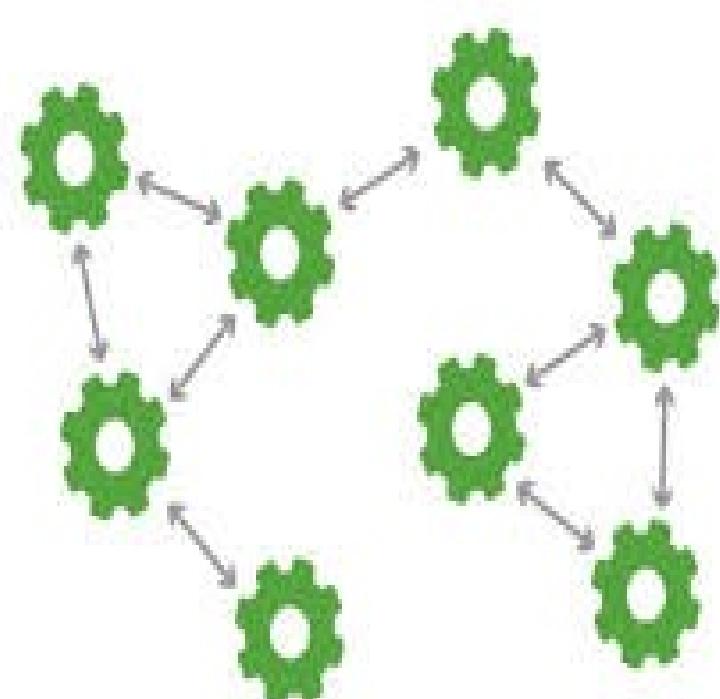


Cloud Computing with Micro Services Architecture

Whoami.json

```
{  
  "professor": {  
    "name": "Dr. Kamlendu Pandey",  
    "website": "vnsquit.org",  
    "city": "Surat, India",  
    "bio": "Educationist, Java Enthusiast, Spiritual",  
    "work": {  
      "company": "VNSGU",  
      "role": "Professor, Coordinator"  
      "sub": "cloud computing"  
      "topic": "micro services architecture"  
    }  
  },  
  "links": [  
    "vnsquit.org",  
    "vnsgu.ac.in"  
  ]  
}
```

MICROSERVICES ARCHITECTURE



What is a Software Architecture



Software architecture refers to the **fundamental structures or methodology** of a software system and the **discipline** of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations.

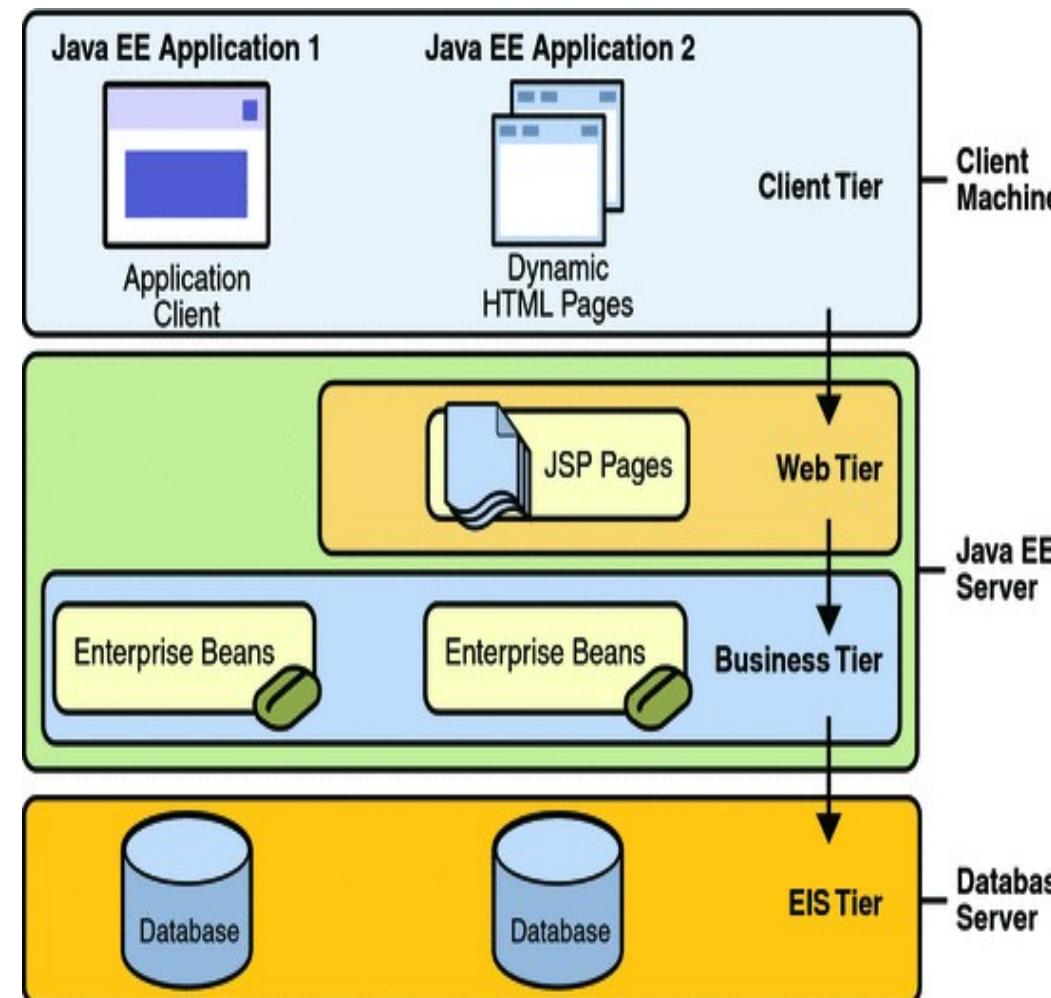
Types of Software Architectures

- Old Monolithic (COBOL, FoxPro)
- Blackboard (AI Problems)
- Client-Server (Data and Web Apps, Visual Basic)
- Component Based (Over the shelf-COM, Beans etc)
- Distributed (EJBs, DCOM, CORBA)
- Model-View-Control
- Layered- CS (Enterprise Java)
- Service Oriented (SOA)
- REST (http Methods based Web Services)
- Micro Services (Microprofile, Springboot)

What We have been doing till now?

We have been developing developing applications using

- Java EE Layered Architecture
- Distributed Components
- Messaging Infrastructure
- SOAP and REST communication Pattern
- A UI/UX using JSF/JSP
- MVC Design Pattern
- Central Security Control and Propagation
- JNDI
- Application Servers
- CDI

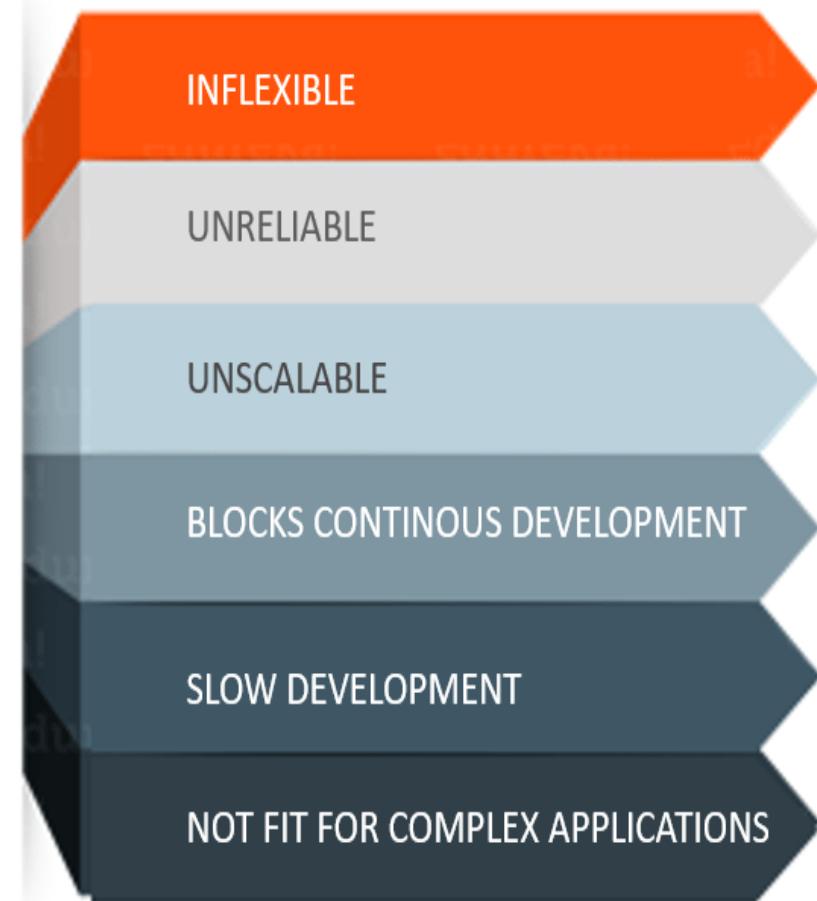
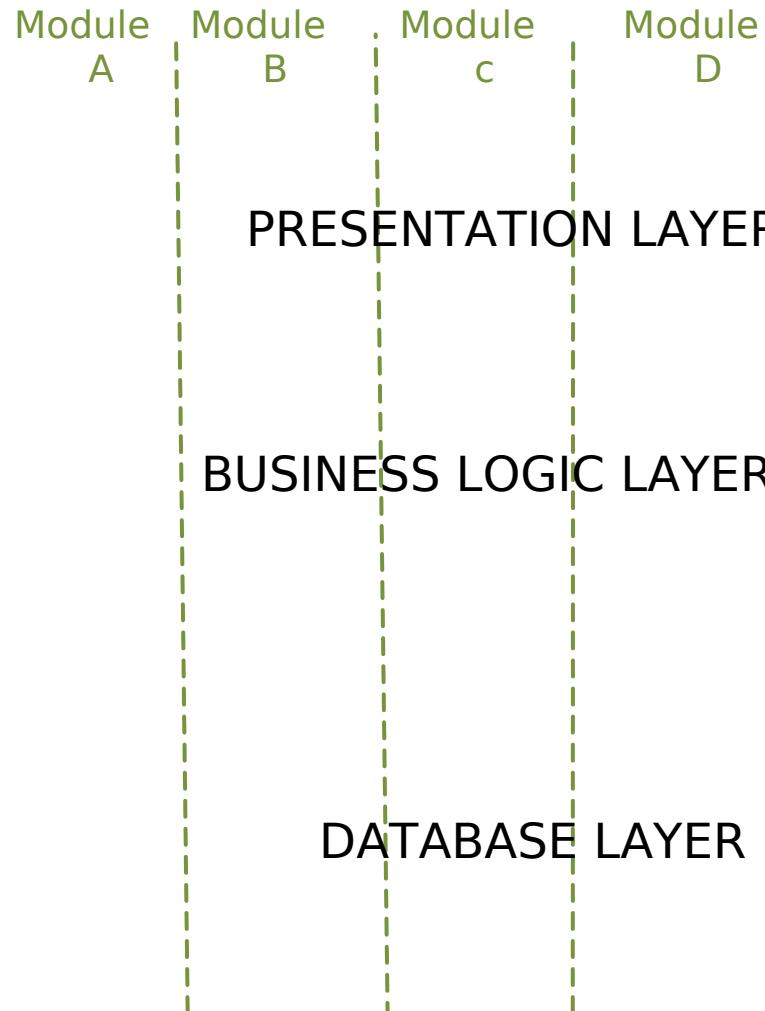


So What is wrong in this ?

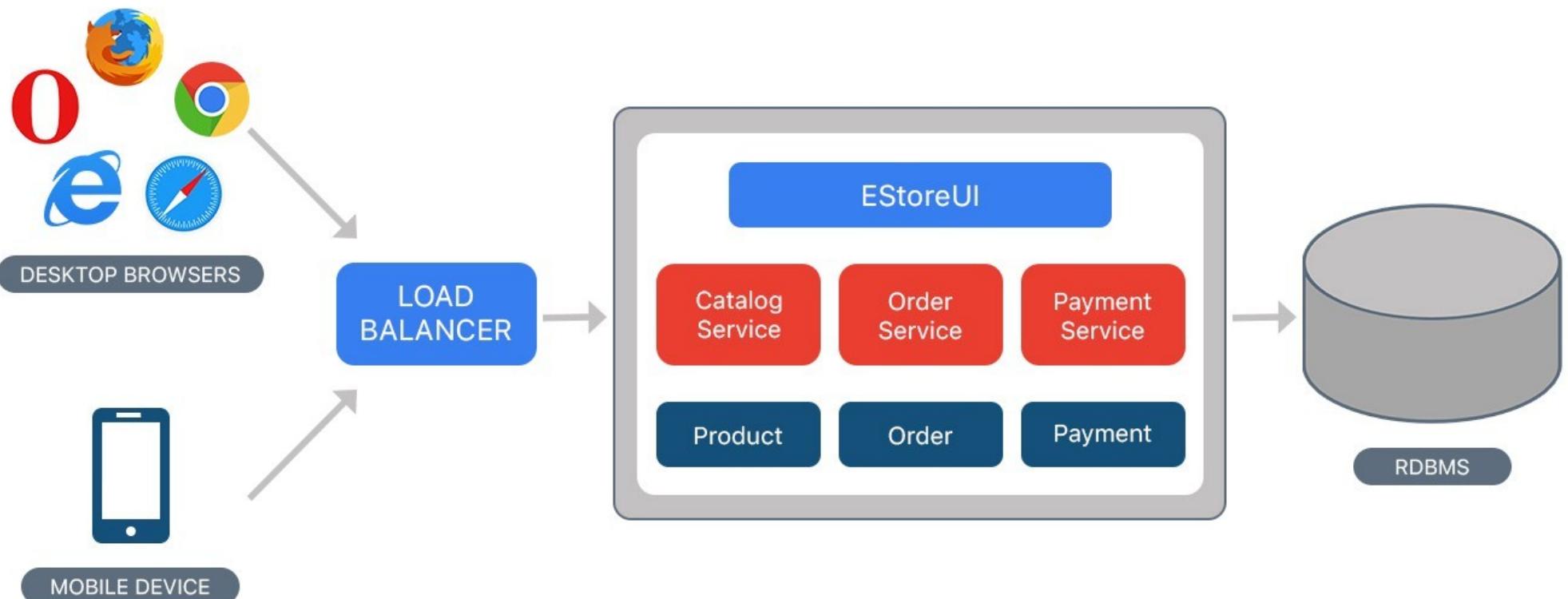
Well Nothing wrong as such, but it may not meet the current expectations and demands of performance industry and its users. The reasons are

- All the software layers like BL, UI and Data Layer are packed into one Enterprise Application (jar, war, ear)
-
- The Entire Application is deployed on a production server.
- As requirement grows the application changes and it is undeployed and deployed many times. (Interruption to services)
- The old team may leave and new team joins. Loses track of logic and thought process
- When Load increases scaling becomes a difficult issue
- Dissatisfied customers and painstaking development

They are still Monoliths



And This is how they work

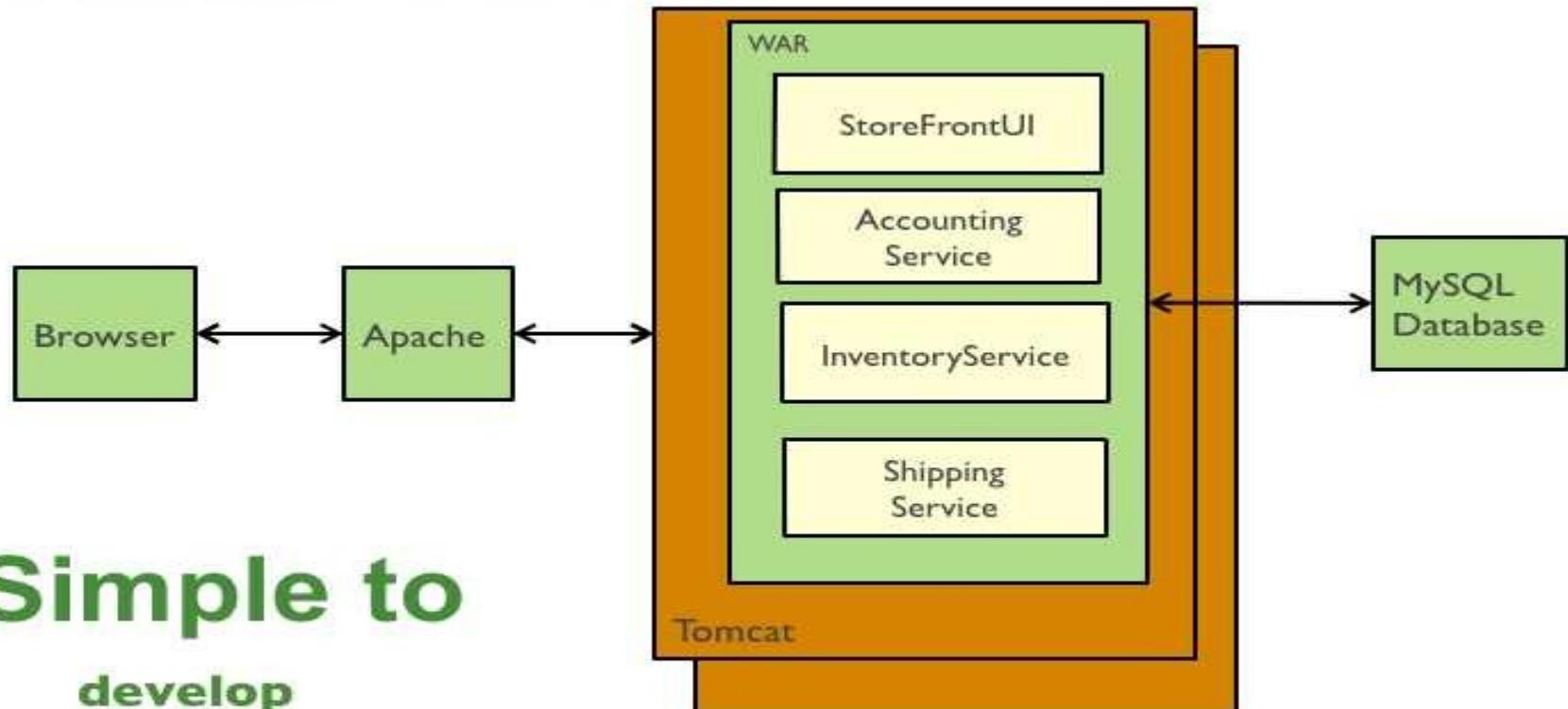


Limitations of Monolithic Applications

- A simple elegant monolith will grow into a uncontrollable beast as LOC grows exponentially with more client stories
- Over the time impossible to understand by single developer
- Sheer size slows down development and increases startup time on server. This is a problem when application server is frequently restarted
- Continuous development and delivery is a unthinkable in complex monolith
- Difficult to scale as different modules have different memory requirement (A cpu intensive image processing and an inline db require different vms on amazon)
- Reliability is big issue as a small bug may effect whole project and instances
- Difficult to adopt new smart incoming frameworks. It becomes white elephant over the time

An Example

Traditional web application architecture



Simple to

**develop
test
deploy
scale**

When the time goes by. The Problems which appear

- Overloaded IDE
- Overloaded web container
- Continuous deployment is difficult
- Scaling the application can be difficult
- Obstacle to scaling development
- Requires a long-term commitment to a technology stack

So, What is the solution ?

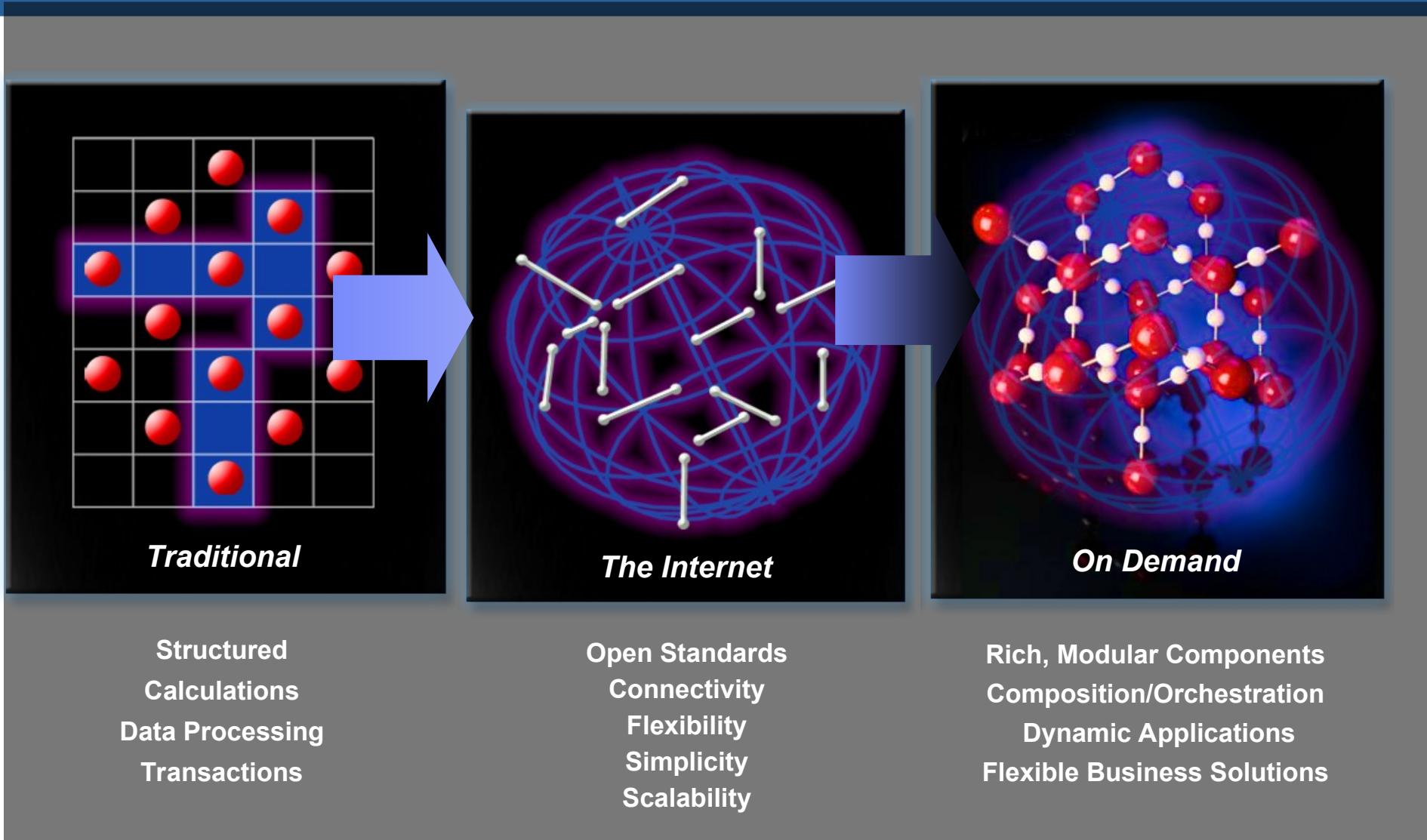
We shall look for the solutions which can overcome the existing issues of monolith and allows

- Smart Decomposition of Application Domain
- Small elegant independent services capable of being deployed and scaled independently
- Support of Continuous Integration and Delivery
- Supports Services Discovery
- Support orchestration and choreography
- Support API gateways

The Solution 1 : Service Oriented Architecture



The information technology industry



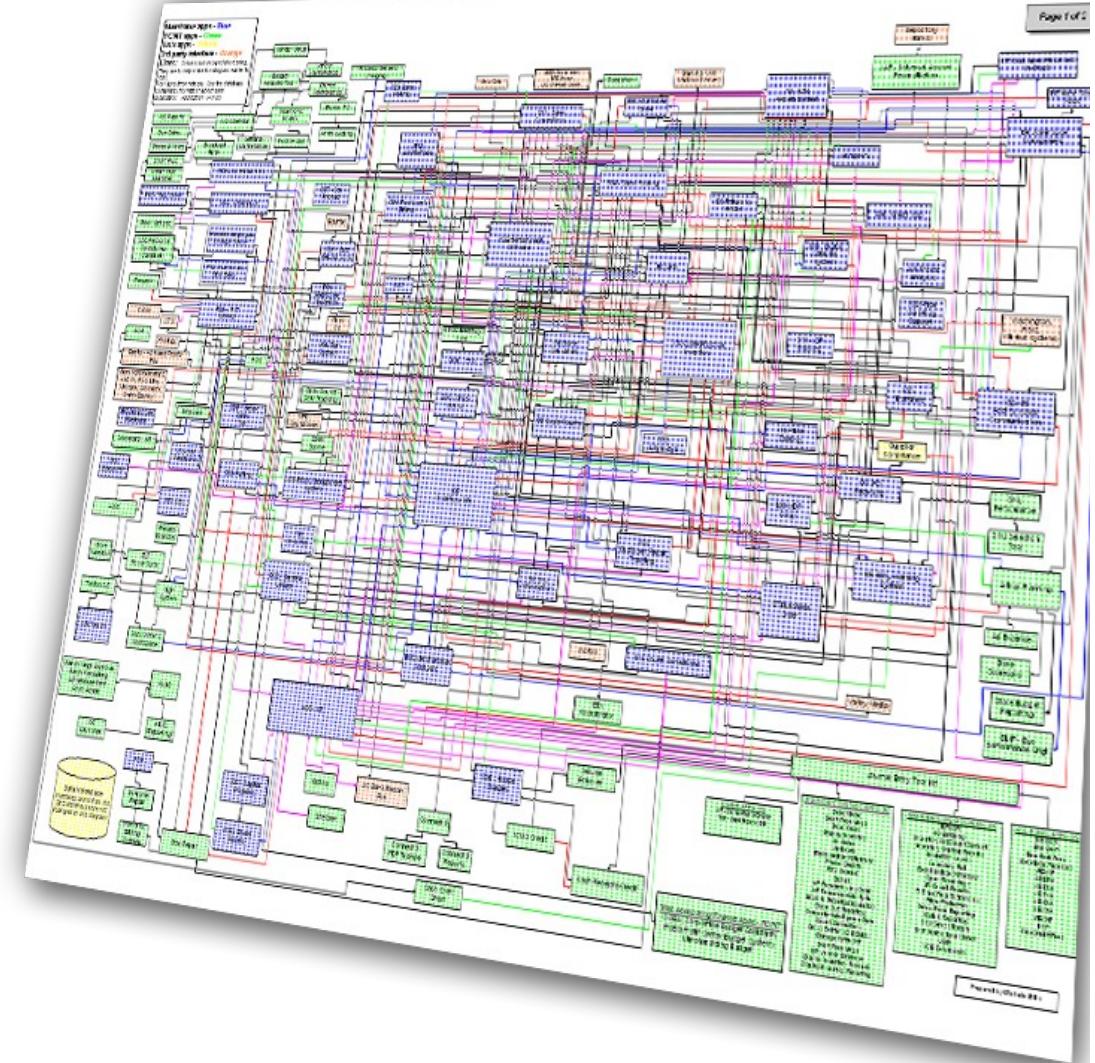
Moving to a new state...

On Demand Business

An enterprise whose **business processes — integrated end-to-end** across the company and with key partners, suppliers and customers — can **respond with flexibility and speed** to any customer demand, market opportunity or external threat.

What are the barriers to business flexibility and reuse?

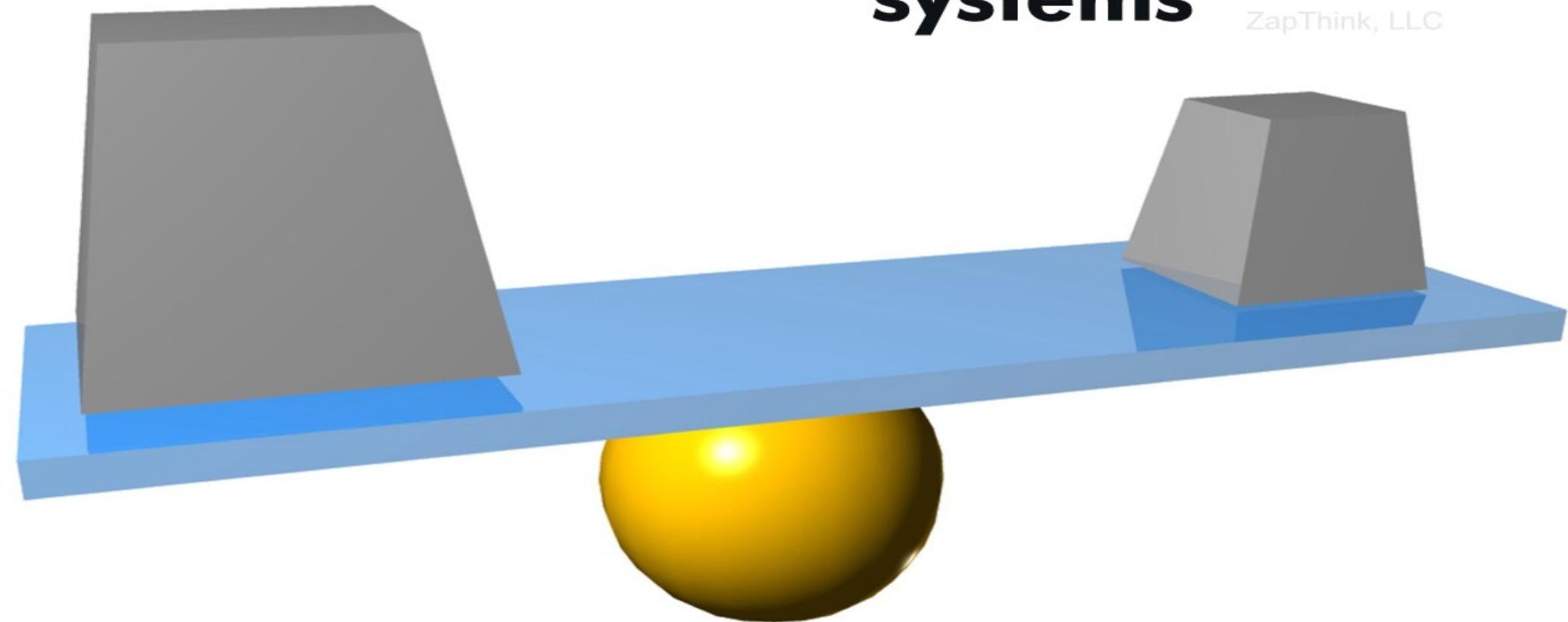
- Lack of business process standards
- Architectural policy limited
- Point application buys to support redundant LOB needs
- Infrastructure built with no roadmap



The Cost of Business Integration ...

70% of IT development budgets are spent on integrating different systems

ZapThink, LLC

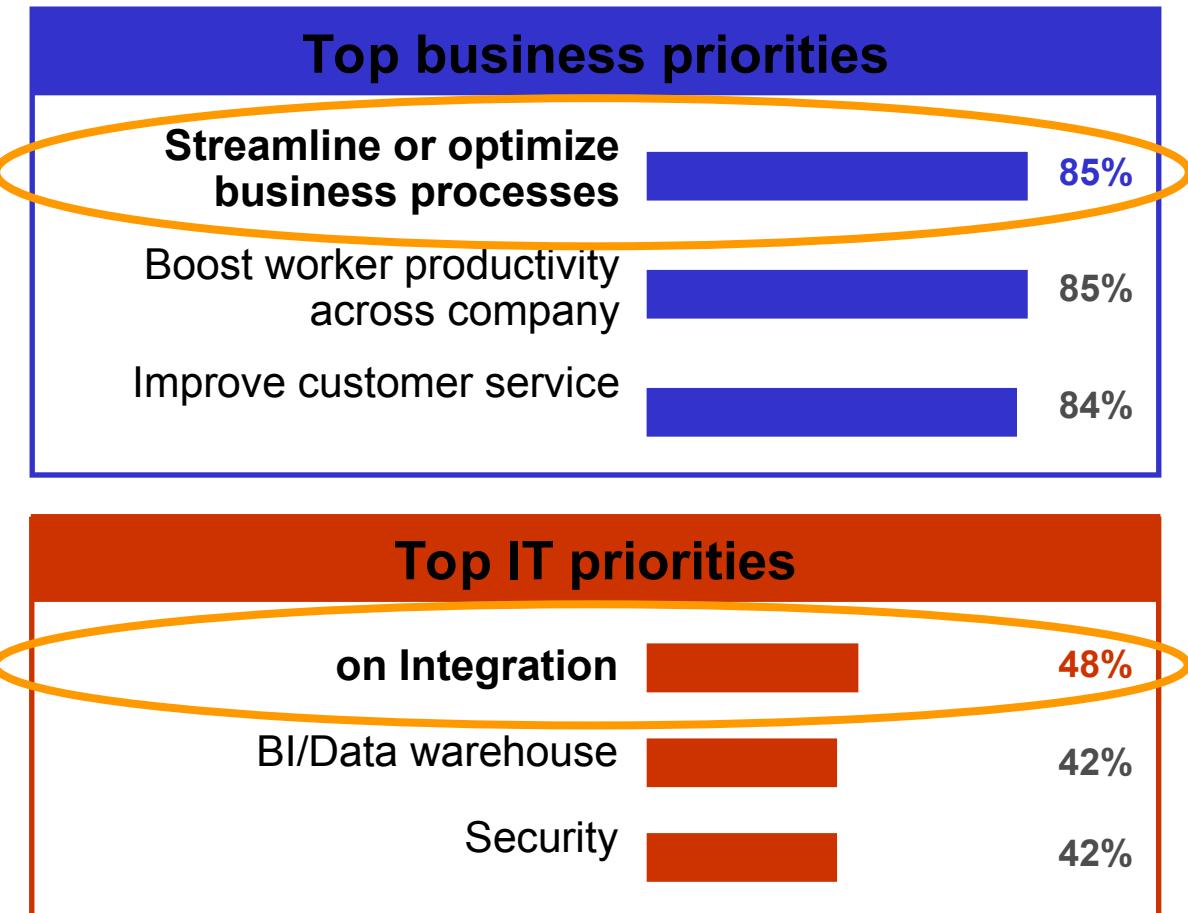


Changing Market Conditions Drive On Demand Business Needs: Addressing Top Business and IT Priorities

“More than 80% of CEOs see unpredictable market forces as the key inhibitor to growth.”

Source: IBM's Global CEO Survey, February 2004

- Economic volatility and globalization
- Increasing consolidation across industries
- Increasing regulations and industry standards
- Technical realities



Sources: *Outlook 2004: Priorities 1Q InformationWeek Research, January 2004;*
Merrill Lynch CIO Survey Results, September 2004

The Challenge: Business and IT Alignment

Business Pressures:

Launch new and innovative products

Shorter change cycles

Customized products for niche markets

Partner



Customers



Channels

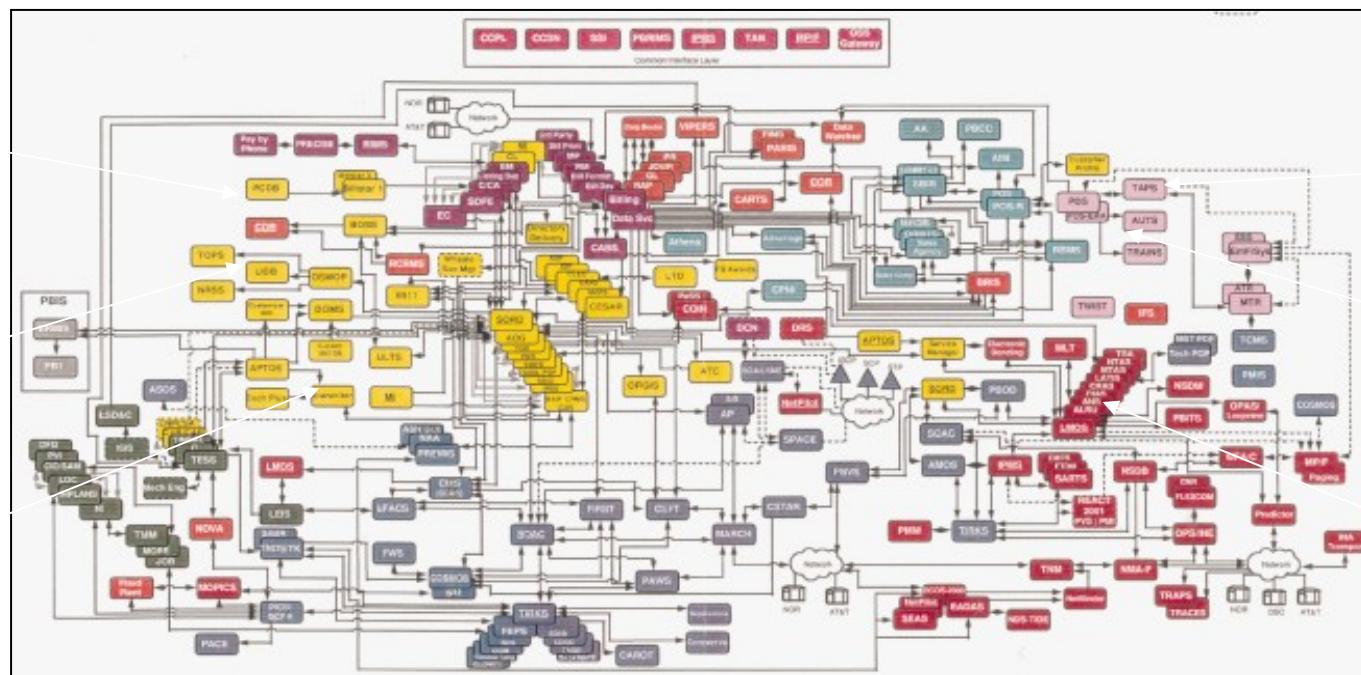


IT Constraints:

Complex processes and systems

Complex applications and interfaces

IT budget priorities on maintenance, not new investments



Employees



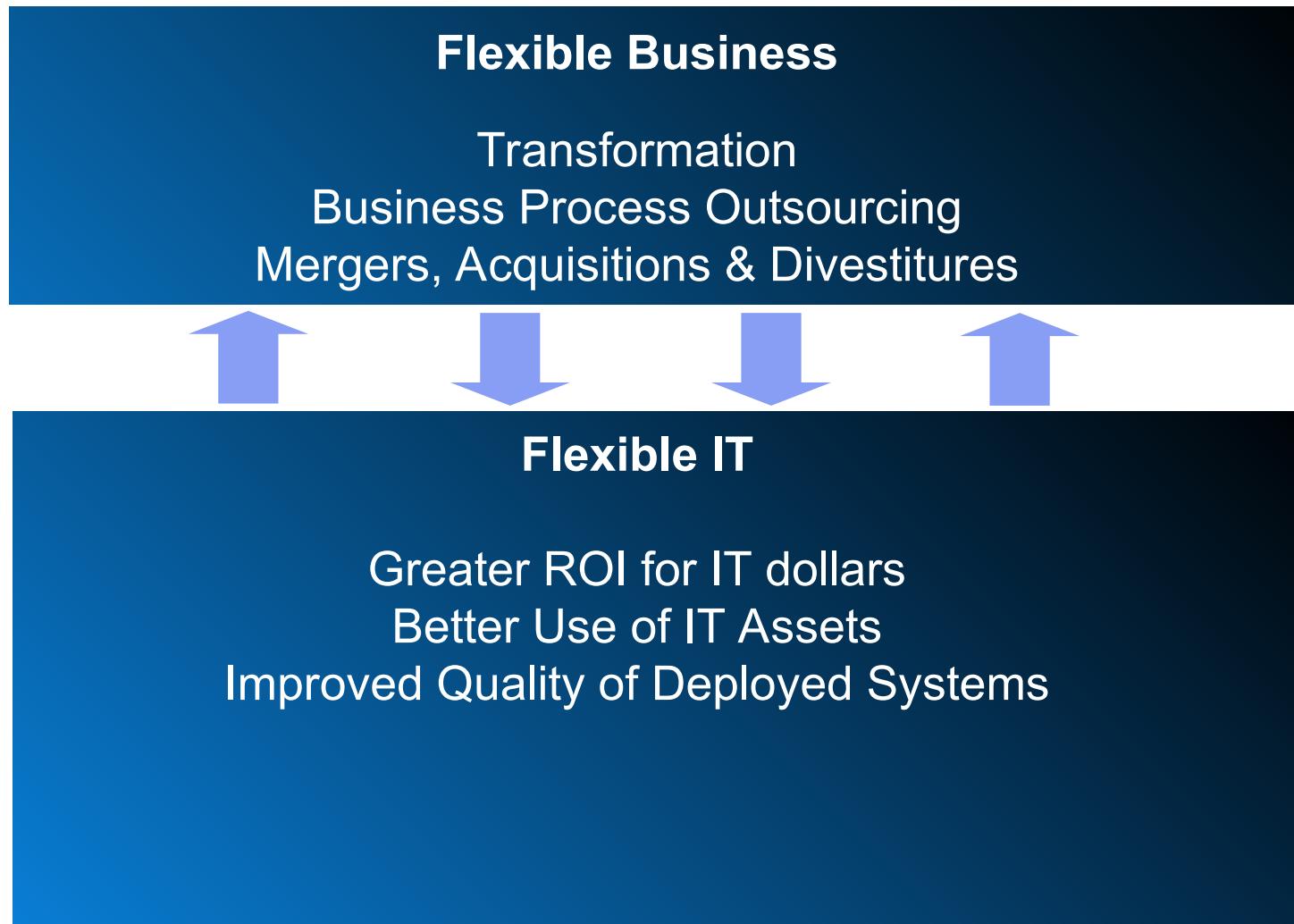
Suppliers



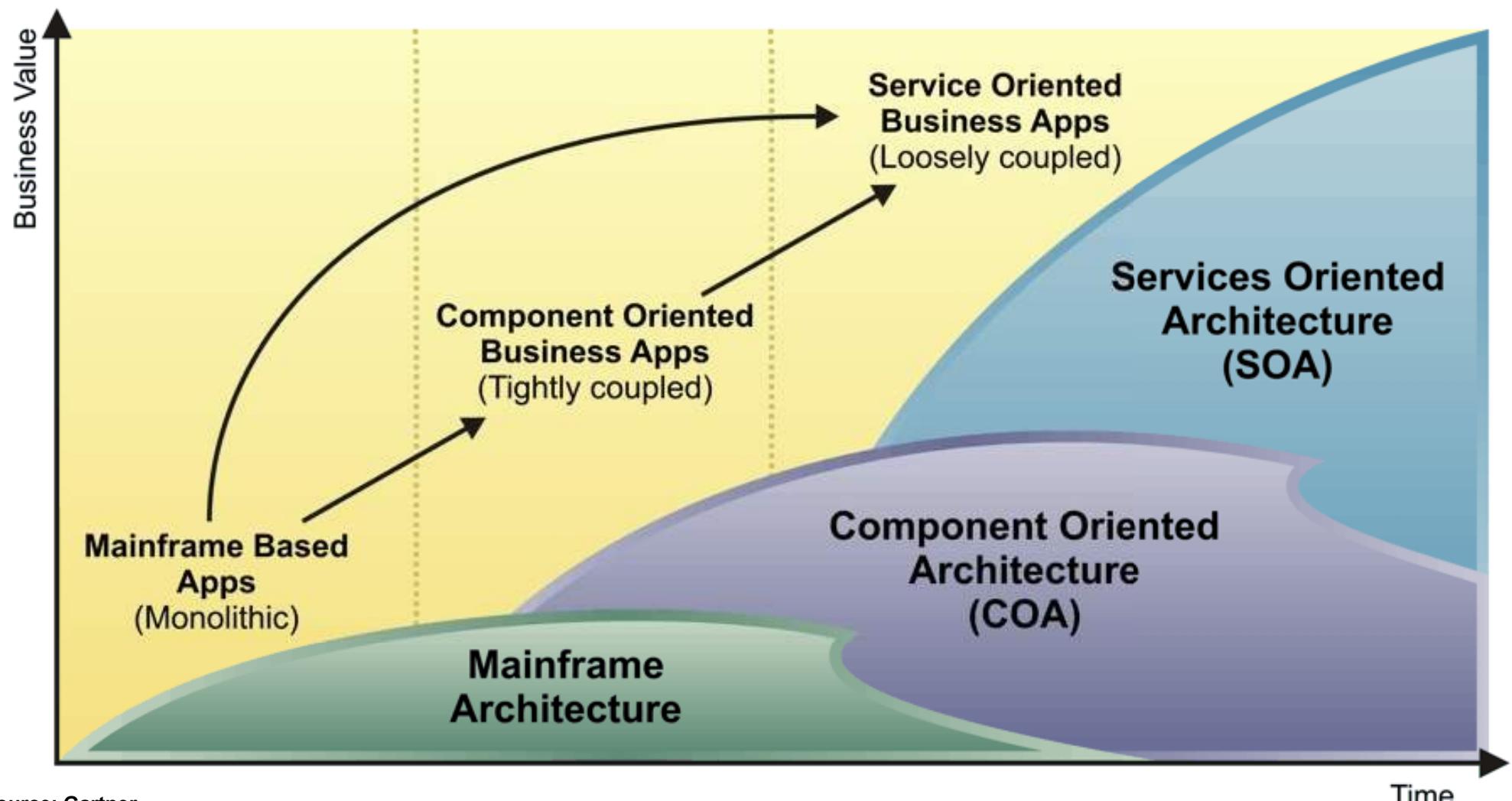
3rd Party Providers



Whatever you want are key drivers for SOA



SOA Represents the Next **Evolutionary Step** to Improve Business Agility and Flexibility



What is SOA?

... a service?

A **repeatable business task** – e.g., check customer credit; open new account

... service orientation?

A way to integrate your **business as linked services** with their unique outcomes

... service oriented architecture (SOA)?

An **IT architectural style** that supports service orientation

... a composite application?

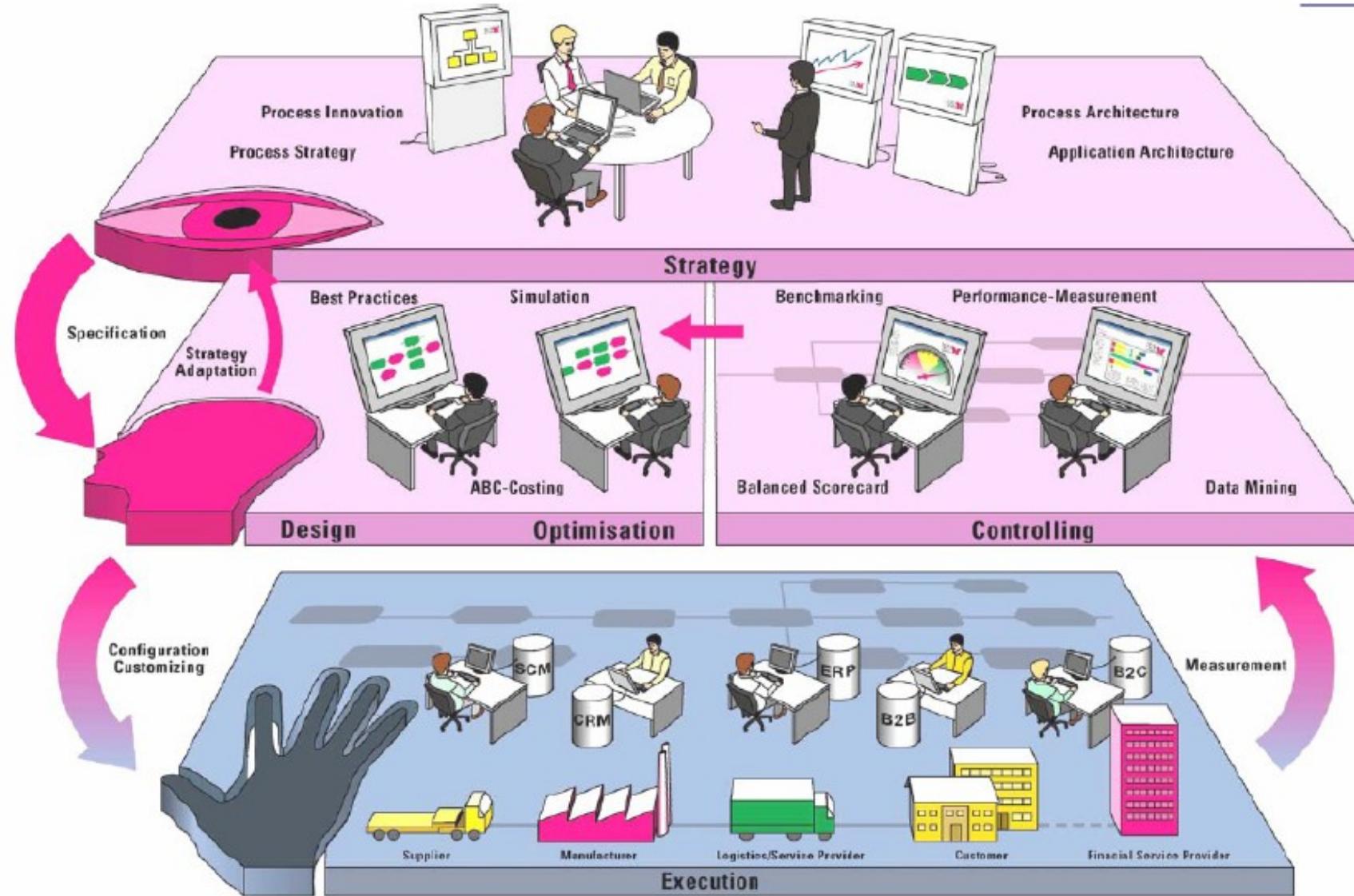
A set of **related & integrated** services that support a business process built with SOA



Key terms for SOA

- A *service* is representative of a repeatable business task. Services are used to encapsulate the functional units of an application by providing an interface that is well defined and implementation independent. Services can be invoked (consumed) by other services or client applications.
- *Service orientation* defines a method of integrating business applications and processes as linked services. we're talking about a thought process and a business philosophy not about technology.

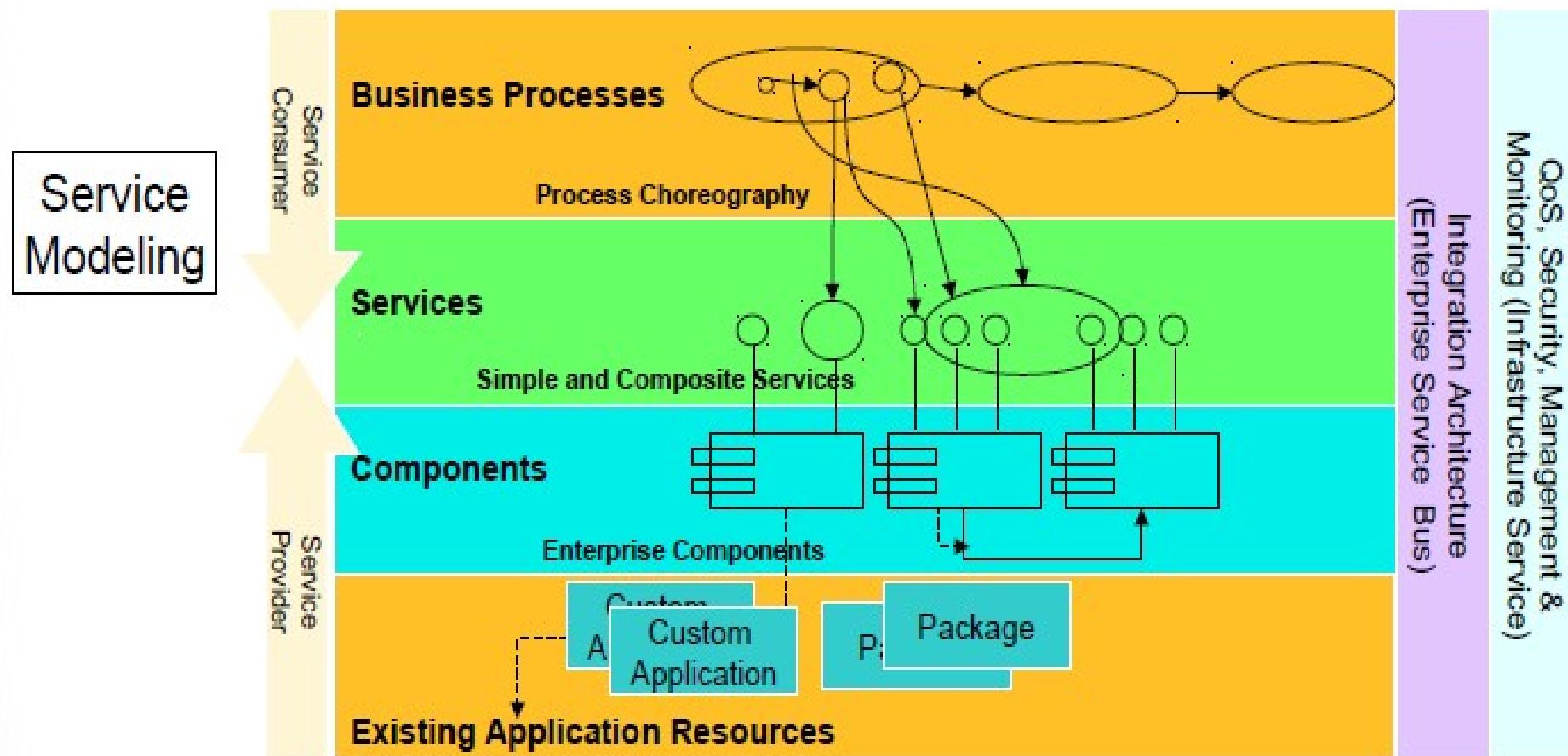
How Business Works?



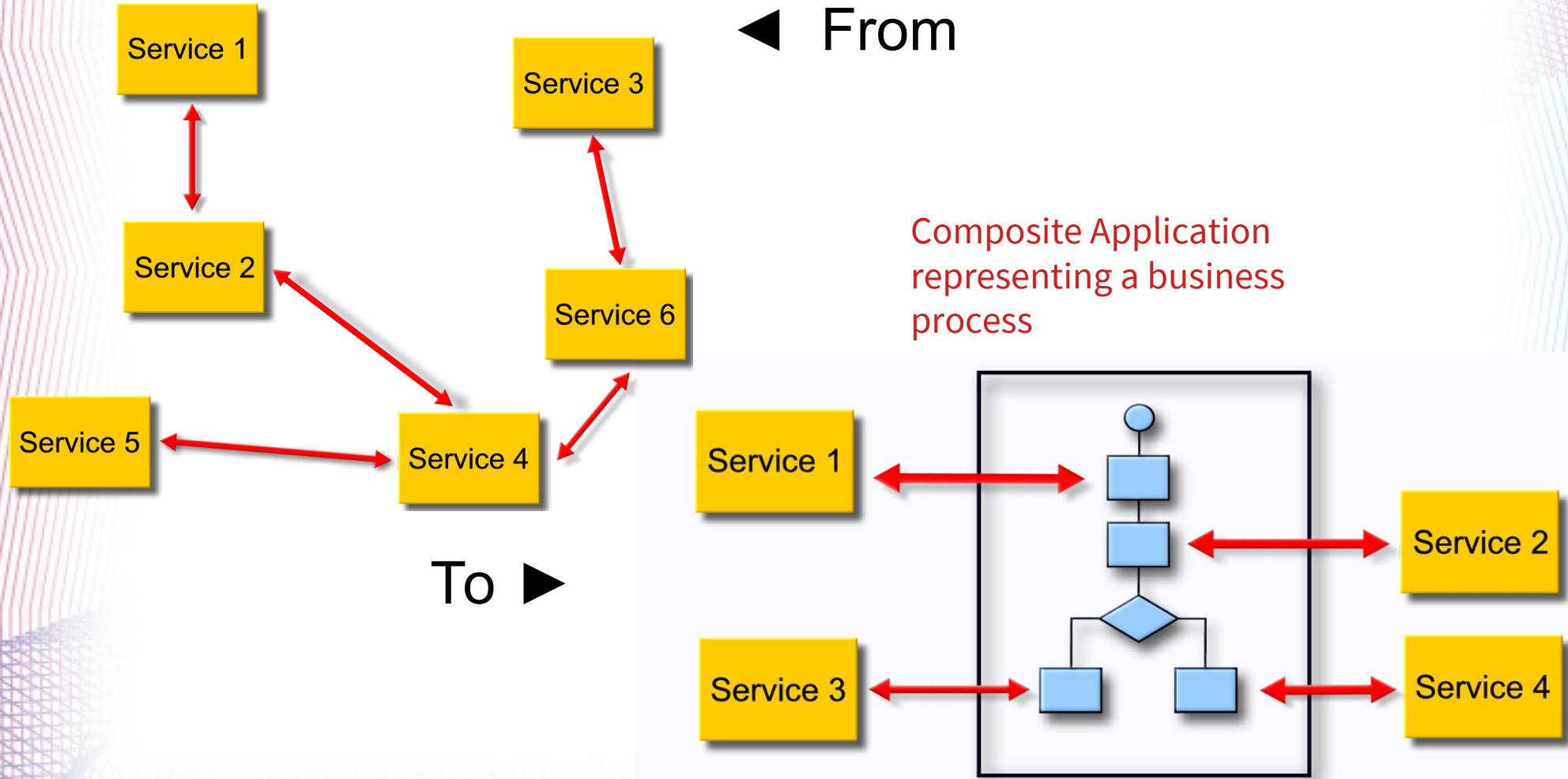
Service-Oriented Solutions – Architecture View

An SOA is composed of multiple layers.

At the heart of the SOA are Services, Components that realize services and Service Flows.



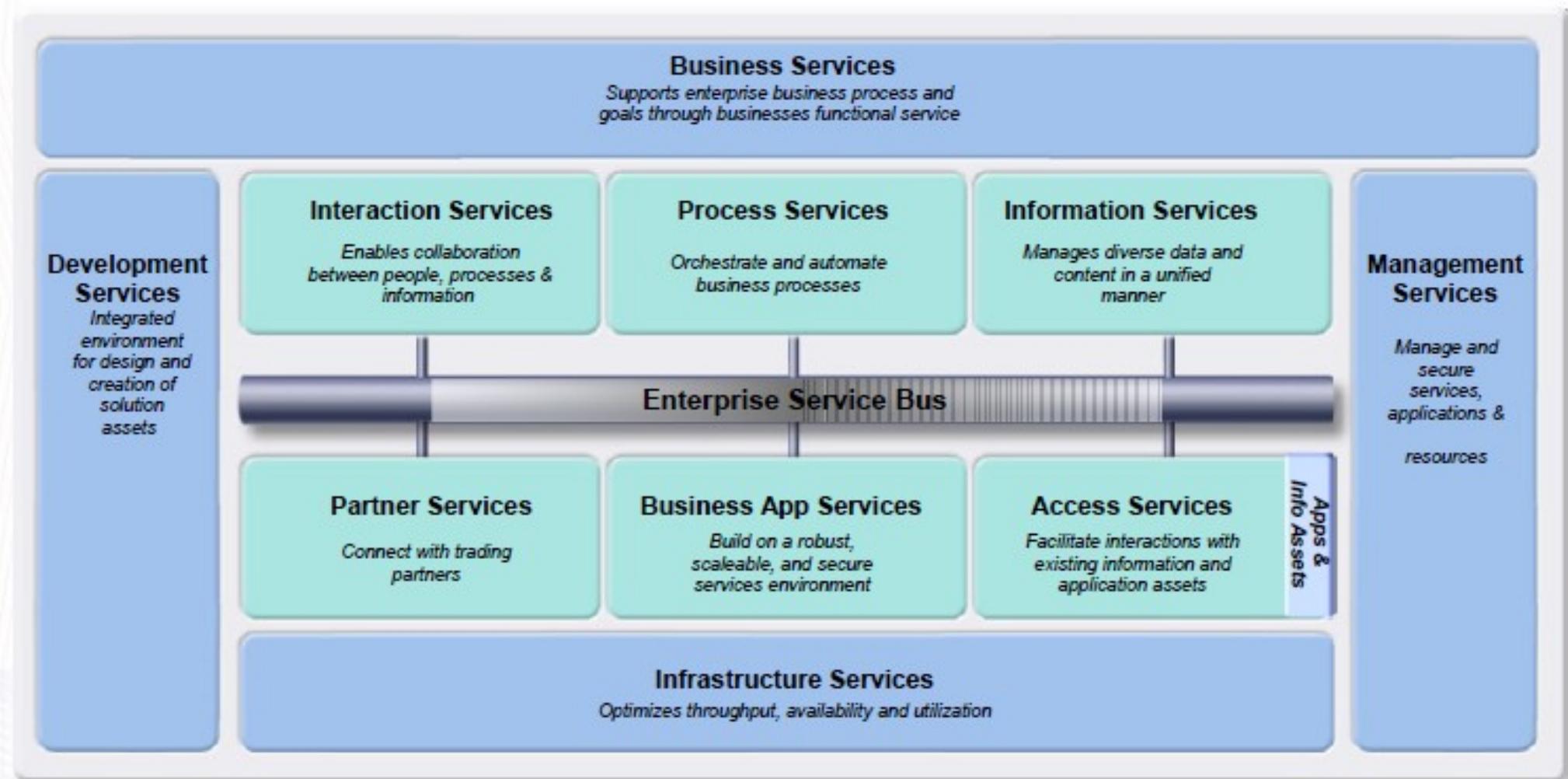
The Choreography Challenge with Composite Application...



Key Feature - Enterprise Service Bus

Defining the capabilities for your SOA environment

SOA Reference Architecture



Key Features of Enterprise Service Bus

- Messaging from one service to other
- Routing
- Centralised Security
- Profiling
- Metering
- Message Transformation on Fly
- Protocol Independency

View performance and modify dashboards in real time

- Scorecard view implemented through Key Performance Indicators
- Track and modify business process flows
 - Eliminate redundancies or inefficiencies
 - Identify bottlenecks – balance workloads
 - Reduce latencies
- View information the way you want to see it
 - Management dashboards and reporting capabilities,
 - trending information
 - Tools to customize or define new dashboards
- Monitor different perspectives of business process metrics
 - Cost, time, resources



SOA Benefits

- Saves money, time, and people
- Eliminates frustrations with IT
- Justifies IT investments
- Provides business executives with a clear understanding of what IT does and its value
- Eliminates IT's 6-6 answer (that is, the project will take 6 months and cost 6 figures)
- Provides a business and competitive differentiator

When a change in business process no longer requires a change to application programming logic, you have a successful SOA; your company has attained competitive business agility.

When SOA is beneficial?

An SOA could be the difference between the success and failure of the next:

- Department, intra-company, or inter-company merger
- Acquisition
- Divestiture
- Product or service rollout
- Business partner, customer, or supplier addition
- Geographical expansion
- Competitive onslaught



When not to implement SOA?

- When you have a homogeneous IT environment
- When true real-time performance (nanoseconds response times) is absolutely critical
- When flexibility is not needed
- When tight coupling is needed
- If the organization isn't ready for it



Pitfalls of SOA

- Highly Proprietary (Oracle, IBM) . Less stable open source products
- Relying on JAX-WS and XML standards which is obsolete now a days
- Provision of Enterprise Service Bus made it highly bulky and centralised
- Could not convince the community about its utility
- Scaling was a major issue

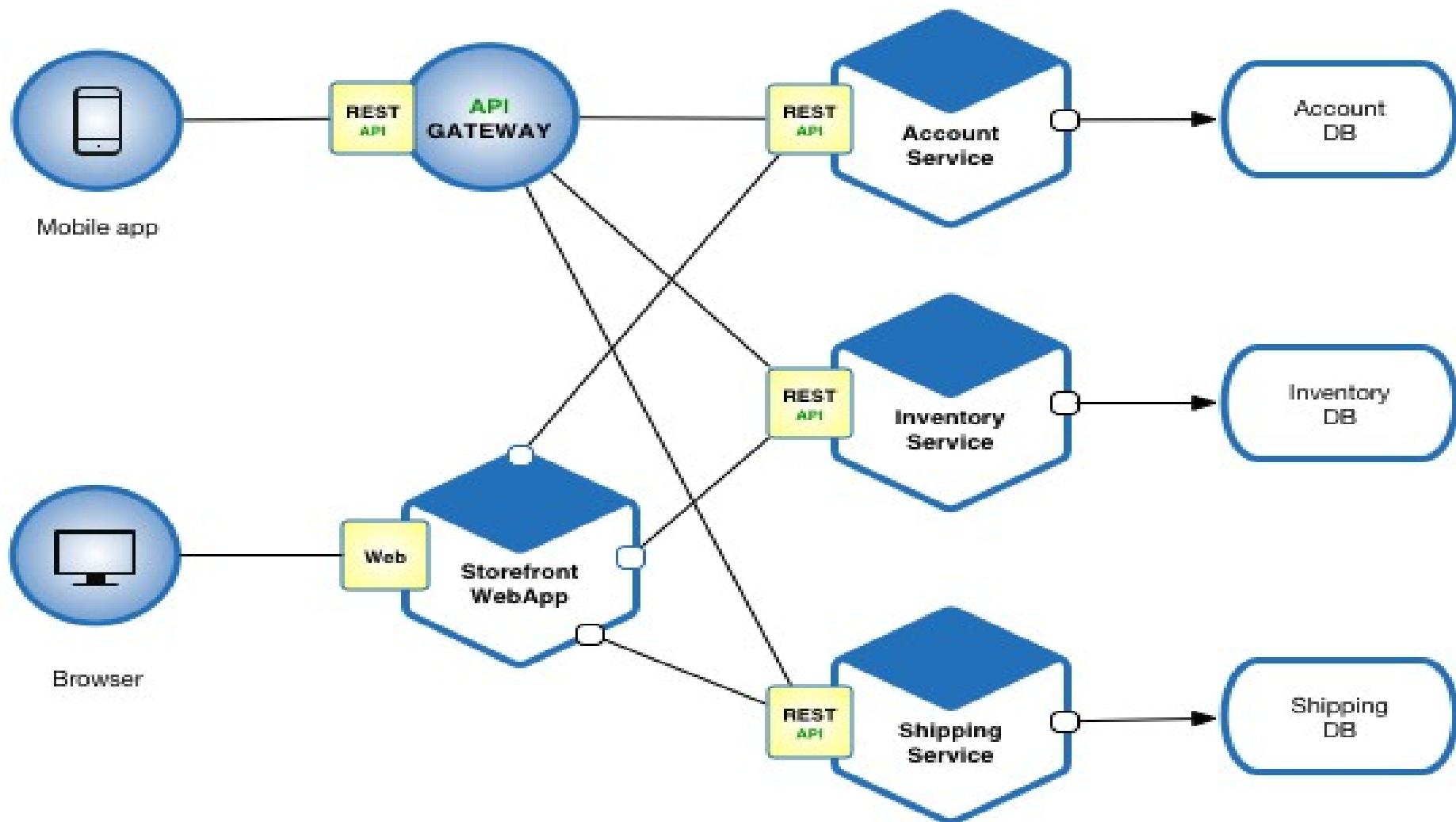
What we need now is

- There is a team of developers working on the application
- New team members must quickly become productive
- The application must be easy to understand and modify
- You want to practice continuous deployment of the application
- You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements
- You want to take advantage of emerging technologies (frameworks, programming languages, etc)

What we need now is

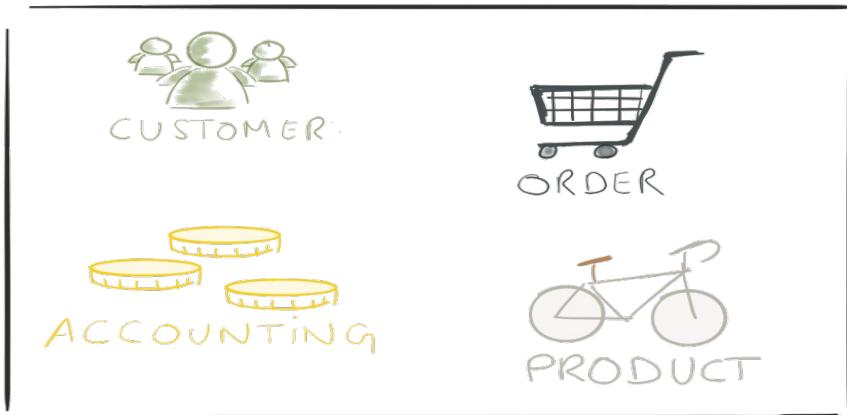
- There is a team of developers working on the application
- New team members must quickly become productive
- The application must be easy to understand and modify
- You want to practice continuous deployment of the application
- You must run multiple instances of the application on multiple machines in order to satisfy scalability and availability requirements
- You want to take advantage of emerging technologies (frameworks, programming languages, etc)

Solution 2- Micro Services Architecture



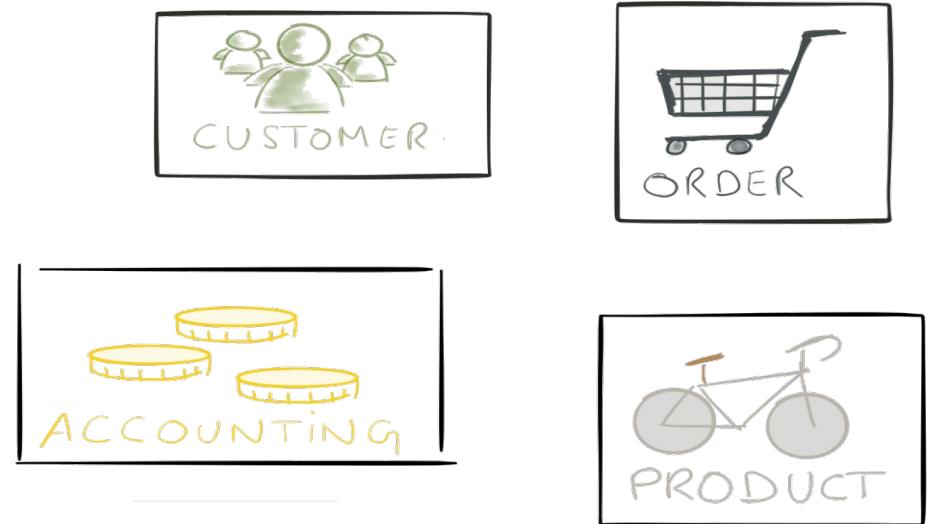
A Monolith Dicomposed into Microservices

VS



One Server
(Container)

monolith

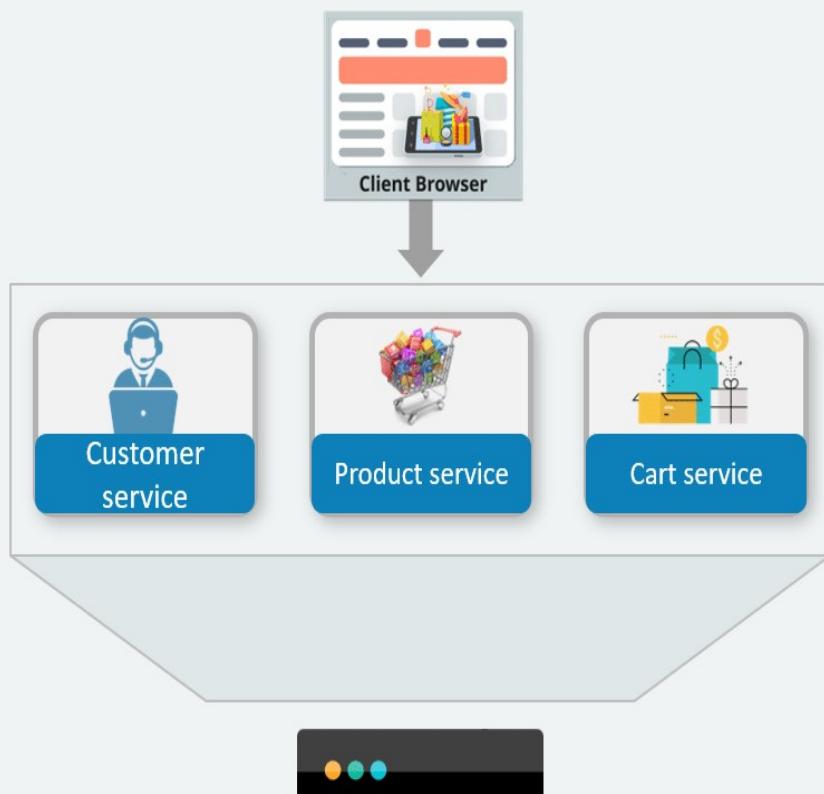


Multiple Container
Instances

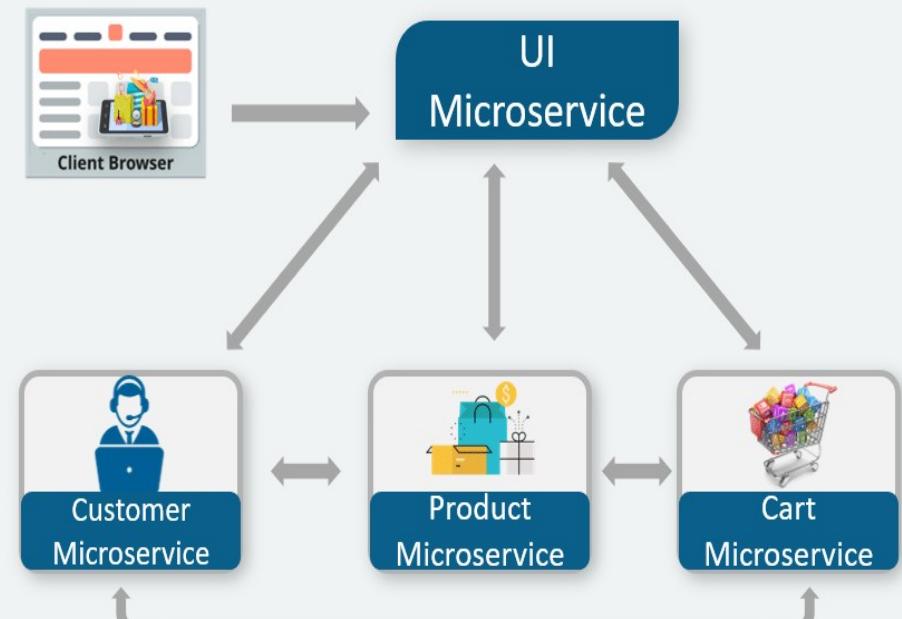
microservices

A Monolith Decomposed into Microservices

Monolithic Architecture

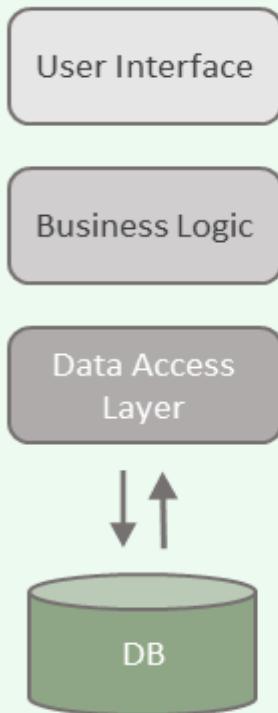


Microservice Architecture

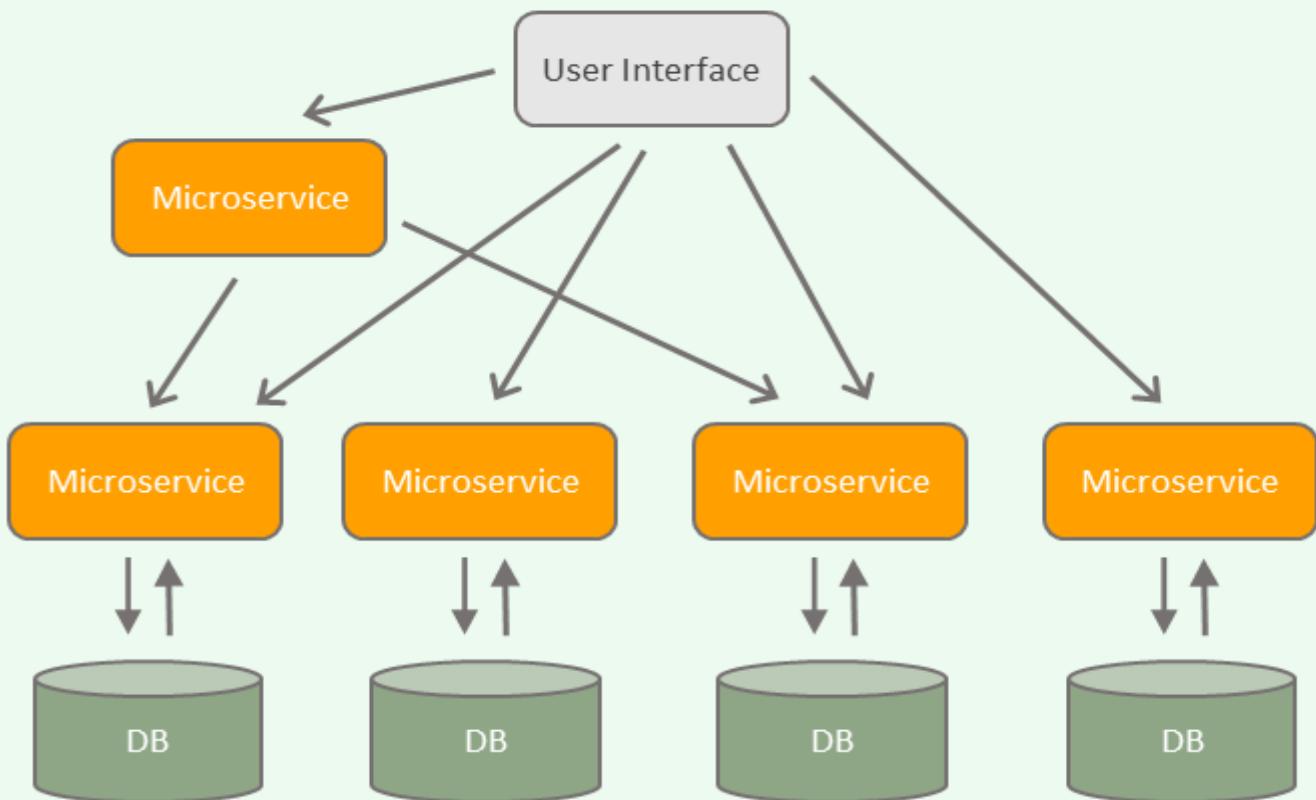


This architecture called - Micro Services Architecture

MONOLITHIC ARCHITECTURE

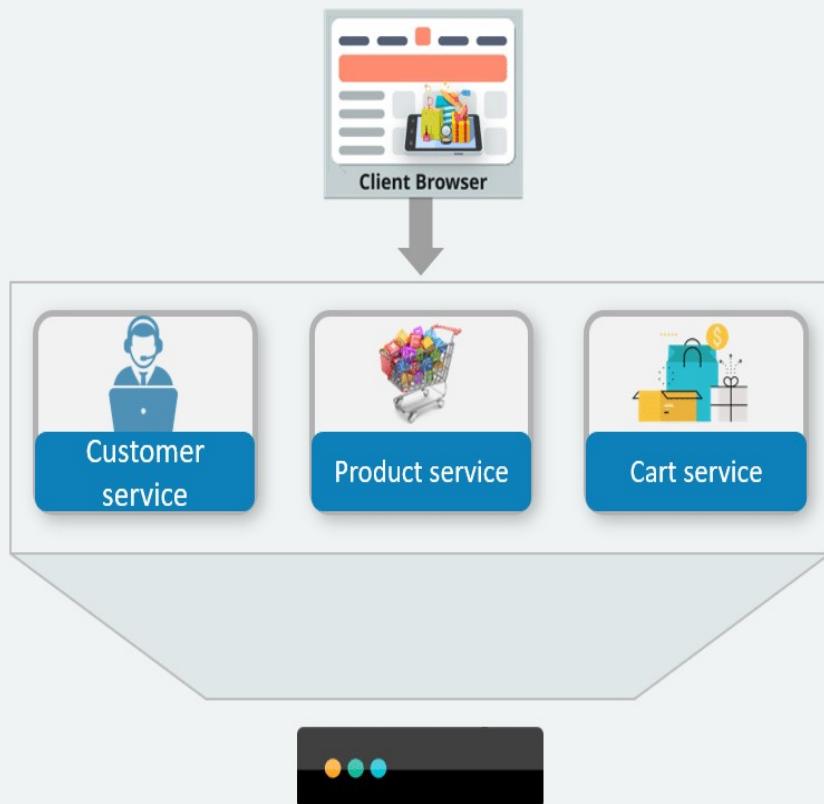


MICROSERVICES ARCHITECTURE

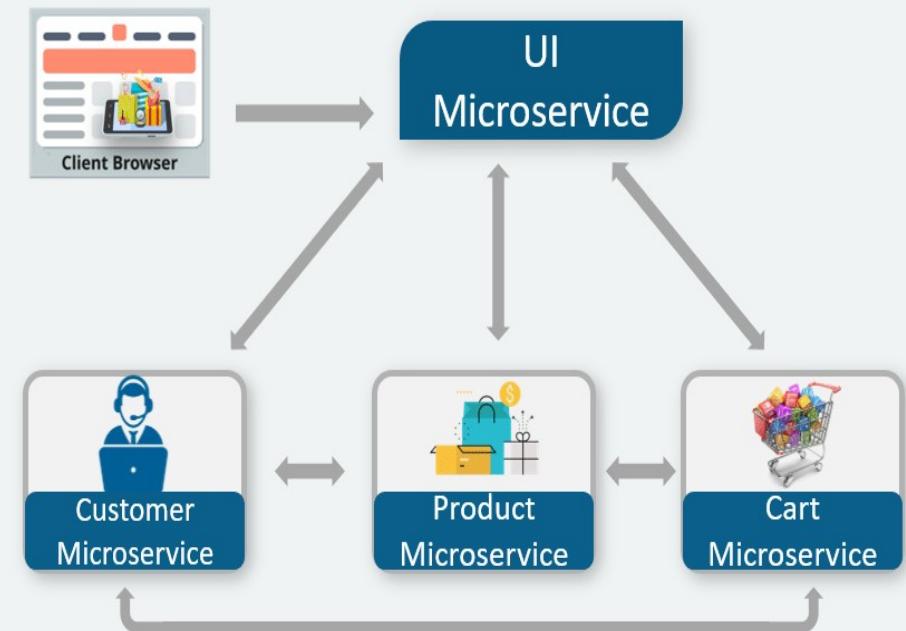


This architecture called - Micro Services Architecture

Monolithic Architecture



Microservice Architecture



Single Instance

22:12:42

We Realize microservices in Java by JAVA MICROPROFILE ARCHITECTURE

Eclipse java Microprofile has following features

- JAX-RS - REST
- CDI
- JSON-P Json
- Common Annotations
- Config
- Metrics
- Health Check
- Fault Tolerance
- JWT Authentication
- OpenTracing
- OpenAPI
- Rest Client

The Expectations are :

- Enables the continuous delivery and deployment of large, complex applications.
- Improved maintainability
- Better testability - services are smaller and faster to test
- Better deployability - services can be deployed independently
- Easier for a developer to understand
- The IDE is faster making developers more productive
- The application starts faster, which makes developers more productive, and speeds up deployments
- Improved fault isolation.
- Eliminates any long-term commitment to a technology stack

How to achieve this : The Steps

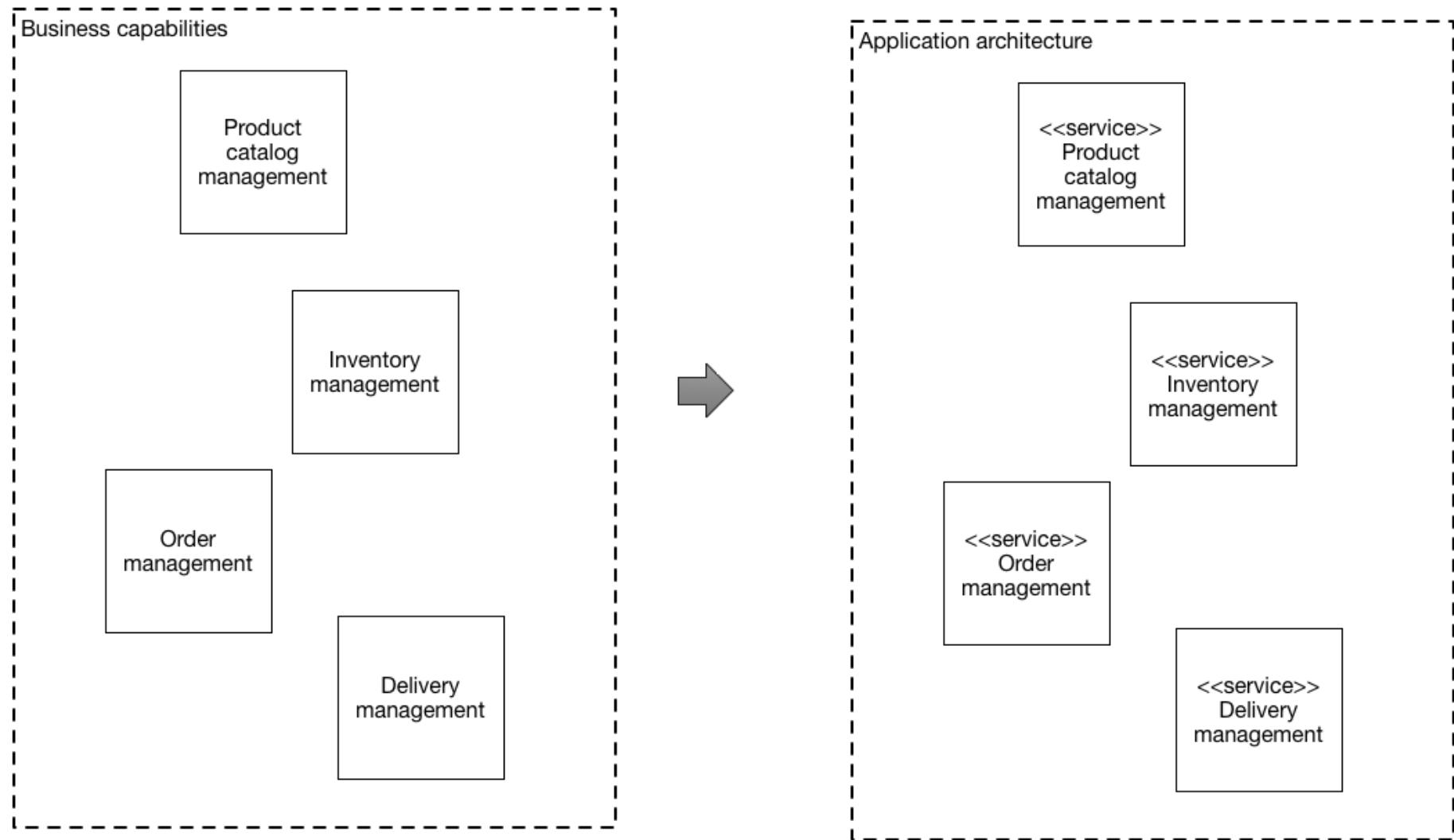
1. Decomposition of Monolith into Microservices
2. Database per Service
3. Transaction Management Strategy
4. Deployment Patterns
5. Discovery Access Restrictions with API- Gateways
6. Observation and Monitoring

Decomposition with Domain Driven Design

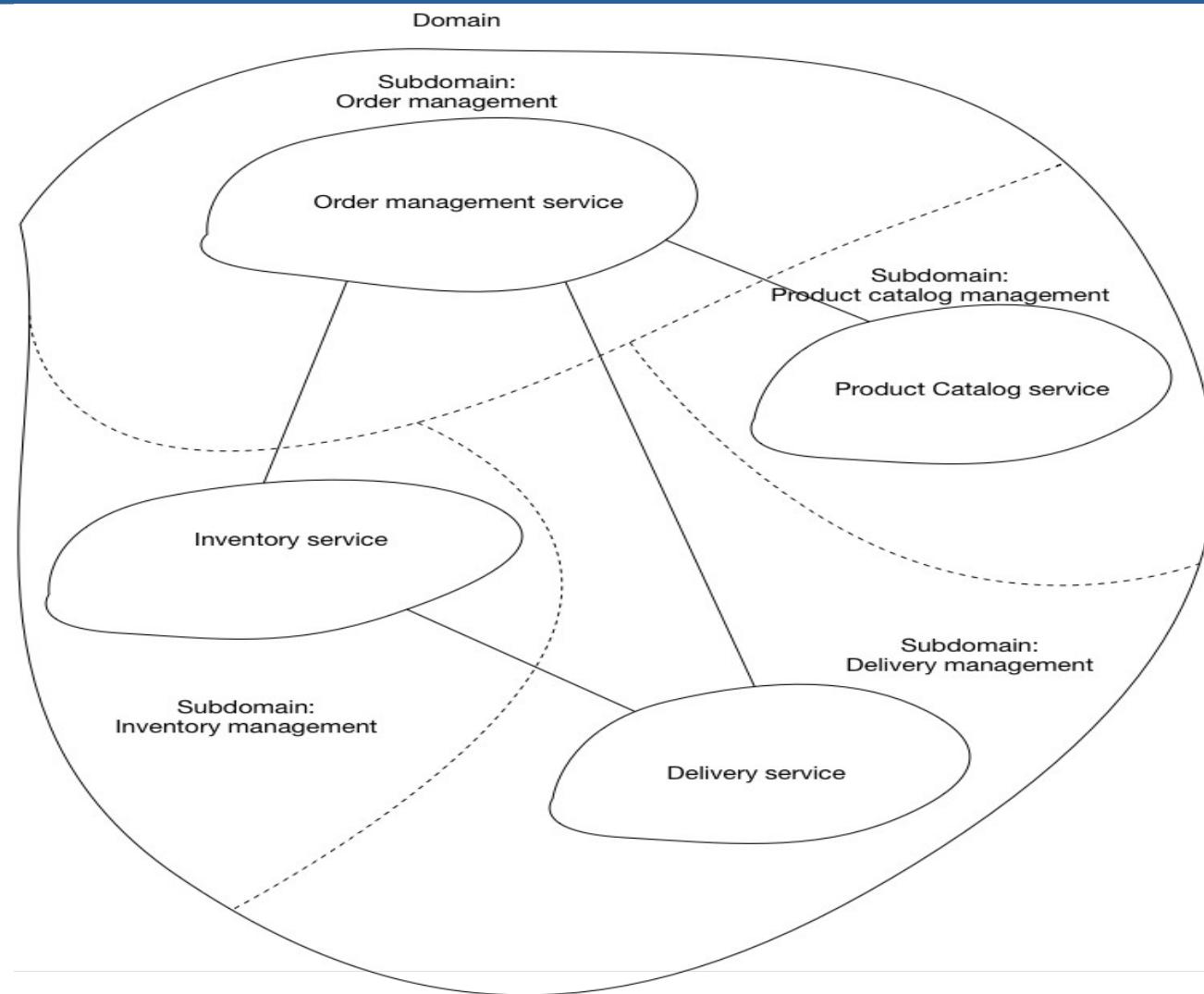
The ways to decompose a monolith

1. Decomposition by Business Capability
2. Decomposition by Sub-Domains
3. Service Per Team

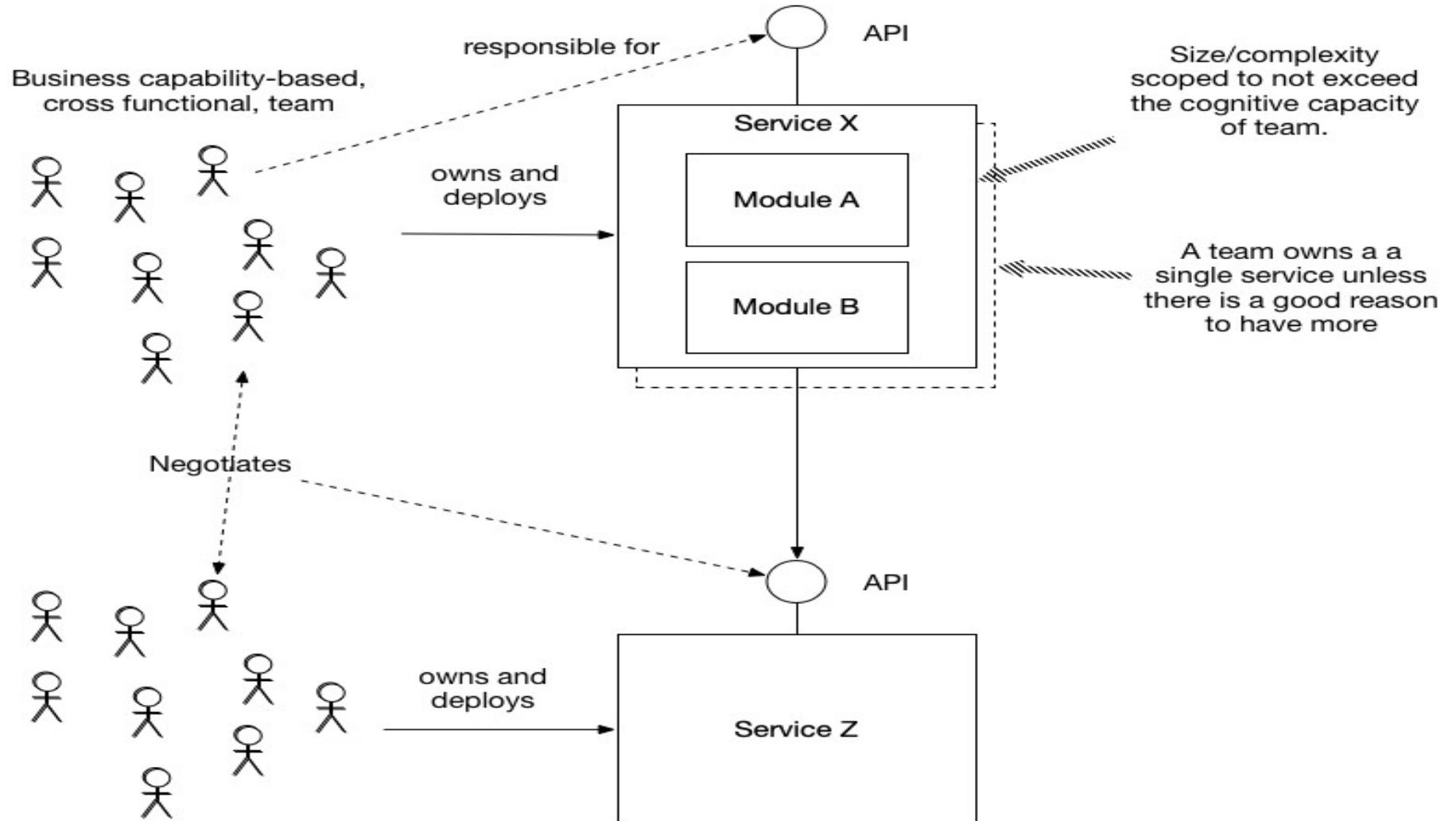
Decomposition by Business Capability



Decomposition by Sub domain



Decomposition by Service Per Teams

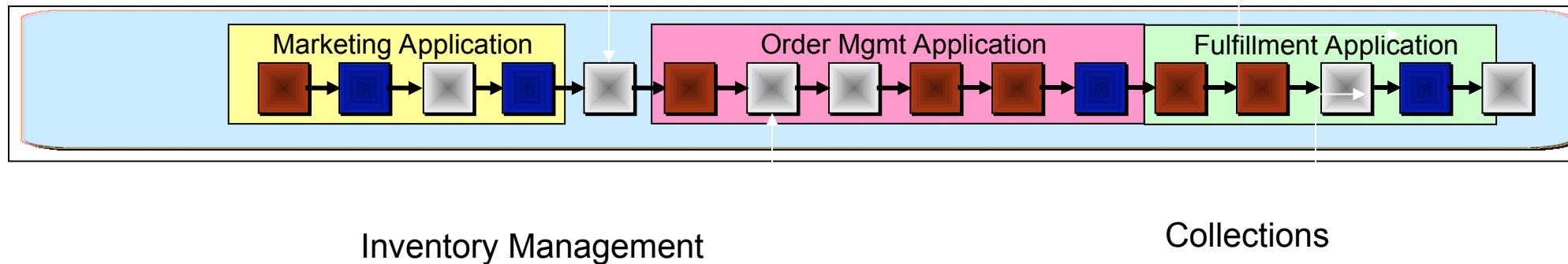


Traditional Business Process

Example: Order to Cash Process

 = business function

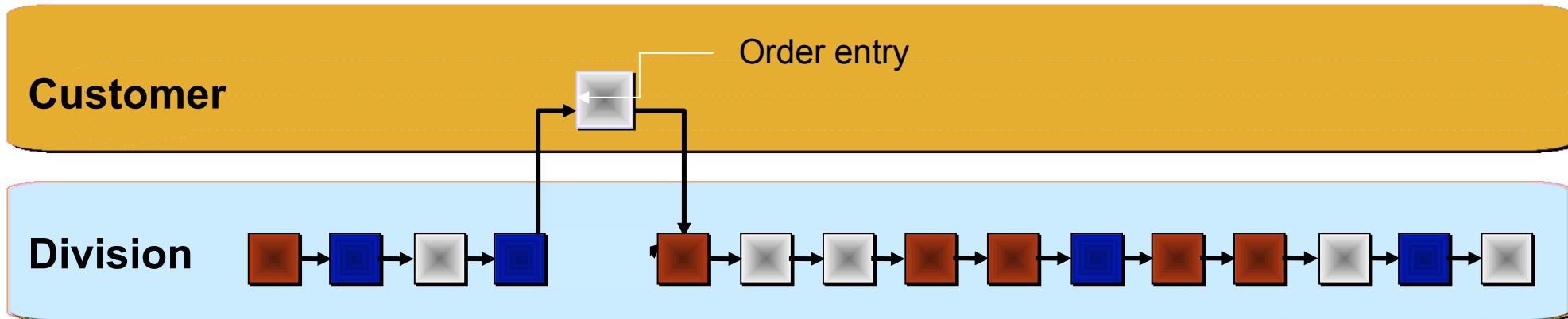
Manual Steps



- Business process is embedded in three separate applications
- Changes to the process are difficult to implement
- New processes which are designed this way require long development cycles
- Business functions are tightly coupled within applications
- Process cannot be easily measured and managed

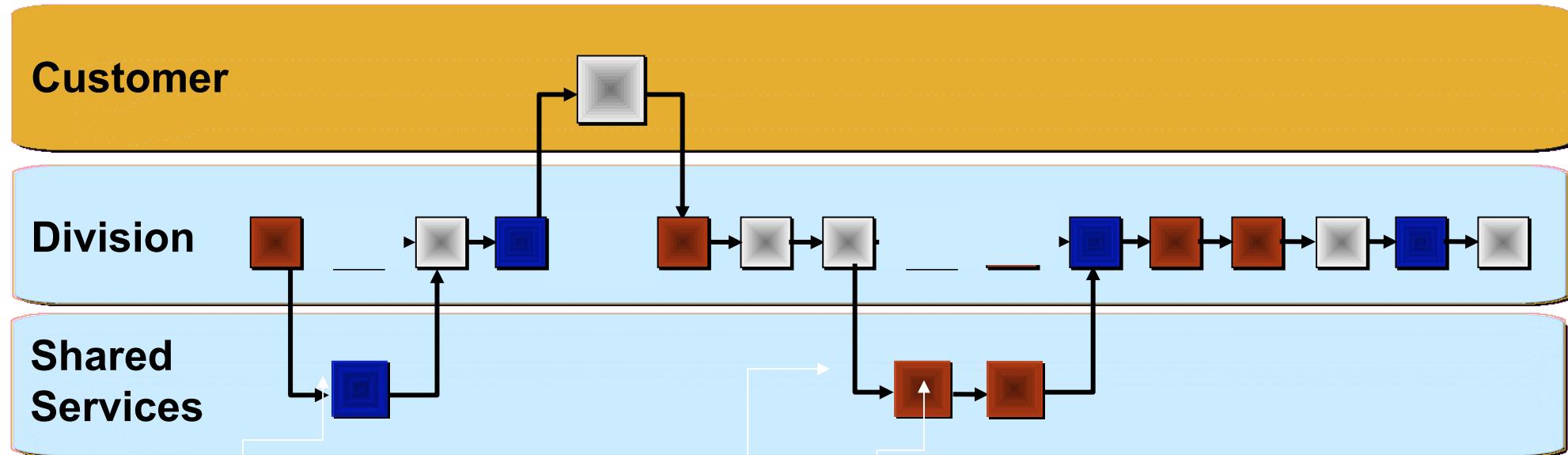
Result: Business cannot respond quickly to demand

On Demand Flexibility: Customer Self Service



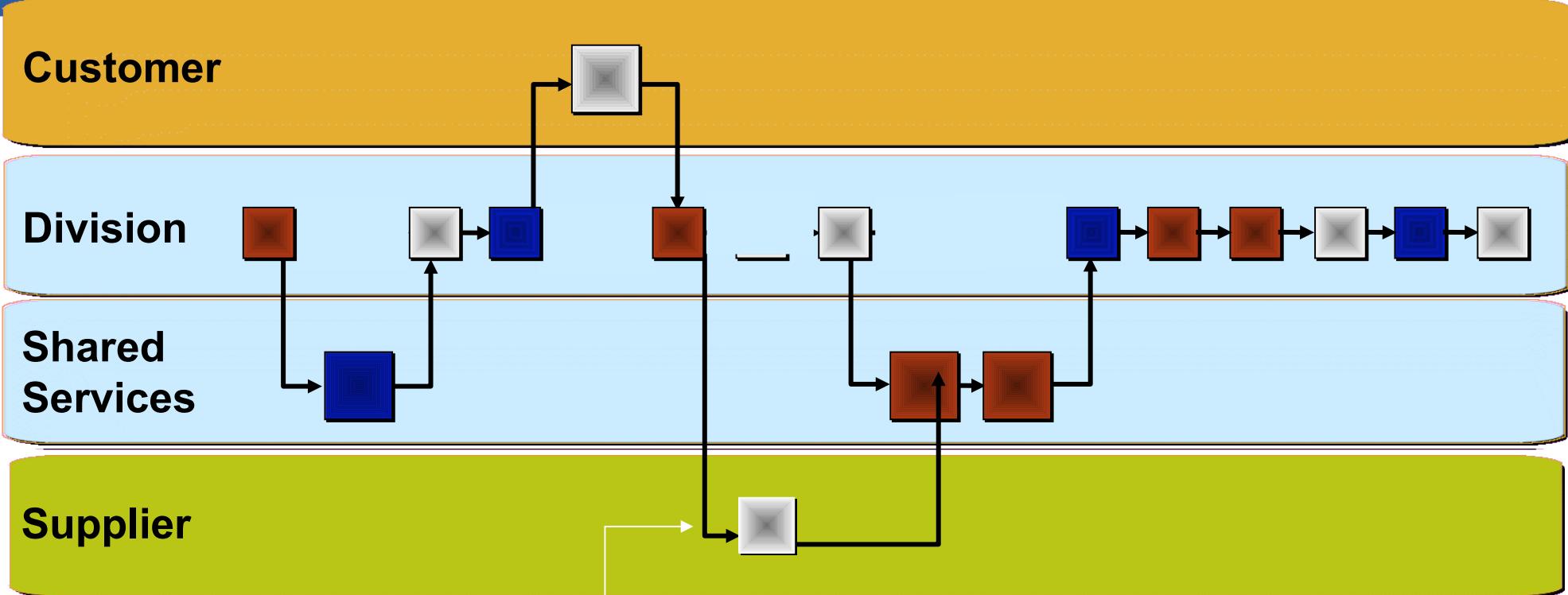
- Customers now order online using a web browser and the Internet
- Business partners can order using a web service call from their own process
- Customers are better served

On Demand Flexibility: Shared Services



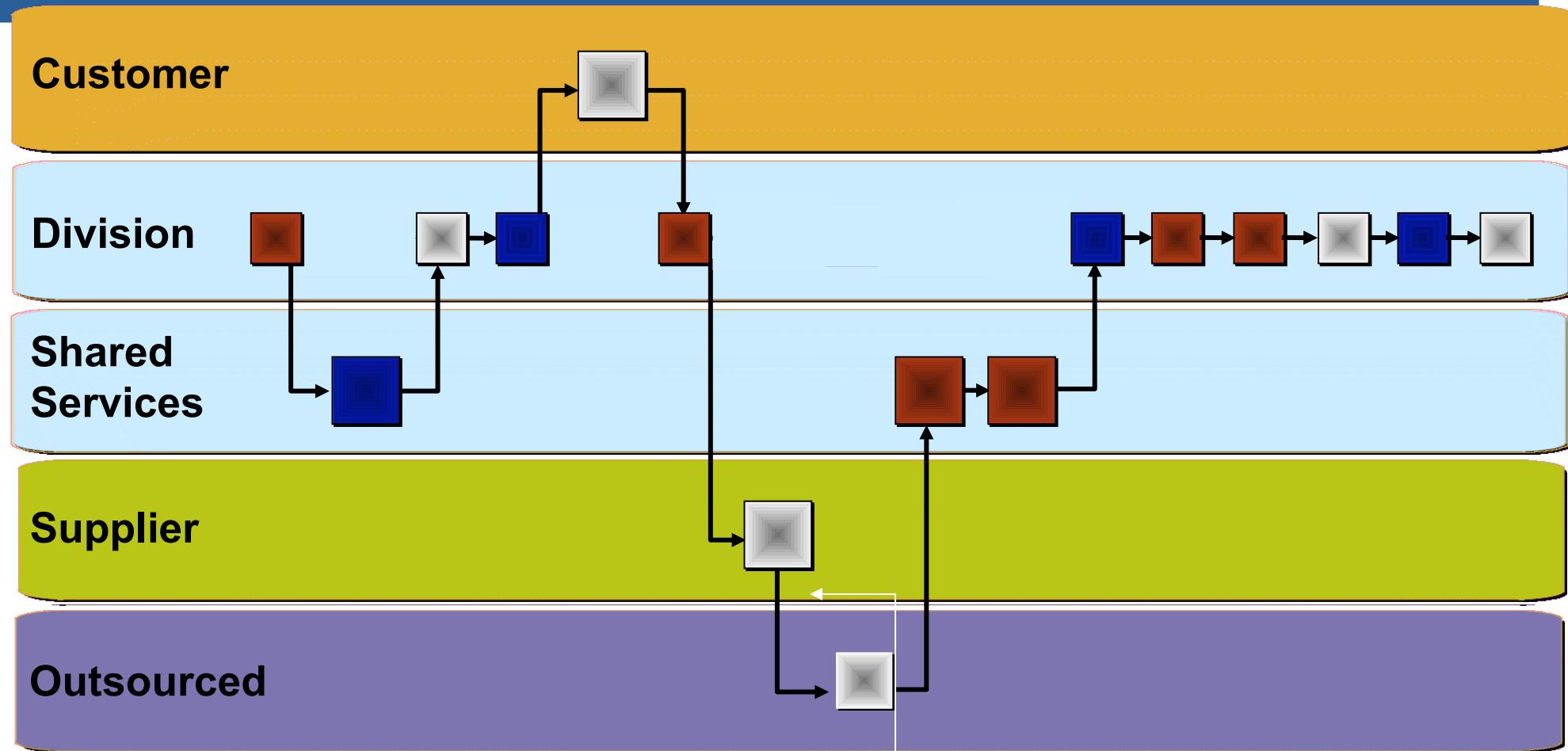
- Common business functions are shared across the enterprise
- Marketing, Billing, and Receivables are handled uniformly
- Enterprise can scale across divisions and lower costs

On Demand Flexibility: Vendor Managed Inventory



- Minimize or eliminate inventory management function
- Costs are reduced because less inventory is needed
- Inventory servicing is better because of supplier integration with the process

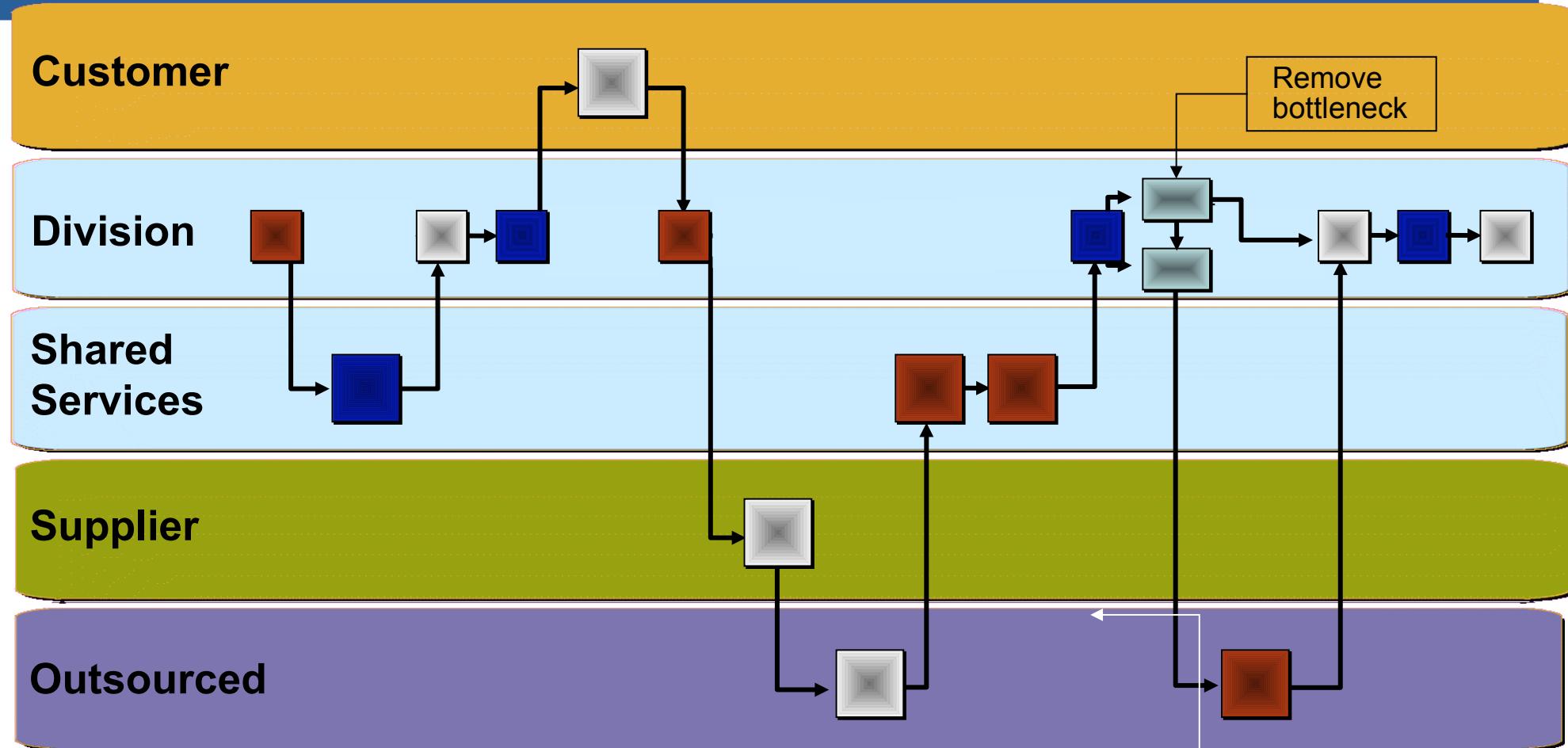
On Demand Flexibility: Outsource



- Shipping is not a core competency
- Shipping companies (e.g. FedEx, DHL, UPS) have more capabilities
- Reduce shipping infrastructure and overhead costs

Shipping

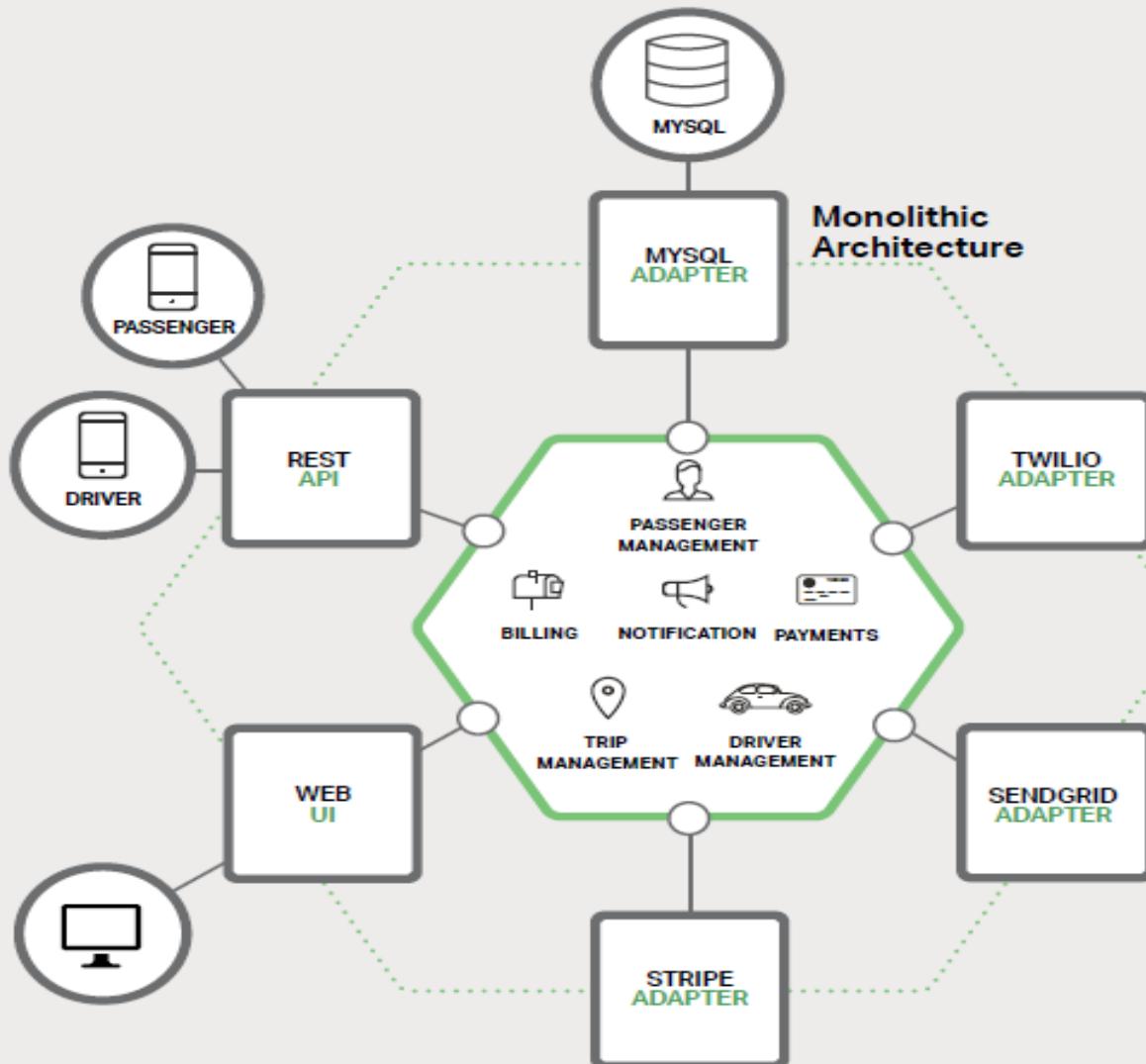
On Demand Flexibility: Improve the Process



- Identify and remove bottlenecks in the process
- Customize the business rules and policies to better serve customers
- A more efficient business process costs less

Alternative flow path

Another Example – A Monolith Cab App

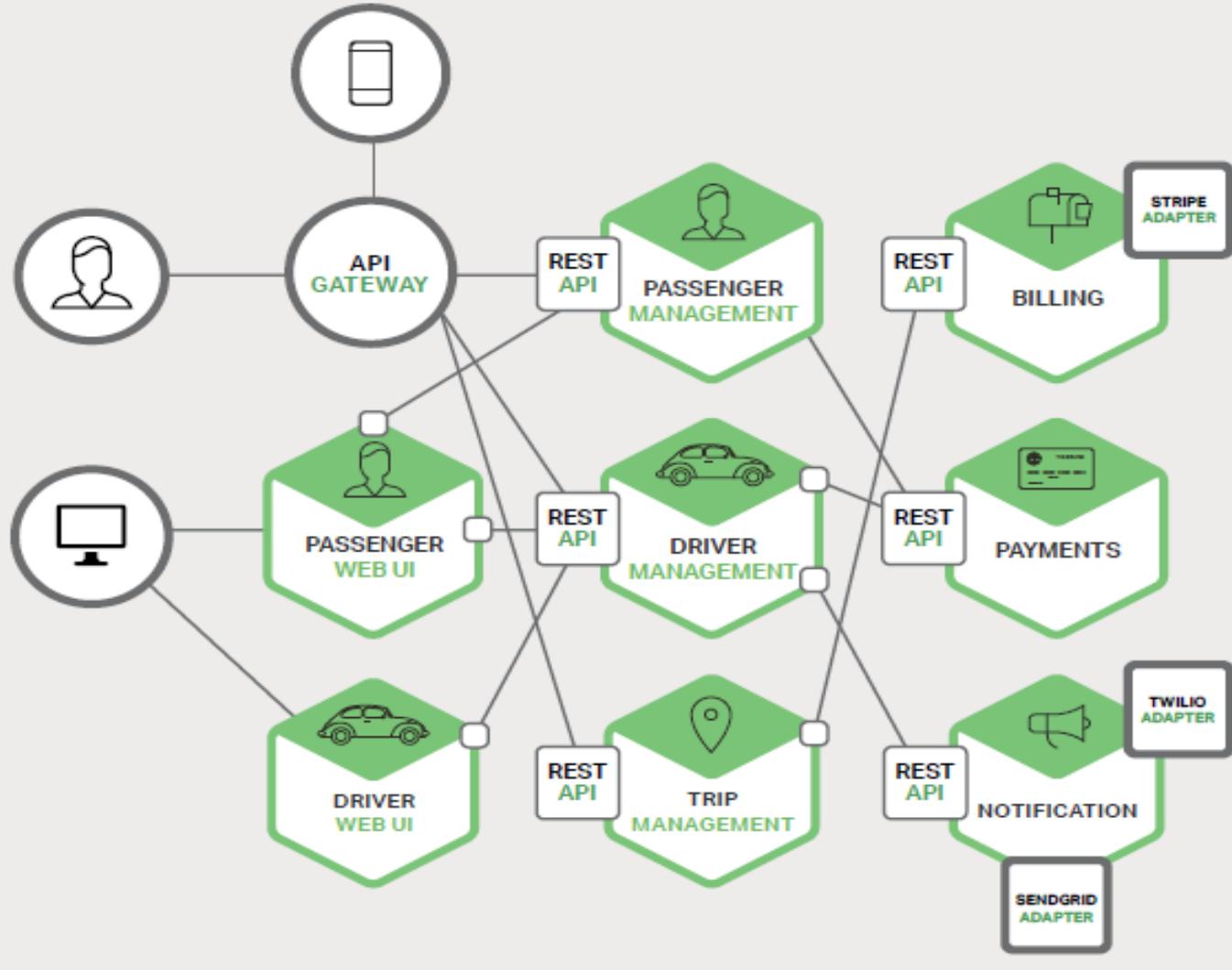


An Enterprise Application of a Cab based Application like that of Uber/OLA

A modular application deployed as Monolith as

1. Written in a single language.
2. deployed on one kind of server
3. depending on adapters fro outside world.

Monolith Cab App Decomposed into MSA



The monolith broken into a Micro Services Architecture **as**

1. Every module is independent and running int its own container
2. The services communicate through Rest API calls
3. Loose coupled applications
4. Independent of client interface

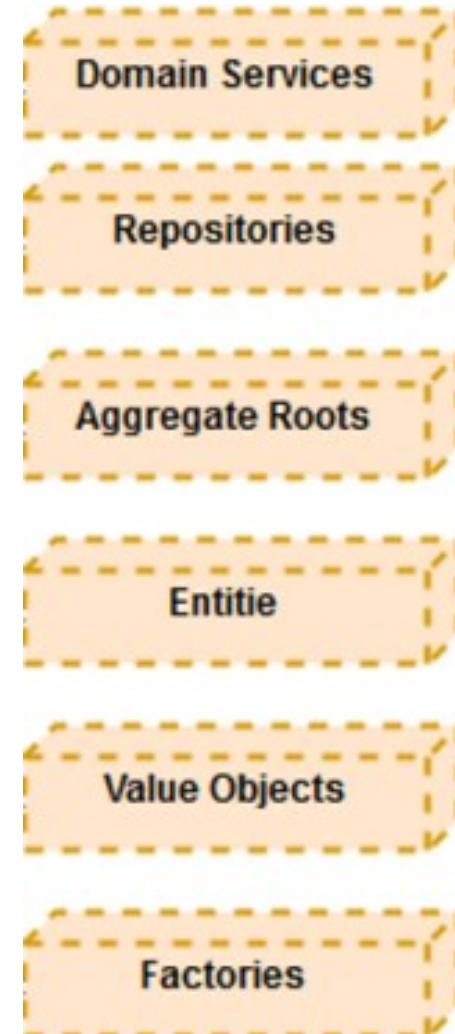
Domain Driven Design for Monolith to MicroServices

The Domain Driven Design involves following terms

1. Domain
2. Subdomains
3. Domain Model
4. Defined Bounded Contexts (Domain Model + DB Schema + UI + Web Services (API))
5. Ubiquitous Language
6. Context Map
7. entity, value object, aggregate, services, factory, and repository are the building blocks also known as the tactical approach to Domain-Driven Design toward its full realization:

Domain Driven Design

Building blocks of Domain Driven Design



Transforming Domain Driven Design to Micro Services Architecture

- 1) Define a set of bounded contexts for a microservices application
 - 1) Then we looked more closely at one of these bounded contexts, the Change management bounded context, and identified a set of entities, aggregates, and domain services for that bounded context
 - 1) by doing so we can define an architecture that structures the application as a set of loosely coupled, collaborating services. Each service implements a set of narrowly, related functions
 - 1) For example, an application might consist of services such as the order management service, the customer management service, etc.

Microservice, a component in this architecture:

1) Each is a miniature application

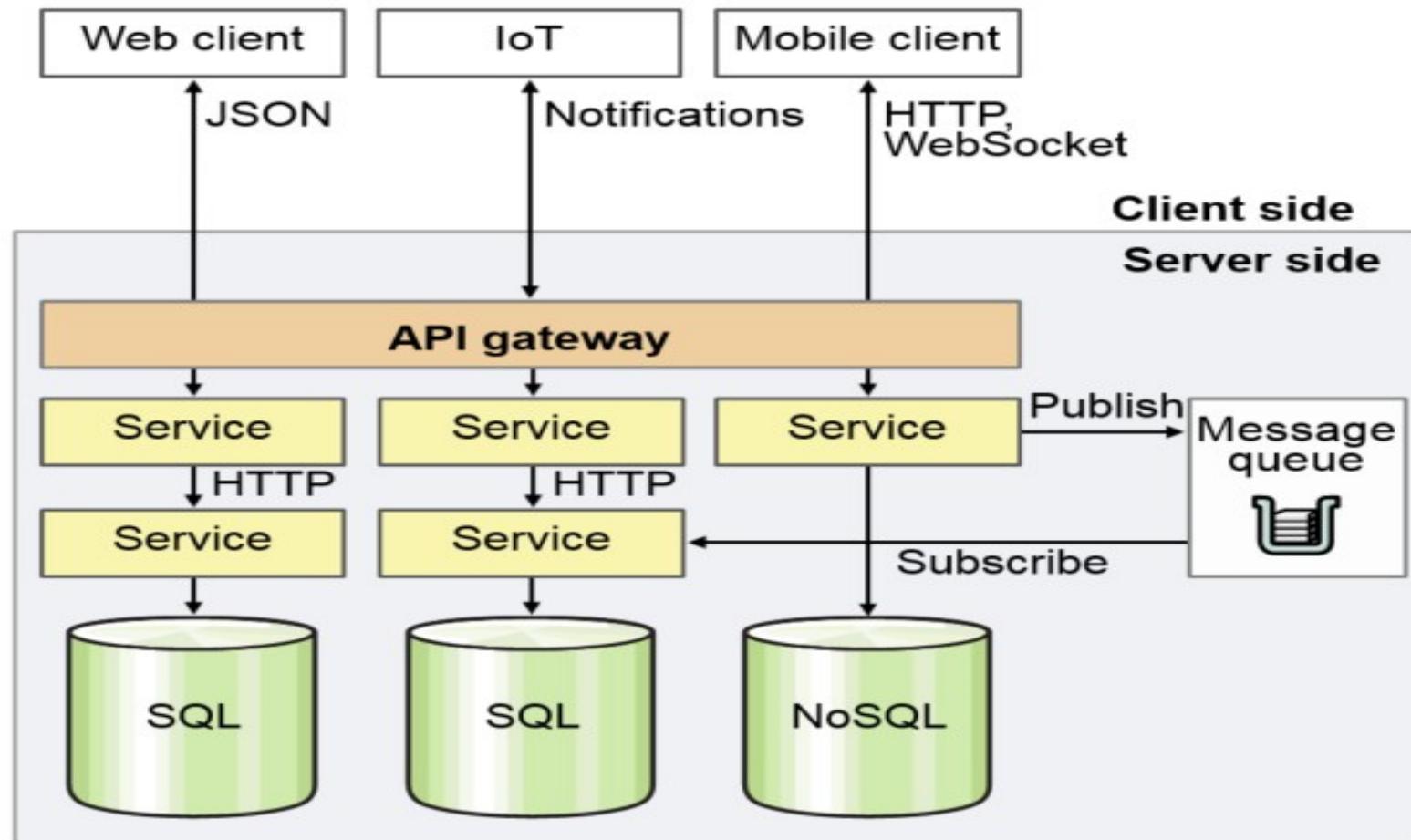
1) Each is focused on one task, a business capability (The Single Responsibility principle: Each microservice implements only one business responsibility from the bounded domain context. From a software point of view, the system needs to be decomposed into multiple components where each component becomes a microservice. Microservices have to be lightweight, in order to facilitate smaller memory footprints and faster startup times.)

1) Each can be deployed and updated independently

1) They are loosely coupled

1) Each has a well-defined interface: REST APIs

Microservice Architecture:



Microservice Architecture: Benefits

- 1) Microservices architecture is a very promising way of designing and building highly scalable applications in an agile way. The services themselves are straightforward, focusing on doing only one thing well, so they're easier to test and ensure higher quality.
- 1) Each service can be built with the best-suited technologies and tools, allowing polyglot persistence and such. You don't have to be stuck with a new choice of technology for the rest of the project.
- 1) Multiple developers and teams can deliver independently under this architecture. This is great for continuous delivery, allowing frequent releases while keeping the rest of the system stable.

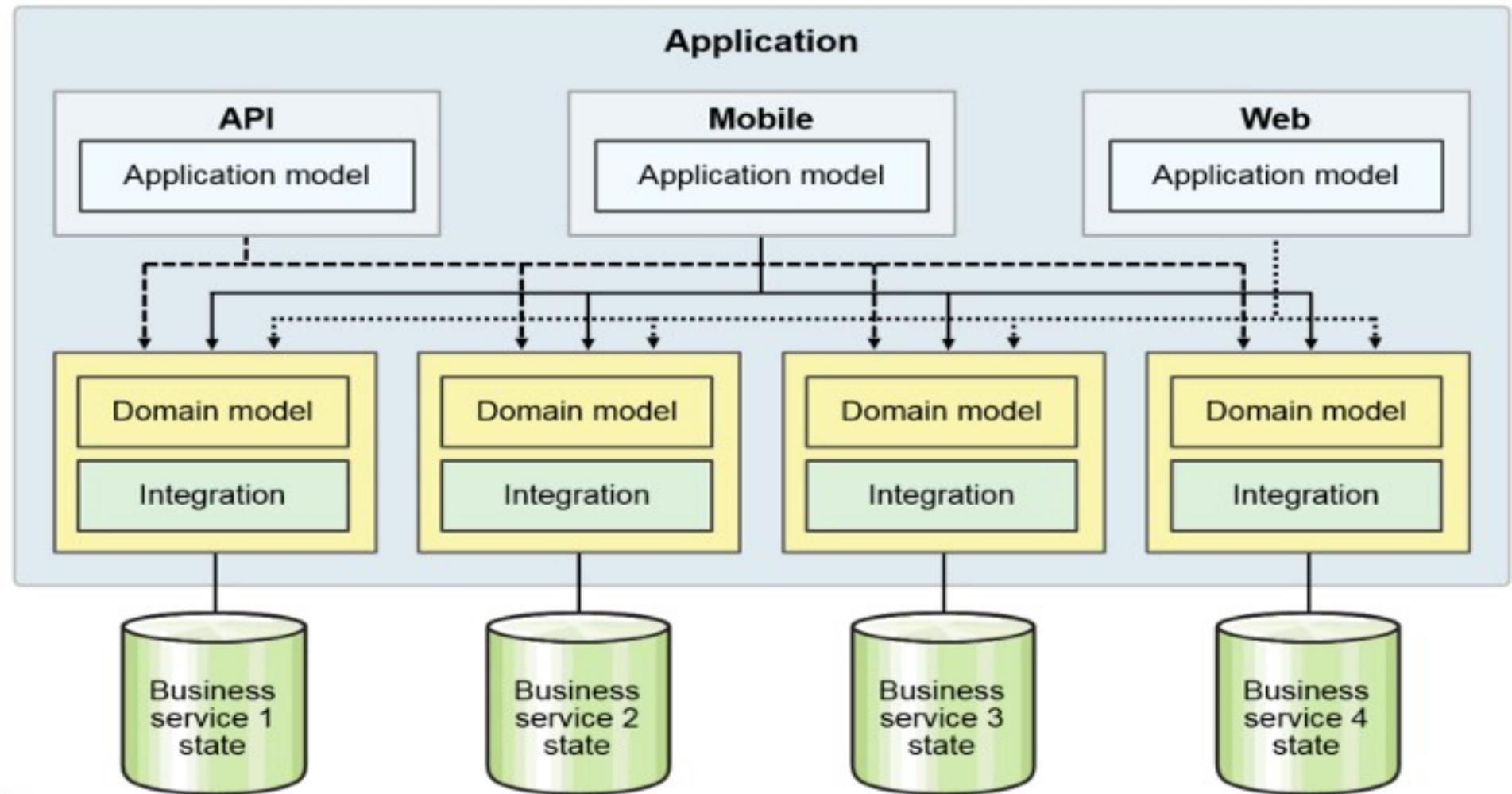
Microservice Architecture: Benefits

- 4) In case a service goes down, it will only affect the parts that directly depend on it (if there are such parts). The other parts will continue to function well.
- 5) Externalized configuration: This externalizes the configurations in the config server so that it can be maintained in a hierarchical structure, per environment.
- 6) Consistent: Services should be written in a consistent style, as per the coding standards and naming convention guidelines...

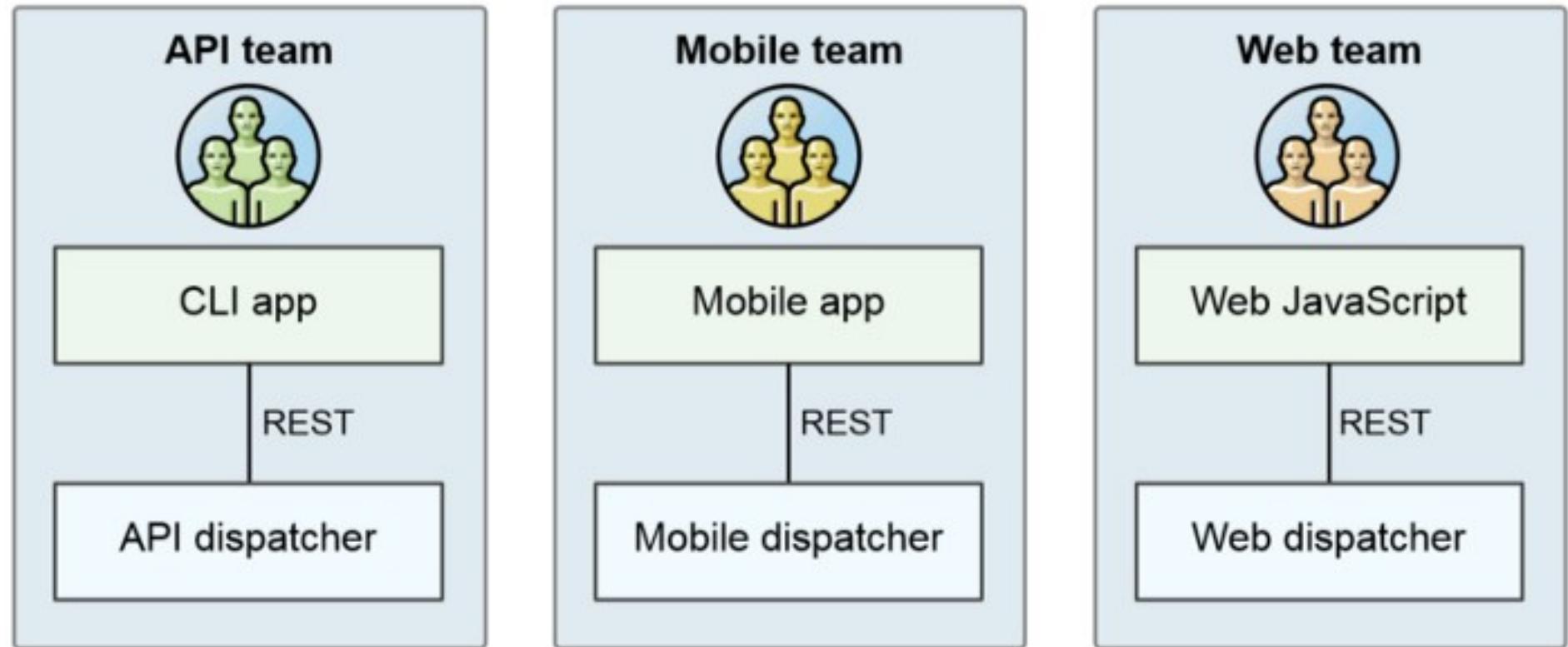
Microservice Architecture: Benefits

- 7) Resilient: Services should handle exceptions arising from technical reasons (connectivity and runtime), and business reasons (invalid inputs) and not crash. Patterns, such as circuit breakers and bulk headers, help isolate and contain failures.
- 8) Good citizens: Microservices should report their usage statistics, the number of times they are accessed, their average response time, and so on through the JMX API or the HTTP API.
- 9) Versioned: Microservices may need to support multiple versions for different clients until all clients migrate to higher versions. There should be a clear version strategy in terms of supporting new features and bug fixing...

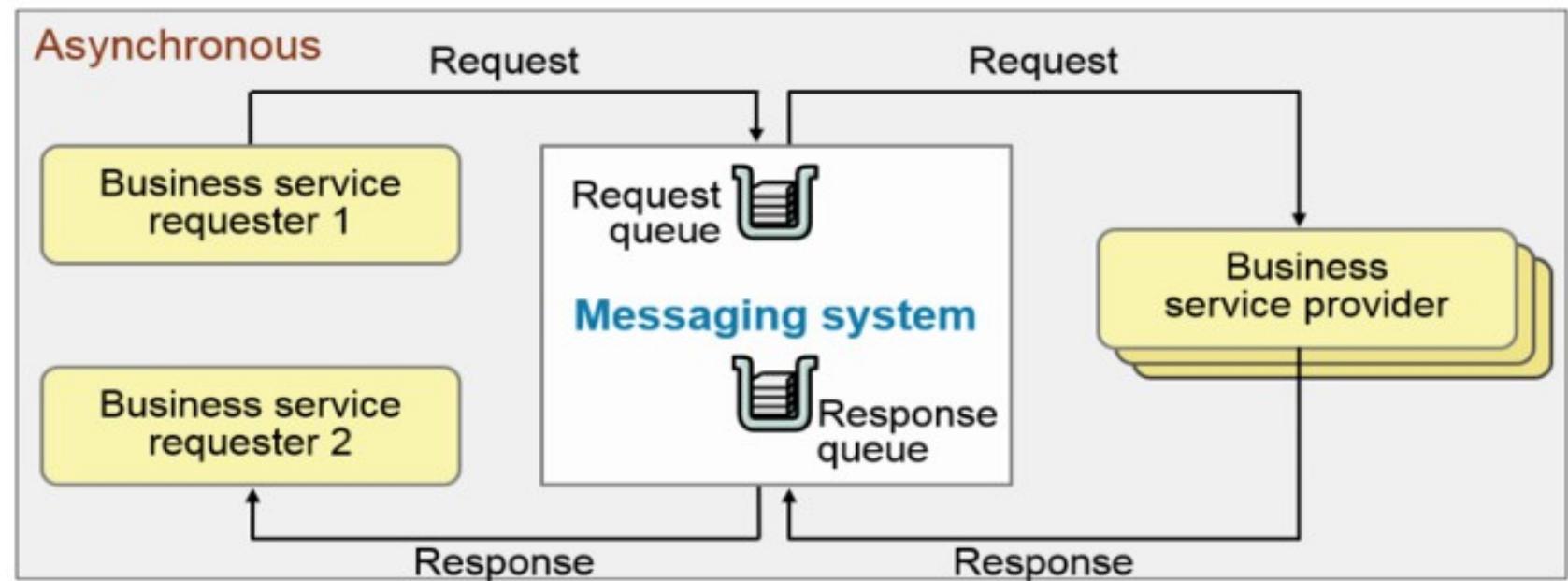
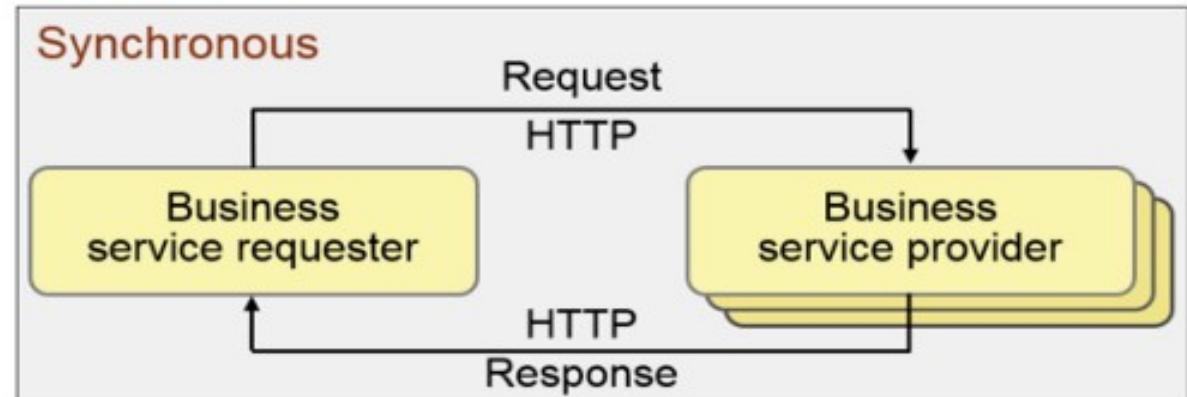
Microservice Architecture: Layers



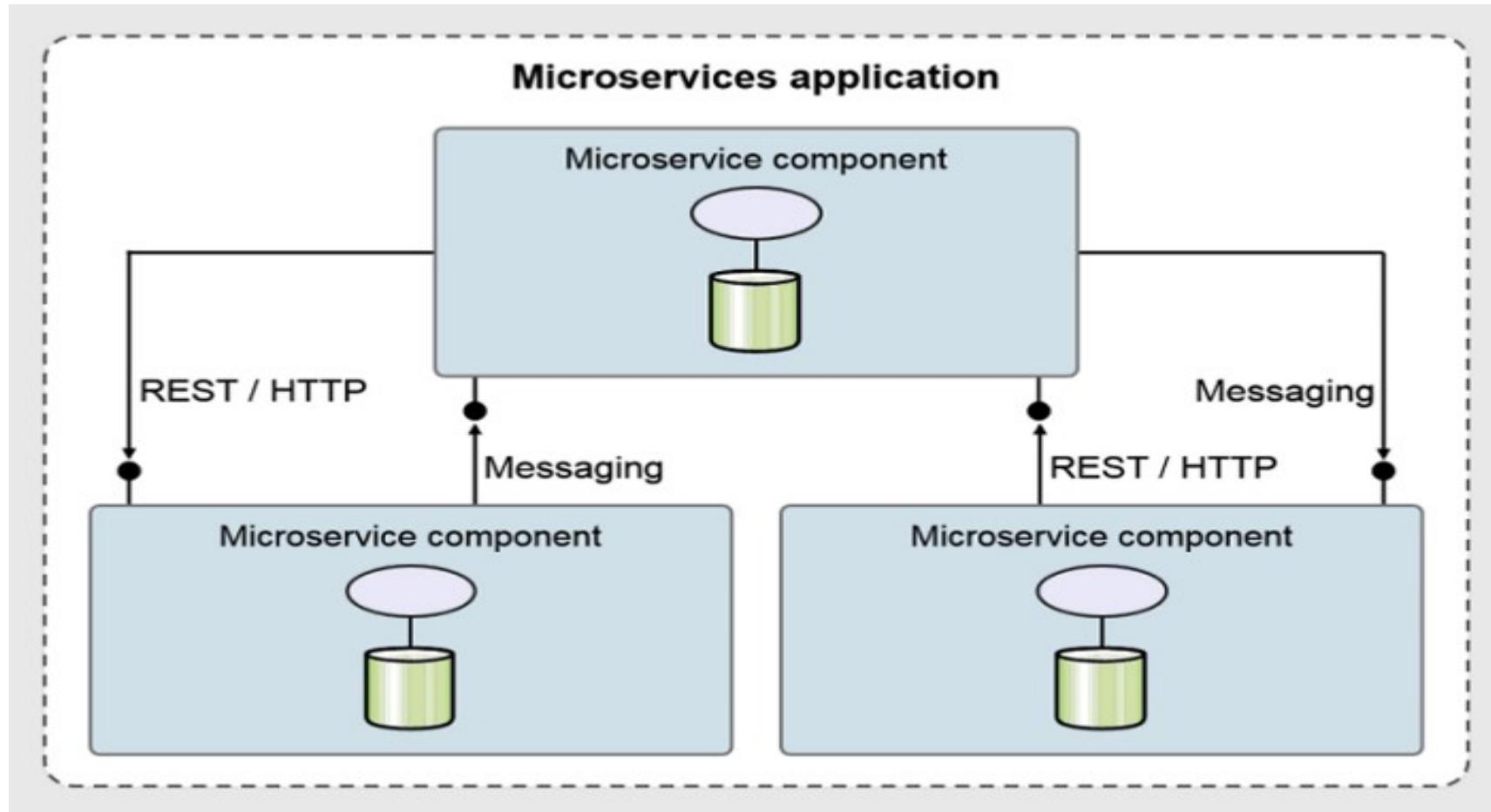
Microservice Architecture: Dispatchers and Client



Microservice Architecture: Inter communication



Microservice Architecture: Hybrid Inter communication



Decomposition Outcomes

- The architecture must be stable
- Services must be cohesive. A service should implement a small set of strongly related functions.
- Services must conform to the Common Closure Principle - things that change together should be packaged together - to ensure that each change affect only one service
- Services must be loosely coupled - each service as an API that encapsulates its implementation. The implementation can be changed without affecting clients
- A service should be testable
- Each service be small enough to be developed by a “two pizza” team, i.e. a team of 6-10 people
- Each team that owns one or more services must be autonomous. A team must be able to develop and deploy their services with minimal collaboration with other teams.

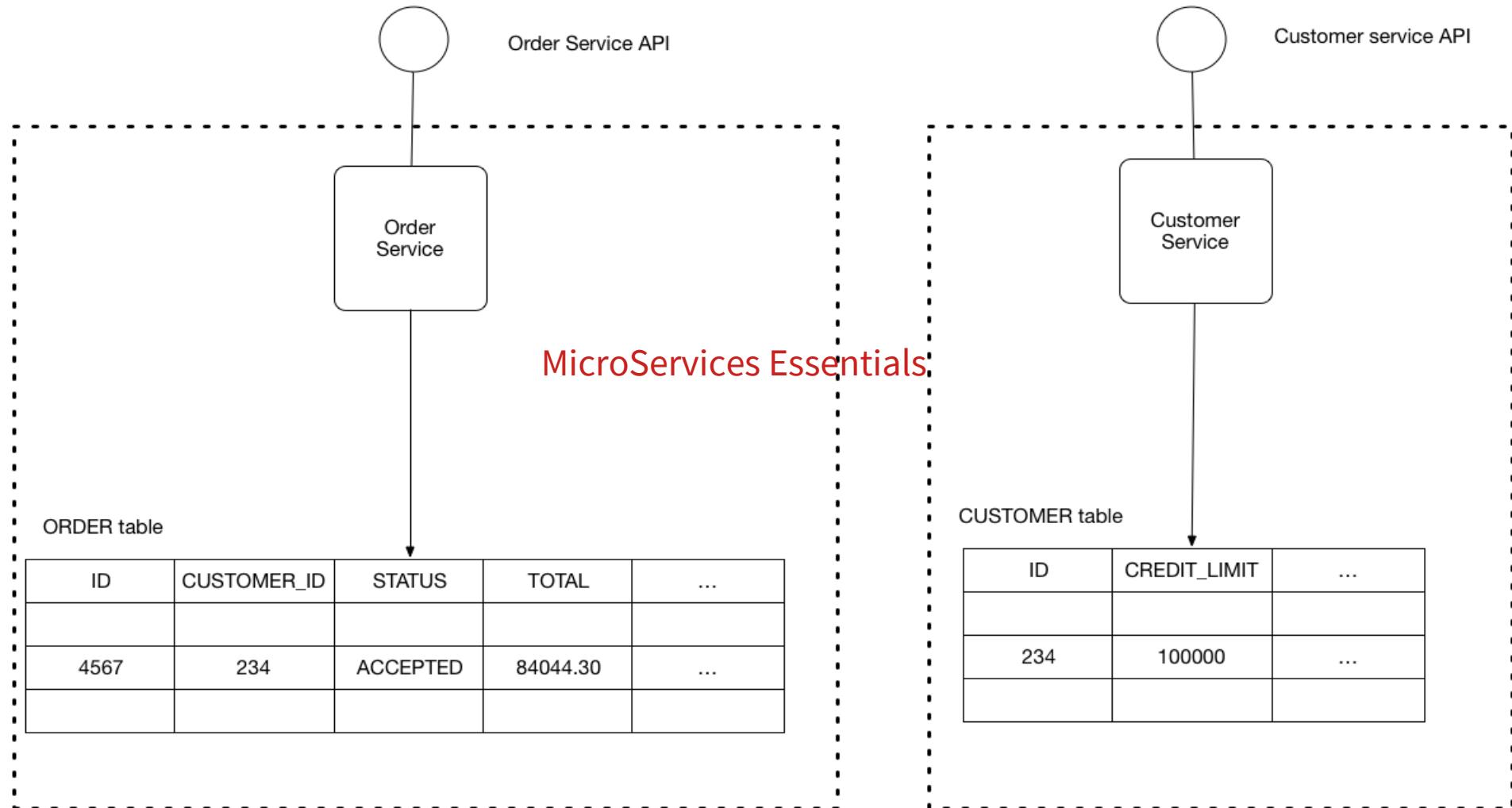
MicroServices Essentials -

Database per Service

- The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services. If you are using a relational database then the options are:
- **Private-tables-per-service** – each service owns a set of tables that must only be accessed by that service
- **Schema-per-service** – each service has a database schema that's private to that service
- **Database-server-per-service** – each service has it's own database server.

MicroServices Essentials

Database per Service

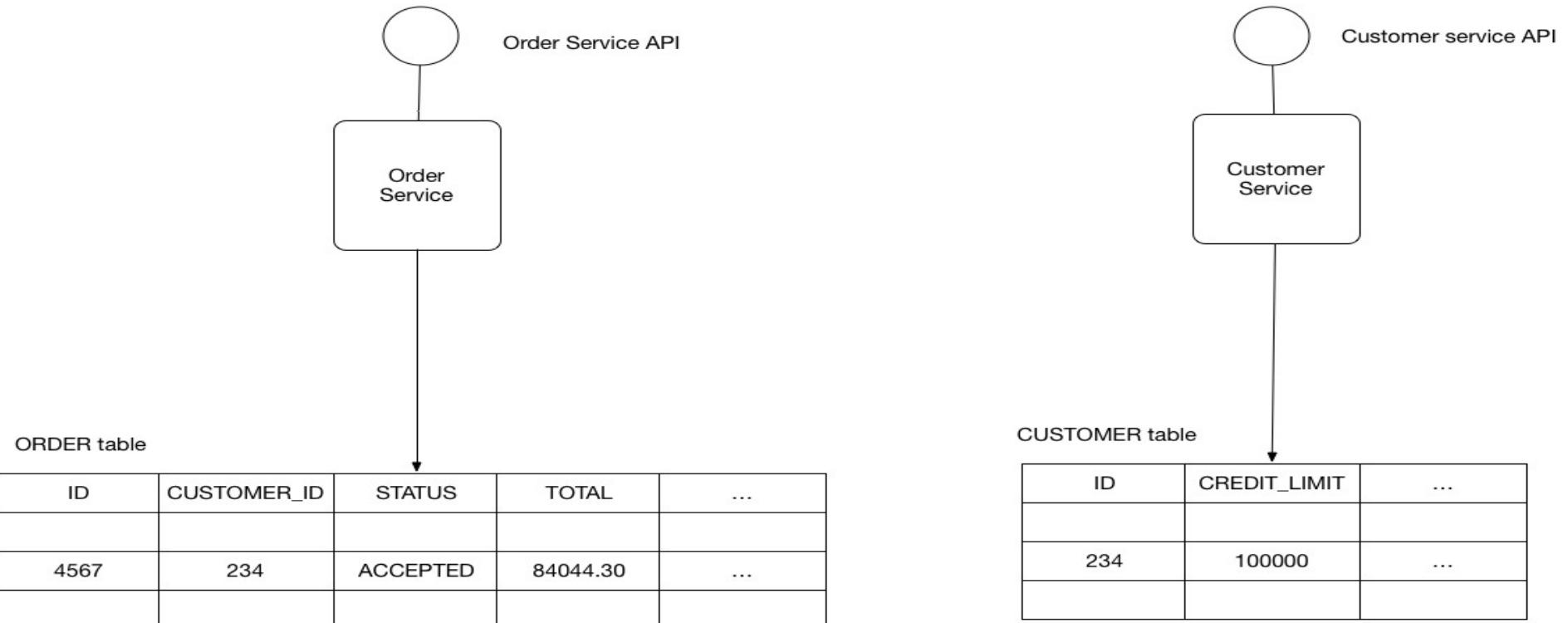


MicroServices Essentials

Shared Database

The reason for going for shared database for multiple services

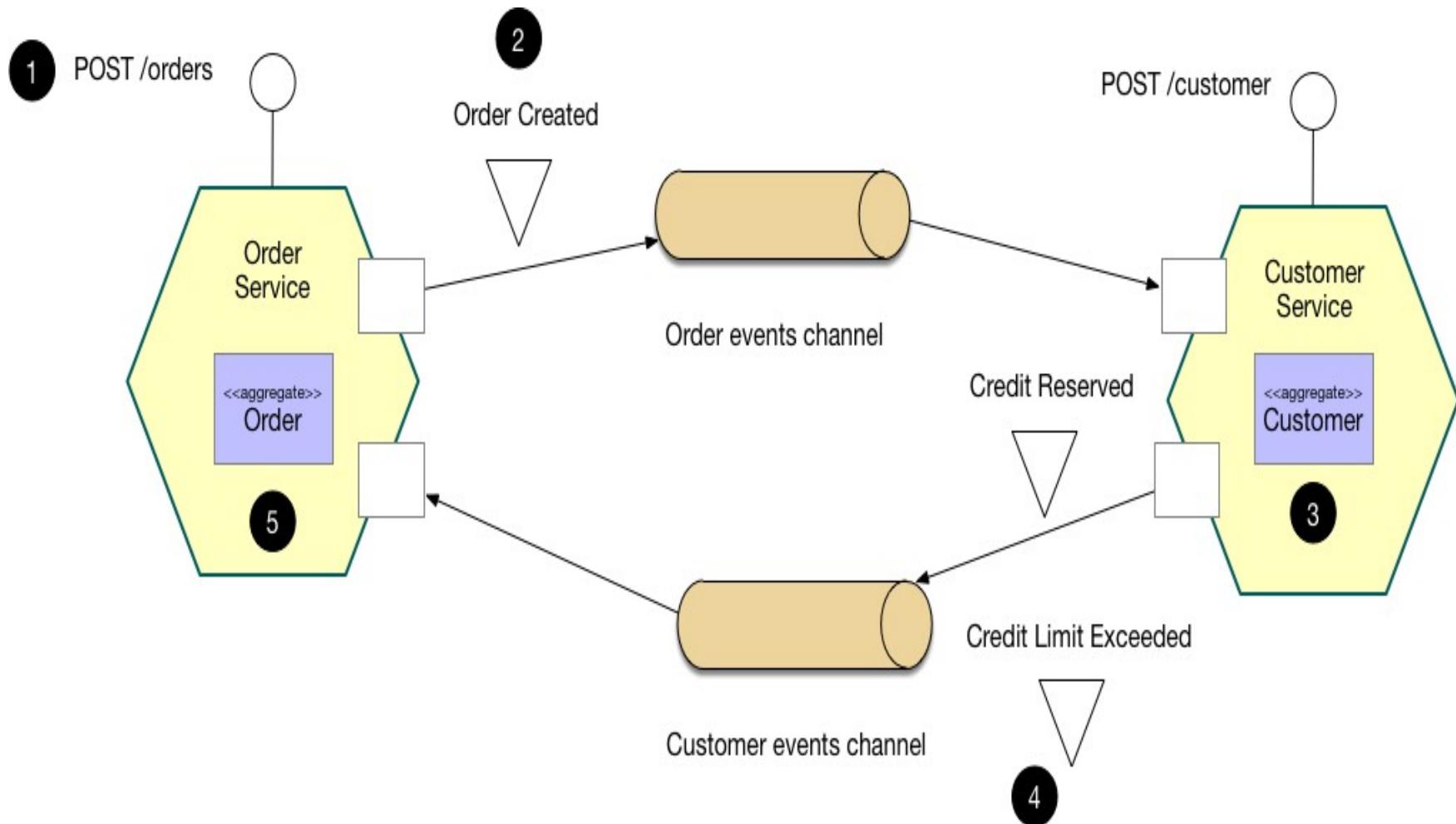
- Some transaction spawn over many services
- Some operations require data from multiple services
- Database replication is required



MicroServices Essentials

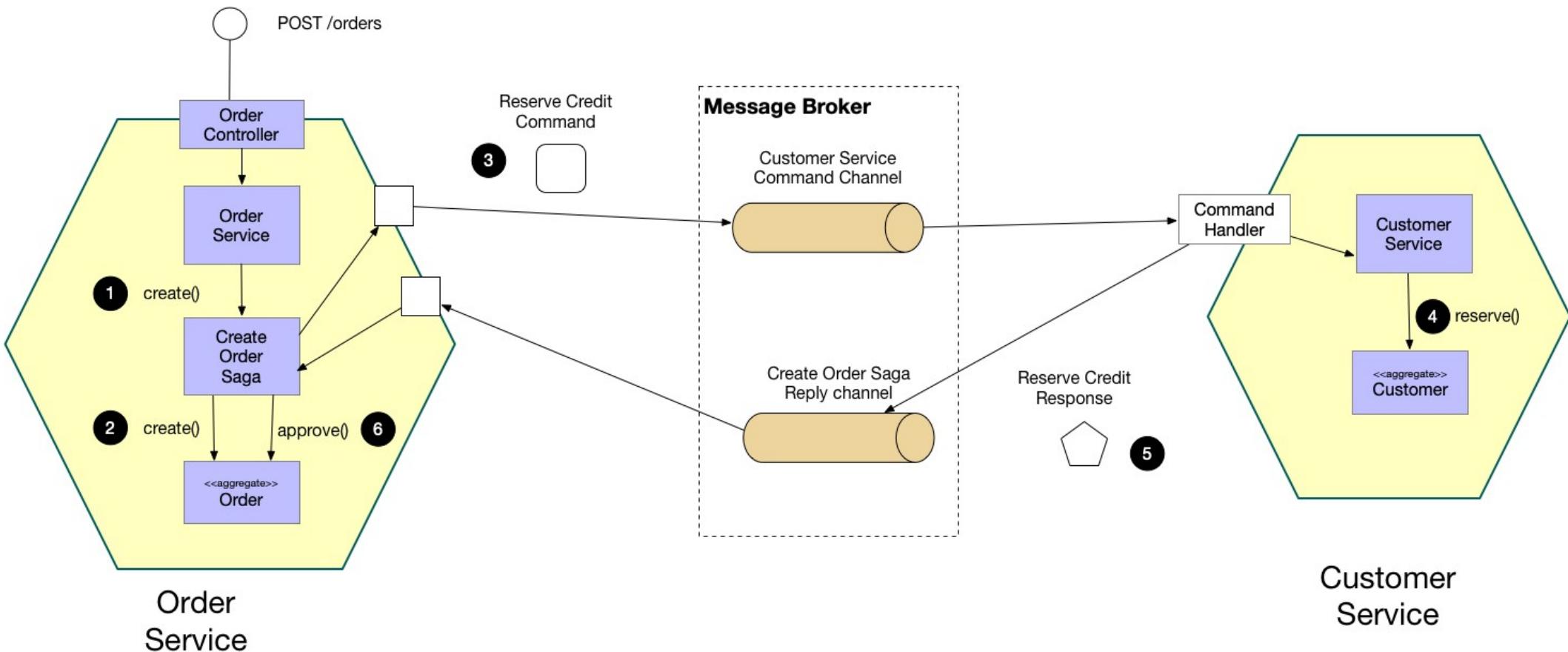
Using SAGA – Choreography Based

A saga is a sequence of local transactions where each transaction updates data within a single service. The first transaction is initiated by an external request corresponding to the system operation, and then each subsequent step is triggered by the completion of the previous one



MicroServices Essentials

Using SAGA – Orchestration Based



Customer
Service

22:12:43

MicroServices Essentials

Command Query Responsibility Segregation (CQRS) - Query from Multiple Services

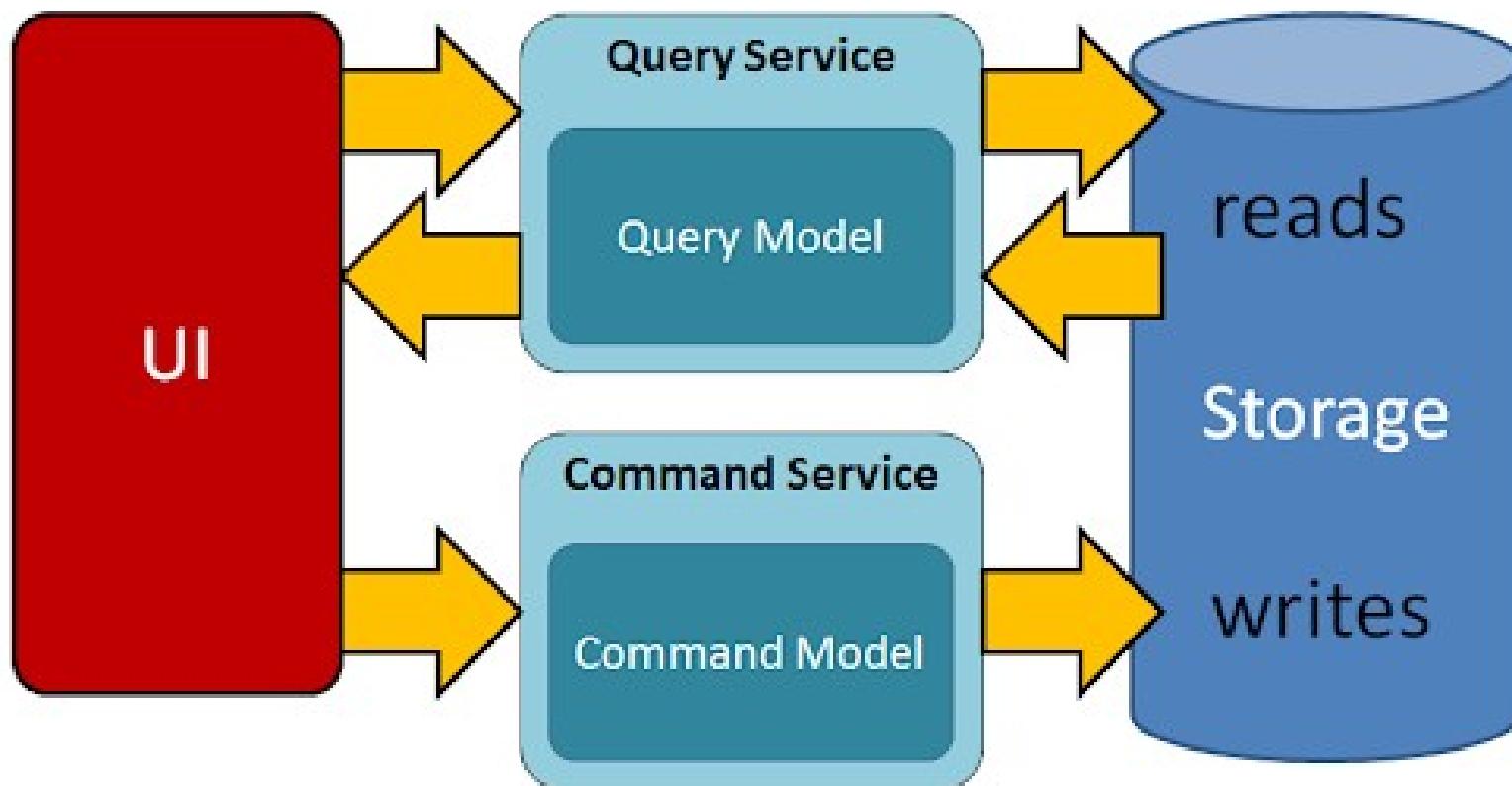
The Command and Query Responsibility Segregation (CQRS) pattern propose separating the write data model from the read data model.

This separation of responsibilities would provide the flexibility to decide whether the read and write services should coexist in the same data store or be managed in completely different databases.

MicroServices Essentials

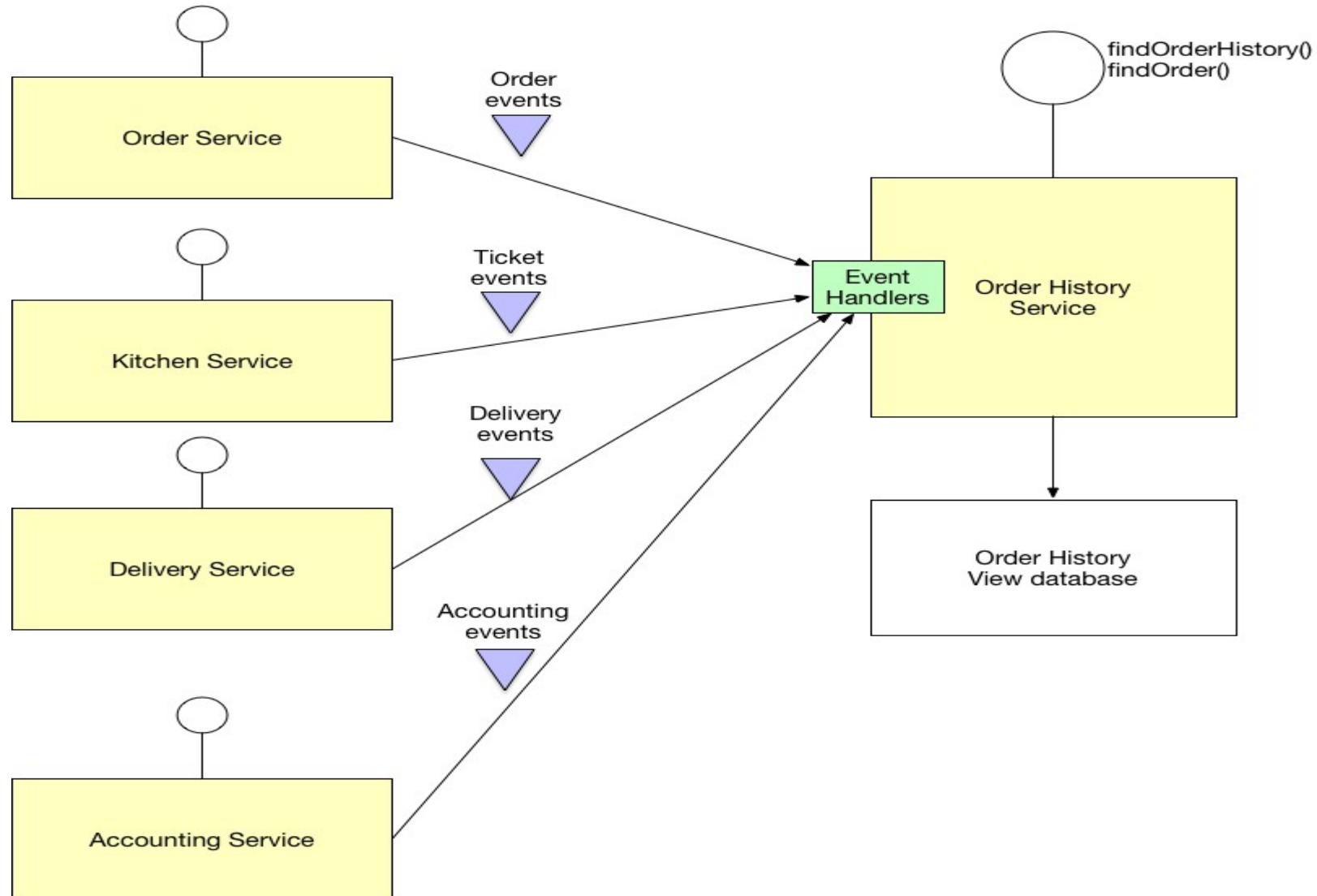
Command Query Responsibility Segregation (CQRS) - Query from Multiple Services

CQRS Pattern



MicroServices Essentials

Command Query Responsibility Segregation (CQRS) - Query from Multiple Services



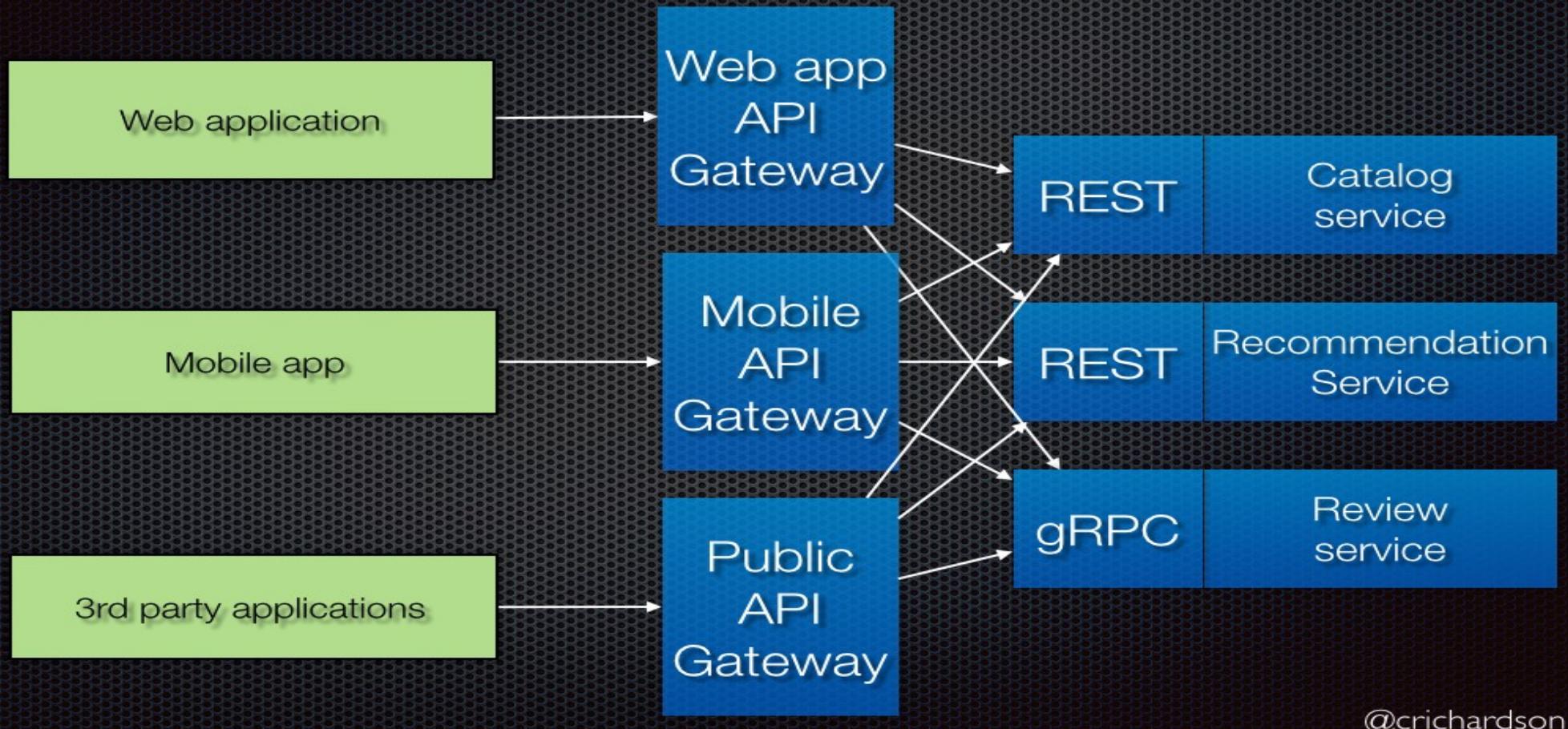
Command Query Responsibility Segregation (CQRS) - Benefits

- **Advantages**
- **Facilitates distributing development workload**
- **Easier maintenance**
- **Optimized database design**
- **Easier scalability**
- **Enhanced security**
- **Disadvantages**
- **Added complexity**
- **Eventual consistency**

MicroServices Essentials

Accessing Services using API Gateway

Variation: Backends for frontends



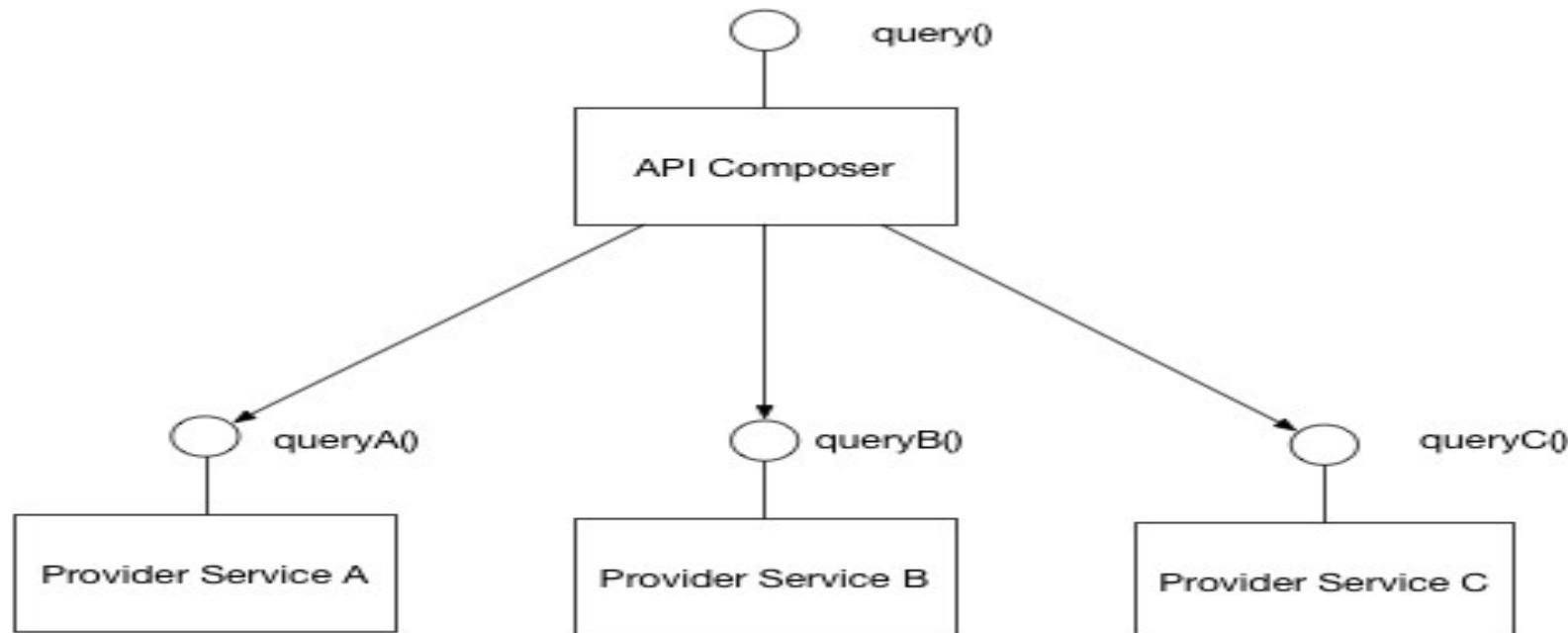
@crichton

22:12:43

MicroServices Essentials

API Composition

Implement a query by defining an API Composer, which invoking the services that own the data and performs an in-memory join of the results.

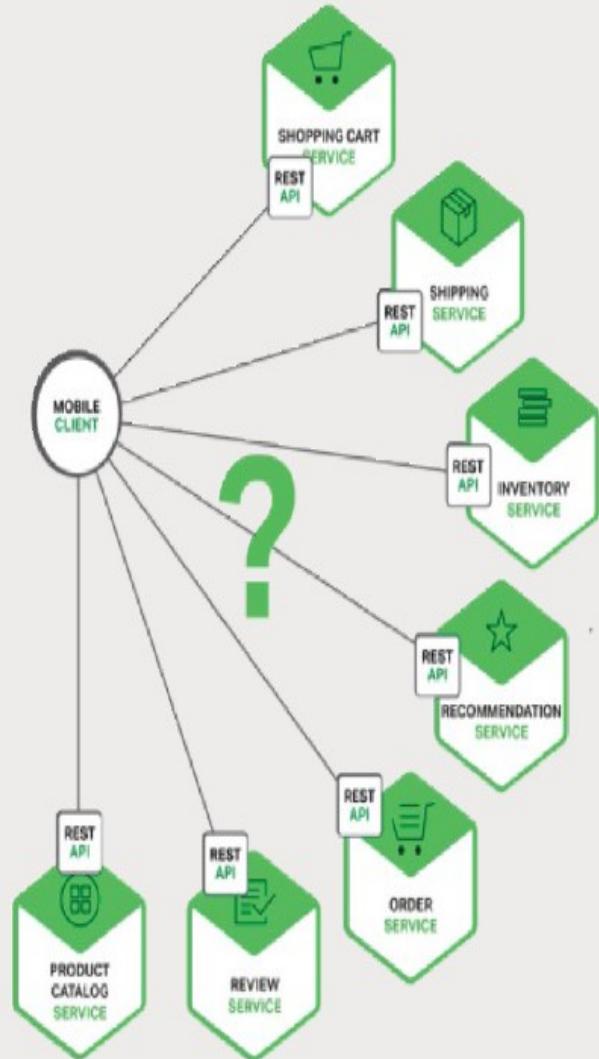


Accessing Services using API Gateway

Using an API gateway has the following benefits:

- Insulates the clients from how the application is partitioned into microservices
- Insulates the clients from the problem of determining the locations of service instances
- Provides the optimal API for each client
- Reduces the number of requests/roundtrips. For example, the API gateway enables clients to retrieve data from multiple services with a single round-trip. Fewer requests also means less overhead and improves the user experience. An API gateway is essential for mobile applications.
- Simplifies the client by moving logic for calling multiple services from the client to API gateway
- Translates from a “standard” public web-friendly API protocol to whatever protocols are used internally

- Problems without API Gateways

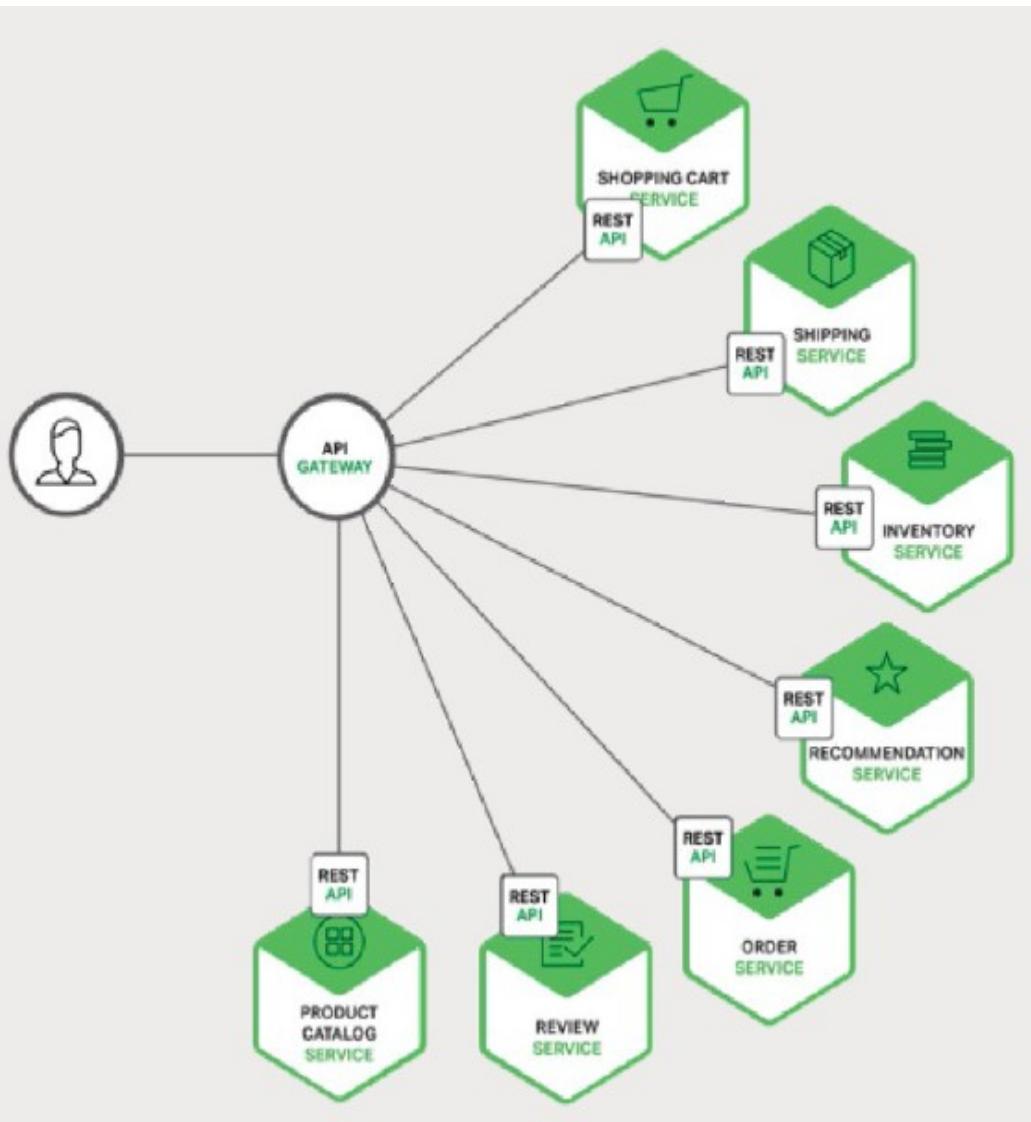


In a conventional micro service based application, The client page has responsibility of calling each service and render all the results on page. The services may be 10, 100 or may be thousand :

The challenges are

- 1) The services may change their position dynamically as per the decision of orchestrator or load balancers.
- 2) The URL of all the services must be on hand. For a few it can be maintained but when services are large in number it becomes difficult.
- 3) The diverse range of protocols and their configuration may require change in code of rendering

- With API Gateways



An API gateway is responsible for

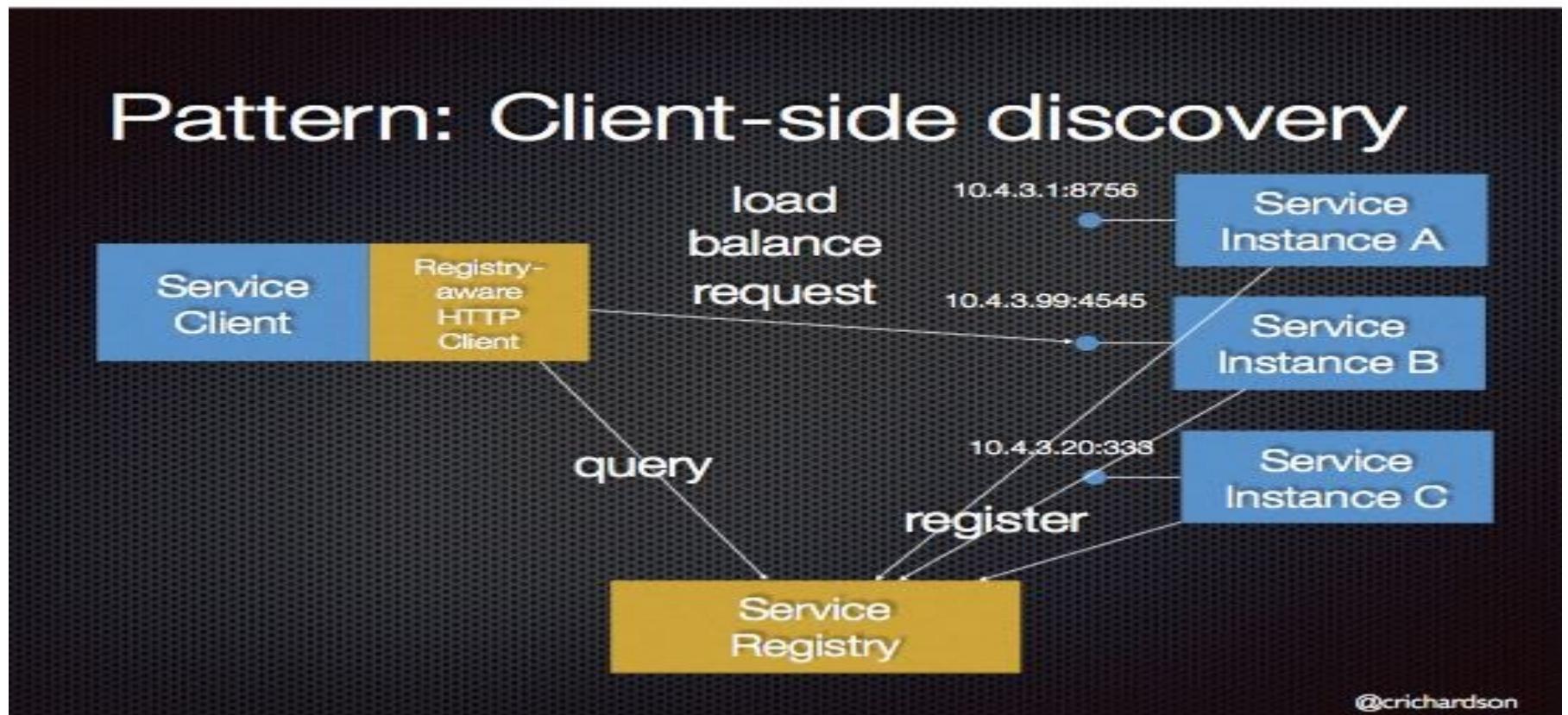
- Request routing
- Composition
- Protocol translation

The client does not have to remember all the service location nor it has to call them individually. On single URL to API server will do everything on behalf of client.

API gateway is an independent server meant for proxying and reverse proxying the client-server request on micro service based applications. The API Gateway will often handle a request by invoking multiple microservices and aggregating the results. It can translate between web protocols such as HTTP and WebSocket and web-unfriendly protocols that are used internally.

Other Aspects of API Gateways : Service Discovery

Services typically need to call one another. A modern microservice-based application typically runs in a virtualized or containerized environments where the number of instances of a service and their locations changes dynamically.



@crichardson

Putting MSA on Amazon Web Services Cloud



Putting MSA on Amazon Web Services Cloud

We are using following services of AWS for MSA

1. AWS Educate / Private Account
2. AWS client for command line tasks
3. IAM for creating users, groups and roles
4. Elastic Compute 2 (EC2) for machine instance
5. Elastic Container Service (ECS) for docker image repository
5. Elastic IP
6. Docker, Minikube and Kubernetes instance on EC2 instance
7. API Gateway – Ingress Controller



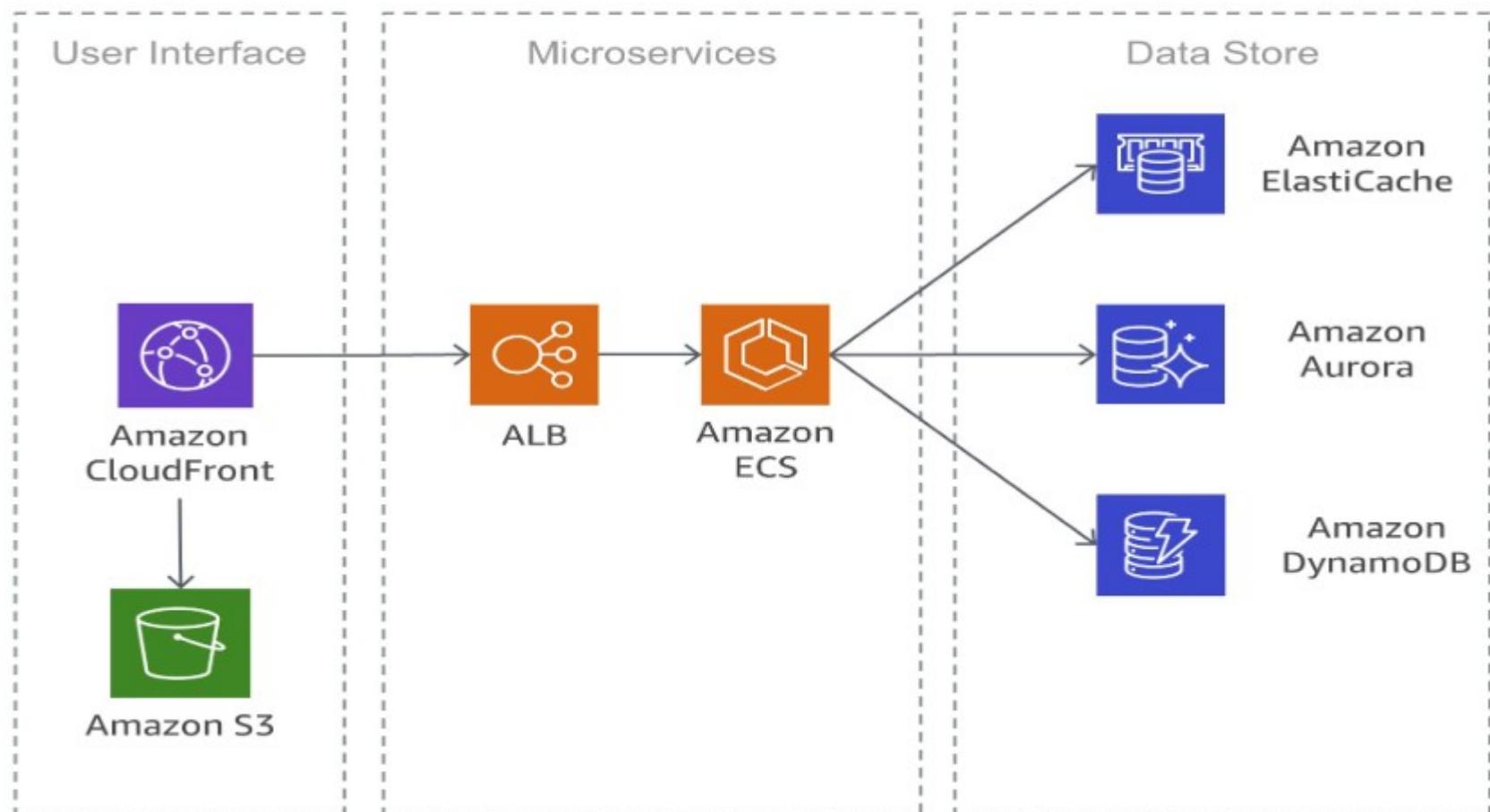
Service Observability

- Log aggregation
- Application metrics
- Audit logging
- Distributed tracing
- Exception tracking
- Health check API
- Log deployments and changes

Deployments - Types

- Multiple service instances per host
- Service instance per host
- Service instance per VM
- Service instance per Container (Docker based)
- Serverless deployment (AWS, Azure, EKS, Google Cloud)
- Service deployment platform (Kubernetes, Swarm)

Amazon Cloud and Micro Services



MICRO SERVICES SOFTWARE ENGINEERING MODEL



EVERY STAGE HAS ITS OWN TOOL
– ALL THE TOOLS TOGATHER FORM A TOOL CHAIN

Modularity (TEAM)



A product not a project

UI - team



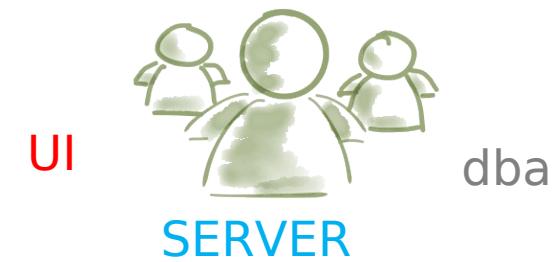
SERVER - team



Dba - team



monolith



microservices

MICRO SERVICES APPLICATION DEVELOPMENT ARCHITECTURE

Some Development platforms like Java or .NET are providing a complete architecture to develop microservice based applications

Java Provides us with 2 strong platforms

1.



2. Java Spring Boot



JAVA MICROPROFILE ARCHITECTURE

- JAX-RS - REST
- CDI
- JSON-P Json
- Common Annotations
- Config
- Metrics
- Health Check
- Fault Tolerance
- JWT Authentication
- OpenTracing
- OpenAPI
- Rest Client

MICRO SERVICES TOOL CHAIN FOR DEVELOPMENT

PLAN

CODE

Code Management



Code Configuration Management



Code Integration



ANSIBLE



Code Testing



Code Packaging



MICRO SERVICES TOOL CHAIN

DEPLOYMENT

CONTAINARIZATION



ORCHESTRATION AND LOAD BALANCING



ACCESS CONTROL or API GATEWAYS



INGRESS

MONITORING



A SCENARIO OF MICRO SERVICES – Developer Prospective

1. The developers write the code in favourite IDE
And Eclipse Micro profile



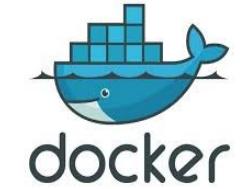
2. The developers chooses the lightest possible server in this cas – Payara Micro Edition / TomEE And Eclipse Micro profile



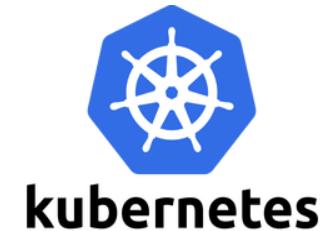
3. The developers Build and Test the project using Maven /Ant



4. The developers create the image and deploy the project on Docker Container with network and volume settings ocker Hub.
Push the Images to Docker Hub



5. The developers Puts the images from Docker Hub to Kubernetes Pods and Services for orchestration and load balancing



5. The developers give access to the outside client using a API gateway



A SCENARIO OF MICRO SERVICES – Client Perspective

1. The Client connects to the IP of API gateway specifying the service



2. The API gateway Calls the K8 cluster and services
K8 responsible for load balancing and persistence



kubernetes

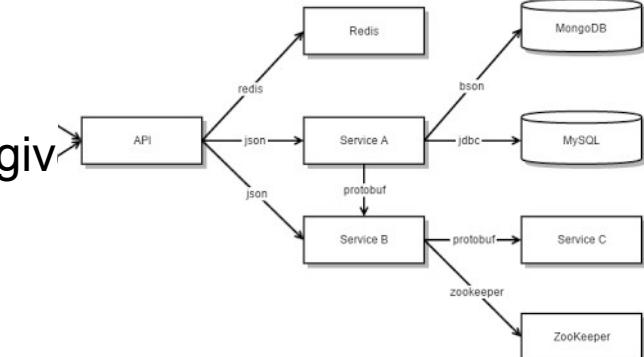
3. The pods invoke the containers



4. The container calls the service end point

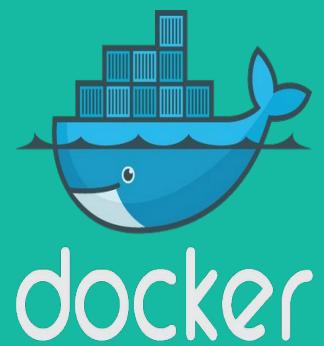
JAX-RS

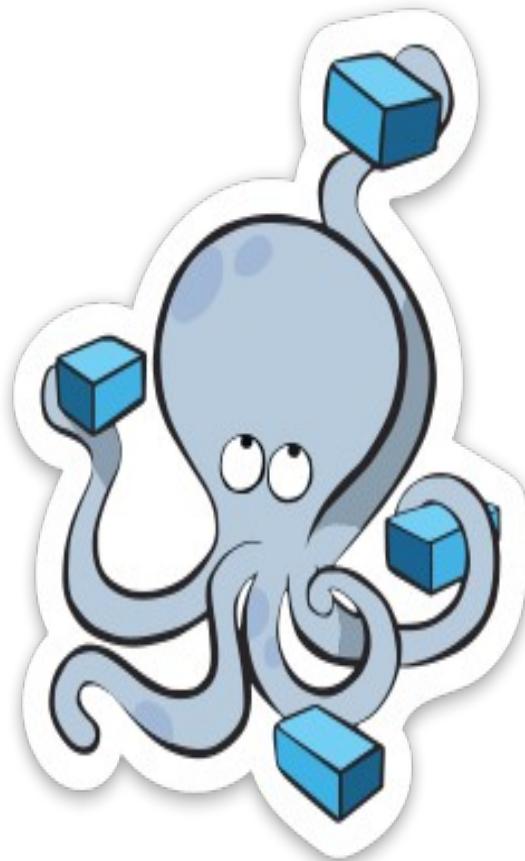
5. Service May call other services internally or database to give output



6. Finally Client gets the response

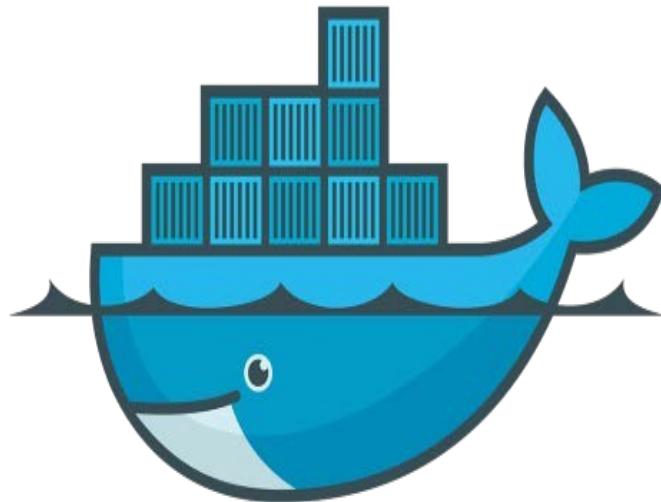
Introduction to Docker





- Lightweight, open, secure platform
- Simplify building, shipping, running apps

What Is Docker?



- Runs natively on Linux or Windows Server
- Runs on Windows or Mac Development machines (with a virtual machine)
- Relies on "images" and "containers"

Some Docker vocabulary



Docker Image

The basis of a Docker container. Represents a full application



Docker Container

The standard unit in which the application service resides and executes



Docker Engine

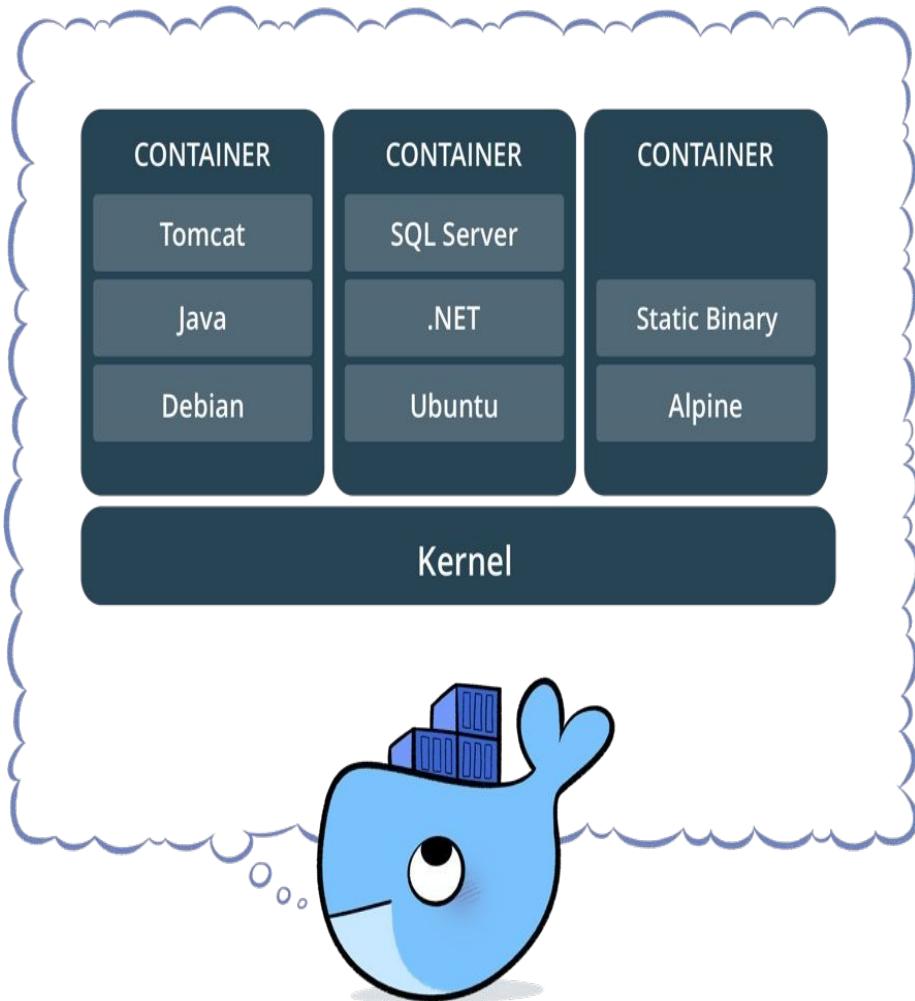
Creates, ships and runs Docker containers deployable on a physical or virtual, host locally, in a datacenter or cloud service provider



Registry Service (Docker Hub(Public) or Docker Trusted Registry(Private))

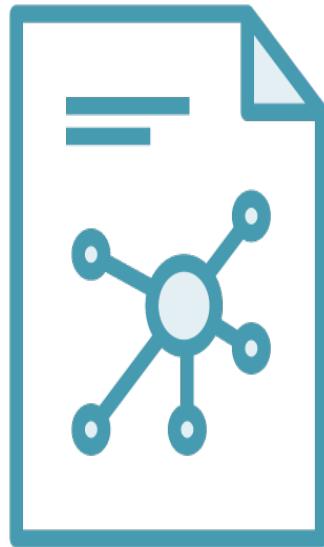
Cloud or server based storage and distribution service for your images

What is a container?



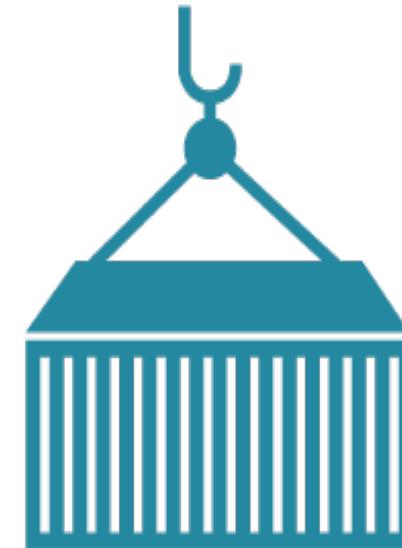
- Standardized packaging for software and dependencies
- Isolate apps from each other
- Share the same OS kernel
- Works for all major Linux distributions
- Containers native to Windows Server 2016

The Role of Images and Containers



Docker Image

Example: Ubuntu with Node.js and
Application Code



Docker Container

Created by using an image. Runs your
application.

Docker containers are NOT VMs

- Easy connection to make
- Fundamentally different architectures
- Fundamentally different benefits



Maquina Virtual



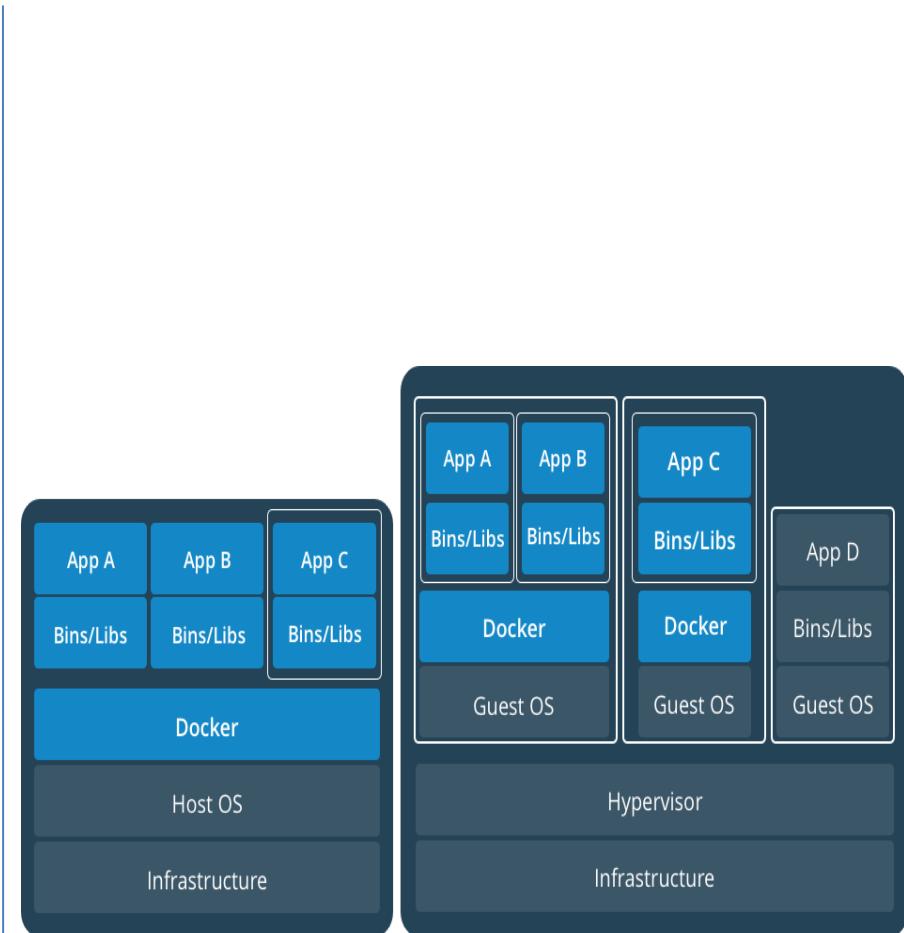
Contenedores

Docker Containers Versus Virtual Machines

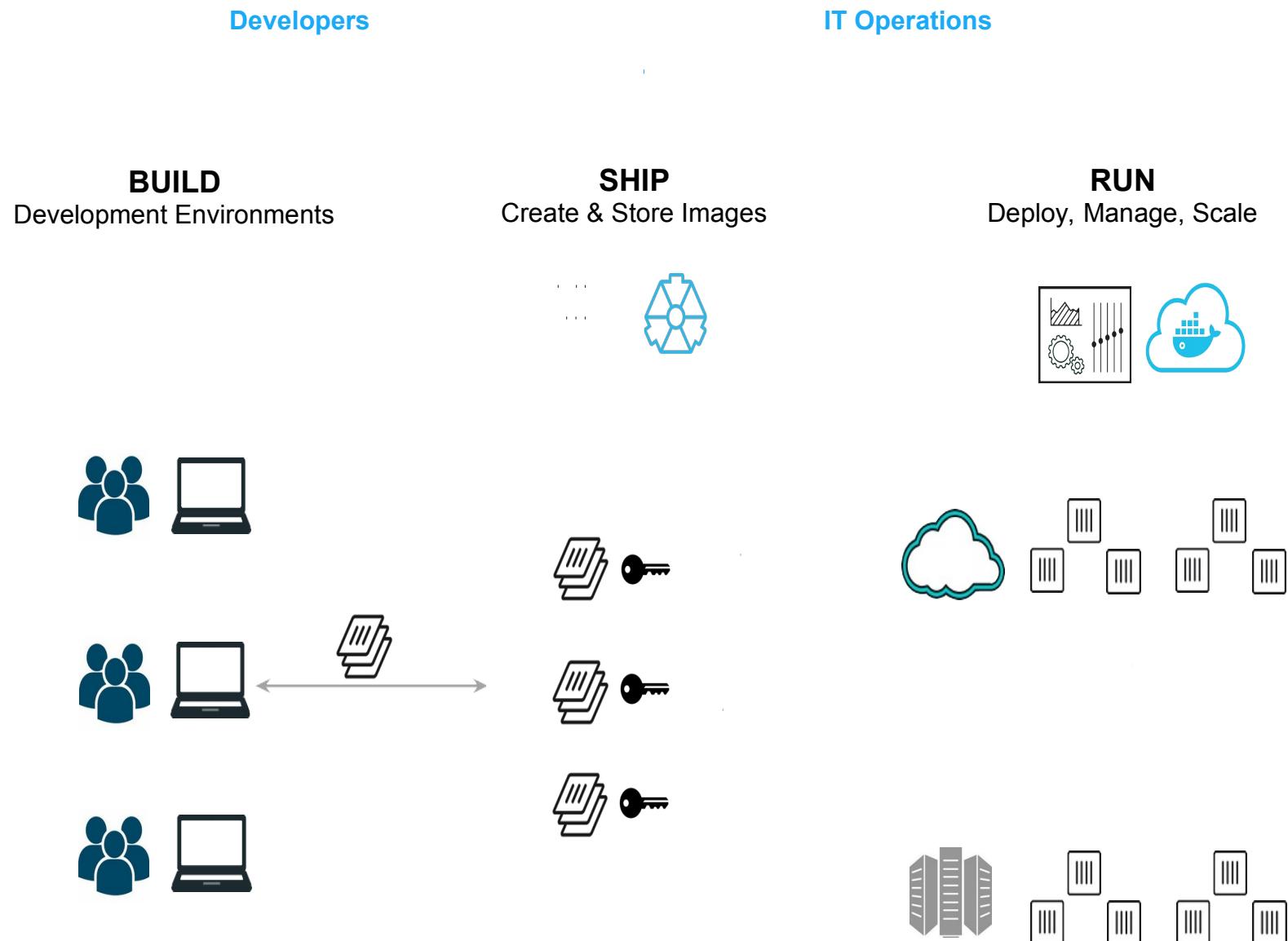


Virtual Machines

Docker Containers



Using Docker: Build, Ship, Run Workflow



Basic Docker Commands

```
$ docker image pull node:latest  
$ docker image ls  
$ docker container run -d -p 5000:5000 --name node node:latest  
$ docker container ps  
  
$ docker container stop node(or <container id>)  
$ docker container rm node (or <container id>)  
$ docker image rmi (or <image id>)  
$ docker build -t node:2.0 .  
$ docker image push node:2.0  
$ docker --help
```

Dockerfile – Linux Example

```
👉 Dockerfile ✘  
1  # Create image based on the official Node 6 image from dockerhub  
2  FROM node:latest  
3  
4  # Create a directory where our app will be placed  
5  RUN mkdir -p /usr/src/app  
6  
7  # Change directory so that our commands run inside this new directory  
8  WORKDIR /usr/src/app  
9  
10 # Copy dependency definitions  
11 COPY package.json /usr/src/app  
12  
13 # Install dependencies  
14 RUN npm install  
15  
16 # Get all the code needed to run the app  
17 COPY . /usr/src/app  
18  
19 # Expose the port the app runs in  
20 EXPOSE 4200  
21  
22 # Serve the app  
23 CMD ["npm", "start"]
```

- Instructions on how to build a Docker image
- Looks very similar to “native” commands
- Important to optimize your Dockerfile

Section 2:

Anatomy of a Docker Container Docker Volume

Volume Use Cases



Let's Go Back to Our Dockerfile

```
👉 Dockerfile ✘  
1  # Create image based on the official Node 6 image from dockerhub  
2  FROM node:latest  
3  
4  # Create a directory where our app will be placed  
5  RUN mkdir -p /usr/src/app  
6  
7  # Change directory so that our commands run inside this new directory  
8  WORKDIR /usr/src/app  
9  
10 # Copy dependency definitions  
11 COPY package.json /usr/src/app  
12  
13 # Install dependencies  
14 RUN npm install  
15  
16 # Get all the code needed to run the app  
17 COPY . /usr/src/app  
18  
19 # Expose the port the app runs in  
20 EXPOSE 4200  
21  
22 # Serve the app  
23 CMD ["npm", "start"]
```

Each Dockerfile Command Creates a Layer

...

EXPOSE

COPY

WORKDIR

RUN

FROM

Kernel

Docker Image Pull: Pulls Layers

```
Alexander@DESKTOP-90ATKET MINGW64 ~/Docker/Demo
$ docker pull nginx:latest
latest: Pulling from library/nginx
bc95e04b23c0: Pull complete
f3186e650f4e: Pull complete
9ac7d6621708: Pull complete
Digest: sha256:b81f317384d7388708a498555c28a7cce778a8f291d90021208b3eba3fe74887
Status: Downloaded newer image for nginx:latest
```

Docker Volumes

- Volumes mount a directory on the host into the container at a specific location
- Can be used to share (and persist) data between containers
 - Directory persists after the container is deleted
 - Unless you explicitly delete it
- Can be created in a Dockerfile or via CLI

Why Use Volumes

- Mount local source code into a running container

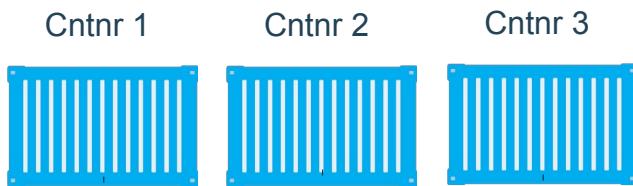
```
docker container run -v $(pwd):/usr/src/app/ myapp
```
- Improve performance
 - As directory structures get complicated traversing the tree can slow system performance
- Data persistence

Section 3: Networking



What is Docker Bridge Networking

Docker host

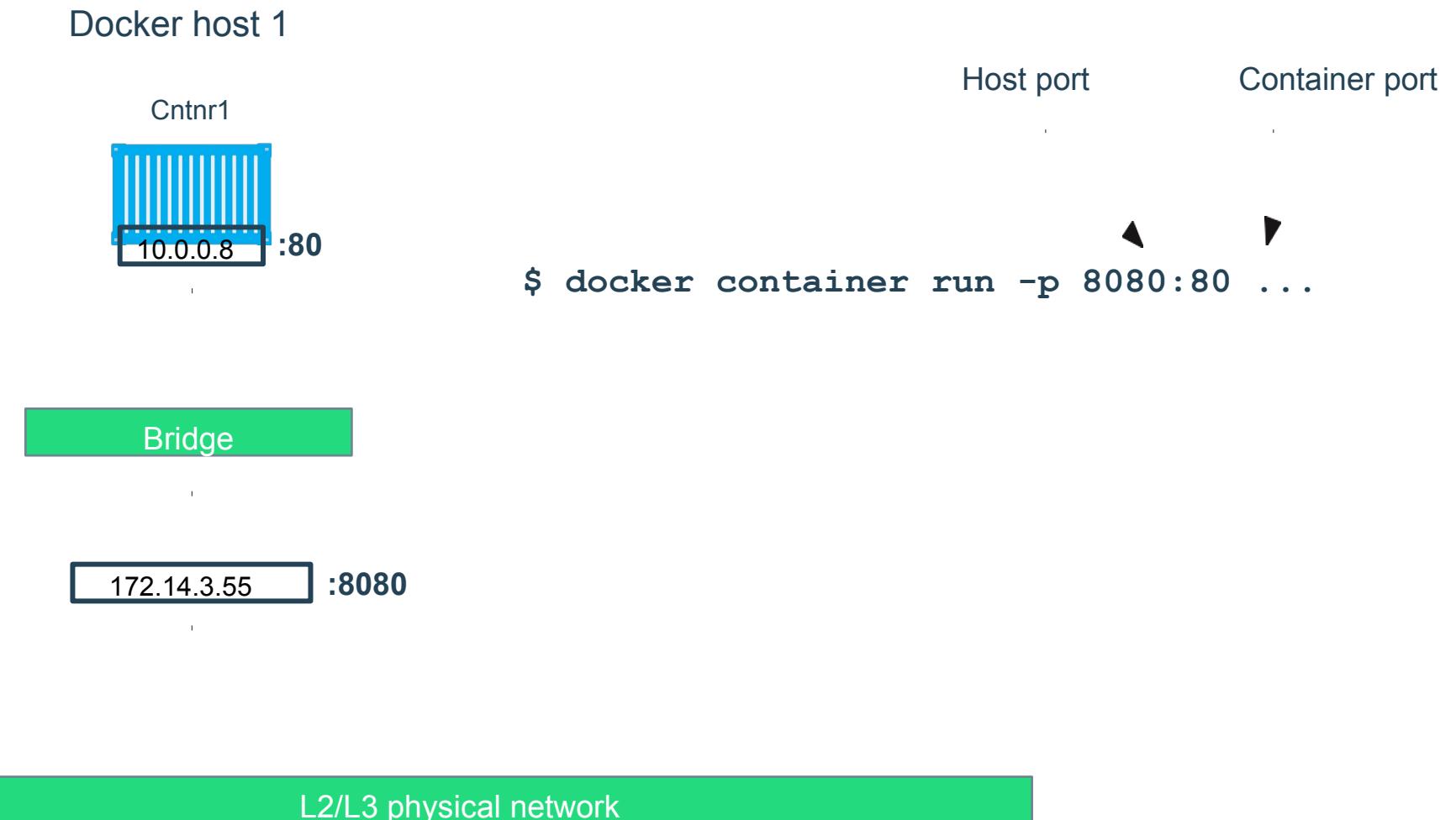


Docker host



```
docker network create -d bridge --name bridgenet1
```

Docker Bridge Networking and Port Mapping



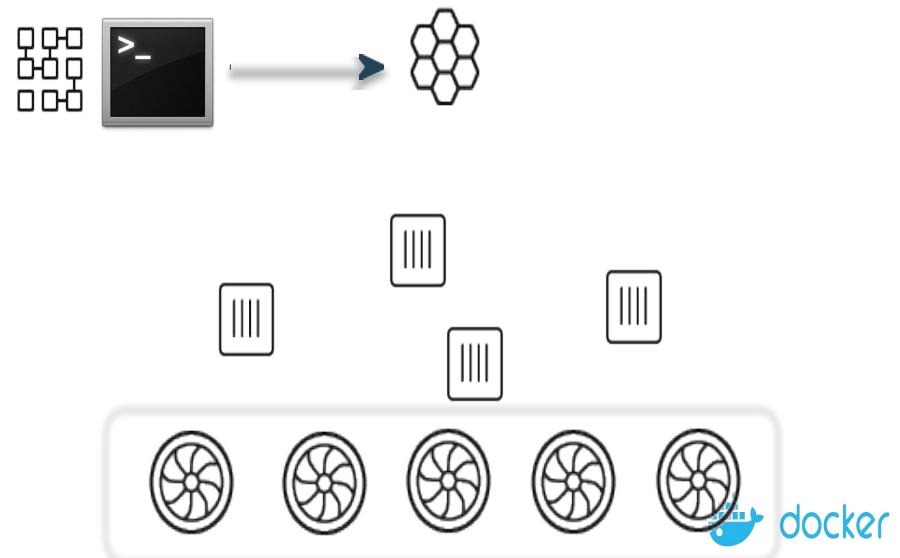
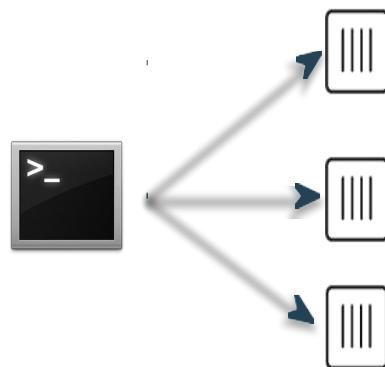
Section 4: Docker Compose



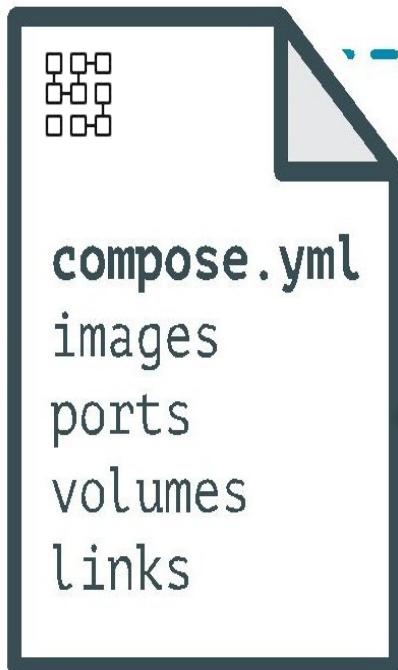
Docker Compose: Multi Container Applications

- Build and run one container at a time
- Manually connect containers together
- Must be careful with dependencies and start up order

- Define multi container app in compose.yml file
- Single command to deploy entire app
- Handles container dependencies
- Works with Docker Swarm, Networking, Volumes, Universal Control Plane



Docker Compose: Multi Container Applications



```
version: '2' # specify docker-compose version

# Define the services/containers to be run
services:
  angular: # name of the first service
    build: client # specify the directory of the Dockerfile
    ports:
      - "4200:4200" # specify port forwarding

  express: #name of the second service
    build: api # specify the directory of the Dockerfile
    ports:
      - "3977:3977" #specify ports forwarding

  database: # name of the third service
    image: mongo # specify image to build container from
    ports:
      - "27017:27017" # specify port forwarding
```

Docker Compose: Scale Container Applications

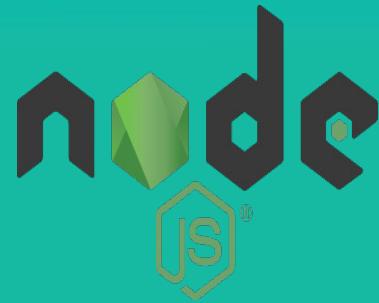


Demo

Angular

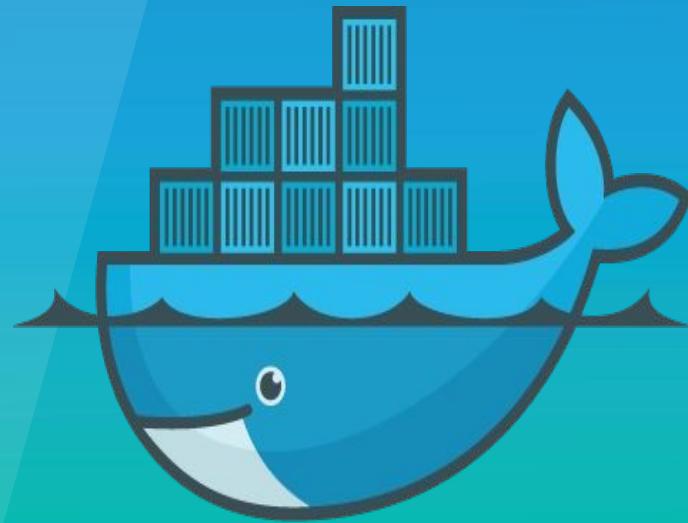


Node.js/Express



Mongo DB

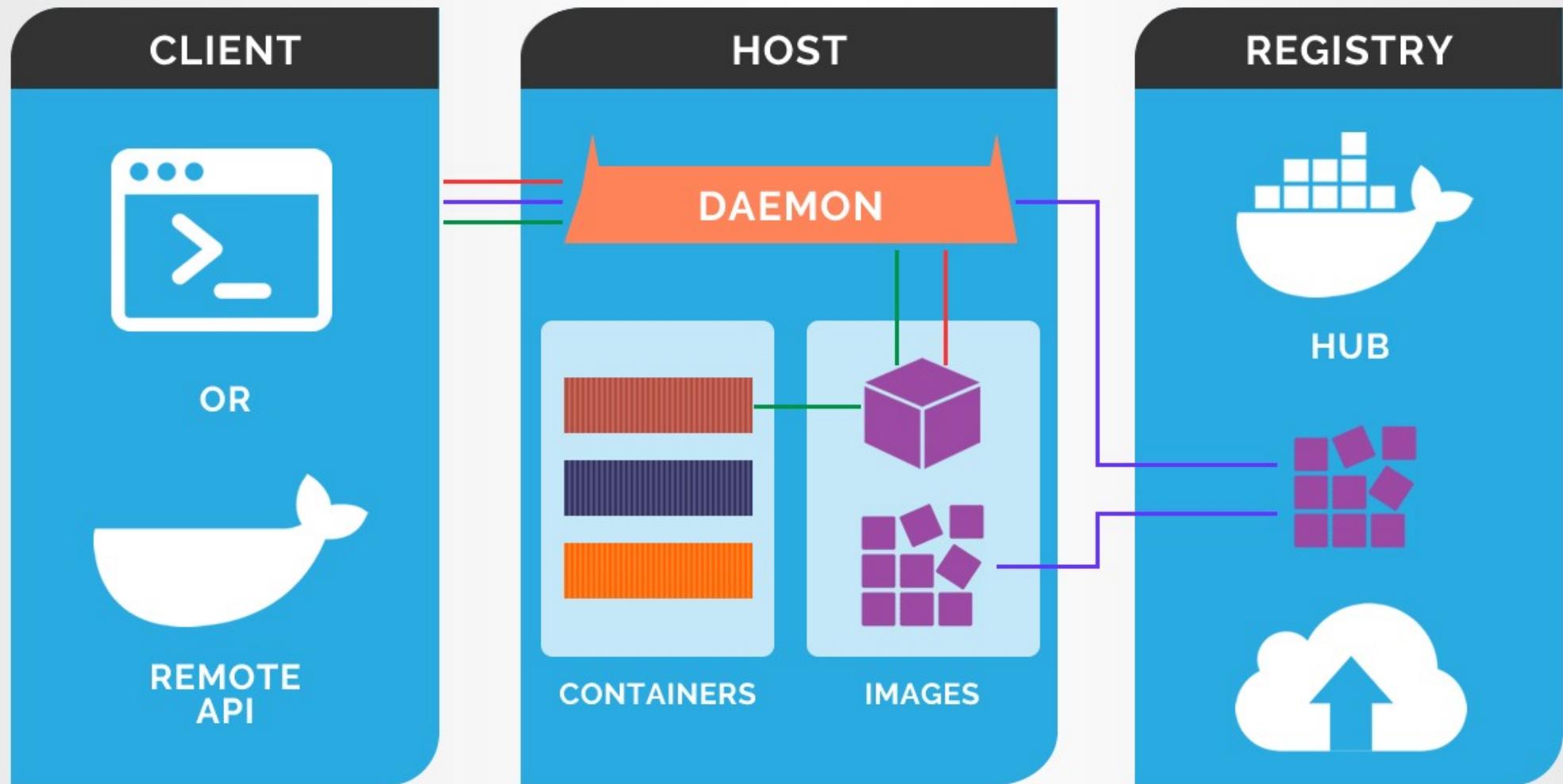




docker

Docker Architecture

DOCKER ARCHITECTURE

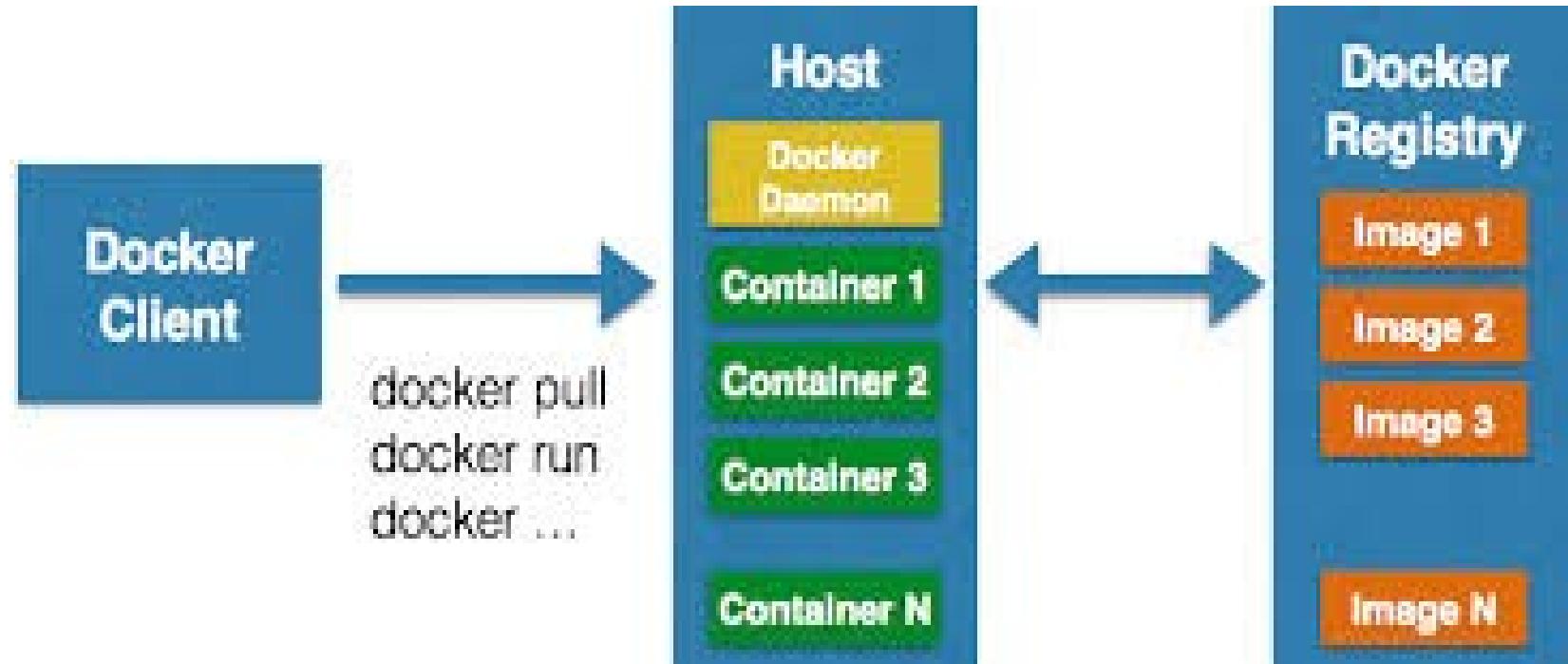


BUILD

PULL

RUN

Docker Execution



Steps of Using Docker

Installing Docker

|

```
$ apt-get install docker-ce
```

2. Registering on dockerhub (www.dockerhub.com)

3. Pull Image from docker hub

```
$ docker pull <image_name>:tagname  
eg docker pull mysql/mysql-server:latest
```

Steps of Using Docker

4. create your own image with the help of existing image on docker hub

File Name : Dockerfile |

FROM payara/micro ## image from docker hub

COPY ExampleProject1.war \$DEPLOY_DIR

-- Navigate to the folder where Dockerfile is lying

\$ docker build -t mywebapp:1.0 .

5. Now Run the image as container

\$ docker run –name=web -p 9070:8080 mywebapp:1.0

6. Open the browser and use url <http://localhost:9070/MyWebApp/>

Advance users can use docker-compose to run multiple images in one command

KUBERNETES – THE MASTER ORCHESTRATOR



kubernetes

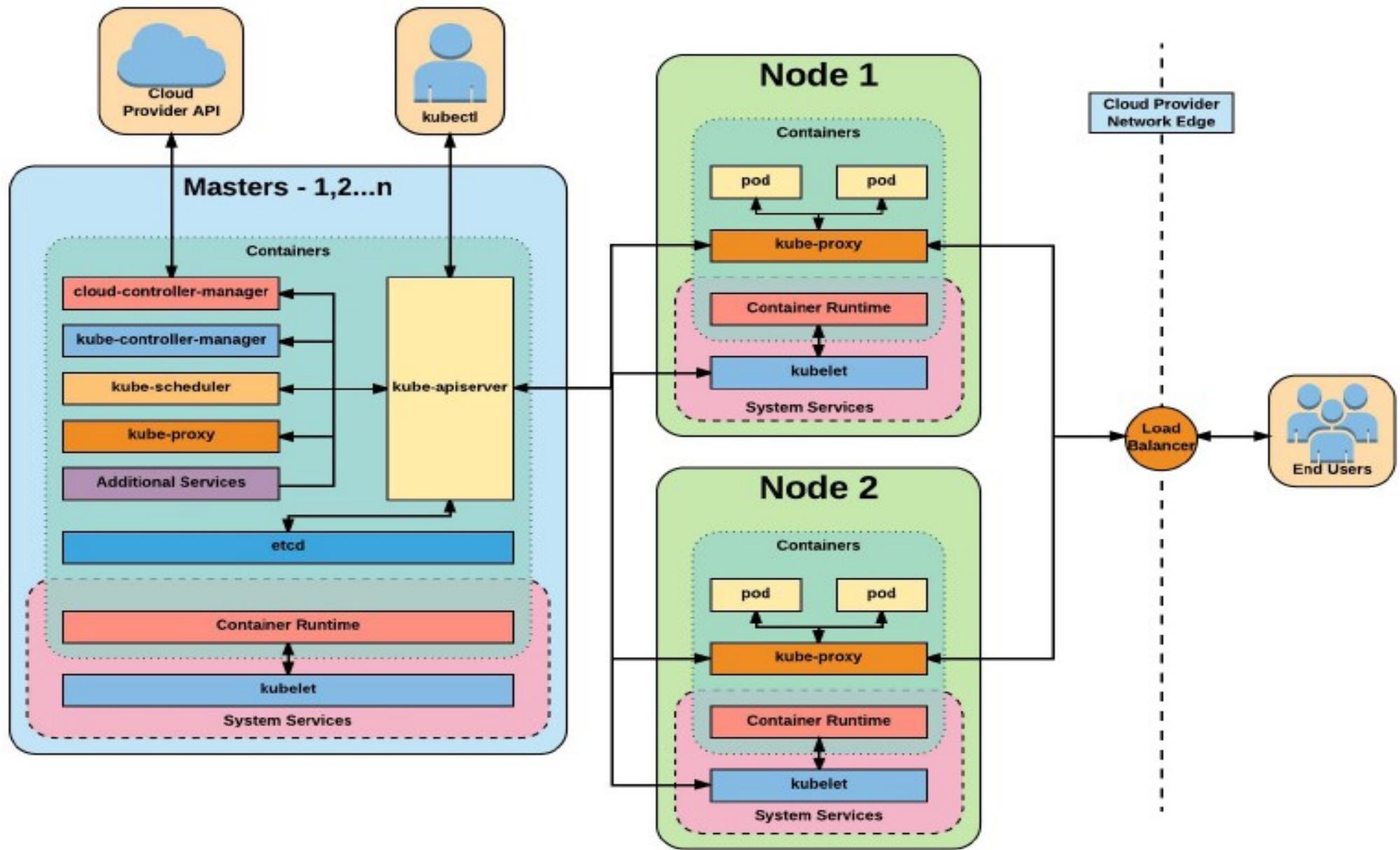
KUBERNETES – THE MASTER ORCHESTRATOR



kubernetes

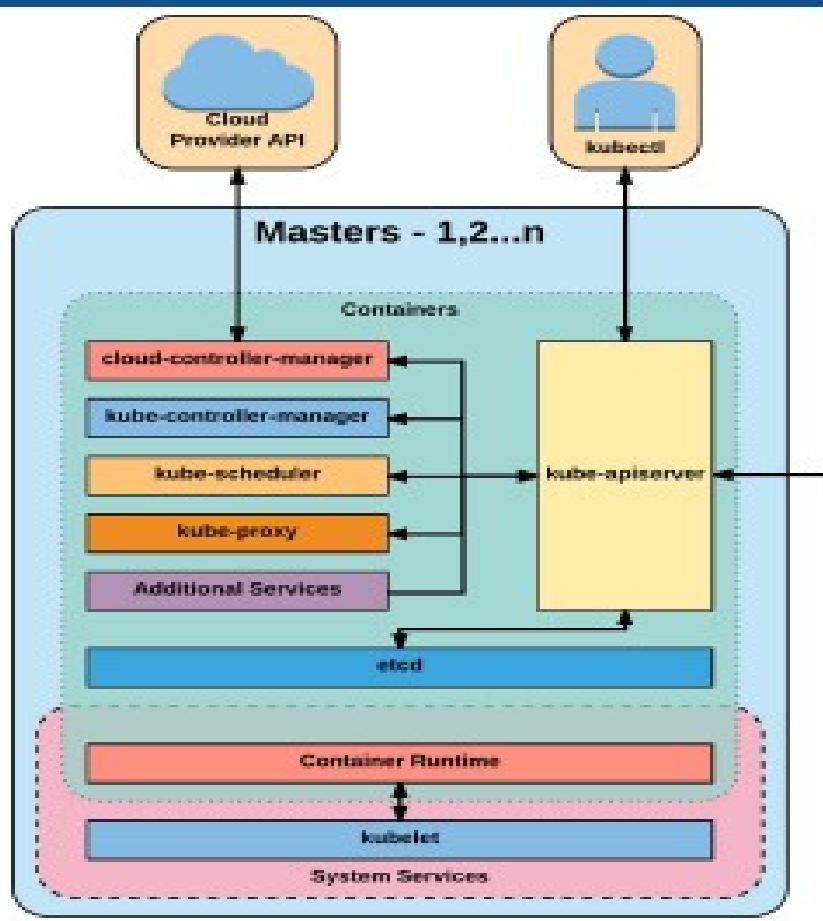
Kubernetes is a portable, extensible, open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

KUBERNETES - ARCHITECTURE



KUBERNETES - ARCHITECTURE

Master Components



- Kube-apiserver
- Etcd
- Kube-controller-manager
- Cloud-controller-manager
- Kube-scheduler

KUBERNETES – ARCHITECTURE

kube-apiserver

The apiserver provides a forward facing REST interface into the kubernetes control plane and datastore. All clients, including nodes, users and other applications interact with kubernetes **strictly** through the API Server.

It is the true core of Kubernetes acting as the gatekeeper to the cluster by handling authentication and authorization, request validation, mutation, and admission control in addition to being the front-end to the backing datastore.

etcd

Etcd acts as the cluster datastore; providing a strong, consistent and highly available key-value store used for persisting cluster state.

KUBERNETES – ARCHITECTURE

kube-controller-manager

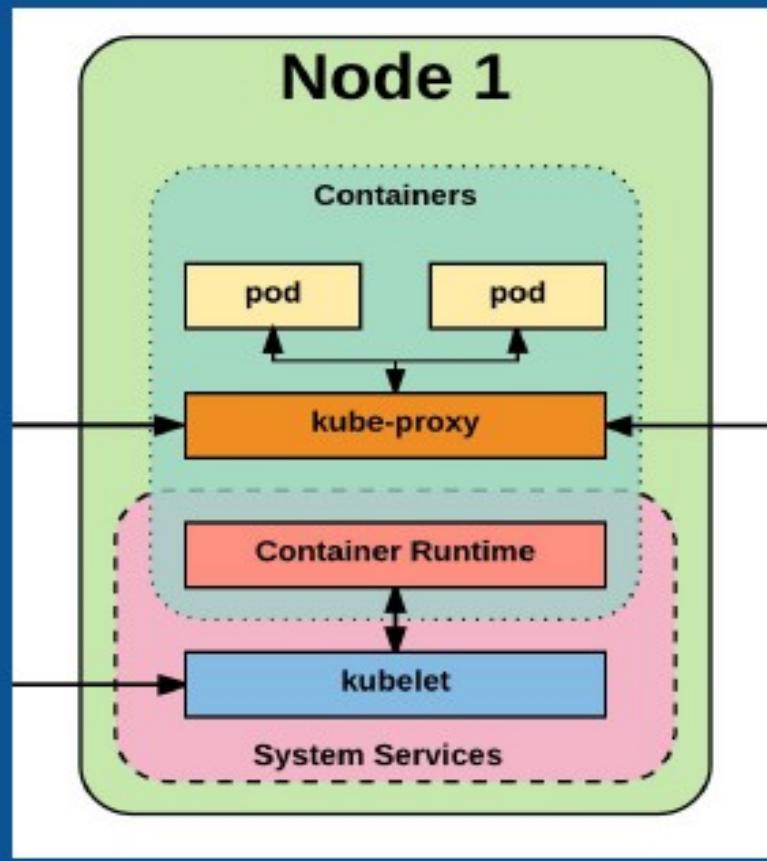
The controller-manager is the primary daemon that manages all core component control loops. It monitors the cluster state via the apiserver and steers the cluster towards the desired state.

kube-scheduler

Kube-scheduler is a verbose policy-rich engine that evaluates workload requirements and attempts to place it on a matching resource. These requirements can include such things as general hardware reqs, affinity, anti-affinity, and other custom resource requirements.

KUBERNETES – ARCHITECTURE

Node Components



- Kubelet
- Kube-proxy
- Container runtime engine

KUBERNETES – ARCHITECTURE

kubelet

Acts as the node agent responsible for managing pod lifecycle on its host. Kubelet understands YAML container manifests that it can read from several sources:

- File path
- HTTP Endpoint
- Etcd watch acting on any changes
- HTTP Server mode accepting container manifests over a simple API.

kube-proxy

Manages the network rules on each node and performs connection forwarding or load balancing for Kubernetes cluster services.

Available Proxy Modes:

- Userspace
- iptables

KUBERNETES – ARCHITECTURE

Container Runtime

With respect to Kubernetes, A container runtime is a CRI (Container Runtime Interface) compatible application that executes and manages containers.

- Containerd (docker)
- Cri-o
- Rkt
- Kata (formerly clear and hyper)
- Virtlet (VM CRI compatible runtime)

Containers in a pod exist within the same network namespace and share an IP; allowing for intrapod communication over *localhost*.

Pods are given a cluster unique IP for the duration of its lifecycle, but the pods themselves are fundamentally ephemeral.

Services are given a persistent cluster unique IP that spans the Pods lifecycle.

External Connectivity is generally handed by an integrated cloud provider or other external entity (load balancer)

KUBERNETES – ARCHITECTURE

Kubernetes Concepts - Core

Cluster - A collection of hosts that aggregate their available resources including cpu, ram, disk, and their devices into a usable pool.

Master - The master(s) represent a collection of components that make up the control plane of Kubernetes. These components are responsible for all cluster decisions including both scheduling and responding to cluster events.

Node - A single host, physical or virtual capable of running pods. A node is managed by the master(s), and at a minimum runs both kubelet and kube-proxy to be considered part of the cluster.

Namespace - A logical cluster or environment. Primary method of dividing a cluster or scoping access.

KUBERNETES – ARCHITECTURE

Concepts - Core (cont.)

Label - Key-value pairs that are used to **identify**, describe and group together related sets of objects. Labels have a strict syntax and available character set. *

Annotation - Key-value pairs that contain **non-identifying** information or metadata. Annotations do not have the the syntax limitations as labels and can contain structured or unstructured data.

Selector - Selectors use labels to filter or select objects. Both equality-based (=, ==, !=) or simple key-value matching selectors are supported.

KUBERNETES – ARCHITECTURE

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Labels, and Annotations, and Selectors

Labels:
app: nginx
tier: frontned

Annotations
description: “nginx frontend”

Selector:
app: nginx
tier: frontend

KUBERNETES – ARCHITECTURE

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchExpressions:
      - {key: app, operator: In, values: [nginx]}
      - {key: tier, operator: In, values: [frontend]}
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Set-based selectors

Valid Operators:

- In
- NotIn
- Exists
- DoesNotExist

Supported Objects with set-based selectors:

- Job
- Deployment
- ReplicaSet
- DaemonSet
- PersistentVolumeClaims

KUBERNETES – ARCHITECTURE

Concepts - Workloads

Pod - A pod is the smallest unit of work or management resource within Kubernetes. It is comprised of one or more containers that share their storage, network, and context (namespace, cgroups etc).

ReplicationController - Method of managing pod replicas and their lifecycle. Their scheduling, scaling, and deletion.

ReplicaSet - Next Generation ReplicationController. Supports set-based selectors.

Deployment - A declarative method of managing stateless Pods and ReplicaSets. Provides rollback functionality in addition to more granular update control mechanisms.

KUBERNETES – ARCHITECTURE

```
apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  minReadySeconds: 10
  strategy:
    type: RollingUpdate
    rollingUpdate:
      maxSurge: 5
      maxUnavailable: 2
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

Deployment

Contains configuration of how updates or 'deployments' should be managed in addition to the pod template used to generate the ReplicaSet.

ReplicaSet

Generated ReplicaSet from Deployment spec.

```
apiVersion: apps/v1beta2
kind: ReplicaSet
metadata:
  name: nginx
  annotations:
    description: "nginx frontend"
  labels:
    app: nginx
    tier: frontend
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
      tier: frontend
  template:
    metadata:
      labels:
        app: nginx
        tier: frontend
    spec:
      containers:
        - name: nginx
          image: nginx:latest
          ports:
            - containerPort: 80
```

KUBERNETES – ARCHITECTURE

Service

- Acts as the unified method of accessing replicated pods.
- Four major Service Types:
 - ClusterIP - Exposes service on a strictly cluster-internal IP (default)
 - NodePort - Service is exposed on each node's IP on a statically defined port.
 - LoadBalancer - Works in combination with a cloud provider to expose a service outside the cluster on a static external IP.
 - ExternalName - used to reference endpoints OUTSIDE the cluster by providing a static internally referenced DNS name.

```
kind: Service
apiVersion: v1
metadata:
  name: nginx
spec:
  type: ClusterIP
  selector:
    app: nginx
  ports:
    - protocol: TCP
      port: 80
      targetPort: 80
```

KUBERNETES – ARCHITECTURE

Ingress Controller

- Deployed as a pod to one or more hosts
- Ingress controllers are an external controller with multiple options.
 - Nginx
 - HAproxy
 - Contour
 - Traefik
- Specific features and controller specific configuration is passed through annotations.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  annotations:
    kubernetes.io/ingress.class: "nginx"
    name: nginx-ingress
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /nginx
        backend:
          service: nginx
          servicePort: 80
```

KUBERNETES – ARCHITECTURE

Concepts - Storage

Volume - Storage that is tied to the Pod Lifecycle, consumable by one or more containers within the pod.

PersistentVolume - A PersistentVolume (PV) represents a storage resource. PVs are commonly linked to a backing storage resource, NFS, GCEPersistentDisk, RBD etc. and are provisioned ahead of time. Their lifecycle is handled independently from a pod.

PersistentVolumeClaim - A PersistentVolumeClaim (PVC) is a request for storage that satisfies a set of requirements instead of mapping to a storage resource directly. Commonly used with dynamically provisioned storage.

StorageClass - Storage classes are an abstraction on top of an external storage resource. These will include a provisioner, provisioner configuration parameters as well as a PV reclaimPolicy.

KUBERNETES – ARCHITECTURE

Persistent Volumes

```
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv-nfs
spec:
  capacity:
    storage: 500Gi
  volumeMode: Filesystem
  accessModes:
    - ReadWriteMany
  persistentVolumeReclaimPolicy: Recycle
  storageClassName: slow
  mountOptions:
    - hard
    - nfsvers=4.1
  nfs:
    path: /data
    server: 10.255.100.10
```

- PVs are a cluster-wide resource
- Not directly consumable by a Pod
- PV Parameters:
 - Capacity
 - accessModes
 - ReadOnlyMany (ROX)
 - ReadWriteOnce (RWO)
 - ReadWriteMany (RWX)
 - persistentVolumeReclaimPolicy
 - Retain
 - Recycle
 - Delete
 - StorageClass

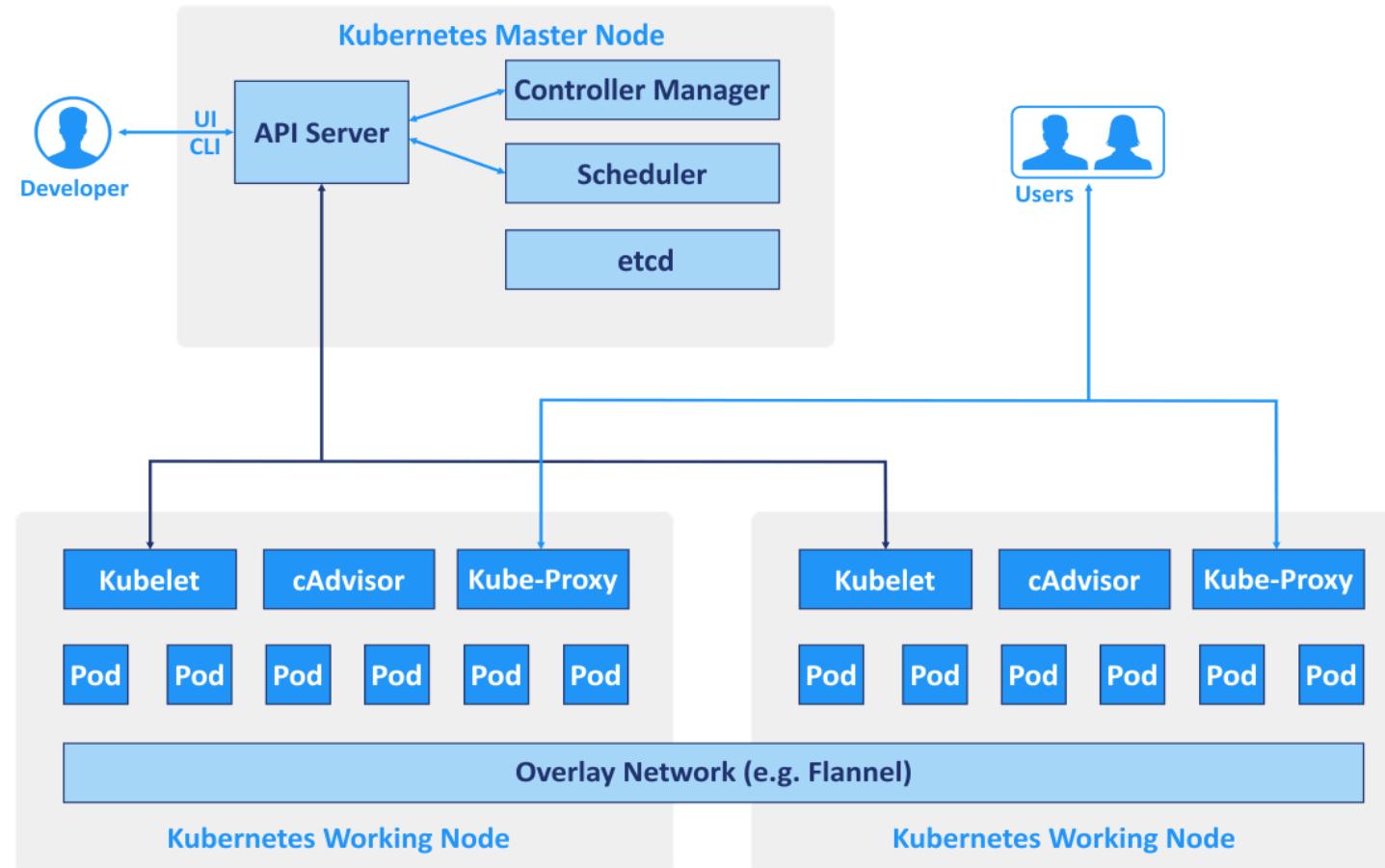
KUBERNETES – ARCHITECTURE

Persistent Volume Claims

```
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: my-nfs-pvc
spec:
  accessModes:
  - ReadWriteMany
  resources:
    requests:
      storage: 50Gi
  storageClass: slow
```

- PVCs are scoped to namespaces
- Supports accessModes like PVs
- Uses resource request model similar to Pods
- Claims will consume storage from matching PVs or StorageClasses based on *storageClass* and *selectors*.

Working of Kubernetes



Pod : Container manager ,
etcd : Persistent Storage
minikube : Managed Kubernetes

Kube-Proxy : Service Connector
ambassador : API gateway
Ingress : Load Balancer

Steps of Using Kubernetes

1. Install minikube – a single cluster kubernetes engine (see on line)
2. Install kubectl |
3. Using kubectl we can issue commands to minikube

Pods : They contain container instances (one or more replicas)

Services : The are front end connecting to respective pods

-- We write a yaml file for services, pods, networks and storages

4. command to create service and pods

```
$ kubectl apply -f stockmanager-service.yaml
```

5. viewing services and pods

```
$ kubectl get svc and kubectl get pods
```

6. execute the service

```
$ minikube service service_name
```

Micro Services Transactions with Eventual Consistency

Distributed
Transactions Patterns In
**Microservice
Architecture**



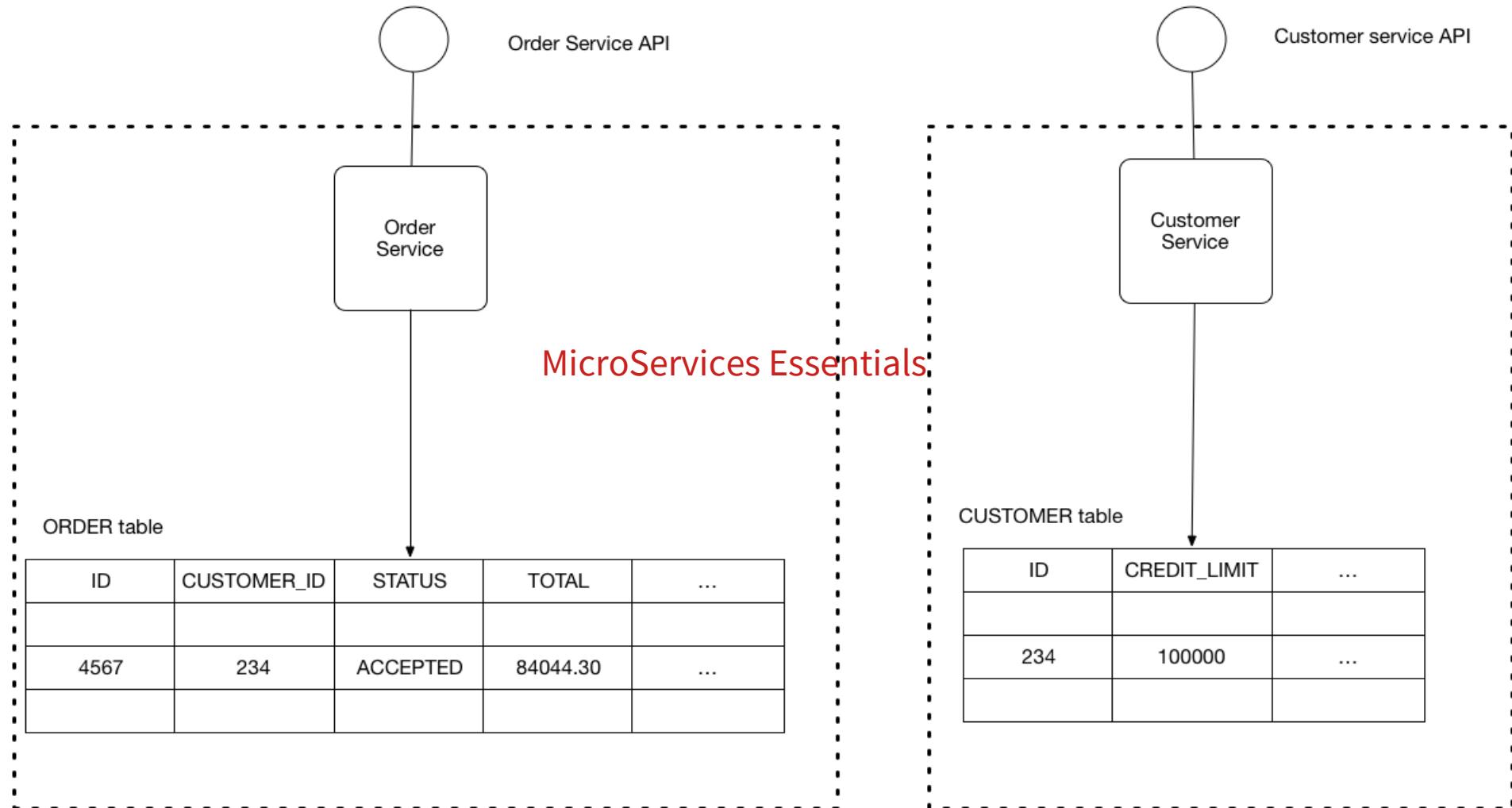
MicroServices Essentials -

Database per Service

- The service's database is effectively part of the implementation of that service. It cannot be accessed directly by other services. If you are using a relational database then the options are:
- **Private-tables-per-service** – each service owns a set of tables that must only be accessed by that service
- **Schema-per-service** – each service has a database schema that's private to that service
- **Database-server-per-service** – each service has it's own database server.

MicroServices Essentials

Database per Service

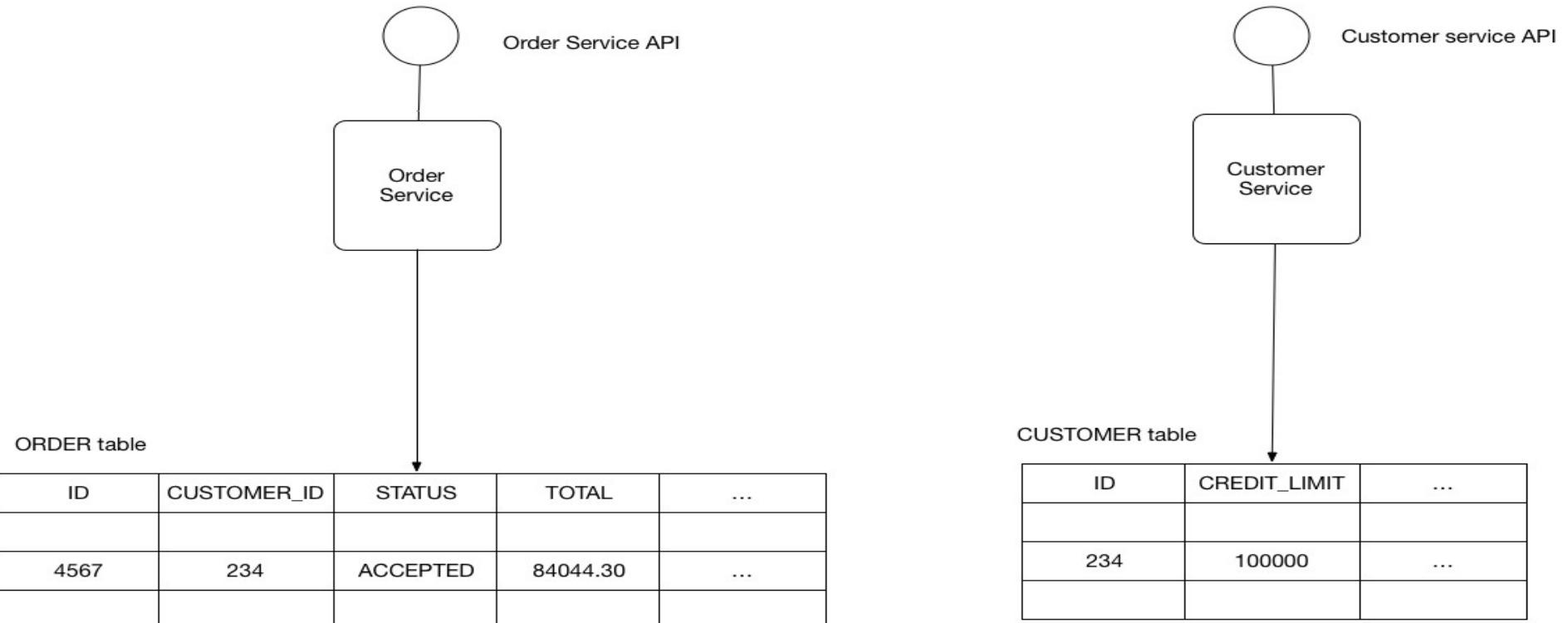


MicroServices Essentials

Shared Database

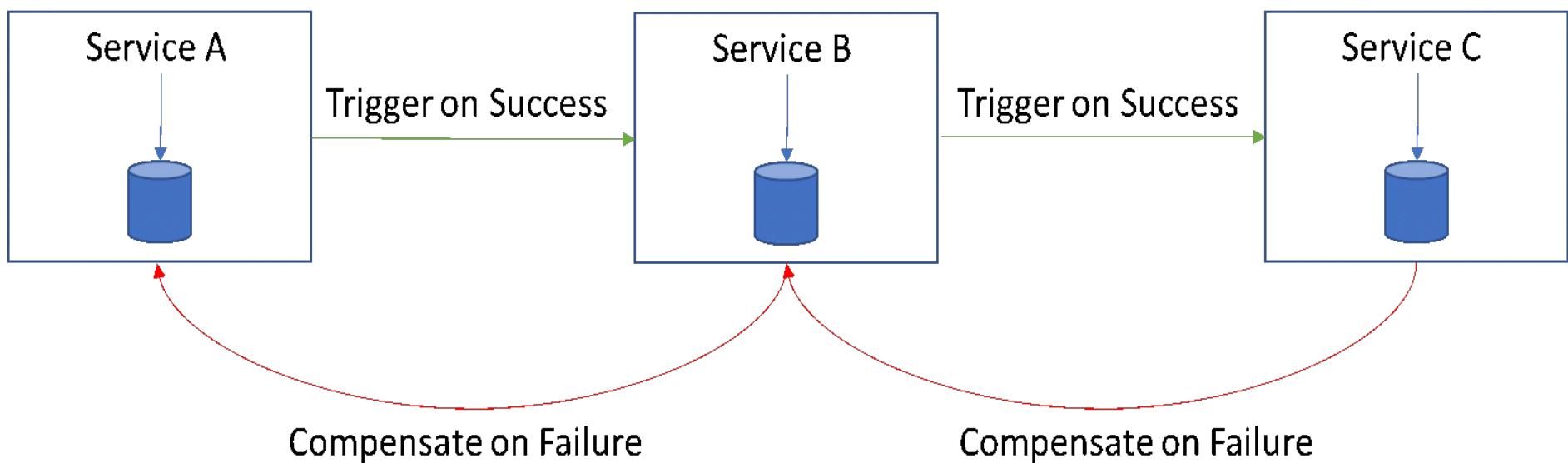
The reason for going for shared database for multiple services

- Some transaction spawn over many services
- Some operations require data from multiple services
- Database replication is required



Data Transactions : Eventual Consistency

WITH SAGA PATTERN

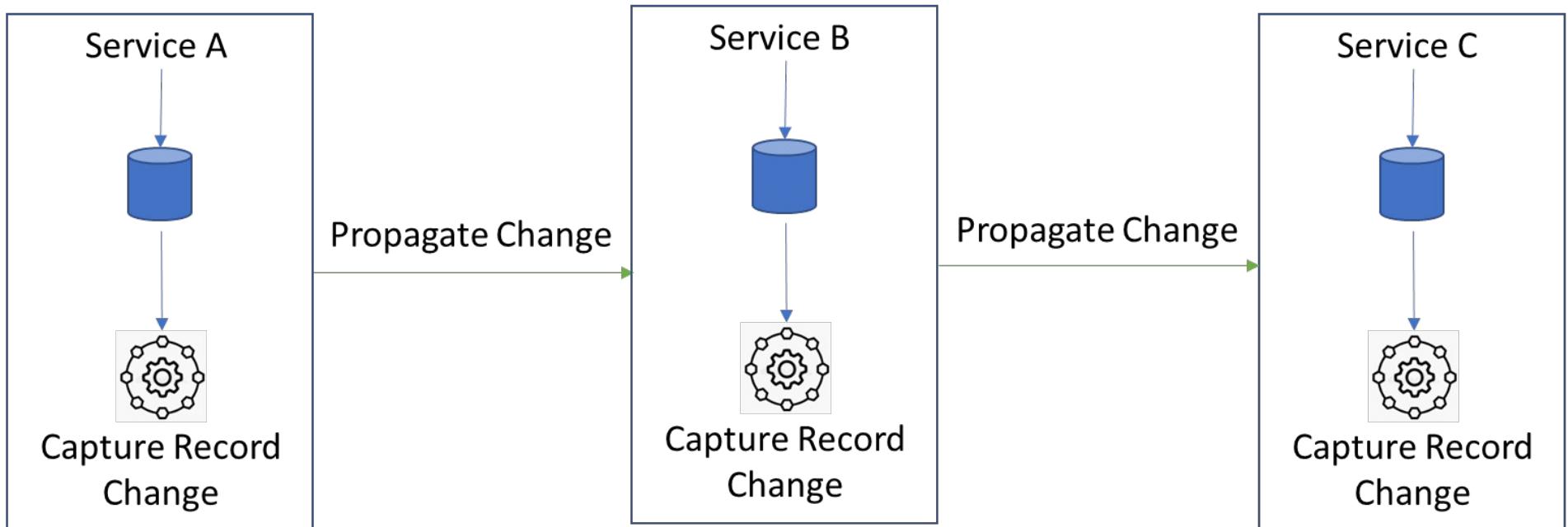


Data Transactions : Eventual Consistency

Transactions that span multiple services are viewed as a **SAGA**/chain of simple atomic local transactions at each service level. Thus, a service completes and commits its transaction, notifies the next service in the chain with an event/message to trigger the next local transaction and so on and so forth. If one transaction in this chain fails, owing to any particular reason, it basically triggers an undo operation coming backward in the chain. It is thus imperative to address pattern failures while designing the architecture

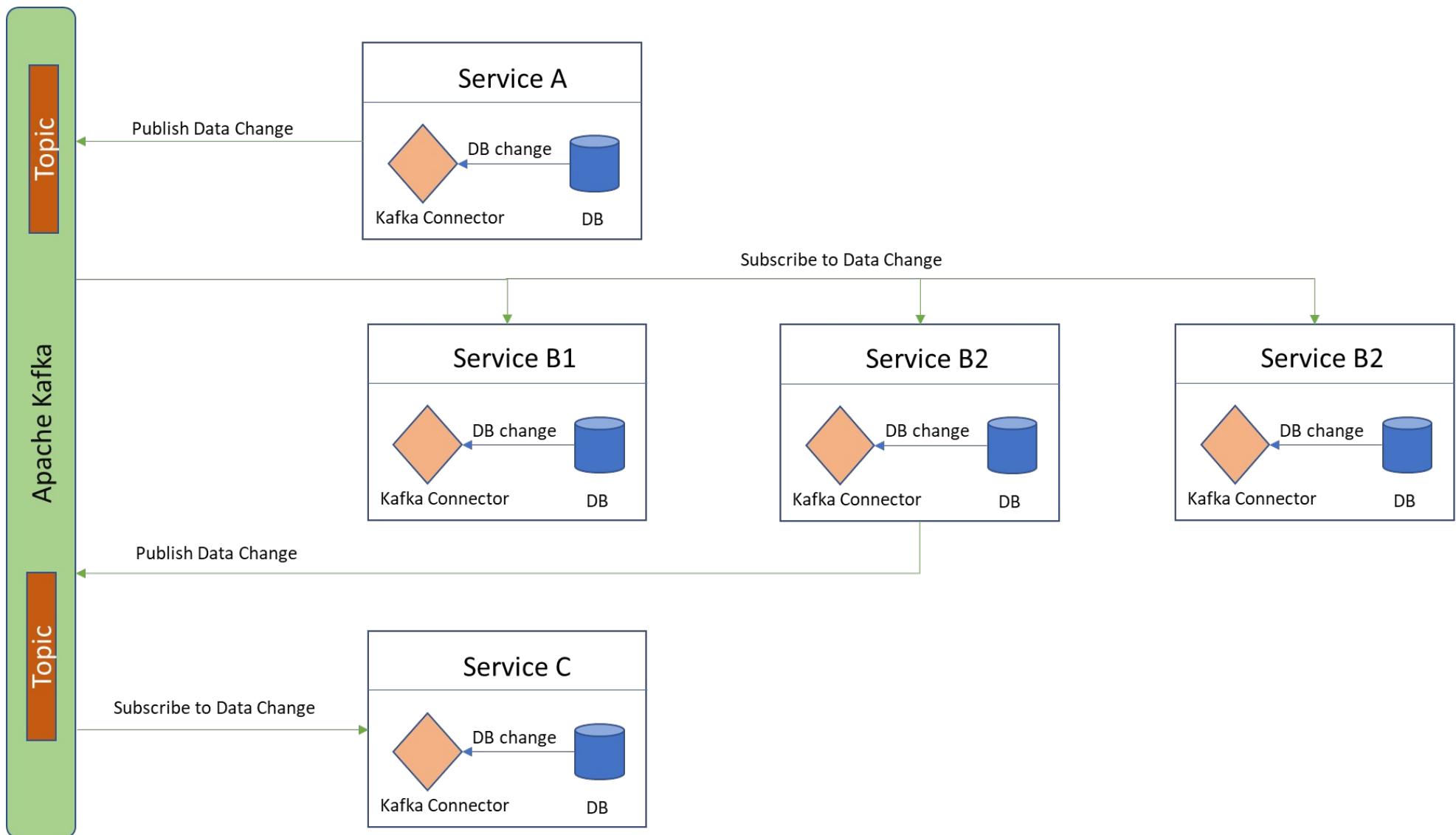
Data Transactions : Eventual Consistency

WITH TRANSACTION LOGS



Data Transactions : Eventual Consistency

WITH MESSAGE QUEUES



Data Transactions : Eventual Consistency : BASE vs ACID

For Micro Services

BASE is an acronym for Basically Available, Soft state and Eventual Consistency

For Conventional

ACID is an acronym for Atomicity, Consistent, Isolated and Durable

Apache Kafka – from Apache

- Apache Kafka is a distributed publish-subscribe messaging system. It is designed to support the following
 - Persistent messaging that provide constant time performance even with many TB of stored messages.
 -
 - High-throughput: even with very modest hardware Kafka can support hundreds of thousands of messages per second
 - .
 - Explicit support for partitioning messages over Kafka servers and distributing consumption over a cluster of consumer machines while maintaining per-partition ordering semantics.
 -
 - Support for parallel data load into Hadoop.

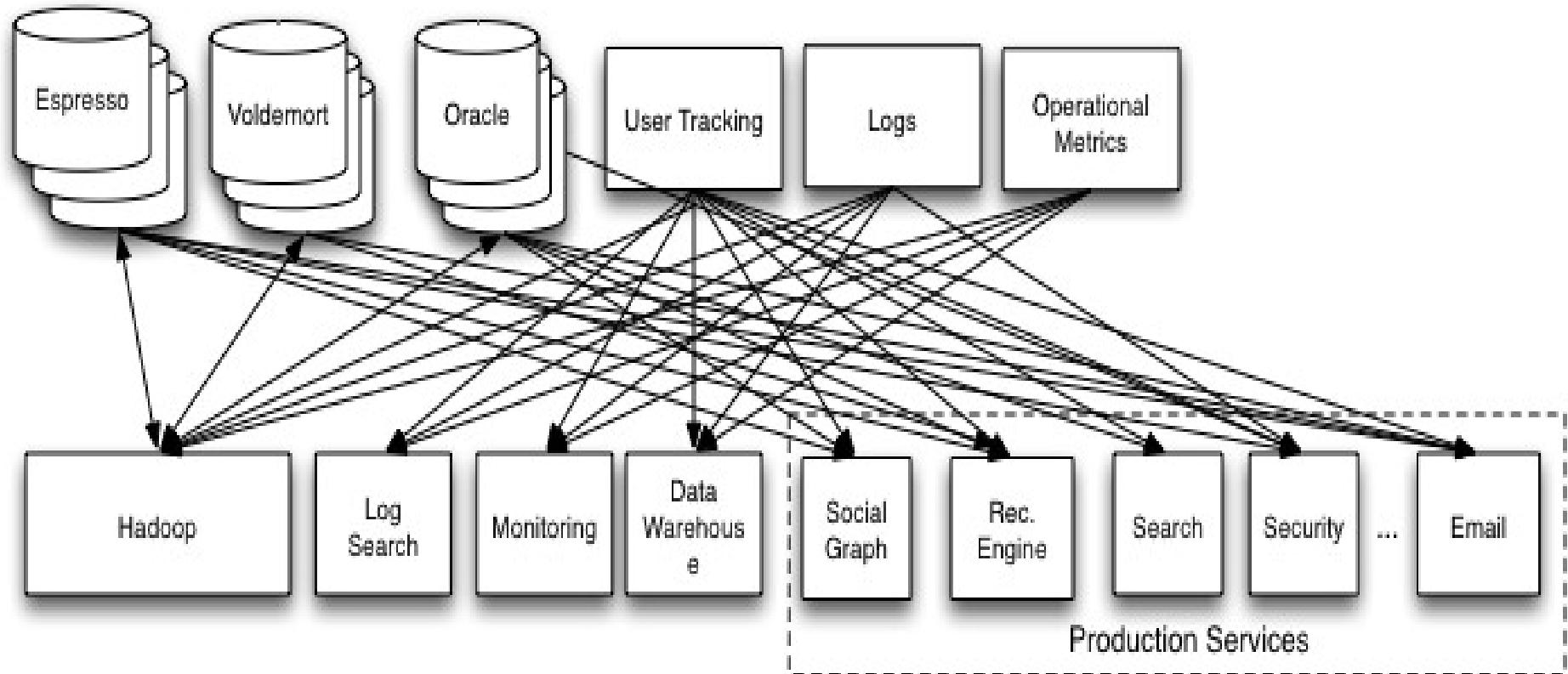
Background

- LinkedIn product donated to Apache
- Most core developers are from LinkedIn
- Pretty good pickup outside of LinkedIn: Netflix , Air BnB & Urban Airship for example
- Fun fact: no logo yet

Why?

Data Integration

Point to Point integration (thanks to LinkedIn for slide)

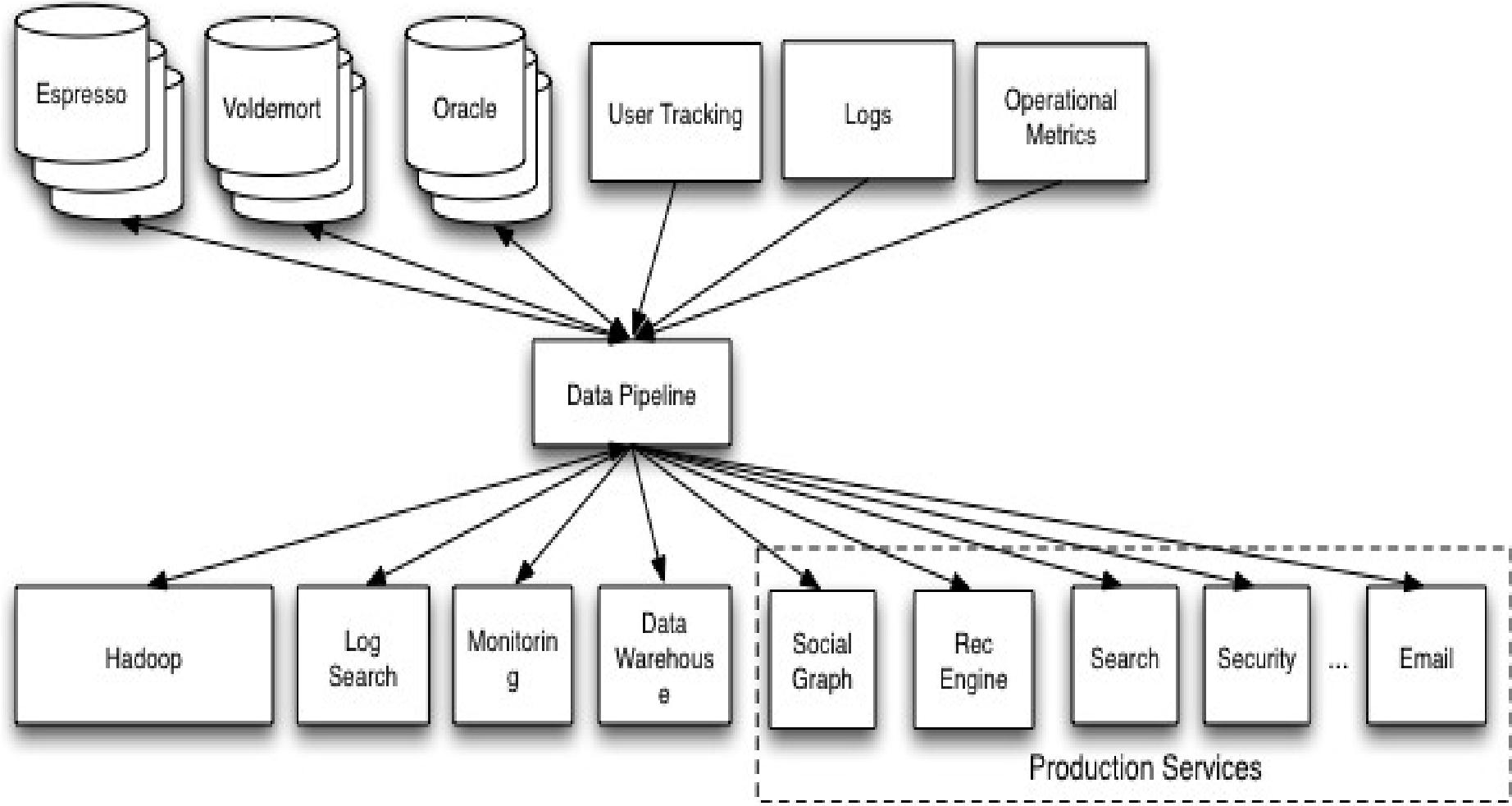




(Thanks to <http://linkstate.wordpress.com/2011/04/19/recabling-project/>)



What we'd really like (thanks to LinkedIn for slide)



Looks Familiar: JMS to the rescue!

Okay: Data warehouse to the rescue!

Okay: CICS to the rescue!

Kafka changes the paradigm

Kafka doesn't keep track of who consumed which message

Consumption Management

- Kafka leaves management of what was consumed up to the business logic
- Each message has a unique identifier (within the topic and partition)
- Consumers can ask for message by identifier, even if they are days old
- Identifiers are sequential within a topic and partition.

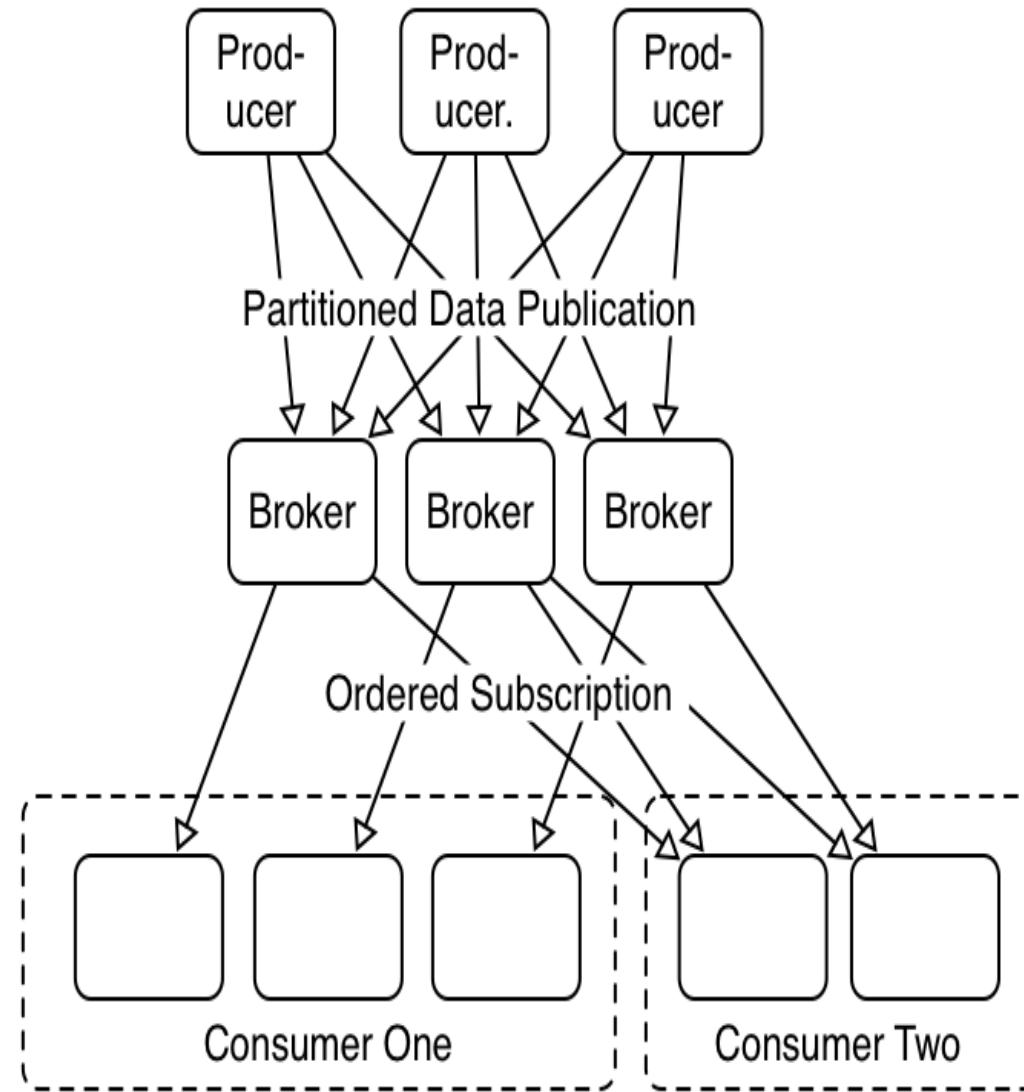
Why is Kafka Interesting?

Horizontally scalable messaging system

Terminology

- Topics are the main grouping mechanism for messages
-
- Brokers store the messages, take care of redundancy issues
-
- Producers write messages to a broker for a specific topic
-
- Consumers read from Brokers for a specific topic
-
- Topics can be further segmented by partitions
-
- Consumers can read a specific partition from a Topic

Architecture (thanks to LinkedIn for slide)



Producers

- Pretty Basic API
- Partitioning is a little odd, requires Producers to know about partition scheme
- Producers DO NOT know about consumers

Consumers: Consumer Groups

- Easiest to get started with
-
- Kafka makes sure only one thread in the group sees a message for a topic (or a message within a Partition)
-
- Uses Zookeeper to keep track of what messages were consumed in which topic/partitions
-
- No ‘once and only once’ delivery semantics here
-
- Rebalance may mean a message gets replayed

Consumers: Simple Consumer

- Consumer subscribes to a specific topic and partition
-
- Consumer has to keep track of what message offset was last consumed
-
- A lot more error handling required if Brokers have issues
-
- But a lot more control over which messages are read. Does allow for 'exactly once' messaging

Consumer Model Design

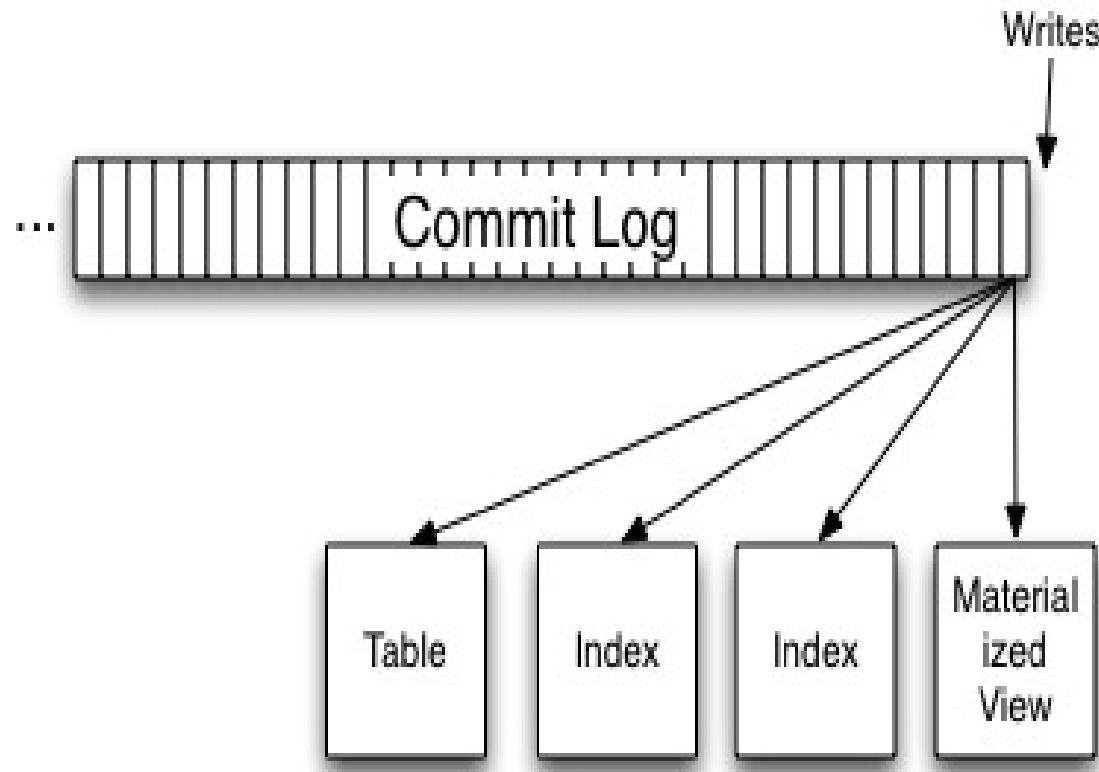
- Partition design impacts overall throughput
 - Producers know partitioning class
 - Producers write to single Broker ‘leader’ for a partition
- Offsets as only transaction identifier complicates consumer
 - ‘throw more hardware’ at the backlog is complicated
 - Consumer Groups == 1 thread per partition
 - If expensive operations can’t throw more threads at it
- Not a lot of ‘real world’ examples on balancing # of topics vs. # of partitions

Why is Kafka Interesting?

Memory Mapped Files

Kernel-space processing

What is a commit log? (thanks to LinkedIn for slide)



Brokers

- Lightweight, very fast message storing
-
- Writes messages to disk using kernel space NOT JVM
-
- Uses OS Pagecache
-
- Data is stored in flat files on disk, directory per topic and partition
-
- Handles the replication

Brokers continued

- Very low memory utilization - almost nothing is held in memory
-
- (Remember, Broker doesn't keep track of who has consumed a message)
-
- Handle TTL operations on data
-
- Drop a **file** when the data is too old

Why is Kafka Interesting?

Stuff just works

Producers and Consumers are about
business logic

Consumer Use Case: batch loading

- Consumers don't have to be online all the time
-
- Wake up every hour, ask Kafka for events since last request
-
- Load into a database, push to external systems etc.
-
- Load into Hadoop (Stream if using MapR)

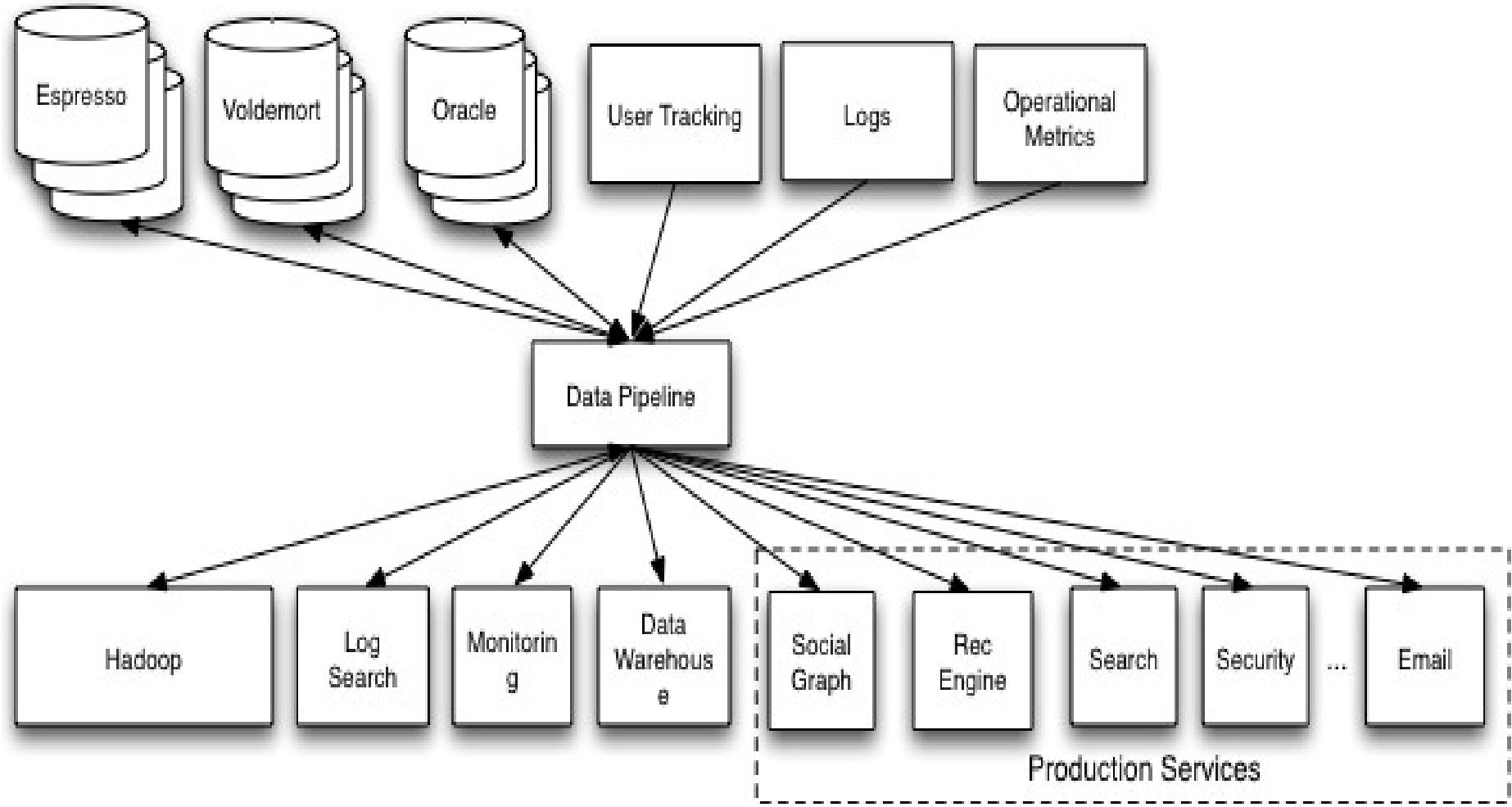
Adding Producers

- Define Topics and # of partitions via Kafka tools
-
- (possibly tell Kafka to balance leaders across machines)
-
- Start producing

Adding Consumers

- Using Kafka adding consumers doesn't impact producers
-
- Minor impact on Brokers (just keeping track of connections)

Finally Apache Kafka Works Everywhere and is best suited for PubSub Applications and Event Outsourcing



Micro Service Observability

Service Observability

- Fault Tolerance
- Health check
- Log aggregation
- Application metrics
- Audit logging
- Distributed tracing
- Log deployments and changes

Service Observability – Fault Tolerance

Annotations for Fault Tolerance

RETRY

@Retry // Retry indicates that this should trigger retry the method call several times in case the remote server call results in an exception

//@Retry(maxDuration = 5000, maxRetries = 3, delay = 500, jitter = 200)

@Fallback(fallbackMethod = "getCachedAuthor") // Fallback indicates that we should fall back to the local cache if the method fails even after several retries the circuit is open

@Timeout(value = 4, unit = ChronoUnit.SECONDS)

Service Observability – Fault Tolerance

Add a Circuit Breaker around a method invocation to fail fast

@CircuitBreaker // CircuitBreaker wraps the call in a circuit breaker which opens after several failures and closes again after some time

Once an external system you call is down or returning 503 as it is currently unavailable to process further requests, you might not want to access it for a given timeframe again. This might help the other system to recover and your methods can fail fast as you already know the expected response from requests in the past. For this scenario, the Circuit Breaker pattern comes into place.

The **Circuit Breaker** offers a way to fail fast by directly failing the method execution to prevent further overloading of the target system and indefinite wait or timeouts. With MicroProfile Fault Tolerance we have an annotation to achieve this with ease:
@CircuitBreaker

Service Observability – Fault Tolerance

States of circuit breaker

There are three different states a Circuit Breaker can have:

- closed,
- Opened,
- half-open.

In the **closed state**, the operation is executed as expected. If a failure occurs while e.g. calling an external service, the Circuit Breaker records such an event. If a particular threshold of failures is met, it will switch to the **open state**.

Once the Circuit Breaker enters the **open state**, further calls will fail immediately. After a given delay the circuit enters the half-open state.

Within the **half-open** state, trial executions will happen. Once such a trial execution fails, the circuit transitions to the open state again. When a predefined number of these trial executions succeed, the circuit enters the original closed state.

Service Observability – Fault Tolerance

This annotation also allows defining the exception types to skip and to fail on:

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 5, delay = 500,  
    skipOn = {NumberFormatException.class},  
    failOn = {RuntimeException.class})
```

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 5, delay = 500,  
    skipOn = {NumberFormatException.class},  
    failOn = {RuntimeException.class})
```

Service Observability – Fault Tolerance

Let's have a look at the following example:

```
@CircuitBreaker(successThreshold = 10, requestVolumeThreshold = 5, failureRatio  
= 0.5, delay = 500)  
@Fallback(fallbackMethod = "getFallbackData")  
public String getRandomData() {  
    if (ThreadLocalRandom.current().nextLong(1000) < 300) {  
        return "random duke";  
    } else {  
        throw new RuntimeException("Random data not available");  
    }  
}
```

In the example above I define a Circuit Breaker which enters the open state once 50% (failureRatio=0.5) of five consecutive executions (requestVolumeThreshold=5) fail. After a delay of 500 milliseconds in the open state, the circuit transitions to half-open. Once ten trial executions (successThreshold=10) in the half-open state succeed, the circuit will be back in the closed state.

Service Observability – Asynchronous Execution

Execute a method asynchronously with MicroProfile Fault Tolerance

Some use cases of your system might not require synchronous and in-order execution of different tasks. For instance, you can fetch data for a customer (purchased orders, contact information, invoices) from different services in parallel. The MicroProfile Fault Tolerance specification offers a convenient way for achieving such asynchronous method executions: `@Asynchronous`:

@Asynchronous

```
public Future<String> getConcurrentServiceData(String name) {  
    System.out.println(name + " is accessing the concurrent service");  
    return CompletableFuture.completedFuture("concurrent duke");  
}
```

Service Observability – Bulk Heads

Apply Bulkheads to limit the number of concurrent calls

The Bulkhead pattern is a way of isolating failures in your system while the rest can still function. It's named after the sectioned parts (bulkheads) of a ship. If one bulkhead of a ship is damaged and filled with water, the other bulkheads aren't affected, which prevents the ship from sinking.

Imagine a scenario where all your threads are occupied for a request to a (slow-responding) external system and your application can't process other tasks. To prevent such a scenario, we can apply the @Bulkhead annotation and limit concurrent calls:

Service Observability – Bulk Heads

```
@Bulkhead(5)
@Asynchronous
public Future<String> getConcurrentServiceData(String name) throws
InterruptedException {
    Thread.sleep(1000);
    System.out.println(name + " is accessing the concurrent service");
    return CompletableFuture.completedFuture("concurrent duke");
}
```

In this example, only five concurrent calls can enter this method and further have to wait. If this annotation is used together with @Asynchronous, as in the example above, it means thread isolation. In addition and only for asynchronous methods we can specify the length of the waiting queue with the attribute waitingTasksQueue. For non-async methods, the specification defines to utilize semaphores for isolation.

Service Observability – Bulk Heads

Configurability of fault Tolerance – write in config file

The pattern for external configuration is the following:

<classname>/<methodname>/<annotation>/<parameter>:

de.rieckpil.blog.RandomDataProvider/accessFlakyService/Retry/maxRetries=10

de.rieckpil.blog.RandomDataProvider/accessFlakyService/Retry/delay=300

Service Observability - Health Check

Health Check

// You need to write this class every project

@Health

@ApplicationScoped

public class HealthCheckService implements HealthCheck {

 @Override

 public HealthCheckResponse call() {

 return HealthCheckResponse.named("health")

 .up()

 .withData("Author", "Kamlendu Pandey")

 .withData("Website", "https://localhost")

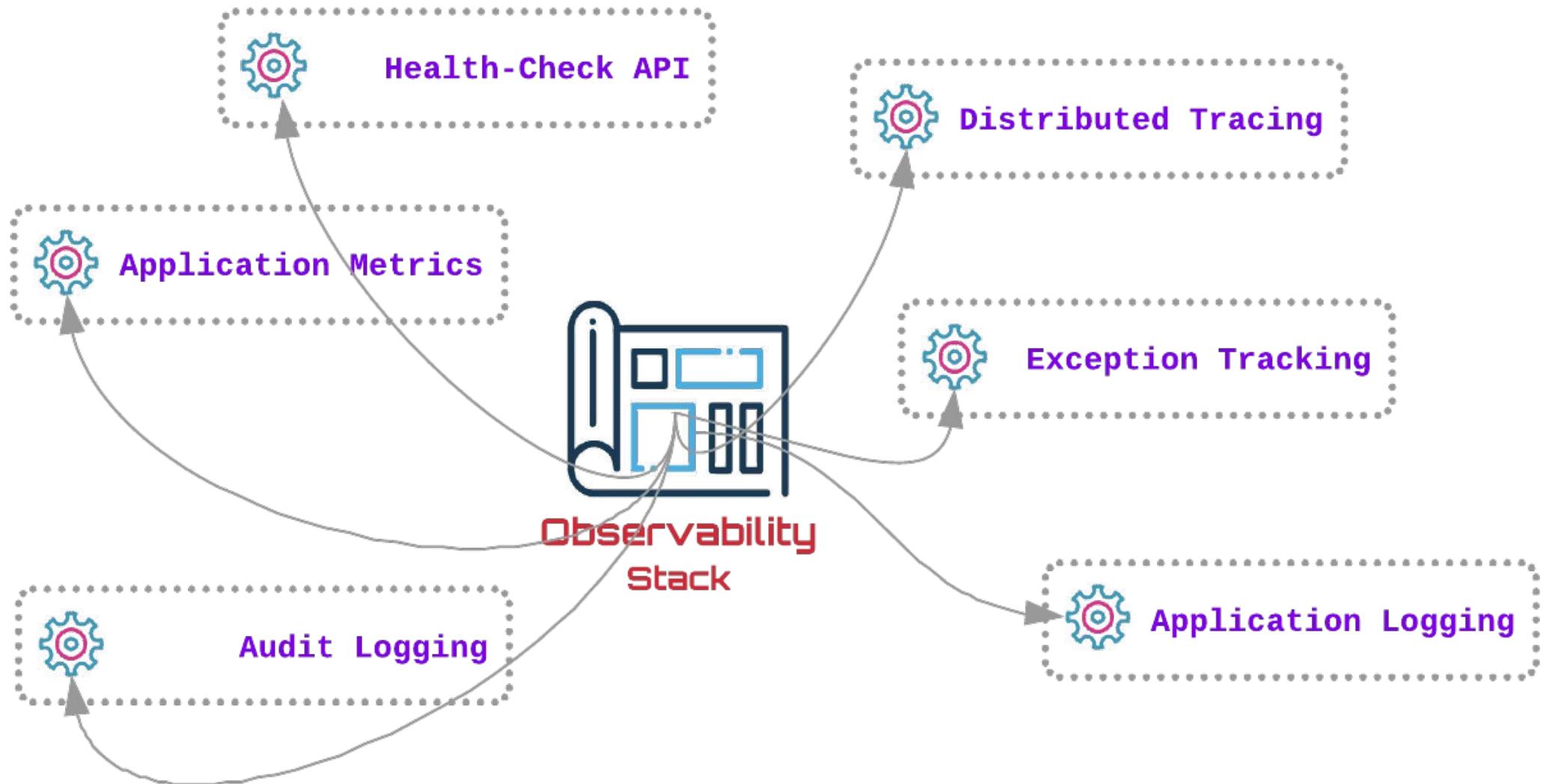
 .build();

 }

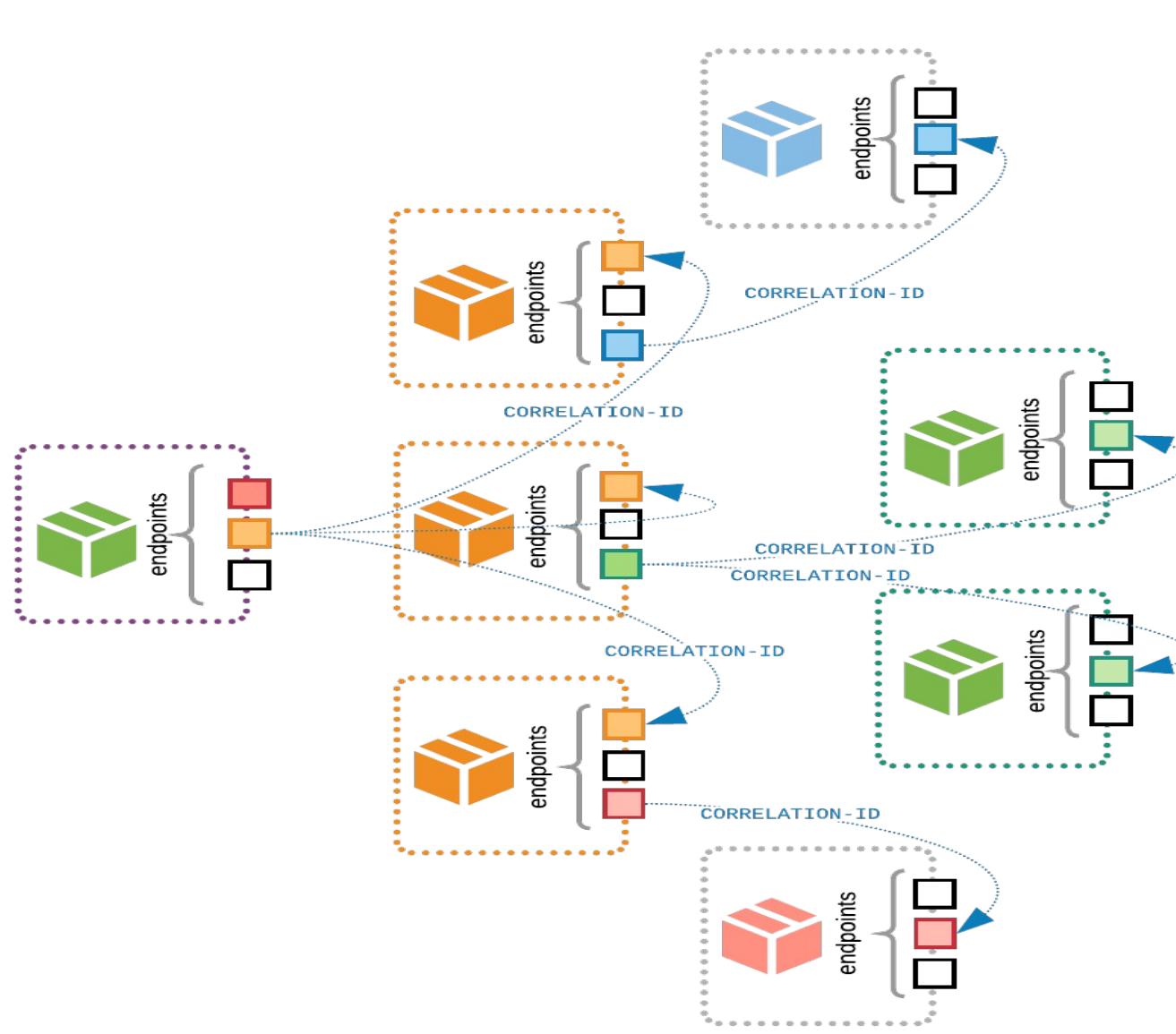
}

<http://localhost:8080/health> will give you the status of deployed services on payara

Service Observability - Observability Stack



Service Observability - Observability Tools



Prometheus



JAEGER



Grafana 22.12.43

OBSERVABILITY

22.12.43

Service Observability - Tool Chain Prometheus

OBSERVABILITY



Prometheus is an open-source tool for instrumenting and monitoring metrics. It works in a pull-based manner, makes HTTP requests to your metric endpoint with the pre-determined time intervals (default 10 seconds), and store these metrics in its own time-series database. Prometheus also provides a GUI to make queries over these metrics with its own query language PromQL. It provides basic graphics to visualize metrics. Prometheus also has an alert plugin to produce alerts according to metrics values.

Service Observability – Tool Chain - Jagger

OBSERVABILITY



High Scalability – Jaeger backend is designed to have no single points of failure and to scale with the business needs.

Native support for OpenTracing – Jaeger backend, Web UI, and instrumentation libraries have been designed from the ground up to support the OpenTracing standard.

Storage Backend – Jaeger supports two popular open-source NoSQL databases as trace storage backends: Cassandra 3.4+ and Elasticsearch 5.x/6.x.

Modern UI – Jaeger Web UI is implemented in Javascript using popular open-source frameworks like React.

Cloud-Native Deployment – Jaeger backend is distributed as a collection of Docker images. - Deployment to Kubernetes clusters is assisted by Kubernetes templates and a Helm chart.

All Jaeger backend components expose Prometheus metrics by default

Service Observability - Tool Chain - Grafana

OBSErvABILITY



Prometheus

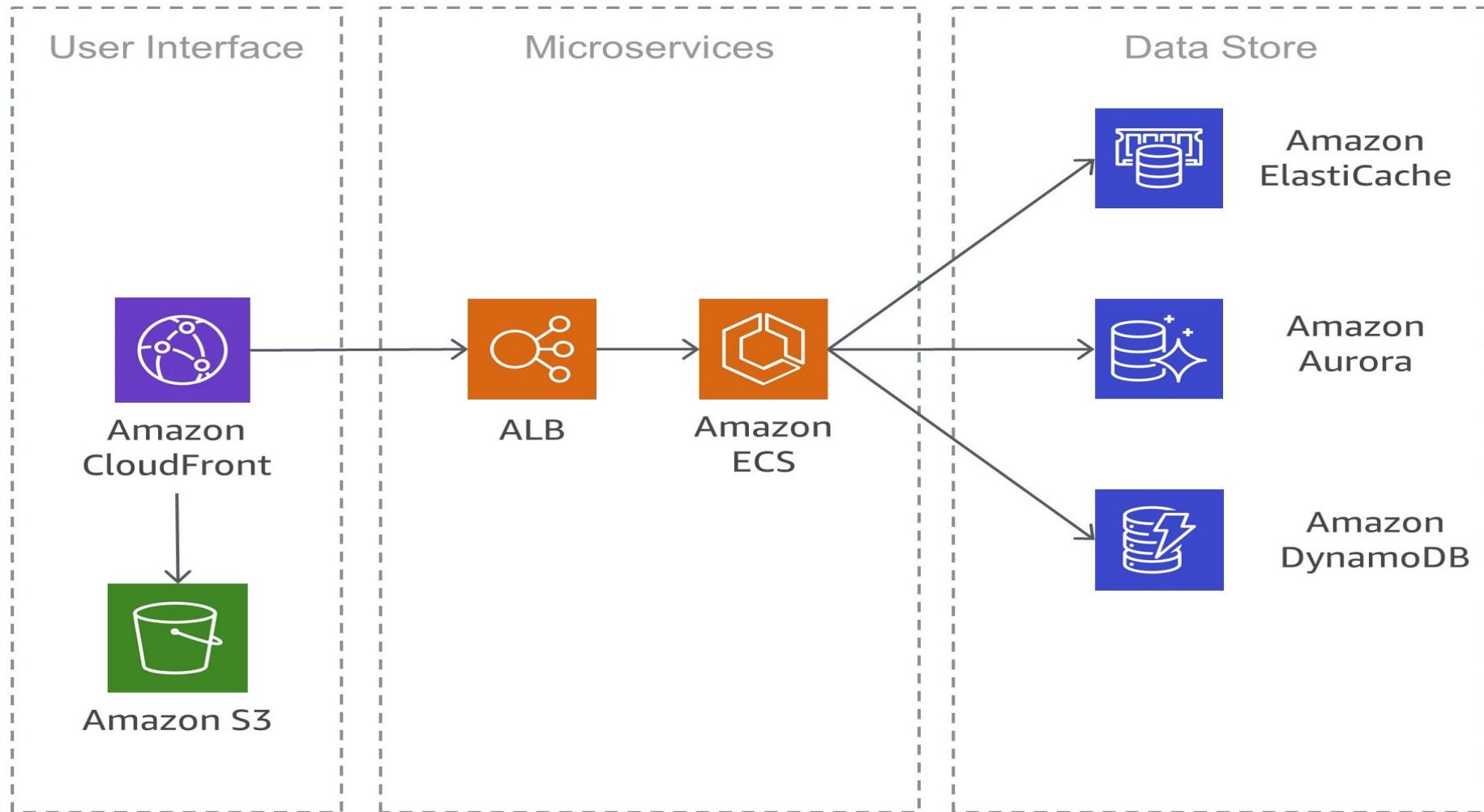


JAEGER

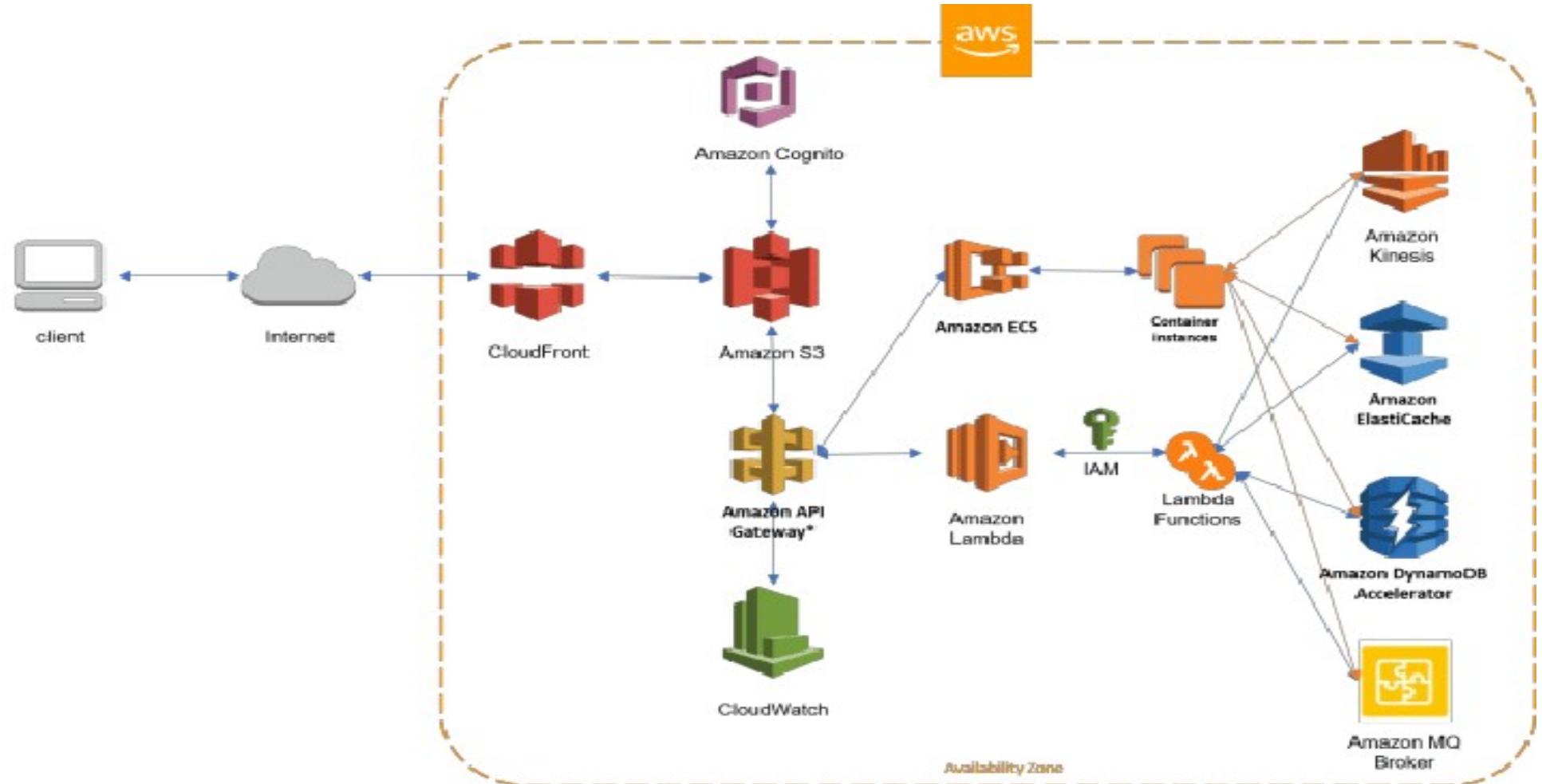


Grafana is an open-source metric analytics and visualization suite. It is most commonly used for visualizing time series data for infrastructure and application analytics but many use it in other domains including industrial sensors, home automation, weather, and process control.

Amazon Cloud and Micro Services



Amazon Cloud and Micro Services (More Specific)



Cloud Front : Cloud Access ,
Amazon ECS : containers
Amazon EKS : Managed Kubernetes

Amazon S3 : Data Storage
IAM : Authentication/Authorization
ALB : Load Balancer

Lets go for Micro Services a Serverless Architecture



THANKS !