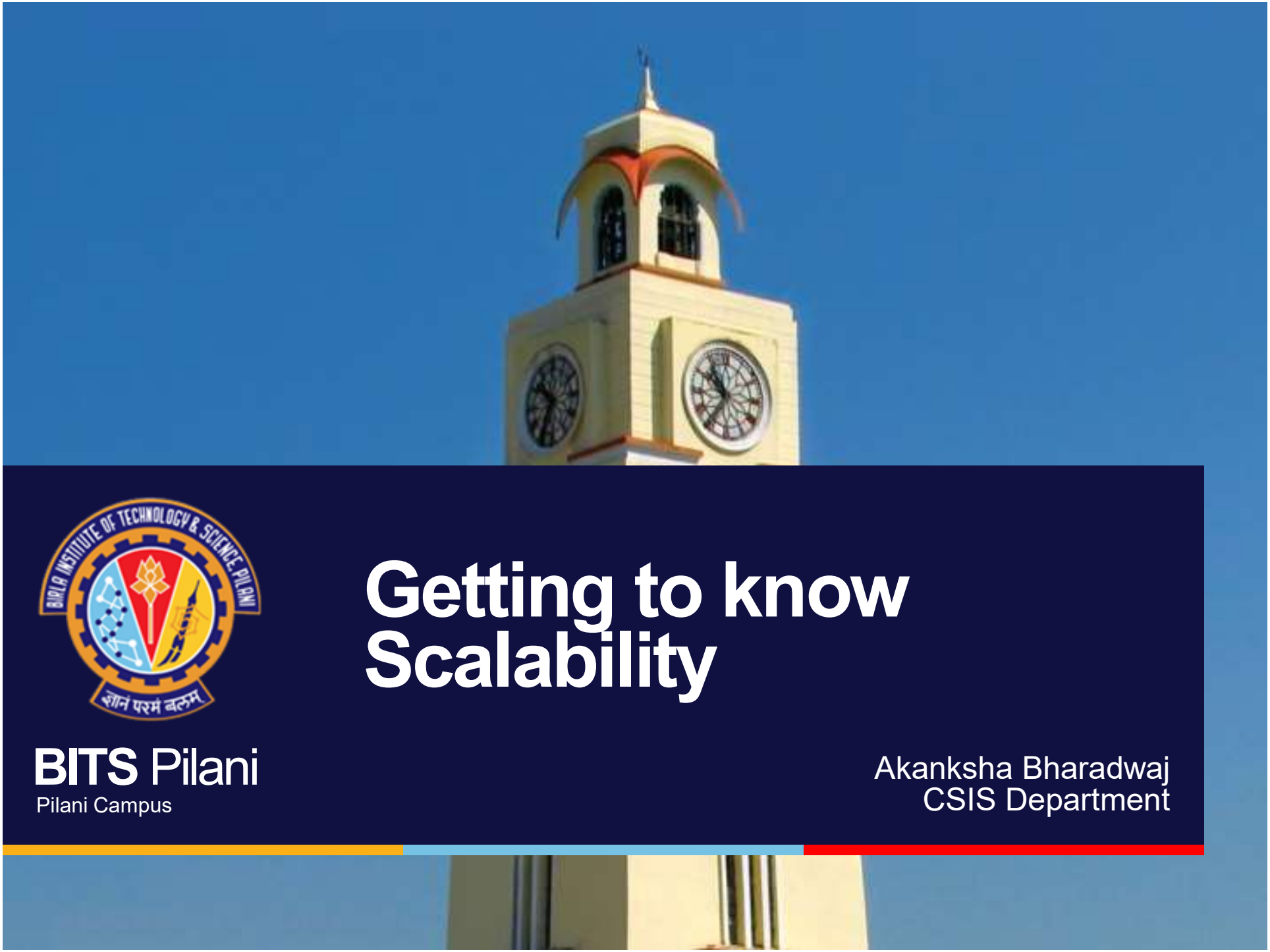


Getting to know Scalability

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 1

innovate achieve lead

BITS Pilani
Pilani Campus

Introduction to Performance, Consistency and availability

CAP Theorem

- The **CAP theorem** states that a distributed system can only guarantee two out of these three characteristics: Consistency, Availability, and Partition Tolerance.

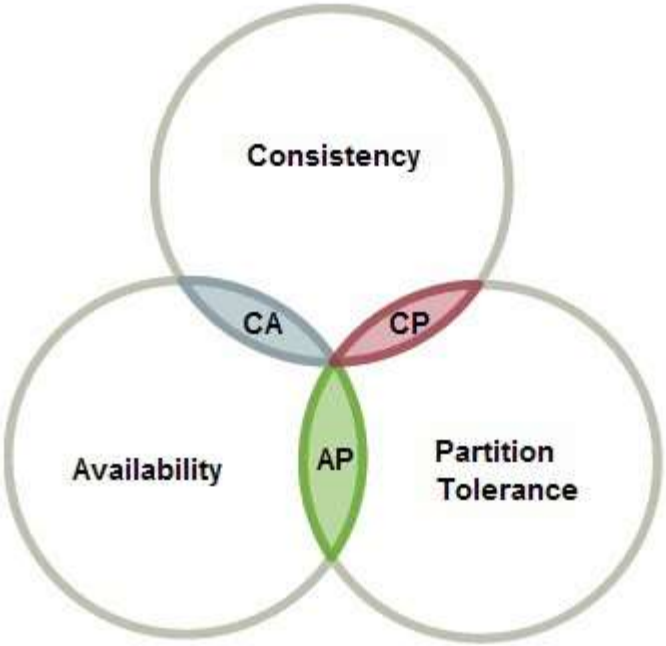


Image: Towardsdatascience

CAP Theorem

Consistency

Availability

Partition tolerance

Eventual Vs Strong Consistency

- Eventual consistency is a consistency model that enables the data store to be highly available.
- Strong Consistency simply means that all the nodes across the world should contain the same value for an entity at any point in time.

Performance

- The topmost reason for performance concerns is that the tasks we set our systems to perform have become much more complex over a period of time

Availability of a system

- Availability refers to a property of software that it is there and ready to carry out its task when you need it to be.
- Here are some of the key resources you can implement to make high availability possible:
  - Use multiple application servers
  - Spread out physically
  - Backup system

  .....

What is scalability?

- Scalability of an architecture refers to the fact that it can scale up to meet increased work loads.
- Types of Scalability
  - Vertical Scalability
  - Horizontal Scalability

Need for scalable architectures

Monolithic Architecture

- Monolith means composed all in one piece.
- Traditionally, applications were built on a monolithic architecture, a model in which all modules in an application are interconnected in a single, self-contained unit.
- They’re typically complex applications that encompass several tightly coupled functions.
- When all functionality in a system had to be deployed together, we consider it a **monolith**.

Advantages of Monolith

- Simplicity
- Network latency and security

Disadvantages of Monolith

- Scalability
- Slow development
- Long deployment cycle

Principles of Scalability

- Avoid single point of failure
- Scale horizontally, not vertically
- API
- Cache
- Maintenance and automation
- Asynchronous

All these mainly target three areas **Availability, Performance, and Reliability**

Guidelines for Building Highly Scalable Systems

- Avoid shared resources as they might become a bottleneck
- Avoid slow services
- Scaling Data tier is tricky
- Cache is the key
- Monitoring is important

Architecture's scalability requirements

- How important are the scalability requirements?
- Identify the scalability requirements early in the software life cycle so that that it allows the architectural framework to become sound enough as the development proceeds.
- System scalability criteria could include the ability to accommodate
  - Increasing number of users,
  - Increasing number of transactions per millisecond,
  - Increase in the amount of data

Challenges for Scalability

- Centralized approach
- Synchronous communication
- Cost

innovate achieve lead

BITS Pilani

Pilani Campus

Case Study

YouTube case Study

- YouTube is a video sharing website which uses the Client/server architecture.
- The NetScaler is implemented in the front of the web servers.
- YouTube uses the Apache with mod_fastcgi as the web servers.
- Python is used in the YouTube

Video Servers in YouTube

- In the video streaming, the bandwidth, hardware, and power consumption are three important issues
- YouTube applies cluster method to solve the consumption problem.
- YouTube also uses different strategies to deal with most popular videos and less popular videos.

Databases in YouTube

- Started with MySQL
- A shard architecture is designed to solve the replication problem of MySQL

Self Study

Example of scalable architecture:

https://www.youtube.com/watch?v=VHELcOe1gy0

References

- Textbooks and reference books mentioned in the handout
- https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf
- http://highscalability.com/youtube-architecture

Popular scaling approaches

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 2

Partitioning and Sharding

Introduction

- In many large-scale solutions, data is divided into *partitions* that can be managed and accessed separately.

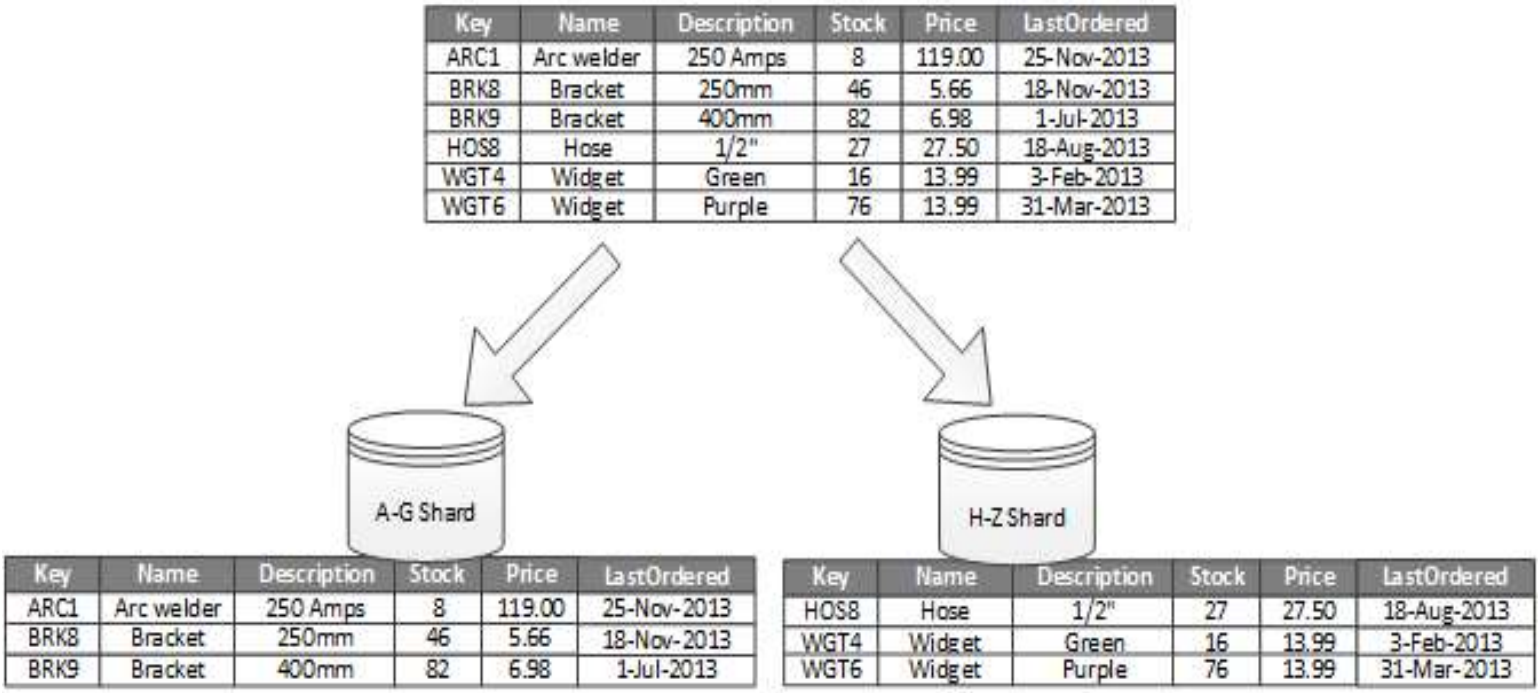
Why partition data?

- Improve scalability
- Improve performance
- Improve security
- Provide operational flexibility
- Improve availability

Types of Partitioning

- Horizontal partitioning
- Vertical partitioning
- Functional partitioning

Horizontal partitioning (Sharding)



Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013

Key	Name	Description	Stock	Price	LastOrdered
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

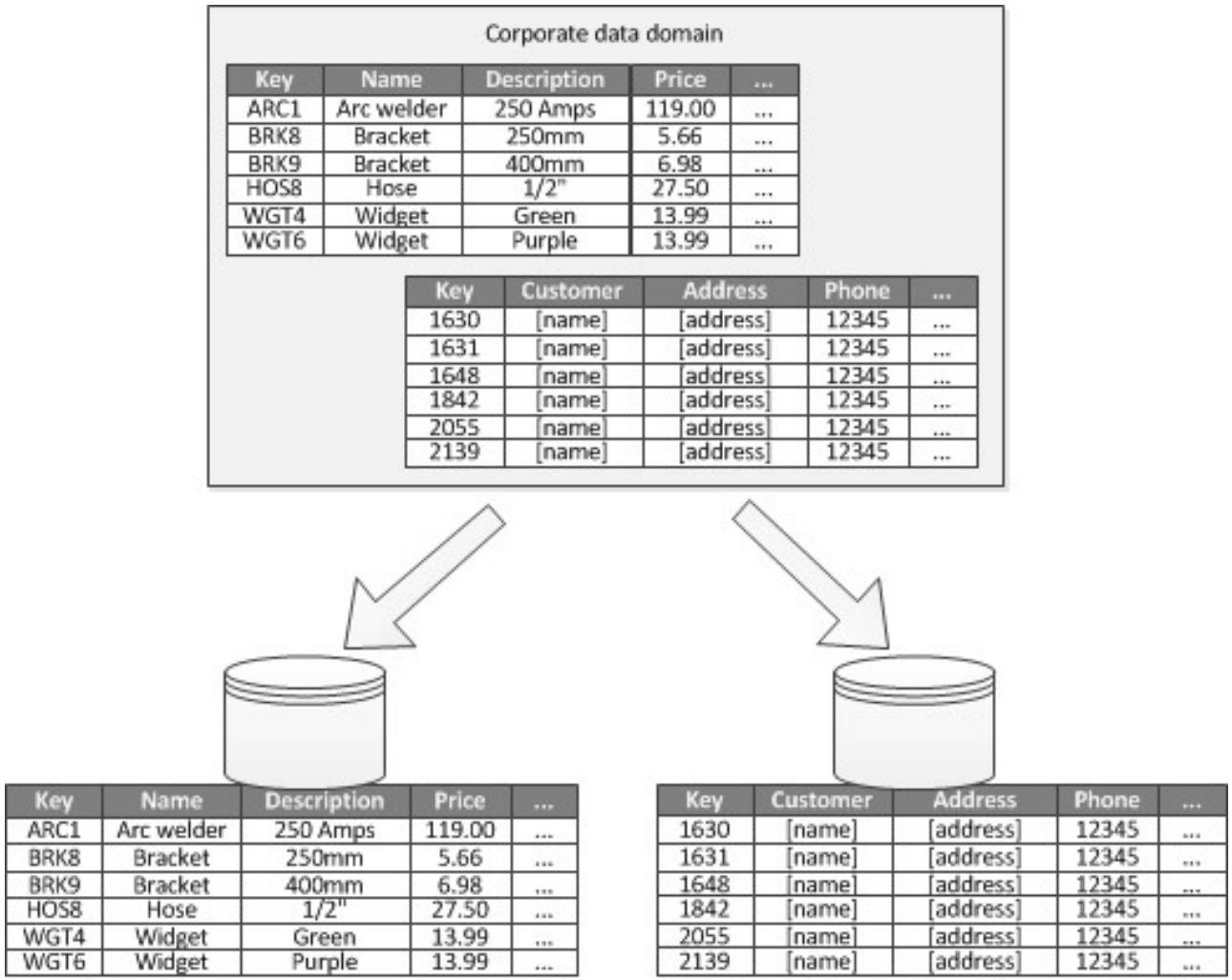
Vertical partitioning

Key	Name	Description	Stock	Price	LastOrdered
ARC1	Arc welder	250 Amps	8	119.00	25-Nov-2013
BRK8	Bracket	250mm	46	5.66	18-Nov-2013
BRK9	Bracket	400mm	82	6.98	1-Jul-2013
HOS8	Hose	1/2"	27	27.50	18-Aug-2013
WGT4	Widget	Green	16	13.99	3-Feb-2013
WGT6	Widget	Purple	76	13.99	31-Mar-2013

Key	Name	Description	Price
ARC1	Arc welder	250 Amps	119.00
BRK8	Bracket	250mm	5.66
BRK9	Bracket	400mm	6.98
HOS8	Hose	1/2"	27.50
WGT4	Widget	Green	13.99
WGT6	Widget	Purple	13.99

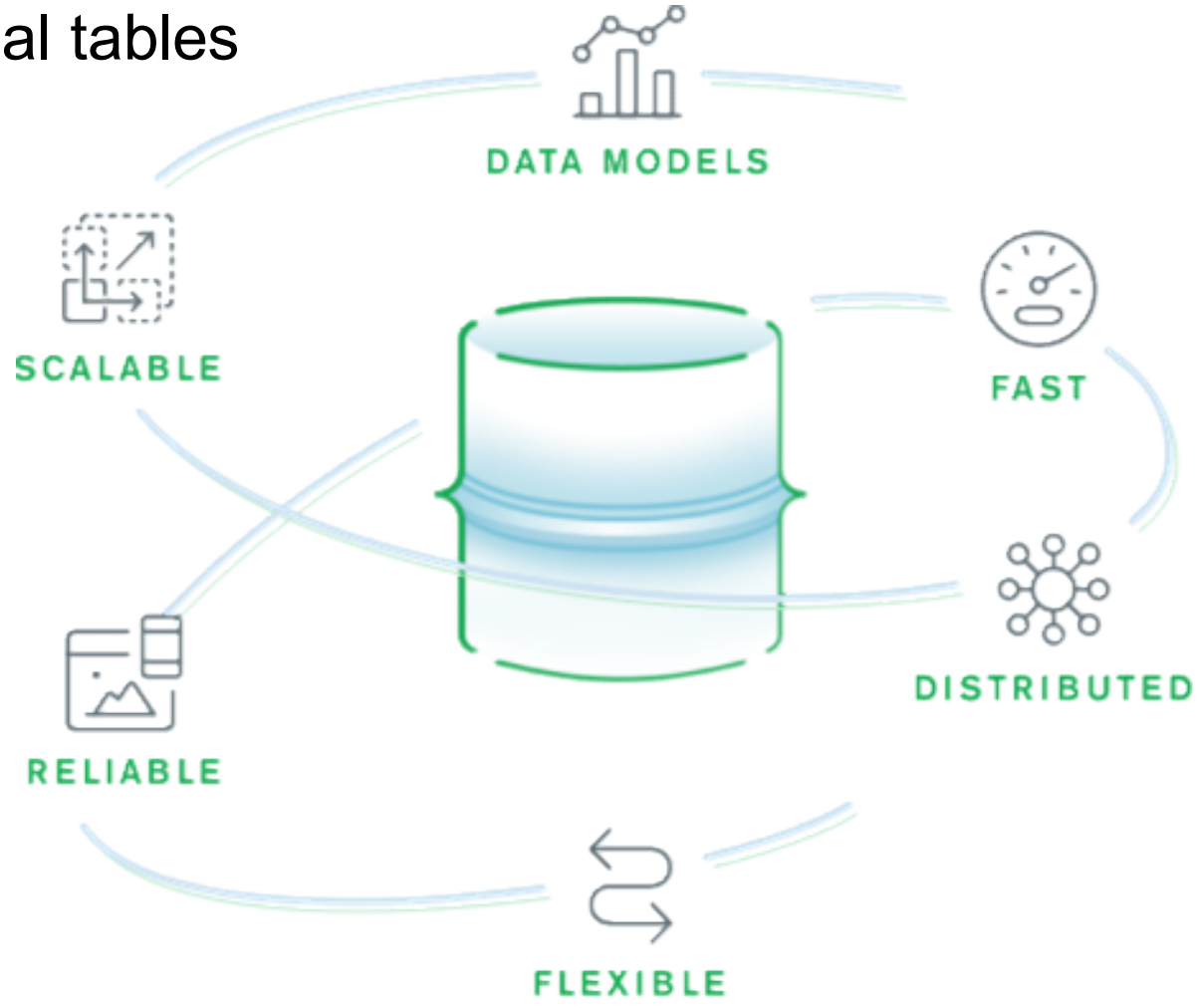
Key	Stock	LastOrdered
ARC1	8	25-Nov-2013
BRK8	46	18-Nov-2013
BRK9	82	1-Jul-2013
HOS8	27	18-Aug-2013
WGT4	16	3-Feb-2013
WGT6	76	31-Mar-2013

Functional partitioning



NoSQL

NoSQL databases are non tabular, and store data differently than relational tables



Document model

- These NoSQL databases replace the familiar rows and columns structure with a document storage model.

- Document-Oriented NoSQL DB stores and retrieves data as a key value pair

Graph model

- It is database that uses graph structures for semantic queries with nodes and edges
- The entity is stored as a node with the relationship as edges.
- Every node and edge has a unique identifier.

Key-value model

- In this NoSQL database model, a key is required to retrieve and update data.

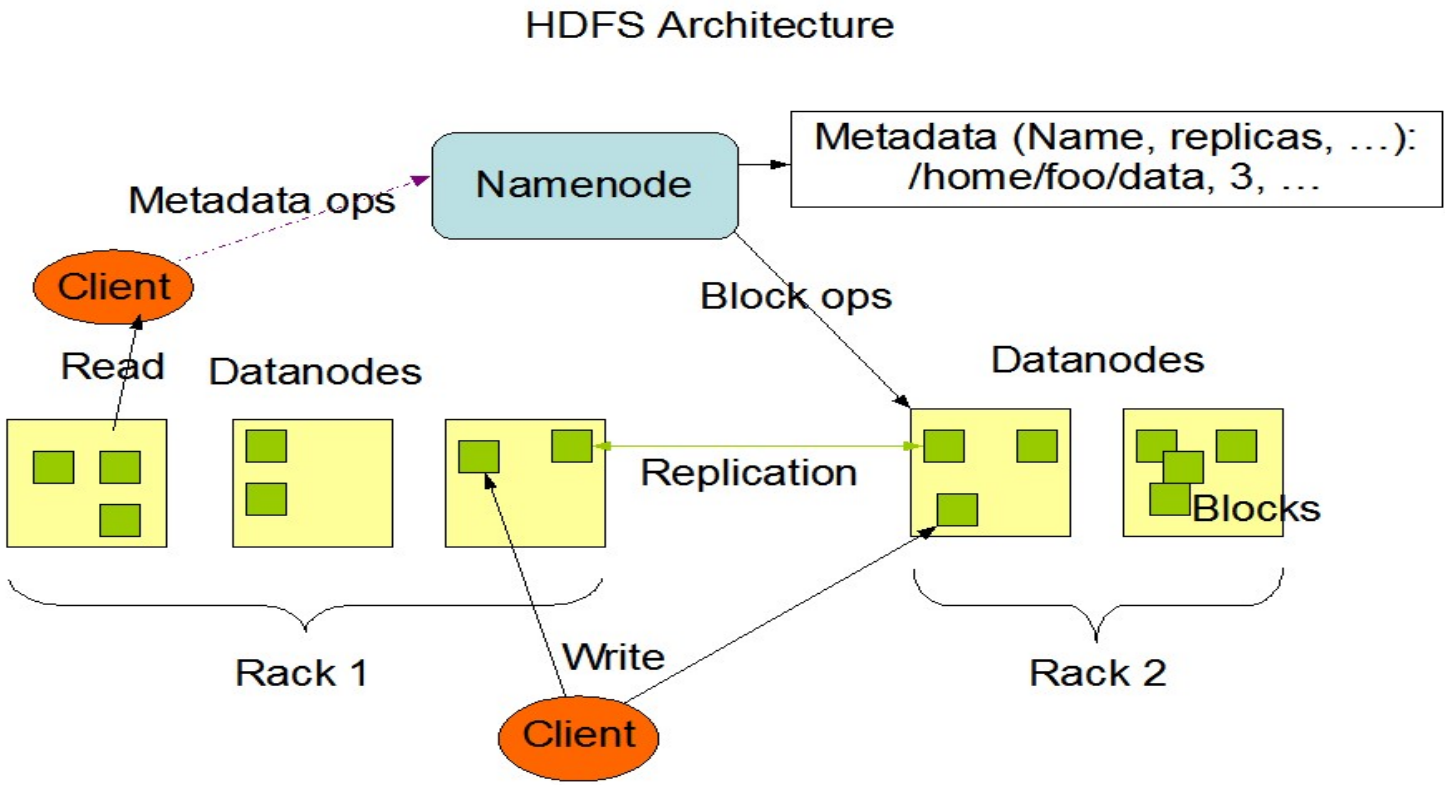
Column-based

- Column-oriented databases work on columns and are based on BigTable paper by Google.

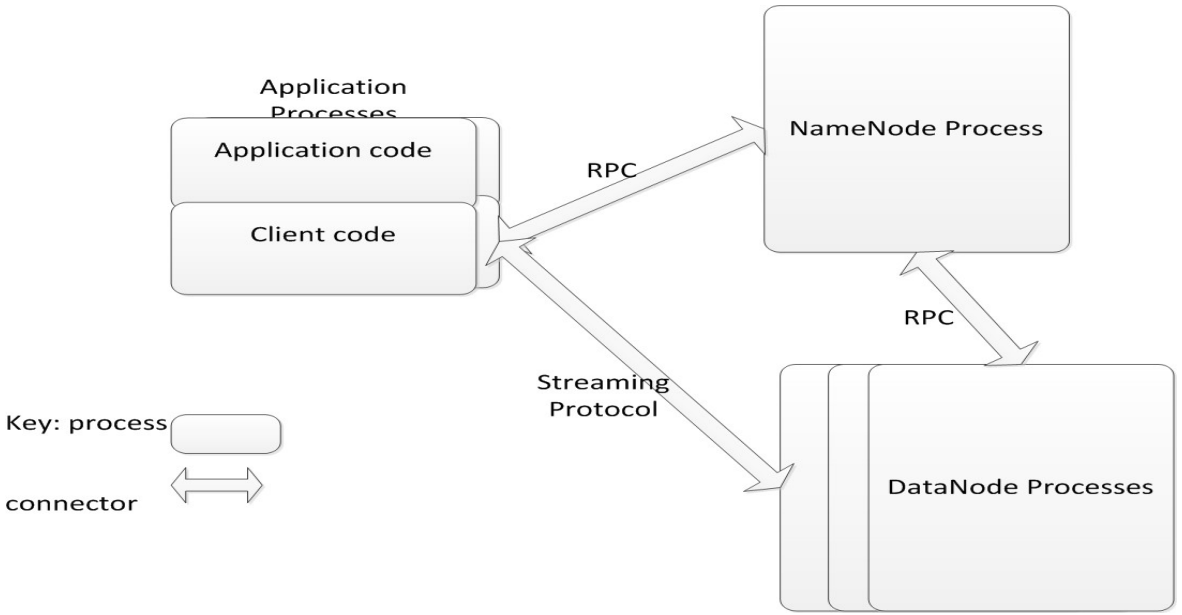
- Every column is treated separately. Values of single column databases are stored contiguously.

HDFS

- The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware.



HDFS Components



HDFS Write

- Application writes as to any file system
- Client buffers until it gets 64K block
- Client informs NameNode it wishes to write a new block
- NameNode returns list of three DataNodes to hold block
- Client sends block to first DataNode and informs DataNode of other two replicas.
- First DataNode writes block and sends it to second DataNode. Second DataNode writes block and sends it to last DataNode.
- Each DataNode reports to client when it has completed its write
- Client commits write to NameNode when it has heard from all three DataNodes.

HDFS Write – Failure Cases

- Client fails
  - Application detects and retries
  - Write is not complete until committed by Client
- NameNode fails
  - Backup NameNode takes over
  - Log file maintained to avoid losing information
  - DataNodes maintain true list of which blocks they each have
  - Client detects and retries
- DataNode fails
  - Client (or earlier DataNode in pipeline) detects and asks NameNode for different DataNode.
  - Since each block is replicated three times, a failure in a DataNode does not lose any data.

Goals of HDFS

- Fast recovery from hardware failures
- Access to streaming data
- Accommodation of large data sets
- Portability

How MapReduce Works

MapReduce are two functions: Map and Reduce. They are sequenced one after the other.

- The **Map** function takes input from the disk as <key,value> pairs, processes them, and produces another set of intermediate <key,value> pairs as output.

- The **Reduce** function also takes inputs as <key,value> pairs, and produces <key,value> pairs as output.

Combine and Partition

There are two intermediate steps between Map and Reduce.

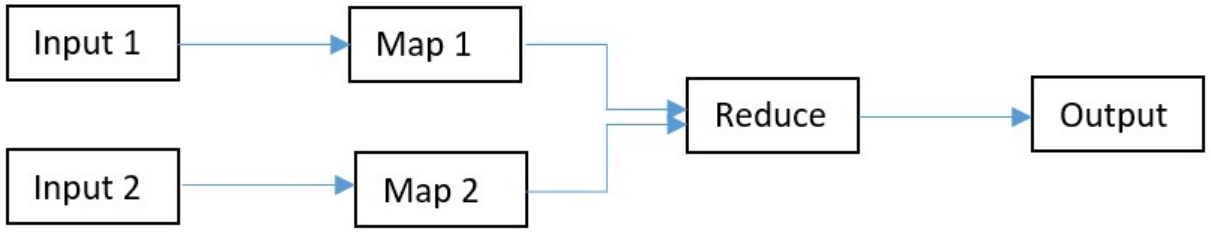
- **Combine** is an optional process. The combiner is a reducer that runs individually on each mapper server. It reduces the data on each mapper further to a simplified form before passing it downstream.

- **Partition** is the process that translates the <key, value> pairs resulting from mappers to another set of <key, value> pairs to feed into the reducer. It decides how the data has to be presented to the reducer and also assigns it to a particular reducer.

MapReduce Pattern

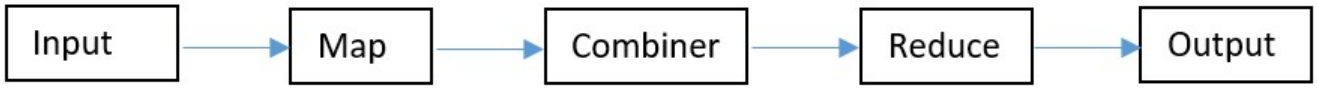
Input-Map-Reduce-Output

Input-Multiple Maps-Reduce-Output



MapReduce Pattern

Input-Map-Combiner-Reduce-Output



Example: Word count problem

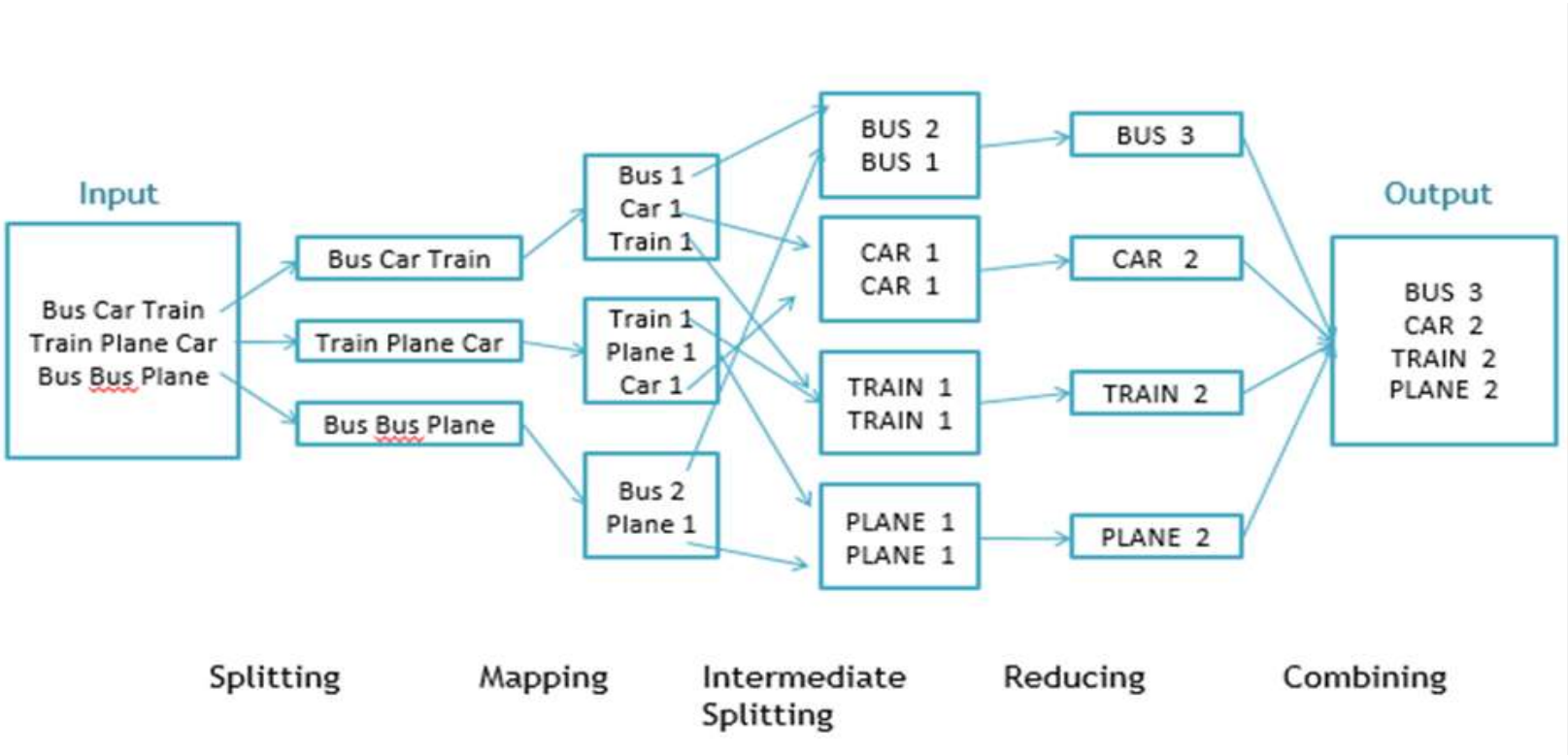


Image: dzone

Managing high velocity data streams

Content Delivery Network

- While back in 2005 about 1 billion people used the internet on a daily basis, today there are 3.5 billion internet users that share 4 Exabytes (4,000,000,000 Gigabytes) of data every single day.
- Basically a CDN is nothing more than a bunch of globally distributed computers that are directly connected and move data from one end to another.
- A good example of this is YouTube.

YouTube working

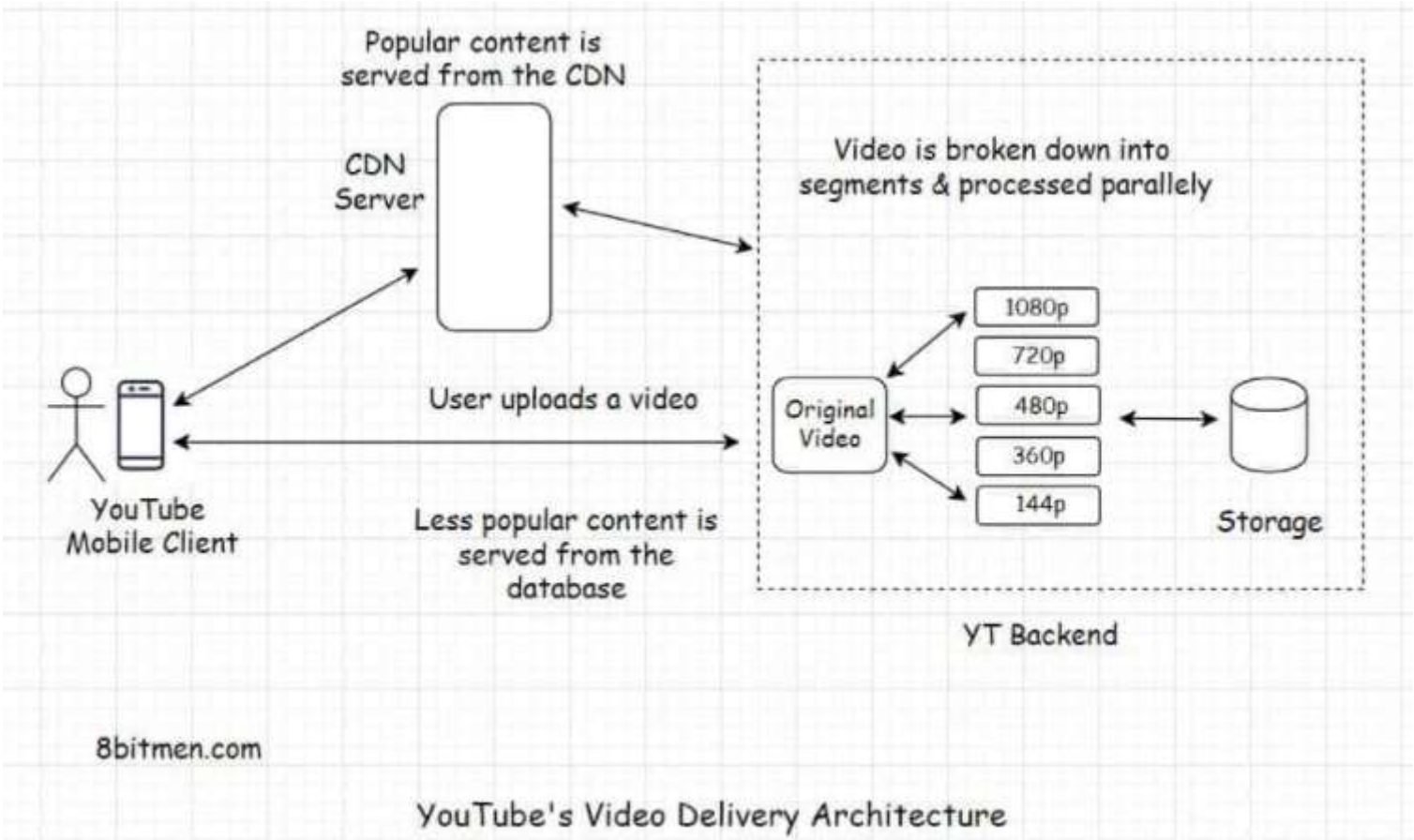
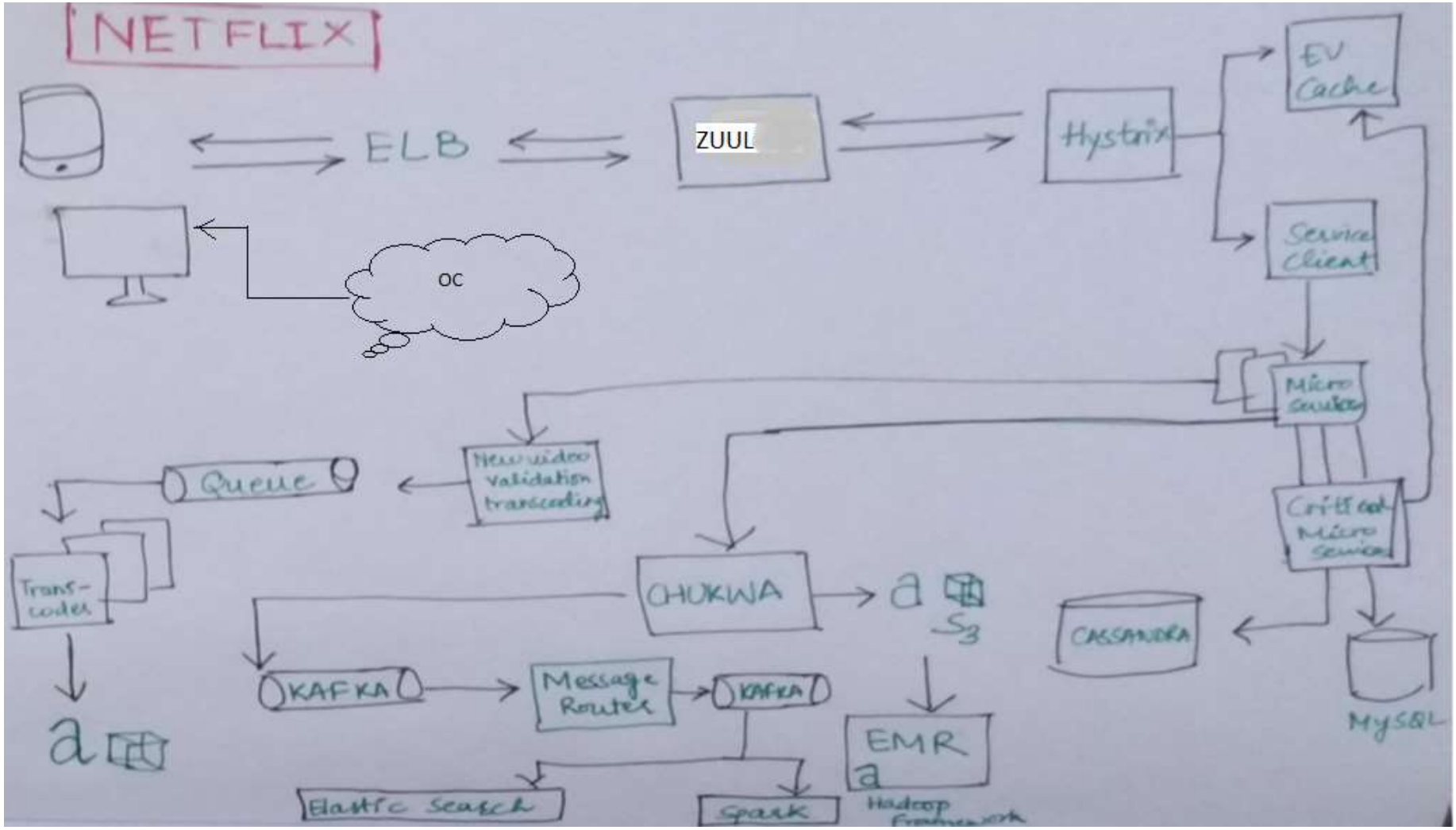


Image: google

Video streaming: Netflix

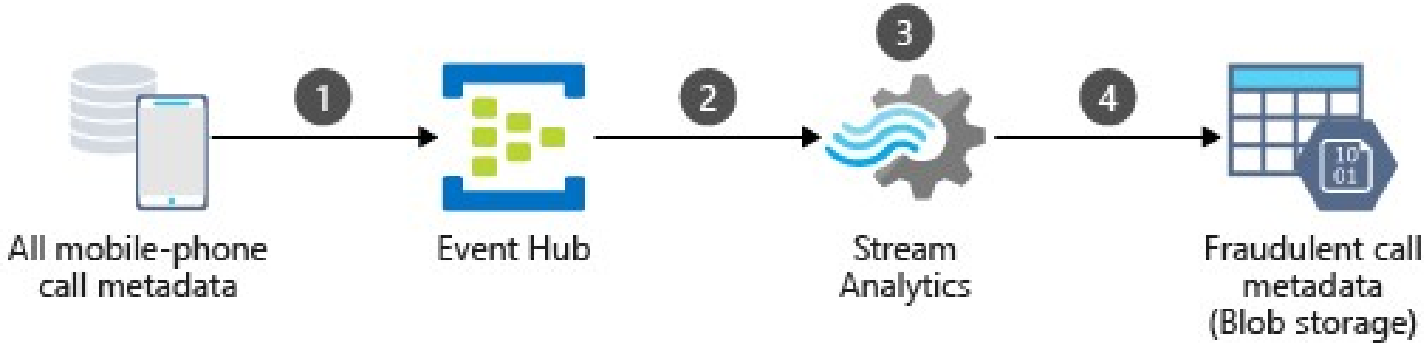
- Netflix launched in 1998. At first they rented DVDs through the US Postal Service. But Netflix saw the future was on-demand streaming video
- In 2007 Netflix introduced their streaming video-on-demand service
- It starts when you hit ‘Play.’
- When Netflix hands off your video to your ISP, they must carry it through their network to your home.

Netflix Architecture



Real Time Fraud Detection

Architecture



- Mobile phone call metadata is sent from the source system to an Azure Event Hubs instance.
- A Stream Analytics job is started, which receives data via the event hub source.
- The Stream Analytics job runs a predefined query to transform the input stream and analyze it based on a fraudulent-transaction algorithm.
- The Stream Analytics job writes the transformed stream representing detected fraudulent calls to an output sink in Azure Blob storage.

Web conferencing: Zoom

- Zoom customers with Business subscriptions can enjoy three times as many video participants in their meetings at no additional cost — and without doing a thing.
- such an increase is made possible in the way the Zoom platform is engineered
- From the very beginning, Zoom was engineered to be cloud-native and optimized for video.

Web conferencing: Zoom

There are two important aspects of Zoom's technology stack:

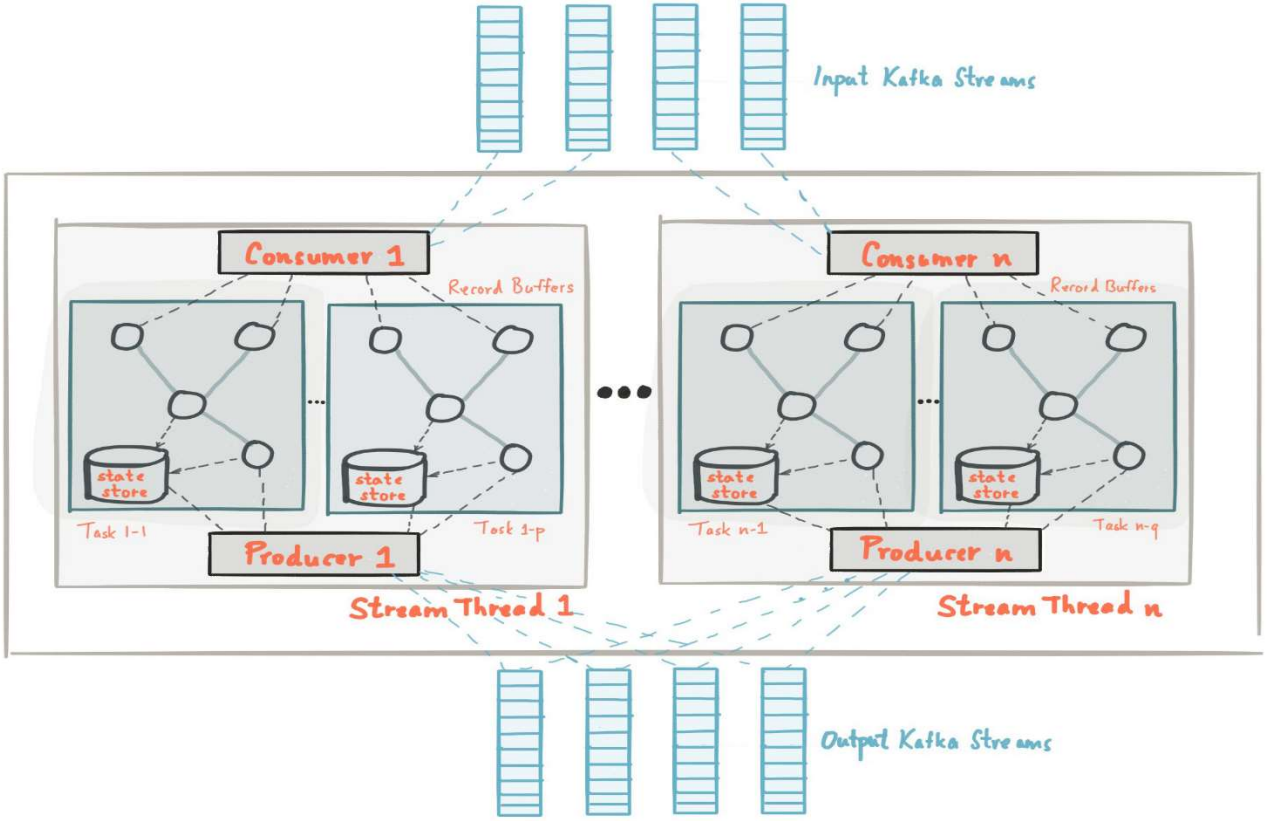
- Cloud network
- Video architecture
  - Distributed architecture
  - Multimedia routing
  - Multi-bitrate encoding
  - Application layer quality of service

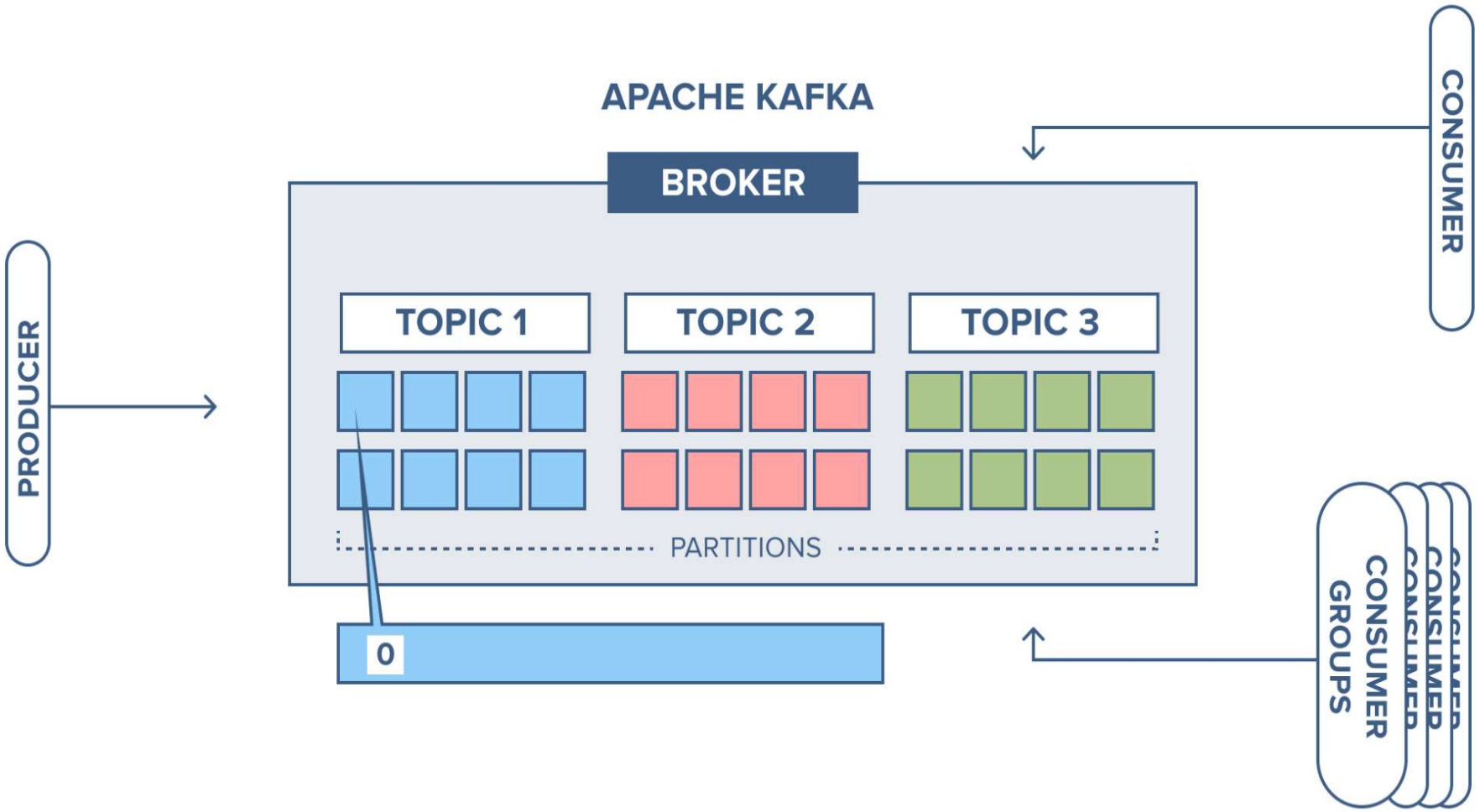
What is Kafka?

- Apache Kafka is a **publish-subscribe based durable messaging system**.
- A messaging system sends messages between processes, applications, and servers.
- Apache Kafka is a software where topics can be defined (think of a topic as a category), applications can add, process and reprocess records.

Kafka

- Kafka Streams simplifies application development by building on the Kafka producer and consumer libraries




APACHE KAFKA
BROKER
TOPIC 1
TOPIC 2
TOPIC 3
PARTITIONS
0
PRODUCER
CONSUMER
CONSUMER GROUPS

Kafka Related Concepts

- Kafka Topics
- Partitioning
- Kafka brokers
- Replication
- Kafka Producers
- Kafka Consumers
- Kafka Connect
- Kafka Streams

Kafka

There are close links between Kafka Streams and Kafka in the context of parallelism:

- Each **stream partition** is a totally ordered sequence of data records and maps to a Kafka **topic partition**.
- A **data record** in the stream maps to a Kafka **message** from that topic.
- The **keys** of data records determine the partitioning of data in both Kafka and Kafka Streams, i.e., how data is routed to specific partitions within topics.

What is Edge Computing?

- Edge computing is a distributed information technology (IT) architecture in which client data is processed at the periphery of the network, as close to the originating source as possible.

- It helps to provide server resources, data analysis, and artificial intelligence to data collection sources and cyber-physical sources like smart sensors and actuators

Edge computing: IoT systems

- Rapidly increasing numbers of IoT devices and resultant data, mean that new techniques are needed to meet customer requirements and ensure effective management need to be explored

Key Benefits of Edge for the IoT

- Low latency
- Longer battery life for IoT devices
- Access to data analytics and AI
- Resilience
- Scalability
- More efficient data management

Example: IoT Image and Audio Processing

- IoT edge introduces new ways of analysing data without having to backhaul the entire image or audio stream
- An edge cloudlet can be used to process the image, video or audio data to determine key information, such as licence plate numbers or the number of people in an area.

Self Study

- https://developer.cisco.com/docs/webex-meetings/#!architecture/overview
- Kafka

References

- https://docs.microsoft.com/en-us/azure/architecture/best-practices/data-partitioning
- https://www.mongodb.com/nosql-explained
- https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html
- https://netflixtechblog.com/
- http://highscalability.com/youtube-architecture
- https://docs.microsoft.com/en-us/azure/architecture/example-scenario/data/fraud-detection
- https://blog.zoom.us/
- https://kafka.apache.org/11/documentation/streams/architecture
- https://www.gsma.com/iot/wp-content/uploads/2018/11/IoT-Edge-Opportunities-c.pdf
- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7696529/
- Reading material available on Confluent

BITS Pilani presentation

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 3

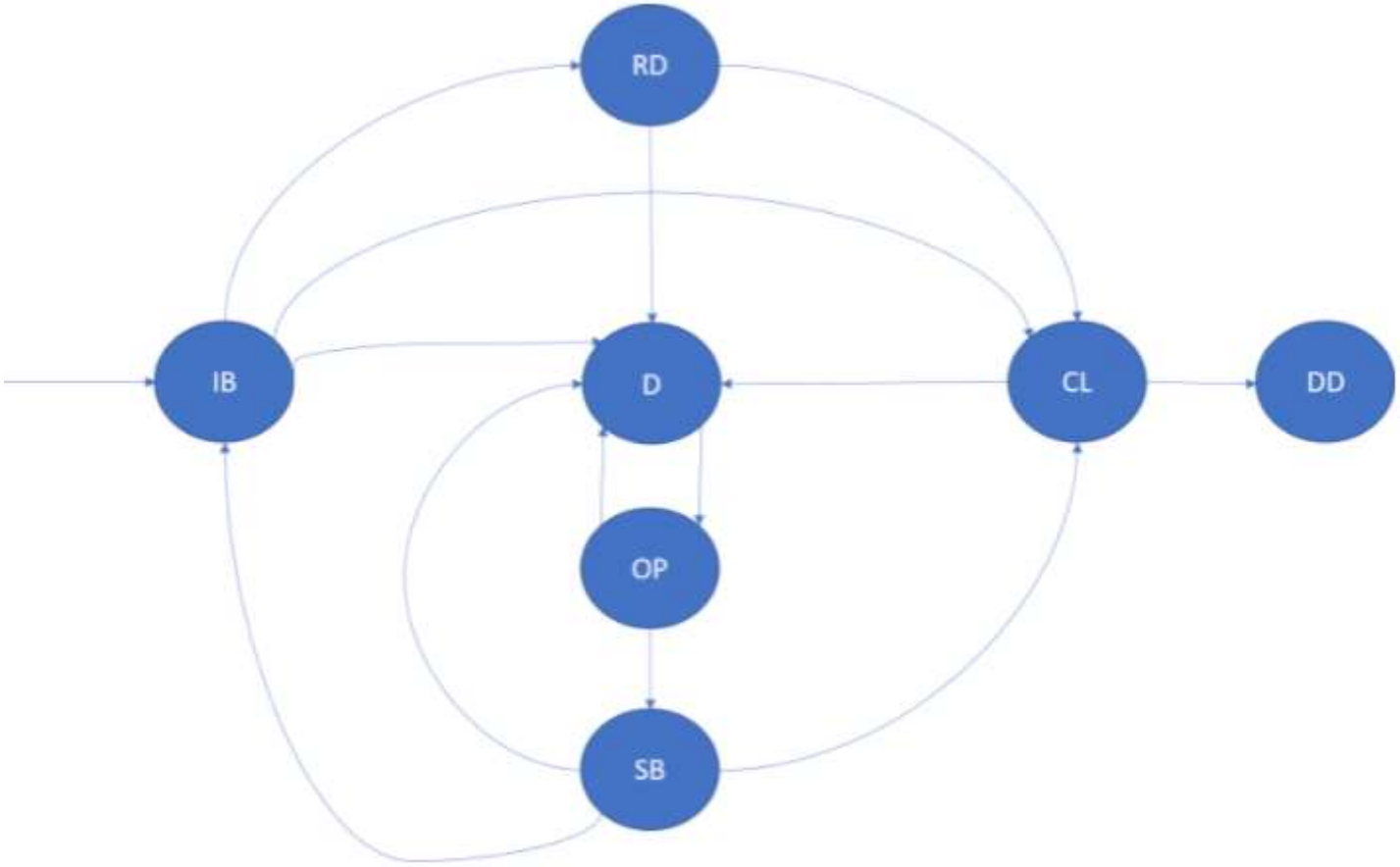
Managing high volume transactions

Service Replicas

- An instance of a stateless service is a copy of the service logic that runs on one of the nodes of the cluster.

- A replica of a stateful service is a copy of the service logic running on one of the nodes of the cluster.

Lifecycle of stateful replicas



InBuild (IB)

- It is a replica that's created or prepared for joining the replica set.

Types:

- Primary InBuild replicas
- IdleSecondary InBuild replicas
- ActiveSecondary InBuild replicas

Ready (RD)

- A Ready replica is a replica that's participating in replication and quorum acknowledgement of operations.
- The ready state is applicable to primary and active secondary replicas.

Closing (CL)

A replica enters the closing state in the following scenarios:

- Shutting down the code for the replica
- Removing the replica from the cluster

Dropped (DD)

- In the dropped state, the instance is no longer running on the node.
- There is also no state left on the node.

Down (D)

- In the down state, the replica code is not running, but the persisted state for that replica exists on that node.
- A down replica is opened by Service Fabric as required, for example, when the upgrade finishes on the node.
- The replica role is not relevant in the down state.

Opening (OP)

- A down replica enters the opening state when Service Fabric needs to bring the replica back up again.
- If the application host or the node for an opening replica crashes, it transitions to the down state.
- The replica role is not relevant in the opening state.

StandBy (SB)

- A StandBy replica is a replica of a persisted service that went down and was then opened.
- After the StandBy Replica Keep Duration expires, the standby replica is discarded.
- If the application host or the node for a standby replica crashes, it transitions to the down state.

Replica role

The role of the replica determines its function in the replica set:

- Primary (P)
- ActiveSecondary (S)
- IdleSecondary (I)
- None (N)
- Unknown (U)

What is load balancing?

- Load balancing is defined as the methodical and efficient distribution of network or application traffic across multiple servers in a server farm.

What are load balancers?

A load balancer may be:

- It can be a physical device.
- It can be an incorporated into application delivery controllers (ADCs)

Command Query Responsibility Segregation (CQRS)

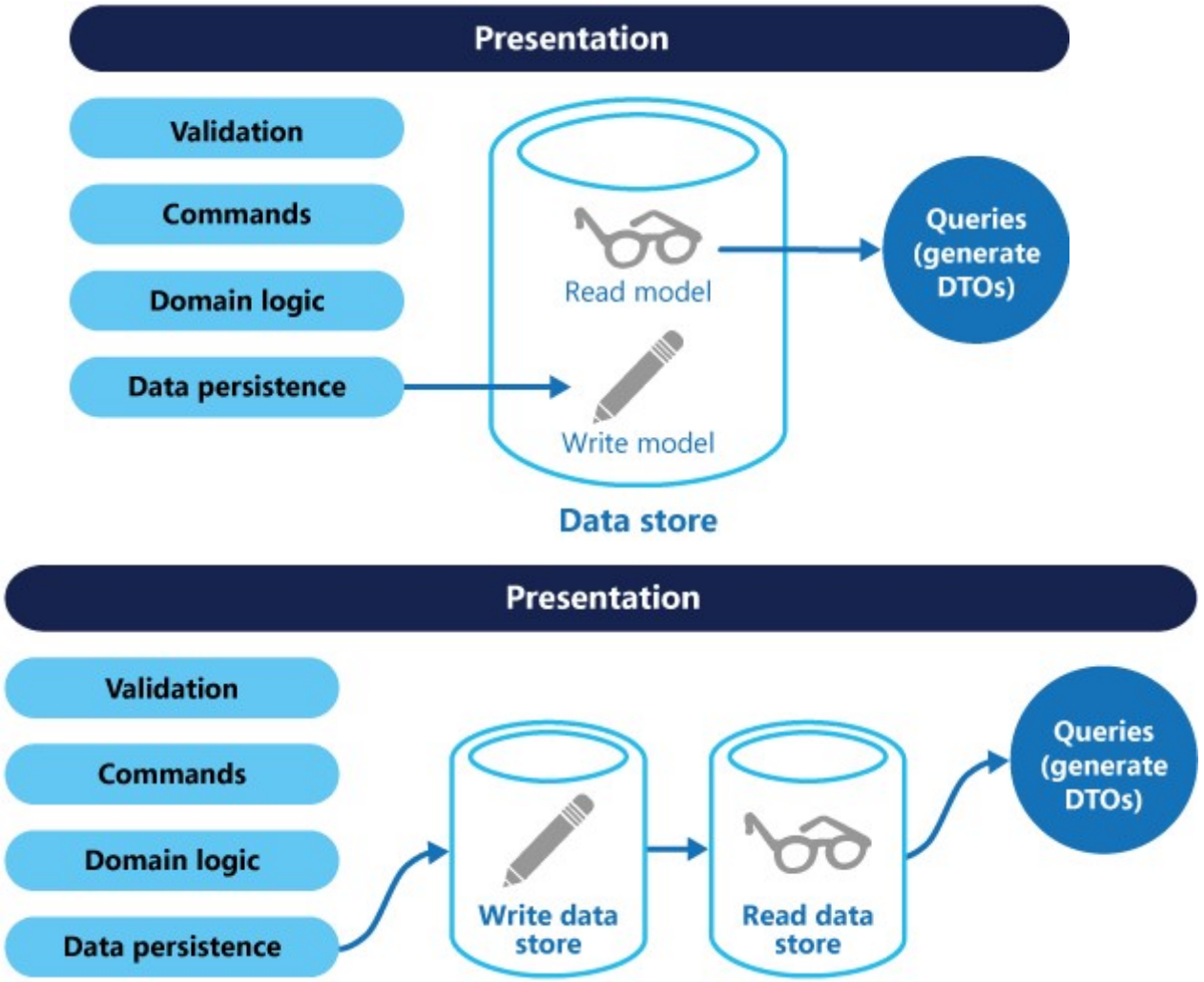
- It is a pattern that separates read and update operations for a data store.

- Implementing CQRS in your application can maximize its performance, scalability, and security.

Need for CQRS

- In traditional architectures, the same data model is used to query and update a database.
- Data contention can occur when operations are performed in parallel on the same set of data
- Managing security and permissions can become complex, because each entity is subject to both read and write operations

How CQRS works?



Benefits of CQRS

- Independent scaling
- Optimized data schemas
- Security
- Separation of concerns
- Simpler queries

Some challenges of implementing CQRS

- Complexity
- Messaging
- Eventual consistency

Protocols for communication

Synchronous protocol

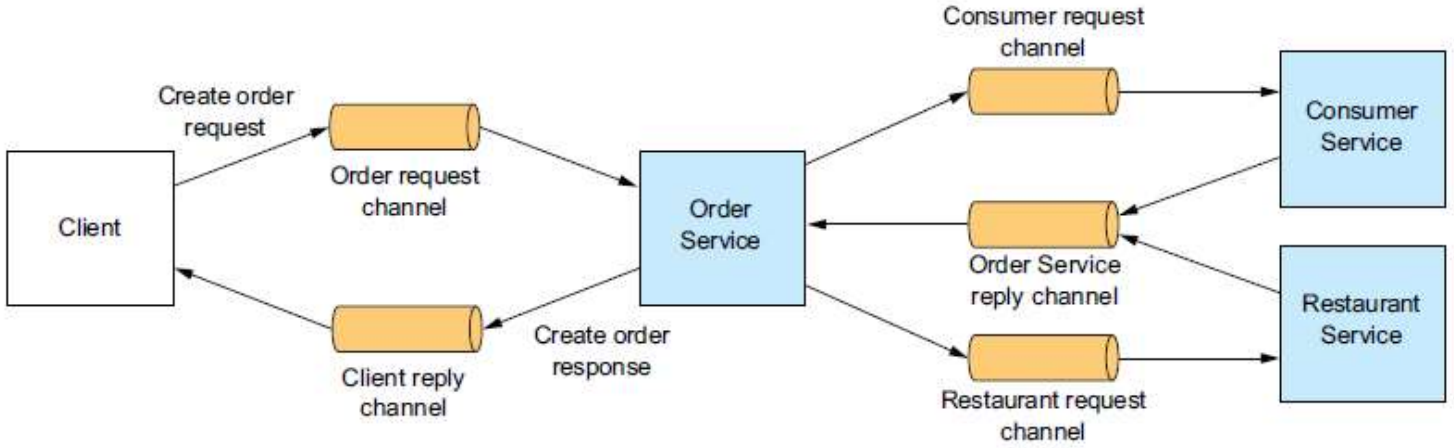
- The client sends a request and waits for a response from the service.

Asynchronous protocol

- The client code or message sender usually doesn't wait for a response.

Asynchronous Communication

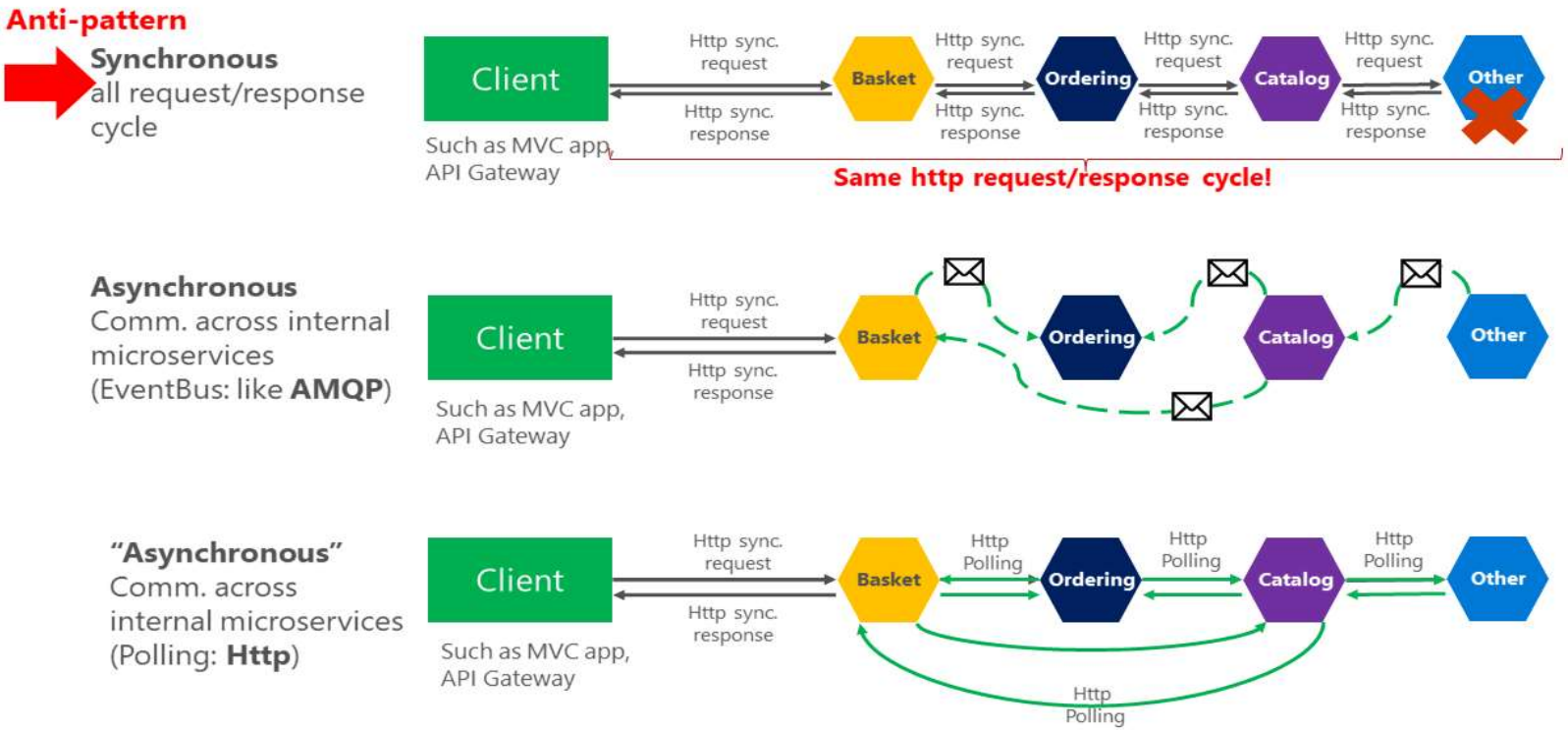
- Services communicating by exchanging messages over messaging channels.



Example

innovate achieve lead

Sync Vs Async communication across microservices



Message Broker

- It is a way of implementing asynchronous communication
- A message broker is an intermediary through which all messages flow.

Examples of popular open source message brokers include the following:

- ActiveMQ
- RabbitMQ
- Apache Kafka

Benefits of Message Broker

- *Loose coupling*
- *Message buffering*
- *Explicit interprocess communication*
- *Resiliency*

Drawbacks of Message Broker

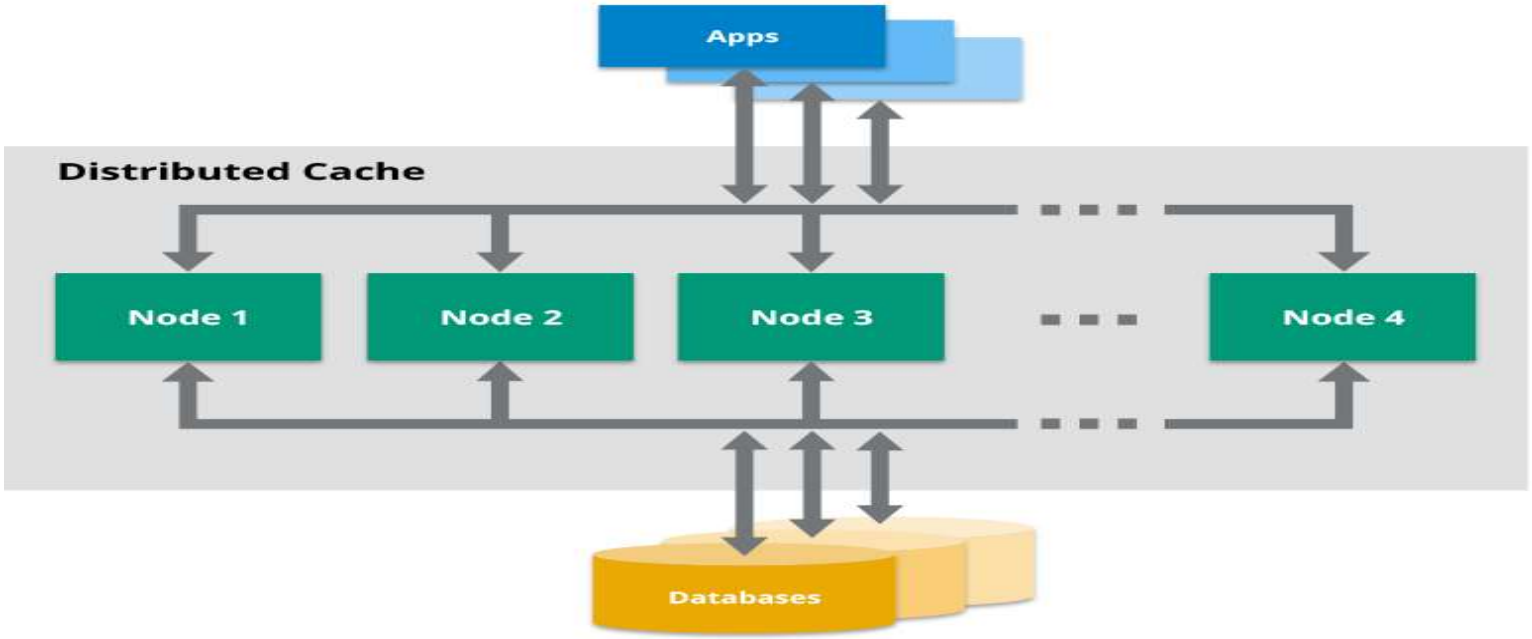
- *Potential performance bottleneck*
- *Potential single point of failure*
- *Additional operational complexity*

What is Caching?

- In computing, a cache is a high-speed data storage layer which stores a subset of data, typically transient in nature, so that future requests for that data are served up faster than is possible by accessing the data's primary storage location

Distributed Caches

- A distributed cache may span multiple servers so that it can grow in size and in transactional capacity.
- It is mainly used to store application data residing in database and web session data.



Advantages of Distributed Cache

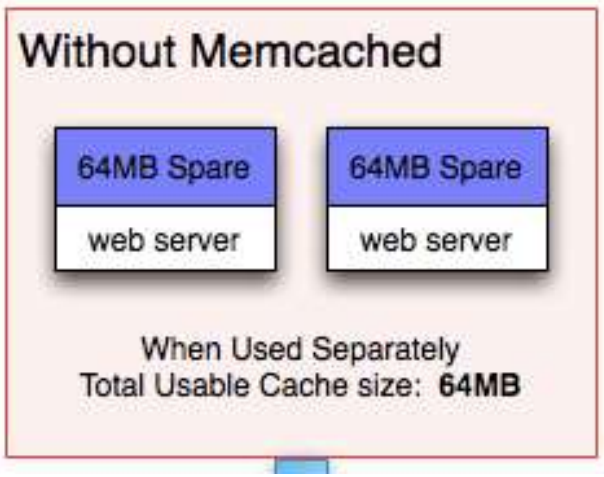
When cached data is distributed, the data:

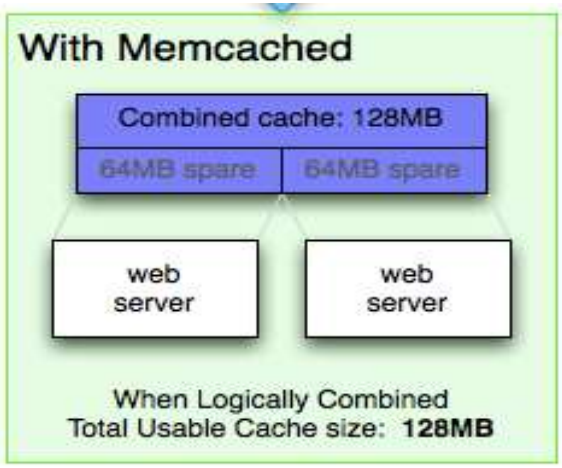
- Is *coherent* (consistent) across requests to multiple servers.
- Survives server restarts and app deployments.
- Doesn't use local memory.

Use Cases for a Distributed Cache

- Application acceleration
- Storing web session data
- Extreme scaling
- Reducing the impact of interruptions

Example: Memchached





Global Caches

- A global cache is a repository for data that you want to reuse.

- The cache facilitates sharing of data across processes (both in the same integration node, and across integration nodes) and eliminates the need for an alternative solution, such as a database.

Google Global Cache (GGC)

- It allows ISPs to serve certain Google content from within their own networks.
- This eases congestion within your network, and reduces the amount on traffic on your peering and transit links.

GGC features

- Transparent to users
- Reduced external traffic
- Robust
- Easy to set up

Scalability features in the Cloud

Horizontal and vertical scaling

- Vertical Scaling is defined as increasing a single machine's capacity with the rising resources in the same logical server or unit. .
- Horizontal Scaling is an approach to enhance the performance of the server node by adding new instances of the server to the existing servers to distribute the workload equally

Image: Google

Comparison

Vertical Vs Horizontal Scaling	Vertical scaling	Horizontal Scaling
Data	Data is executed on a single node	Data is partitioned and executed on multiple nodes
Data Management	Easy to manage – share data reference	Complex task as there is no shared address space
Downtime	Downtime while upgrading the machine	No downtime
Upper limit	Limited by machine specifications	Not limited by machine specifications
Cost	Lower licensing fee	Higher licensing fee

Image: Google

Auto-scaling

- Scaling monitors your applications and automatically adjusts capacity to maintain steady, predictable performance at the lowest possible cost.

- AWS Auto Scaling makes scaling simple with recommendations that allow you to optimize performance, costs, or balance between them

What are the Benefits of Cloud Scaling?

- Fast and Easy
- Cost efficiency
- Optimized Performance
- Capacity

Load Balancing

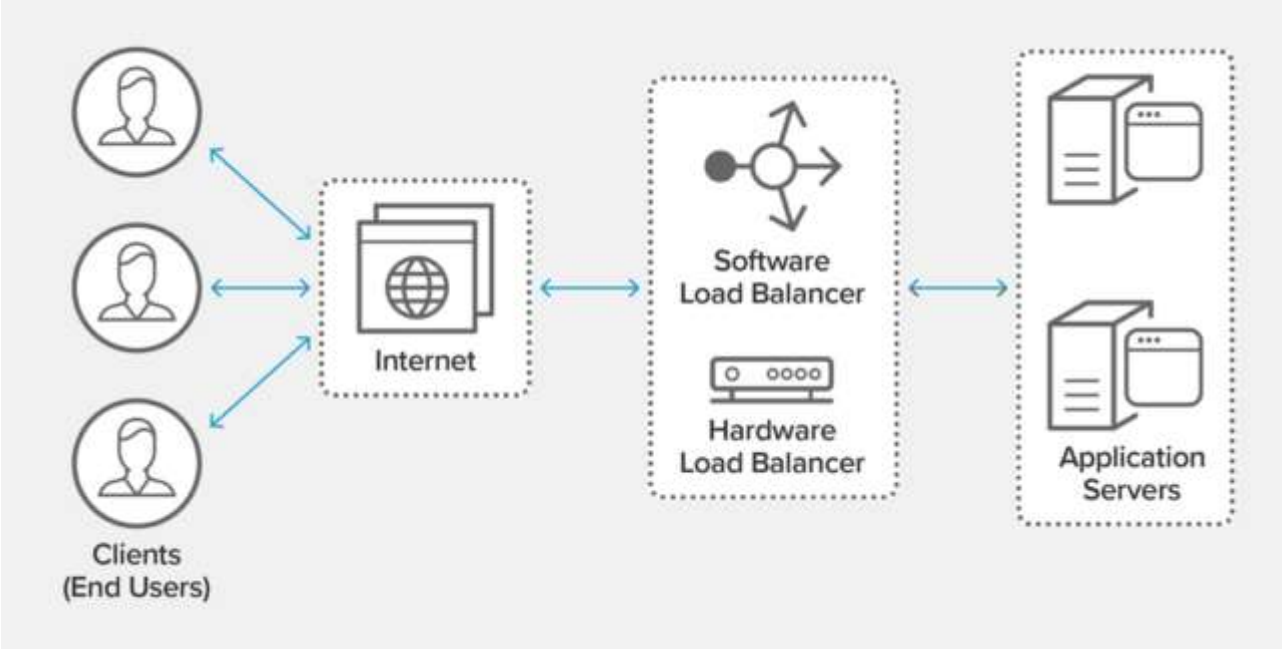
- **Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a *server farm* or *server pool*.

- If a single server goes down, the load balancer redirects traffic to the remaining online servers.

- When a new server is added to the server group, the load balancer automatically starts to send requests to it.

Functions of Load Balancer

- Distributes client requests or network load efficiently across multiple servers
- Ensures high availability and reliability by sending requests only to servers that are online
- Provides the flexibility to add or subtract servers as demand dictates



Benefits of Load Balancing

- Reduced downtime
- Scalable
- Redundancy
- Flexibility
- Efficiency

What is virtualization?

- Virtualization uses software to create an abstraction layer over computer hardware that allows the hardware elements of a single computer—processors, memory, storage and more—to be divided into multiple virtual computers, commonly called virtual machines (VMs).

Benefits of virtualization

- Resource efficiency
- Easier management
- Minimal downtime
- Faster provisioning

Types of virtualization

- Desktop virtualization
- Network virtualization
- Storage virtualization
- Data virtualization
- Application virtualization
- Data center virtualization
- CPU virtualization
- GPU virtualization
- Linux virtualization
- Cloud virtualization

What is a serverless architecture?

- A serverless architecture is a way to build and run applications and services without having to manage infrastructure.

- By using a serverless architecture, the developers can focus on their core product instead of worrying about managing and operating servers or runtimes, either in the cloud or on-premises.

Models under serverless

- Backend-as-a-Service(BaaS)
- Function-as-a-Service(FaaS)

When to use Serverless Computing?

Example: Simform built a responsive single-page application integrated with AWS Lambda to automate confirmation of bookings and manage online transactions.

Some Use cases where we can use serverless:

- Build high-latency, real-time applications like multimedia apps, to execute automatic allocation of memory and complex data processing
- To get precise device status and process smart device applications using the IoT.
- Support service integrations for multi-language to meet the demands of modern software.

Advantages of Serverless Computing

- Your developers can now focus on writing codes
- Since serverless architecture executes the business logic/code as functions, you no longer need to manage infrastructures manually.
- Failures do not impact the entire application
- You can deploy apps faster and become more flexible in releases.

Limitations of Serverless Computing

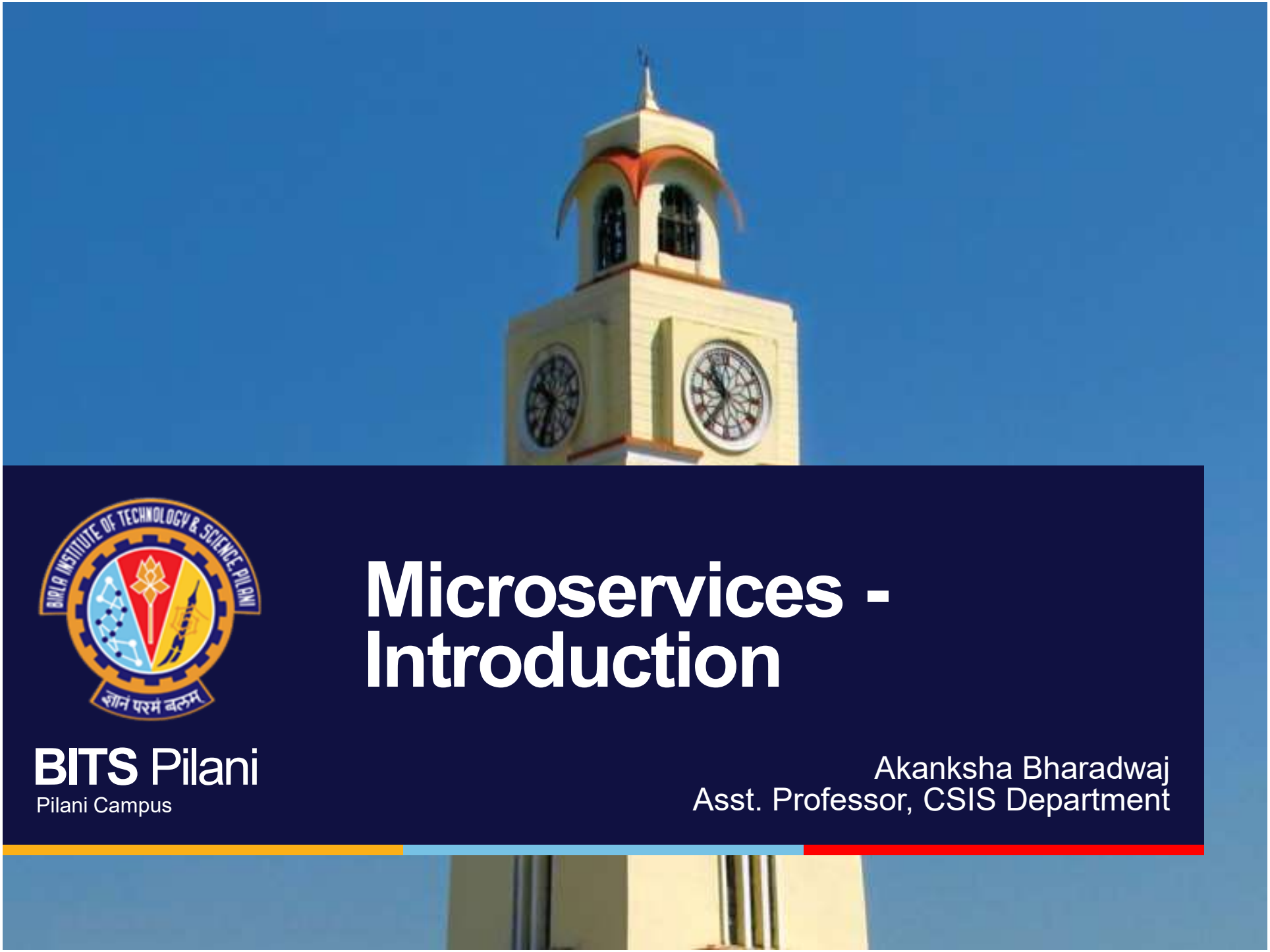
- Long-running workloads could prove to be more costly on serverless than dedicated servers
- You will be dependent on your providers for debugging and monitoring tools
- You have limited control over the platform's architecture and availability.

Best Practices for Achieving Scalability

- Breaking the Monolithic Application
- Distributed Caching
- CDN
- Asynchronous Communication
- Be Stateless

References

- https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-replica-lifecycle
- https://www.citrix.com/en-in/solutions/application-delivery-controller/load-balancing/what-is-load-balancing.html#:~:text=Load%20balancing%20is%20defined%20as,server%20capable%20of%20fulfilling%20them.
- https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs#:~:text=CQRS%20stands%20for%20Command%20and,performance%2C%20scalability%2C%20and%20security.
- https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture
- https://memcached.org/
- https://www.ibm.com/docs/en/integration-bus/10.0?topic=caching-data-overview
- https://support.google.com/interconnect/answer/9058809?hl=en
- https://aws.amazon.com/
- https://www.nginx.com/resources/glossary/load-balancing/
- https://www.ibm.com/in-en/cloud/learn/virtualization-a-complete-guide



Microservices - Introduction

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 4

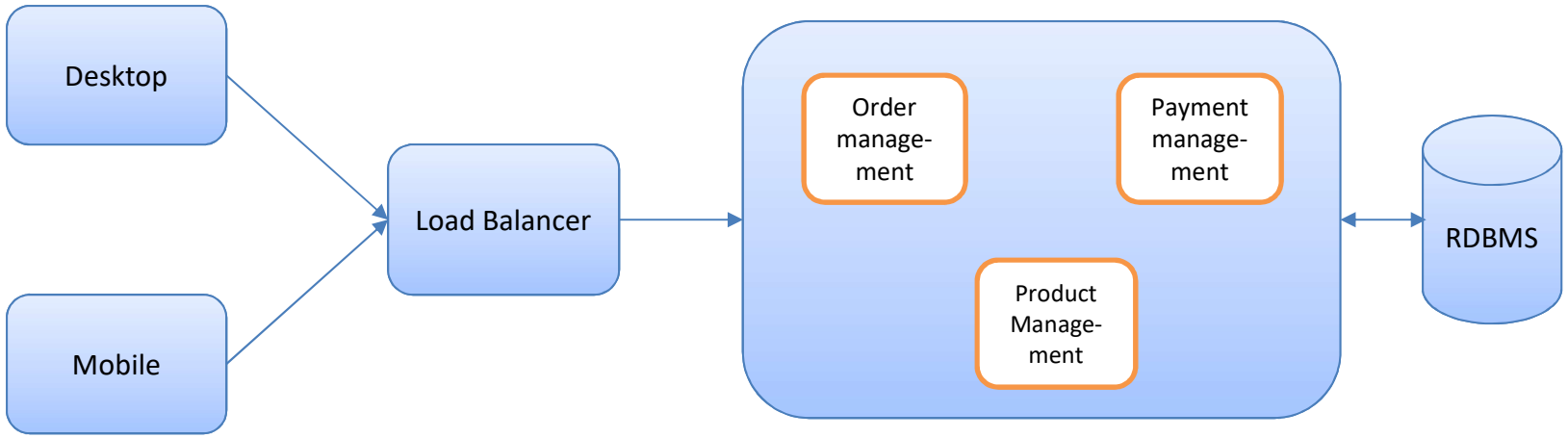
What is Monolithic Architecture?

- Monolith means composed all in one piece.
- They're typically complex applications that encompass several tightly coupled functions.
- When all functionality in a system had to be deployed together, we consider it a **monolith**.

Example Architecture

innovate achieve lead

Online shopping



Disadvantages of Monolith

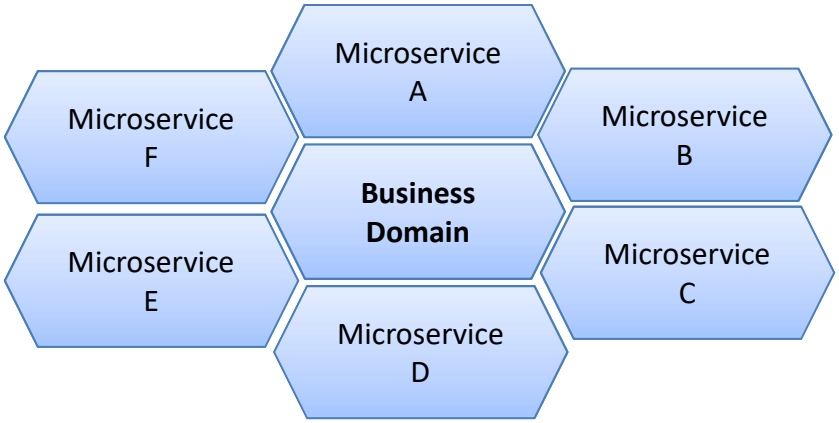
- It becomes too large in size with time and hence, difficult to manage.
- Slow development and deployment
- Adoption of a new technology is very difficult and expensive

Need for Microservices

- Why is there a need to convert a fully functional monolithic application to Microservices ?
- Is the conversion worth the pain and effort?
- Should I be converting all my applications to Microservices?

What is Microservices?

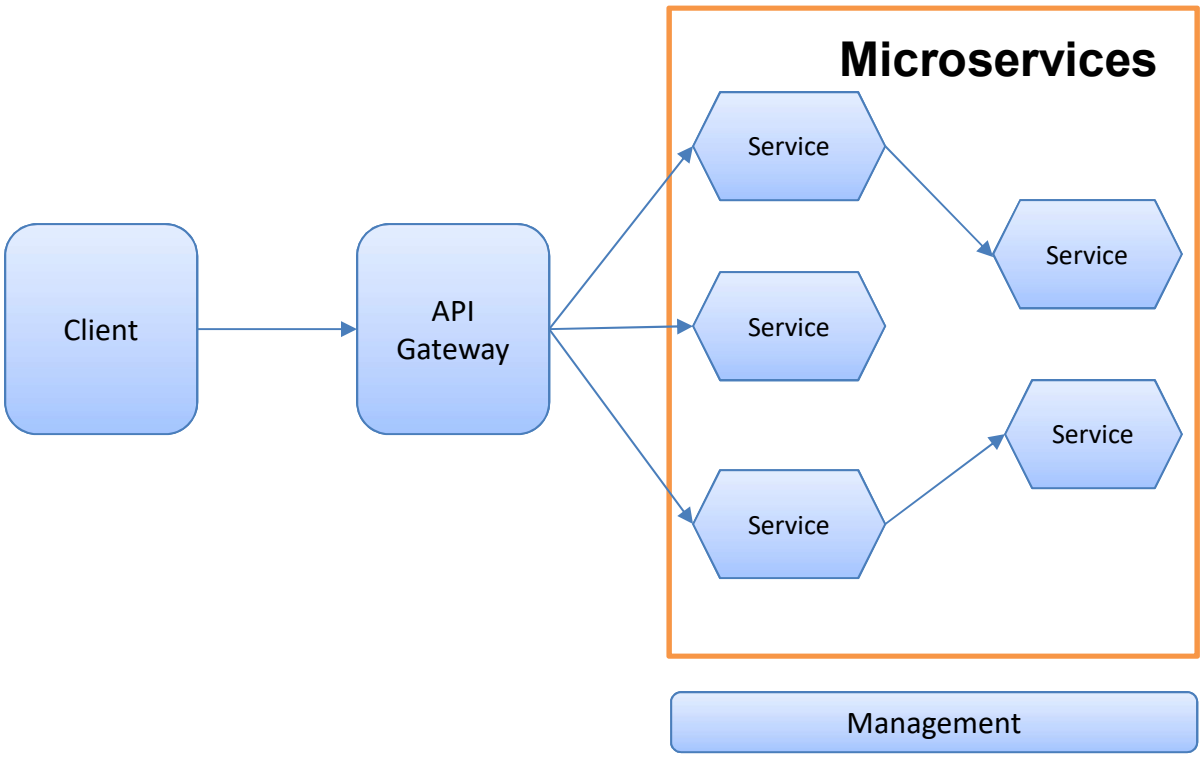
- Microservices are independently deployable services modeled around a business domain.
- They communicate with each other via networks,
- Each microservice can focus on a single business capability



Main characteristics of microservices

- Small, and Focused on Doing One Thing Well
- Autonomous
- Loosely coupled

Example Architecture



SOA

- SOA, or service-oriented architecture, defines a way to make software components reusable via service interfaces.
- These interfaces utilize common communication standards in such a way that they can be rapidly incorporated into new applications without having to perform deep integration each time.

SOA Vs Microservices

- SOA is an enterprise-wide concept.
- Microservices architecture is an application-scoped concept

SOA Vs Microservices

Communication

- In a microservices architecture, each service is developed independently, with its own communication protocol.

- With SOA, each service must share a common communication mechanism called an enterprise service bus

SOA Vs Microservices

Data Duplication

- Providing services in SOA applications get hold of and make changes to data directly at its primary source, which reduces the need to maintain complex data synchronization patterns.

- In microservices applications, each microservice ideally has local access to all the data it needs to ensure its independence from other microservices, and indeed from other applications, even if this means some duplication of data in other systems.

SOA vs Microservices: Which is best for you?

- Both approaches have their advantages, so how can you determine which one will work best for your purposes?
- In general, it depends on how large and diverse your application environment is.

innovate achieve lead

BITS Pilani

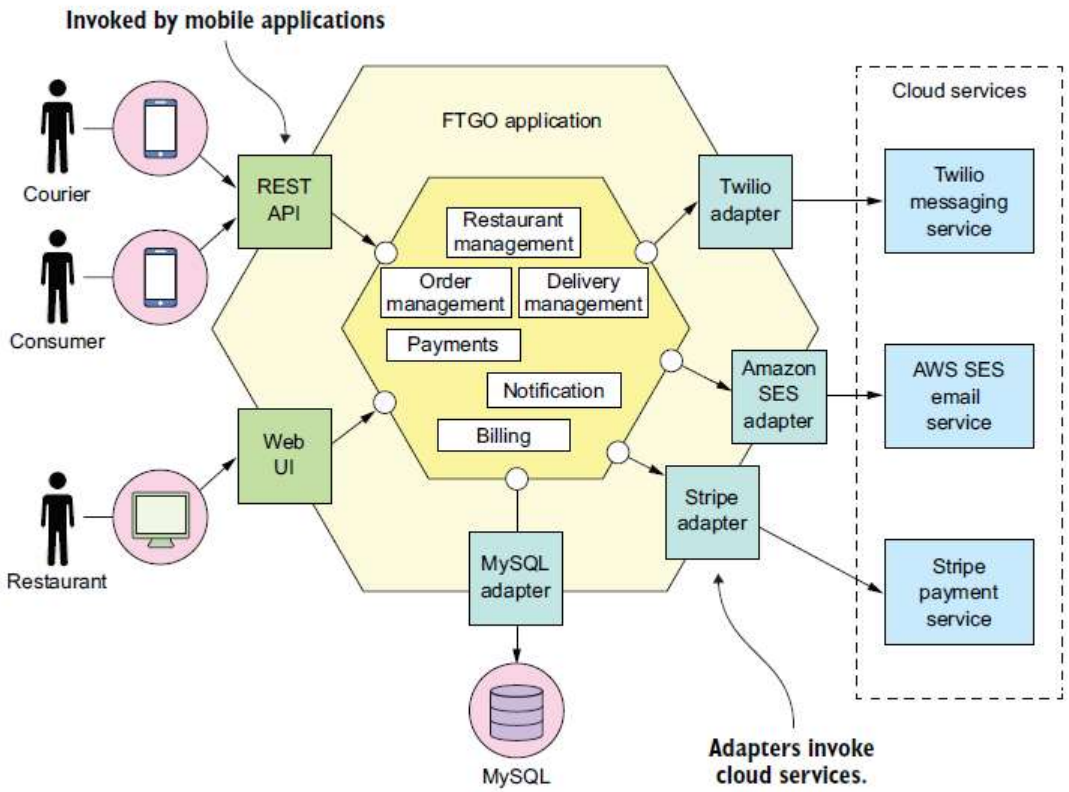
Pilani Campus

Case Study

FTGO Case Study

- Since its launch in late 2005, Food to Go, Inc. (FTGO) had grown by leaps and bounds. Today, it's one of the leading online food delivery companies in the United States.
- The business even plans to expand overseas, although those plans are in jeopardy because of delays in implementing the necessary features.
- At its core, the FTGO application is quite simple. Consumers use the FTGO website or mobile application to place food orders at local restaurants.
- FTGO coordinates a network of couriers who deliver the orders. It's also responsible for paying couriers and restaurants. Restaurants use the FTGO website to edit their menus and manage orders.
- The application uses various web services, including Stripe for payments, Twilio for messaging, and Amazon Simple Email Service (SES) for email.

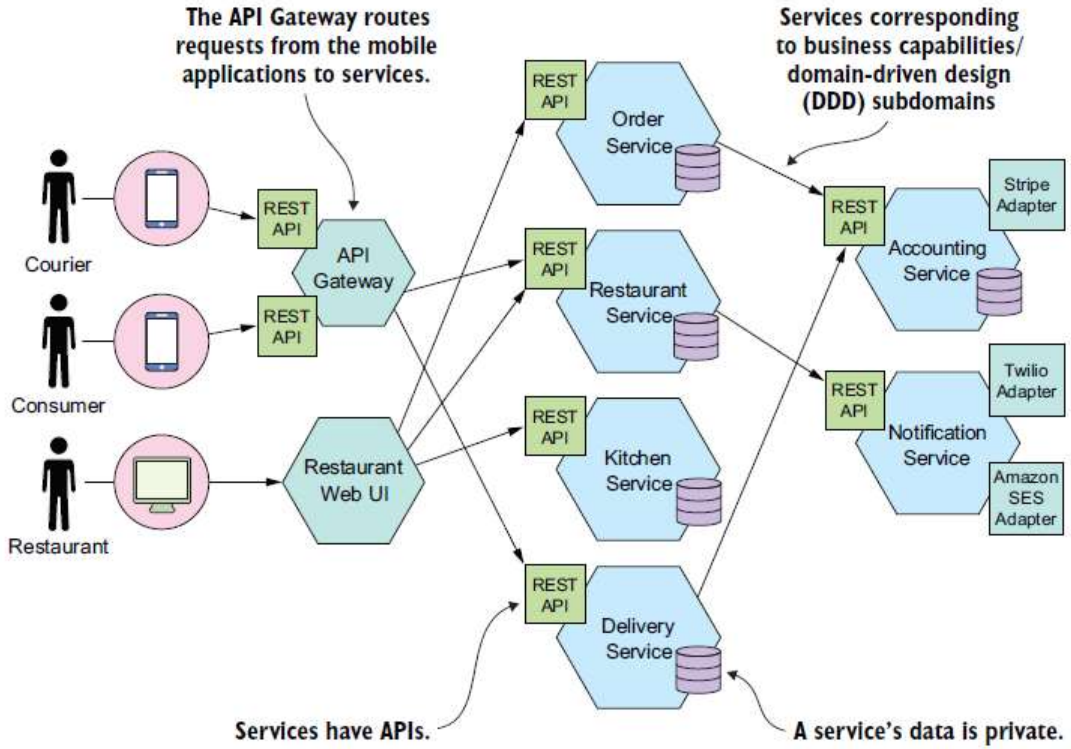
Old Architecture of the FTGO application



Problems faced in old FTGO architecture

- Complexity
- Slow development
- Scaling is difficult

Microservices Architecture for FTGO



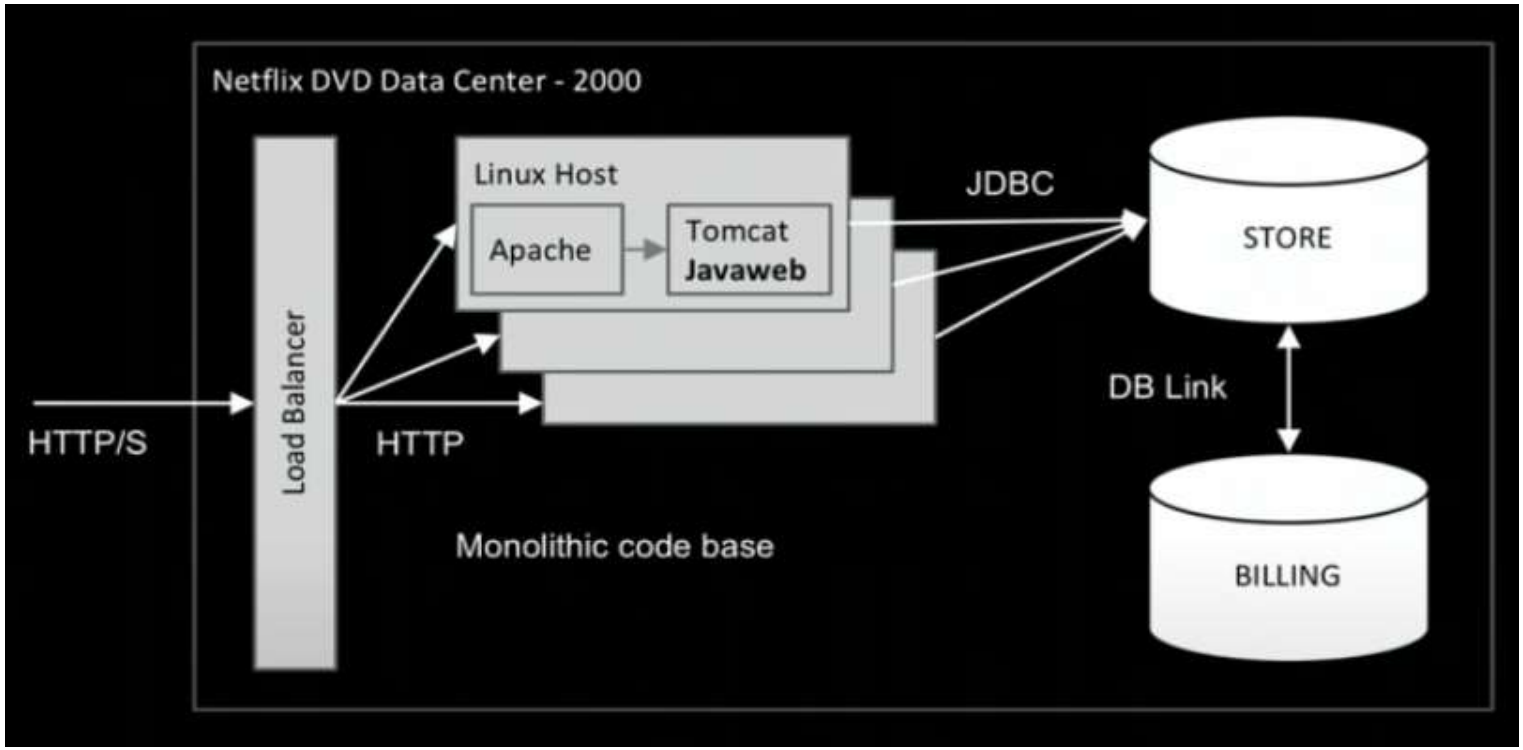
Netflix Case Study

- Netflix launched in 1998. At first they rented DVDs through the US Postal Service. But Netflix saw the future was on-demand streaming video
- In 2007 Netflix introduced their streaming video-on-demand service
- Why are we considering this case study?

How Netflix worked earlier?

innovate achieve lead

Netflix Architecture earlier



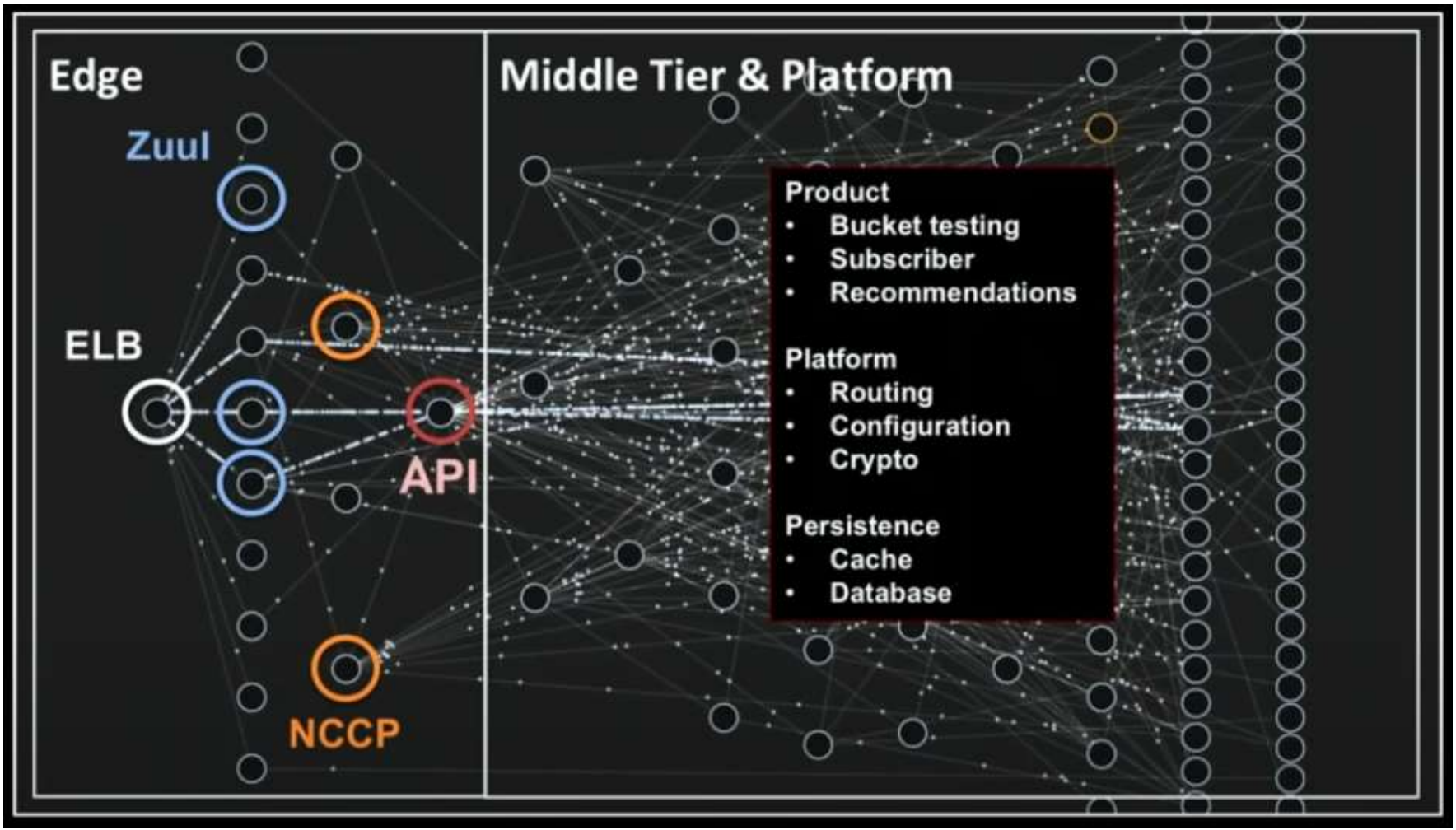
Challenges in previous architecture

- Monolithic Code base
- Monolithic Database
- Tightly coupled Architecture

What they need in new Architecture?

- Modularity and encapsulation
- Scalability
- Virtualization and Elasticity

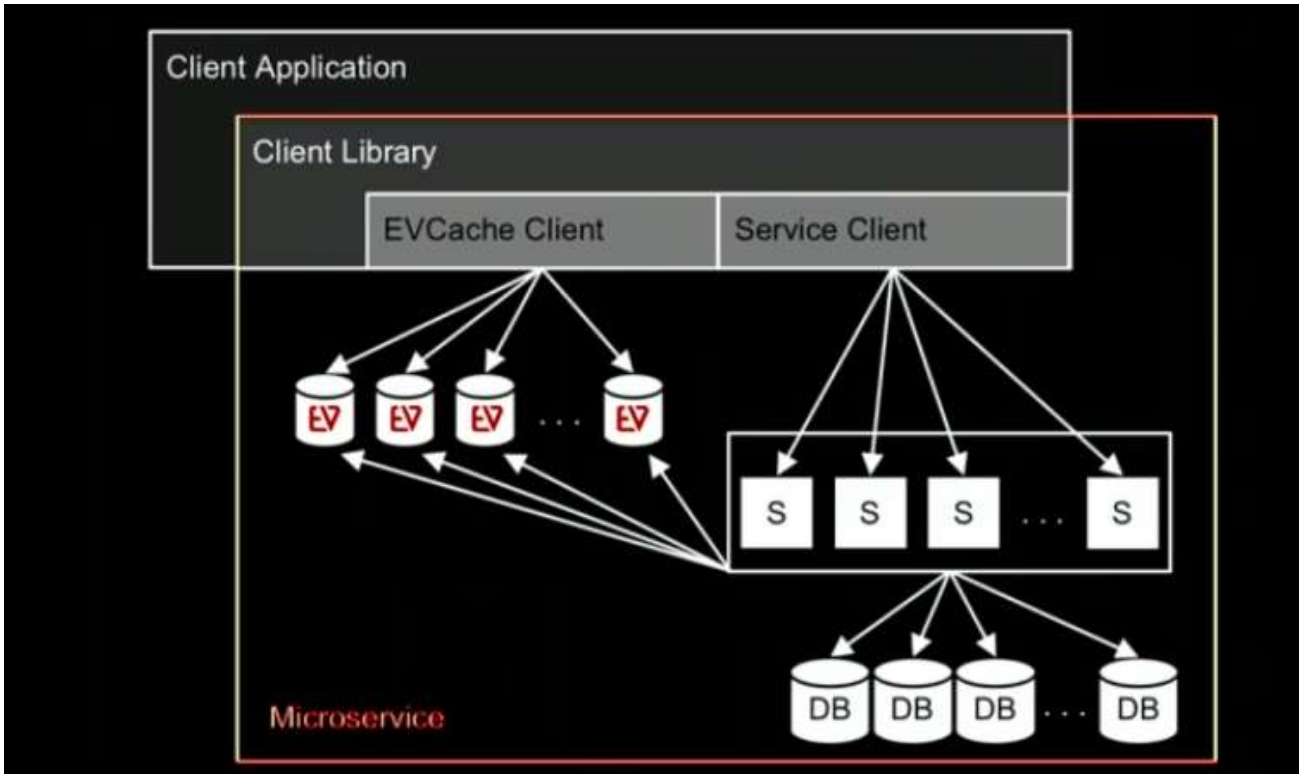
Updated Netflix architecture



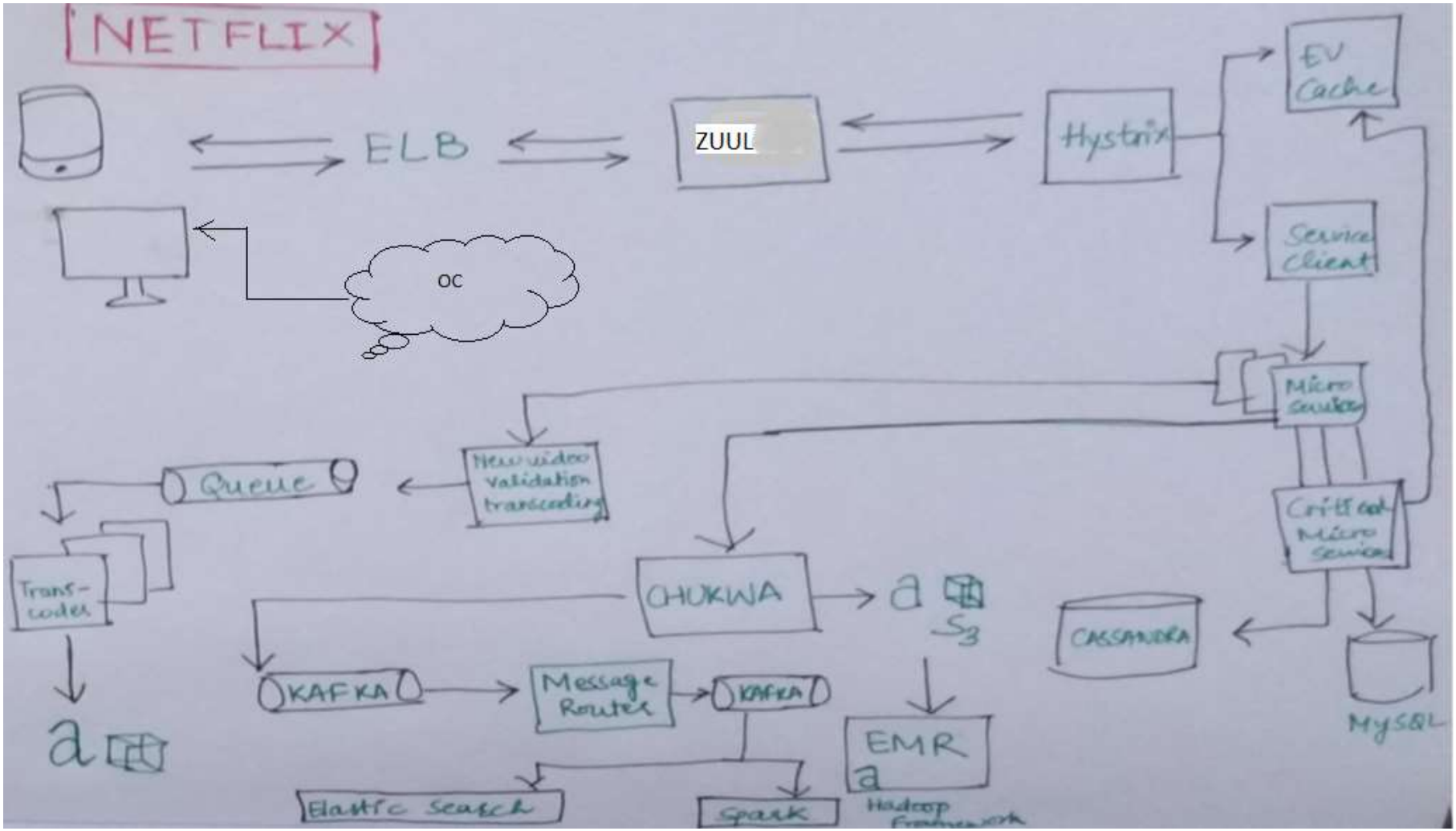
Netflix Approach

innovate achieve lead

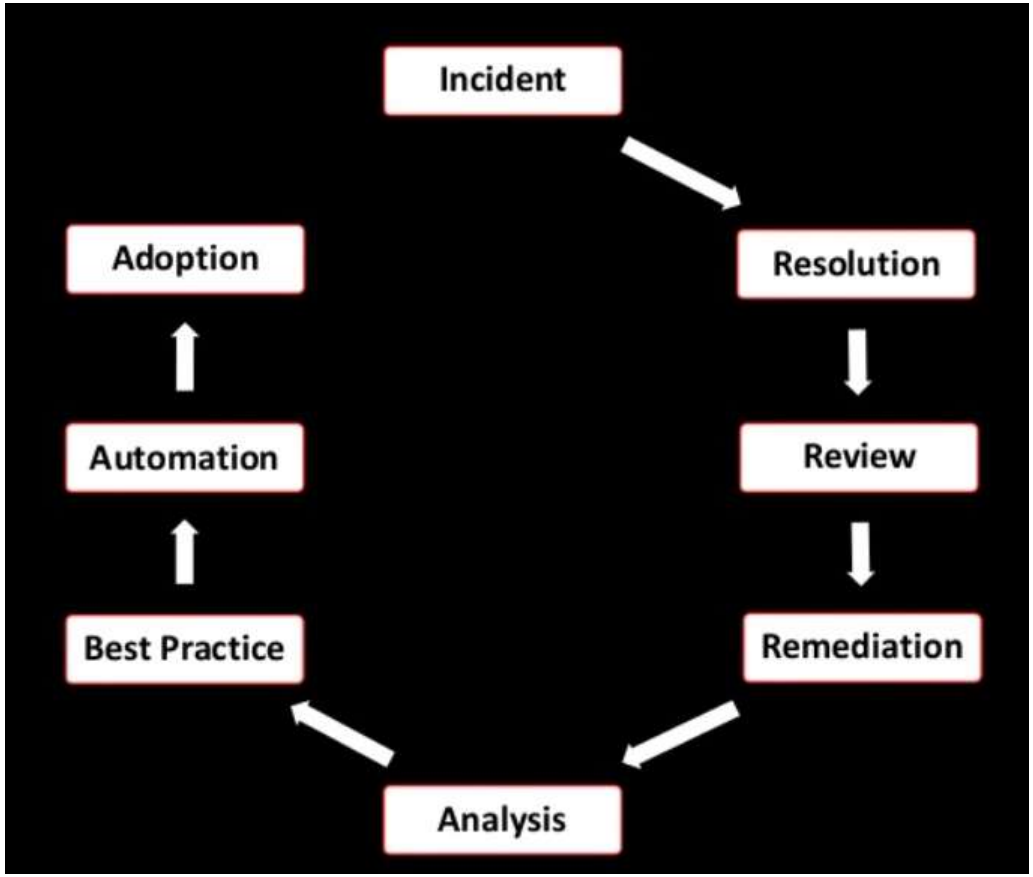
Microservice at Netflix



Latest Netflix Architecture



Netflix follows continuous learning



Self Study

- Uber case study – Read about Domain Oriented Microservices Architecture

- Link: https://eng.uber.com/

References

- Research Paper: Challenges When Moving from Monolith to Microservice Architecture, Miika Kalske, Niko M¨akitalo, and Tommi Mikkonen
- Book: Monolith to Microservices by Sam Newman
- Book: Building Microservices by Sam Newman
- Book: Microservices Vs Service Oriented Architecture by Mark Richards
- Book: Microservices Patterns by Chris Richardson
- Link: https://www.ibm.com/cloud/blog/soa-vs-microservices
- Link: https://www.slideshare.net/adrianco
- Talks about Netflix by Josh Evans

Microservices Contd.

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS DEpartment

innovate achieve lead

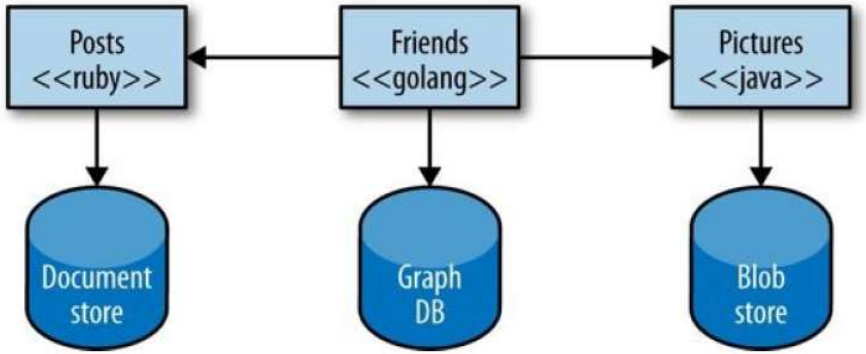
BITS Pilani

Pilani Campus

SE ZG583, Scalable Services
Lecture No. 5

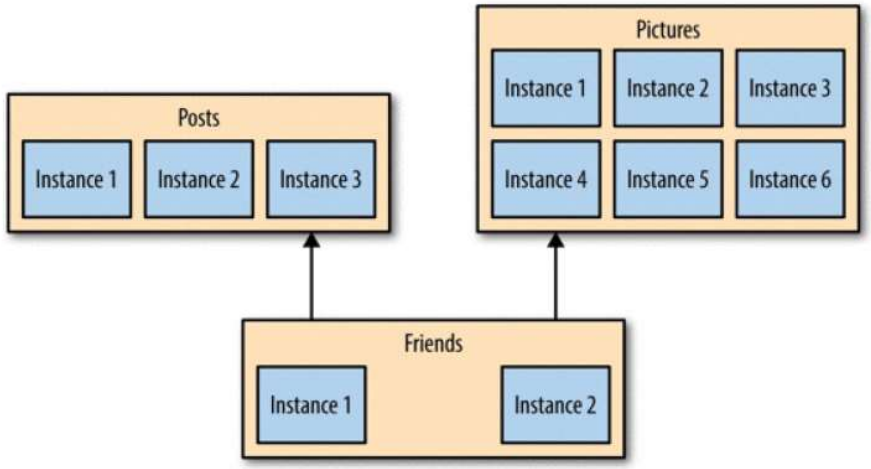
Advantages of Microservices

- Technology Heterogeneity



Advantages contd.

- Resilience
- Ease of Deployment
- Scaling



Advantages contd.

- Organizational Alignment
- Reusability
- Agility

Microservices is not a silver bullet

- Complexity
- Finding the right services is challenging
- Development and testing
- Network congestion and latency
- Data integrity
- Careful Coordination
- Versioning

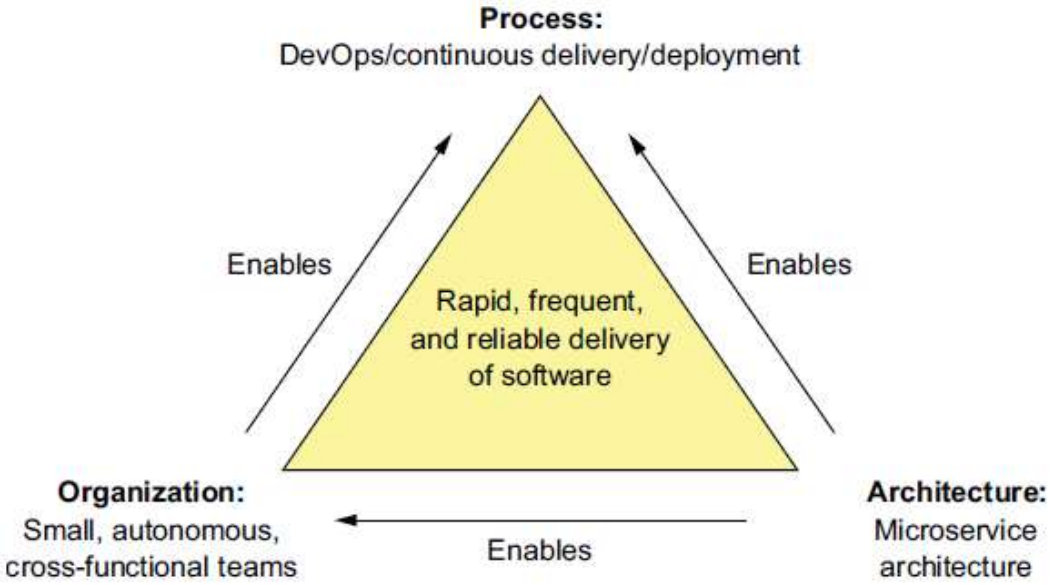
innovate achieve lead

BITS Pilani

Pilani Campus

Process and Organization

Introduction



Software development and delivery organization

- Success means that the engineering team will grow
- The solution is to refactor a large single team into a team of teams.
- The velocity of the team of teams is significantly higher than that of a single large team.
- Moreover, it's very clear who to contact when a service isn't meeting its SLA.

Software development and delivery process

- If you want to develop an application with the microservice architecture, it's essential that you adopt agile development and deployment practices
- Practice continuous delivery/deployment, which is a part of DevOps.
- A key characteristic of continuous delivery is that software is always releasable

The human side of adopting microservices

- Ultimately, it changes the working environment of people and thus impact them emotionally

Three stage transition model

- Ending, Losing, and Letting Go
- The Neutral Zone
- The New Beginning

innovate achieve lead

BITS Pilani

Pilani Campus

Microservices Design Principles

Single Responsibility

- Sometimes it's important to maintain data consistency by putting functionality into a single microservice.
- Each microservice implements only one business responsibility from the bounded domain context.
- The rule of thumb is

"Gather together those things that change for the same reason, and separate those things that change for different reasons." — Robert C. Martin

Abstraction and Information Hiding

- A service should only be consumed through a standardized API and should not expose its internal implementation details to its consumers

Loose coupling

- Dependencies between services and their consumers are minimized with the application of the principle of loose coupling

Fault tolerance

- Each service is necessarily fault tolerant so that failures on the side of its collaborating services will have minimal impact

Discoverability

- The aim of discoverability is to communicate a clear understanding of the business purpose and technical interface of the microservice.

Reusability

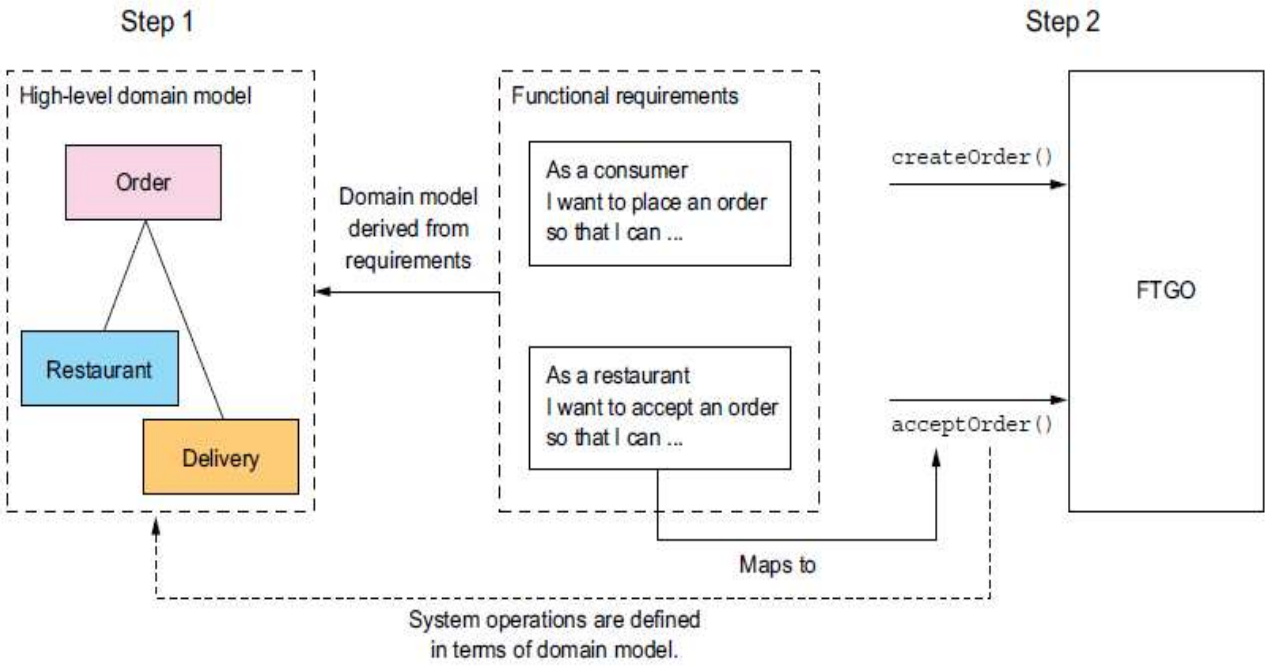
- Reuse continues to be a principle of microservice design. However, the scope of reuse has been reduced to specific domains within the business.

Steps for defining an application's microservice architecture

Step 1: Identify system operations

Step 2: Identify services

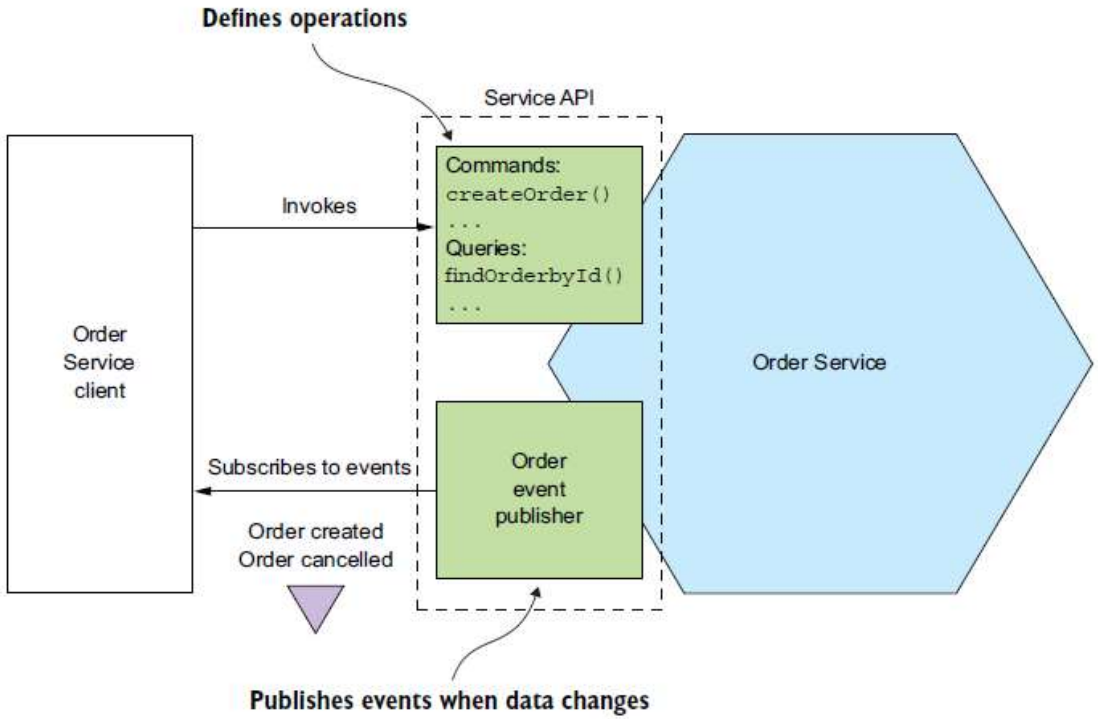
Step 3: Define service APIs and collaborations



Important concepts

What is a Service?

- The first and most important aspect of the architecture is, the definition of the services.



The role of Shared Libraries

- On the surface, it looks like a good way to reduce code duplication in your services.
- But you need to ensure that you don’t accidentally introduce coupling between your services.
- You should strive to use libraries for functionality that’s unlikely to change.

Size of a Service

- size isn’t a useful metric.
- A much better goal is to define a well-designed service
- Build a set of small, loosely coupled services.

innovate achieve lead

BITS Pilani

Pilani Campus

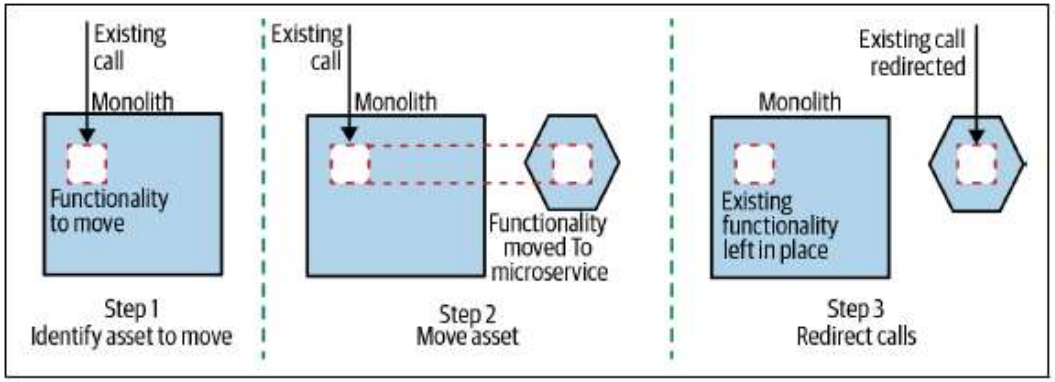
Monolith to Microservices

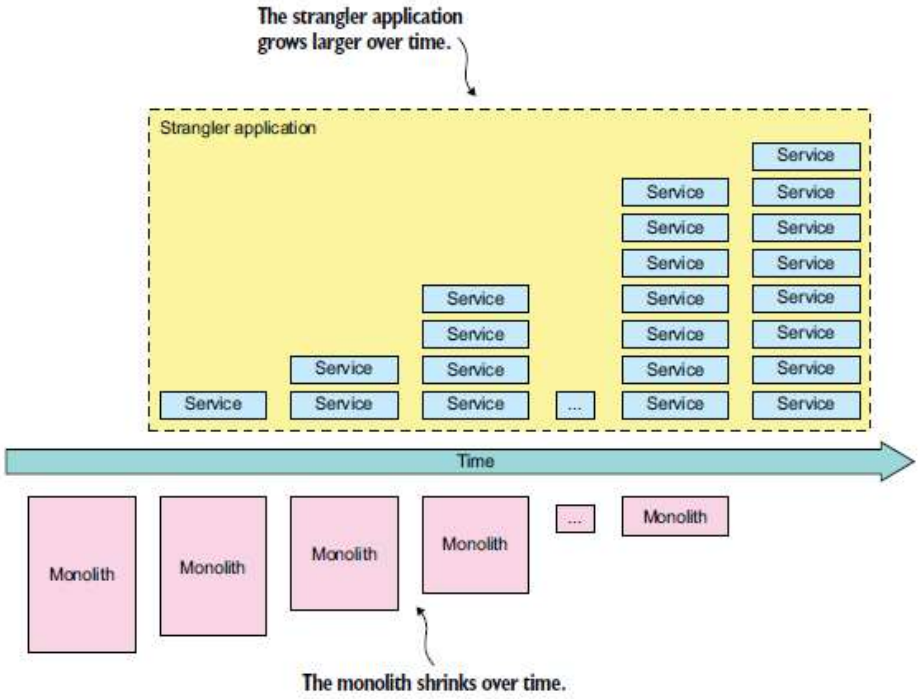
Rebuild From Scratch

- One of the biggest challenges for us was having a good understanding of the legacy system.
- Can not use the system until complete
- Longer duration required

Strangler Pattern

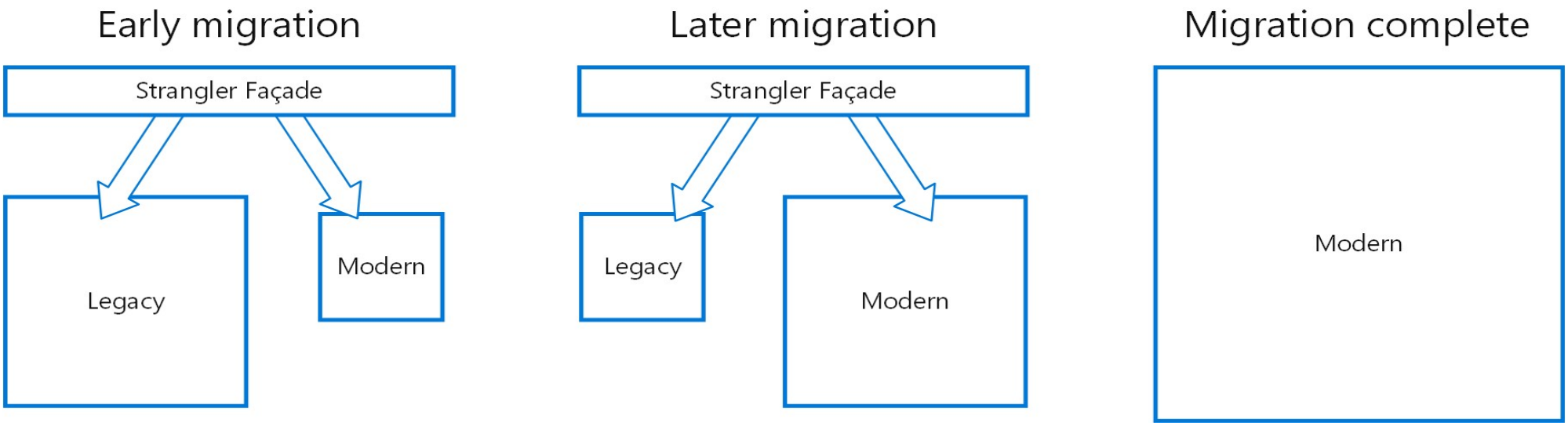
- The Strangler Pattern is a popular design pattern to incrementally transform your monolithic application into microservices by replacing a particular functionality with a new service.
- Any new feature to be added is done as part of the new service





Strangler Pattern

Steps involved in transition



Strangler Pattern: Issues

- What component to start with?
- How to handle services and data stores that are potentially used by both new and legacy systems?
- Migration

When not to use Strangler Pattern?

- When requests to the back-end system cannot be intercepted.
- For smaller systems where the complexity of wholesale replacement is low.

innovate achieve lead

BITS Pilani

Pilani Campus

Decomposition based patterns

Decompose by business capability pattern

- Business capability is something that a business does in order to generate value.
- **Example**: The capabilities of an online store include Order management, Inventory management, Shipping, and so on.

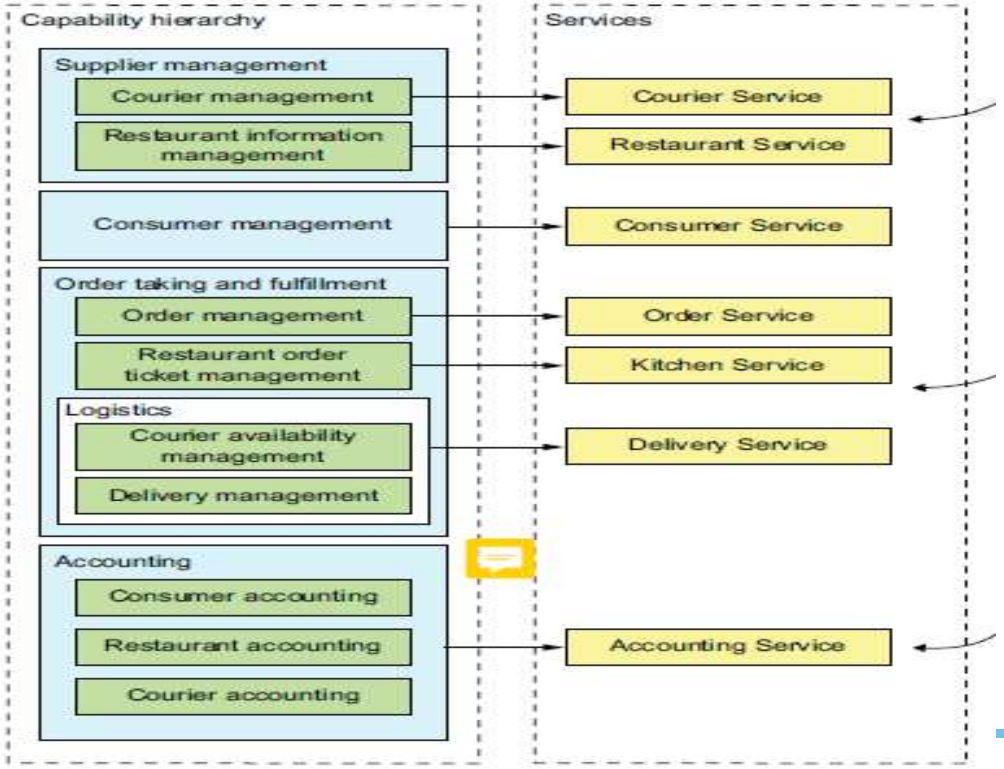
Identifying Business Capabilities

Business capabilities for FTGO include the following:

- Supplier management
- Consumer management
- Order taking and fulfilment
- Accounting

From business capabilities to services

- Once you've identified the business capabilities, you then define a service for each capability or group of related capabilities



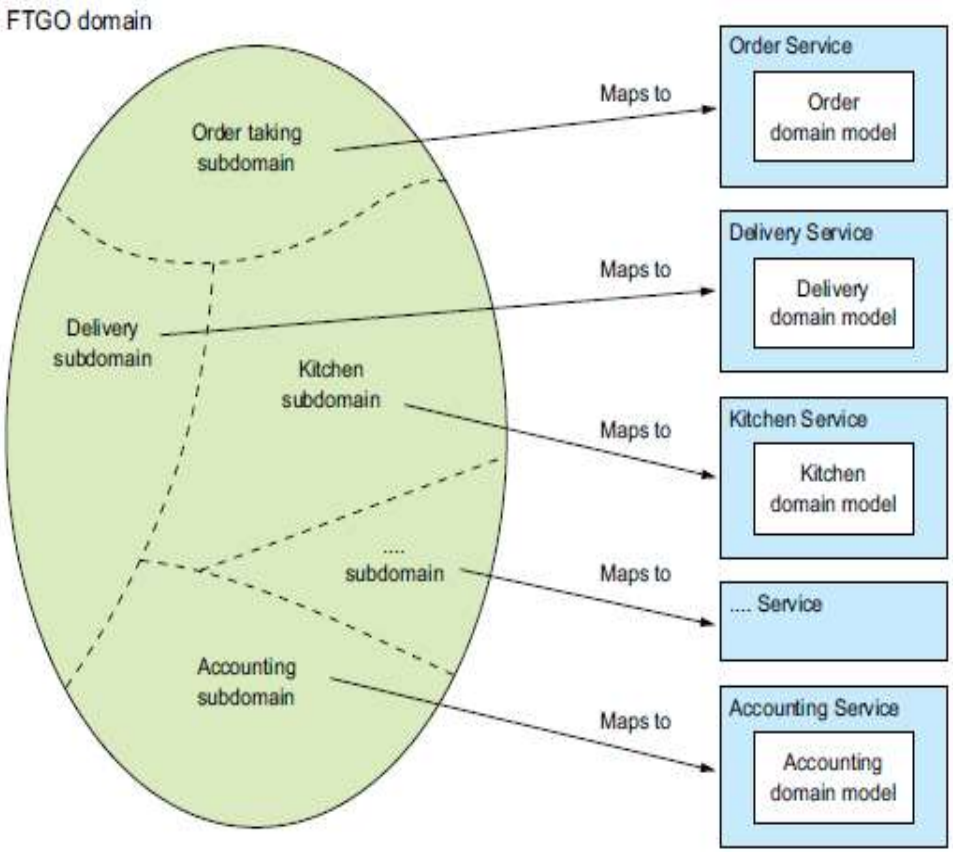
Decompose by sub-domain pattern

- DDD is an approach for building complex software applications centered on the development of an object-oriented, domain model.
- DDD has two concepts that are incredibly useful when applying the microservice architecture: subdomains and bounded contexts.

From Subdomains to Services

- DDD defines a separate domain model for each subdomain
- The examples of subdomains in FTGO include order taking, order management, restaurant order management, delivery, and financials.
- DDD calls the scope of a domain model a “bounded context.”
- When using the microservice architecture, each bounded context is a service or possibly a set of services.

Decompose by sub-domain pattern



Decomposition guidelines

- Single Responsibility Principle
- Common Closure Principle

References

- Book: Microservices Patterns by Chris Richardson
- Book: Building Microservices by Sam Newman
- Book: Monolith to Microservices by Sam Newman
- Link: https://docs.microsoft.com/en-us/azure/architecture/microservices/design/
- Link: https://docs.microsoft.com/en-us/azure/architecture/microservices/design/patterns

Microservices Contd.

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani

Pilani Campus

SE ZG583, Scalable Services
Lecture No. 6

Database related patterns for Microservices

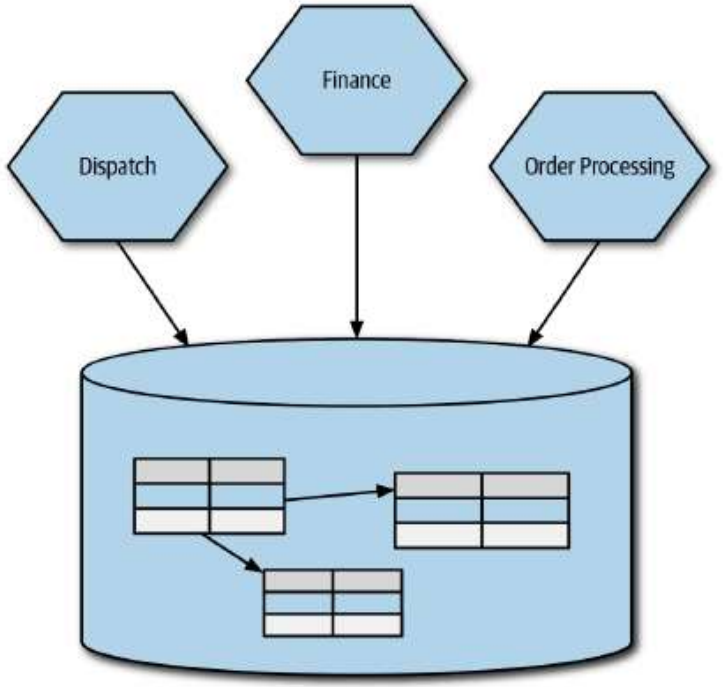
Shared Database pattern

Problems:

- Shared Vs hidden data
- Control on data

When to use:

- Read-only static reference data
- During transition

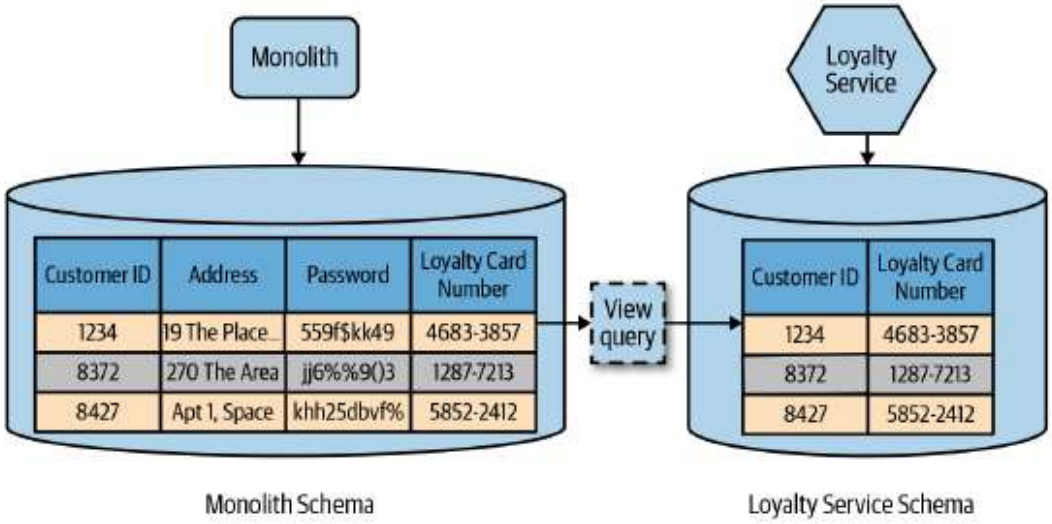


Database View Pattern

- For a single source of data for multiple services, a view can be used to mitigate the concerns regarding coupling

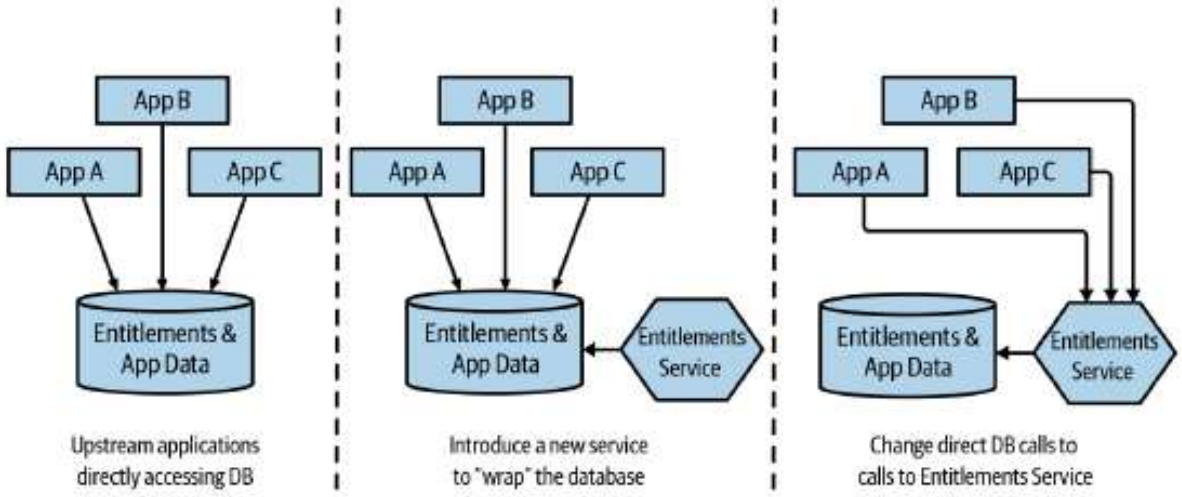
Limitations:

- Views are read-only.



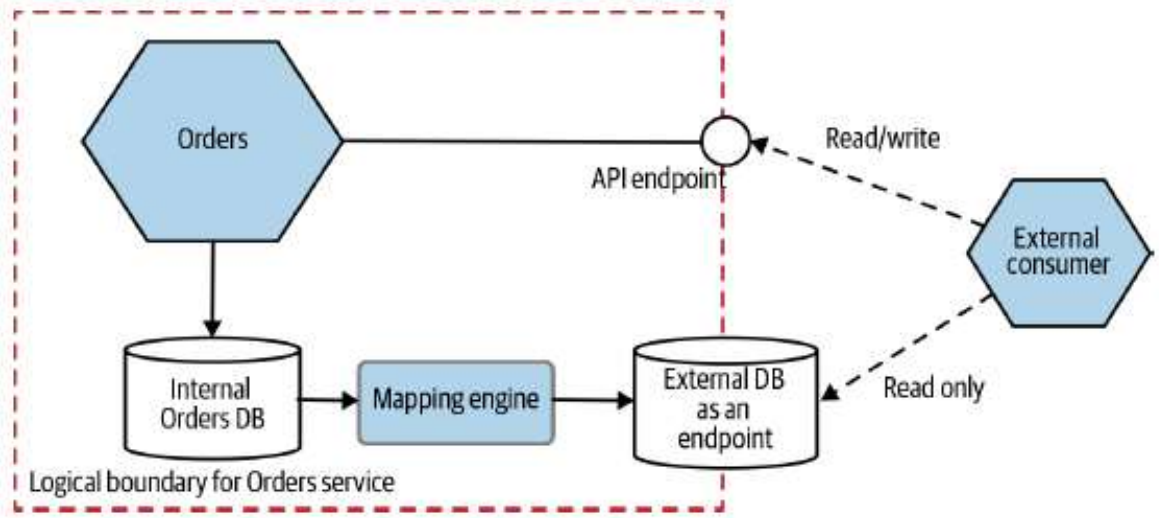
Database Wrapping Service Pattern

- Here we hide the database behind a service that acts as a thin wrapper



Database-as-a-Service Pattern

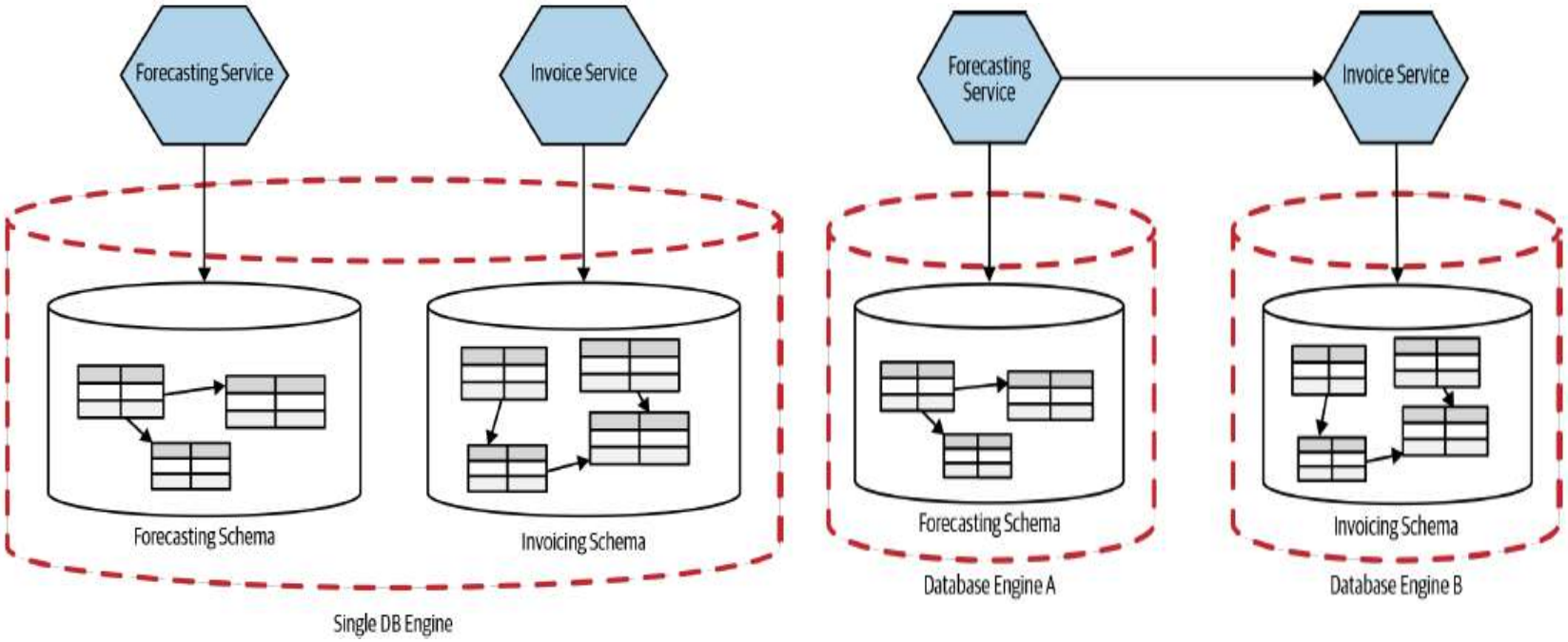
Can be used when clients just need a database to query.



Splitting Apart the Database

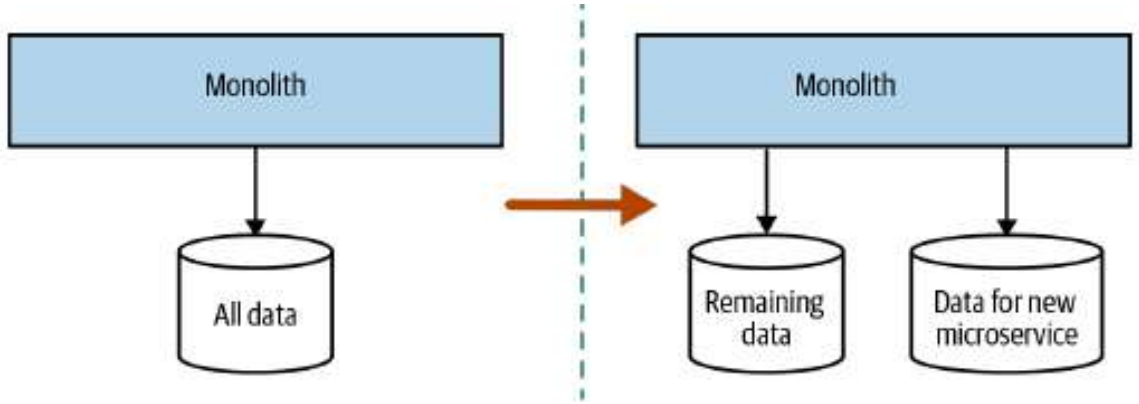
innovate achieve lead

Physical Versus Logical Database Separation



Transition from monolith to microservices

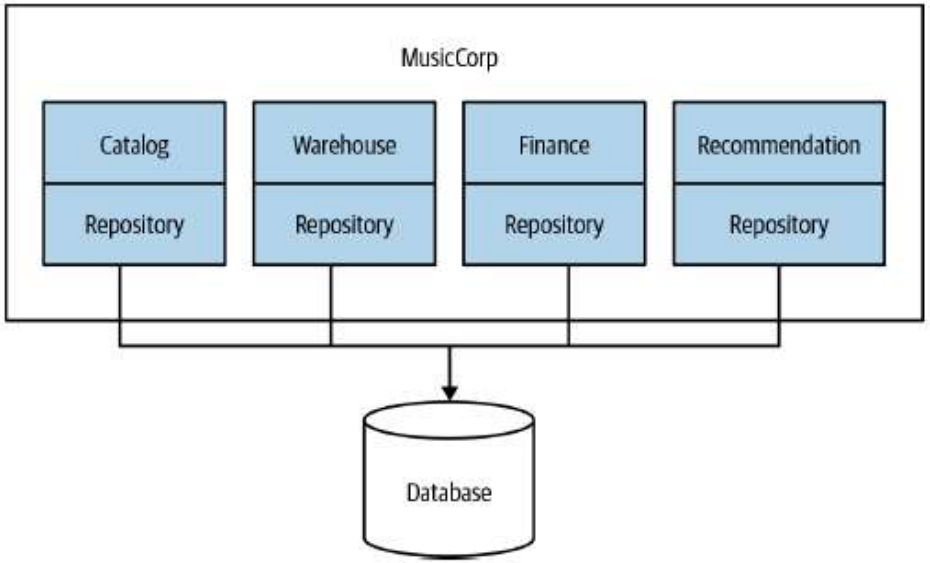
Split the Database First



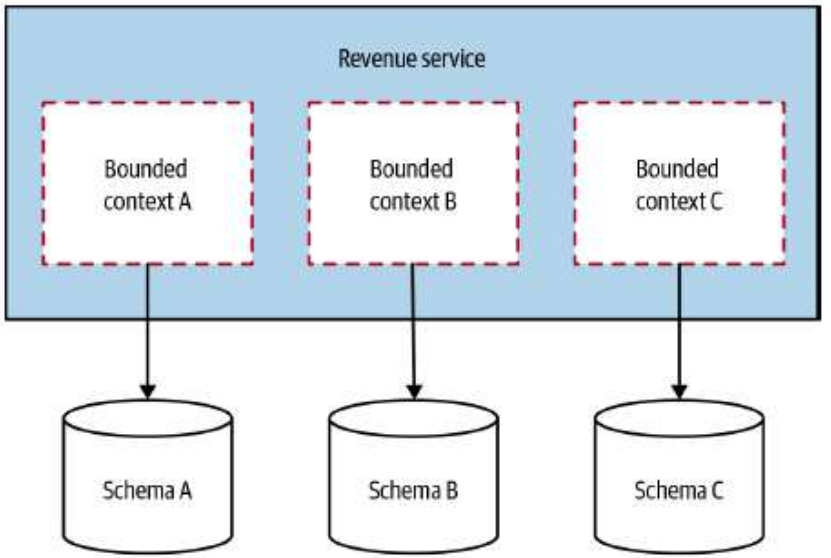
Split the Database First

innovate achieve lead

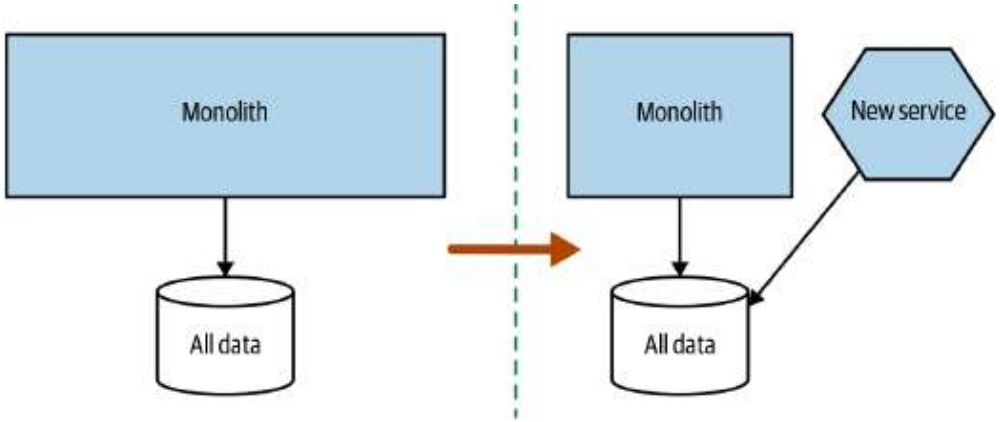
Pattern: Repository per bounded context



Pattern: Database per bounded context



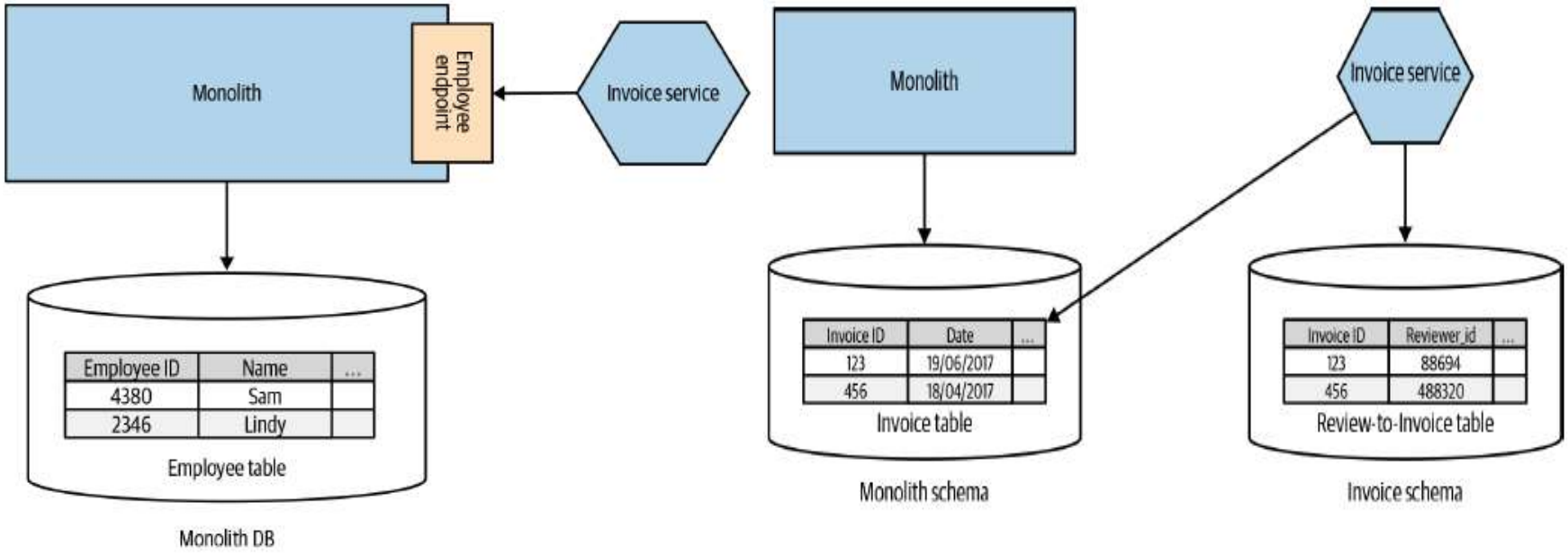
Split the Code First



Split the Code First

Pattern: Monolith as data access layer

Pattern: Multi-schema storage



Obstacles to decomposing an application into services

- Network latency
- Reduced availability due to synchronous communication
- Maintaining data consistency across services
- Obtaining a consistent view of the data
- God classes preventing decomposition

Defining service APIs

Introduction

A service API operation exists for one of two reasons:

- some operations correspond to system operations. They are invoked by external clients and perhaps by other services.
- The other operations exist to support collaboration between services. These operations are only invoked by other services

Assigning system operations to services

- Many system operations neatly map to a service, but sometimes the mapping is less obvious.

We can either

- Assign an operation to a service that needs the information provided by the operation is a better choice

- Assign an operation to the service that has the information necessary to handle it

Example

- Mapping system operations to services in the FTGO application

Service	Operations
Consumer Service	createConsumer()
Order Service	createOrder()
Restaurant Service	findAvailableRestaurants()
Kitchen Service	▪ acceptOrder() ▪ noteOrderReadyForPickup()
Delivery Service	▪ noteUpdatedLocation() ▪ noteDeliveryPickedUp() ▪ noteDeliveryDelivered()

Determining the APIs required to support collaboration between services

- Some system operations are handled entirely by a single service

- Some system operations span across multiple services.

Example

Service	Operations	Collaborators
Consumer Service	verifyConsumerDetails()	—
Order Service	createOrder()	▪ Consumer Service verifyConsumerDetails() ▪ Restaurant Service verifyOrderDetails() ▪ Kitchen Service createTicket() ▪ Accounting Service authorizeCard()
Restaurant Service	▪ findAvailableRestaurants() ▪ verifyOrderDetails()	—
Kitchen Service	▪ createTicket() ▪ acceptOrder() ▪ noteOrderReadyForPickup()	▪ Delivery Service scheduleDelivery()
Delivery Service	▪ scheduleDelivery() ▪ noteUpdatedLocation() ▪ noteDeliveryPickedUp() ▪ noteDeliveryDelivered()	—
Accounting Service	▪ authorizeCard()	—

Communication Protocols

Synchronous protocol

- The client sends a request and waits for a response from the service.

Asynchronous protocol

- The client code or message sender usually doesn't wait for a response.

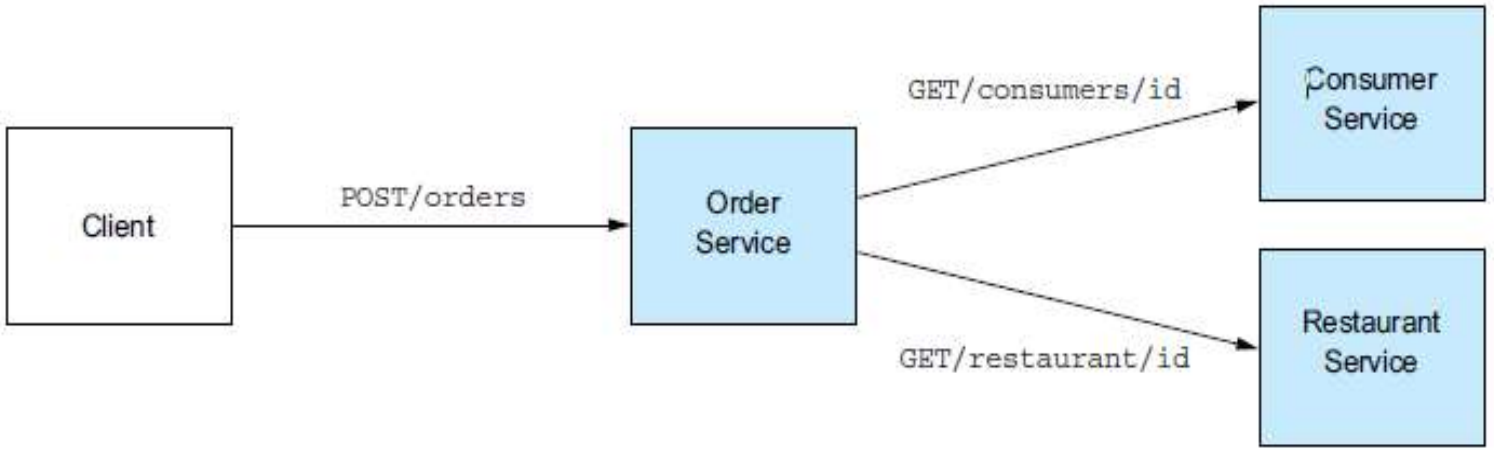
Number of Receivers

- Single receiver
- Multiple receivers

Synchronous communication

- REST is an extremely popular IPC mechanism under this category

Example: FTGO CreateOrder request



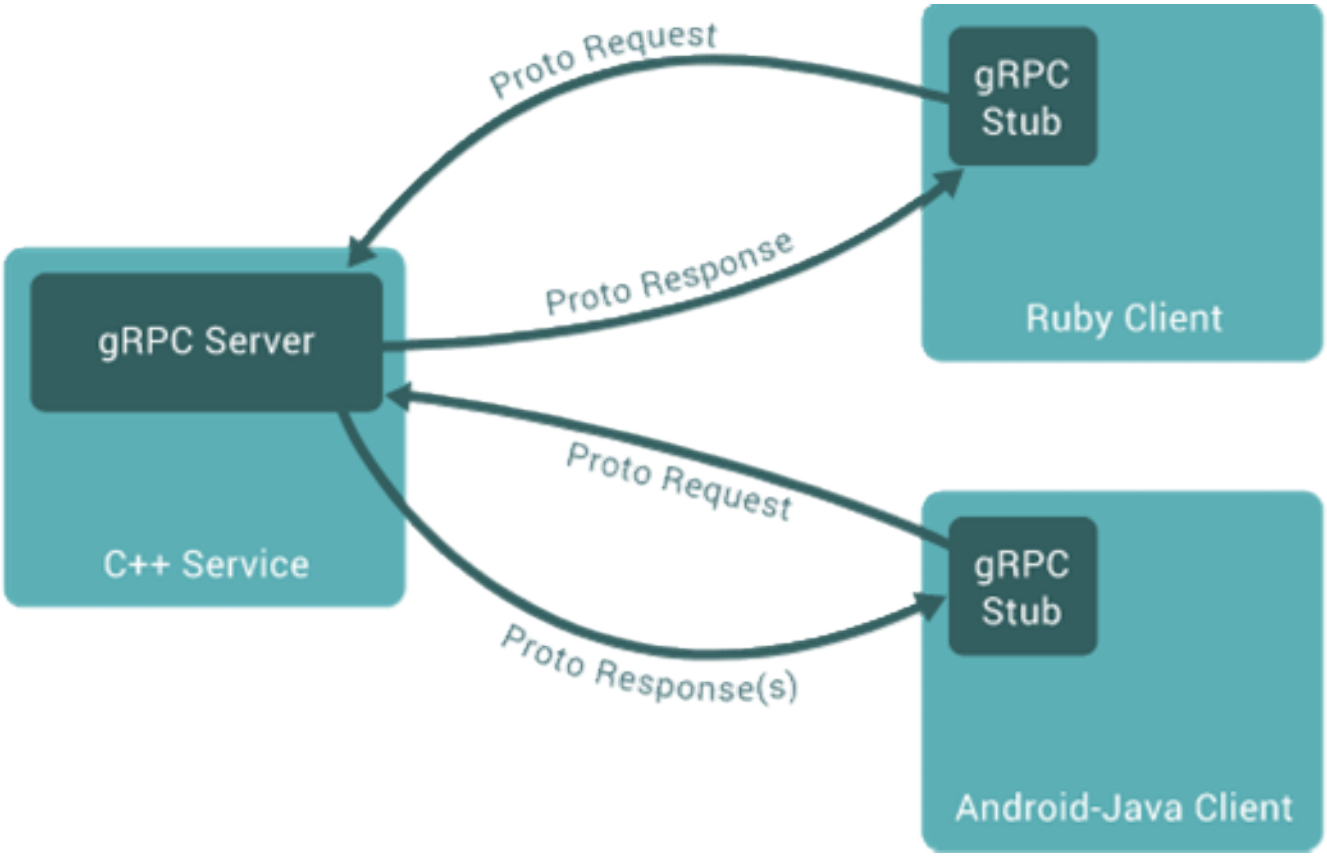
Representational state transfer (REST)

- It is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web.

Guiding Principles of REST

- Client–server
- Stateless
- Cacheable
- Uniform interface
- Layered system
- Code on demand (optional)

gRPC: Introduction



gRPC

- gRPC clients and servers can run and talk to each other in a variety of environments - from servers inside Google to your own desktop - and can be written in any of gRPC's supported languages.

- So, for example, you can easily create a gRPC server in Java with clients in Go, Python, or Ruby.

- In addition, the latest Google APIs will have gRPC versions of their interfaces, letting you easily build Google functionality into your applications

gRPC: Using the API

- On the server side, the server implements the methods declared by the service and runs a gRPC server to handle client calls.

- On the client side, the client has a local object known as *stub* that implements the same methods as the service.

RPC life cycle

Unary RPC

- Once the client calls a stub method, the server is notified that the RPC has been invoked
- The server can then either send back its own initial metadata straight away, or wait for the client's request message.
- Once the server has the client's request message, it does whatever work is necessary to create and populate a response. The response is then returned (if successful) to the client together with status details.
- If the response status is OK, then the client gets the response, which completes the call on the client side

Server streaming RPC

- A server-streaming RPC is similar to a unary RPC, except that the server returns a stream of messages in response to a client's request.
- After sending all its messages, the server's status details and optional trailing metadata are sent to the client.
- This completes processing on the server side. The client completes once it has all the server's messages.

Client streaming RPC

- A client-streaming RPC is similar to a unary RPC, except that the client sends a stream of messages to the server instead of a single message.

- The server responds with a single message along with its status details and optional trailing metadata

Bidirectional streaming RPC

- In a bidirectional streaming RPC, the call is initiated by the client invoking the method. The server can choose to send back its initial metadata or wait for the client to start streaming messages.

- Client- and server-side stream processing is application specific. Since the two streams are independent, the client and server can read and write messages in any order.

Deadlines/Timeouts

- gRPC allows clients to specify how long they are willing to wait for an RPC to complete before the RPC is terminated

- On the server side, the server can query to see if a particular RPC has timed out, or how much time is left to complete the RPC.

RPC termination

- In gRPC, both the client and server make independent and local determinations of the success of the call, and their conclusions may not match.

- This means that, for example, you could have an RPC that finishes successfully on the server side but fails on the client side. It's possible for a server to decide to complete before a client has sent all its requests

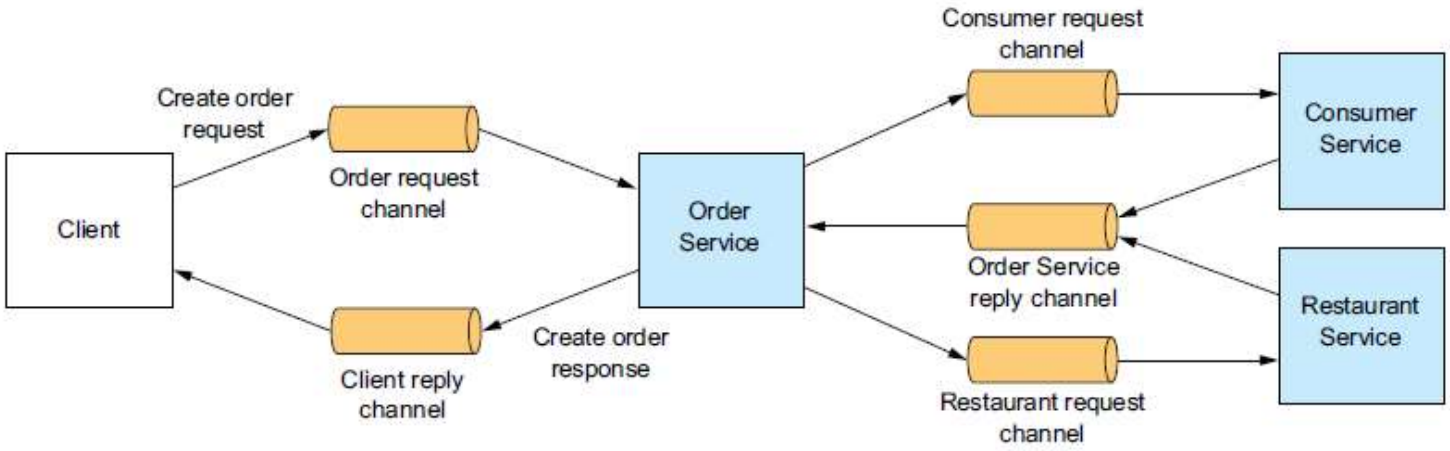
Asynchronous communication

- It means communication which happens ‘out of sync’ or in other words; not in real-time.

- For example, an email to a colleague would be classed as asynchronous communication

Asynchronous Communication

- Services communicating by exchanging messages over messaging channels.



Real Time Examples

- Email
- Messages via any instant messaging app (e.g. WhatsApp messenger, RingCentral Message, Slack)
- Messaging via project management tools such as Basecamp, Trello, Mondays etc.
- Intranets such as Yammer or Sharepoint.

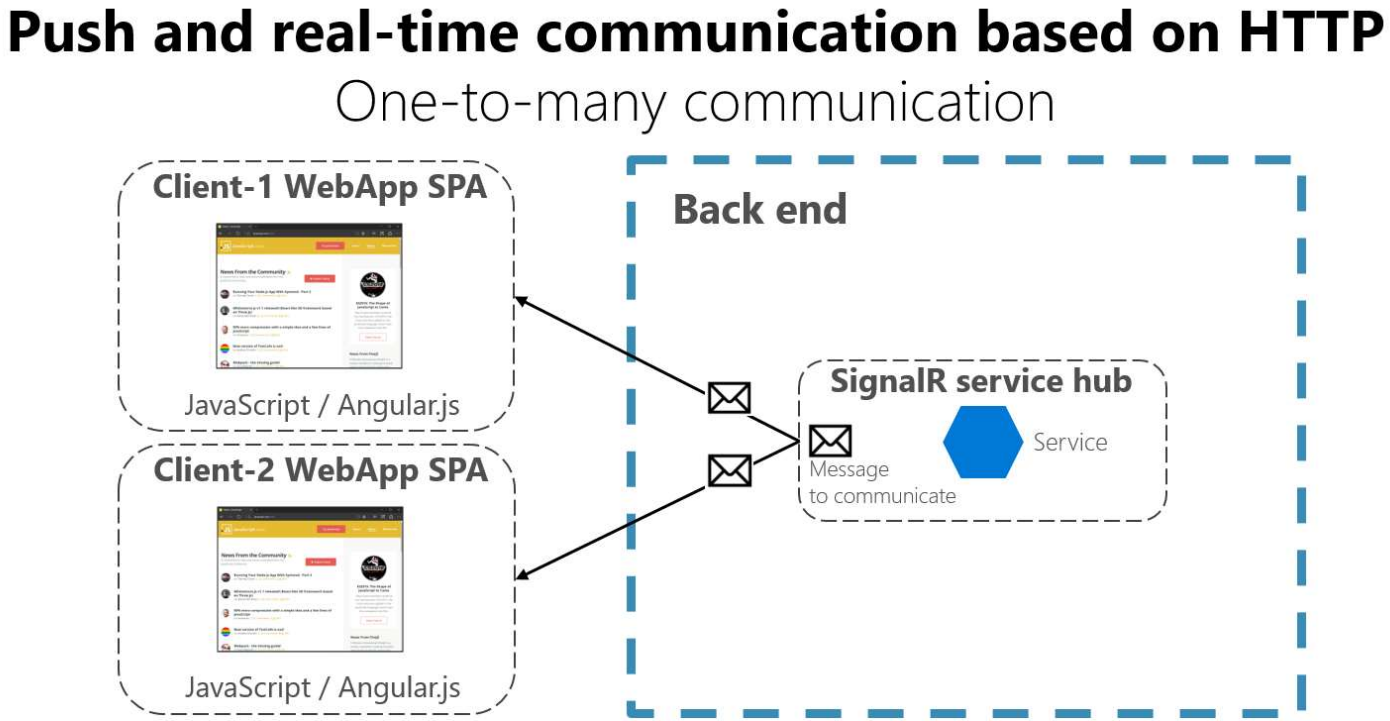
innovate achieve lead

BITS Pilani

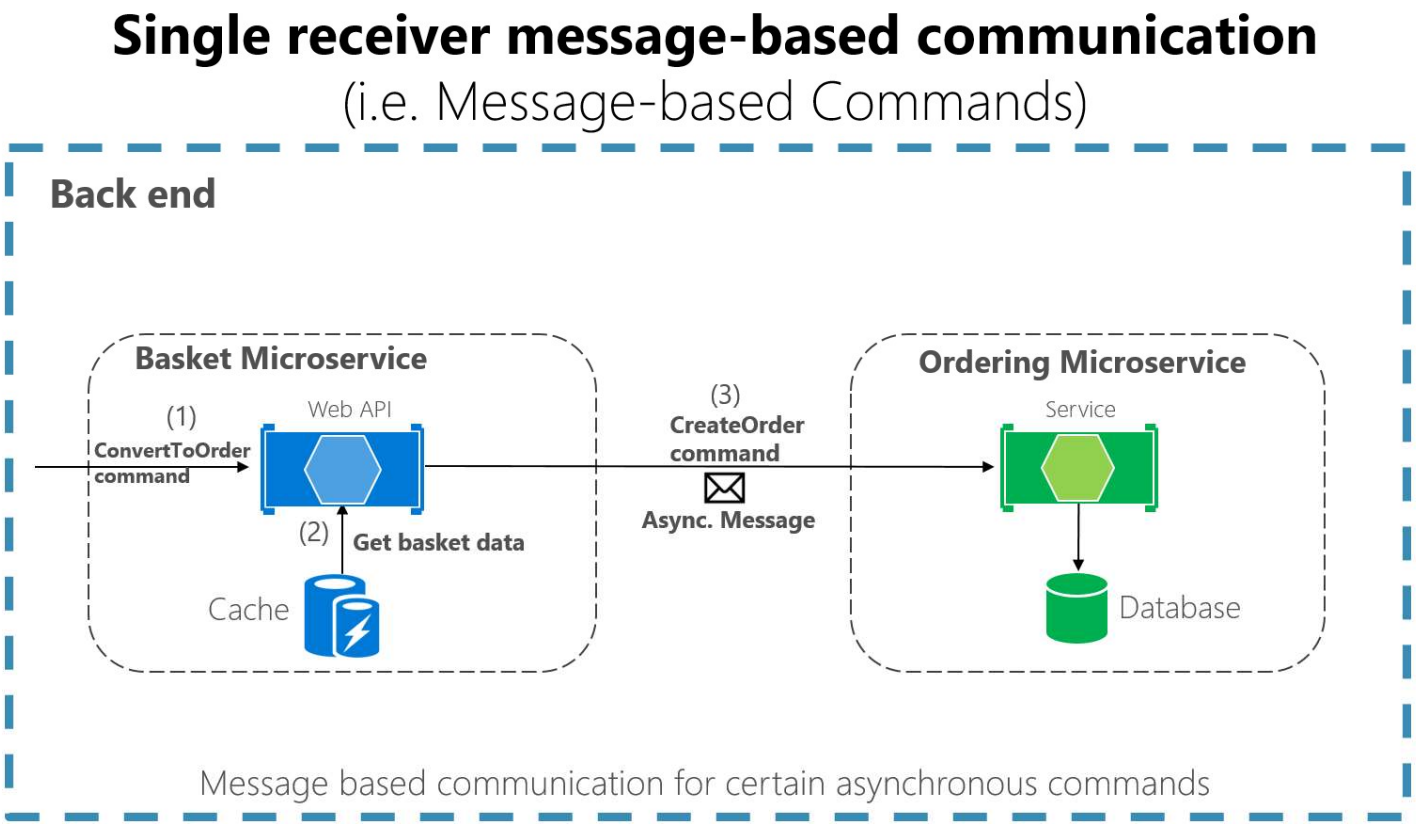
Pilani Campus

Communication Styles

Push and real-time communication based on HTTP



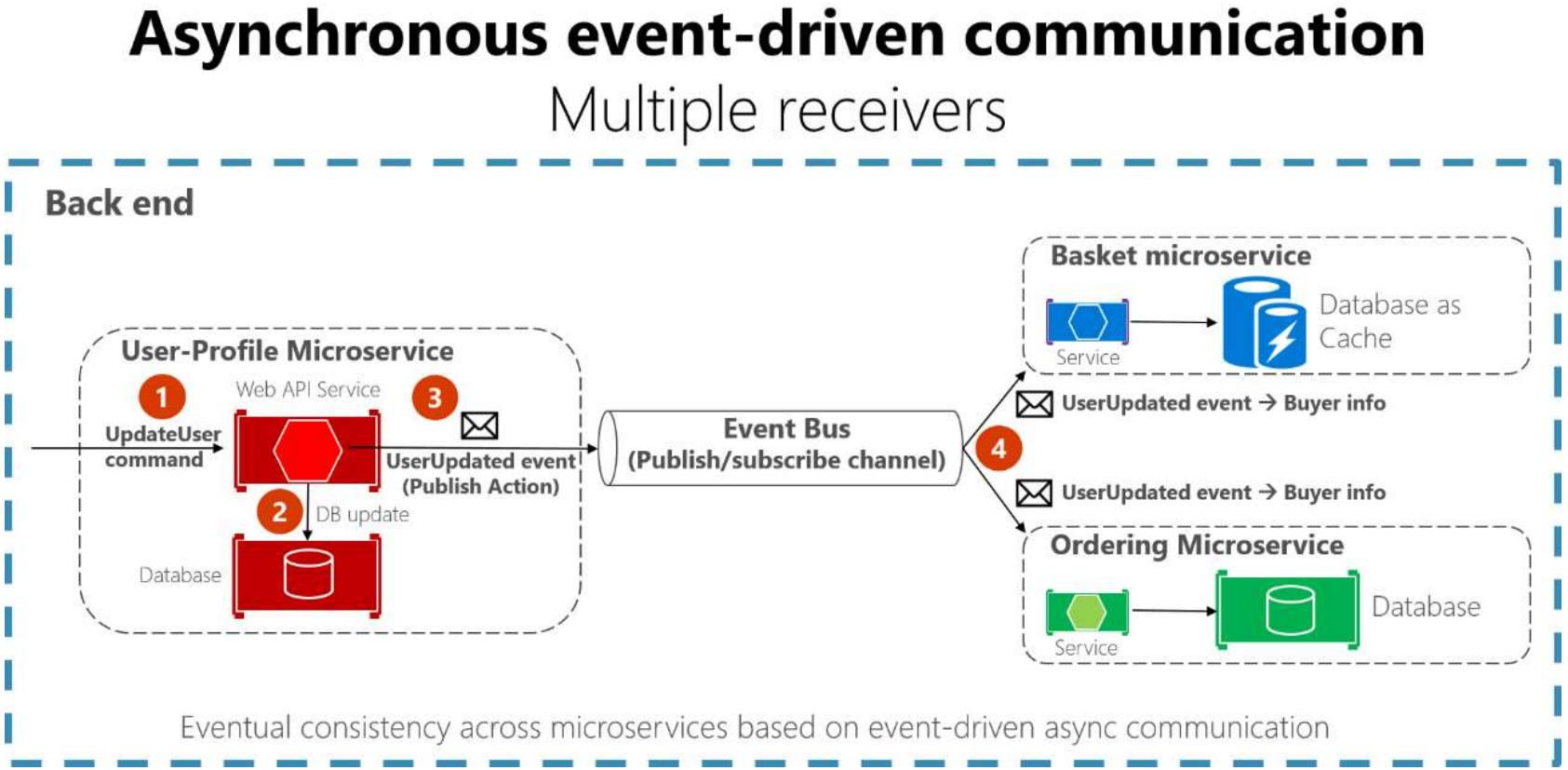
Single-receiver message-based communication



Multiple-receivers message-based communication

Use a publish/subscribe mechanism so that your communication from the sender will be available to all the subscriber microservices or to external applications.

Asynchronous event-driven communication



References

- Book: Microservices Patterns by Chris Richardson
- Book: Monolith to Microservices by Sam Newman
- Link: https://microservices.io/patterns
- Link: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/communication-in-microservice-architecture
- https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
- https://grpc.io/docs/what-is-grpc/introduction/

Communication and Transaction management

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani

Pilani Campus

SE ZG583, Scalable Services
Lecture No. 7

Introduction

Let's imagine that you are developing a native mobile client for a shopping application.

The product details page displays a lot of information

- Number of items in the shopping cart
- Order history
- Basic Product Information
- Customer reviews
- Low inventory warning
- Shipping options
- Various suggested items

Direct Communication

A client could make requests to each of the microservices directly

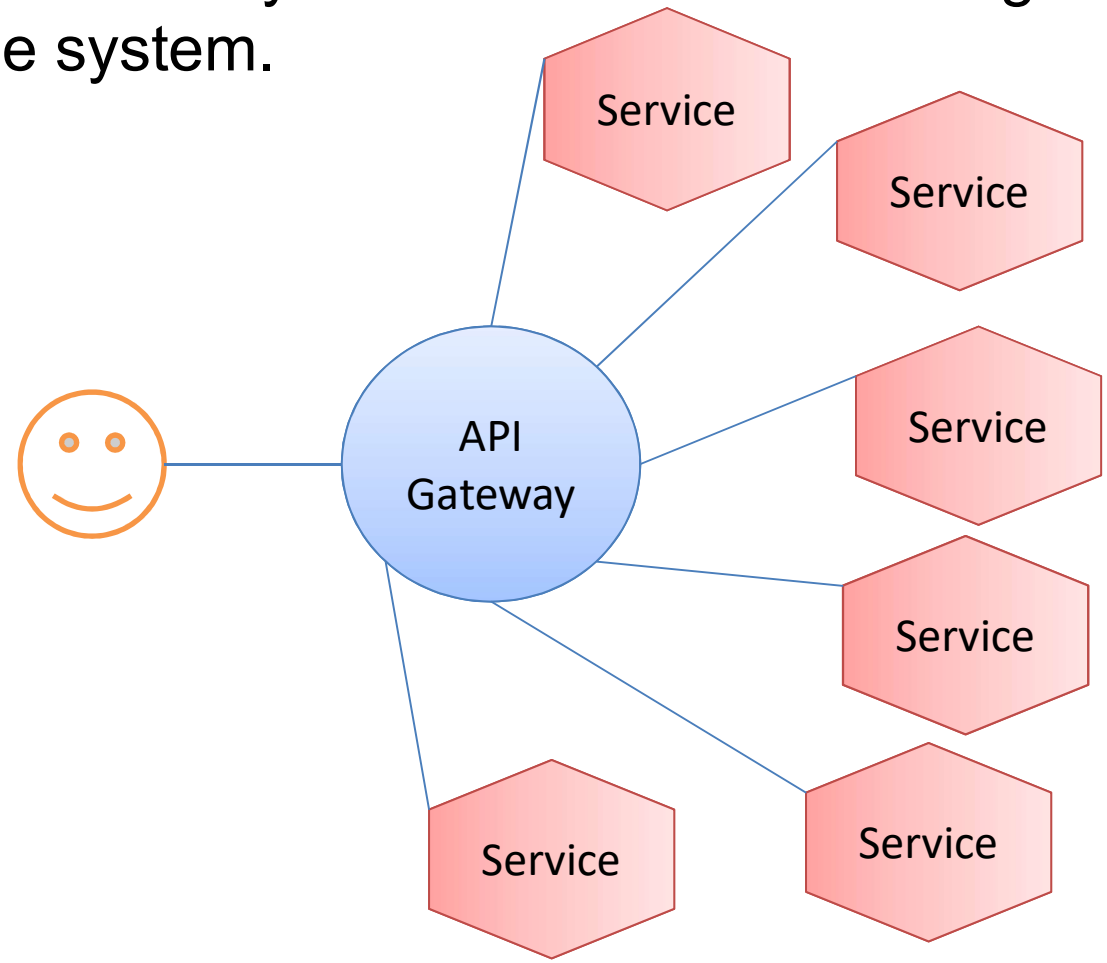
But there are challenges and limitations with this option

- Inefficient to call directly.
- When the client directly is calling the microservices they might use protocols that are not web-friendly.

Using an API Gateway

innovate achieve lead

An API Gateway is a server that is the single entry point into the system.

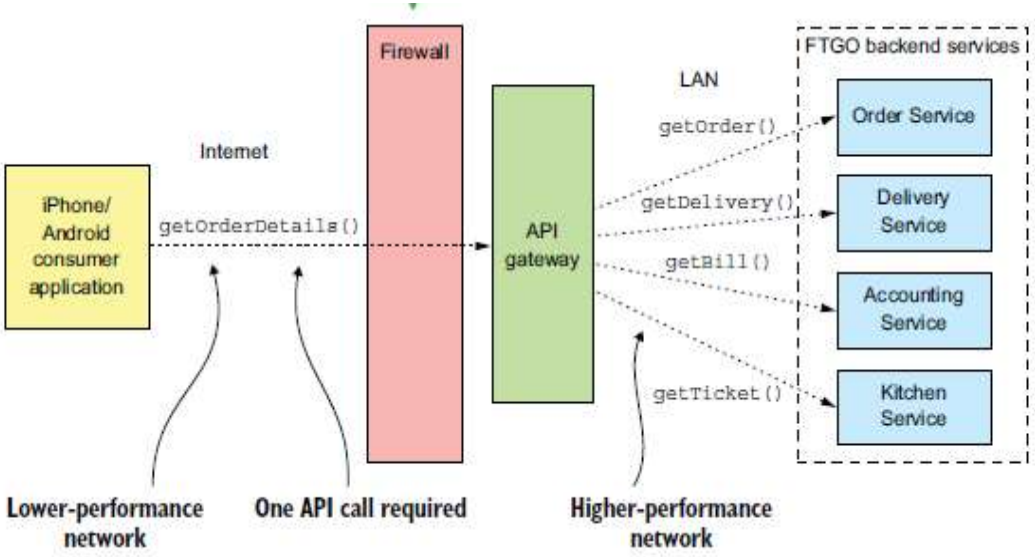


Using an API Gateway: Request Routing

An API gateway implements some API operations by routing requests to the corresponding service.

API Composition

API Gateway might implement some API operations using API composition.



Using an API Gateway: Protocol Translation

It might provide a RESTful API to external clients

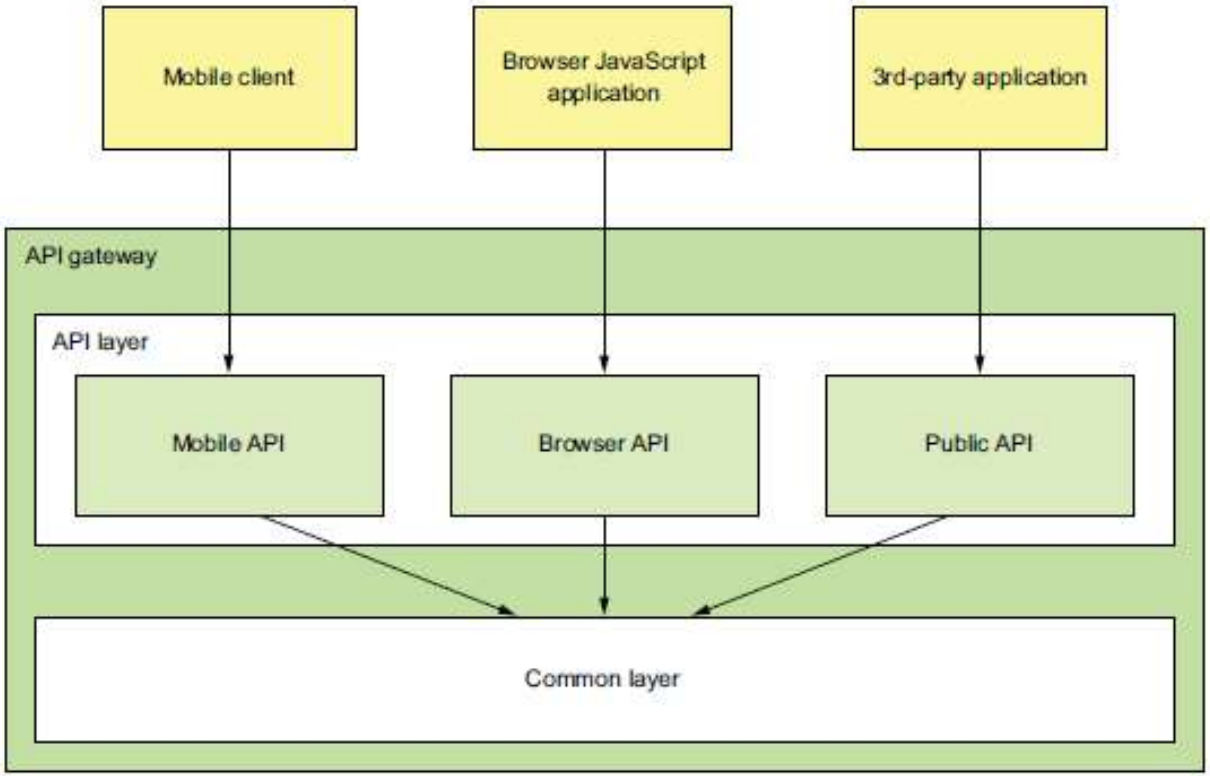
API Gateway provides each client with client-specific API

- Android client and IOS client separate

Implementing edge functions

- Authentication
- Authorization
- Rate limiting
- Caching
- Metrics collection
- Request logging

API Gateway Architecture



API Gateway design issues

- Performance and scalability
- Writing maintainable code by using reactive programming abstractions
- Handling partial failure
- Being a good citizen in the application's architecture

Benefits of an API Gateway

- It encapsulates internal structure of the application.
- It also simplifies the client code

Drawbacks of an API Gateway

- Another component that must be developed, deployed, and managed.
- API gateway can become a development bottleneck

Microservices API Gateway Vs. Traditional API Gateway

Traditional API Gateway	Microservices API Gateway
It comes up with a simple mechanism to create new services.	It makes use of tools quickly that allows service teams to easily and safely create new services.
It enables functionality for monitoring usage of API per client.	It monitors the user-visible metrics like availability to clients.
Routing manages to change API versions.	It integrates routing to roll out the new versions of services quickly.

What is API design?

API design refers to the process of developing application programming interfaces (APIs) that expose data and application functionality for use by developers and users.

Designing APIs

- A service's API is a contract between the service and its clients.
- The challenge is that a service API isn't defined using a simple programming language construct.
- The nature of the API definition depends on which IPC mechanism you're using.
- APIs invariably change over time as new features are added, existing features are changed, and old features are removed

How do we design our API program

Teams must ask themselves:

- What technology is used to build the APIs?
- How are the APIs designed?
- How are the APIs maintained?
- How are the APIs promoted inside the organization or marketed to the outside world?
- What resources are available?
- Who should be on the team?

Good API design

- Simplicity
- Flexibility

External API design issues

- One approach to API design is for clients to invoke the services directly.
- But this approach is rarely used in a microservice architecture because of a lot of drawbacks

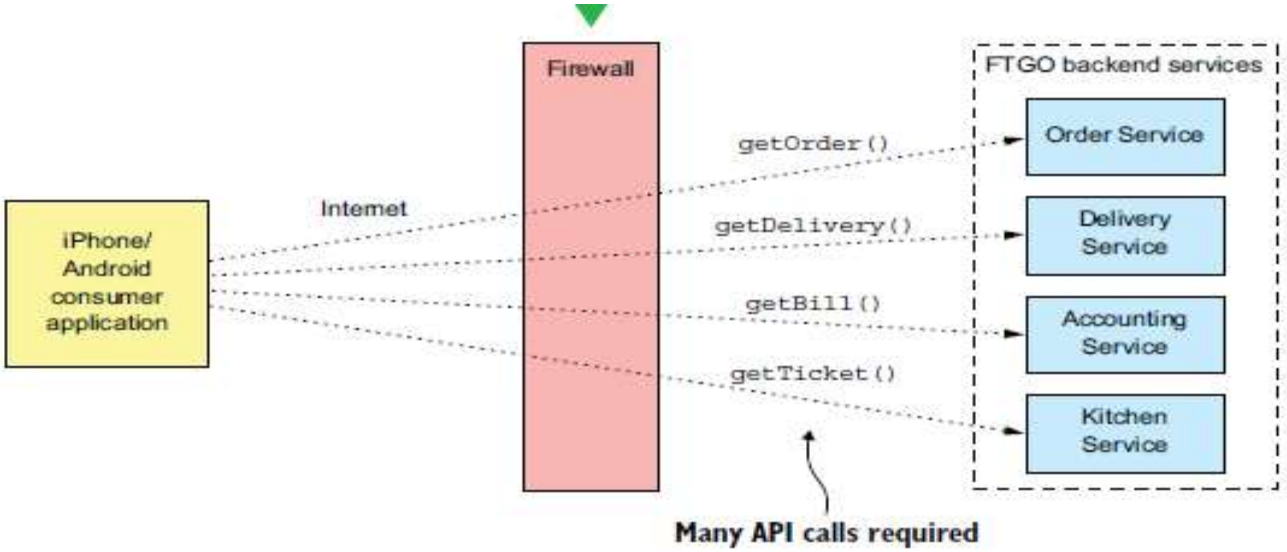


Figure 8.2 A client can retrieve the order details from the monolithic FTGO application with a single request. But the client must make multiple requests to retrieve the same information in a microservice architecture.

When To Create An API Version

API versioning should occur any time you:

- Change fields or routing after an API is released.
- Change payload structures, such as changing a variable from an integer to float, for instance.
- Remove endpoints to fix design or poor implementation of HTTP.

What Is The API Contract?

The API contract is the agreement between the API producer and consumer.

An API contract is also a way to keep track of the changes made to an API.

Handling changes in Microservice related APIs

- Use semantic versioning
- Making minor, backward-compatible changes
- Making major, breaking changes

API versioning methods

- URI Versioning
- Query Parameter Versioning
- Custom Headers

What is API security?

- API security is the protection of the integrity of APIs, both the ones you own and the ones you use.

Why is API security important?

- Broken, exposed, or hacked APIs are behind major data breaches.

- If your API connects to a third party application, understand how that app is funneling information back to the internet.

REST API security

- REST APIs use HTTP and support Transport Layer Security (TLS) encryption.
- TLS is a standard that keeps an internet connection private and checks that the data sent between two systems (a server and a server, or a server and a client) is encrypted and unmodified.

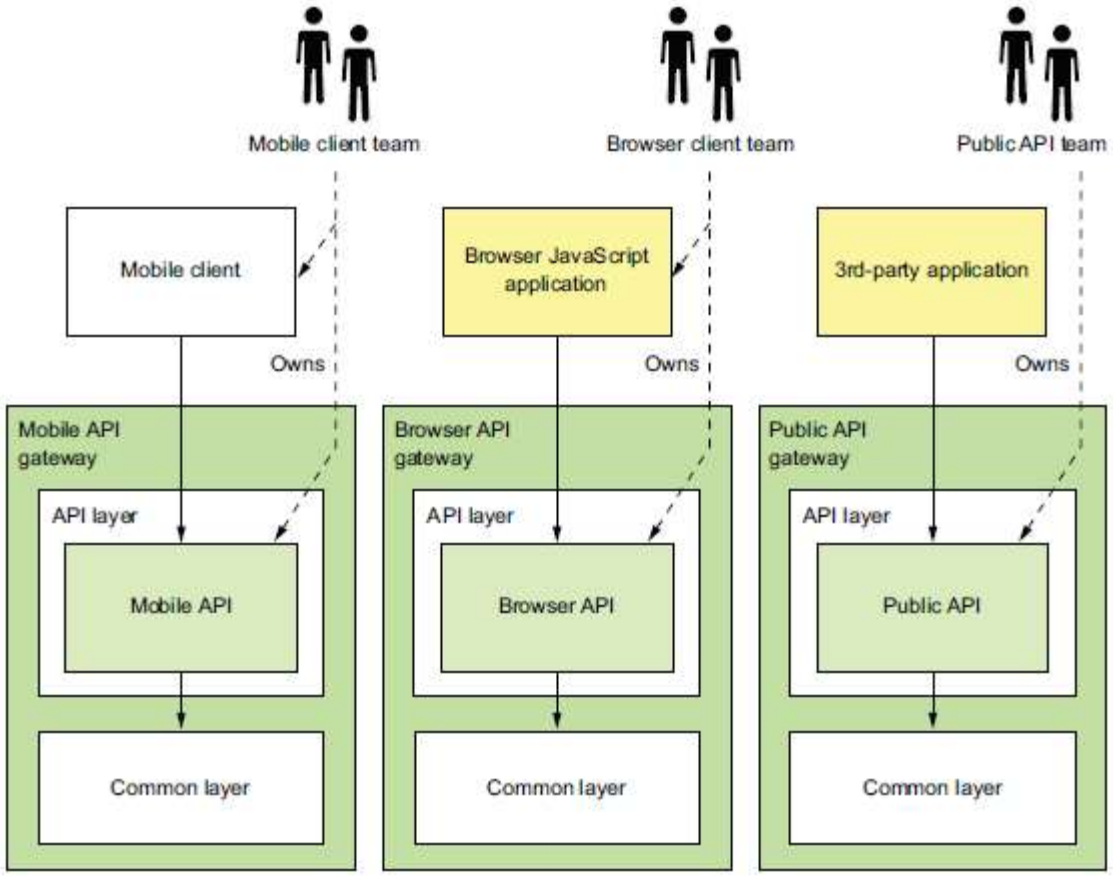
Most common API security best practices?

- Use tokens
- Use encryption and signatures
- Identify vulnerabilities
- Use quotas and throttling
- Use an API gateway

API management and security

- An API key
- Basic Authentication
- OpenID Connect

Backends for Frontends



Benefits of BFF pattern

- The API modules are isolated from one another, which improves reliability.
- It also improves observability, because different API modules are different processes
- Another benefit of the BFF pattern is that each API is independently scalable.
- The BFF pattern also reduces startup time because each API gateway is a smaller, simpler application.

Transaction management in a microservice architecture

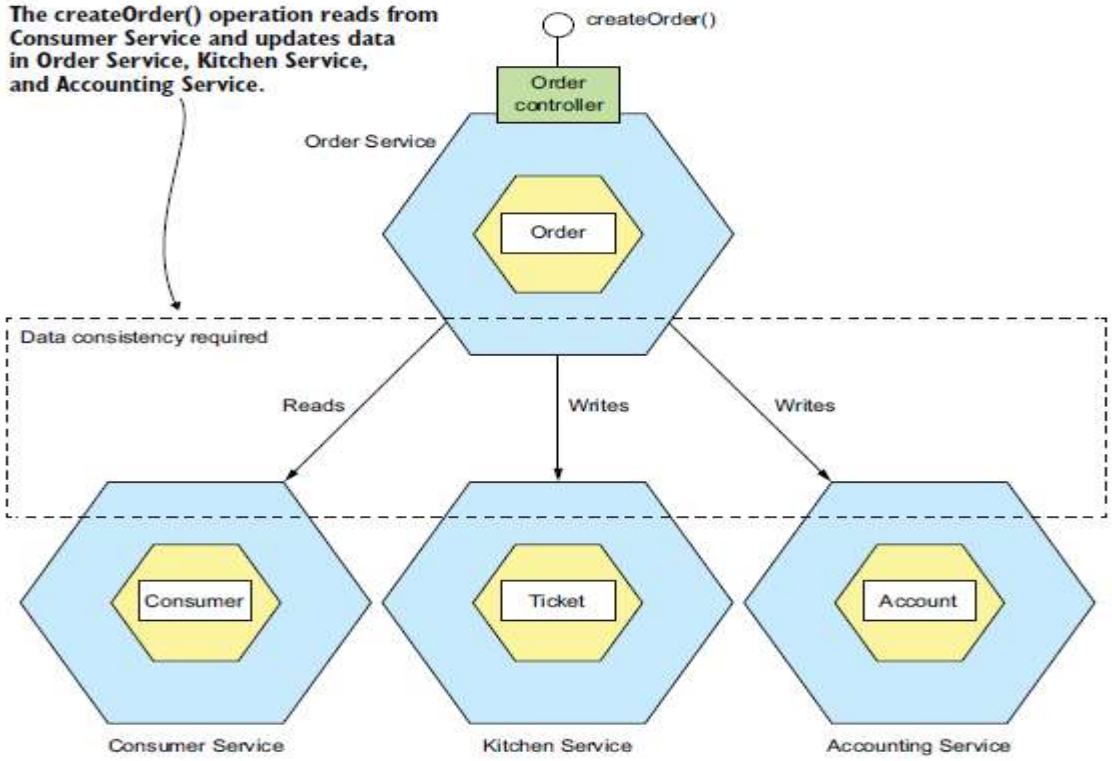
Need for distributed transactions in a MSA

- Imagine that you're the FTGO developer responsible for implementing the create- Order() system operation

- this operation must verify that the consumer can place an order, verify the order details, authorize the consumer's credit card, and create an Order in the database

Challenges with distributed transactions

- The traditional approach to maintaining data consistency across multiple services, databases, or message brokers is to use distributed transactions

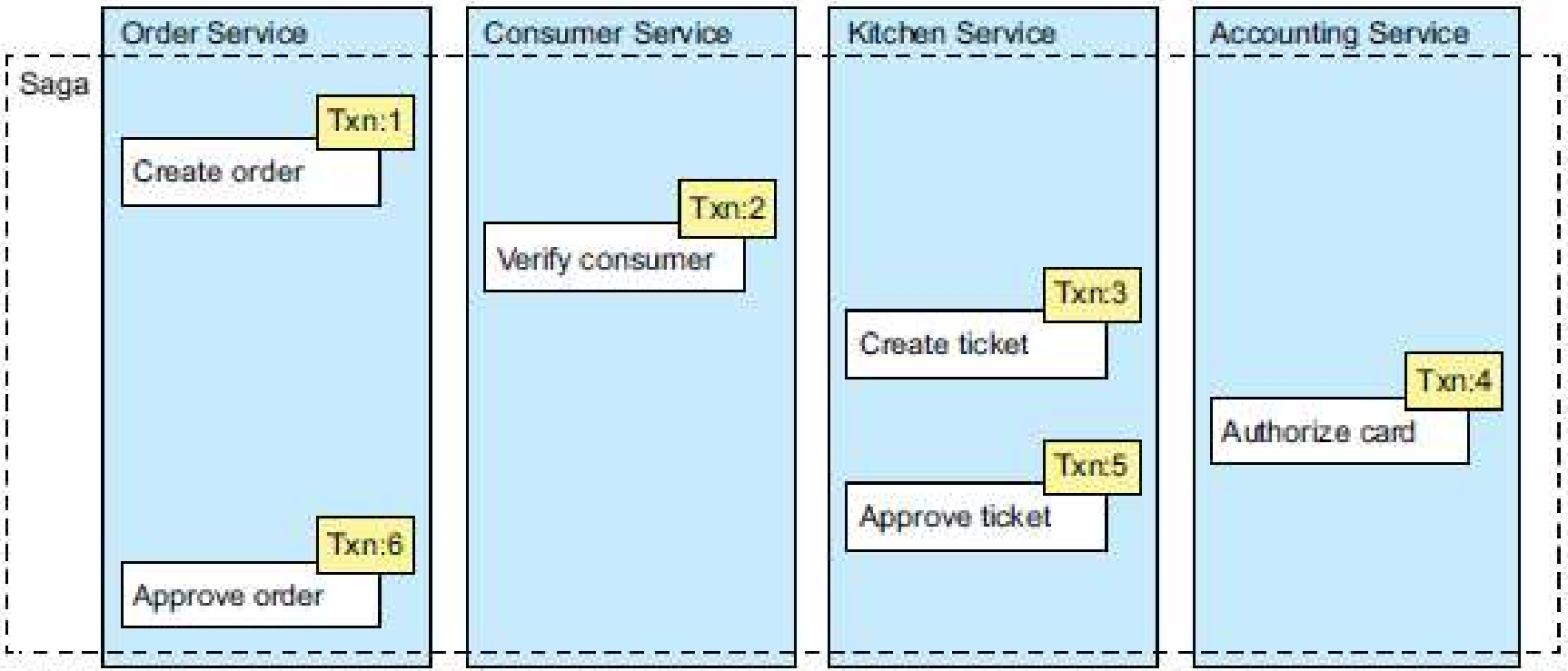


- One problem is that many modern technologies, including NoSQL databases such as MongoDB and Cassandra, don’t support them.
- They are a form of synchronous IPC, which reduces availability. In order for a distributed transaction to commit, all the participating services must be available.
- To solve the more complex problem of maintaining data consistency in a microservice architecture, an application must use a different mechanism that builds on the concept of loosely coupled, asynchronous services.

Solution: Saga pattern

- Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions.

Example

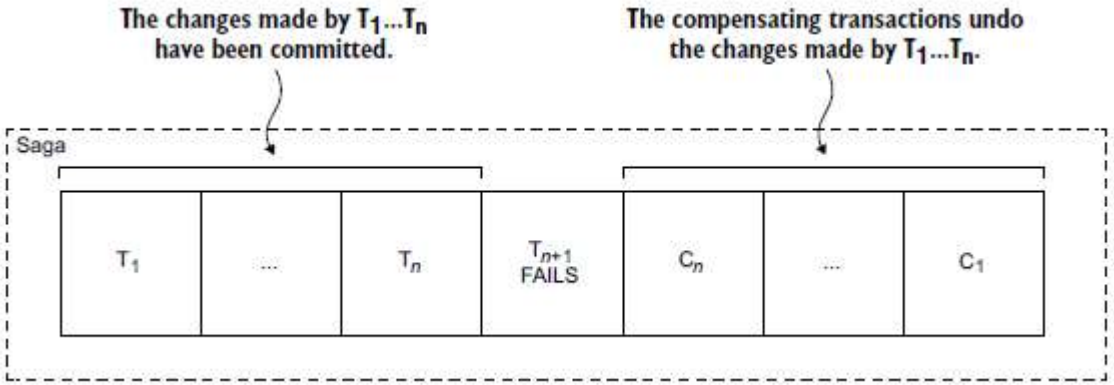

Order Service
Consumer Service
Kitchen Service
Accounting Service
Saga
Txn:1
Create order
Txn:2
Verify consumer
Txn:3
Create ticket
Txn:4
Authorize card
Txn:5
Approve ticket
Txn:6
Approve order

Challenges in Saga

- lack of isolation between sagas
- rolling back changes when an error occurs

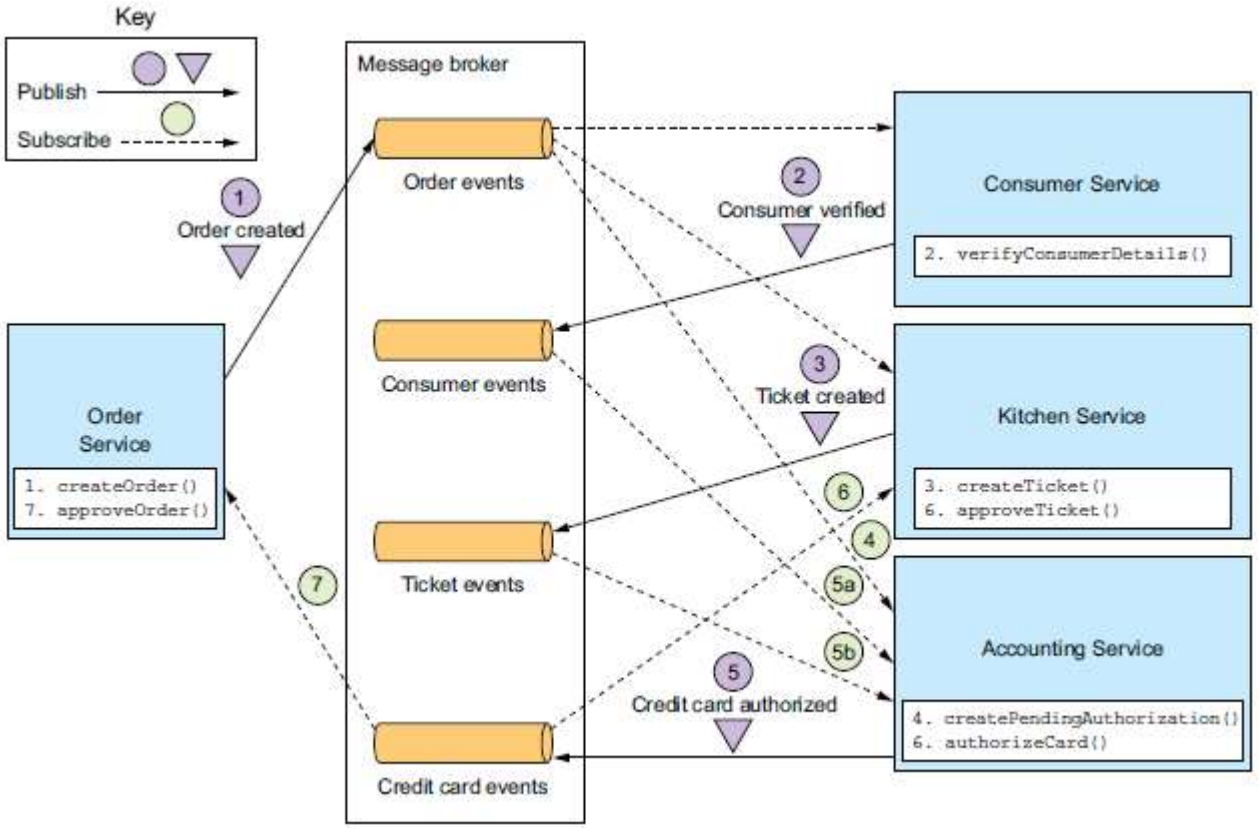
Compensating transactions

- Unfortunately, sagas can't be automatically rolled back, because each step commits its changes to the local database.



Step	Service	Transaction	Compensating transaction
1	Order Service	createOrder()	rejectOrder()
2	Consumer Service	verifyConsumerDetails()	—
3	Kitchen Service	createTicket()	rejectTicket()
4	Accounting Service	authorizeCreditCard()	—
5	Kitchen Service	approveTicket()	—
6	Order Service	approveOrder()	—

Create Order Saga



Self Study

- API in detail
- Sagas in detail

References

1. Book: Microservices Patterns by Chris Richardson
2. Link: https://www.nginx.com/blog/building-microservices-using-an-api-gateway/
3. https://www.redhat.com/en/topics/api/what-is-api-design
4. https://www.redhat.com/en/topics/security/api-security
5. https://marutitech.com/api-gateway-in-microservices-architecture/

Microservices demo

BITS Pilani
Pilani Campus

Akanksha Bharadwaj
Asst. Professor, CSIS Department

innovate achieve lead

BITS Pilani

Pilani Campus

SE ZG583, Scalable Services
Lecture No. 8

Design and build Microservice

- Demo of making a sample service in Django
- Demo of making a sample service in Flask

Connecting Database to service

- Connect database to Django service
- Connect Database to Flask service

Using Shared DB

- Demo on how can two services communicate with each other by using a shared database

Revision

When to Use Hadoop

- For Processing Really BIG Data
- For Storing a Diverse Set of Data
- For Parallel Data Processing

Kafka Vs Rabbit MQ

- RabbitMQ and Apache Kafka allow producers to send messages to consumers.
- Producers and consumers interact differently in RabbitMQ and Kafka.
  - In RabbitMQ, the producer sends and monitors if the message reaches the intended consumer.
  - On the other hand, Kafka producers publish messages to the queue regardless of whether consumers have retrieved them.
- Kafka is suitable for applications that need to reanalyze the received data. You can process streaming data multiple times within the retention period or collect log files for analysis. Log aggregation with RabbitMQ is more challenging, as messages are deleted once consumed.
- Rabbit MQ suits applications that must adhere to specific sequences and delivery guarantees when exchanging and analyzing data.

When to use CDN

- If your platform serves a global audience, and you aim to provide a seamless viewing experience, then a CDN is a must-have.
- CDN for large high-load websites.

When to use Load balancer

- Load Balancing for Scale
- High availability

Edge computing scenarios

- Enterprise edge
- Operations edge
- Provider edge

When to use CQRS pattern

- Collaborative domains where many users access the same data in parallel.
- You can use the CQRS pattern to separate updates and queries if they have different requirements for throughput, latency, or consistency.

When serverless architecture is not the right choice

- Entirely Serverless application is not suitable for real-time applications that use WebSockets because FaaS functions have limited lifetime
- After some time of being idle, function will require to go through a cold start which can take up to a few seconds.
- Different FaaS providers may differ in some particularities of using their services which will make the switch to another provider troublesome.