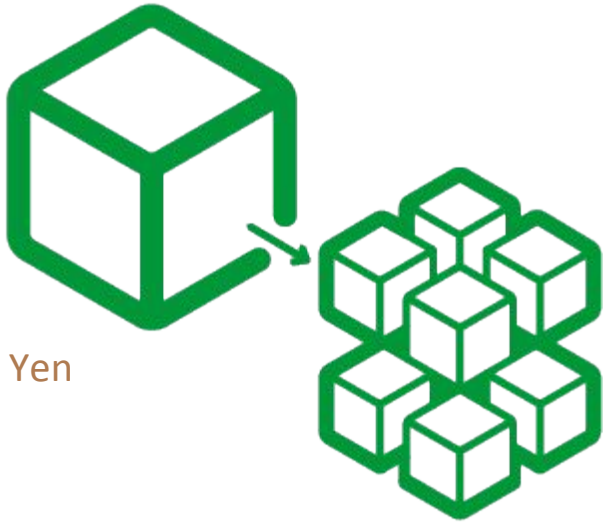
Microservices



CS249

Team #2

Rashmeet Khanuja, Anusha Vijay, Steven Yen

Problems with monolith and SOA that microservices address

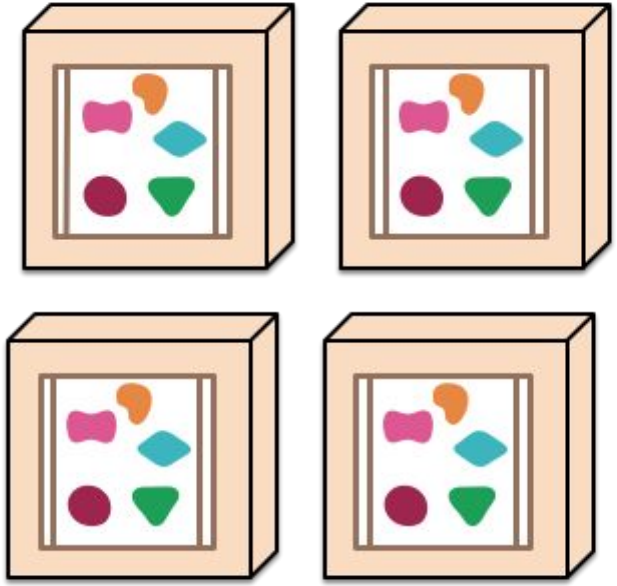
- **Context Mapping -**
  - Breaking monoliths into atomic microservices, teams simplify the problem space, work independently, and accelerate execution. For example, in a trading system, a bounded domain context may represent account, security, watch list, order book, or trade
- **Loosely Coupled, High Cohesion -**
  - With tight coupling as in monolithic systems, any business change requires making changes to the whole application
  - While SOA focuses on loosely coupling service consumers and providers from a technical perspective, a microservices approach focuses on loosely coupling services from a business domain perspective.
  - While SOA does not prescribe service capabilities, a microservices approach guides service interfaces towards atomic business capabilities.
- **Shared nothing architecture -**
  - Atomic microservices stand-alone, function independently, and don’t contain cross-service dependencies
  - Do not share data, process, or rules, they can operate in parallel, and a microservice failure will not impact others
- **Scalability -**
  - Demanding services can be deployed in multiple servers to enhance performance and keep away from other services so that they don’t impact other services

Problems with monolith and SOA that microservices address

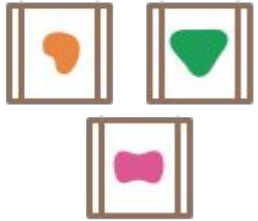
- **Full Stack, Dynamic Deployment-**
  - Each microservice encapsulates its own database, process server, application server, and integration server.
  - Unlike monolith, services do not share a single database
- **Parallel non blocking development -**
  - By reducing project dependencies and developing without coordination checkpoints, teams can rapidly evolve their business capability
- **Code maintainability -**
  - Code organized around business capabilities
  - Code for different services can be written in different languages without being impacted (helps in decentralized governance)
- **Efficient use of Resources -**
  - By giving business-critical microservices a greater share of the resources.
  - Resource needs can be prioritized by categorizing various microservices within the ecosystem according to their importance and value to the overall business

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

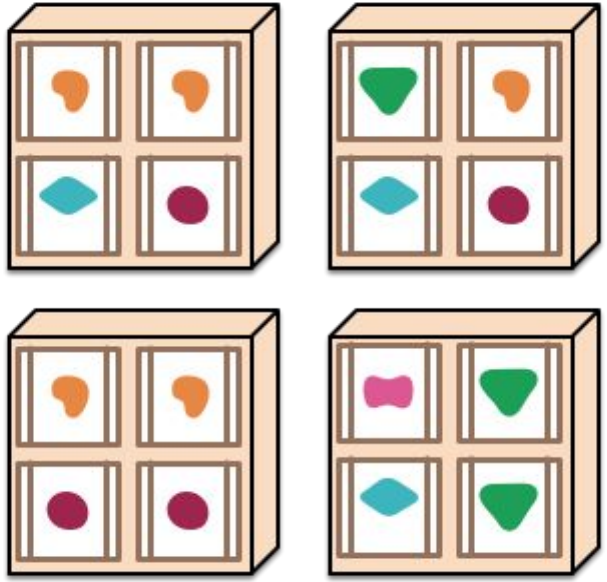
... and scales by replicating the monolith on multiple servers



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



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



Communication methods in microservices

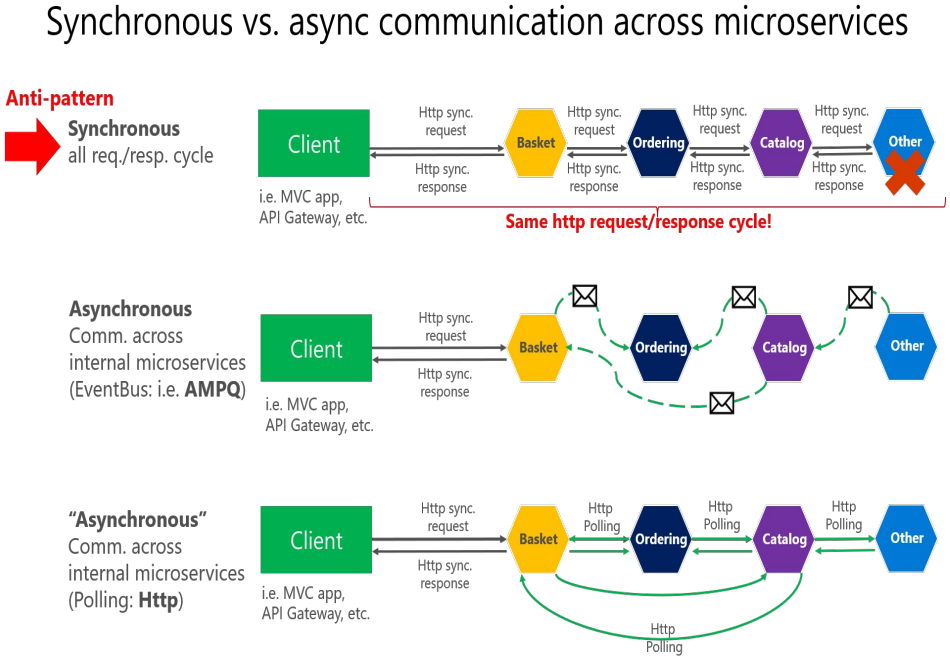
- Can be divided in two axis -
  - First axis defines if the protocol is synchronous (eg. - HTTP) or asynchronous (eg. - AMQP)
  - Second axis defines if the communication has a single receiver or multiple receivers

	One-to-One	**One-to-Many**
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Synchronous and Asynchronous

- **Synchronous model:** the *calling service* makes a request to another service, then the *calling service* would wait (or block) until the other service responds before continuing.

- **Asynchronous model:** the *calling service* makes a request to another service, then the *calling service* moves on to perform other work, without blocking or waiting for the other service to respond. The *calling service* has a separate thread that would listen for responses and perform some actions when it receives a response.



Asynchronous use-cases

- **Event Firehose use case:**
  - Many-to-many relationship between event/message producers and consumers.
  - Common in social networks where many events are generated by different users, and the received messages need to be delivered to many users' feeds.
  - Apache Kafka is an open-source platform suitable for this use case.
- **Asynchronous Command calls:**
  - Microservices exchanges messages with a need that delivery is guaranteed.
  - Messages are point-to-point.
  - RabbitMQ messaging platform (message broker) is suitable for this use-case.
- **Data Events Exchange:**
  - Event driver model
  - Microservices subscribe to data lifecycle events, and are notified when there's a data change operation such as update, delete, modification.

Synchronous use-cases

The asynchronous model is preferred with Microservices, however, there are situations when the synchronous model is more suitable:

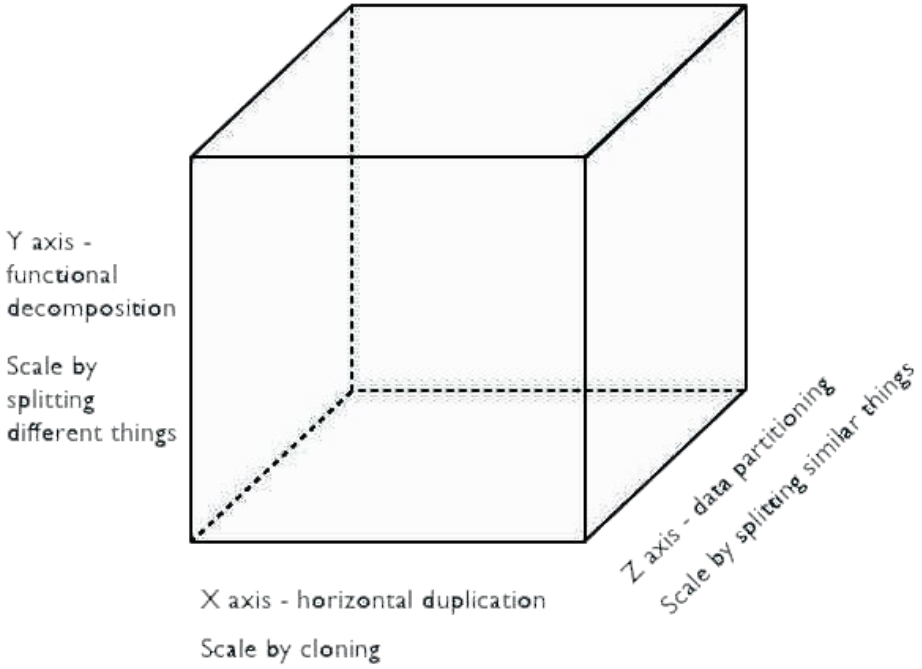
- When requesting information from databases, there’s a real need for a microservice to block or wait until another one responds before proceeding.
- When a user’s login details need to be authenticated, the system needs to follow a request/response pattern
- While booking airline tickets, the travel agent service waits for the user to select various attributes before displaying the relevant search results. Also at the payment step, the payment service waits for card authentication service

Single and Multiple Receivers

- **Single receiver -** Each request must be processed by exactly one receiver or service. An example of this communication is the Command pattern.
  - Type of one-to-one interactions -
    - Request/Response
    - Notification (a.k.a one-way request)
    - Request/Async Response
- **Multiple receivers** - Each request can be processed by zero to multiple receivers. This type of communication must be asynchronous.
  - Type of one-to-many interactions -
    - Publish/Subscribe
    - Publish/Async Responses
  - Example : publish/subscribe mechanism used in patterns like event-driven architecture. This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events; it is usually implemented through a service bus or similar artifact like Azure Service Bus by using topics and subscriptions.

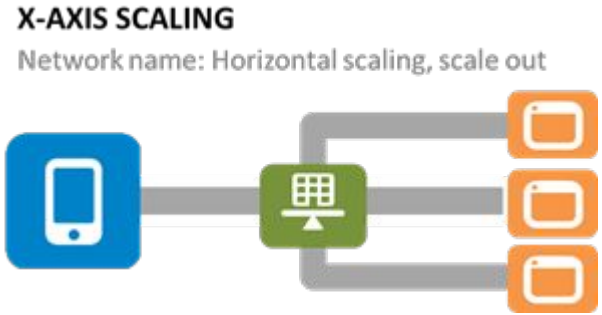
Scaling techniques in microservices architecture

- X-axis scaling
- Y-axis scaling
- Z-axis scaling



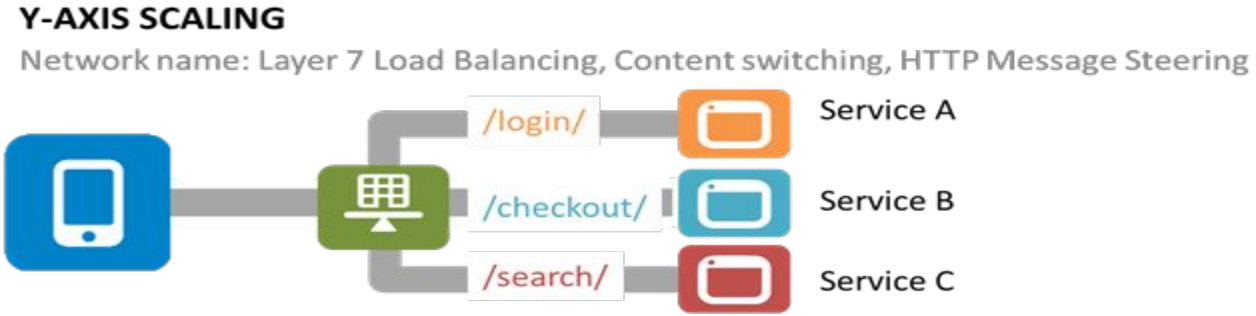
X-Axis Scaling (Horizontal Scale-out)

- Running multiple copies of an application behind a load balancer
  - If there are N copies then each copy handles 1/N of the load
- Can easily be implemented using layer 4 (TCP) load balancing if state is not important, but more often than not requires layer 7 (HTTP) load balancing due to the need to examine headers or other variables to ensure persistence to the right application instance (think sticky sessions).
- Drawbacks -
  - Since each copy potentially accesses all of the data, caches require more memory to be effective.
  - Another problem with this approach is that it does not tackle the problems of increasing development and application complexity.



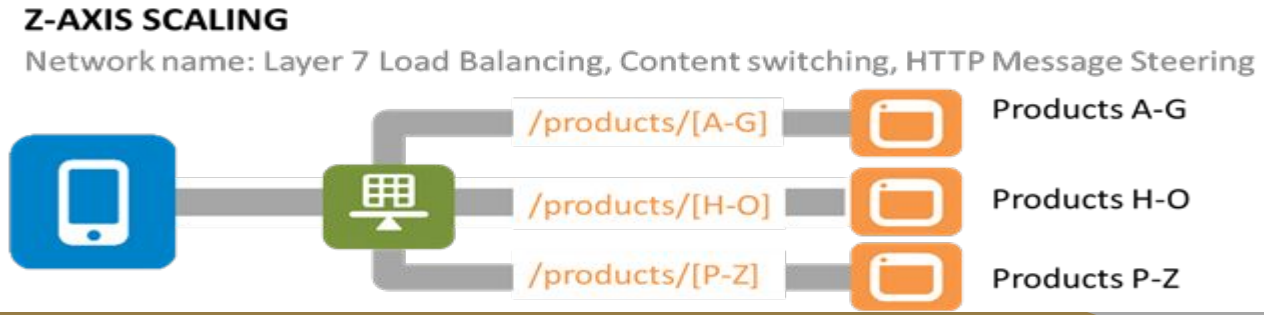
Y-Axis Scaling (Sharding)

- Instead of running multiple, identical copies of the application, Y-axis axis scaling splits the application into multiple, different services.
- Multiple ways to decompose applications into services
- One of the ways is verb-based decomposition and define services that implement a single use case such as checkout.
- Another way is decompose noun and create services responsible for all operations related to a particular entity such as customer management
- An application might use a combination of verb-based and noun-based approaches



Z-Axis Scaling

- Each server runs an identical copy of the code - similar to X-axis scaling
  - Difference is that each server is responsible for only a subset of the data.
- Some component of the system is responsible for routing each request to the appropriate server.
- Routing criteria can be -
  - An attribute of the request such as the primary key of the entity being accessed.
  - Customer type. For example, an application might provide paying customers with a higher SLA than free customers by routing their requests to a different set of servers with more capacity.
- Useful for premium access policies, where certain users or customers are afforded higher performance guarantees
- Commonly used to scale databases
  - Data is partitioned (a.k.a. sharded) across a set of servers based on an attribute of each record



Thank You!