DANIEL STORI {TURNOFF.US}

» 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

> Your Technology stack
> Total Years of experience

MONOLITH

Let's all give it up for Monolith, who is about to retire. I'd also like to take this opportunity to proudly introduce...

COOL DEV

CLAP CLAP CLAP CLAP CLAP

GET OUT, CHUBBY!

THE MICROSERVICES

COOL DEV

CLAP CLAP CLAP CLAP HUHUUUU CLAP CLAP CLAP CLAP CLAP YEAH CLAP CLAP YEAH HUHUUUU YEAH

Gimme five, buddy.

Talk to my API gateway please...

I'm not sharing a database with you!

Well, they might need some ORCHESTRATION first.

MONOLITH

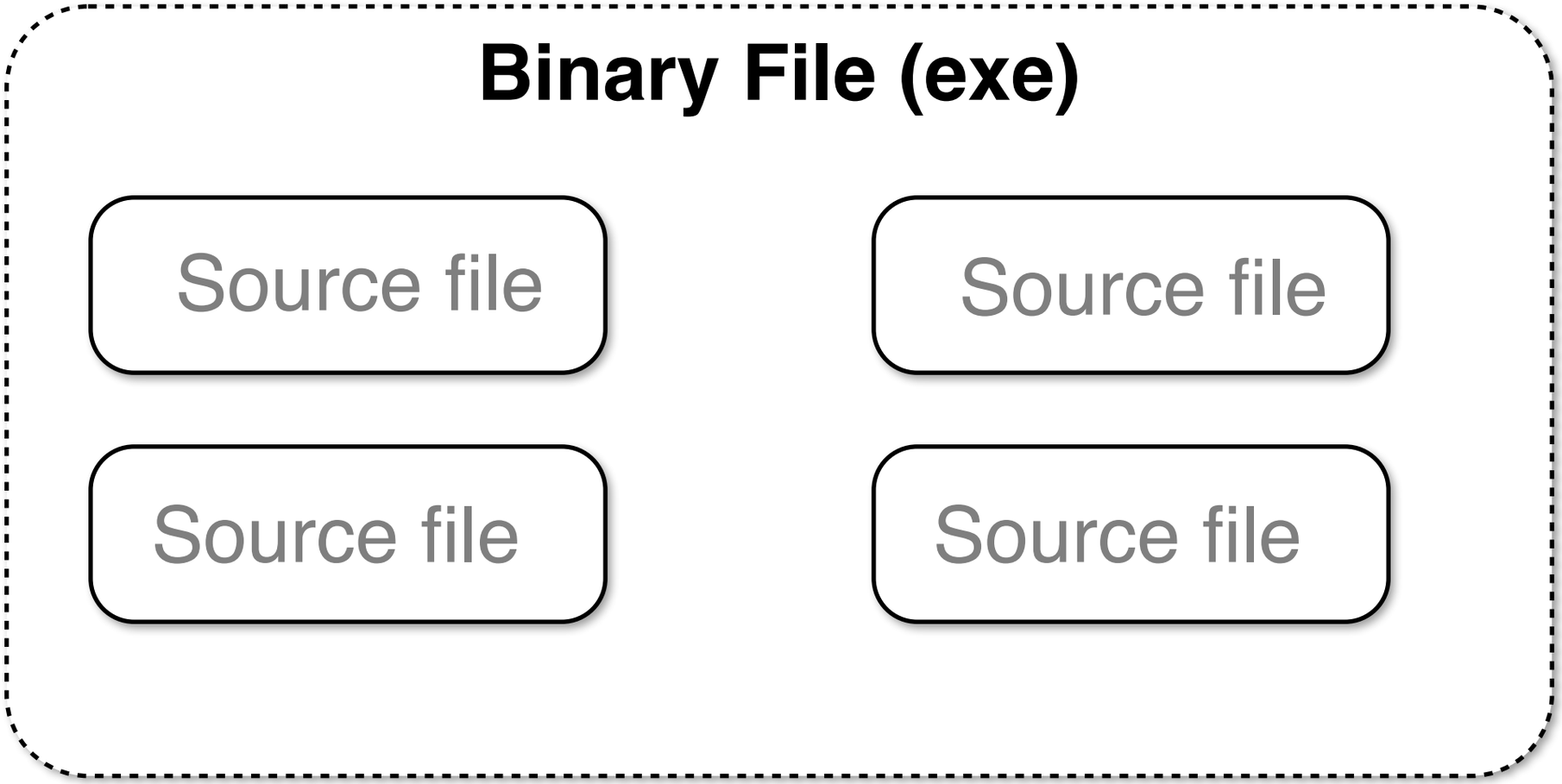
COOL DEV

Daniel Stori {turnoff.us}

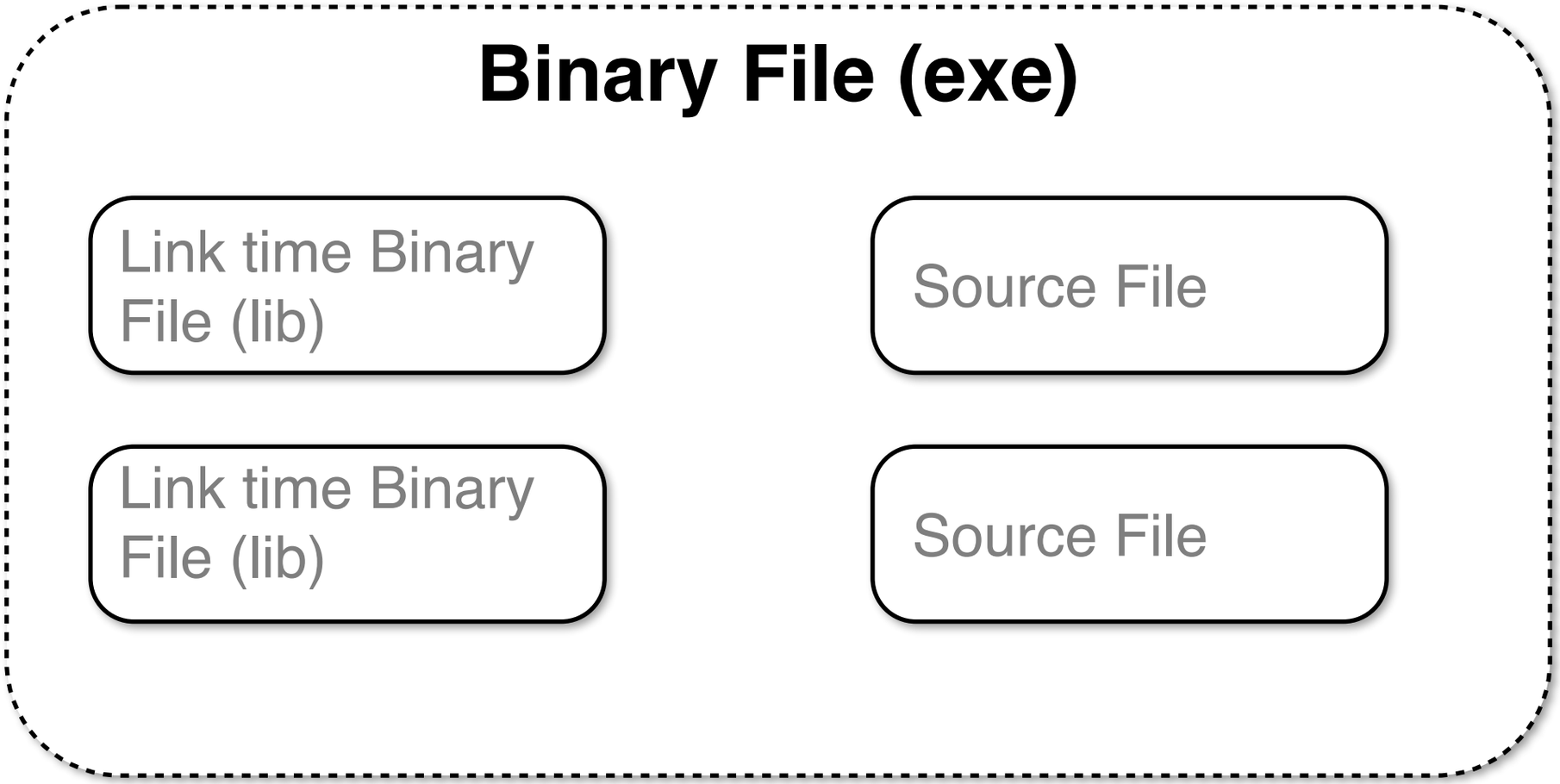
Day 1

What is Microservice ?

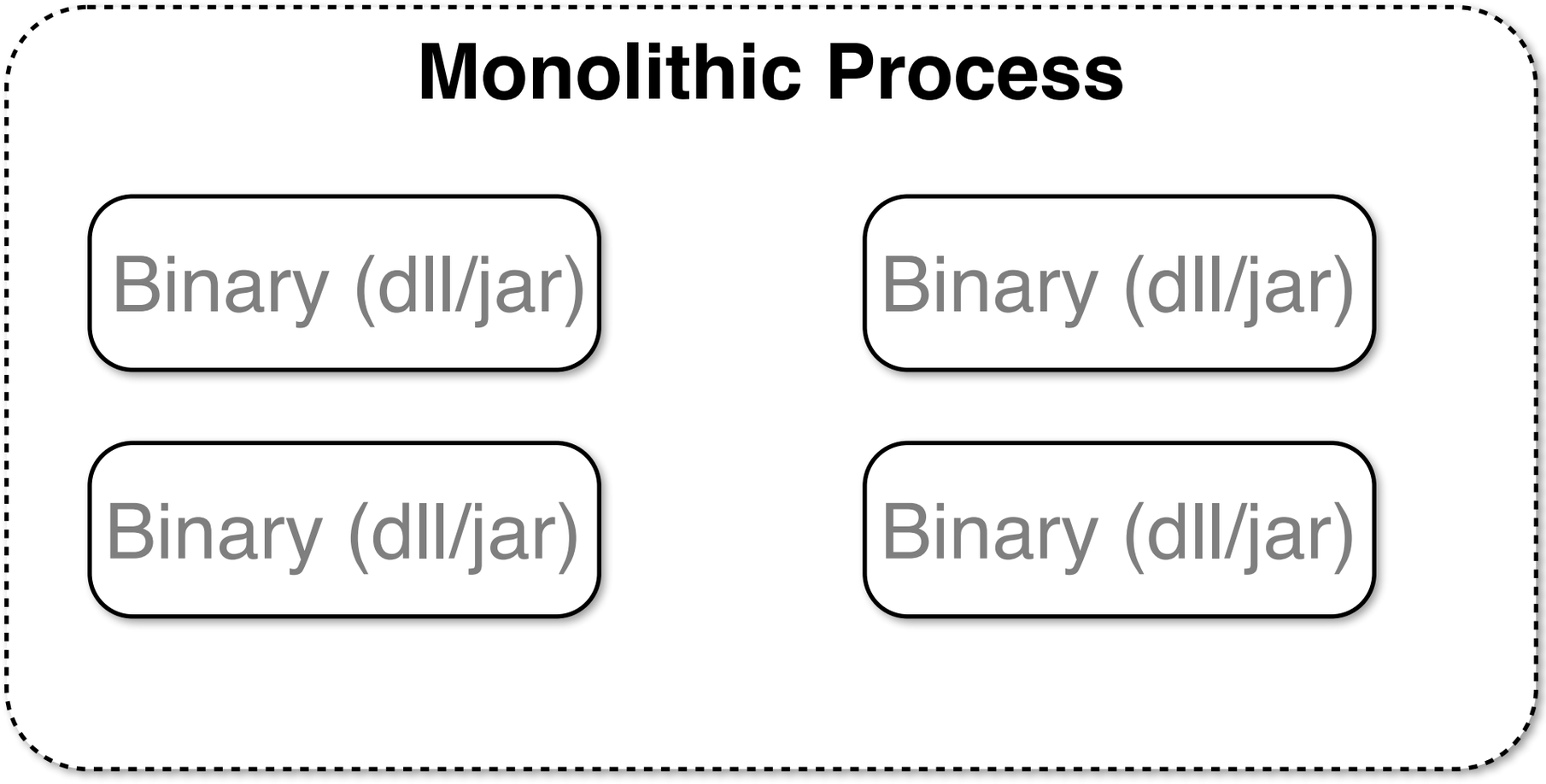
Compiletime Monolithic



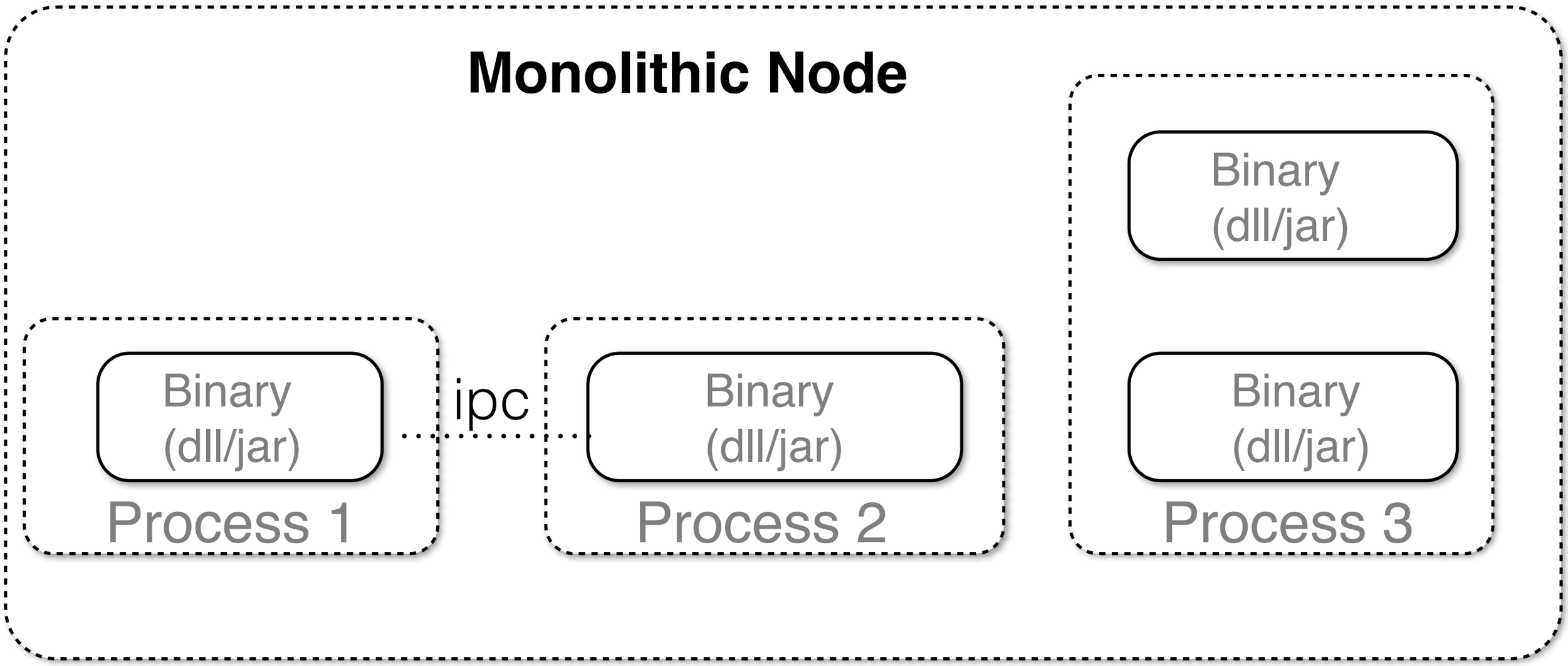
Linktime Monolithic / Monolithic Deployment



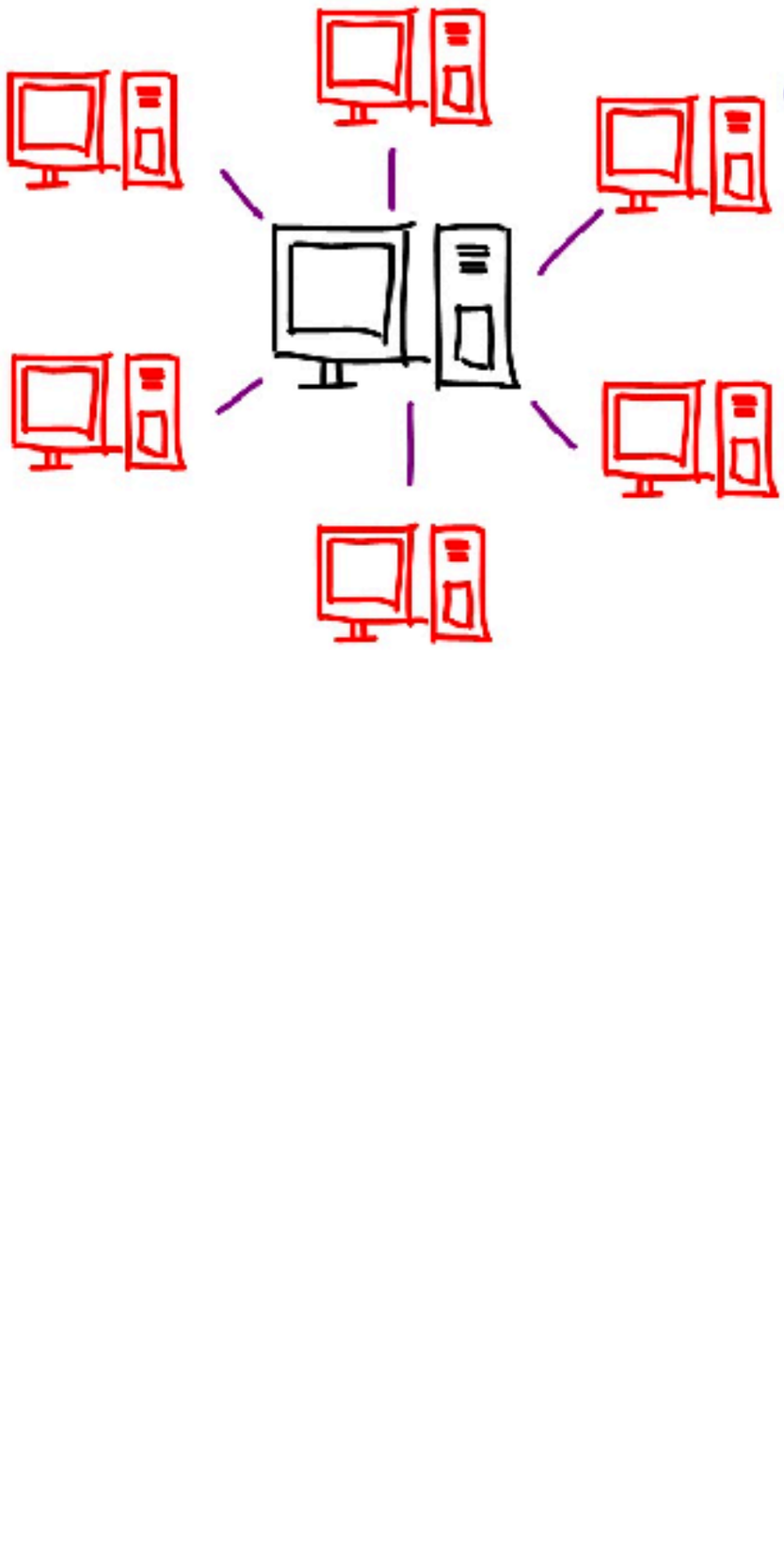
Runtime Monolithic



Multi Process Application

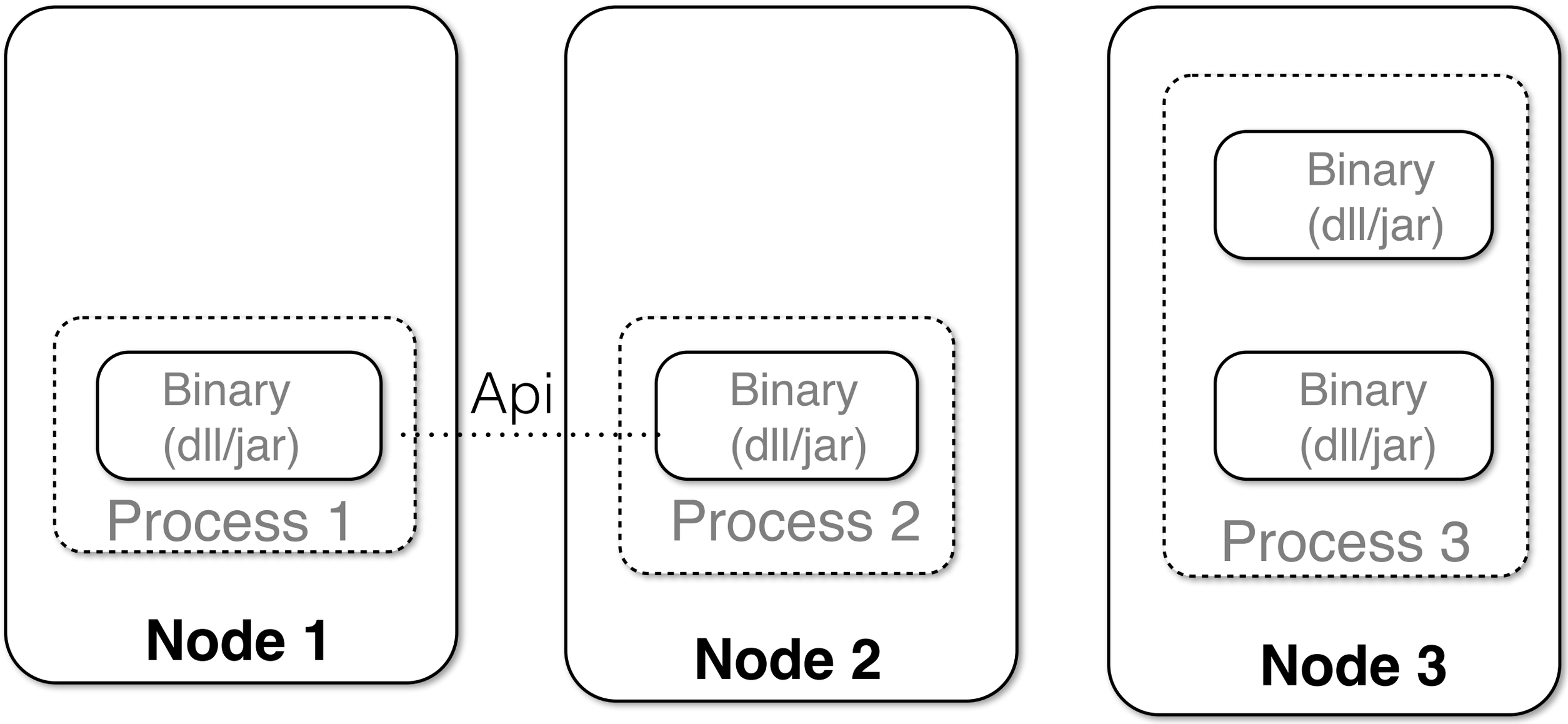


Distributed Computing

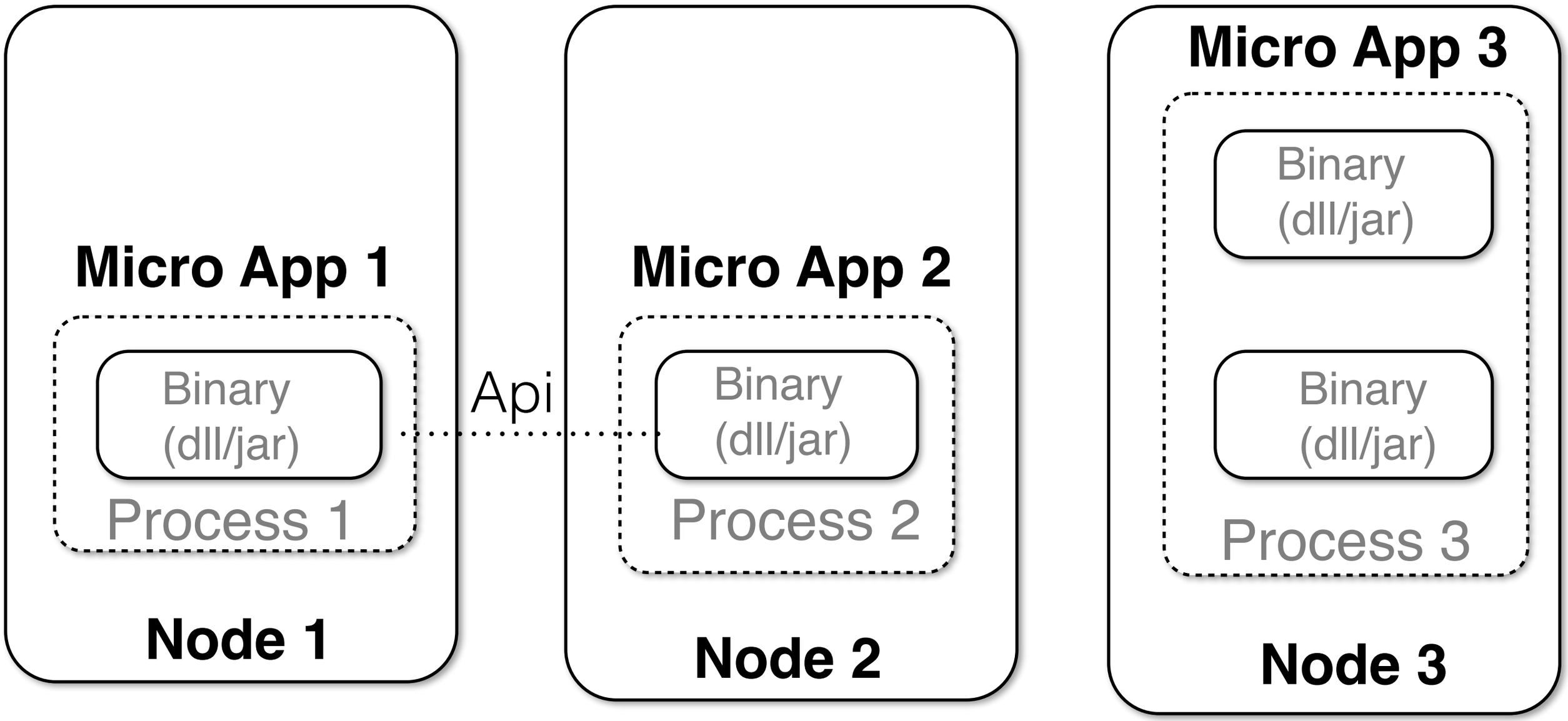


- DCOM
- CORBA
- SOAP
- Message Queue
- REST
- GRPC
- THRIFT

Distributed Application



Micro Distributed Application

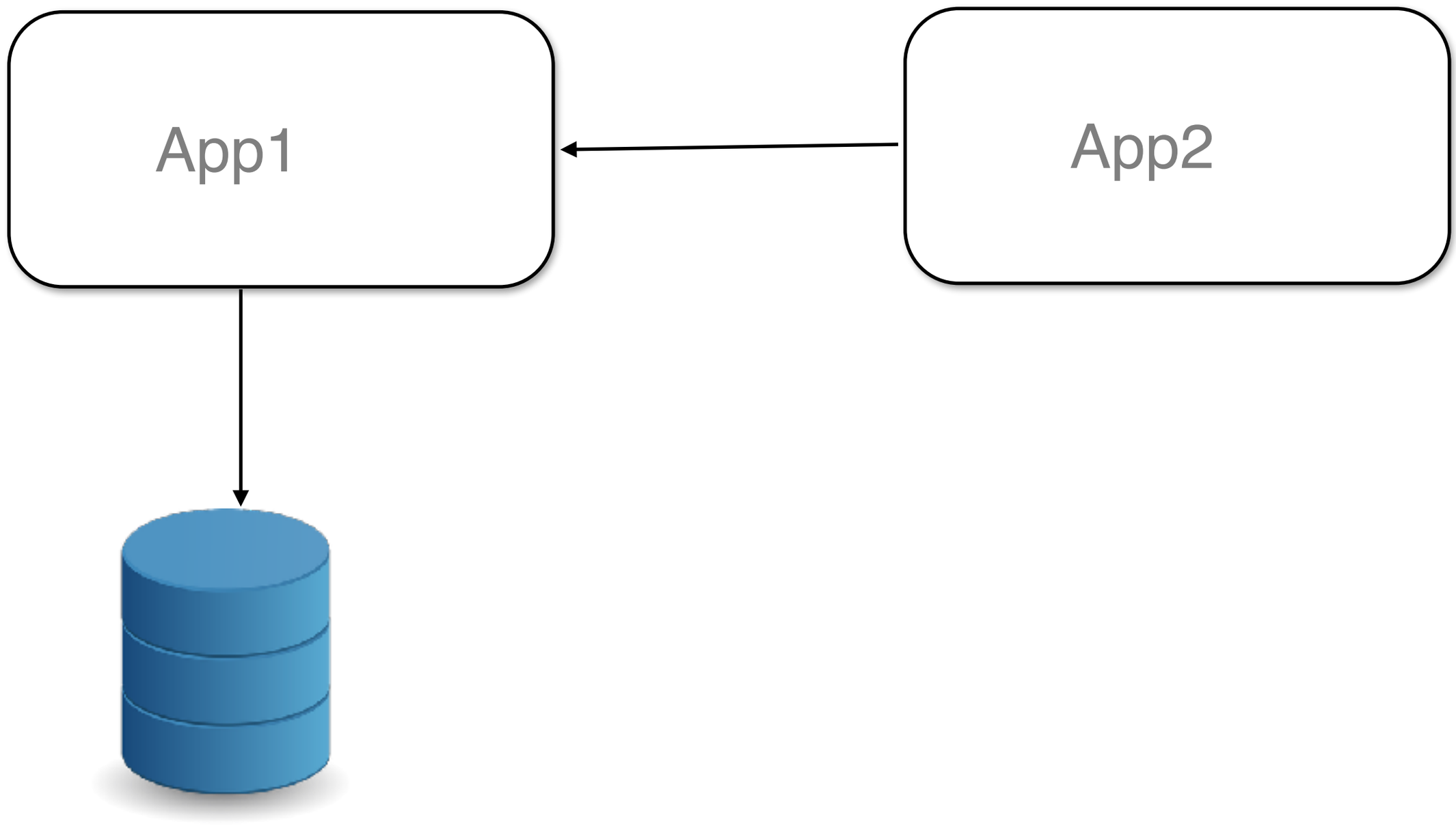


Traditional Distributed Application
vs
Micro Distributed Application

	Monolithic	**Micro service**
Database / Storage	Shared	Not shared
Infra (Hosting)	Shared	Not Shared
CI/CD (Build Server)	Shared	Not Shared
Fun Requirements	Shared	Not Shared
SCRUM Team / Sprint	Shared	Not Shared
Test Cases	Shared	Not Shared
Sorce Control	Shared	Not Shared
Architecture	Shared	Not Shared
Technology Stack / Fwks	Shared	Not shared

App 1

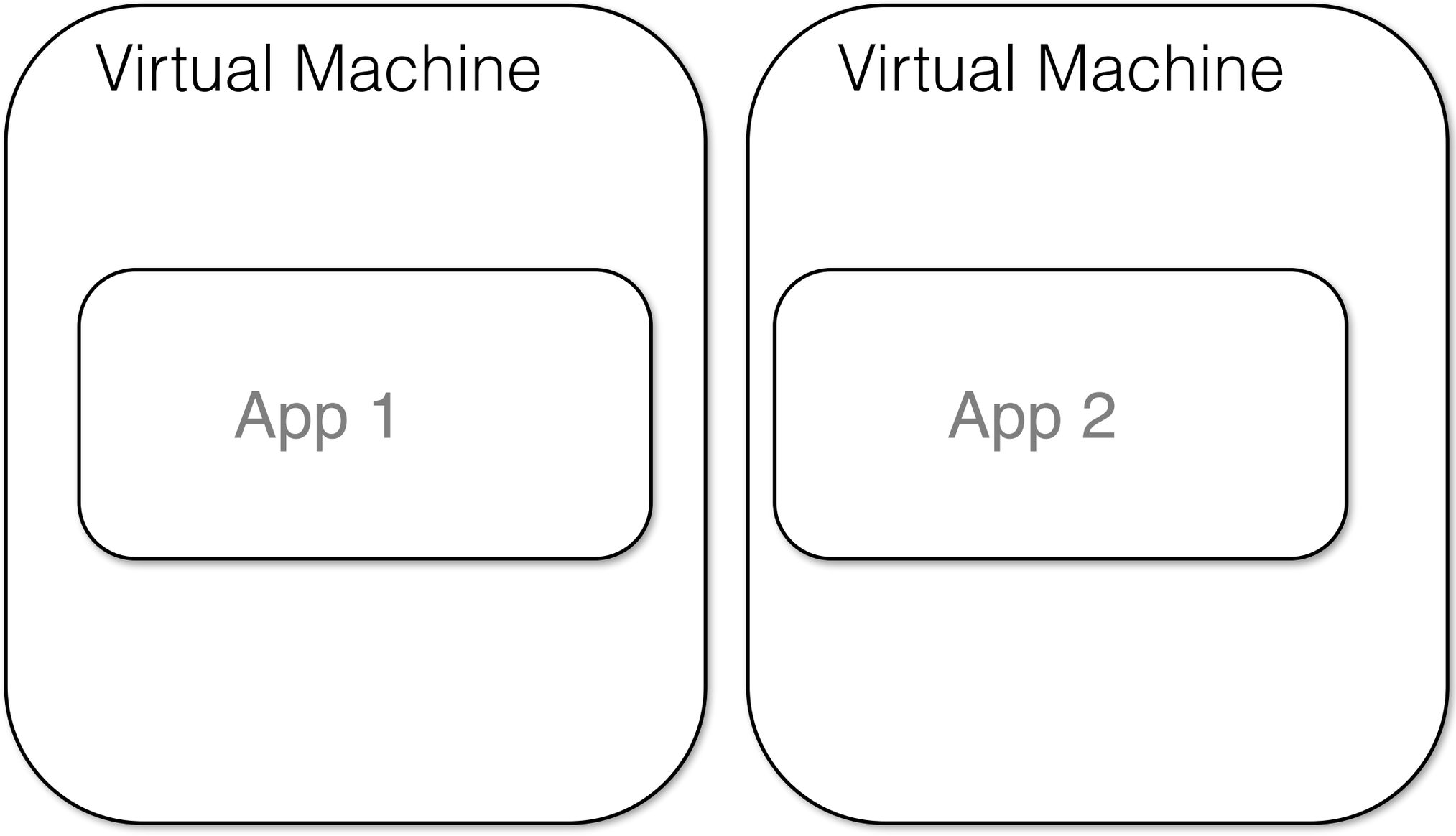
App 2


App1
App2

Virtual Machine

App 1

App 2

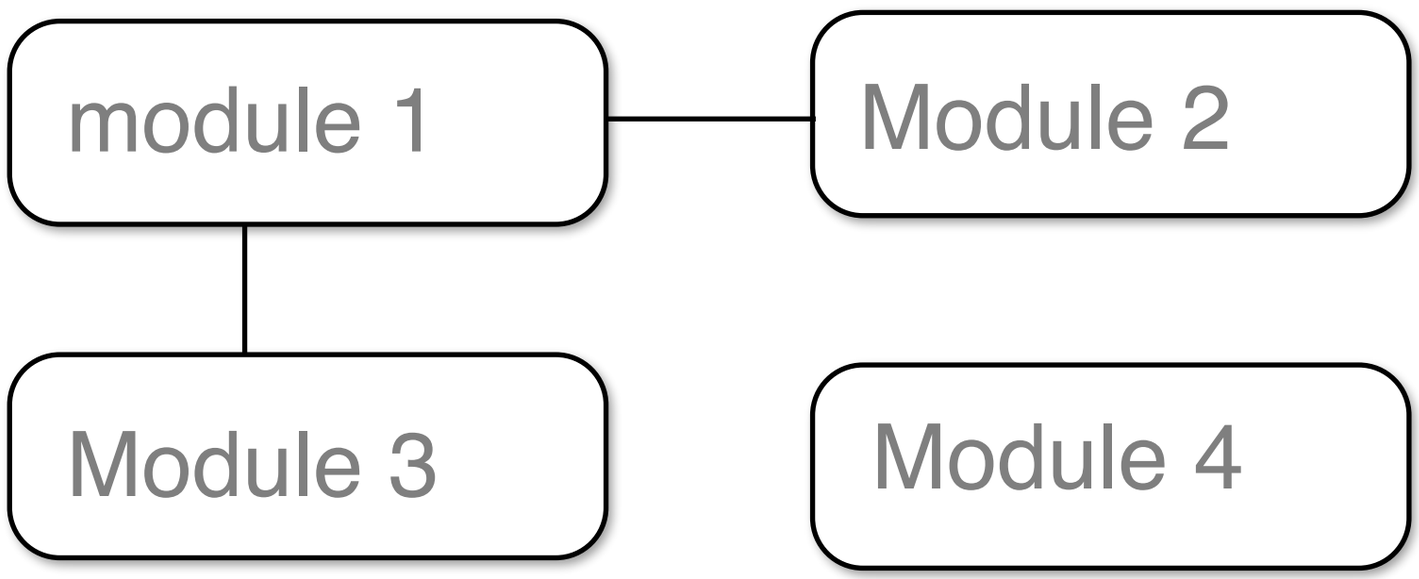

Virtual Machine
App 1
Virtual Machine
App 2

App/mod 1

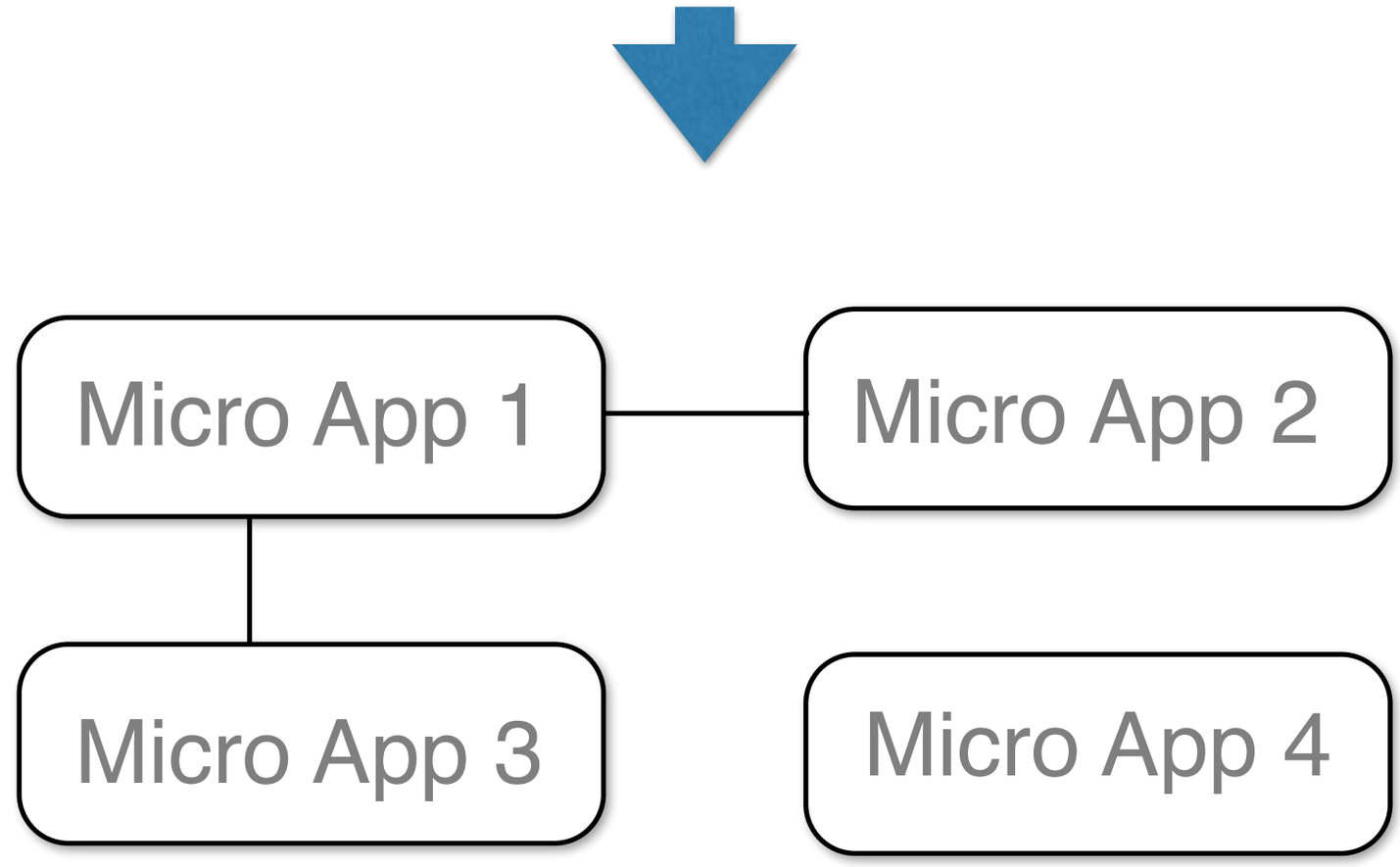
App/mod 2

Shared
Domain logic

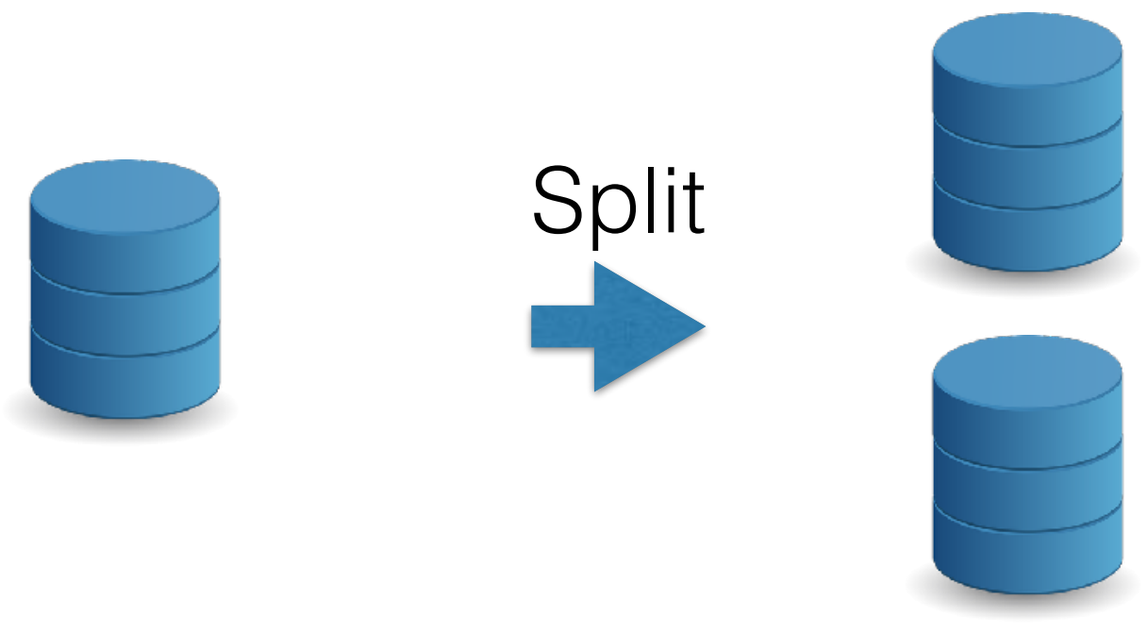
Application

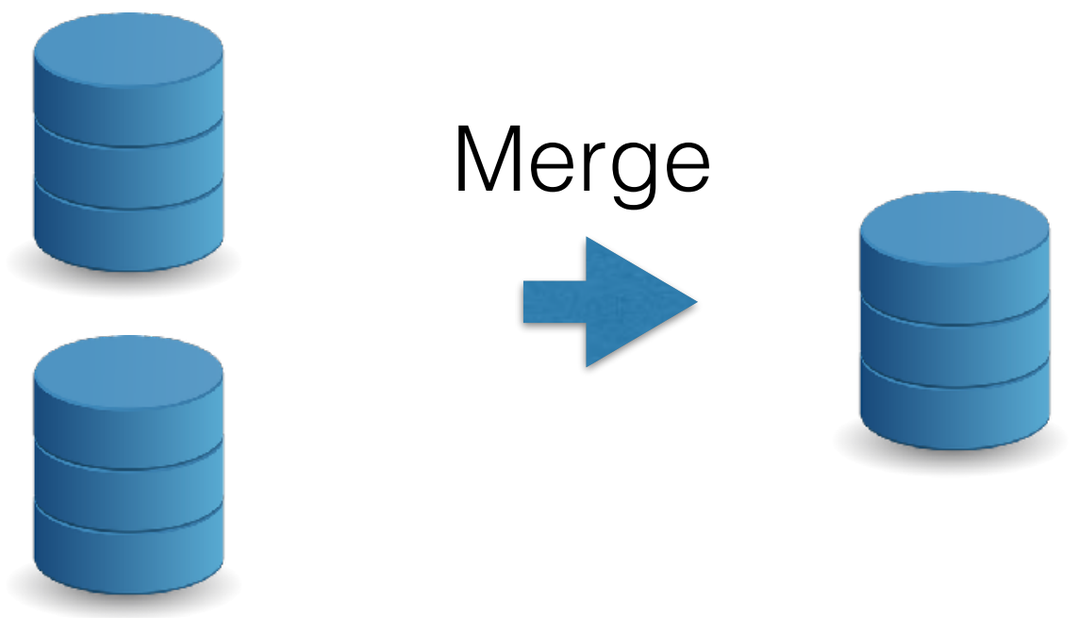


Application

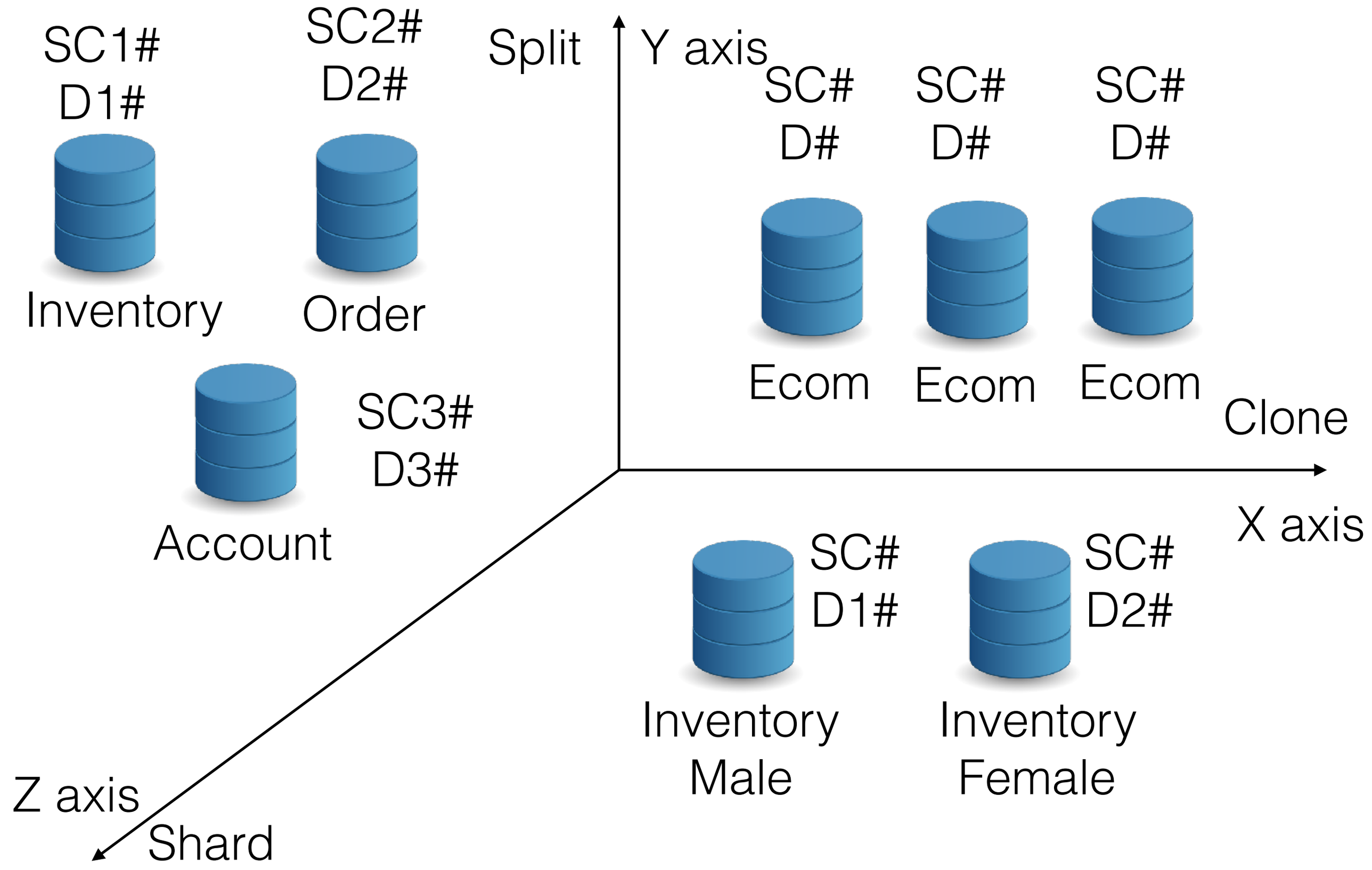


	Pros/ Cons
Performance	- -
Transaction Management	- -
Joins / Report / Materialized view	- -
Development time (1:3)	- -
Learning Curve	- -
Infra Cost	- -
Debugging (Finding bugs)	- -
Integration Test	- -
Log Mgmt	- -
Config Mgmt	- -
Authentication	- -
Authorization	- -
Monitoring / Alerting	- -
CI/CD	- -
Agile Architecture (Incremental arch)	+ +
Feature Shipping	+ +
Scalability	+ +
Resilent	+ +
Polygot	+ +
Maintainability	+ +


Split


Merge

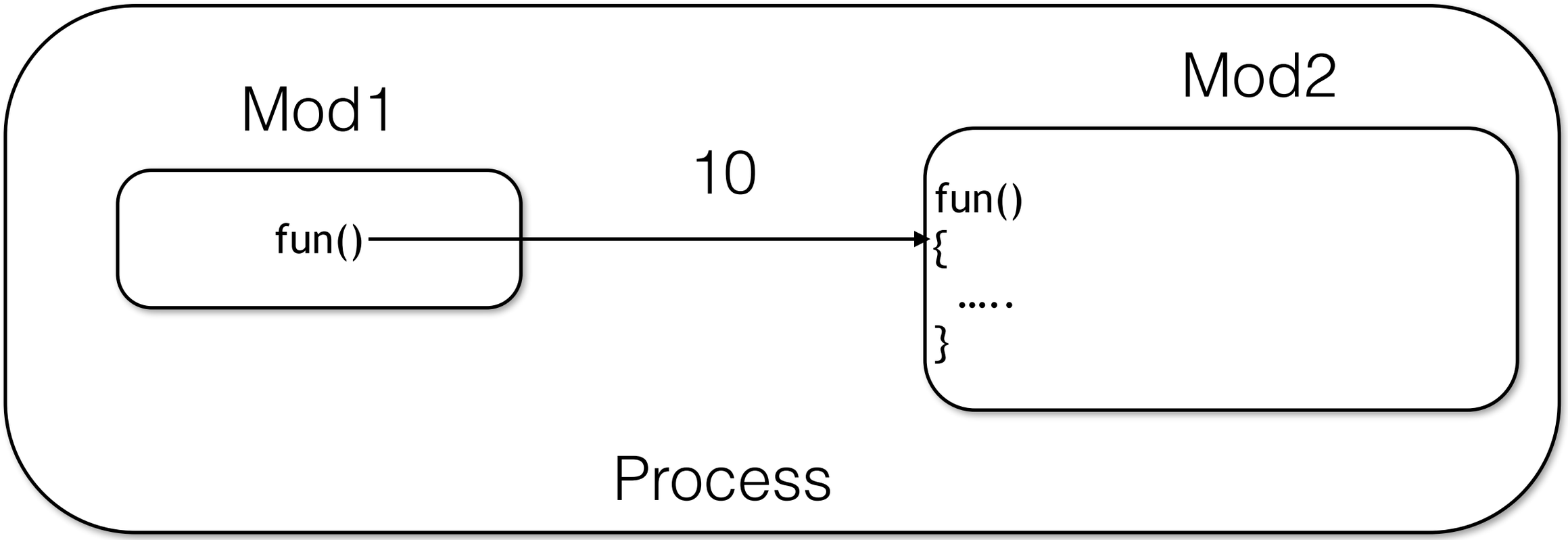
Scalability Cube - 50 rules for high Scalability

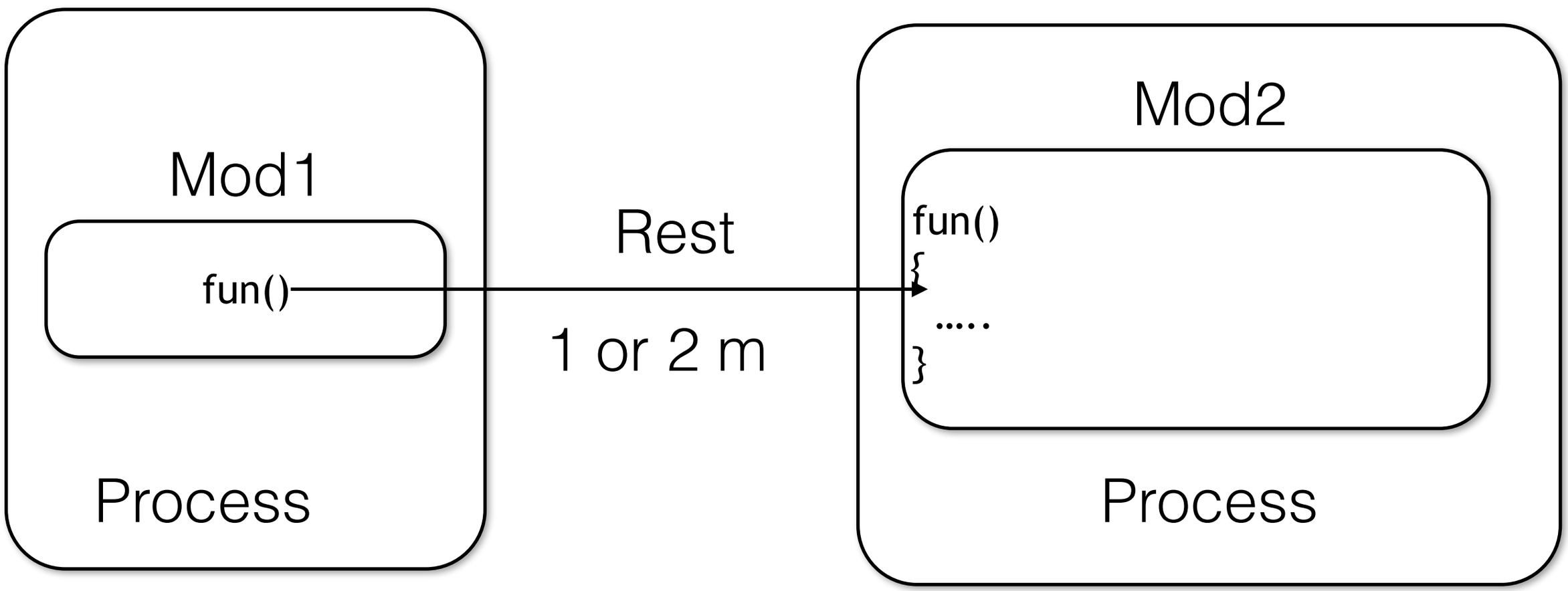


YAGNI - you aren’t gonna need it

KISS

Solve tomorrow’s problem tomrrow


Mod1
Mod2
10
fun()
fun()
{
.....
}
Process


Mod1
Mod2
Rest
1 or 2 m
fun()
fun()
{
.....
}
Process
Process

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

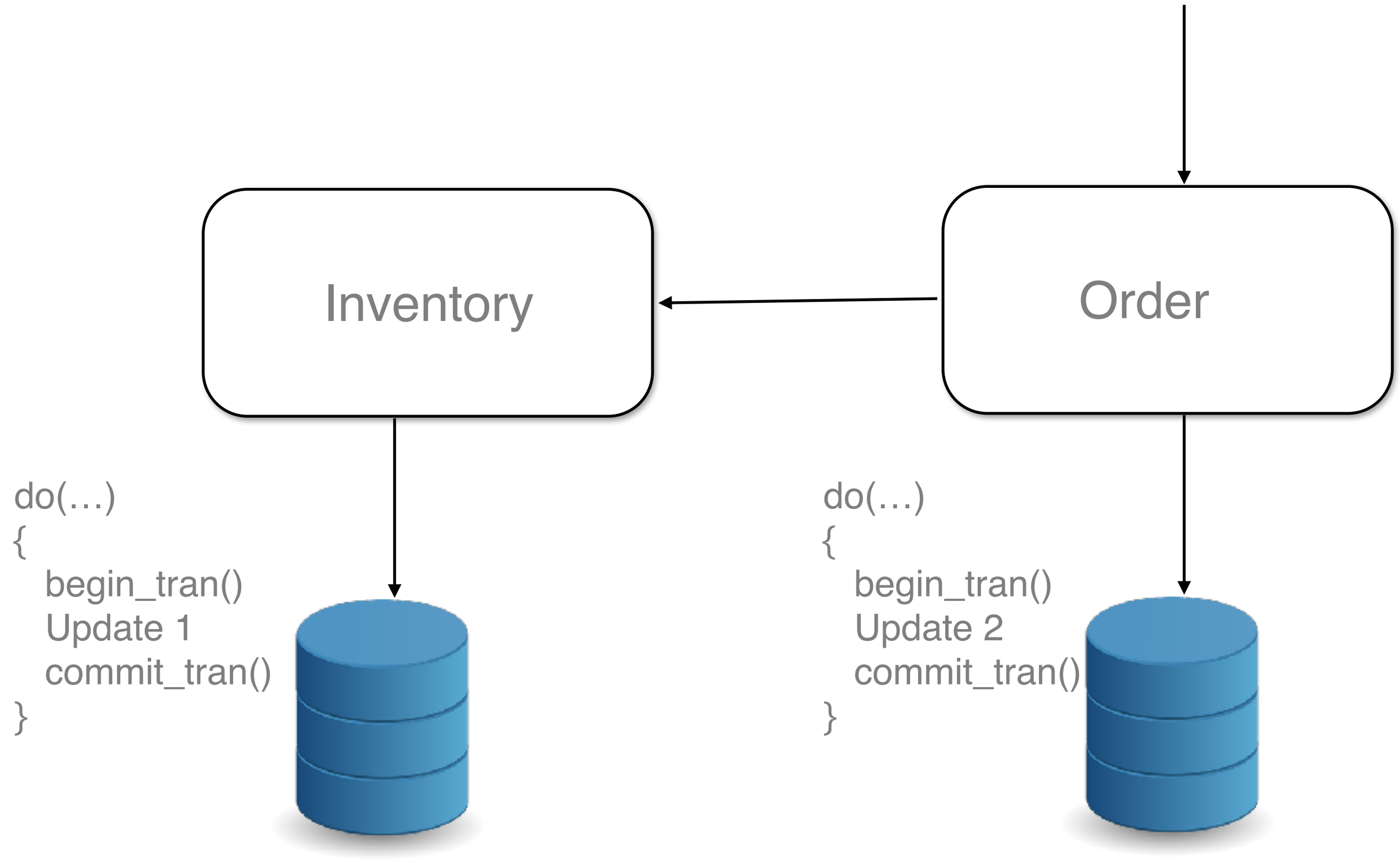
Transaction

Inventory

Order

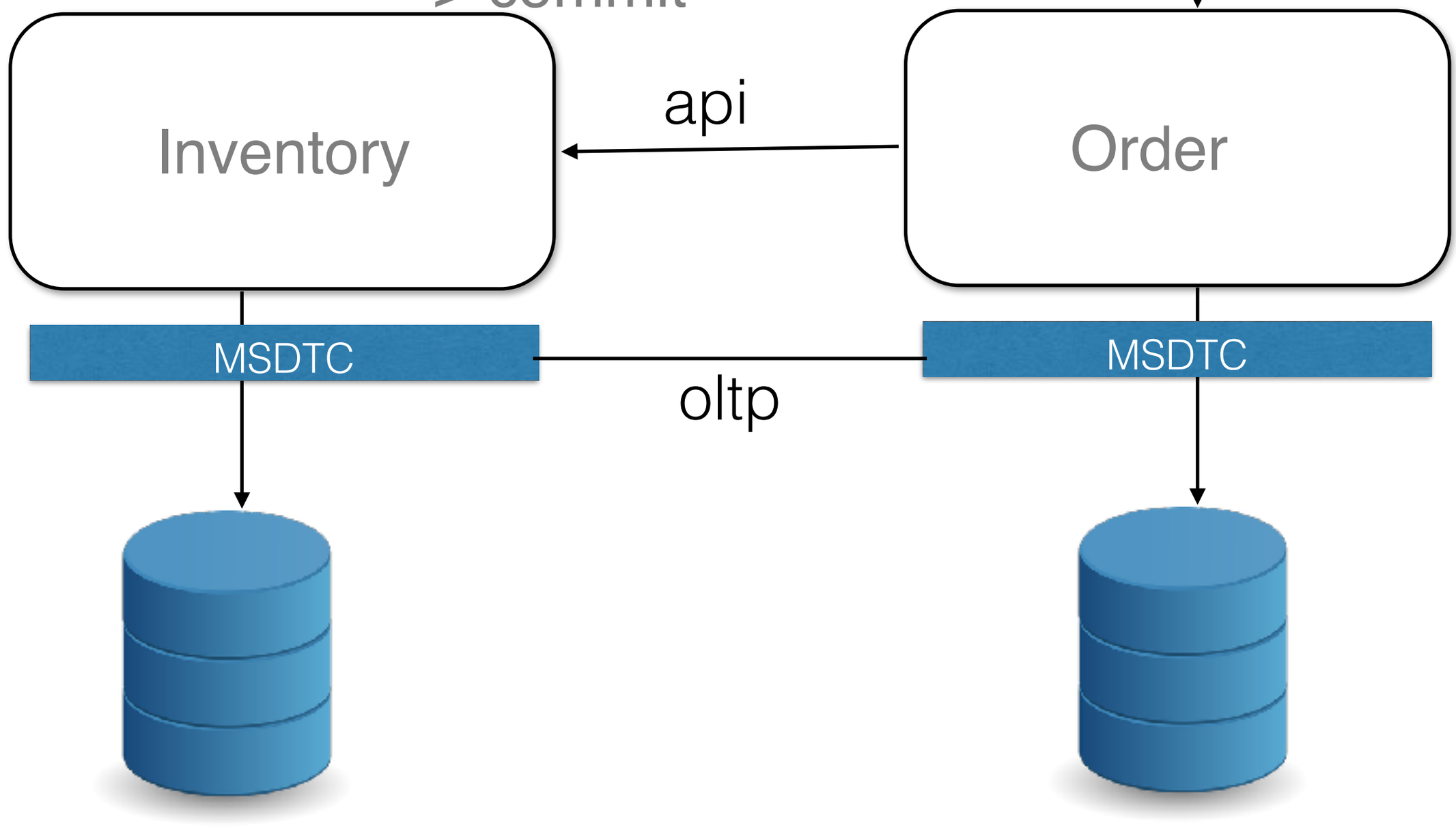
```
do(...)
{
   begin_tran()
   Update 1
   Update 2
   commit_tran()
}
```

Transaction



2 phase commit
JTX , MSDTC, …

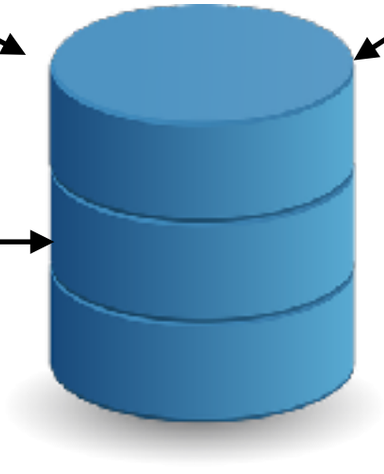
msdtc.begin_tran()
Update 1
Update 2
mstdc.commit_tran()
-> take votes
-> commit



Mod1

Mod1

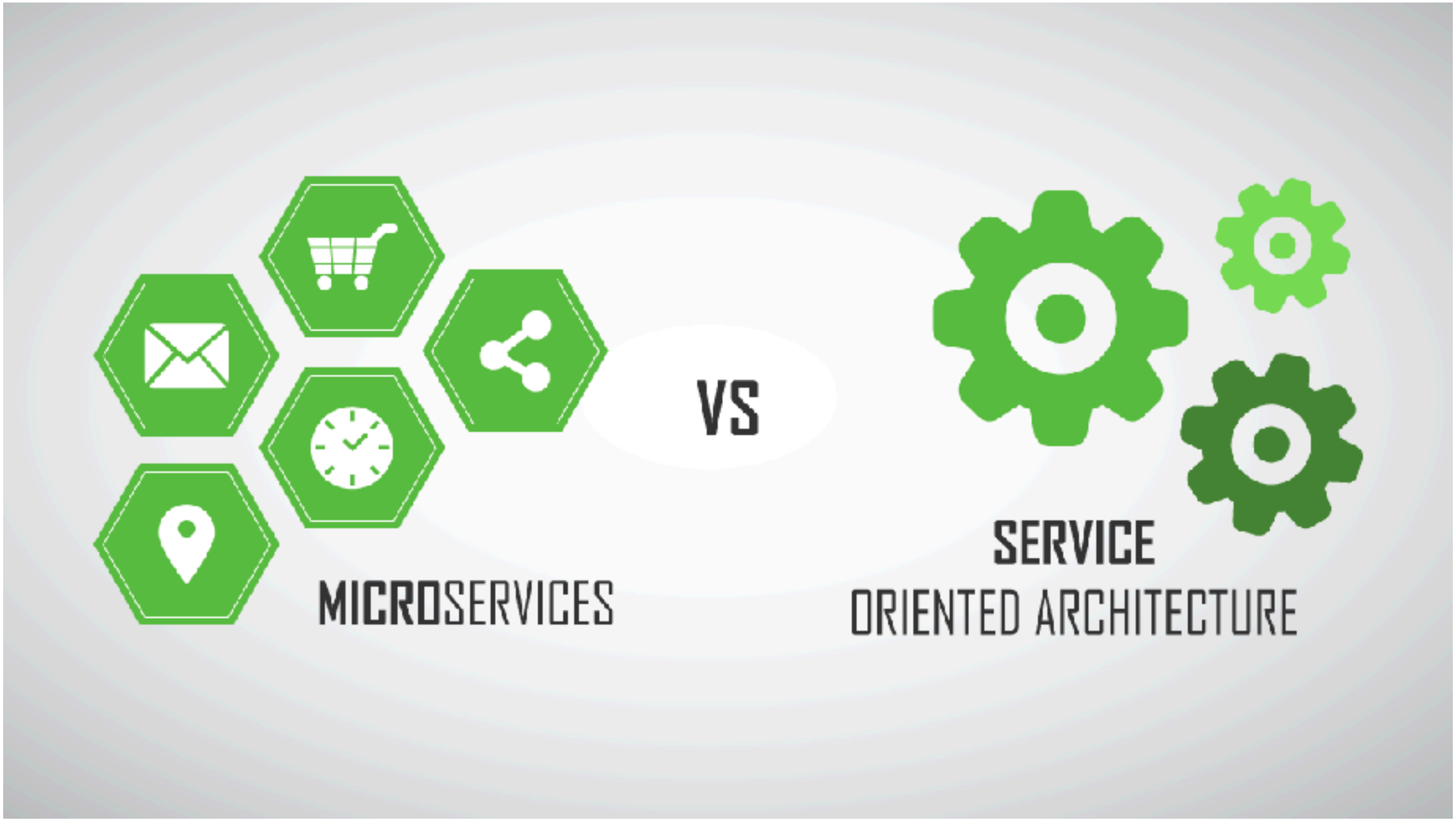
Join Query



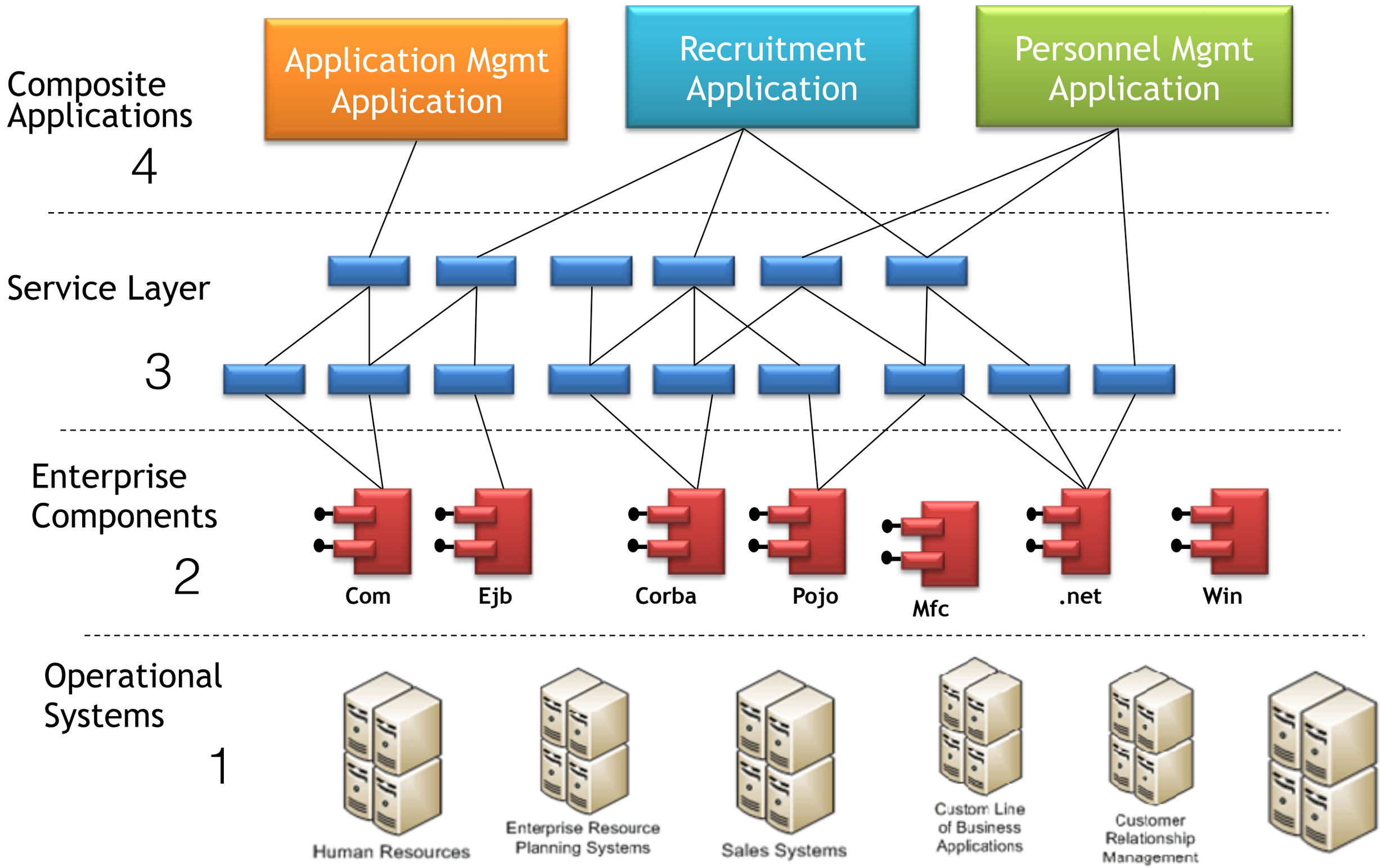
App1

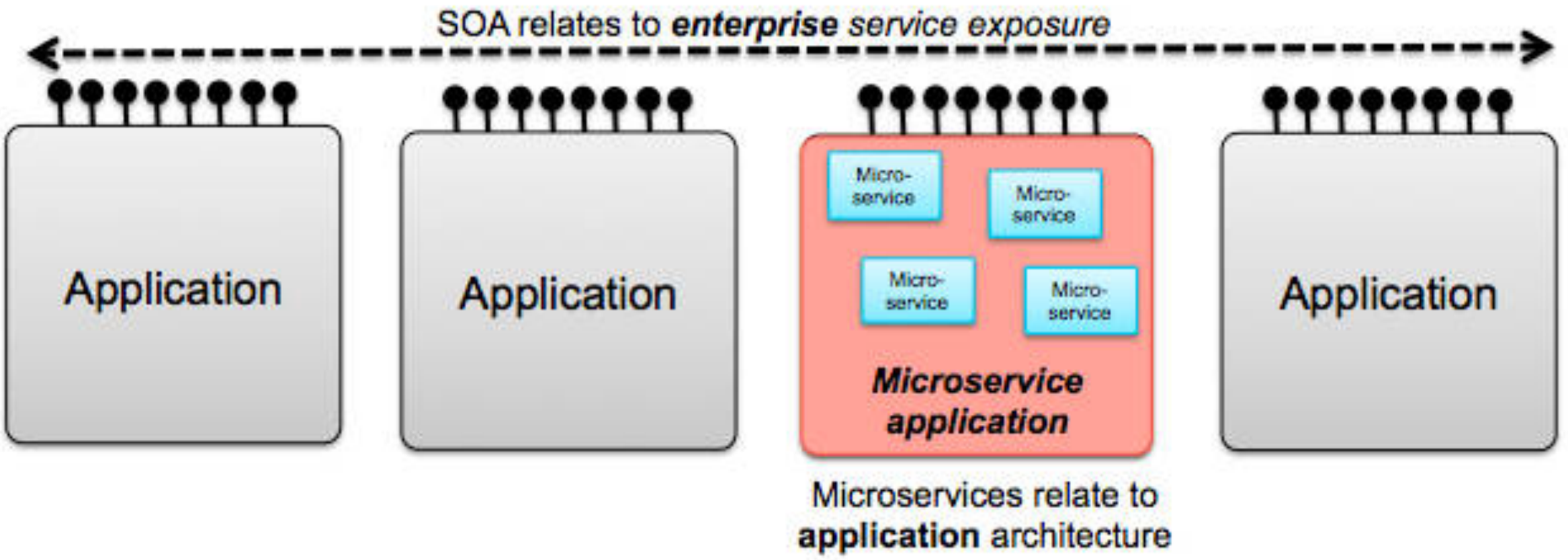
App2

Join Query


VS
MICROSERVICES
SERVICE
ORIENTED ARCHITECTURE

Service Oriented Architecture





Microservices relate to **application** architecture

Day 2

	Approach
Performance	
Transaction Management	
Joins / Reports	
Infra Cost	
Deployment	
Debugging (Finding bugs)	
Log Mgmt	
Config Mgmt	
Authentication	
Authorization	
Monitoring / Alerting	

Isolated Env.

Virtual Machine

App 1

App 2

VM1

VM2

My app 1

My app 2

Middlewear

Middlewear

Unix Guest Os

Win Guest Os

Cpu
RAM
Disk
I/o

Cpu
RAM
Disk
I/o

Virtualisation layer

Host Os

Node

Container 1

Container 2

Cpu
RAM
Disk
I/o

My app 1

Middlewear

My app 2

Middlewear

Cpu
RAM
Disk
I/o

Containerization layer (Docker)

Host Os

Node

VM1

VM2

Container 1

Container 2

My app 1

My app 2

Middlewear

Middlewear

Cpu RAM Disk I/o

Cpu RAM Disk I/o

Containerization layer (Docker)

Unix Guest Os

Win Guest Os

Cpu RAM Disk I/o

Cpu RAM Disk I/o

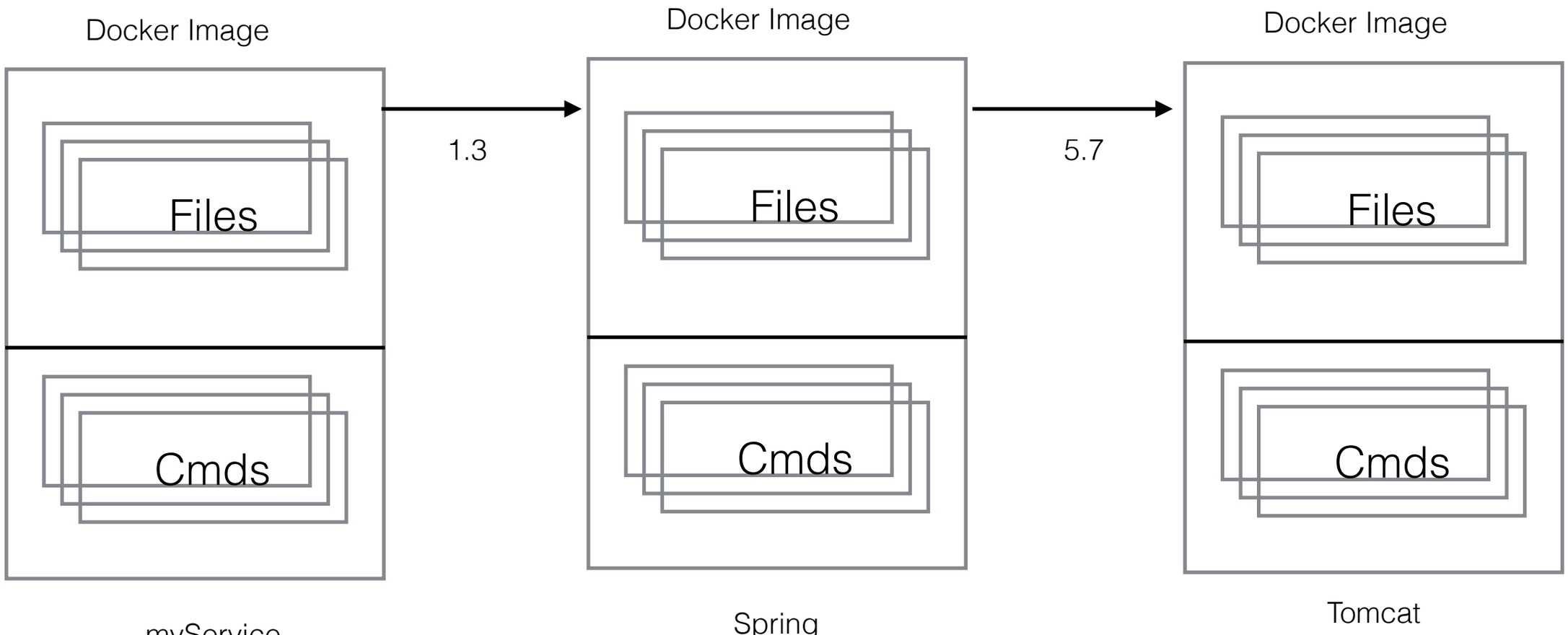
Virtualisation layer

Host Os (azure)

Machine

	Approach
Performance	
Transaction Management	
Joins / Reports	
Infra Cost	Container
Deployment	
Debugging (Finding bugs)	
Log Mgmt	
Config Mgmt	
Authentication	
Authorization	
Monitoring / Alerting	

Reproducable Env.


Docker Image
Files
Cmds
myService
1.3
Docker Image
Files
Cmds
Spring
5.7
Docker Image
Files
Cmds
Tomcat

Image

Docker file (cmds)

Config Files

Binary Files

Push

hub.docker.com (Repository)

Image

Pull

Build

VM

Container

8080 8080

Nodes

CPU
RAM
DISK
I/O

Run **cmds** in docker file

Config File

Source File

Source File

	Approach
Performance	
Transaction Management	
Joins / Reports	
Infra Cost	Container
Deployment	Docker Image, Hub
Debugging (Finding bugs)	
Log Mgmt	
Config Mgmt	
Authentication	
Authorization	
Monitoring / Alerting	

Managing Container Life cycle

VM1

VM2

Container 1

Container 2

My app 1

Middlewear

Cpu
RAM
Disk
I/o

Kubernetes (slave)

Containerization layer (Docker)

Unix Guest Os

Container 1

Container 2

My app 1

Middlewear

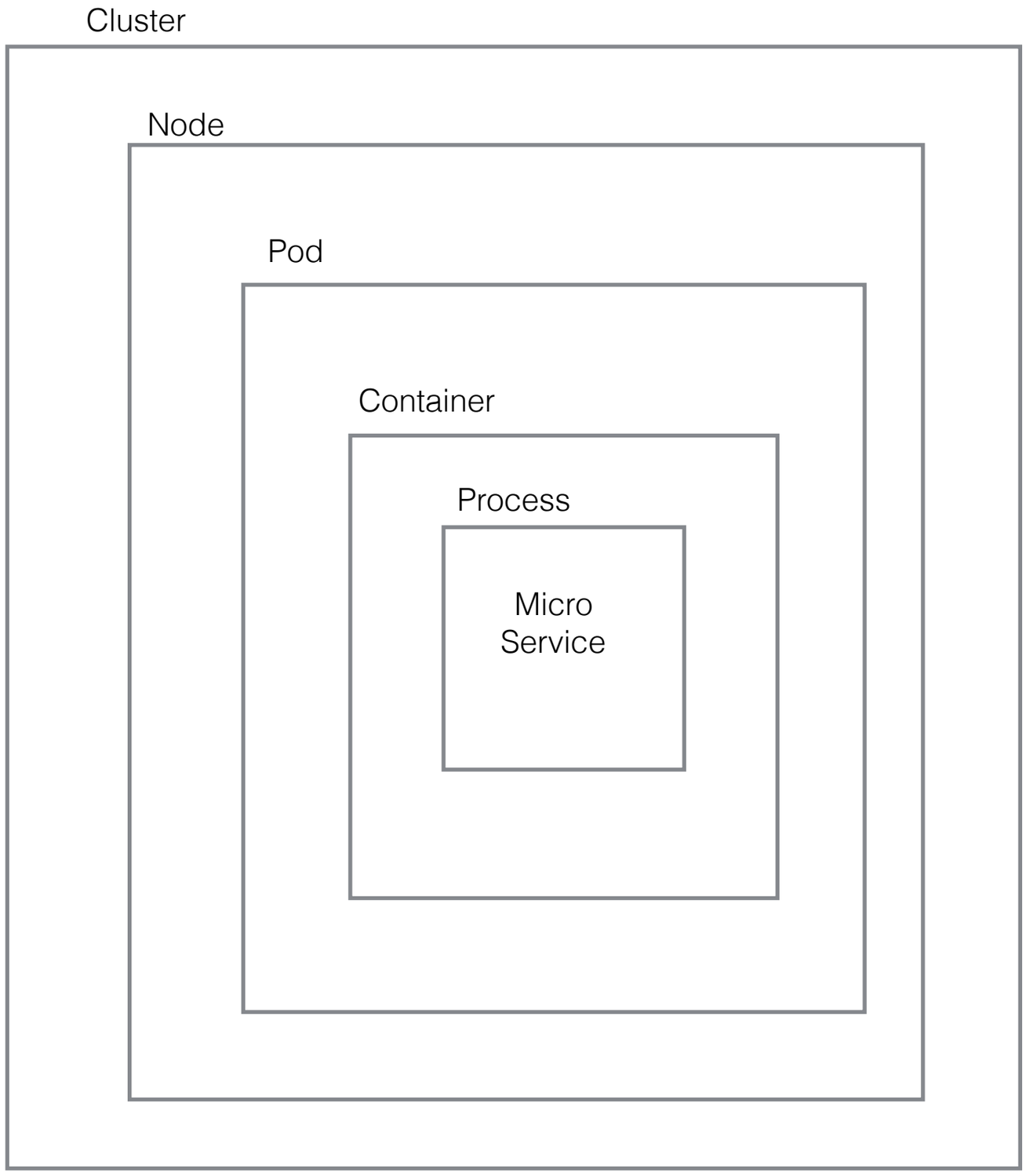
Cpu
RAM
Disk
I/o

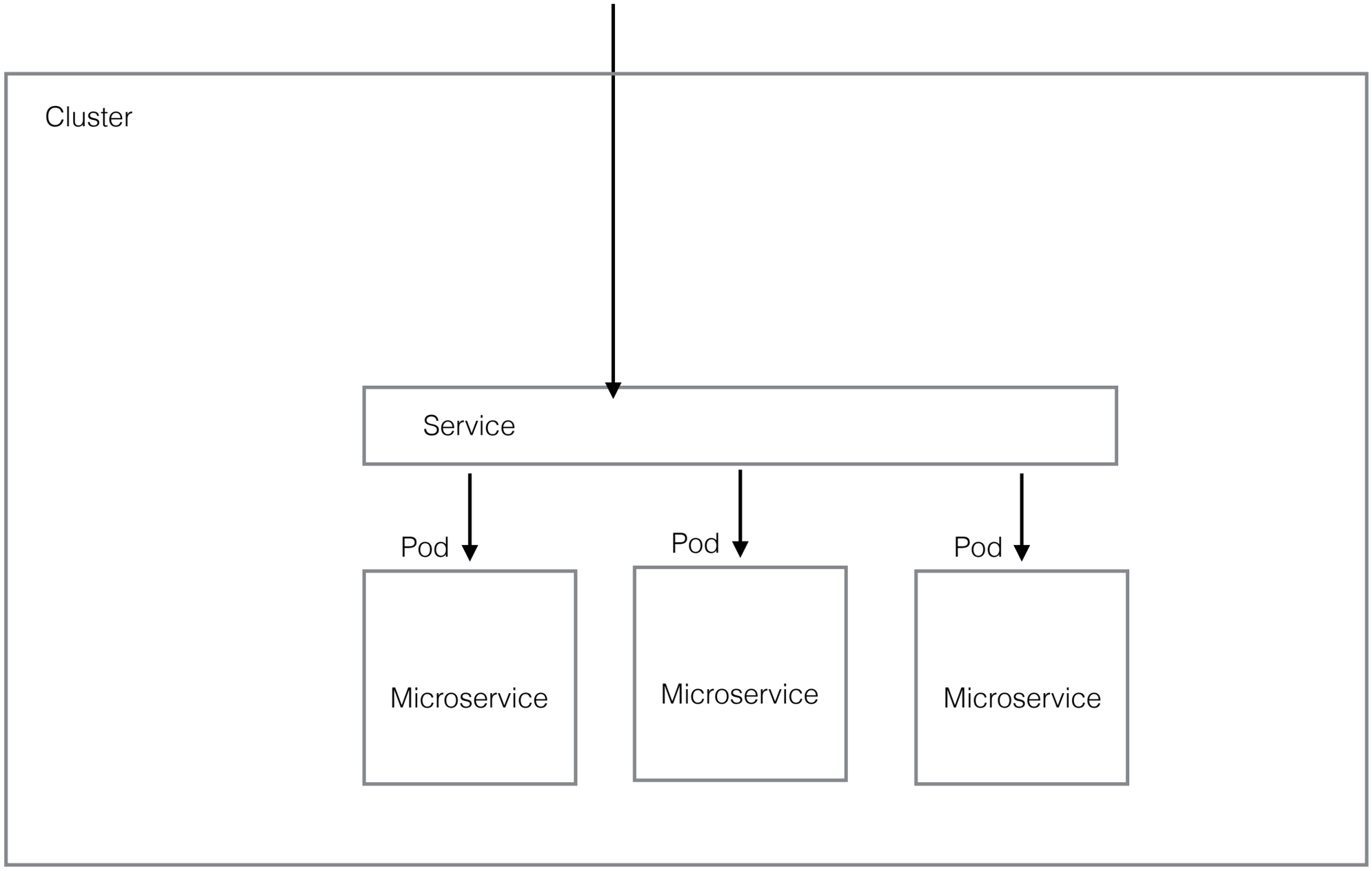
Kubernetes (Slave)

Containerization layer (Docker)

Unix Guest Os

Kubernetes (Master)

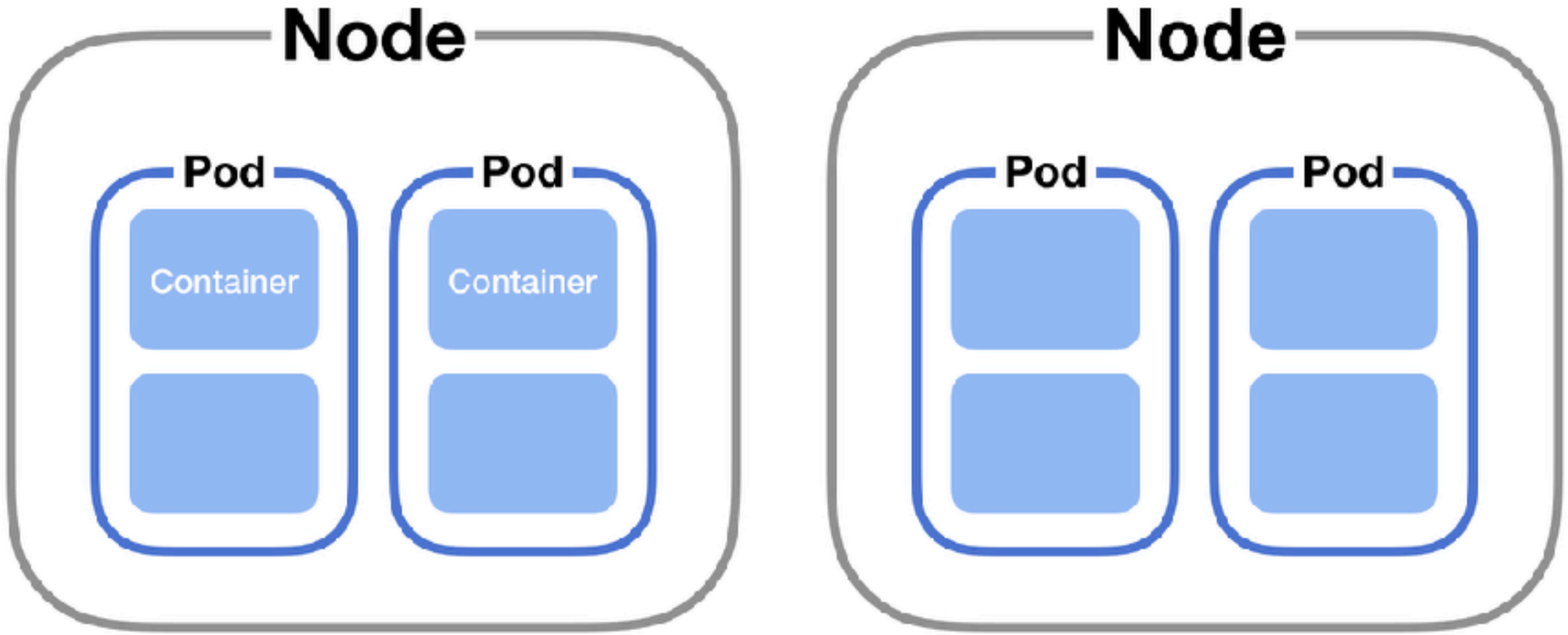

Cluster
Node
Pod
Container
Process
Micro
Service

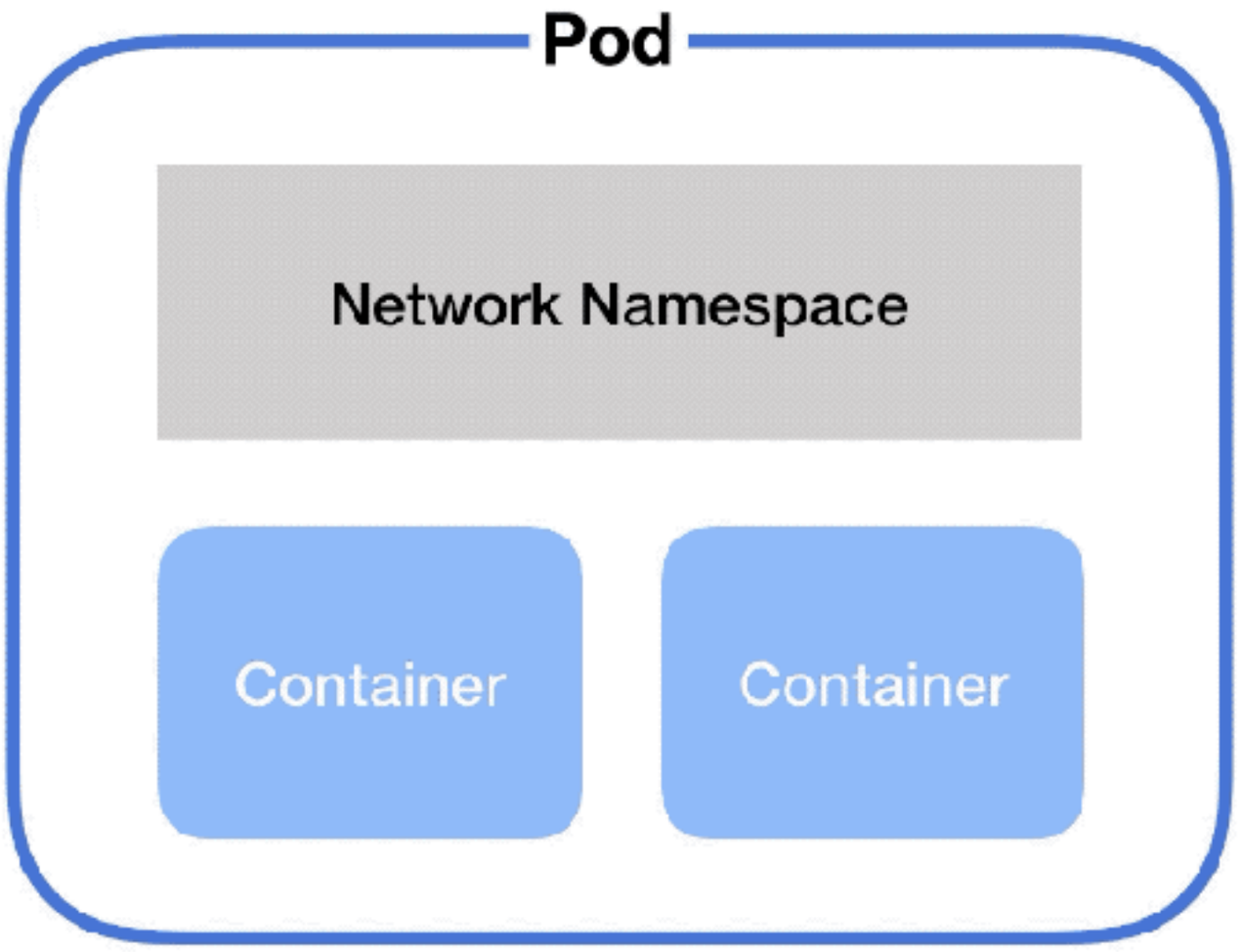

Cluster
Service
Pod
Pod
Pod
Microservice
Microservice
Microservice

	Approach
Performance	
Transaction Management	
Joins / Reports	
Infra Cost	Container, Kubernetes
Deployment	Docker Image, Hub
Debugging (Finding bugs)	
Log Mgmt	
Config Mgmt	
Authentication	
Authorization	
Monitoring / Alerting	Kubernetes

Service Discovery

Cluster




Pod
Network Namespace
Container
Container

Node

eth0

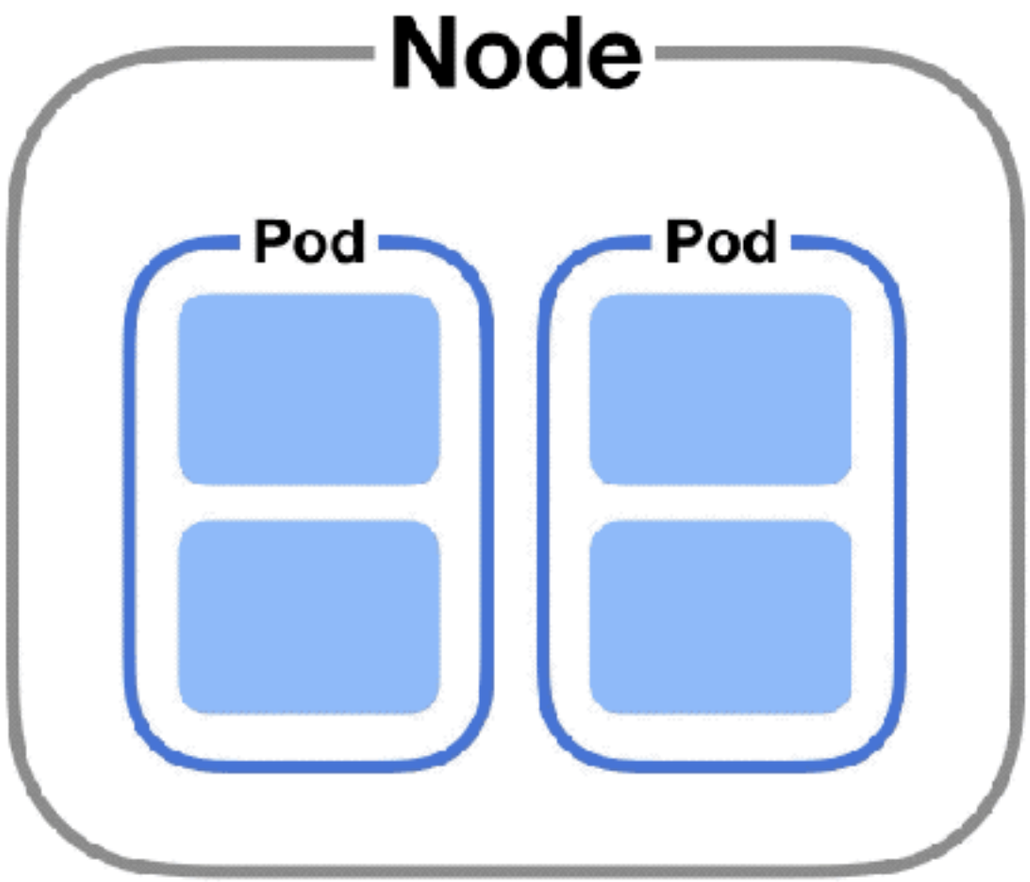
Pod 100.68.1.1

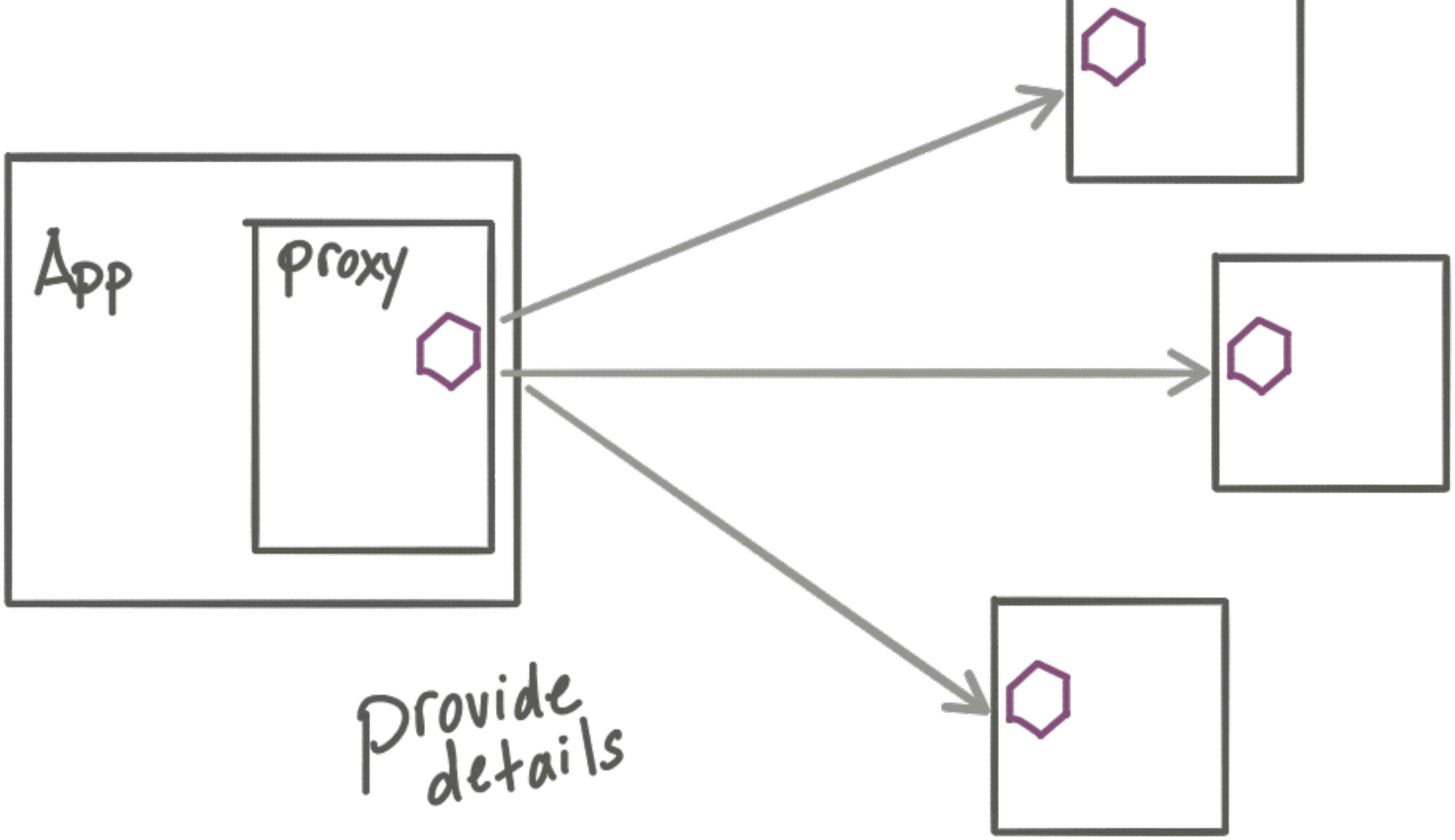
Network Namespace

Container

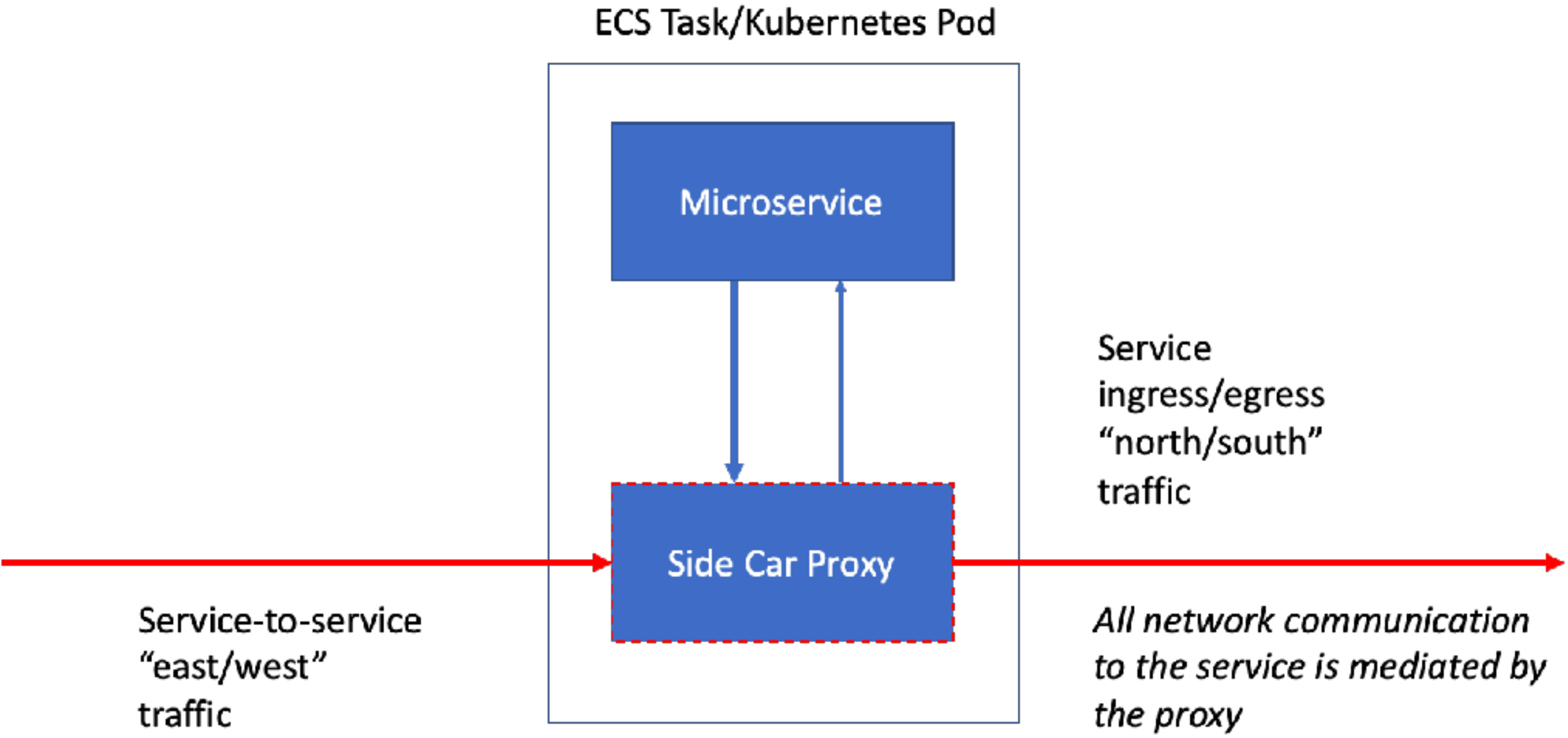
Container

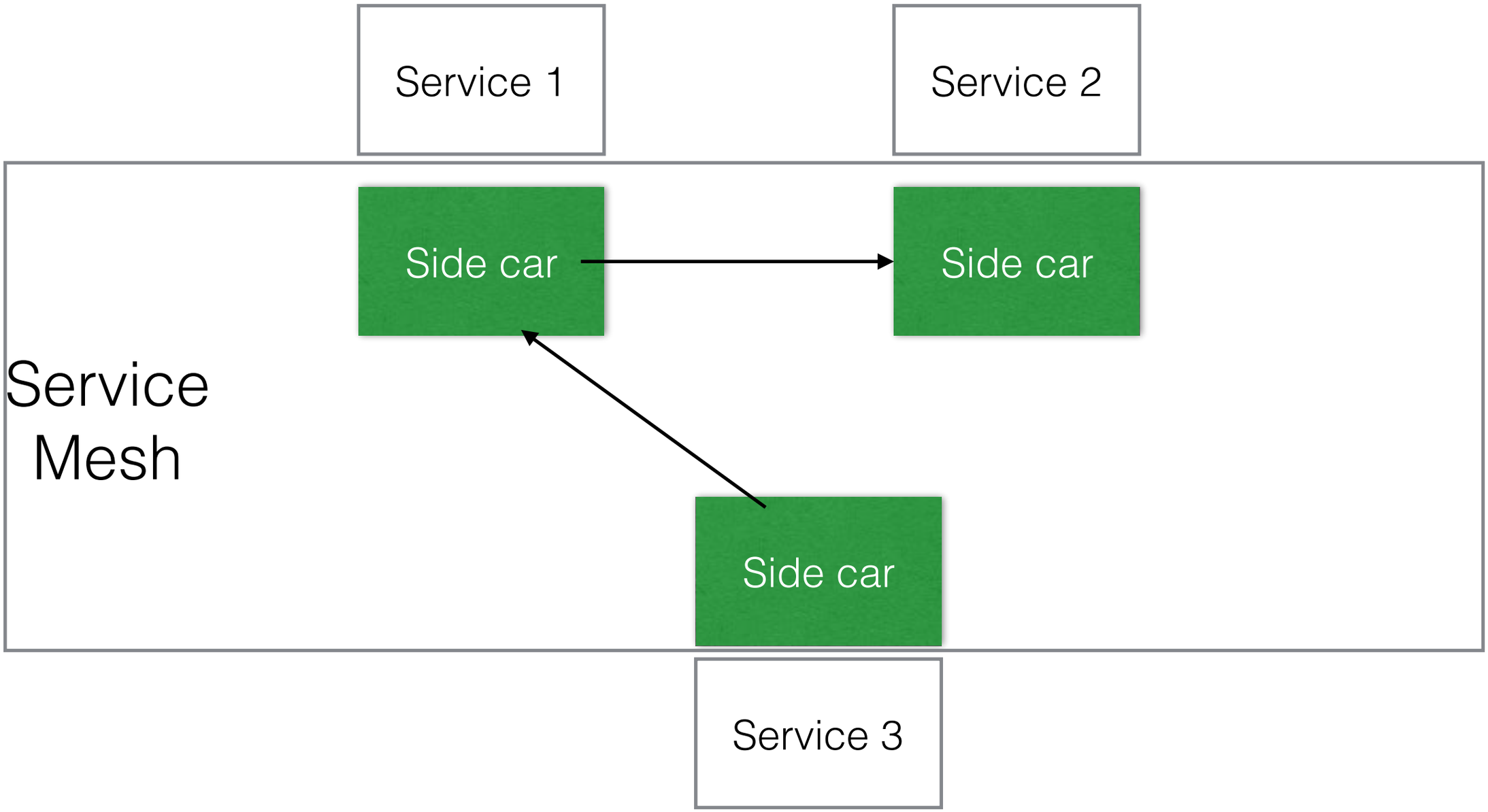
100.68.1.xxx

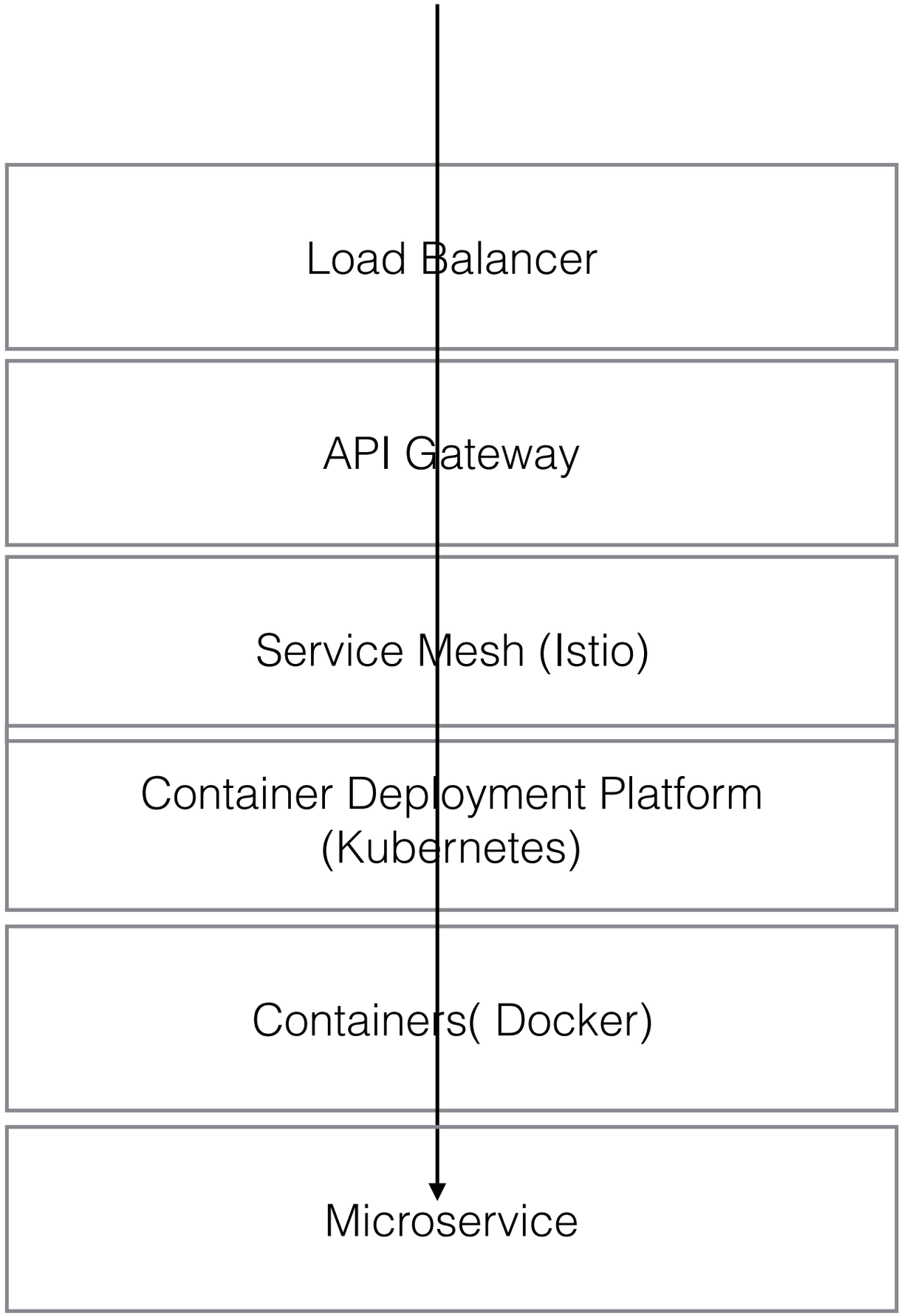




Service Mesh


ECS Task/Kubernetes Pod
Microservice
Side Car Proxy
Service
ingress/egress
“north/south”
traffic
Service-to-service
“east/west”
traffic
All network communication
to the service is mediated by
the proxy

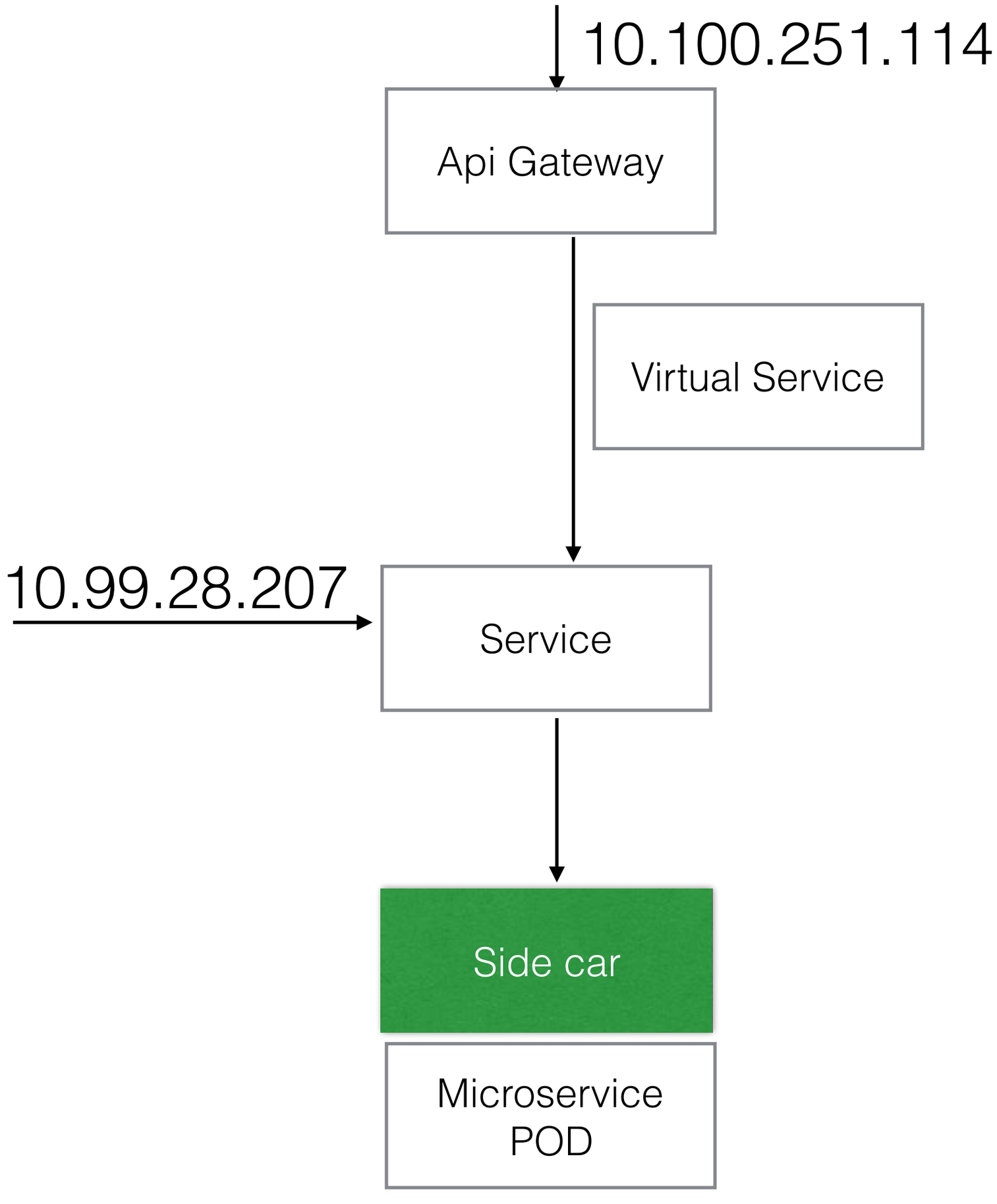

Service 1
Service 2
Side car
Side car
Service
Mesh
Side car
Service 3

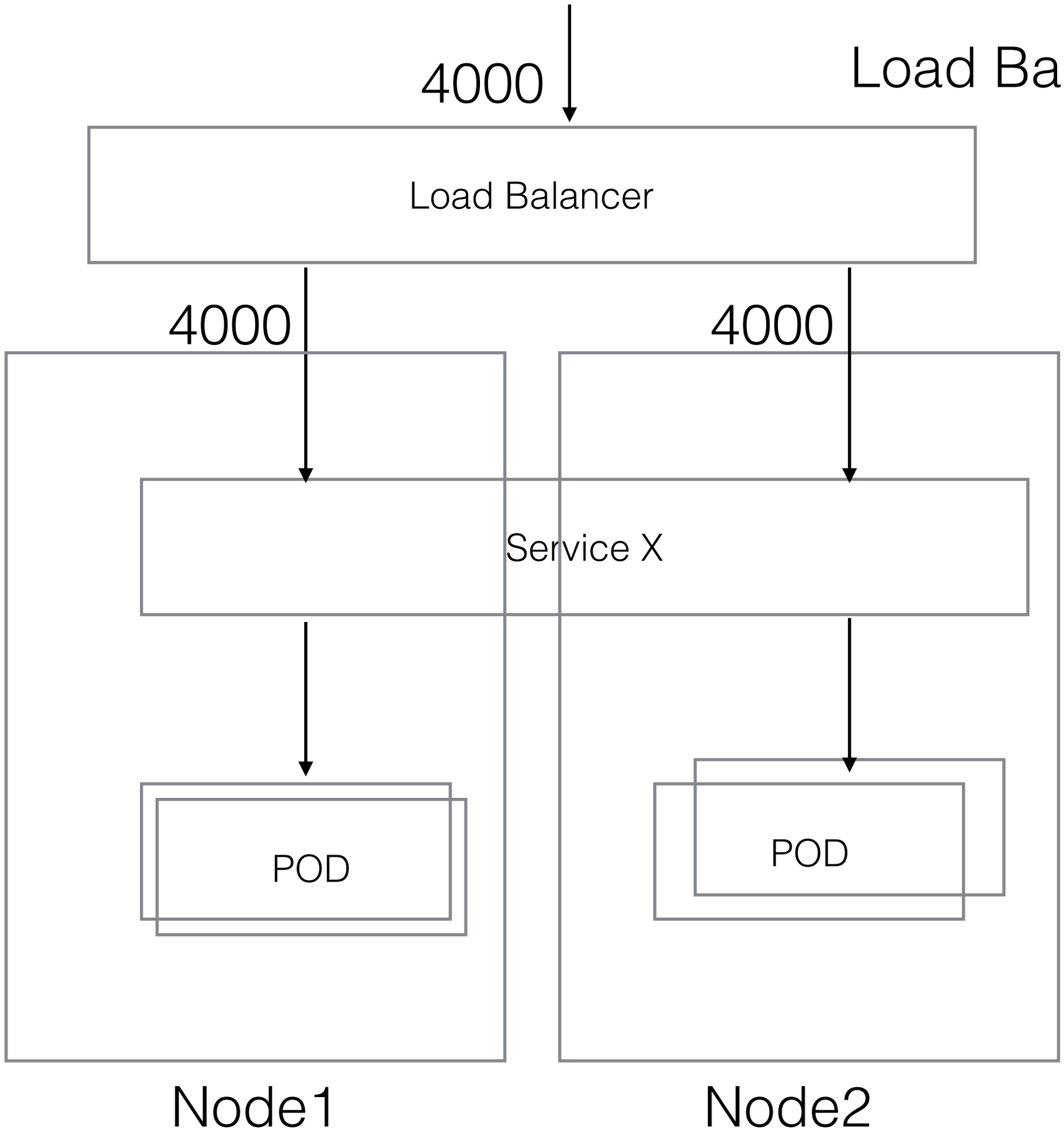


L4
North south

L7
North south

L7
East West


10.100.251.114
Api Gateway
Virtual Service
10.99.28.207
Service
Side car
Microservice
POD


4000
Load Balancer
Load Balancer
4000
4000
Service X
POD
POD
Node1
Node2

4000

Load Balancer

Layer 4
(tcp)

Cloud

4000

4000

Kubernetes
Cluster

API Gateway Service

Virtual Service

8000

Service X

Layer 7
(http)

Side Car

Side Car

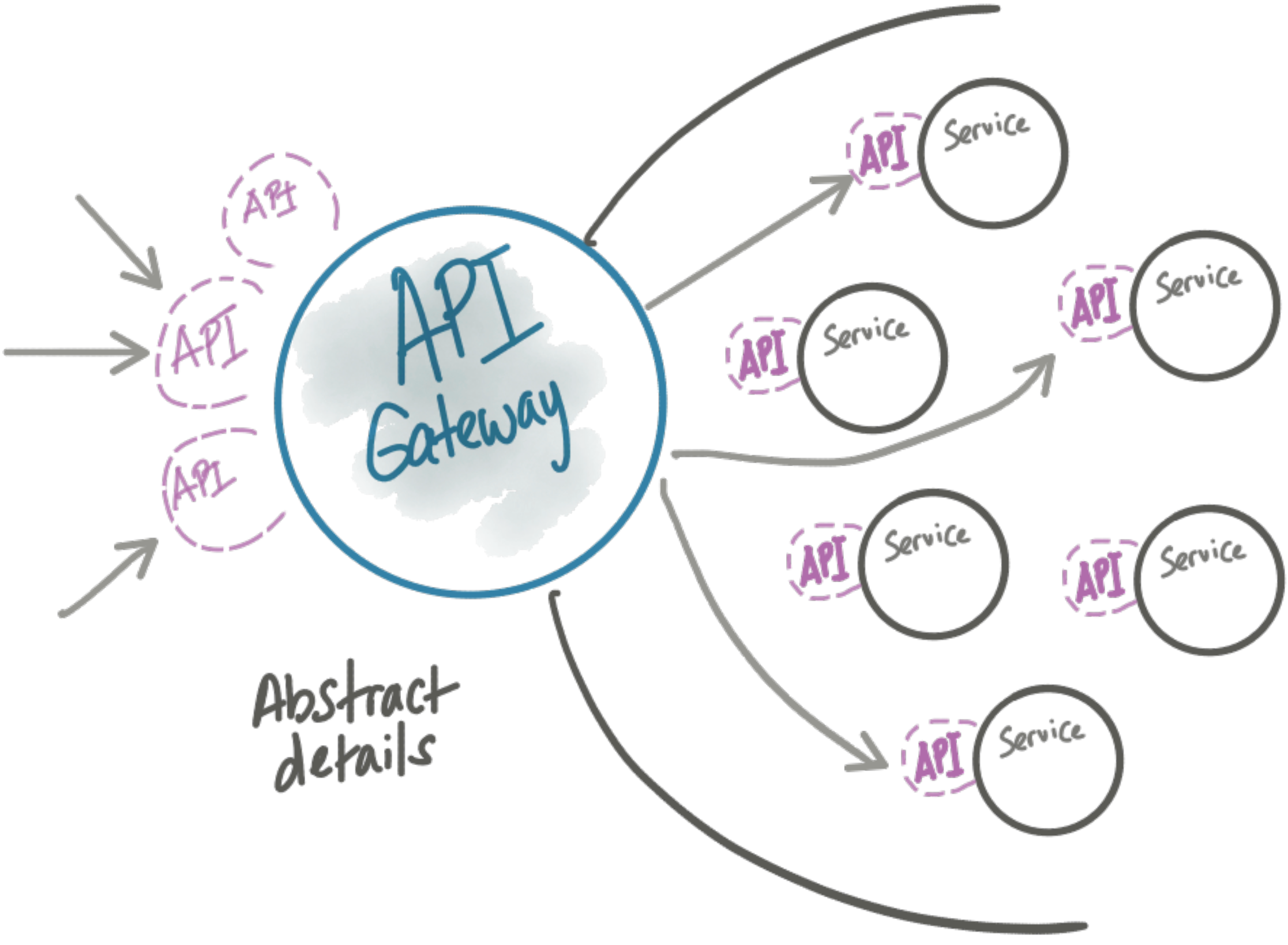
POD

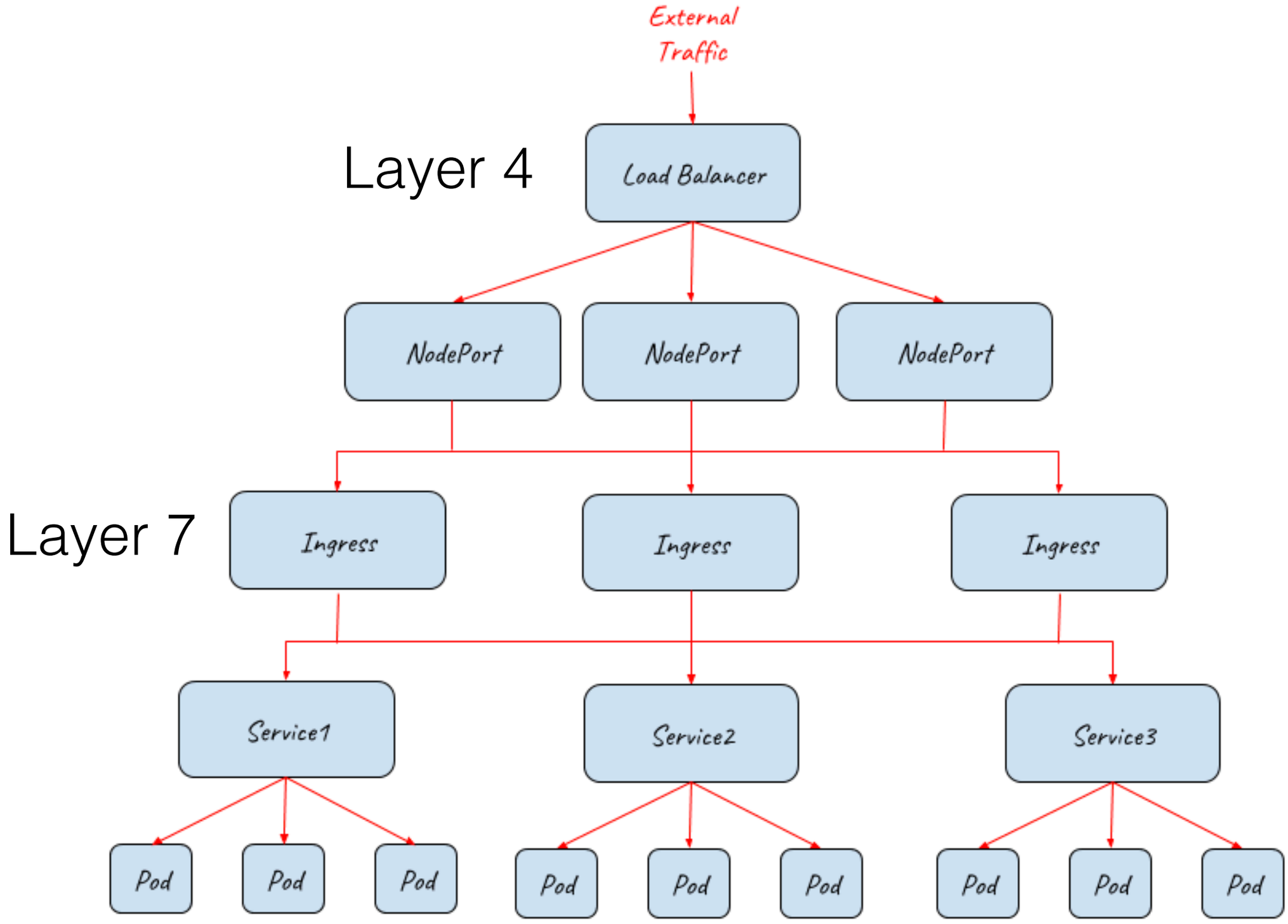
POD

Node1

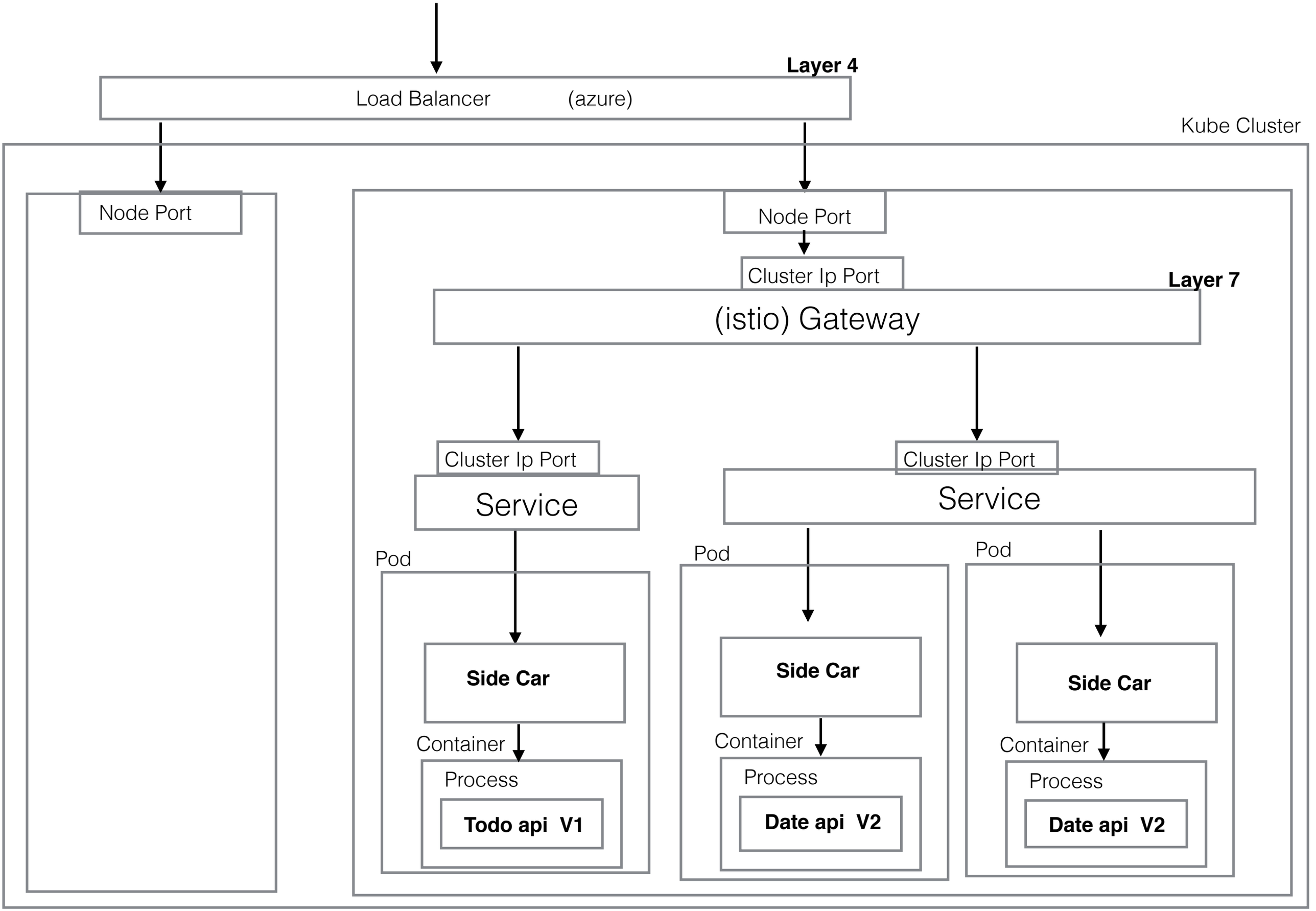
Node2

	Approach
Performance	
Transaction Management	
Joins / Reports	
Infra Cost	Container, Kubernetes
Deployment	Docker Image, Hub
Debugging (Finding bugs)	ServiceMesh
Log Mgmt	
Config Mgmt	
Authentication	
Authorization	
Monitoring / Alerting	Kubernetes , ServiceMesh


API
API
API
API Gateway
API
Service
API
Service
API
Service
API
Service
API
Service
API
Service
Abstract details



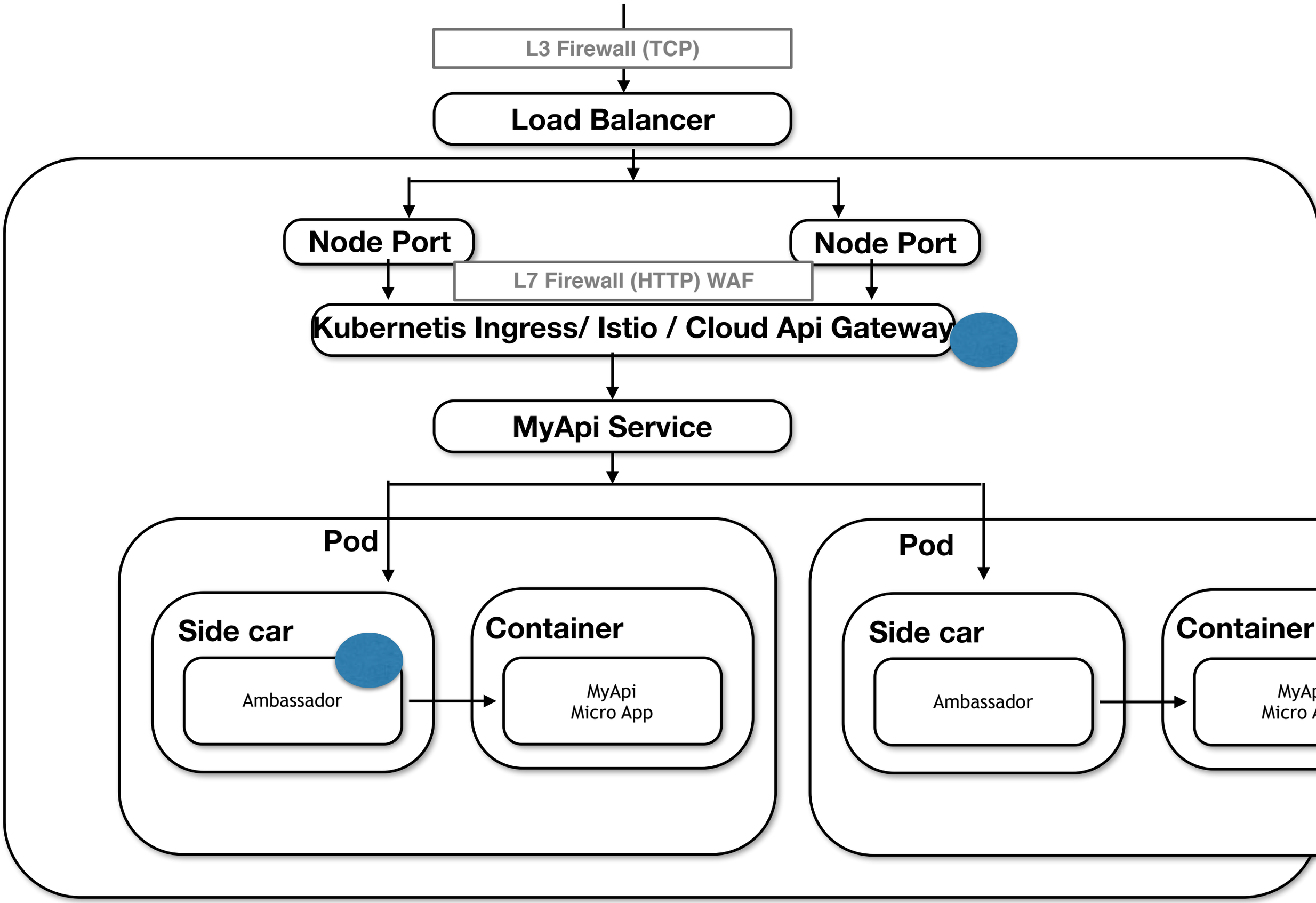
Each of the NodePort, Ingress or Pod layers can be scale out/in accordingly to handle different working loads.

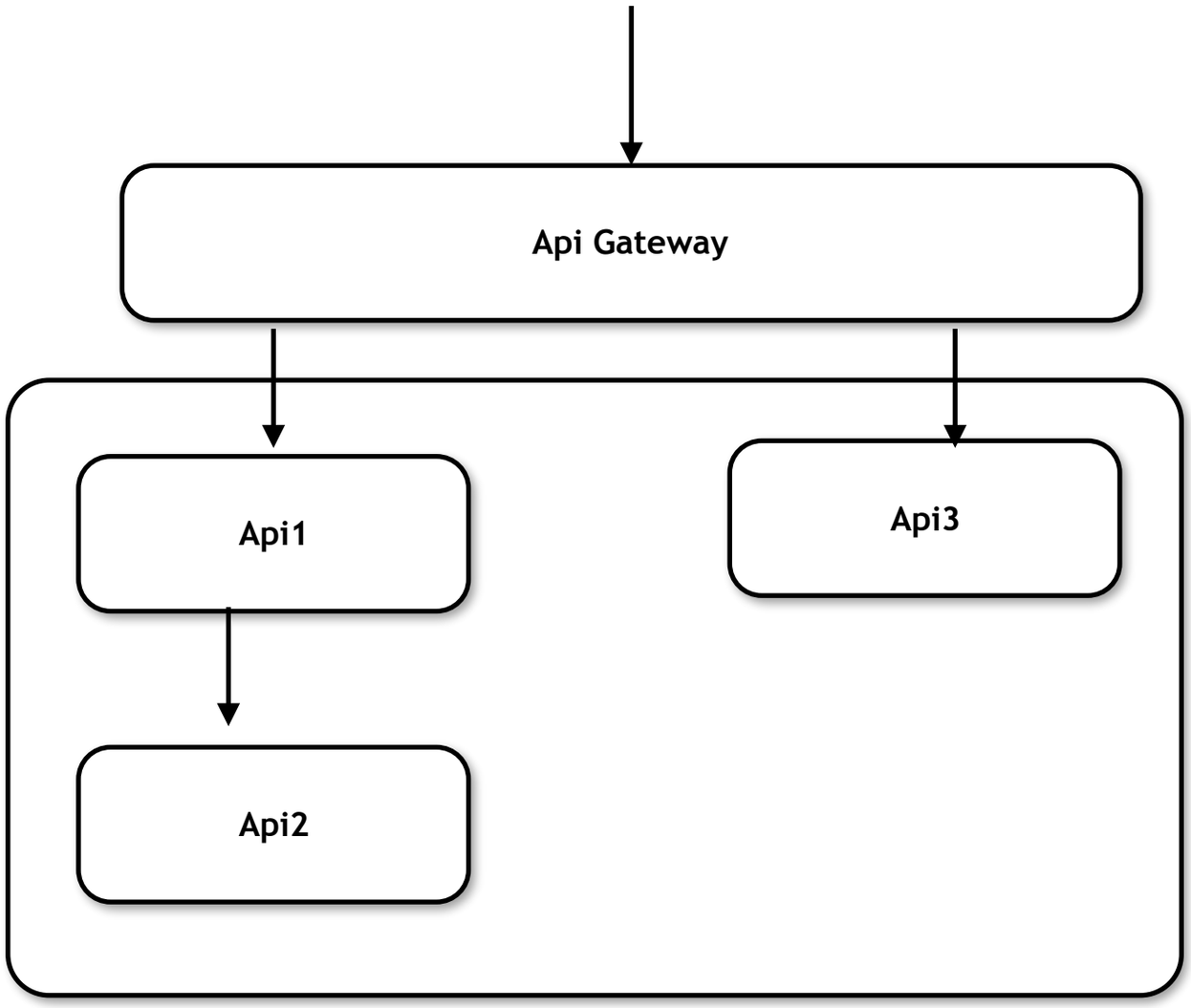

Layer 4
Load Balancer (azure)
Kube Cluster
Node Port
Node Port
Cluster Ip Port
Layer 7
(istio) Gateway
Cluster Ip Port
Service
Cluster Ip Port
Service
Pod
Side Car
Container
Process
Todo api V1
Pod
Side Car
Container
Process
Date api V2
Pod
Side Car
Container
Process
Date api V2

Kubernetes Ingress v/s Istio Gateway v/s API Gateway?

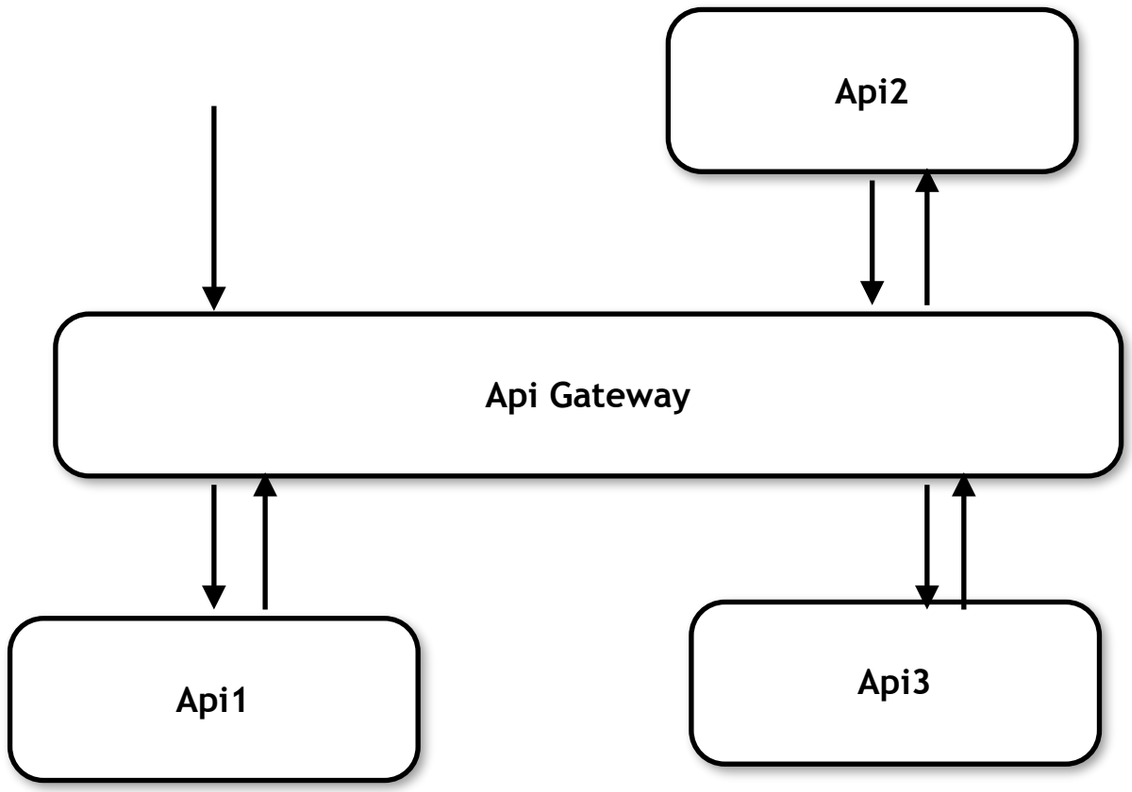
Given that it’s difficult to find an ideal out-of-box implementation which can provide both the functions of an application-layer API gateway and an Istio ingress gateway, a practical solution could be using a cascade of an API Gateway and a mesh sidecar proxy as the external traffic entrance.

Deployment Diagram



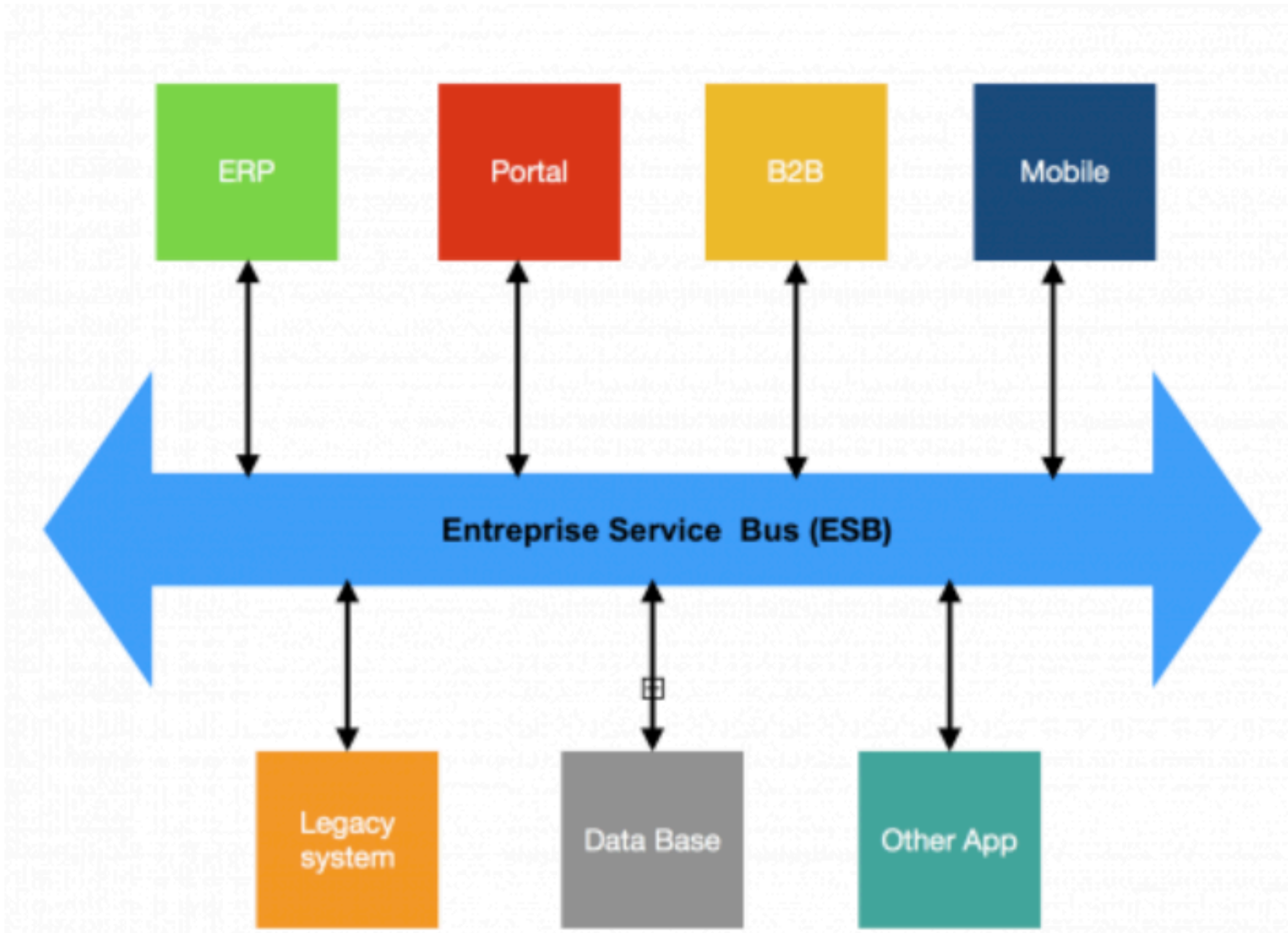


Facade



Bus

API Gateway vs ESB


ERP
Portal
B2B
Mobile
Entreprise Service Bus (ESB)
Legacy system
Data Base
Other App

Api Gateway
vs
Service Mesh

Load Balancer

API Gateway (Kubernetes Ingress, Istio Ingress, aws)

Service Mesh (Istio)

Deployment Platform (Kubernetes)

Edge Ingres

Service Mesh Control Plane

Service Mesh data plane

Integration

Integration

Integration

Integration

Integration

North-South Traffic

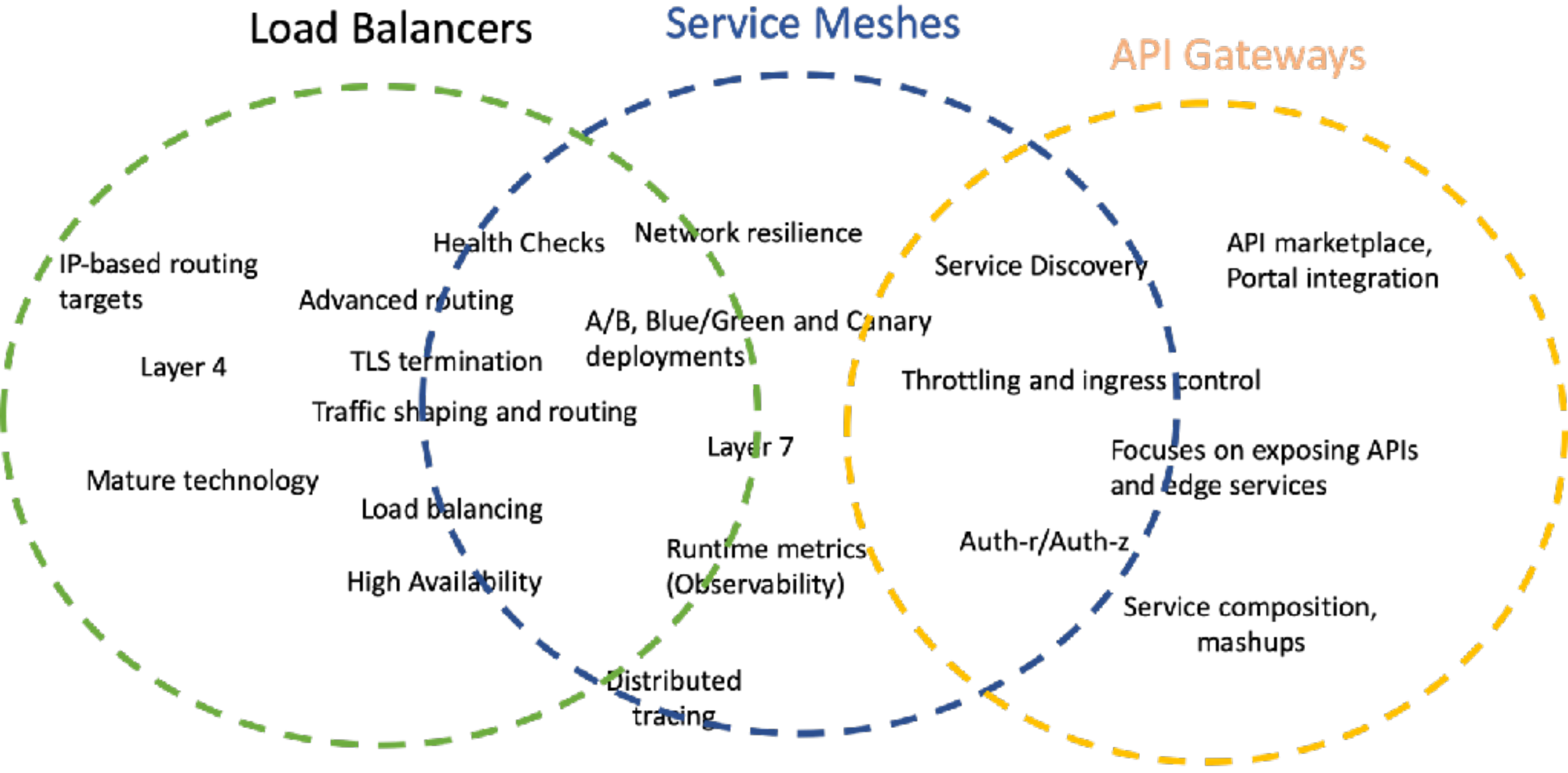
East-West traffic

Istio

API Gateway: Abstraction, decoupling, edge routing / security

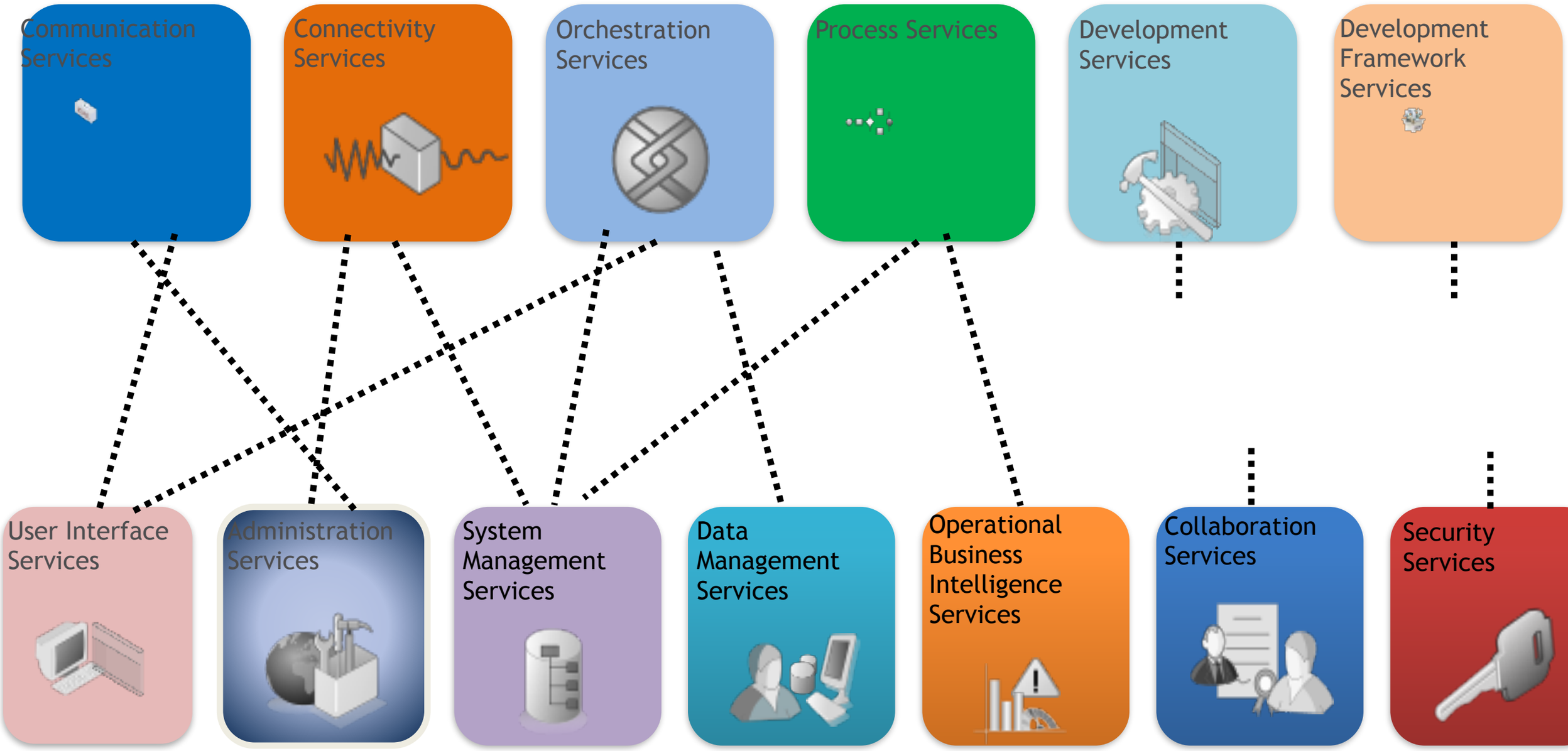
Service Mesh: endpoints, hosts, ports, metric collection, traffic routing, security

Deployment Platform: Cluster nodes, container schedule, resource management


Load Balancers
Service Meshes
API Gateways
IP-based routing targets
Layer 4
Mature technology
Load balancing
High Availability
Health Checks
Network resilience
Advanced routing
TLS termination
Traffic shaping and routing
A/B, Blue/Green and Canary deployments
Layer 7
Runtime metrics (Observability)
Distributed tracing
Service Discovery
Throttling and ingress control
Auth-r/Auth-z
API marketplace, Portal integration
Focuses on exposing APIs and edge services
Service composition, mashups

	Approach
Performance	
Transaction Management	
Joins / Reports	
Infra Cost	Container, Kubernetes, API Gateway
Deployment	Docker Image, Hub
Debugging (Finding bugs)	ServiceMesh
Log Mgmt	
Config Mgmt	
Authentication	API Gateway
Authorization	
Monitoring / Alerting	Kubernetes , ServiceMesh

Event Driven Architecture


Communication Services
Connectivity Services
Orchestration Services
Process Services
Development Services
Development Framework Services
User Interface Services
Administration Services
System Management Services
Data Management Services
Operational Business Intelligence Services
Collaboration Services
Security Services

Communication Services

Connectivity Services

Orchestration Services

Process Services

Development Services

Development Framework Services

Enterprise Service Bus (ESB)

User Interface Services

Administration Services

System Management Services

Data Management Services

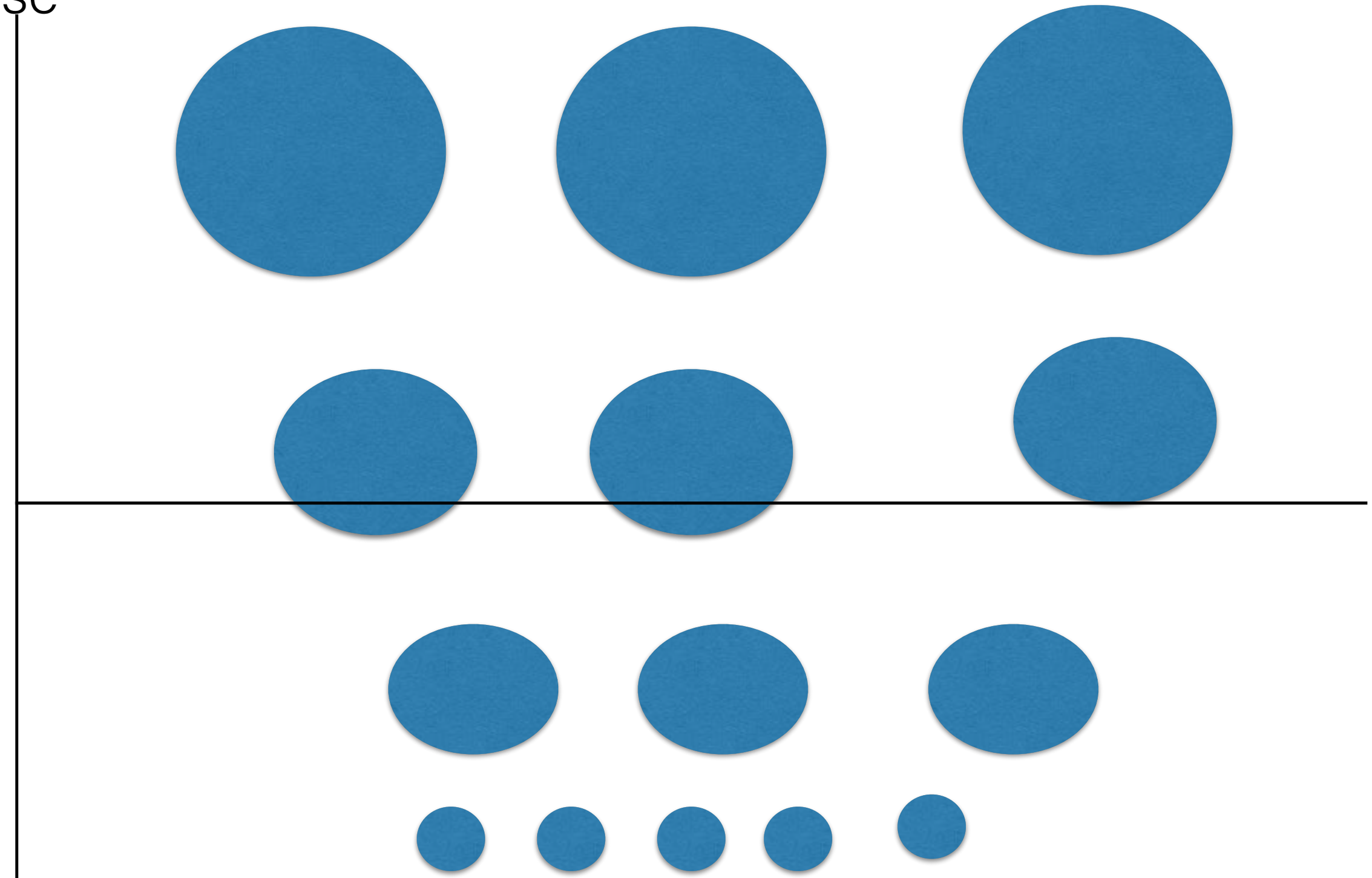
Operational Business Intelligence Services

Collaboration Services

Security Services

Case Study

Coarse

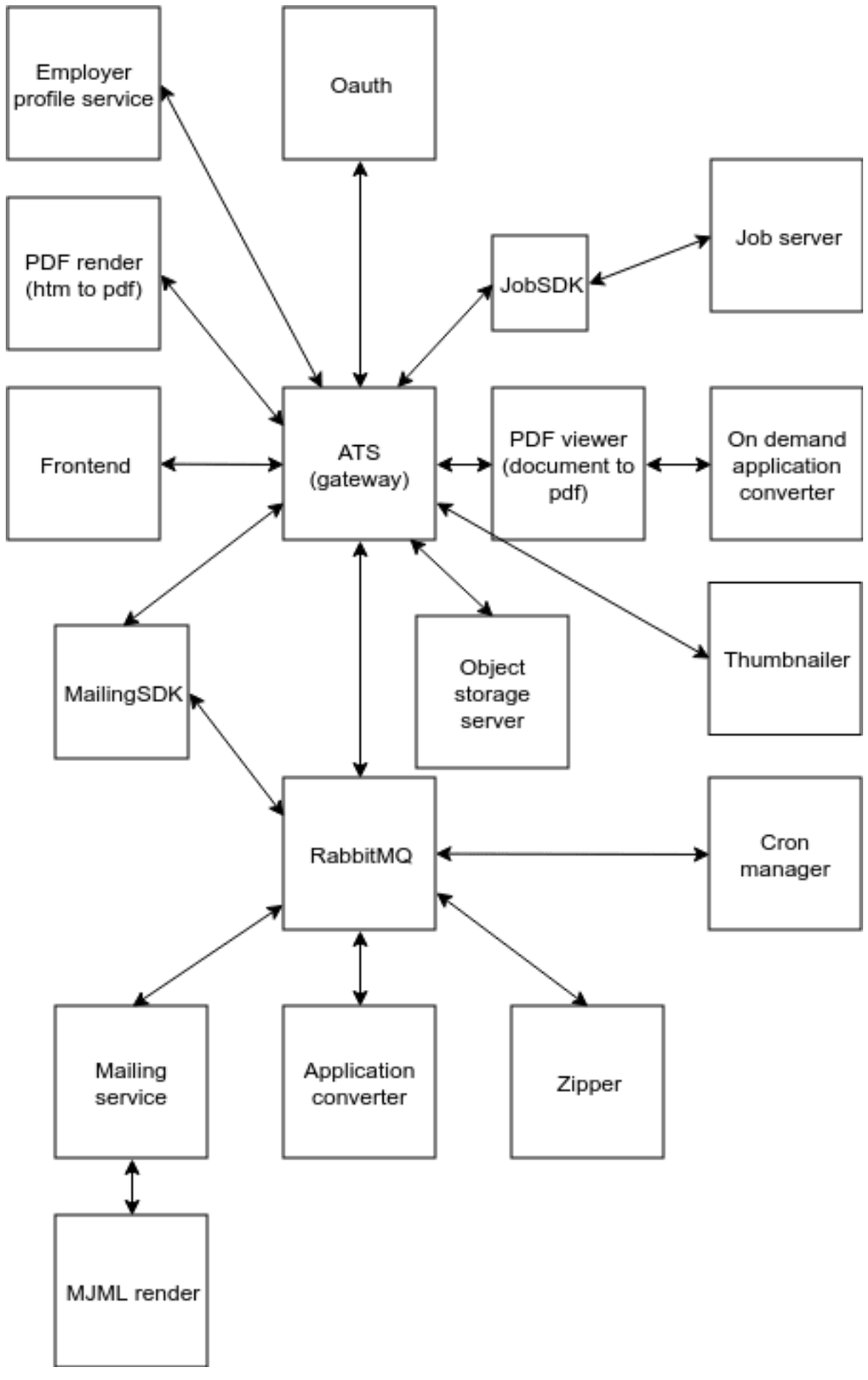


Fine

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.

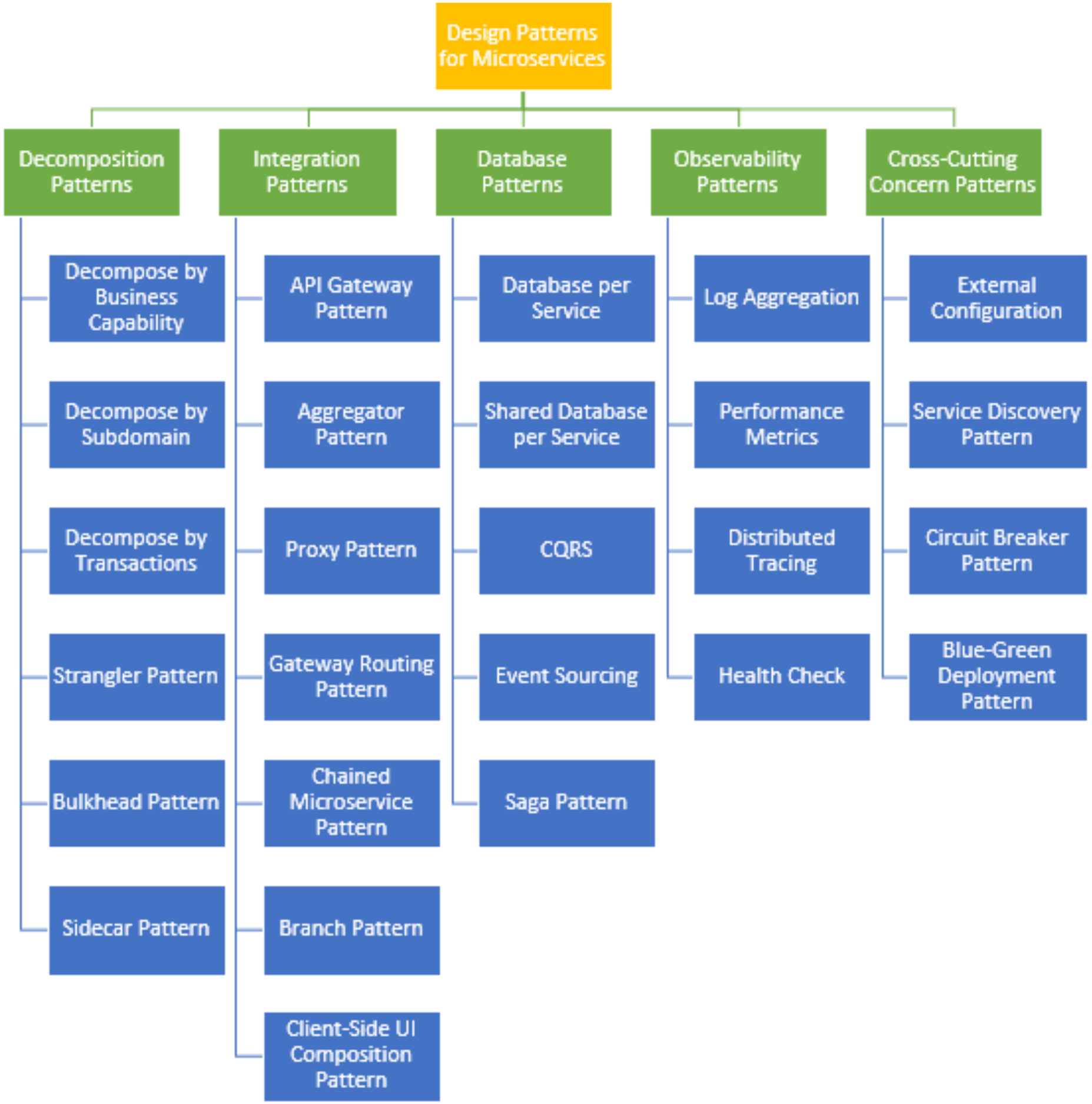
- One Year
- ~50,000 lines of code
- ~2,600 commits
- a team of 3 developers
- consisting of a senior full stack developer and two medior backend developers
- 18 services
- 200 tests


Employer profile service
Oauth
PDF render (htm to pdf)
JobSDK
Job server
Frontend
ATS (gateway)
PDF viewer (document to pdf)
On demand application converter
MailingSDK
Object storage server
Thumbnailer
RabbitMQ
Cron manager
Mailing service
Application converter
Zipper
MJML render

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 is 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.


Design Patterns for Microservices
Decomposition Patterns
Integration Patterns
Database Patterns
Observability Patterns
Cross-Cutting Concern Patterns
Decompose by Business Capability
Decompose by Subdomain
Decompose by Transactions
Strangler Pattern
Bulkhead Pattern
Sidecar Pattern
API Gateway Pattern
Aggregator Pattern
Proxy Pattern
Gateway Routing Pattern
Chained Microservice Pattern
Branch Pattern
Client-Side UI Composition Pattern
Database per Service
Shared Database per Service
CQRS
Event Sourcing
Saga Pattern
Log Aggregation
Performance Metrics
Distributed Tracing
Health Check
External Configuration
Service Discovery Pattern
Circuit Breaker Pattern
Blue-Green Deployment Pattern

 # Event Bus
 # EDA
 # compensatale transaction
 # SAGA
 # Event Sourcing
 # CQRS

Load Balancer

Gateway

Inventory Api

CUD

Event

R

Grpc

Stream Analytics

Event Bus

Event

Event

Event

Audit Api

Inventory Cmd Api

Do

Undo

Inventory Query Api

Do

Undo

Data Lake

Event db

Full data

Data Ware House

snap shot data
- Weekly
- Monthly
- Yearly

PostgreSQL

rdbms

3rd normal form
removes duplicates
increases joins
write friendly (no duplicates)
no read friendly (joins)
current data

mongoDB

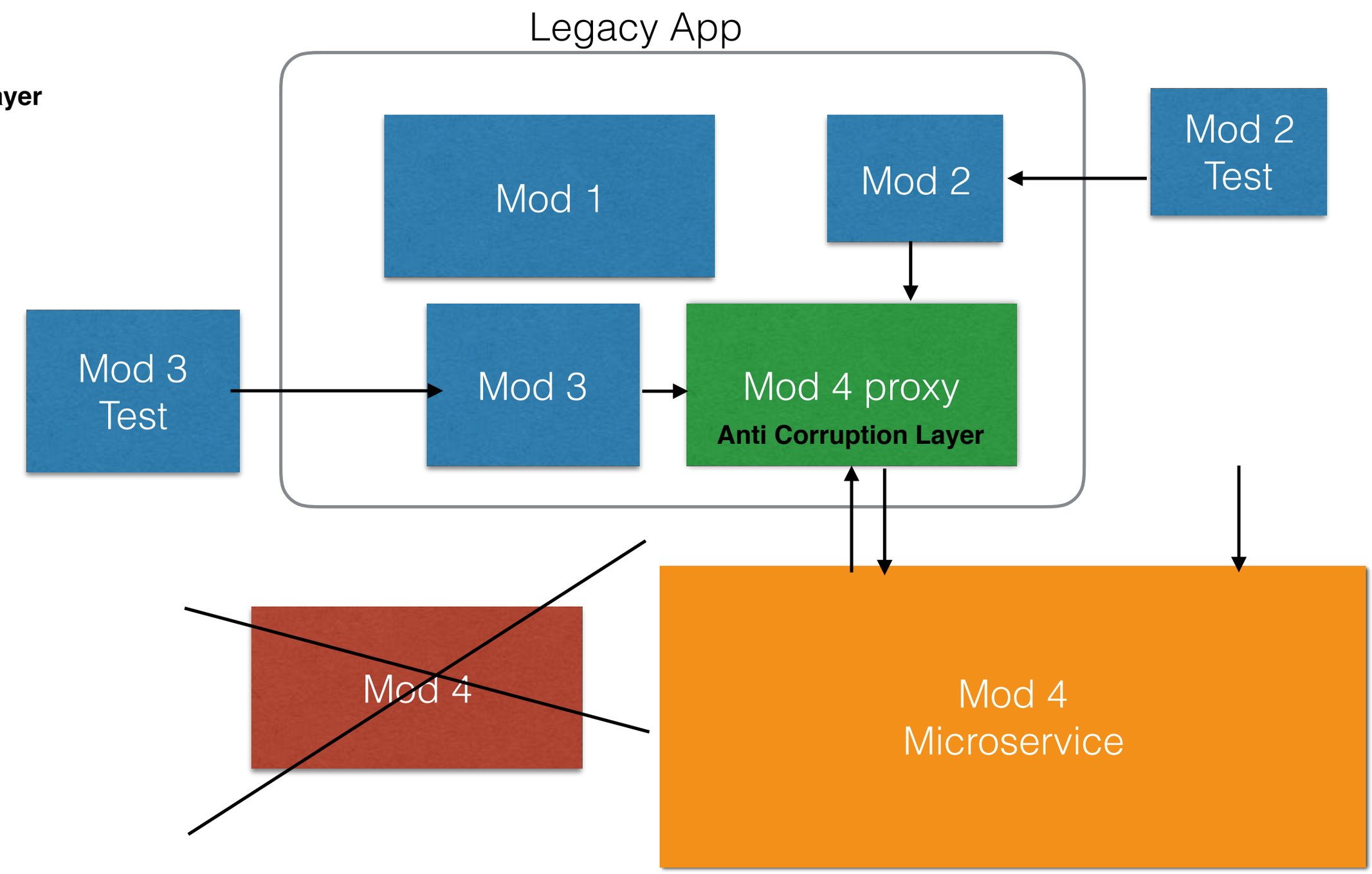
Document Db

de normalised form
increase duplicates
remove joins
not write friendly (duplicates)
read friendly (no joins)

Batch/Bigdata Analytics

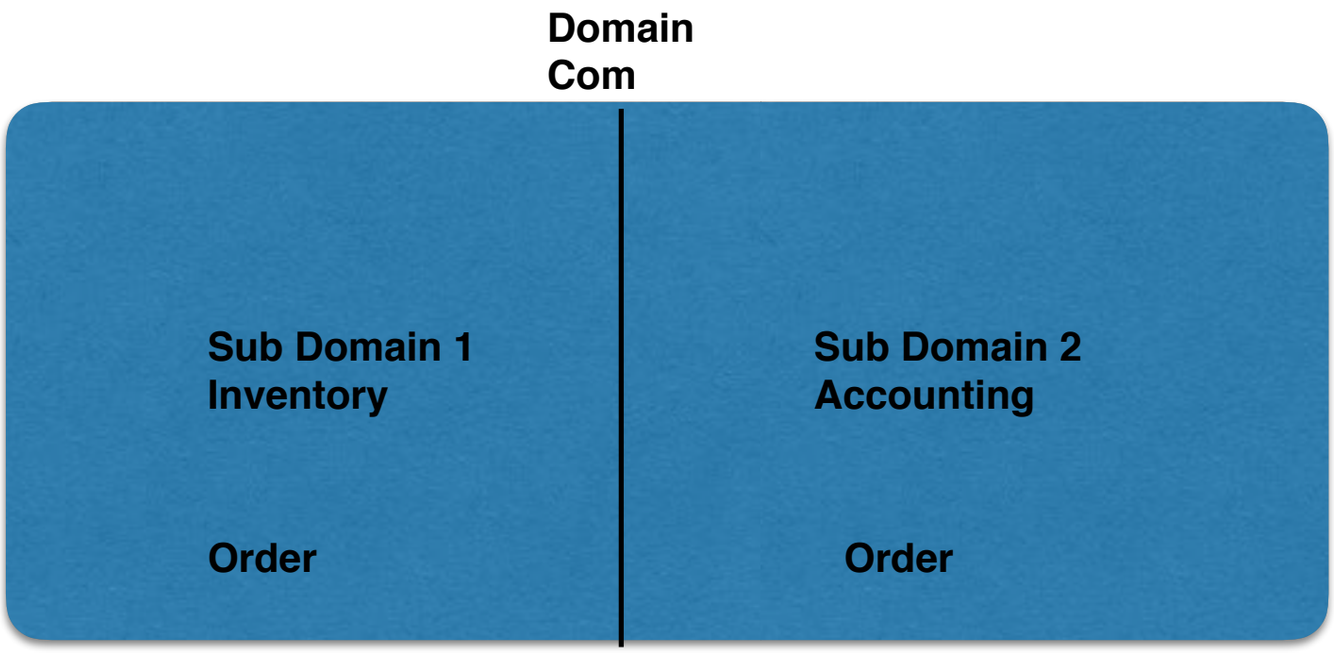
OLAP Analytics

 # unit test
 # Strangler Pattern
 # Anti Corruption Layer



*Transaction Boundary

Bounded Context (DDD)



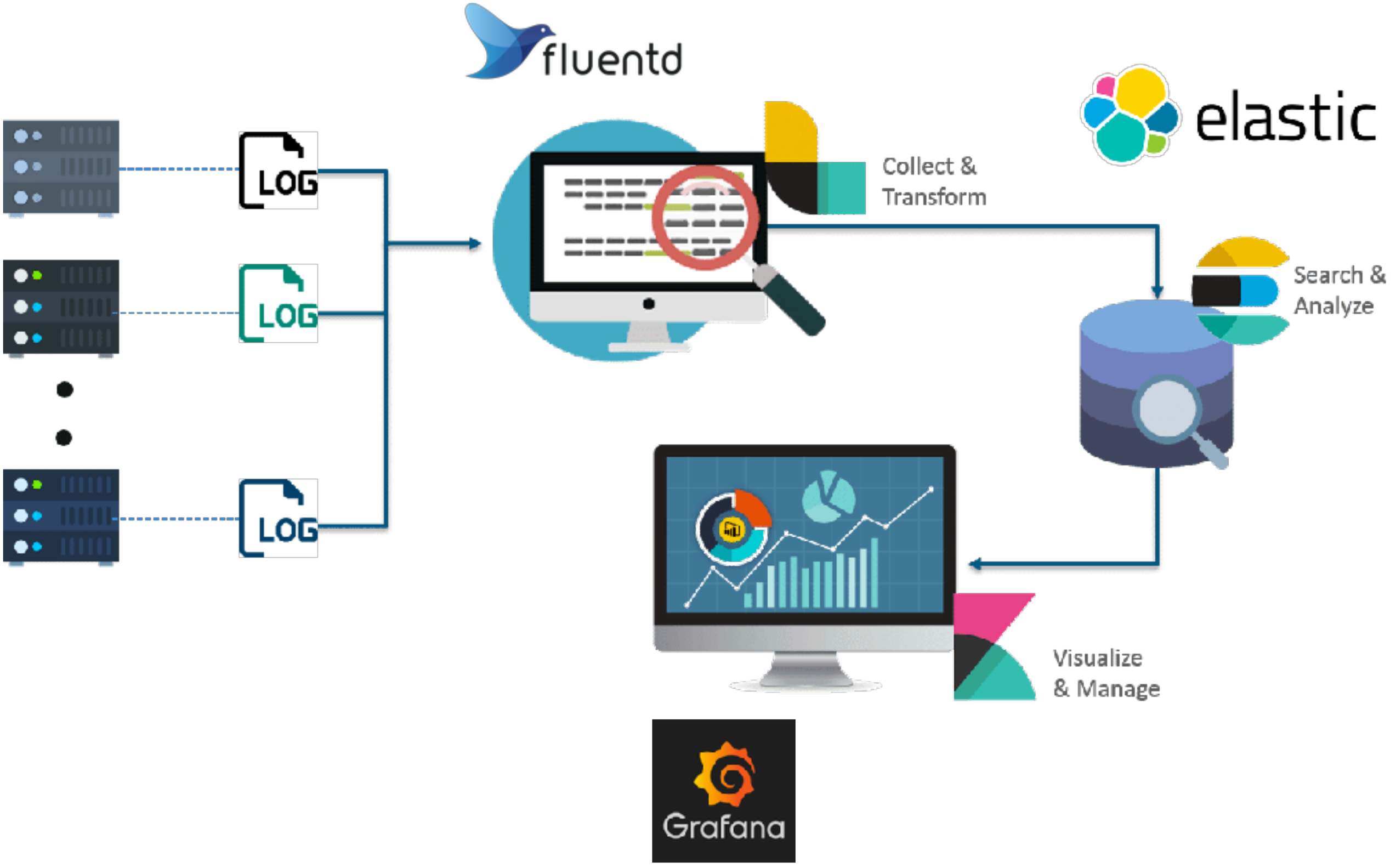
Aggregate Root (DDD)

Invoice -> Line Item
-> Address

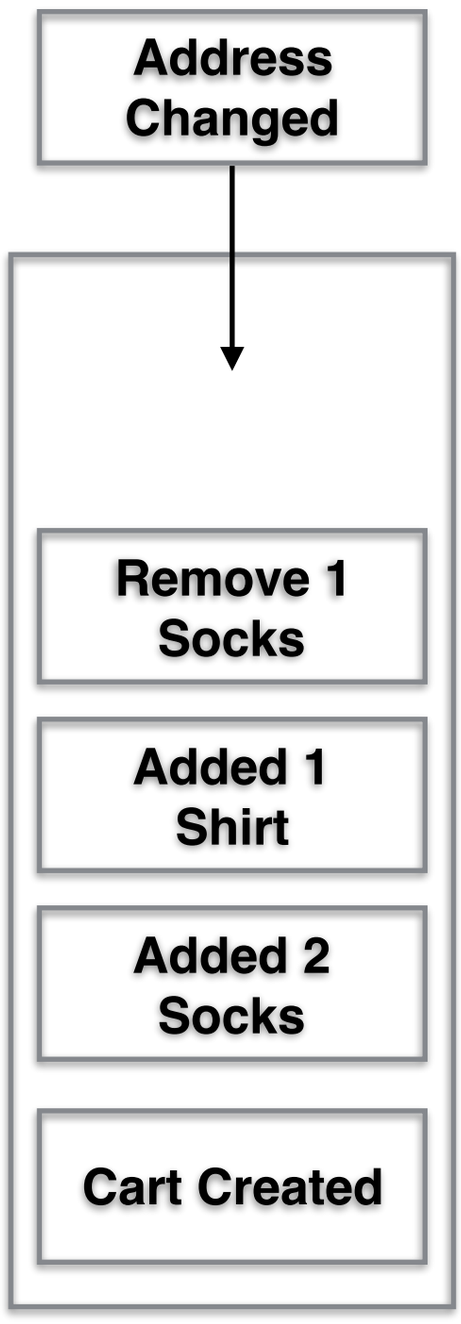
Feature

Patterns

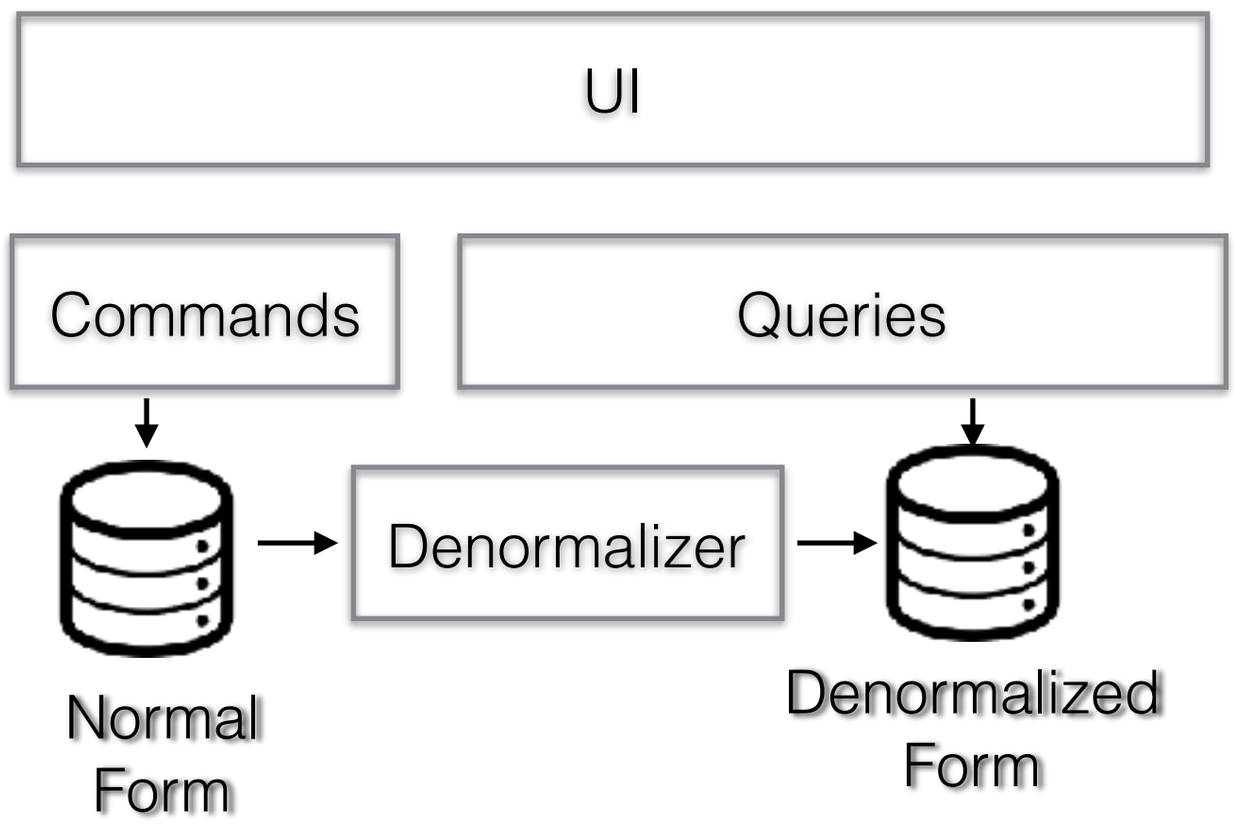
Log Aggregation



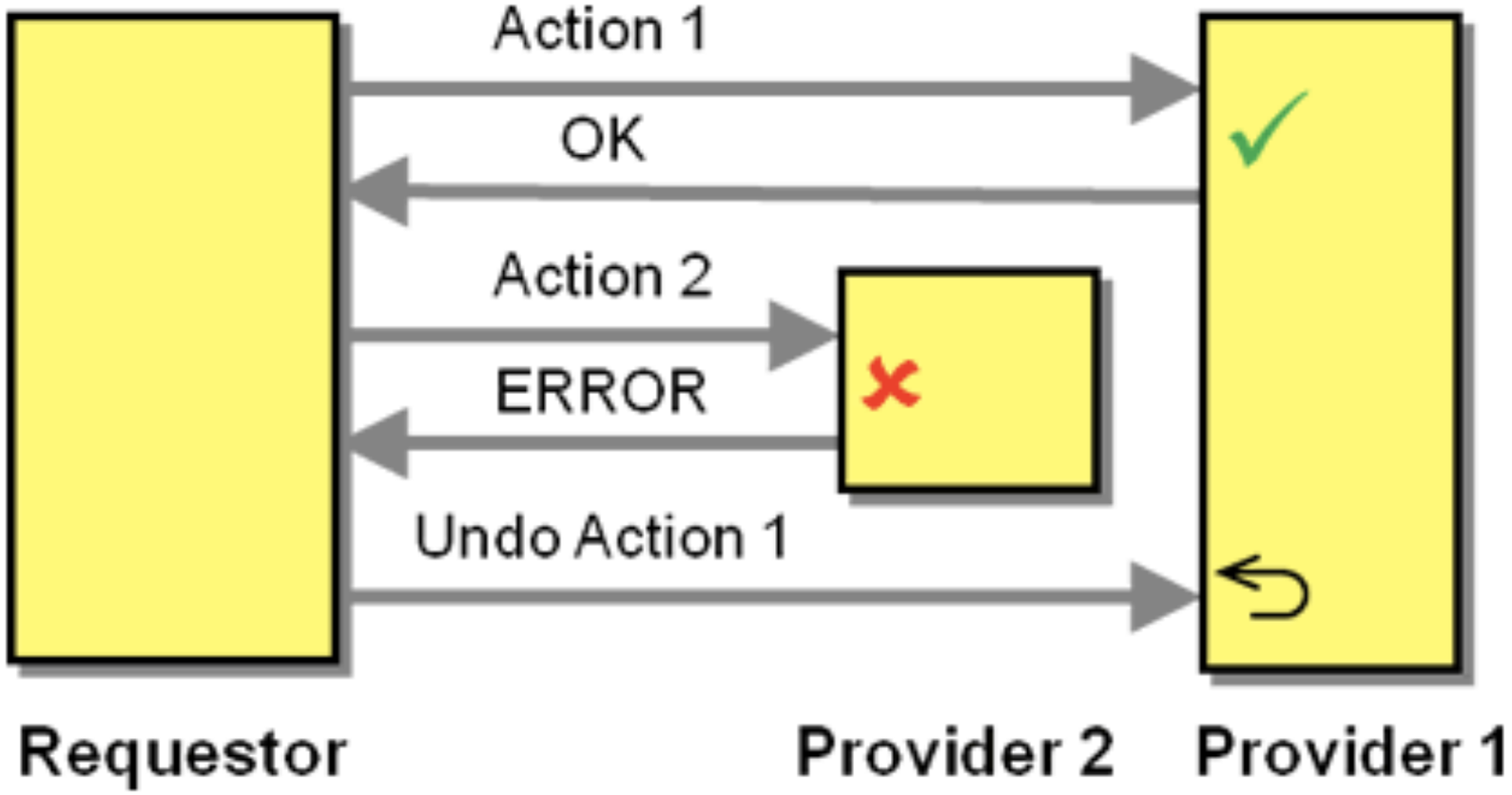
Event Source



CQRS

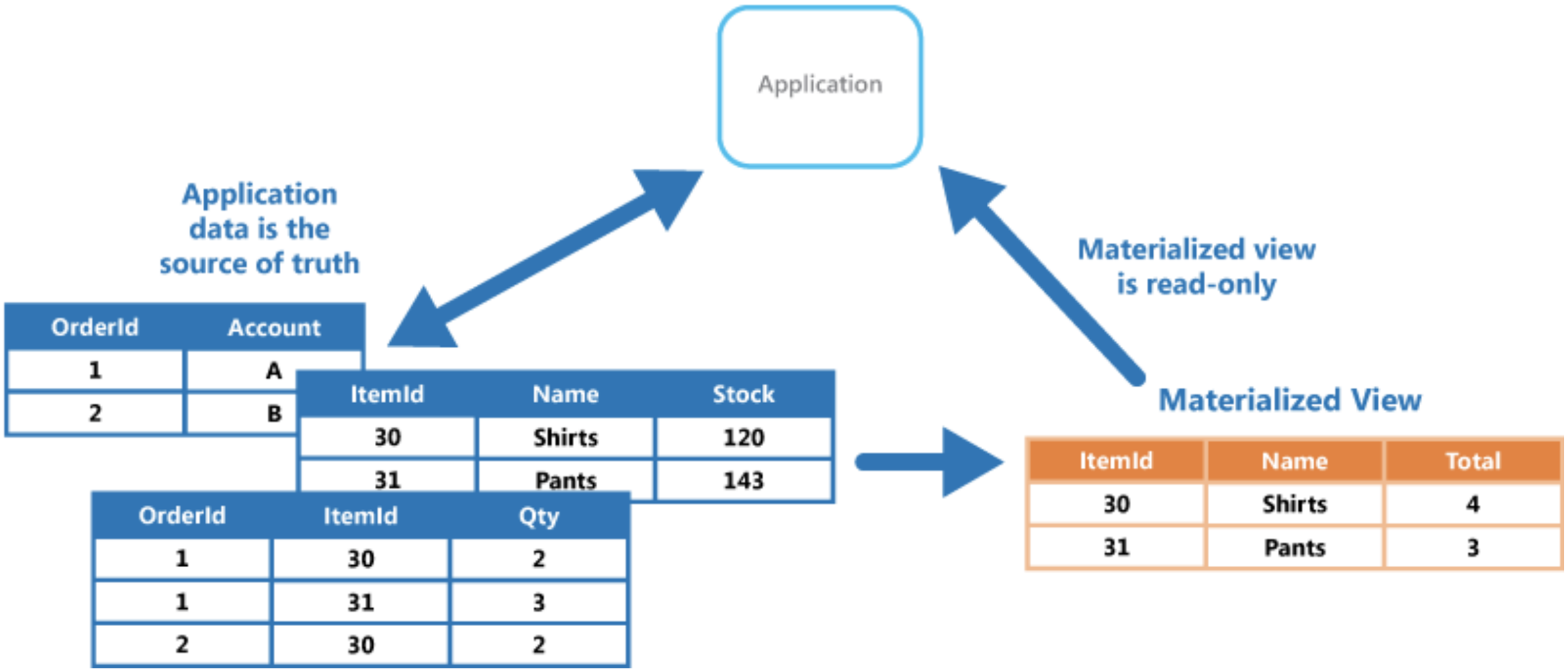

UI
Commands
Queries
Denormalizer
Normal Form
Denormalized Form

Saga



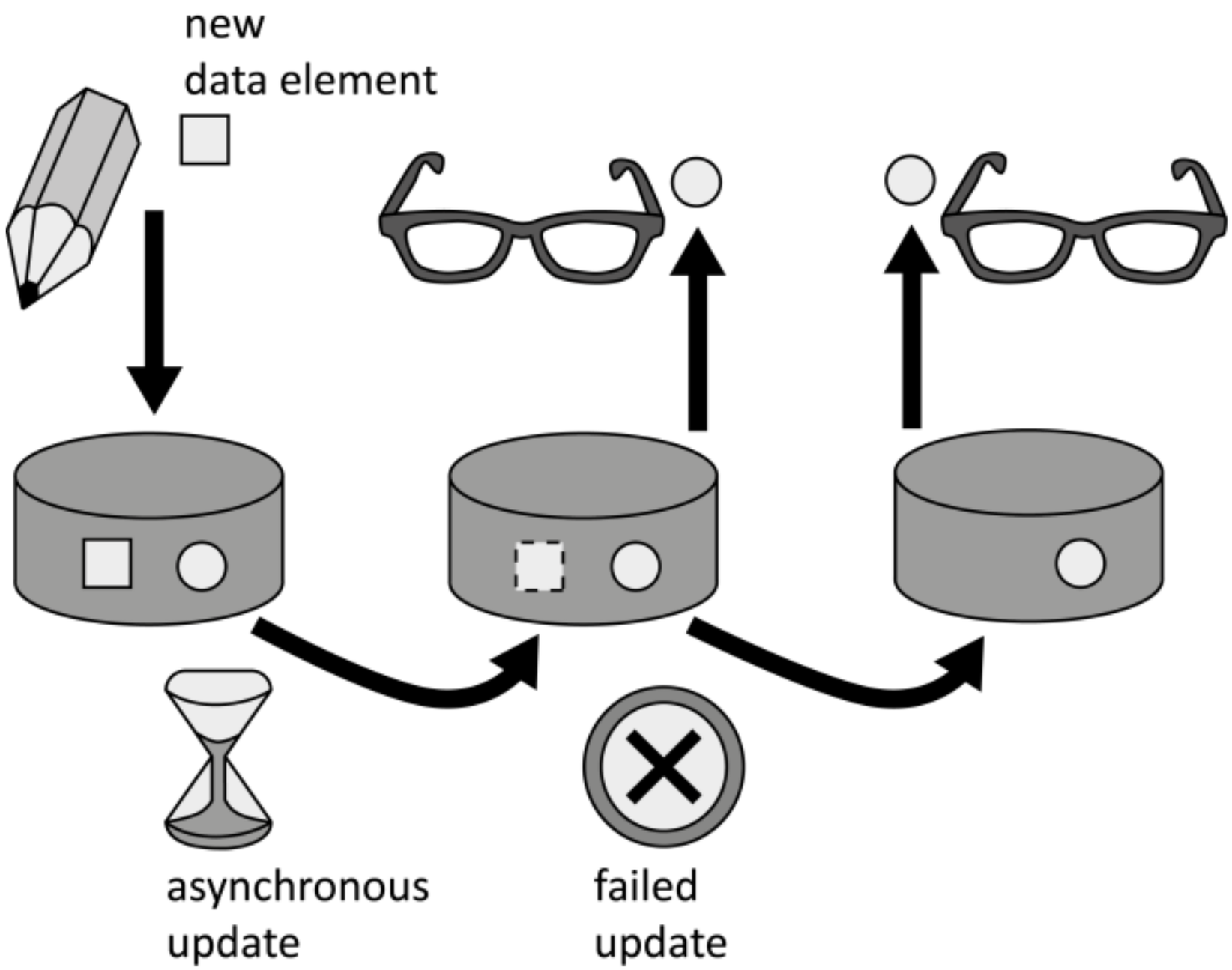
Saga Pattern is a direct solution to implementing distributed transactions in a microservices architecture.

Materialized View

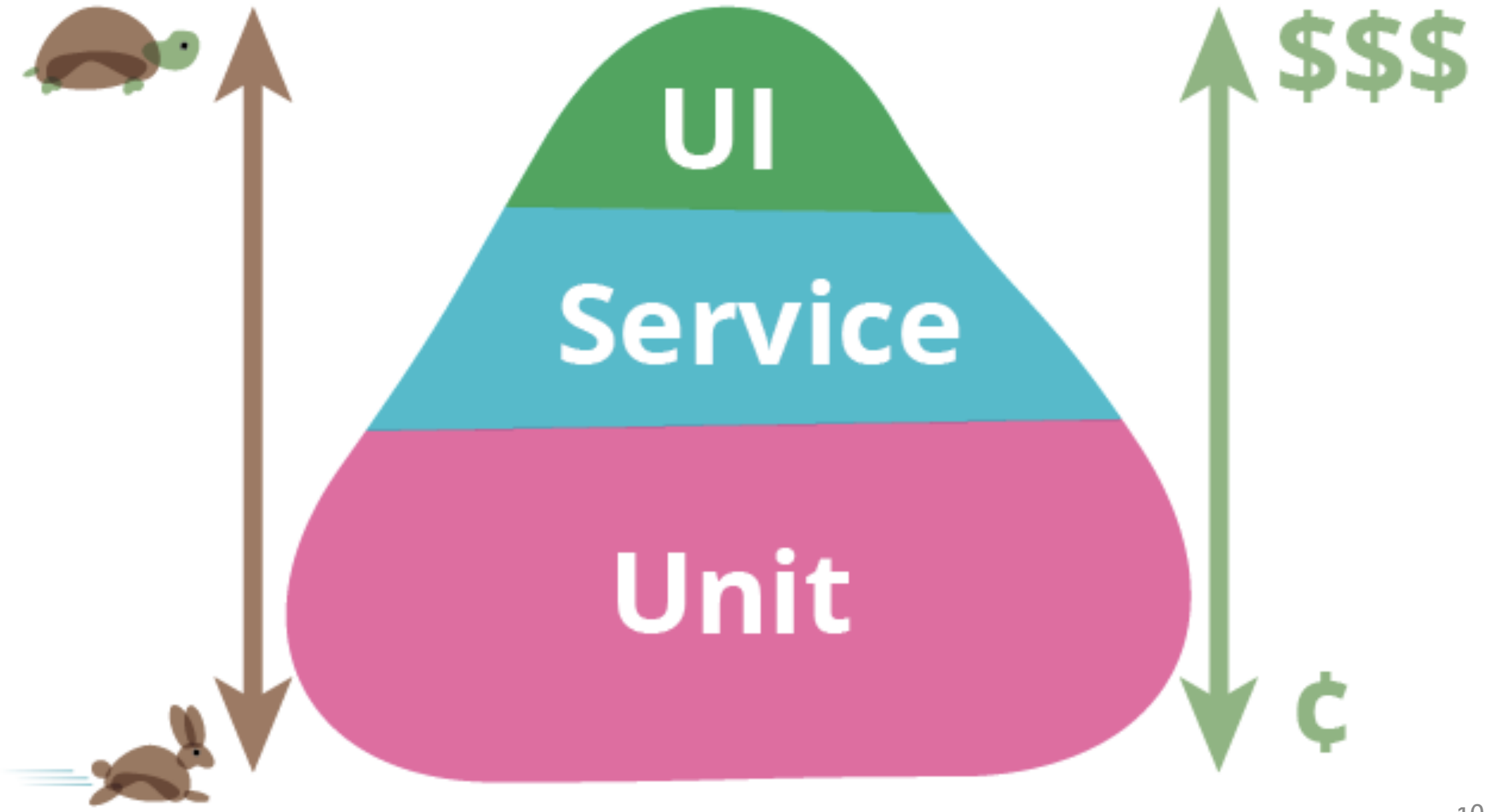


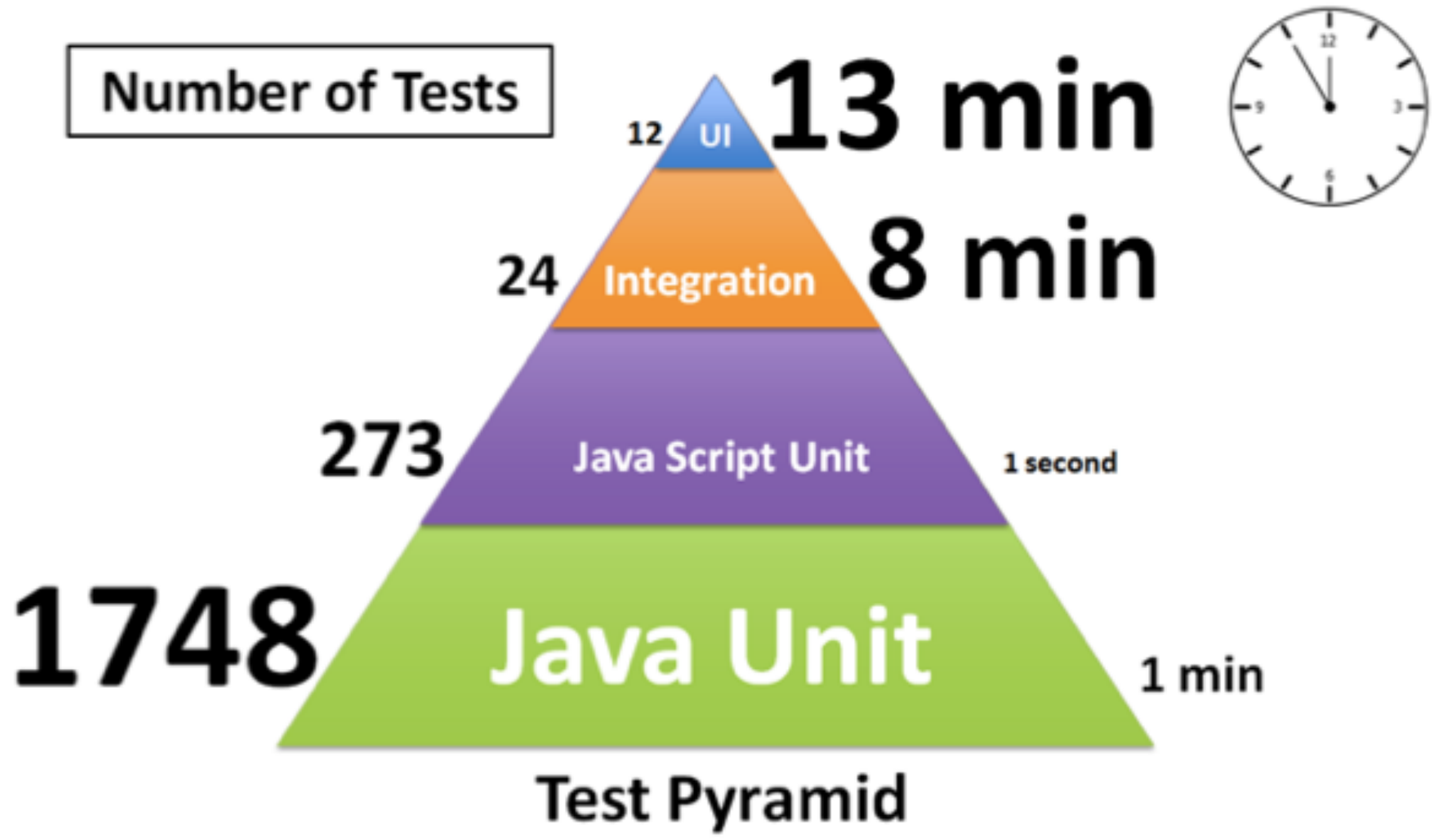
source:msdn

BASE

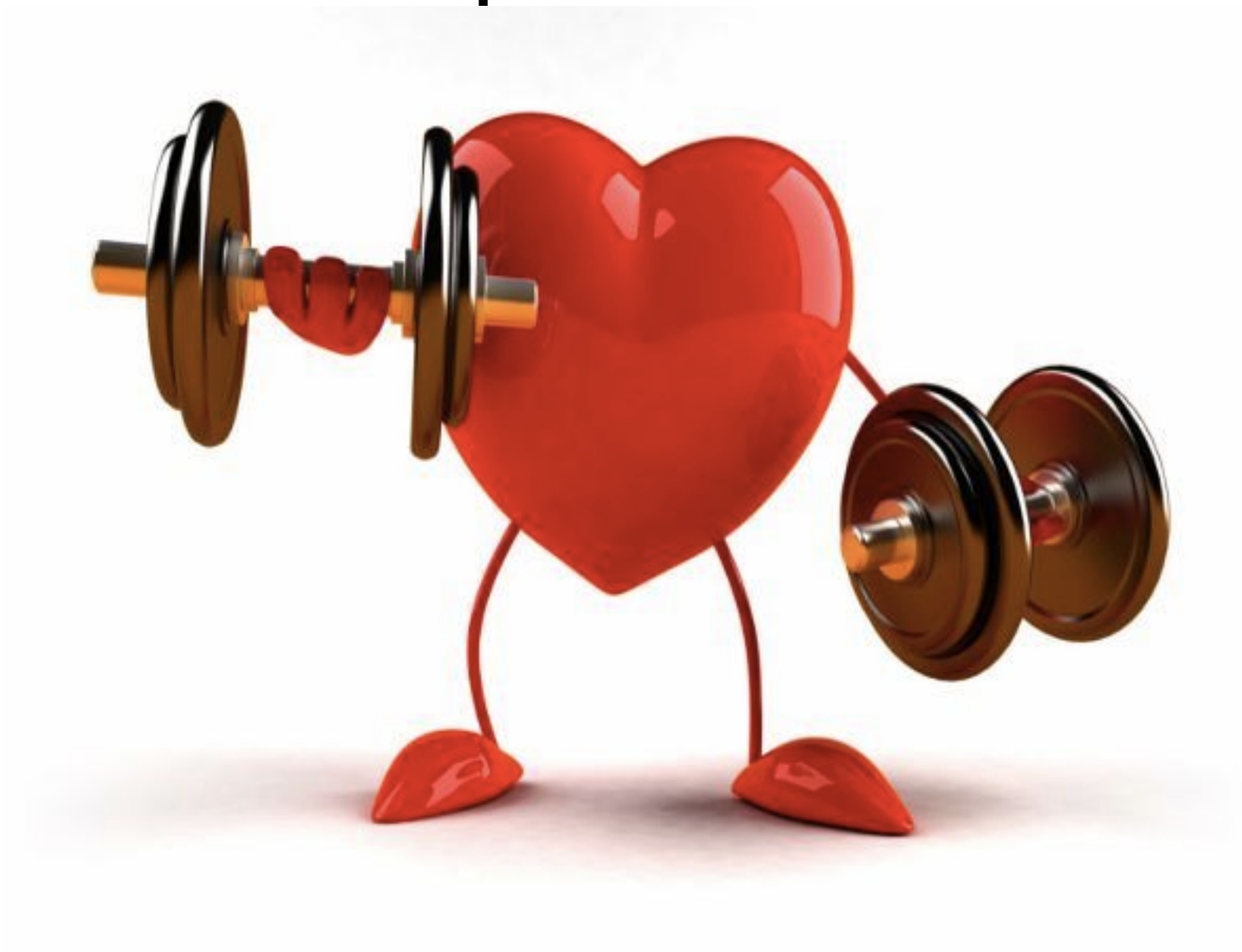


Test Pyramid



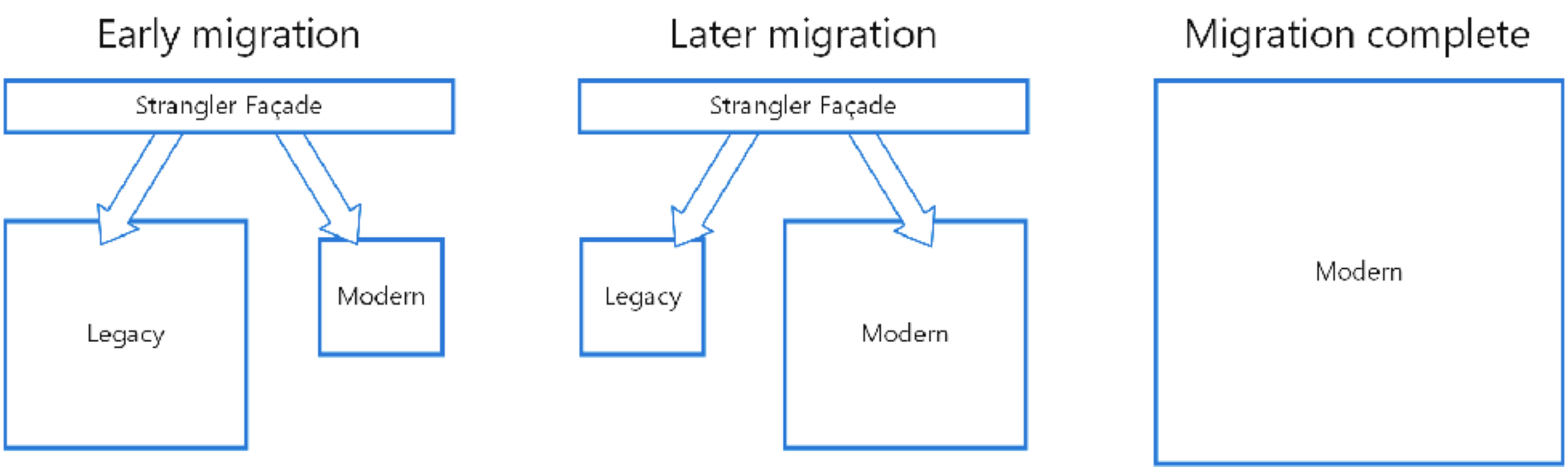

Number of Tests
12
UI
13 min
24
Integration
8 min
273
Java Script Unit
1 second
1748
Java Unit
1 min
Test Pyramid
www.fabiopereira.me
ThoughtWorks

Health Endpoint Monitoring



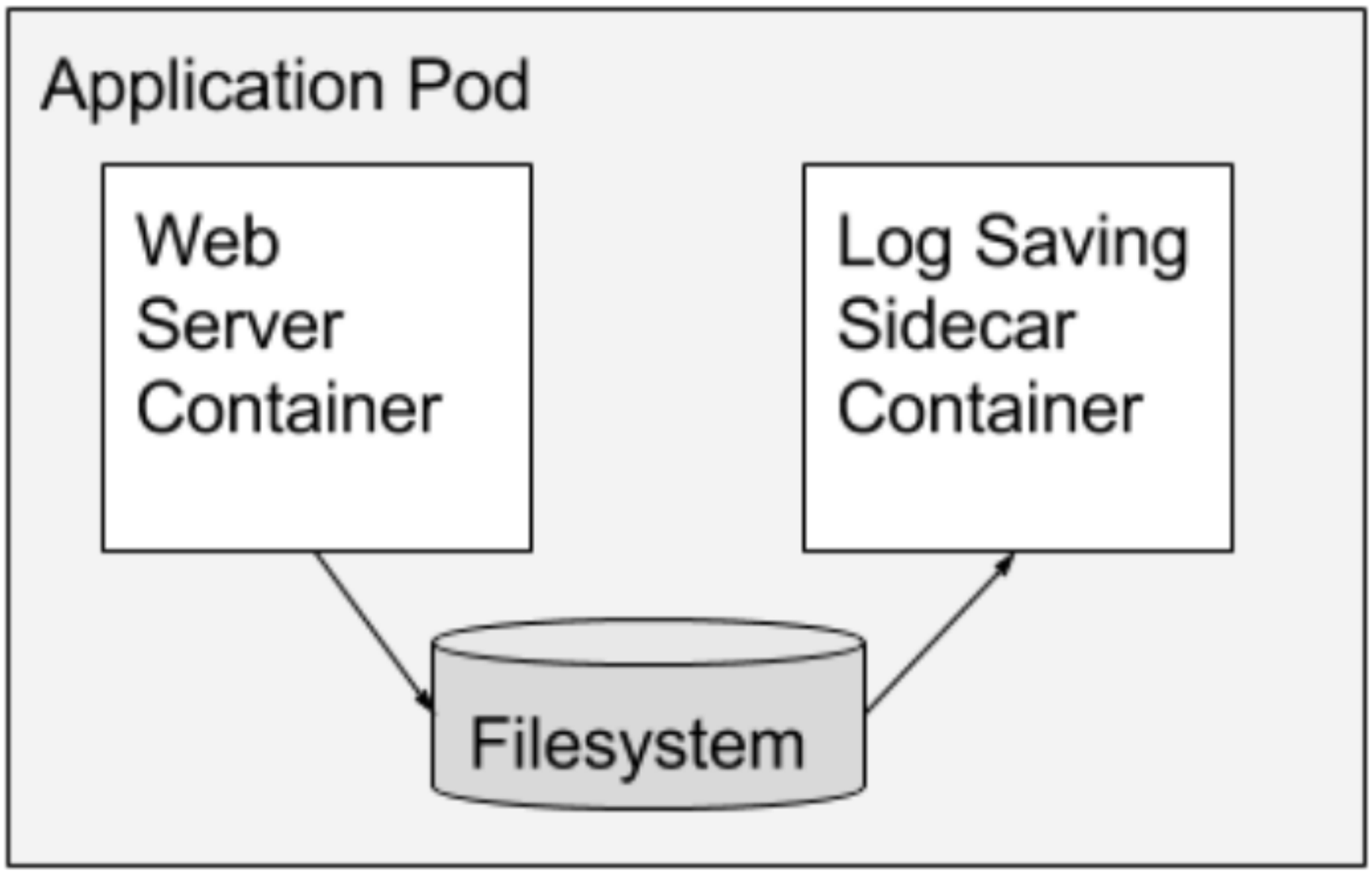
- Implement health monitoring by sending requests to an endpoint on the application. The application should perform the necessary checks, and return an indication of its status.

Strangler pattern



- 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.

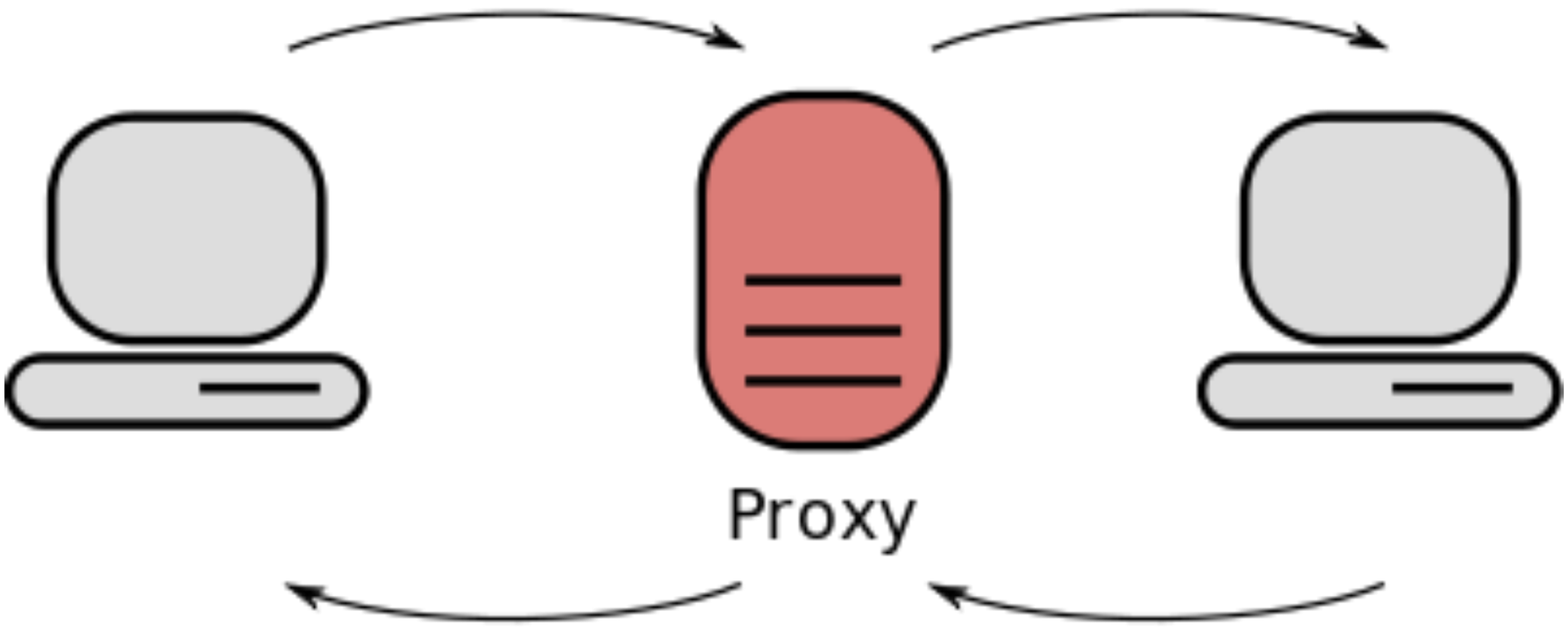
Sidecar


Application Pod
Web
Server
Container
Log Saving
Sidecar
Container
Filesystem

- 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:

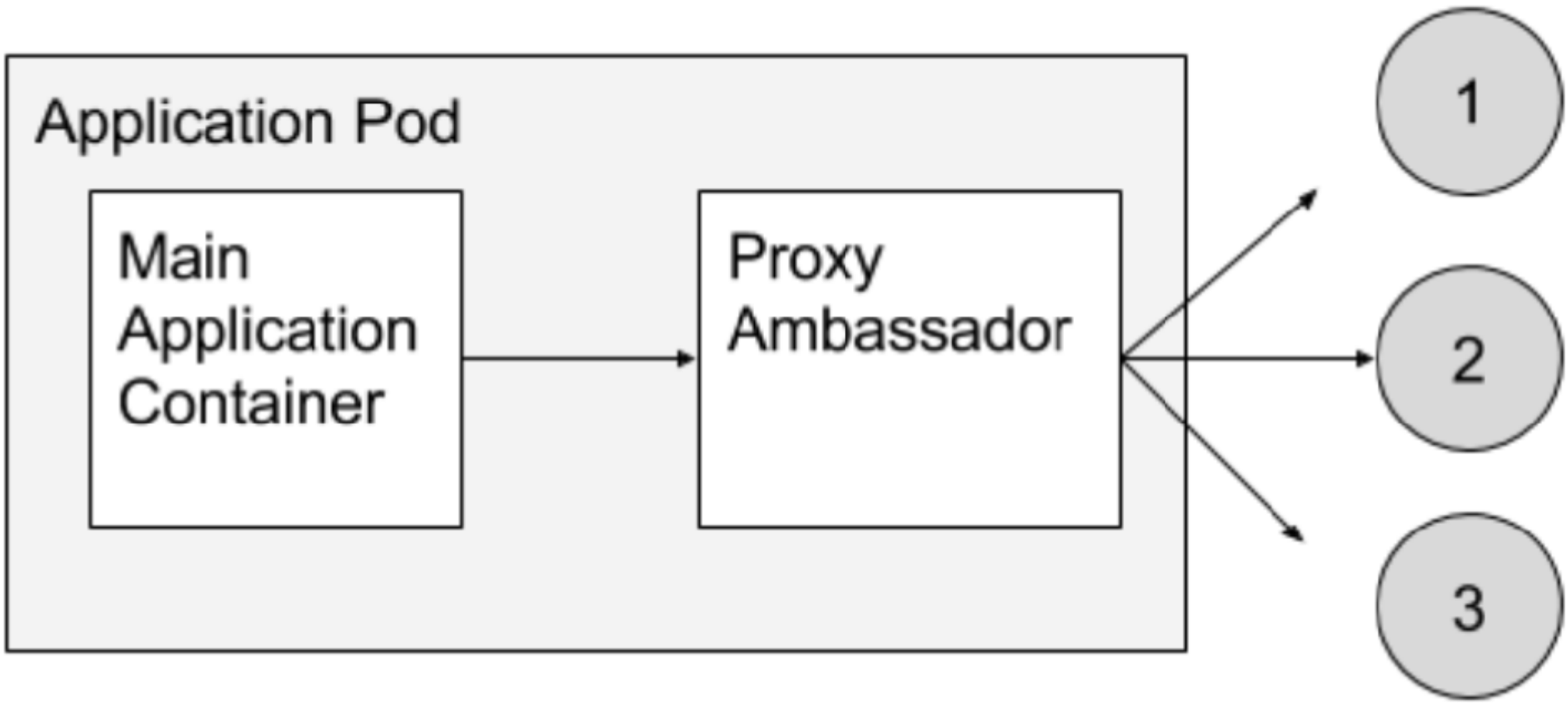
- Remote proxy ("Ambassador") - instead of communicating between server's and client's classes we create a proxy on client side and server side, and only the proxies communicate

- Virtual proxy - creates expensive object on first demand

- Protection proxy - to control access

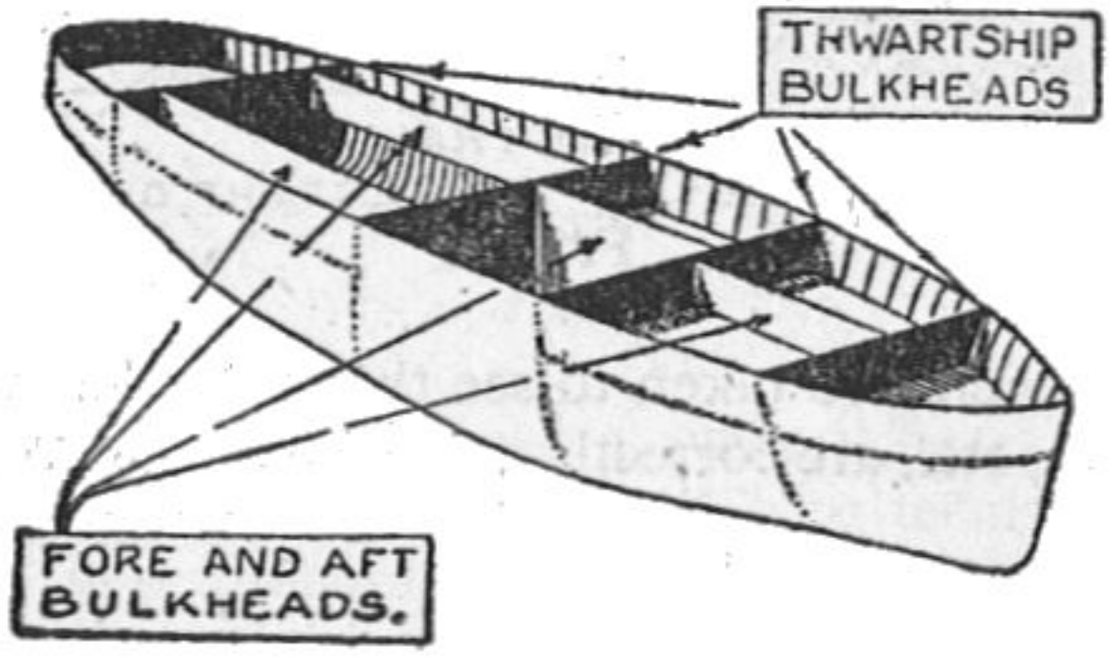
- Smart proxy - enrich an object

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.

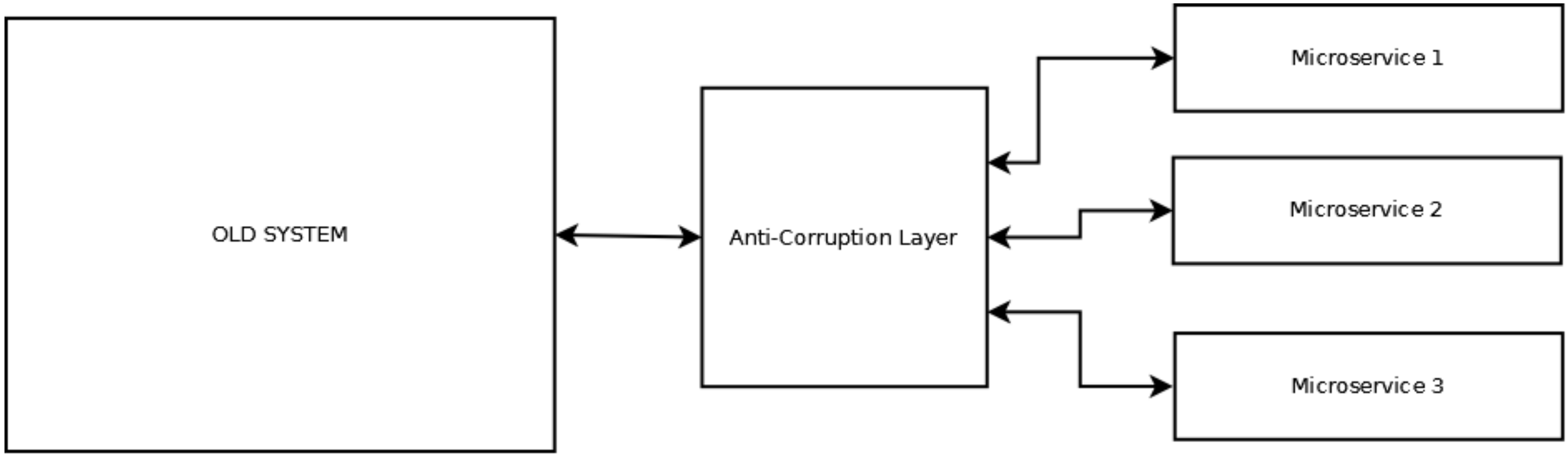
bulkhead



- 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.

Anti-Corruption Layer

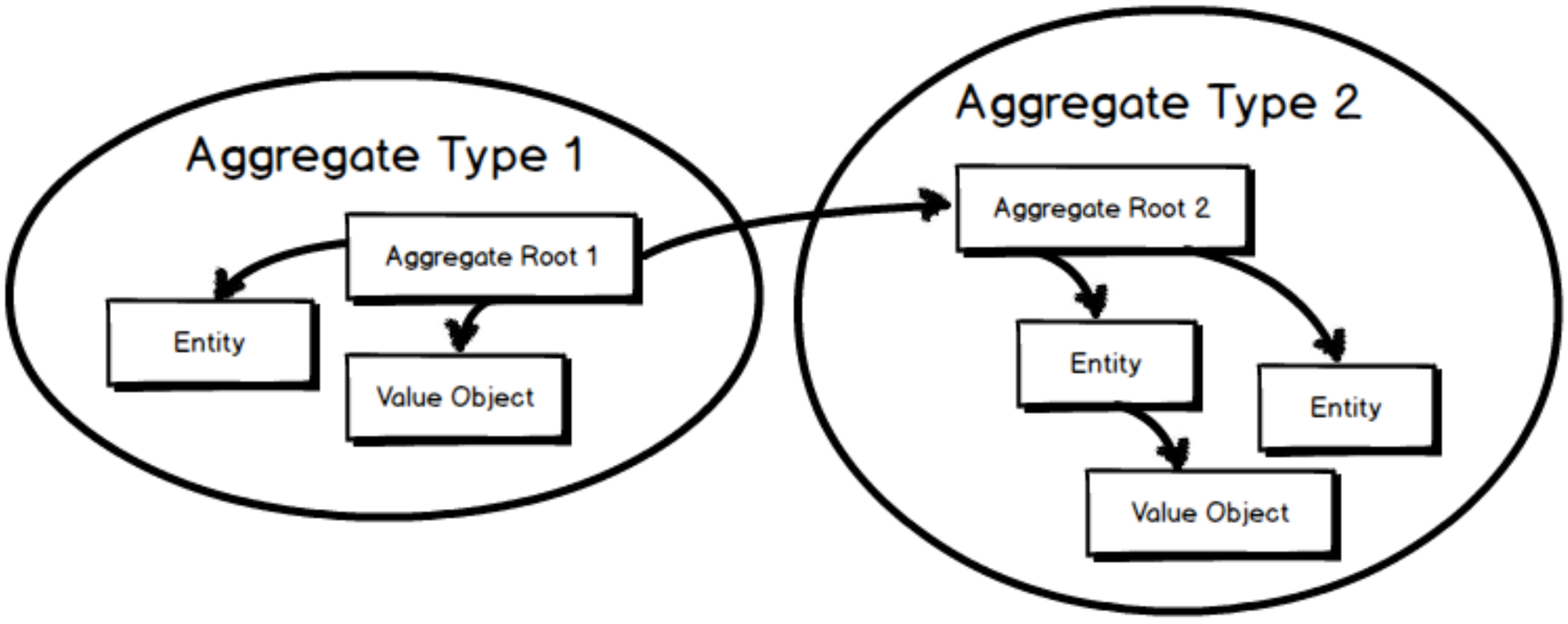


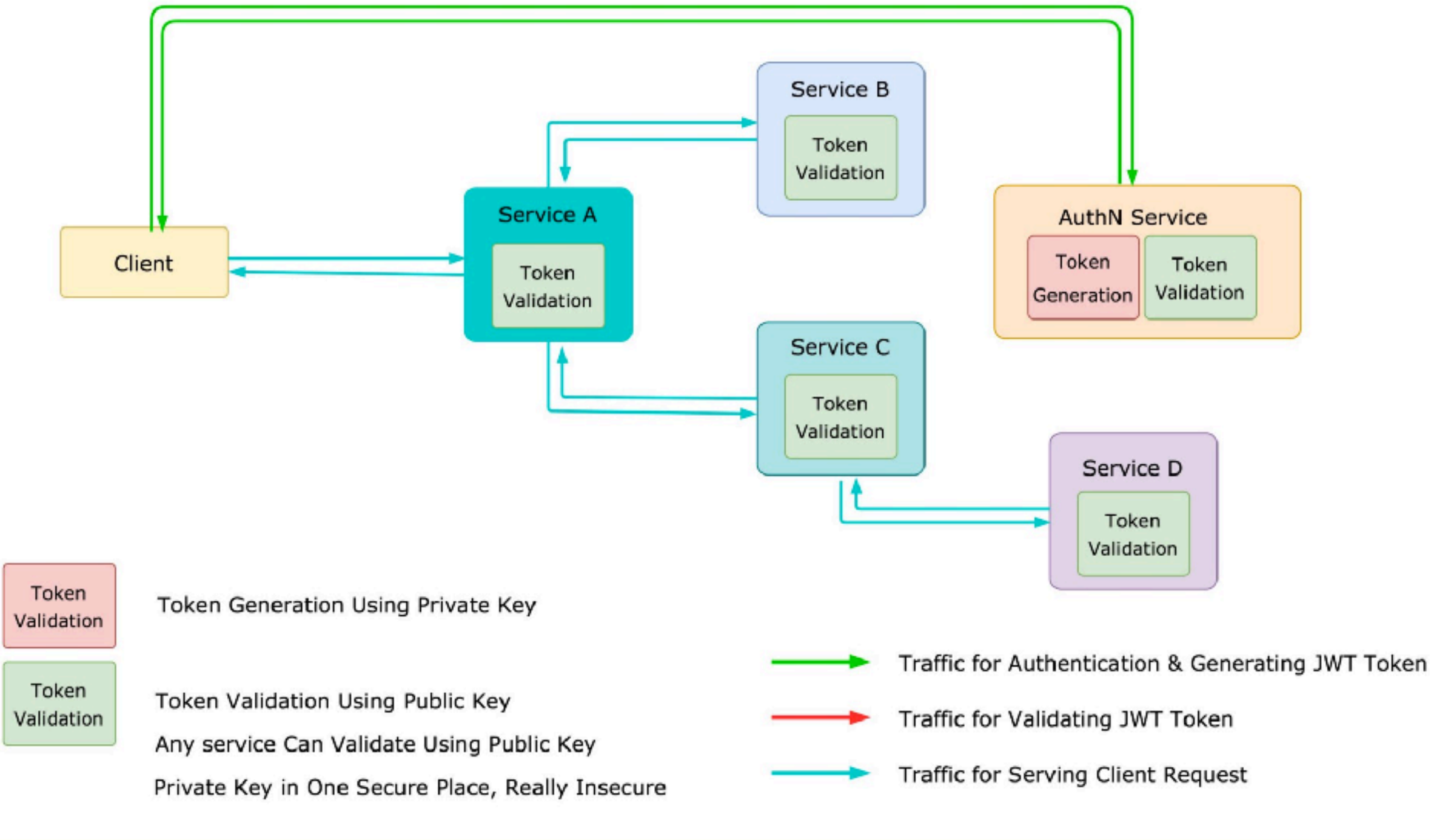
- When you create a new microservice, you notice that some business or technical assumptions or flow should be remodeled. instead of a Big Rewrite ACL maps one domain onto another so that services that use second domain do not have to be "corrupted" by concepts from the first.

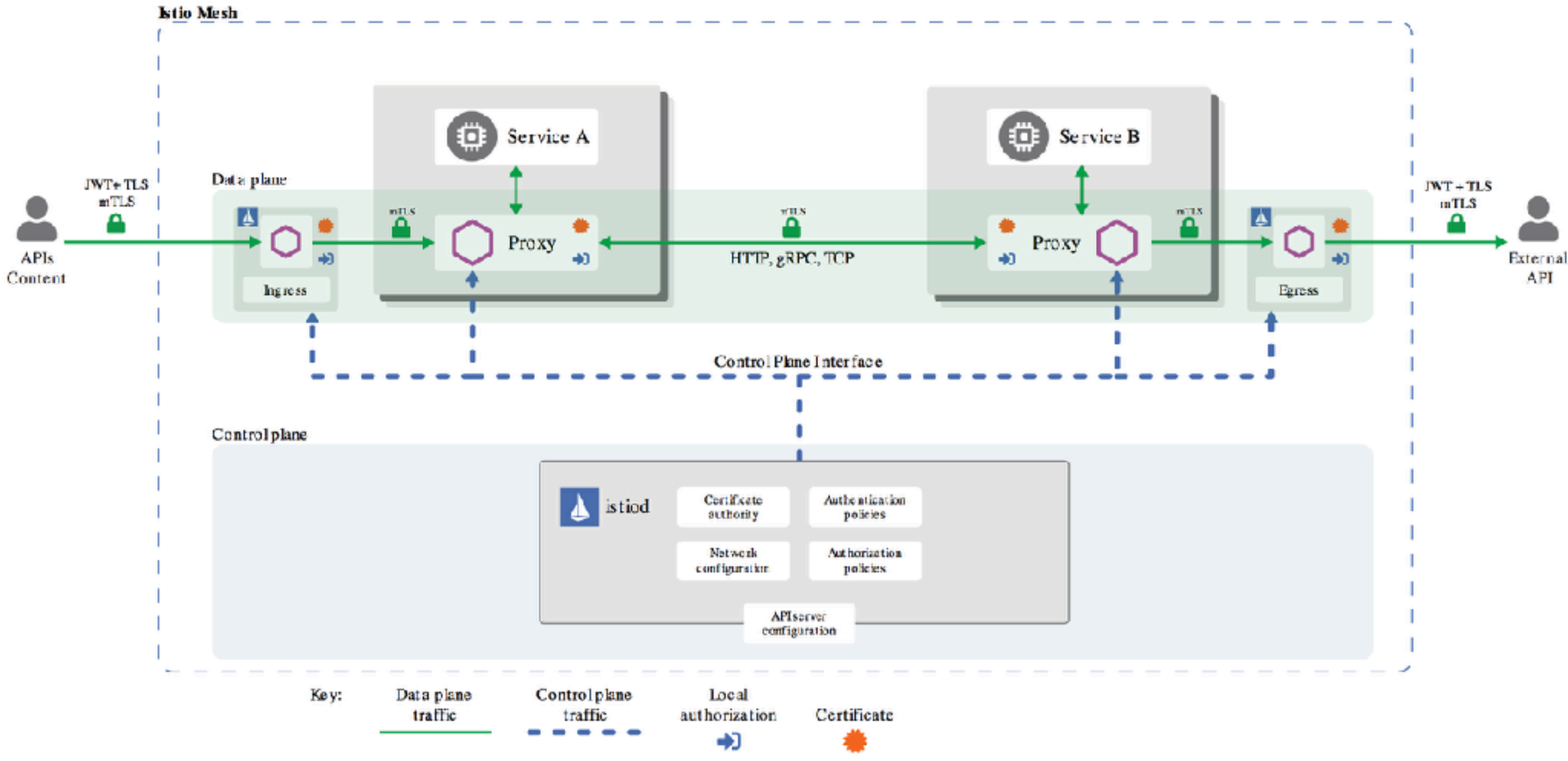
- Such modifications may result in changes to the events that a bounded context publishes.

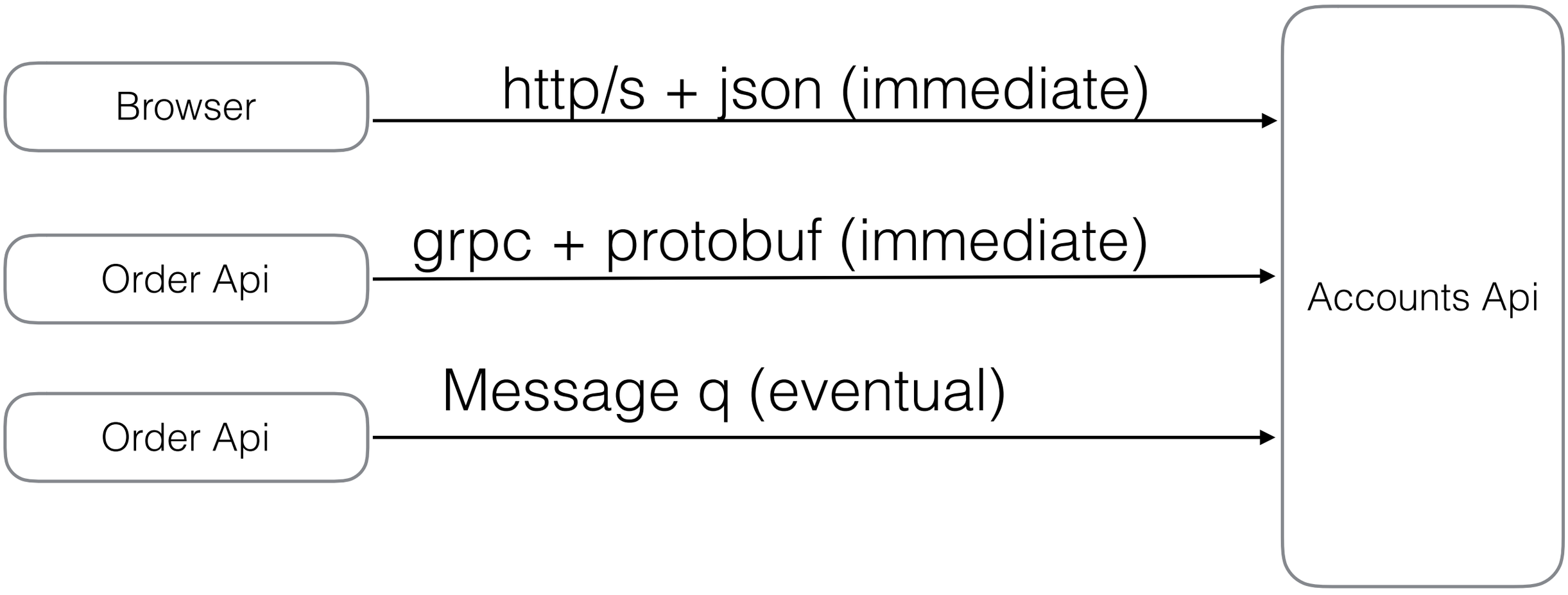
Bounded Context

Aggregate Root




Client
Service A
Token Validation
Service B
Token Validation
Service C
Token Validation
Service D
Token Validation
AuthN Service
Token Generation
Token Validation
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

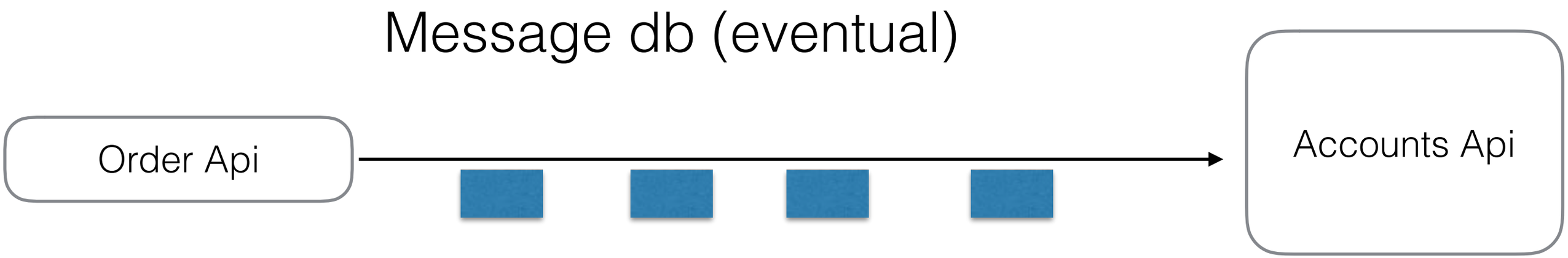

Istio Mesh
Service A
Service B
JWT+ TLS
mTLS
APIs
Content
Data plane
mTLS
Proxy
HTTP, gRPC, TCP
Proxy
mTLS
JWT + TLS
mTLS
External
API
Ingress
Egress
Control Plane Interface
Control plane
istiod
Certificate
authority
Authentication
policies
Network
configuration
Authorization
policies
APIserver
configuration
Key:
Data plane
traffic
Control plane
traffic
Local
authorization
Certificate


Browser
http/s + json (immediate)
Order Api
grpc + protobuf (immediate)
Accounts Api
Order Api
Message q (eventual)

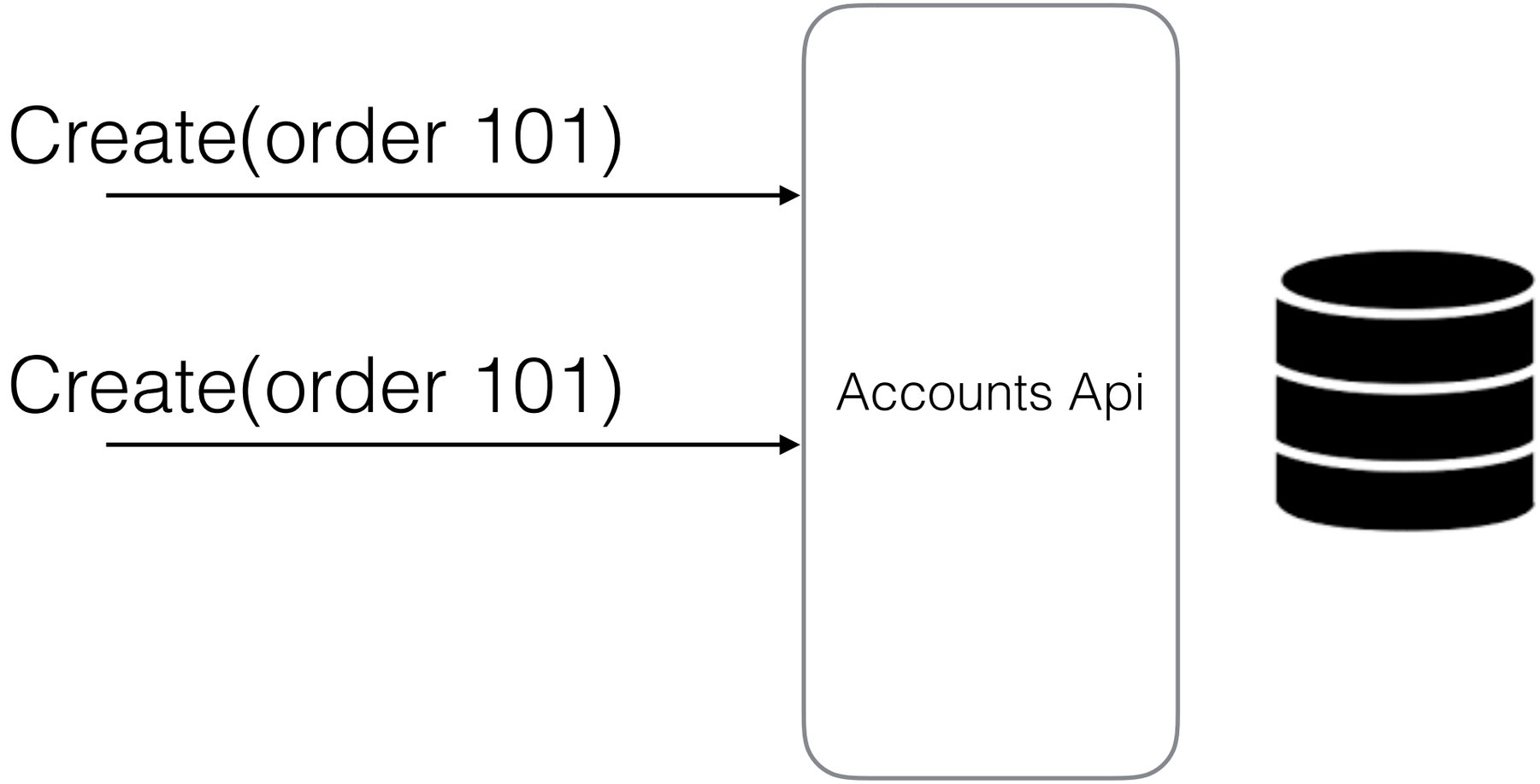
	http	Grpc	Message Q
Pay load	JSON	Protobuf	Any
Immediate	Y	Y	Eventual
Reliable	N	N	Y (retry)
Type	Connected	Connected	Disconnected
Scale	- (chatty)	+ (chunky)	+++ (load leveling)
Protocol	Request-response	Request-response	One way

Message Q

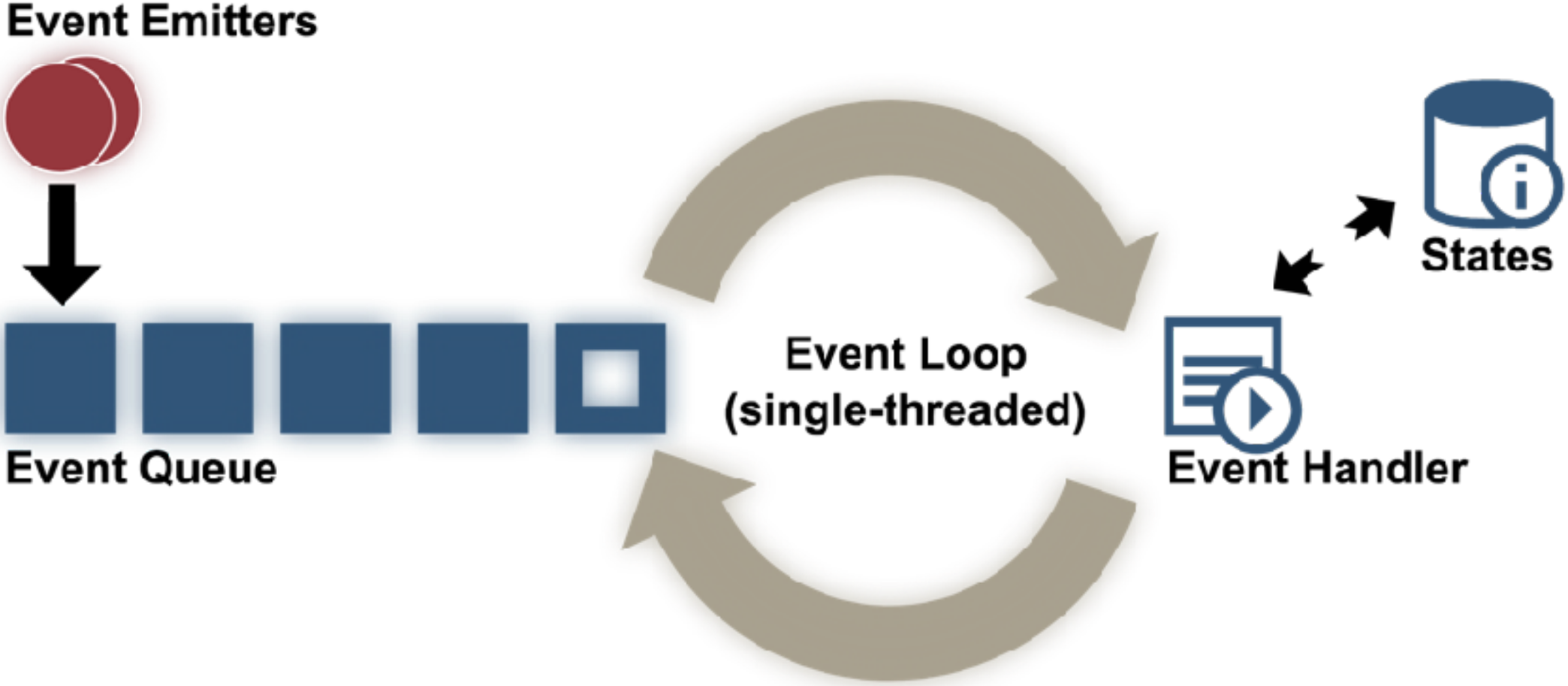
- One way protocol (result and error)
- UnOrdered Delivery
- Duplicate delivery (Idempotent)
- Retention Period



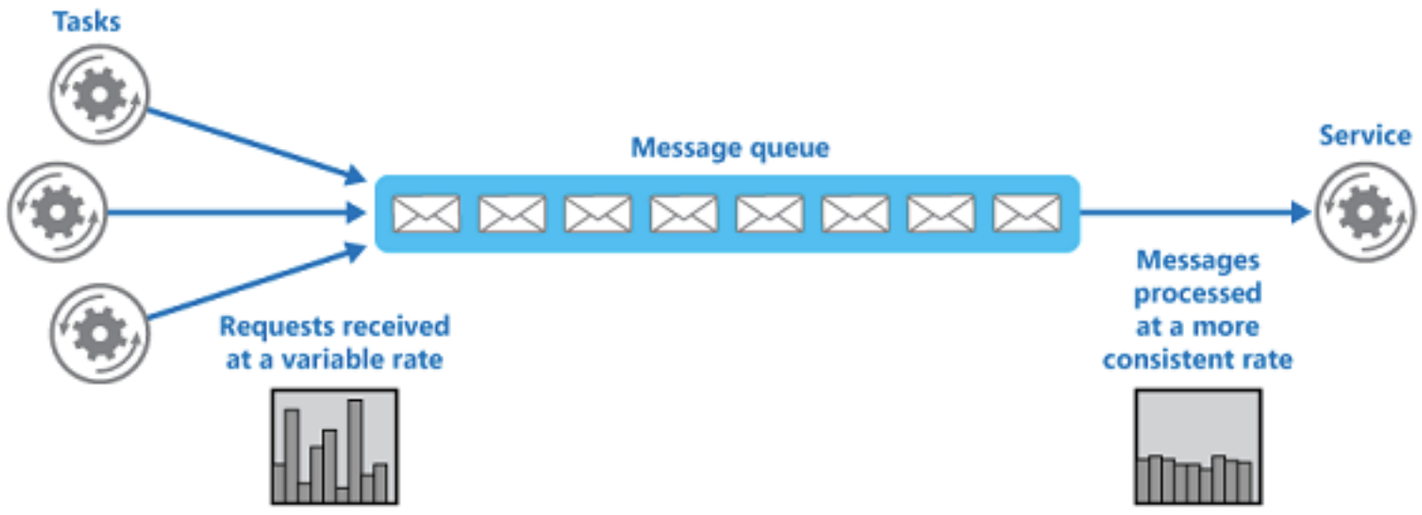

```
msg = getmessage()
...
...
msg.ack()
```


Create(order 101)
Create(order 101)
Accounts Api

Event Driven Architecture

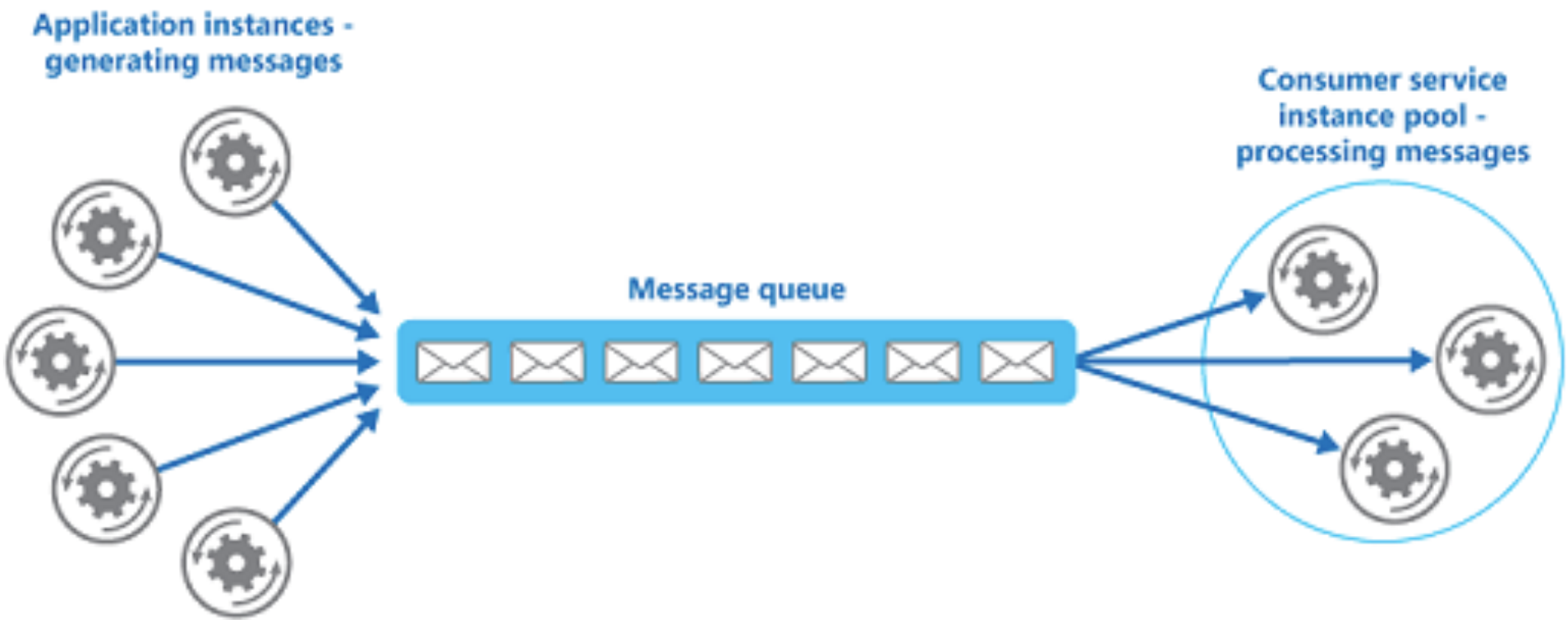


Queue-based Load Levelling



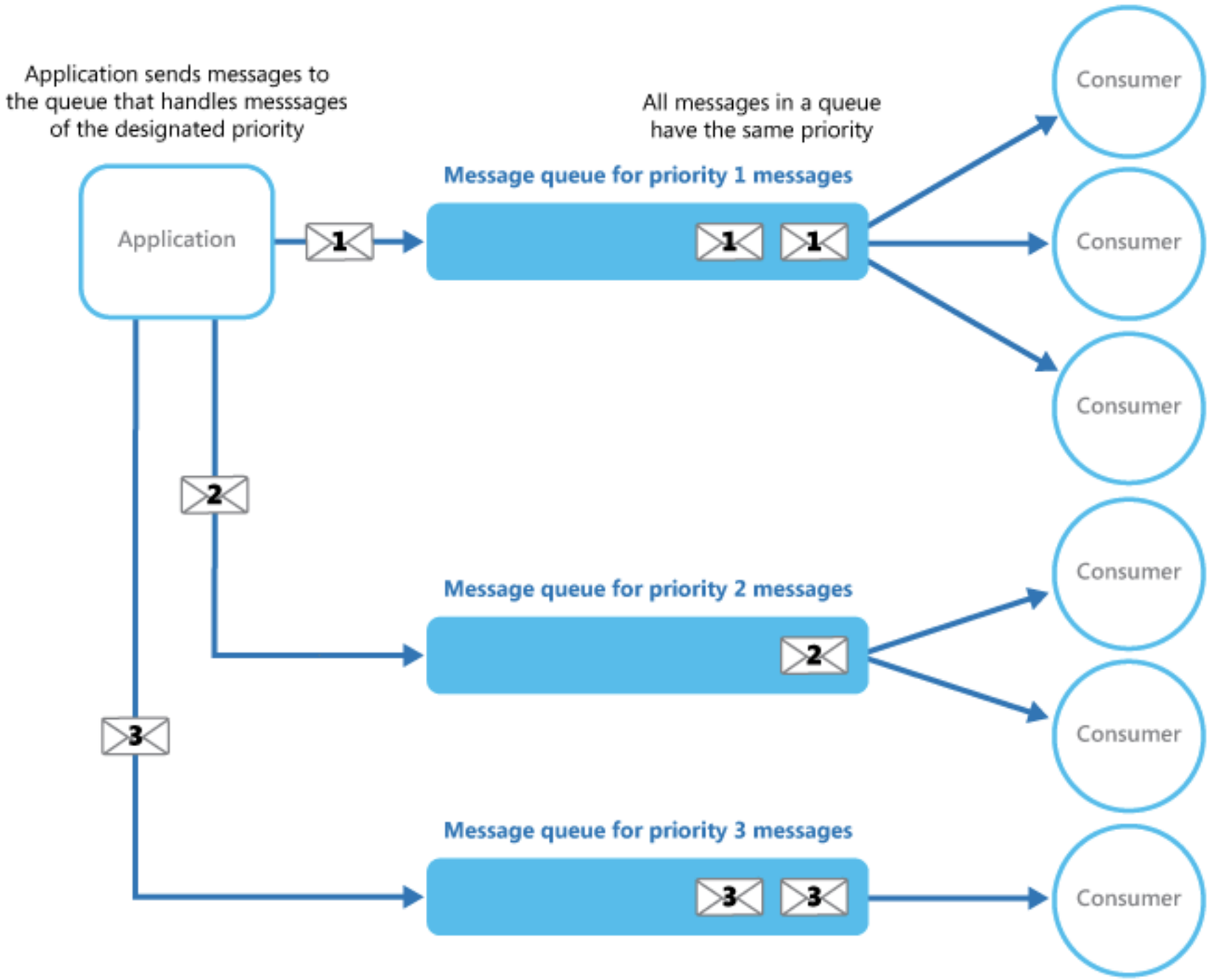
source:msdn

Competing Consumers



source:msdn

Priority Queue



source:msdn

 # Event Bus
 # EDA
 # compensatale transaction
 # SAGA
 # Event Sourcing
 # CQRS

Load Balancer

Gateway

Inventory Api

CUD

Event

R

Grpc

Stream Analytics

Event Bus

Event

Event

Event

Audit Api

Inventory Cmd Api

Do

Undo

Inventory Query Api

Do

Undo

Data Lake

Event db

Full data

Data Ware House

snap shot data
- Weekly
- Monthly
- Yearly

PostgreSQL

rdbms

3rd normal form
removes duplicates
increases joins
write friendly (no duplicates)
no read friendly (joins)
current data

mongoDB

Document Db

de normalised form
increase duplicates
remove joins
not write friendly (duplicates)
read friendly (no joins)

Batch/Bigdata Analytics

OLAP Analytics

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

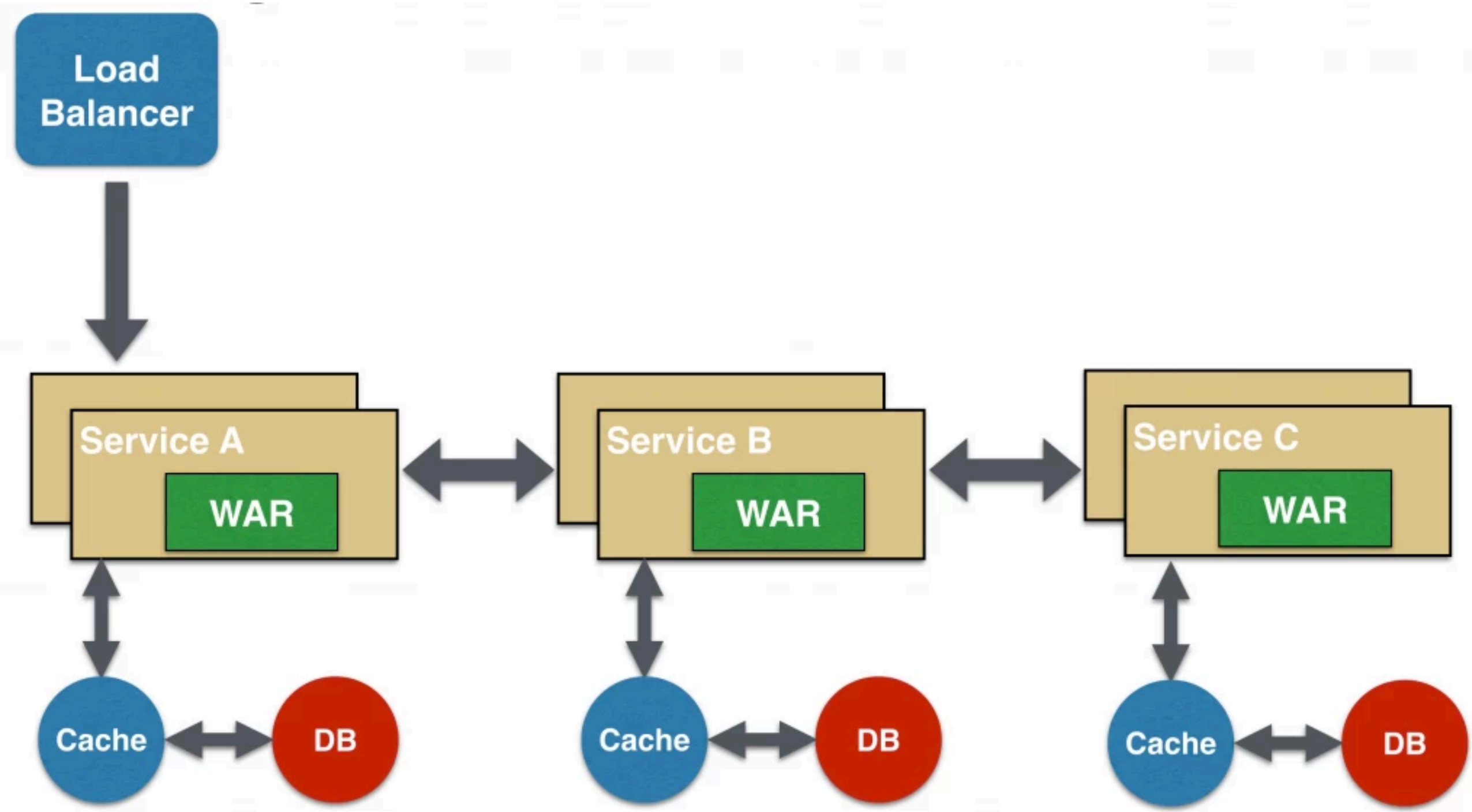
TABLE_GENRE

Genre ID	Genre Type
1	Gardening
2	Sports
3	Travel

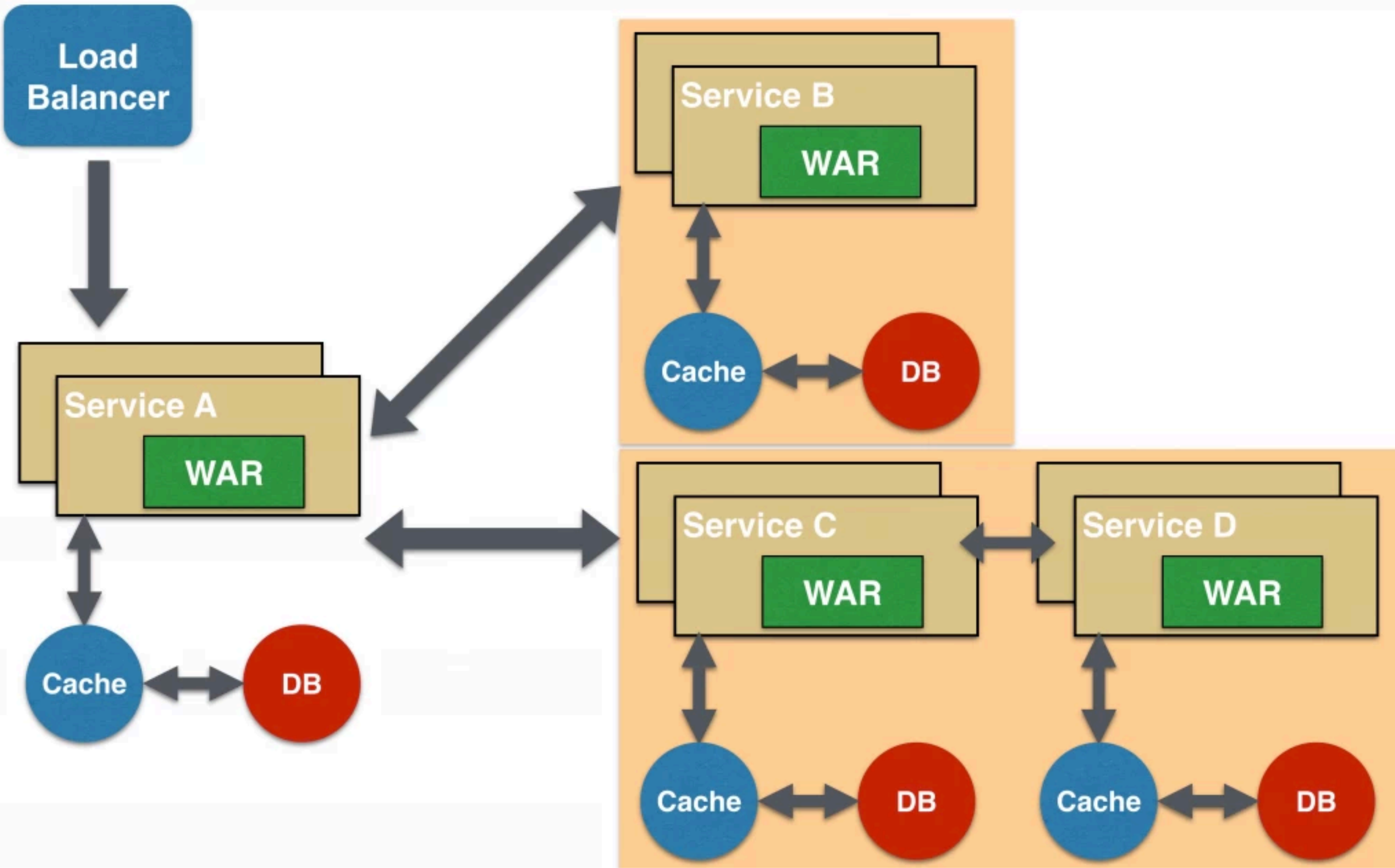
Weather

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

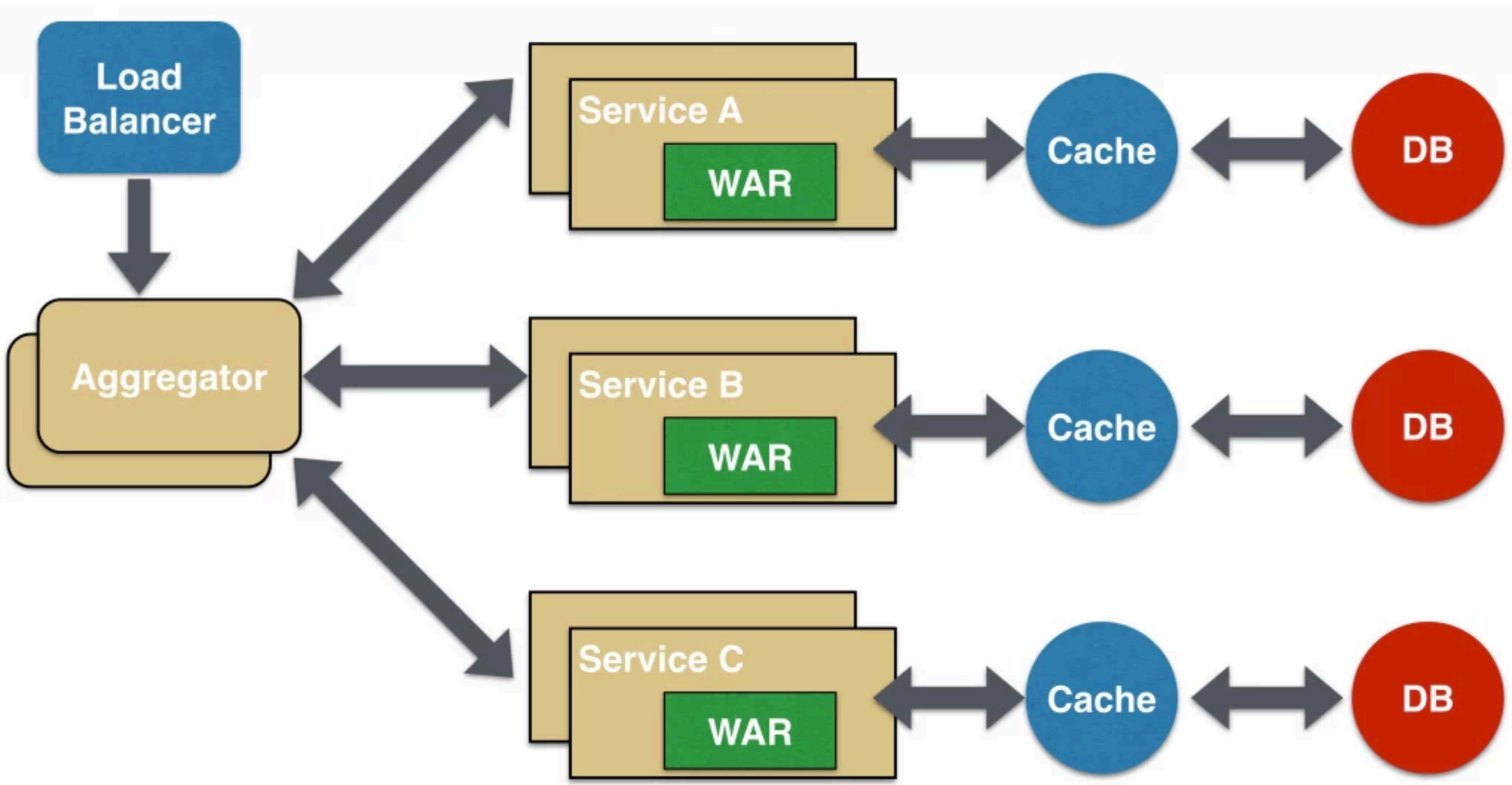
Chained Microservice

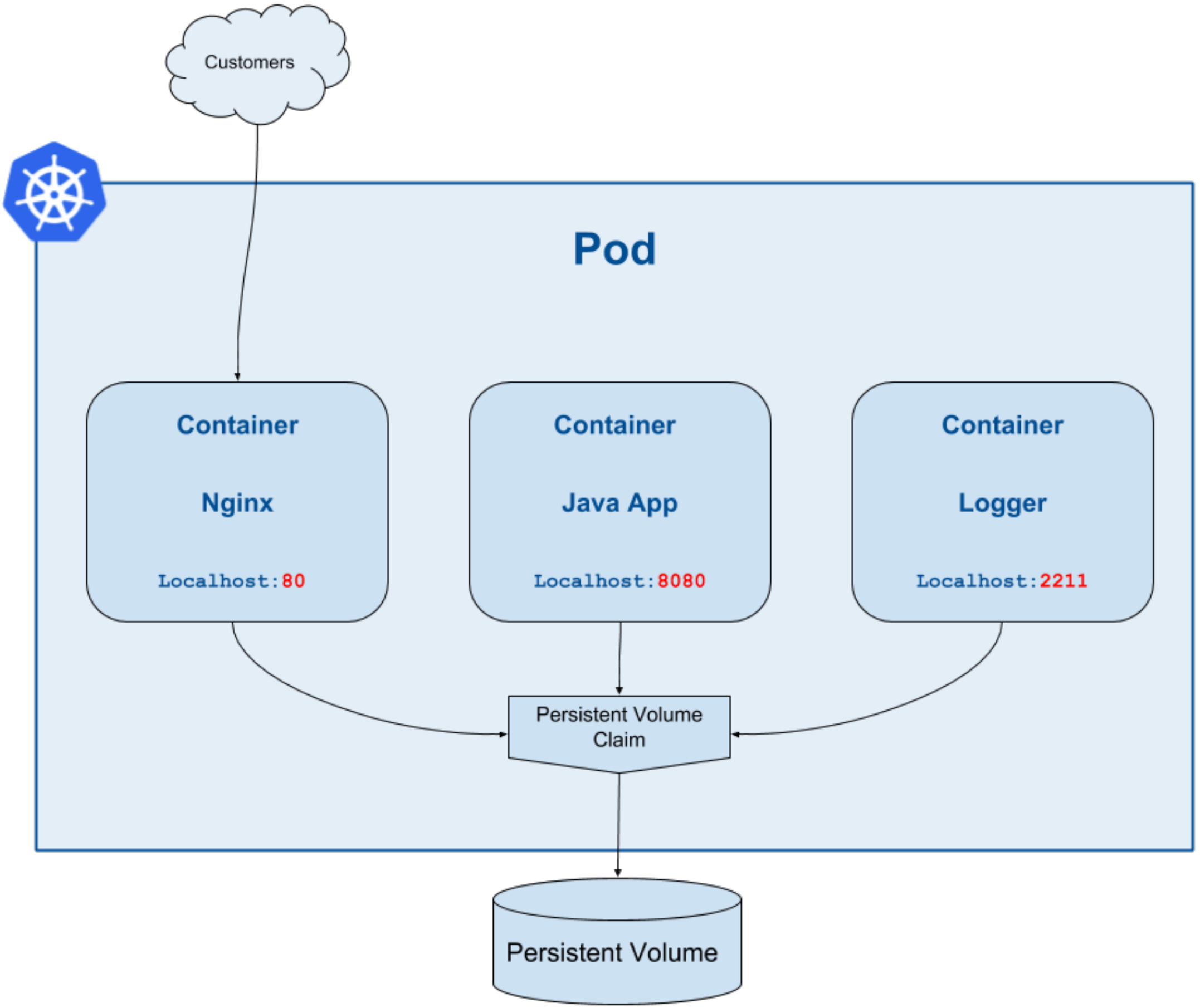


Branch Microservice



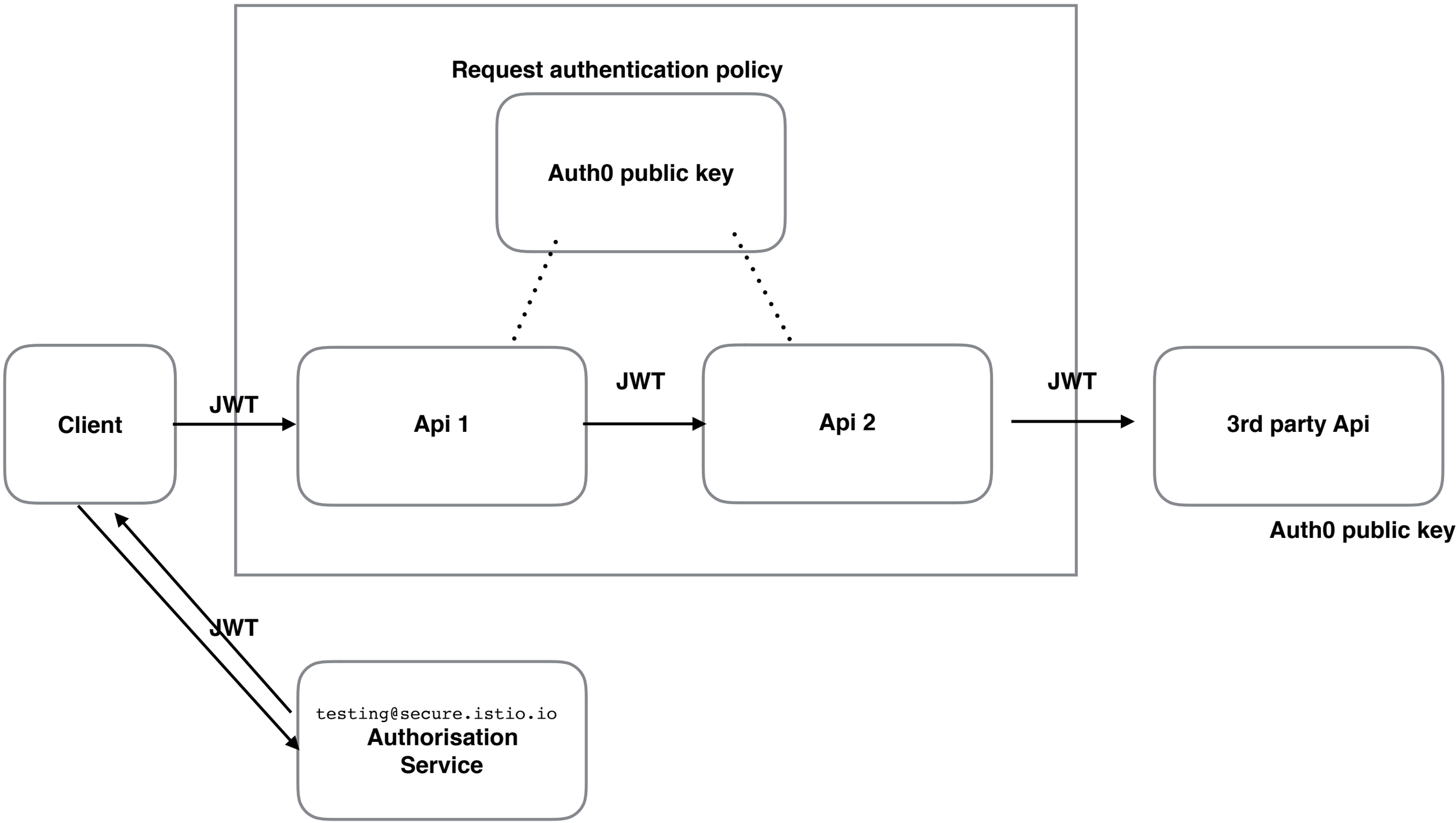
Aggregator Microservice




Customers
Pod
Container
Nginx
Localhost:80
Container
Java App
Localhost:8080
Container
Logger
Localhost:2211
Persistent Volume Claim
Persistent Volume

Concern	Approach
Package runtime dependencies Reproducable Environment “it works on my machine”	Docker image
Multiple Service Deployment Isolation	Containers
Simplify Deployment of many services	yaml, Helm
Distributed Deployment, scale	Kubernetes
Observability	Kubernetes dash board, weave scope, Kiali, graffana, kibana
Service Discovery	Cluster-IP, Node Port, Load Balancer, Ingress Gateway
Traffic Management (timeouts, retry, circuit breaker, throttle, canary, ...)	Service Mesh (Istio)
Log Management (debug & error)	EFK
Traceability	Promethus, jager
Transaction	SAGA
Join Query (report query)	CQRS , Materialized View
Configuration	ConfigMap, Consul, KeyValut, ..
Performance	Shared db, grpc, Async, branch pattern
Authentication	OAuth2, JWT, OpenID connect

Oath 2


Request authentication policy
Auth0 public key
Client
JWT
Api 1
JWT
Api 2
JWT
3rd party Api
Auth0 public key
JWT
testing@secure.istio.io
Authorisation
Service

Appendix

- 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:

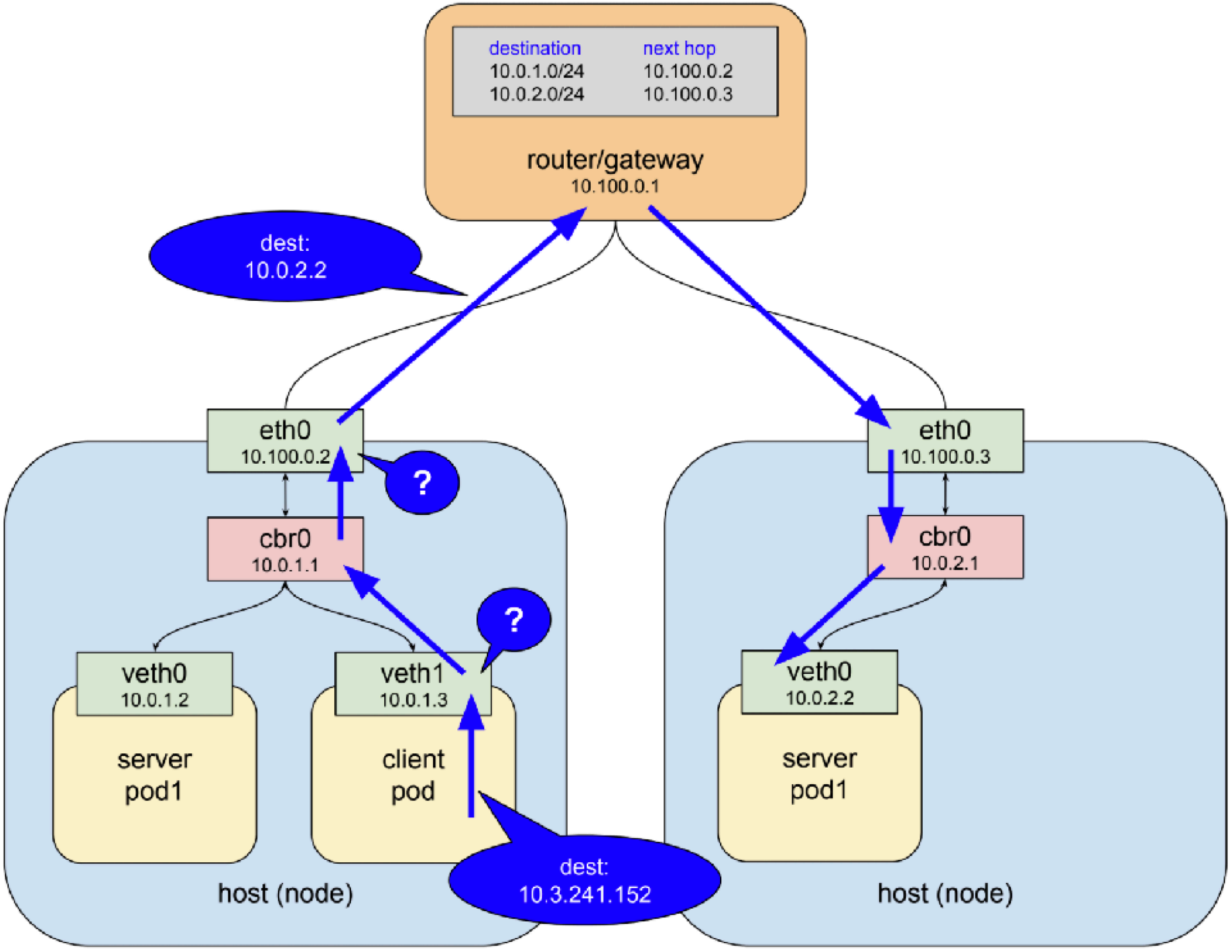
- Needs more public IPs, which normally are limited resources.
- Introduces coupling between the client and the server, making it hard to adjust your backend services when business requirements change.

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.

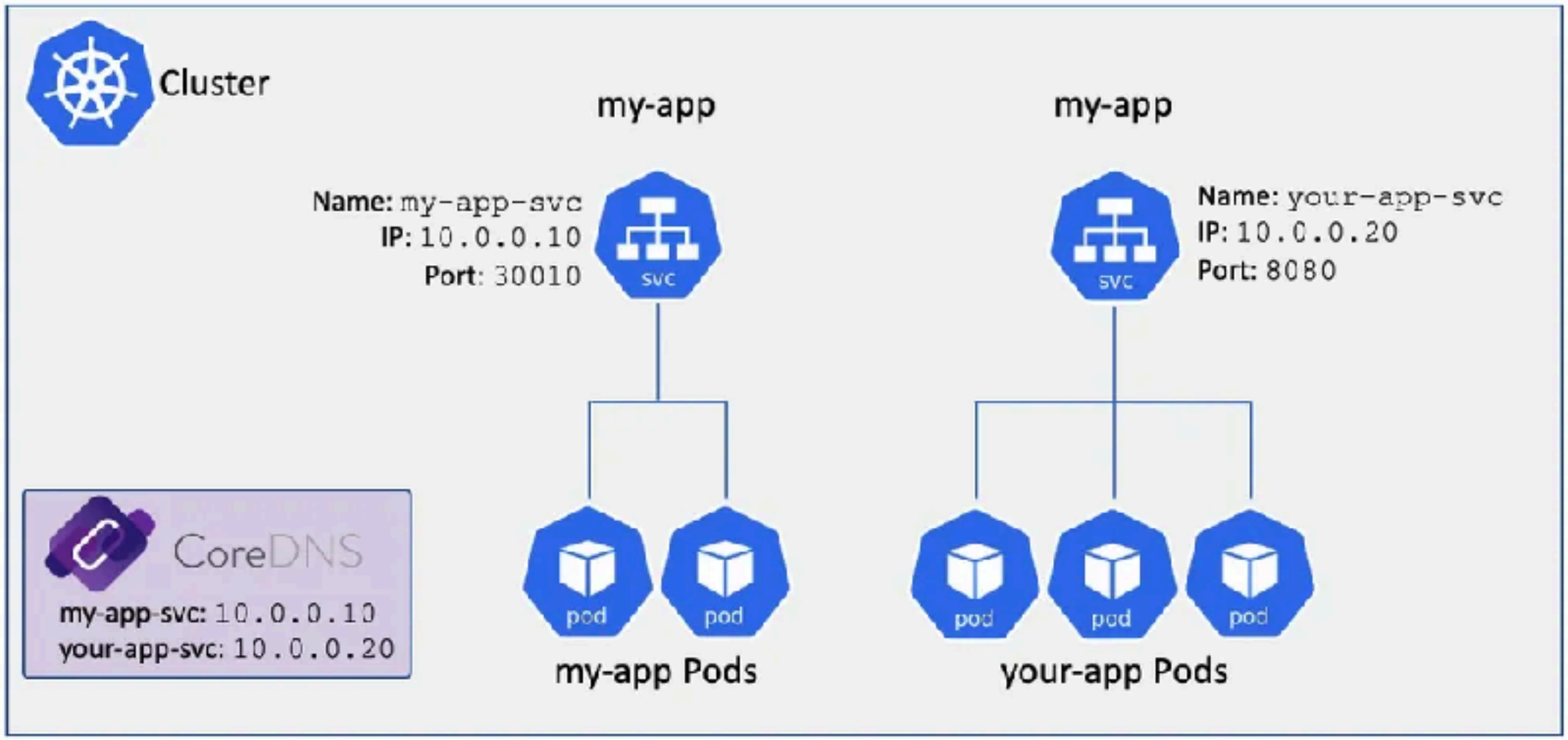
- Ingress controller itself is also deployed as Pods inside the cluster.

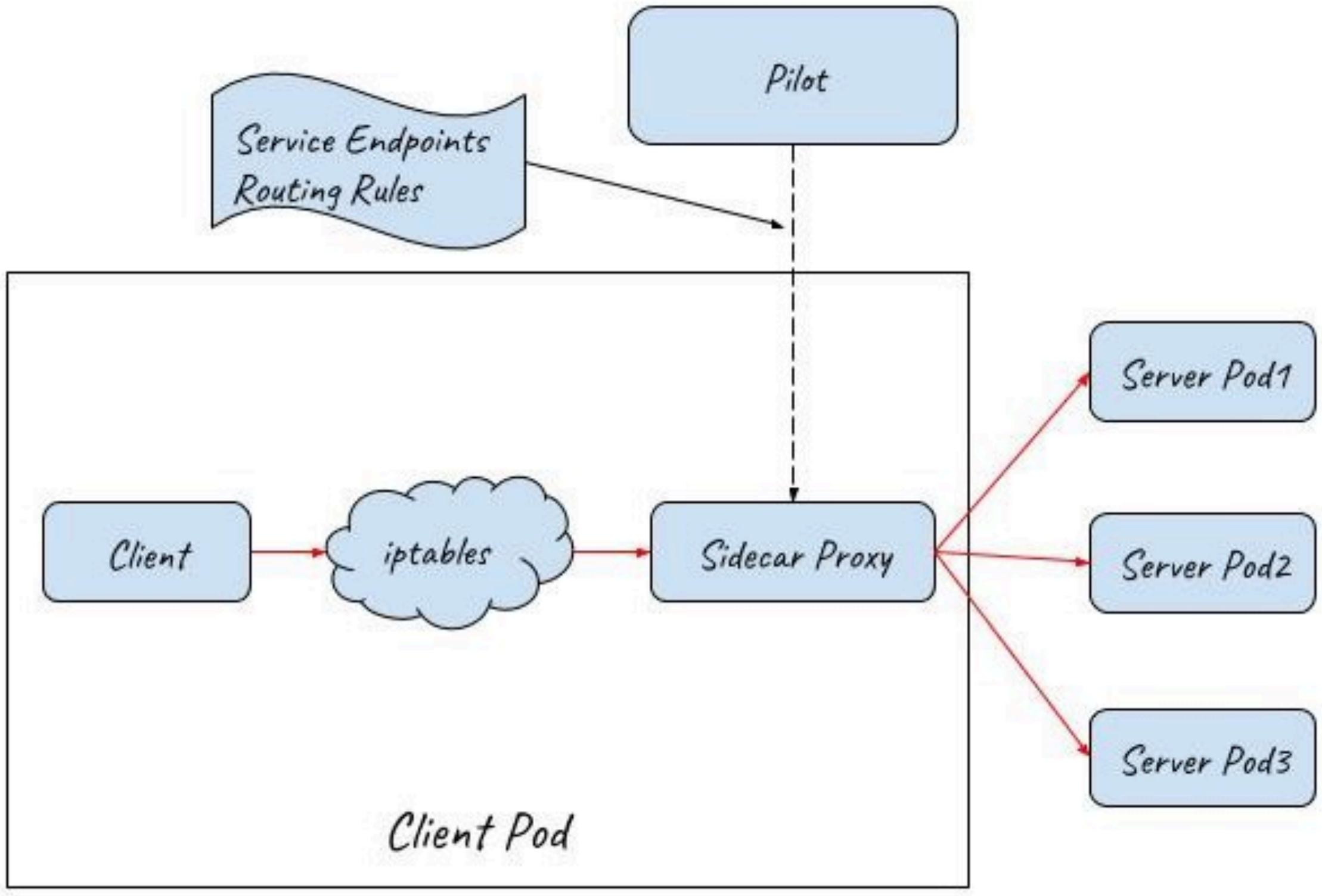
- Ingress controller provides a unified entrance for the HTTP services in a cluster, but it can’t be accessed directly from outside because the ingress controller itself is also deployed as Pods inside the cluster.

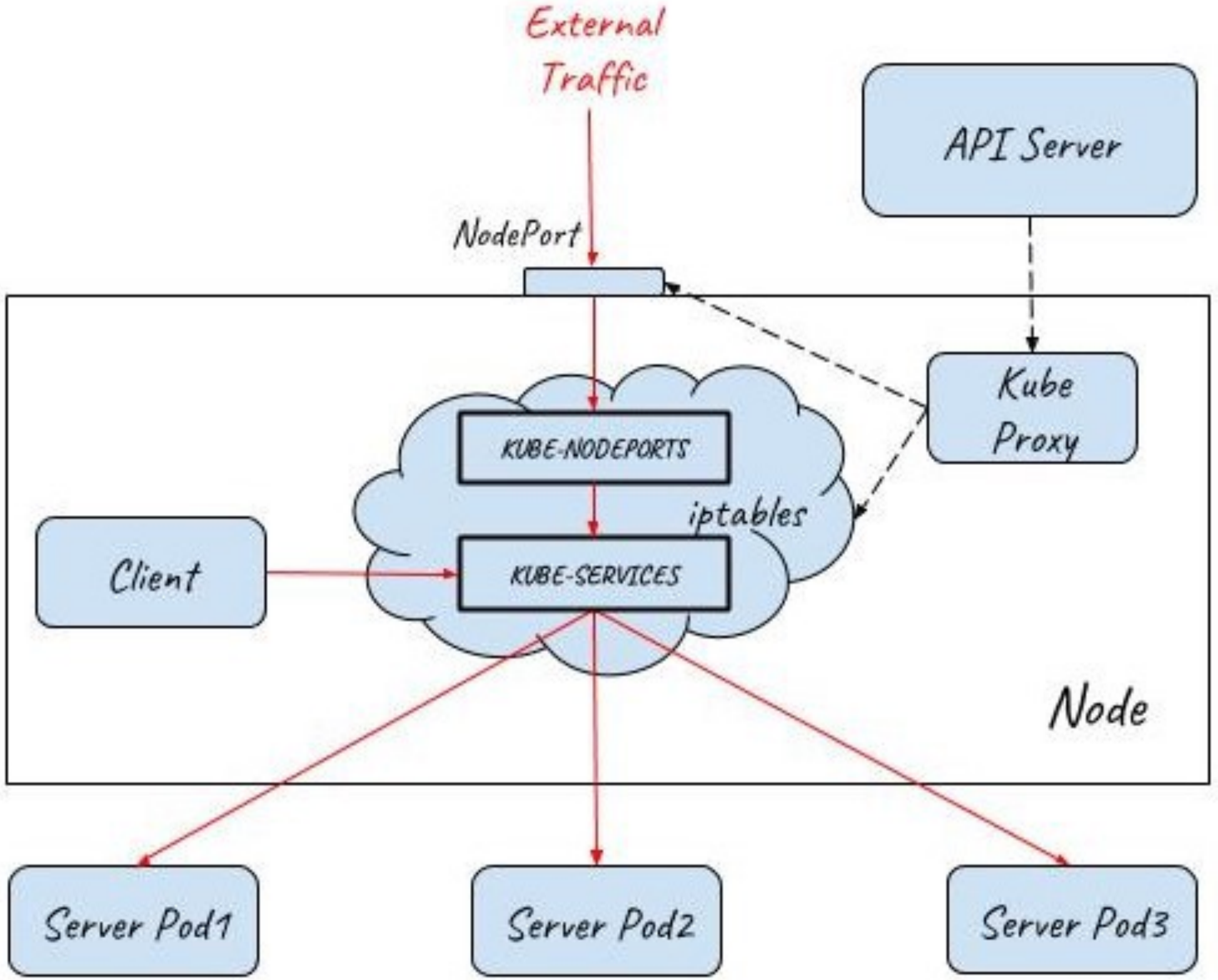
- Ingress controller must work together with NodePort and LoadBalancer to provide the full path for the external traffic to enter the cluster.

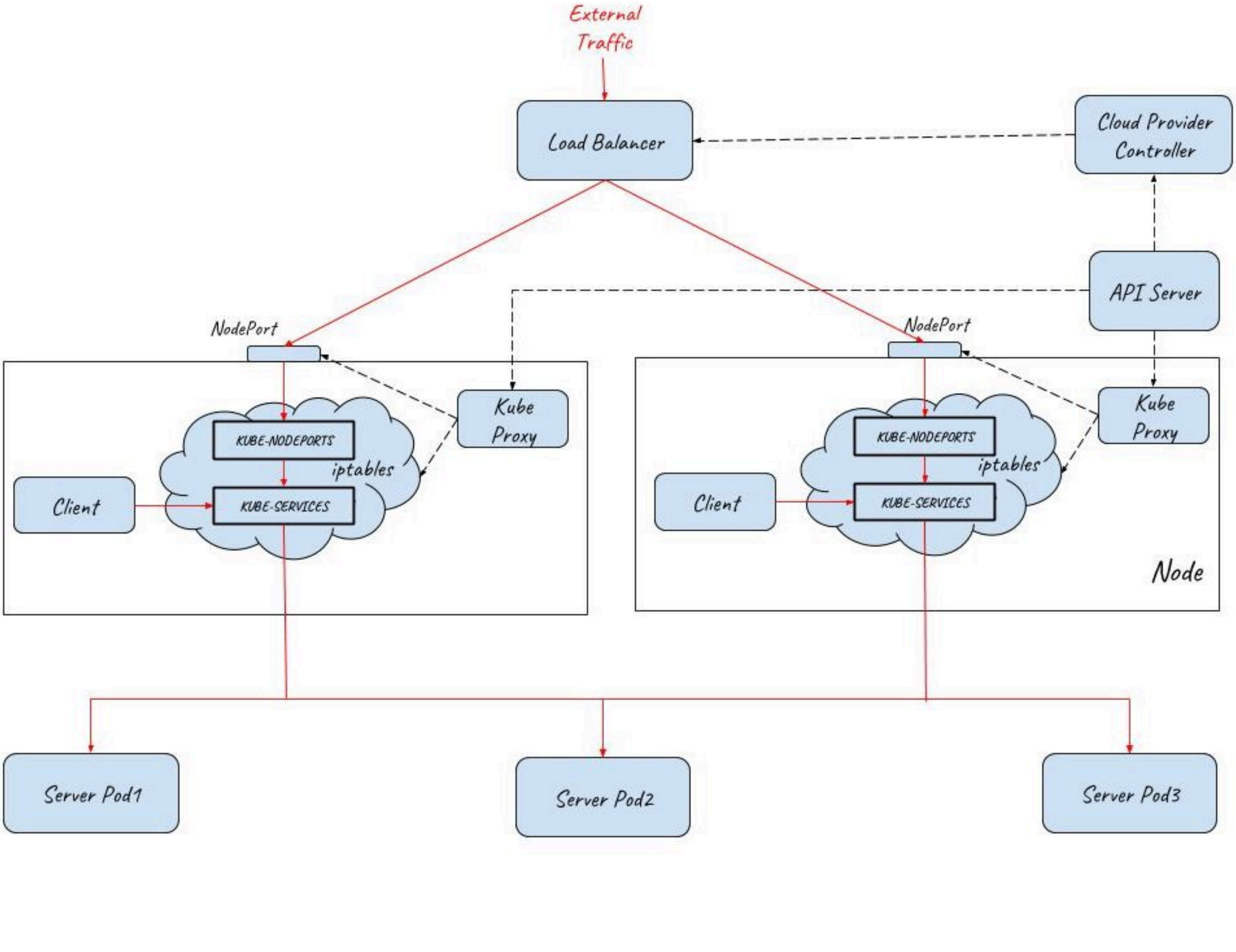

destination
next hop
10.0.1.0/24
10.100.0.2
10.0.2.0/24
10.100.0.3
router/gateway
10.100.0.1
dest:
10.0.2.2
eth0
10.100.0.2
?
cbr0
10.0.1.1
?
veth0
10.0.1.2
veth1
10.0.1.3
server
pod1
client
pod
dest:
10.3.241.152
host (node)
eth0
10.100.0.3
cbr0
10.0.2.1
veth0
10.0.2.2
server
pod1
host (node)

	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, IPSec, MPLS, ARP	**Packet**
Hardware	**Layer 2** Data Link	Ethernet, 802.1x, PPP, ATM, Fiber Channel, MPLS, FDDI, MAC Addresses	**Frame**
	Layer 1 Physical	Cables, Connectors, Hubs (DLS, RS232, 10BaseT, 100BaseTX, ISDN, T1)	**Bits**

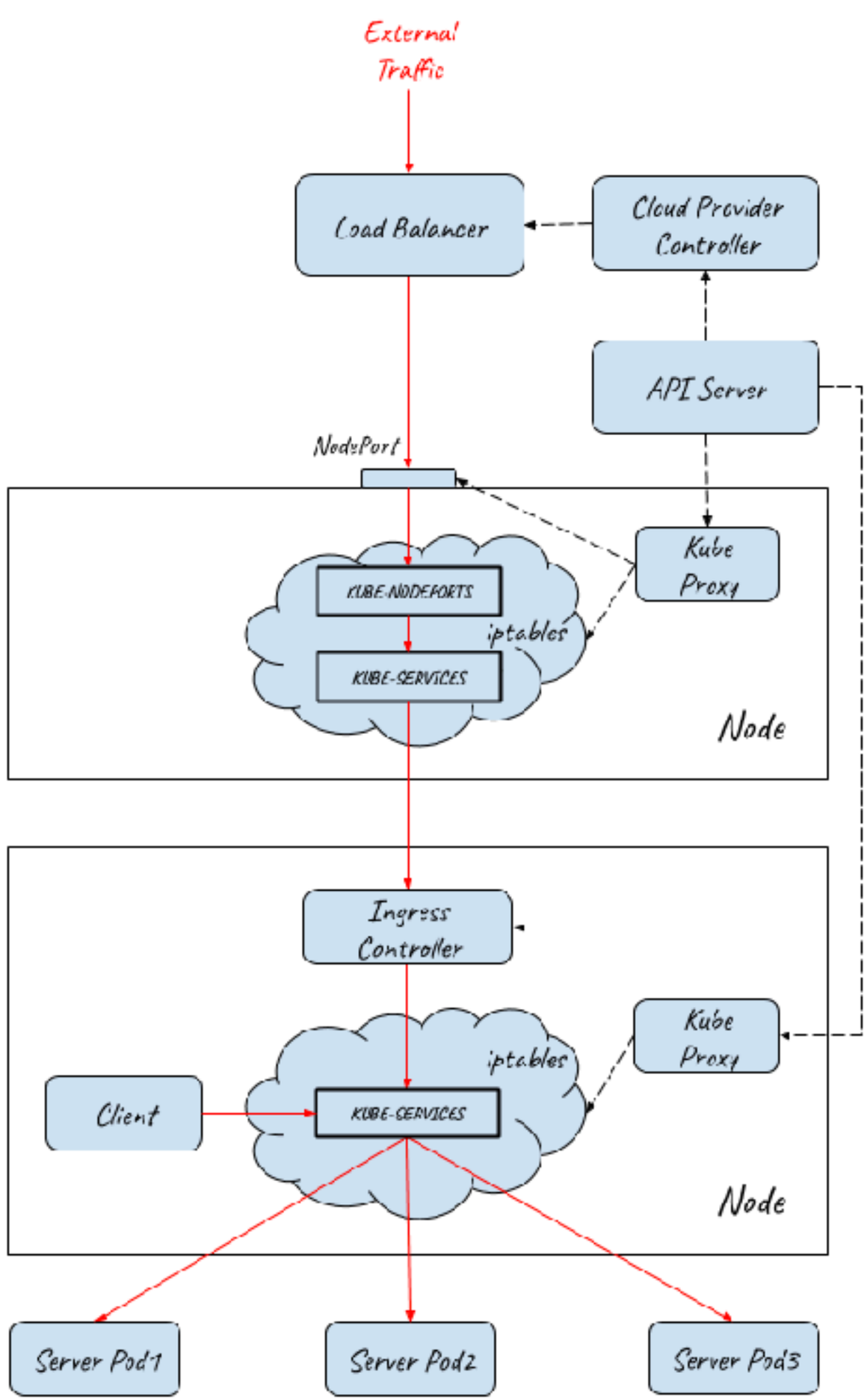

Cluster
my-app
Name: my-app-svc
IP: 10.0.0.10
Port: 30010
svc
my-app
Name: your-app-svc
IP: 10.0.0.20
Port: 8080
svc
CoreDNS
my-app-svc: 10.0.0.10
your-app-svc: 10.0.0.20
pod
pod
my-app Pods
pod
pod
pod
your-app Pods


Pilot
Service Endpoints
Routing Rules
Server Pod1
Client
iptables
Sidecar Proxy
Server Pod2
Server Pod3
Client Pod


External
Traffic
API Server
NodePort
Kube
Proxy
KUBE-NODEPORTS
iptables
Client
KUBE-SERVICES
Node
Server Pod1
Server Pod2
Server Pod3


External Traffic
Load Balancer
Cloud Provider Controller
API Server
NodePort
NodePort
Kube Proxy
Kube Proxy
KUBE-NODEPORTS
KUBE-NODEPORTS
iptables
iptables
KUBE-SERVICES
KUBE-SERVICES
Client
Client
Node
Server Pod1
Server Pod2
Server Pod3

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)

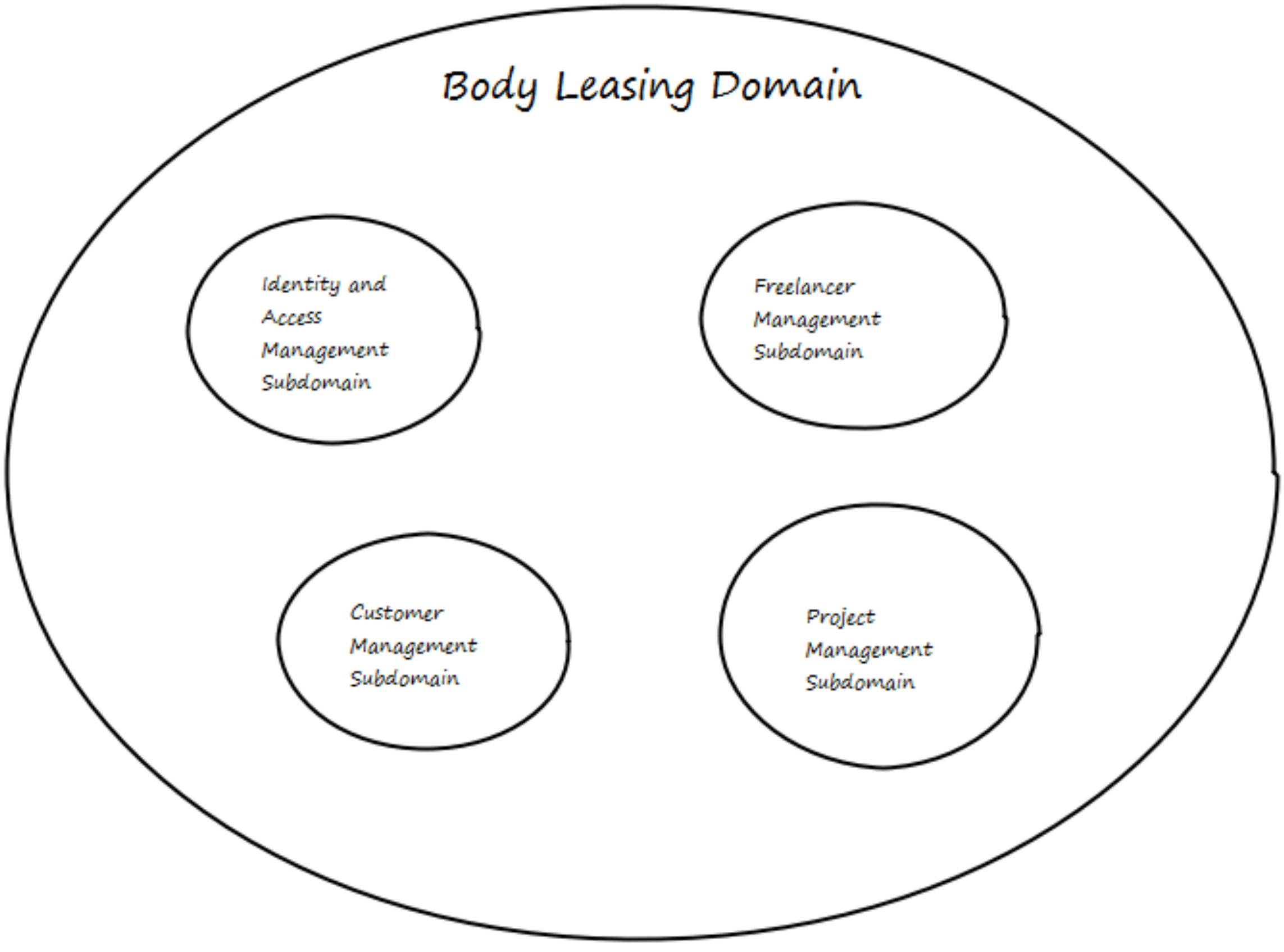
A searchable catalog of Customers

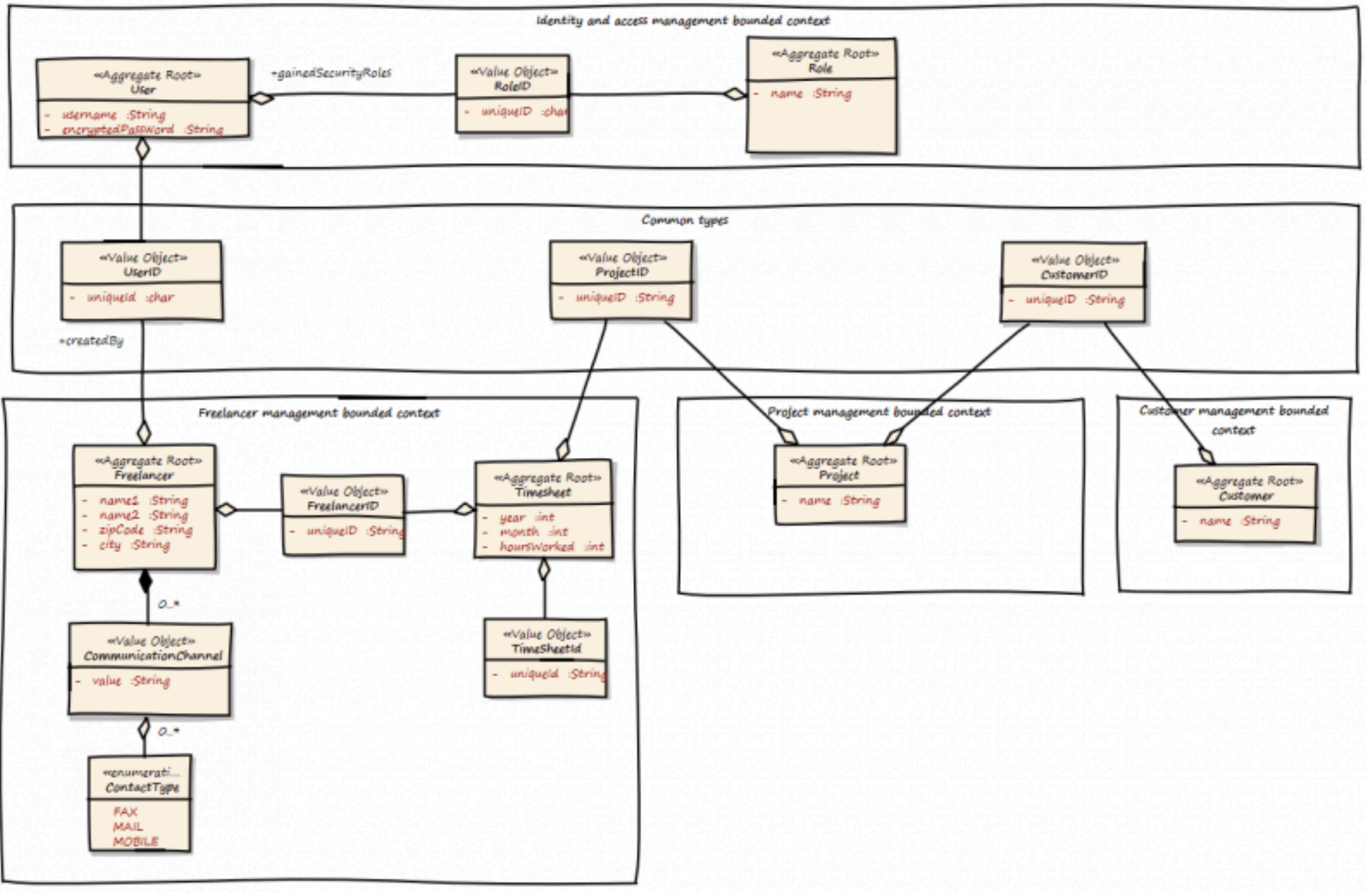
A searchable catalog of Freelancer

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


Identity and access management bounded context
«Aggregate Root»
User
- username :String
- encryptedPassWord :String
+gainedSecurityRoles
«Value Object»
RoleID
- uniqueID :char
«Aggregate Root»
Role
- name :String
Common types
«Value Object»
UserID
- uniqueId :char
+createdBy
«Value Object»
ProjectID
- uniqueID :String
«Value Object»
CustomerID
- uniqueID :String
Freelancer management bounded context
«Aggregate Root»
Freelancer
- name1 :String
- name2 :String
- zipCode :String
- city :String
«Value Object»
FreelancerID
- uniqueID :String
«Aggregate Root»
Timesheet
- year :int
- month :int
- hoursWorked :int
0..*
«Value Object»
CommunicationChannel
- value :String
«Value Object»
TimeSheetId
- uniqueId :String
0..*
«enumerati...
ContactType
FAX
MAIL
MOBILE
Project management bounded context
«Aggregate Root»
Project
- name :String
Customer management bounded context
«Aggregate Root»
Customer
- name :String

«Presentation»
Client

«use»

Transaction boundary

«Application Service»
FreelancerApplicationService

+ freelancerMoved(FreelancerMovedCommand) :void
+ getPresentationModelFor(FreelancerId) :FreelancerPresentationModel

Invokes usecase on

«Value Object»
Address

- name1 :String
- name2 :String
- zipCode :String
- city :String

+ getName1() :String
+ getName2() :String
+ getCity() :String
+ getZipCode() :String
+ changeName1(String) :Address
+ changeName2(String) :Address
+ changeCity(String) :Address
+ changZipCode(String) :Address

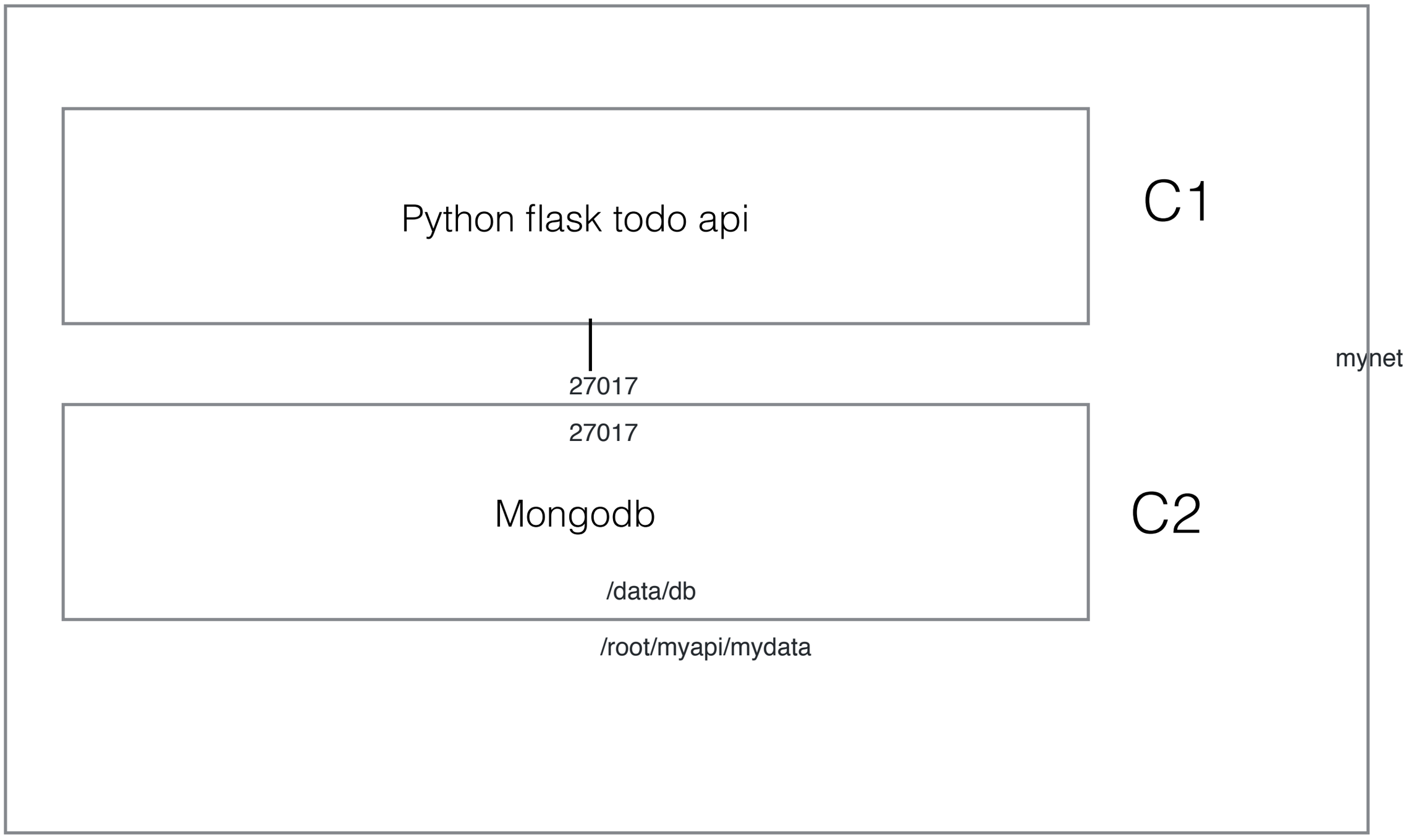
«Aggregate Root»
Freelancer

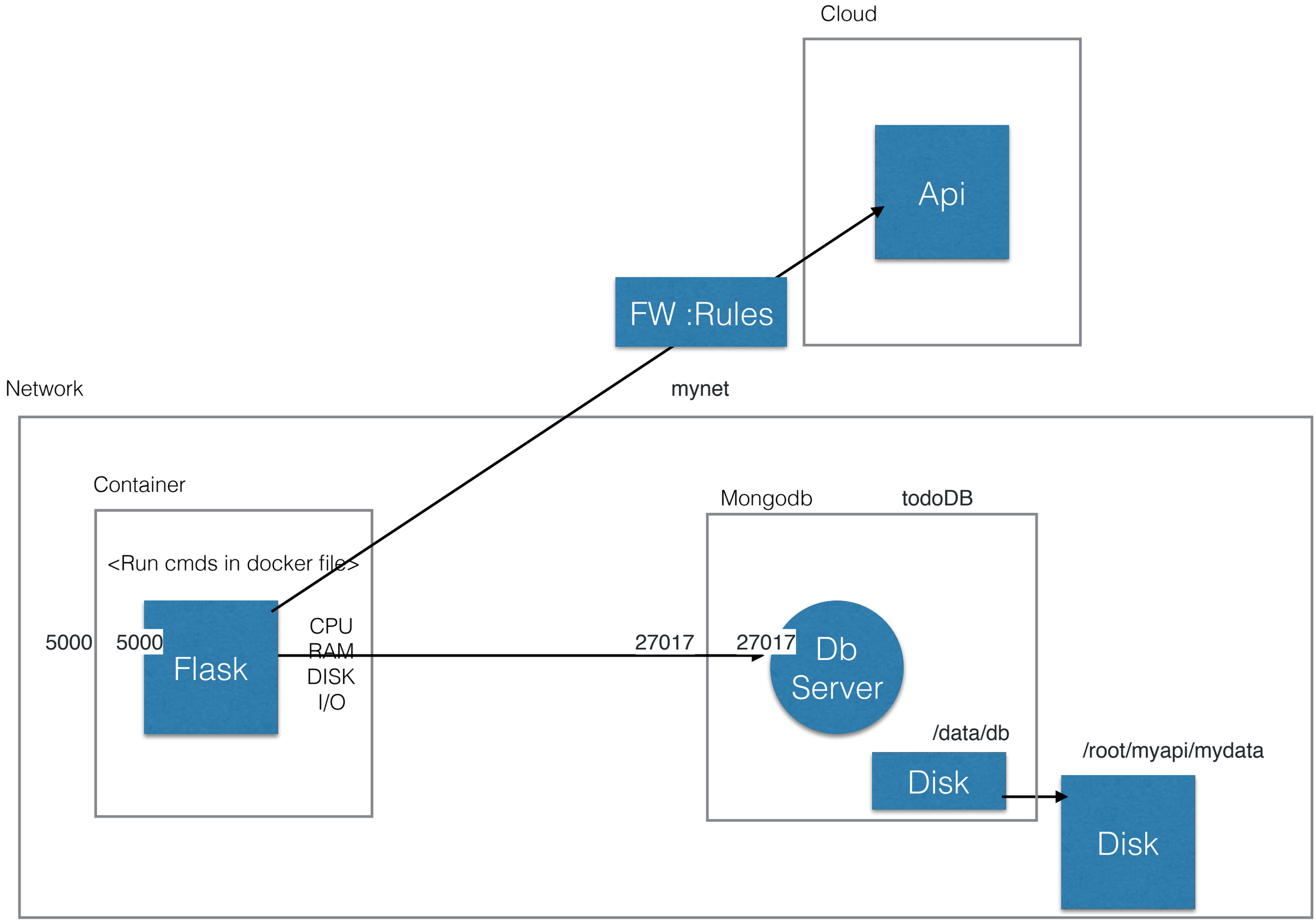
+ getAddress() :Address
+ movedTo(Address) :void

«Domain Repository»
FreelancerRepository

+ findById(FreelancerId) :Freelancer
+ add(Freelancer) :void
+ remove(Freelancer) :void
+ findByName(String) :Collection<Freelancer>

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


Python flask todo api
C1
mynet
27017
27017
Mongodb
/data/db
C2
/root/myapi/mydata


Cloud
Api
FW :Rules
Network
mynet
Container
<Run cmds in docker file>
5000
5000
Flask
CPU
RAM
DISK
I/O
Mongodb
todoDB
27017
27017
Db
Server
/data/db
Disk
/root/myapi/mydata
Disk

- Docker File (cmd to execute in the deployed env)
- Image (installer/ setup/war/ package)
- Container (small vm)
-

Load Balancer

NodePort Ip Address

Cluster Ip Address

Service

Ip Address

Pod

Container

My Api : v1

Ip Address

Pod

Container

My Api :v2