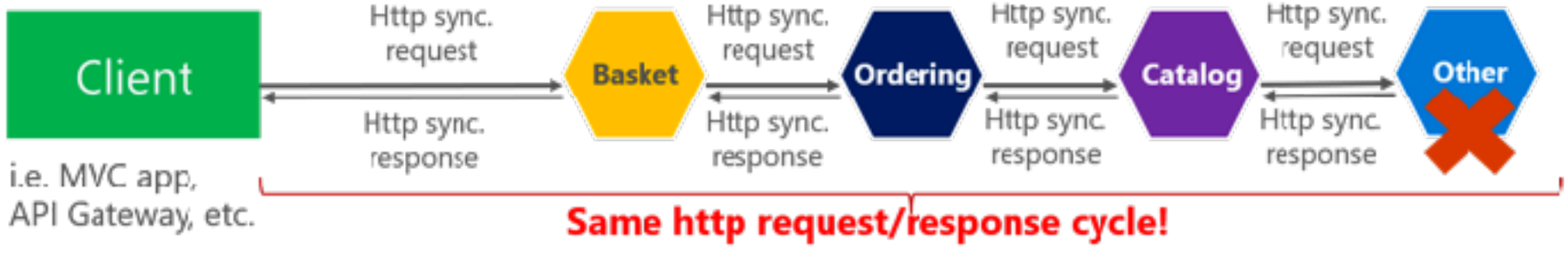
Performance bottleneck
Single point of failure
Operational complexity

Communication !!

Synchronous vs. async communication across microservices

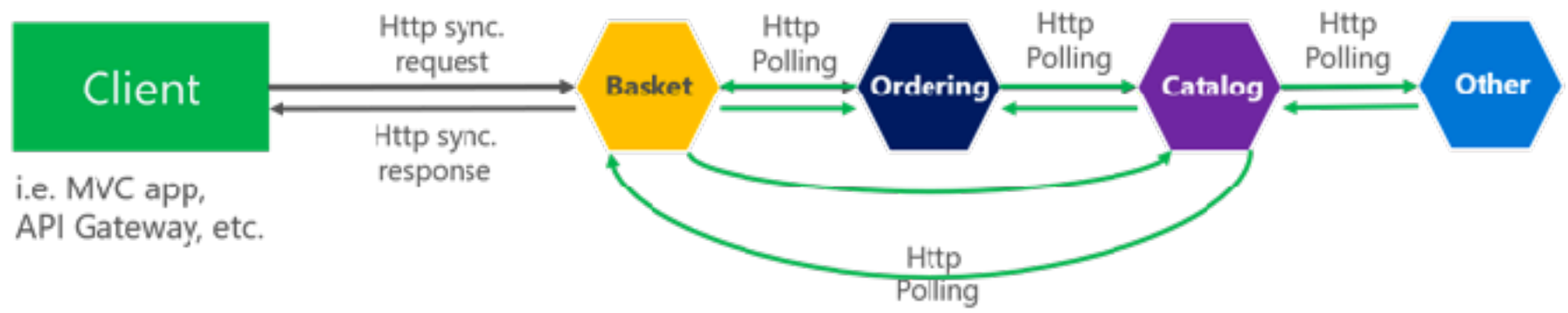
Anti-pattern

Synchronous
all req./resp. cycle

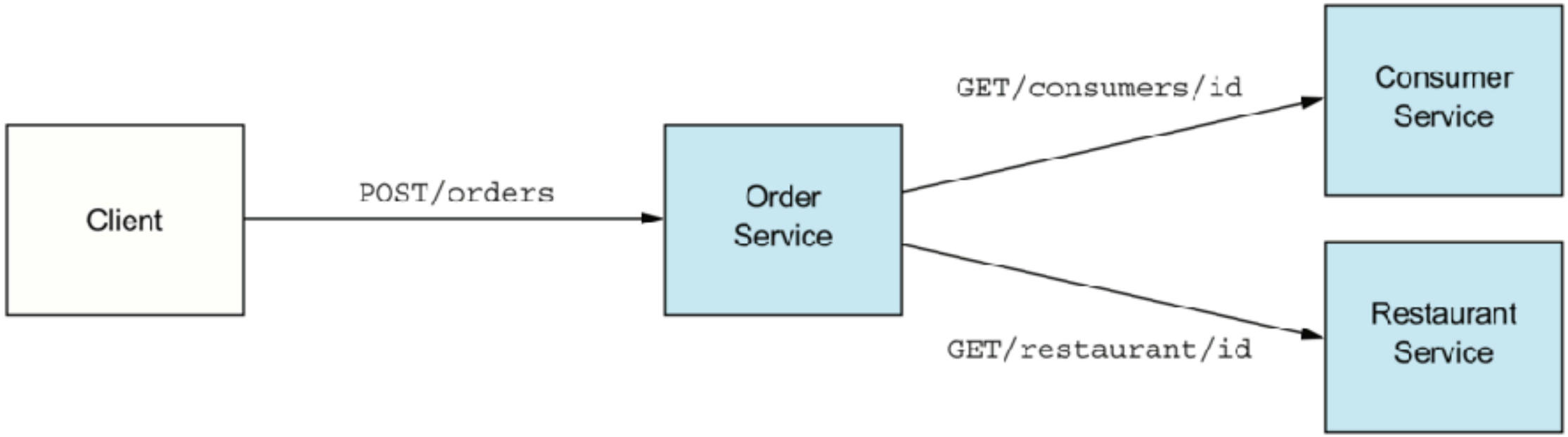


Asynchronous
Comm. across
internal microservices
(EventBus: i.e. **AMPQ**)

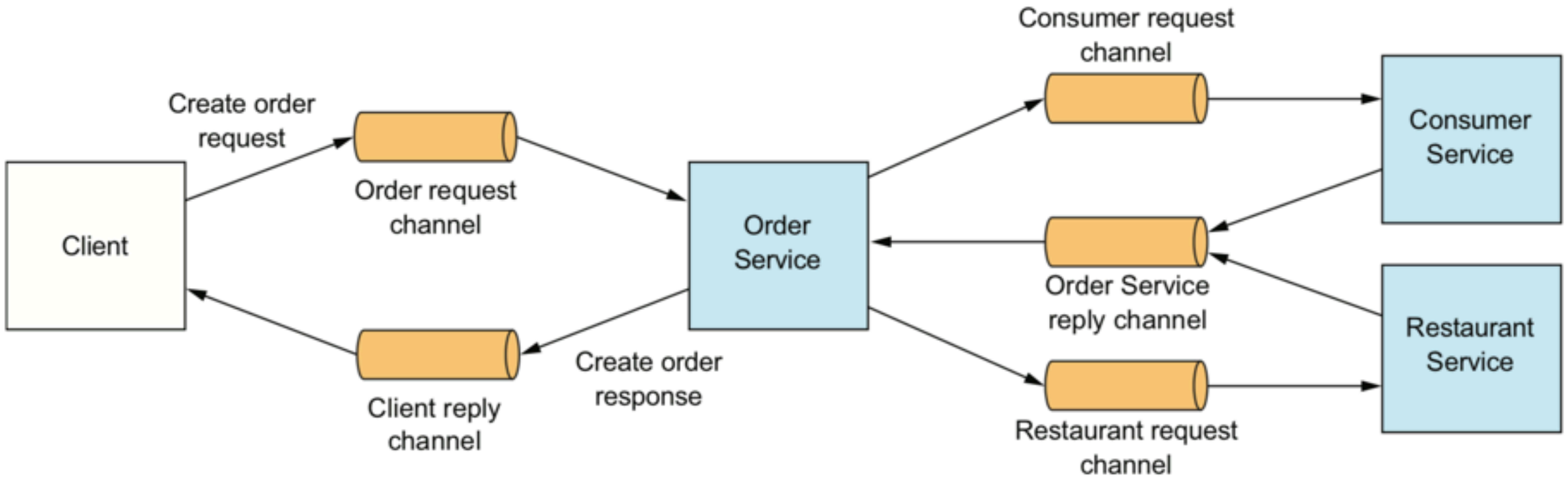
"Asynchronous"
Comm. across
internal microservices
(Polling: **Http**)



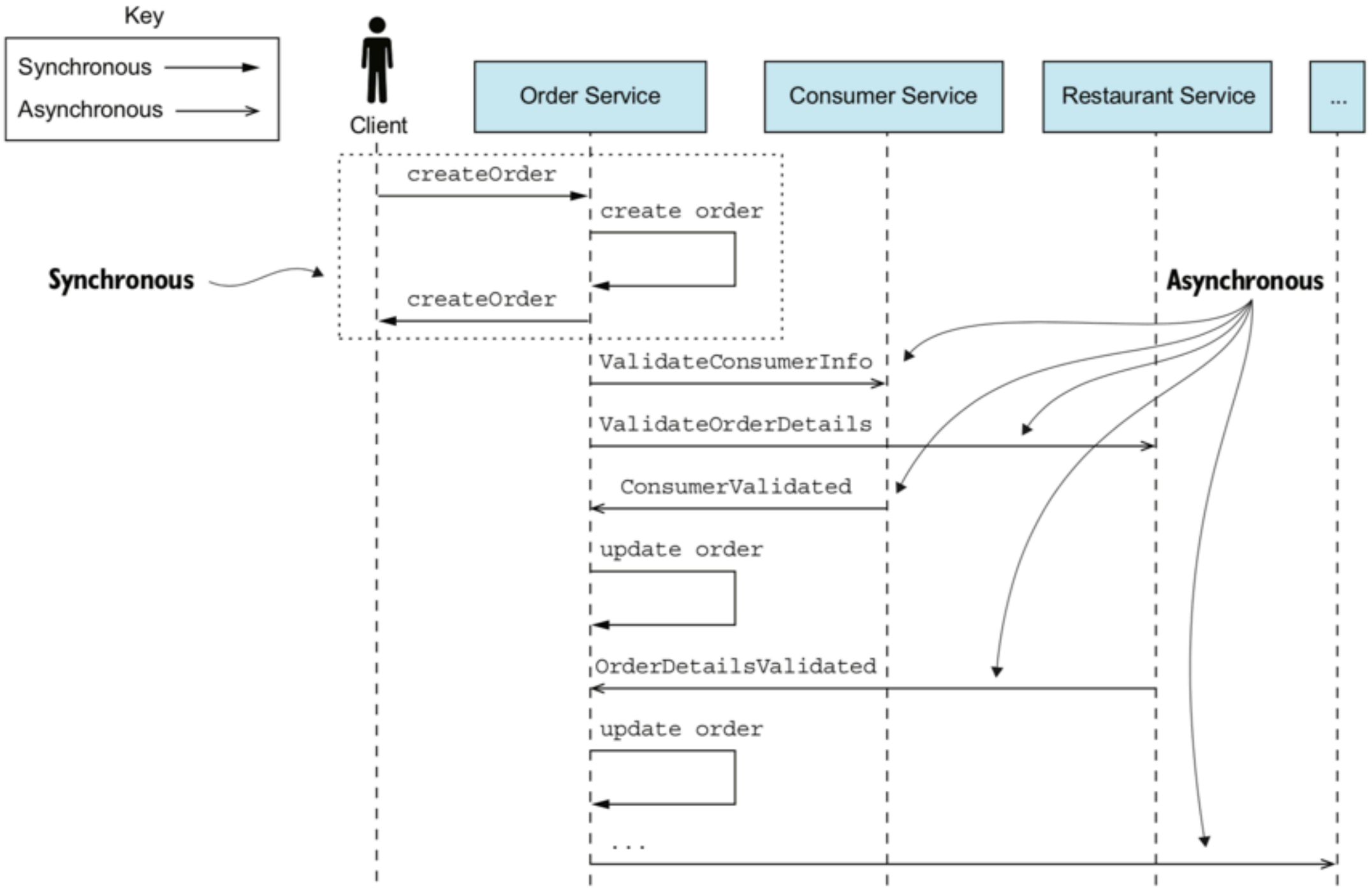
Synchronous reduce availability



Replace with asynchronous (1)

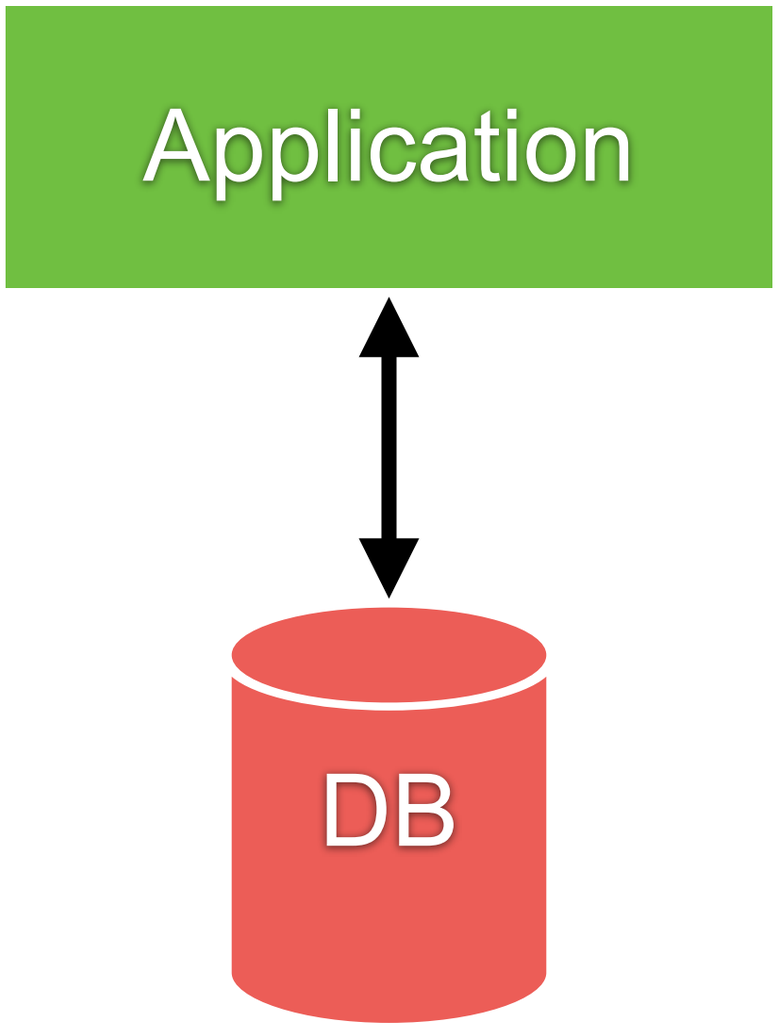


Replace with asynchronous (2)



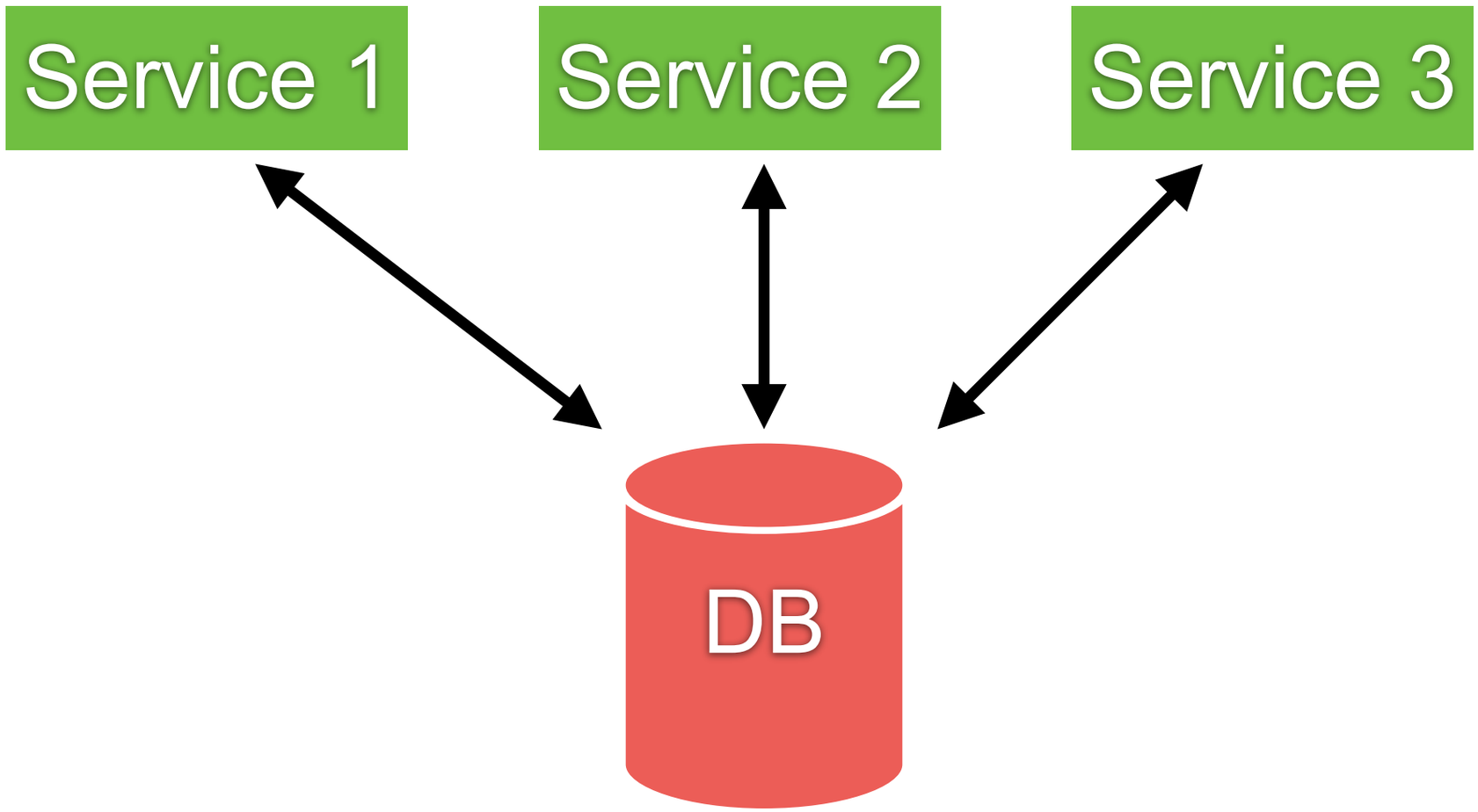
Manage Data Consistency

Monolithic



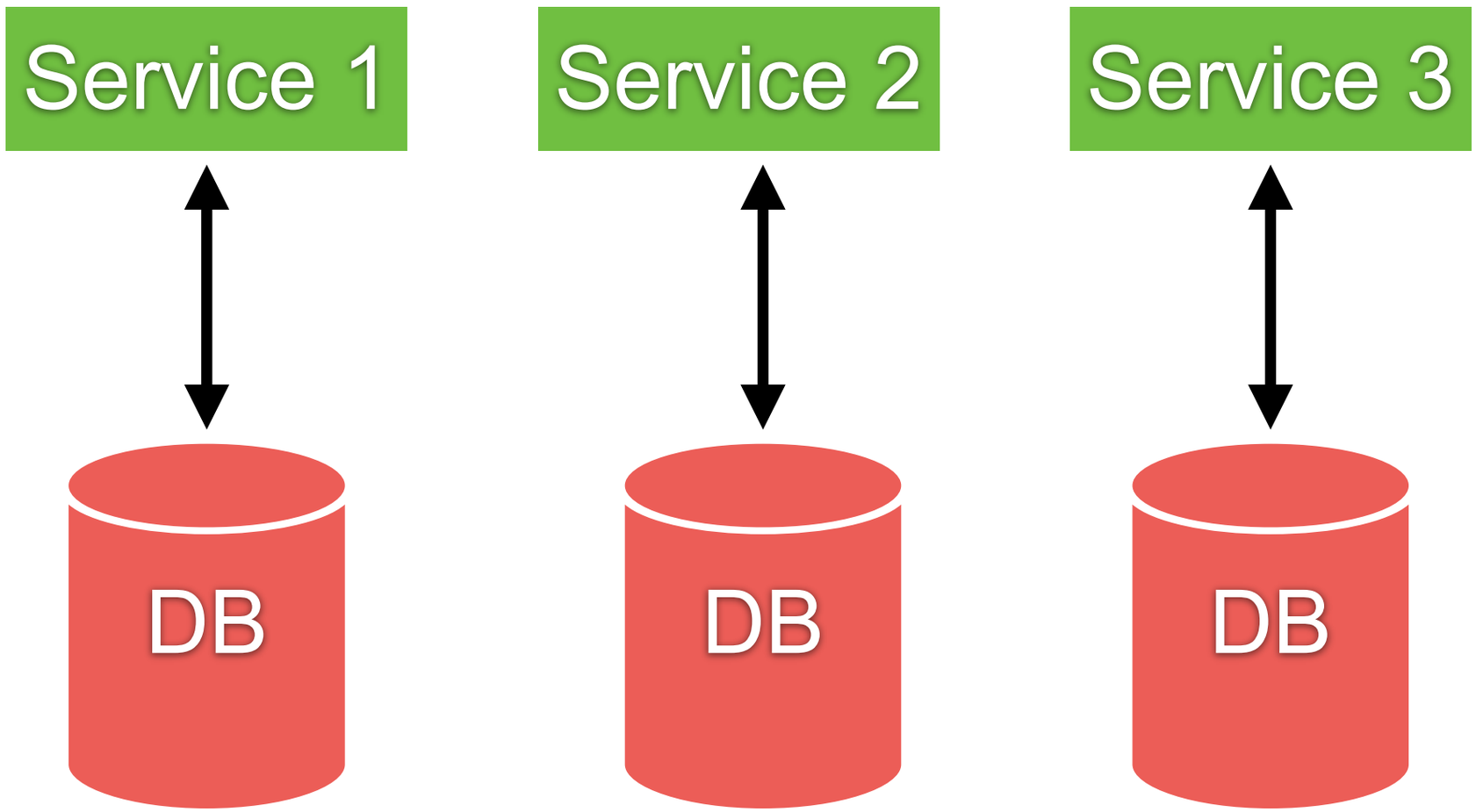
Microservices

Shared database

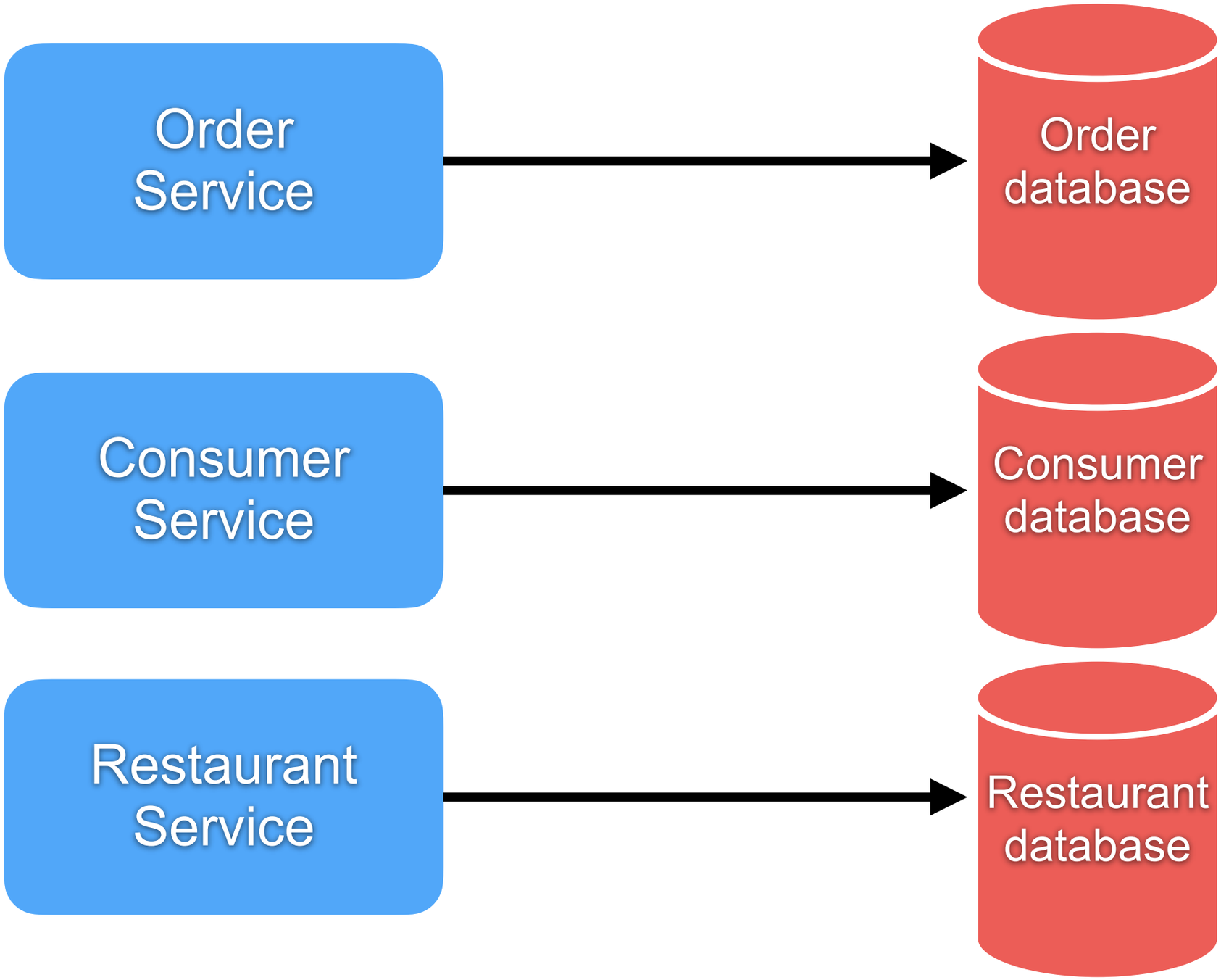


Microservices

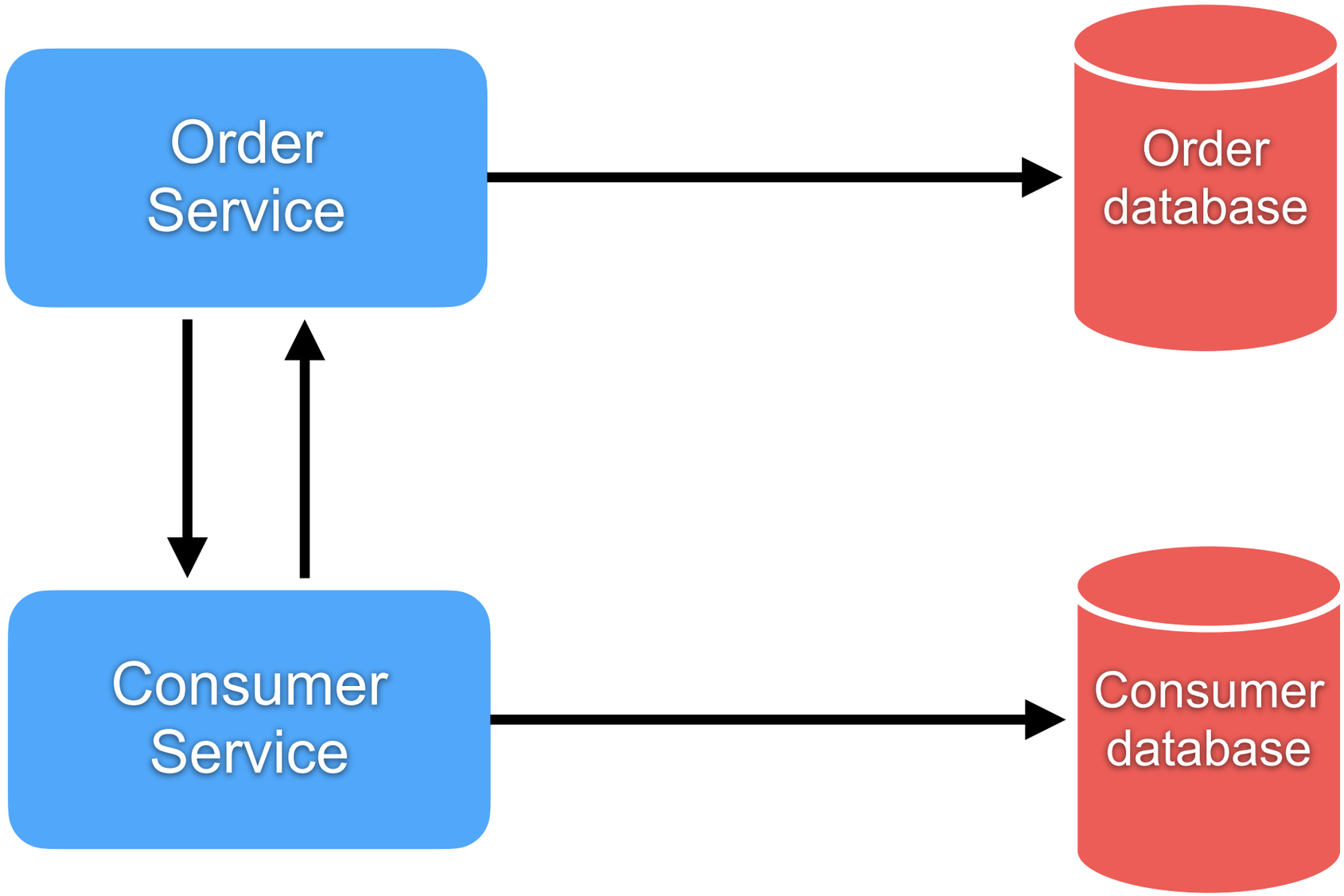
Database per service

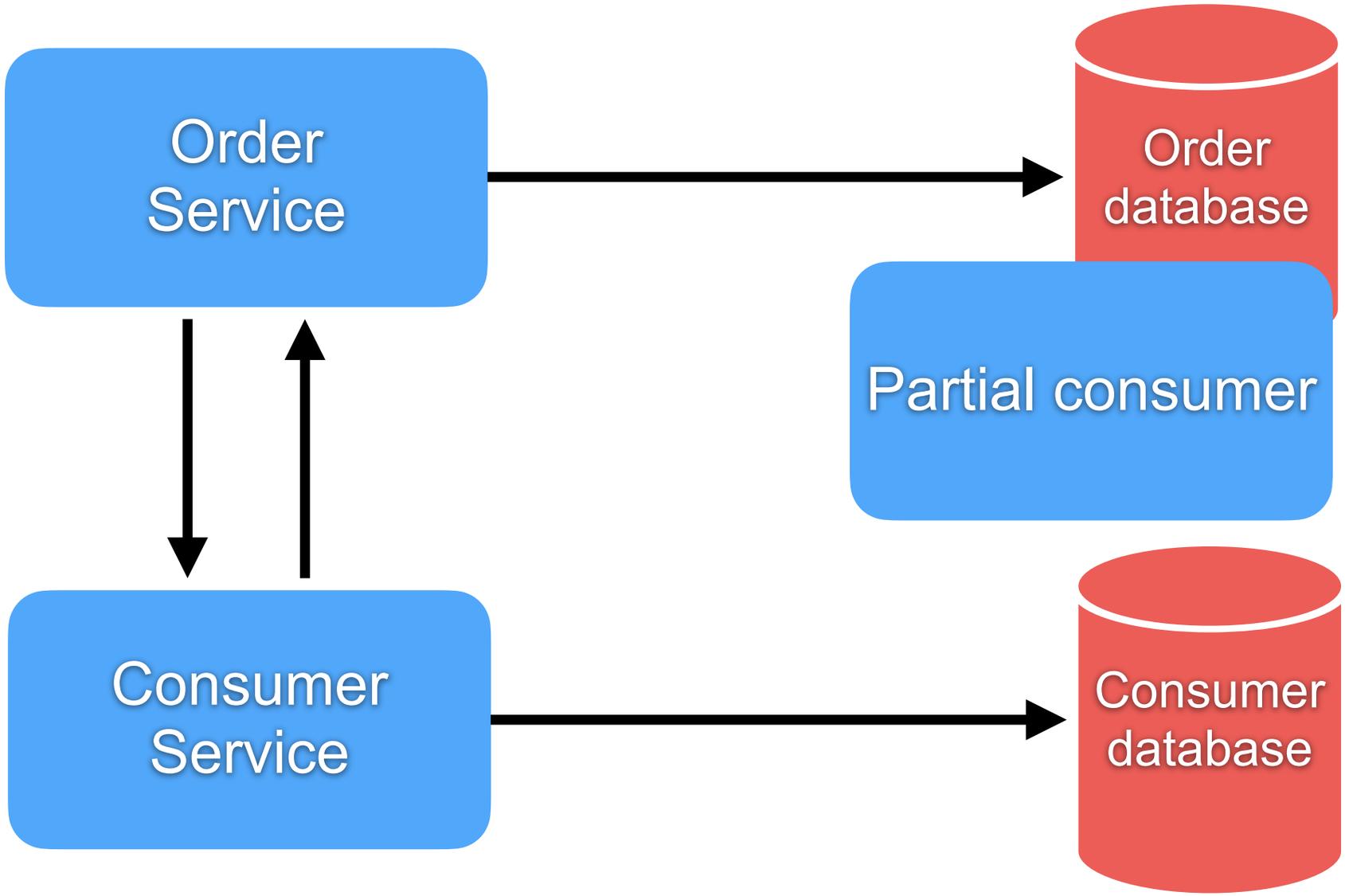


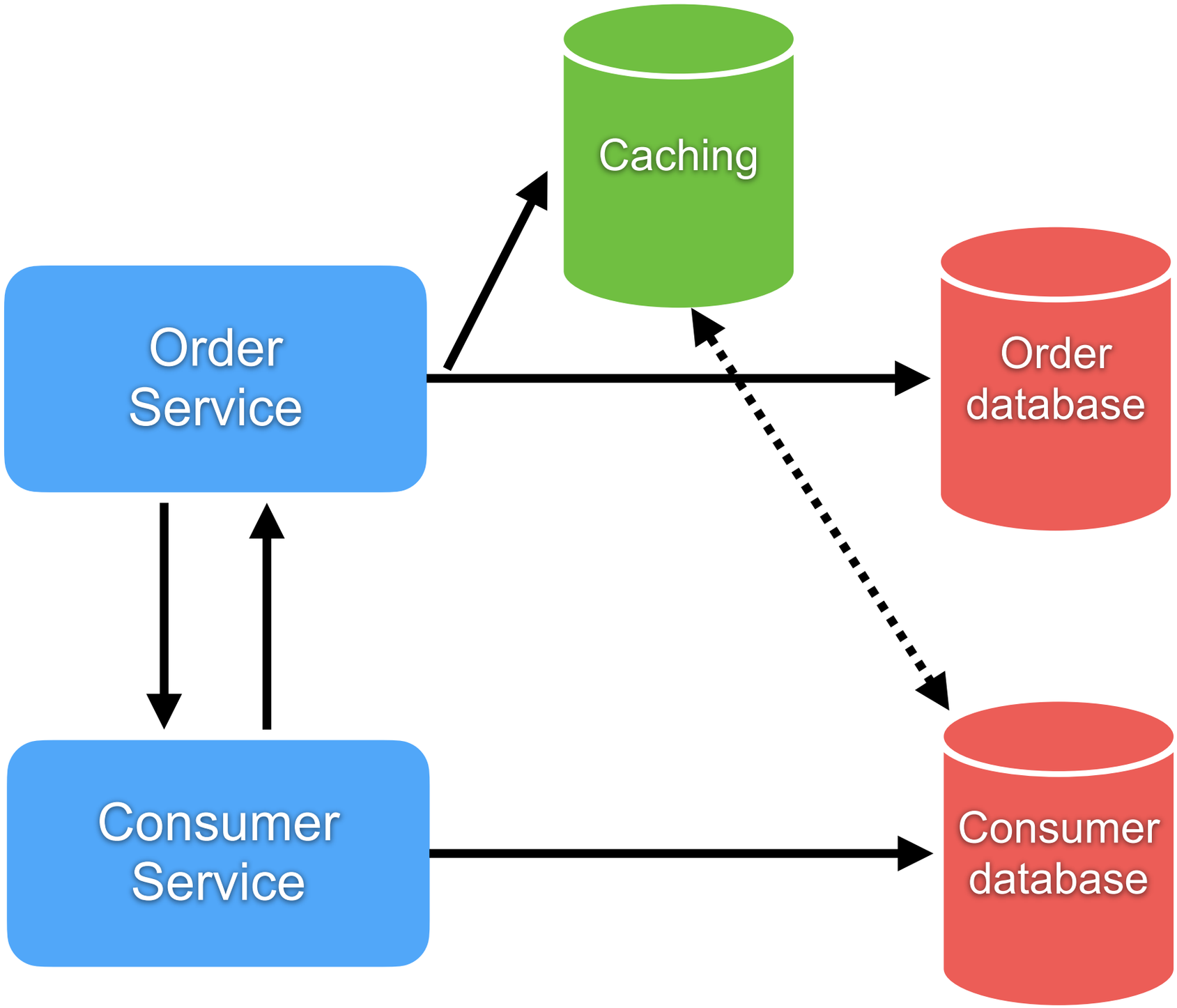
Database per service



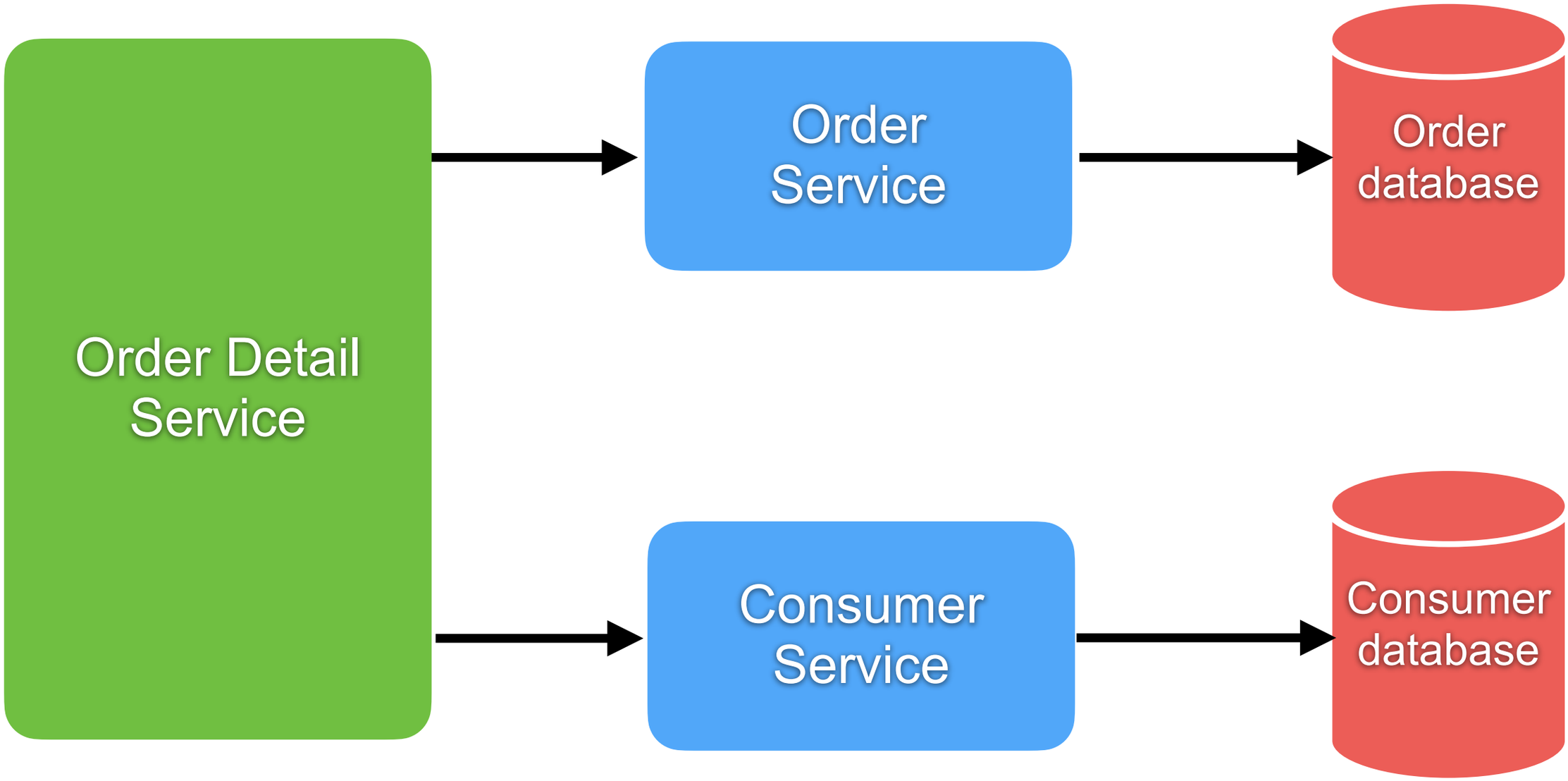
Loose coupling = encapsulation data







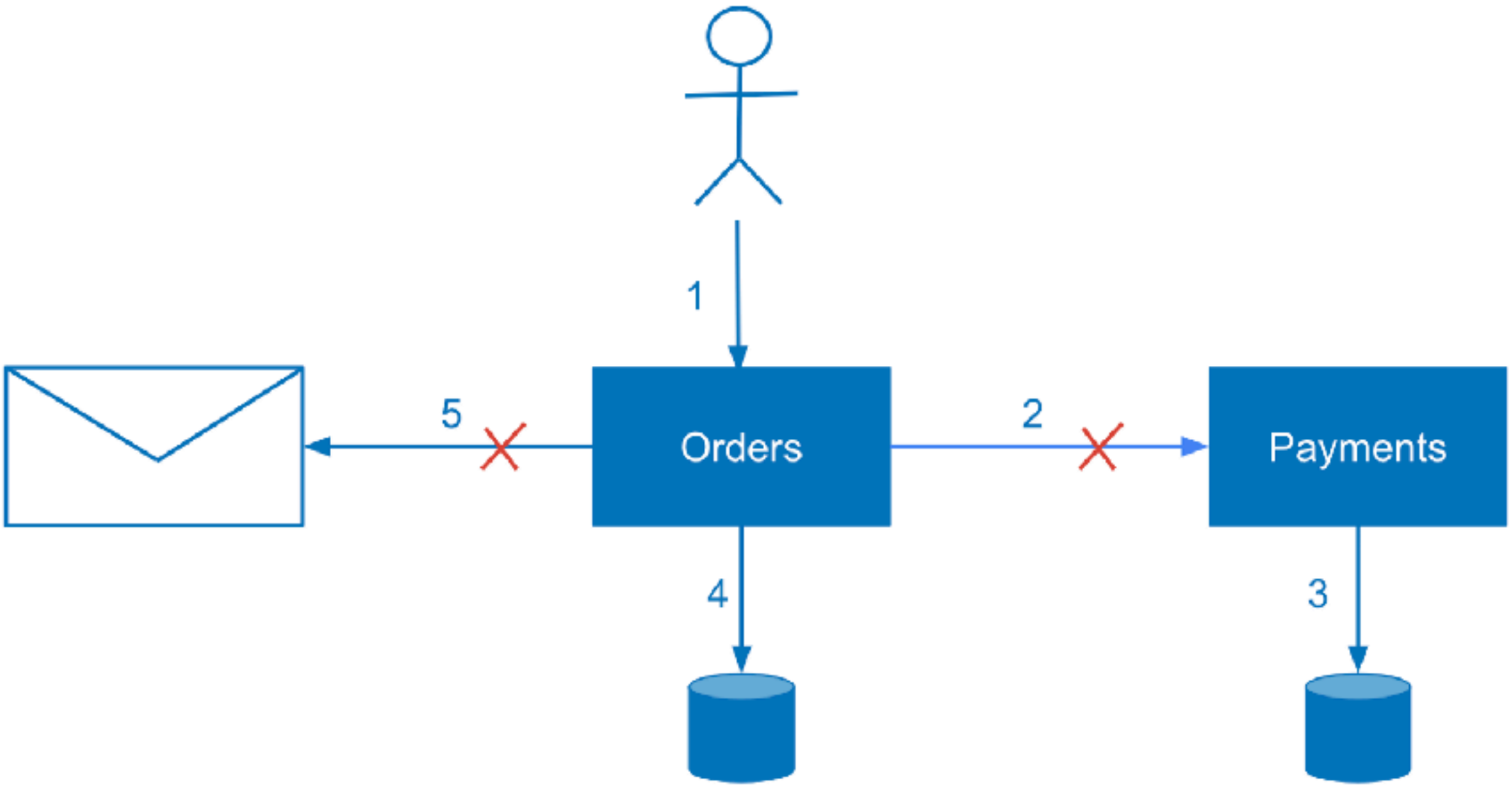
Order detail orchestration service



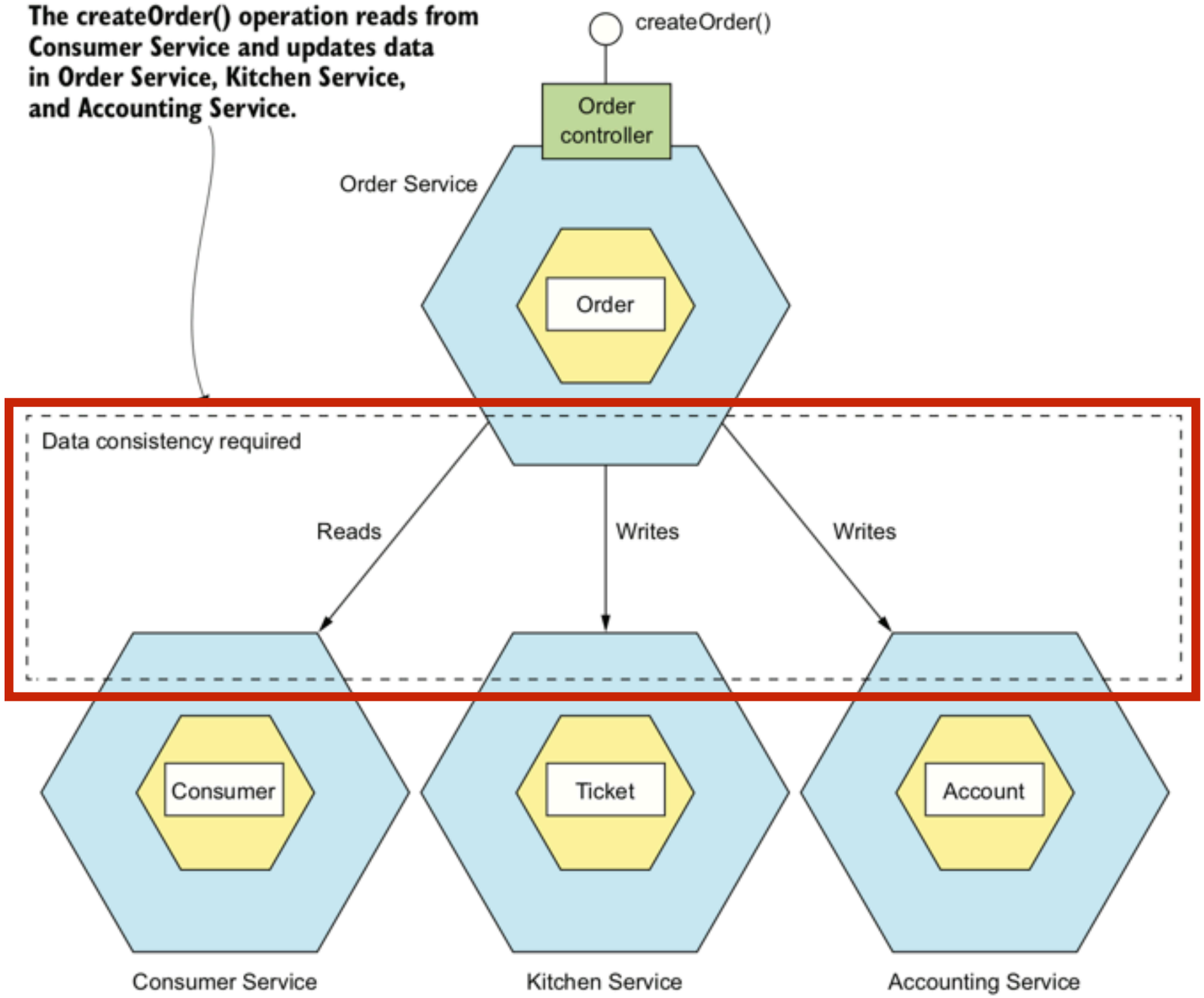
Manage transaction ?

Manage data consistency ?

Distributed process failure !!



Problem ?



How to maintain data consistency across services ?

Can’t use ACID transaction !!

A - Atomicity

All or Nothing Transactions

C - Consistency

Guarantees Committed Transaction State

I - Isolation

Transactions are Independent

D – Durability

Committed Data is Never Lost

(c) http://blog.sqlauthority.com

CAP Theorem

Consistency

Availability

Partition Tolerance

http://robertgreiner.com/2014/08/cap-theorem-revisited/

Database per service

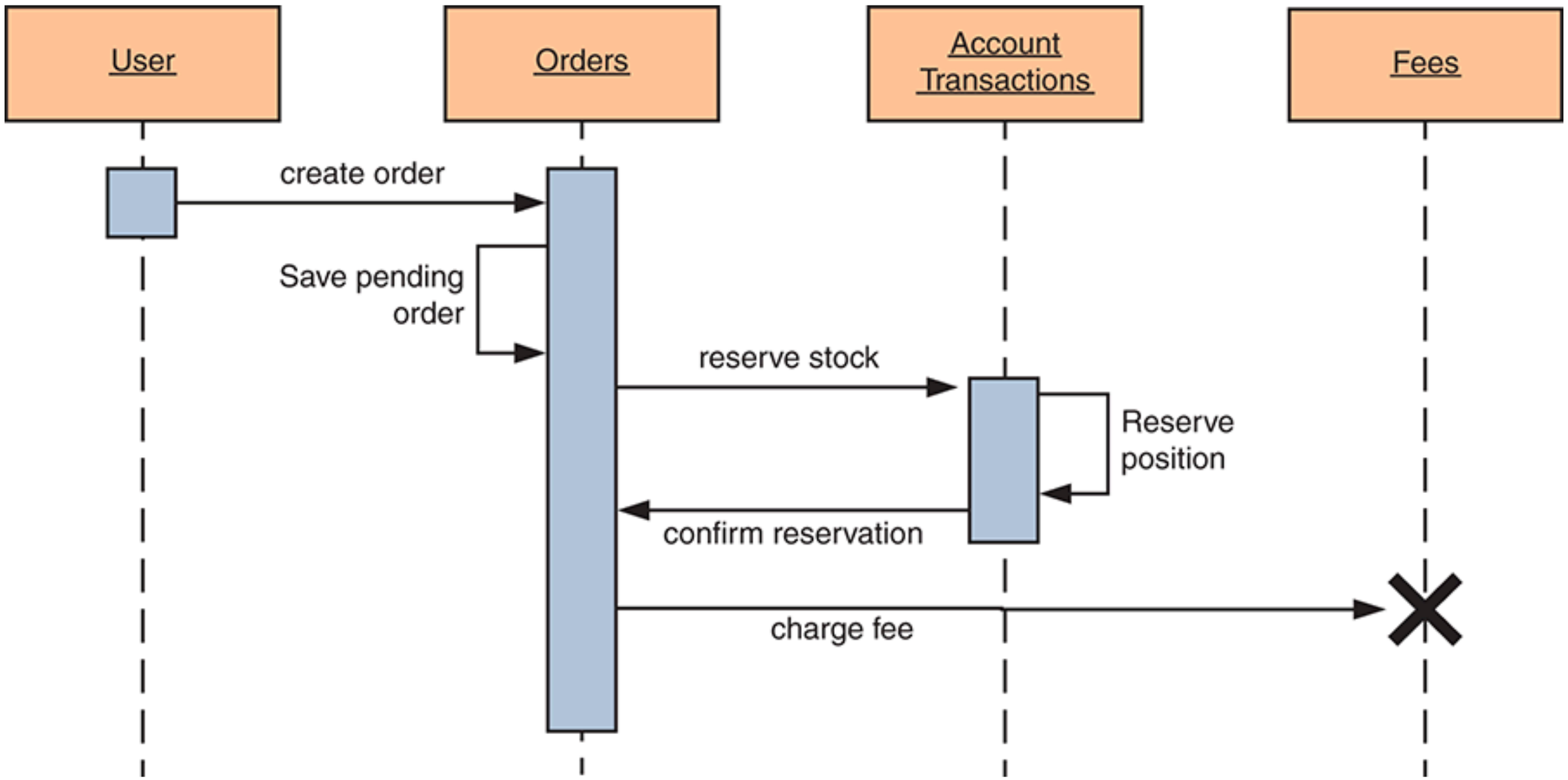
High complexity

Query data across multiple databases

Challenge “How to join data ?”

Maintain database consistency

Problem



Distributed transaction

Common approach is Two-phase commit(2PC)

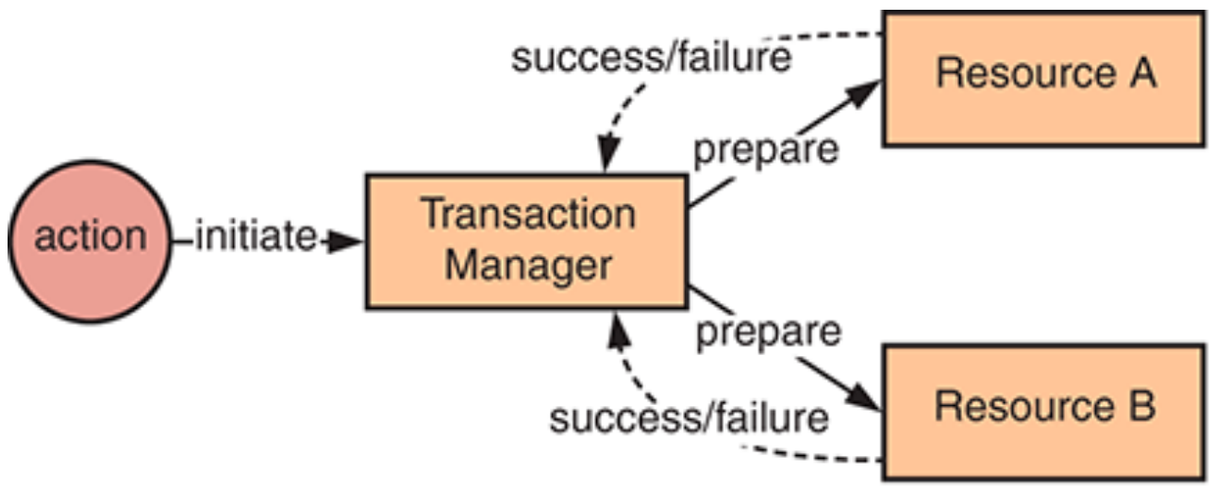
Use transaction manager to split operations across multiple resources in 2 phases

1. Prepare

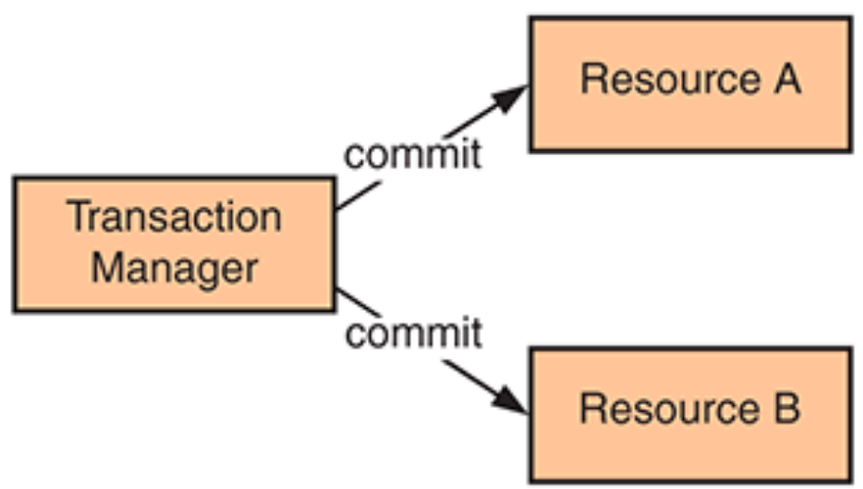
2. Commit

Two-phase commit

Phase 1: prepare



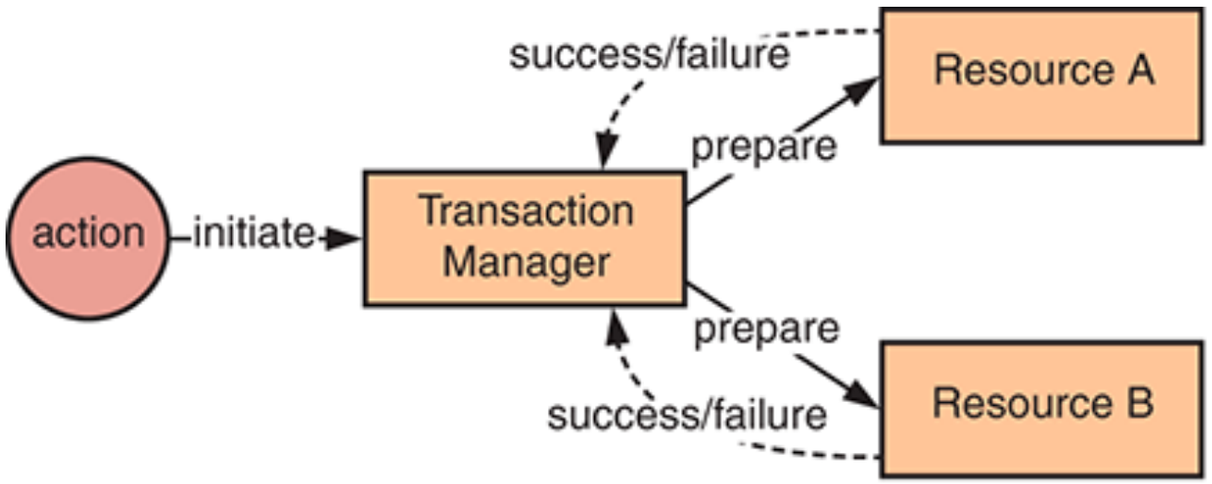
In the **prepare** phase, the transaction manager instructs resources to prepare their relevant action.



In the **commit** phase, the transaction manager instructs resources to commit or abort the prepared action.

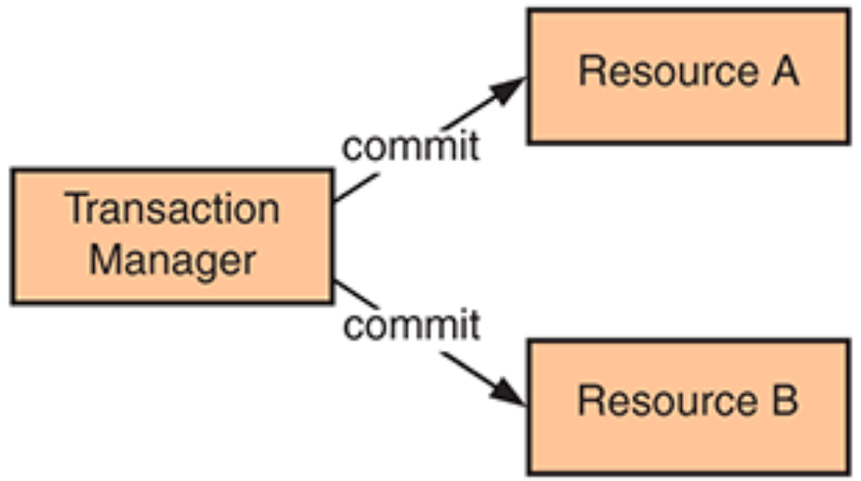
https://en.wikipedia.org/wiki/Two-phase_commit_protocol

Two-phase commit



In the **prepare** phase, the transaction manager instructs resources to prepare their relevant action.

Phase 2: commit



In the **commit** phase, the transaction manager instructs resources to commit or abort the prepared action.

https://en.wikipedia.org/wiki/Two-phase_commit_protocol

Distributed transaction places a lock on resources under transaction to ensure isolation

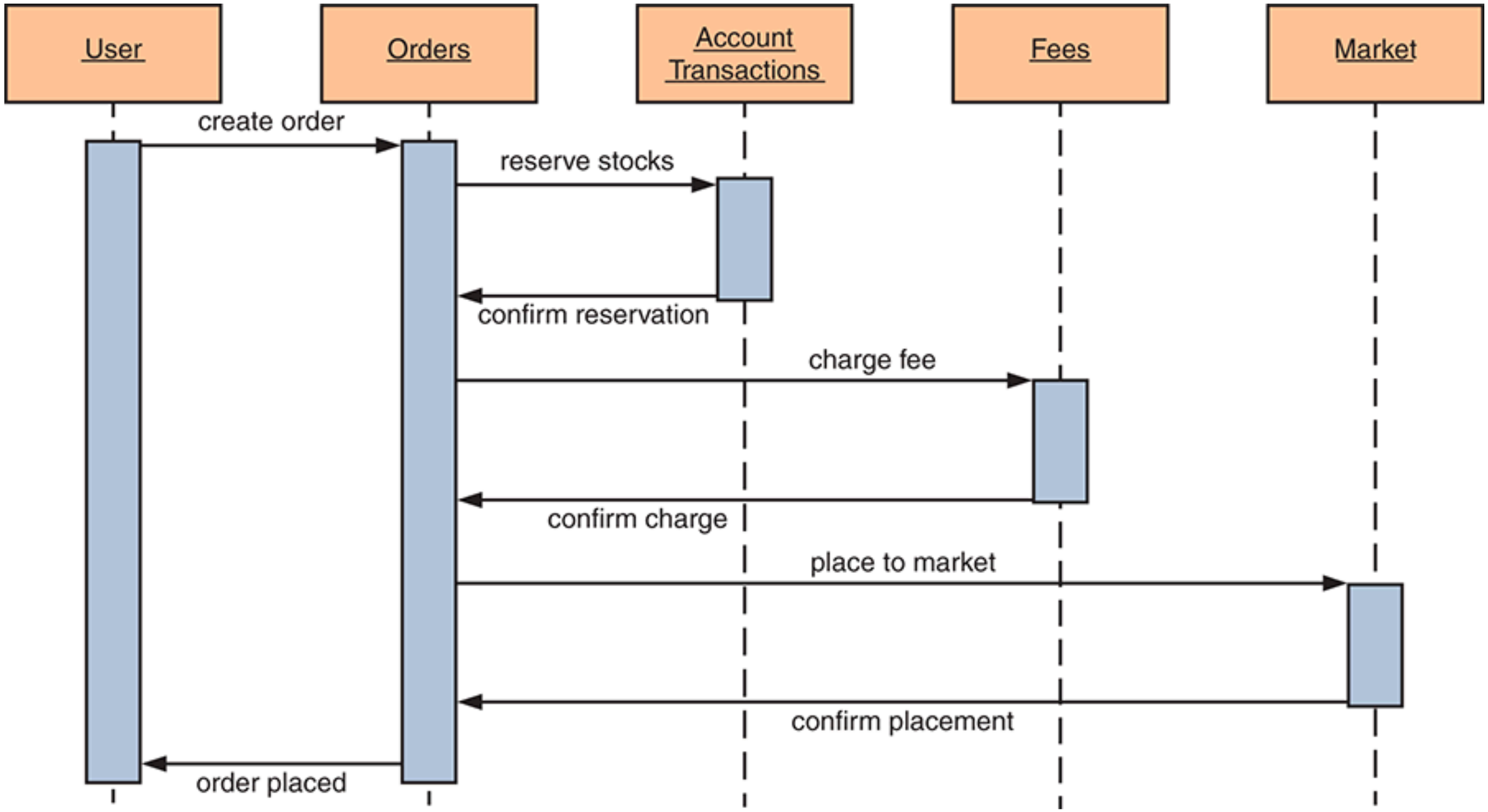
Inappropriate for long-running operations

Increase risk of contention and deadlock

We need …

Don’t need distributed transaction
Reduce complexity of code
Reliable

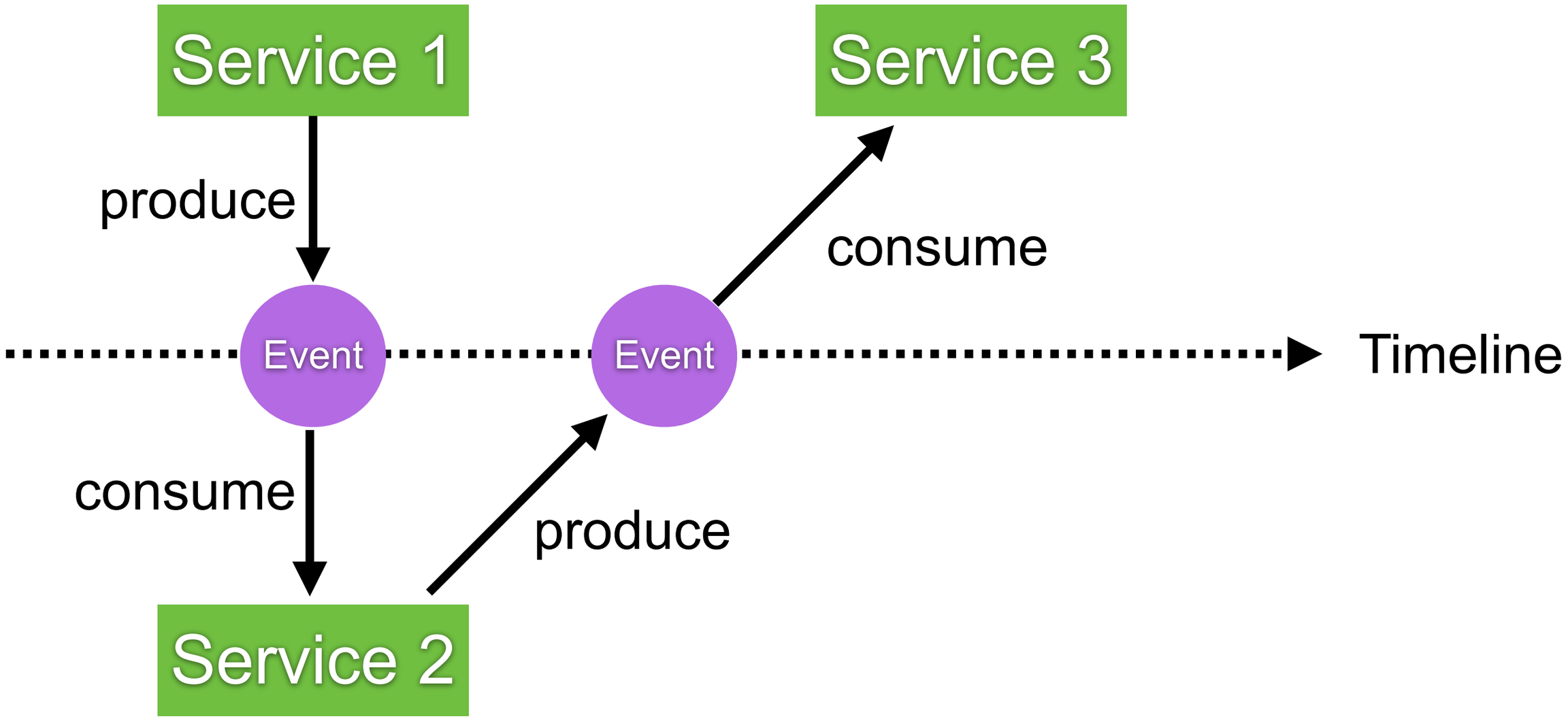
Synchronous process



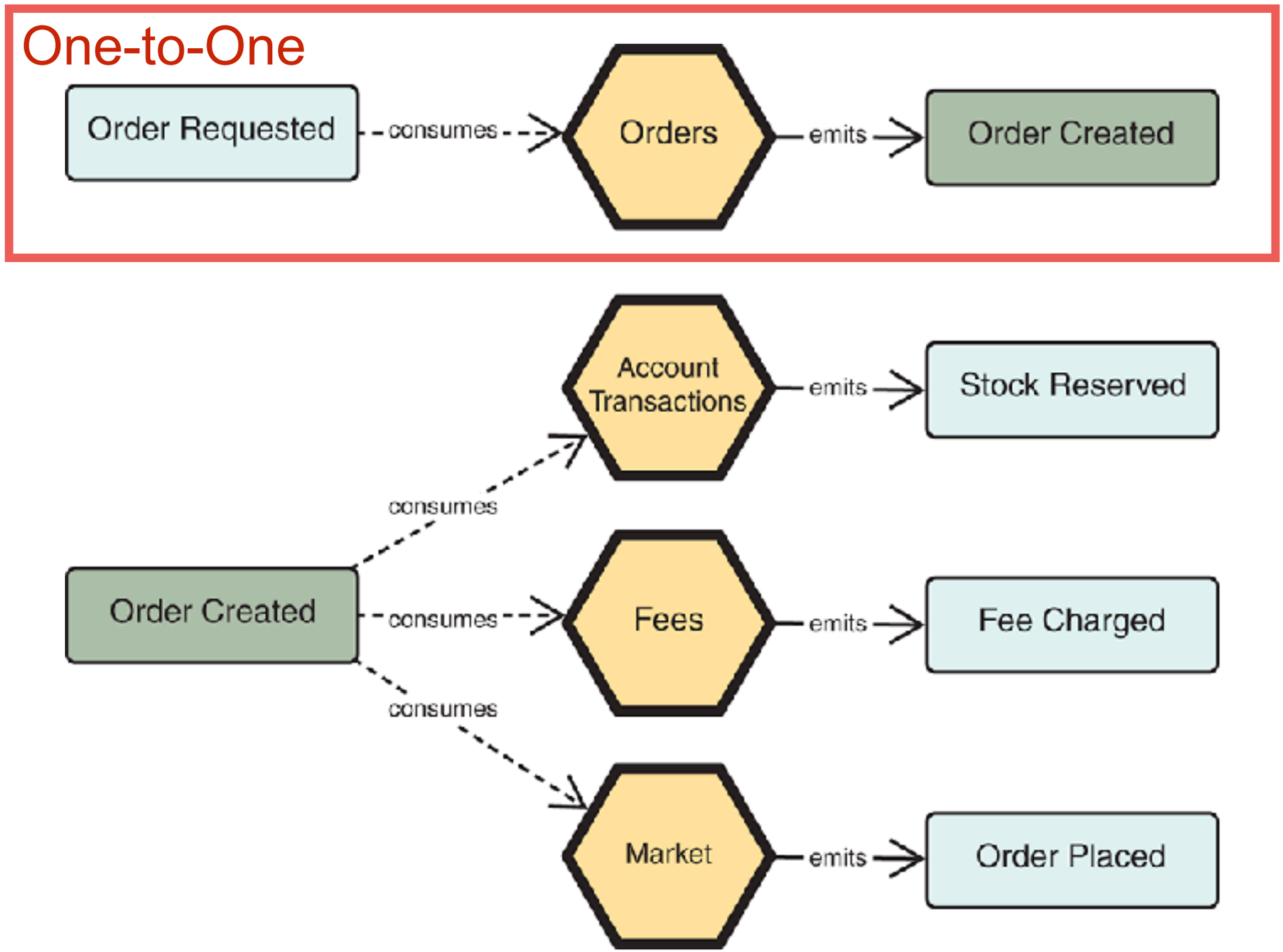
Event-based communication

Choreography pattern

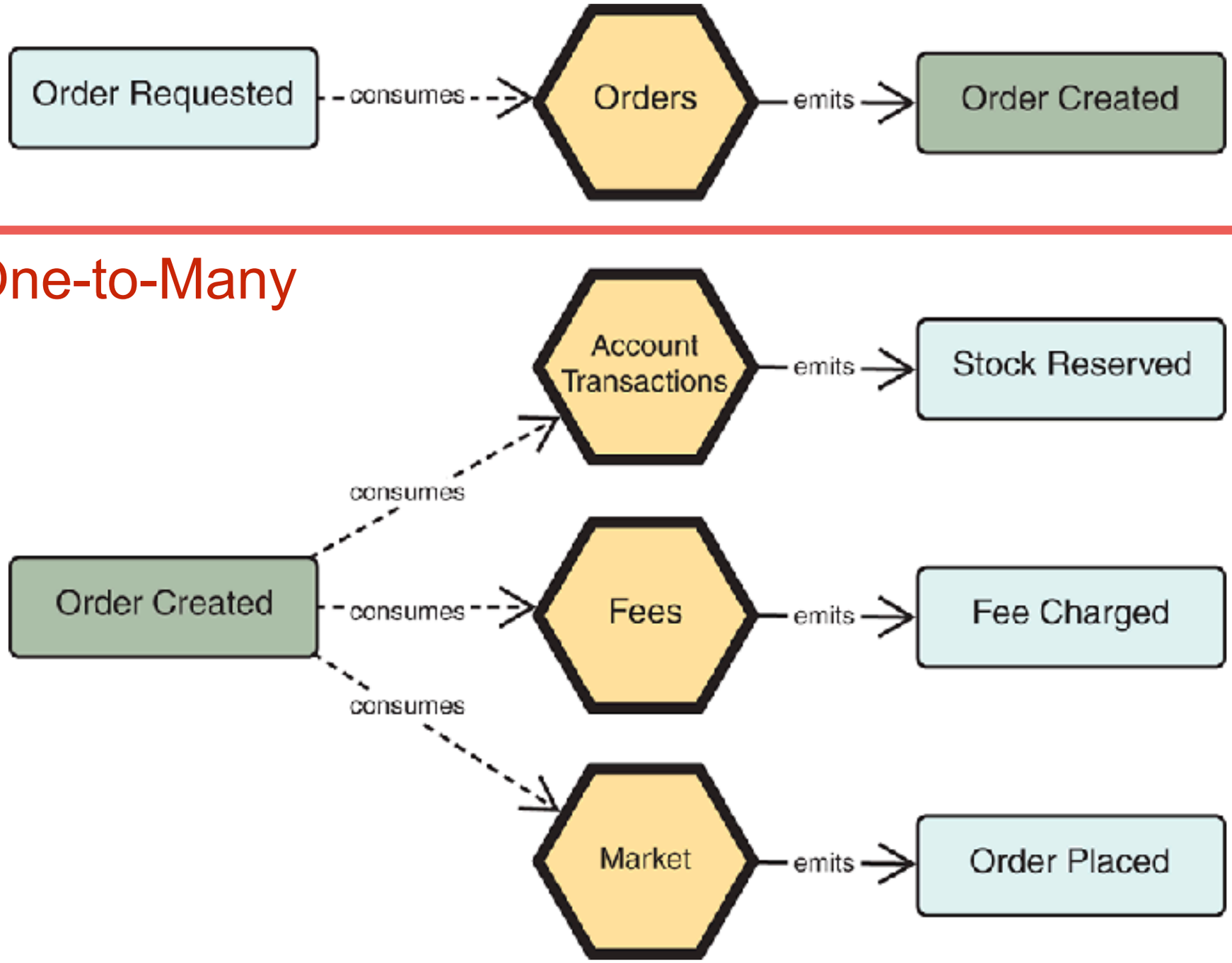
Sequence of Tx and emit **event** or **message** that trigger the next process in Tx



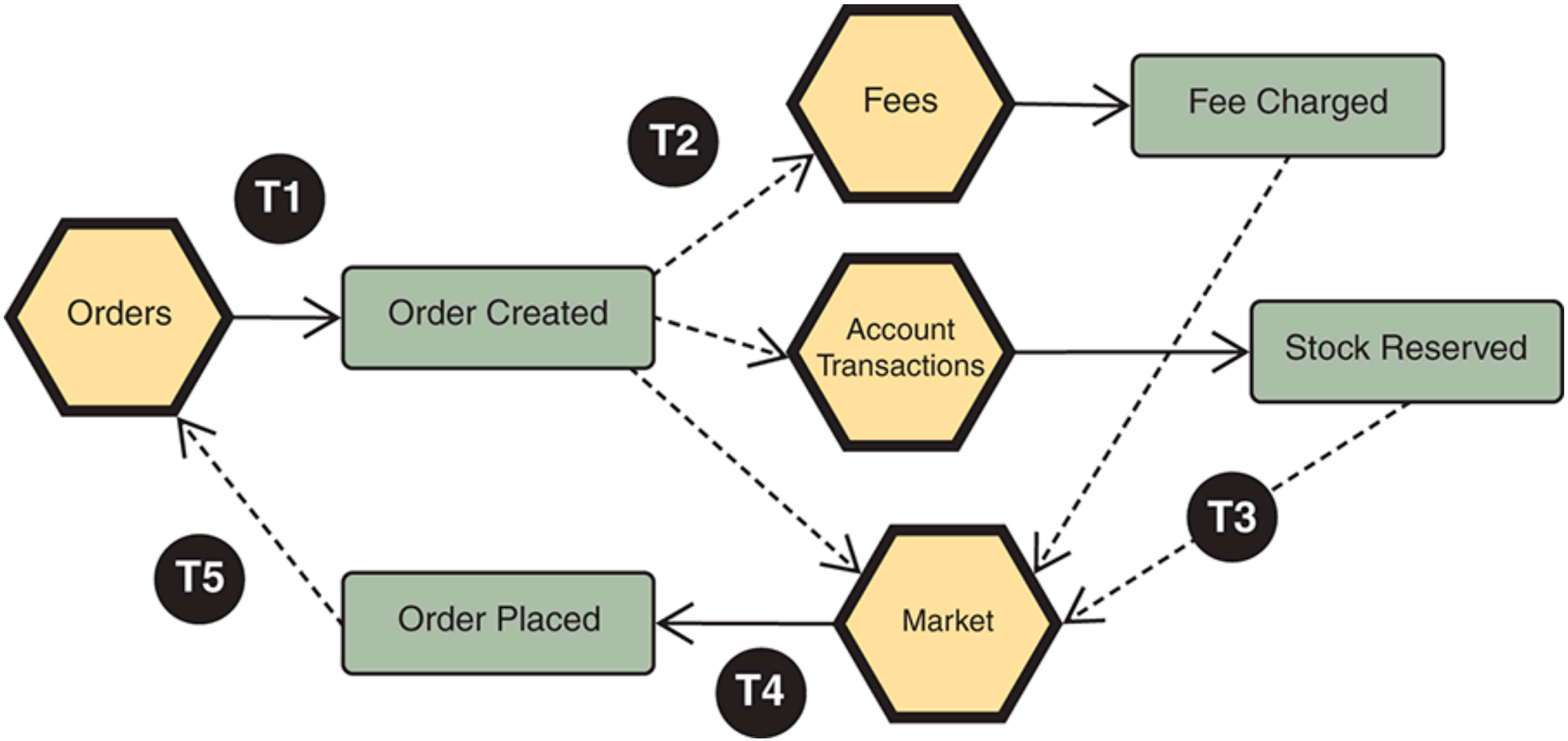
Service consume and emit event



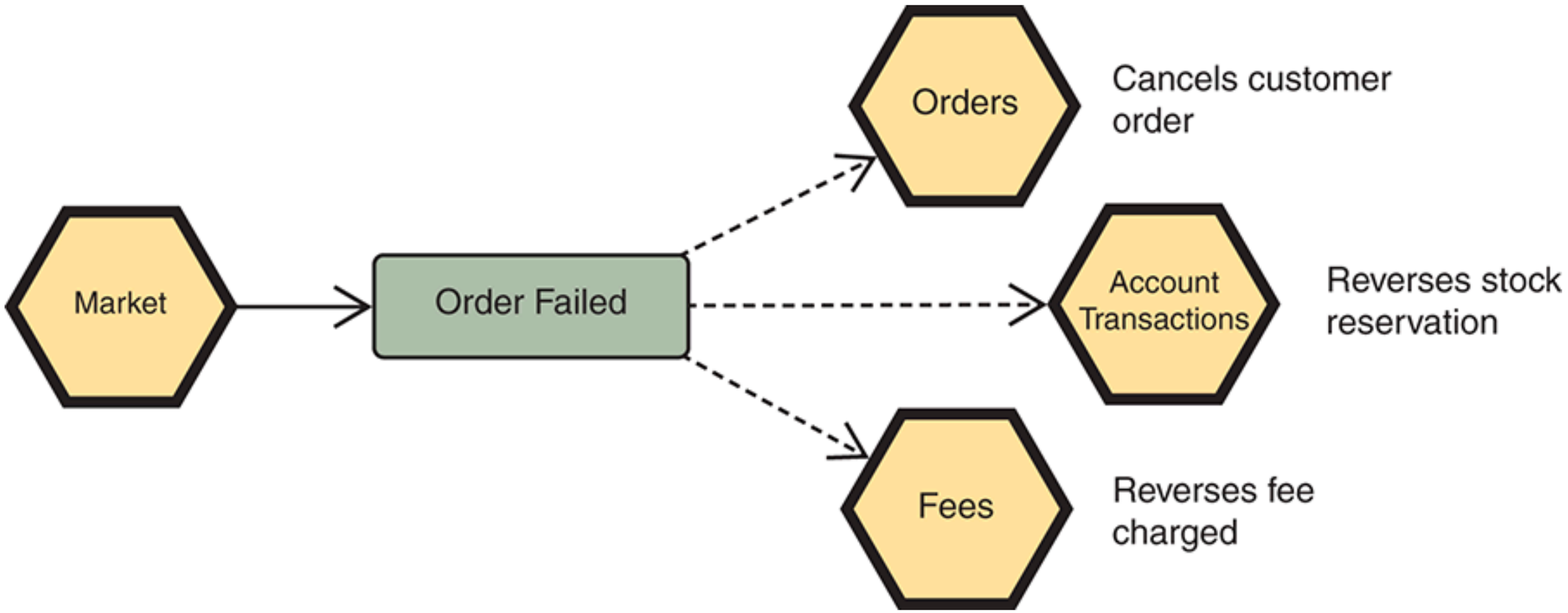
Service consume and emit event



Example (Success)

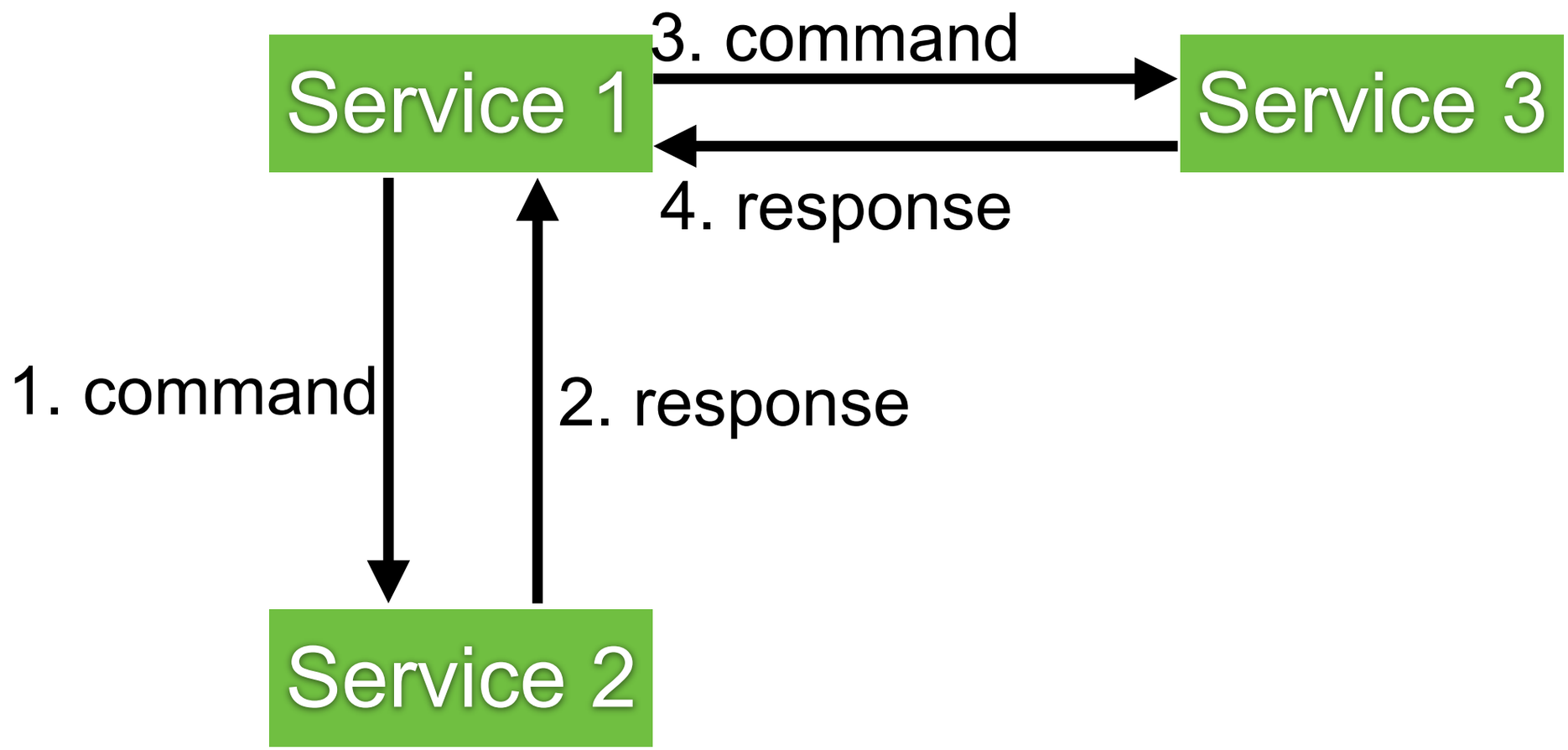


Example (Fail)

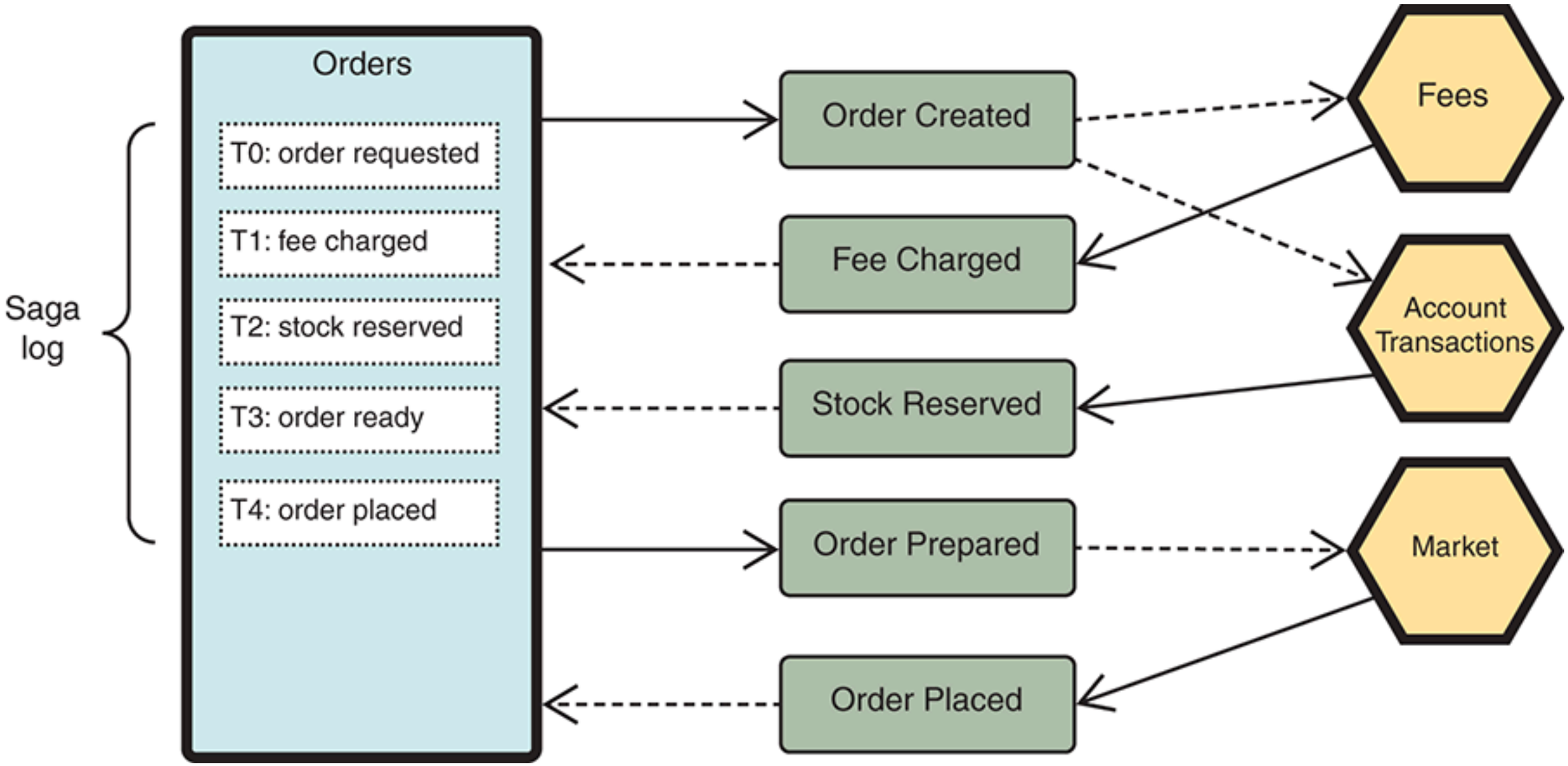


Orchestrate pattern

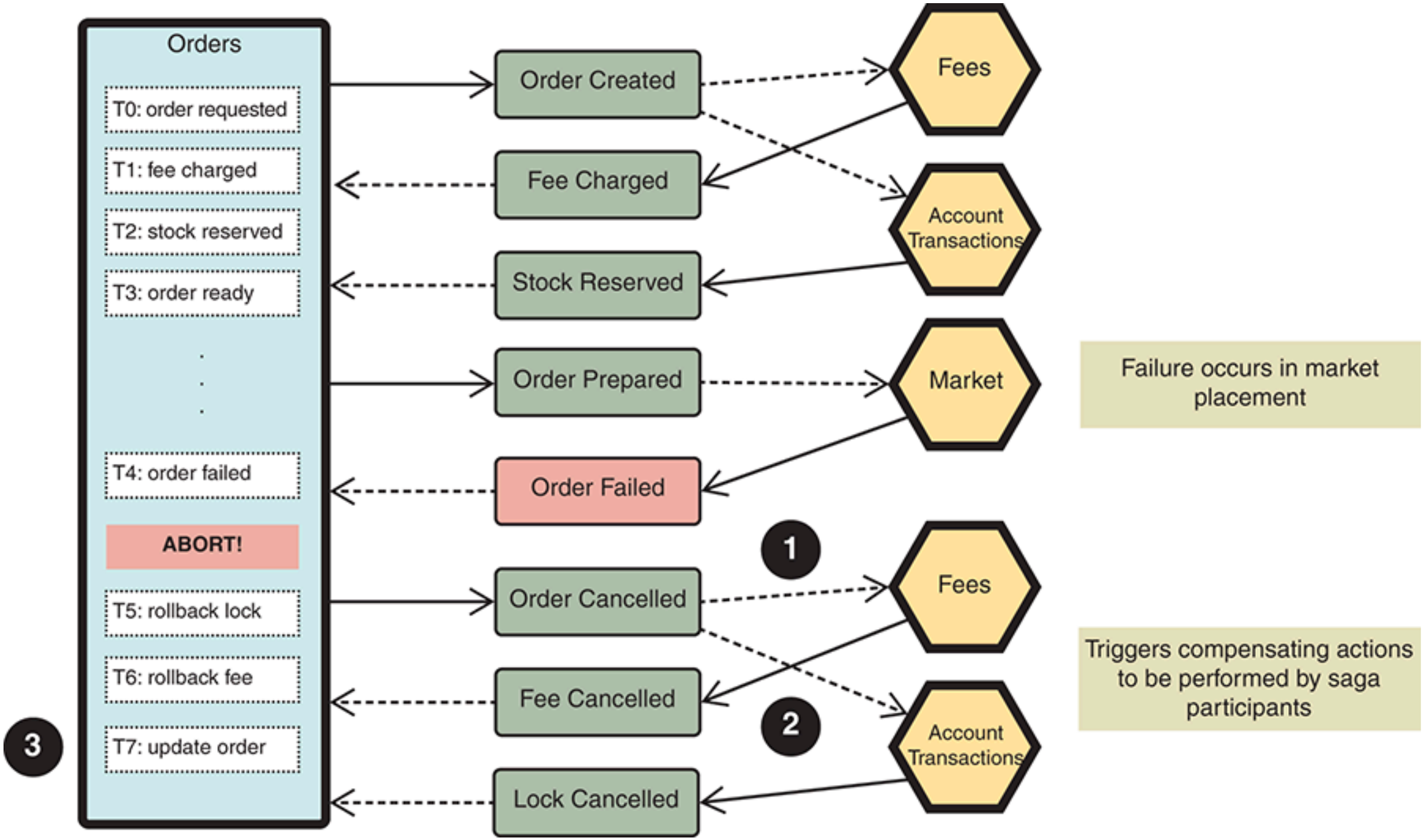
Sequence of Tx and emit **event** or **message** that trigger the next process in Tx



Example (Success)



Example (Fail)



Consistency patterns

Name	Strategy
Compensating action	Perform action that undo prior action(s)
Retry	Retry until success of timeout
Ignore	Do nothing in the event of errors
Restart	Reset to the original state and start again
Tentative operation	Perform a tentative operation and confirm (or cancel) later

“Saga”

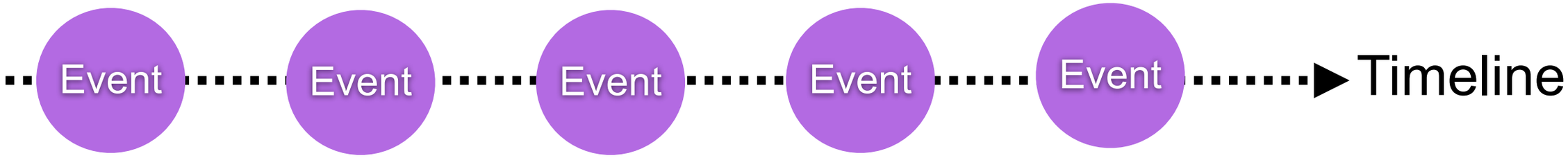
Message vs Event

Message is addressed to someone

Event is something that happen and someone can react to that

Event sourcing

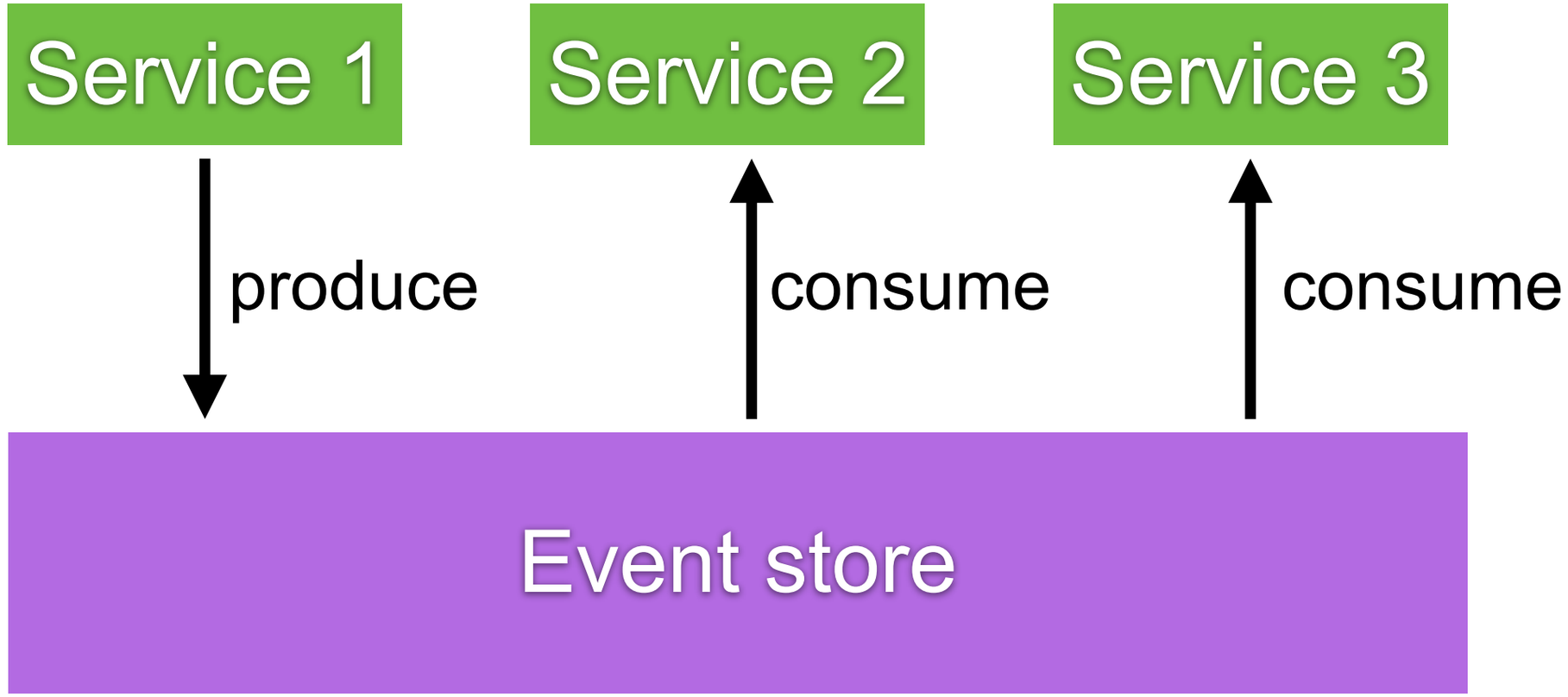
Maintain source of truth of business
Immutable sequence of events



Sequence of event is keep in **Event store**

Event store

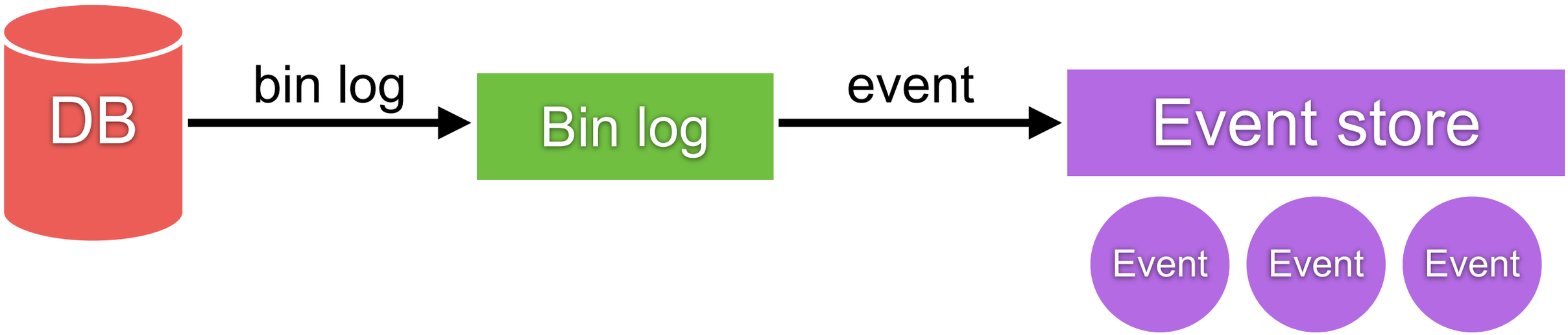
Keep events in order
Broadcast new event



With Event sourcing, we can **decouple services** (query and command)

Event sourcing with database

Working with database



Binlog or Binary log event is information about data modification made to database

Event sourcing

Duplication data (denormalize database)
Complexity (separate query and command)
Difficult to maintain

Event sourcing can’t solve all problems

Start with **understand** your **purpose**

Start with understand your **problem** to resolve

How to implement queries data ?

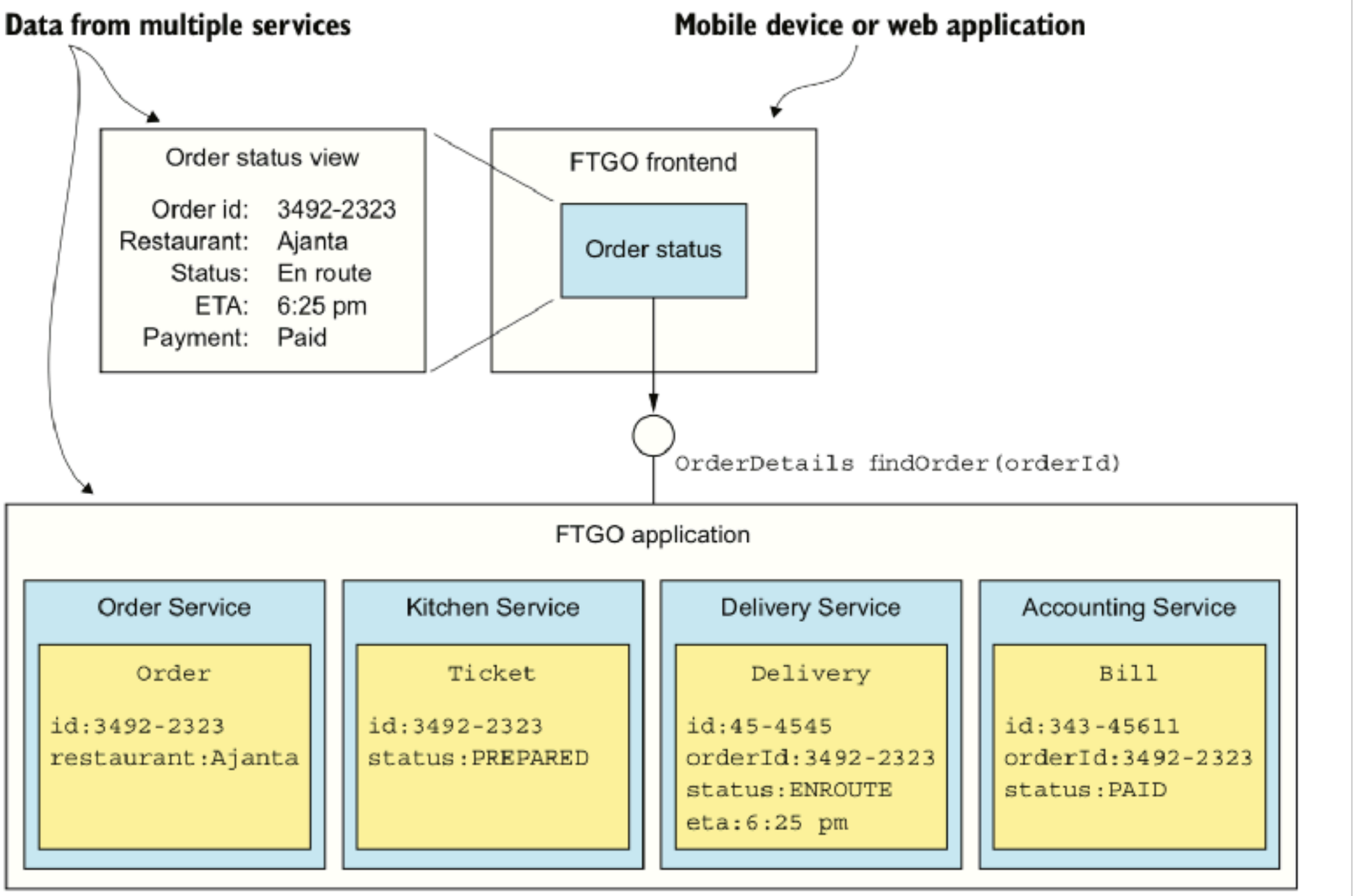
Query data from multiple services

API composition

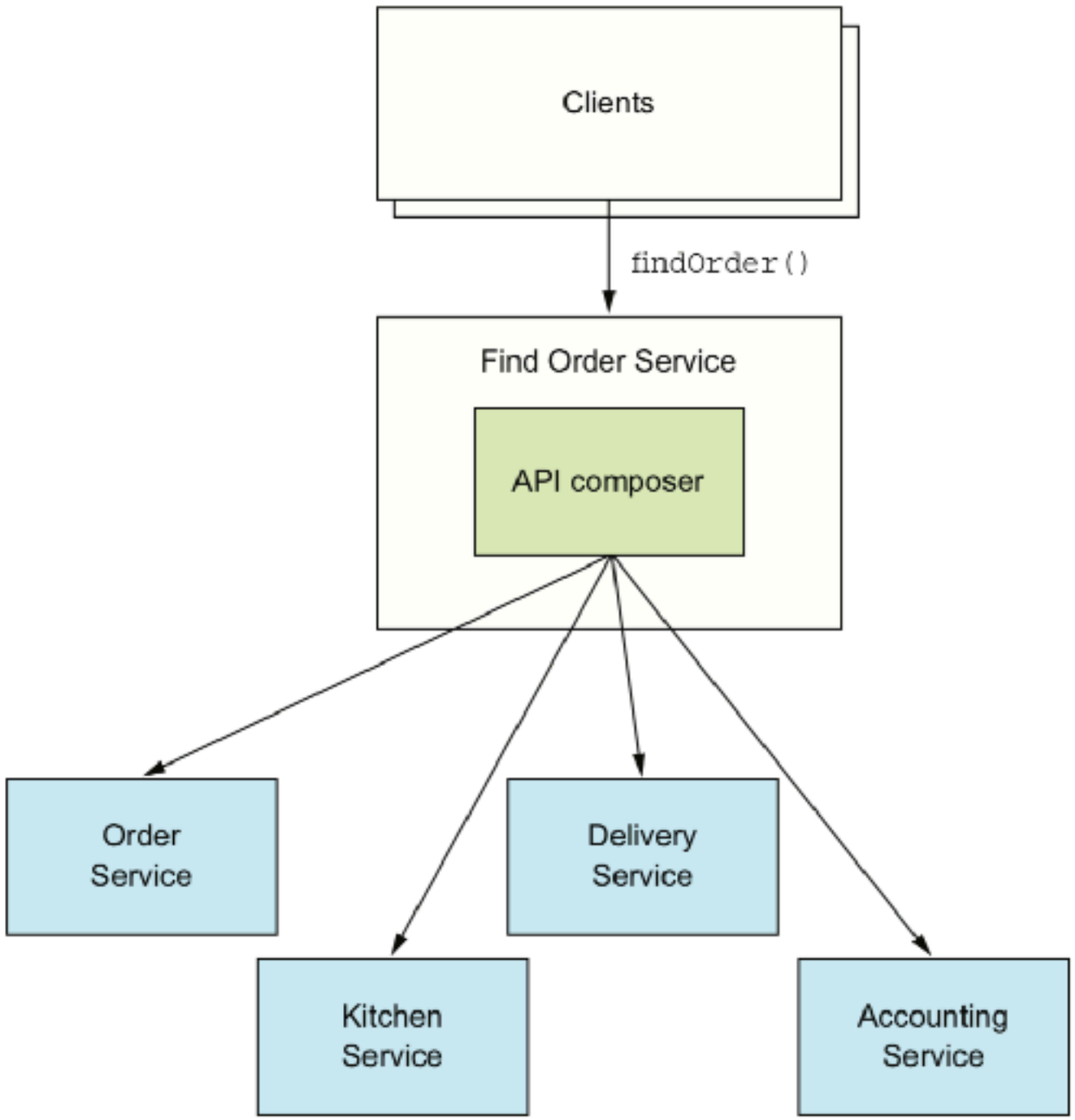
Cold data from services

CQRS (Command Query Responsibility Segregation)

Problem ?



API composition pattern



Drawbacks

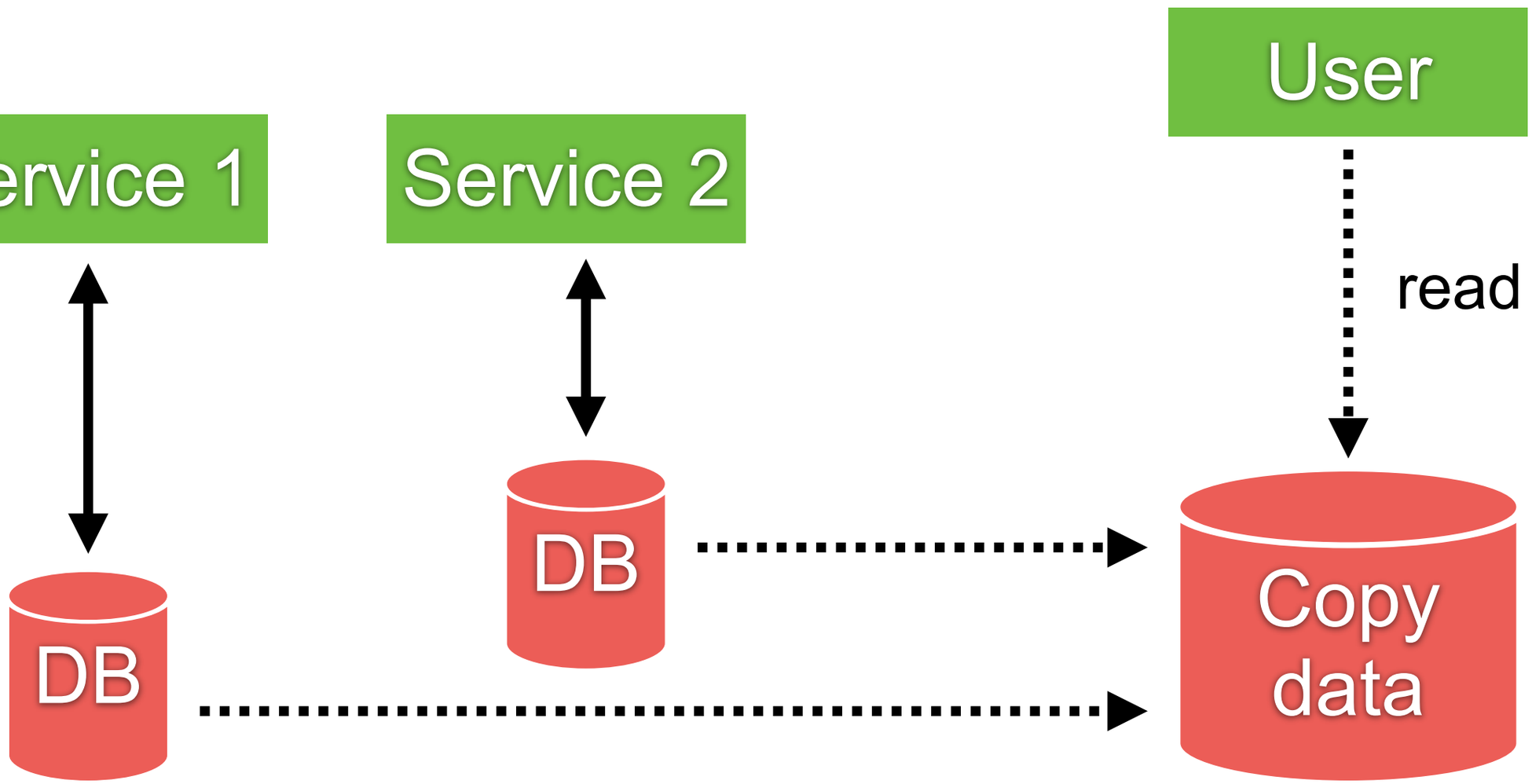
Increase overhead

Lack of transactional data consistency

Reduce availability

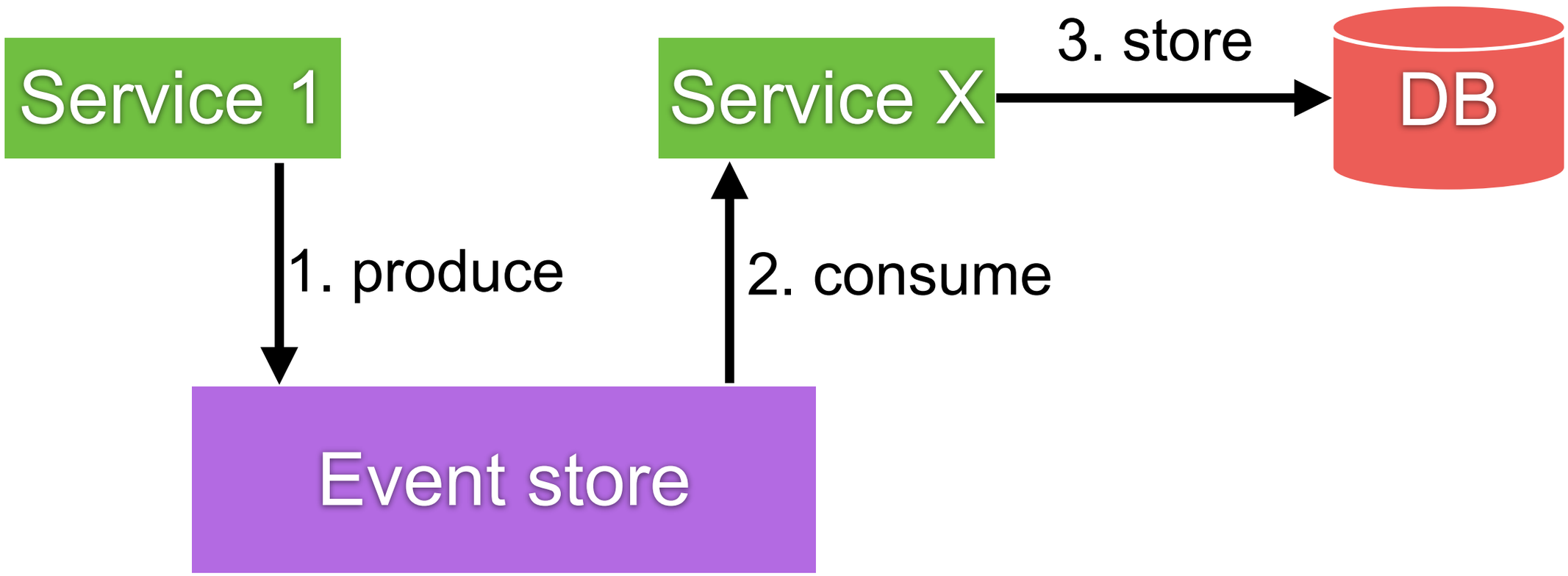
Storing copy data

Copy or caching data to other database



Working with Event-based

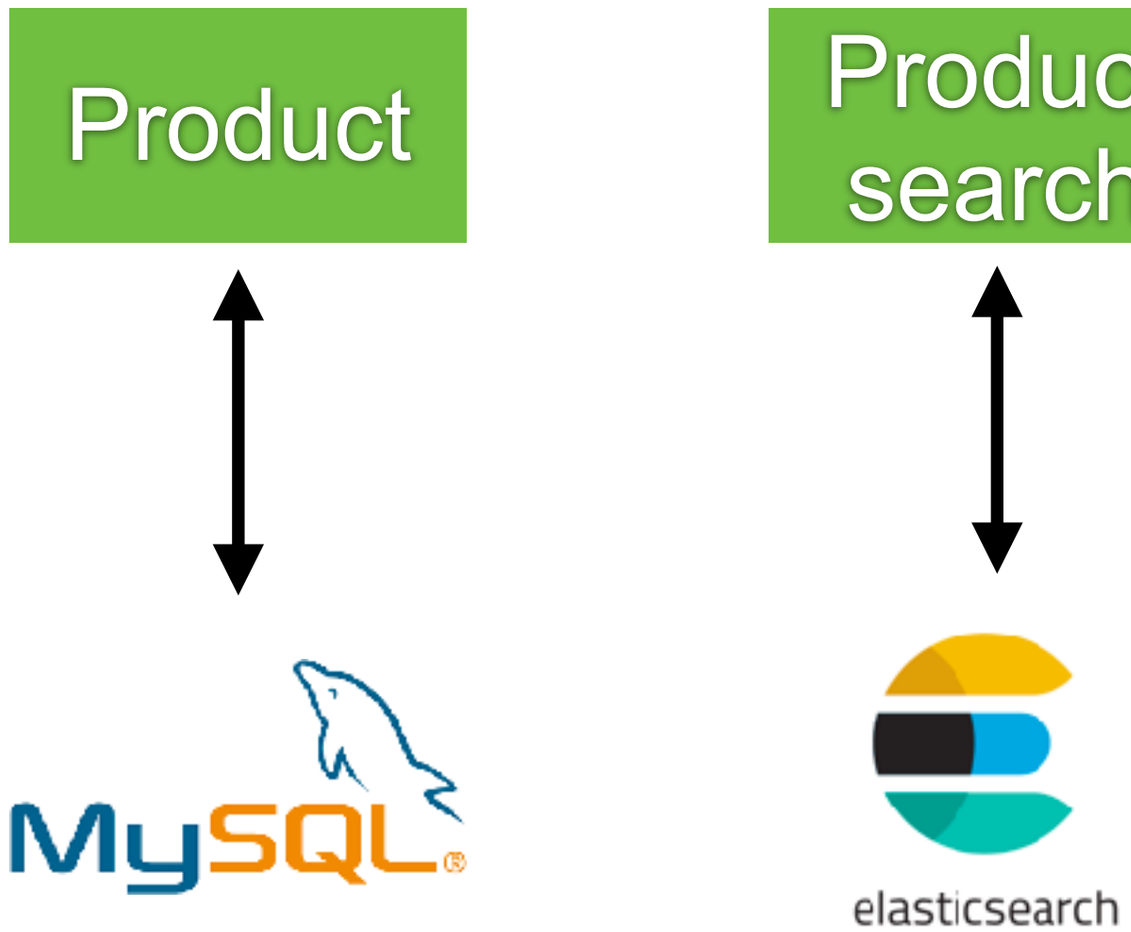
Publish event to store copy data



Separate query and command

Separate data for read and write

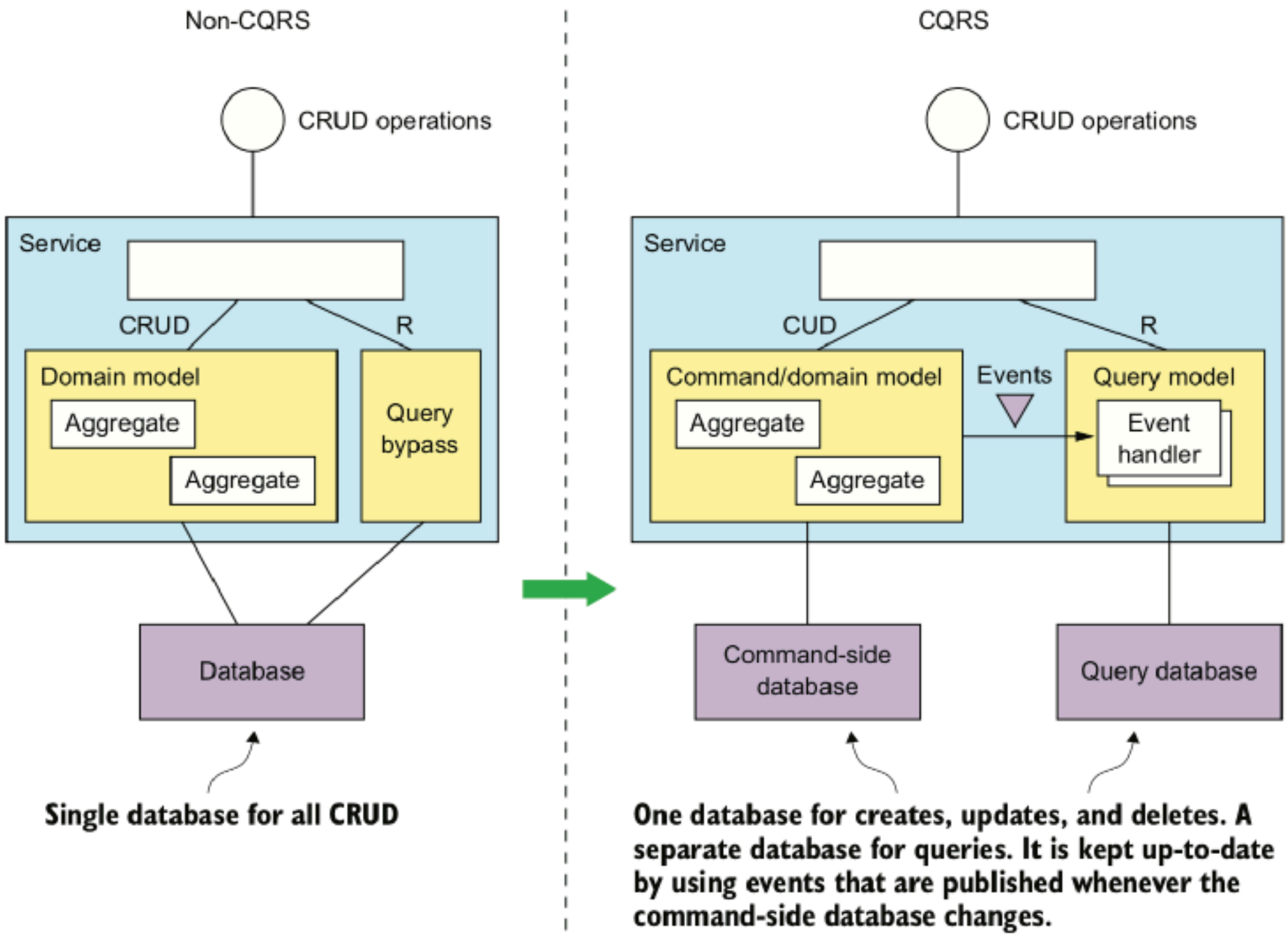
For example MySQL to write, Elasticsearch to search



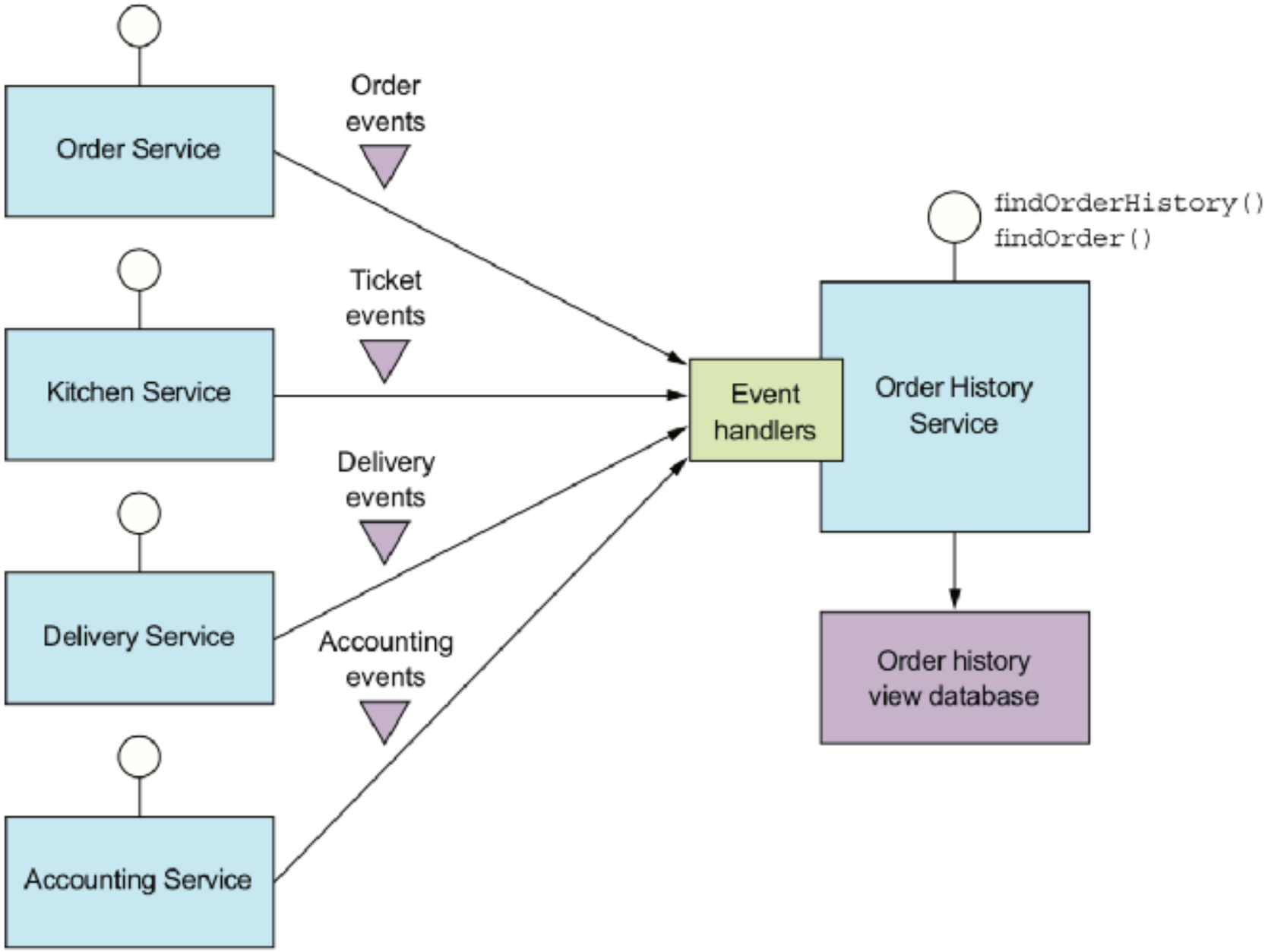
CQRS

Command **Q**uery **R**esponsibility **S**egregation

CQRS = Separate command from query



CQRS = Query-side service



Benefits

Improve separation of concern
Efficiency query

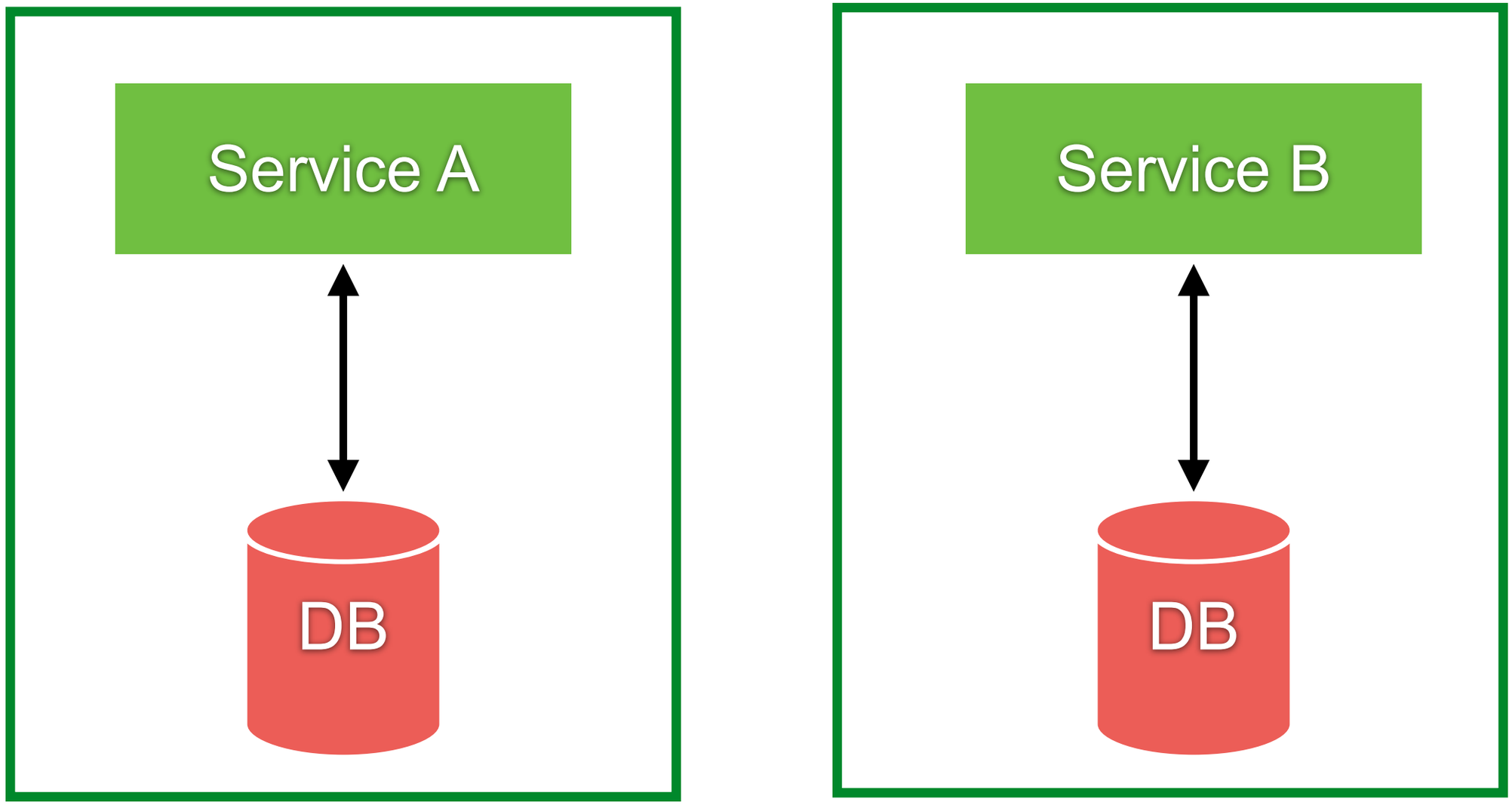
Drawbacks

More complex

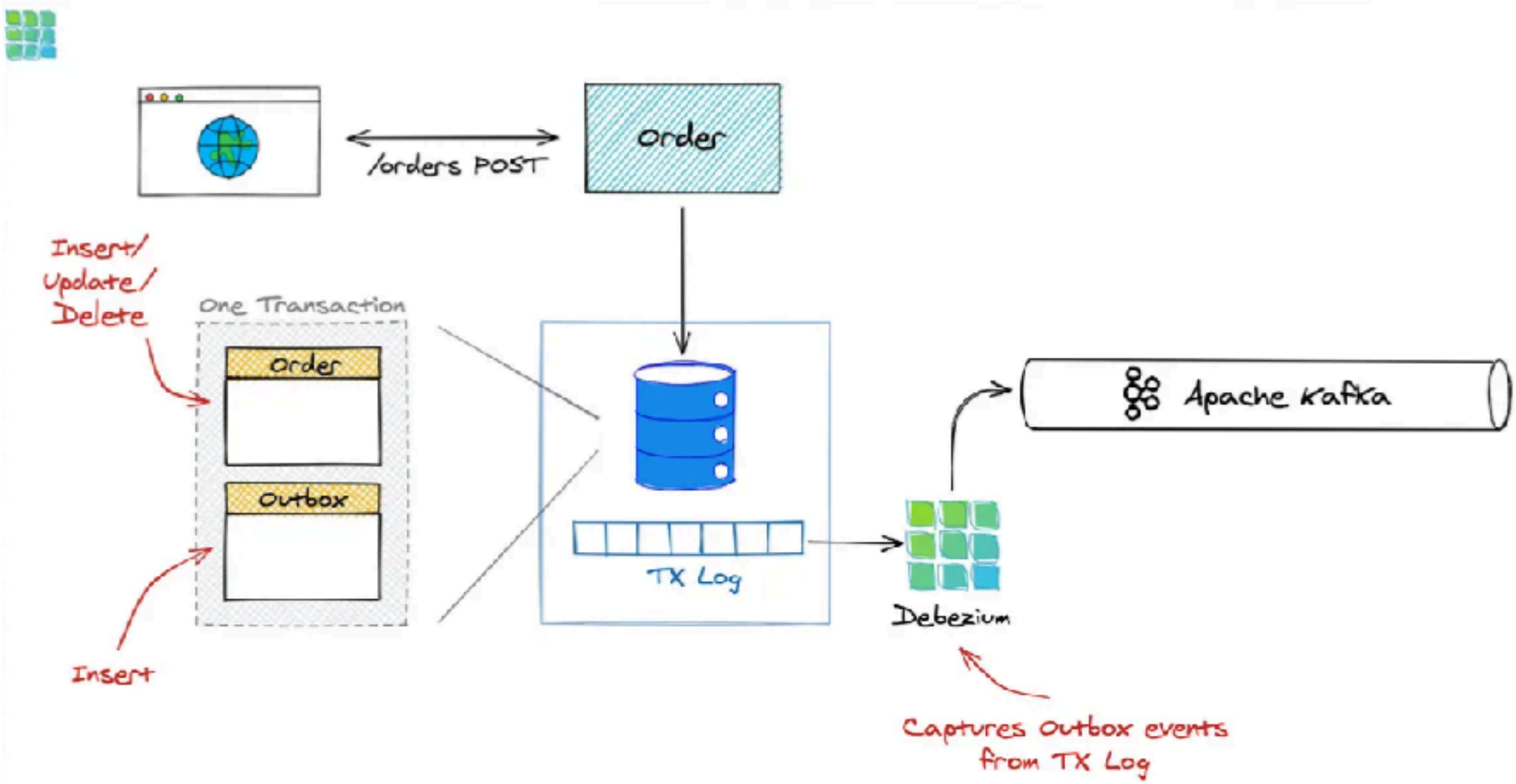
Lag between **command** and **query** side

Outbox pattern

Decouple services using Pub/Sub
to exchange data between services

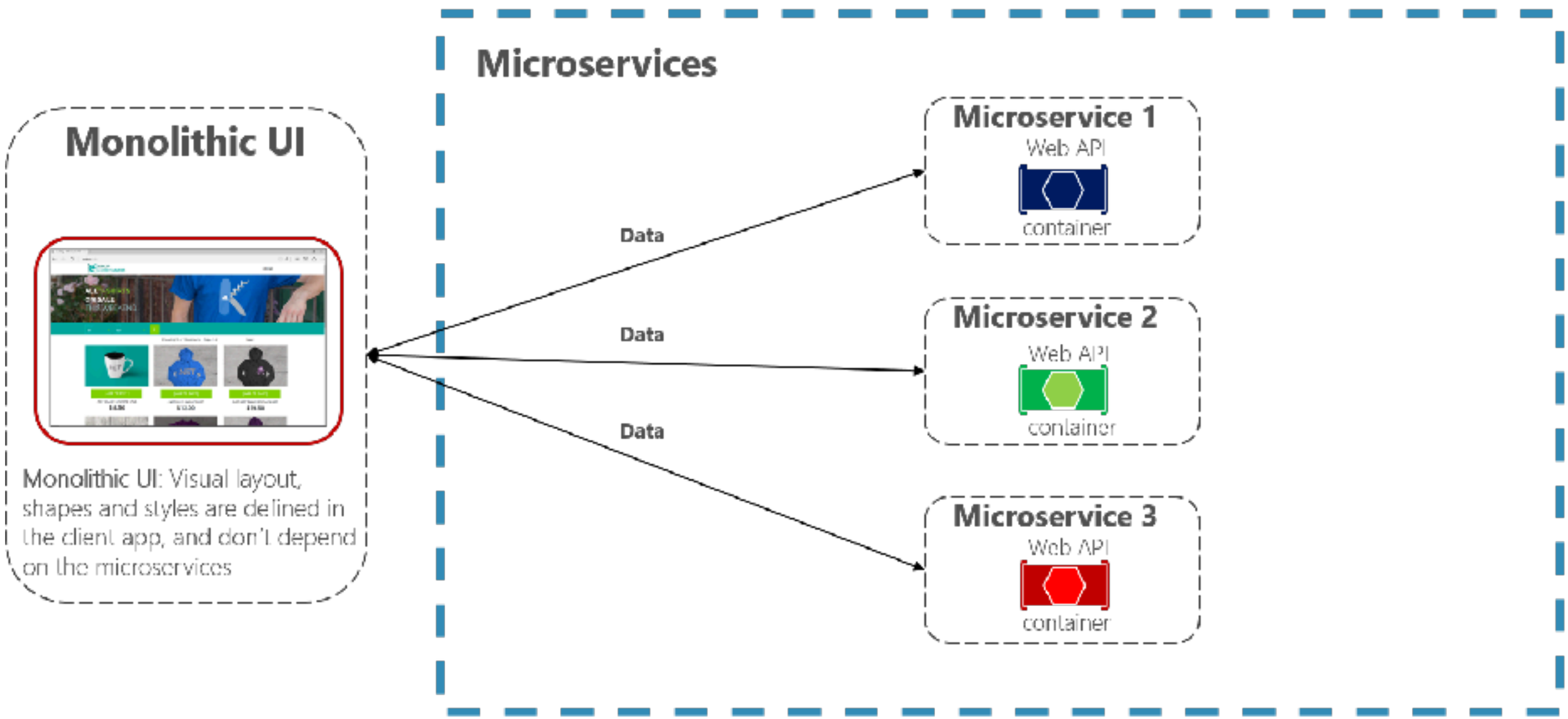


Example of Outbox pattern



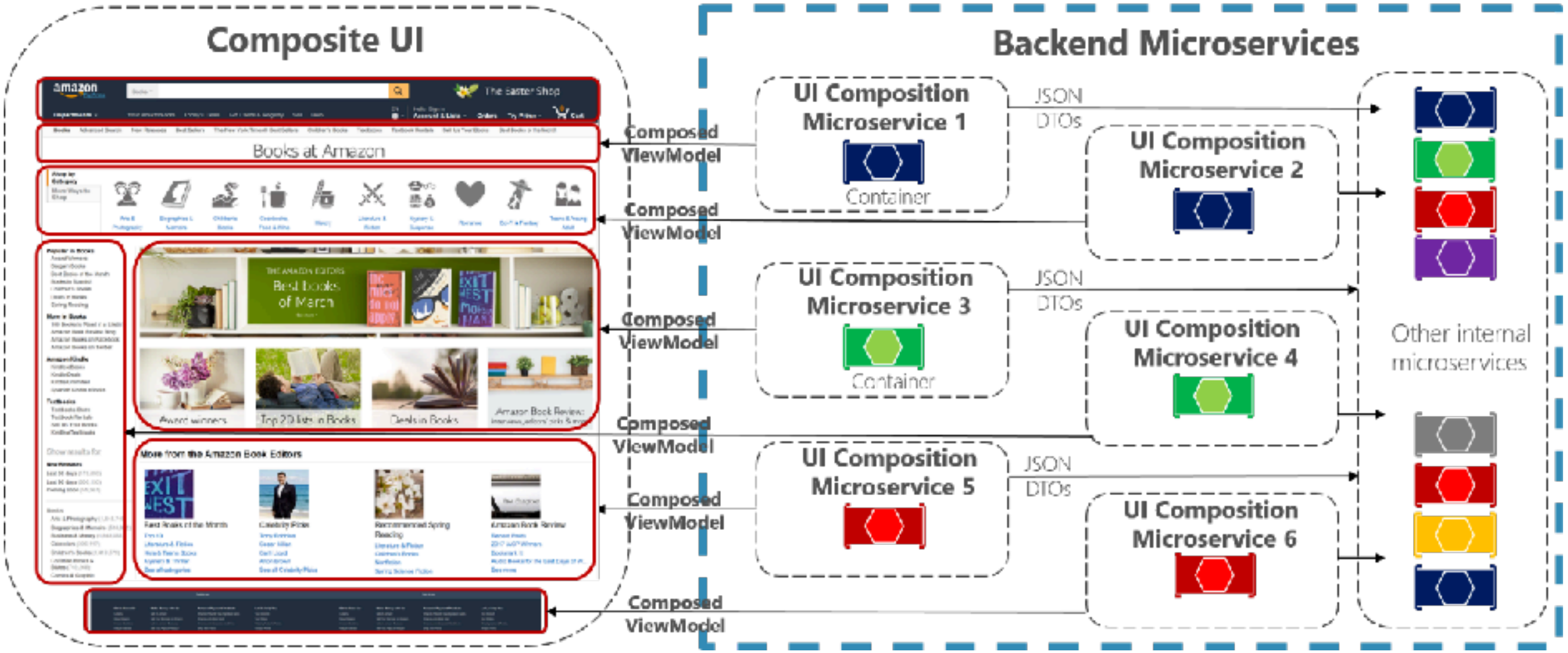
Integrate services with User Interface

Monolithic UI consuming microservices



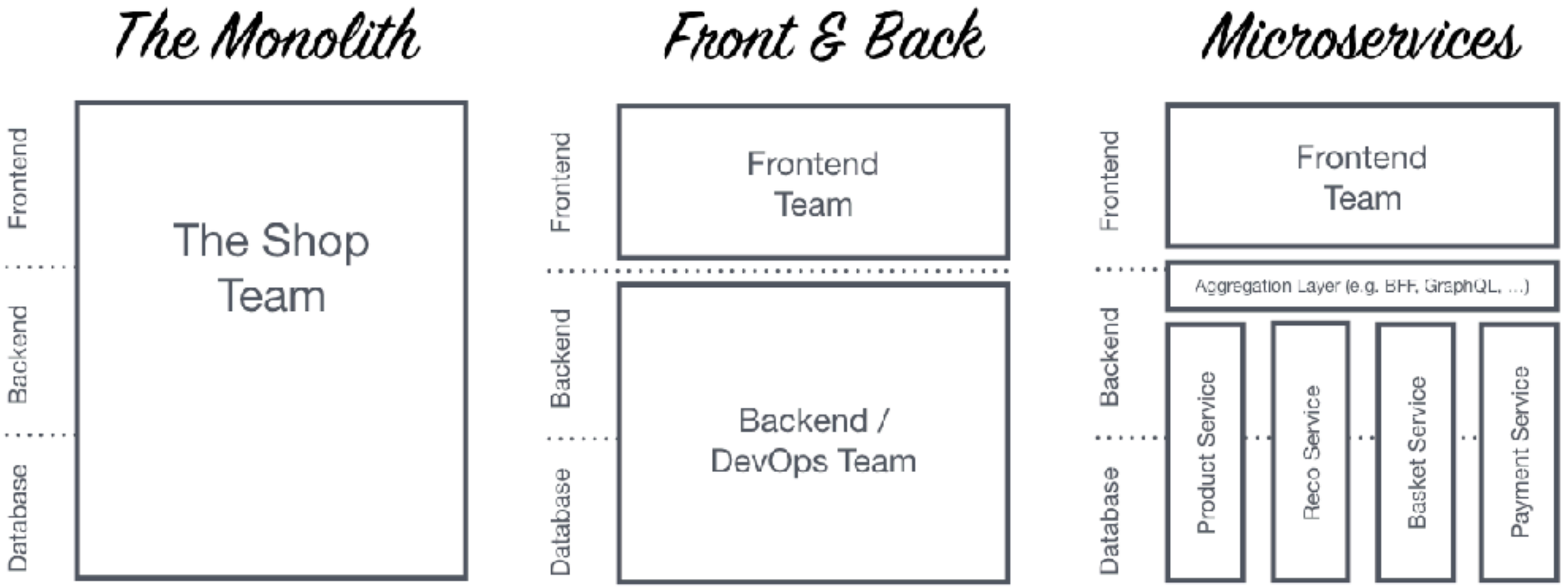
https://docs.microsoft.com

Composite UI *generated* by microservices



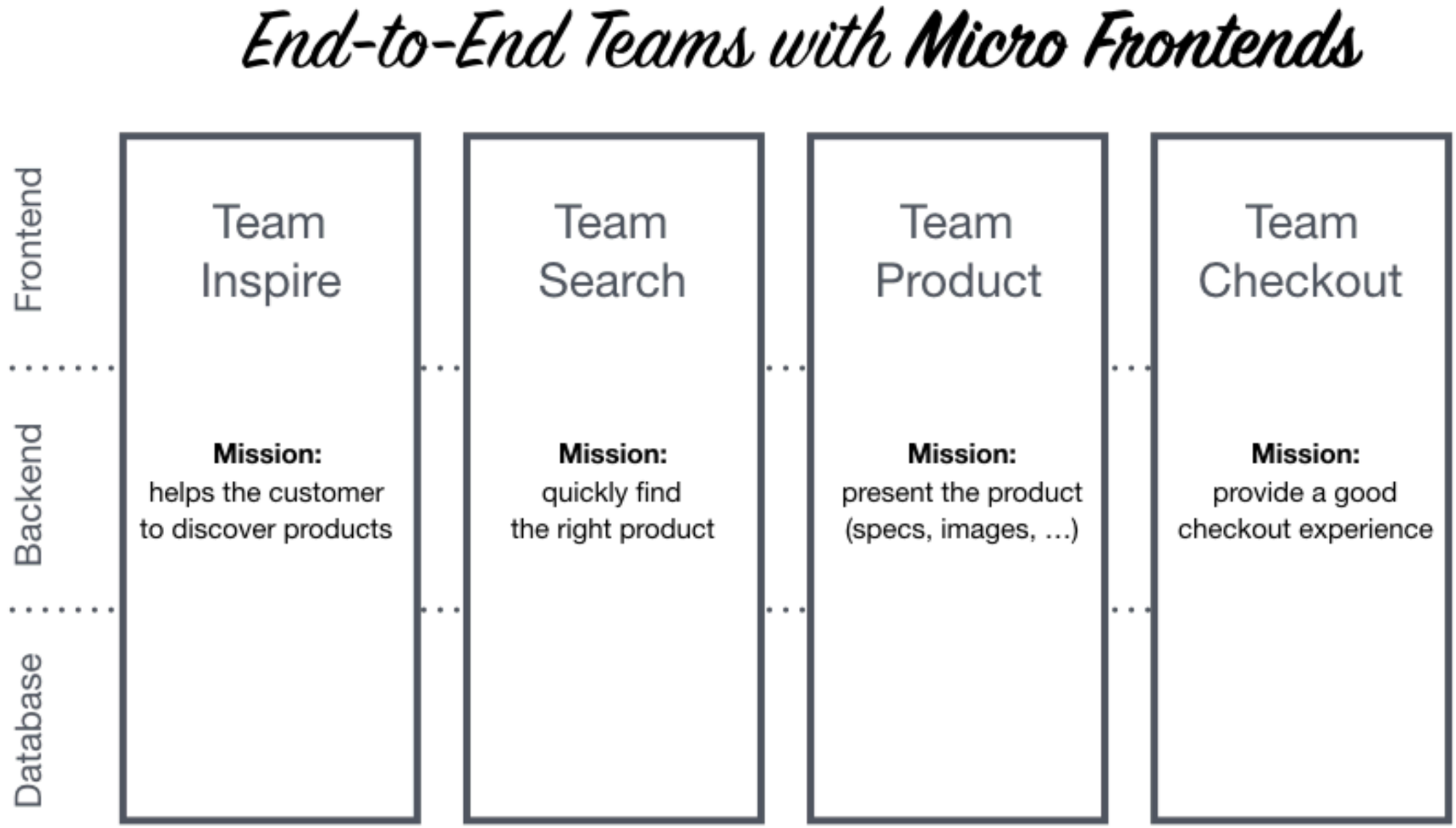
https://docs.microsoft.com

Monolith frontend



https://micro-frontends.org/

Micro frontend



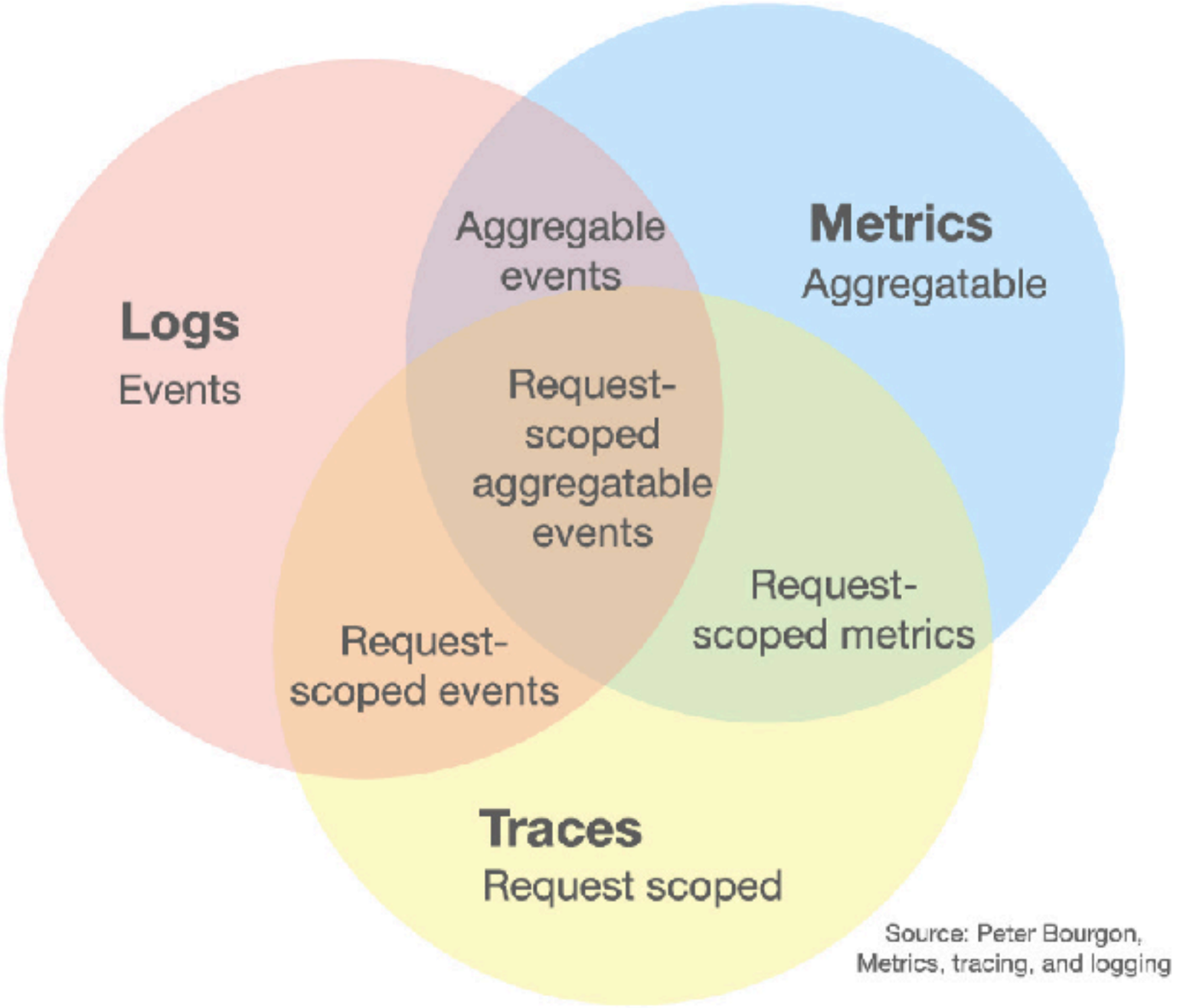
https://micro-frontends.org/

Observability of services

Observability of services

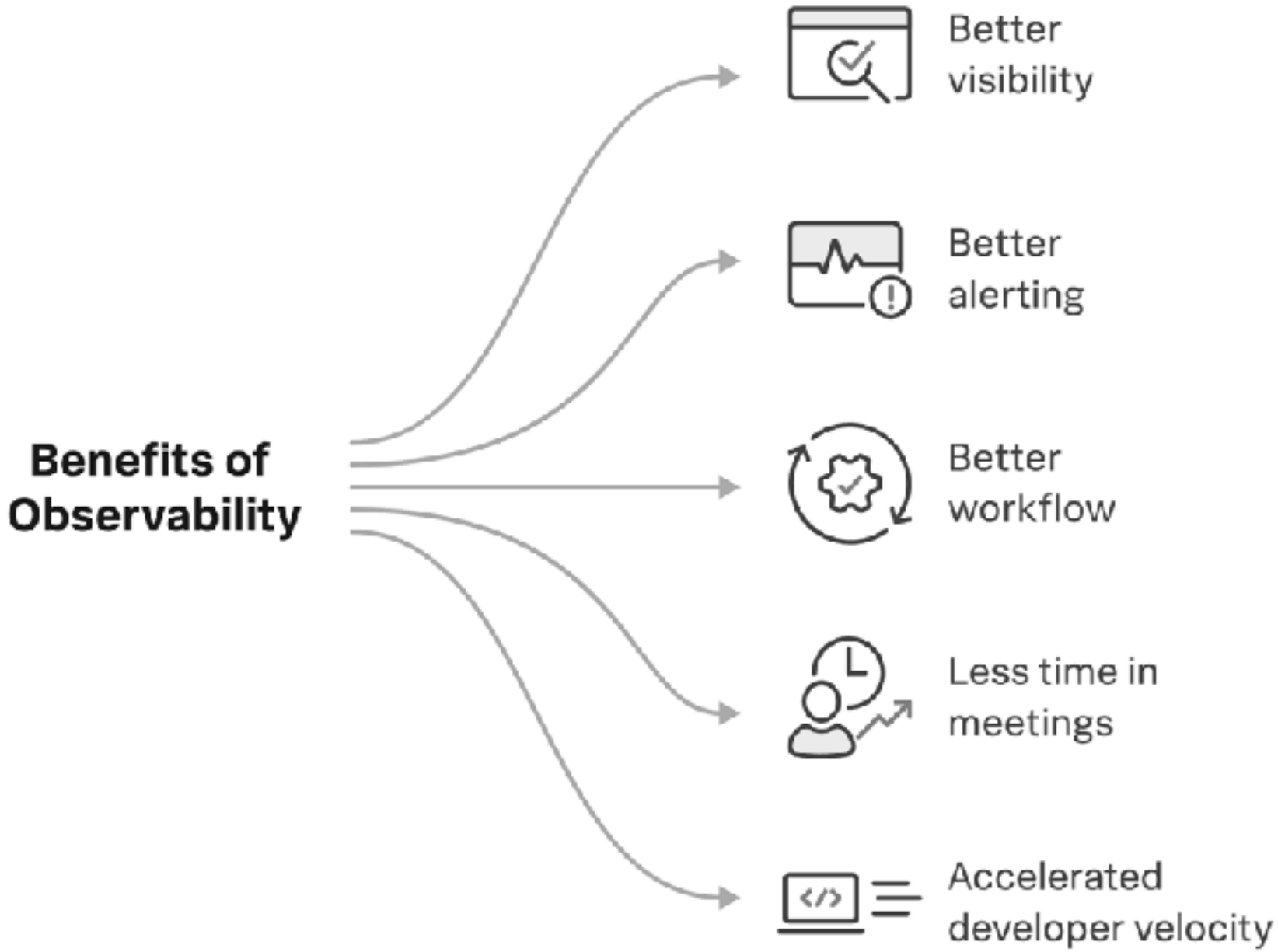
Application metrics
Distributed tracing
Log aggregation
Alert system
Exception tracking

Observability of services



Source: Peter Bourgon, Metrics, tracing, and logging

Observability of services

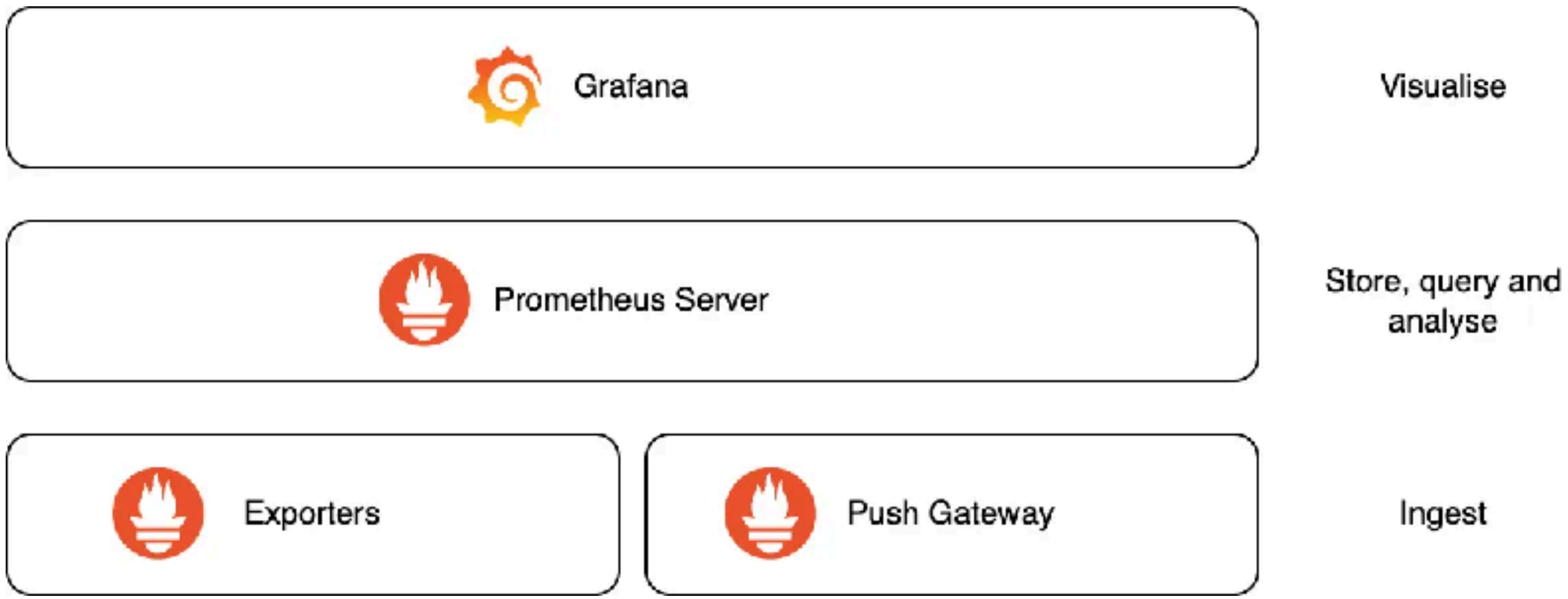


Application metrics

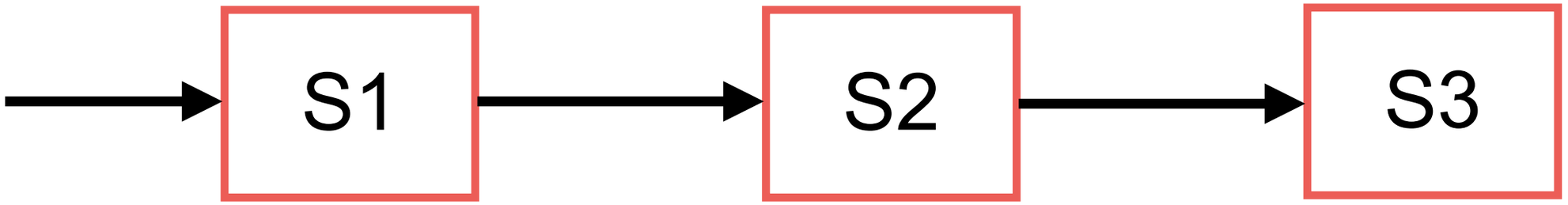
Metrics represent logs **numerically** over intervals of time

Request rate	Error rate	Health
Latency	Utilization	Availability

Application metrics



Distributed tracing



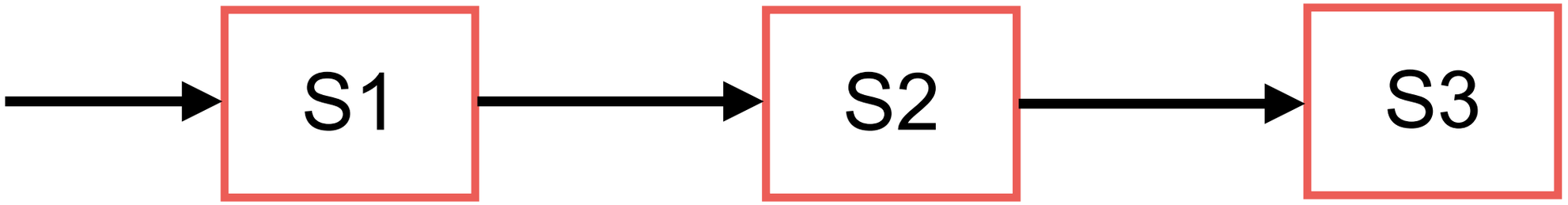
Response time

Time ?

Time ?

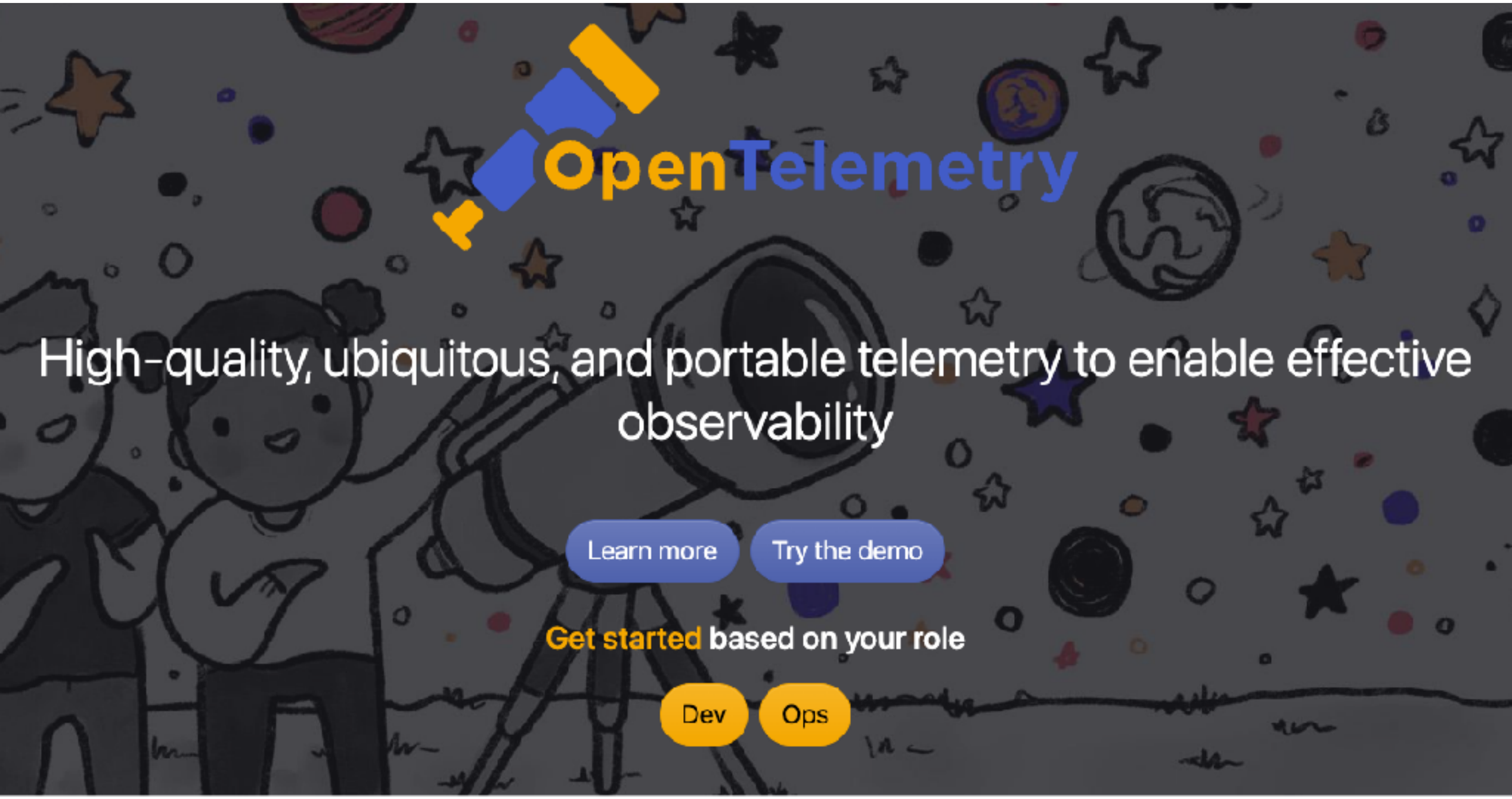
Time ?

Correlation IDs

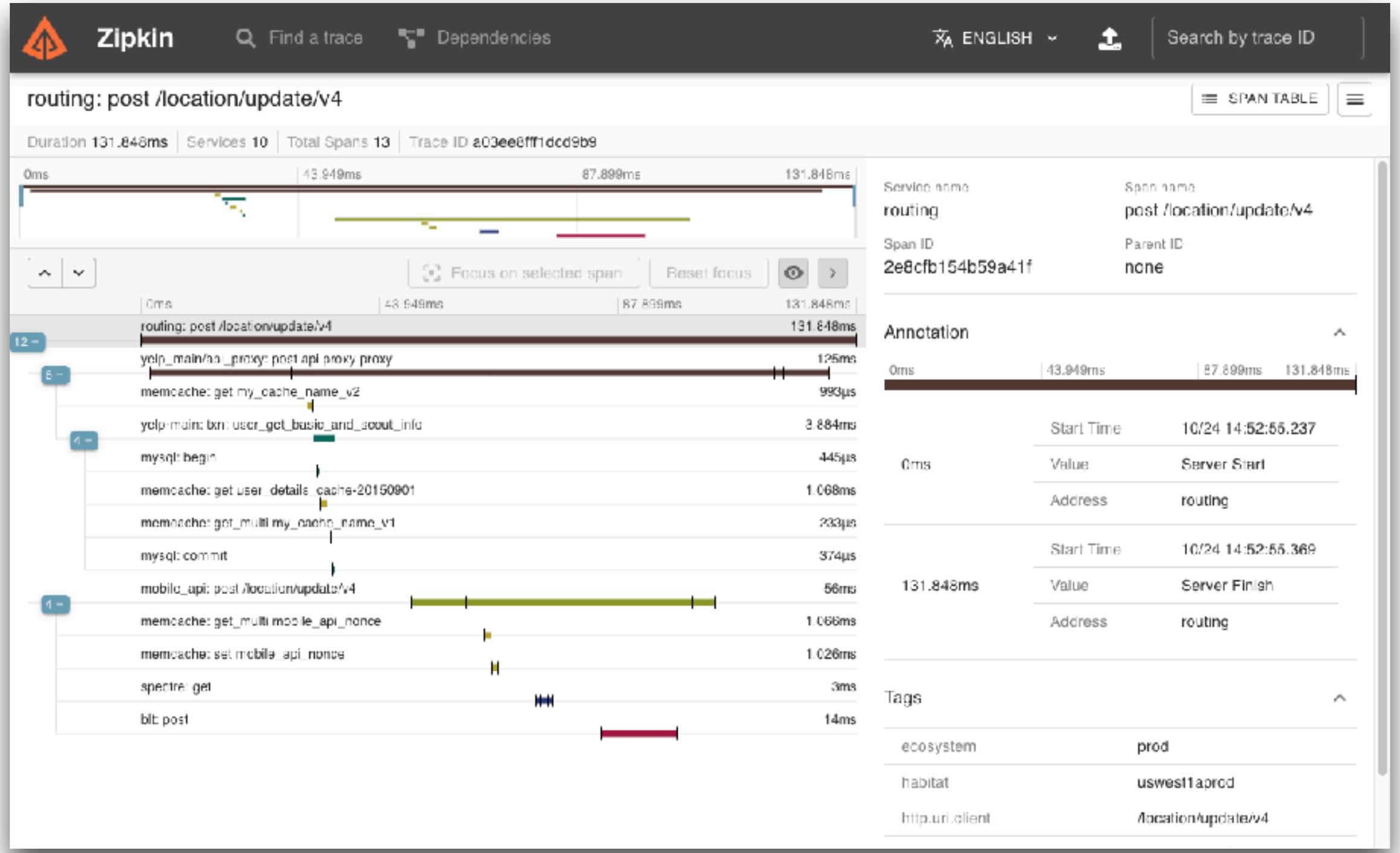


Response time **id=111**

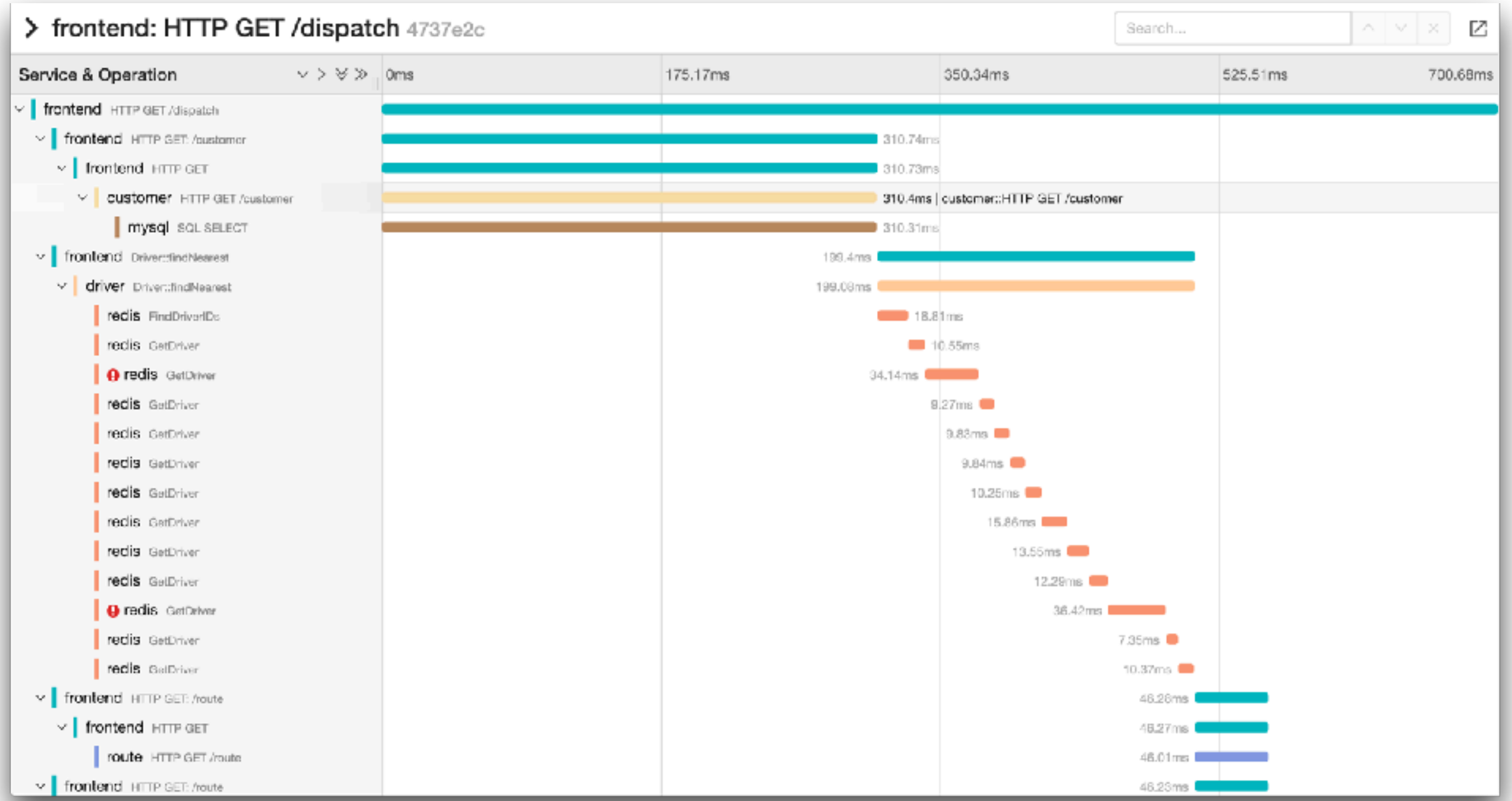
Time ?	Time ?	Time ?
id=222 **parent_id=111**	id=333 **parent_id=111**	id=444 **parent_id=111**



https://opentelemetry.io/

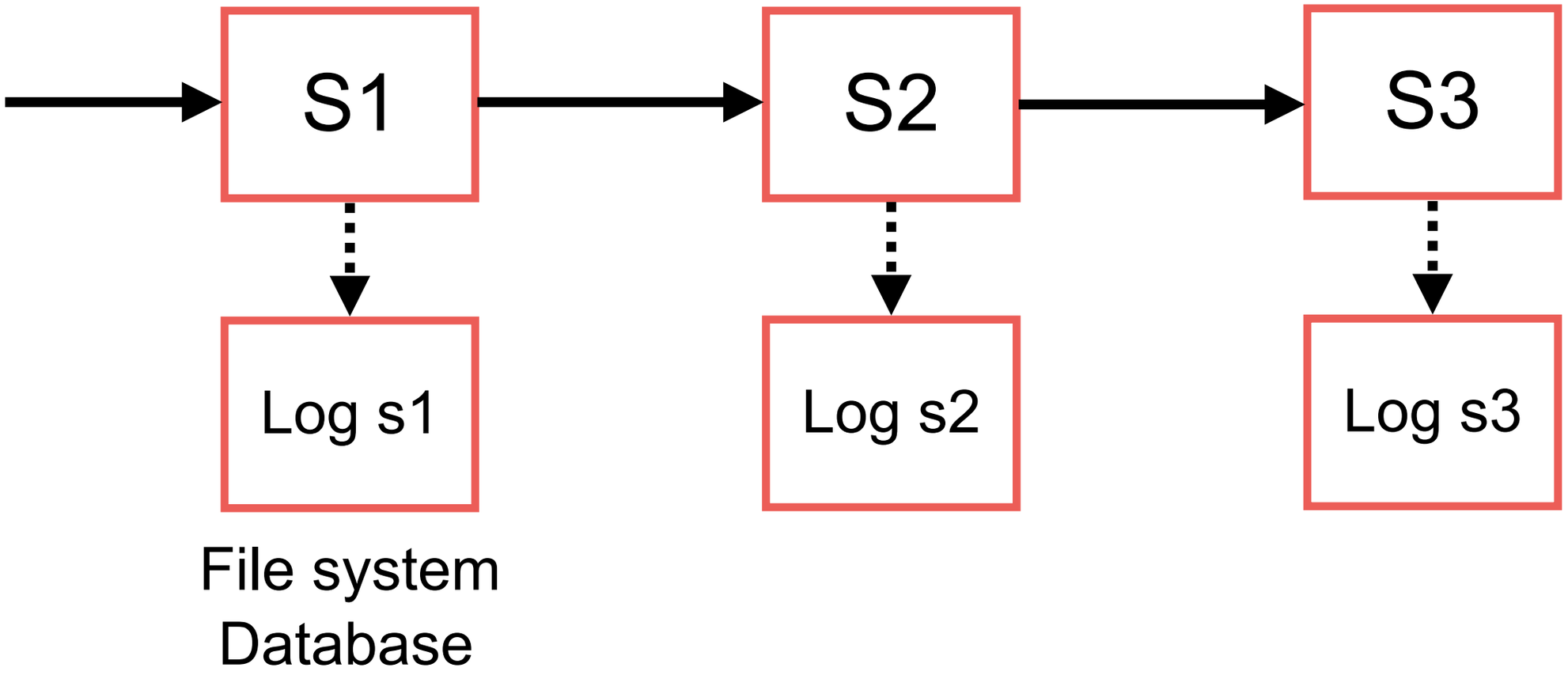


https://zipkin.io/



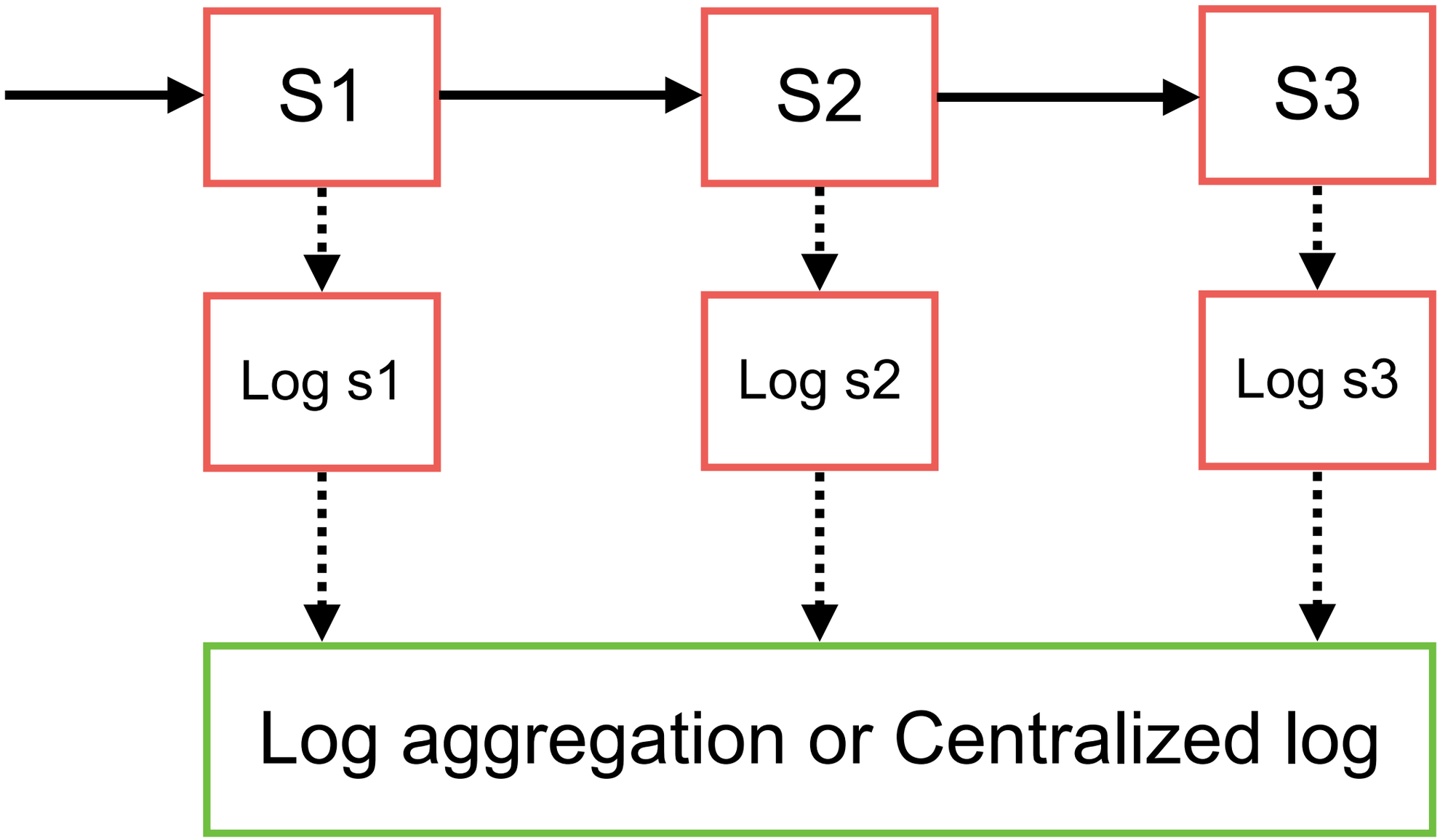
https://www.jaegertracing.io/

Log aggregation

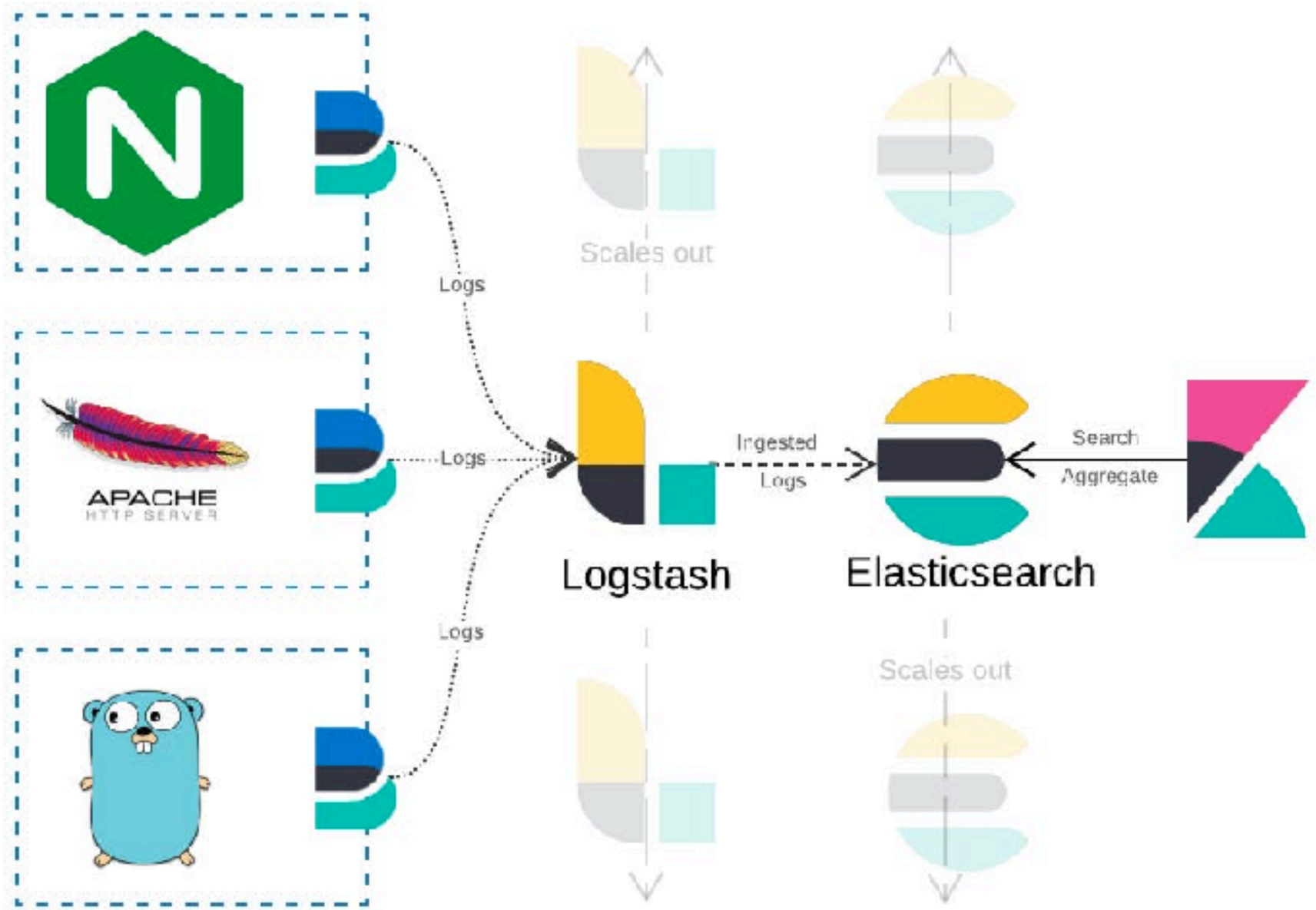


How to find errors from log ?

Log aggregation + Correlation IDs



ELK stack



More ...

How to develop ?
How to test ?
How to deploy ?

Summary

Beyond Microservice
Process and **Organization**

Flexible organizational structure

With clear roles and accountabilities

Efficient meeting formats

Geared toward action and eliminating over-analysis

More autonomy to teams and individuals

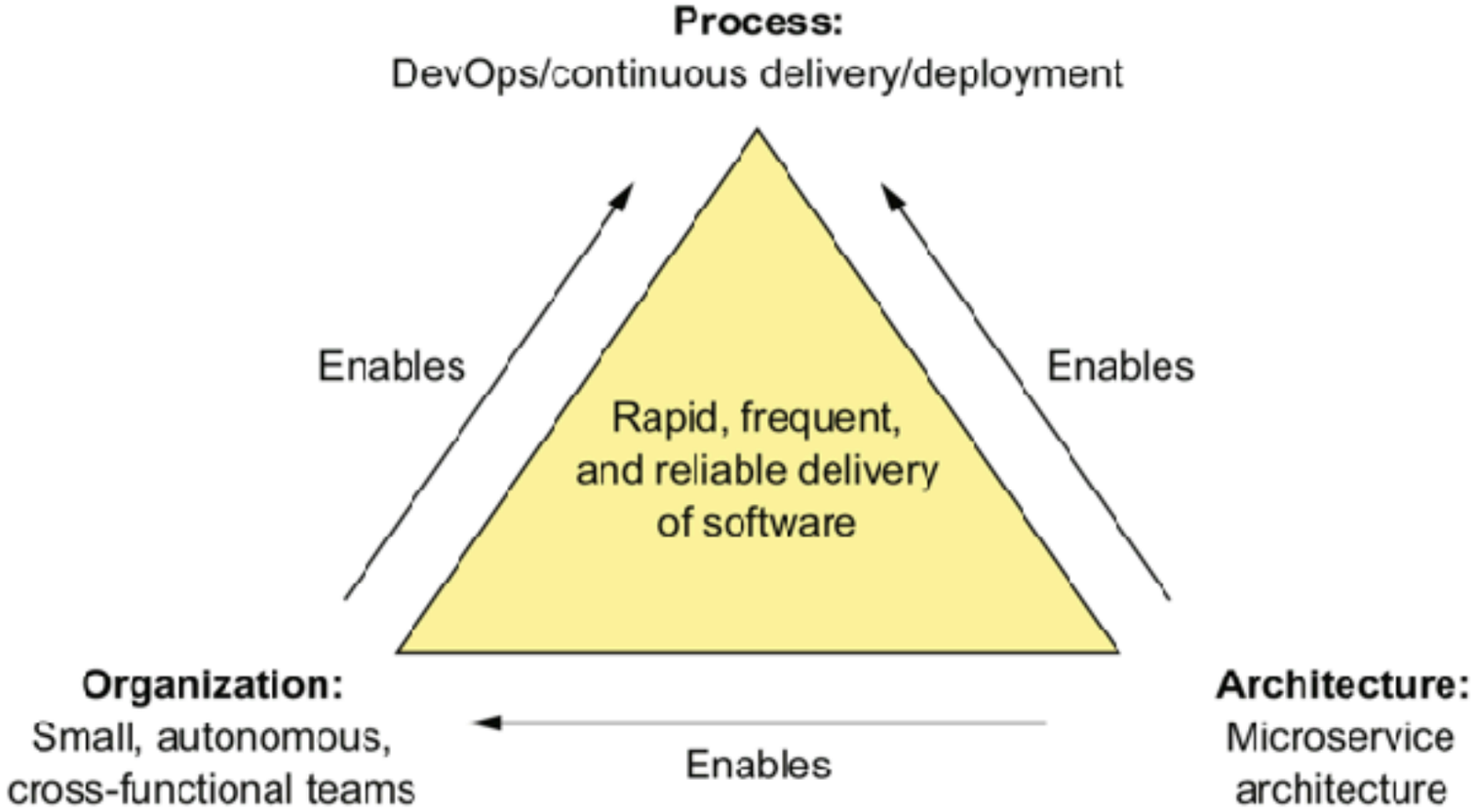
Individuals solve issues directly without bureaucracy

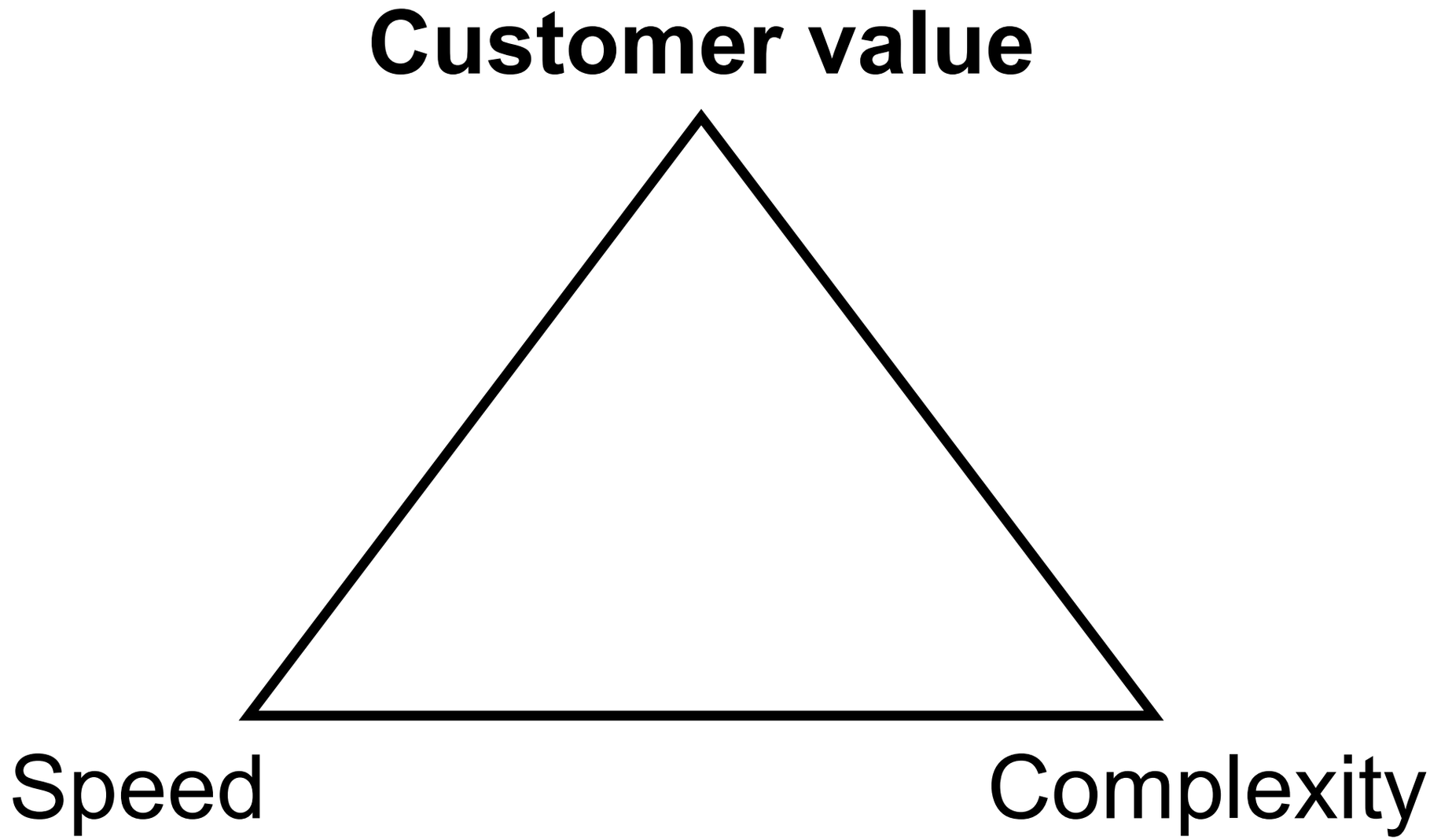
Unique decision-making process

To continuously evolve the organization's structure.

Success project needs …

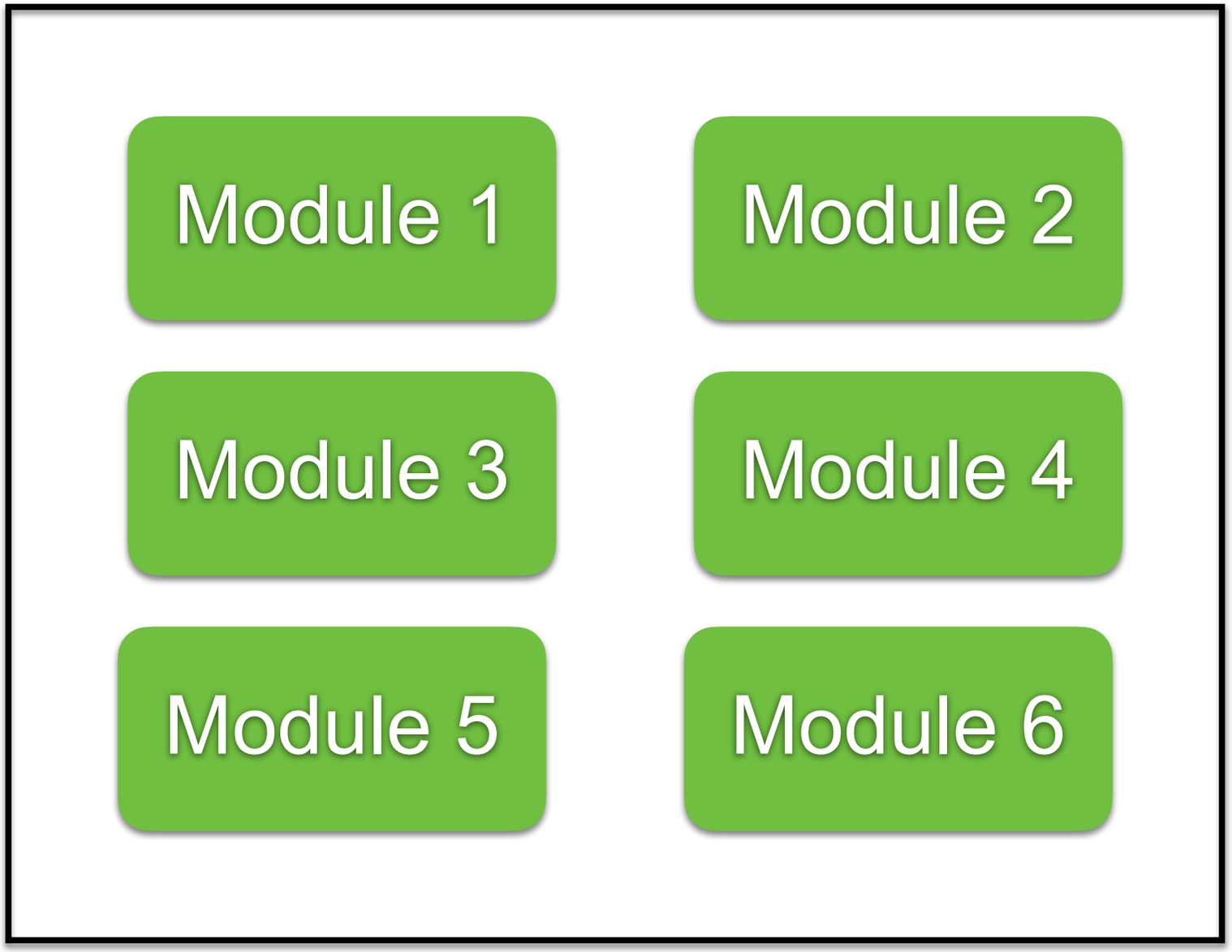
Organization process
Development process
Delivery process



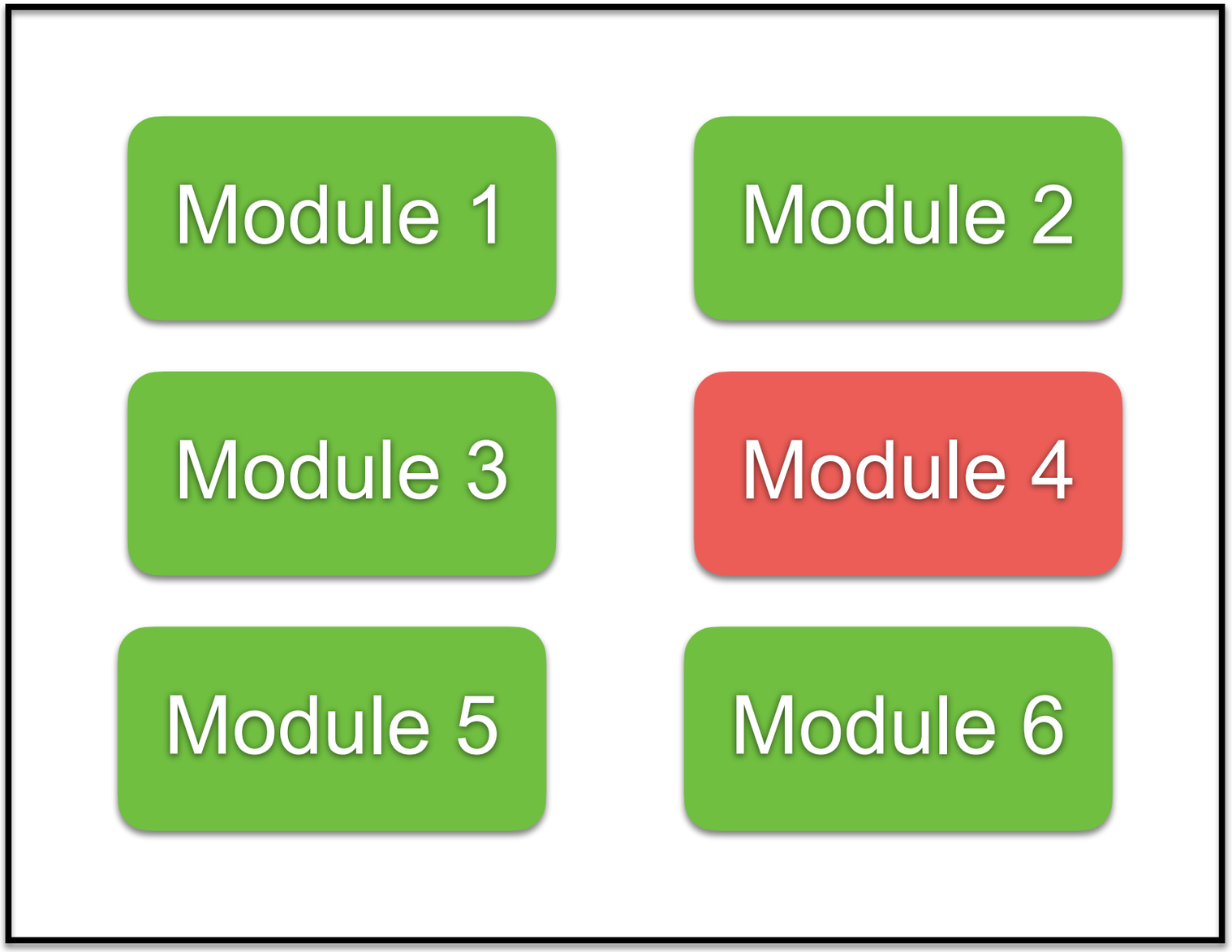


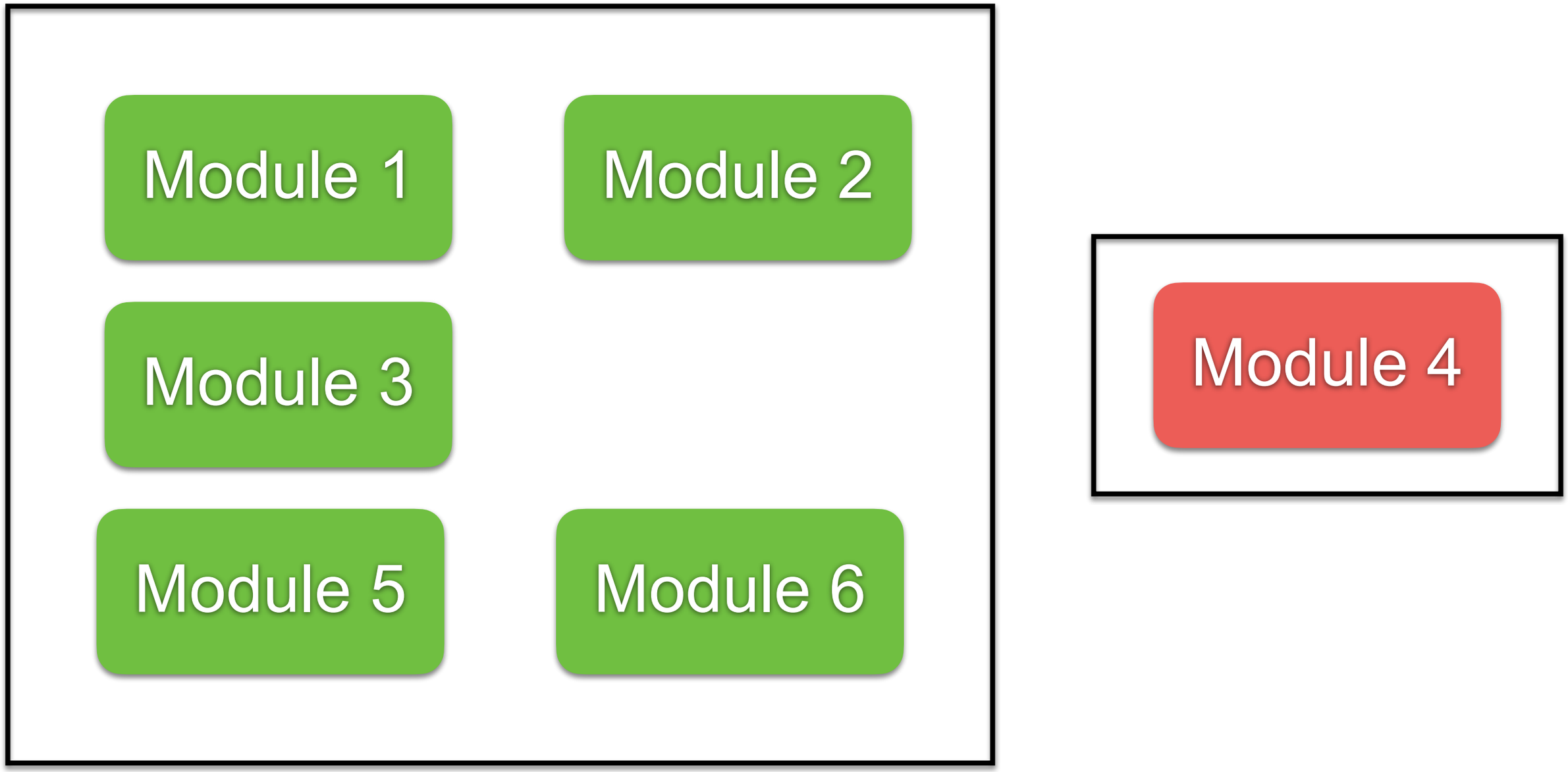
Don’t forget about the **human** side when adopt Microservice

Let’s start with good monolith

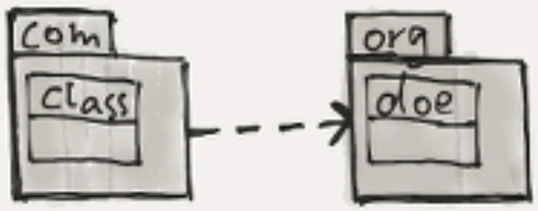


Find your problem

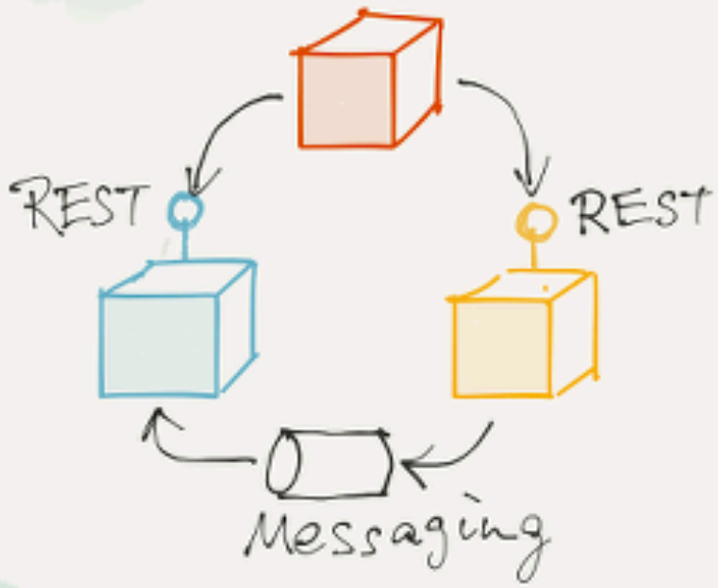




Architecture



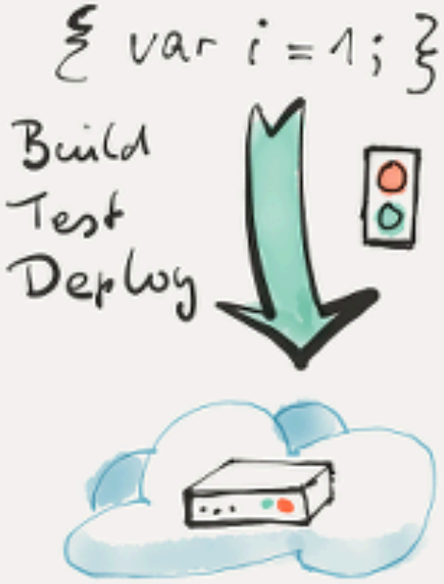
Microservices



People & Teams

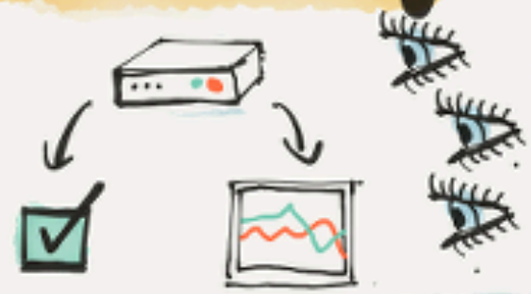
Deployment

Continous Delivery



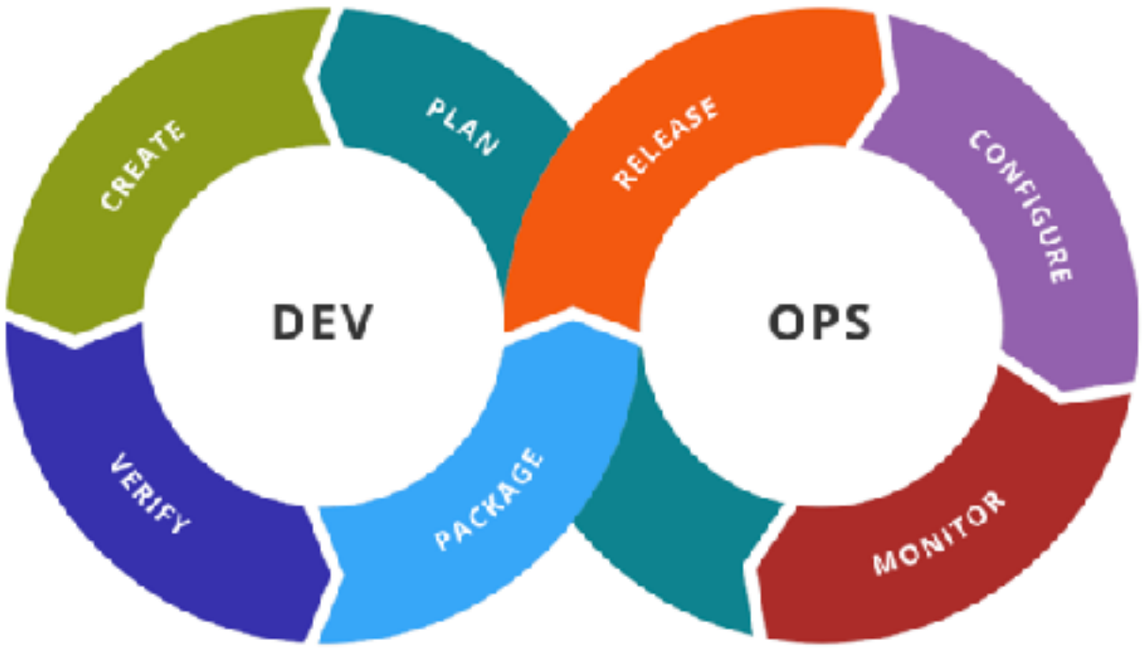
Infrastructure

Monitoring



Microservice prerequisites

Rapid provisioning
Basic monitoring
Rapid Application Deployment
DevOps culture



Let’s start your journey !!