



- » Skan.ai - chief Architect
- » Ai.robotics - chief Architect
- » Genpact - solution Architect
- » Welldoc - chief Architect
- » Microsoft
- » Mercedes
- » Siemens
- » Honeywell



Mubarak



- » How to build Microservice
- » Patterns
- » Anti Patterns
- » Hands on Demos
- » Case Studies

what do
YOU?
expect?

- > Your Technology stack
- > Total Years of experience
- > Docker / Kubernetes ?

Day 1

System

↓
Apply Patterns

App 1

↓
Apply Patterns

App 2

↓
Apply Patterns

M1

App

M2

App

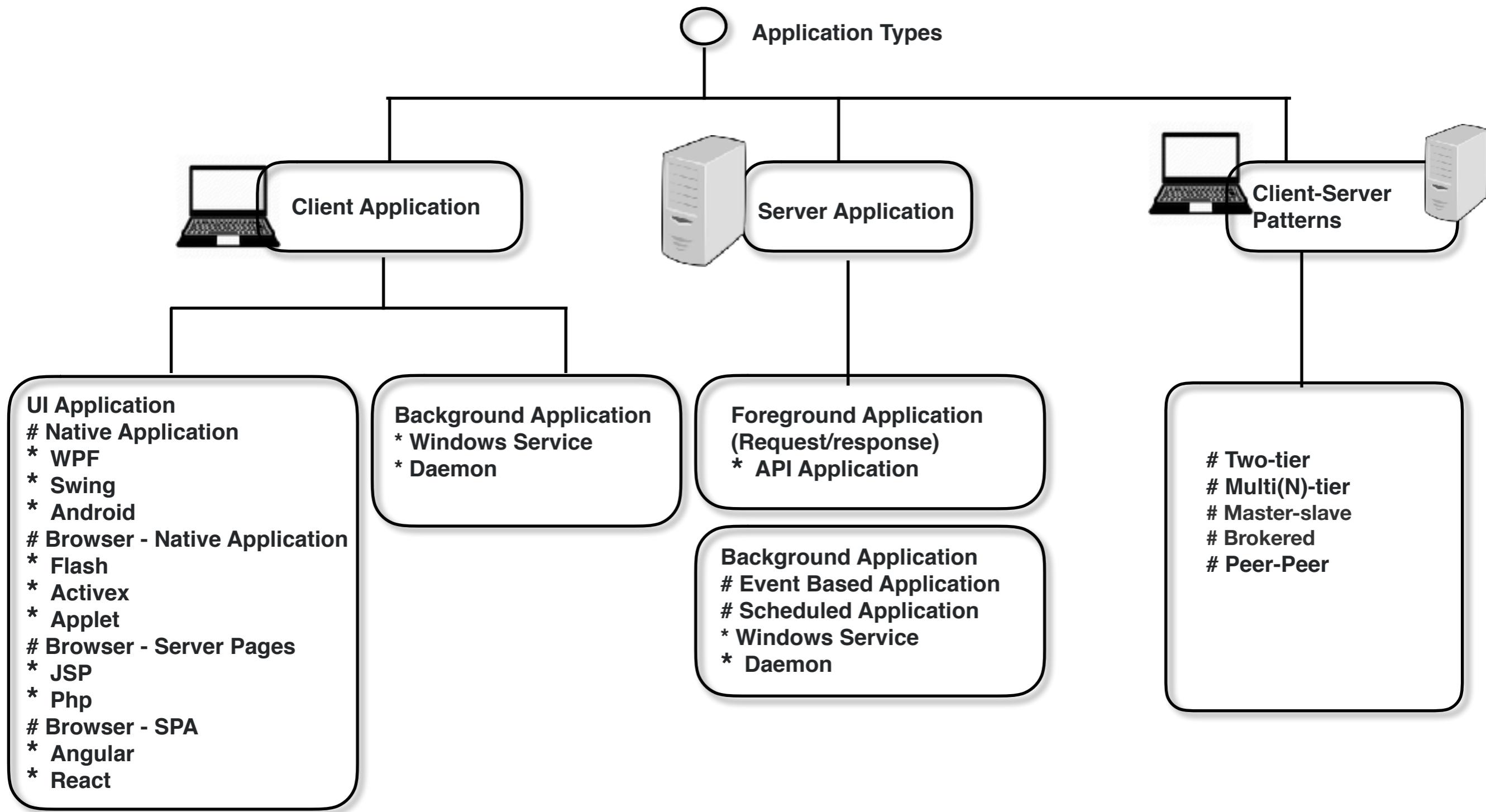
M3

M3

M4

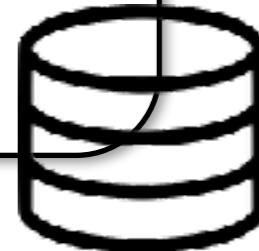
M4

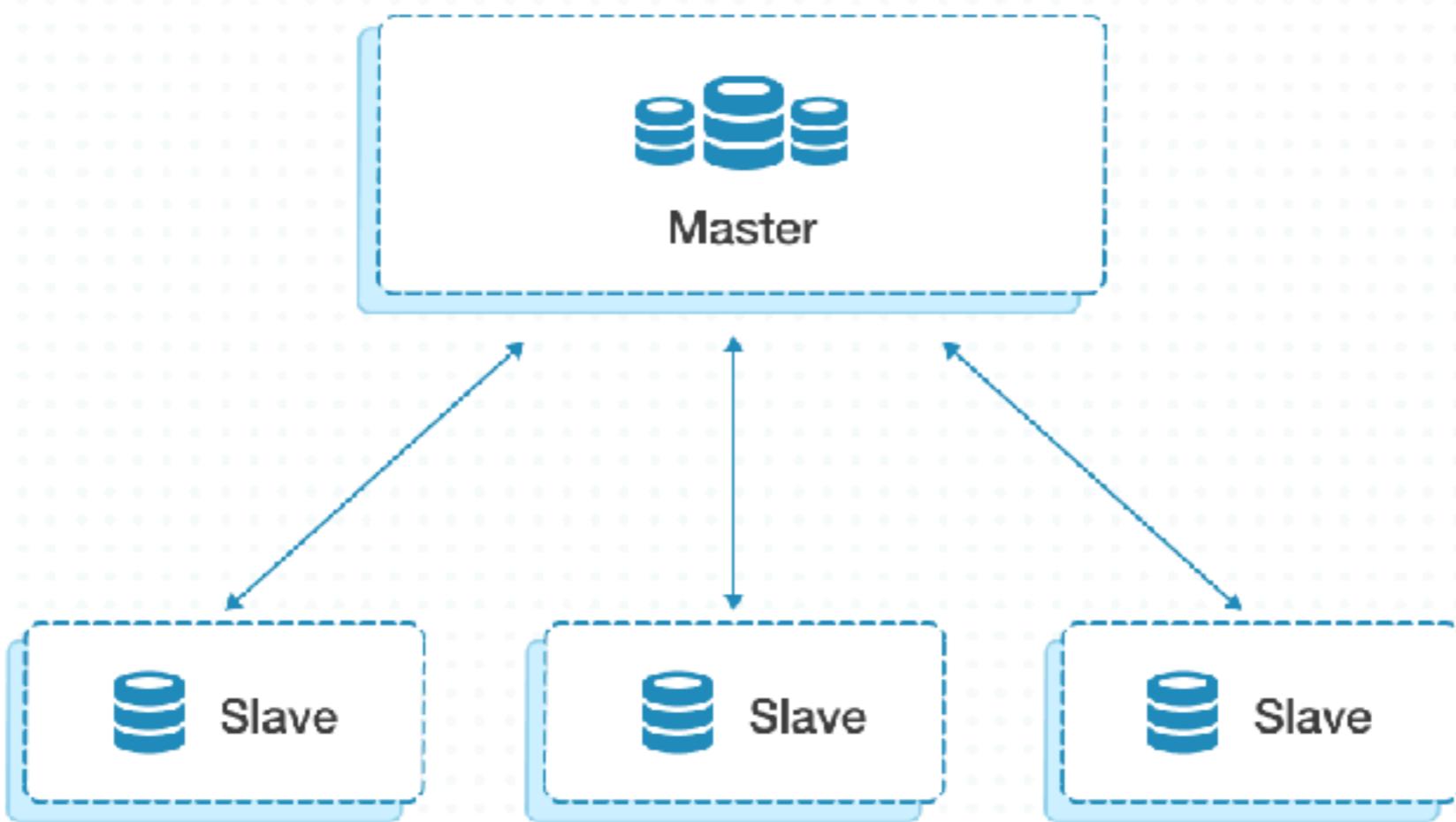
Decompose the system into Applications/Tiers



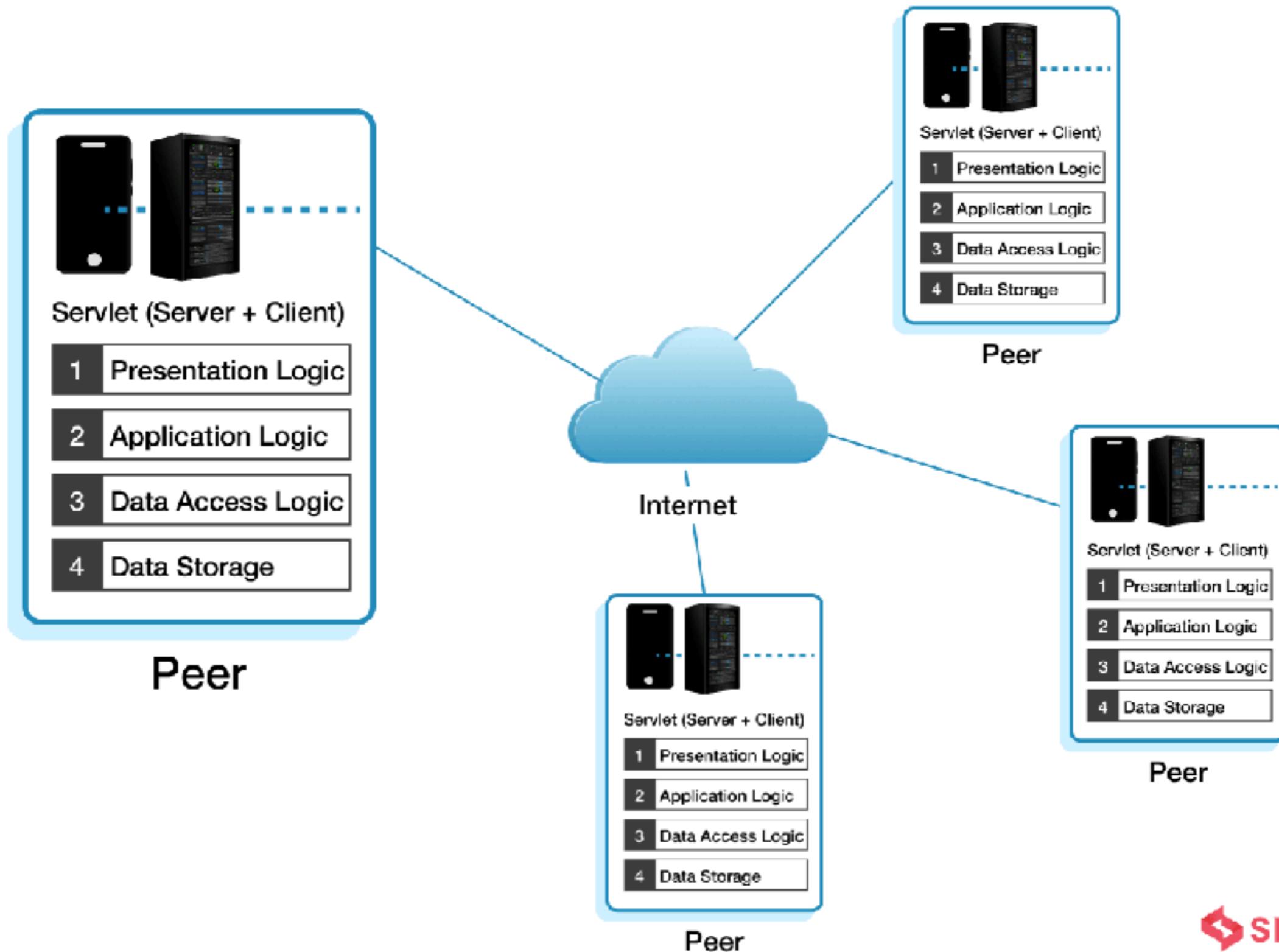
ToDo Portal Application

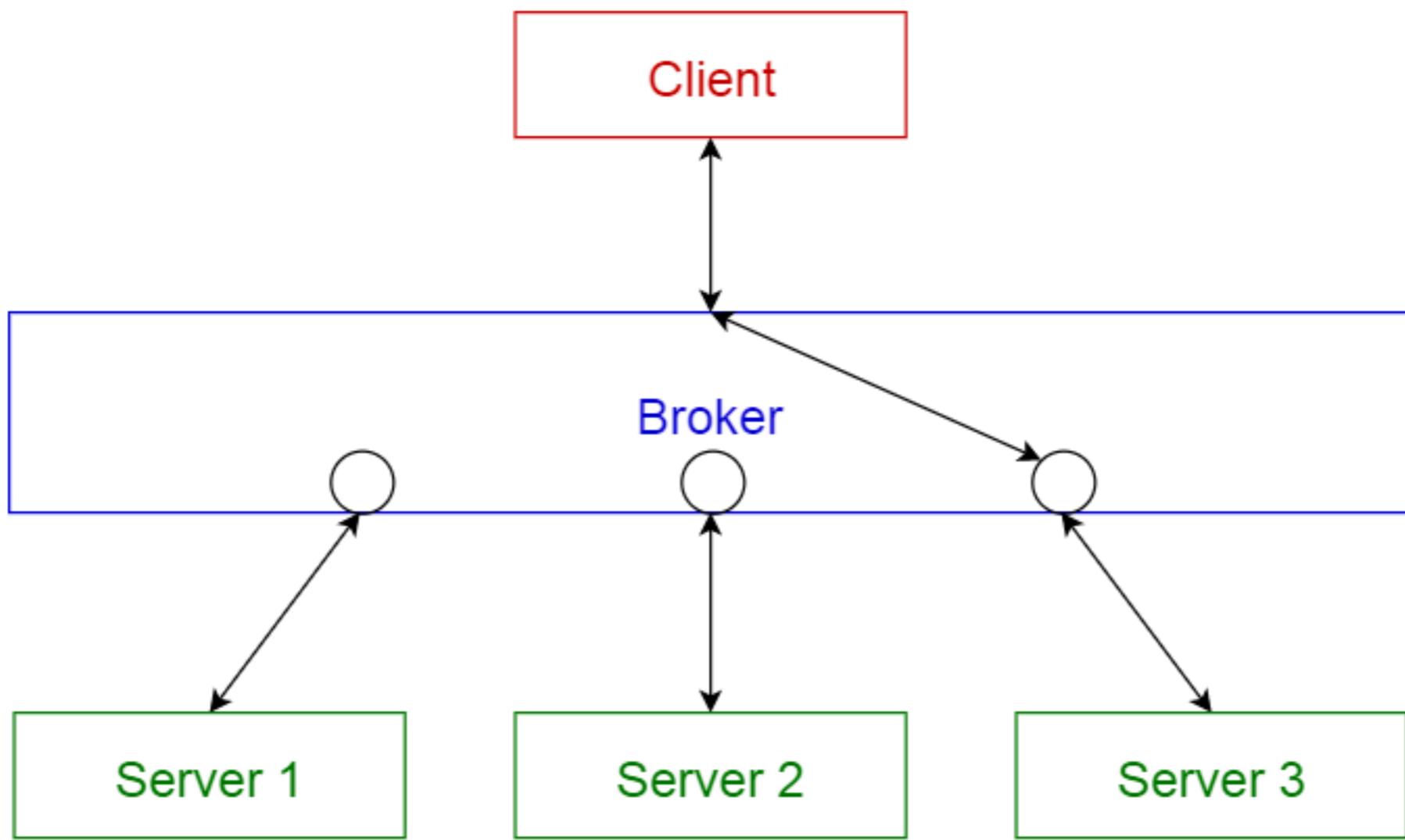
ToDo API Application



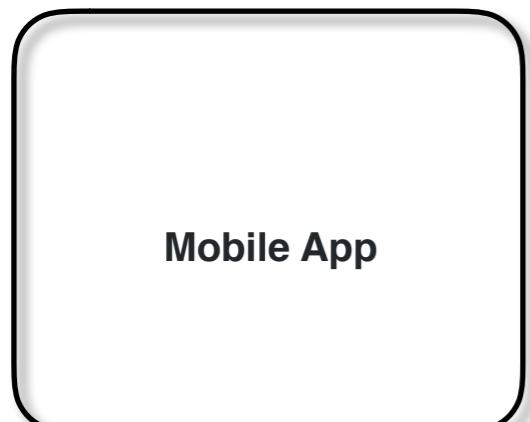
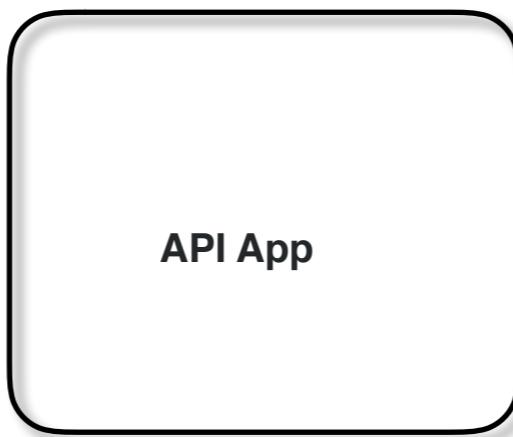
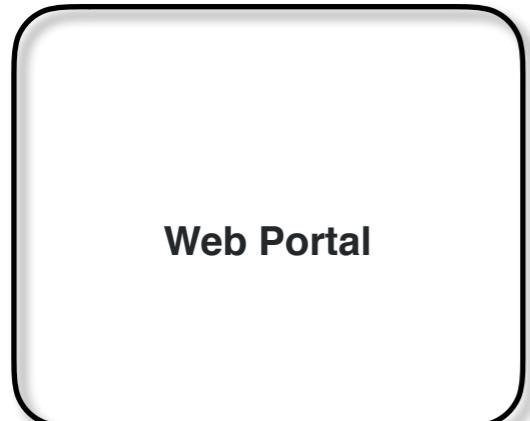


Peer-to-Peer architecture

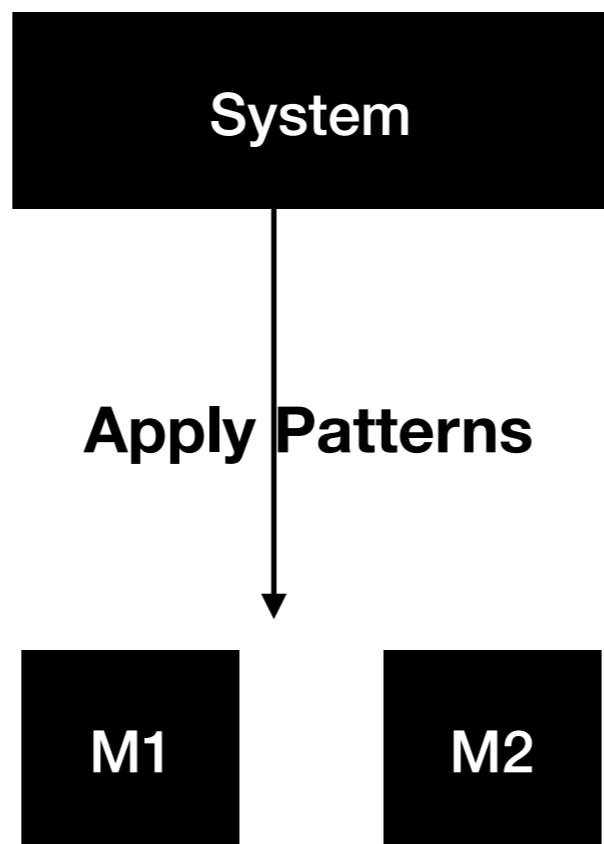


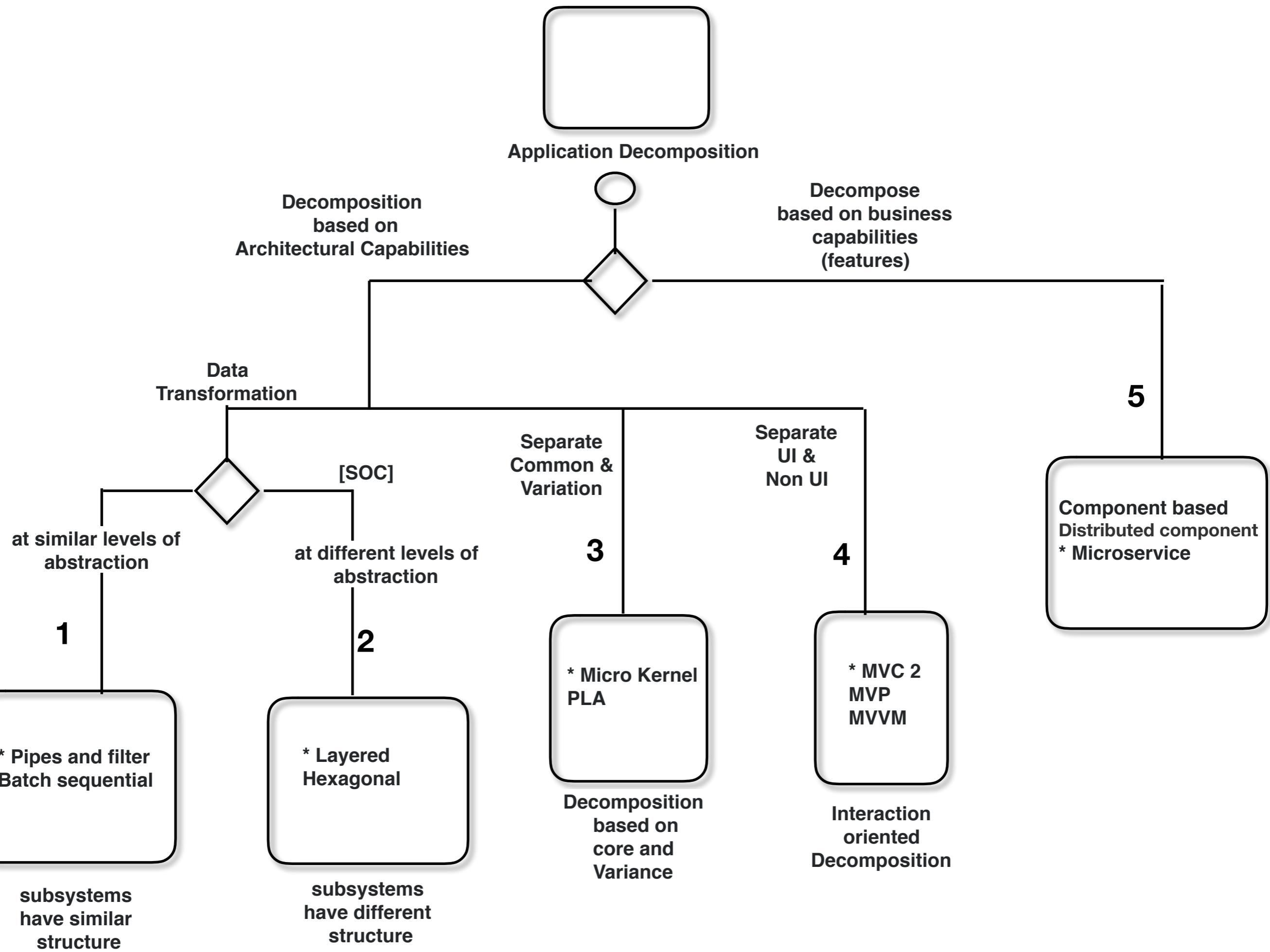


RedBus



System Decomposition

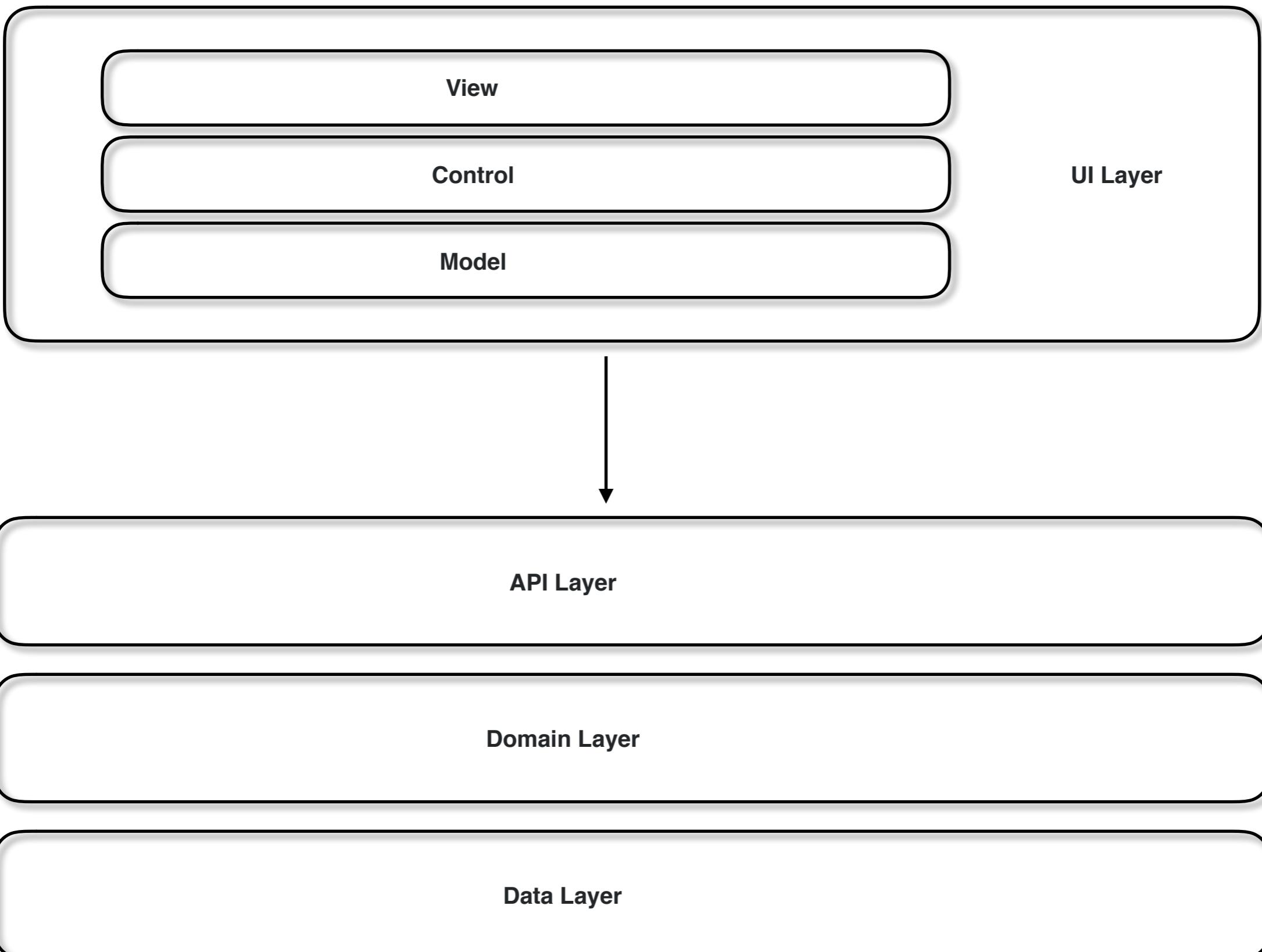


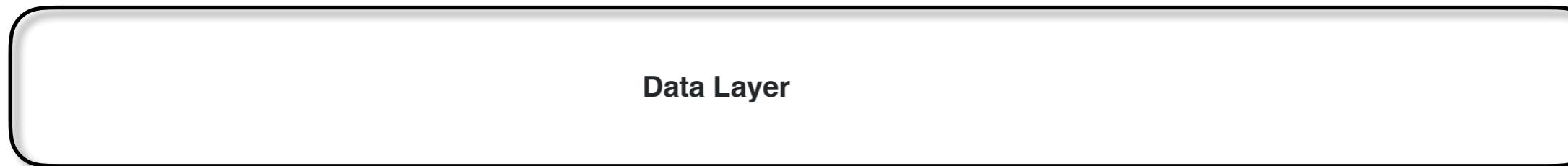
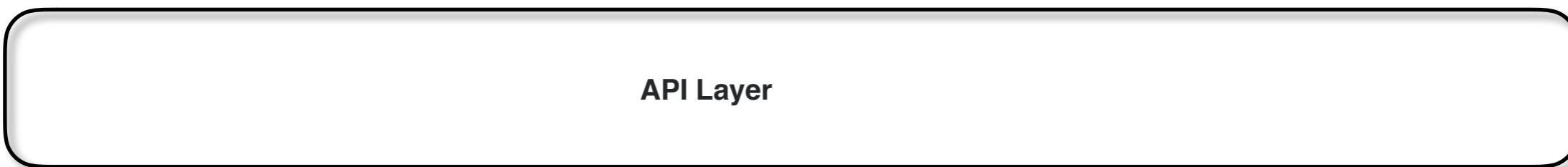
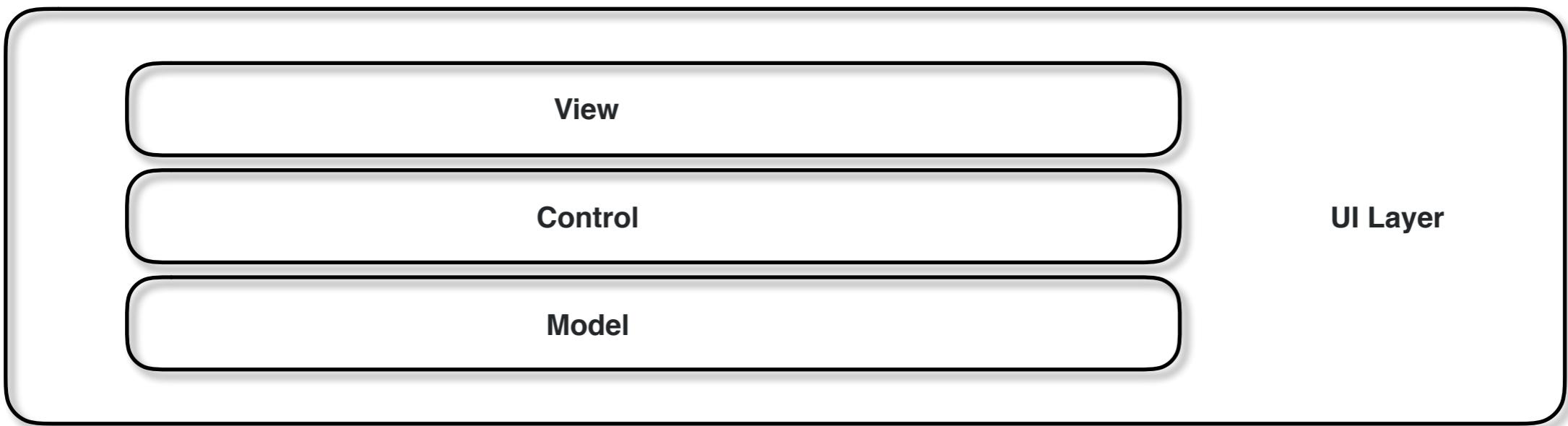


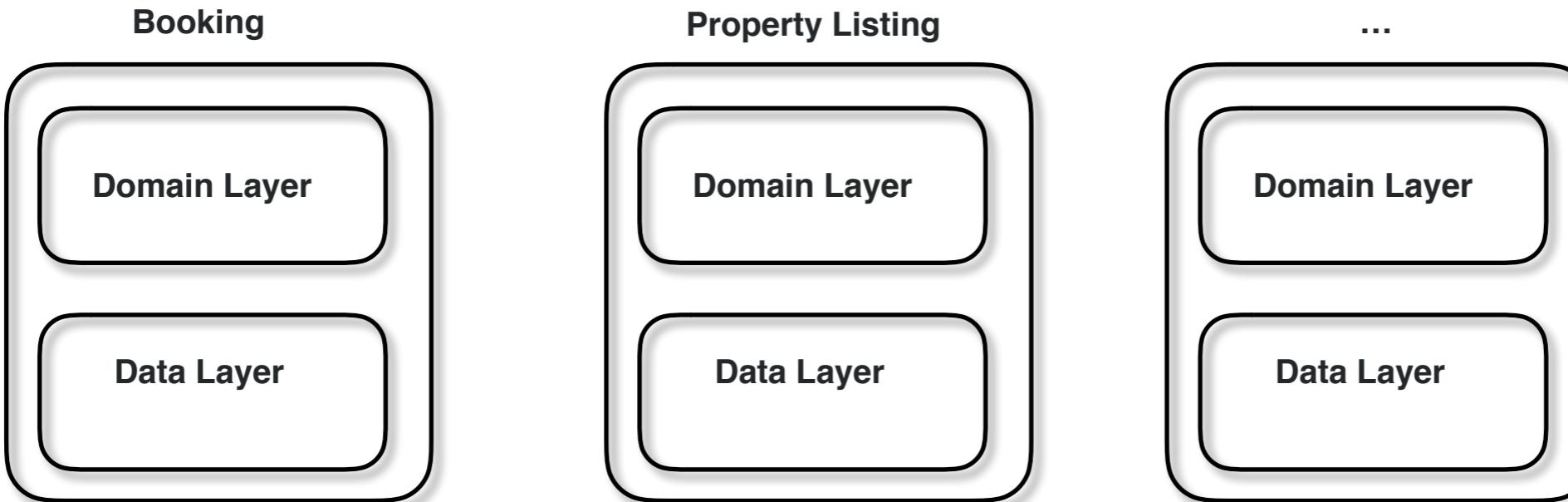
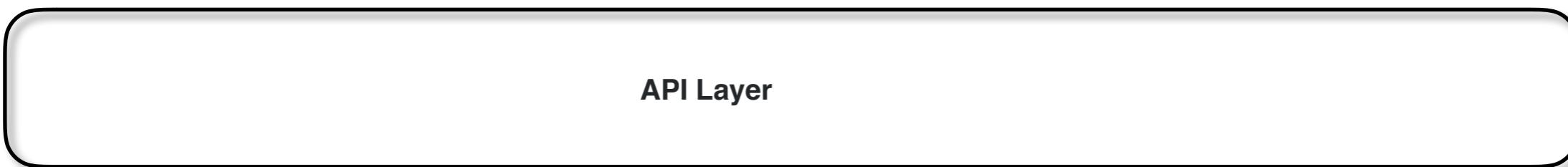
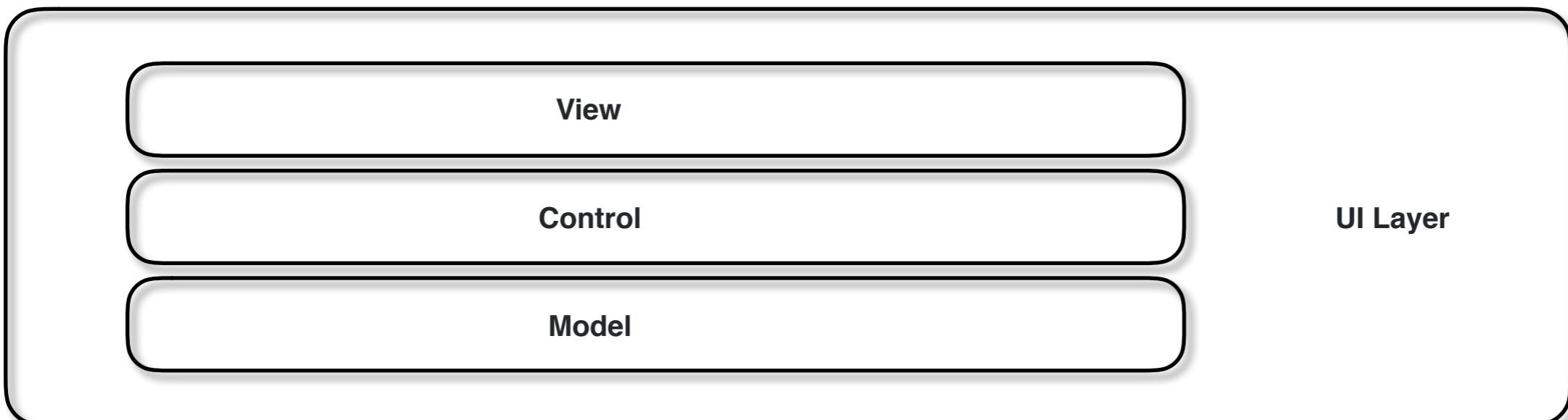
View

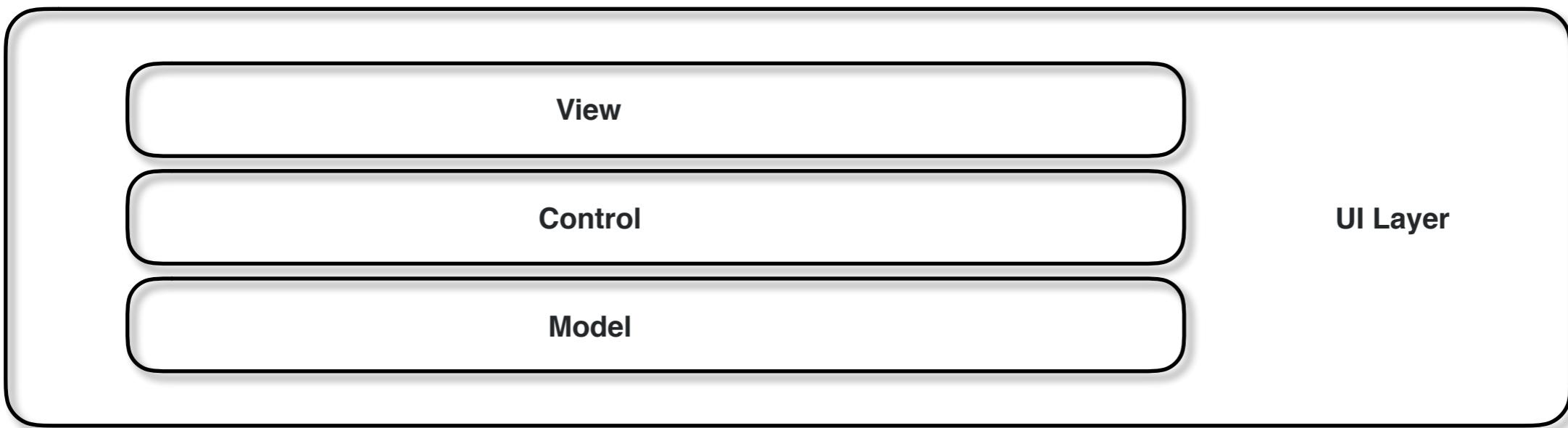
Control

Model

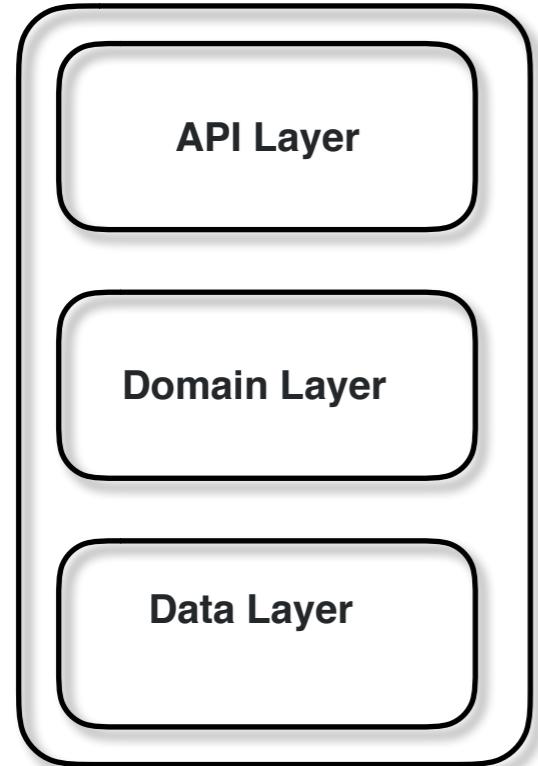




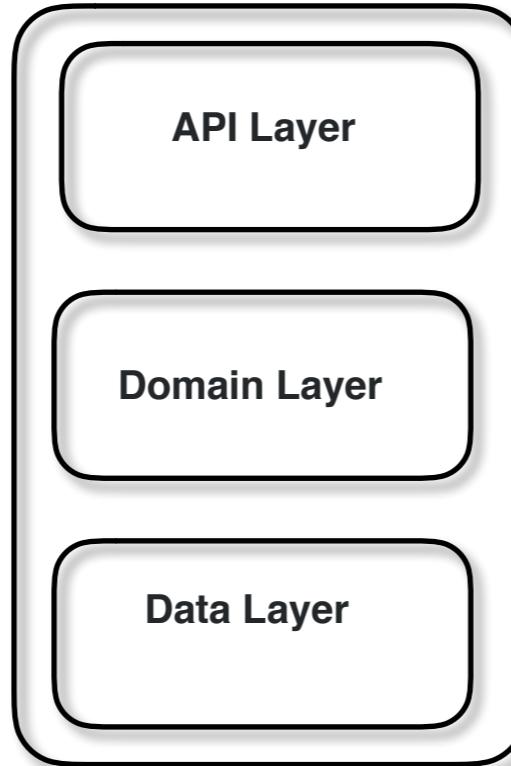




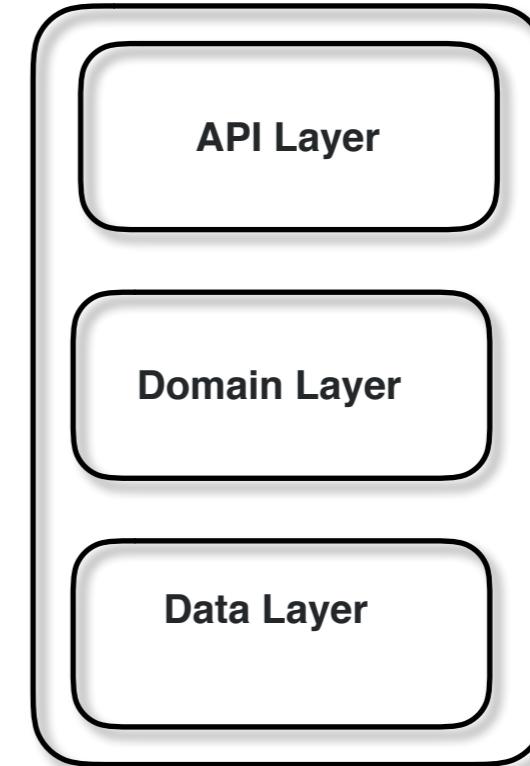
Booking

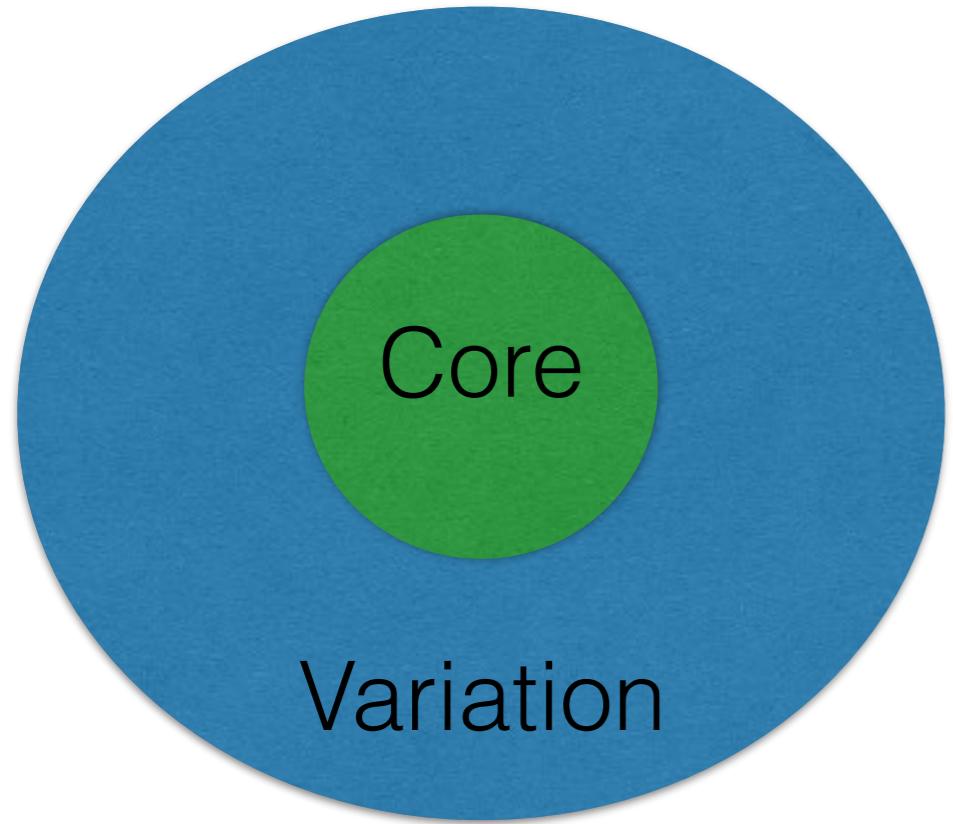


Property Listing



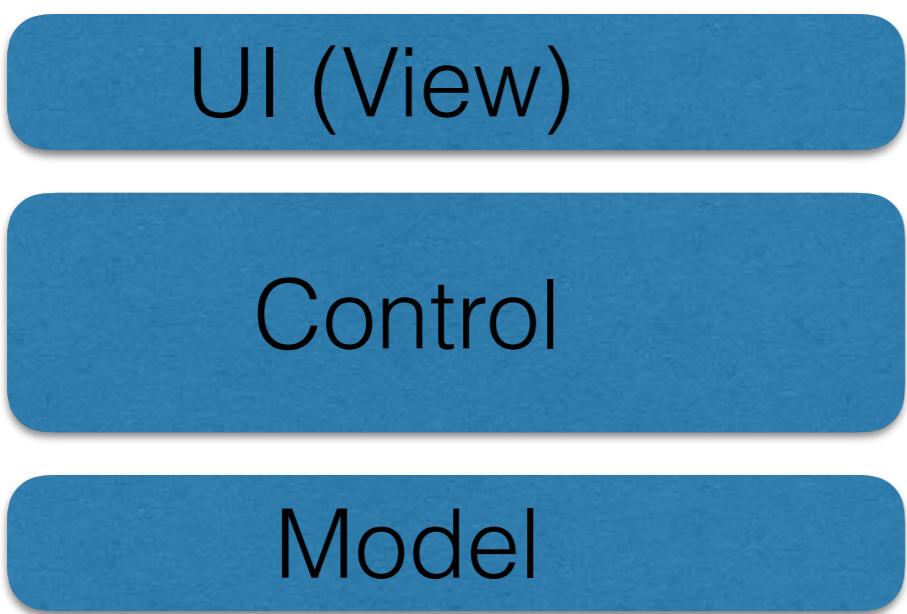
...





Data

Data



Inventory

Accounts

1

Data
fun(DataFormat1)

fun(DataFormat2)

fun(DataFormat3)

Data

2

Data
fun(DataFormat)

fun(DataFormat)

fun(DataFormat)

Data

Abstraction

Parrot
Bird

fun(Parrot p)
fun(Bird b) <--

```
Interface Bird extend LivingThing{
```

```
    MakeNoise
```

LSP (Liskov)

```
}
```

```
Interface FlyingBird extends Bird{
```

```
    Fly <—
```

```
        fun(Bird bird){  
            bird.fly();  
        }
```

Break up the User Interface layout and Logic into separate components. This way, it's much easier to manage and make changes to either side without them interfering with each other.

claims processing system contains the basic business logic required by the insurance company to process a claim, except without any custom processing. most insurance claims applications leverage large and complex rules engines to handle much of this complexity.

However, these rules engines can grow into a complex big ball of mud where changing one rule impacts other rules, or making a simple rule change requires an army of analysts, developers, and testers. State-specific rules and processing is separate from the claims system and can be added, removed, and changed with little or no effect on the rest of the system or other modules.

Build two health care applications

1. Diabetes Management
2. Asthma Management.

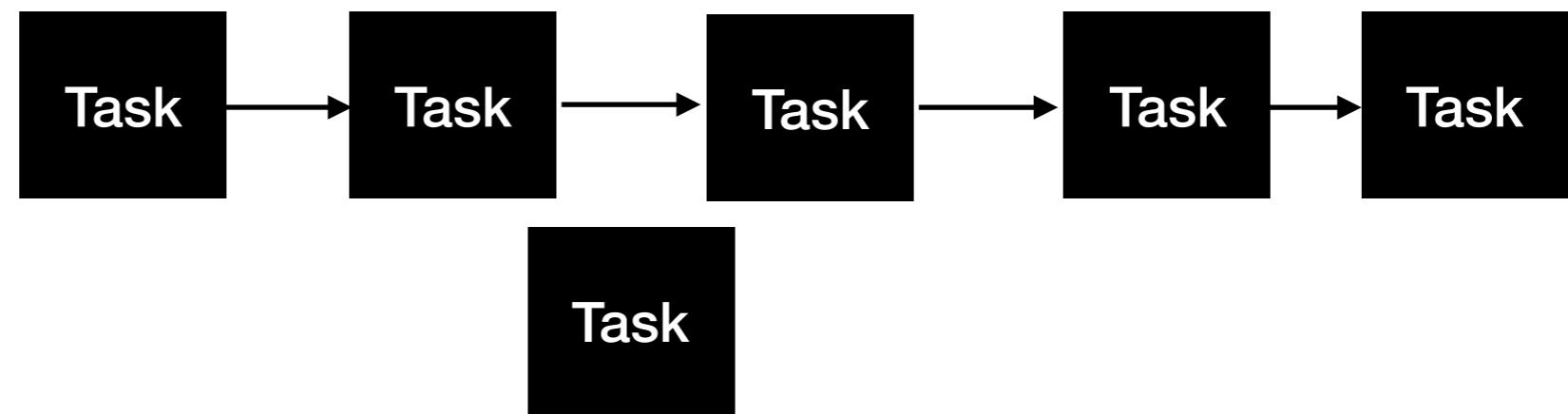
Diabetes and asthma share a lot in common

Networking engineering is a complicated task, which involves software, firmware, chip level engineering, hardware, and electric pulses. To ease network engineering, the whole networking concept is decomposed into more manageable parts.

Eclipse IDE. Downloading the basic Eclipse product provides you little more than a fancy editor. However, once you start adding plug-ins, it becomes a highly customizable and useful product.

Internet browsers plug-ins add additional capabilities that are not otherwise found in the basic browser

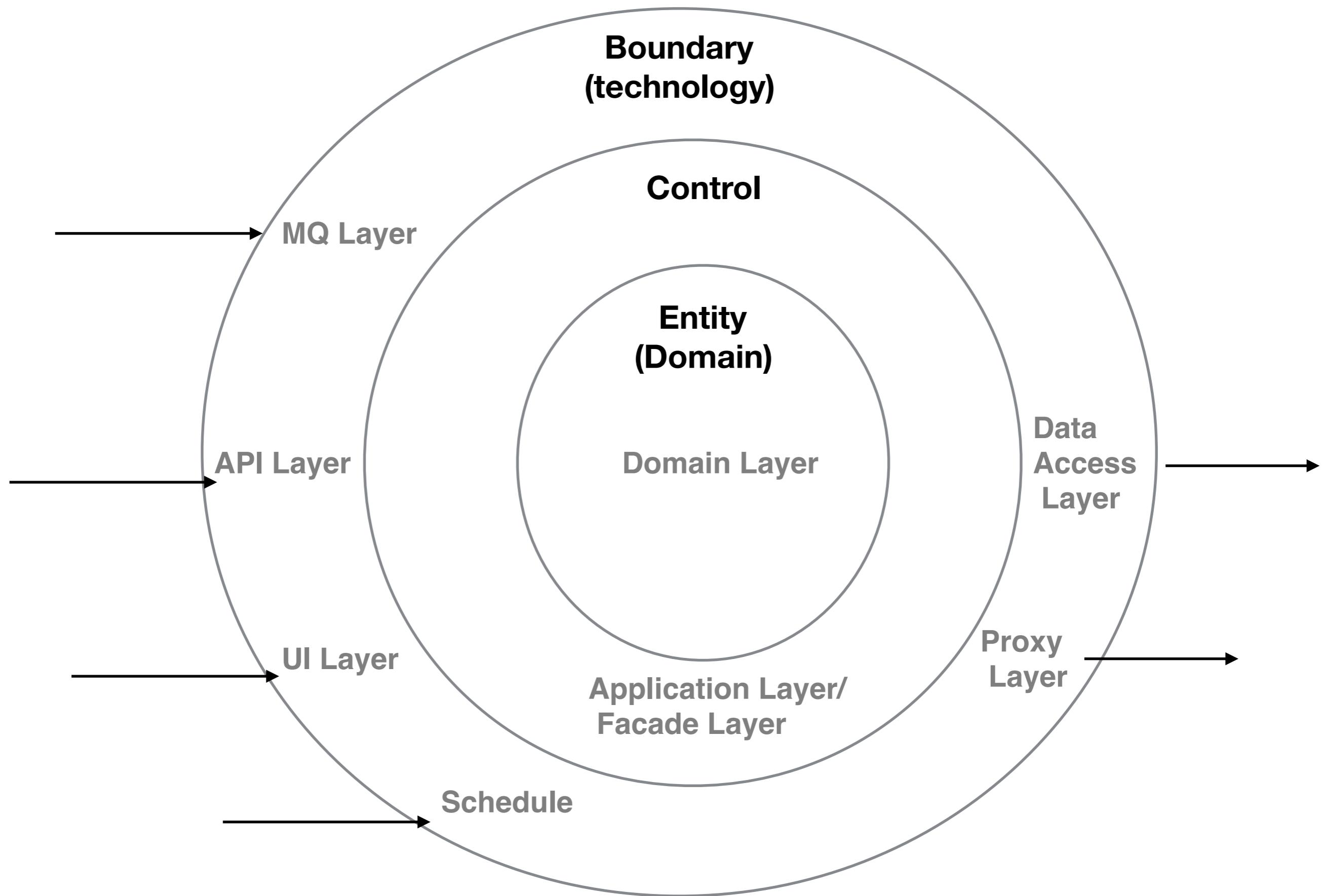
An application is required to perform a variety of tasks of varying complexity on the information that it processes. The processing tasks performed by each module, or the deployment requirements for each task, could change as business requirements are updated. Also, additional processing might be required in the future, or the order in which the tasks performed by the processing could change. A solution is required that addresses these issues, and increases the possibilities for code reuse.

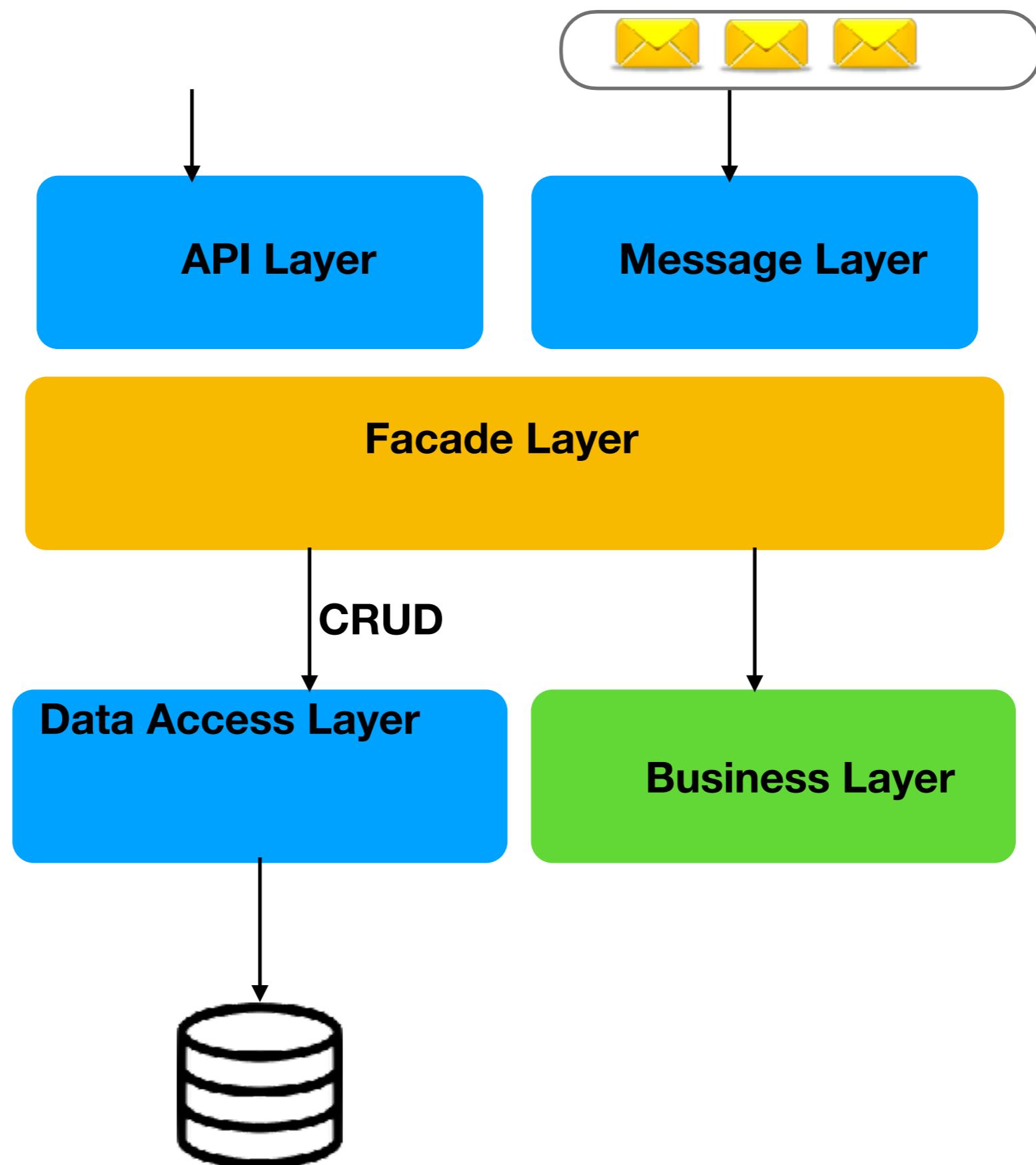


A workflow implementation. The implementation of a workflow contains concepts like the order of the different steps, evaluating the results of steps, deciding what the next step is, etc.

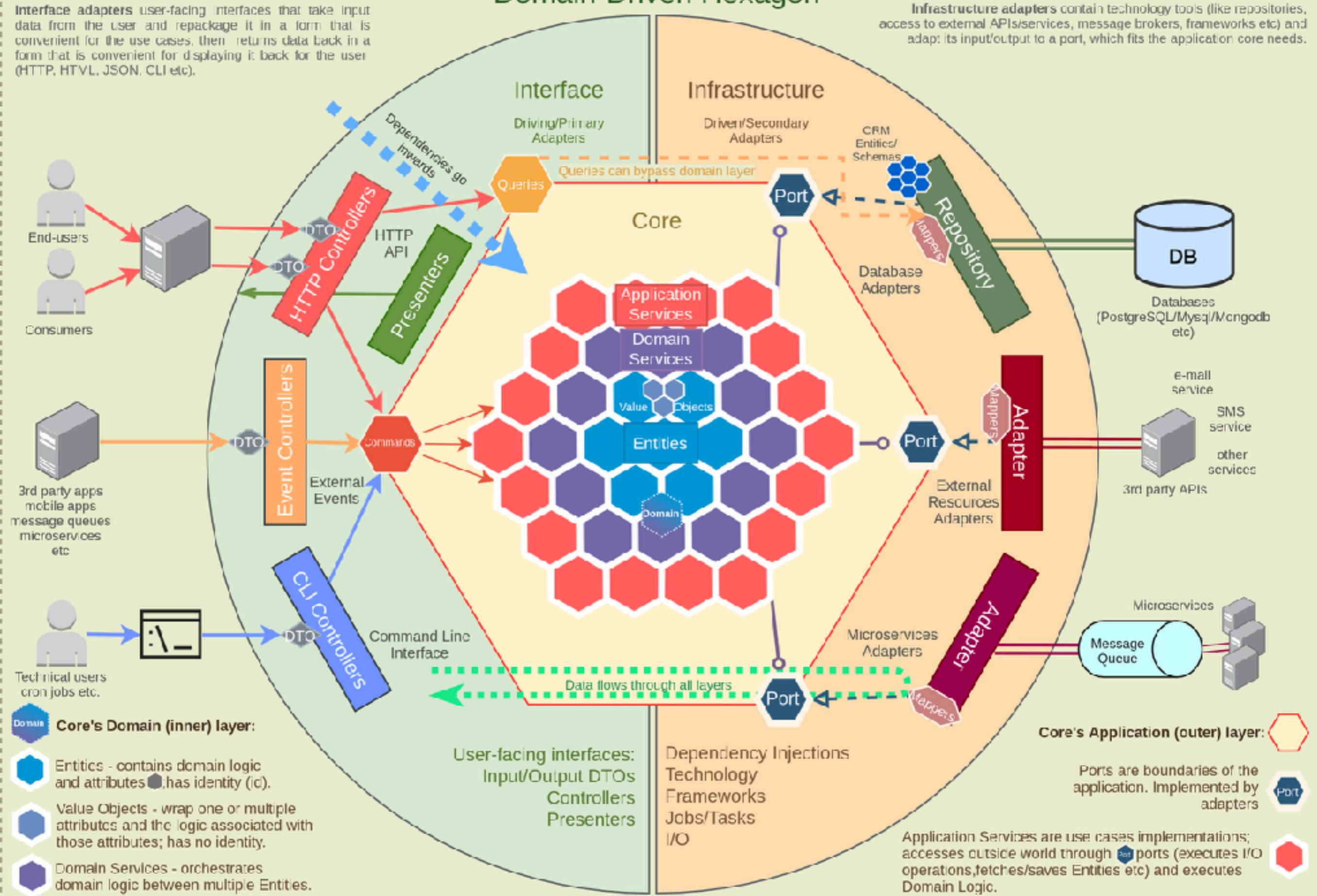
A task scheduler. A scheduler contains all the logic for scheduling and triggering tasks

Hexagonal Arch

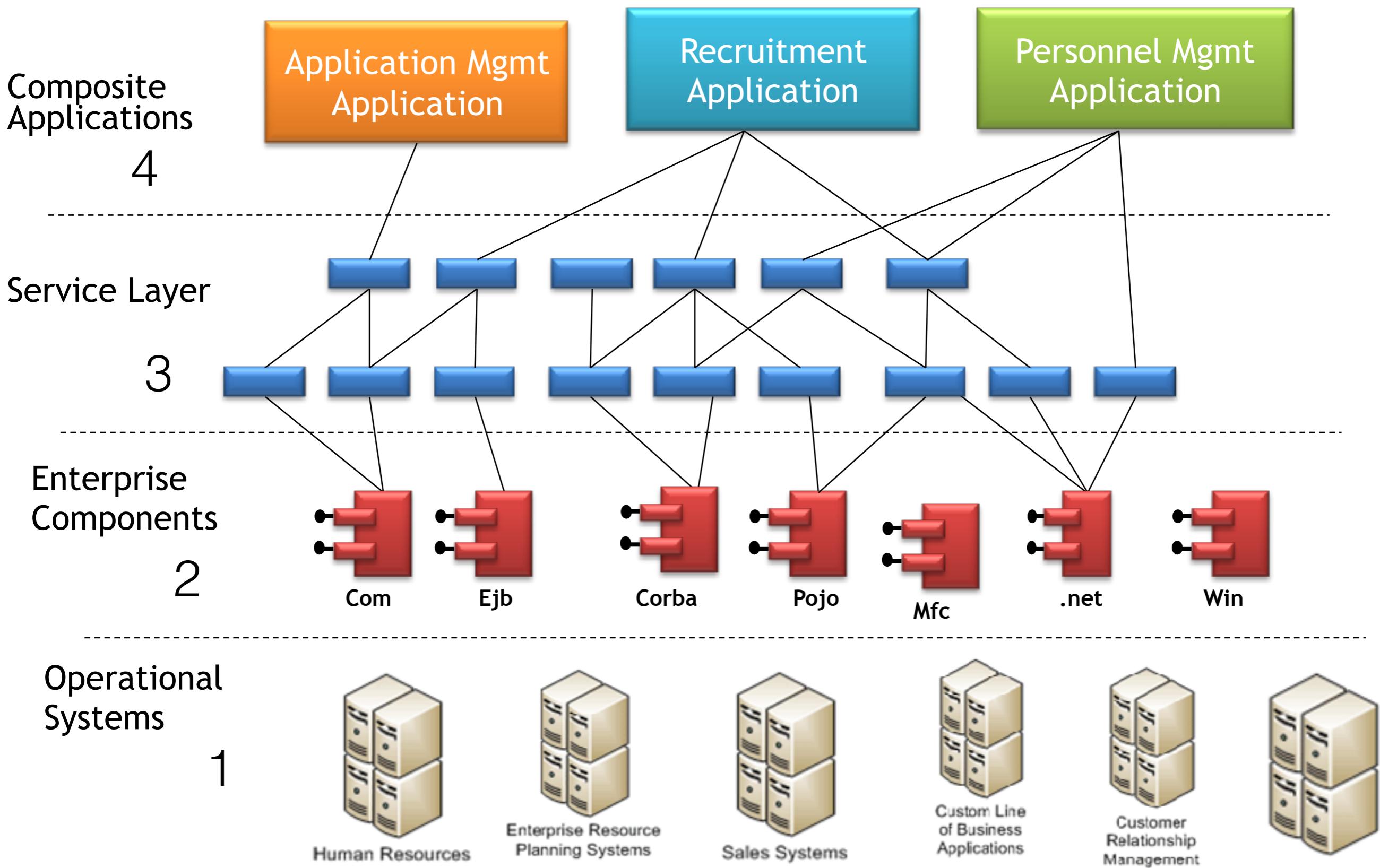


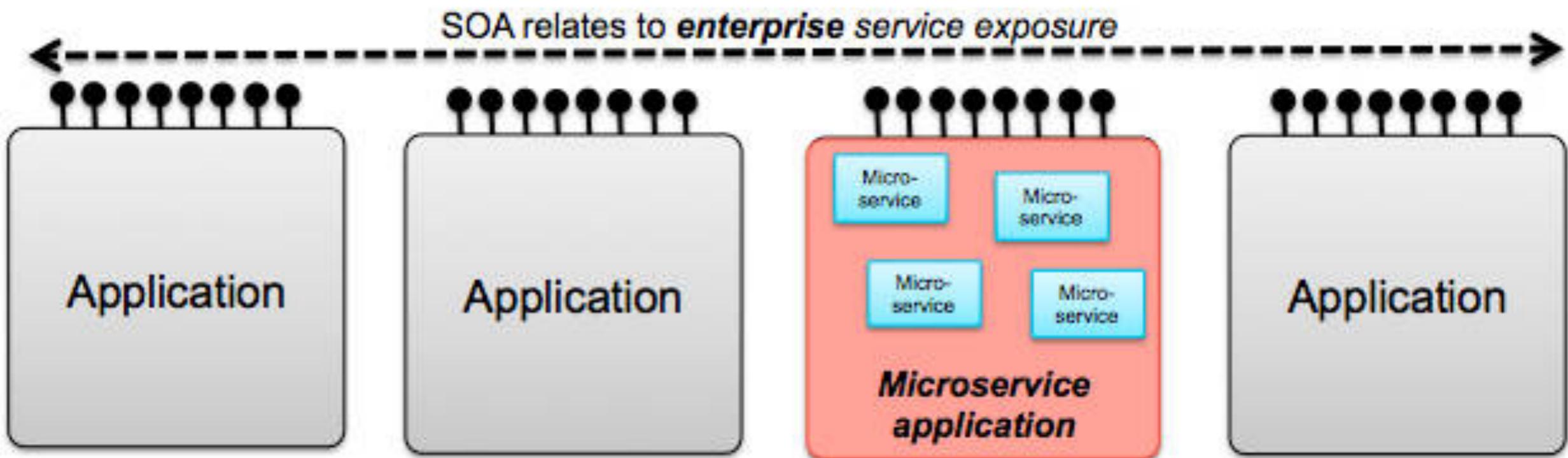


Domain-Driven Hexagon

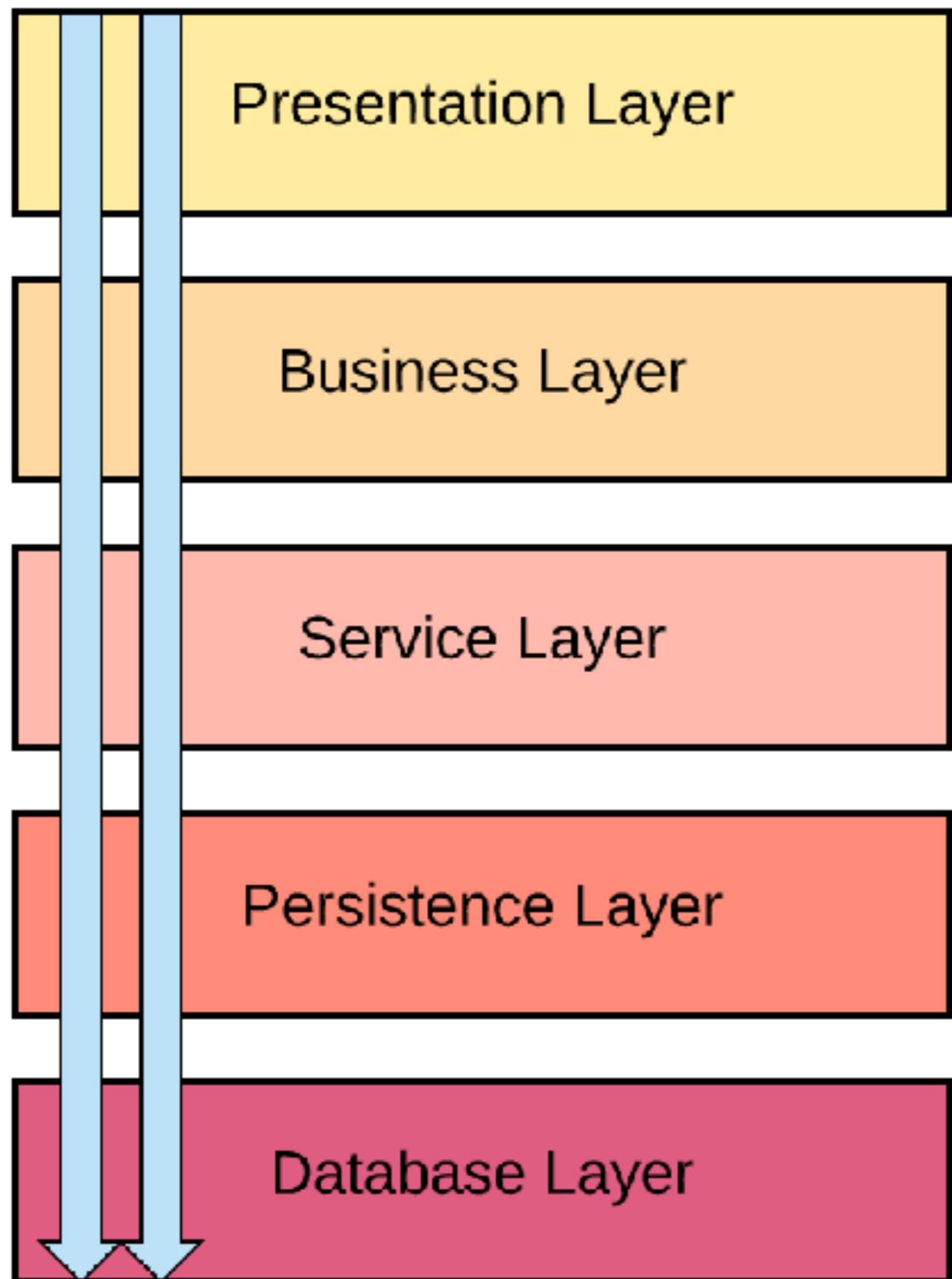


Service Oriented Architecture

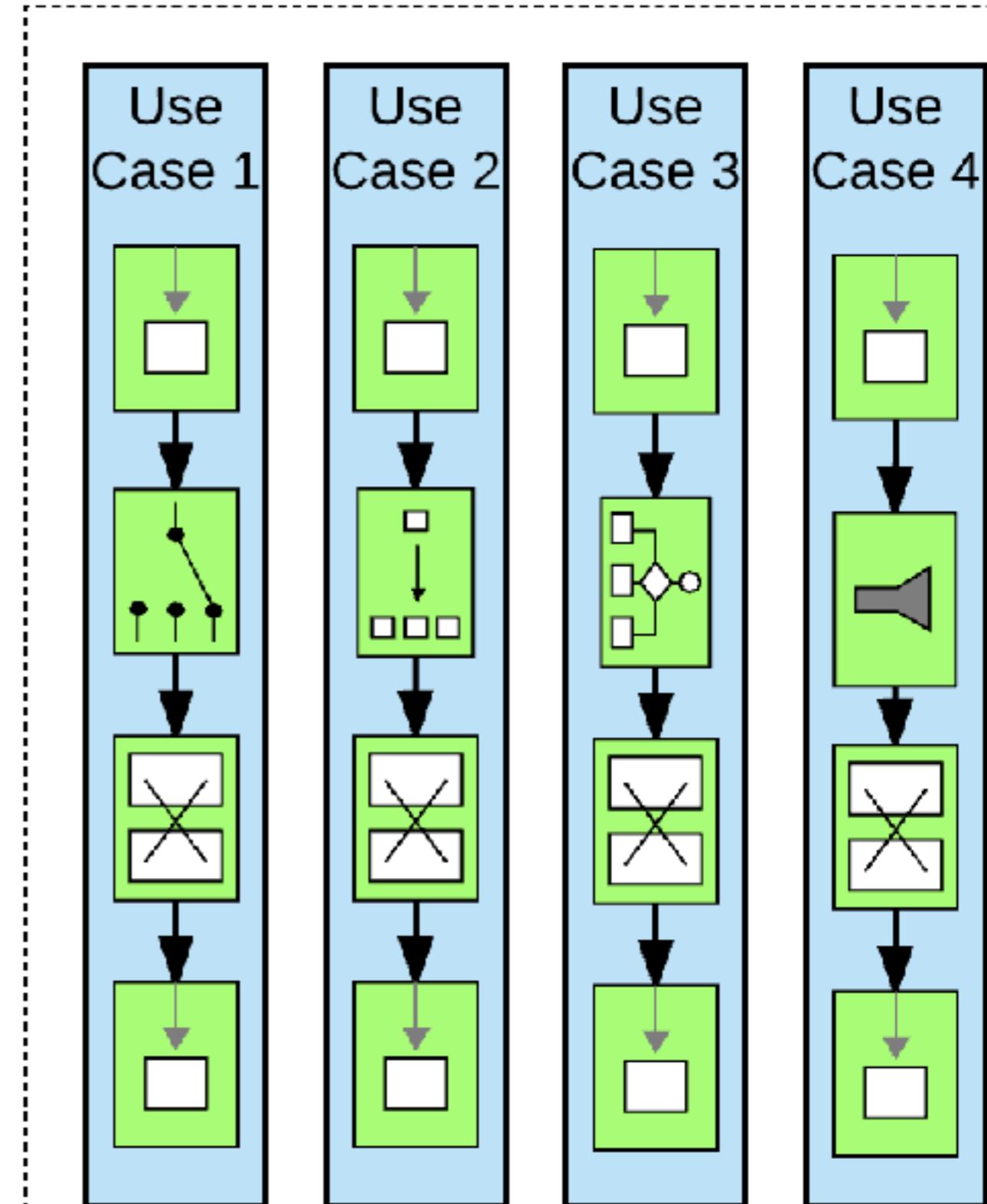




Microservices relate to
application architecture



Layered Architecture



Pipes and Filters

Core

Layer

Layer

Filter

Filter

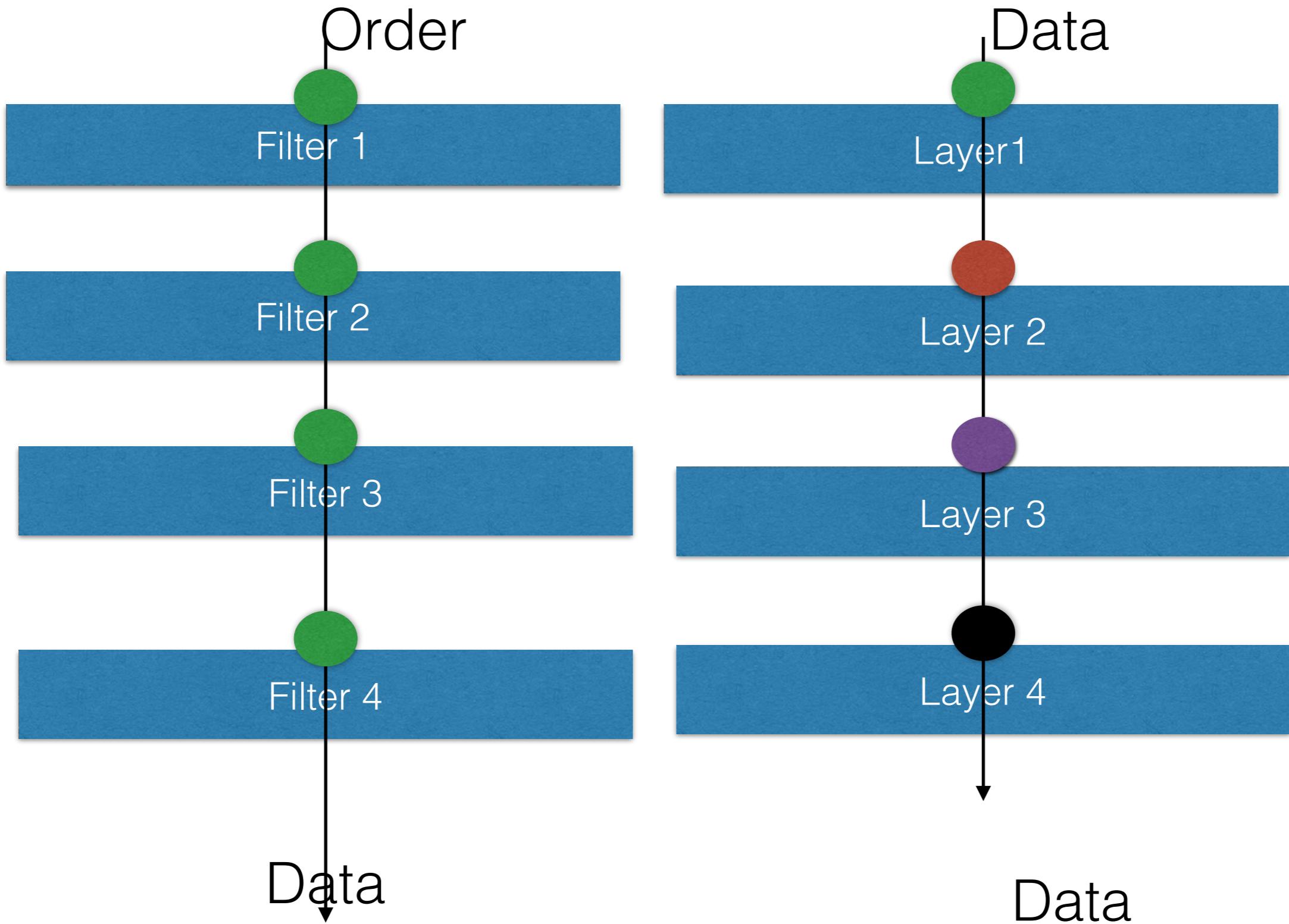
Filter

Layer

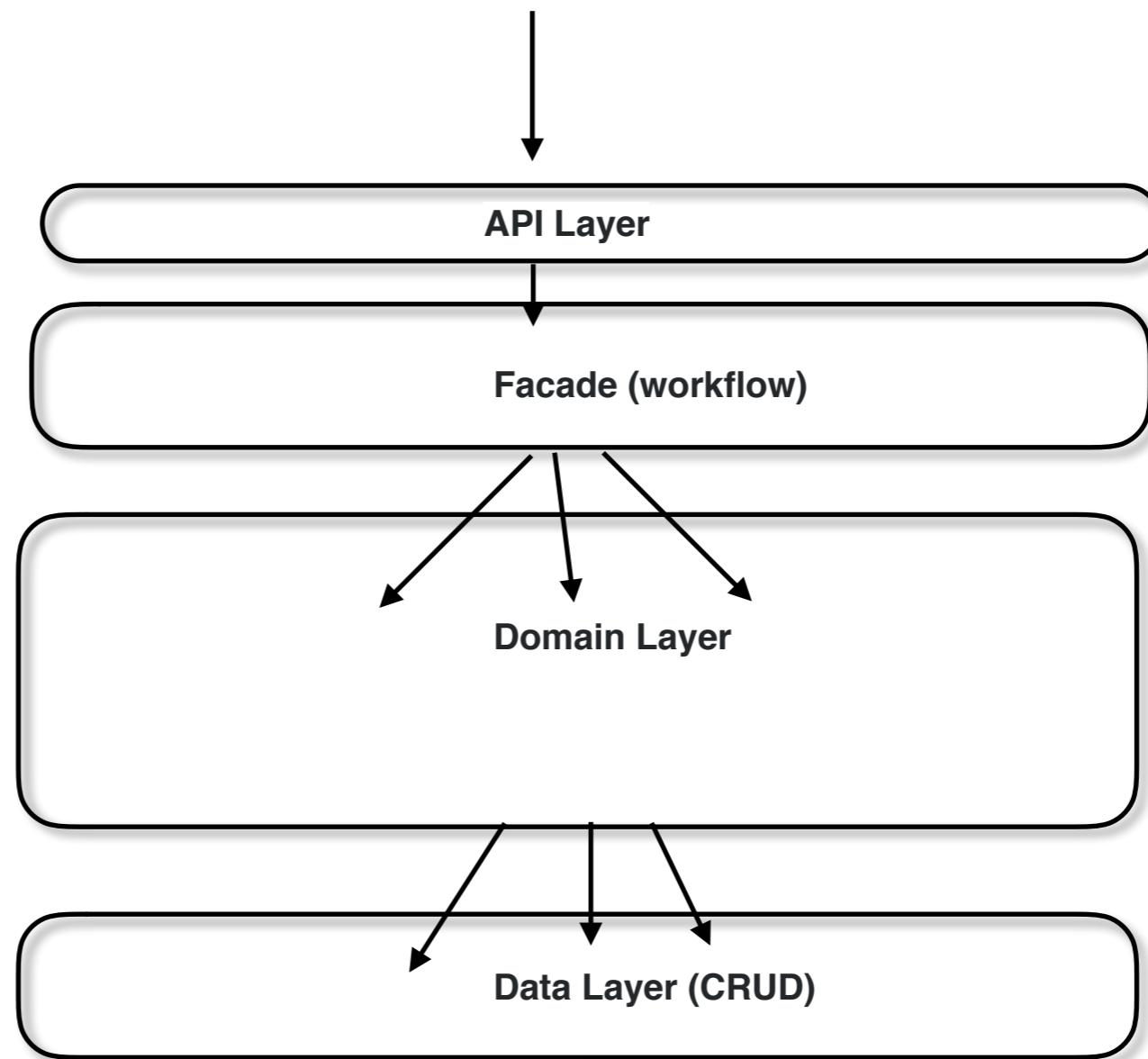
Plugins

Plugins

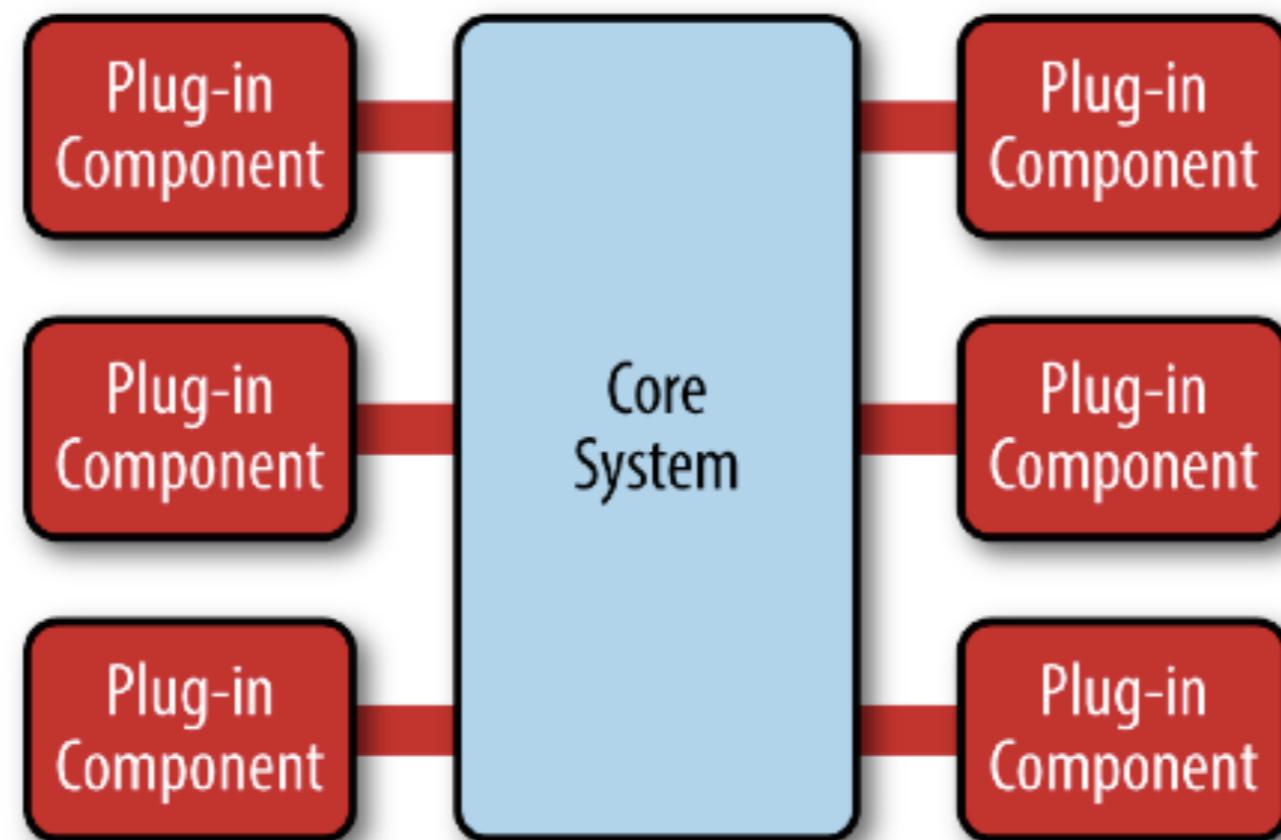
Plugins

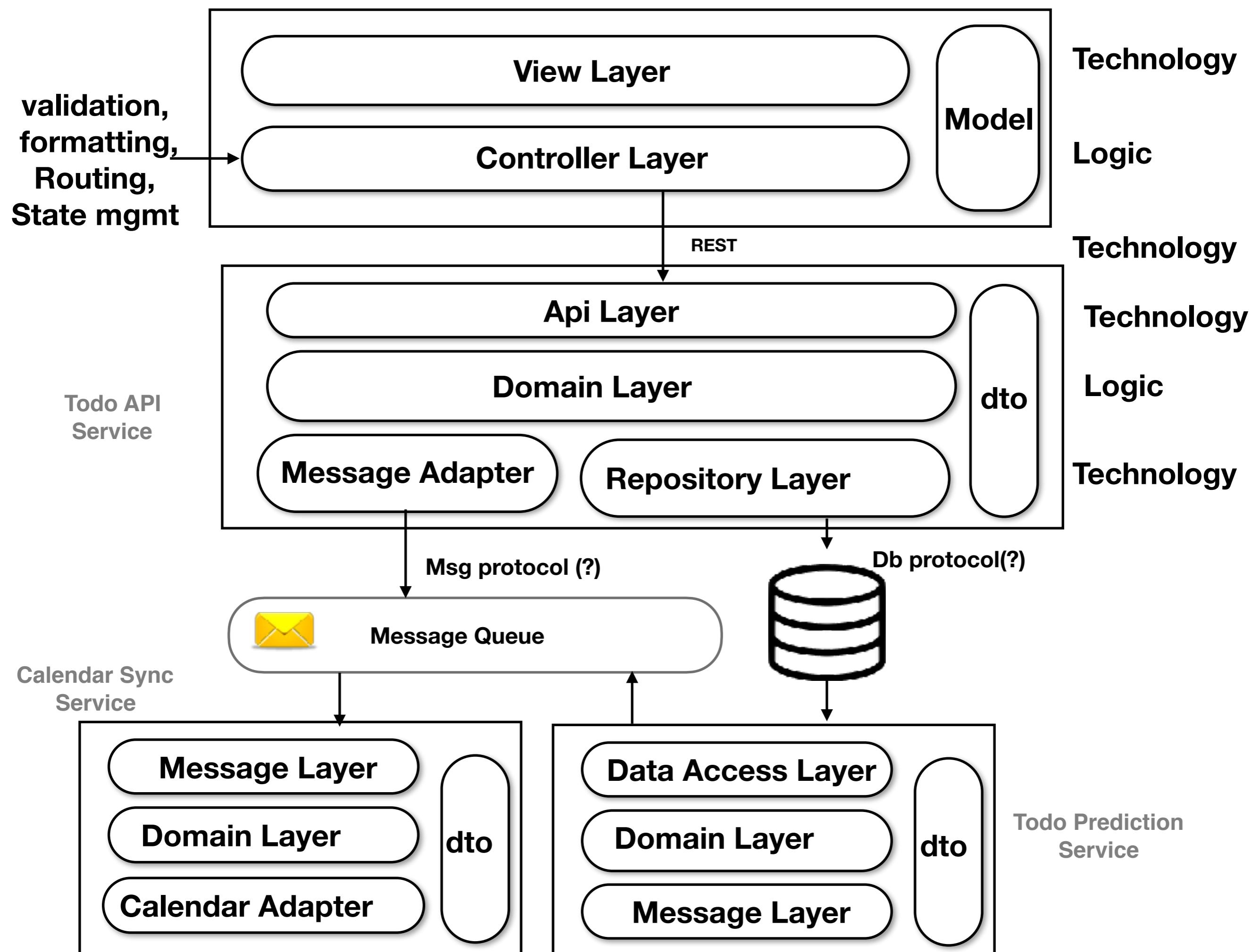


API App



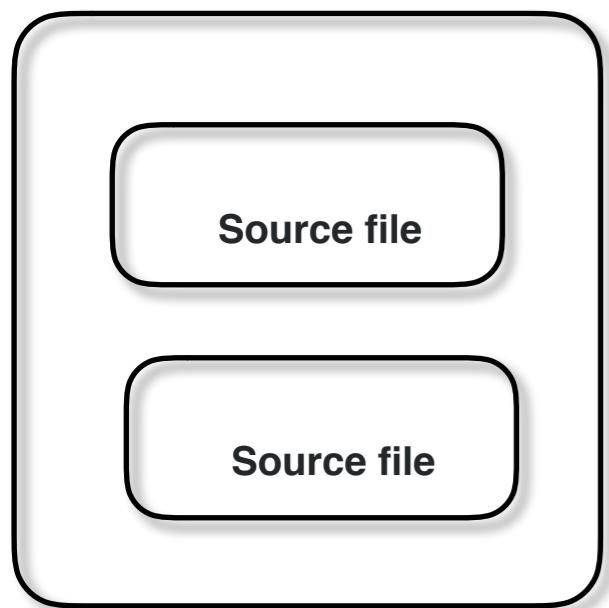
Microkernel Architecture



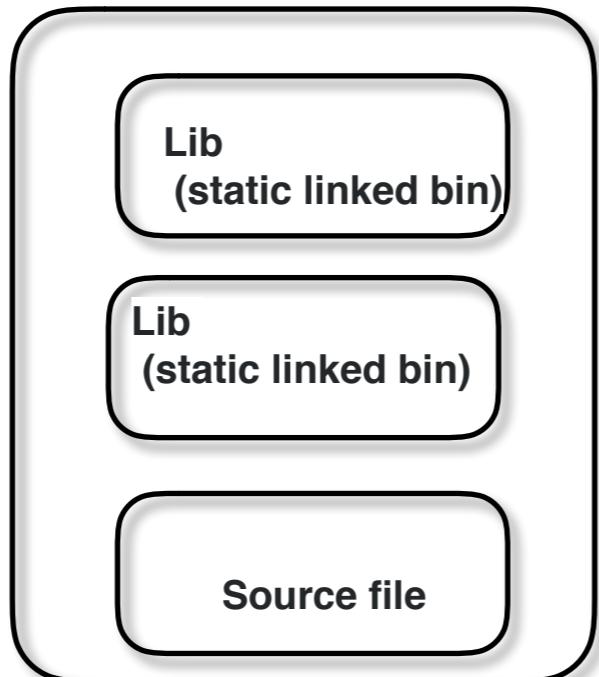


1

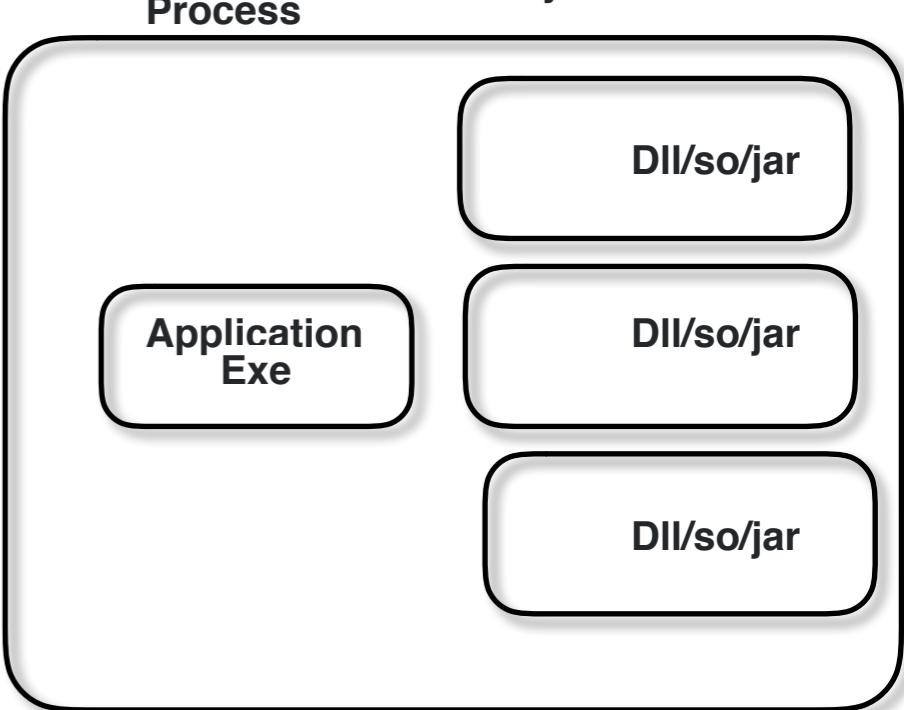
**Single Mono
Binary**

**2**

**Single Mono
Binary file**

**3**

**Multiple
Binary file**



**Compile time
monolithic**

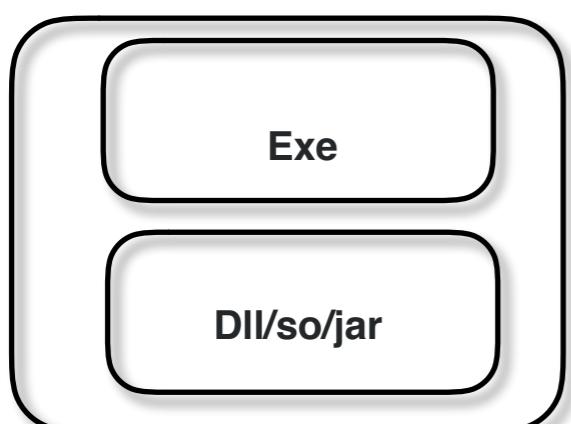
**Link time
monolithic**

**Runtime
monolithic**

In process
comp

4

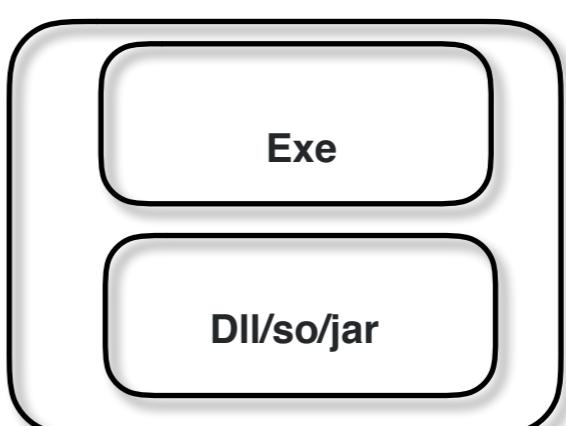
Process



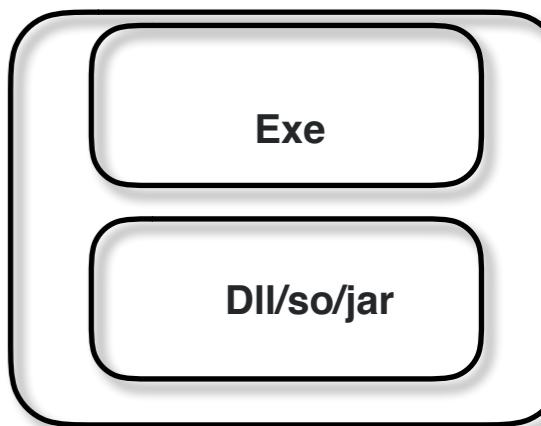
Process

**5**

Process



Process



Module1

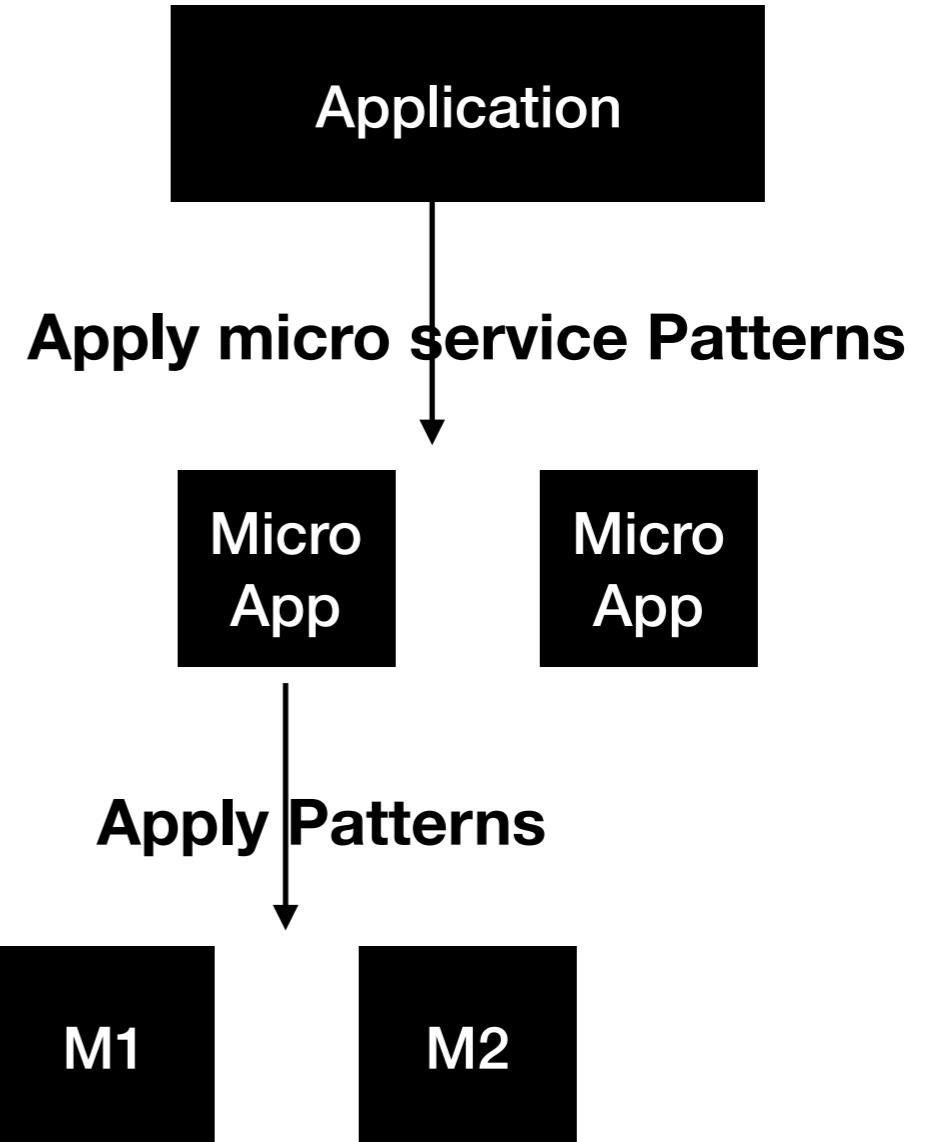
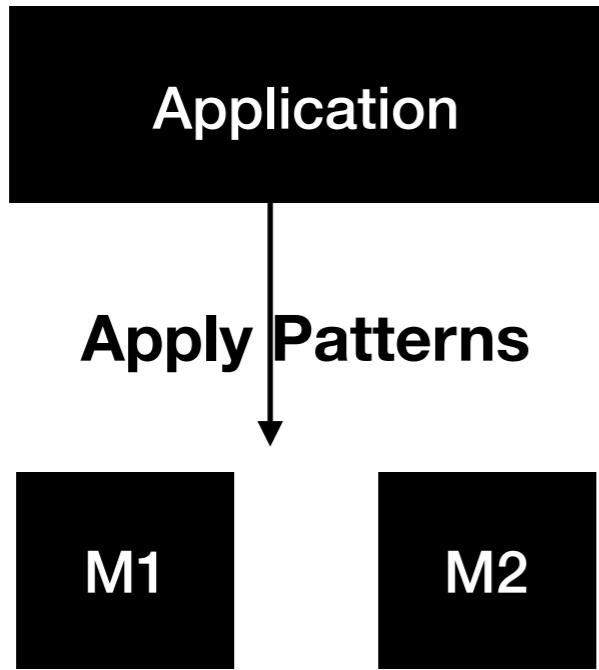
Module2

Micro App1

Micro App2

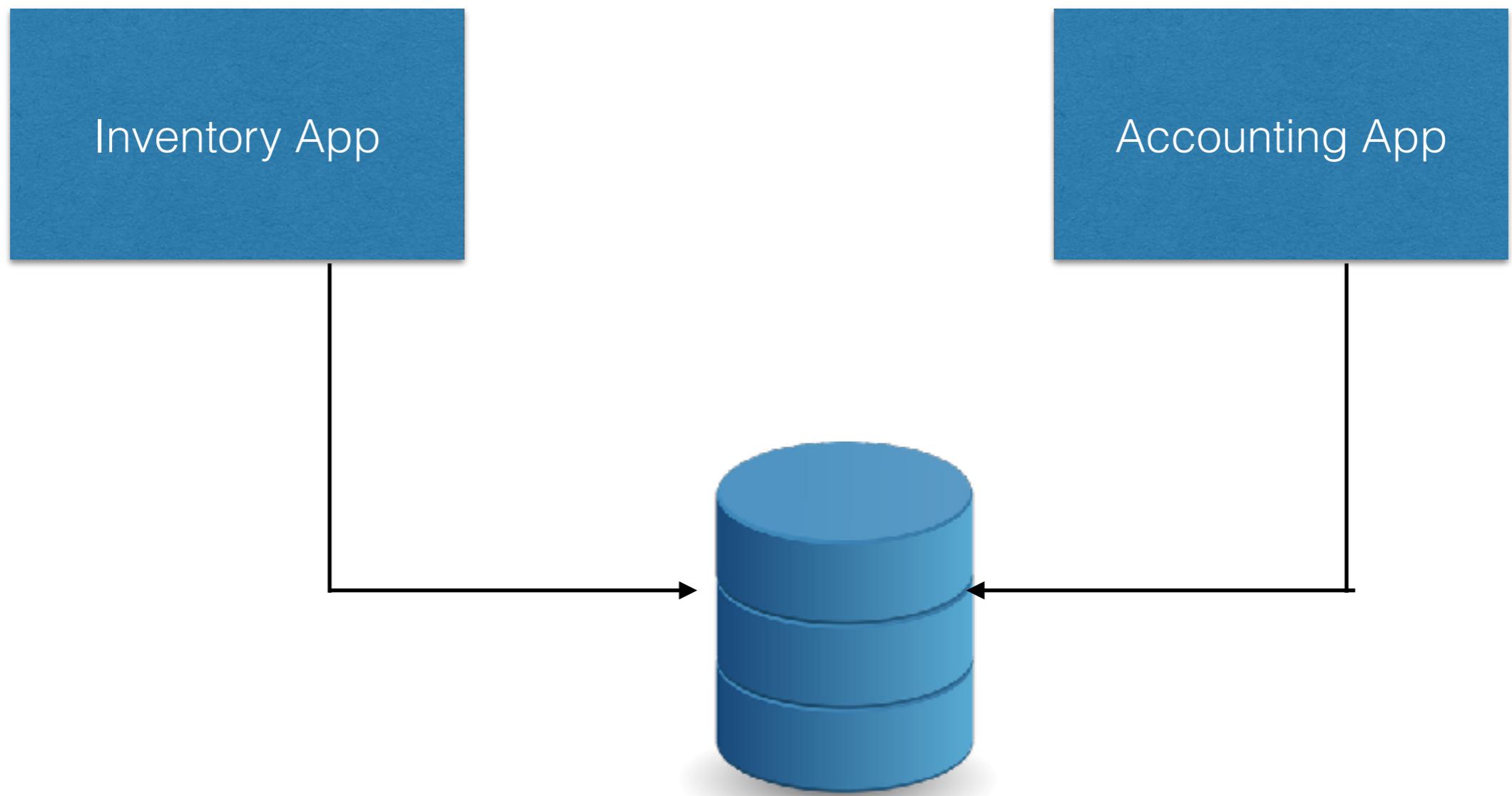
**Distributed Application
(group of modules)**

**Micro Application
(group of small Apps)**

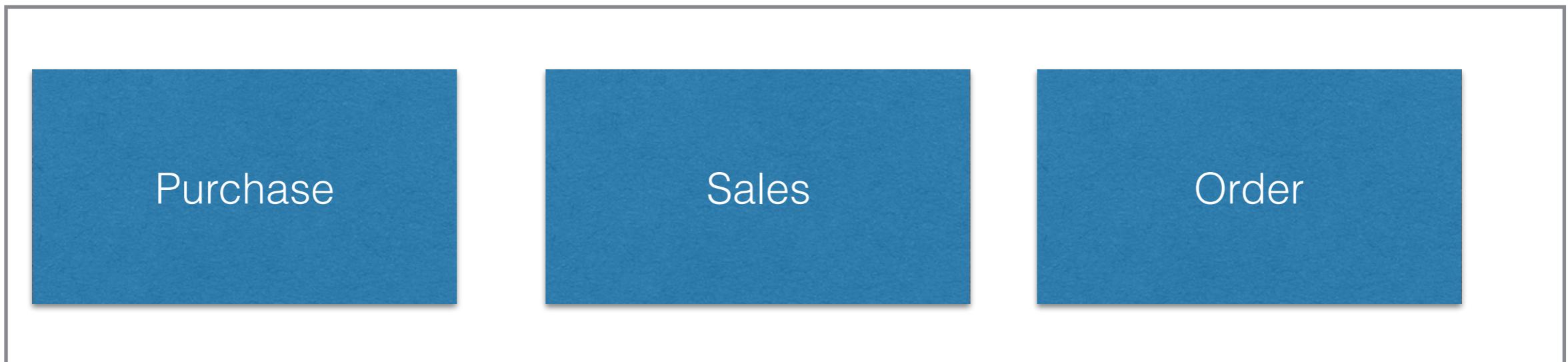


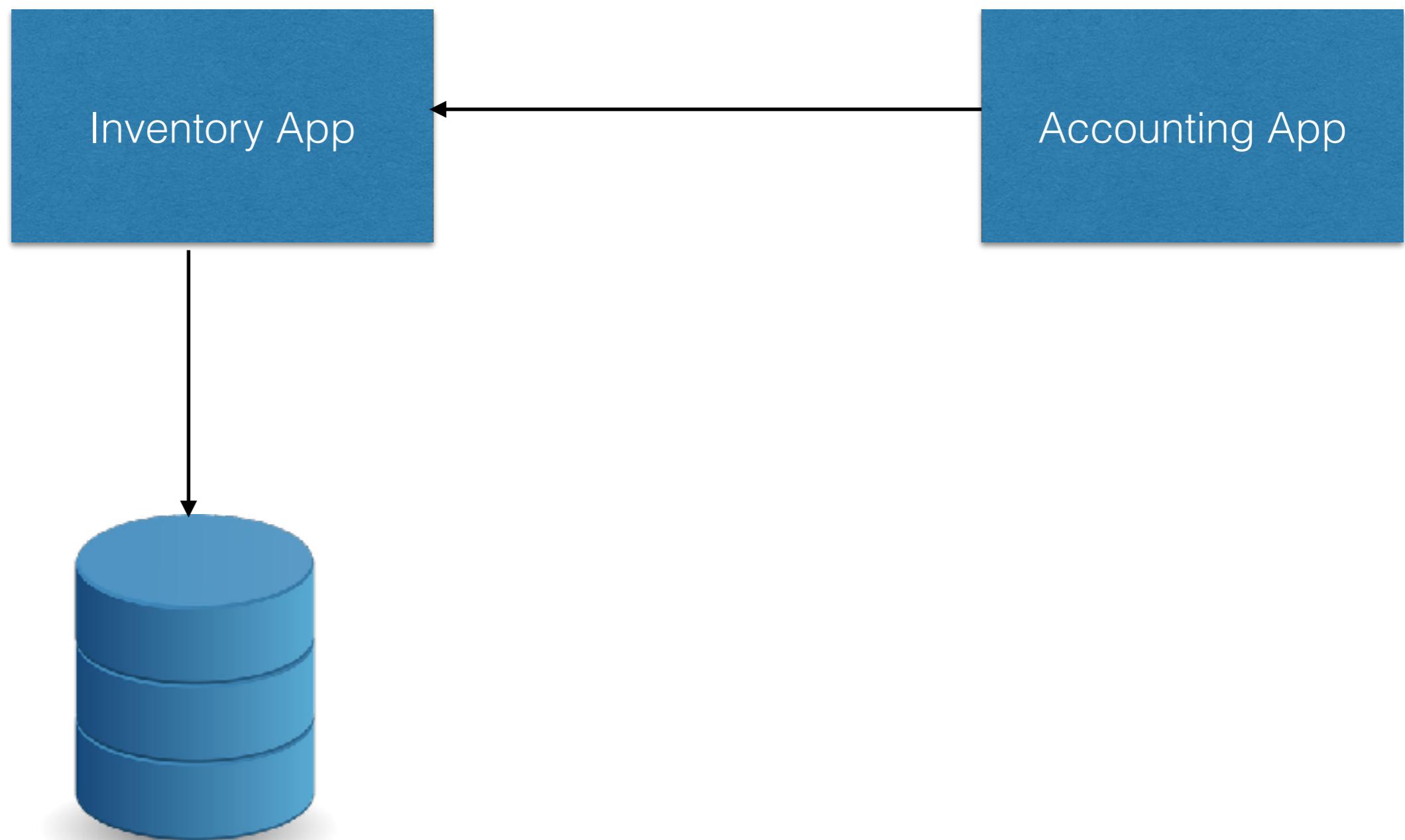
Decompose into smaller manageable piece

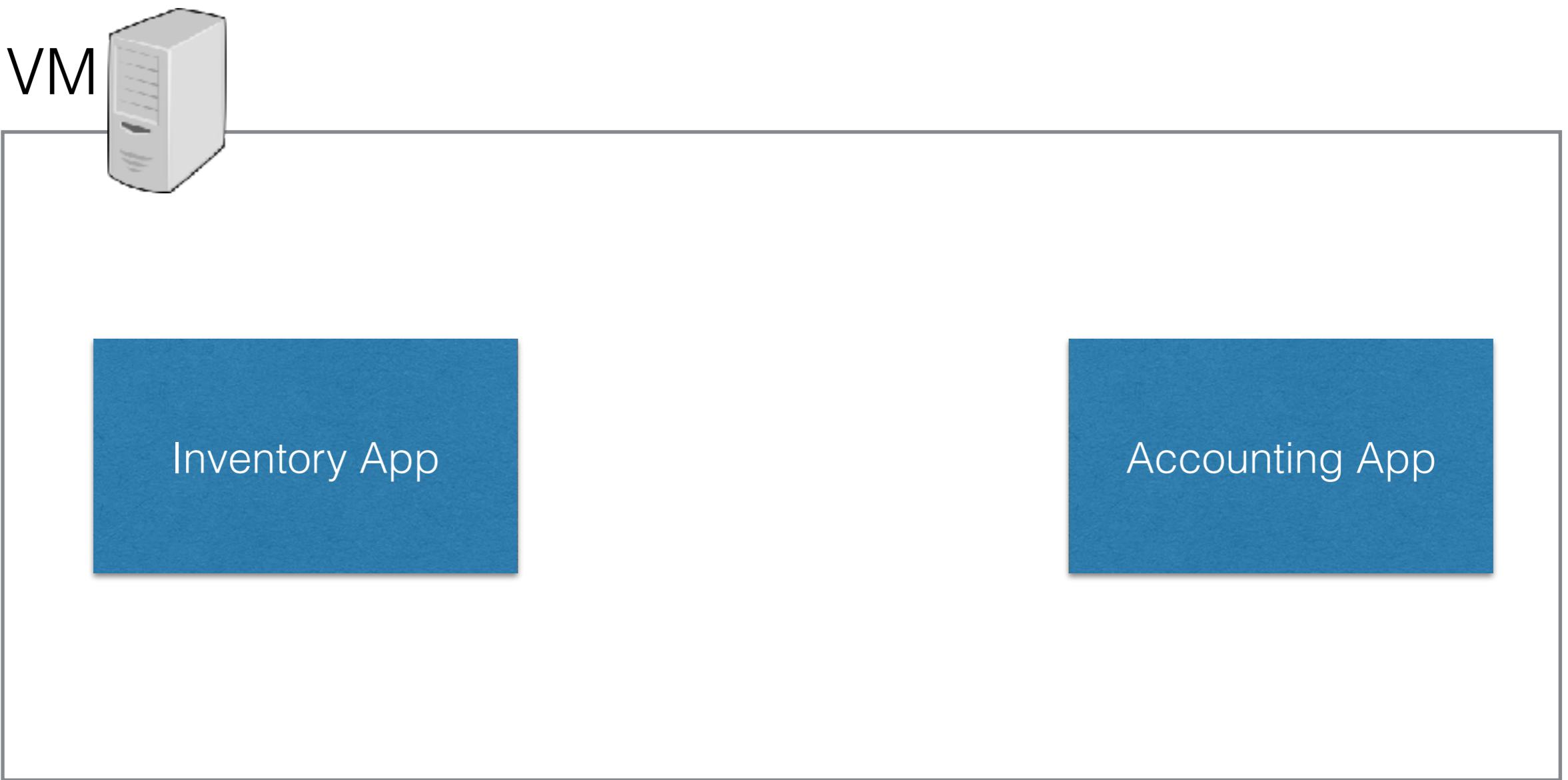
	Binary 1	Binary 2	W	Score
	2 Modules	2 Applications		
Database / Storage	Shared	Its own	2	
Infra (Hosting)	Shared	Its own	3	
Sorce Control	Shared	Its own	2	
CI/CD (Build Server)	Shared	Its own	3	
Common Code	Shared	Domain (no) Tech (~ yes) Discount Date		
Fun Requirements	Shared	Its own	1	
SCRUM Team / Sprint	Shared	Its own	1	
Test Cases	Shared	Its own	1	
Architecture	Shared	Its own	1	
Technology Stack / Fwks	Shared	Its own	1	

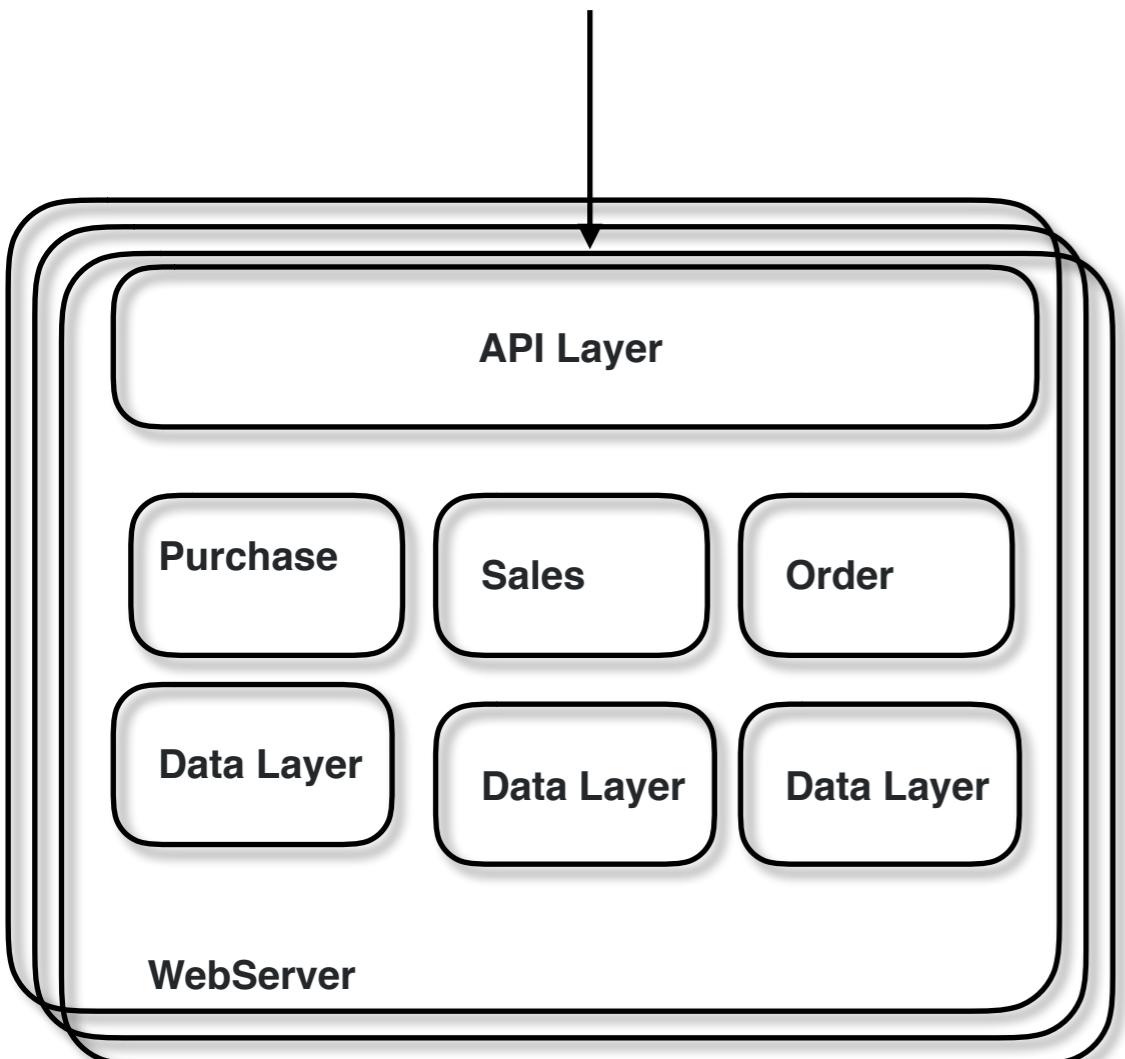


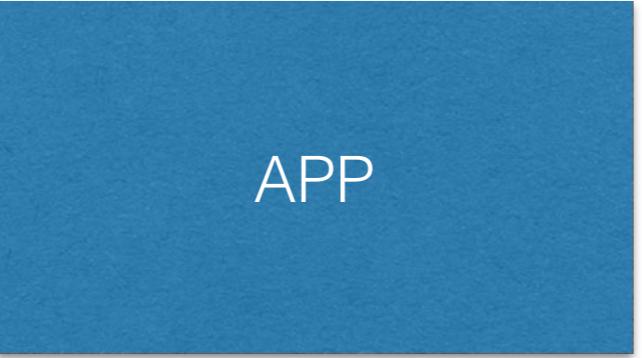
Inventory Application







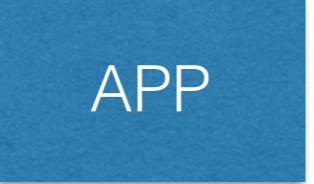




APP



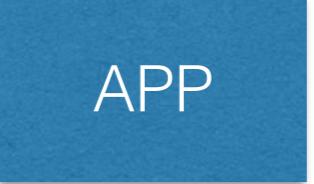
APP



APP

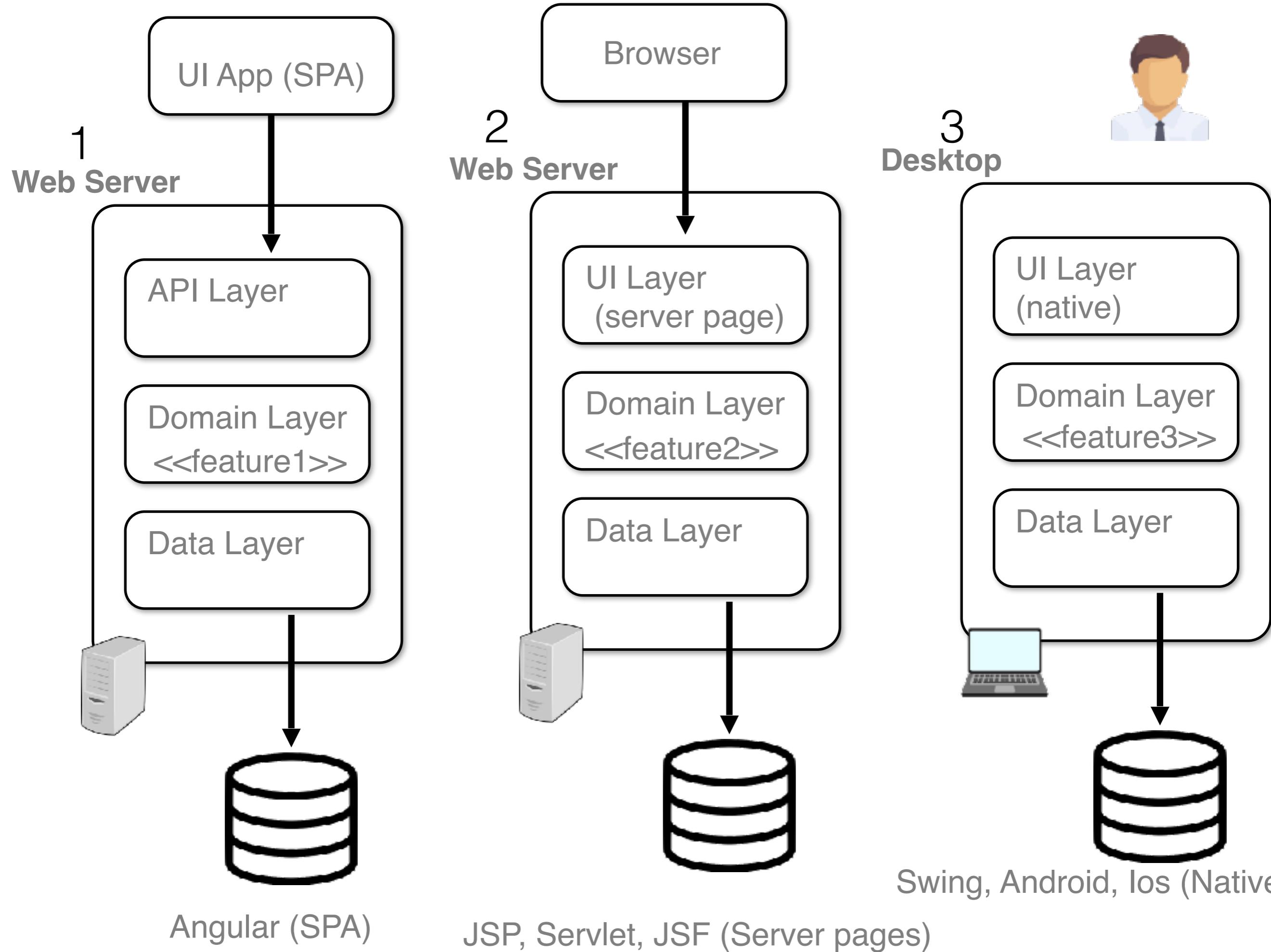


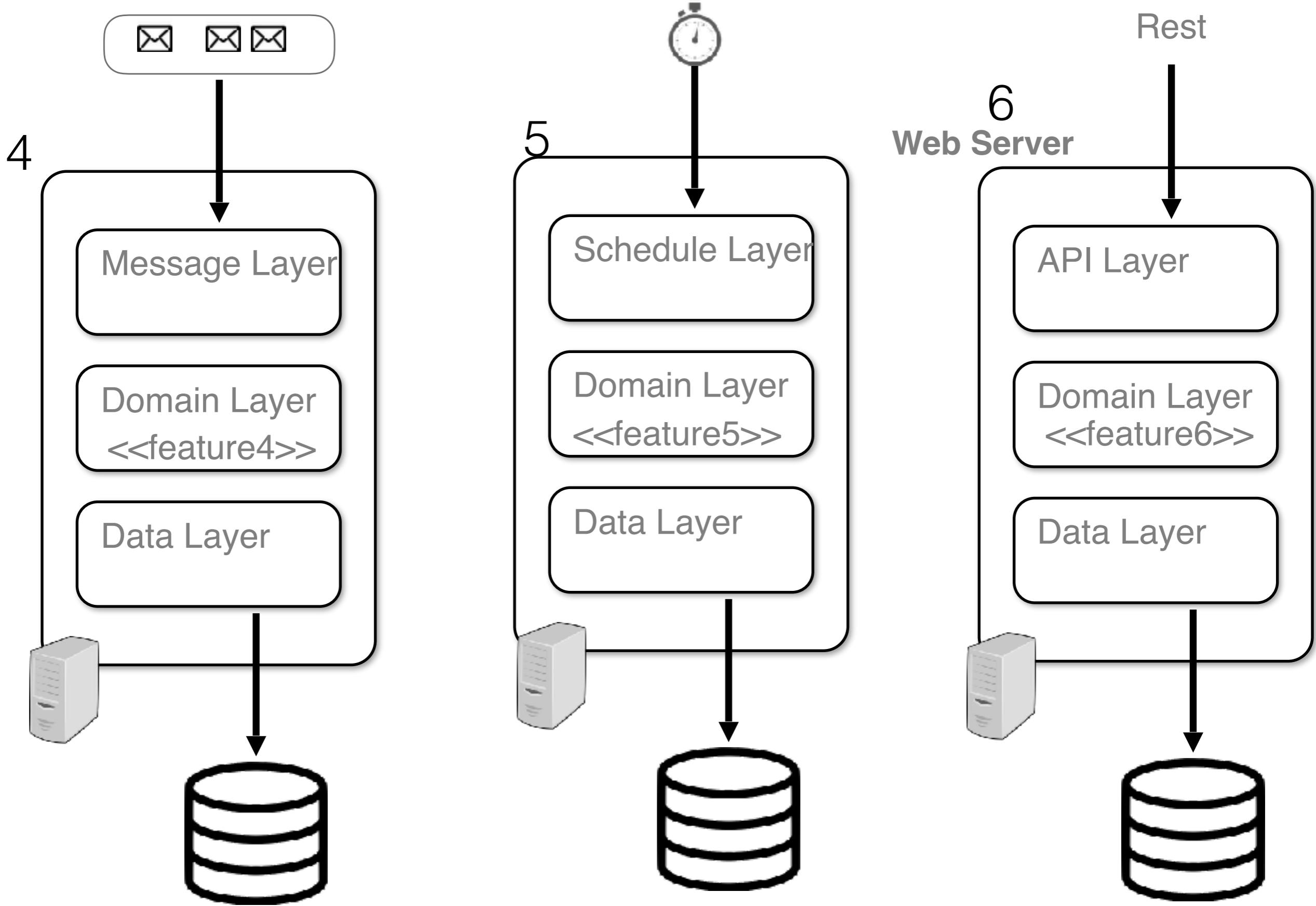
APP

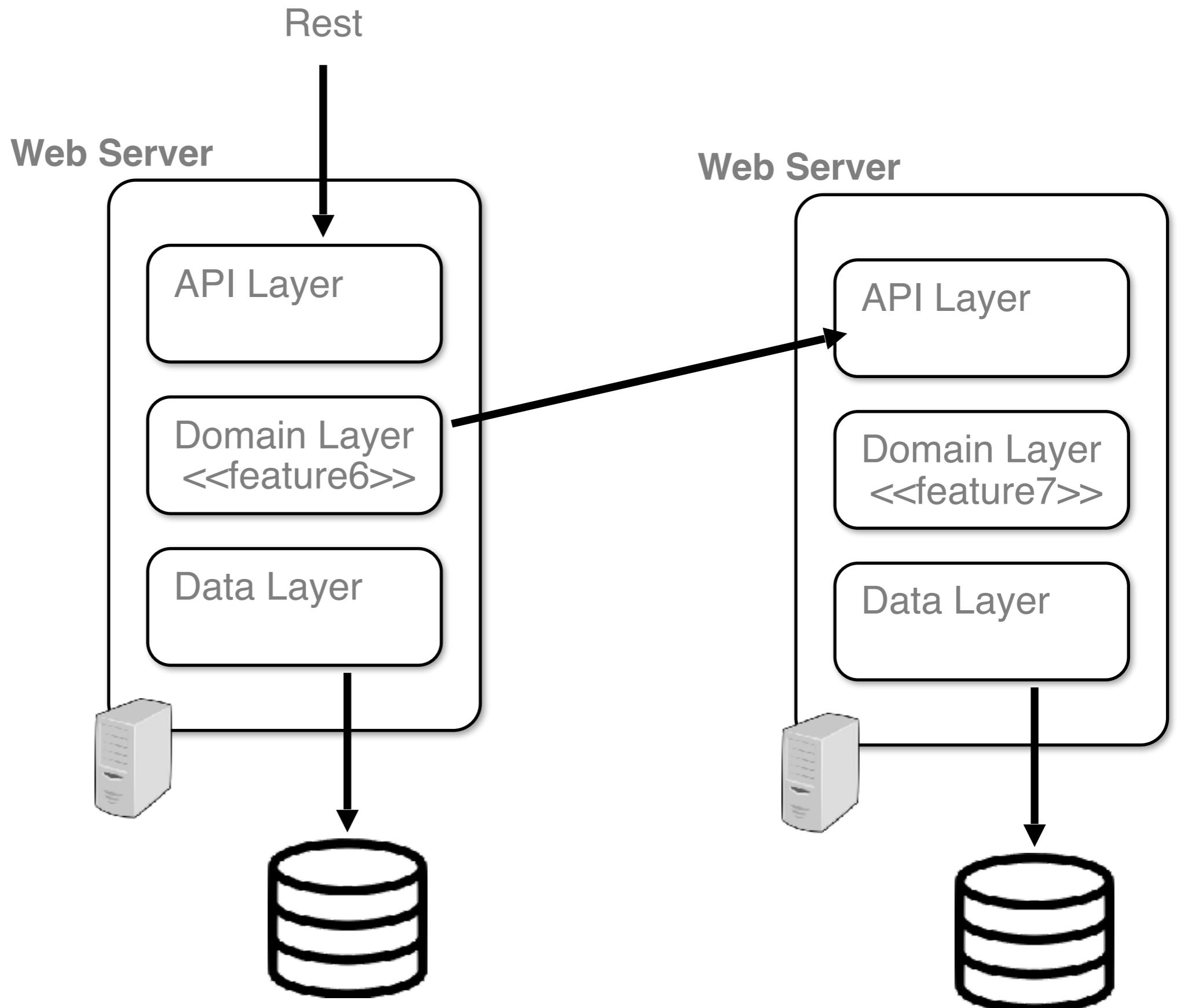


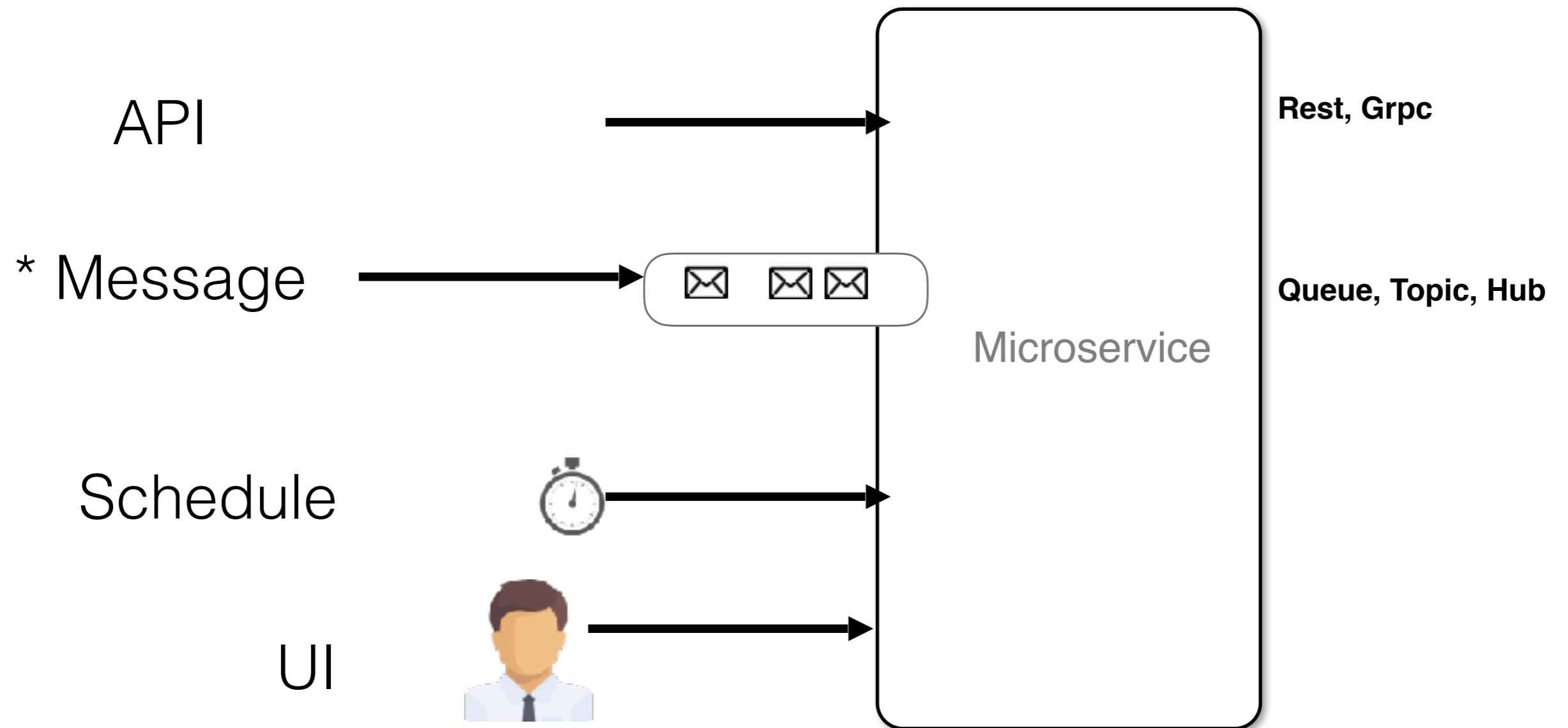
APP

Types of Microservice









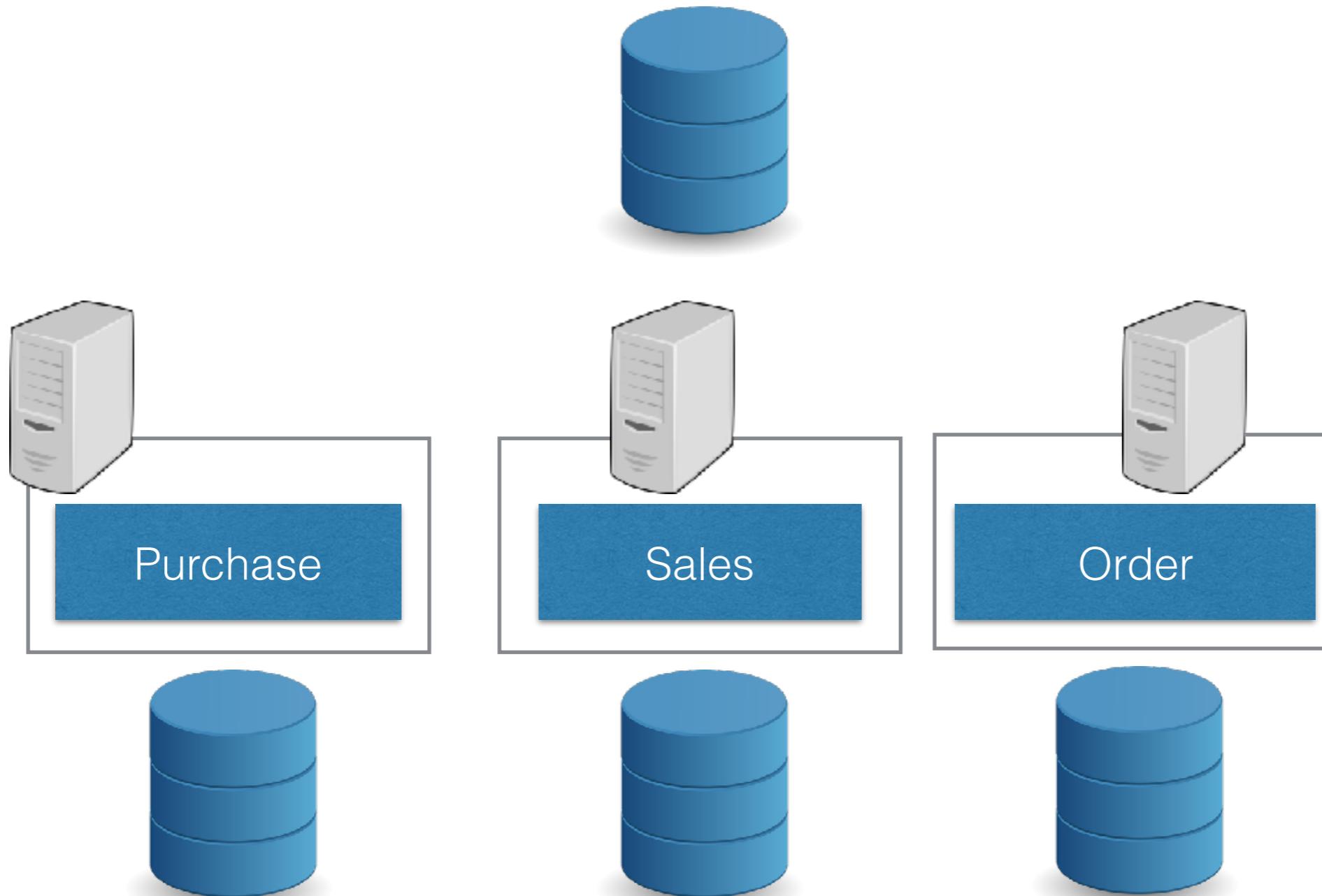
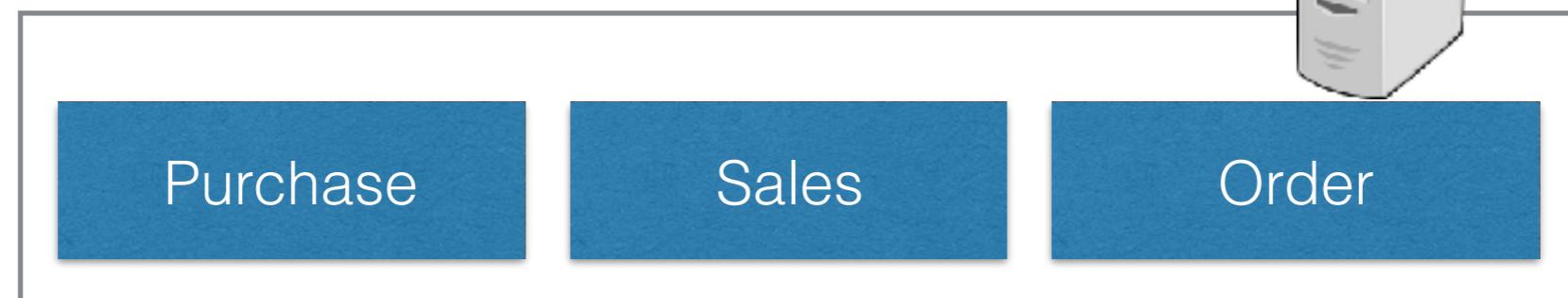
	Pros/ Cons	Solution
Development time	--	
Micro-service practices Learning Curve	--	
Resource Performance (CPU, Memory, I/O)	--	GRPC,
Db Transaction Management	--	SAGA / Compensatable Transaction
Views / Report / Dash board/ join	--	Materialised view
Infra Cost	--	Containerisation, K8s
Preparing for Deployment	--	ci/cd
Debugging, Error Handling (End to End)	--	Distributed tracing (jaeger)
Integration Test	-	Pyramid Test
Log Mgmt (debug/ error)	--	Centralize (elastic search) EFK, ELK, Splunk
Config Mgmt	--	Centralize (Consul, Redis, Vault)
Authentication (who)	--	Centralize (Oauth2, OpenID Connect,...)
Authorization (what can they do)	--	Centralize (Claims)
Audit Log mgmt (who accessed what)	--	Centralize
Monitoring / Alerting	--	Centralize (Kibana, Grafana, Prometheus, ...)
Data Security and Privacy (transit, storage)	--	
Build Pipeline (CI)	--	Ifra as code
Agile Architecture (Agility to change)	+++	
Feature Shipping (Agility to ship)/ CD	++	
Scalability (volume - request, data,	+	Scalability Cube
Availability	+	
Ability to do Polygot	+	

Decentralize
functionality

Centralize
Technical Concerns

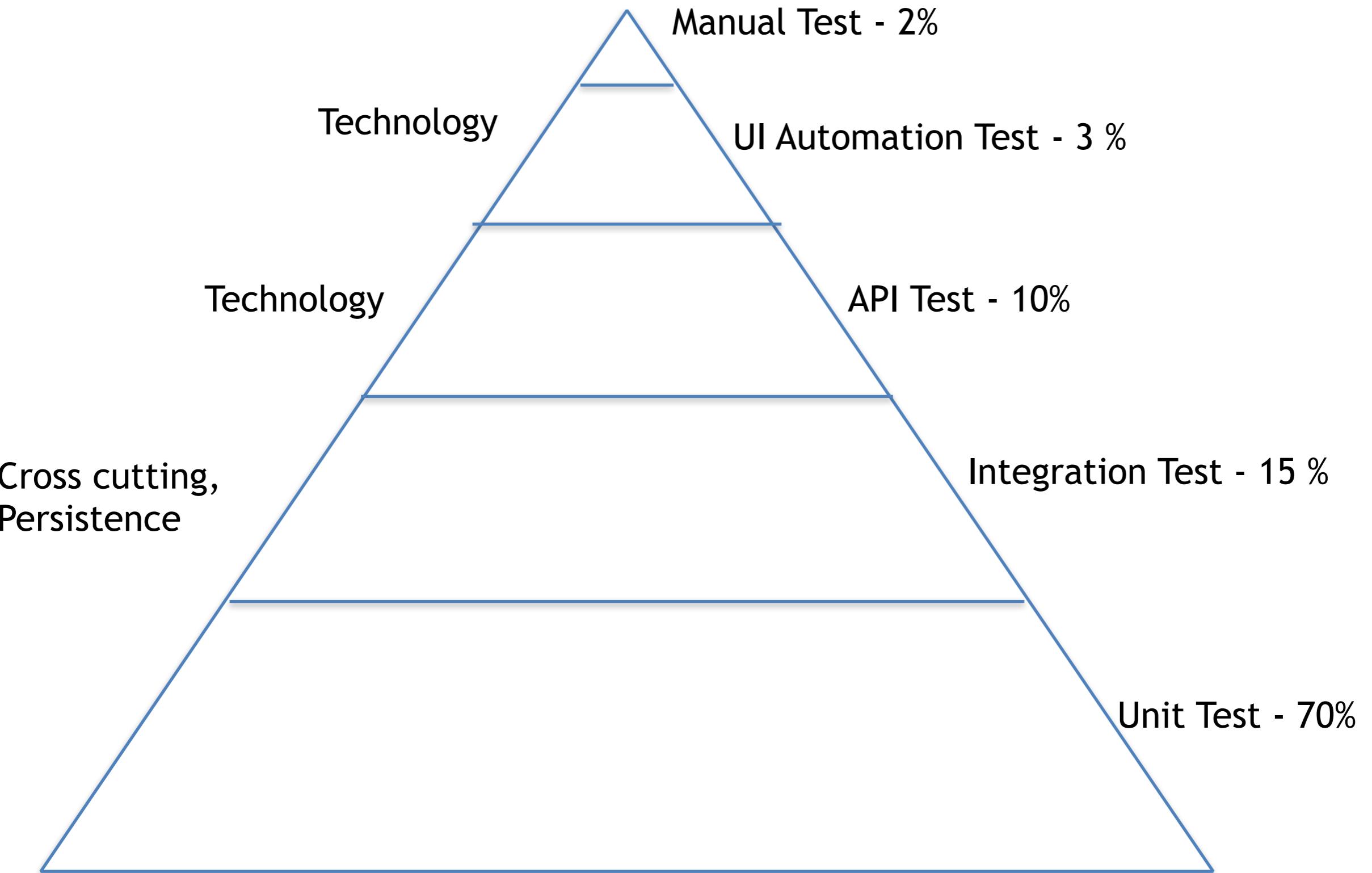
- Developer config -> (development) constants, types, ...
- Operational config -> (deployment) user name, pwds, host, port, ...
- User config -> (runtime) preference, profile, ...

Inventory Application



Decentralize Domain
Centralize Tech

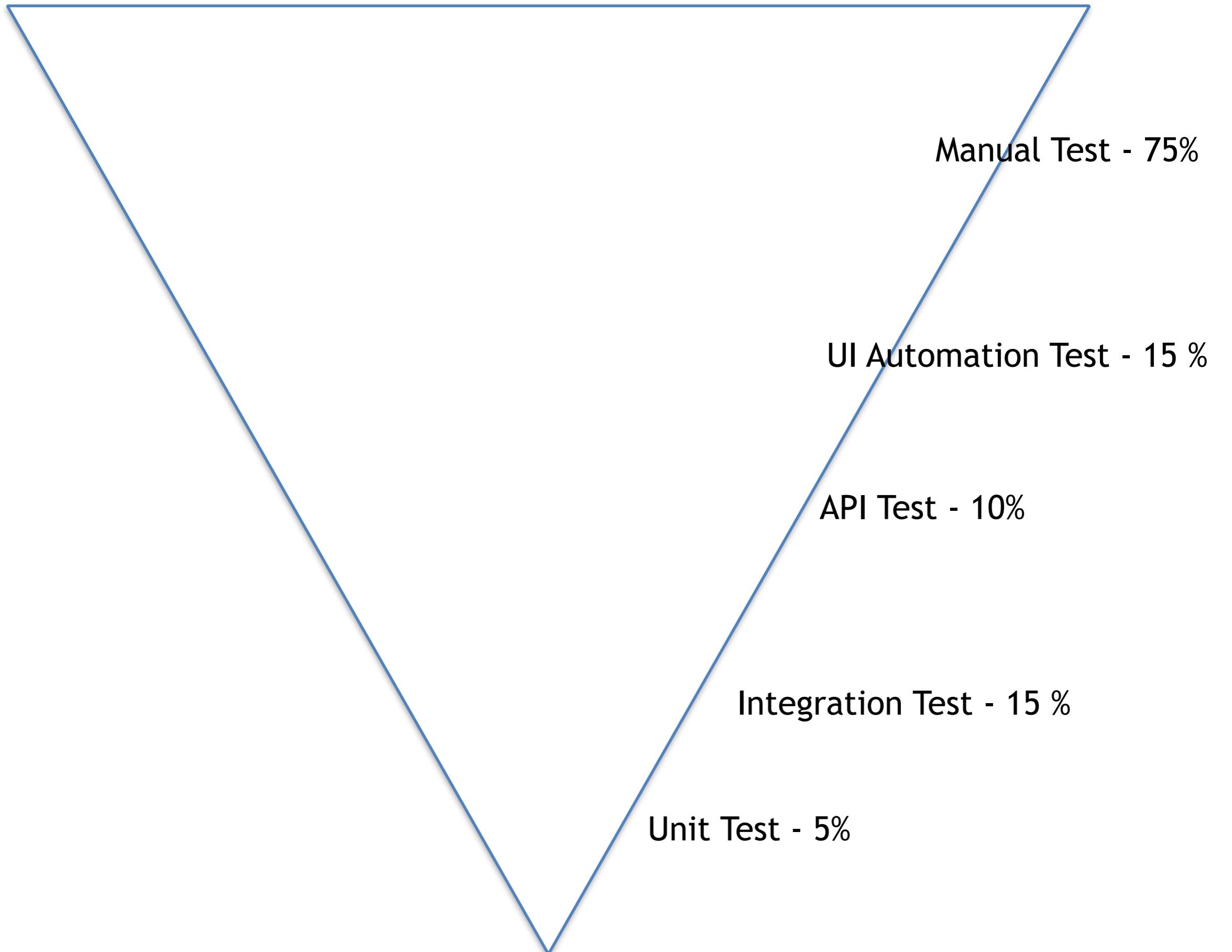
Pyramid test

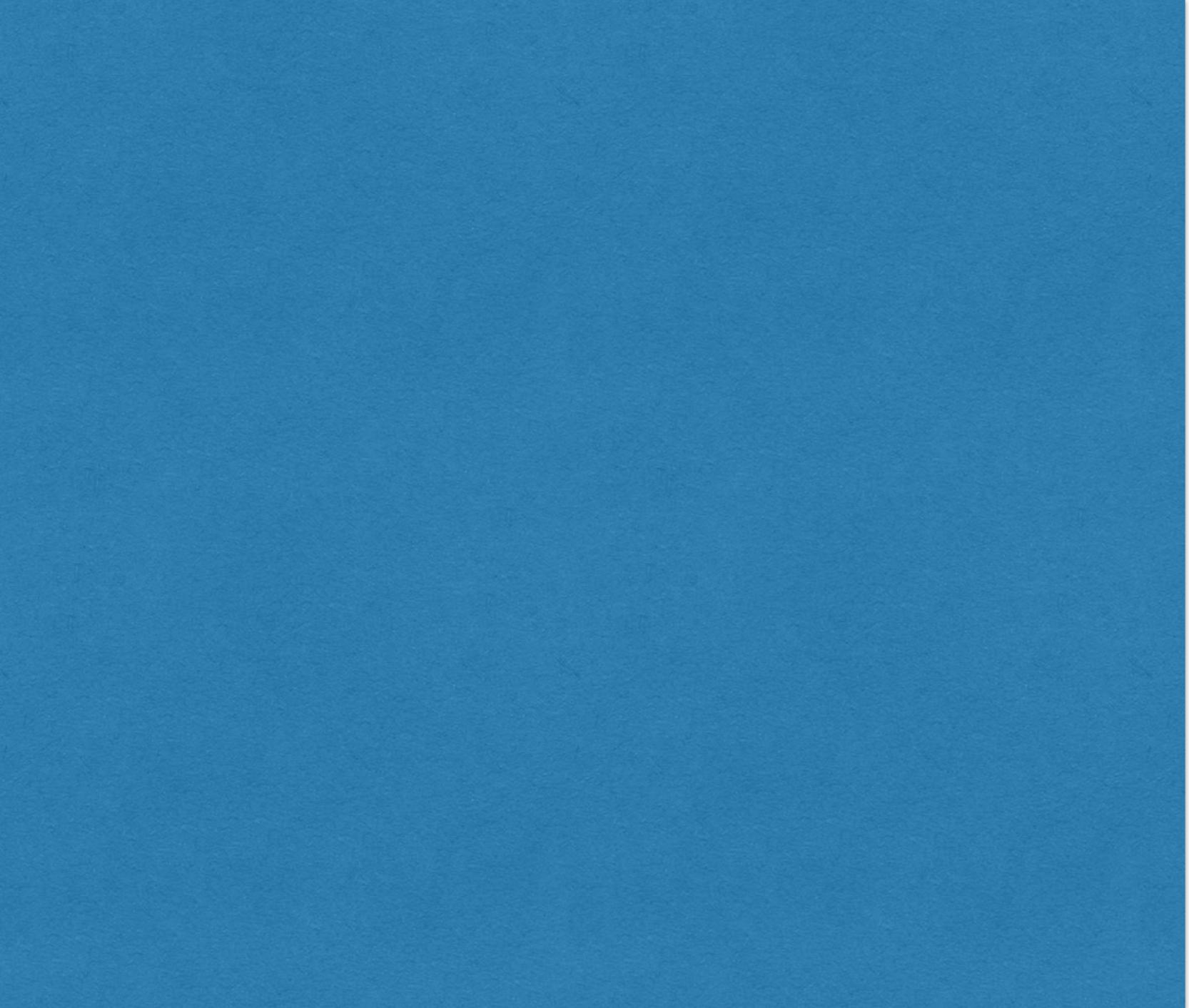


Unit Test

- Documentation for Code
- Design Code
- Regression
- Find Bugs

Its working don't touch it





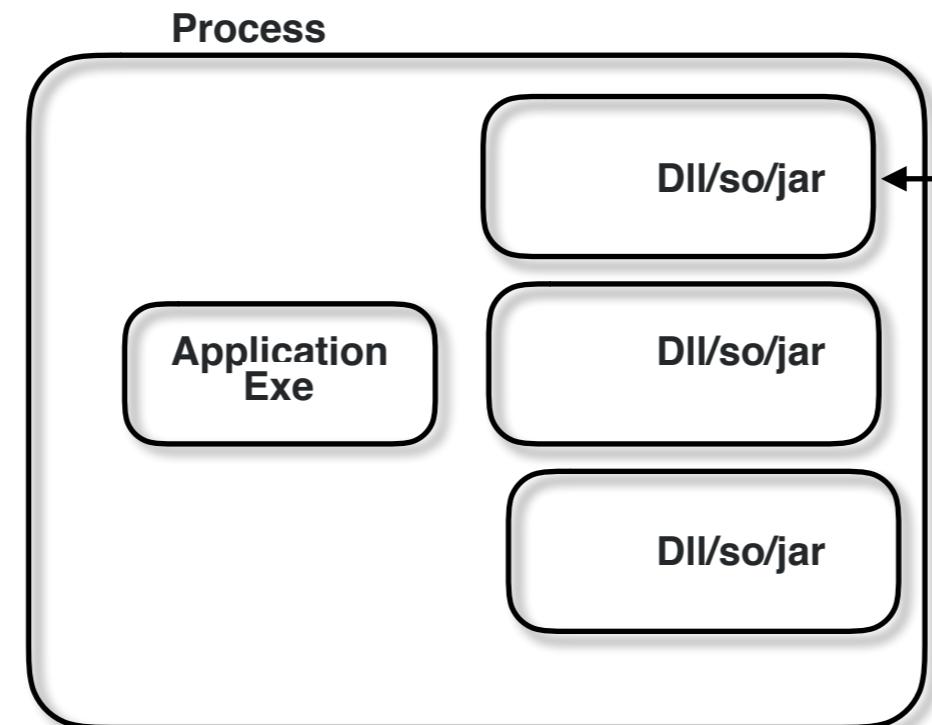
Manual Test - 70%

UI Automation Test - 3 %

API Test - 10%

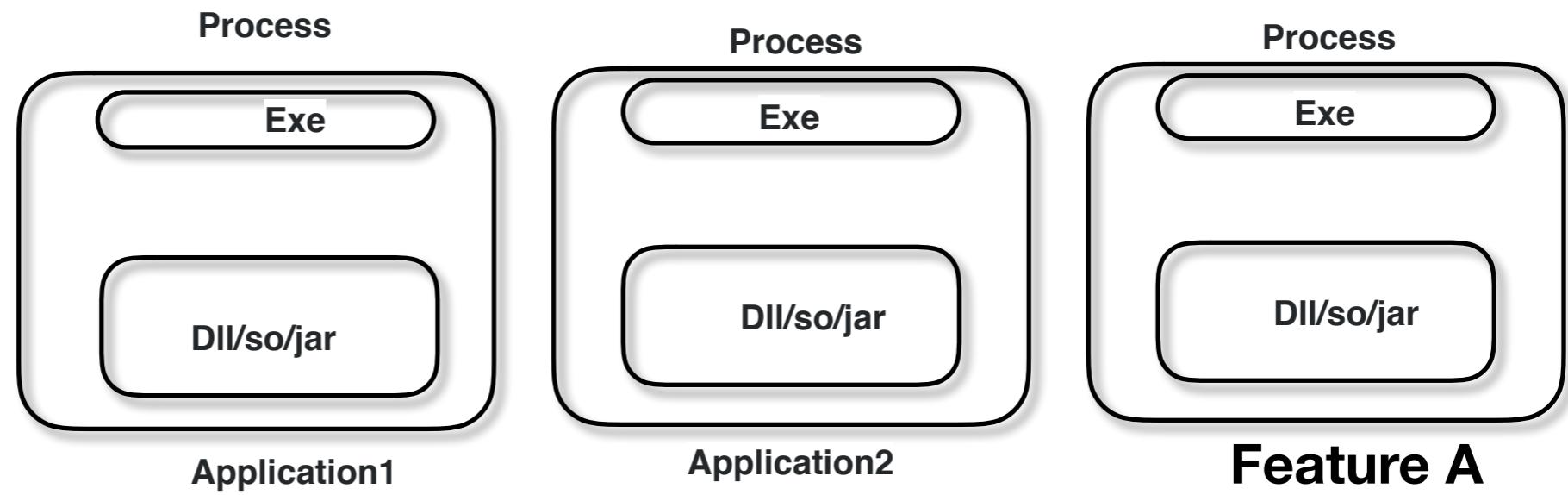
Integration Test - 15 %

Unit Test - 70%



**Runtime
monolithic**

In process
comp

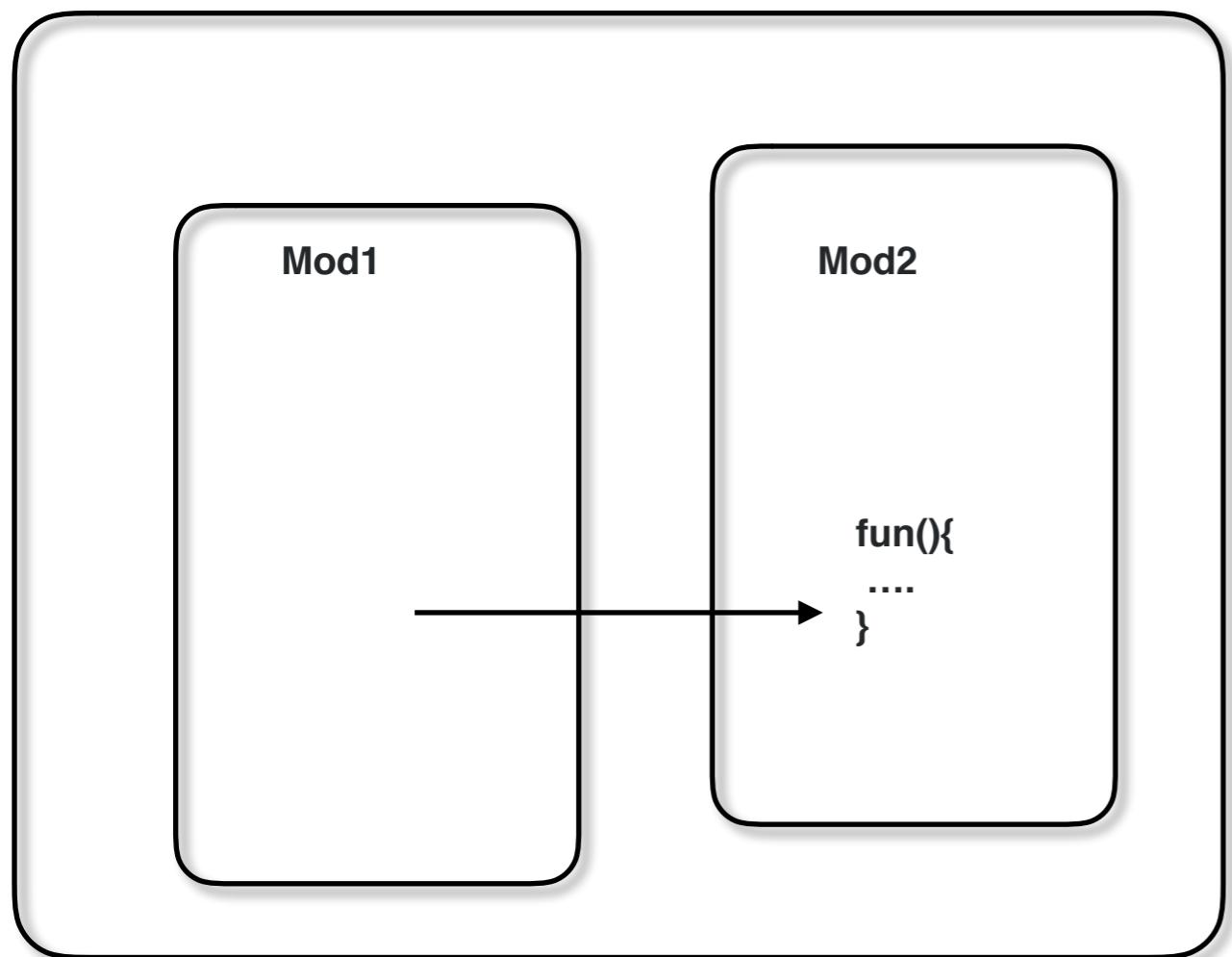


Micro Application

1. Performance

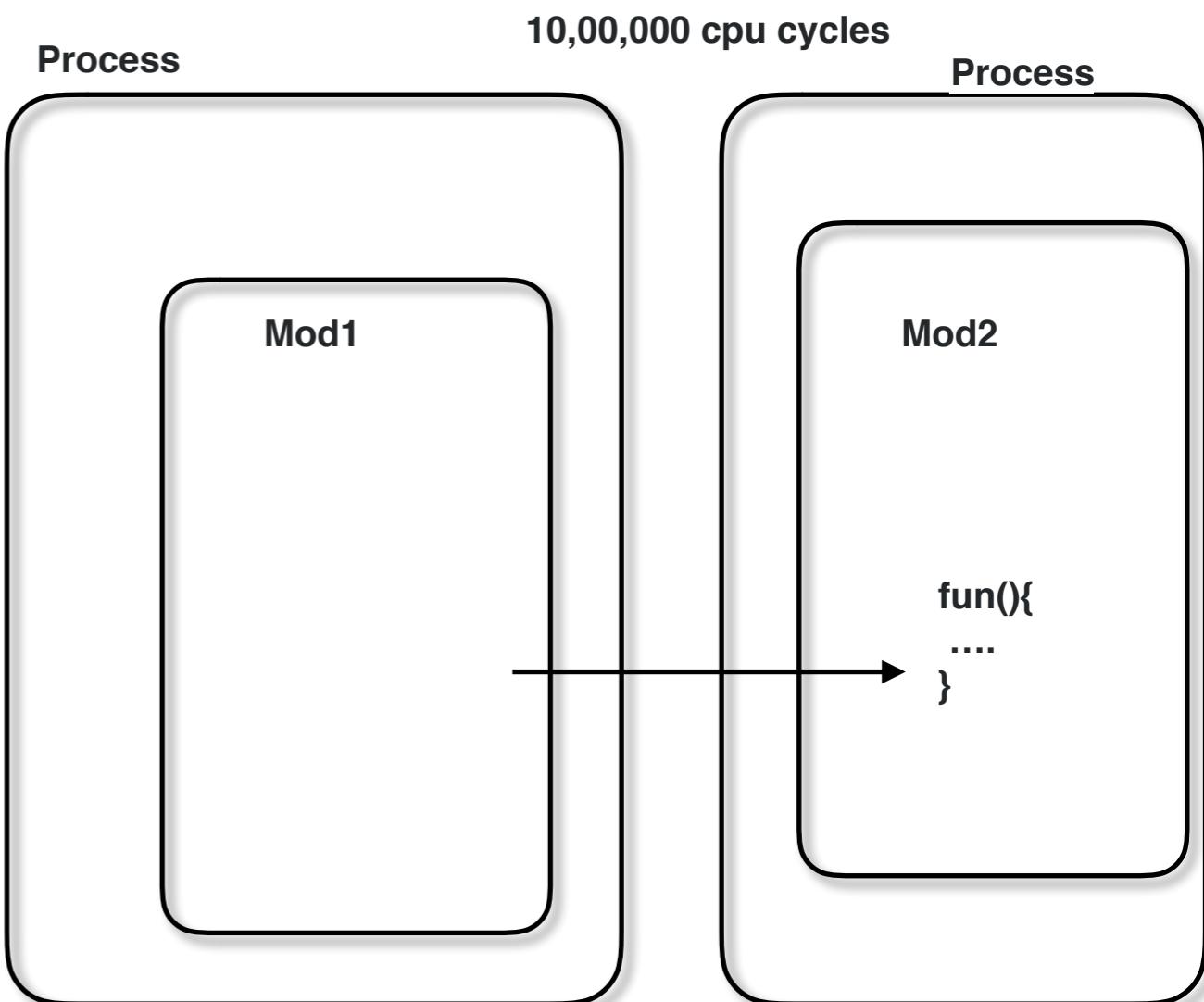
Operation	Cpu Cycles
• 10 + 12	3
• Calling a in-memory Method	~10
• Create Thread	2,00,000
• Destroy Thread	1,00,000
• Database Call	40,00,000
• Distributed Fun Call	20,00,000
Write to disk	10,00,000

Process



10 cpu cycles

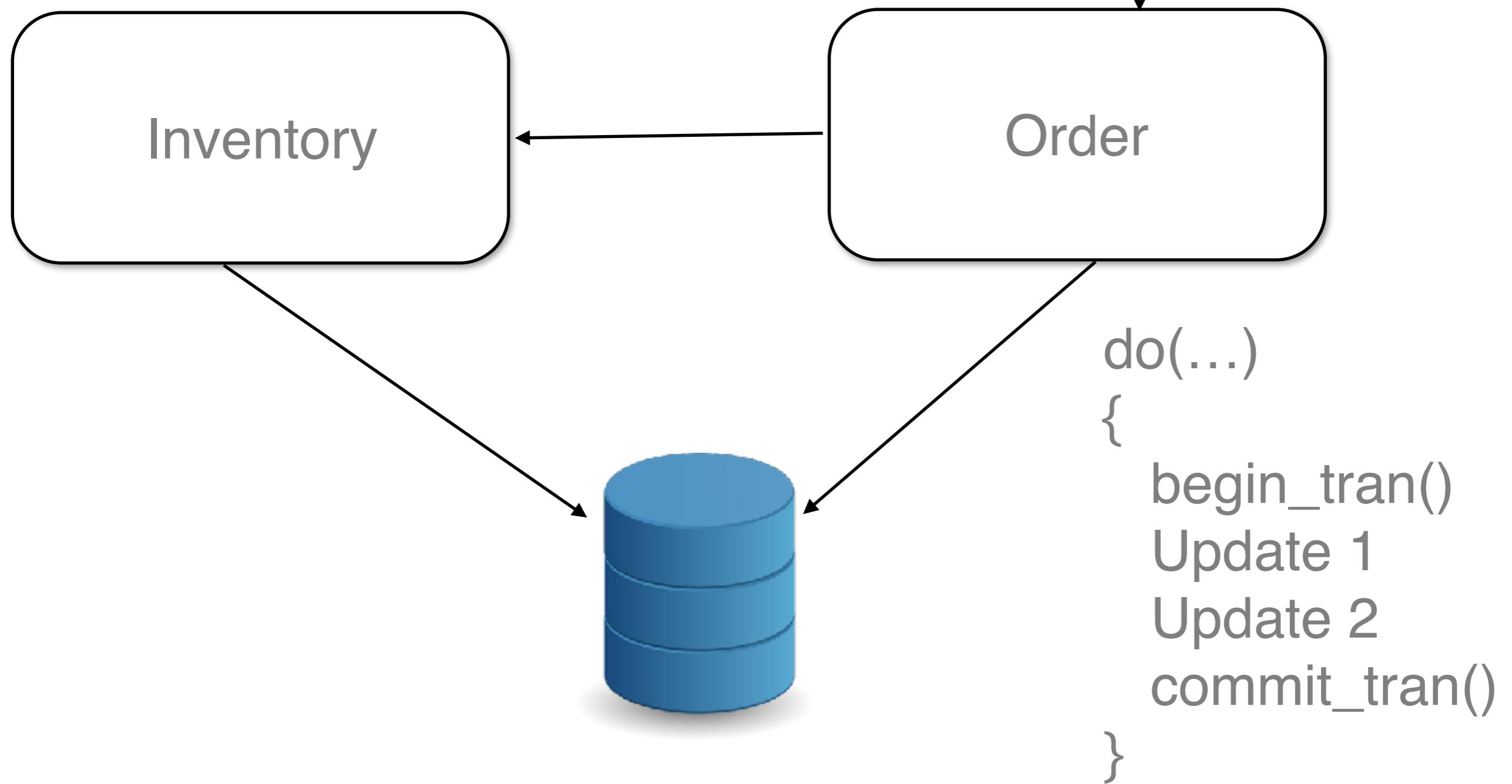
Process



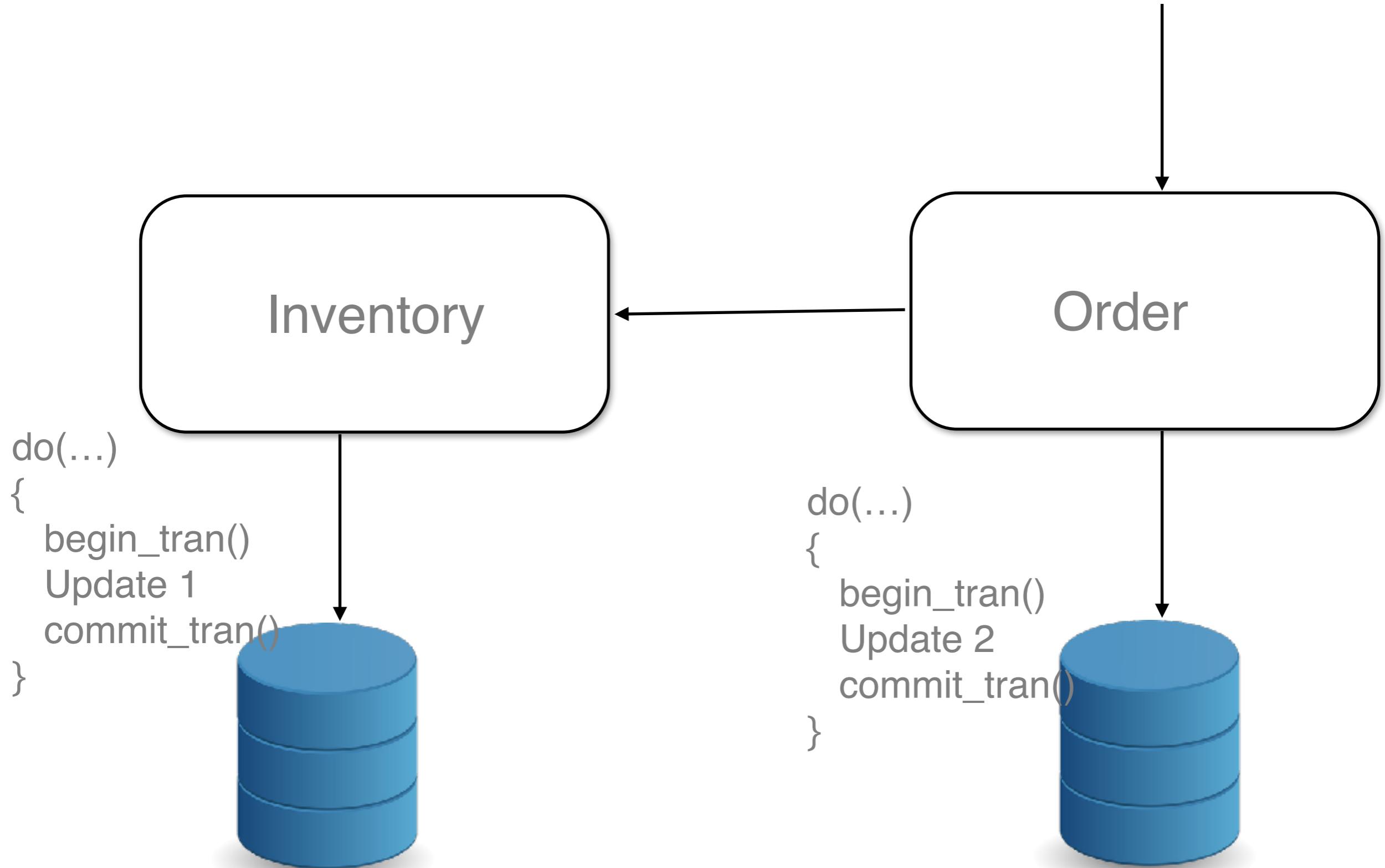
10,00,000 cpu cycles

Process

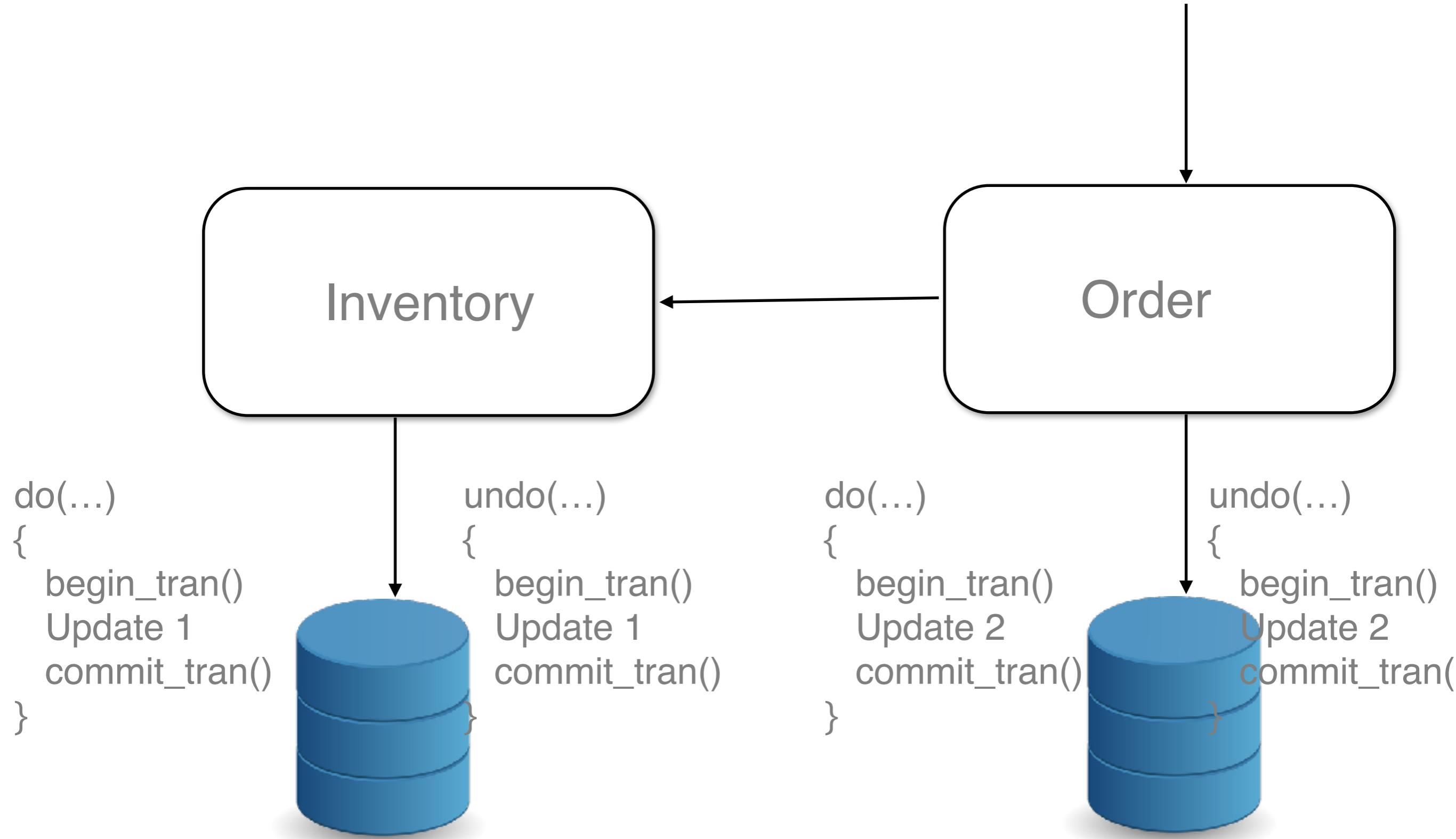
2. Db transaction

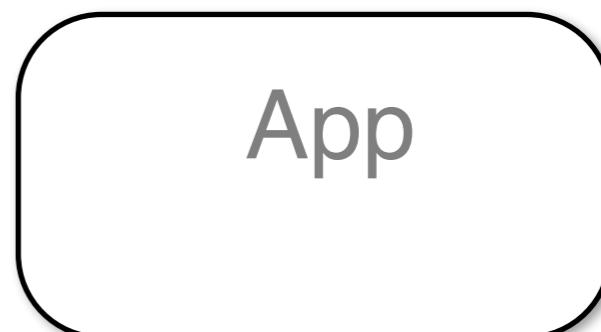
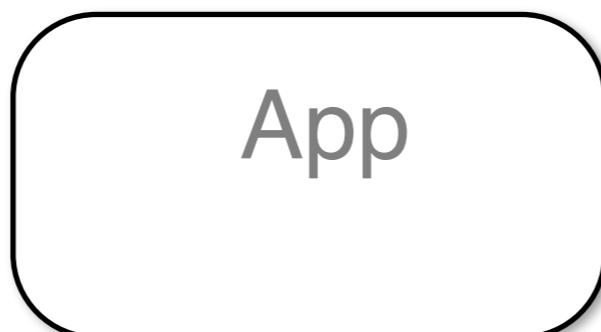
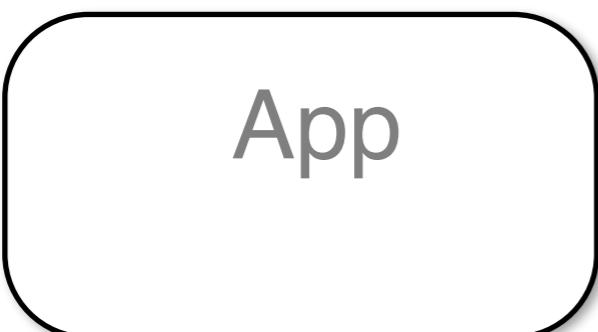
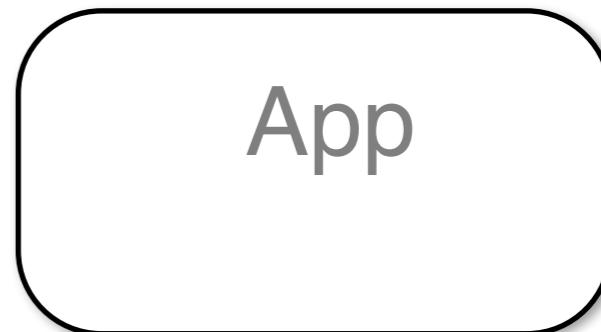
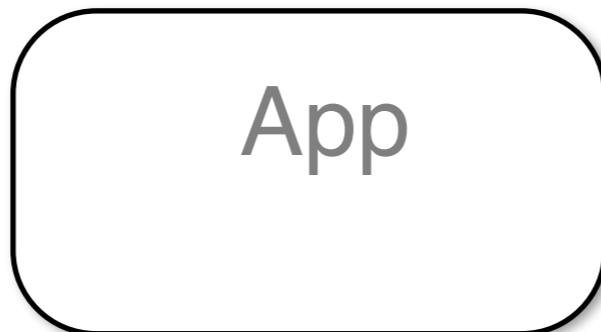
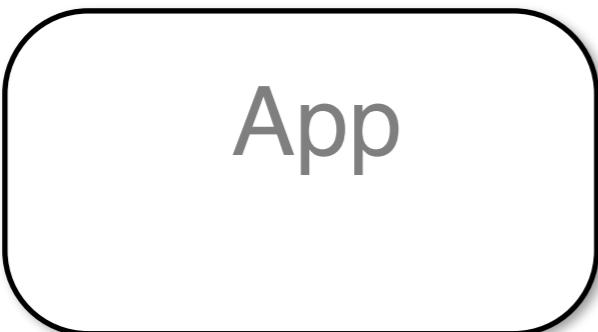


2. Db transaction



2. Db transaction





App

Mod

Mod

Mod



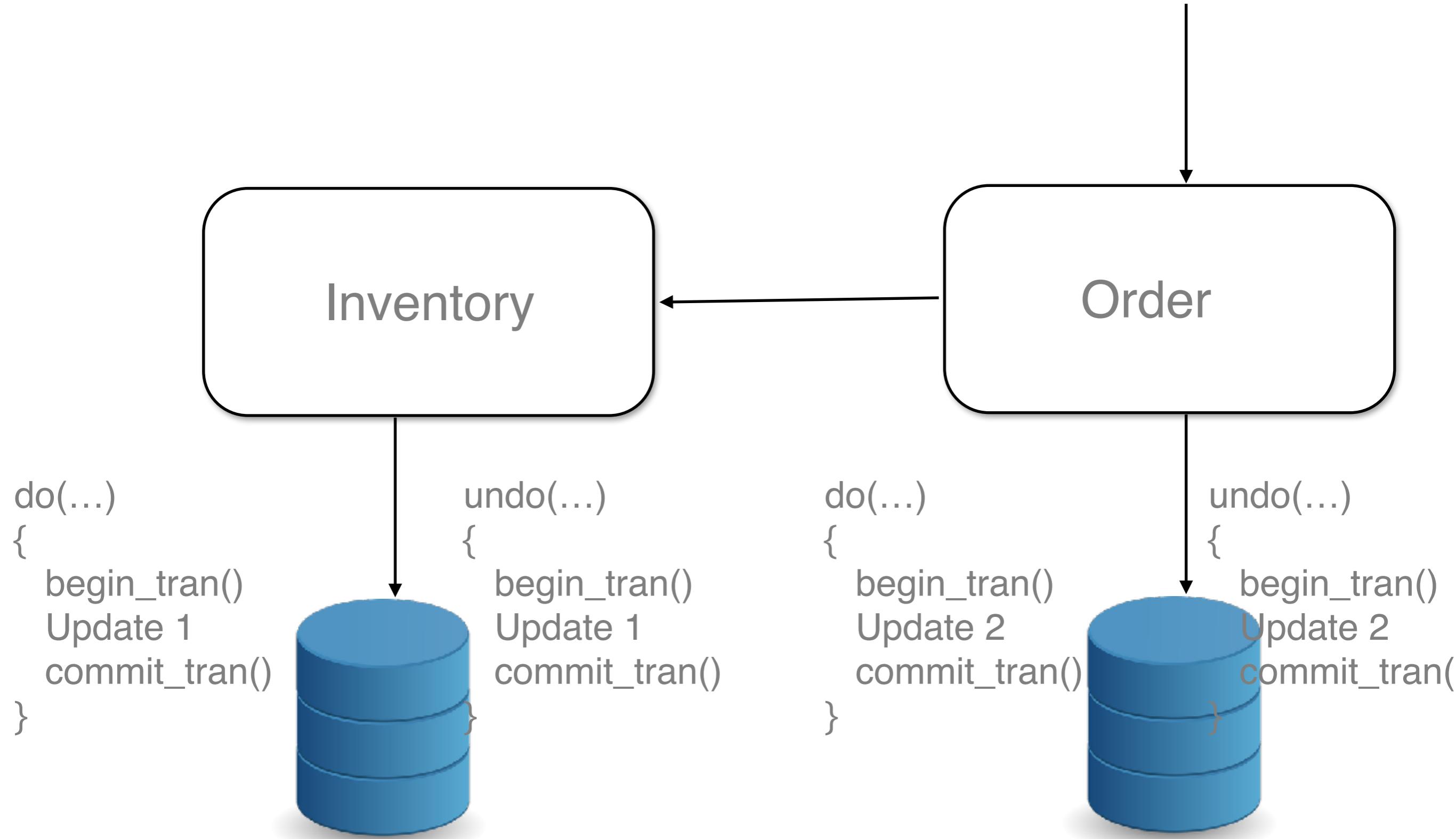
App

App

App

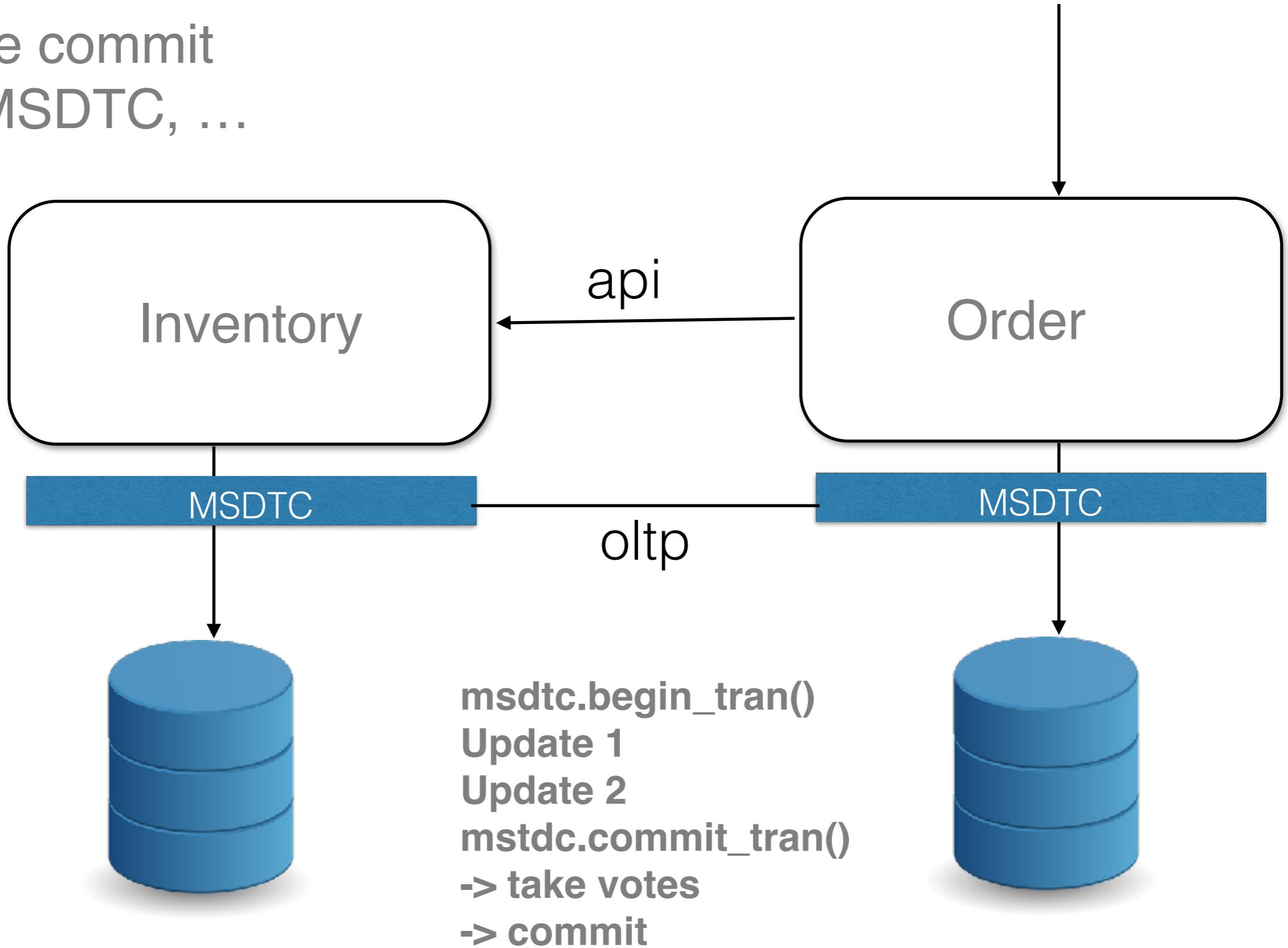


2. Db transaction

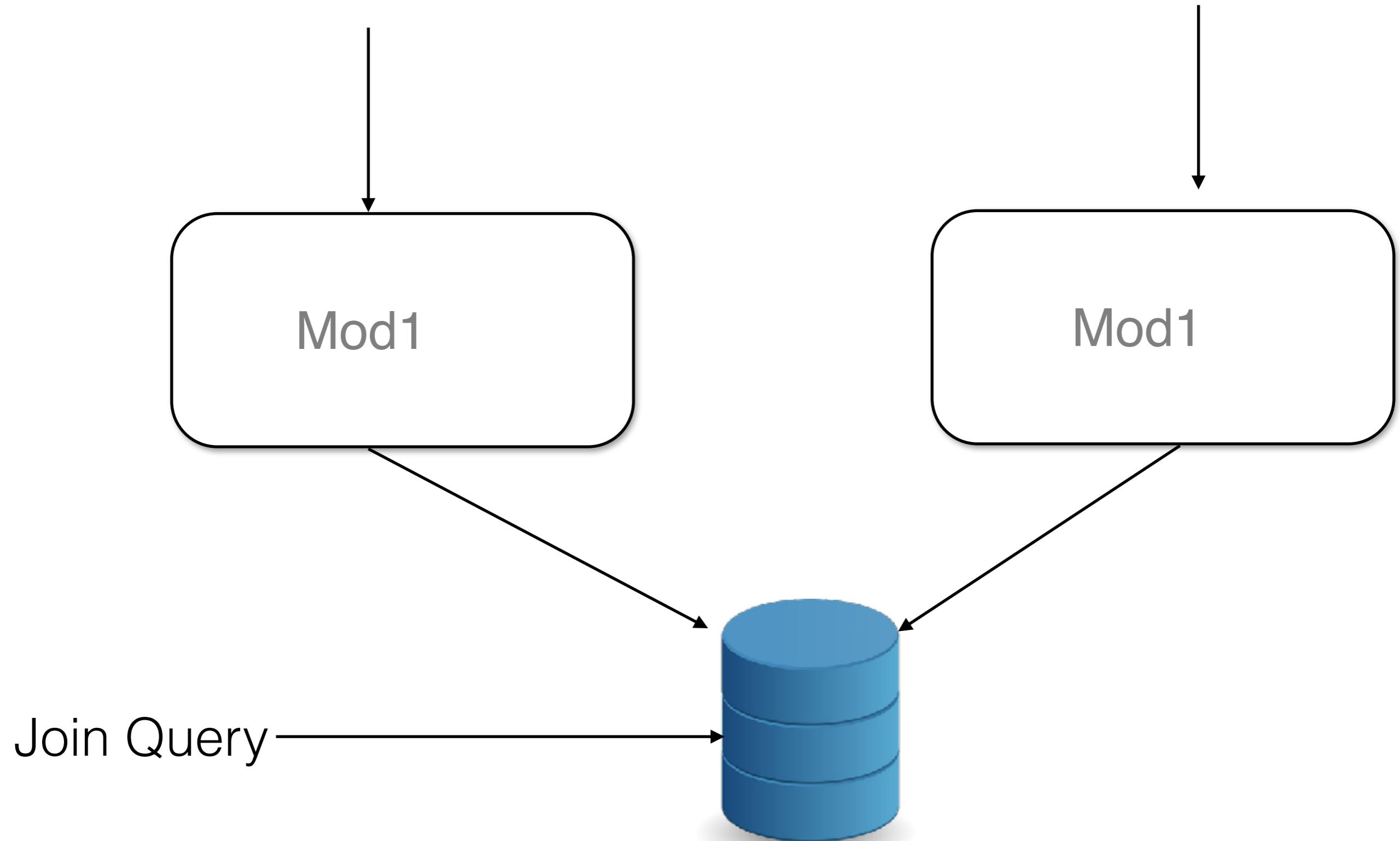


2 phase commit

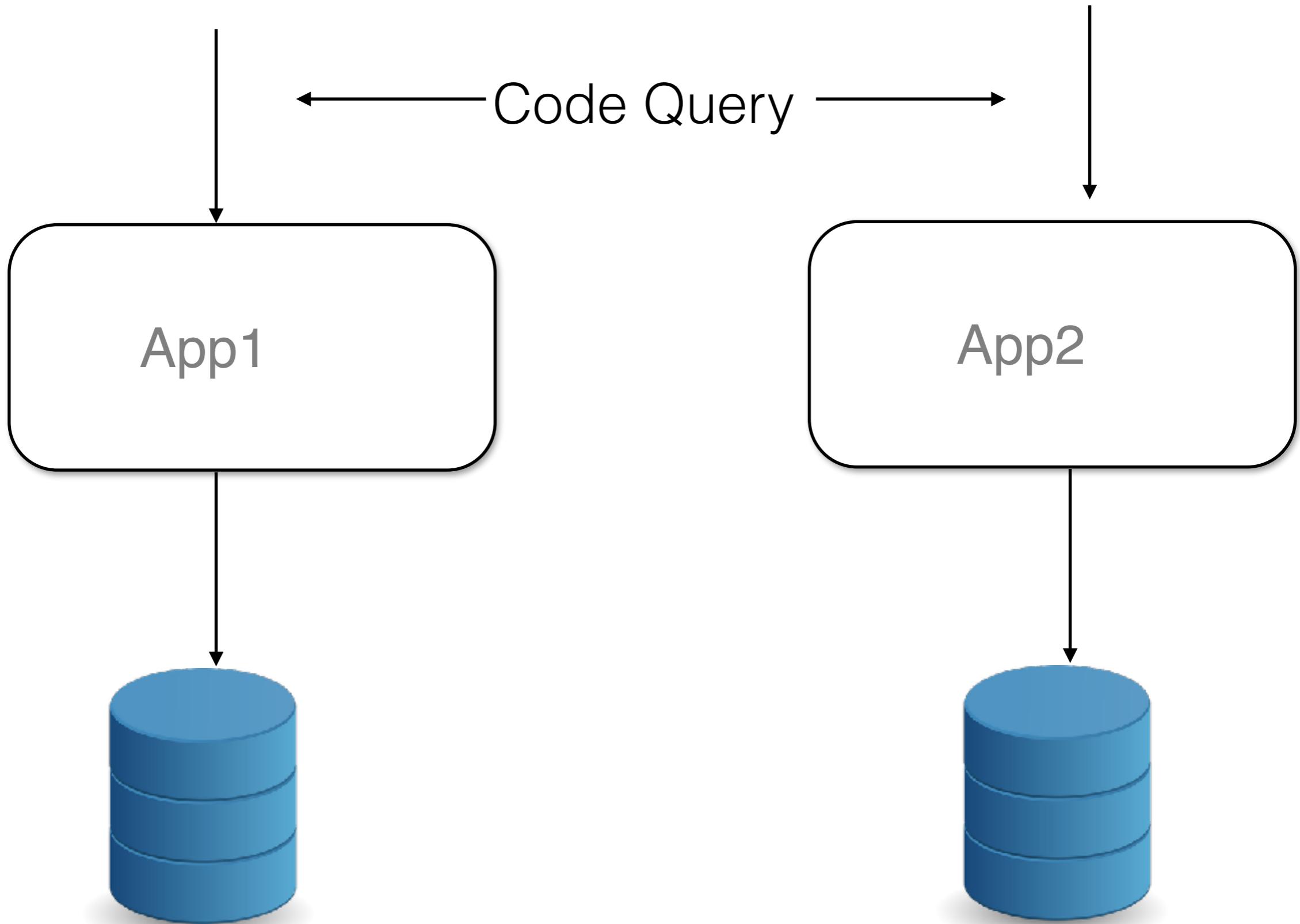
2 phase commit
JTX , MSDTC, ...



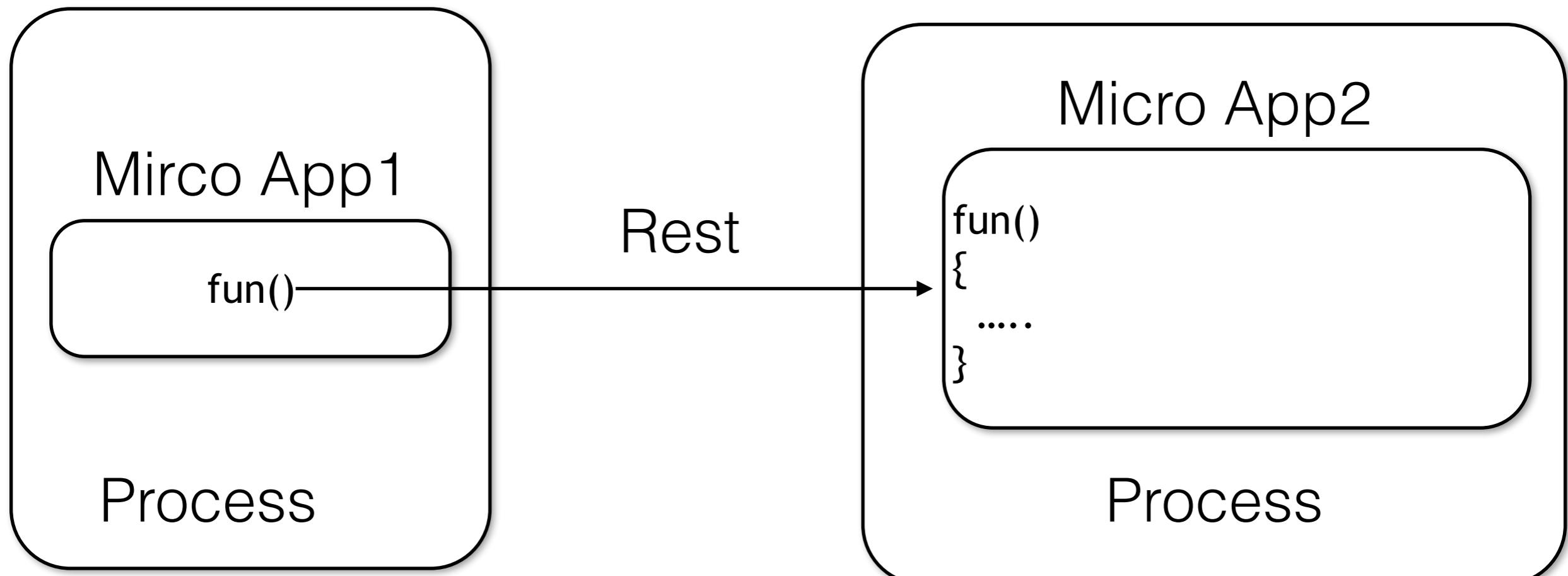
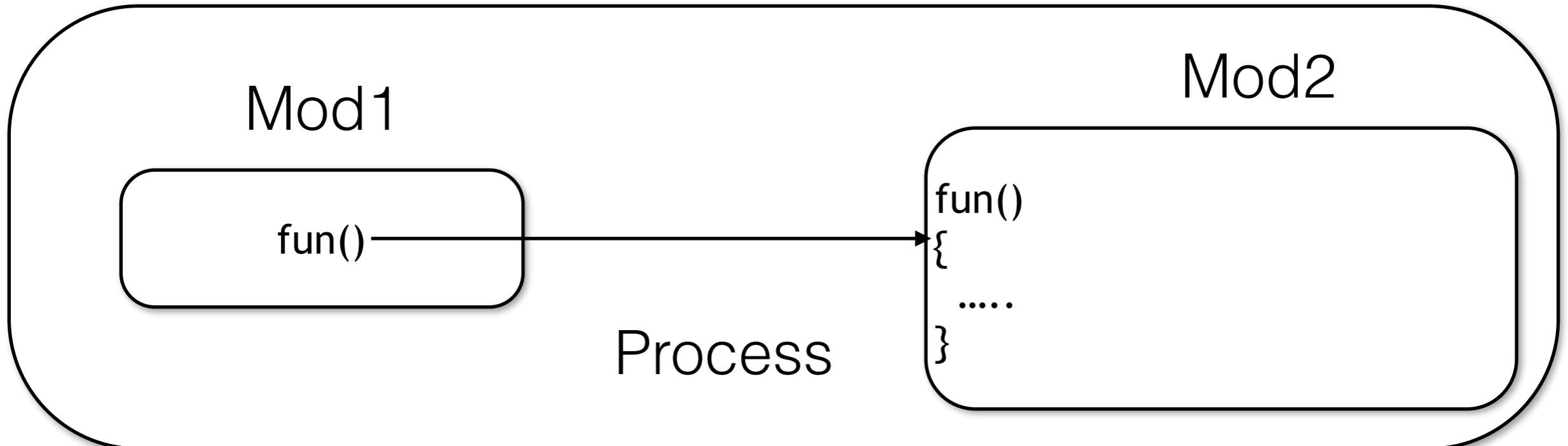
3. Db query



3. Query



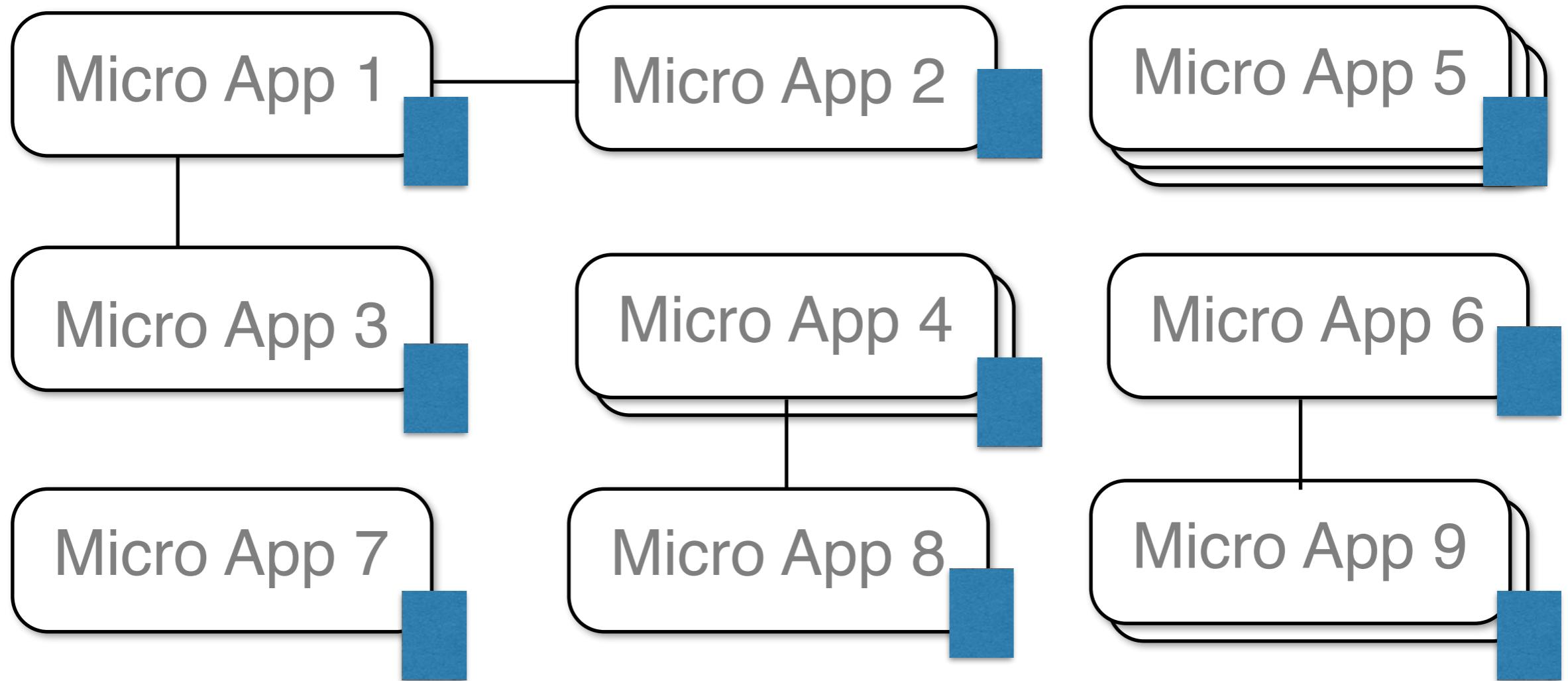
4. Development time



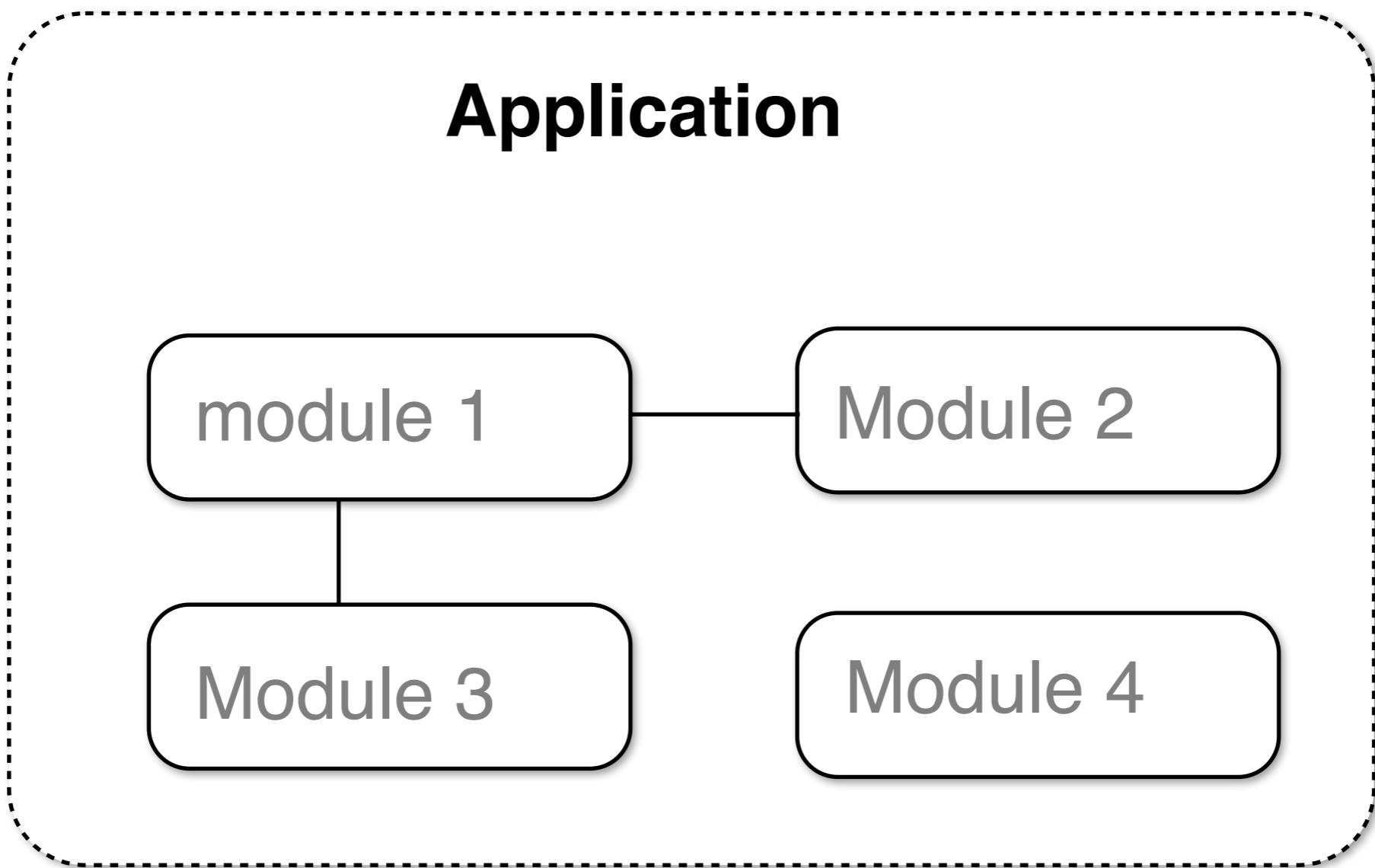
5. Learning Curve



6. Infra Cost



7. Debugging



Dev & Ops Teams



Log Data

Web Logs
App Logs
Database Logs
Container Logs

Metrics Data

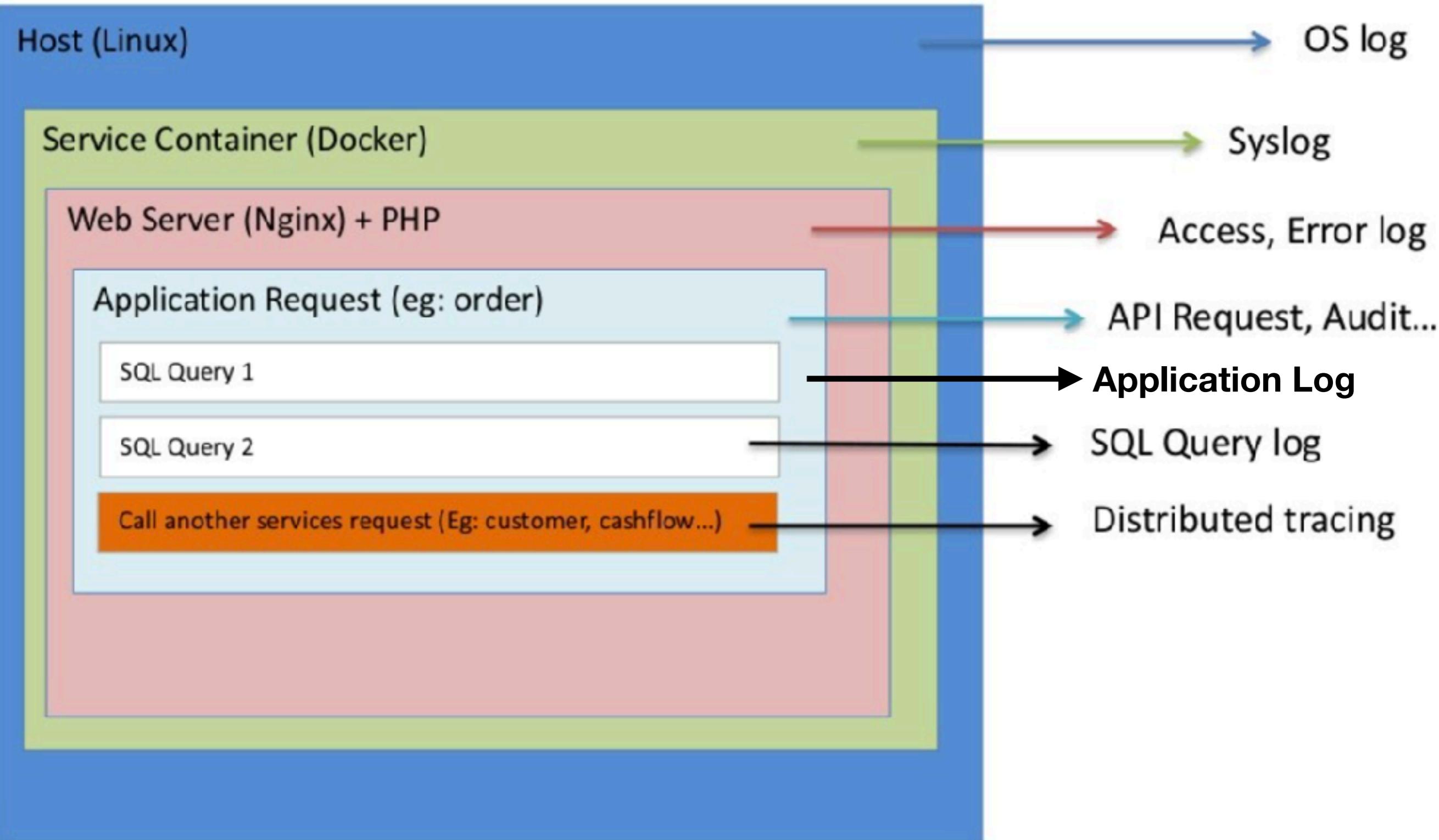
Container Metrics
Host Metrics
Database Metrics
Network Metrics
Storage Metrics

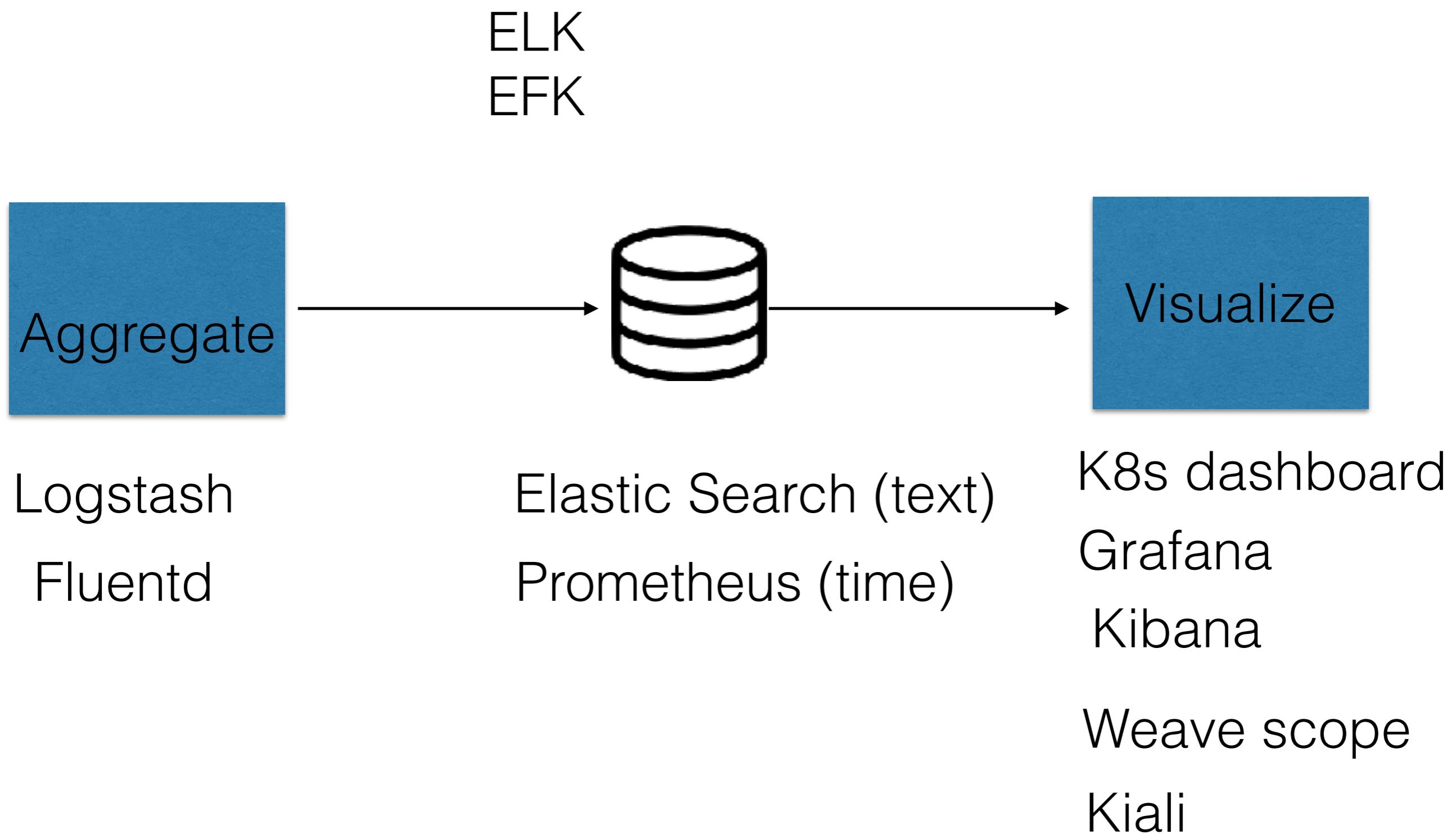
APM Data

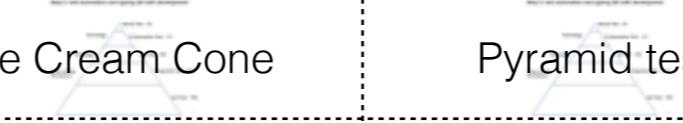
Real User Monitoring
Txn Perf Monitoring
Distributed Tracing

Uptime Data

Uptime
Response Time

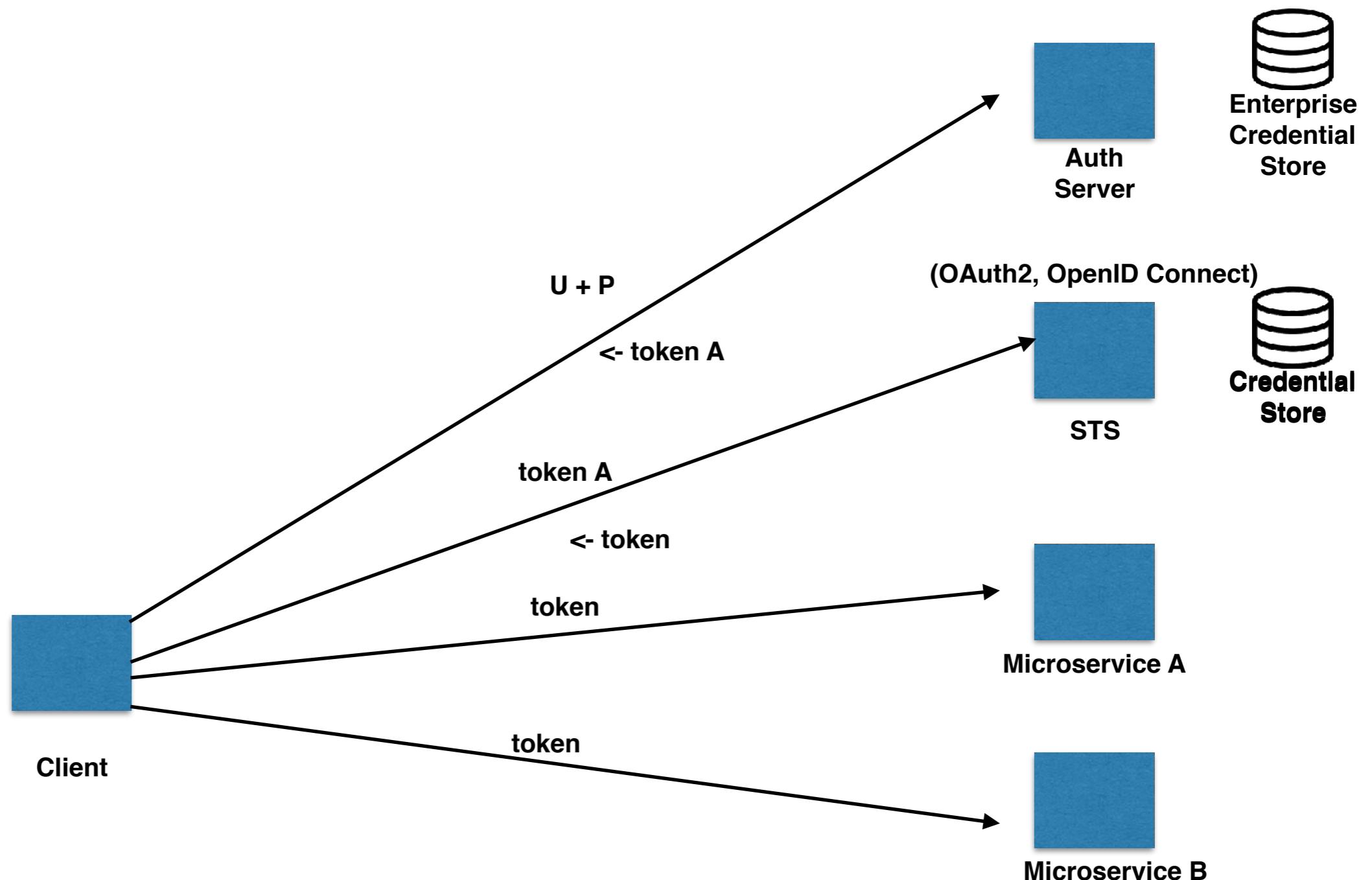




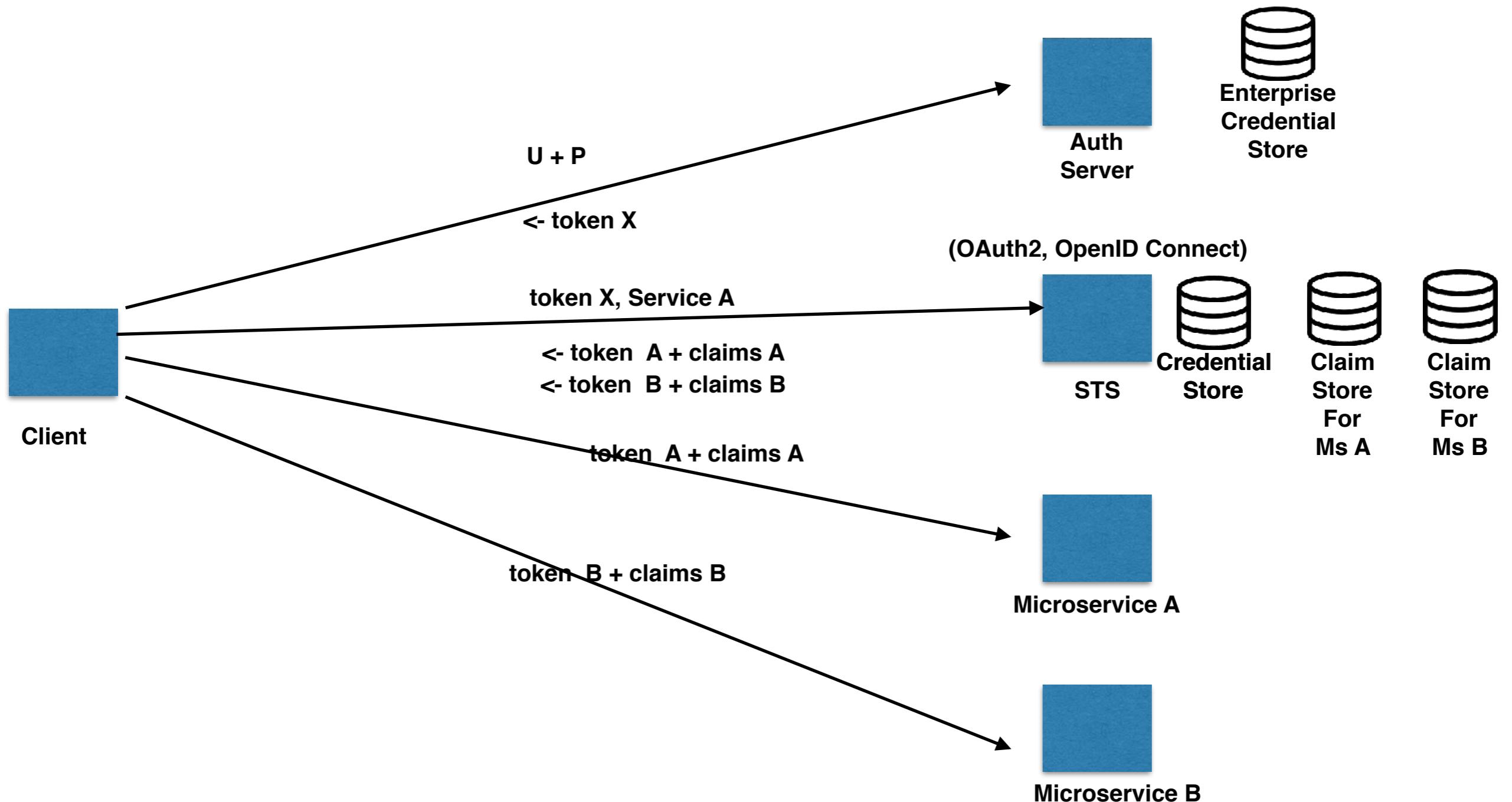
	Bad	Recommended	Challenges	Alternate
Resource Performance (CPU, Memory, I/O)	Chatty network calls	Grpc, Protobuf, ...		Merge
Db Transaction Management (multiple db)	2 phase commit	Compensatable tran, SAGA	Design complexity	Shared db
Views / Report / Dash board/ join (multiple db)	Application code will join the data	Materialized View	Duplicate data	Shared db
Infra Cost	VM per service	Docker Containers	Learning	Pod man
Preparing for Deployment	Manual deployment	Infra as code	Learning	
Debugging, Error Handling (End to End)	Manual	Jaeger Distributed tracing, Kiali	Learning/ config	Elastic APM
Integration Test	Ice Cream Cone	Pyramid test	Learning	
Log Mgmt (debug/ error)	Not centralising logs	EFK	Additional DEvOPs effort	ELK
Config Mgmt	Not centralized	Redis, Consul, ...	Learning	Key-value with change
Authentication (who)	Not centralized	STS, OAuth2, OpenIDConnect	Additional DEvOPs effort	
Authorization (what can they do)	Not centralized	STS claim Mapping	Additional DEvOPs effort, config	
Audit Log mgmt (who accessed what)	Not centralized	Event Sourcing	Learning Curve, Complexity	
Monitoring / Alerting	No centralised monitoring tools	Weave scope, Grafana		K8s dashboard
Data Security and Privacy (transit, storage)		HTTPS, Encryption		Hash
Build Pipeline (CI)	Manual	Infra as code	Learning	

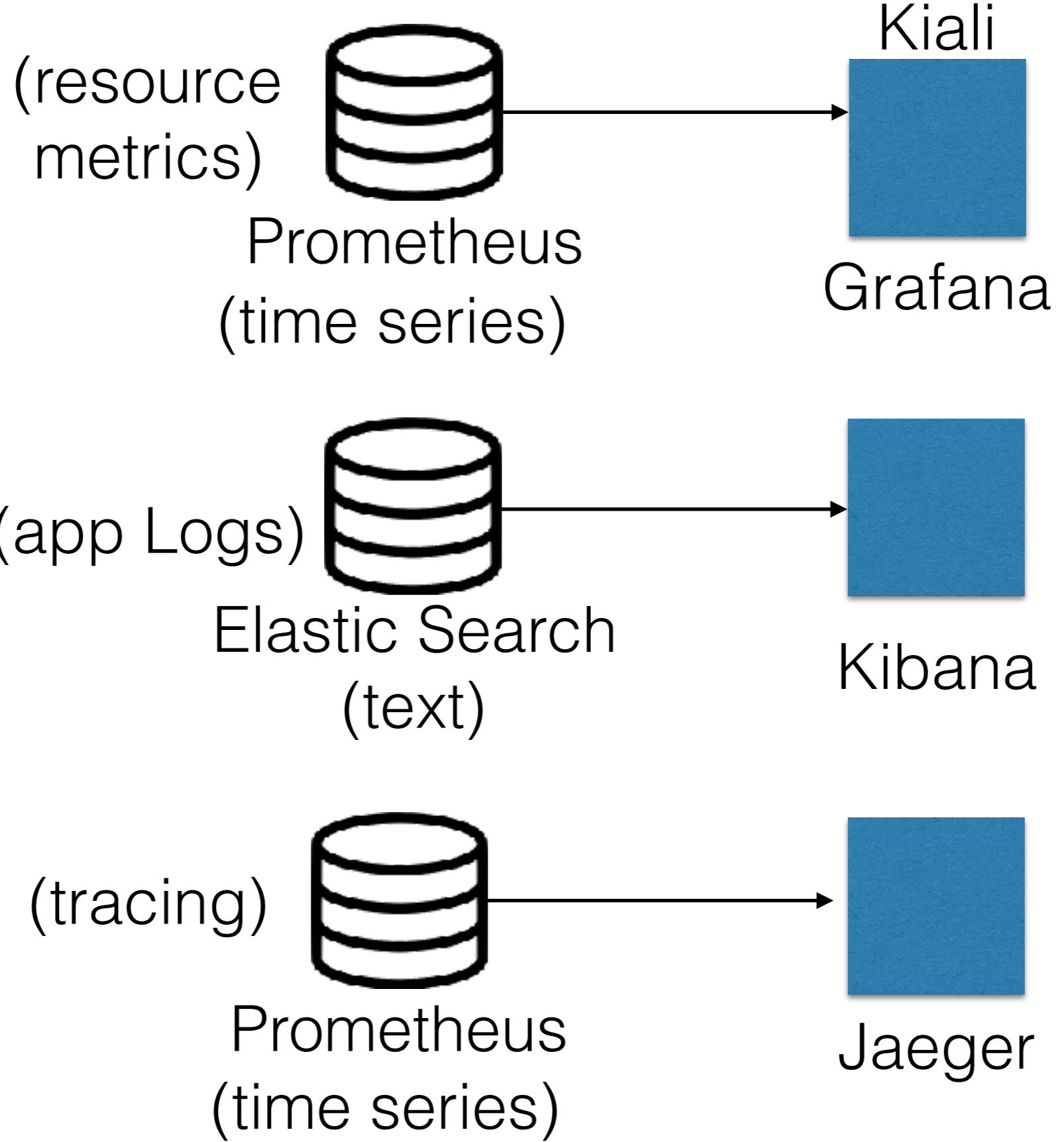
Security

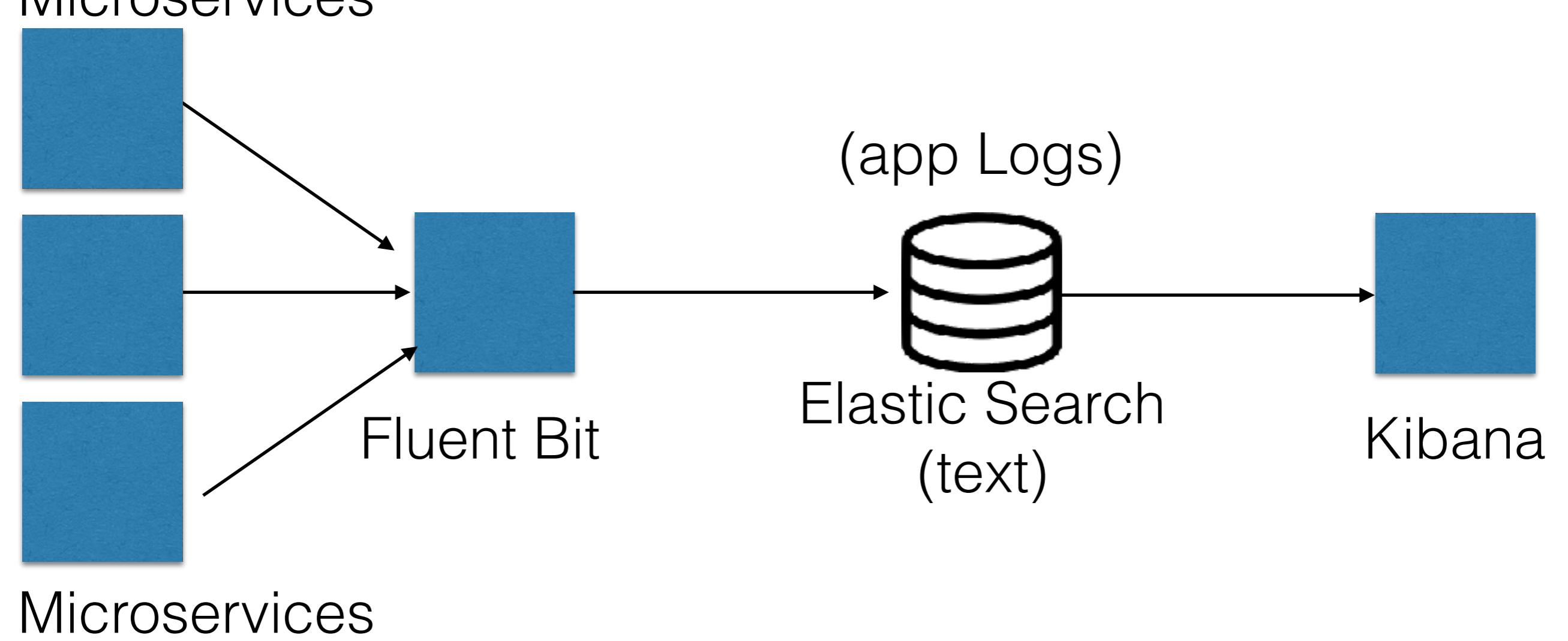
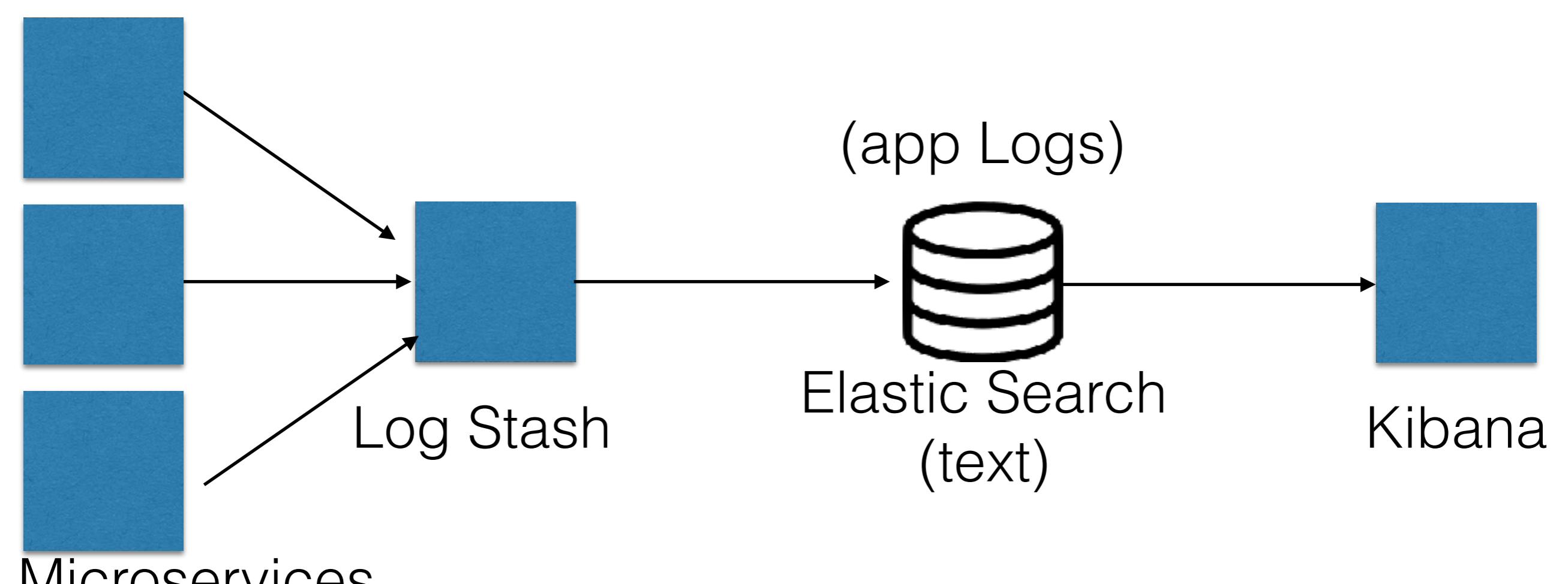
- Authentication (who are you)
- Authorization (what can you do)
- Auditing (what did you do)
- Input validation (fwk)
- Exception handling (fwk)
- Session Management (fwk)
- Data Security
 - at Rest
 - in Transit
- Key management (Vault)



- By what you know (pwd, secret)
- By What you Have (otp, email, smart card, cert, rsa)
- By What you are (face, voice, finger print,....)
- By Where you are (location, time, ...)
- By Behaviour



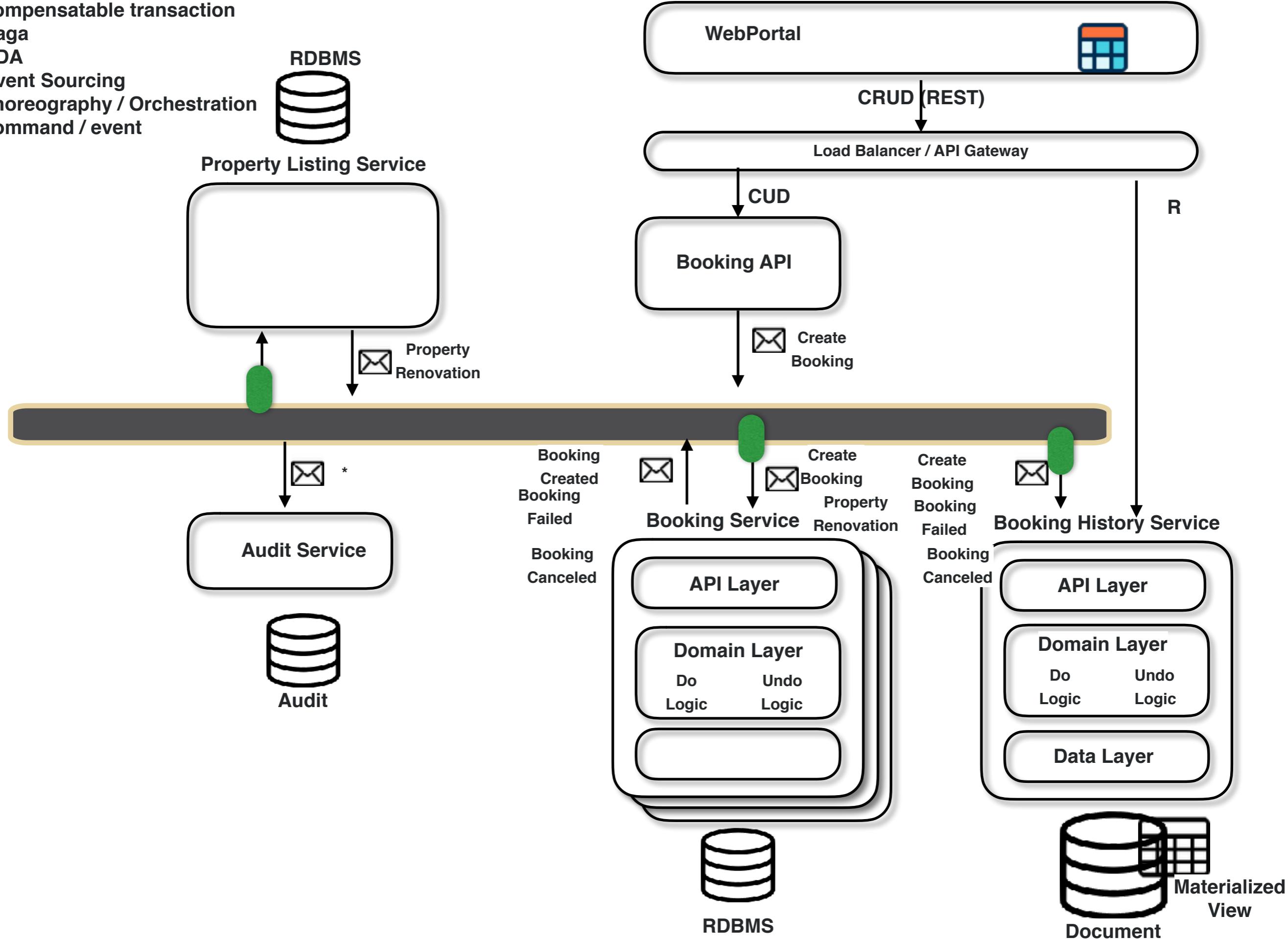


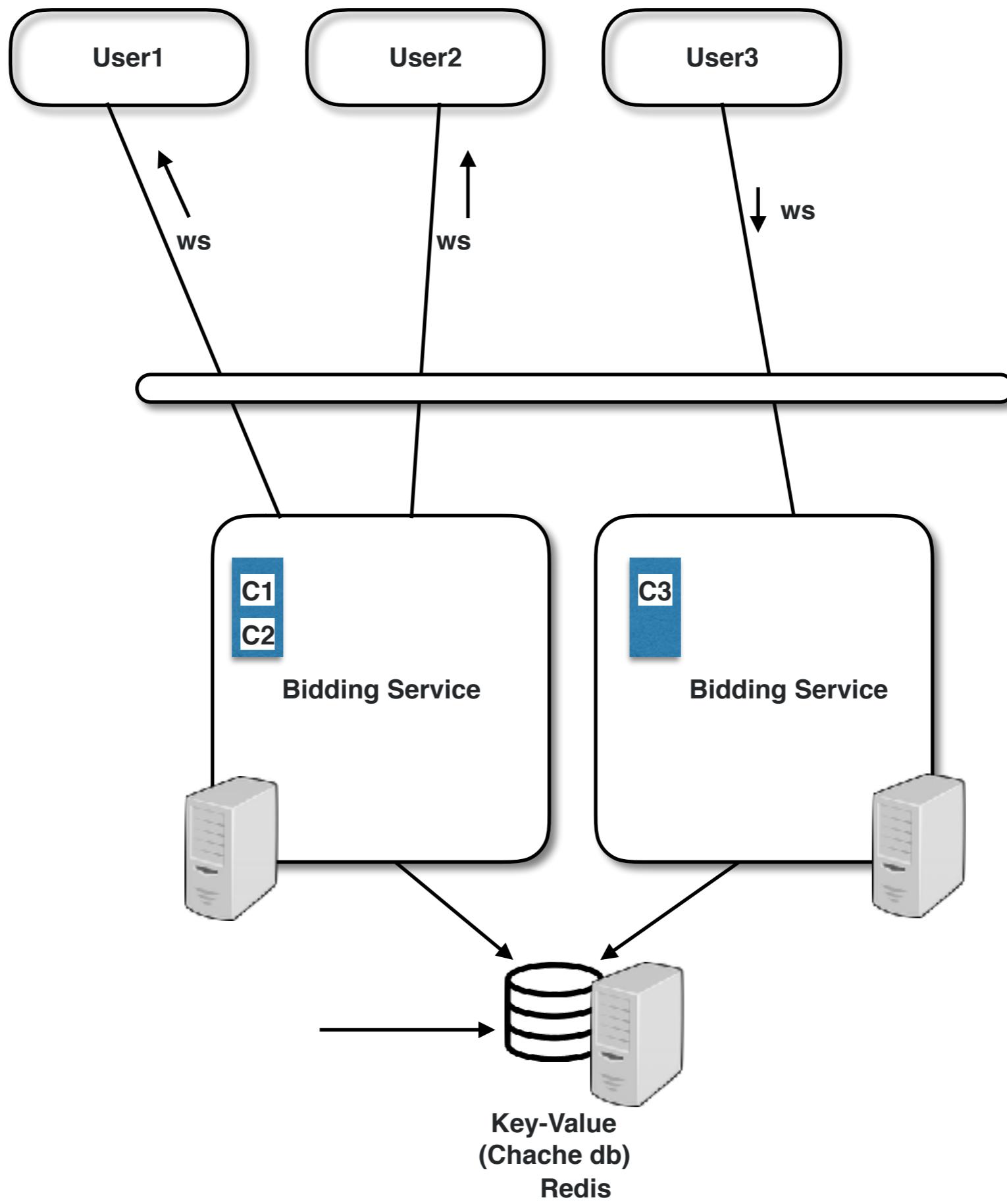


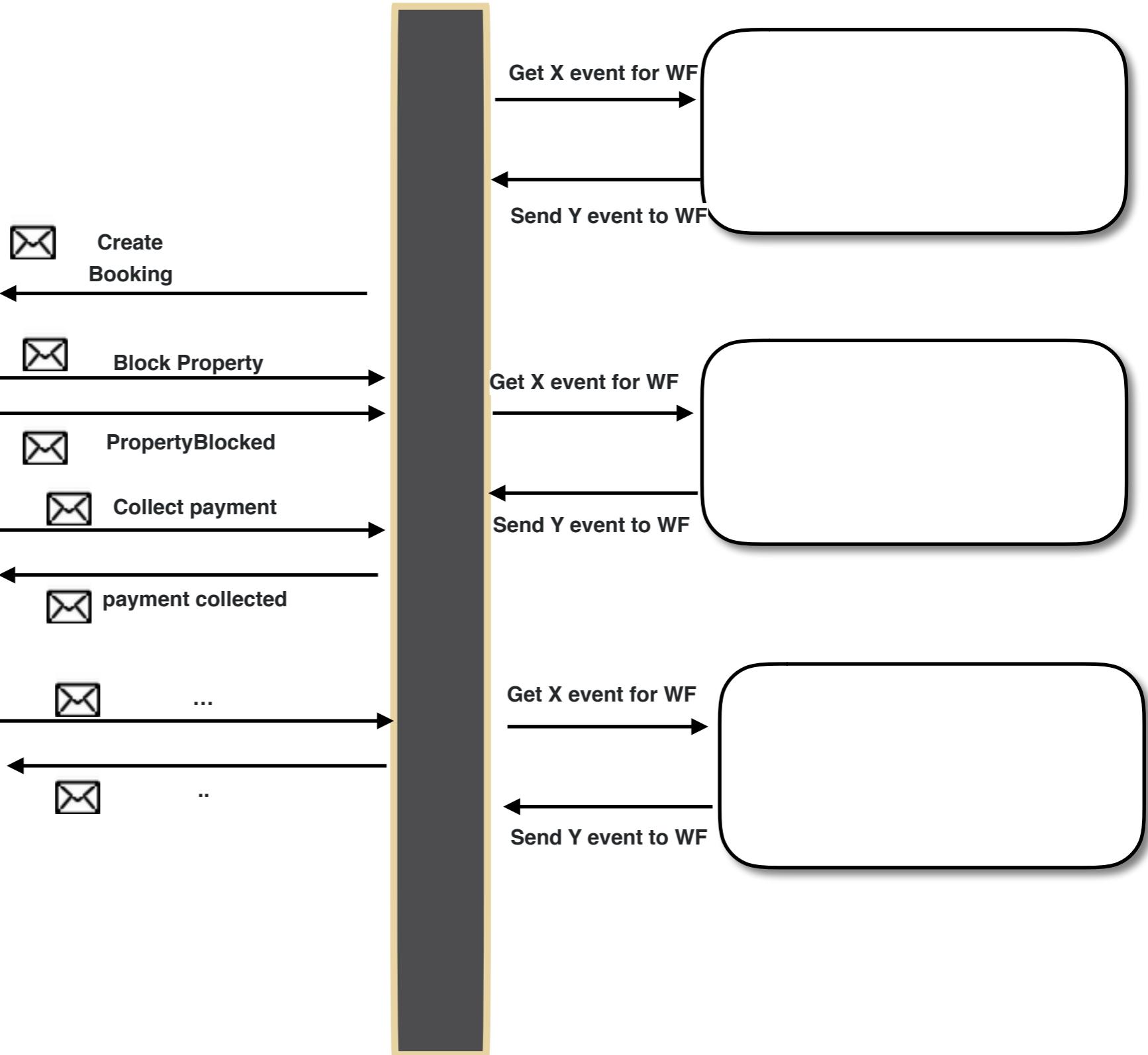
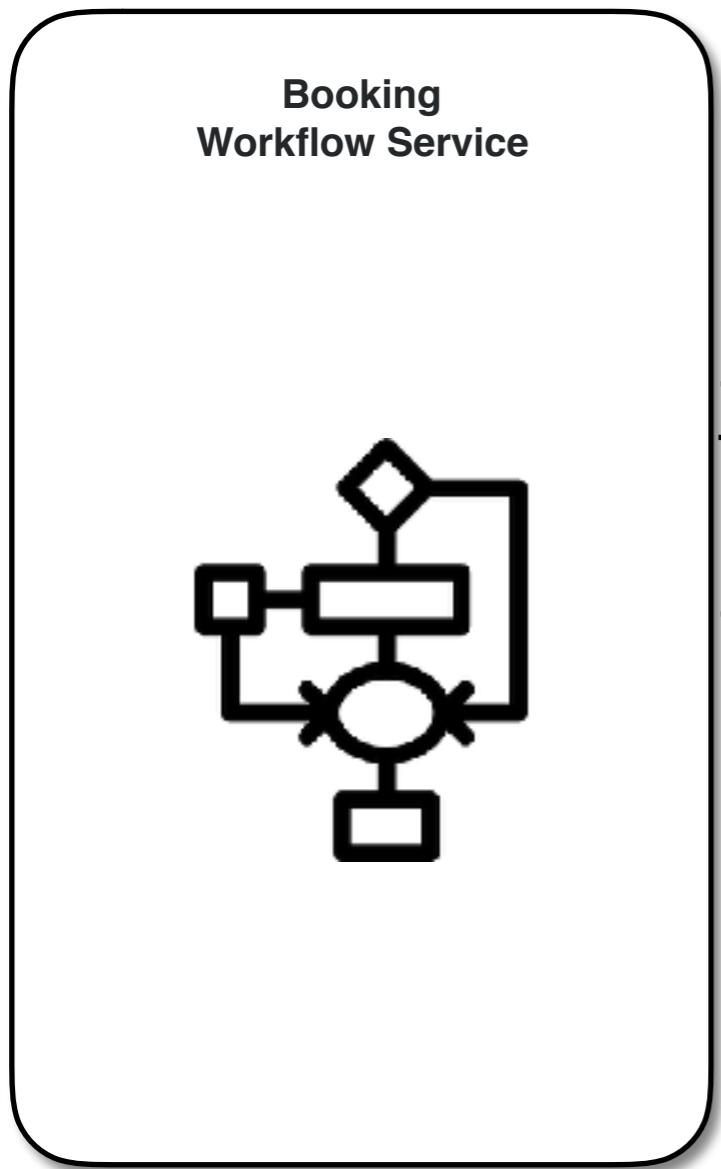
```

# Materialized View
# compensatable transaction
# Saga
# EDA
# Event Sourcing
# choreography / Orchestration
# command / event

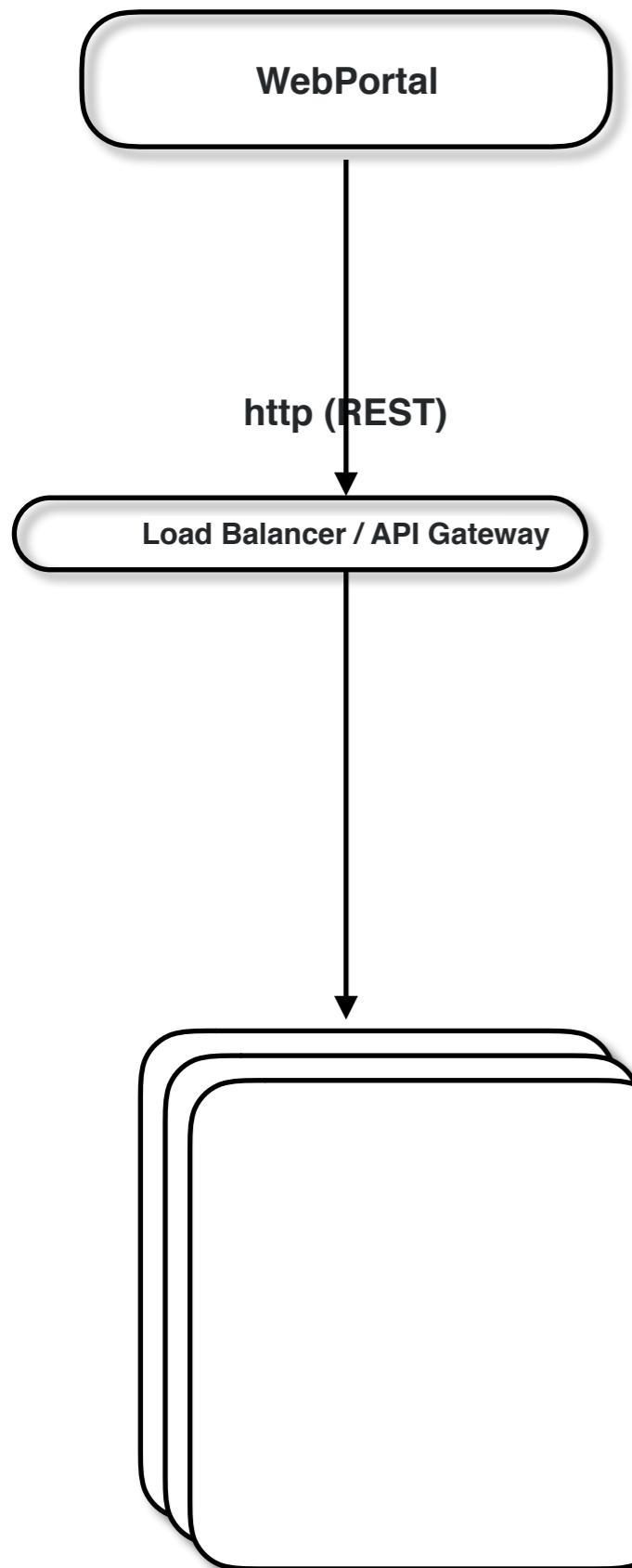
```



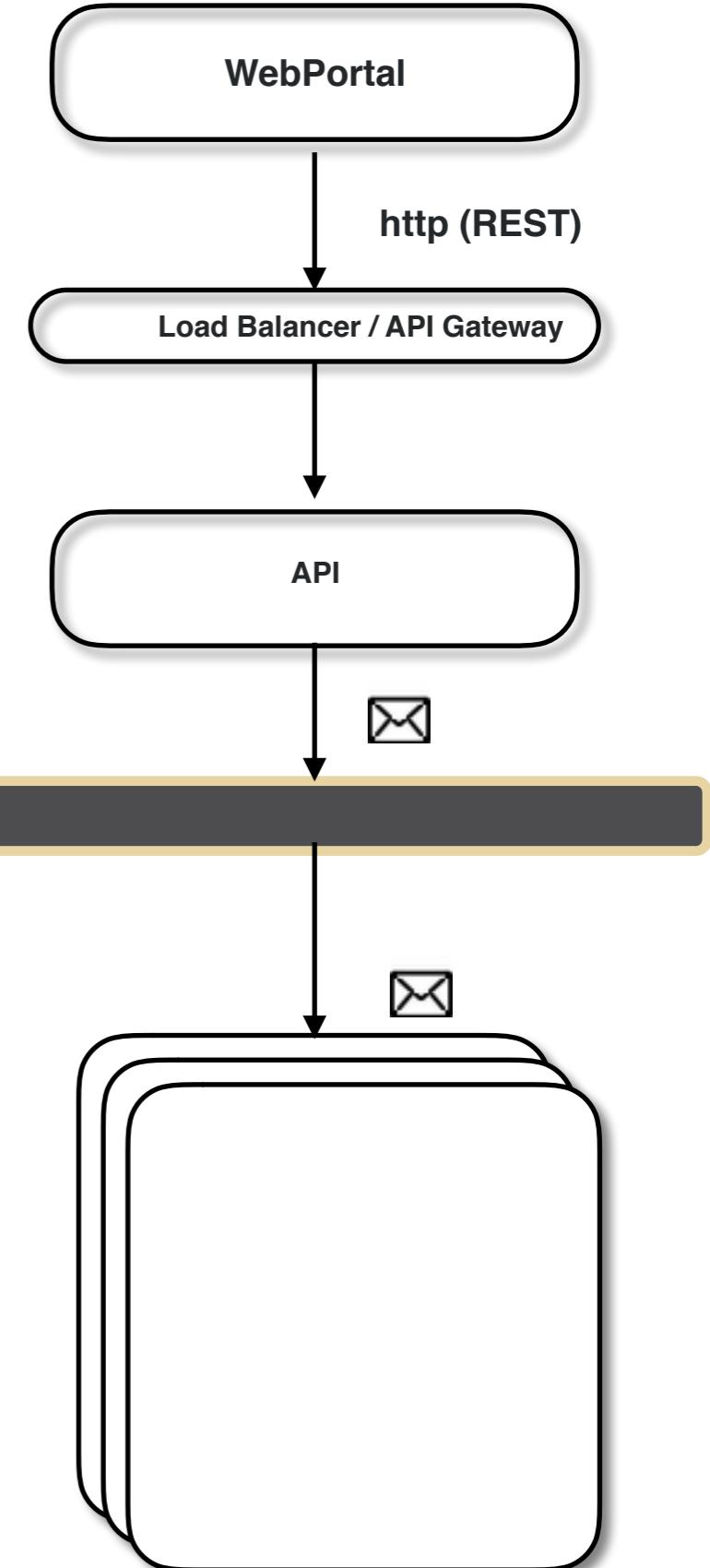




API vs Messaging

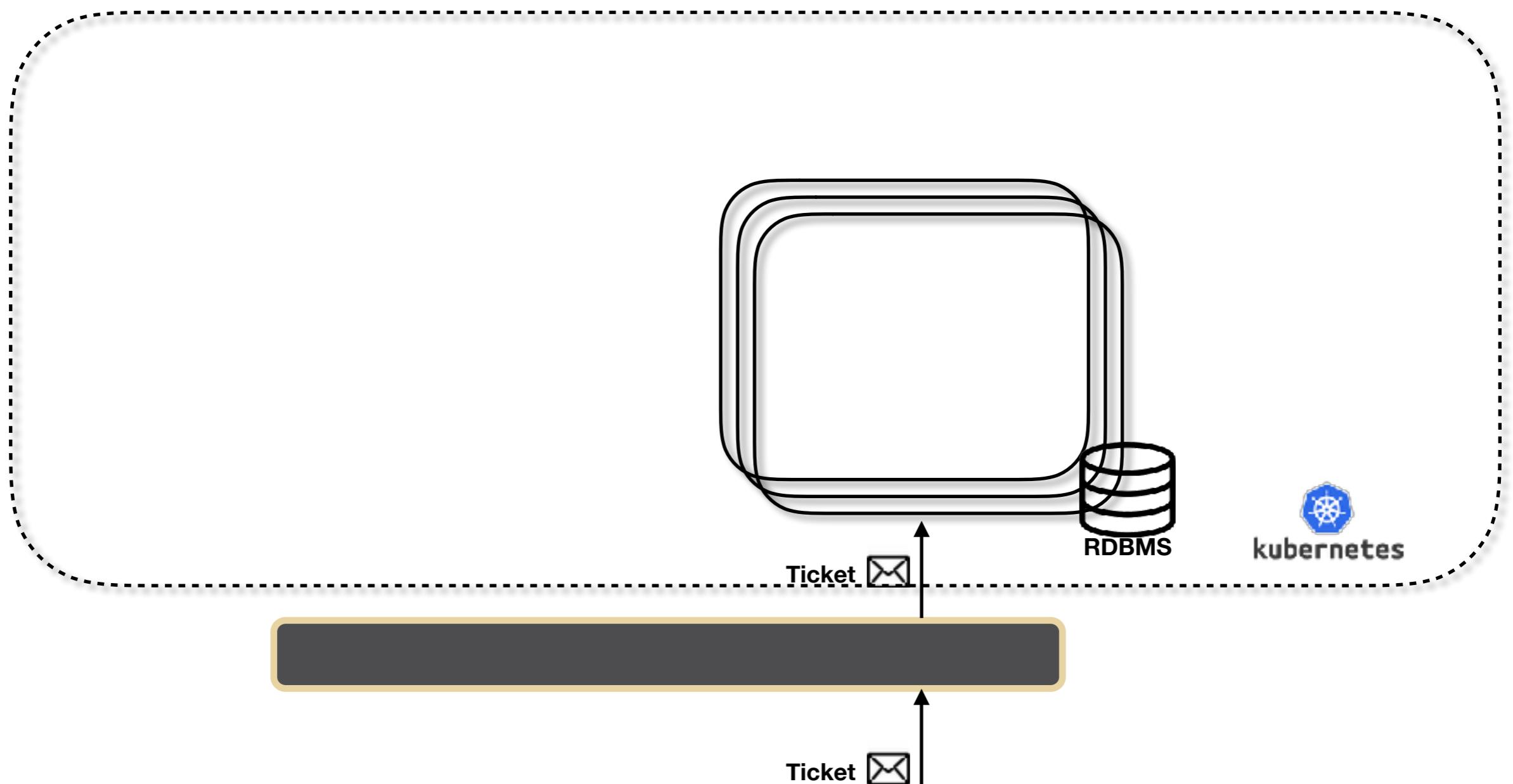


- # Scale
- # Coupling
- # Complex pattern



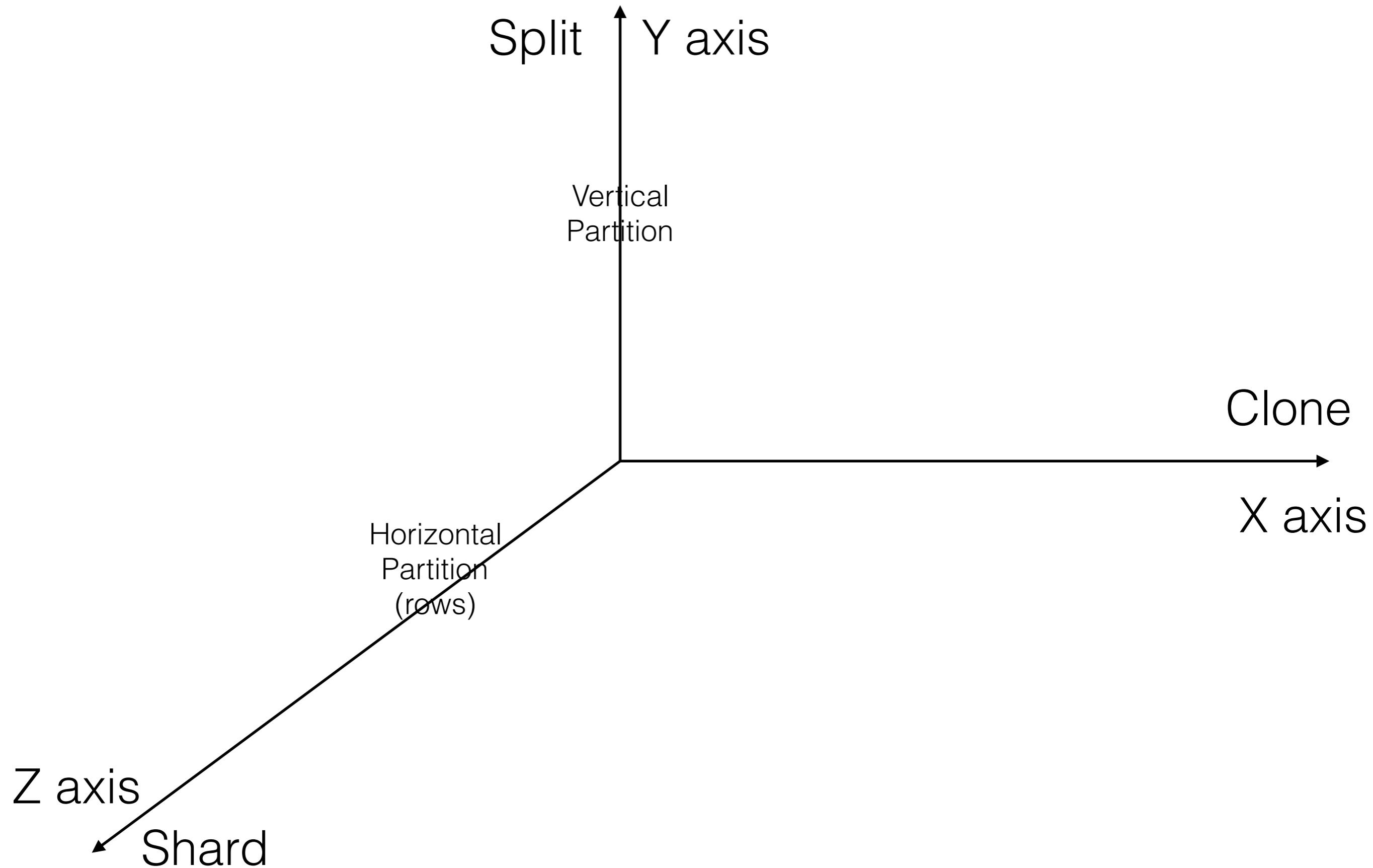
- # Unordered delivery
- # one way
- # duplicate message



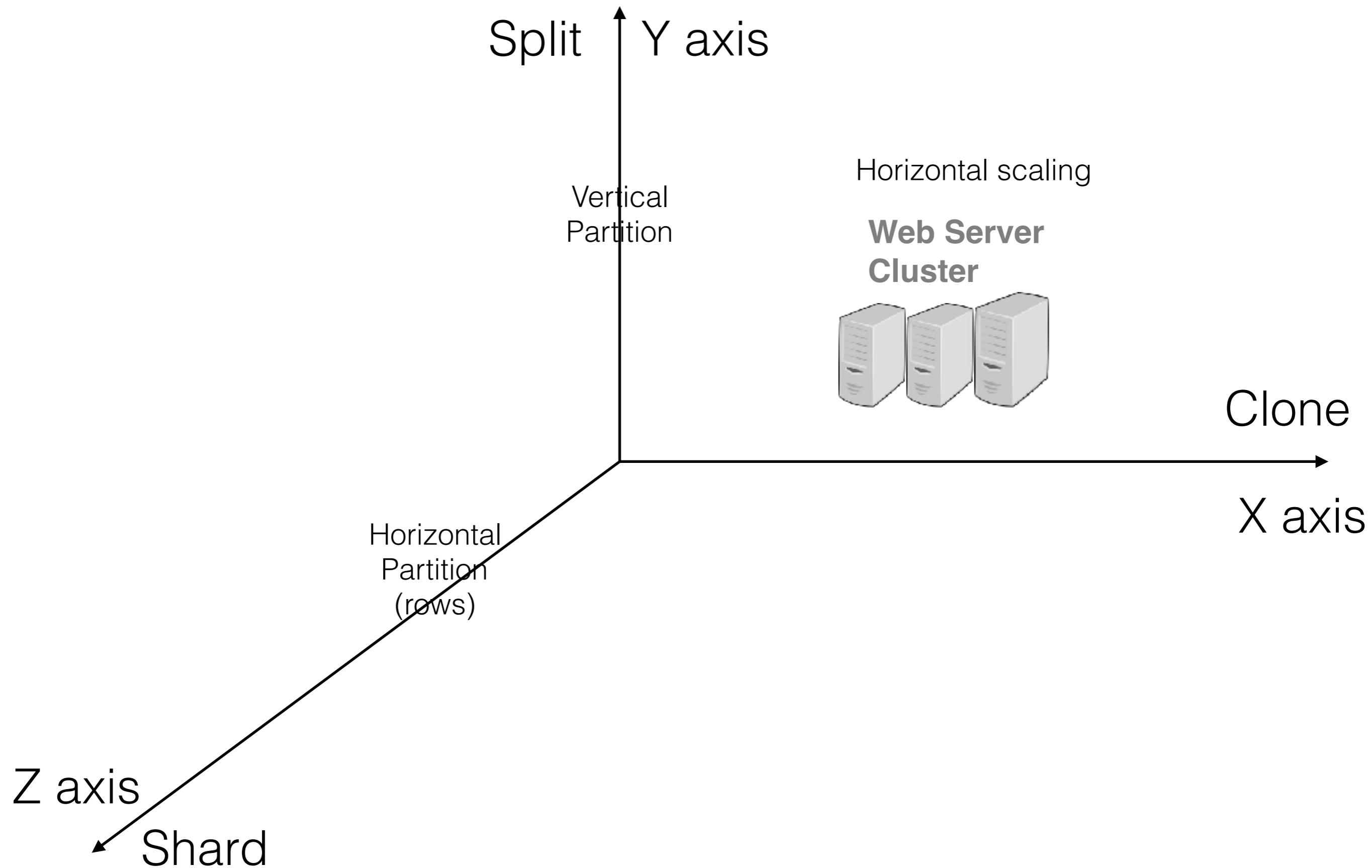


Scalability

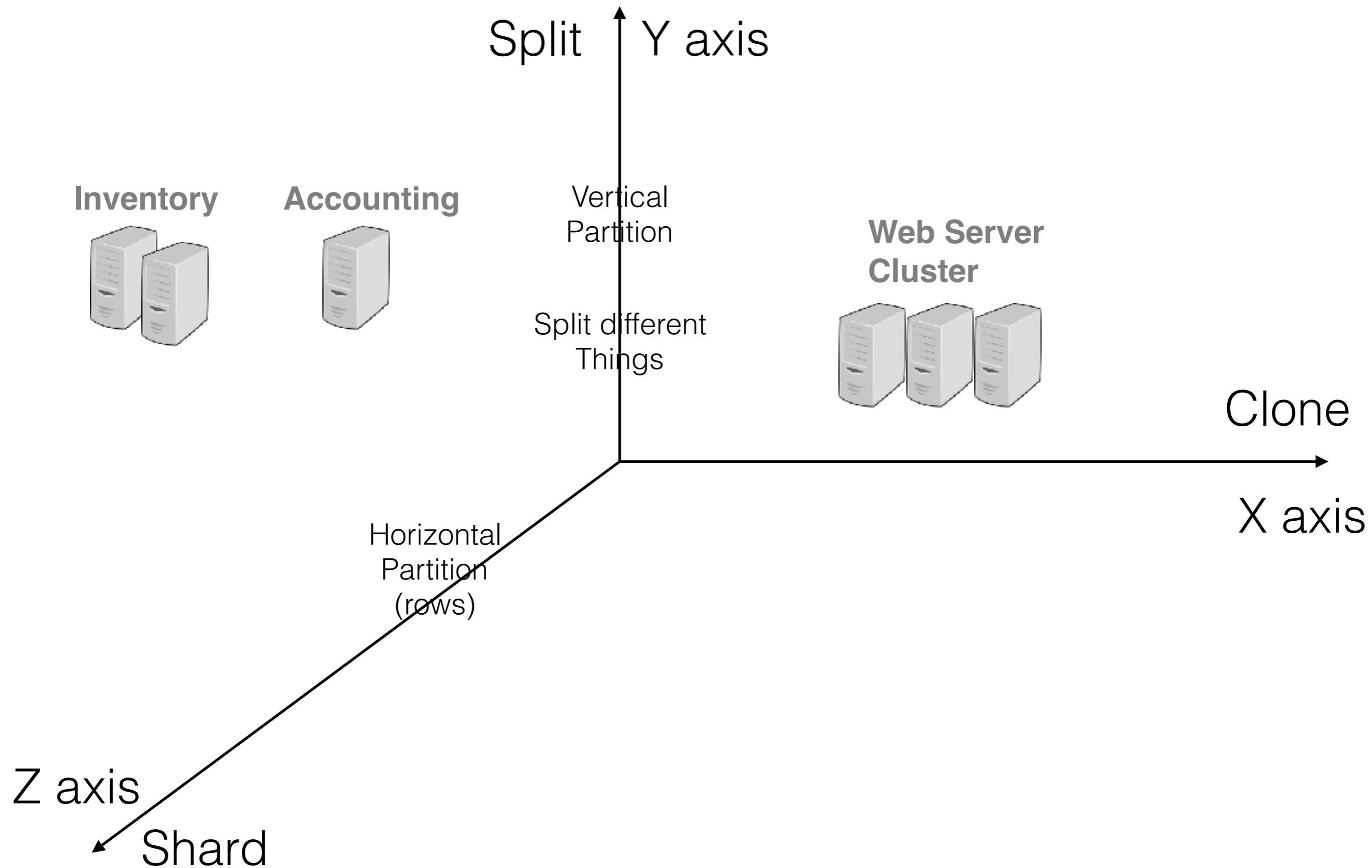
“Scalability Cube” - 50 rules for high Scalability



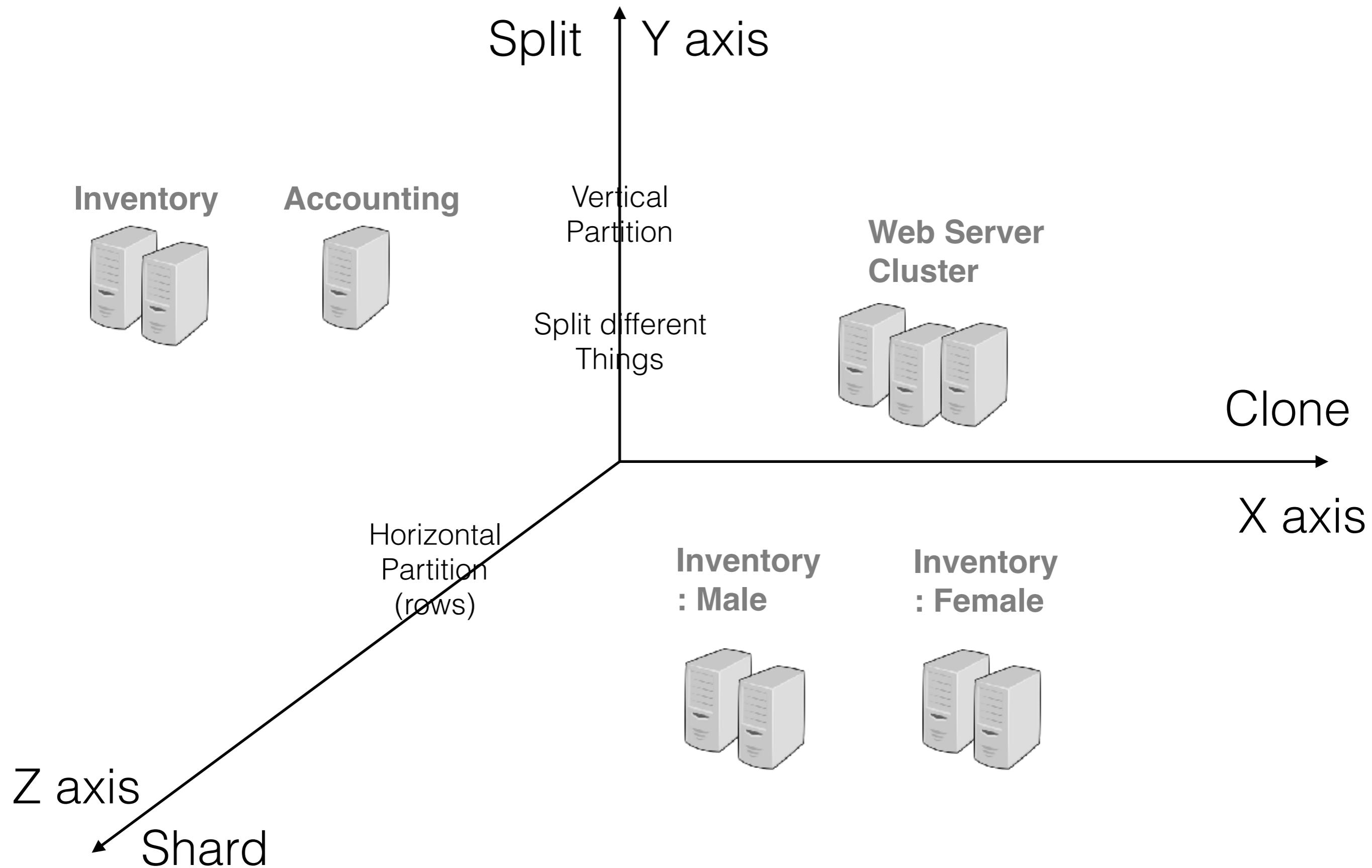
“Scalability Cube” - 50 rules for high Scalability



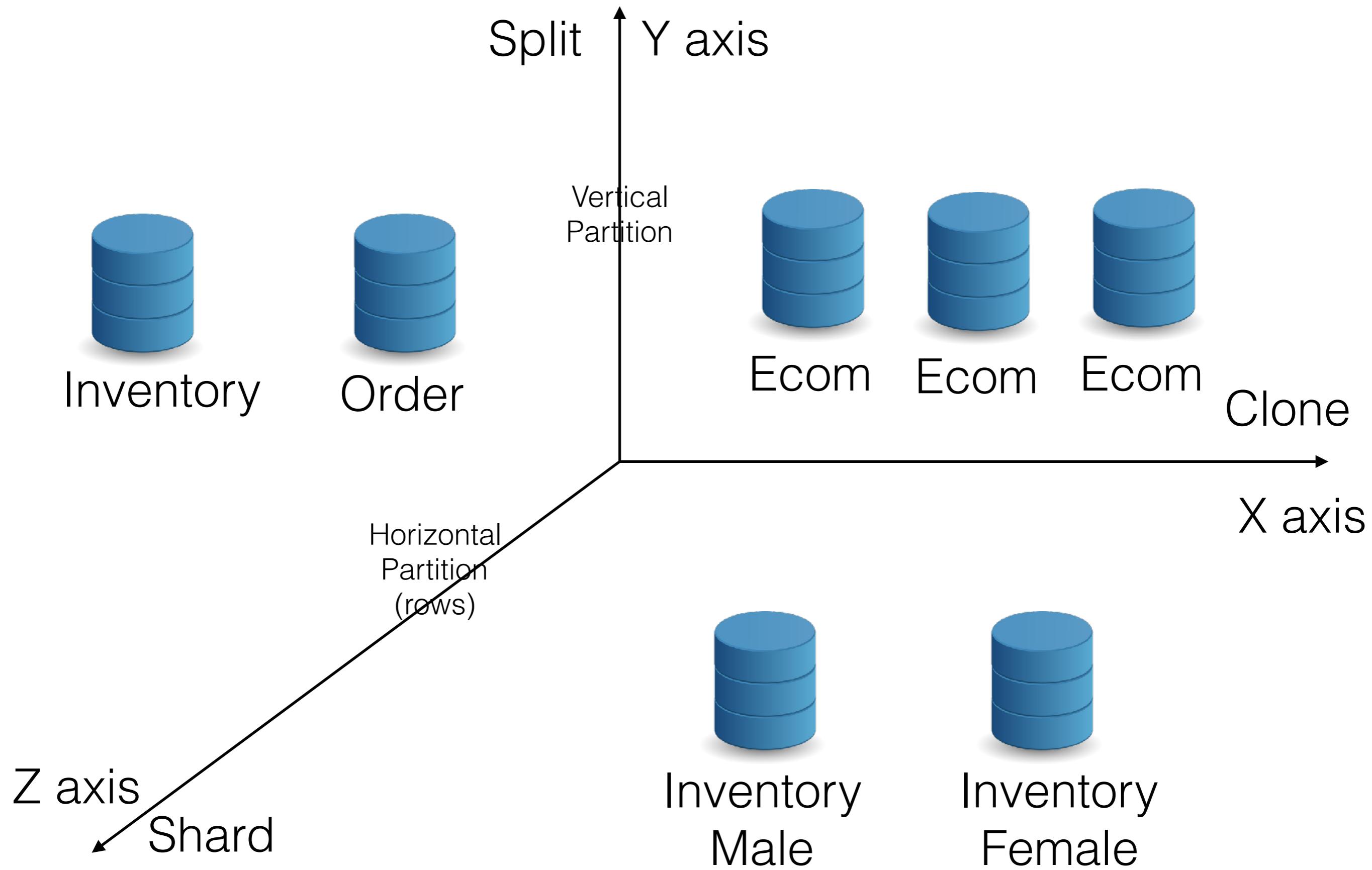
“Scalability Cube” - 50 rules for high Scalability



“Scalability Cube” - 50 rules for high Scalability

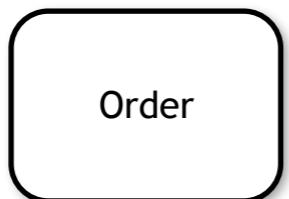


“Scalability Cube” - 50 rules for high Scalability



“Scalability Cube” - 50 rules for high Scalability

Separate domain functionality



Inventory

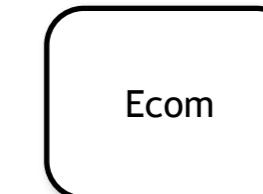
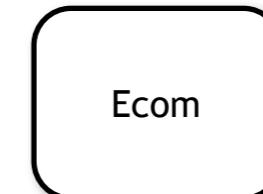
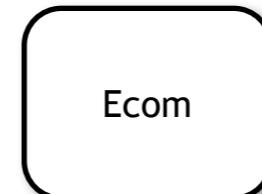


Order

Split

Y axis

No design (ops decision)



3

Vertical Partition



Ecom



Ecom



Ecom

Clone

1

design

Horizontal Partition
(rows)

X axis

Z axis

Shard

design
2

Separate by data

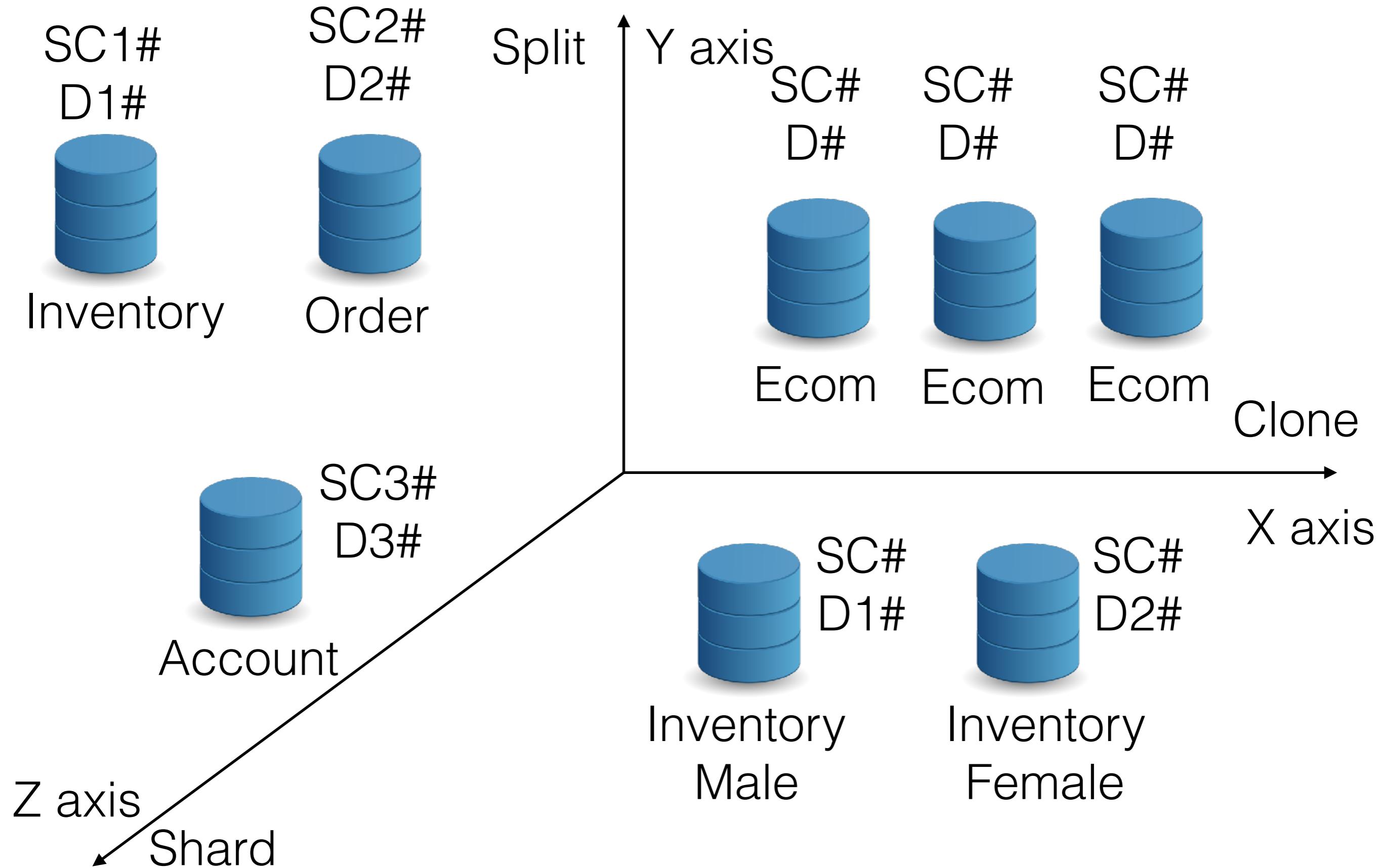


Inventory
Male

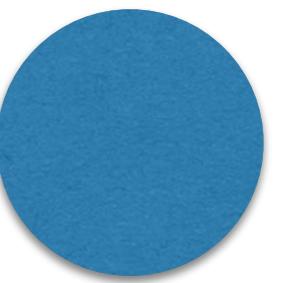
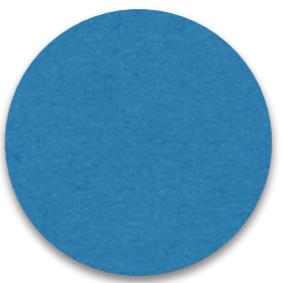
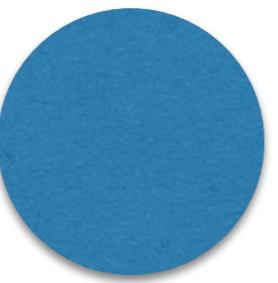


Inventory
Female

Scalability Cube - 50 rules for high Scalability



Counter



Counter

C

BP

Counter

E

BP

Counter

?

BP

General

C

E

?

BP

BP

BP

BP

BP

BP

General

C

E

?

BP

D BP

I BP

BP

D BP

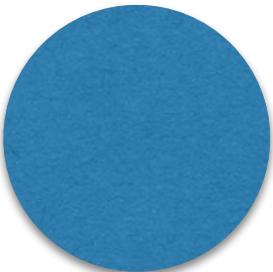
I BP

General

C

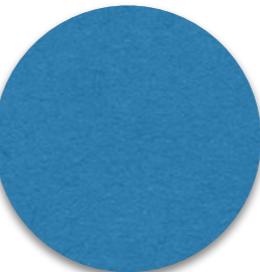
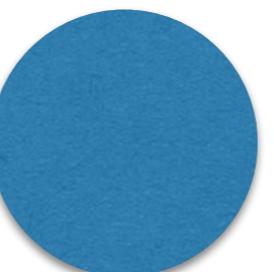
E

?



D BP

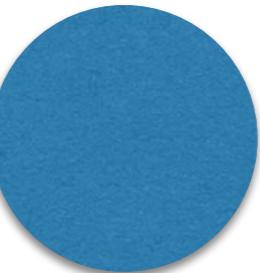
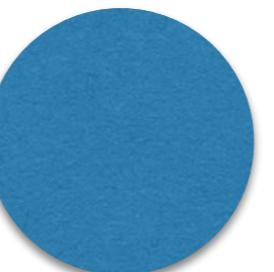
D BP



D BP

D BP

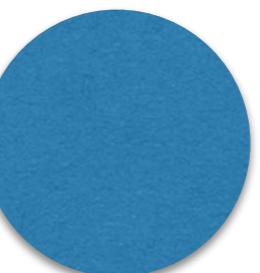
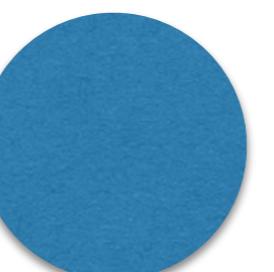
D BP



D BP

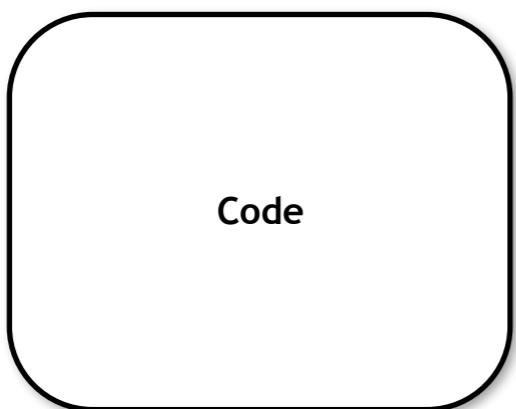
I BP

I BP



I BP

Day 2



CQRS

Event driven Architecture

message : command vs event

Materialized View

SAGA : Compensatable transaction

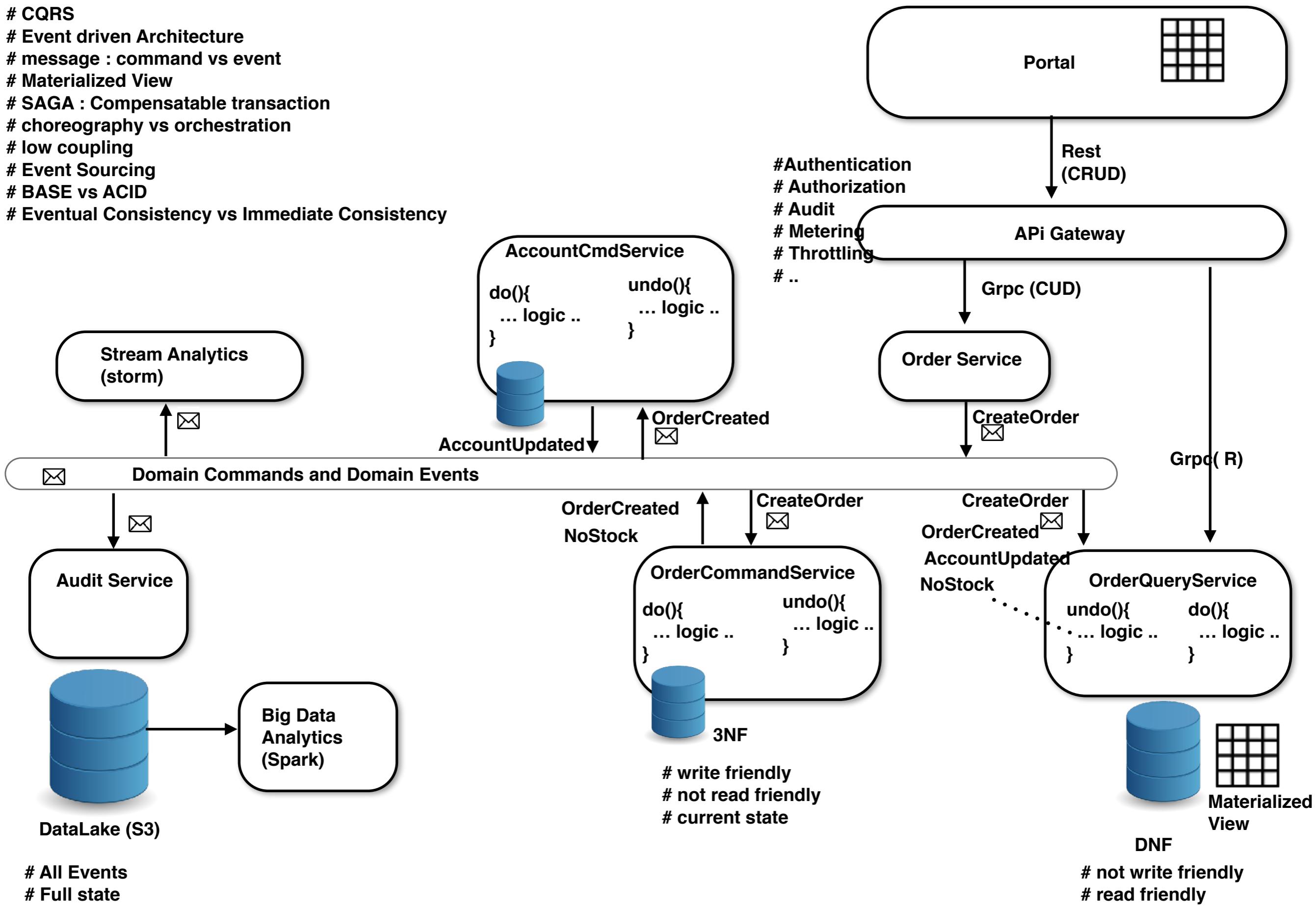
choreography vs orchestration

low coupling

Event Sourcing

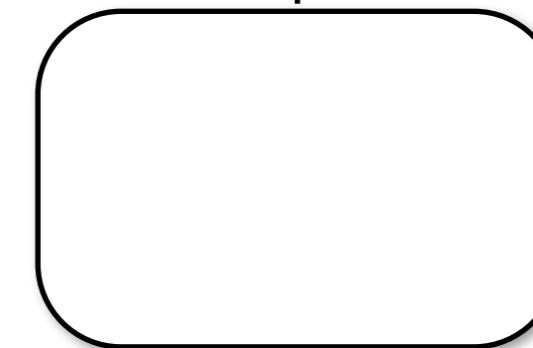
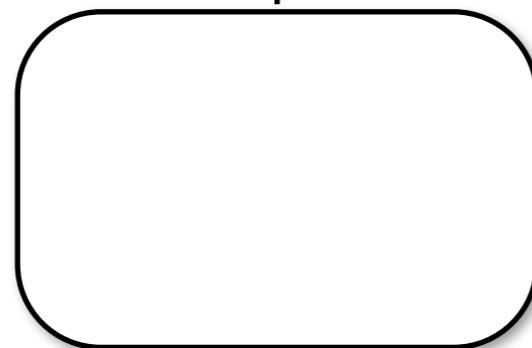
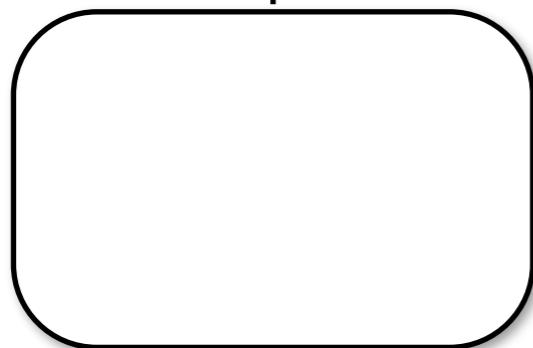
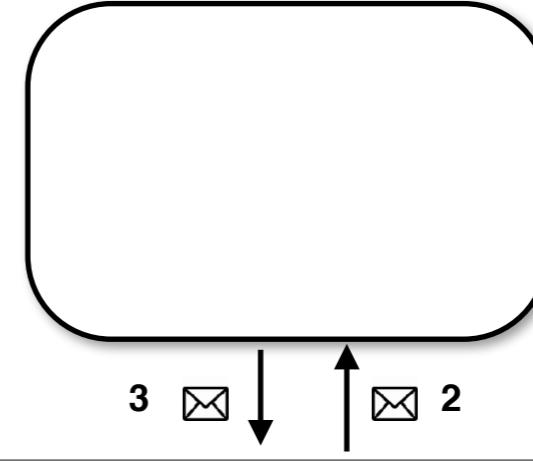
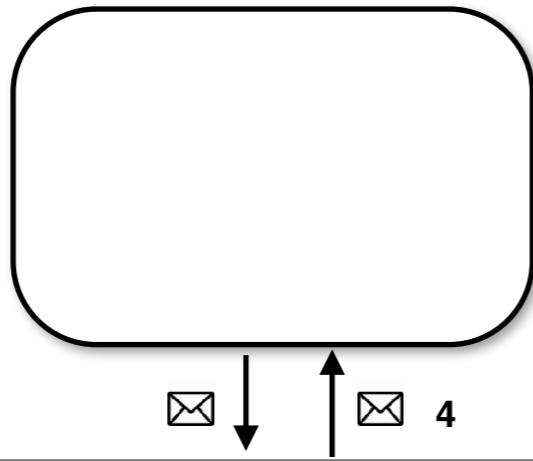
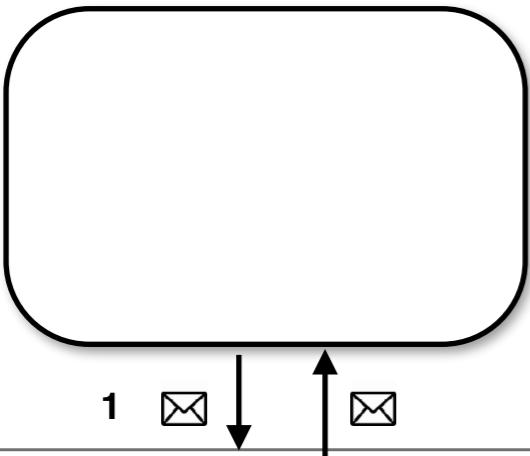
BASE vs ACID

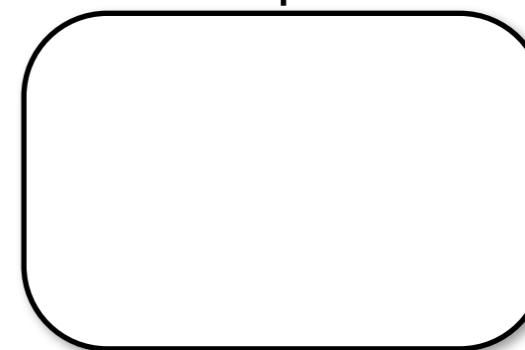
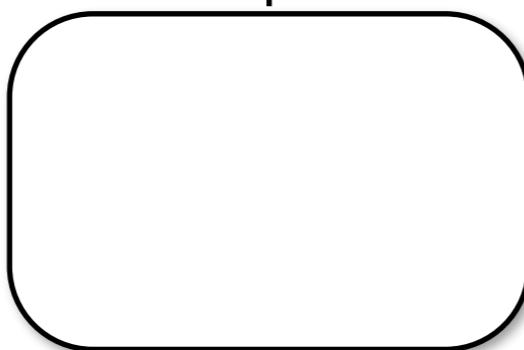
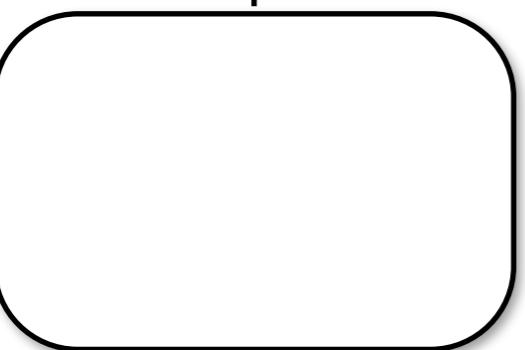
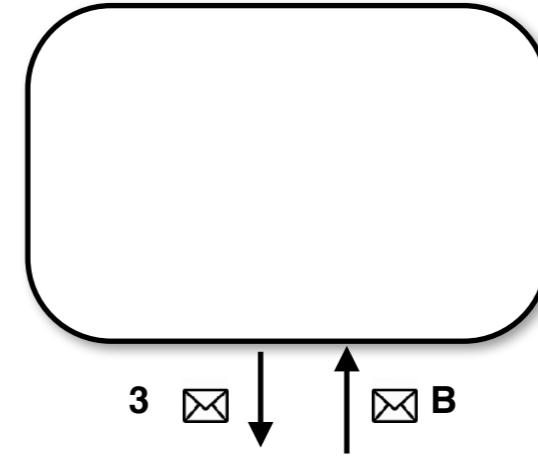
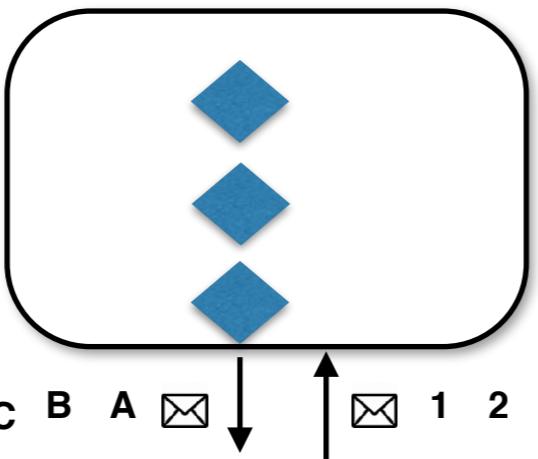
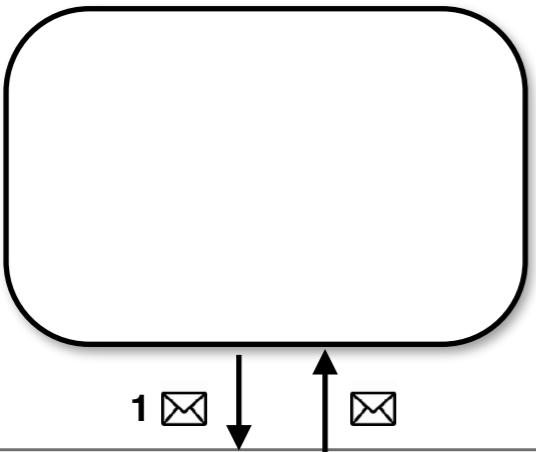
Eventual Consistency vs Immediate Consistency



All Events

Full state



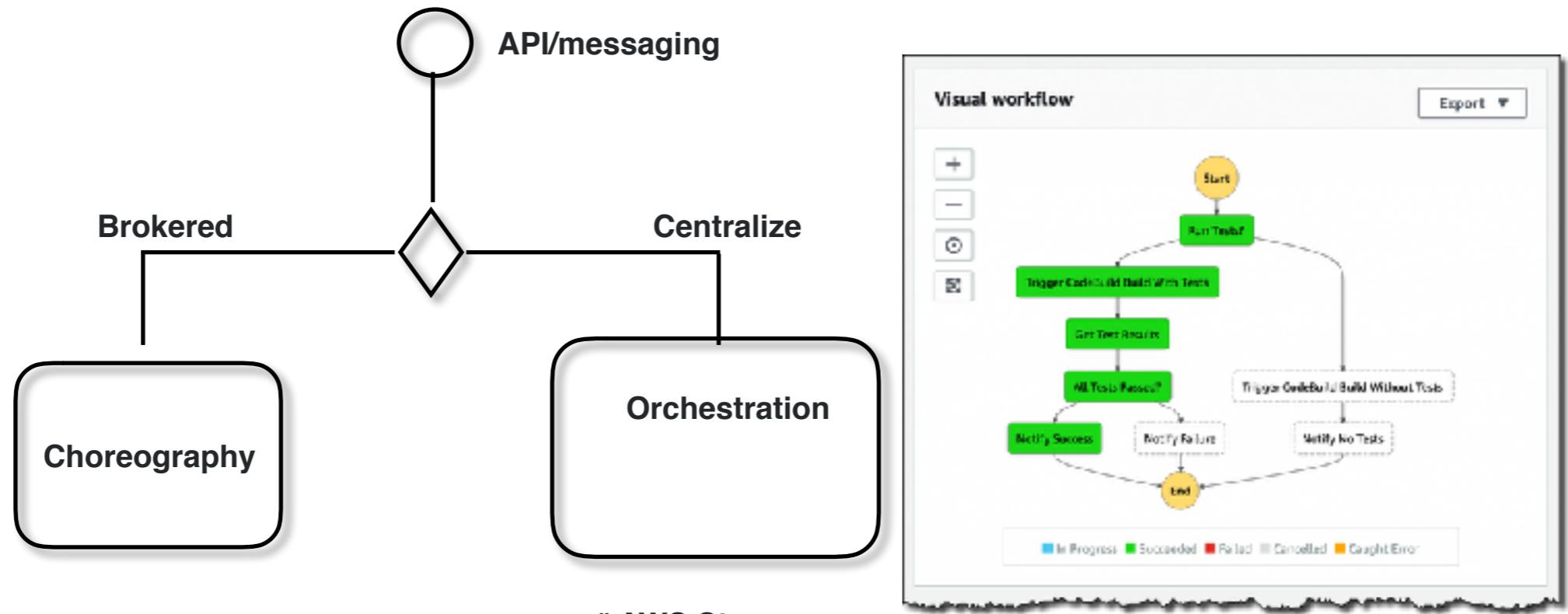


C \square \downarrow \uparrow \square 4

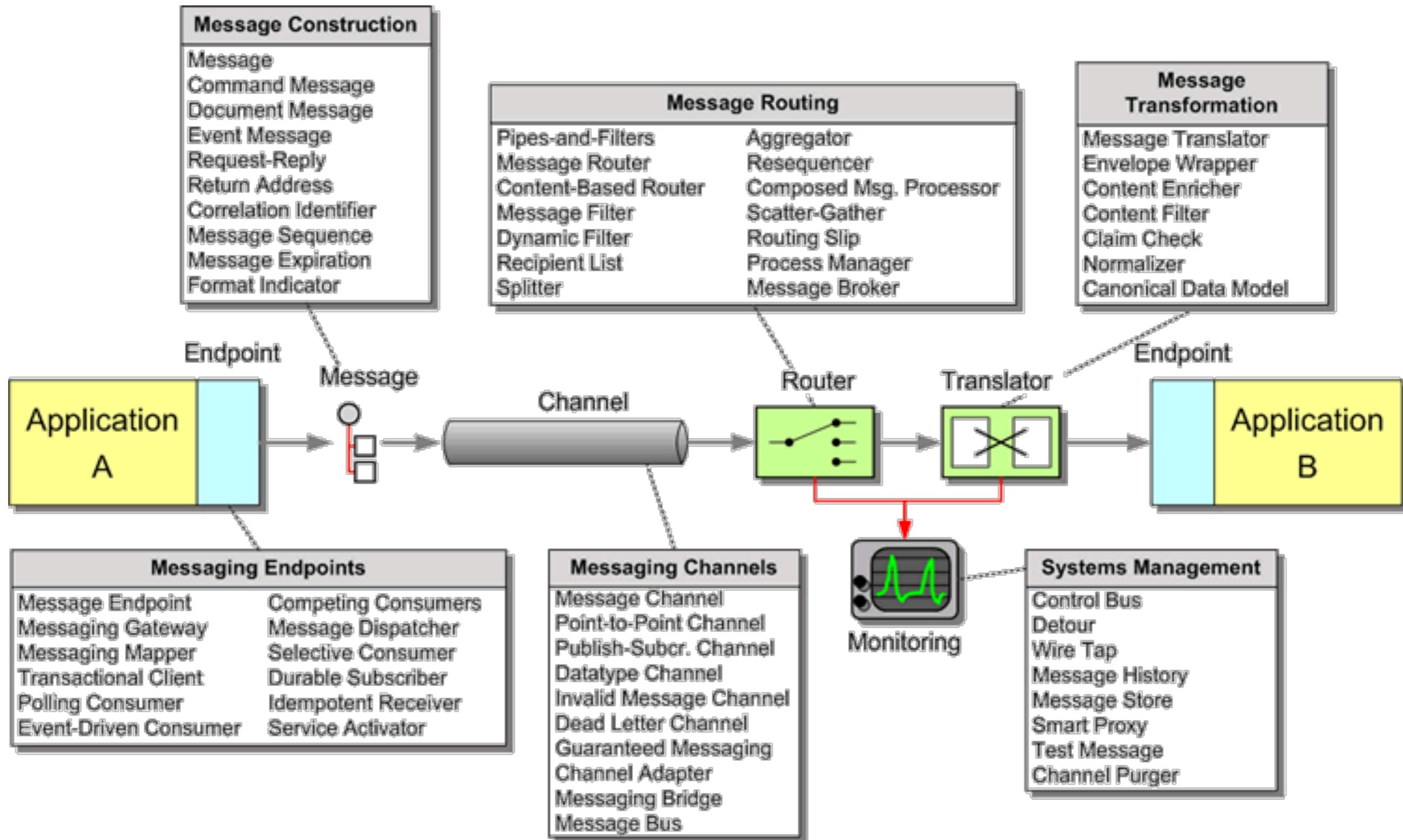
\square \downarrow \uparrow \square

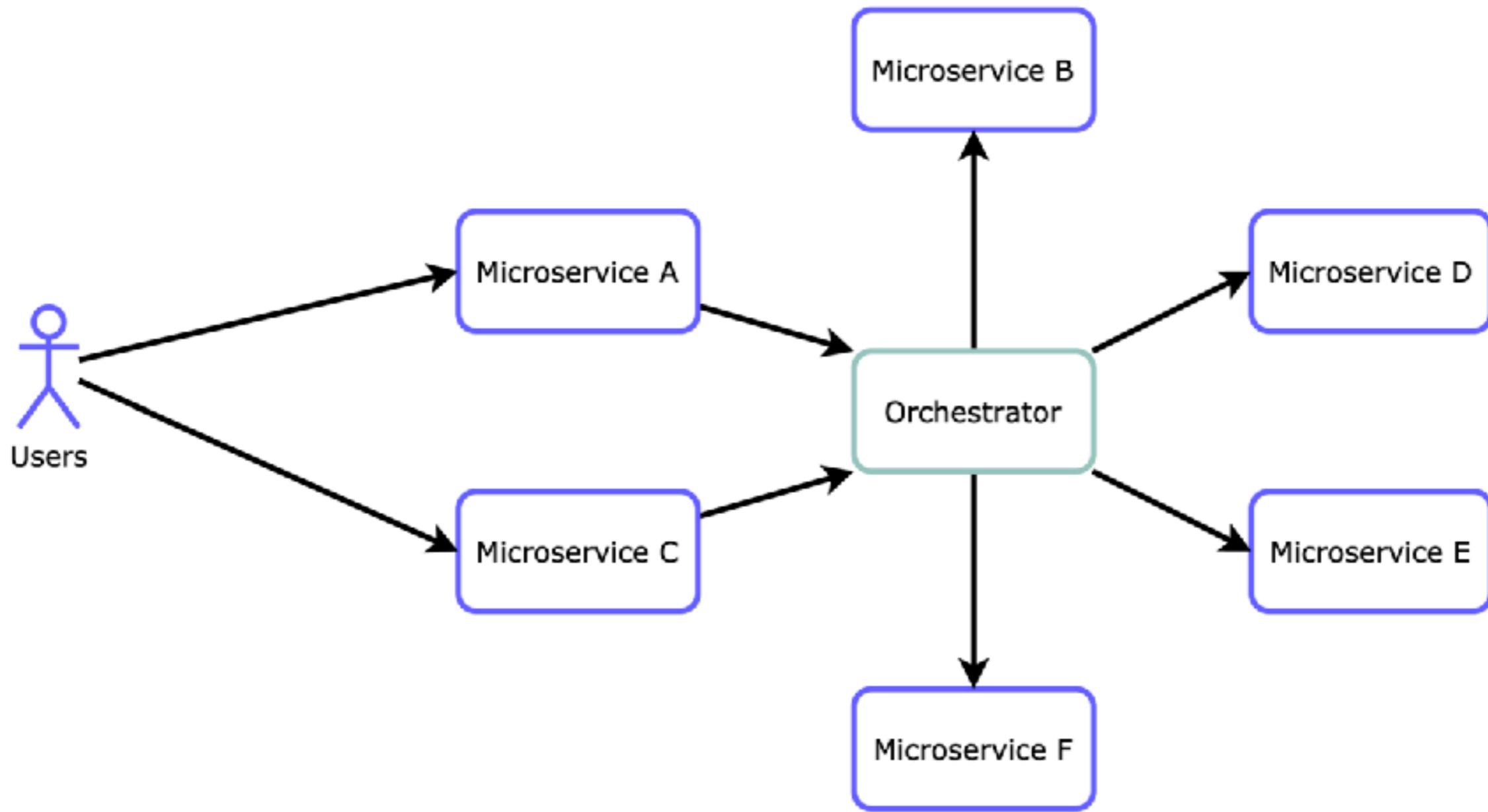
A \square \downarrow \uparrow \square 2

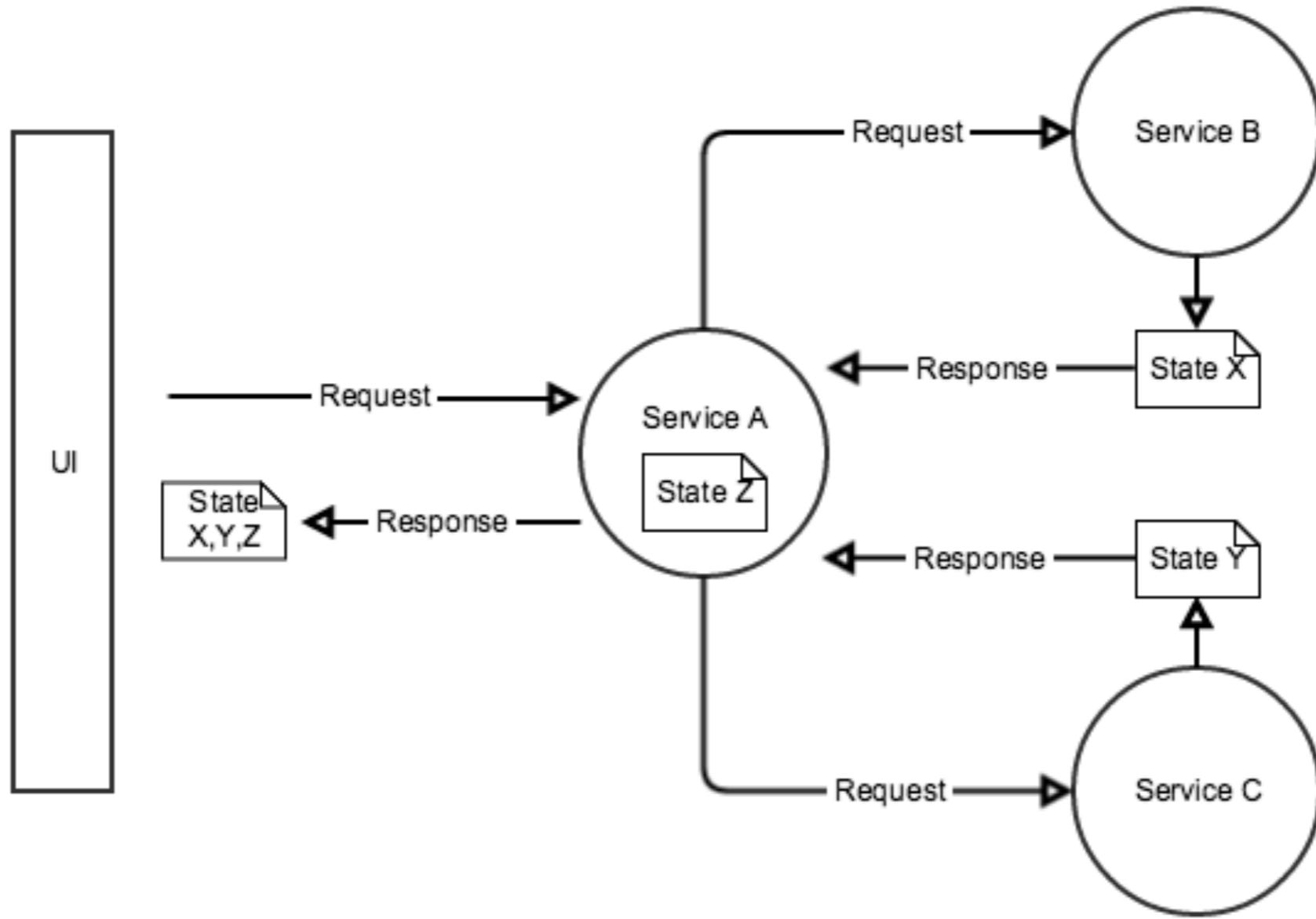
Choose Communication Patterns

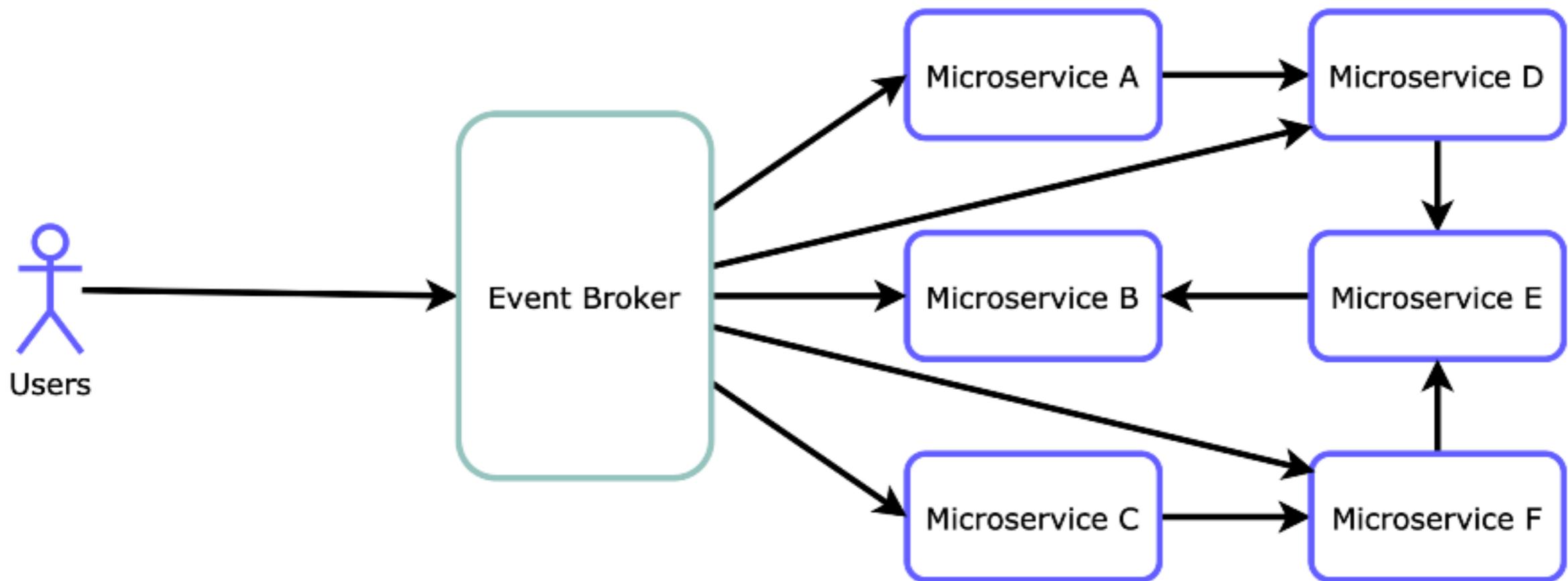


AWS Step
Azure Logic App
Biztalk
Webmethods
Mule
ServiceNow
#

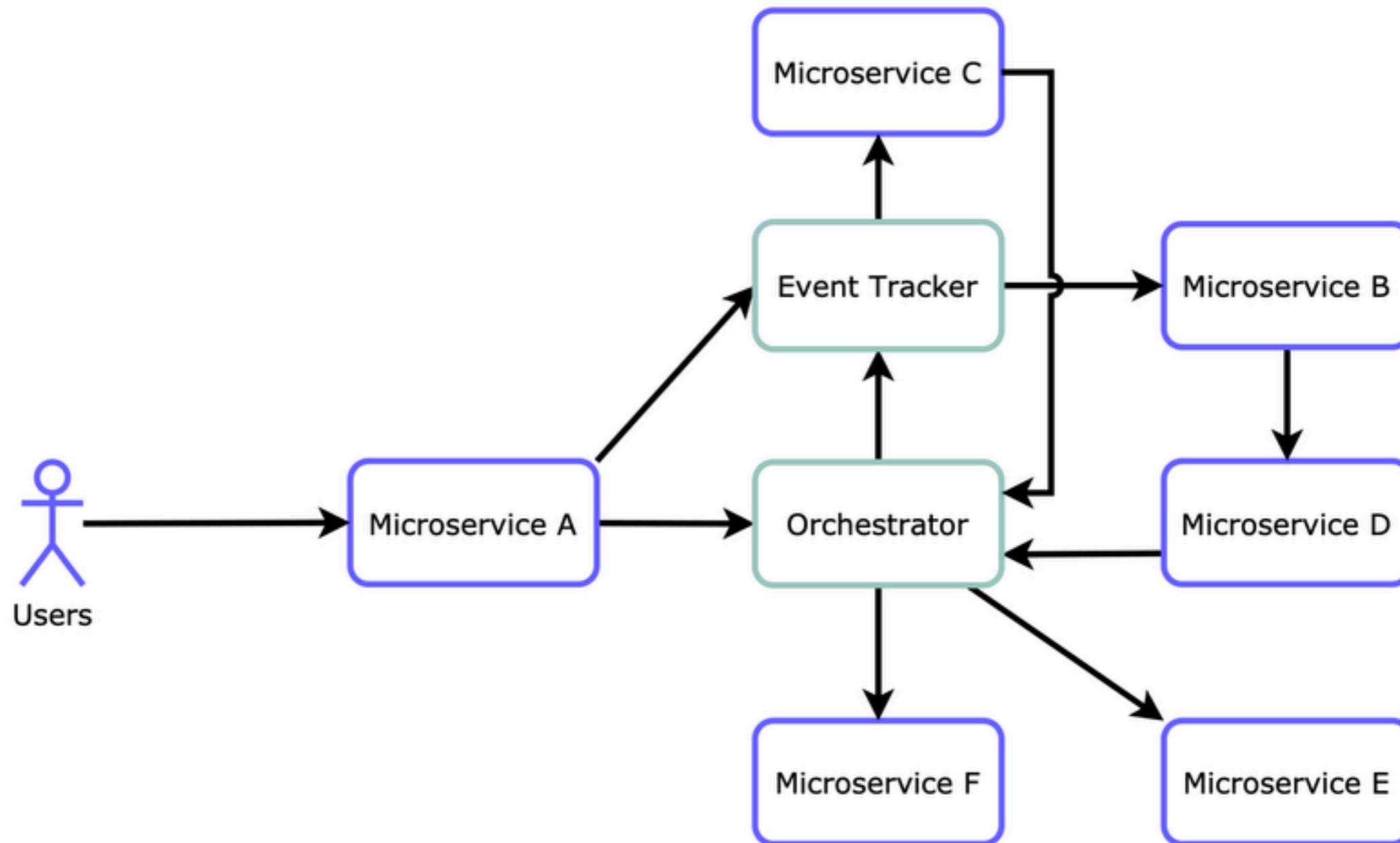


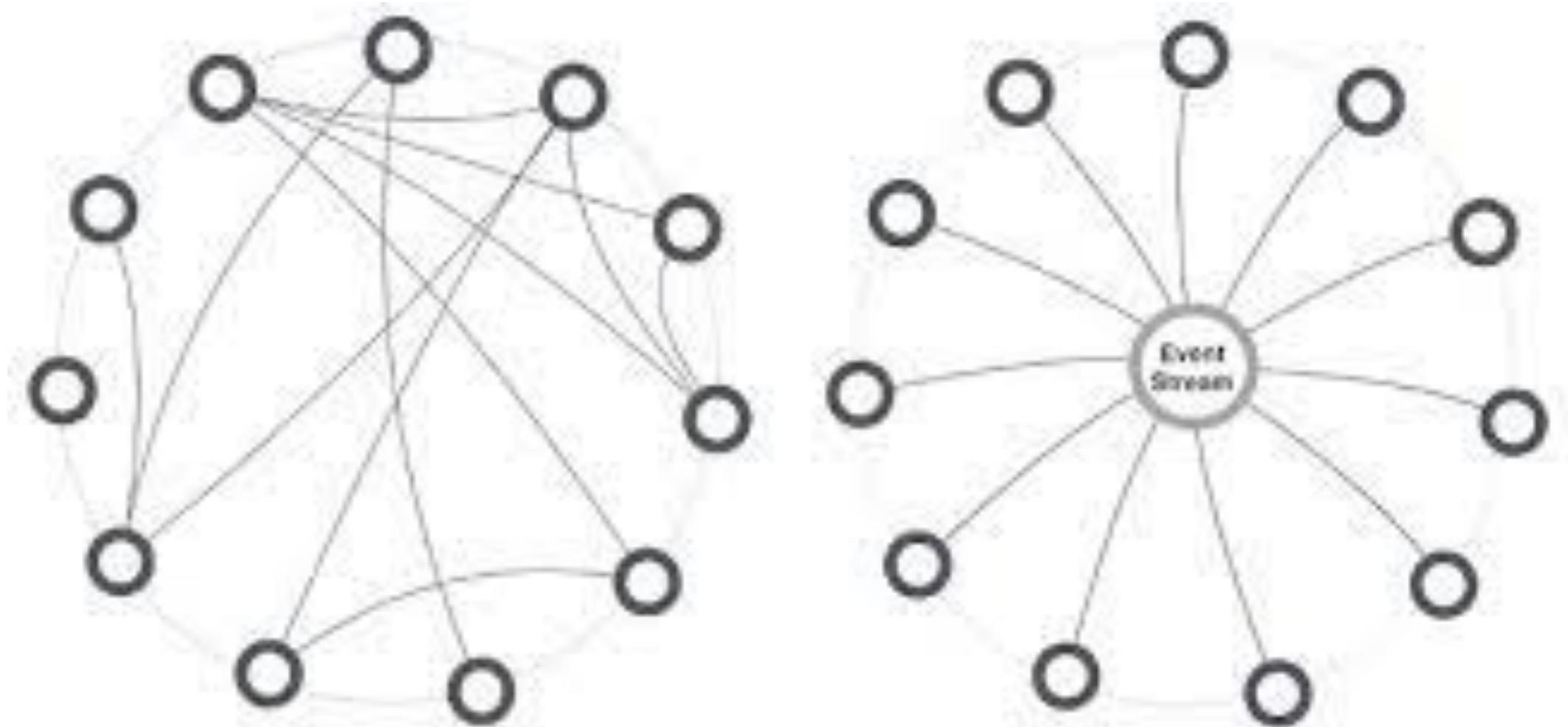


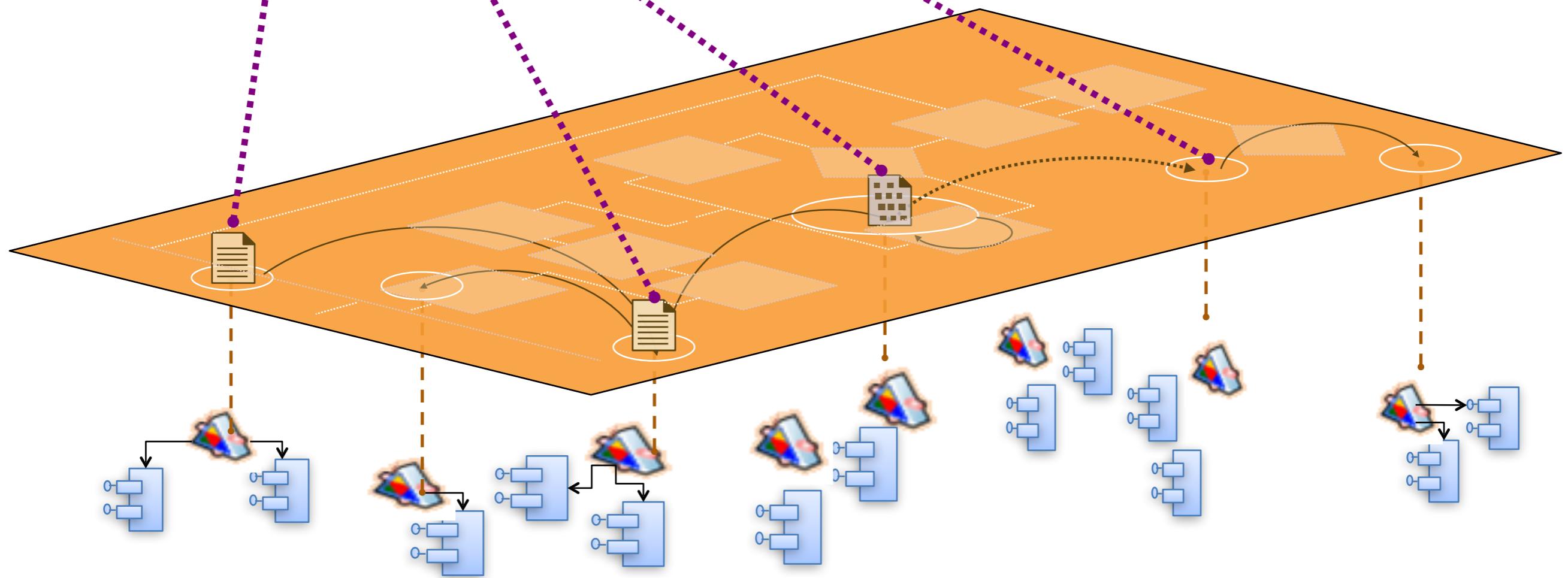
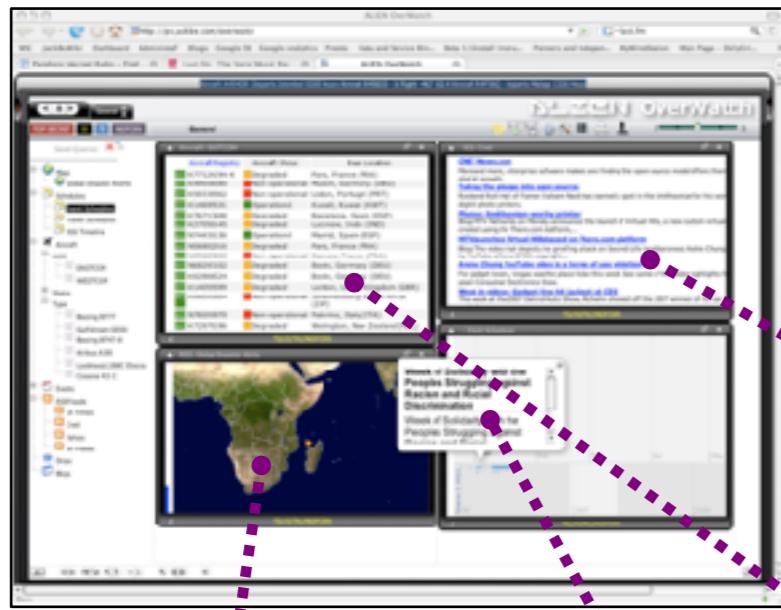




Hybrid







DNF

TABLE_BOOK_DETAIL

Book ID	Genre ID	Genre Type	Price
1	1	Gardening	25.99
2	2	Sports	14.99
3	1	Gardening	10.00
4	3	Travel	12.99
5	2	Sports	17.99

<— duplicates
<— no joins

3NF

TABLE_BOOK

Book ID	Genre ID	Price
1	1	25.99
2	2	14.99
3	1	10.00
4	3	12.99
5	2	17.99

TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

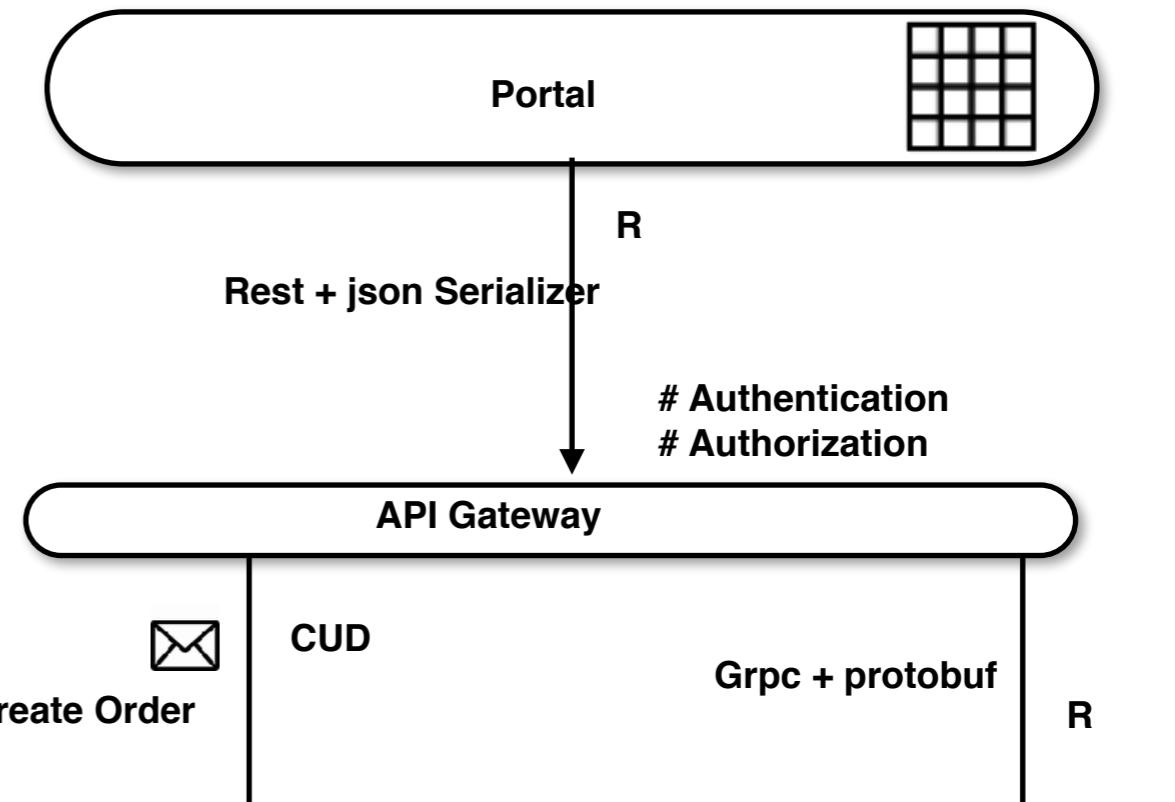
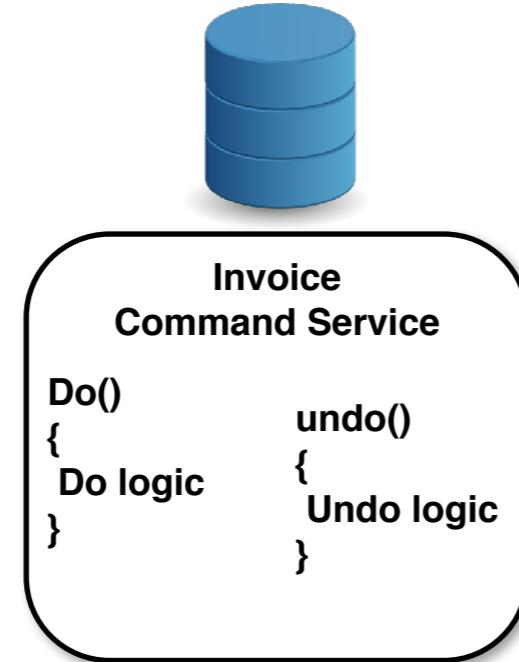
<— no duplicates
<— more joins

```

# CQRS
# Materialized View
# SAGA (compensatable transaction)
# Eventual Consistency
# CAP
# Event sourcing (DDD)
# ubiquitous language
# EDA
# Choreography
# API Gateway
# Event & Command

```

Stream Analytics
(Look for patterns in < 7 days)



Domain Events & Domain Commands

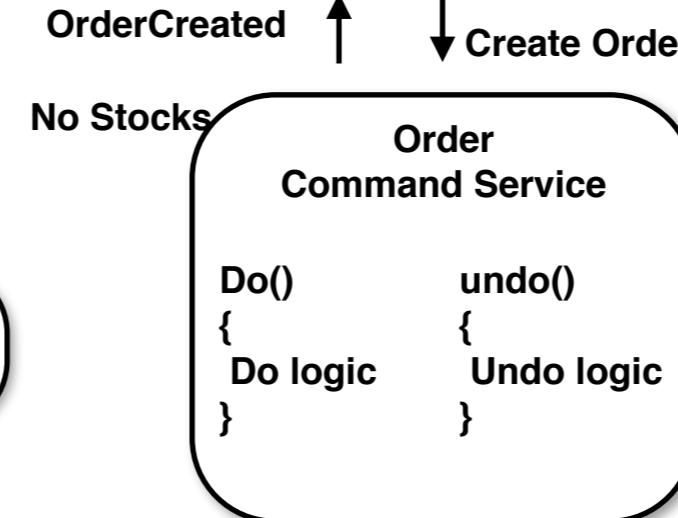


Full State
Security Audit

(Look for patterns for years)
DWH Analytics



DWH
Snapshot



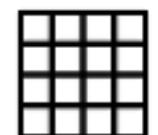
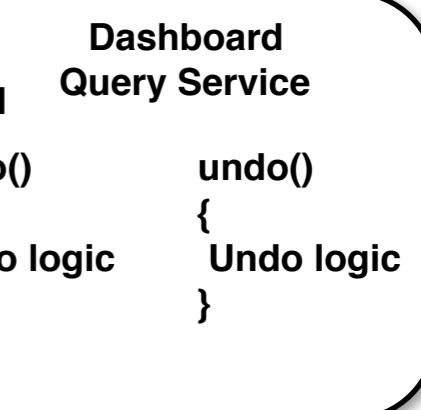
3NF

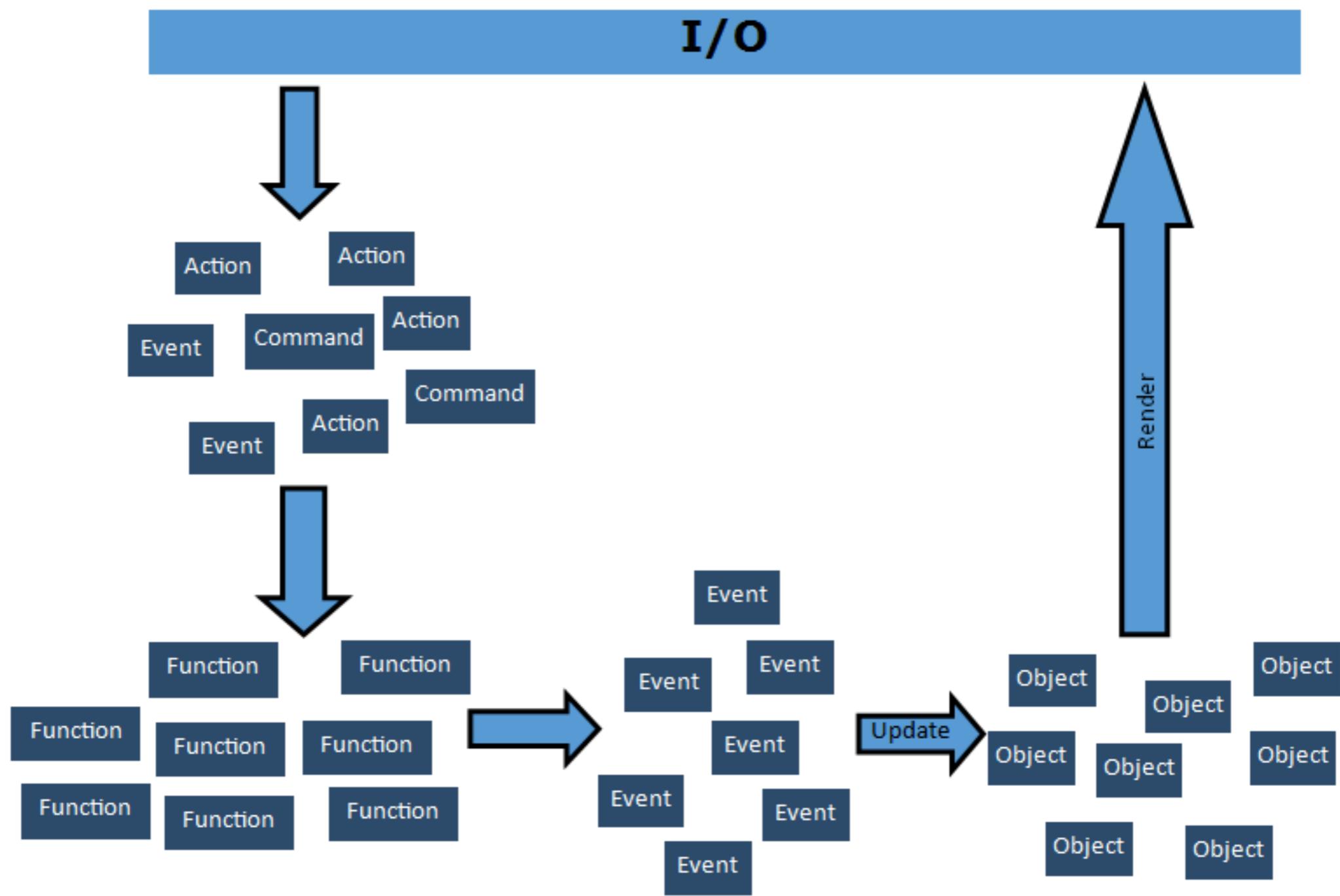
write friendly
Current State



DNF

read friendly
Current State



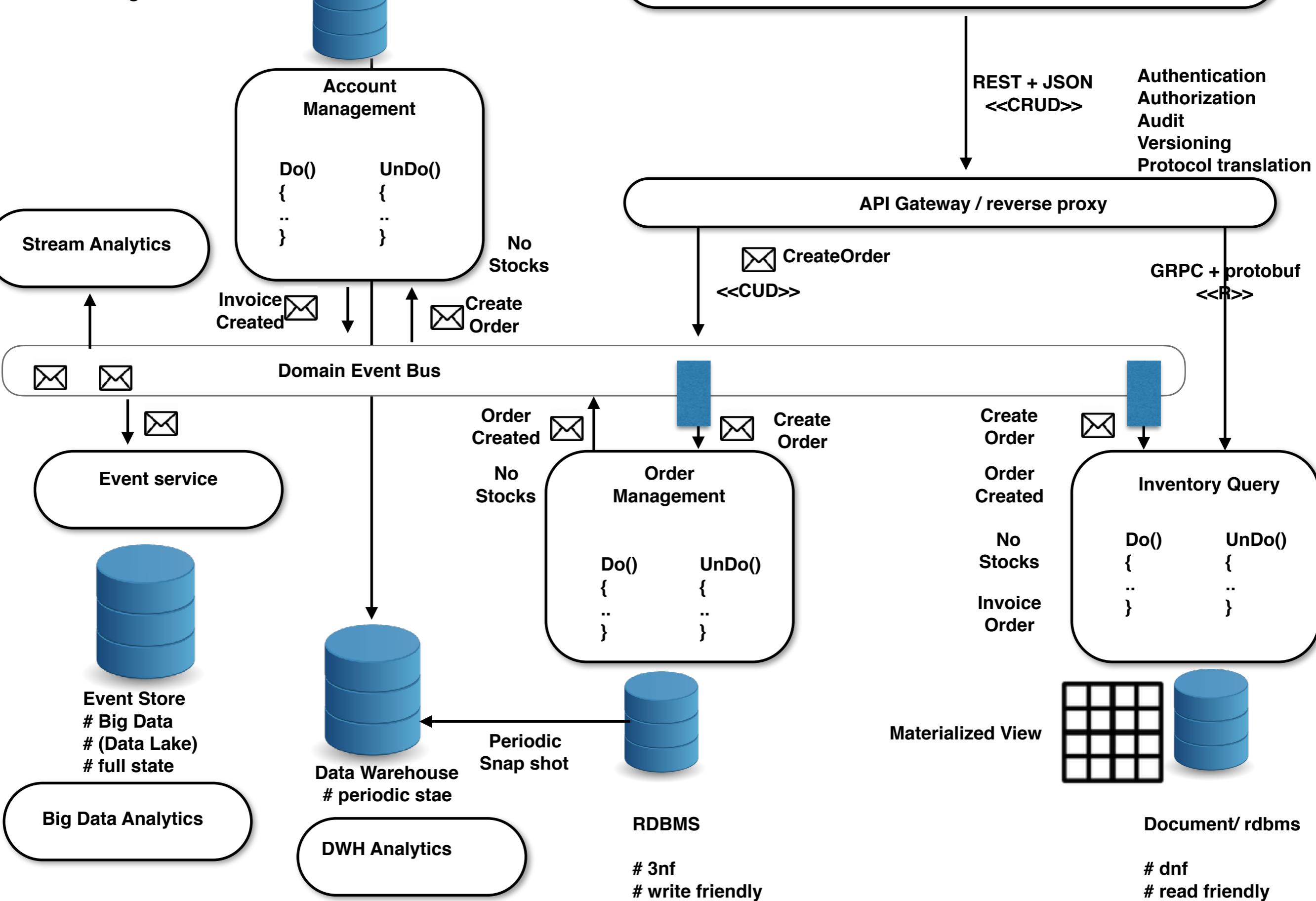


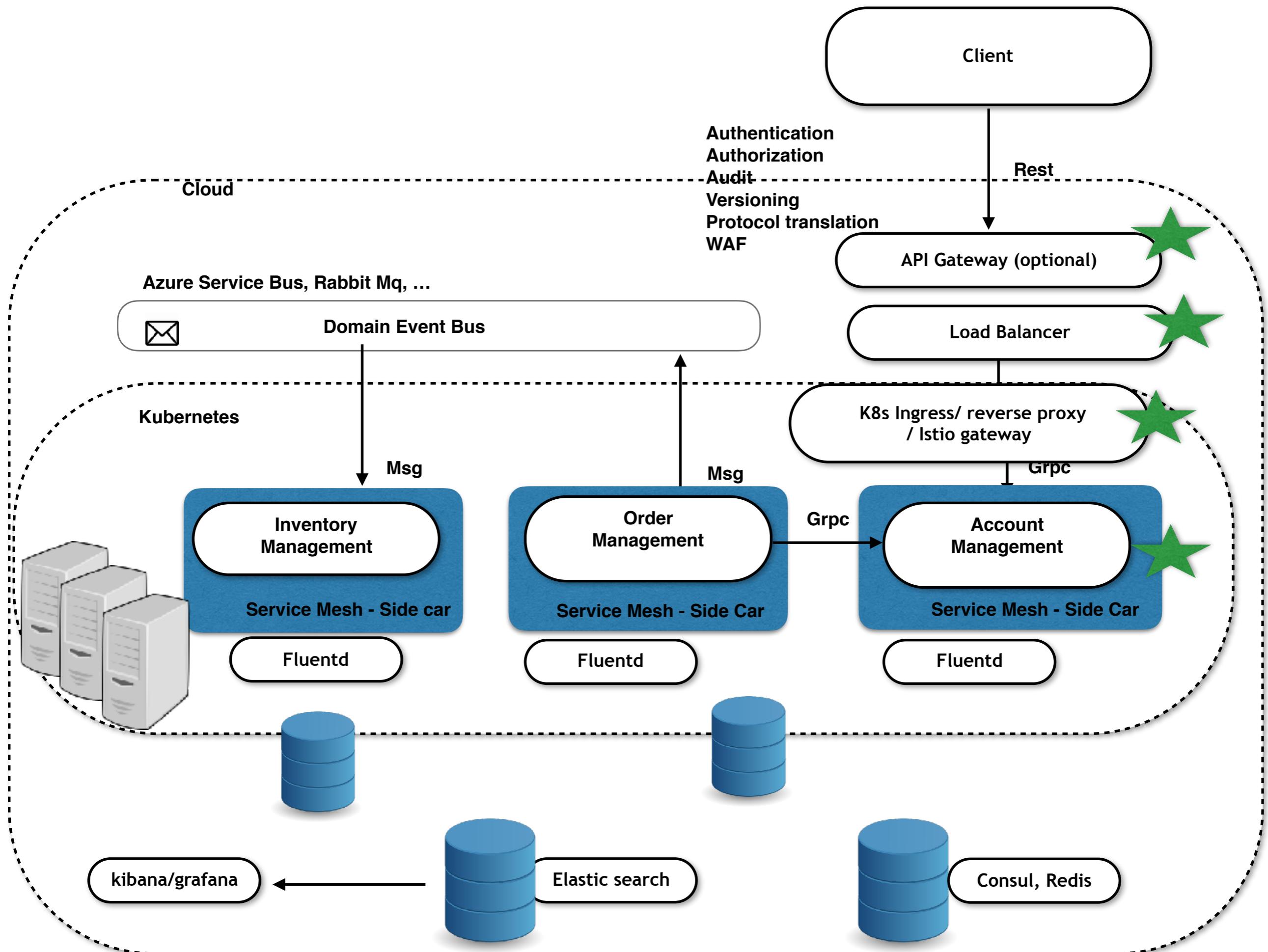
Reference Arch

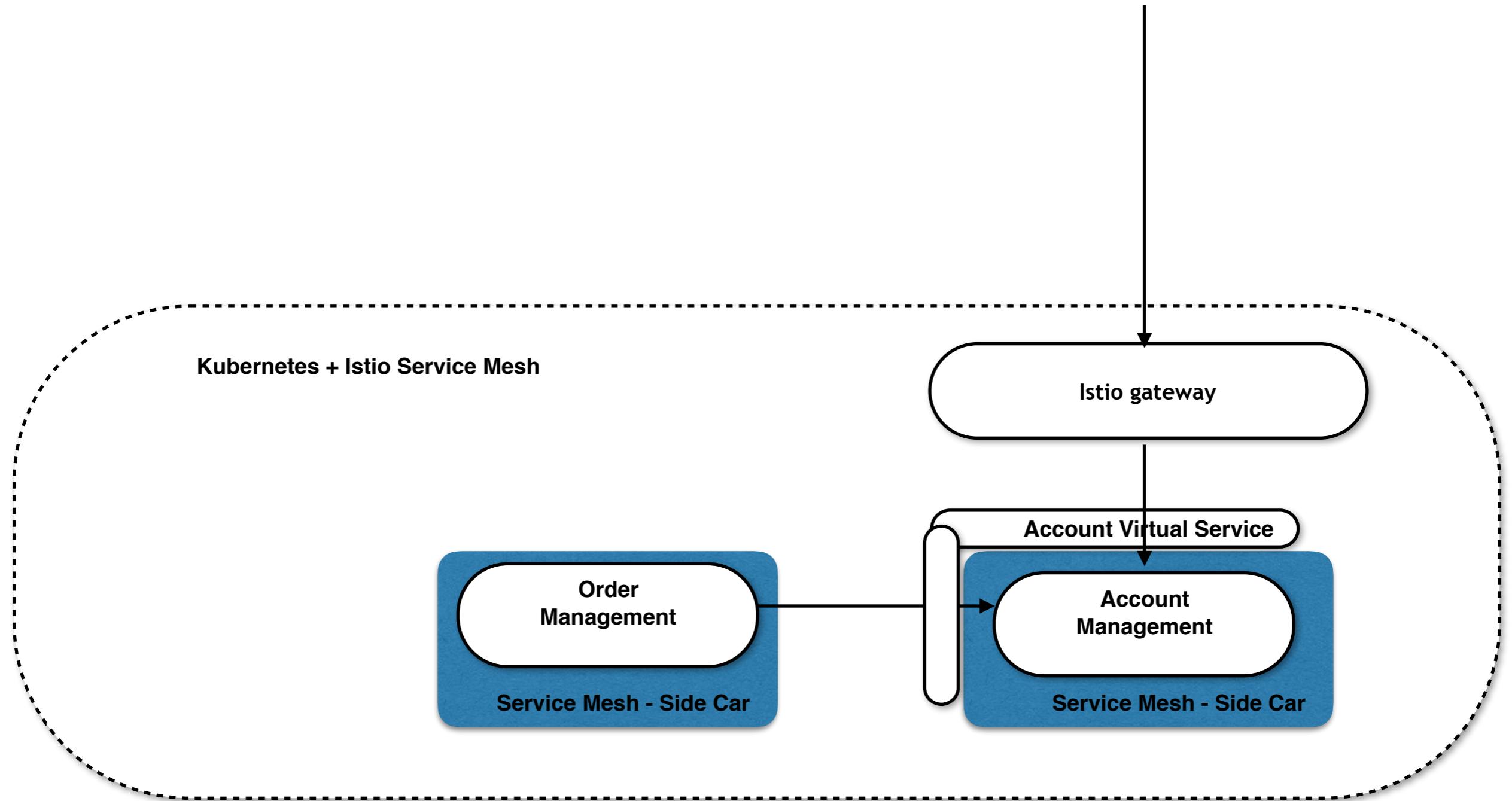
Event Driven Architecture

SAGA

Event sourcing







Jaeger, elastic APM

K8s dashboard

kibana

Grafana, Kiali, ...

Weave scope



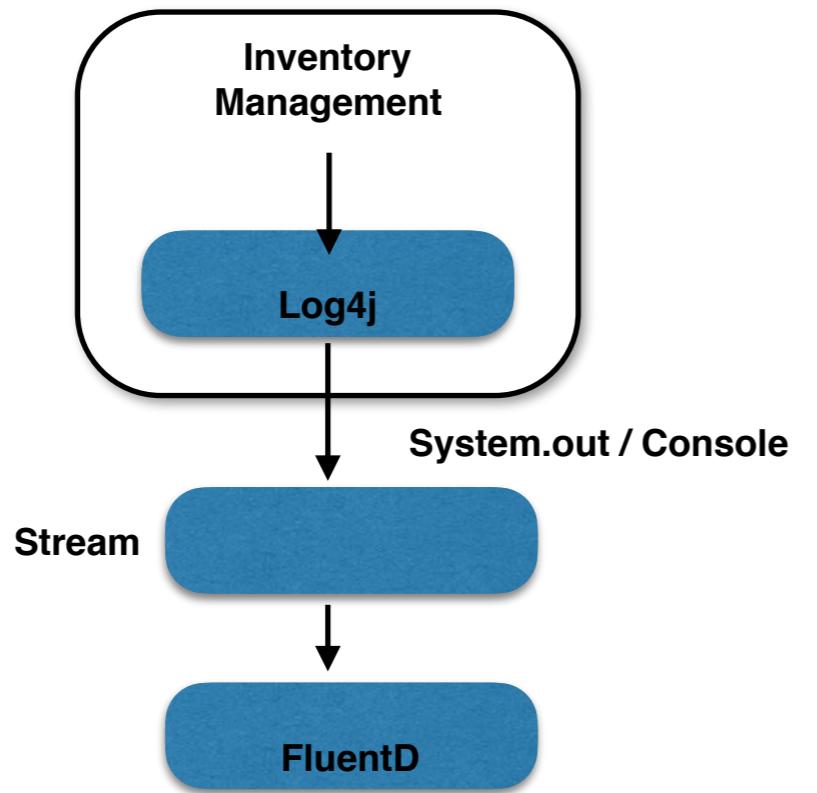
Elastic Search

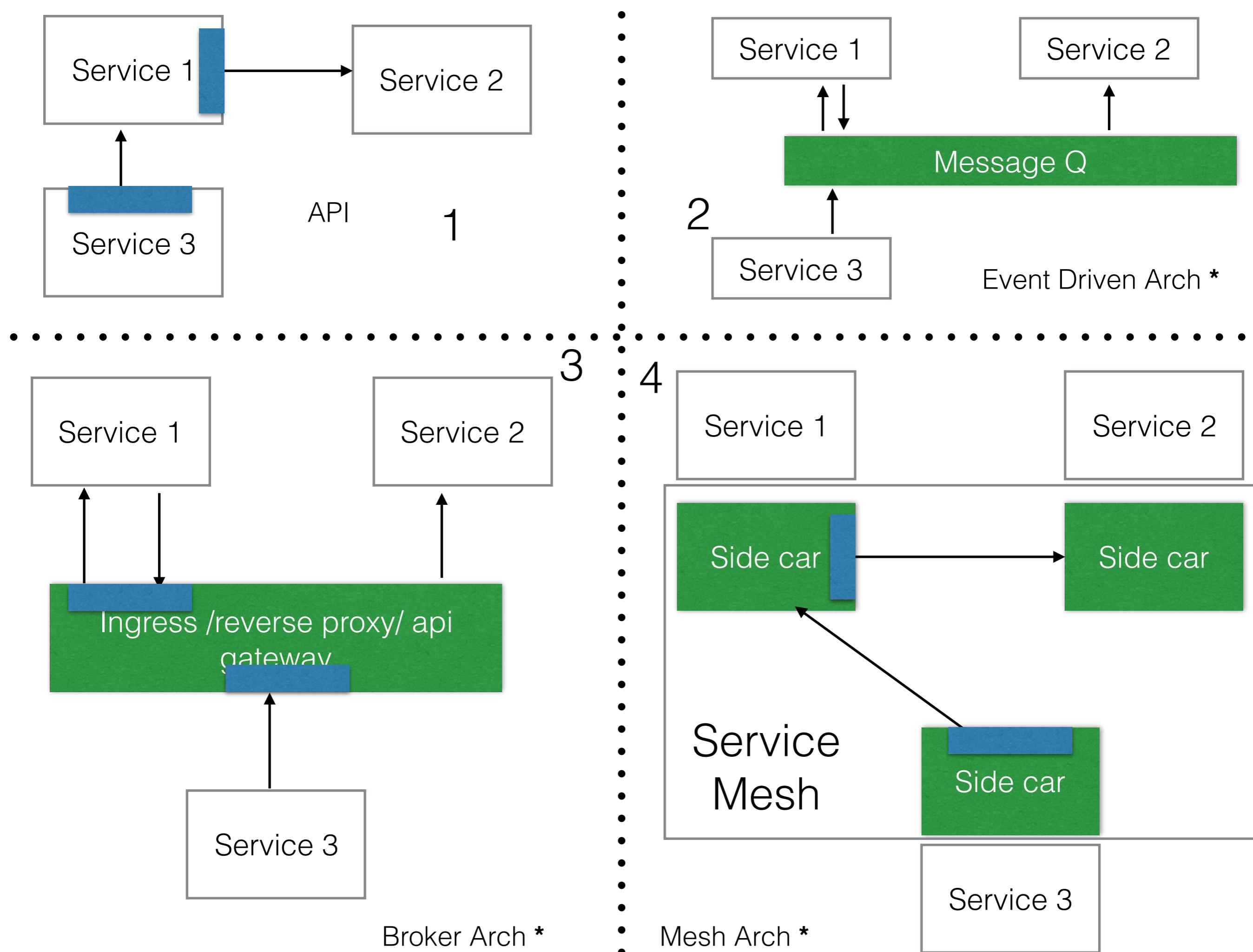
Fluentd

time series (metrics)

Prometheus

Service Mesh





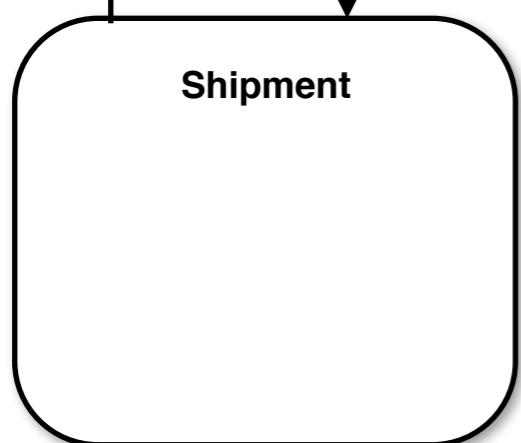
Intranet Eventual	Intranet Immediate	Internet
Msg + Service Mesh	API + Service Mesh	API gate/ Ingress/ Reverse Proxy

Portal



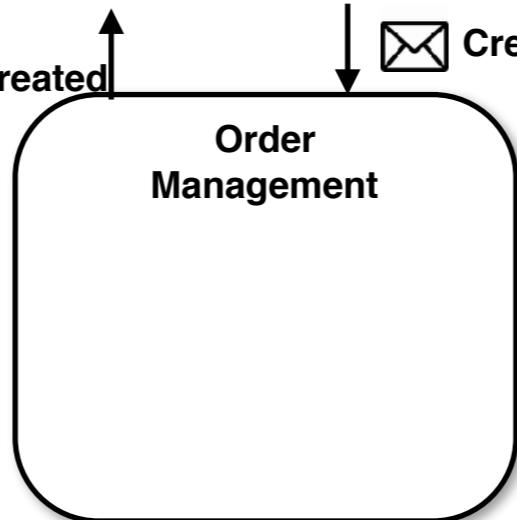
Shipment
Completed

OrderCreated

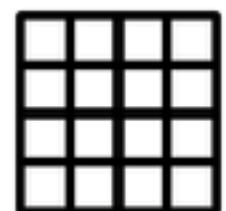


RDBMS

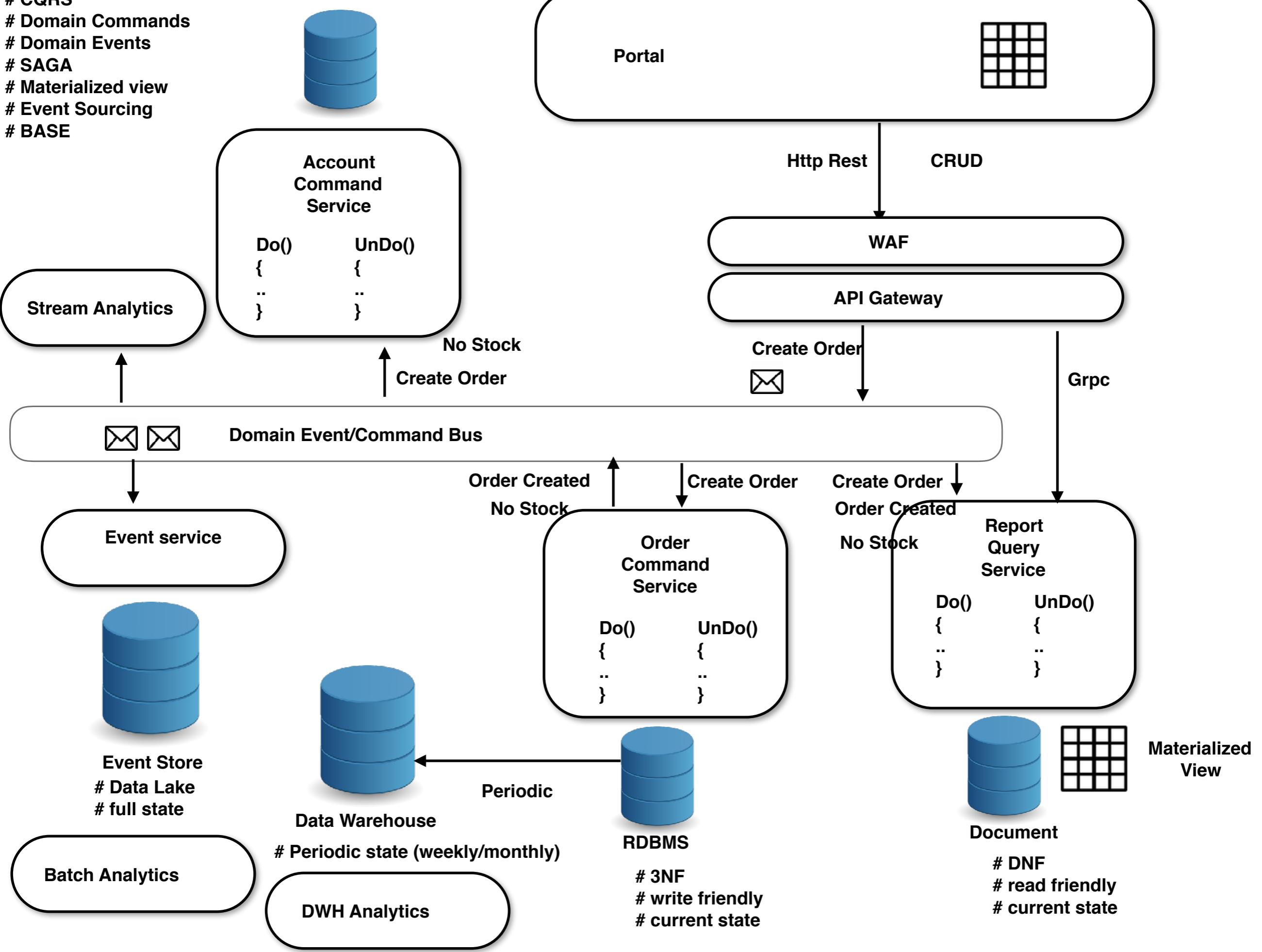
OrderCreated

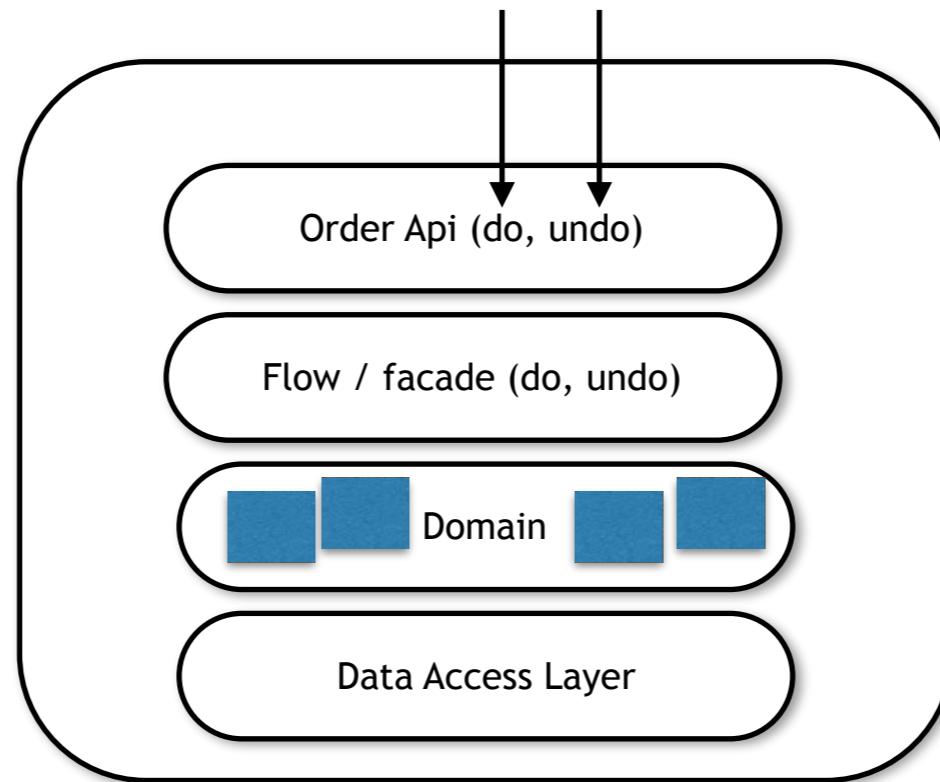


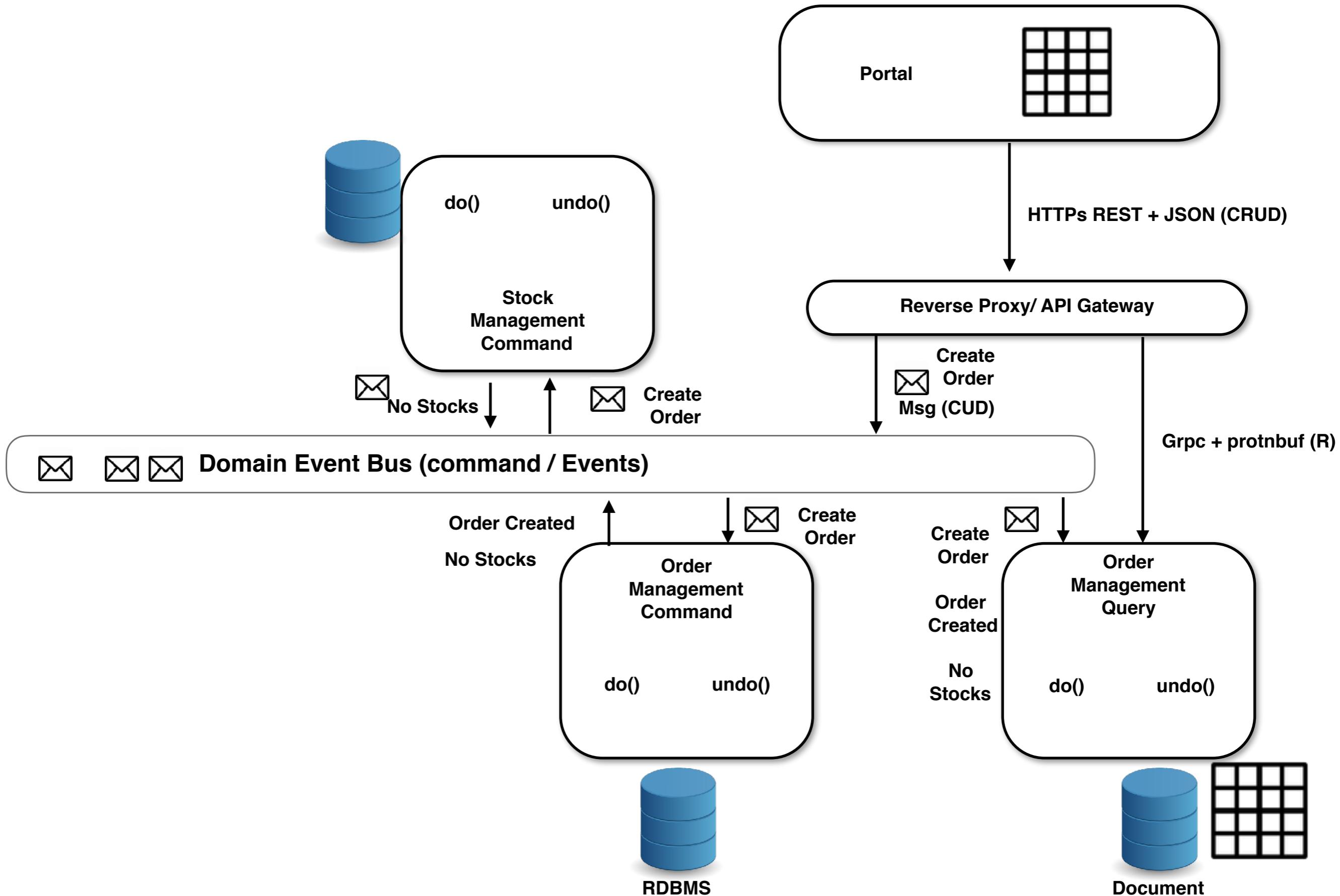
RDBMS



```
# CQRS  
# Domain Commands  
# Domain Events  
# SAGA  
# Materialized view  
# Event Sourcing  
# BASE
```







CQRS
 # Materialised View
 # SAGA - compensable transaction
 # Eventual Consistency
 # Event Driven Architecture
 # DDD
 # Bounded context
 # Aggregate
 # Event Sourcing
 # ubiquitous language
 # Command Message
 # Event Message

Log mngmt
 # config mngmt
 # devops



do() undo()

Stock Management Command

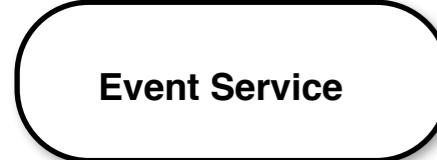
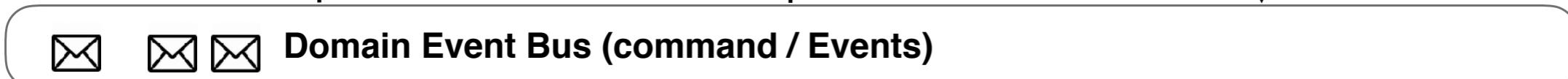
No Stocks Create Order

Authentication, Authorization

Reverse Proxy/ API Gateway

Create Order Msg (CUD)

Grpc + protobuf (R)

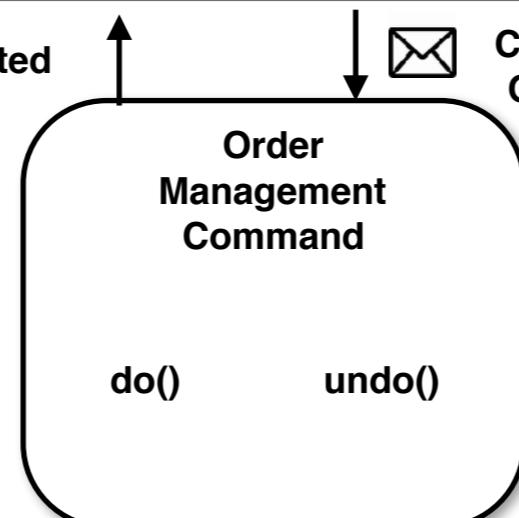


Data lake
Event Store
Audit
full state



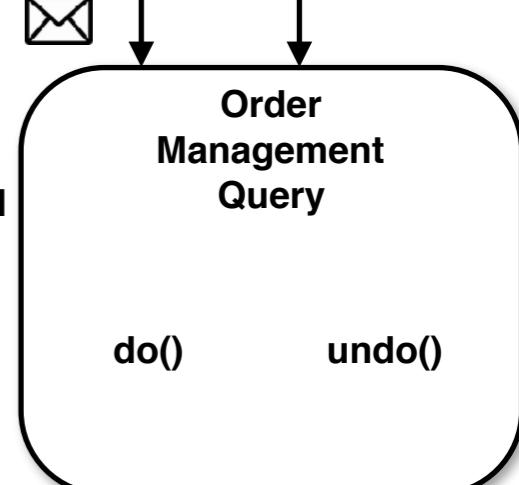
Data Ware House
snapshot (day/week)

Order Created
No Stocks



Create Order

Create Order
Order Created



No Stocks

.....
weekly/ monthly



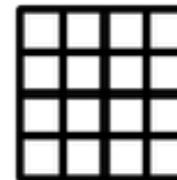
RDBMS

3NF
no duplicates, more joins
Write Friendly
not read friendly
current state



Document

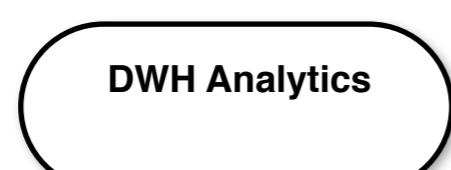
DNF
duplicates, no joins
not Write Friendly
read friendly

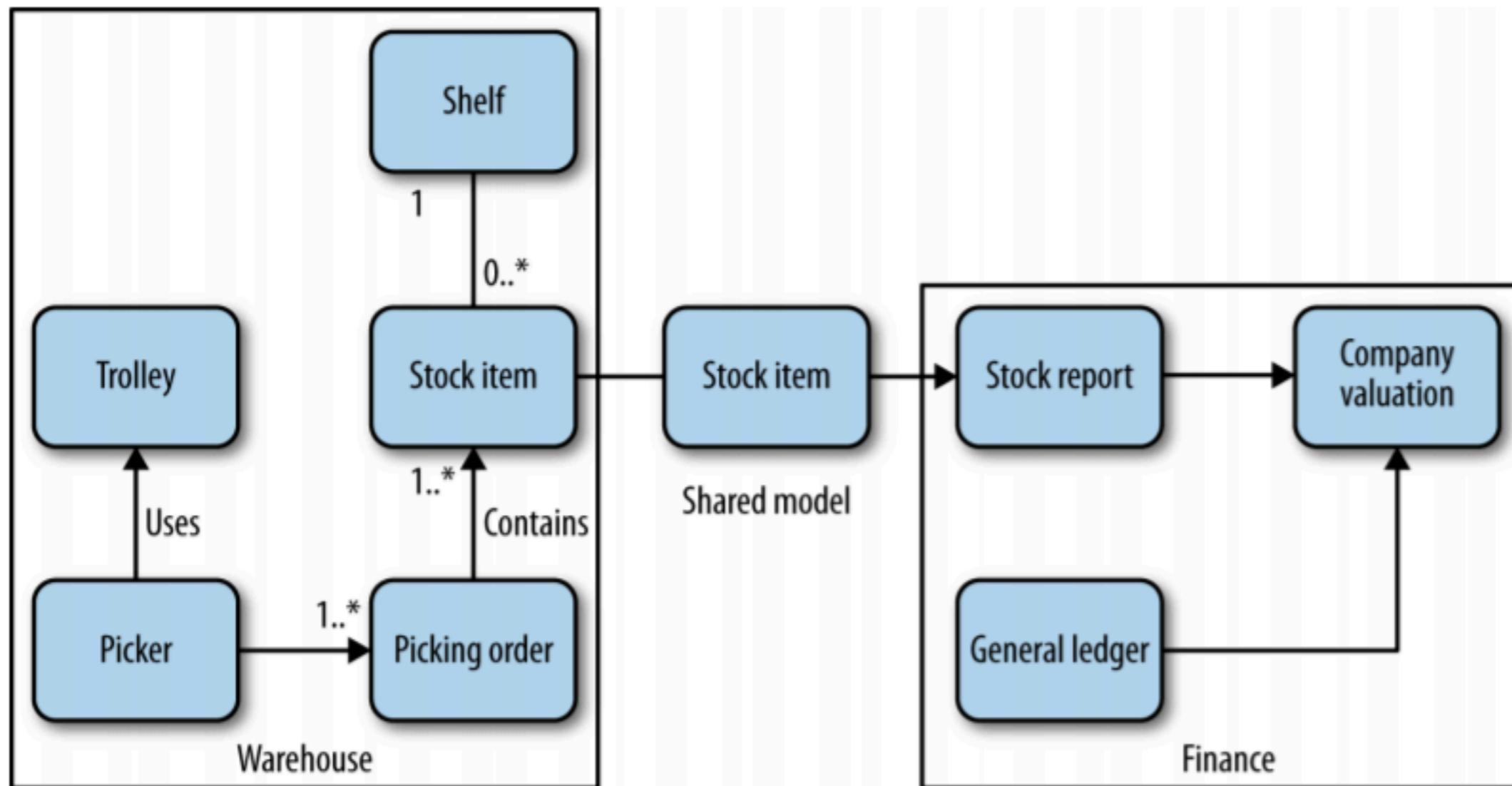


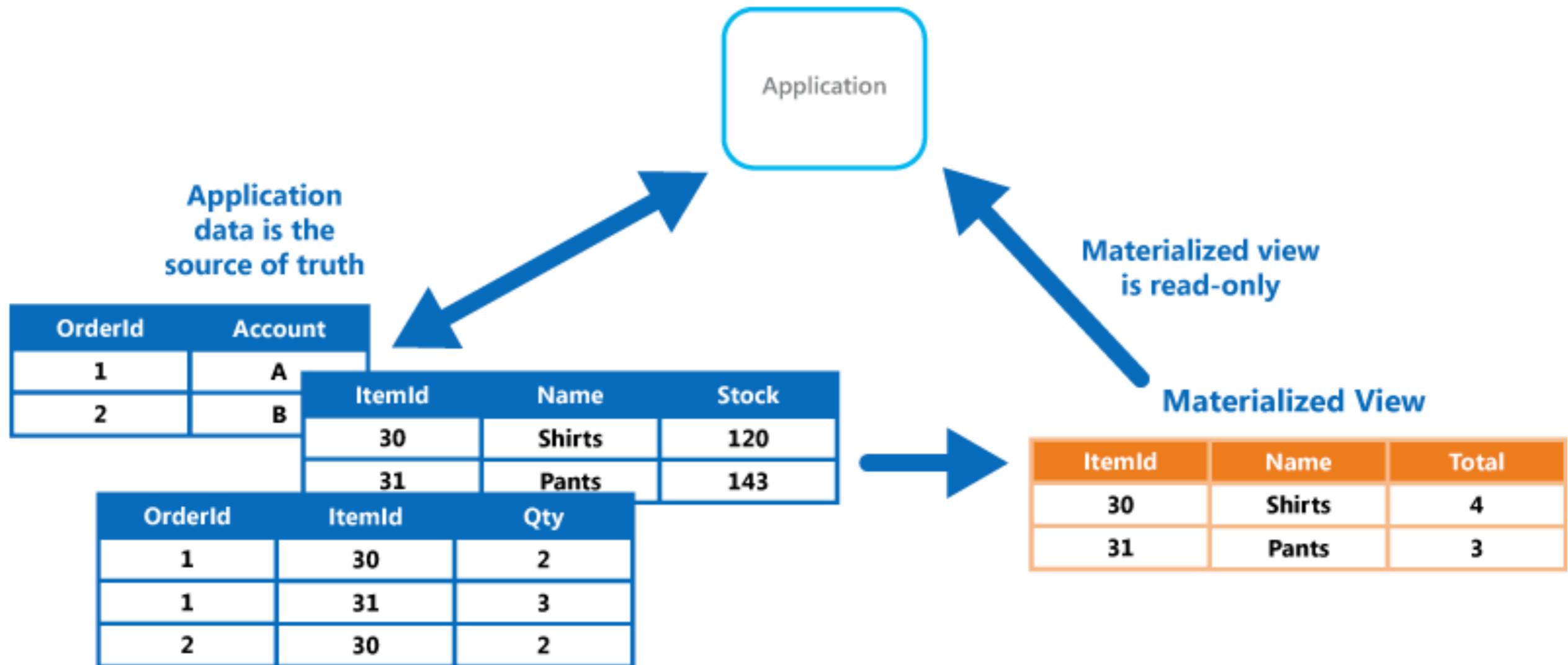
Portal

HTTPs REST + JSON (CRUD)

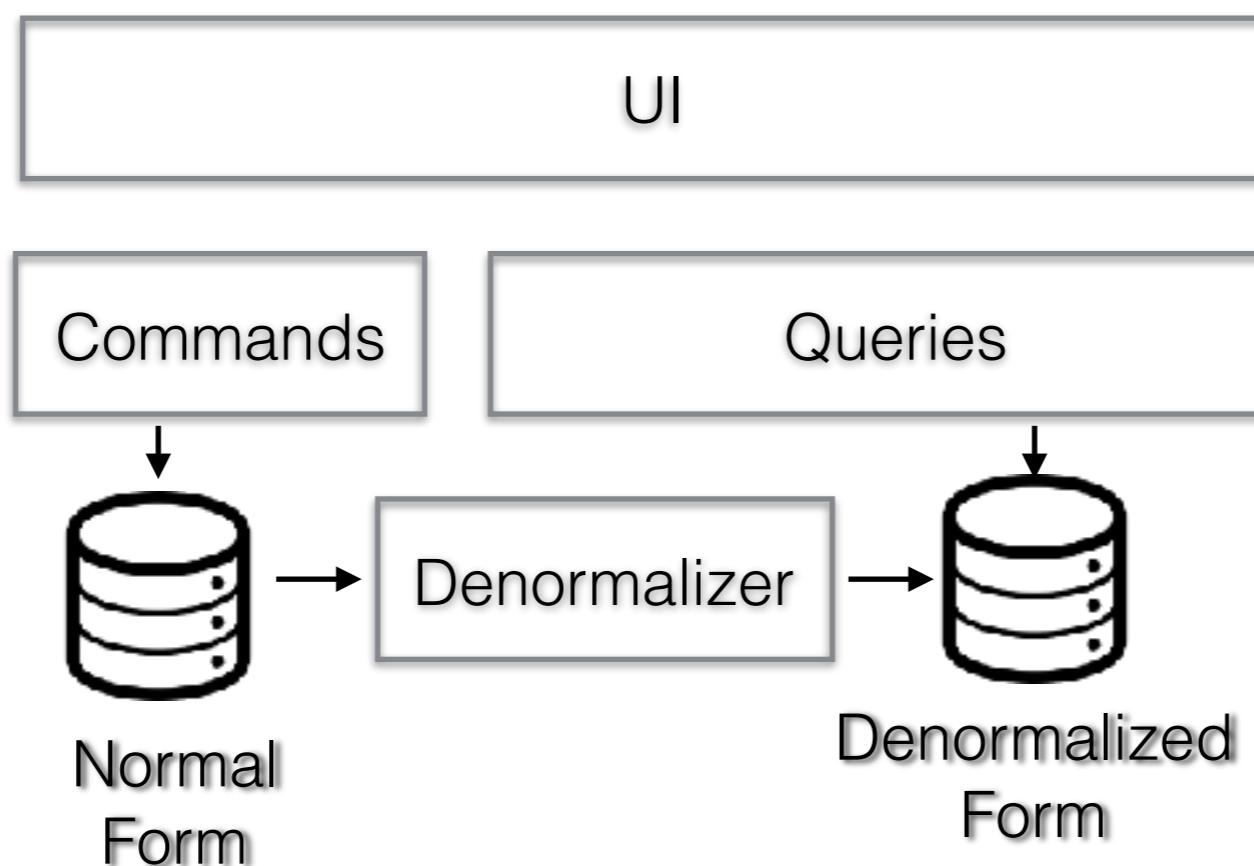
WAF



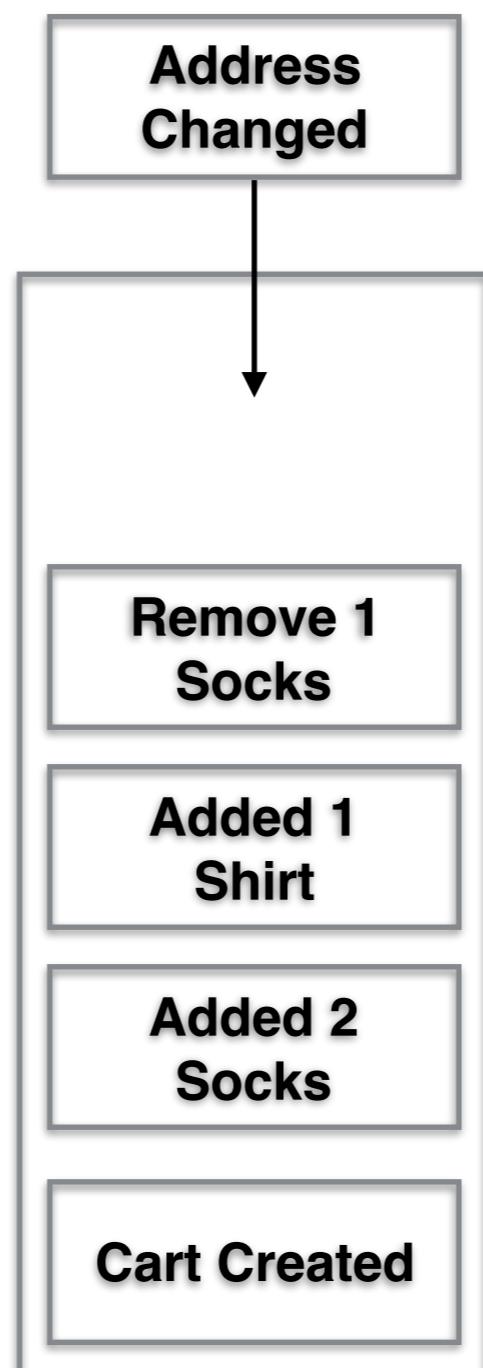




CQRS

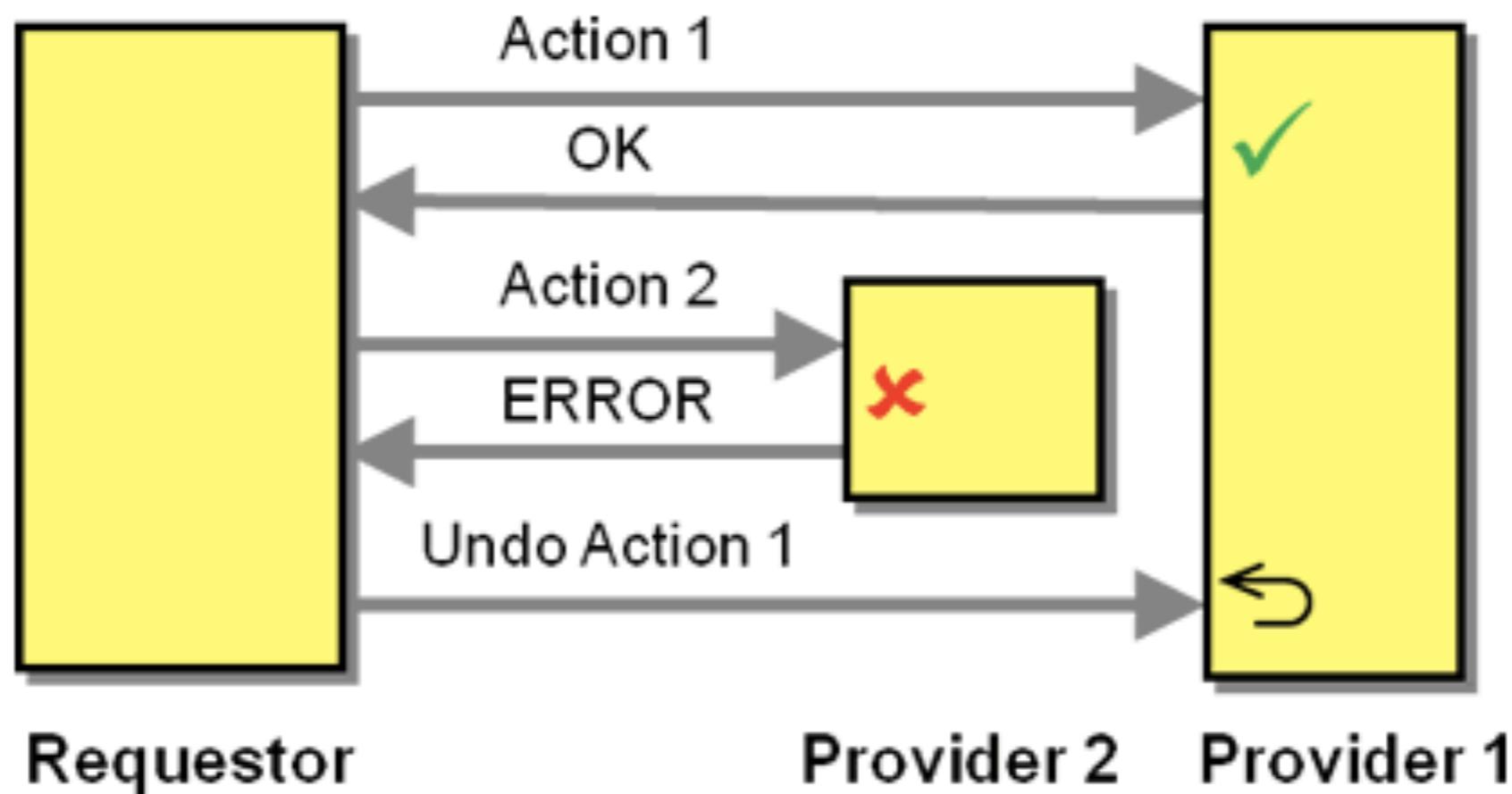


Event Source



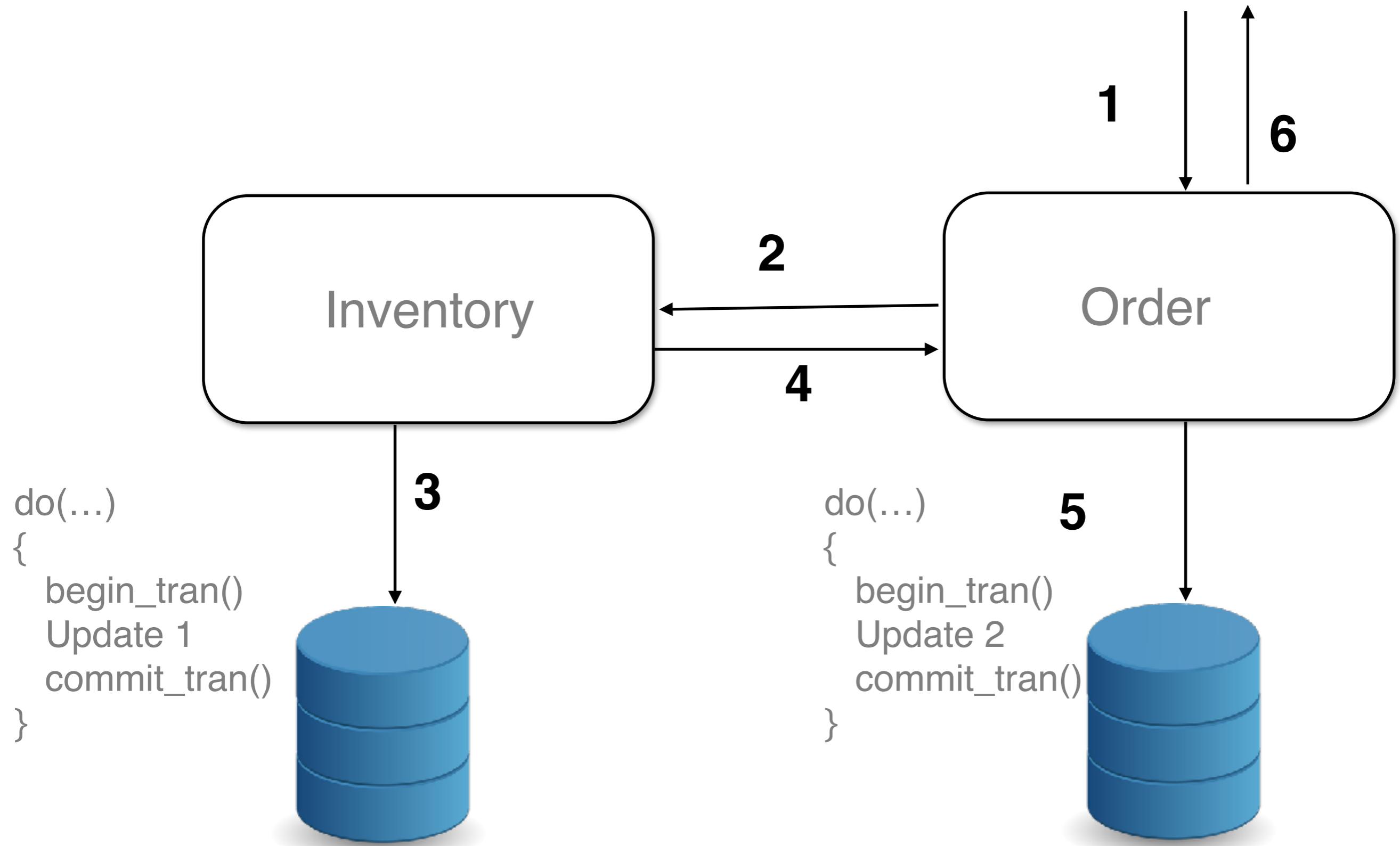
Event Store

Saga

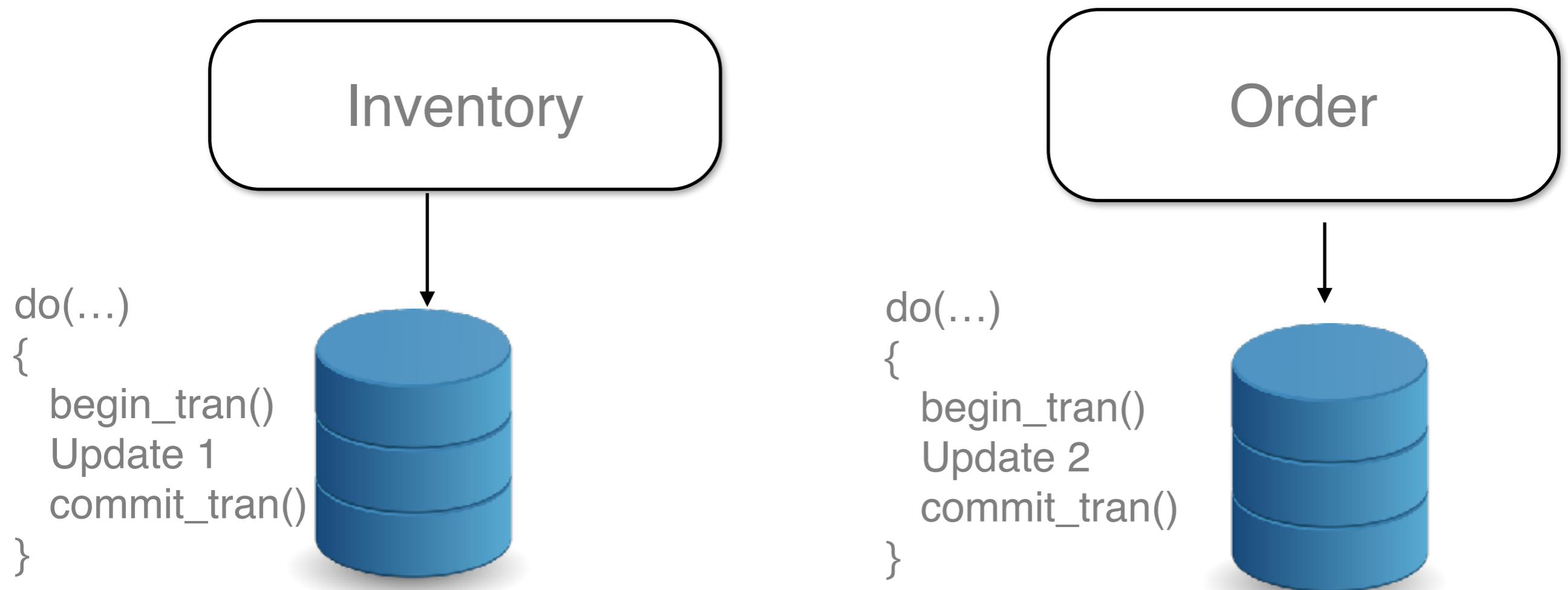


Saga Pattern is a direct solution to implementing distributed transactions in a microservices architecture.

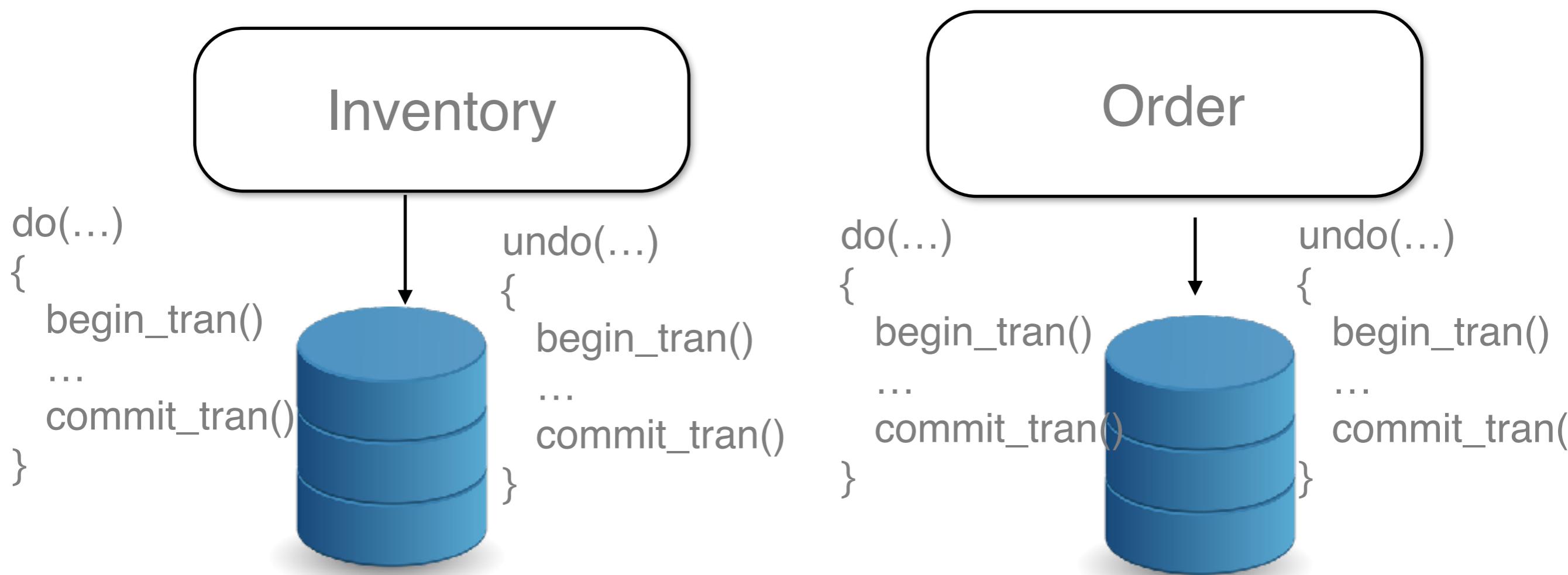
Transaction



SAGA pattern

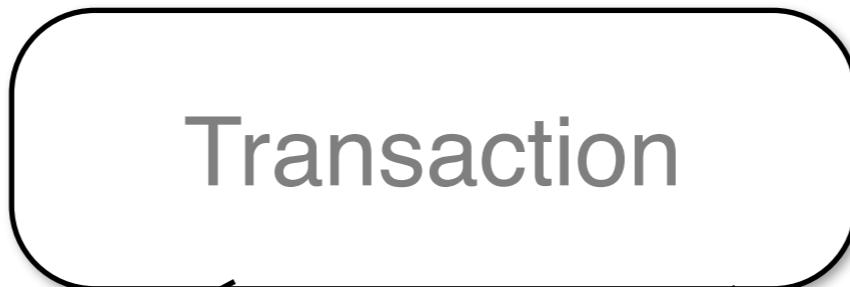


Compensatable Transaction

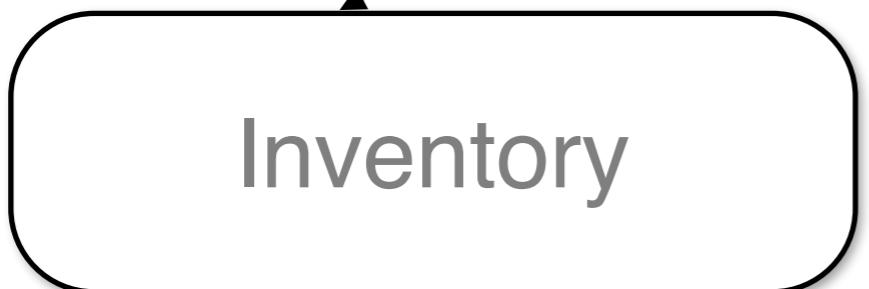




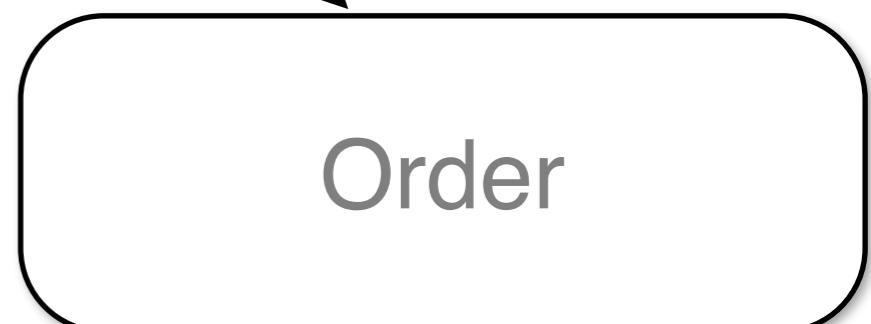
Tranid,order,inventory, ...
101, y, y,



2



5



```
do(...)  
{  
    begin_tran()  
    ...  
    commit_tran()  
}
```



3

```
undo(...)  
{  
    begin_tran()  
    ...  
    commit_tran()  
}
```

6

```
do(...)  
{  
    begin_tran()  
    ...  
    commit_tran()  
}
```



```
undo(...)  
{  
    begin_tran()  
    ...  
    commit_tran()  
}
```

ACID

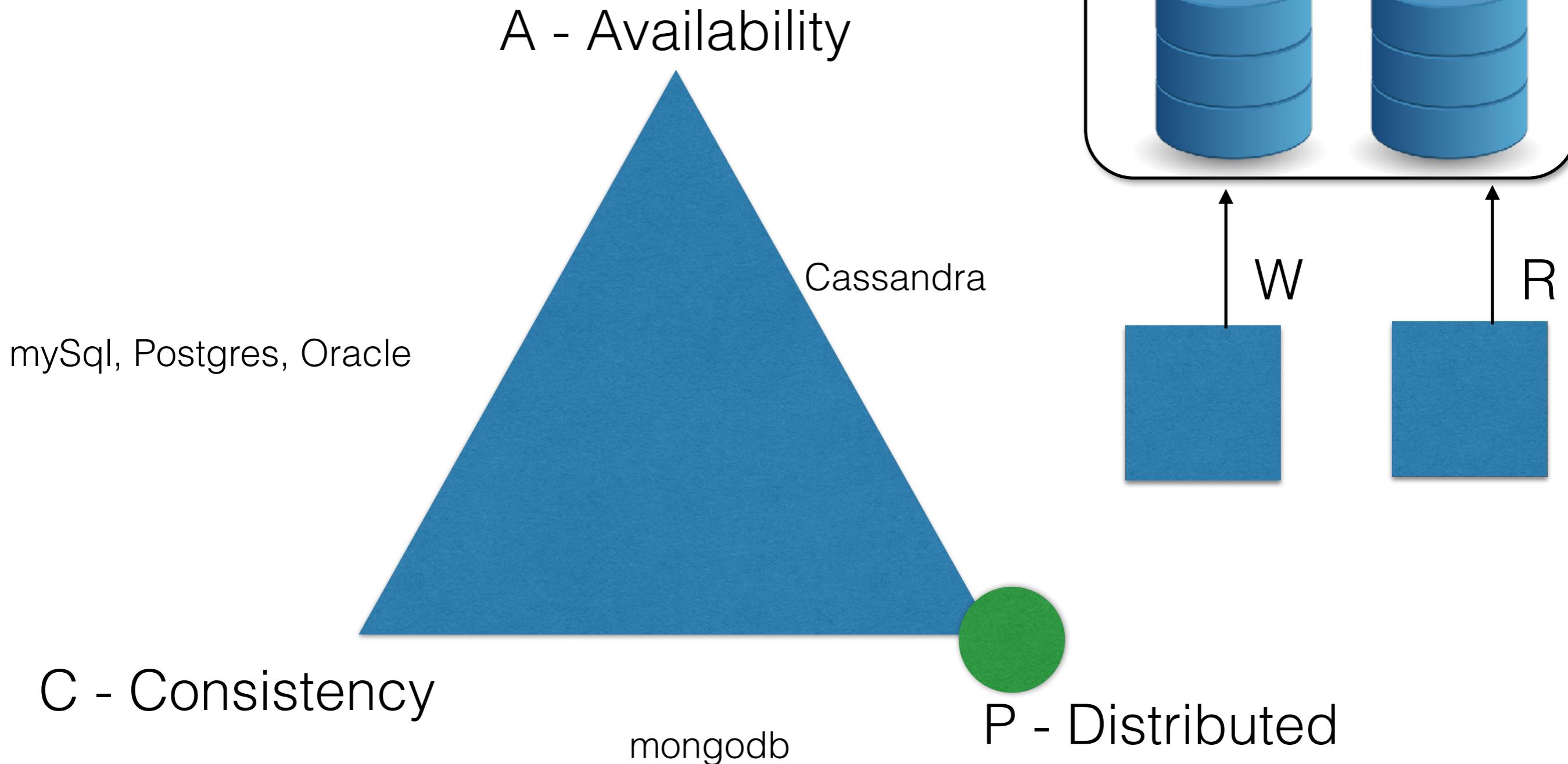
BASE

Immediate
Consistency

Eventual
Consistency

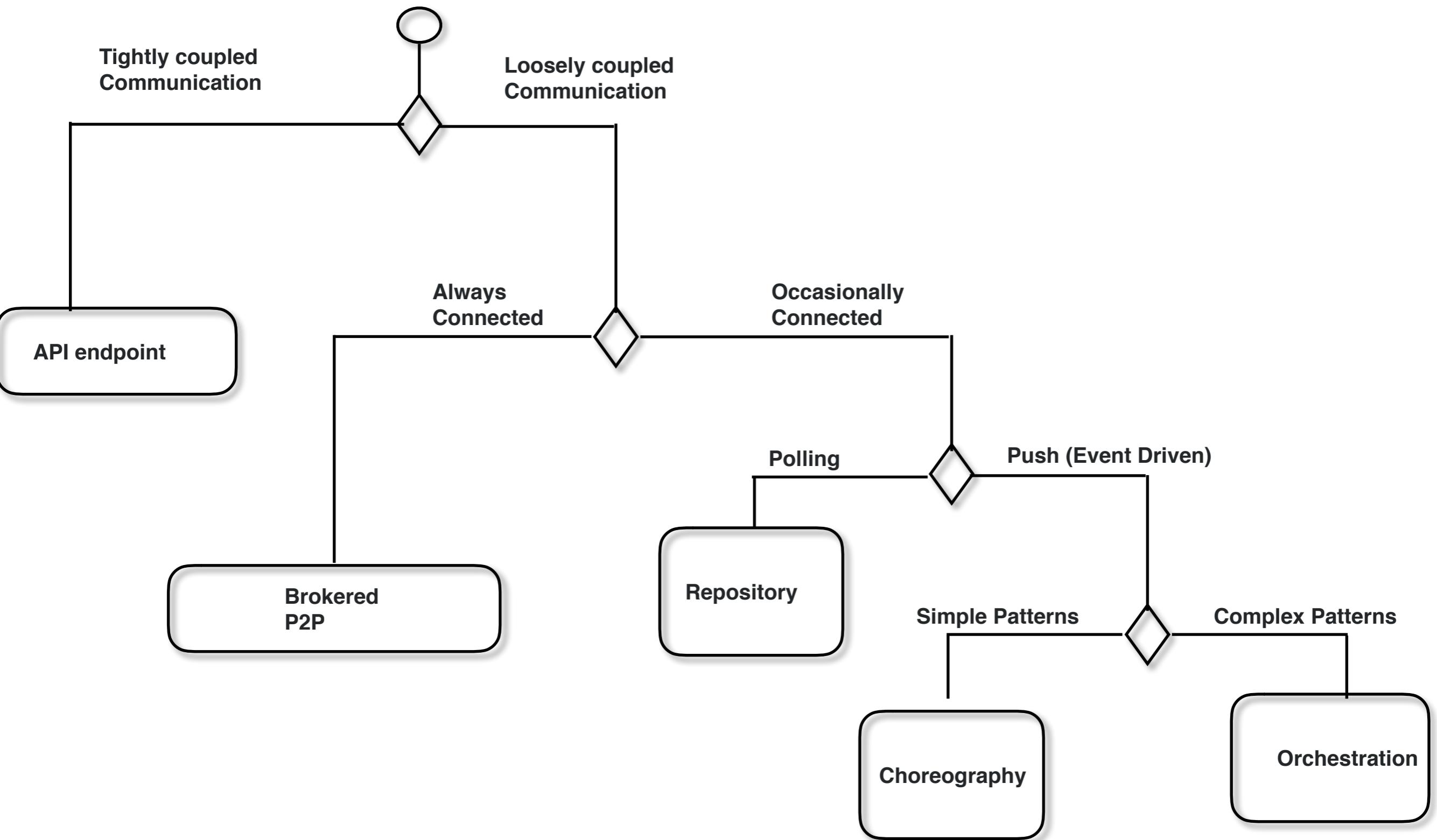


CAP Theorem

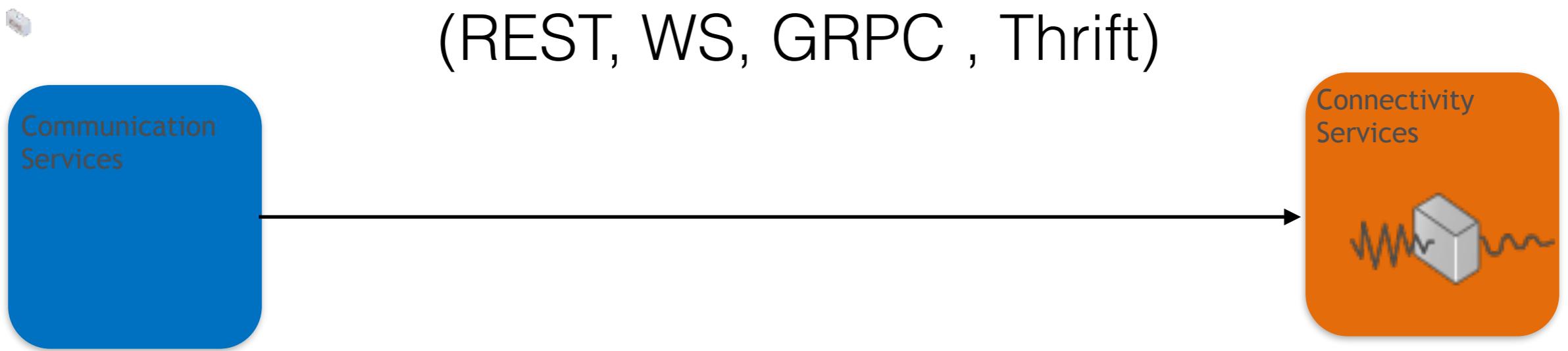




Choose Communication Mechanism

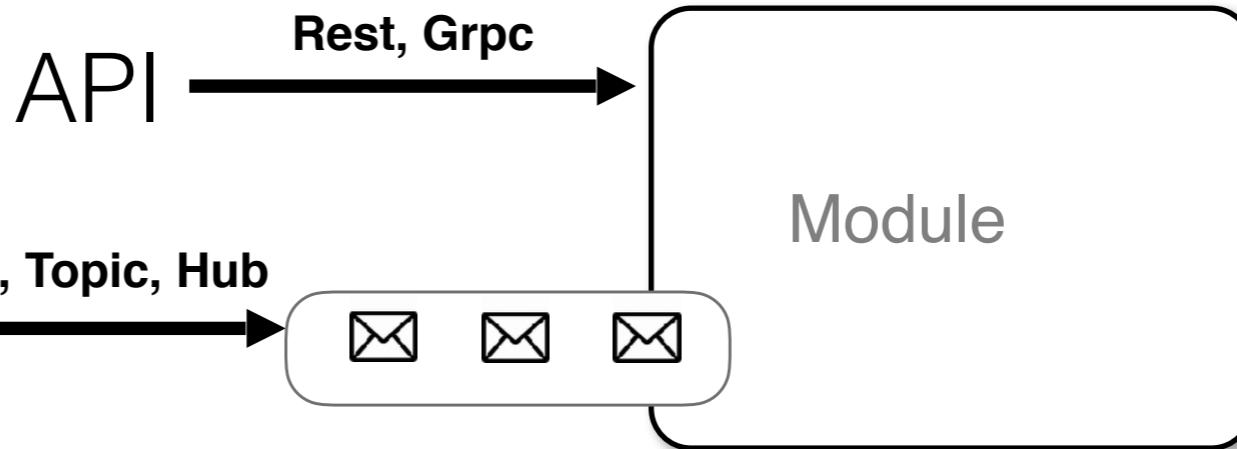


Connected protocol
(REST, WS, GRPC , Thrift)

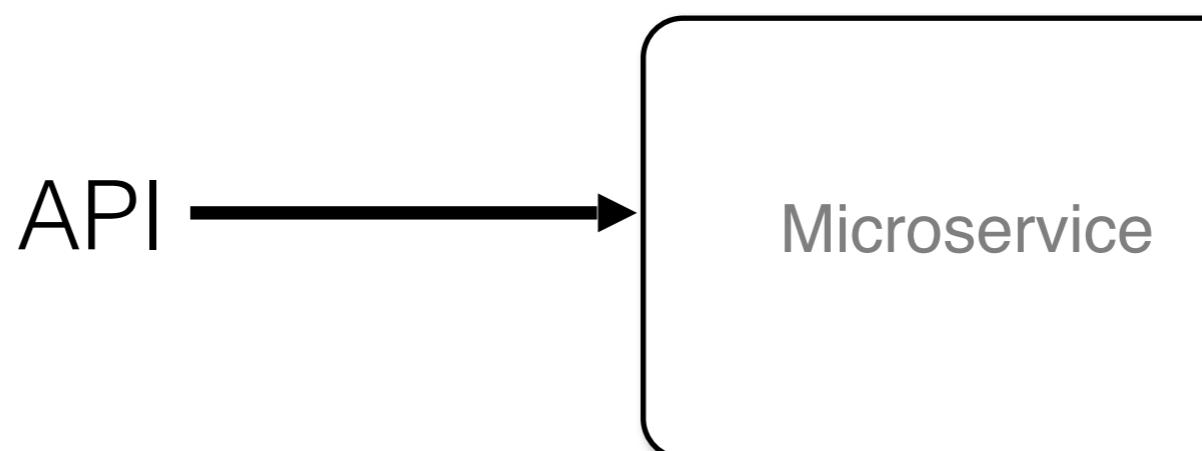


Message protocol
(AMQP, MQTT, ...)

* Message



	<< API >>	<< Message >>	
Ordering	Ordered (+)	Unordered(-)	
Duplicate	Yes (-)	Yes(-)	Idempotency
Protocol	2 way (+)	One way (-)	
Resilience (recover)	No (-) retry logic	Yes (+)	
Connection	Always connected	Occousionaly connected	
Scalability	--	+++	
Consistency	Immediate (++)	Eventual (- -)	
Load Leveling	--	++	
Low Coupling (maintainability)	--	++	
Distributed Comm Patterns	--	++	SAGA, Materialized View
Internet	Yes	No	
Browser support	Yes	No	
Dev effort	++	--	
Operational effort	++	--	



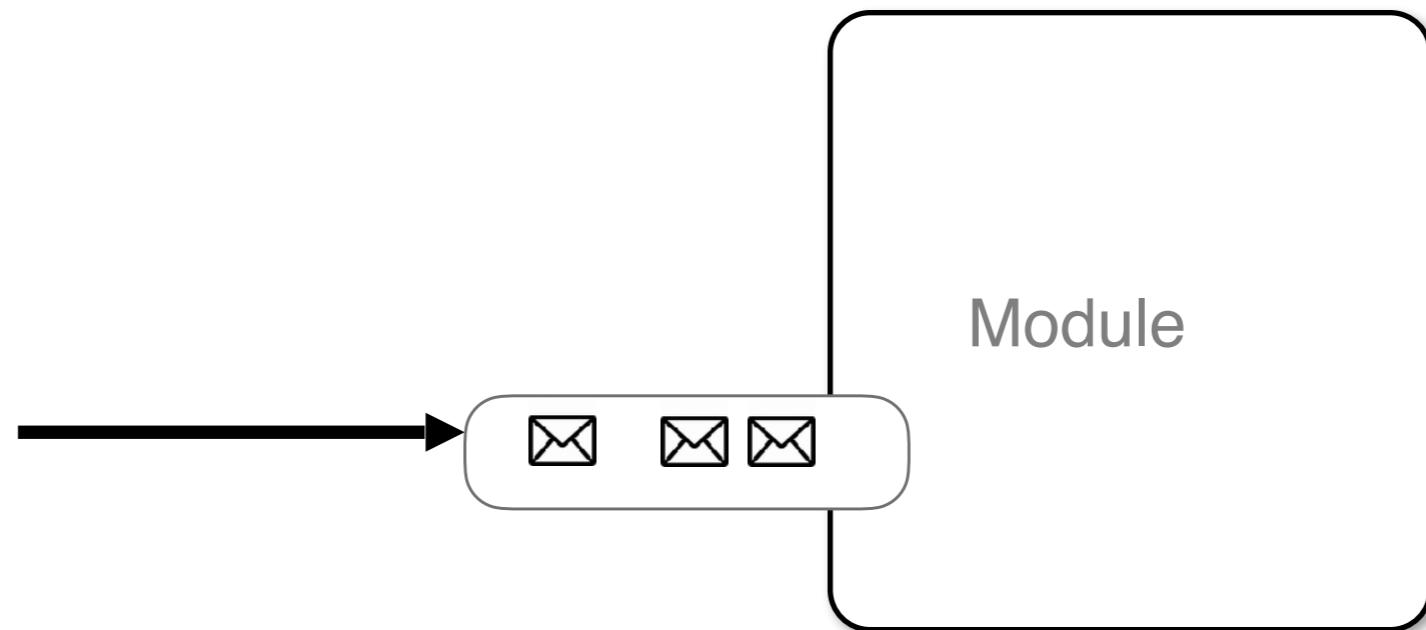
The diagram illustrates the relationship between an API and a Microservice, and how the Microservice can be implemented using different protocols.

The API points to the Microservice, which is represented by a rounded rectangle. From the Microservice, three arrows point to the right, labeled REST, WSS, and GRPC, representing different implementation choices.

The table below provides a comparison of these three protocols based on various criteria:

	REST	WSS	GRPC
Browser Support	Y	Y	N
Format	TEXT	TEXT	BINARY, HTTP2
Serialization	JSON	JSON	Proto Buf
Performance	--	+	++
Use case	North South	Streaming	East West

* Message

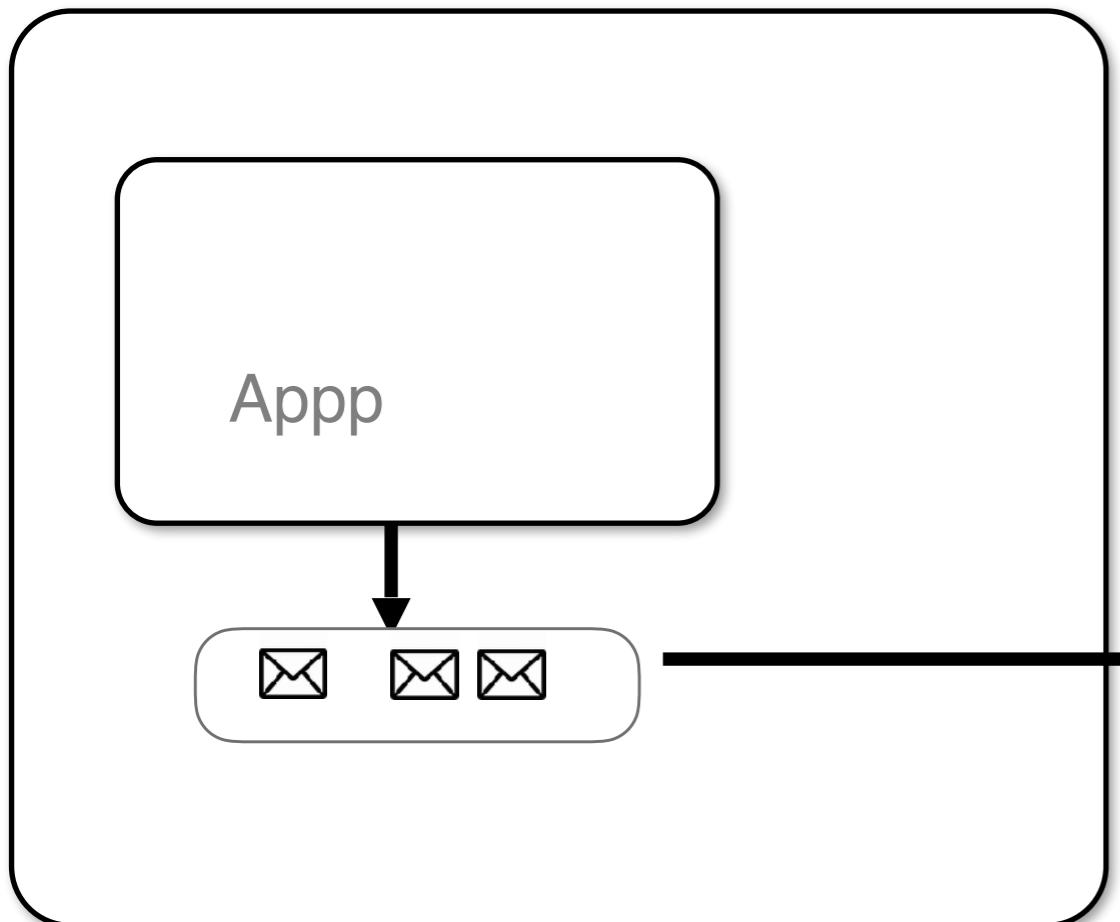


```
fun(){  
    while(true){  
        Msg m = GetMessage();  
        ....  
        m.ack();  
    }  
}
```

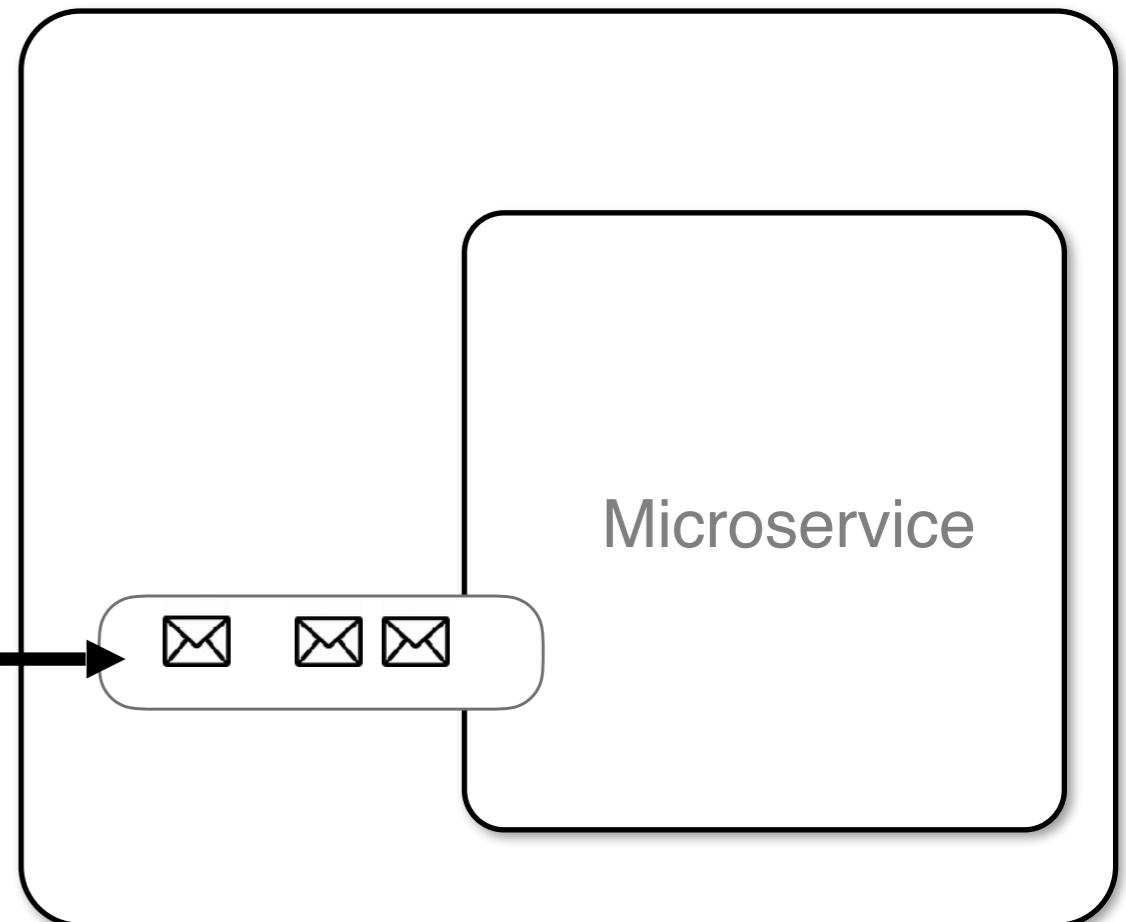
MSMQ

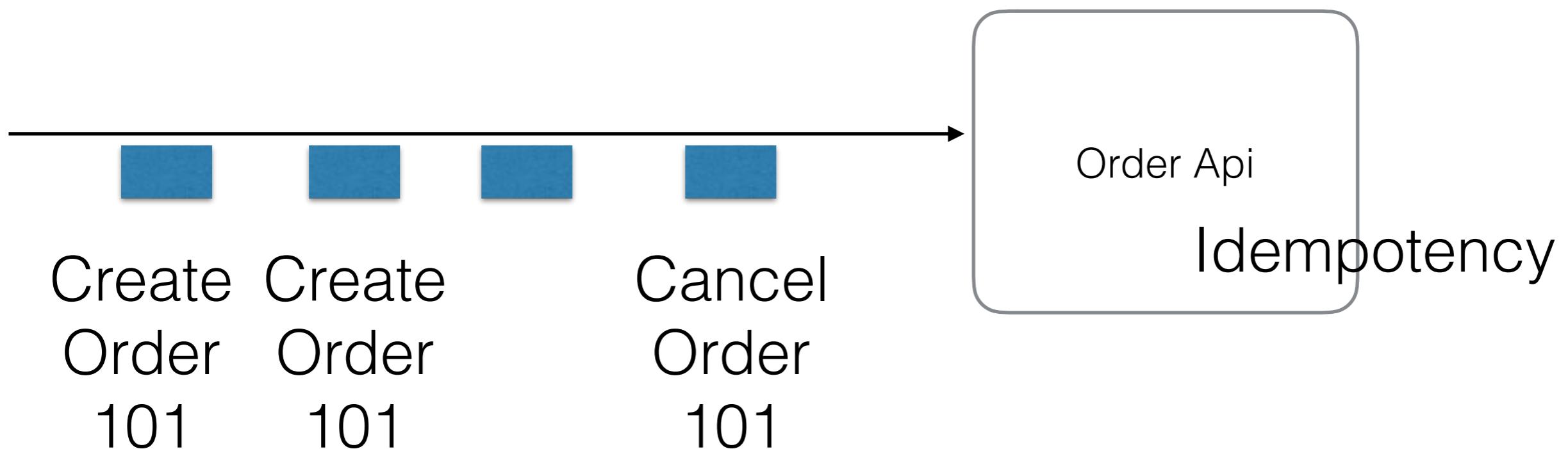
* Message

Client



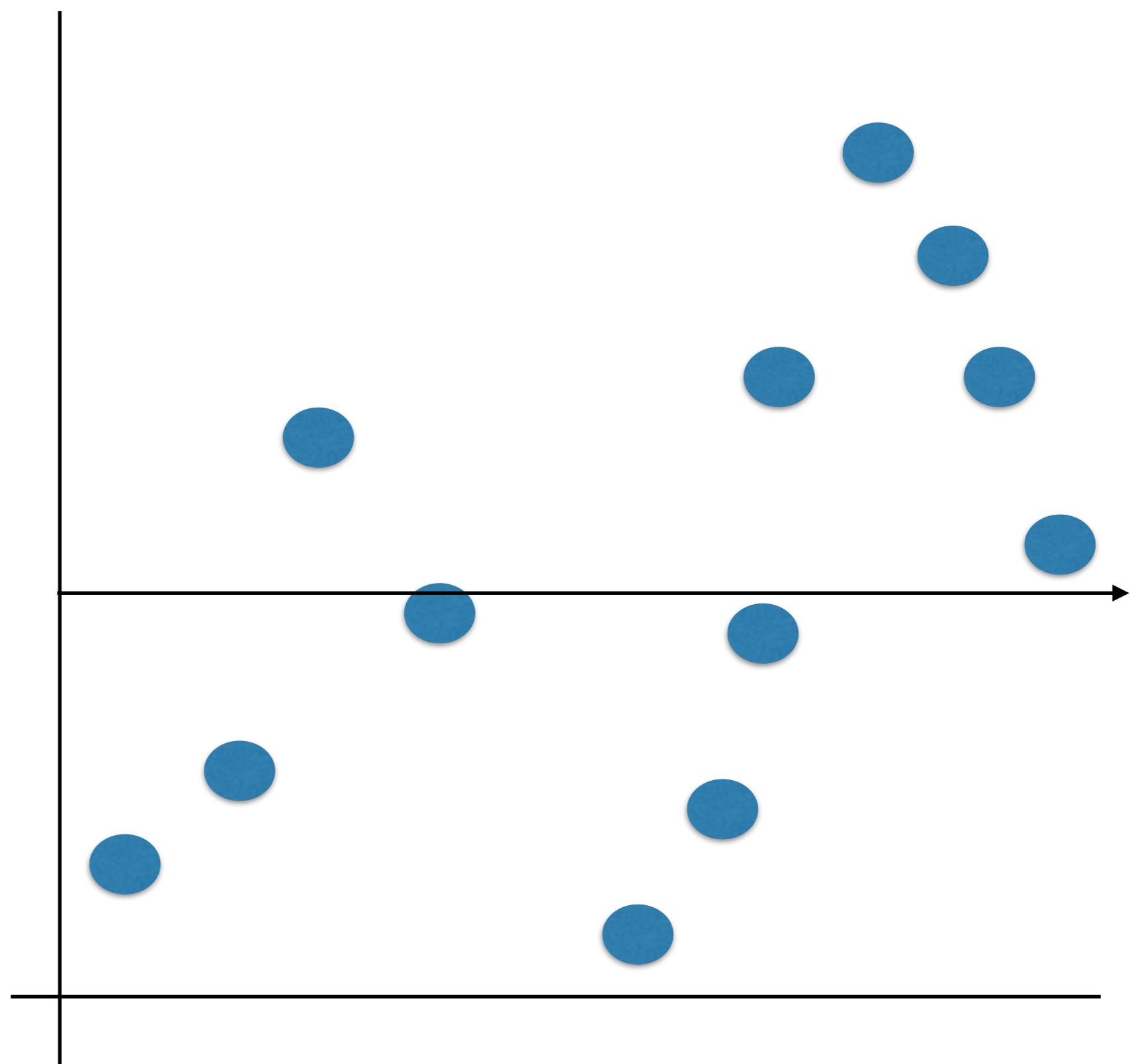
Server



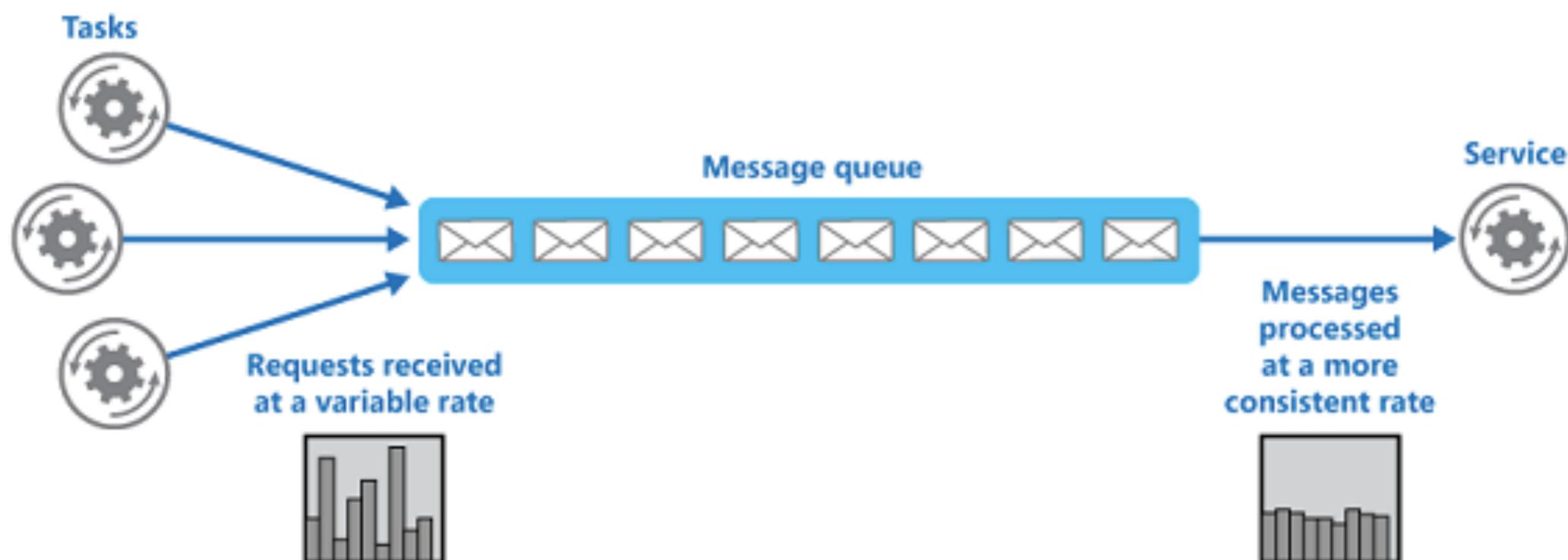


Request

Time



Queue-based Load Levelling



source:msdn

Event vs Command

Post

Pre

OrderCreated



CreateOrder

InvoiceCreated



OrderCanceled



OrderCreated



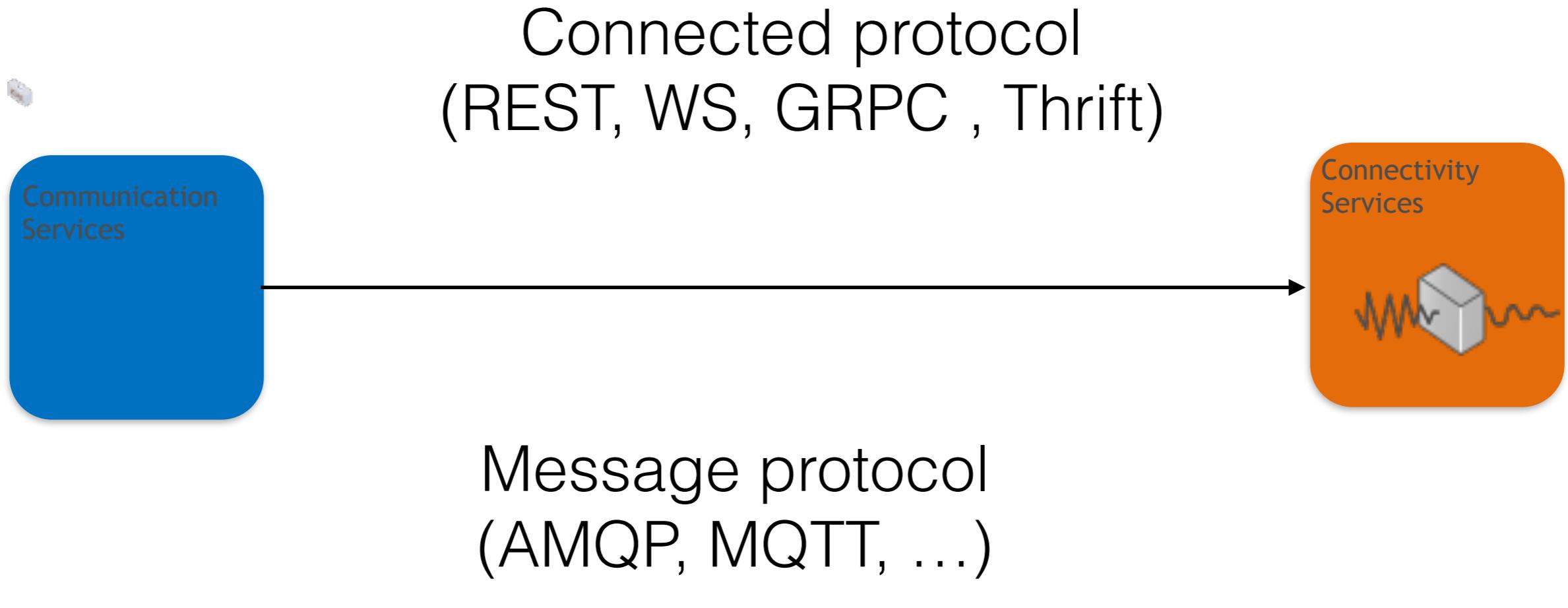
OderCanceled



OrderCreated



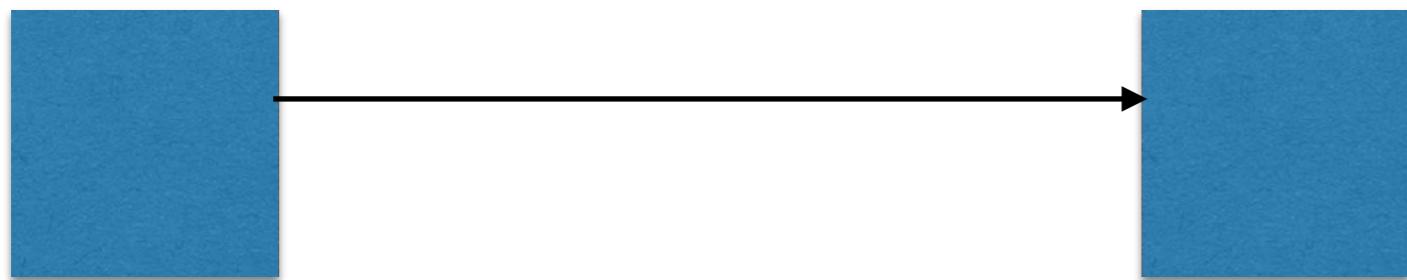
Request-reply, immediate



One-way, eventual, scale

```
graph LR; A[ ] --> B[ ]
```

Storage



Old val : 100
Val = 100 - 20
Lock (transaction)
if(cur_val == old_val)
 Update to val
Unlock (commit)

Kitkat : 20

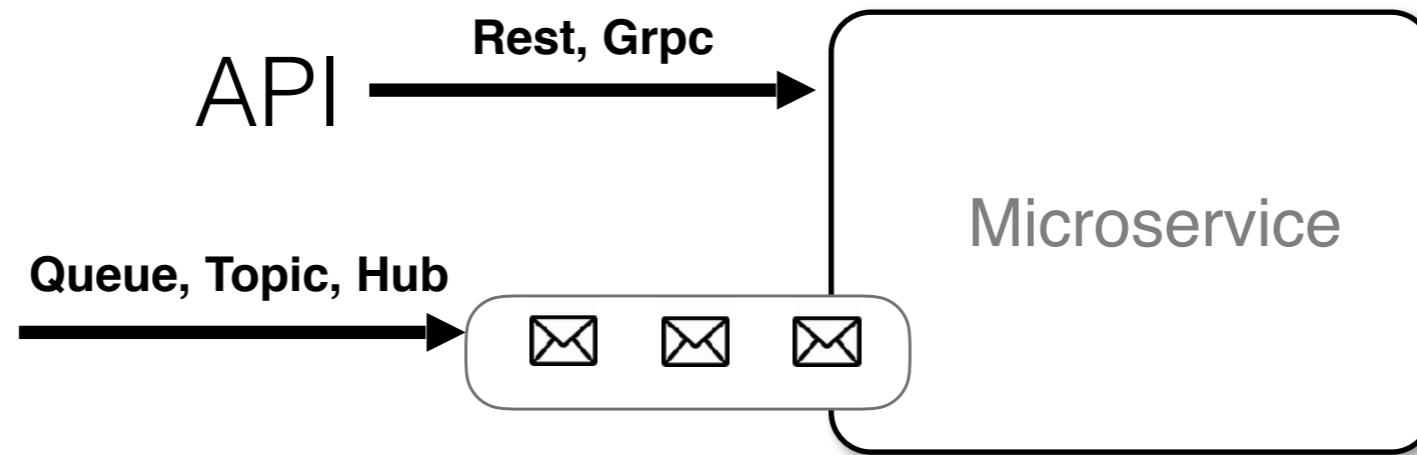
Old val : 100
Val = 100 - 20
Lock (transaction)
if(cur_val == old_val)
 Update to val
Unlock (commit)

Kitkat : 100

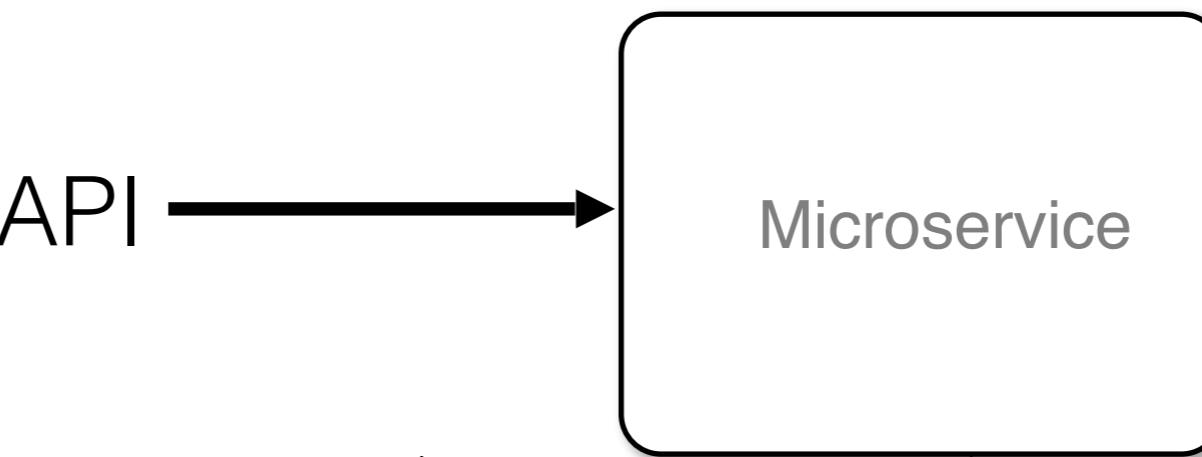
Kitkat : 50

Lock (transaction)
Old val : 100
Val = 100 - 20
Update to val
Unlock (commit)

* Message



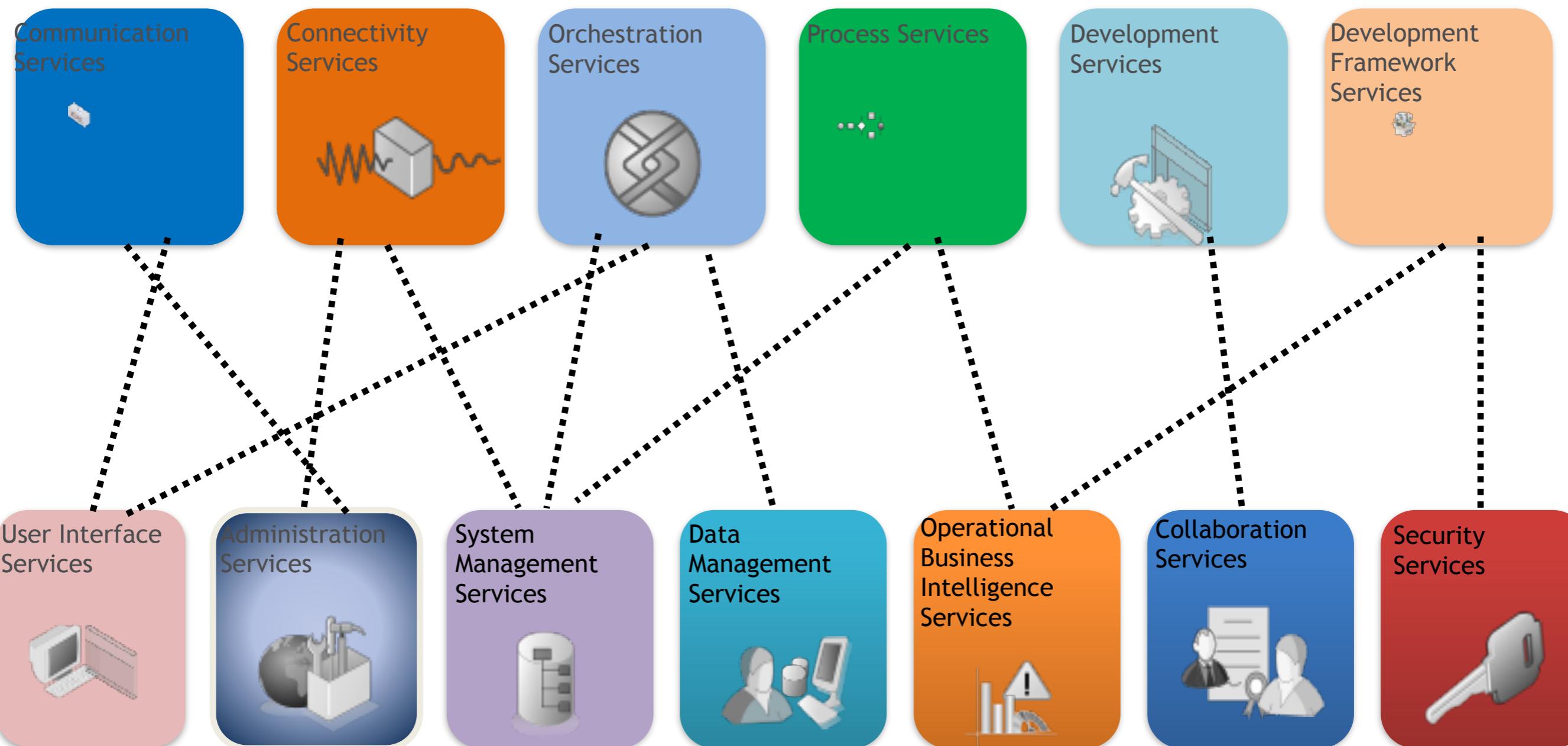
	<< API >>	<< Message >>	
Ordering	Ordered (+)	Unordered(-)	
Duplicate	Yes (-)	Yes(-)	Idempotency
Protocol	2 way (+)	One way (-)	
Resilience (recover)	No (-)	Yes (+)	
Connection	Always connected	Occousionaly connected	
Scalability	--	+++	
Consistency	Immediate (++)	Eventual (- -)	
Load Leveling	--	++	
Low Coupling	--	++	
Distributed Comm Patterns	--	++	SAGA, Materialized View
Internet	Yes	No	
Browser support	Yes	No	
Dev effort	++	--	

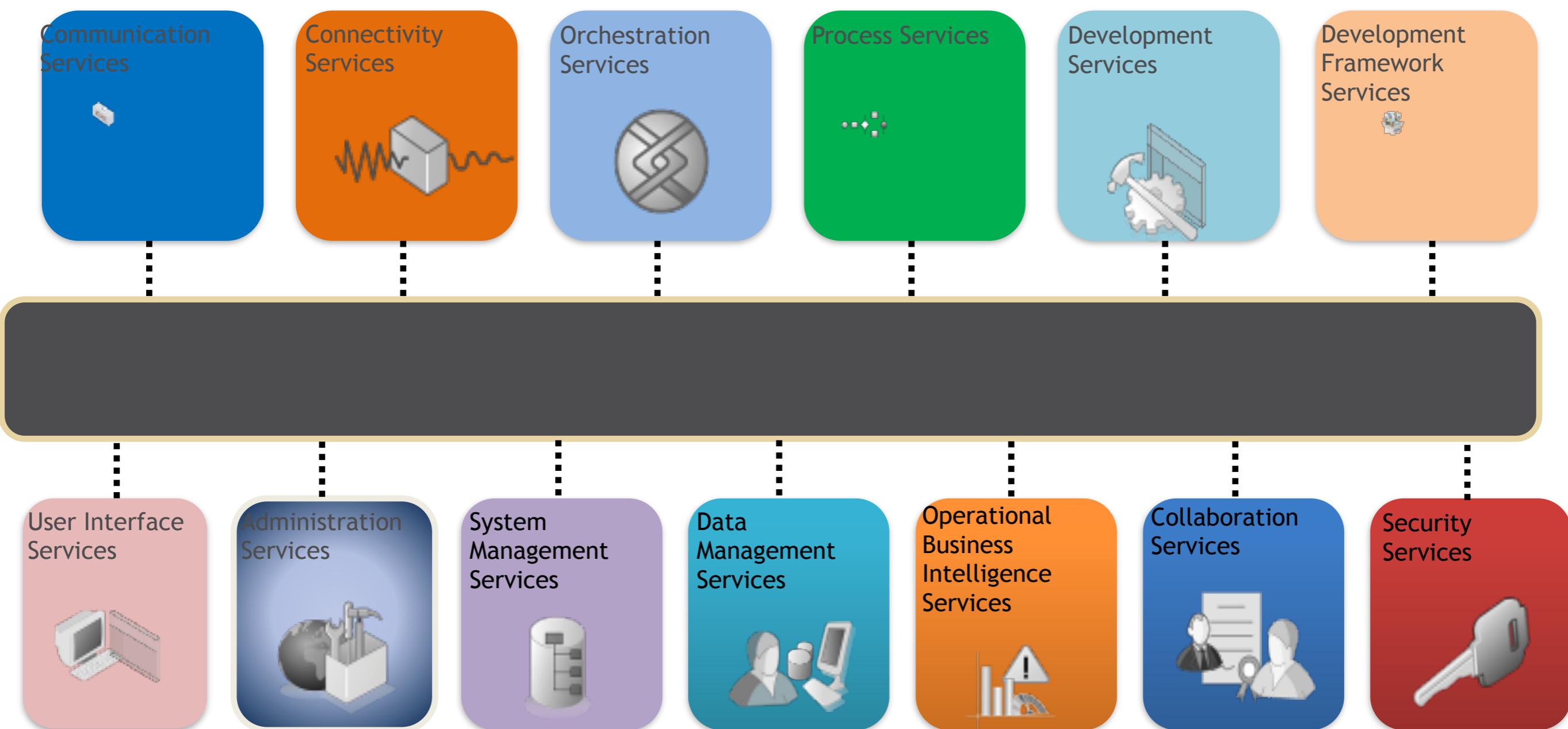


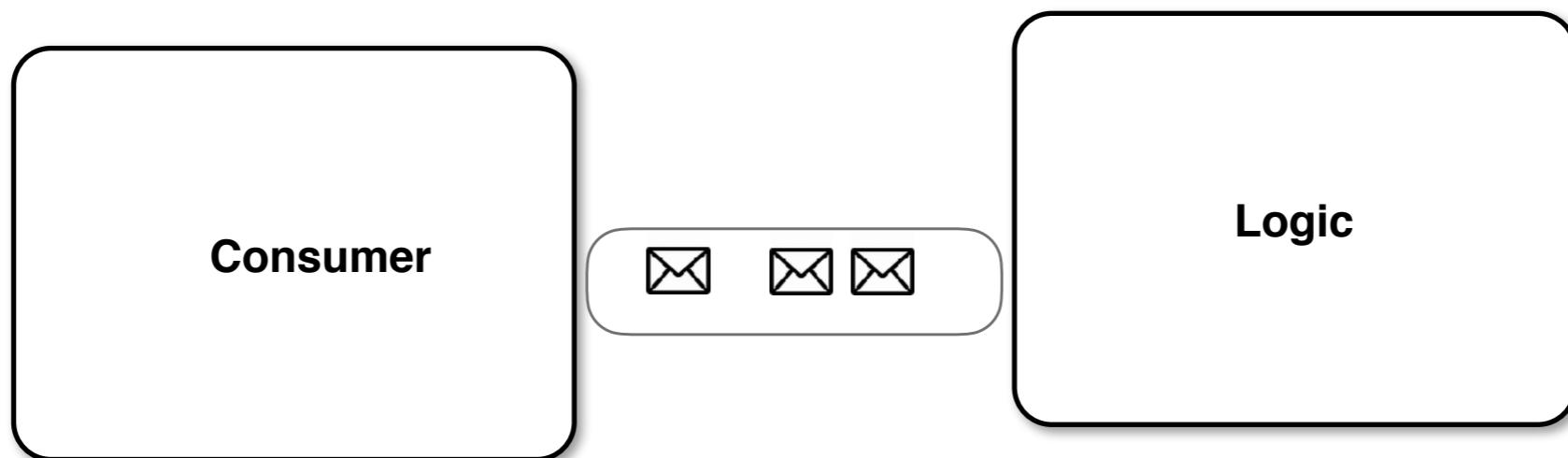
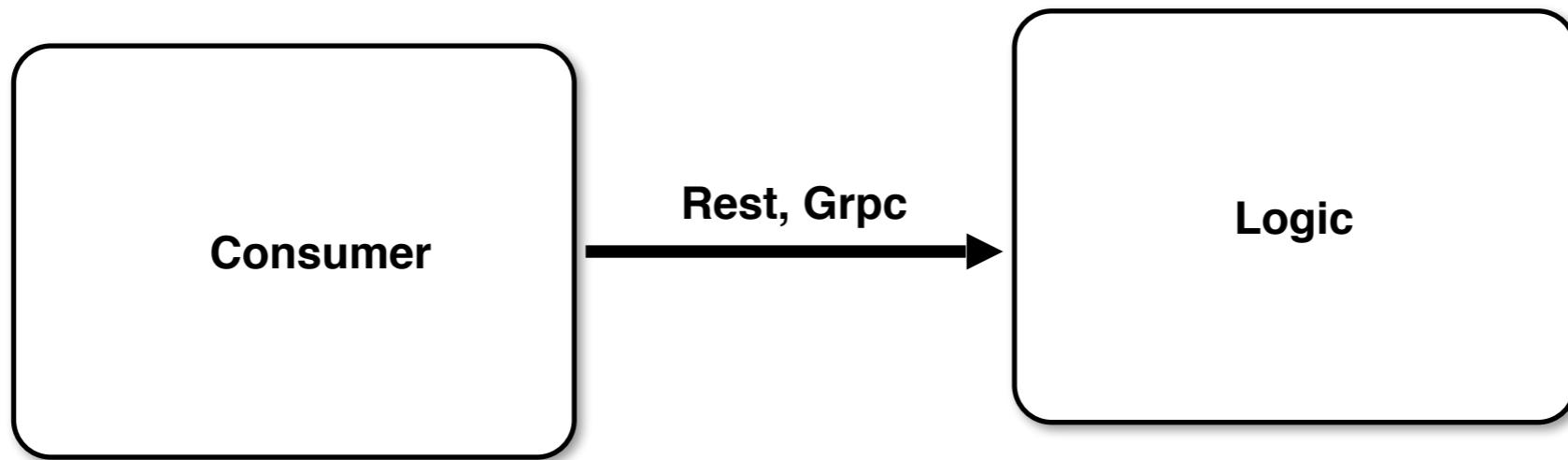
The diagram illustrates the relationship between an API and a Microservice, with the Microservice further divided into three communication protocols: REST, WSS, and GRPC.

API → **Microservice**

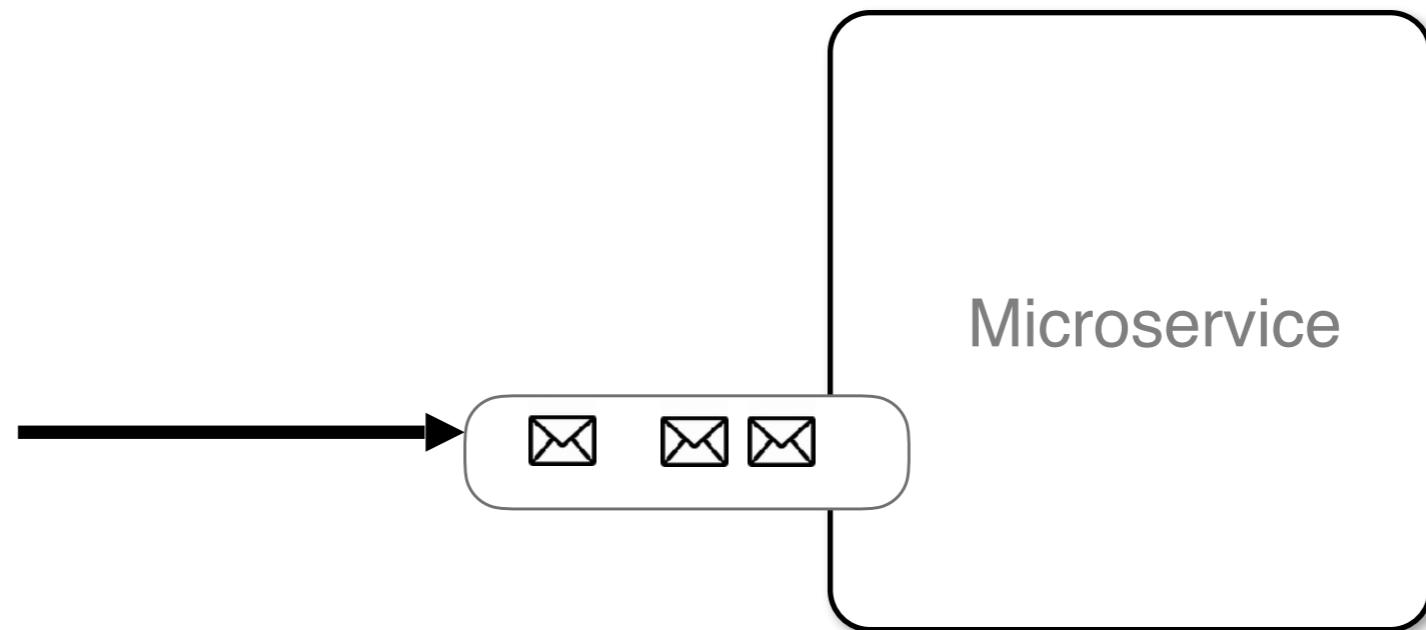
	REST	WSS	GRPC
Browser Support	Y	Y	N
Format	TEXT	TEXT	BINARY, HTTP2
Serialization	JSON	JSON	Proto Buf
Performance	--	+	++
Use case	North South	Streaming	East West







* Message

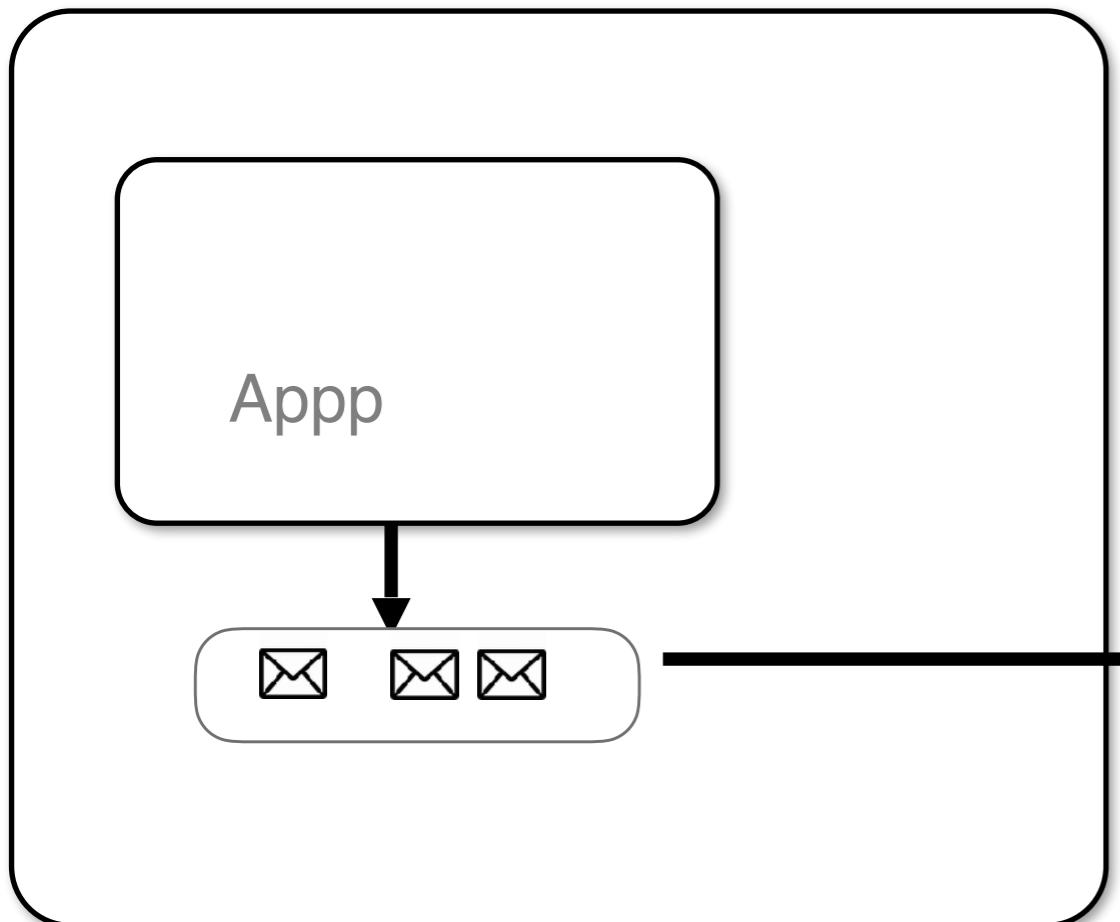


```
fun(){  
    while(true){  
        Msg m = GetMessage();  
        ....  
        m.ack();  
    }  
}
```

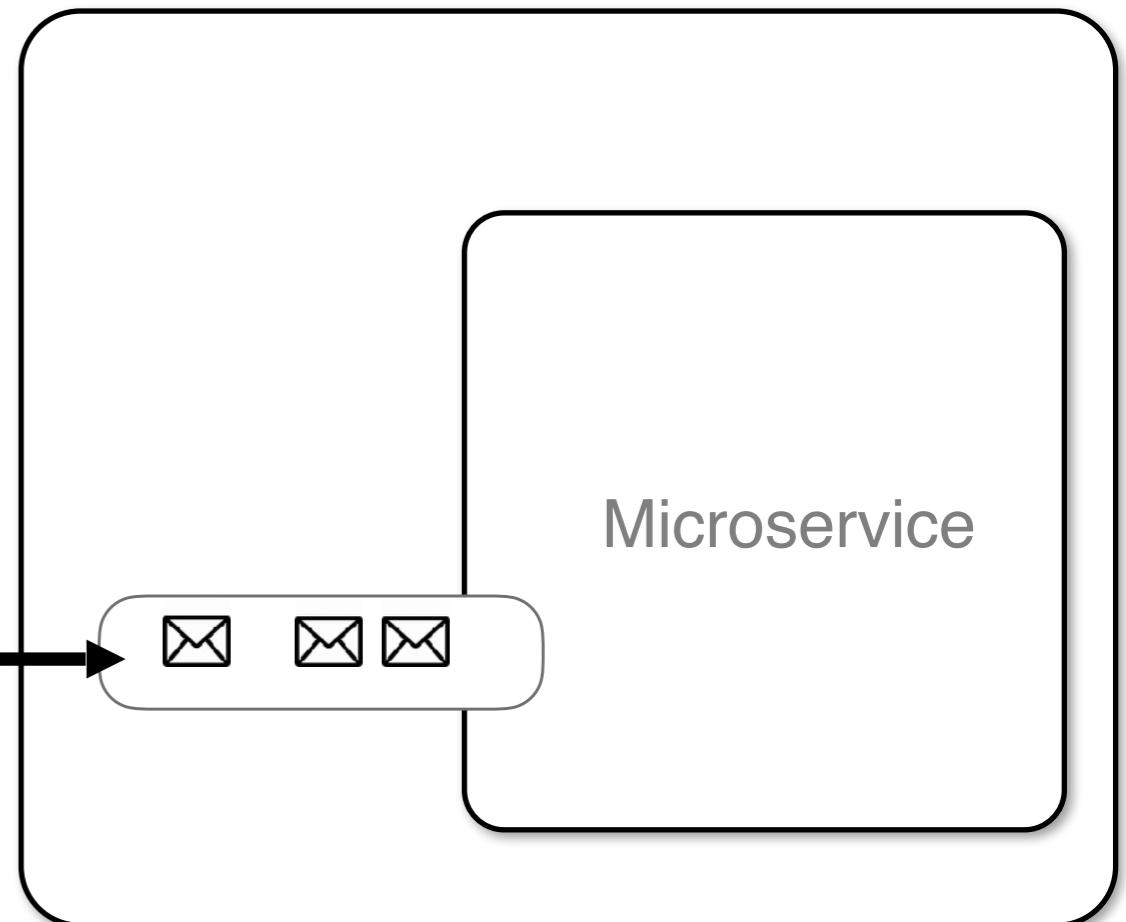
MSMQ

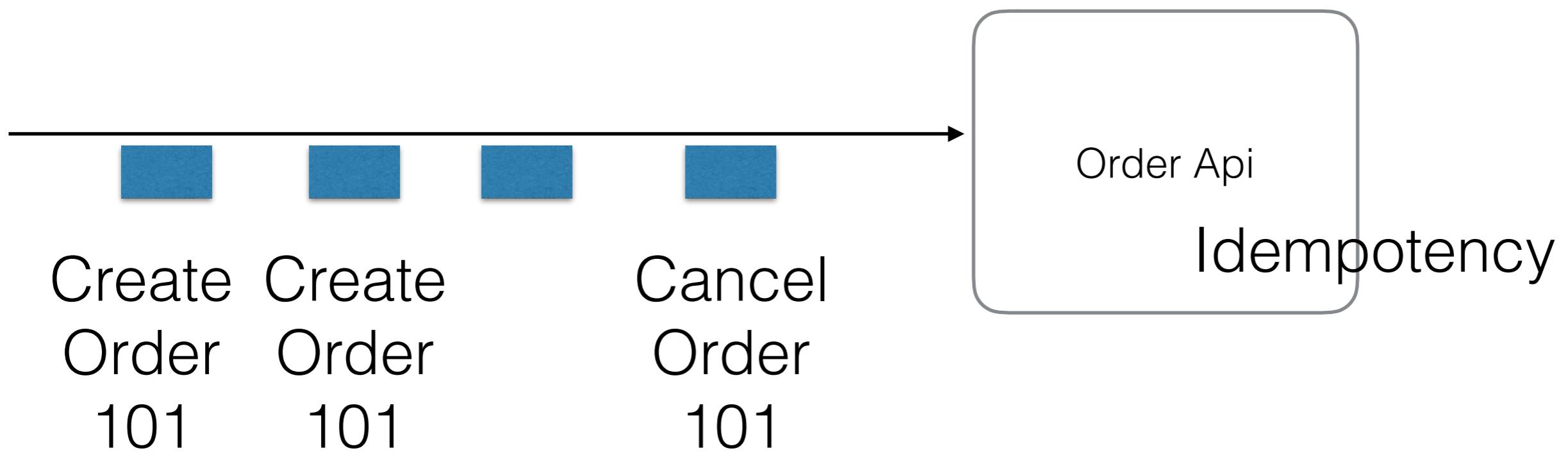
* Message

Client



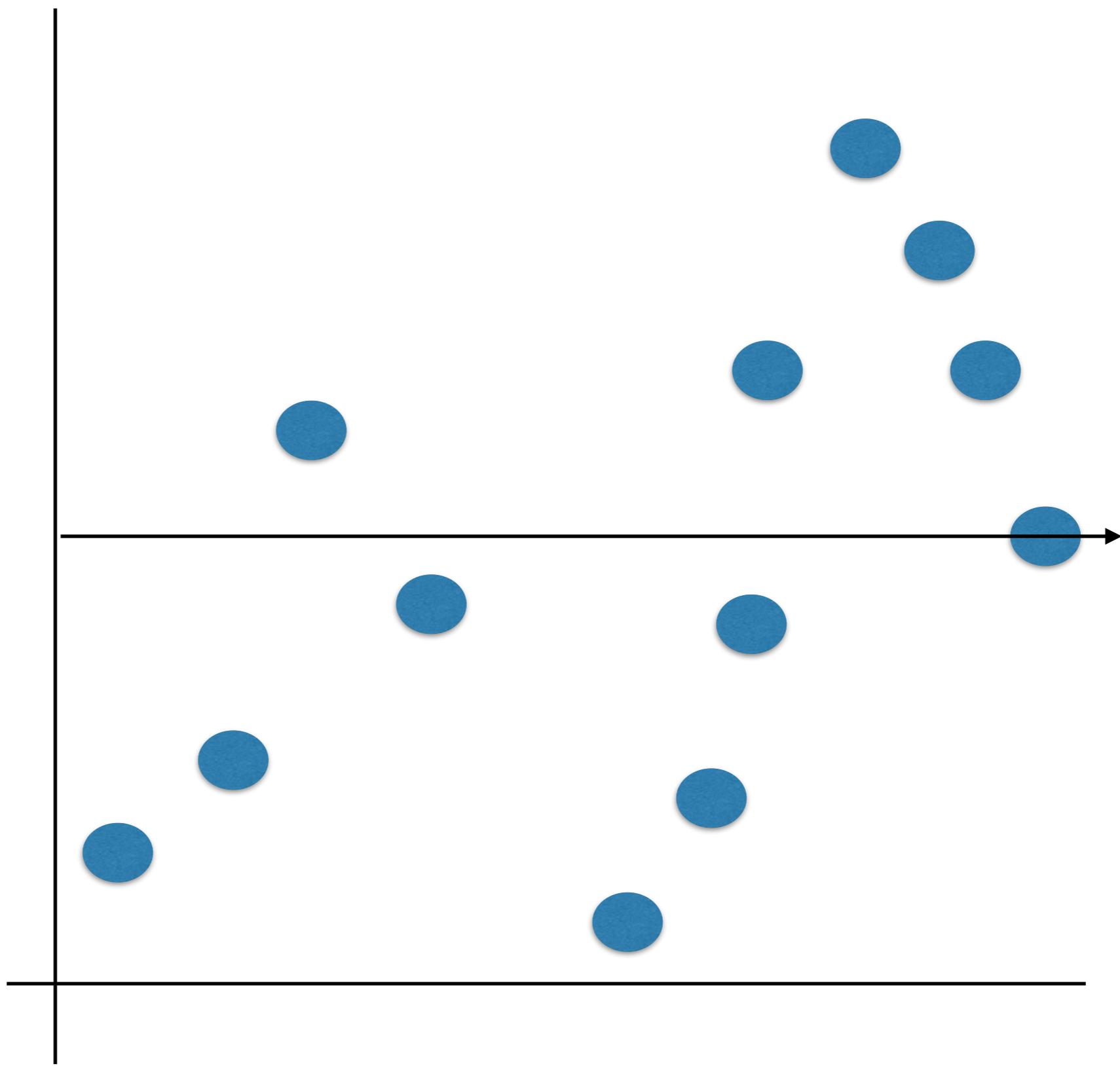
Server



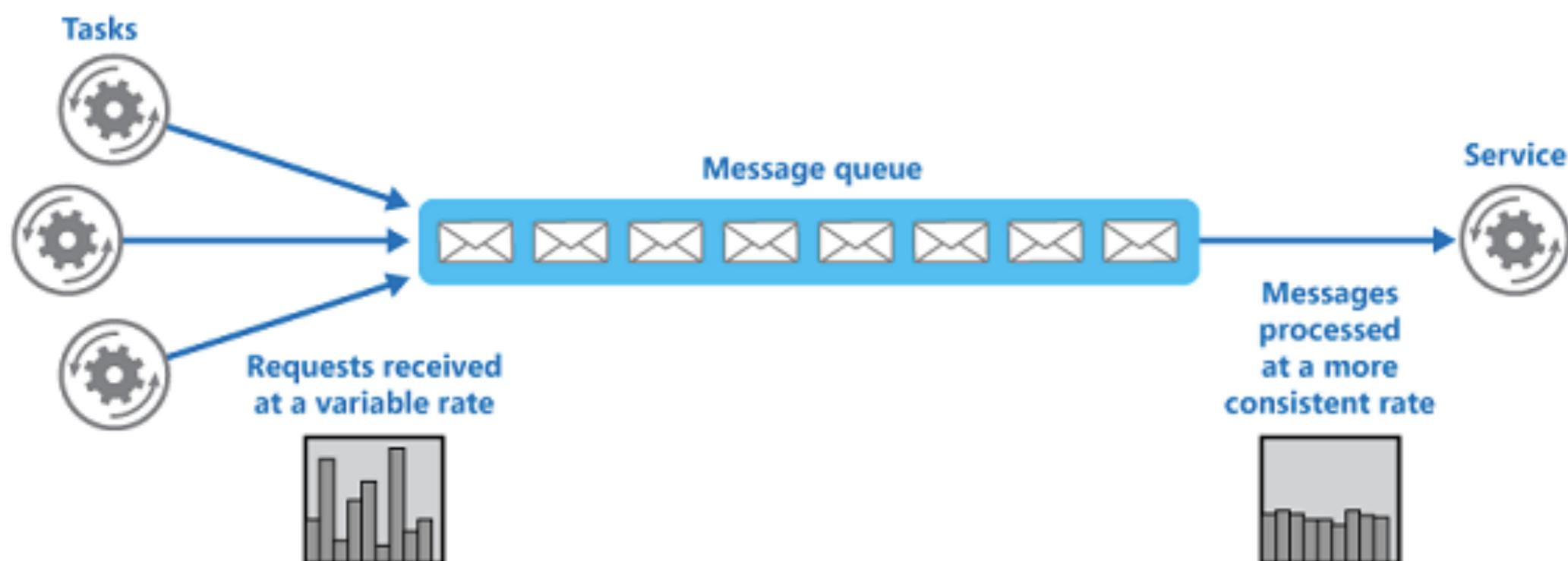


Request

Time

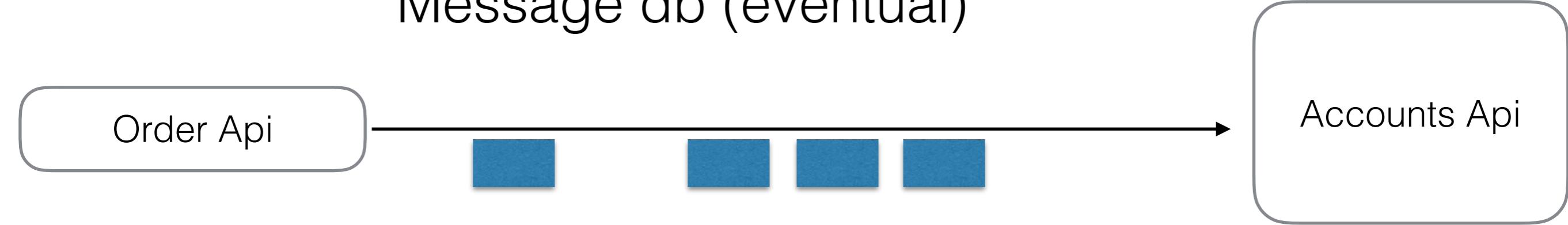


Queue-based Load Levelling



source:msdn

Message db (eventual)

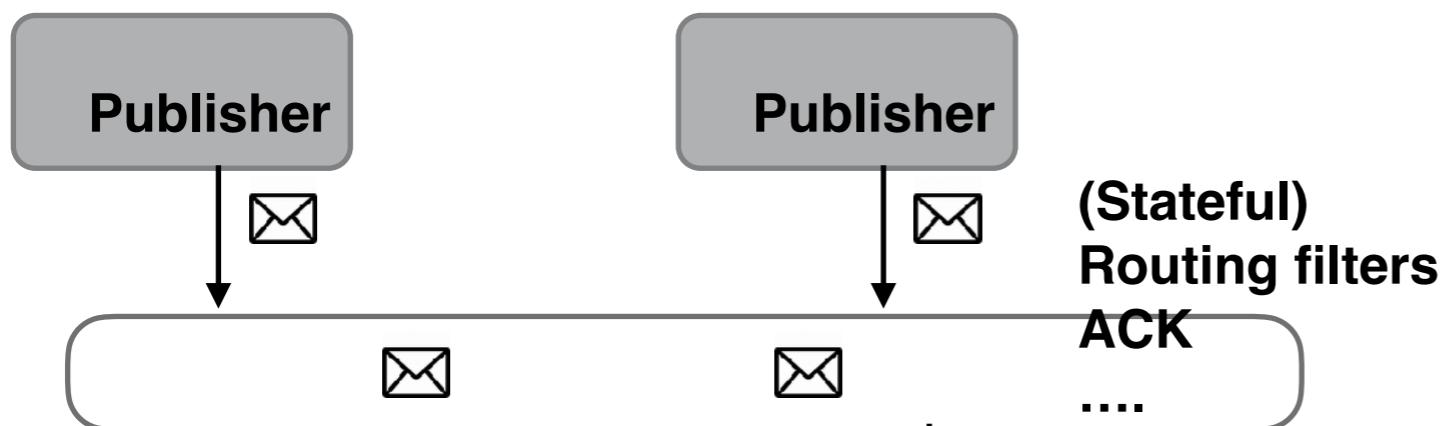
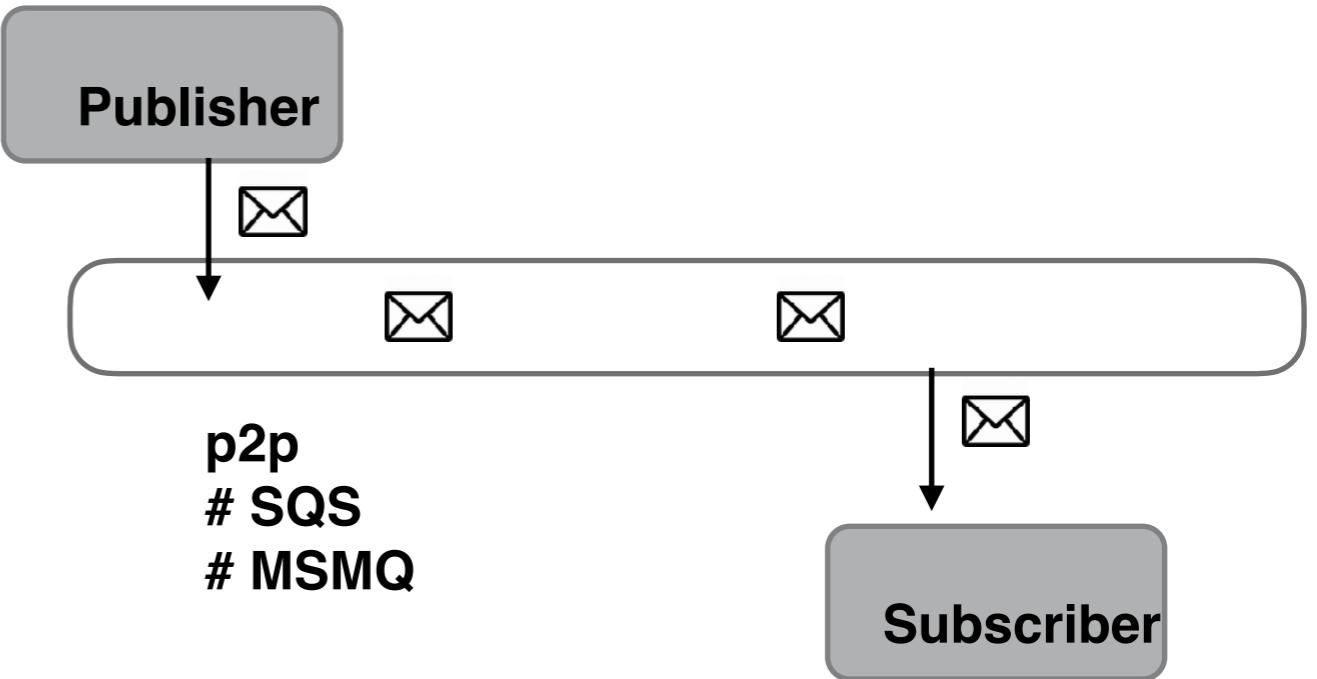


`msg = getmessage()`

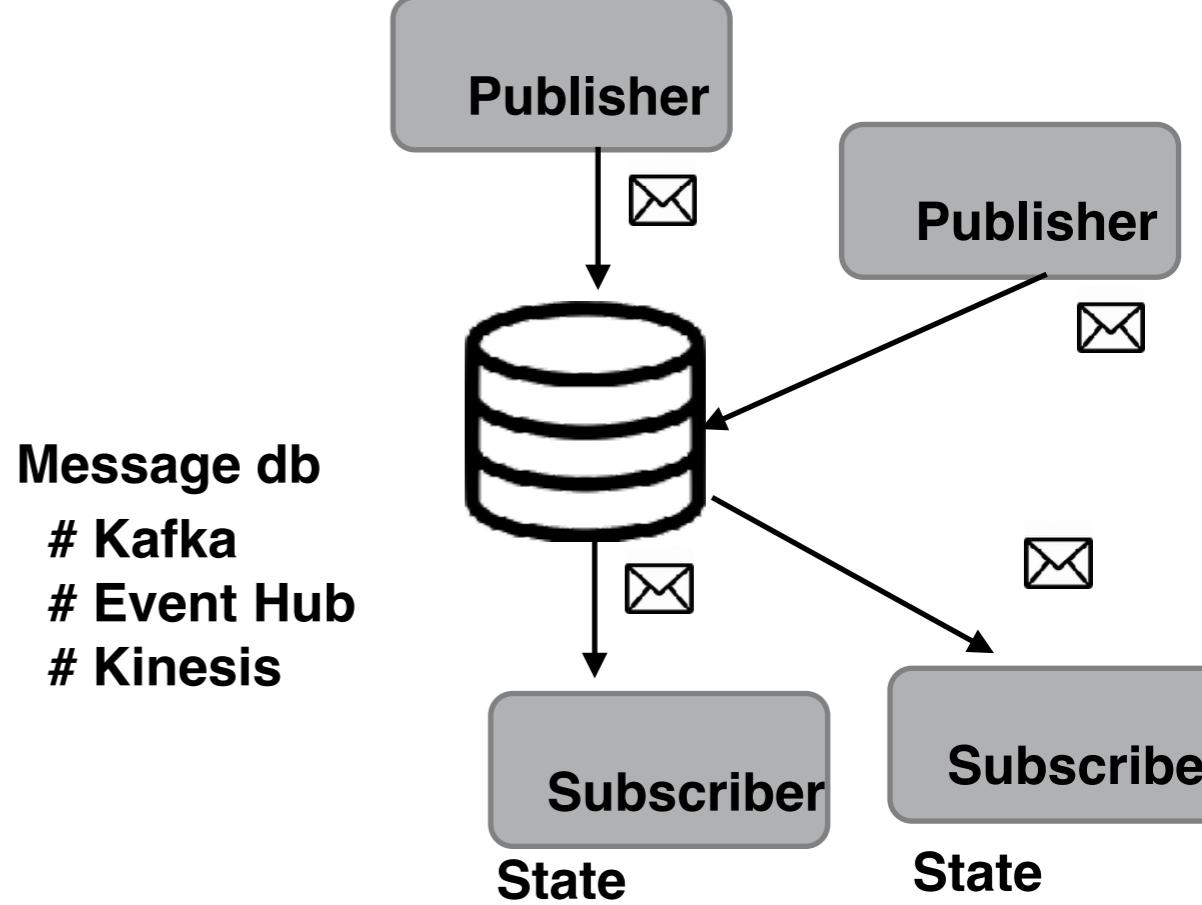
...

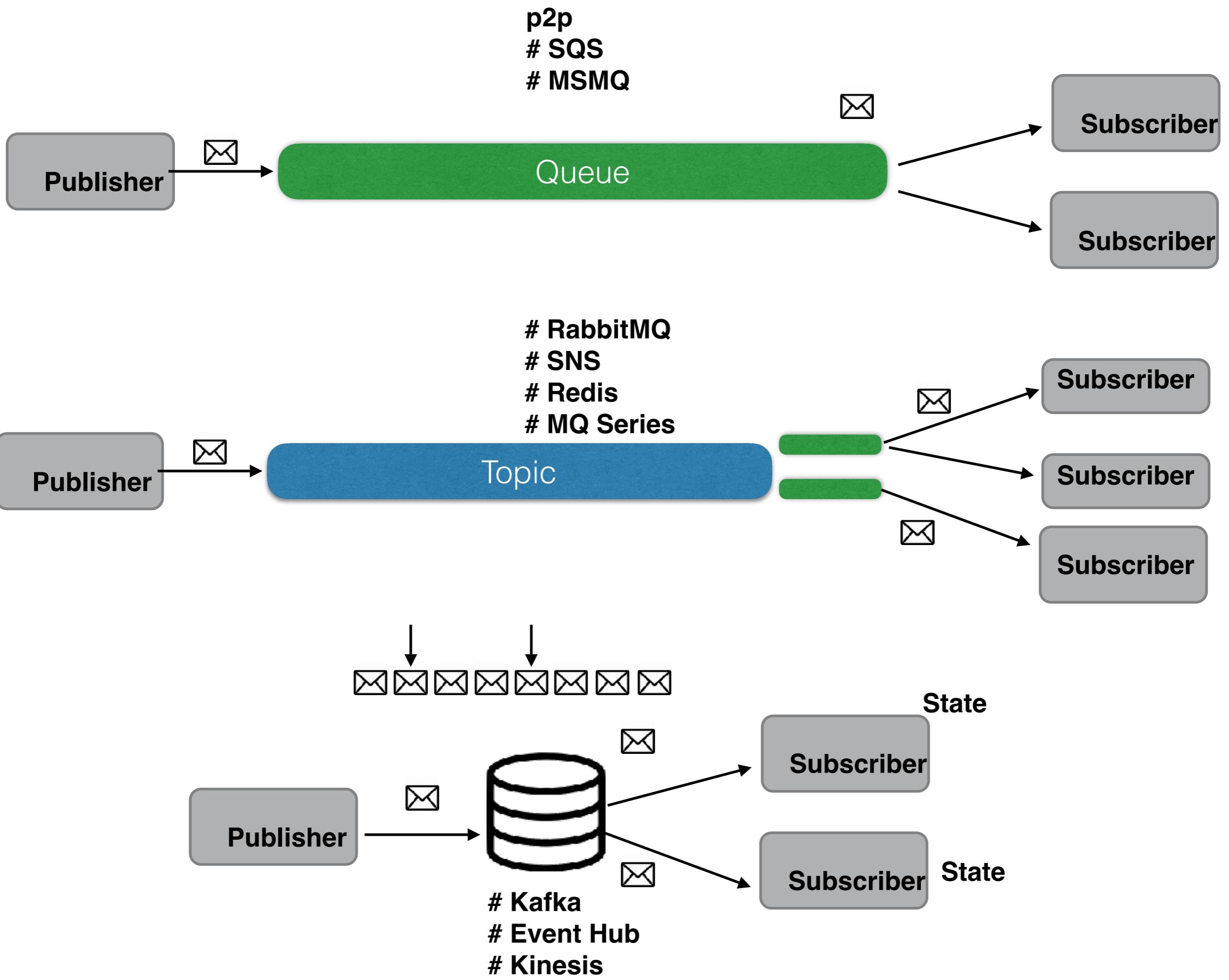
...

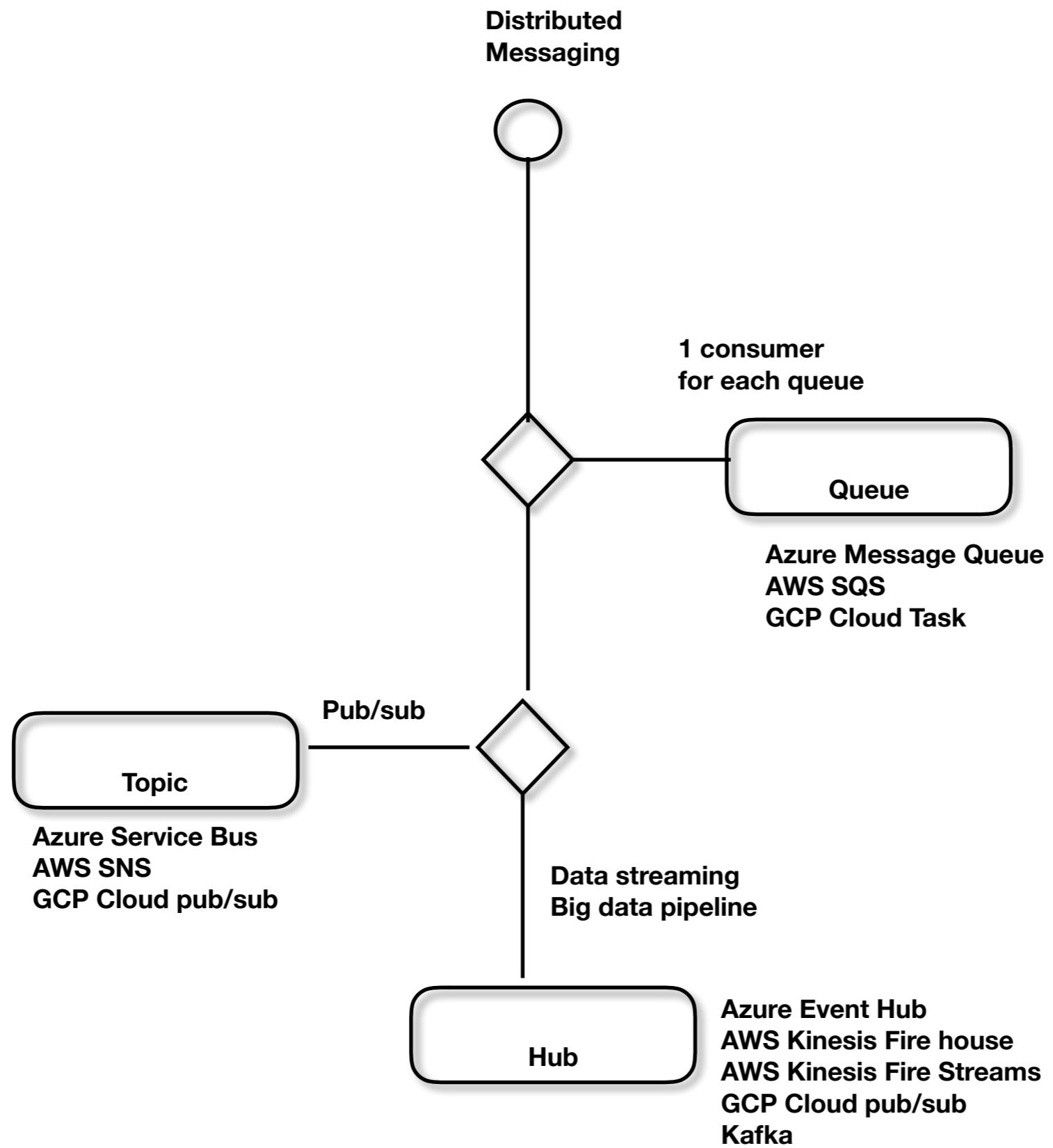
`msg.ack()`



Pub sub
RabbitMQ
SNS
Redis
MQ Series

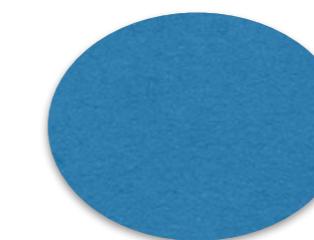
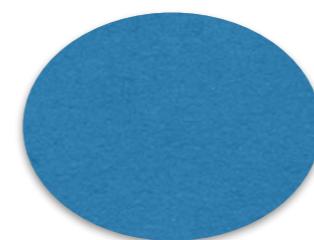
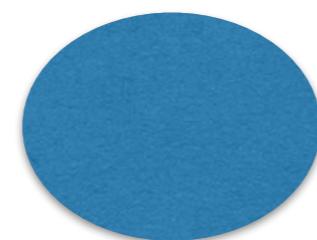
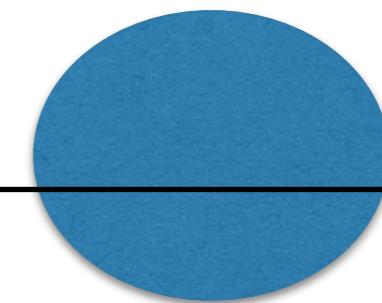
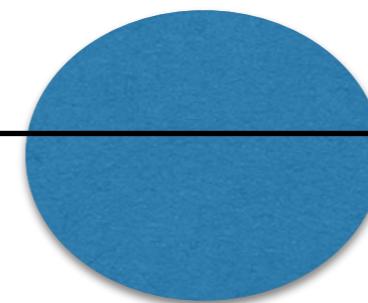
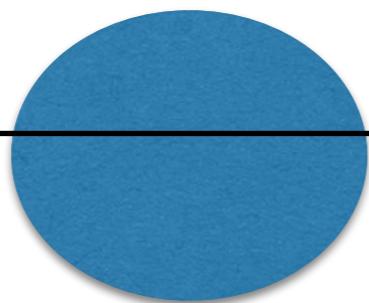
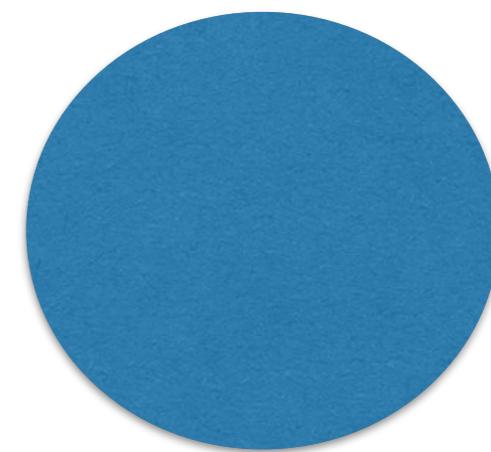
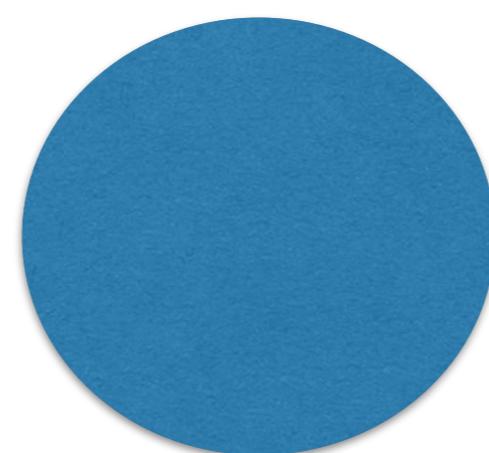
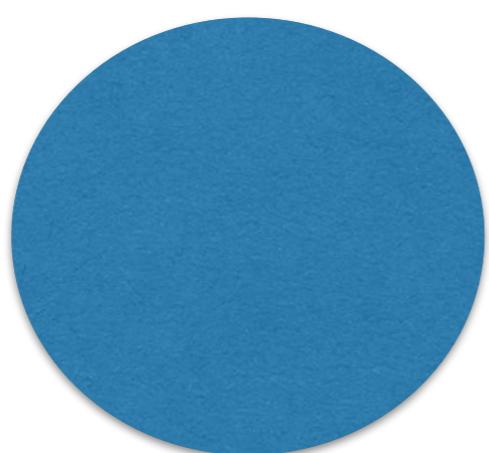






Modeling Microservice

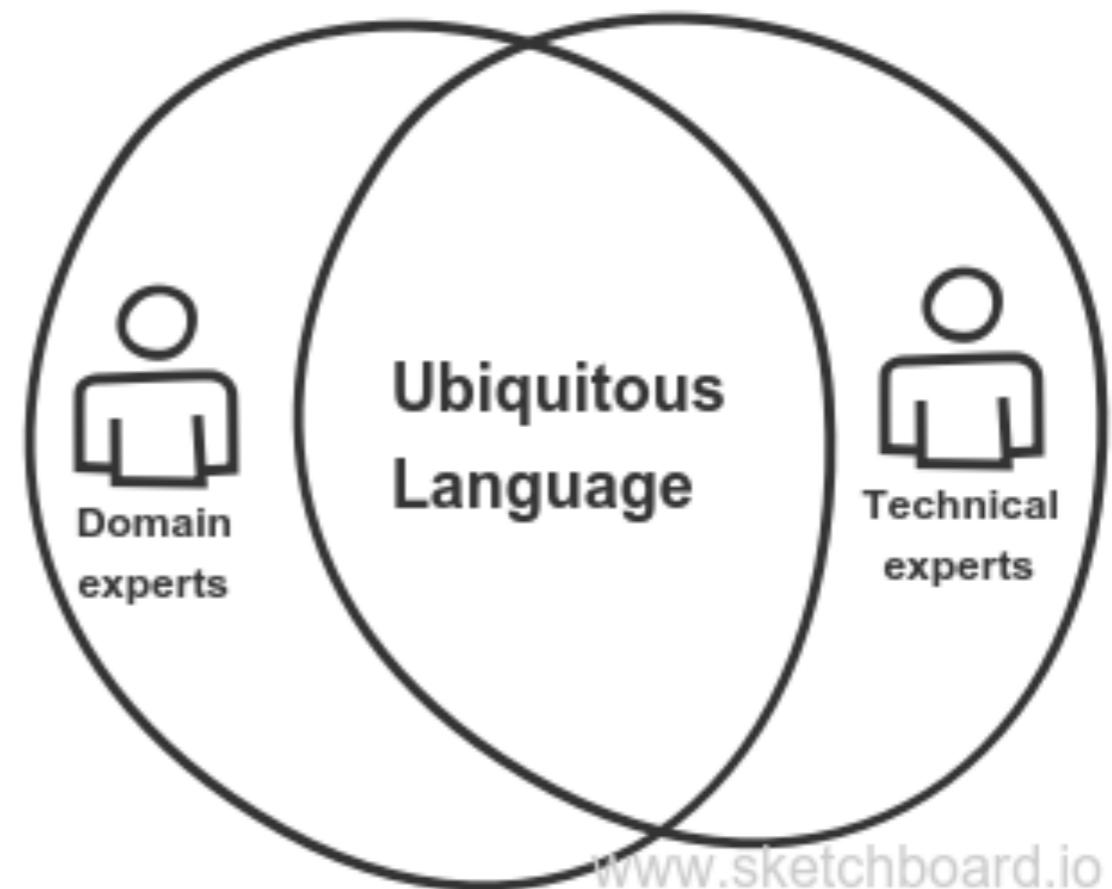
Coarse



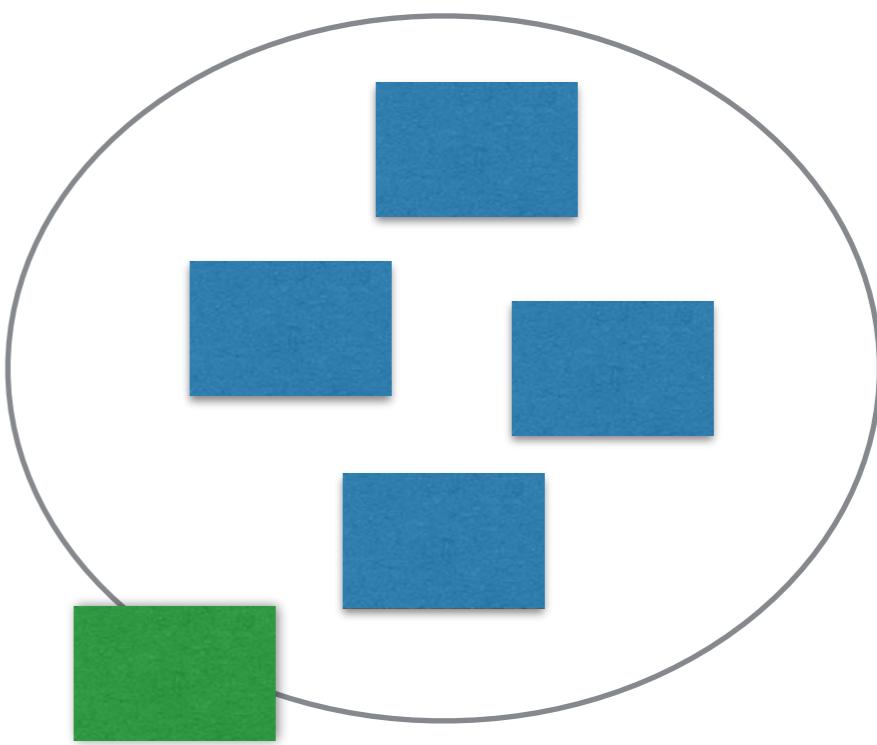
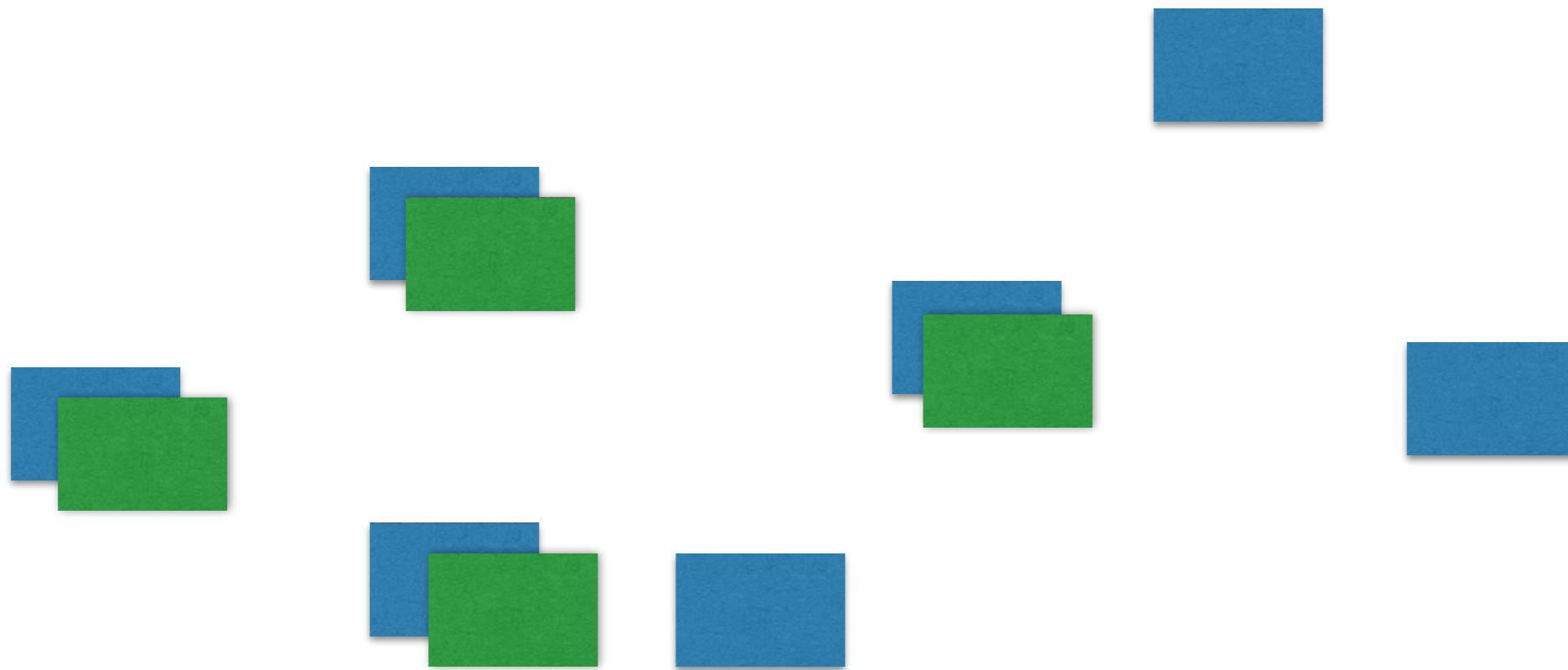
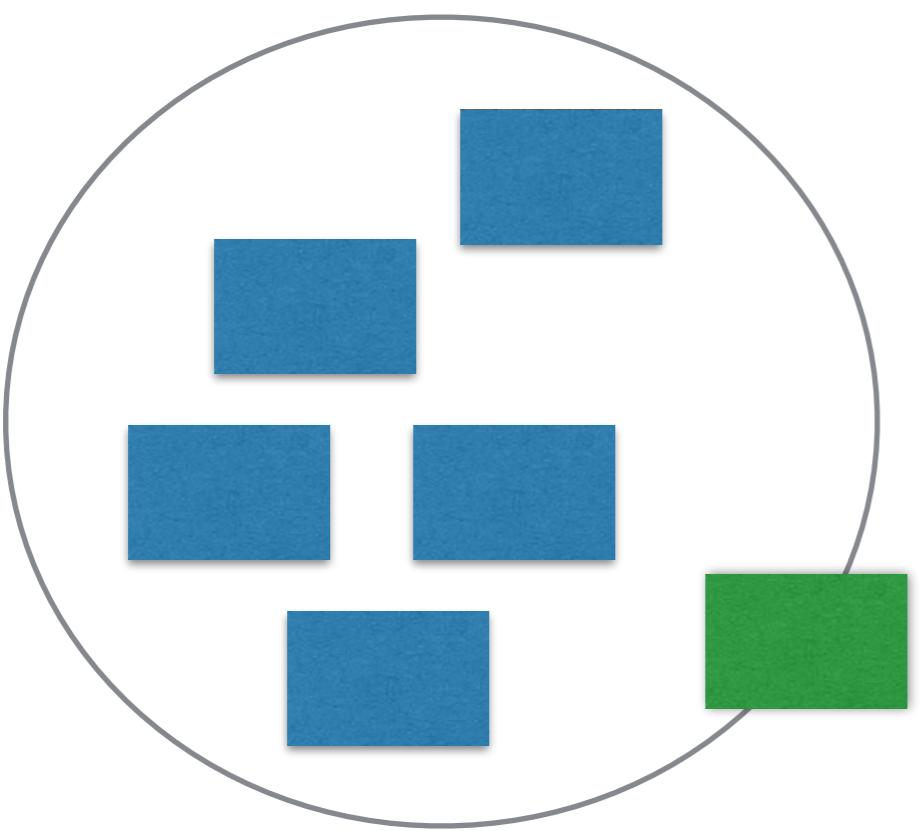
Fine

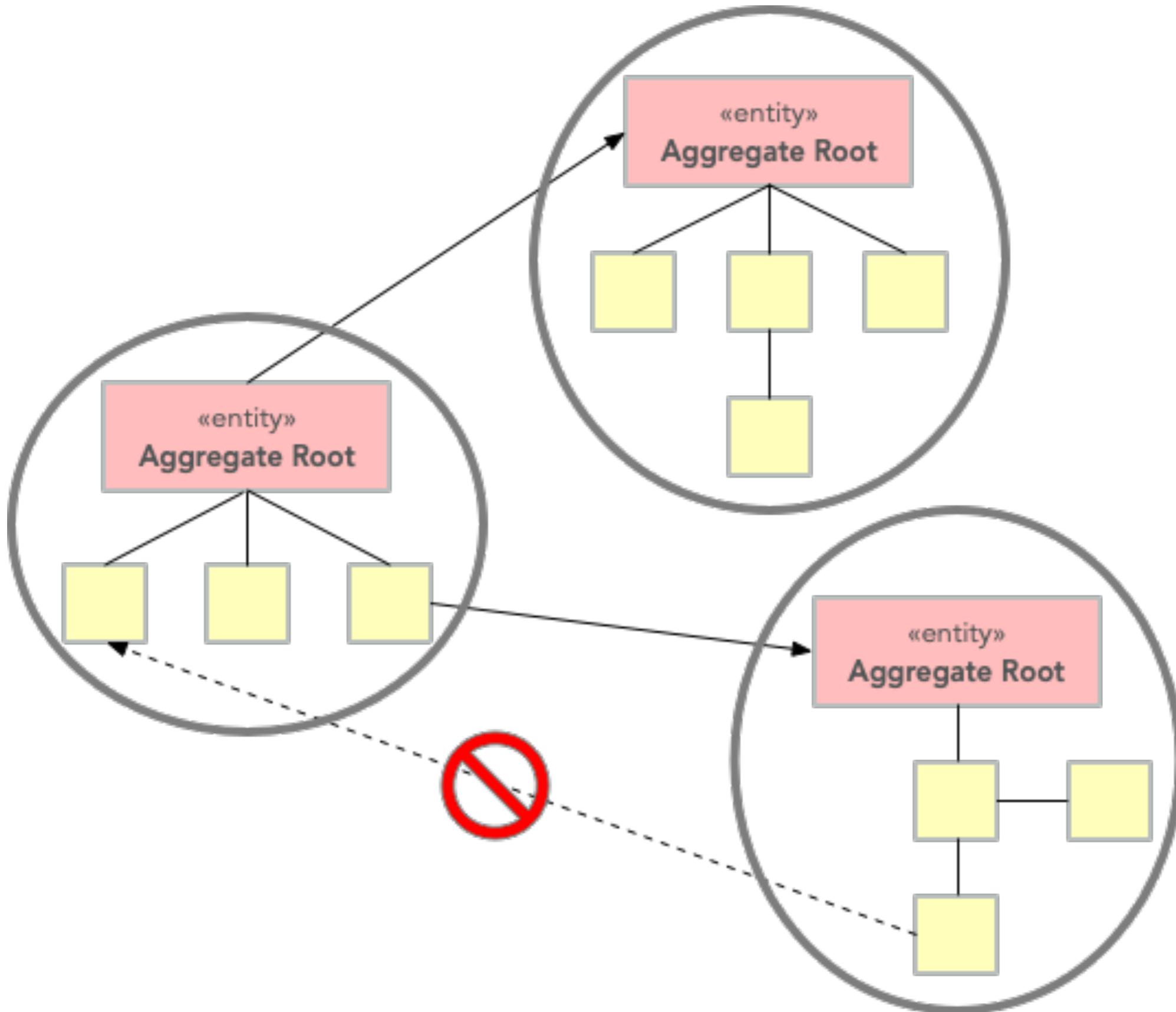
DDD

Ubiquitous language
Bounded Context
Aggregates



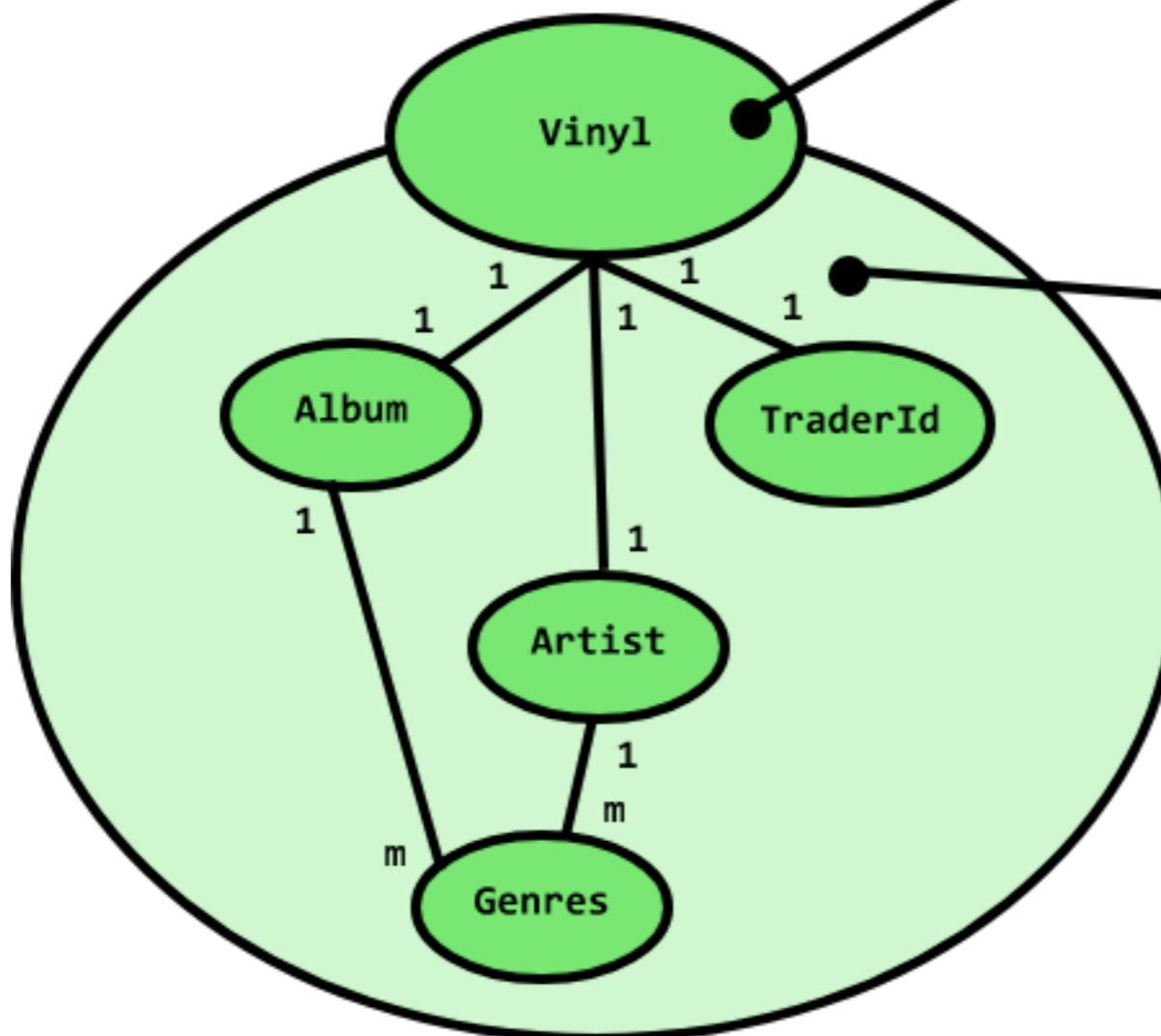
- Stack
 - -> push
 - TestPush
 - TestPop
 - -> pop
- ```
Stack s = new Stack();
s.push(100);
s.pop();
```





**Aggregate  
Root**

**Aggregate  
Boundary**



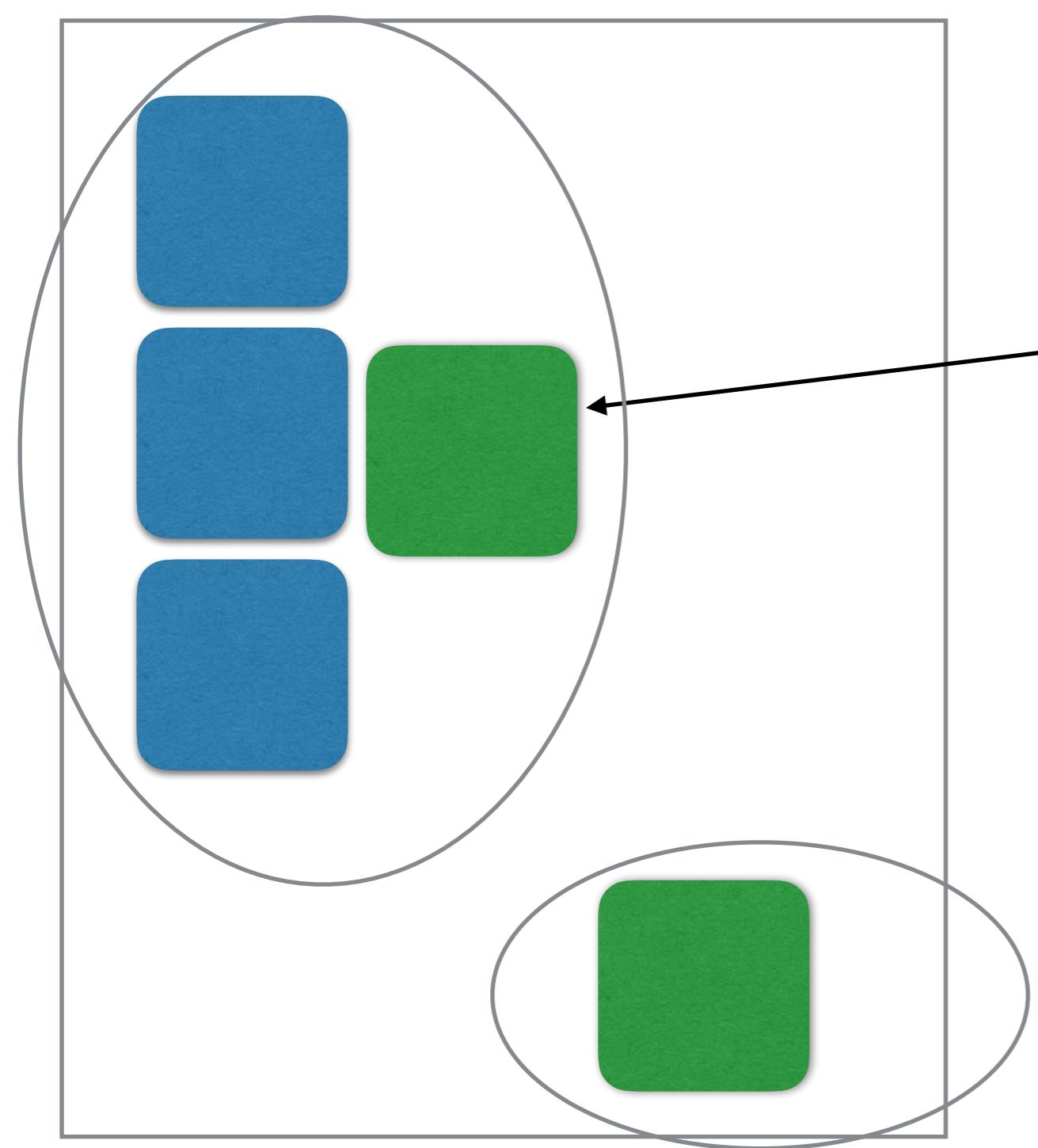
Purchase  
order  
ABC432

Line Item

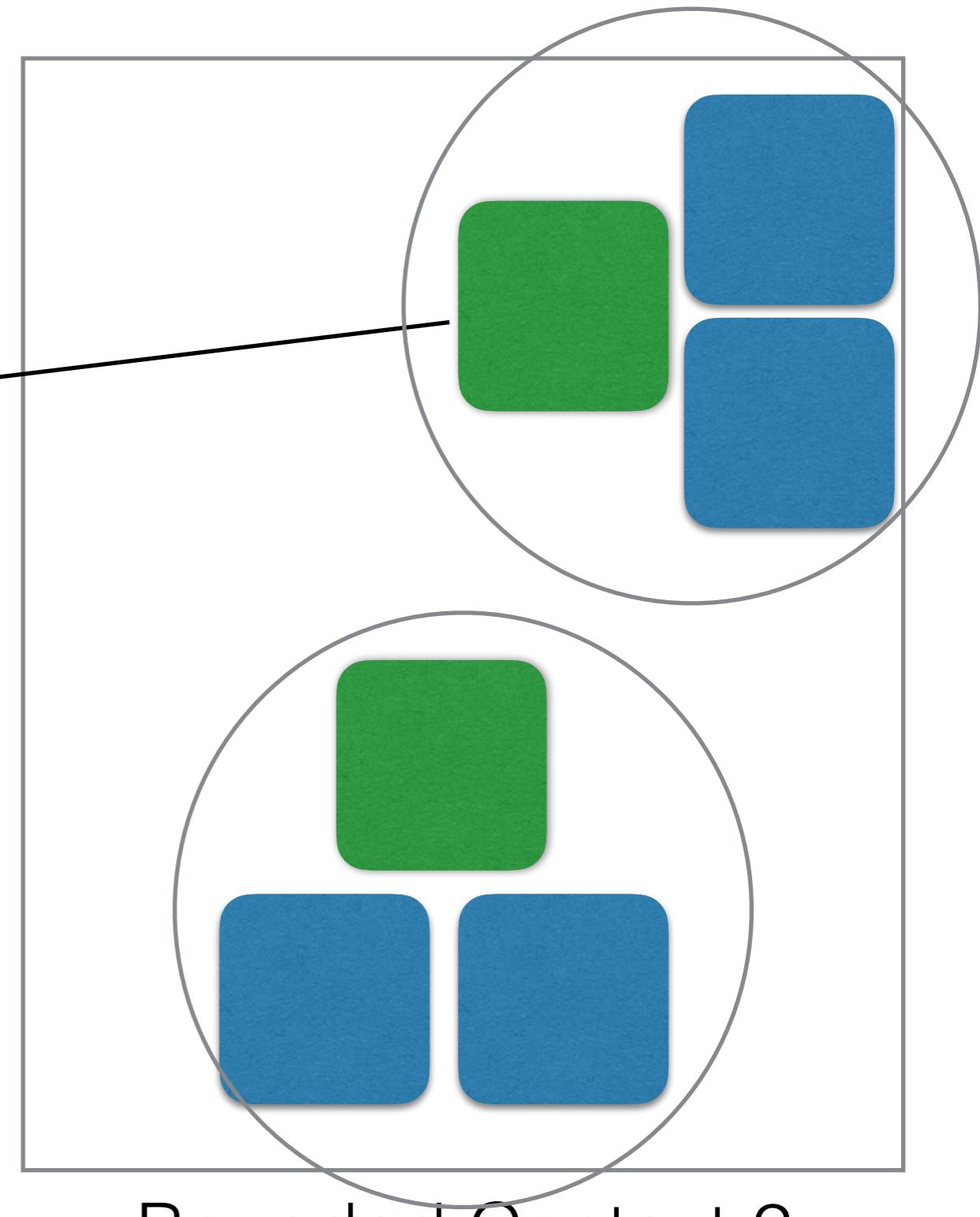
Purchase  
order  
XYZ124

Line Item

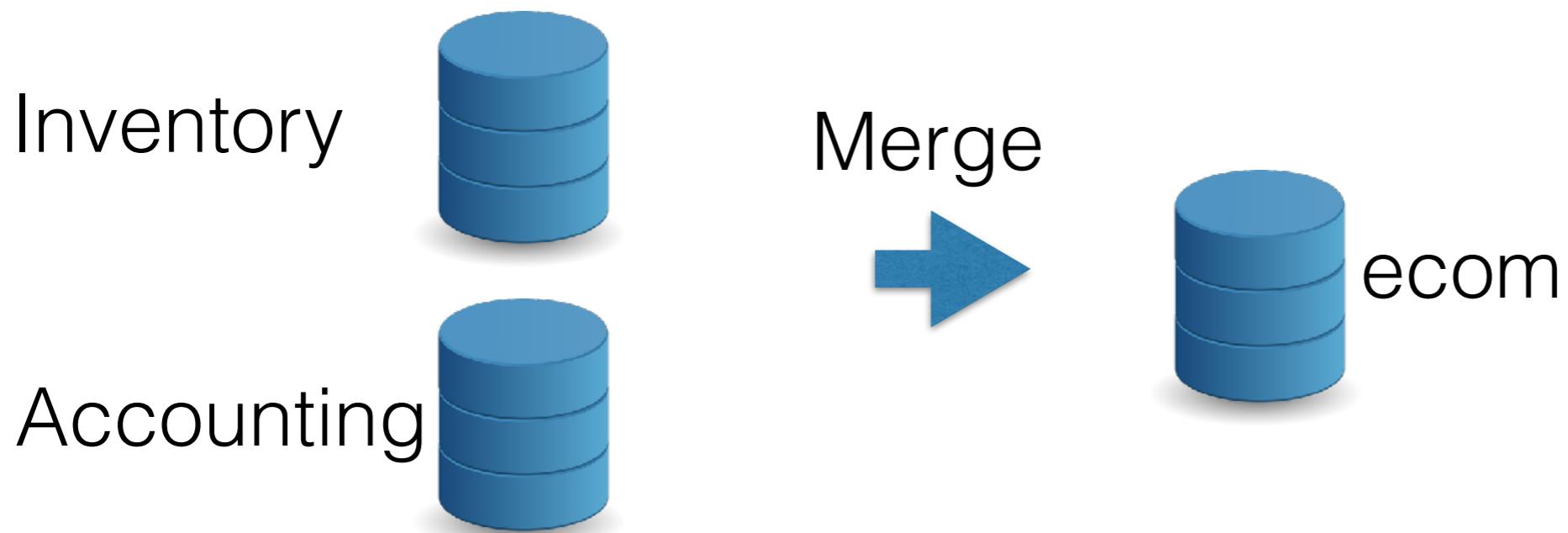
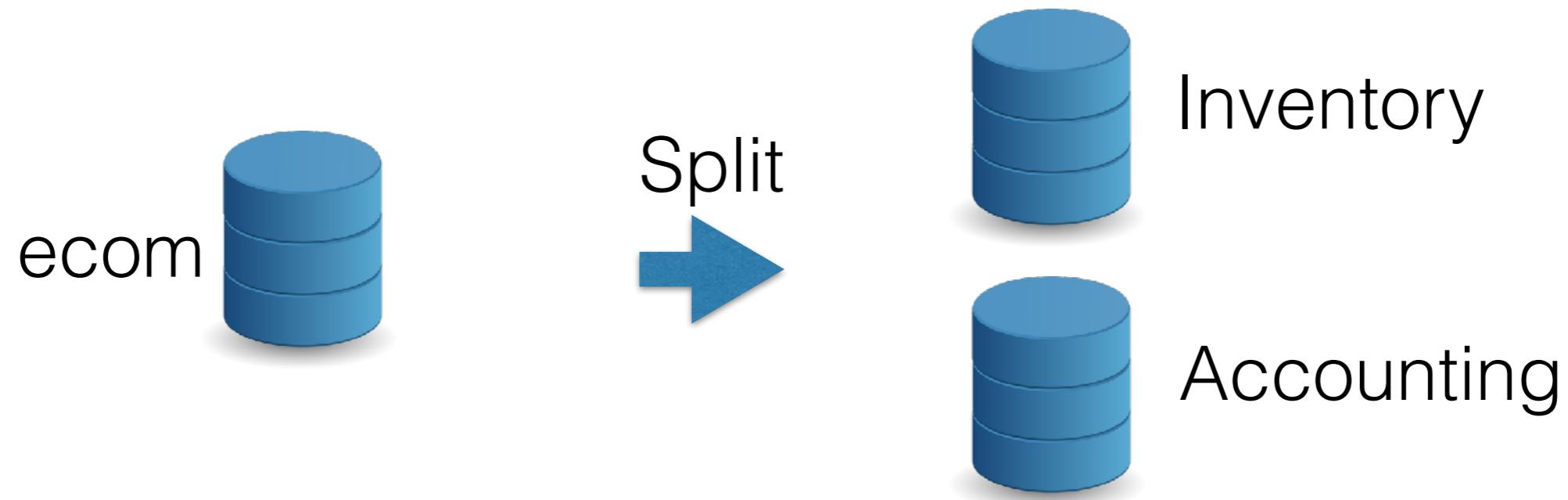
Line Item



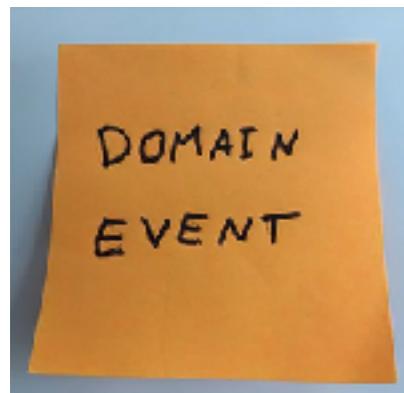
Bounded Context 1



Bounded Context 2



# Event storming



Step 1: Create domain events



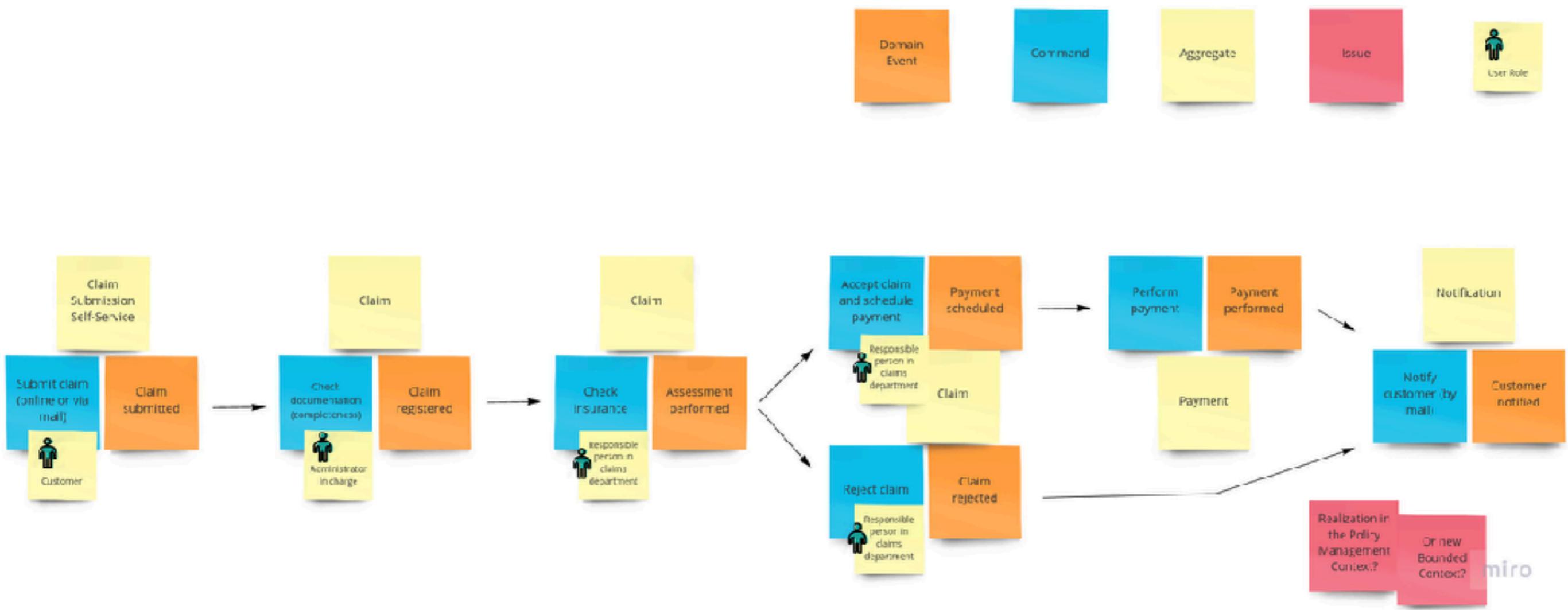
Step 2: Add the commands that caused the domain event



Step 2b: Add the actor that executes the command



Step 3: Add corresponding aggregate

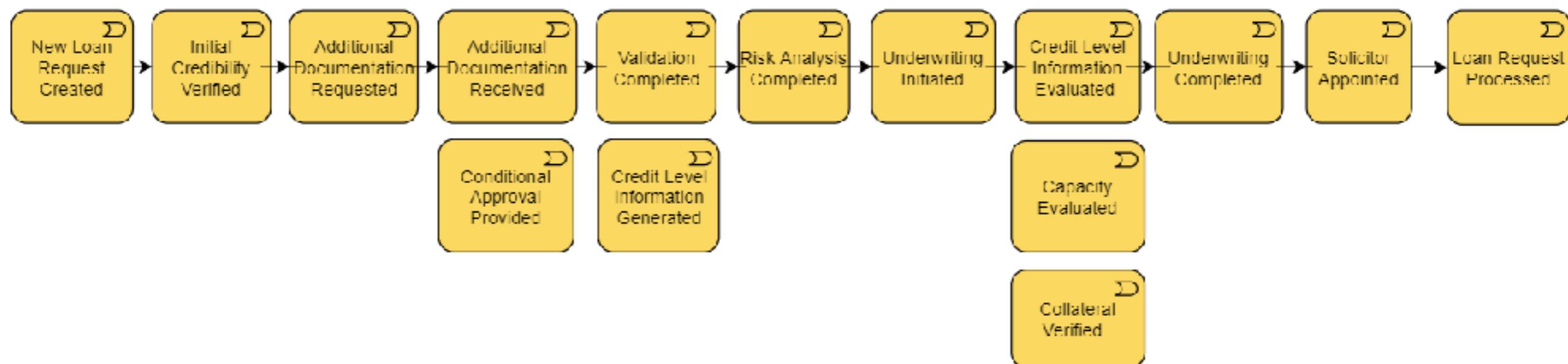


[https://github.com/getmubarak/Microservice/blob/master/  
case%20study/Loan%20Approval%20Process.md](https://github.com/getmubarak/Microservice/blob/master/case%20study/Loan%20Approval%20Process.md)

# Identification of Domain Events

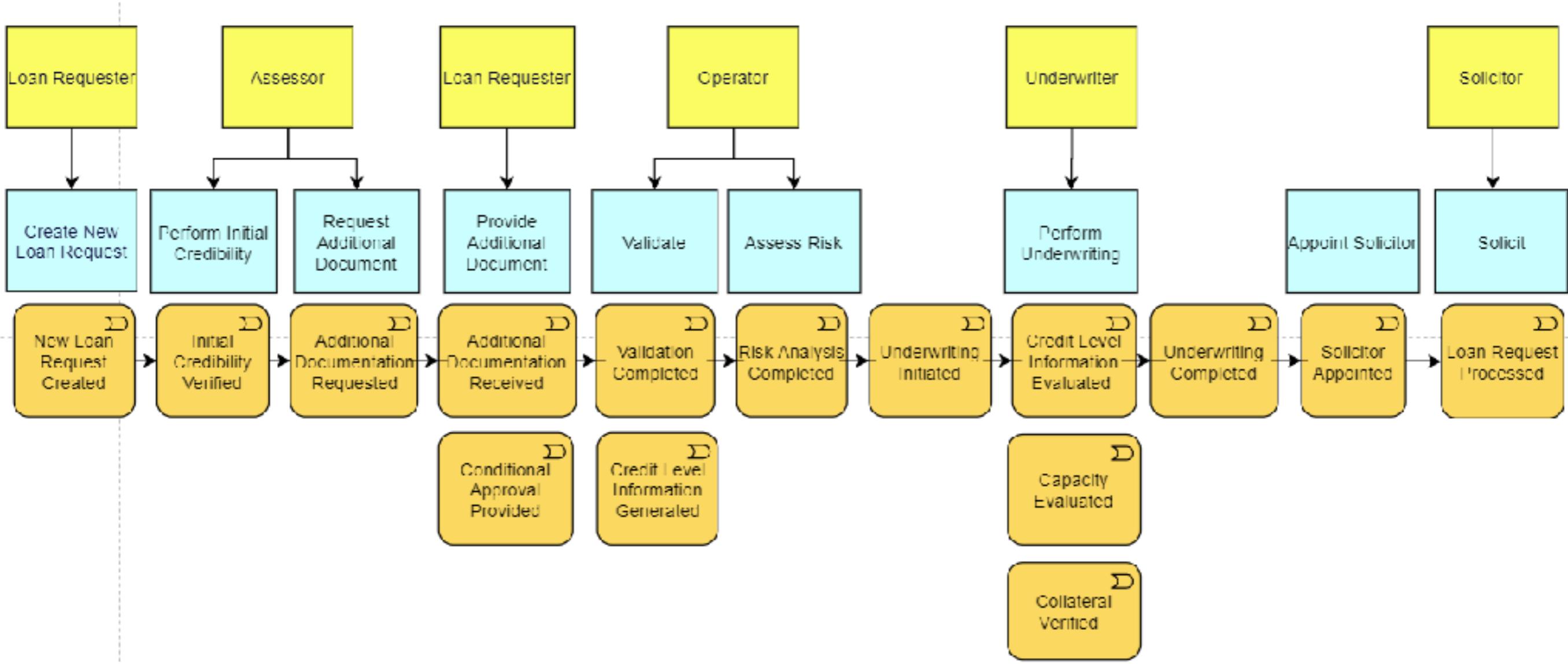


# Time Sequencing of Domain Events



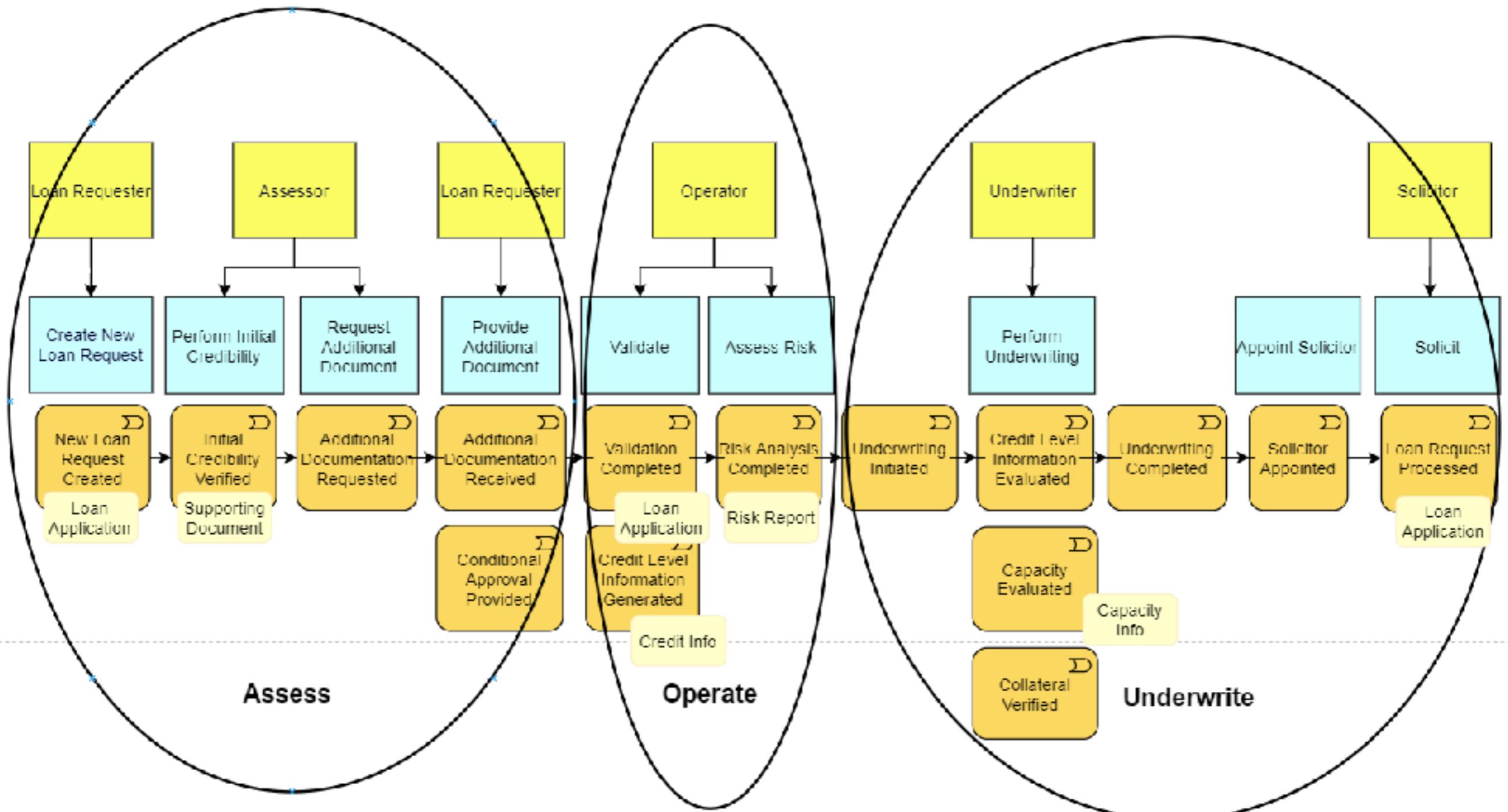
parallel have been stacked up vertically

# Identification of Triggers/Commands

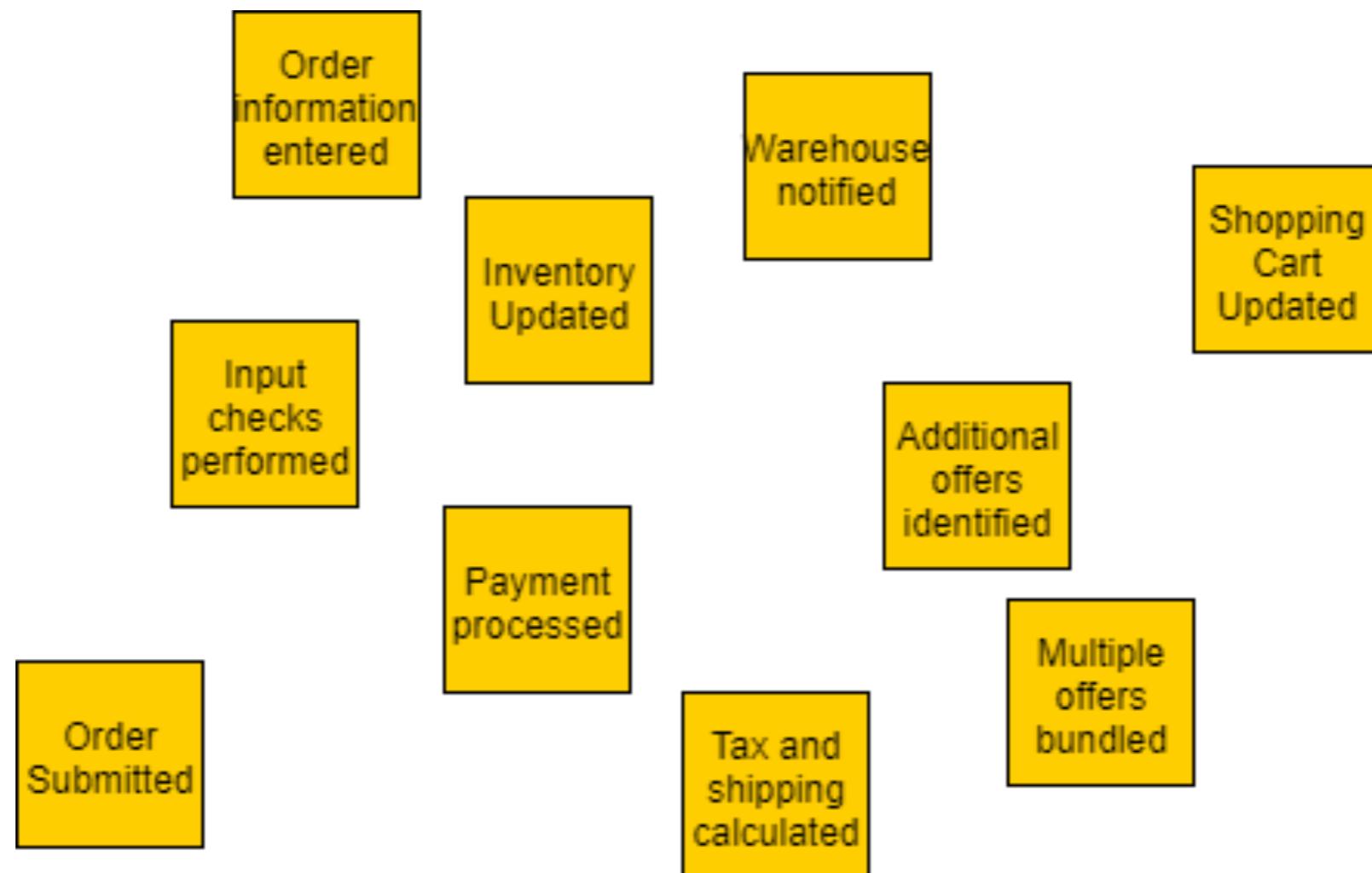


- Yellow represents the actors
- Blue represents the commands issued by the actors
- Orange are the events we started with

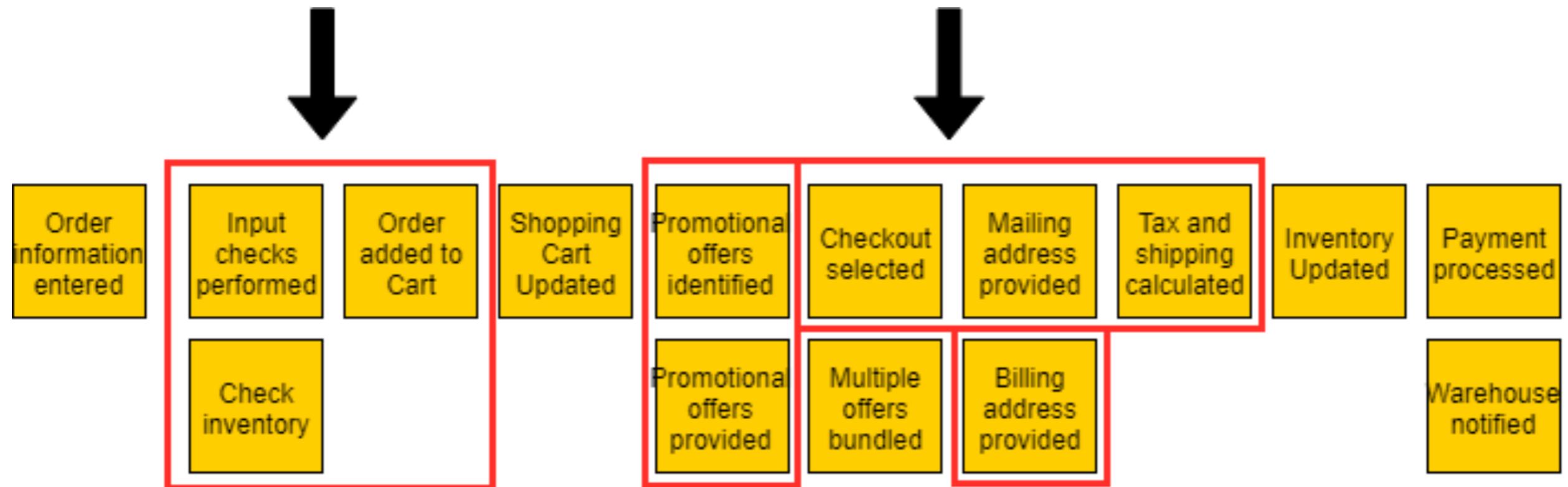
# Identification of Aggregates



## Step #1 – Event Discovery

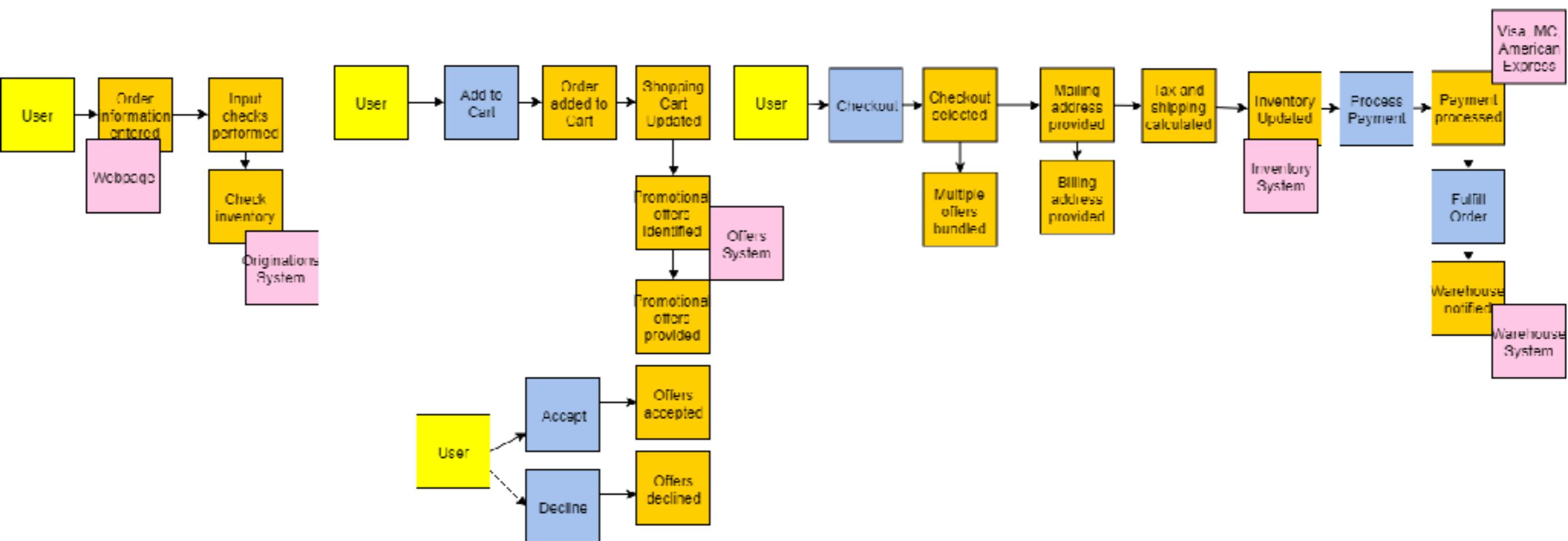


## Step #2 – Placing the Events in Sequence

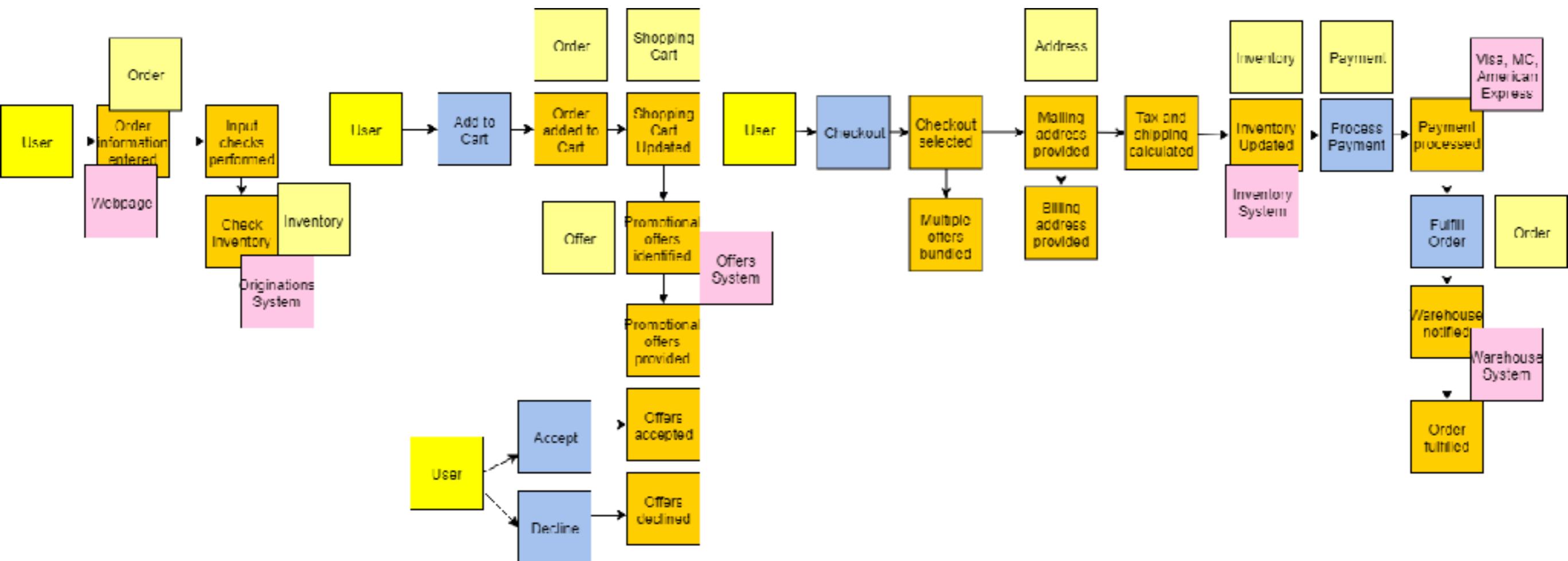


missing events (outlined in red)

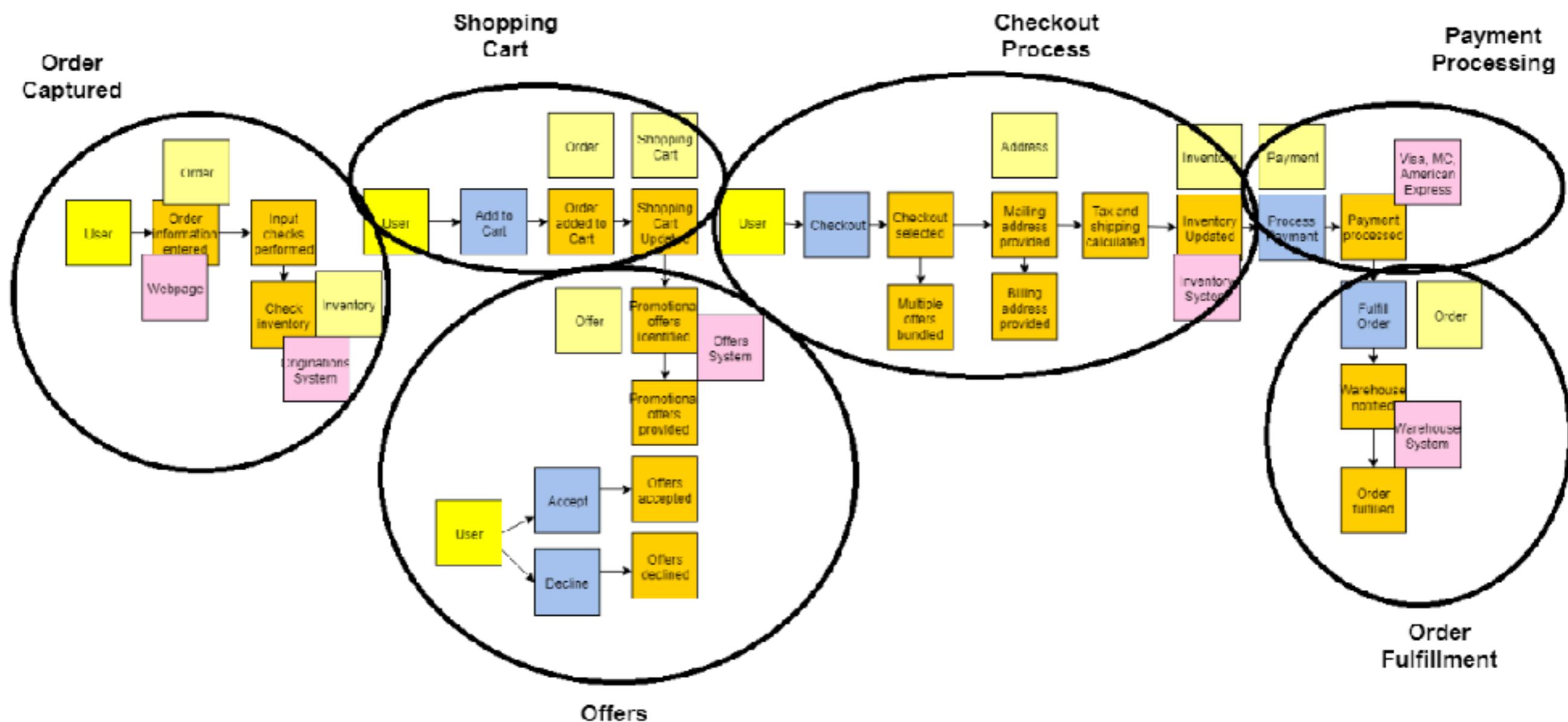
## Step #3 – Modeling Out the Broader Ecosystem



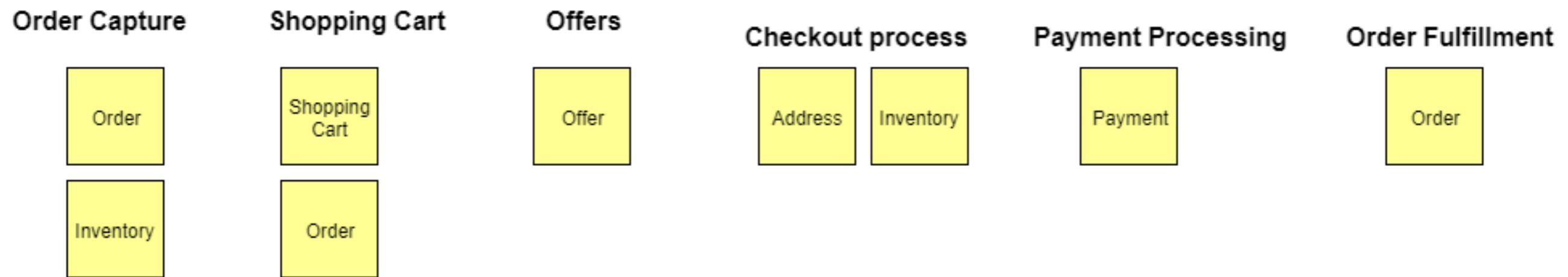
## Step #4 – Identify Aggregates



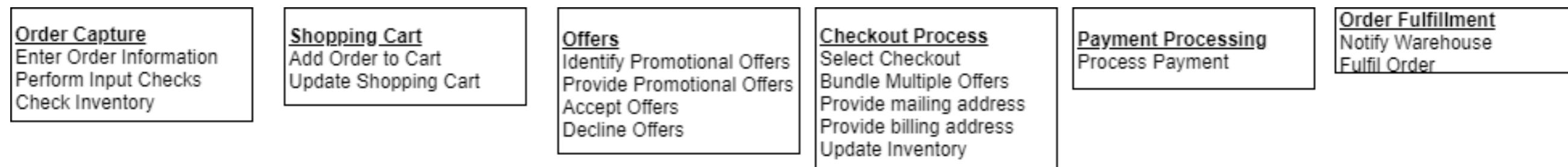
# Step #5 – Bounded Context Categorization of Events

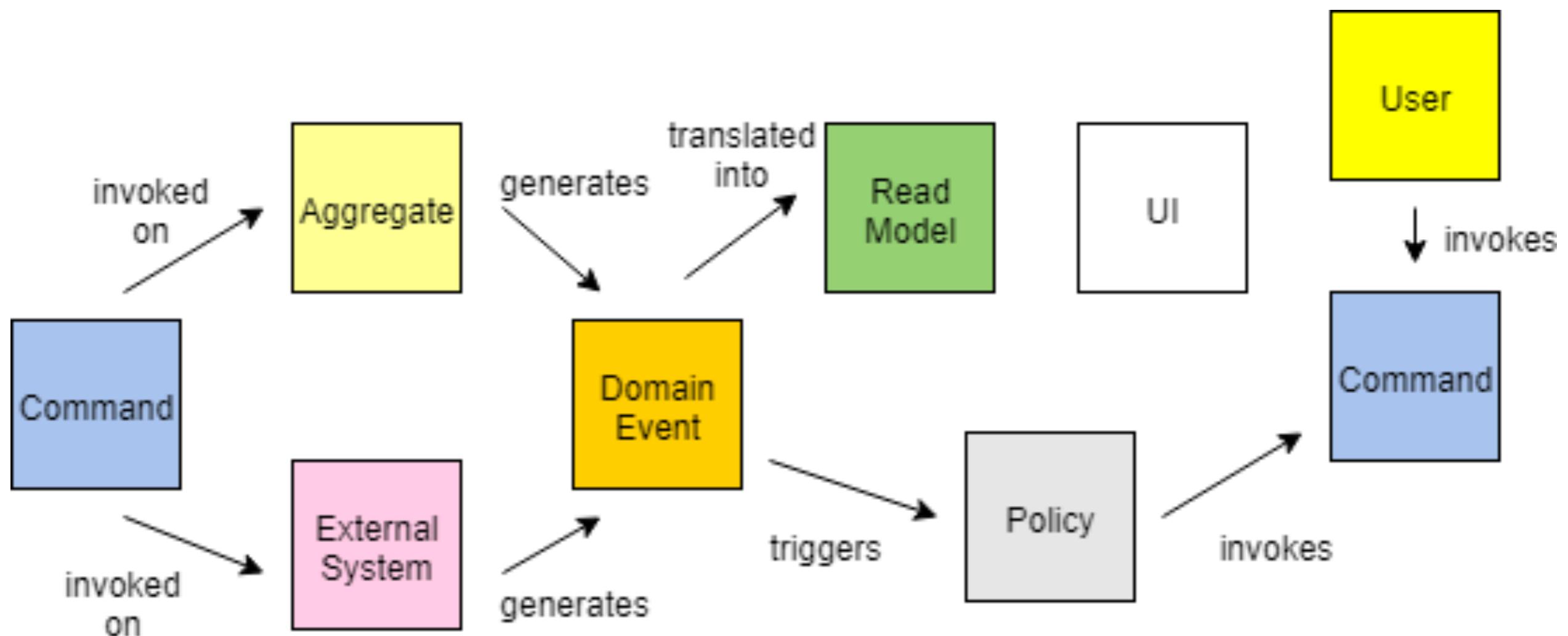


## Step #6: Create Services

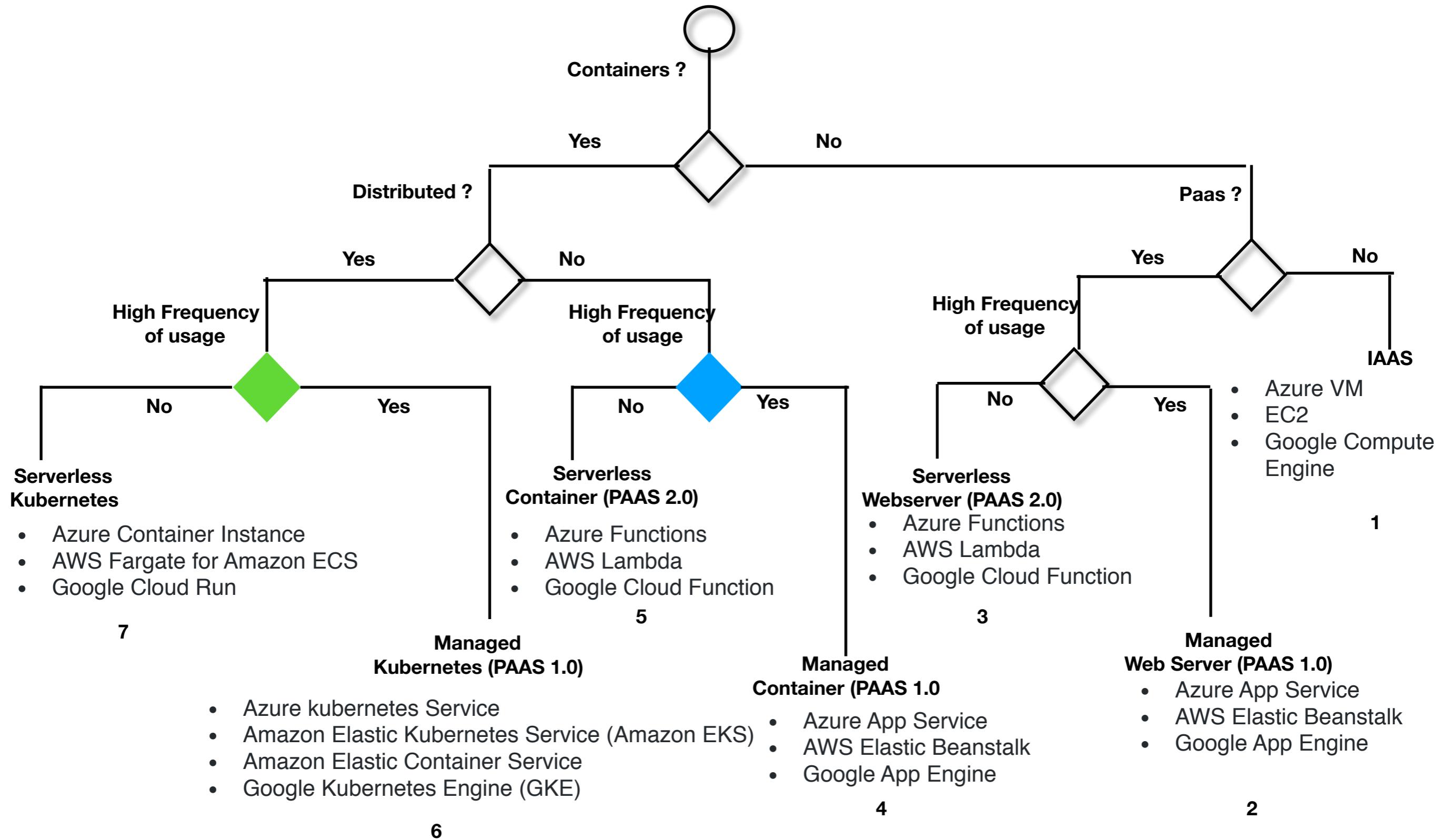


## Step #7: Create Capabilities





# Deployment View



7

## Managed Kubernetes (PAAS 1.0)

- Azure Kubernetes Service
- Amazon Elastic Kubernetes Service (Amazon EKS)
- Amazon Elastic Container Service
- Google Kubernetes Engine (GKE)

6

5

## Serverless Container (PAAS 2.0)

- Azure Functions
- AWS Lambda
- Google Cloud Function

4

## Managed Container (PAAS 1.0)

- Azure App Service
- AWS Elastic Beanstalk
- Google App Engine

2

## Managed Web Server (PAAS 1.0)

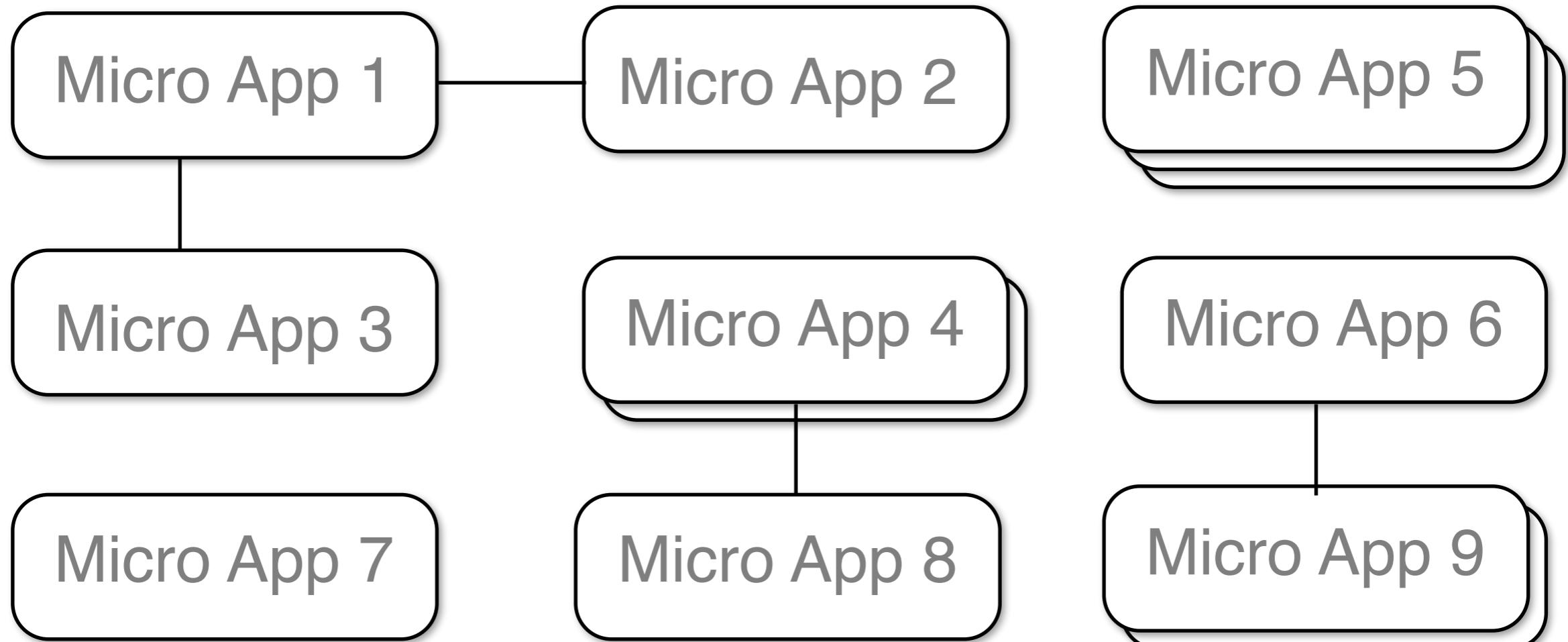
- Azure App Service
- AWS Elastic Beanstalk
- Google App Engine

1

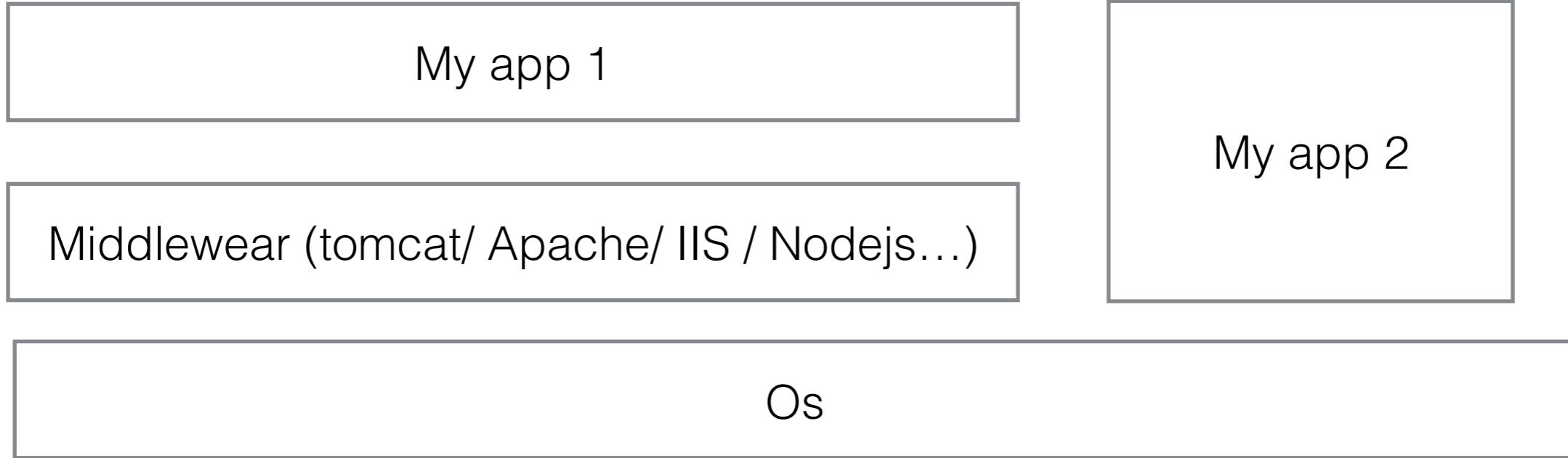
- Azure VM
- EC2
- Google Compute Engine

1

Isolated Env.

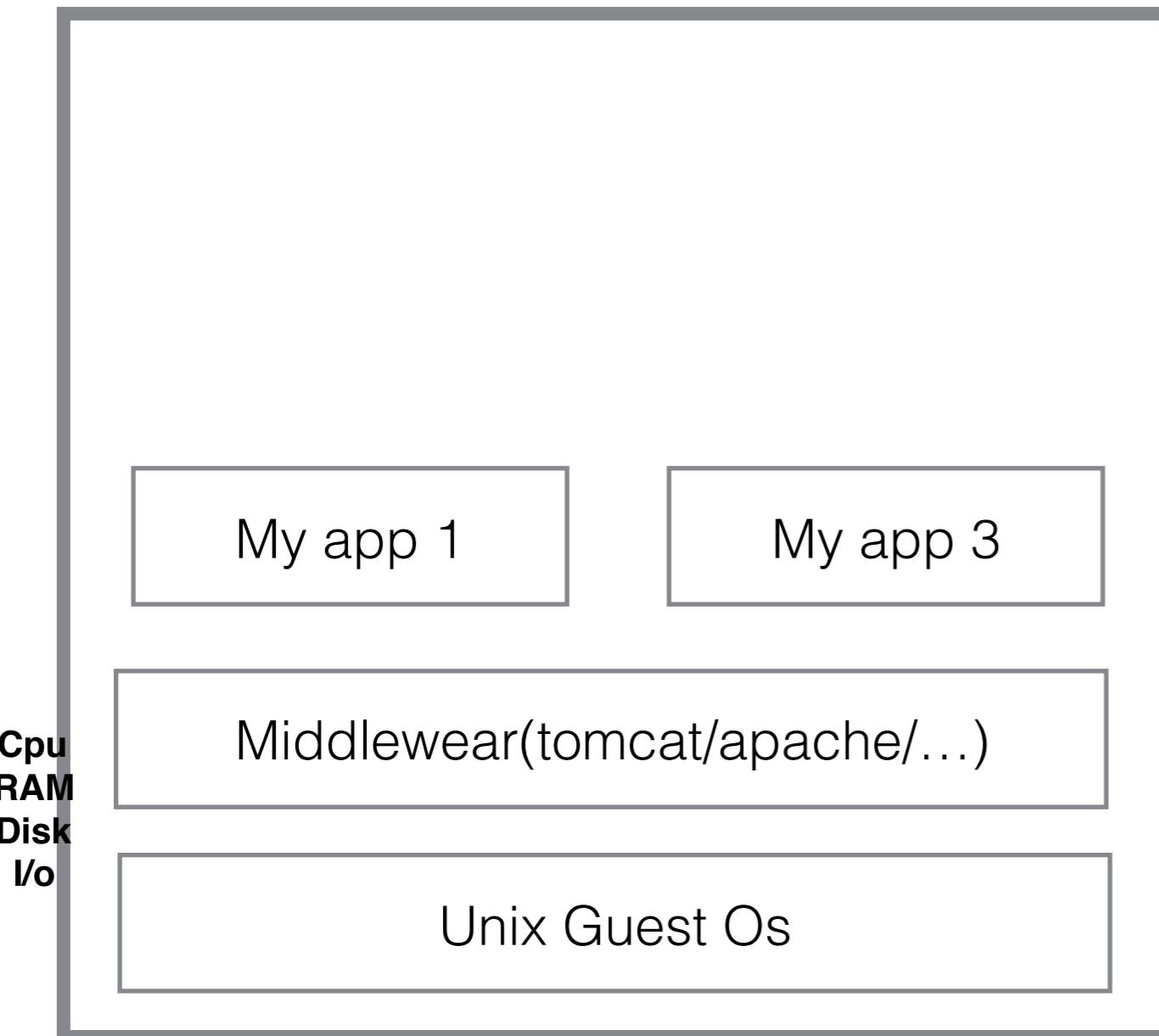


Cpu  
RAM  
Disk  
I/o



**Machine**

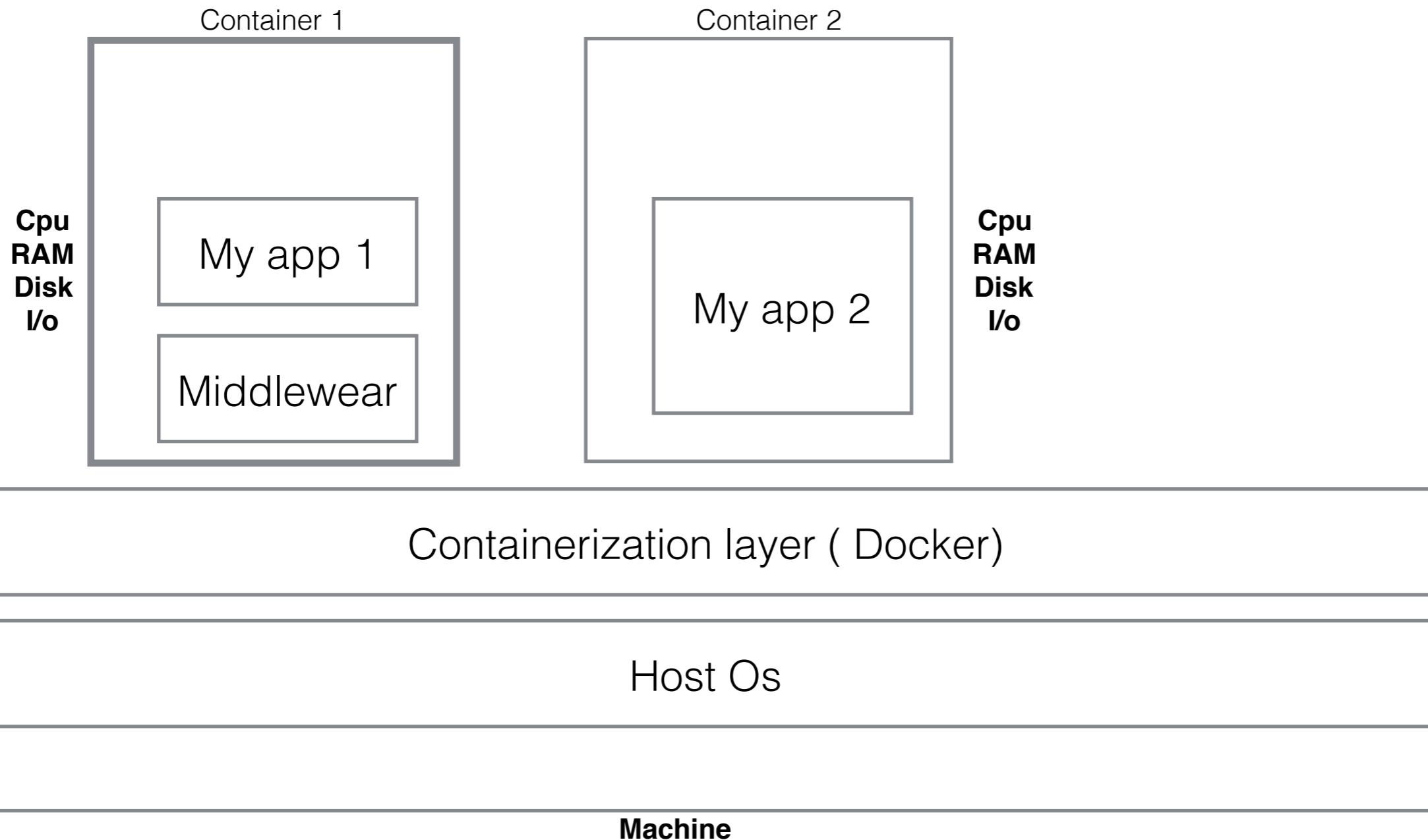
VM1



VM2



**Machine**



VM1

VM2

Container 1 8080

Container 2

My app 1

My app 2

Cpu  
RAM  
Disk  
I/o

Middlewear

Middlewear

Cpu  
RAM  
Disk  
I/o

Containerization layer ( Docker)

Cpu  
RAM  
Disk  
I/o

Unix Guest Os

Win Guest Os

Cpu  
RAM  
Disk  
I/o

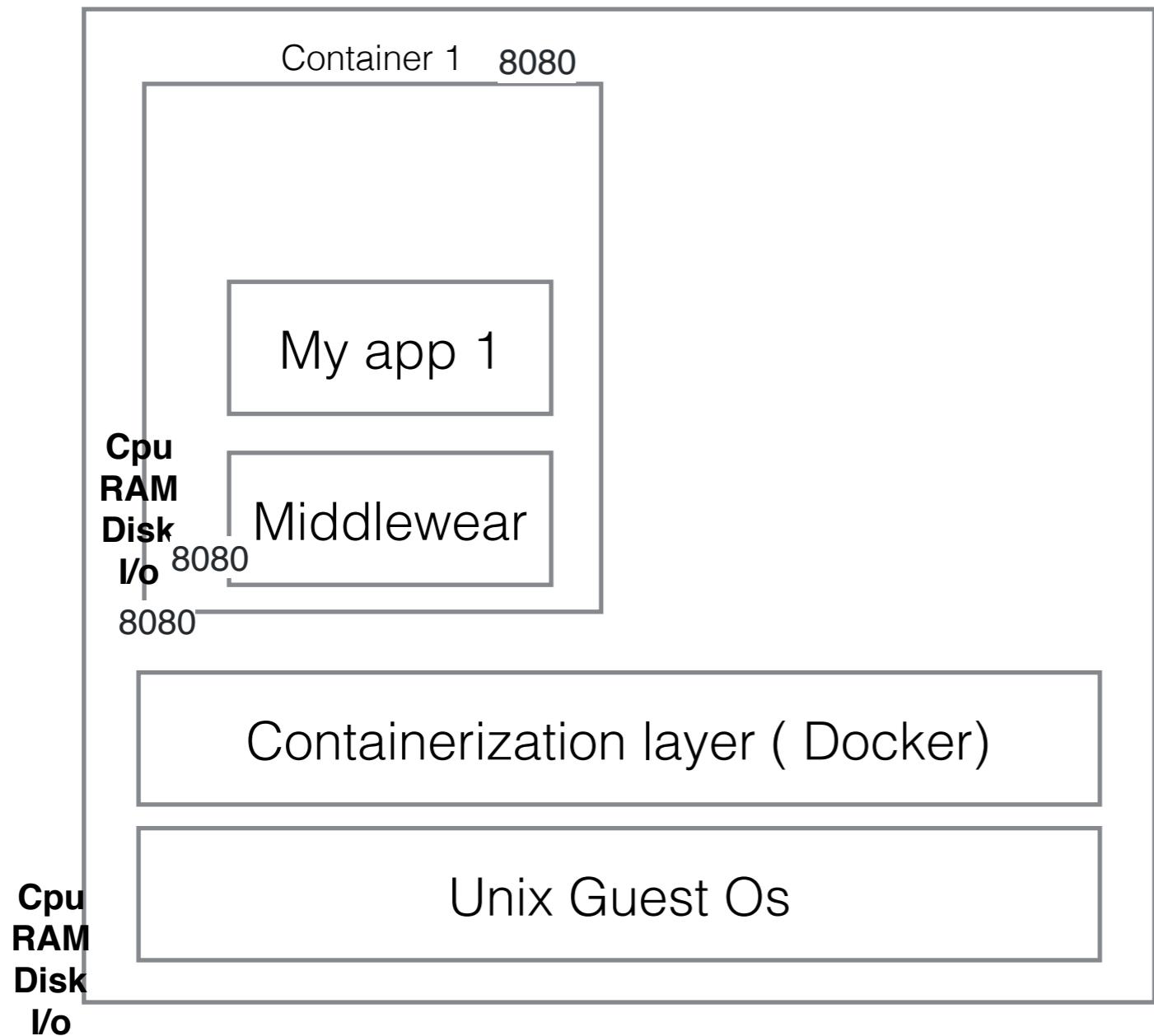
Virtualisation layer (Hyperviser)

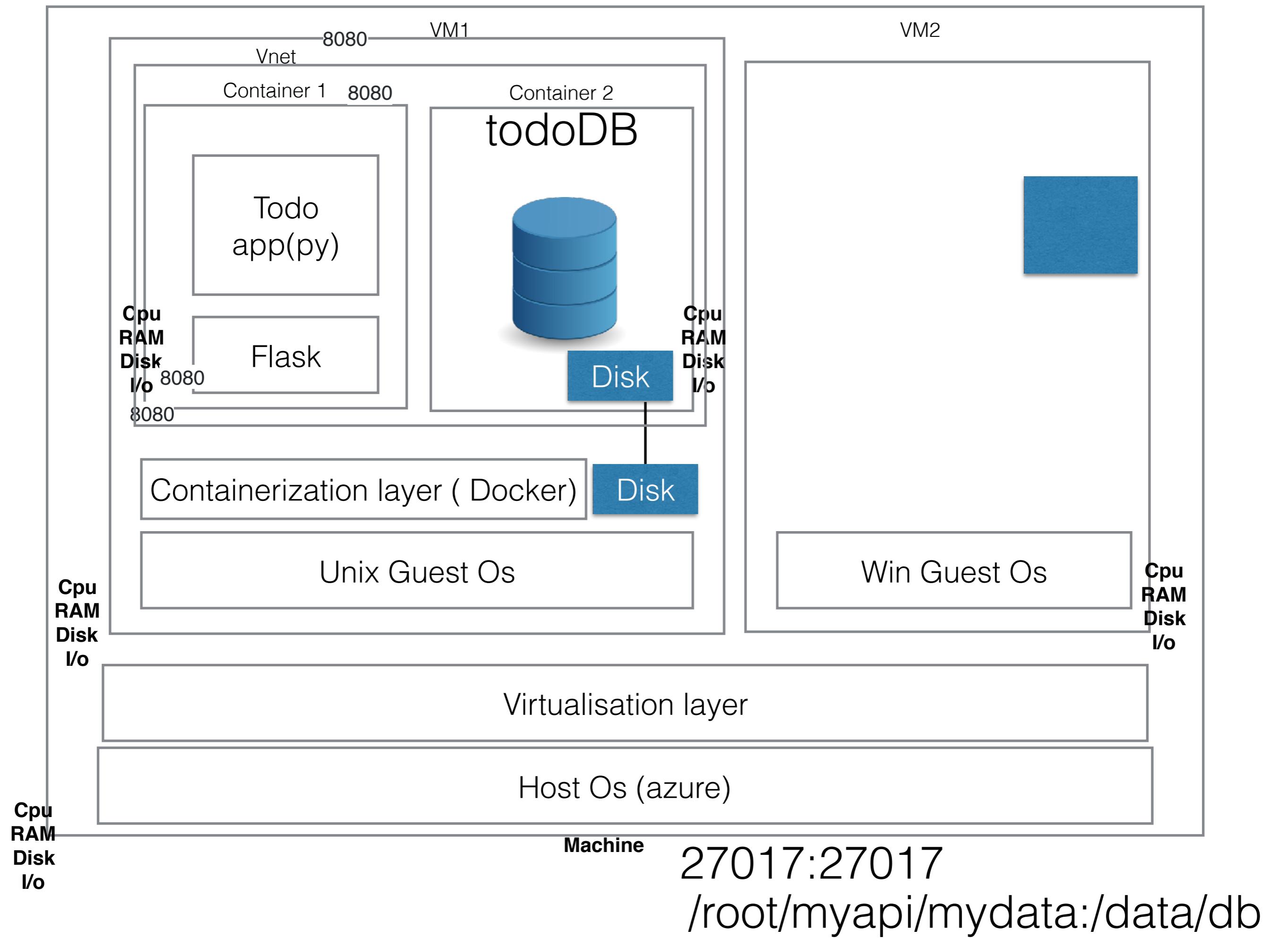
Host Os (azure)

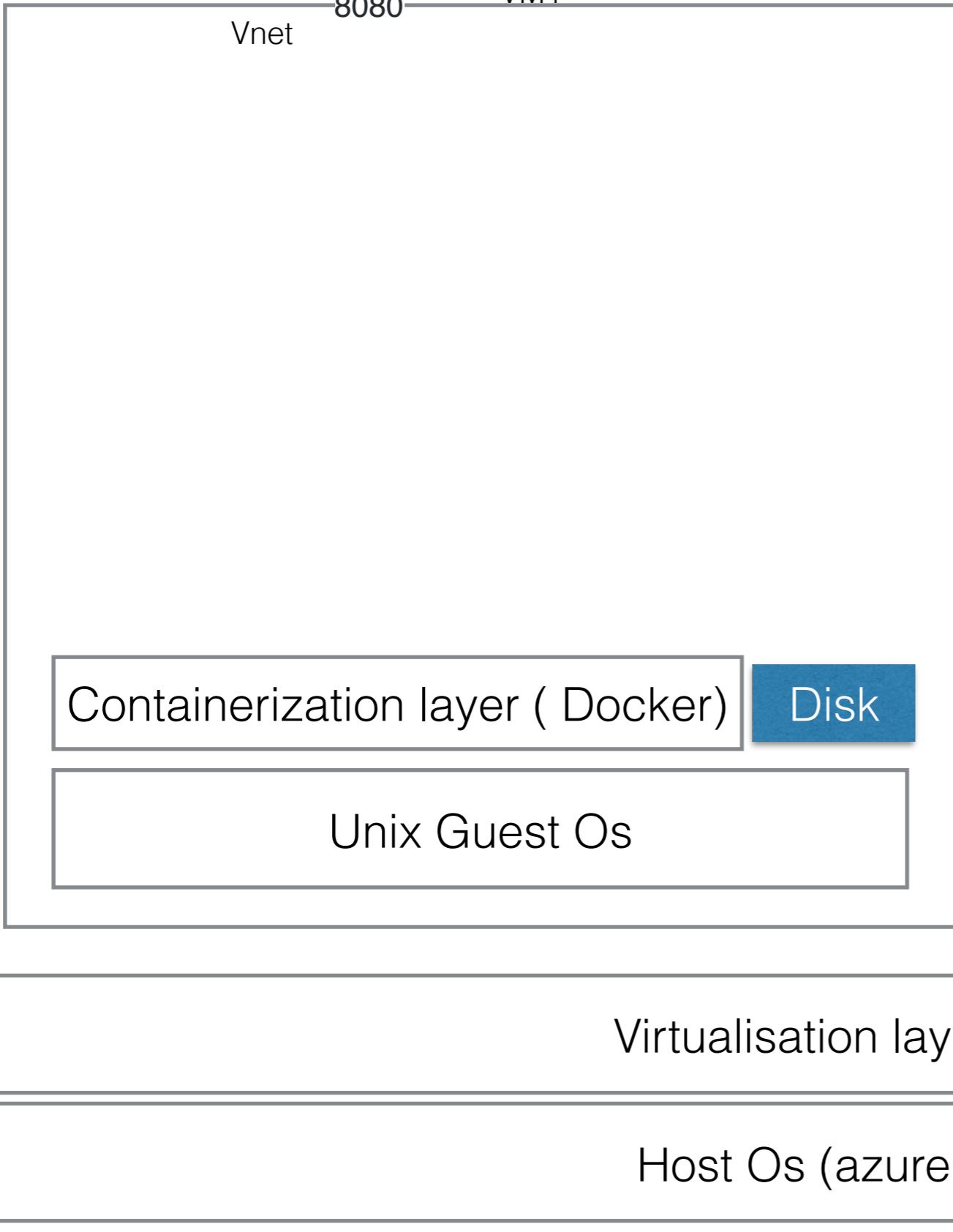
Cpu  
RAM  
Disk  
I/o

Machine

VM1







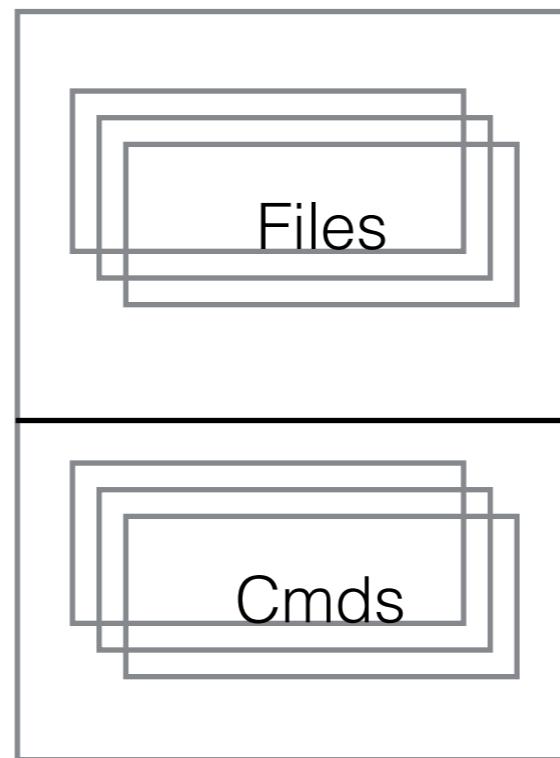
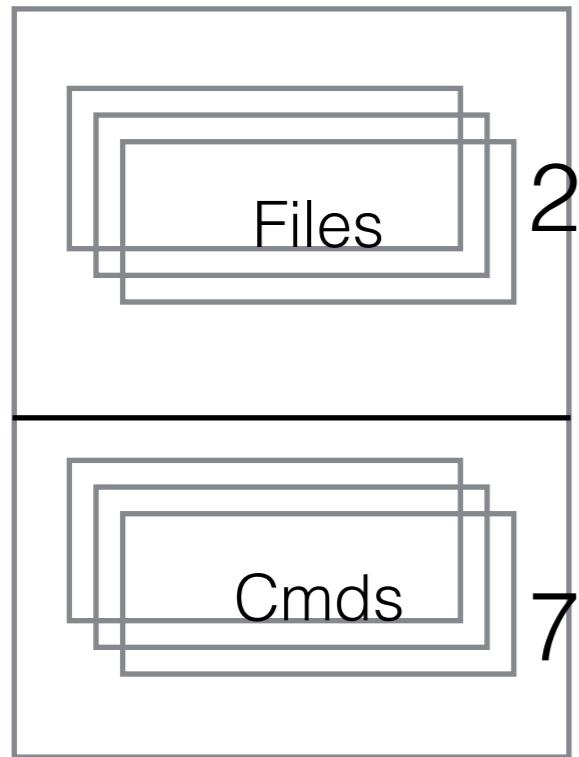
Machine

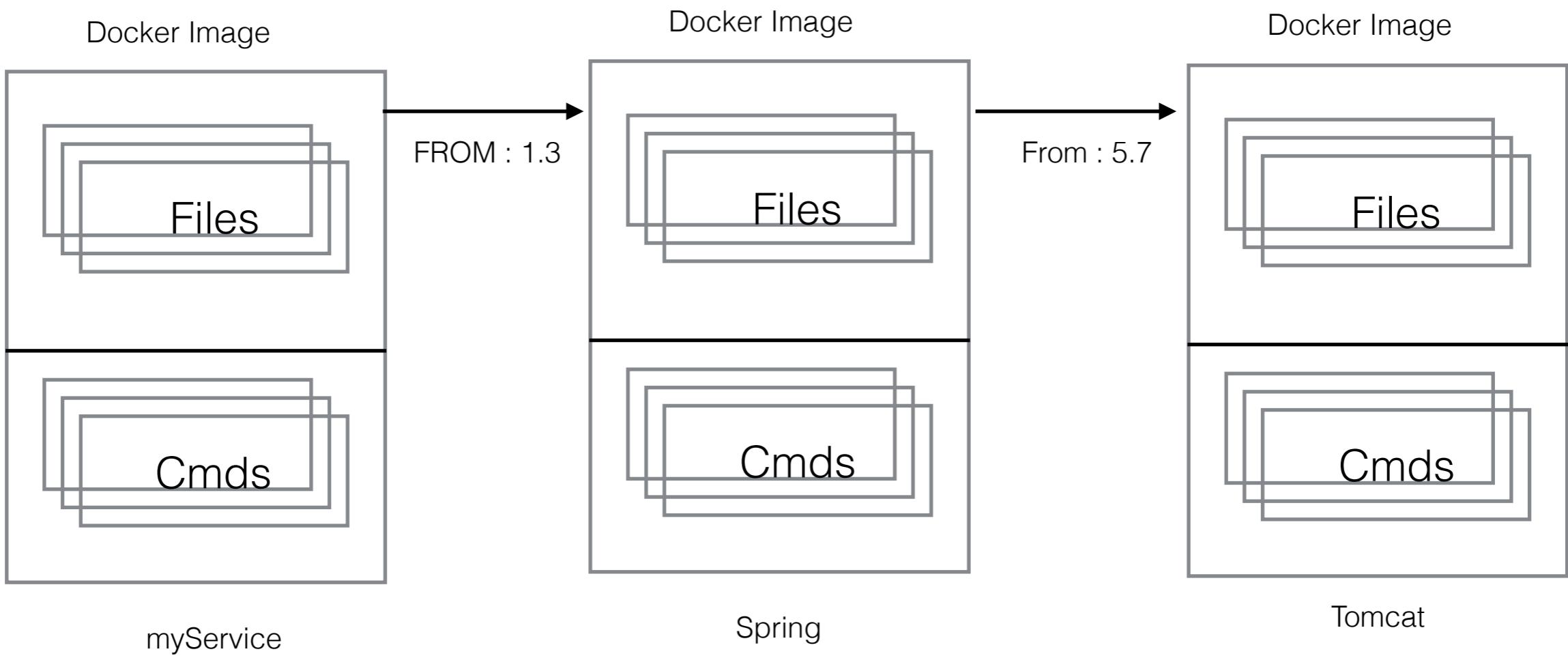
27017:27017

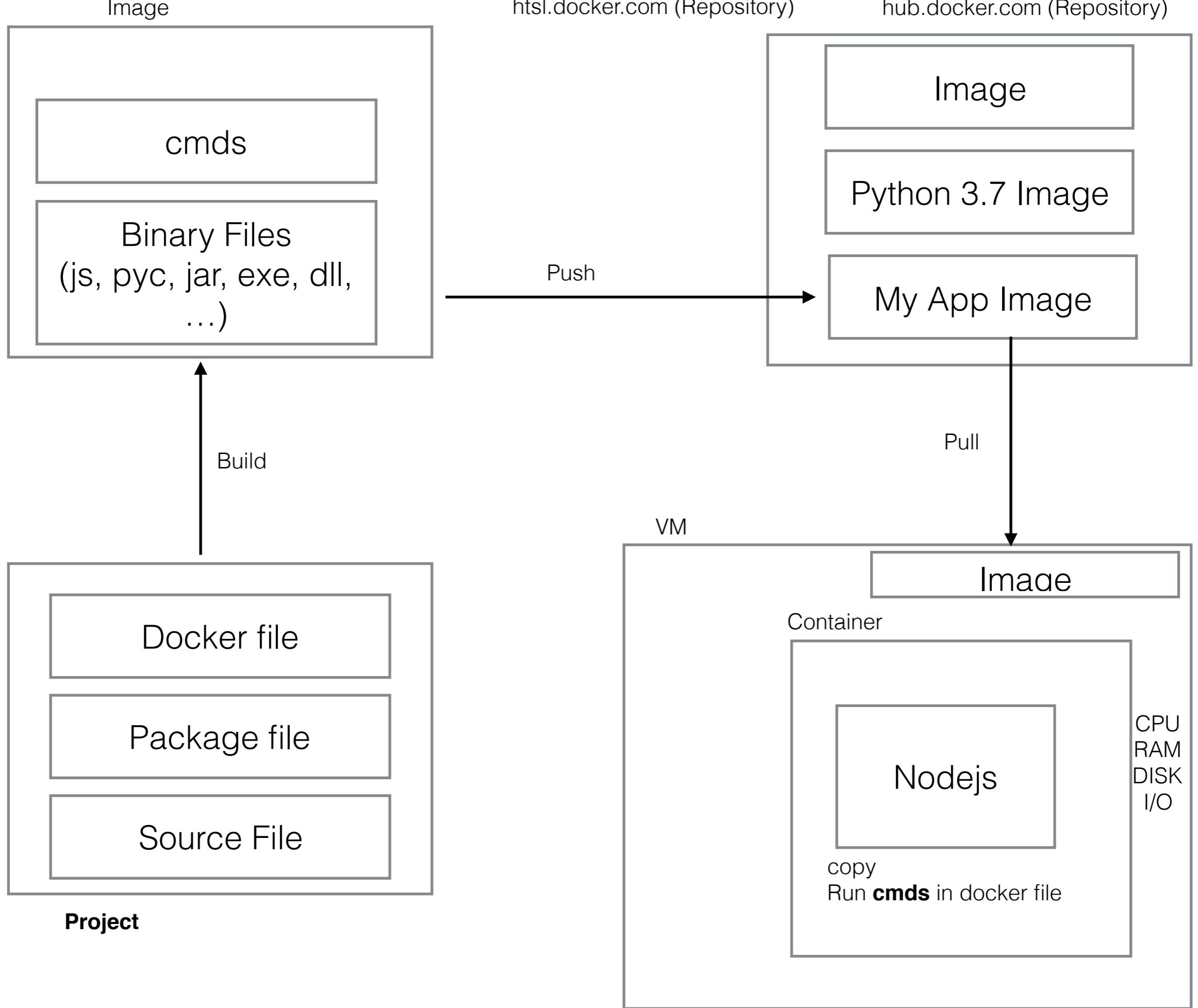
/root/myapi/mydata:/data/db

# Reproducible Env.

setup/msi/war/..



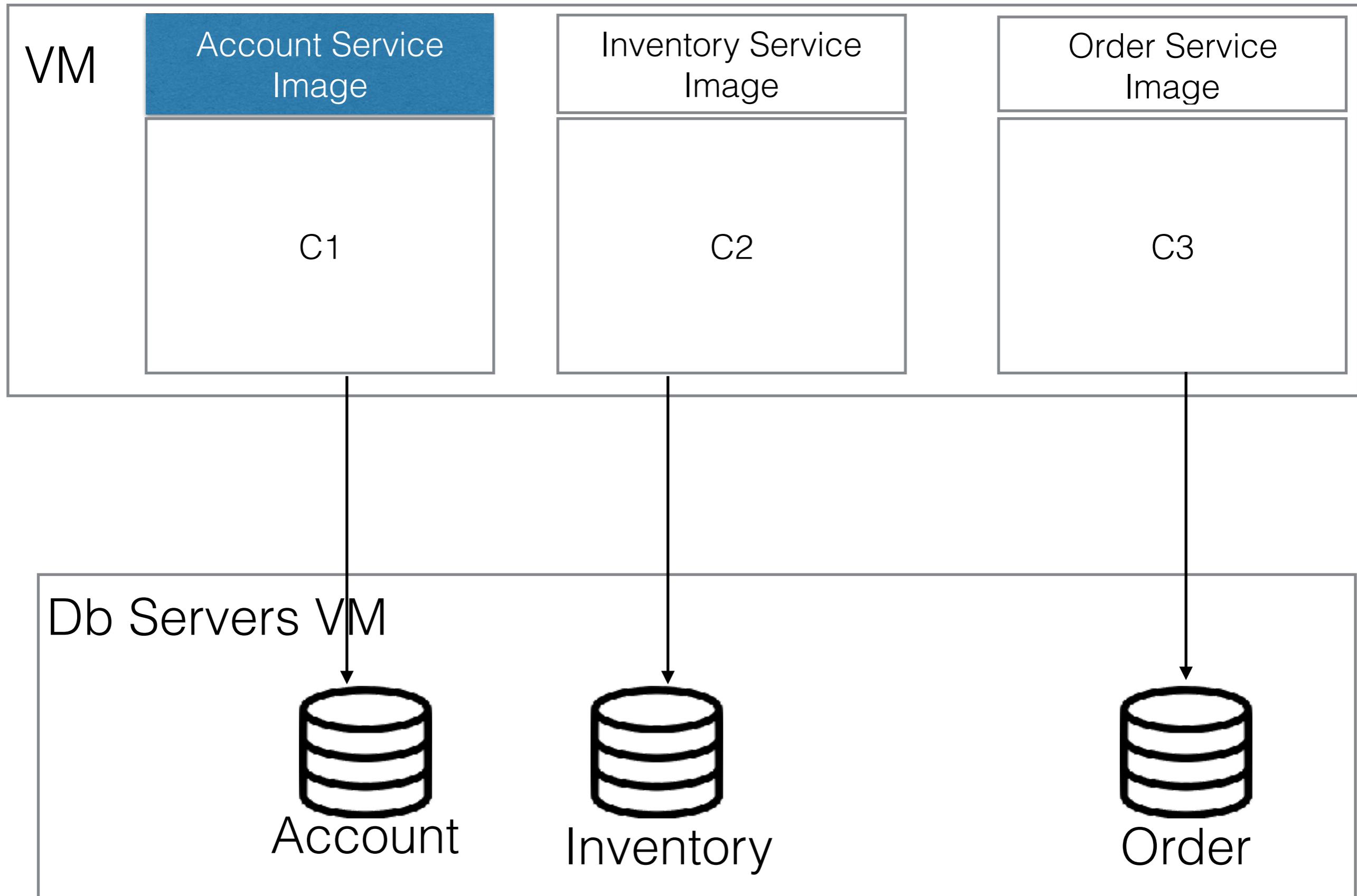




Account Service  
Image

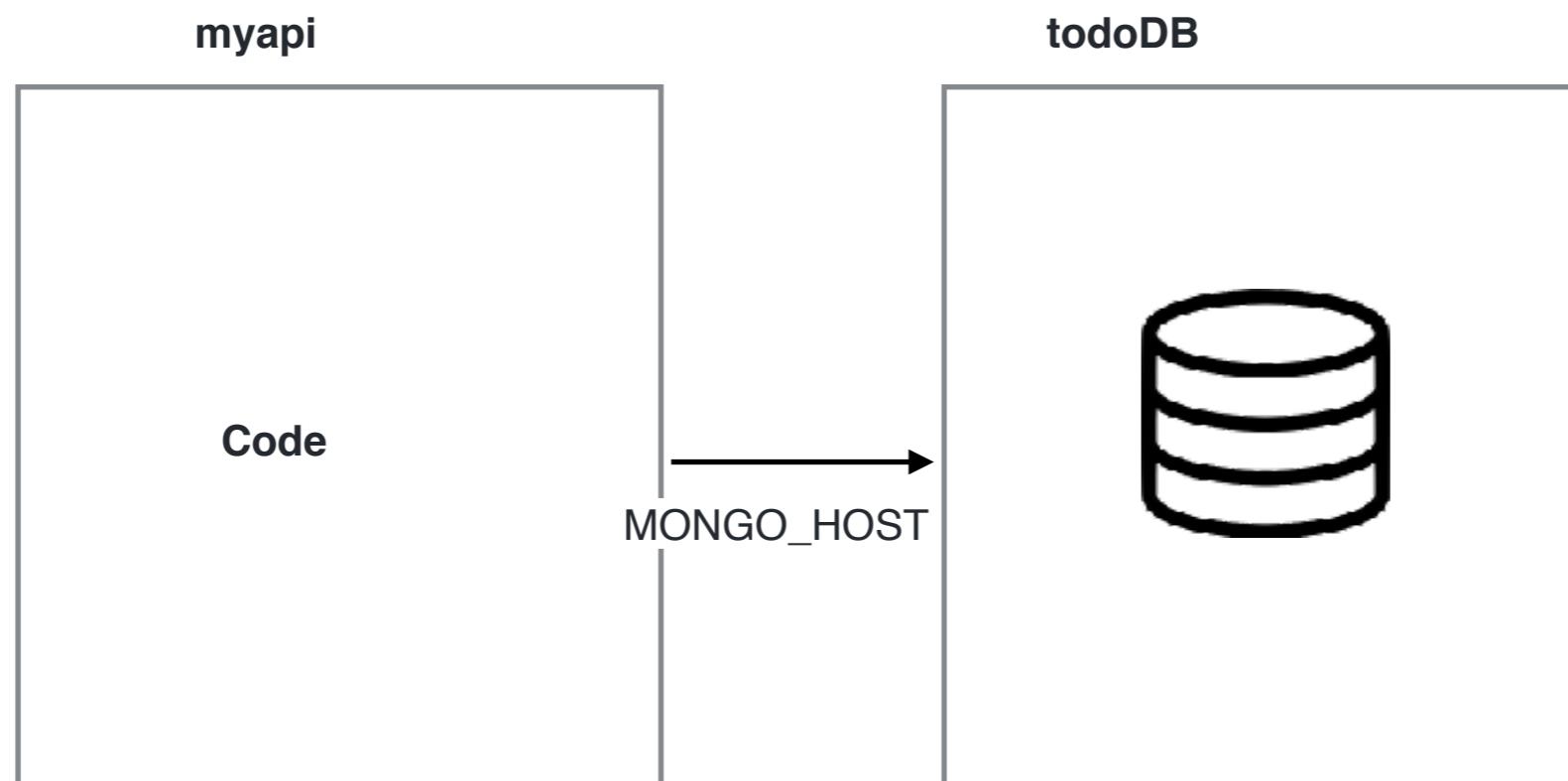
Inventory Service  
Image

Order Service  
Image

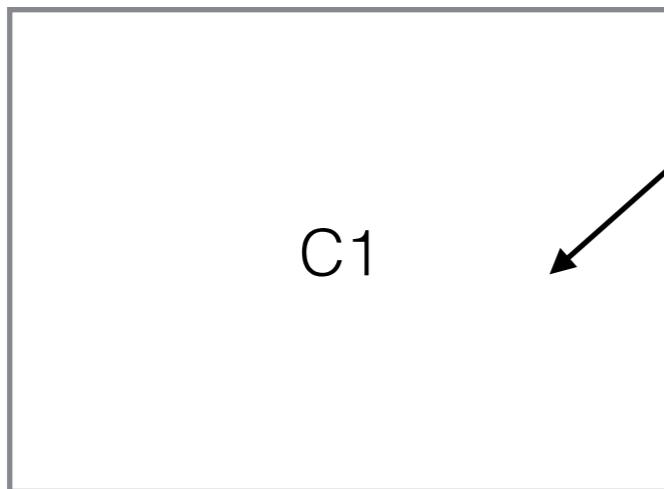


VM

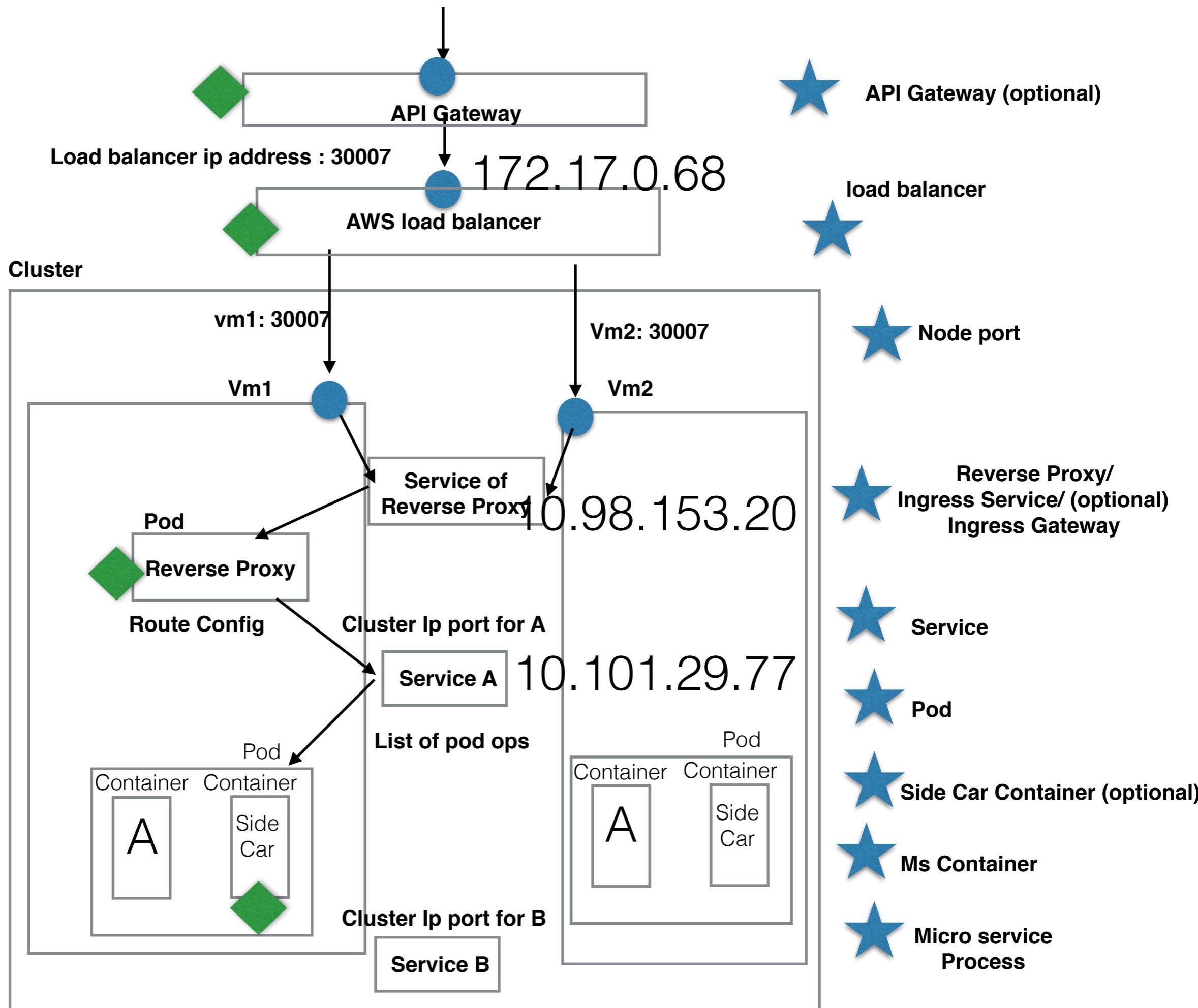
# vNetwork



Vendor image  
Docker file (cmds)



Custom image  
Docker file (cmds)



# Day 3

# Managing Container Life cycle

VM1

VM2

Container 1

Container 2

Container 3

My app 1

My app 2

My app 3

Middlewear

Middlewear

Middlewear

Cpu  
RAM  
Disk  
I/o

Cpu  
RAM  
Disk  
I/o

Cpu  
RAM  
Disk  
I/o

Containerization layer ( Docker)

Unix Guest Os

Cpu  
RAM  
Disk  
I/o

Virtualisation layer

Host Os (azure)

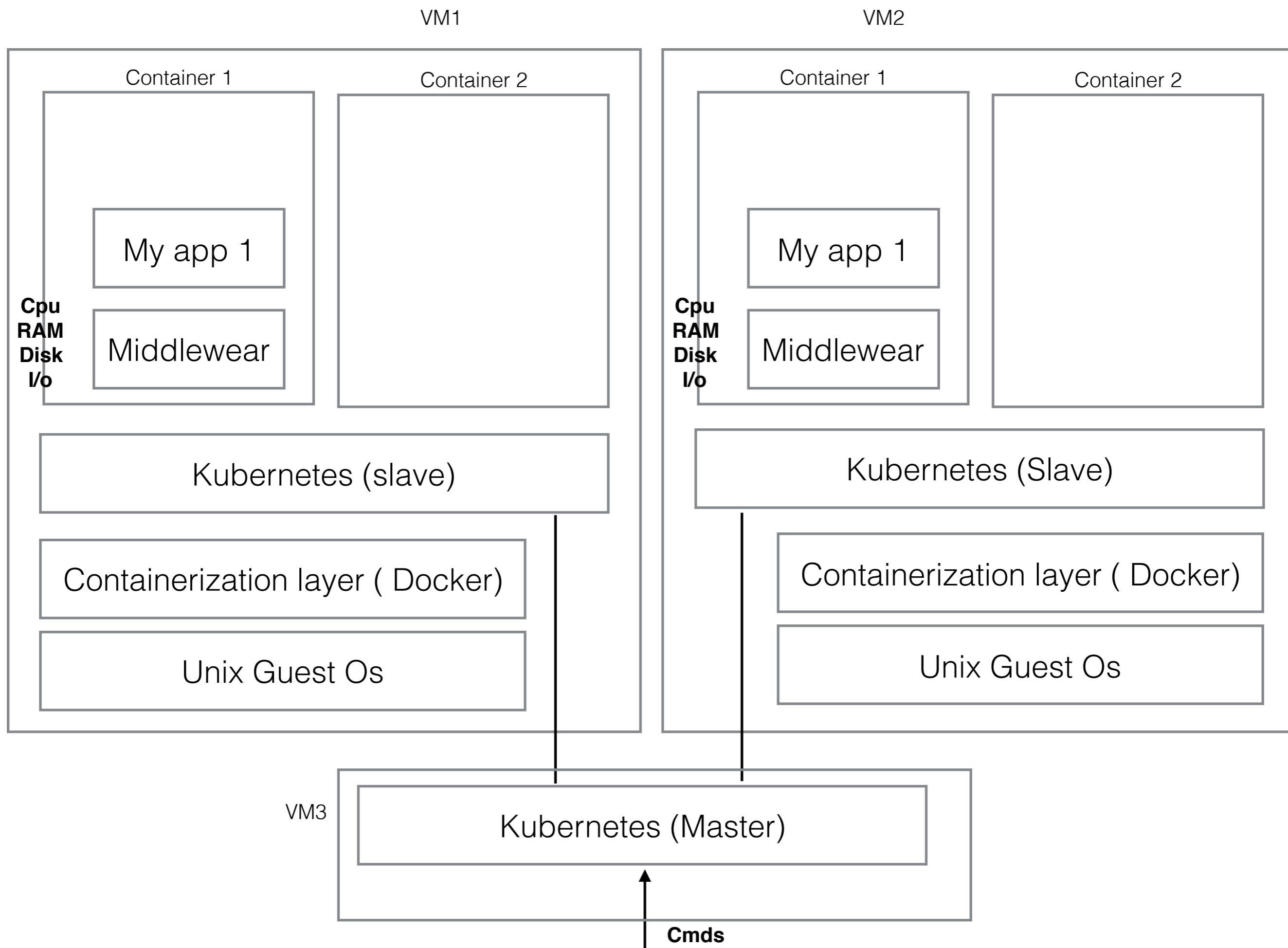
**Machine**

Kubernetes (slave)

Kubernetes (Slave)

Kubernetes (Master)





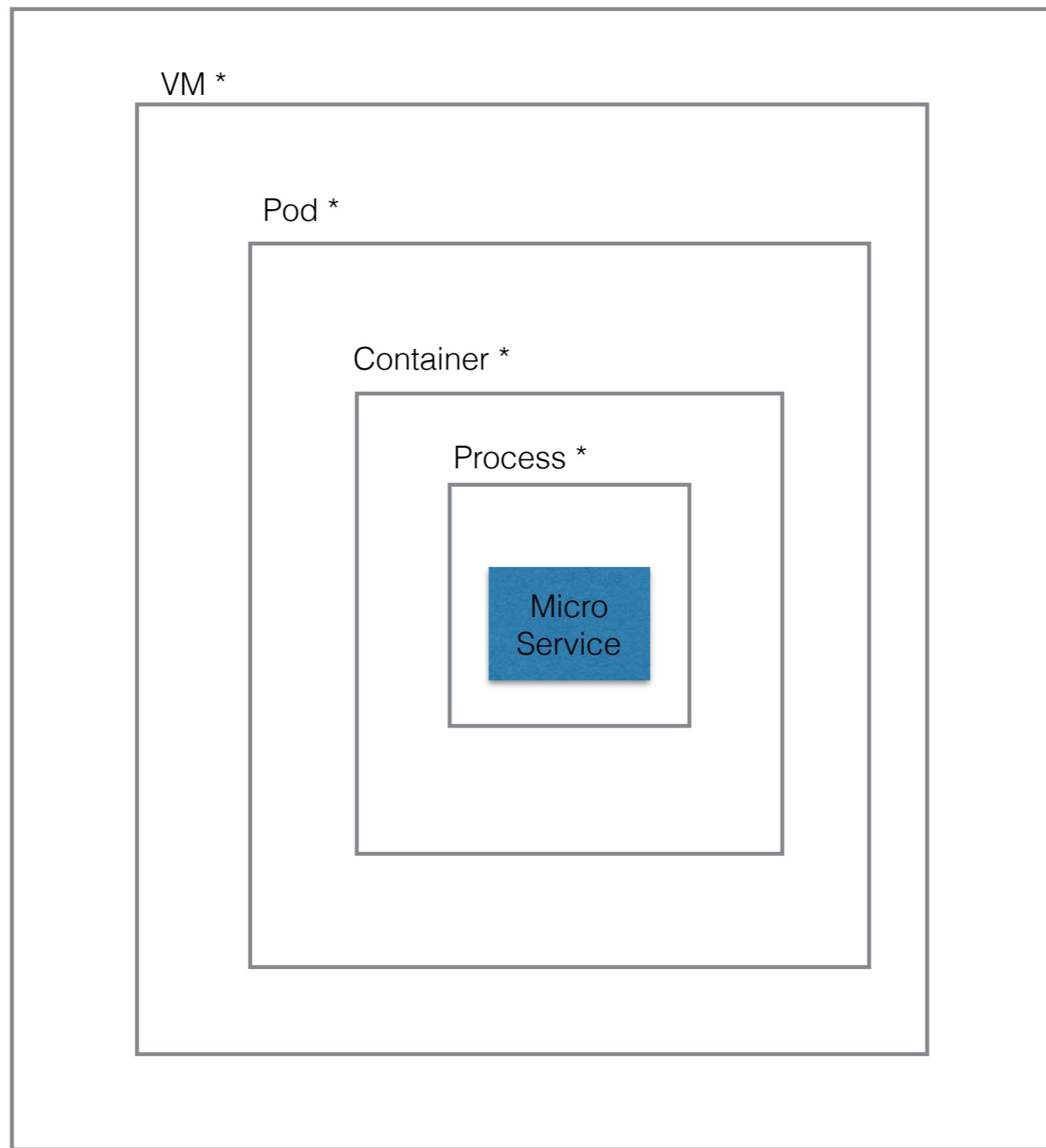
VM1

Cpu  
RAM  
Disk  
I/o

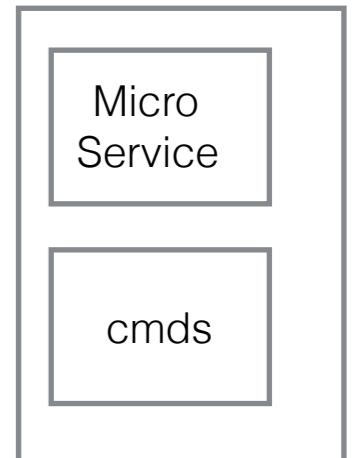
Containerization layer ( Docker)

Unix Guest Os

Cluster



Docker Image



Docker /k8s

Cluster

Node (VM)

Pod

Container

Process

Micro  
Service : v1

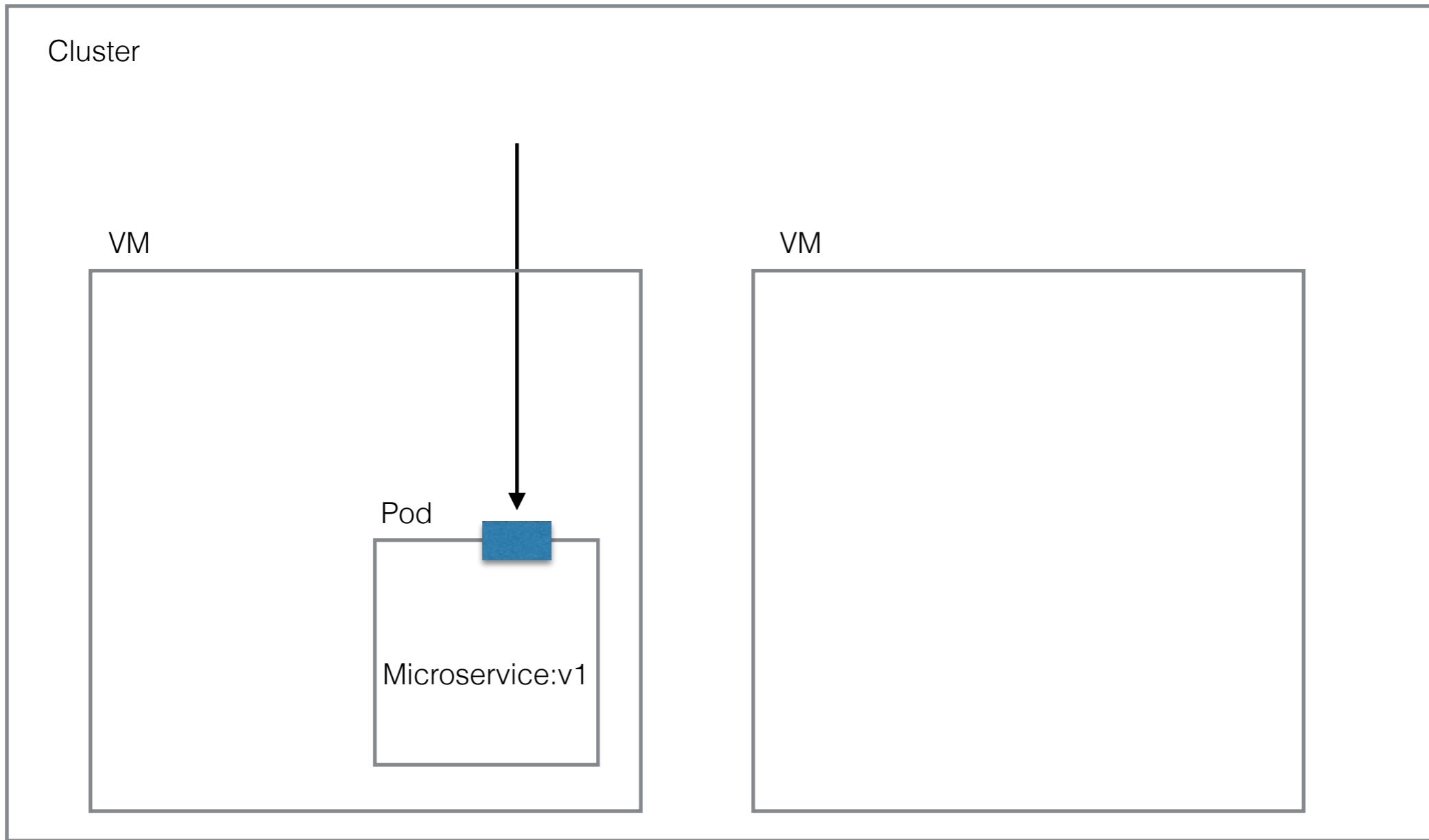
Pod

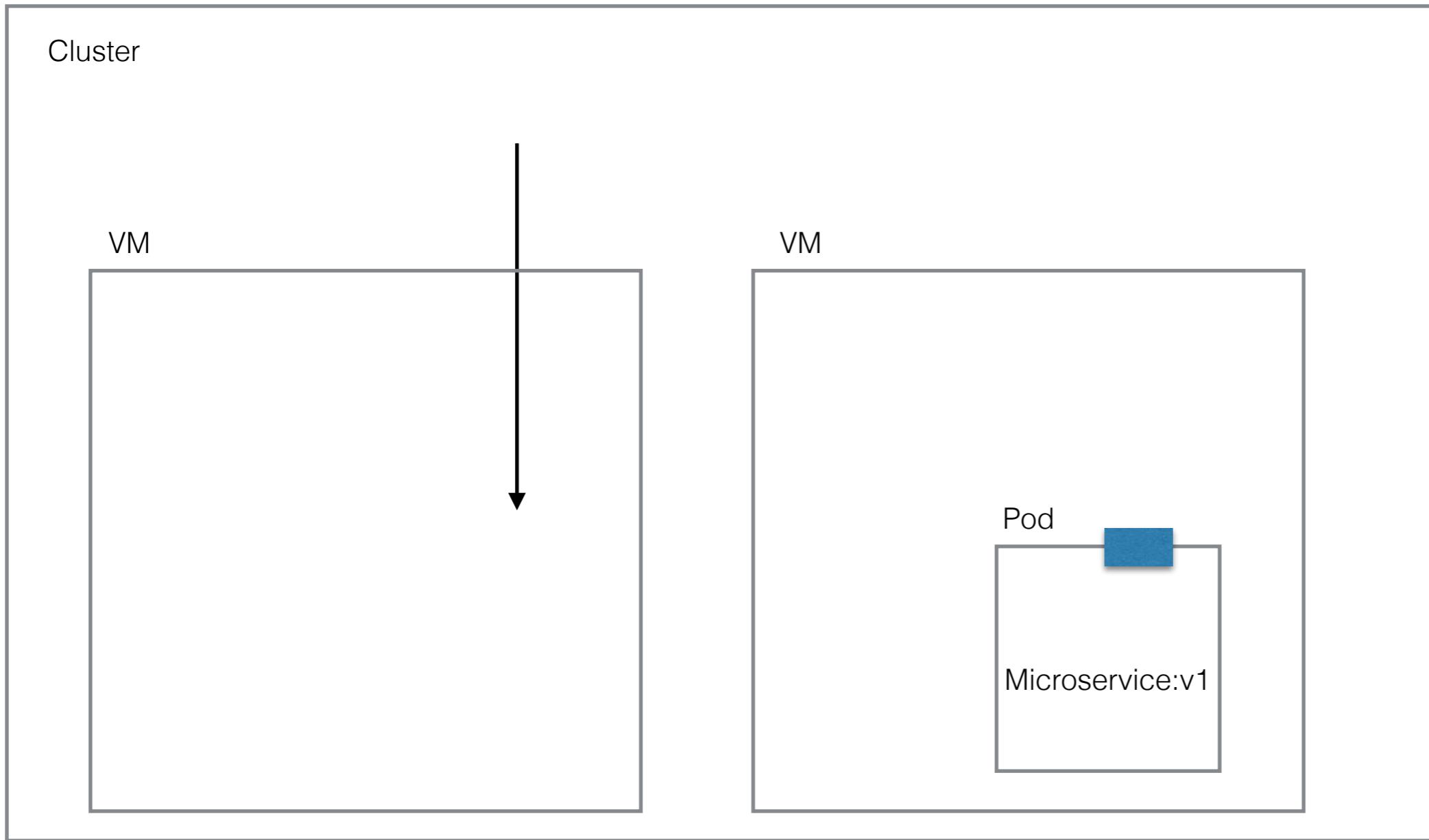
Container

Process

Micro  
Service : v2

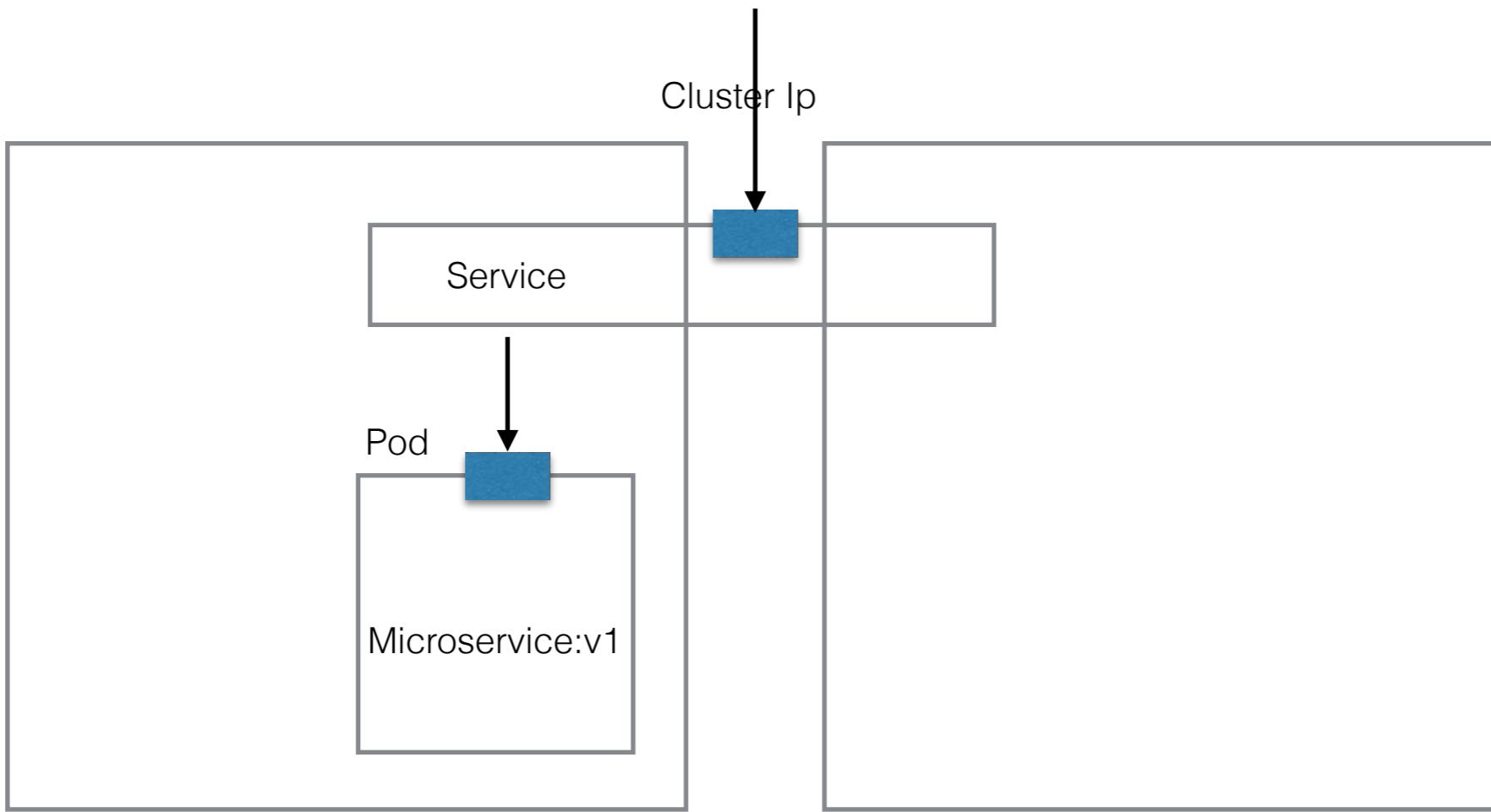




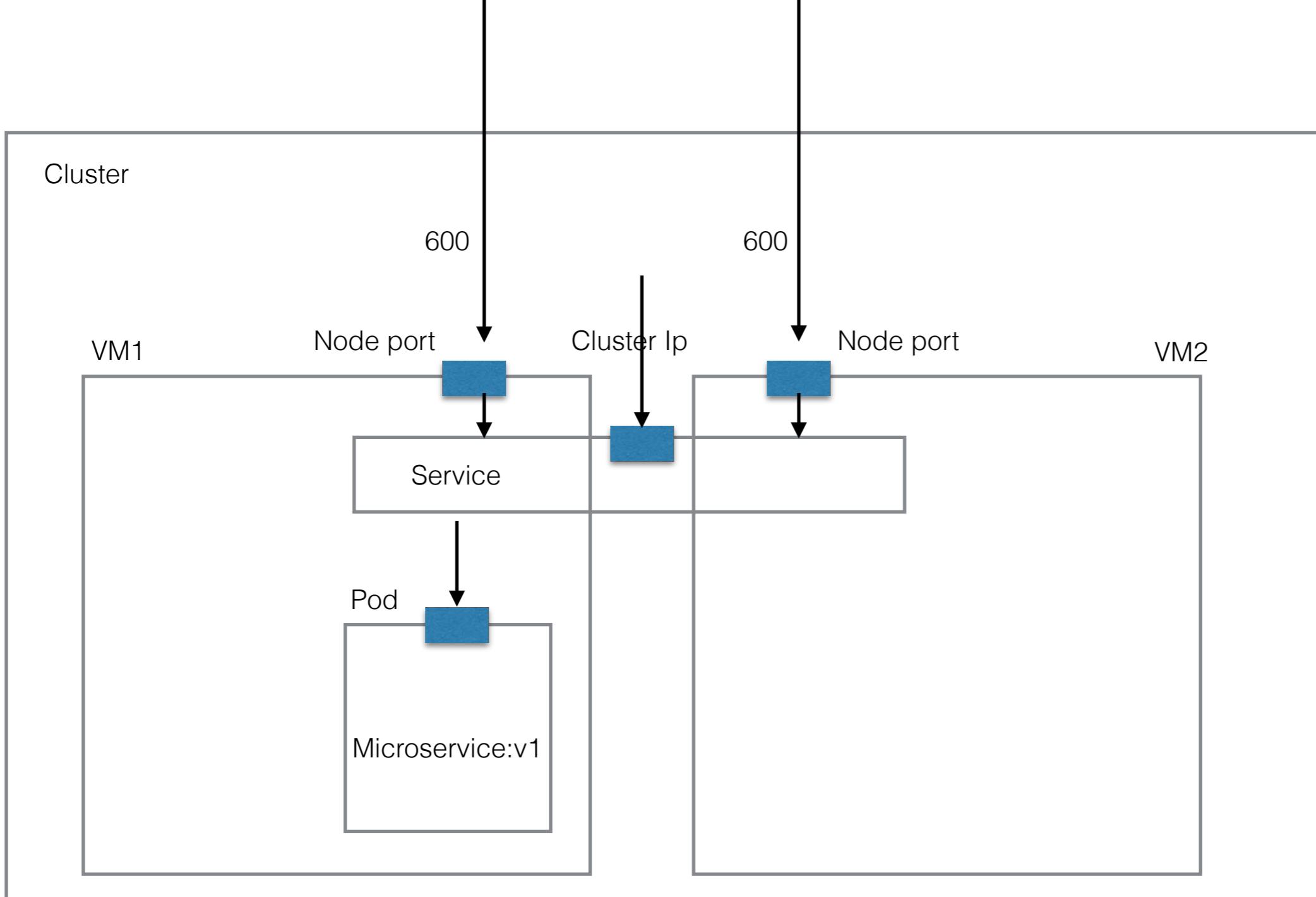


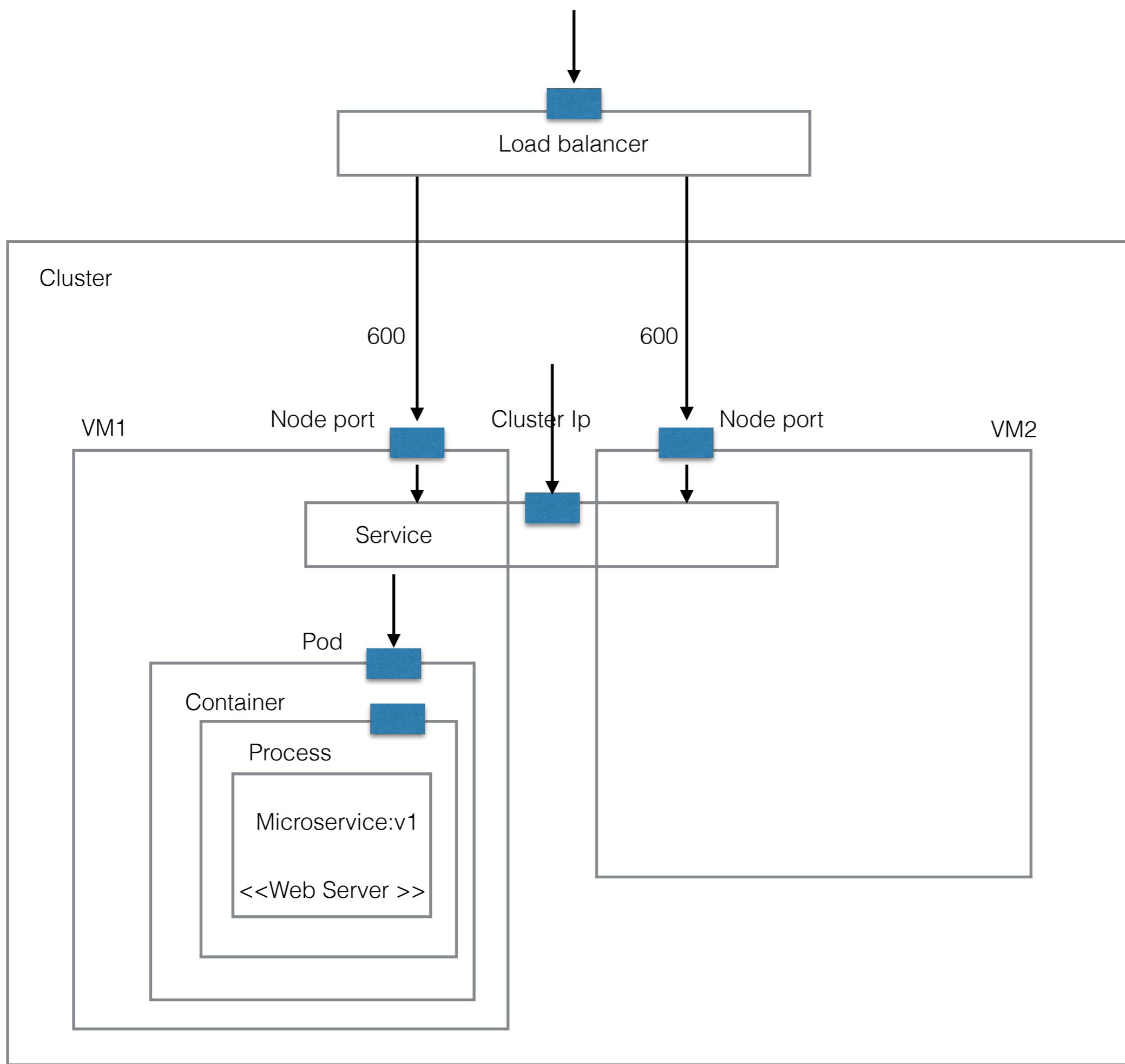
Cluster

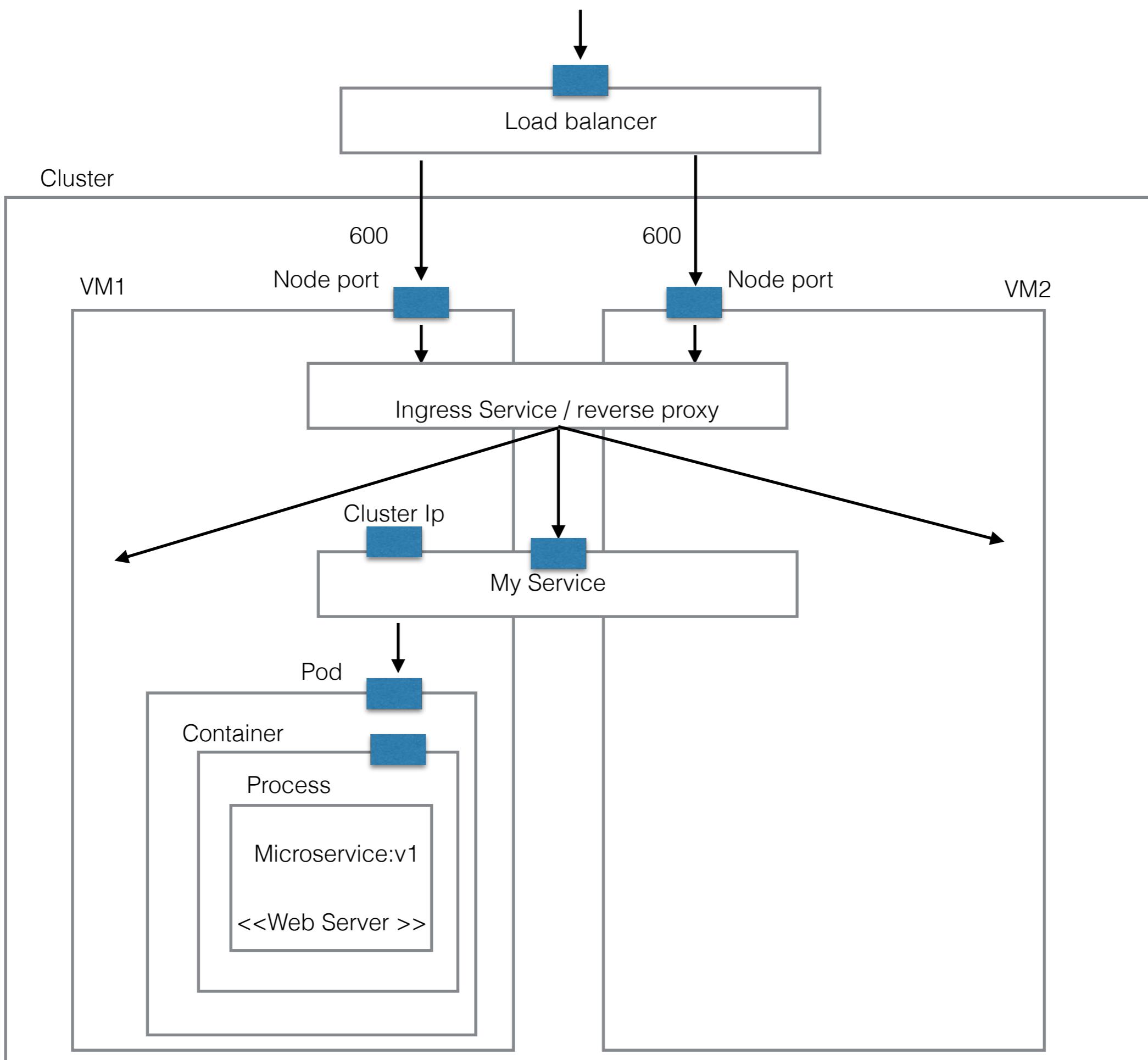
`curl http://10.104.204.83:8080/date`

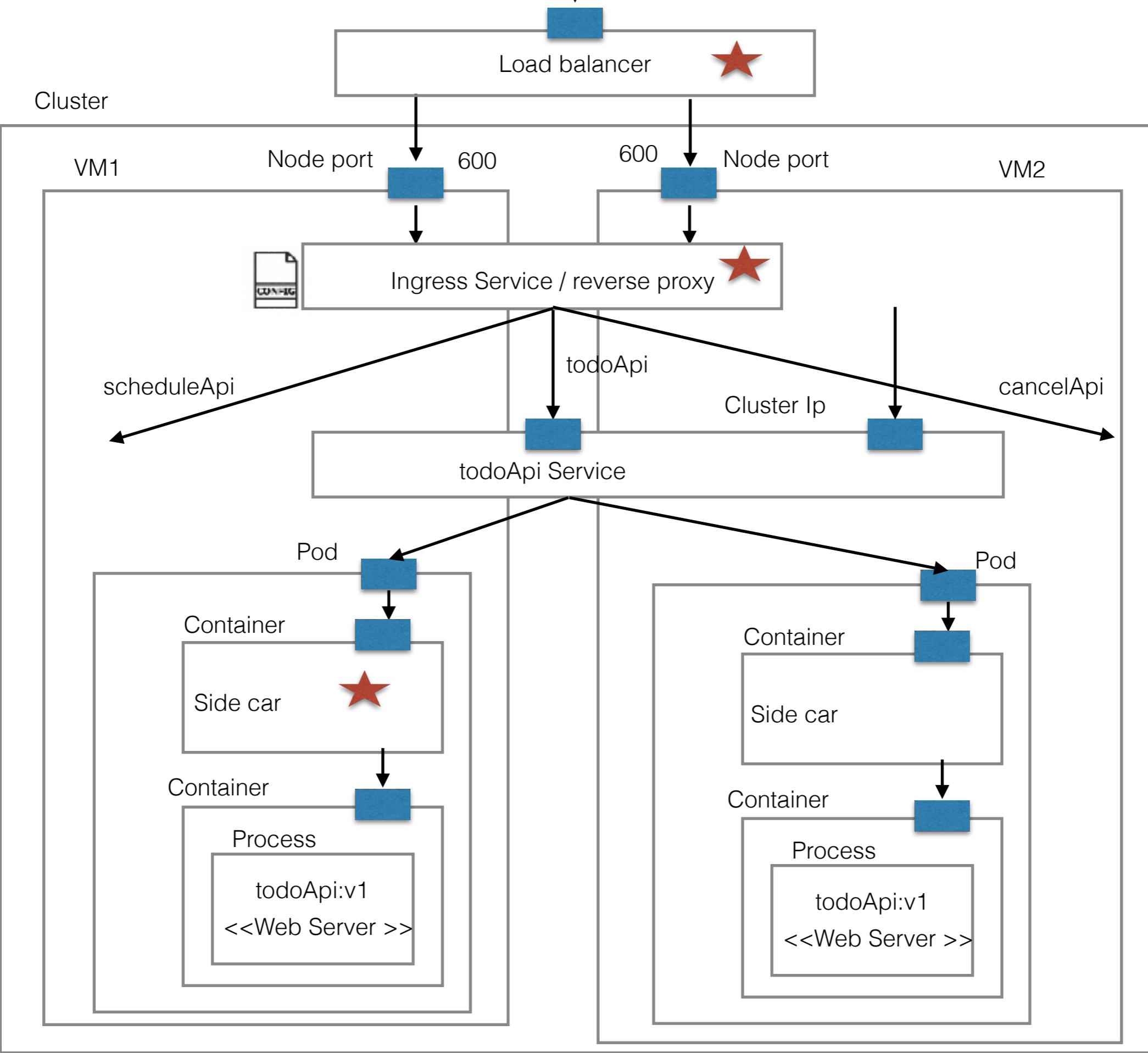


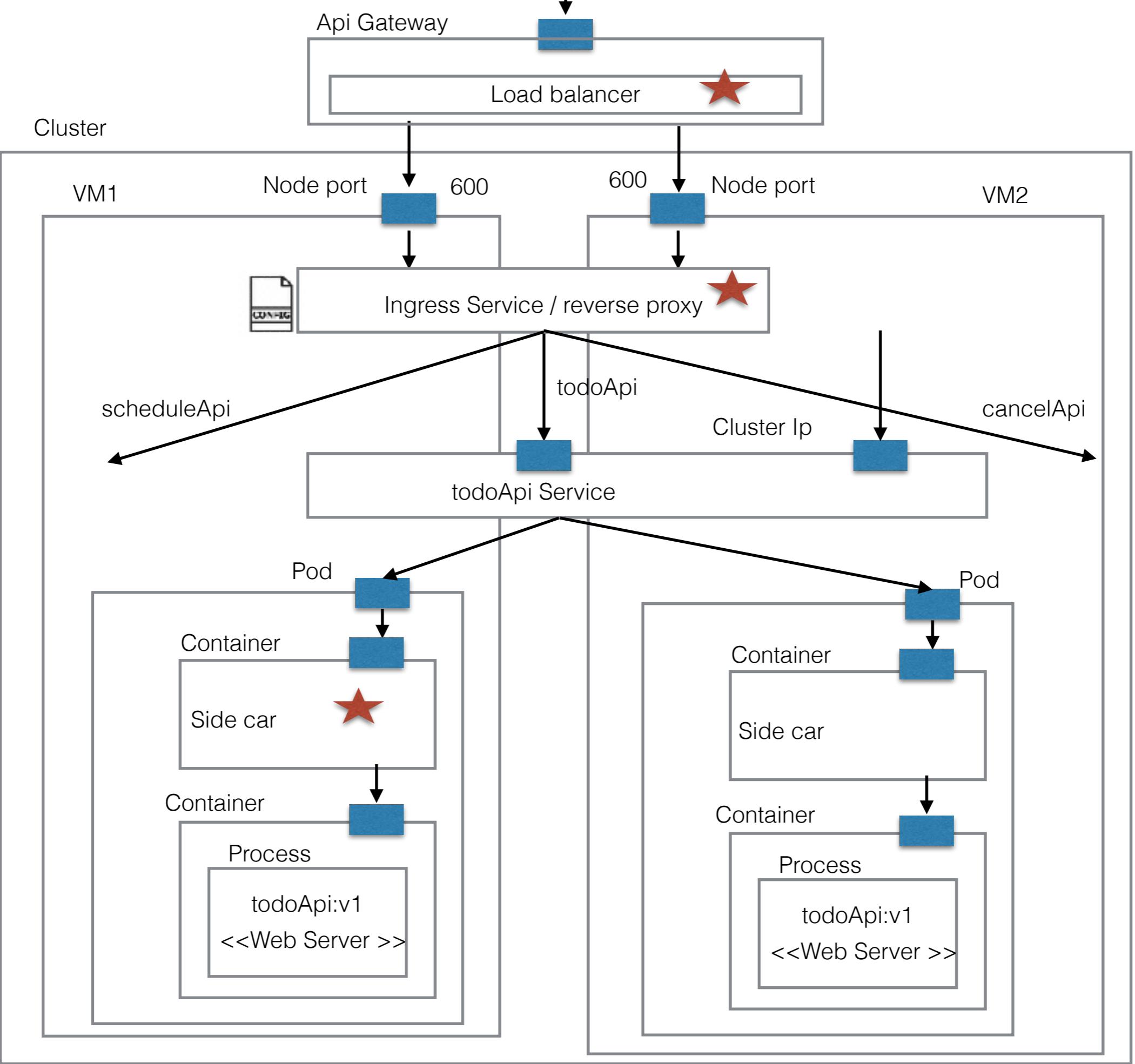
600 -> microserice:v1



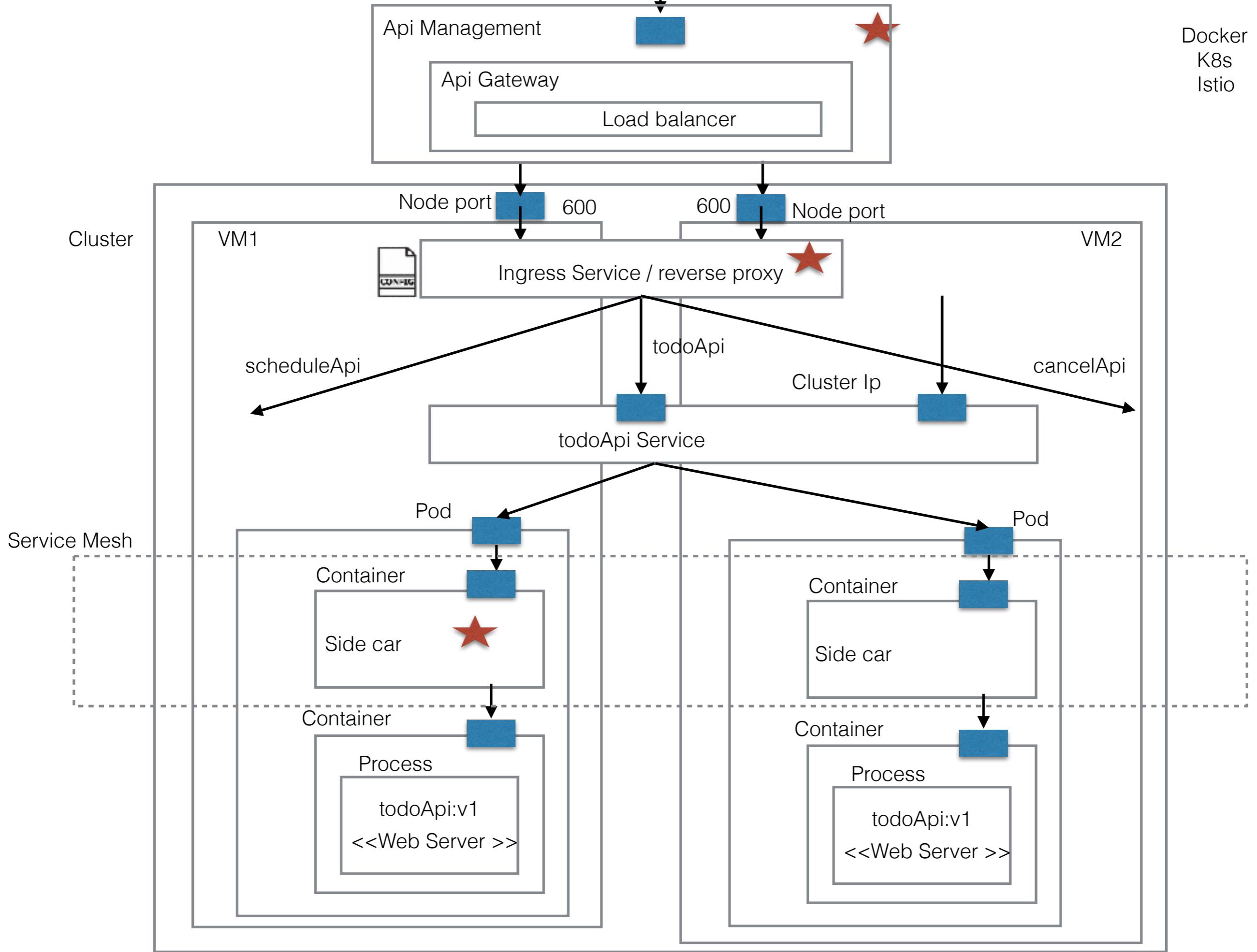






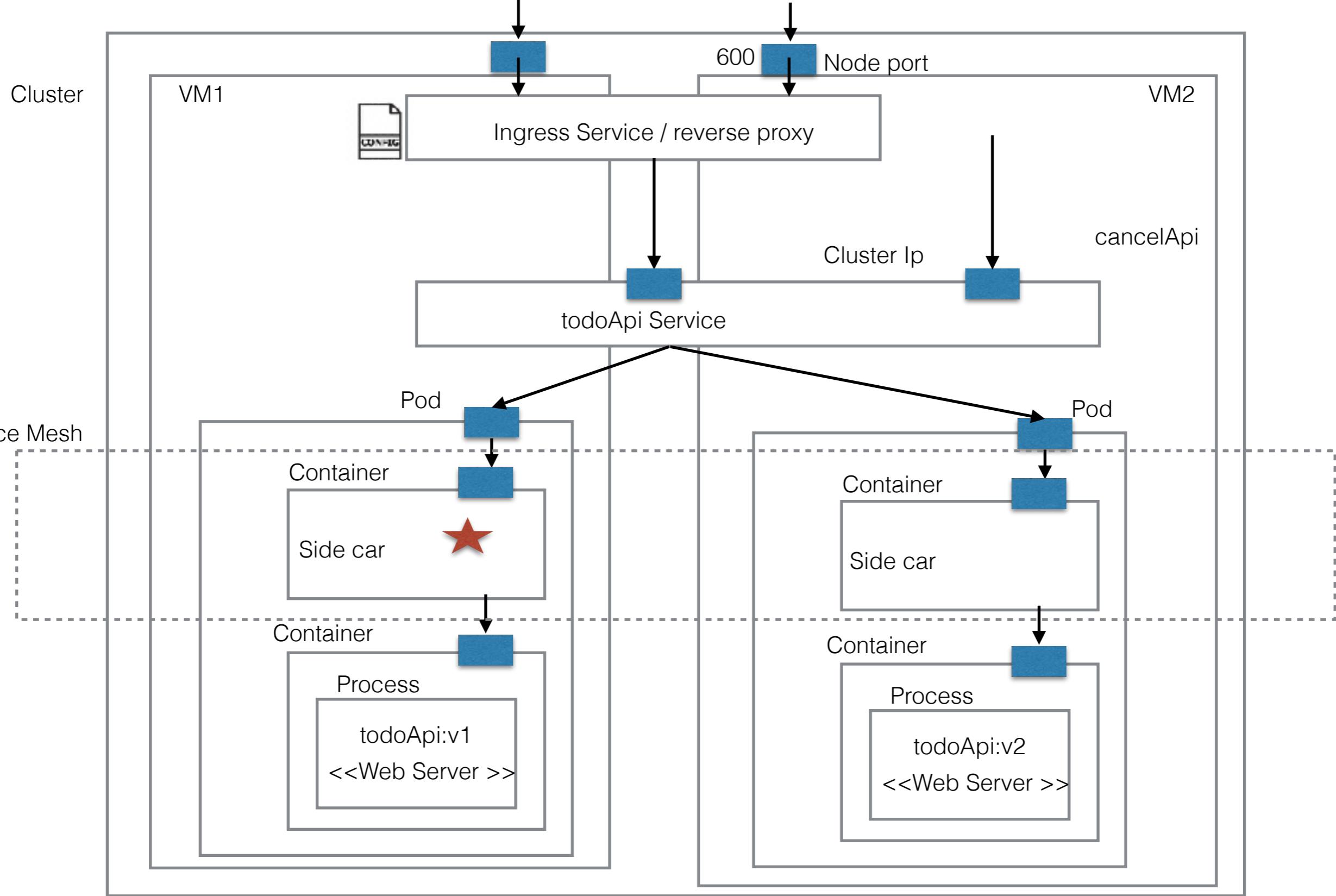


Docker  
K8s  
Istio

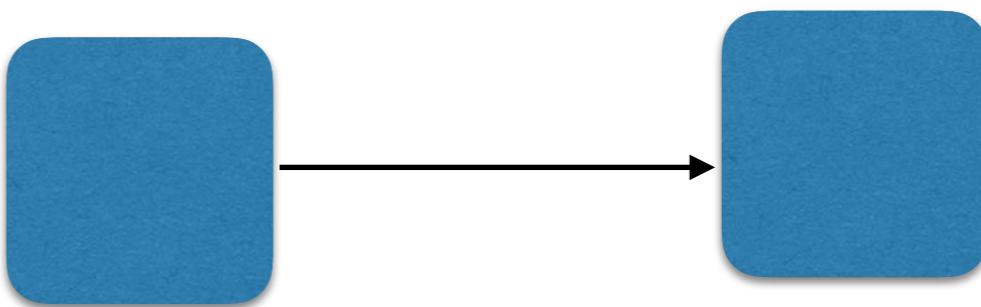


[https://ip\\_address:32637/date](https://ip_address:32637/date)

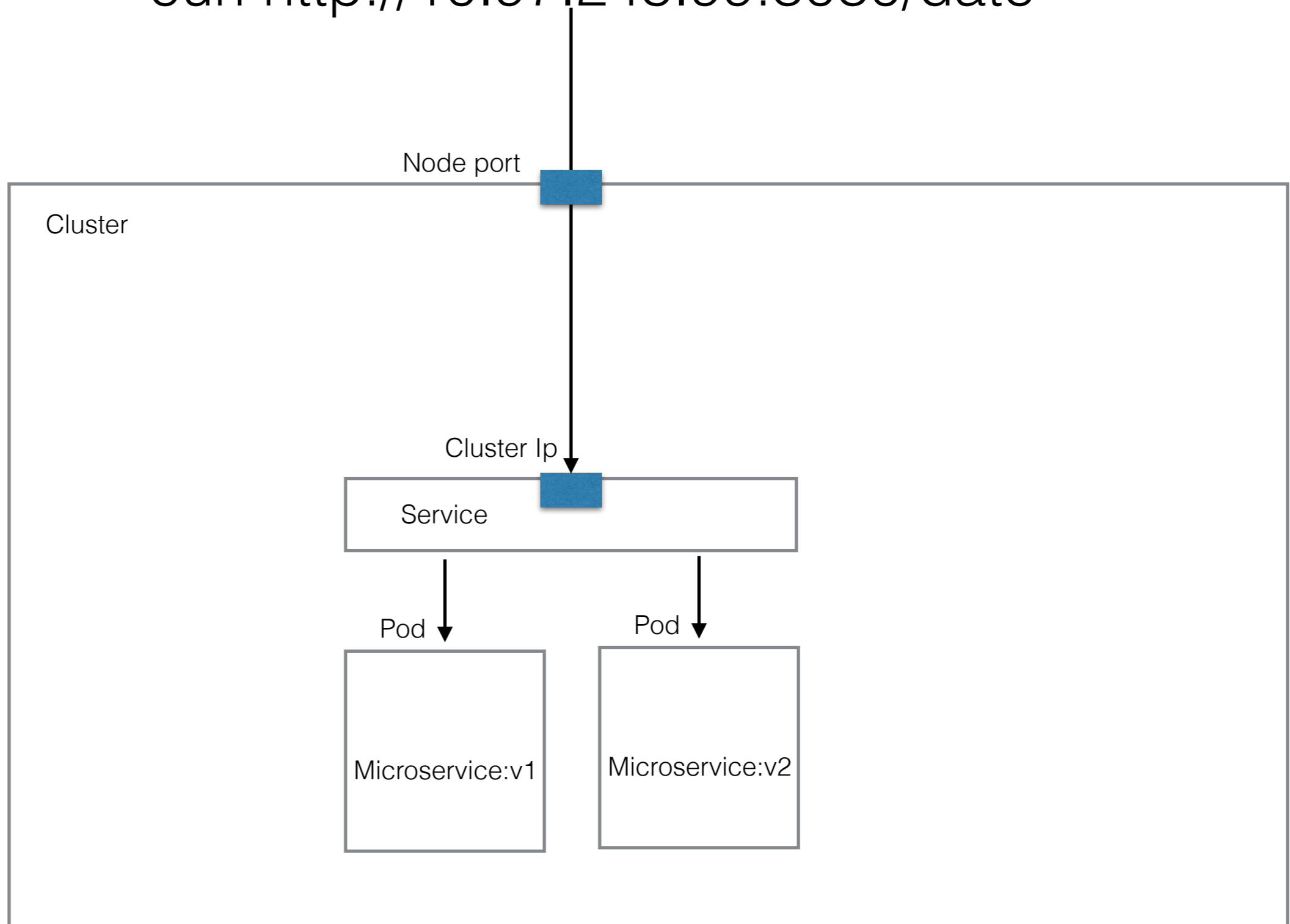
Docker  
K8s  
Istio

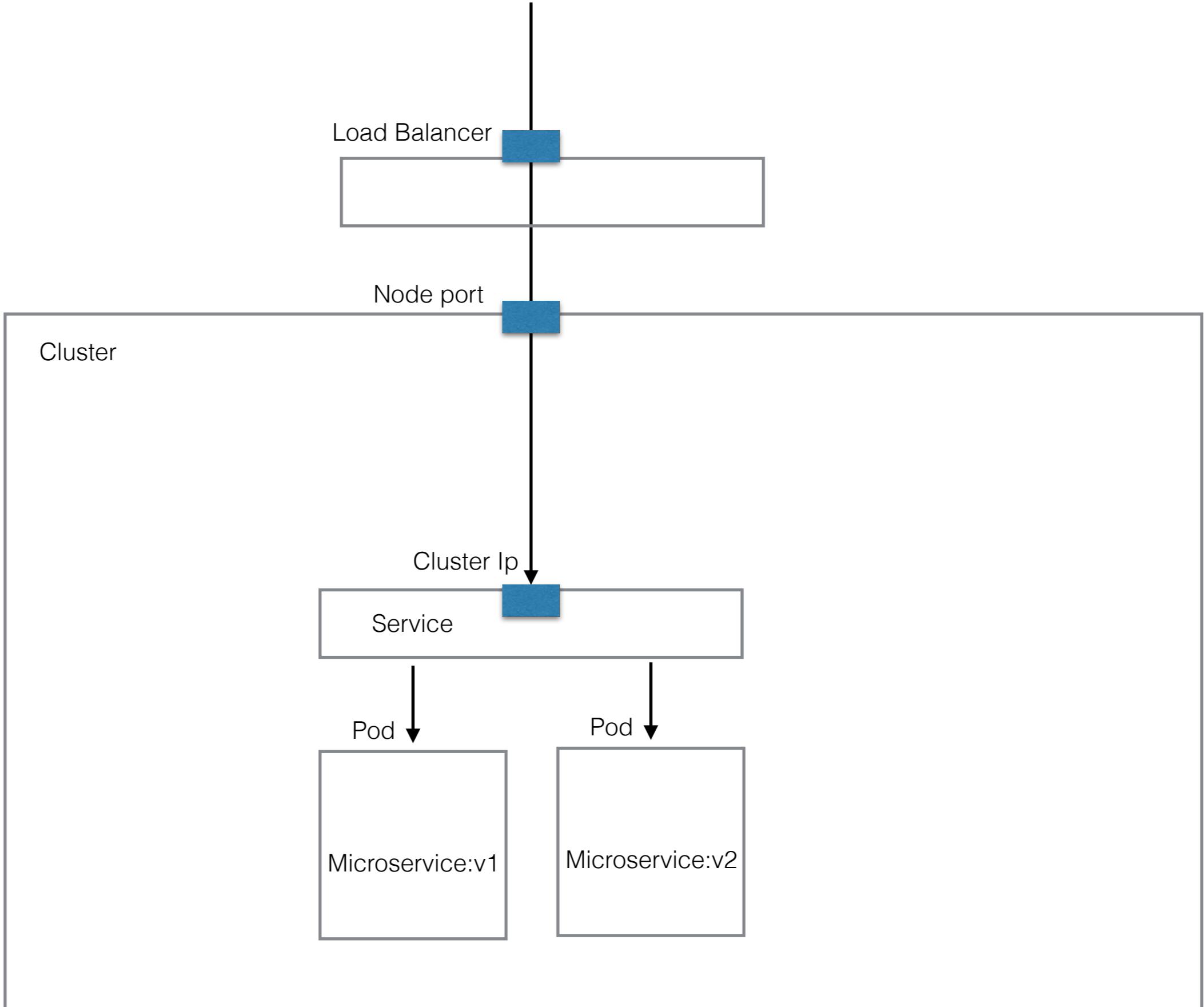


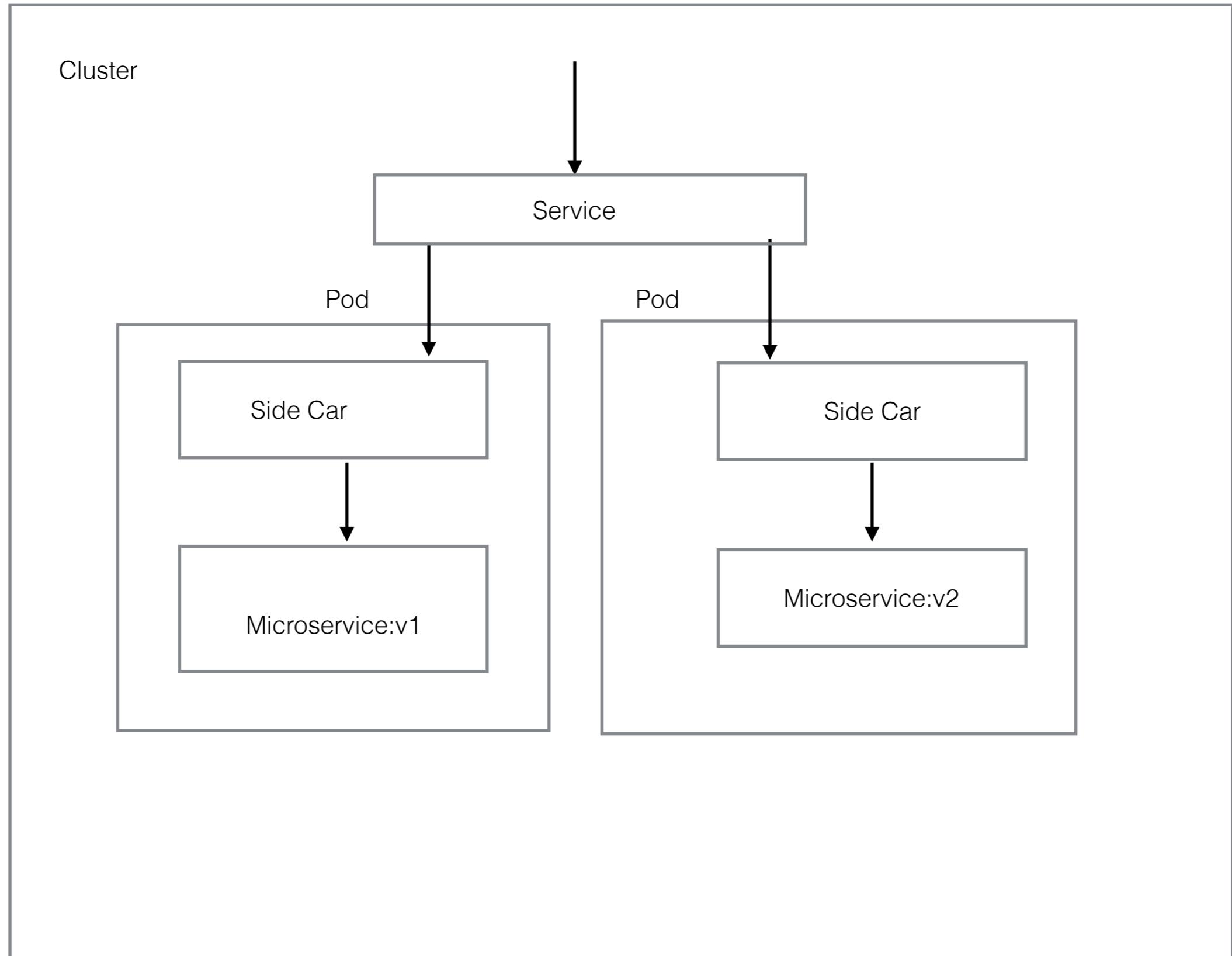
- Fail fast
- 

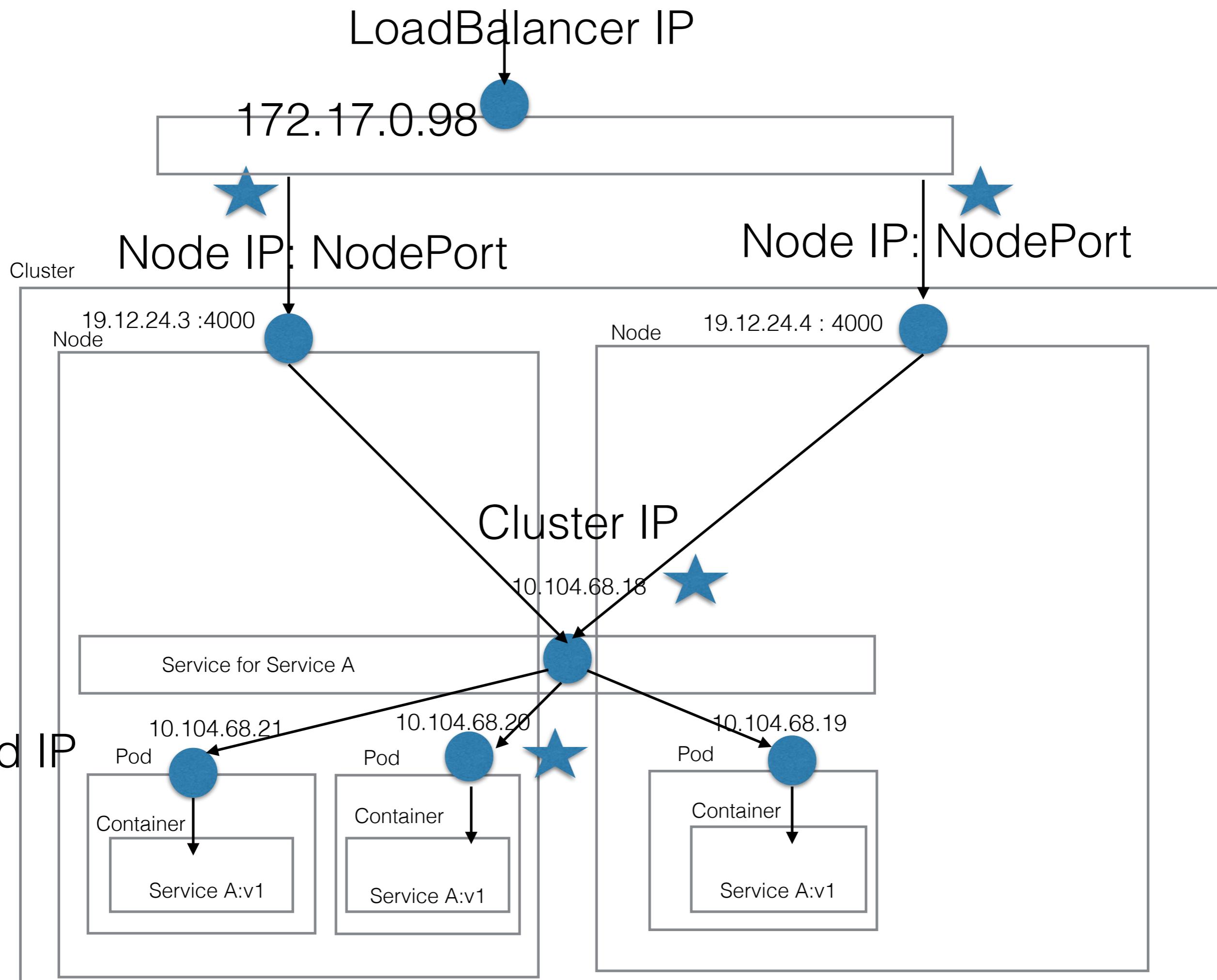


`curl http://10.97.245.99:8080/date`





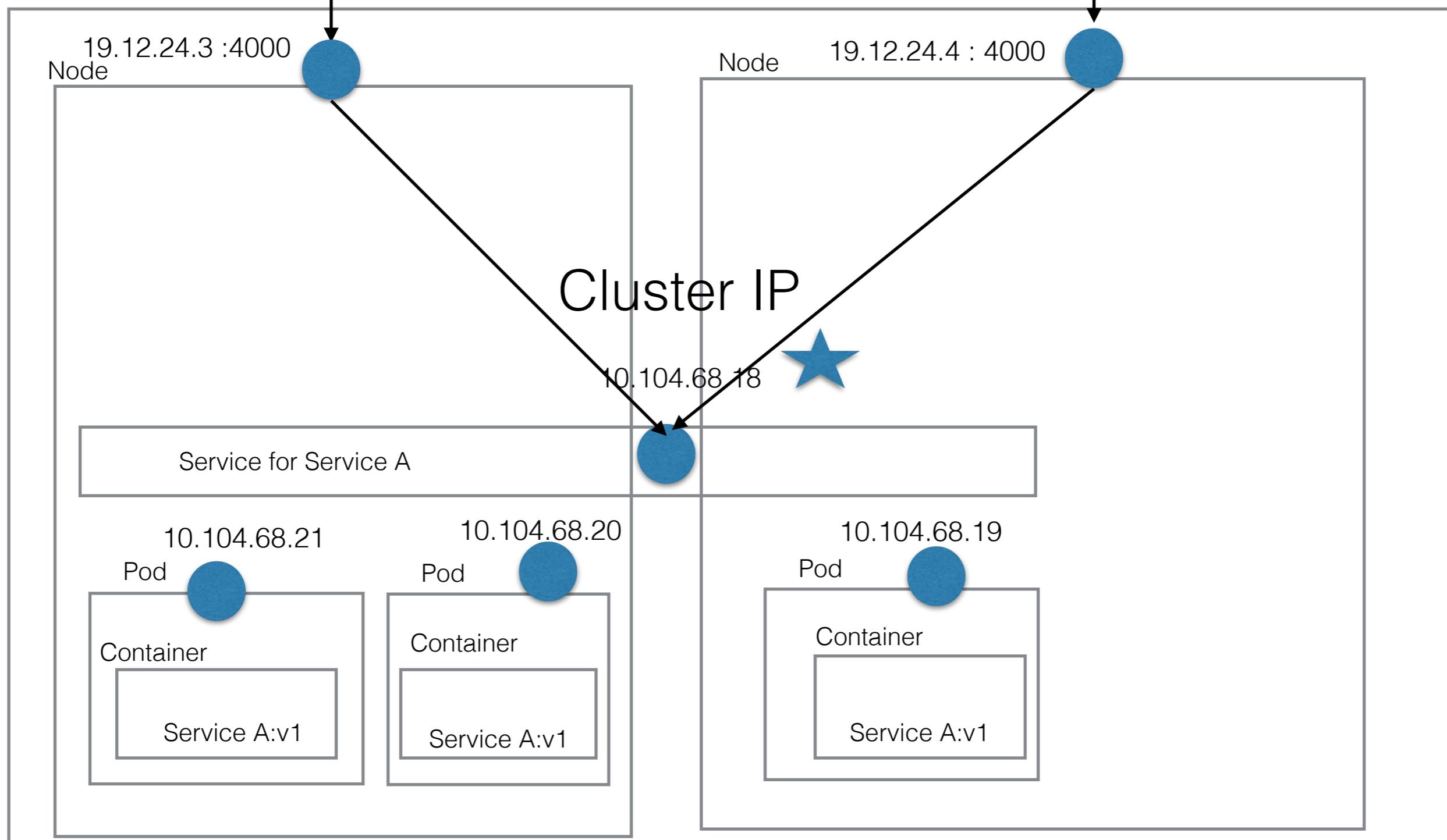




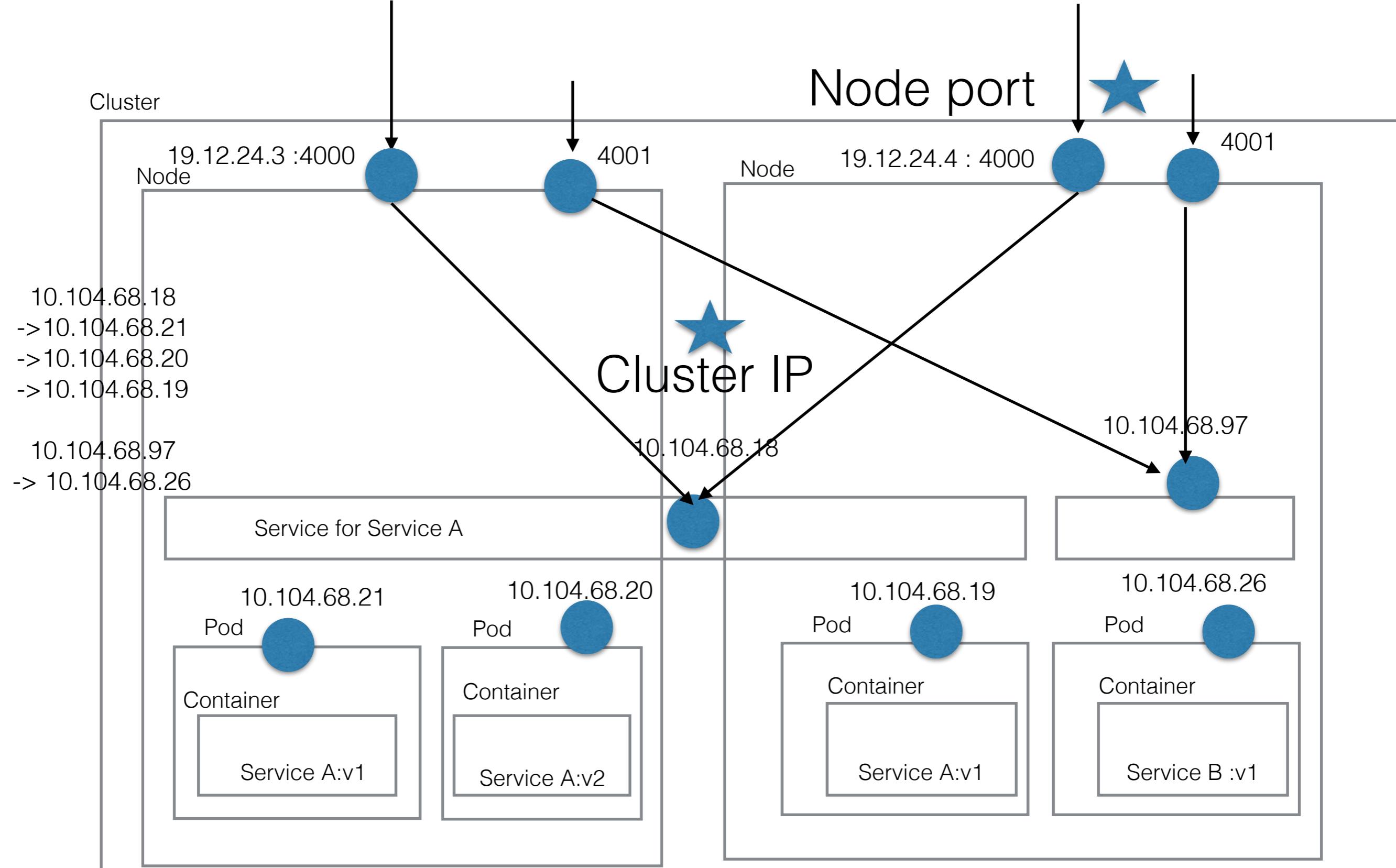
Node IP: NodePort

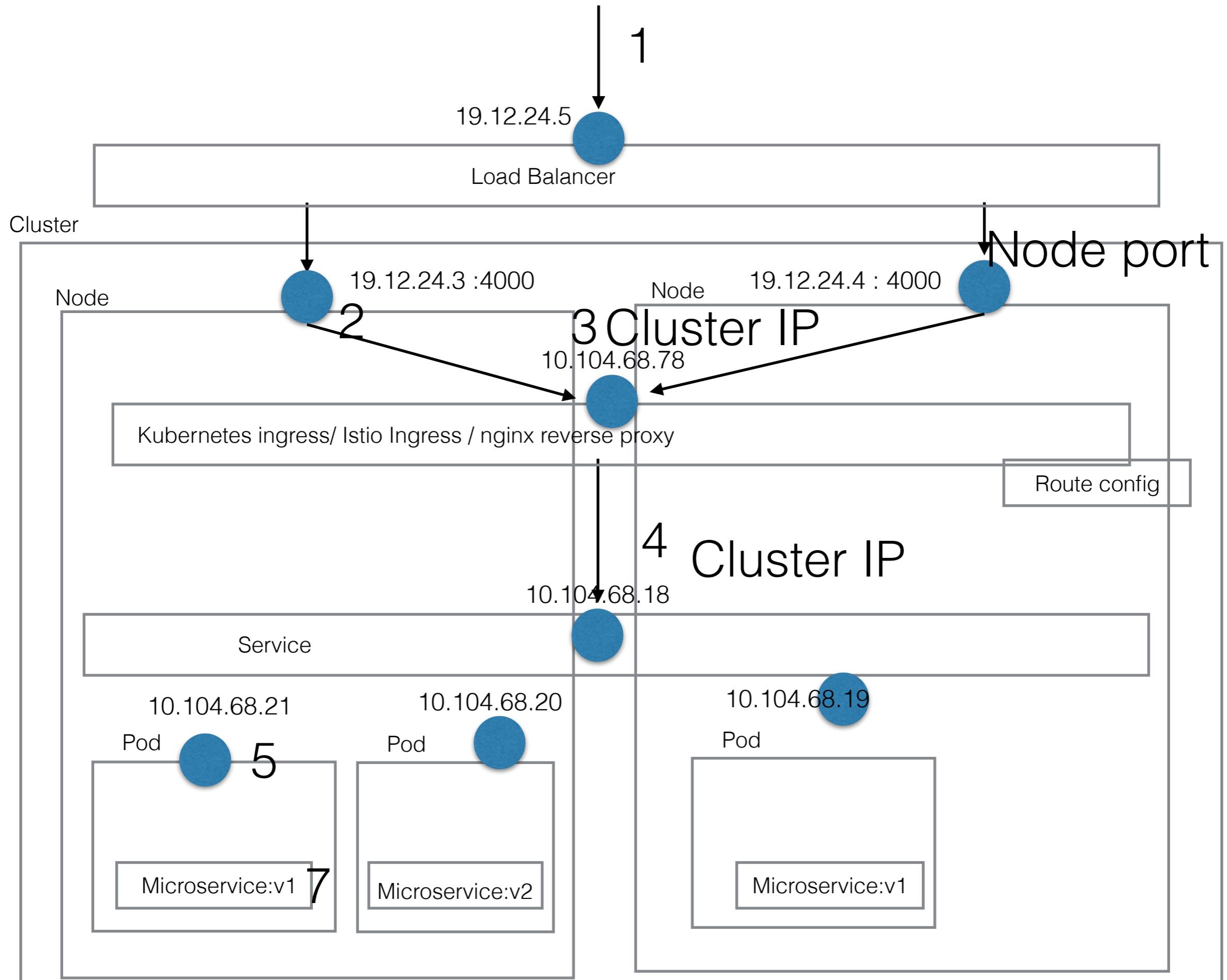
Node IP: NodePort

Cluster

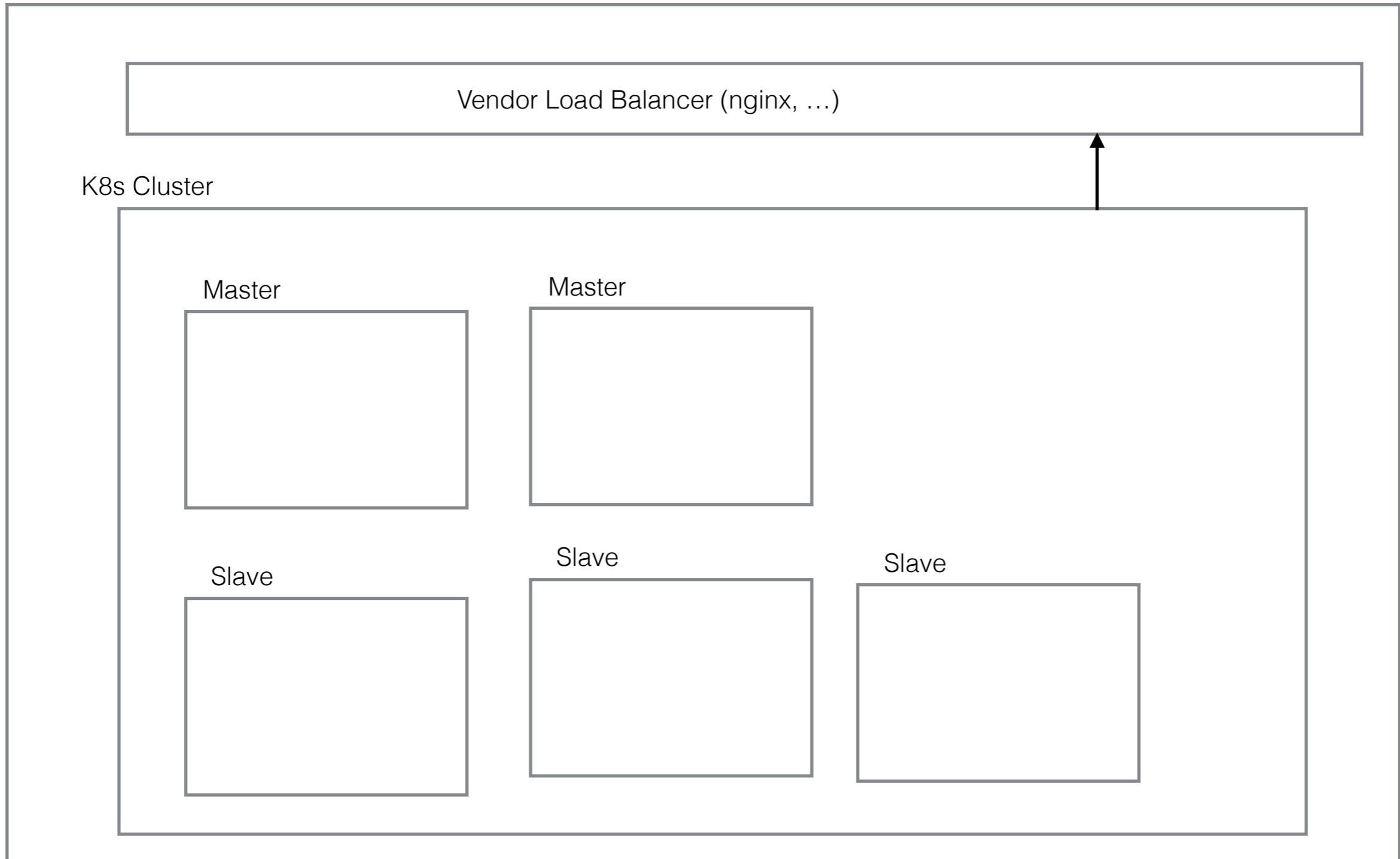


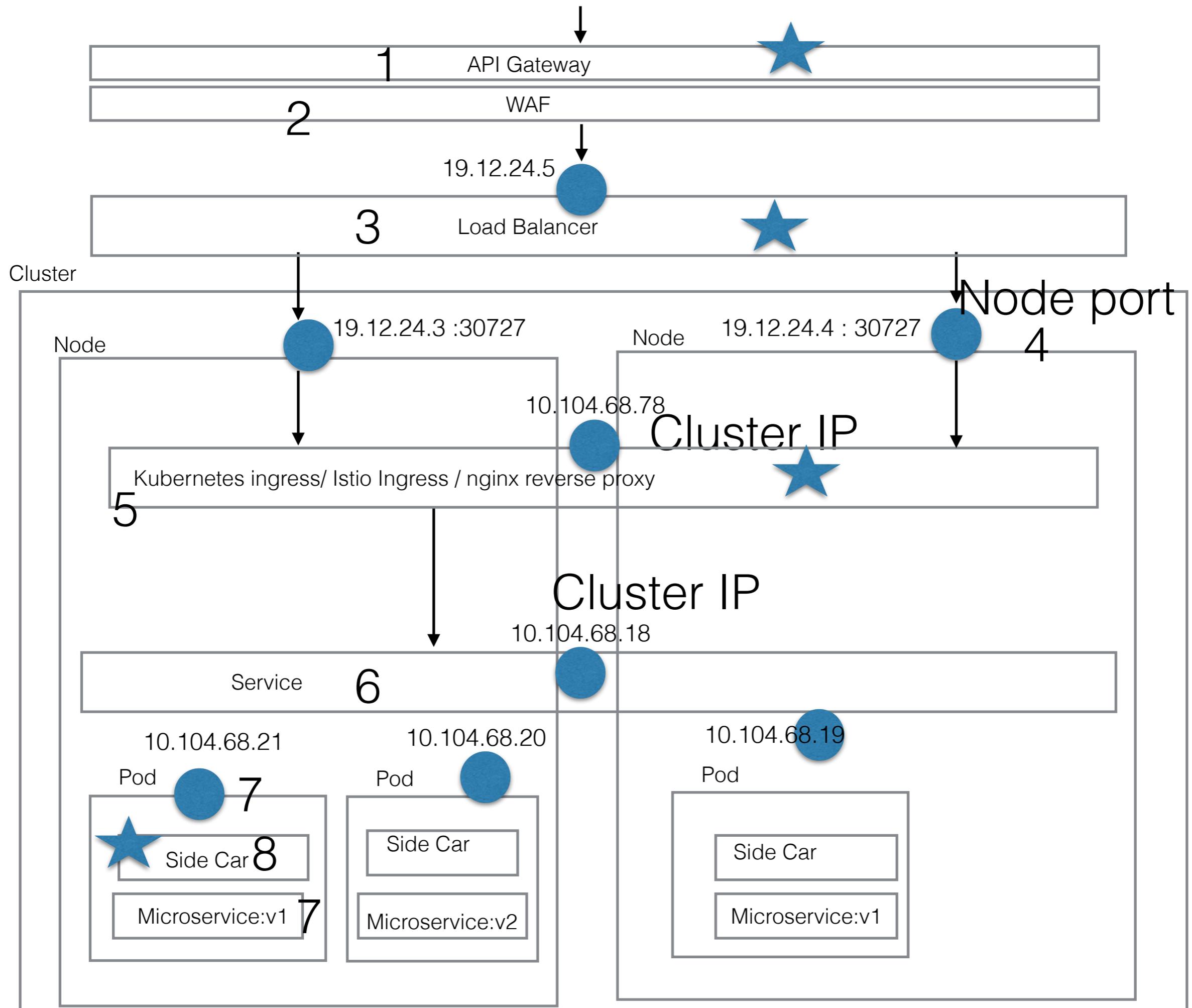
# Node IP: NodePort

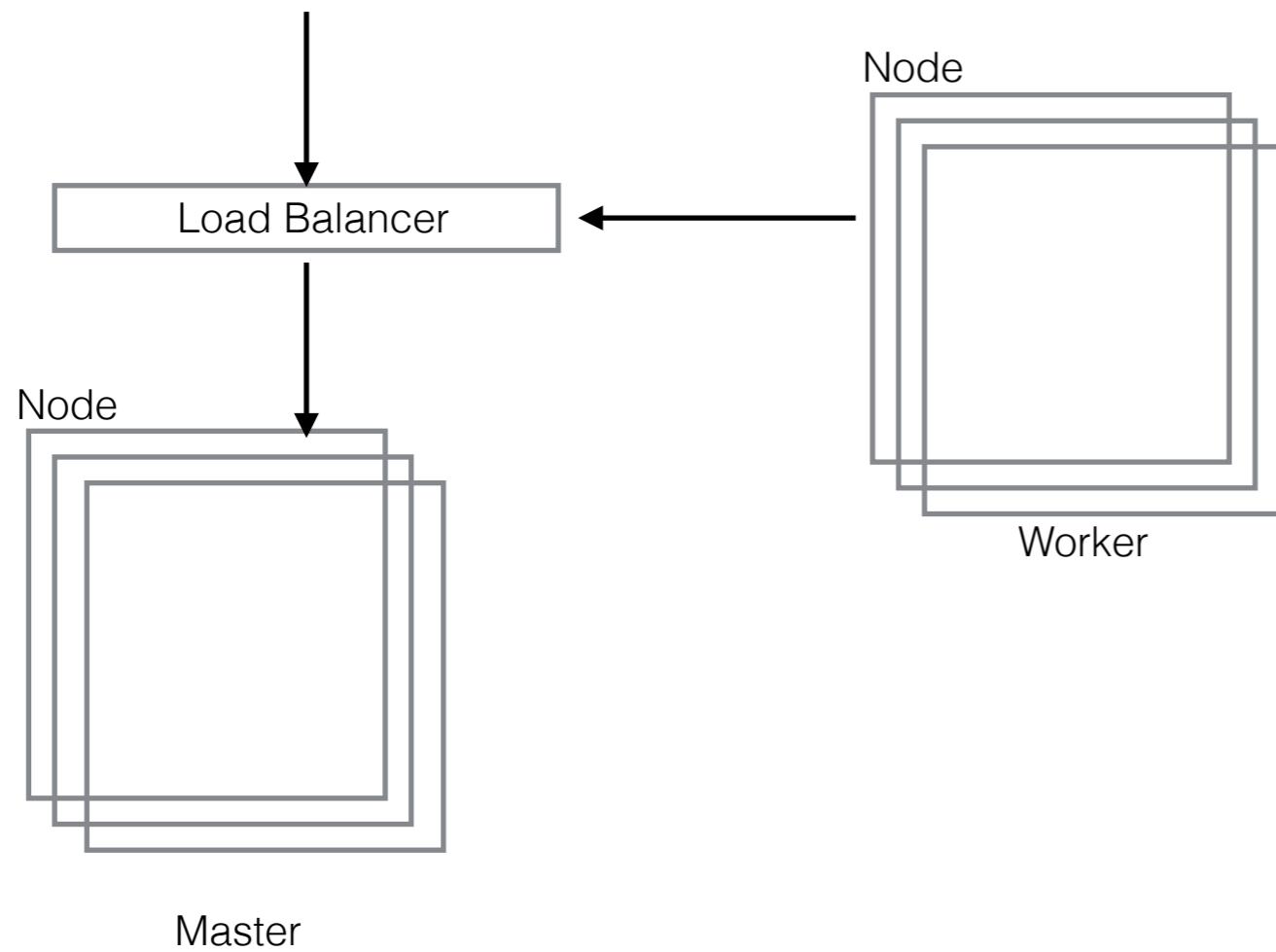




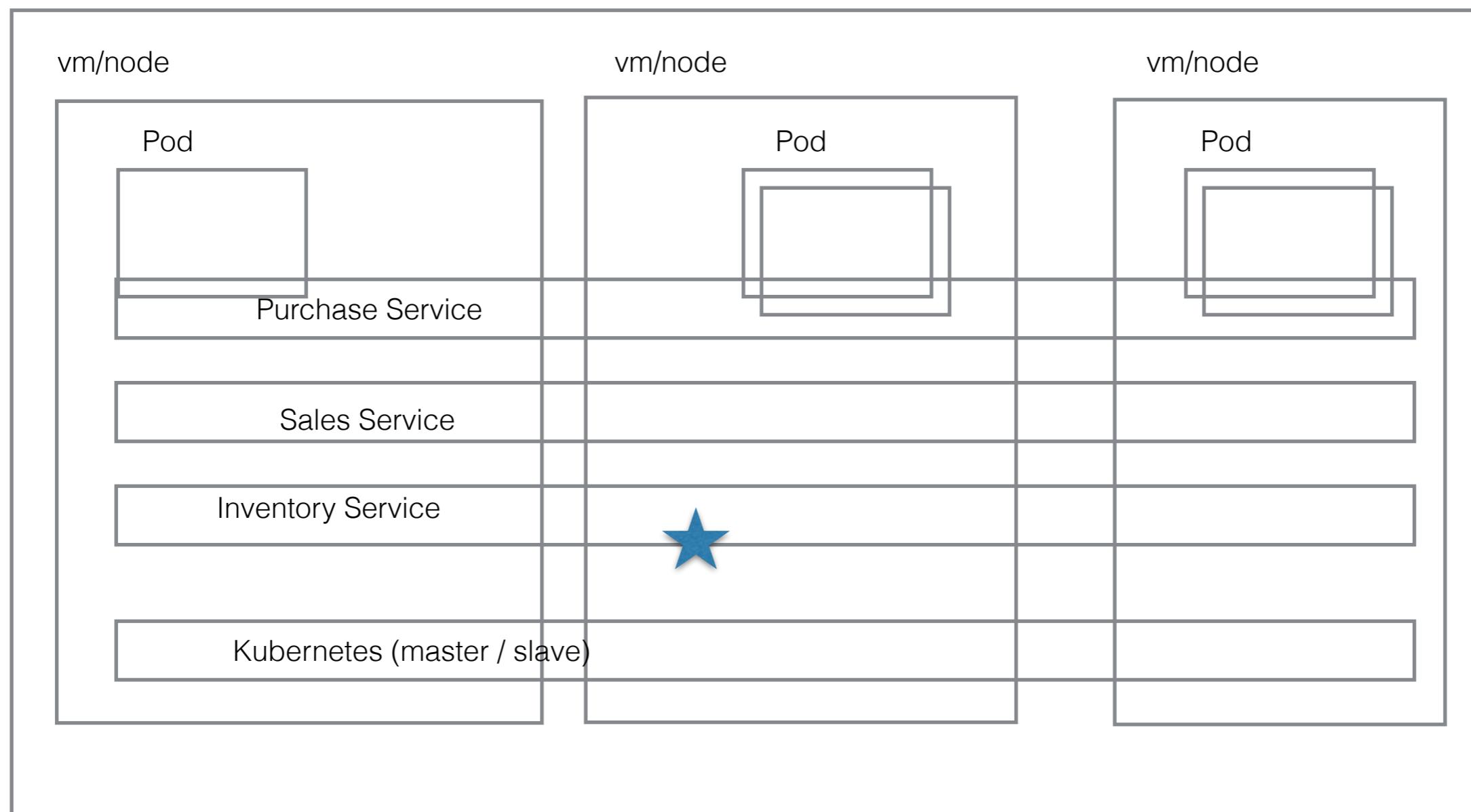
HTSL

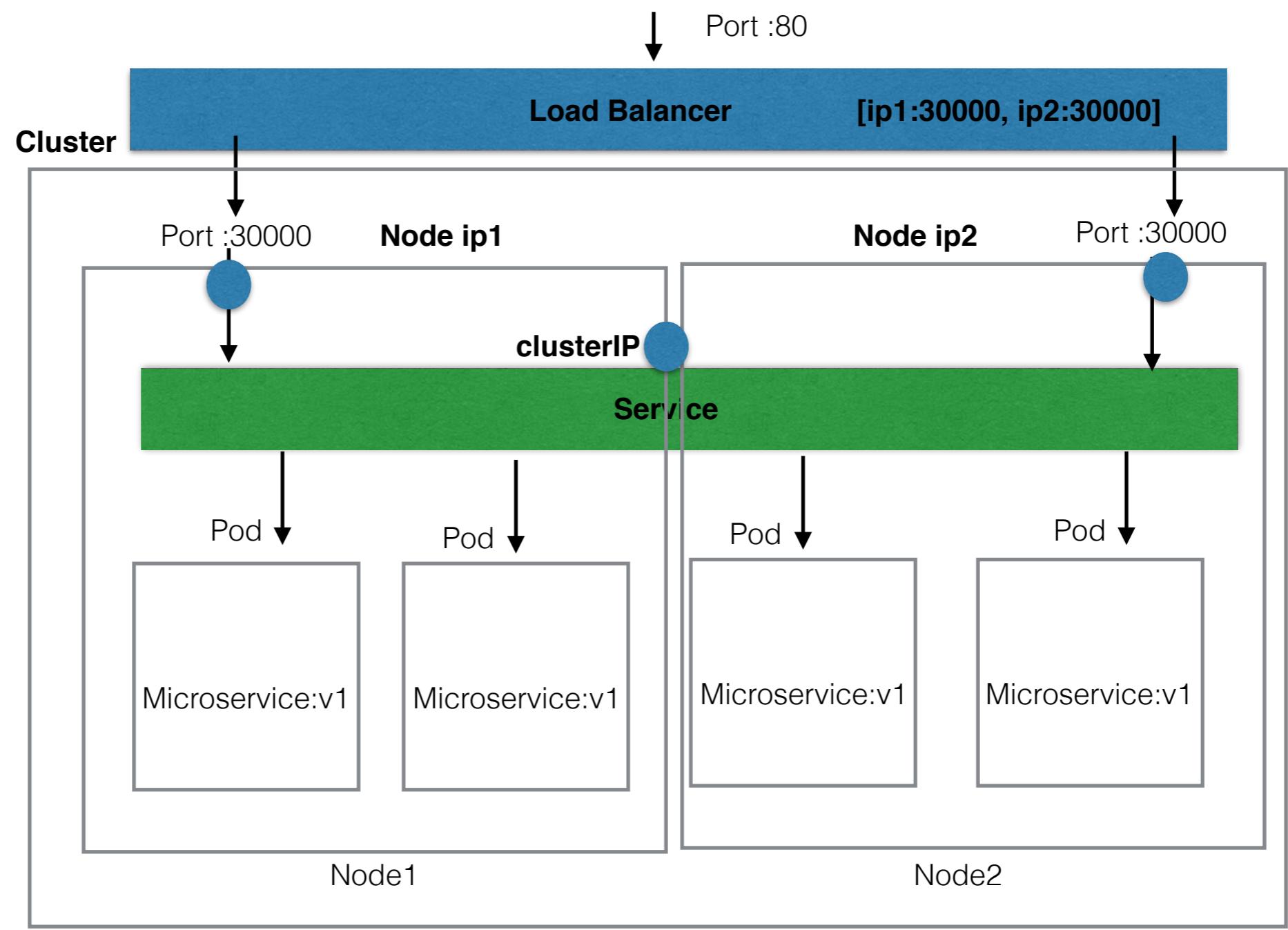


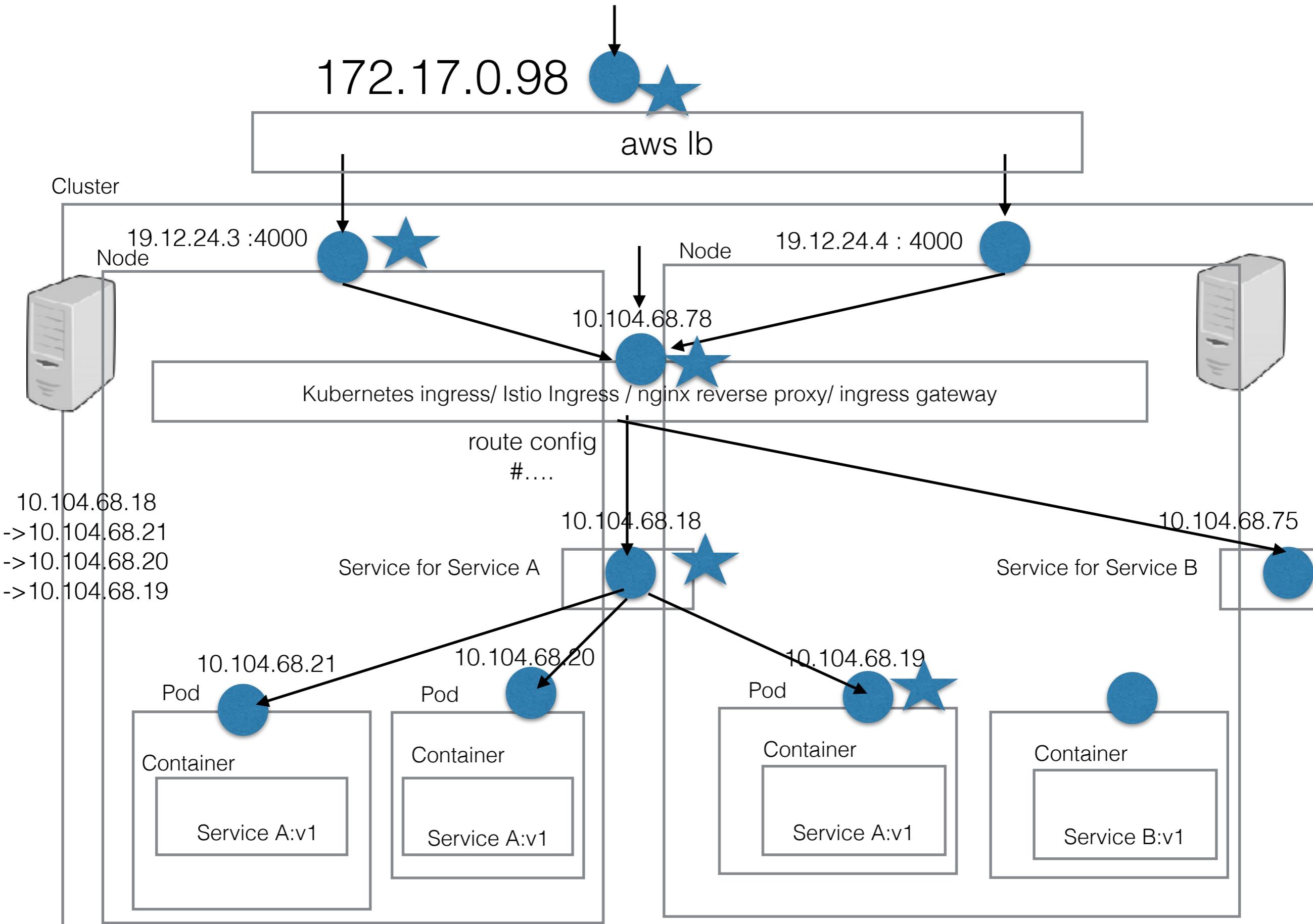


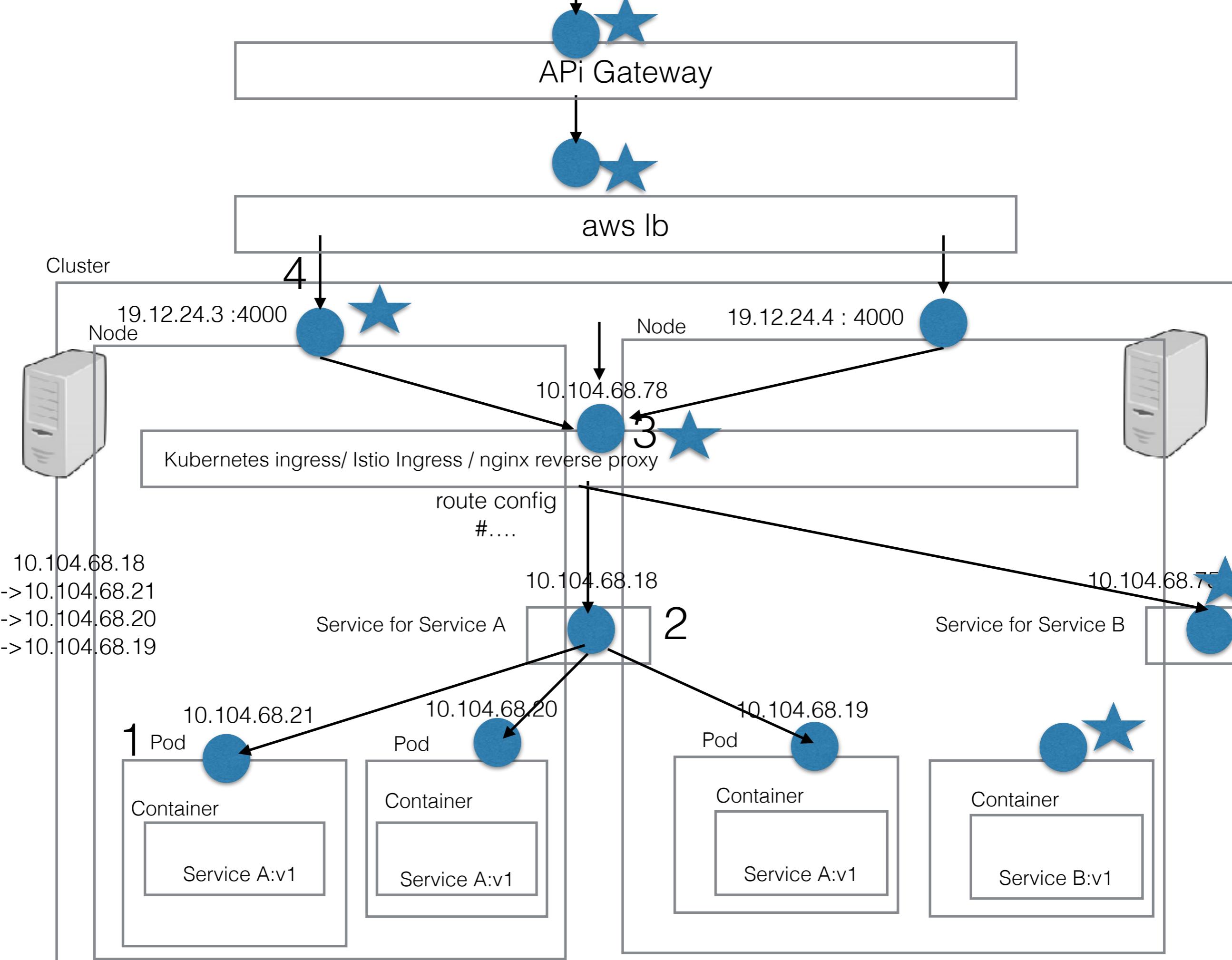


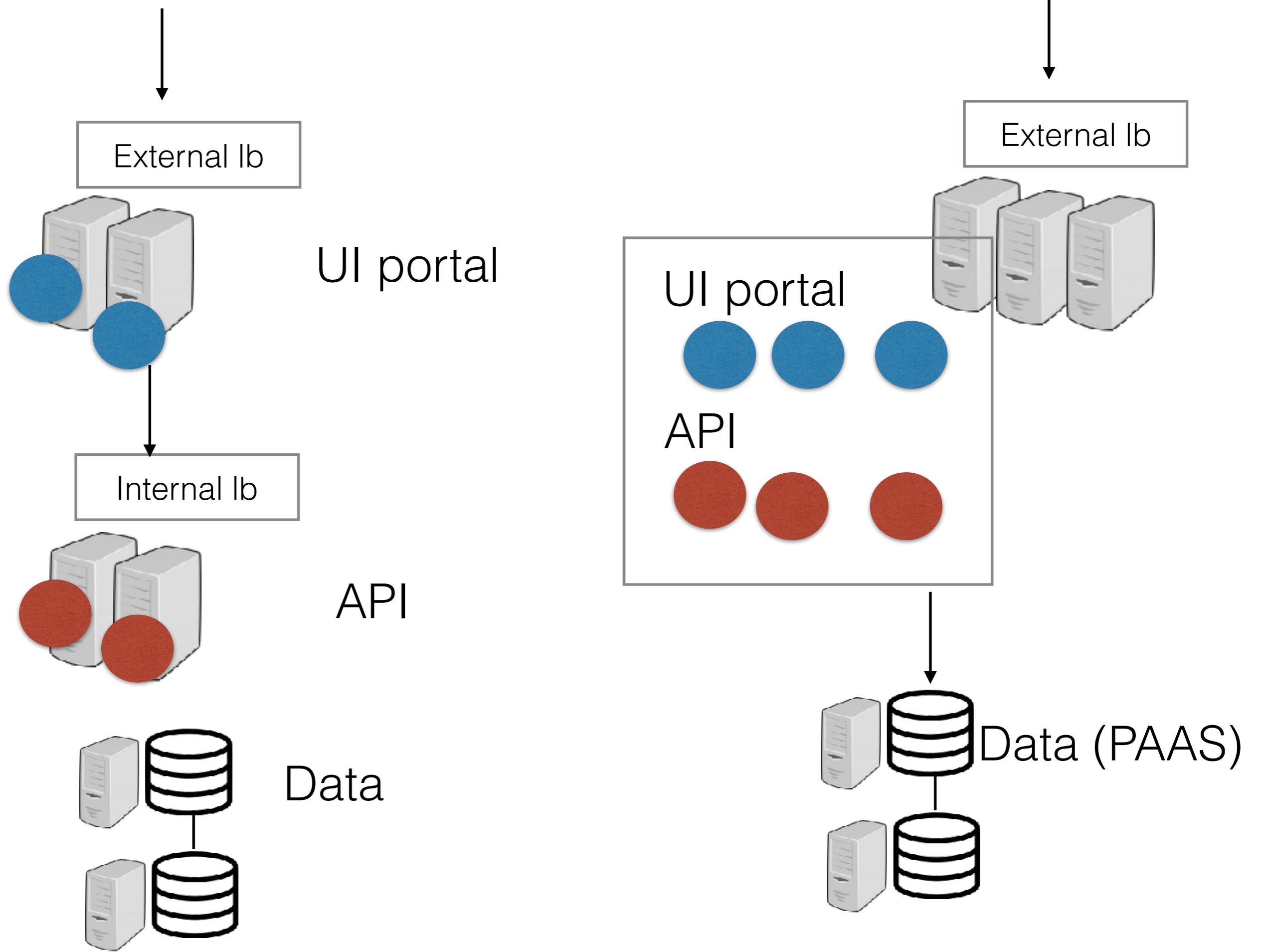
Cluster

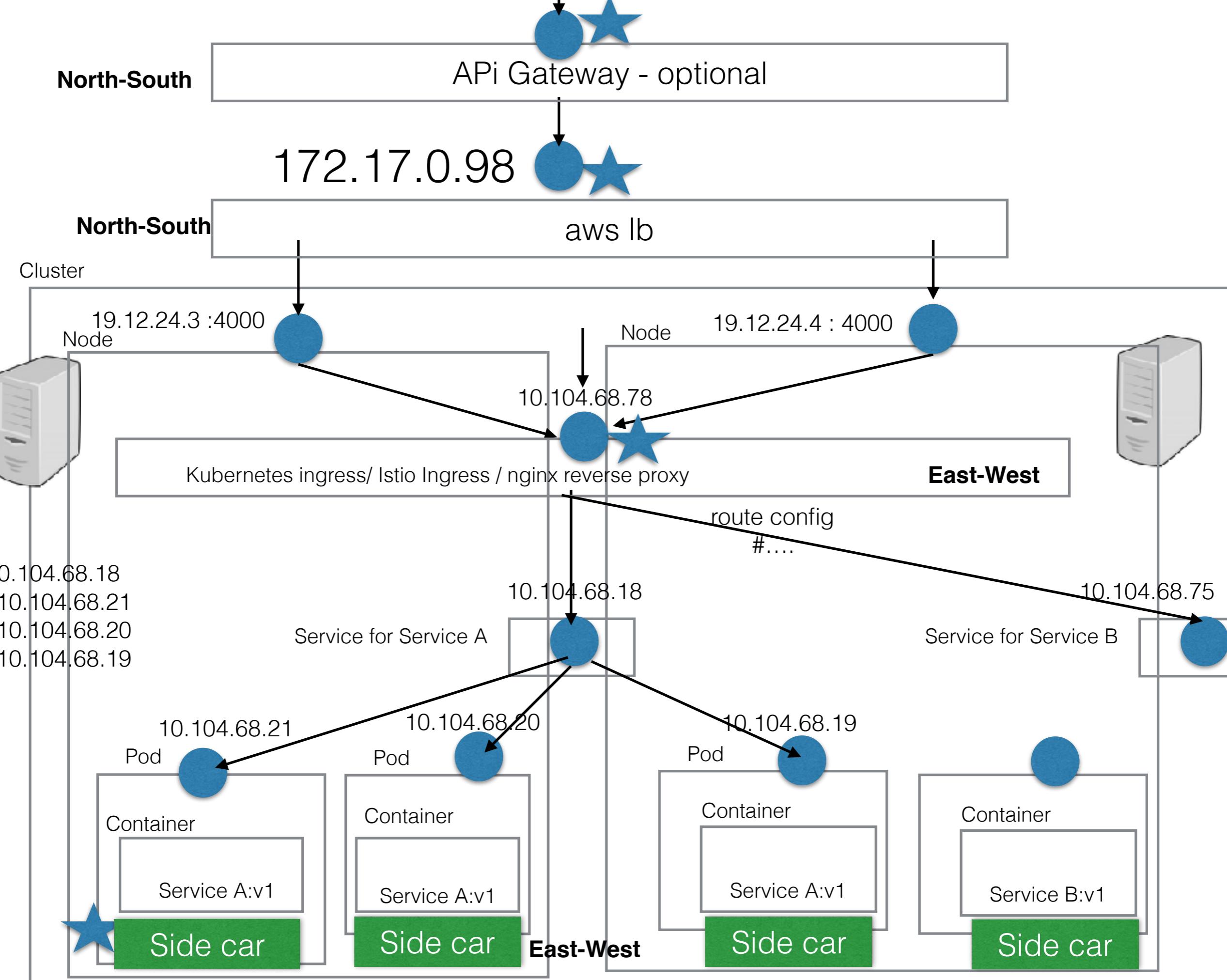




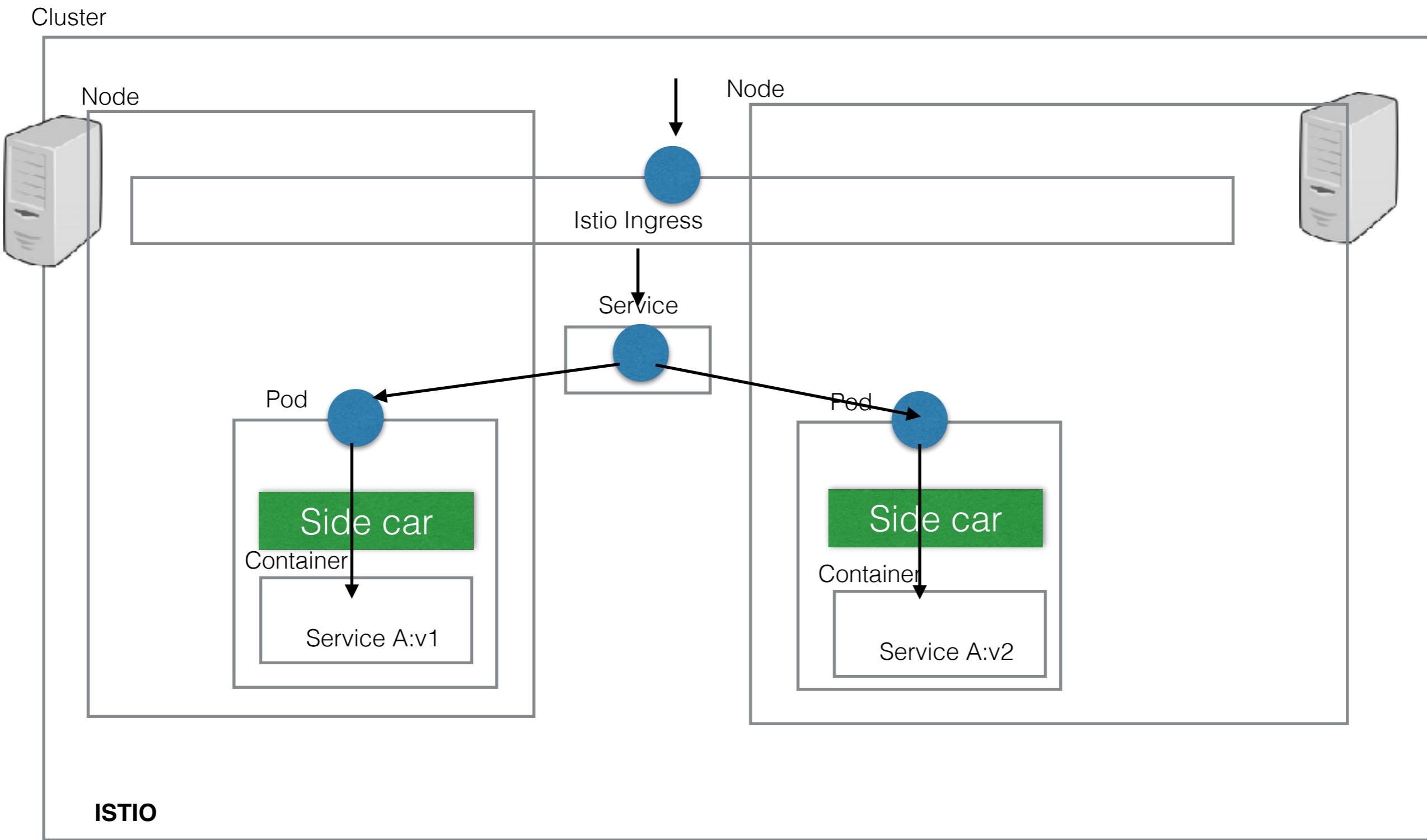


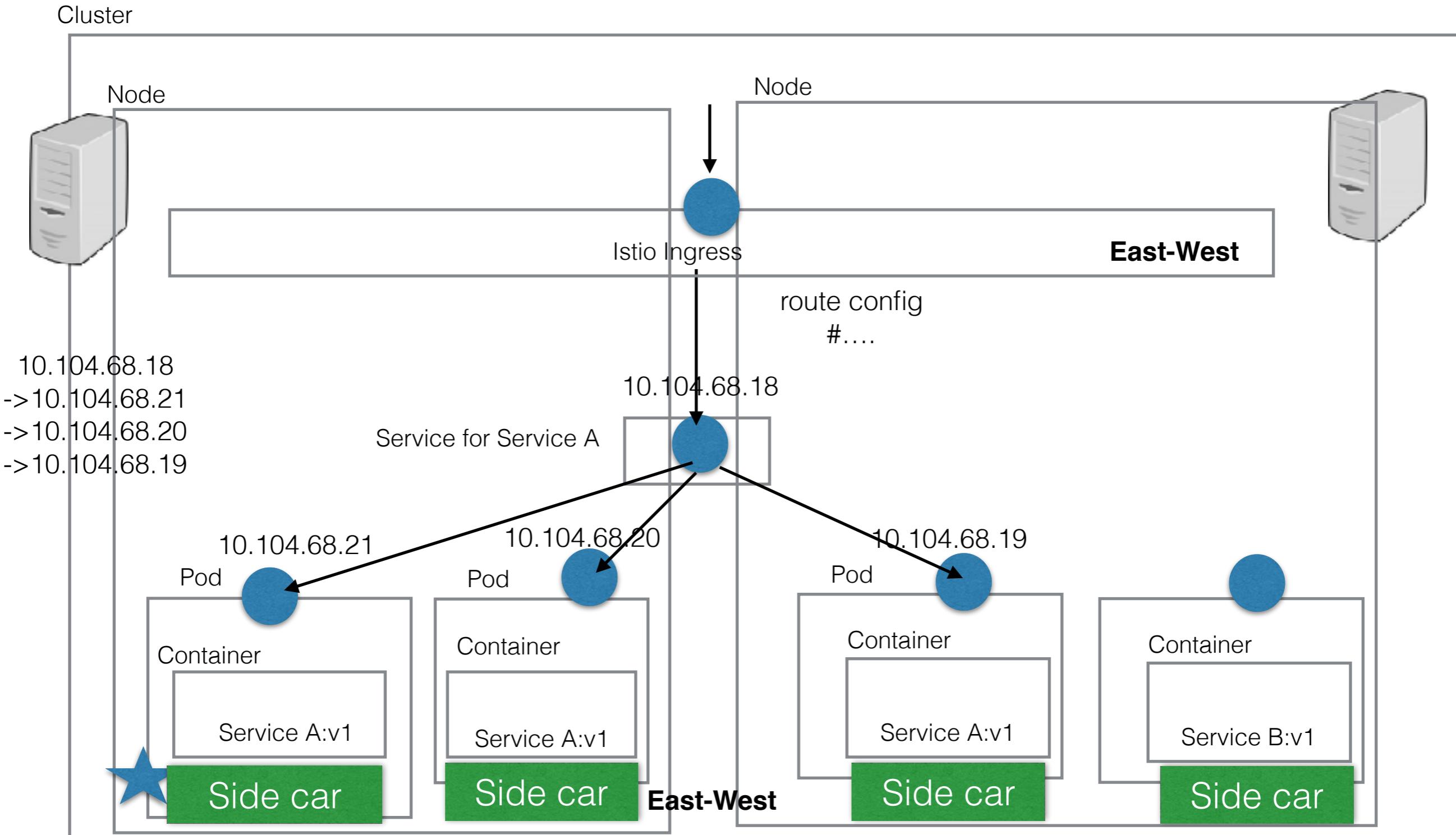


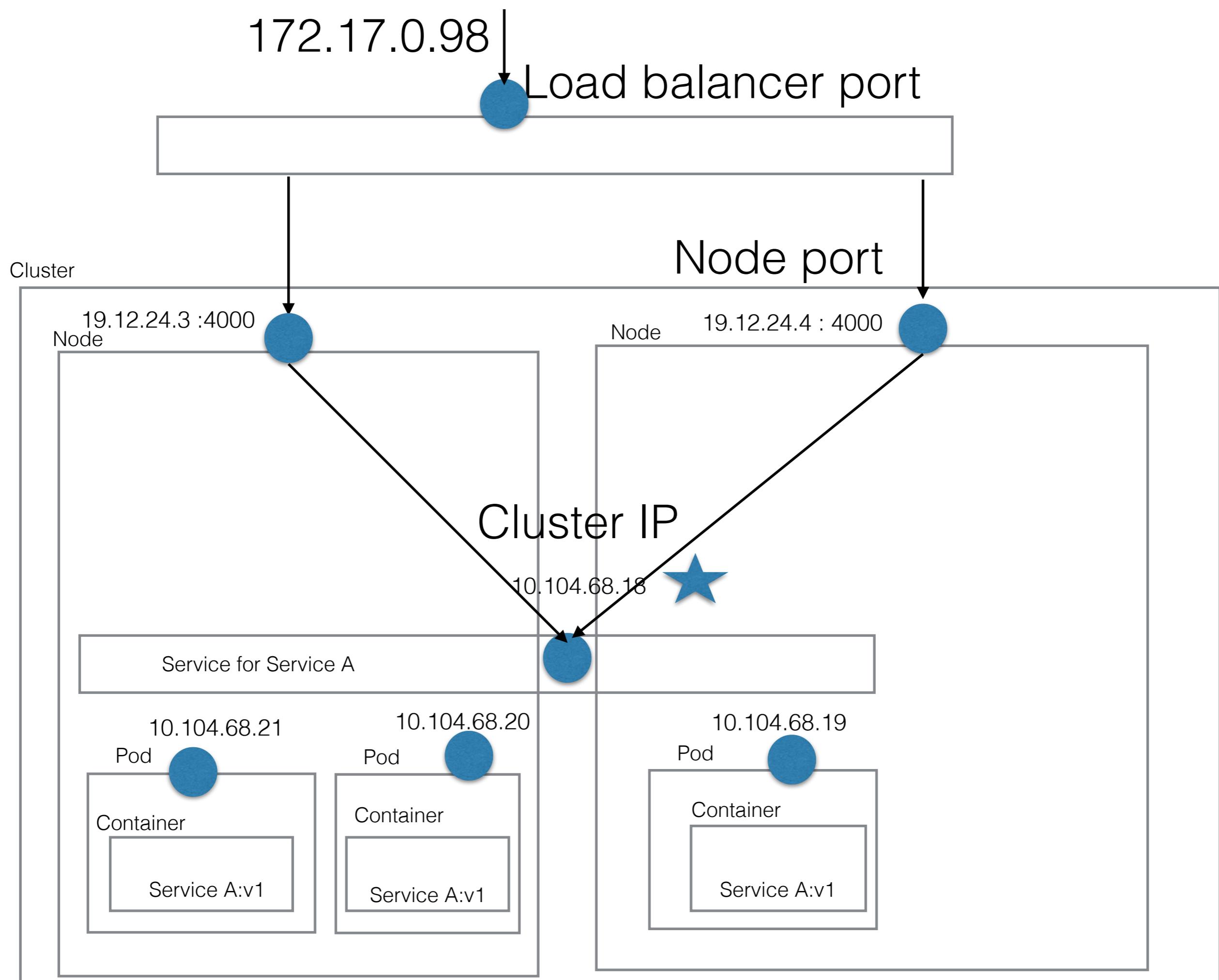


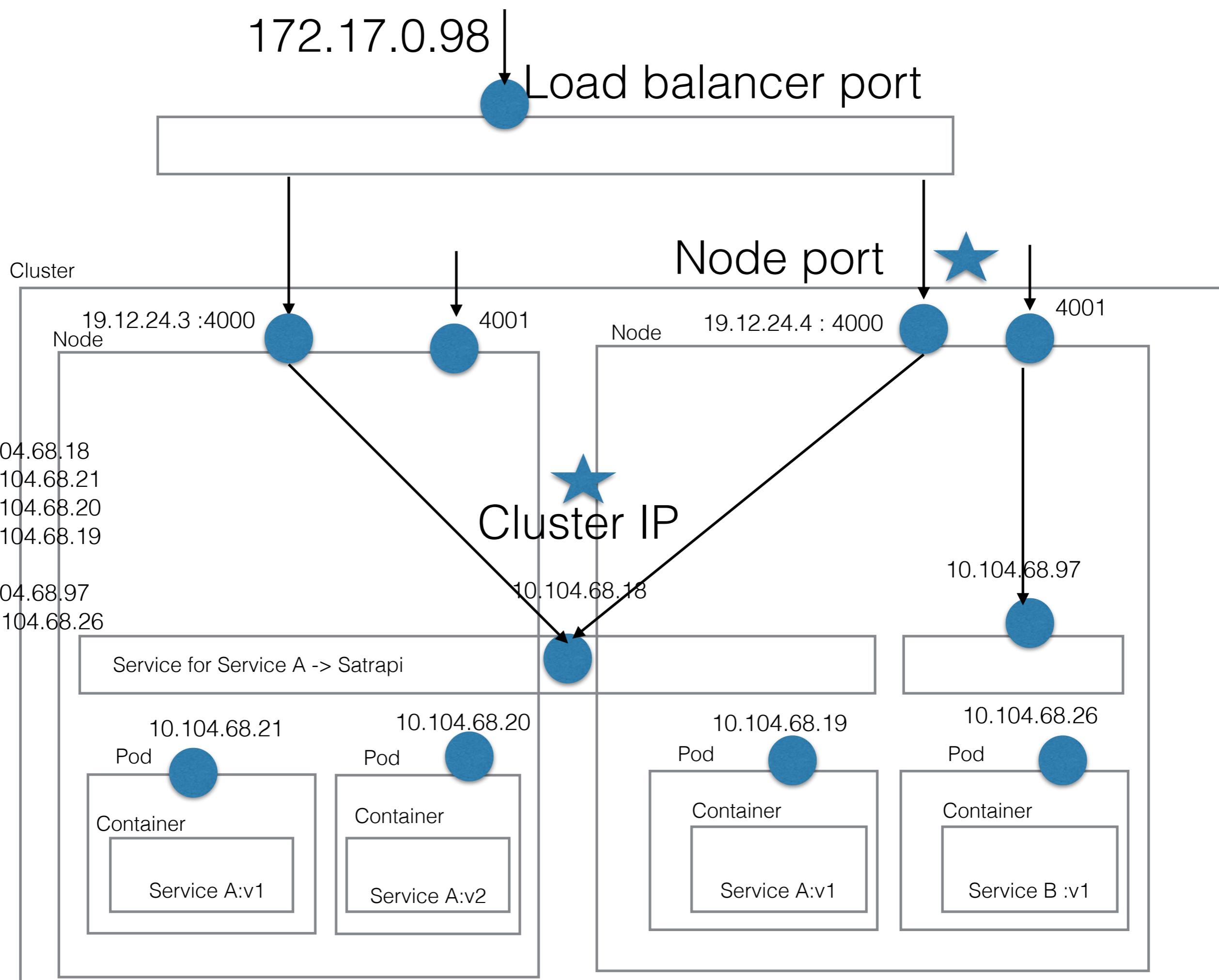


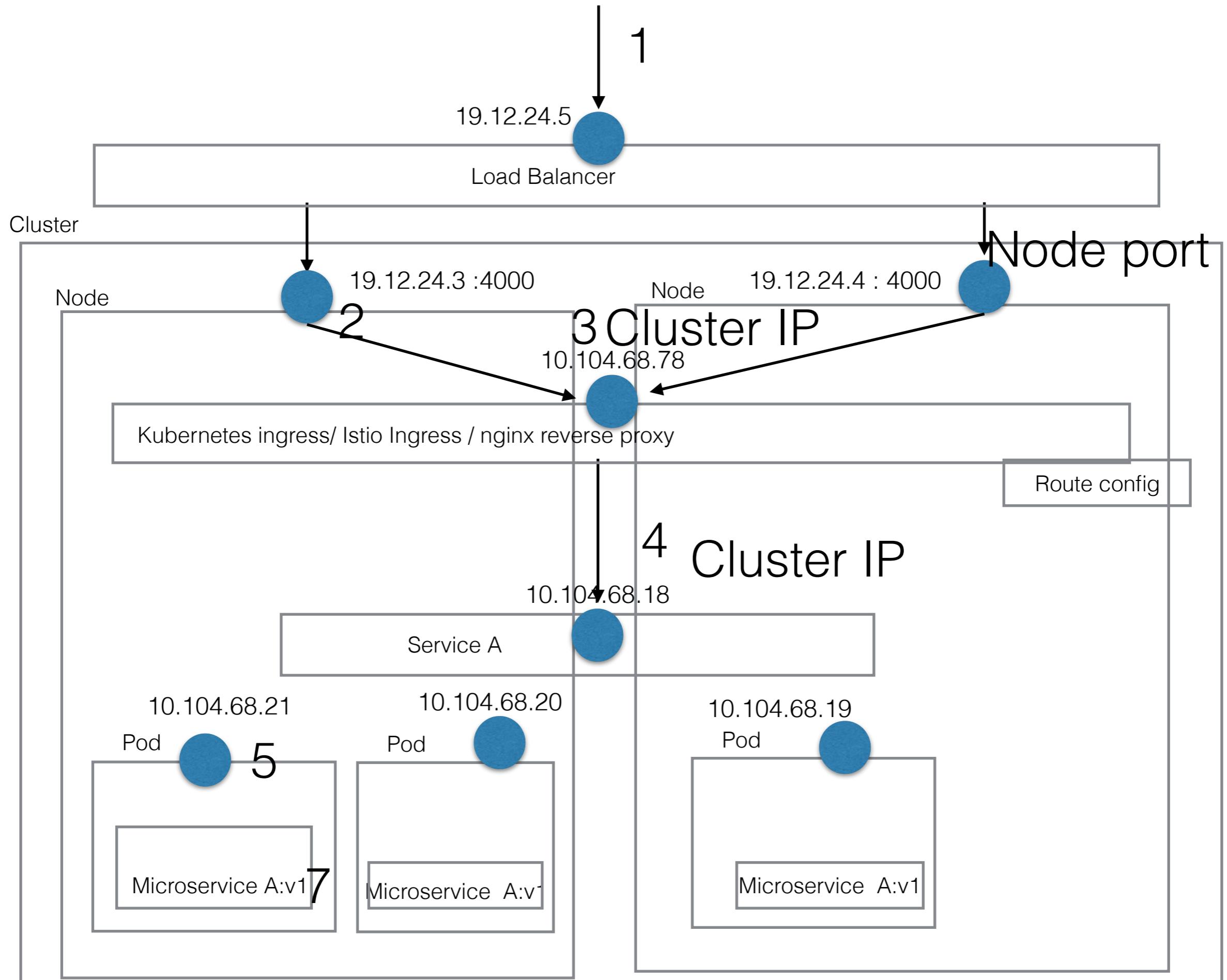
<https://www.katacoda.com/courses/kubernetes/first-steps-to-ckad-certification>



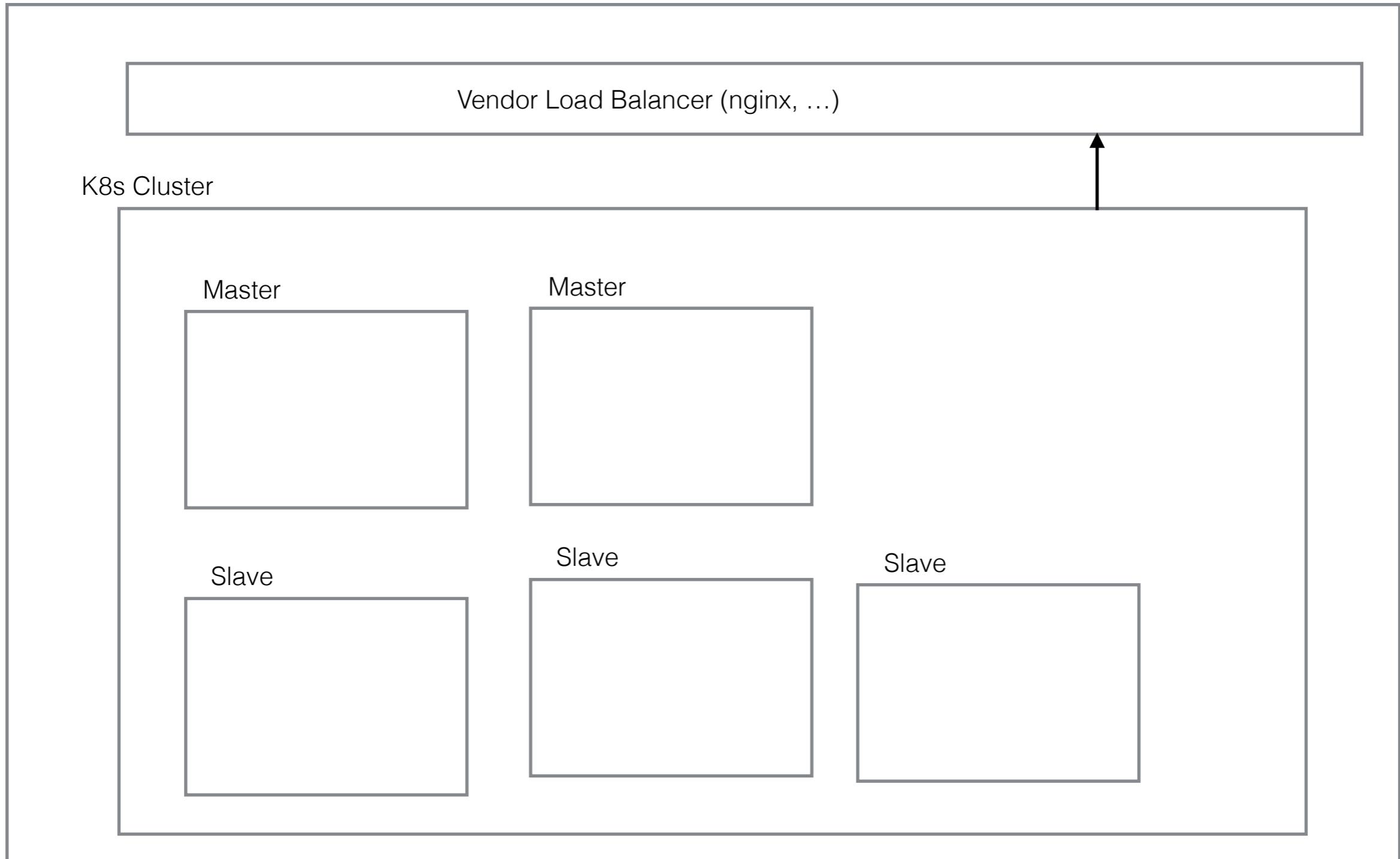


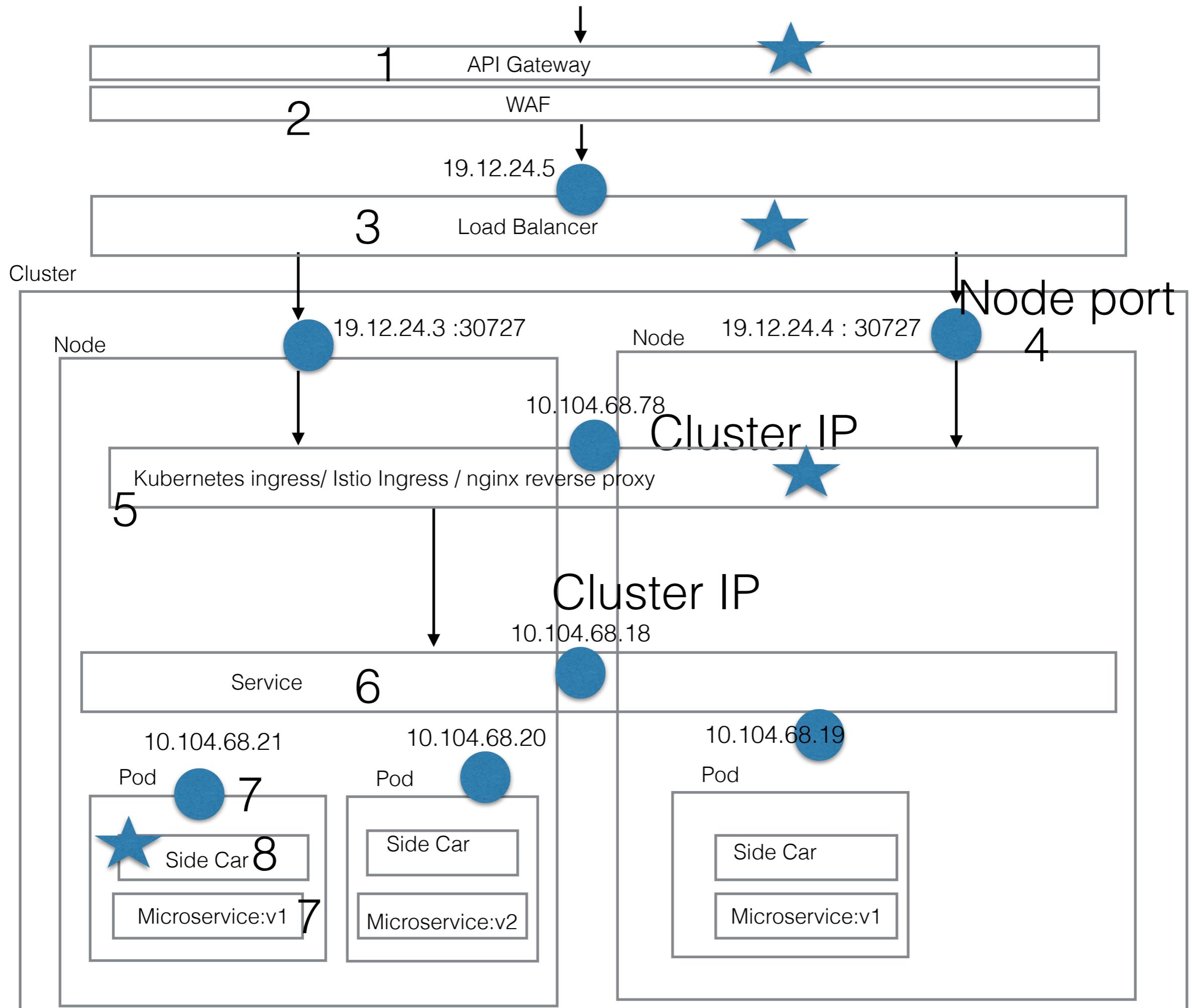


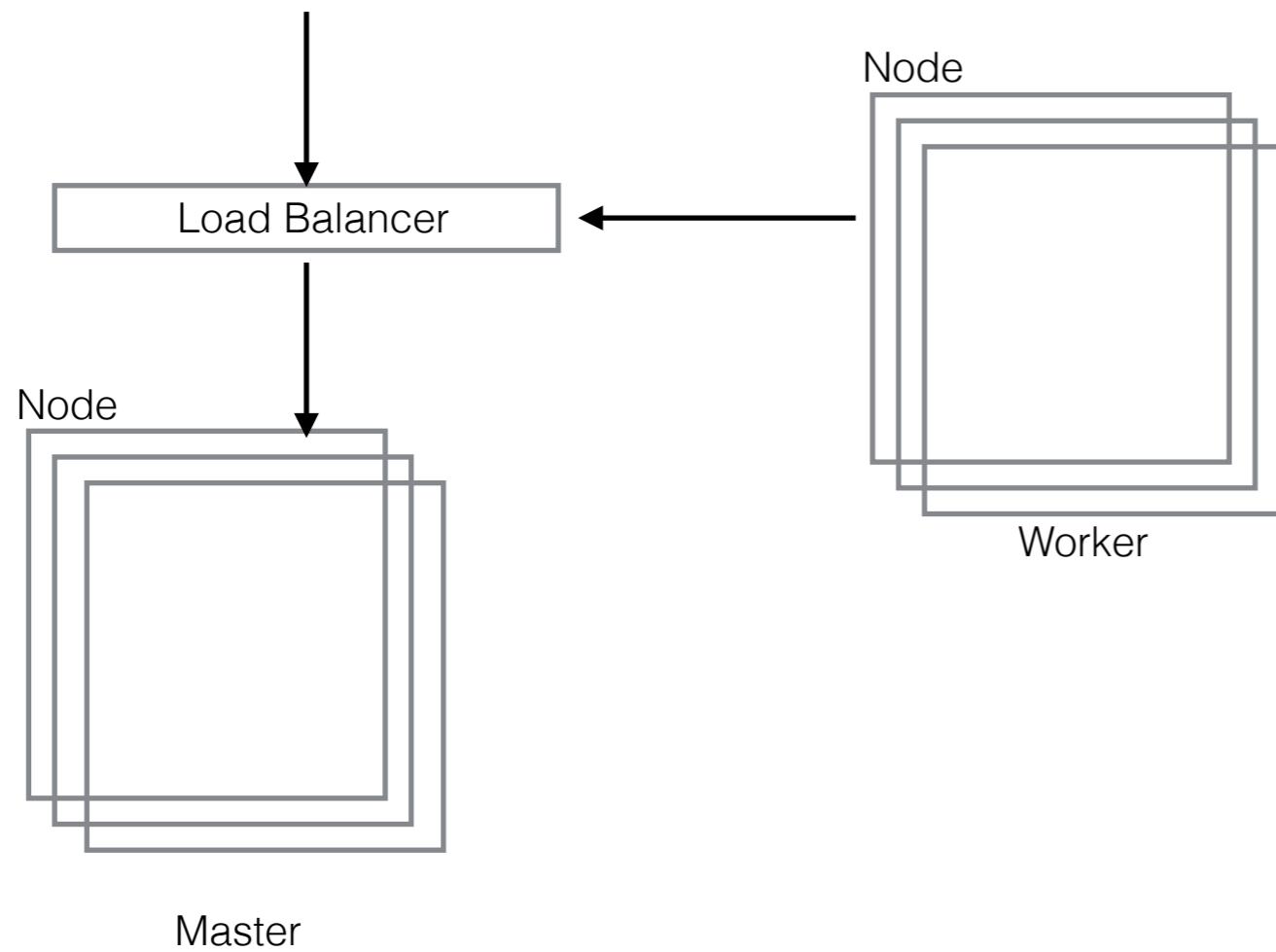




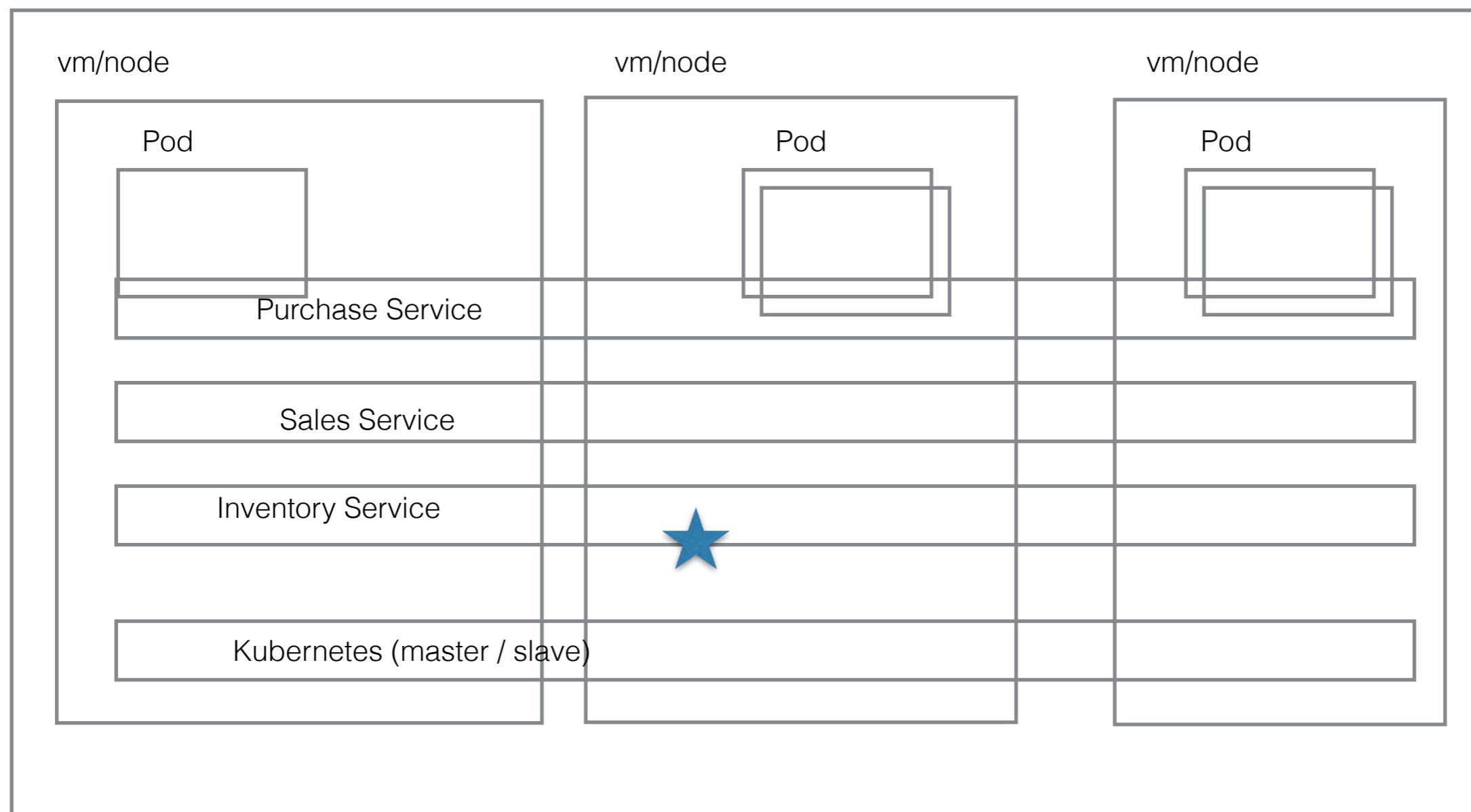
HTSL

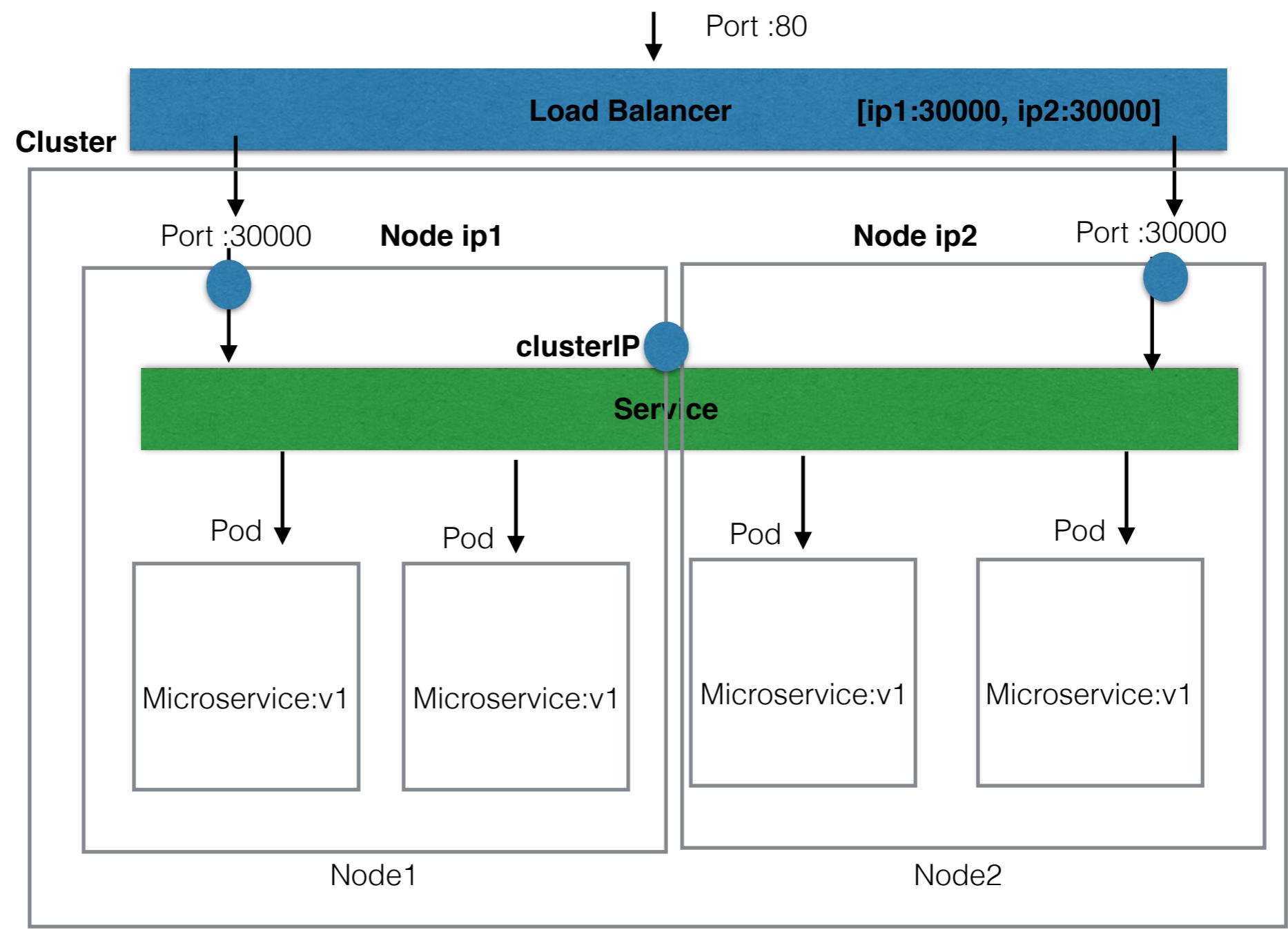


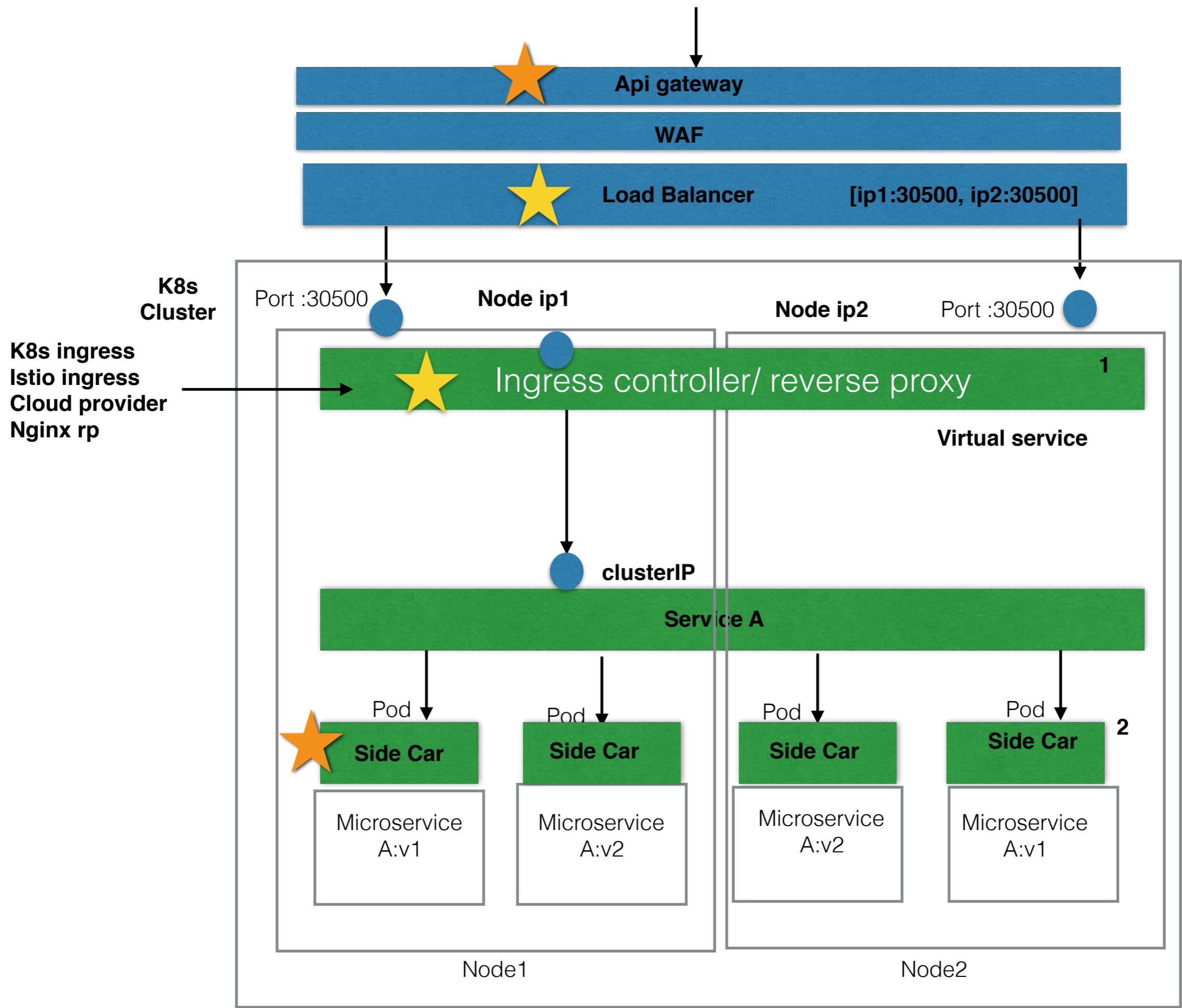




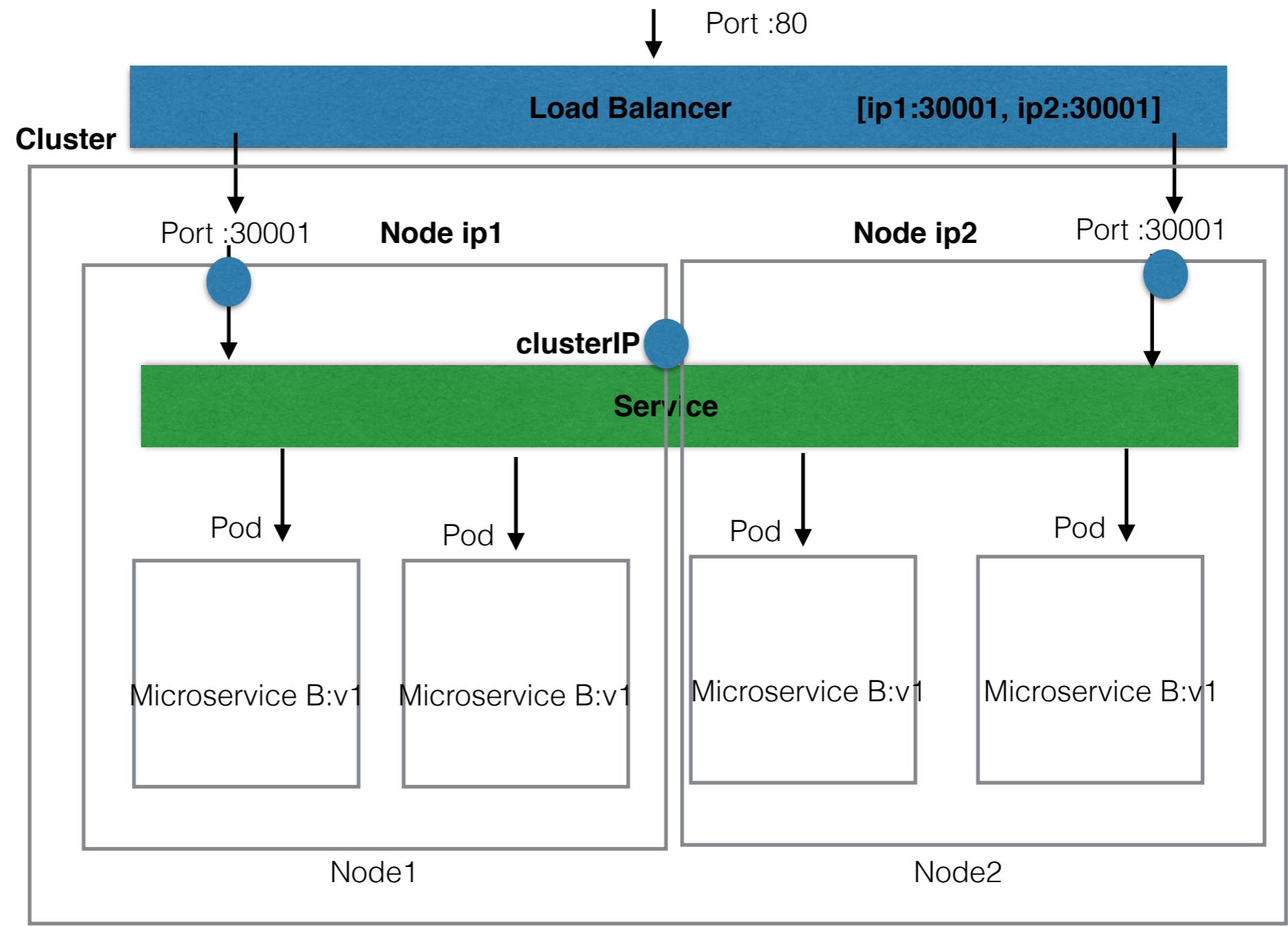
Cluster

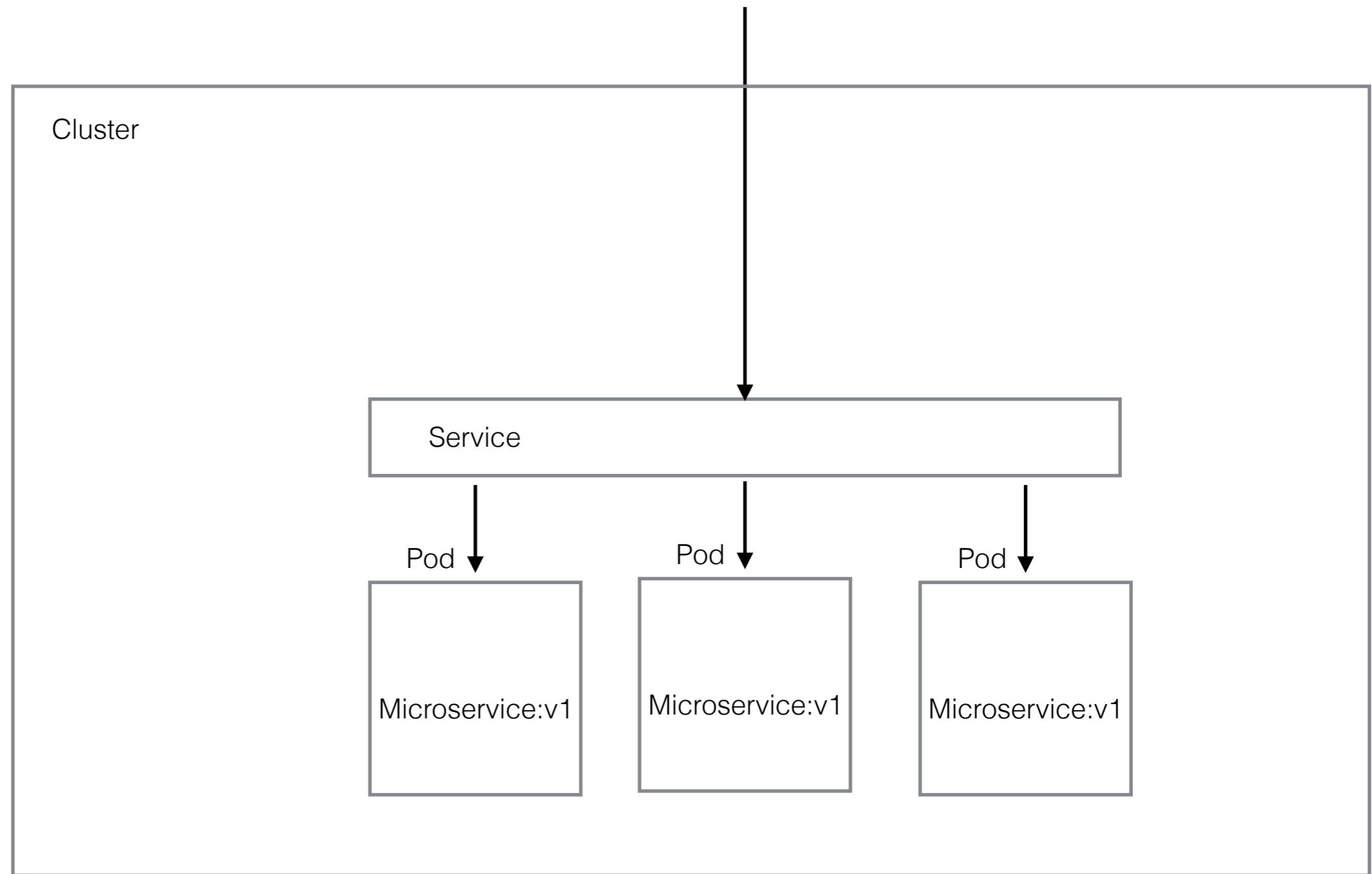


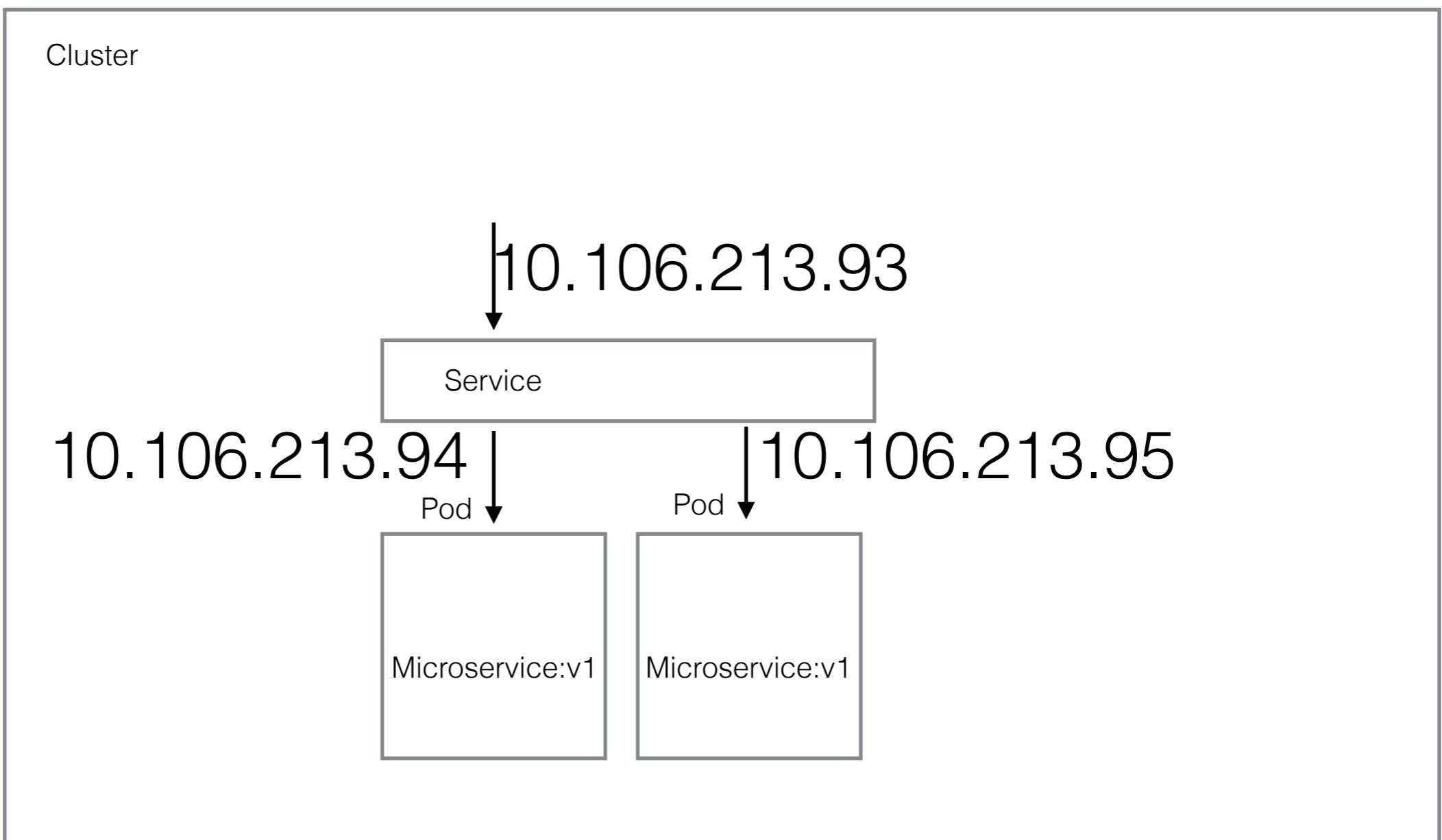


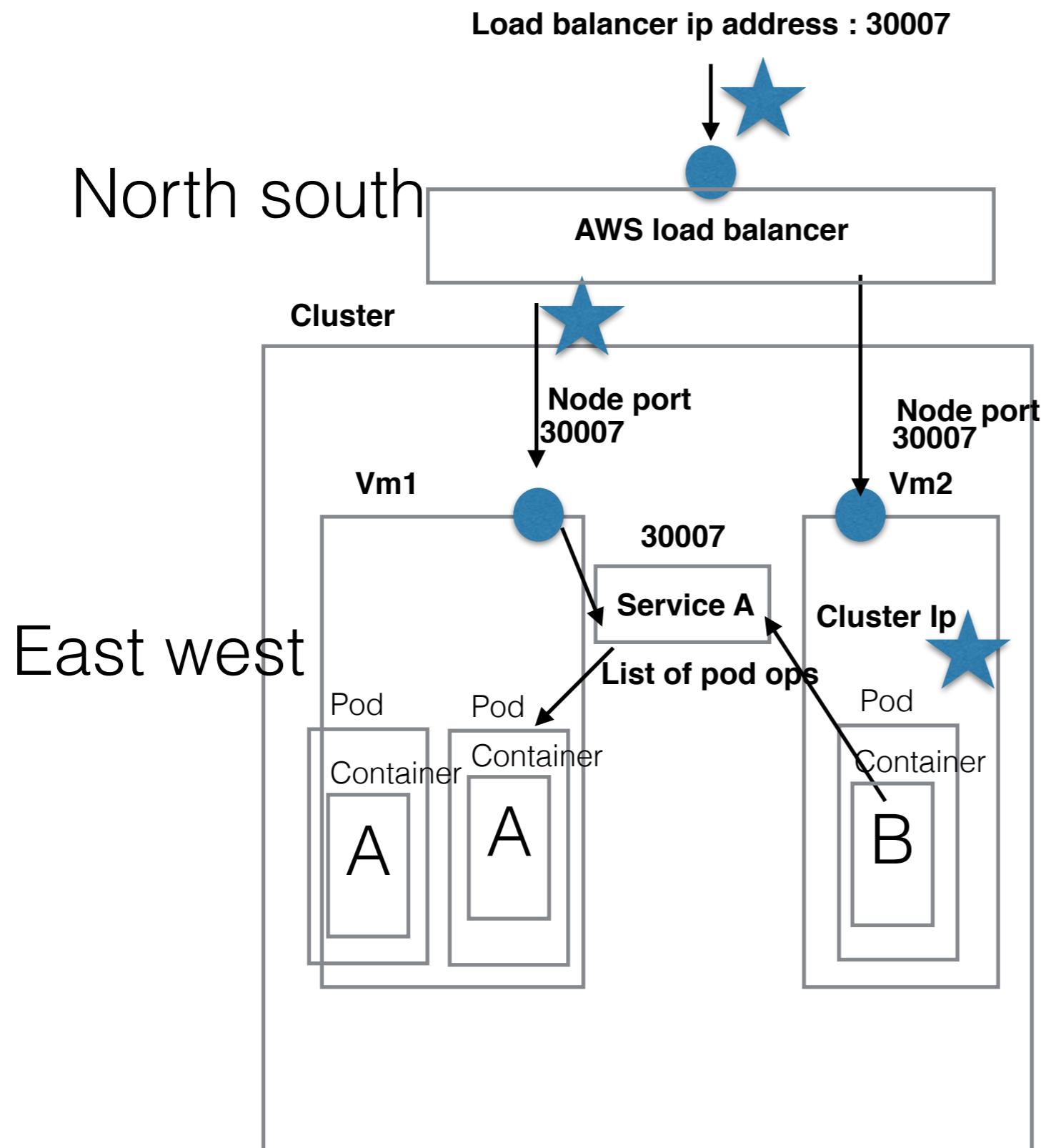


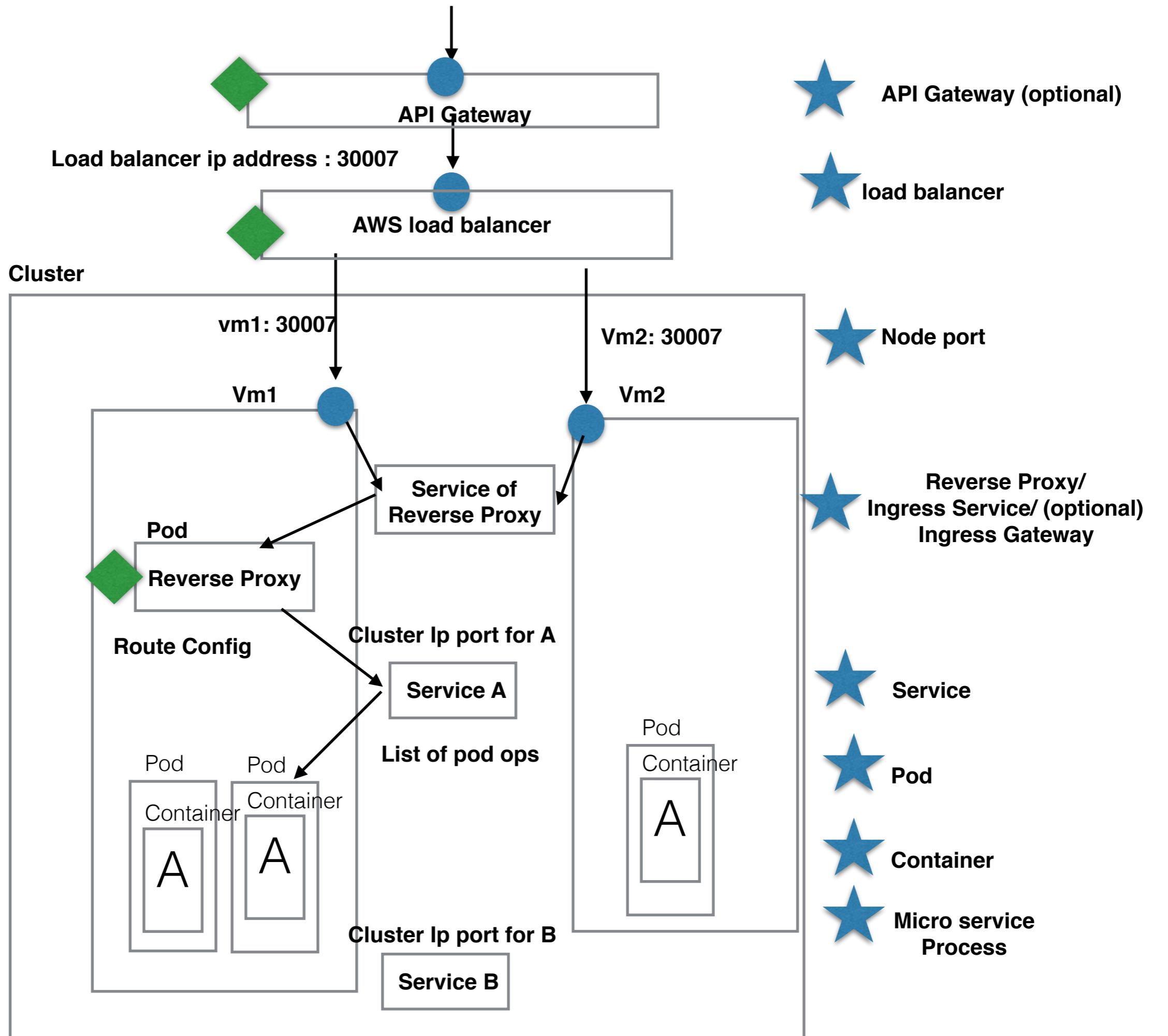
<http://server/microservice/api/param...>

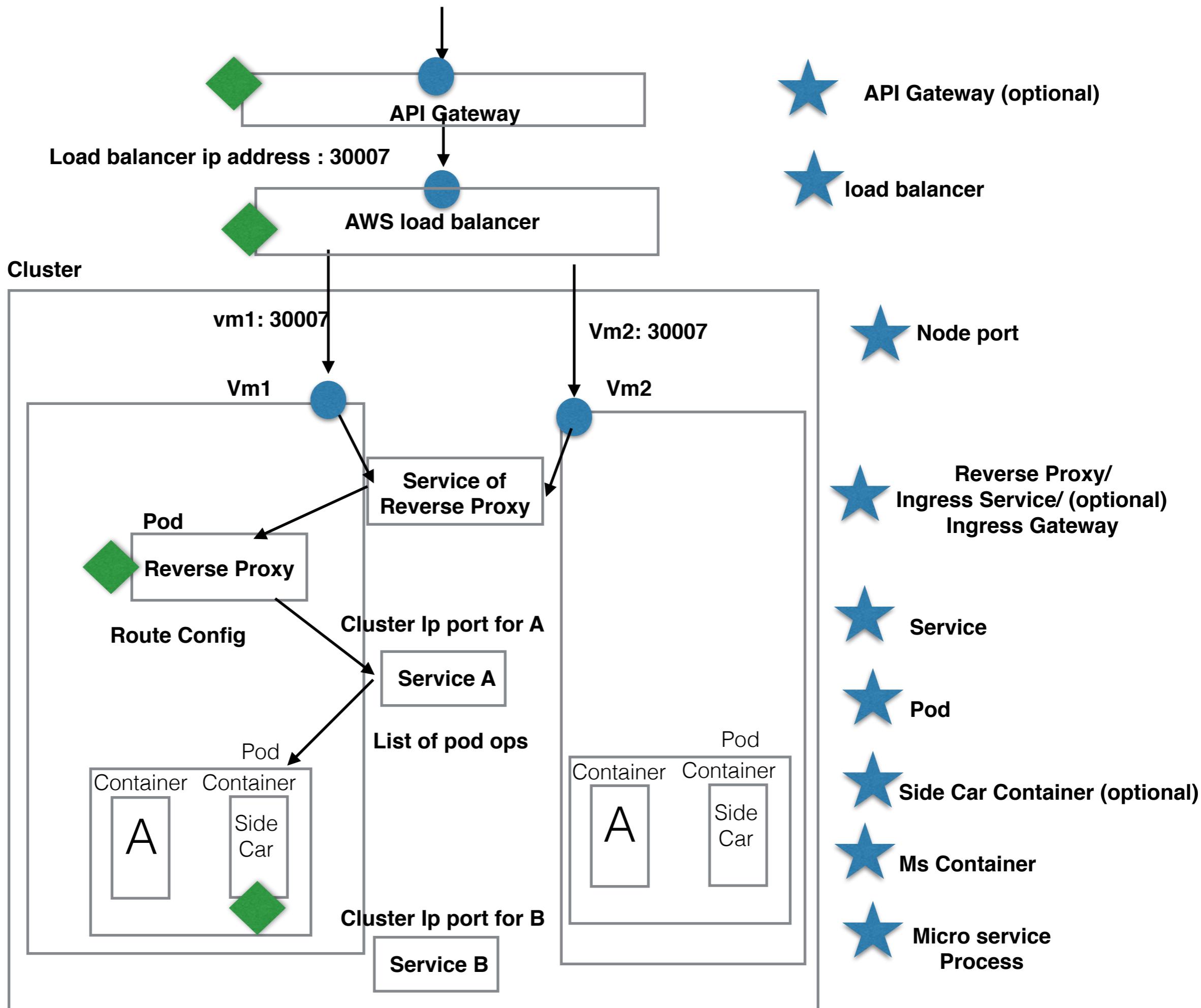


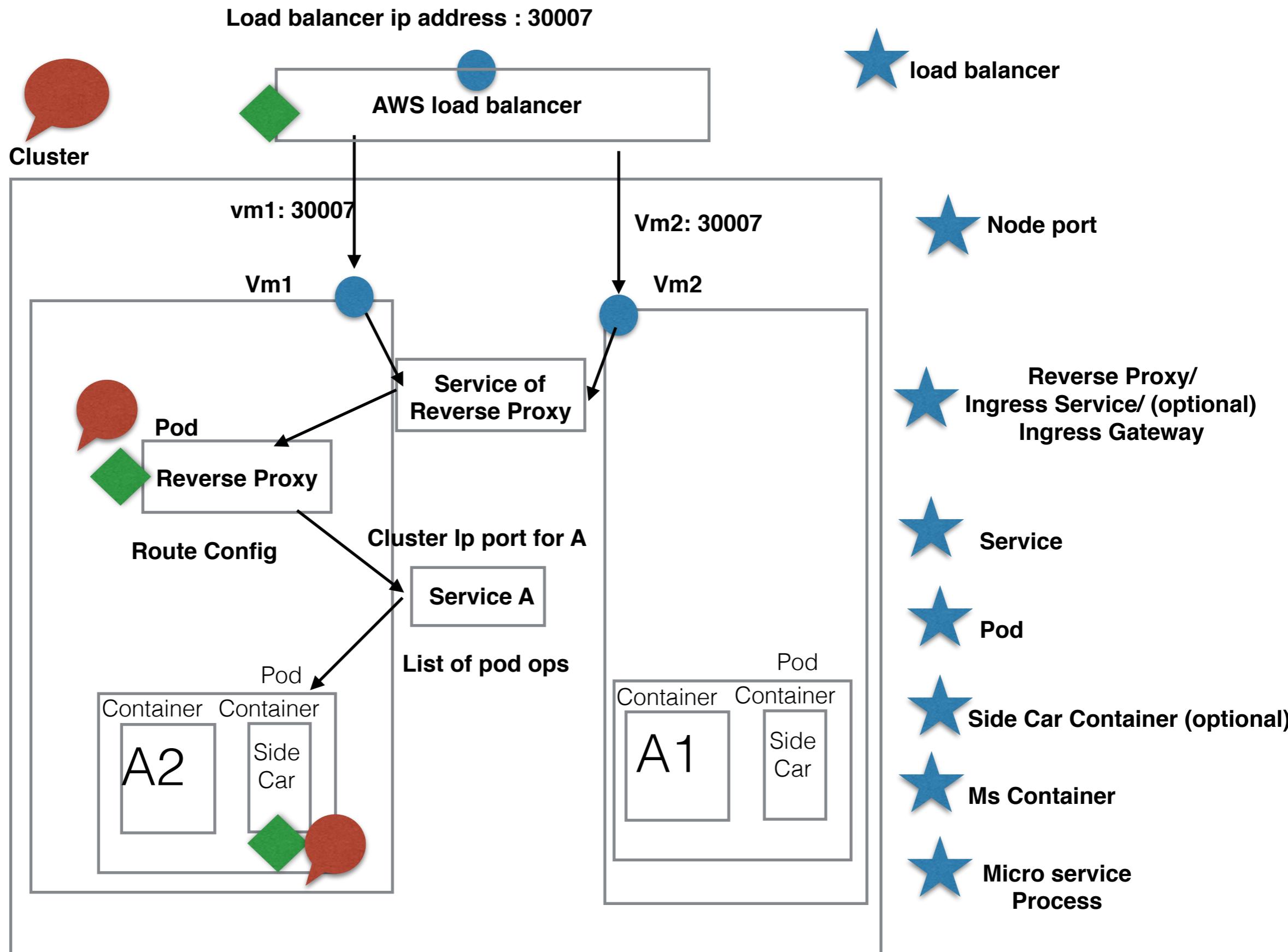


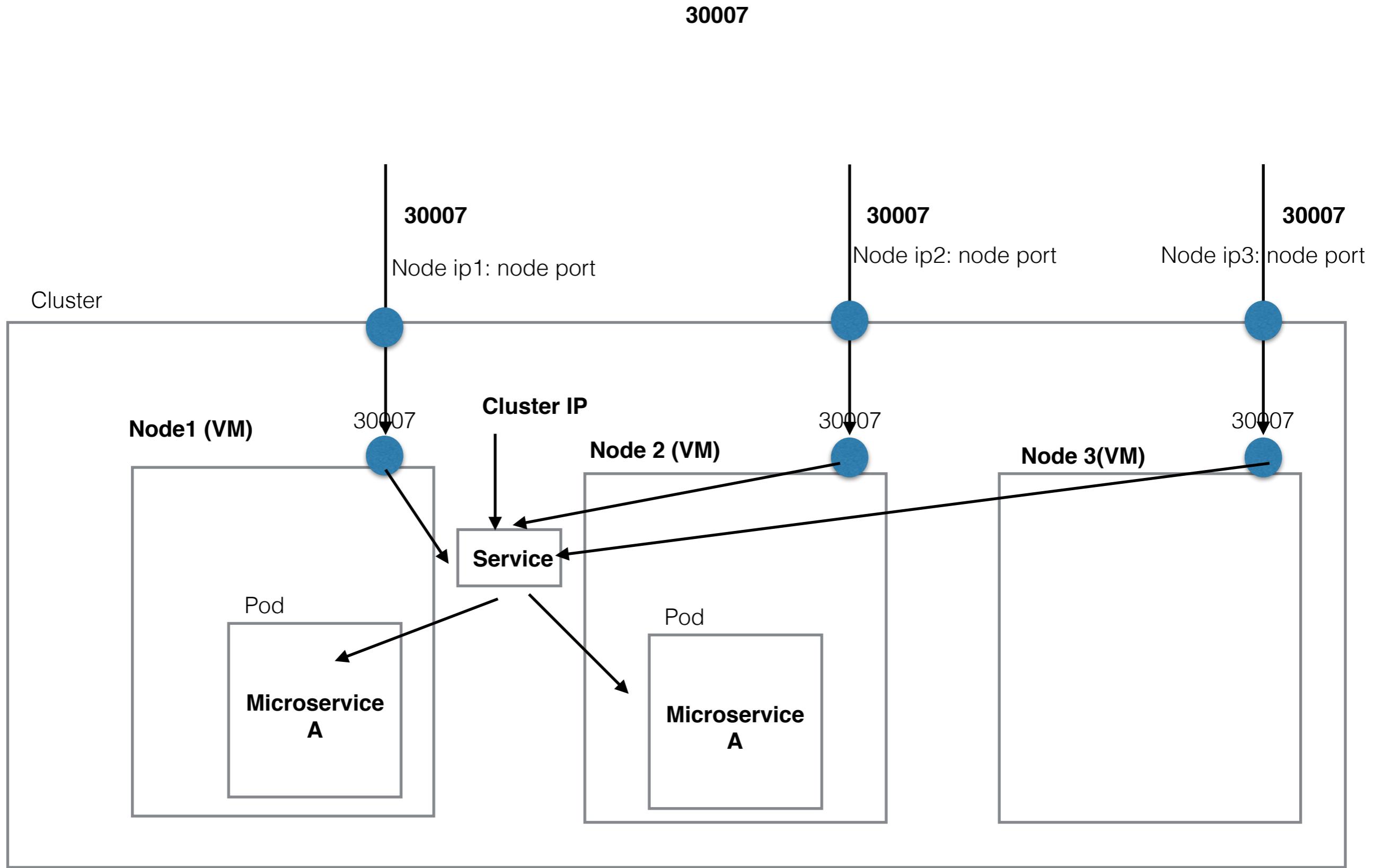




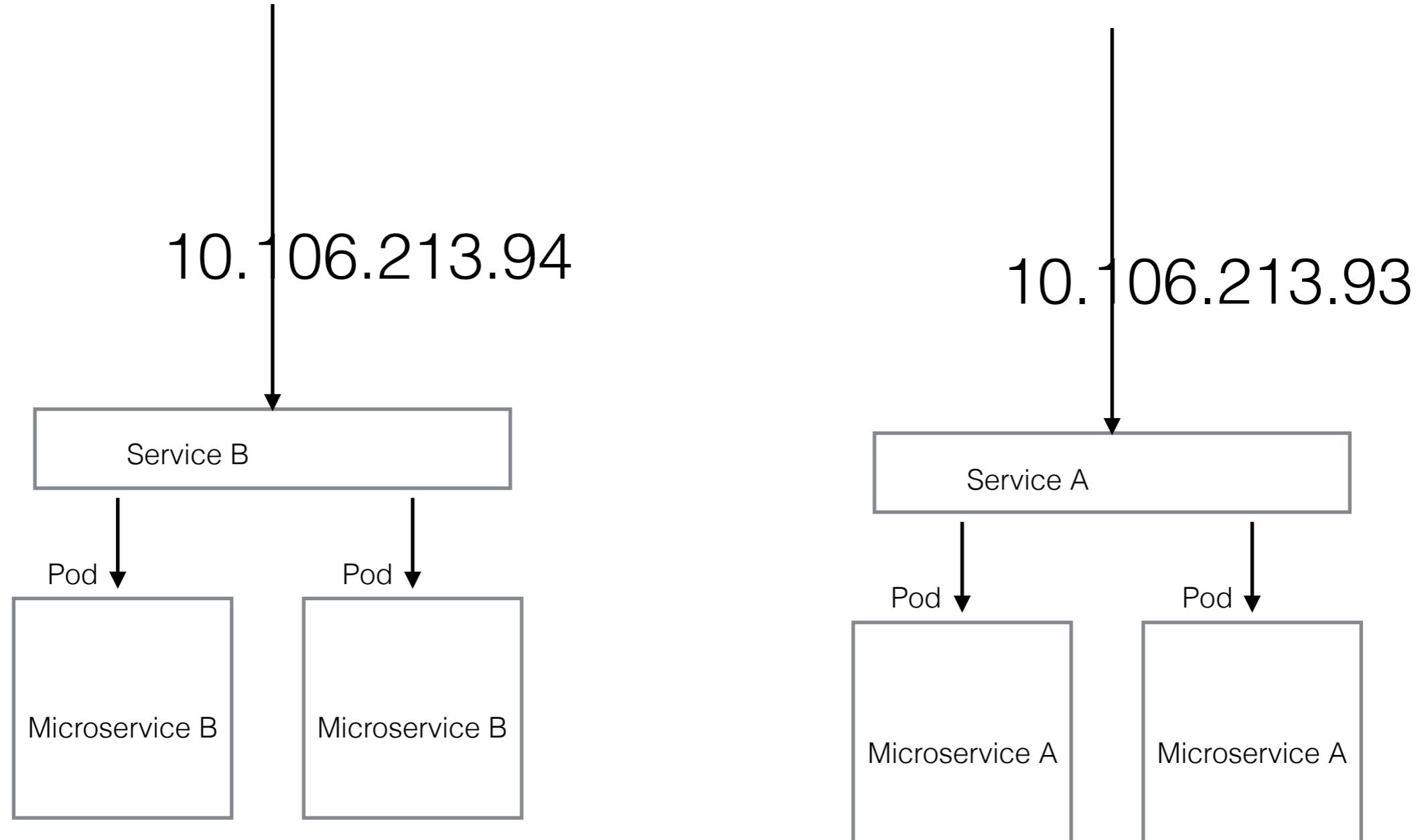


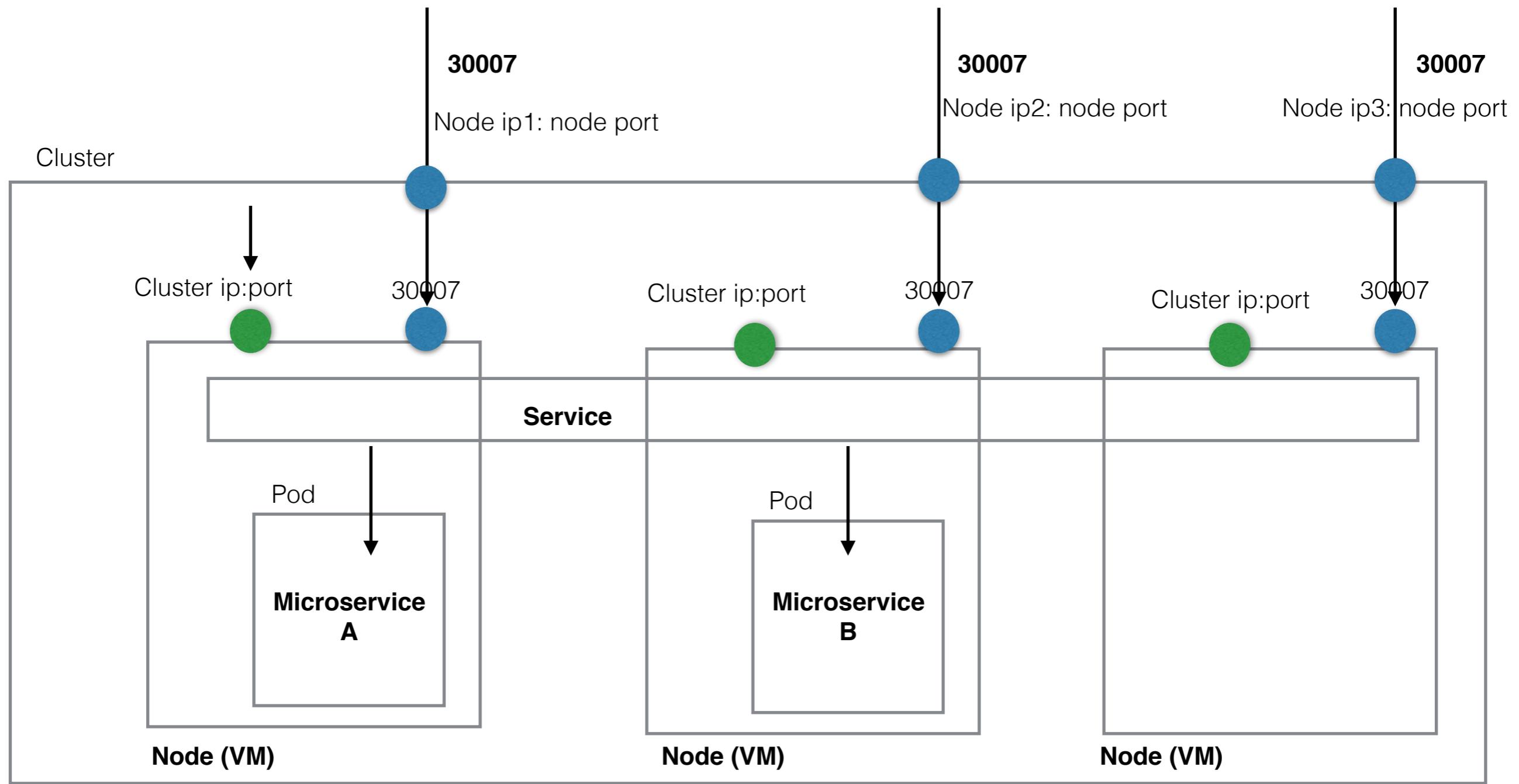


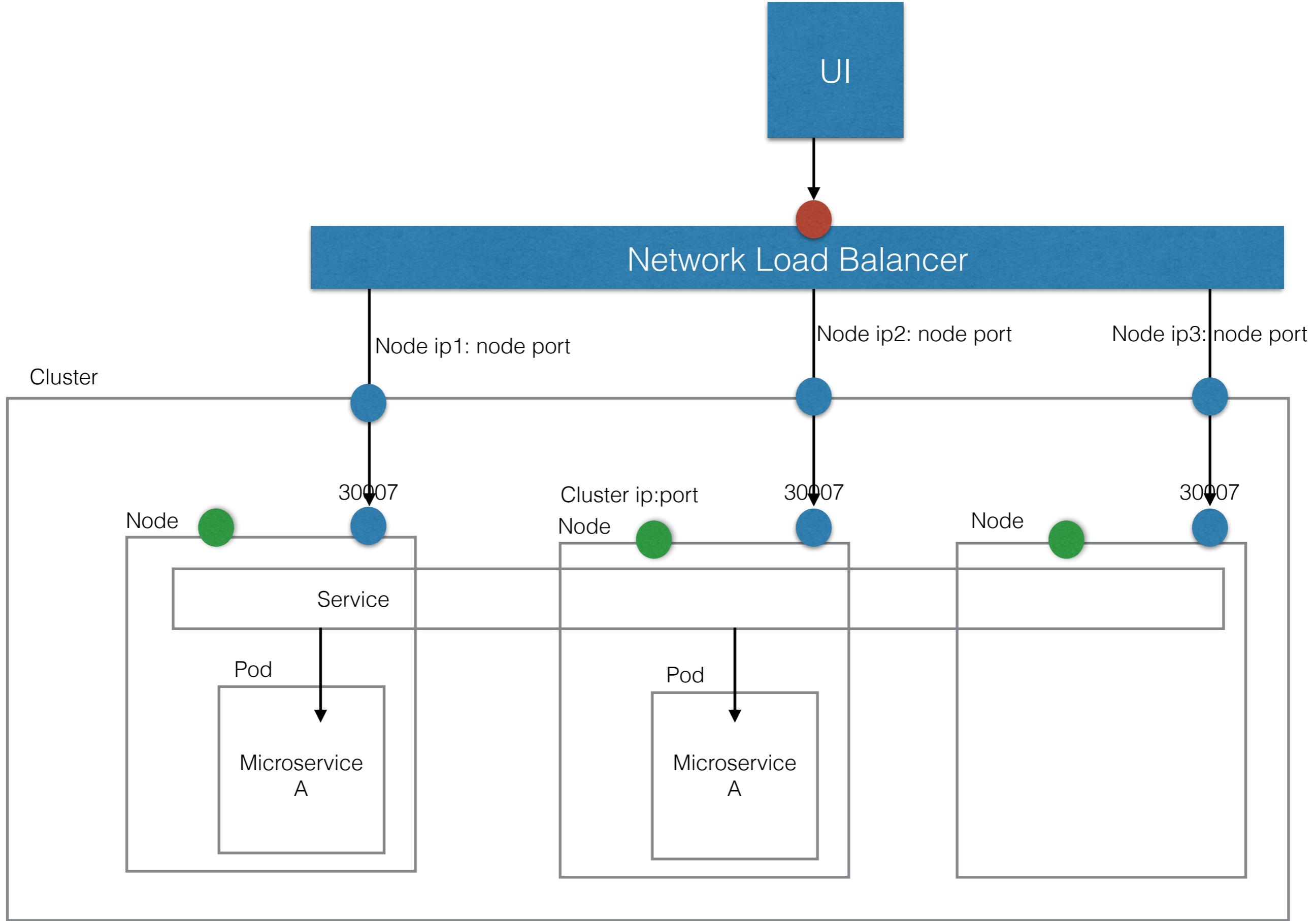


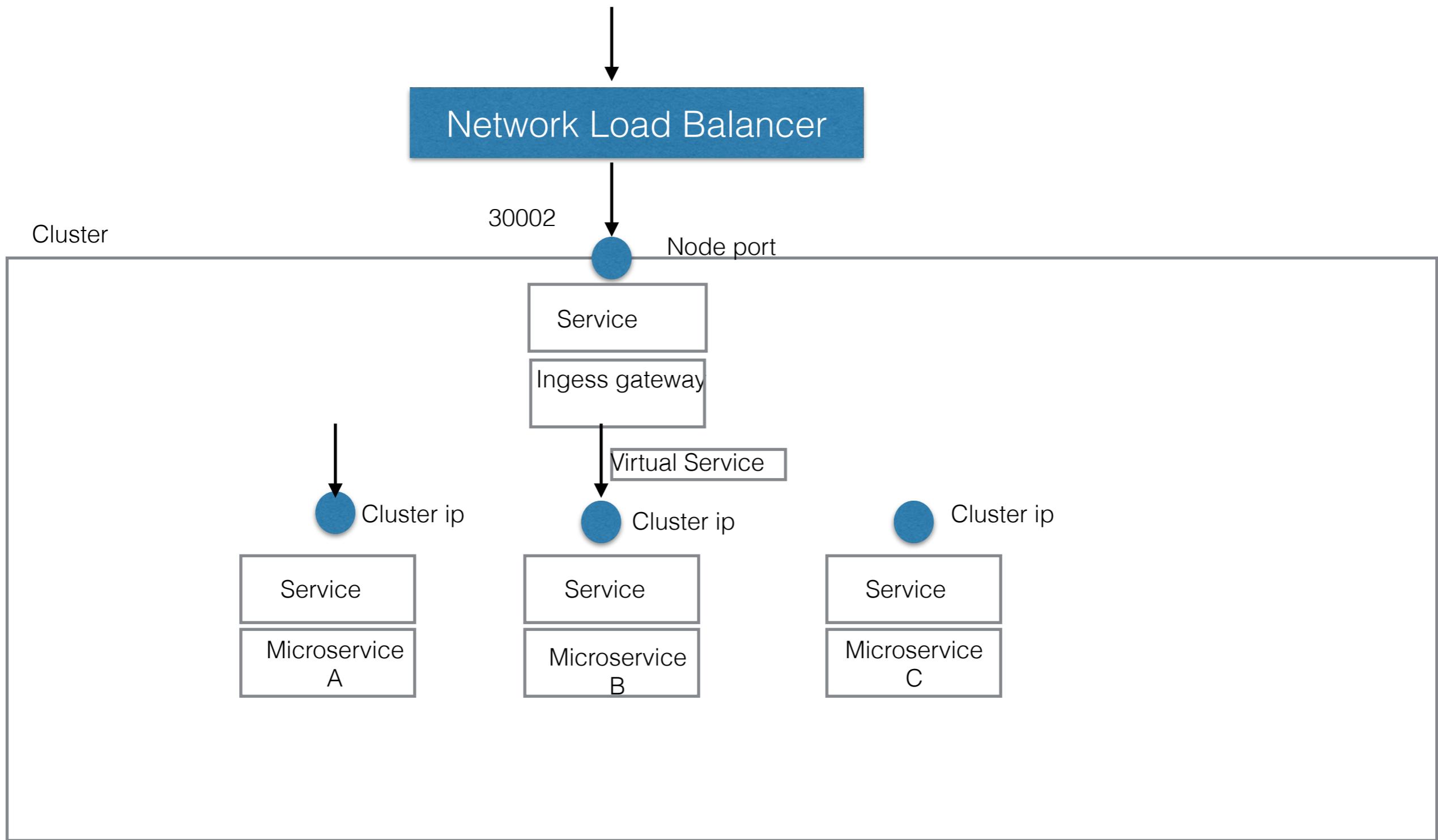


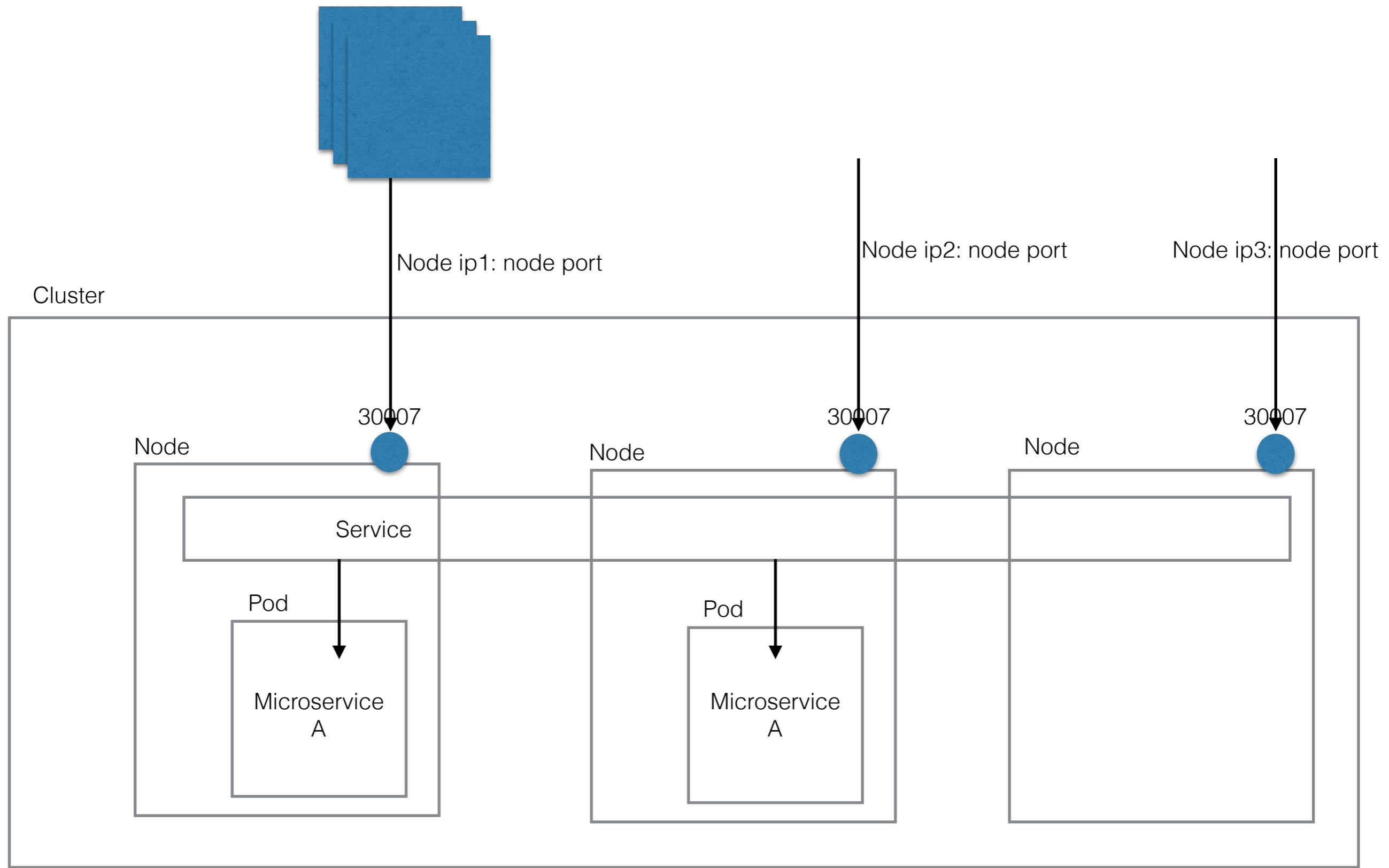
Cluster

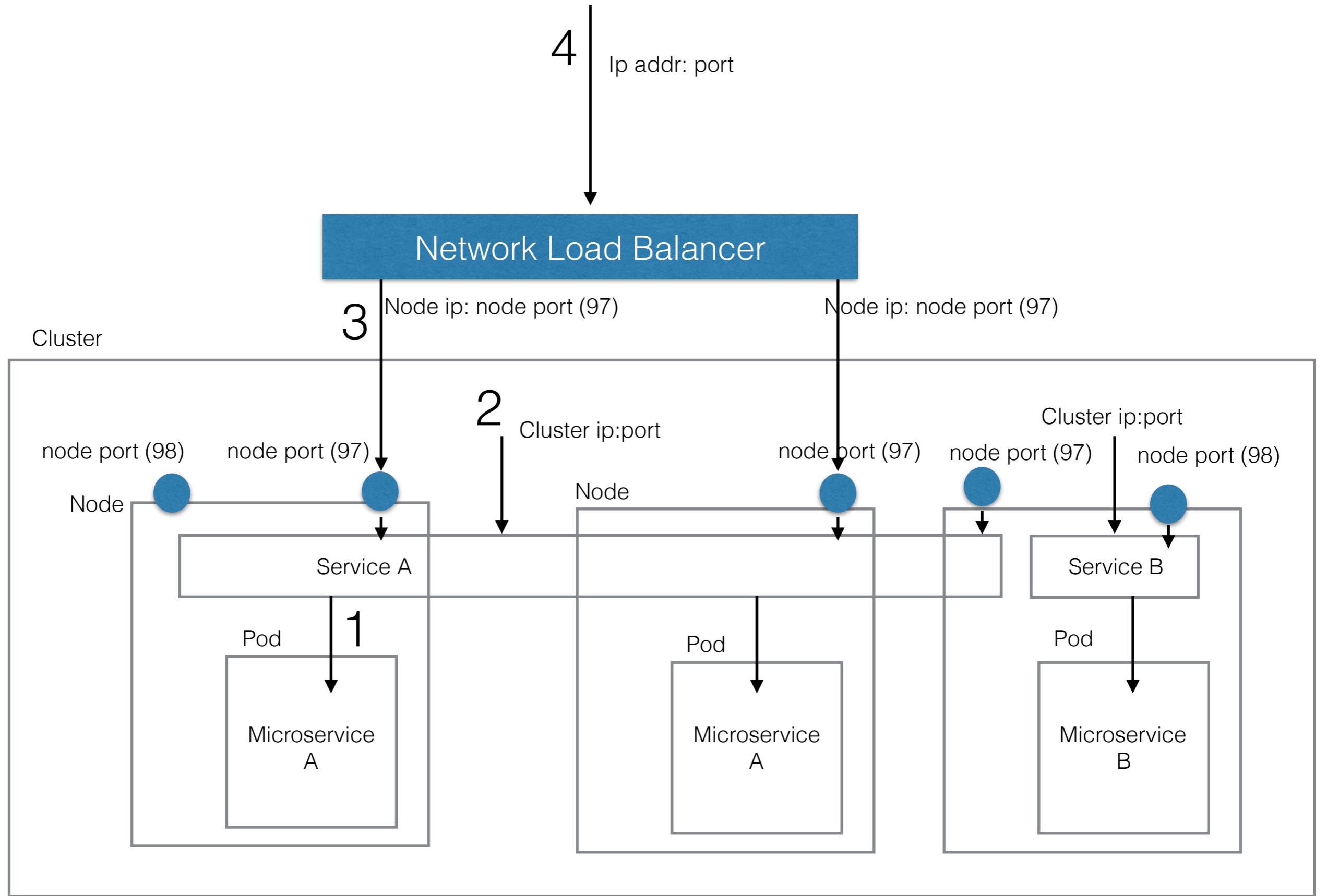


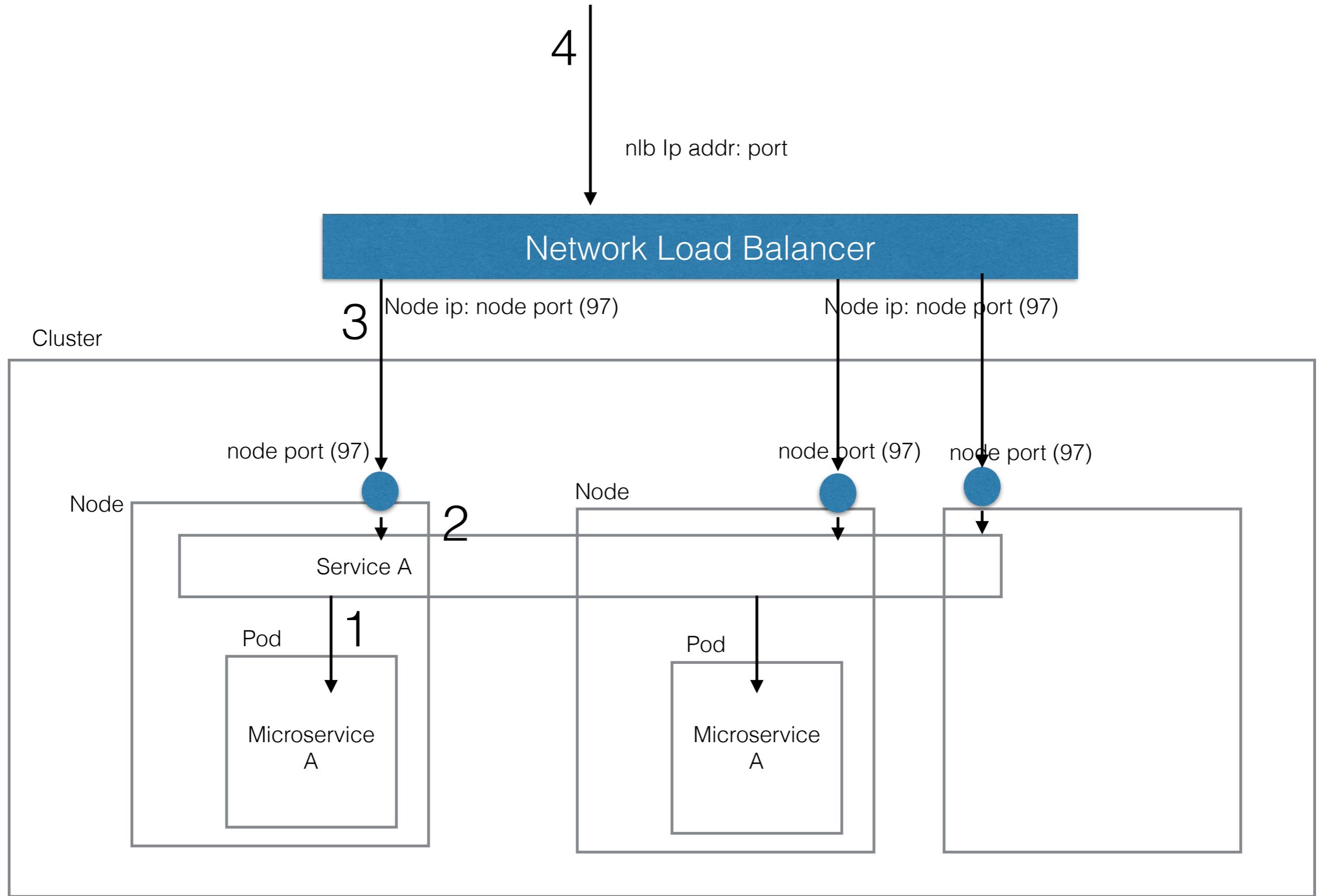


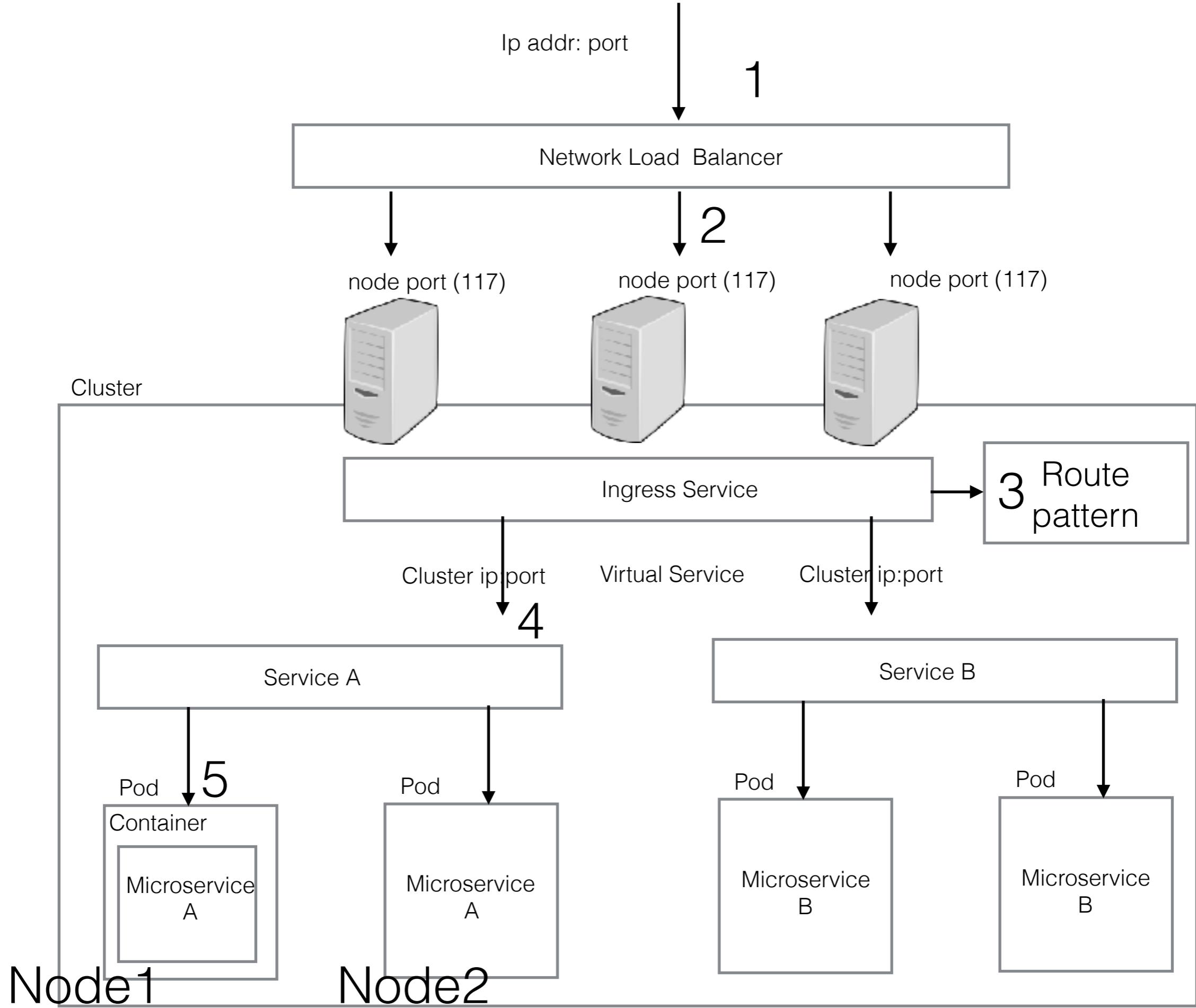




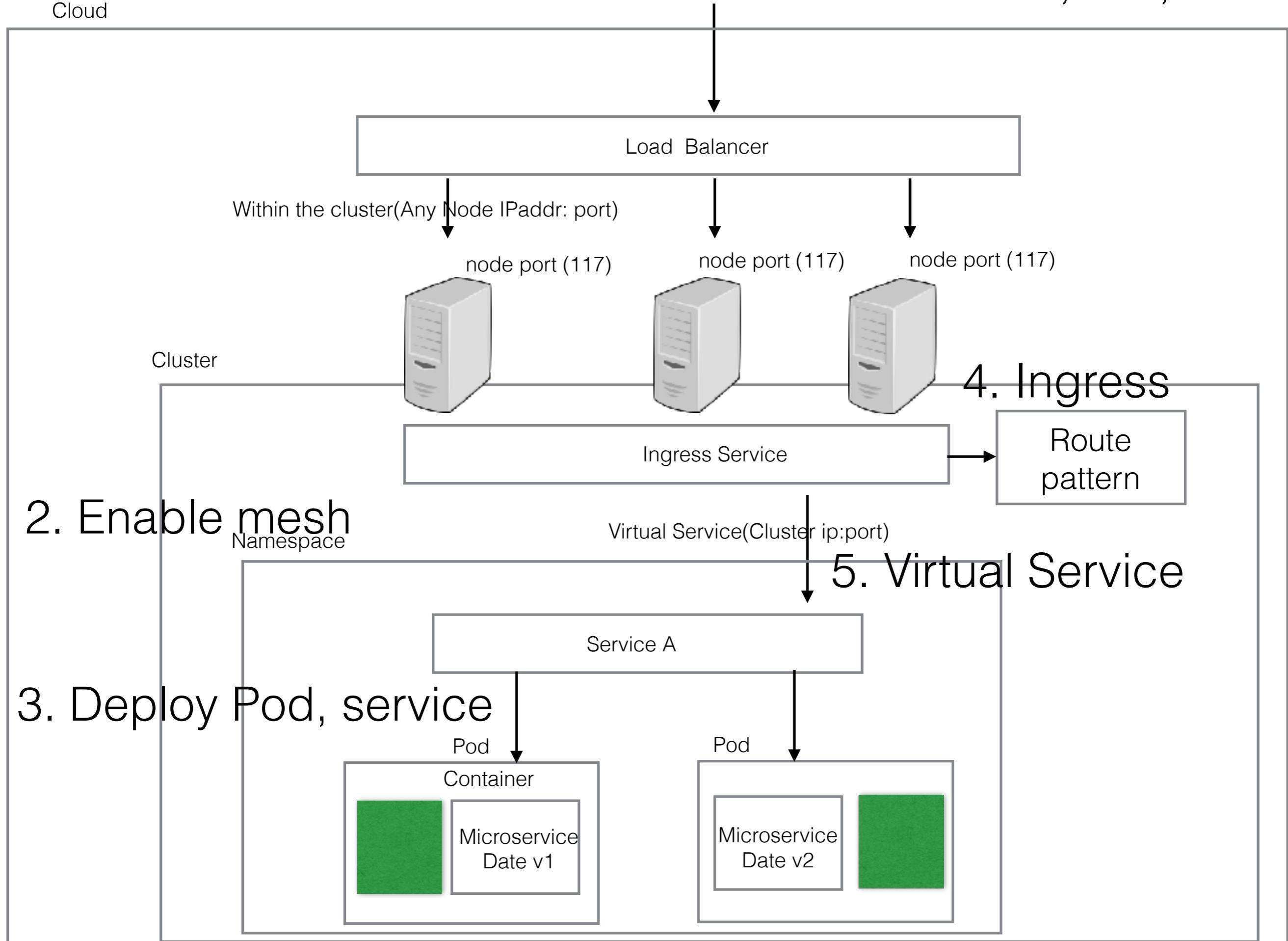


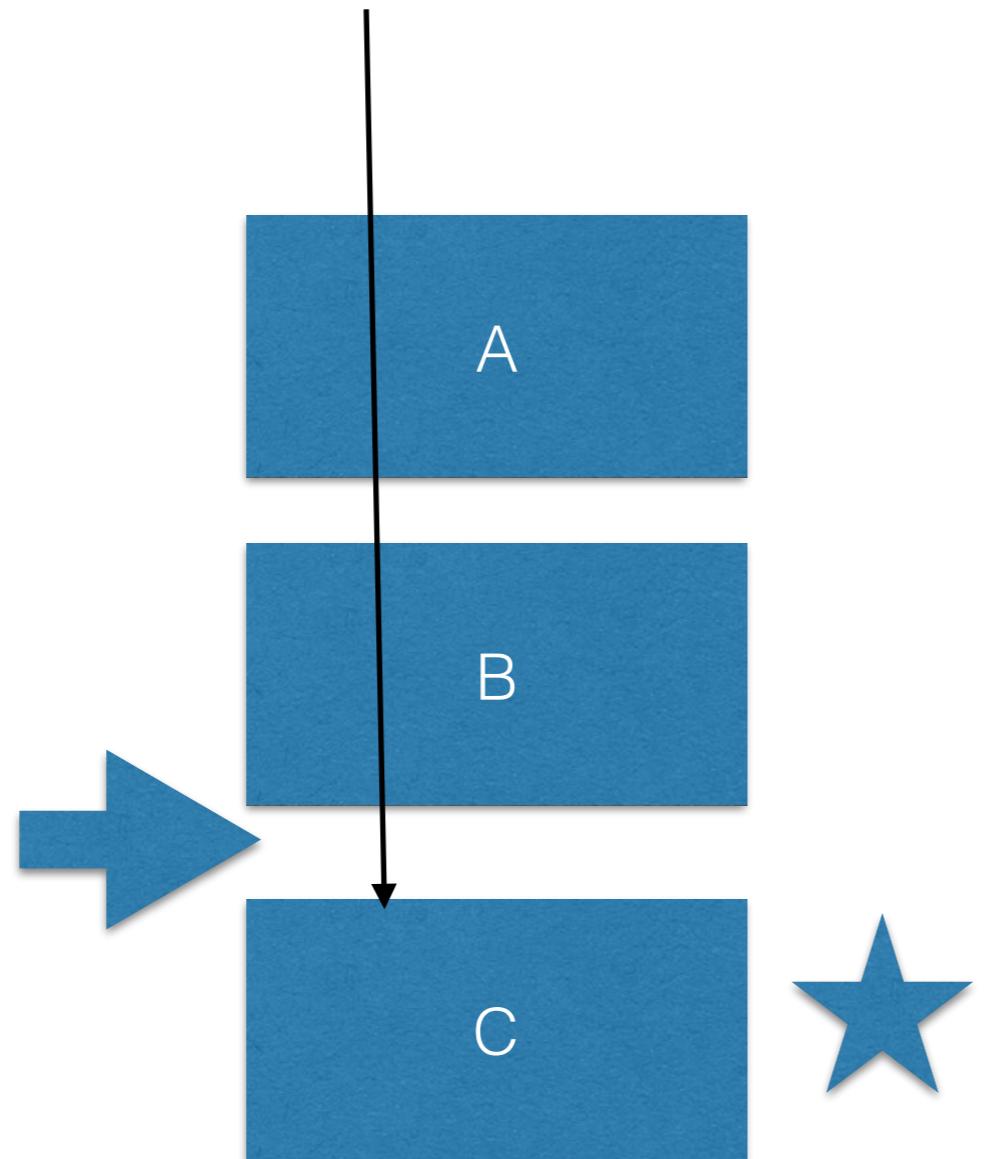


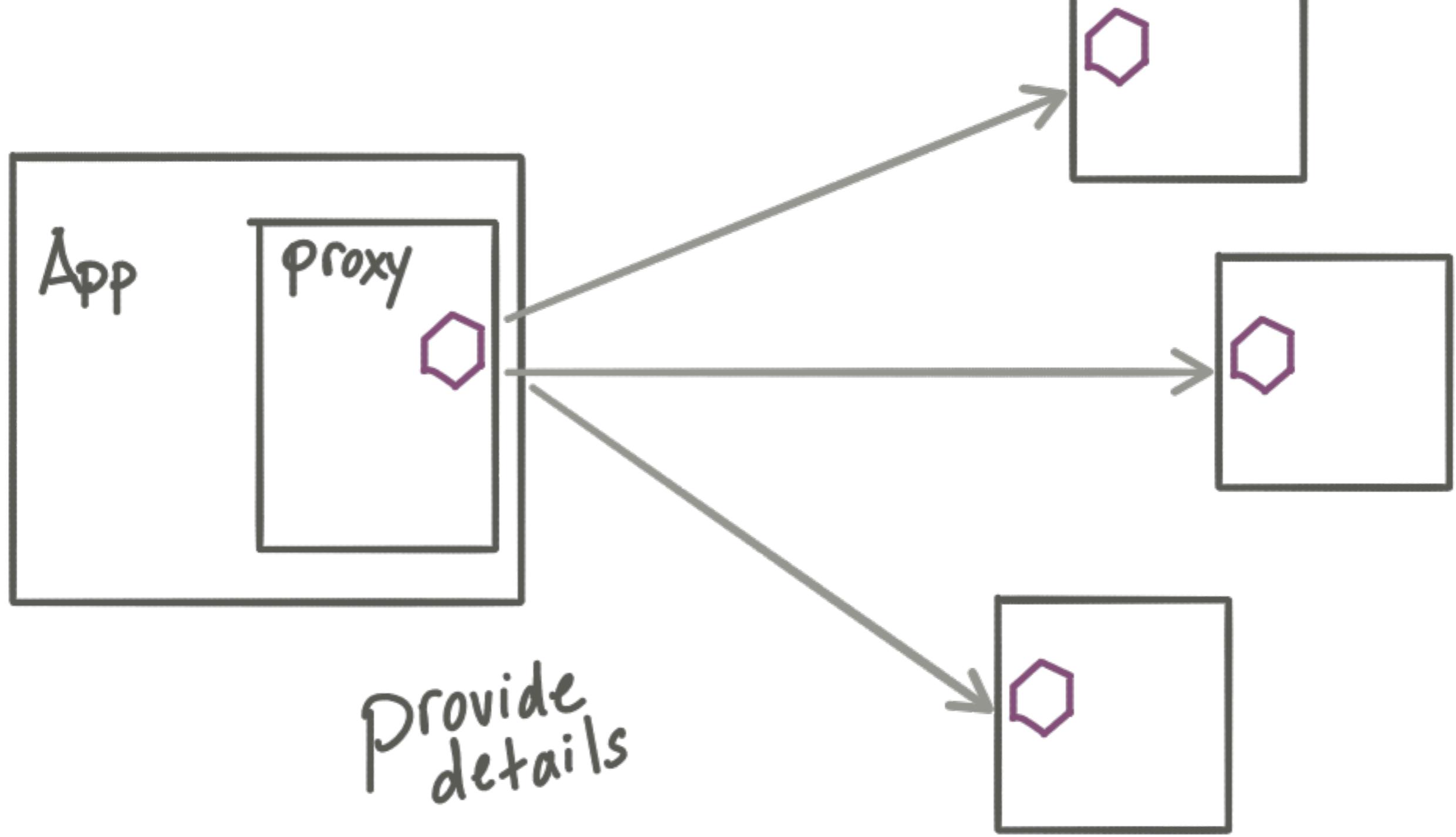




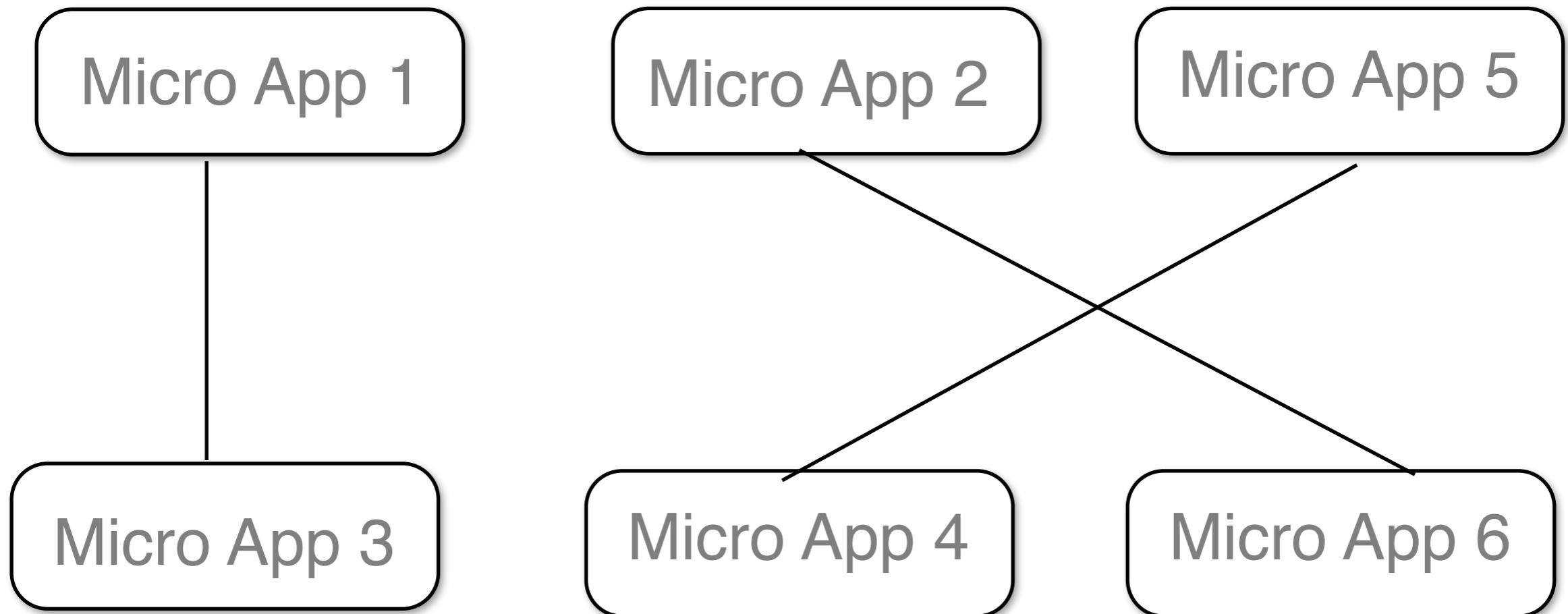
# 1. Docker, k8s, Istio

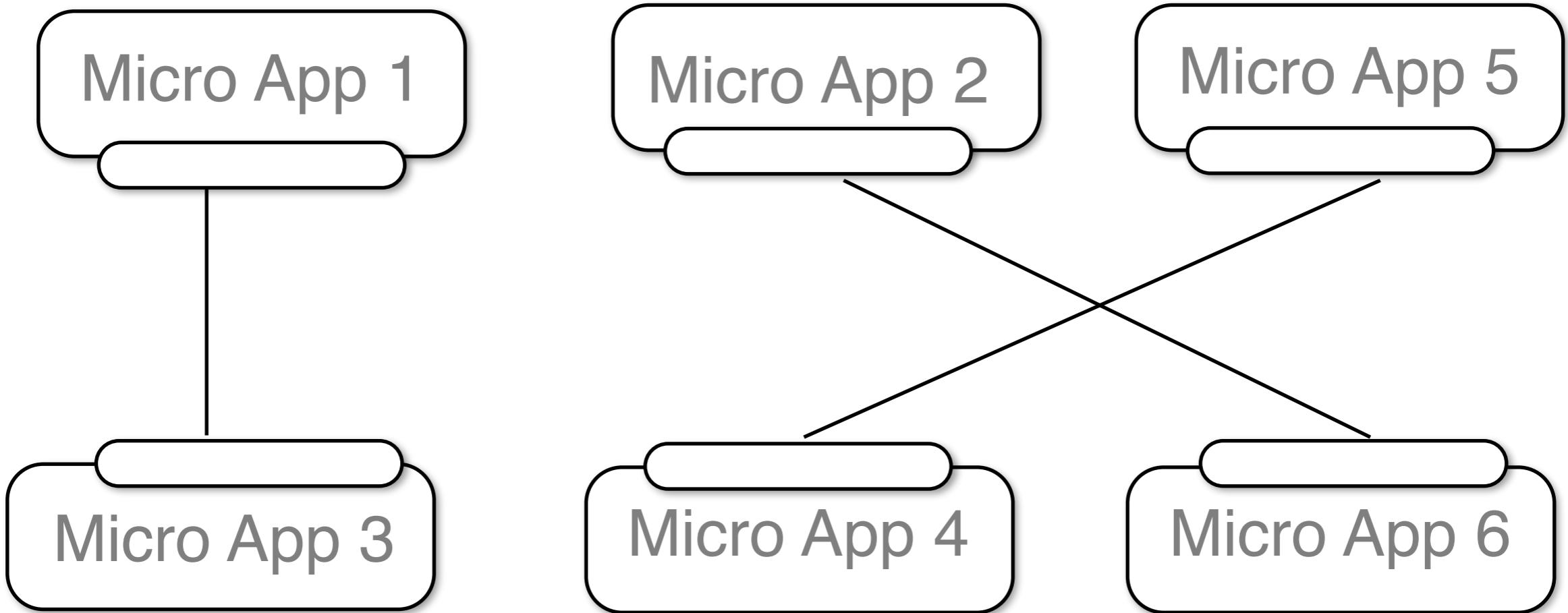


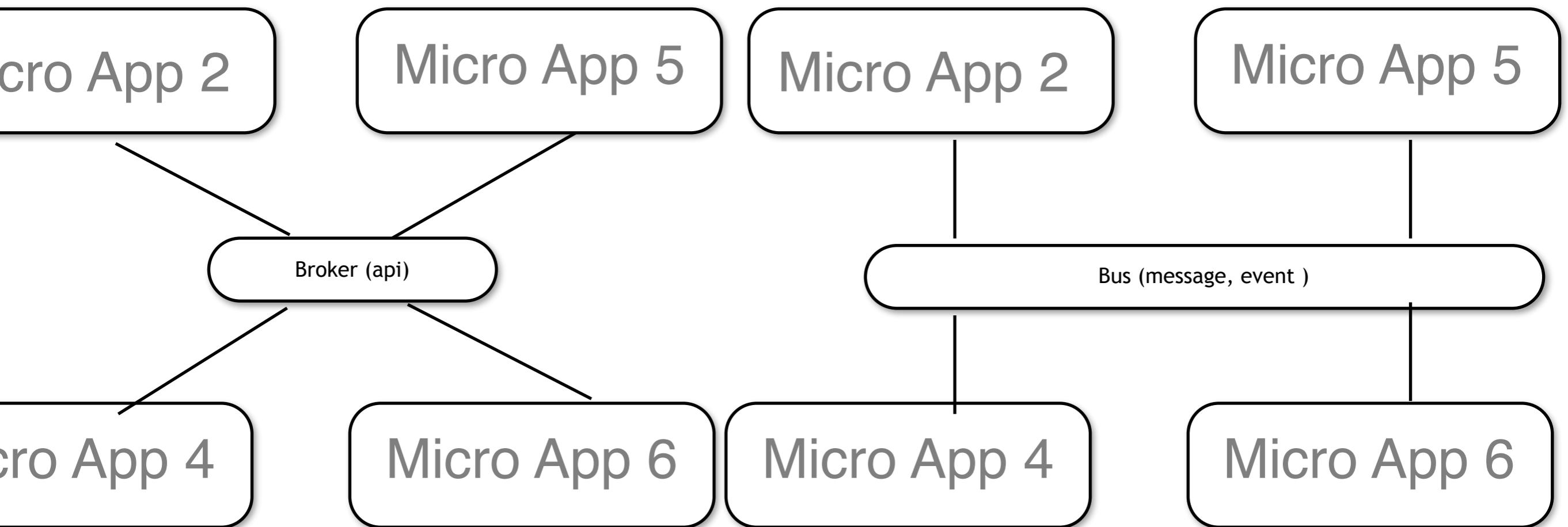
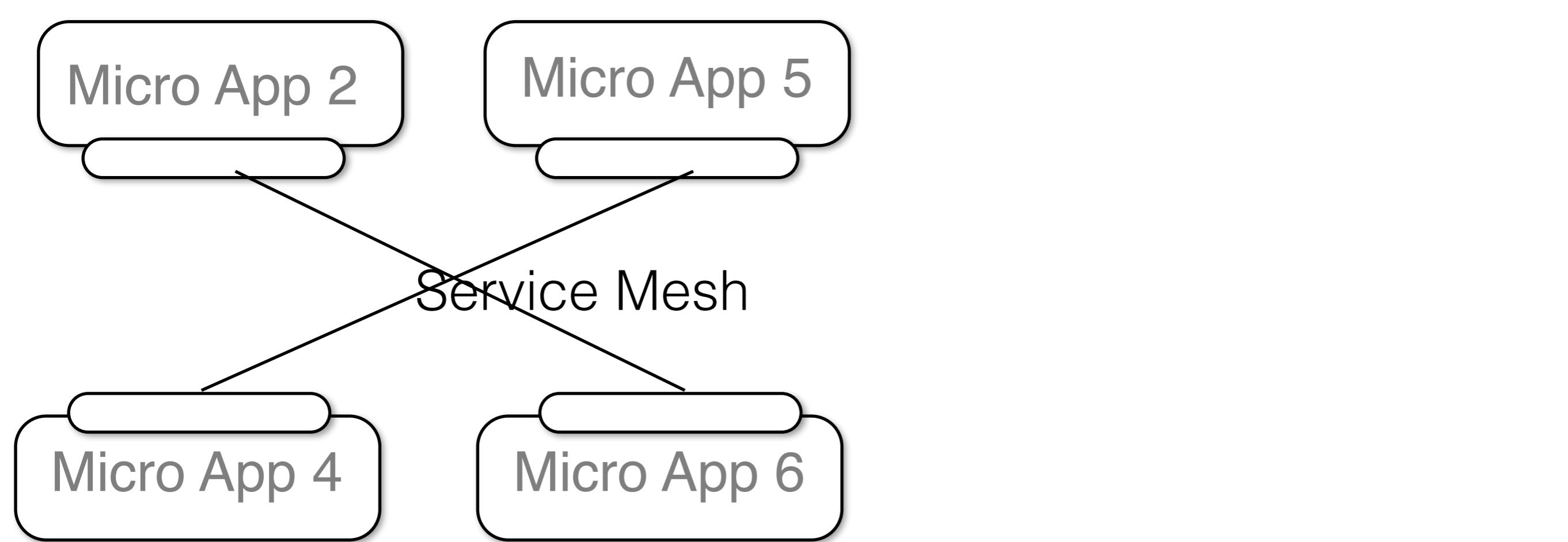


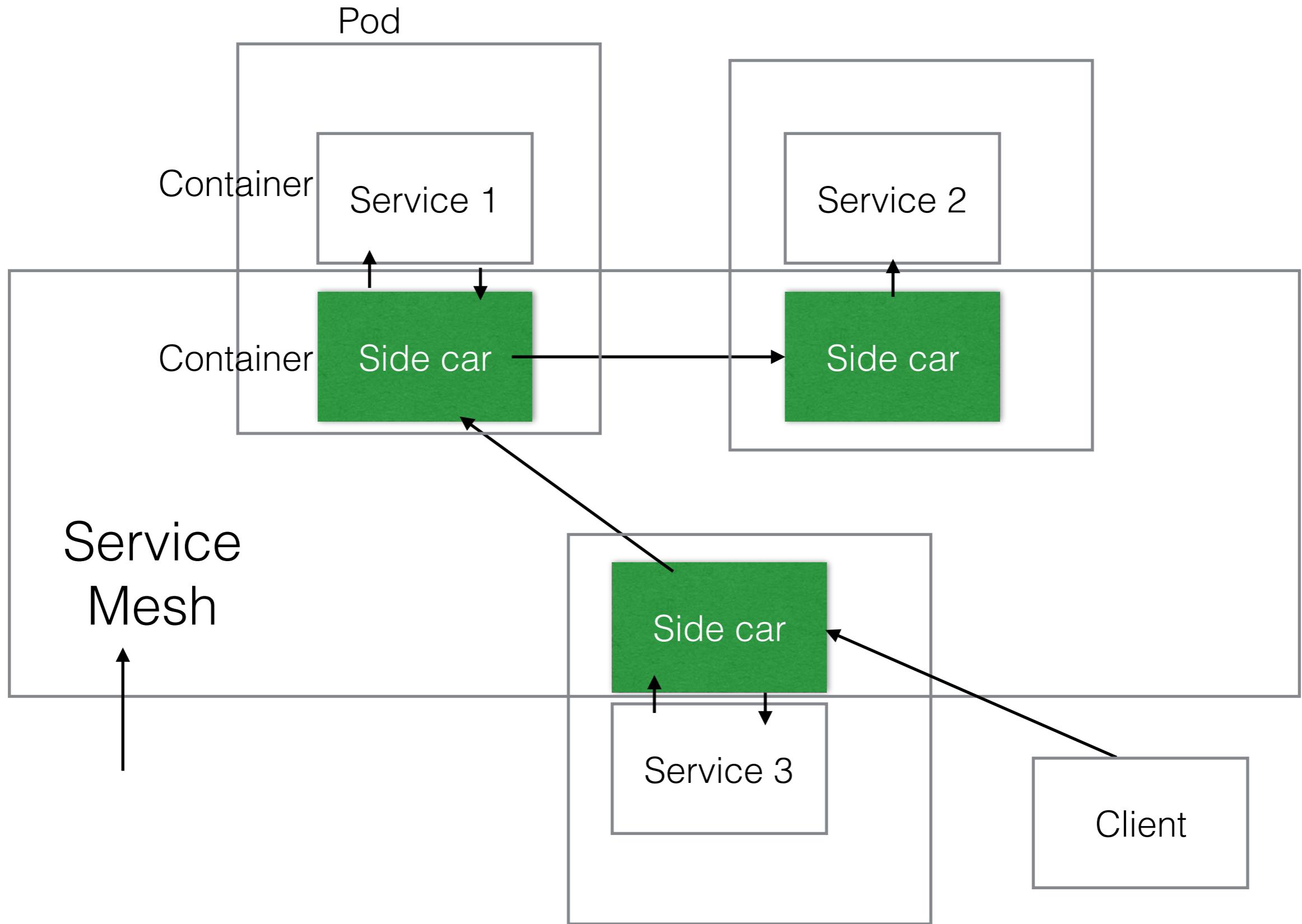


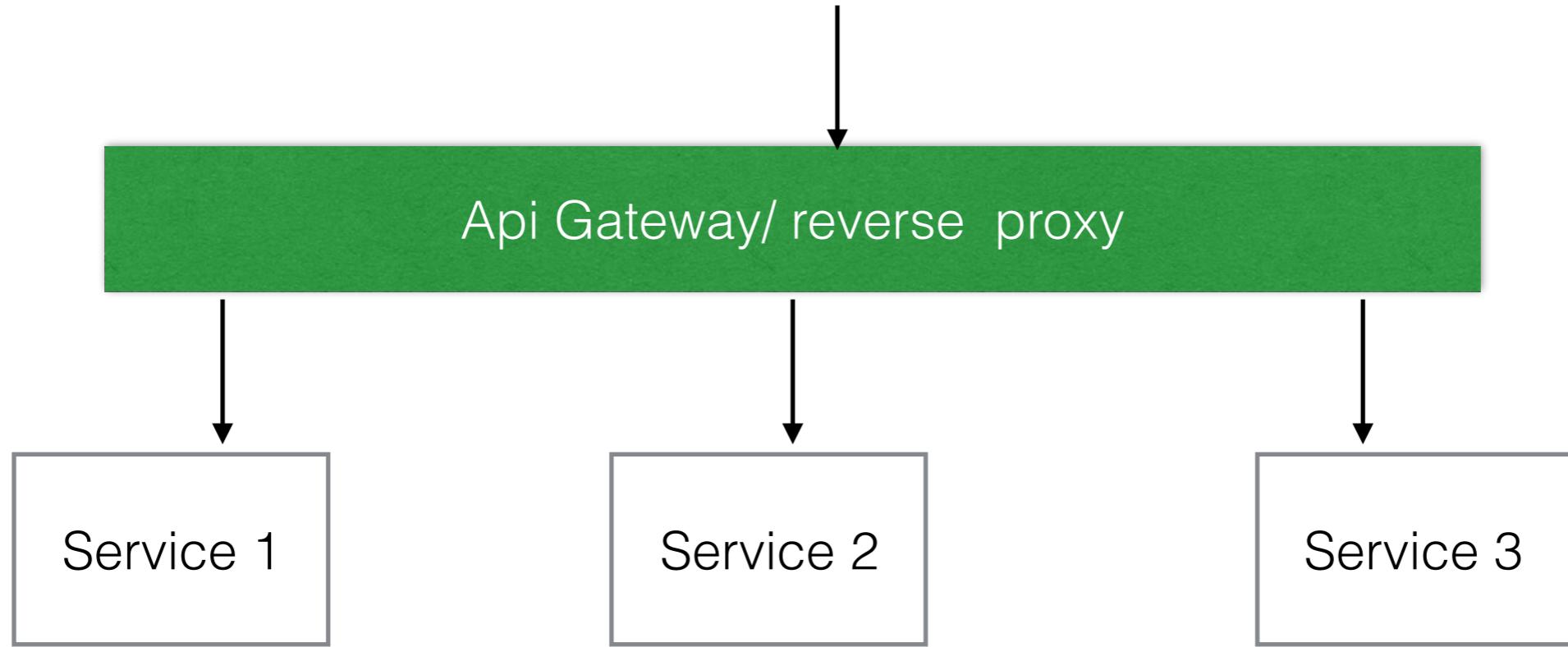
# Service Mesh

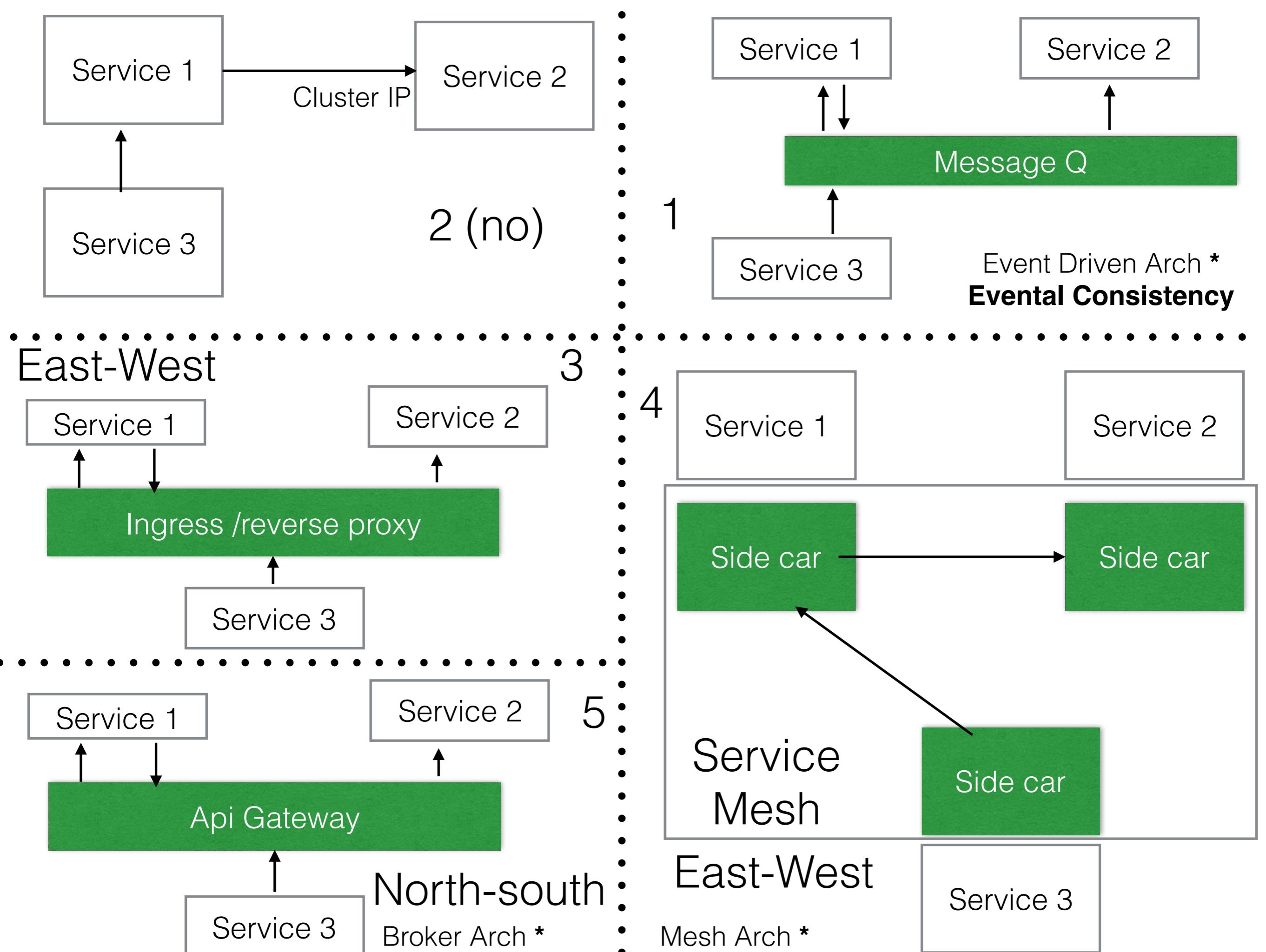


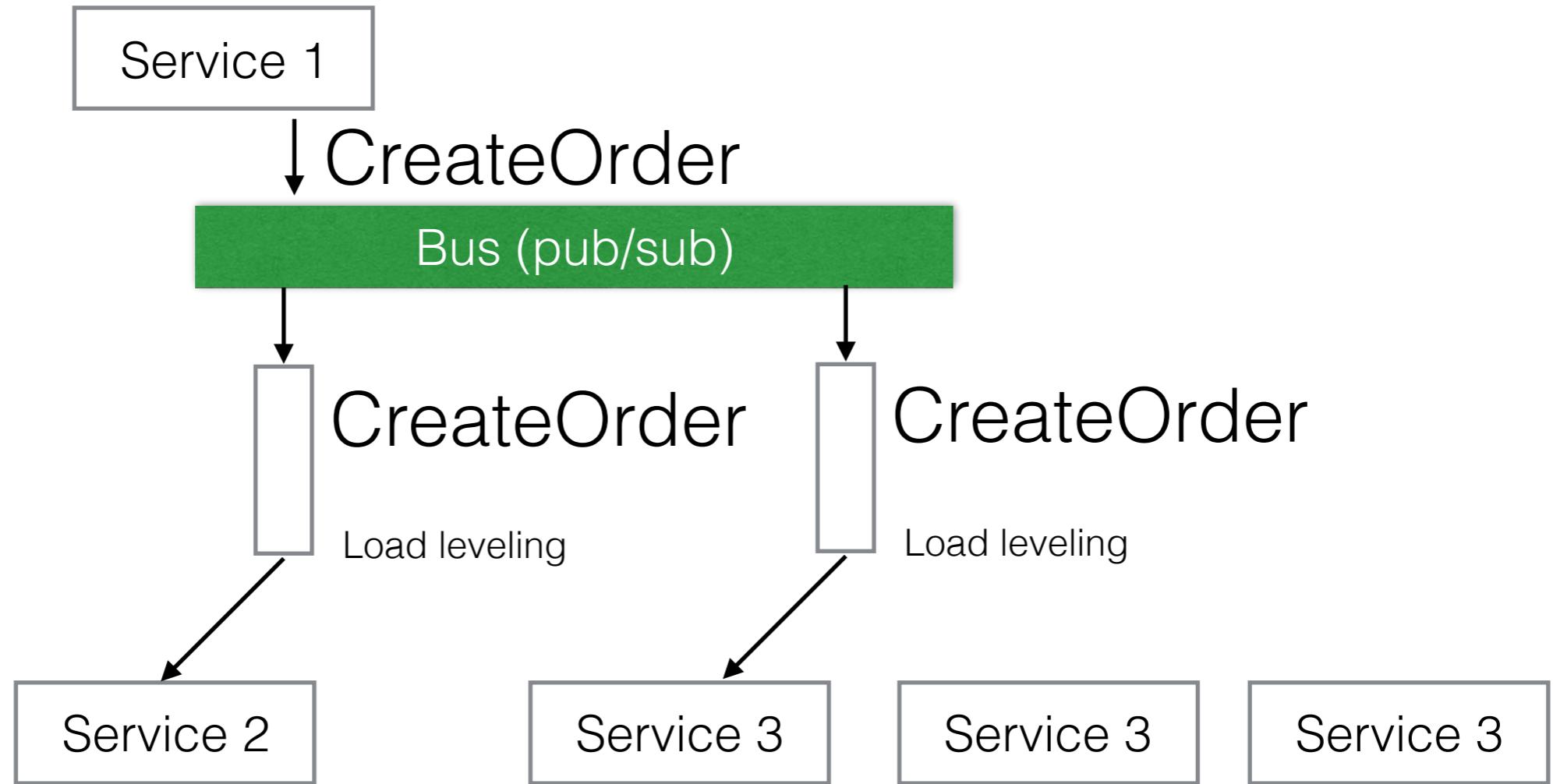


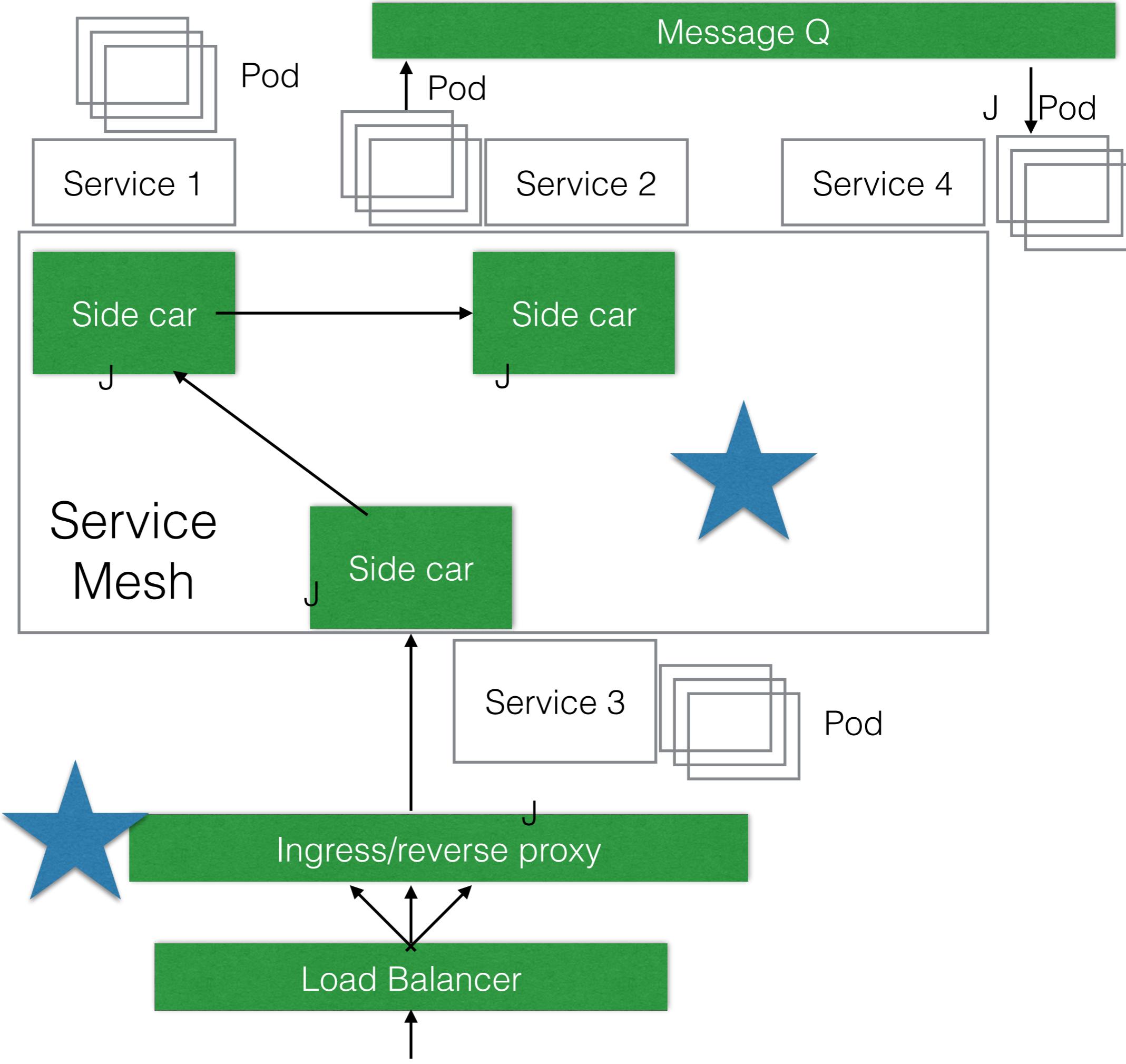




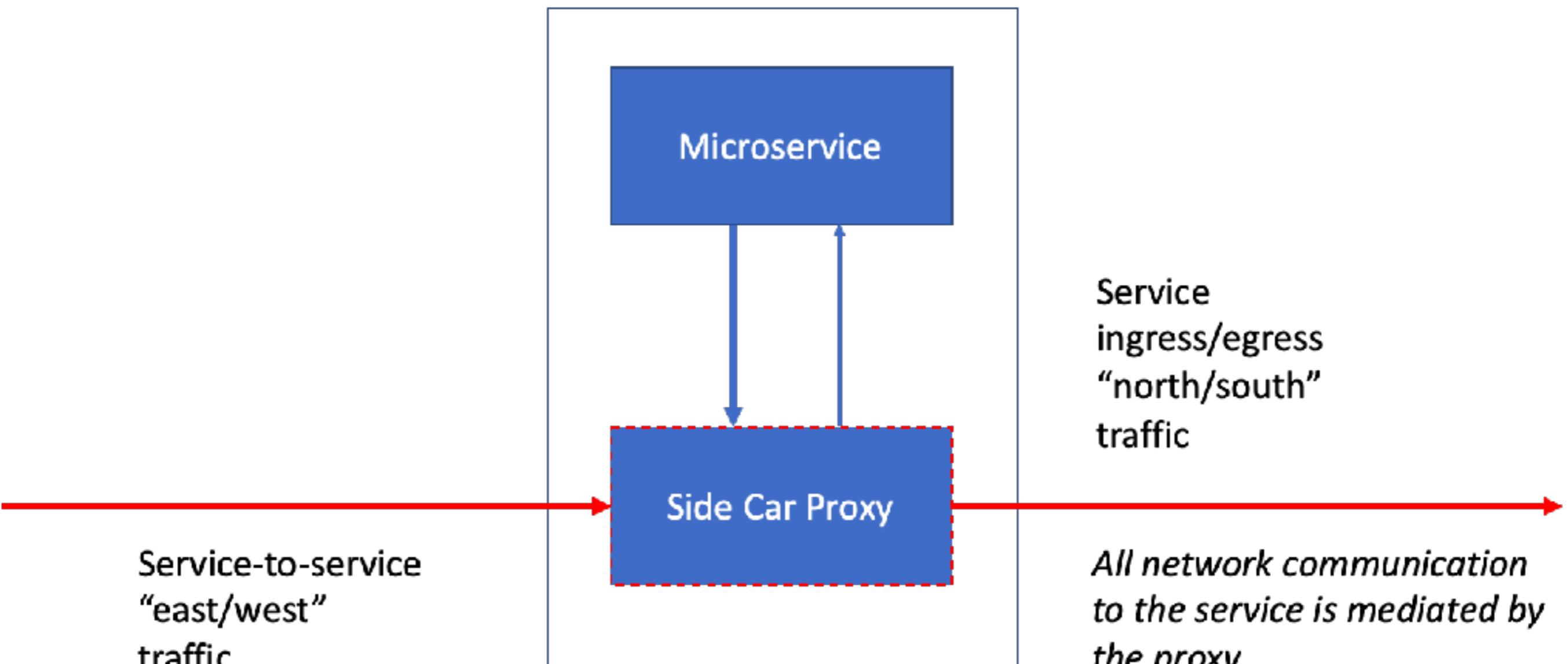








## ECS Task/Kubernetes Pod



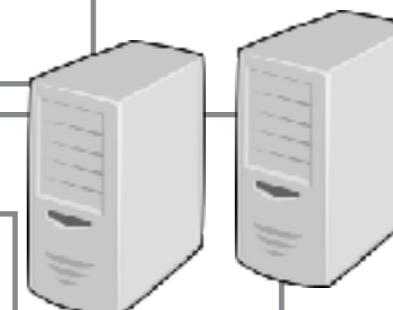
http://172.17.0.22:80/date

North- South

Cluster



LoadBalancer : L4 (tcp)



Ingress : L7 (http)

http://10.97.33.128:8080/date

East - west

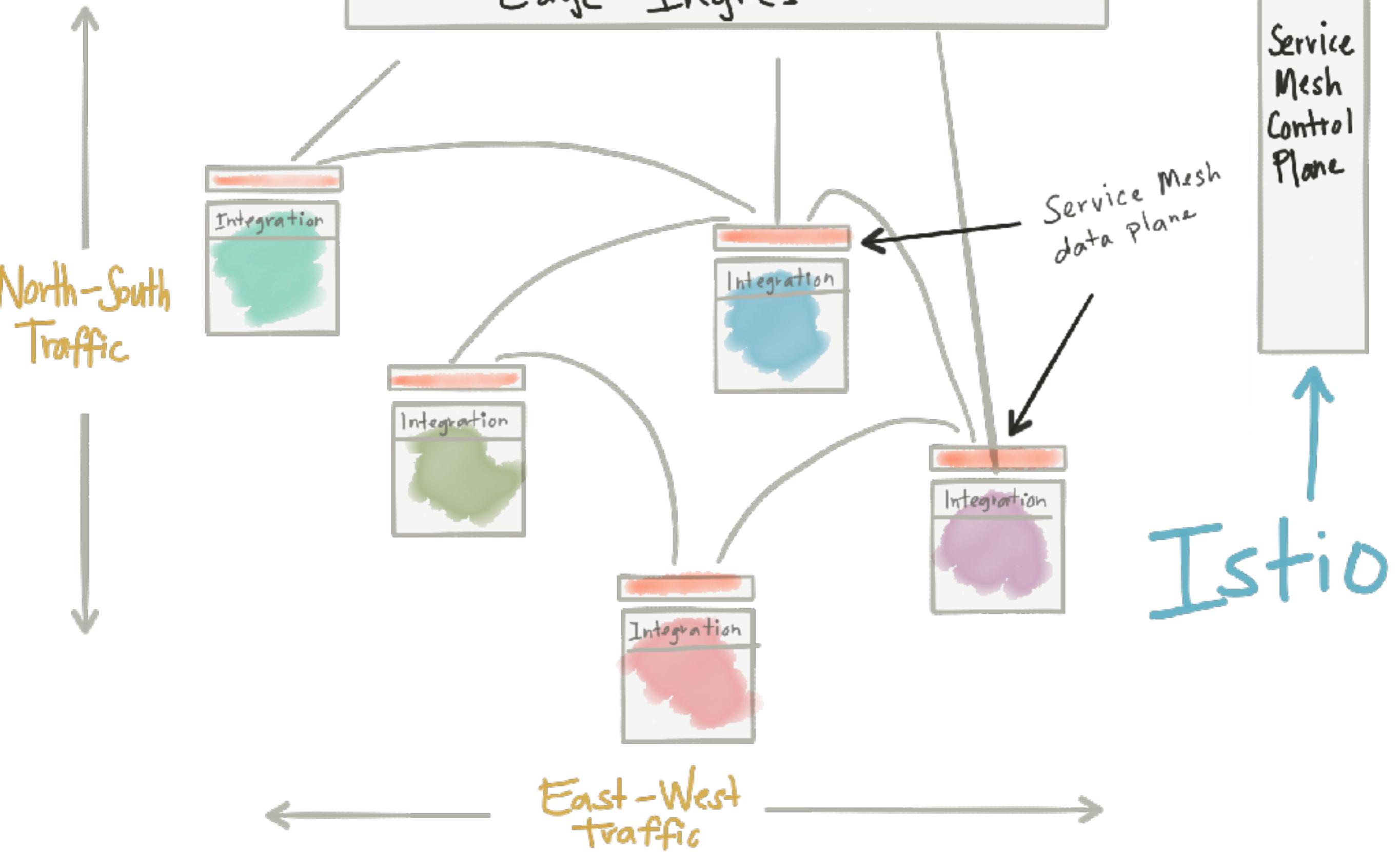
Service

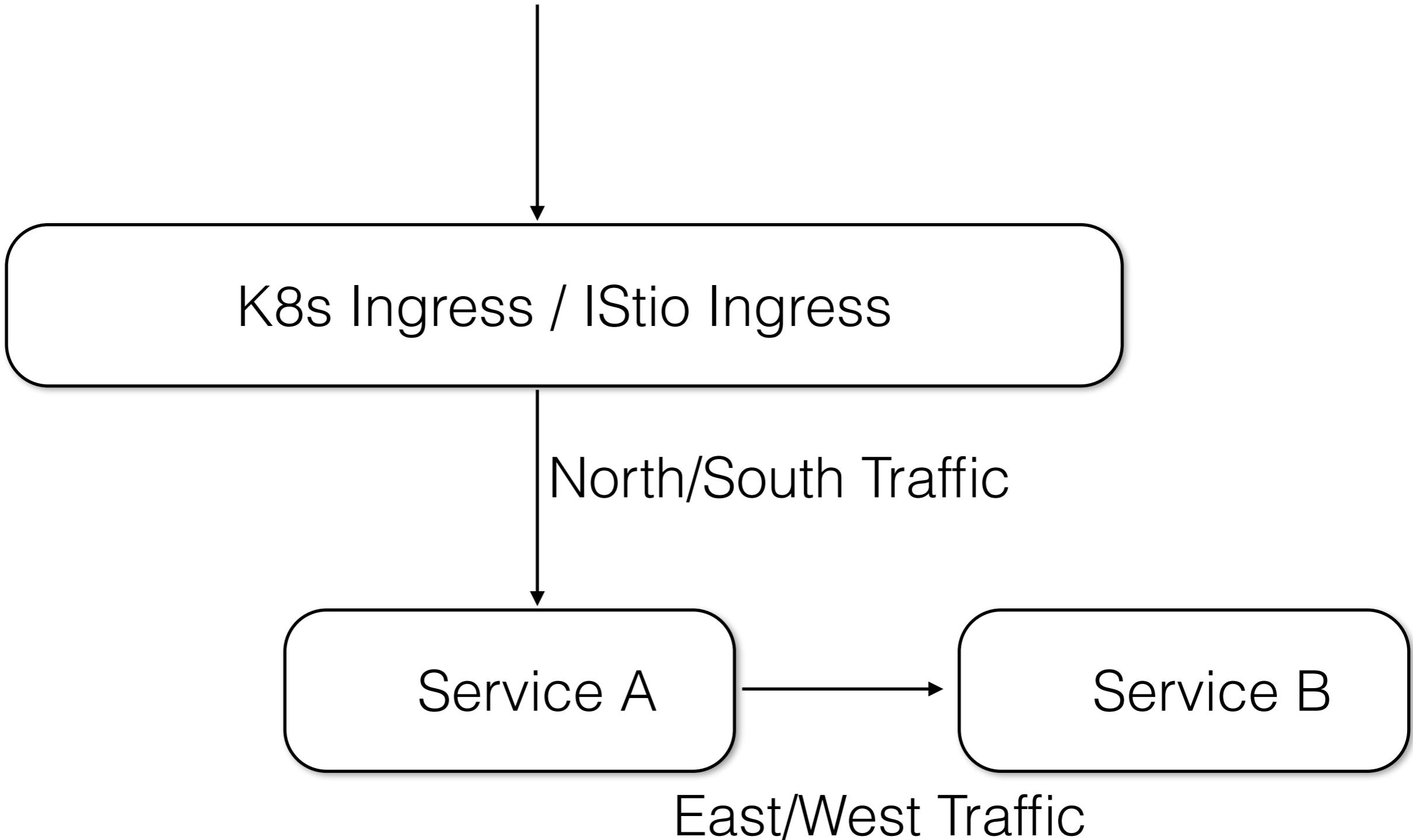
Side car

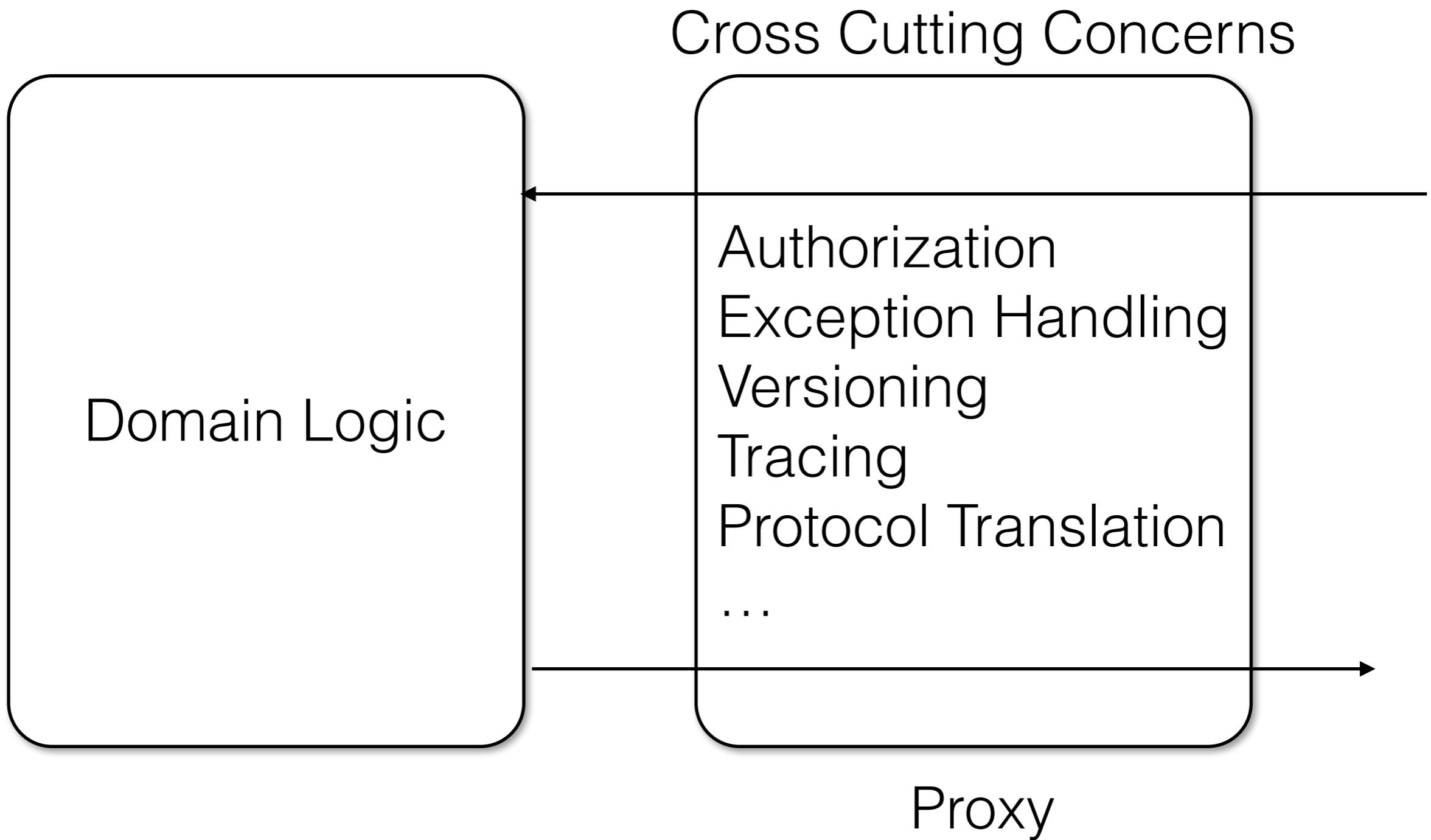


Microservice  
POD

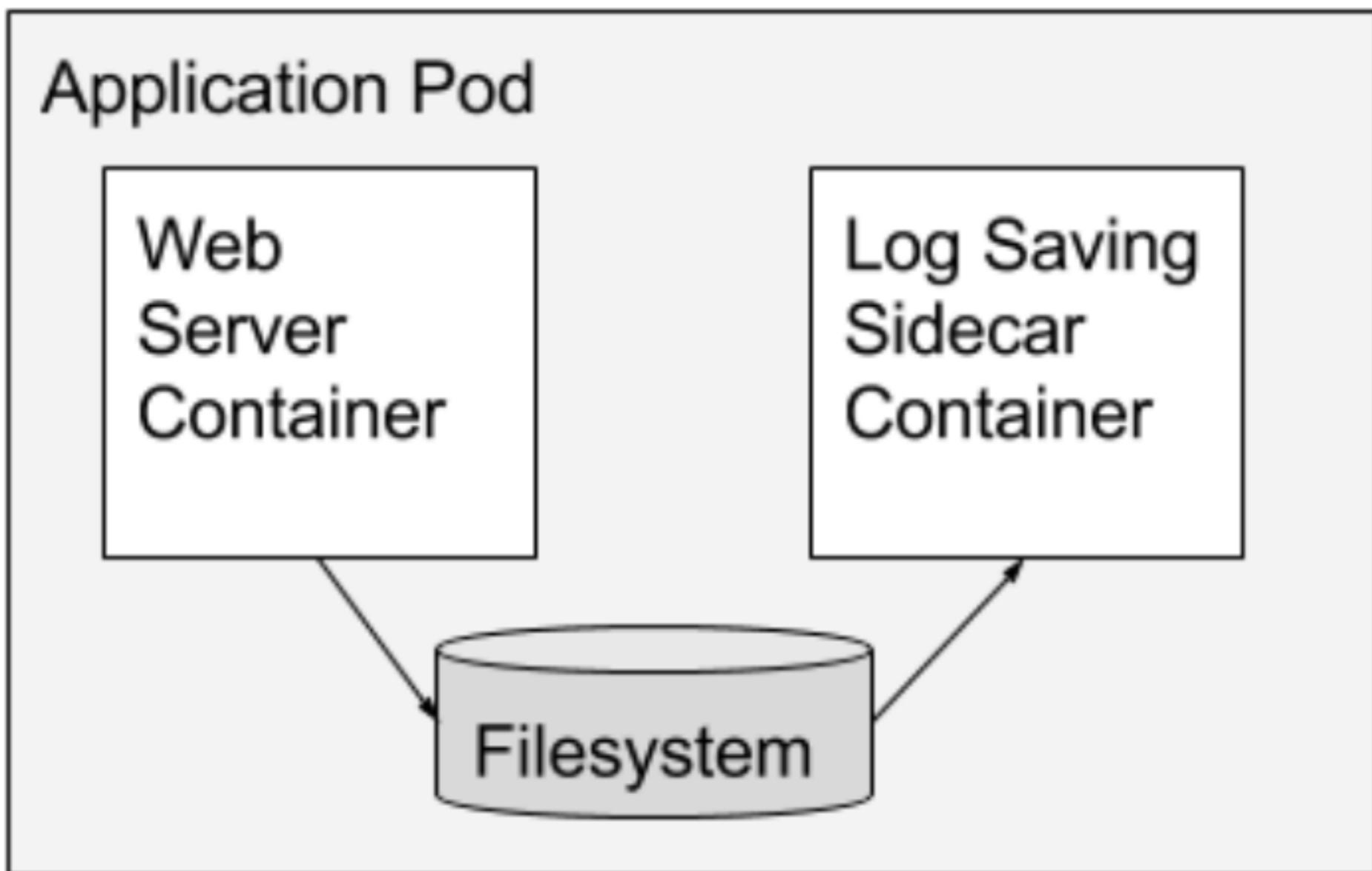
## Edge Ingres





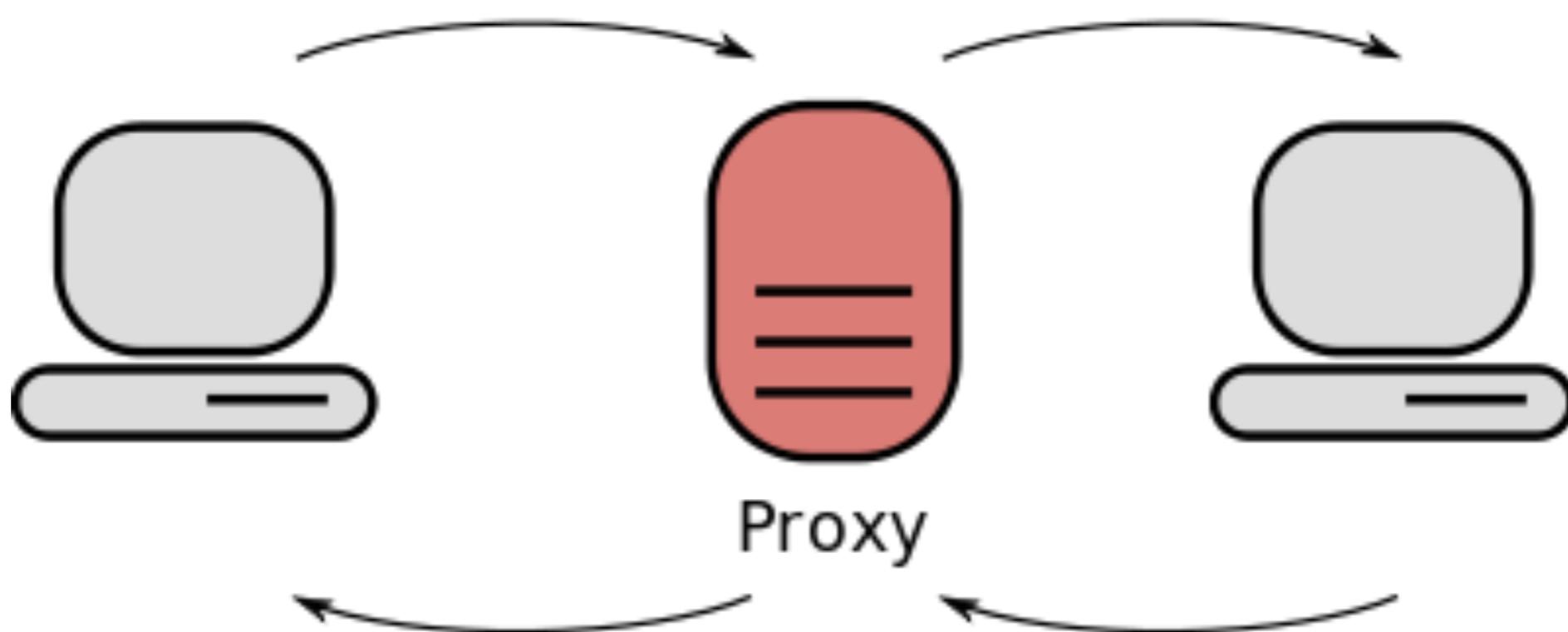


# Sidecar



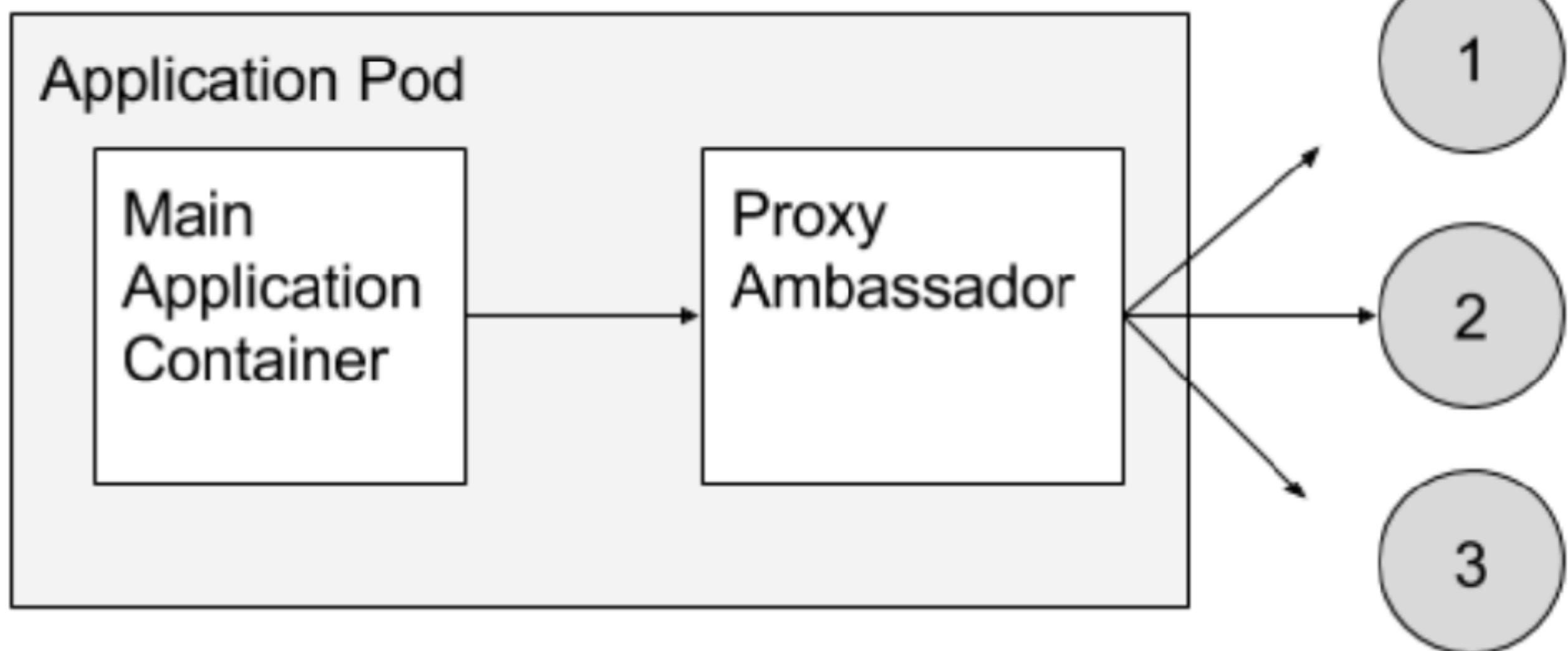
- In a sidecar pattern, the functionality of the main container is extended or enhanced by a sidecar container without strong coupling between two.
- eg. main container is a web server which is paired with a log saver sidecar container that collects the web server's logs from local disk and streams them to centralized log collector.

# Proxy pattern



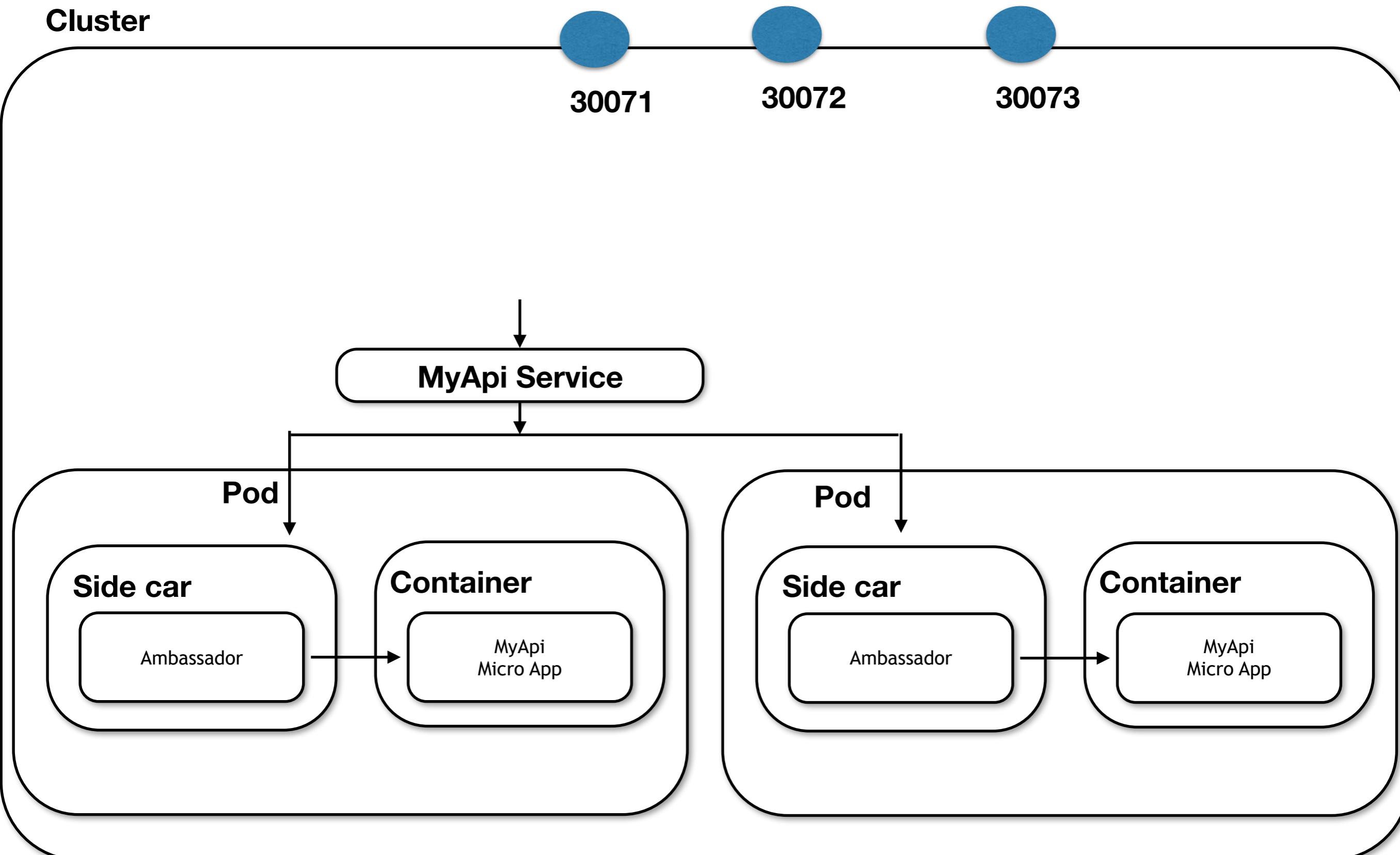
- Proxy pretends being an object when in fact it's not.  
There are several types of proxies:
- Remote proxy ("Ambassador") - instead of communicating between server's and client's classes we create a proxy on client side and server side, and only the proxies communicate
- Virtual proxy - creates expensive object on first demand
- Protection proxy - to control access
- Smart proxy - enrich an object

# Ambassador pattern

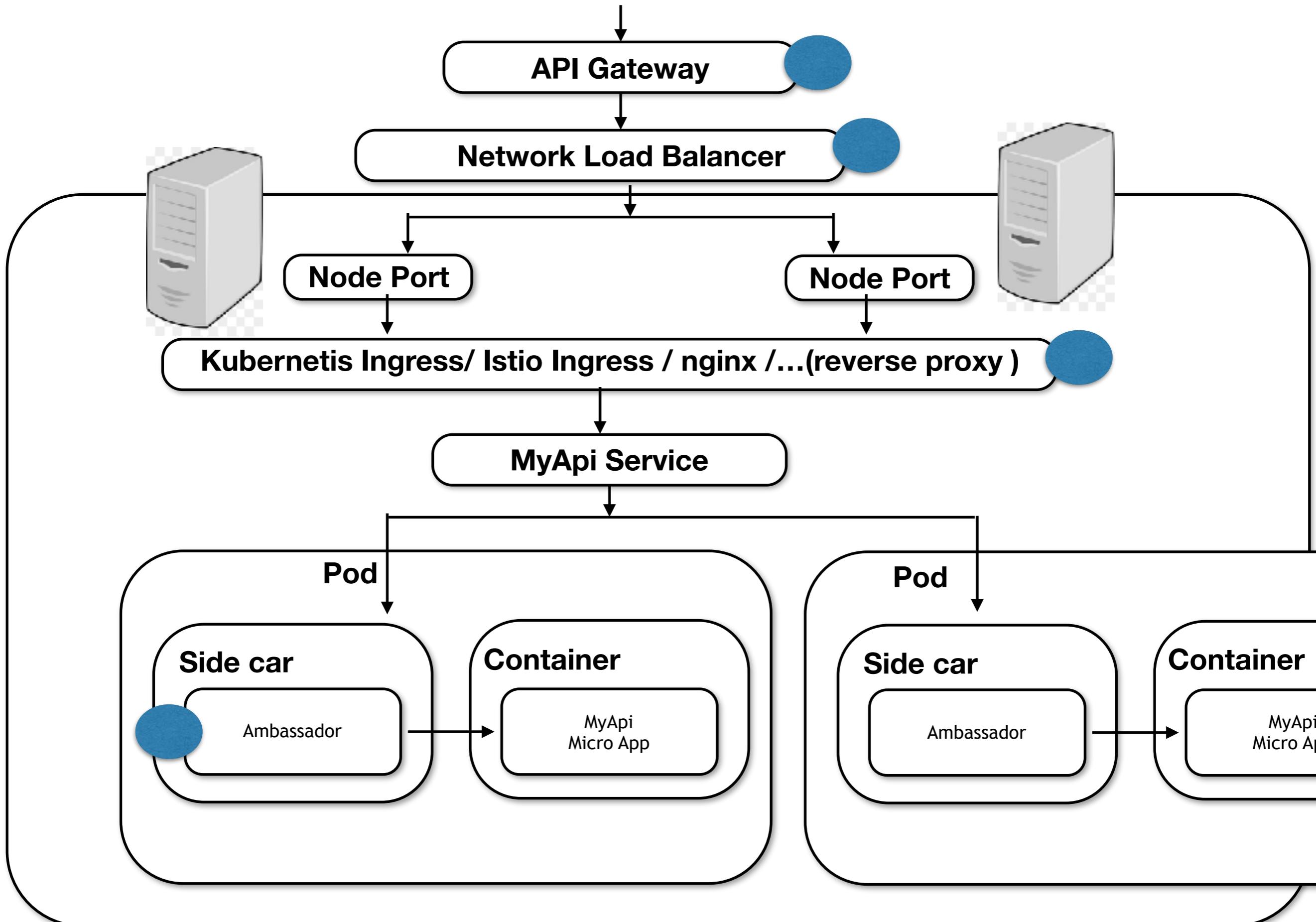


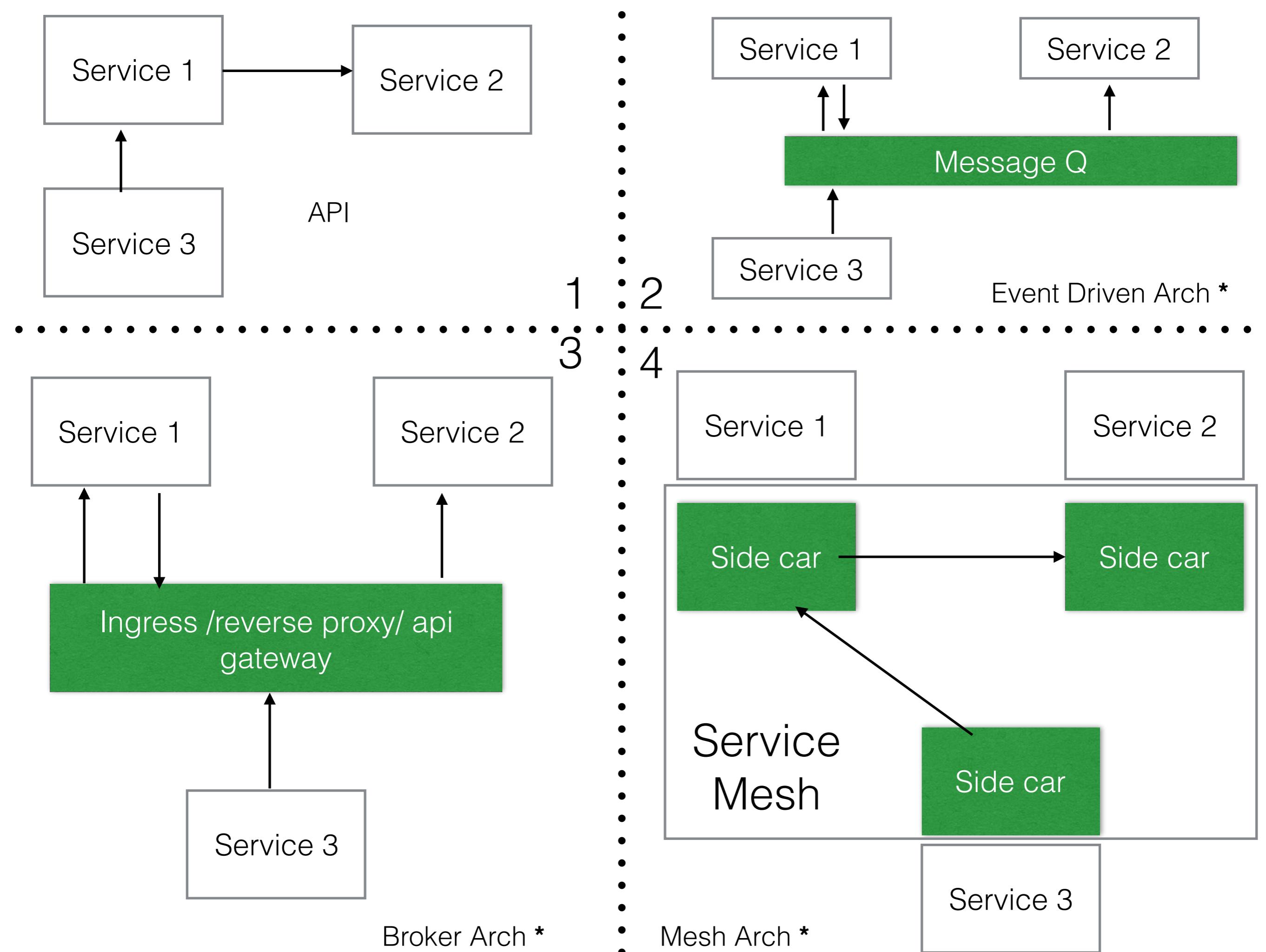
- an Ambassador container act as a proxy between two different types of main containers. Typical use case involves proxy communication related to load balancing and/or sharding to hide the complexity from the application.

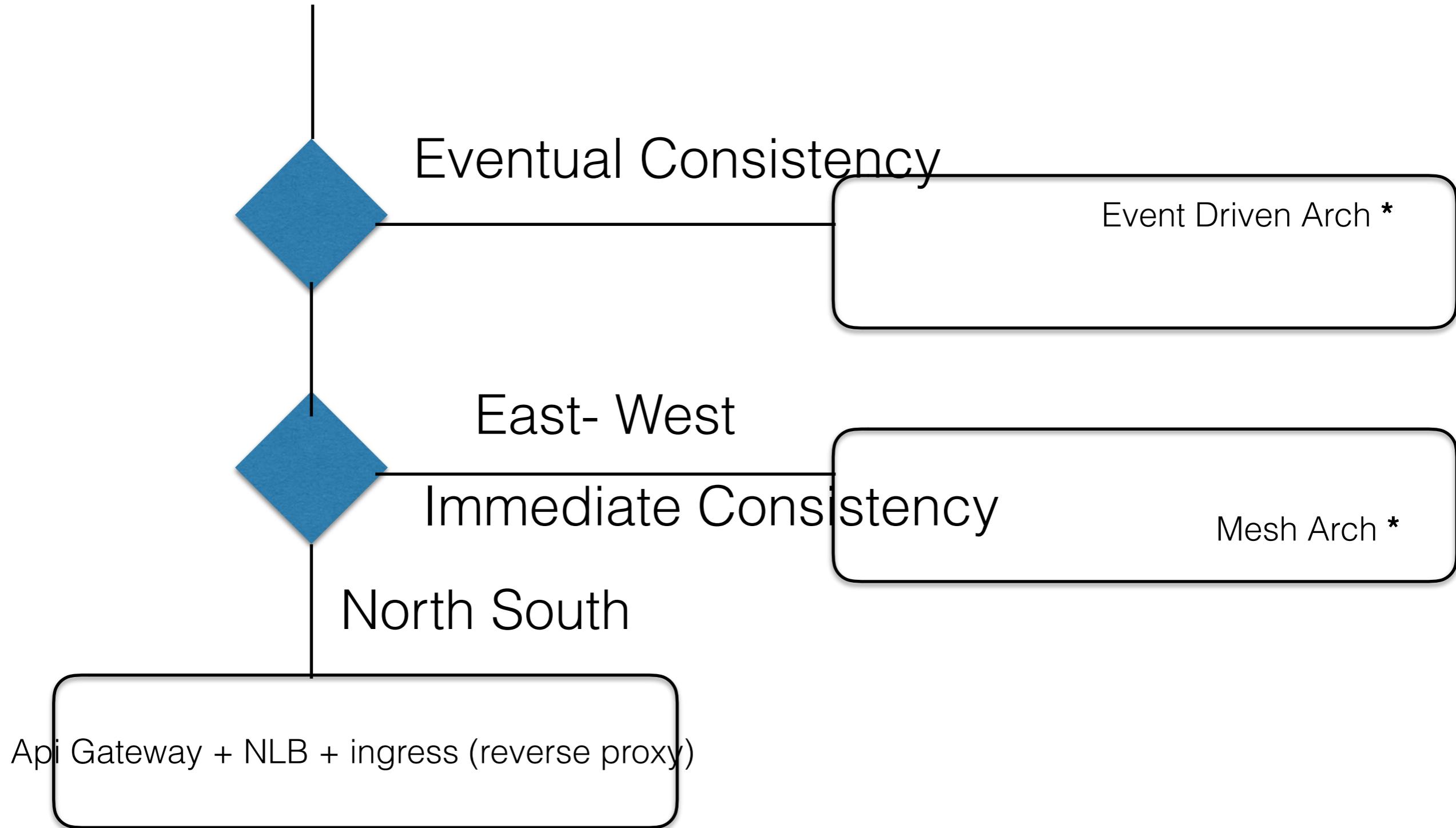
# Deployment Diagram

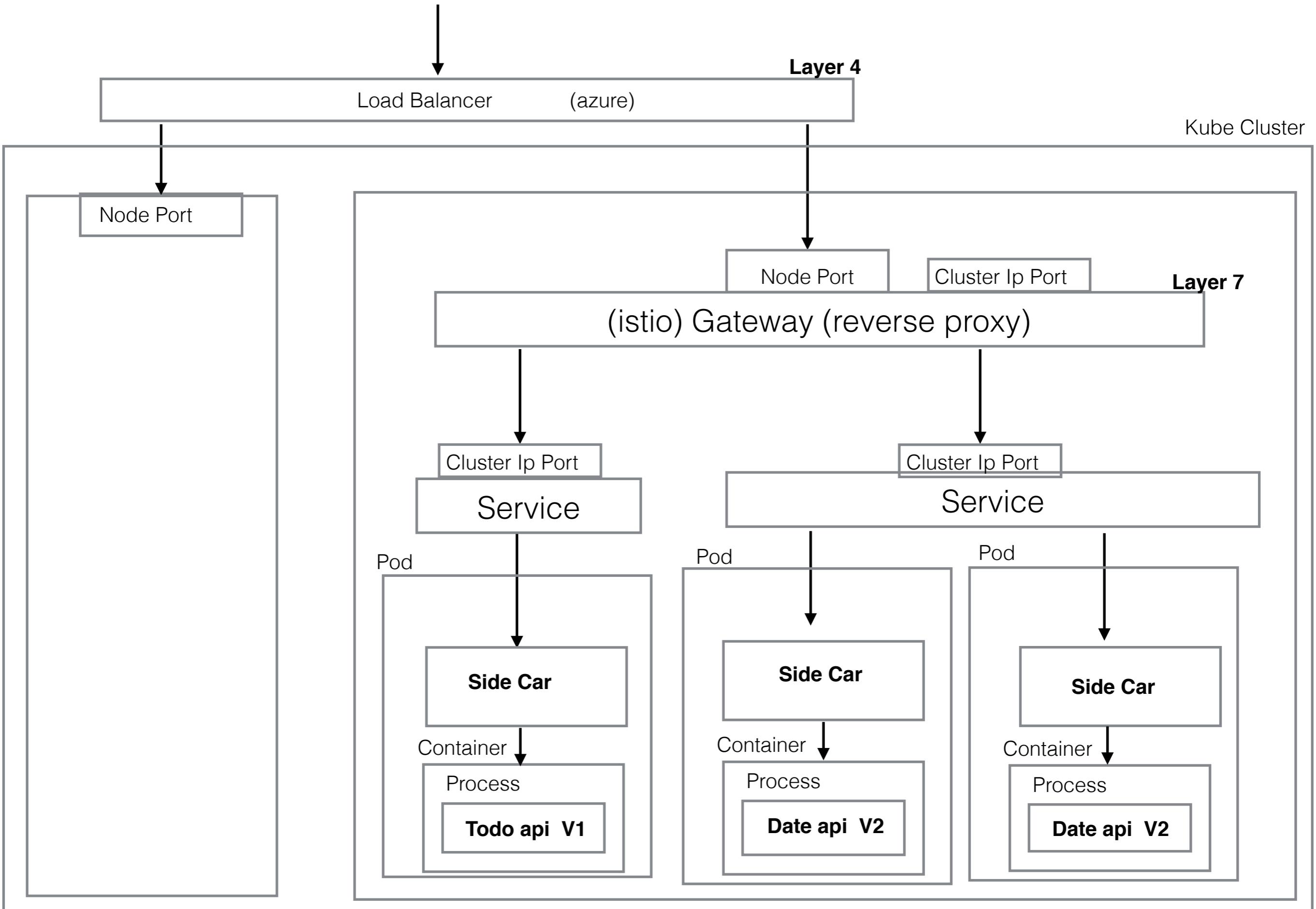


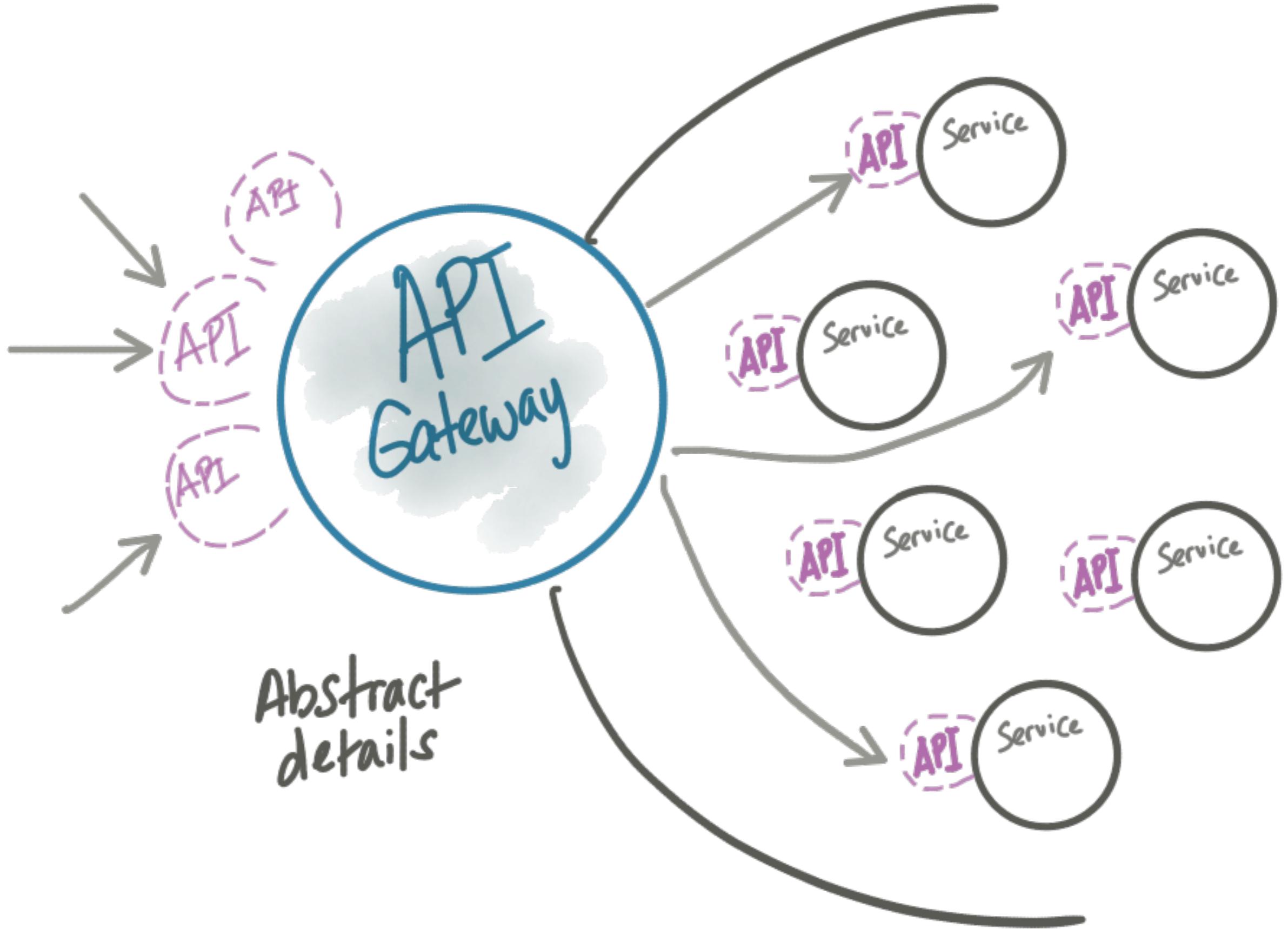
# Deployment Diagram

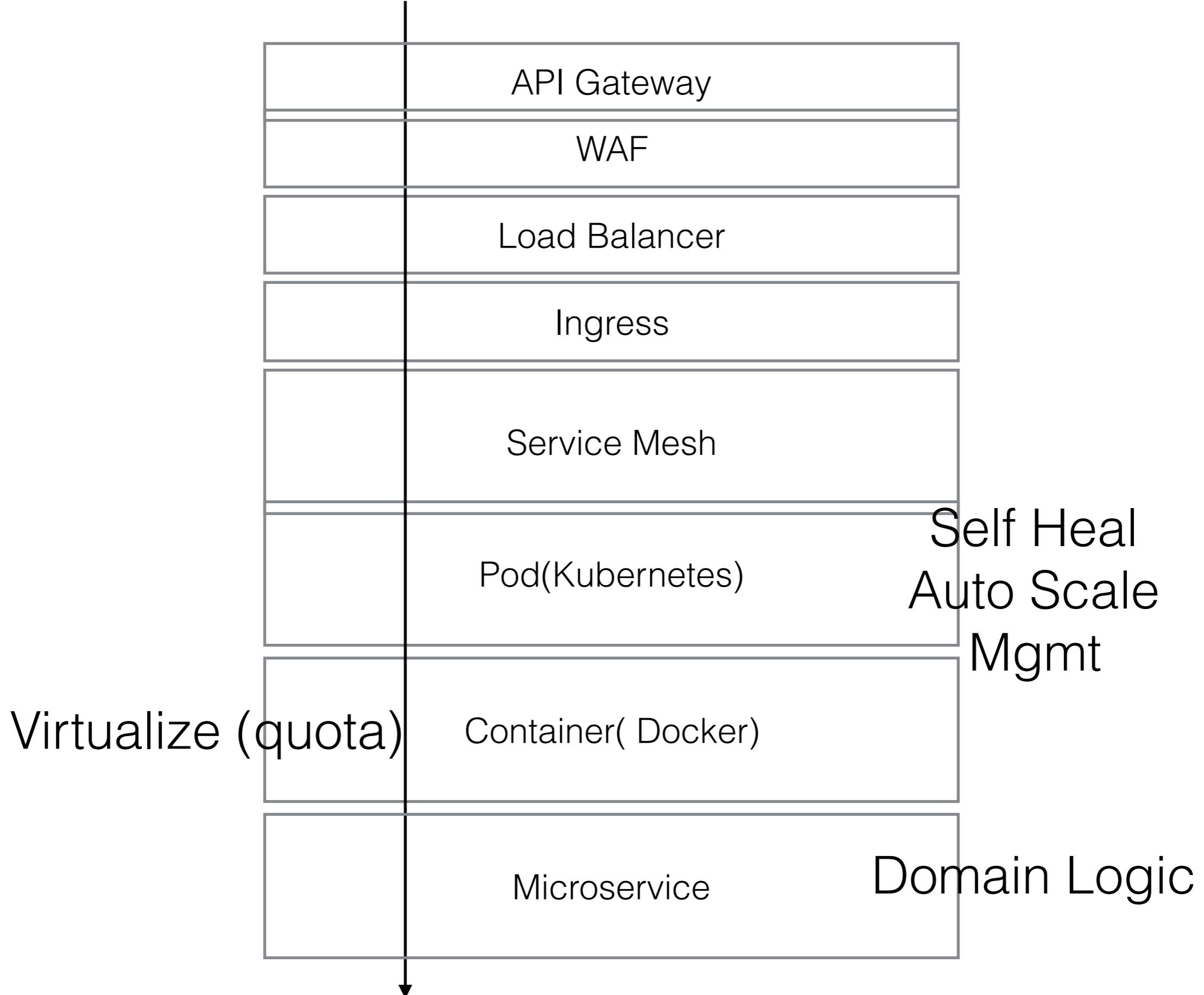




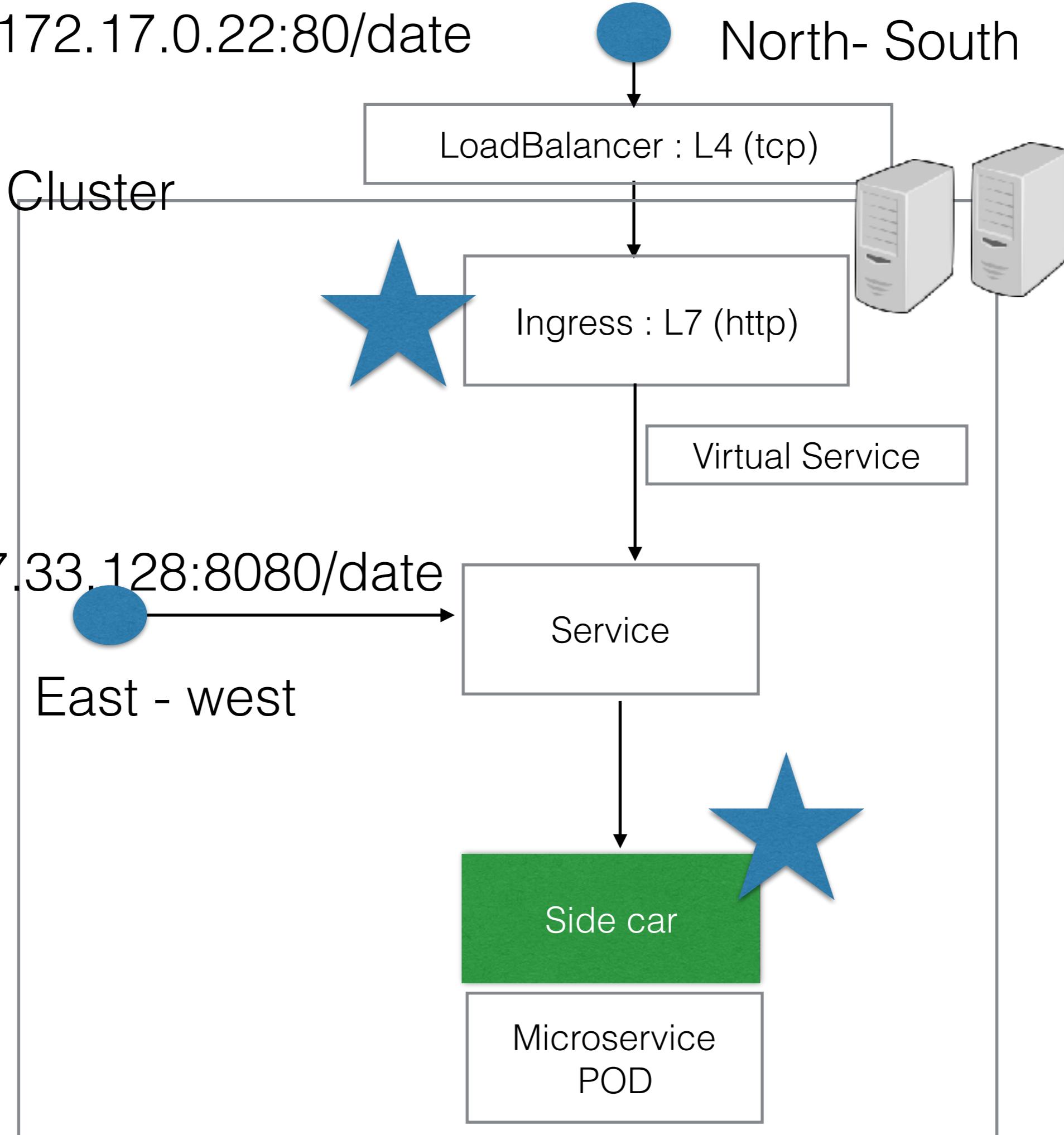




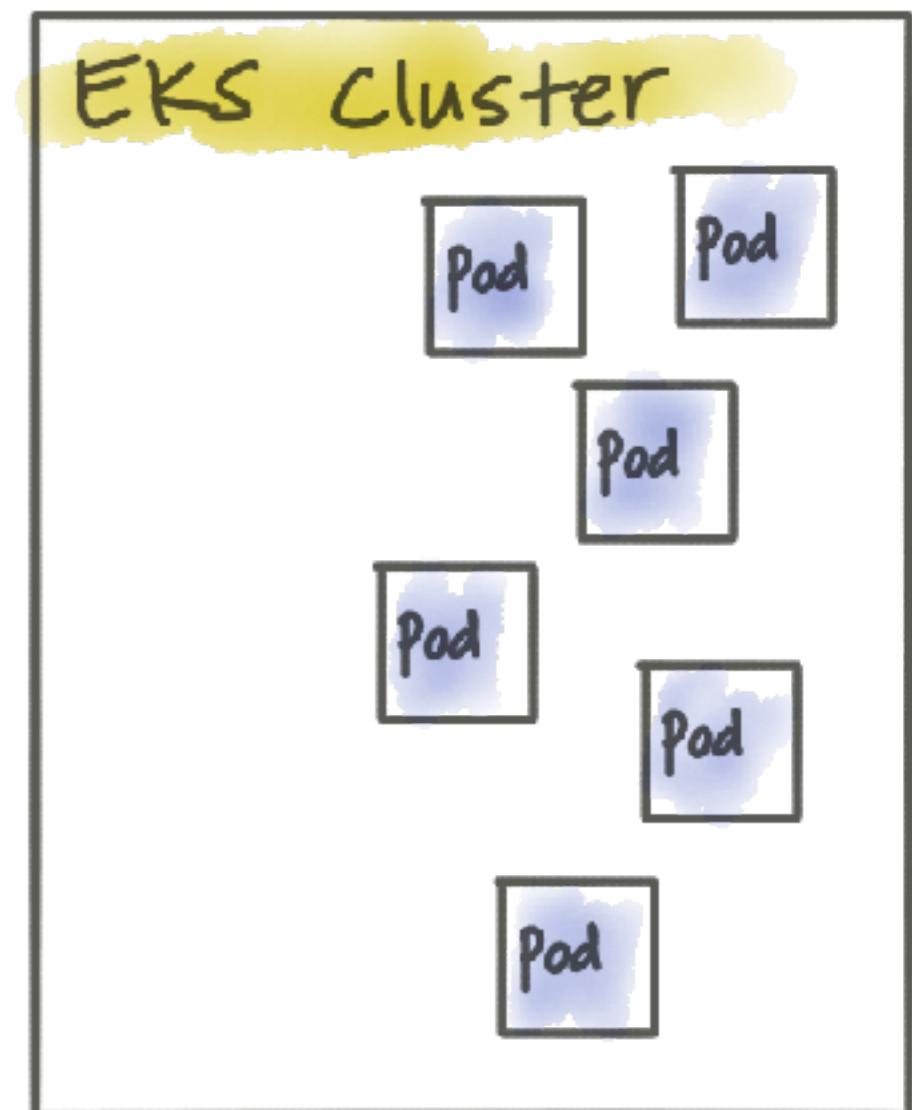
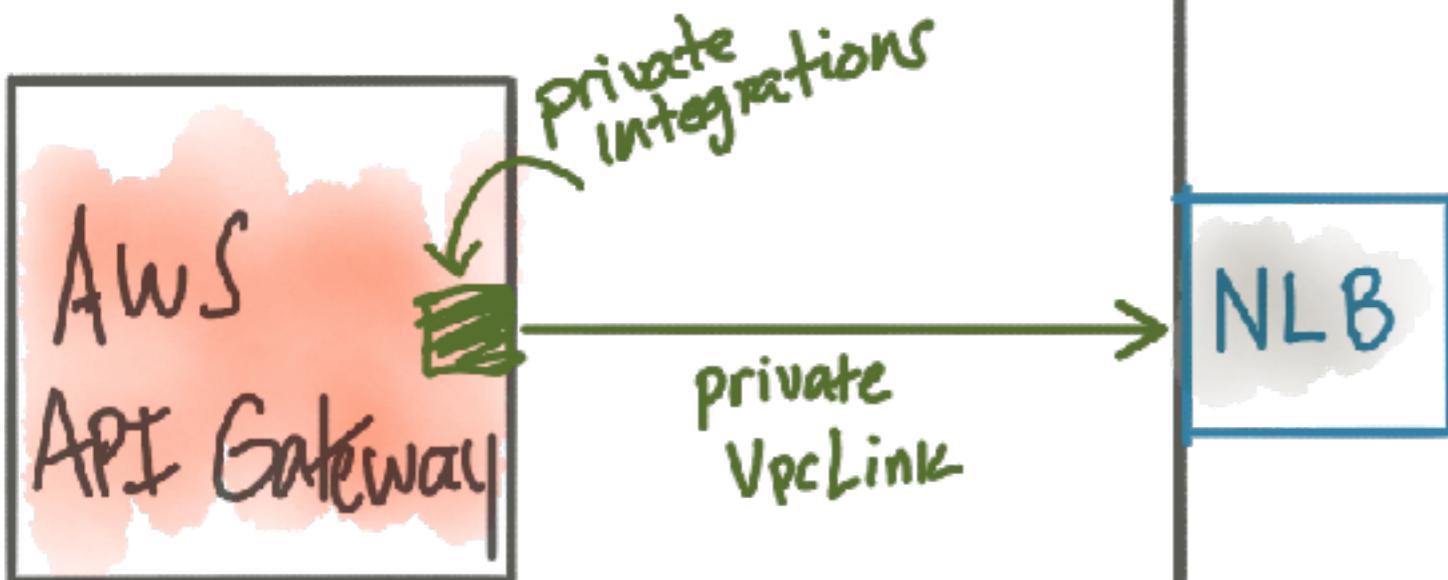




http://172.17.0.22:80/date



# VPC / private subnets



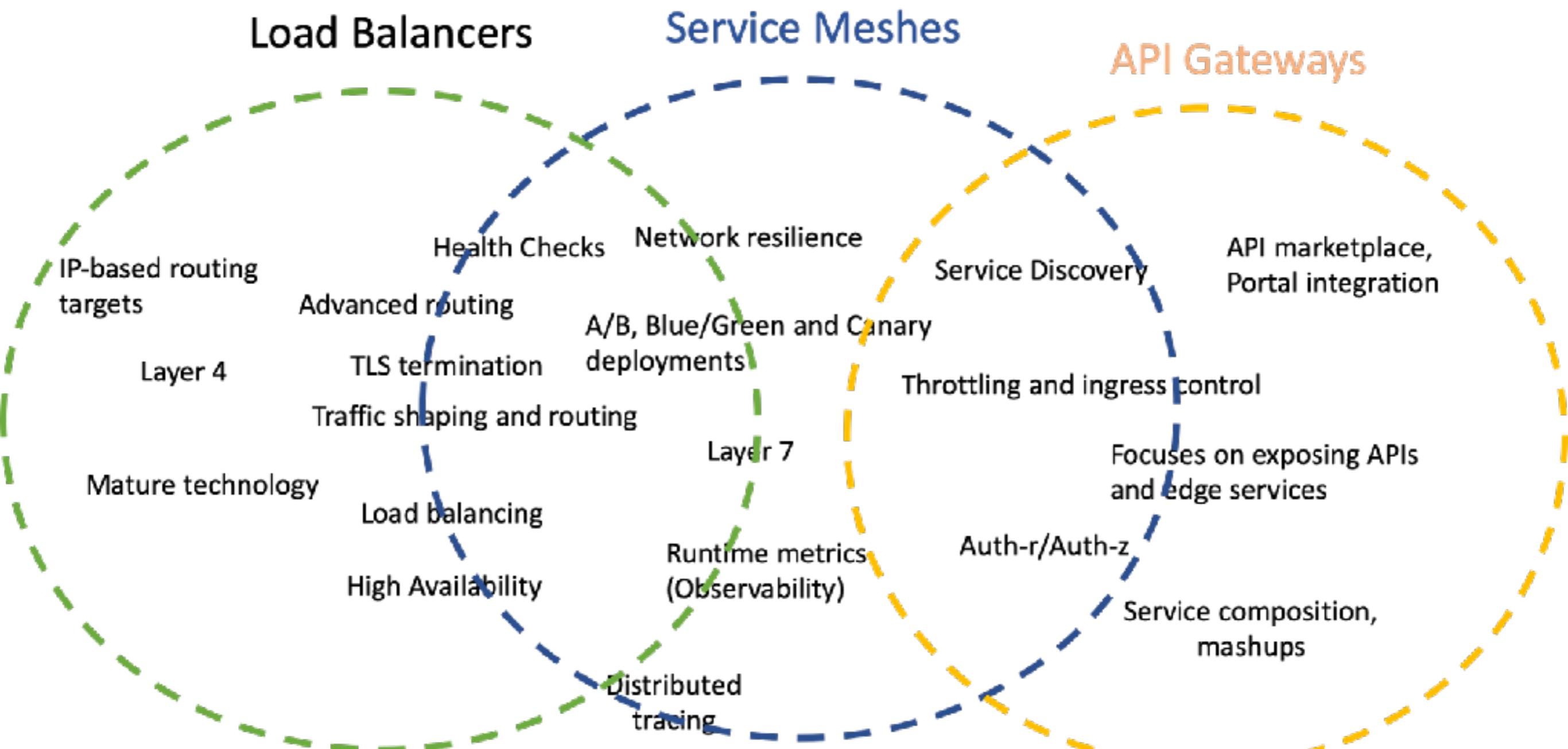
AWS API Gateway runs in its own VPC and is completely managed so you cannot see any details about its infrastructure.

**API Gateway :** Abstraction,  
decoupling,  
edge routing / security

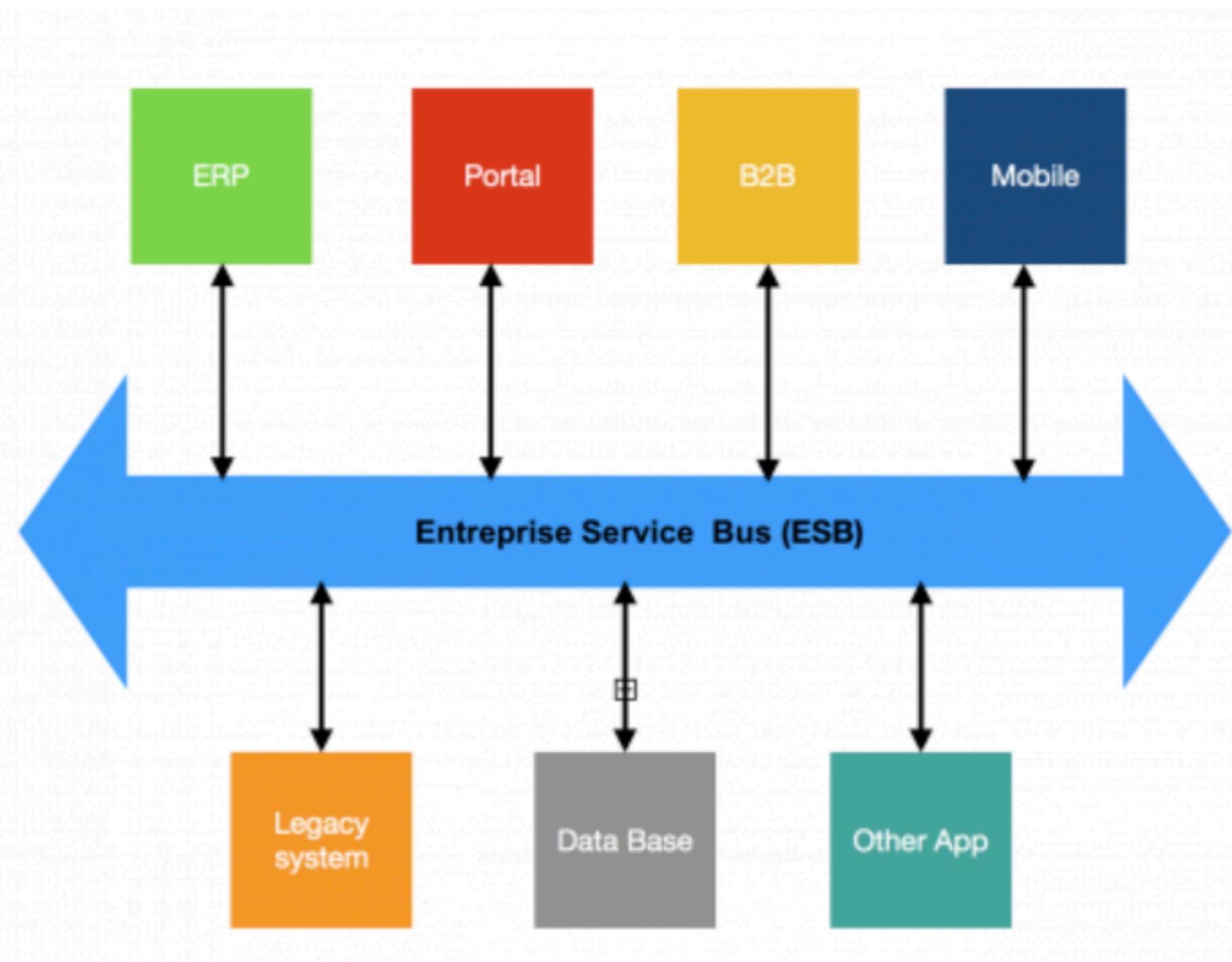
**Service Mesh :** endpoints, hosts, ports,  
metric collection,  
traffic routing, security

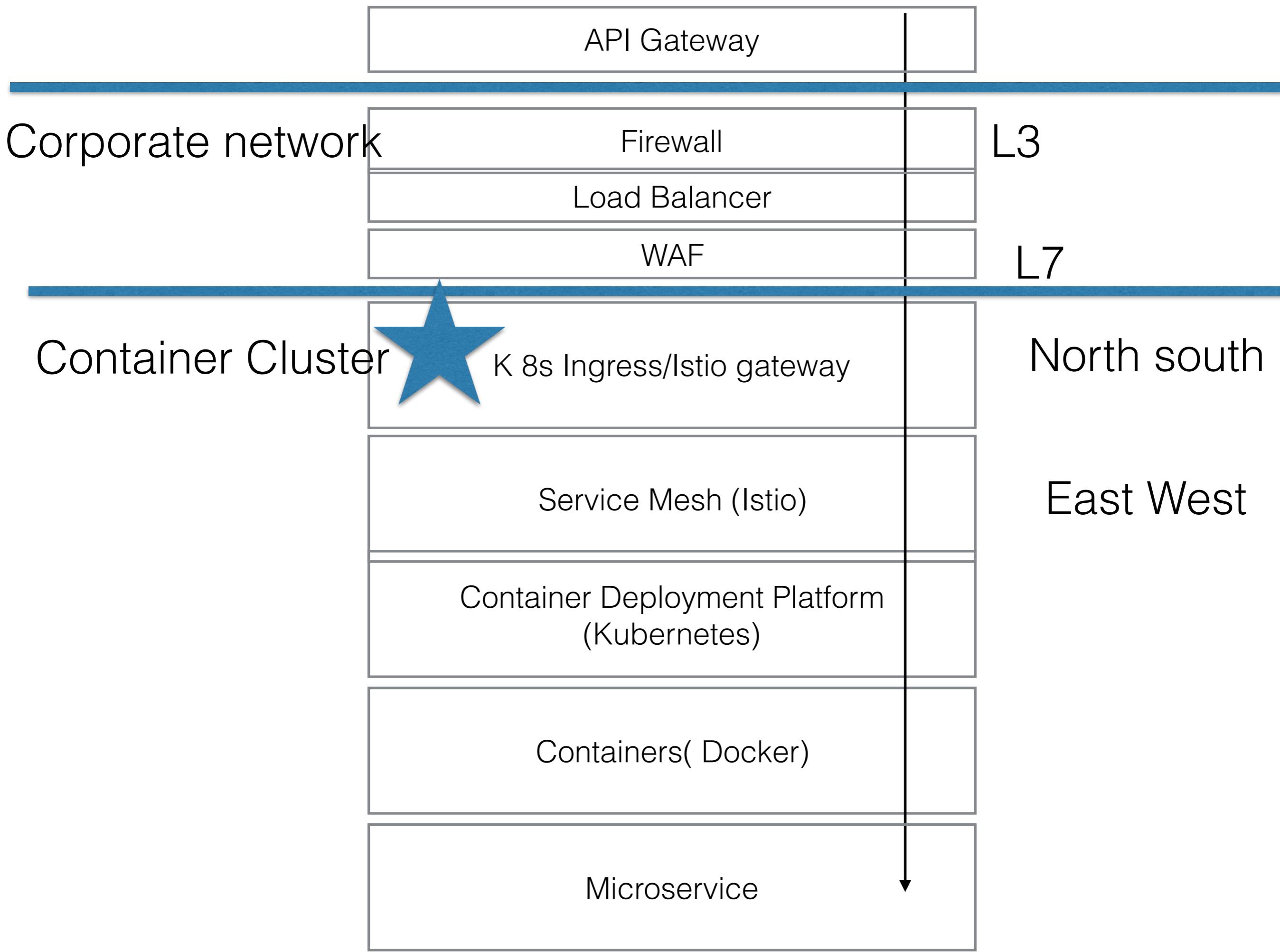
**Deployment Platform :** Cluster nodes,  
container schedule,  
resource management

# Reverse Proxy/



# API Gateway vs ESB





http://172.17.0.22:80/date



North- South



API Gateway

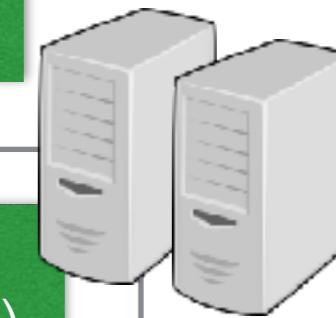


LoadBalancer : L4 (tcp)

Cluster



Ingress or ISTIO Gateway : L7 (http)



Virtual Service

Service

East - west

http://10.97.33.128:8080/date



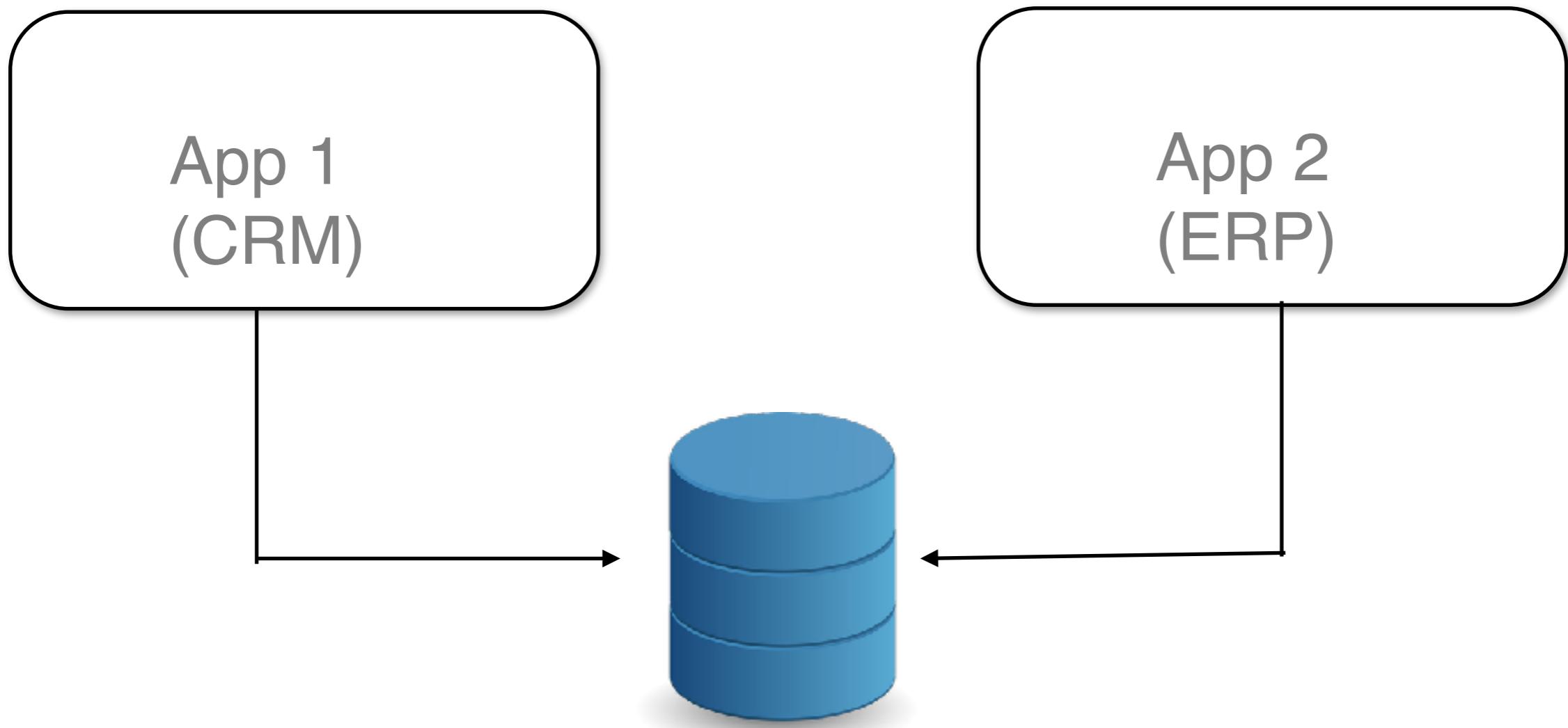
Side car



Microservice  
POD

Log Agg  
Tracability  
Monitoring

# 1. Database



## 2. Source Control

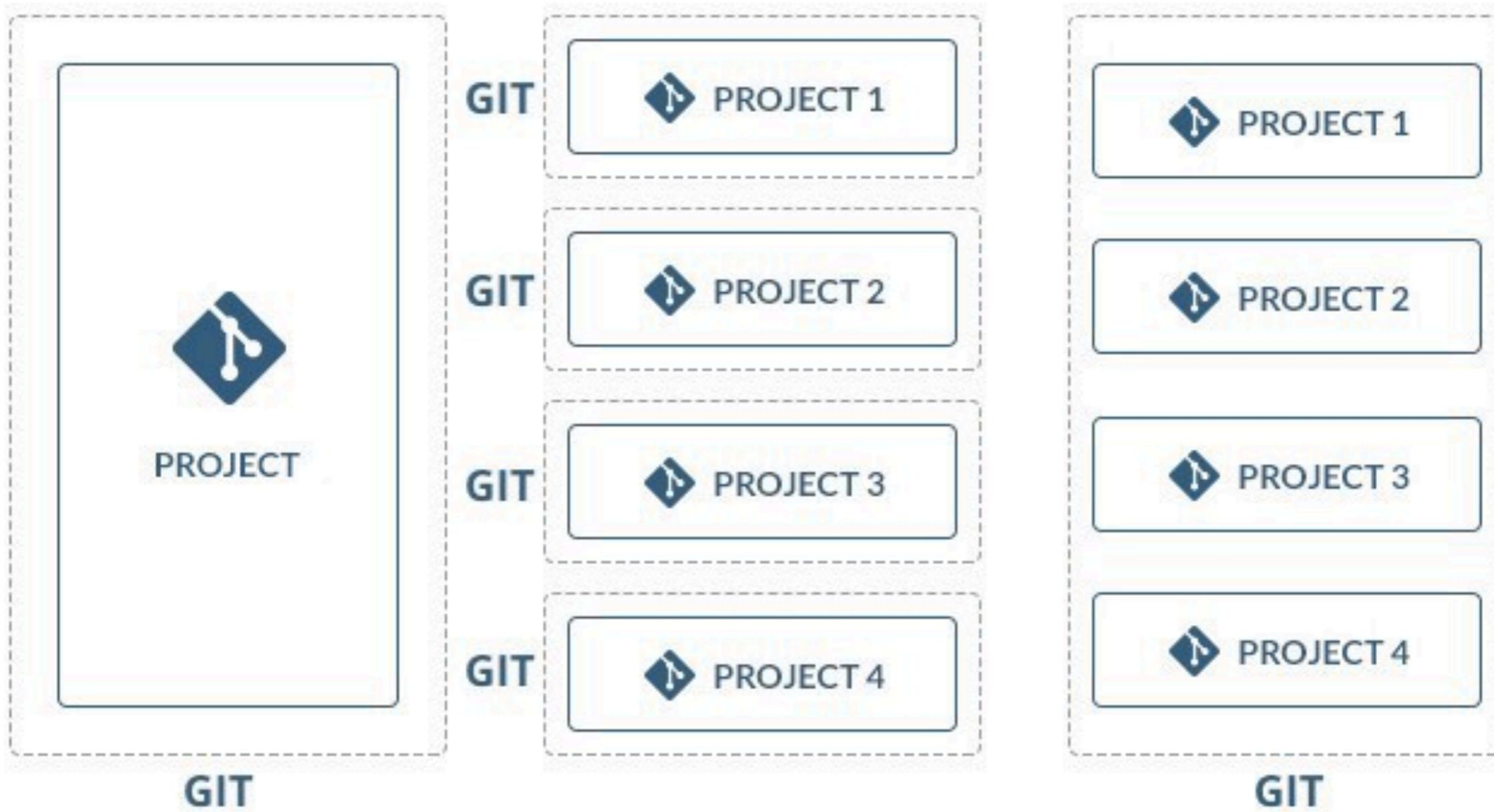
MONOLITH



MULTI-REPO



MONO-REPO

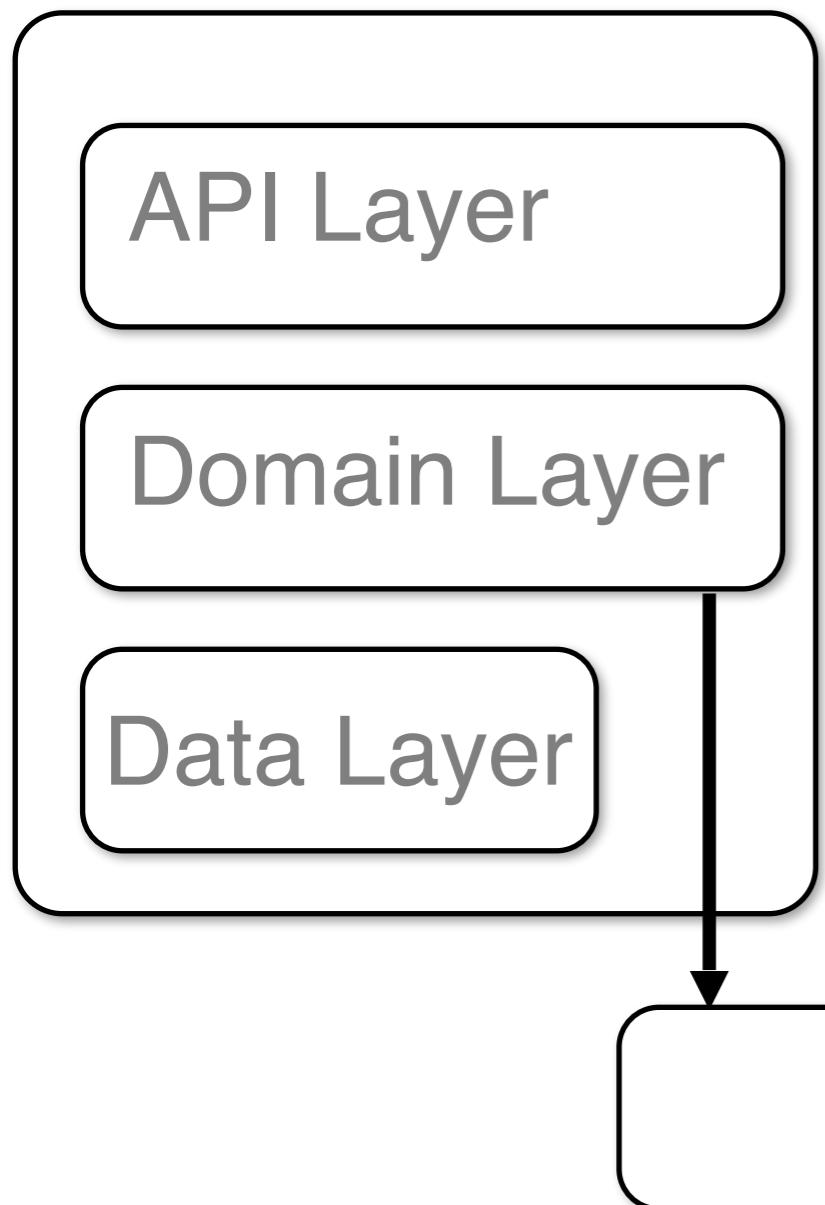


## 2. Source Control

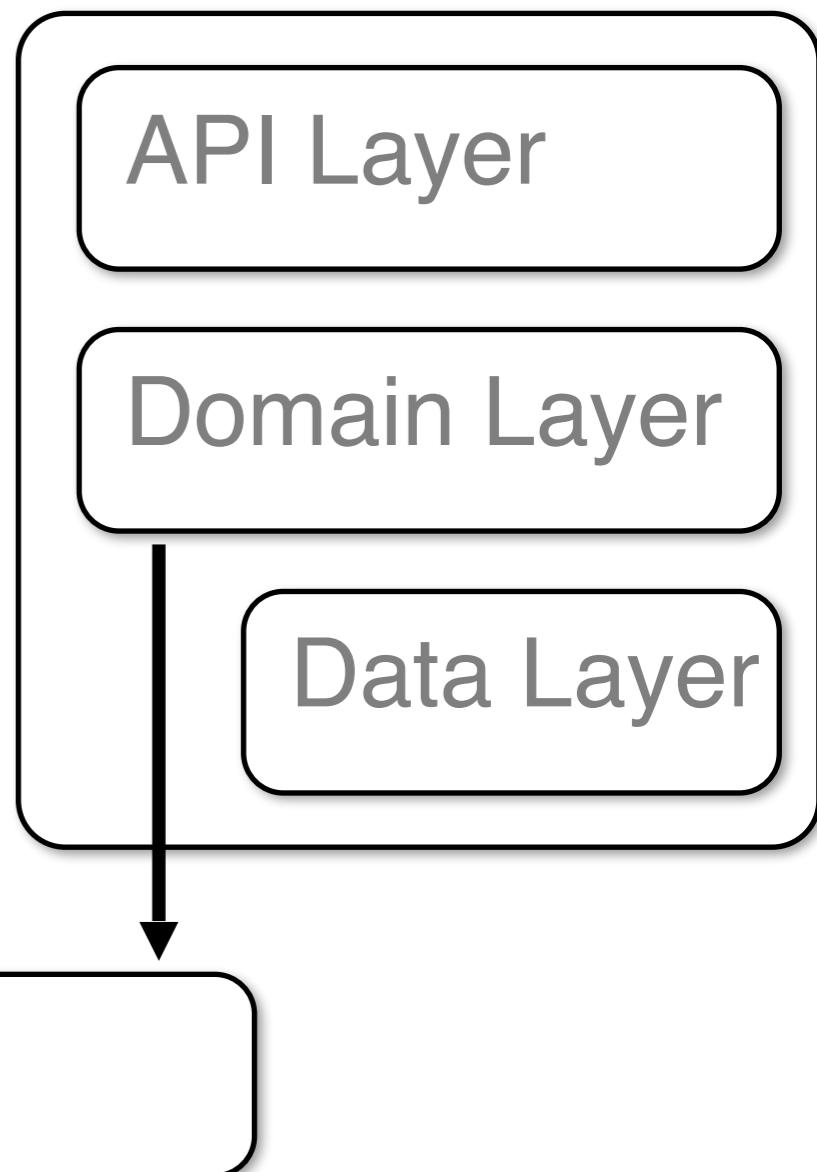
|                   | <b>Monorepo</b>                                                                                                                                                                                   | <b>Poly repos</b>                                                                                                                                               |
|-------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Advantages</b> | Code sharing<br>Easier to standardize code and tooling<br>Easier to refactor code<br>Discoverability - single view of the code                                                                    | Clear ownership per team<br>Potentially fewer merge conflicts<br>Helps to enforce decoupling of microservices                                                   |
| <b>Challenges</b> | Changes to shared code can affect multiple microservices<br>Greater potential for merge conflicts<br>Tooling must scale to a large code base<br>Access control<br>More complex deployment process | Harder to share code<br>Harder to enforce coding standards<br>Dependency management<br>Diffuse code base, poor discoverability<br>Lack of shared infrastructure |

### 3. Common Logic

Purchase  
Module



Sales  
Module

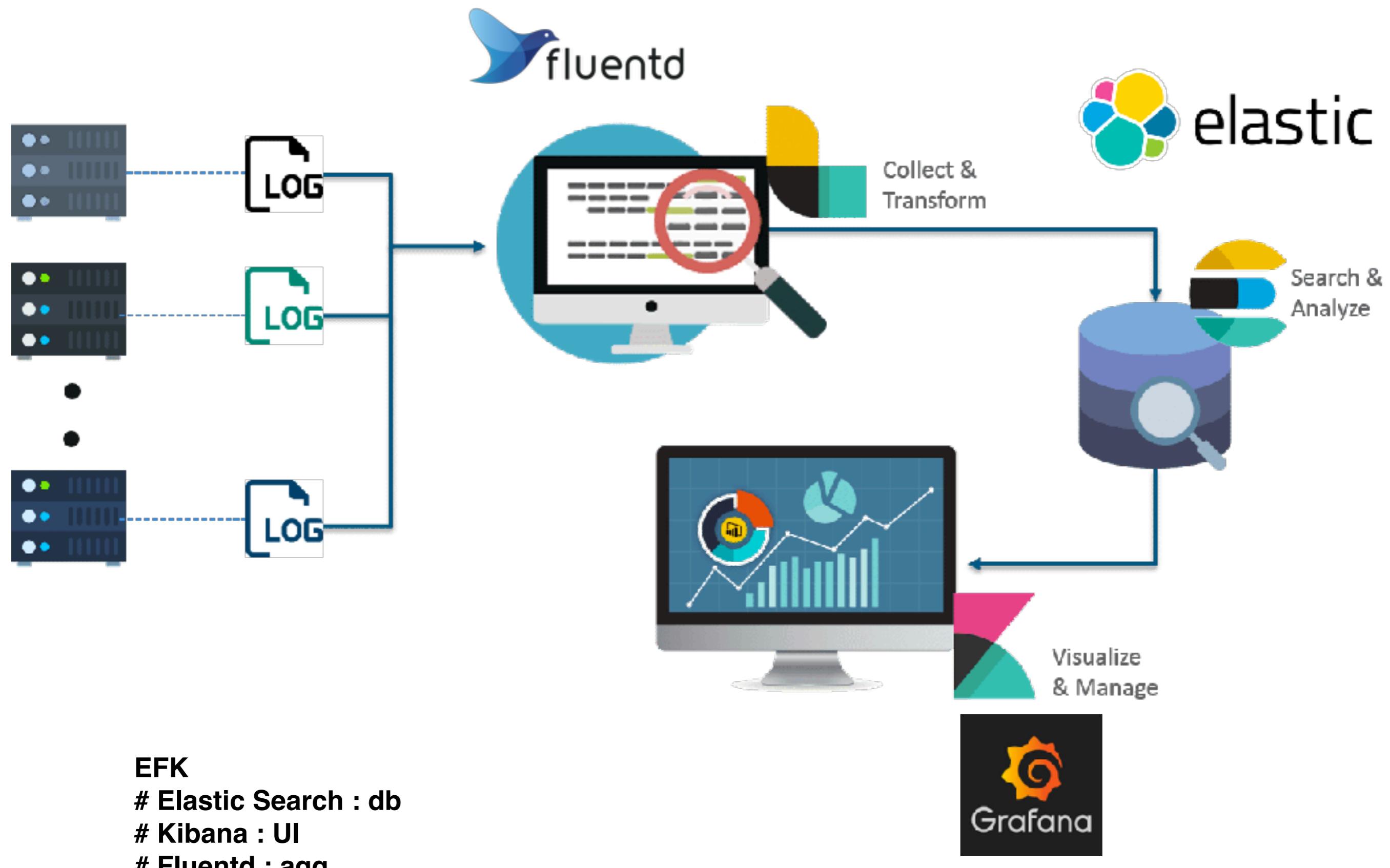


Pure fabrication # stateless, domain independent

# Monitoring

- Kubernetes
  - Kubernetes. Dashboard (metrics)
  - Weavescope (dependancy)
- ISTIO
  - Kiali (dependancy)
  - Grafana (metrics)
- End to End Traceability
  - Jaeger
- Logs
  - EFK

# Log Aggregation



# Health Endpoint Monitoring

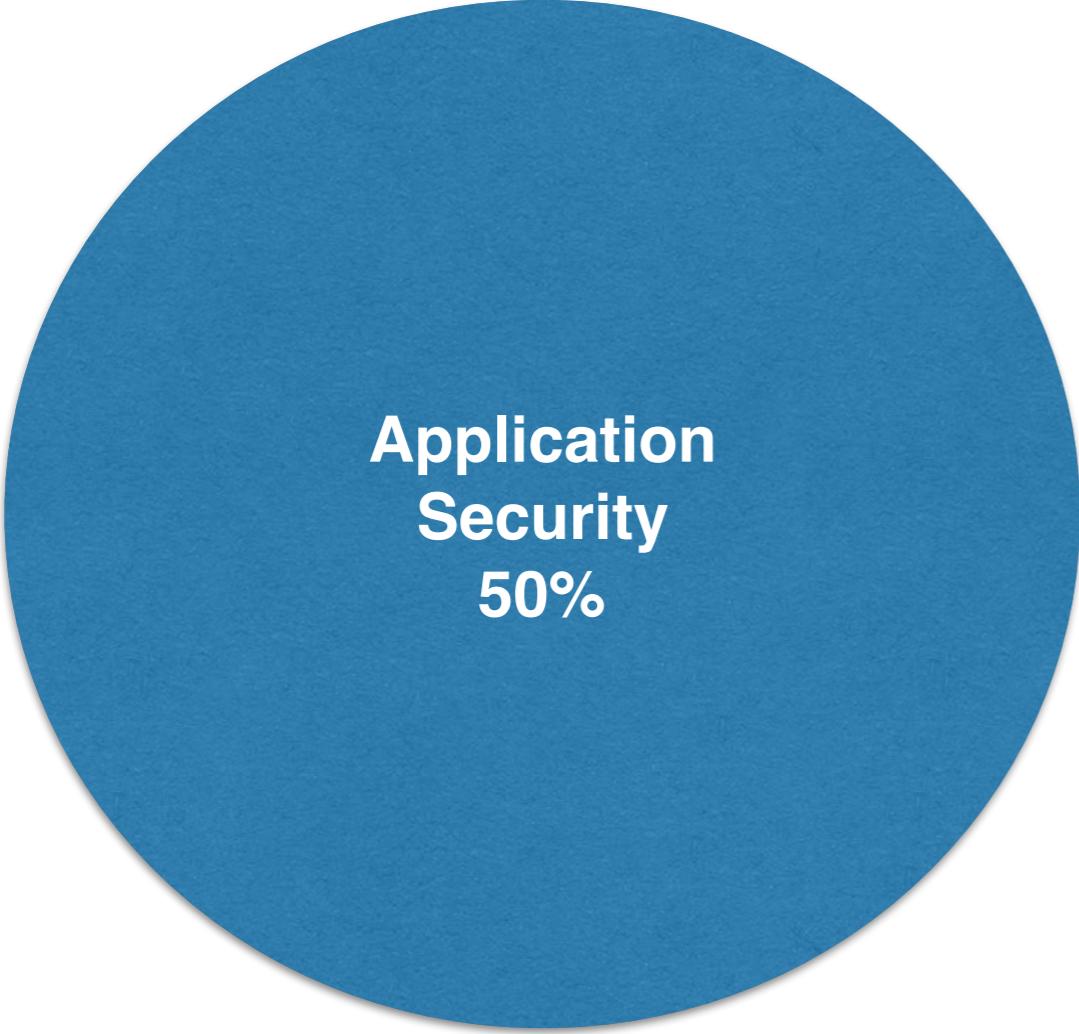


Storage  
Compute  
Messaging

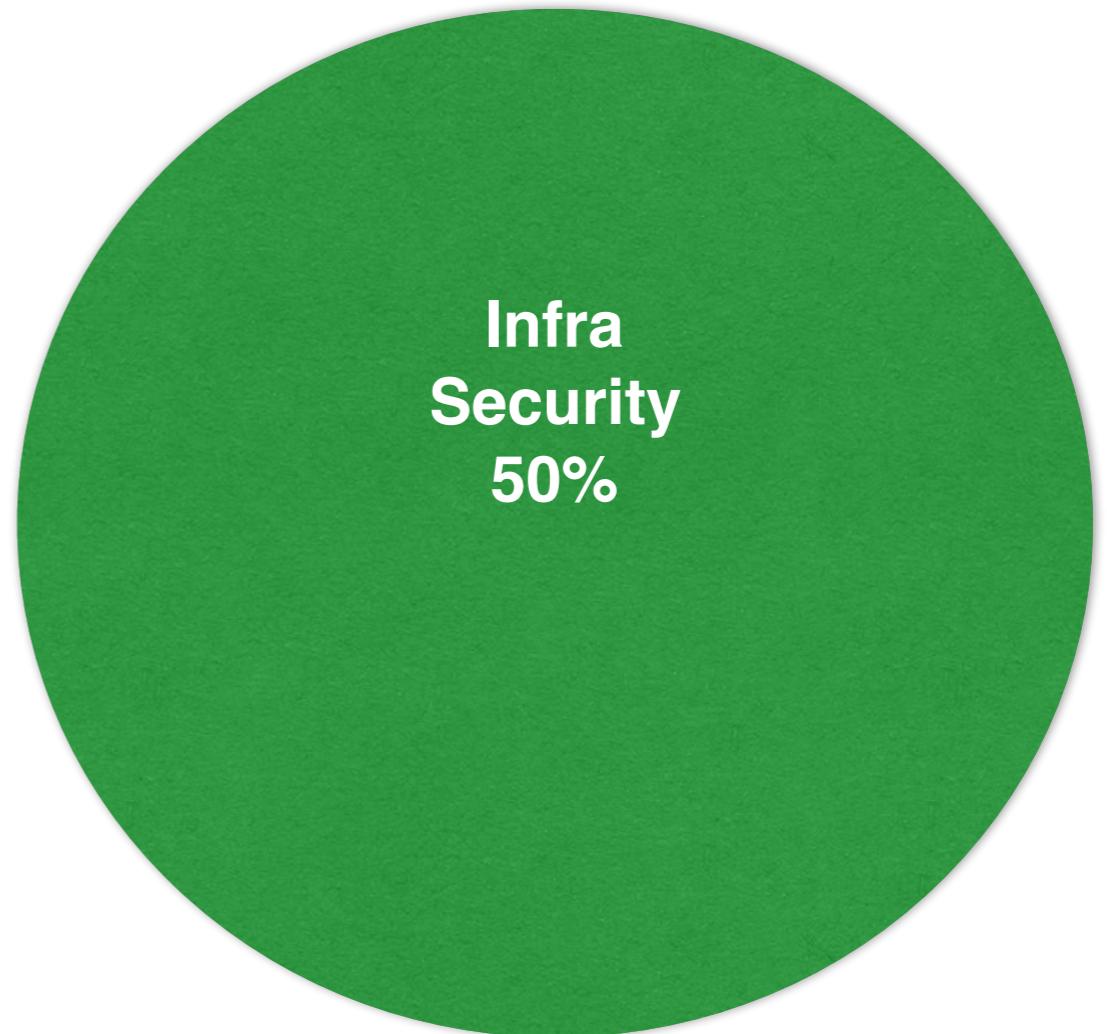
# Day 4

# Sharing





**Application  
Security**  
**50%**



**Infra  
Security**  
**50%**

**otp, email, device, rsa, cert, ..**

**pwd, secret, ..**

**By What you have \*\***

**face, retina, voice, finger, ...**

**By what you know \*\*\***

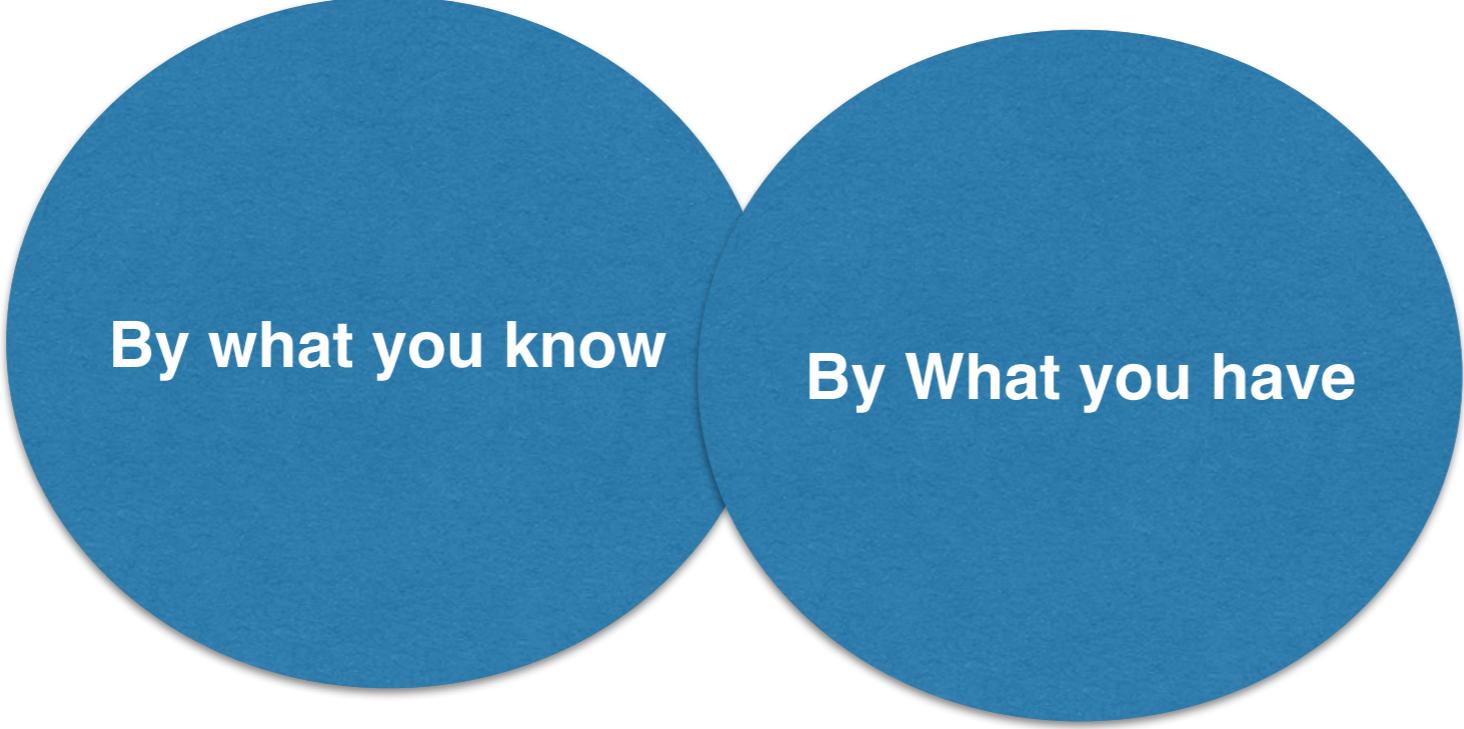
**By who you are**

**By behaviour**

**By where you are \***

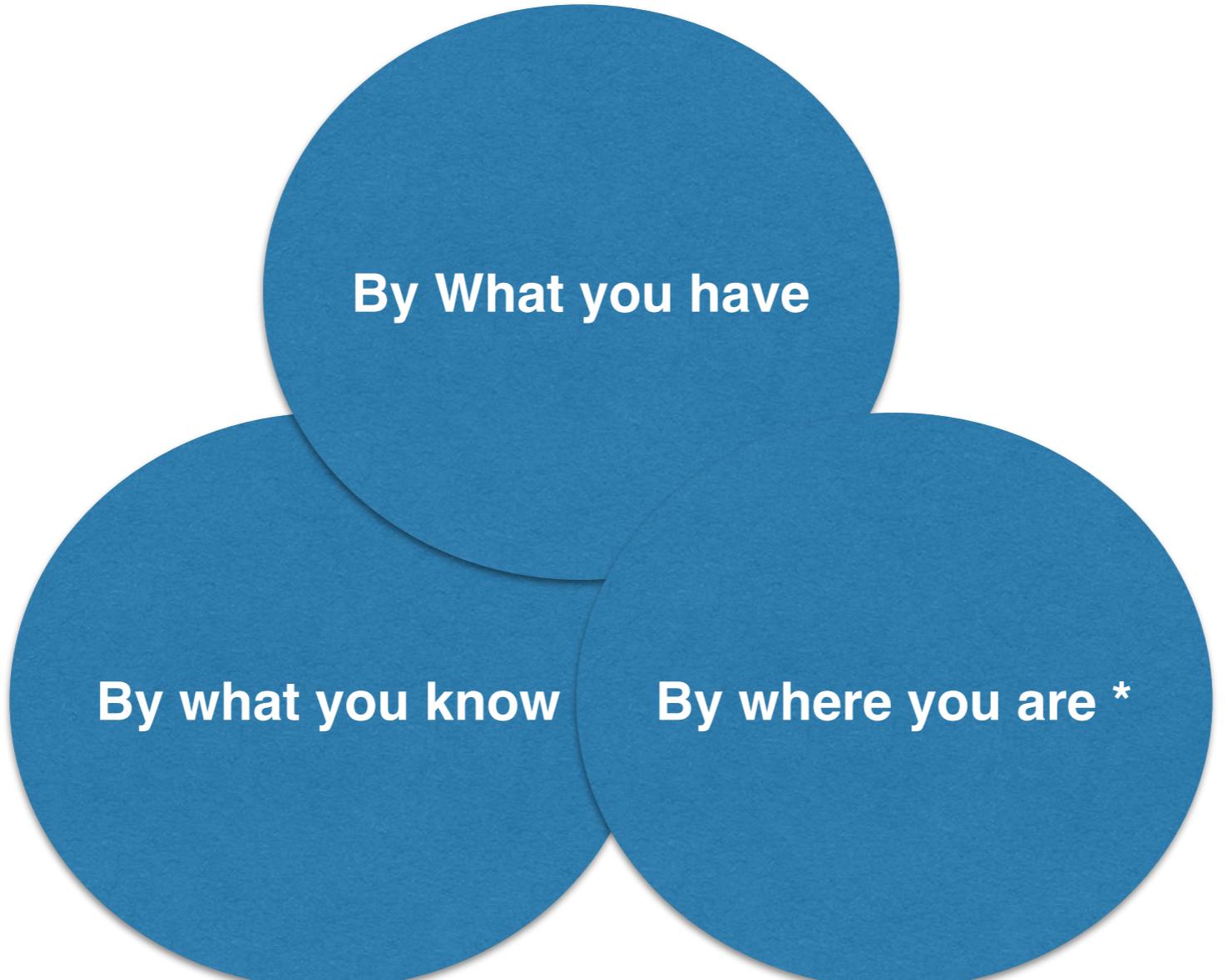
**Pattern of usage**

**location, ip address, Mac address, ...**



**By what you know**

**By What you have**



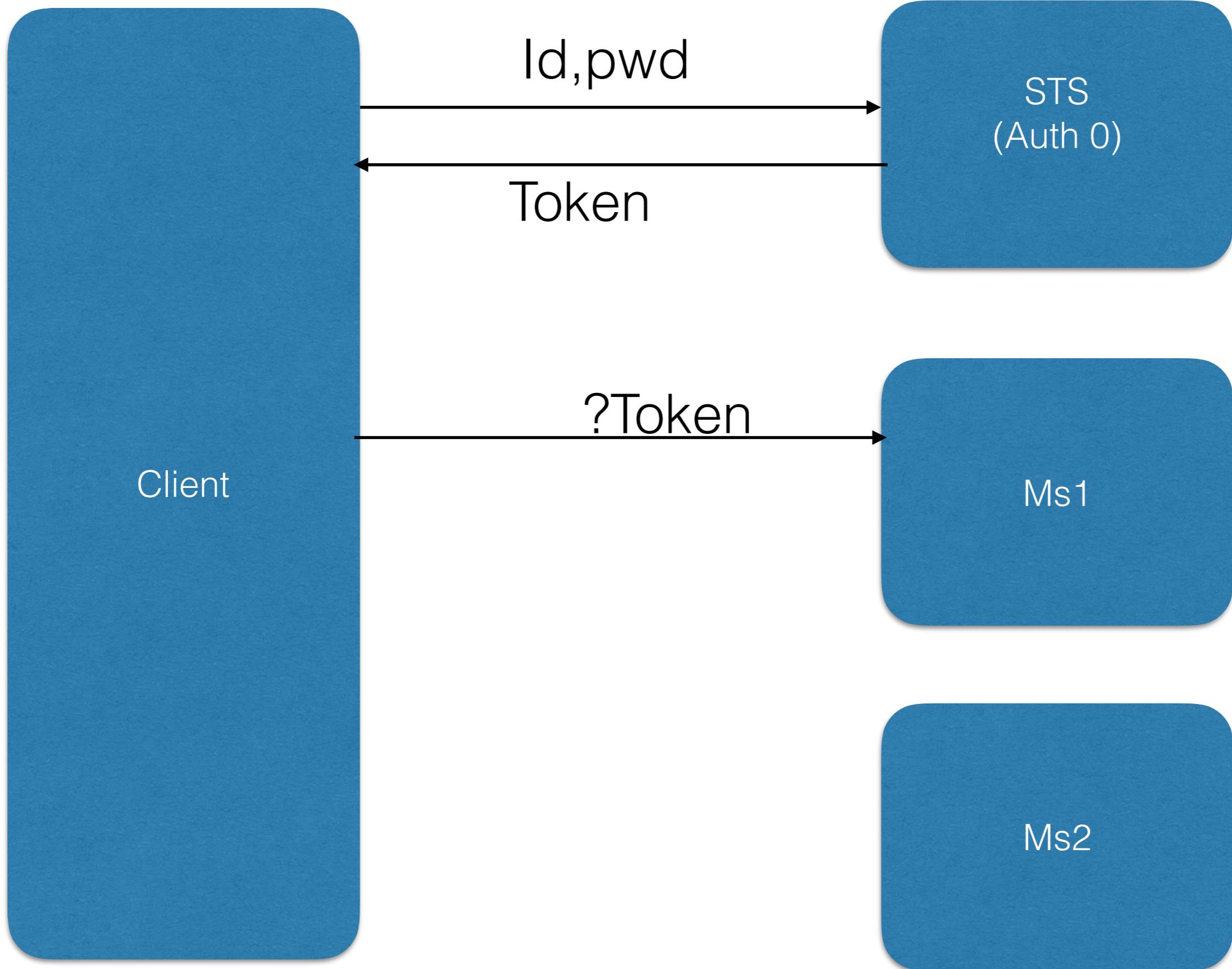
**By What you have**

**By what you know**

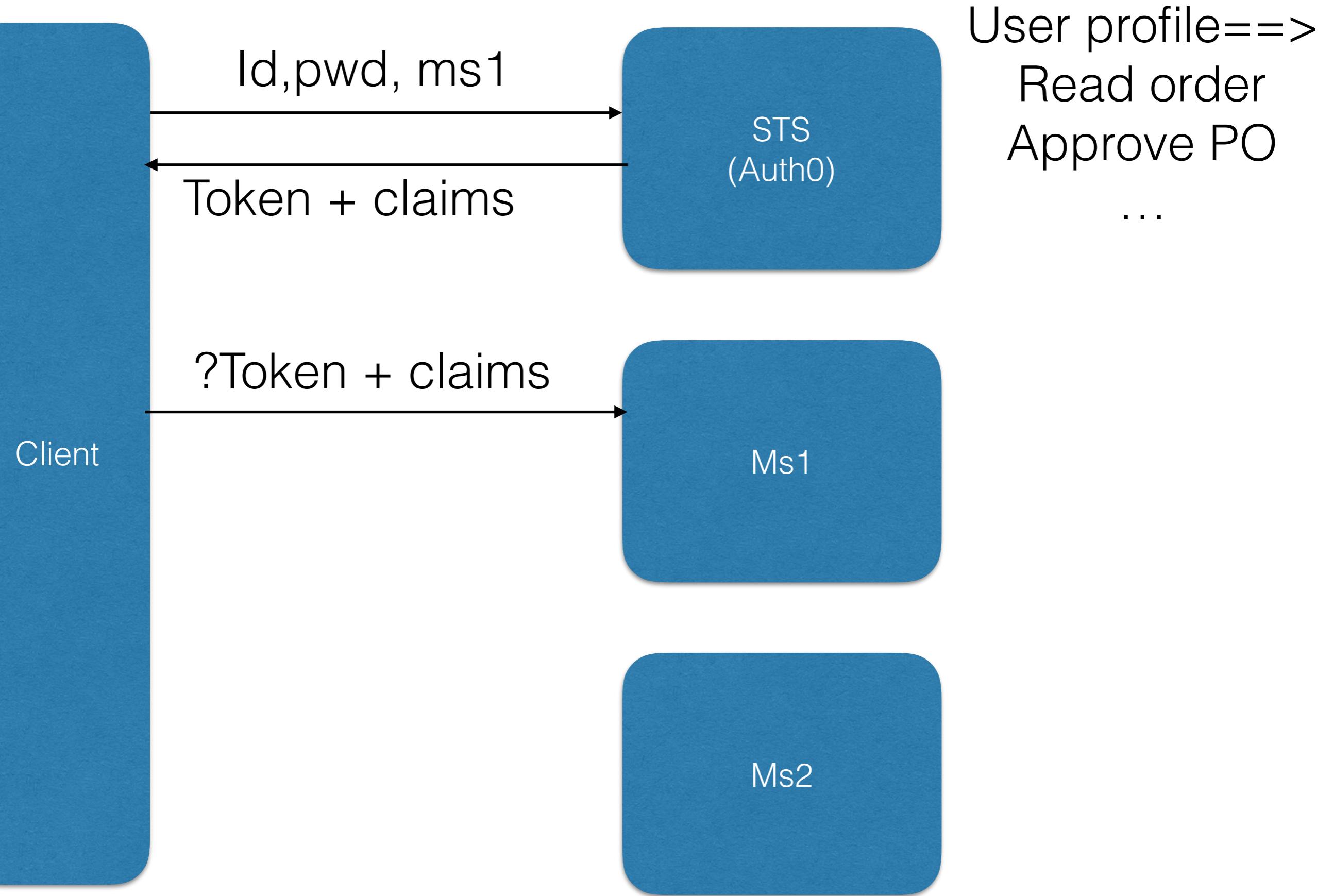
**By where you are \***

- Authentication (who is he? )-> (C) OAuth2
- Authorization (what can he do?)> (C) Claims
- Audit (what did he do?) -> (C) Centralize : event sourcing
- Asset Handling in Transit -> TLS / SSL
- Asset Handling in Rest -> (m) Encryption , Hashing, Signature
- Session Management -> (m) fwk
- Exception Handling -> (m) fwk
- Input Validation/ Encoding -> (m) fwk, (C) Api Gateway - WAF
- Throttling -> (C) ingress, APi gateway
- Secret Mgmt -> (C) Vault, Cert Store, ...

- vnet
- Subnet
- nsg
- VPN
- Tcp Layer firewall
- WAF firewall
-



OAuth2 + OpenID connect + JWT token  
SAML



L4 Firewall (TCP)

DOS, ..

L7 Firewall (http) - WAF

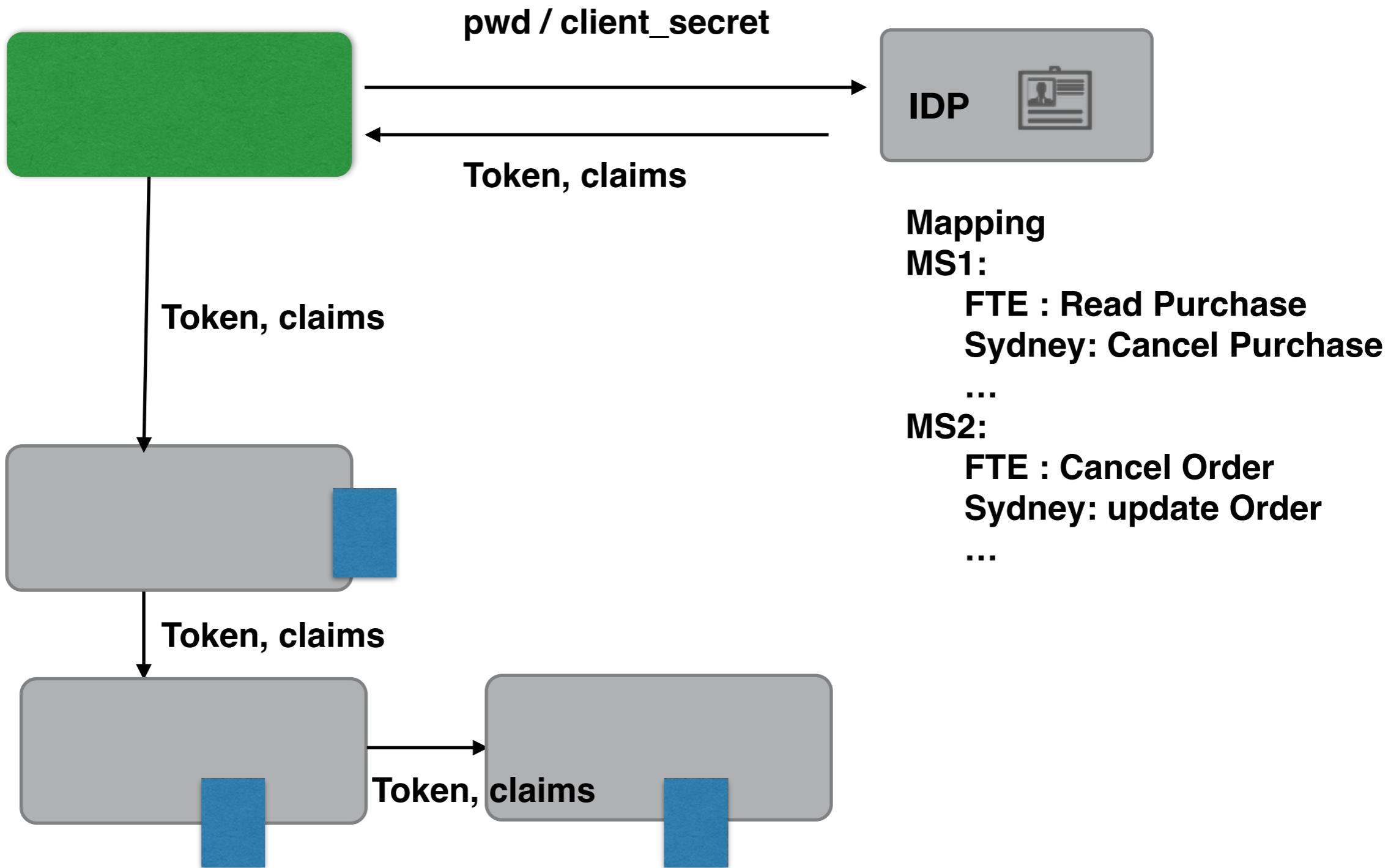
XSS, sql injection, one click, ....

Vnet

Subnet

Route Table

VPN



# STRIDE

- S : Spoofing <— Authentication
- T : Tampering <— Data in Transit, Data in Rest
- R : Repudation <— Audit
- I : Information Disclosure <— Authorization
- D : Deniel of Service <— input validation
- E : Elevation of Privileage

## Microservice App - 50%

- Authentication - OAuth2 + OpenID Connect + JWT token
- Authorization - RBAC, Claims
- Audit - Event Sourcing
- Data In Transit - https, was
- Data In Rest - Hash, Sym, Asym, cloud storage (\*)
- Input Validation - WAF, library (\*)
- Exception Handling - fwk
- Session Management- fwk
- Key Management - key vault

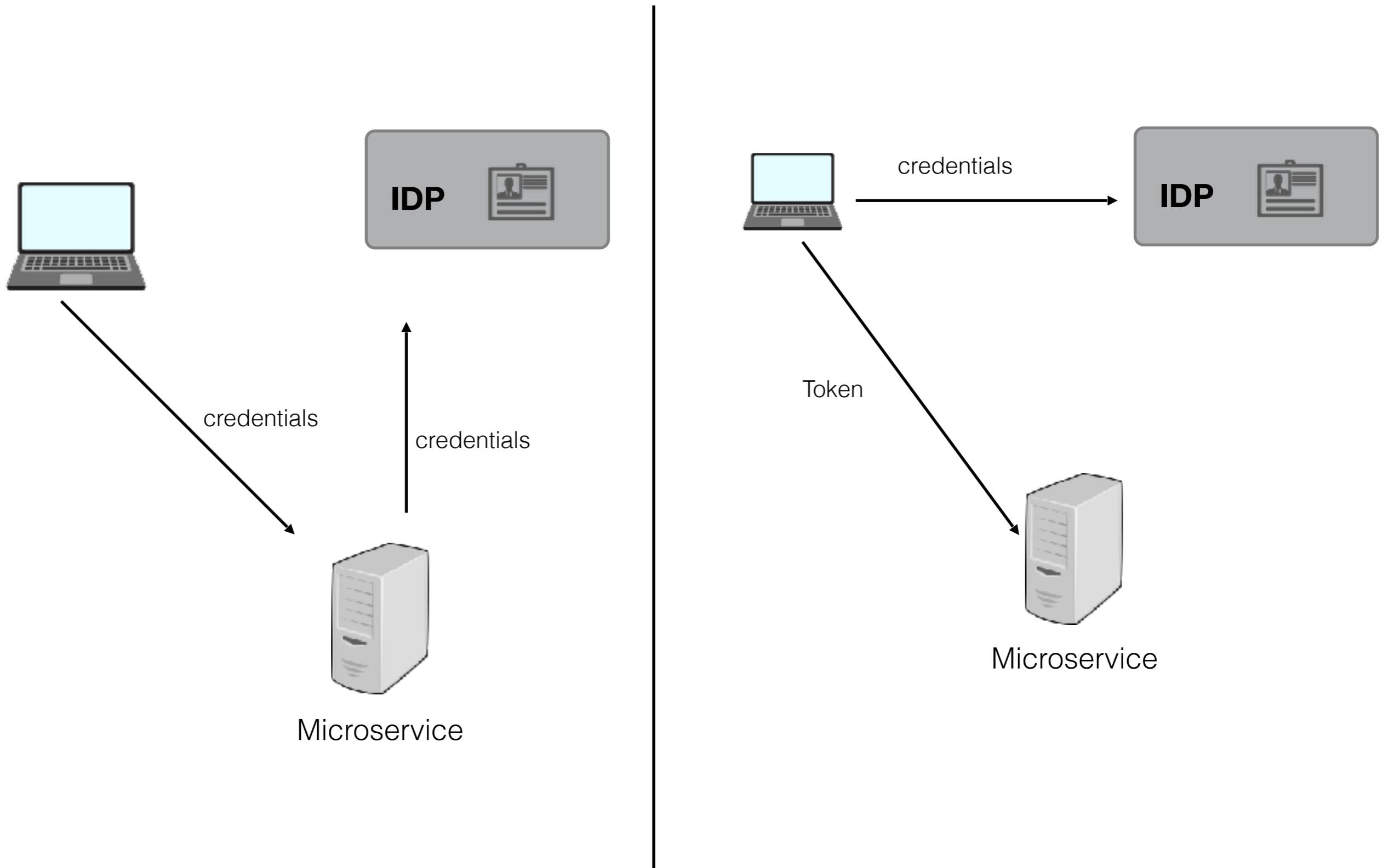
## DevSecops - 50%

- L4 Firewall (TCP)
- L7 Firewall (HTTP) - WAF
- vnet, subnet
- Route table
- Scanning tools
- Monitoring & Alerts

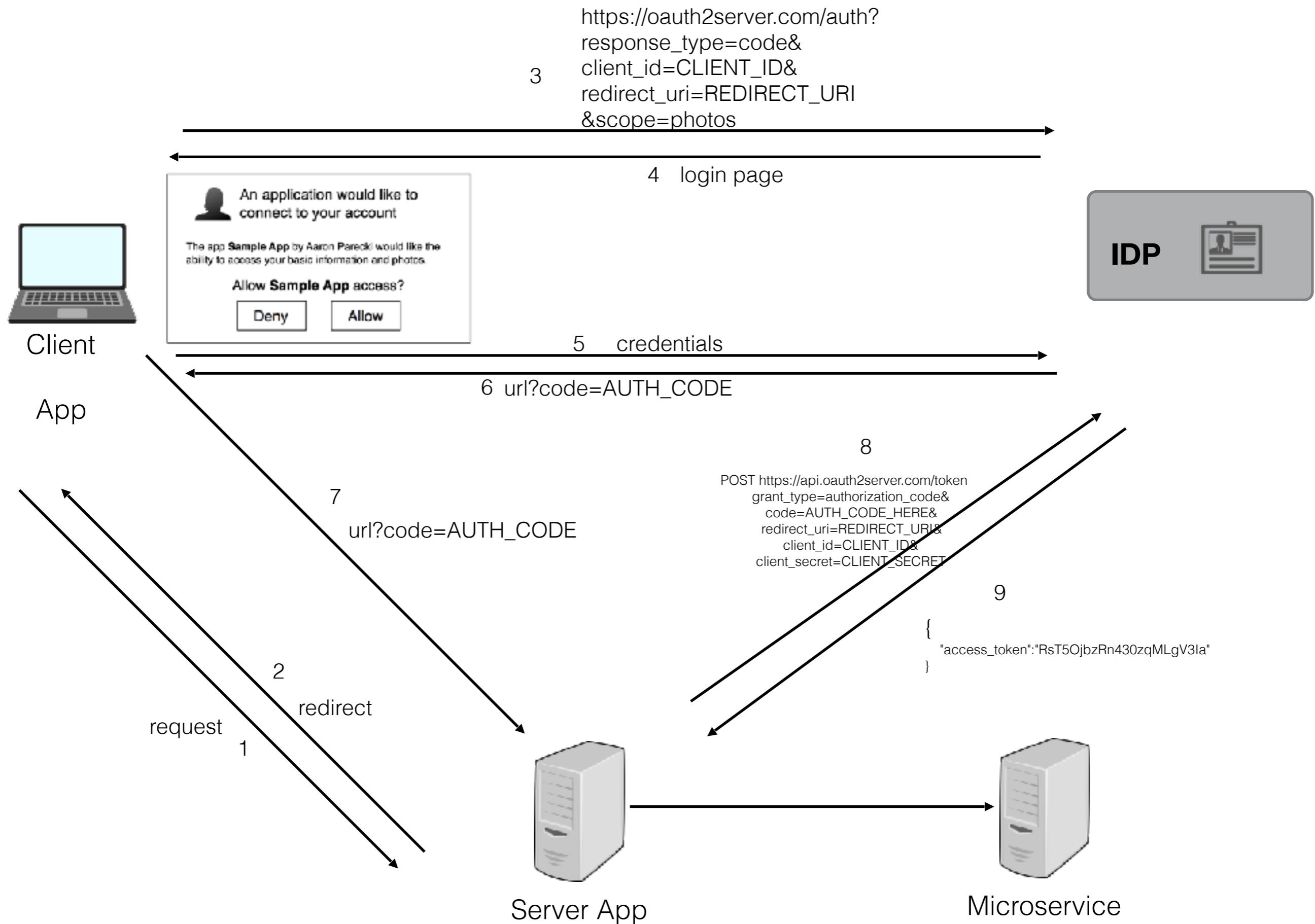
- By what you know (Knowledge)
  - Custom (name &pwd), **Oauth2, Openid connect**, Secret question,
- By what you have <— stolen
  - OTP, RSA, Cert, ...
- By what you are <— stolen
  - Bio (voice, retina, face, finger print, dna, ...)



# 2 legged vs 3 legged



# OAuth2: Authorization Code



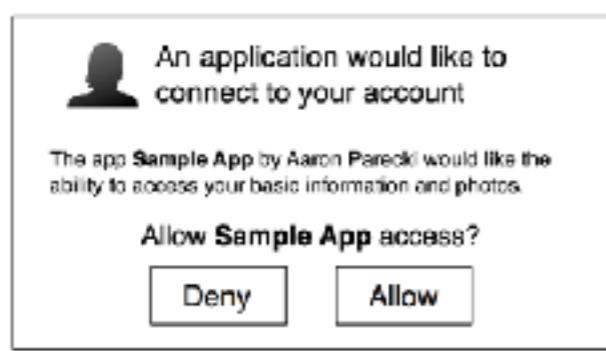
# Implicit

[https://oauth2server.com/auth?  
response\\_type=token&  
client\\_id=CLIENT\\_ID&  
redirect\\_uri=REDIRECT\\_URI&  
scope=photos](https://oauth2server.com/auth?response_type=token&client_id=CLIENT_ID&redirect_uri=REDIRECT_URI&scope=photos)



Client

App



2 login page



3 credentials

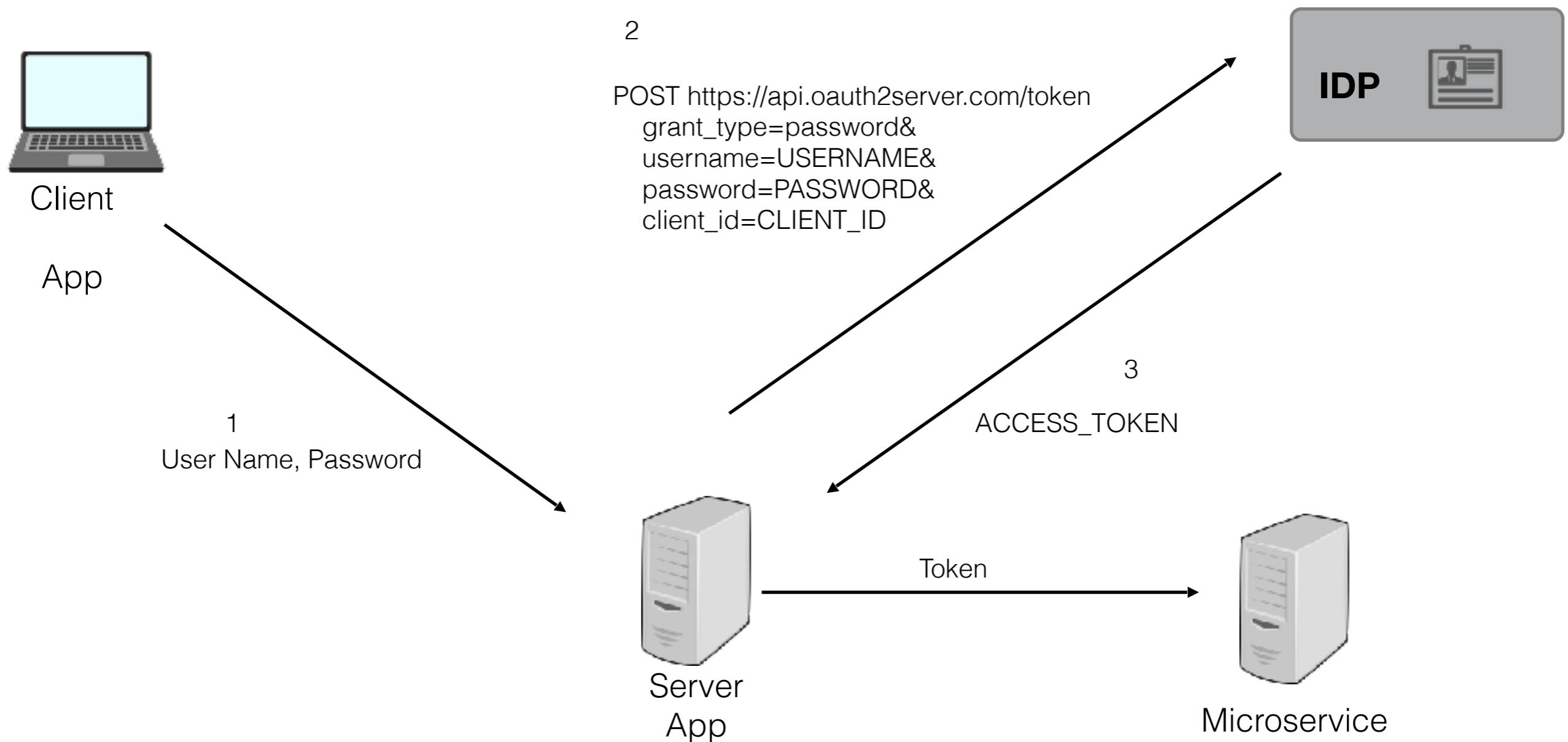
4 [https://oauth2client.com/cb#token=ACCESS\\_TOKEN](https://oauth2client.com/cb#token=ACCESS_TOKEN)

5 url?token

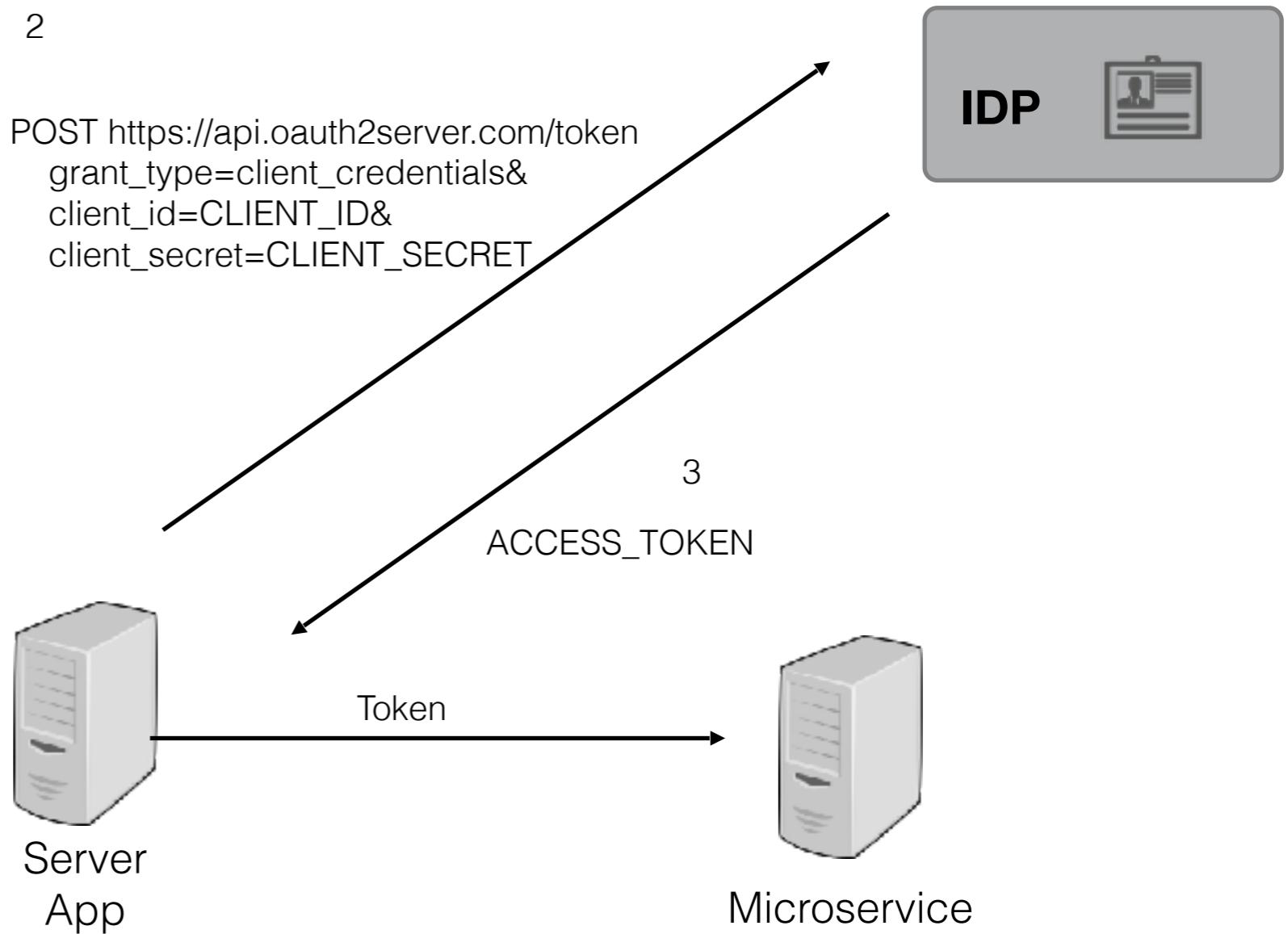


Microservice

# Password



# Clients Credential



# Security

App - AAA

Infra

- AuthN
- Authz
- Audit
- Data in Transit - SSL
- Data in Rest - cloud
- Input Handling - fwk
- Exception Handling - fwk
- Session Handling - fwk
- Key Management - vault
- Vnet/ subnet/ route
- VPN
- Firewall
-

## **1# Something you know**

- pwd\*, pin/key, secret

## **2# Something you Have**

- otp, email, message, rsa, cert, ....

## **3# Something you are**

- facial, finger, voice, retina, dna, ...

## **4# Where you are**

- country, property, ...

## **5# Something you do**

- walking pattern, ...

## **# Something you know**

- pwd\*, pin/key, secret

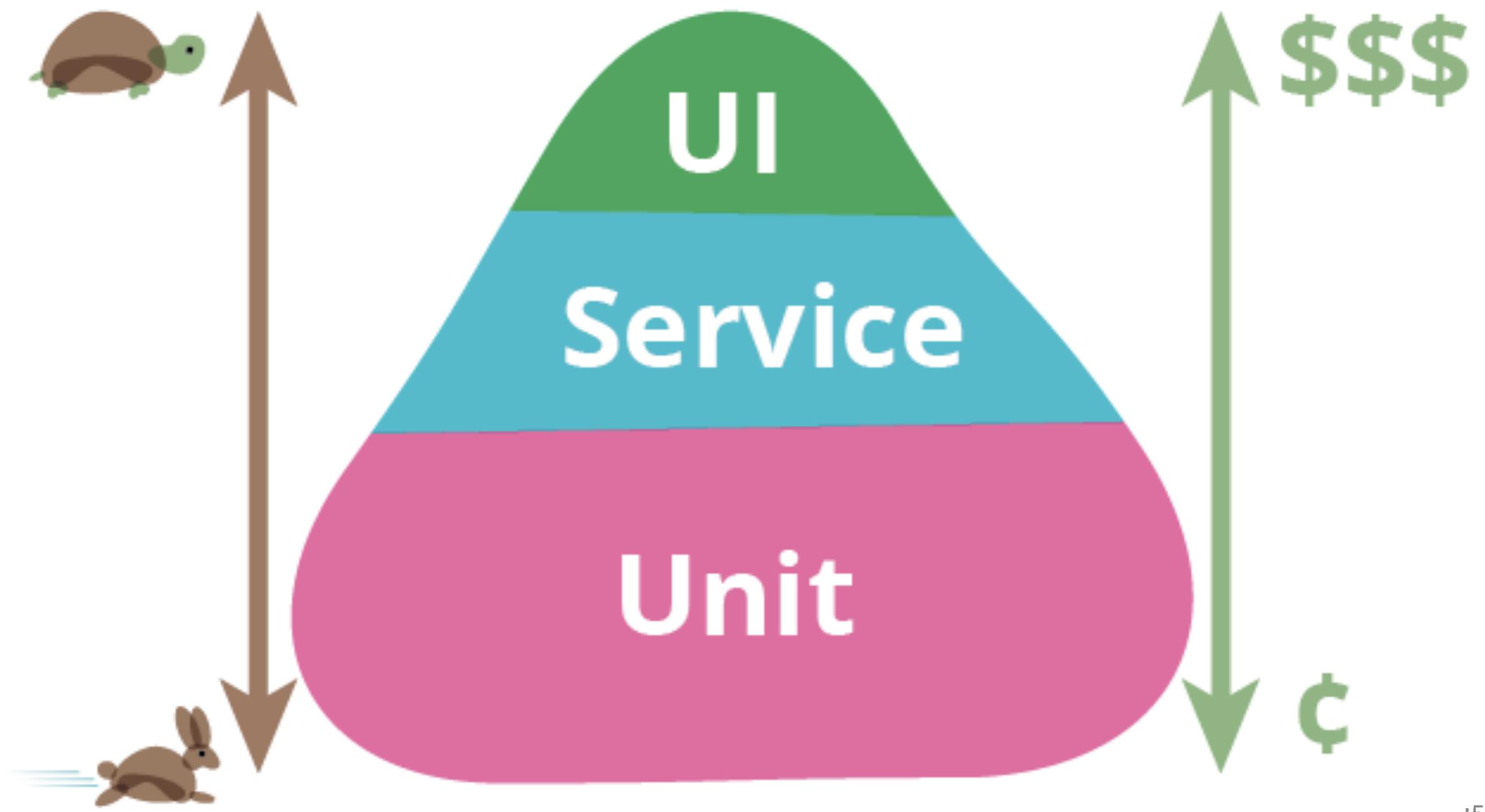
## **# Something you Have**

- otp, email, message, rsa, cert, ....

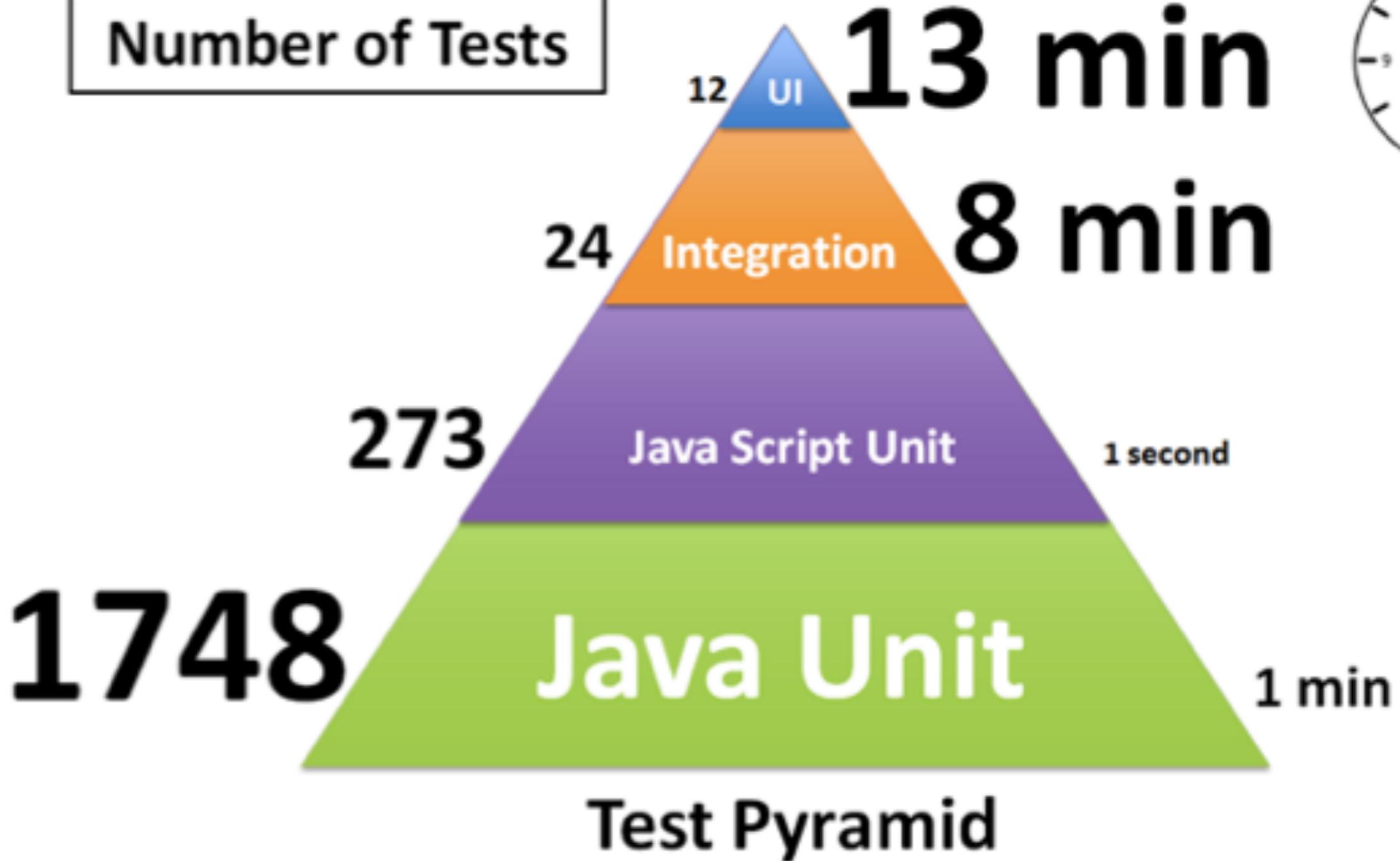
## **PWD**

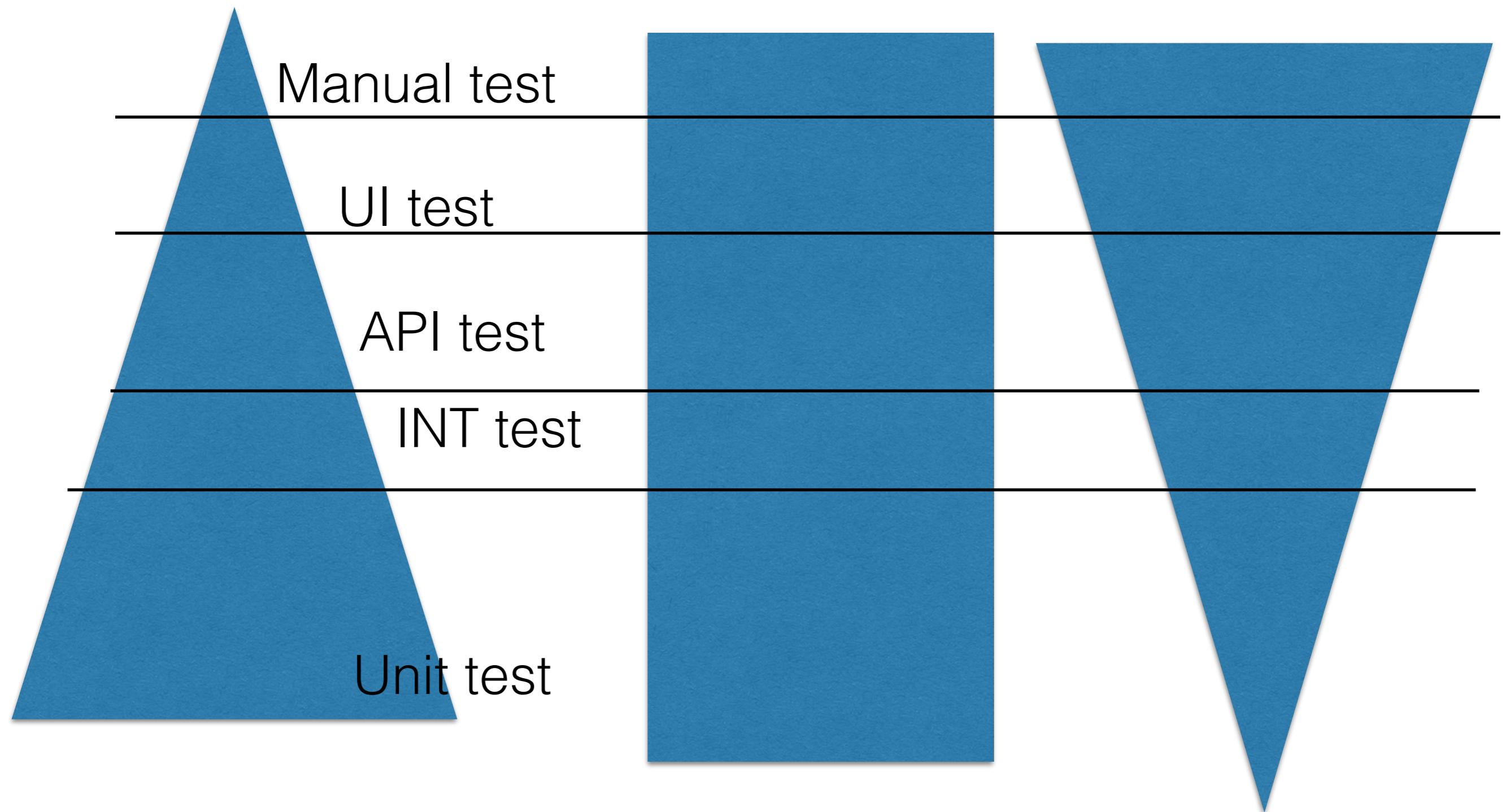
- Oauth2 + Openid connect
- SAML

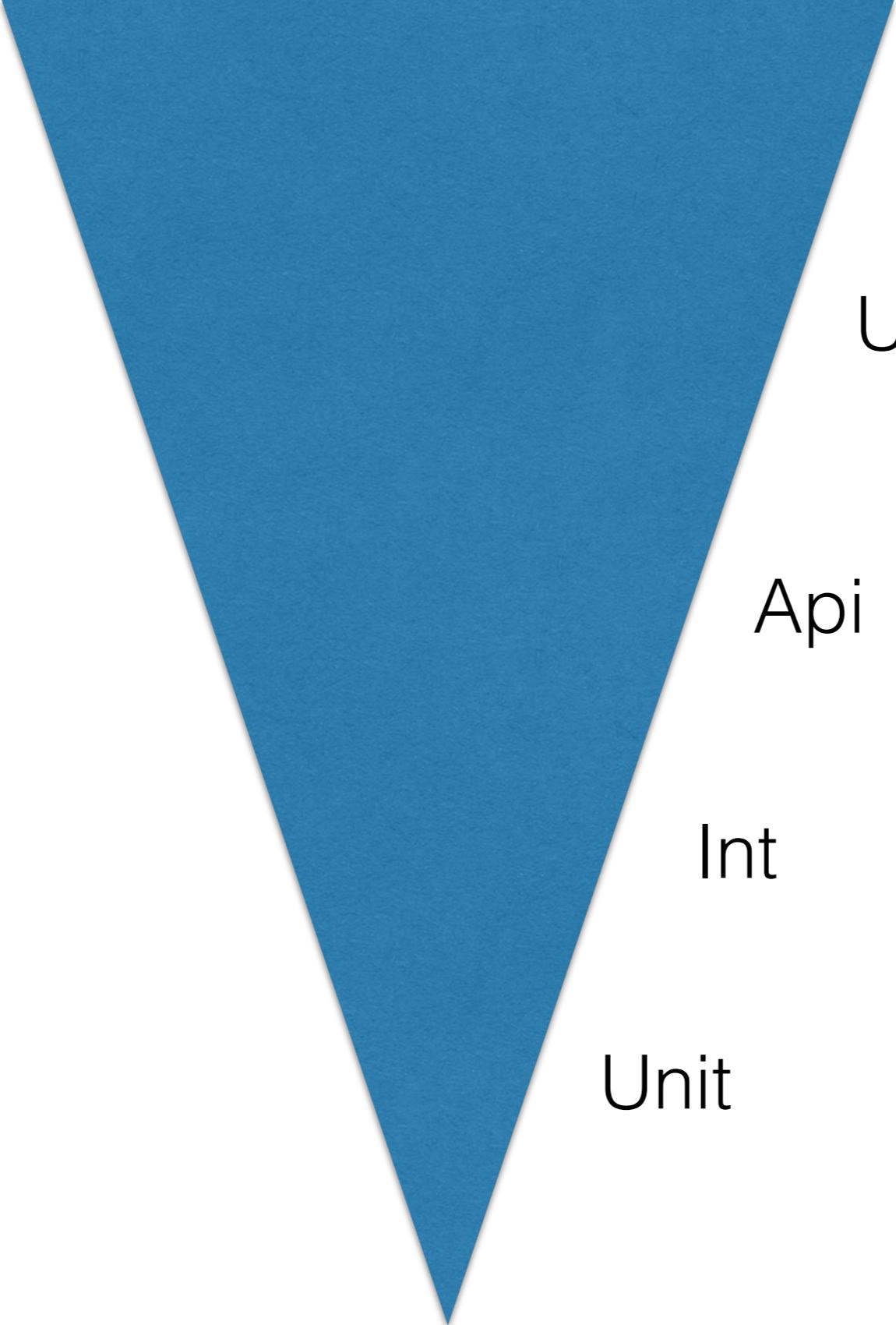
# Test Pyramid



**Number of Tests**







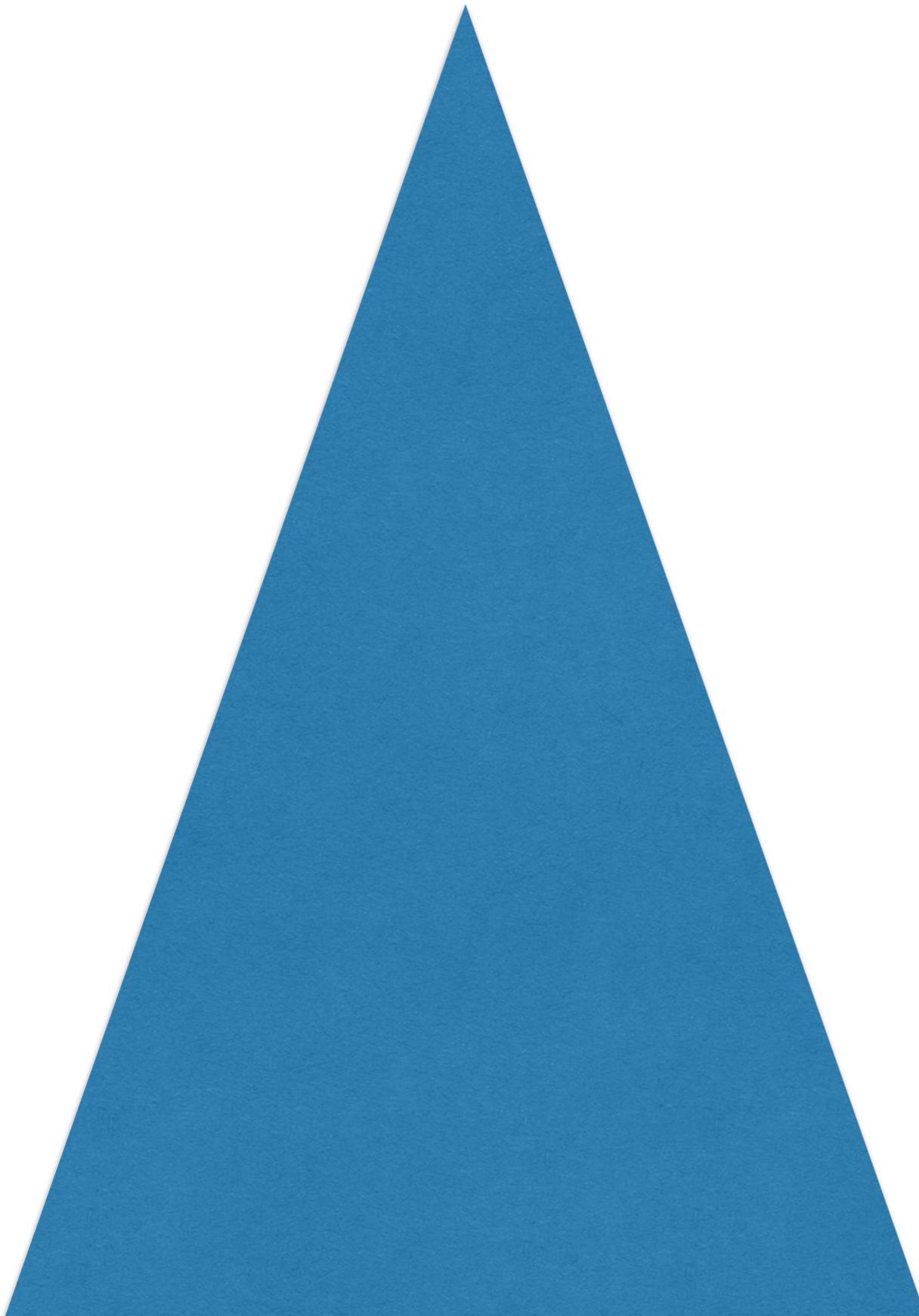
Manual

Ui automation

Api

Int

Unit



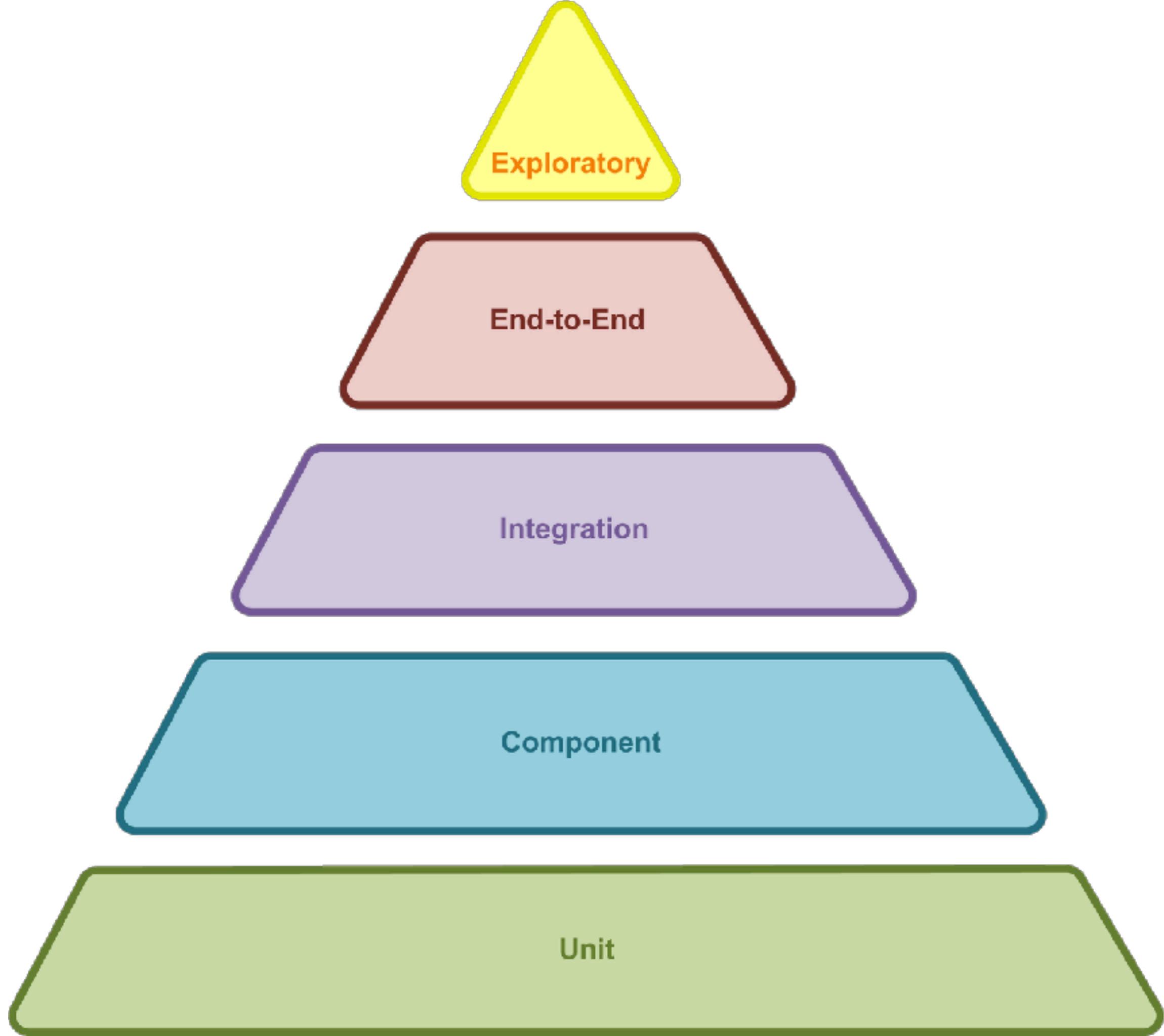
Manual

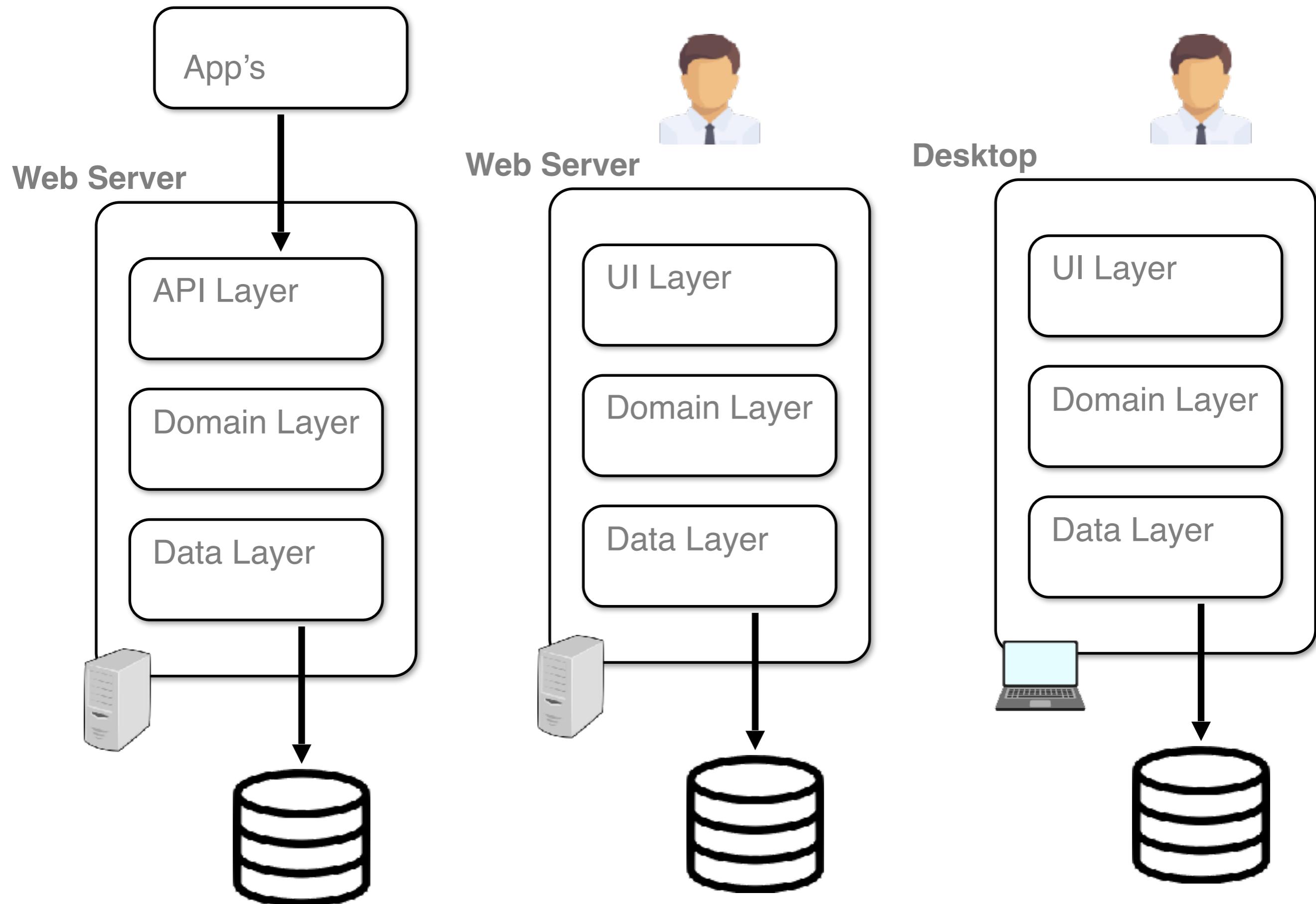
Ui automation

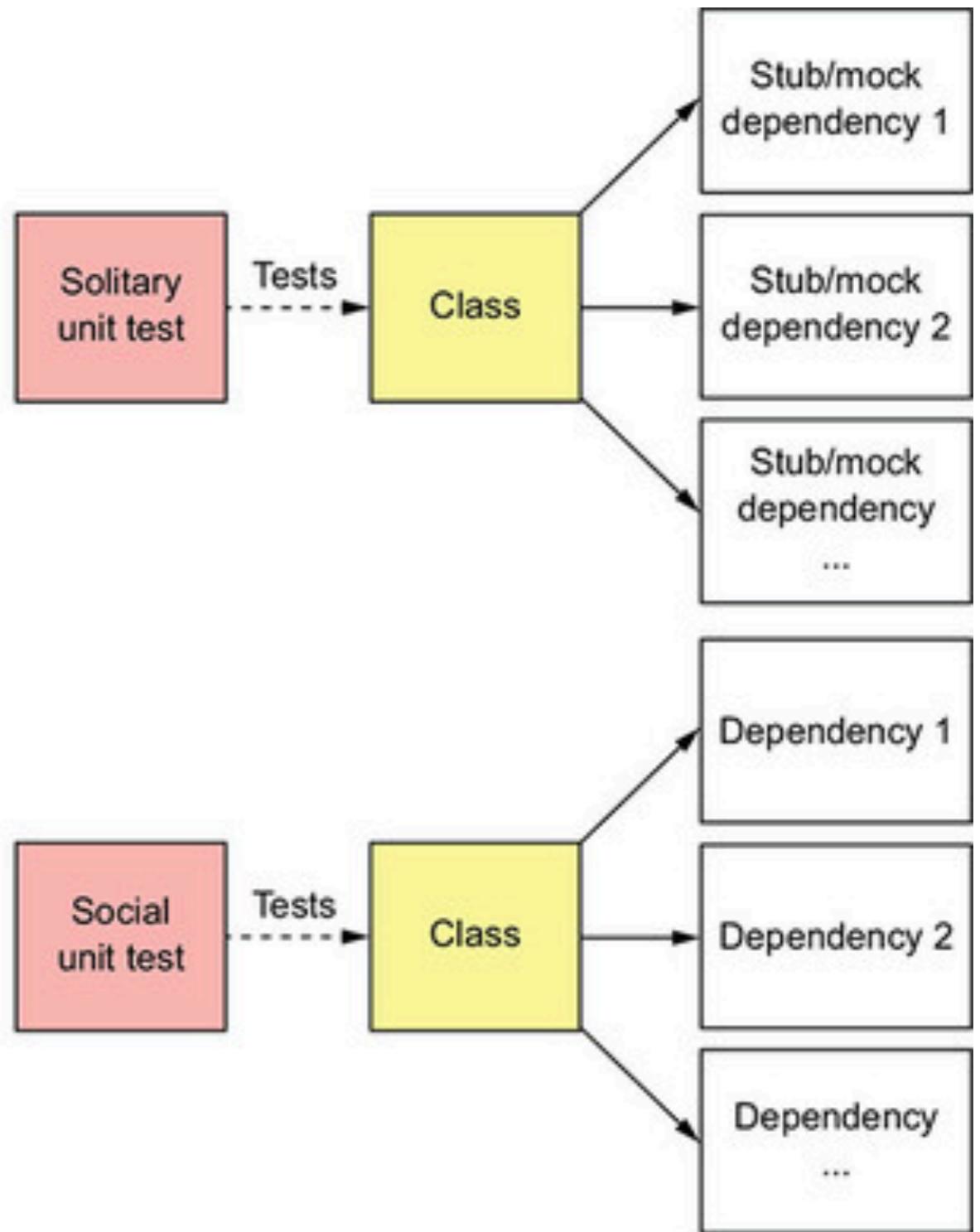
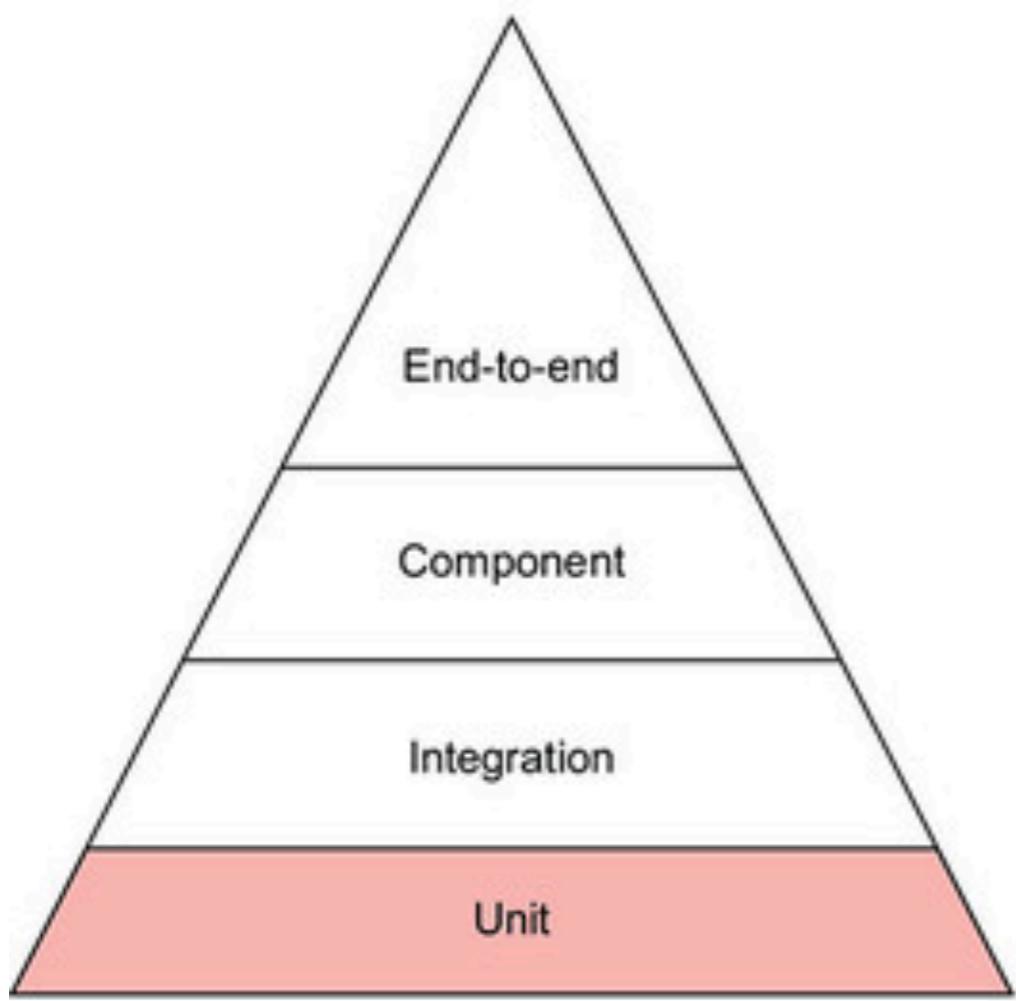
Api

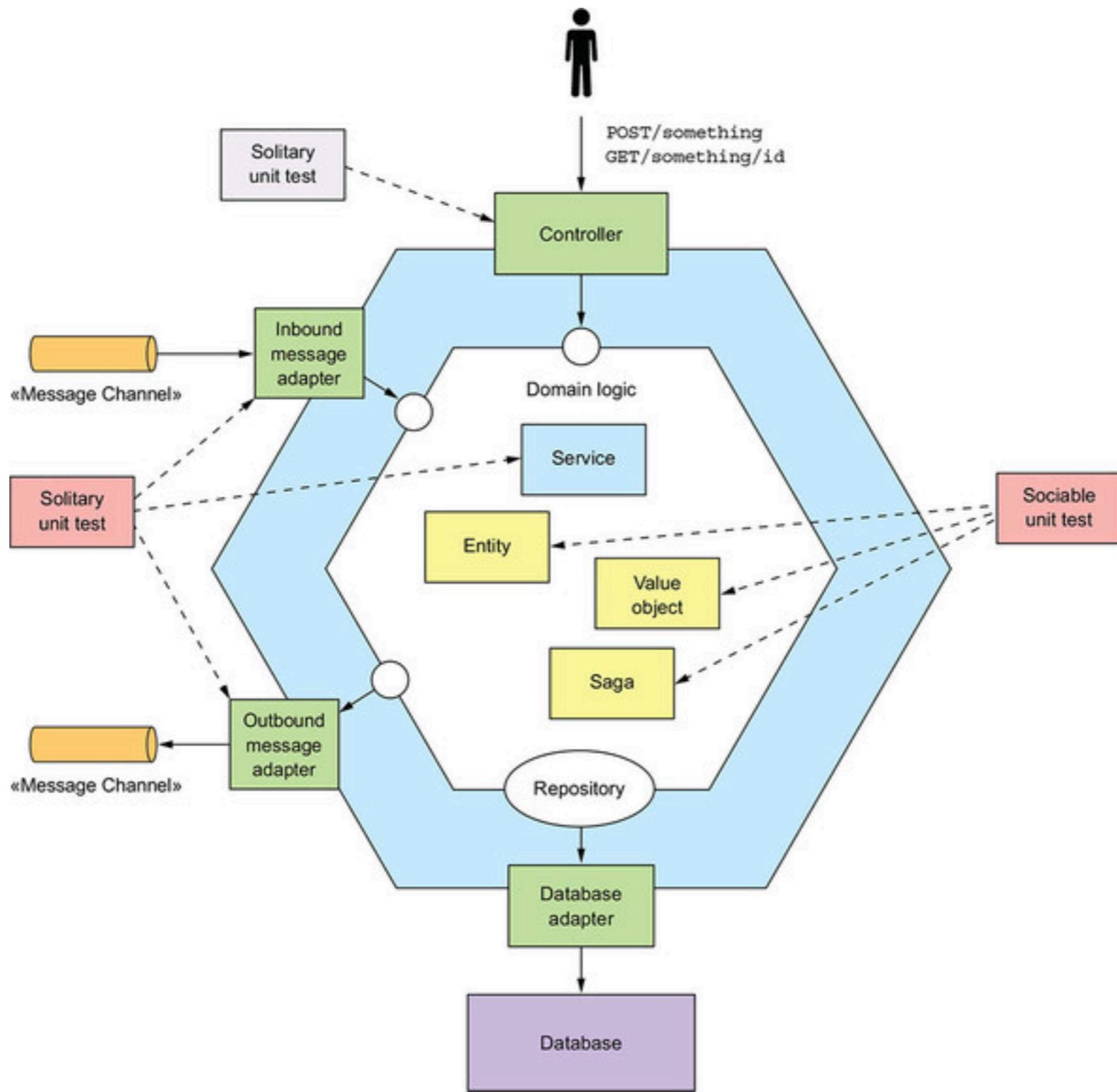
Int

Unit







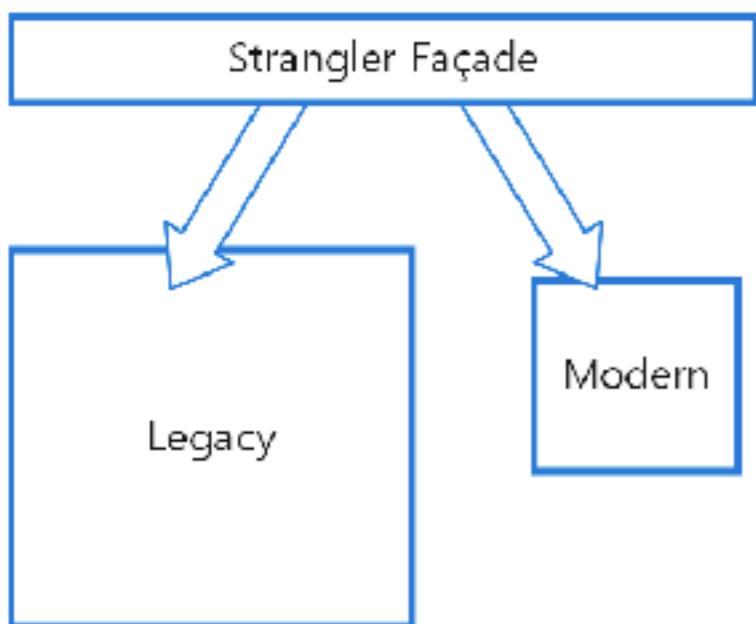


# Patterns

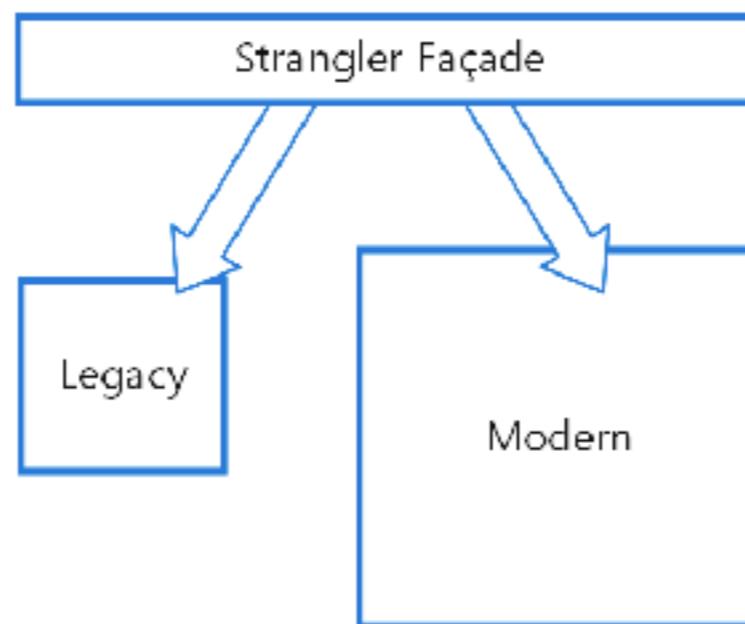


# Strangler pattern

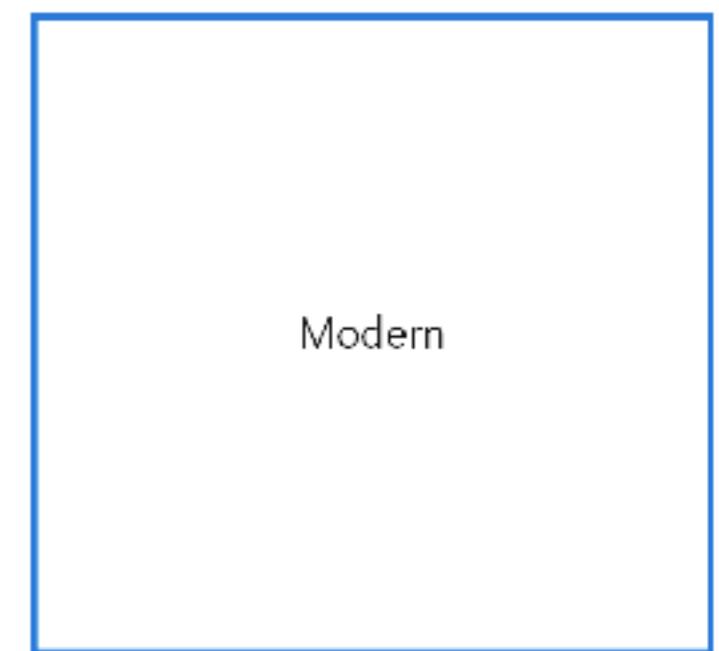
Early migration



Later migration

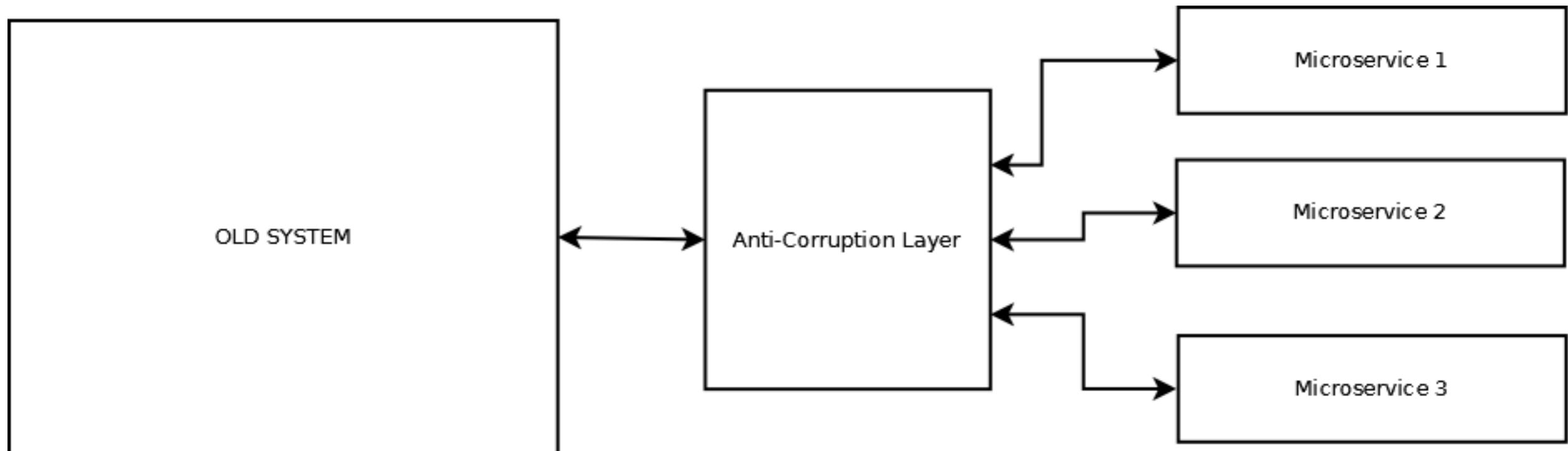


Migration complete



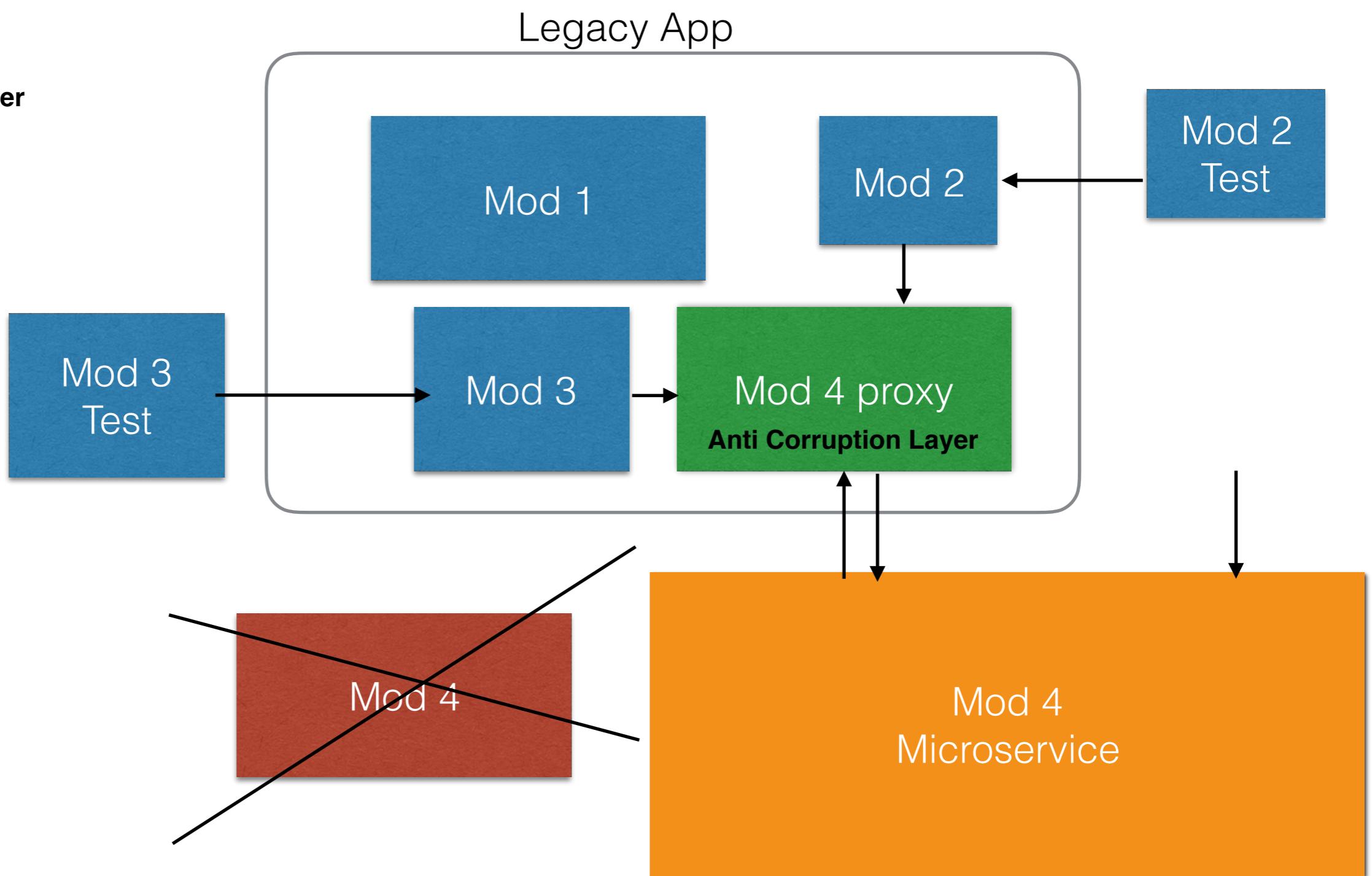
- Create a façade that intercepts requests going to the backend legacy system. Over time, as features are migrated to the new system, the legacy system is eventually "strangled" and is no longer necessary.

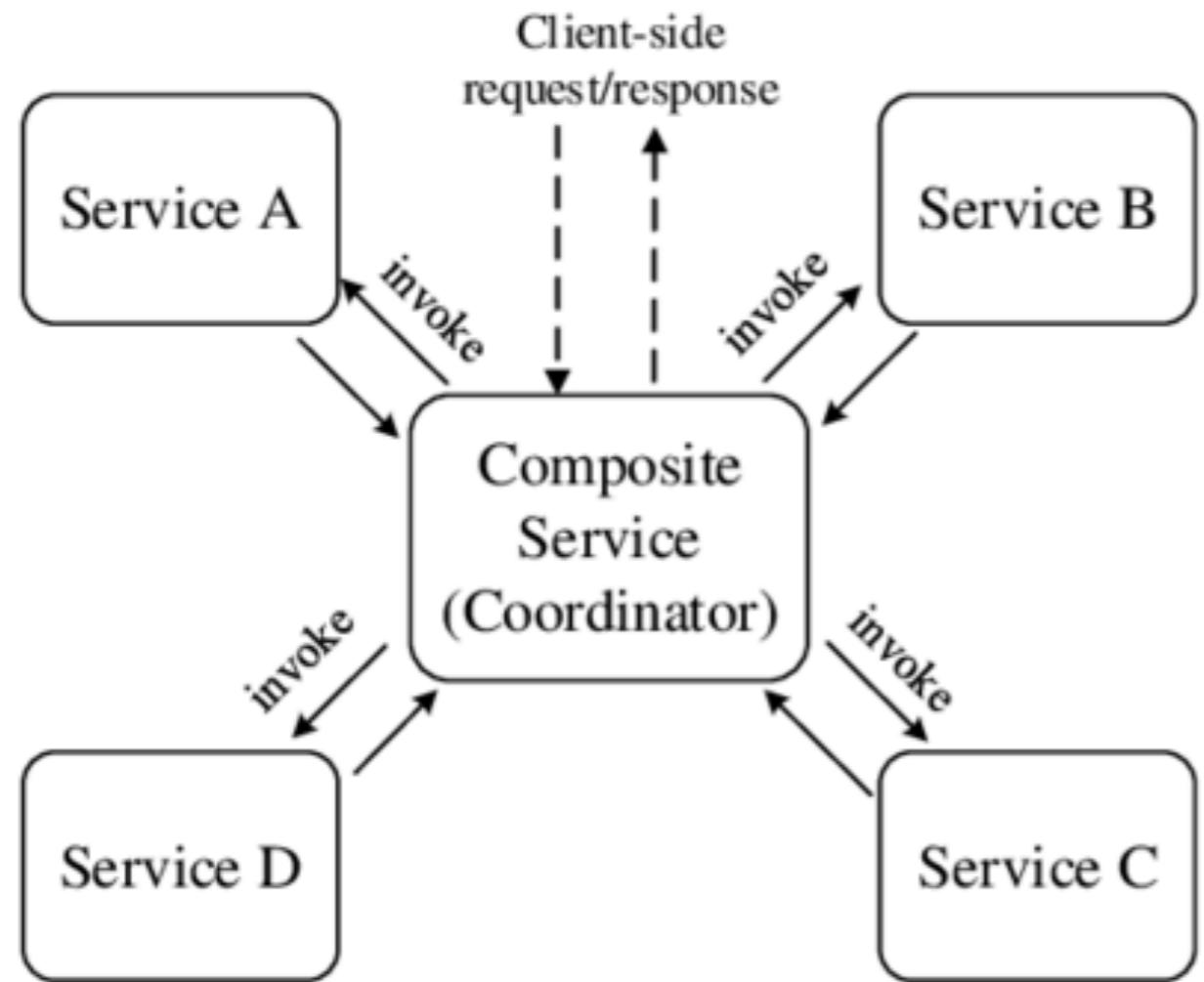
# Anti-Corruption Layer



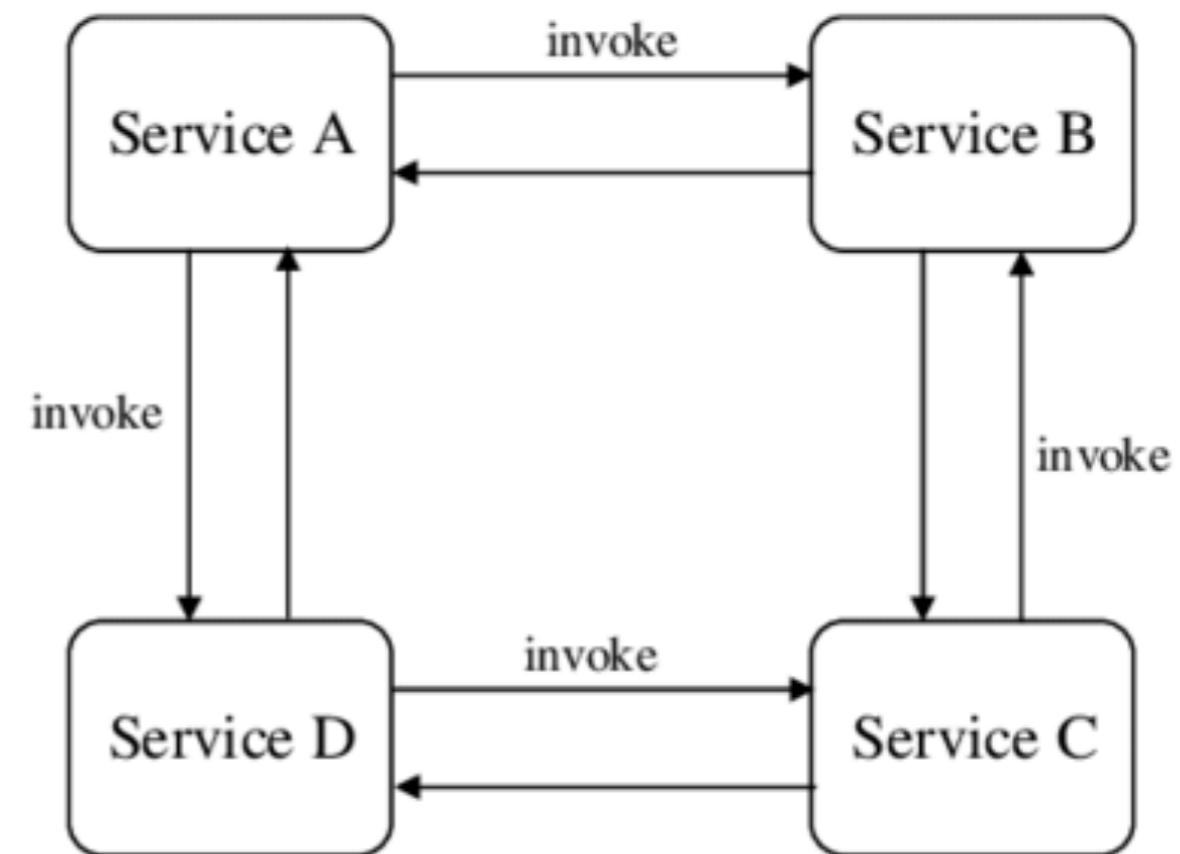
- When you create a new microservice, you notice that some business or technical assumptions or flow should be remodeled. instead of a Big Rewrite ACL maps one domain onto another so that services that use second domain do not have to be "corrupted" by concepts from the first.
- Such modifications may result in changes to the events that a bounded context publishes.

```
unit test
Strangler Pattern
Anti Corruption Layer
```





(a) Web Service Orchestration



(b) Web Service Choreography

# Libreria Angular

App 1

App 2

App 3

App 4

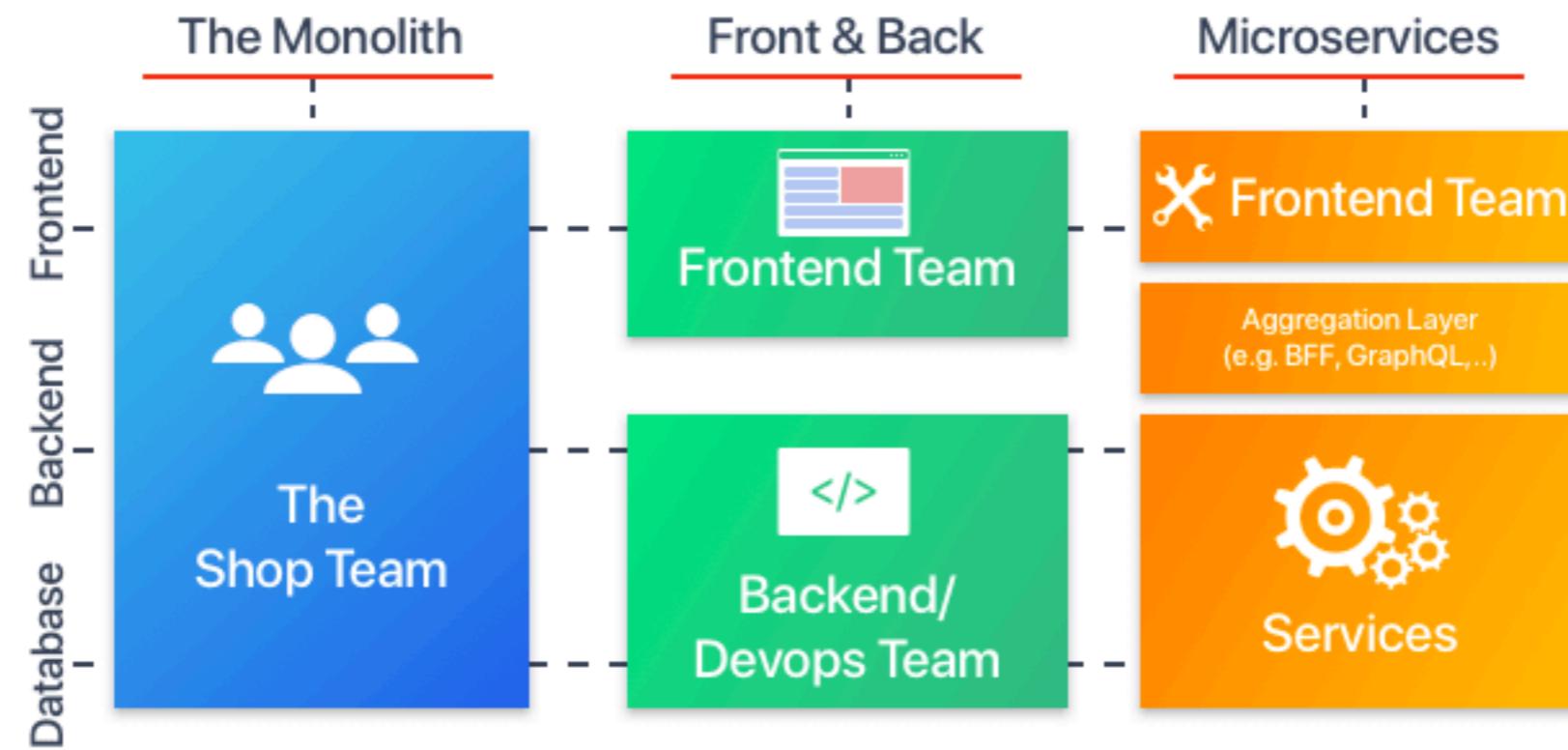
Libreria  
Angular

Micro-  
Frontends



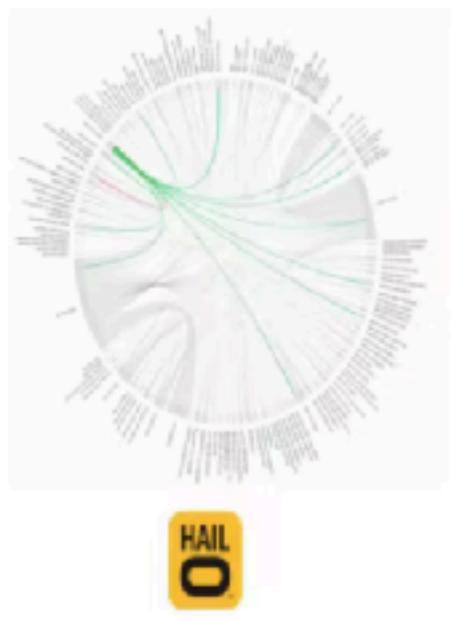
# Single-SPA Framework

# Micro Frontend Architecture



# Case Study

450 microservices

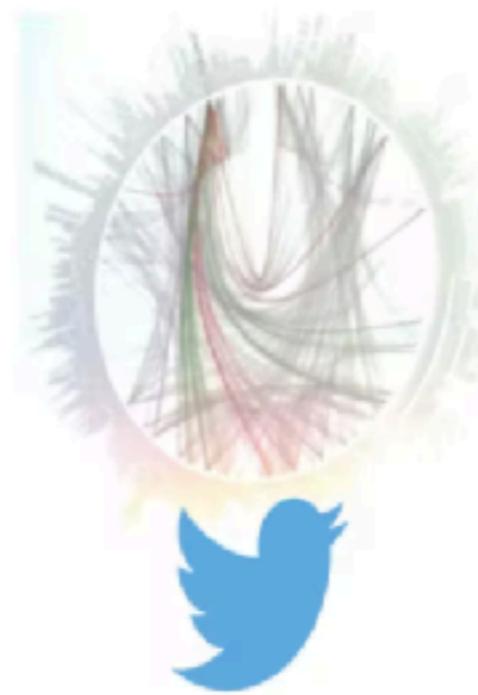


500+ microservices

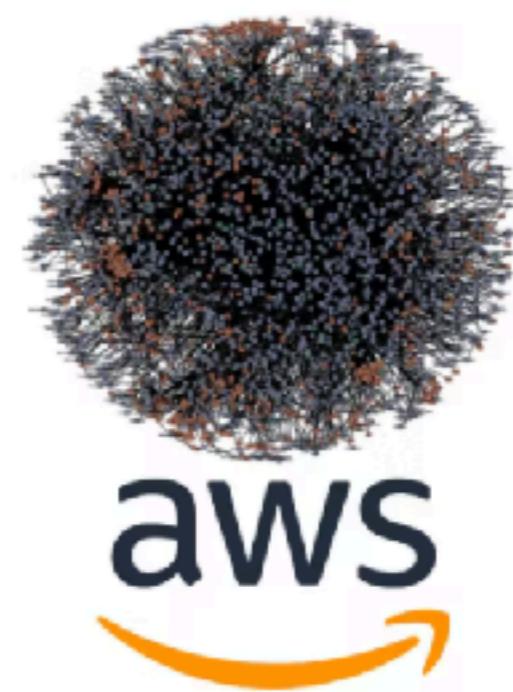


NETFLIX

500+ microservices



500+ microservices



aws

coursera

Coca-Cola

ebay

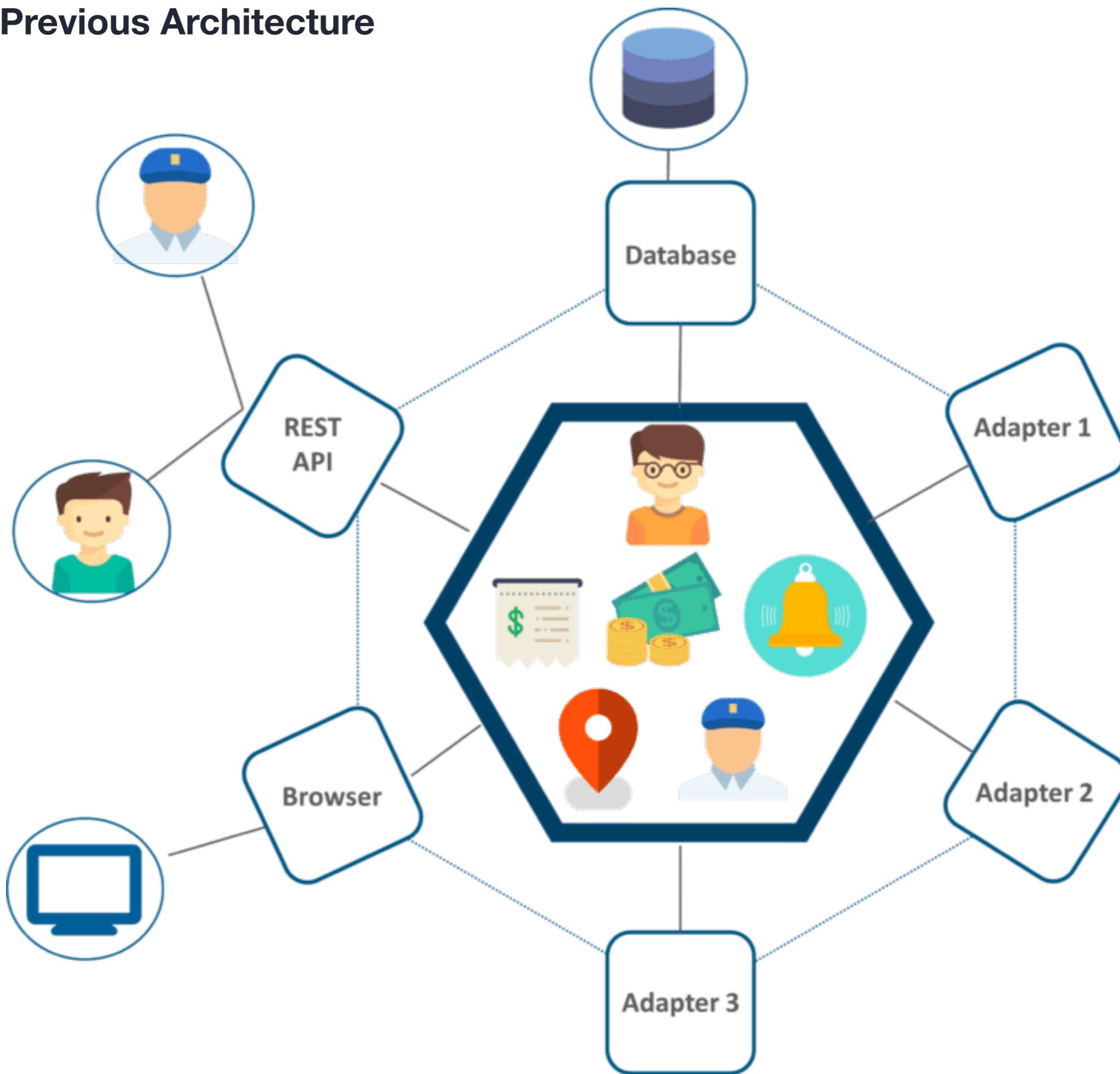
Etsy

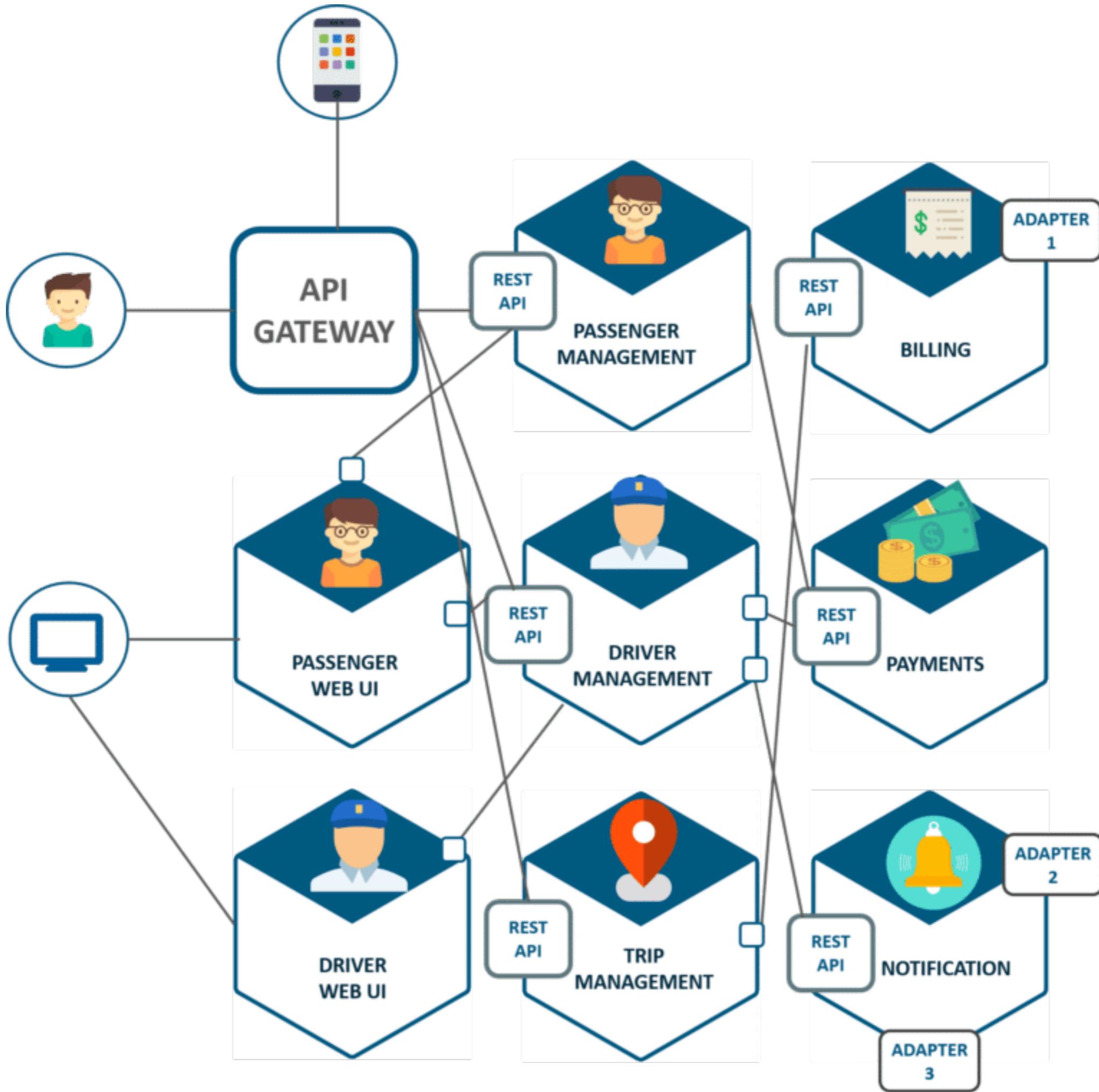
Spotify

UBER



# Uber's Previous Architecture





Netflix turning towards microservices starts in 2009,

Netflix needed 2 years to split their monolith into microservices, and in 2011 announced end of redesigning their structure and organizing it using microservice architecture.

the Netflix application leverages 500+ microservices and API Gateways that handles over 2 billion API edge requests daily.

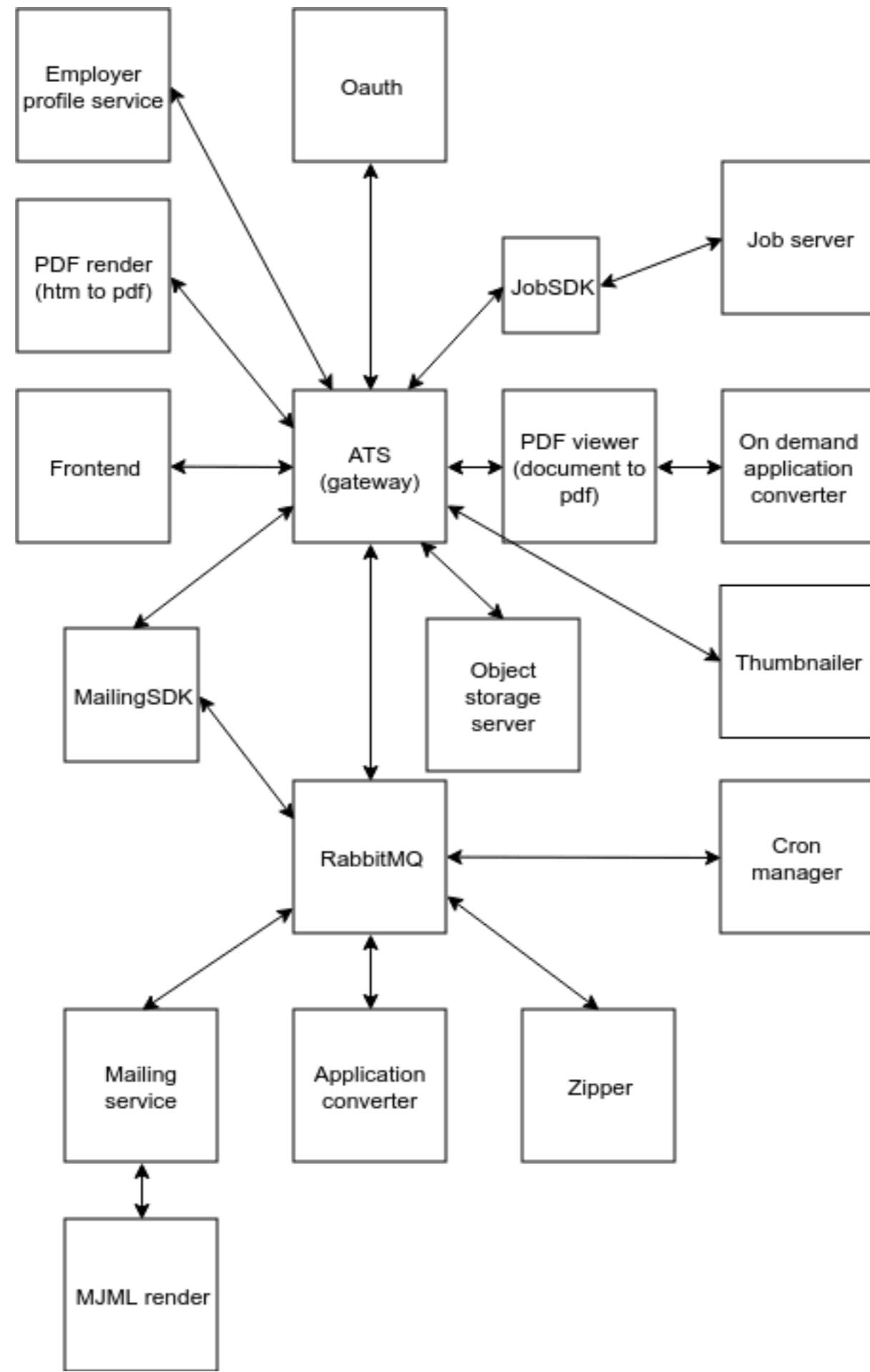
# Gilt.com

Gilt reconstructed their monolithic architecture, which was based on Ruby on Rails, to a microservice architecture in 2011. based it on Scala, Docker and AWS technologies and initially created 156 services.

# Application Tracking System (ATS)

1. After adding a new feature, it goes into the manual testing phase. Once it's greenlit, you deploy it to production — but what often happens is you miss a testing scenario, and you end up with lots of bugs in production.
2. This problem is tightly connected to the previous one: If you have an older codebase, its foundation starts to slowly break down, so adding new code on top of it ends up relying on other code. With that kind of codebase, sometimes you end up breaking code that should be totally irrelevant to what you're currently writing.
3. When you have 11 people working on the same codebase day in, day out, without any safeties in place whatsoever, problems are bound to arise.

- One Year
- ~50,000 lines of code
- ~2,600 commits
- a team of 3 developers
- consisting of a senior full stack developer and two medior backend developers
- 18 services
- 200 tests



For example, in a hospital domain, a **Patient** being treated in the outpatients department might have a list of **Referrals**, and methods such as *BookAppointment()*. A **Patient** being treated as an Inpatient however, will have a **Ward** property and methods such as *TransferToTheatre()*. Given this, there are two bounded contexts that patients exist in: Outpatients & Inpatients.

In the insurance domain, the sales team put together a **Policy** that has a degree of risk associated to it and therefore cost. But if it reaches the claims department, that information is meaningless to them. They only need to verify whether the policy is valid for the claim. So there are two contexts here: Sales & Claims

the buyer entity might have most of a person's attributes that are defined in the user entity in the profile or identity microservice, including the identity. But the buyer entity in the ordering microservice might have fewer attributes, because only certain buyer data is related to the order process. The context of each microservice or Bounded Context impacts its domain model.

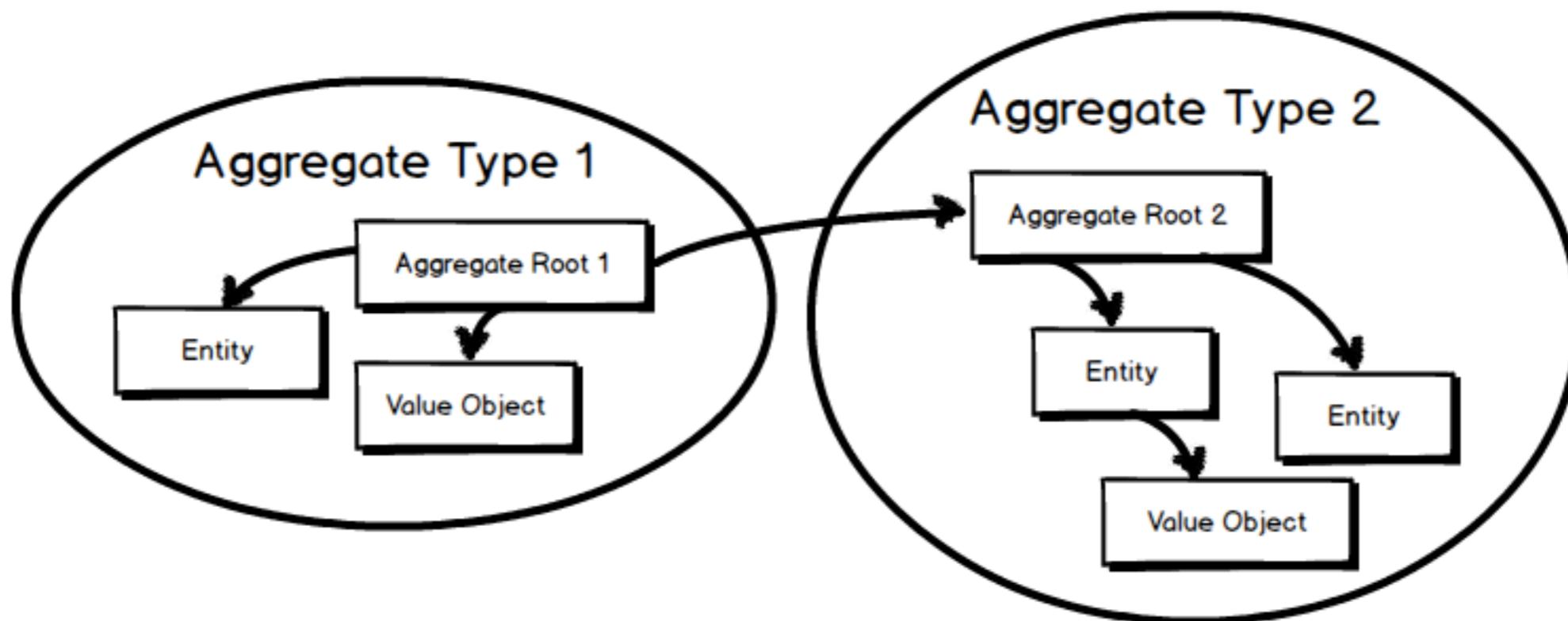
# Patterns

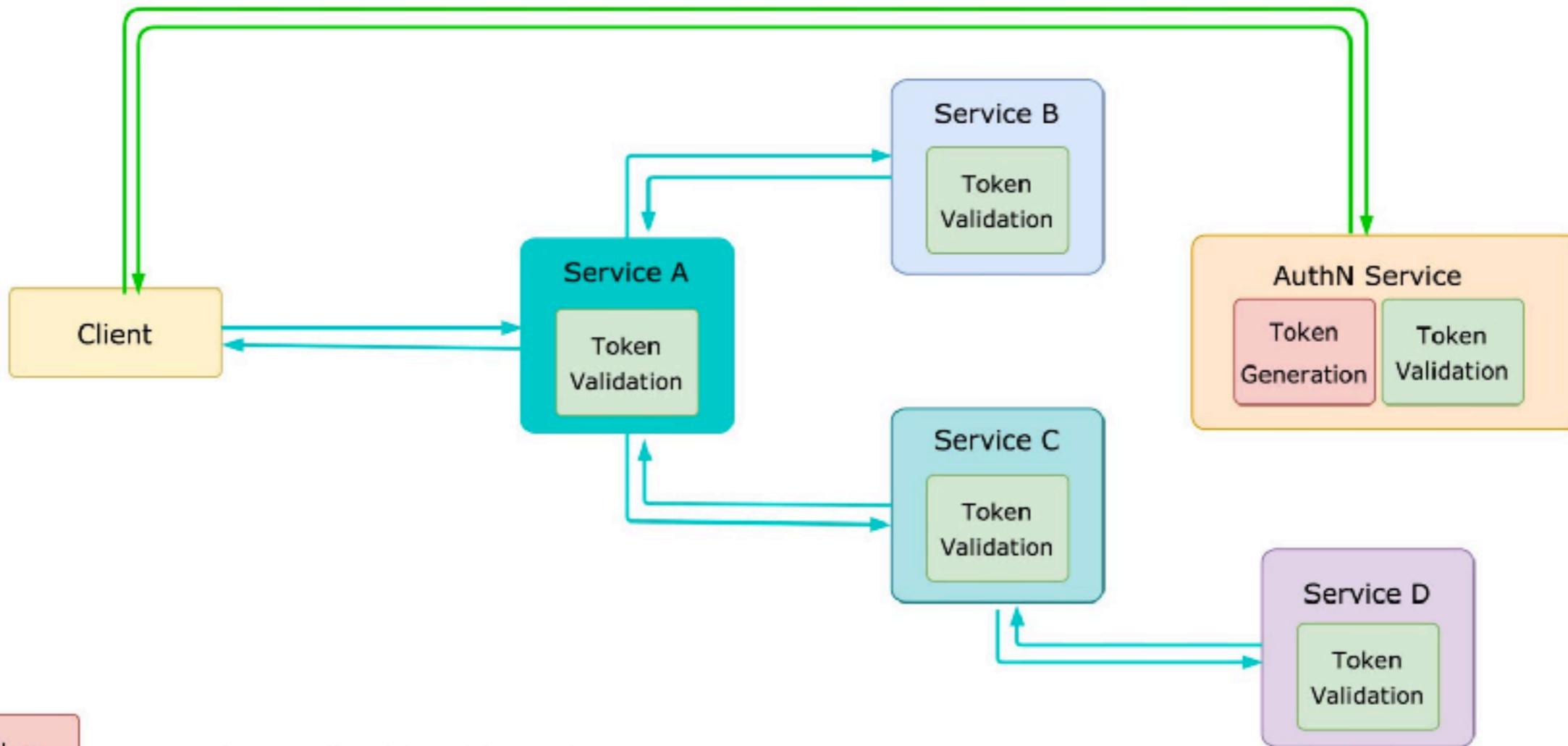
- Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.

# Bounded Context



# Aggregate Root





Token  
Validation

Token Generation Using Private Key

Token  
Validation

Token Validation Using Public Key

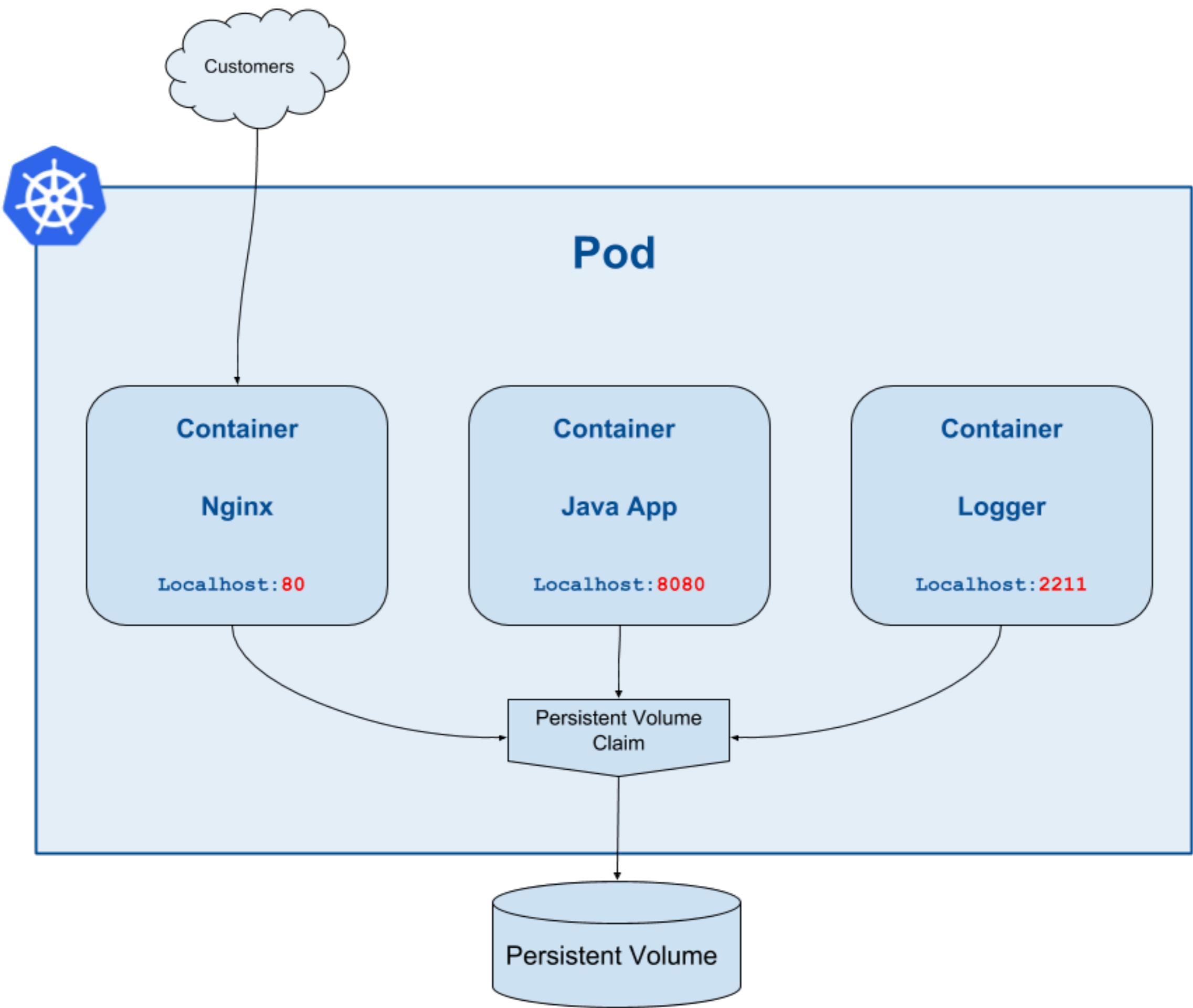
Any service Can Validate Using Public Key

Private Key in One Secure Place, Really Insecure

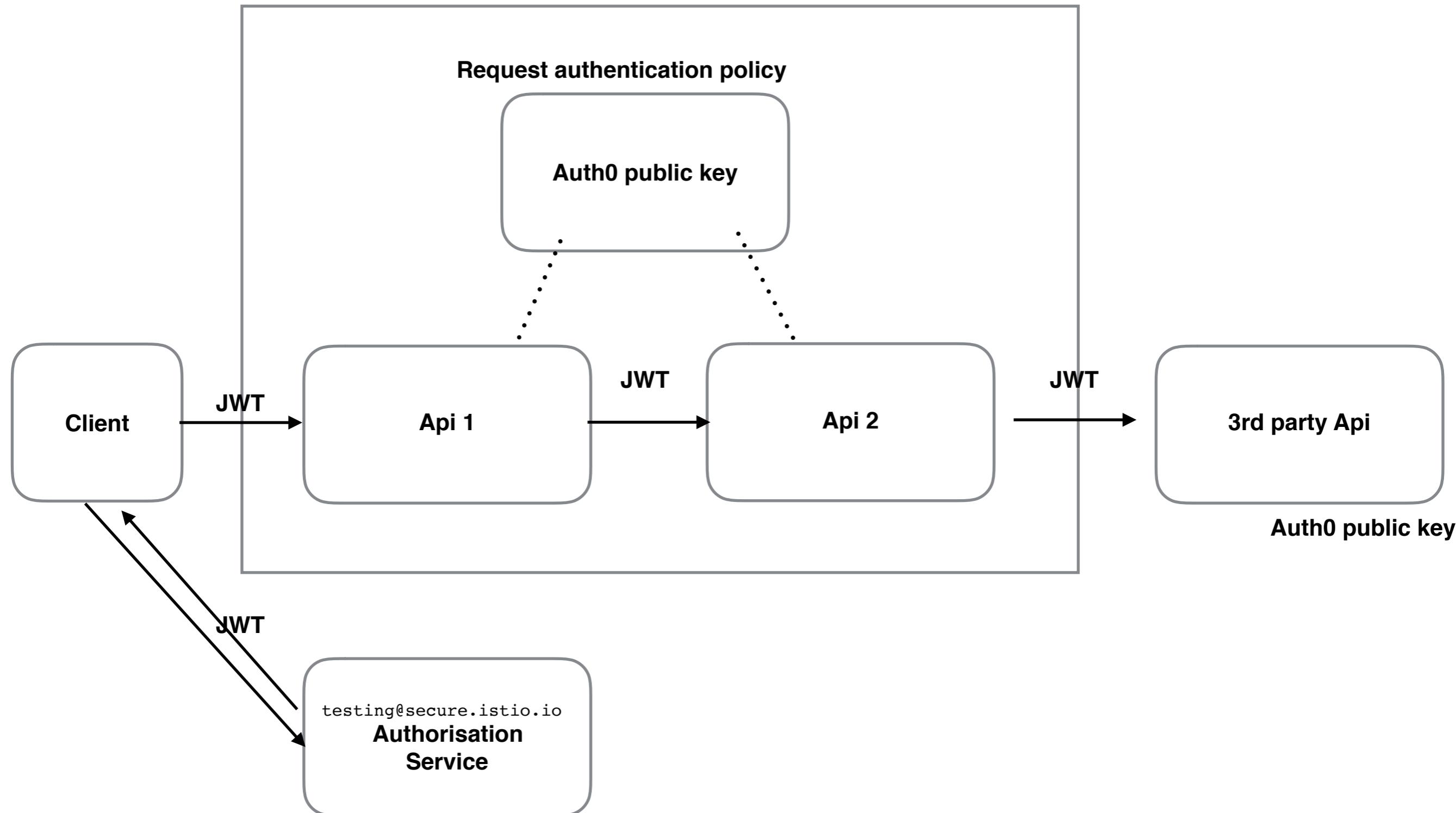
→ Traffic for Authentication & Generating JWT Token

→ Traffic for Validating JWT Token

→ Traffic for Serving Client Request



## # Oath 2



- Applications run in *containers*, which in turn run inside of *Pods*.
- All Pods in your Kubernetes cluster have their own IP address and are attached to the same *network*.
- This means all Pods can talk directly to all other Pods.
- However, Pods are unreliable and come and go as scaling operations, rolling updates, rollbacks, failures and other events occur

- Each pod has its own IP address. And each pod thinks it has a totally normal ethernet device called `eth0` to make network requests through.
- When a pod makes a request to the IP address of another node, it makes that request through its own `eth0` interface. This tunnels to the node's respective virtual `vethX` interface.
- A network bridge connects two networks together. When a request hits the bridge, the bridge asks all the connected pods if they have the right IP address to handle the original request.
- In Kubernetes, this bridge is called `cbr0`. Every pod on a node is part of the bridge, and the bridge connects all pods on the same node together.
-

- when the network bridge asks all the pods if they have the right IP address, none of them will say yes.
- After that, the bridge falls back to the default gateway. This goes up to the cluster level and looks for the IP address.
- At the cluster level, there's a table that maps IP address ranges to various nodes. Pods on those nodes will have been assigned IP addresses from those ranges.
- For example, Kubernetes might give pods on node 1 addresses like 100.96.1.1, 100.96.1.2, etc. And Kubernetes gives pods on node 2 addresses like 100.96.2.1, 100.96.2.2, and so on.
- Then this table will store the fact that IP addresses that look like 100.96.1.xxx should go to node 1, and addresses like 100.96.2.xxx need to go to node 2.

- A Service is bound to a ClusterIP, which is a virtual IP address, and no matter what happens to the backend Pods, the ClusterIP never changes, so a client can always send requests to the ClusterIP of the Service.
- A Kubernetes Service object creates a stable network endpoint that sits in front of a set of Pods and load-balances traffic across them.
- You always put a Service in front of a set of Pods that do the same job. For example, you could put a Service in front of your web front-end Pods, and another in front of your authentication Pods. You never put a Service in front of Pods that do different jobs.
- Every service in a cluster is assigned a domain name like my-service.my-namespace.svc.cluster.local.
- when a request is made to a service via its domain name, the DNS service resolves it to the IP address of the service.
- Then kube-proxy converts that service's IP address into a pod IP address.

- You POST a new Service definition to the API Server
- The Service is allocated a ClusterIP (virtual IP address) and persisted to the cluster store
- The cluster's DNS service notices the new Service and creates the necessary DNS A records

```
apiVersion: v1
kind: Service
metadata:
 name: web-svc
labels:
 blog: svc-discovery
spec:
 type: LoadBalancer
```

It's important to understand that the name registered with DNS is the value of **metadata.name** and that the ClusterIP is dynamically assigned by Kubernetes.

- With service ClusterIP and Kubernetes DNS, service can be easily reached inside a cluster
- If you want more advanced features, such as flexible routing rules, more options for LB, reliable service communication, metrics collection and distributed tracing, etc., then you will need to consider Istio.
- The communication between services is no longer through Kube-proxy but through Istio's sidecar proxies. Istio sidecar proxy works just like Kube-proxy.

- ClusterIP is only reachable inside a Kubernetes cluster, but what if we need to access some services from outside of the cluster?
- With NodePort, Kubernetes creates a port for a Service on the host, which allows access to the service from the node network.
-

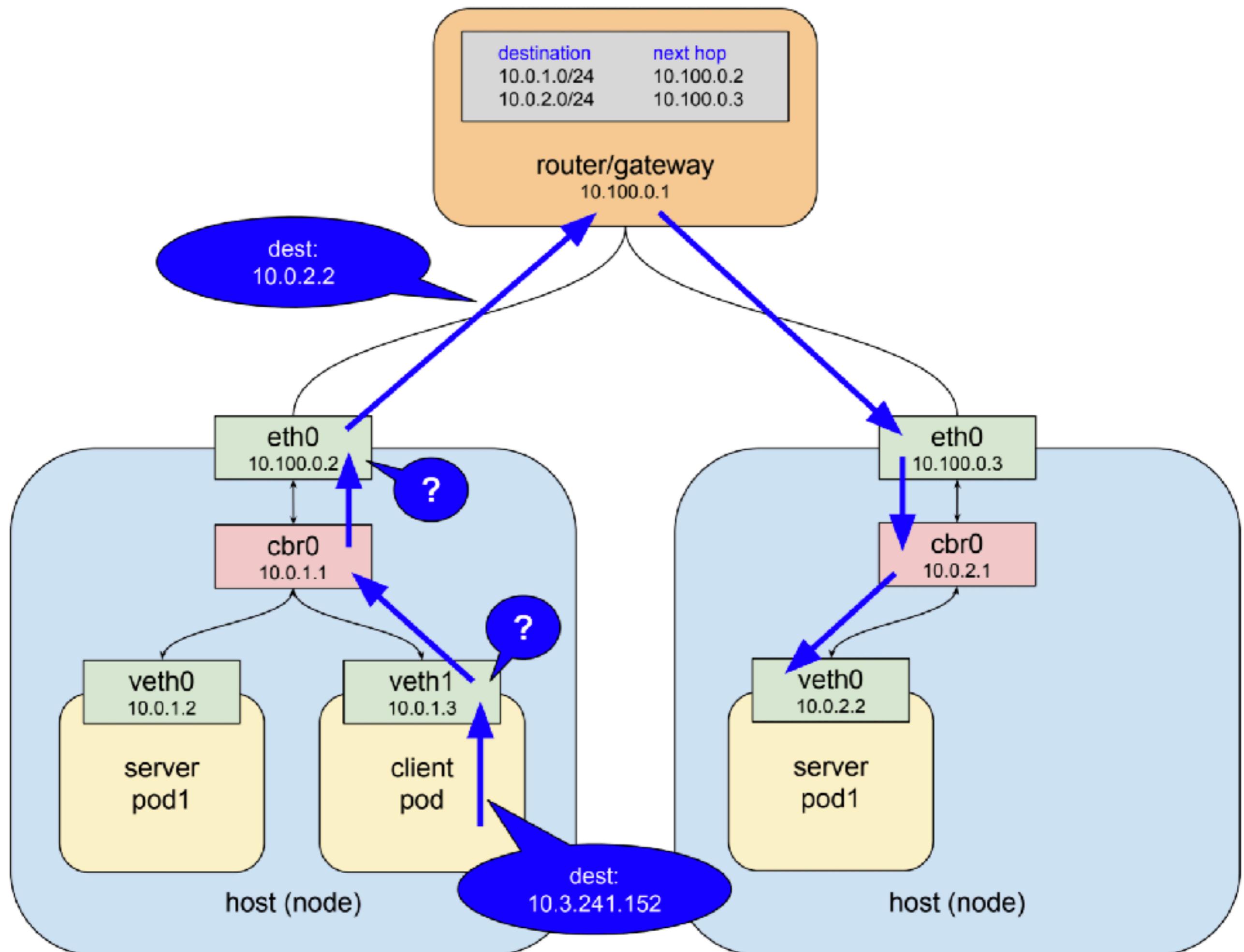
- NodePort is a convenient tool for testing in your local Kubernetes cluster, but it's not suitable for production because of these limitations.
- A single node is a single point of failure for the system. Once the node is down, clients can't access the cluster any more.
- A service can be declared as LoadBalancer type to create a layer 4 load balancer in front of multiple nodes. As this layer 4 load balancer is outside of the Kubernetes network, a Cloud Provider Controller is needed for its provision. This Cloud Provider Controller watches the Kubernetes master for the addition and removal of Service resources and configures a layer 4 load balancer in the cloud provider network to proxy the NodePorts on multiple Kubernetes nodes.

if we need to expose multiple services to the outside of a cluster, we must create a LoadBalancer for each service. However, creating multiple LoadBalancers can cause some problems:

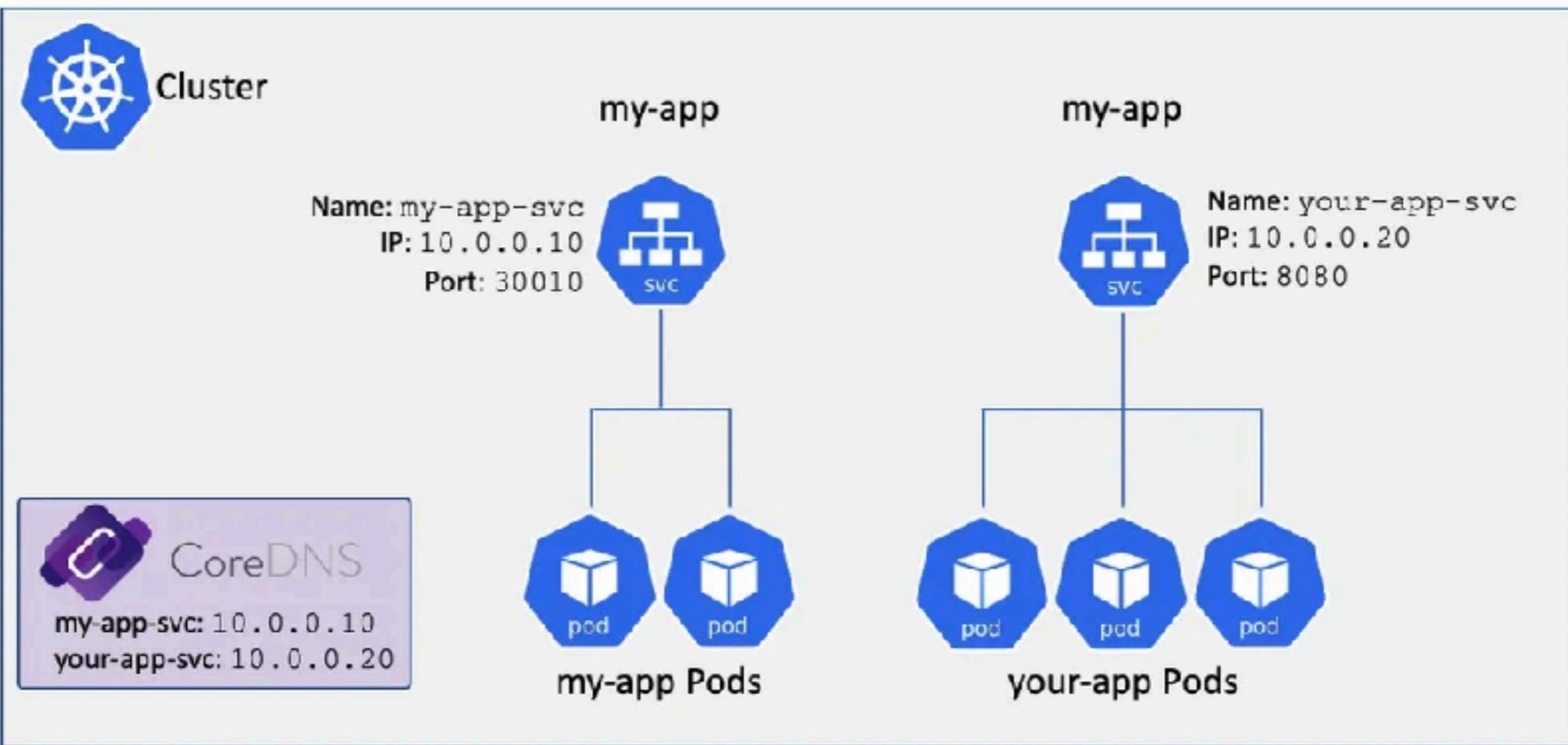
- Needs more public IPs, which normally are limited resources.
- Introduces coupling between the client and the server, making it hard to adjust your backend services when business requirements change.

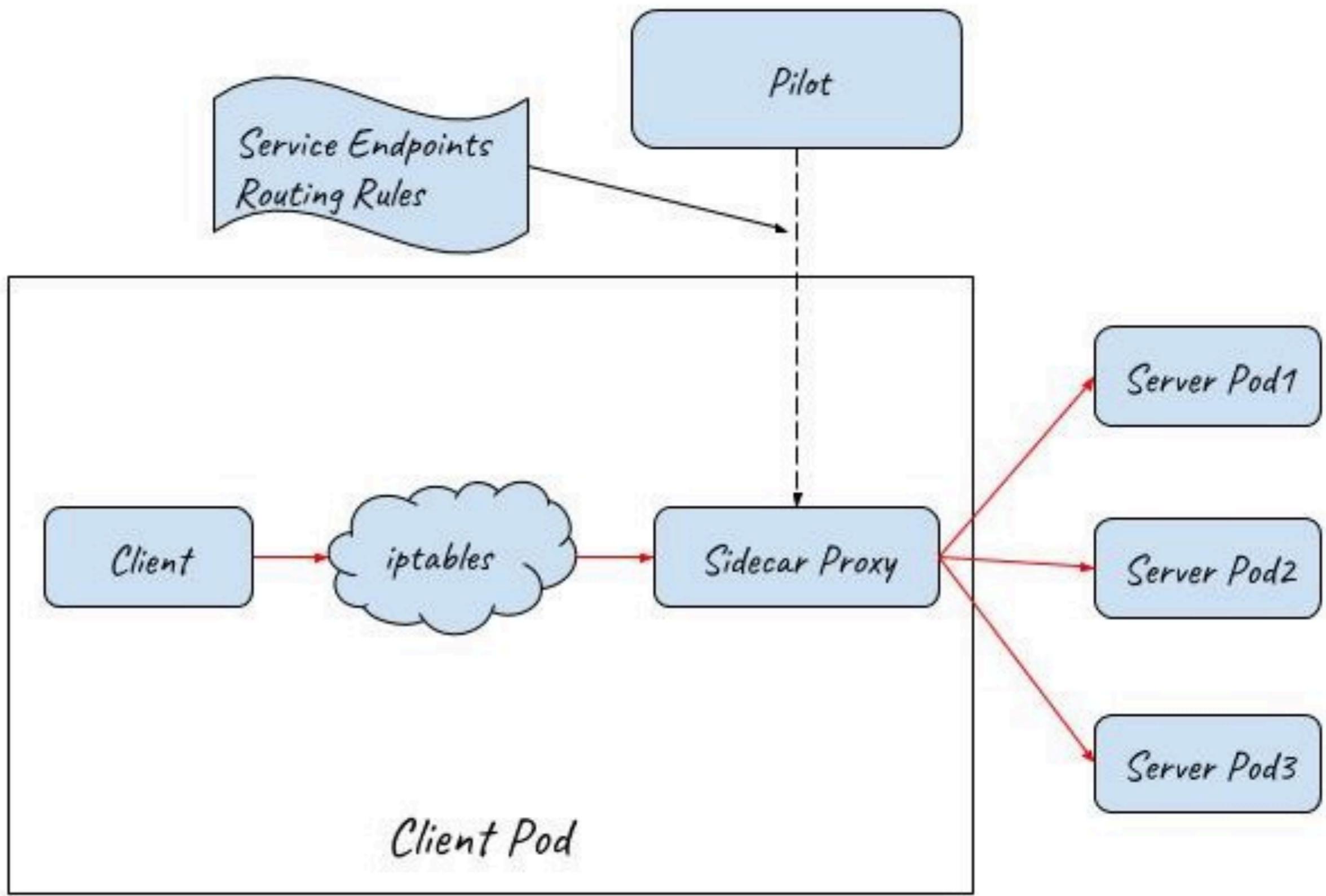
Kubernetes Ingress resource is used to declare an OSI layer 7 load balancer, which can understand HTTP protocol and dispatch inbound traffic based on the HTTP URL or Host.

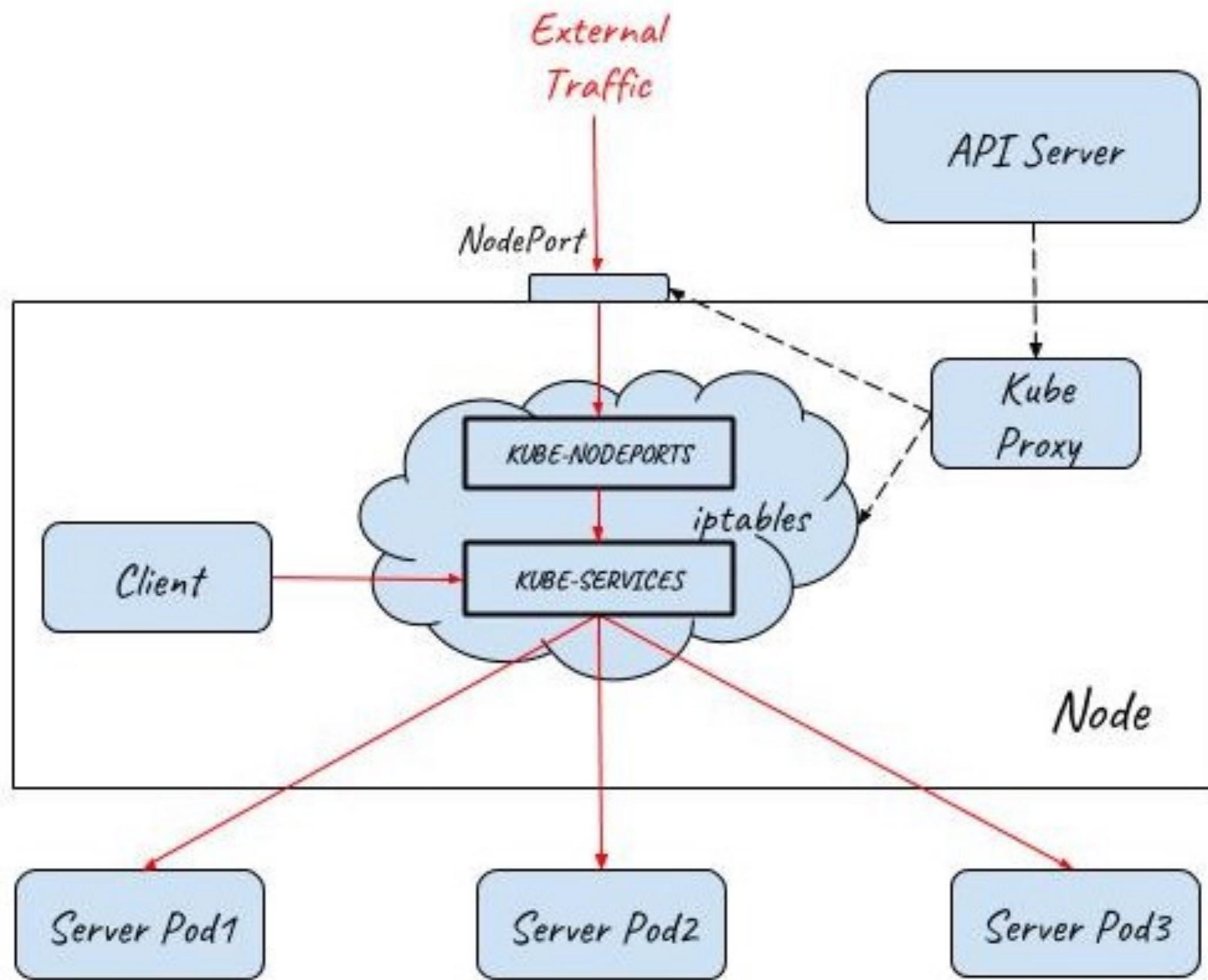
- Ingress controller itself is also deployed as Pods inside the cluster.
- Ingress controller provides a unified entrance for the HTTP services in a cluster, but it can't be accessed directly from outside because the ingress controller itself is also deployed as Pods inside the cluster.
- Ingress controller must work together with NodePort and LoadBalancer to provide the full path for the external traffic to enter the cluster.

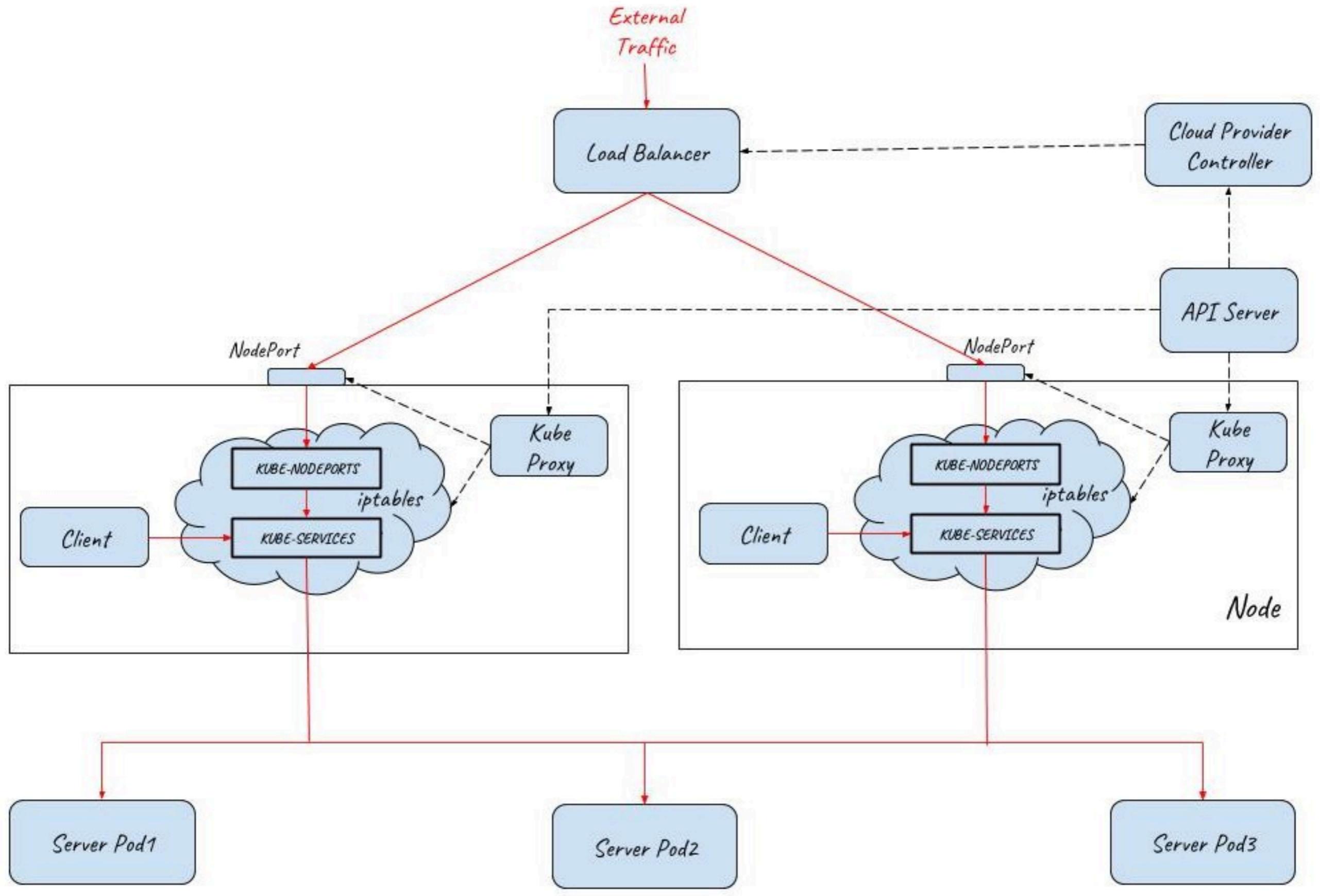


|                 | <b>OSI Layer</b>               | <b>TCP/IP</b>                                                               | <b>Datagrams are called</b> |
|-----------------|--------------------------------|-----------------------------------------------------------------------------|-----------------------------|
| <b>Software</b> | <b>Layer 7</b><br>Application  | HTTP, SMTP, IMAP, SNMP, POP3, FTP                                           | <b>Upper Layer Data</b>     |
|                 | <b>Layer 6</b><br>Presentation | ASCII Characters, MPEG, SSL, TSL, Compression (Encryption & Decryption)     |                             |
|                 | <b>Layer 5</b><br>Session      | NetBIOS, SAP, Handshaking connection                                        |                             |
|                 | <b>Layer 4</b><br>Transport    | TCP, UDP                                                                    | <b>Segment</b>              |
|                 | <b>Layer 3</b><br>Network      | IPv4, IPv6, ICMP, <u>IPSec</u> , MPLS, ARP                                  | <b>Packet</b>               |
| <b>Hardware</b> | <b>Layer 2</b><br>Data Link    | Ethernet, 802.1x, PPP, ATM, <u>Fiber</u> Channel, MPLS, FDDI, MAC Addresses | <b>Frame</b>                |
|                 | <b>Layer 1</b><br>Physical     | Cables, Connectors, Hubs (DLS, RS232, 10BaseT, 100BaseTX, ISDN, T1)         | <b>Bits</b>                 |

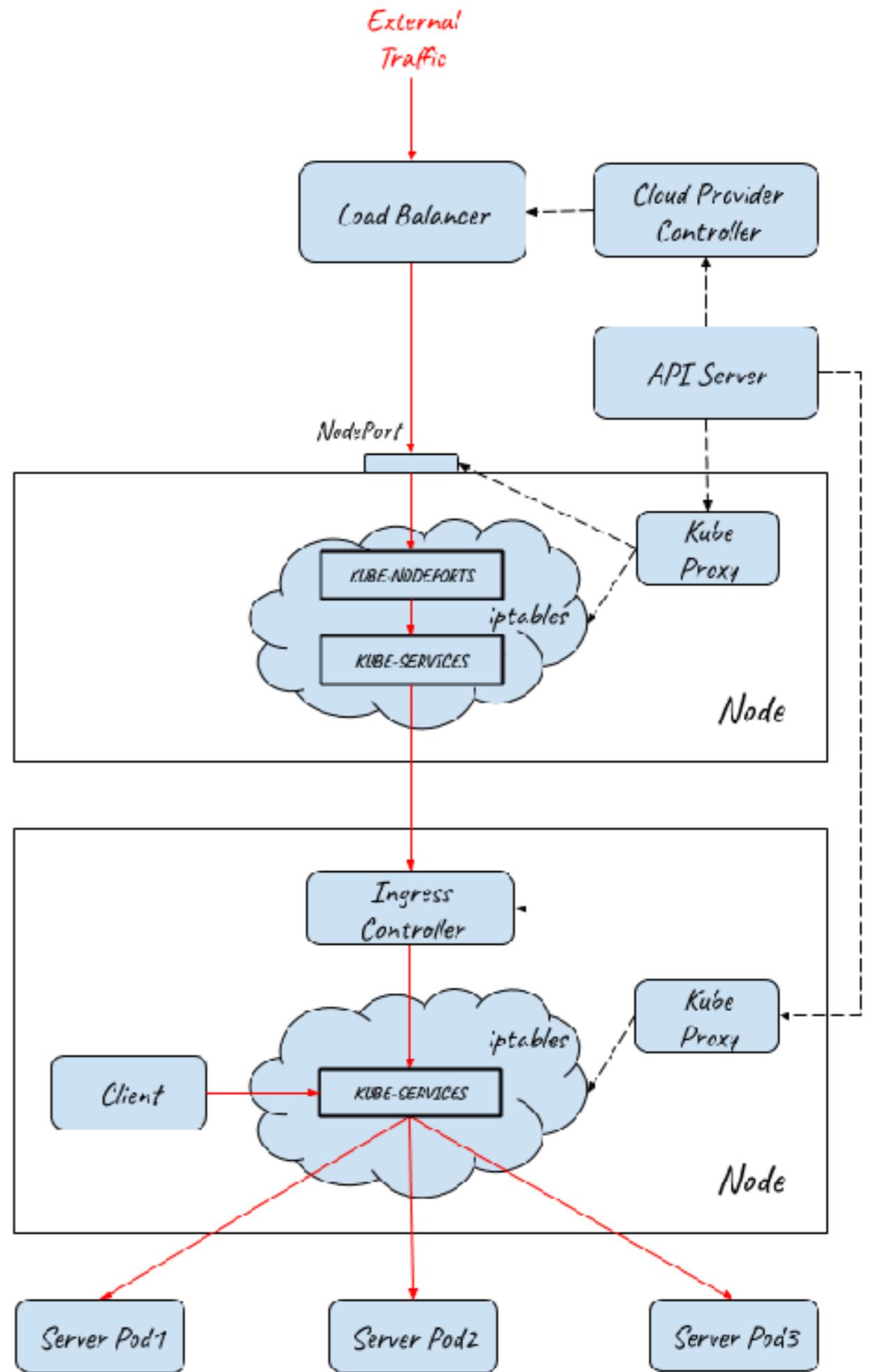








1. Internet/External traffic reaches the layer 4 load balancer.
2. Load balancer dispatches traffic to multiple NodePorts on the Kubernetes.
3. Traffic is captured by iptables and redirected to ingress controller Pods.
4. Ingress controller sends traffic to different Services according to ingress rules.
5. Finally, traffic is redirected to the backend Pods by iptables.



### Client Request

- Load Balancer(External IP)
- Load Balancer (Node IP)
- Ingress Controller Service(ClusterIP)
- Ingress Controller Pod(Pod IP)
- Backend Service(ClusterIP)
- Backend Pod(Pod IP)

A searchable catalog of Customers

A searchable catalog of Freelancer

A searchable catalog of Projects

Timesheets for the Freelancers

Make Payments to the Freelancers

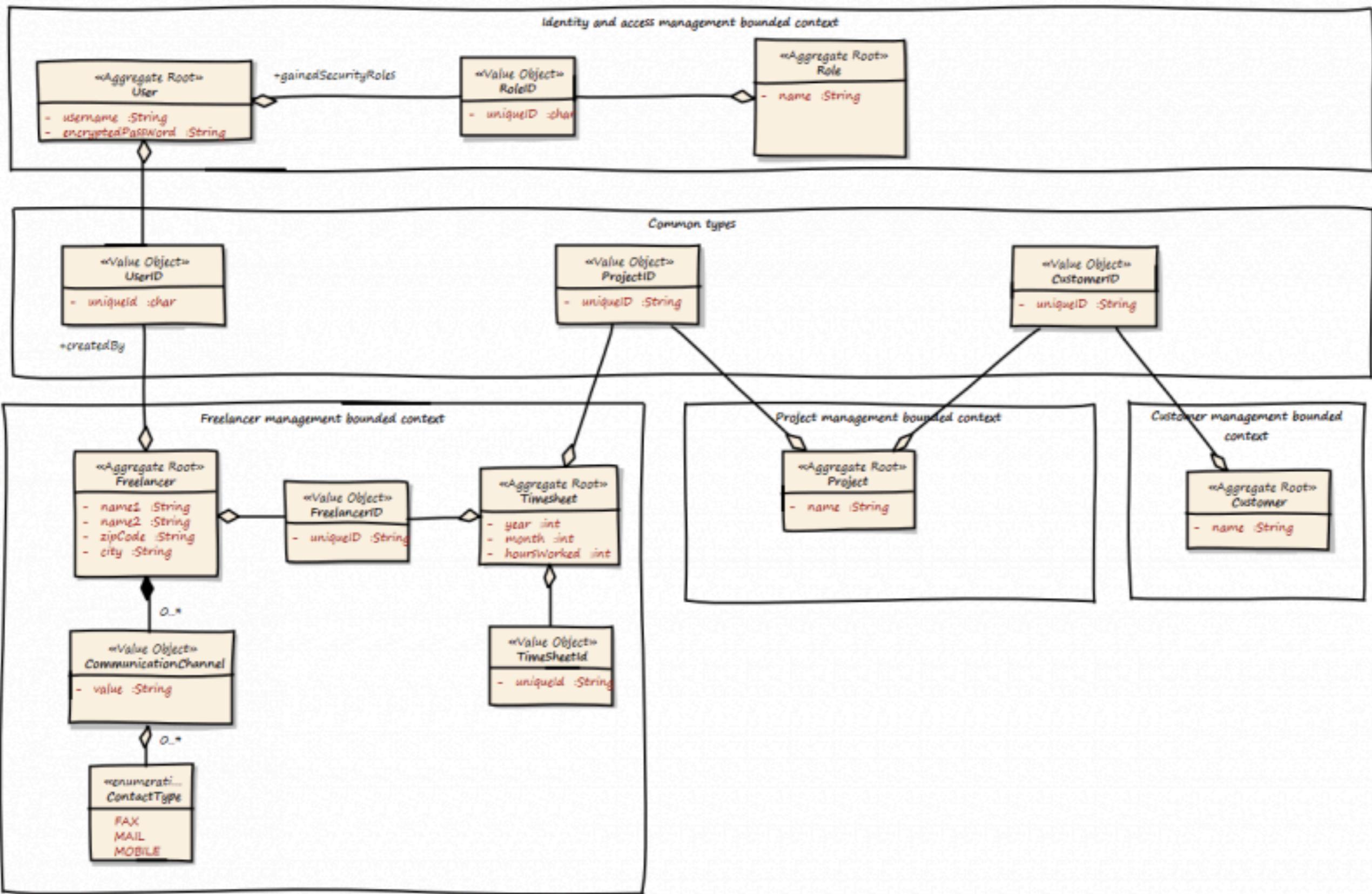
## Body Leasing Domain

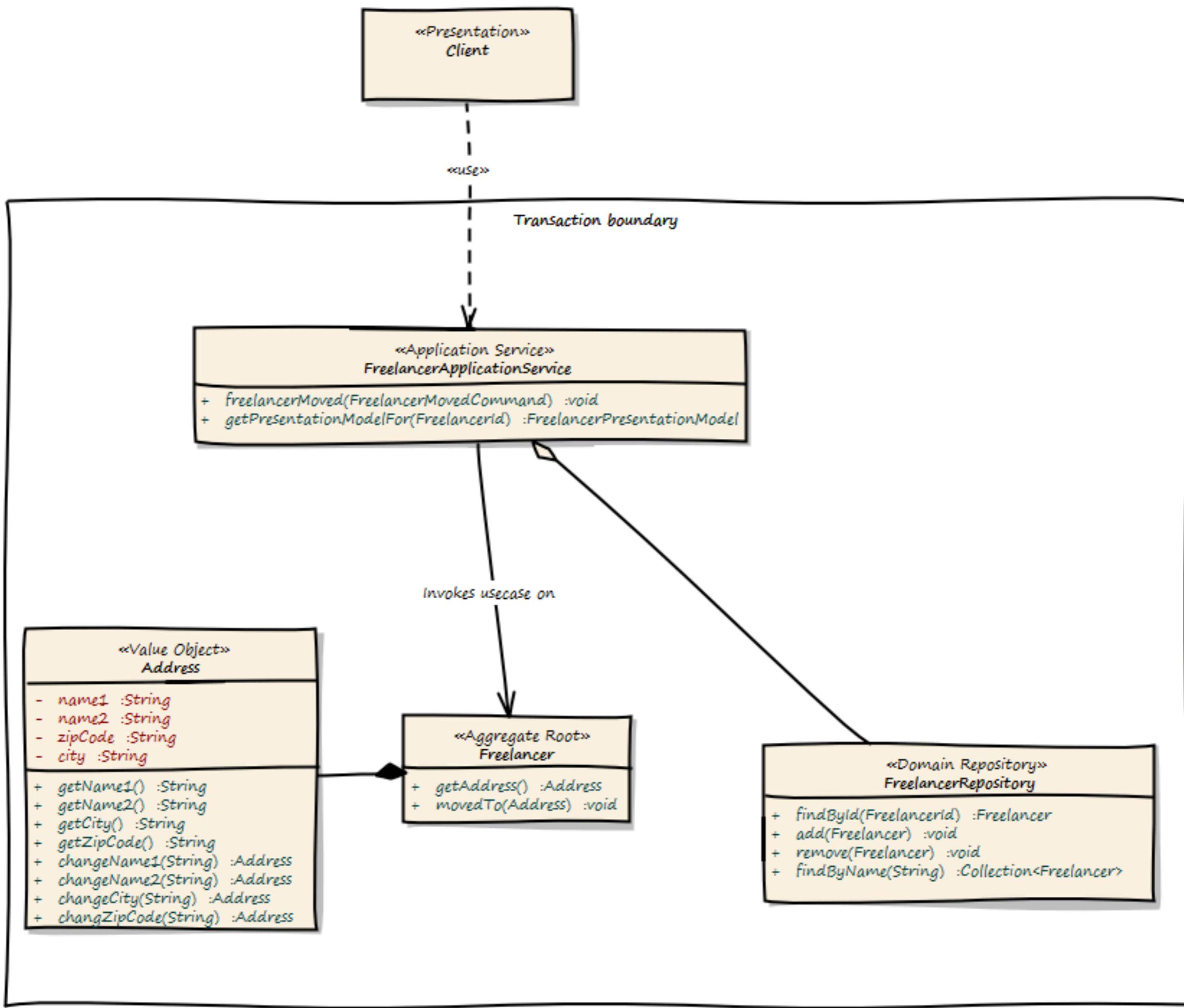
Identity and  
Access  
Management  
Subdomain

Freelancer  
Management  
Subdomain

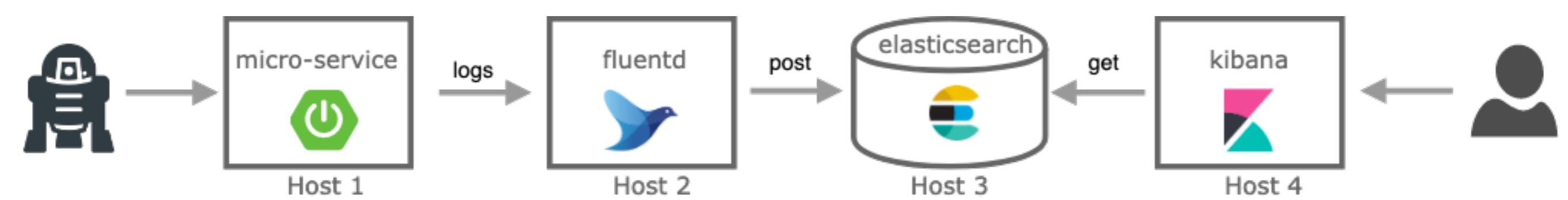
Customer  
Management  
Subdomain

Project  
Management  
Subdomain





<https://www.mirkosertic.de/blog/2013/04/domain-driven-design-example/>



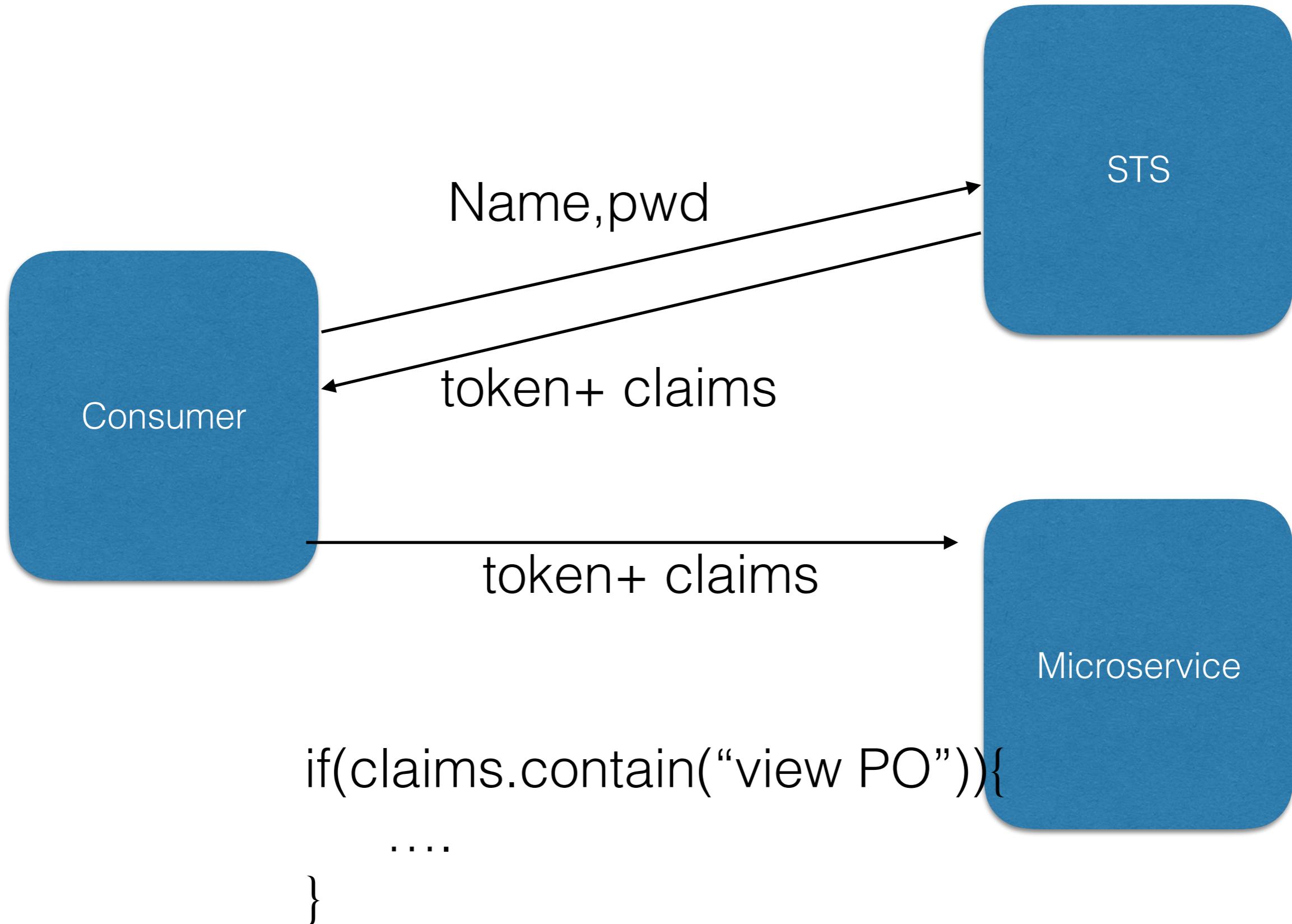
- AAA
- Authentication
- Authorization
- Audit
- Asset Handling (In Rest and Transit)
- Exception Handling
- Session management
- Input Validation
- Key
- STRIDE
  - Spoofing
  - Tampering
  - Repudation
  - Information Disclosure
  - Deniel of Service
  - Elevation of Privilege

|                      |                             |
|----------------------|-----------------------------|
| • Authentication     | Opened Connect + OAuth2     |
| • Authorization      | Claim based                 |
| • Audit              | Event Store, Event Sourcing |
| • Asset in Transit   | HTTPS/ WSS                  |
| • Asset in Rest      | Storage Encryption          |
| • Exception Handling | Fwk                         |
| • Session management | Fwk                         |
| • Input Validation   | myCode (fwk) , WAF          |
| • Key                | Vault                       |

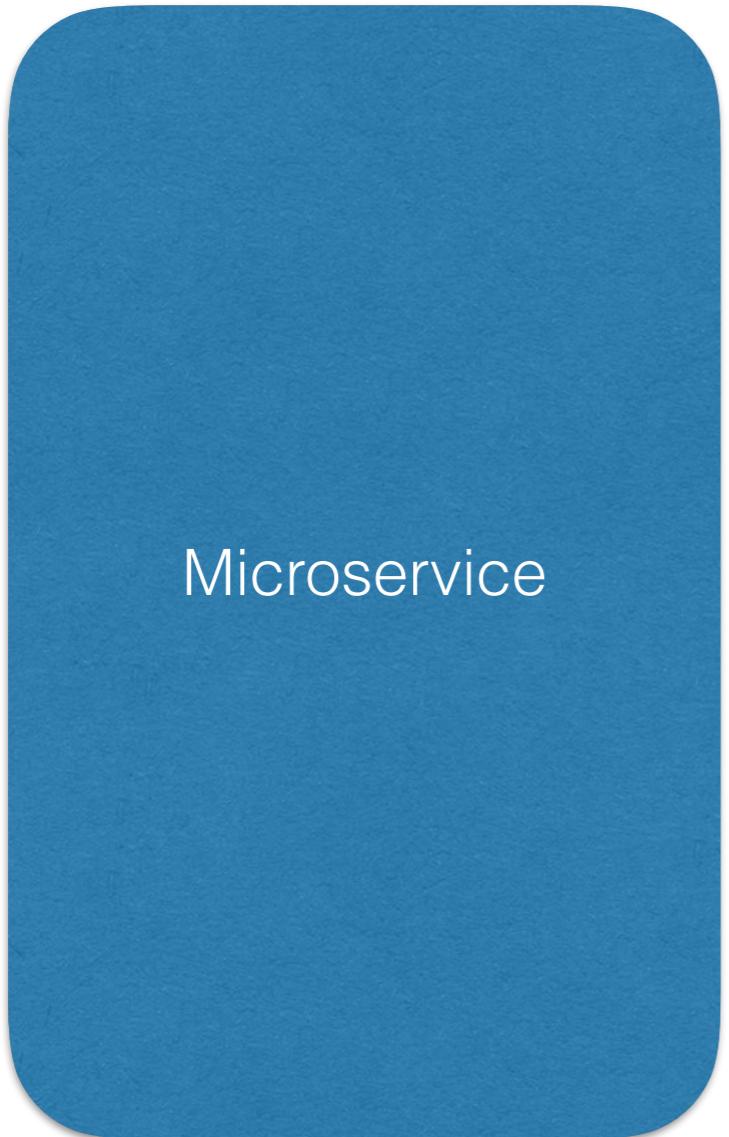
- By what you know: password, secret <— \*\*\*
- By What you have : otp, mail, cert, rsa <—
- By What you Are : Bio (fingerprint, voice, retina, ...)
- Your Location:
- Behaviourial
- 2 factor
- multifactor

- By what you know
- Custom Credential Store (not recommended)
- Oauth2 (...)
- OpenIDconnect
- SAML

Claim mapping  
Microservice1-> Purchase->c1,c2,c3



token (userid)



Microservice

```
if(role == "Owner"){\n ...\n}
```



userid->role

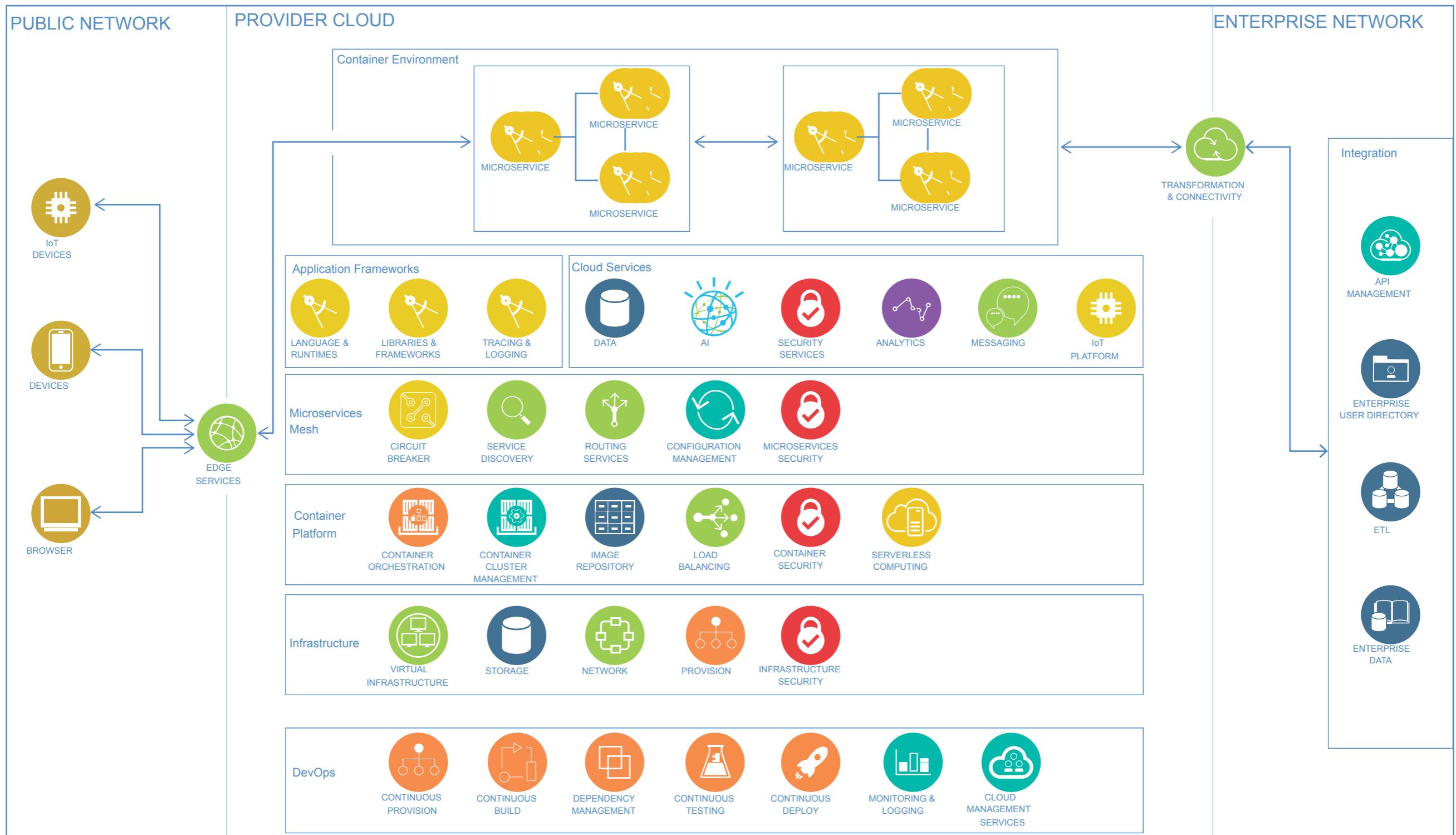
token, (viewPurchase,Delete sales)



Microservice

```
if(viewPurchase){

}
```



# 12 factor App

|                     |                                                                                                                                                                                                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Codebase            | <b><i>One codebase per service, tracked in revision control; many deploys.</i></b>                                                                                                                                                |
| Dependencies        | <b><i>Explicitly declare and isolate dependencies</i></b> <ul style="list-style-type: none"><li>Noncontainerized environments : Chef, Puppet, Ansible,Maven, Gradle,npn</li><li>Containerized environment : Dockerfile.</li></ul> |
| Config              | <b><i>Store configuration in the environment</i></b><br>environment variable, application.properties                                                                                                                              |
| Backing Services    | <b><i>Treat backing services as attached resources</i></b><br>datastores , messaging , SMTP , caching . The current production database could be detached, and the new database attached – all without any code changes.          |
| Build, Release, Run | <b><i>Strictly separate build and run stages</i></b><br>Docker images make it easy to separate the build and run stages.                                                                                                          |
| Processes           | <b><i>Execute the app in one or more stateless processes</i></b><br>Store any stateful data, or data that needs to be shared between instances, in a backing service.                                                             |

|                 |                                                                                                                                                                                                                                                                                                                                    |
|-----------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Data Isolation  | <b><i>Each service manages its own data</i></b>                                                                                                                                                                                                                                                                                    |
| Concurrency     | <b><i>Scale out via the process model</i></b> <p>The <a href="#">Unix process model</a> is largely a predecessor to a true microservices architecture. By leveraging independent deployment feature of microservices, we can individually scale the most needed microservice by using on-demand scaling feature of containers.</p> |
| Disposability   | <b><i>Maximize robustness with fast startup and graceful shutdown</i></b> <p>Instances of a service need <a href="#">to be disposable</a> so they can be started, stopped, and redeployed quickly, and with no loss of data.</p>                                                                                                   |
| Dev/Prod Parity | <b><i>Keep development, staging, and production as similar as possible</i></b>                                                                                                                                                                                                                                                     |
| Logs            | <b><i>Treat logs as event streams</i></b>                                                                                                                                                                                                                                                                                          |
| Admin Processes | <b><i>In a production environment, run <u>administrative and maintenance tasks</u> as a separate microservice.</i></b>                                                                                                                                                                                                             |



# Cloud Deployment



