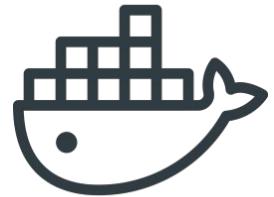


# Microservices: From Theory to Practice



Guest Seminar Lecture  
University of L'Aquila

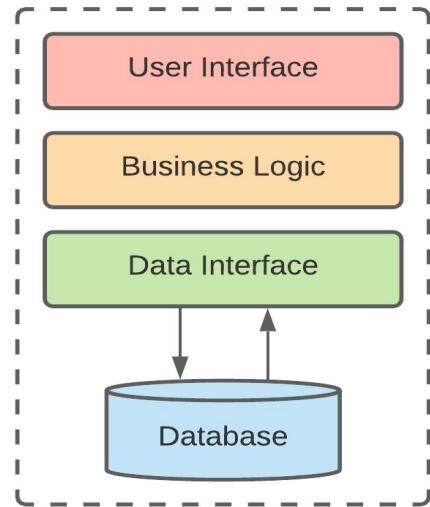
January 2021

Presentation By:  
Karthik Vaidhyanathan  
Gran Sasso Science Institute

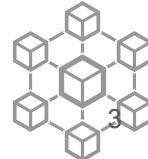
**To the world of Monoliths**

# Brief History - The World of Monoliths (90's)

- Applications built as a **single unit**
- Traditional 2-tier, 3-tier applications - Separation of client, business logic and database
- Packaged as one bundle - eg: Entire application bundled into one WAR

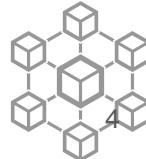


Standard monolith application architecture



# The World of Monoliths - An Example System

- Develop an E-commerce application with following features:
  - User registration and authentication
  - Browse catalog
  - Place orders
  - Add/modify delivery information
  - Add/modify billing information
  - Make payment
- How were such systems developed ?

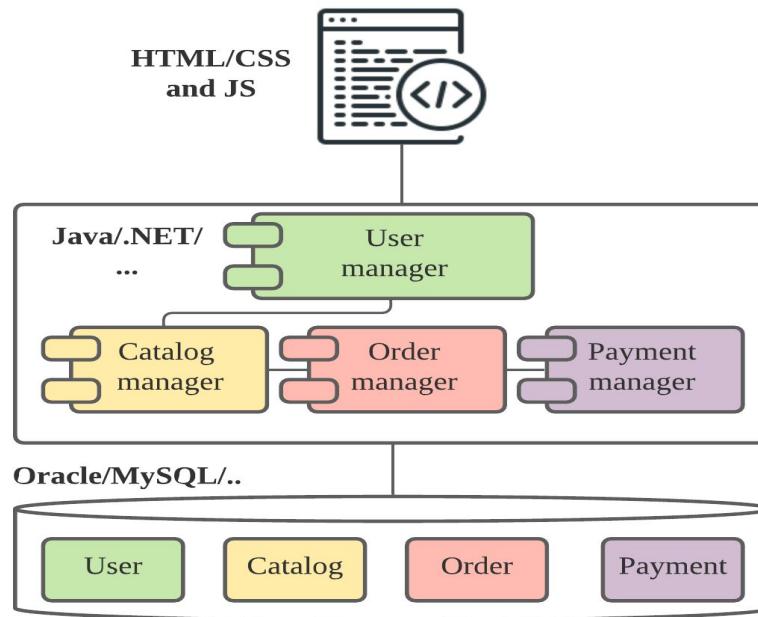


# The World of Monoliths - Monolithic Approach

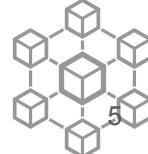
## Organizational



## Architecture



## Deployment



# The World of Monoliths - Monolithic Approach

- High degree of coupling - everyone needs to know everything !!!
- Change cycle and bug fix can take weeks - Modifiability and time to market
- Adding new feature can be challenging - Extensibility
- Separation of concerns via components with inherent coupling - Modularity
- Scaling system implies scaling the whole stack - Scalability
- Limited by the language of choice - eg: add recommendation feature to e-commerce (Java or Python ?)
- Database is centralized - addition or modification is a costly process



# Moving into SOA

# The Age of Service Oriented Architectures (SOA)

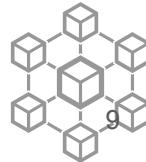
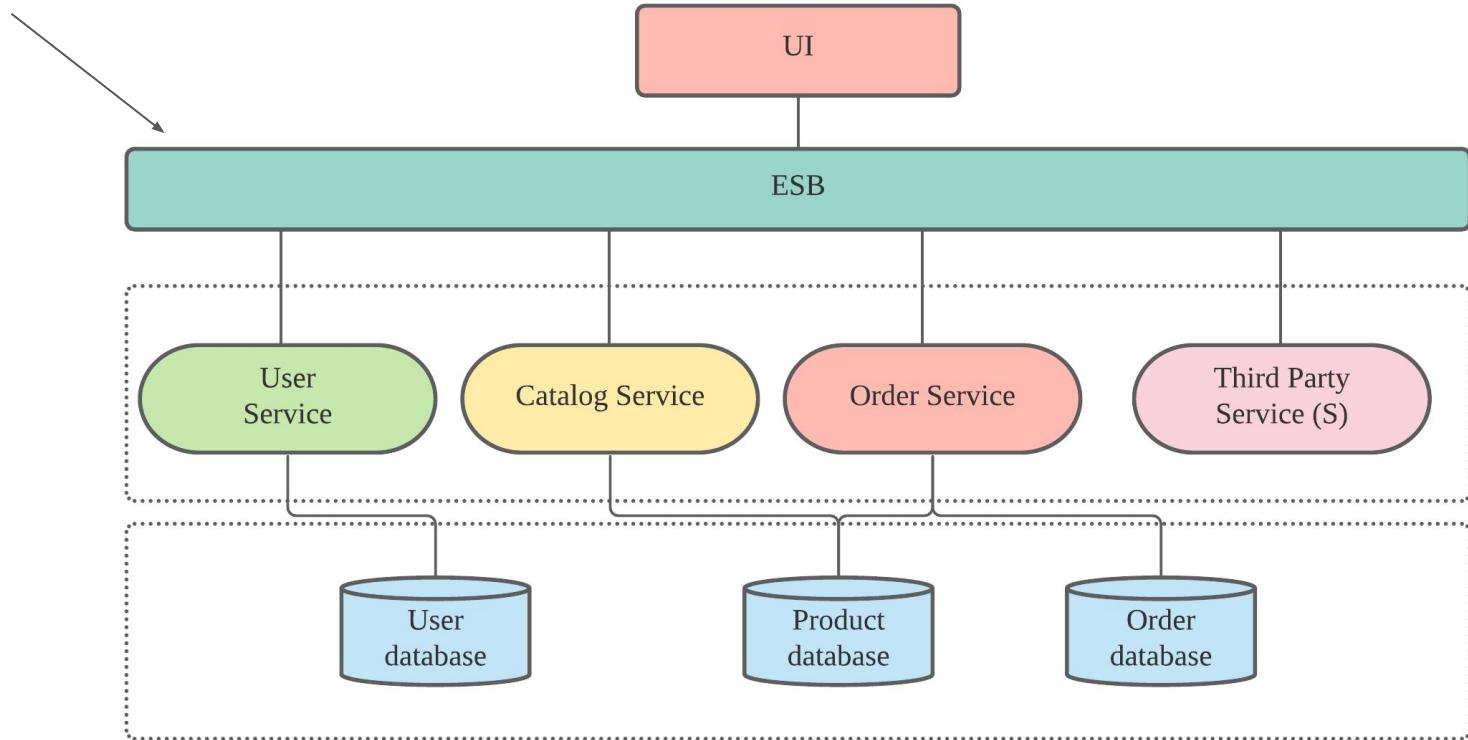
*“SOA means too many different things to different people” -- Martin Fowler*

- Very popular style for architecting systems from early 2000's
- Make software components reusable via service interfaces, accessible over network through standard communication protocols
- Services are loosely coupled
- Developers just focus on building services. Integration through **Enterprise Service Bus (ESB)**
- Services are more **coarse grained** - Small services to large applications (No clearly defined boundaries)
- It is more at an **enterprise level** - Eg: integrate multiple applications via ESB



# SOA Approach to E-Commerce Application

**Overhead !!**



# Time to Evolve: Make things more fine grained

Domain Driven Design

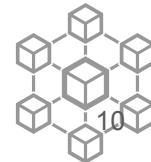
Large Scale Systems

Continuous Delivery

Infrastructure Automation

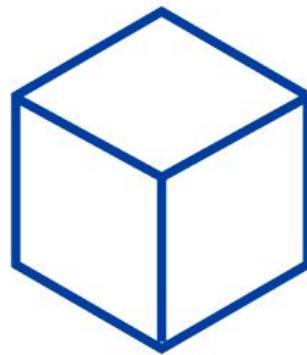
Small Autonomous Teams

**Microservices**

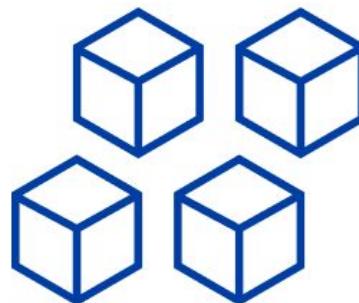


# Into the world of Microservices

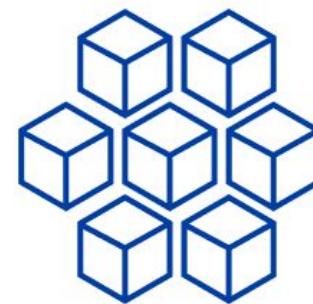
# Moving Towards Microservices



**MONOLITHIC**  
Single unit



**SOA**  
Coarse-grained

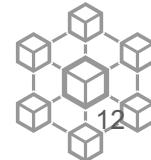


**MICROSERVICES**  
Fine-grained

1990

2000

2010



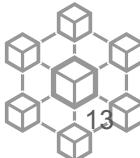
# Microservices: What does it mean ?

*“Small autonomous services that work together”*

-- Sam Newman

*“It is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API”*

-- Martin Fowler

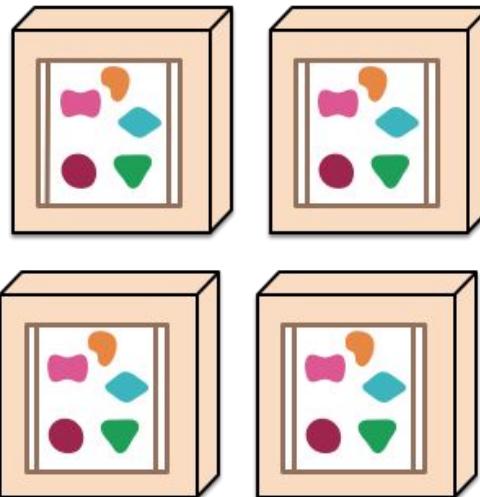


# Microservices: What does it mean ?

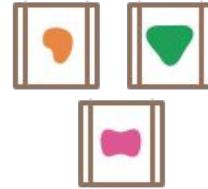
*A monolithic application puts all its functionality into a single process...*



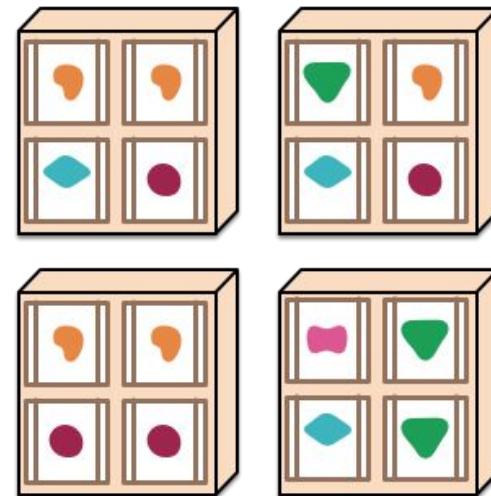
*... and scales by replicating the monolith on multiple servers*



*A microservices architecture puts each element of functionality into a separate service...*

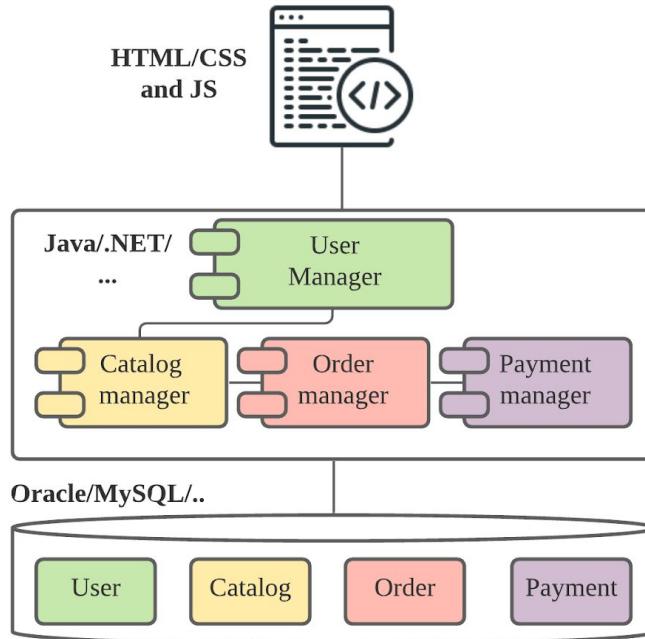


*... and scales by distributing these services across servers, replicating as needed.*

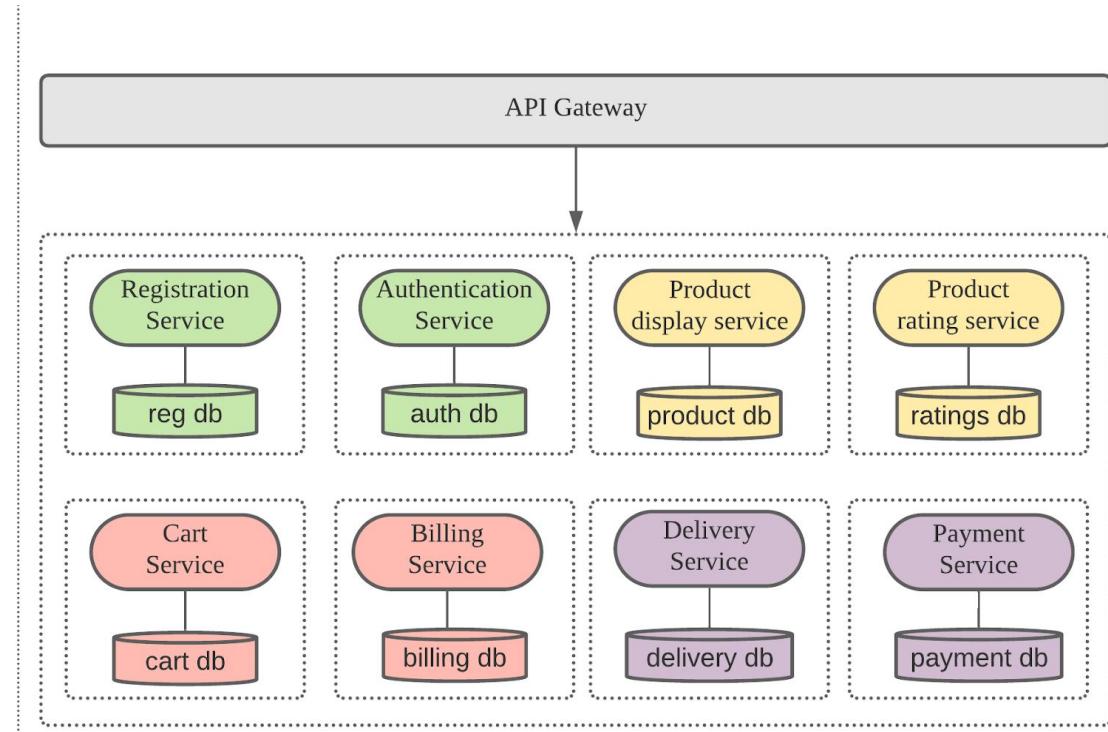


# Microservices: What does it mean ?

## Monolith version

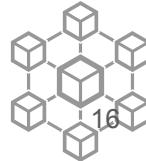
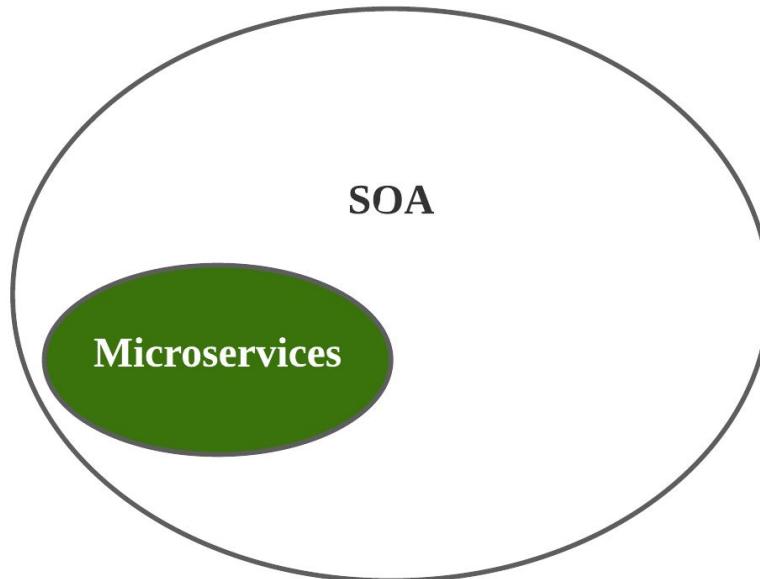


## Microservices Version



# Microservices: Is it not SOA ?

- It is more SOA done in a more clear manner - Difference in scope
- Fine grained as opposed to Coarse grained - Small functional pieces



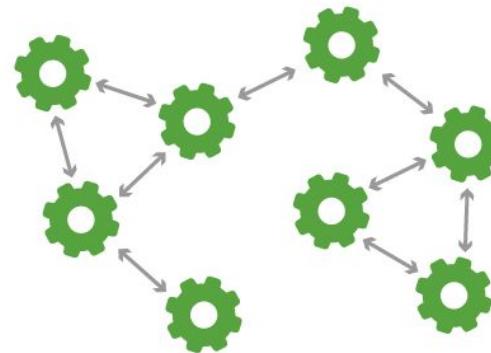
# Microservices: Is it not SOA ?

## 2000's SERVICE ORIENTED ARCHITECTURE

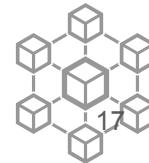


SOA based applications are comprised of more loosely coupled components that use an Enterprise Services Bus messaging protocol to communicate between themselves.

## 2010's MICROSERVICES ARCHITECTURE

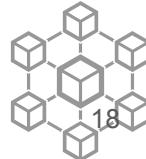


**Microservices** are a number of independent application services delivering one single functionality in a loosely connected and self-contained fashion, communicating through light-weight messaging protocols such as HTTP, REST or Thrift API.



# Microservices: Key Characteristics

- Compensation via Services
- Organized around business capabilities
- Highly maintainable and testable
- Decentralized governance and decentralized data management
- Widespread industry adoption in the last decade: Netflix, Uber, Google, etc



# Microservices: Advantages

## Scaling is Easy

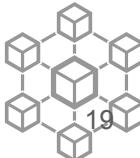
- Scale only the required microservices
- Adding a new feature can be just adding one another microservice

## Heterogeneity

- Each microservice can be developed in different technologies
- Experimenting with new technology is easy

## Resilience

- Only specific microservices goes down
- Grouping microservices as critical and non-critical can be done to add more resilience



# Microservices: Advantages

## Organizational Alignment

- Easily distribute teams around microservices - eg: Amazon 2 pizza rule
- Minimize people working on one less codebase

## Composability

- Easily compose microservices to get new functionality

## Replaceability

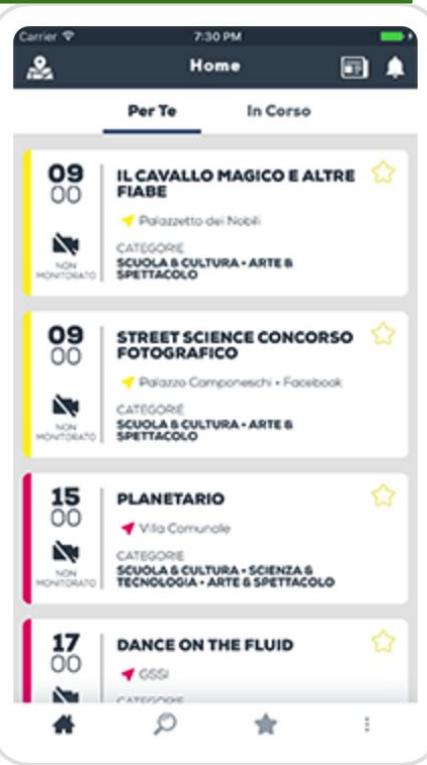
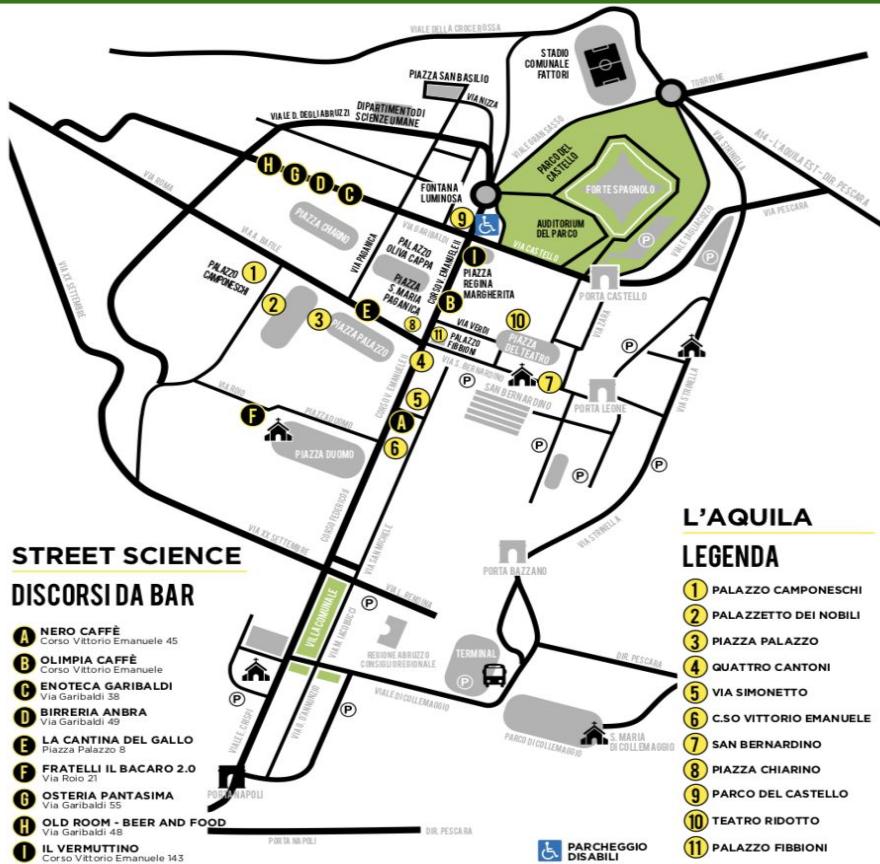
- Cost of replacement is small - should not take more than 2 weeks
- Imagine replacing a 25 year old legacy system !!

## Ease of Deployment

- Check and rollback easily
- Continuous integration and deployment is easier - DevOps!!!



# NdR: A Case Study



<https://www.streetscience.it>

# NdR: A Case Study

**Goal:** Develop a microservice based IoT application for NdR system

**Features:** User registration, book venues, book parking lots, provide venue and parking lot recommendation, priority booking based on small payment, check weather

## **Data Sources:**

- Parking mats at entrances and exits of parking lot to get count of cars
- Handheld RFID reader to capture the count of people entering venue
- People counter at venue exits to count people exiting venue



# Designing Microservices

# Microservices: How to Design ?

## Ensure loose coupling

- Minimize coupling between microservices
- Should be easy to change and deploy one without affecting others
- Each microservice needs to know as little as possible about others

## Maintain high cohesion

- Bundle one end to end feature or complete part of it inside one microservice
- Promotes robustness and reliability
- One change should **never require** change in 10 different places

## Follow the principle of bounded contexts

- Identify different contexts inside the main domain
- Only share what is important rest remains within context



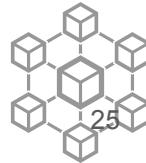
# Contexts within NdR System

IoT

Booking

Weather

Financial

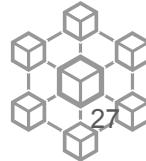
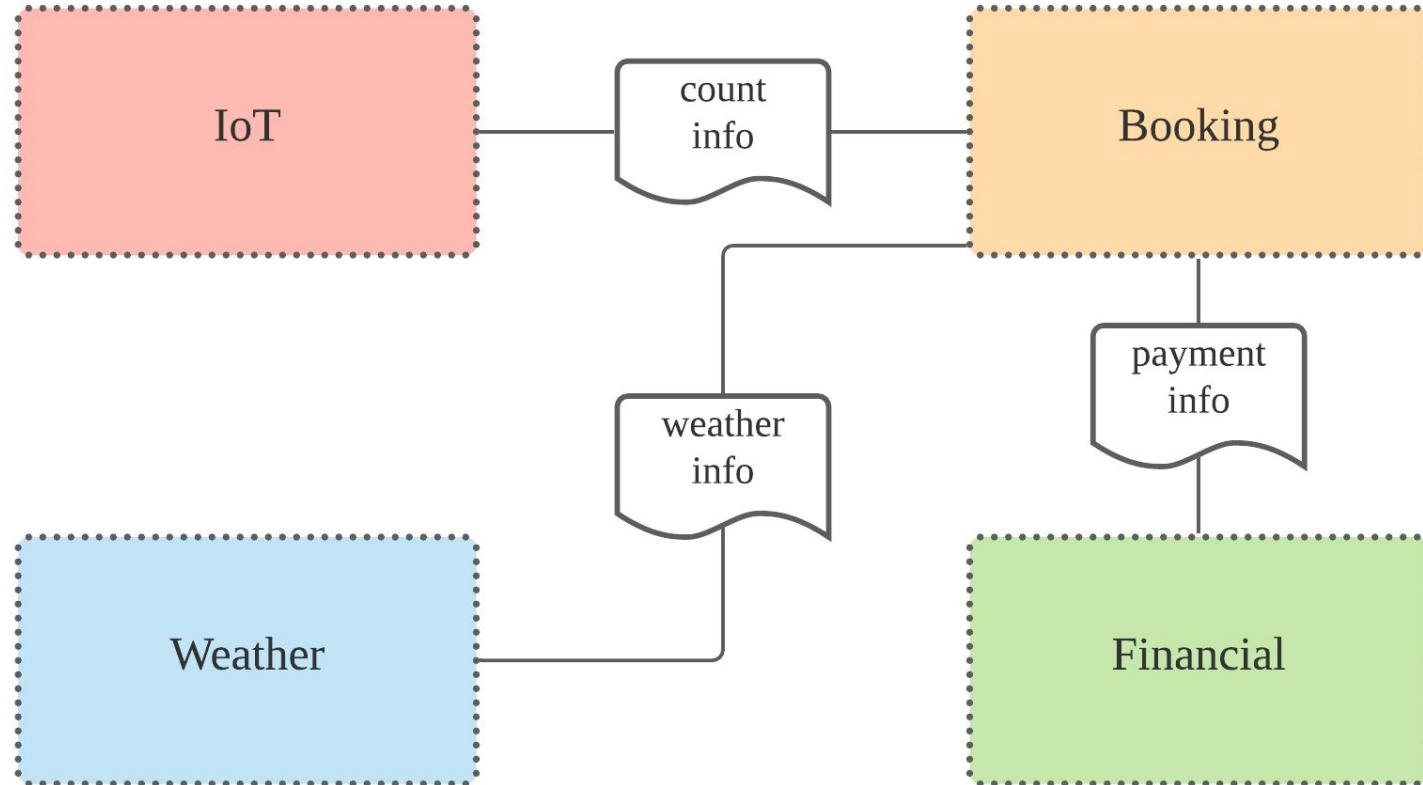


# Bounded Context: Shared and Hidden Models

- Identify what needs to be shared
  - Eg: Sharing of information on people and car count to booking context
- Same things may have different meaning in different contexts
  - Eg: Sensor data in IoT context and booking context
- This process will facilitate avoiding of high coupling (**Pitfall !!**)
- Microservices should never be chatty !
  - Adds to performance issues
  - Lack of cohesion
  - Eg: too many back and forth communication between two microservices



# Shared Models in NdR System

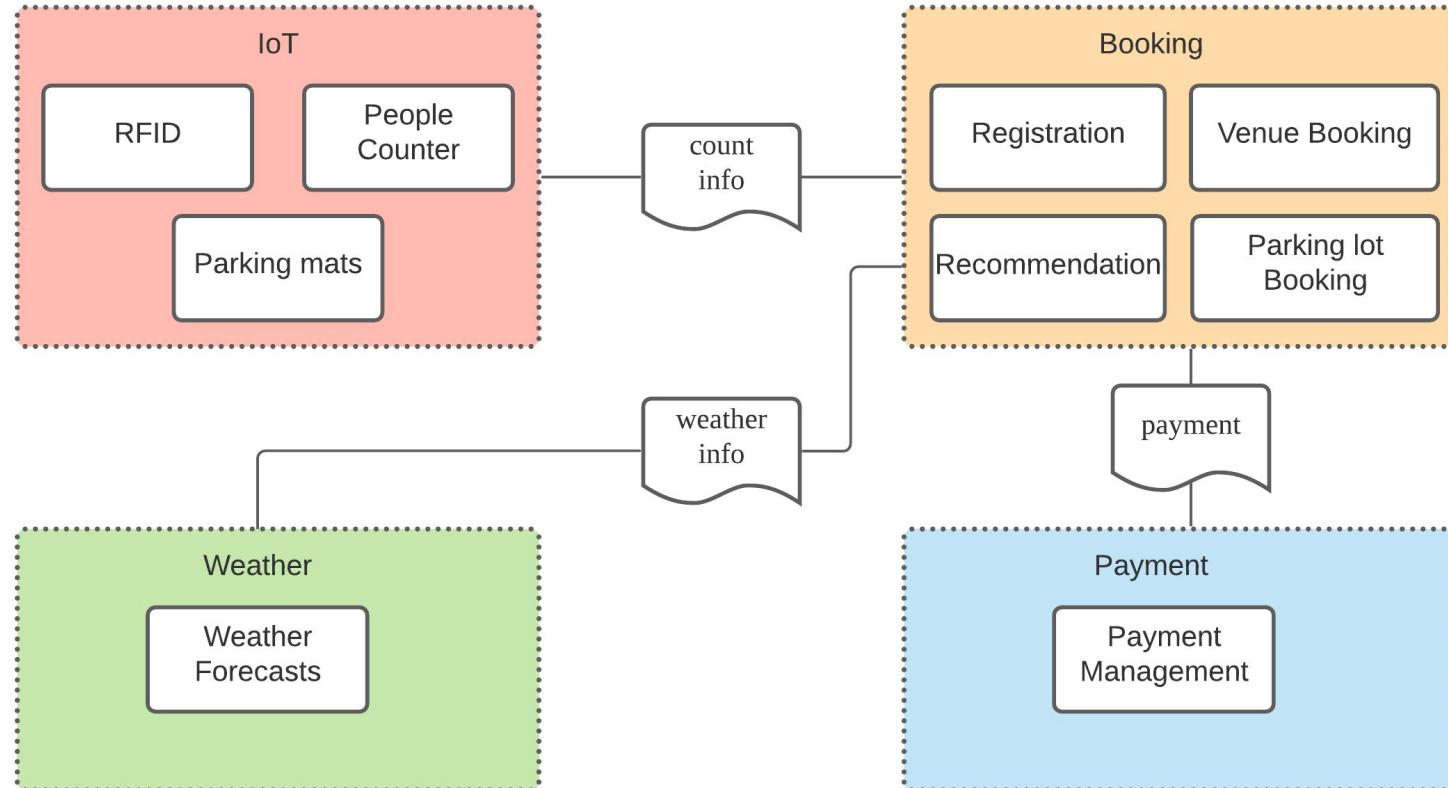


# Bounded Context: Modules and Services

- Separate the contexts into modules
  - Eg: Venue booking and Parking lot booking inside booking
- Use the help of hidden and shared models
  - Shared becomes the bridge and hidden becomes the separation points
- The modules becomes candidates for microservices
  - High Cohesion - Everything stays within context and modules are independent
  - Loose Coupling - Only what is needed is shared
- **Avoid** premature decomposition
  - Early decisions can be costly (eg: entire IoT inside one microservice)
  - Re-decomposition may take time, effort and expenditure



# Modules and Services in NdR System



# How to Integrate Microservices

# Microservices: Integration

*If done well, microservices works really well, if not then a disaster awaits*

## Avoid breaking changes

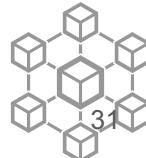
- Choose techniques and technologies wisely
- Eg: change in data sent from microservices should not cause change in all consumers

## API's should be technology agnostic

- API's that integrates microservices should be technology agnostic
- If API is tied to technology, changing microservice implementation implies changing API usage and vice versa. Eg: Java and RPC

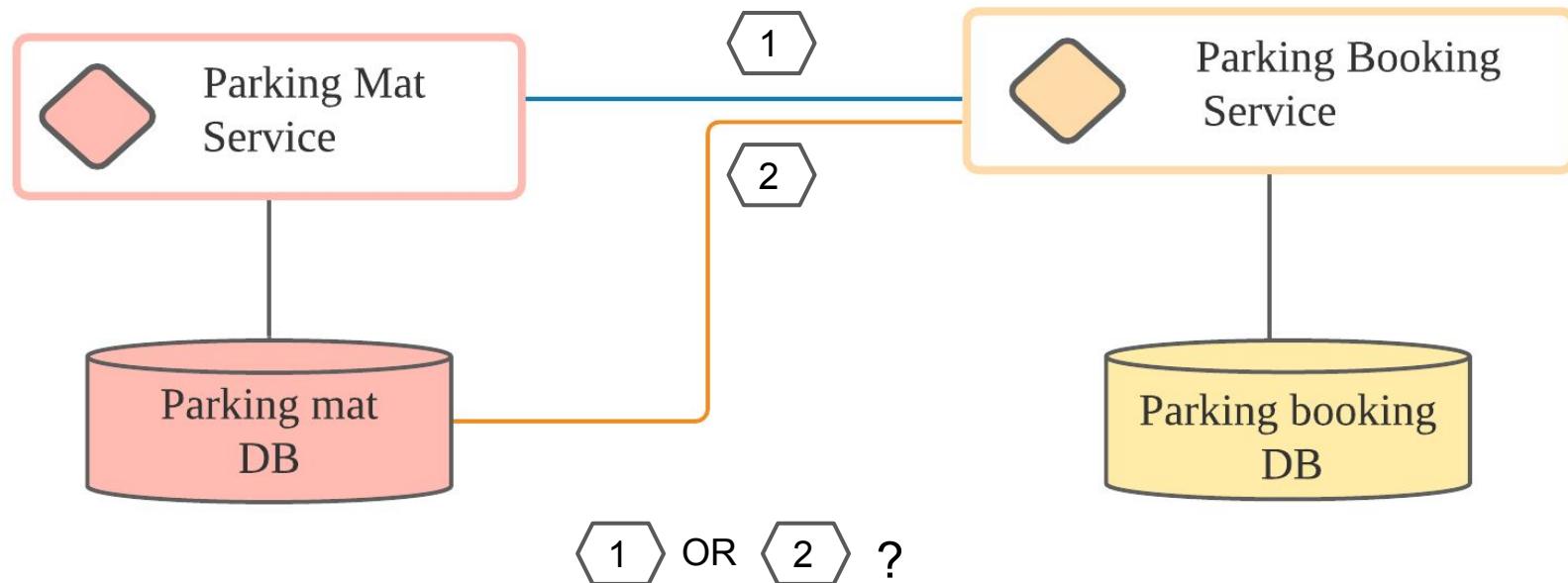
## Make services simple for consumers

- Consumer should not worry about libraries to access API's
- If not well done, increases coupling !!



# Integration with Shared DB ?

**Scenario:** When a user wants to book a parking lot, a request is sent to parking booking service which checks the status of availability from the parking mat data



# Integration with Shared DB ?

Avoid integration with shared db as much as possible:

- **Changing DB schema** based on one microservice need affects others
- **Affects evolution** of system eg: changing from relational to non-relational
- Choice of DB might **constrain the choice of language** for implementing microservice eg: Java might have more db driver available for MySQL
- **Goodbye** high cohesion and loose coupling !!!



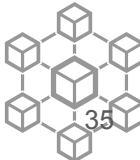
# Synchronous v/s Asynchronous Communication

- **Synchronous:** request -> response based communication
  - Service A sends a request to Service B and waits for response
  - Request that triggers long running jobs can affect performance
- **Asynchronous:** request -> response or Event-based communication
  - Service A sends a request to Service B and does not wait for response (gets when available)
  - Suited very well for long running jobs
  - Implementation can be tricky also based on language of choice
  - Methods to implement: registering callbacks or event queues (pub sub)
  - eg: many modern applications use Event-driven Microservice Architectures
- Compare choices with complexity of system and use case before deciding



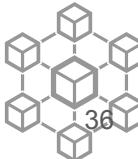
# Orchestration v/s Choreography

- **Orchestration:** One central coordinator manages the flow
  - Too much governance: **Coupling and chatty**
  - Eg: A central service that first calls booking then based on response calls payment
- **Choreography:** Everyone knows what to do
  - More decoupled approach: every service subscribes to events
  - More asynchronous in nature
  - **Sophisticated mechanisms** to ensure that flow is executed properly
- **General advice:** Prefer choreography over orchestration
  - More flexible and loosely coupled in nature
  - Cost of change is less



# Protocol and Data Exchange Format

- **RPC or REST or simple HTTP:** What protocol to use ?
  - Avoid technology coupling with protocol eg: Java RMI (RPC)
  - Consider the network: which protocol is more network friendly
  - REST is more preferred: More decoupling and easy to understand and program
- **XML or JSON:** What about data exchange format ?
  - JSON has more become a defacto standard ever since its introduction
  - JSON is more lightweight compared to XML hence more lighter on network
  - Structure is more in XML. Teams need to standardize JSON exchange (Exemplars or templates)
  - Using XML may create a technology coupling. For eg: Java has better ways to handle XML



# Patterns for Integration

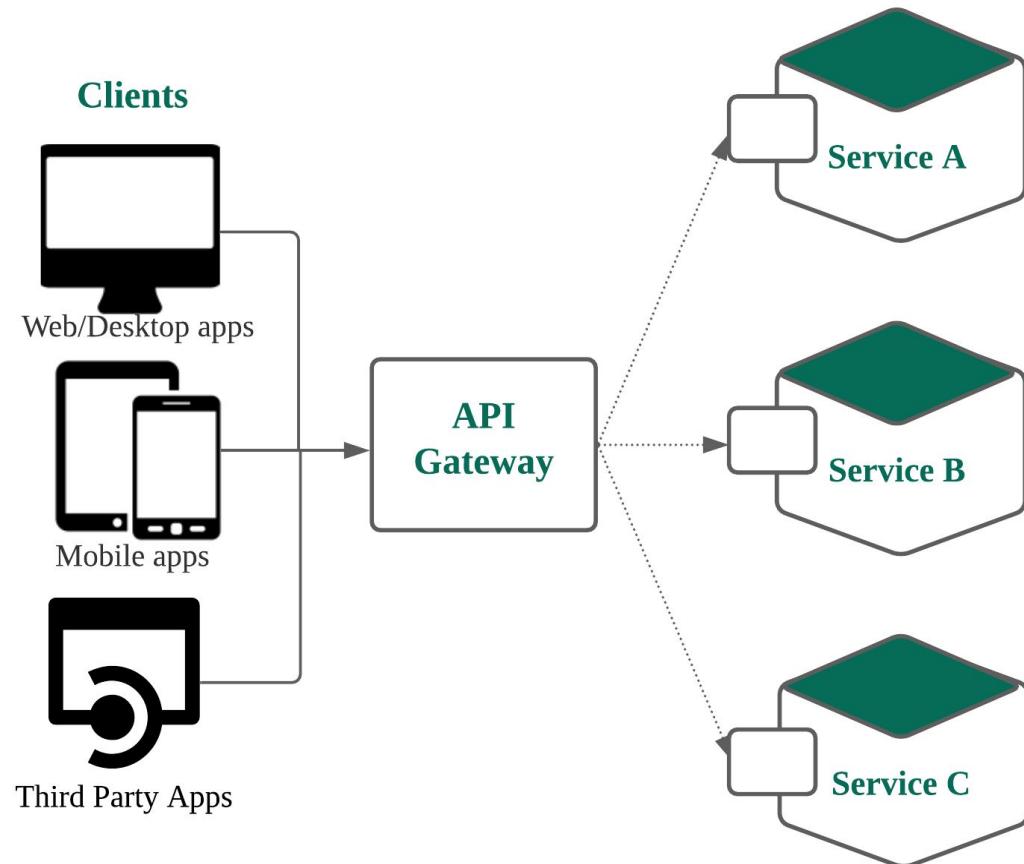
# Integration Patterns: API Gateway Pattern

## Scenario:

- Different type of request/response formats and protocols
- Offloading authentication from microservices

## Uses:

- Single point entry for all microservice calls
- Efficiently handling request routing (**Gateway routing pattern**)
- Handle authentication effectively
- Can perform aggregation
- Can also provide a proxy for different types of client (**Proxy pattern**)



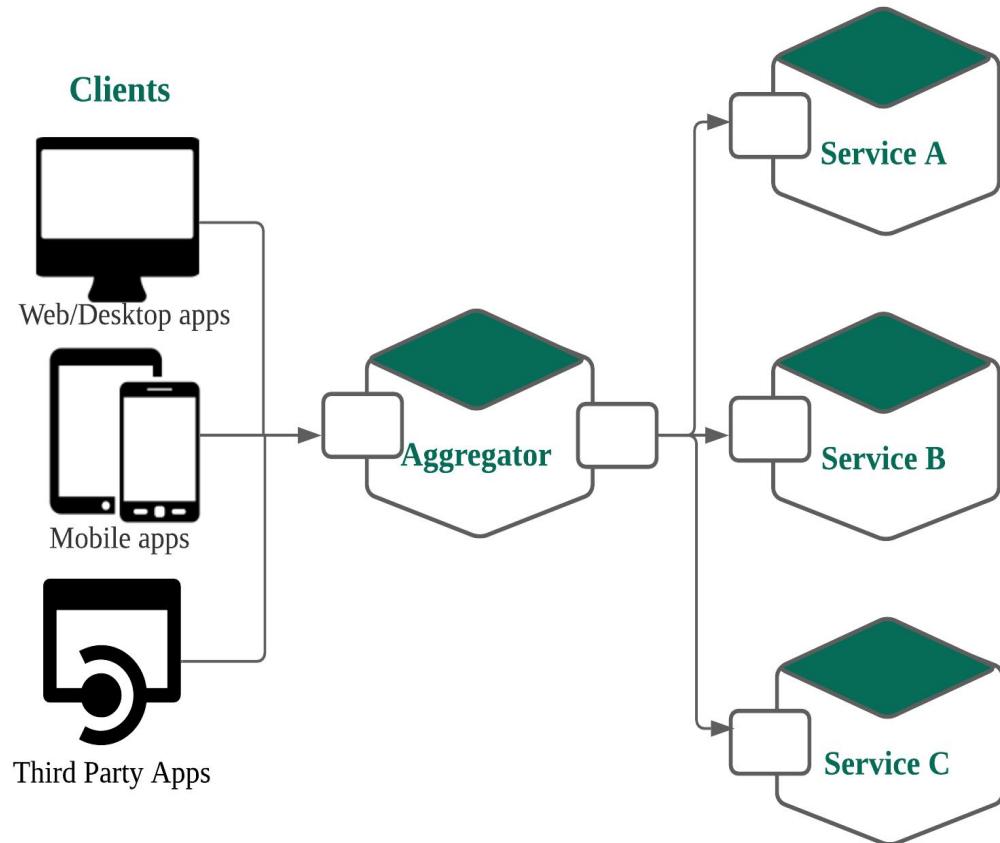
# Integration Patterns: Aggregator Pattern

## Scenario:

- Response from different microservice needs to be composed

## Uses:

- Composite microservice, performs some business logic on responses from different microservices
- Aggregator can be a microservice or the API gateway itself
- If the operation is complex, better to use a microservice
- If it is a simple operation then aggregation can be at API gateway



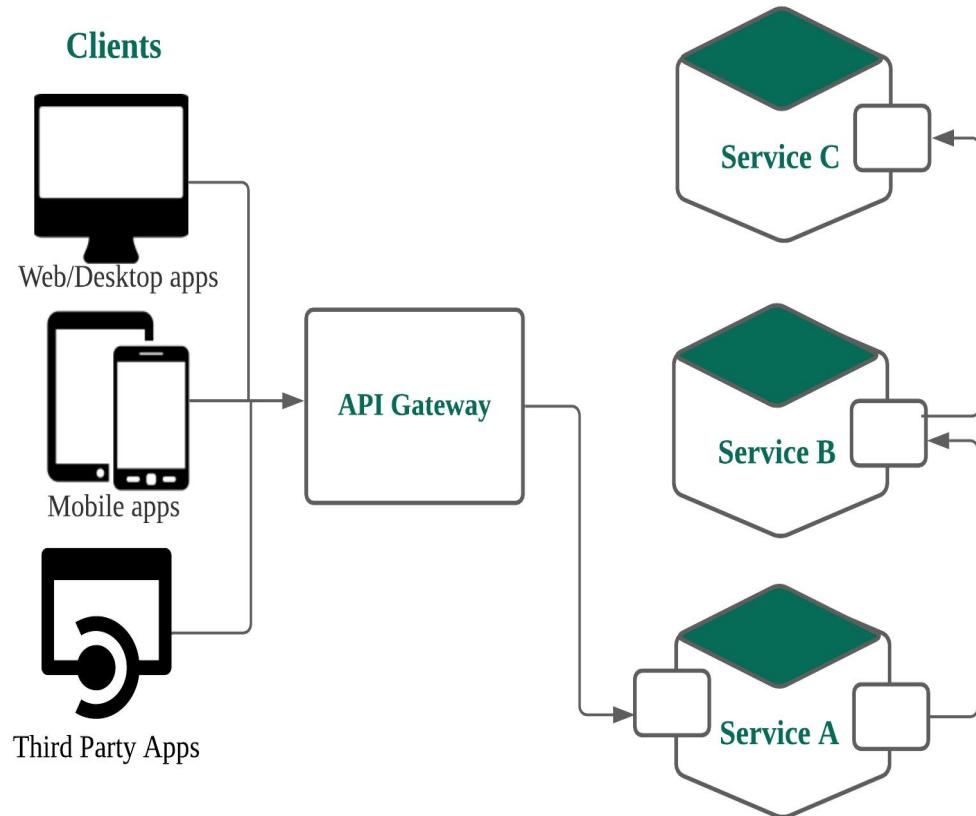
# Integration Patterns: Chained Microservice

## Scenario:

- Dependency of one microservice with another

## Uses:

- One microservice might have to depend on another microservice to accomplish a functionality
- Communication happens in a sequential manner
- Eg: Booking microservice needs to communicate with sensor microservices to get real-time availability



**Keep the length of sequence to minimal**

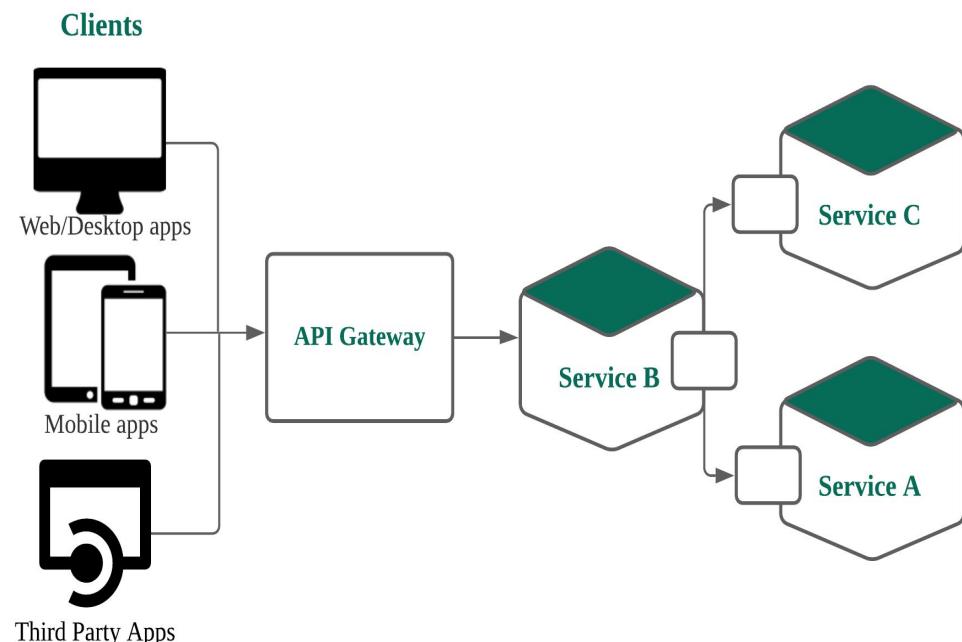
# Integration Patterns: Branch Microservice

## Scenario:

- One microservice might have to compose data obtained from different microservices

## Uses:

- Mix of **aggregator** and **chain** pattern
- One microservice can make parallel calls to different microservice to obtain data and perform operations
- The participating microservices in turn may or may not have branches
- Eg: Parking recommendation requires data from weather and parking lot availability to make recommendations



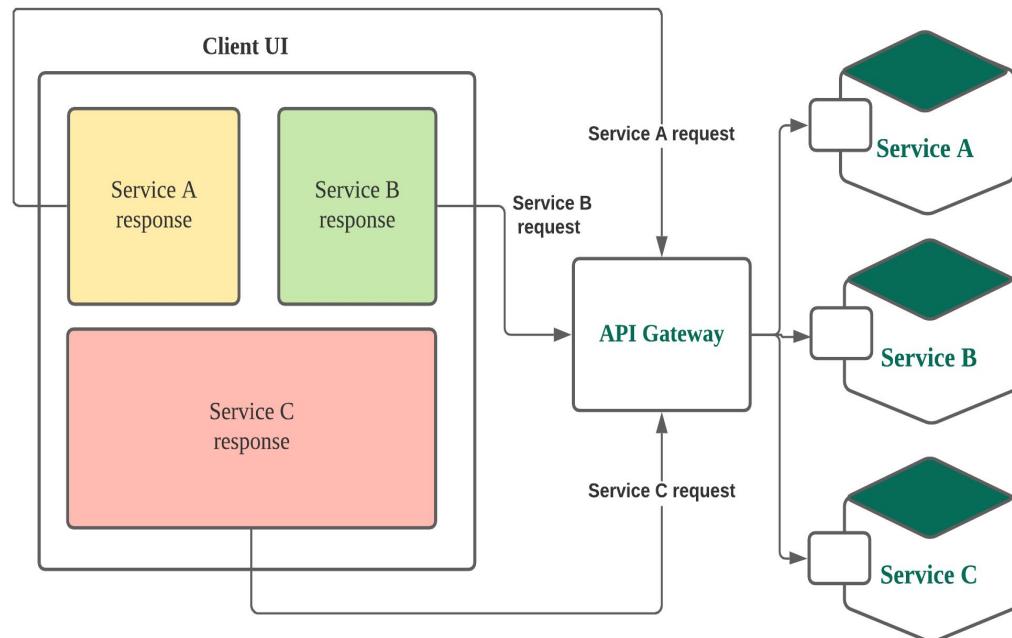
# Integration Patterns: Client UI Composition

## Scenario:

- User request is no longer driven by single backend request
- UI can be decomposed to different parts based on the required functionality

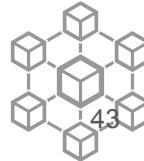
## Uses:

- Multiple services are responsible for populating different parts of UI
- Even if one section of a page goes down due to one microservice other parts still work (think about monolithic scenario !!)
- Eg: Single page application in ReactJS or Angular JS



# Database Integration: Patterns

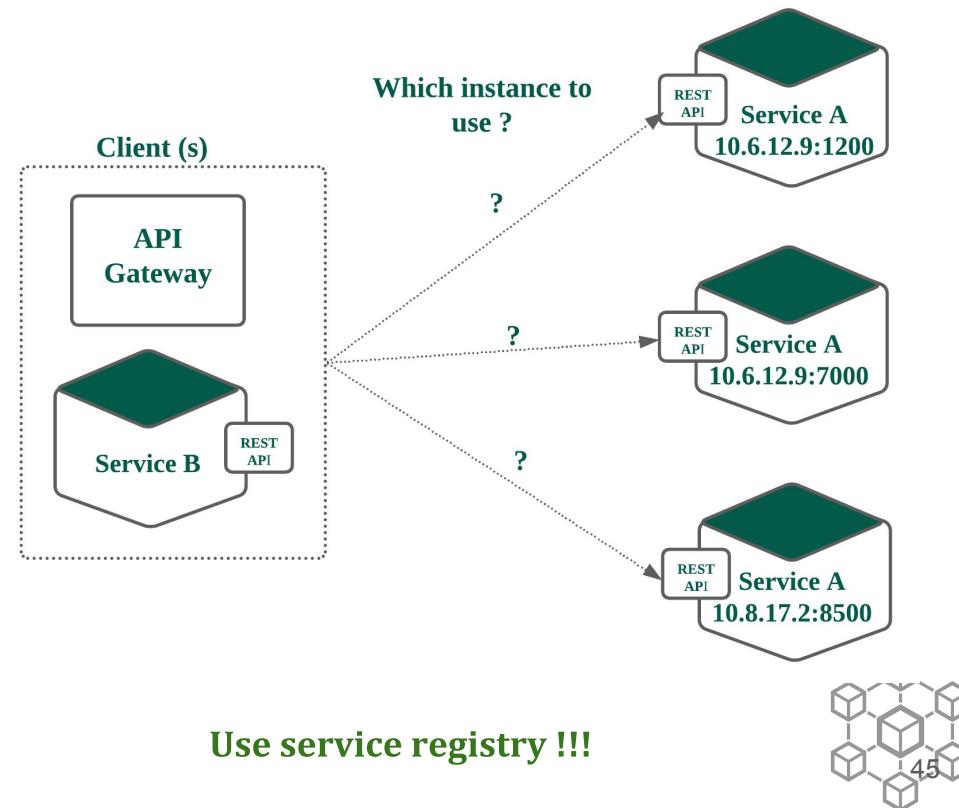
- **Shared Database:** Use if it is a brownfield application
  - Multiple microservices access the same database
  - Simpler to work with and data consistency can easily be ensured
  - High degree of coupling, use only if necessary between small set of services
- **Database per service:** The de-facto pattern (**Greenfield applications**)
  - Each service has its own private database with its own schema - access only via service API
  - Loose coupling - no need to depend on other service or database for data
  - Complexity to maintain different SQL and NoSQL databases
  - Transactions that requires updates/additions across different databases is hard to accomplish (use **SAGA** pattern)
  - Querying data from multiple databases is hard (API Composition, **CQRS Pattern**)



# **Microservice Deployment: Service Discovery**

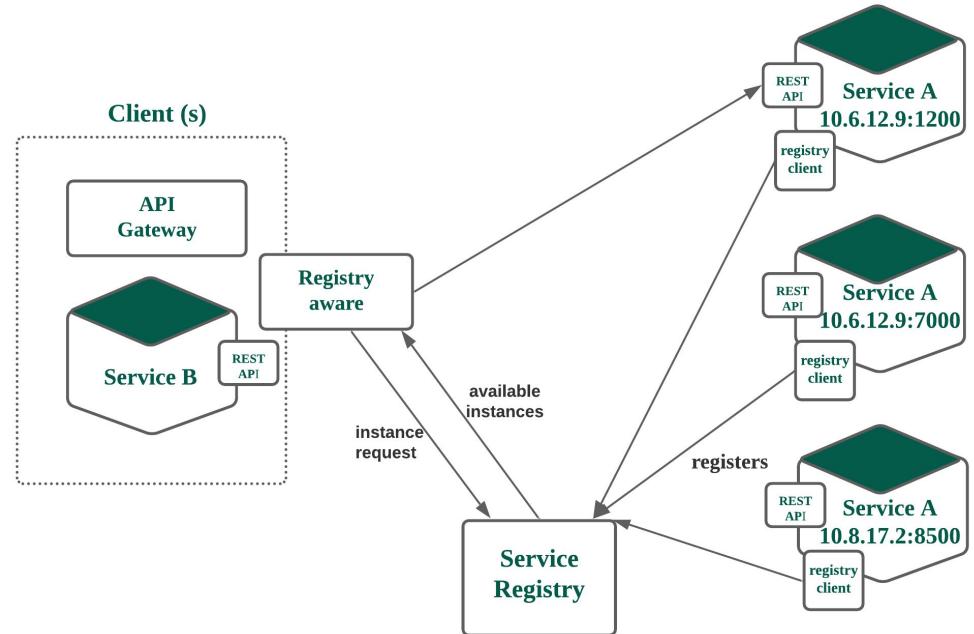
# Service Discovery

- Many instances of the same microservices are deployed in production for scalability, feature testing etc.
- Clients/third party services needs to discover instances of microservices
- Each microservice instance might be deployed in same/different machines (IP and Port)
- The service instances might appear and disappear dynamically

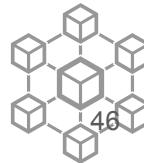


# Client Side Service Discovery

- Registry aware clients (sends request to service registry)
- Each microservice registers itself to service registry (as and when they are available)
- Service registry responds with the instance of the requested service to client
- Fewer network calls (just query service registry)
- Coupling between client and service registry

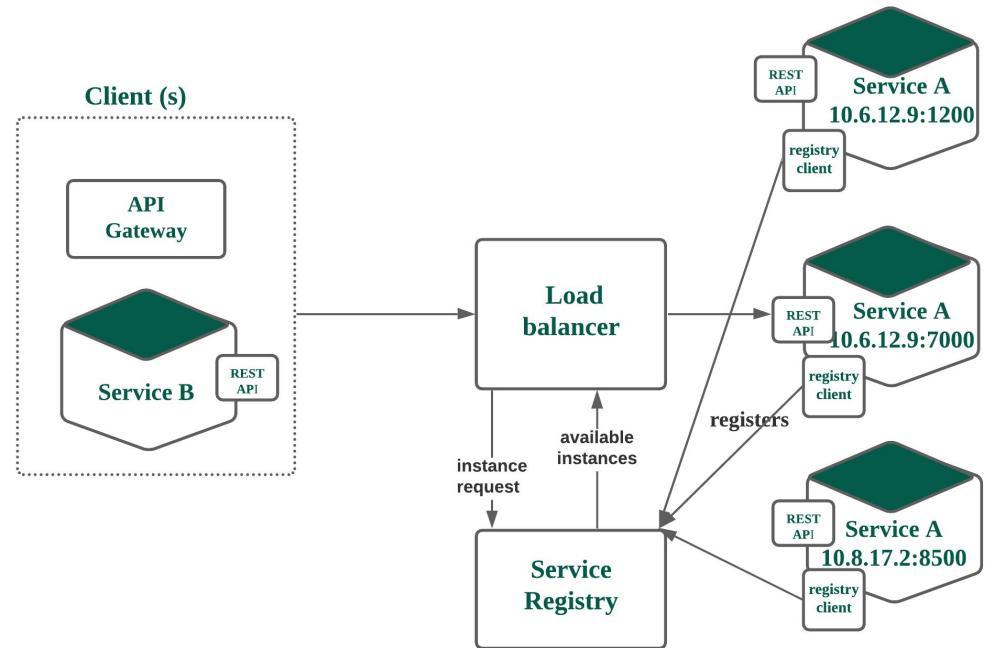


Eg: Netflix Eureka

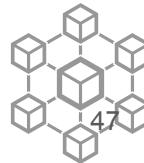


# Server Side Service Discovery

- Client (s) have no information on service registry (no coupling)
- Client (s) sends request to API gateway or load balancer
- The load balancer or API gateway uses Service registry to discover services
- Separation of logic from client
- Load balancer needs to be managed  
And replicated



Eg: Amazon ELB, Zookeeper



# Service Discovery: Ongoing Research

## Scenario:

- How to identify service instance that offers the best QoS ?
- Context of the client as well as the service can play a role

**Master thesis available !!!**



**UNIVERSITA' DEGLI STUDI - L'AQUILA**

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica  
Corso di Laurea in Informatica



L'Aquila, June 8th 2020

## Our Research:

- Use the QoS data as a driver
- Combination of machine learning techniques (deep learning for prediction and reinforcement learning for selection)

A thesis on the following topic is available "**Machine learning-driven Context-aware service discovery for Microservice architectures**"

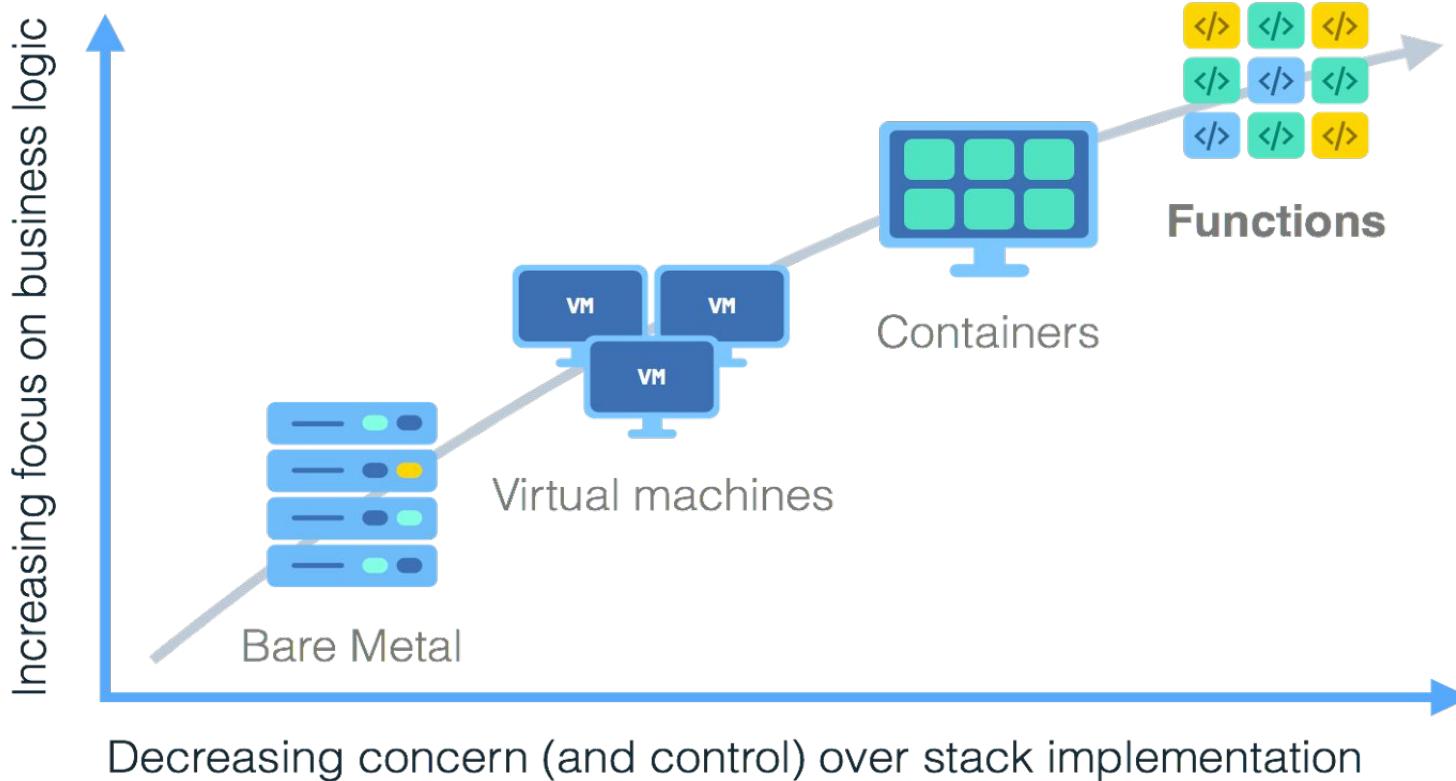
Thesis Degree: Master Thesis in Computer Science

Prior Knowledge: Basic understanding of Microservice architectures, Basic to intermediate knowledge of Machine Learning, Knowledge of dockers and deployment technologies is a plus

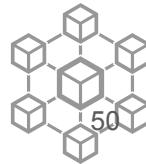
In collaboration with: Mauro Caporuscio, Associate Professor, Linnaeus University, Sweden and Karthik Vaidhyanathan, Ph.D. student, Gran Sasso Science Institute

# Microservice Deployment: How to Deploy ?

# Deployment: Towards Containers

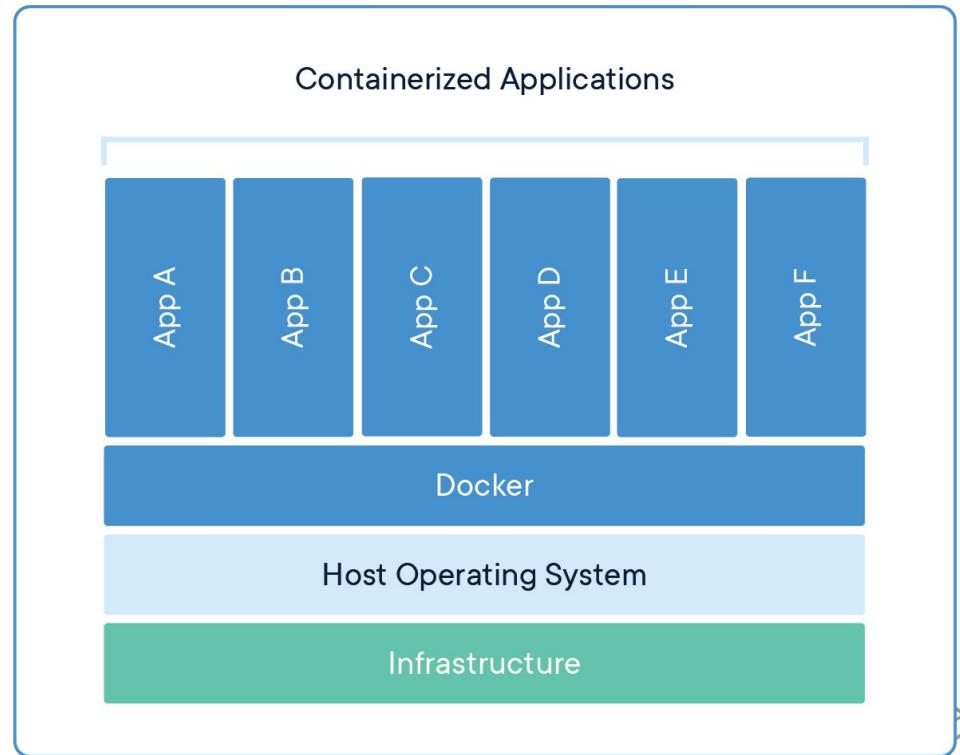


Source: Opening presentation of third international conference on serverless computing (WoSC), 2018

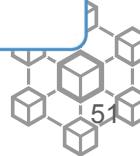


# Deployment: Using Containers

- Bundle microservice and dependencies in one single unit
- Increases portability
- Fosters continuous integration and deployment (milliseconds)
- Scaling and lifecycle management of services becomes easier
- Lightweight (MB's in size), better isolation and security
- Containers need to be managed (Kubernetes, Docker Swarm, etc)



Eg: Docker, Redshift, Mesos, GKE, Amazon ECS, etc



# **Microservices: So is it the holy grail**

# Some Funny yet Serious Facts



**Honest Status Page** @honest\_update · Oct 8, 2015

We replaced our monolith with micro services so that every outage could be more like a murder mystery.

21

3K

2.6K



**Gert de Pagter** @BackEndTea · Jan 7

...

Thanks to **microservices**, our JOINS are now over HTTP.

39

345

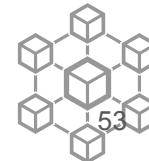
1.4K



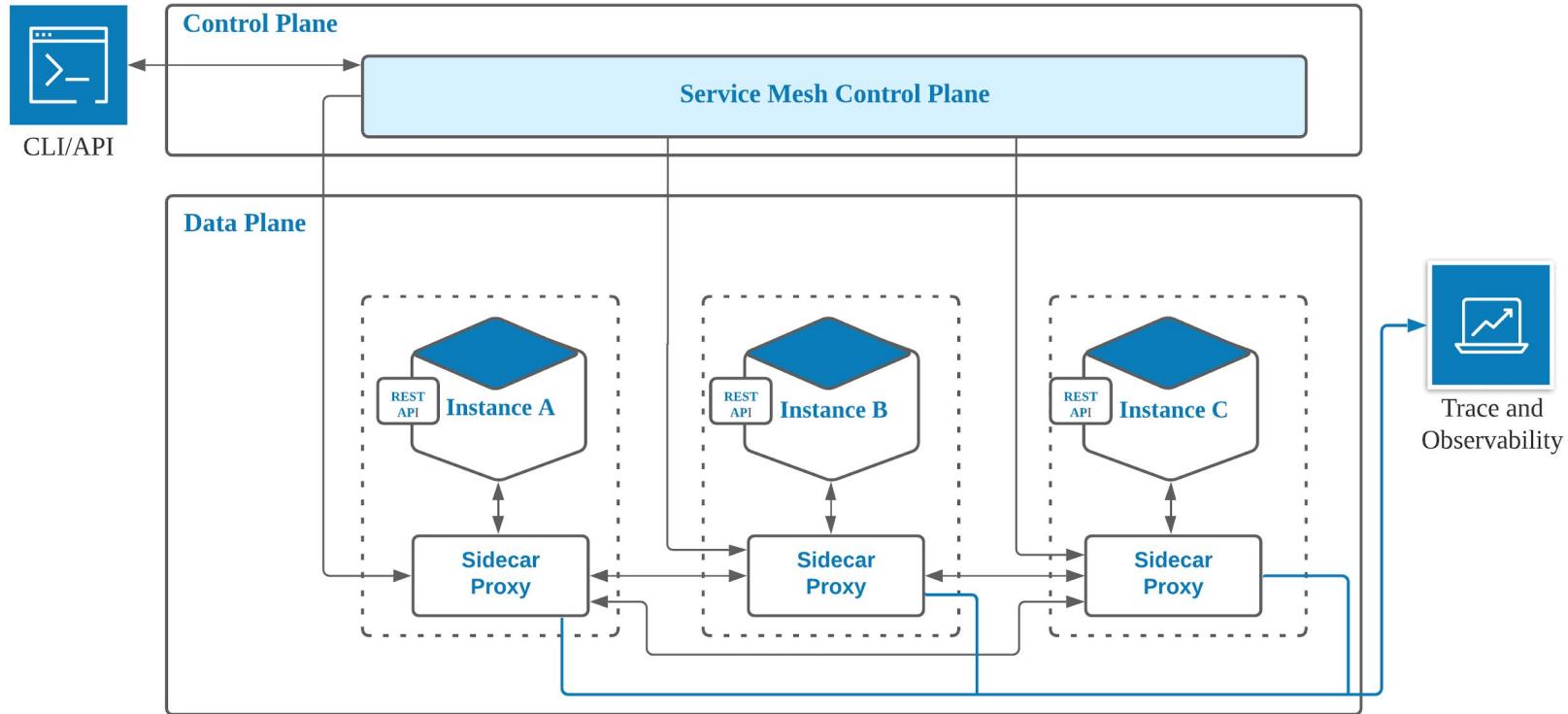
Monolith -> microservice but then we need docker, kubernetes, monitoring and what not  
!!!!

Source: Twitter

Karthik Vaidhyanathan



# Microservices: Service Mesh

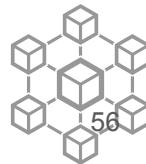
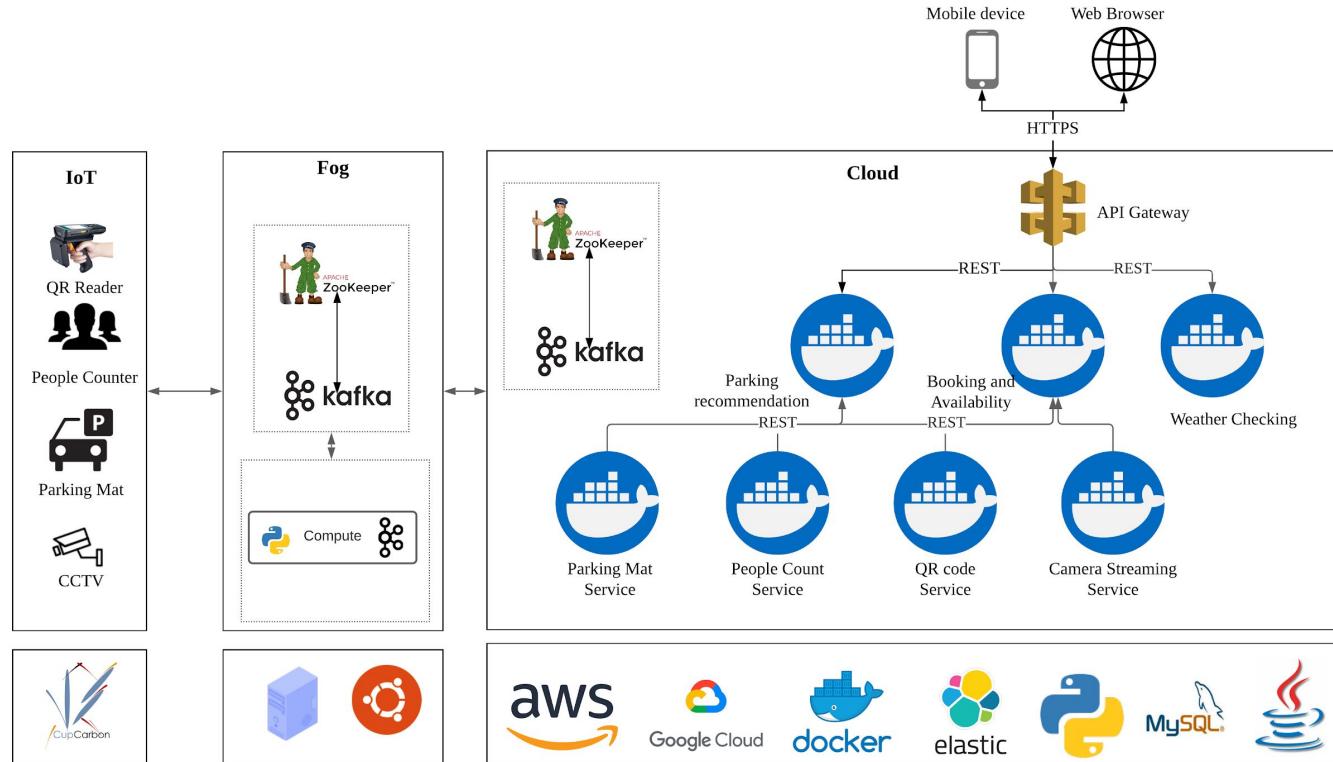


Gather performance metrics, traffic routing, load balancing, security, etc

Technologies: Istio, Envoy, etc

# Microservices: Small Demo

# Implementation of NdR Application



# Suggested Materials

- Books
  - Building Microservices, Sam Newman (<https://tinyurl.com/y5p5ajfl>)
  - Microservices Patterns: with Java, Chris Richardson (<https://tinyurl.com/yy9zyu9x>)
- Websites/blogs
  - <https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/>
  - <https://microservices.io>
  - <https://martinfowler.com/articles/microservices.html>
- Academic articles
  - [https://link.springer.com/chapter/10.1007/978-3-319-67425-4\\_12](https://link.springer.com/chapter/10.1007/978-3-319-67425-4_12)
- Videos
  - <https://www.youtube.com/watch?v=CZ3wIuvmHeM&t=1032s> - Chaos Engineering at Netflix
  - <https://www.youtube.com/watch?v=2yko4TbC8cI&t=169s> - Martin Fowler, Thoughtworks

