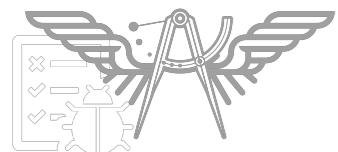


Evolution of Architecture





Somkiat Puisungnoen

Somkiat Puisungnoen

Update Info 1

View Activity Log 10+

...
Timeline About Friends 3,138 Photos More

When did you work at Opendream?
... 22 Pending Items

Post Photo/Video Live Video Life Event

What's on your mind?

Public Post

Intro
Software Craftsmanship

Software Practitioner at สยามชัมนาภิกิจ พ.ศ. 2556

Agile Practitioner and Technical at SPRINT3r

Somkiat Puisungnoen 15 mins · Bangkok · ⚙️

Java and Bigdata





Page

Messages

Notifications 3

Insights

Publishing Tools

Settings

Help ▾



somkiat.cc

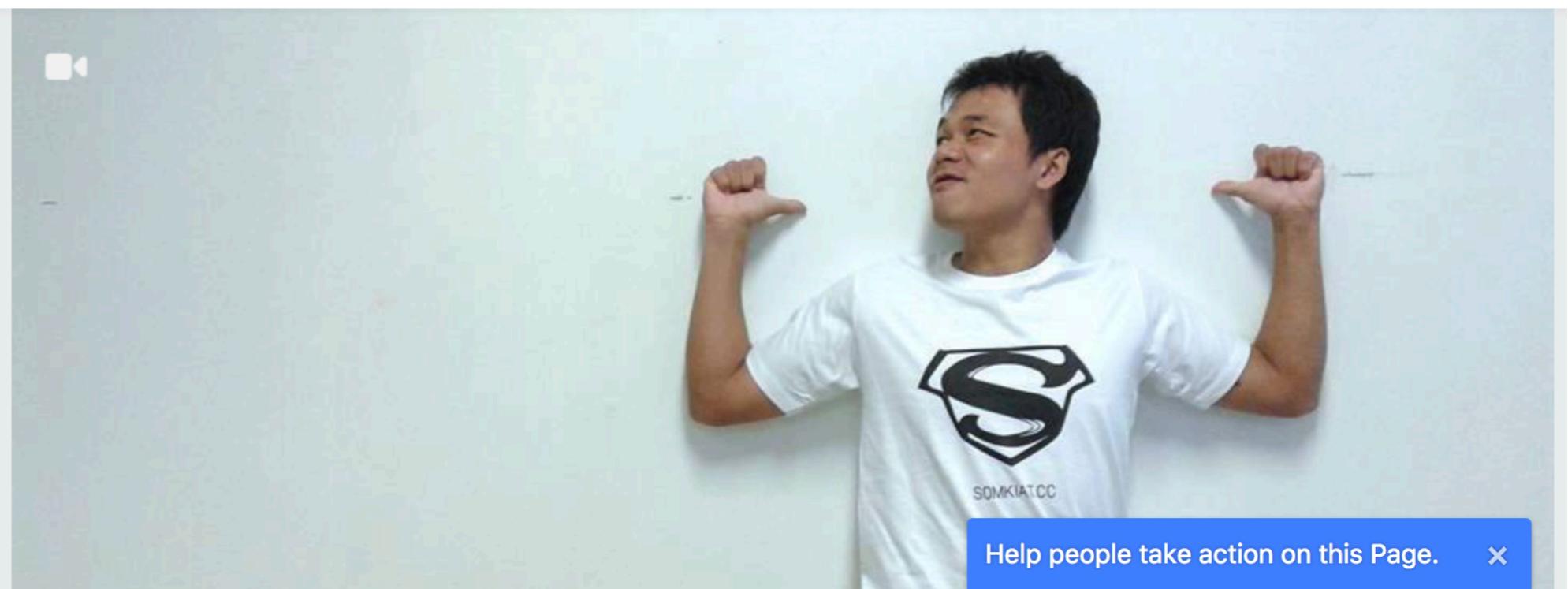
@somkiat.cc

Home

Posts

Videos

Photos



Help people take action on this Page. 

+ Add a Button



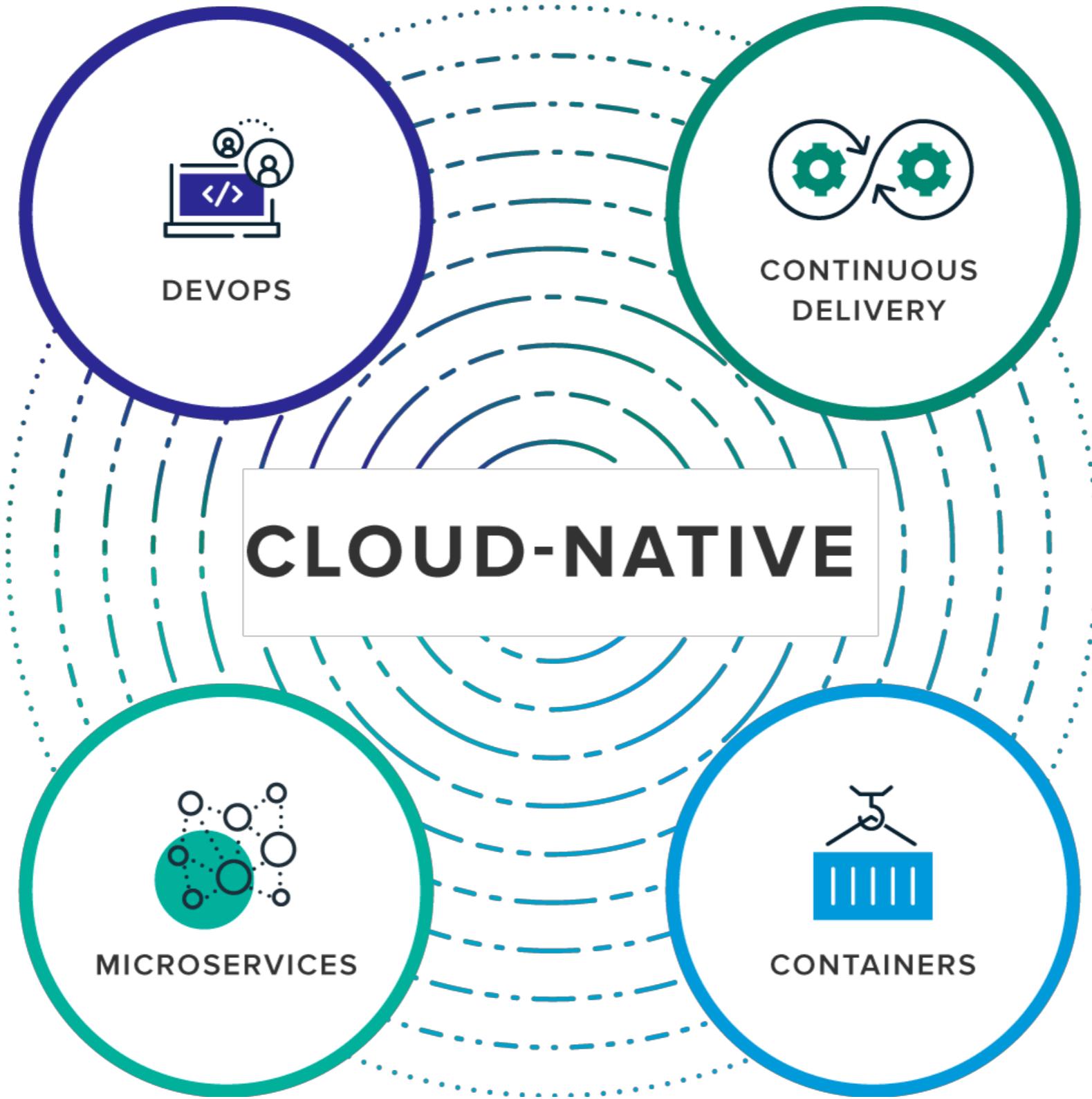
Let's start



"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"

- *Melvin Conwey, 1968* -





<https://pivotal.io/cloud-native>



Evolution of Architecture

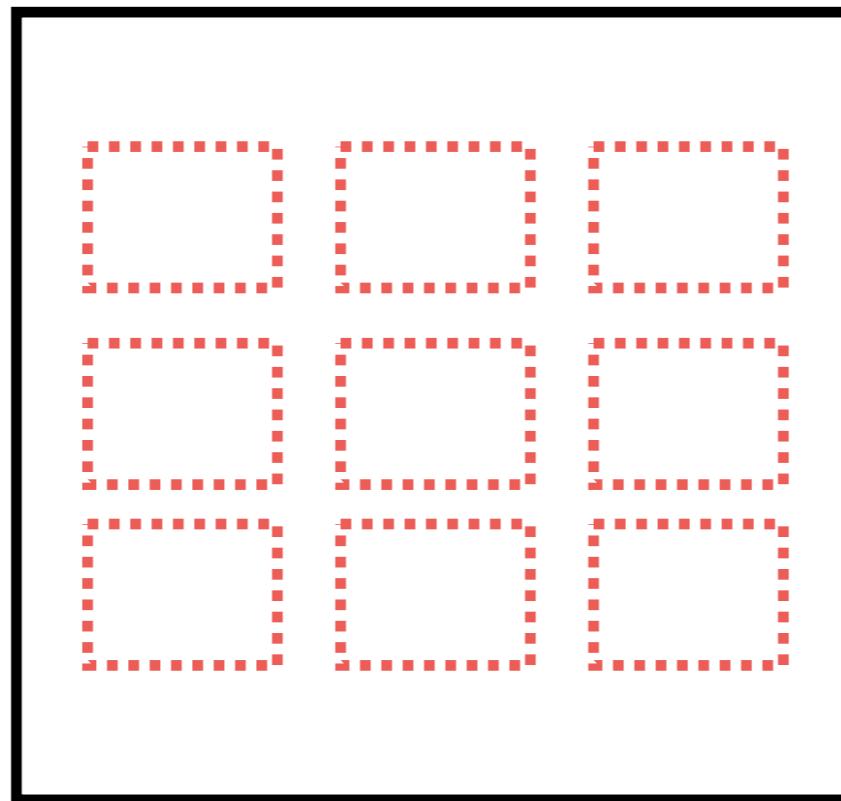


Monolith

App



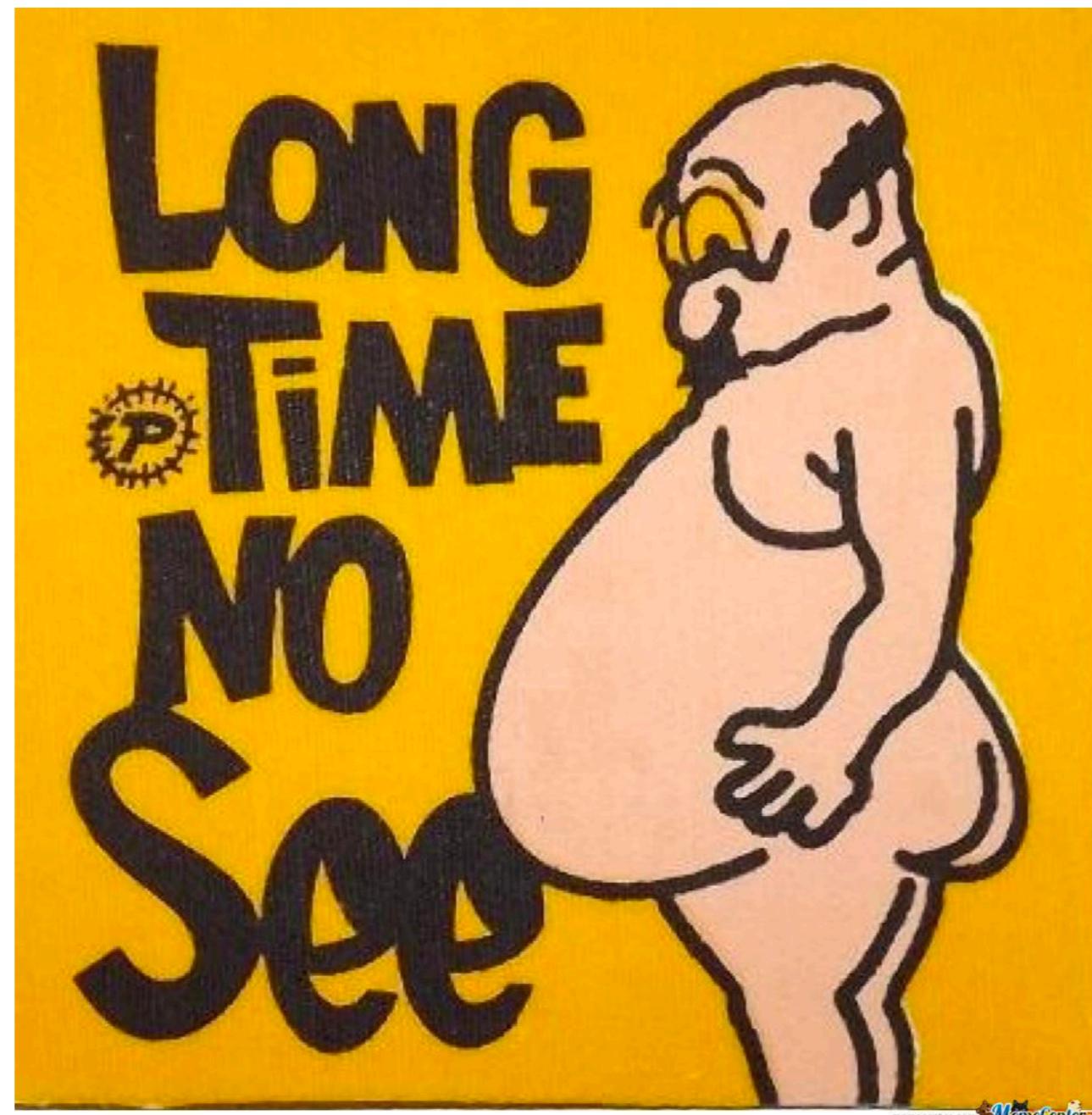
Modules



Monolith

Easy to develop
Easy to change
Easy to testing
Easy to deploy
Easy to scale





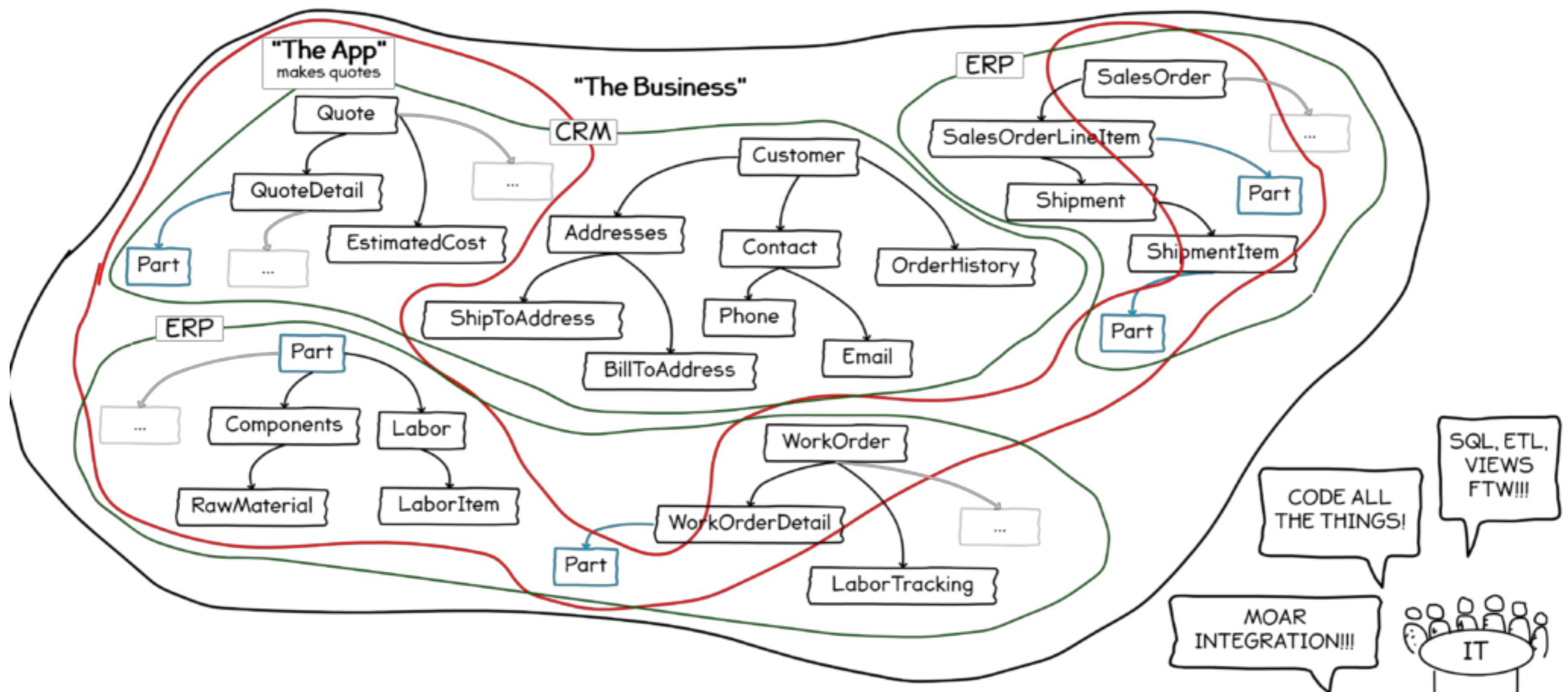
memecenter.com MemeCenter



Microservices

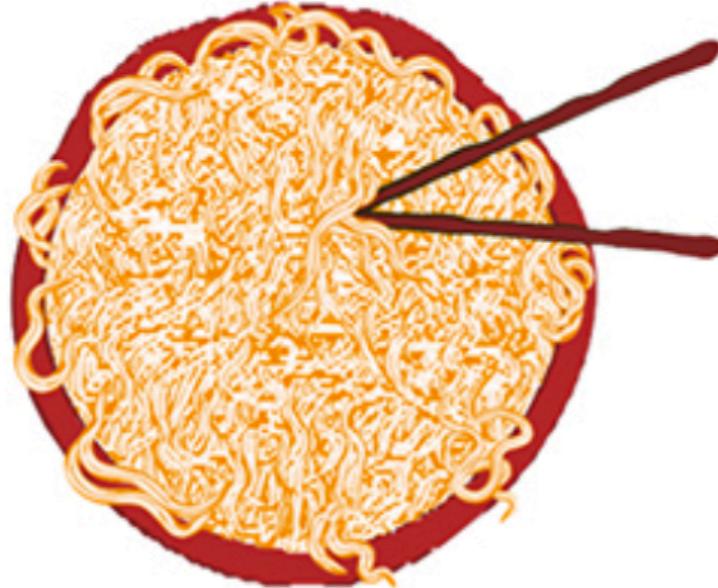
© 2017 - 2018 Siam Chamnankit Company Limited. All rights reserved.

Monolith Hell



1990s and earlier

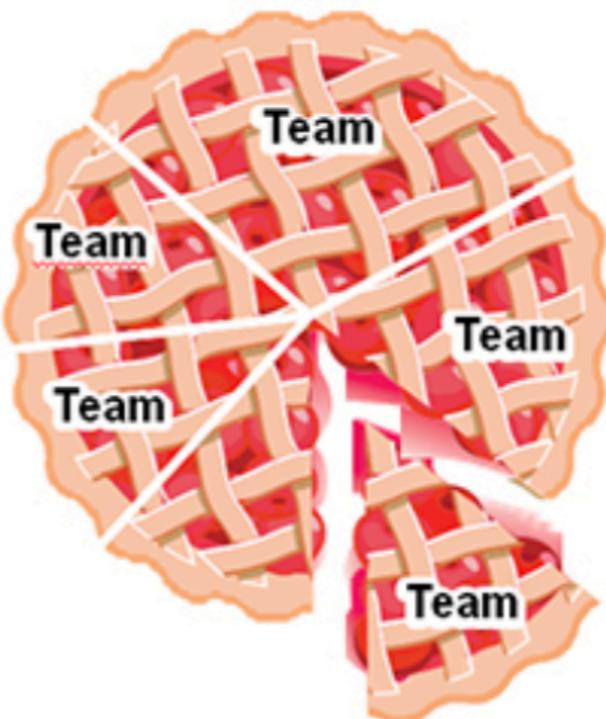
Pre-SOA (monolithic)
Tight coupling



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



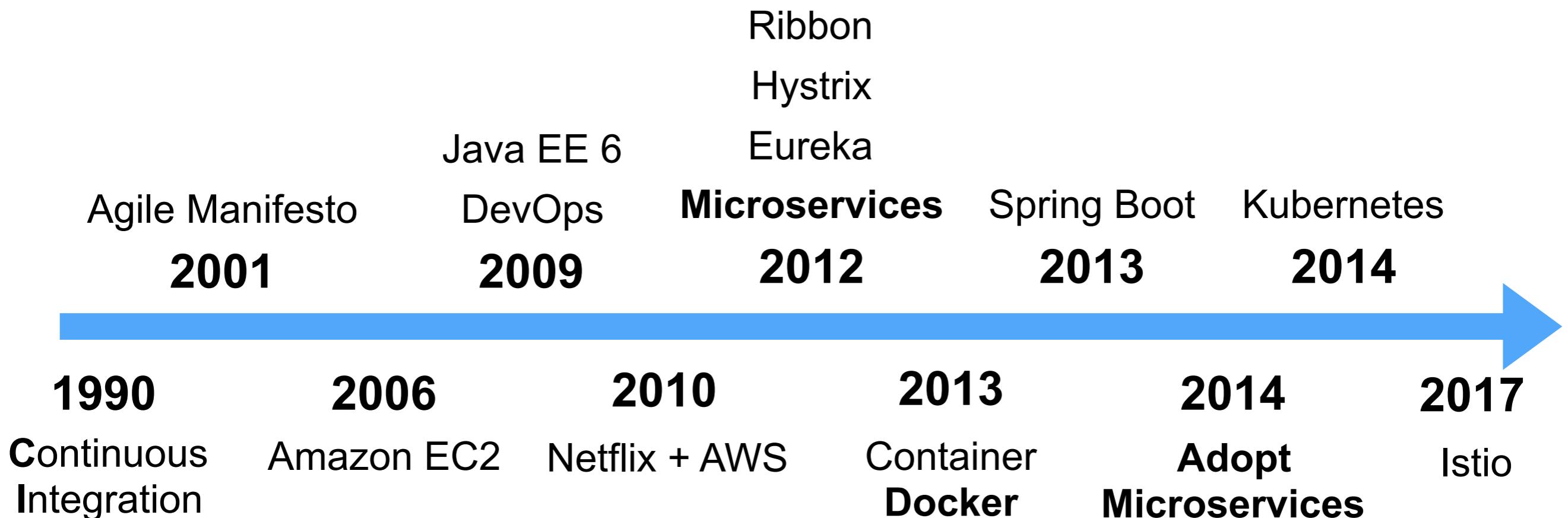
Developers can create and activate new microservices without prior coordination with others. Their adherence to MSA principles makes continuous delivery of new or modified services possible.



Microservice



Microservice history



Microservice

Small, Do one thing (Single Responsibility)

Modular

Easy to understand

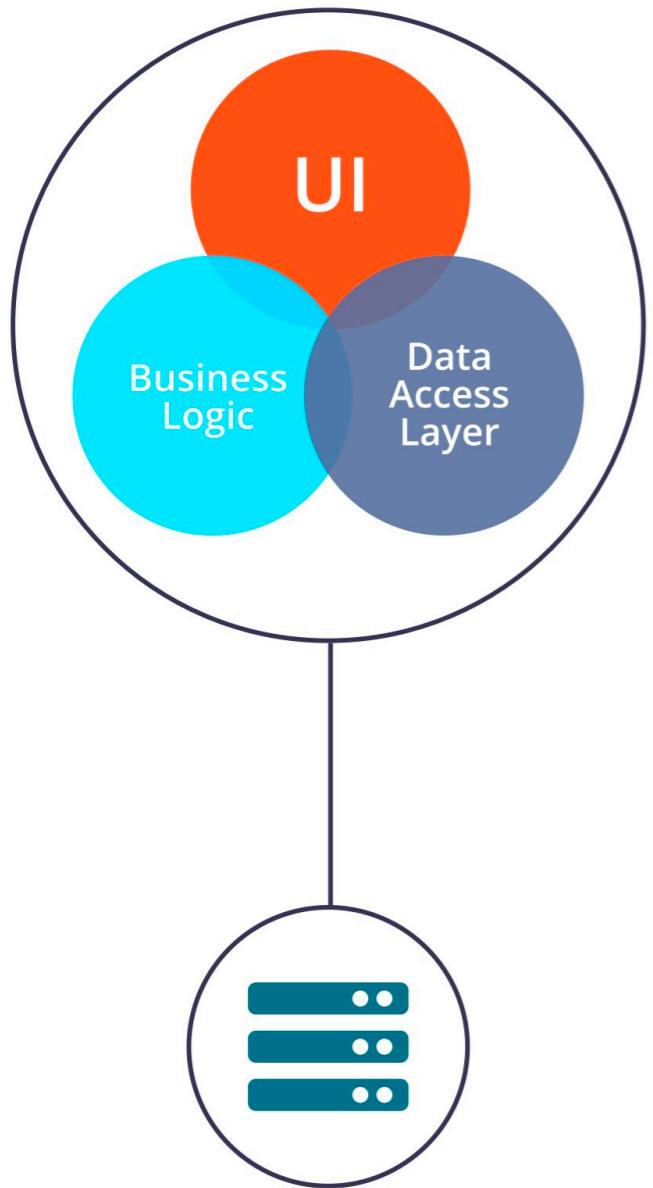
Easy to develop

Easy to deploy

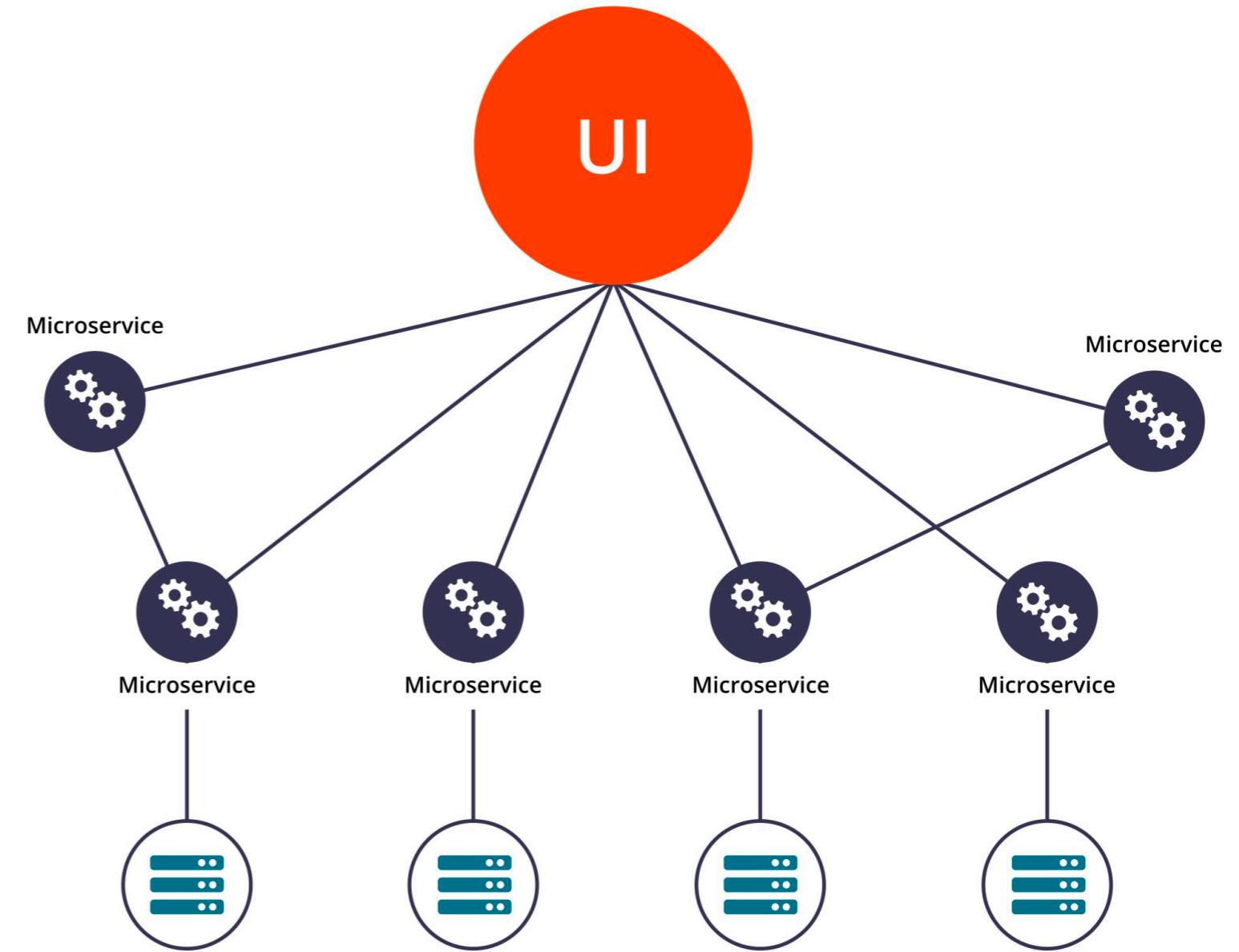
Easy to maintain

Scale independently





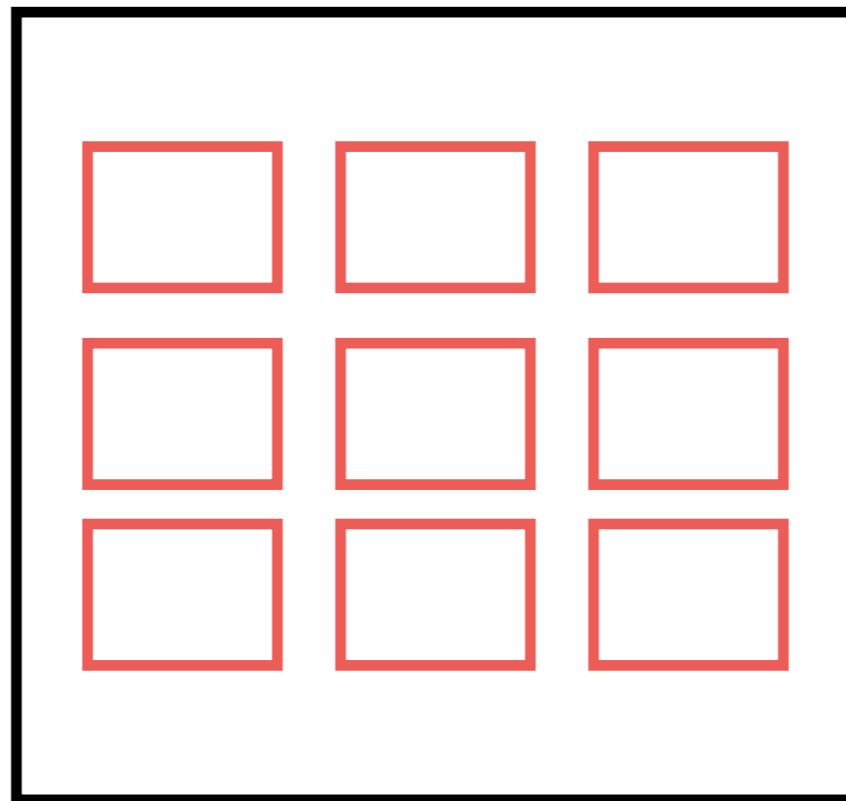
Monolithic Architecture



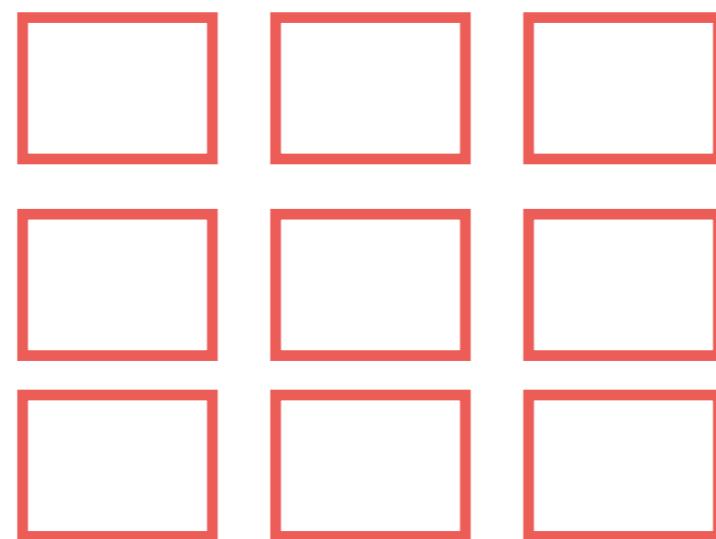
Microservice Architecture



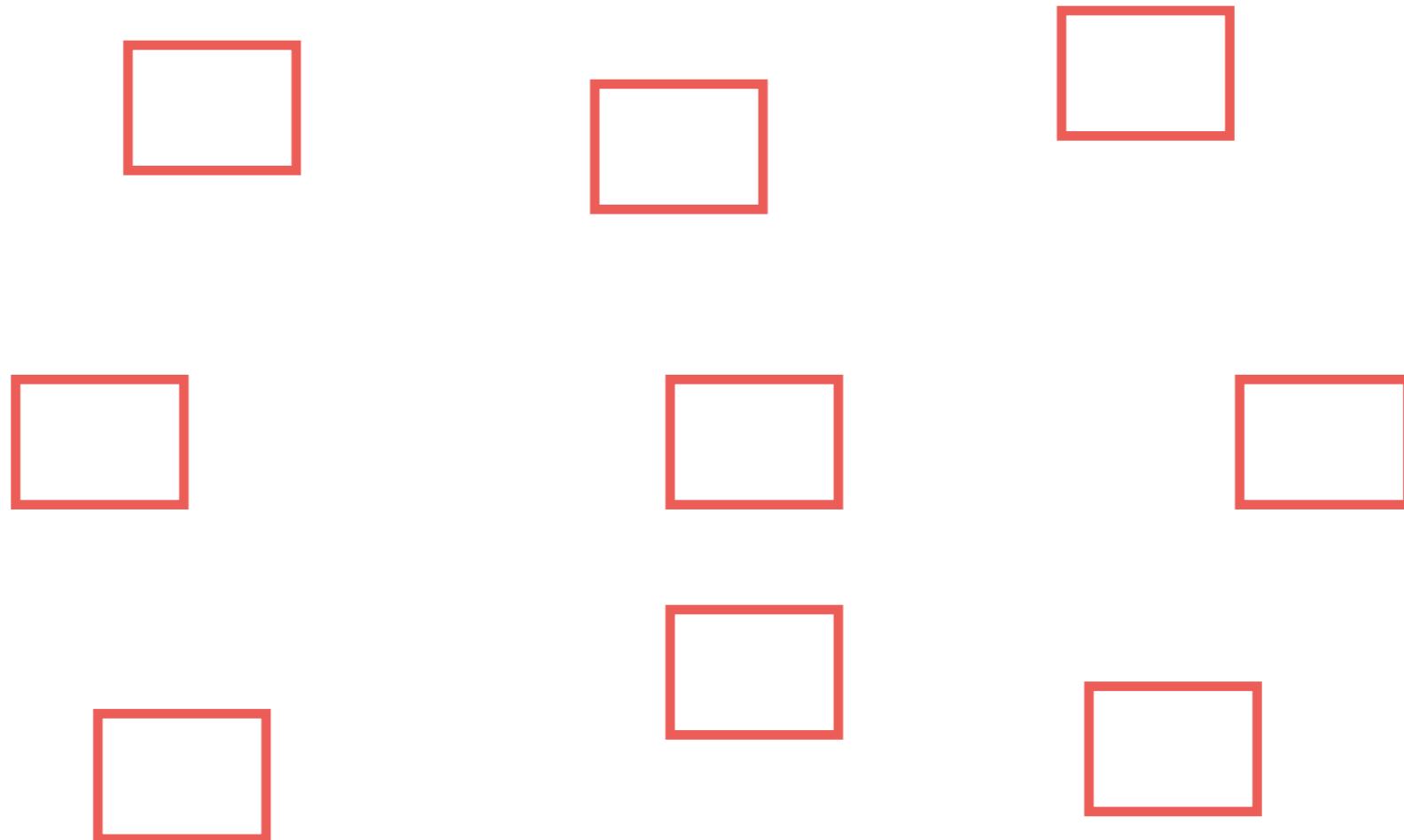
Microservice



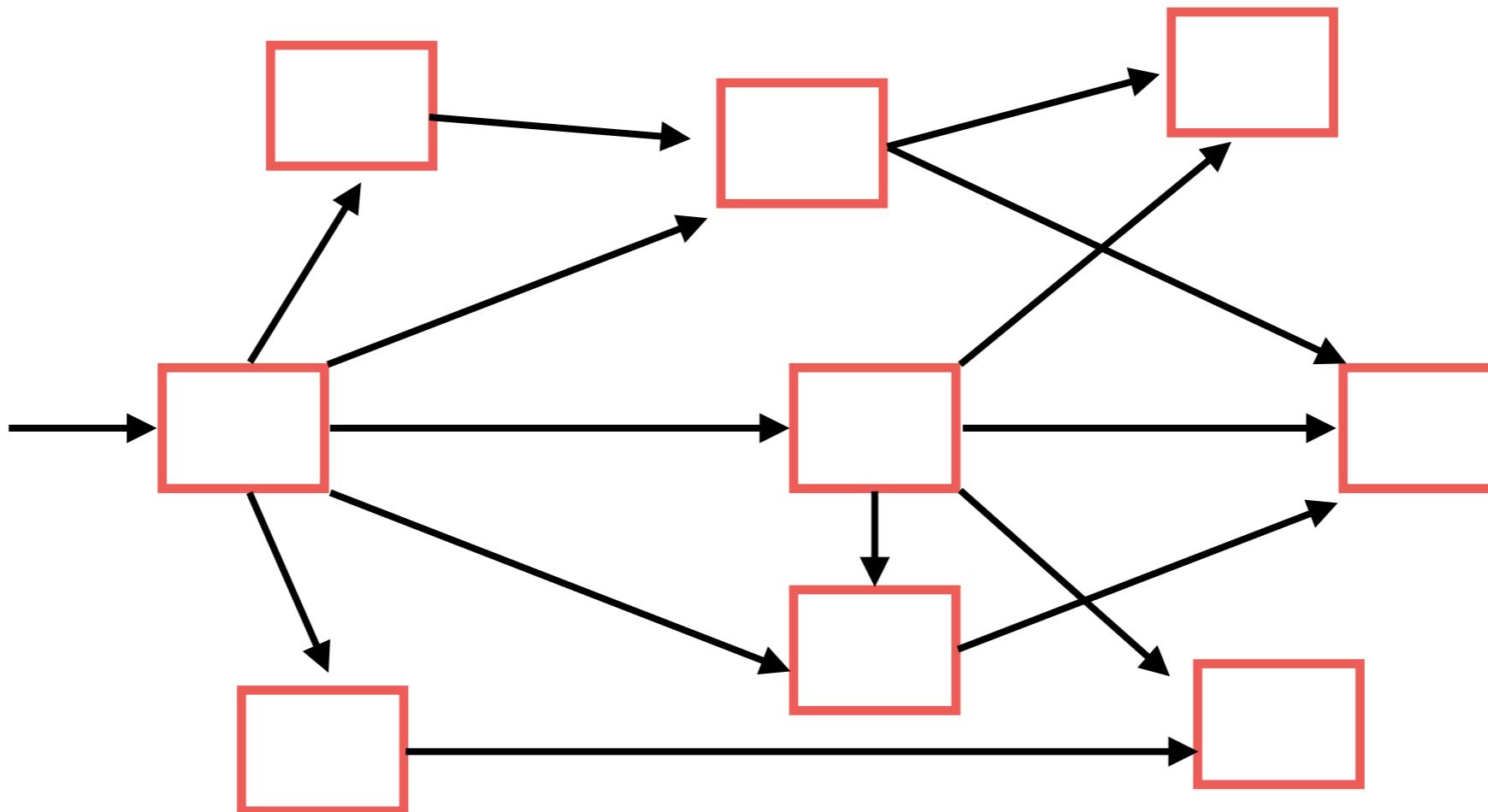
Microservice



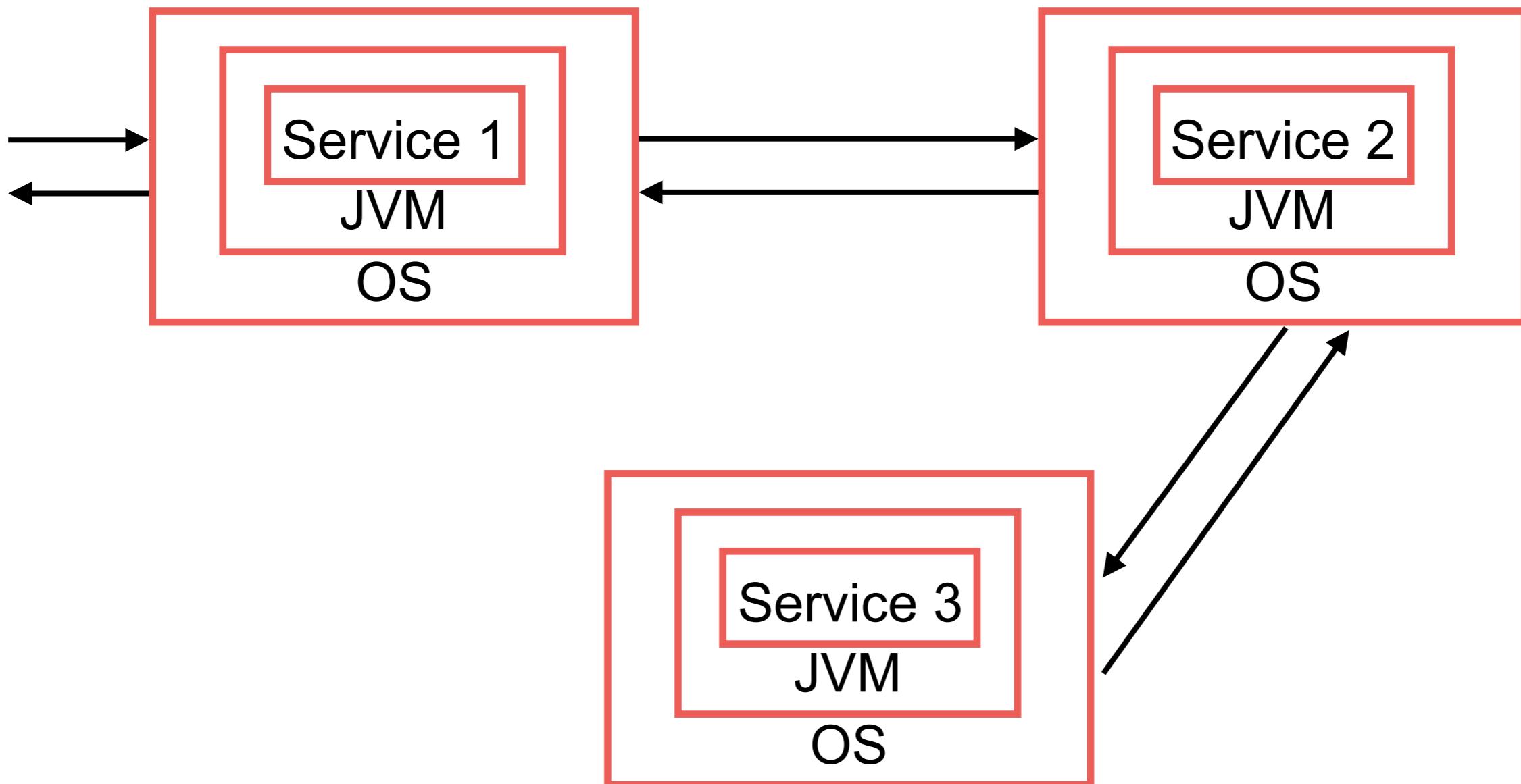
Microservice



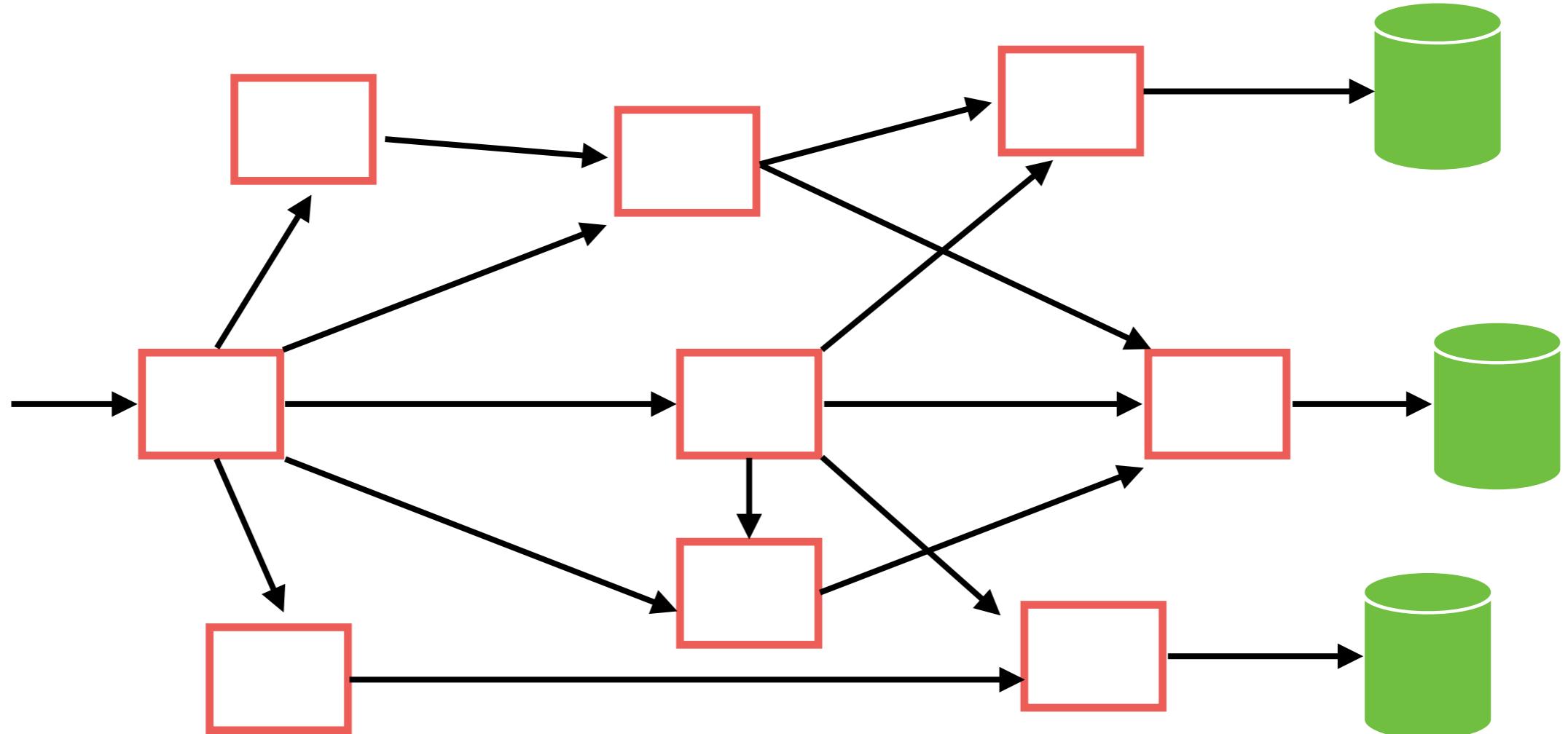
Microservice == Distributed



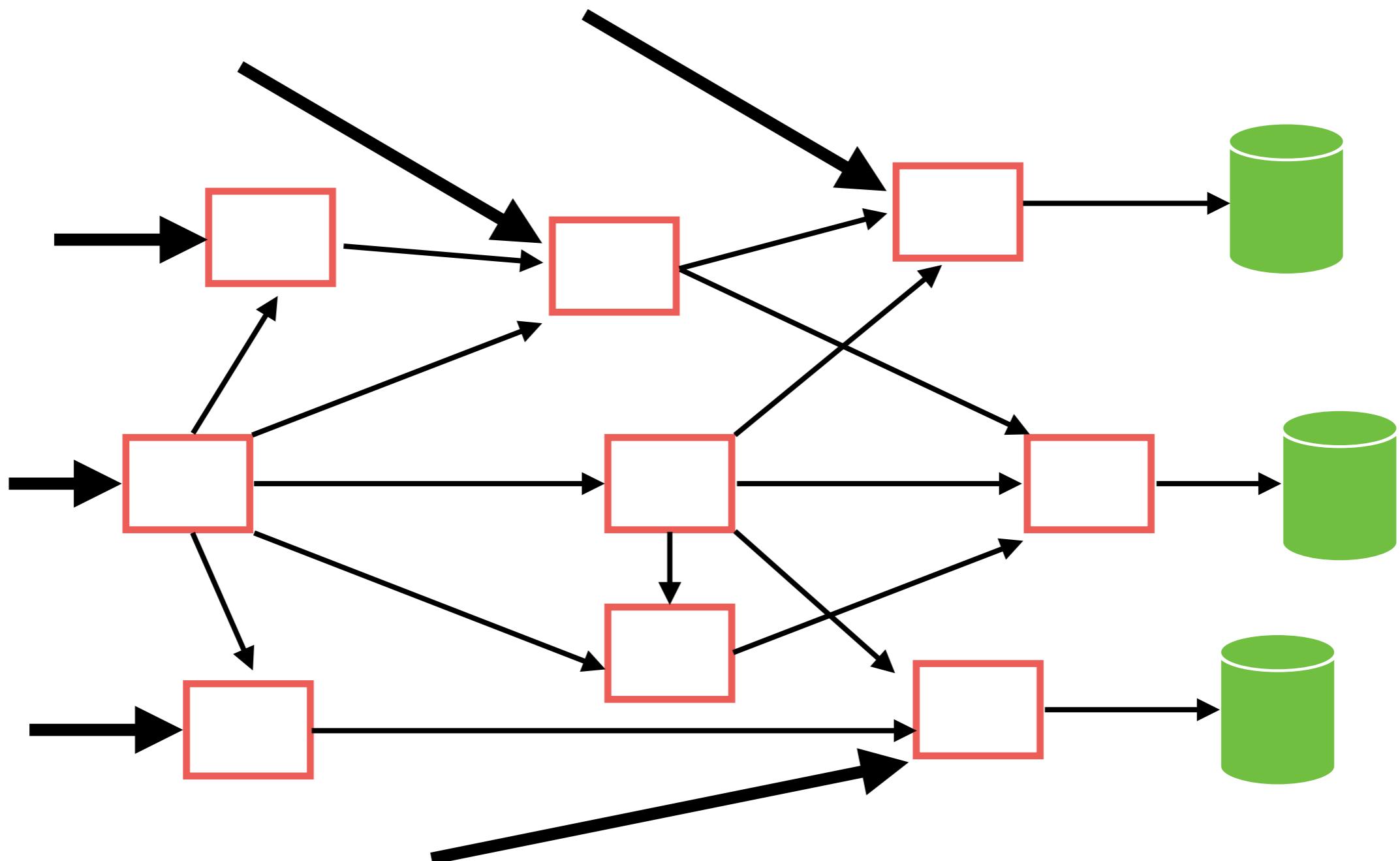
Network of services



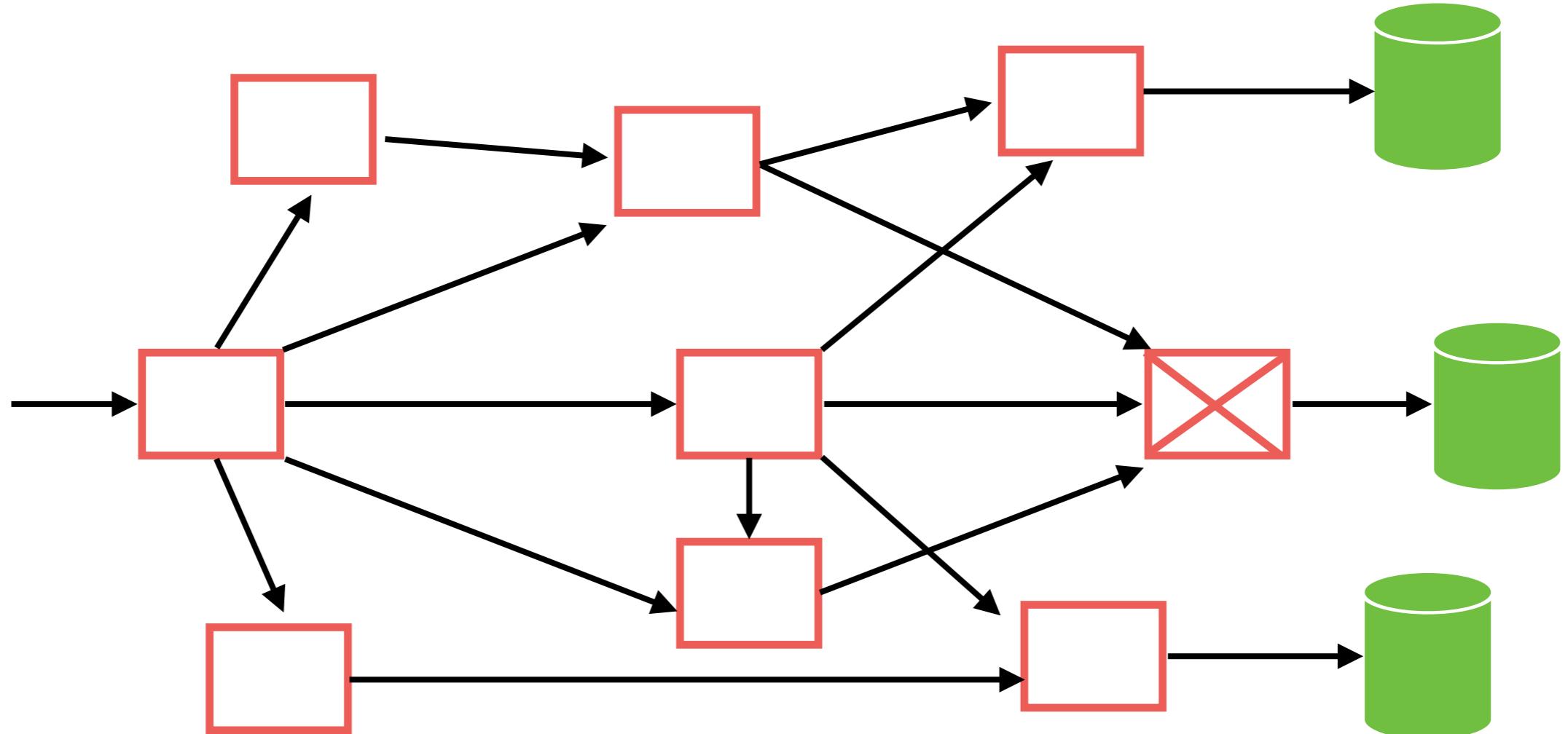
Own data



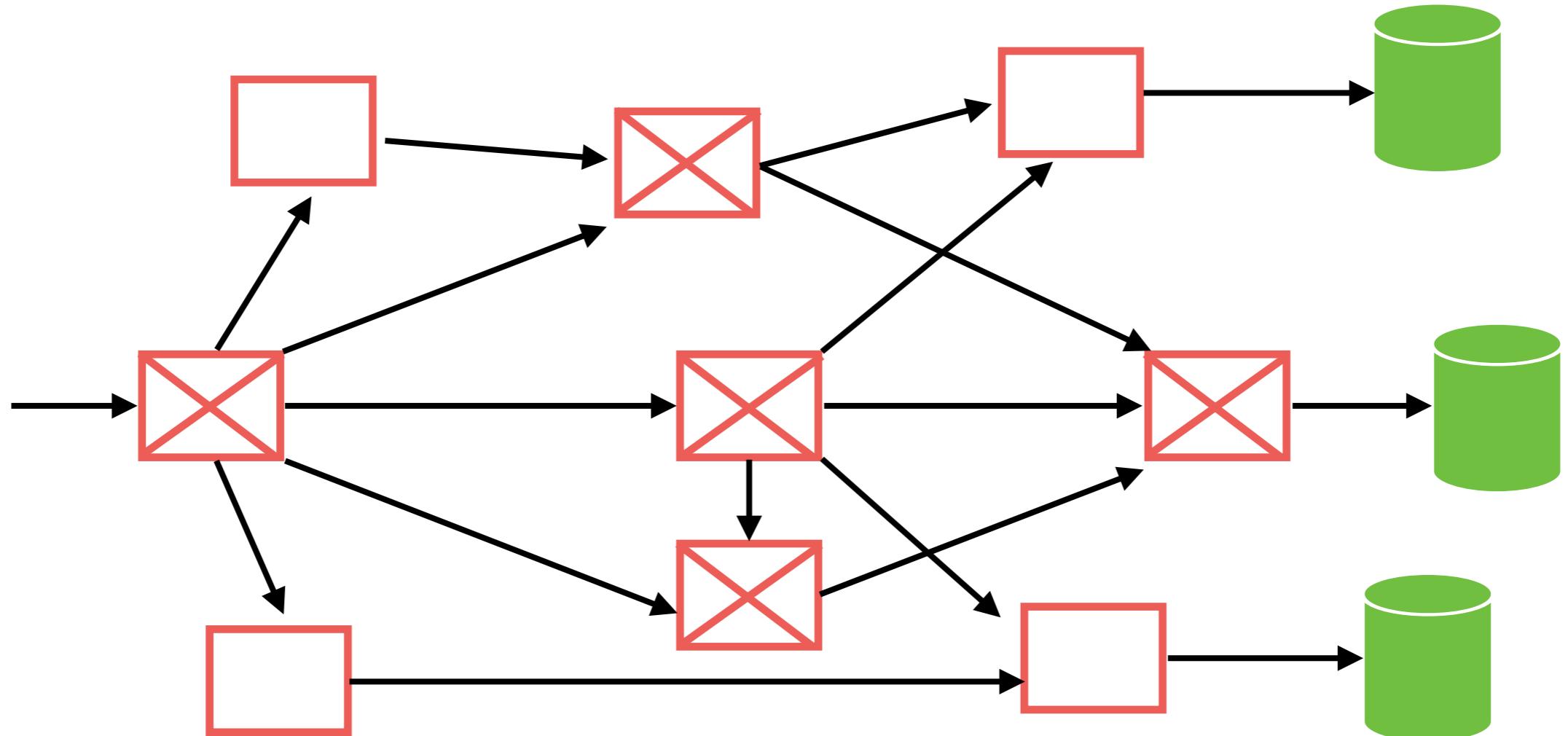
Multiple entry points



Failure !!



Cascading Failure !!

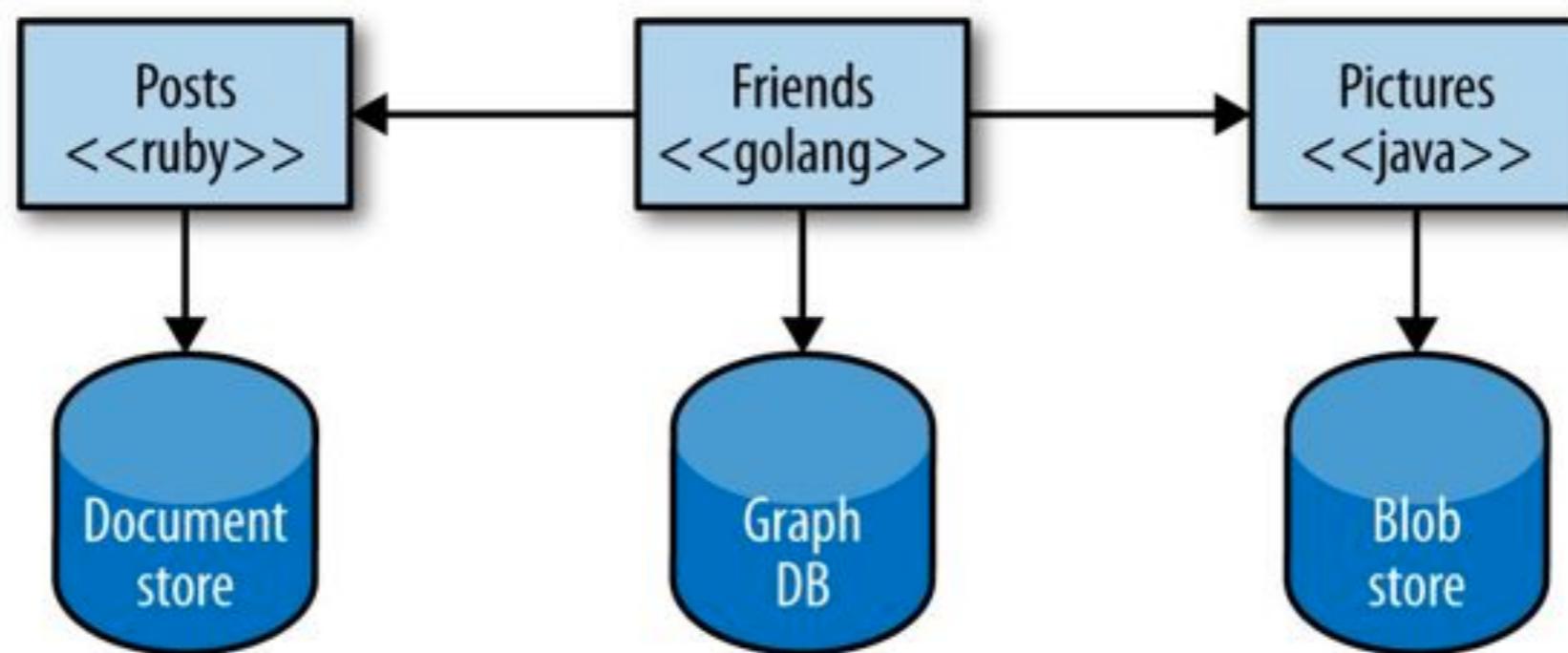


Key Benefits

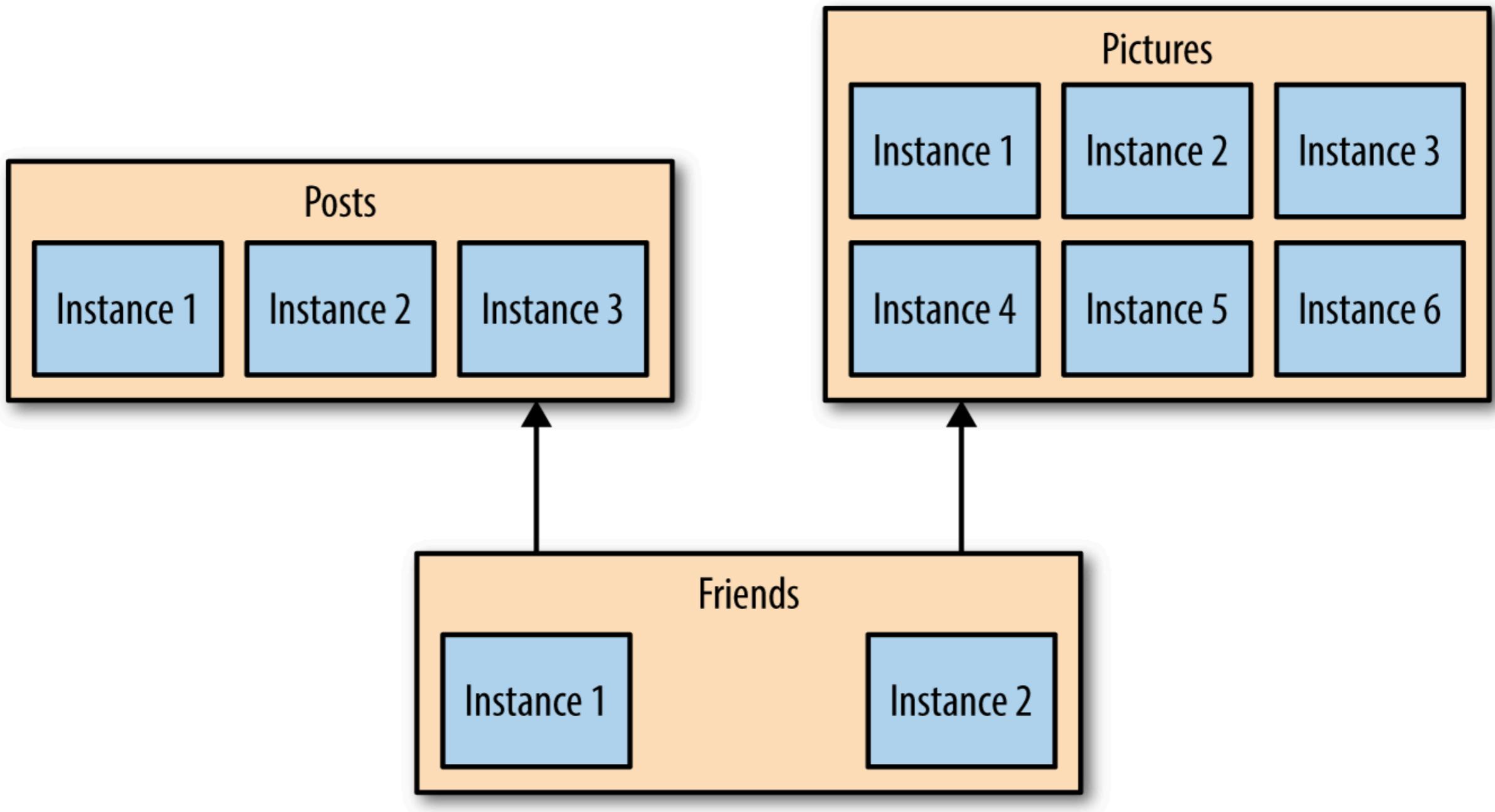


1. Technology heterogeneous

The right tool for each job
Allow easy experimenting and adoption of new technology

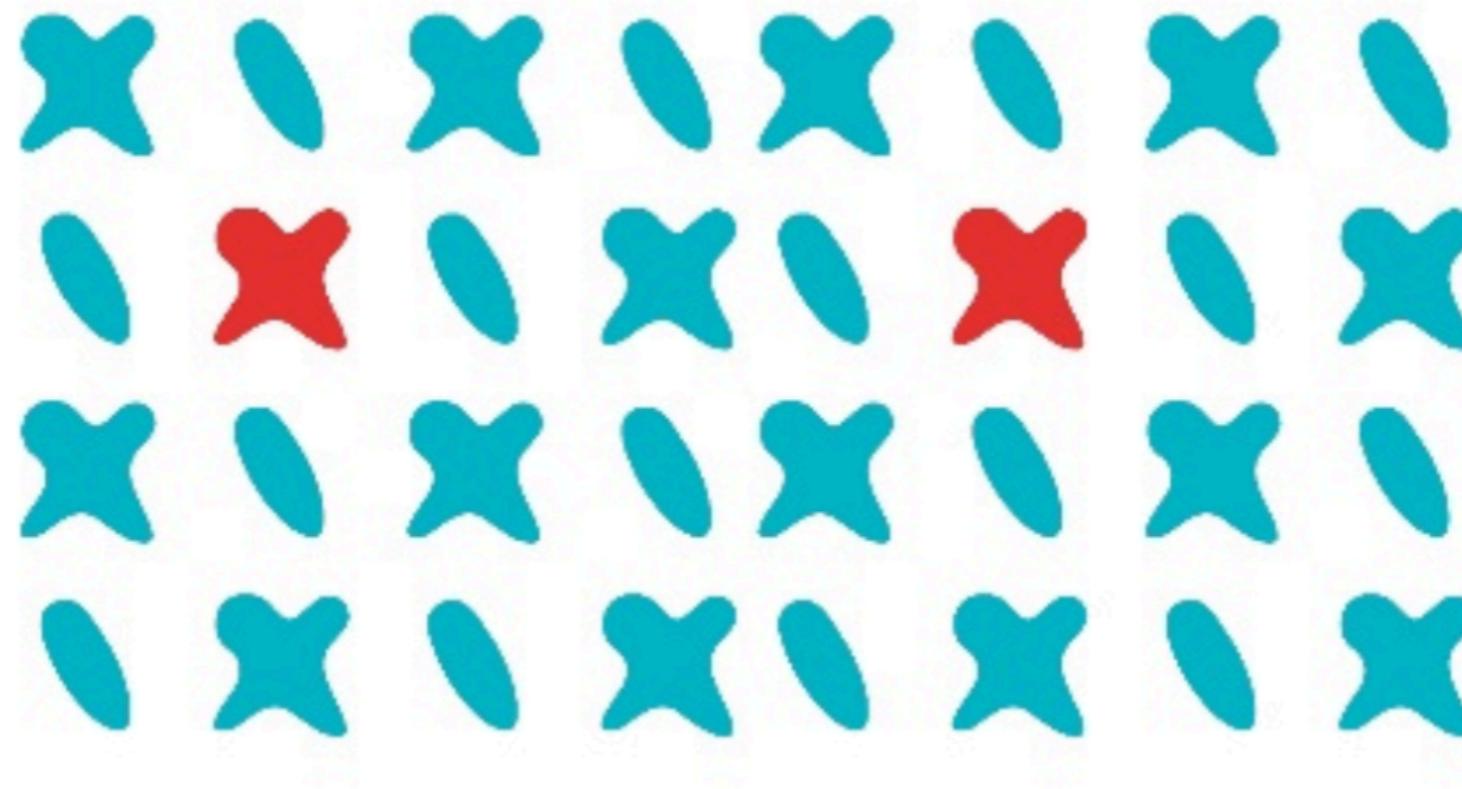


2. Scaling

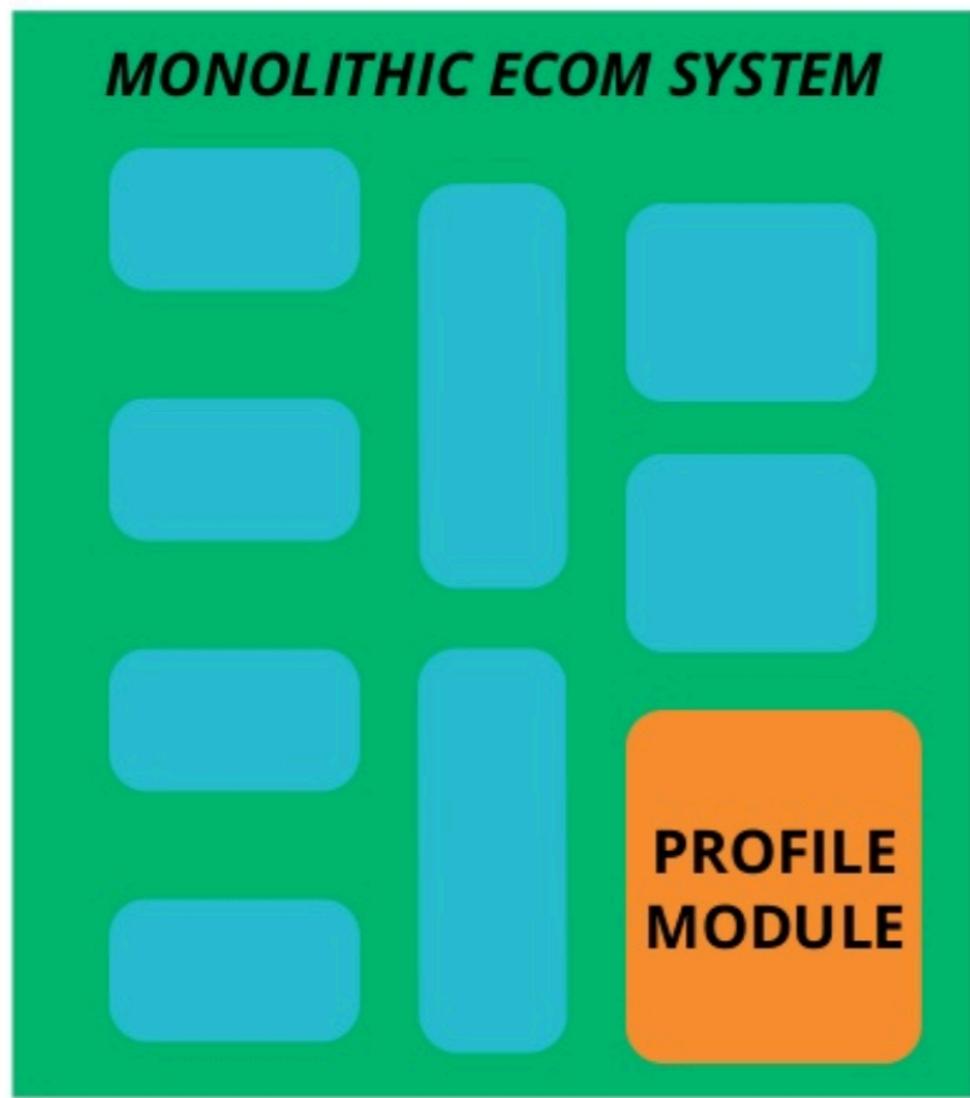


3. Ease of deployment

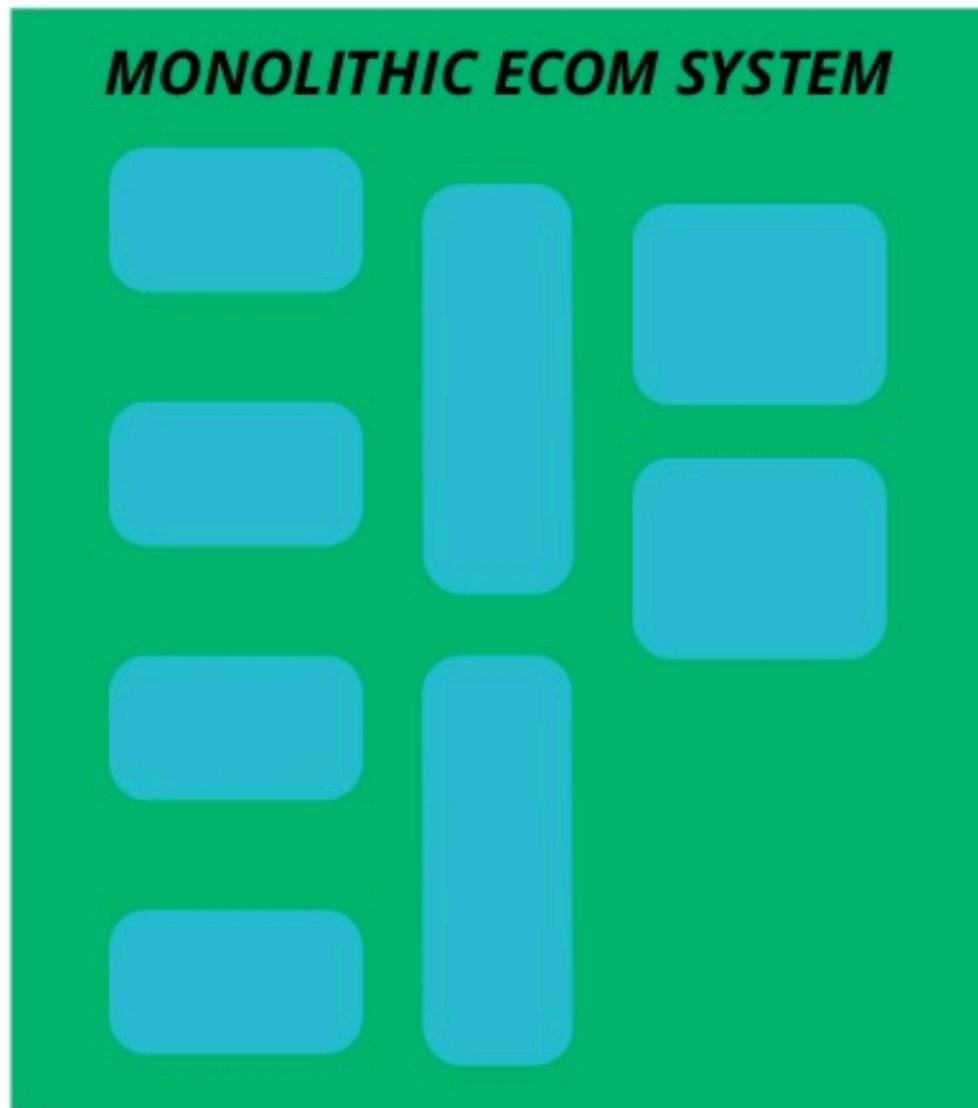
Deploys are faster, independent and problems can be isolated more easily



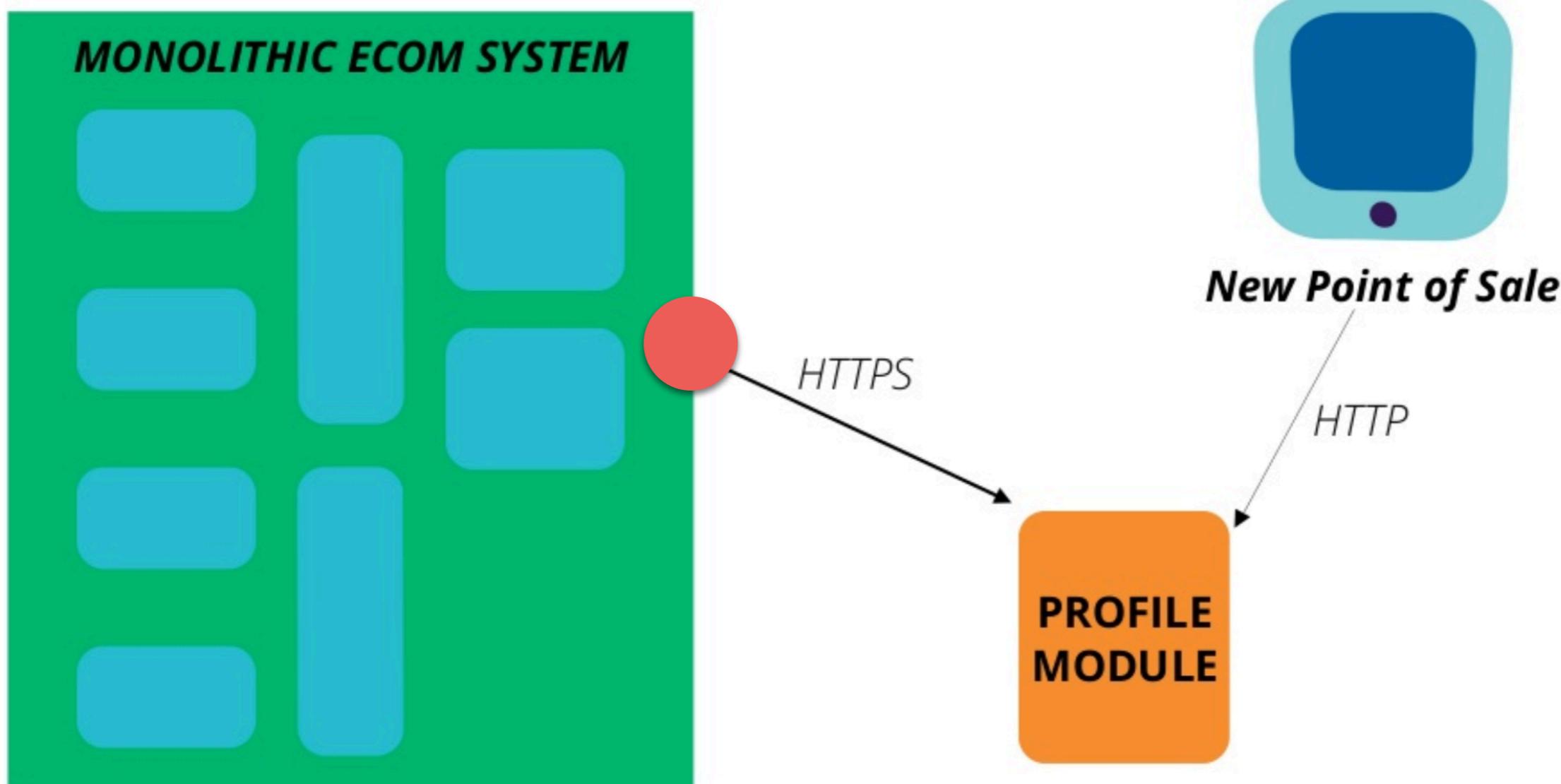
4. Composability and replaceability



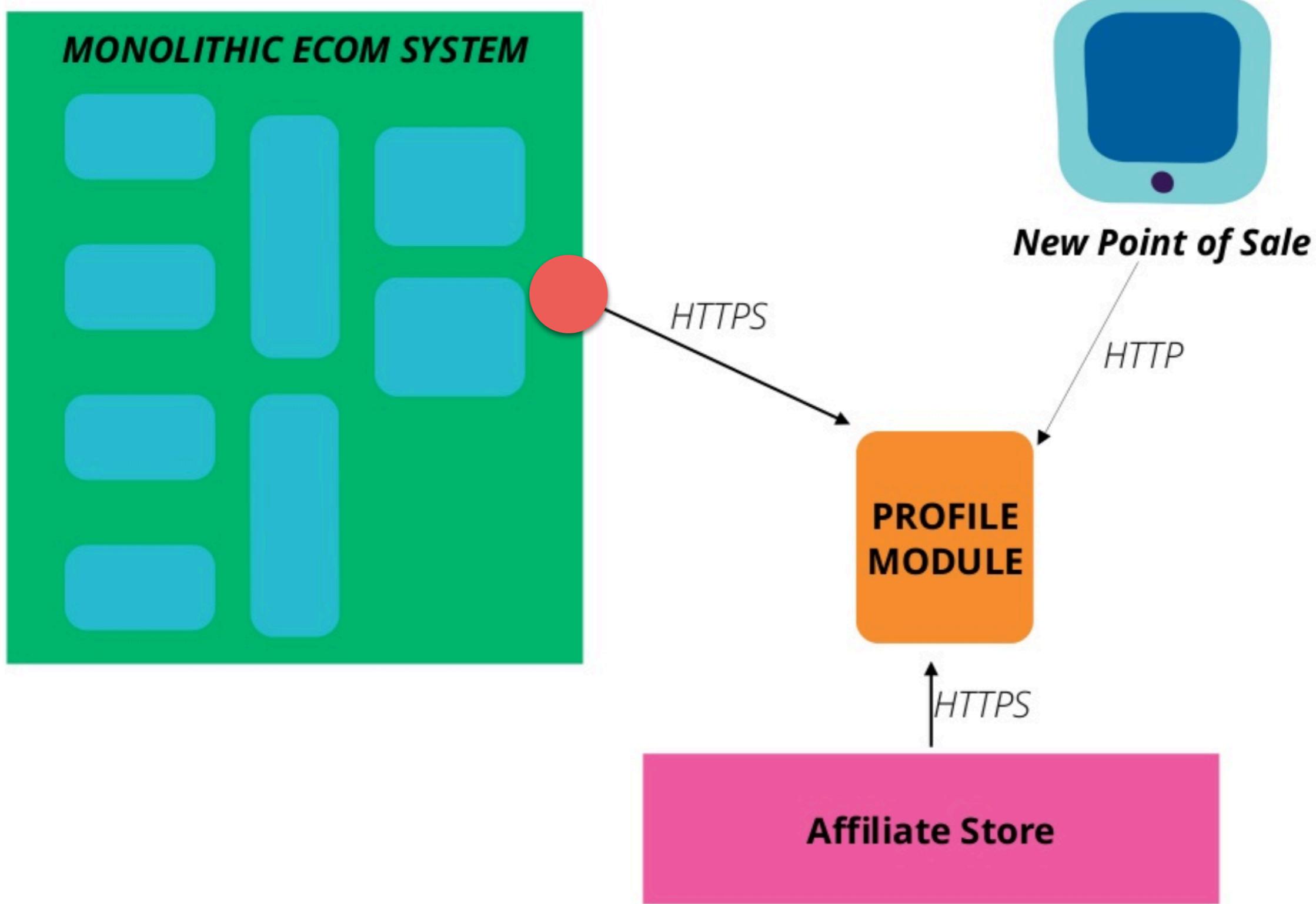
4. Composability and replaceability



4. Composability and replaceability



4. Composability and replaceability

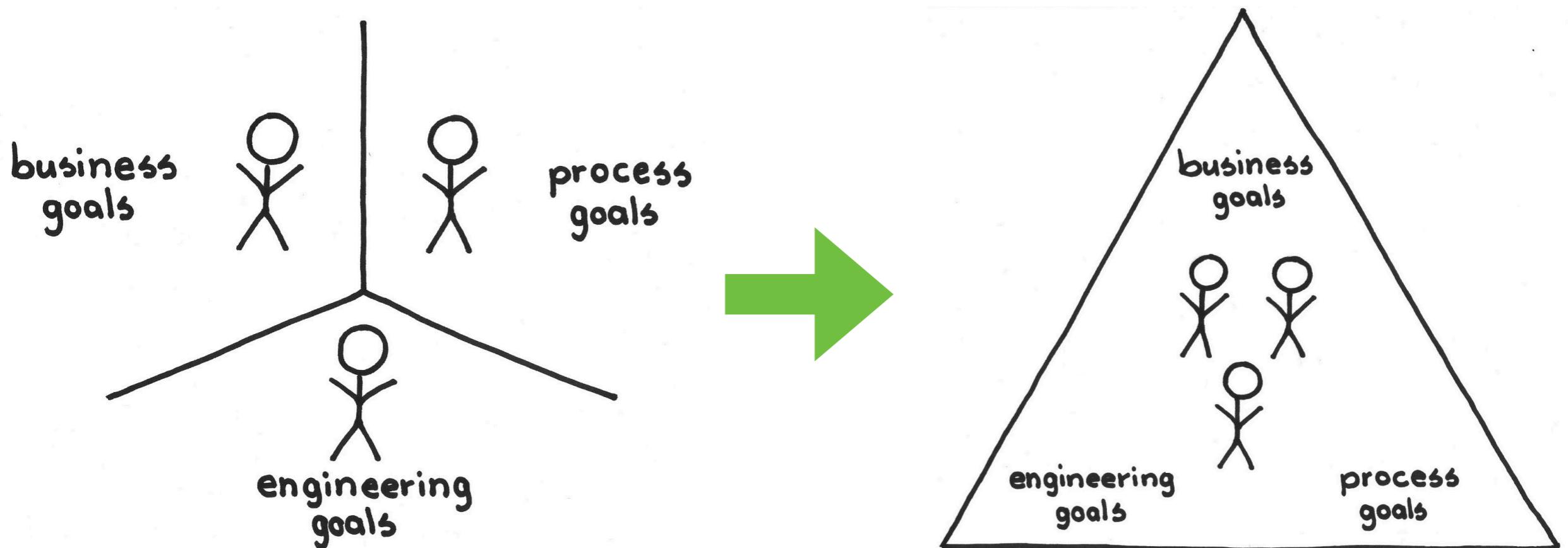


5. Organization alignment

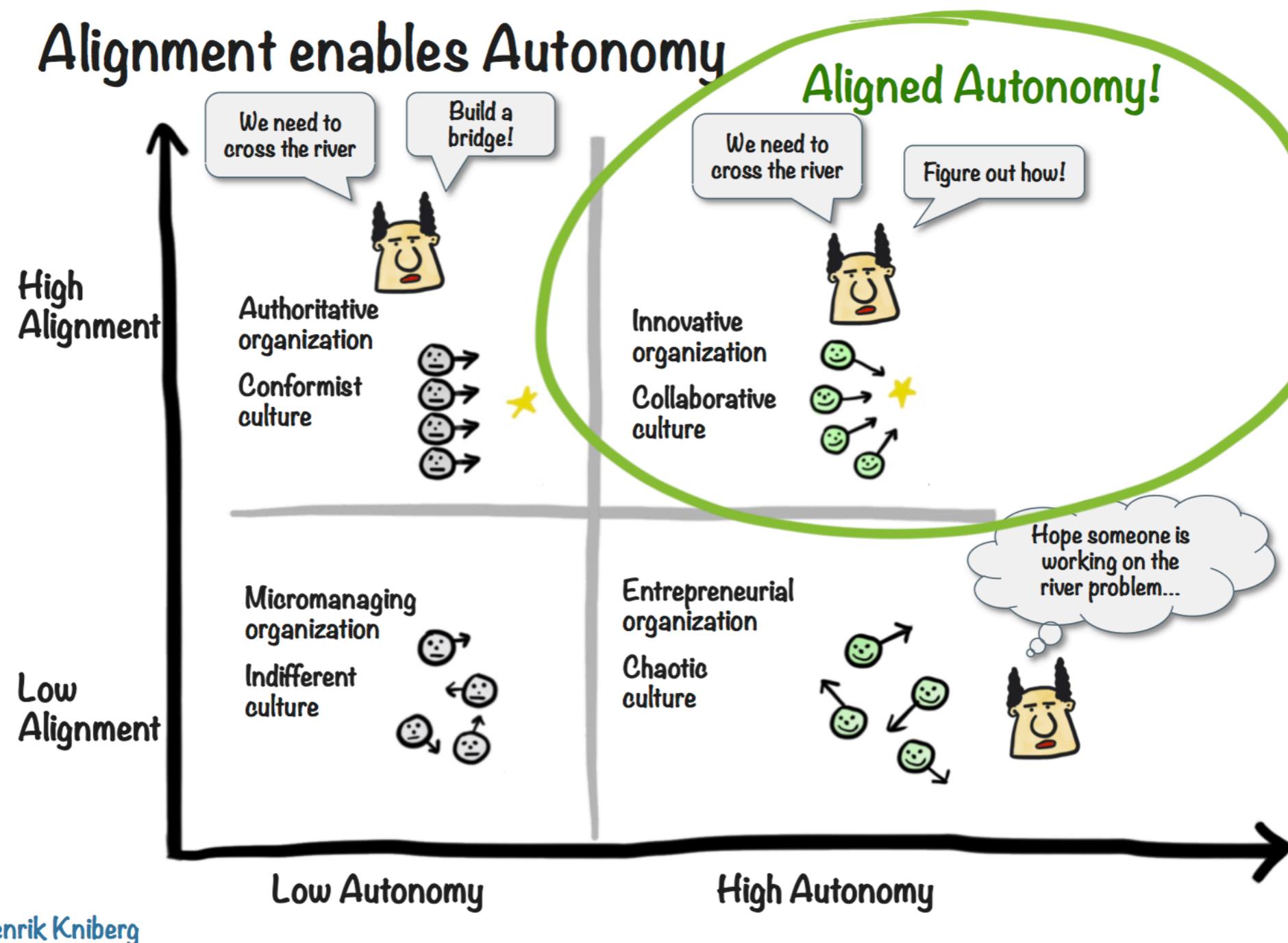
Small teams and smaller codebases



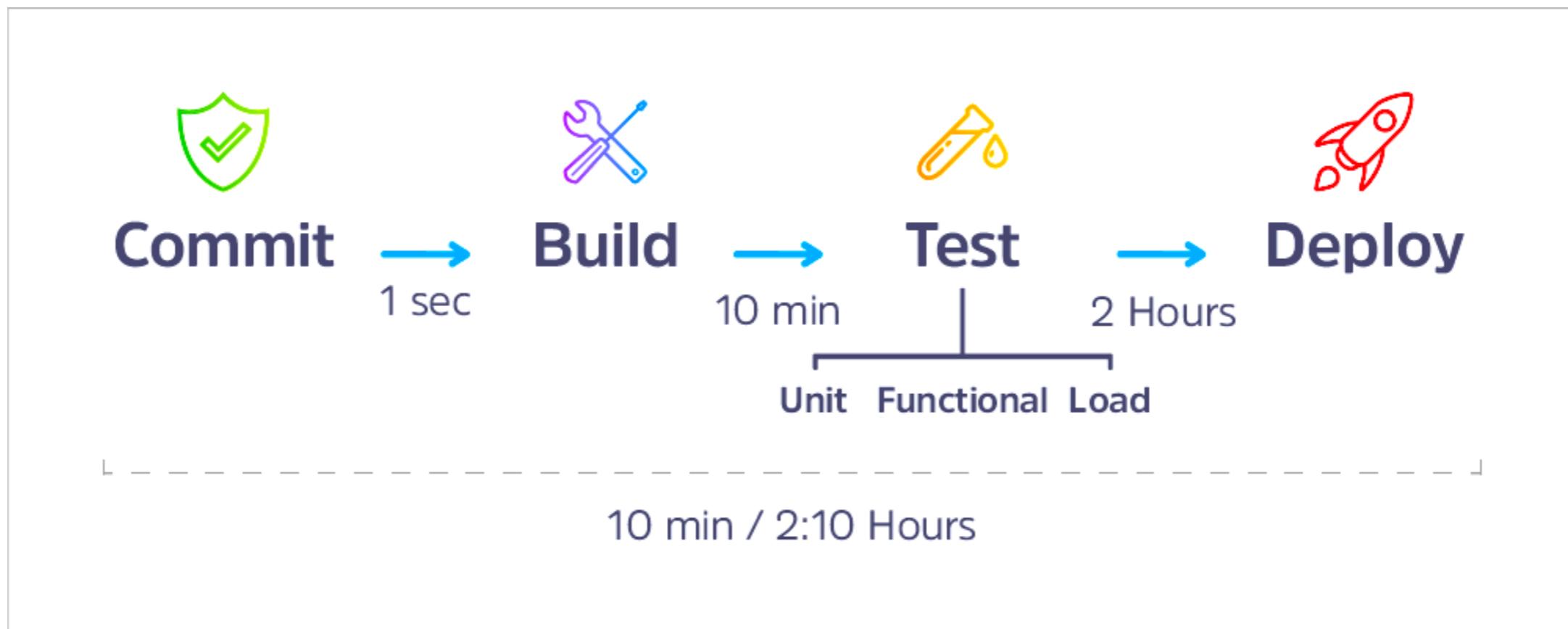
Autonomous team



Autonomous team



Autonomous team



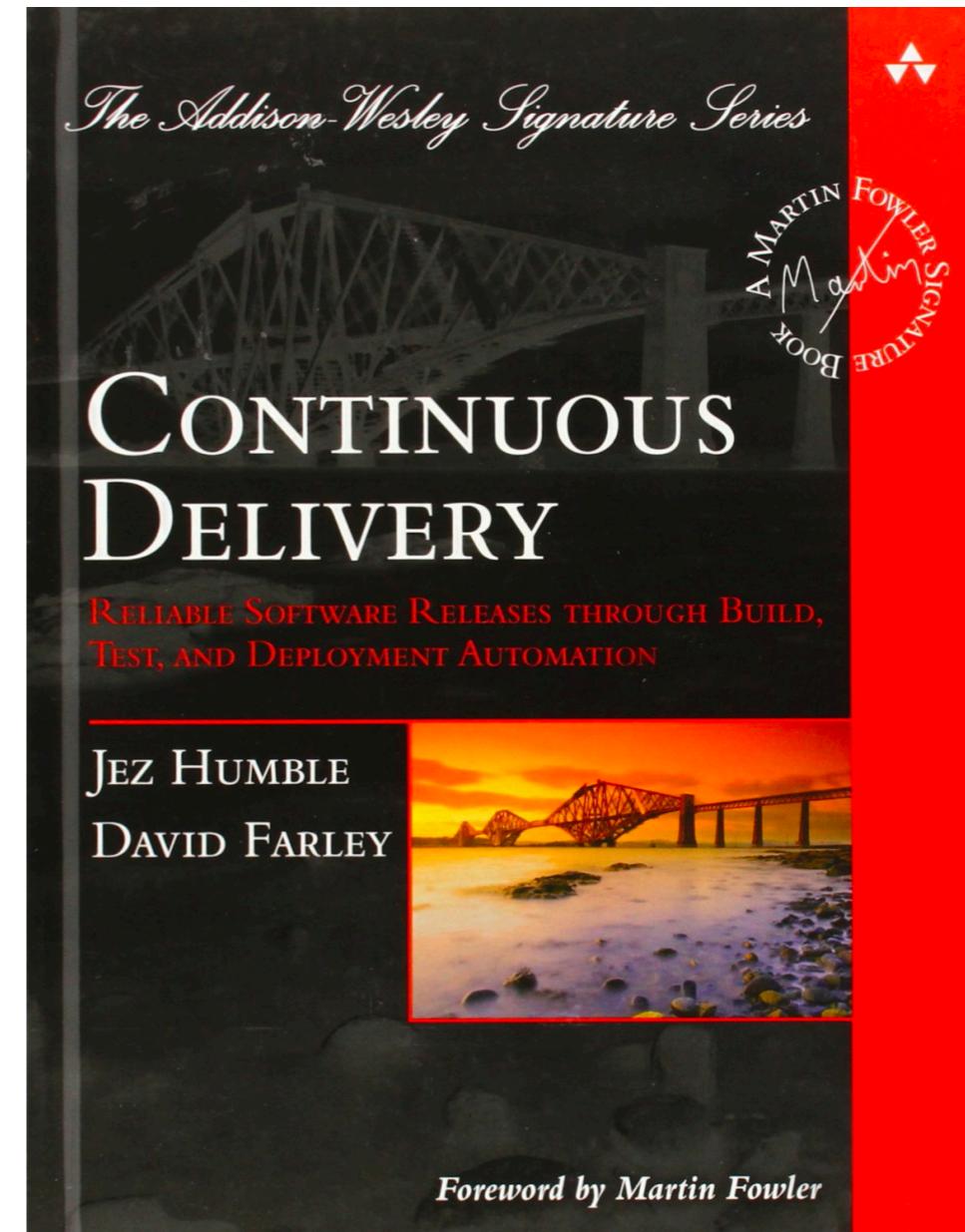
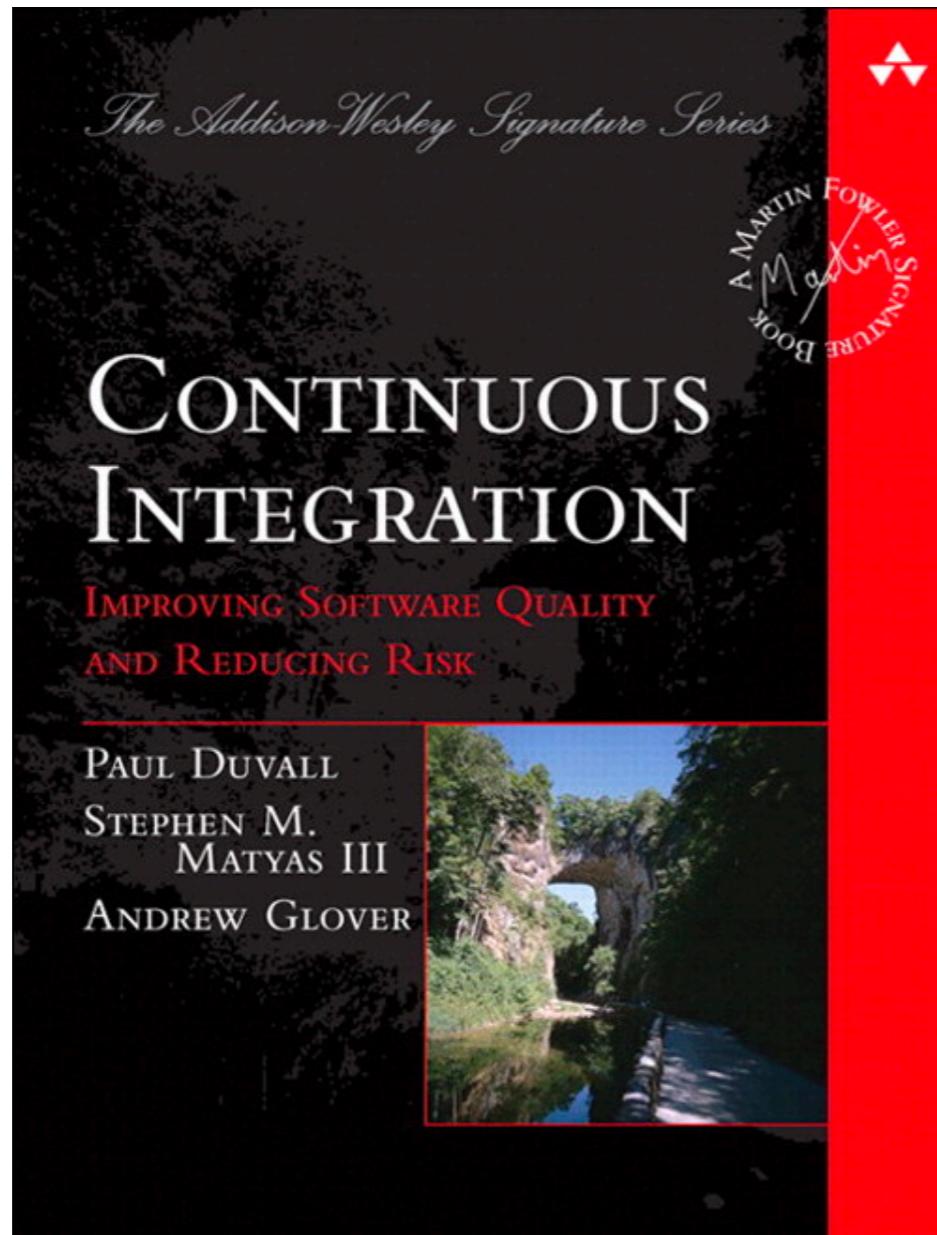
6. Enable Continuous Delivery

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

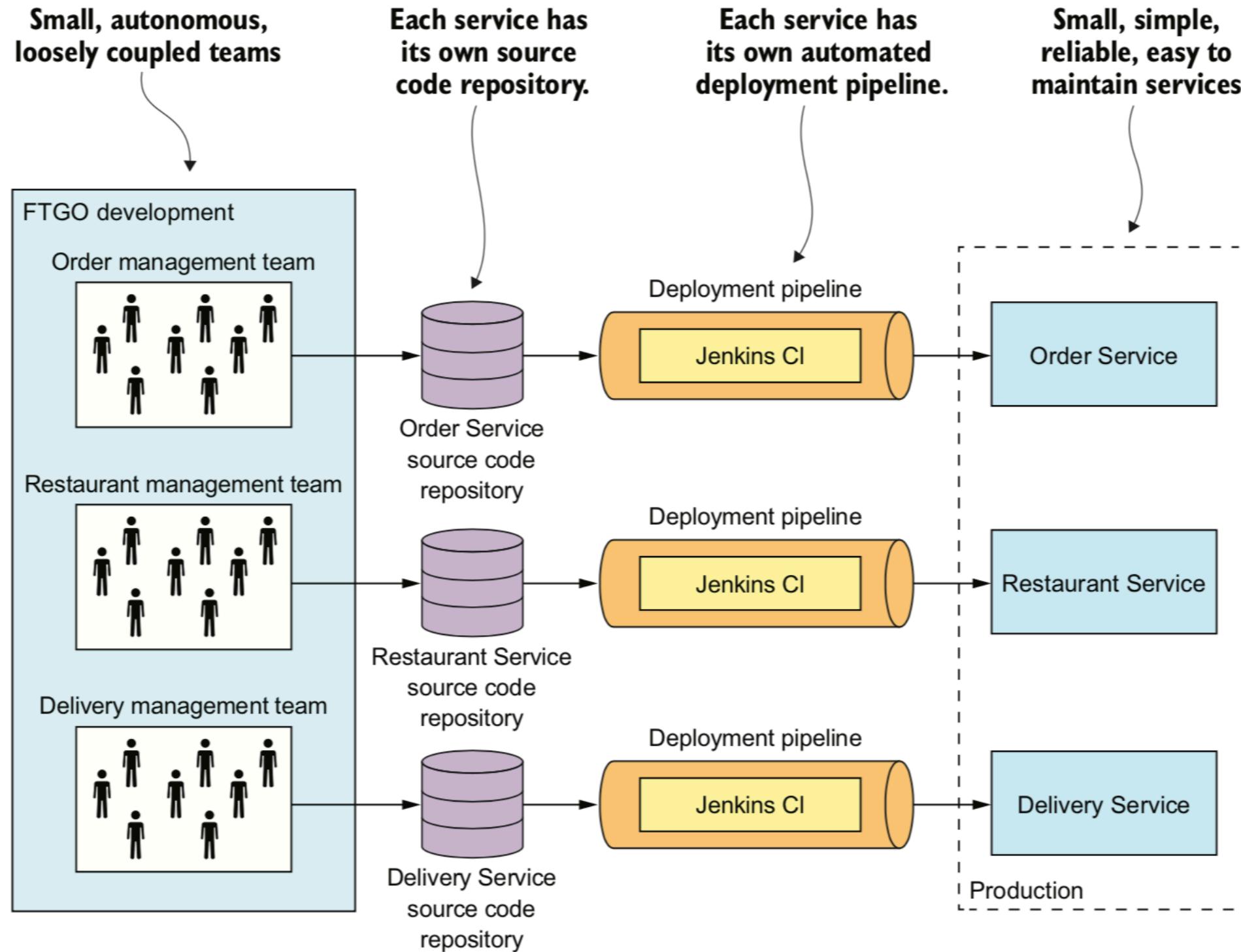
Reduce time to market

Enable **business** to provide reliable service

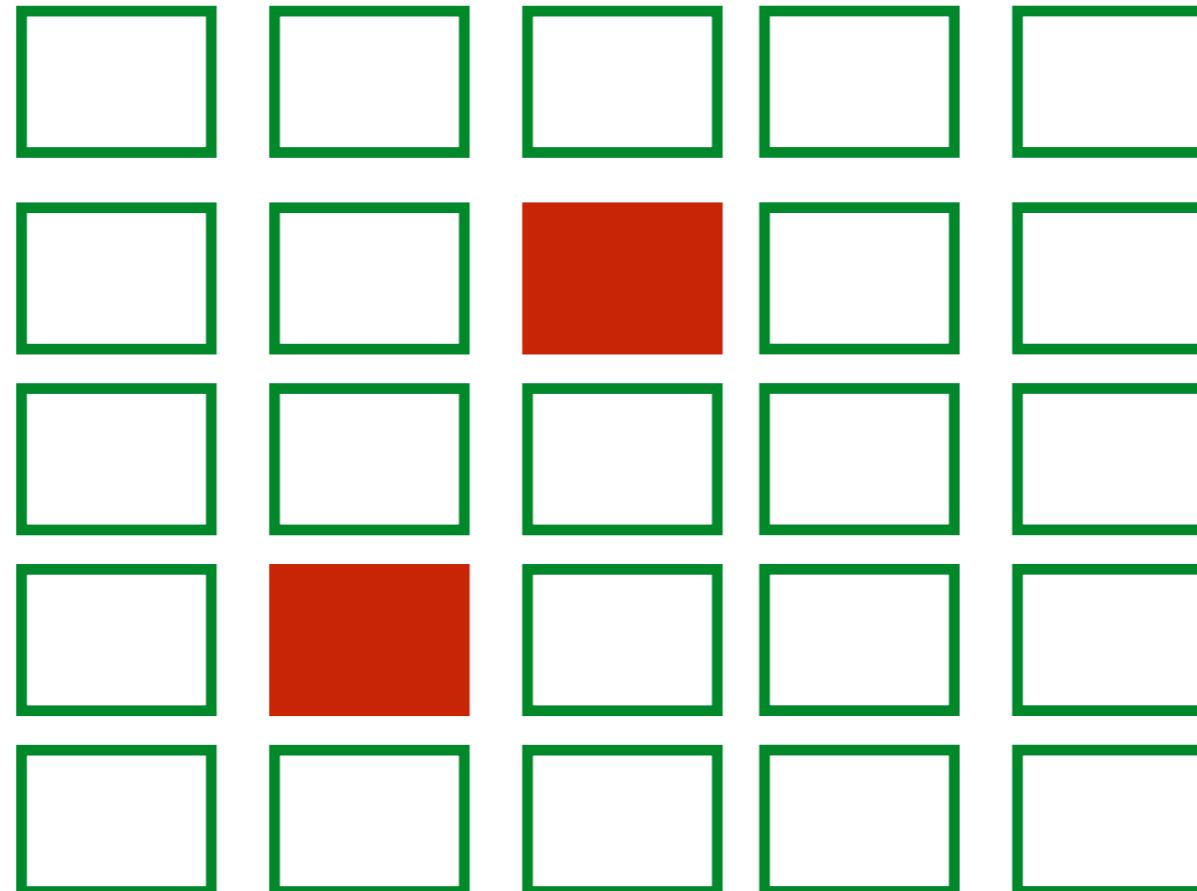
Employee satisfaction is higher



Continuous Delivery for service



7. Better Fault isolation



Drawbacks of Microservice



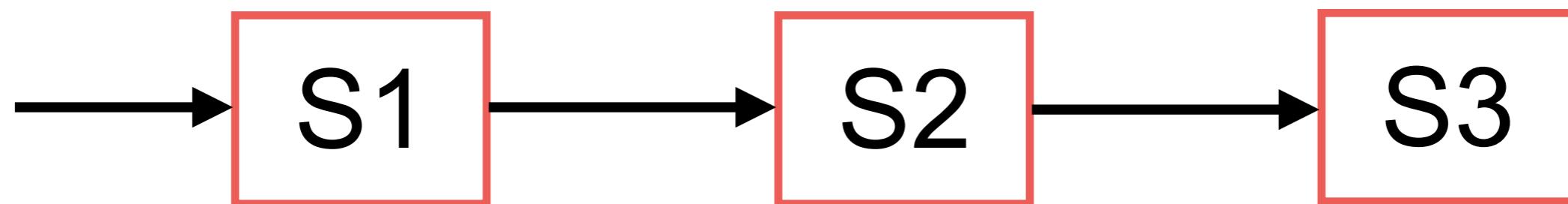
Drawbacks of Microservice

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



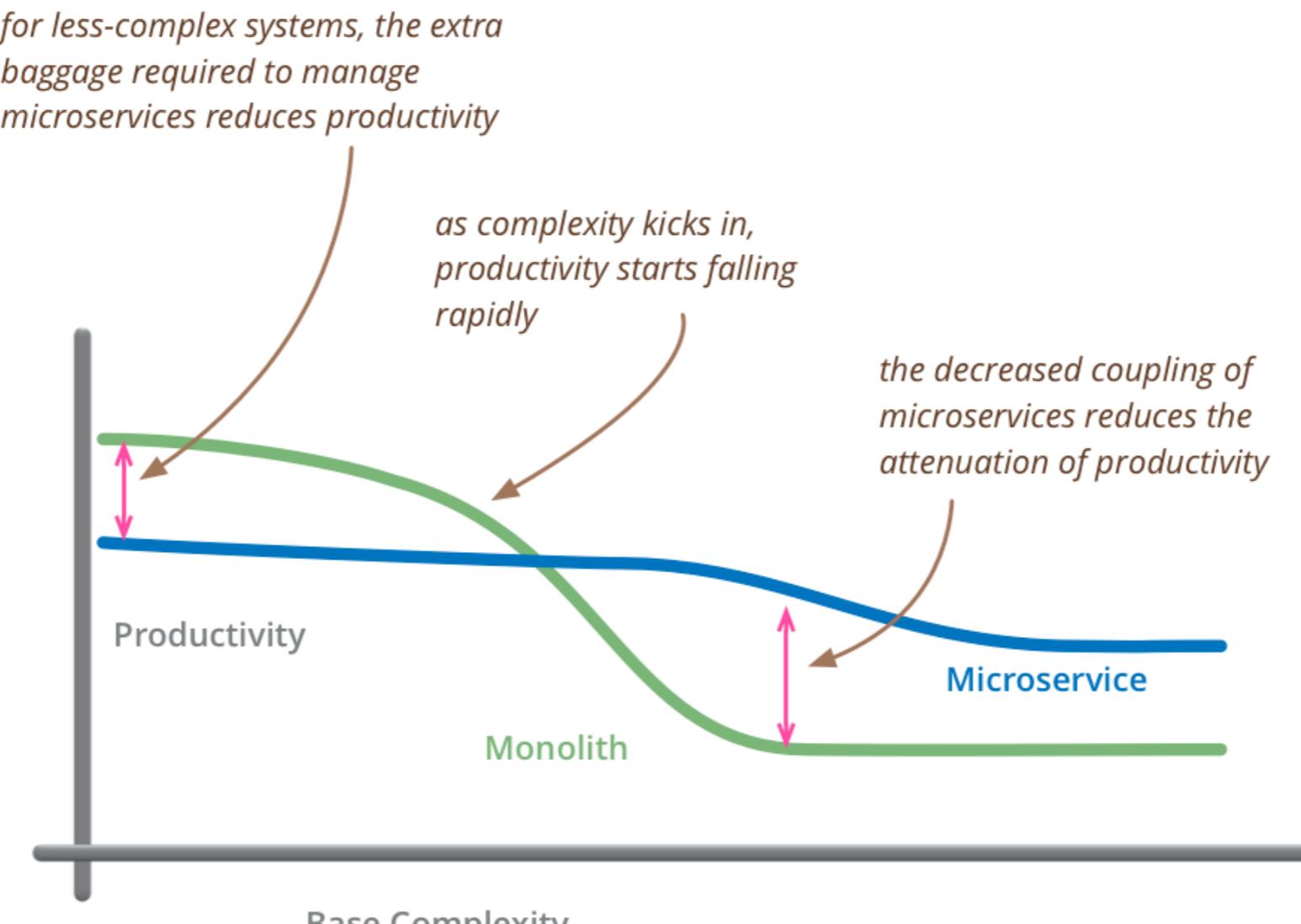
Drawbacks of Microservice

Deploy feature that required multiple service ?



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

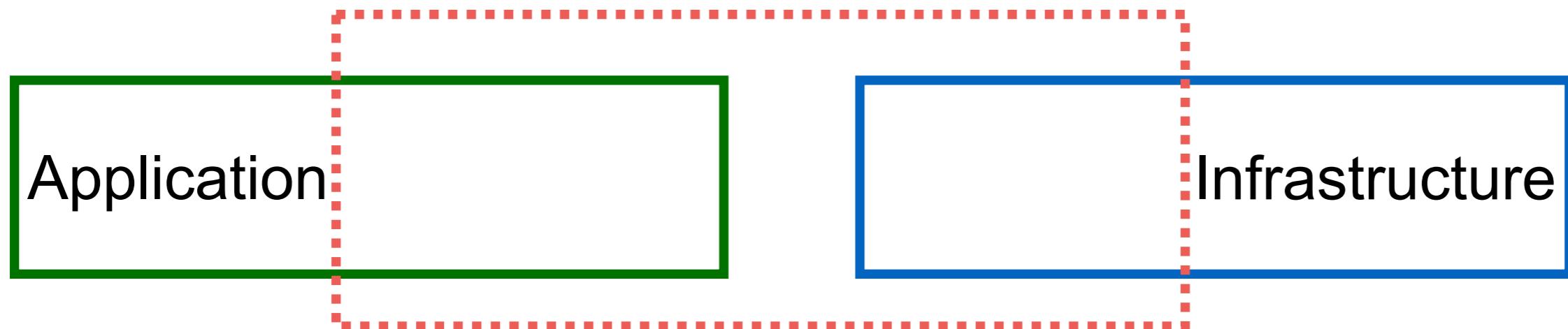


3 Layers of service patterns

Application patterns

Application infrastructure pattern

Infrastructure pattern



Patterns for Microservice

Decompose application into services

 Communication patterns

 Data consistency patterns

 Service deployment patterns

 Observability patterns

 Automated testing of services

 Security patterns



Decompose application into services



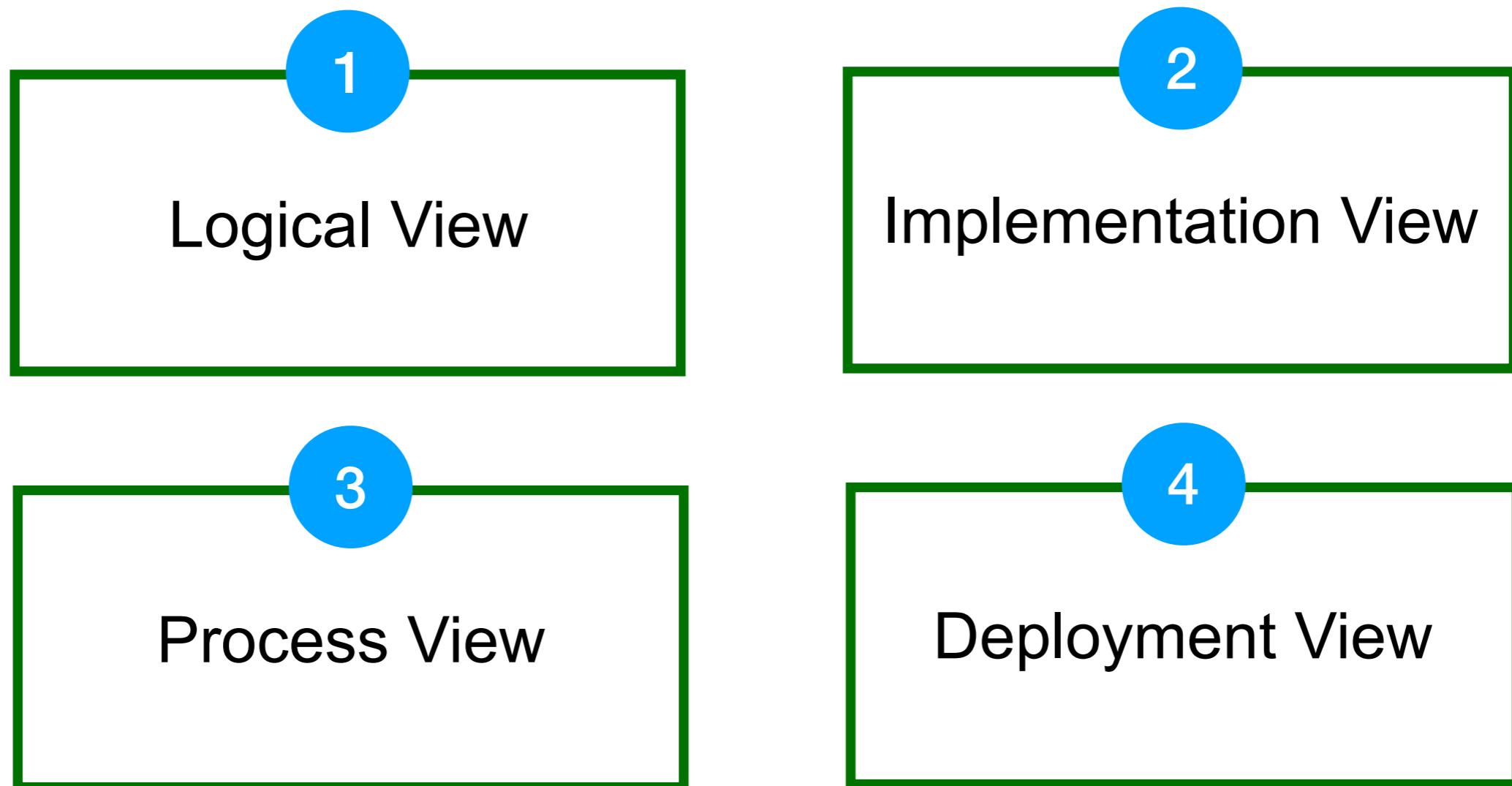
Premature splitting is the root of
all evil.



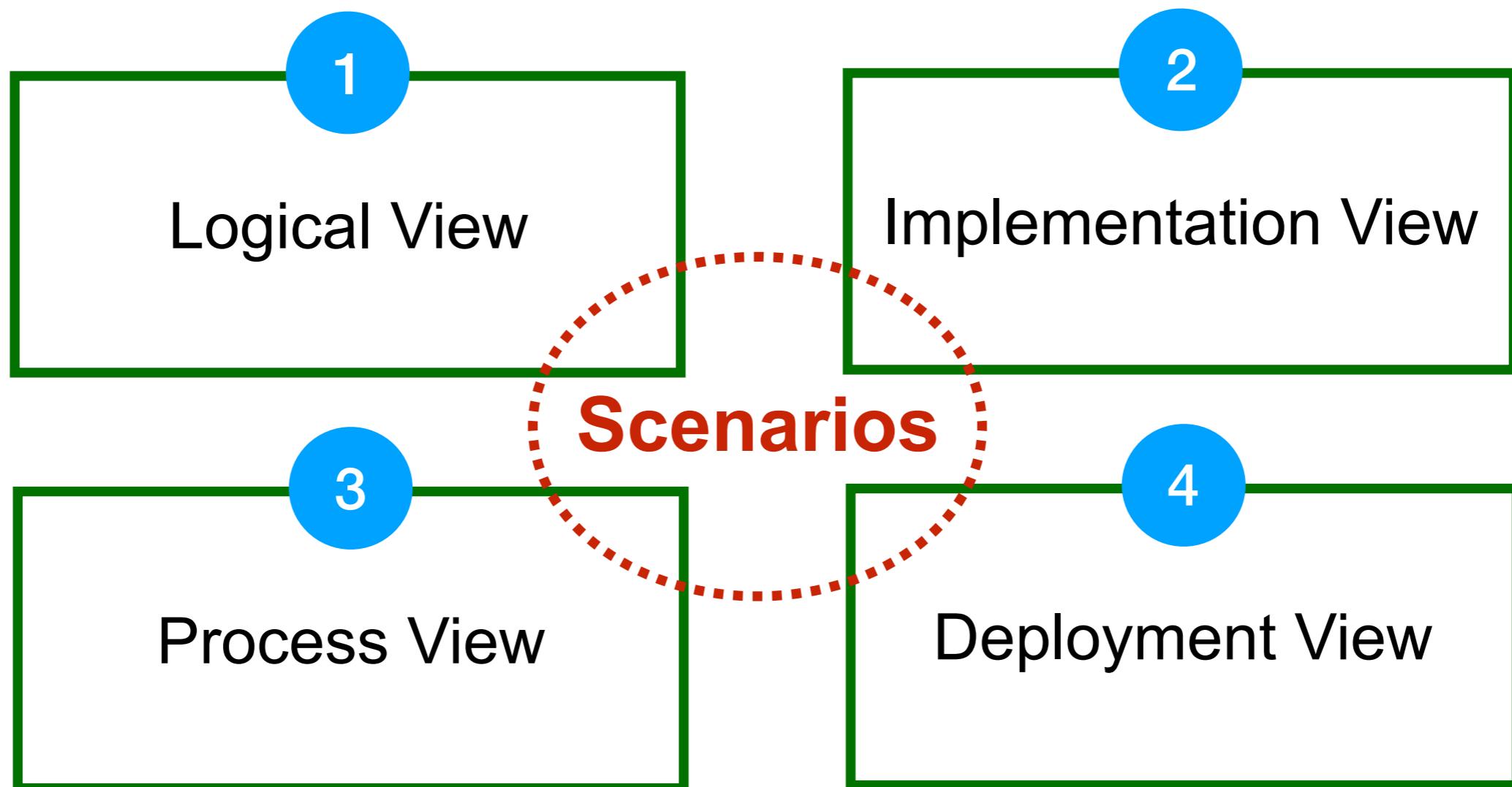
Every time you make the decision
to split out a new microservice,
there's a **risk** of ending up with a
bloated app.



4 View model of Software Architecture



4 View model of Software Architecture



What is service ?

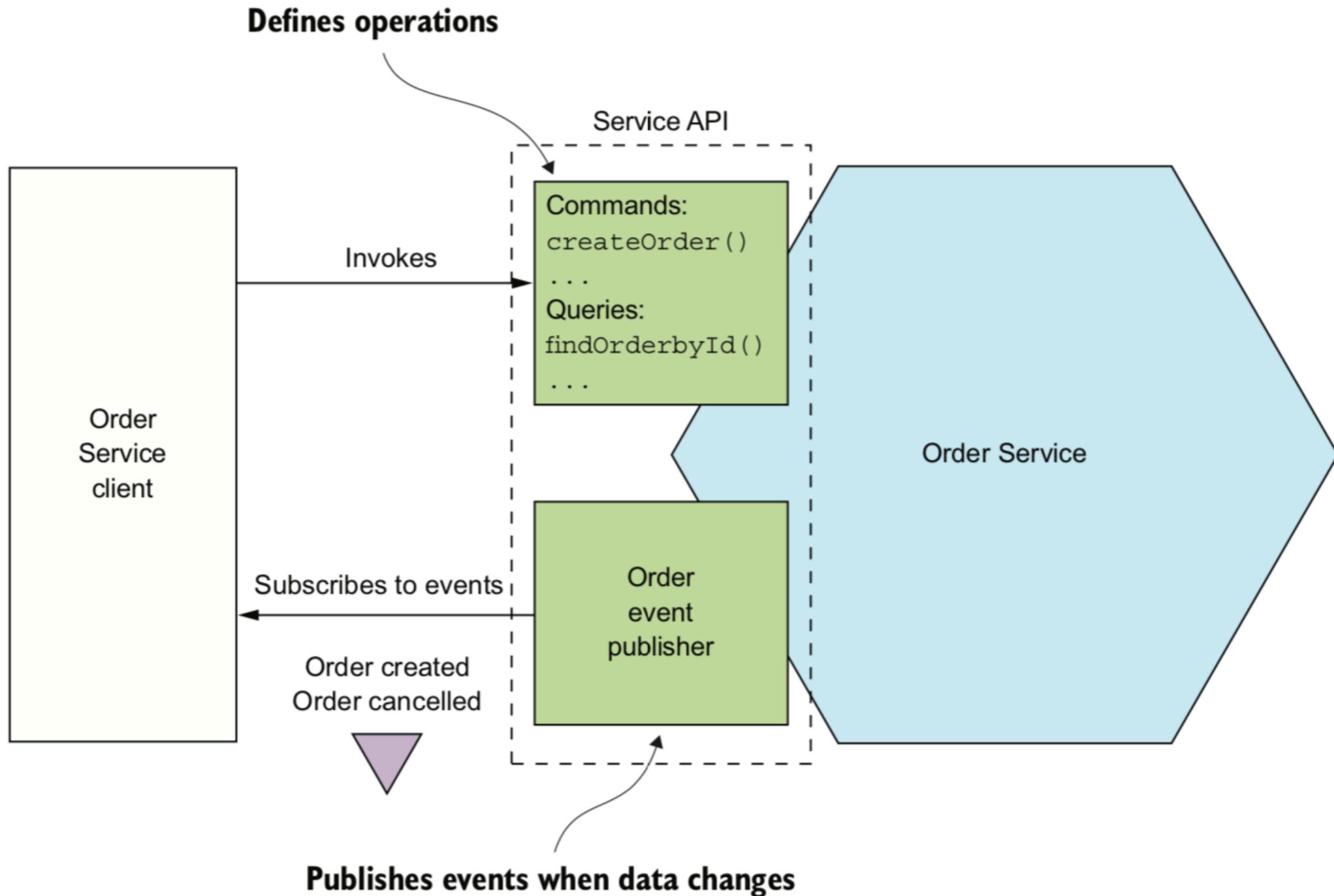
Standalone and loosely couple

Independently deployable software component

Implement some useful functionality



Example of service



Decompose application into services

By business capability ?
By subdomain/technical ?



Step to define services

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

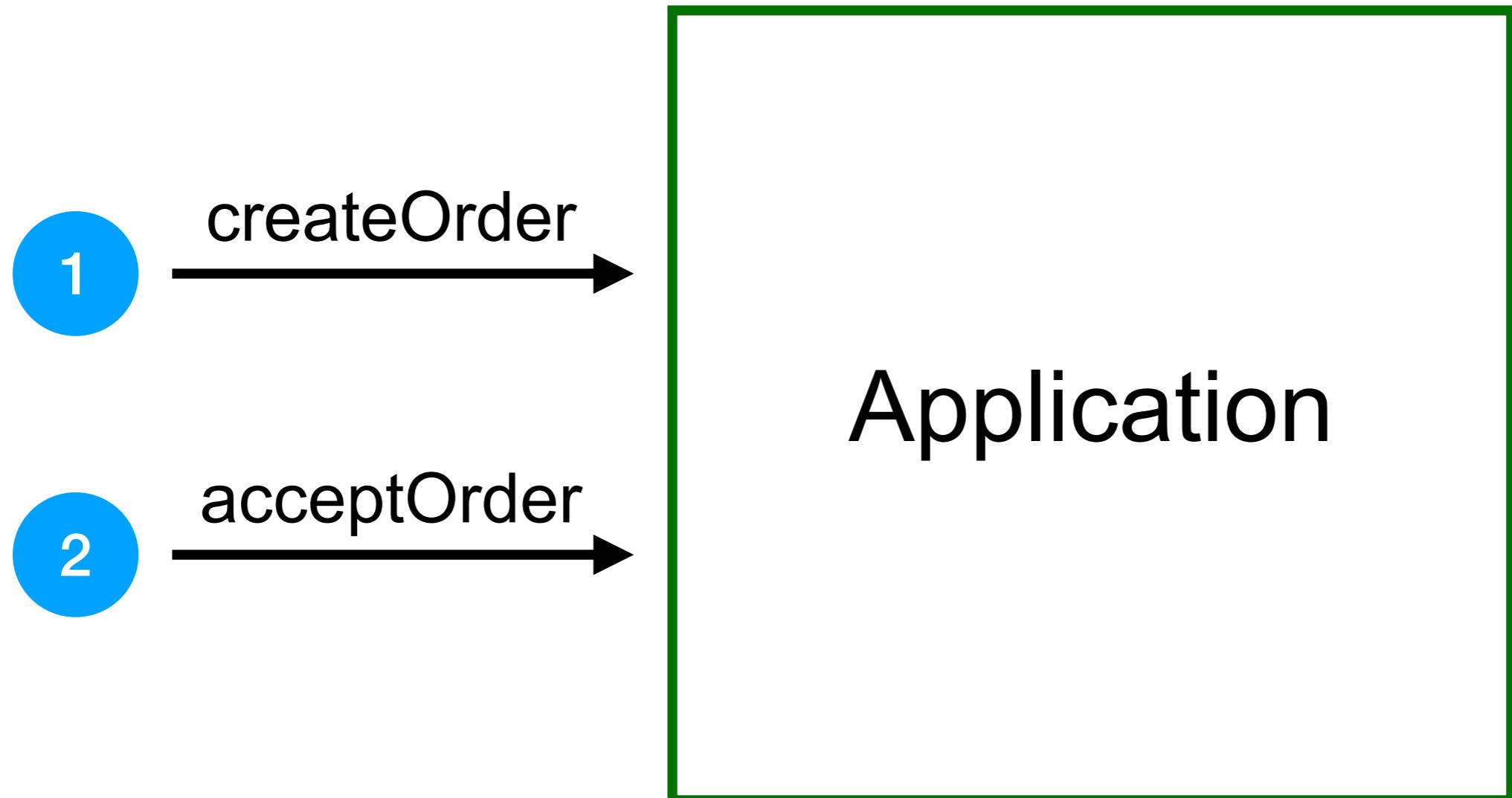


1. Identify system operations

Start with functional requirements
User story



1. Identify system operations



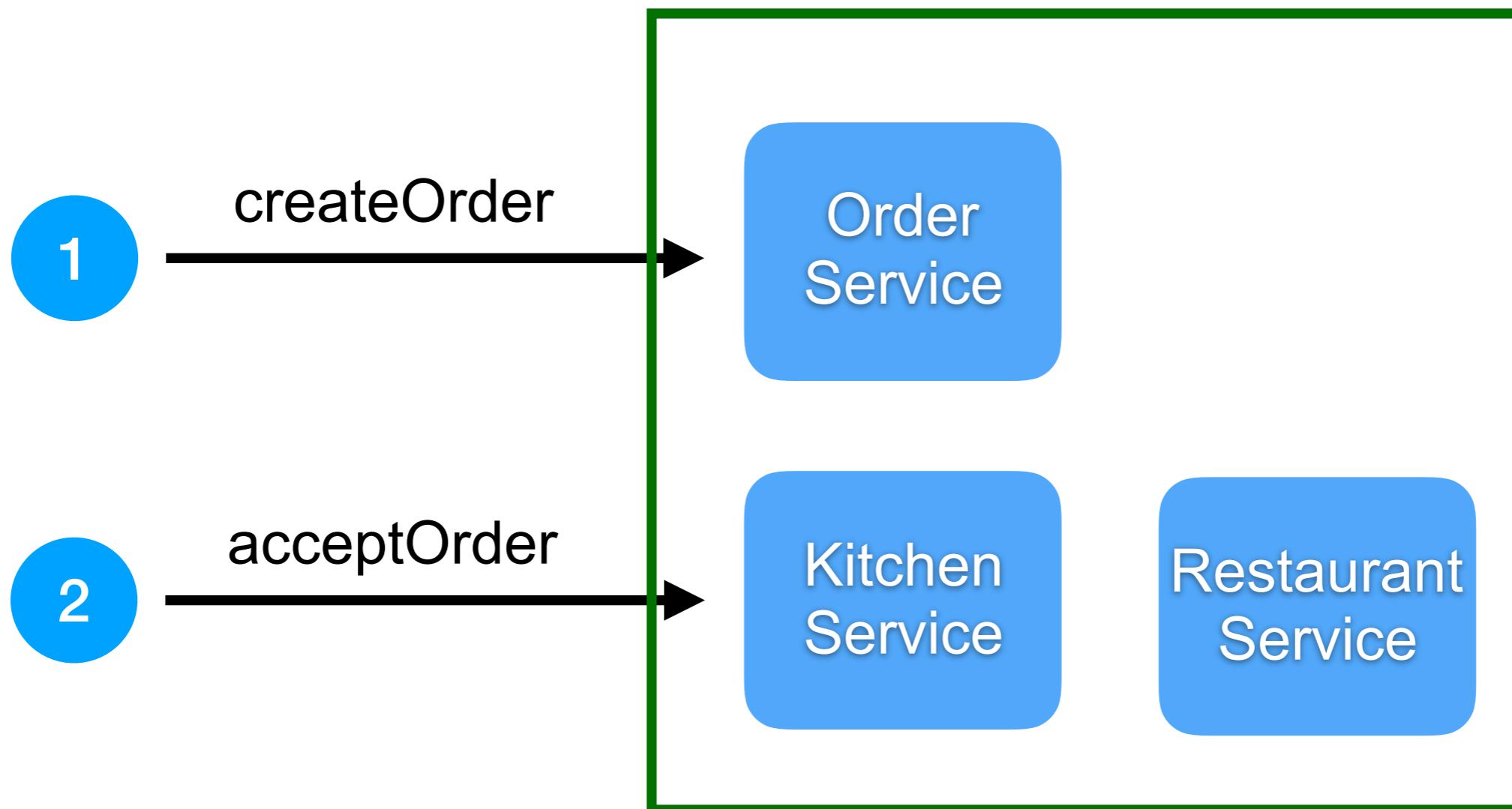
2. Identify services

Try to decomposition into services

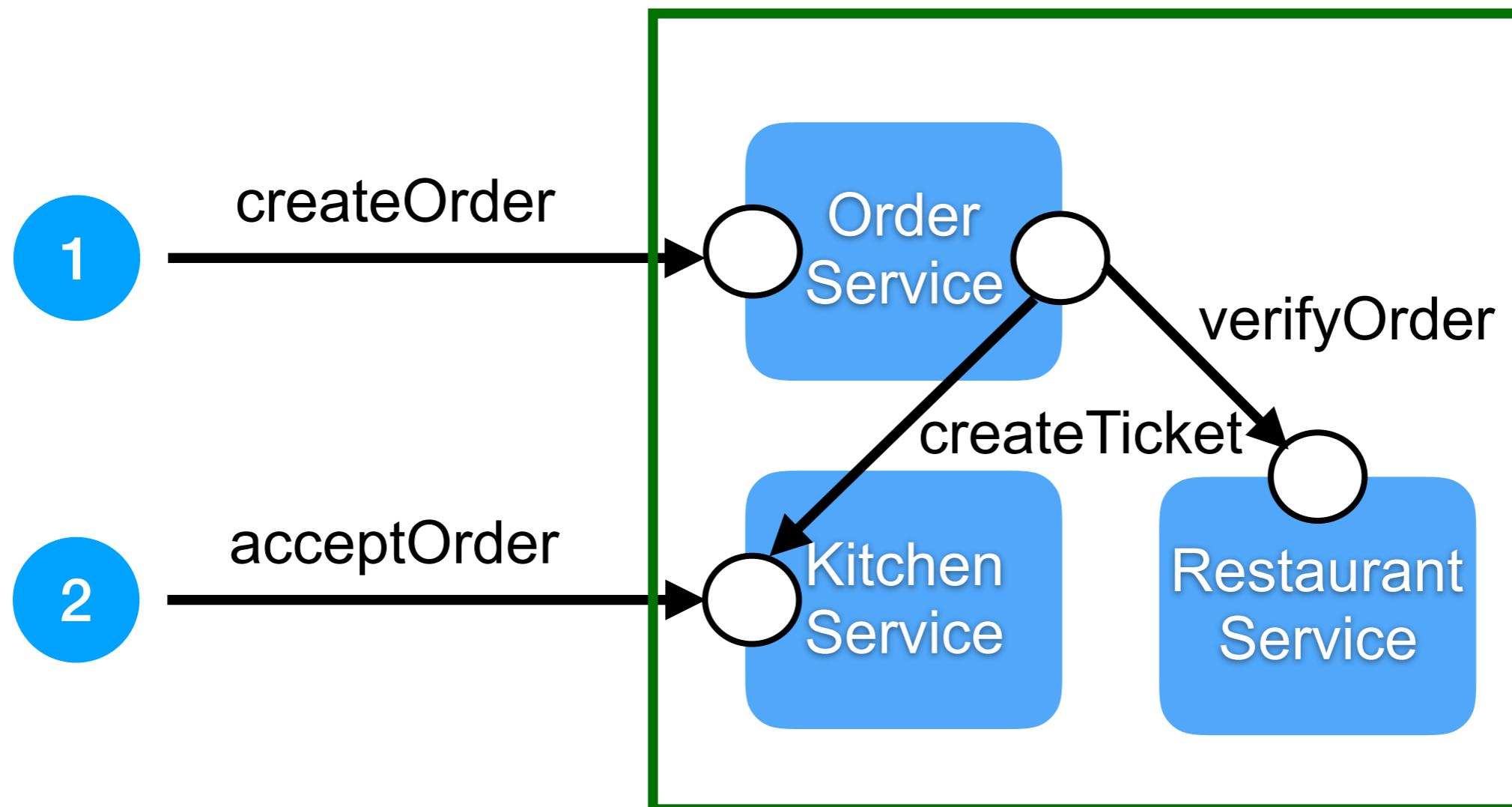
Business > Technical concept
What > How



2. Identify services



3. Define service APIs and collaborations



Problems when decompose services ?

Network latency

Reliability of communication (sync)

Maintain data consistency

God classes



Communication between services



Communication patterns

Communication style

Discovery service

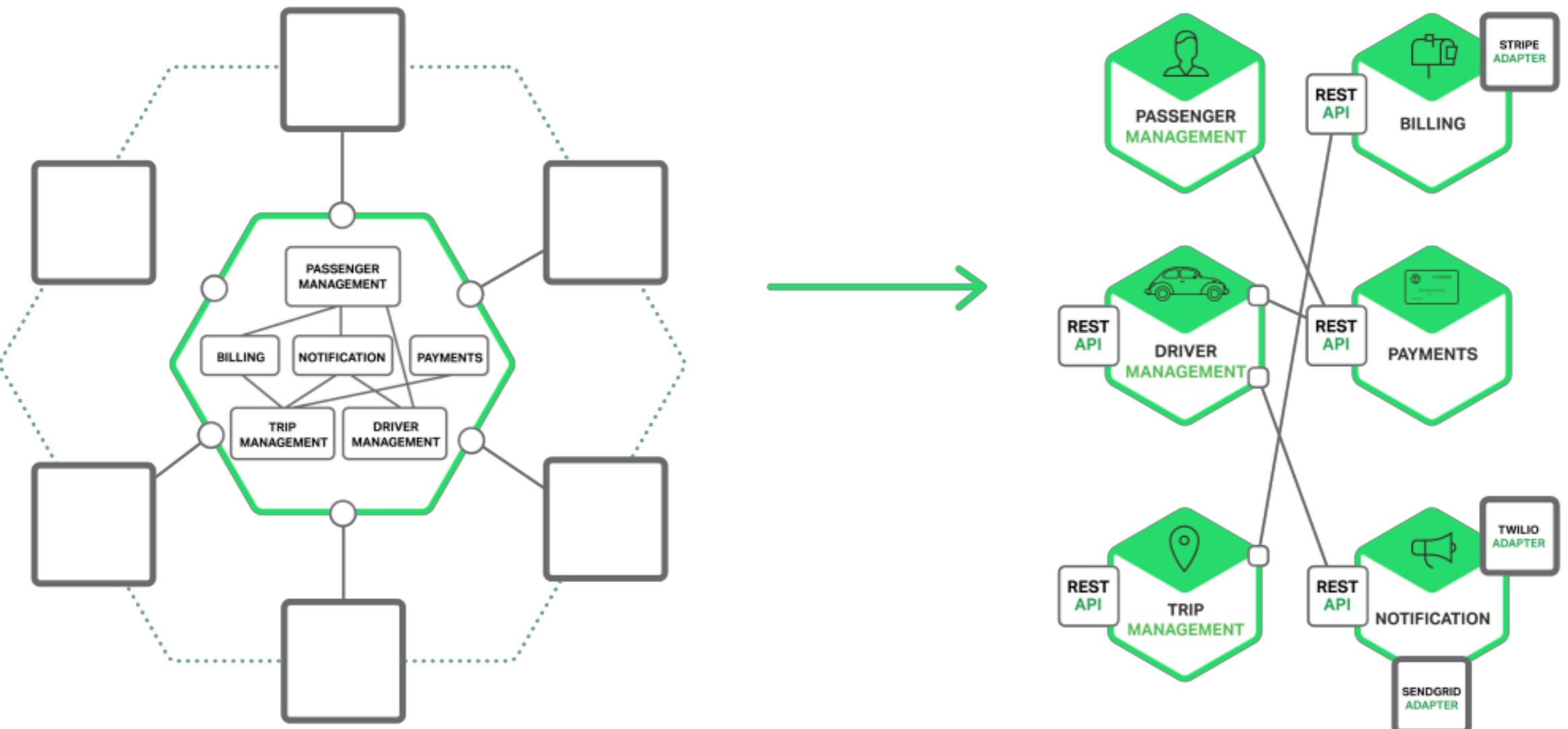
Reliability of communication

Transactional messaging

External API



Inter Process Communication (IPC)



<https://www.nginx.com>

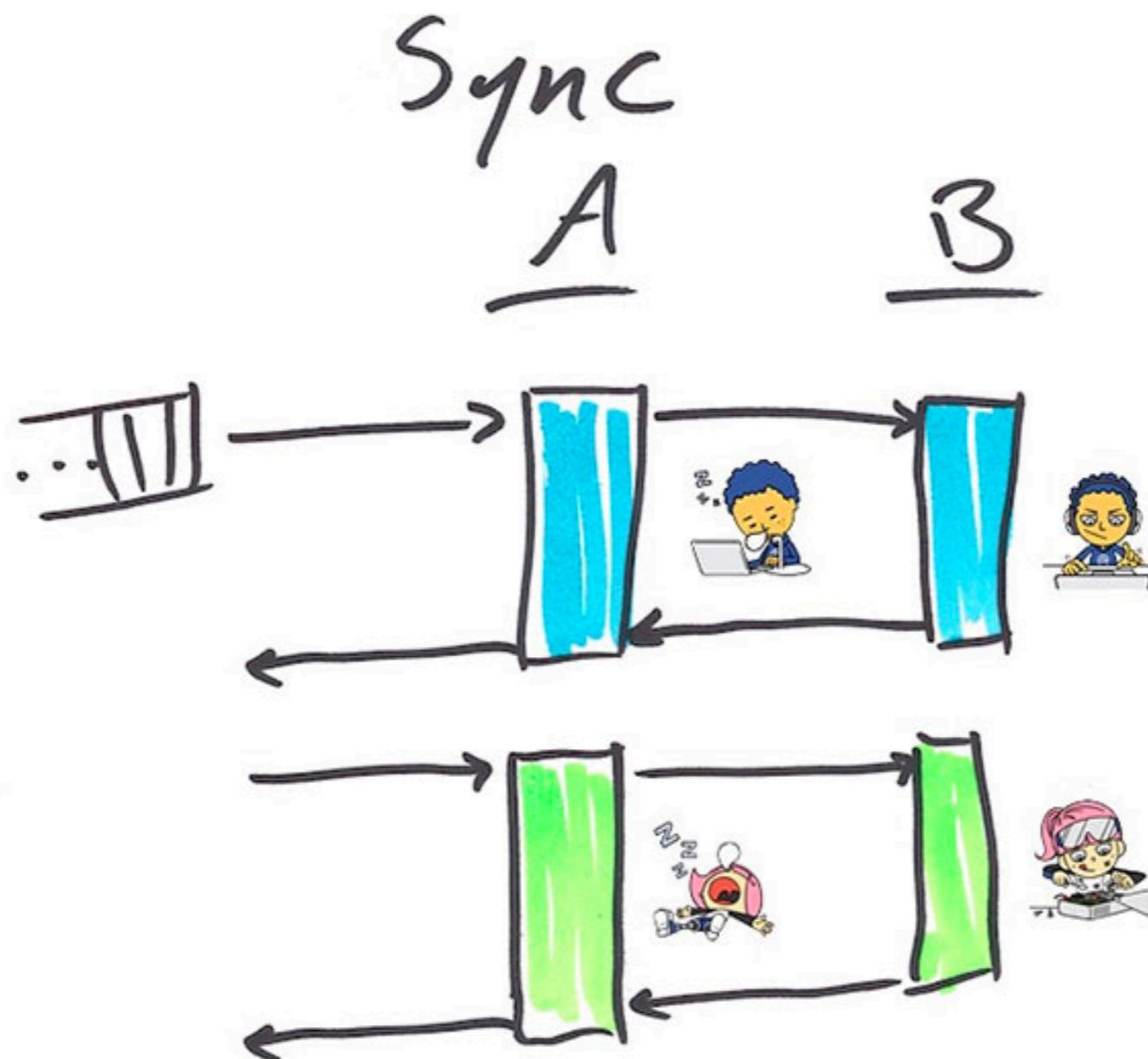


Interaction Styles

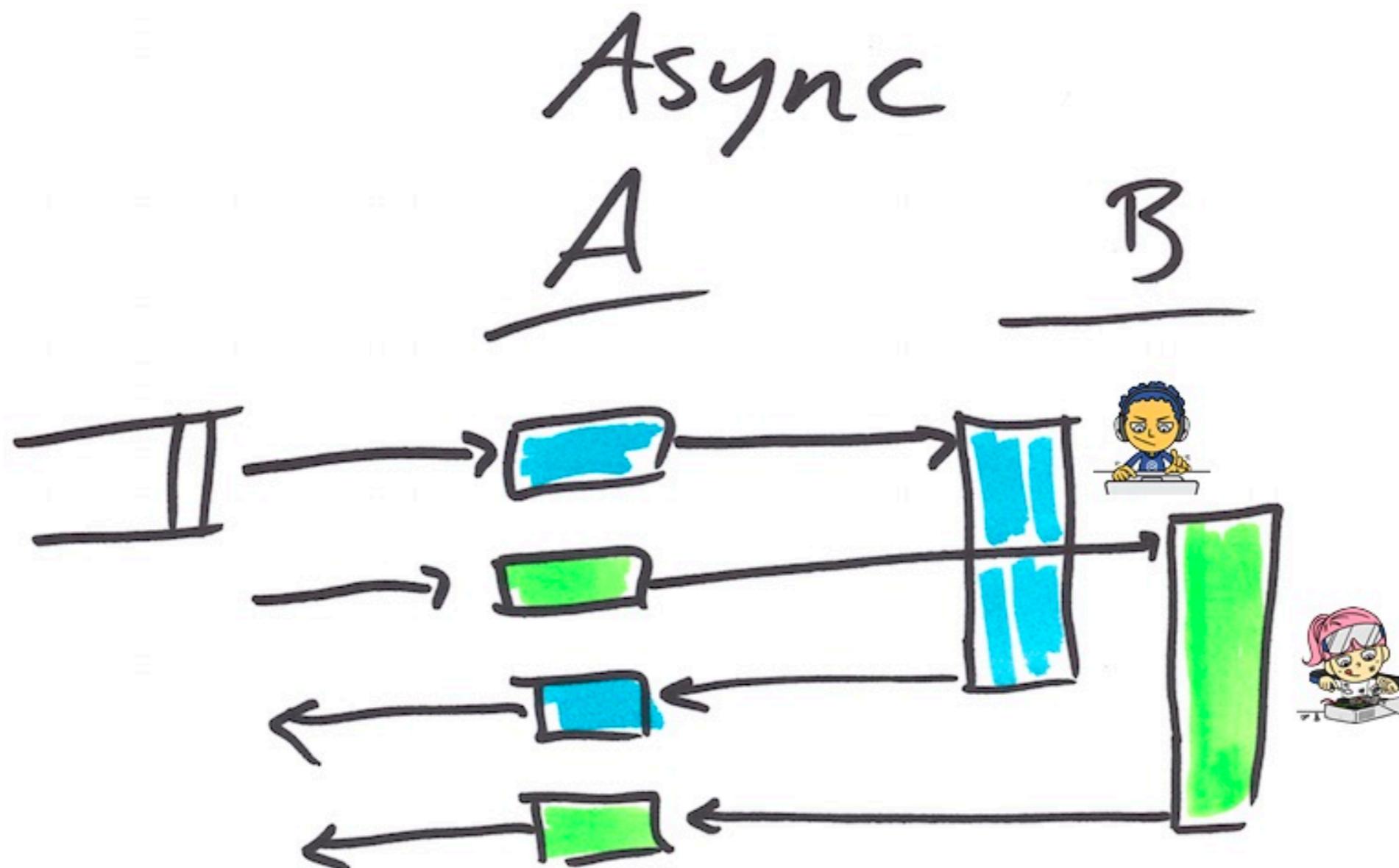
| | One-to-one | One-to-many |
|---------------------|------------------------|------------------------|
| Synchronous | Request/response | - |
| Asynchronous | Async request/response | Publish/subscribe |
| | Notification | Publish/async response |



Synchronous



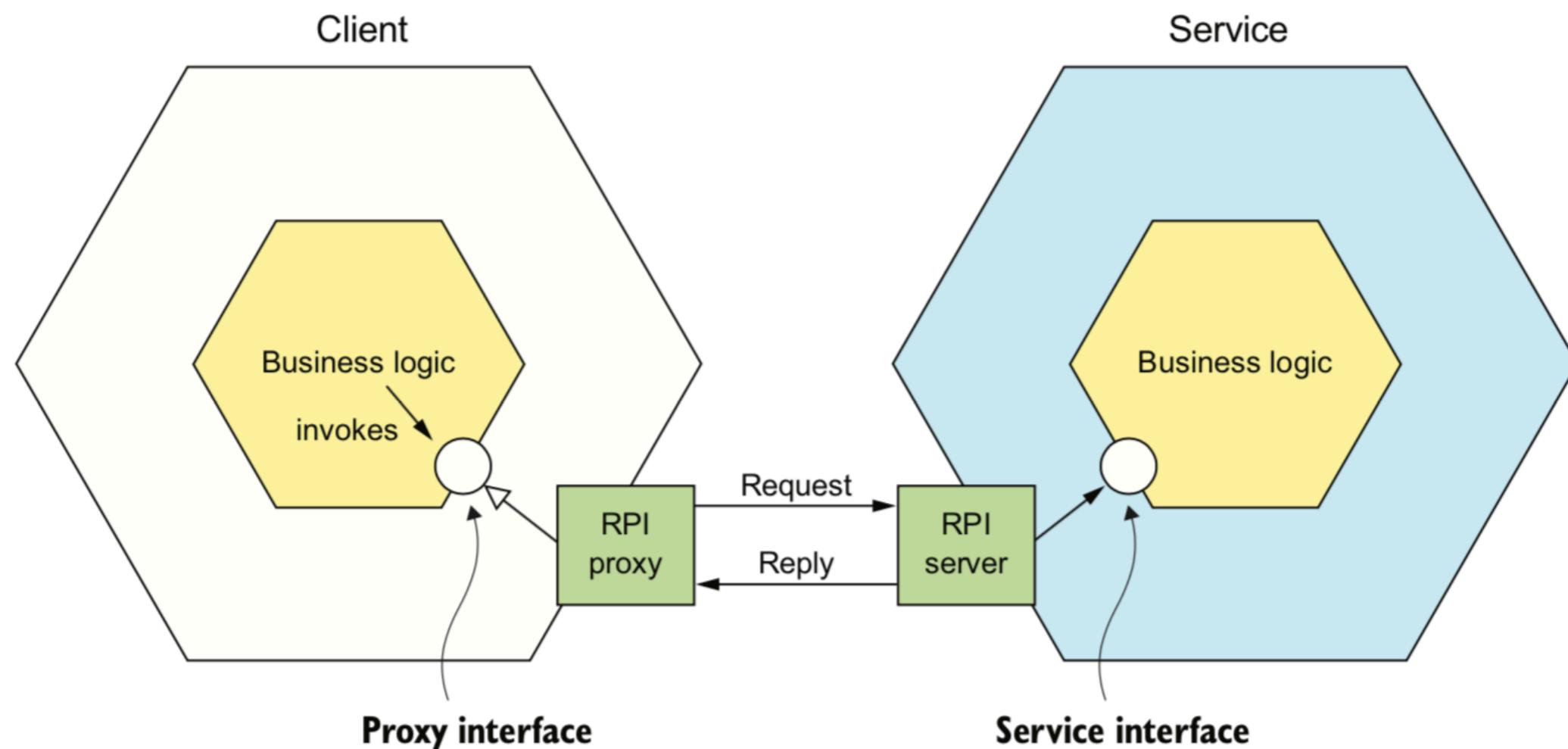
Asynchronous



Synchronous Remote Procedure Invocation



Synchronous Remote Procedure Invocation



Synchronous Remote Procedure Invocation

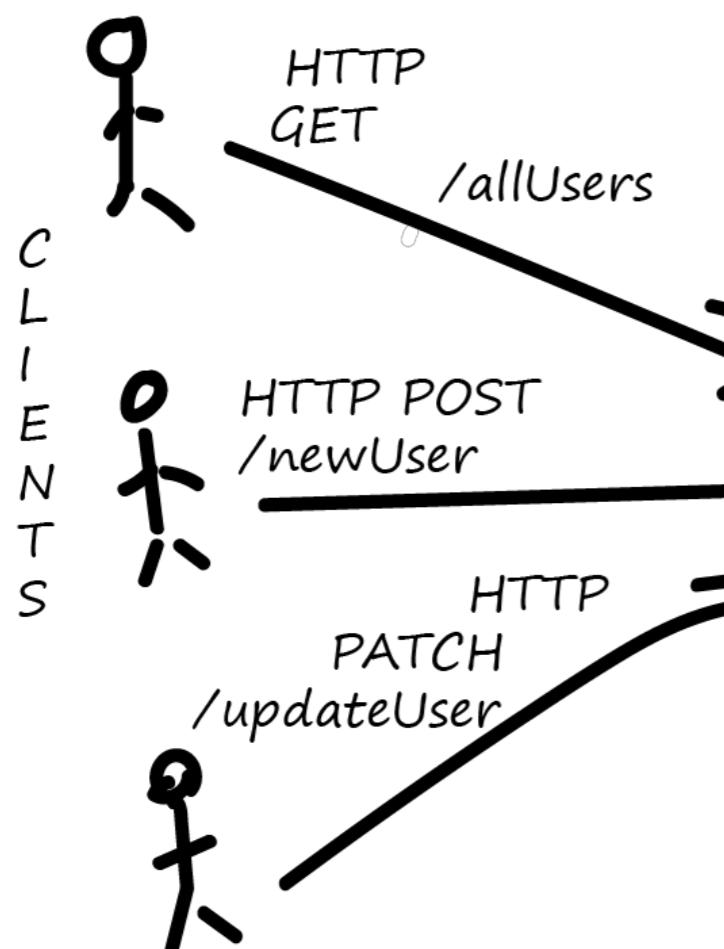
REST
gRPC

Handling failure with Circuit breaker
Service discovery



REpresentational State Transfer

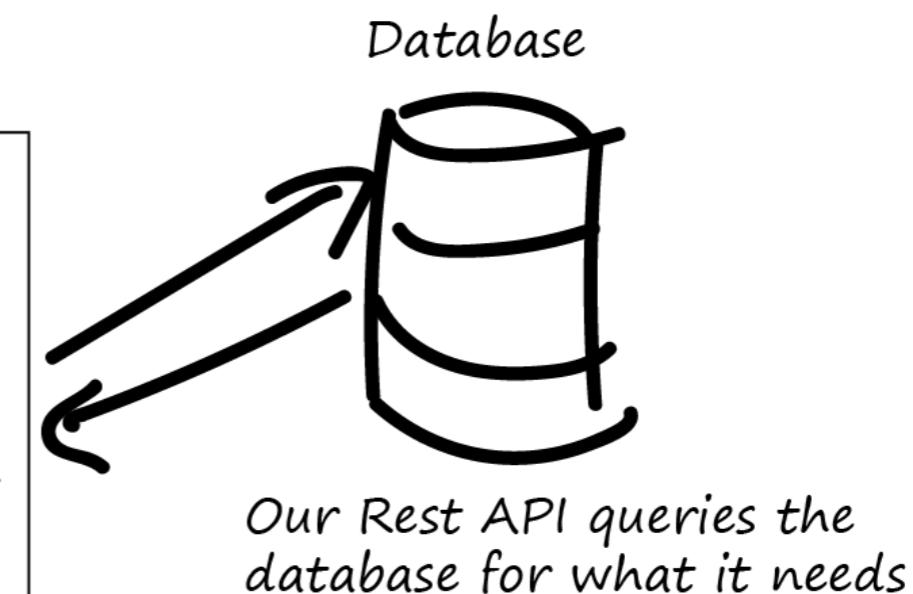
Rest API Basics



Our Clients, send HTTP Requests and wait for responses

Rest API
Receives HTTP requests from Clients and does whatever request needs. i.e create users

Typical HTTP Verbs:
GET → Read from Database
PUT → Update/Replace row in Database
PATCH → Update/Modify row in Database
POST → Create a new record in the database
DELETE → Delete from the database



Response: When the Rest API has what it needs, it sends back a response to the clients. This would typically be in JSON or XML format.

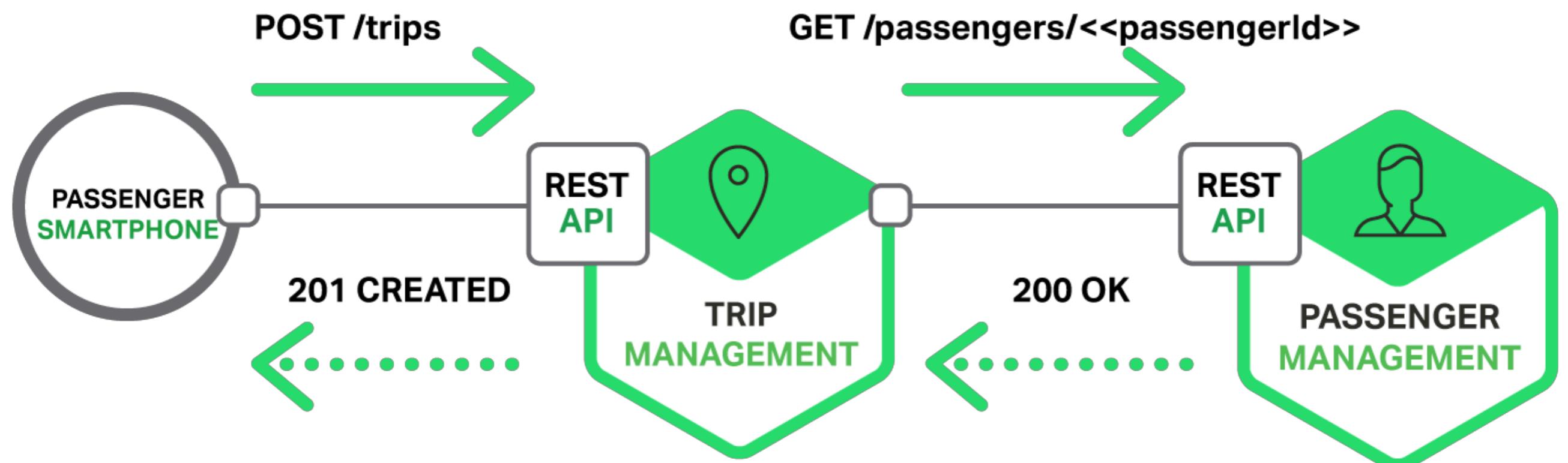


REST :: mapping with HTTP verbs

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



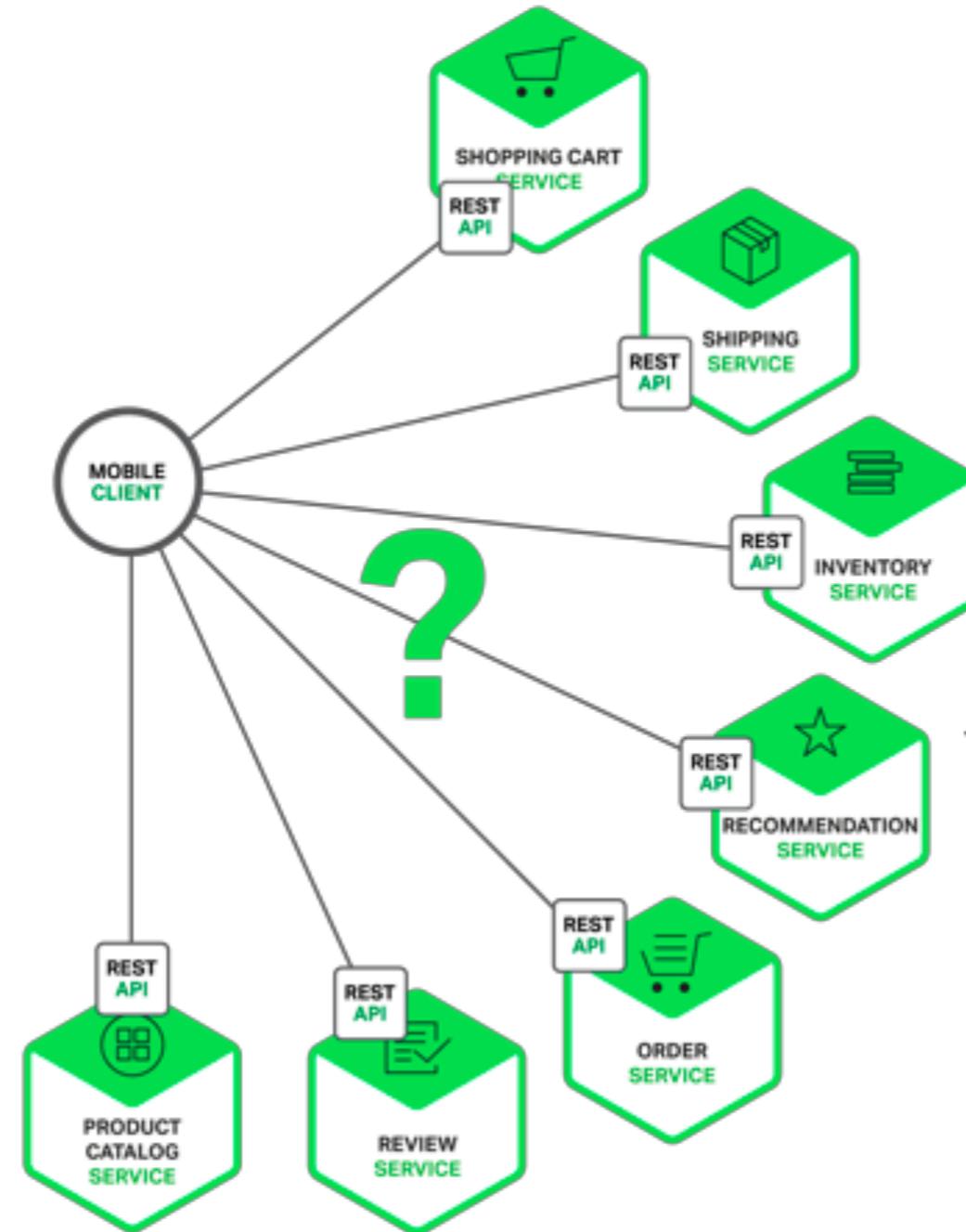
Request and response format



<https://www.nginx.com>



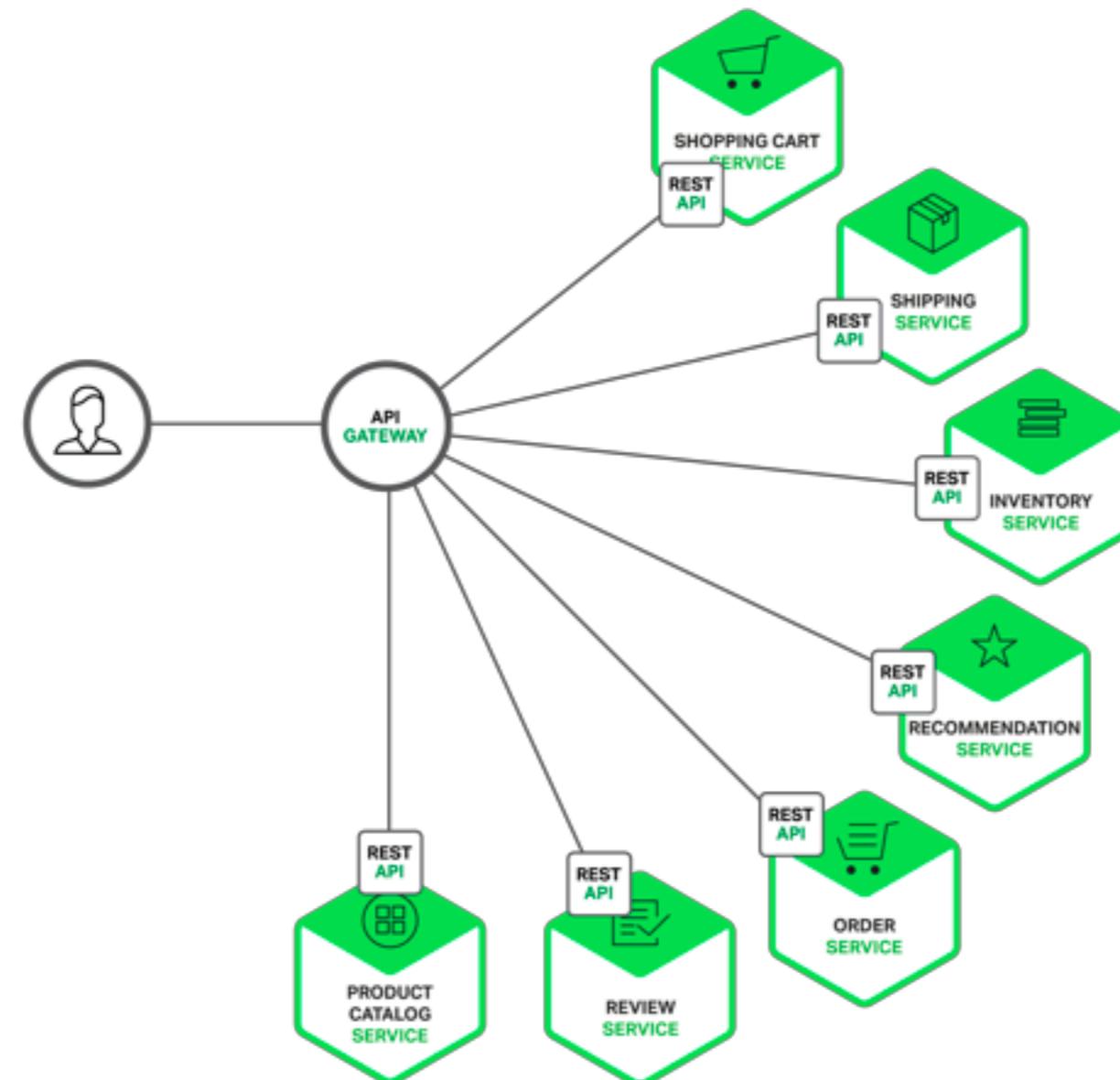
Direct communication !!



<https://www.nginx.com>



Working with API gateway



<https://www.nginx.com>



Functions of API gateway

Authentication

Authorization

Rate limiting

Caching

Metrics collection

Request logging



Benefits

Encapsulation interface structure

Client talk to service via gateway

Reduce round trip between service



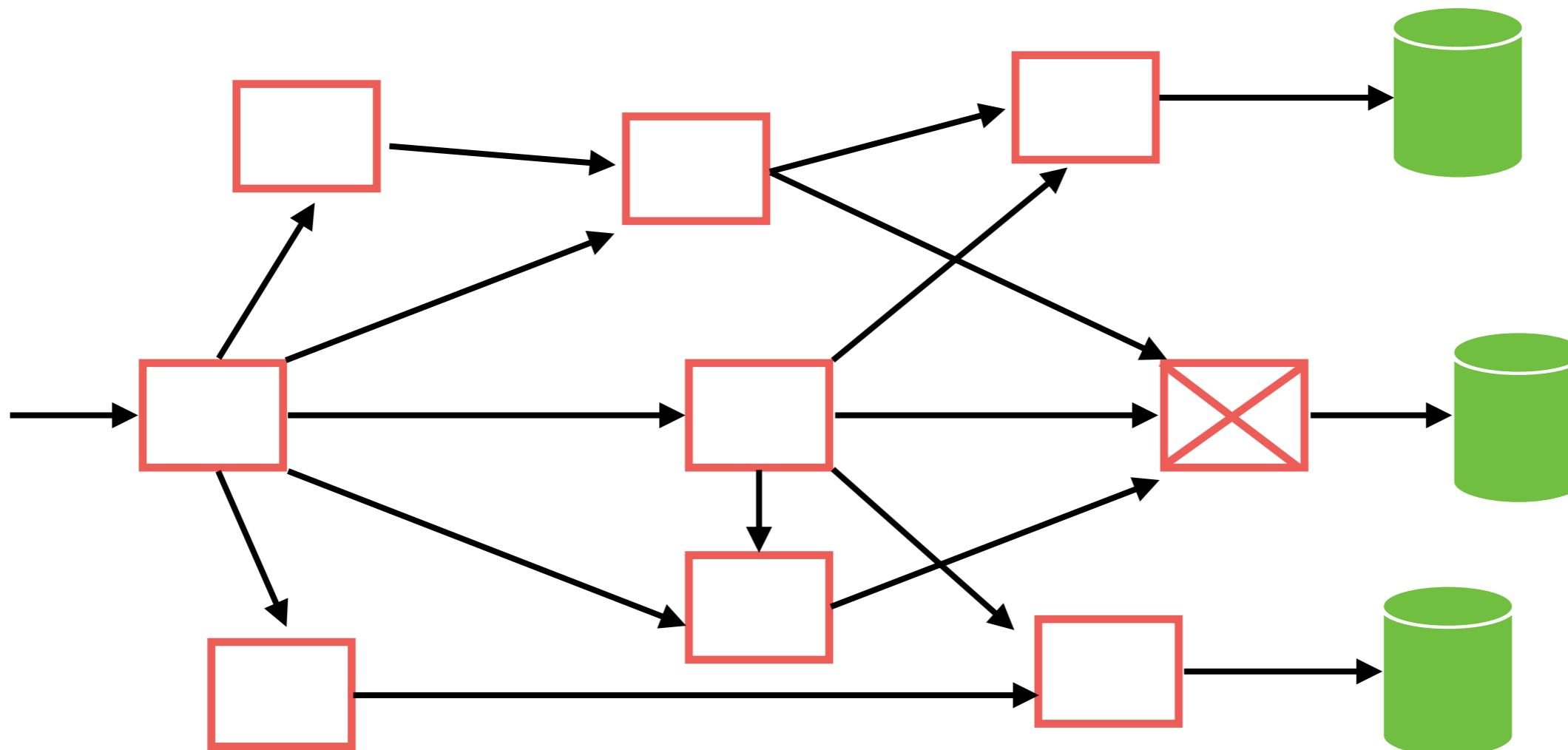
Drawbacks

Development/Performance bottleneck
Single point of failure
Waiting for update data in gateway



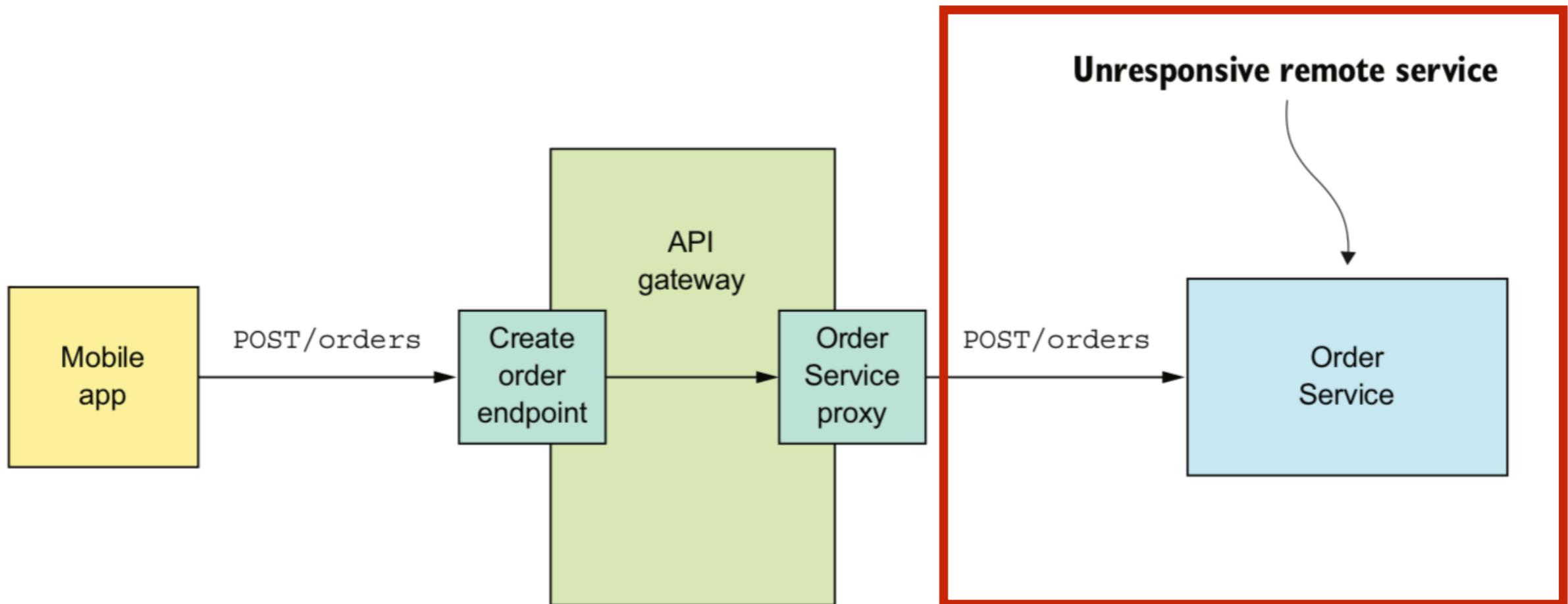
Circuit Breaker pattern

How to handling partial failure ?



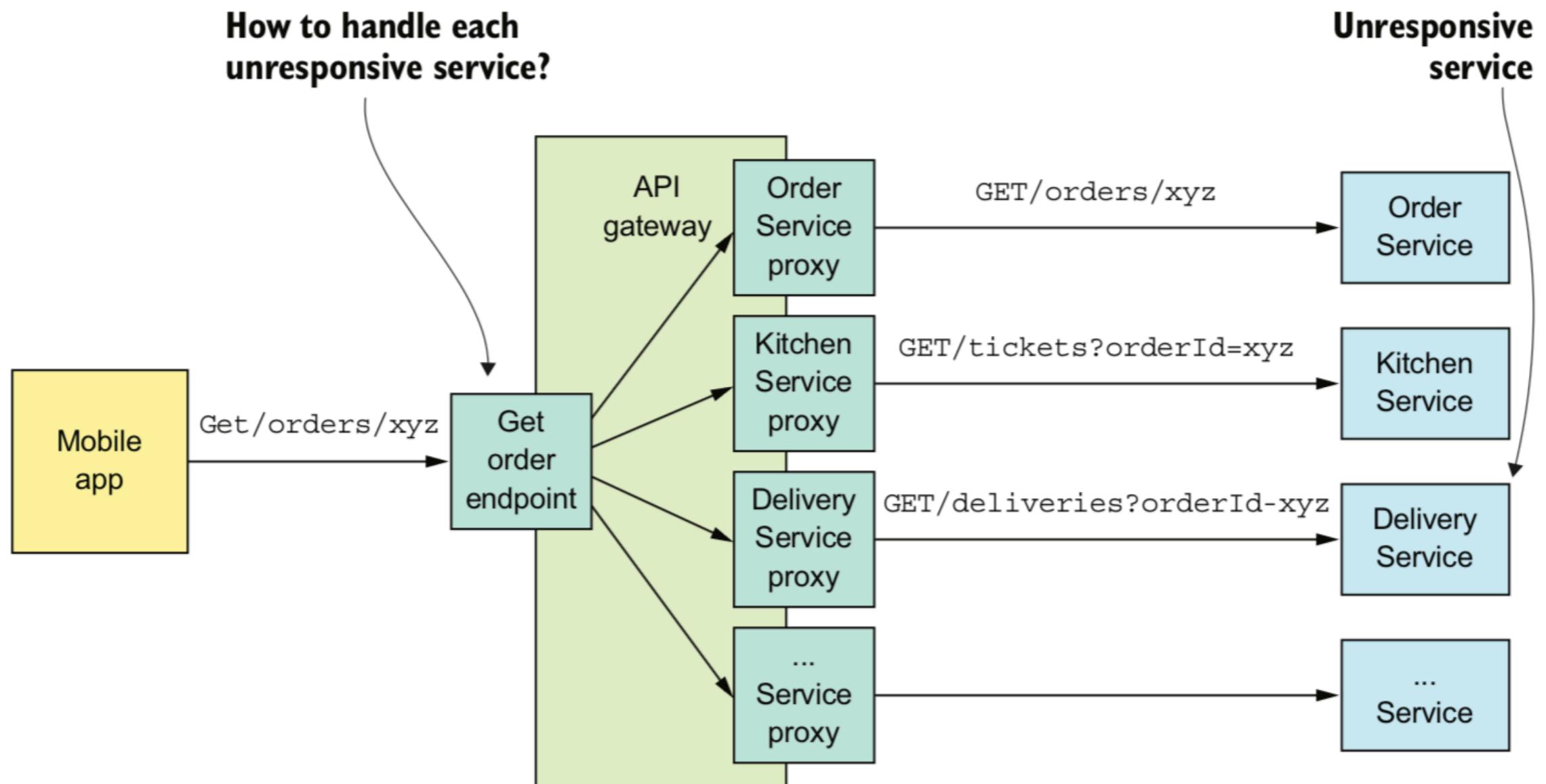
Circuit Breaker pattern

How to handling partial failure ?



Circuit Breaker pattern

How to handling partial failure ?



Develop robust patterns

Network timeout

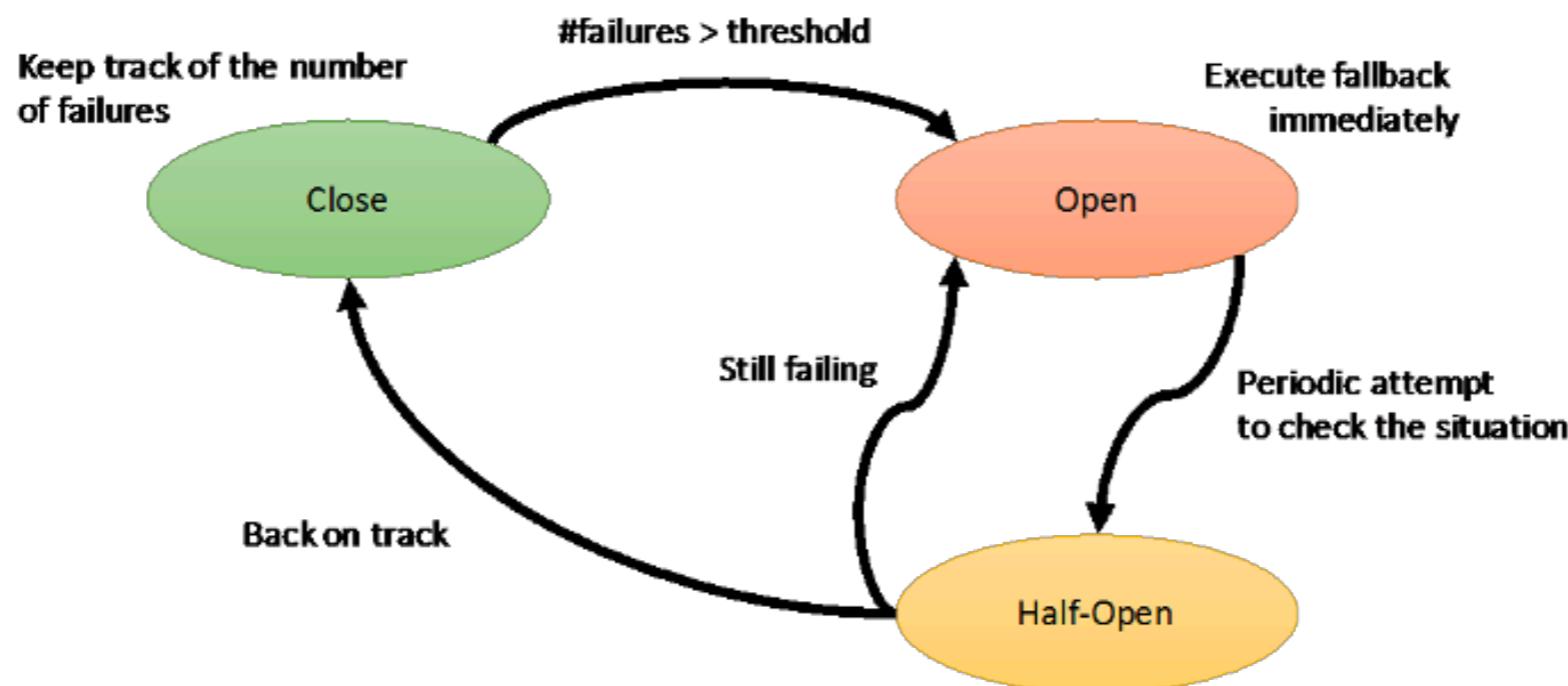
Limit request from client to service
Circuit breaker pattern



Circuit Breaker pattern

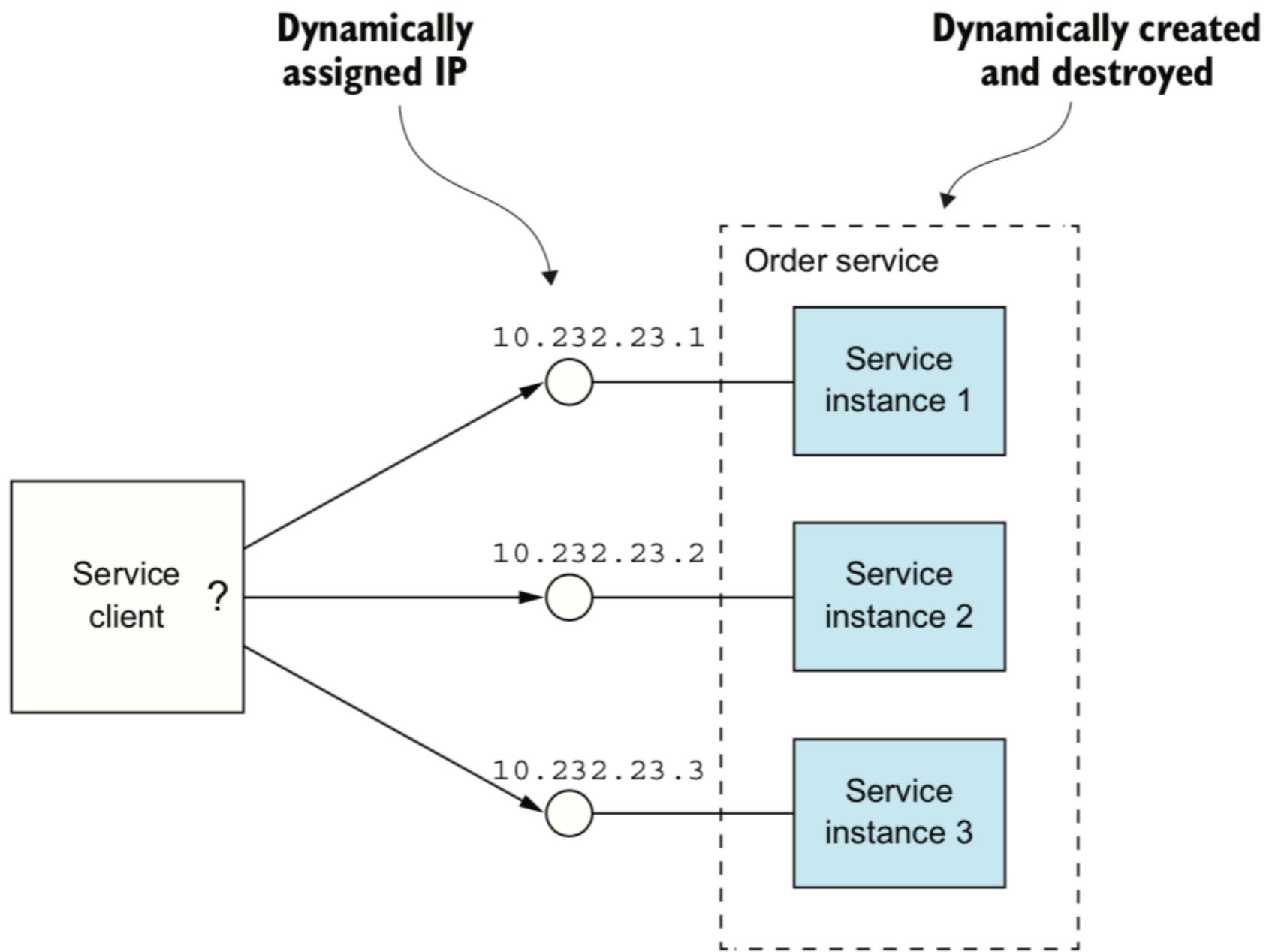
Track the number of success and failure

***If error rate exceed some threshold
then enable circuits breaker***

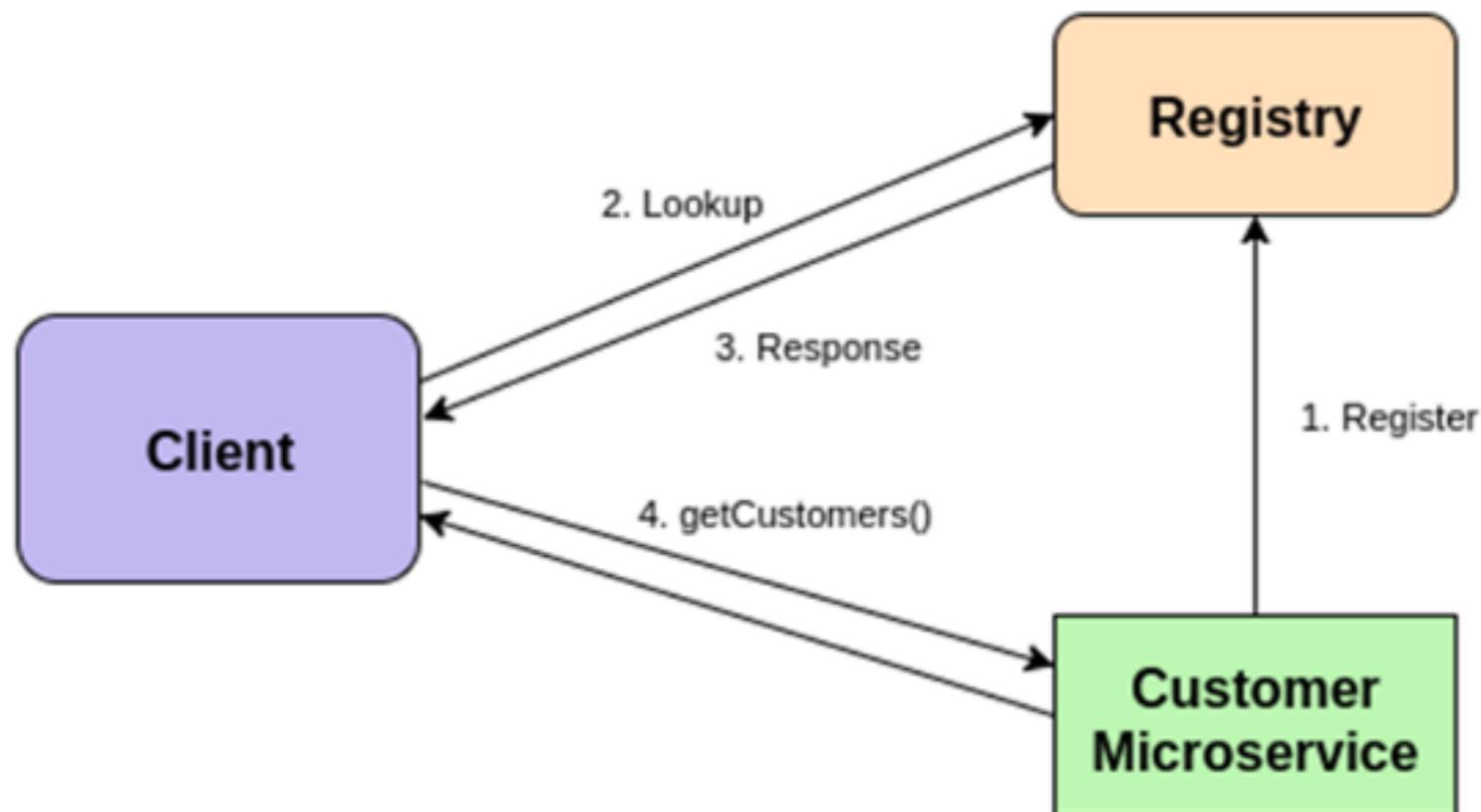


Service Discovery

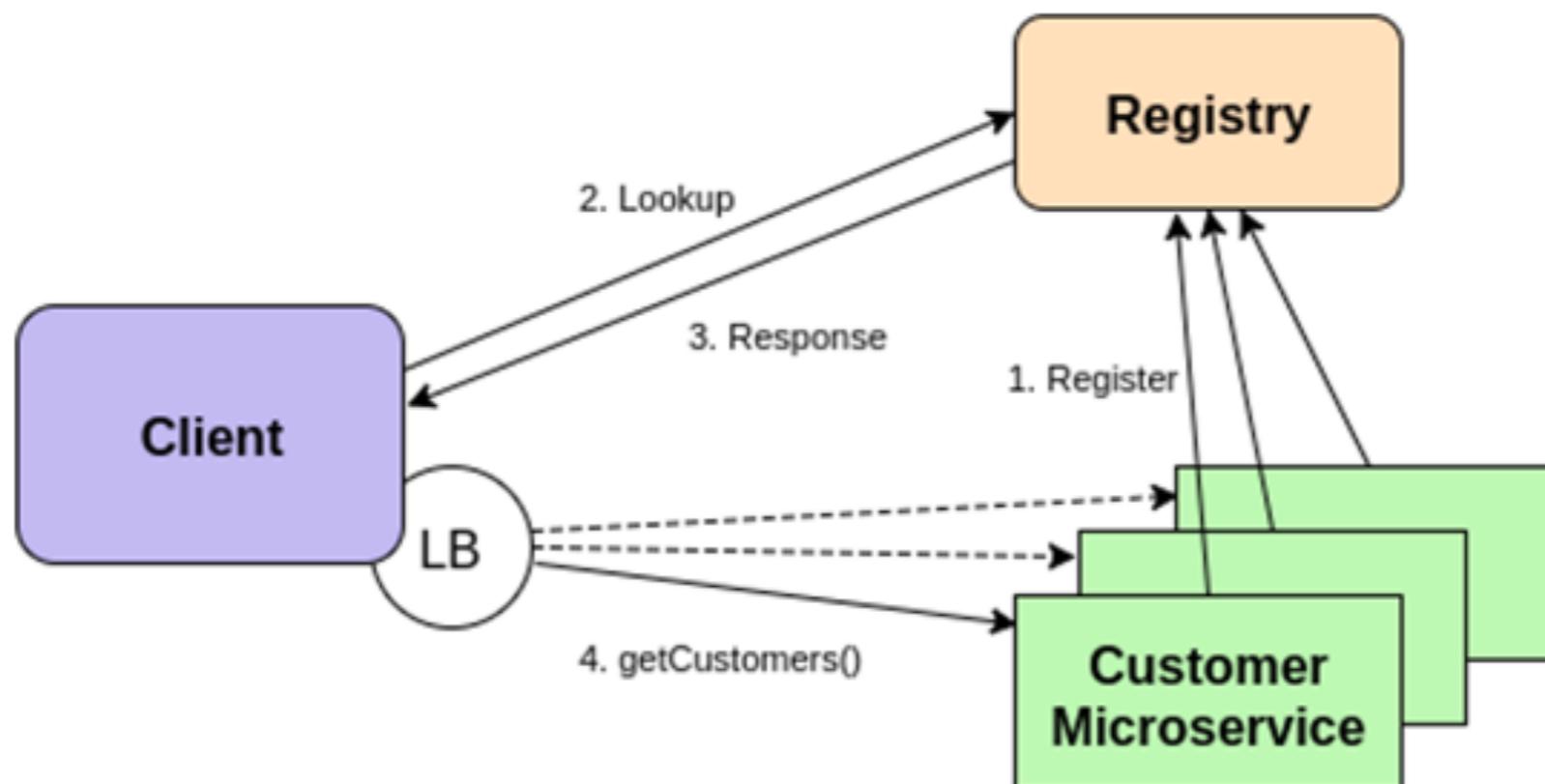
How to call another services ?



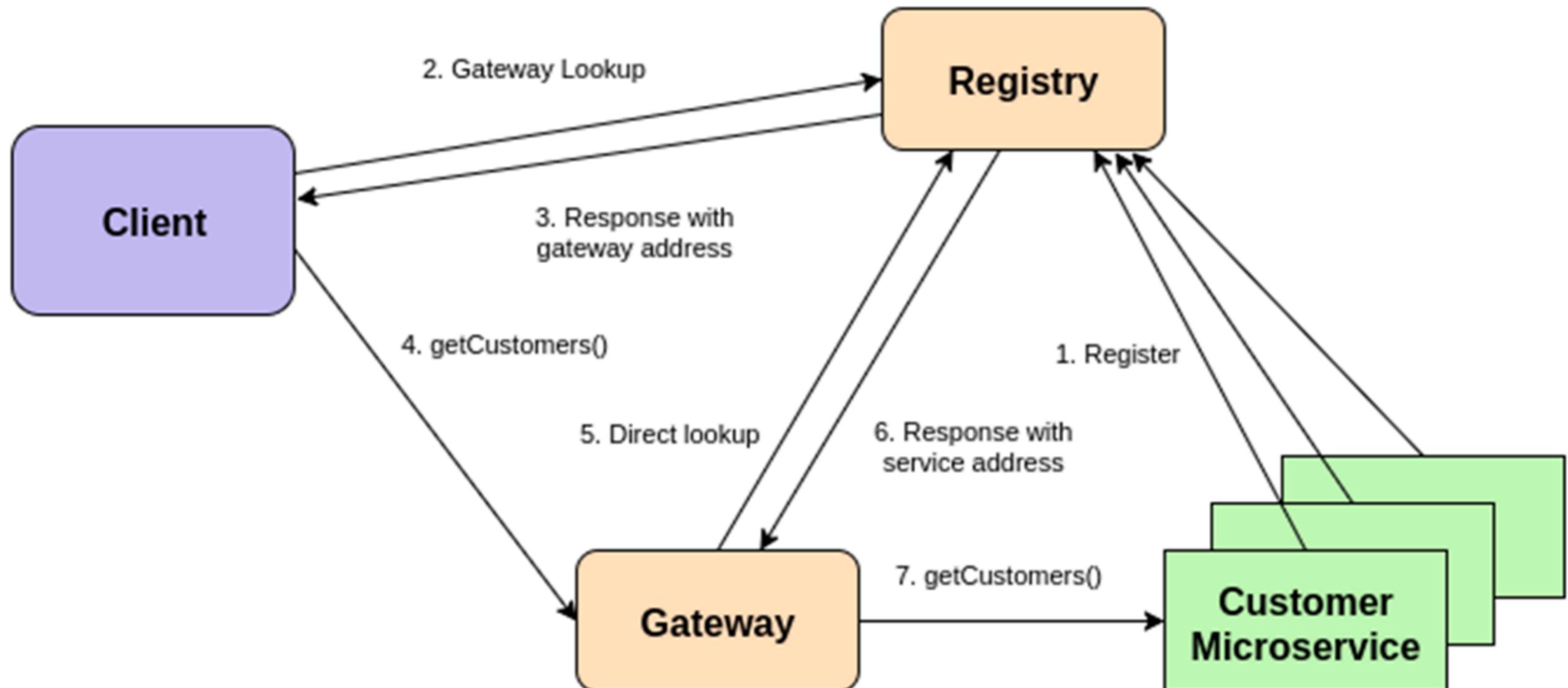
Service Discovery



Service discovery with Load balance



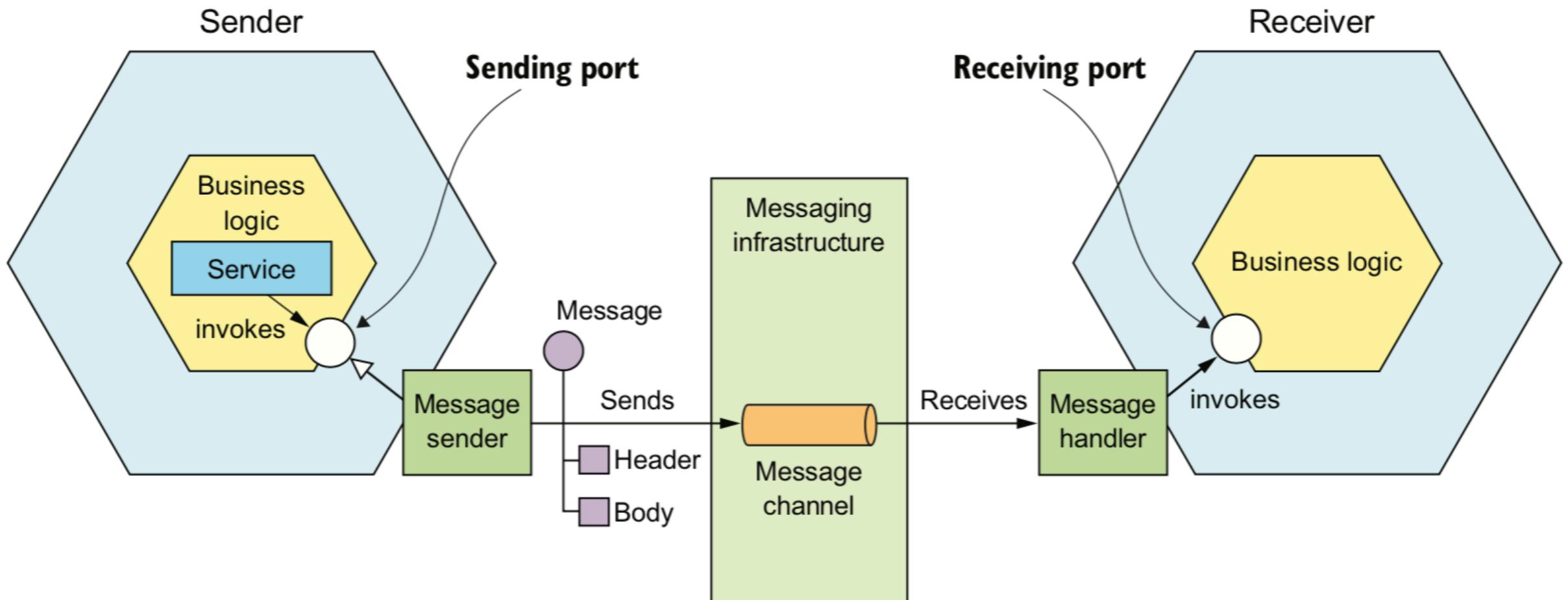
Service discovery with API gateway



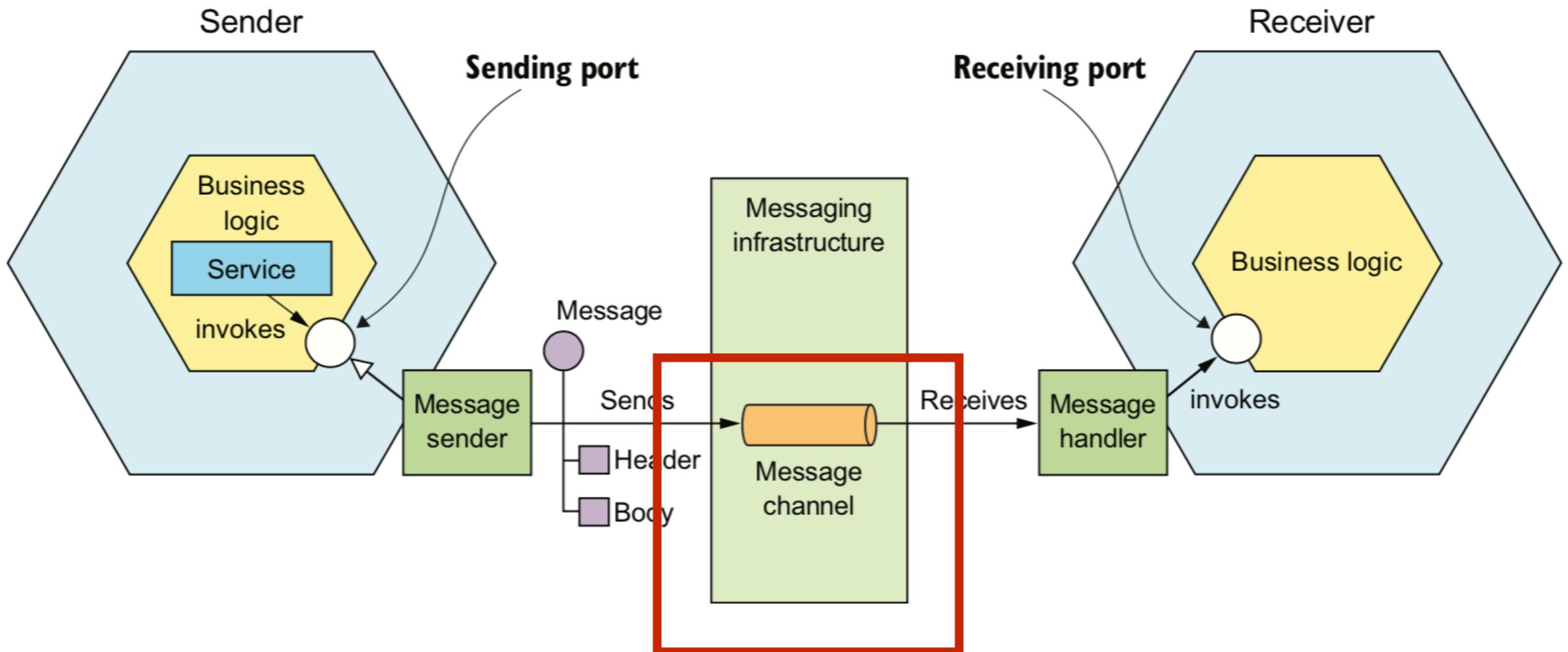
Asynchronous messaging pattern



Asynchronous messaging pattern

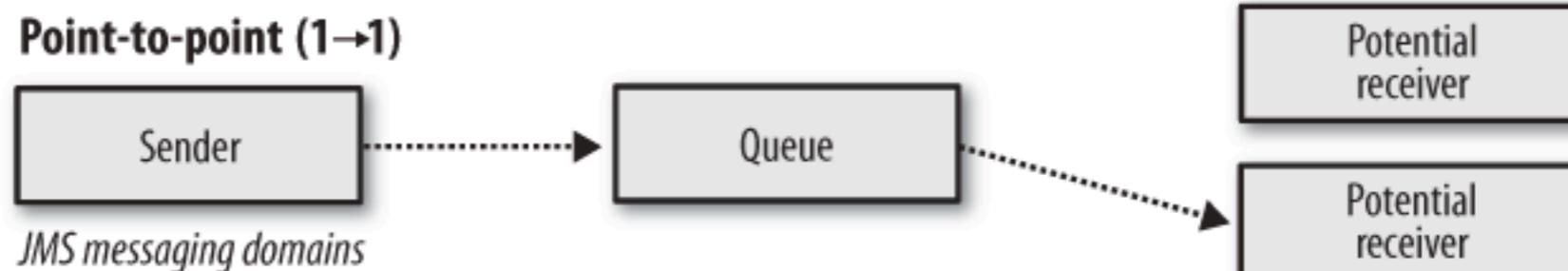
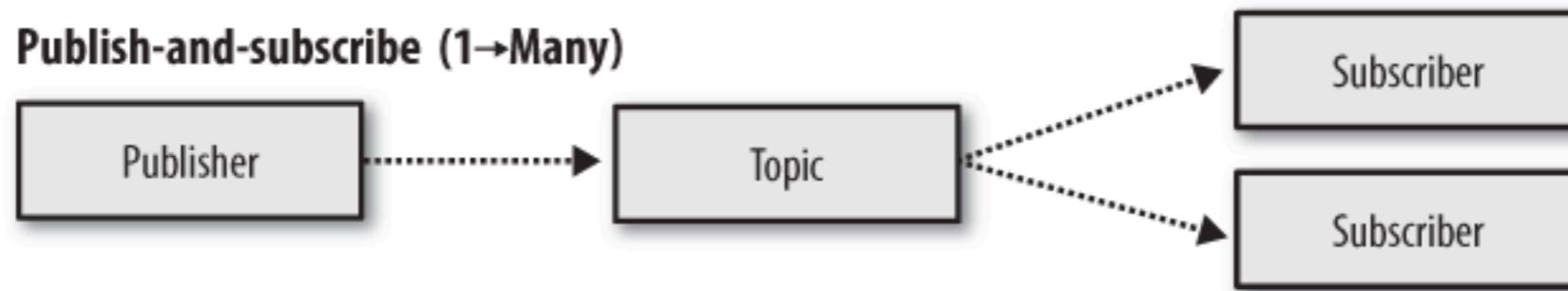


Asynchronous messaging pattern



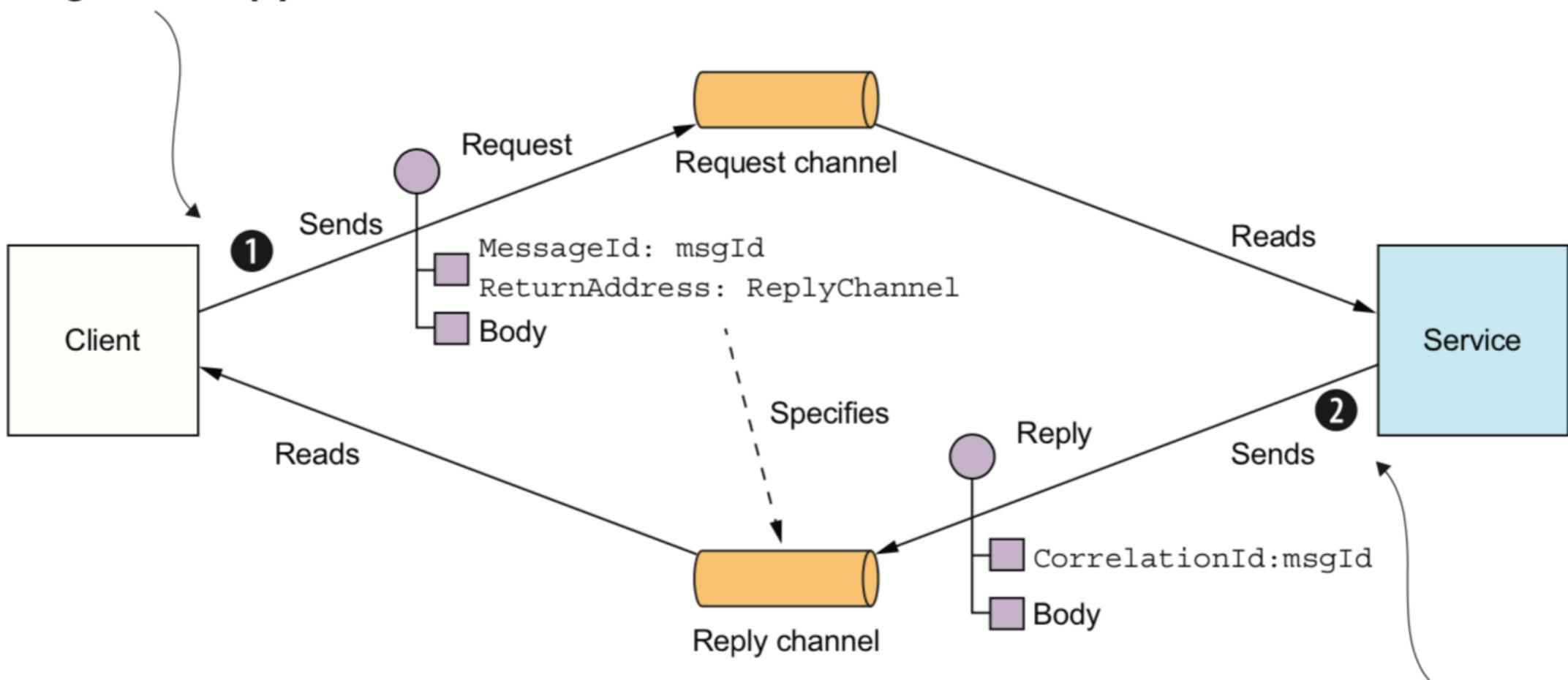
Messaging channels

Point-to-point Publish-subscribe



Async request/response (1)

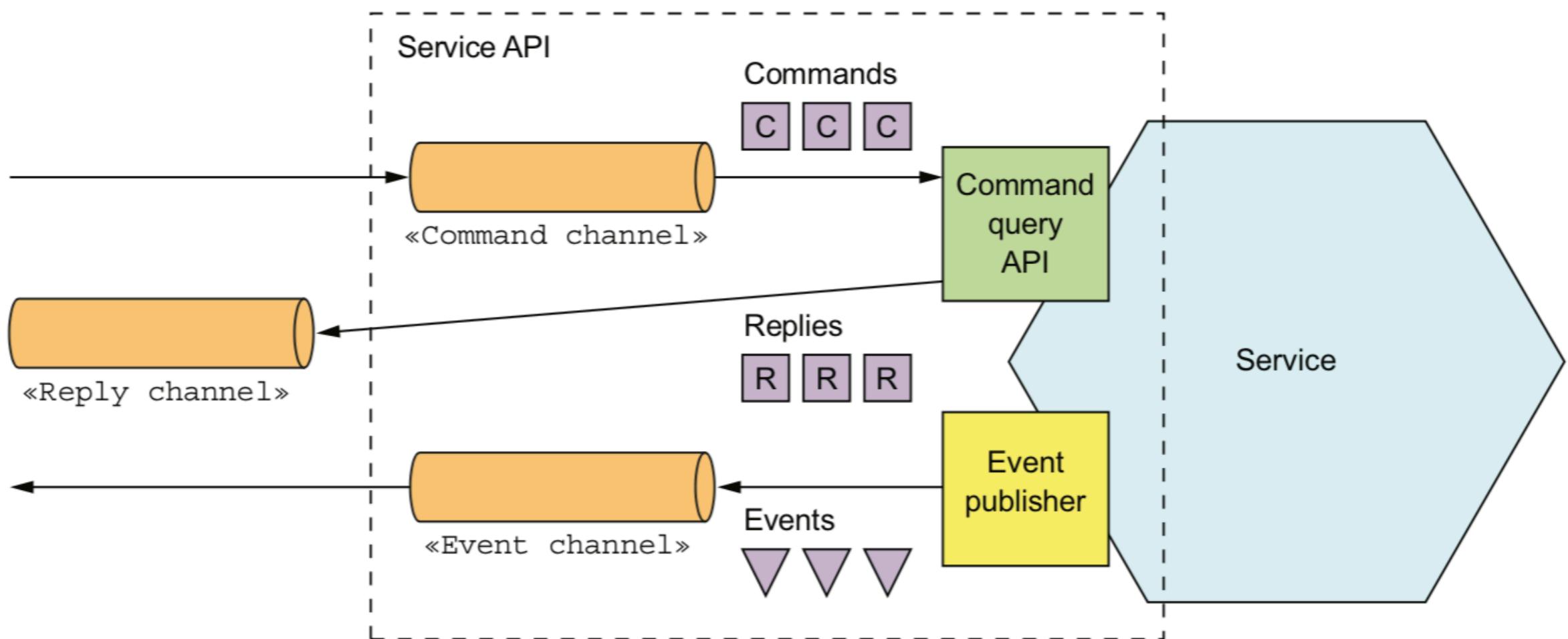
Client sends message containing msgId and a reply channel.



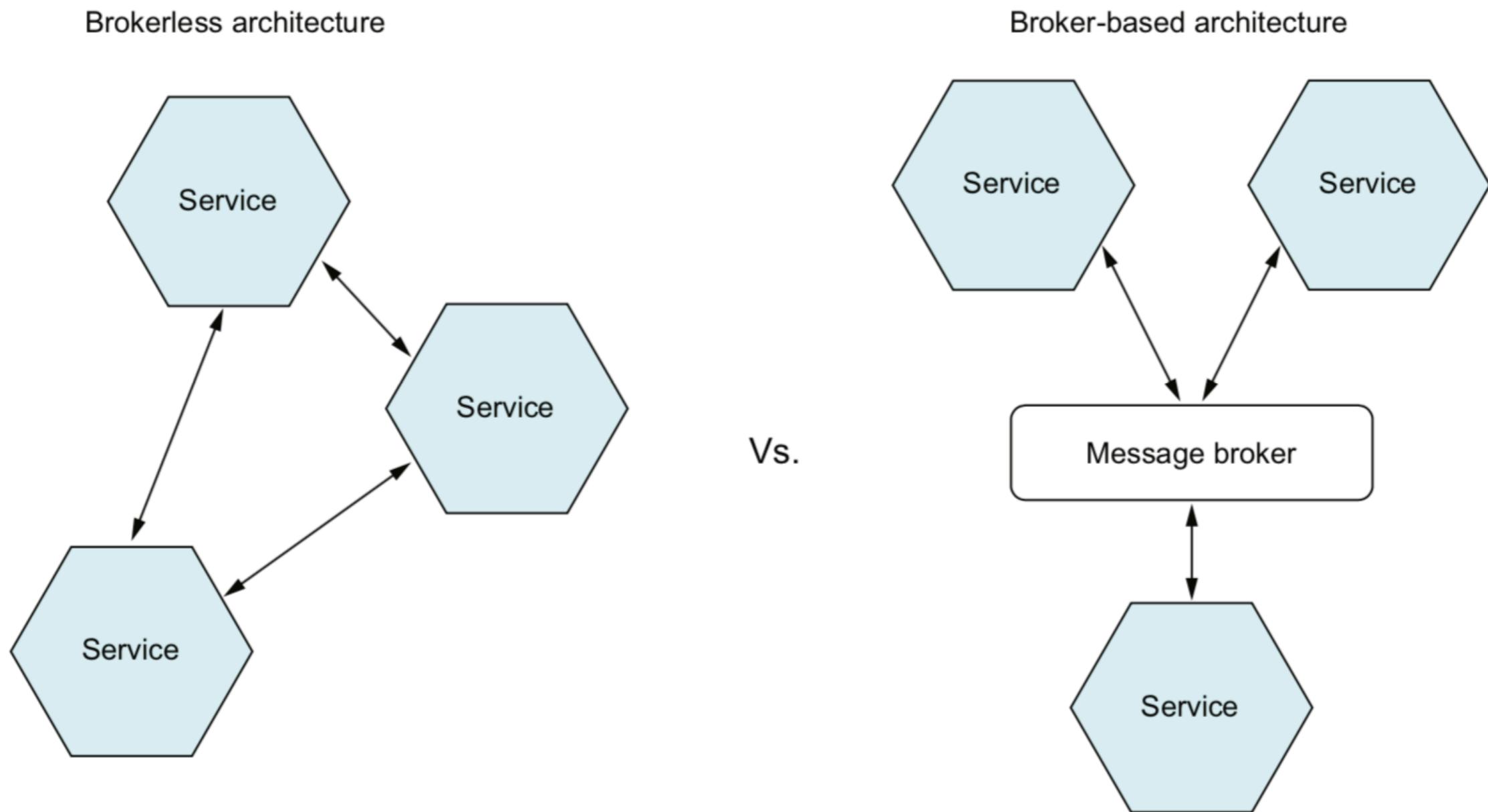
Service sends reply to the specified reply channel. The reply contains a correlationId, which is the request's msgId.



Async request/response (2)



Messaging-based use message broker



Message broker



Benefits

Loose-coupling
Message buffer
Flexible communication



Drawbacks

Performance bottleneck
Single point of failure
Operational complexity

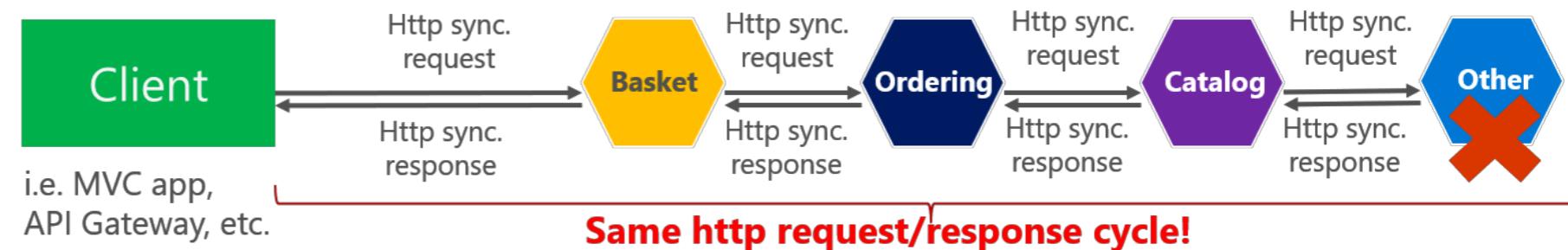


Communication

Synchronous vs. async communication across microservices

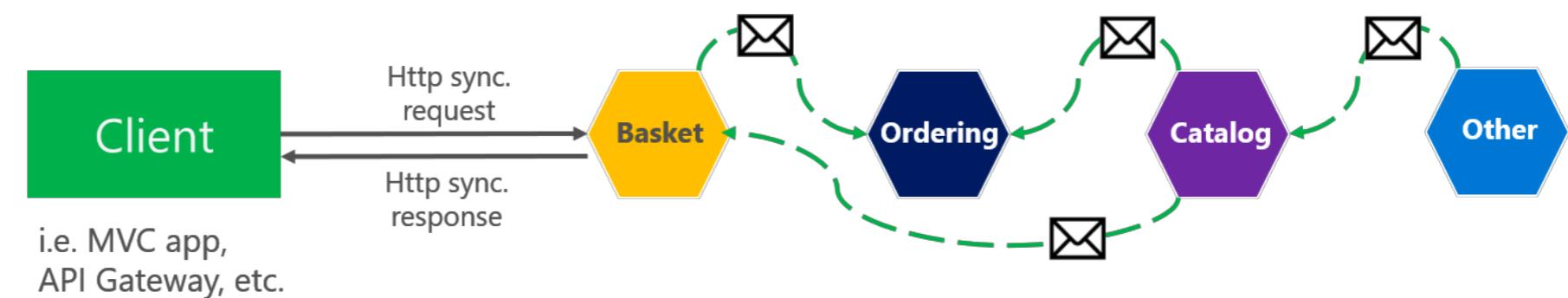
Anti-pattern

Synchronous
all req./resp. cycle



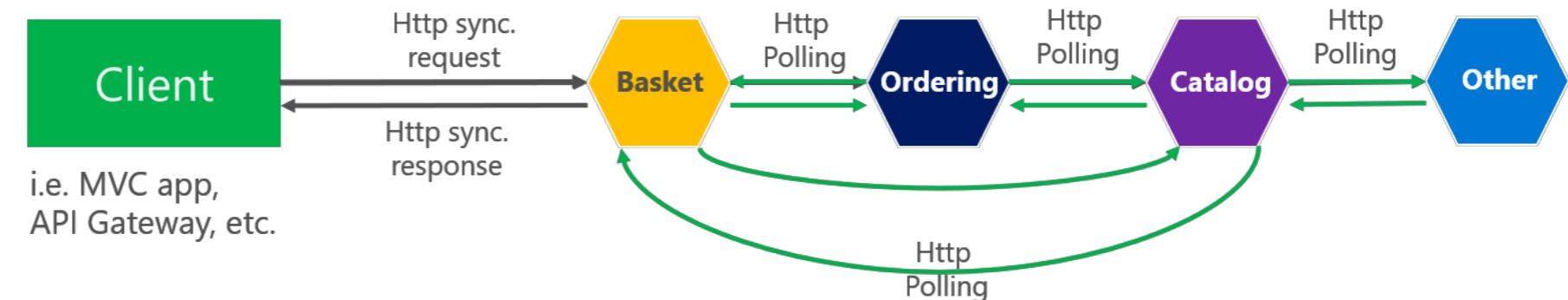
Asynchronous

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

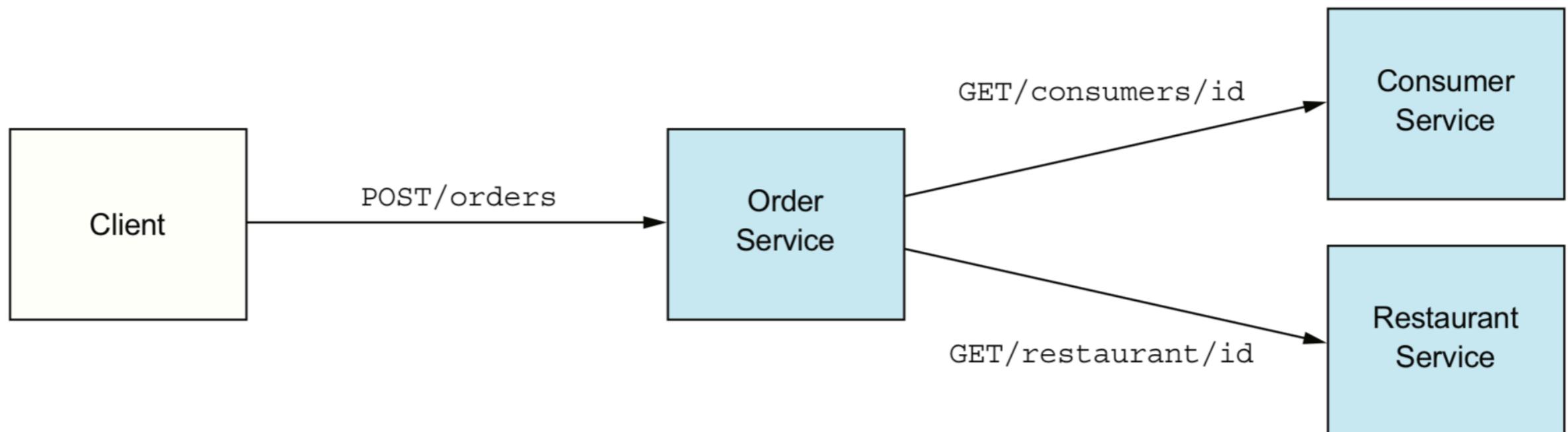


"Asynchronous"

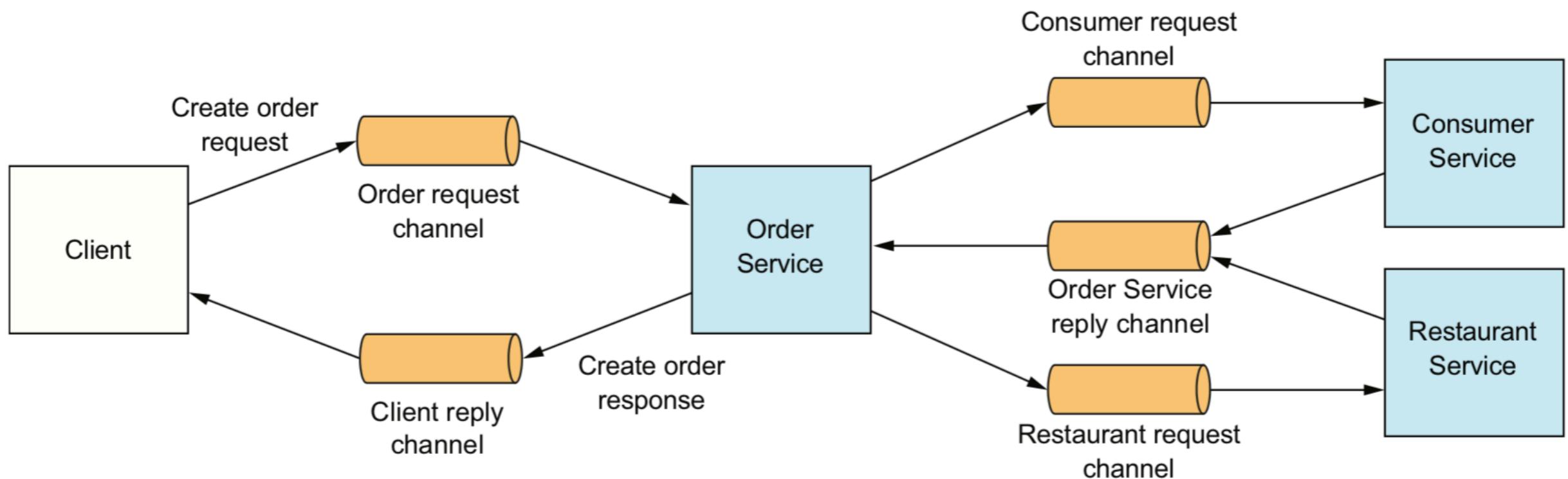
Comm. across
internal microservices
(Polling: **Http**)



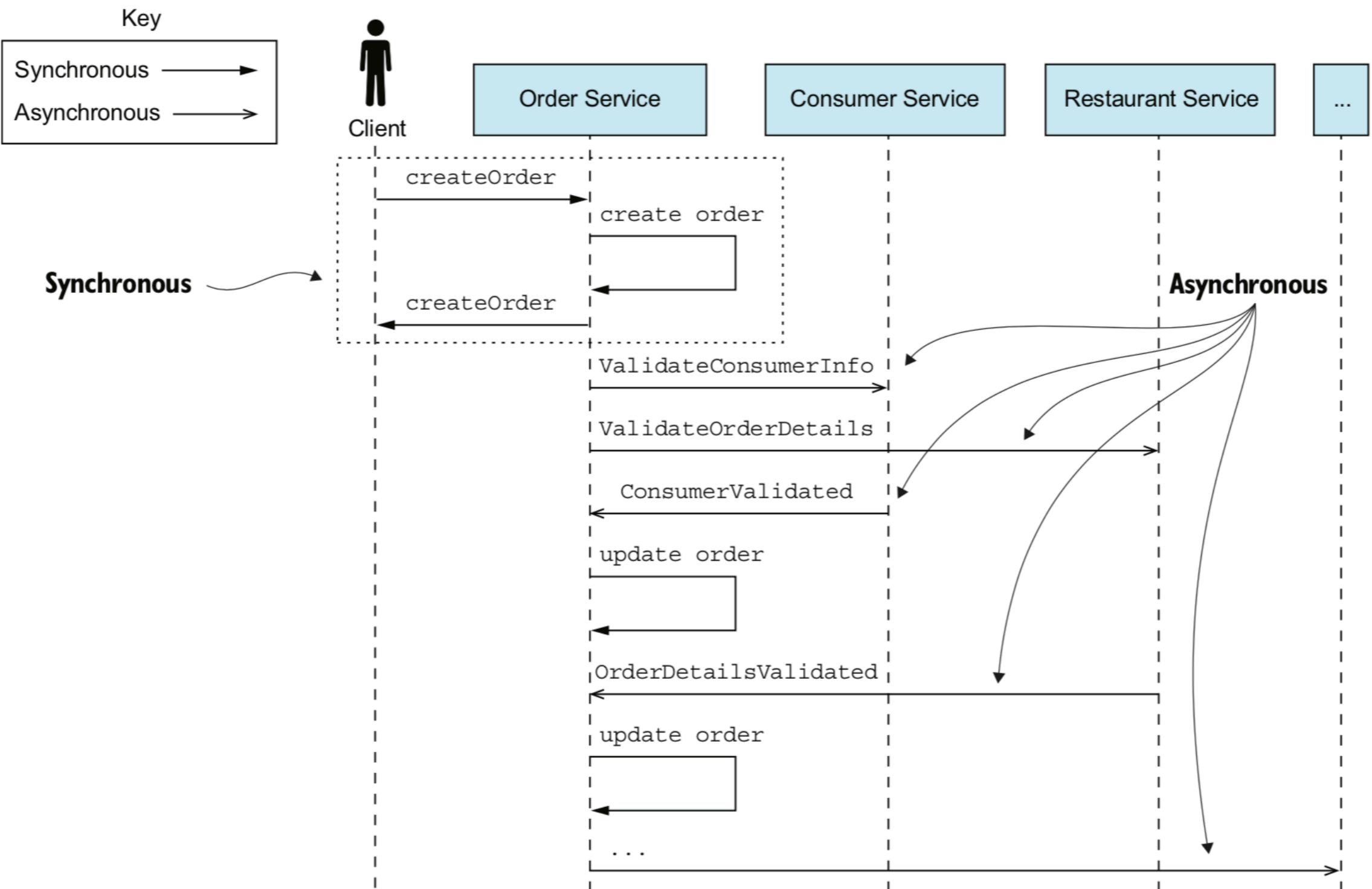
Synchronous reduce availability



Replace with asynchronous (1)



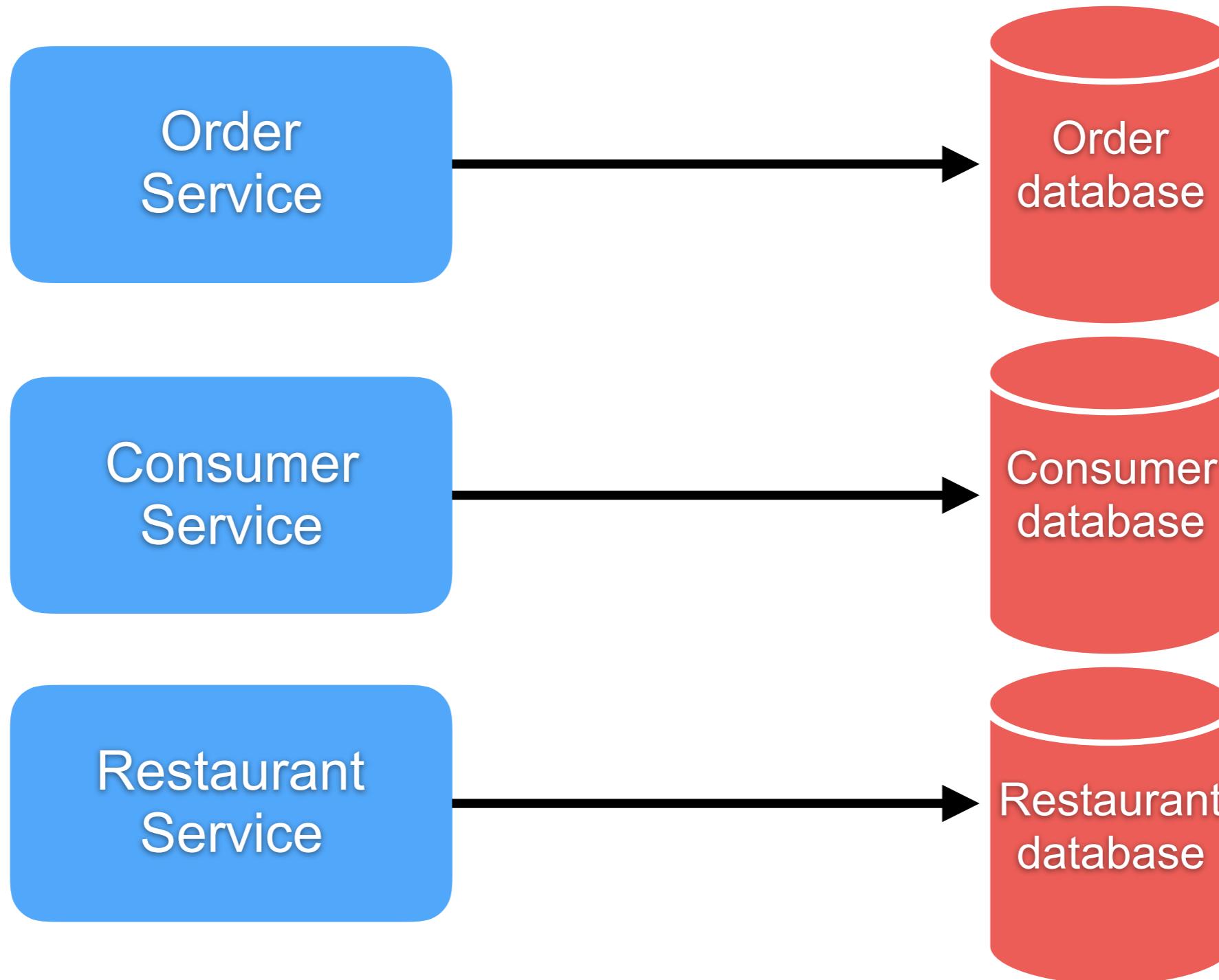
Replace with asynchronous (2)



Manage data consistency



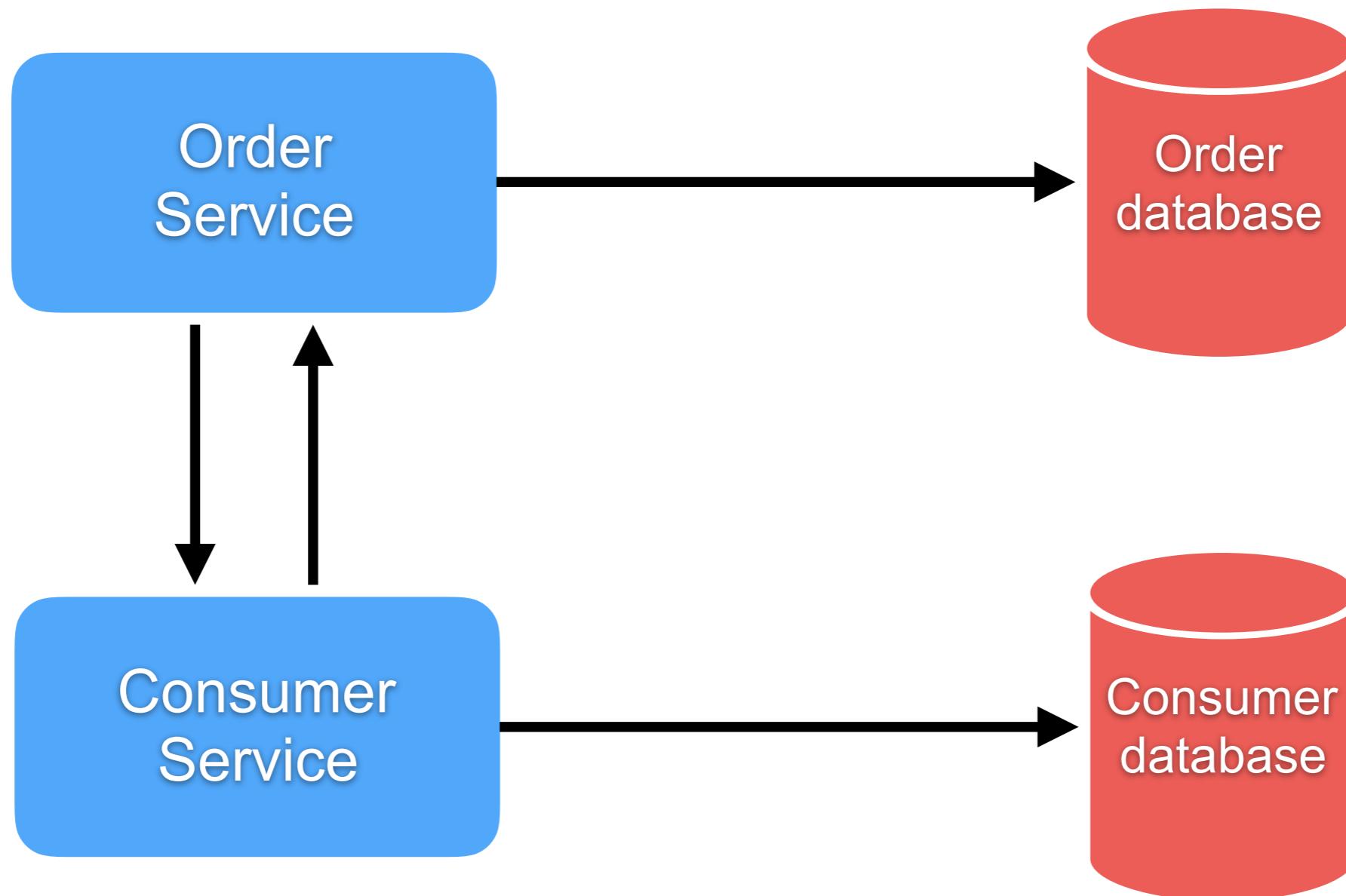
Database per service



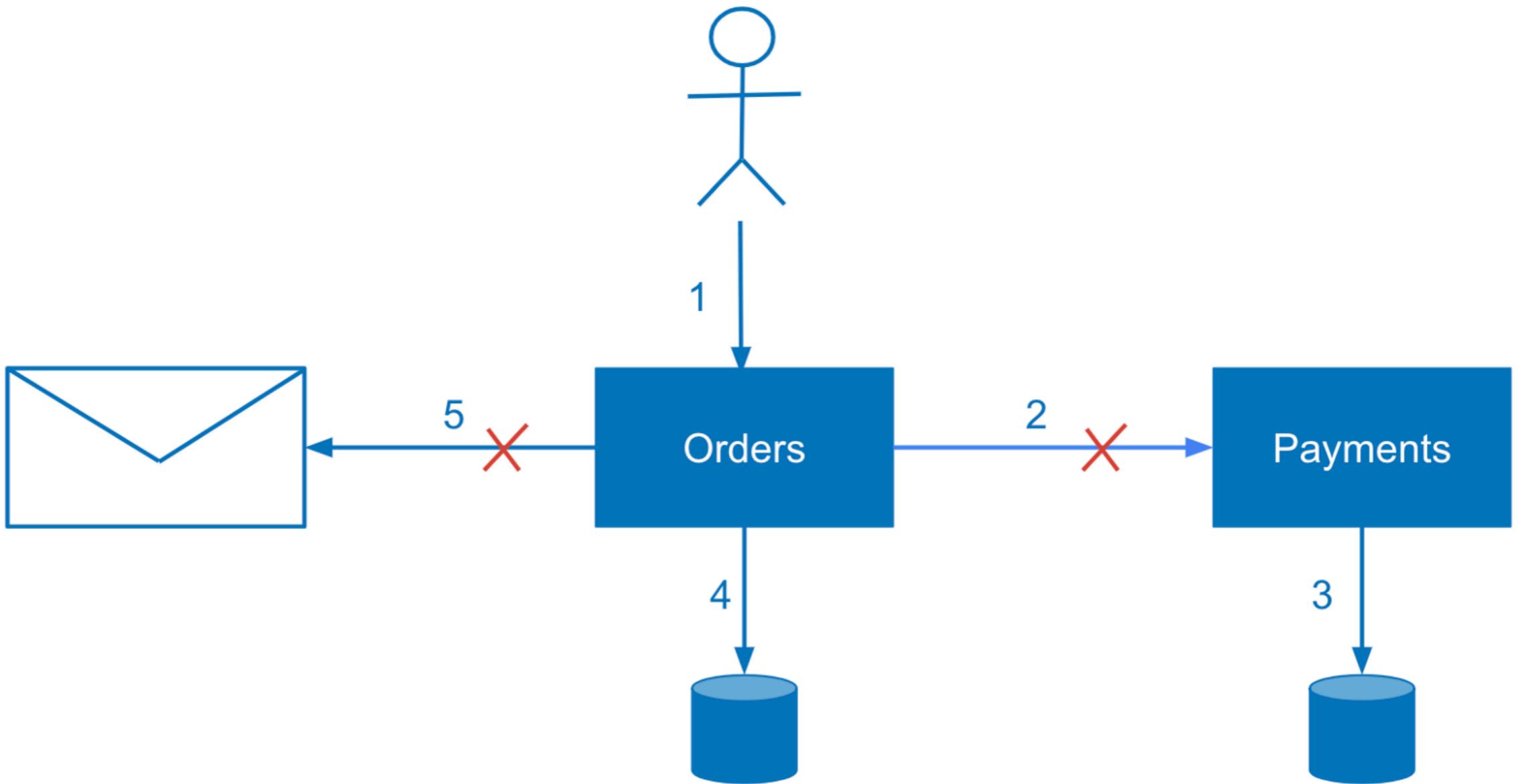
Private database
!=
Private database server



Loose coupling = encapsulation data

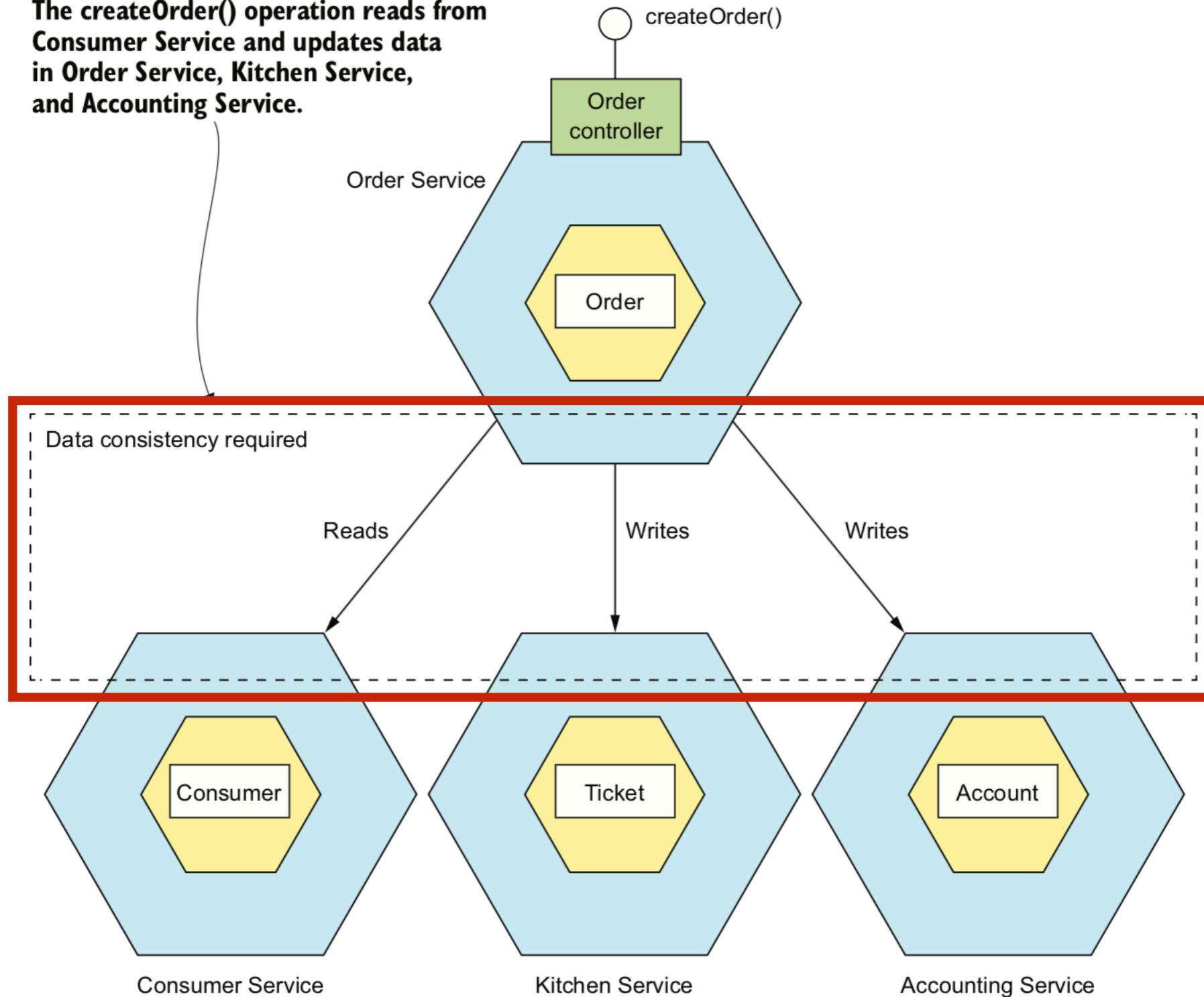


Distributed process failure !!



Problem ?

The **createOrder()** operation reads from Consumer Service and updates data in Order Service, Kitchen Service, and Accounting Service.



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>

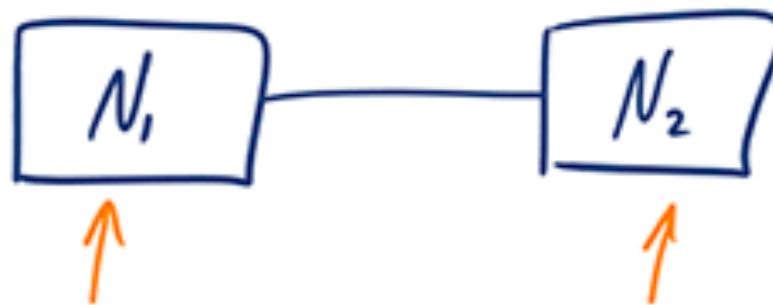


CAP Theorem

Consistency



Availability



Partition Tolerance



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



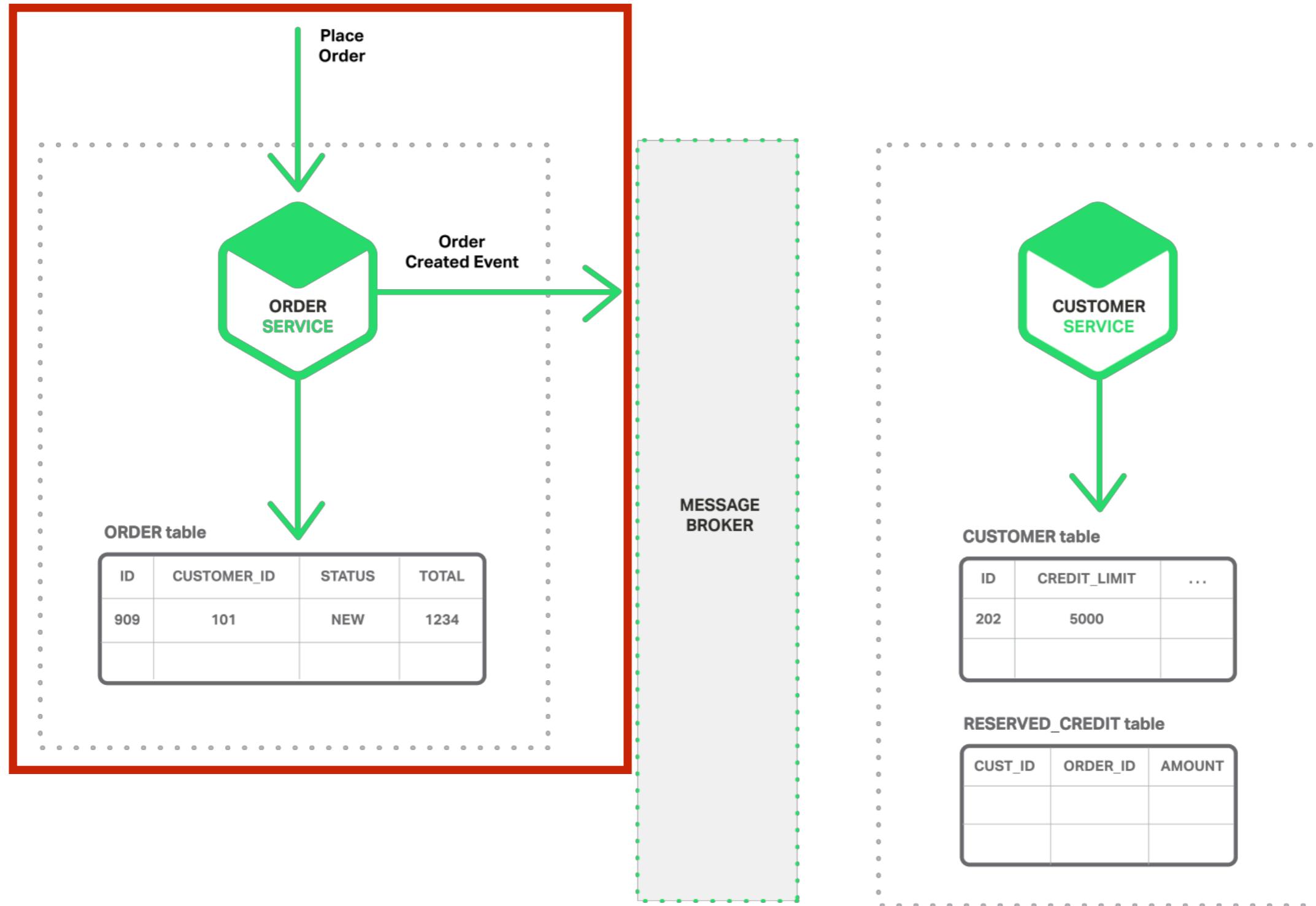
How to maintain data consistency ?



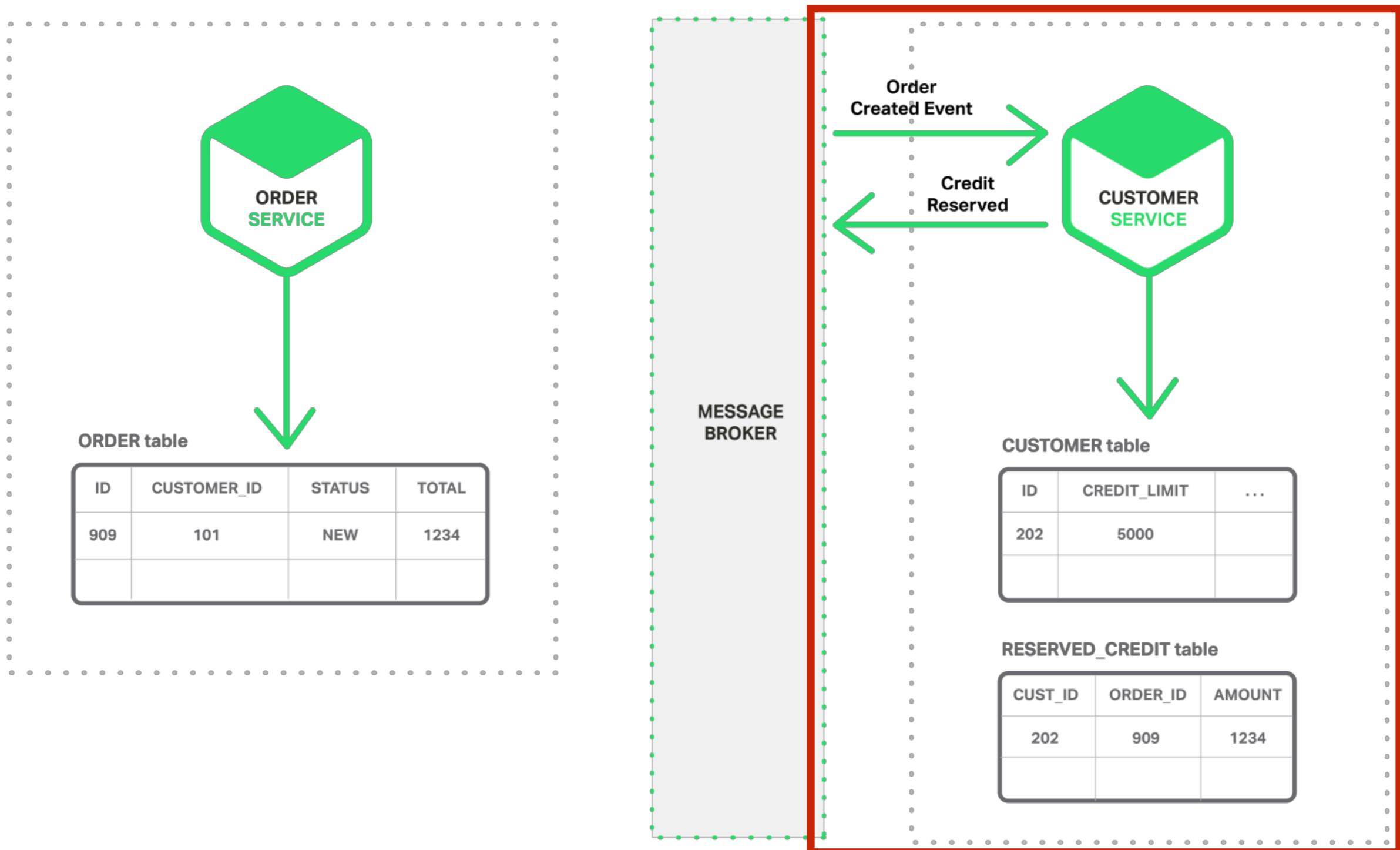
Using Event-driven architecture



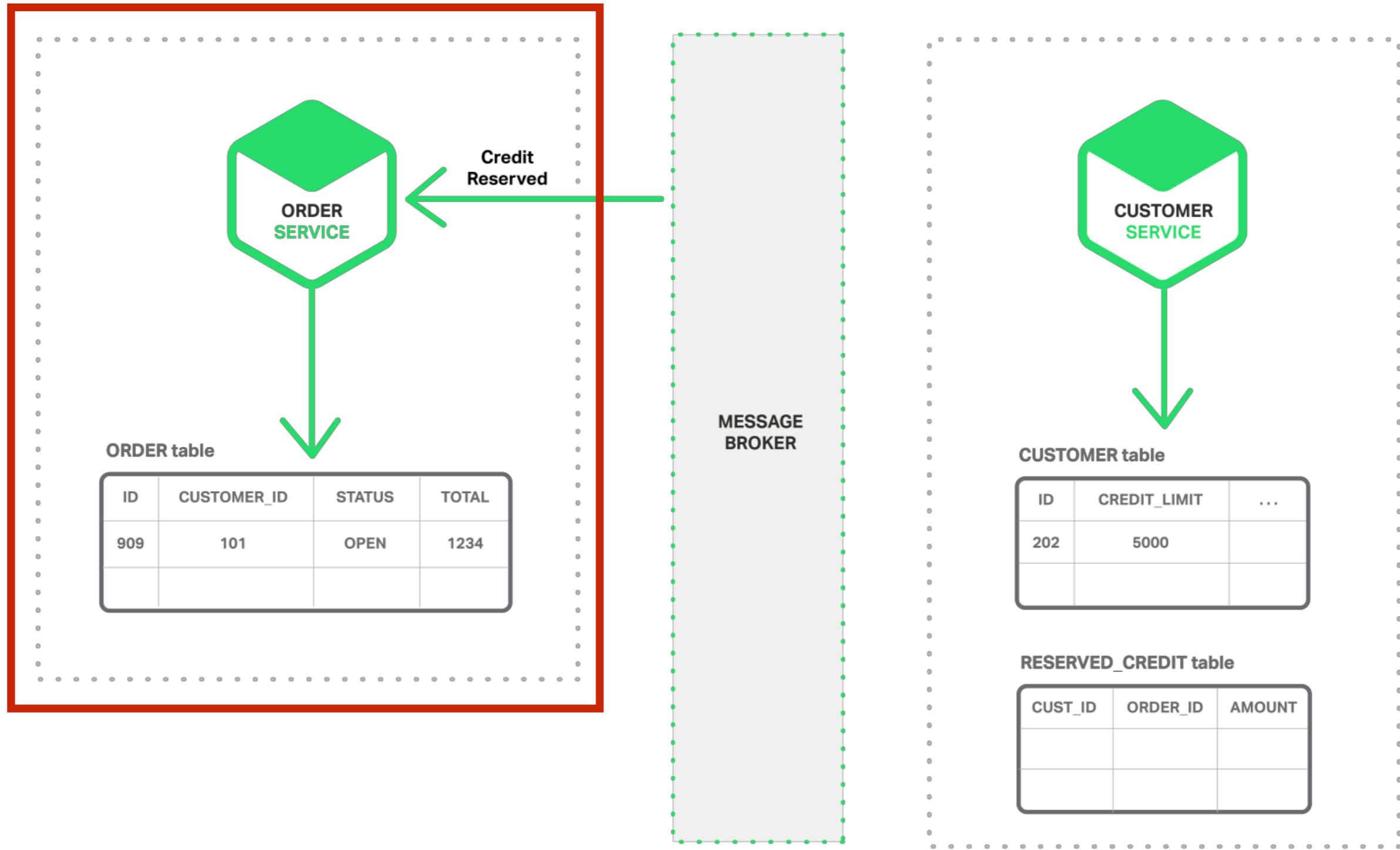
Step 1:: Create new order



Step 2:: Customer service consume



Step 3:: Order service consume



How to implement queries ?



Query data from multiple services

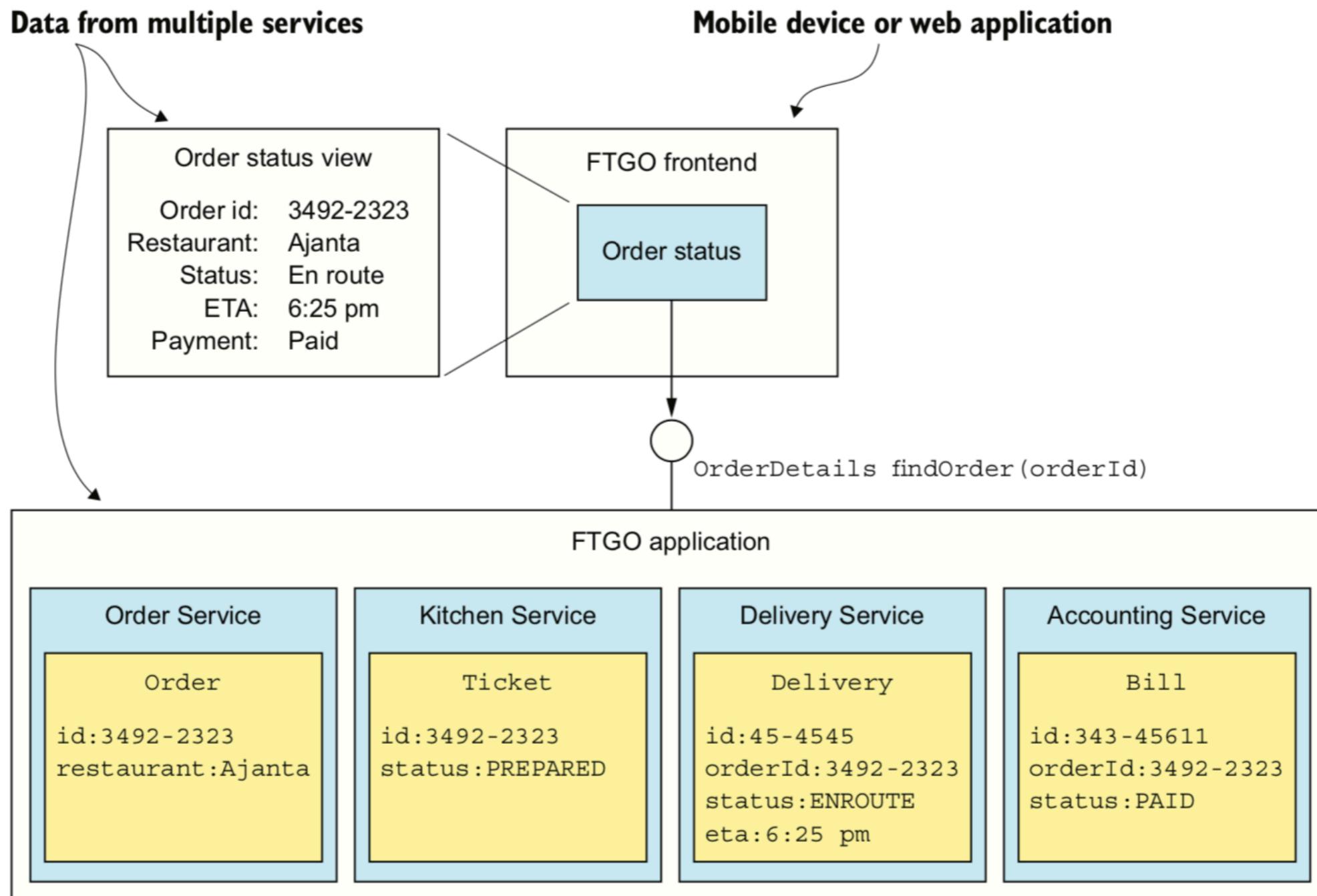
API composition

CQRS (Command Query Responsibility Segregation)

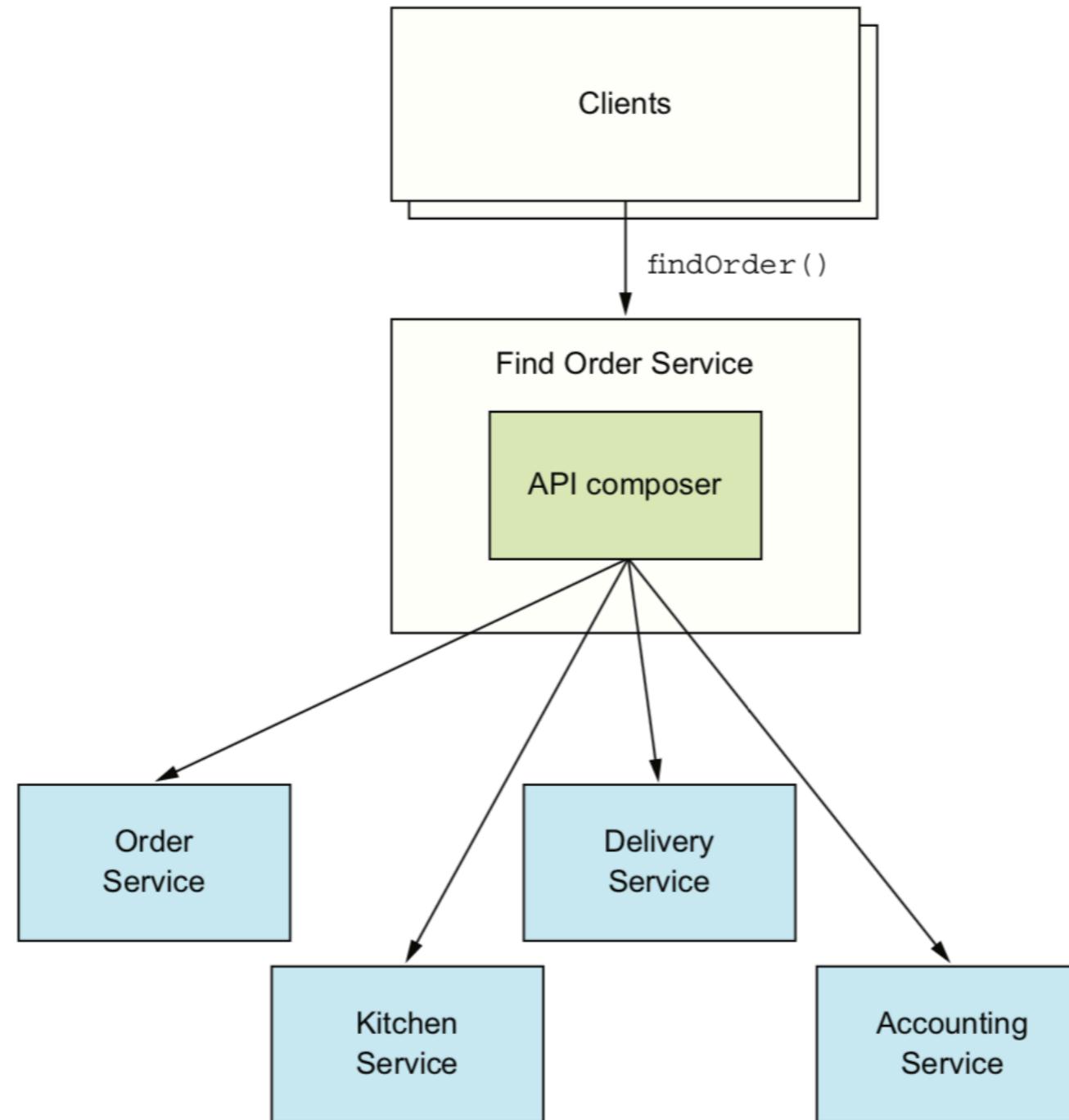
Cold data from services



Problem ?



API composition pattern

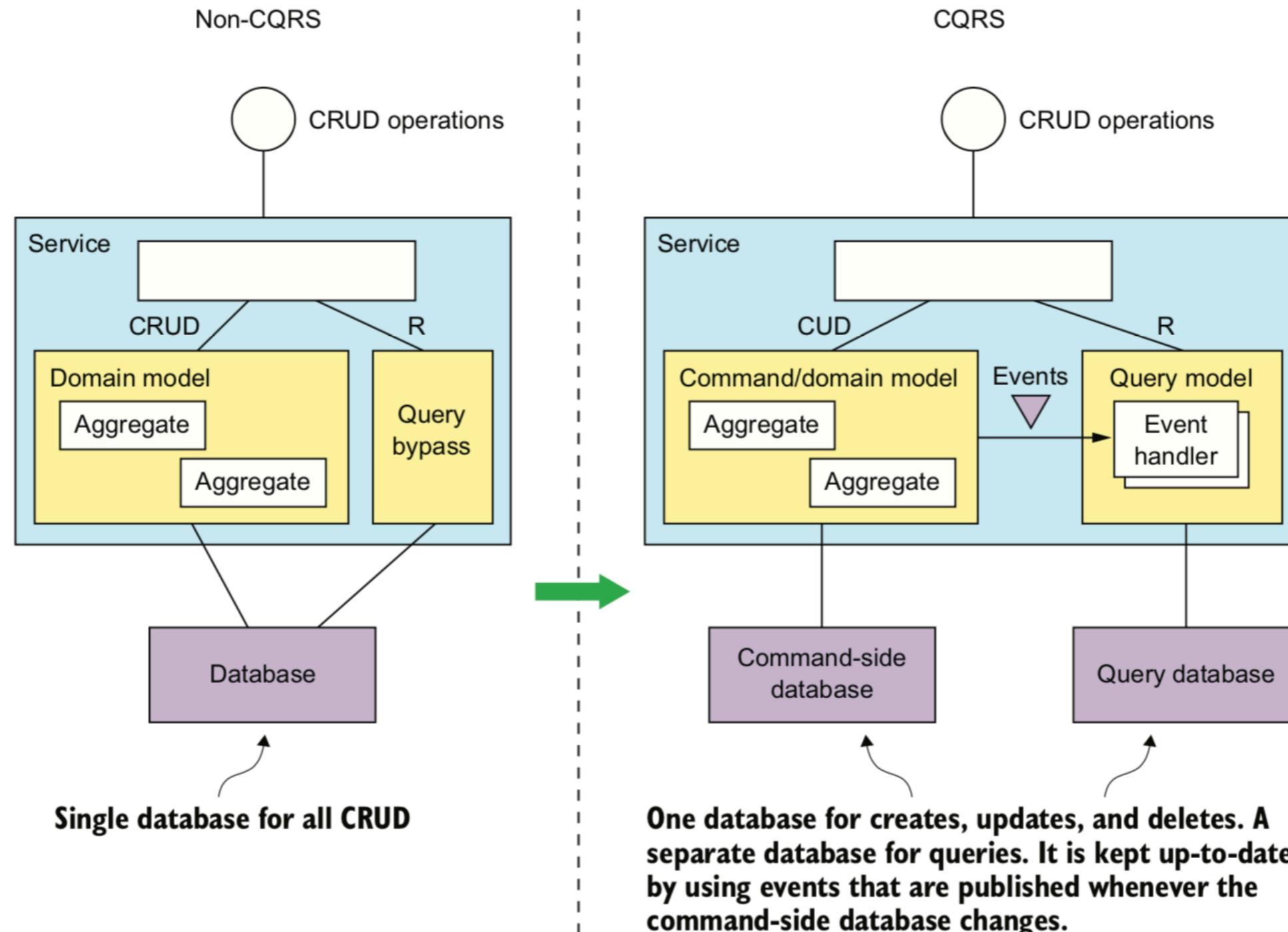


Drawbacks

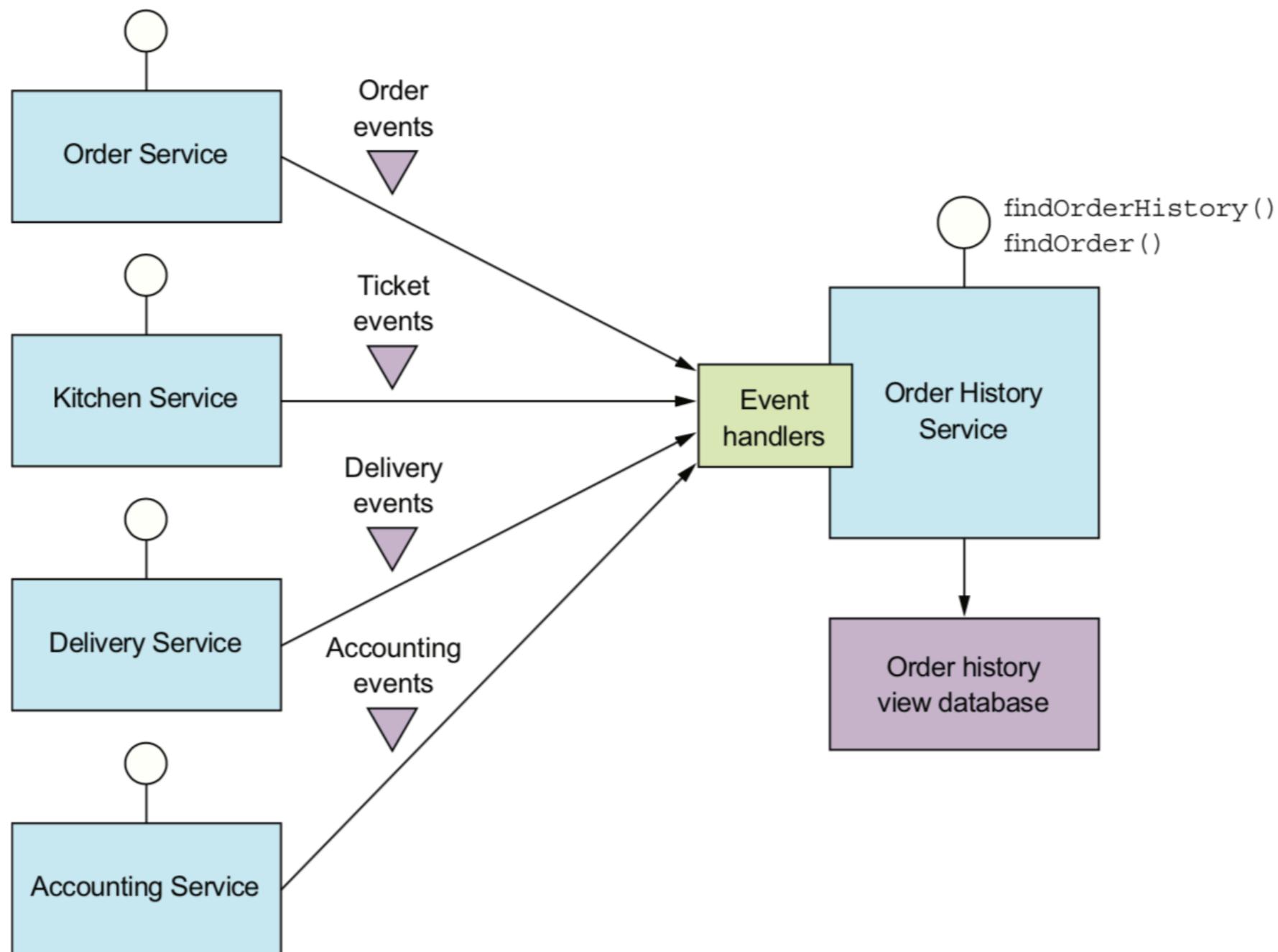
Increase overhead
Lack of transactional data consistency
Reduce availability



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

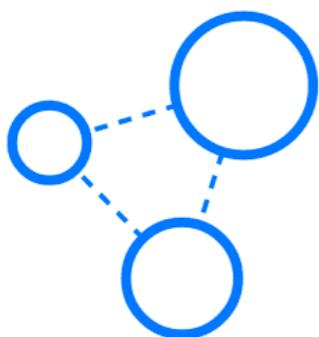


More ...



Beyond Microservice

Process and Organization



Flexible organizational structure

With clear roles and accountabilities



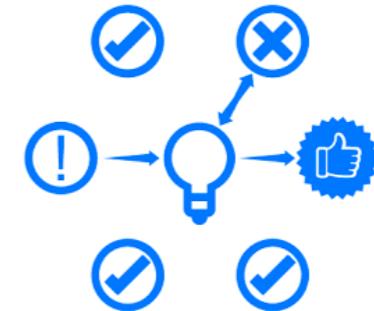
Efficient meeting formats

Geared toward action and eliminating over-analysis



More autonomy to teams and individuals

Individuals solve issues directly without bureaucracy



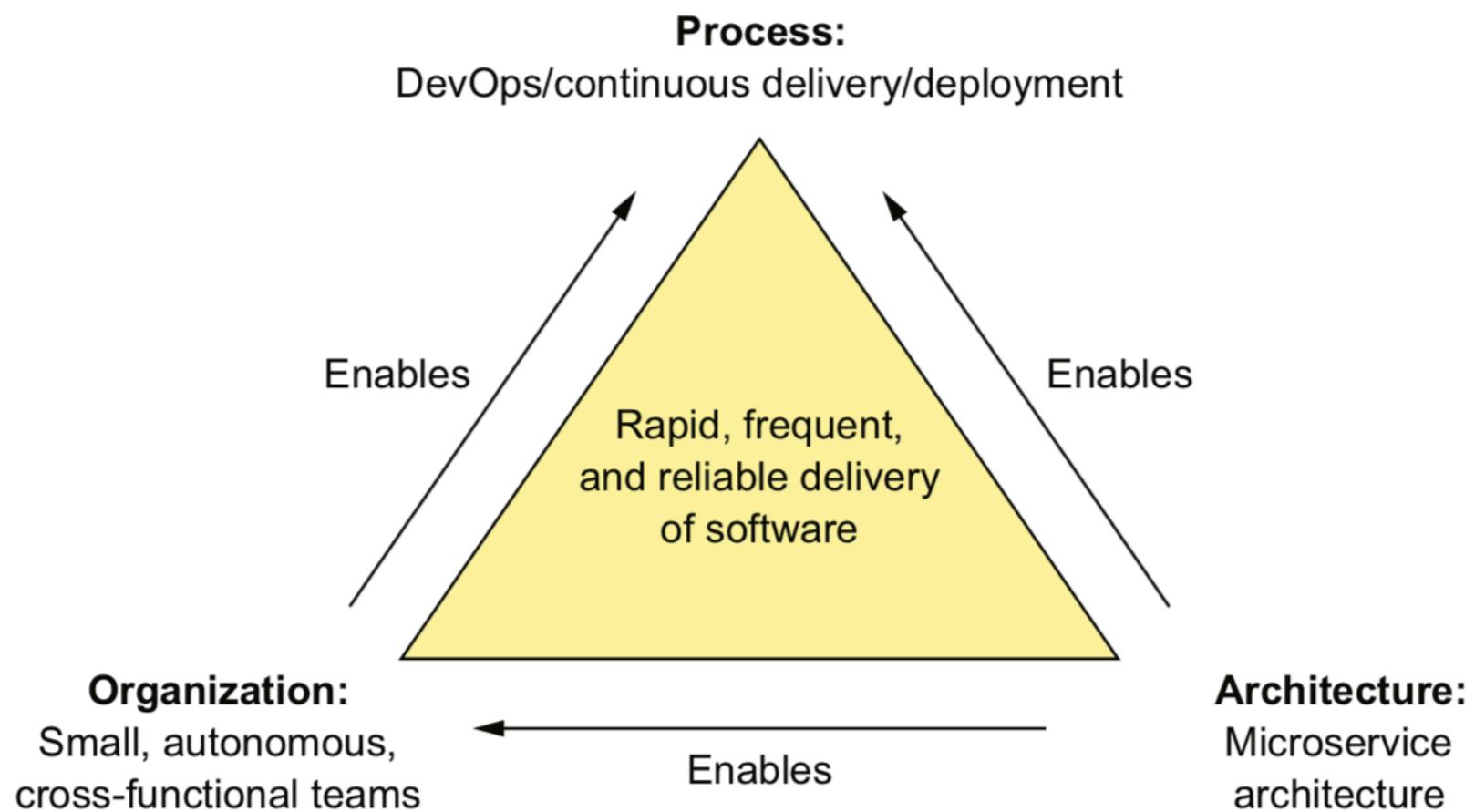
Unique decision-making process

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



"organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations"

- *Melvin Conwey, 1968* -



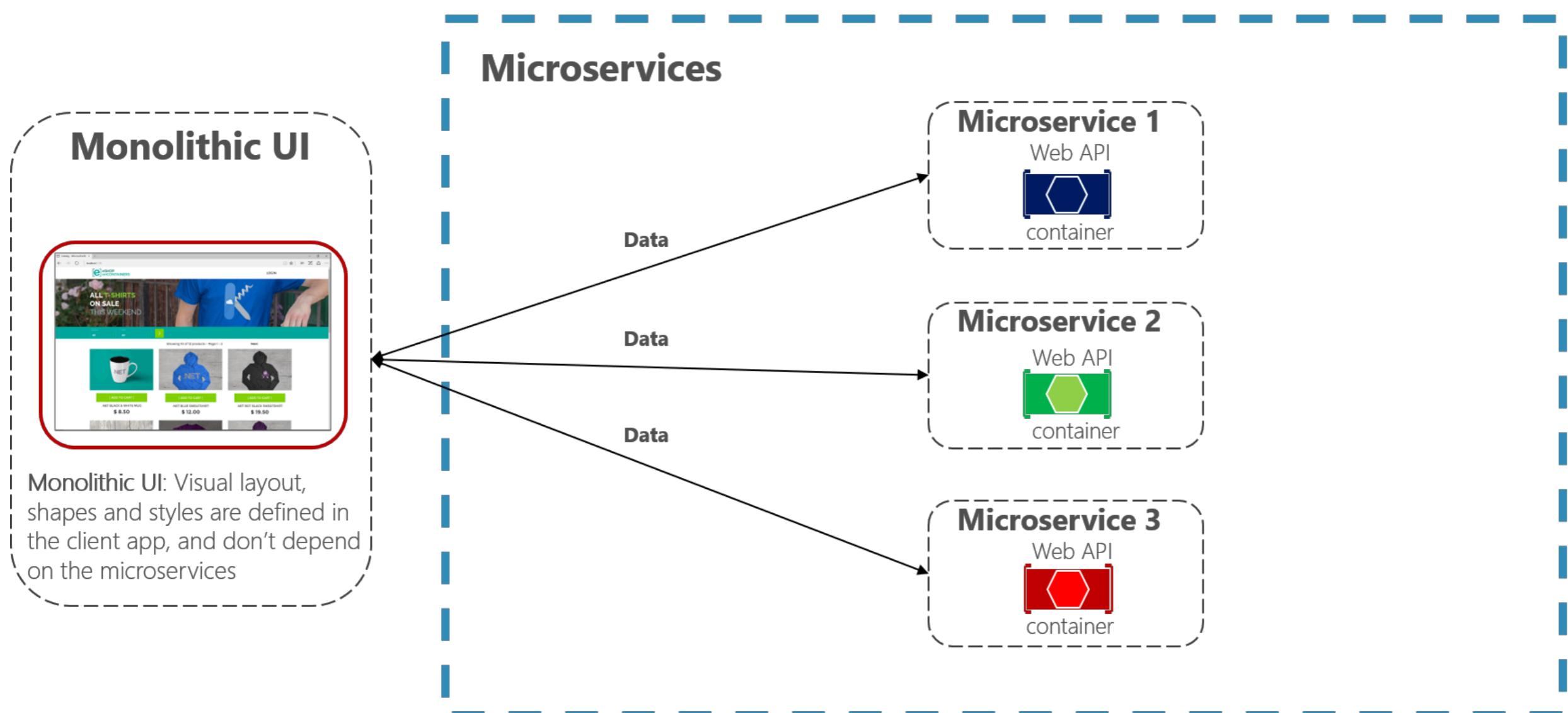
More ...



Integrate services with User Interface



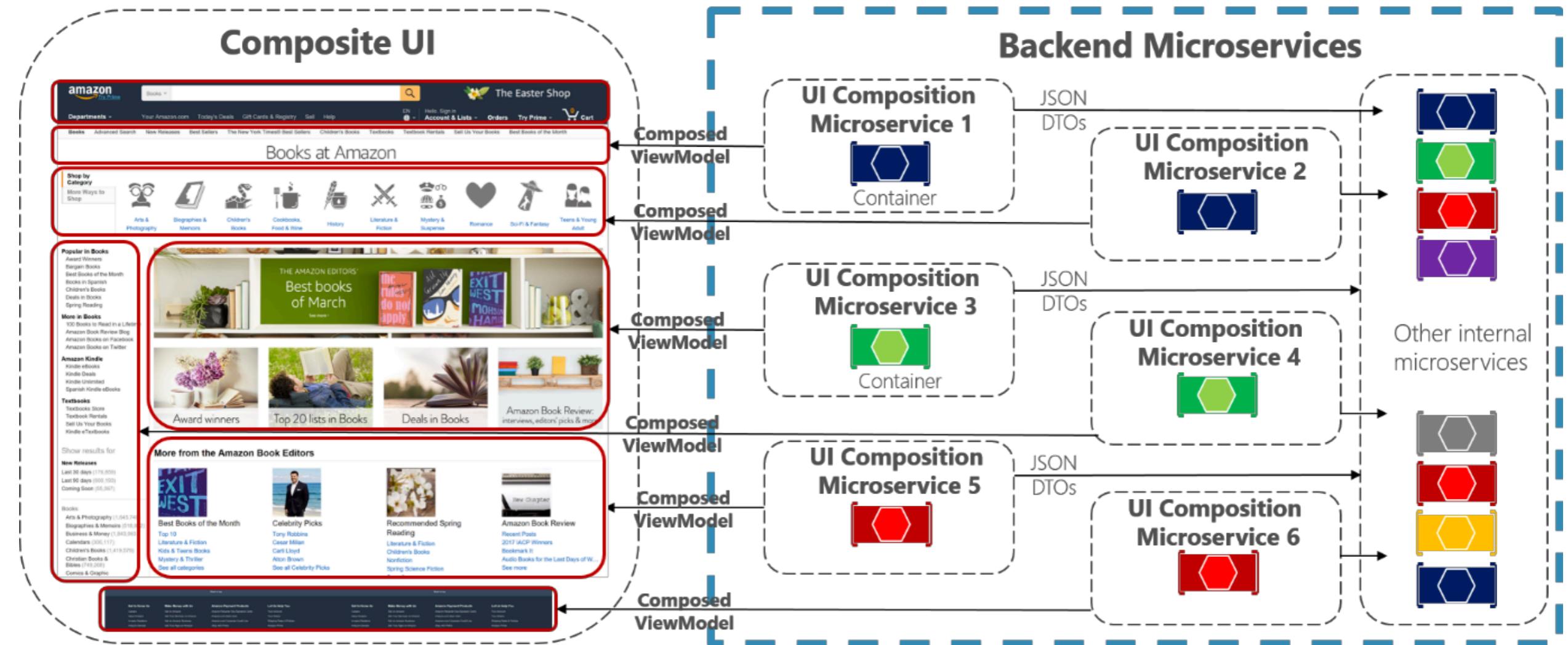
Monolithic UI consuming microservices



<https://docs.microsoft.com>



Composite UI generated by microservices



<https://docs.microsoft.com>

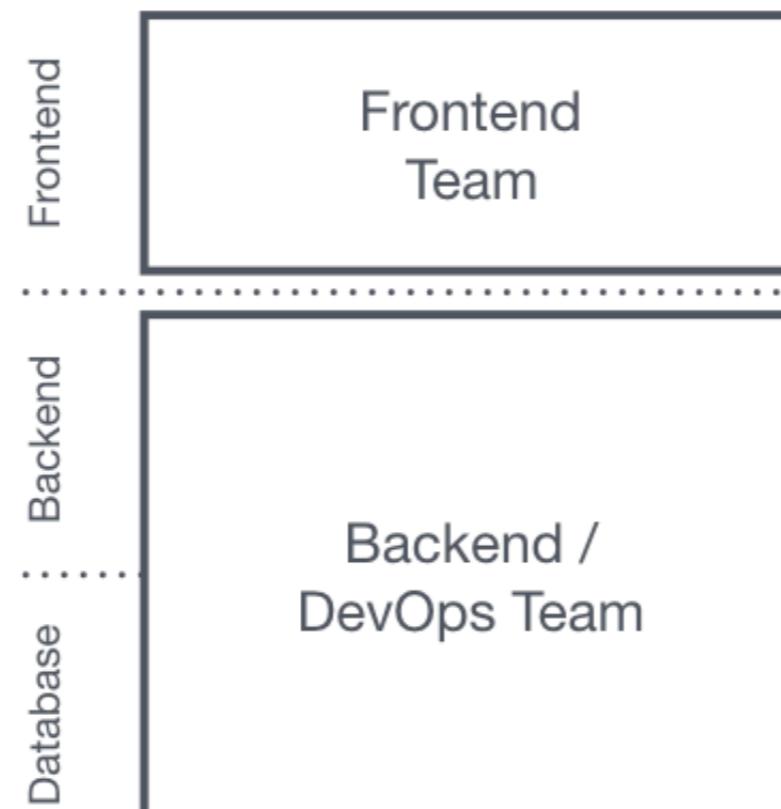


Monolith frontend

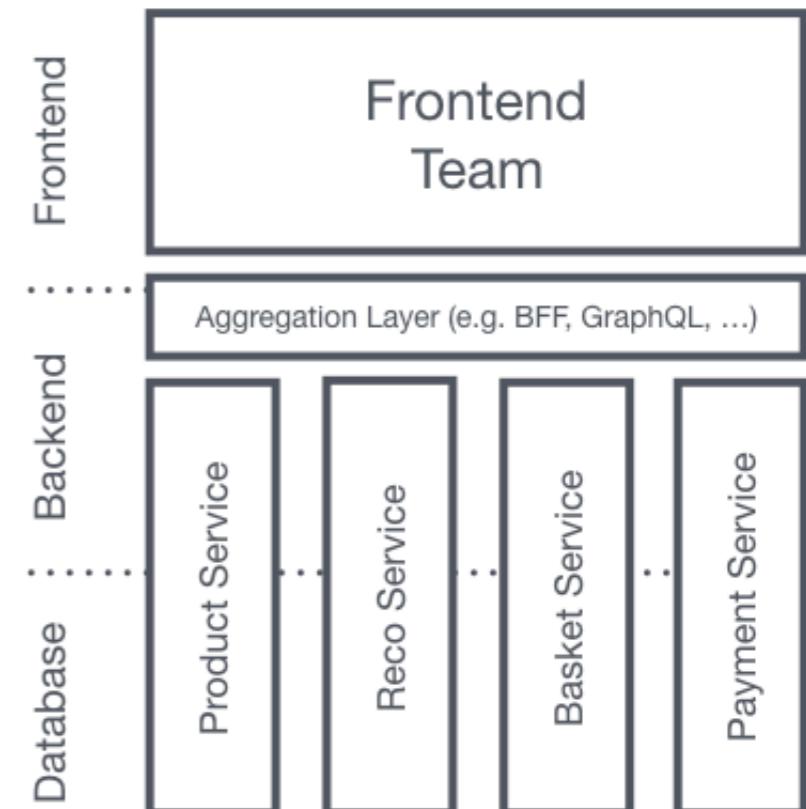
The Monolith



Front & Back



Microservices

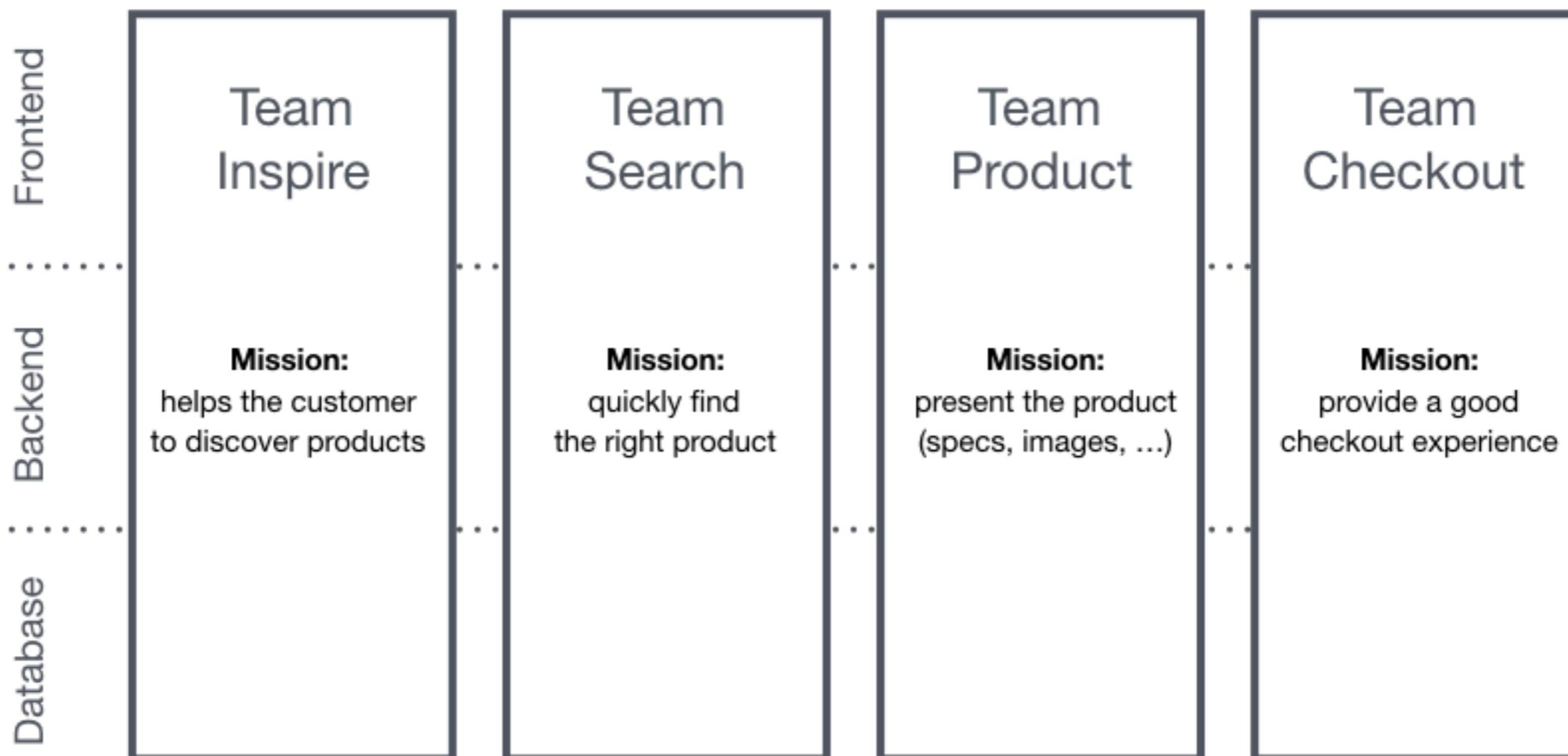


<https://micro-frontends.org/>



Micro frontend

End-to-End Teams with Micro Frontends



<https://micro-frontends.org/>



Summary



Let's start with good monolith



Module 1

Module 2

Module 3

Module 4

Module 5

Module 6



Find your problem



Module 1

Module 2

Module 3

Module 4

Module 5

Module 6



Module 1

Module 2

Module 3

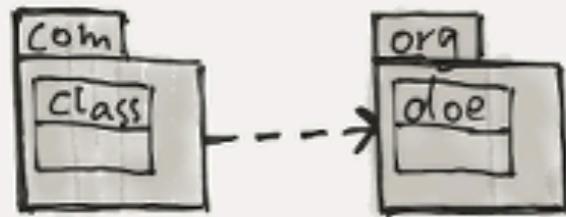
Module 5

Module 6

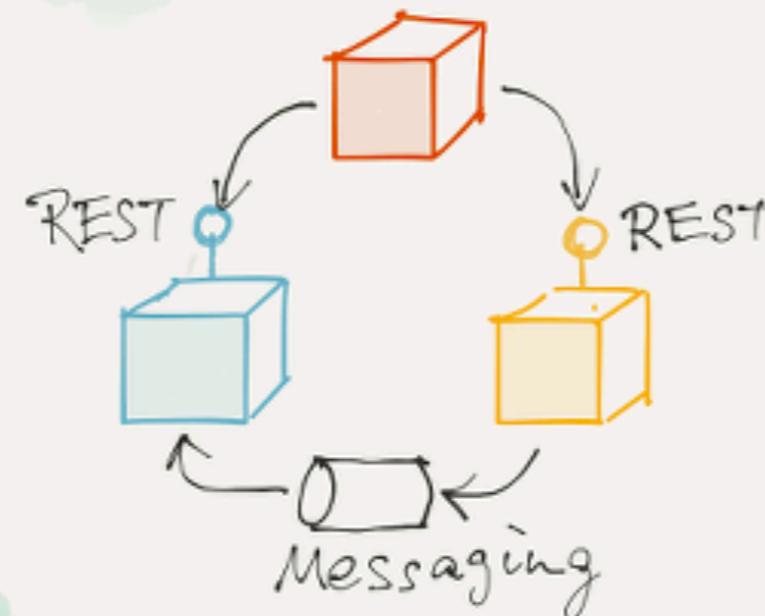
Module 4



Architecture



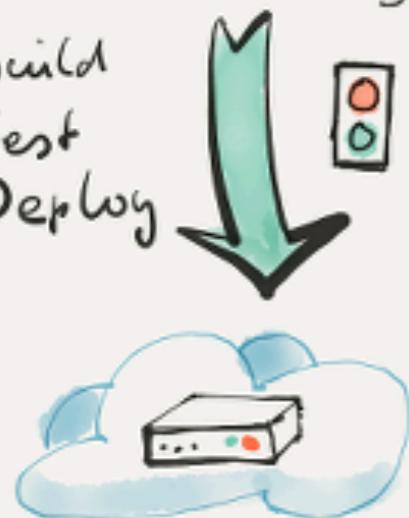
Microservices



Deployment

Continuous Delivery

`{ var i=1; }`
Build
Test
Deploy



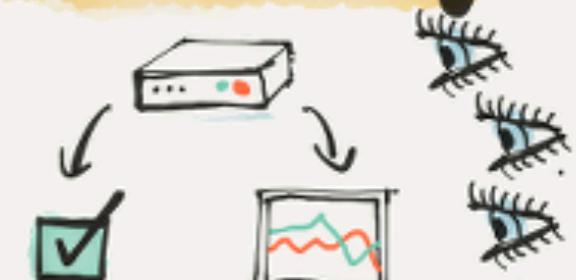
Infrastructure



People & Teams



Monitoring



Features & Technology



What's Next ?



Develop services ?



Testing services ?



Deployment services ?



Observability services ?



Security of services ?

