



- » 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

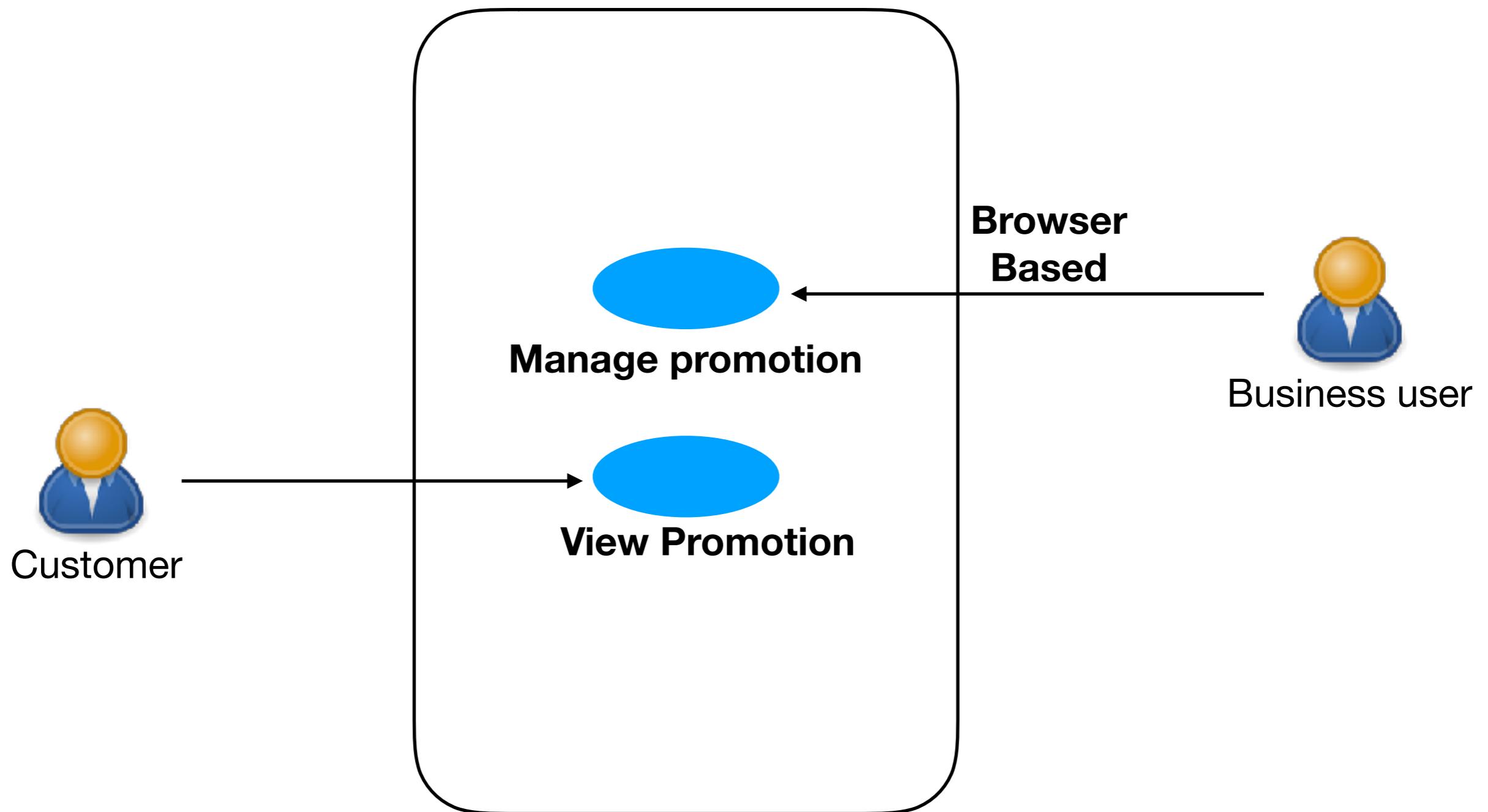
what do
YOU
expect?

- > Your Technology stack
- > Total Years of experience
- > what do you know about microservice ?
- > Docker ?
- > Kubernetes ?

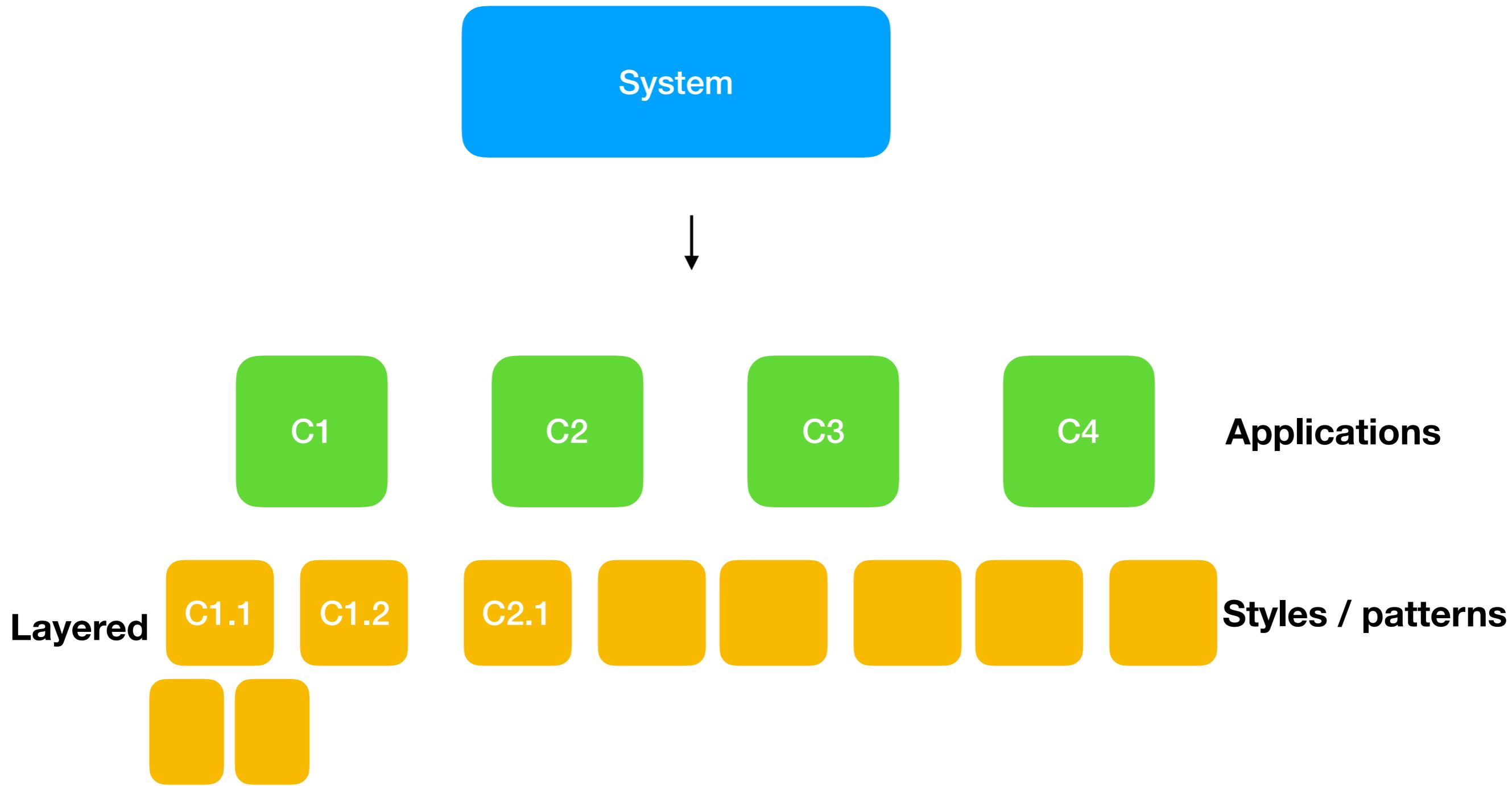
Context view

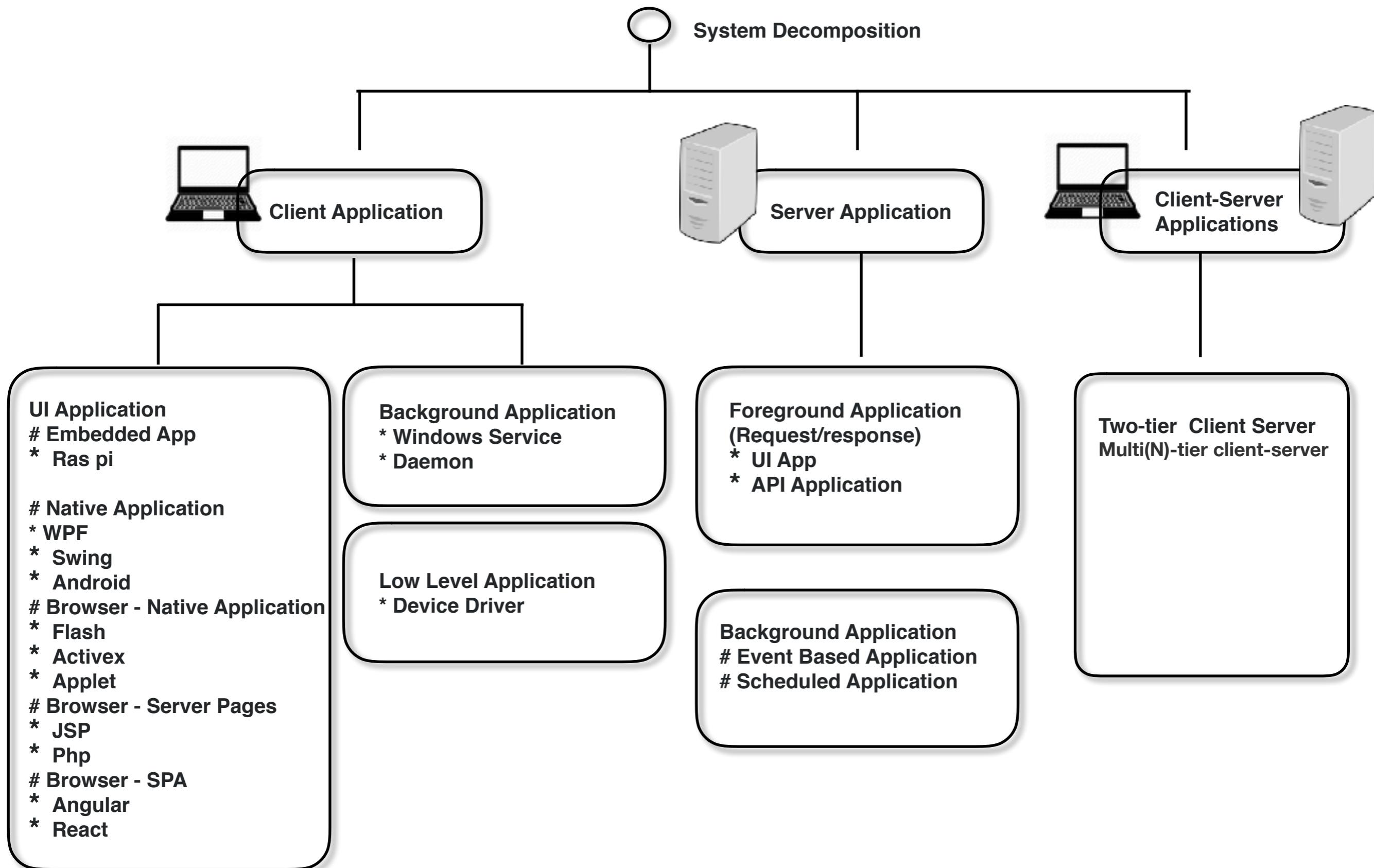


Functional view

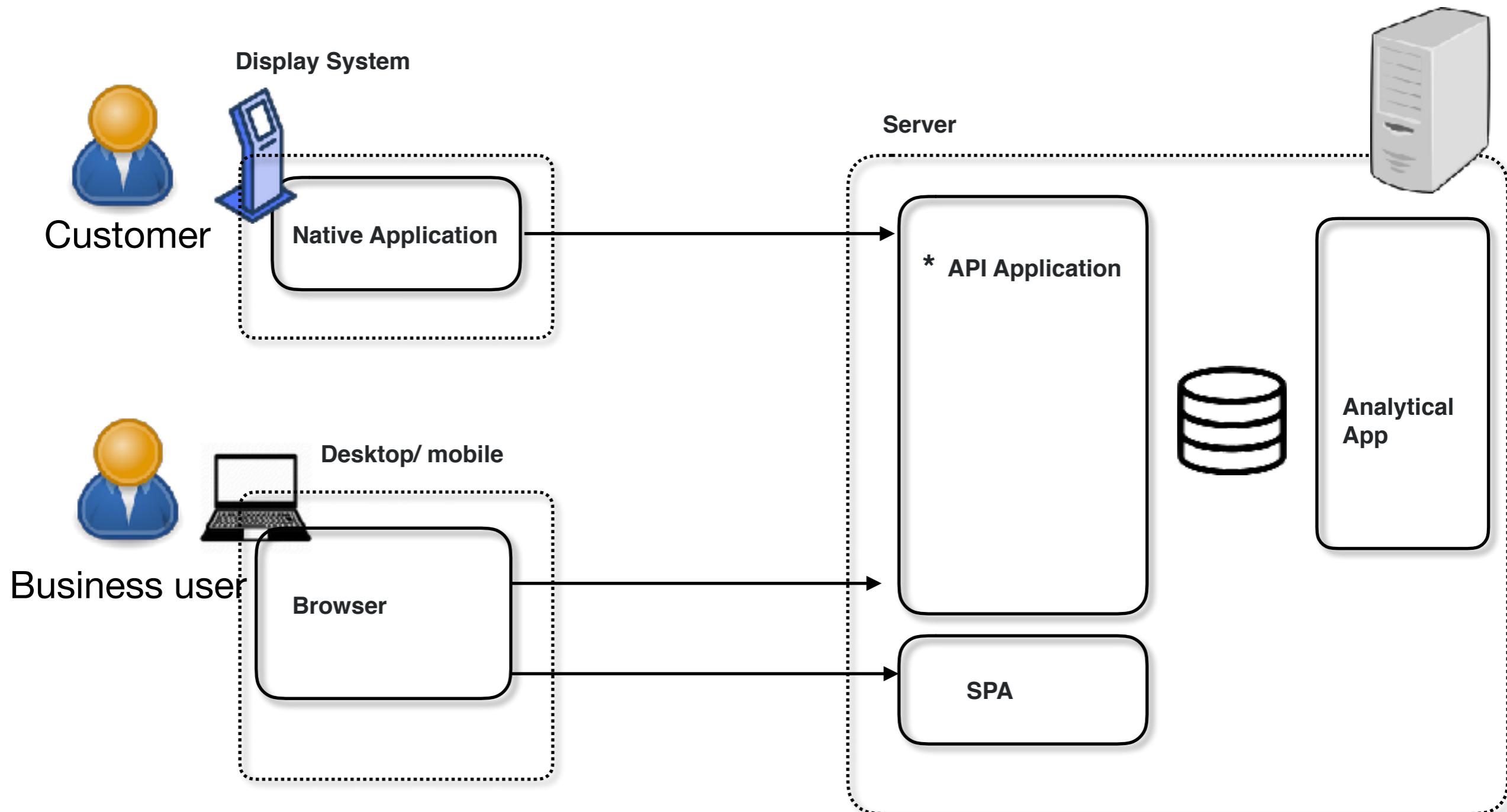


Logical view



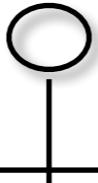


Logical view - todo





System Decomposition



Transformations at
Similar levels of
abstraction

1

Pipes and filter
Batch sequential

subsystems
have similar
structure



Transformations at
Different levels of
abstraction

2

Layered
Hexagonal

subsystems
have different
structure



Decomposition
based on
core and
Variance

3

Micro Kernel
PLA

subsystems
are divided into
Low level and
High level



Interaction
oriented
Decomposition

4

MVC 2
MVP
MVVM

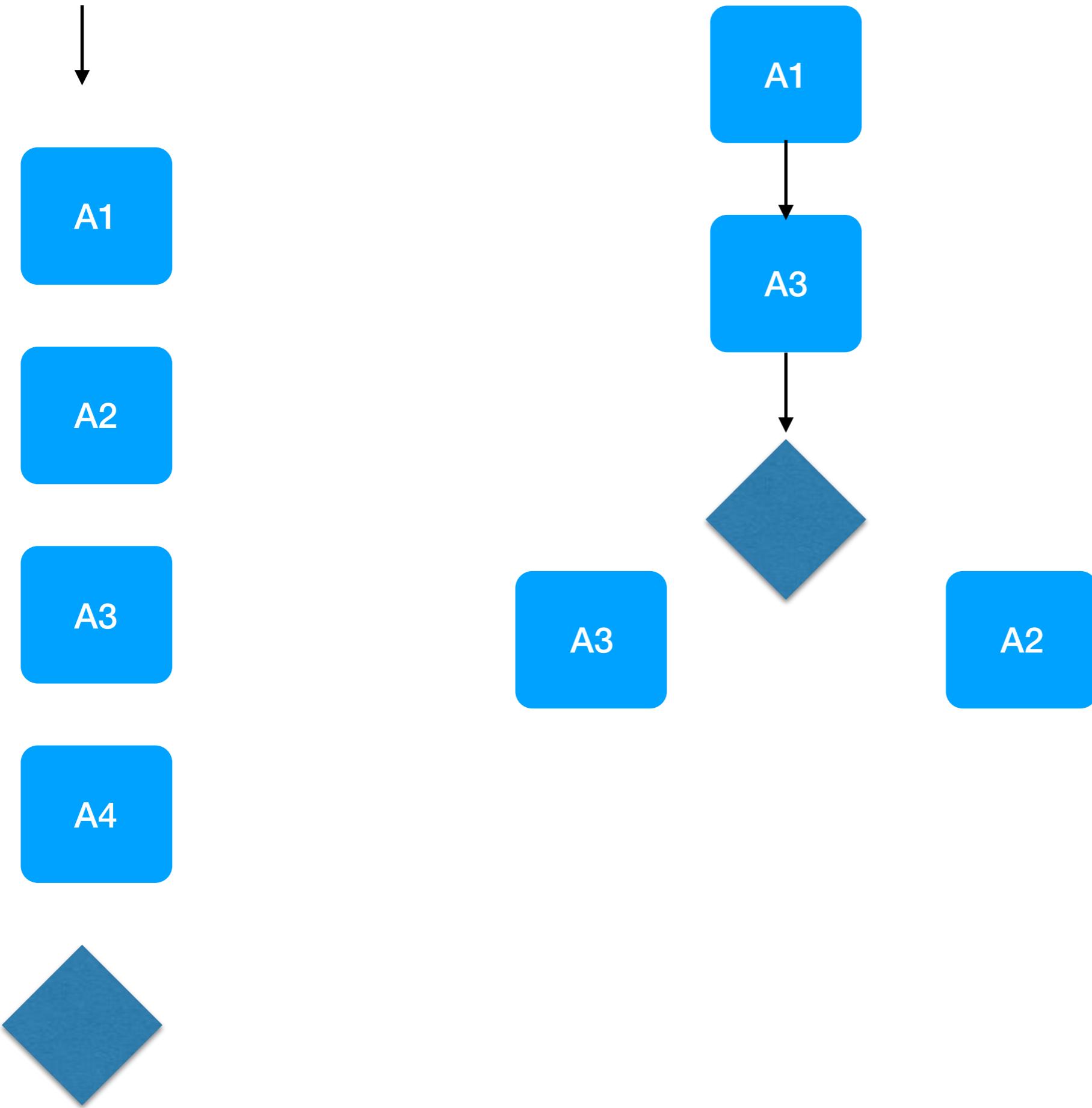


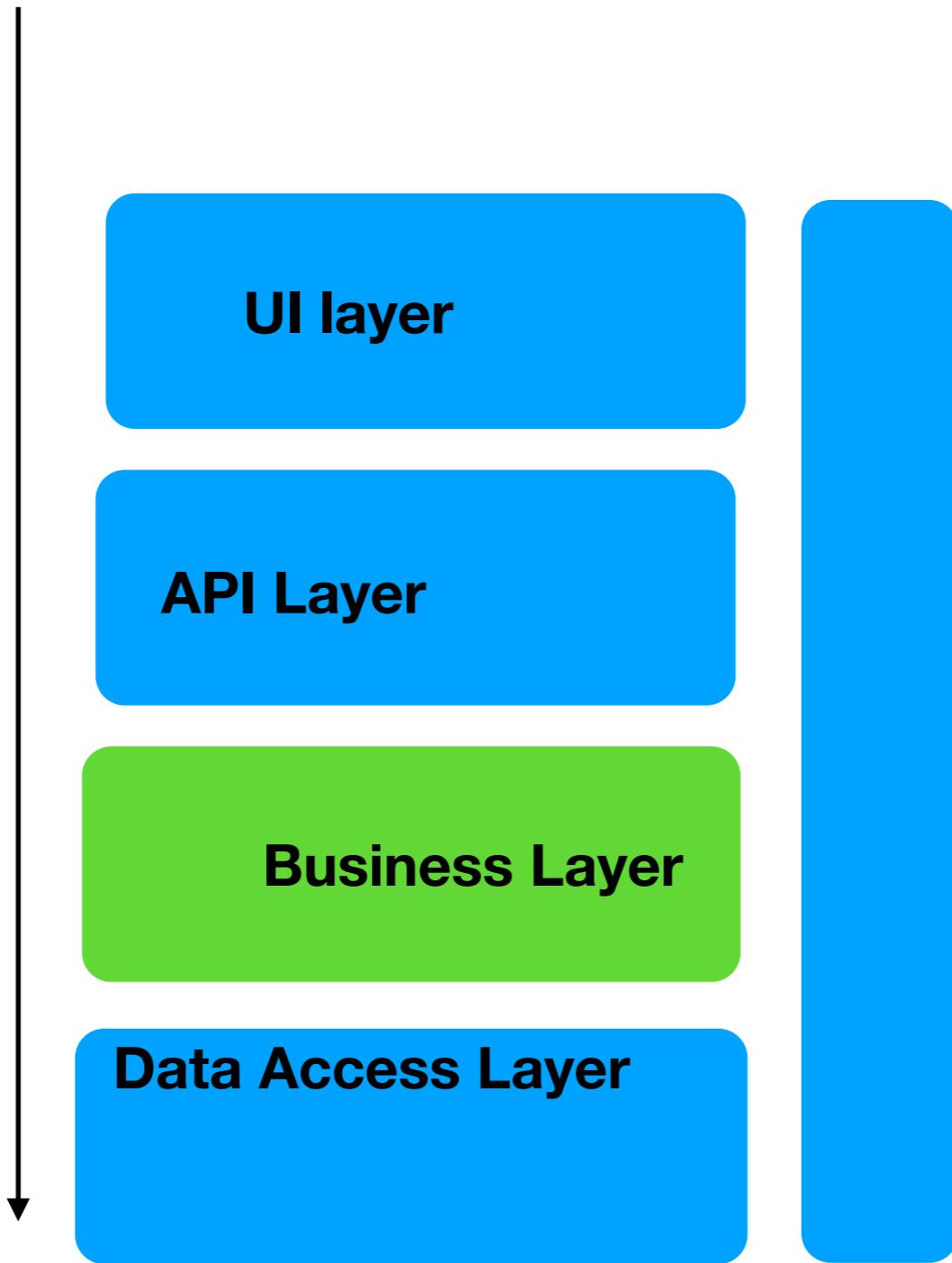
Decompose
based on business
capabilities
(features)

5

Component based
Distributed component
Microservice
SOA







Invoke logic()

Layer 1

f1(int) f2() f3(int,int)

Layer 2

f1(bool) f3(int,int)
f2(double) f4()

Layer 3

f1() f4()
f2(int) f3()
 f5(int)

Layer 4

f1(int) f2()

Process Data()

Filter 1

Pipe

f1(int)

Filter 2

f2(int)

Filter 3

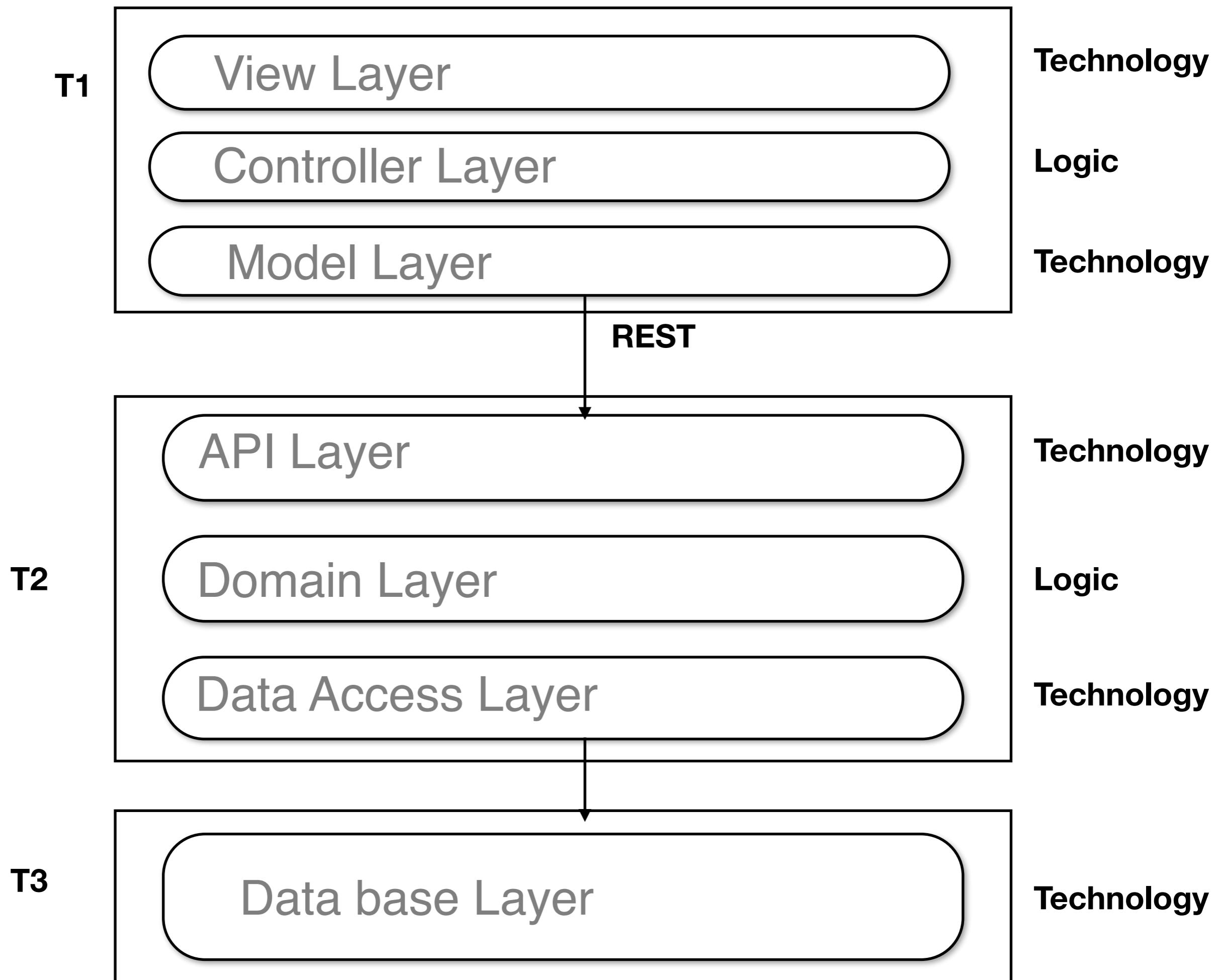
* Can swap filters

f3(int)

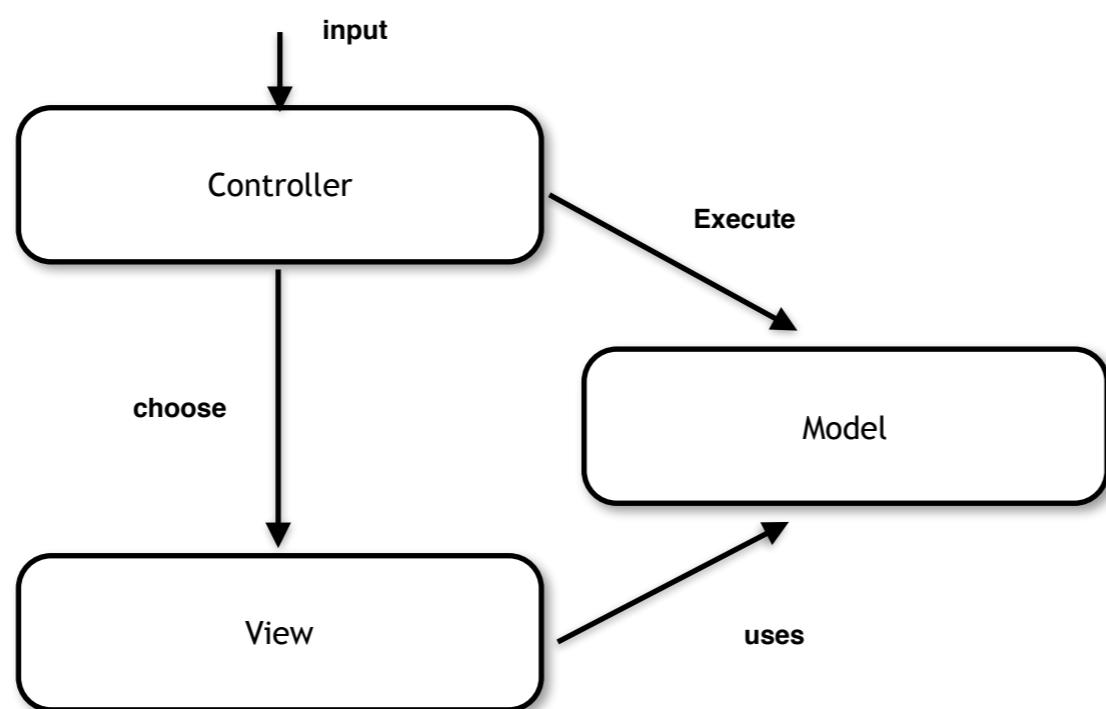
Filter 4

f4(int)

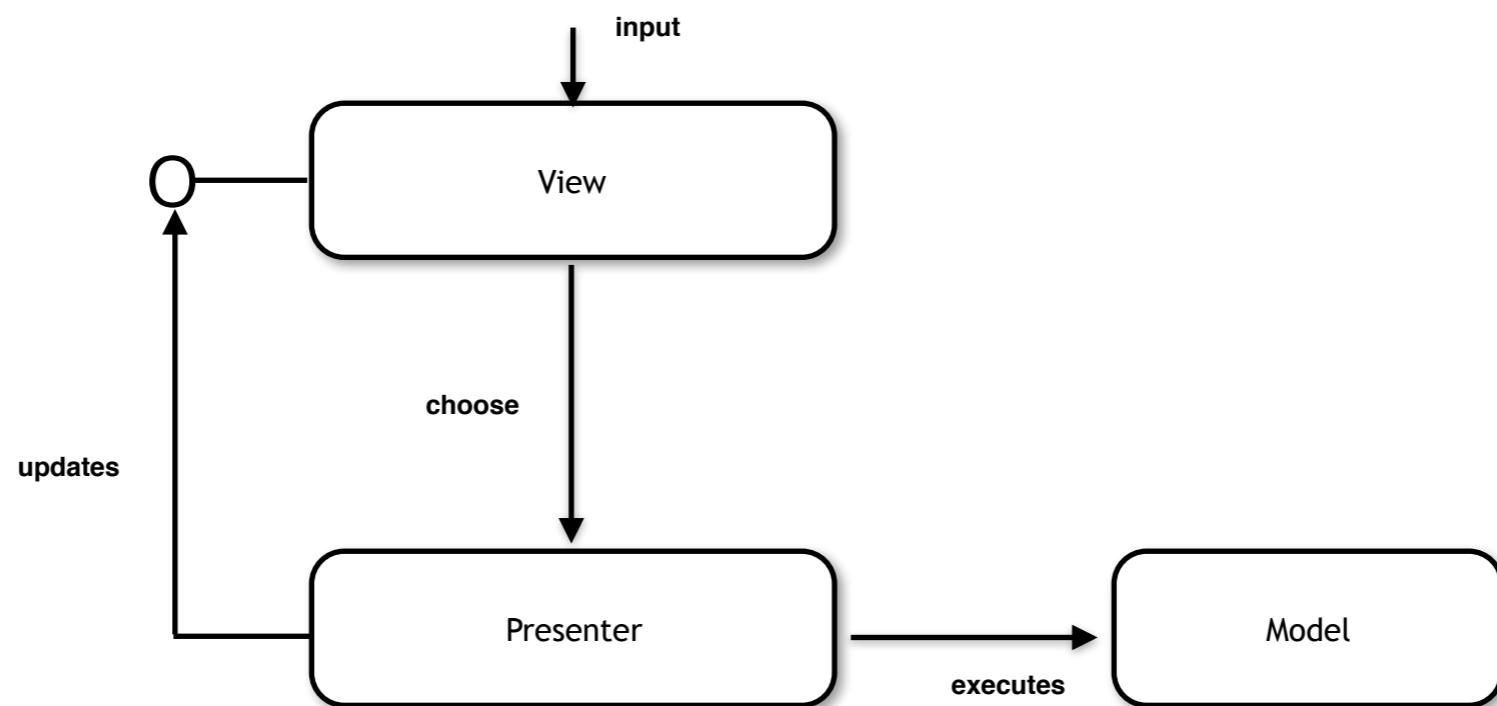
Data



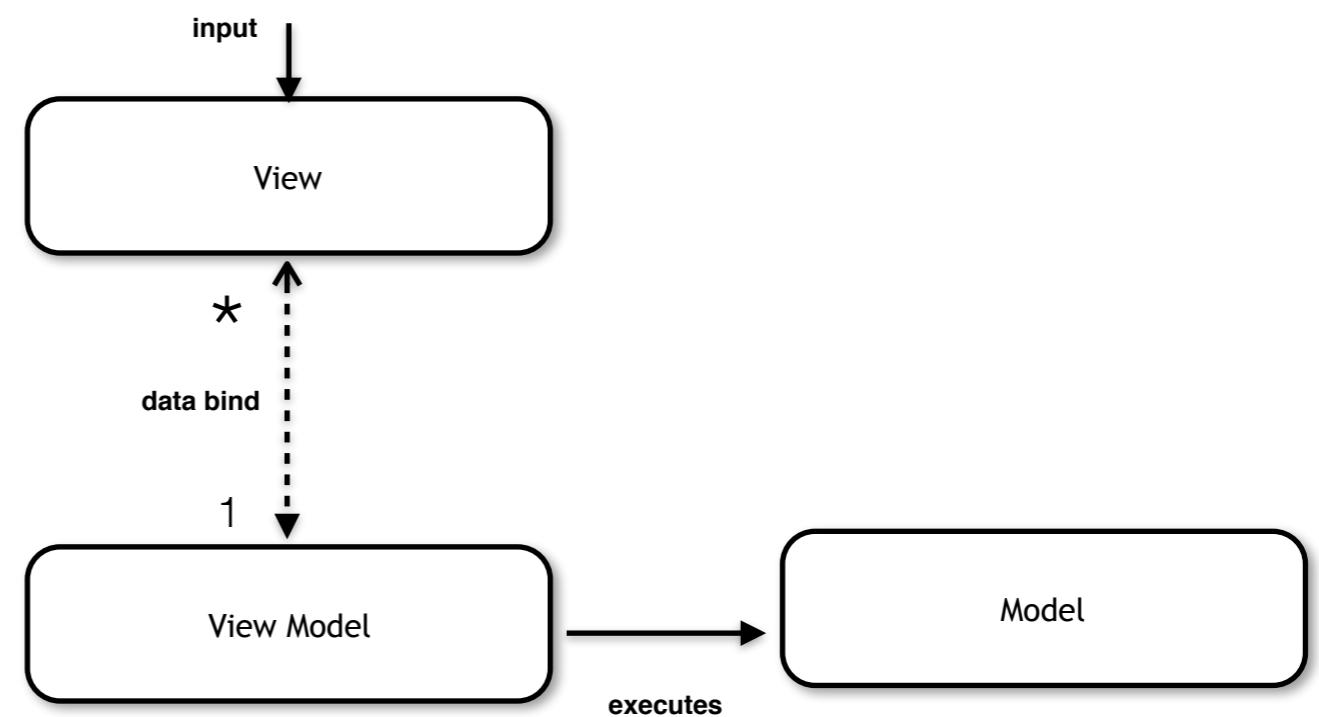
MVC 2



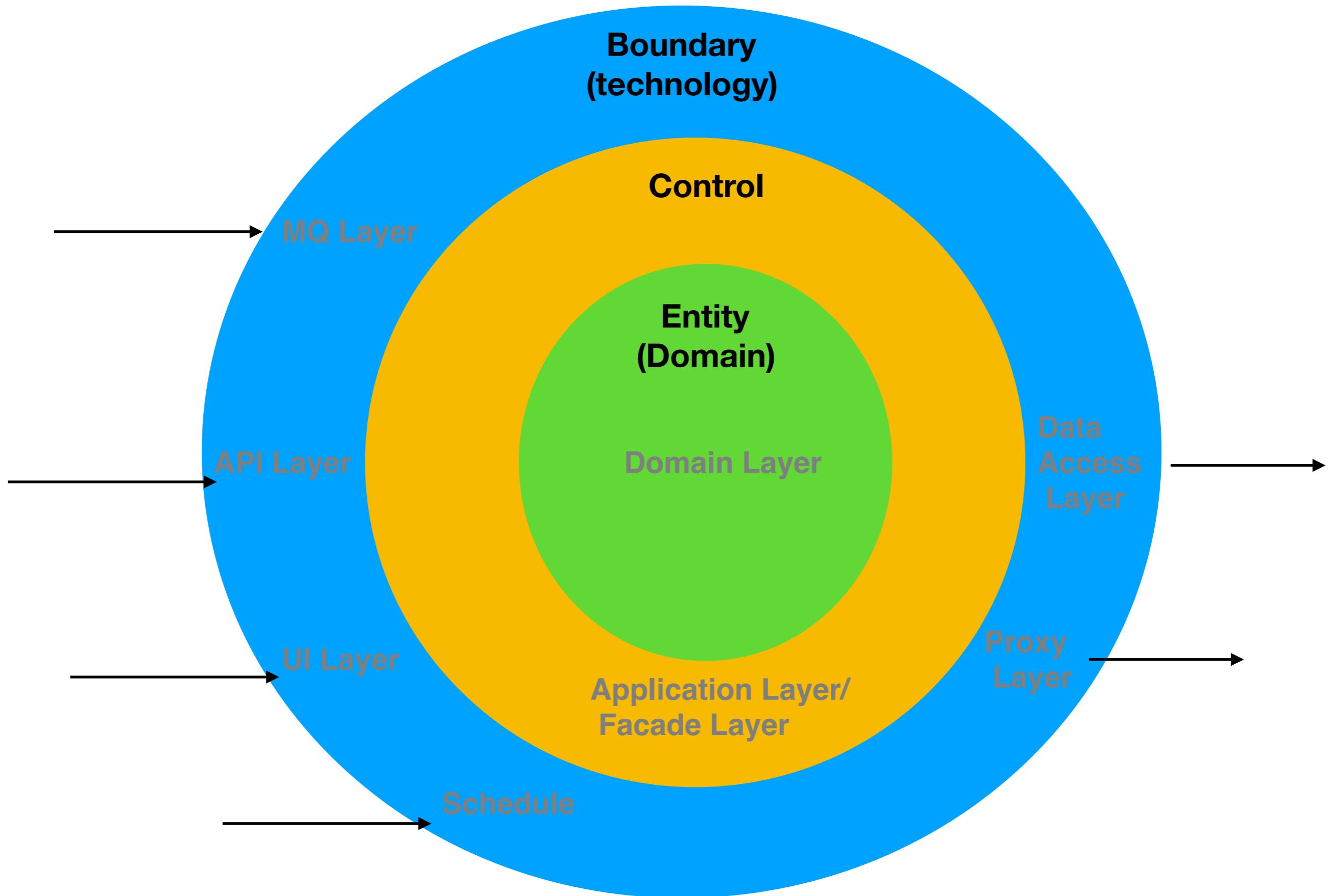
mvp

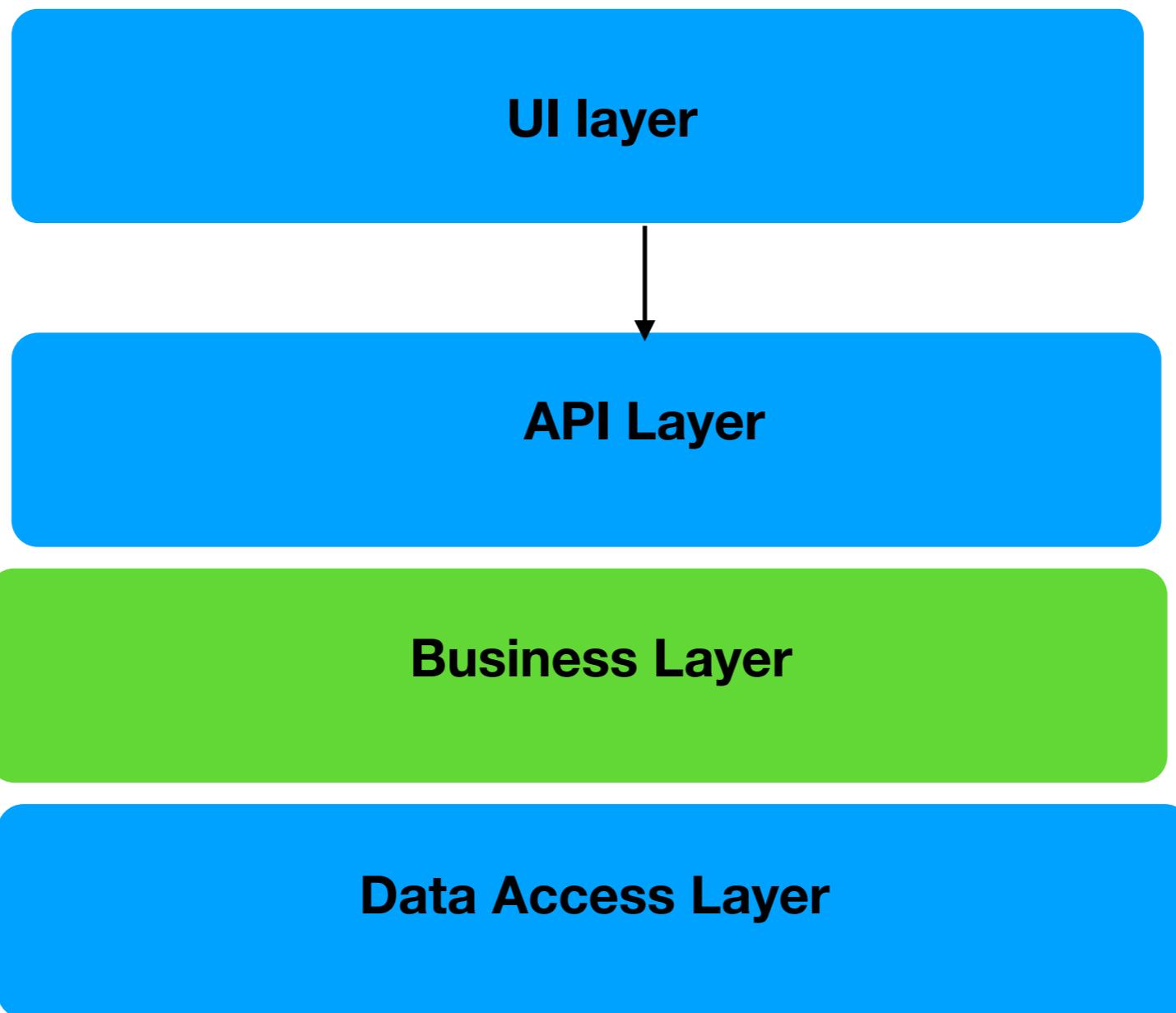


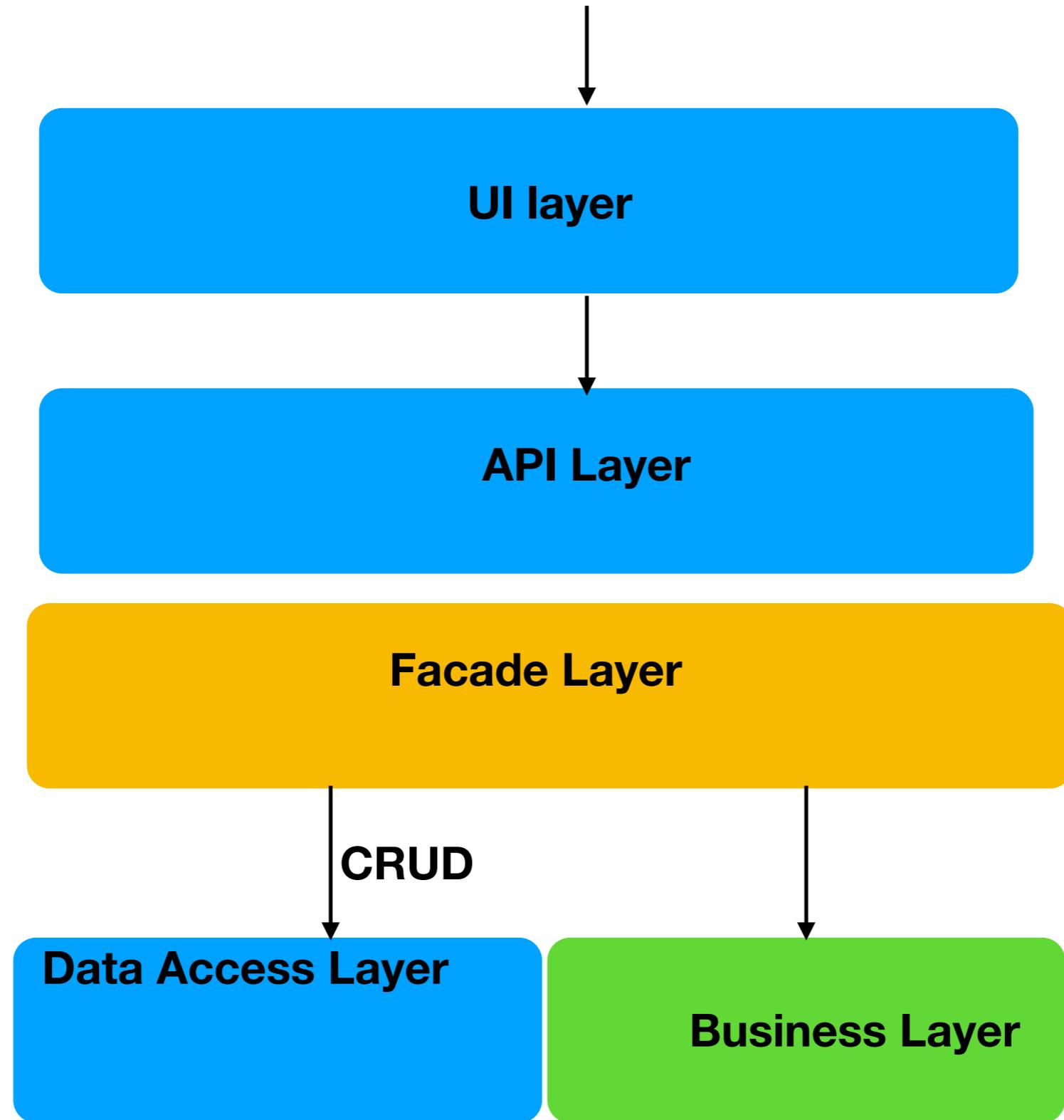
MVVM



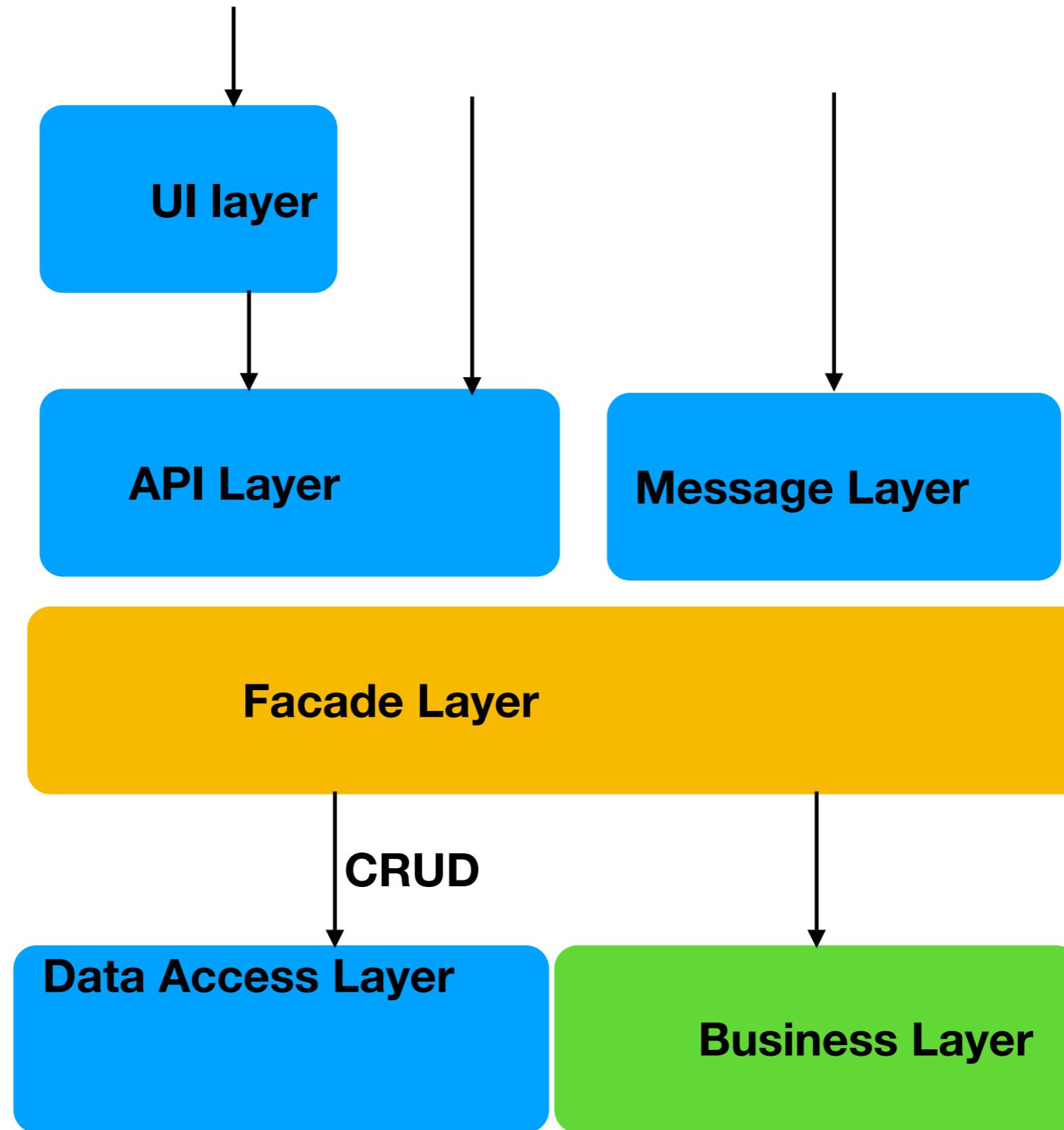
Hexagonal Arch



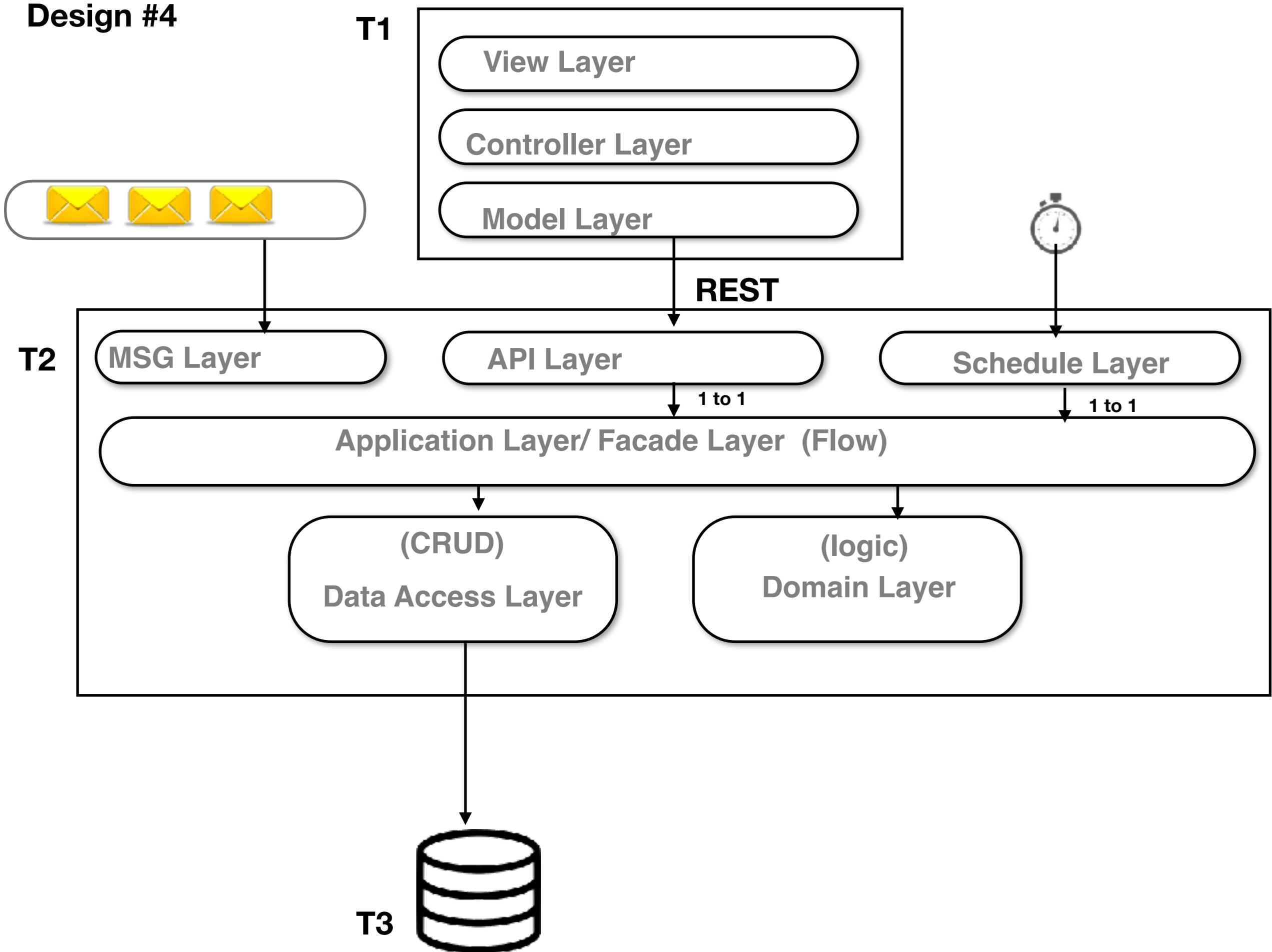




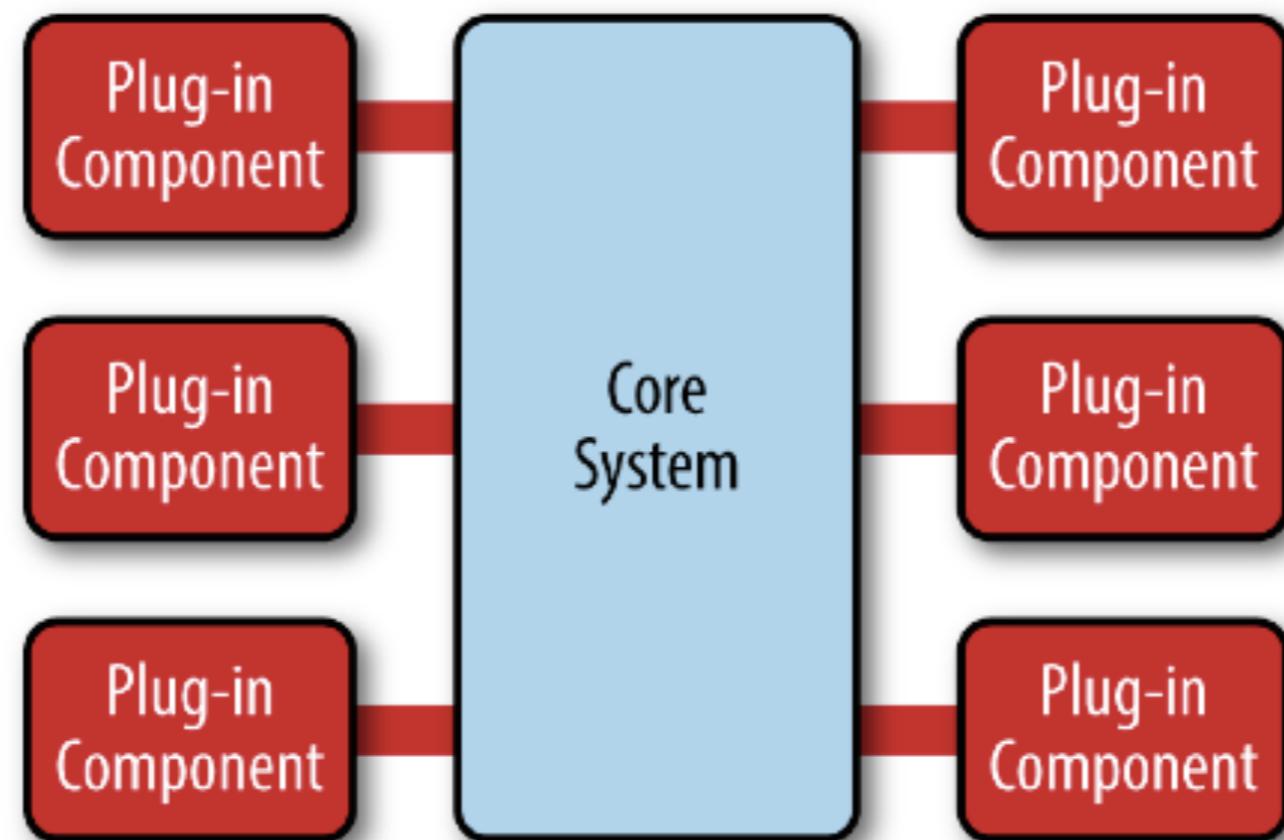
Hexagonal Arch



Design #4



Microkernel Architecture



ToDo App is simple and awesome app to organize your tasks with very easy to use interface. ToDo can help you to make list of your tasks and also you can set Reminder with specific tasks. It reminds you at you specified Time.

Order Processing. Receive an order, Decode an encrypted order, Authenticate the order, then Catch duplicated orders (for example if the order was sent twice) and send to order management system.

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

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.

Payment Service receives a data file which consists of,

- Payment information related to the invoices already delivered
- Notes regarding already delivered invoices
- Invoice information,
- And Credit Notes (Invoice Cancellations)

we need to import into the system.

- Invoices
- Payments
- Notes
- Credit Notes

Build two health care applications

1. Diabetes Management
2. Asthma Management.

Diabetes and asthma share a lot in common

1

**Application
Exe**

2

**Application
Exe**

3

Process

**Compile time
monolithic**

**Link time
monolithic**

**Runtime
monolithic**

In process
comp

4

Process

Process

5

Process

Process

Distributed
comp

Distributed
comp

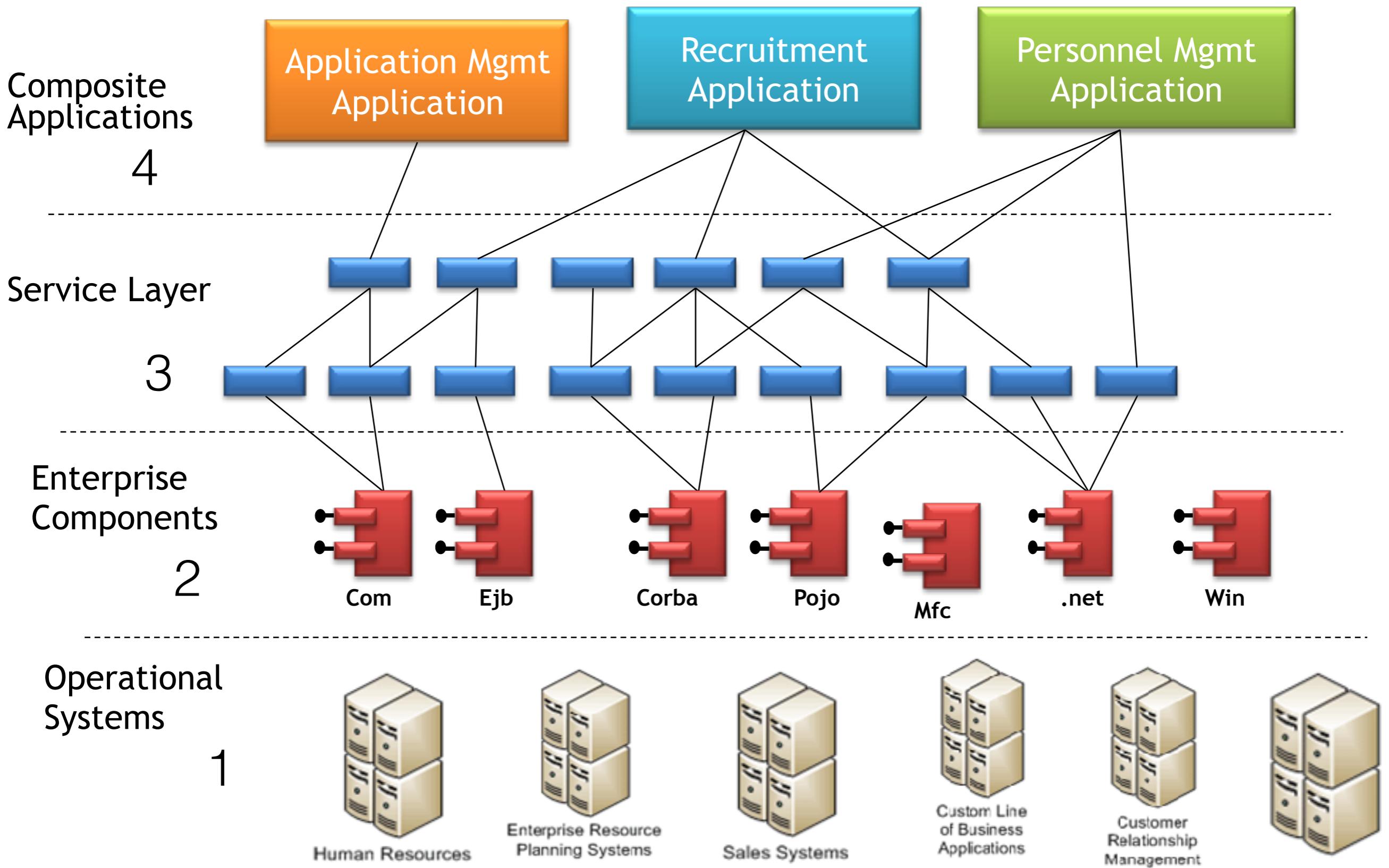
Application1

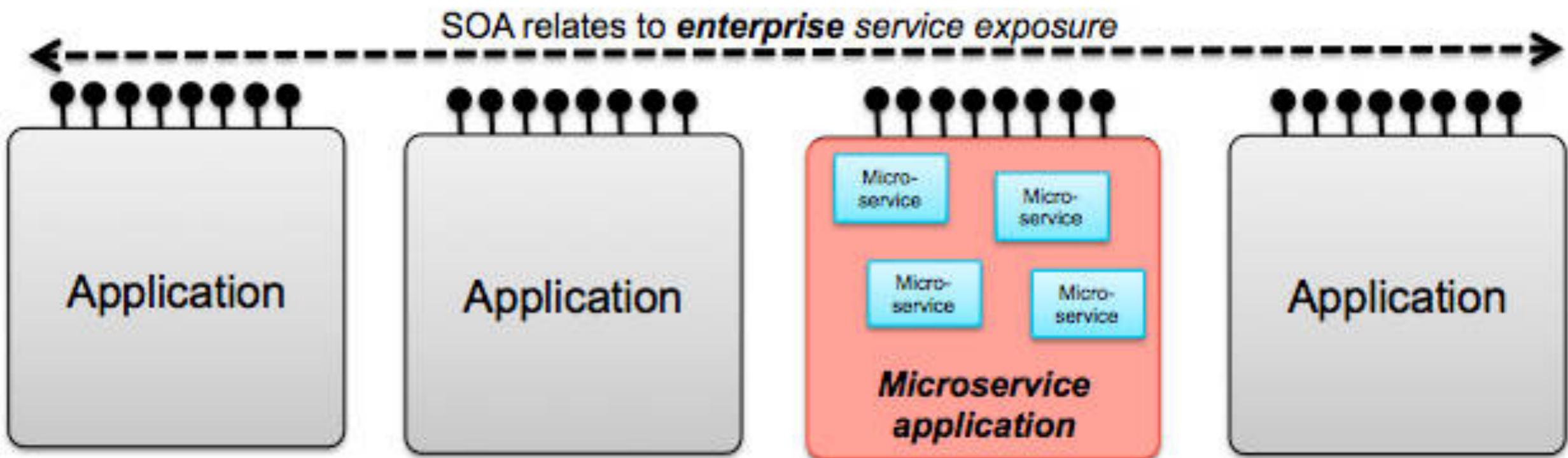
Application2

Distributed Application

Micro Application

Service Oriented Architecture

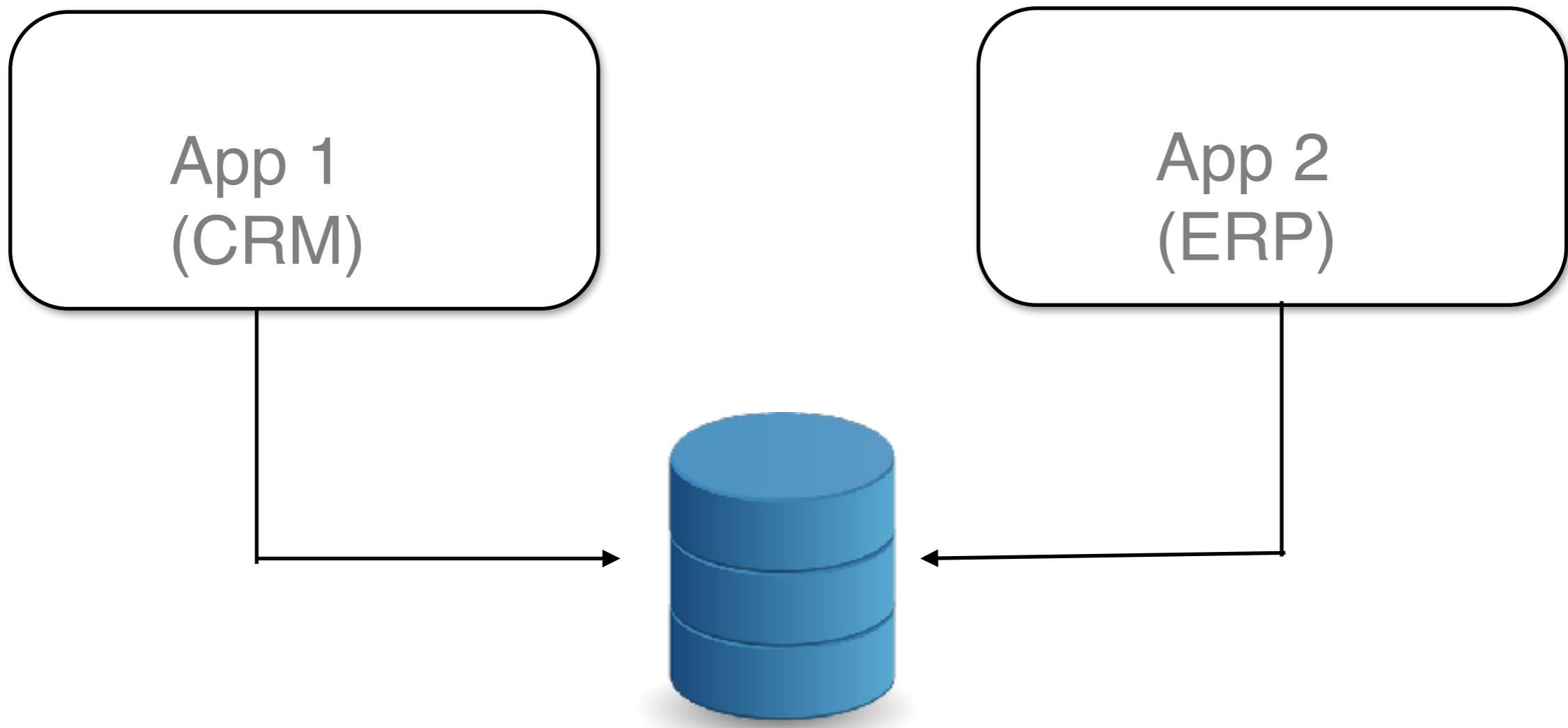




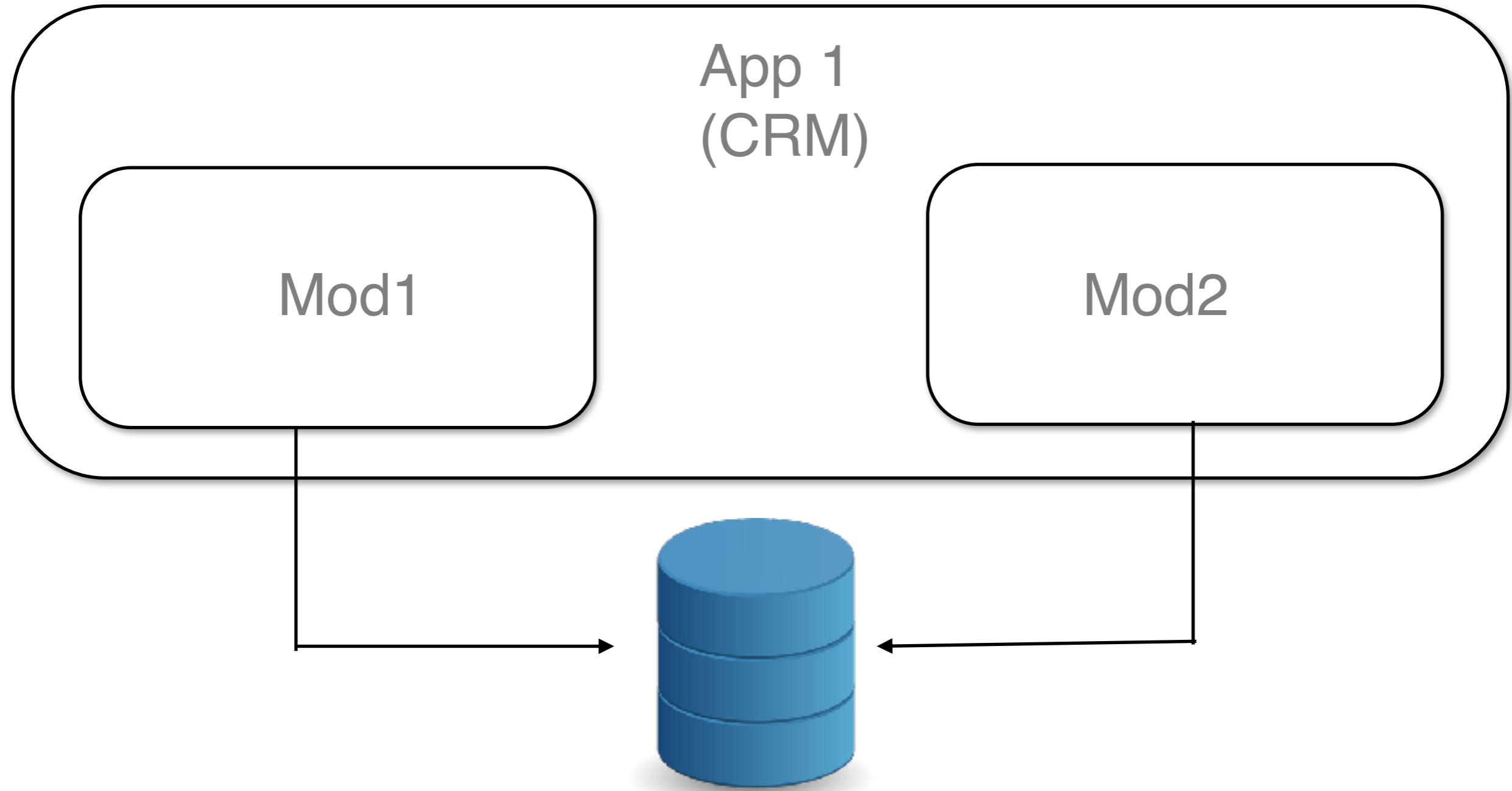
Microservices relate to
application architecture

	Distributed Application	Micro Application	W	Score
Database / Storage			2	
Infra (Hosting)			3	
Sorce Control			2	
CI/CD (Build Server)			3	
Fun Requirements			1	
SCRUM Team / Sprint			1	
Test Cases			1	
Architecture			1	
Technology Stack / Fwks			1	

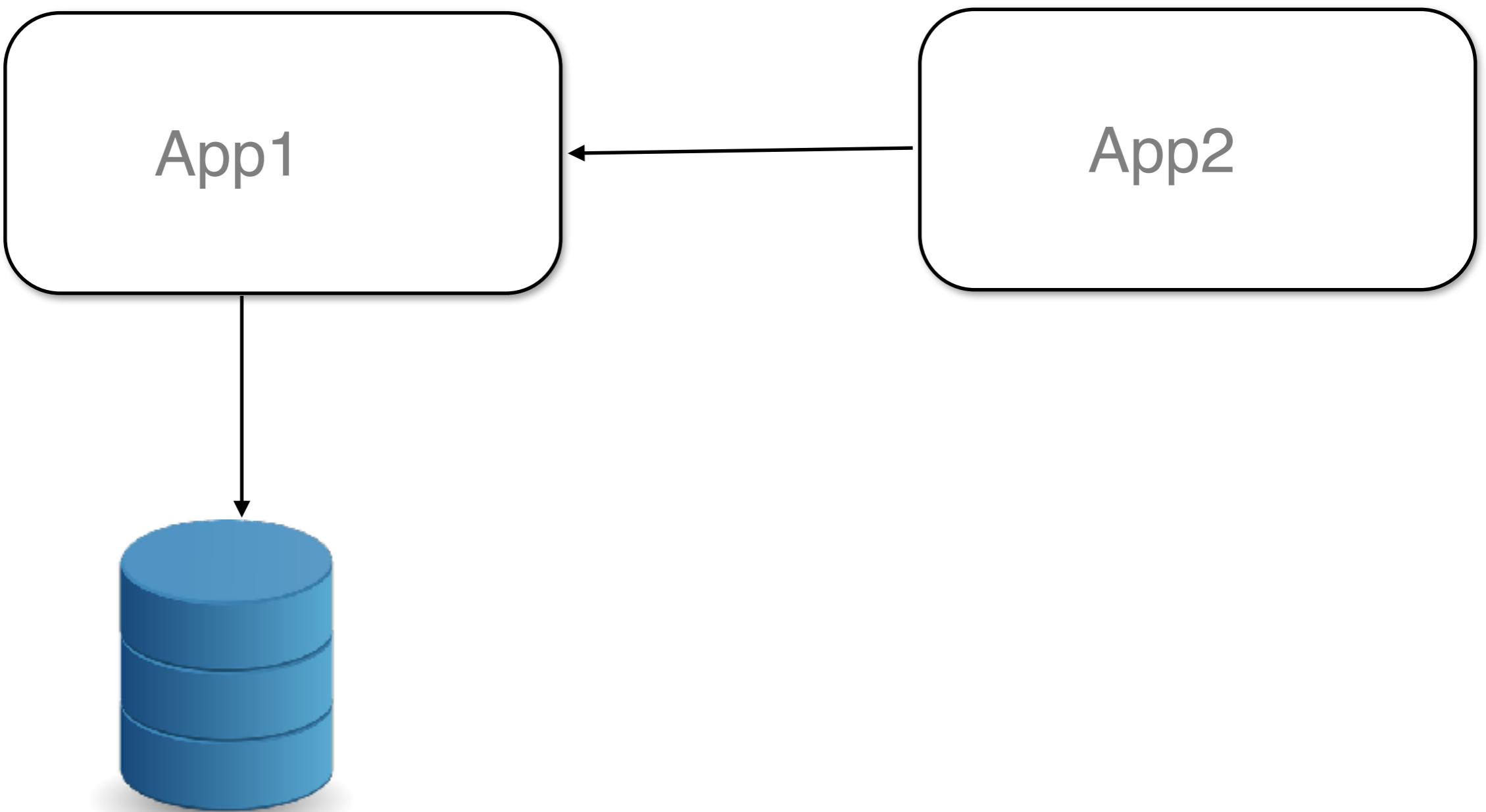
1. Database



1. Database

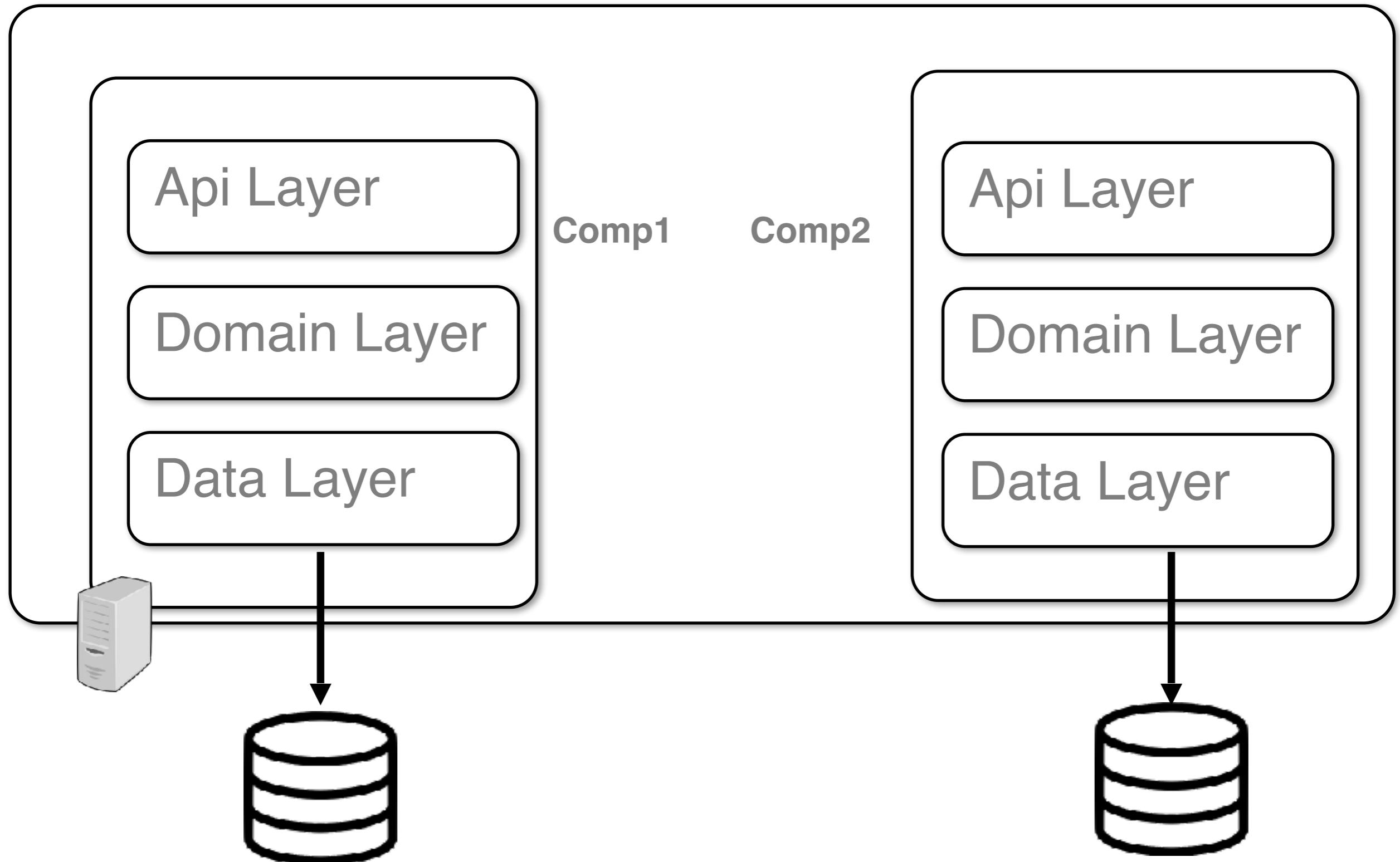


1. Database



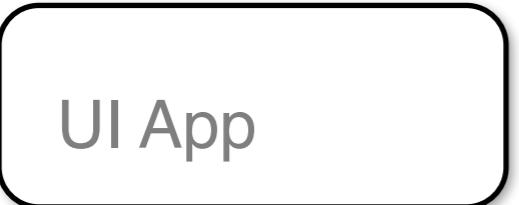
2. Infrastructure

Web Server

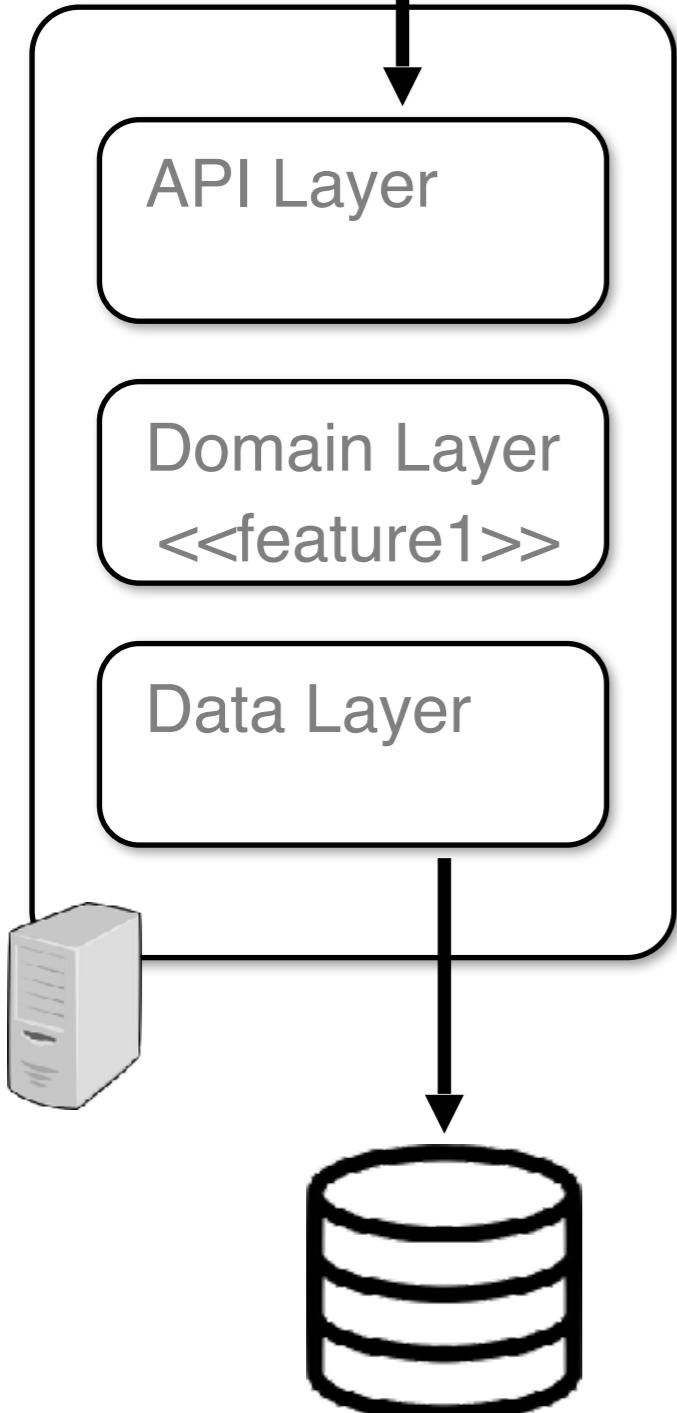


Types of Microservice

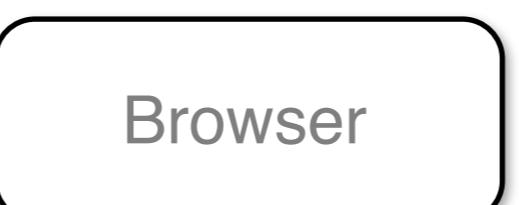
1

 UI App

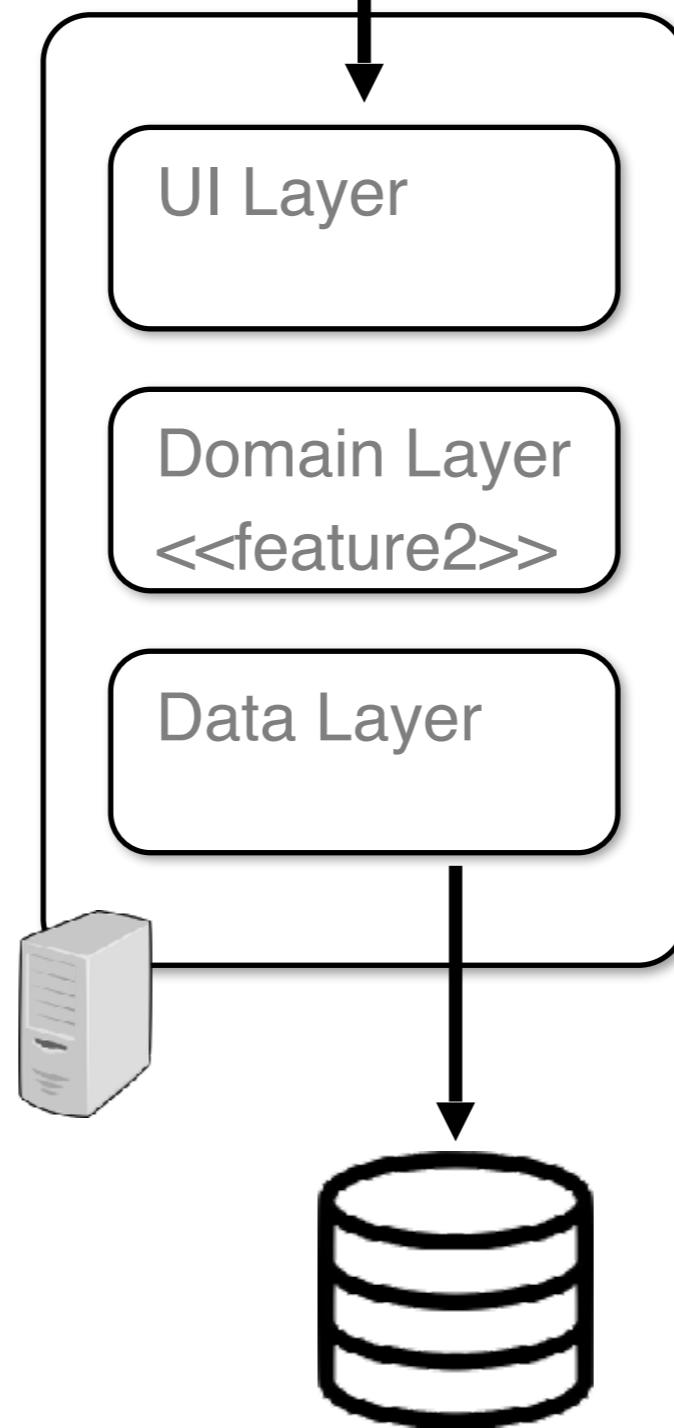
Web Server



2

 Browser

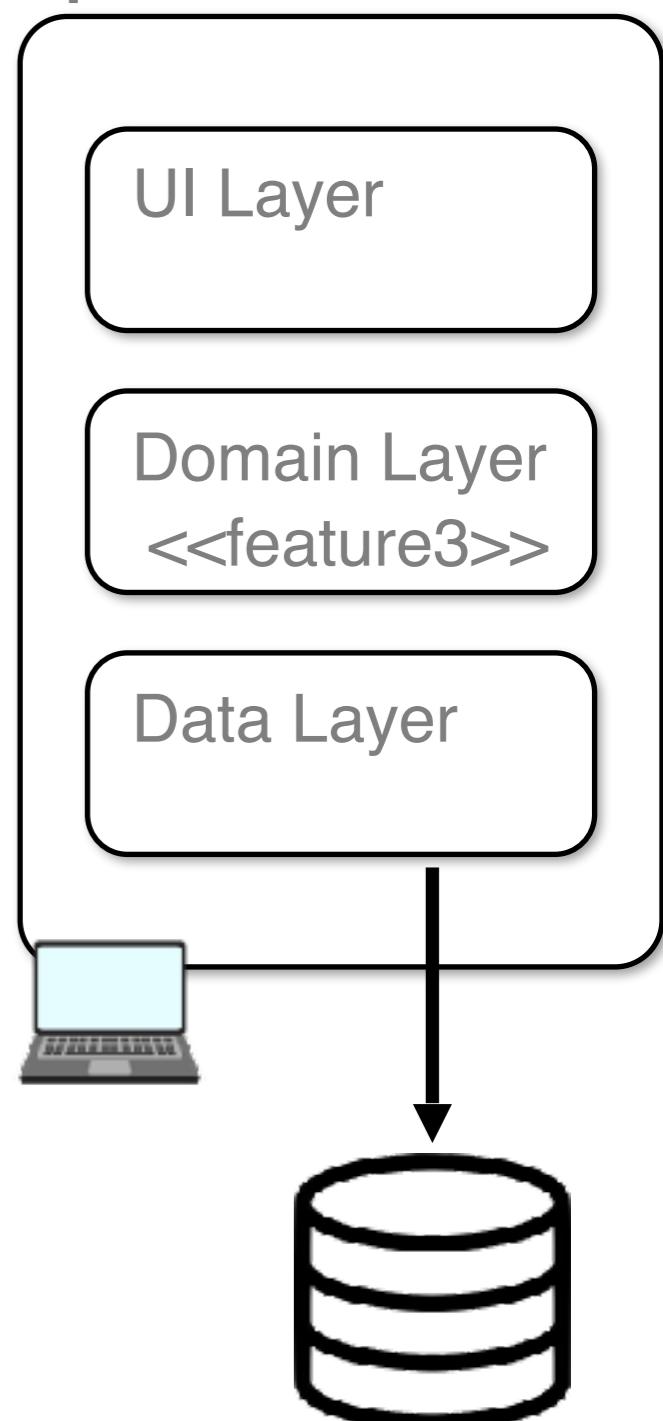
Web Server



3



Desktop

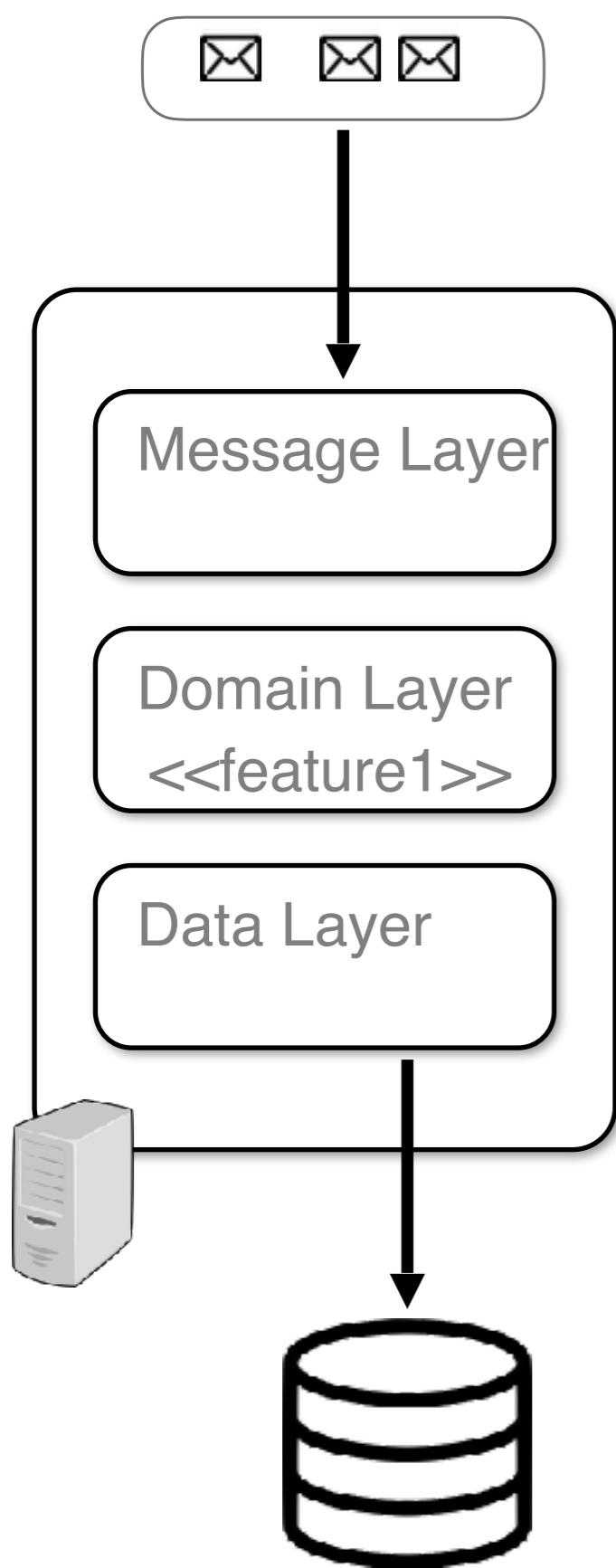


Angular (SPA)

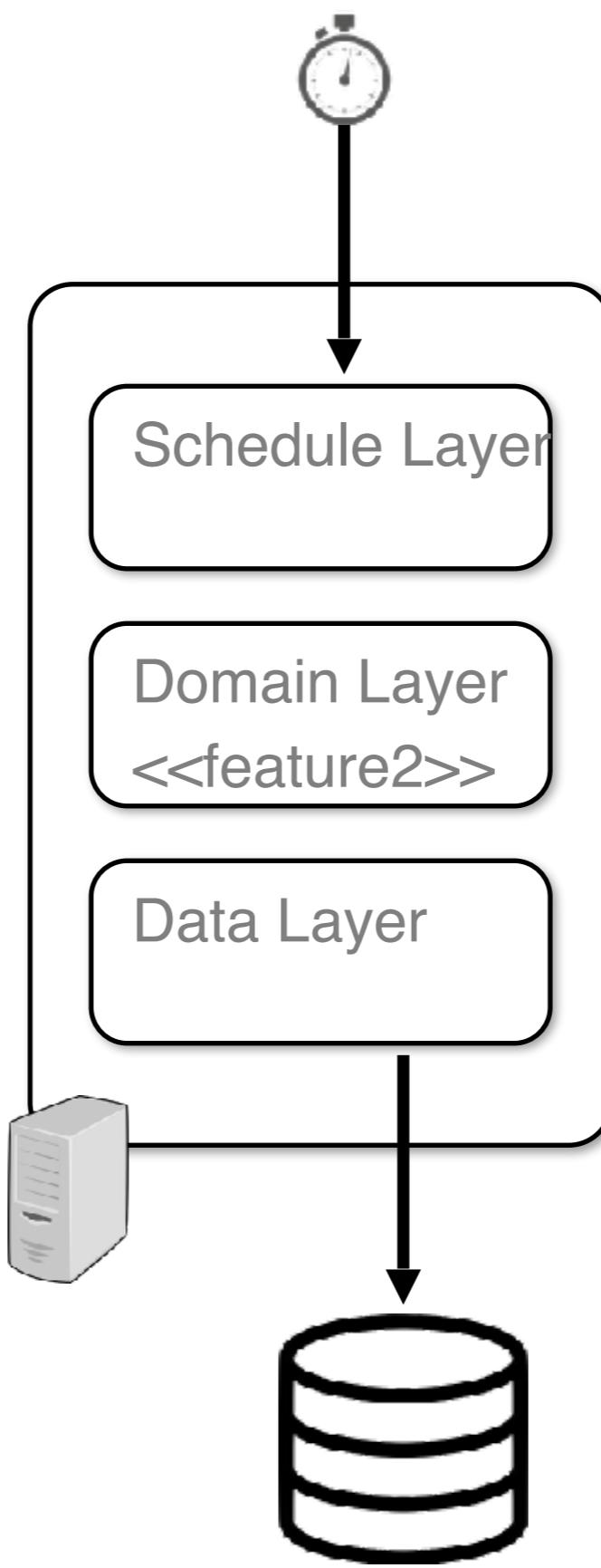
JSP, Servlet, JSF (Server pages)

Swing, Android, iOS (Native)

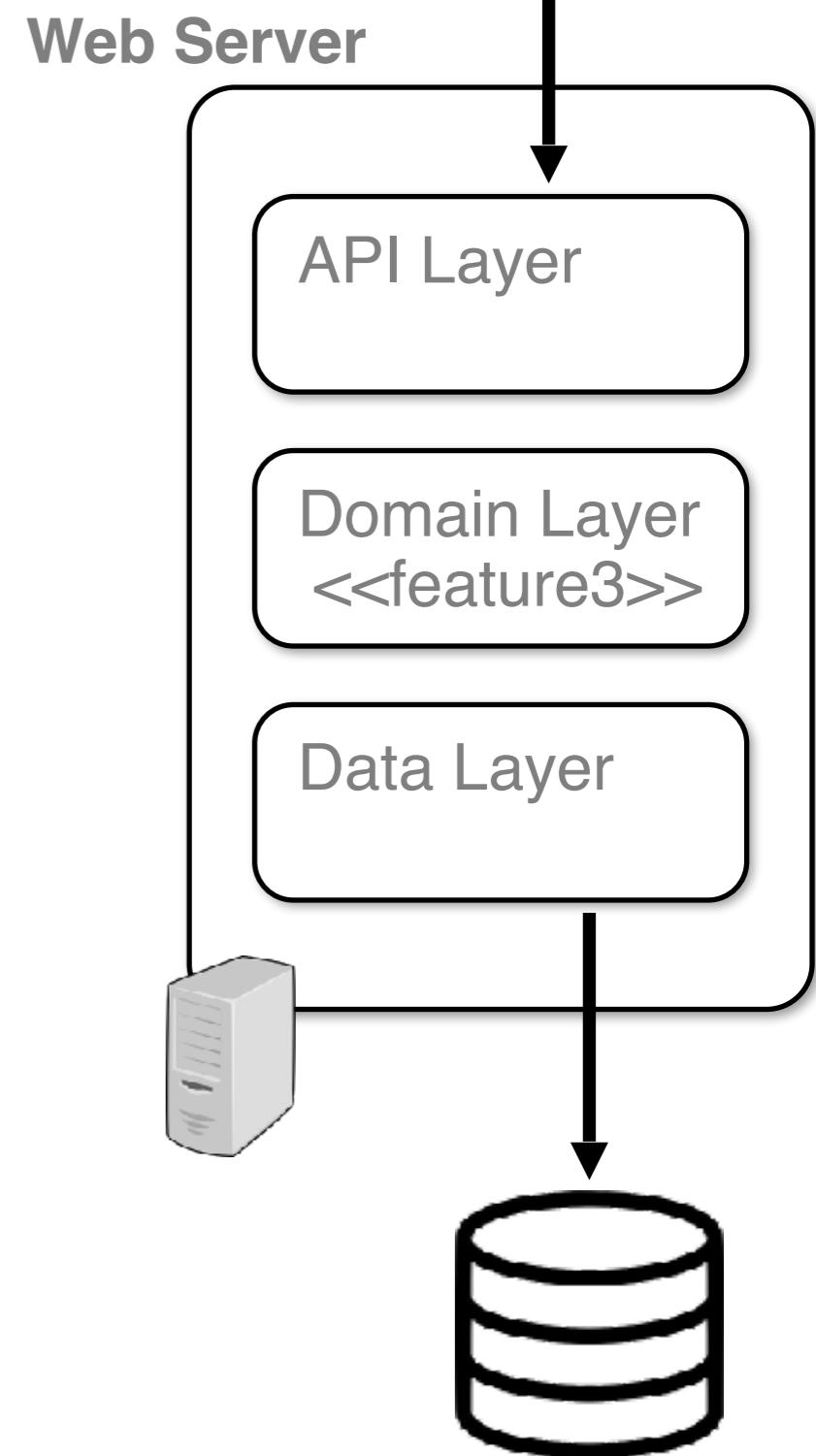
1



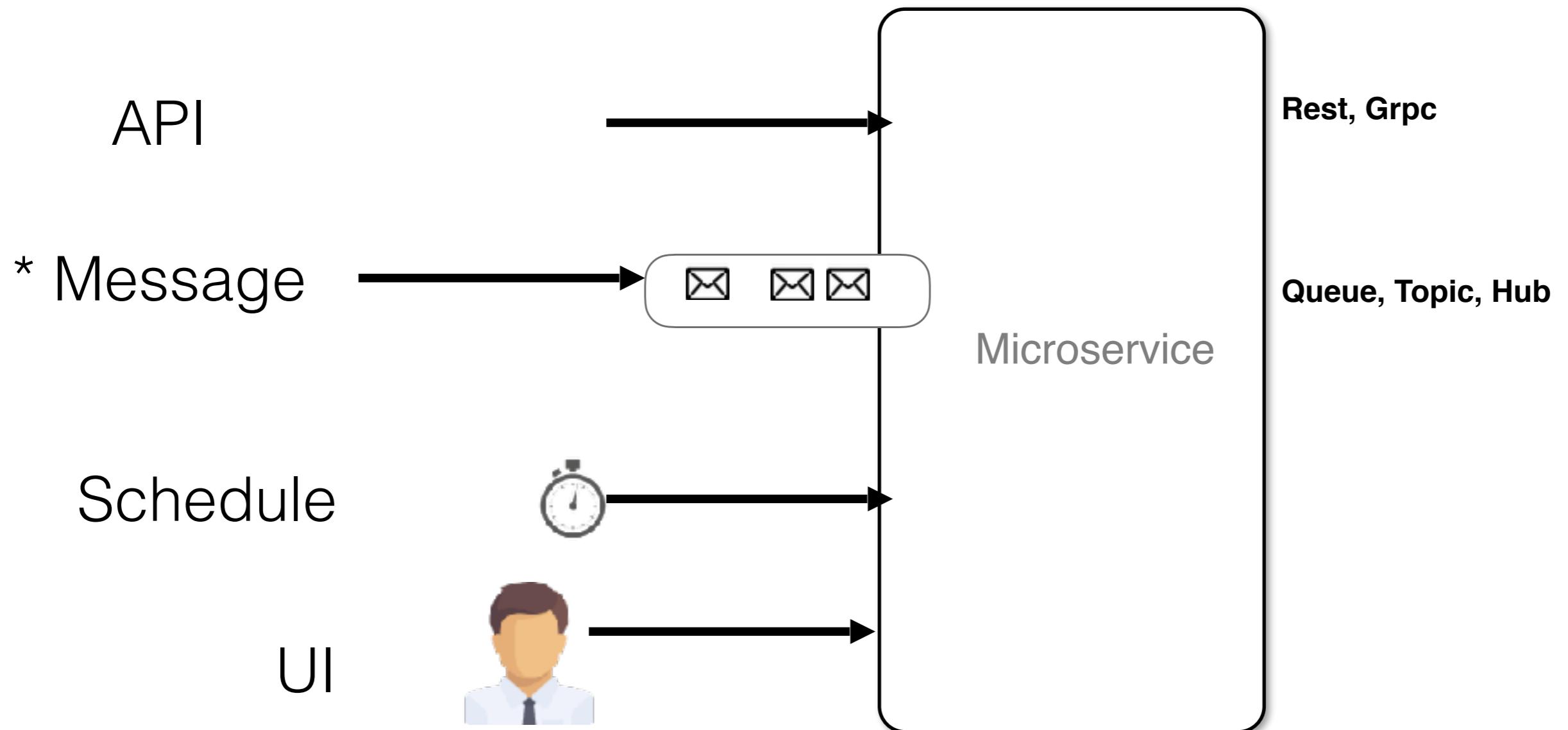
2



3

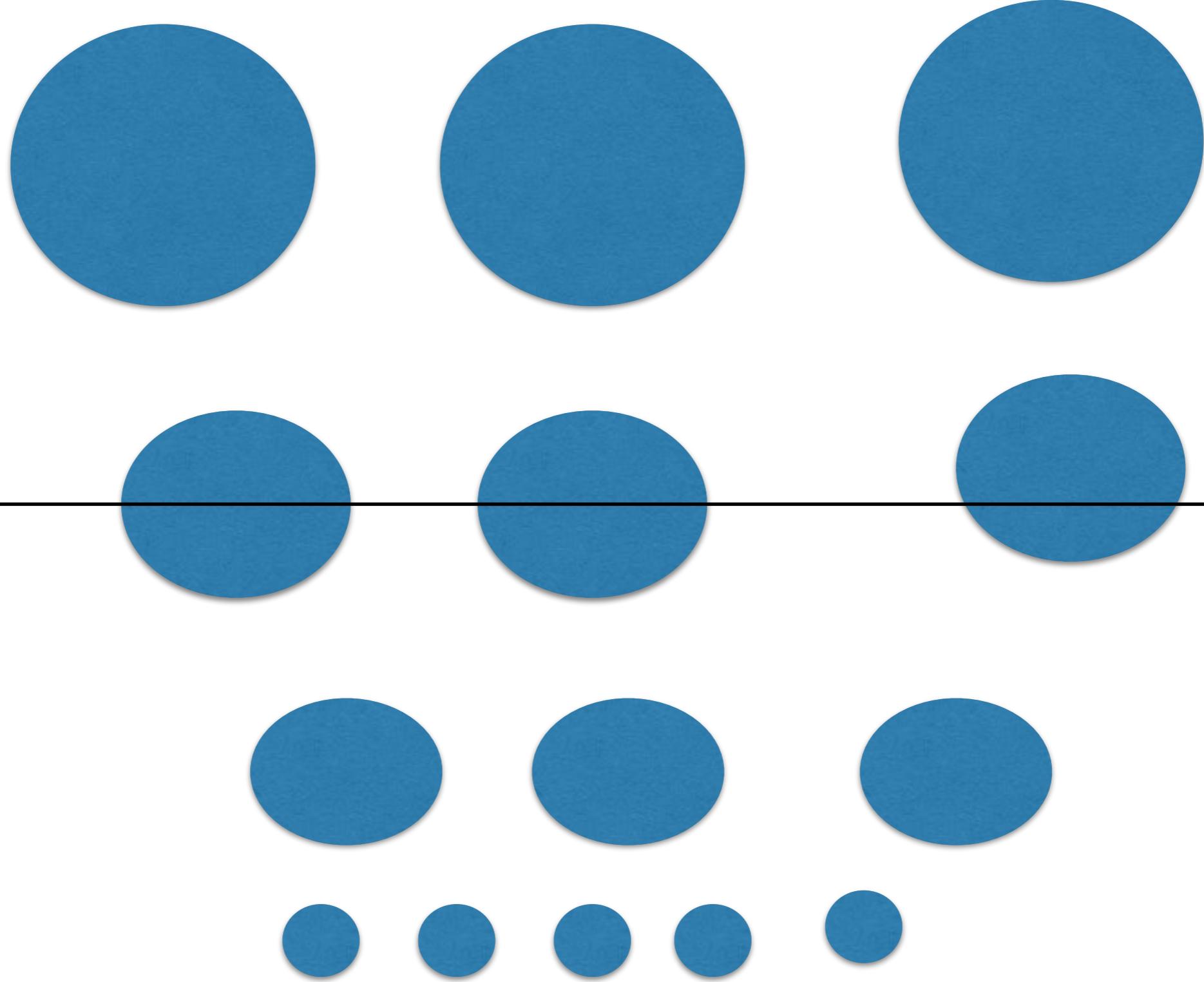


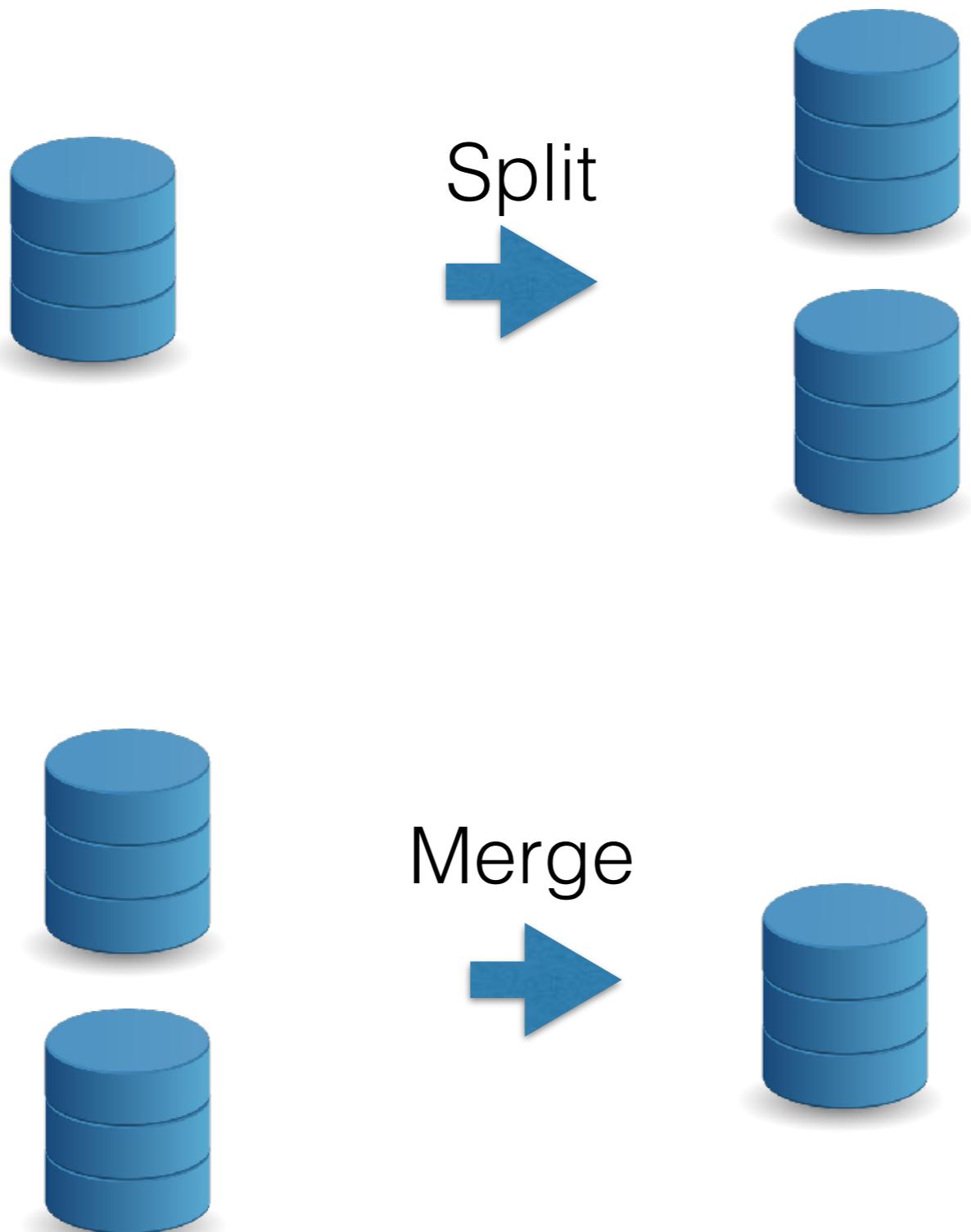
Hexagonal Architecture



Coarse

Fine



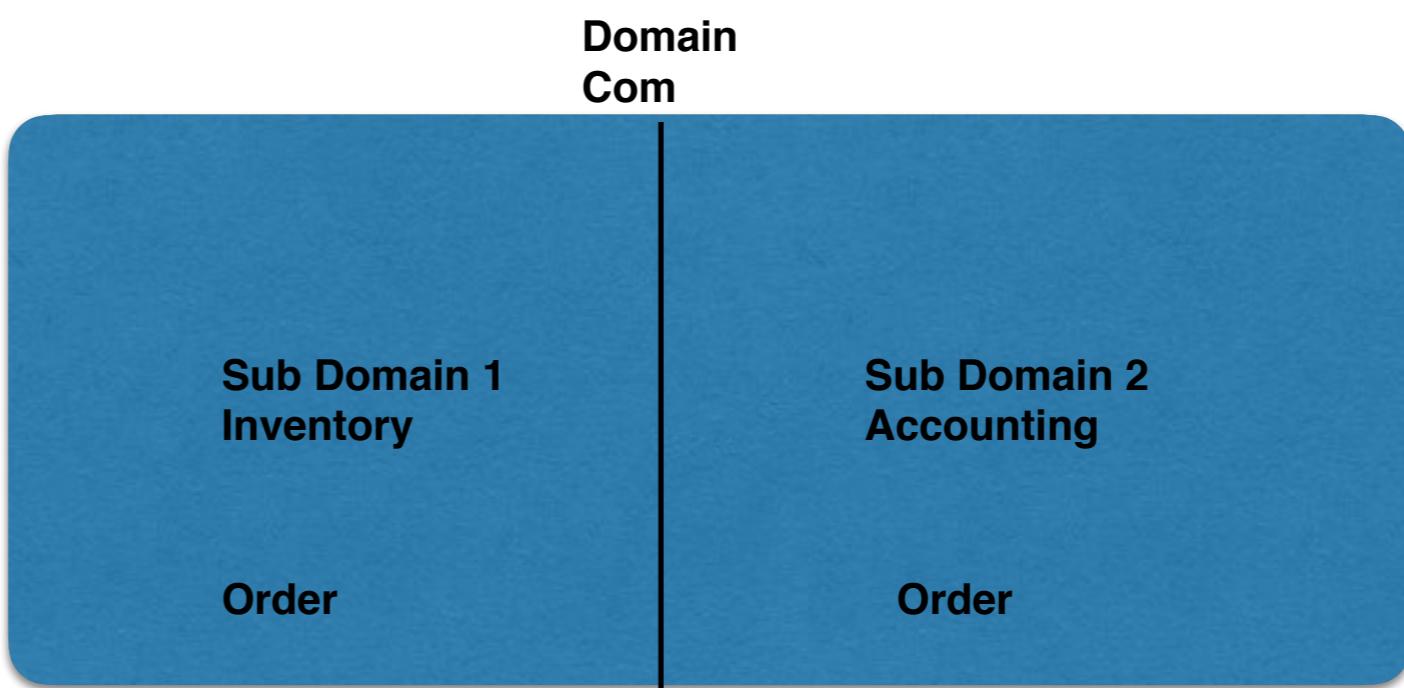


*Transaction
Boundary

Bounded
Context (DDD)

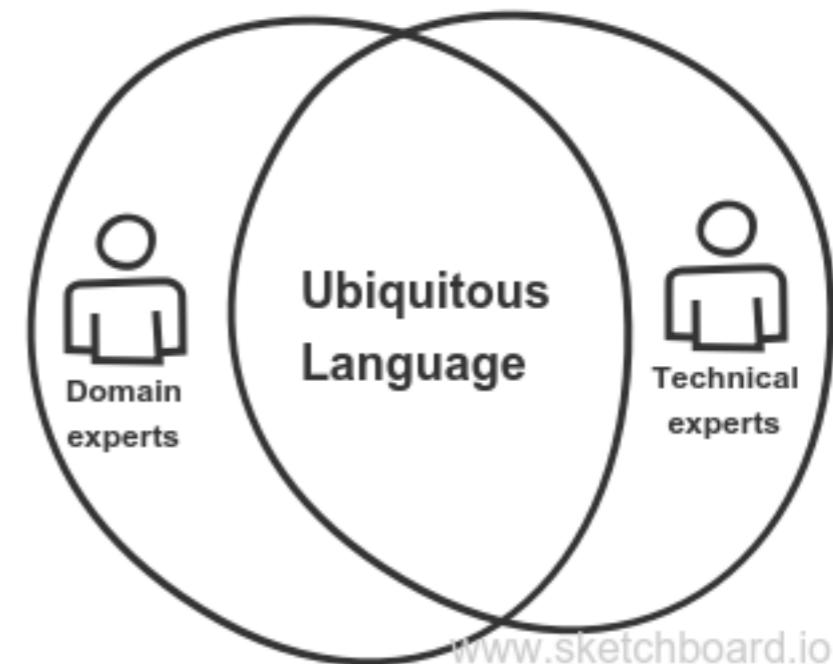
Aggregate
(DDD)

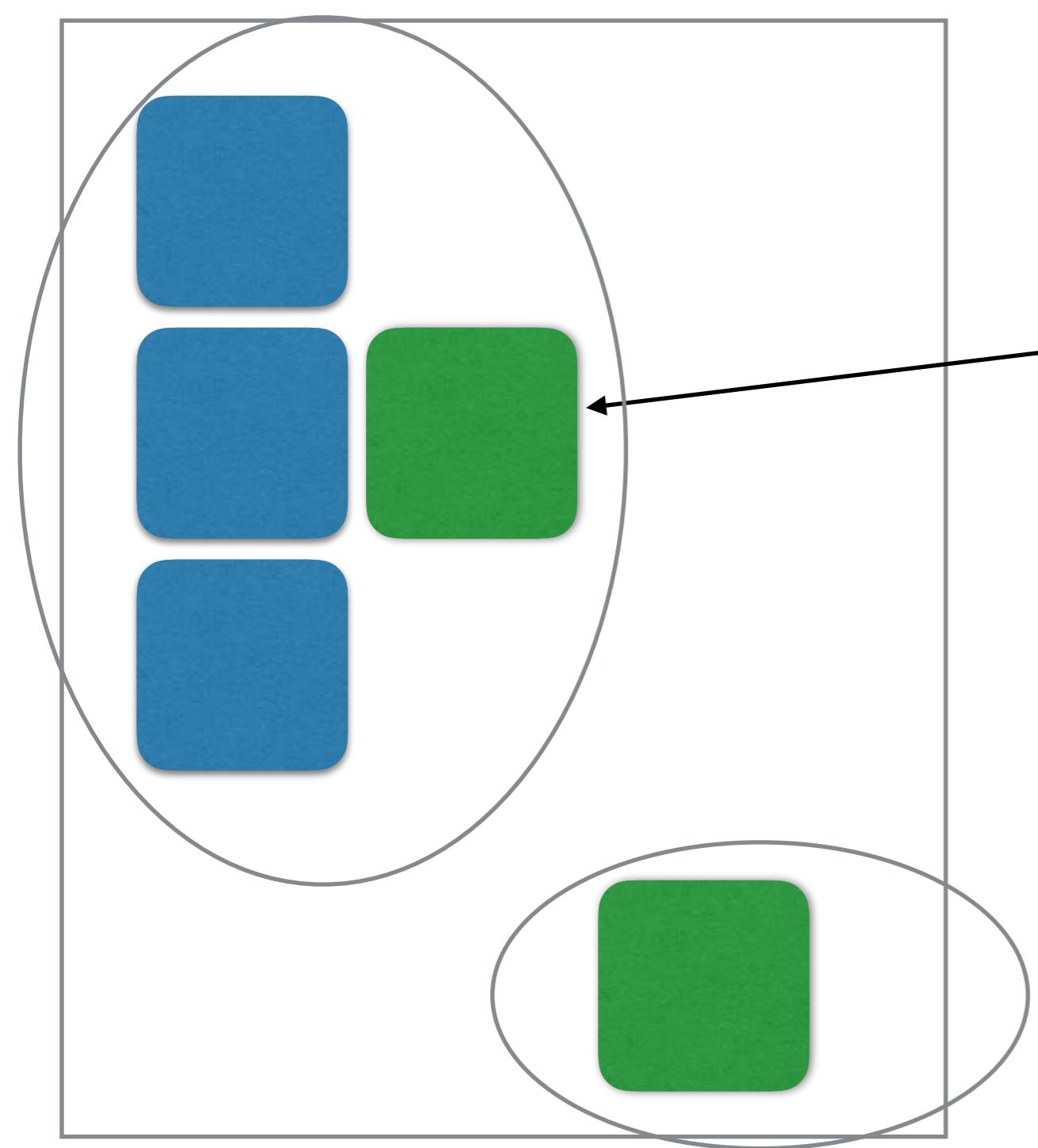
Feature



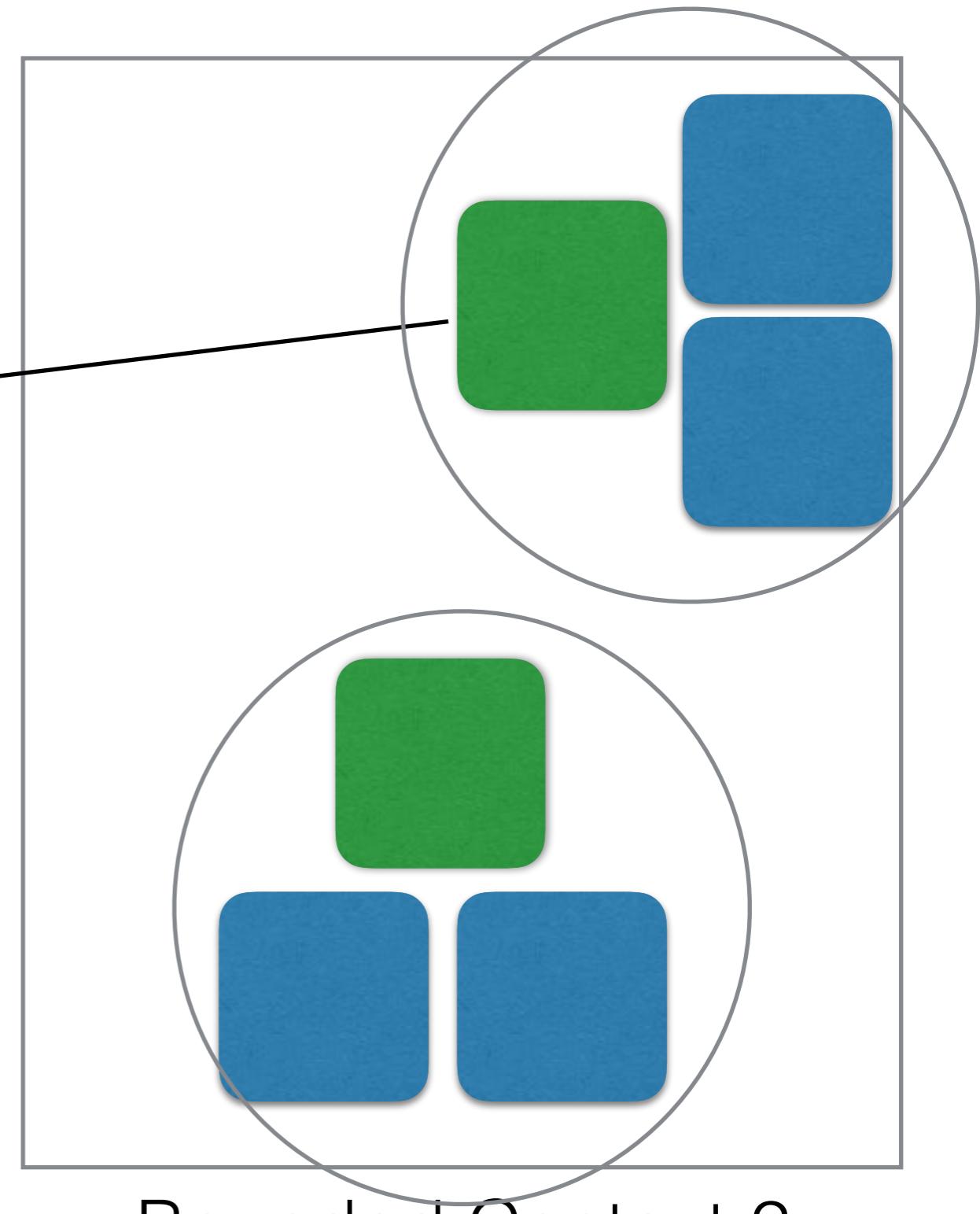
DDD

Ubiquitous language
Bounded Context
Aggregates

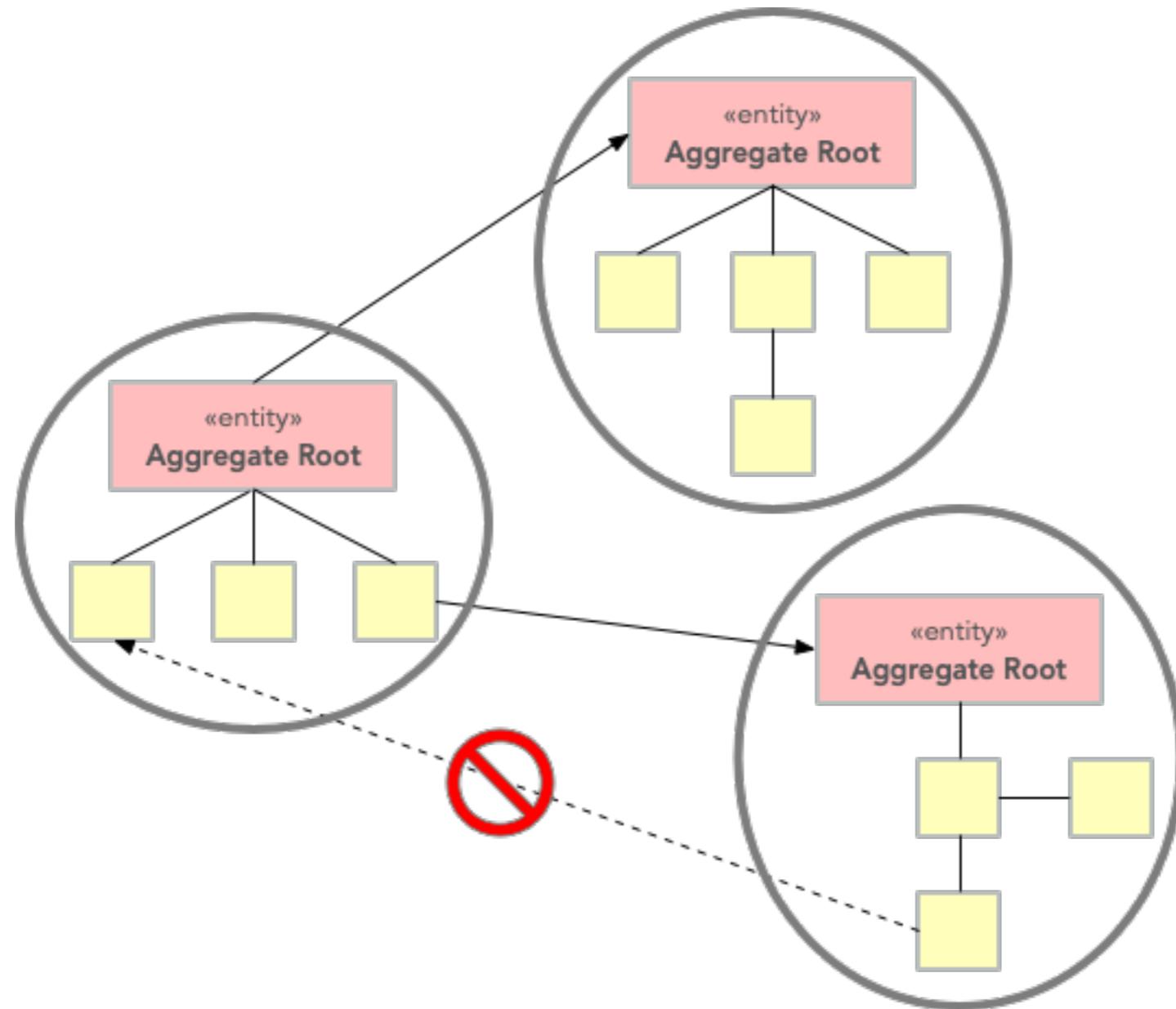




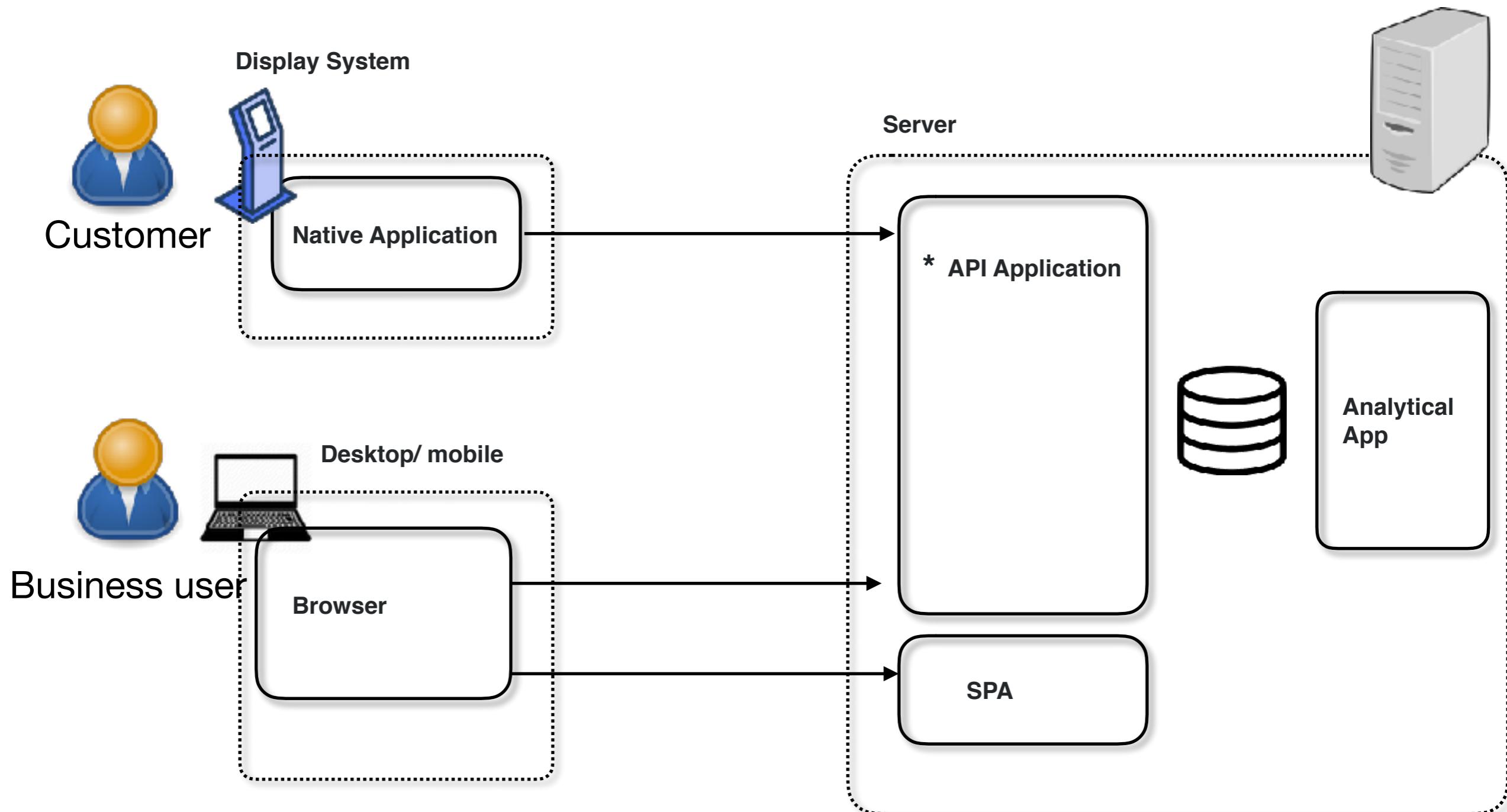
Bounded Context 1



Bounded Context 2



Logical view - todo



	Pros/ Cons	Solution
Development time		
Learning Curve		
Resource Performance (CPU, Memory, I/O)		
Db Transaction Management		
Views / Report / Dash board/ join		
Infra Cost		
Deployment effort		
Debugging, Error Handling,		
Integration Test		
debug/ error Log Mgmt		
Config Mgmt		
Authentication		
Authorization		
Audit Log mgmt		
Monitoring / Alerting		
Data Security and Privacy		
Build Pipeline (CI)		
Agile Architecture (Agility to change)		
Feature Shipping (Agility to ship)/ CD		
Scalability (volume - request, data,		
Resilience		
Availability		
Ability to do Polygot		
Maintainability (easy to change)		

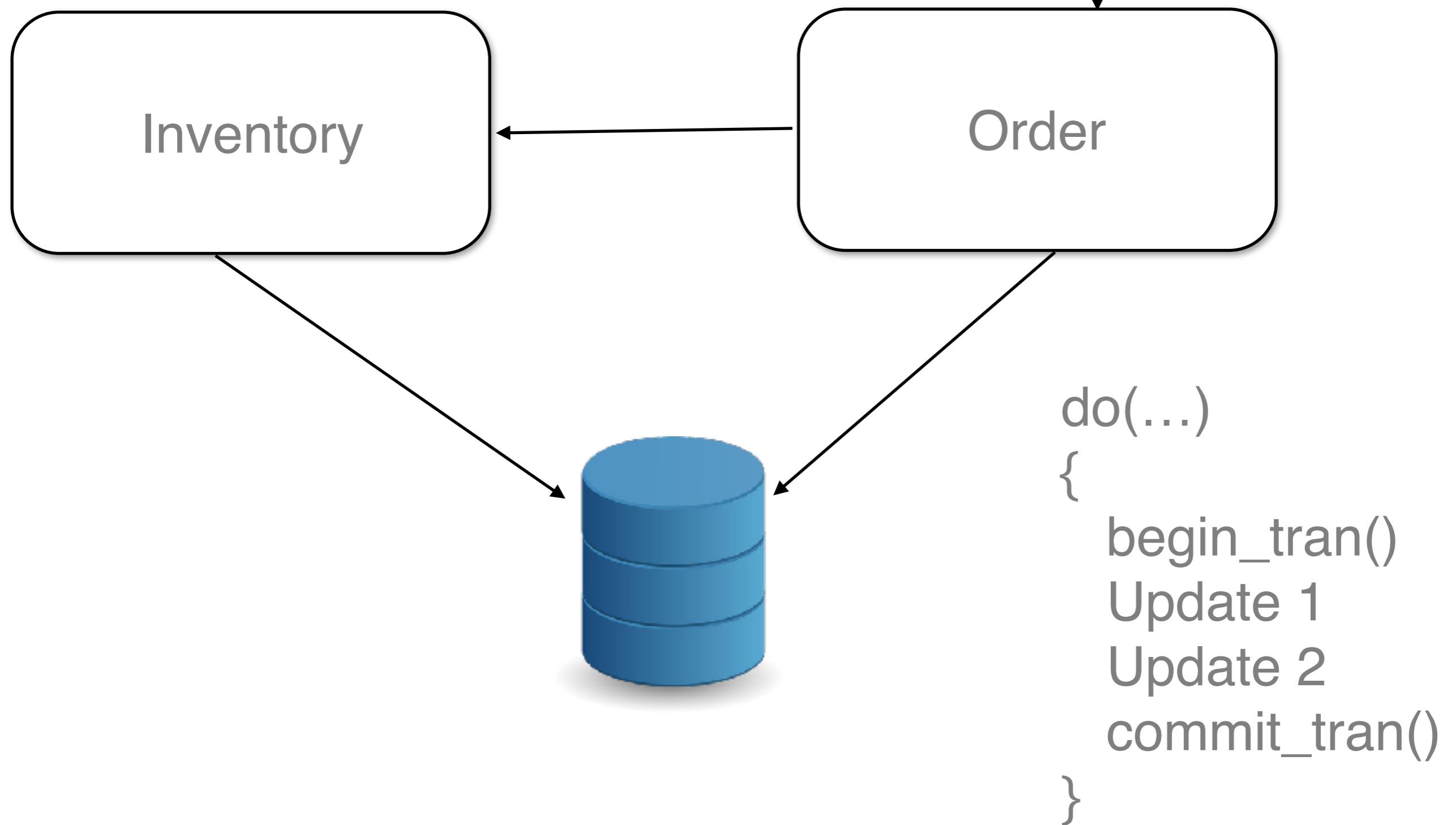
	Cpu Cycles
a + b	3
Fun call	10
CreateThread	2,00,000
Destroy thread	1,00,000
Db call	40,00,000
Api Call	10, 00, 000

YAGNI - you aren't gonna need it
KISS
Solve tomorrow's problem tomorrow

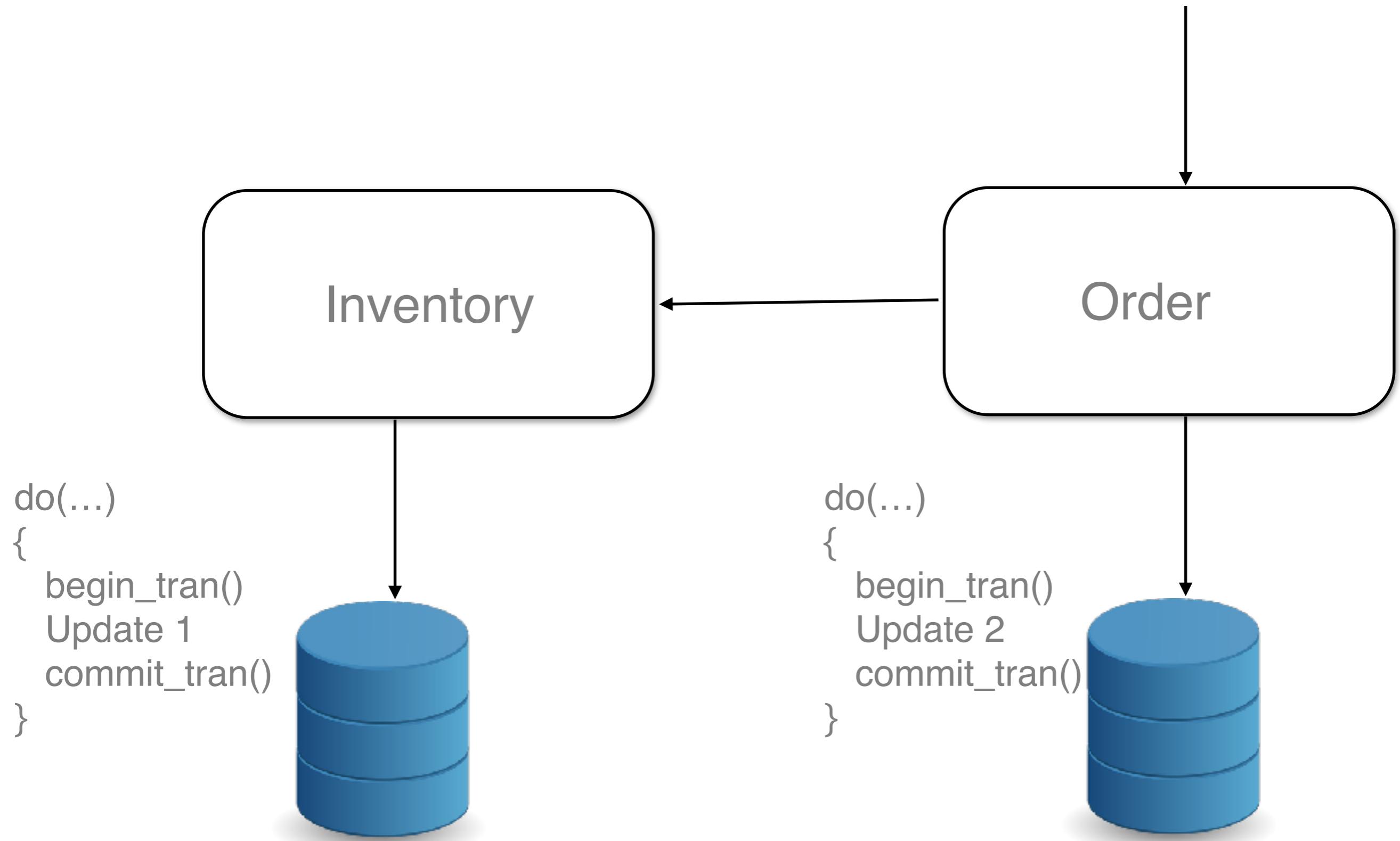
1. Performance

Operation	Cpu Cycles
• $10 + 12$	5
• 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

2. Db transaction

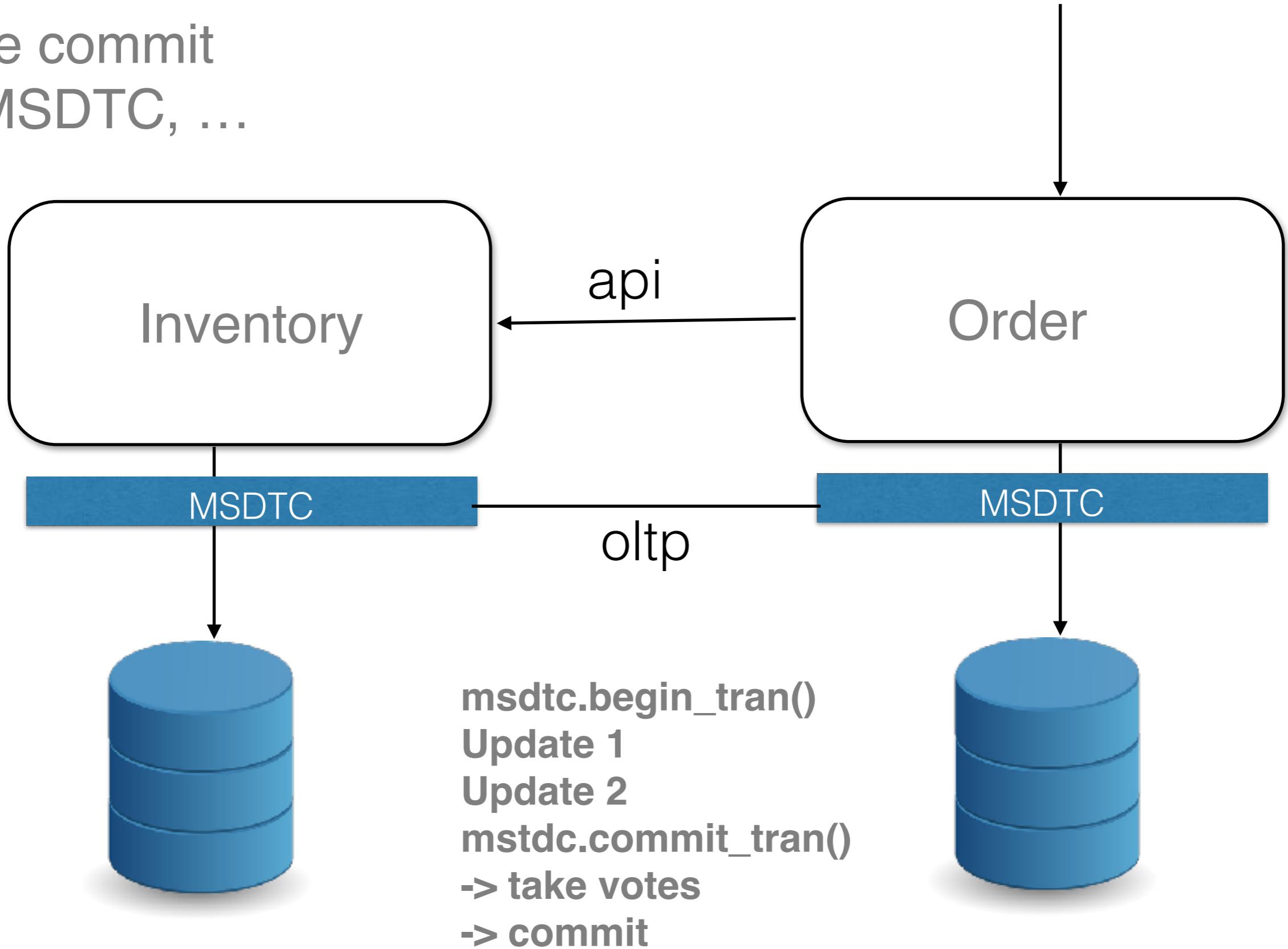


2. Db transaction

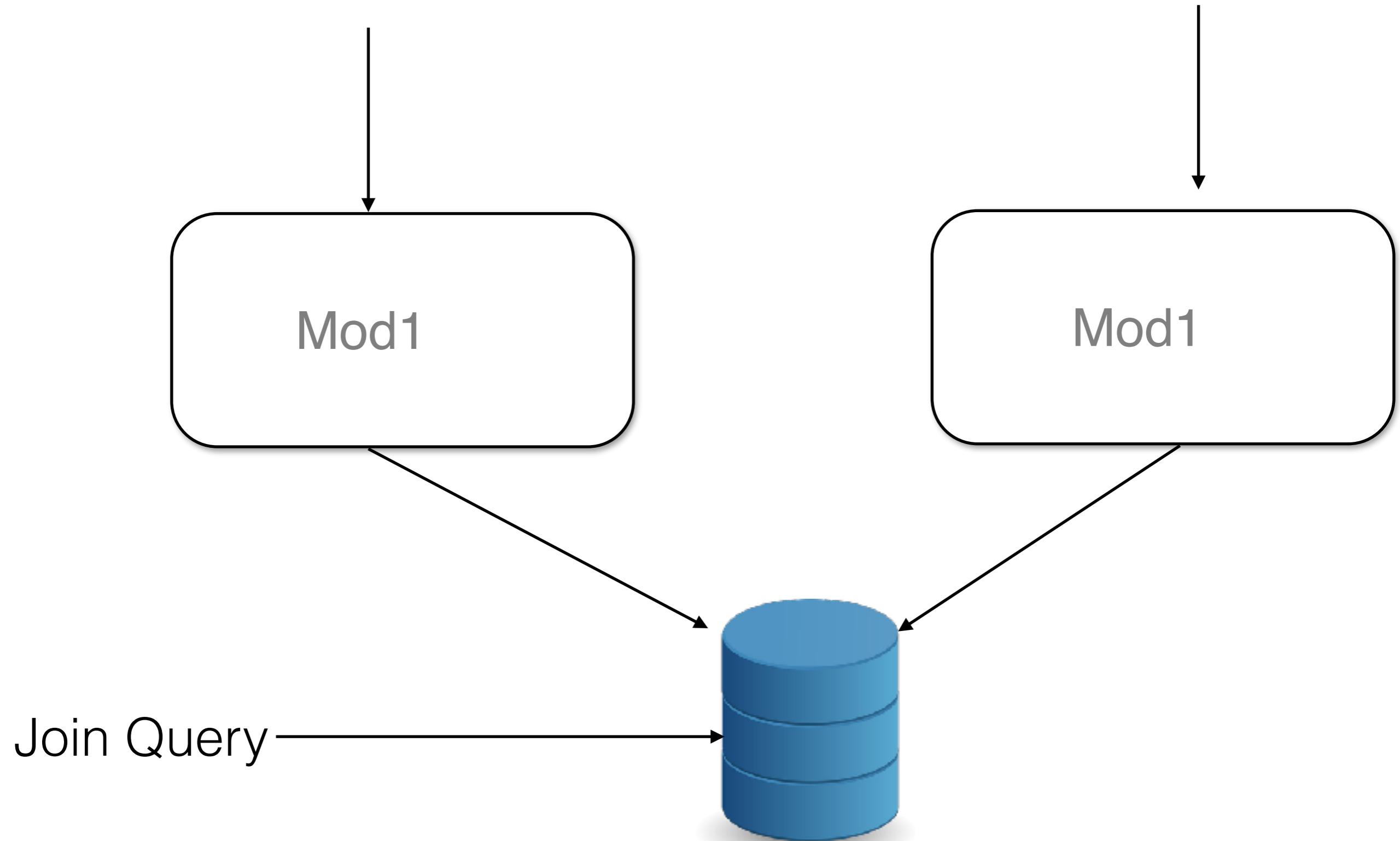


2. Db transaction

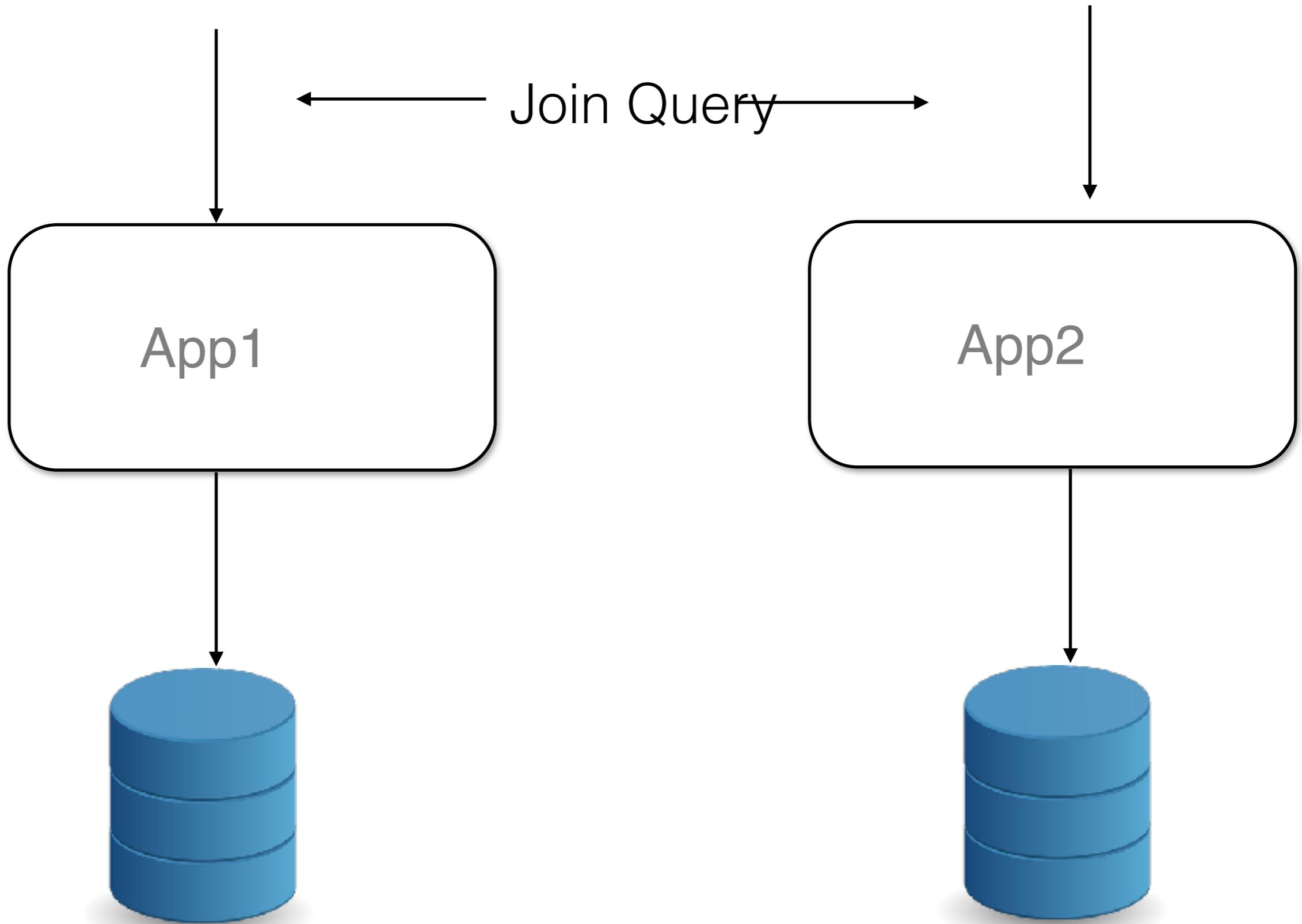
2 phase commit
JTX , MSDTC, ...



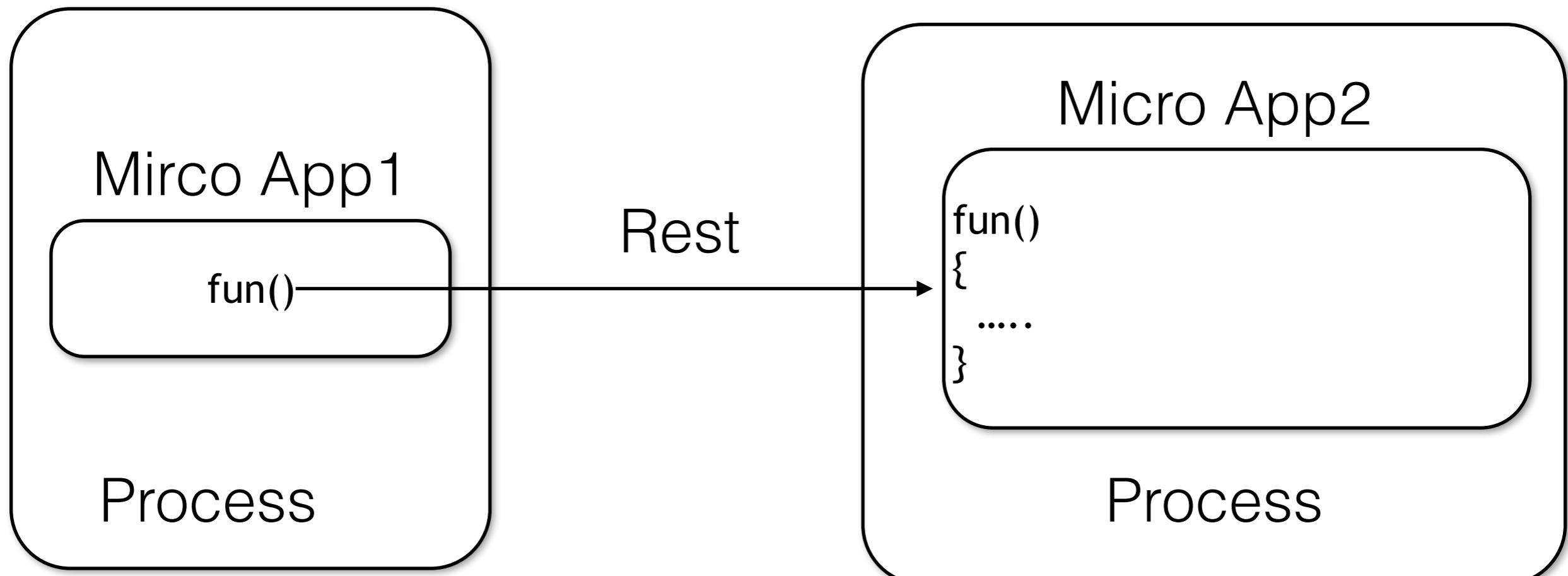
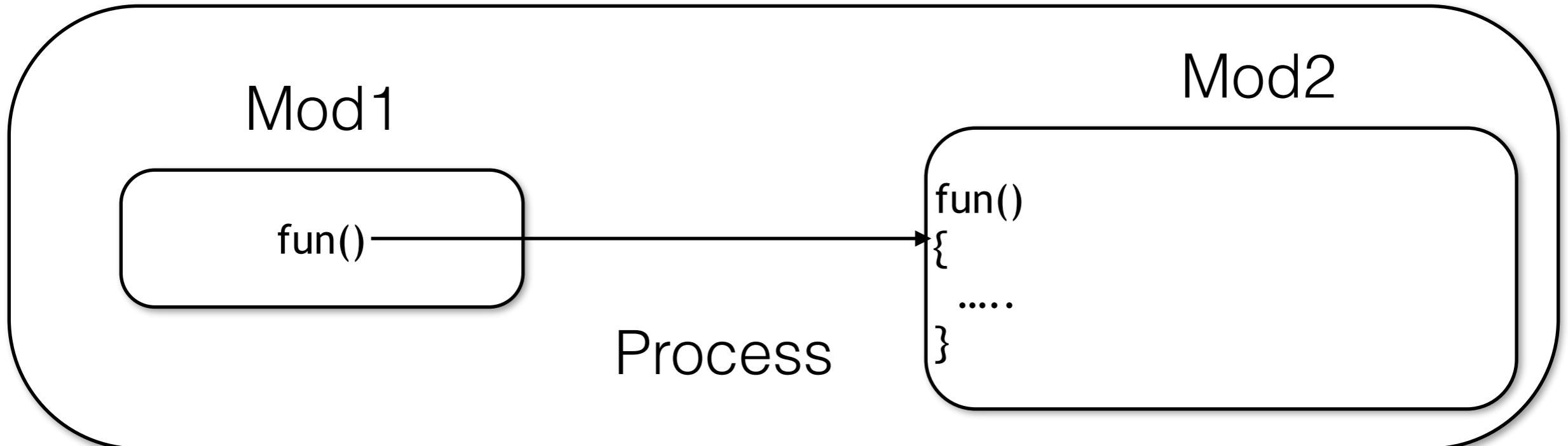
3. Db query



3. Query



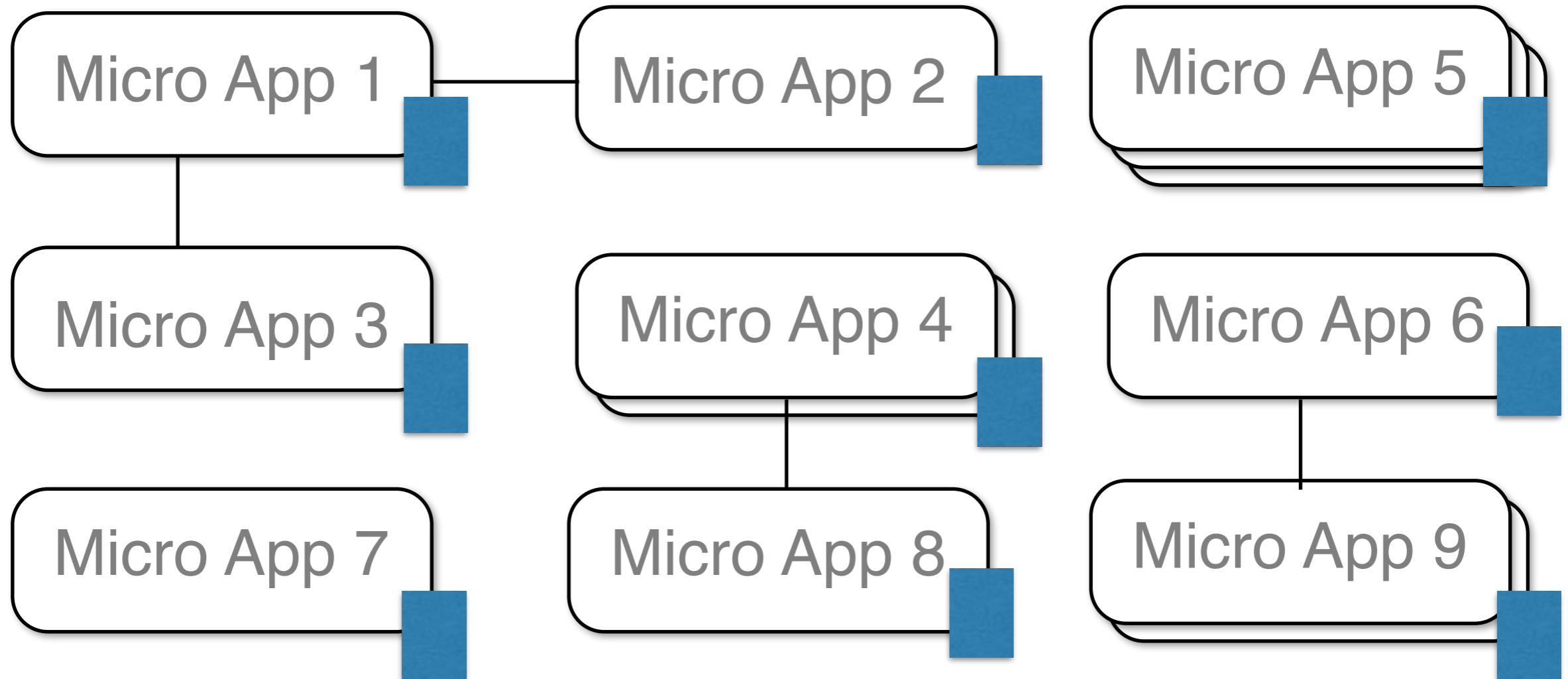
4. Development time



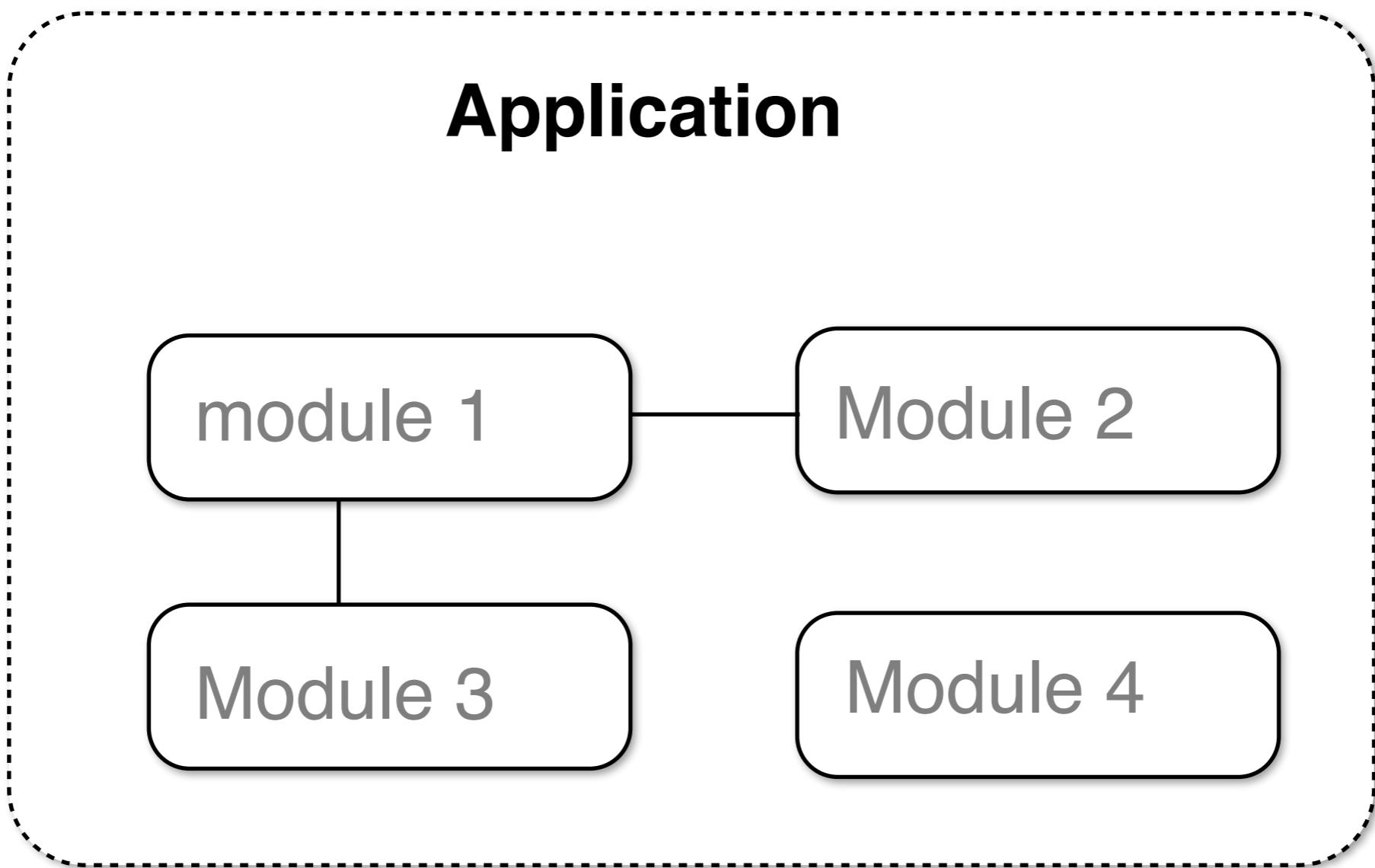
5. Learning Curve



6. Infra Cost

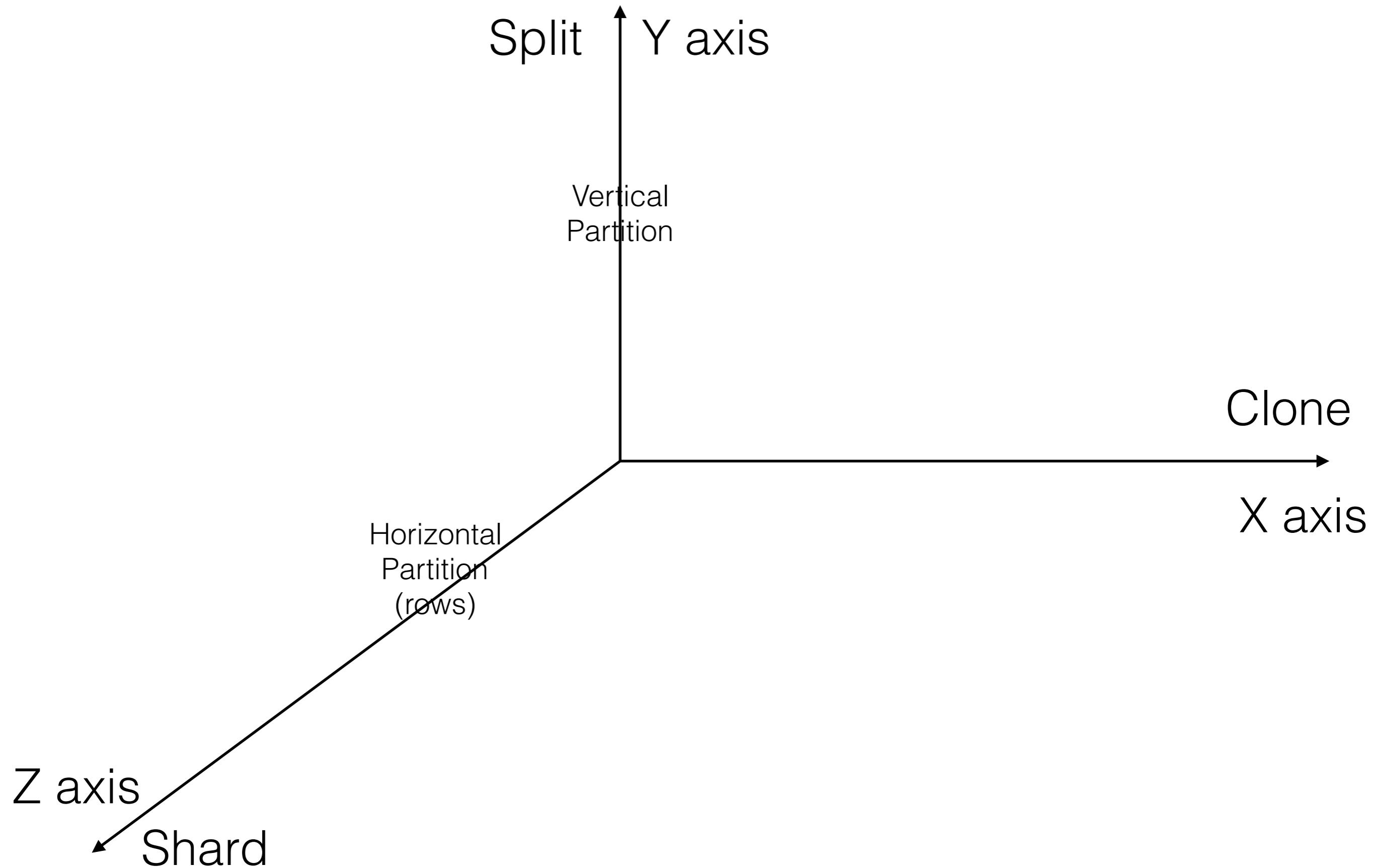


7. Debugging

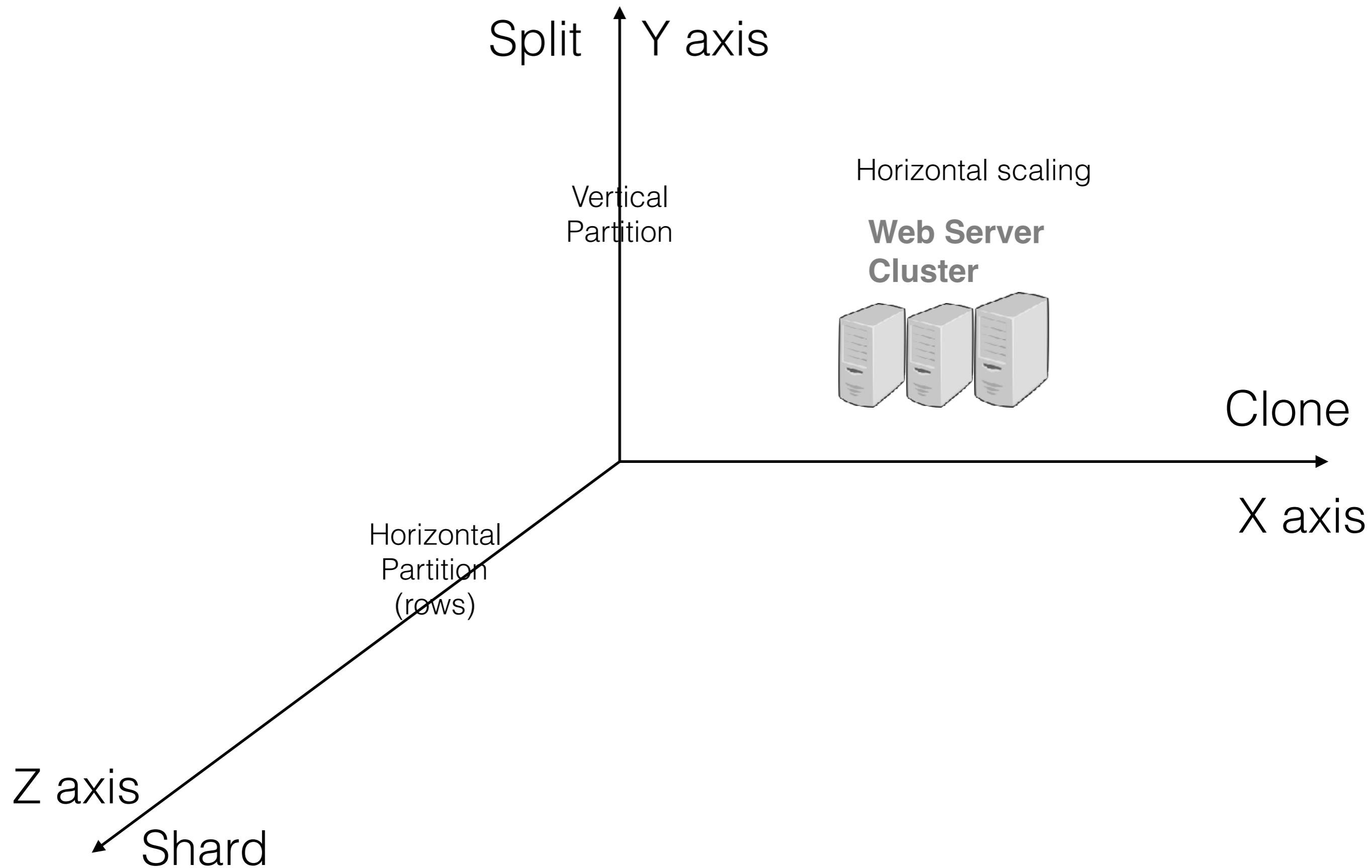


Scalability

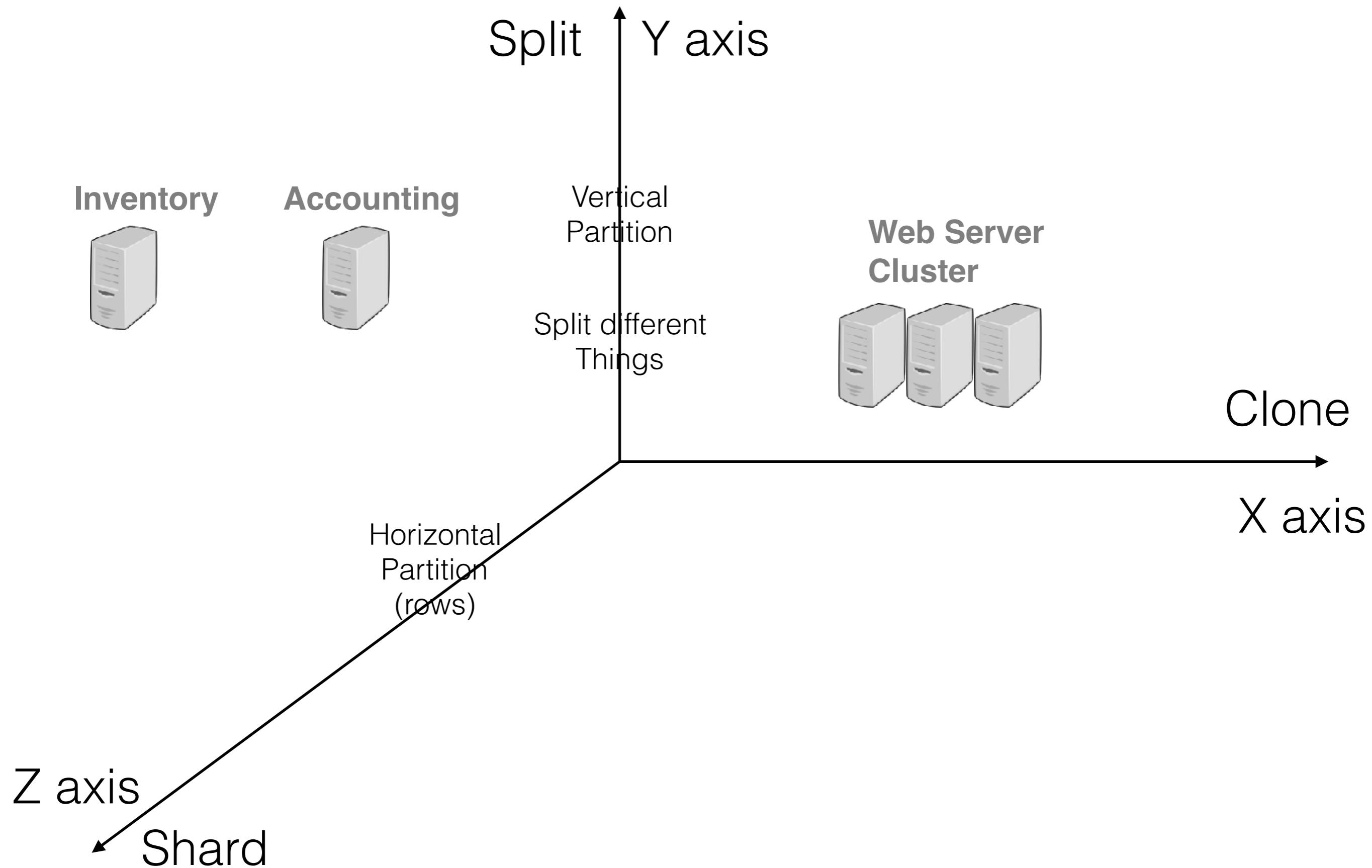
“Scalability Cube” - 50 rules for high Scalability



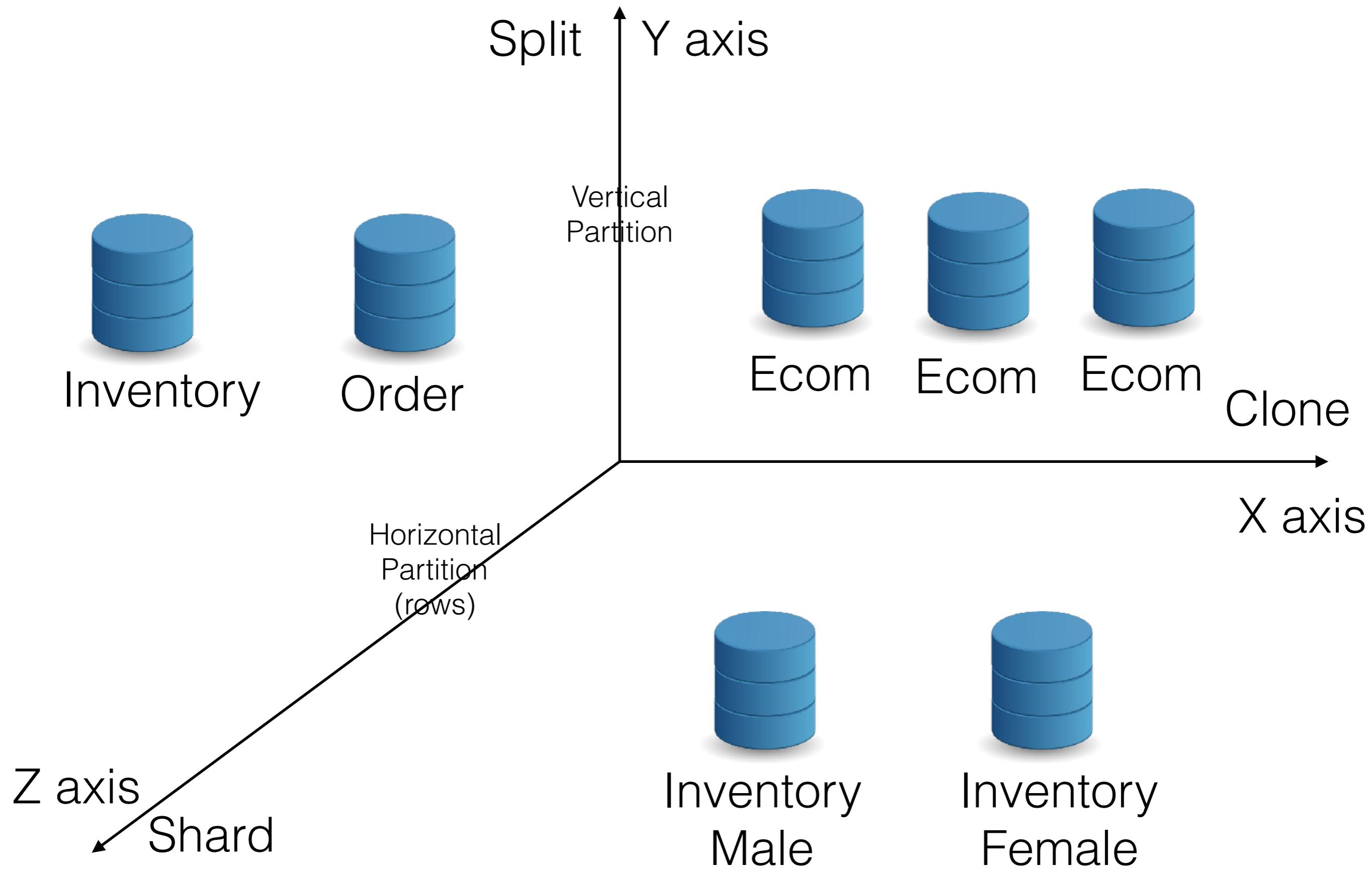
“Scalability Cube” - 50 rules for high Scalability



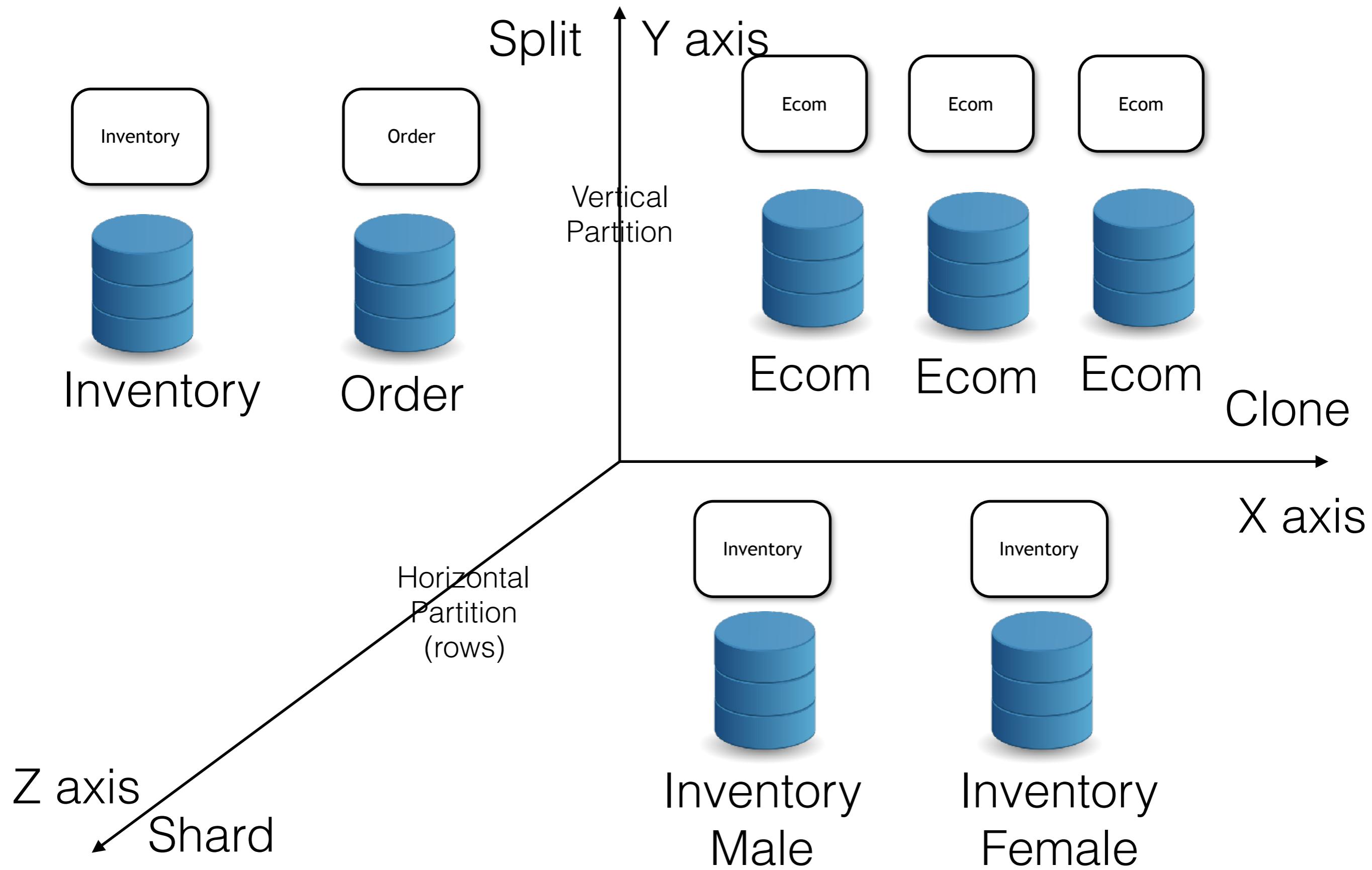
“Scalability Cube” - 50 rules for high Scalability



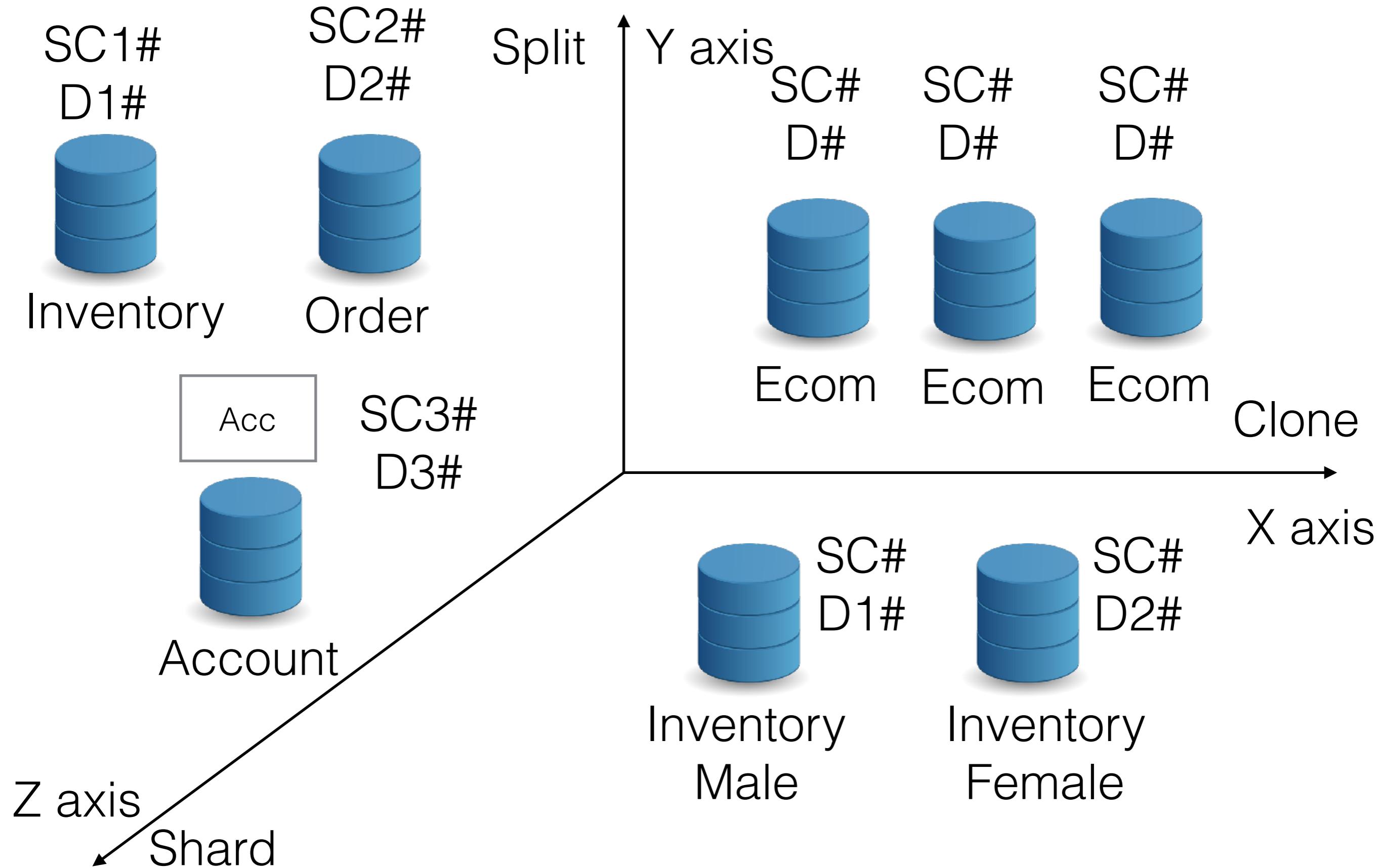
“Scalability Cube” - 50 rules for high Scalability



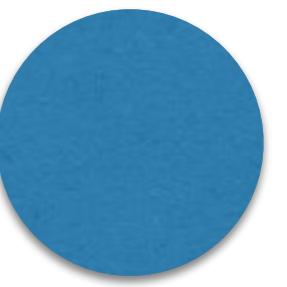
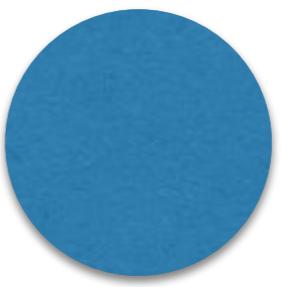
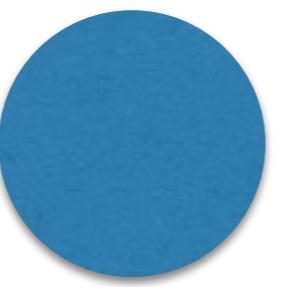
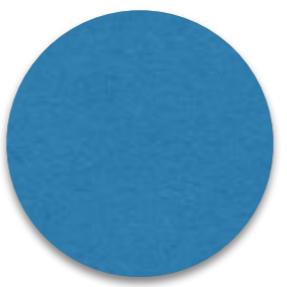
“Scalability Cube” - 50 rules for high Scalability



Scalability Cube - 50 rules for high Scalability



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

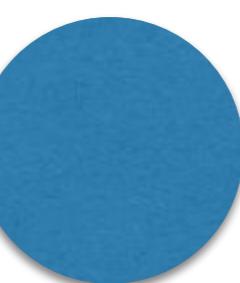
D BP



D BP

D BP

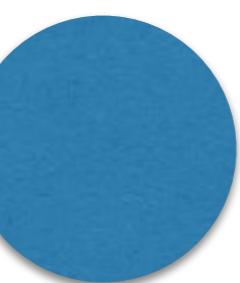
D BP



D BP

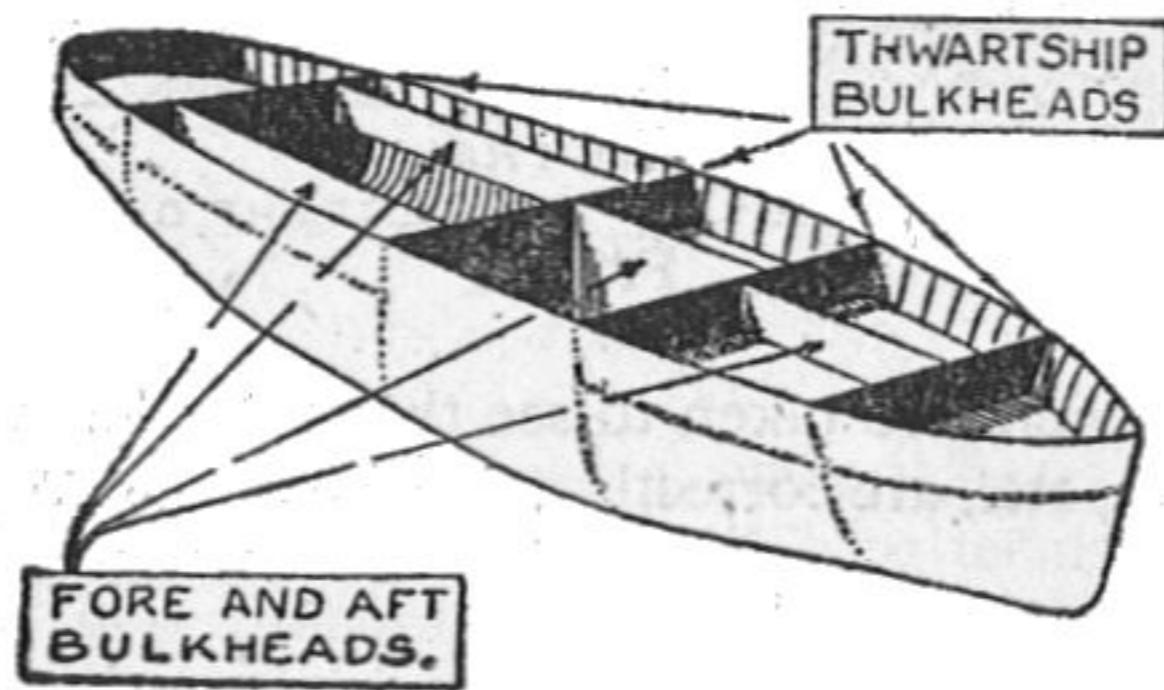
I BP

I BP



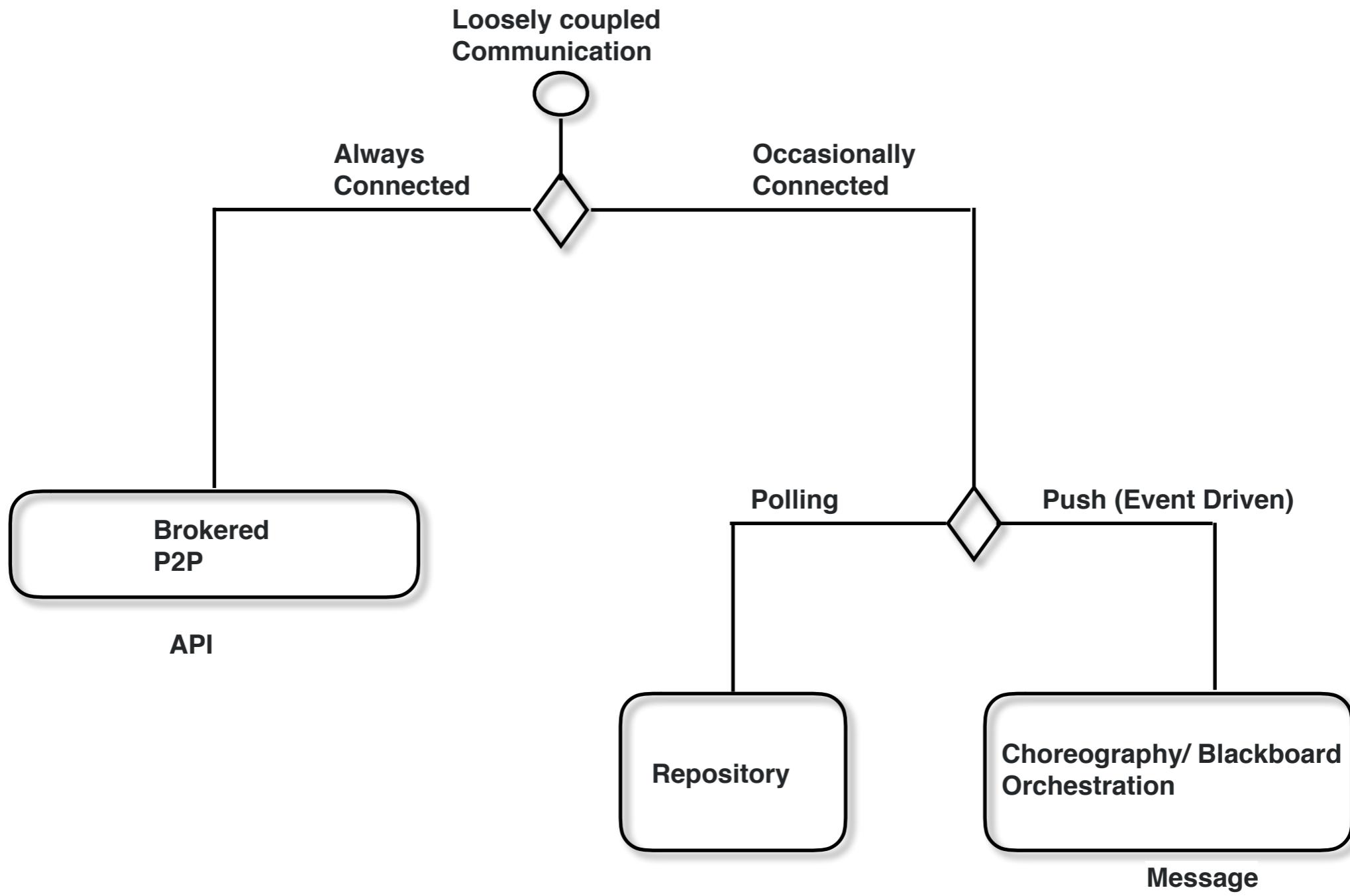
I BP

bulkhead

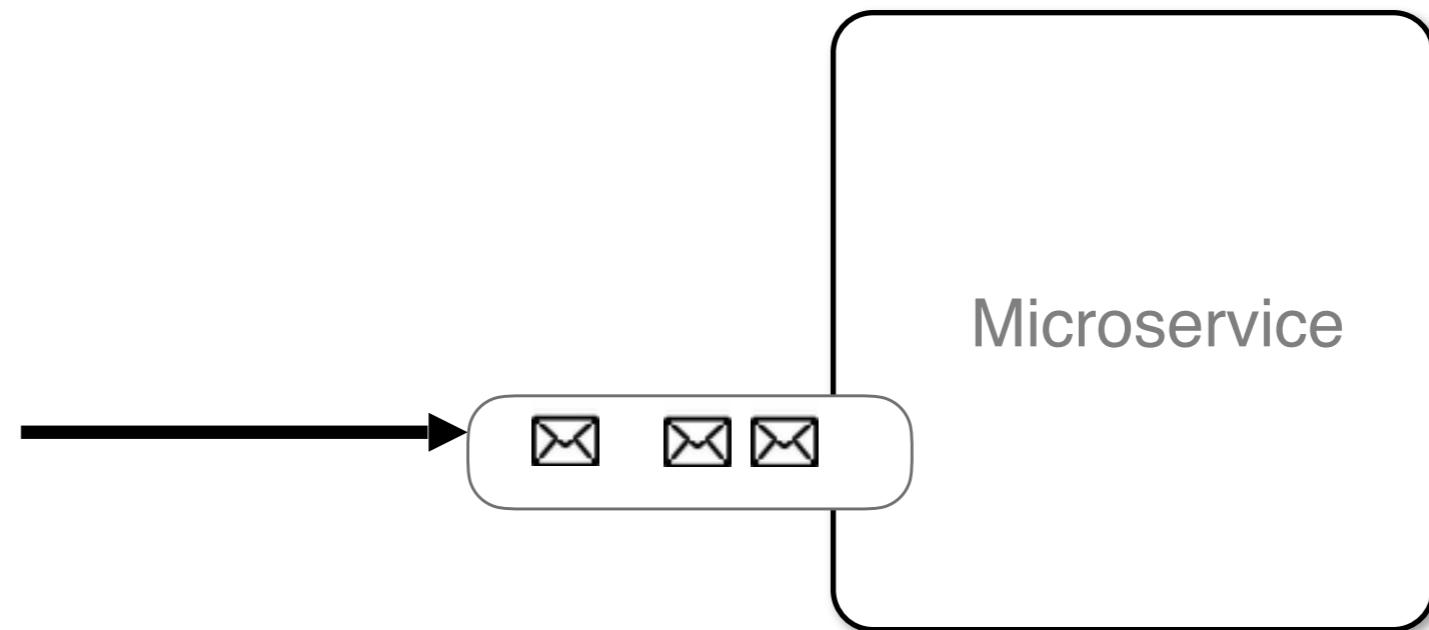


- In general, the goal of the bulkhead pattern is to avoid faults in one part of a system to take the entire system down. The term comes from ships where a ship is divided in separate watertight compartments to avoid a single hull breach to flood the entire ship; it will only flood one bulkhead.cir
- Partition service instances into different groups, based on consumer load and availability requirements.





* Message



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

* Message

API → Microservice

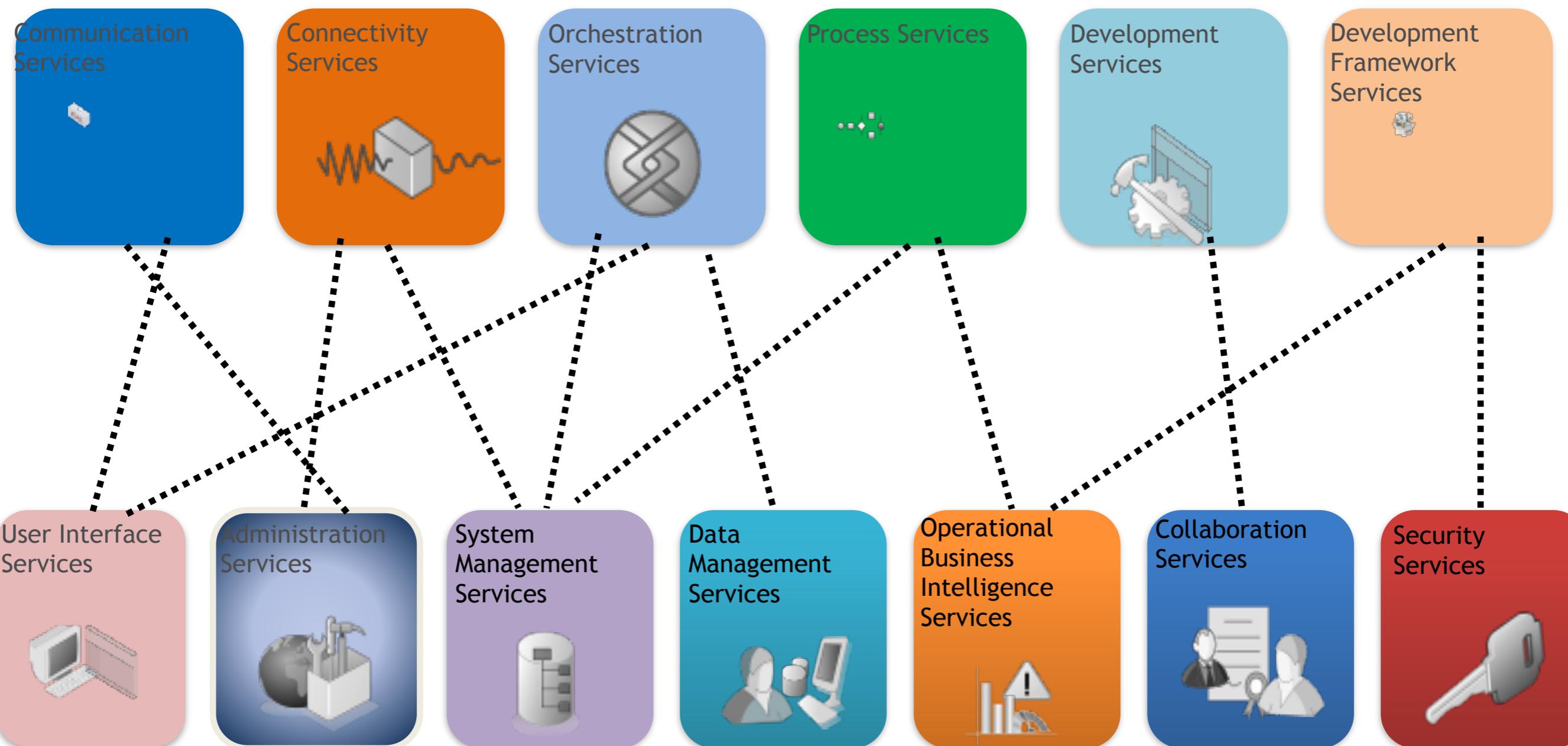
Rest, Grpc

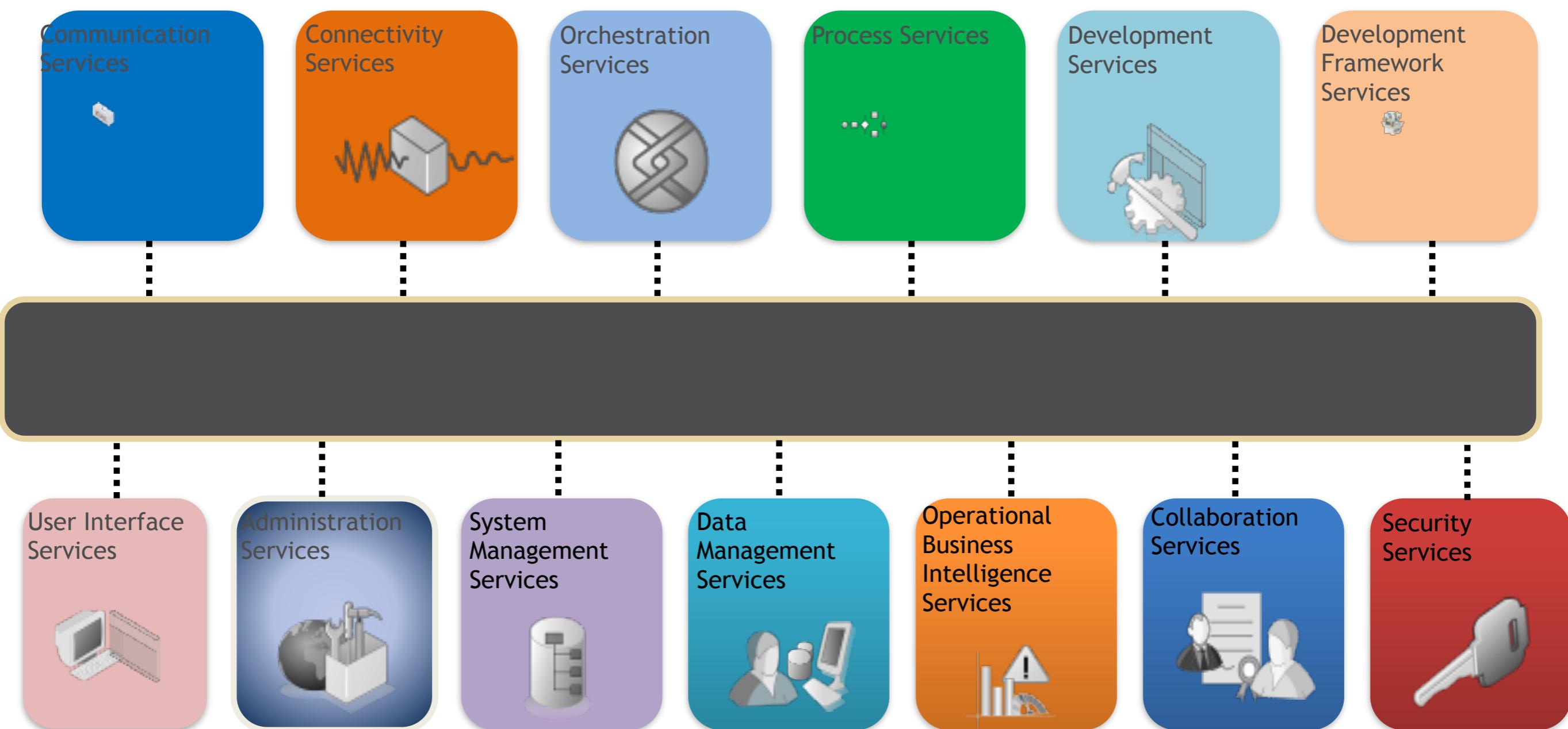


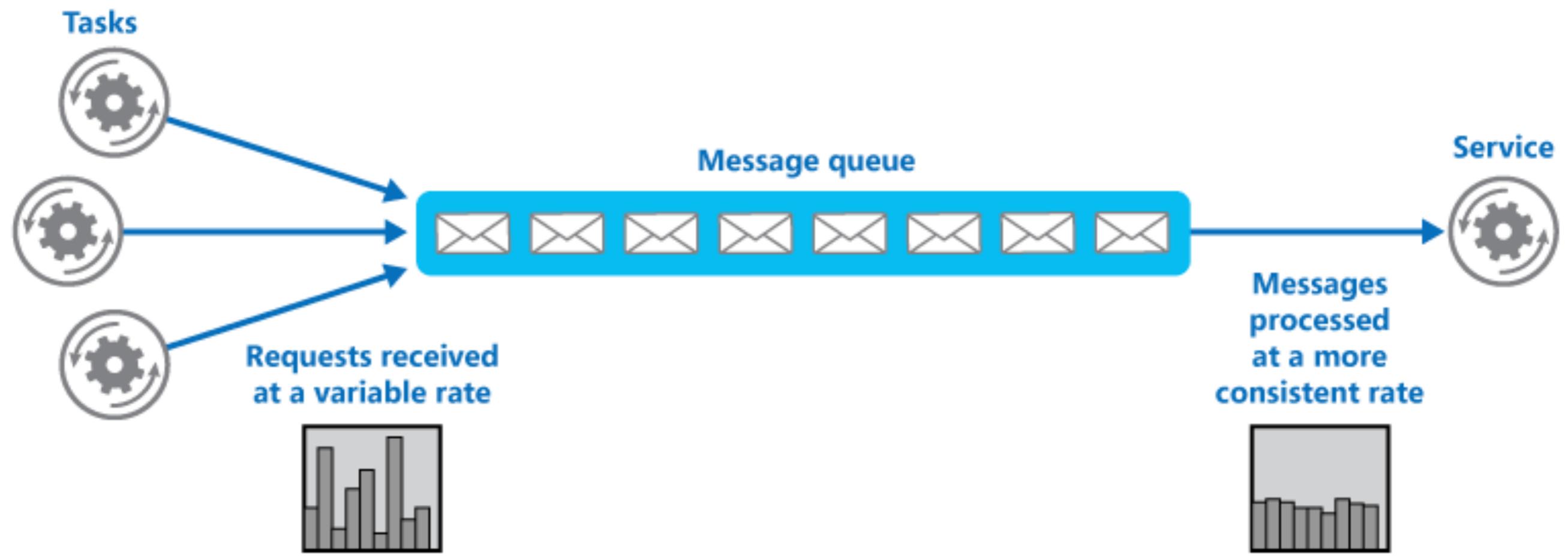
Microservice

Queue, Topic, Hub

Concerns	API	Message	
Ordering	Ordered	Unordered	
Duplicate	No	Yes	Idempotency
Protocol	2 way	One way	
Resilience	No	Yes	
Occasionally connected	Always connected	Yes	
Scalability	--	++	
Consistency	Immediate (++)	Eventual (- -)	
Load Leveling	--	++	
Low Coupling	--	++	
Distributed Comm Patterns	--	++	



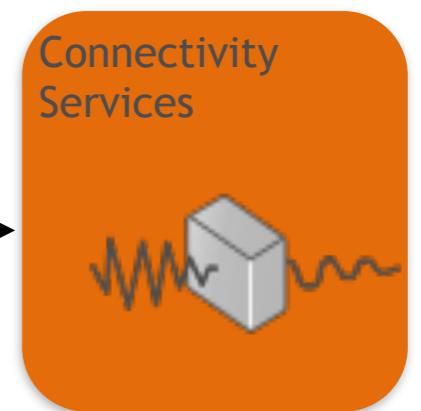






Communication
Services

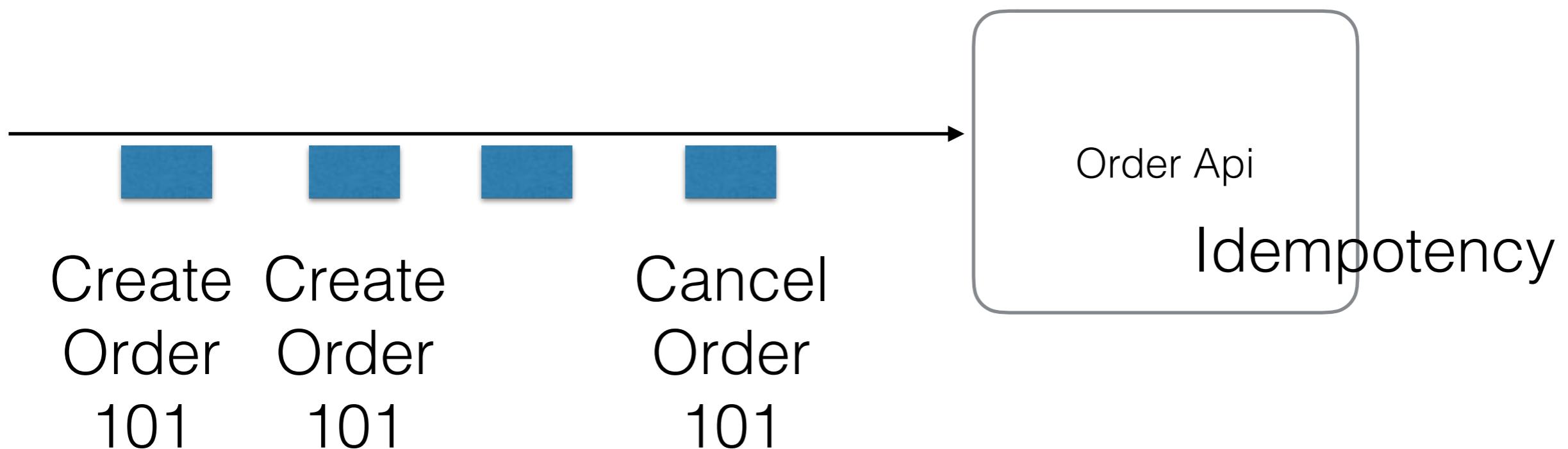
Connected protocol
(REST, WS, GRPC , Thrift)



Connectivity
Services

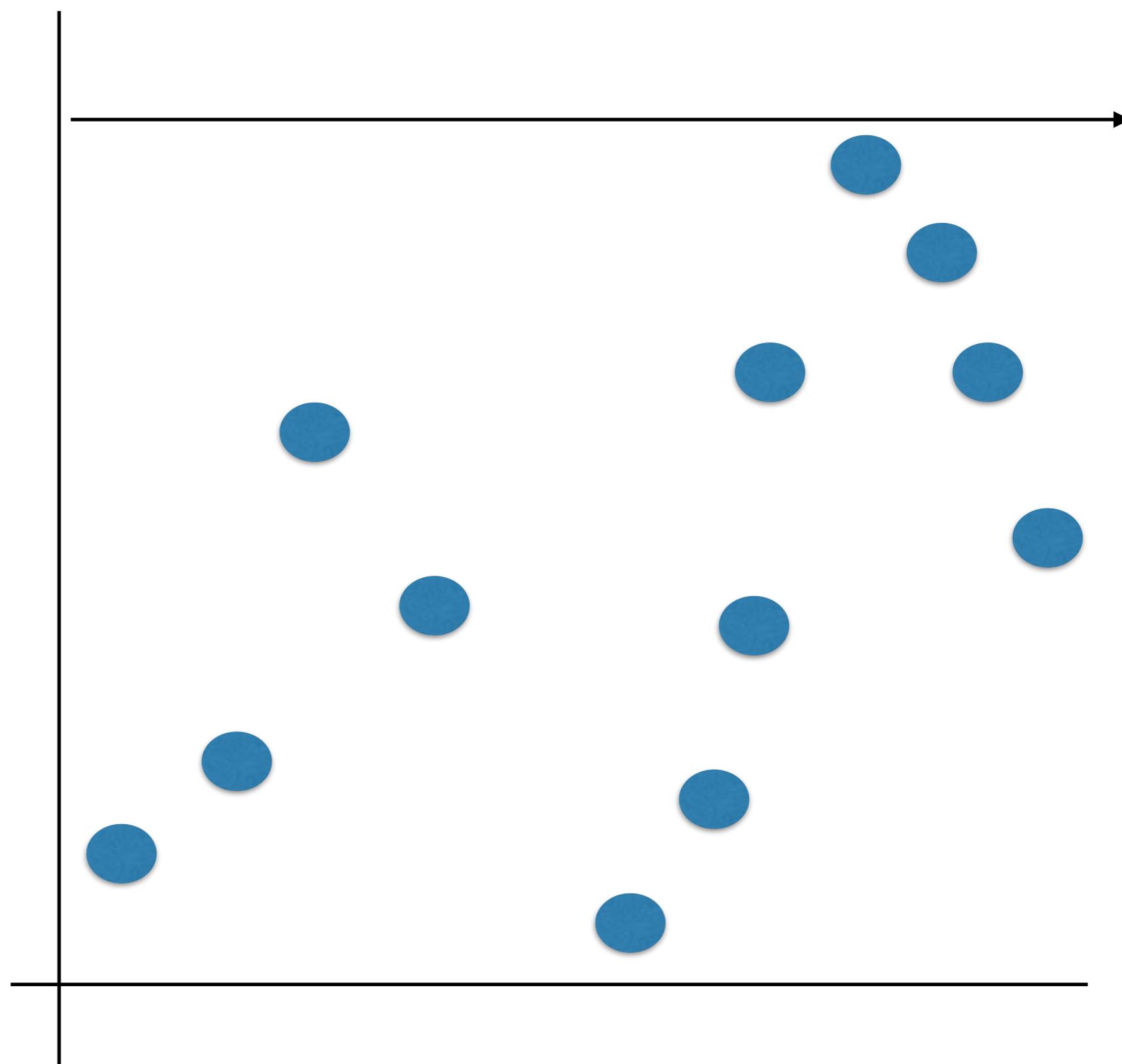
Message protocol
(AMQP, MQTT, ...)

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

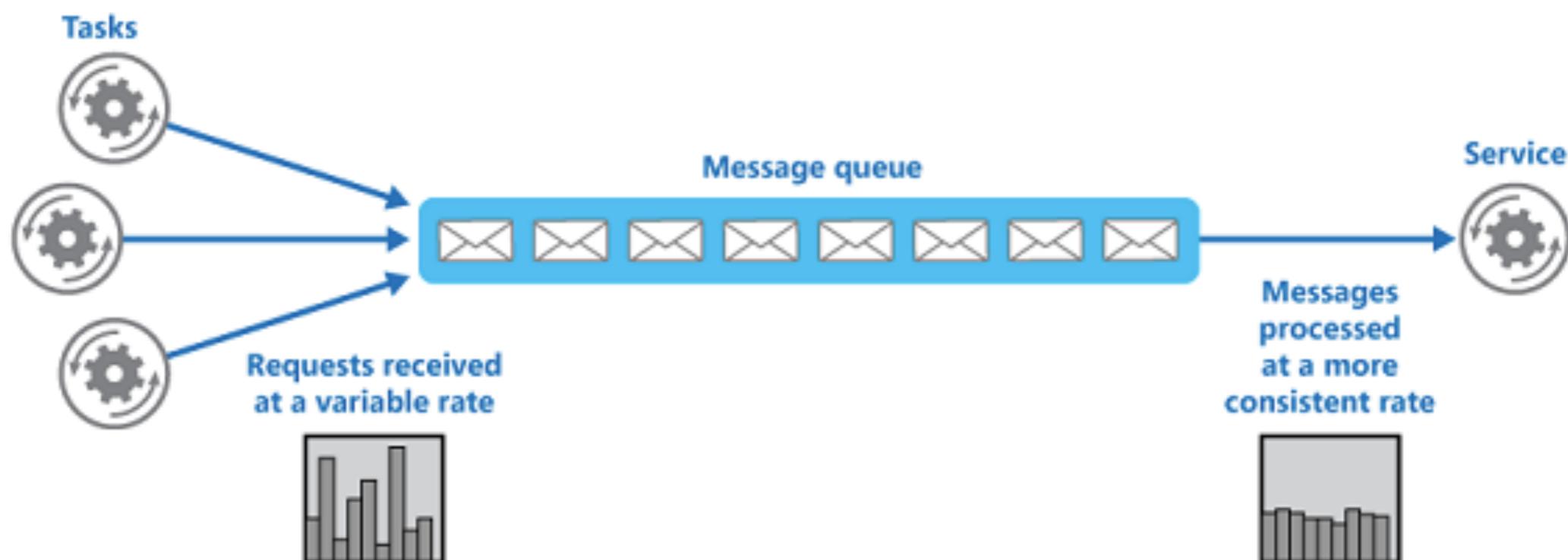


Request

Time

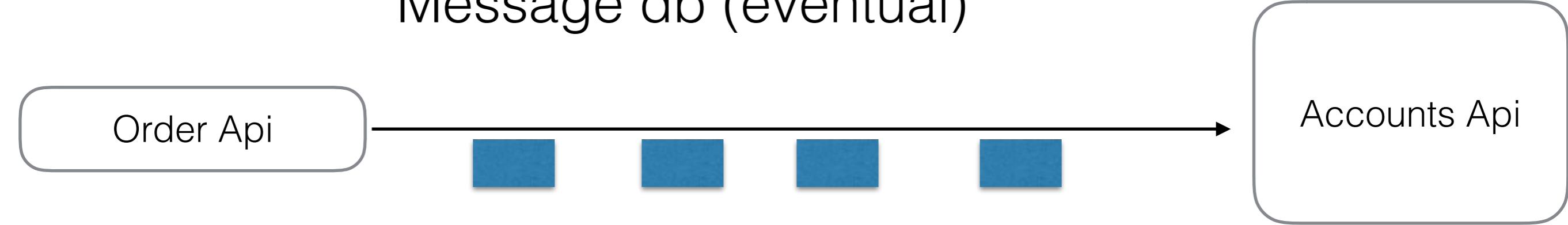


Queue-based Load Levelling



source:msdn

Message db (eventual)



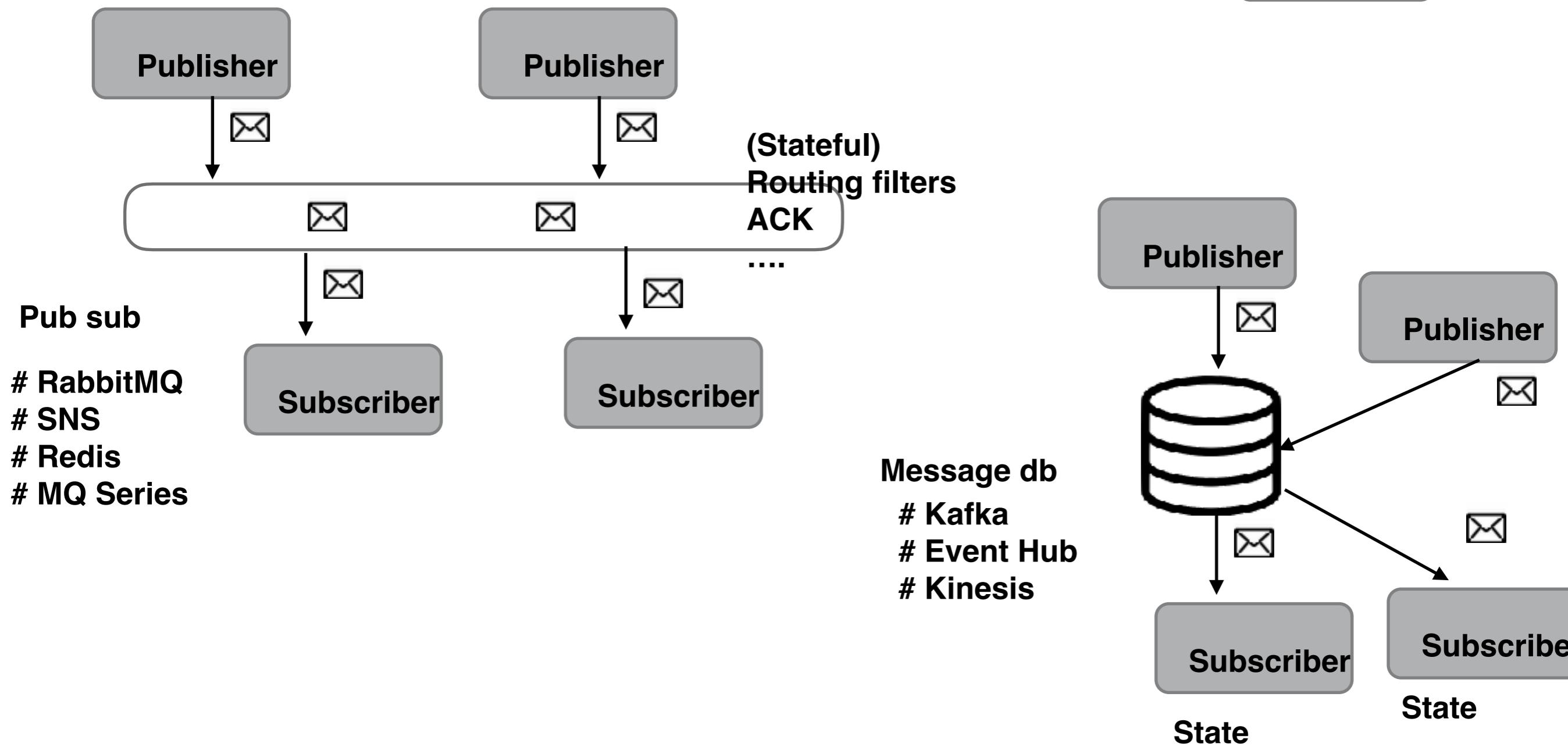
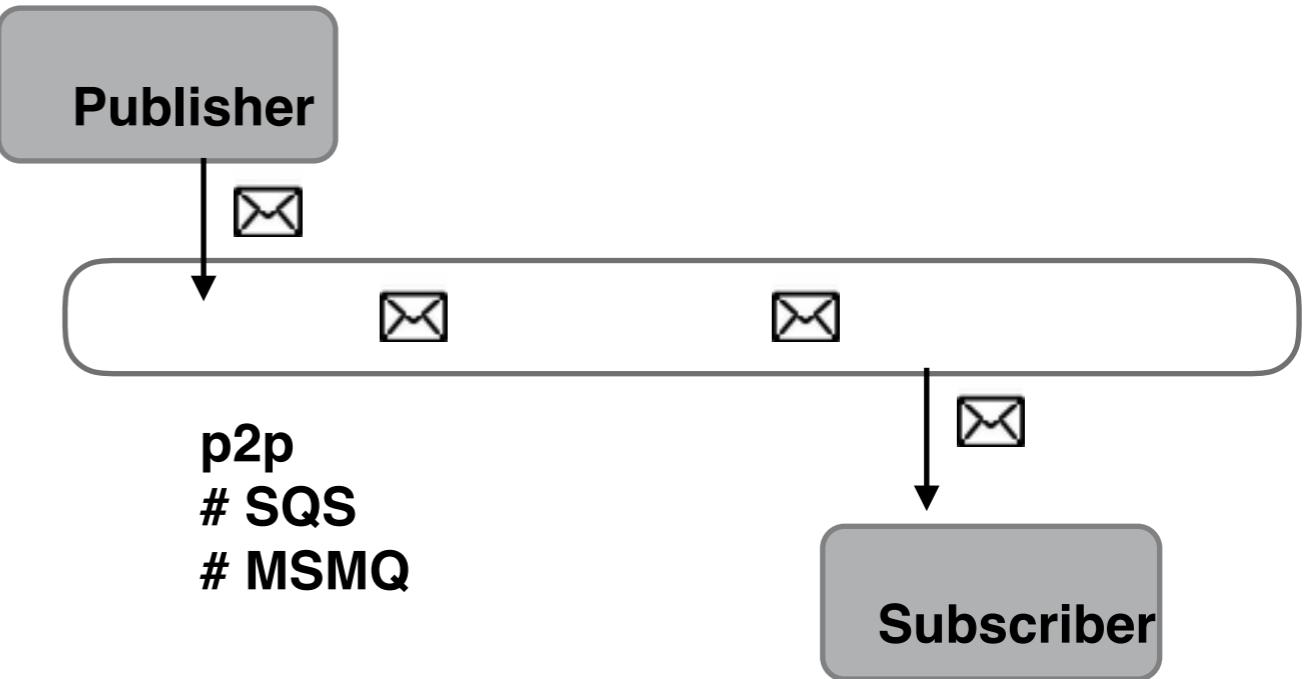
`msg = getmessage()`

...

...

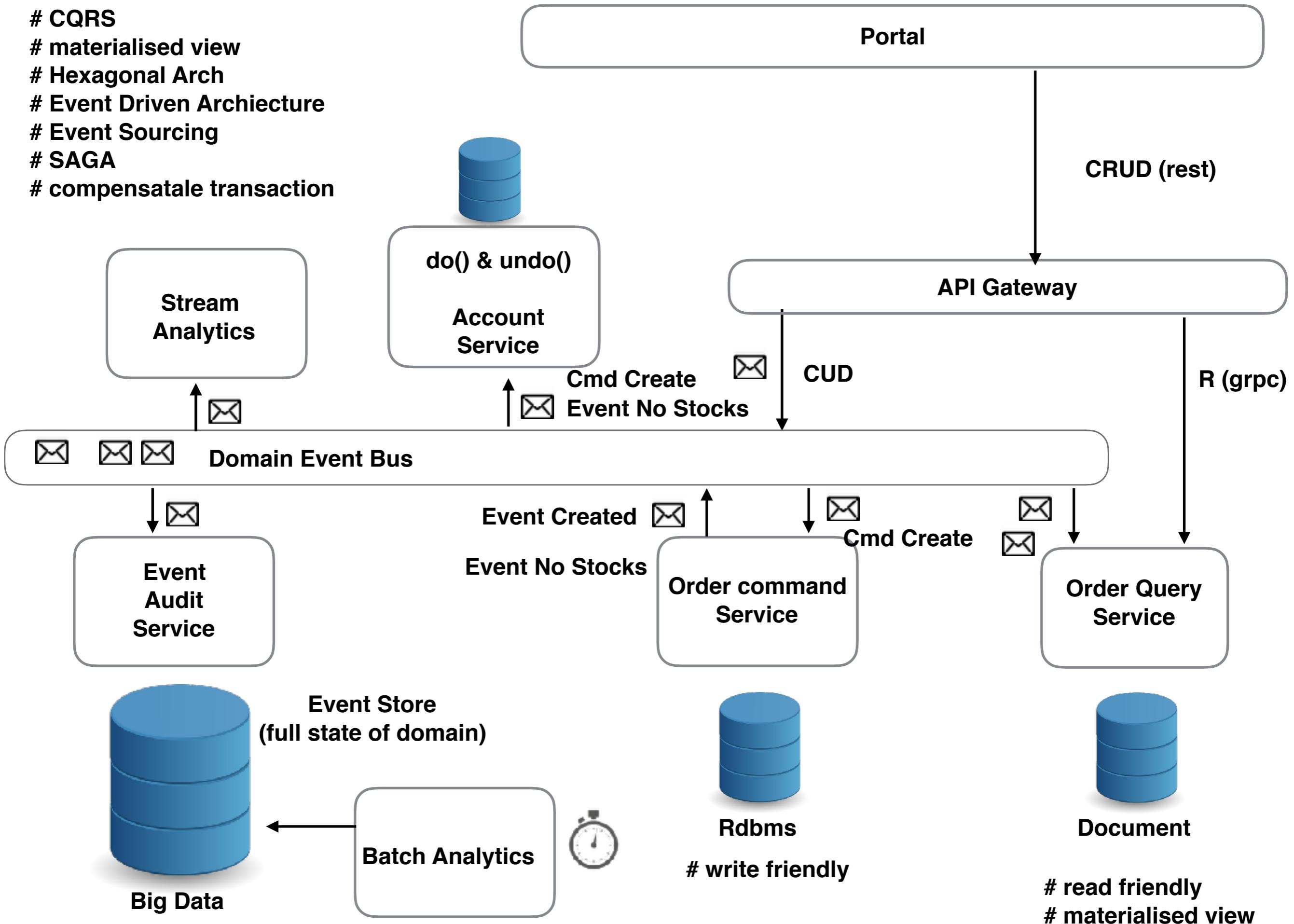
`msg.ack()`

1. Message Queue
2. Message Bus (Topic)
3. Message hub



Reference Arch

CQRS
materialised view
Hexagonal Arch
Event Driven Architecture
Event Sourcing
SAGA
compensatale transaction



**Order Query
Service**

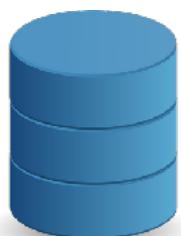
Message Service

API Service

Facade/ Control

Repository

Domain



Boundary (technology)

Control

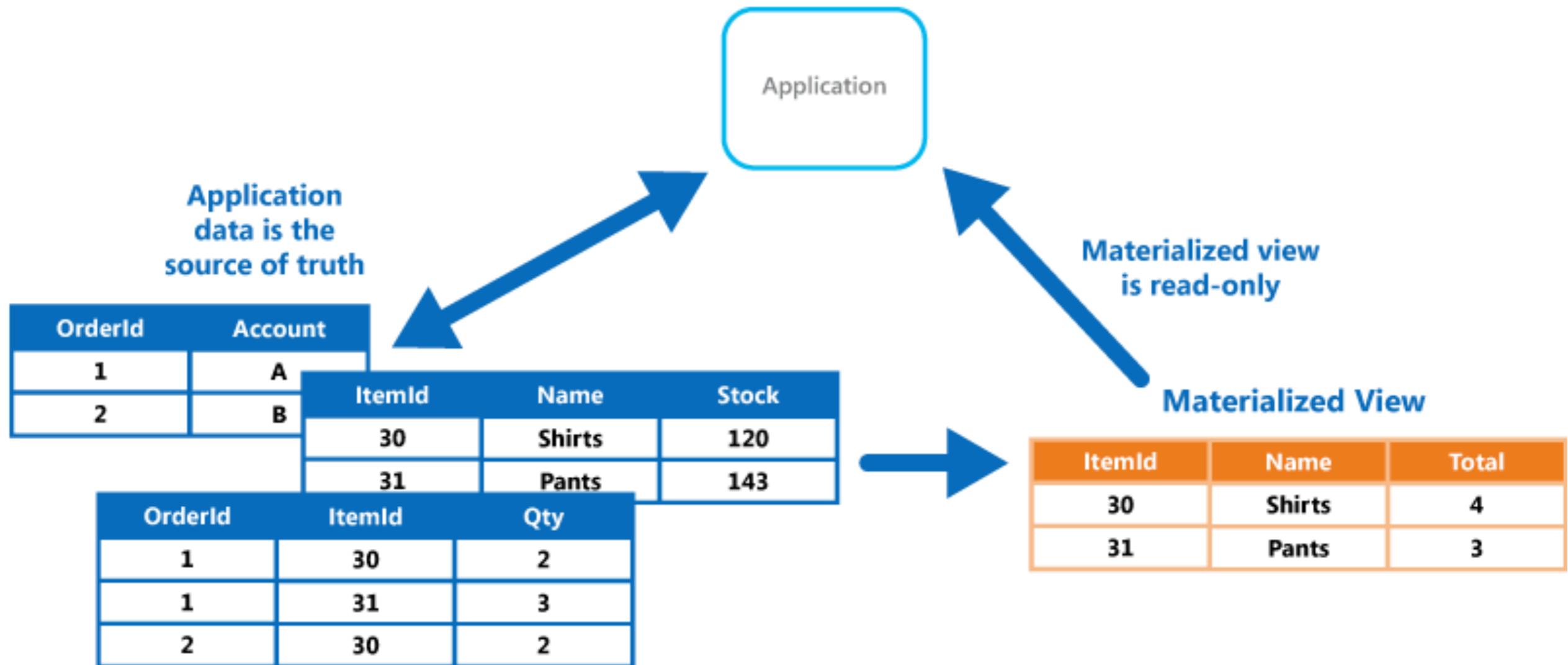
Entity (domain)

Ui

API

Msg





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

DNF

<— duplicates

<— no joins

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

3NF

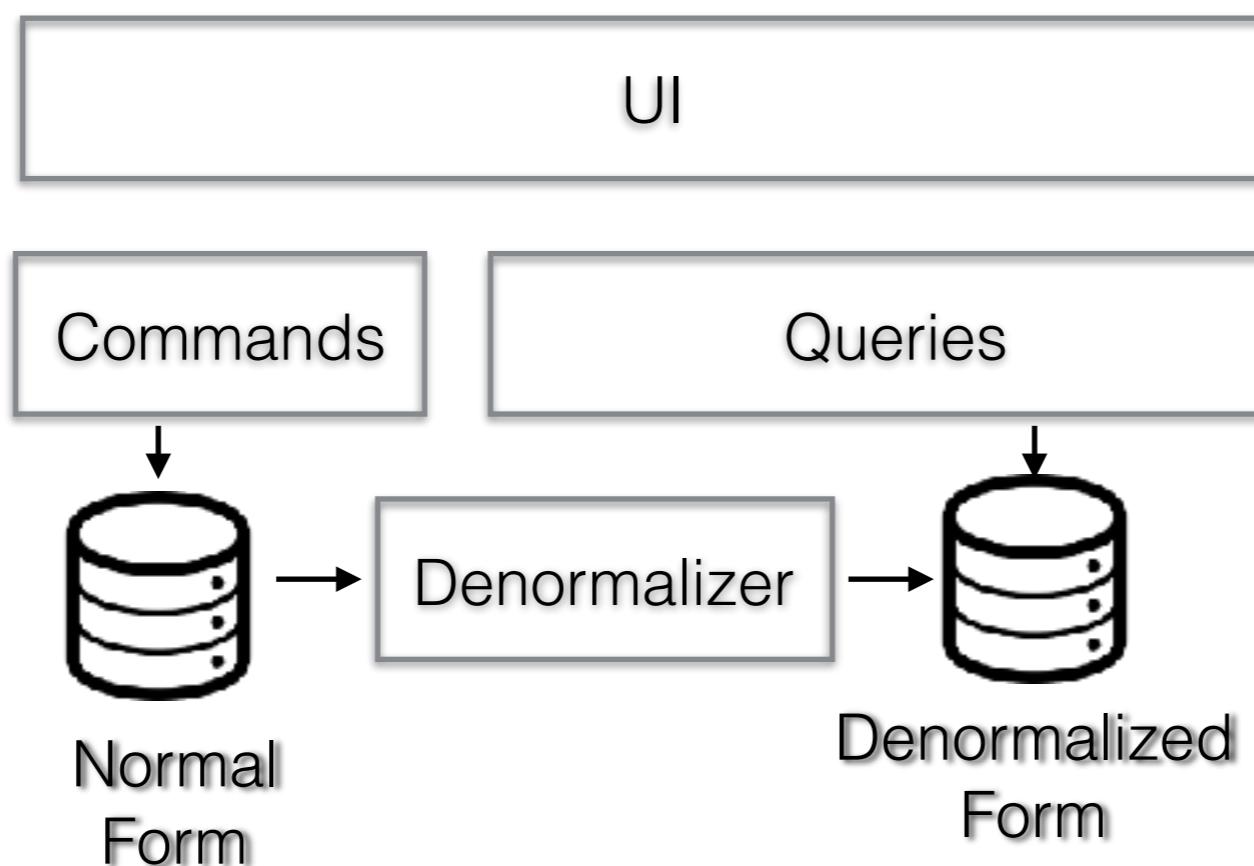
TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

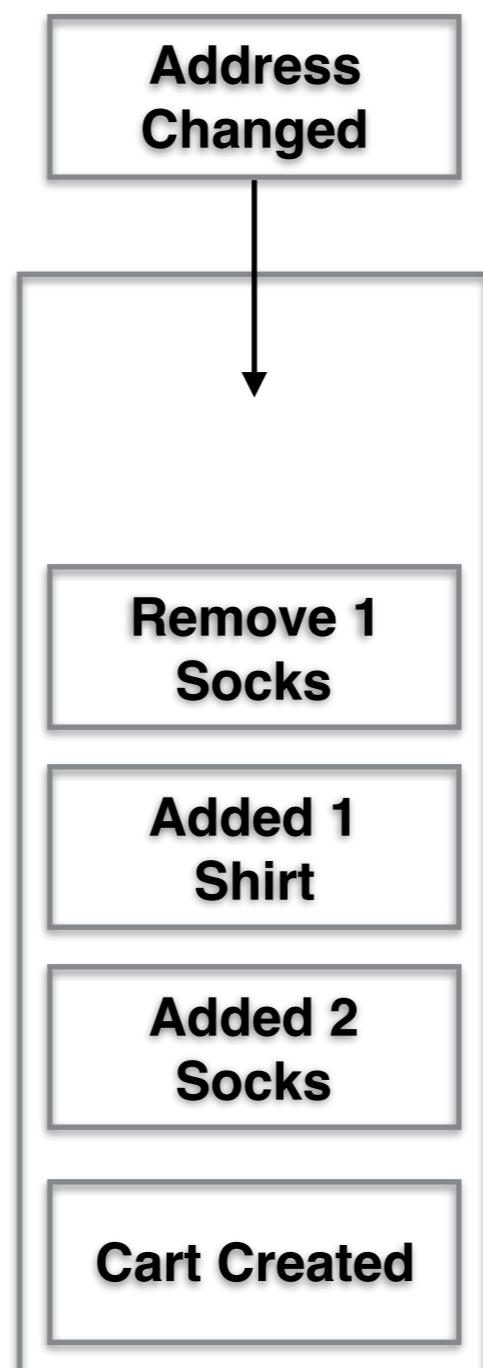
<— no duplicates

<— more joins

CQRS

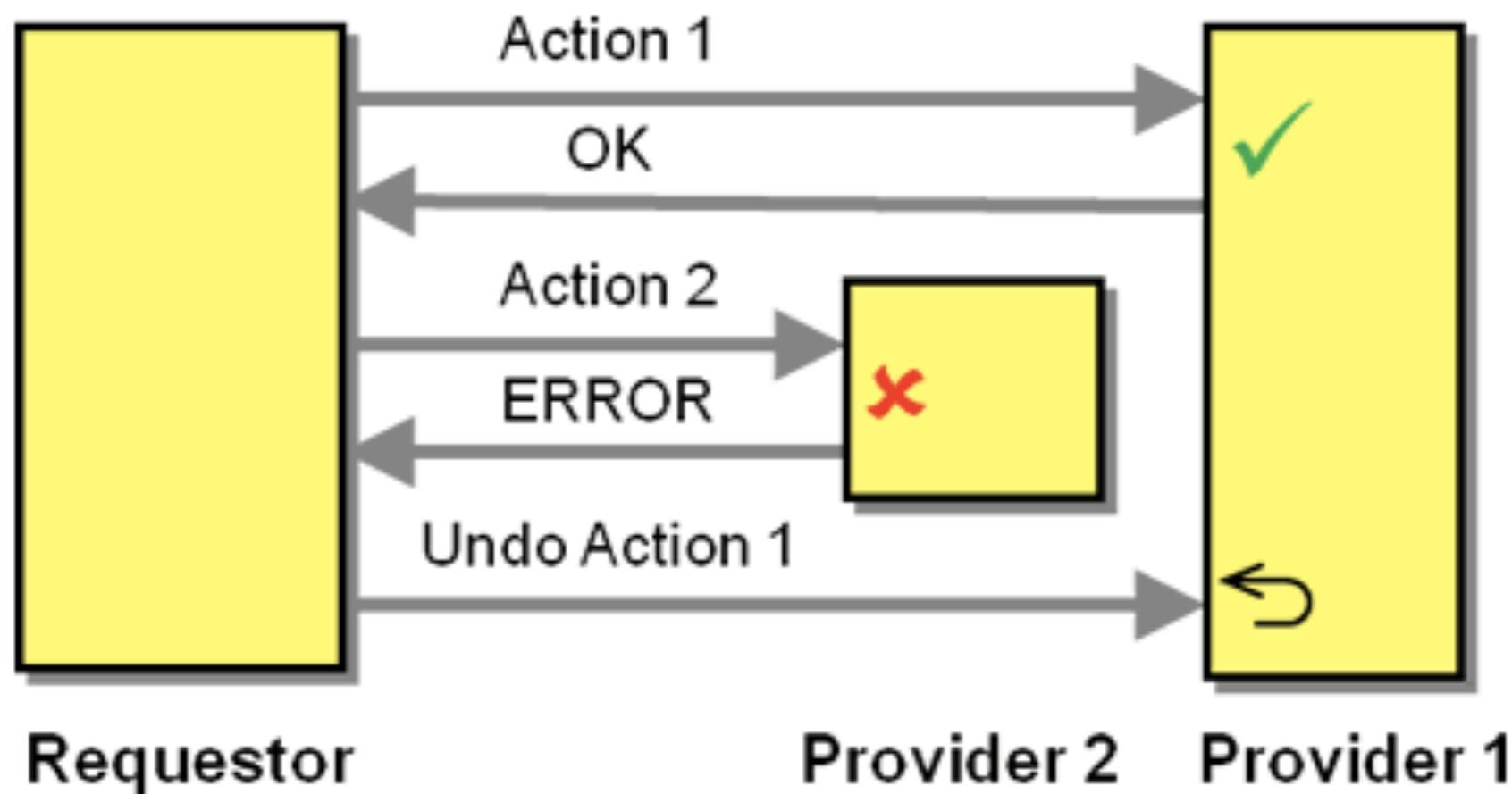


Event Source



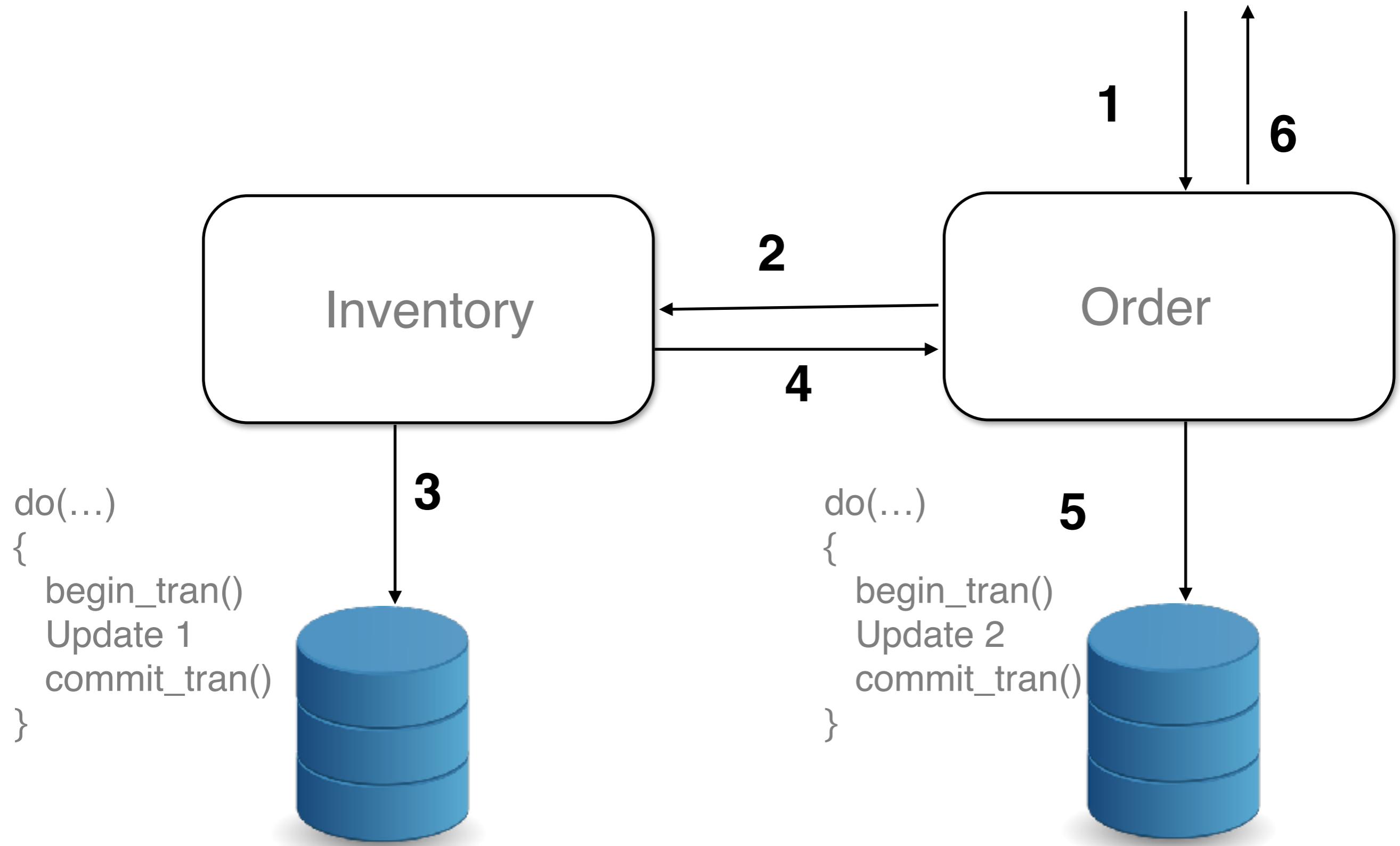
Event Store

Saga

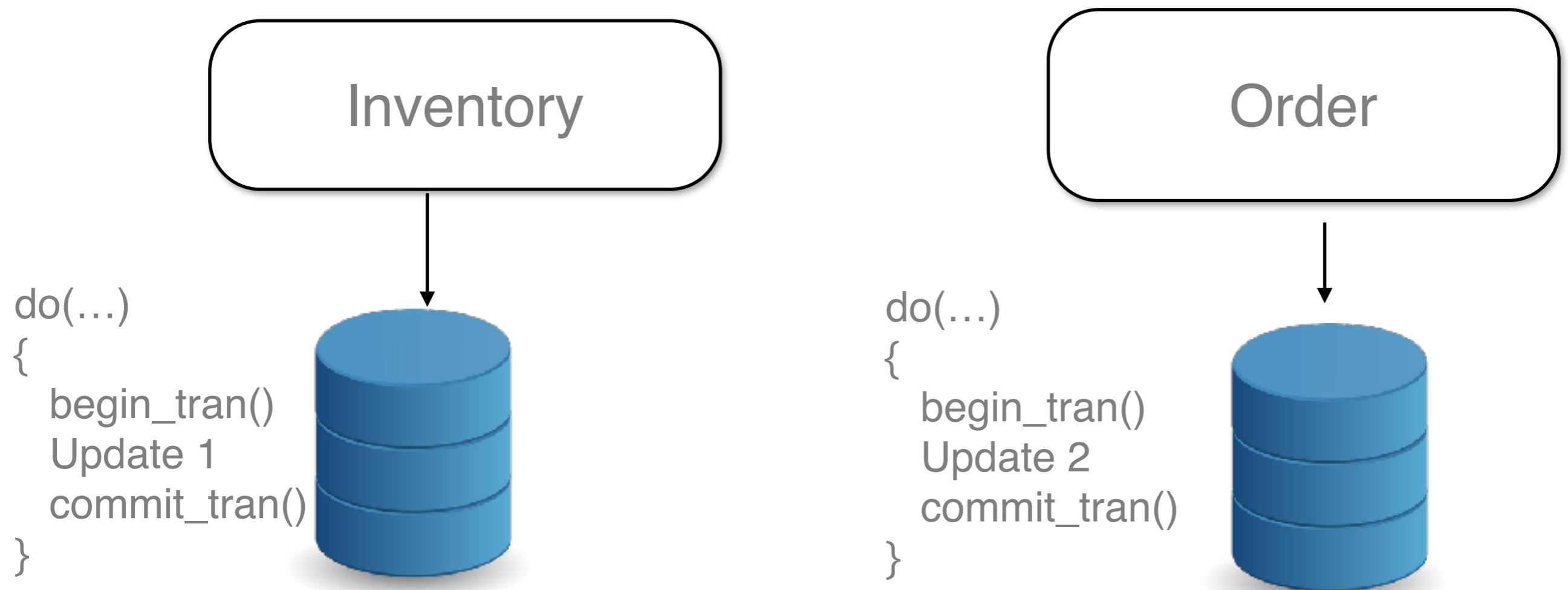


Saga Pattern is a direct solution to implementing distributed transactions in a microservices architecture.

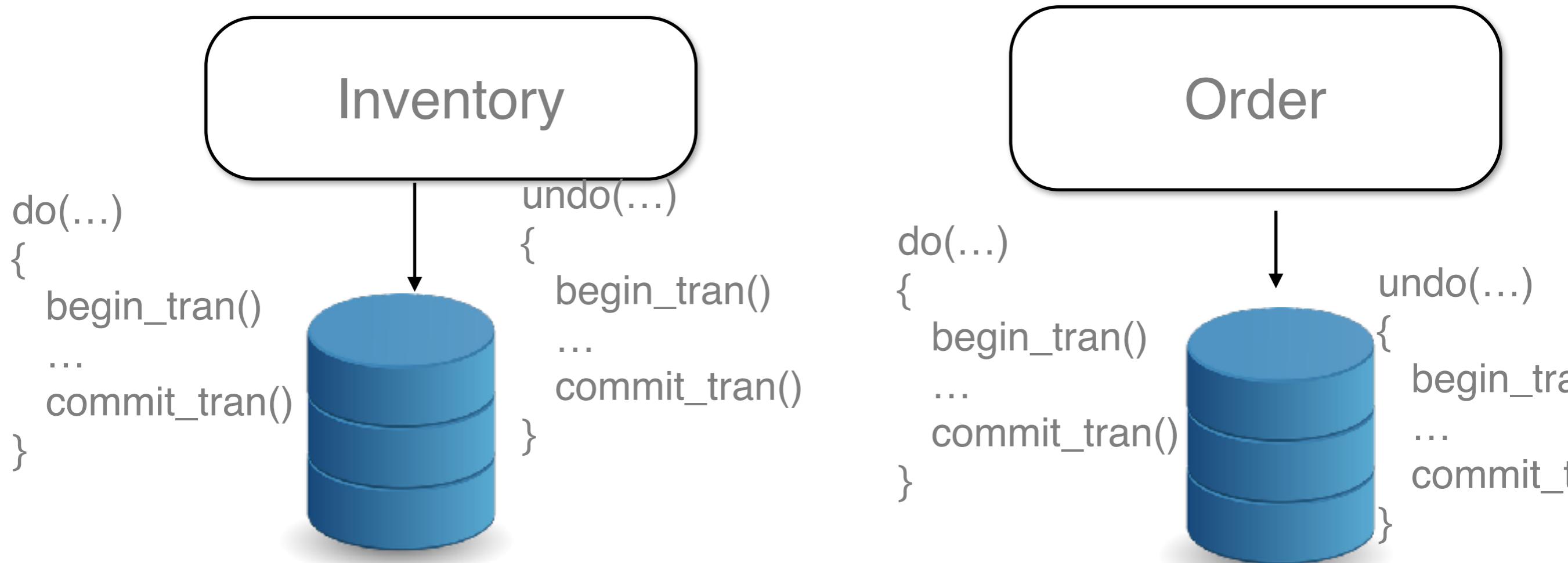
Transaction

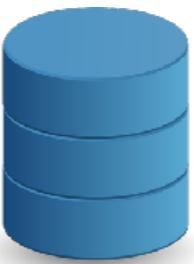


SAGA pattern

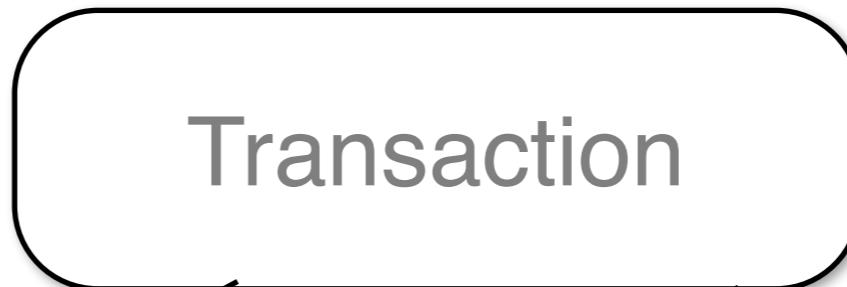


Compensatable Transaction

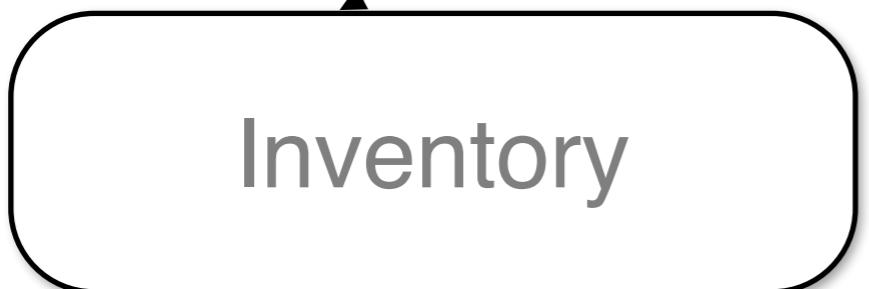




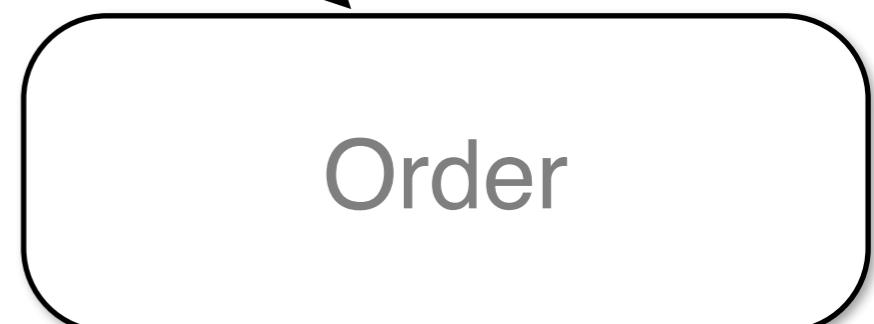
Tranid,order,inventory, ...
101, y, y,



2



5



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



3

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

6

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



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

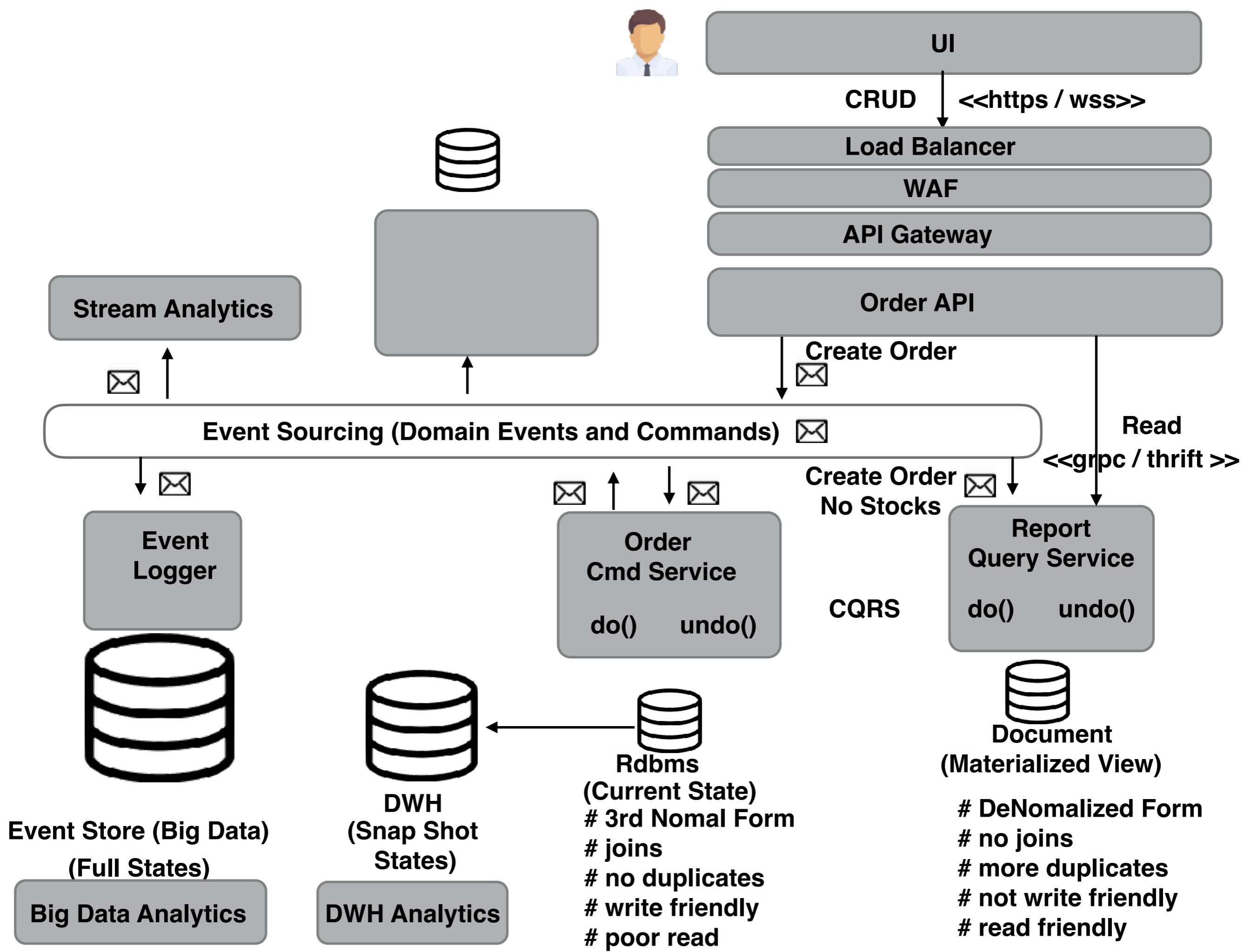
ACID

BASE

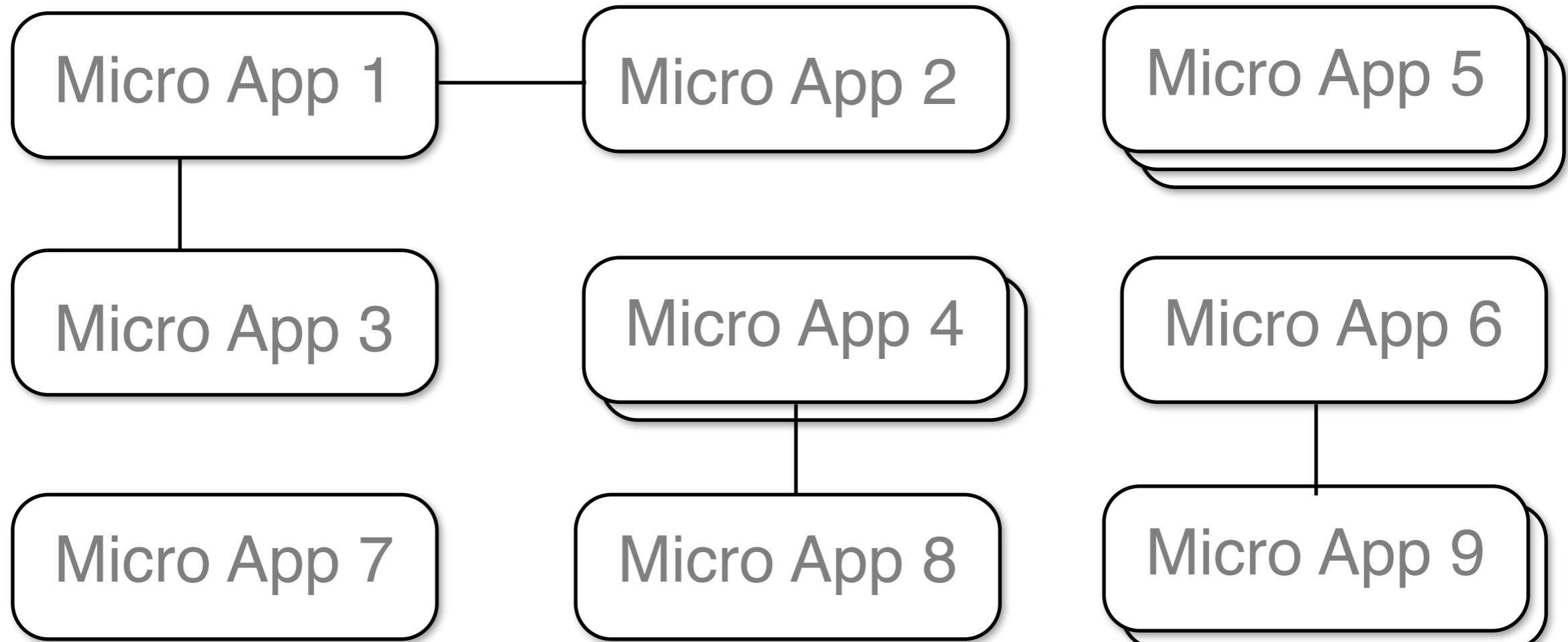
Immediate
Consistency

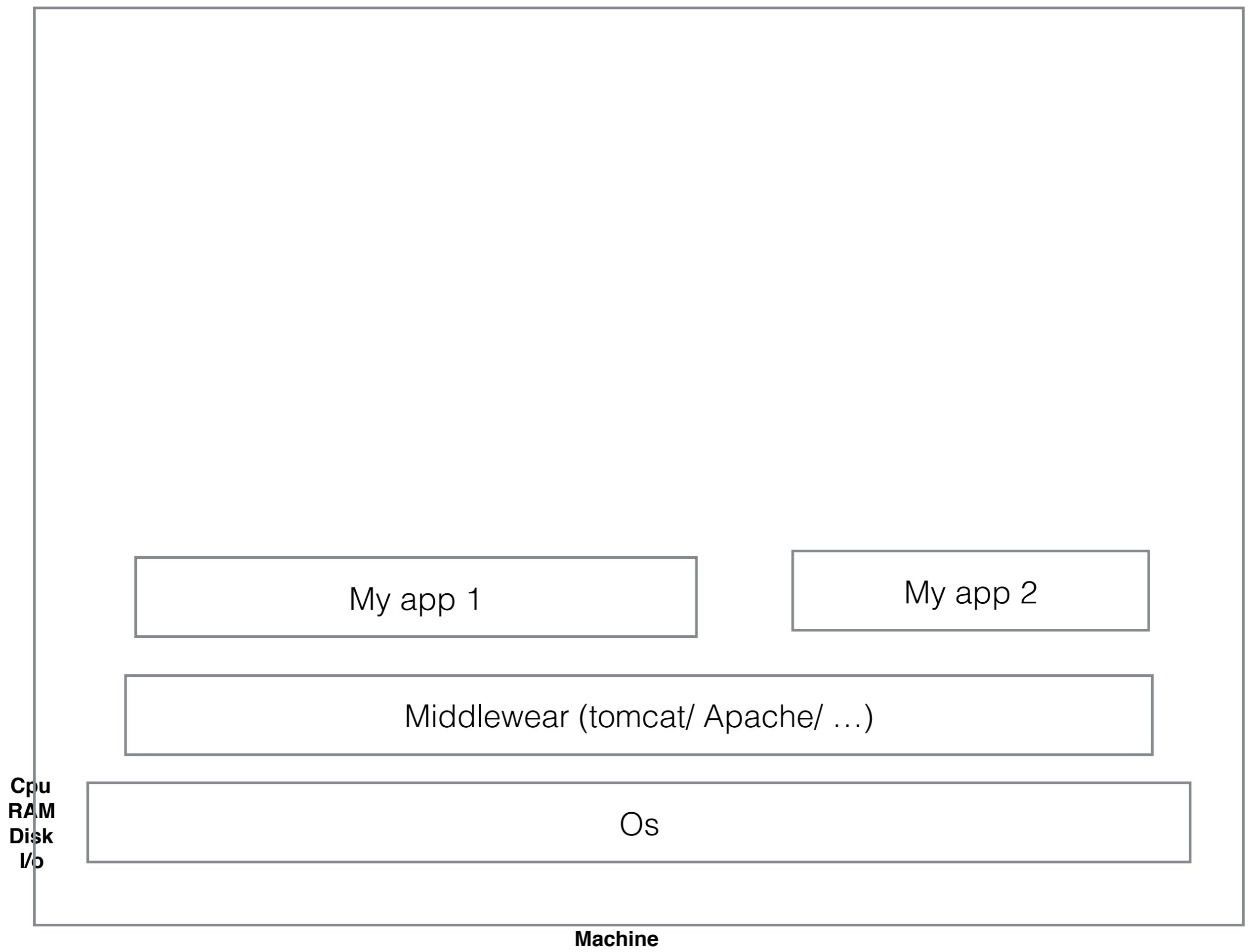
Eventual
Consistency





Isolated Env.





VM1

VM2

My app 1

Middlewear(tomcat/apache/...)

Unix Guest Os

My app 2

Middlewear

Win Guest Os

Cpu
RAM
Disk
I/o

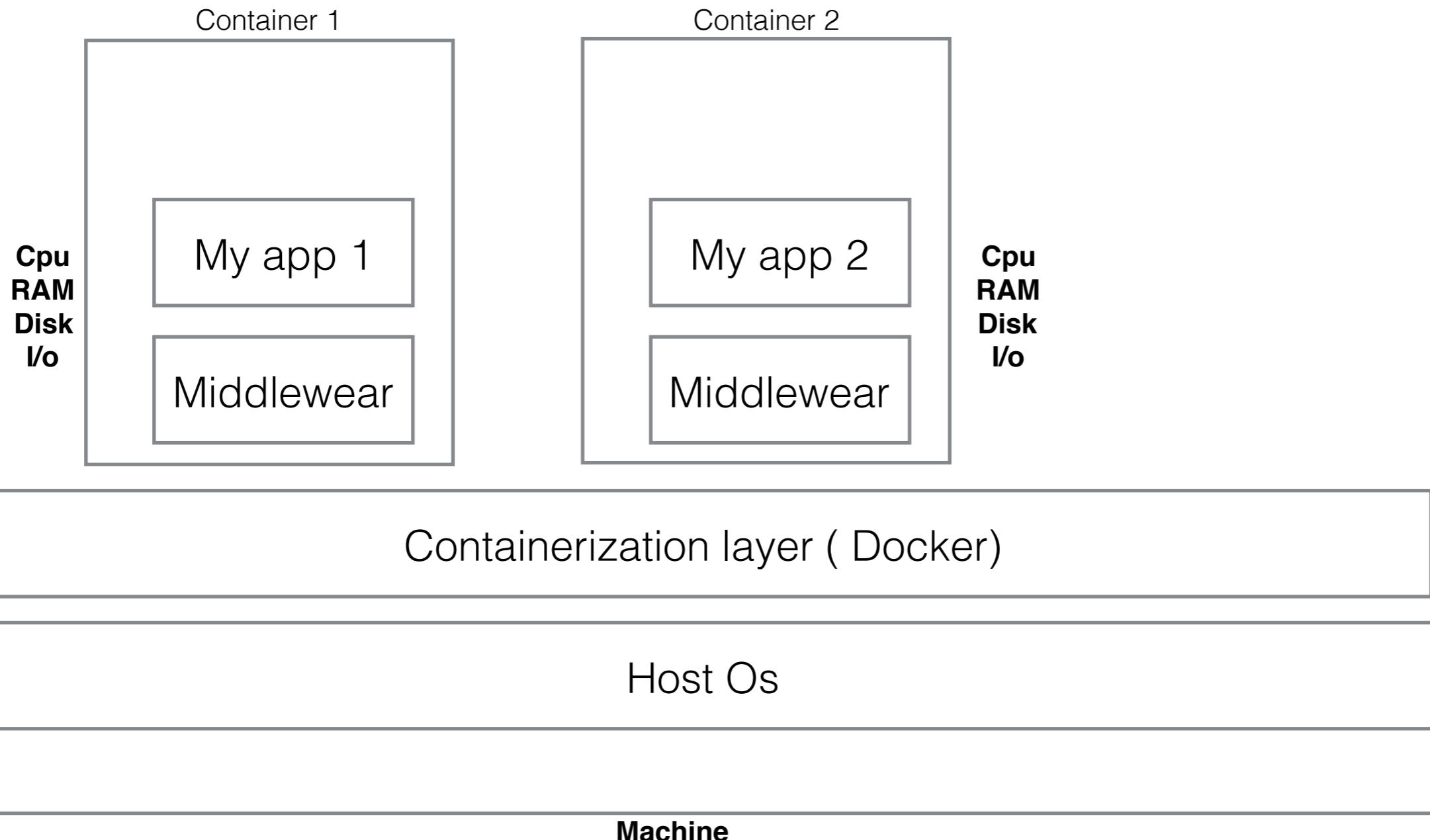
Cpu
RAM
Disk
I/o

Virtualisation layer

Host Os

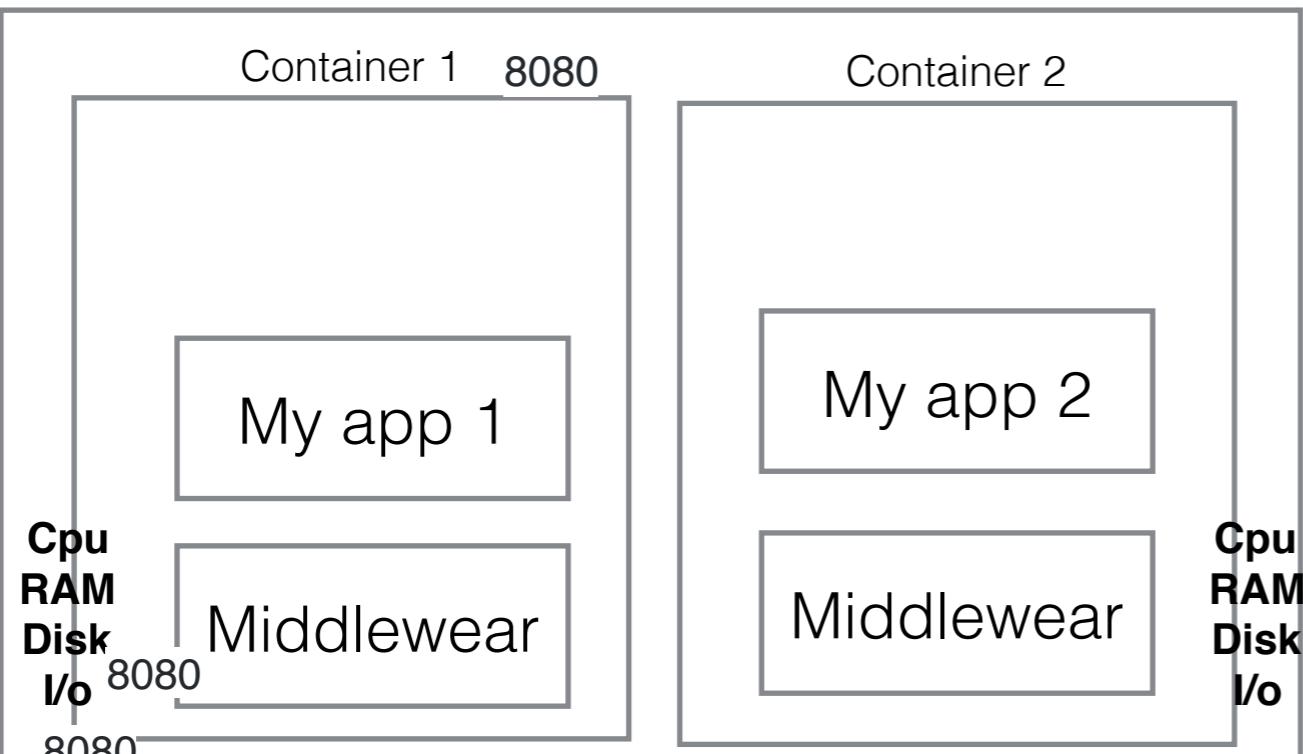
Cpu
RAM
Disk
I/o

Machine



VM1

VM2



Containerization layer (Docker)

Unix Guest Os

Win Guest Os

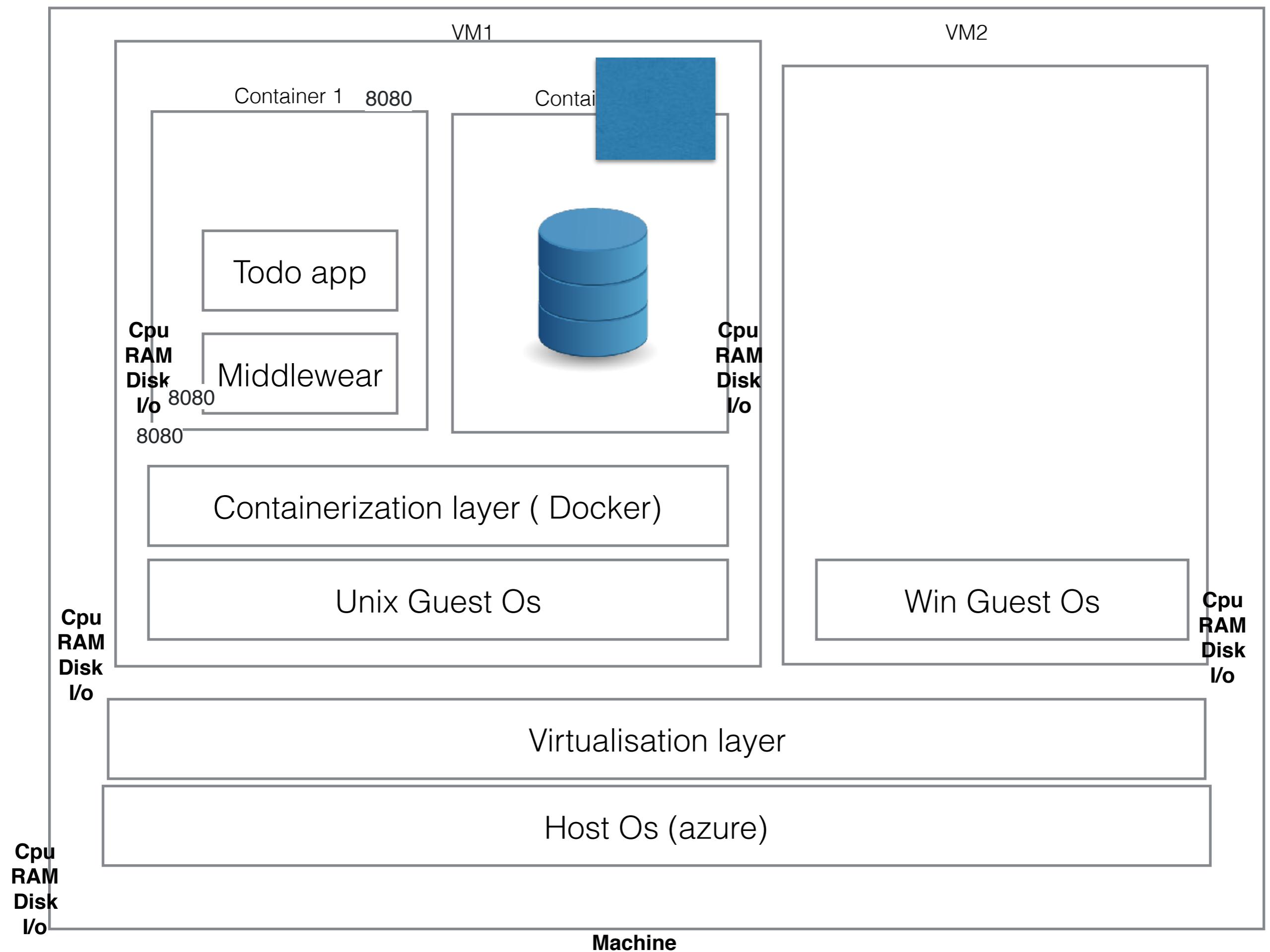
Cpu
RAM
Disk
I/o

Virtualisation layer

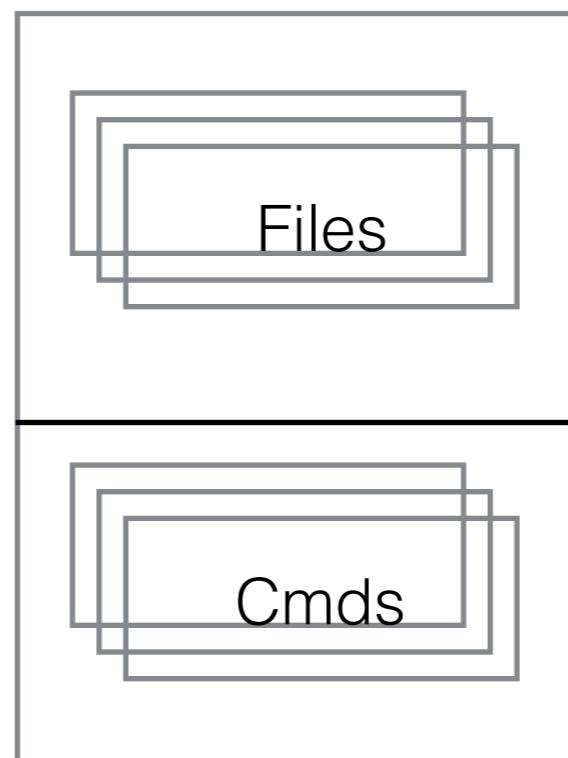
Host Os (azure)

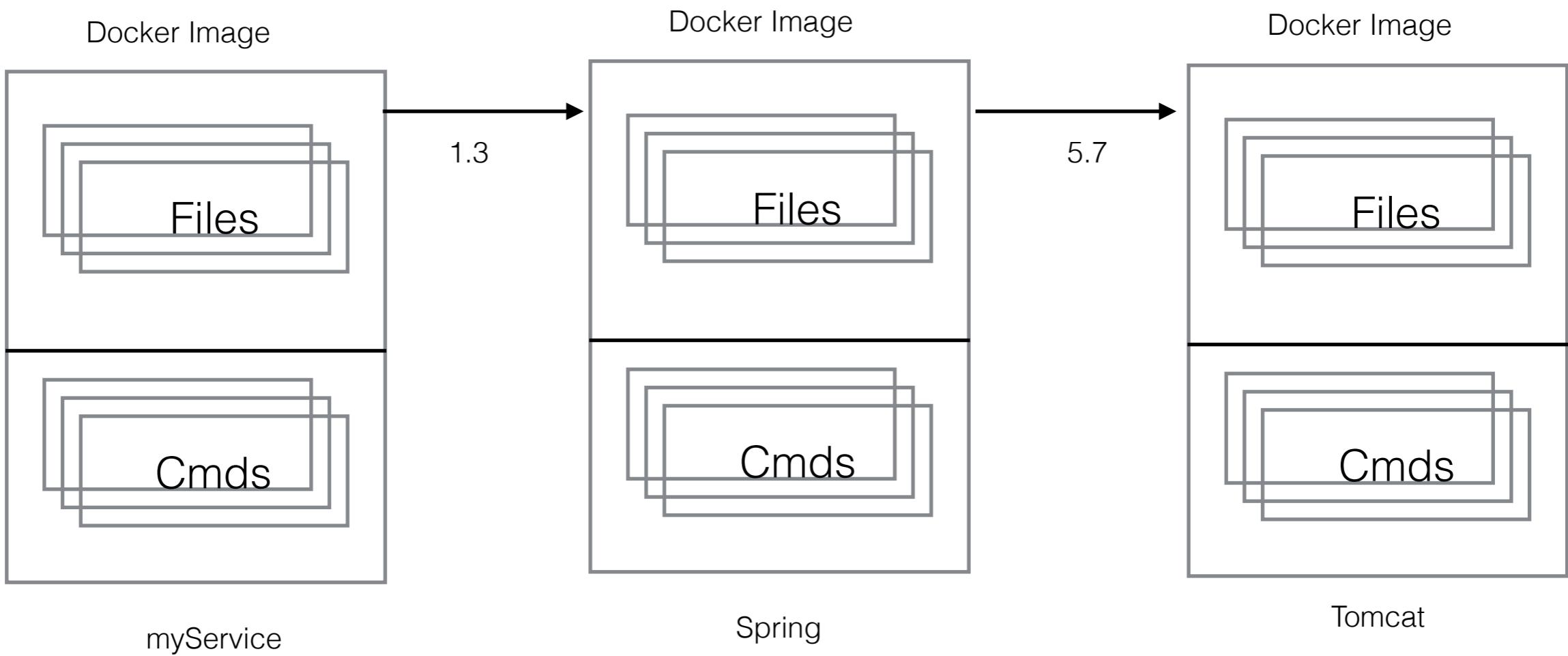
Cpu
RAM
Disk
I/o

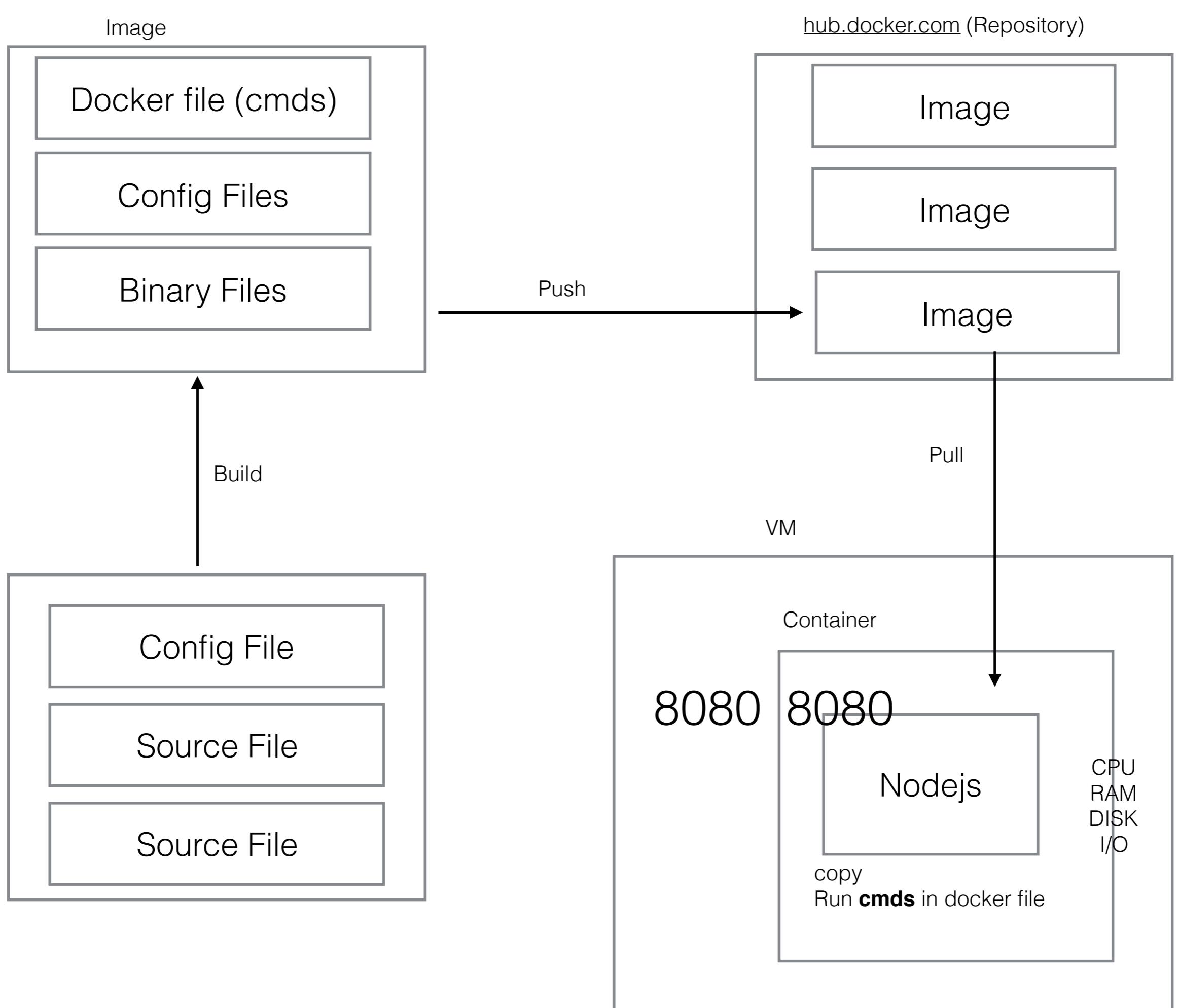
Machine



Reproducible Env.

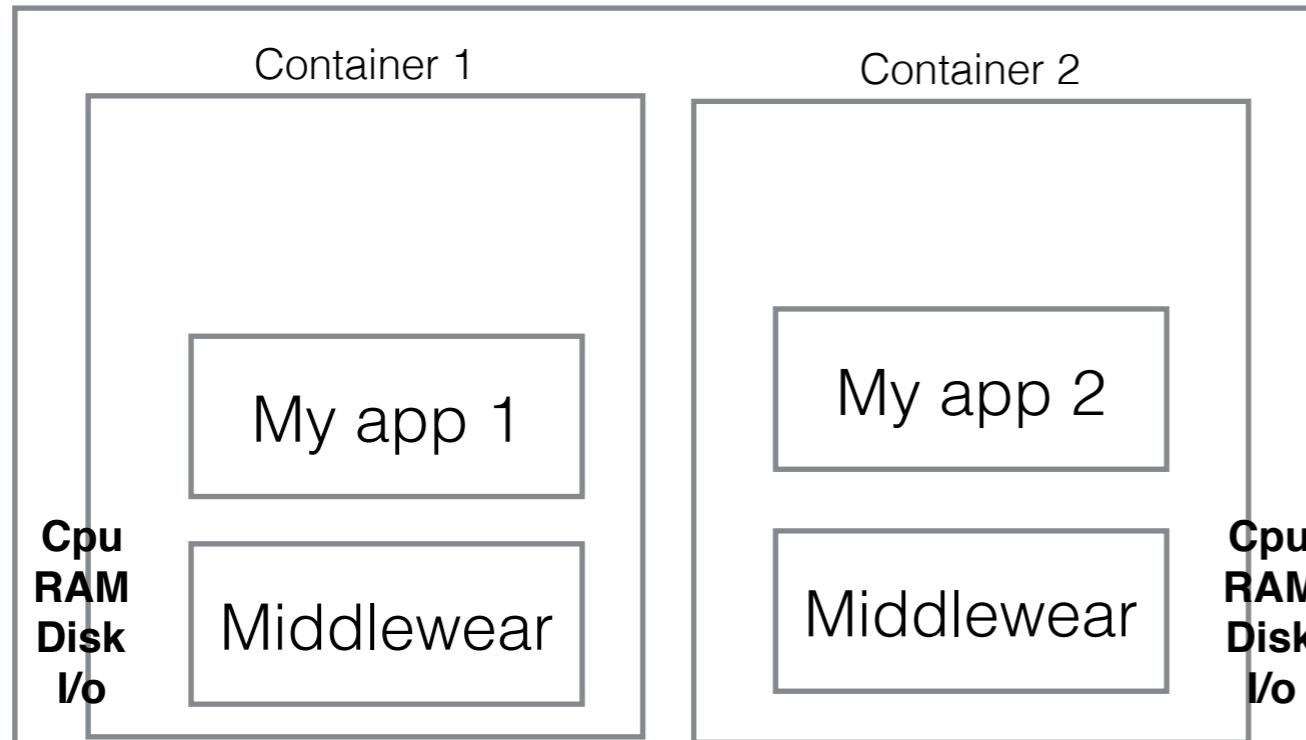




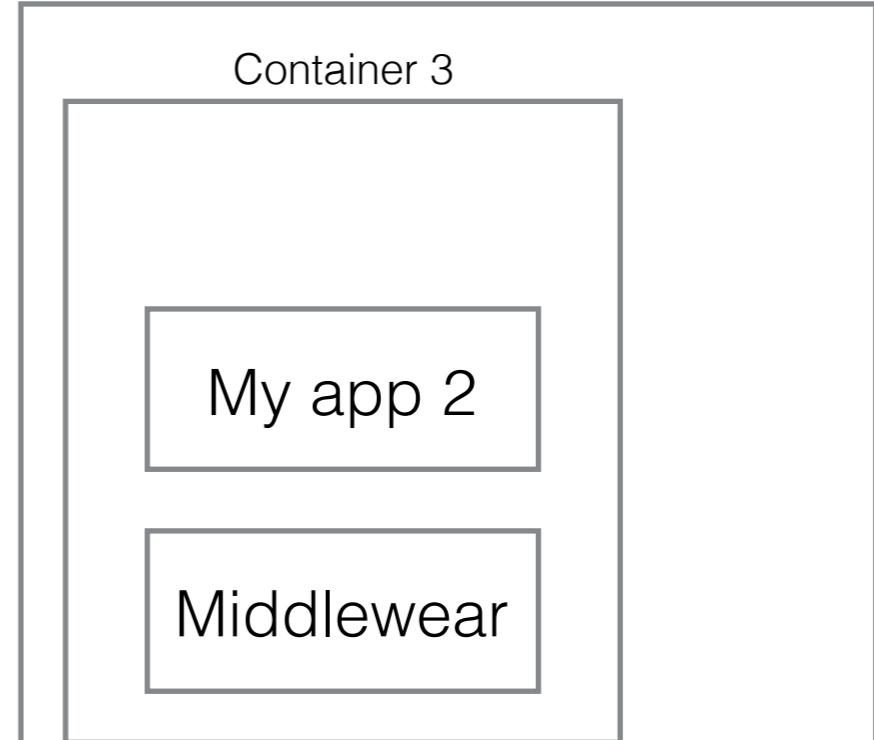


Managing Container Life cycle

VM1



VM2



Containerization layer (Docker)

Unix Guest Os

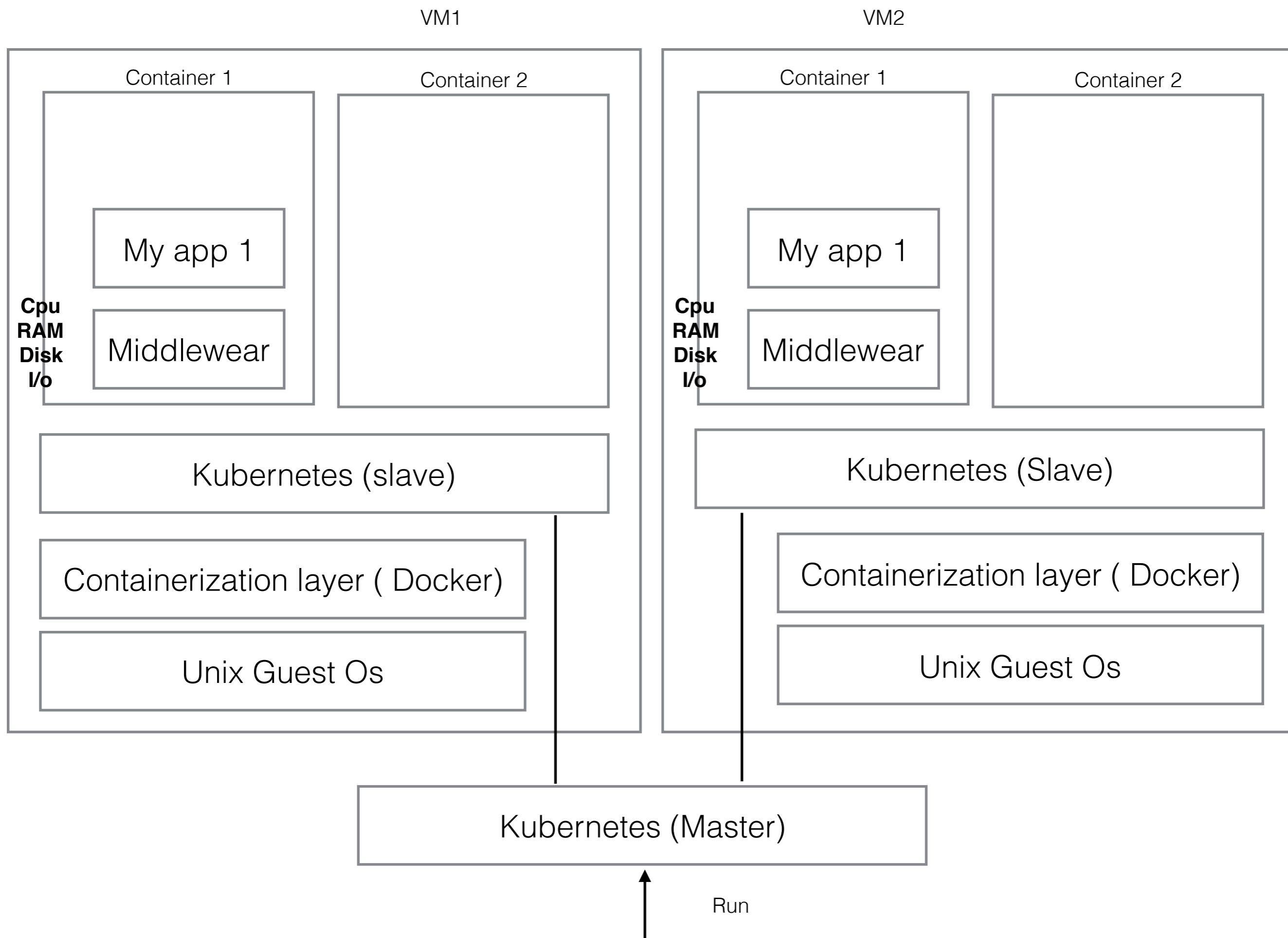
Cpu
RAM
Disk
I/o

Cpu
RAM
Disk
I/o

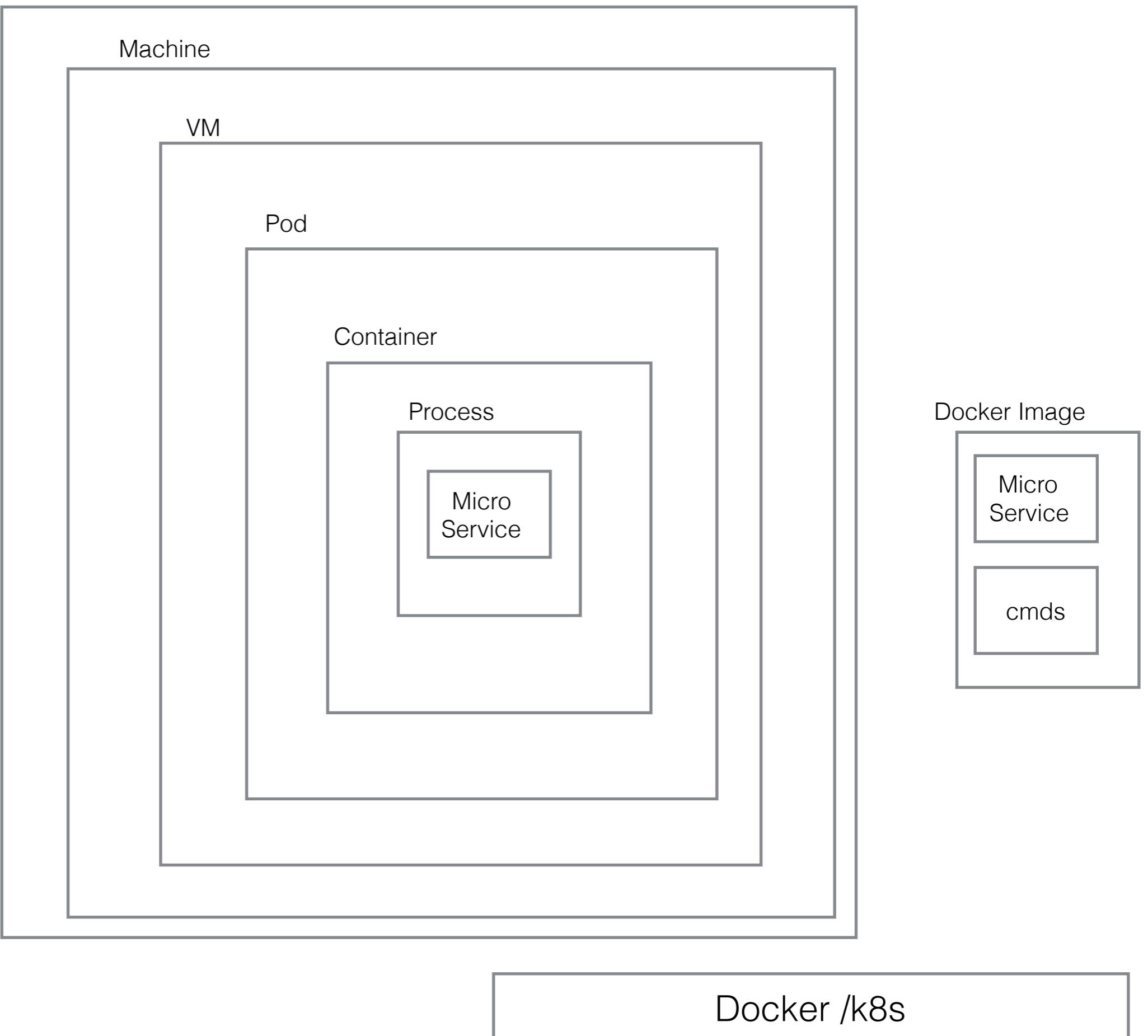
Virtualisation layer

Host Os (azure)

Machine



Cluster



Cluster

Node

Pod

Container

Process

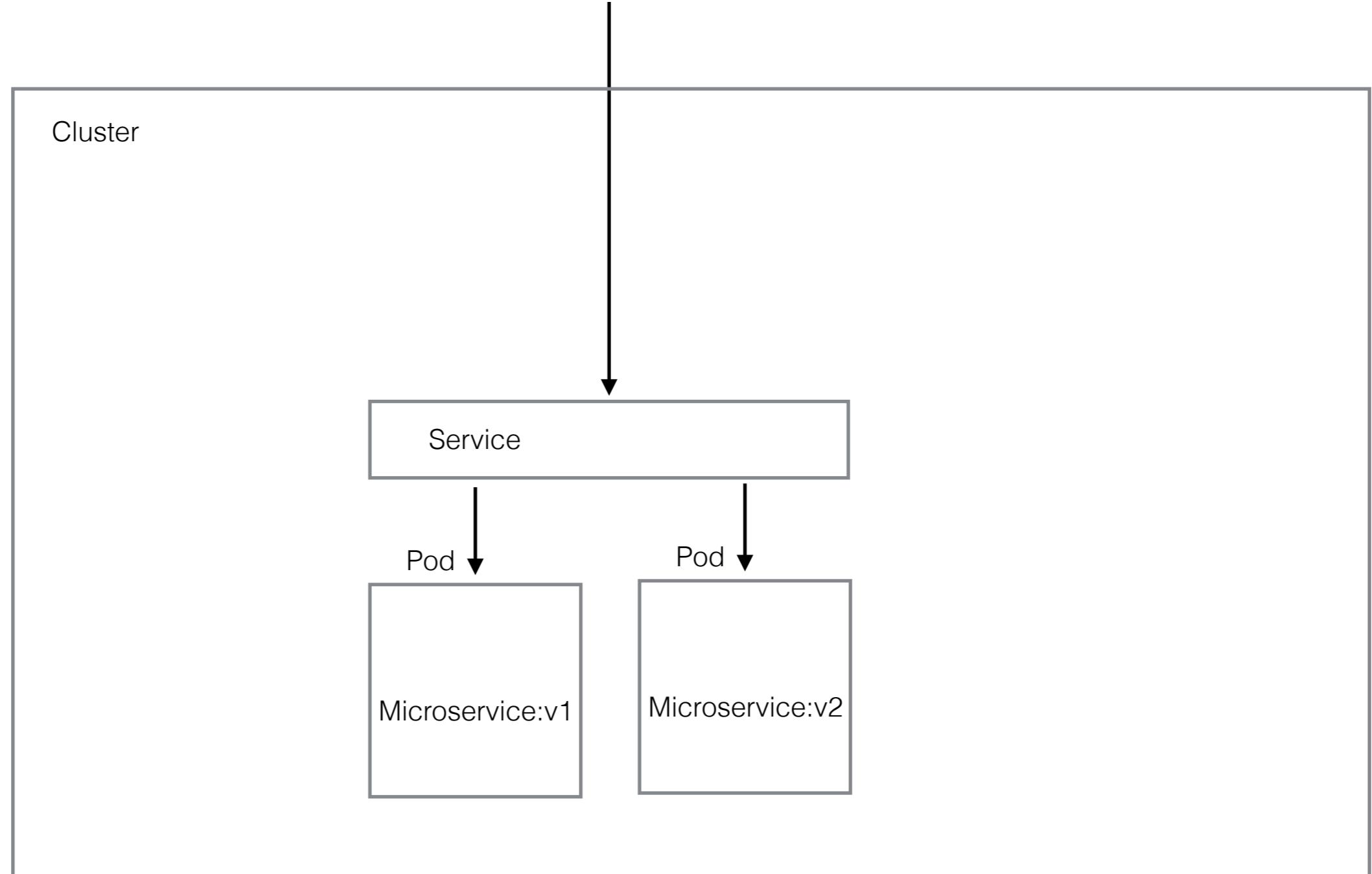
Micro
Service : v1

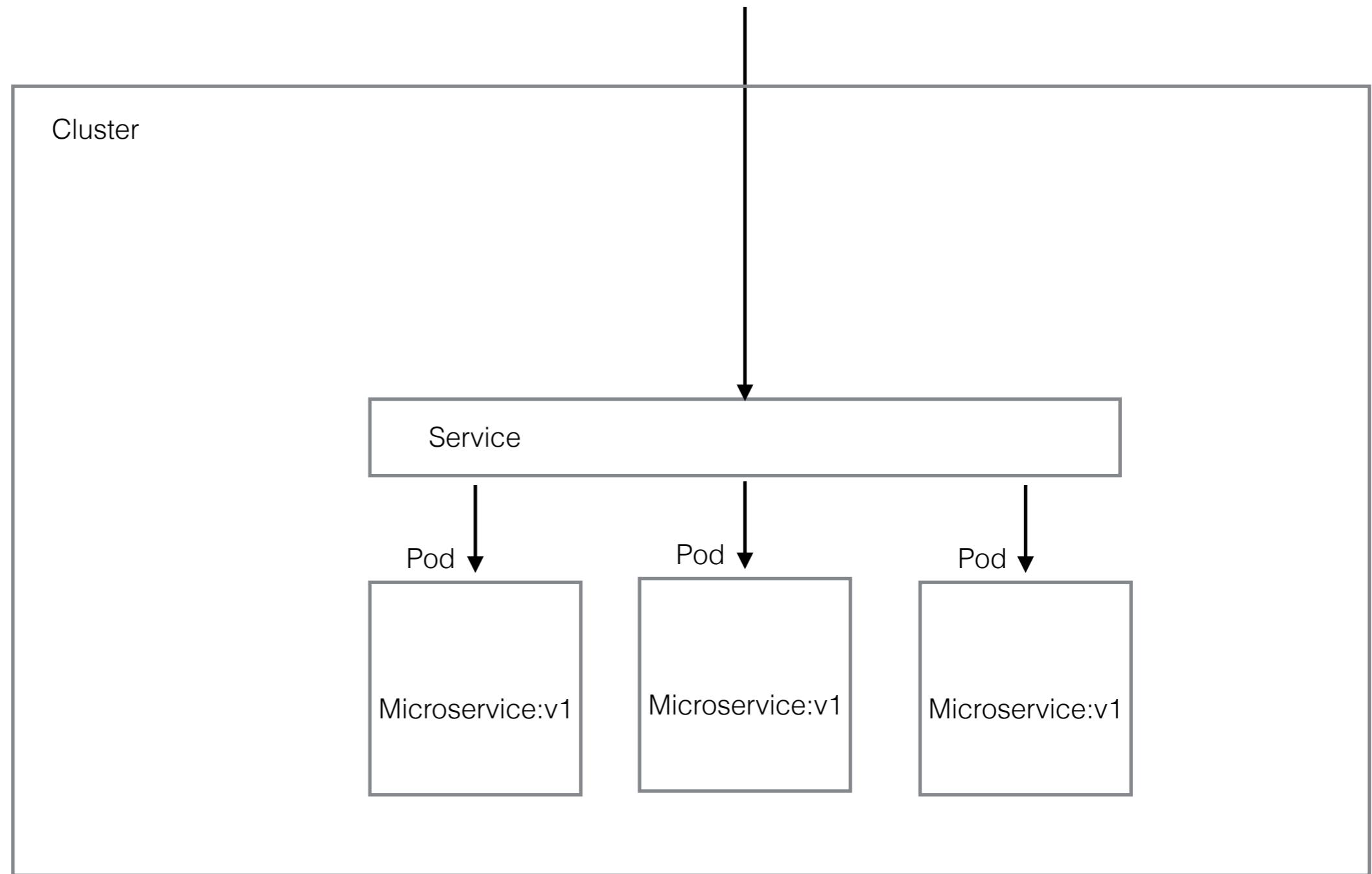
Pod

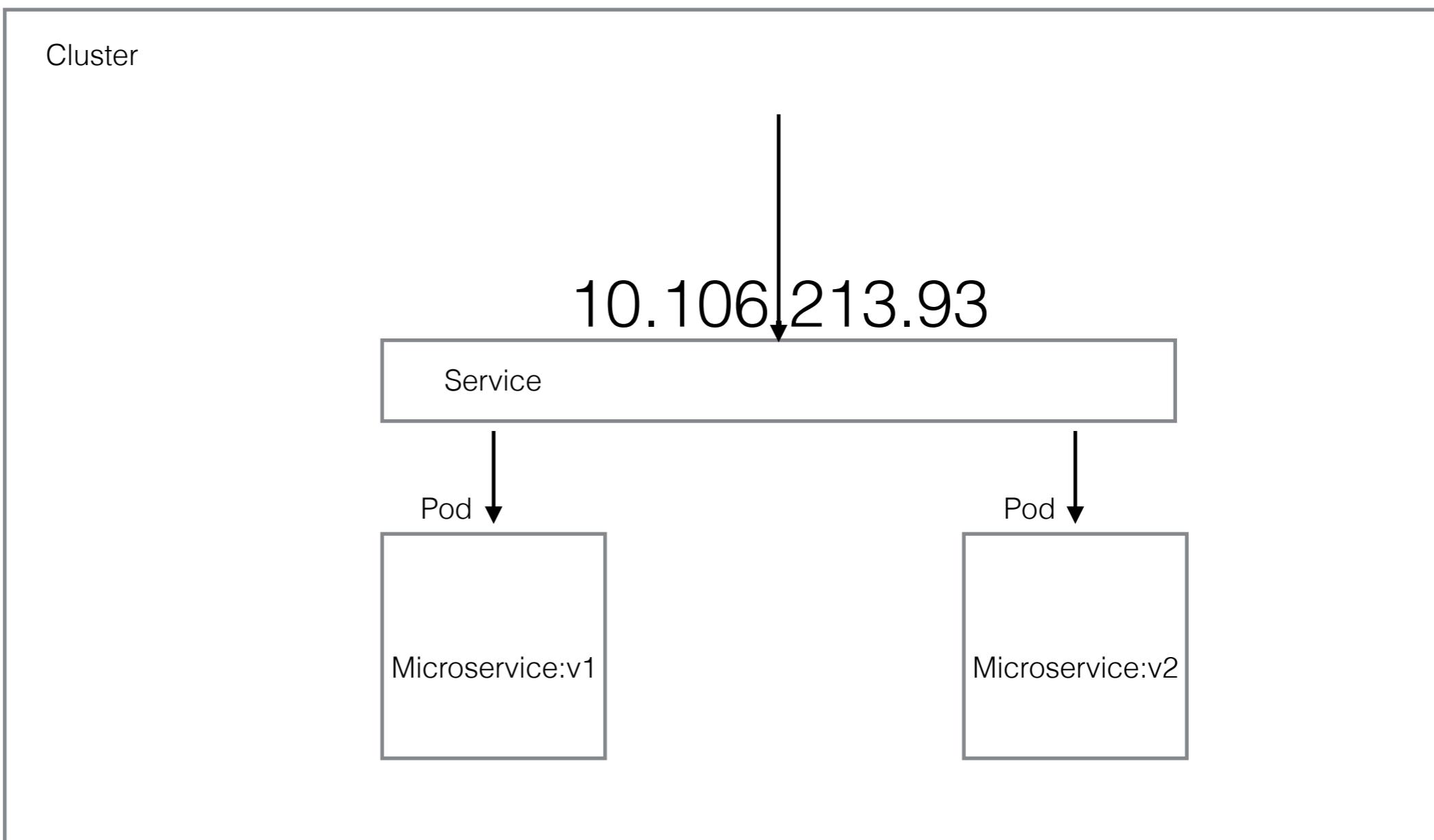
Container

Process

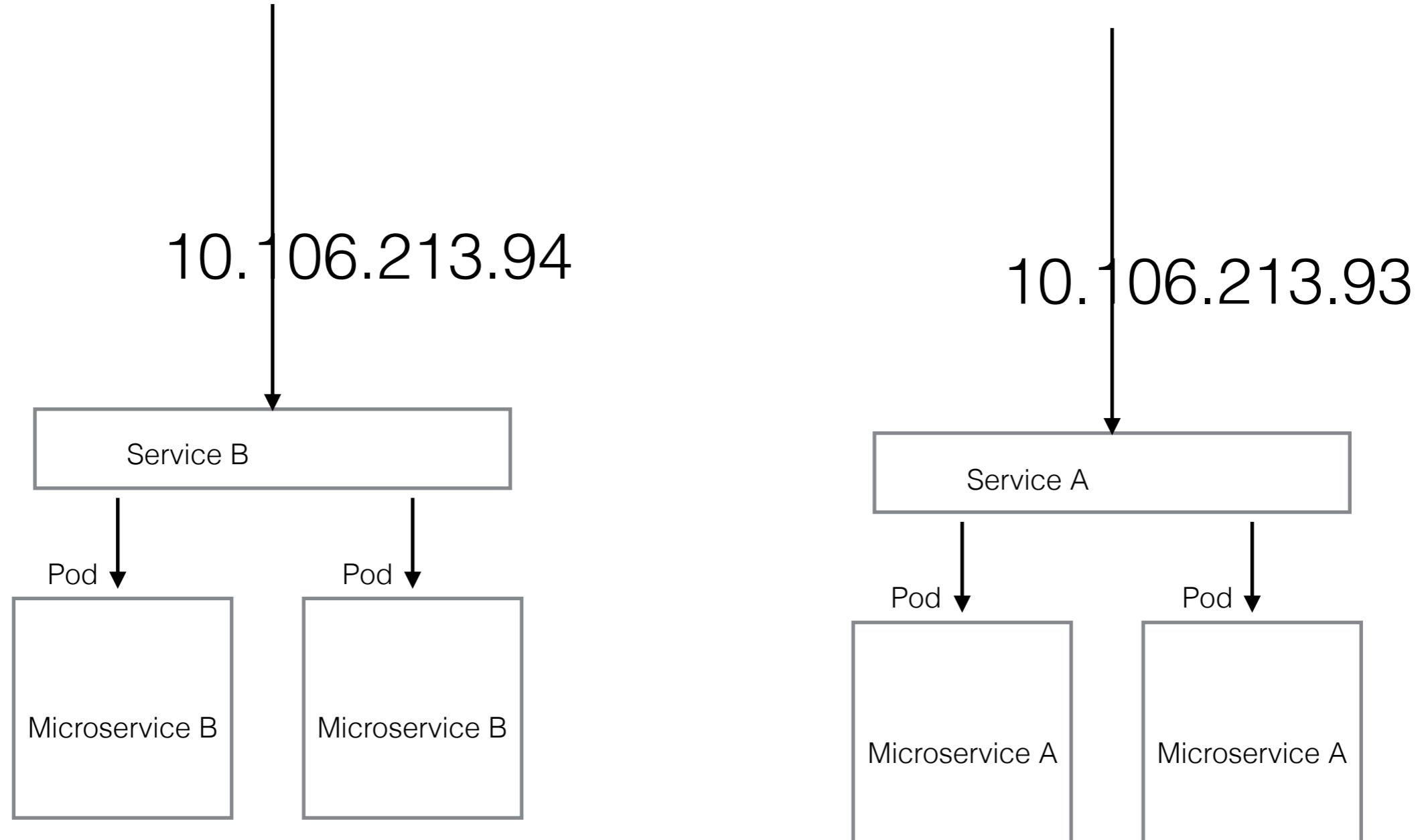
Micro
Service : v2

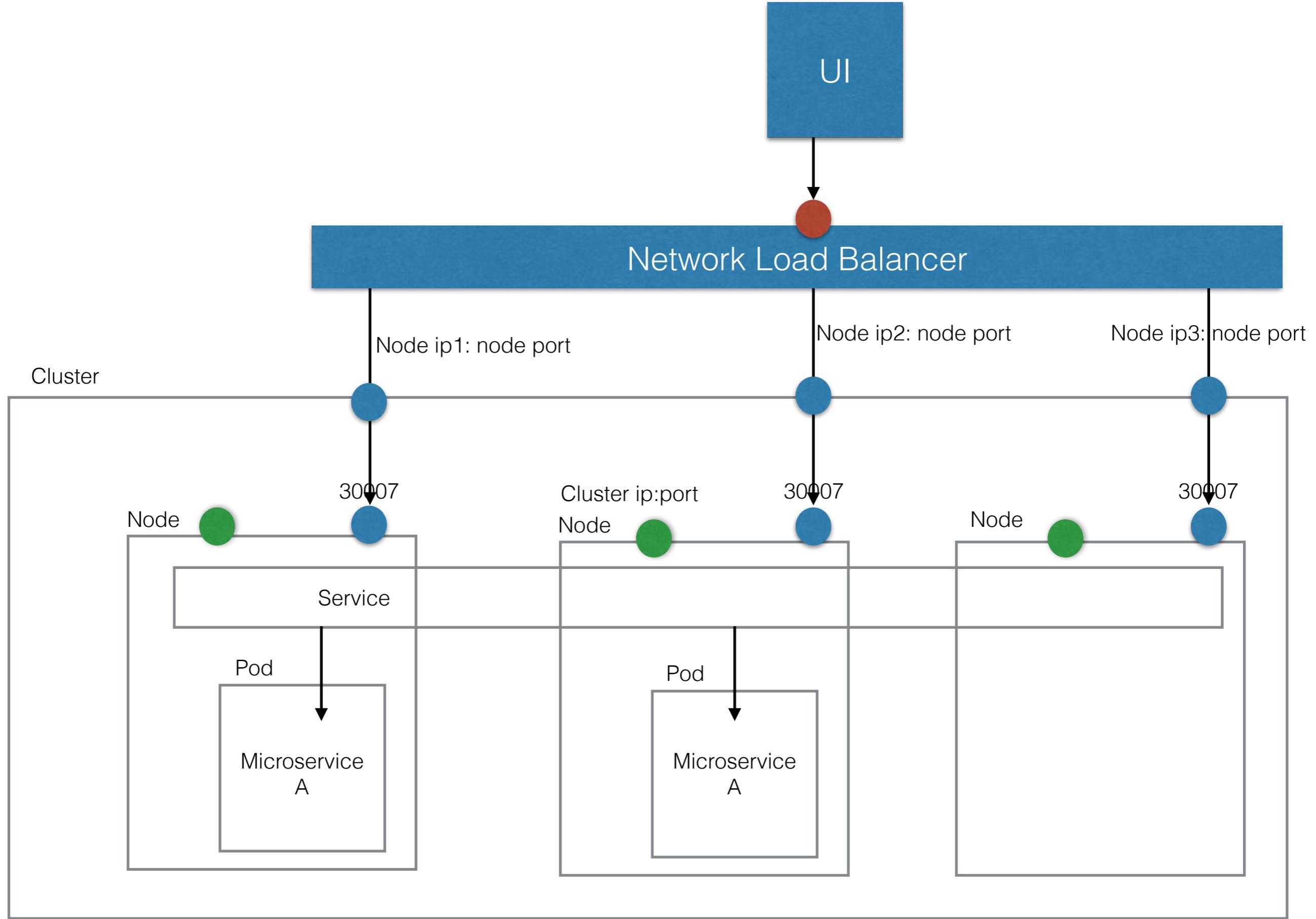


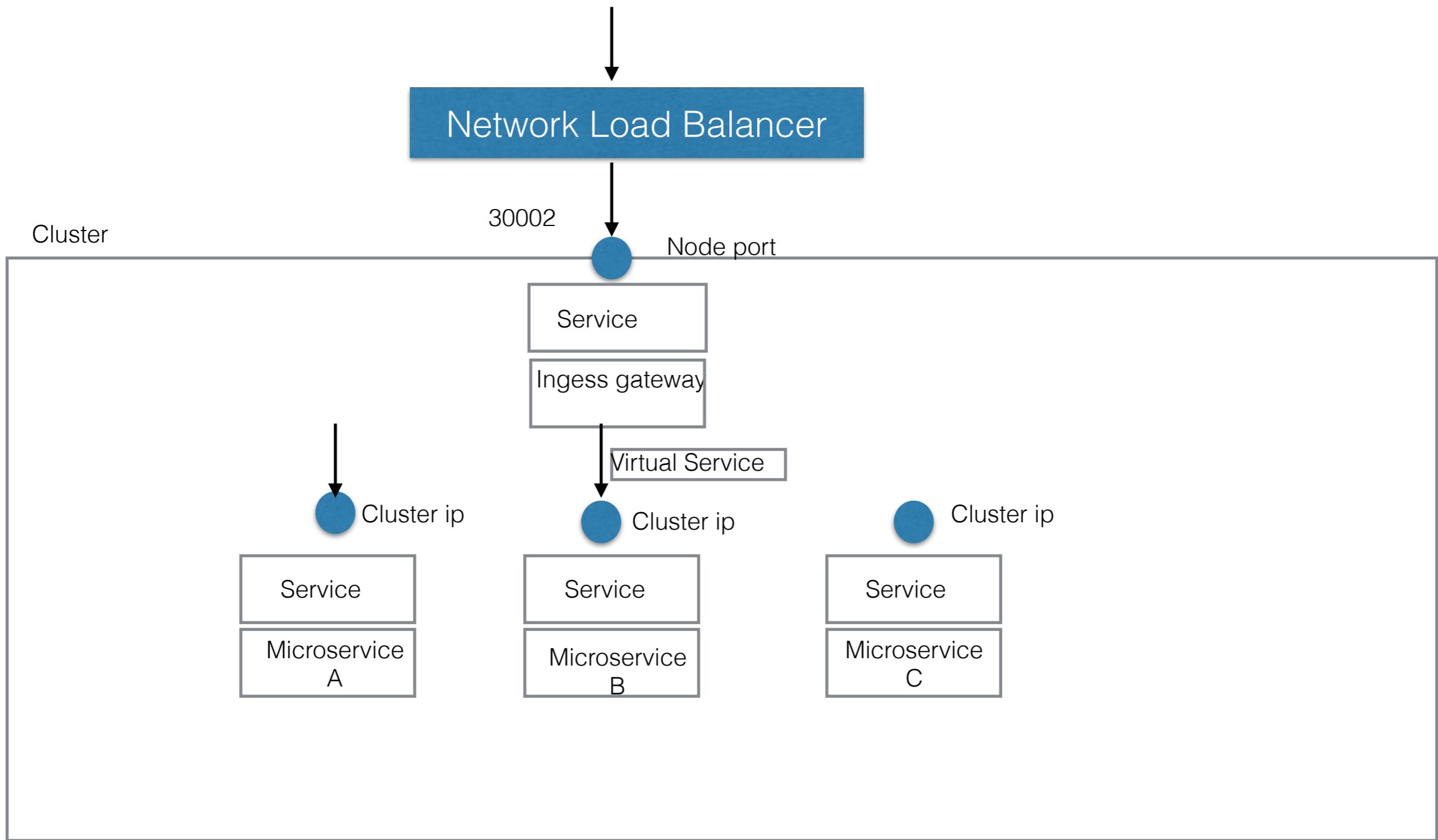


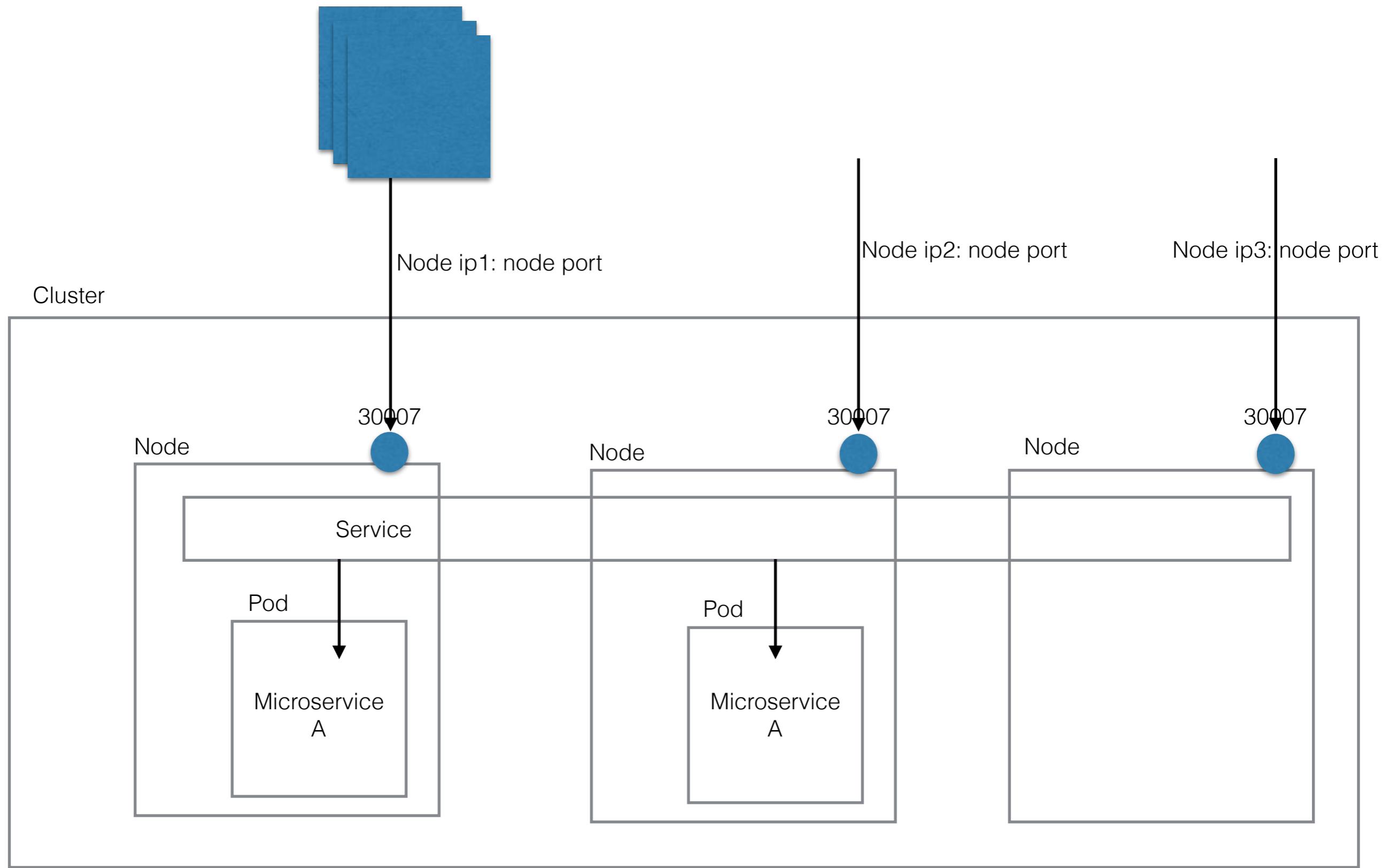


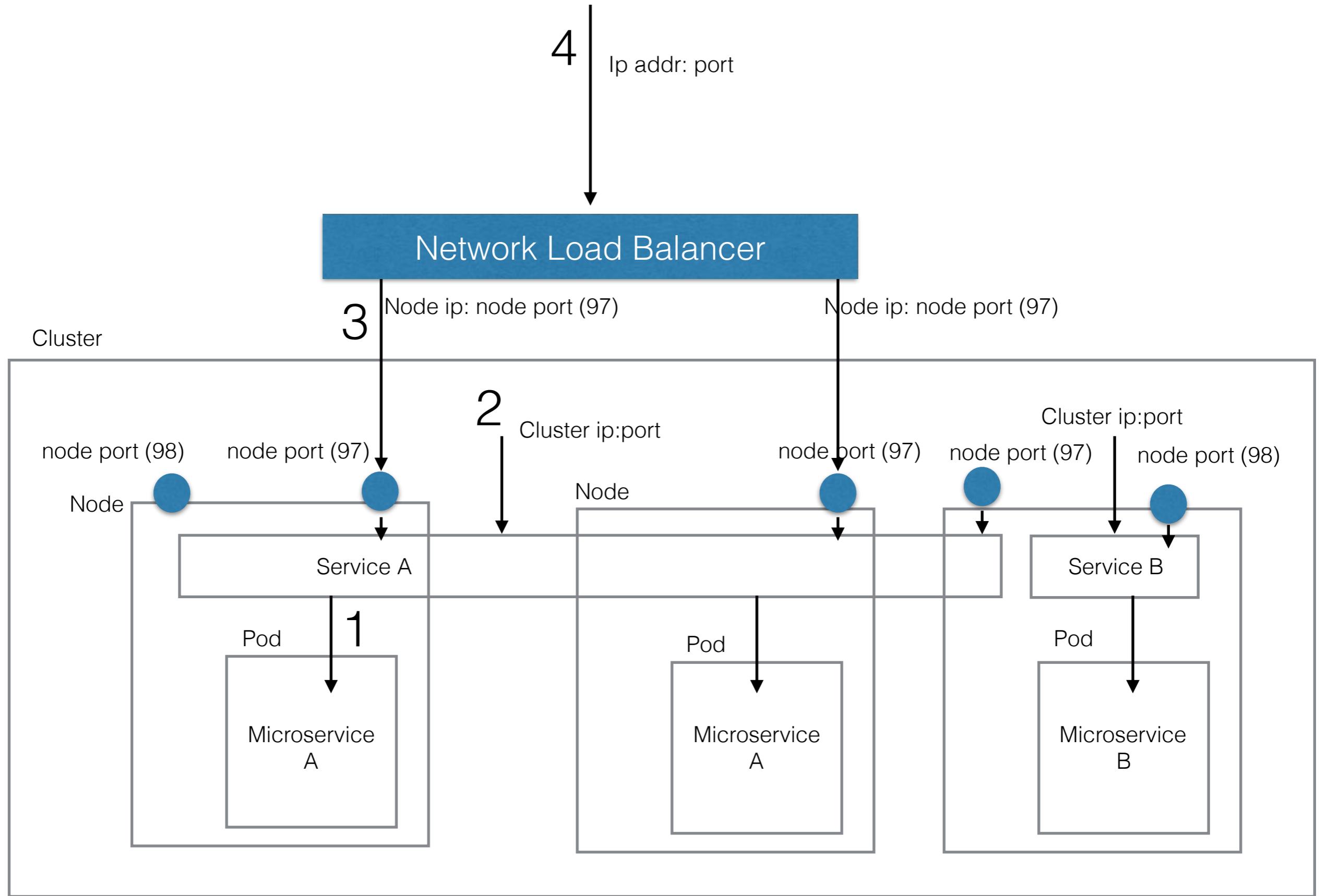
Cluster

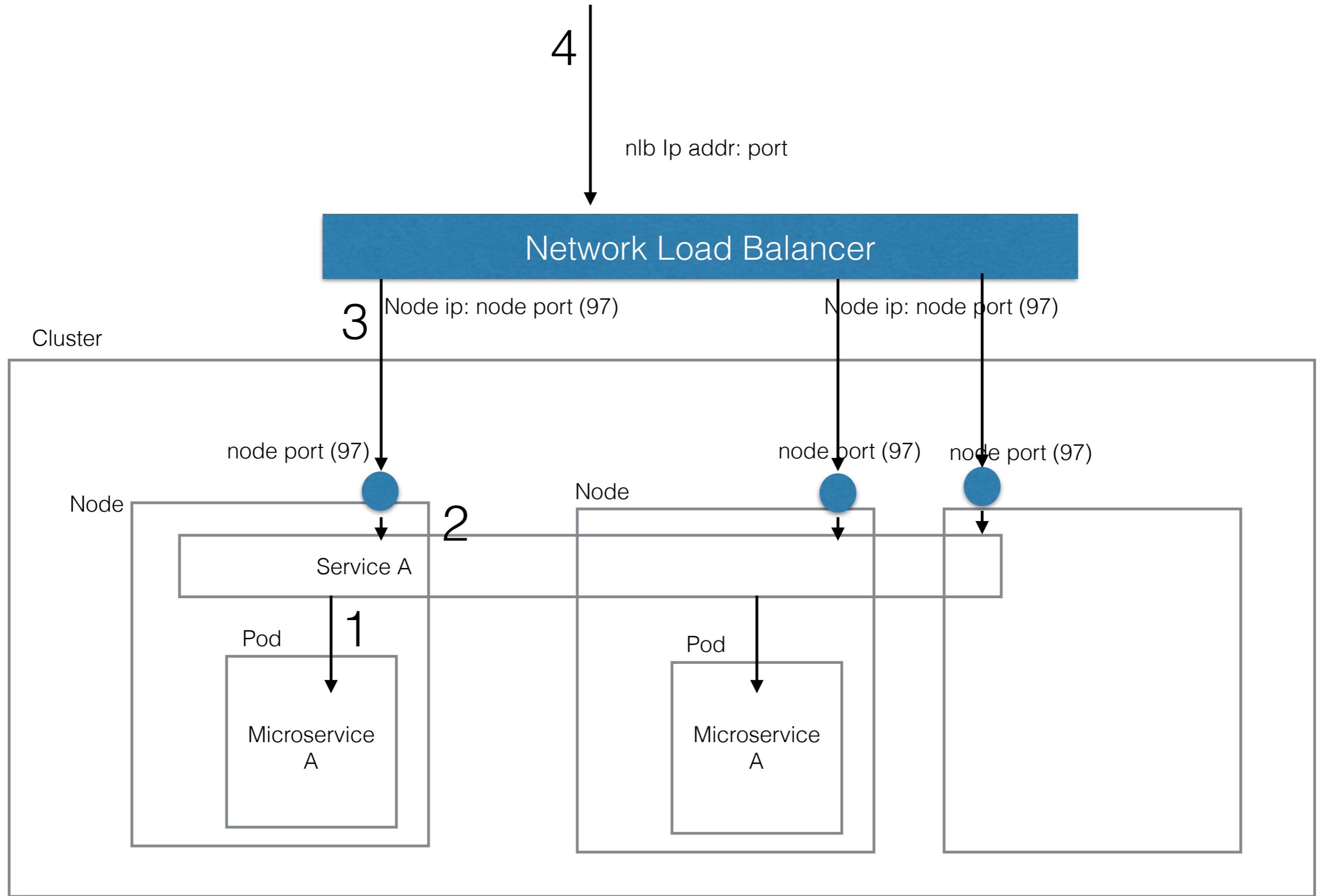


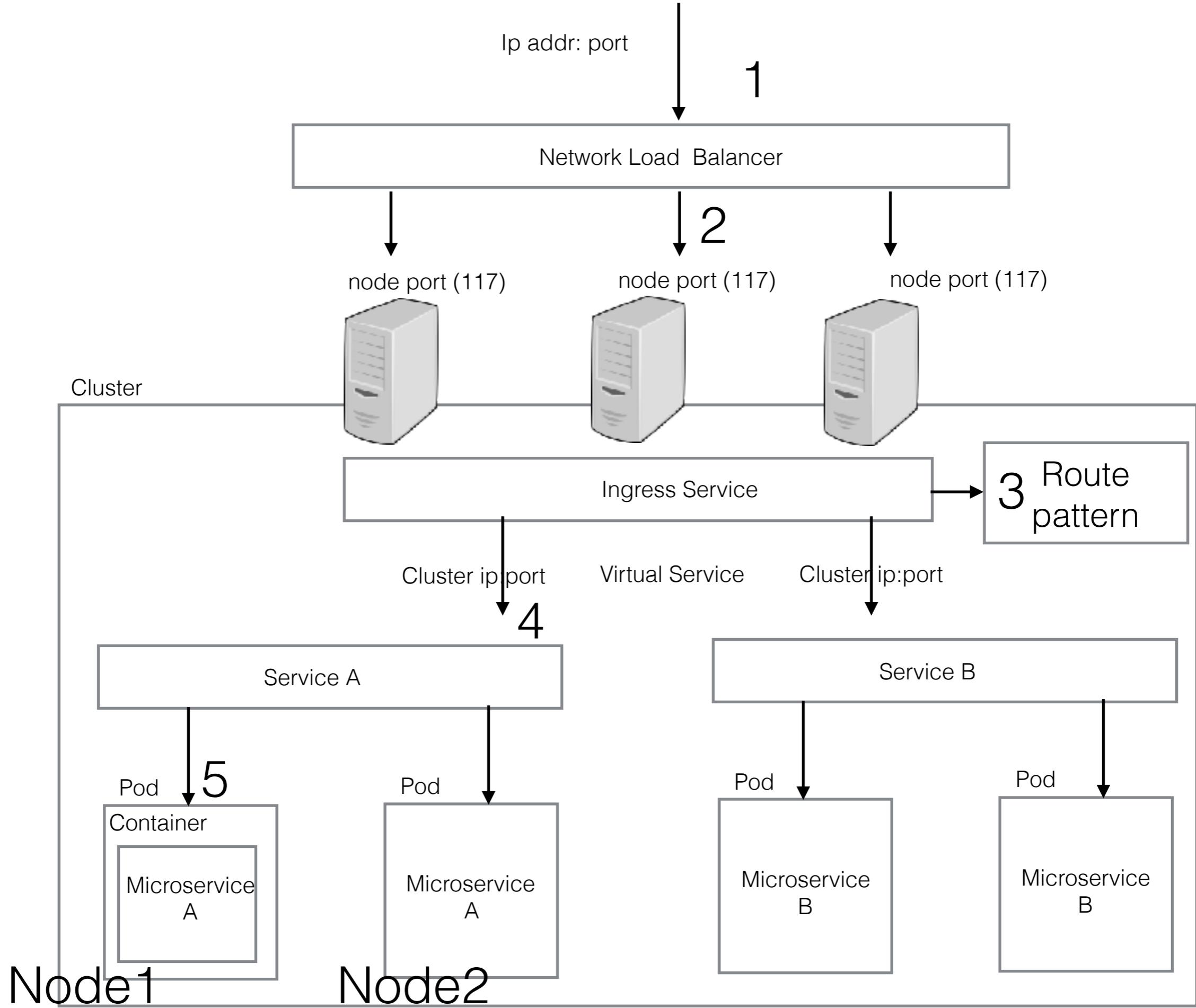




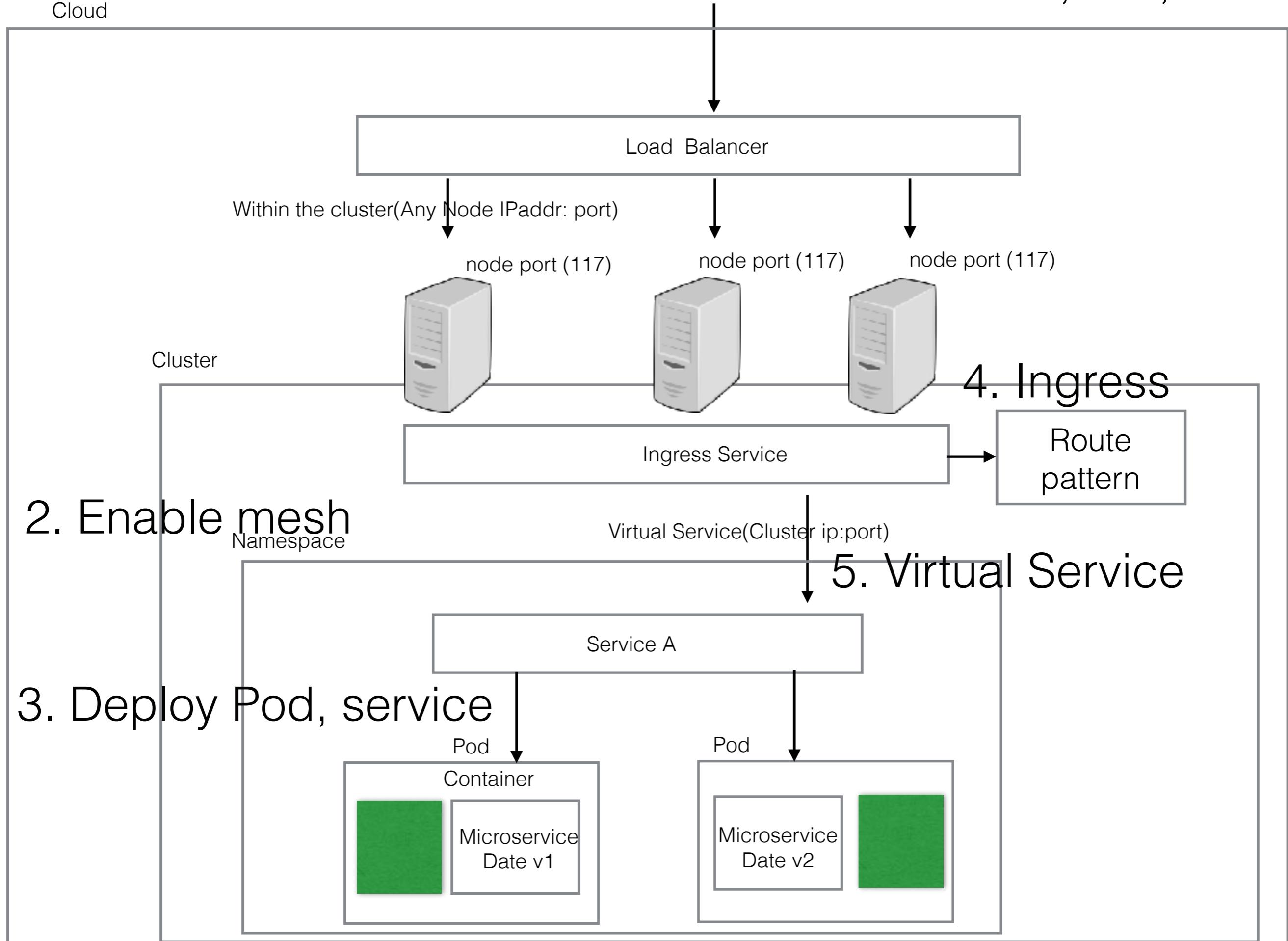






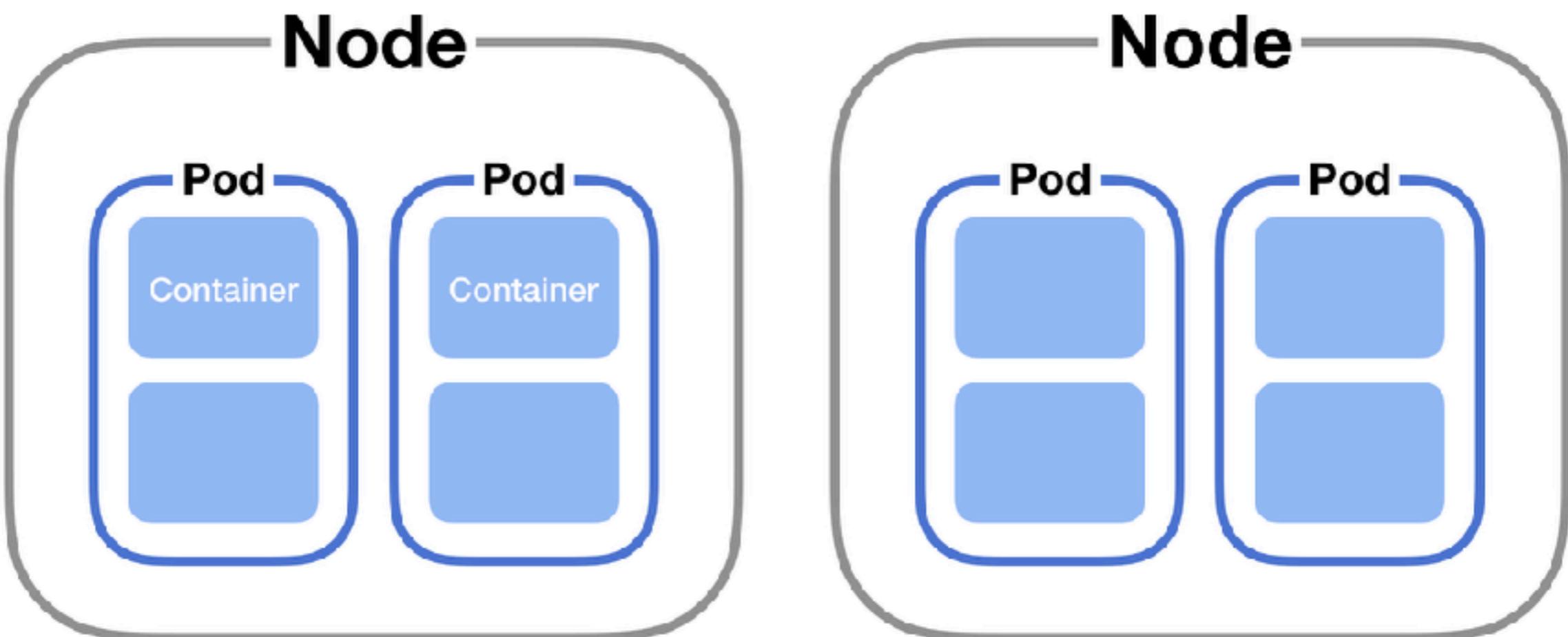


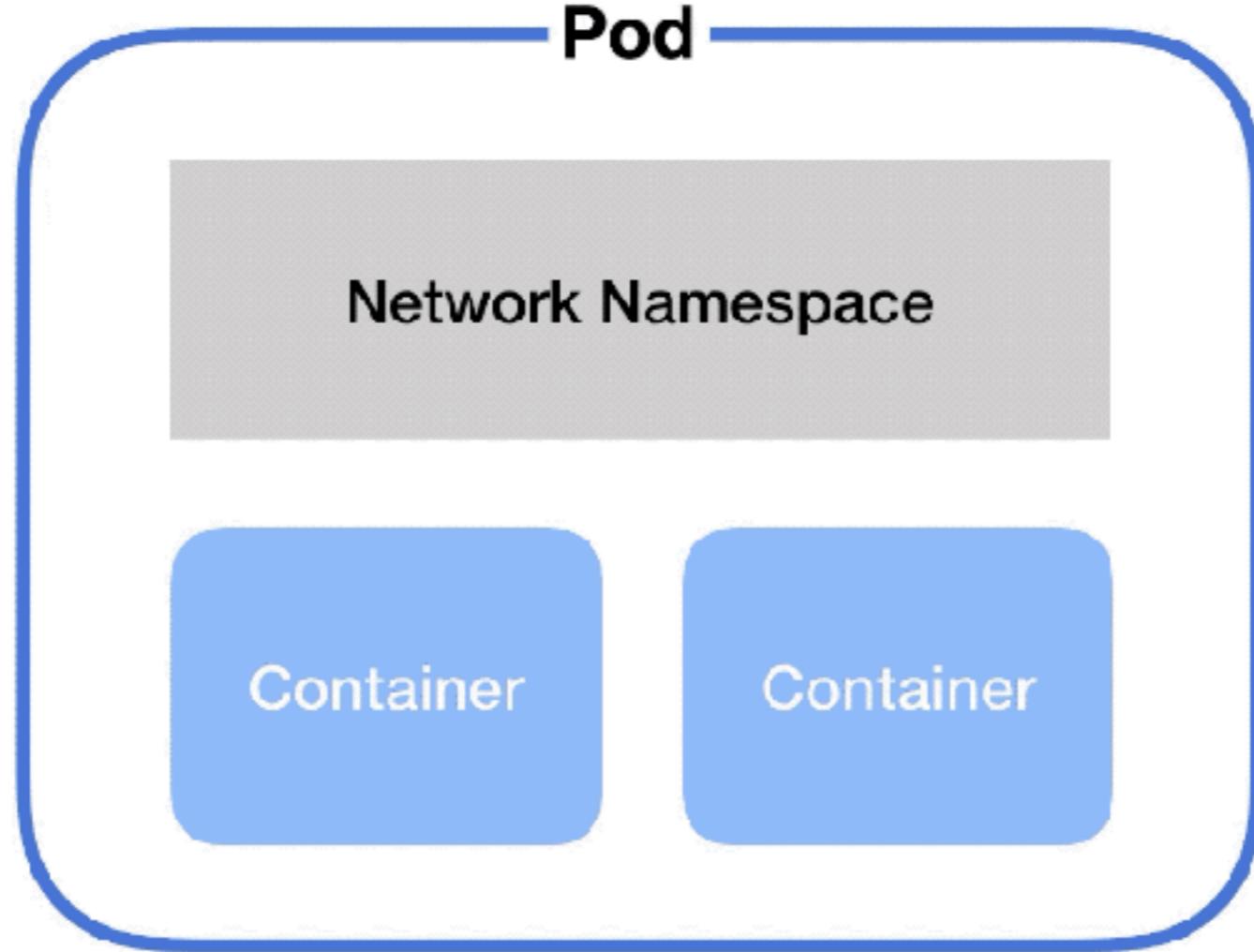
1. Docker, k8s, Istio



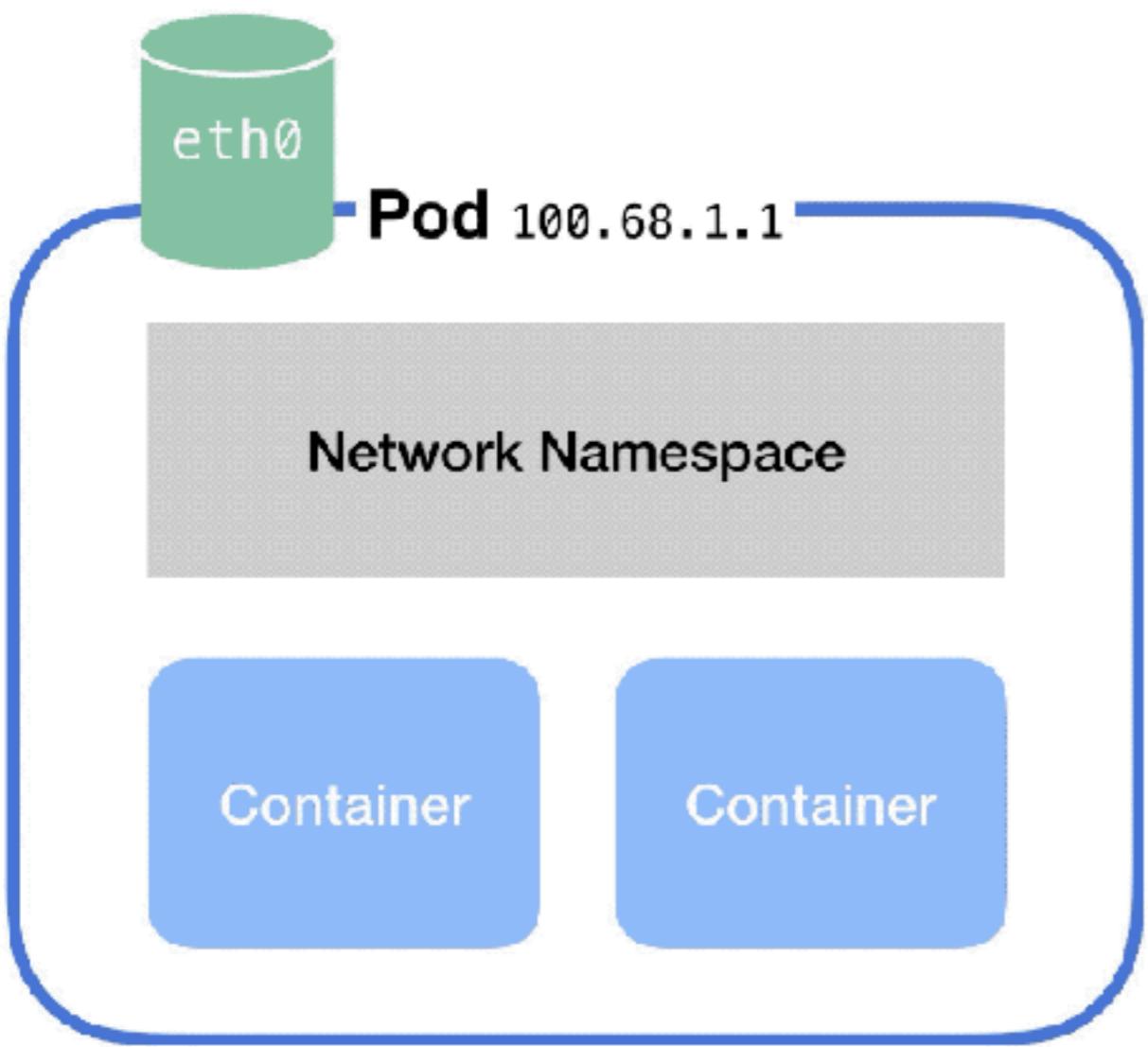
Service Discovery

Cluster

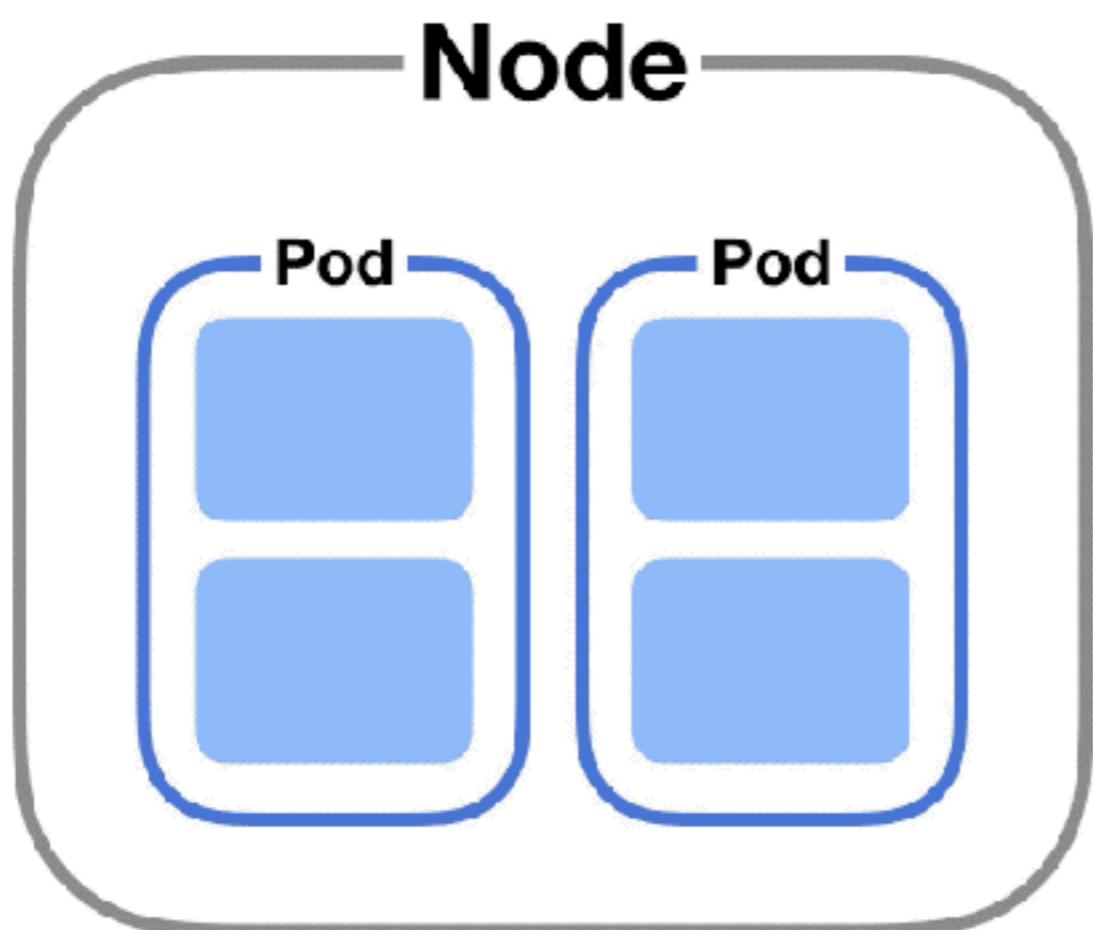


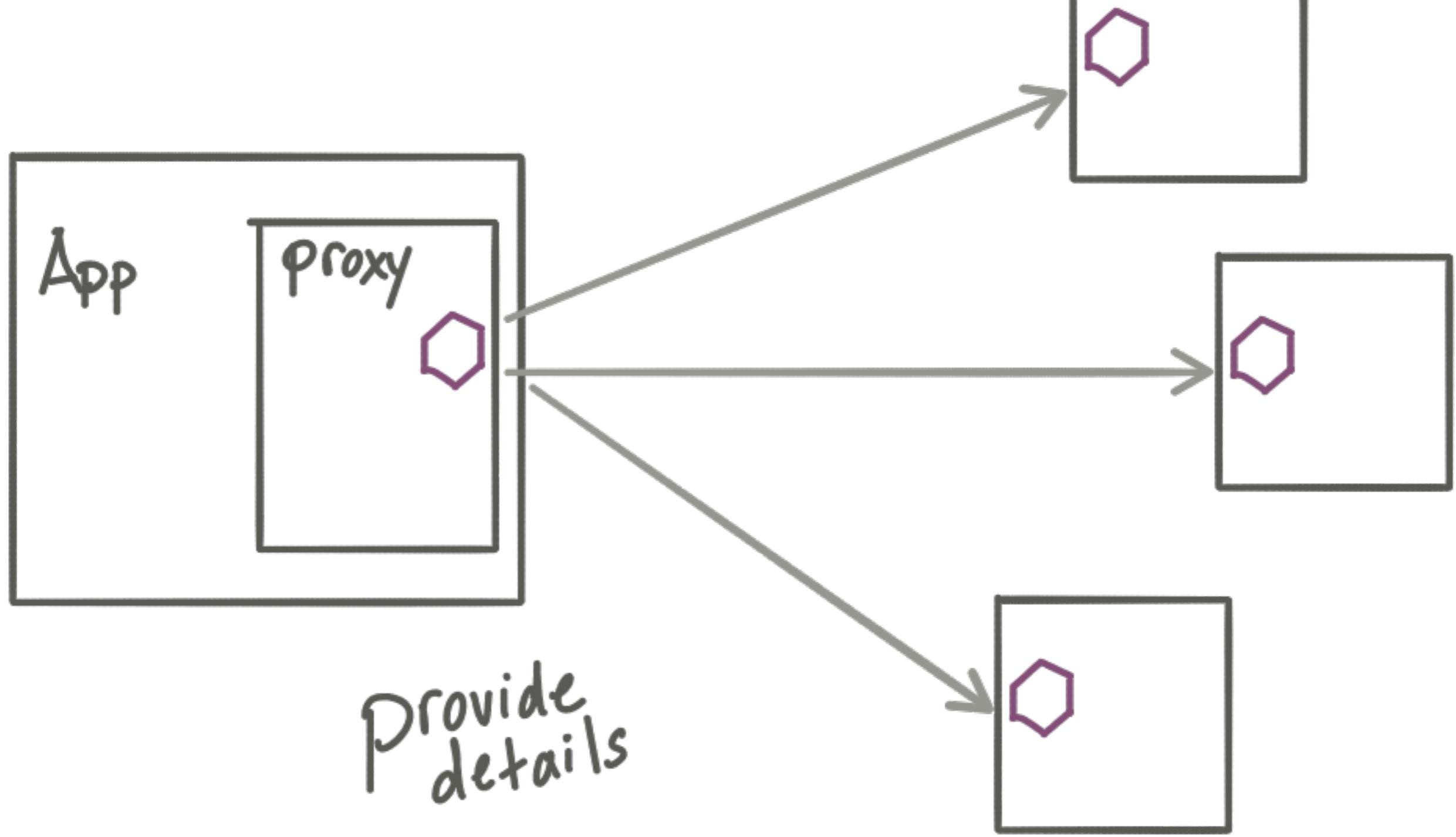


Node

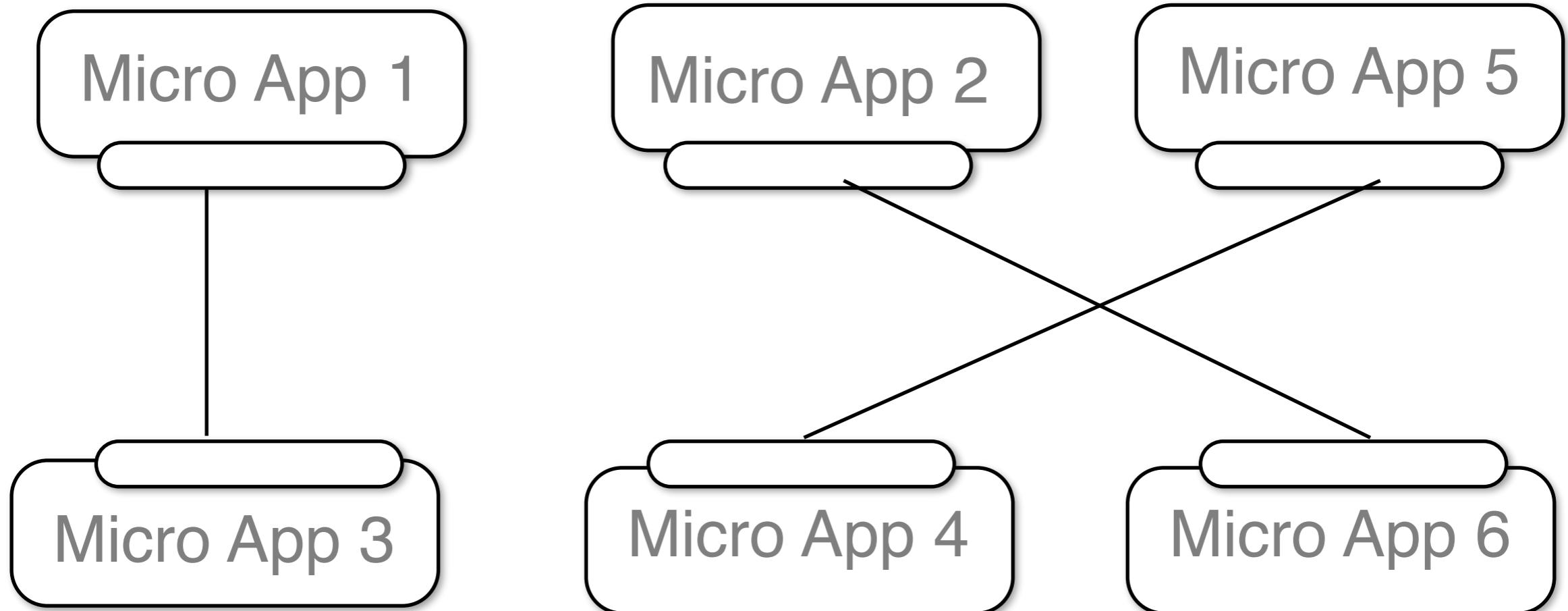


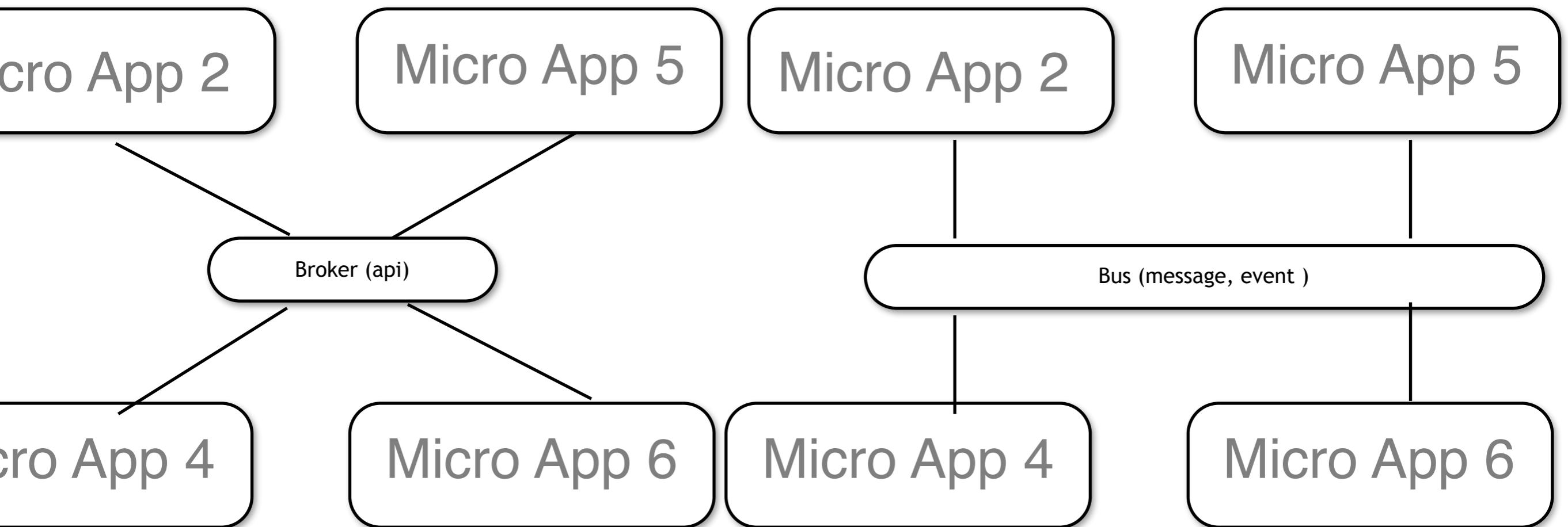
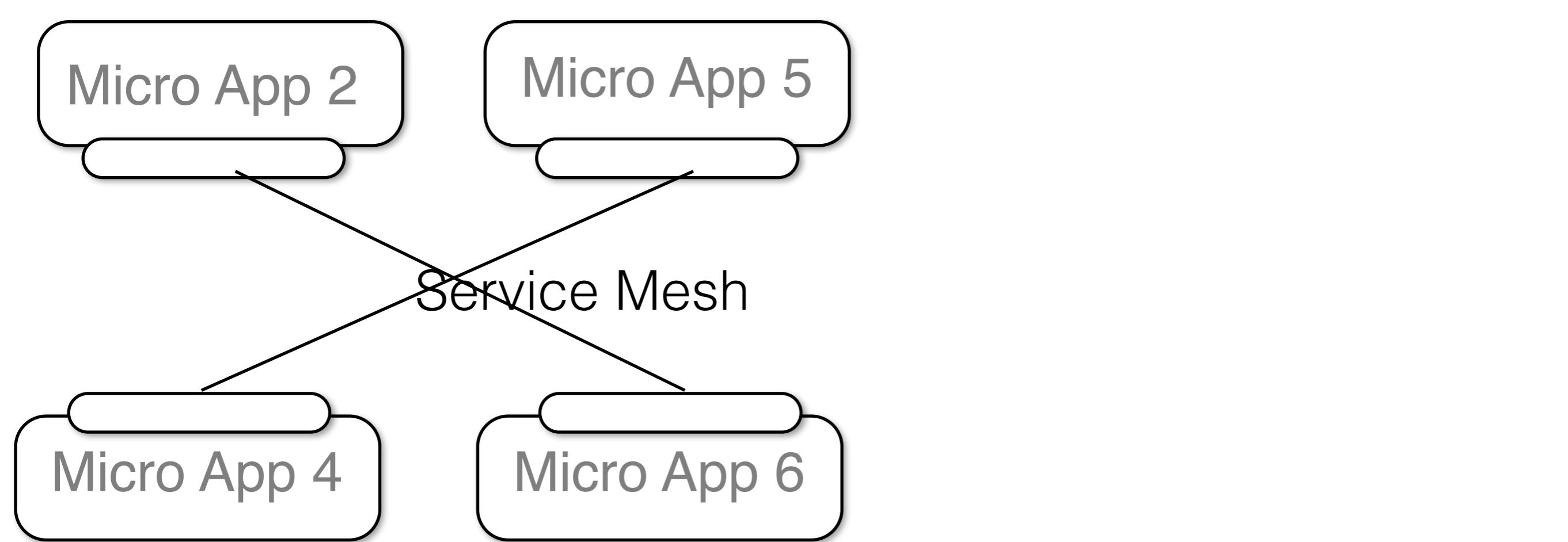
100.68.1.xxx

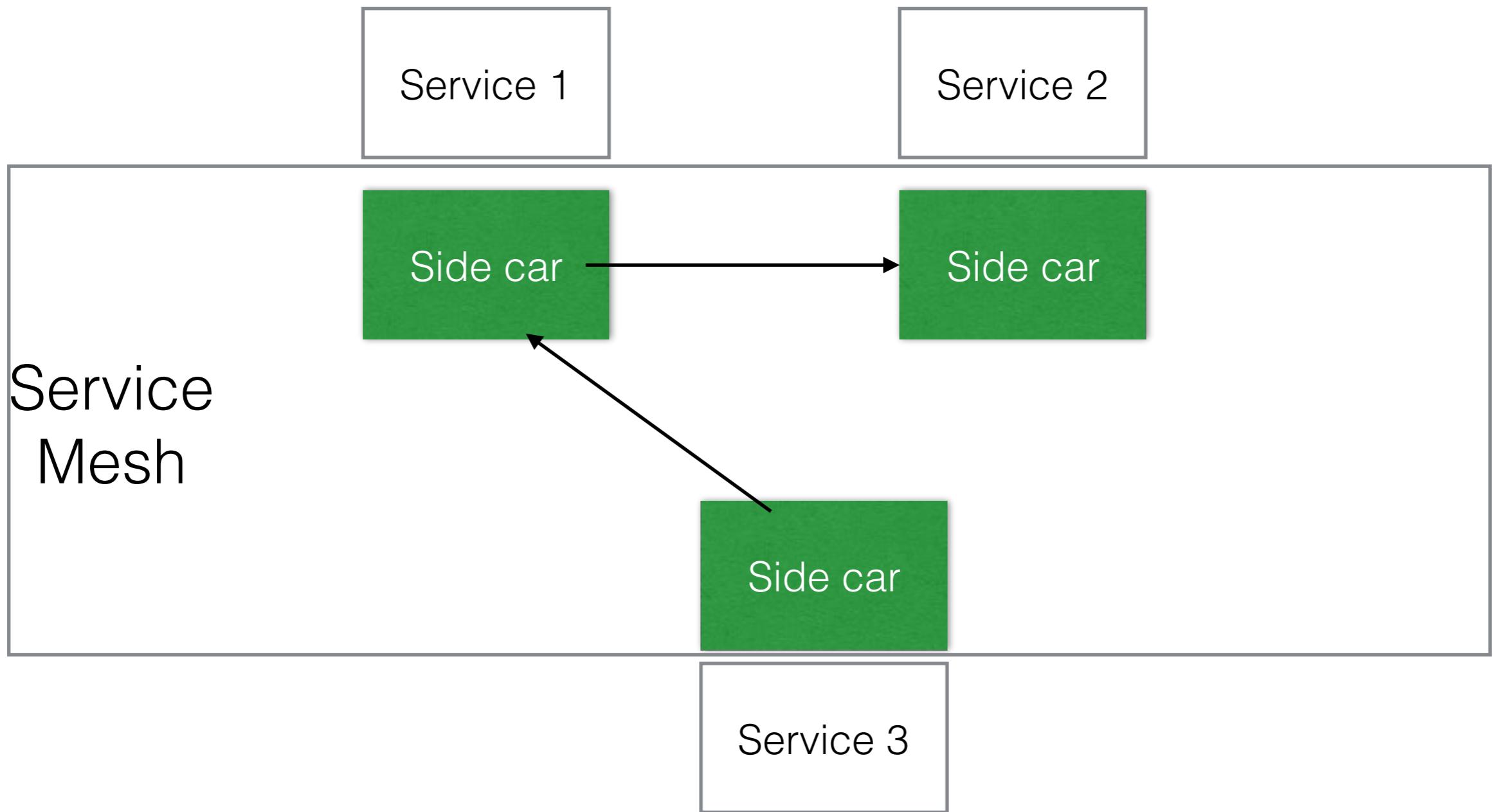


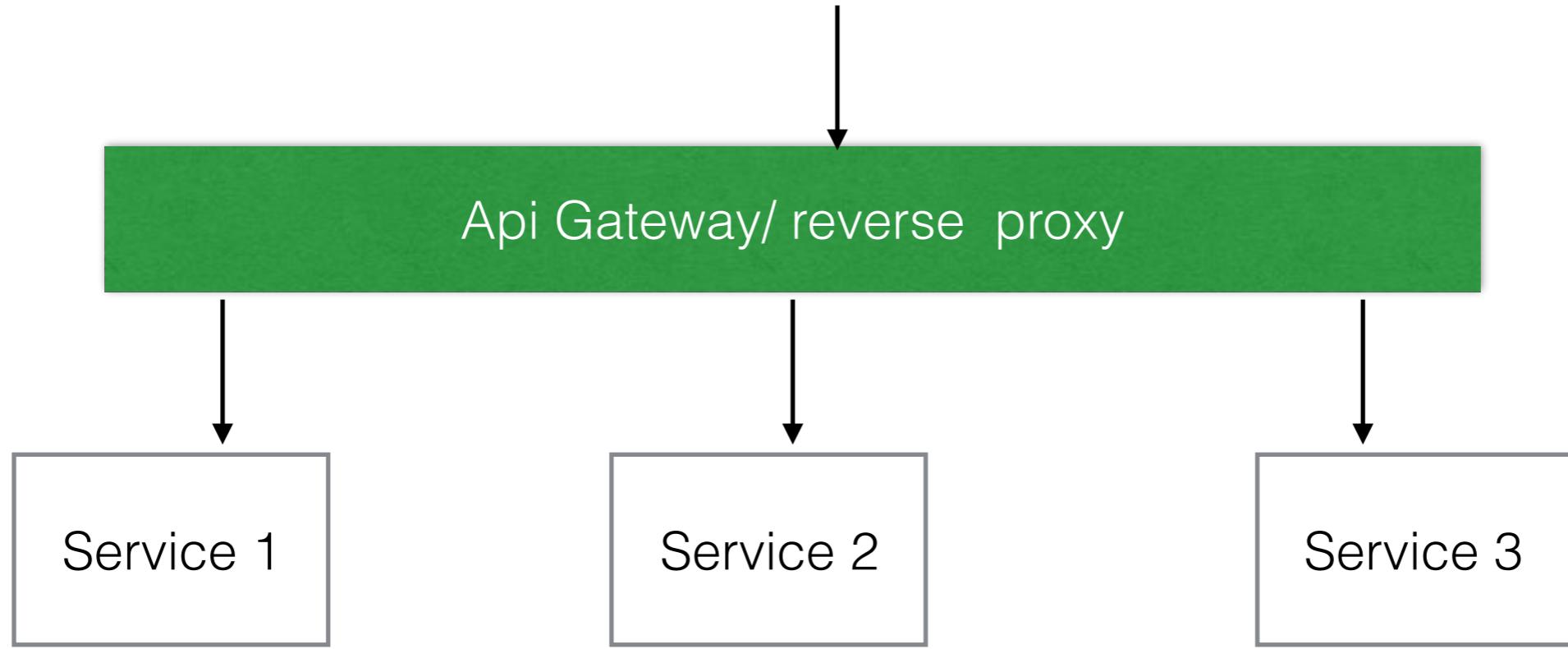


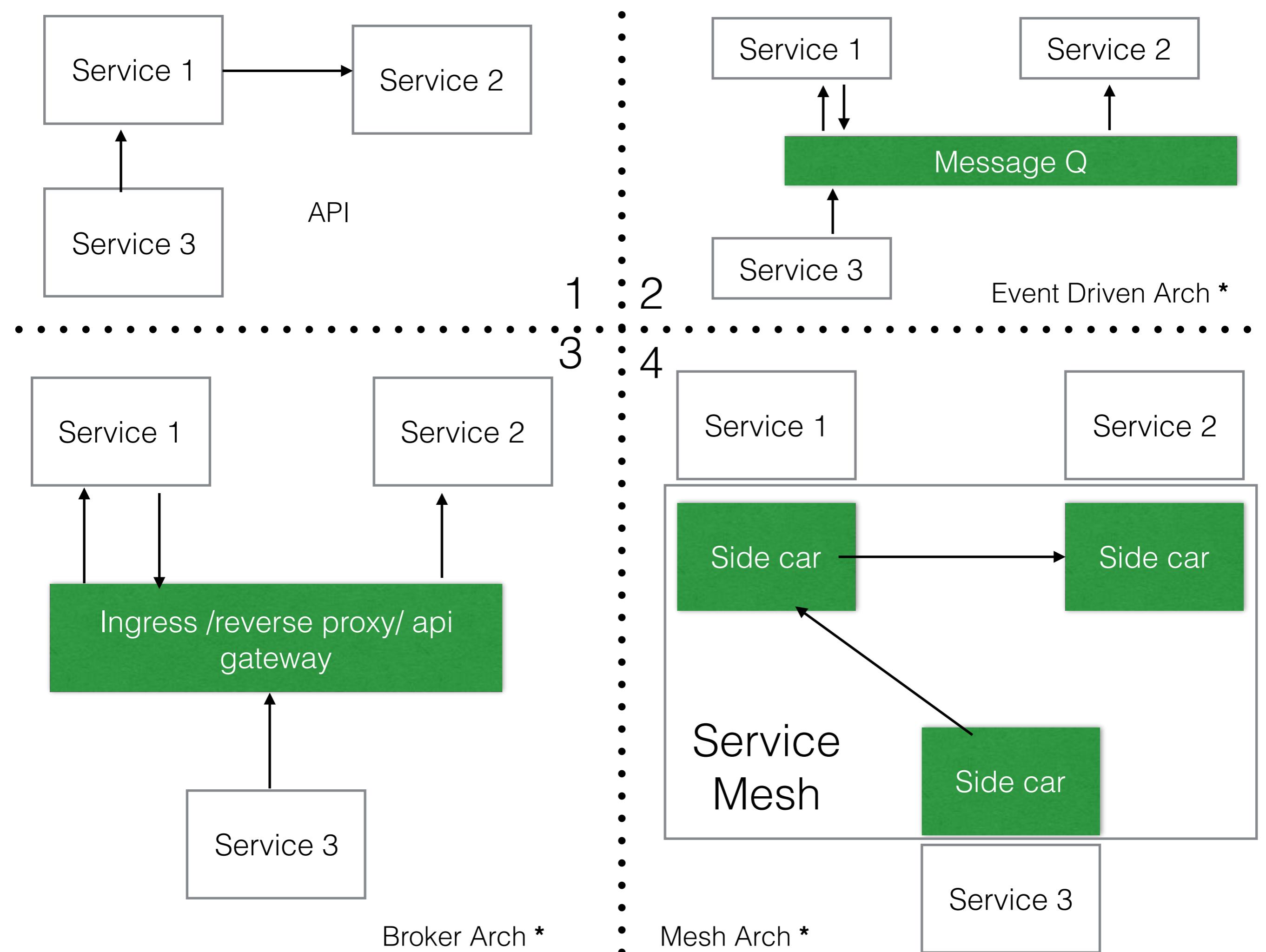
Service Mesh

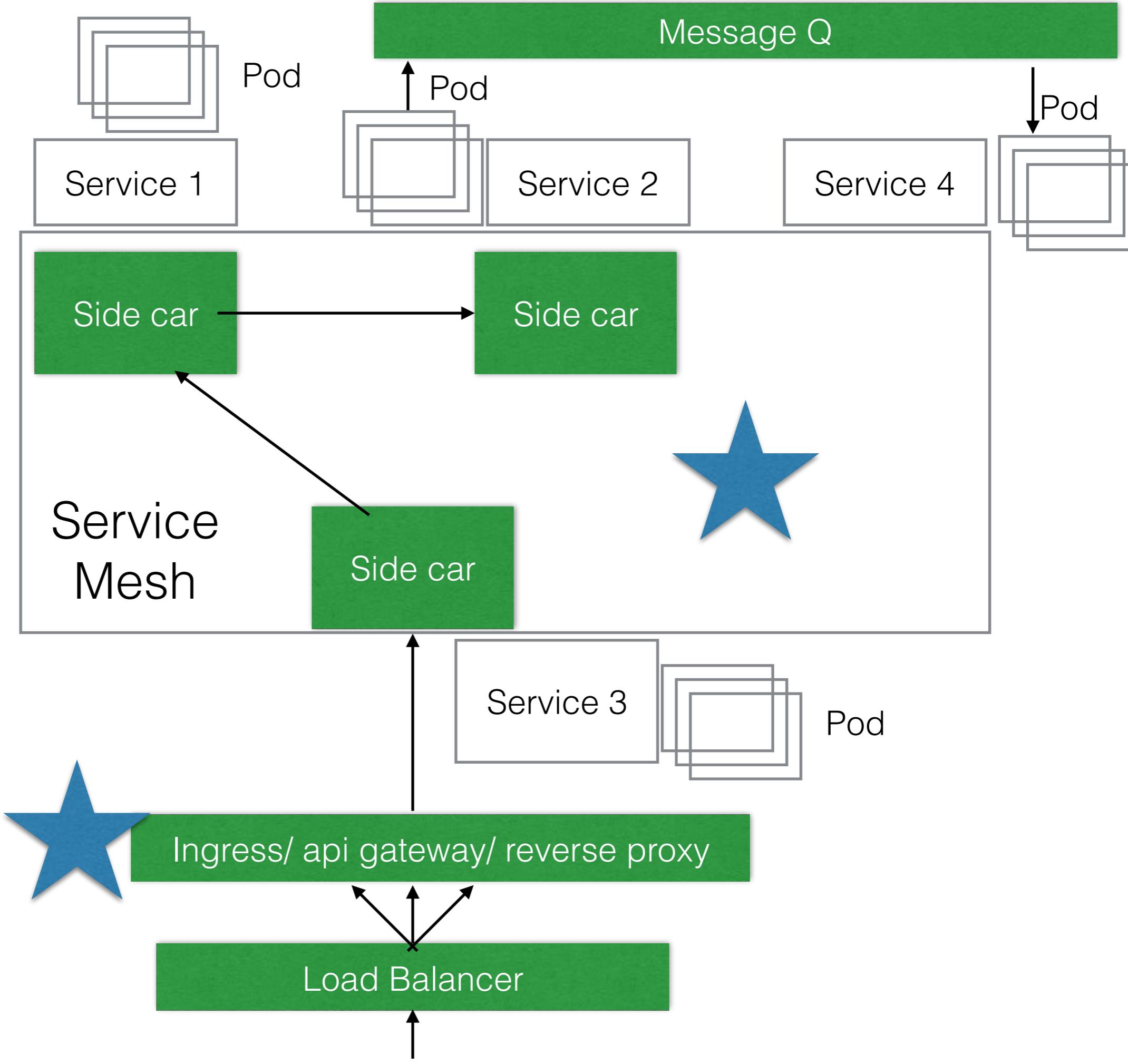




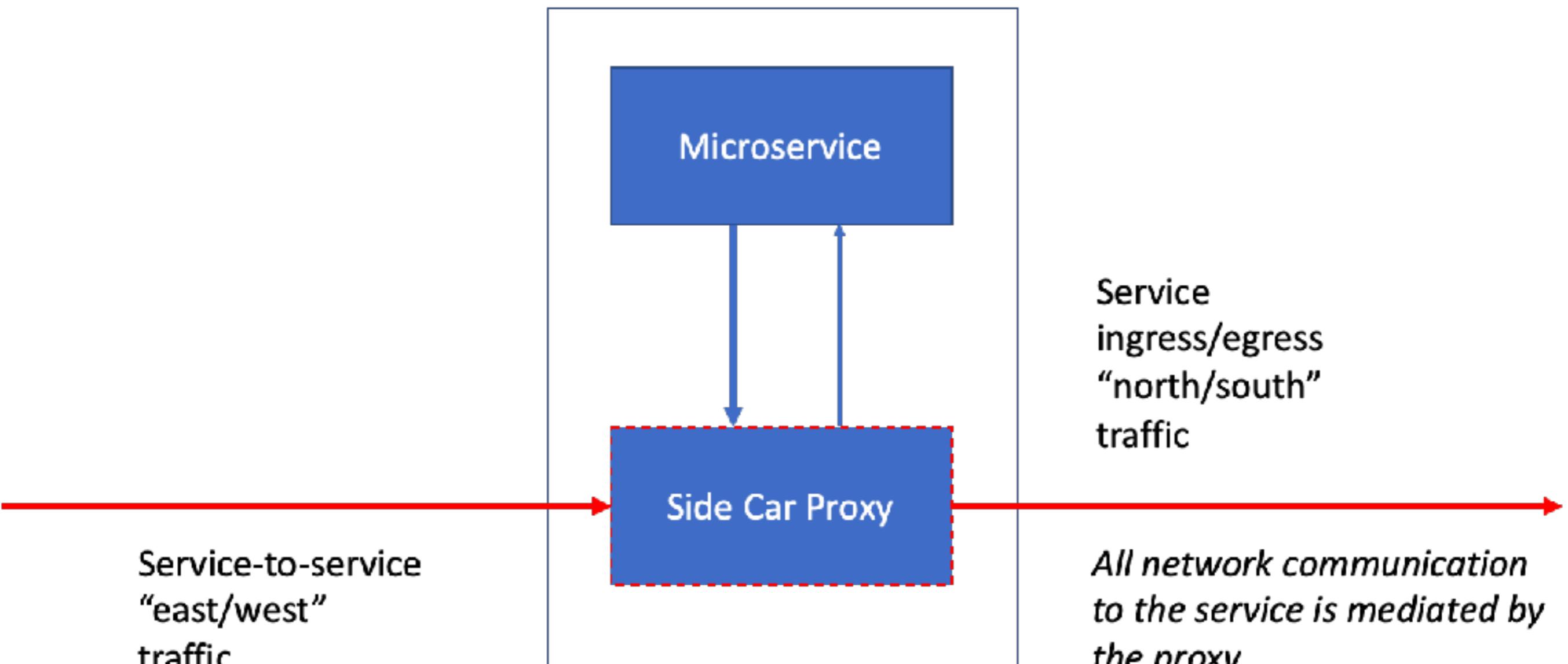




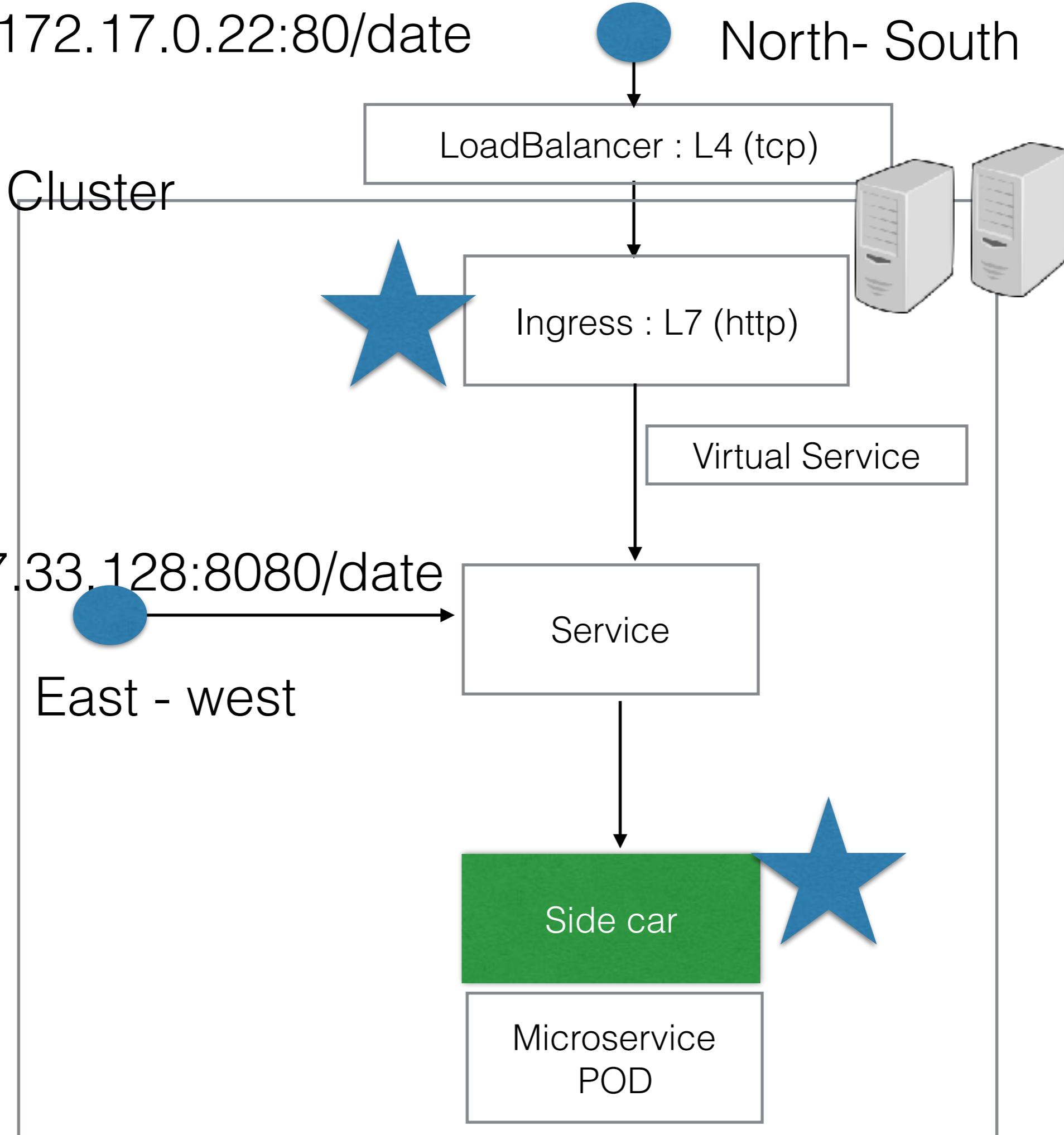




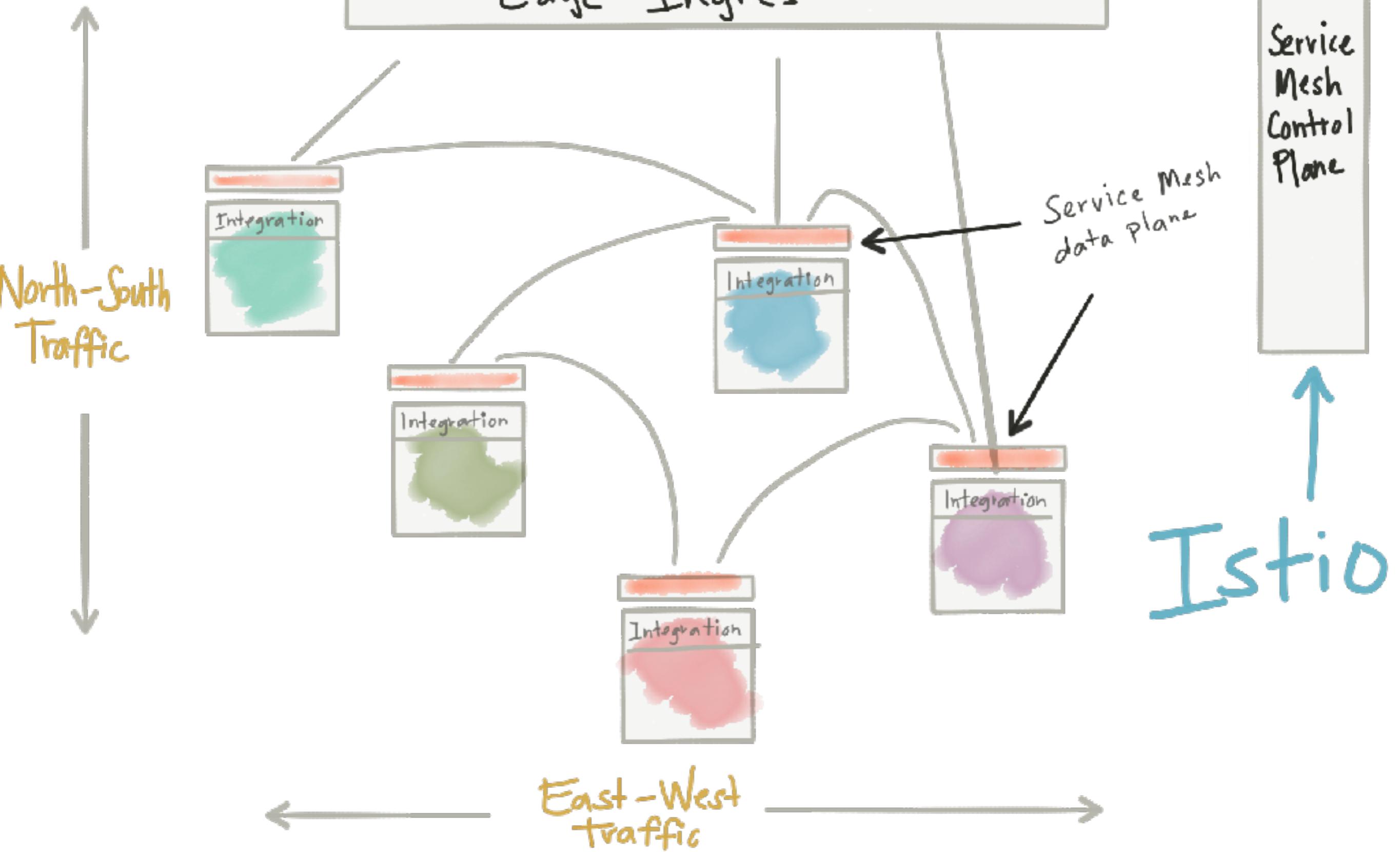
ECS Task/Kubernetes Pod

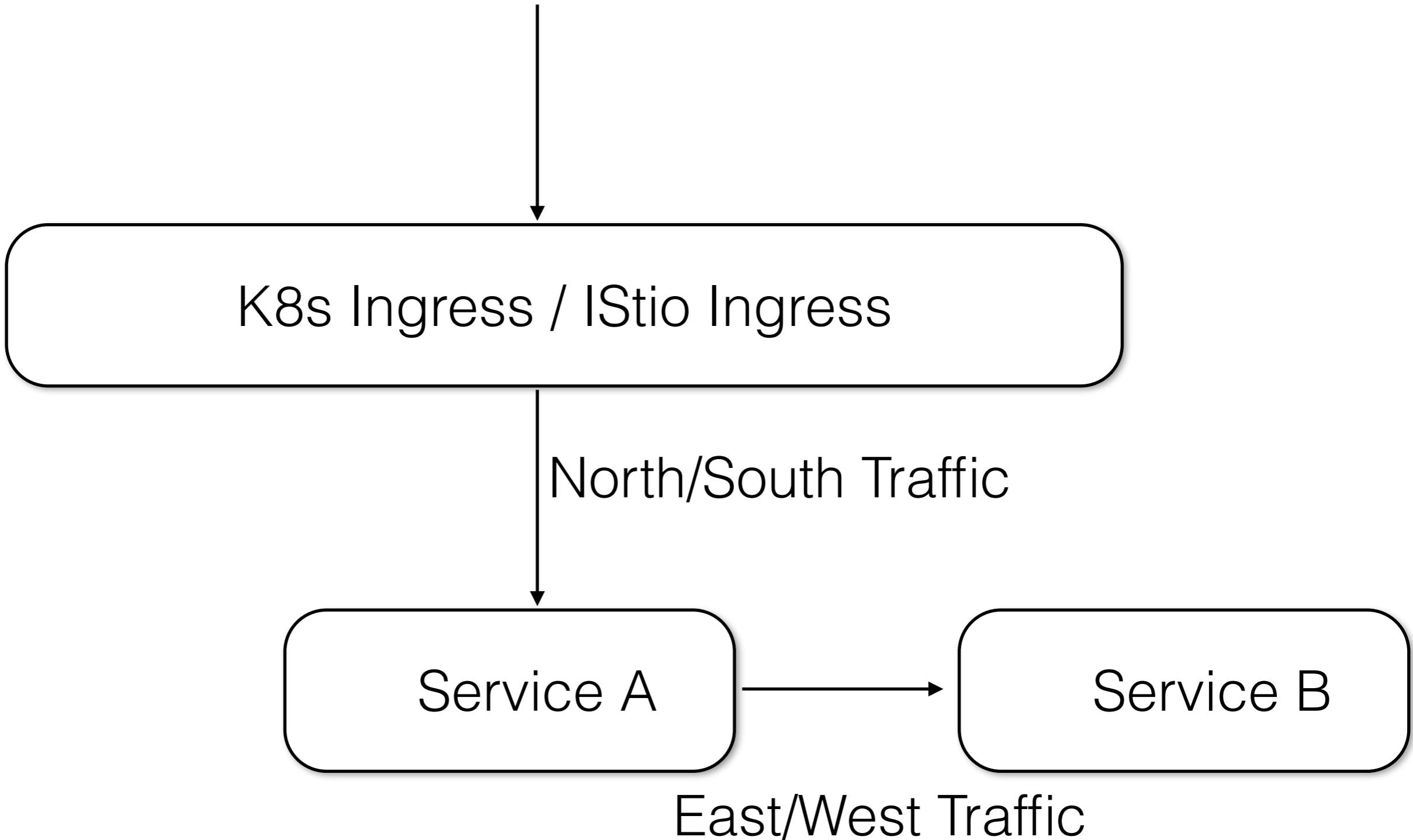


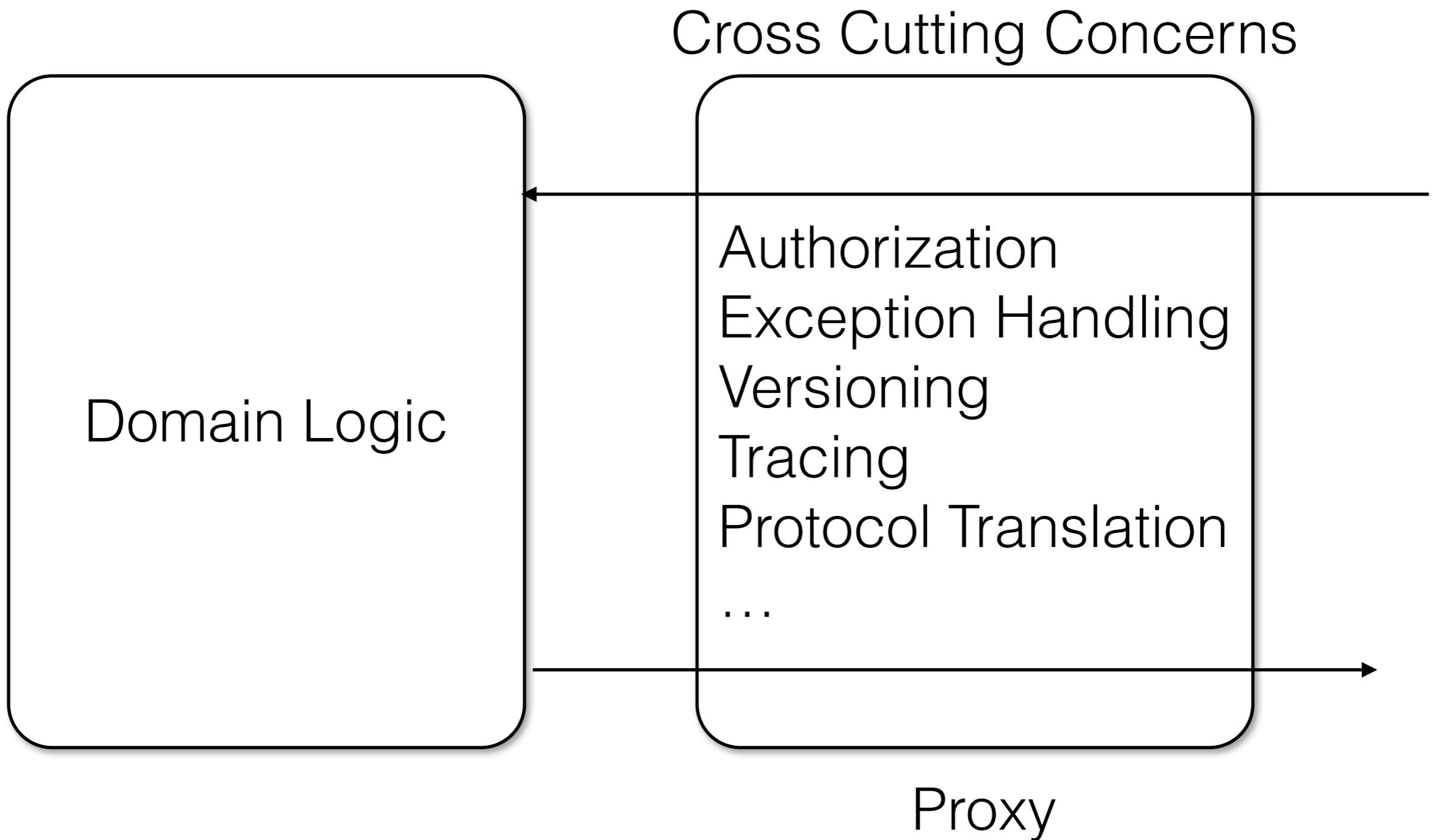
http://172.17.0.22:80/date



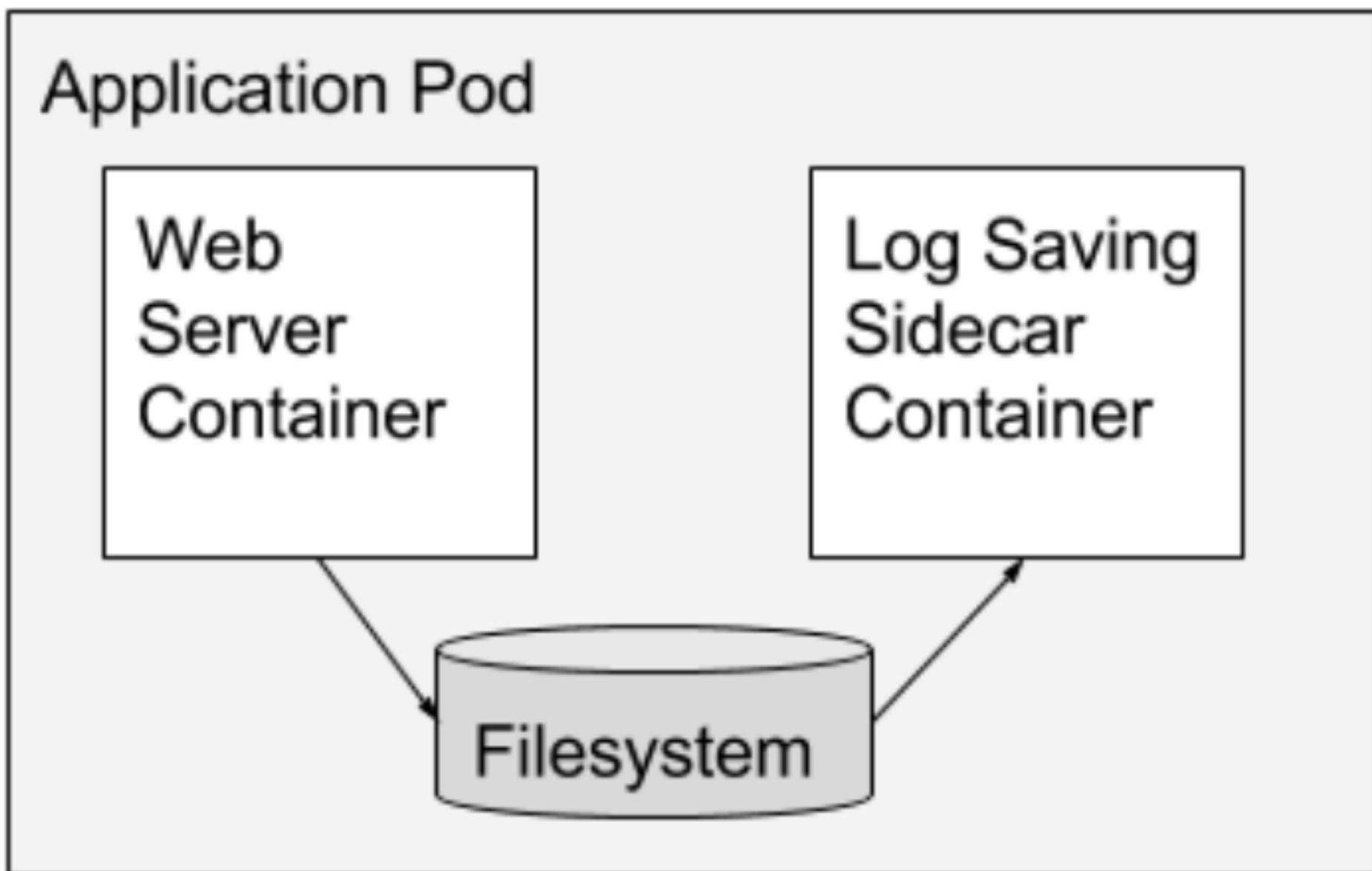
Edge Ingres





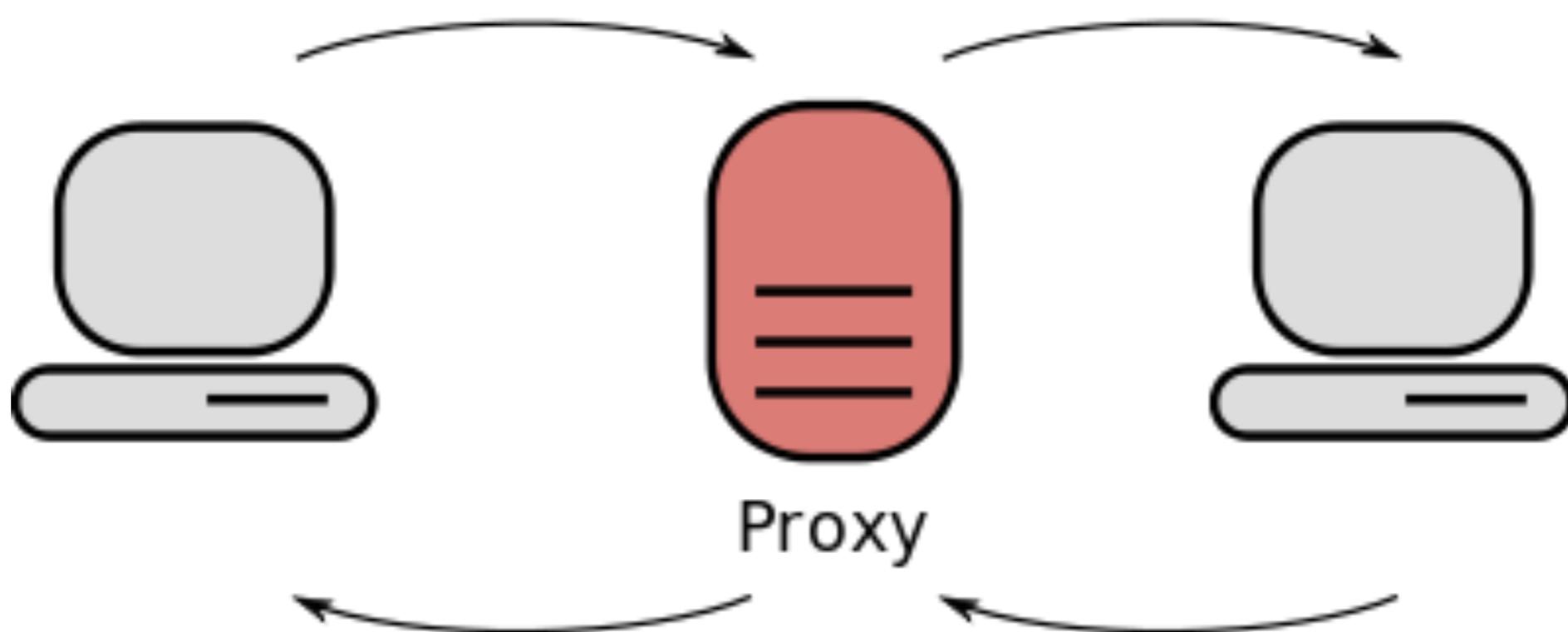


Sidecar



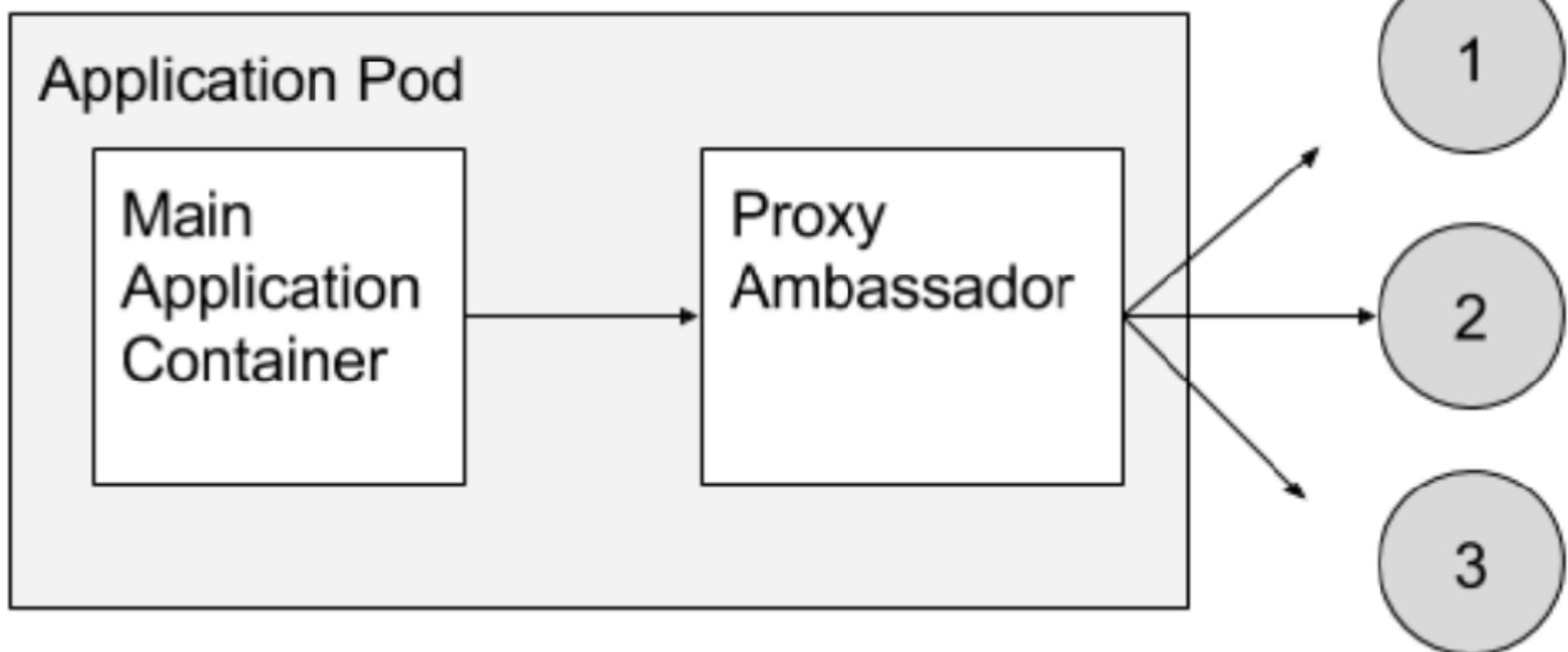
- In a sidecar pattern, the functionality of the main container is extended or enhanced by a sidecar container without strong coupling between two.
- eg. main container is a web server which is paired with a log saver sidecar container that collects the web server's logs from local disk and streams them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

Proxy pattern



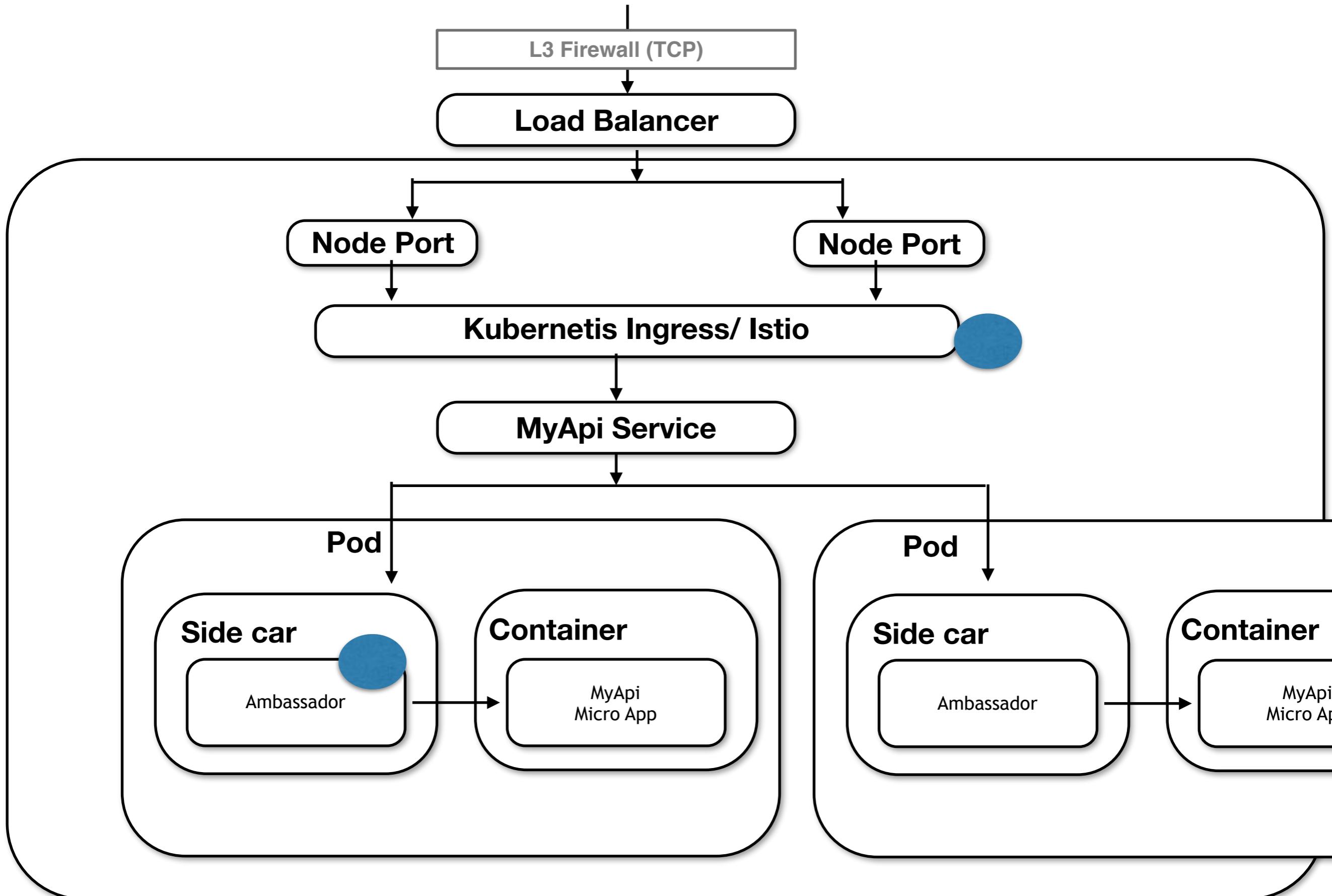
- Proxy pretends being an object when in fact it's not.
There are several types of proxies:
- Remote proxy ("Ambassador") - instead of communicating between server's and client's classes we create a proxy on client side and server side, and only the proxies communicate
- Virtual proxy - creates expensive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

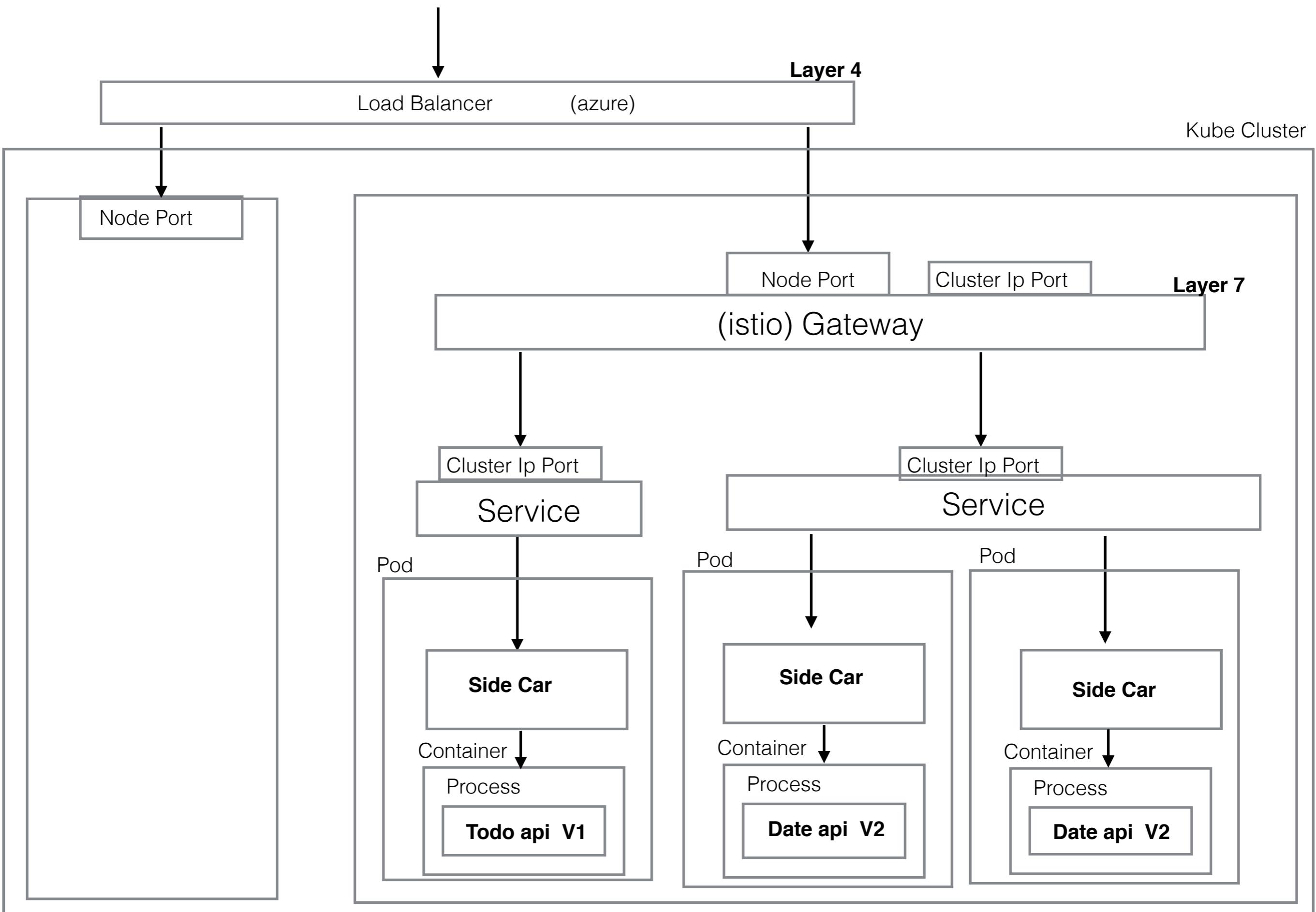
Ambassador pattern

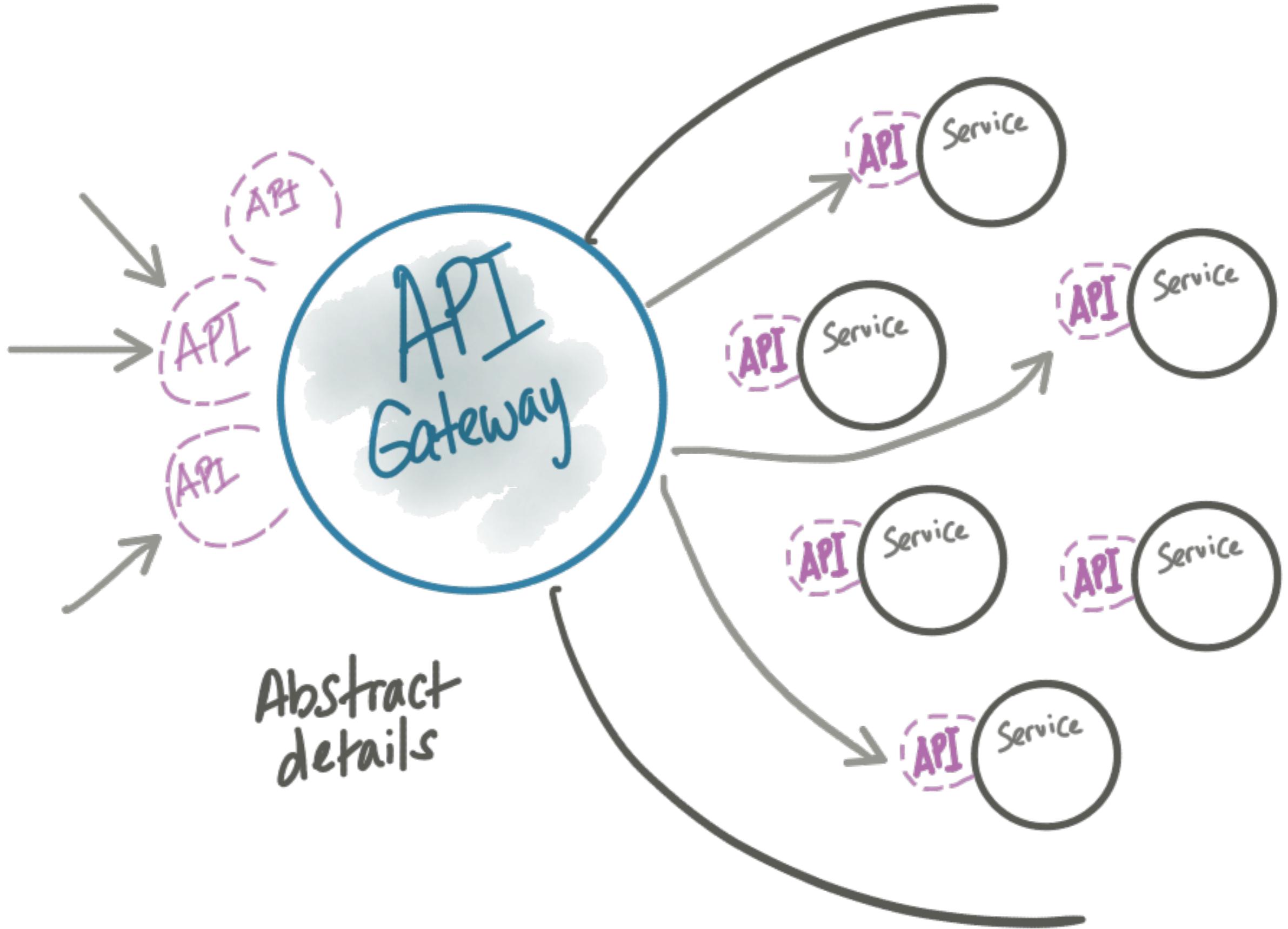


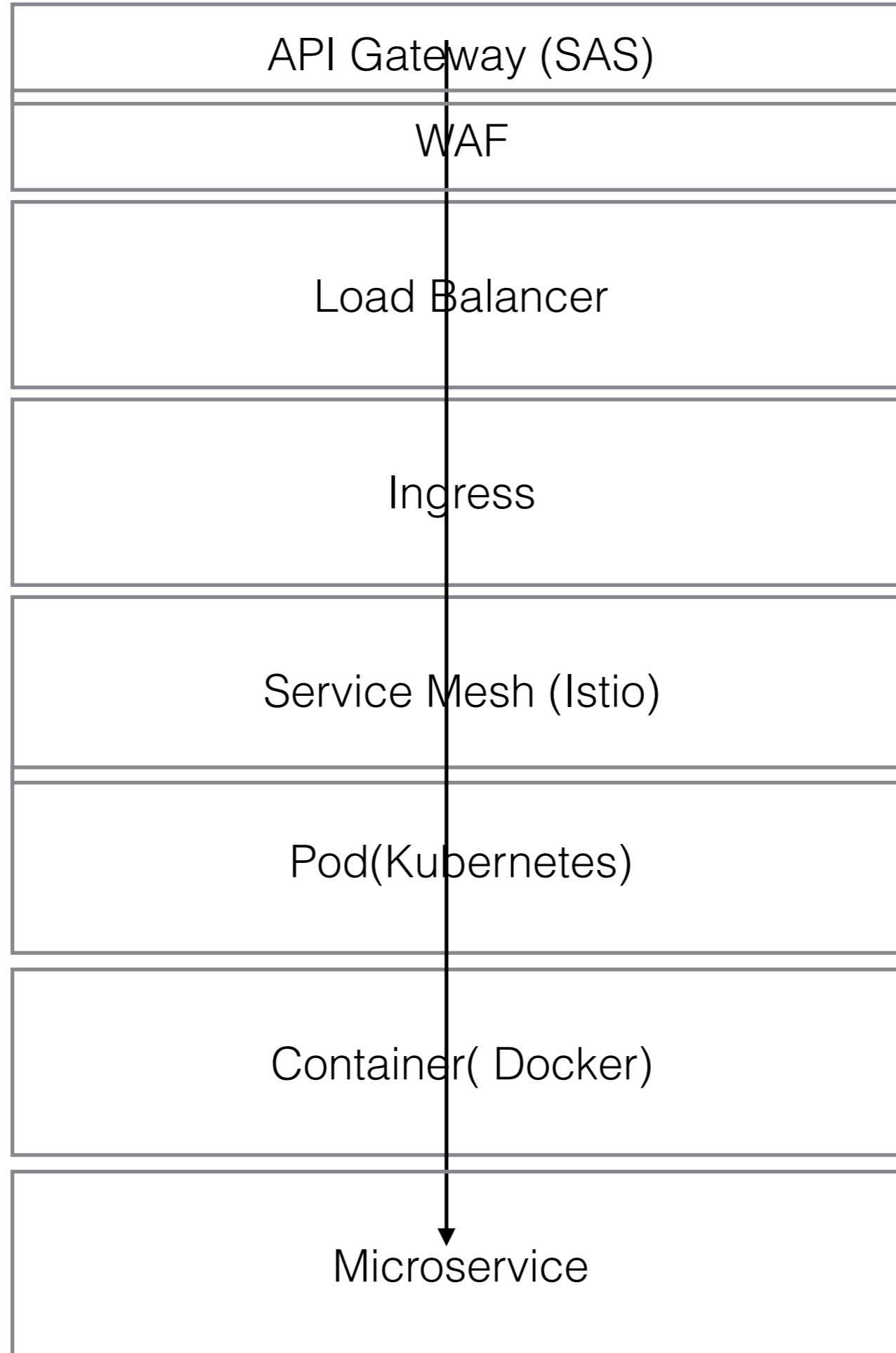
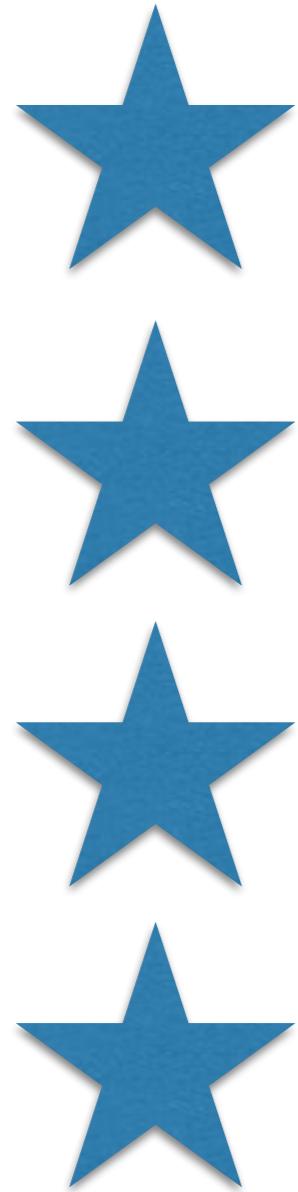
- an Ambassador container act as a proxy between two different types of main containers. Typical use case involves proxy communication related to load balancing and/or sharding to hide the complexity from the application.

Deployment Diagram







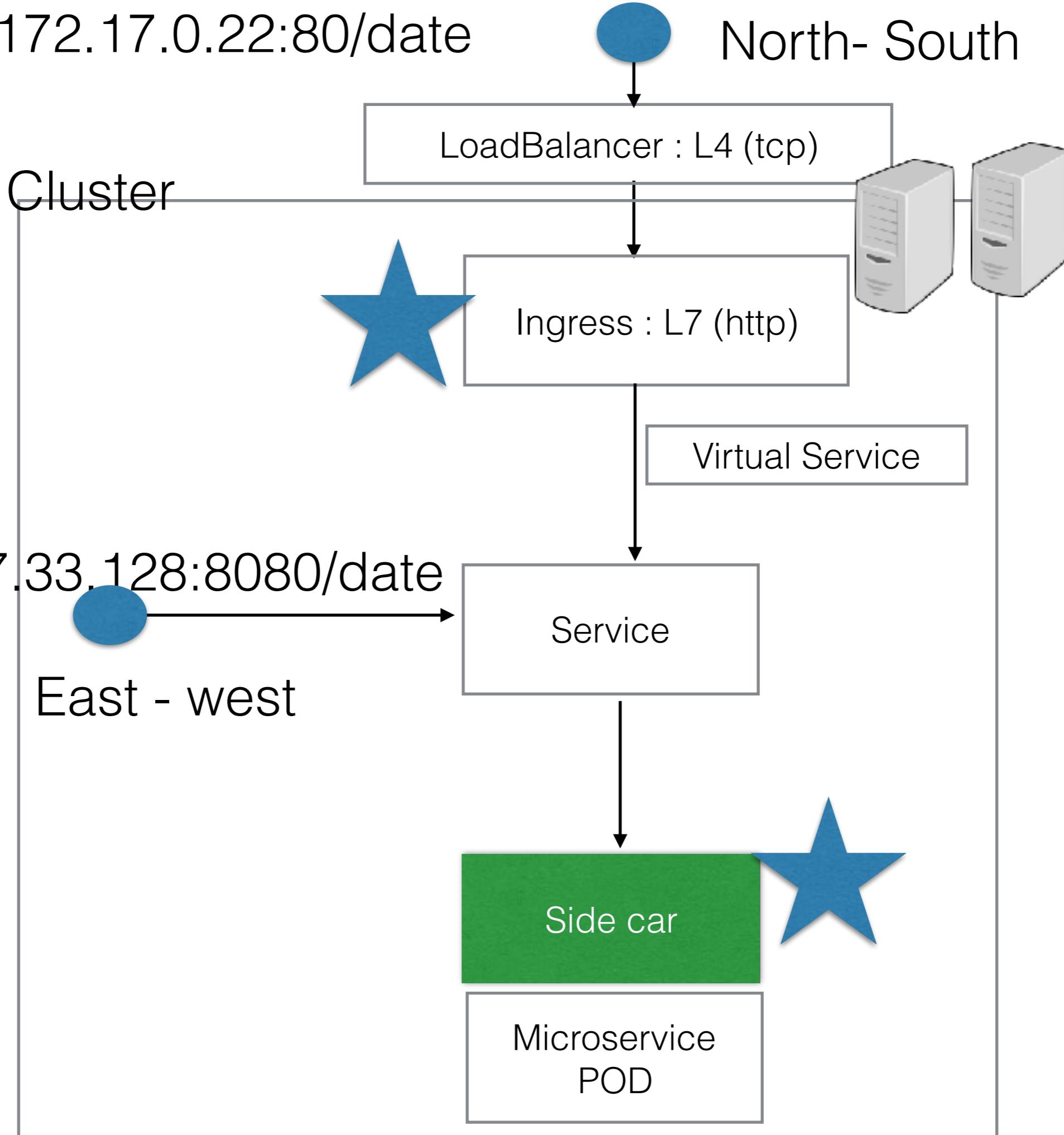


L4
North south

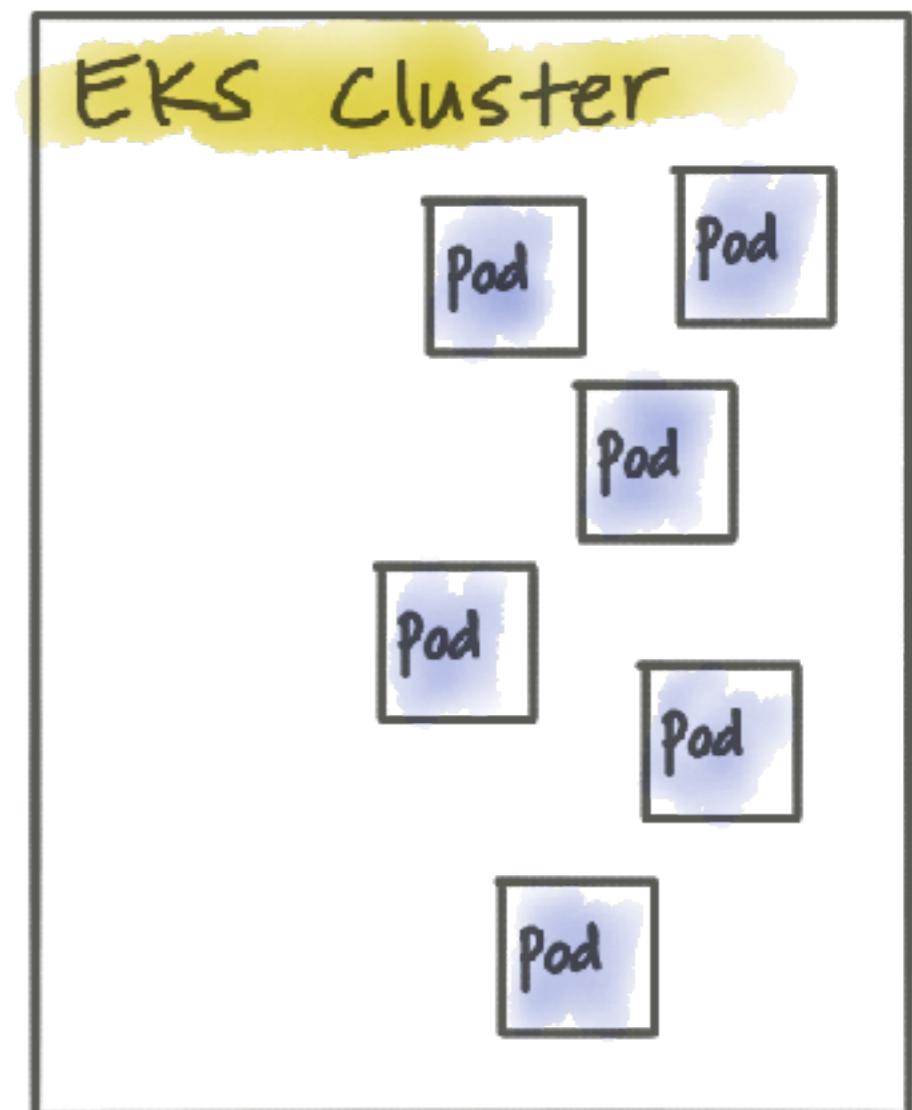
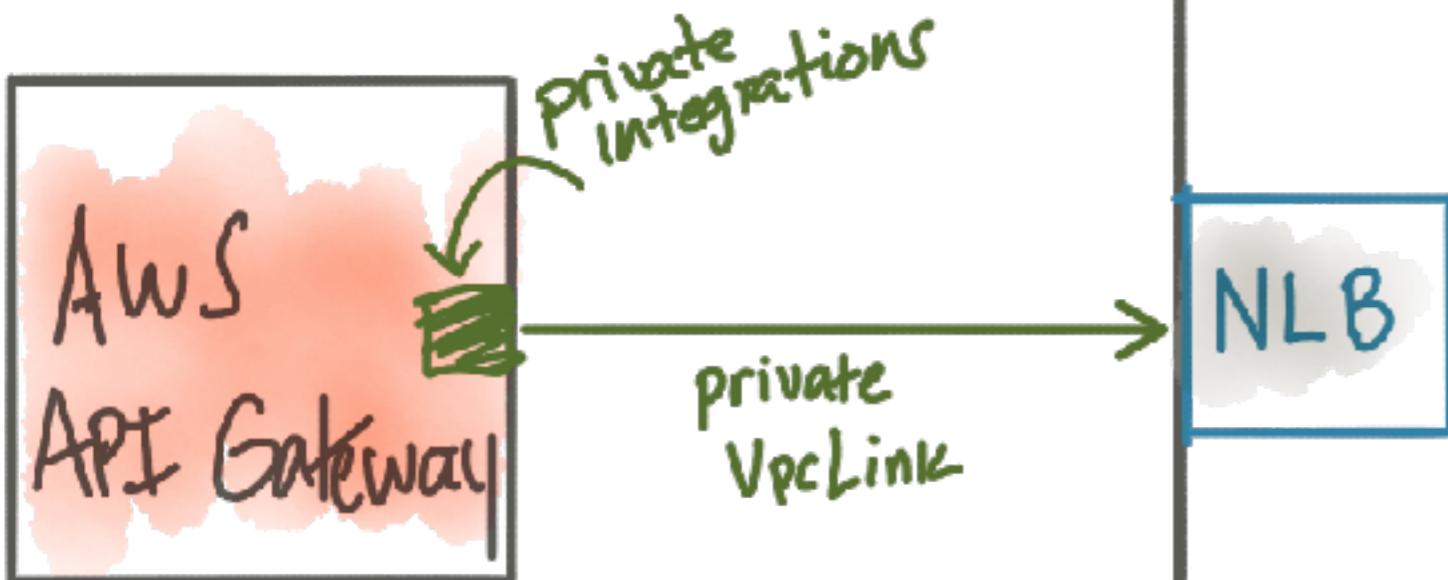
L7
North south

L7
East West

http://172.17.0.22:80/date



VPC / private subnets

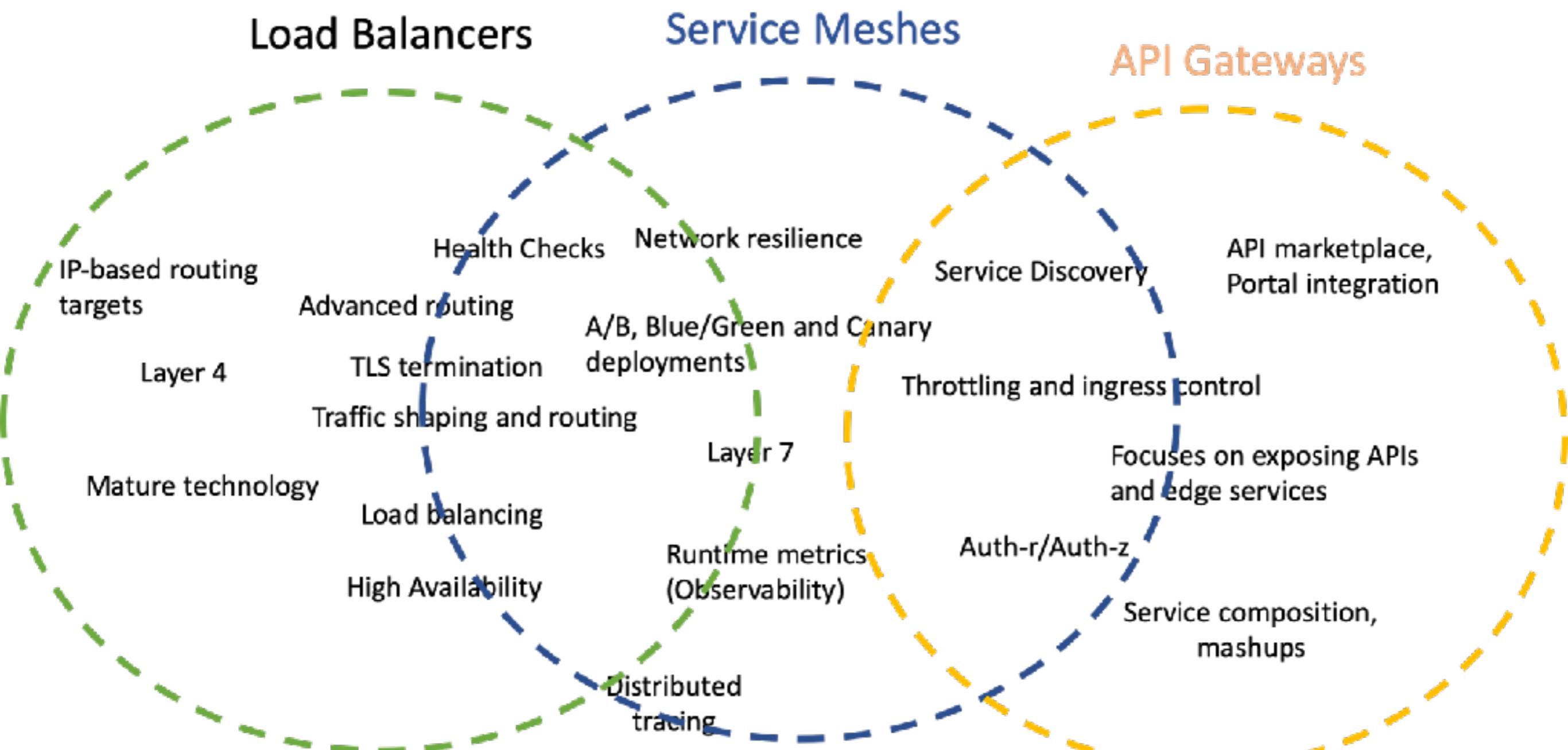


AWS API Gateway runs in its own VPC and is completely managed so you cannot see any details about its infrastructure.

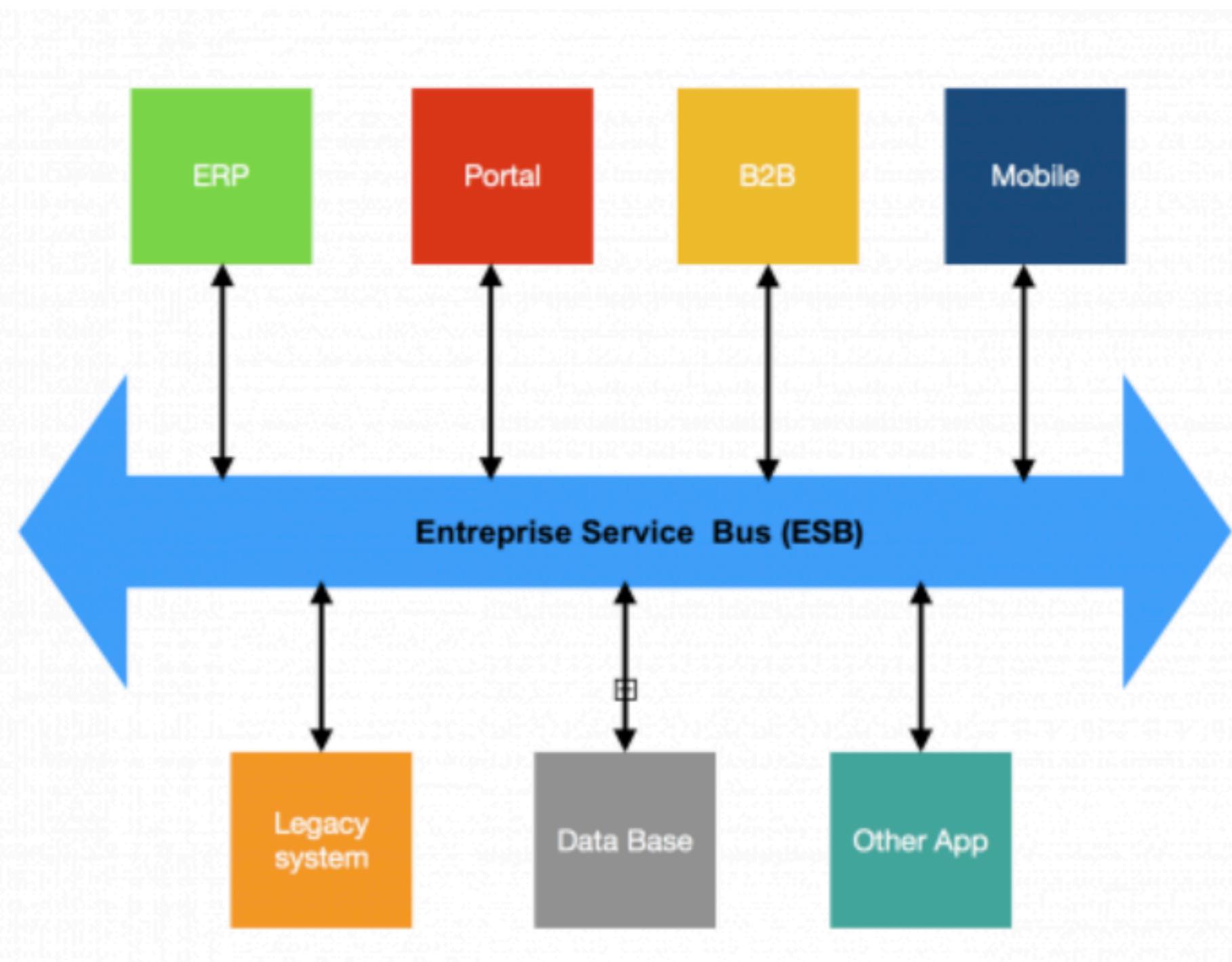
API Gateway : Abstraction,
decoupling,
edge routing / security

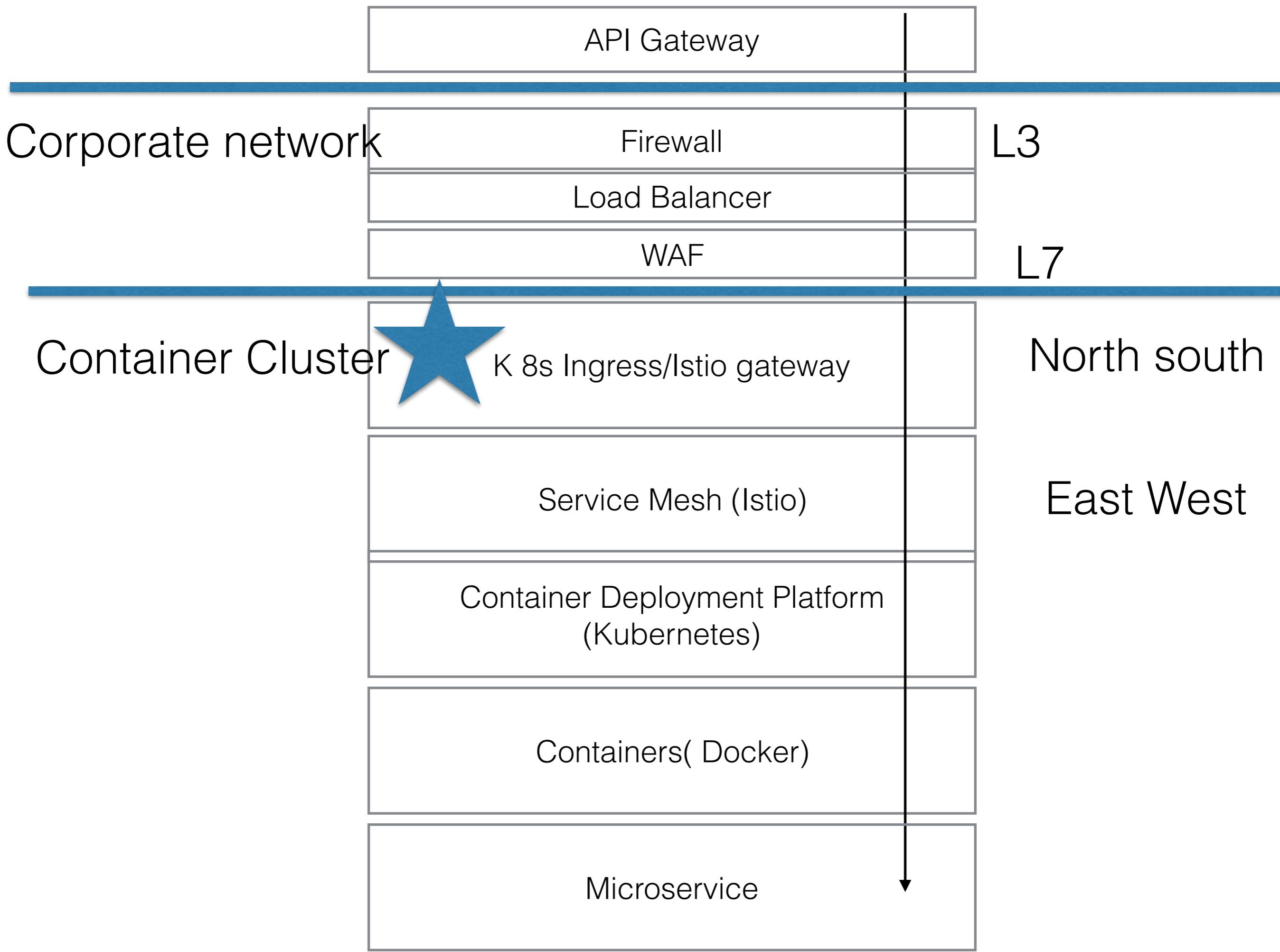
Service Mesh : endpoints, hosts, ports,
metric collection,
traffic routing, security

Deployment Platform : Cluster nodes,
container schedule,
resource management

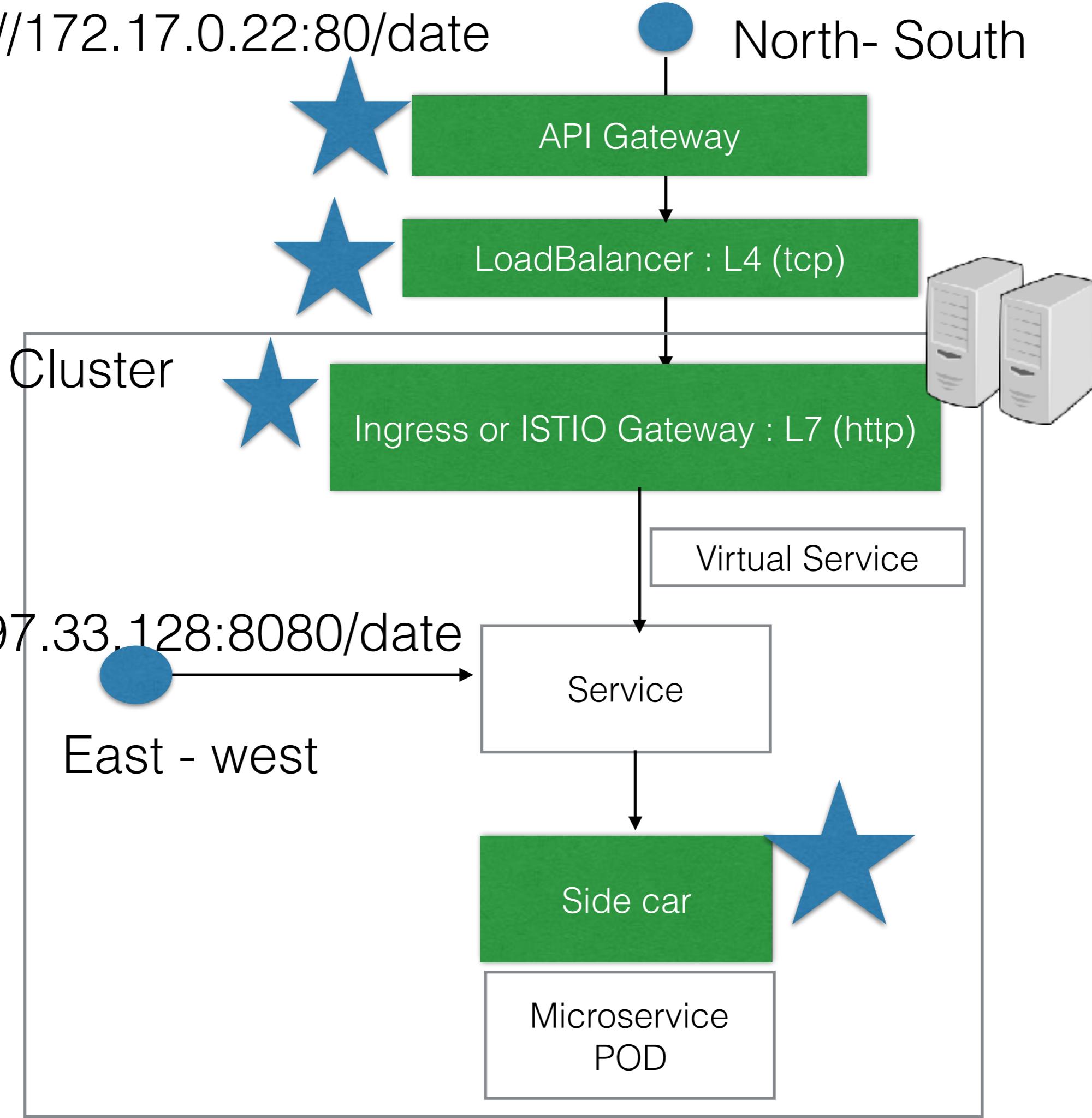


API Gateway vs ESB





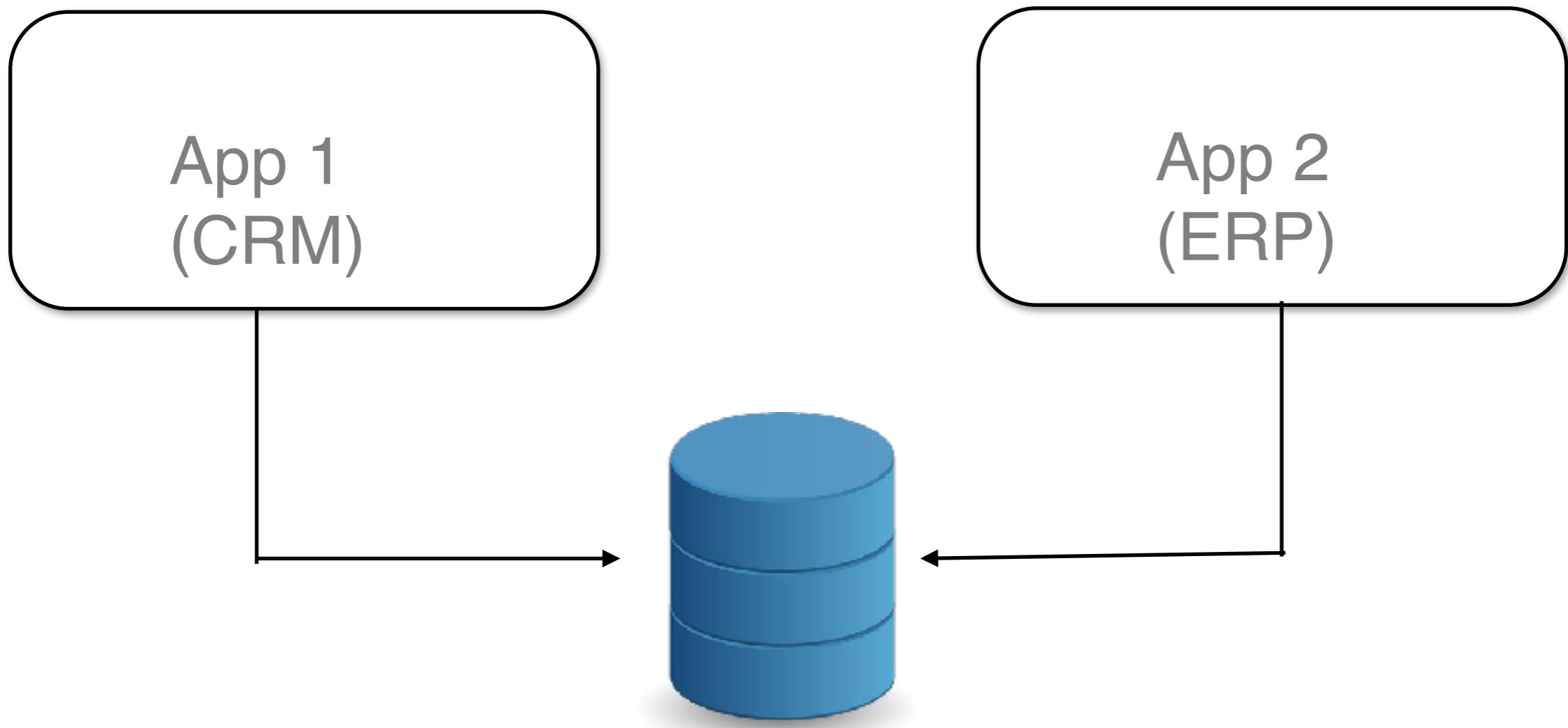
http://172.17.0.22:80/date



Sharing

Log Agg
Tracability
Monitoring

1. Database



2. Source Control

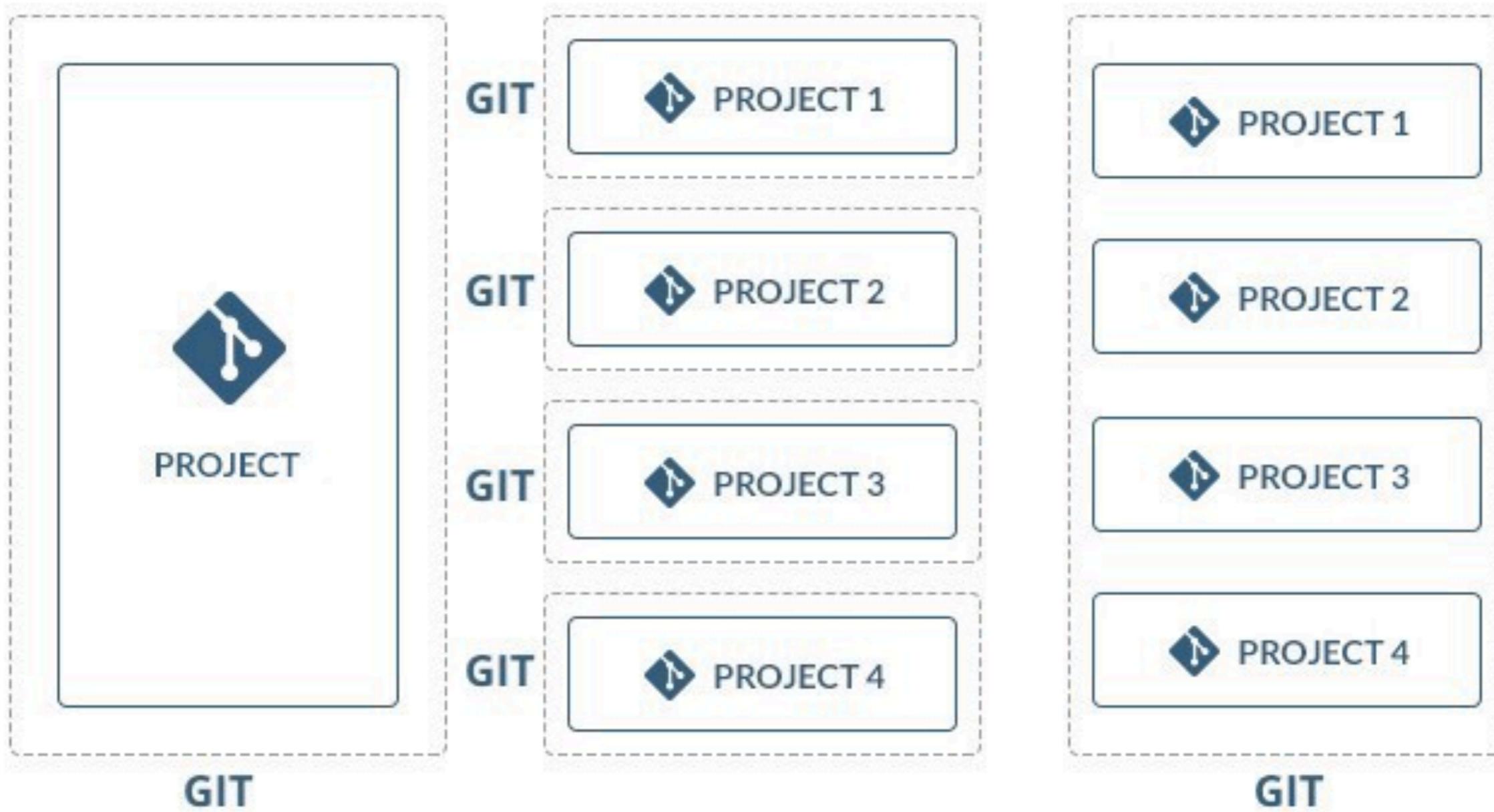
MONOLITH



MULTI-REPO



MONO-REPO

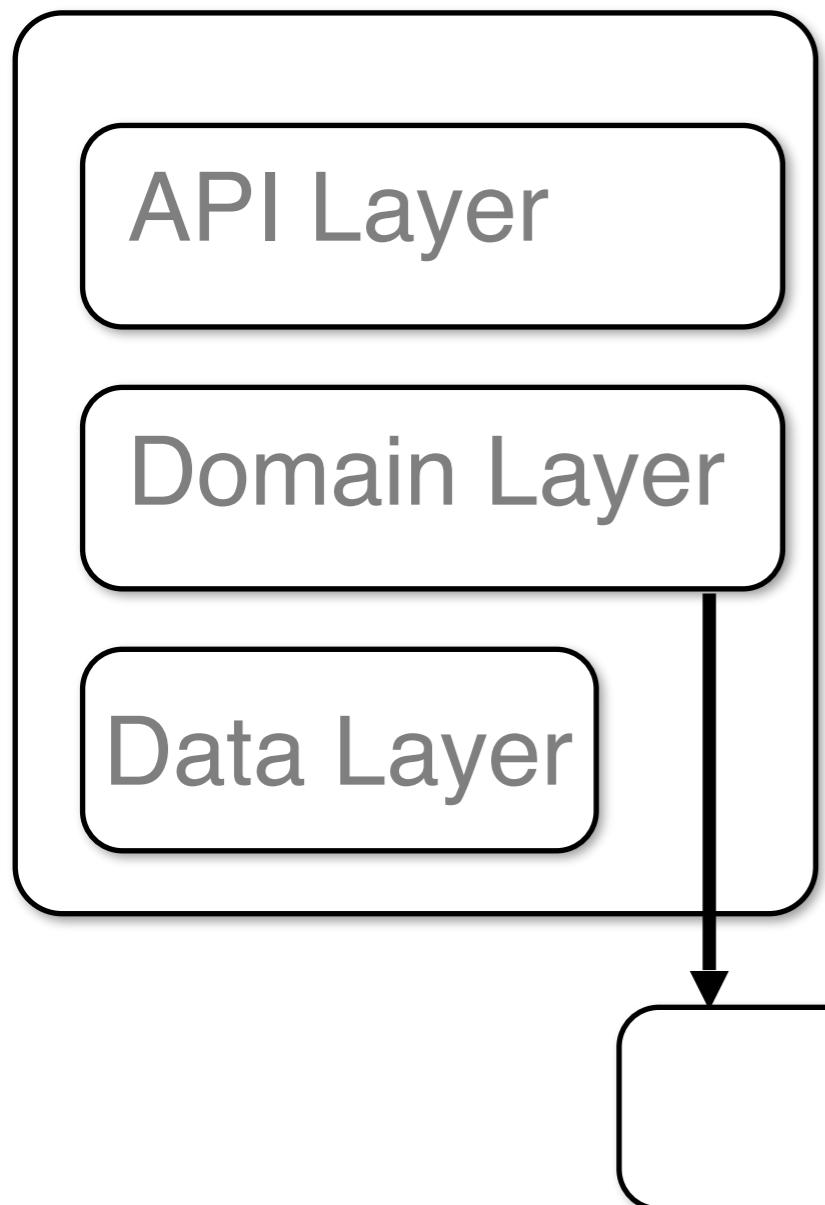


2. Source Control

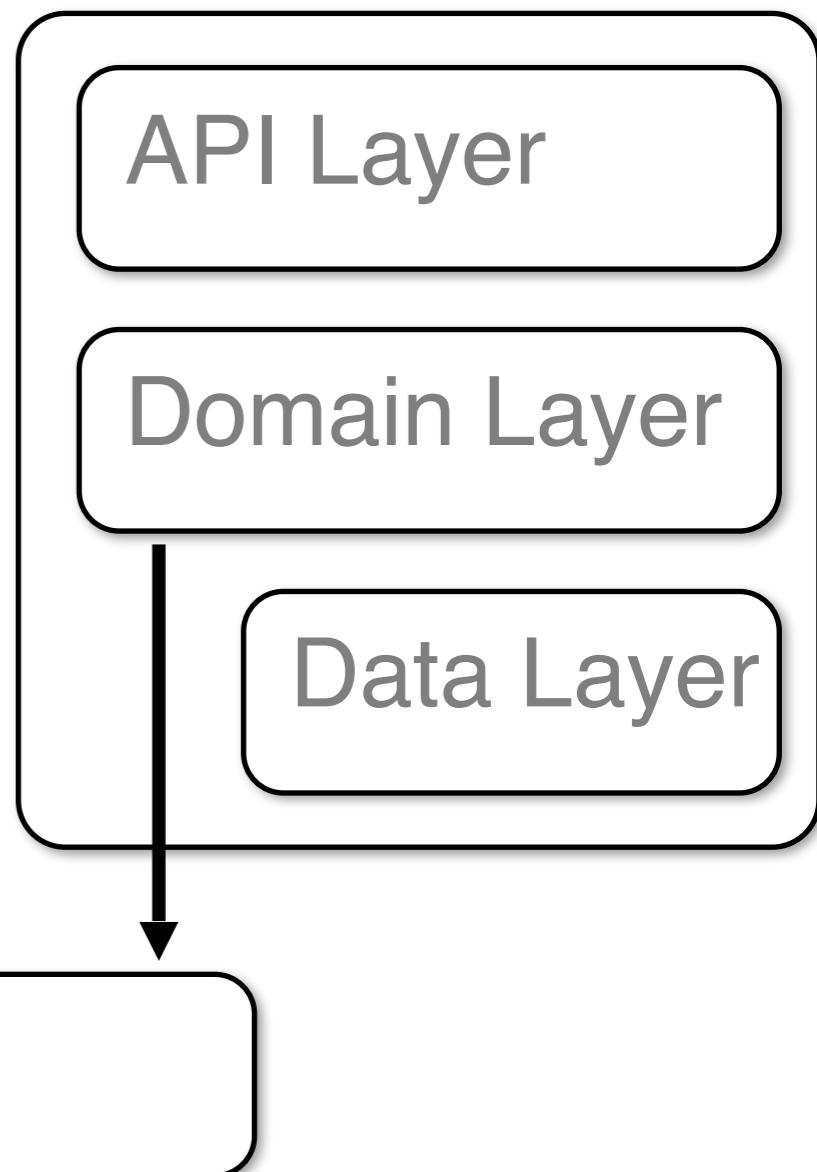
	Monorepo	Poly repos
Advantages	Code sharing Easier to standardize code and tooling Easier to refactor code Discoverability - single view of the code	Clear ownership per team Potentially fewer merge conflicts Helps to enforce decoupling of microservices
Challenges	Changes to shared code can affect multiple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

3. Common Logic

Purchase
Module



Sales
Module

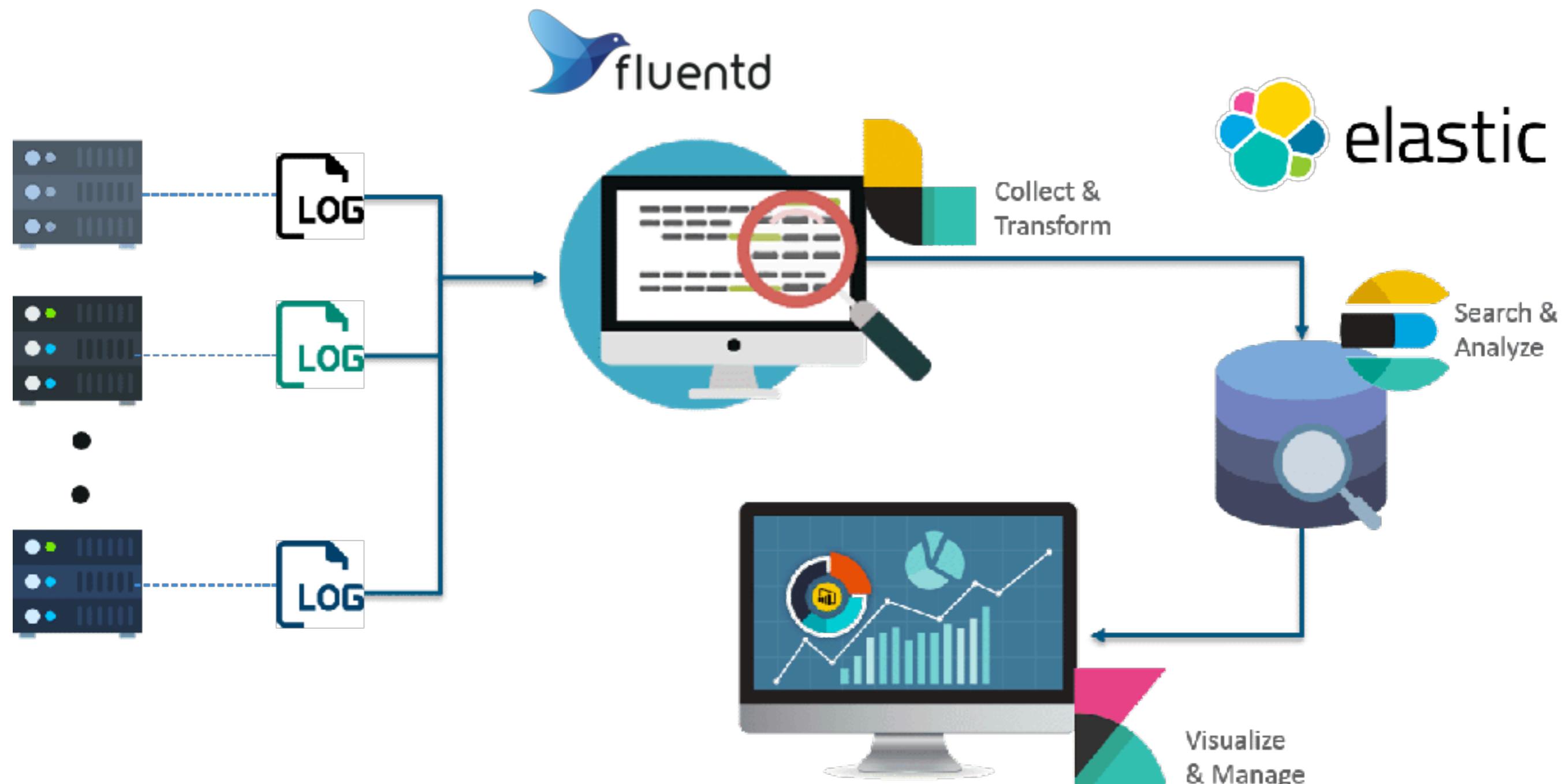


Pure fabrication # stateless, domain independent

Monitoring

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

Log Aggregation



EFK

Elastic Search : db
Kibana : UI
Fluentd : agg

Health Endpoint Monitoring



Cloud Deployment

Storage
Compute
Messaging

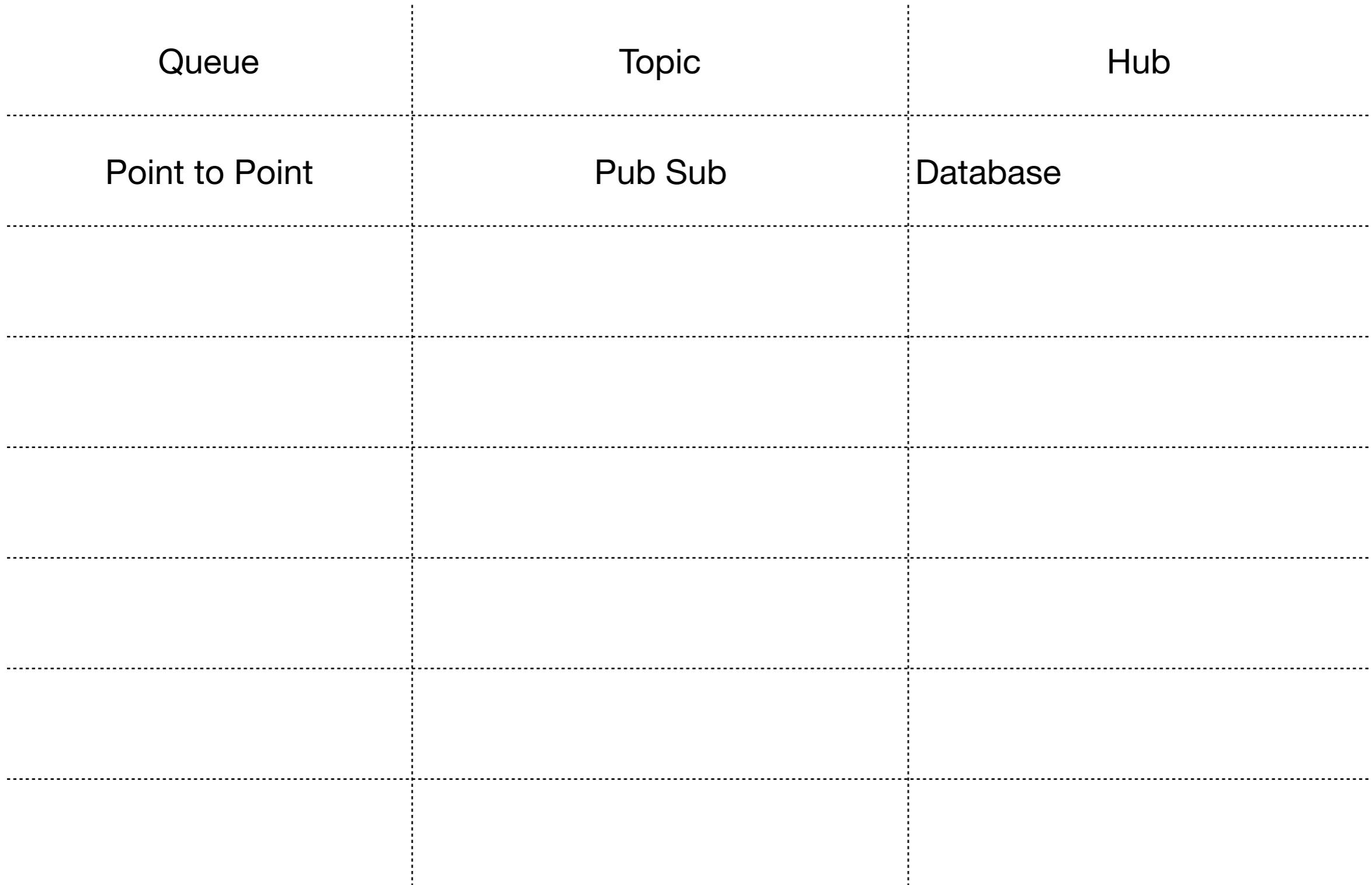
Persistence Matrix

Disk Based File Storage	Rdbms	Document	Key- Valued	Graph	Column DB	Time Series	Distributed File Storage
Disk	MySQL Oracle Postgres	Mongo Dynamo	Redis Couch	Neo4j	HBase Cassandra	Influx TSDB	S3 Azure Blob
Binary	Referential Integrity	Hierarchy (Tree)	KEY = VALUE	Graph	Big Data	Time	Binary
No Duplicates	Tree queries	Performance	Graph queries	TB's	Time series queries		Multiple copies
Transaction (ACID)			Transactio n (ACID)				

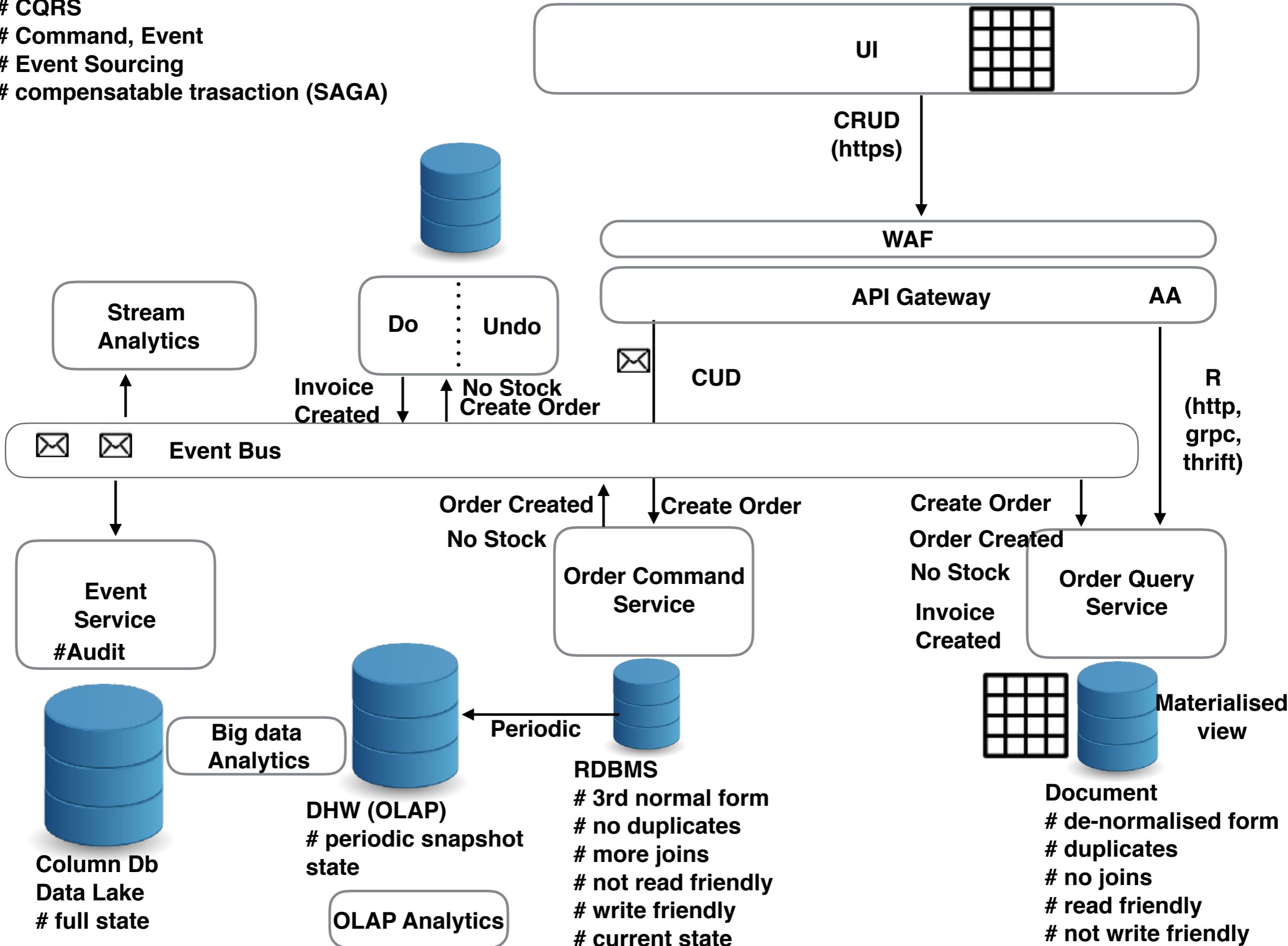
Application Hosting Matrix

IAAS	Managed OS + Middlewear	Serverless Middlewear (PAAS 2. 0)	Managed OS With Docker	Managed Distributed Containers	Serverless Distributed Containers
VM (1 or *)	VM (1 or *) with Webserver	VM (1 or *) with Webserver		VM (1 or *) With Kuberenetes	VM (1 or *) With Kuberenetes
Win Unix ...	Tomcat, Jboss, Nginx, ...				
Ec2	Beanstalk	aws Lamda		aws	aws
Azure VM	Azure App service	Azure functions		Azure	Azure

Types of Messaging



CQRS
Command, Event
Event Sourcing
compensatable transaction (SAGA)



Pupil Table

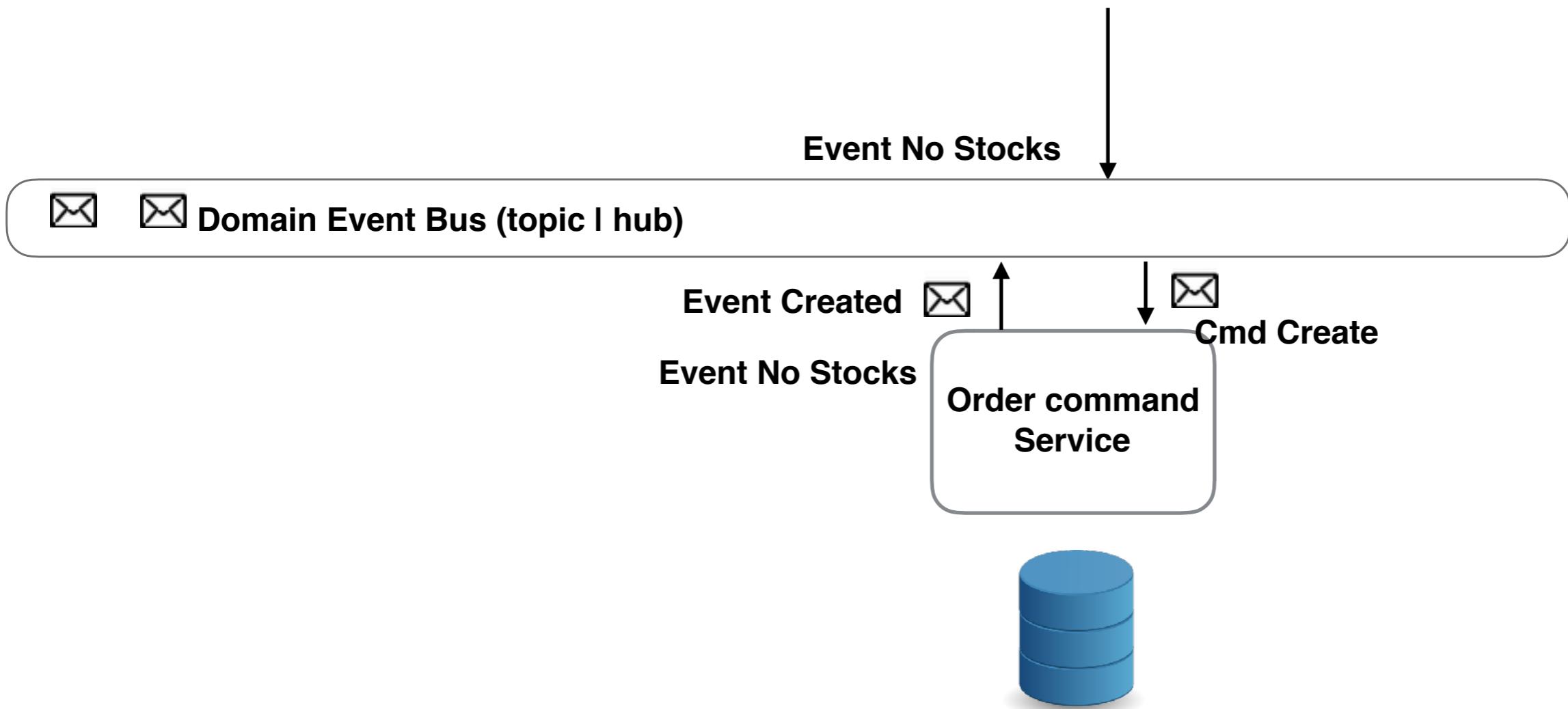
Pupil ID	First Name	Last Name	ClassID
1	Bob	Jones	1
2	Bill	Jones	2
3	Fred	Jones	1

Class Table

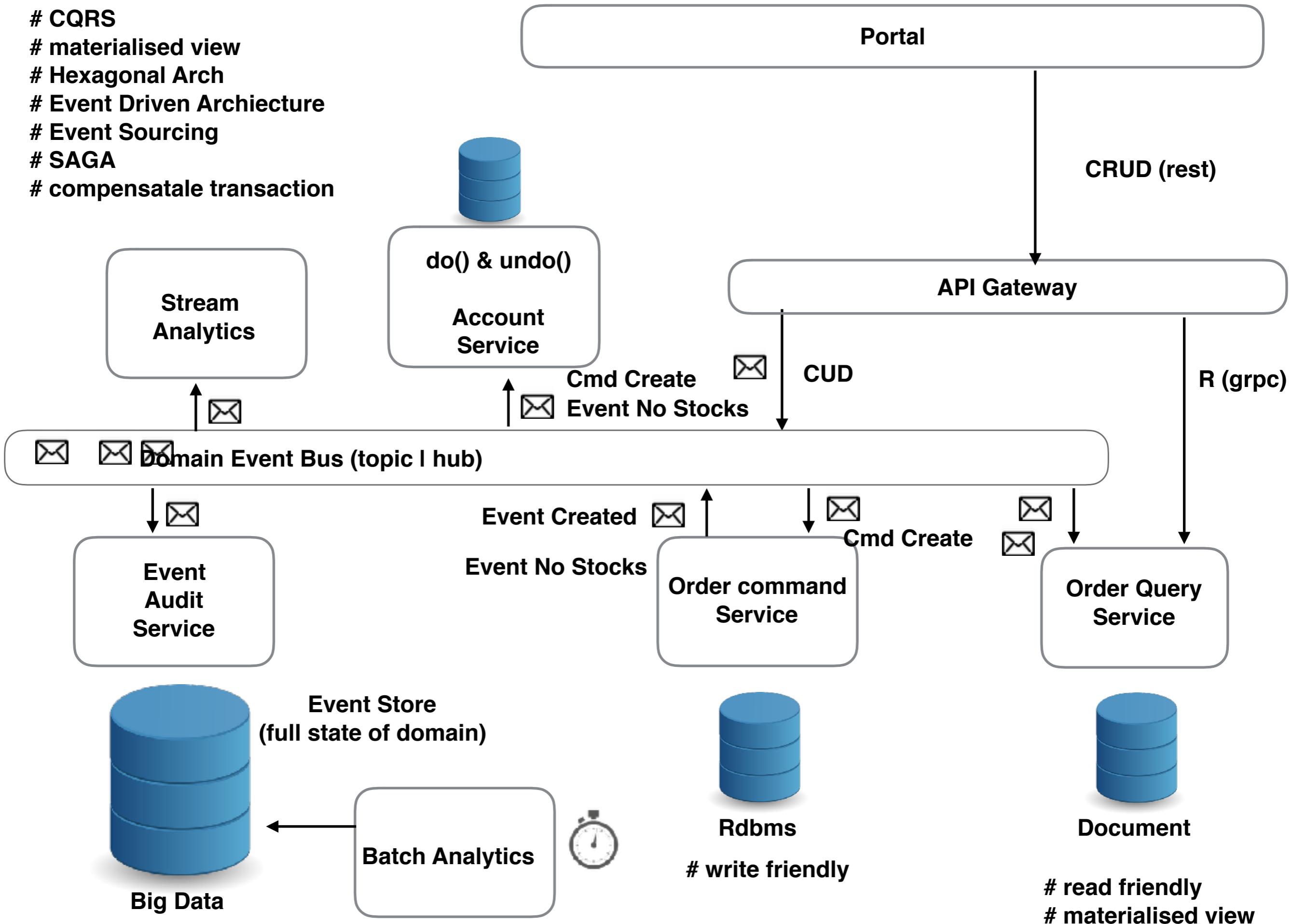
Class ID	Class Name	Room
1	CompSci 101	S16
2	English 101	M42

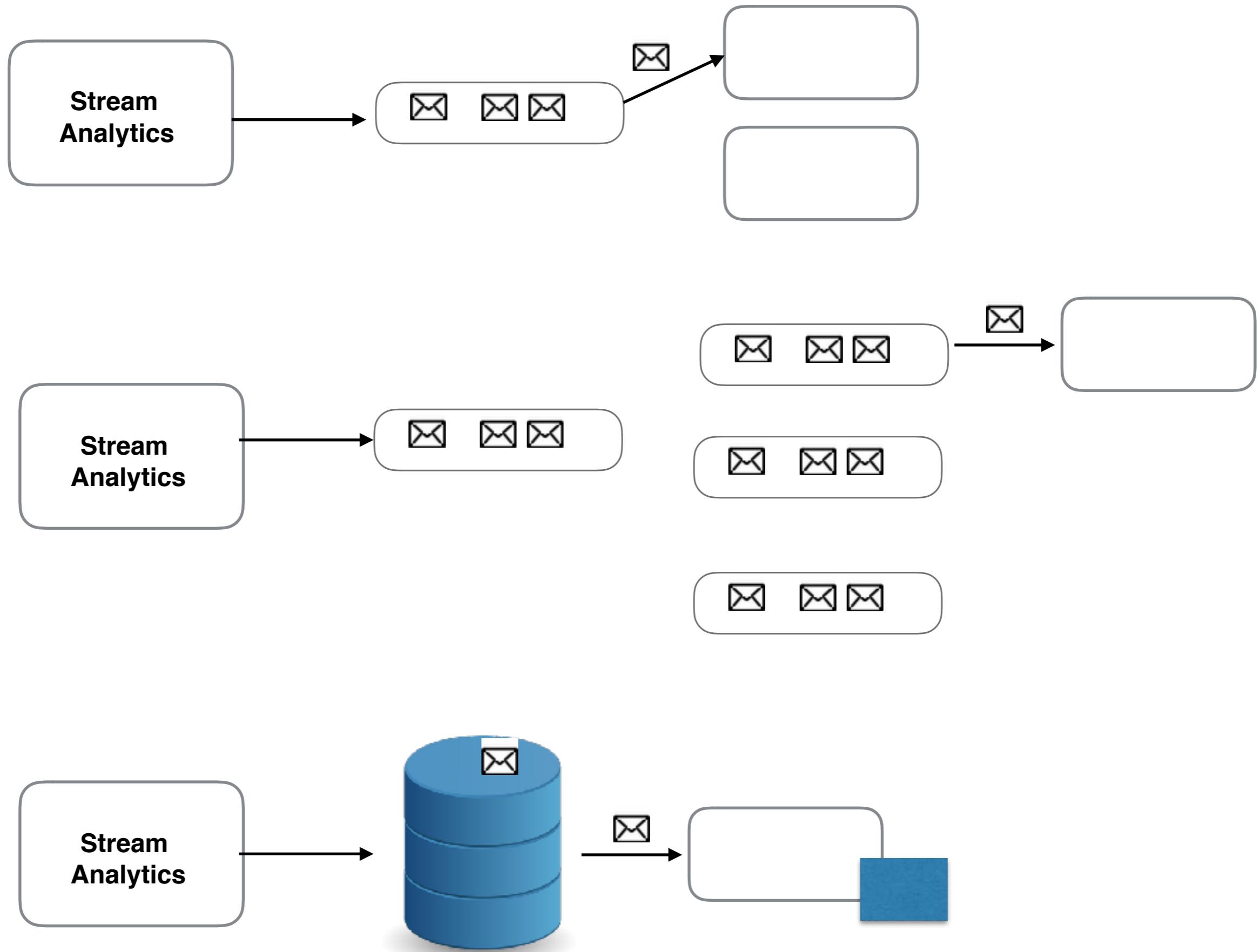
Weather

city	state	high	low
Phoenix	Arizona	105	90
Tucson	Arizona	101	92
Flagstaff	Arizona	88	69
San Diego	California	77	60
Albuquerque	New Mexico	80	72

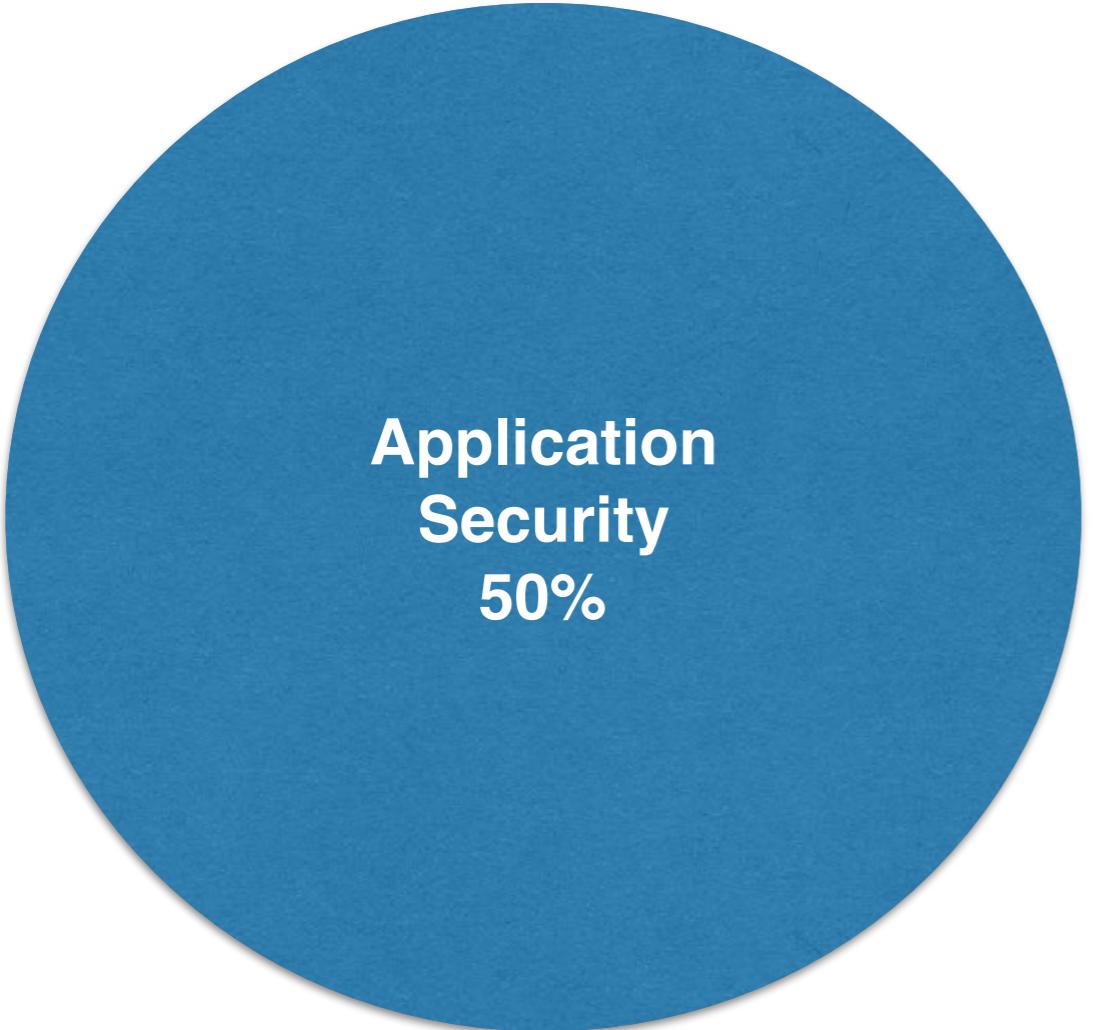


CQRS
materialised view
Hexagonal Arch
Event Driven Architecture
Event Sourcing
SAGA
compensatale transaction

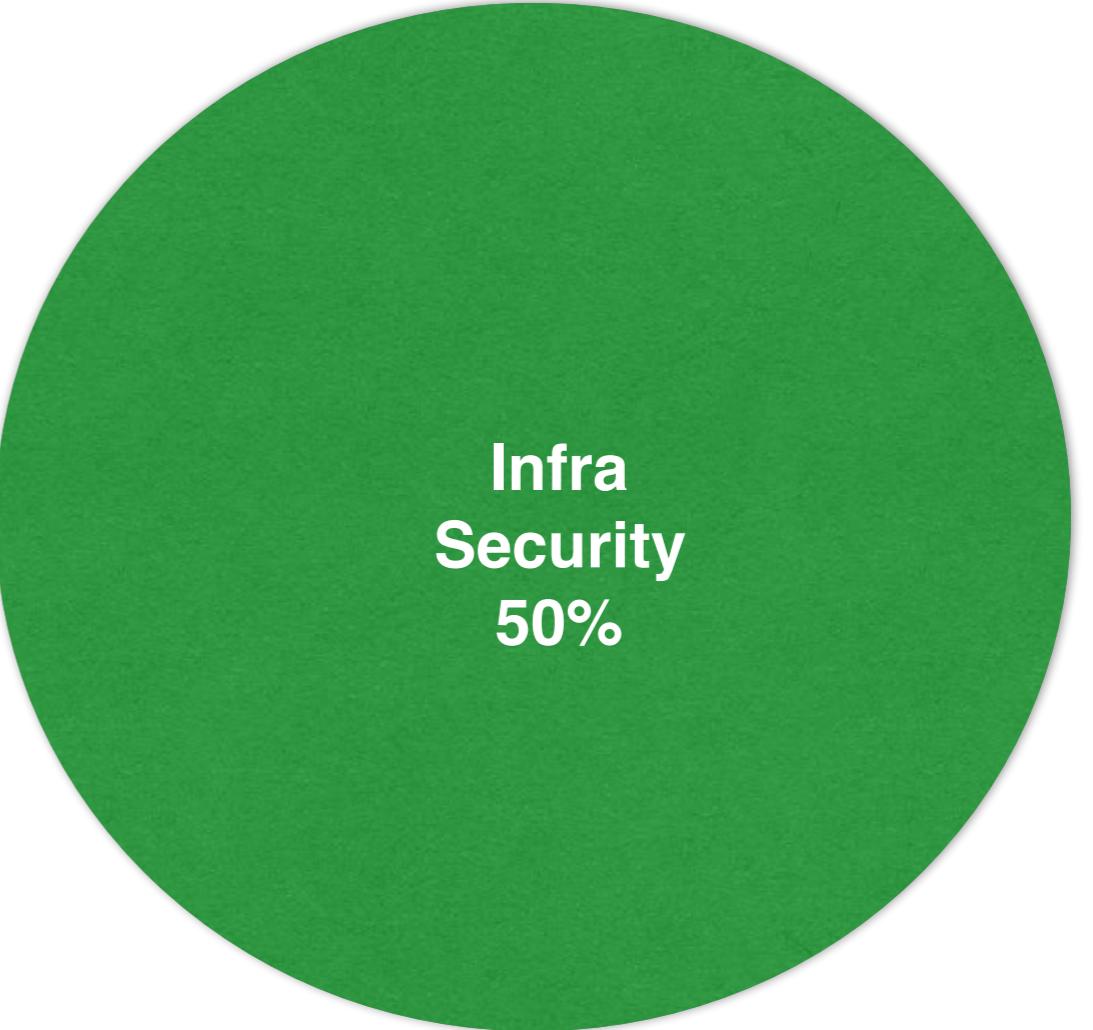








**Application
Security**
50%



**Infra
Security**
50%

Authentication

First Defense

Authorization

Auditing - Last Defense

Data in Transit

Data in Rest

Input Validation

Exception Handling

Session Management

Secret Management

Throttling

L4 Firewall (TCP)

DOS, ..

L7 Firewall (http) - WAF

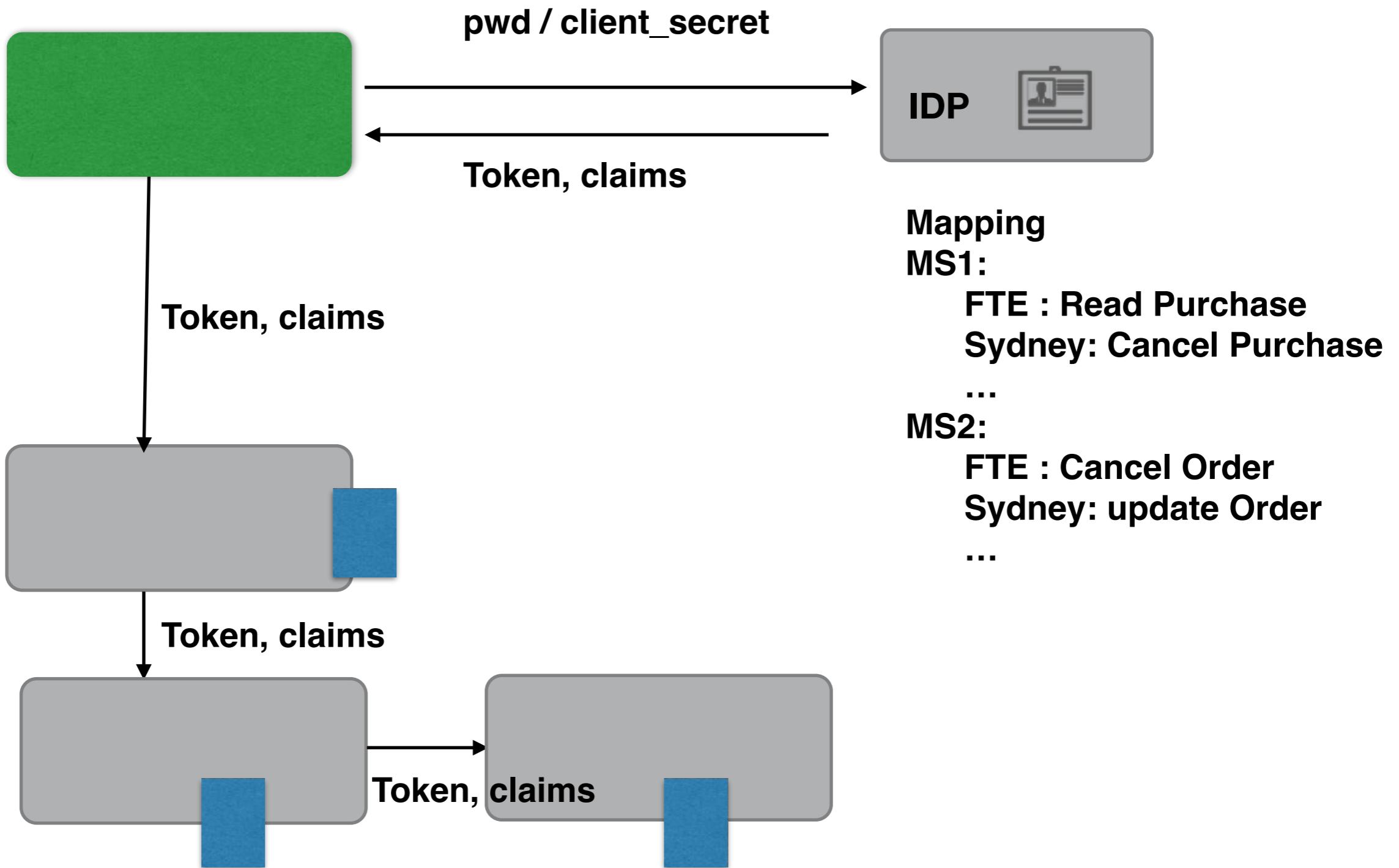
XSS, sql injection, one click,

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%

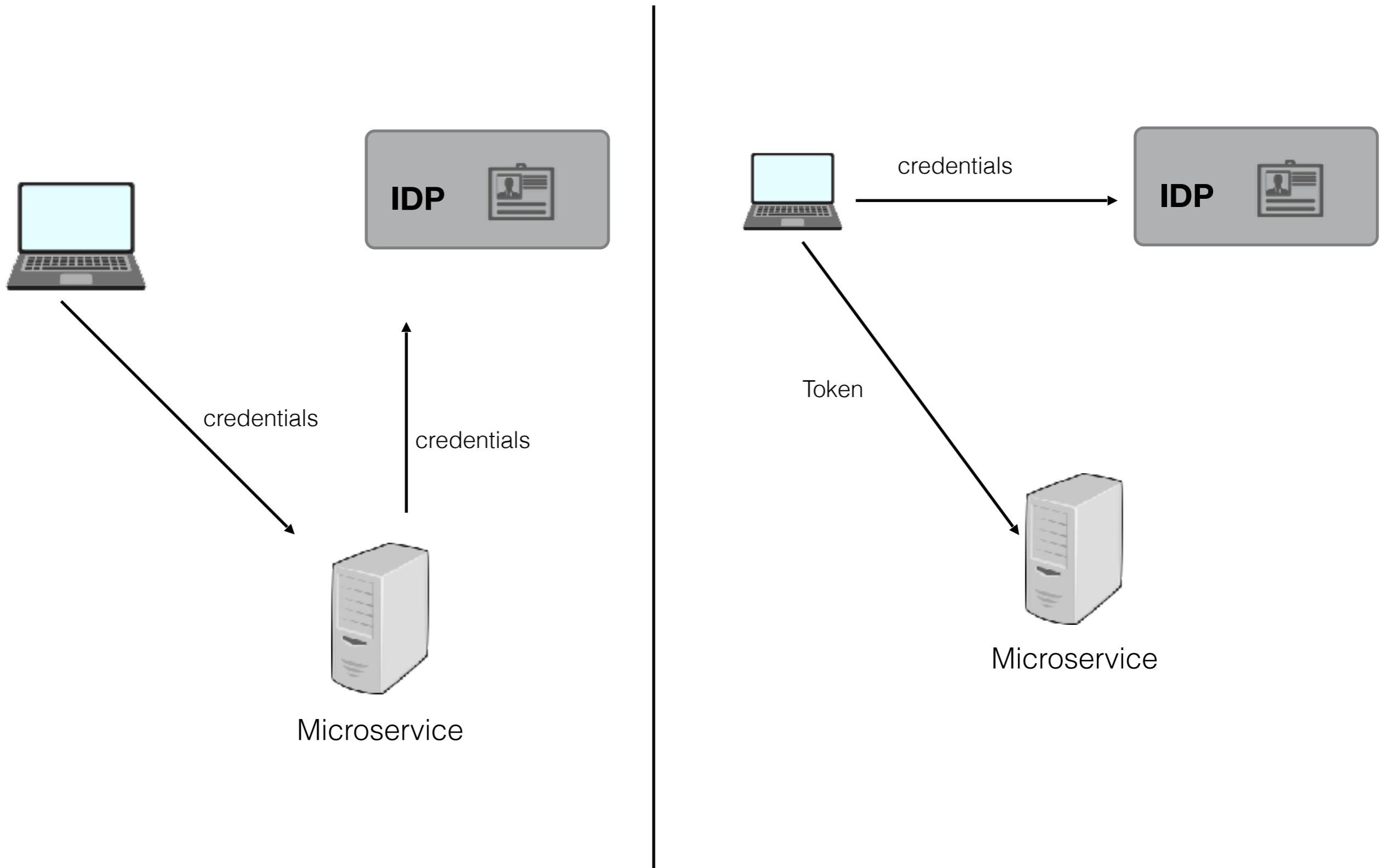
- Authentication - OAuth2 + OpenID Connect + JWT token
- Authorization - RBAC, Claims
- Audit - Event Sourcing
- Data In Transit - https, was
- Data In Rest - Hash, Sym, Asym, cloud storage (*)
- Input Validation - WAF, library (*)
- Exception Handling - fwk
- Session Management- fwk
- Key Management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

DevSecops - 50%

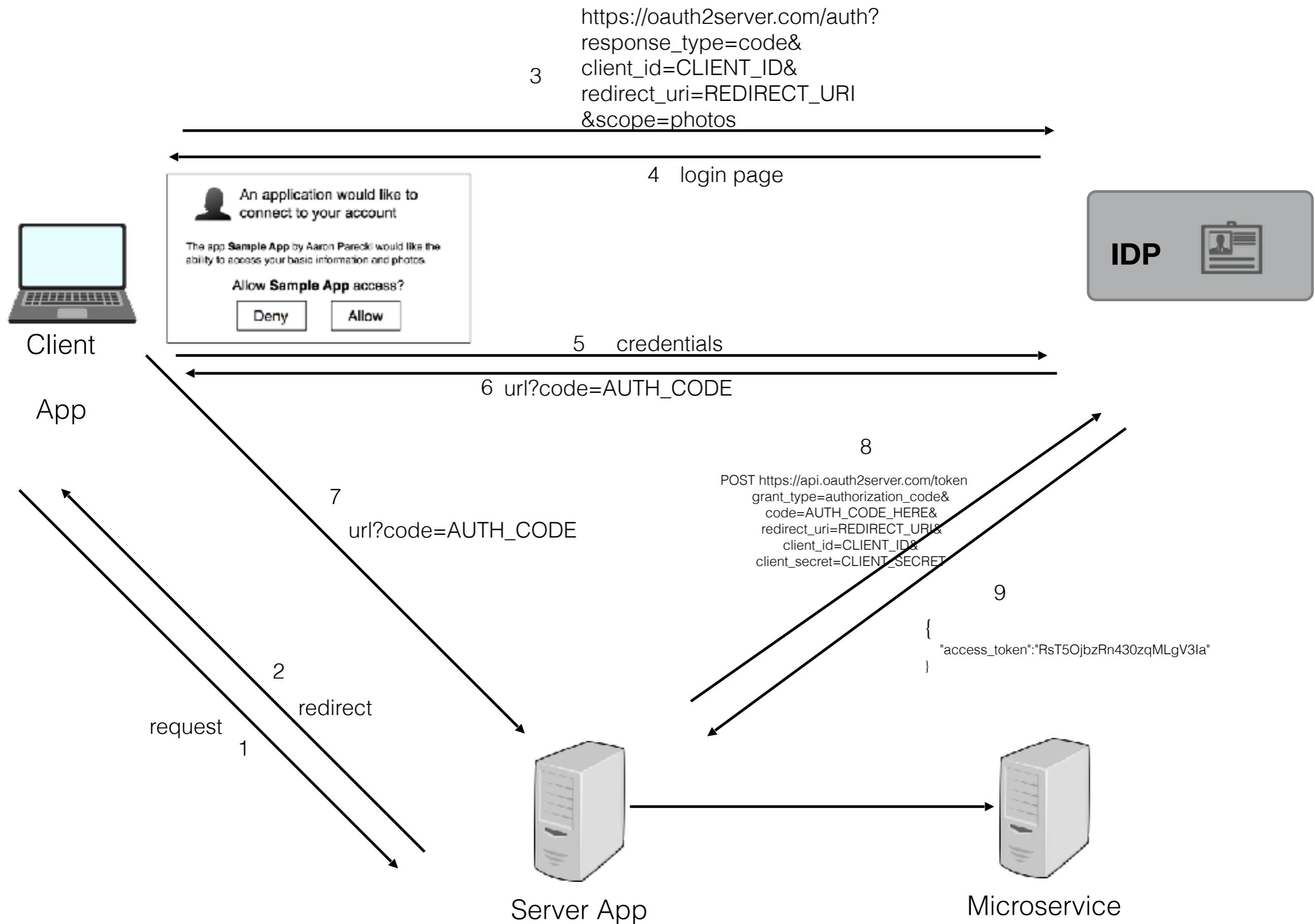
- L4 Firewall (TCP)
- L7 Firewall (HTTP) - WAF
- vnet, subnet
- Route table
- Scanning tools
- Monitoring & Alerts

- 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



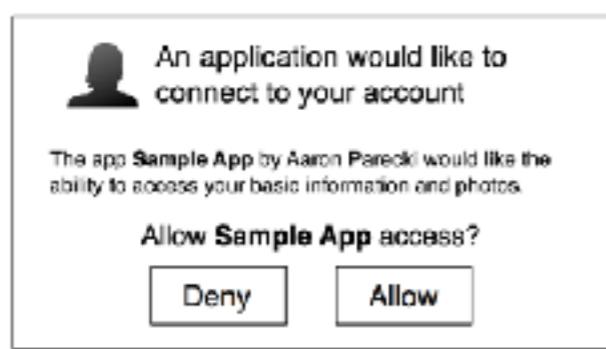
Implicit

[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

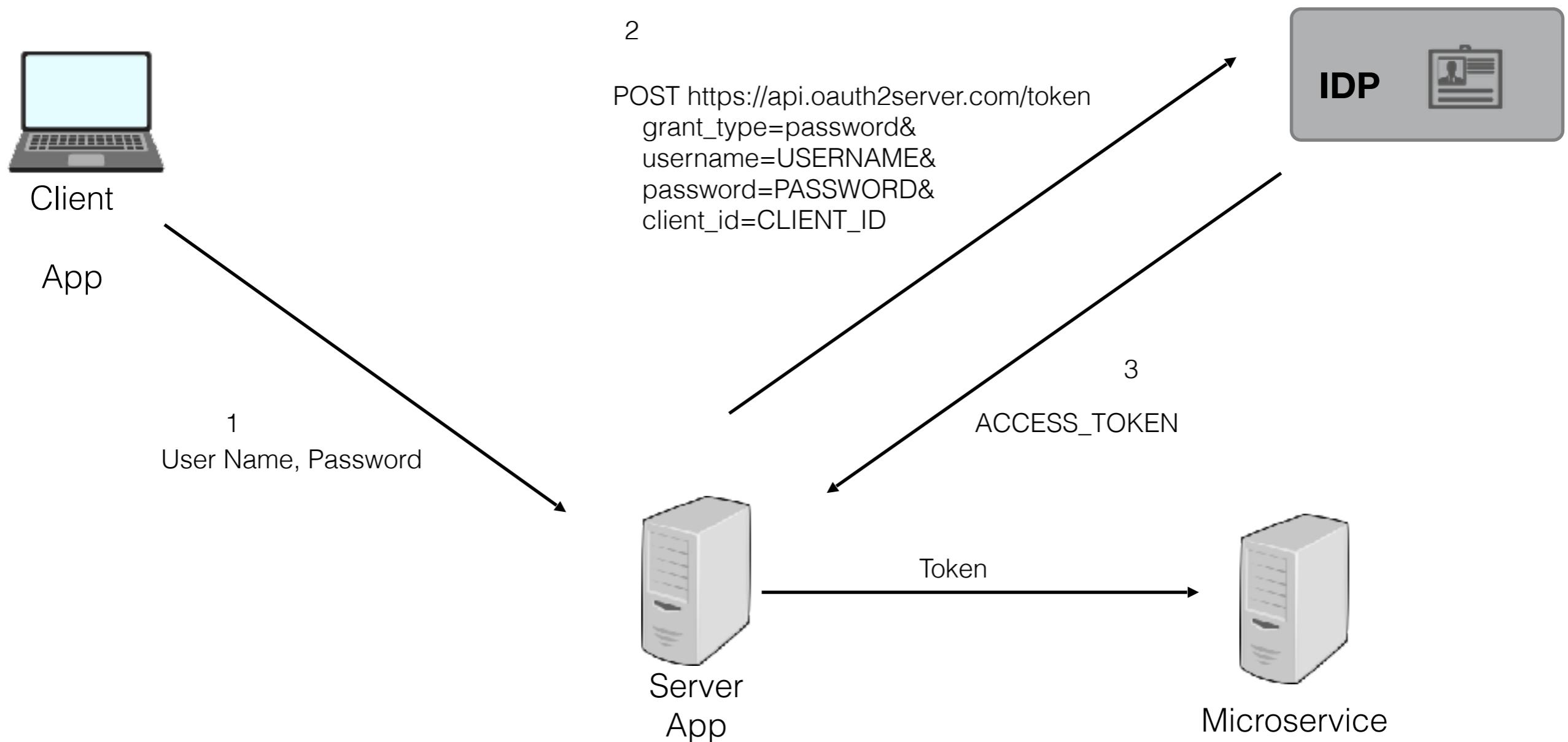
4 https://oauth2client.com/cb#token=ACCESS_TOKEN

5 url?token

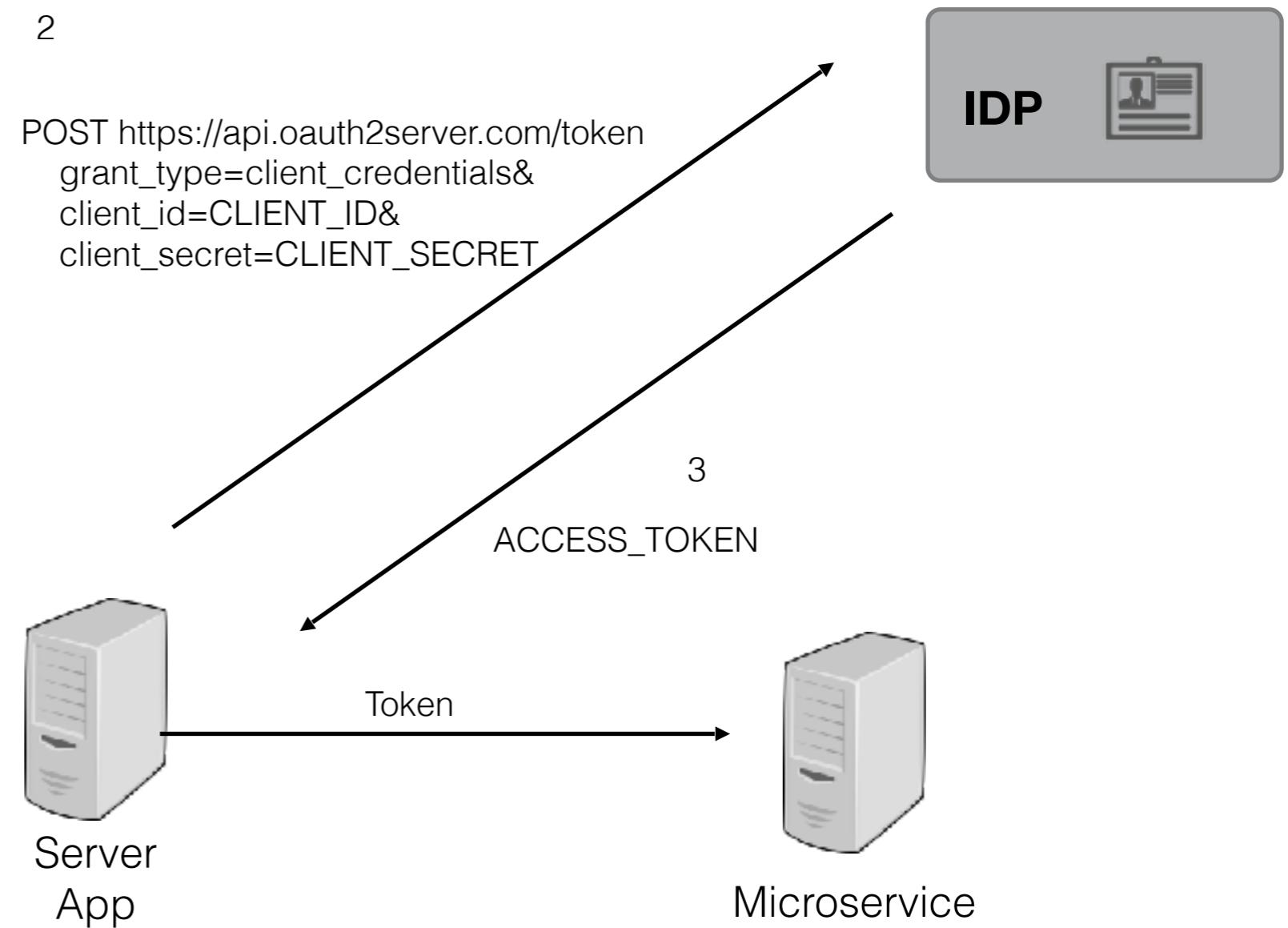


Microservice

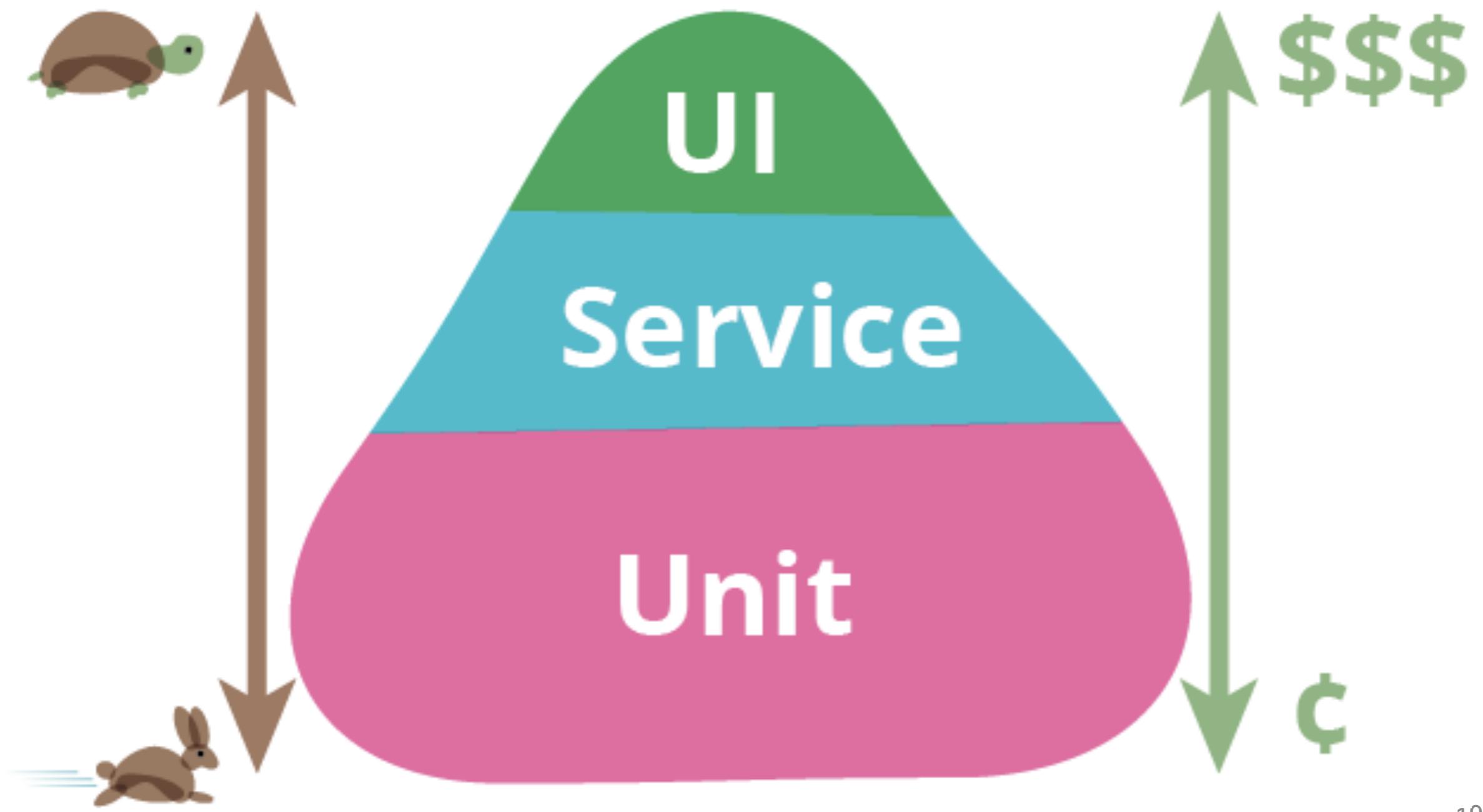
Password

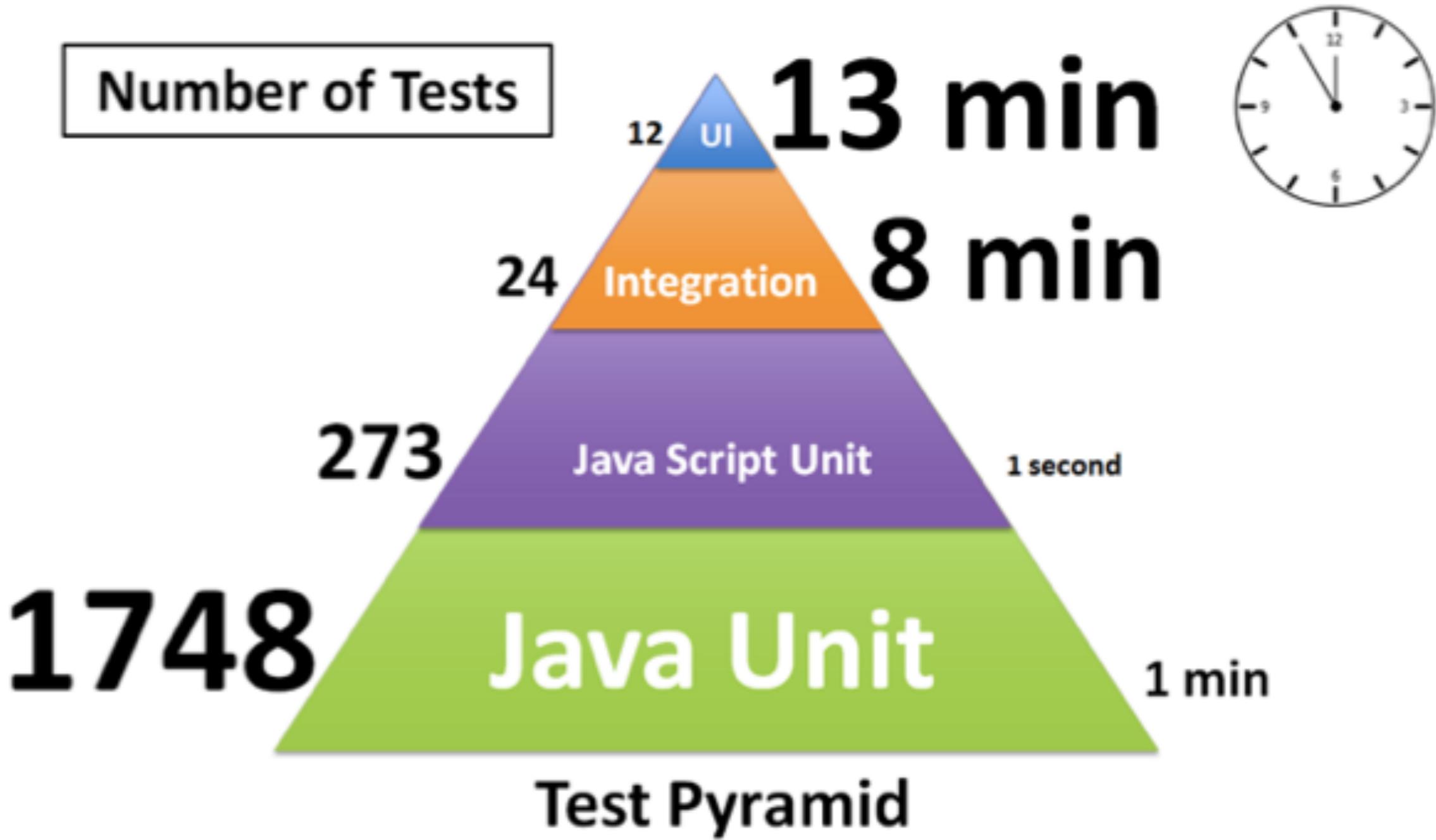


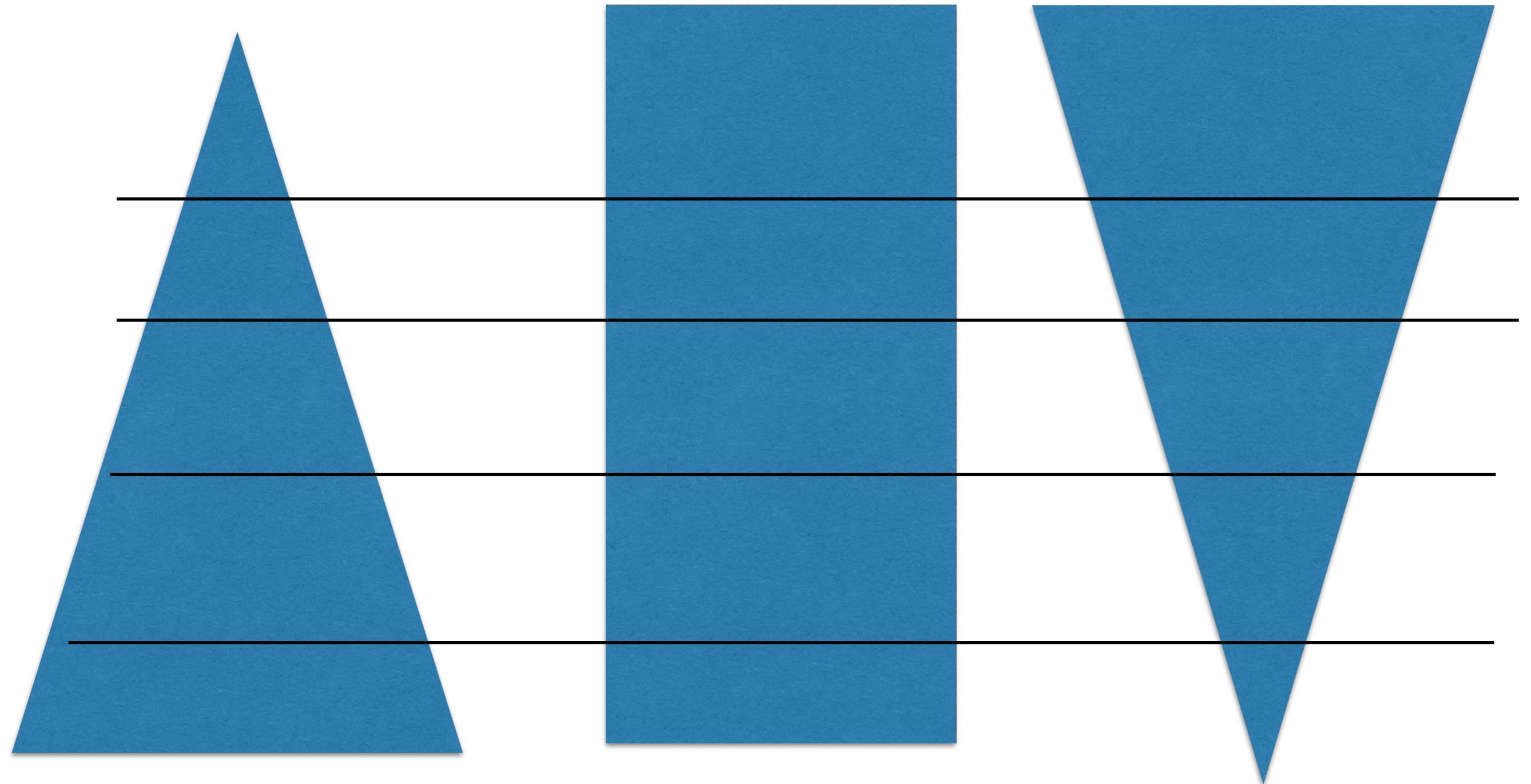
Clients Credential

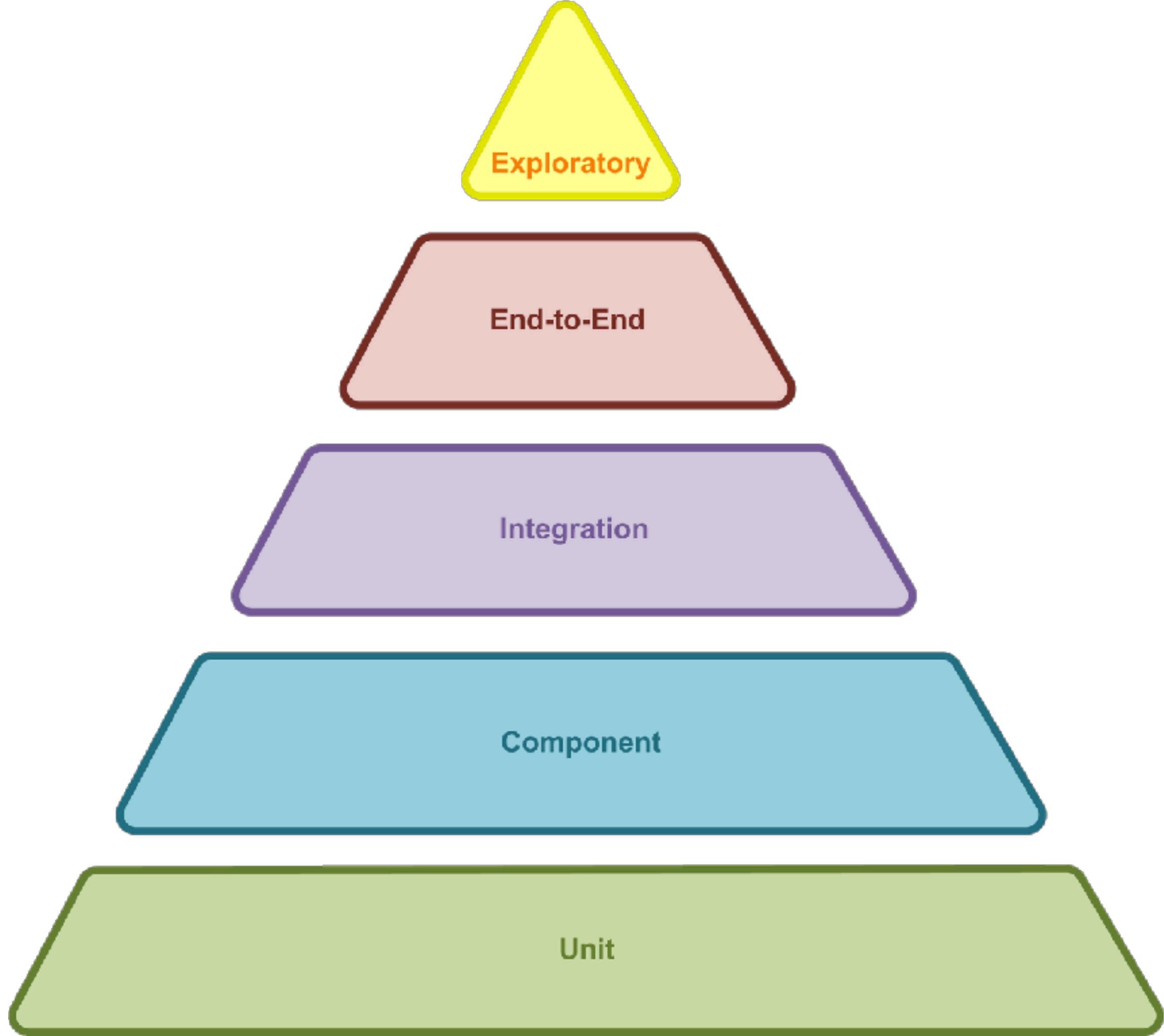


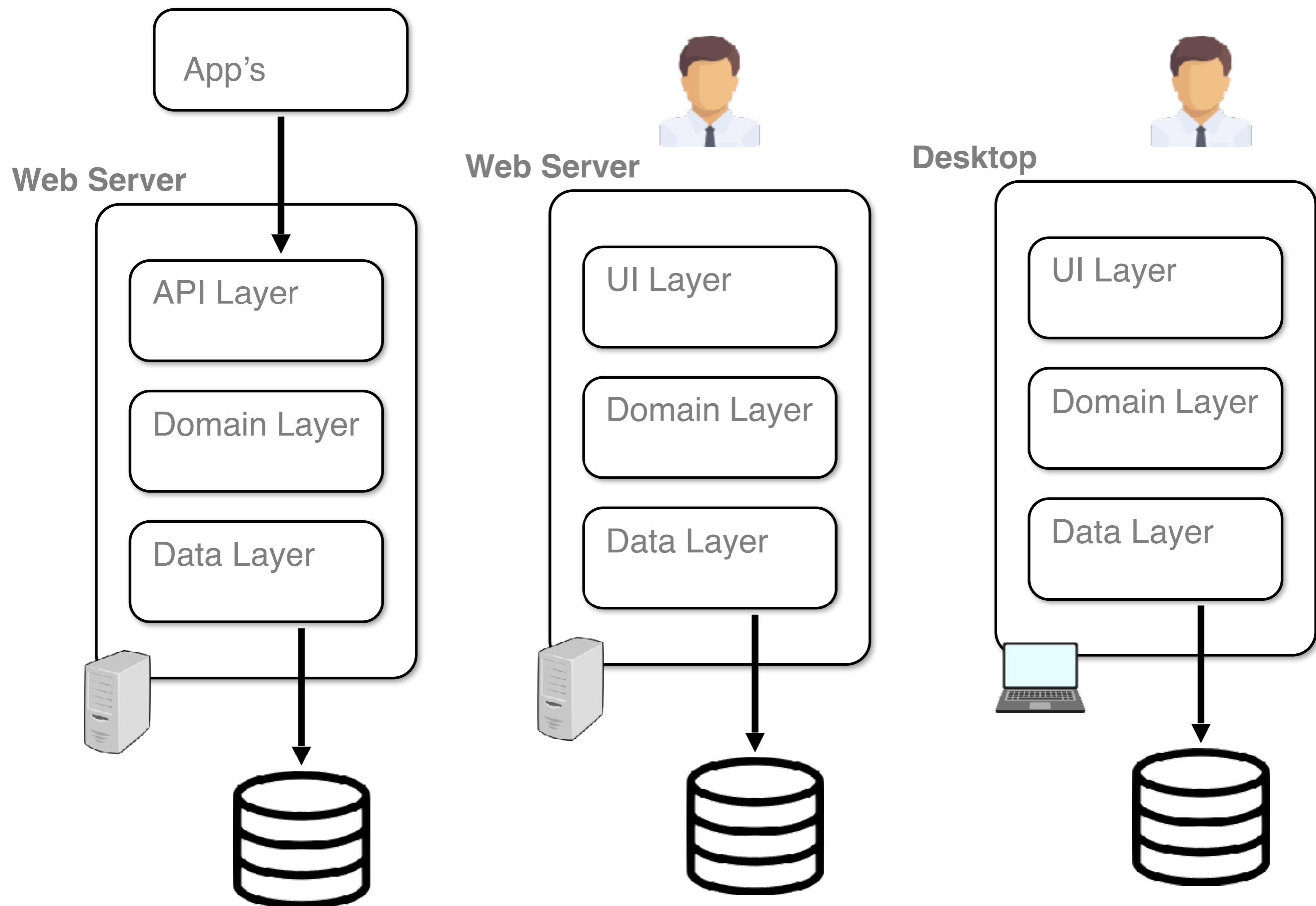
Test Pyramid

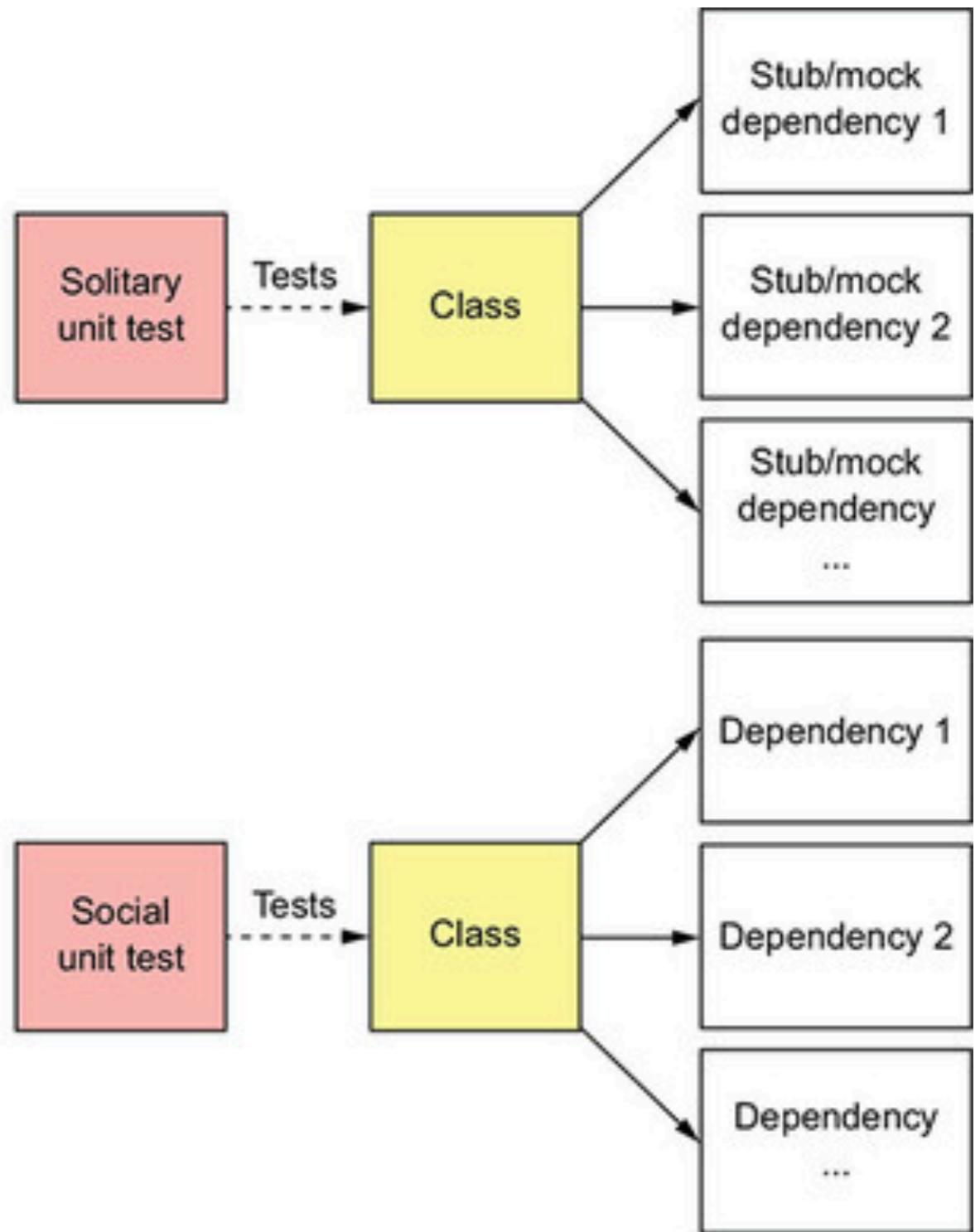
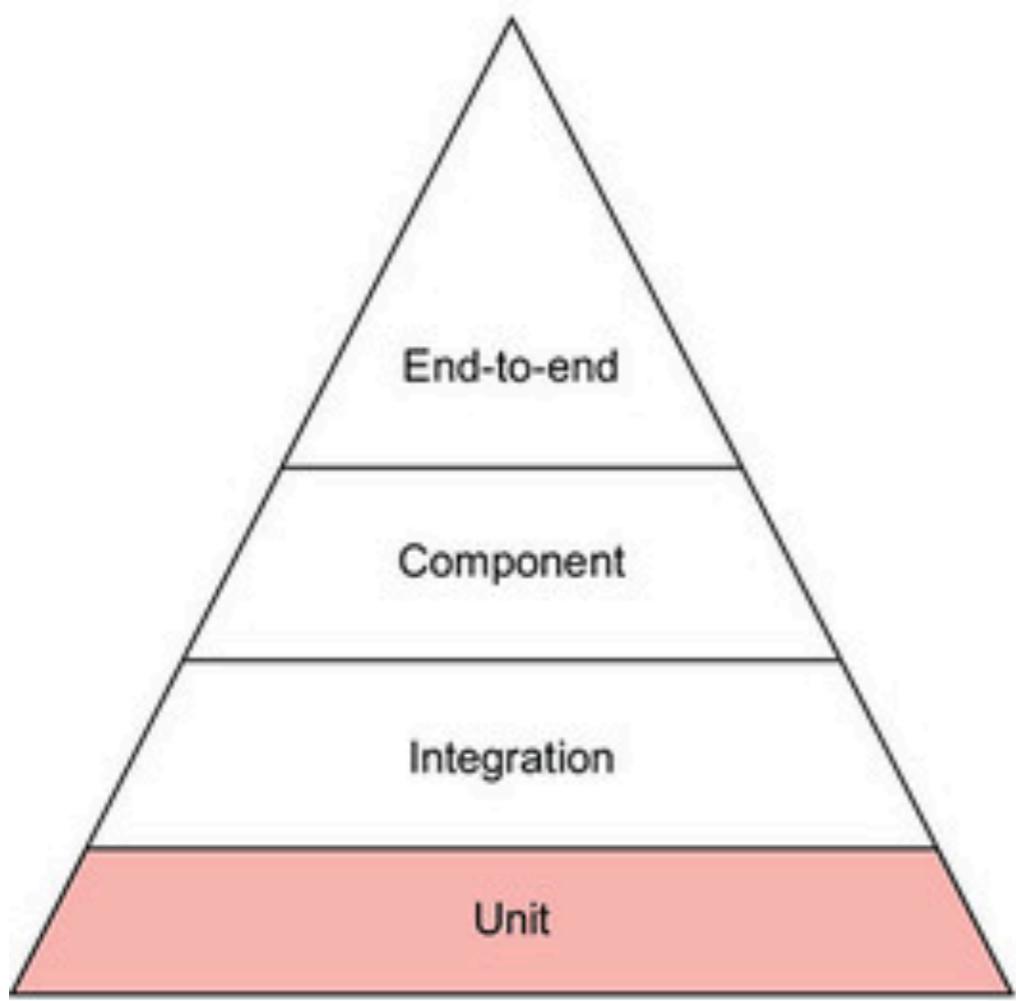


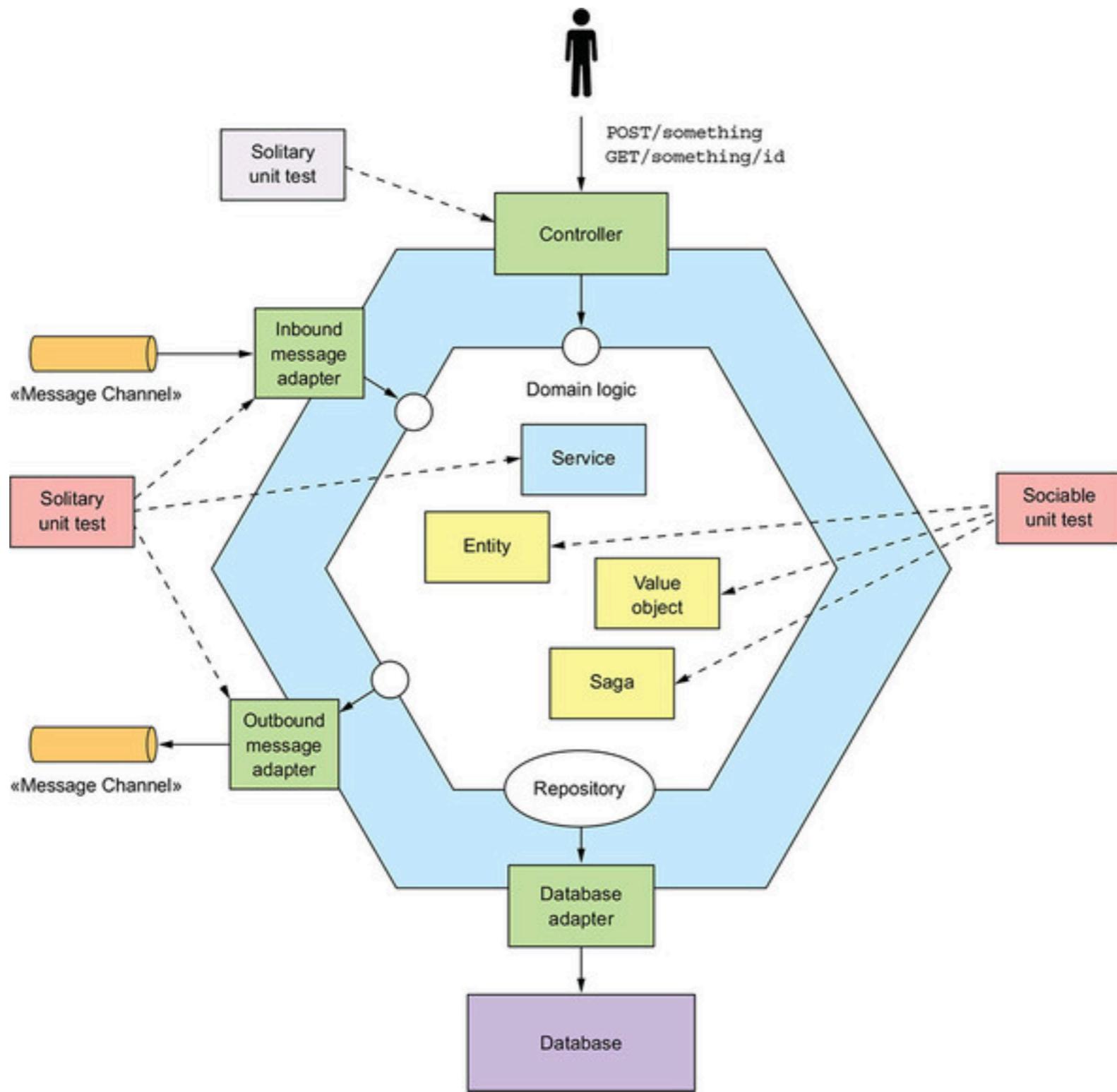








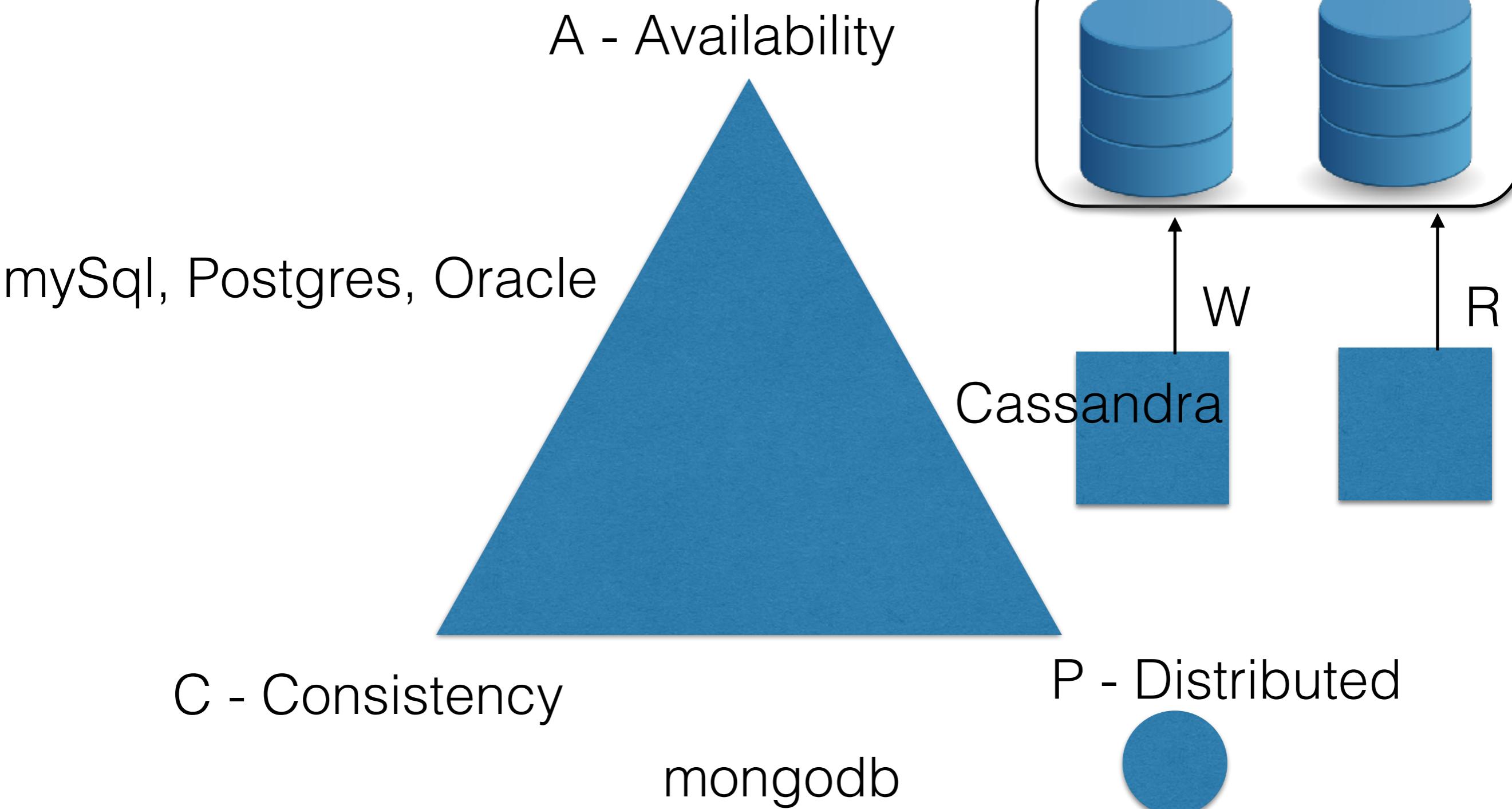




Patterns

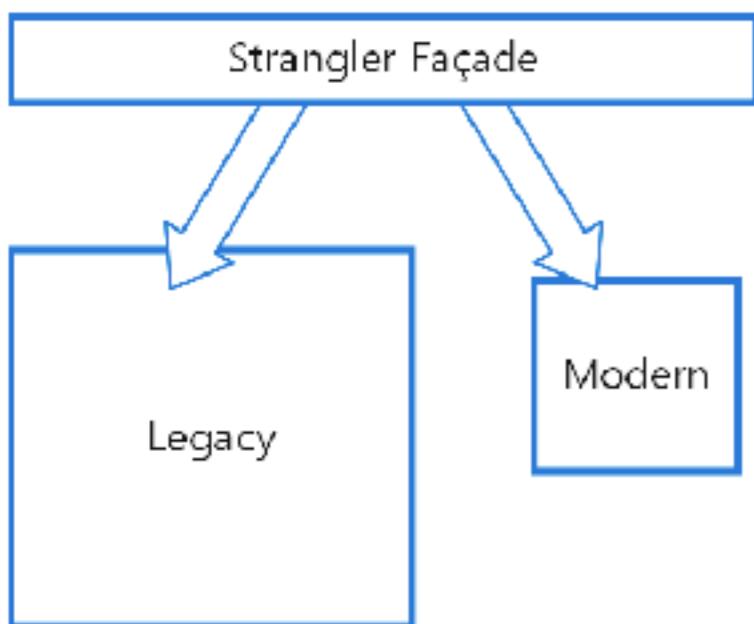


CAP Theorem

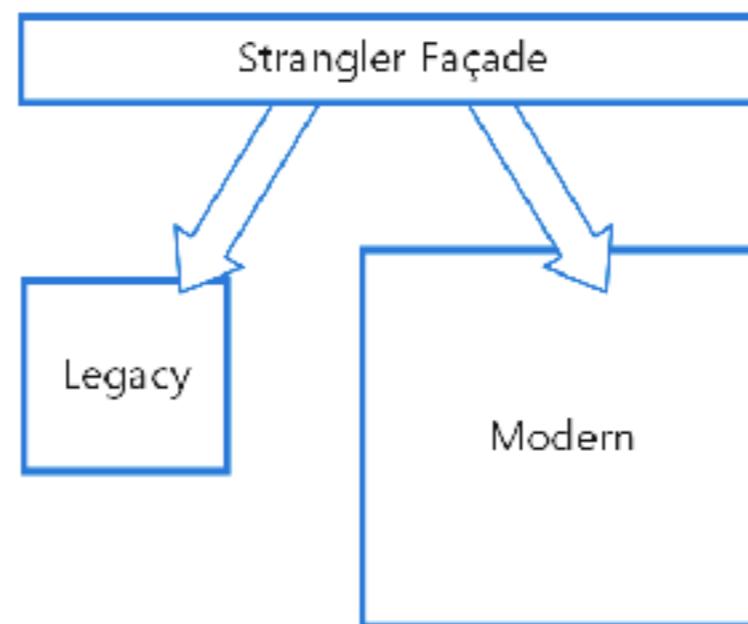


Strangler pattern

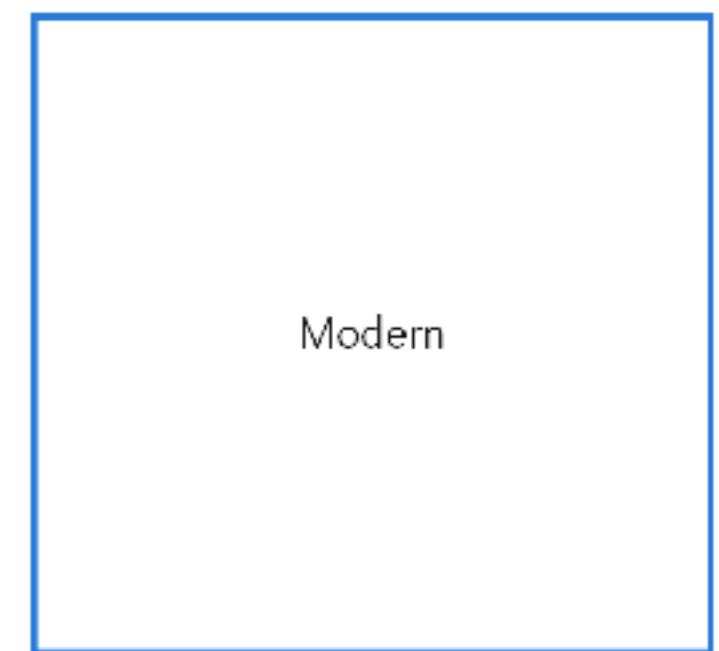
Early migration



Later migration

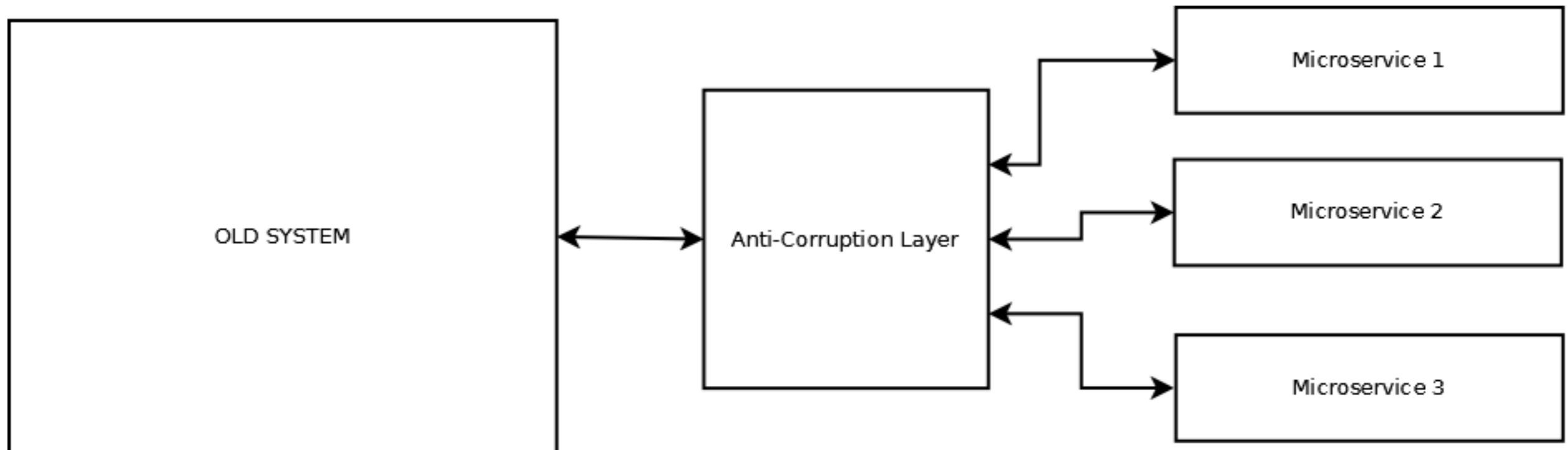


Migration complete



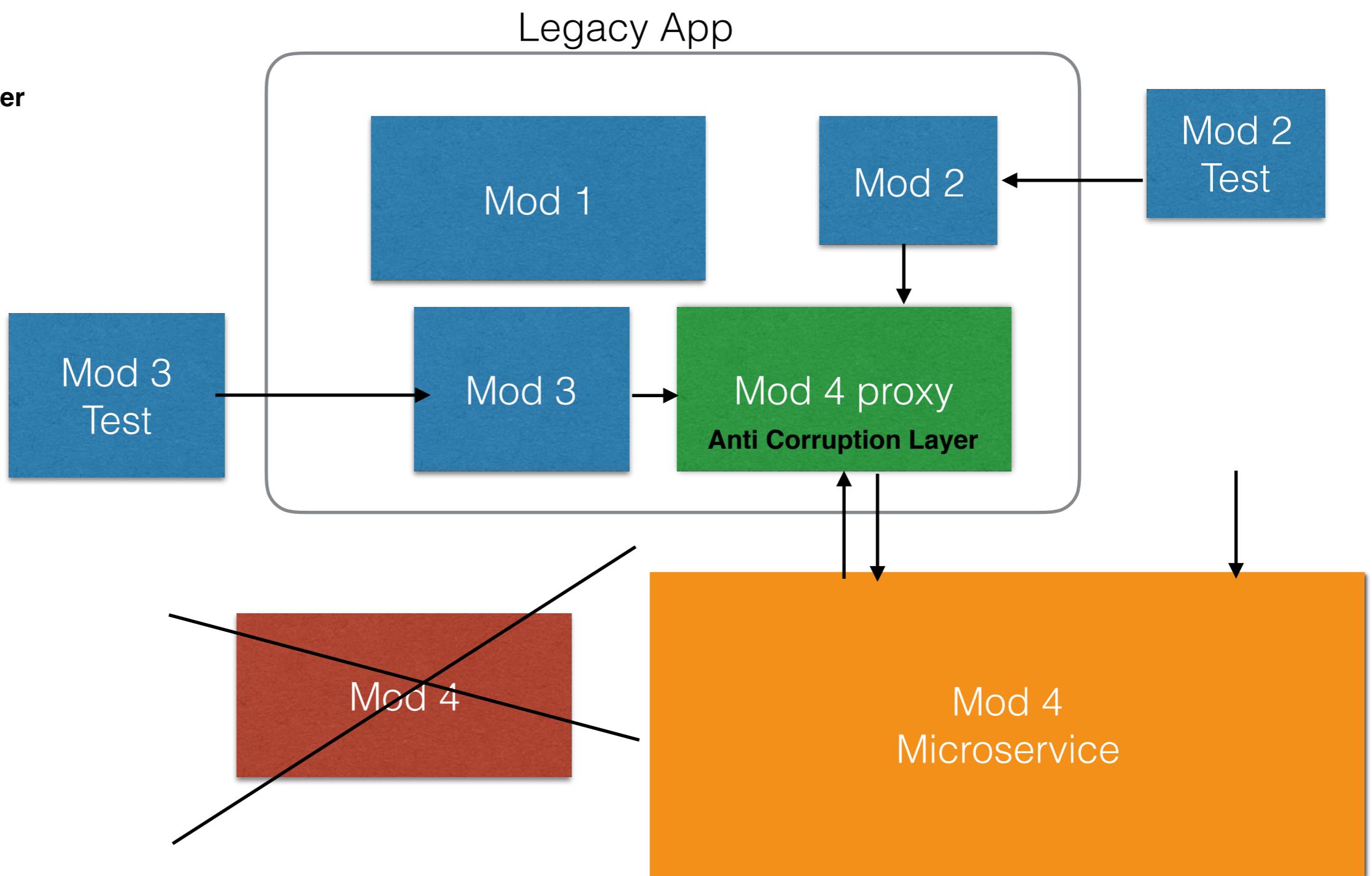
- Create a façade that intercepts requests going to the backend legacy system. Over time, as features are migrated to the new system, the legacy system is eventually "strangled" and is no longer necessary.

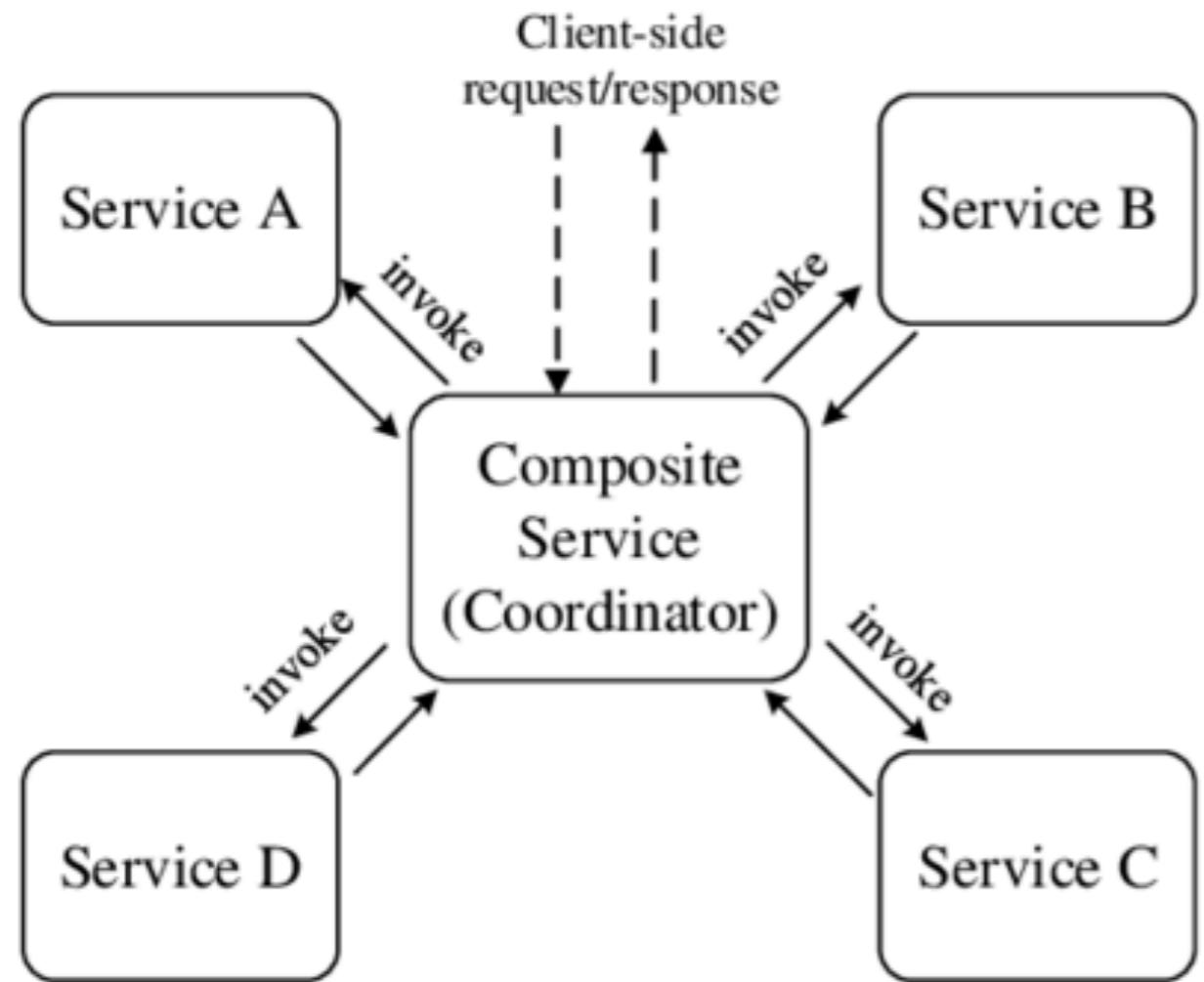
Anti-Corruption Layer



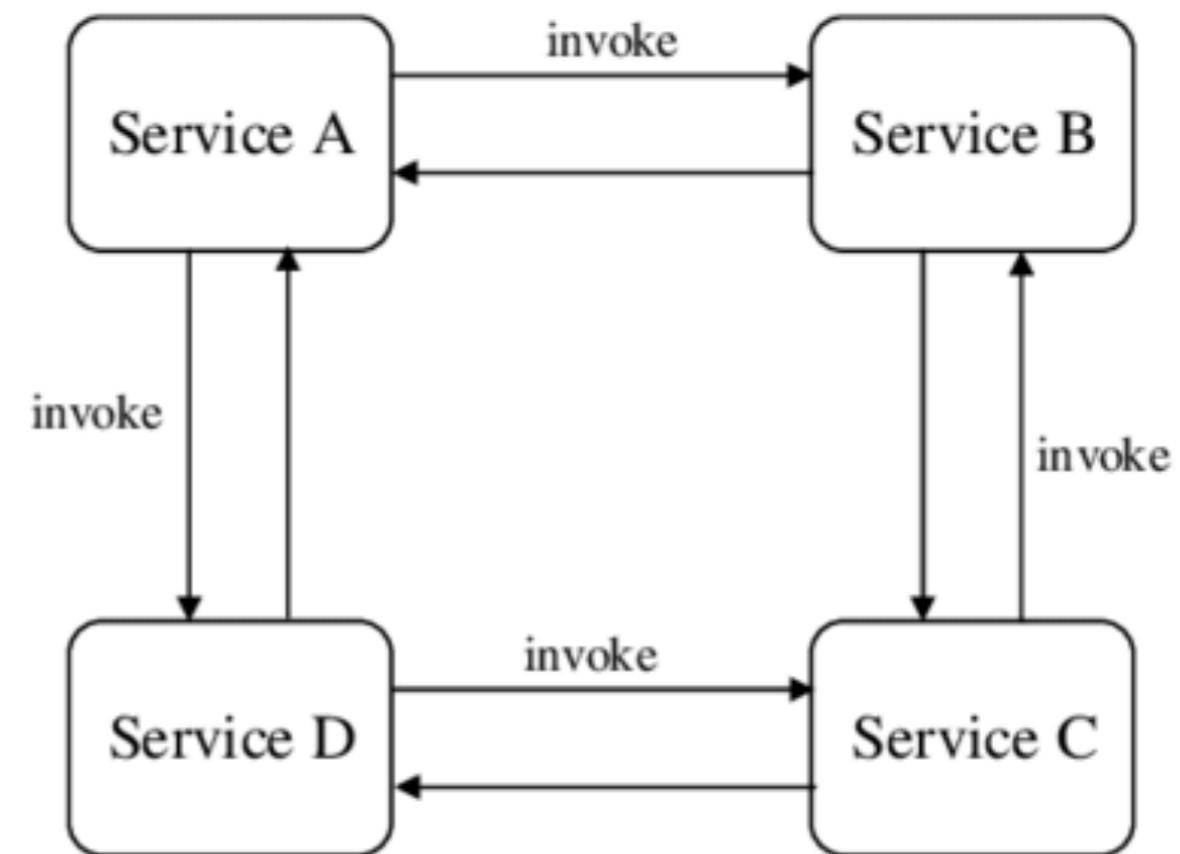
- When you create a new microservice, you notice that some business or technical assumptions or flow should be remodeled. instead of a Big Rewrite ACL maps one domain onto another so that services that use second domain do not have to be "corrupted" by concepts from the first.
- Such modifications may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

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





(a) Web Service Orchestration



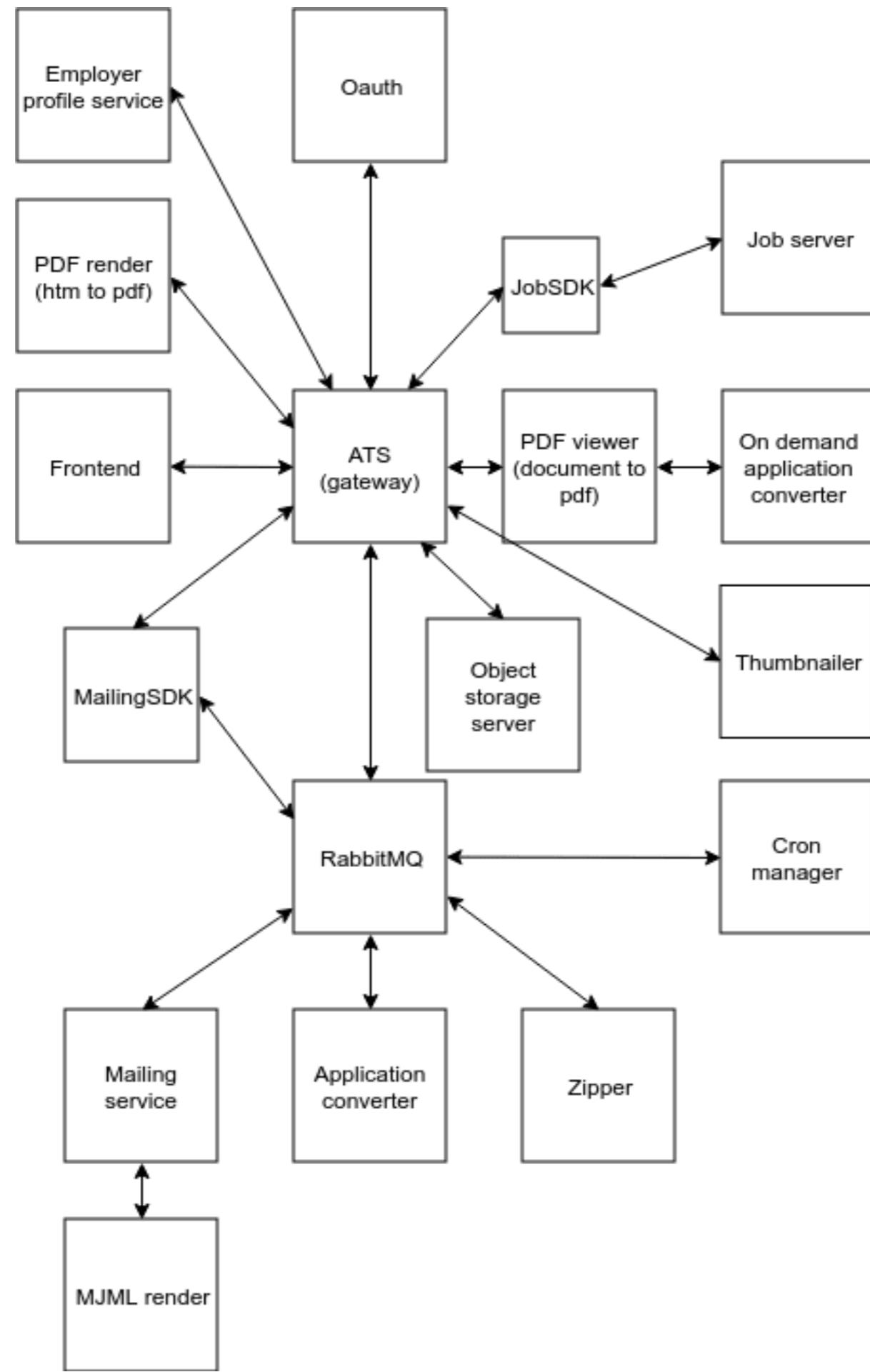
(b) Web Service Choreography

Case Study

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.

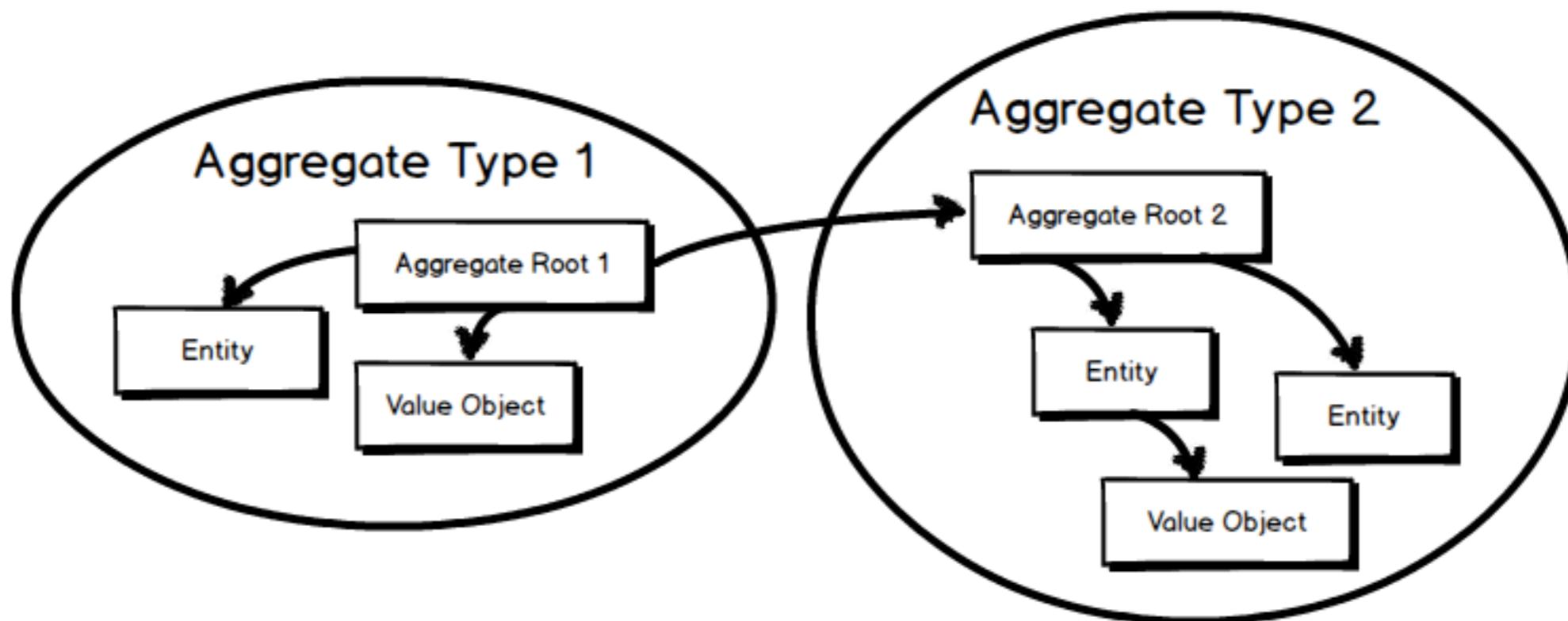
Patterns

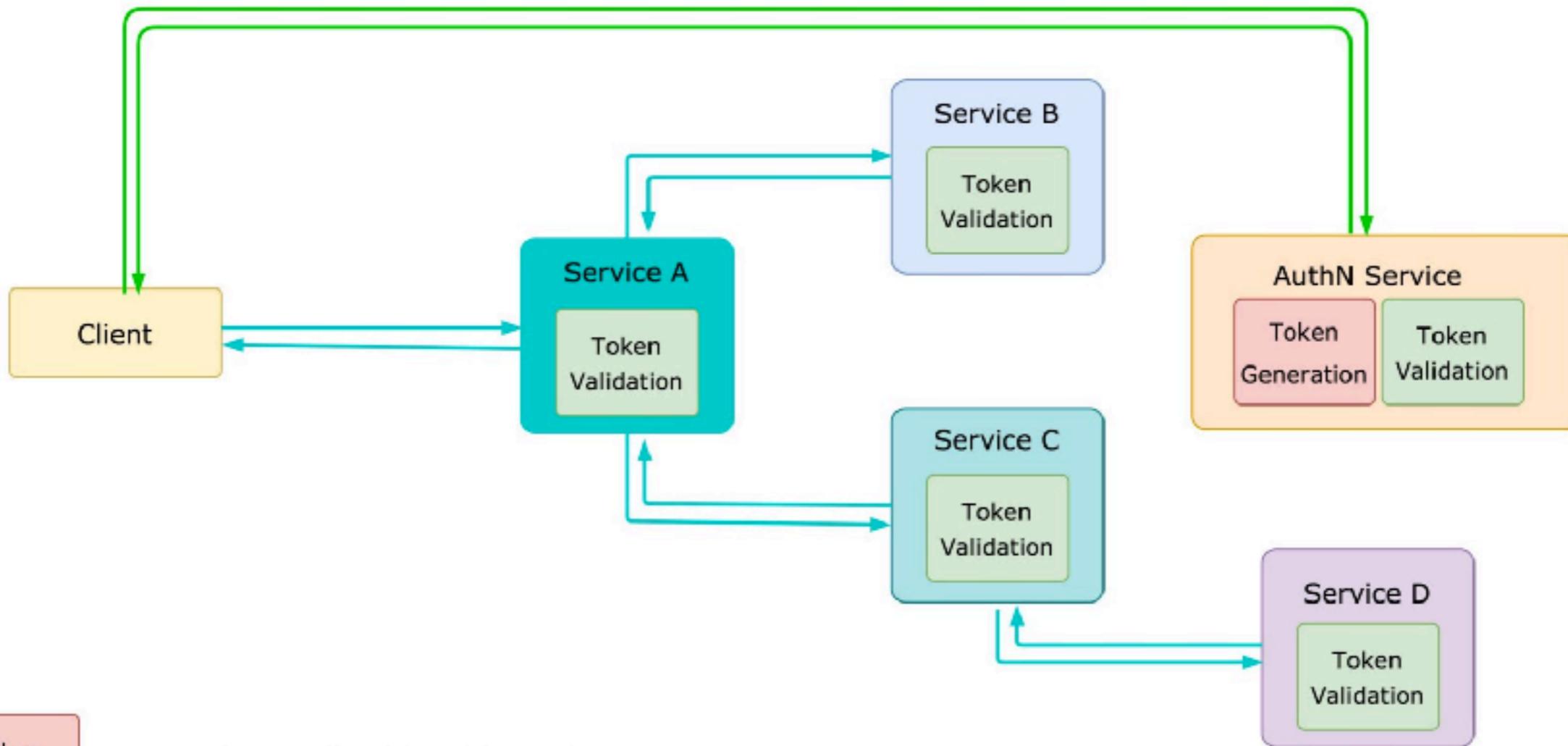
- Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.

Bounded Context



Aggregate Root





Token
Validation

Token Generation Using Private Key

Token
Validation

Token Validation Using Public Key

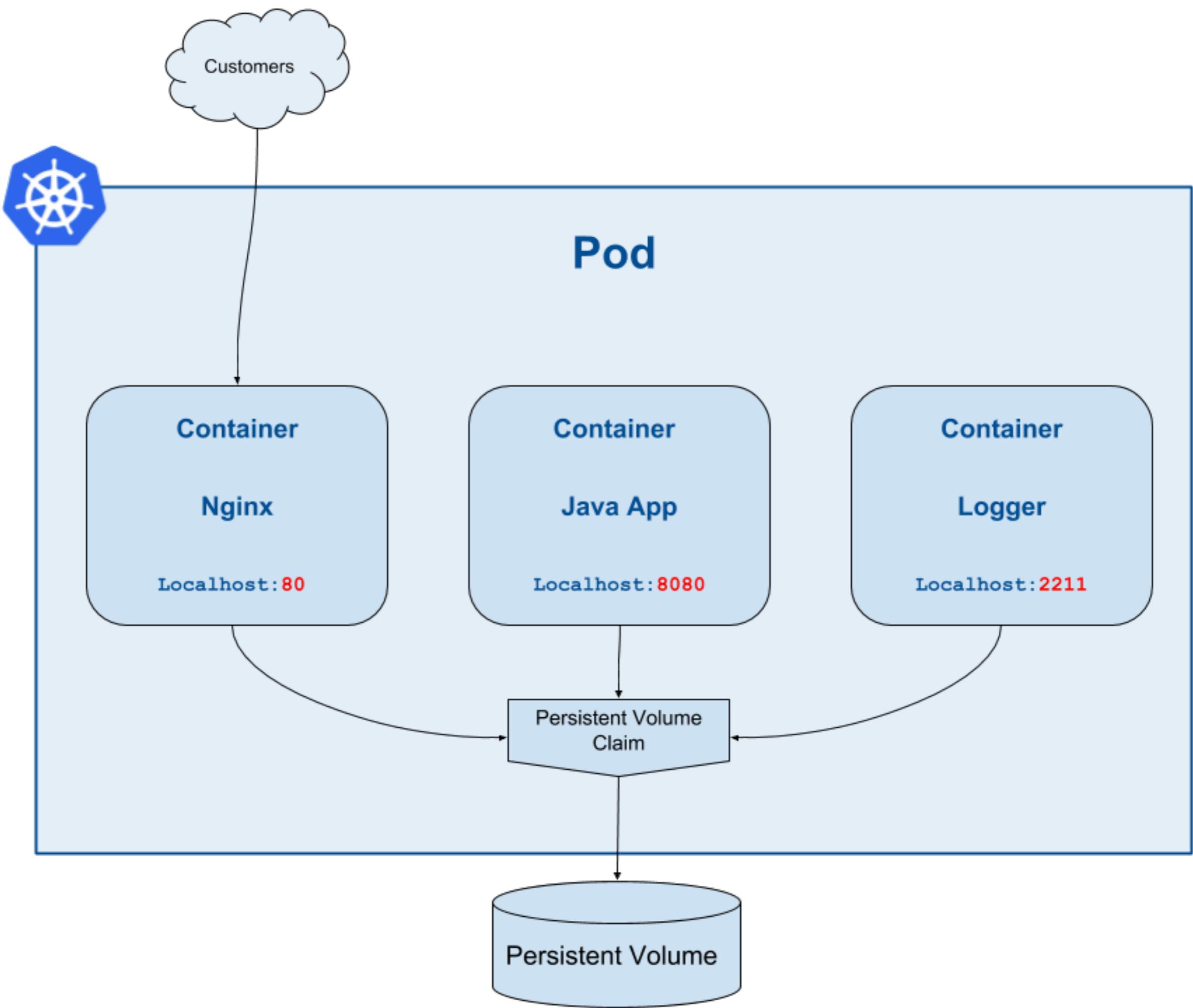
Any service Can Validate Using Public Key

Private Key in One Secure Place, Really Insecure

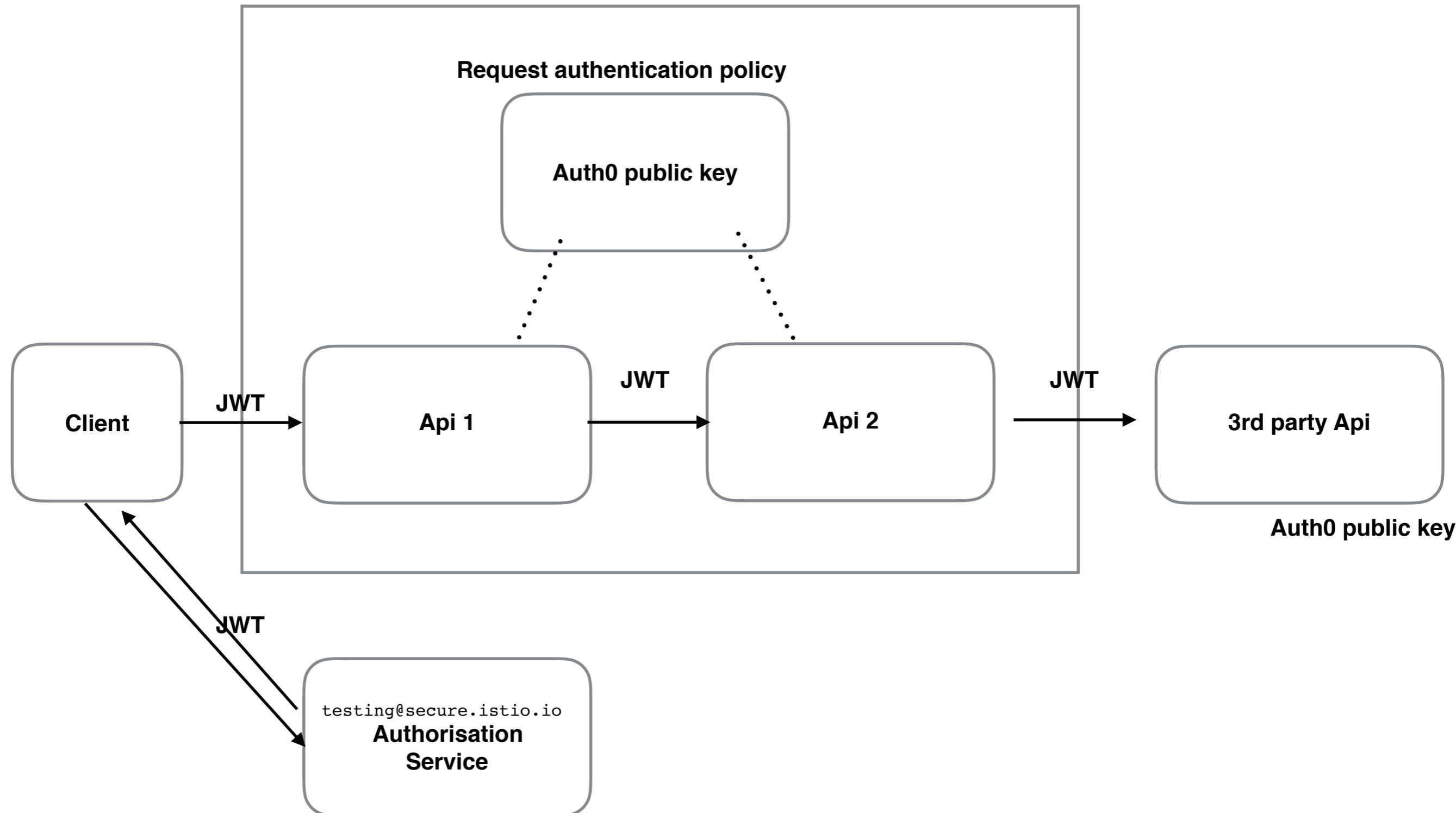
→ Traffic for Authentication & Generating JWT Token

→ 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.
-

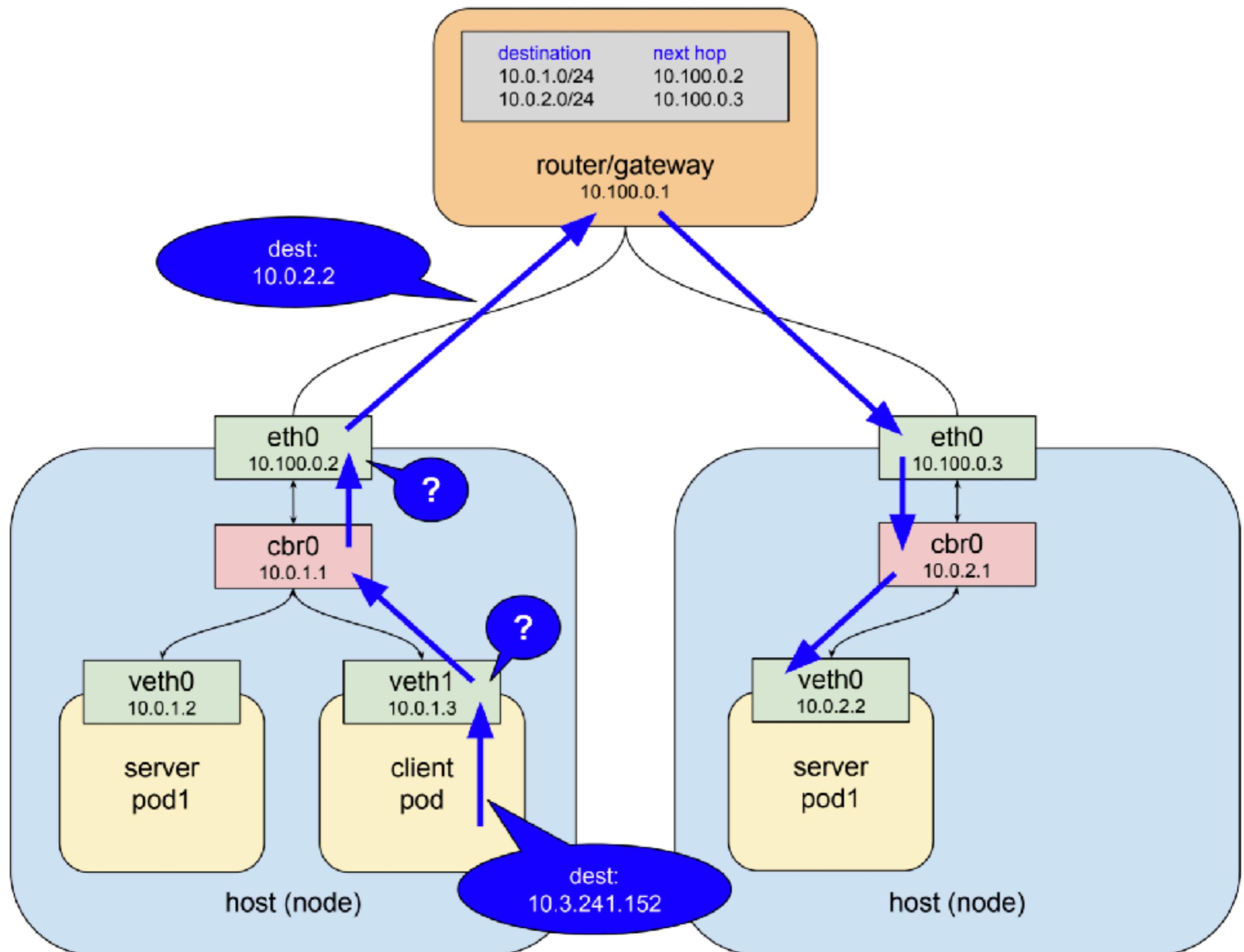
- NodePort is a convenient tool for testing in your local Kubernetes cluster, but it's not suitable for production because of these limitations.
- A single node is a single point of failure for the system. Once the node is down, clients can't access the cluster any more.
- A service can be declared as 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:

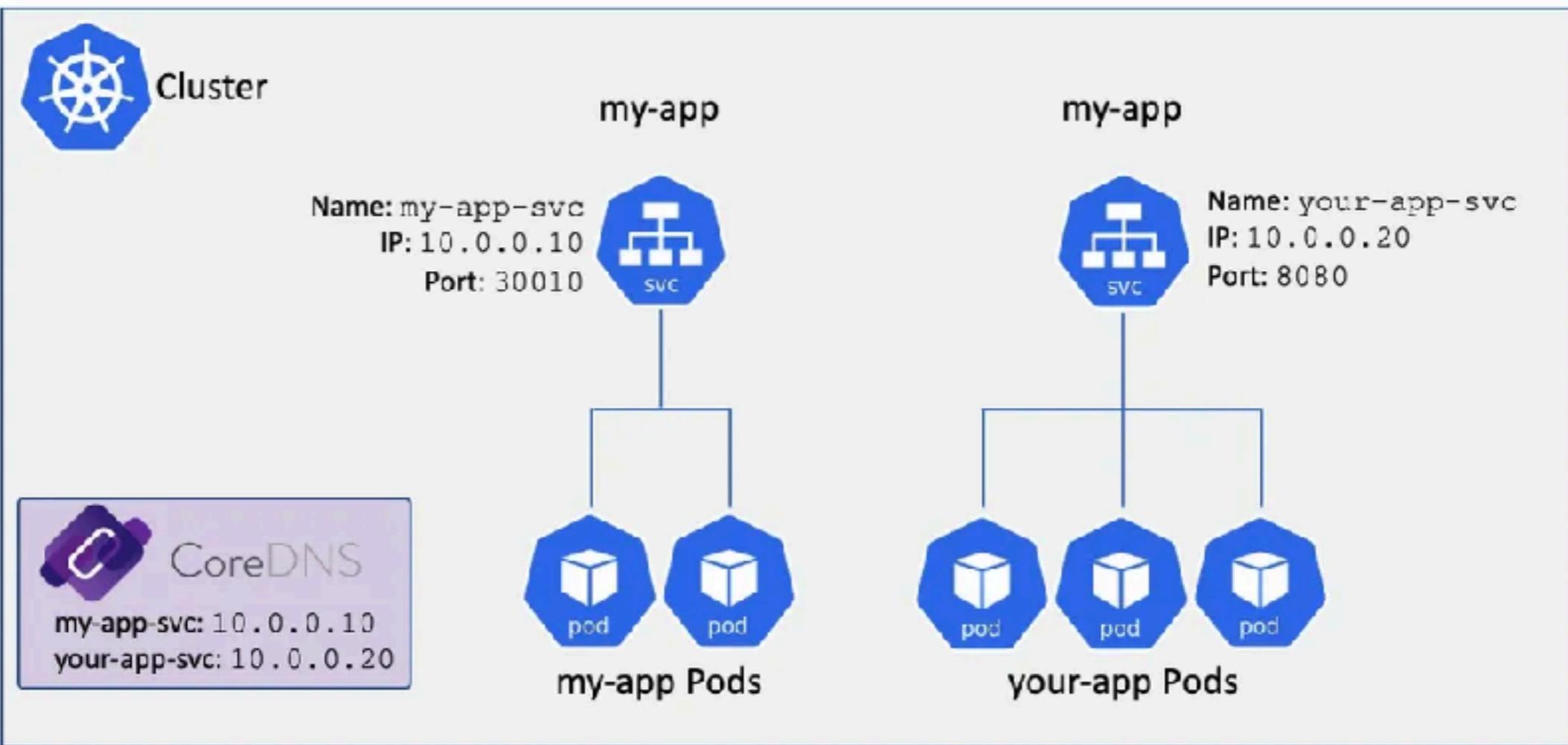
- Needs more public IPs, which normally are limited resources.
- Introduces coupling between the client and the server, making it hard to adjust your backend services when business requirements change.

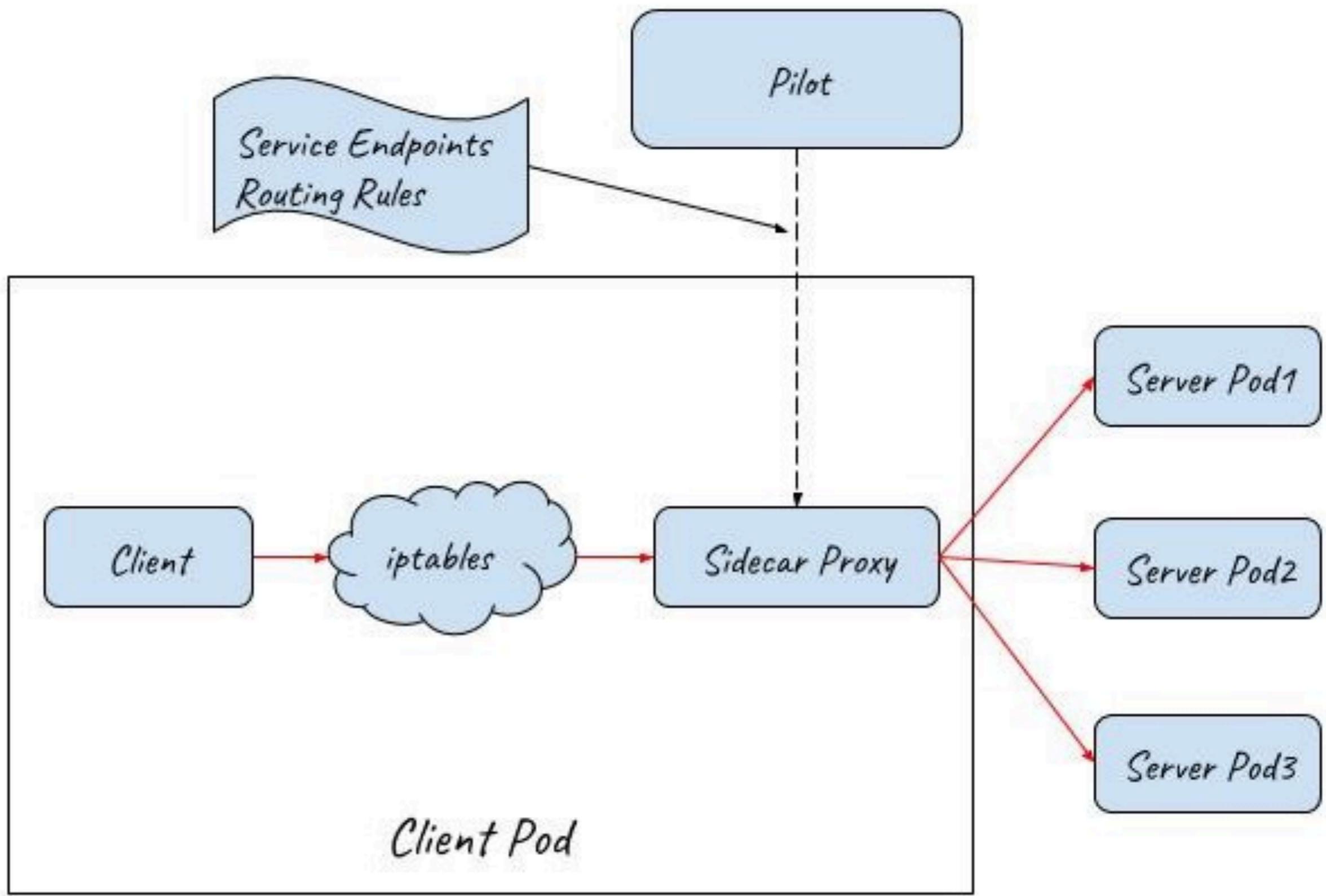
Kubernetes Ingress resource is used to declare an OSI layer 7 load balancer, which can understand HTTP protocol and dispatch inbound traffic based on the HTTP URL or Host.

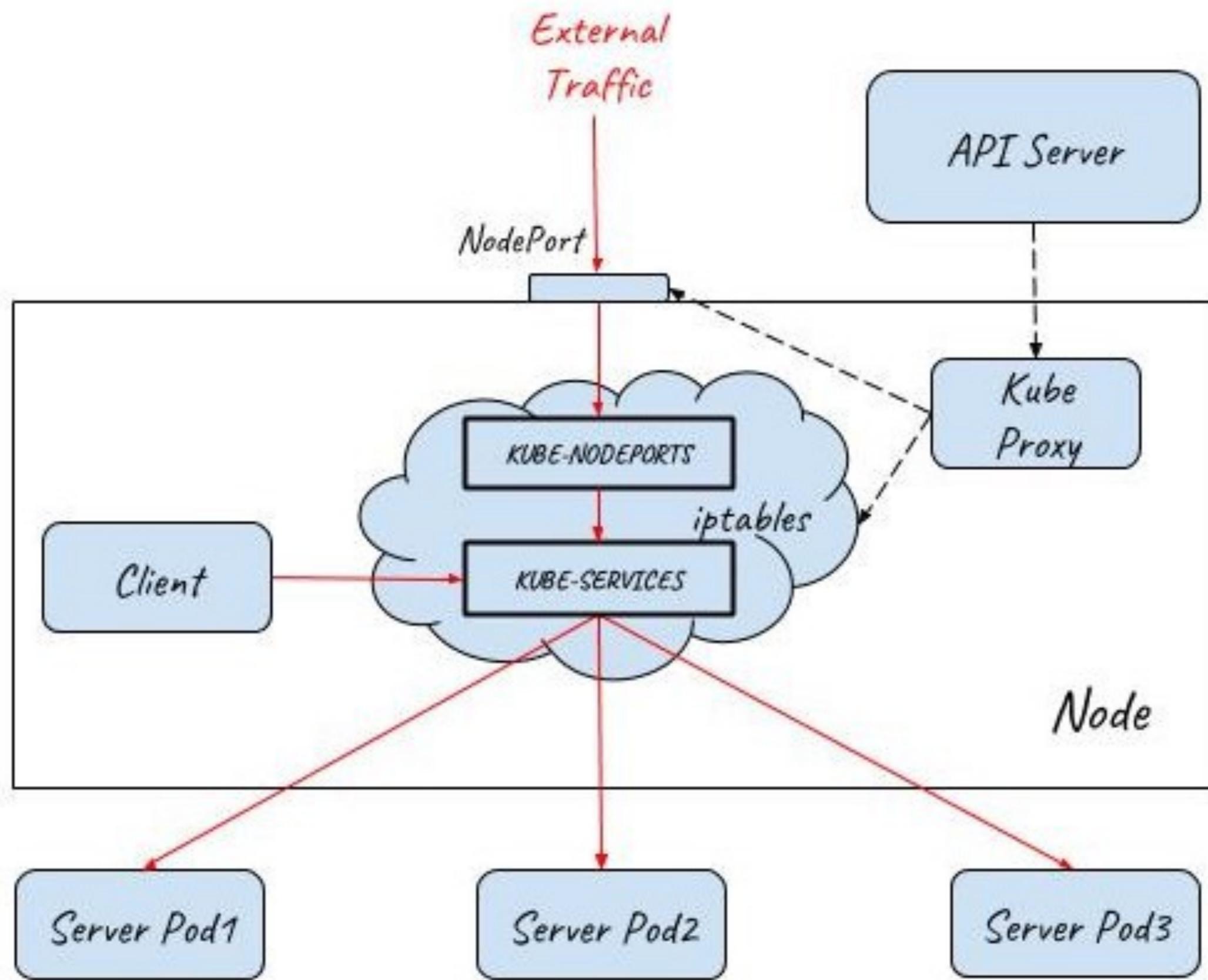
- Ingress controller itself is also deployed as Pods inside the cluster.
- Ingress controller provides a unified entrance for the HTTP services in a cluster, but it can't be accessed directly from outside because the ingress controller itself is also deployed as Pods inside the cluster.
- Ingress controller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

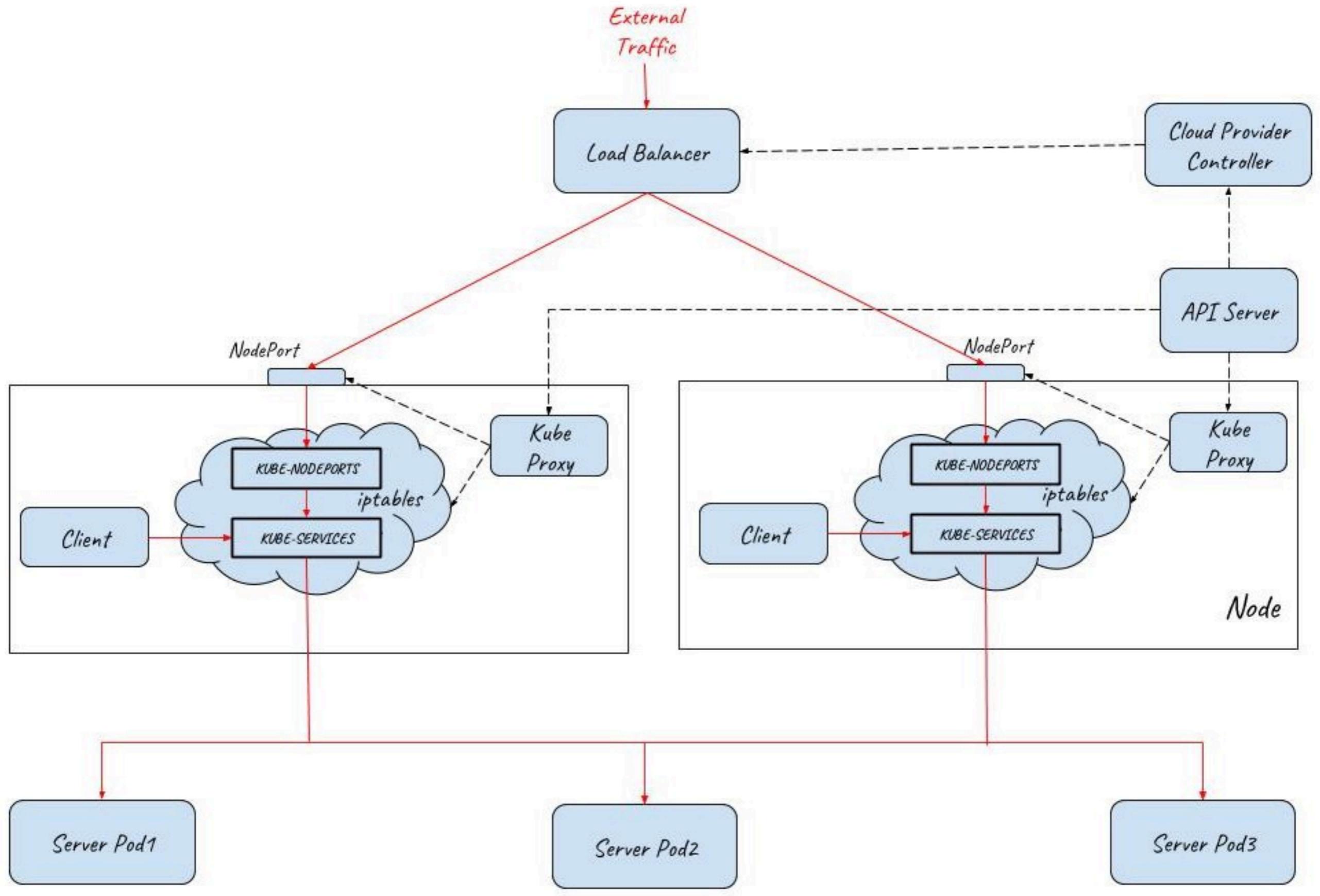


	OSI Layer	TCP/IP	Datagrams are called
Software	Layer 7 Application	HTTP, SMTP, IMAP, SNMP, POP3, FTP	Upper Layer Data
	Layer 6 Presentation	ASCII Characters, MPEG, SSL, TSL, Compression (Encryption & Decryption)	
	Layer 5 Session	NetBIOS, SAP, Handshaking connection	
	Layer 4 Transport	TCP, UDP	Segment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

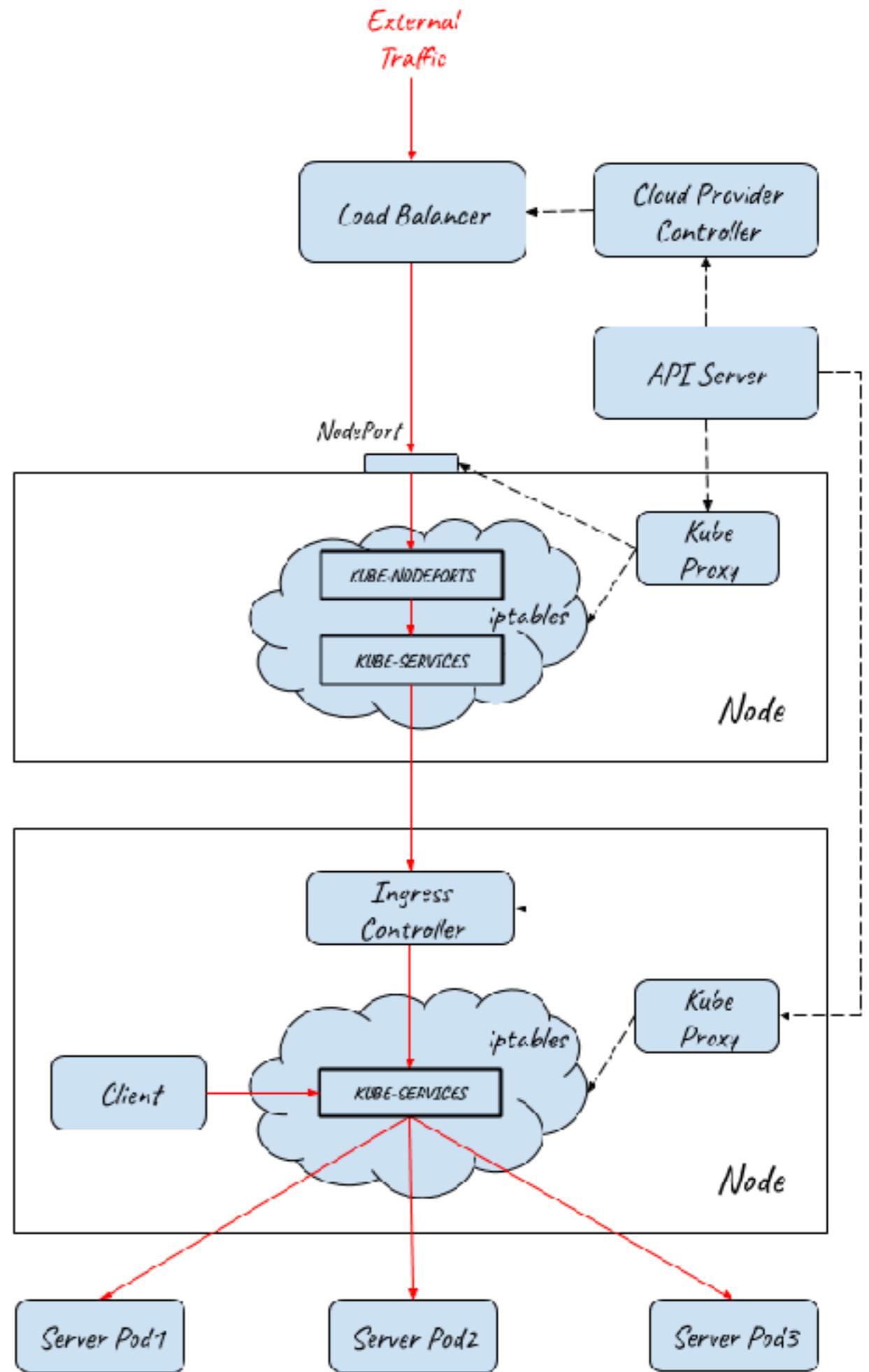








1. Internet/External traffic reaches the layer 4 load balancer.
2. Load balancer dispatches traffic to multiple NodePorts on the Kubernetes.
3. Traffic is captured by iptables and redirected to ingress controller Pods.
4. Ingress controller sends traffic to different Services according to ingress rules.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

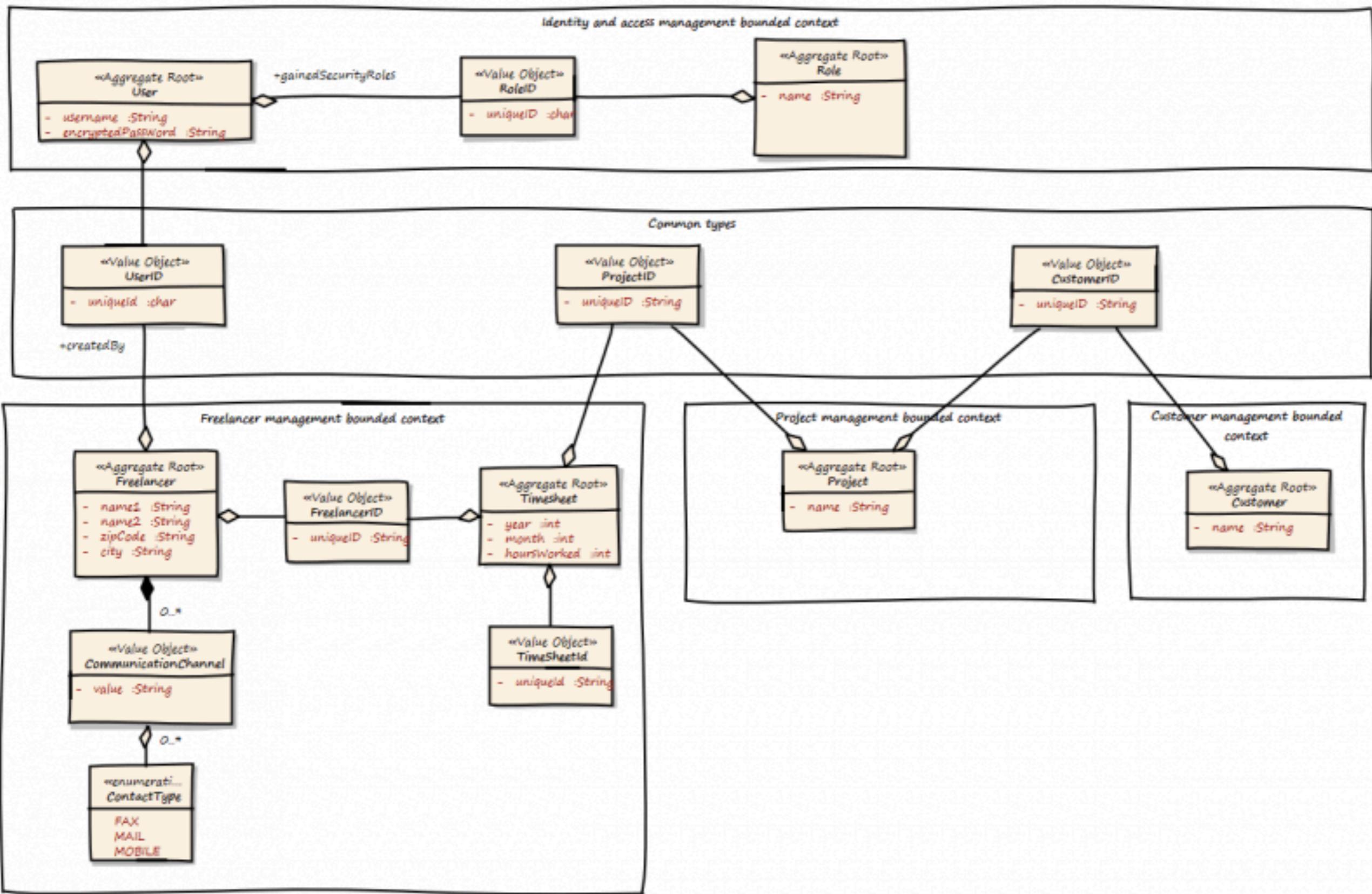
Body Leasing Domain

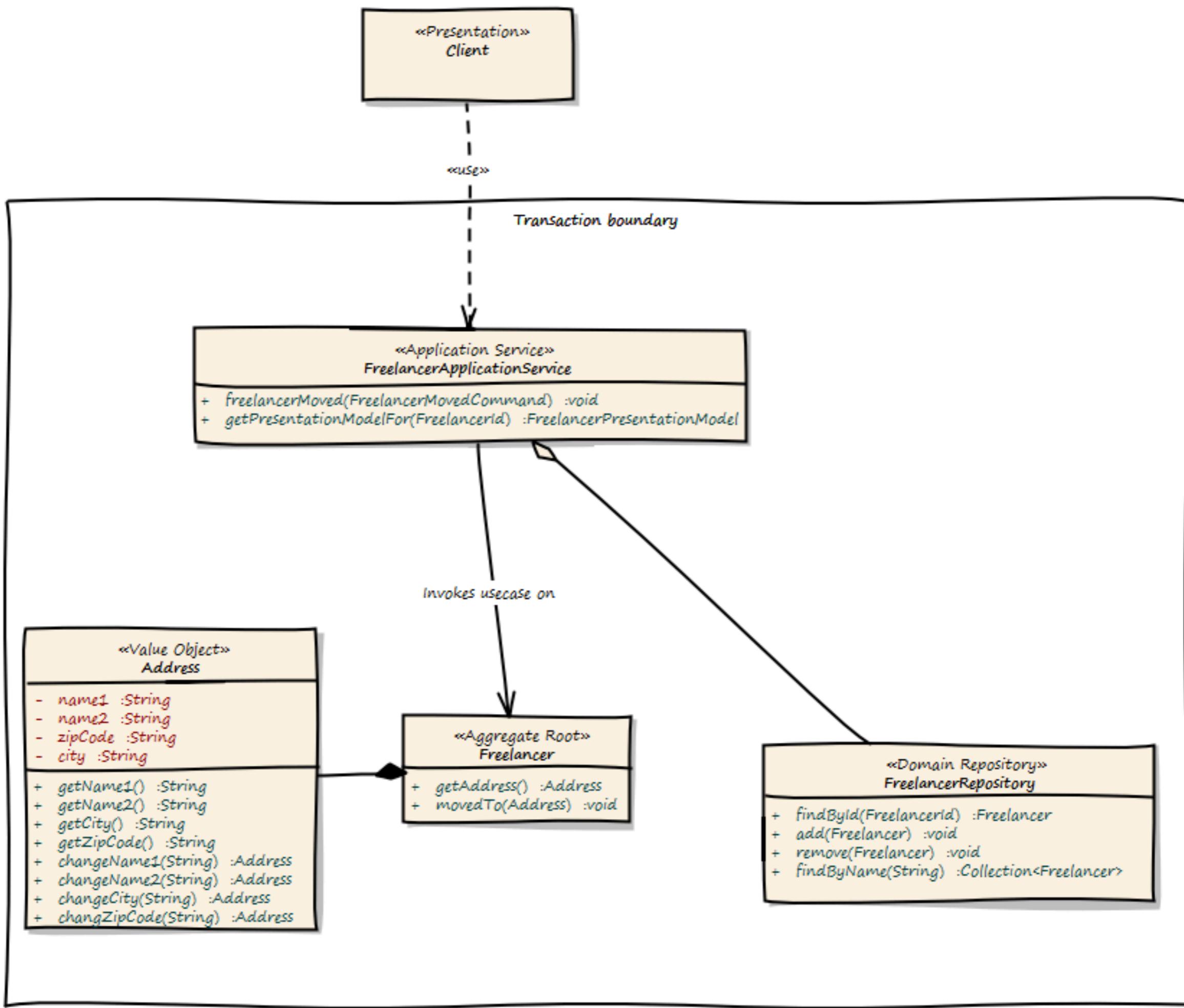
Identity and
Access
Management
Subdomain

Freelancer
Management
Subdomain

Customer
Management
Subdomain

Project
Management
Subdomain





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