Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

By- Akanksha Bharadwaj (Asst. Professor CSIS department)

Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

By- Akanksha Bharadwaj (Asst. Professor CSIS department)

Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

By- Akanksha Bharadwaj (Asst. Professor CSIS department)

Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

By- Akanksha Bharadwaj (Asst. Professor CSIS department)

Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

By- Akanksha Bharadwaj (Asst. Professor CSIS department)

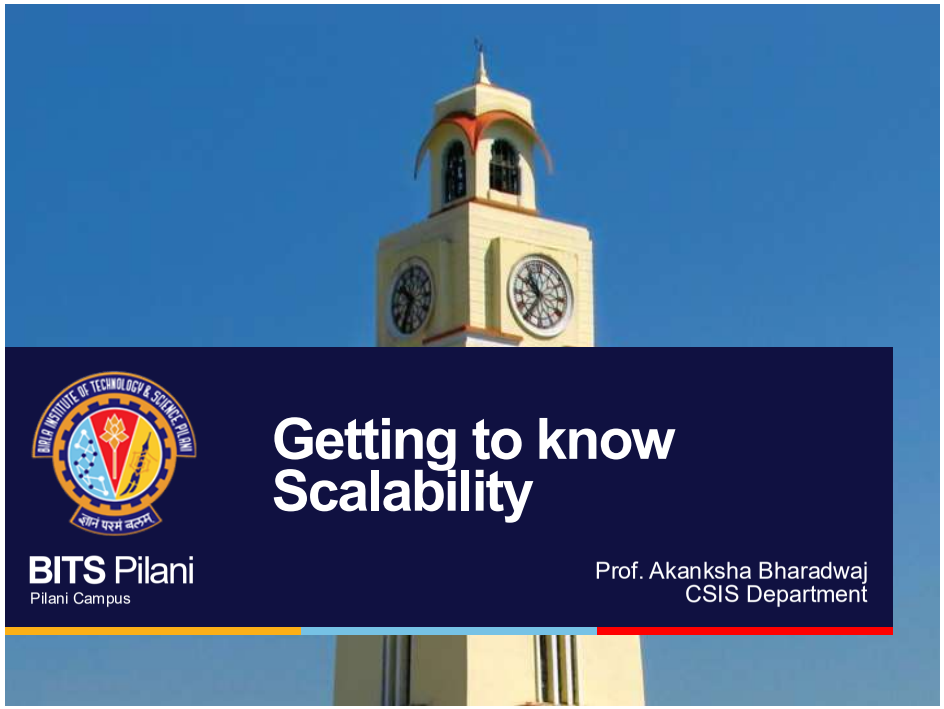
Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

Work Integrated Learning Programmes BITS Pilani Pilani | Dubai | Goa | Hyderabad

By- Akanksha Bharadwaj (Asst. Professor CSIS department)

SE ZG583, Scalable Services
Lecture No. 1



Getting to know Scalability

Prof. Akanksha Bharadwaj
CSIS Department

CAP Theorem

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

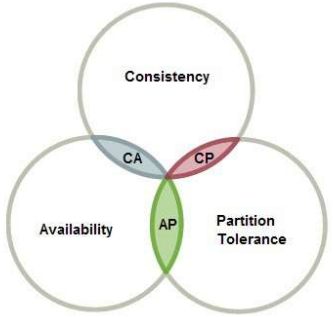


Image: Towardsdatascience

Introduction to Performance, Consistency and availability

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.

CAP Theorem

Consistency
Availability
Partition tolerance

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

  .....

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
- Can be measured using response time, throughput etc.

Need for scalable architectures

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

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

Principles of Scalability

- Scale horizontally, not vertically
- Statelessness
- Cache
- Load Balancing
- Modular Architecture
- Asynchronous Processing
- Monitoring and Metrics
- Resilience and Fault Tolerance

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

Disadvantages of Monolith

- Scalability
- Slow development
- Long deployment cycle

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

Guidelines for Building Highly Scalable Systems

- Avoid shared resources as they might become a bottleneck
- Avoid slow services
- Optimize Database Design
- Auto-Scaling
- Prioritize API Design
- Ensure Data Consistency and Partitioning
- Optimize Network Usage
- Minimize Latency
- Plan for Growth

Case Study

Challenges for Scalability

Here are some common **challenges for scalability** in real-world applications:

- Unanticipated Traffic Spikes
- Database Scalability
- Resource Contention
- Distributed System Complexities
- Legacy Systems
- Cost of Scalability
- Security at Scale
- Scalability Testing

Scalability in YouTube's Architecture

Video Storage and Distribution

Storage Challenges:

- YouTube stores petabytes of data and must ensure quick access for playback.

YouTube case Study

Key Challenges in YouTube's Scalability:

- Massive User Base
- High Data Throughput
- Global Availability
- Real-Time Features.
- Storage
- Traffic Spikes

Data Management

Challenges:

- Storing and retrieving metadata for billions of videos efficiently.

Video Processing and Transcoding

Challenges:

- Each uploaded video must be transcoded into multiple resolutions and formats to support devices with different network conditions and capabilities.

Live Streaming Scalability

Challenges:

- Streaming live events like concerts, sports, or news to millions of concurrent viewers.

Real-Time Recommendations

Challenges:

- Generating personalized video recommendations for billions of users in real-time.

Self Study

Example of scalable architecture:
https://www.youtube.com/watch?v=VHELcOe1gy0

Fault Tolerance and Resilience

Challenges:

- Ensuring uninterrupted service even during server failures or network issues.

References

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

Partitioning and Sharding

SE ZG583, Scalable Services
Lecture No. 2

Introduction

- Partitioning and Sharding are scalable approaches to manage and distribute large volumes of data in a database system.
- 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

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

Scalability using NoSQL and HDFS

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

A-G Shard

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

H-Z Shard

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

Functional partitioning

Corporate data domain

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	Customer	Address	Phone	...
1630	[name]	[address]	12345	...
1631	[name]	[address]	12345	...
1648	[name]	[address]	12345	...
1842	[name]	[address]	12345	...
2055	[name]	[address]	12345	...
2139	[name]	[address]	12345	...

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	Customer	Address	Phone	...
1630	[name]	[address]	12345	...
1631	[name]	[address]	12345	...
1648	[name]	[address]	12345	...
1842	[name]	[address]	12345	...
2055	[name]	[address]	12345	...
2139	[name]	[address]	12345	...

NoSQL and Scalability

NoSQL databases (e.g., MongoDB, Cassandra, DynamoDB) are specifically built for high scalability, especially in scenarios requiring distributed systems, real-time data access, and schema flexibility.

Scalability Features in NoSQL:

- Horizontal Scalability (Sharding)
- Dynamic Schema
- Replication
- High Throughput
- Decentralized Architecture

Use Cases for NoSQL Scalability

- Social media platforms managing large volumes of user interactions.
- E-commerce systems requiring fast, scalable databases for product catalogs and transactions.
- Real-time analytics on streaming data.

HDFS and Scalability

HDFS is a distributed file system designed for storing and processing massive datasets (big data) in a scalable and fault-tolerant manner. It forms the foundation of the Hadoop ecosystem.

Scalability Features in HDFS:

- Horizontal Scalability
- Replication for Fault Tolerance
- Write Once, Read Many (WORM) Design
- MapReduce Integration
- Block Storage
- Cluster Management

Use Cases for HDFS Scalability

- Storing and analyzing massive datasets (e.g., petabytes of log data).
- Batch processing for data warehouses and ETL pipelines.
- Data lakes for structured and unstructured data.

Managing high velocity data streams

Key Challenges

- High Throughput
- Low Latency
- Scalability
- Fault Tolerance
- Storage Efficiency

Strategies for Managing High-Velocity Video Data

1. Content Delivery Network

- CDN is nothing more than a bunch of globally distributed computers that are directly connected and move data from one end to another.
- CDN stores multiple copies of video content at edge locations.
- We can use CDNs to cache and distribute video content closer to end-users, reducing latency and bandwidth usage.
- Examples: Akamai, Cloudflare, AWS CloudFront.

2. Adaptive Bitrate Streaming (ABR)

- Deliver video streams in different quality levels (bitrates) and dynamically switch between them based on the user's network speed and device capabilities.
- Example: HLS (HTTP Live Streaming), DASH (Dynamic Adaptive Streaming over HTTP).

CDN Case Study: YouTube

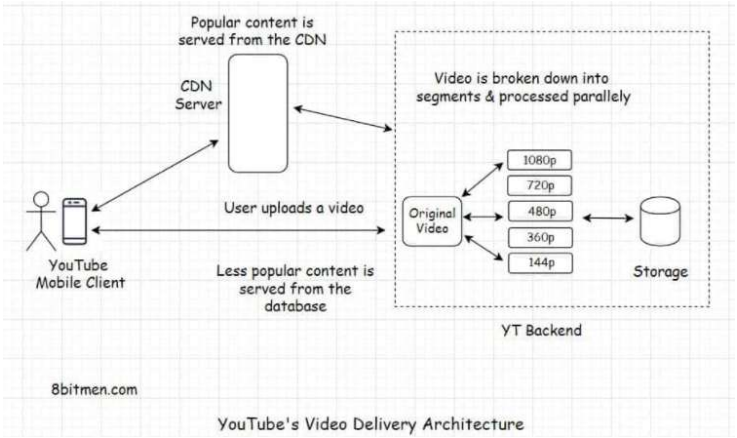


Image: google

4. 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.
- We can process and cache video content at edge locations to reduce latency and offload computation from the central servers.
- Example: Transcode video streams at the edge for faster delivery.

3. Efficient Video Compression

- Use advanced codecs to reduce video file size while maintaining quality.
- **Example for video Streaming**: To make the videos viewable on different devices Netflix performs transcoding or encoding which involves finding errors and converting the original video into different formats and resolutions.

5. Real-Time Data Processing

- Process telemetry data (e.g., playback statistics, user behavior) in real time to optimize streaming.
- Use stream processing frameworks like Apache Flink or Kafka Streams to analyze and react to user interactions instantly.

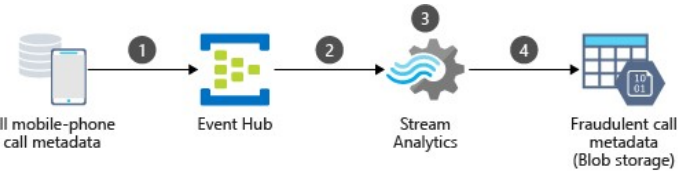
Case Study: Smart Security Camera

- **Device with Edge Capability**: A smart security camera equipped with an edge AI chip processes video feeds locally.
- **Edge Processing**: The camera runs motion detection algorithms (e.g., using TensorFlow Lite) directly on the device to identify unusual activity, like someone entering the camera's field of view.
- **Action at the Edge:** If motion is detected: Capture the event (e.g., save a short video clip). Send an alert to the homeowner's mobile device (e.g., "Motion detected at the front door"). If no motion is detected, discard the video feed to save bandwidth.
- **Cloud Involvement:** Only the processed, important data (e.g., the detected motion event) is sent to the cloud for storage or additional analysis.

Real Time Analytics

Case Study: 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.

Steps in Processing Streaming Data for Web Conferencing

1. Data Capture
2. Data Compression
3. Data Transmission
4. Stream Processing at the Server
5. Adaptive Bitrate Streaming
6. Content Delivery
7. Client-Side Rendering
8. Data Analytics and Feedback

Web Conferencing

- Web conferencing involves real-time audio, video, and data sharing, requiring efficient processing of high-velocity data streams to ensure a seamless user experience.

Managing high velocity Data Streams using Kafka

- **Kafka** is a powerful, distributed streaming platform designed to handle massive amounts of data in real-time. It's widely used for building real-time data pipelines and applications.

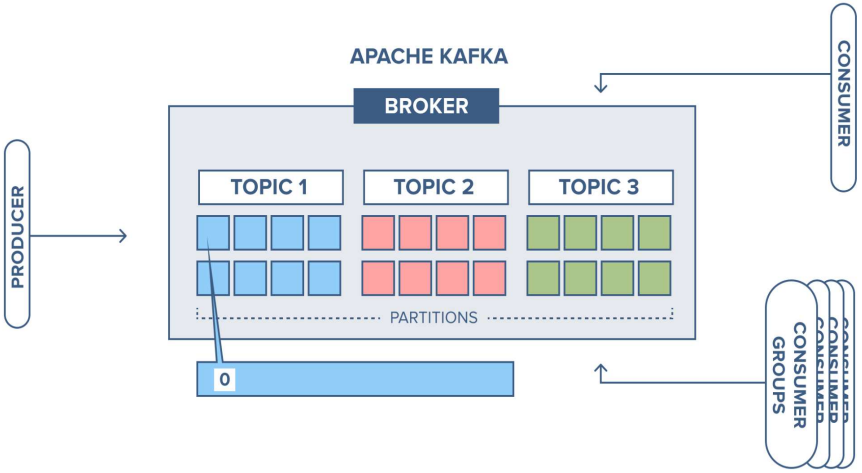
Real-World Example: Zoom

- **Data Capture:** Captures audio, video, and chat data from participants.
- **Compression:** Compresses streams to optimize for bandwidth.
- **Streaming Servers:** Use SFU architecture to minimize latency.
- **Scaling:** Scales dynamically using cloud servers to accommodate user load.

Kafka for Stream Processing and Analytics

- **Use Case:** Processing and analyzing continuous streams of data for real-time insights, anomaly detection, and predictive analytics.
- **Example:** Financial institutions use Kafka to process market data feeds, detect trading anomalies, and make real-time trading decisions. Retailers analyze customer behavior and preferences in real-time to offer personalized recommendations and promotions.

Ref: https://medium.com/@psnavya90/



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

SE ZG583, Scalable Services Lecture No. 3

Agenda

Popular scaling approaches

Managing high volume transactions

- Service Replicas & load balancing
- Minimizing event processing: Command Query Responsibility Segregation (CQRS)
- Asynchronous communication
- Caching techniques: Distributed cache, global cache

Scalability features in the Cloud (AWS, Azure, Google)

- Auto-scaling
- Horizontal and vertical scaling
- Use of Load balancers
- Virtualization
- Serverless computing
- Best Practices for Achieving Scalability

Case Study

Managing high volume transactions

Introduction to replication

- Replication is the practice of keeping several copies in different places.
- These replicas help distribute the workload and ensure that failures in one instance do not bring down the entire system.
- What is the need of replication?
  - High Availability
  - Fault Tolerance
  - Load sharing

Service Replicas

- It is a copy of the service logic that runs on one or more nodes of the cluster.
- Types of replicas: These replicas can be categorized based on their deployment model, scaling approach, and management strategy.
  - Stateful
  - Stateless
  - Active-Passive
  - Active-active
  - Leader-follower

Active-Active and Active-Passive Replicas

- **Active-Active:** All replicas handle requests simultaneously in a distributed load-sharing model.
- **Use Case**: High-traffic applications needing high availability.

- **Active-Passive:** One primary instance handles traffic, while replicas remain on standby and become active in case of failure.
- **Use Case**: Database replication, disaster recovery systems.

Stateful and Stateless Replicas

- **Stateful Replicas**: Each replica maintains its own persistent state, requiring synchronization across instances.
- **Use Case**: Databases, caching services, distributed file systems.

- **Stateless Replicas**: Each replica operates independently and does not maintain any session-specific data.
- **Use Case:** API gateways, web servers, RESTful microservices.

Leader-Follower Replica

- A replication model where a **Leader (Primary)** node handles **write operations** and **Follower (Replica)** nodes handle **read operations**. If the Leader fails, a Follower is promoted as the new Leader.
- Usecases: Databases, Caching system, Message Brokers

What are load balancers?

- A load balancer is a software or hardware device that keeps any one server from becoming overloaded.
- A load-balancing algorithm is the logic that a load balancer uses to distribute network traffic between servers
- Types Based on Deployment Location:
  - Software-based load balancers
  - Hardware-based load balancers
  - Cloud Load Balancer

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.

Command Query Responsibility Segregation (CQRS)

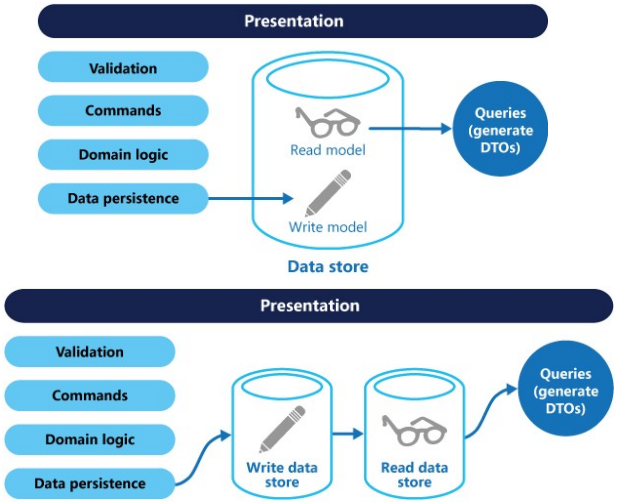
- It is a pattern that separates read and update operations for a data store.
- This pattern is particularly useful in **highly scalable and distributed applications**.

Why is Load Balancing Important for Scalability?

Load balancing ensures:

- Even distribution of requests across multiple instances.
- Reduced latency and improved response time by directing traffic to less loaded servers.
- Fault tolerance and high availability by rerouting traffic in case of server failures.
- Efficient resource utilization and better cost management.

How CQRS works?



Why CQRS for Scalability?

CQRS improves **scalability** by:

- Separating concerns
- Independent scaling
- Optimizing for reads and writes separately
- Supporting multiple databases
- Simpler queries

What is Event sourcing?

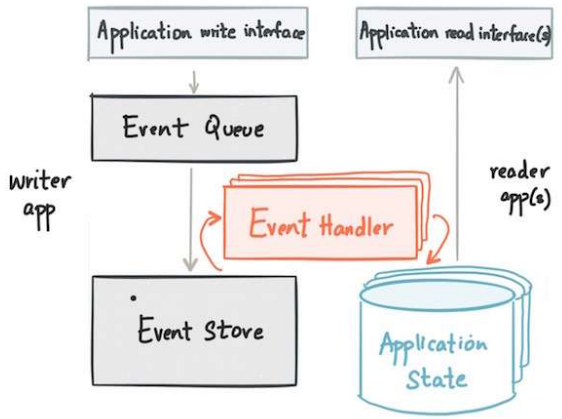
- **Event Sourcing** is an architectural pattern where **state changes** in a system are stored as a **sequence of events** instead of modifying the current state directly.

https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/

Some challenges of implementing CQRS

- Complexity
- Messaging
- Eventual consistency

Event Sourcing and CQRS



https://mrwersa.medium.com/cqrs-pattern-with-kafka-streams-part-1-112f381e9b98

How Event Sourcing and CQRS Work Together

- **Write operations (Commands)** → Generate **immutable events** instead of updating the database directly.
- **Events are stored** in an **event store** (e.g., Kafka, EventStoreDB).
- **Read operations (Queries)** → Use a **separate, read-optimized database** that is updated asynchronously.

Architecture of CQRS with Event Sourcing

A. Command Side (Write Model)

- Handles create, update, delete operations.
- Instead of updating a relational DB, writes generate events (e.g., OrderPlaced).
- Events are stored in an event store and published to event consumers.

B. Event Store

- A log of all past events, acting as the source of truth.
- Example storage: Kafka, EventStoreDB, PostgreSQL (JSONB), DynamoDB Streams.

.

C. Query Side (Read Model)

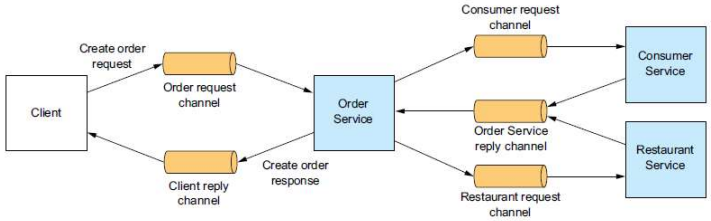
- Events from the event store are processed to update a read-optimized database.
- Read models can use SQL (denormalized tables), NoSQL (MongoDB, Redis), or search engines (Elasticsearch).

D. Event Handlers & Projections

- As events are stored, event handlers process them to update projections (optimized views for queries).
- Example: A UserRegistered event updates a UserProfile table in a read database

Asynchronous Communication

- Services communicating by exchanging messages over messaging channels.



Protocols for communication

In **distributed systems,** communication between components plays a crucial role in **scalability, performance, and reliability**.

Two main communication patterns are:

- **Synchronous Communication**: The sender waits for a response before proceeding.
- **Asynchronous Communication**: The sender does not wait for an immediate response, improving system efficiency.

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

Synchronous Vs Asynchronous

Feature	Synchronous (Blocking)	Asynchronous (Non-Blocking)
Use Case	Request-response APIs	Event-driven systems, messaging
Latency	Higher due to waiting	Lower, as operations continue
Scalability	Limited by concurrent requests	High scalability
Resilience	Prone to failures/timeouts	More resilient to failures
Complexity	Easier to implement	More complex (requires event handling)
Examples	REST APIs, RPC calls	Kafka, RabbitMQ, WebSockets

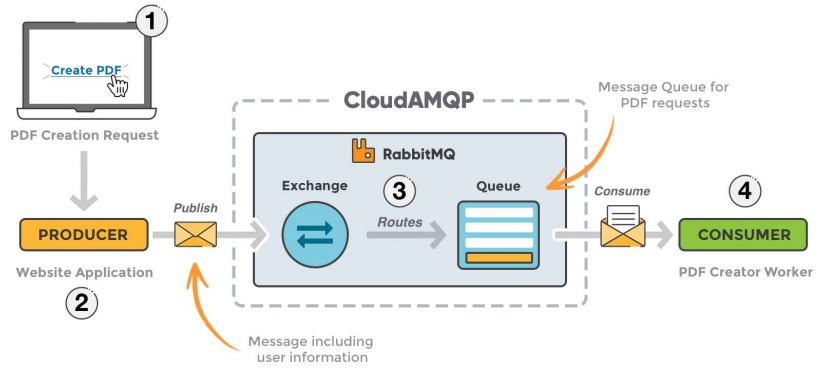
Benefits of Message Broker

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

Drawbacks of Message Broker

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

Example



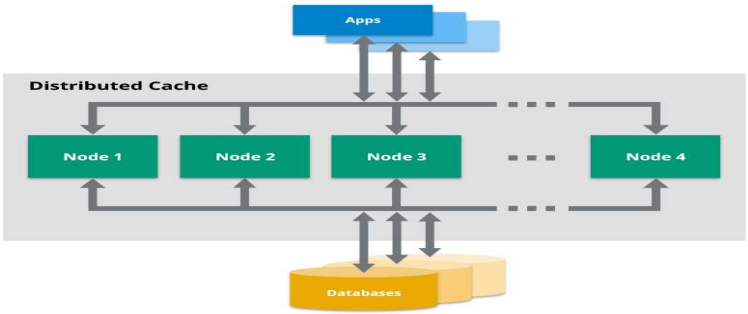
https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html

What is Caching?

- Caching is a critical technique in **distributed systems** to **reduce latency, improve performance, and optimize resource usage**.
- Caching allows you to efficiently reuse previously retrieved or computed data.

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.

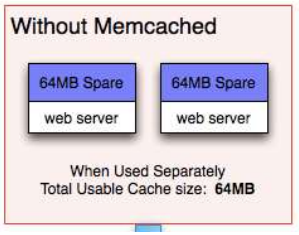


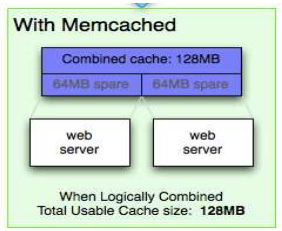
Types of cache

Different types of caching strategies are used depending on system architecture, data consistency needs, and workload patterns.

- Server side
- Client side
- Application-Level Cache
- Distributed Cache

Example: Memchached





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.

Google Global Cache (GGC)

- **Google Global Cache (GGC)** is a specialized caching system deployed by **Google in Internet Service Providers' (ISPs) networks** to improve the delivery of Google services such as:
  - YouTube
  - Google Search
  - Google Drive
  - Google Play Store
- GGC is part of **Google's Content Delivery Network (CDN)**, designed to **reduce bandwidth costs and improve access speed** by caching content **closer to users**.

Global Caches

- **Global Cache** refers to a caching mechanism that operates across multiple geographic regions or data centers, ensuring faster data access and improved system scalability.

Key Characteristics of Global Cache:

- ✓ Distributed Across Multiple Regions
- ✓ Reduces Latency
- ✓ Load Balancing
- ✓ High Availability

Scalability features in the Cloud

Auto-scaling

- Auto Scaling is a technique used in **cloud computing and distributed systems** to **dynamically adjust computing resources** based on real-time demand.

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

Types of Auto 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

What is virtualization?

- Virtualization is a technology that allows multiple **virtual instances** (e.g., virtual machines, containers) to run on a single **physical machine** by abstracting hardware resources.

Types of virtualization

- Server virtualization
- Storage virtualization
- Network virtualization
- Data virtualization
- Application virtualization
- Containerization

……

What is Serverless Computing?

- Serverless computing is a cloud computing model where cloud providers automatically manage the infrastructure and dynamically allocate resources as needed.

How Virtualization Improves Scalability?

- On-Demand Resource Allocation
- Efficient Load Balancing
- Multi-Tenancy Support
- High Availability & Fault Tolerance
- Cost-Effective Scaling

How Serverless Computing Enhances Scalability?

- Auto Scaling Based on Demand
- Instant Scaling and Elasticity
- Cost Efficiency
- Statelessness
- Managed Load Balancing

Key Characteristics of Serverless Computing

- No Server Management
- Event-Driven
- Automatic Scaling
- Pay-as-You-Go

Types of Serverless Architectures for Scalability

- **Function-as-a-Service (FaaS):** developers write individual functions that are executed in response to specific events
- **Backend-as-a-Service (BaaS):** use of managed backend services that are provided by cloud vendors
- **Serverless Containers:** allow the abstraction of infrastructure management while using containerized applications.

Challenges of Serverless Computing for Scalability

- ✗ Cold Start Latency
- ✗ State Management
- ✗ Vendor Lock-In
- ✗ Debugging and Monitoring

Best Practices for Achieving Scalability

- Be Stateless
- Use Load Balancers
- Auto Scaling
- Optimize Database Performance
- Use Caching Mechanisms
- Microservices Architecture
- Event-Driven Architecture
- Use Serverless Computing
- Monitor and Analyze Performance

By following these best practices, you can build highly scalable applications that handle increased demand without compromising performance or reliability.

Case Study: Amazon Prime

- Amazon Prime Video Uses AWS to Deliver Solid Streaming Experience to More Than 18 Million Football Fans
- Amazon Prime needed an architecture that could quickly scale, handle spikes, and have sufficient caching in different layers to manage the demand.
  - Scalable Cloud Infrastructure
  - Content Delivery with Amazon CloudFront
  - High-Performance Video Encoding and Storage
  - Serverless Technology
  - Real-Time Analytics with Amazon Kinesis
  - Global Availability and Reliability
  - Enhanced Security and Compliance

https://aws.amazon.com/solutions/case-studies/amazon-prime-video/

References

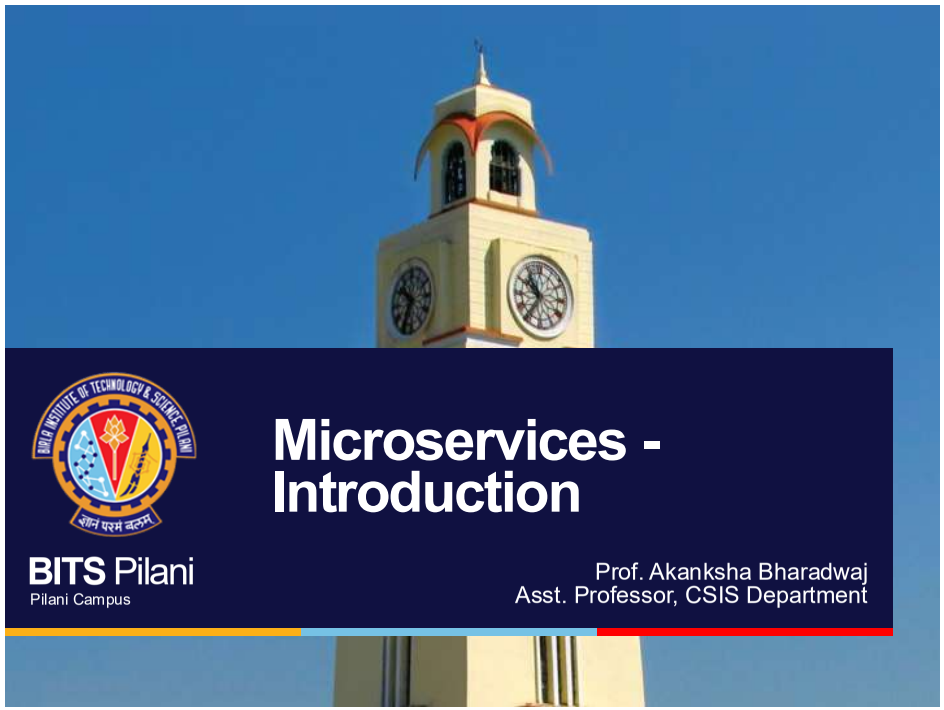
- Text Books
- Book: The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise, Second Edition by Michael T. Fisher; Martin L. Abbott Published by Addison-Wesley Professional, 2015
- https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-concepts-replica-lifecycle
- 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
- https://www.confluent.io/blog/event-sourcing-cqrs-stream-processing-apache-kafka-whats-connection/
- https://aws.amazon.com/solutions/case-studies/amazon-prime-video/

Self Study

- Amazon Prime case study: https://aws.amazon.com/solutions/case-studies/amazon-prime-video/
- Hotstar case study: https://laveena-j-21.medium.com/cloud-computing-aws-and-disney-hotstar-case-study-f4a3be4669a
- Article on serverless: https://www.simform.com/blog/serverless-architecture-guide/

BITS Pilani
Pilani Campus

SE ZG583, Scalable Services
Lecture No. 4



Microservices - Introduction

BITS Pilani
Pilani Campus

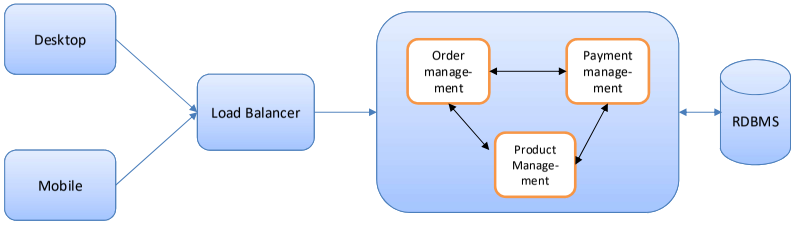
Prof. Akanksha Bharadwaj
Asst. Professor, CSIS Department

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

Online shopping



Advantages of Monolithic

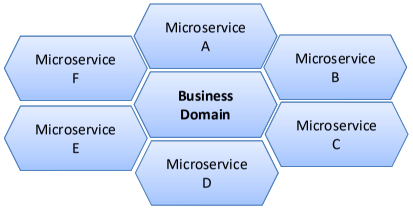
- Simplicity
- Easier to Debug and Troubleshoot
- Easier Performance Optimization
- Easy Data Management

Disadvantages of Monolith

- Scalability Challenges
- Limited Technology Flexibility
- Development Bottlenecks
- Slow & Risky Updates
- Harder Maintenance in Large Systems

What is Microservices?

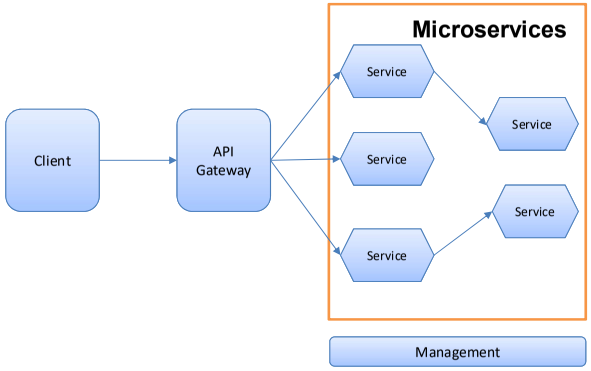
- Microservices are independently deployable services modeled around a business domain.
- They communicate with each other via networks,
- Each microservice can focus on a single business capability



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?

Example Architecture



Main characteristics of microservices

- Independent Deployment
- Decentralized Data Management
- Scalability
- Technology Diversity
- Fault Isolation
- Lightweight Communication

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.

Case Study

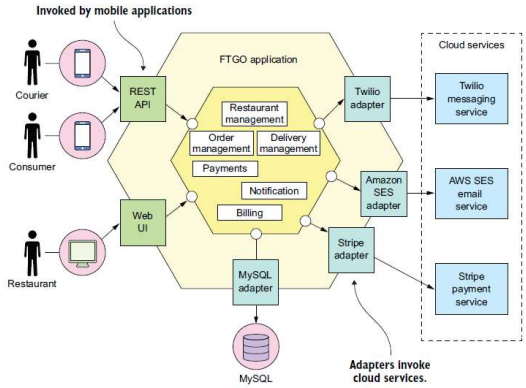
SOA vs Microservices: Which is best for you?

- 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.

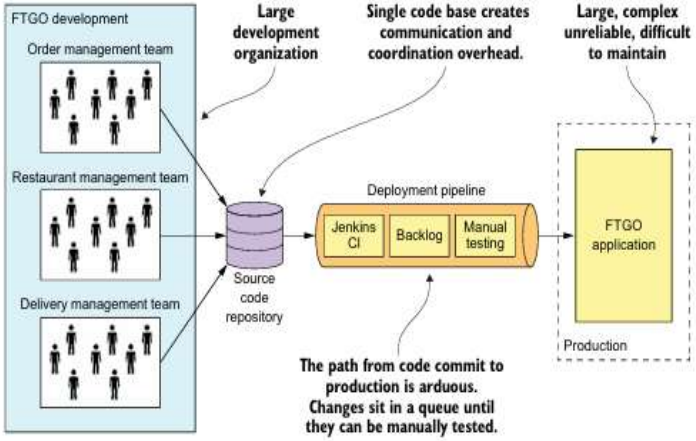
FTGO Case Study

- 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



Monolithic Hell

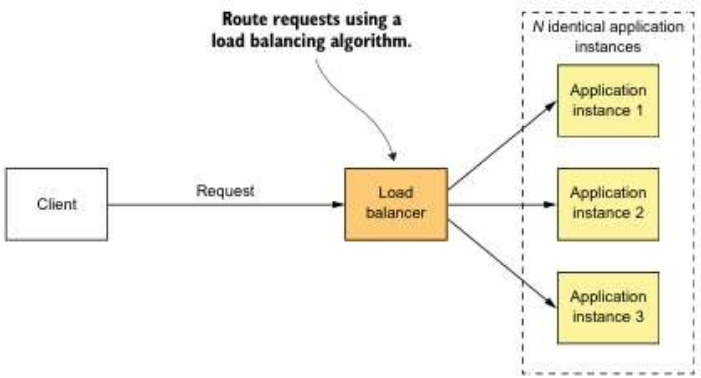


Problems faced in old FTGO architecture

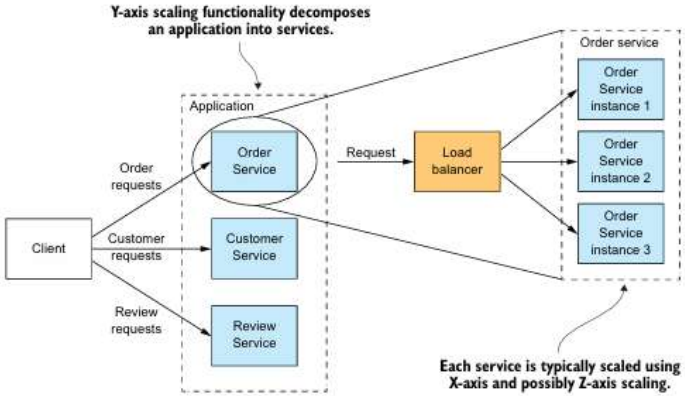
- Complexity
- Slow development and deployment cycle
- Scaling is difficult
- Delivering a reliable monolith is challenging

Possible solution

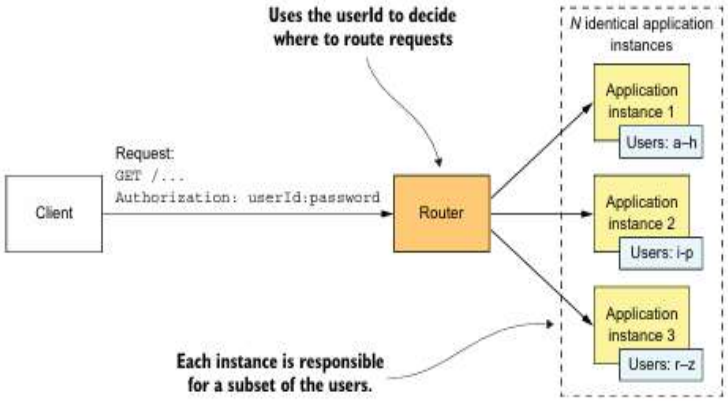
X- axis scaling



Y-axis scaling



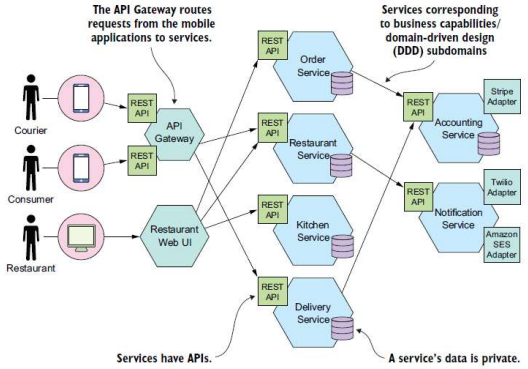
Z-axis scaling



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?

Microservices Architecture for FTGO

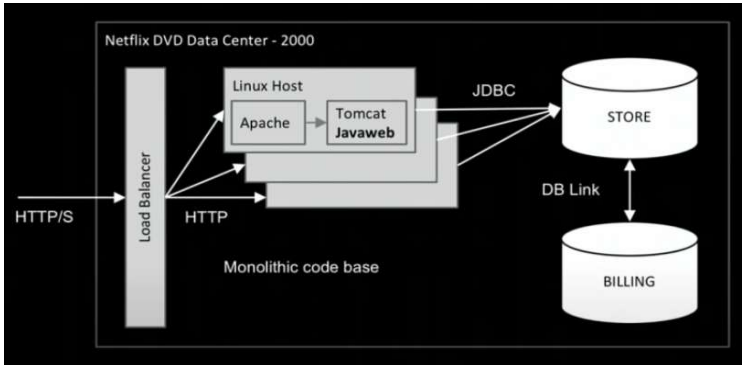


Challenges in previous architecture

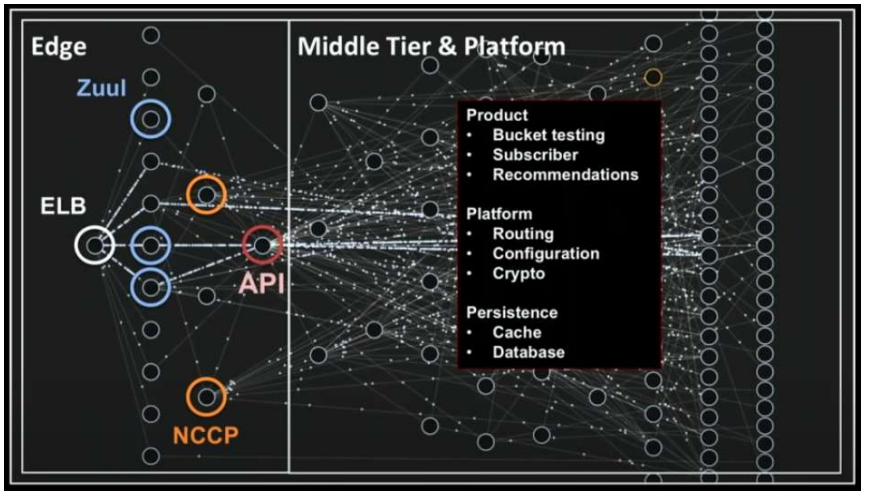
- Monolithic Code base
- Monolithic Database
- Tightly coupled Architecture

How Netflix worked earlier?

Netflix Architecture earlier



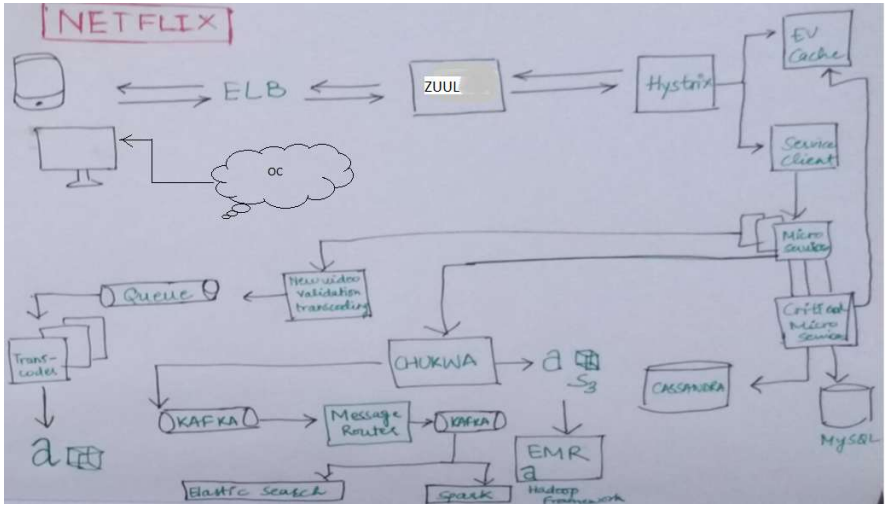
Updated Netflix architecture



What they need in new Architecture?

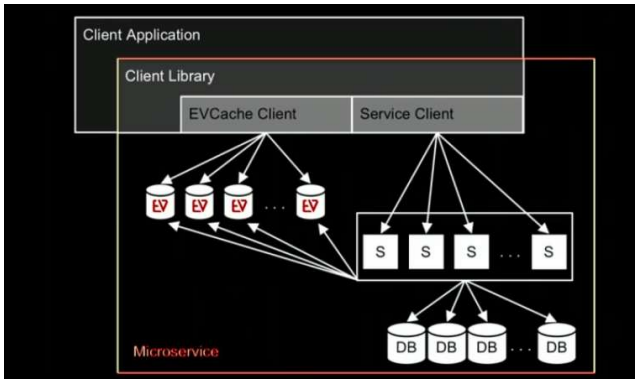
- Modularity and encapsulation
- Scalability
- Virtualization and Elasticity

Latest Netflix Architecture



Netflix Approach

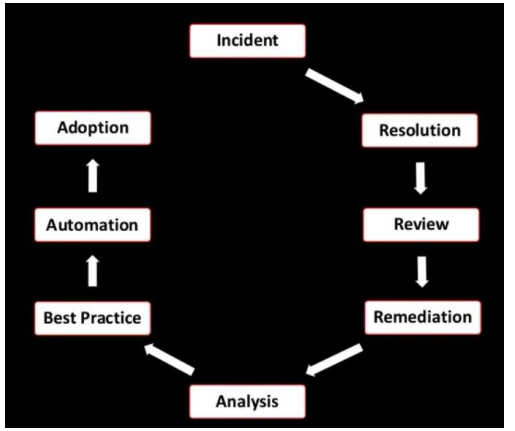
Microservice at Netflix



Self Study

- Journey from monolithic to SOA to Microservices: https://www.linkedin.com/pulse/evolving-architecture-journey-from-monolithic-soa-avita-katal/
- Uber case study – Read about Domain Oriented Microservices Architecture. Link: https://eng.uber.com/
- Github journey: https://www.infoq.com/articles/github-monolith-microservices/

Netflix follows continuous learning



Microservices Contd.

BITS Pilani
Pilani Campus

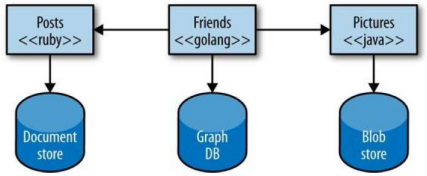
Akanksha Bharadwaj
Asst. Professor, CSIS DEpartment

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

Advantages of Microservices

- Technology Heterogeneity
  - Different microservices can use different programming languages, databases, or frameworks.
  - Teams can choose the best technology for each service.



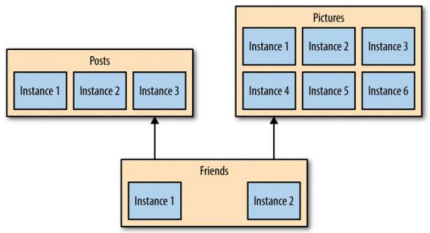
SE ZG583, Scalable Services
Lecture No. 5

Advantages contd.

- Better Maintainability & Modularity
- Enhanced Security
- Support for DevOps & Agile Methodologies

Advantages contd.

- Resilience
- Faster Development & Deployment
- Scalability



Amazon Prime Video's Transition

- Amazon's Prime Video team initially adopted a microservices architecture utilizing serverless components like AWS Lambda and Step Functions to manage their video streaming services. However, they encountered significant challenges
- To address these issues, the team consolidated their services into a monolithic application hosted on Amazon EC2 and ECS.
- It's essential for organizations to carefully assess their specific needs and constraints when deciding between microservices and monolithic architectures.

Microservices is not a silver bullet

- Increased Operational Complexity
- Inter-Service Communication Overhead
- Data Consistency & Distributed Transactions
- Security Risks
- Higher Infrastructure & Maintenance Costs
- Steep Learning Curve
- Not Always the Best Fit

Process and Organization

Introduction

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

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.

Microservices Design Principles

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

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

Single Responsibility Principle (SRP)

- 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

Loose Coupling & High Cohesion

- Services should be loosely coupled to minimize dependencies.
- High cohesion ensures that each service is self-contained and modular.

Resilience & Fault Tolerance

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

Observability & Monitoring

- Implement logging, monitoring, and tracing (e.g., Prometheus, ELK Stack).
- Enable distributed tracing to track request flows across microservices.

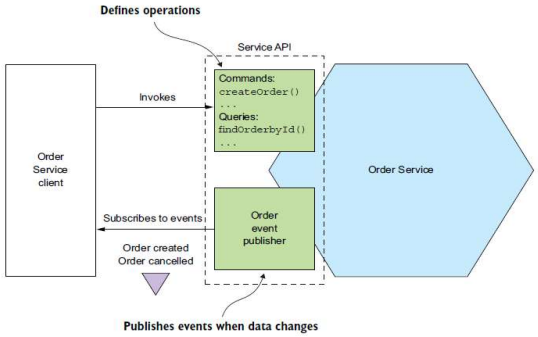
Statelessness

- Services should be stateless and avoid storing session data.
- Use external caching mechanisms (e.g., Redis, Memcached) for performance optimization.

Important concepts

What is a Service?

- It is a standalone, independently deployable software component that implements some useful functionality



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.

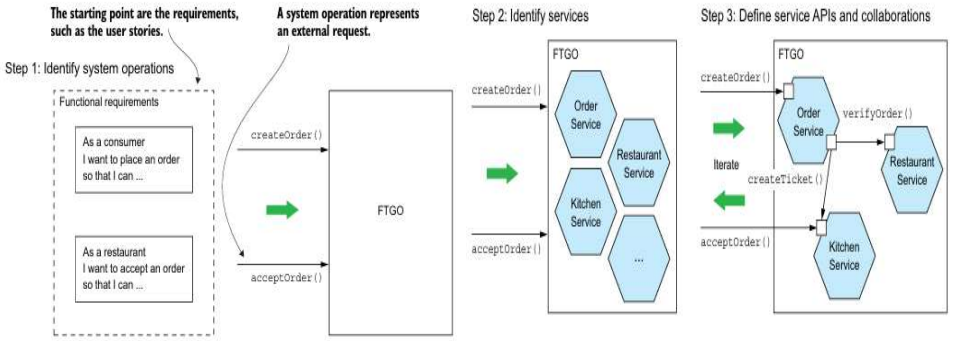
Size of a Service

- Each service should be small enough to be independently developed, deployed, and scaled, yet large enough to encapsulate a meaningful business capability.

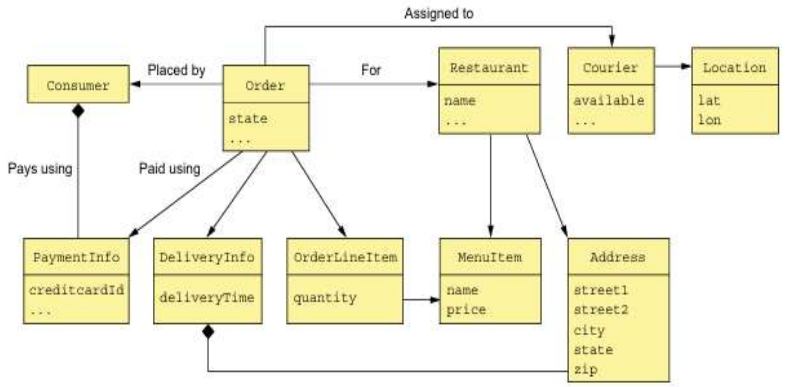
✓ **Too Small → Increased Complexity:** Leads to too many inter-service calls, higher network latency, and dependency issues.
✓ **Too Large → Monolithic Behavior:** Reduces scalability, reusability, and independent deployments.
✓ **Right Size → Business-Oriented, Autonomous, Scalable**

Steps for defining an application's microservice architecture

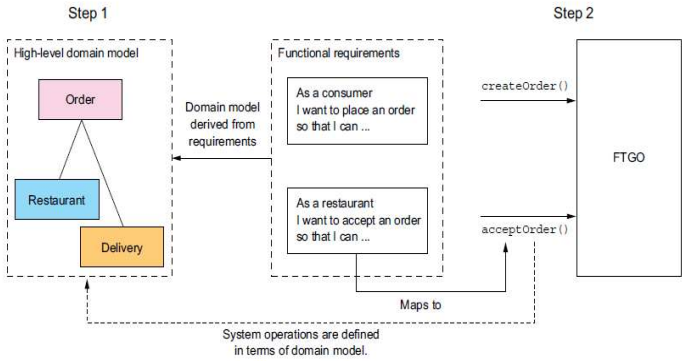
Step 1: Identify system operations
Step 2: Identify services
Step 3: Define service APIs and collaborations



FTGO domain model



Step1: Identify system operations



Step 2 and 3: Defining services and service API

- One strategy is defining services by applying the Decompose by business capability pattern and another is defining services by applying the Decompose by sub-domain pattern

- The next step is to define each service's API: its operations and events.
- 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
- The starting point for defining the service APIs is to map each system operation to a service.

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.

Defining system operations

There are two types of system operations:

- Commands—System operations that create, update, and delete data
- Queries—System operations that read (query) data

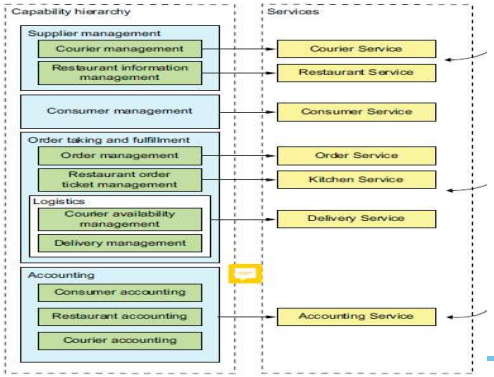
Actor	Story	Command	Description
Consumer	Create Order	`createOrder()`	Creates an order
Restaurant	Accept Order	`acceptOrder()`	Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time

Operation	`createOrder` (consumer id, payment method, delivery address, delivery time, restaurant id, order line items)
Returns	`orderId`, ...
Preconditions	▪ The consumer exists and can place orders. ▪ The line items correspond to the restaurant's menu items. ▪ The delivery address and time can be serviced by the restaurant.
Post-conditions	▪ The consumer's credit card was authorized for the order total. ▪ An order was created in the `PENDING_ACCEPTANCE` state.

Decomposition based patterns to define services

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



Identifying Business Capabilities

Business capabilities for FTGO include the following:

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

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.

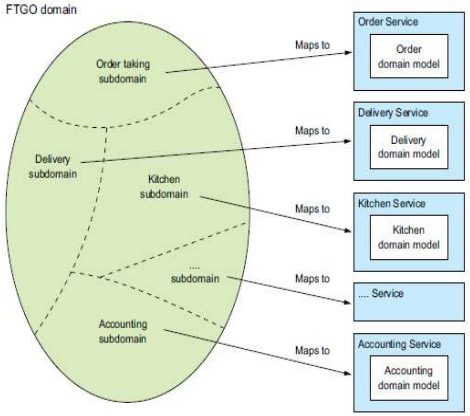
Decompose by sub-domain pattern

- 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**.

Decomposition guidelines

- Single Responsibility Principle
- Common Closure Principle

Decompose by sub-domain pattern



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
- Amazon Prime Case Study: https://theodore.ie/the-monolith-returns-why-companies-are-moving-away-from-microservices-and-serverless-architecture/

Self Sudy

Article: https://kitrum.com/blog/is-microservice-architecture-still-a-trend/

Article: https://devops.com/microservices-amazon-monolithic-richixbw/

SE ZG583, Scalable Services
Lecture No. 6

Defining service APIs

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 services preventing decomposition

Key Principles of Service APIs

- **Loose Coupling** – Minimize dependencies between services.
- **High Cohesion** – Each API should be specific to a microservice's functionality.
- **Standardized Communication** – Use REST, gRPC, GraphQL, or event-driven messaging.
- **Backward Compatibility** – Ensure versioning to prevent breaking changes.
- **Security** – Implement authentication, authorization, and encryption.

Introduction

A service API operation exists for one of two reasons:

- **To Expose Business Capabilities** – The operation provides functionality that aligns with a business use case, such as processing orders, managing users, or retrieving product details.

- **To Support System Integration** – The operation facilitates interaction between services, allowing them to communicate, share data, and maintain consistency across the system.

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()

Assigning system operations to services

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

We can either

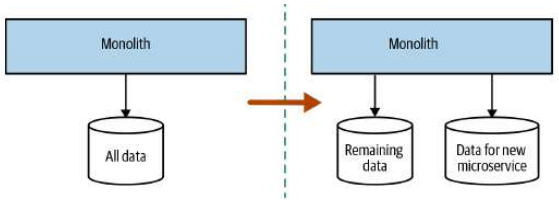
- Assign an operation to a service that needs the information provided by the operation

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

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()	—

Split the Database First

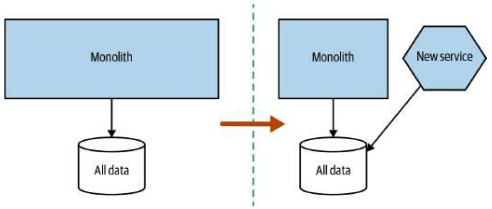


Determining the APIs required to support collaboration between services

- Some system operations are handled entirely by a single service
- Some system operations span across multiple services.

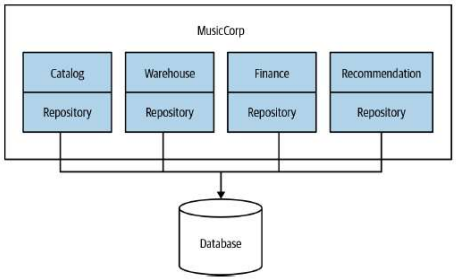
Transition from monolith to microservices

Split the Code First

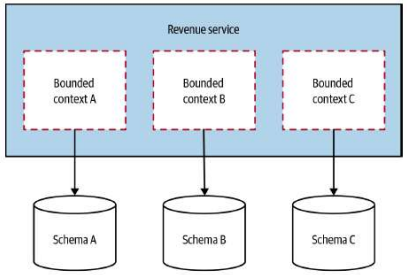


Split the Database First

Pattern: Repository per bounded context



Pattern: Database per bounded context

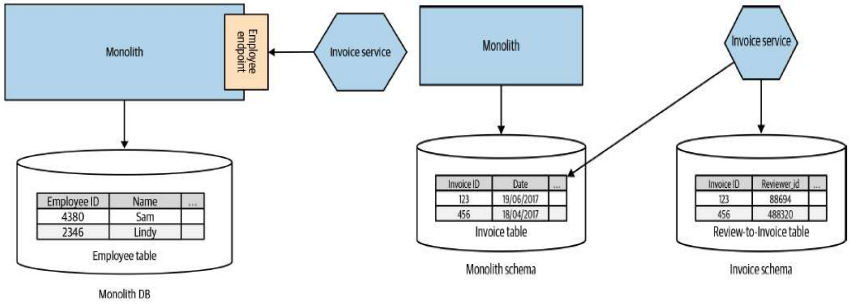


Patterns for Monolith to Microservices

Split the Code First

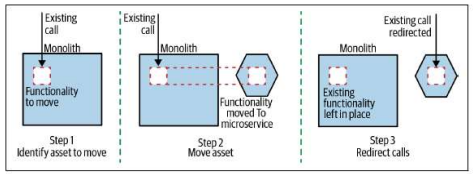
Pattern: Monolith as data access layer

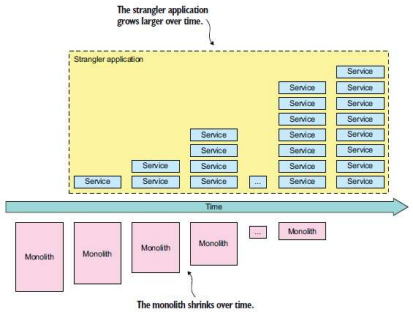
Pattern: Multi-schema storage



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





Rebuild From Scratch

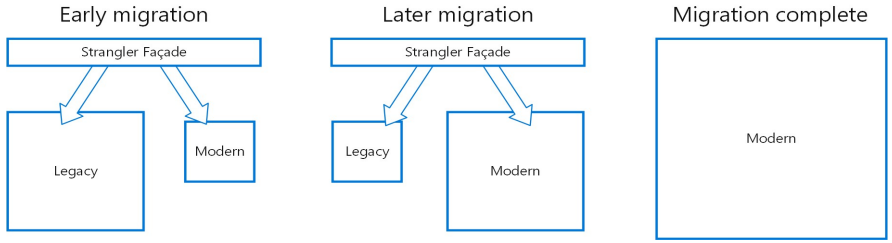
- One of the biggest challenges for us was having a good understanding of the legacy system.
- Can not use the system until complete
- Longer duration required

Strangler Pattern: Issues

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

Strangler Pattern

Steps involved in transition



Communication Protocols

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.

Aspects of communication

Communication Type

- **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.

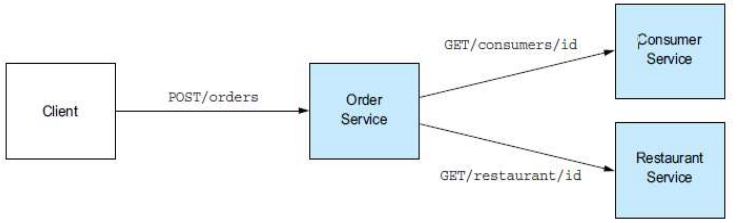
Number of Receivers

- Single receiver
- Multiple receivers

Synchronous communication

- REST is an extremely popular IPC mechanism under this category

Example: FTGO CreateOrder request



Guiding Principles of REST

- Resource-Oriented Design
- Stateless
- Cacheable
- Uniform interface
- API Versioning
- Service Discovery & Load Balancing

Representational state transfer (REST)

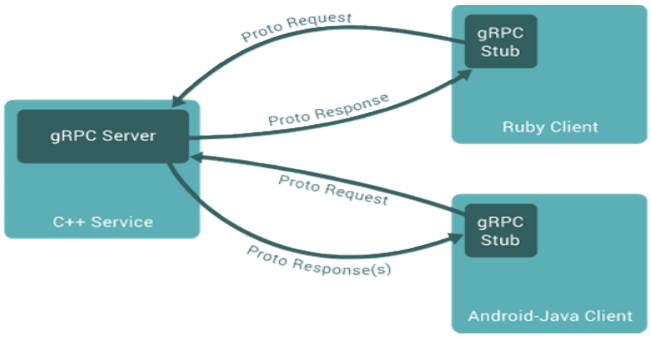
- It is a software architectural style that was created to guide the design and development of the architecture for the World Wide Web.

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: Introduction

gRPC (Google Remote Procedure Call) is a high-performance, language-agnostic RPC (Remote Procedure Call) framework that is well-suited for microservices-based architectures



Why use gRPC in Microservices?

1. High Performance & Efficiency

- Uses Protocol Buffers (Protobuf), a compact, binary format that is faster and smaller than JSON over HTTP.
- Uses HTTP/2 for multiplexed connections, reducing latency and improving efficiency.

3. Supports Streaming

Unlike REST, gRPC supports real-time communication using:

- Unary RPC (Single request-response, like REST)
- Server Streaming (Server sends multiple responses for a single request)
- Client Streaming (Client sends multiple requests and gets a single response)
- Bi-directional Streaming (Both client and server send multiple messages in real-time)

2. Strongly Typed Contracts

- Enforces a strict schema using .proto files, ensuring consistency between services.
- Auto-generates client and server code in multiple languages, reducing manual coding errors.

4. Language-Agnostic

- gRPC supports multiple programming languages (**Java, Python, Go, C++, etc.**), making it ideal for polyglot microservices environments.

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

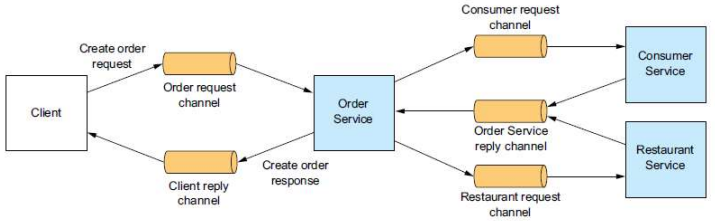
Communication Styles

5. Built-in Authentication & Security

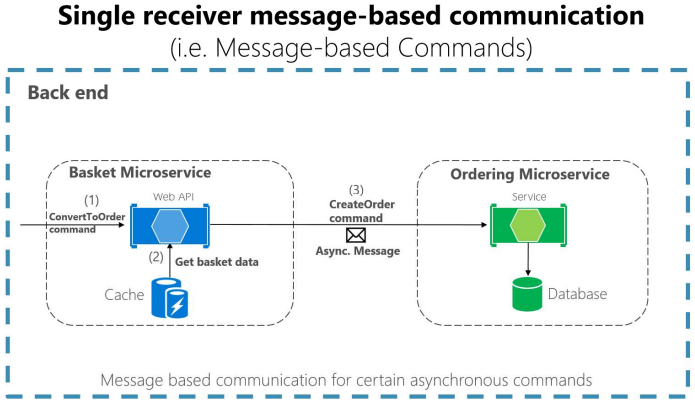
- Supports **TLS encryption** for secure communication.
- Works with authentication mechanisms like **OAuth 2.0 and JWT**.

Asynchronous Communication Example

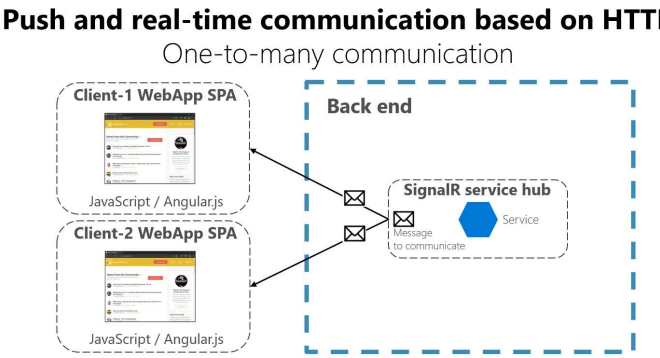
- Services communicating by exchanging messages over messaging channels.



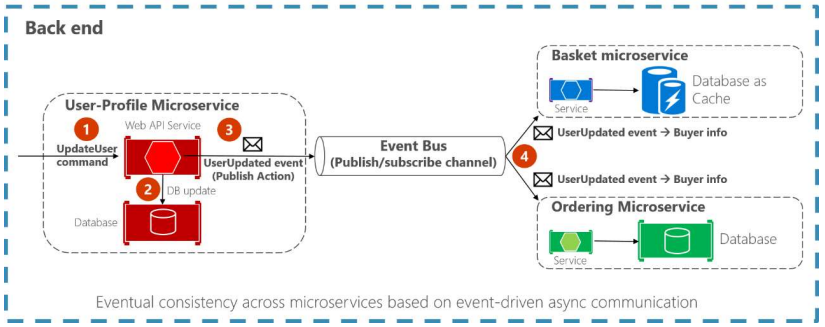
Single-receiver message-based communication



Push and real-time communication based on HTTP



Asynchronous event-driven 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.

Choosing the Right Communication Pattern for Microservices

REST API (Synchronous, Request-Response)

Example Scenario: User Authentication & Profile Management

- A user logs into an **e-commerce platform** using email and password. The system validates the credentials and returns a response.
- Similarly, fetching user details (name, address, order history) should be a simple request-response operation.

gRPC (High-Performance Remote Procedure Calls)

Example Scenario: Ride-Sharing App's Driver Location Tracking

- A passenger requests a ride, and the system continuously updates the **driver's real-time location** every few seconds. Since performance and efficiency are crucial, gRPC's low latency and streaming features make it ideal.

Kafka (Event-Driven, Asynchronous Messaging)

Example Scenario: Order Processing in an E-Commerce System

- When a user places an order, multiple microservices need to react to this event:
  Order Service → Saves order details.
  Inventory Service → Updates stock levels.
  Payment Service → Charges the user.
  Notification Service → Sends confirmation emails/SMS.
- Kafka acts as an **event bus** that ensures all these services process the event without direct coupling.

WebSockets (Persistent, Real-Time Bi-Directional Communication)

Example Scenario: Stock Market Price Updates

- A stock trading platform needs to show **real-time stock price changes** to users without them having to refresh the page.

RabbitMQ (Message Queuing, Reliable Delivery)

Example Scenario: Asynchronous Task Processing in a Banking System

- A banking system needs to process transactions in a **fraud detection microservice**. Instead of waiting synchronously, the **Transaction Service** places a message in a queue, and the **Fraud Detection Service** processes it asynchronously.

Comparison

Feature	REST	gRPC	Kafka	RabbitMQ	WebSockets
Type	Request-Response	Remote Procedure Call	Event Streaming	Message Queue	Persistent Connection
Best For	Web APIs, CRUD	Microservices, Low Latency	High-Volume Events	Reliable Async Tasks	Real-Time Updates
Data Format	JSON/XML	Protobuf	Binary	Any	JSON/Binary
Communication	Synchronous	Synchronous/Streaming	Asynchronous	Asynchronous	Full Duplex
Browser Support	✅ Yes	❌ No	❌ No	❌ No	✅ Yes
Throughput	Medium	High	Very High	High	Medium
Latency	High	Low	Medium	Low	Very Low
Persistence	❌ No	❌ No	✅ Yes	✅ Yes	❌ No

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/

SE ZG583, Scalable Services
Lecture No. 7

Communication and Transaction management

Akanksha Bharadwaj
Asst. Professor, CSIS Department

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.

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

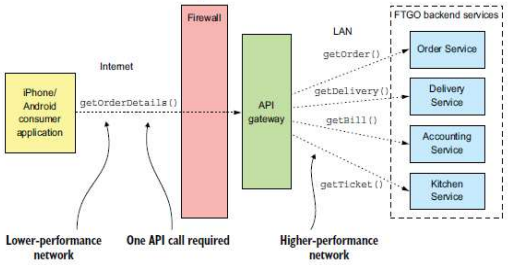
How the mobile client accesses these services?

Using an API Gateway: Request Routing

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

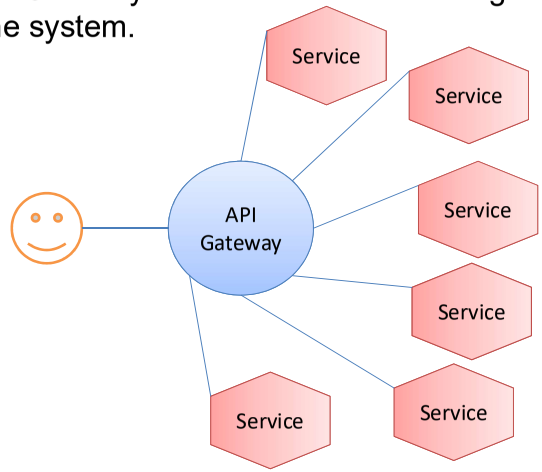
API Composition

API Gateway might implement some API operations using API composition.

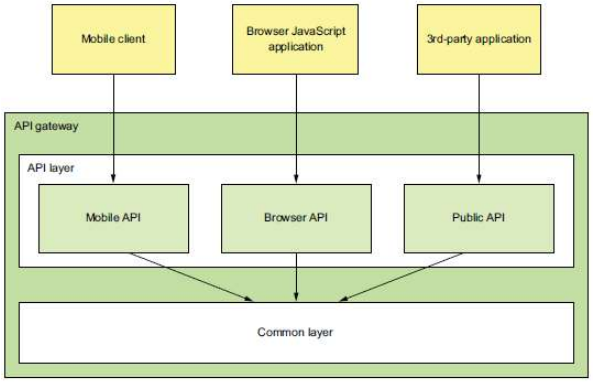


Using an API Gateway

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



API Gateway Architecture



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

Drawbacks of an API Gateway

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

Benefits of an API Gateway

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

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

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.

Good API design

- Simplicity
- Flexibility
- Consistency
- Scalability

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?

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.

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

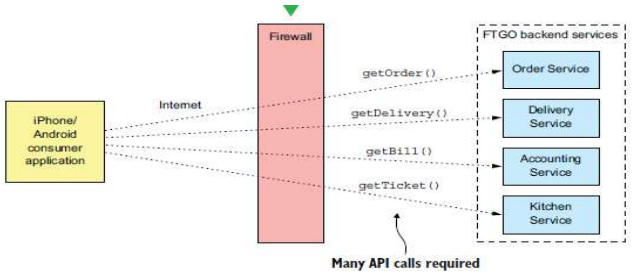


Figure 8.2 A client can retrieve the order details from the monolithic FTGO application with a single request. But the client must make multiple requests to retrieve the same information in a microservice architecture.

What is API security?

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

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.

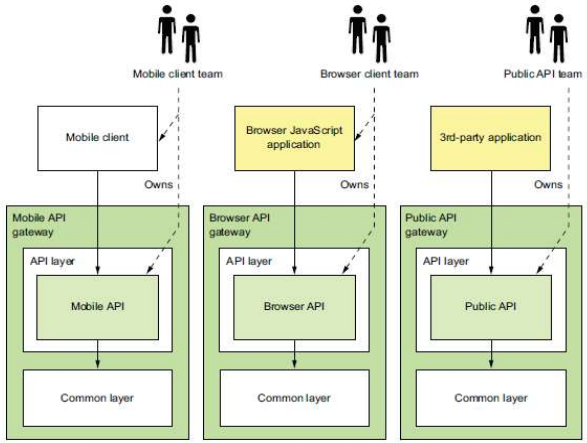
REST API security

- 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.

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.

Backends for Frontends



Database related patterns for Microservices

Most common API security best practices

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

Benefits of BFF pattern

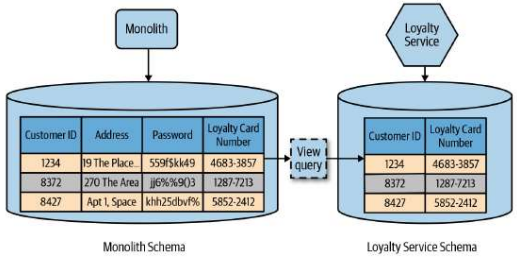
- The API modules are isolated from one another, which improves reliability.
- It also improves observability, because different API modules are different processes
- Another benefit of the BFF pattern is that each API is independently scalable.
- The BFF pattern also reduces startup time because each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

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.



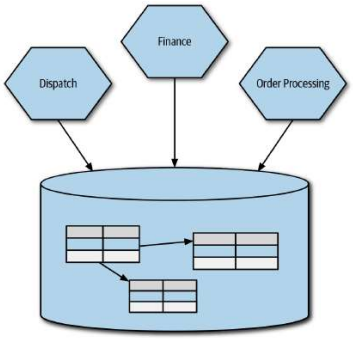
Shared Database pattern

Problems:

- Shared Vs hidden data
- Control on data

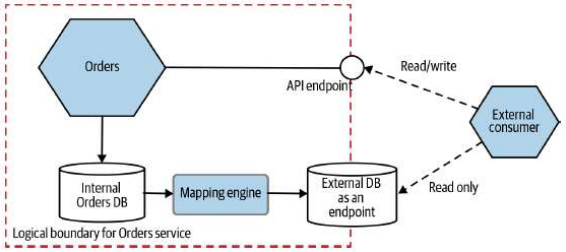
When to use:

- Read-only static reference data
- During transition



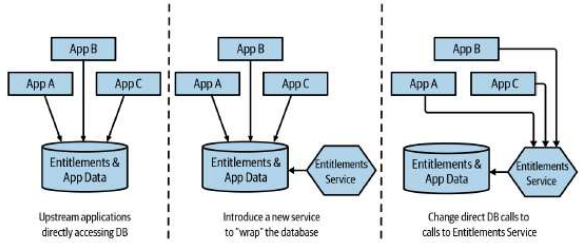
Database-as-a-Service Pattern

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



Database Wrapping Service Pattern

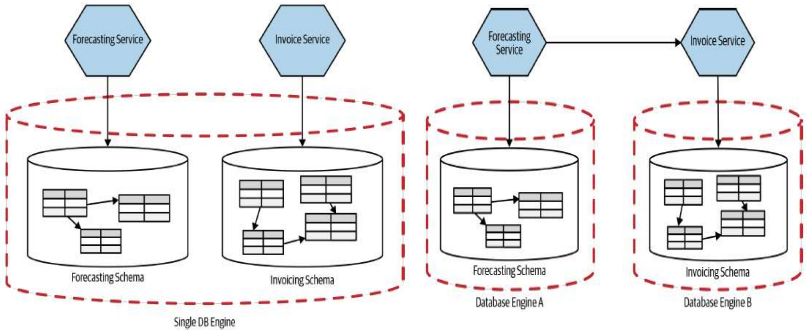
- Here we hide the database behind a service that acts as a thin wrapper



Transaction management in a microservice architecture

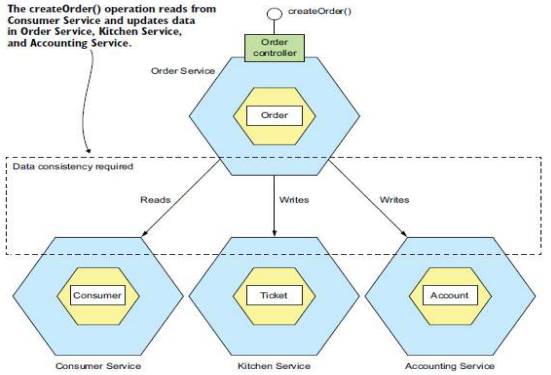
Splitting Apart the Database

Physical Versus Logical Database Separation



Challenges with distributed transactions

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



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

Solution: Saga pattern

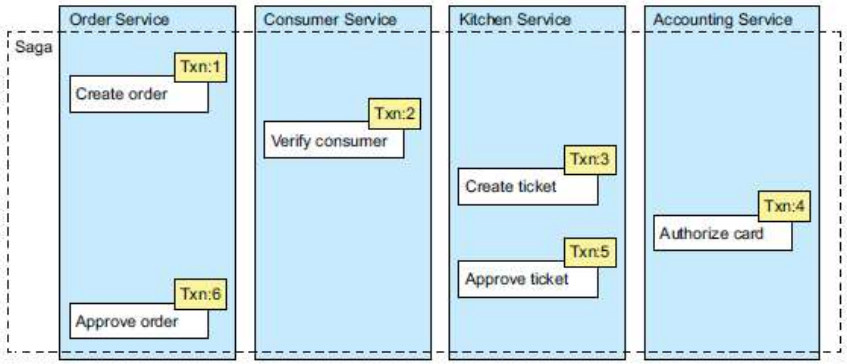
- Sagas are mechanisms to maintain data consistency in a microservice architecture without having to use distributed transactions.

- 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.

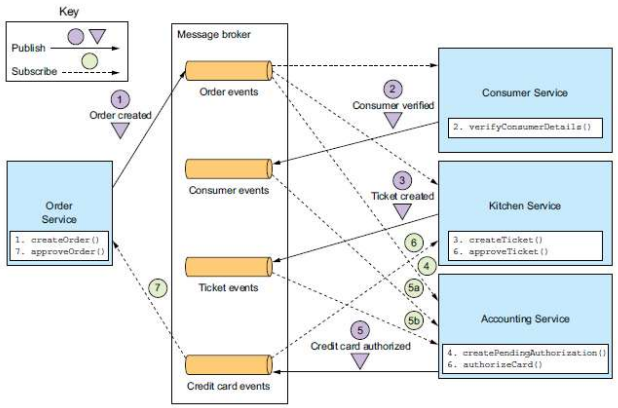
Challenges in Saga

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

Example

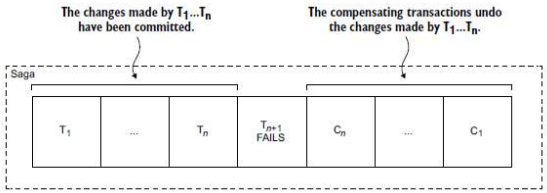


Create Order Saga



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()	—

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/
6. https://www.getambassador.io/blog/optimizing-microservices-architecture
7. https://www.postman.com/api-platform/api-design/

Self Study

- API in detail
- Sagas in detail

SE ZG583, Scalable Services
Lecture No. 8

Microservices Review

When to Use Hadoop

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

Review

Homework: Kafka

Compare and contrast Kafka's event streaming architecture with traditional message queuing systems. What advantages does Kafka provide in terms of scalability and fault tolerance?

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.

Aspect	Kafka Event Streaming	Traditional Message Queuing
Data Model	Log-centric, events stored in immutable logs.	Point-to-point or publish-subscribe messaging model.
Storage	Persists messages for a configurable retention period.	Messages are often transient and may be deleted after consumption.
Message Distribution	Publish-subscribe model with partitioned topics.	Follows either point-to-point or publish-subscribe patterns.
Scalability	Scales horizontally by adding more brokers and partitions.	Scales often involves vertical scaling by increasing resources.
Fault Tolerance	Achieves fault tolerance through replication of partitions.	Relies on mechanisms like clustering for fault tolerance.
Acknowledgments and Guarantees	Configurable delivery guarantees (at most once, at least once, exactly once).	Various acknowledgment and delivery guarantee mechanisms.
Use Cases	High-throughput, durability, real-time data pipelines, streaming analytics.	Point-to-point communication, traditional messaging scenarios.
Integration Ecosystem	Rich ecosystem with connectors for various data sources and sinks.	Often integrated with middleware and enterprise messaging systems.
Horizontal Scalability	Enabled through partitioning and adding more brokers.	Scaling may involve vertical scaling by increasing resources.

Edge computing scenarios

- Enterprise edge
- Operations edge
- Provider edge

When to use Load balancer

- Load Balancing for Scale
- High availability

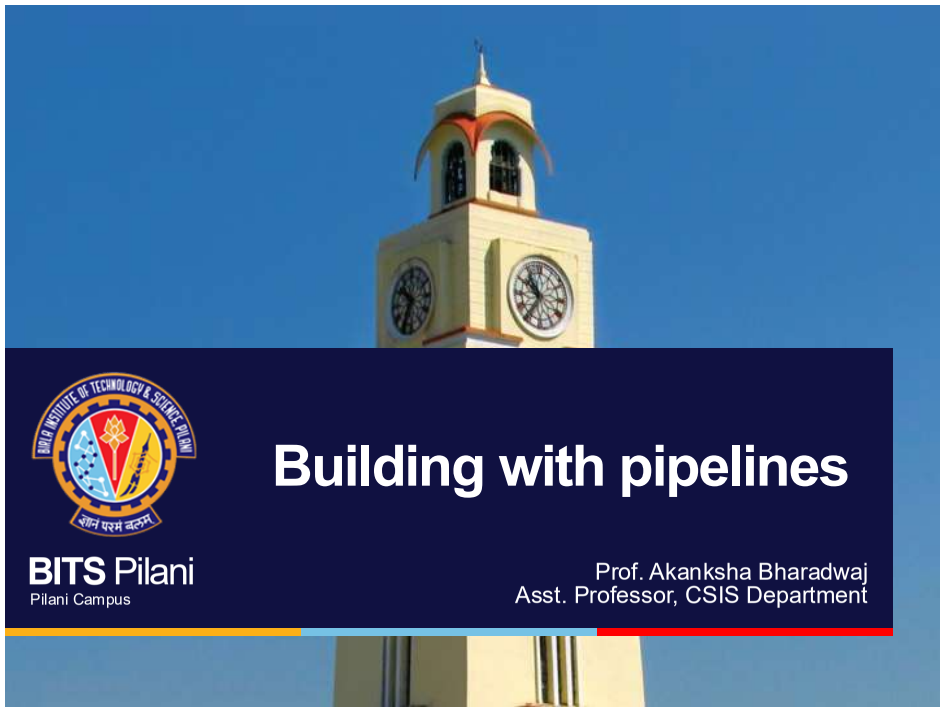
When serverless architecture is not the right choice

- 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.

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.

SE ZG583, Scalable Services
Lecture No. 9



DevOps

- DevOps is a **software development methodology** that integrates **development (Dev)** and **IT operations (Ops)** to improve collaboration, automate processes, and accelerate software delivery.

Agenda

Building with pipelines

- Continuous integration
- Tooling
- Repository patterns – Multi-repo, mono-repo

Continuous Integration

- Continuous integration is a DevOps software development practice where developers regularly merge their code changes into a central repository, after which automated builds and tests are run.
- As part of this process, we often create artifact(s) that are used for further validation, such as deploying a running service to run tests against it.
- To enable the artifacts to be reused, we place them in a repository of some sort, either provided by the CI tool itself or on a separate system.

Key Principles of DevOps

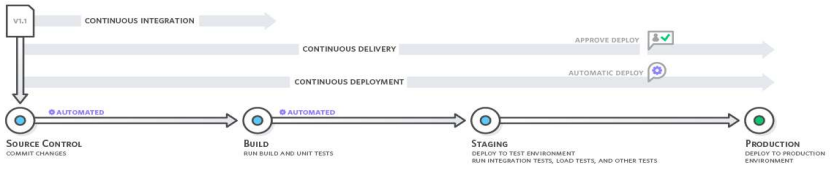
- Collaboration & Communication
- Automation
- Continuous Integration (CI)
- Continuous Delivery (CD)
- Monitoring & Feedback
- Security (DevSecOps)

Benefits of Continuous Integration

- Early Bug Detection
- Faster Release Cycles
- Improved Code Quality
- Reduced Integration Risks
- Efficient Collaboration

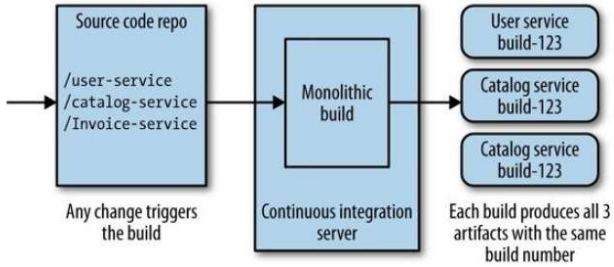
CI Workflow

- Code Commit
- Automated Build
- Automated Testing
- Feedback and Notifications
- Integration into Main Branch



Mapping Continuous Integration to Microservices

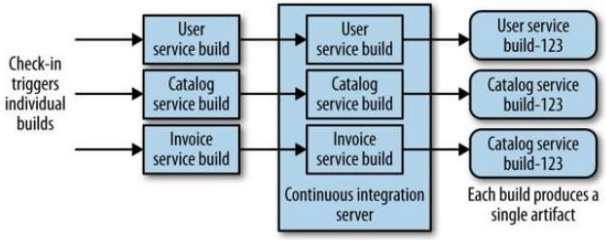
- The simplest option is we could lump everything in together. We will have a single, giant repository storing all our code, and have one single build.



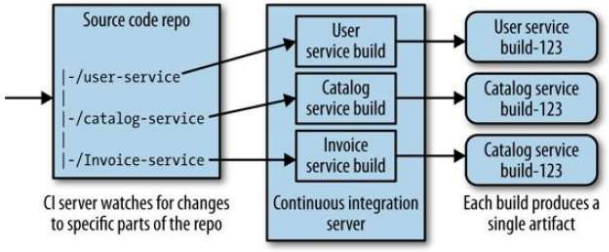
Continuous Integration for Microservices

- Continuous Integration (CI) in microservices ensures that each service is built, tested, and integrated frequently while maintaining **independence and scalability**.

- Best approach is to have a single CI build per microservice, to allow us to quickly make and validate a change prior to deployment into production



- A variation of previous approach is to have one single source tree with all of the code in it, with multiple CI builds mapping to parts of this source tree



Continuous Delivery in Microservices

- Continuous Delivery (CD) is the practice of **automating the deployment process** so that software can be released at any time with minimal manual intervention.
- In a **microservices** architecture, each service is independently built, tested, and deployed, enabling faster and more flexible software delivery.

Key Considerations for CI in Microservices

- Independent Pipelines
- API Contracts & Testing
- Parallel Testing
- Dependency Management

Best Practices for Continuous Delivery in Microservices

- Independent Deployments
- Versioning & Rollbacks
- Deployment Automation
- Monitoring & Logging

CD Pipeline for Microservices

A typical **Continuous Delivery pipeline** for microservices consists of:

- Continuous Integration (CI)
- Artifact Storage
- Deployment to Staging
- Automated Deployment Strategies
- Monitoring & Feedback Loop

Deployment Strategies in Microservices

- Rolling Update: Gradually replaces old instances with new ones.
- Blue-Green: Deploys to a parallel environment, then switches traffic.
- Canary Release: Releases to a small % of users before full rollout.
- Feature Flags: Enables/disables features dynamically without redeploying.

Continuous Deployment in Microservices

- Continuous Deployment takes automation a step further by automatically deploying every code change that passes automated tests directly to the production environment without manual intervention.

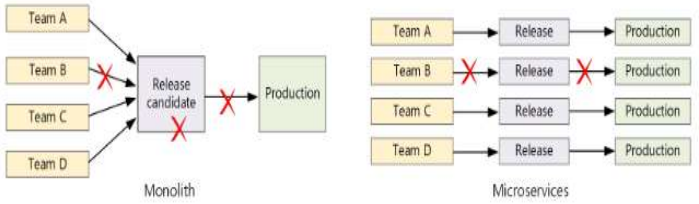
CI/CD pipeline for Monolith Vs Microservices

Monolithic

- Single build pipeline
- Bug fixes impacts release

Microservice

- One pipeline per service
- Issue in one service doesn't impact the other service



Pipeline

- Continuous delivery is the approach whereby we get constant feedback on the production readiness of each and every check-in, and furthermore treat each and every check-in as a release candidate

Figure 6-4. A standard release process modeled as a build pipeline

- In a microservices world, where we want to ensure we can release our services independently of each other, it follows that as with CI, we'll want one pipeline per service.

Service Configuration

- If we have some configuration for our service that does change from one environment to another, how should we handle this as part of our deployment process?
- One option is to build one artifact per environment, with configuration inside the artifact itself.
- A better approach is to create one single artifact, and manage configuration separately.

Platform-Specific Artifacts

- From the point of view of a microservice, depending on your technology stack, the artifact may not be enough by itself.
- While a Java JAR file can be made to be executable and run an embedded HTTP process, for things like Ruby and Python applications, you'll expect to use a process manager running inside Apache or Nginx.
- So we may need some way of installing and configuring other software that we need in order to deploy and launch our artifacts.
- This is where **automated configuration management** tools like Puppet and Chef can help

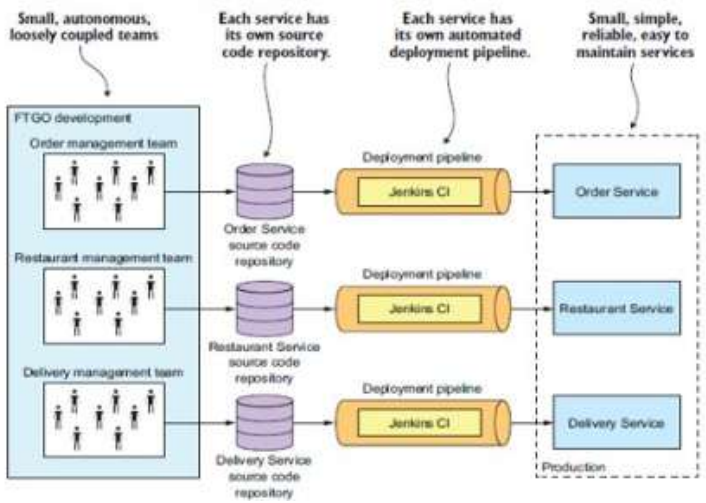
Mono-repo vs. Multi-repo

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

Repository Strategies for Microservices

- Managing code repositories in a **microservices architecture** is crucial for efficient development, version control, and deployment.
- Types of Repository Structures for Microservices:
  - Monorepo (Single Repository for All Microservices): A **single Git repository** containing code for all microservices.
  - Multirepo (One Repository per Microservice): Each microservice has its **own separate Git repository**.

Pipeline for FTGO application



Deployment Challenges

- Many small independent code bases.
- Multiple languages and frameworks
- Release management
- Service updates
- Integration and load testing

References

- Books as per the handout
- https://aws.amazon.com/devops/continuous-integration/
- https://docs.microsoft.com/en-us/azure/architecture/microservices/ci-cd
- https://medium.com/@dmosyan/ci-cd-for-microservices-1b5582f3e1fd

SE ZG583, Scalable Services
Lecture No. 10

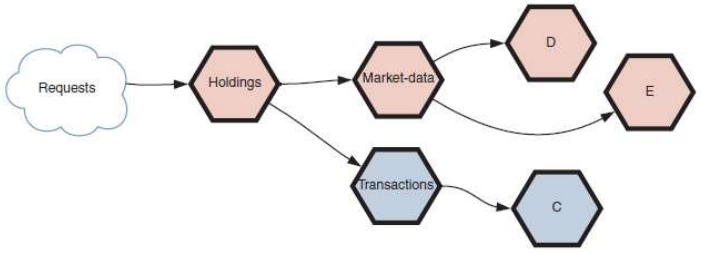
Agenda

Designing reliable Microservices

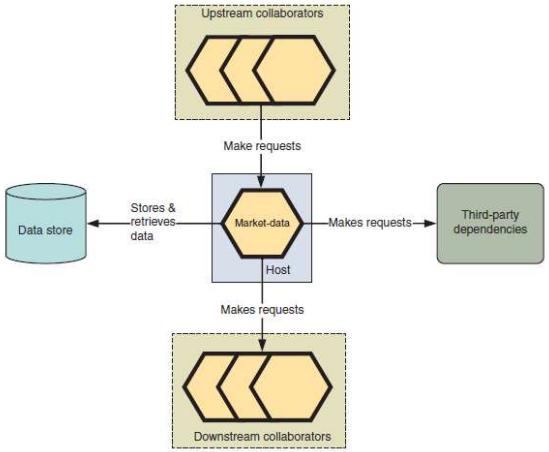
- Sources of failure, cascading failures
- Designing reliable communication: Retires, async. Comm., circuit breakers
- Maximizing service reliability: Load balancing, Rate limiting (Queues, Throttling)
- Service mesh

What is reliability?

- A service's availability is a measure of how reliable you can expect it to be.



Example



Hardware Failure

Source of failure	Frequency	Description
Host	Often	Individual hosts (physical or virtual) may fail.
Data center	Rare	Data centers or components within them may fail.
Host configuration	Occasionally	Hosts may be misconfigured— for example, through errors in provisioning tools.
Physical network	Rare	Physical networking (within or between data centers) may fail.
Operating system and resource isolation	Occasionally	The OS or the isolation system—for example, Docker—may fail to operate correctly.

Sources of failure

Every point of interaction between your service and another component indicates a possible area of failure. Failures could occur in four major areas:

- *Hardware*
- *Communication*
- *Dependencies*
- *Internal*

Dependency-related failure

Source of failure	Description
Timeouts	Requests to services may time out, resulting in erroneous behavior.
Decommissioned or nonbackwards-compatible functionality	Design doesn't take service dependencies into account, unexpectedly changing or removing functionality.
Internal component failures	Problems with databases or caches prevent services from working correctly.
External dependencies	Services may have dependencies outside of the application that don't work correctly or as expected—for example, third-party APIs.

Communication Failure

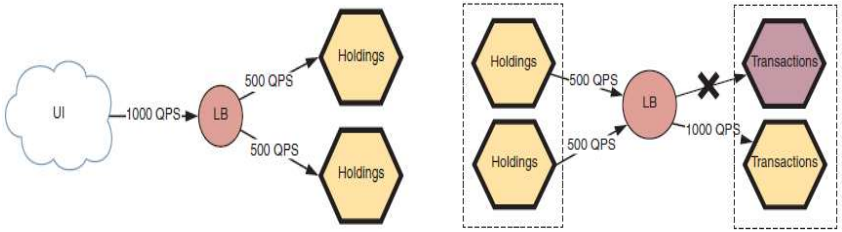
Source of failure	Description
Network	Network connectivity may not be possible.
Firewall	Configuration management can set security rules inappropriately.
DNS errors	Hostnames may not be correctly propagated or resolved across an application.
Messaging	Messaging systems—for example, RPC—can fail.
Inadequate health checks	Health checks may not adequately represent instance state, causing requests to be routed to broken instances.

Cascading failures

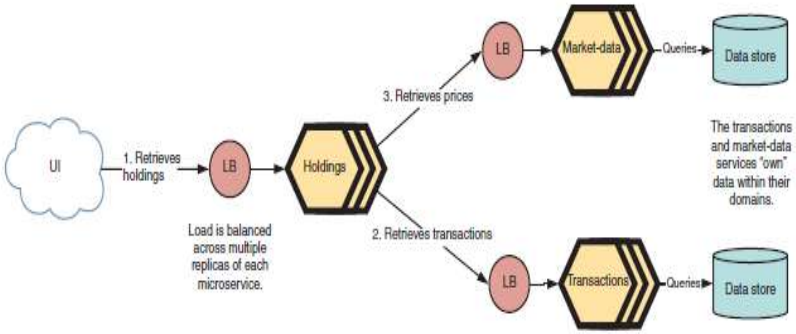
- In a microservice application, overload can cause a domino effect: failure in one service increases failure in upstream services, and in turn their upstream services. At worst, the result is widespread unavailability

Service Practices

- Inadequate or limited engineering practices when developing and deploying services may lead to failure in production
- Because each service contributes to the effectiveness of the whole system, one poor quality service can have a detrimental effect on the availability of swathes of functionality.



Example: cascading failure

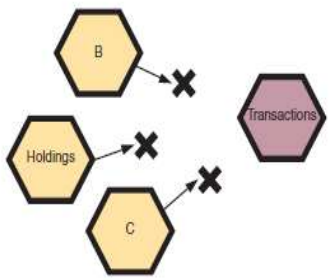


Designing reliable communication

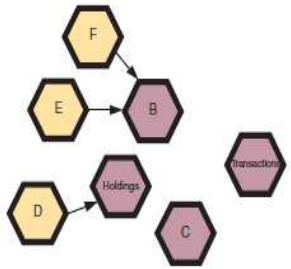
Several techniques for ensuring that services behave appropriately

- Retries
- Fallbacks, caching, and graceful degradation
- Timeouts and deadlines
- Circuit breakers
- Communication brokers

Upstream dependencies are unable to service requests that rely on transactions.



Increased failure in upstream dependencies leads to retries, repeating the cycle of failure.

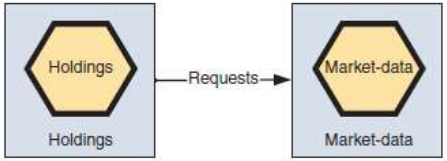


Retries

- How can you use retries to improve your resiliency in the face of intermittent failures without contributing to wider system failure if persistent failures occur?
- First, you could use a variable time between successive retries to try to spread them out evenly and reduce the frequency of retry-based load.

Retries

- Imagine that a call from the holdings service to retrieve prices fails, returning an error.
- From the perspective of the calling service, it's not clear yet whether this failure is isolated
- Repeating that call is likely to succeed, or systemic i.e. the next call has a high likelihood of failing.



Asynchronous communication

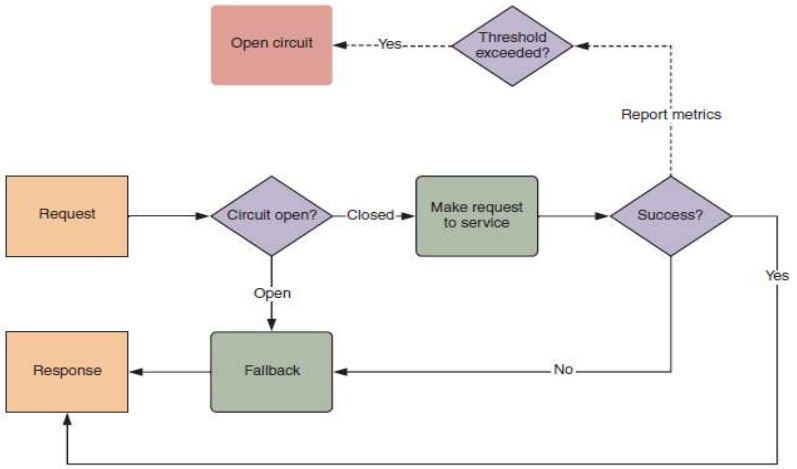
- It's recommended to use asynchronous communication when microservices need to interact **without blocking execution**.

Use Cases:

- **Event-Driven Architectures:** When a service needs to notify other services about an event but does not require an immediate response.
- When you want to **prevent a failure in one service from affecting others**.
- When multiple microservices handle a high volume of requests and blocking calls can degrade performance.
- When an operation takes too long and holding an HTTP connection is impractical.
- When multiple services must consume an event without dependencies.

Retries: few points to remember

- Always limit the total number of retries.
- Use exponential back-off with jitter to smoothly distribute retry requests and avoid compounding load.
- Consider which error conditions should trigger a retry and, therefore, which retries are unlikely to, or will never, succeed.



Circuit breakers

- A circuit breaker is a pattern for pausing requests made to a failing service to prevent cascading failures.

Approach

- When making a request to a service, you can track the number of times that request succeeds or fails
- In case of normal operation, we consider the circuit to be closed.
- If the number of failures seen or the rate of failures within a certain time window passes a threshold, then the circuit is opened

Maximizing service reliability

Why Use Circuit Breakers?

- **Prevent System Overload:** Stops excessive requests to a failing service, reducing unnecessary resource consumption.
- **Improve System Stability:** Ensures that failures in one microservice don't bring down the entire system.
- **Enable Fast Failure & Recovery:** Redirects traffic or returns fallback responses instead of waiting for a failed service.

Load Balancer

The load balancer plays two important roles:

- **Health Check:** Identifying which underlying instances are healthy and able to serve requests
- **Routing requests** to different underlying instances of the service

We can classify health checks based on two criteria:

- **Liveness**: check that the application has started and is running correctly.
- **Readiness**: indicates whether a service is ready to serve traffic

Rate limits

- Technique used to control the number of requests a client can make to a service within a specific time period.

Strategy	Description
Drop requests above volume	Drop consumer requests above a specified volume
Prioritize critical traffic	Drop requests to low-priority endpoints to prioritize resources for critical traffic
Drop uncommon clients	Prioritize frequent consumers of the service over infrequent users
Limit concurrent requests	Limit the overall number of requests an upstream service can make over a time period

Example Use Cases

✓ **Protecting Authentication Services**

Limit failed login attempts (e.g., **5 attempts per minute** to prevent brute force attacks).

✓ **Ensuring Fair API Usage**

Free-tier users can make **100 API calls per hour**, while premium users get **1000 API calls per hour**.

✓ **Preventing Traffic Spikes**

A content delivery API limits requests to **200 per second** to avoid overwhelming backend servers.

Rate limiting in Microservices

- API Gateway-based Rate Limiting
- Middleware-based Rate Limiting
- Cloud Provider Solutions

Throttling Pattern

- **Throttling** is a technique used to **control the rate at which requests are processed** to prevent system overload and ensure fair resource distribution.

Types of Throttling

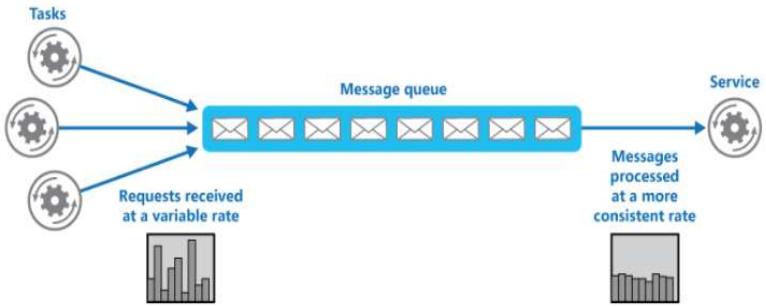
- User-Based Throttling
- Server-Based Throttling
- Network-Based Throttling
- IP-Based Throttling

How to Implement Throttling?

- **API Gateway-Based Throttling**: Use **AWS API Gateway**, Kong, NGINX, or Azure API Management to throttle requests per client or user.
- **Application-Level Throttling**: Implement in code using Redis + Express Rate Limit (Node.js) or Spring Boot Bucket4j (Java).
- **Message Queue-Based Throttling**: Use **Kafka, RabbitMQ, or SQS** to control the rate at which requests are processed asynchronously.
- **Load Balancer-Based Throttling**: Use **AWS ALB, Nginx,** or HAProxy to distribute and throttle traffic before reaching backend services.

How the Pattern Works?

Solution:



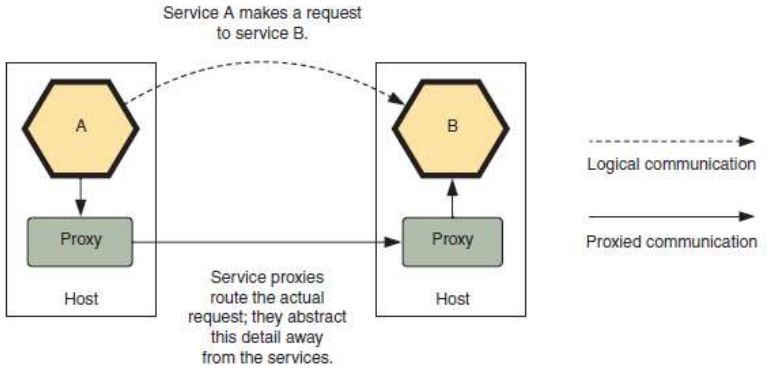
Queue-Based Load Levelling pattern

- The **Queue-Based Load Leveling Pattern** is a **scalability and resilience pattern** used in microservices to handle **high traffic spikes** by placing requests in a queue.
- This prevents services from being overwhelmed and allows them to process tasks **asynchronously at their own pace**.

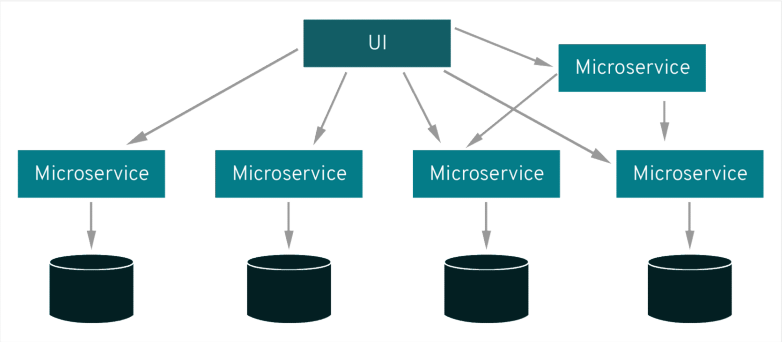
Service Mesh

- A service mesh is a dedicated infrastructure layer designed to handle service-to-service communication within a network of microservices.
- It manages and controls how different parts of an application or service interact with each other.

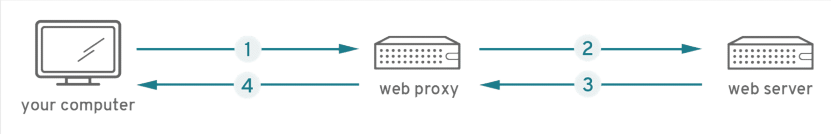
Service mesh



Service Mesh and Microservices



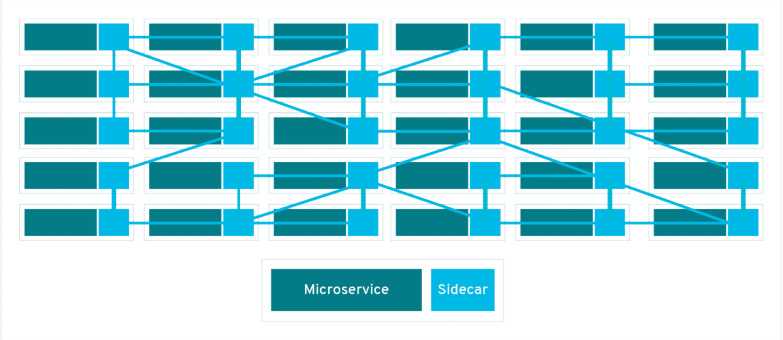
How does service mesh work?



- A service mesh is built into an app as an array of network proxies.
- As your request for this page went out, it was first received by your company's web proxy
- After passing the proxy's security measure, it was sent to the server that hosts this page
- Next, this page was returned to the proxy and again checked against its security measures
- And then it was finally sent from the proxy to you.

- In a service mesh, requests are routed between microservices through proxies in their own infrastructure layer.
- For this reason, individual proxies that make up a service mesh are sometimes called "sidecars," since they run alongside each service, rather than within them.
- Taken together, these "sidecar" proxies—decoupled from each service—form a mesh network.
- Without a service mesh, each microservice needs to be coded with logic to govern service-to-service communication, which means developers are less focused on business goals.

Service Mesh and Microservices



Self study

- https://cloud.google.com/architecture/rate-limiting-strategies-techniques
- https://aws.amazon.com/blogs/architecture/chaos-testing-with-aws-fault-injection-simulator-and-aws-codepipeline/
- https://learn.microsoft.com/en-us/azure/architecture/patterns/sidecar

Does service mesh optimize communication?

- Within a complex microservices architecture, it can become nearly impossible to locate where problems have occurred without a service mesh.
- Over time, data made visible by the service mesh can be applied to the rules for interservice communication, resulting in more efficient and reliable service requests.

References

- Book: Chapter6, Microservices in Action
- https://docs.microsoft.com/en-us/dotnet/architecture/microservices/implement-resilient-applications/partial-failure-strategies
- https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling
- https://docs.microsoft.com/en-us/azure/architecture/patterns/throttling
- https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh

SE ZG583, Scalable Services
Lecture No. 11

Agenda

Securing and Testing scalable services

- Securing code and repositories
- Using Authentication and Authorization
- Testing

Securing code and repository

- Your code is only as secure as the systems used to create it.
- Version control, peer review and built-in auditing are some of the advantages which come with using a code repository.
- If proper attention is paid to security measures, the benefits of using a repository far outweigh the risks.

Measures for securing the code and repository

- Choose a repository you trust
- Consider the exposure of your repository
- Protect access credentials
- Access to the repository should be revoked when no longer required, or in the event of compromise
- Encryption
- Code Signing
- Ensure your code is backed up
- Auditing and Logging

Authentication implementation options

- One option is for the individual services to authenticate the user.
- Other approach is for the API gateway to authenticate a request before forwarding it to the services.

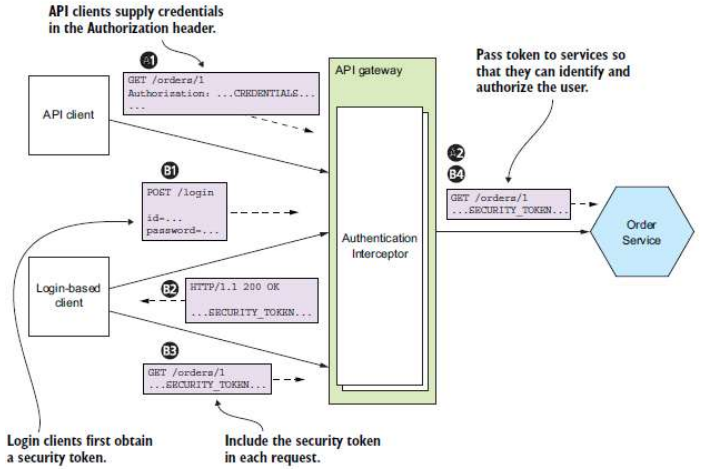
What is Authentication?

- Verifying the identity of the application or human that's attempting to access the application.

The sequence of events for API clients is as follows:

- A client makes a request containing credentials.
- The API gateway authenticates the credentials
- It creates a security token, and passes that to the service or services.

Authentication in API Gateway



Authorization

- Verifying that the principal is allowed to perform the requested operation on the specified data.
- Applications often use a combination of rolebased security and access control lists (ACLs).

The sequence of events for login-based clients is as follows:

- A client makes a login request containing credentials.
- The API gateway returns a security token.
- The client includes the security token in requests that invoke operations.
- The API gateway validates the security token and forwards it to the service or services.

Authorization in API Gateway

- If a user isn't allowed to access a particular path, the API gateway can reject the request before forwarding it on to the service.
- We can implement authorization in the API gateway using a security framework, such as Spring Security.

Key aspects and benefits of implementing authorization in an API gateway

- Centralized Access Control
- Scalability and Performance
- Flexibility
- Logging and Auditing
- Protection Against Attacks

Explore: What are the possible drawbacks of this approach?

Key aspects and benefits of implementing authorization within services

- Fine-Grained Control
- Decentralization
- Service Autonomy
- Scalability
- Flexibility
- Security Isolation

- **Explore**: What are the possible drawbacks of this approach?

Authorization in Services

- The other place to implement authorization is in the services.

- A service can implement role-based authorization for URLs and for service methods.

- It can also implement ACLs to manage access to aggregates.

What is OAuth?

- OAuth is an open-standard authorization protocol or framework

- The simplest example of OAuth is when you go to log onto a website and it offers one or more opportunities to log on using another website's/service's logon.

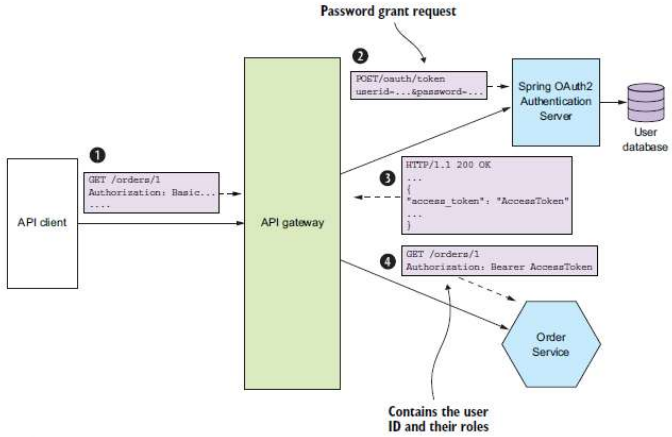
- You then click on the button linked to the other website, the other website authenticates you

Which approach is better?

The preferred approach depends on the specific requirements and characteristics of the system:

- **Centralized Control:** If uniformity and centralized management of access control policies are paramount, implementing authorization in an API gateway might be preferred.
- **Fine-Grained Control:** For systems with diverse access control requirements or where services have unique authorization needs, implementing authorization within services may be more suitable.
- **Scalability and Performance:** If scalability and performance are critical concerns, especially in high-traffic environments, offloading authorization to an API gateway might be preferable.
- **Security Isolation:** If security isolation is a priority and potential single points of failure are a concern, embedding authorization within services could be favored.
- In many cases, a hybrid approach may be appropriate, combining elements of both centralized and decentralized authorization mechanisms to achieve a balance between control, scalability, performance, and security.

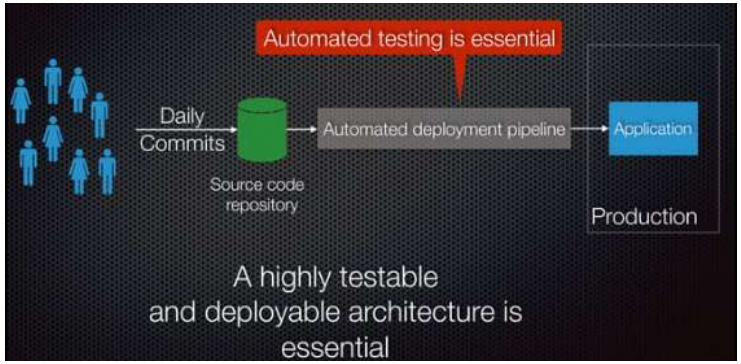
How OAuth works for Microservices



Terms used in OAuth

- Authorization Server
- Access Token
- Refresh Token
- Resource Server
- Client

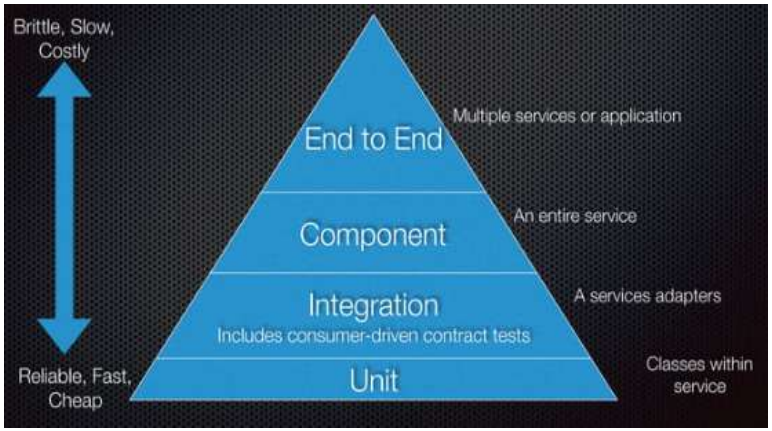
Use DevOps



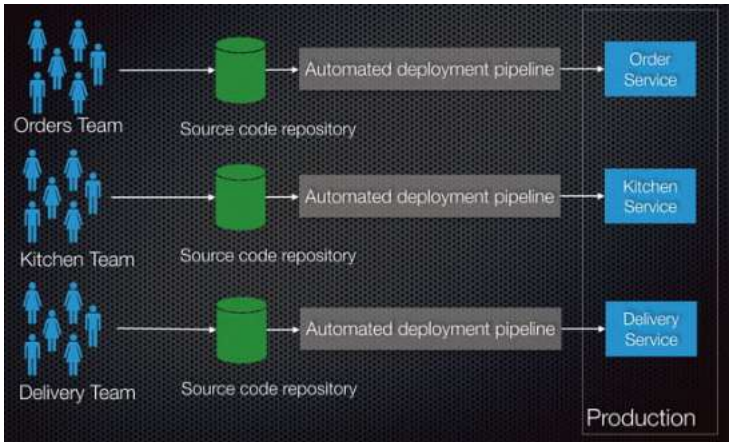
https://microservices.io/testing/

Testing in Microservices

Testing Pyramid and Microservices



Autonomous service teams



Unit Tests

- The size of the unit under test is not defined anywhere some firms, consider writing them at a class level or at a group of related classes.
- Consider keeping the testing units as small as possible. It becomes easier to express the behavior if the test is small since the branching complexity of the unit is lower.

Testing strategies for Microservice architectures

- Unit Testing
- Component Testing
- Integration Testing
- Contract Testing
- End-to-End testing
- Fault Injection Testing
- **Monitoring and Observability**

Component Testing

- A component or microservice is a well-defined coherent and independently replaceable part of a larger system. Once we execute unit tests of all the functions within microservices, it's time to test microservice itself in isolation.
- To test a single microservice in isolation, we need to mock the other microservices. Isolating the microservices in this way using test doubles avoids any complex behavior they exert on execution.

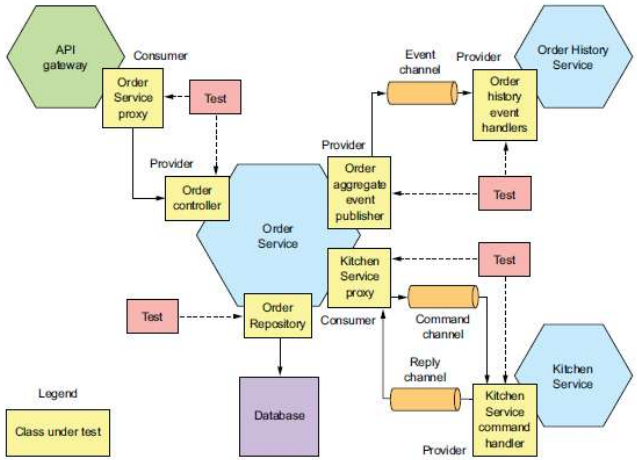
Types of Unit Test

- *Solitary unit* — looks at the interactions and collaborations between an object and its dependencies, which are replaced by the test doubles.
- *Sociable unit test*— focuses on testing the behavior of the module by observing changes in their state.

Approaches for integration testing

- Service Integration Testing
- Testing Service Contracts
- Mocking External Dependencies
- End-to-End Testing

Integration Test



Contract testing

- For simplifying integration tests that verify interactions between application services is to use contracts

Interaction style	Consumer	Provider	Contract
REST-based, request/response	`API Gateway`	`Order Service`	HTTP request and response
Publish/subscribe	`Order History Service`	`Order Service`	Domain event
Asynchronous request/response	`Order Service`	`Kitchen Service`	Command message and reply message

Consumer Driven Contact testing

Verify that a service and its clients can communicate while testing them in isolation

End-to-End Testing

- We usually treat the system as a black box while performing end-to-end tests.
- As Microservice includes more moving parts for the same behavior, it provides coverage of the gaps between the services.

- *Consumer-side contract tests*: These are tests for the consumer's adapter. They use the contracts to configure stubs that simulate the provider, enabling you to write integration tests for a consumer that don't require a running provider.
- *Provider-side contract tests*: These are tests for the provider's adapter. They use the contracts to test the adapters using mocks for the adapters's dependencies.

Fault injection testing

- Fault injection testing, also known as chaos testing or failure testing, is a technique used to validate the resilience and fault tolerance of systems by deliberately introducing faults or failures into the system and observing how it responds.

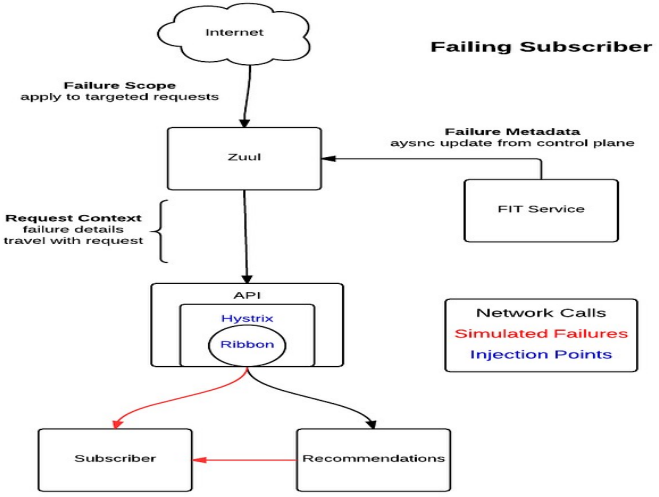
There are various best practices to adhere to while writing efficient E2E tests for microservices:

- Define end-to-end test scenarios that cover critical user journeys or business processes across multiple Microservices.
- To write automated tests that are simple to execute and manage, testing team should make use of testing frameworks and tools.
- Developers should use a continuous integration and delivery (CI/CD) strategy, in which the tests are incorporated into the development cycle and executed automatically anytime the application is changed or released.

Tools for Unit testing

- JUnit (for Java), Pytest (for Python), JUnit (for .NET): These are popular unit testing frameworks for writing and executing unit tests for individual components, functions, or classes within microservices.
- Mockito, Mockito.NET: These mocking frameworks help simulate dependencies and external systems during unit testing by creating mock objects or stubs.

Chaos Engineering At Netflix



Tools for End-to-End Testing

- **Selenium**: Selenium is a widely used tool for automating web browser interactions and testing web applications. It allows you to create automated end-to-end tests that simulate user interactions across multiple microservices.
- **Cypress**: Cypress is a modern testing tool for web applications that provides an elegant API for writing end-to-end tests. It offers features such as real-time testing, automatic waiting, and debugging capabilities.

Tools for Integration Testing

- **Postman**: Postman is a popular API testing tool that allows you to create and execute tests for RESTful APIs. It supports automated testing, scripting, and validation of API responses.
- **SoapUI**: SoapUI is another API testing tool that specializes in testing SOAP and RESTful web services. It provides features for functional testing, security testing, and performance testing of APIs.
- **WireMock**: WireMock is a lightweight HTTP server for stubbing and mocking HTTP responses. It is commonly used for simulating dependencies and external services during integration testing.

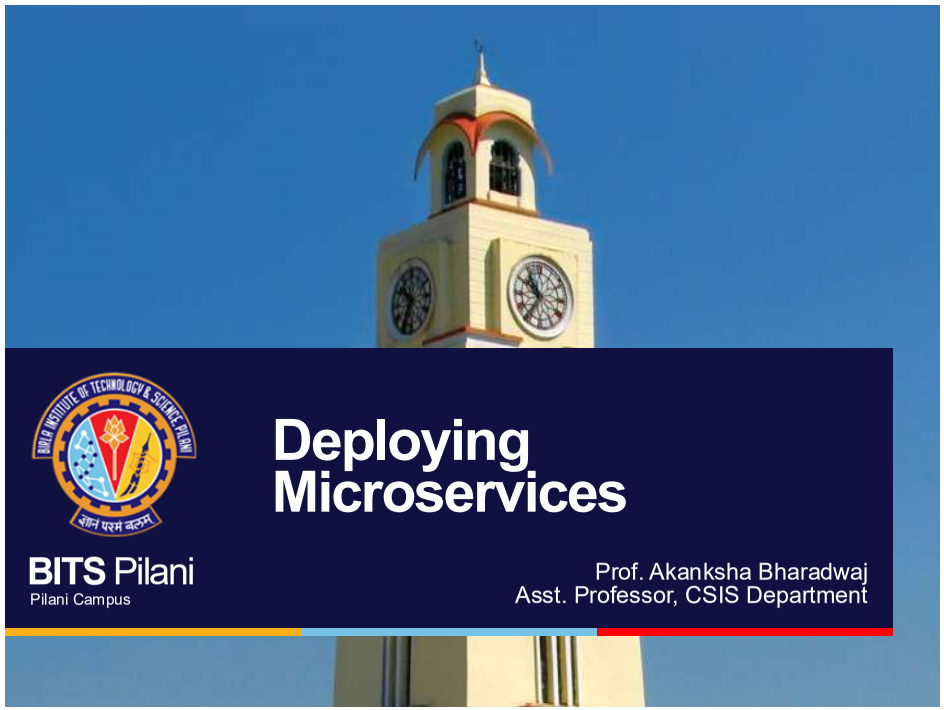
References

- Chapter 9,10,11, Microservives Patterns by Chris Richardson
- Chapter 6.4.3, Microservices in Acyion
- https://www.ncsc.gov.uk/collection/developers-collection/principles/protect-your-code-repository
- https://performancelabus.com/load-testing-for-microservices/
- https://netflixtechblog.com/from-chaos-to-control-testing-the-resiliency-of-netflixs-content-discovery-platform-ce5566aef0a4
- https://spectrum.ieee.org/chaos-engineering-saved-your-netflix
- https://microservices.io/testing/
- https://netflixtechblog.com/fit-failure-injection-testing-35d8e2a9bb2

Self Study

- https://docs.github.com/en/code-security/getting-started/securing-your-repository
- OAuth in detail from textbook
- Testing in detail from textbook
- https://www.neotys.com/insights/microservices-load-testing
- https://netflix.github.io/chaosmonkey/

SE ZG583, Scalable Services
Lecture No. 12



Deploying Microservices

Prof. Akanksha Bharadwaj
Asst. Professor, CSIS Department

Agenda

Deploying Microservices

Service startup

Running multiple instances

Adding load balancer

Service to host models

- Single Service Instance to Host
- Multiple static Service Per Host
- Multiple scheduled services per host

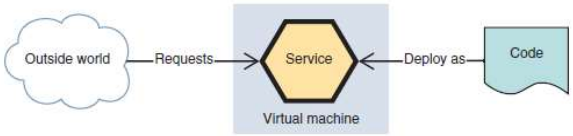
Challenges in deploying Microservices

Installing microservices can be complex due to their distributed nature. Here are the key challenges:

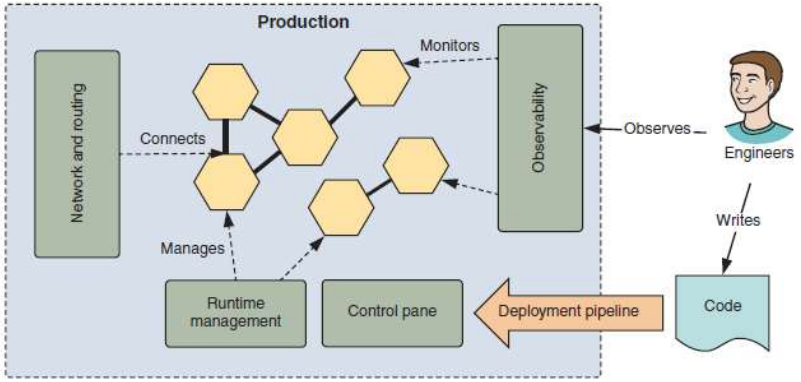
- **Dependency Management**: Microservices often rely on multiple libraries, frameworks, and external services.
- **Configuration Complexity**: Each microservice may require its own configuration for databases, APIs, or environments (dev, staging, production).
- **Networking and Communication**: Microservices communicate over networks, introducing issues like latency, timeouts, and service discovery.
- **Containerization and Orchestration**: Most microservices use containers (e.g., Docker) and orchestration tools (e.g., Kubernetes).
- **Monitoring and Logging**: With many services running independently, tracking performance, errors, and logs becomes challenging.

Service Startup

- You need to take your code, get it running on a virtual machine, and make it accessible from the outside world

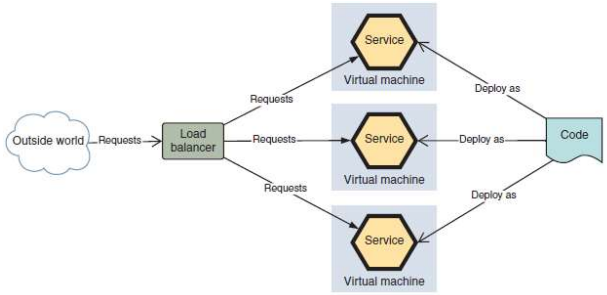


Features of a microservice production environment



Run multiple instances of a service

- Think about Scale horizontally
- Deploy the redundancy

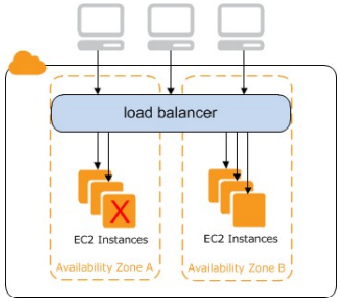


Service Startup

- Installs binary dependencies required to run the application
- Downloads your service code from Github or any other repository
- Installs that code's dependencies
- Configure a supervisor if required

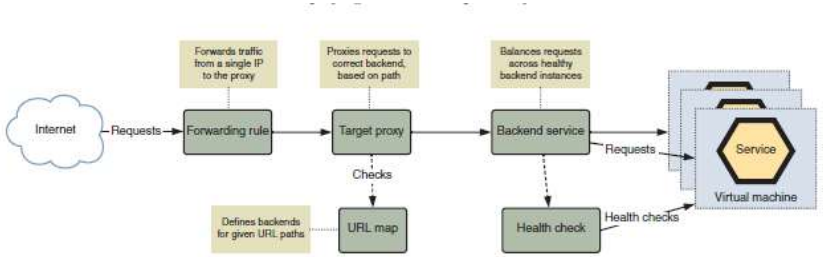
AWS: What is a Classic Load Balancer?

- Elastic Load Balancing automatically distributes your incoming traffic across multiple targets, such as EC2 instances, containers, and IP addresses, in one or more Availability Zones.



Adding a load balancer

- The load balancer uses routing rules, proxies, and maps to forward requests from the outside world to a set of healthy service instances.



Multiple Service Instances per Host Pattern

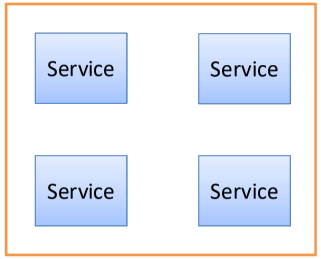
Deploying multiple services to a single host in production is synonymous with deploying multiple services to a local dev workstation or laptop.

Benefits:

- Attractive from a host management point of view
- Cost

Challenges:

- Monitoring
- Isolation of instances
- Resource consumption



Models for Deploying Services

Service Instance per Container Pattern

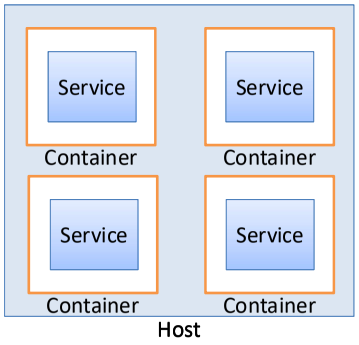
Package the service as a container image and deploy each service instance as a container

Benefits

- Easy to scale up and down a service.
- Isolation
- Limit the resource utilization by a service instance
- Faster deployment

Drawbacks

- Infrastructure for deploying containers is not rich



Single Service Instance per Host Pattern

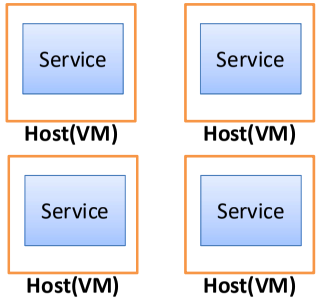
We can avoid the side effects of multiple services living on a single host, by making monitoring and remediation much simpler

Benefits

- Service instance Isolation
- Easy to manage, monitor and deploy
- No conflicting resource requirements

Drawbacks

- Less efficient resource utilization
- cost



Running a single service on local machine

- Open the service that you have created
- Use an appropriate command in the command prompt to run the service
- Use local browser/postman application to check the functioning

Best Deployment Strategy

3 main practical and popular approaches to deploy Microservices:

- Deploy each Microservice instance on a separate Virtual Machine
- Deploy each Microservice instance as a (Docker) container on Kubernetes
- Deploy each Microservice as a Serverless Function

Dockerfile best practices

- Pick the right base image
- Keep the number of steps in the Dockerfile to minimum
- Control the structure of your dockerfile

Dockerfile

- Docker can build images automatically by reading the instructions from a Dockerfile

Basic Docker commands:

- FROM
- RUN
- ENV
- COPY
- EXPOSE
- ENTRYPOINT
- CMD
- VOLUME
- WORKDIR
- LABEL
- ADD
- ARG

Run multiple services on a container

If you need to run more than one service within a container, you can accomplish this in a few different ways.

- Use wrapper script
- Use a process manager like supervisord

Docker Compose

- Compose is a tool for defining and running multi-container Docker applications.

Basic Structure of Docker compose file

- version
- services
- build
- command
- ports
- volumes
- links
- image
- environment
- restart
- depends_on

Self Study

- https://docs.aws.amazon.com/elasticloadbalancing/latest/classic/elb-getting-started.html
- https://docs.docker.com/

Run a single service using Docker desktop

- Create a dockerfile
- Build a Docker image
- Run in Docker
- Check your running application
- Observe the running container
- Stop the container

References

- Chapter8, Microservices in Action
- https://docs.aws.amazon.com/elasticloadbalancing

SE ZG583, Scalable Services
Lecture No. 13

Agenda

Deploying microservices

- Deploying services without downtime: Canaries, Blue-Green, & rolling deploys
- Deploying microservices using Serverless deployment

Deployment with Containers

- Introduction to containers
- Containerizing a service
- Deploying to a cluster

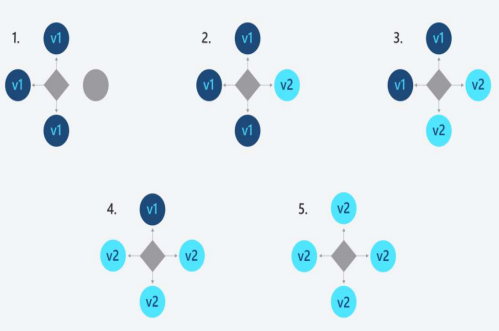
Introduction of deployment Strategies

Here are four common deployment strategies that organizations use in production

- Ramped
- Blue/Green
- Canary
- A/B testing

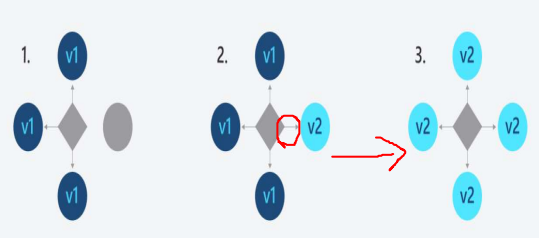
Ramped

- This is the default roll-out method in Kubernetes.
- This deployment slowly replaces pods one at a time to avoid downtime.



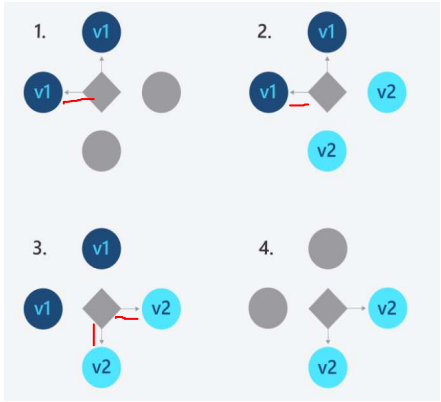
Canary

- Allow the customers to test your Kubernetes deployment by releasing the new version to a small group of them.



Blue/Green

- In this you release a new version of your application or workflow while your current version is still running.

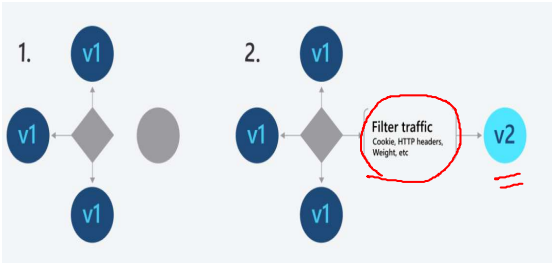


Serverless Deployment

- It is a deployment infrastructure that hides any concept of servers physical or virtual hosts, or containers.
- To deploy your service using this approach, you package the code (e.g. as a ZIP file), upload it to the deployment infrastructure and describe the desired performance characteristics.

A/B testing

- Much like the canary Kubernetes deployment strategy, an A/B testing strategy targets a specific group of customers.



Examples of Serverless Deployment

- AWS Lambda
- Google Cloud Functions
- Azure Functions

Drawbacks of using Serverless Deployment

- It's not the best scenario for executing long-running applications.
- Vendor lock in
- Cold start

Container Image

- A container image is a lightweight, standalone, executable package that contains everything needed to run a piece of software, including the application code, runtime, libraries, dependencies, and configuration files.

Introduction to containers

- A container is a lightweight, portable, and self-contained unit of software that packages together an application and its dependencies, libraries, and configuration files, ensuring consistency and reliability across different computing environments.
- Containers provide a standardized runtime environment for applications, enabling them to run reliably and consistently on any platform that supports containerization.

Key aspects of Docker containers

- Image-Based
- Isolation
- Portability
- Efficiency
- Orchestration
- Microservices
- DevOps

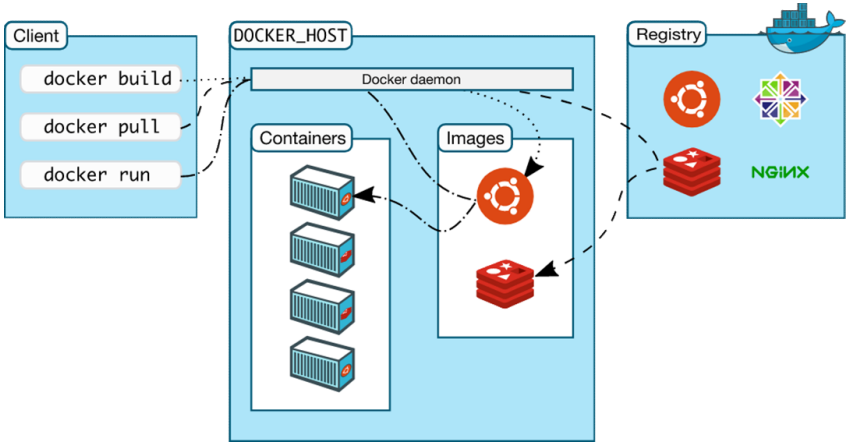
Docker Containers

- Docker containers are lightweight, portable, and self-contained units of software that encapsulate an application along with its dependencies, libraries, and configuration files.
- Docker is a containerization platform that allows developers to create, deploy, and manage containers easily, providing a consistent runtime environment across different computing environments.
- Docker provides the ability to package and run an application in a loosely isolated environment called a container.

What can I use Docker for?

- Fast and consistent delivery of your applications
- Microservices Architecture
- Scalability

Docker architecture



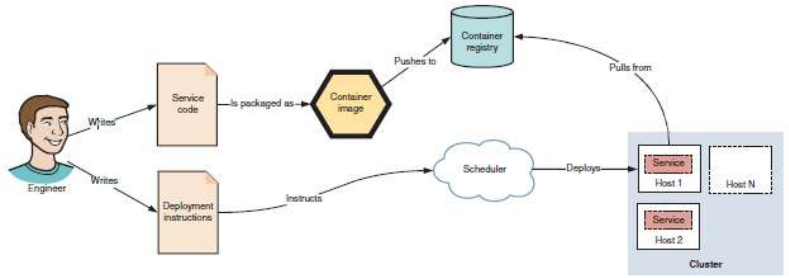
Build and Run Docker Image

Steps required:

1. Install Docker
2. Build your Docker image
3. Run your Docker image

Containerizing a service

- Build an image for a service
- Run multiple instances or containers of your image
- Push your image to a shared repository, or registry



Building your image

- To build image, first you need to create a Dockerfile
- Example: **docker build -t market-data:first-build .**

Working with images

- The image will include the file system that your application needs to run code and dependencies and other metadata, such as the command that starts your application.
- Example: **docker pull python:3.6**

Storing an image

- To push to this repository, you need to tag your image with an appropriate name.
- Docker image names follow the format <registry>/<repository>:<tag>.

Running containers

- Example:
- **docker run -d -p 80:80 docker/getting-started**
- **docker run -d -p 8000:8000 --name market-data market-data:latest**

Update the application

Steps

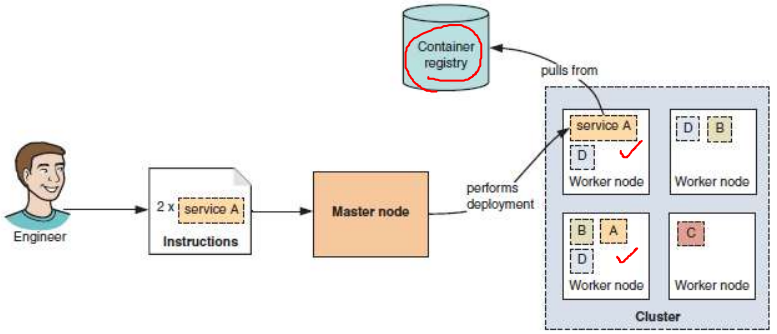
1. Update the source code
2. Build our updated version of the image, using the same command we used before.

   docker build -t getting-started:v2 .

3. After deleting old container, start a new container using the updated code.

   docker run -dp 3000:3000 getting-started:v2

Deploying to a cluster



High-level scheduler architecture and deployment process

Play with Docker

1. Open your browser to Play with Docker: https://labs.play-with-docker.com/
2. https://training.play-with-docker.com/

Share the application

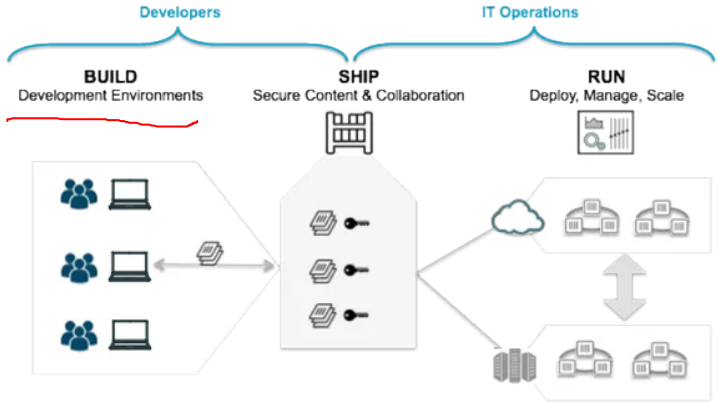
Create a repo

1. Sign in to Docker Hub.
2. Click the Create Repository button.
3. Give the desired repo name and make sure the Visibility is Public.
4. Click the Create button!

Containers as a Service (CaaS)

- Containers as a Service is a model where IT organizations and developers can work together to build, ship and run their applications anywhere.
- CaaS enables an IT secured and managed application environment consisting of content and infrastructure, from which developers are able to build and deploy applications in a self service manner.

CaaS workflow



Docker and git

Ques. How to use GitHub with Docker

Self study:
Link: https://docs.docker.com/ci-cd/github-actions/

Self Study

- https://aws.amazon.com/blogs/apn/deploying-code-faster-with-serverless-framework-and-aws-service-catalog/
- How to use GitHub with Docker: https://docs.docker.com/ci-cd/github-actions/
- Minikube: https://minikube.sigs.k8s.io/docs/start/

References

- Chapter8,9 Microservices in Action
- Link: https://azure.microsoft.com/en-in/overview/kubernetes-deployment-strategy/
- https://www.docker.com/blog/containers-as-a-service-caas/
- https://docs.docker.com/

SE ZG583, Scalable Services
Lecture No. 14

Monitoring

- Monitoring involves the collection, analysis, and visualization of various metrics and data points from a system.
- It focuses on tracking predefined metrics and observing the system's behavior to ensure its health, performance, and availability.
- Monitoring is typically based on pre-established monitoring tools or frameworks that capture specific aspects of the system, such as CPU usage, memory utilization, response times, error rates, and network traffic.

Observability

- The ability to measure a system's current state based on the data it generates, such as logs, metrics, and traces.
- Observability relies on telemetry derived from instrumentation that comes from the endpoints and services in your multi-cloud computing environments.
- Observability encompasses Monitoring

Difference between Observability and Monitoring

In a web application

- Monitoring might involve tracking the number of requests per second, response times, HTTP error rates, database connection pool utilization, and other relevant metrics. These metrics are typically collected at regular intervals and displayed on dashboards or sent as alerts to notify system administrators or developers about potential issues or anomalies in the system's performance.
- Observability could involve collecting and analyzing detailed logs of requests and responses, tracing individual requests across different microservices, and capturing custom events or signals that indicate specific conditions or scenarios within the application. This richer set of data enables engineers to explore and investigate the system's behavior in greater depth, even in complex distributed architectures.

Monitoring Stack

- A monitoring stack comprises various components working together to collect, store, analyze, and visualize data about the health, performance, and behavior of a system.
- Here are the key components typically found in a monitoring stack:
  - Metrics
  - Logs
  - Traces

Monitoring

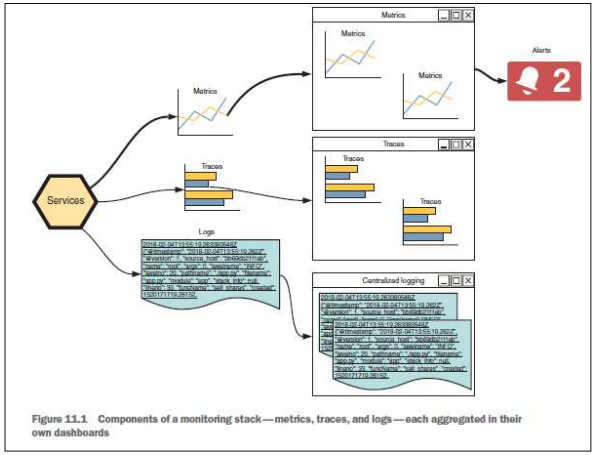


Figure 11.1 Components of a monitoring stack—metrics, traces, and logs—each aggregated in their own dashboards

Monitoring

- Can use Monitoring to: **Diagnose issues** OR **Prevent them from arising**

Diagnosing Issues

- Broken services (network problem, internal service problem)
- Communication (connectivity lost between services, or between service and event queue)
- Alert on the abnormal decrease in throughput of a service

Prevent

- Send automatic notification alerts when the number of messages in a queue go below a certain threshold
- Accumulating messages in the queue - indicates that the consuming services not working
- Predict load increases and alert

Monitoring - Example

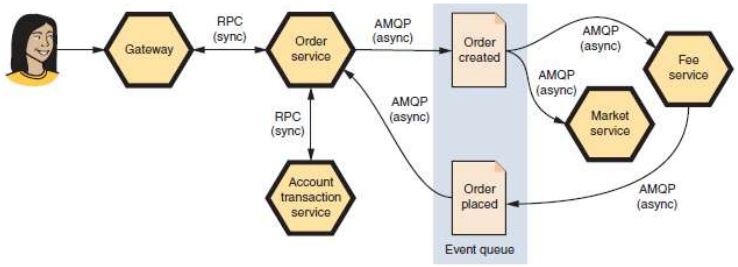


Figure 11.2 Services involved in placing orders and their communication protocols

Golden Signals

3. Traffic (Synonym for Load) - Amount of use of service per time unit

- This signal measures the **demand placed on a system**.
- It can vary depending on the type of system being observed, the number of requests per second, network I/O, and so on.
- *Correlation to Throttling of the system?? Reliability? Steps to overcome?*

4.Saturation - Consumption of System Resources

- How much resources (Memory, I/O) your application takes. Application performance degrades when the system reaches 100% utilisation.
- At a given point, this measures the capacity of the service.
- It mainly applies to resources that tend to be more constrained, like CPU, memory, and network.

You need to monitor these signals and keep them in control for making your service available throughout the time.

Golden Signals (Metrics)

- Focus on 4 golden signals while collecting metrics - **Latency, Errors, Traffic and Saturation**

1. **Latency** (Synonym for Lag) - **Time it takes to serve a request**

- Measures how much time passes between when you make a request to a given service and when the service completes the request. We can infer that the service is degrading if it shows increasing latency.

2. Errors - Rate of requests that fail

This signal determines the number of requests that don't result in a successful outcome.

- Explicit error - HTTP 500 (Unexpected Server Error)
- Implicit error - HTTP 200 but with wrong content

Types of Metrics

2. Gauges

Gauges are metrics representing single numerical values that can arbitrarily go up or down.

Some examples of metrics using gauges are:

- Number of connections to a database
- Current Memory used
- CPU used
- Load average
- Number of services operating abnormally
- Number of currently running processes

Types of Metrics

Visualising the output of Golden Signals

Four ways of Showcasing Metrics - Counters, Gauges, Histograms, Summaries

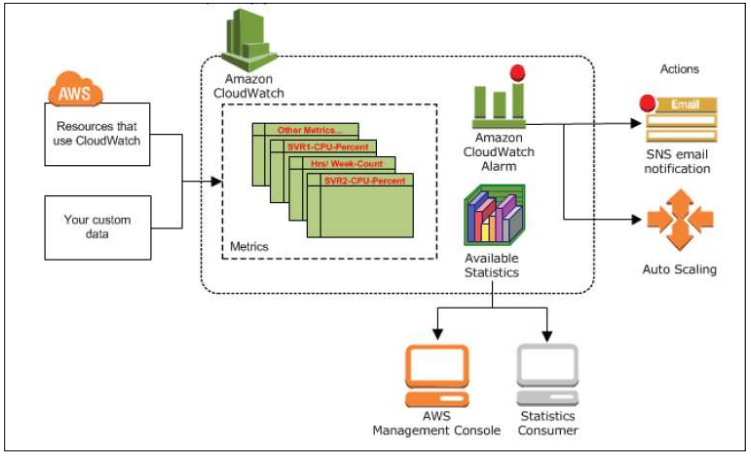
1. Counters

Counters are a cumulative metric representing a single numerical value that'll always Increase or be reset to zero on restart. Examples of metrics using counters are:

- Number of requests
- Number of errors
- Number of each HTTP code received [HTTP 200, 400x, 500x etc]
- Bytes transmitted

You shouldn't use a counter if the metric it represents **can also decrease**. Ex: Number of currently running processes. For that, you should use a gauge instead.

AWS CloudWatch Monitoring Tool



Types of Metrics

3. Histograms

- You use histograms to sample observations and categorize them in configurable buckets
- per type, time, and so on.
- Metrics which samples observations (usually things like request durations or response sizes) and counts them in configurable buckets. It also provides a sum of all observed values.

Examples of metrics represented by histograms are:

- Latency of a request
- I/O latency
- Bytes per response

Key aspects of Alerts

1. **Thresholds**: Alerts are typically triggered when a monitored metric exceeds or falls below a specified threshold. For example, an alert might be set to trigger when CPU utilization exceeds 90% or when the response time of a service exceeds a certain duration.
2. **Conditions**: Alerts can be based on various conditions and combinations of metrics. They can include simple conditions like a single metric crossing a threshold or complex conditions that involve multiple metrics and their relationships. For instance, an alert might be triggered when both CPU utilization and Memory usage exceed certain levels simultaneously.
3. **Severity levels:** Alerts can be categorized into different severity levels to indicate the urgency or impact of the issue. For example, critical alerts indicate severe problems that require immediate attention, while warnings or informational alerts may represent less critical situations.

Alerts

- Alerts in monitoring refer to notifications or warnings triggered by predefined conditions or thresholds to indicate potential issues or anomalies in a system.
- When monitoring systems detect that a specific metric or event has crossed a predefined threshold, an alert is generated and sent to system administrators, operators, or developers.
- Alerts play a crucial role in timely identification and response to problems, allowing for proactive troubleshooting and mitigation.

Alerts

Raising Sensible & Actionable Alerts

- Aim to have as few alerts as possible.
- Alerting on symptoms that are associated with end-user pain rather than trying to catch every possible way that pain could be caused.
- Alerts should link to relevant consoles and make it easy to figure out which component is at fault.
- For online servicing systems, the key metric is on high latency and error rates as high up in the stack as possible.
- For offline processing systems, the key metric is how long data takes to get through the system, so page if that gets high enough to cause user impact.
- For batch jobs it makes sense to page if the batch job has not succeeded recently enough, and this will cause user-visible problems.
- While not a problem causing immediate user impact, being close to capacity often requires human intervention to avoid an outage in the near future.
- It is important to have confidence that monitoring is working. Accordingly, have alerts to ensure that Prometheus servers, Alertmanagers, PushGateways, and other monitoring infrastructure are available and running correctly.

Key aspects of Alerts

4. **Notification channels:** When an alert is triggered, it needs to be communicated to the appropriate individuals or teams responsible for addressing the issue. Notification channels can include Email, SMS, Slack, PagerDuty, or other collaboration and incident management tools. [Or Cloud ones like Amzon SNS, SES etc.]
5. **Escalation policies**: In complex systems or larger organizations, alerts may follow predefined escalation policies. These policies determine who receives the initial alert and specify subsequent steps if the alert is not acknowledged or resolved within a certain timeframe.
6. **Alert suppression and de-duplication**: Monitoring systems often employ mechanisms to prevent excessive alerting or alert floods. This can include suppressing redundant alerts for the same underlying issue or consolidating similar alerts into a single notification.

Logs

Useful Info in Log Entries & Logging the Right Information

- Provide the logs wherever necessary.
- Use Tags to identify Modules, Methods, Requests / Response etc.,
- Provide the Warning / Error Code along with Description aptly based on the operation you perform.
- Make the logs user friendly with simple language.
- Don't put logs everywhere which will be too noisy when you are stuck on an error and debugging it.

Logs and Traces

Using Logs & Traces

- **Logs provide information about what happened inside the service.**
- Tracing tells you what happened between services/components and their relationships.
- This is extremely important for microservices, where many issues are caused due to the failed integration between components.

Logs

Tools for Logging

- **Elk** - Log management platform that works by enabling you to gather massive amounts of log data from anywhere across your infrastructure into a single place, then search, analyse and visualize it in real time.
- **Splunk** - Based on event management and handles the data to give their enterprise with alerts and event logs.
- **Grafana** - Provides charts, graphs, and alerts for the web when connected to supported data sources
- **Fluentd** - Data collector for building the unified logging layer. Once installed on a server, it runs in the background to collect, parse, transform, analyse and store various types of data.

Log - Centralized

Single Application running on Single Machine – Single Application Log

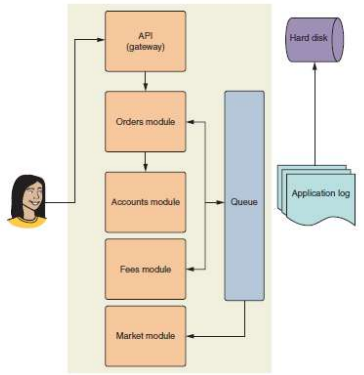


Figure 12.1 The sell order use case implemented in a single application

Situation in Microservice

- Multiple Services
- Each service multiple replicas?
- How to have centralized log?

Log – Distributed – Difficulties in accessing?

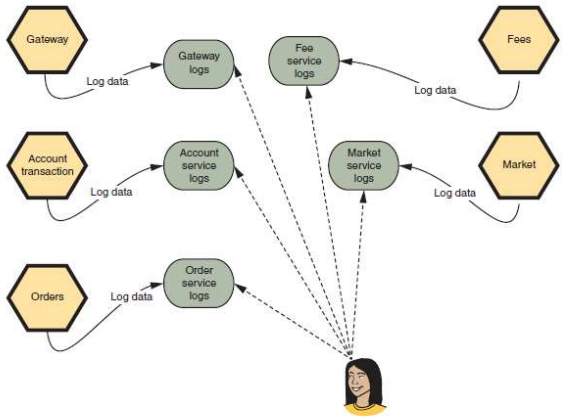


Figure 12.3 Accessing logs for each service in different running instances is challenging.

1. **Persistence** - Make sure you persist log data so it survives through service restarts and scaling
2. **Aggregation** - Aggregate all log data from multiple services and instances of those services in a central location
3. **Querying** - Make the stored data usable, allowing for easy searches and further processing

Log - Distributed

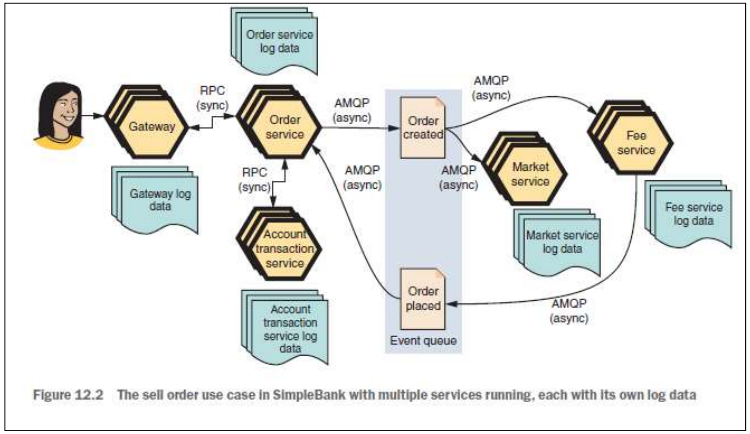
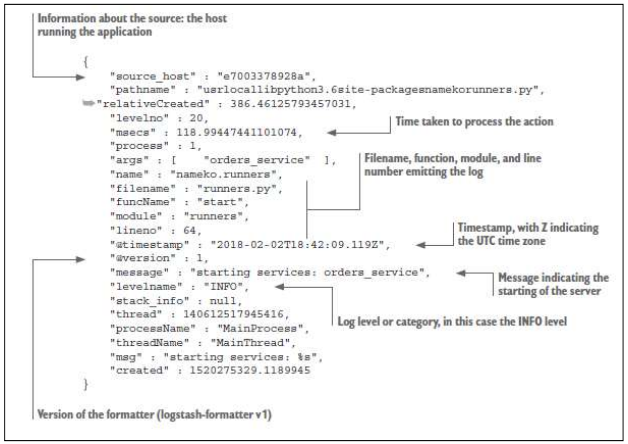


Figure 12.2 The sell order use case in SimpleBank with multiple services running, each with its own log data

Sample Log collected from Logstash library Python

Logstash - A tool to collect, process, and forward events and log messages from multiple sources. It provides multiple plugins to configure data collecting.



Types of Log

- Application logs
- Database logs
- Network logs
- Performance data collected from the underlying operating system

Traces [Distributed Tracing]

X-Ray Tracing

- X-Ray helps developers analyze and debug distributed applications, such as those built using a microservices architecture.
- With X-Ray, you can understand how your application and its underlying services are performing, and identify and troubleshoot the root cause of performance issues and errors.
- **Complete Tracing** - X-Ray provides an end-to-end, cross-service view of requests made to your application. It gives you a view of requests flowing through your application by aggregating the data gathered from individual services in your applications.

Traces [Distributed Tracing]

Tracing Interaction between Services

- Tracing is a fundamental process in software engineering, used by programmers along with other forms of logging, to gather information about an application's behaviour.
- Traditional tracing runs into problems when it is used to troubleshoot applications built on a distributed software architecture.
- Since microservices scale independently, it's common to have multiple iterations of a single service running across different servers, locations, and environments simultaneously, creating a complex web through which a request must travel.
- These requests are nearly impossible to track with traditional techniques designed for a single service application.
- **Distributed tracing solutions** solve this problem, and numerous other performance issues, because it can track requests through each service or module and provide an end-to-end narrative account of that request.
- **Distributed tracing, sometimes called distributed request tracing, is a method to monitor applications built on a microservices architecture.**
- IT and DevOps teams use distributed tracing to follow the course of a request or transaction as it travels through the application that is being monitored.
- Analysts, SREs, developers and others can observe each iteration of a function, enabling them to conduct performance monitoring by seeing which instance of that function is causing the app to slow down or fail, and how to resolve it.

Monitoring in Kubernetes [Self Study – Exercise]

1. **Kunernetes Monitoring -** https://logz.io/blog/kubernetes-monitoring/

2. **Prometheus and Grafana setup in Minikube -** https://brain2life.hashnode.dev/prometheus-and-grafana-setup-in-minikube

Logs and Traces

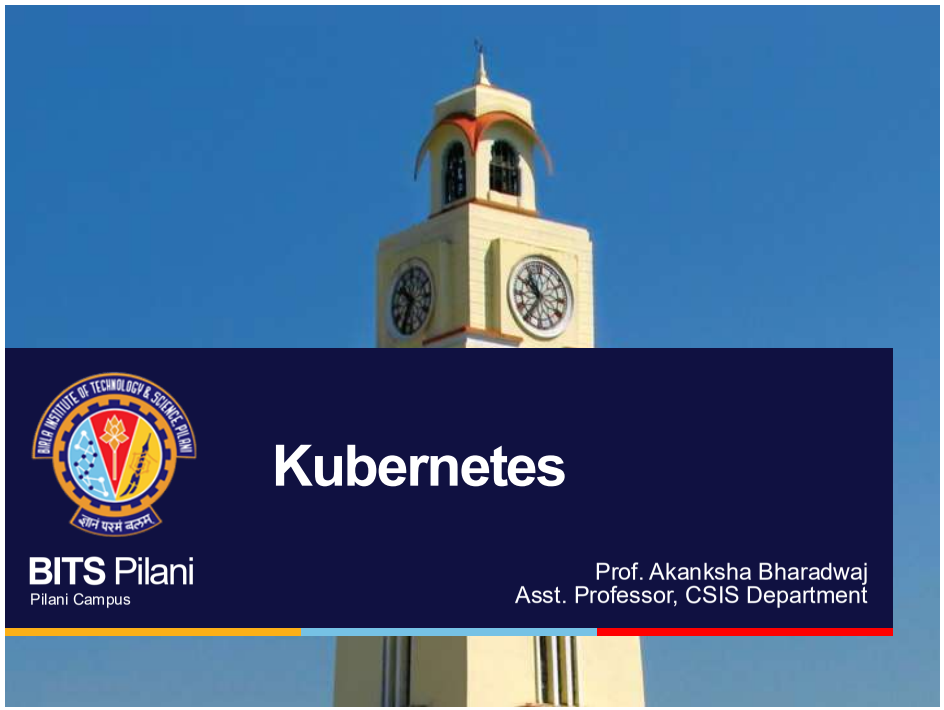
Logging	Tracing
Consumed primarily by system administrators	Consumed primarily by developers
Logs "high level" information (e.g. failed installation of a program)	Logs "low level" information (e.g. a thrown exception)
Must not be too "noisy" (containing many duplicate events or information that is not helpful for its intended audience)	Can be noisy
Event log messages are often localized	Localization is rarely a concern
Addition of new types of events, as well as new event messages, need not be agile	Addition of new tracing messages *must* be agile

References

- Book: Chapter 11, 12 Microservices in Action
- Explore more about Prometheus: https://www.youtube.com/@PromLabs

Thank You!

SE ZG583, Scalable Services
Lecture No. 15



What is Kubernetes?

- Container management tool?????
- Kubernetes is a container management tool
- How is it different from Docker?

Agenda

Kubernetes

- What is Kubernetes?
- Deployment of Microservices using Kubernetes
- Scalability in Kubernetes
- Security in Kubernetes
- CI/CD using Kubernetes
- Kubernetes Dashboard

Kubernetes design principles

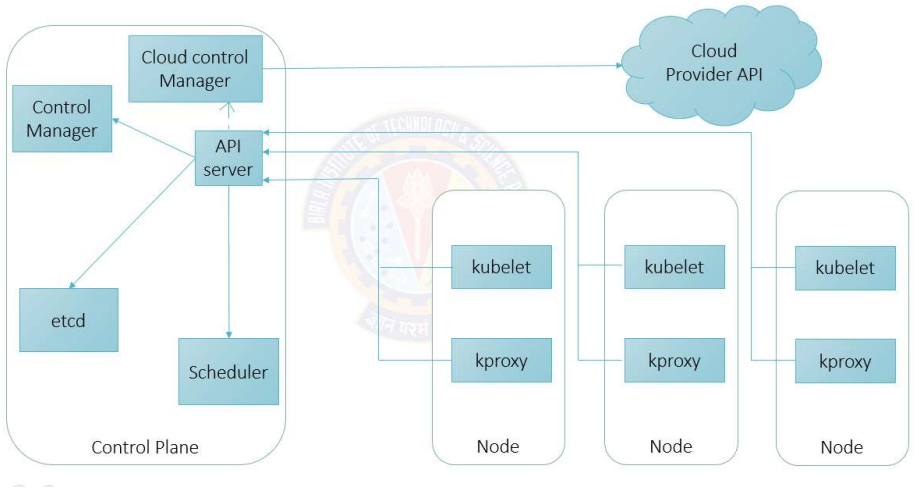
A Kubernetes cluster should be:

- Secure
- Easy to use
- Extendable

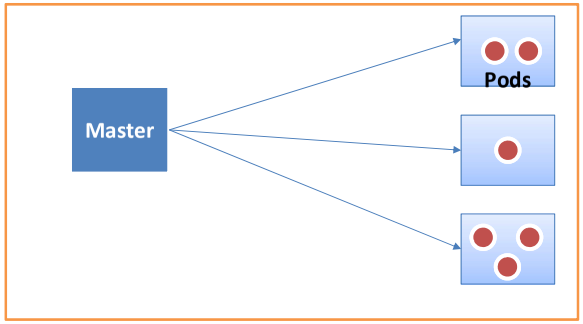
What does K8s provides?

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- Automatic bin packing
- Self-healing
- Secret and configuration management

Kubernetes Architecture

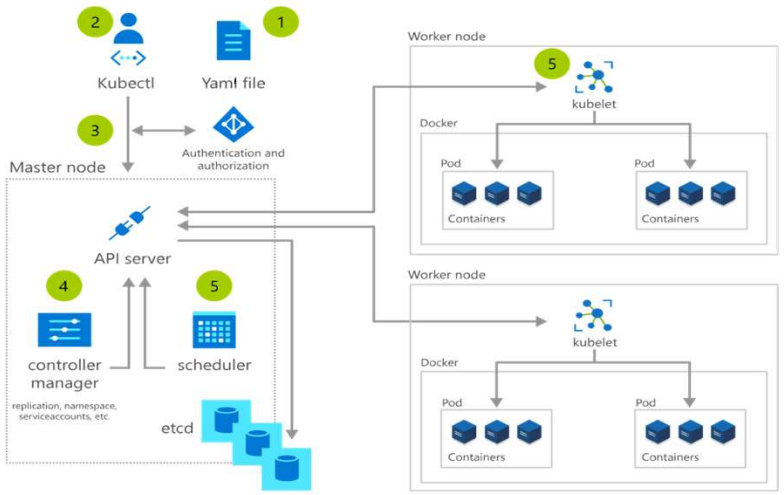


Basic Kubernetes Architecture



Worker Nodes

How Deployment works in Kubernetes



Important elements of a Kubernetes deployment

- YAML file
- Pods
- ReplicaSet
- Kube-controller-manager
- Kube-scheduler
- Rollout

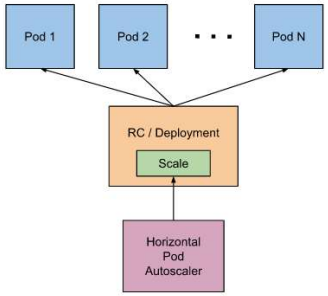
Scalability in Kubernetes

Kubernetes practice lab sources

https://training.play-with-docker.com/
https://training.play-with-kubernetes.com/
https://labs.play-with-k8s.com/

Horizontal Pod Autoscaler

- We can use the Kubernetes Horizontal Pod Autoscaler to automatically scale the number of pods in a deployment.



Introduction

With Auto scaling Kubernetes will automatically scale up your cluster as soon as you need it, and scale it back down to save your money

Types of scaling IN Kubernetes:

- Horizontal Pod Autoscaler (HPA)
- Vertical Pod Autoscaler (VPA)
- Cluster Autoscaler

Cluster Autoscaler

- While HPA scales the number of running pods in a cluster, the cluster autoscaler can change the number of nodes in a cluster.

Vertical Pod Autoscaler

- We can use the Kubernetes Vertical Pod Autoscaler to automatically adjust the resource requests and limits for containers running in a deployment's pods.

Storage in Kubernetes

Introduction

- The Kubernetes storage architecture is based on Volumes as a central abstraction.
- Volumes can be persistent or non-persistent
- Kubernetes allows containers to request storage resources dynamically, using a mechanism called volume claims.

Volumes

- Volumes are the basic entity containers use to access storage in Kubernetes.
- Support for local storage devices, NFS and cloud storage services.
- Regular Volumes are deleted when the Pod hosting them shuts down.

Non-Persistent Storage

- By default, Kubernetes storage is temporary
- Any storage defined as part of a container in a Kubernetes Pod, is held in the host's temporary storage space, which exists as long as the pod exists, and is then removed.
- Container storage is portable, but not durable.

Kubernetes Security

Persistent Volumes and Persistent Volume Claims

- A Persistent Volume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes.
- PVs are volume plugins like Volumes, but have a lifecycle independent of any individual Pod that uses the PV.
- In order for a Pod to start using these volumes, it must request a volume by issuing a persistent volume claim (PVC).

Why Security Matters in Kubernetes

- Dynamic, distributed architecture
- Multiple attack surfaces: API server, etcd, network, containers
- Shared environments and multitenancy

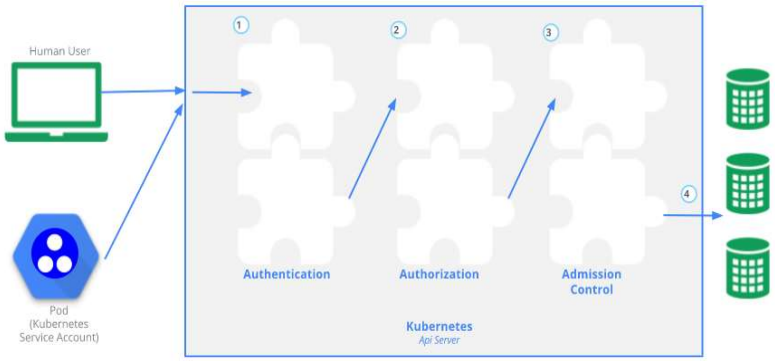
Core Security Areas in Kubernetes

- API Access Control
- Workload Security
- Network Policies
- Secrets Management
- Cluster Hardening

Security Tools & Ecosystem

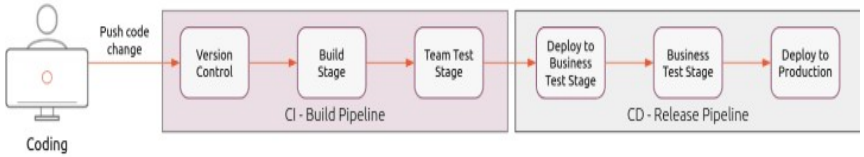
- **Kube-bench** (CIS benchmark checks)
- **Kube-hunter** (penetration testing)
- **Trivy** or **Grype** (image scanning)
- **OPA/Gatekeeper** (policy enforcement)
- **Falco** (runtime security)

Access control in Kubernetes



What is CI/CD pipeline?

- CI/CD pipeline is a series of stages and automated steps software goes through, from code development to production deployment.



Other Security measures in Kubernetes

- Use namespaces to establish security boundaries
- Separate sensitive workloads
- Secure cloud metadata access
- Create and define cluster network policies
- Run a cluster-wide Pod Security Policy
- Harden node security
- Turn on audit logging
- Keep the Kubernetes version up to date

CI/CD with Kubernetes

Why CI/CD in Kubernetes?

- Native support for automation and scaling
- Ideal for microservices and container-based apps
- Easily integrates with modern DevOps tools
- Supports rolling updates, blue-green, and canary deployments

Core CI/CD Components for Kubernetes

- **Source control**: GitHub, GitLab
- **CI servers**: Jenkins, GitHub Actions, GitLab CI, CircleCI
- **Build tools**: Docker, Kaniko, BuildKit
- **Artifact storage**: Docker Hub, Amazon ECR, Harbor
- **CD tools**: ArgoCD, Flux, Spinnaker
- **Kubernetes manifests**: Helm, Kustomize, YAML

CI/CD Workflow in Kubernetes

- Developer pushes code to Git
- CI pipeline runs tests and builds image
- Image is pushed to registry
- CD tool pulls new image
- Kubernetes deployment is updated
- Monitoring and rollback mechanisms activated

Testing in CI/CD Pipelines

- Unit & integration tests
- Linting (YAML, Helm)
- Security scans (Trivy, Snyk)
- Kubernetes manifest validation
- Dry runs (kubectl apply --dry-run)
- Canary tests in production

GitOps Approach to CD

- Uses Git as the **single source of truth** for deployments
- Tools like **ArgoCD** and **Flux** sync cluster state with Git
- Benefits: Auditable, repeatable, easy rollbacks

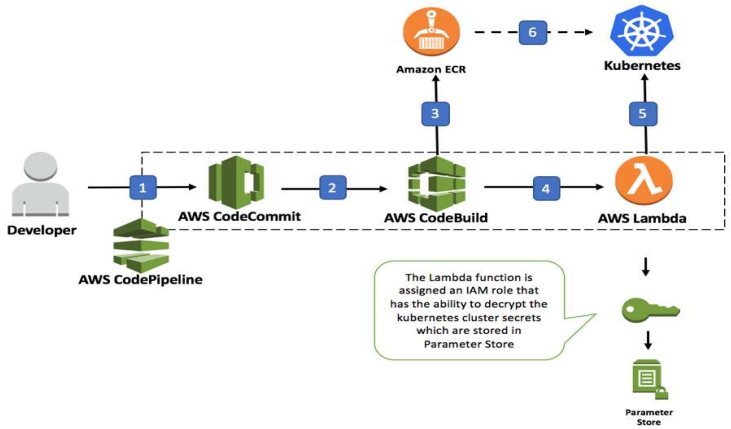
Packaging & Deployment

- Use **Helm charts** or **Kustomize** to manage deployment logic
- Automate rollouts with **ArgoCD**, **Flux**, or native Kubernetes
- Support for **blue-green**, **canary**, and **rolling updates**

Security & Observability in CI/CD

- Image scanning in CI (e.g., Trivy)
- RBAC and Secrets Management in CD
- Monitoring with Proametheus/Grafana
- Logging with Fluentd, Loki
- Alerting with Alertmanager or Slack integration

Example of Continuous Deployment to Kubernetes



Infrastructure as Code in Kubernetes

- Kubernetes infrastructure and configuration are managed as code using tools like Terraform or Kubernetes-specific tools (e.g., kops, kubectl apply) to provision and manage Kubernetes clusters and associated resources.

Deploying the Dashboard UI

- Command:

kubectl apply -f https://raw.githubusercontent.com/kubernetes/dashboard/v2.3.1/aio/deploy/recommended.yaml

Kubernetes dashboard

- Dashboard is a web-based Kubernetes user interface.
- You can use Dashboard to get an overview of applications running on your cluster, as well as for creating or modifying individual Kubernetes resources
- Dashboard also provides information on the state of Kubernetes resources in your cluster and on any errors that may have occurred.

References

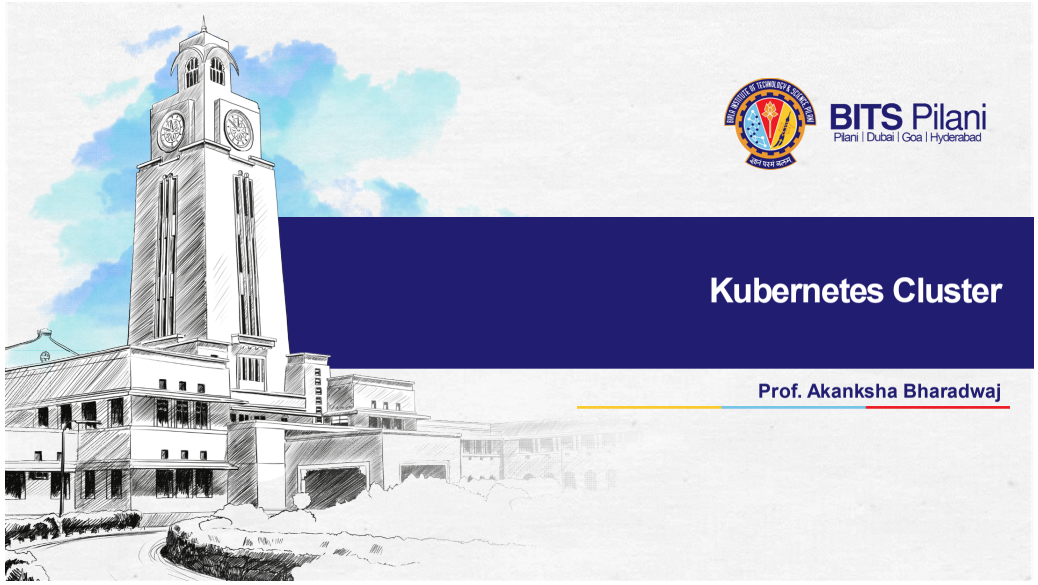
1. Link: https://kubernetes.io/docs
2. Book: Kubernetes in Action by Marko Luksa
3. Link: https://azure.microsoft.com/en-in/overview/kubernetes-deployment-strategy/
4. Link: https://aws.amazon.com/blogs/devops/continuous-deployment-to-kubernetes-using-aws-codepipeline-aws-codecommit-aws-codebuild-amazon-ecr-and-aws-lambda

Self Study

- https://www.magalix.com/blog/create-a-ci/cd-pipeline-with-kubernetes-and-jenkins
- https://ubuntu.com/tutorials/gitlab-cicd-pipelines-with-microk8s#1-overview
- https://www.cncf.io/blog/2021/03/22/kubernetes-security/
- https://www.katacoda.com/courses/kubernetes/launch-single-node-cluster
- Article: https://techynurse.medium.com/deploying-microservices-to-kubernetes-minikube-b807994ee4bd
- Article: https://techdozo.dev/building-and-deploying-microservices-application-locally-in-the-kubernetes/

Agenda

- How to create Kubernetes Cluster on AWS
- Create a Kubernetes cluster on AWS using eksctl
- How to deploy an application on Kubernetes (cloud)



BITS Pilani
Pilani | Dubai | Goa | Hyderabad

Kubernetes Cluster

Prof. Akanksha Bharadwaj

How to create an Amazon EKS cluster

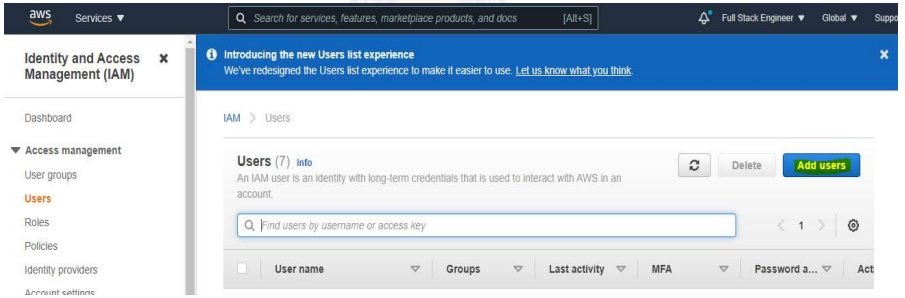
- Create using eksctl
- Create using AWS management Console
- Create using AWS CLI

https://docs.aws.amazon.com/eks/latest/userguide/create-cluster.html

Create a Kubernetes cluster on AWS using eksctl

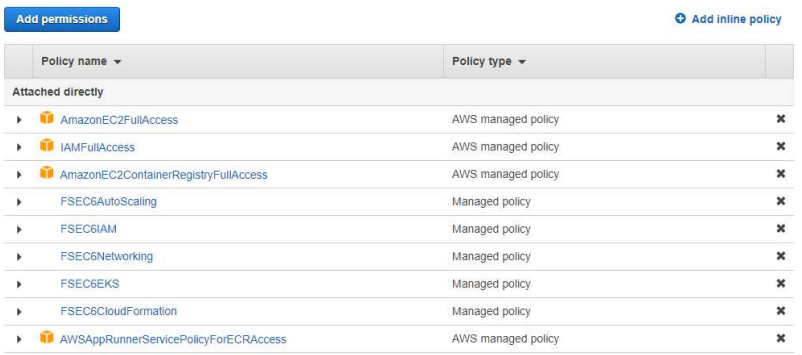
Pre-requisites

- Create an AWS account
- Create a user in IAM management console of AWS and download csv file that contains the AWS Secret Access Key



Pre-requisites

- Assign minimum IAM policies required for eksctl usage



Pre-requisites

Install kubectl- – A command line tool for working with Kubernetes clusters. Use the latest version. I used Kubernetes version 1.20

- Open a PowerShell terminal in Windows
- Download the Amazon EKS vended kubectl binary for your cluster's Kubernetes version from Amazon S3. Command for windows:

curl -o kubectl.exe https://amazon-eks.s3.us-west-2.amazonaws.com/1.20.4/2021-04-12/bin/windows/amd64/kubectl.exe

- Download the SHA-256 sum for your cluster's Kubernetes version for Windows. Command on windows

curl -o kubectl.exe.sha256 https://amazon-eks.s3.us-west-2.amazonaws.com/1.20.4/2021-04-12/bin/windows/amd64/kubectl.exe.sha256

- Check the SHA-256 sum for your downloaded binary. Compare the generated SHA-256 sum in the command output against your downloaded SHA-256 file. The two should match, although the PowerShell output will be uppercase.

Get-FileHash kubectl.exe

- Edit your system PATH environment variable to add the directory containing kubectl.exe binary
- Check the download by using:

```
PS C:\Users\admin> kubectl version --short --client
Client Version: v1.20.4-eks-6b7464
```


Pre-requisites

- Download, Install and configure AWS CLI
- Check the AWS CLI version to confirm

```
PS C:\Users\admin> aws --version
aws-cli/2.2.20 Python/3.8.8 Windows/10 exe/AMD64 prompt/off
```

- To configure use the below-mentioned command and then specify AWS Access key id (get from IAM user), AWS Secret Access Key (get from IAM user), default region and default format

aws configure

```
PS C:\Users\admin> aws configure
AWS Access Key ID [****************IQEV]: [redacted]
AWS Secret Access Key [****************3ZYM]: [redacted]
Default region name [us-east-1]: us-east-1
Default output format [json]: json
PS C:\Users\admin>
```

Access key ID	Created	Last used	Status
[redacted]	2021-07-27 15:59 UTC+0530	2021-07-28 16:12 UTC+0530 with sts in us-east-1	Active | Make inactive

Create your Amazon EKS cluster and nodes

To create your cluster with Amazon EC2 Linux managed nodes

- Create key-pair with the following command: *aws ec2 create-key-pair --region us-east-1 --key-name myKeyPair*

```
PS C:\Users\admin\awskubernetes\k8s-postman-tutorial> aws ec2 create-key-pair --region us-east-1 --key-name myKeyPair
{
    "KeyFingerprint": "ac:86:22:ab:1b:14:85:d5:01:cc:69:8f:4d:64:57:3f:58:f4:70:ef",
    "KeyMaterial": "-----BEGIN RSA PRIVATE KEY-----\nMIIEogIBAAKCAQEAgOFlX6czsGsZlFTuQNCczIabFKf/QX7dtqNLV7S4U2eKJDgs\nQy4
QJjWzc65KQycm7mgxucMvSee6DH/FpErrE69KDE1g+wB+9SupdPS3+h7EeIgp\n41Ft0gj7pa6tL8DsoI+m2DdqcAHdJMdkWRgnpV8hAZN7+2Yi9PZaWyO5jqp
WIfRN\nwS7XNnYD81B8rjNjFLML/qWirP6yqE28c+eHs12305xYRp4hjc7GDc/jofloy+KJ\nDTPm3CBPcCSieii4k+UXSH/F3YB59BRRYwPe3CTm15dIrnV4H
cUGQR0NhPewlJc7\nBDwwmR156YgIQ5iTZUsDqgWkJ8BKRBGuJq8ktwIDAQABAoIBAHYho5DsqmVPle+v\ndX81eDPEQcvKPR8kkUcCIGIxBS913Vpbnr6pLyZ
G3Mjw/aLV9UpbeRDDhsngPFmO\nmHOCND0j0+S43GbVyokhTz8jvKrnnsC4VmTFU7tMfpUcKiR2Wq7FXXIcQVSPcMtw\ndFtx140uSAuwytjhgrM0SBf4A/Jug
22weH9wLQ5A1FFv52L8X8rfqiEQntjS82aW\nmF6Ema2on6P+FaOYsirLt6XDJ6codDL48SOpkab1rw/EadDPS4IBZKYzmyW+F33H\n6Guja0TGURt2DBB1zJb
bjFINGFphe99XtJZhOUQqz77TNnTzLJ1+cUDBA6y5jUO2\n3goZtIkCgYEAxEcVLNlzev7Odspd/34sLHlDgfTMCI3JJTZgHflZEQlpQZdEpB6H\nYKhbsM73Q
IOm+jLy81P2GN4AiBjlts+yxXm5XAU0d5zNQCgT+a2n9+ma145hCb8U\nUIbI7BG7OzpNYtBGjpNUEV28iCdjqy74bqSu1/aJp1HPNxW/UCjSPiMCgYEAqBhz\
n1kwevTs/XouQg9lX8Bw+3xfWdRP2sDezn2/nDr46/qXNSasi+ynS0p8ZoSKtSuZ1\nXPEfnBY44SoQ+CFSO0nhT+5sPx0sl3AmG7cZMgGuv5/YhZ/uPne3dry
JwWI+2aE8\nOOzTuAbLT4QA5JUcXyr5xRN8GupVJXvFnYP7Rl0CgYA91HqaL0IOm9gkWal0w6Vc\nuTKH7LqdczTLES8EIjlpgYJz1BgebzOMQlzn/Lqsyww9w
vD/X22l9cFLHjS2+wTA\n6pB2qXkWuQF7VWS9gXPpR+CsSMDO1i+rElqW6xsFFF+uYhfn8AdKlTG/3lCqa1h9\nWklt5M8GM2JIYwRqVwGr6wKBgGocFXNfVcA
qWFQ8UZuJZYOCsQVvLMd3NLAEWXT5\n1Kdv2hnJXA924nwd6KFgIXOgSjgPxmJHK6cCOCGaxl972Ubn2jz0yowExJNMDiJM\nsLDqcv8vziR70o7IT526YlMay
4Xhy/w7M2Lllw24zD9/CQviOwz2wc7TwbAJs7M9\nDy9ZAoGABcbo6zK+JD356rJOiXBxS9m4A2ixiI+jIo9bRcxHedtij2dVs68xuZ1q\nMqRrsREmmLsKz7a
pOmjR3cAPHvILz8Z7MZ9e8Eja7KdpUpIe0UbEnrtaISv8oSGN\nMJ75a/dm+4VmxEutTnymiOMtBN3kmEJjDUr/yPuNf6vEP3lxlfk=\n-----END RSA PRIV
ATE KEY-----",
    "KeyName": "myKeyPair",
    "KeyPairId": "[redacted]"
}
```


Pre-requisites

Install eksctl – A command line tool for working with EKS clusters that automates many individual tasks. Use version 0.58 or later

- Install the binaries with the following command: choco install -y eksctl
- If they are already installed, run the following command to upgrade: choco upgrade -y eksctl
- Test that your installation was successful with the following command: eksctl version

```
PS C:\Users\admin> eksctl version
0.58.0
```

Create your Amazon EKS cluster and nodes

- Create cluster command: *eksctl create cluster --name test1-cluster --region us-east-1 --zones=us-east-1a,us-east-1b --with-oidc --ssh-access --ssh-public-key myKeyPair --managed*

```
PS C:\Users\admin\awskubernetes\k8s-postman-tutorial> eksctl create cluster --name test1-cluster --region us-east-1 --zone
s=us-east-1a,us-east-1b --with-oidc --ssh-access --ssh-public-key myKeyPair --managed
2021-07-28 12:18:47 [ℹ]  eksctl version 0.58.0
2021-07-28 12:18:47 [ℹ]  using region us-east-1
2021-07-28 12:18:47 [ℹ]  subnets for us-east-1a - public:192.168.0.0/19 private:192.168.64.0/19
2021-07-28 12:18:47 [ℹ]  subnets for us-east-1b - public:192.168.32.0/19 private:192.168.96.0/19
2021-07-28 12:18:47 [ℹ]  nodegroup "ng-e4f43d07" will use "" [AmazonLinux2/1.20]
2021-07-28 12:18:49 [ℹ]  using EC2 key pair %!q(*string=<nil>)
2021-07-28 12:18:49 [ℹ]  using Kubernetes version 1.20
2021-07-28 12:18:49 [ℹ]  creating EKS cluster "test1-cluster" in "us-east-1" region with managed nodes
2021-07-28 12:18:49 [ℹ]  will create 2 separate CloudFormation stacks for cluster itself and the initial managed nodegroup

2021-07-28 12:18:49 [ℹ]  if you encounter any issues, check CloudFormation console or try 'eksctl utils describe-stacks --
region=us-east-1 --cluster=test1-cluster'
2021-07-28 12:18:49 [ℹ]  CloudWatch logging will not be enabled for cluster "test1-cluster" in "us-east-1"
2021-07-28 12:18:49 [ℹ]  you can enable it with 'eksctl utils update-cluster-logging --enable-types={SPECIFY-YOUR-LOG-TYPE
S-HERE (e.g. all)} --region=us-east-1 --cluster=test1-cluster'
2021-07-28 12:18:49 [ℹ]  Kubernetes API endpoint access will use default of {publicAccess=true, privateAccess=false} for c
luster "test1-cluster" in "us-east-1"
```


Create your Amazon EKS cluster and nodes

```
2021-07-28 12:43:21 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:43:40 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:44:02 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:44:21 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:44:39 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:45:01 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:45:21 [ℹ]  waiting for CloudFormation stack "eksctl-test1-cluster-nodegroup-ng-e4f43d07"
2021-07-28 12:45:24 [ℹ]  waiting for the control plane availability...
2021-07-28 12:45:24 [✔]  saved kubeconfig as "C:\\Users\\admin\\.kube\\config"
2021-07-28 12:45:24 [ℹ]  no tasks
2021-07-28 12:45:24 [✔]  all EKS cluster resources for "test1-cluster" have been created
2021-07-28 12:45:26 [ℹ]  nodegroup "ng-e4f43d07" has 2 node(s)
2021-07-28 12:45:26 [ℹ]  node "ip-192-168-0-231.ec2.internal" is ready
2021-07-28 12:45:26 [ℹ]  node "ip-192-168-43-39.ec2.internal" is ready
2021-07-28 12:45:26 [ℹ]  waiting for at least 2 node(s) to become ready in "ng-e4f43d07"
2021-07-28 12:45:26 [ℹ]  nodegroup "ng-e4f43d07" has 2 node(s)
2021-07-28 12:45:26 [ℹ]  node "ip-192-168-0-231.ec2.internal" is ready
2021-07-28 12:45:26 [ℹ]  node "ip-192-168-43-39.ec2.internal" is ready
2021-07-28 12:47:46 [ℹ]  kubectl command should work with "C:\\Users\\admin\\.kube\\config", try 'kubectl get nodes'
2021-07-28 12:47:46 [✔]  EKS cluster "test1-cluster" in "us-east-1" region is ready
```


View resources

- get cluster details: eksctl get cluster --region us-east-1

```
PS C:\Users\admin\awskubernetes\k8s-postman-tutorial> eksctl get cluster --region us-east-1
2021-07-28 12:58:11 [ℹ]  eksctl version 0.58.0
2021-07-28 12:58:11 [ℹ]  using region us-east-1
NAME            REGION          EKSCTL CREATED
test1-cluster   us-east-1       True
```


View resources

- kubectl get nodes -o wide

```
PS C:\Users\admin\awskubernetes\k8s-postman-tutorial> kubectl get nodes -o wide
NAME                           STATUS   ROLES    AGE   VERSION              INTERNAL-IP     EXTERNAL-IP      OS-IMAGE
     KERNEL-VERSION                CONTAINER-RUNTIME
ip-192-168-0-231.ec2.internal   Ready    <none>   11m   v1.20.4-eks-6b7464   192.168.0.231   54.227.203.142   Amazon Linux
 2   5.4.129-63.229.amzn2.x86_64   docker://19.3.13
ip-192-168-43-39.ec2.internal   Ready    <none>   11m   v1.20.4-eks-6b7464   192.168.43.39   54.174.148.158   Amazon Linux
 2   5.4.129-63.229.amzn2.x86_64   docker://19.3.13
```

- kubectl get pods --all-namespaces -o wide

```
PS C:\Users\admin\awskubernetes\k8s-postman-tutorial> kubectl get pods --all-namespaces -o wide
NAMESPACE     NAME                       READY   STATUS    RESTARTS   AGE   IP                NODE
   NOMINATED NODE   READINESS GATES
kube-system   aws-node-dhhxz             1/1     Running   0          11m   192.168.0.231     ip-192-168-0-231.ec2.internal
   <none>           <none>
kube-system   aws-node-m7wqh             1/1     Running   0          11m   192.168.43.39     ip-192-168-43-39.ec2.internal
   <none>           <none>
kube-system   coredns-65bfc5645f-7z84q   1/1     Running   0          26m   192.168.27.57     ip-192-168-0-231.ec2.internal
   <none>           <none>
kube-system   coredns-65bfc5645f-84dkl   1/1     Running   0          26m   192.168.24.205    ip-192-168-0-231.ec2.internal
   <none>           <none>
kube-system   kube-proxy-6fd7k           1/1     Running   0          11m   192.168.0.231     ip-192-168-0-231.ec2.internal
   <none>           <none>
kube-system   kube-proxy-bh4wr           1/1     Running   0          11m   192.168.43.39     ip-192-168-43-39.ec2.internal
   <none>           <none>
```

How to deploy an application on Kubernetes (cloud)

Deploy application

Building and containerizing the microservices

The first step of deploying to Kubernetes is to build your microservices and containerize them. So every microservice should be made into a docker image with the relevant ports that should be exposed configured.

Example:

docker build -t system:1.0-SNAPSHOT system/.

Deploy application

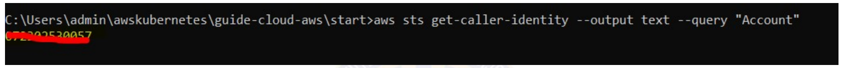
Pushing the images to a container registry

- First, you must authenticate your Docker client to your ECR registry. Start by running the get-login command:

aws ecr get-login-password --region ap-south-1

Deploy application

- get AWS account_id: aws sts get-caller-identity --output text --query "Account"



- Authenticate your docker client: docker login -u AWS -p [password_string] https://[aws_account_id].dkr.ecr.[region].amazonaws.com

Deploy application

Make a repository to store the images

Example: *aws ecr create-repository --repository-name awsguide/system*

```
C:\Users\admin\awskubernetes\guide-cloud-aws\start>aws ecr create-repository --repository-name awsguide/system
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:072302530057:repository/awsguide/system",
        "registryId": "072302530057",
        "repositoryName": "awsguide/system",
        "repositoryUri": "072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/system",
        "createdAt": "2021-07-28T15:50:37+05:30",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": false
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}

C:\Users\admin\awskubernetes\guide-cloud-aws\start>aws ecr create-repository --repository-name awsguide/inventory
{
    "repository": {
        "repositoryArn": "arn:aws:ecr:us-east-1:072302530057:repository/awsguide/inventory",
        "registryId": "072302530057",
        "repositoryName": "awsguide/inventory",
        "repositoryUri": "072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/inventory",
        "createdAt": "2021-07-28T15:51:30+05:30",
        "imageTagMutability": "MUTABLE",
        "imageScanningConfiguration": {
            "scanOnPush": false
        },
        "encryptionConfiguration": {
            "encryptionType": "AES256"
        }
    }
}
```


Deploy application

Tag your container images

Next, you need to tag your container images with the relevant data about your registry:

Example :*docker tag system:1.0-SNAPSHOT 072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/system:1.0-SNAPSHOT*

```
C:\Users\admin\awskubernetes\guide-cloud-aws\start>docker tag system:1.0-SNAPSHOT 072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/system:1.0-SNAPSHOT

C:\Users\admin\awskubernetes\guide-cloud-aws\start>docker tag inventory:1.0-SNAPSHOT 072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/inventory:1.0-SNAPSHOT
```


Deploy application

Push your images to the registry

Finally, push your images to the registry:

Example:

docker push [system-repository-uri]:1.0-SNAPSHOT

```
C:\Users\admin\awskubernetes\guide-cloud-aws\start>docker push 072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/system:1.0-SNAPSHOT
The push refers to repository [072302530057.dkr.ecr.us-east-1.amazonaws.com/awsguide/system]
13906170d397: Preparing
d095c05cad95: Preparing
01a0c009d9d2: Preparing
01bb95e4e4fc: Preparing
4d90ddb5aac8: Preparing
bc63f37a077d: Preparing
bf17a30101db: Preparing
4a35cfa1e747: Preparing
49be94cb2ec3: Waiting
d5190c2002ef: Waiting
5711c4a69233: Waiting
a6b5c337ed78: Waiting
5b2f0b9e81c7: Waiting
d7b836ae9a00: Waiting
no basic auth credentials
```


Understanding Kubernetes Yaml file

Creating Deployment Example file

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myapp-deployment
  labels:
    app: myapp
spec:
  replicas: 3
  selector:
    matchLabels:
      app: myapp
  template:
    metadata:
      labels:
        app: myapp
    spec:
      containers:
      - name: myapp
        image: nginx
```

Deploying the microservices

- Prepare kubernetes.yaml file.
- kubectl apply -f kubernetes.yaml

```
C:\Users\admin\awskubernetes\guide-cloud-aws\start>kubectl apply -f kubernetes.yaml
deployment.apps/system-deployment created
deployment.apps/inventory-deployment created
service/system-service created
service/inventory-service created
```

•kubectl get pods

```
C:\Users\admin\awskubernetes\guide-cloud-aws\start>kubectl get pods
NAME                                  READY   STATUS             RESTARTS   AGE
inventory-deployment-559dfc694d-259dl   0/1     ImagePullBackOff   0          47s
system-deployment-8655bc87d5-27kq5      0/1     ImagePullBackOff   0          48s
```


Understanding Kubernetes Yaml file

Creating Services Example file

```
apiVersion: v1
kind: Service
metadata:
  name: myapp-clusterip-srv
spec:
  selector:
    app: myapp
  type: ClusterIP
  ports:
  - name: myapp-clusterip
    protocol: TCP
    port: 4000
    targetPort: 5000
```


Clean up

- When you no longer need your deployed microservices, you can delete all Kubernetes resources by running the kubectl delete command:

  kubectl delete -f kubernetes.yaml

- Delete the ECR repositories used to store the images:

  Example: *aws ecr delete-repository --repository-name awsguide/system --force*

Remove your EKS cluster:

Example: *eksctl delete cluster --name guide-cluster*

Deploying new version of an existing microservice

- Build the new version of the container image

  Example: *docker build -t system:2.0-SNAPSHOT system/.*

- Tag your container image with the relevant data about your registry:

  Example: *docker tag system:2.0-SNAPSHOT [system-repository-uri]:2.0-SNAPSHOT*

- Push your image to the registry

  Example: *docker push [system-repository-uri]:2.0-SNAPSHOT*

- Update the deployment to use the new container image that you just pushed to the registry

  Example: *kubectl set image deployment/system-deployment system-container=[system-repository-uri]:2.0-SNAPSHOT*

- Use the following command to find the name of the pod that is running the system microservice:

  kubectl get pods

Amazon EKS security best practices

IAM administrators control who can be *authenticated* (signed in) and *authorized* (have permissions) to use Amazon EKS resources. IAM is an AWS service that you can use with no additional charge.

Restricting access to the Amazon EC2 instance metadata service (IMDS)

Recommendation is to block pod access to IMDS to minimize the permissions available to your containers if:

- You've implemented IAM roles for service accounts and have assigned necessary permissions directly to all pods that require access to AWS services.
- No pods in your cluster require access to IMDS for other reasons, such as retrieving the current Region.

Best Practices to use Kubernetes Cluster

- Use the latest version
- Namespaces
- Small container images
- Resource requests and limit
- Autoscale
- Monitoring
- Use readiness Probe
- Use liveliness Probe
- Use Role Based Access Controls
- Git-based workflow

Example container spec

```
containers:
- name: ABC
  image: XYZ
  resources:
    requests:
      memory: "256Mi"
      cpu: "300m"
    limits:
      memory: "1024Mi"
      cpu: "900m"
```


References

1. Link: https://kubernetes.io/docs/home/
2. Link: https://docs.aws.amazon.com/eks/latest/userguide
3. Link: https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html
4. Link: https://openliberty.io/

Self Study

- Exercise: https://kubernetes.io/docs/tutorials/stateless-application/guestbook/
- Exercise: https://kubernetes.io/docs/tutorials/stateful-application/mysql-wordpress-persistent-volume/
- Exercise: https://documenter.getpostman.com/view/13059338/TVmLCyNi
- Kubernetes dashboard: https://docs.aws.amazon.com/eks/latest/userguide/dashboard-tutorial.html
- Security in AWS EKS: https://docs.aws.amazon.com/eks/latest/userguide/security.html
- Horizontal pod autoscaler: https://docs.aws.amazon.com/eks/latest/userguide/horizontal-pod-autoscaler.html
- CI/CD: https://aws.amazon.com/blogs/devops/ci-cd-on-amazon-eks-using-aws-codecommit-aws-codepipeline-aws-codebuild-and-fluxcd/
- https://kubernetes.io/docs/setup/#production-environment

Cloud Computing - Notes on the CAP Theorem
Prof. Douglas Thain, March 2016

Caution: These are high level notes that I use to organize my lectures. You main find them useful for reviewing main concepts, but they aren't a substitute for participating in class.

References

1. Eric Brewer, Towards robust distributed systems. In Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (July 16-19, Portland, Oregon): 7
2. Werner Vogels, Eventually Consistent, Communications of the ACM, Volume 52, Number 1, January 2009.
3. Eric Brewer, CAP Twelve Years Later: How the "Rules" Have Changed, IEEE Explore, Volume 45, Issue 2, 2012.
4. Chapter 7 of Tanenbaum and Steen, "Distributed Systems: Principles and Paradigms", Pearson, 2007.

The CAP Theorem

The CAP theorem[1] is an observation about the tradeoffs inherent in designing a distributed system for storing data. Simply put, the CAP theorem states that a given system design involves a tradeoff between the desirable properties of Consistency, Availability, and Partitionability. A given system cannot maximize all three of these properties simultaneously. The concept is generally attributed to Eric Brewer, who posed it as a conjecture at the PODC conference in 2000, building on ideas of consistency and performance that had been explored by systems and databases for some time.

The three properties may be loosely defined as follows:

Consistency – All clients of the system see the same data.
Availability – Clients are able to access and update data rapidly.
Partitionability – The system is able to operate even when the network fails.

You may sometimes hear this explained as "pick two of three" but that is an over-simplification. Each of these properties exists upon a continuum, and attempting to increase one of them requires decreasing another to a certain degree.

This is all very abstract, and best explained with some examples. To get the examples right, we will have to discuss them in some detail, and then come back to discuss the CAP theorem more generally.

[1] The CAP "theorem" isn't really a proper theorem, since it uses some rather fuzzy

Example System 1 – Direct Access

Let's start off with a model of a distributed file system. Suppose that we have a server S that contains some files named X, Y, and Z. The server is accessible via a network to clients A, B, and C. The clients occasionally wish to modify those files, so they can send messages like "read X" or "write Y" to change those files. As the server receives these requests, it sends as response message back to the client, indicating that the change is complete. The client waits for the response to come back before attempting another request.

(Sketch the system and some example interactions here.)

Now, keep in mind that these messages flow over a network. This has two effects on the users of the system:

- First, the network imposes some minimum latency (let's say 1ms) on each message. As a result, reading and writing a large amount of data from this file server is going to be very slow – much slower than accessing a local disk.
- Second, the network makes message delivery unreliable. Either a request or a response could be delayed arbitrarily, or completely dropped. In the case of a backhoe cutting through a network cable, there may be no communication at all along a particularly link. If a client doesn't receive a response to a message, then it has no choice but to wait and try the request again. It could wait a very long time!

Let's evaluate this system according to the CAP criteria:

- Consistency – HIGH – Example 1 is highly consistent because every operation is applied in a known order, and all clients have an unfiltered view of the central server.
- Availability – LOW – Every single read or write requires a network operation, making this system much slower than accessing a local file system.
- Partitionability – MEDIUM – If a single client is partitioned from the file server, it cannot perform any operations. However, all other connected clients are able to continue.

(The careful reader will note that we have not yet defined precisely what constitutes higher or lower for each of these three properties. Bear with me and let that be vague for a while, and we will formalize it more below.)

Example System 2 – Write-Through Cache

Take the system from example 1, and modify it by adding a cache of finite capacity to each client node. Let's give each client some simple logic for managing these caches:

- Read – When a client attempts to read a file, it first looks in its cache to see if that file's data is already present. If so, the read is satisfied from that data. If not, then the client issues a read request to the server, waits for the response, and replaces the least recently used (LRU) item in the cache.
- Write – When a client attempts to write a file, it first issues a write command to the server, and waits for a response. If an older version of the file exists in the cache, it is updated to the new value. If not, then the client replaces the LRU item in the cache with the newly written value.

(Sketch the system and some example operations here.)

- Consistency – MEDIUM – System 2 (Write Through Cache) is less consistent than System 1 (Direct Access) because a client may fail to see writes made by other clients, when a value is available within its own cache.
- Availability – MEDIUM – System 2 will see much better read performance than System 1, because reads can be satisfied directly from cache without consulting the central server. However, writes are no faster because they always result in a network operation.
- Partitionability – MEDIUM – A client might be able to continue operating even when the network is down, if it is only performing reads on cached data. However, any write operation must block until the network results.

Thought Experiment: Consider a modification of System 2 in which clients write data only to their local caches, and send data back to the server only when evicted from the cache. How would this affect the CAP properties?

Example System 3 – Consistent Caches

The previous system had a very simple method of dealing with cached data that (obviously) resulted in some serious inconsistencies. Let's try to solve the problem in a different way, by making the system avoid inconsistencies in cached data.

Take the design from System 2, where each client has a cache of finite capacity. Now, when each client tries to read or write a file, do this:

- Read – (Same as System 2)
- Write – When a client attempts to modify a file, it first sends a message to every other client, instructing it to invalidate its cached copy of that file. Once those caches acknowledge that they have purged their copies of the file, the client sends the write request to the server, and updates its own cache.

(Sketch the system and some example operations here.)

Ok, now you evaluate this system according to the CAP criteria:

- Consistency?
- Availability?
- Partitionability?

Thought Experiment: Now you design a system that has a combination of CAP that we haven't seen so far.

How do we quantify CAP?

Now that we have considered several model systems, it should be clear that there is a real tradeoff between Consistency, Availability, and Partitionability. Dealing with the P is central to distributed computing: when we cannot communicate, should we optimistically try to make progress, or pessimistically wait, in order to achieve consistency? There is no single answer to this problem: different applications will require different solutions.

In the analysis above, we were a bit sloppy about stating exactly what is meant by "more" or "less" of each of the three properties.

Availability is probably the easiest to quantify. For a given system, one could measure every attempt to read or write a value, and then compute a statistic like the mean, median, or 99th percentile of latency for various operations. A system that provides a lower mean is providing "more" availability.

Partition tolerance might be measured by evaluating what set of clients and operations can continue in the presence of network outages. A system that allows all clients to perform reads during a network outage is providing "more" partition tolerance than a system that allows no clients to perform reads.

Consistency is the most complex property to describe. There exist a variety of consistency models that can be implemented by adjusting just how and when caches are updated, and whether clients can continue to operate during a partition.

Strong Consistency:
Once an update is complete, all clients will see that new value.

Causal Consistency:
If process A tells B that it has updated X, then B will see the latest value of X.

Read-Your-Writes:
If process A updates X, then A will never see an older value of X.

Monotonic Reads:
If process A reads a value from X, then it will never read back an older value.

Monotonic Writes:
All writes by process A are applied in the order they are given.

Eventual Consistency:
All updates will become visible to everyone, if you wait long enough.

Chaos:
No guarantees!

Storage Replication

Many distributed systems replicate data across a large number of storage devices. This is done to provide insurance against storage failures, but also to provide a high degree of availability for commonly used data.

But multiple copies results in a new kind of consistency problem. If I update 2 out of 5 copies, is the write "complete" or not? Systems answer this question in various ways, which are known as quorum protocols.

Vogels [2] gives a framework for thinking about this:

Suppose you have a system with N replicated storage units. To update an item, a client must write W of the replicas upfront. (The remainder will get updated eventually in the background.) To read an item, a client must read R of the replicas, in order to decide whether the most recent value has been read. (If the values differ, assume you can tell which one is the newest.)

For example:

N=2, W=2, R=1 is a strongly consistent system: a writer must update both replicas, and a reader can read either one of them. (RAID 1)

N=2, W=1, R=2 is also a strongly consistent system: a writer can update either replica, and a reader must read both to obtain the latest.

N=2, W=1, R=1 is an eventually consistent system: the writer can update either replica, and the reader can read either replica, so you may not see consistent results.

And now some general observations:

(W + R) > N is strongly consistent ,while (W+R) <= N is weakly consistent.

W < (N+1)/2 means write conflicts can occur.

Monotonic read (and write) consistency is achieved by making clients "sticky" with respect to the replicas that they use.

Strong consistency models can only be achieved by delaying either reads or writes during partitions. In a large enough distributed system, partitions are omnipresent. Ergo, very large distributed systems almost always rely on weak or eventual consistency in order to achieve acceptable availability.